Effective GPU-based synthesis and editing of realistic heightfields

Giliam J.P. de Carpentier

giliam@decarpentier.nl

Dissertation

Submitted in partial fulfillment of the requirements of the degree of

Master of Science in Media and Knowledge Engineering

July 7, 2008

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

Abstract

Computer games are expected to get evermore richer and detailed with each new generation of game consoles. Development of content for these game worlds is increasingly time consuming and, thus, would benefit from more effective tools. However, game level designers still work in almost the same way as five or ten years ago, with tools that are typically crude and inflexible. Therefore, investigating ways to improve the workflow and technology available to a level designer is both very necessary and promising. This dissertation investigates techniques to support the creative process of terrain design, which would improve both speed and quality. This research is limited to the most common type of geometry used for large open terrains, called heightfields, defined by horizontal regularly-spaced grids of height samples. To improve results, the applicability of several ideas from other design applications is discussed and several new extensions to existing procedural terrain generation algorithms are given. A comprehensive presentation of underlying theories and related work in these fields is provided. To improve the speed of terrain tools, ideas are explored that use today's powerful graphics cards not only to render 3D terrain, but also to perform actual terrain manipulations. These ideas have been incorporated into a custom testbed editor, together with a number of complex tools that use this new pipeline to their advantage. Details and results of this pipeline and the implemented tools are given. By executing the terrain manipulating algorithms on the graphics card instead of on the CPU, a noticeable speedup is achieved in practice, allowing for better interactive editing. To this purpose, an existing terrain rendering technique has been implemented in the testbed and further optimized for the specific demands of terrain editing. It is concluded that the proposed combination of common and novel heightfield editing techniques represents a valuable step towards overcoming the limitations of current terrain editing applications, by simultaneously improving quality, speed and user control.

KEYWORDS AND PHRASES: computer graphics, heightfields, terrain modeling, procedural modeling, fractals, erosion, terrain rendering, GPGPU.

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 large virtual environments at interactive or even real-time speeds. Consequently, applications like Virtual Reality training simulations and 3D computer games have become feasible.

Figure 1-1 **Screenshot from Wolfenstein 3D (id Software, 1992)**

Figure 1-2 **Screenshot from Gears of War (Microsoft Game Studios, 2006)**

With the ever increasing processing power of computers, it is possible to show more and more complex virtual worlds to the user 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. 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].

This dissertation focuses on one aspect of designing content for virtual environments: outdoor terrain. More specifically, it explores and examines common and novel ideas and techniques that would be helpful in the process of designing outdoor terrain for 3D computer games and offers several insights into, details of and solutions to the problems faced.

1.1 Applications

Realistically modeled terrain is part of all games that exhibit 3D outdoor areas. These terrain models typically are both functional and esthetic. Examples of functional terrain in games are natural barriers that balance gameplay by tweaking distances between key points on a map, and hinting at or enforcing an intended path by specific soil types and undergrowth, mountains, gores, etc. Examples of more esthetic choices are the exact shapes of rock formations, the placing of plants and trees and use of color. A great deal of effort is required to create terrain that is functional, pleasing, realistic and complex at the same time. Having tools, as described in this dissertation, to create terrains more easily empowers level designers to create, experiment and tweak both gameplay and esthetics in less time.

The work discussed in this dissertation is primarily targeted at supporting typical users of terrain tools for games, namely experienced and creative level designers. Therefore, some knowledge of and experience in both 2D and 3D tools like procedural tools, level editors and image editing tools (e.g. Adobe Photoshop) is assumed. This allows common controls, ideas, common workflows and user interface metaphors to be further built upon.

During research, user experimentation led to the discovery of an unintended but valuable secondary application. A system that speeds up the process of creating specific types of terrain can not only support game level designers during the production phase, it can also aid in the creation of concept art in the earlier phases of game development. For example, a screenshot taken from a selected point on or above the terrain might form the basis for a concept art drawing, which can then be edited and augmented with more traditional 2D image tools. Whenever this creation and rendering of 3D terrains can be done in less time than drawing these terrains in 2D by hand, productivity is increased.

1.2 Problem Statement

Current tools that are specifically available for the creation of outdoor terrain are typically found to be crude and inflexible. This dissertation explores and evaluates techniques to aid in the need for efficient and realistic modeling of outdoor terrain by automating and speeding up tedious and difficult work as much as possible, and increase the creative potential of its users by offering a range of different techniques.

Commonly available, lesser-known and novel techniques for the modeling of terrains for use in computer games are described and evaluated in this dissertation for their use in interactive design. 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 improve the workflow of a terrain editor's user.

As terrain models can consist of millions of bytes of data, it is important to offer techniques that are memory efficient. Also, it is important to be time efficient, as editing is typically an iterative process of evaluation and tweaking, thus requiring tools that work at interactive or even real-time speeds to minimize overhead during design. As modern PCs are able to execute more and more code in parallel (e.g. multi-core CPUs and many-core GPUs), trying to run terrain editing operations in parallel is important to get the most out of the hardware. This will be one of the focal points of this dissertation.

Level designers are assumed to use a modern PC with an above-average, reasonably cutting-edge processor and graphics card, as this affects the flexibility and speed of the hardware. No specific hardware vendor is assumed for this research.

The research has been conducted independently from existing tools or game engines, although extending an existing tool would probably have resulted in a more integrated solution. However, as there was no one preferred tool or engine available during this research, a more general and independent approach has been chosen. As there was no code base to start with, an editor had to be developed as part of the research that would function as a testbed for this research, including a terrain renderer. As the render algorithm affects the internally used formats, memory requirements and preprocessing

time, different options from literature will be described in this dissertation as well, and the algorithm that has been implemented in the testbed will be explained in detail.

1.3 Research Questions

This dissertation is focused around exploring, suggesting and evaluating answers and solutions to the following central question:

- *What are the main bottlenecks in terrain design for current computer gaming applications and how can these bottlenecks be alleviated?*

During analysis of this question, the following additional, more specific research questions were found to be relevant:

- *How could the workflow be improved?*
- *What well-known and lesser-known tools and techniques have been or could be used as tools in an offered tool set?*
- *How could the speed of the sculpting tools be combined with the complexity and realism of more advanced, but slower and less controllable tools?*
- *How can the above be achieved best using (only) the hardware capabilities of today's PCs?*

Some of these questions will be covered by in-depth discussion of the problems and ideas to improve the situation. Other questions (namely the last two) are less easy to evaluate from theory as they are concerned with performance in practice and are therefore tried out in the custom testbed. As this proved to be rather involved, the body of some of the presented chapters will be quite technical.

1.4 Dissertation Overview

The dissertation is organized as follows. After the motivation of this work is explained in Chapter 2, both general and specific requirements for a good terrain editor are specified in Chapter 3. To test ideas and evaluate techniques, a stand-alone testbed editor was developed. Although this dissertation is mostly about techniques and algorithms, software design and implementation considerations are shortly presented in Chapter 4. To render terrain with this editor, render algorithms are discussed in Chapter 5. Furthermore, this chapter serves as a general introduction to graphics hardware programming. Chapter 6 discusses many algorithms that have been used for the purpose of terrain generation and editing in the past and explores ideas to improve and extend them. That chapter also serves as a basis for work further discussed in subsequent chapters. Chapter 7 starts with the discussion of ideas to make better use of current hardware to execute heightfield editing operations, leading to the description of the implemented pipeline, which uses the parallel power of the graphics processor. The pipeline is further discussed in Chapter 8, including details of several common and novel tools that have been implemented. Chapter 9 assesses the discussed work in relation to the research questions. In Chapter 10, opportunities for future work and the overall conclusions are presented.

2 Motivation

Game content creation has traditionally been a manual process in order to squeeze the most out of the possibilities of a hardware platform. This manual creation of content is not scaling well with the increasing technological possibilities and users' 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 the geometric models and shading details required for 3D characters and environments to make them look as good as possible. This has become one of the major expenses in all multi-million dollar game productions. Consequently, any improvement that somewhat alleviates this burden is welcomed.

In practice, the tools available to different disciplines of game content creation evolve at different rates. For example, level designers still work in almost the same way as five or ten years ago. Investigating ways to improve the workflow and technology available to a level designer is expected to be very fruitful and is the main motivation for this research. For the evaluation of different techniques, a testbed was created to try out different techniques and tools, which can be used by the intended users. Limiting the scope of this research to heightfields had the advantage of being able to explore different editing techniques for a class of geometry specification that is supported by practically all outdoor games. This makes it possible to create a generic, stand-alone system that can import and export data from and to multiple existing game editors without changing any code. However, it should be noted that integrating the described tools into an existing game editor would shorten the work cycle by eliminating the need to import/export data at the cost of creating a game (engine)-specific system. As this research has been conducted in association with the W!Games game studio (Amsterdam, the Netherlands), which had no need for integration for any specific game engine at the time of writing, the prototype was created as a stand-alone application. Also, developing a custom pipeline capable of executing terrain operations in parallel is potentially quicker to implement and test in an independent testbed than it would be to retrofit it into existing code. Creating this custom parallel pipeline has become one of the focus points of this dissertation. As current terrain editors that support large, complex modifications are generally quite slow, and terrain editing is preferably an interactive process, any gain in speed will be more than welcome.

2.1 Terrain Heightfields

Before discussing terrain rendering, generation and editing, a small overview is given of different classes of terrain geometry representation, each with its strength and weaknesses 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 are supported is often dictated by the graphics engine. Five types of geometry are distinguished and described below, followed by a discussion of why this dissertation has limited itself to heightfields. Also, see Table 2-1.

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

Table 2-1 **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 dissertation.

Voxels

Voxels are values in a regular 3D grid. Like tetrahedrons, voxels represent three-dimensional volumes. So, creating holes, overhangs and caves is relatively easy. However, the amount of local detail is limited by the (uniform) resolution of the regular

grid. Also, the same resolution is present (and takes up memory) where potentially 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-rasterization-based hardware accelerated graphics cards. Consequently, only few games actually use voxels. So, like tetrahedrons, voxels are not relevant enough to be treated in this dissertation.

Irregular Meshes

Irregular meshes, based on some irregular sampling, are surface based and have a flexible topology and can 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 thus much more computationally 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 more expensive collision detection when simpler tests succeed (e.g. bounding box tests).

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 or loops in the surface geometry. Also, most irregular mesh algorithms (e.g. procedural generation, editing, level-of-detail and collision detection algorithms) 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, defined by height samples at regular discrete spacing. The height samples could be triangulated both regularly and irregularly, depending on the requirements and hardware support, and many fast level-of-detail rendering schemes have been devised and optimized for heightfields. Heightfields can be stored very compactly, because only data for the vertical axis needs to be stored, as the horizontal components are completely regular.

Because heightfields are discrete functions of 2D space, they can be stored, visualized and even edited as grayscale images. Represented as a 2D grayscale image, the greyvalue indicates the local height. Editing techniques for heightfields and digital images are therefore interchangeable. By convention, the maximum altitude is represented by 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, i.e. the process of mapping (colour) detail imagery onto the rendered geometry. A simple vertical orthographic projection of a detailed texture image onto the heightfield is generally sufficient. 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 noticeable. See Section 5.3 for more details. 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 of memory and render efficiency, heightfields are still the most common way to specify terrain for real-time 3D applications. Also, the simple data format makes it largely engine-independent, making it ideal to explore ideas that could be useful for many games. Consequently, this dissertation has limited its scope to the rendering and editing of heightfields. Literature on regular heightfields can be found that is either based on quadrilaterals, triangles or hexagons. For example, see [DIXO94] for procedural terrain generation techniques for different topologies. However, most literature assumes a quadrilateral structure and, moreover, almost all applications use regular quadrilaterals, often simply called quads. For this reason, heightfields mentioned in this dissertation are assumed to be based on regular quadrilaterals, unless explicitly stated otherwise.

2.2 Image Processing

As described before, the targeted users are experienced game level designers. These users typically already have practical knowledge of and experience with both 2D and 3D applications. Exploiting this experience by offering interfaces and capabilities similar to software that users are familiar with will shorten the learning curve. To this end, several interface controls and tools have been examined from existing tools such as powerful image processing applications (e.g. Adobe Photoshop) in an effort to compile a somewhat analogous toolset in the context of heightfield editing. These include the use of brushes and layers.

From a technological point of view, the internal representation and processing of heightfields is more similar to images than to 3D geometry. Data structures for both heightfields and images consist of a two-dimensional matrix of values, only differing in the meaning of these values; individual height samples in the context of heightfields versus color or grayscale components in the context of imagery. This allows heightfields to be presented as 1-component (grayscale) images, typically using black and white as the minimum and maximum of a user-defined height range, respectively. Image operations can therefore be executing on heightfields and vice versa. Conversely, ideas, theory and

techniques from the field of image processing can also be used in the context of terrain. Of course, not all image processing techniques will prove to be valuable in this new context, but many of them have been explored in previous applications and will be found to be fruitful throughout this dissertation.

2.3 Current Applications

In this section, a few different applications that are currently available to designers are shortly reviewed for their support in the area of heightfield editing. 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 techniques include noise synthesis, range mapping and erosion, which are 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 5.3.1.5. 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. Some 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. the use of interactive brushes) 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 at the time of writing. 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 connect 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 the Farcry and Crysis games. These offer an impressive set of tools to aid the level designer. They are capable of loading stored heightfields and simple procedural terrain generation. Local editing is supported through the use of brushes. However, only the simple brushes

discussed in Section 6.1 are available. Extensive terrain texturing is supported, similar to the layered representation discussed in Section 5.3.1.5, 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-to-use 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 6.1. Heightfield blending is not supported at all. Texture 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. Basic procedural foliage placement is supported, but is not as advanced as the tools available in the CryENGINE Sandbox. 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 and local terrain editing tools, while offering little or no support for procedural techniques.

- Procedural landscape generators, capable of synthesizing and previewing terrain to render new images (e.g. for movie productions) or export resulting heightfields for further use in other applications. These applications generally generate terrains as a whole and offer no tools to edit terrain locally.

3 System Requirements

Ideally, designing game levels is an iterative process. Often, designers create a fairly detailed level which is then compiled and evaluated. See Figure 3-1. Note that the compile step should be as automated and as fast as possible. 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.

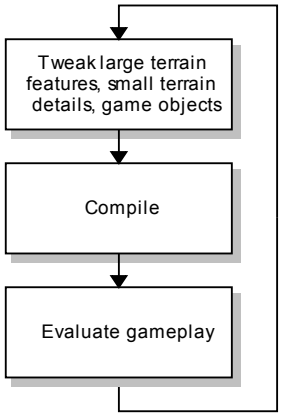


Figure 3-1 **An ideal level design workflow**

However, most applications available to level designers typically are designed solely around the idea of working from large to small. See Figure 3-2. 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 desire. Another way is to have an application generate a random terrain algorithmically (i.e. procedurally). Techniques used to do this are discussed in Section 6.3. 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 generated, followed by small-scale local editing. The disadvantage of the typical workflow using these tools is that, once an approximation for the whole level has been chosen, only lower-level manual editing is possible. This means that only a workflow from left to right in Figure 3-2 is supported, making iterative design of both large and small features, as suggested in Figure 3-1, 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.

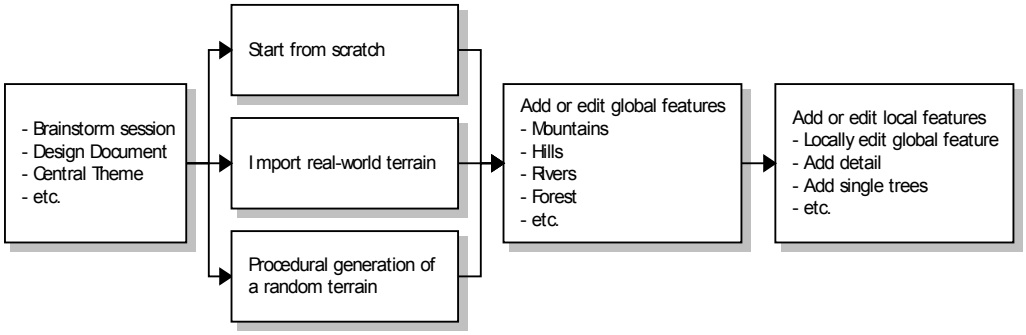


Figure 3-2 **Typical workflow supported by current applications**

3.1 Time Considerations

Even complex tools and parameters become usable to non-technical designers 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 computationally 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 that 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 Graphics Programming Unit, or GPU for short. 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 vector processing units. Many algorithms that can be implemented for partial or full parallel execution can potentially be more than one magnitude faster when work is transferred from the CPU to the GPU. Because terrain manipulation is very data-intensive and very regular, executing these manipulations on the GPU is very likely to increase the performance. Having fast tools increases the user's efficiency, which is why this dissertation is partly dedicated to setting up a pipeline for this type of parallelization (Chapter 7) and testing several translated manipulation algorithms (Chapter 8).

3.2 Memory Considerations

As explained in Section 2.1, heightfields can be stored relatively memory efficient, as only the vertical component of the height samples needs to be stored due to its regular sampling. Still, large outdoor areas can consume many millions of bytes. For example, a terrain of 4 by 4 km using a sampling density of one sample per m^2 and 16 bits of height data per sample results in 32 MB of data. This doesn't sound like much for modern day machines, but when an editor would keep multiple versions (e.g. for undo functionality), and overlapping sections (e.g. layers) of the terrain and multiple masks and texturing data in memory, it could easily take up all available main memory during editing unless special case is taken. Moreover, naïve rendering and heightfield modifications on the GPU would require the heightfield(s) to reside completely in video memory, which is typically only a fraction of the amount of system memory. Consequently, both rendering and editing algorithms and capabilities must be evaluated for their memory usage.

3.3 Platform Considerations

The general ideas discussed in this dissertation are largely platform and hardware independent but are assumed to be used on modern PCs. This is a reasonable assumption, as the vast majority of computers that are used during both console and PC game development are reasonably fast PCs. No vendor-specific capabilities are assumed. Also, the latest advancements in hardware standards are not assumed to be available on all targeted user computers. Consequently, a minimum of DirectX Shader Model 3.0 or

OpenGL with similar capabilities is targeted throughout this dissertation. This would cover practically all PCs new enough to run modern games.

The testbed application was developed largely on an 2 GHz Intel Core 2 PC with 2 GB RAM and an NVIDIA 7900 Go GTX with 512 MB video memory, using Microsoft Windows XP and DirectX 9.0c. The testbed has been successfully tried on other computers for testing and debugging purposes as well. These setups included both older and newer graphics cards (e.g. NVIDIA Geforce 6600 GT 256 MB, NVIDIA 8800 GTS 512 MB and ATI X1600 256 MB), but all with at least Shader Model 3.0 support.

The implementation builds upon multi-platform open source code. Although the testbed was developed primarily for a Windows/DirectX setup, its multi-platform components would allow for a relatively easy migration to other platforms. For example, the testbed has also been tested using OpenGL instead of DirectX, which was achieved by changing only a few lines of code. The testbed has been implemented as a stand-alone application as there was no need to integrate the technology with a specific game at the time of writing. However, the modular design and flexibility of the testbed would simplify conversion and integration with an existing engine.

3.4 Usability

Although higher-level tools that might be offered in a terrain editing application are typically 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 both technical and 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. What follows are some examples that can be used as guidelines for UI design:

- 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 parameters, and novice users might get overwhelmed by the possibilities.
- Parameters should have a descriptive name of their effect, which is not per se a term used in scientific literature, for example.
- The effect of different parameters should be as independent as possible. Presenting multiple parameters that only have slightly different effects should be avoided 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.

Another way to make complex tools more accessible to non-technical users is to offer presets. These presets can be used to store and retrieve particular settings of a tool and can be named after their effect. By allowing these presets to be shared between users, relatively few users would have to create presets, that can then be imported and used by other users. This might also create a more consistent look between parts of a world created by different users.

3.5 Toolset Considerations

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.

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. When multiple undo actions are allowed, the user can 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 in order to render the combined output image. Combining a layer with the layers below is done using a user-selectable combine operation per layer while 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 recursively recalculate dependent 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 considerably increases the memory footprint 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. Experimentation with the node-based World Machine showed that gigabytes of data needed to be processed for more useful (and complex) graphs. As this would be prohibitive for the purpose of interactive terrain editing, the layered representation is preferred.

Even though procedurally generated and placed geometry, texturing and foliage might look nice at a first glance, level designers easily spot the limitations of most current procedural 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 or even a level. Having a plethora of different techniques to choose from enables the designer to pick the right tool for the job at hand.

The quality of any terrain tool is difficult to measure quantitatively. If all processes involved in the creation of a certain type of landscape are fairly well understood, it would be possible to create a model of these physical effects and run a simulation. Although this will result in the 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 the typical user of a terrain editor for games, 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.

4 Software Development

As evident from the chosen problem domain, research questions and the many overviews of different options and techniques throughout this dissertation, this report covers a lot of material. Consequently, it would not have been feasible to create an all-round software solution containing all mentioned ideas and techniques to solve the difficulties at hand, given the time usually taken to complete an MSc. research project. Therefore, only relatively novel techniques have been implemented, which would be hard to discuss without having an implementation. These techniques include a parallel processing pipeline capable of making efficient use of resources and several complex tools.

As the implementation was not expected (or desired) to be integrated with an existing editor or game engine, a stand-alone application has been developed instead. This application has the purpose of trying out and evaluating common and novel ideas, and was set up to serve as a potential starting point for the creation a fully developed terrain editing application in the future.

A classic waterfall design process with incremental phases has taken place to create this application and roughly consisted of the following steps:

- Analysis of existing editing technology, literature techniques and user requirements. This was already partly conducted as part of the MKE literature research phase. This is mainly covered in Chapters 1 - 3

- Analysis of the expectations and requirements of W!Games. This is shortly covered in Chapter 2

- Analysis of multi-threading technologies. See Chapter 7

- Design and implementation of basis framework using carefully selected software libraries

- Analysis and comparison of several terrain rendering algorithms, design and implementation of the most suited algorithm. This is covered in detail in Chapter 6

- Design and implementation of basic brush system, universal settings GUI and import/export functionality

- Design and implementation of several straightforward and single-threaded CPU brushes

- In-depth analysis of graphics processor (GPU) programming capabilities. See Chapter 7

- Incremental and experimental design and implementation of GPU editing pipeline. See Chapter 7

- Optimization of the GPU editing pipeline implementation. See Chapter 7

- Design and implementation of complex, controllable and fast brushes using this pipeline. This is covered in Chapter 8

As the implementation would have an experimental character during this research, but might be used as a basis to create a full fledged application or be integrated with other applications in the future, a good and flexible software design was needed. Furthermore, attention was given to selecting only software libraries that would be free of charge for both commercial and non-commerical use.

The implementation has completely been written in the object-oriented C++ language, as this language is capable of creating fast and efficient applications, offers much control over hardware resources and is typically the main programming language used in game development studios, including W!Games. Microsoft Visual Studio 2005 has been used as development environment.

For rendering 3D graphics, the free and open-source graphics engine Ogre3D was chosen [OGRE3D]. This engine offers a clean abstraction of all DirectX and/or OpenGL related interfaces, is fast, would be flexible enough for the intended purposes and is constantly improved and extended by a large and active community. It is licensed under the LGPL license, meaning that it can be used free of charge without further obligations, as long as the engine itself is not modified. Furthermore, its graphics pipeline supports the use of complex shaders and can be run in both OpenGL and DirectX mode. Also, it is largely platform independent as it runs on Windows, Linux and MacOS systems, increasing portability. Note that Ogre3D is a graphics engine and not a game engine, as it does not offer native support for input devices, actors, game logic, audio, physics or an editor. To get low-level access to input devices like mouse, keyboard and tablet, the light-weight OIS [OIS] and Wintab [WINTAB] libraries have been used.

To create a typical (editor) user interface, the open source library wxWidgets has been used [WXWIDG]. This free library offers easy creation of windows, buttons and dialogs and such. Like Ogre3D, it is licensed under the LGPL license and is multi-platform. To accommodate any future integration of the implemented testbed with another application as best as possible, all GUI code has been separated from the rendering and editing code in the software design.

As the problem statement and research questions demand the in-depth exploration of techniques, theories and solutions, this dissertation will focus more on these topics than the exact software design details of the testbed implementation. Moreover, a detailed description and explanation of all individual classes in the implementation would be quite extensive and would require in-depth knowledge of the used libraries, technology and language paradigms like C++ meta programming. Consequently, the only C++ specific details of the design and implementation can be found in Appendix A and B in the form of UML class diagrams. This does not mean that this dissertation will leave out all design and implementation details, but these will be treated throughout the dissertation on a more functional level, interleaved with theory and explanations. For example, the next section includes specifics of the used rendering algorithm and its implementation.

5 Rendering

Although the main focus of this dissertation lies on terrain editing, different real-time terrain rendering algorithms are described in this chapter. The terrain renderer influences the speed of terrain updates and might or might not be optimized for memory footprint. Therefore, choosing the right renderer, when not already dictated by a used engine, can optimize the terrain editing speed by requiring less preprocessing and less memory cache swapping, for example. Section 5.1 shortly described the basics of hardware accelerated rendering on contemporary PCs. This will form a basis for the remaining sections in this chapter, but also serves as an introduction to the foundation on which the hardware-accelerated editing algorithms are built that are described in Chapter 7 and 8. Although Section 5.2 and 5.3 might be interesting to readers that are looking for suggestions and details to render terrain efficiently in real-time application and they are referred to a few times from the remainder of this dissertation, they are somewhat less relevant to the central research questions in this dissertation. However, they are included here for completeness, as the render implementation can still influence the terrain update and render speed during editing and formed an integral part of the development of the implemented testbed.

After the first section, a survey of different techniques for fast terrain rendering is given in Section 5.2. This survey is followed by some of the details of the testbed implementation in Section 5.2.2. In Section 5.3, different techniques to texture the terrain geometry are discussed, including specifics of the testbed implementation.

As explained before, this dissertation is limited to editing heightfields for games. After creation, these heightfields can then be used to render in real-time using the actual game engine. But, obviously, a heightfield also needs to be rendered during interactive rendering in an (external) editor. Many algorithms exist to render heightfields, each with different considerations, advantages and drawbacks. Differences exist in whether terrain patches have continuous or discrete levels of detail (also called LOD levels), the way different LOD levels morph and stitch, the LOD level update frequency, LOD pixel error metrics, the use of hardware-accelerated operations and data structures, texturing techniques and more.

Algorithms that are commonly used to render terrain (or any type of 3D objects, for that matter) can roughly be divided into two categories: ray tracing and rasterization. Ray tracing is sometimes used in off-line movie productions as certain effects are more easily defined in ray tracing algorithms (e.g. spline surfaces, secondary ray lighting effects and ambient occlusion). But almost all graphics on PCs for real-time purposes use rasterization, as this is far better supported and accelerated with today's graphic cards. For this reason, subsequent subsections only focus on the rasterization pipeline in today's computers and its applications in terrain rendering.

5.1 Graphics Pipeline

Today's graphics hardware found in most PCs (and newer consoles) is a hardware subsystem intended to process graphics data and present it to the user. This subsystem is capable of working along side the central processing unit (i.e. the CPU) and has its own dedicated processor. This Graphics Processing Unit (i.e. the GPU) transforms, lights and rasterizes 3D geometry to calculate the color values for individual pixels on a screen. For this purpose, the GPU is typically capable of executing vector operations and processes instructions in parallel using one or more cores in parallel.

The graphics data to be processed is generally stored in dedicated video memory, which has a higher bandwidth than ordinary system/main memory to accommodate fast data reading and writing. Older systems sometimes do not have dedicated video memory and use (shared) system memory instead. System memory that contains scene data can be copied to video memory using a specialized hardware data bus. The AGP standard has been used for this purpose for many years, but a shift to the faster PCI-Express bus is currently taking place.

To use the GPU on a PC, a software driver is installed that implements a common device driver interface to allow transparent and unified access to the hardware-accelerated capabilities. Microsoft DirectX Direct3D and OpenGL are the best known and most widely used examples of these interfaces. In turn, applications can use the application-side of the DirectX or OpenGL library to drive the hardware. The Microsoft DirectX Direct3D is available on Microsoft Windows. The OpenGL API is available on Microsoft Windows, Linux, MacOS and many more platforms. Both can be accessed from

a wide range of programming languages. There are some differences between these APIs (e.g. amount of helper functions, use of vendor-specific advanced capabilities and the shader languages), but for the most part, the APIs offer the same set of functionality.

The testbed that has been created during this research builds upon the Ogre graphics engine, as already explained in Chapter 4. This engine wraps graphics API-specific calls, among other things, and offers a clean and largely graphics API-independent interface to the programmer. By only using multi-platform libraries (e.g. Ogre and wxWidgets), the created framework would be able to run on different platforms with little modification.

Over the years, the processing power and flexibility of the GPUs have been greatly improved. To make the most of the latest hardware capabilities, many (coexisting) revisions to the OpenGL and DirectX APIs have been made. Today, two different pipeline models are exposed by these libraries which can be used side by side: an older fixed-function pipeline and a newer programmable pipeline. The fixed function pipeline offers many lighting, clipping and texturing options that can be selected and combined, but does not nearly offer as much flexibility as the more general-purpose programmable pipeline. However, not all steps are programmable in the programmable model; only what is called a vertex shader and pixel shader (also called a fragment shader) can be programmed. These allow specialized programs to execute for each triangle vertex and rasterized triangle pixel, respectively. See Figure 5-1. In this figure, the pipeline components operate from left to right. Vertex, topology and texture data resides in video memory. Vertex processing transforms incoming vertices using either fixed functions or a vertex program. Then, the transformed vertices for each triangle are rasterized and all vertex data is interpolated over the triangle surfaces. Each pixel that is rasterized is then given a color using a fixed function or a pixel shader, possibly based on (multiple) texture reads and might undergo different tests (scissor test, alpha test, stencil test and depth test). Based on the pixel processing and test outputs, the calculated pixel color is blended with the previous pixel color and written to the framebuffer.

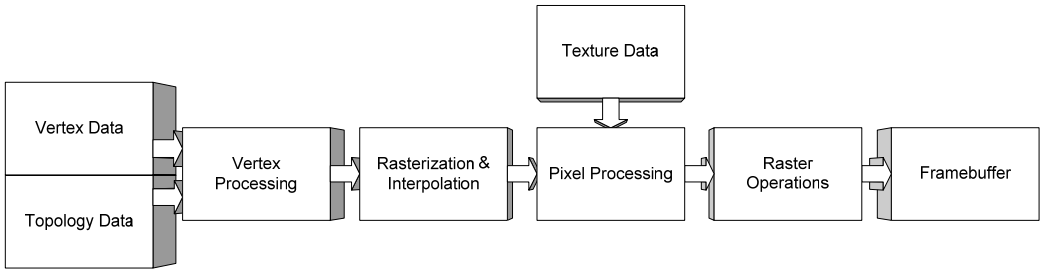


Figure 5-1 **Graphics pipeline**

A recent addition to the programmable pipeline is the geometry shader, which operates between the vertex shader and pixel shader and can transform one set of vertices into another set. One possible use of geometry shaders is hardware-accelerated triangle tessellation, which would certainly be interesting in the field of level-of-detail terrain rendering. However, this technology is relatively new (only supported under DirectX 10) and is still ill supported. As few implementations and papers exist on this topic and this dissertation focuses on today's modern, but widely available hardware, no further attention is paid to this technology. But as hardware is updated regularly, this would certainly be something to look at in the near future.

As the graphics pipeline is becoming more and more programmable with each generation of hardware, applications that use the processing power for purposes other than graphics are becoming feasible. Examples of this are audio processing, matrix manipulation, physics simulations and computationally-intensive scientific experiments. The reason for the interest for the GPU as a general-purpose parallel processor from other fields is simple: it has more processing power and memory bandwidth than CPUs of the same generation and they are relatively cheap and widely available. The gain in performance that can be achieved by moving work from the CPU to a GPU depends on the possibility to massively parallelize an algorithm, limiting its use to specific classes of algorithms. As a matrix of height samples (i.e. a heightfield) can directly be implemented as a texture and the same heightfield editing operation is typically executed for many neighboring height samples (i.e. pixels), heightfield editing is ideal to be implemented as GPU algorithms. This possibility is further explored in Chapter 7 and 8.

Vertex and pixel shaders are (short) programs that can be executed millions (or even billions) of times per second. Specialized languages exist to specify vertex and pixel shaders. As GPUs only accept assembler instructions, higher-level shader languages are

compiled before being uploaded to a GPU. Because the assembler instruction set can vary per hardware generation, these compilers generally can target and optimize for different hardware profiles. Examples of high-level shader languages are High Level Shading Language (HLSL), OpenGL Shading Language (GLSL), and C for Graphics (Cg). The first is only natively supported by Microsoft DirectX, the second only by OpenGL. The Cg language is capable of output both HLSL and GLSL code, making it compatible with both graphics APIs. For this reason, the Cg language was used during this research to create shader programs for both heightfield rendering and heightfield processing. For an in-depth discussion of the Cg language, see [FERN03]. A more thorough discussion of graphics programming languages is given in 7.3.1.

Different generations of programmable GPUs differ in the size of the supported assembler instruction set, the maximum program size, maximum (dependent) texture reads per pixel and maximum amount of variables and constants. This dissertation only targets Microsoft DirectX shader model 3.0 (PS 3.0 and VS 3.0), as this version is supported by most modern PCs. This model is the first to practically overcome limitations such as maximum code and texture reads, making it possible to write much more powerful and generic programs. Shader model 4.0 is the latest in this series but is currently much less supported and requires Microsoft DirectX 10 (which, in turn, requires Microsoft Windows Vista). Hardware that supports DirectX shader model 3.0 typically also supports OpenGL profiles that are equally powerful and compile from the same Cg program, making it possible to execute the shader programs with both graphics APIs.

5.2 Terrain Geometry

The data that makes up a heightfield forms a regular grid of height samples. To render a terrain from these height samples, a surface must be defined from the samples. Also, a material is applied to the surface to shade the triangles. See Figure 5-2.

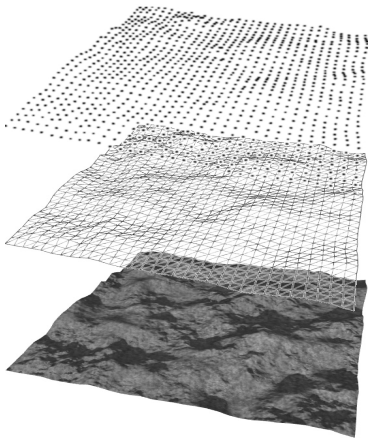
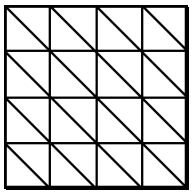
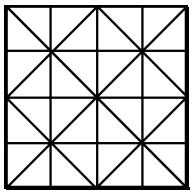


Figure 5-2 **Heightfield render steps. Top to bottom: Regular heightfield grid, surface triangulation, surface shading**

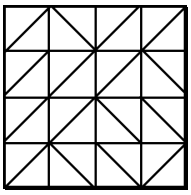
Although it is possible to create and render smooth spline surfaces based on these samples, the sample grid is normally directly triangulated, as triangles are much faster to render using today's graphics hardware. Three possible methods of subdividing the heightfield directly into triangles are depicted in Figure 5-3. Even though the vertices of all three examples are identical, differences between the triangulation would become visible for rough terrain; the left example would result in a smoother shape for a ridge that follows the diagonal triangle edges, but would show a saw-tooth-like pattern for ridges perpendicular to the diagonal triangle edges. In that respect, the example in the middle is more isotropic (i.e. rotation-independent). The example on the right allows the user to specify the orientation per triangle pair, requiring just one extra bit per height sample. Of course, more complex and irregular triangulations are also possible, skipping some height samples during triangulation based on different criteria.



Regular triangulation. Twice as much resolution (i.e. sample density) in one diagonal direction than the other diagonal direction.

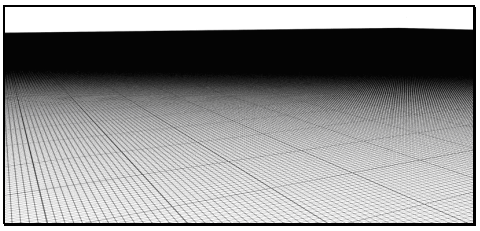


Regular checkerboard-like triangulation. Both diagonals have identical average resolution.

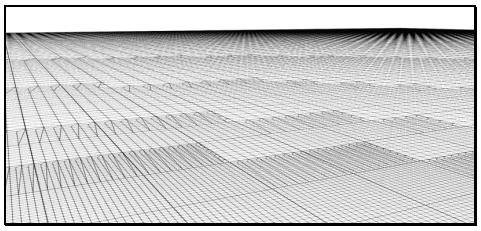


Irregular triangulation. Their diagonal can be selected per cell, gaining control over relative diagonal resolution.

Figure 5-3 **Heightfield triangulation**



Camera-independent mesh density. Note the distance-dependent vertex density in screen space.



Camera-dependent mesh density. Here, tiles (i.e. blocks) of geometry are subdivided at different world-space densities to achieve a roughly distance-independent screen-space vertex density

Figure 5-4 **Camera-(in)dependent heightfield tessellation**

5.2.1 LOD Techniques

A complete heightfield can be rendered using one of the simple schemes depicted in Figure 5-3. Then, however, terrain triangles that are viewed from afar might be smaller than a single pixel. Having sub-pixel triangles does not only cause spatial aliasing, it also wastes valuable bandwidth and processing power. For example, a 4K x 4K heightfield would consist of 32 million triangles. Rendering this many triangles would have a severe impact on the frame rate, even on today's hardware. For this reason, many different so-called level-of-detail (LOD) rendering algorithms have been devised that try to minimize the number of actually rendered triangles by merging triangles together that would be too small to matter when seen from a virtual camera. Consequently, the local mesh density (triangulation) of the terrain will depend on the camera parameters like position, rotation and field of view. See Figure 5-4. As camera movement is typically smooth, frame coherency can be exploited by updating the mesh incrementally, reusing as much

of the triangulation as possible between frames. LOD render algorithms typically are optimized to make use of the simple topology of heightfields and would not work on arbitrary 3D geometry. What follows is a survey of several LOD techniques that have been devised over the years. This survey is by no means complete, but does discuss a wide range of different (once) popular approaches. The algorithms are categorized into three classes: continuous, discrete and (semi-)fixed. Lastly, the algorithm that has been implemented in the testbed is discussed in Section 5.2.2 in more detail. This algorithm has been chosen based on the criteria discussed in Section 5.2.1.4.

5.2.1.1 Continuous LOD

Classic terrain level-of-detail rendering algorithms focus on merging triangles to an optimal solution, given some error metric. Advantages are the high quality of the triangulation and an exact and controllable error bound. However, the triangle mesh requires many (small) updates when the camera moves. The costs are usually amortized by spreading the execution of any queued updates over multiple frames. The overhead of updating the triangulation used to be a small price to pay when compared to less optimal (e.g. brute-force) rendering schemes, as the largest bottleneck in render performance was mainly the triangle count. These optimal algorithms are called continuous level of detail algorithms and were most popular before the year 2000. Although not actually used for this thesis, several well-known algorithms are mentioned below for the sake of context and completeness. Lindstrom et al. introduced a two-step quadtree-based algorithm that uses a screen space error metric and hierarchical blocks of triangles to decide on individual LOD levels, followed by fine-grained triangle simplification step [LIND96]. The error metric is based around measuring the change in height in screen-space that would occur when merging a triangle pair, as seen through the virtual camera.

Duchaineau et al. presented a method called Real-time Optimally Adapting Meshes (ROAM), that both splits and merges individual isosceles right triangles to new isosceles right triangles [DUCH97]. Separate split and merge priority queues are used to have control over the amount of time spent on updates per frame, while executing updates that have a larger effect of the error metric first. Besides a basic screen-space error metric, it also supports more advanced metrics accounting for back-face detail reduction, silhouette edges and frustum culling. Due to its fine update control, a guaranteed exact triangle

count can be achieved. Although this algorithm makes the most of coherency between frames, the overhead of updating these queues limits its performance gain. In [RÖTT98], a hierarchical depth-first quad-splitting algorithm is presented that is optimized for memory efficiency, but does not implement incremental updates of the triangulation once created, limiting its performance for large terrains.

As mesh updates can cause small sudden changes in appearance, called vertex popping, a technique called vertex morphing can be applied to most algorithms, which allows an animated transition of local geometry to a different level of detail [RÖTT98]. This is achieved by linearly blending between the actual finer triangulation and the coarser triangulation that is projected onto the finer triangulation geometry, until the transition is completed.

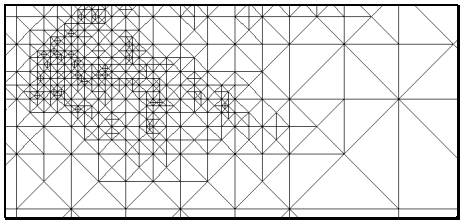


Figure 5-5 **Example of a ROAM terrain. From [DUCH97]**

5.2.1.2 Discrete LOD

Due to advancements in graphics hardware, today's graphics processors are capable of rendering many more triangles per second. More triangles also means more triangulation updates as the camera moves. For this reason, the relatively slow data bus that is used to transfer geometry data from the CPU and main memory to the graphics card's memory has become much more of a bottleneck. Furthermore, these updates require synchronization of the graphics processor, stalling today's GPUs as these have become much better optimized for cached, asynchronous rendering. Also, triangles are rendered most efficiently when they are batched (combined in one render call) in as few batches as possible. Hence, it is generally better for modern PCs to use algorithms that minimize the number of updates, even when this means rendering (slightly) more triangles. Several algorithms have been created with this in mind. Most of these employ a discrete, or static, level of detail technique that partitions the heightfield into equally-sized blocks, called tiles, and only applies updates at this tile level instead of at the triangle level. As a result, the CPU load and the amount of graphics hardware data updates are minimized. Furthermore, by combining triangles into regular tiles, triangles are much easier to batch

into one or a few batches per tile in order to optimize render calls. As a result of tiling, changes happen on a larger scale and are compacted into fewer updates, and so, the vertex popping artifact becomes much more noticeable. Vertex morphing can be applied for most of these discrete level-of-detail algorithms as well, greatly smoothing transitions between tile LOD changes.

One discrete LOD approach was introduced by Hoppe [HOPP98]. It produces several triangulated irregular networks (TINs) of different quality (i.e. LOD levels) for each square tile. The vertices at each tile's edges are kept unmodified, creating a seamless connection to neighboring tiles. See Figure 5-6. This TIN-based approached results in fewer triangles than more regular grid-based methods, but requires more preprocessing and is relatively memory intensive. Most common discrete LOD algorithms are based around semi-regular triangulations, differing only in their error metrics, data structures and ways to seamlessly connect neighboring tiles, as will be discussed next. An example of such a triangulation is depicted in Figure 5-7.

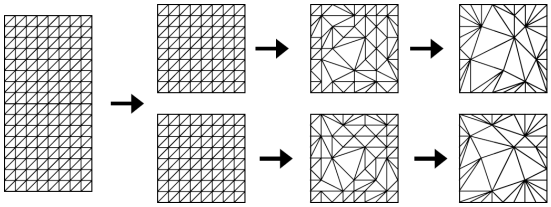


Figure 5-6 **Splitting terrain into tiles and creating TIN meshes of various quality levels for each tile. From [HOPP98]**

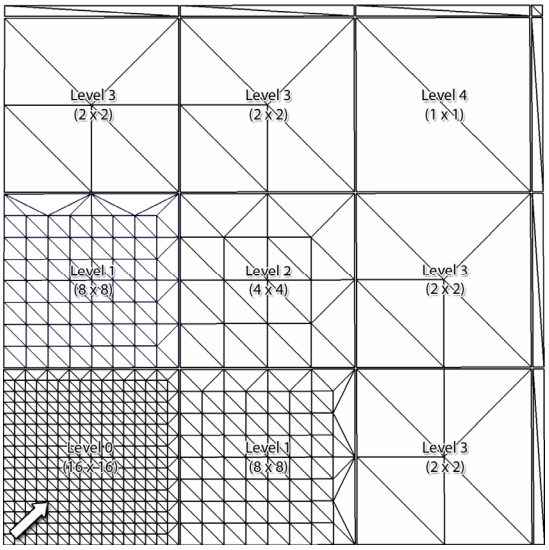


Figure 5-7 **Example of GeoMipMapped tiles using 16 x 16 grid cells per tile for a heightfield of 50 x 50 samples. The camera would be located at the bottom right corner**

Special attention needs to be paid to connect borders of neighboring tiles to prevent possible gaps due to different LOD resolutions. In [ULRI02], a downward vertical polygon 'skirt' of conservative height is connected to the perimeter of each tile to effectively hide these gaps. As these gaps would be relatively small when compared to the tiles, this skirt usually does not stand out if lighted and textures identical to the rest of the tiles.

Obviously, rendering these extra polygons increases triangle count and overdraw for low camera angles.

An algorithm called GeoMipMapping [BOER00] uses a different approach. To close potential gaps at the shared border of neighboring tiles that use a different LOD level, the topology of each the triangles near each border is modified in such a way that the triangles at the edges of a tile of a finer LOD level will match the tile borders' topology of each of its neighbors with a coarser LOD level. See Figure 5-7. A tile is rendered in a single batch by using an indexed triangle list. This means that the geometry is defined by two buffers: a vertex buffer and an index buffer. As the name implies, the vertex buffer is an ordered list of vertices. The index buffer defines triplets of vertex indices, each triplet defining a single triangle. Separating the geometry into these two buffers allows calculated per-vertex data (e.g. transformations) to be shared between neighboring triangles, improving render efficiency. However, it also has the advantage of allowing the index buffers to be shared between tiles of similar topology. The index lists are effectively independent of the actual height data as it only defines the topology or connectivity of a tile. By skipping some indices of vertices at a tile's edges in a vertex index list, the topology of its neighboring tiles can be matched effectively. This is explained in more detail in Section 5.2.2. GeoMipMapping uses one vertex buffer per tile per LOD level. In [SNOO01], this idea is taken one step further. All height samples are stored in a single vertex buffer per tile, used for all LOD levels. To render a tile at any LOD level, one of many precalculated index lists is used. Index lists of decreasing (i.e. coarsening) LOD level would skip (i.e. would not use) more and more vertices. To make a tile connect seamlessly to its neighboring tiles, a tile is defined by two separate index lists: the body list and the link list. The body list represents the major portion of a tile and draws all triangles for given LOD level, except triangles at the tile's edges that connect to neighboring tiles of a coarser LOD level. The link list fills the remaining area of a tile, interlocking seamlessly with both the tile's body and the neighboring tile of a coarser LOD level. Consequently, only one vertex list is present per tile, and a finite number of precalculated index lists are stored, of which two are chosen and rendered per tile. As less and less vertices are used for distant, coarser tiles, cache coherency would suffer during rendering. This drawback can be somewhat minimized by optimizing the vertex order for patterns of common use. Obviously, this algorithm requires more video memory

than GeoMipMapping, as vertex lists must always be present in video memory at the finest LOD level, even for distant tiles.

5.2.1.3 (Semi-)fixed LOD

Lastly, a class of algorithms has been devised that reuses a triangulated mesh and only update its vertex positions, but not its triangle layout (i.e. connectivity). Consequently, it offers a steady triangle count and rendering rate. In [DACH06a], the local mesh density is modified to decrease with density and increase with slope, creating a more uniform triangle size in screen space. This mesh is fixed to the camera, translating and rotating with it. Each vertex of this mesh is then warped on the CPU to approach the preferred local mesh density and then samples the heightfield at high resolution on the GPU to get the vertical component of each vertex. The heightfield sampling applies hardware bi-linear filtering, as the sampling positions are not necessarily aligned with the heightfield's virtual grid. This sampling filtering prevents serious popping, but does suffer from an artifact called 'swimming'. Higher order filters would reduce this effect greatly but are not natively implemented in graphics hardware and are expensive to emulate. This algorithm is quite video-memory unfriendly, as it requires the complete heightfield to be accessible from the graphics hardware.

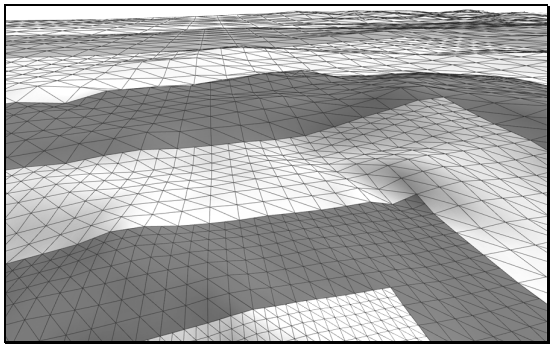


Figure 5-8 **Rectangular geometry rings. Each of the (differently shaded) farther rings halves its vertex density in world space and is formed from a different 'clipmap' pyramid level. From [ASIR05]**

In [LOSA04], a different approach to the camera-fixed mesh is taken. This algorithm uses a pyramid of different LOD levels of heightfield data near the camera's position. Each pyramid's subsequent layer halves its vertex density, but doubles the covered area. See Figure 5-8. To render the heightfield from the camera's position, an world-axis-aligned rectangular 'ring' of geometry is rendered per pyramid level, rendering only the area that is not covered by the other levels, centered around the camera. When the camera moves, the pyramid's data can be updated incrementally. A moving camera will

move a virtual center position inside each layer's vertex buffer, reusing most of its data and replacing only the exact data that moved out of the layer's covered area with new data that just moved in at the opposite side. This is possible by treating each level's data as a toroidal datastructure, wrapping around its edges, effectively making the areas that would move out and move in overlap. To hide vertex popping effects that would occur as heightfield sample points are rendered using different pyramid levels when the camera moves, the vertices near the far border of each ring blend to a coarser LOD level using a special vertex shader, similar to vertex morphing. Most of the processing is done on the CPU. This algorithm was improved in [ASIR05] by moving much of the work to the GPU and splitting the rings into smaller building blocks, improving block reuse and frustum culling. This algorithm, as well as the algorithm described in [DACH06a] relies on the availability of vertex textures, a relatively new hardware feature that is not yet completely standardized. Consequently, it would require different code paths for different vendors.

5.2.1.4 Selection criteria

To render heightfields for use in a terrain editor, a render algorithm must be chosen and implemented. A quantitative performance comparison of the algorithms discussed above would be difficult to present, as most papers use different hardware generations, viewport resolutions and heightfield sizes to measure the frame rate of their algorithms. Several observations can be made, however. As mentioned earlier, a shift is taking place from more complex CPU-based triangulation algorithms to slightly more brute-force methods that are optimized for today's GPU capabilities and strengths. This includes minimizing update size and frequency. To choose an algorithm to use as the basis for the renderer written as part of this research, the following criteria were used:

1. **Fast rendering**. Fast rendering is important as this increases the user interactivity and allows more computing power to be used for actual terrain editing. As explained earlier, efficient use of hardware-accelerated data structures is the key to performance on modern PCs.

2. **Simple data structures**. As a terrain editor would need to update a heightfield regularly, simpler data structures with a minimum of required preprocessing are preferred, as these are faster to update when the terrain is modified.

3. **Small memory footprint**. This thesis describes techniques that use video memory as a cache to process heightfield data other than the actual rendered 3D heightfield geometry, making algorithms that use less system and video memory more preferable. See Chapter 7 and 8 for further details.

4. **Regular access patterns**. The raw heightfield data is partitioned into smaller blocks, called pages, for reasons described in Chapter 7. Render algorithms that have less regular access patterns during geometry updates would be harder to optimize for paged source data access.

Discrete tile-based algorithms fit the fourth criterion best, as these produces very regular triangulation and would only use data from one block or a group of neighboring blocks per tile, as the tiles themselves form a similar partition into blocks. One of the discussed discrete tile-based algorithms, called the GeoMipMapping algorithm, was found to comply best with the combination of the other three criteria. This algorithm was chosen and further modified to the specific needs for this research.

5.2.2 Implementation Details

What follows are the specifics of the modified GeoMipMapping algorithm. For a more in-detail discussion of the original algorithms, the reader is referred to the original paper [BOER00].

As discussed in Section 5.2.1.2, this algorithm splits the terrain in regularly shaped tiles. This has several advantages. First, this allows the CPU to cull larger areas as a whole that are completely outside the camera's view frustum through the use of bounding boxes and frustum culling on a tile basis. This prevents large chucks of the heightfield from being transformed into screen space by the graphics hardware before the many triangles that make these tiles would be culled away individually anyway. Secondly, the regularity of the tile's triangulations is easily exploited by creating a single triangle strip

per tile for efficient rendering and by sharing the connectivity data structure (i.e. index buffer) between tiles for efficient memory use.

5.2.2.1 Tile LOD Selection

Chosen the best LOD level for each terrain tile is done using some form of heuristic or error measurement. The original paper GeoMipMapping paper [BOER00] uses a terrain roughness-dependent LOD error metric to choose each tile's LOD level during rendering. These LOD levels are represented by integer values ranging from level 0 (containing some W x W vertices) up to level $\lceil \log_2 W \rceil$-1 (containing only 2 x 2 vertices) for a tile containing W x W height samples. The original LOD error metric precalculates the minimum distance at which the maximum difference in screen space between vertices of a reduced LOD level and the finest LOD level reaches a specified pixel error, assuming the terrain is always viewed from aside. This metric is evaluated and stored in a lookup table during tile creation for each individual tile at each possible tile LOD level. As this requires calculating the maximum of all vertex errors per LOD level, this lookup table would need to be recalculated when the terrain is edited. As this thesis is concerned with editing terrain, all overhead by preprocessing terrain should be minimized as much as possible. For this reason, a simpler error metric has been chosen. Instead of assuming the terrain is always viewed from aside, as the original paper did, it is assumed it is always viewed from above. This actually might fit this thesis's usage pattern slightly better, as terrain editing is typically done from a bird's eye view instead of viewing the terrain from off the ground. Disregarding projective distortion, this metric is independent from the terrain roughness. Consequently, the LOD level for each tile is made to only depend on the camera's parameters and distance. To support rendering to multiple viewports, this formula can be evaluated once per viewport/camera pair, picking the finest LOD level found.

$$a = \tan(\frac{f}{h})$$

$$L = \frac{ad}{s}e$$

Here, f and h represent the camera's field of view and viewport's width in pixels, respectively. This makes a approximately the size of one world unit perpendicular to the camera per screen pixel per world unit in the direction of the camera (i.e. per pixel per

unit distance). s is the fixed world distance between neighboring height samples in the tile, or the grid cell size. d represents the distance between the camera position and the closest point on the axis-aligned box formed by the tile's position, size and full height range. Assuming the heightfield is viewed from above, ad/s is the (maximum) amount of grid cells per pixel. e is the maximum allowed screen 'error' in pixels. Ideally, exactly one vertex per block of e x e screen pixels would be rendered. L represents the number of height samples per e pixels. Hence, the coarsest LOD level that renders at least one vertex per L height samples is chosen. Each subsequent higher LOD level effectively halves the resolution of a GeoMipMapped tile. Obviously, some camera distances to a tile will result in a sudden change of LOD level. Slightly moving or rotating the camera near these distances can cause the tile to oscillate between LOD levels. To prevent this, a tile's active LOD level is only made to change when the ideal LOD level is outside the range $[k-\varepsilon, k+1)$ instead of $[k, k+1)$, k being the currently active integer level and ε being some small value (e.g. 0.2). Tile LOD updates are queued, updating only a limited number of tiles per frame to amortize large changes over multiple frames.

The camera-tile distance d can be calculated by $((P_{camera} - P_{tile}) \bullet (P_{camera} - P_{tile}))^t$ with $t = 0.5$. By (slightly) increasing t, a practical bias can be created that decreases the LOD level for distant tiles even further. This might be desirable when editing terrain from close by, preferring frame rate over more precise rendering of distant, less relevant terrain. Both e, ε and t can be made adjustable in real-time to accommodate for different user preferences, hardware and editing situations.

5.2.2.2 Tile Triangulation

A heightfield of M x M height samples will be split into tiles that cover N x N grid cells, requiring $(N+1)$ x $(N+1)$ height samples per tile. Note that the algorithm does not require the heightfield and tiles to be square, but this simplifies the notation presented in this section and would often be the case in practice anyway. If M is not divisible by N, the most right column and most bottom row of tiles that make up the heightfield only cover the remaining area. See Figure 5-7. Of course, each border of a rendered tile should connect to the opposite border of each neighboring tile. This requires the height samples at the borders of a tile to be used both by the tile itself and its neighboring tile. Described

in symbols, there are $\lceil M/N \rceil$ x $\lceil M/N \rceil$ tiles and tile (u, *v*) uses the height samples at $[N \cdot u, N \cdot v] - [\min(M, N \cdot (u+1)), \min(M, N \cdot (v+1))]$ to create its mesh from. Note that the 2D height sample coordinates are in 2D space and actually represent row and column indices, counting from 0 and up, while the origin is at the top left corner. This is similar to texture UV space but uses a different scale (in texture UV space, the coordinate (*1.0*, *1.0*) would represent the bottom right corner). In Figure 5-7, the camera would be near u *= 0, v = 50*.

The tile triangulation scheme will be explained using the example depicted in Figure 5-9. In this example, a camera would be located near the center of the figure (at *E*), causing the LOD levels (denoted as L0-L2) to increase (and thus coarsen) from the center outwards. As the tiles cover 9 x 9 heightfield grid cells, 100 height samples are used per tile. The triangulation process is best explained as a three-step process.

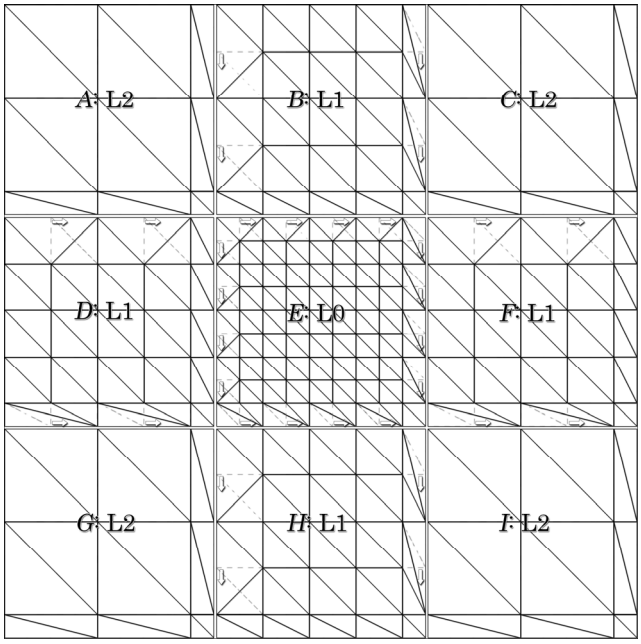


Figure 5-9 **GeoMipMap tiles using 9x9 grid cells per tile for a 28x28 heightfield**

First, the tile is split into quads (i.e. quadrilaterals). At the finest LOD level (L0 in the figure), a tile consists of 9 x 9 quads. Each subsequent LOD level will merge 2 x 2 quads into one quad. However, if the amount of quads in a tile at the previous LOD level isn't divisible by 2, the (bottom/right) remainder will not be merged. Consequently, LOD level 1 through 4 in the example will consist of 5 x 5, 3 x 3, 2 x 2 and 1 x 1 quads, respectively.

Secondly, each quad is split into two triangles. For this implementation, a regular triangulation has been chosen (Figure 5-3, left image), as this simplifies batching all triangles in a tile into a single triangle strip. The diagonal split edge has arbitrarily been chosen to go from the top left to the bottom right of each quad.

Lastly, the triangles that touch the border of a neighboring tile that has a lower LOD level are modified to match the layout of these neighboring triangles. This process is depicted in Figure 5-9 by the arrows, each replacing a vertex for a dotted-lined triangle with another vertex. This replacement process prevents having potential gaps and T-junctions between neighboring tiles. This replacement can be accomplished by rounding the heightfield sample UV coordinates at the tile's border to the first sample coordinate on the border that is used by the neighboring tile with a higher LOD level. This can be implemented using simple bit operators due to the power-of-two nature of the LOD levels. Note that tiles that have a higher LOD level than all their neighboring tiles are not modified in this step. As a result of the vertex replacement, some of the modified become degenerate (zero-area) triangles. Even though keeping degenerate triangles induces a small cost when rendered, the alternative of removing these triangles can be more costly; breaking the regularity in triangulation would either shorten triangle strips or create the need for more complex (and possibly slower) triangulation techniques. Either way, the overhead incurred by keeping these degenerate triangles is relatively small anyway, as there are relatively many more non-degenerate triangles than degenerate triangles in reasonably sized tiles. Moreover, most modern graphics cards handle degenerate triangles very efficiently.

5.2.2.3 Resource Sharing

Triangles are rendered most efficiently when using indexed triangle strips. This is a variation on the indexed triangle list, mentioned in 5.2.1.2. To render an indexed triangle strip, two buffers are used. The first buffer, called the vertex buffer, contains the position of each vertex. The second buffer, called the index buffer, is a list of indices referring to ordered vertices in a vertex buffer. When rendered as a triangle strip, the first three indices in an index list define the three vertices of first triangle. Each subsequent index is used together with the previous two indices to form yet another triangle. Consequently, a list of k indices will render $k - 2$ triangles, making triangle stripping very memory efficient. The triangle

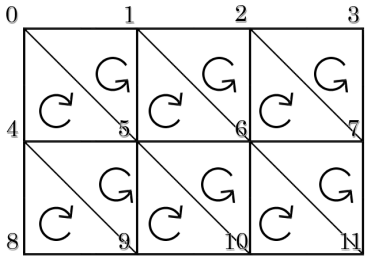


Figure 5-10 **Example of triangles to be 'stripped' using alternate winding orders**

indexing is performance efficient because the vertex sharing allows for sharing of vertex transformation output between neighboring triangles. The winding order, used for back-face culling, of triangles in a strip is alternated for each subsequent triangle. For example, in Figure 5-10, the two triangle index buffers for the two rows would consist of 4-0-5-1-6-2-7 and 8-4-9-5-10-6-11. To merge multiple rows of triangles into one strip, some indices can be repeated and inserted between the strips to create transitional, degenerate zero-area triangles, moving the sequence from the bottom-right of one row to the top-left for the next. For example, in Figure 5-10, the combined triangle index buffer would consist of 4-0-5-1-6-2-7-3-3-8-8-4-9-5-10-6-11, inserting 7-3-3, 3-3-8 and 3-8-8 and 8-8-4 as degenerate triangles. Although the addition of these triangles does not come for free, it is still faster than having one strip (and, hence, one render call) per row of triangles.

Another advantage of the separation of connectivity data (the index buffers) and the geometry data (the vertex buffers) is the potential for reuse. As can be seen from Figure 5-9, the tiles A, C, G and I have similar triangle layouts. As index and vertex buffers can be bound separately to a render call, this separation can be made to exploit the tile similarity to increase memory efficiency. For example, by ordering the vertices in tiles A, C, G and I in the same way, the index buffer can be shared between these tiles. This is also true for tiles B and H and for tiles D and H. For this to work, one vertex buffer is created per tile, consisting of all vertices in scan-line order appropriate for its current LOD level, independent of neighboring LOD levels. This vertex buffer must be updated when either the tile's height samples are modified or when its LOD level changes. However, the index buffer can be shared between tiles that would have identical tile connectivity (i.e. the tile's triangle layouts would be the same). This is the case when two tiles have identical LOD levels and have identical (or finer) LOD levels for their four neighboring tiles. As vertex buffers are independent of neighboring tile LODs, some vertices will be left unused during rendering when a tile has a neighboring tile with a coarser LOD level, as specified in the used index buffer. This way, one vertex buffer is required per tile per LOD level and one index buffer per own/neighbor LOD level combination.

Even this could be extended by having only one vertex buffer per tile, independently of LOD levels, by letting the index buffers of higher LOD levels skip over more and more

vertices. However, this would mean that all vertices (and thus, all heightfield samples) would need to reside in video memory. In contrast, requiring a different vertex buffer for each LOD level and keeping only the current LOD level's vertex buffer in memory will result in much smaller vertex buffers for distant tiles. Furthermore, skipping over many vertices also has a negative effect on performance on modern hardware as this would result in many unnecessary vertex transformations and cache misses unless the vertex ordering is optimized for such usage. Another tradeoff is the required preprocessing time. By not using this last extension, the vertex buffer will need to be recalculated every time the LOD level changes due to camera movement, but as there are many more tiles at coarser LOD levels than finer LOD levels for a large terrain, the average vertex buffer size per tile is relatively small. Also, editing terrain at some distance would only require updating vertex buffers at reduced resolutions. When using the extension, however, camera movement would no longer require updates to vertex buffers when the camera moves some distance, but it would require the vertex buffers to always be updated at the highest LOD level when the terrain is modified. The tradeoffs between update time and speed, together with the much larger memory requirements make this extension less preferable for this research than having a tile's vertex buffer be LOD-dependent.

Another way to exploit the regularity in heightfields is the possibility to split the vertex buffer into two separate vertex buffers: an XZ vertex buffer and a Y vertex buffer (with Y being the vertical component). As can be seen in Figure 5-9, not only the connectivity of tiles like A, C, G and I is identical, the vertex position themselves are identical as well in this top-down projection on the XZ plane, apart from a 2D translational offset per tile. As modern hardware supports complex vertex shaders that are able to combine information from different streams and variables, the XZ components of each vertex can be shared between tiles that use the same LOD level, while the Y-component vertex buffer (i.e. the actual height sample values) is truly unique to each tile. For large terrains, the total number of tiles is much larger than the total number of LOD levels. Hence, memory requirements are effectively reduced to one third. See Figure 5-11.

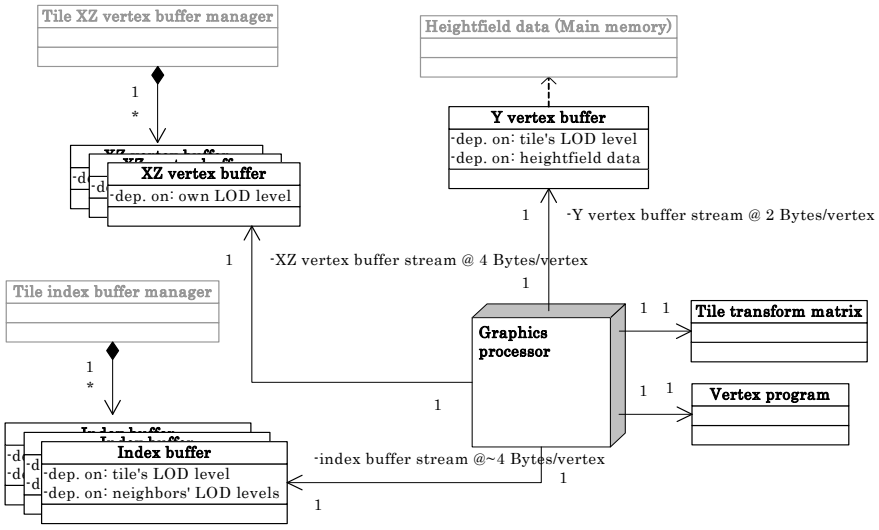


Figure 5-11 **Components required to render a tile**

Using only shared 2 x 16-bit XZ values and unique 16-bit Y values relative to the top-left position of each tile while scaling and offsetting these values in the vertex shader, the vertices can be compressed to a fraction compared to a naïve uncompressed 3x 32-bit floating point implementation, approaching one sixth of the original vertex data (assuming the total number of tiles is far greater than the number of LOD levels).

One parameter of the algorithm that must be chosen carefully is the tile size N. An index buffer can be specified in 16-bit or 32-bit indices. As the maximum number of vertices that can be addressed from a 16-bit indices is 2^{16}, 32-bit indices are required for LOD levels that would need more than 256 x 256 vertices. So, less 16-bit index buffers will be used for larger tile sizes. These 16-bit buffers are both smaller and faster to process in hardware. Furthermore, smaller tiles can be culled more precisely by the viewing frustum, reducing overhead. Smaller tiles have another advantage. As tiles are always updated as a whole, heightfield modifications that are relatively small to a tile waste processing power and bandwidth. So, smaller tiles means less updating overhead for small modifications. As there can only be $\lceil \log_2 N \rceil$ LOD levels for a tile size of N x N, larger tiles will offer more LOD levels. This means that (very) distant tiles will potentially be rendered with unnecessary detail when tiles are too small. Also, modern hardware can only approach its maximum triangle throughput when the triangles are batched together in as less render calls as possible, making larger tiles more preferable. The optimal tile size has been decided on by experimentation. As can be seen in Figure

5-12, the render frame rate was found to be highest near $N = 200$. Note that this frame-rate is hardware dependent and future graphics hardware will probably perform better with even larger tiles. For this research, $N = 192$ has been chosen, which is the exact N with the highest fps in the figure. The frame rate is also dependent on the maximum allowed tile screen error e. As mesh density in both X and Z direction is directly related to e, a quadratic reciprocal relation between e and the frame rate is expected. This relation can also been observed from the experimental results presented in Figure 5-13. The results shown in Figure 5-12 and Figure 5-13 were obtained using an Intel Core 2 Duo 2.0Ghz and NVidia Geforce Go 7950 GTX 512 MB system.

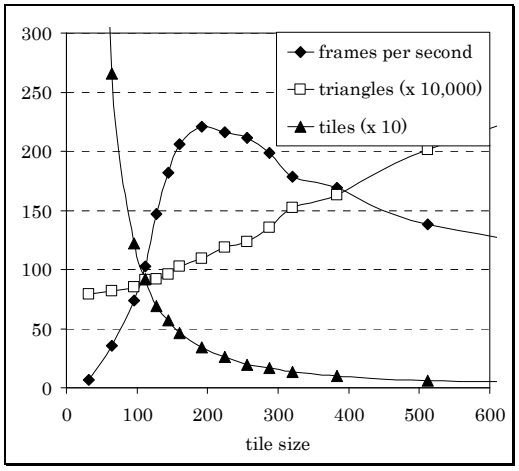


Figure 5-12 **Frames per second, rendered triangles and rendered tiles as a function of (square) tile size. The data points are averaged values, measured from multiple camera positions with *e* = *3* @ 1024 x 768**

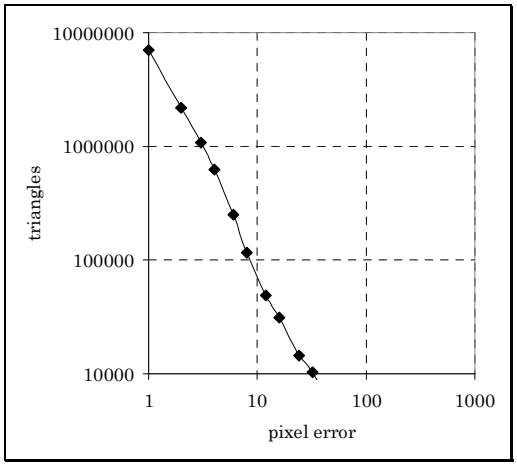


Figure 5-13 **Amount of rendered triangles as a function of the maximum pixel error. The data points are averaged values, measured from multiple camera positions @ 1024 x 768**

5.3 Terrain Texturing

To make interesting rendered images of any type of geometry using modern programmable graphics hardware, geometry (i.e. the triangle meshes) will have surface properties assigned to it. In general, these properties consist of local mapping parameters (i.e. the texture mapping) and a pixel shader. The shader uses the local parameters, the camera direction, the local geometry and possibly multiple input images (called textures) and other parameters to calculate the color of each covered 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). Lighting calculations can be balanced between vertex shaders and pixel shaders to get per-vertex and per-pixel effect, respectively. As there are generally less vertices than pixels rendered at any time, it is often faster to calculate effects in the vertex shader.

When a heightfield is to be used in a real-time engine, this heightfield will generally be rendered as a set of triangles. Simply assigning uniform colors to these polygons will not create very convincing images. Photorealistic textures can be assigned instead to increase the visual resolution of the material the terrain is made of. Typical textures include images of mud, snow, dirt, sand, grass and rock. These terrain textures can be created by artists from edited photographs or might even be generated procedurally.

The exact terrain tessellation and geometry rendering technique can differ per game engine. However, the input to these algorithms is typically always the same: a 2D matrix of height values. Terrain texturing, on the other hand, is much less standardized. Different techniques require different types of data and vary in realism and complexity. An editor that can be used to aid in the process of texturing the terrain would, for example, support brushes and parameters to paint or procedurally assign different textures to different areas, respectively. As this research is mainly concerned with heightfields synthesis and editing, and not with texturing for one specific engine, texturing techniques are less relevant. However, a terrain editor requires some form of lighting and texturing to give more context to the edited heightfield.

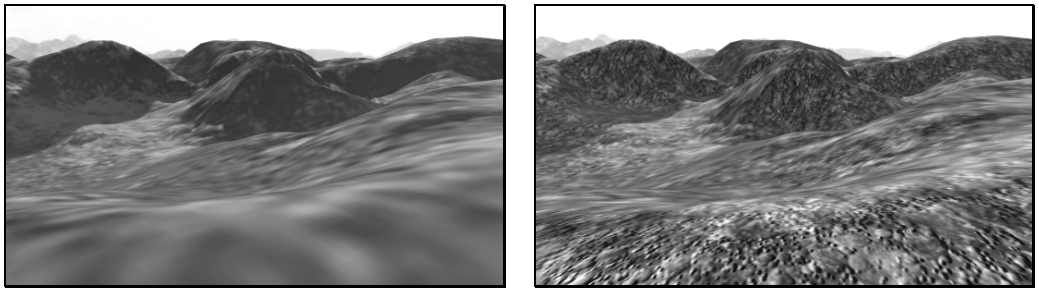
The testbed was fitted with a procedural texture mapping implementation of its own. This method somewhat resembles the possibilities of some of the more modern engines and can be used to visually approximate the texturing of these engines. Because the obtained results were found to be quite good and interesting, the details of this texturing method are discussed in some detail this chapter. To create a more complete context, several texturing techniques found in practice are discussed first in Section 5.3.1 before the details of the implemented texturing method is discussed in Section 5.3.2. That last section also includes a technique to hide a common texture distortion artifact found in heightfield-rendering games. Although it is expected that some newer renderer implementations use a similar technique to tackle this problem, it is worth discussing

some of its details, as no other literature has been found during this research that covers this solution.

As texturing itself is not enough to create appealing scenes, a lighting model is typically applied by a game engine as well. As these lighting models are even more engine-specific, only the testbed's implementation is discussed in some detail in Section 5.3.2.3.

5.3.1 Texturing Techniques

What follows is a survey of different texturing techniques found in practice with different tradeoffs between speed, detail and flexibility. This section creates a base and context for the discussion on the testbed texturing implementation found in Section 5.3.2.



Global texturing **Global + detail texturing**

Figure 5-14 **Global texturing with and without detail texturing**

5.3.1.1 The Global Texture

The simplest texturing technique is analogous to the idea of heightfields. A single color image (i.e. texture) is assigned 1:1 to the whole terrain using a vertical orthographic projection. Obviously, the disadvantage of this technique is memory usage, as using images that consist of more than only a few color samples per height sample are prohibitively memory intensive. It might suffice for flight simulators, thus rendering the terrain from a great distance, but produces poor imagery for applications that show the terrain from only a few meters above ground level. Also, creating a global texture 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 5-14.

5.3.1.2 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 fine detail look (from the detail texture). See Figure 5-14.

5.3.1.3 Quad Texture

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 grid cell. This can be implemented by replacing or extending a global texture to assign a number to each cell, indexing into an array of detail textures. 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 5-15. This technique can be interpreted as a 3D application of classic 2D arcade-style 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.

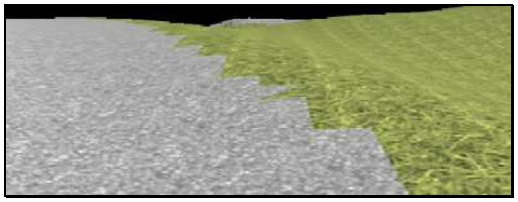
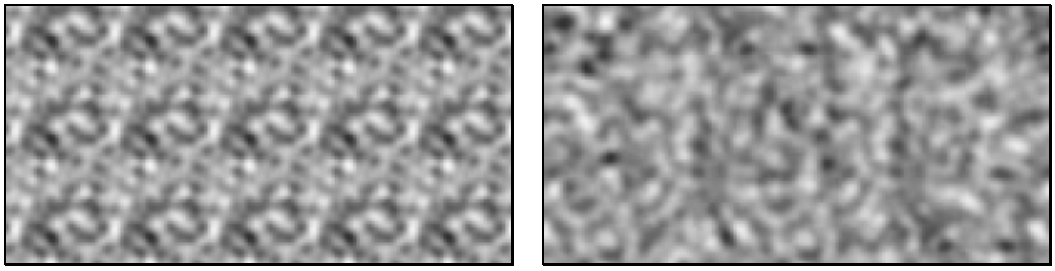


Figure 5-15 **Quad texturing without transitions. Note the seams between the rock and grass texture. From [DEXT05]**

The number of transition textures is quadratic in the number of different terrain types, becoming the limiting factor.

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 5-16. 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 a seamless result. A set of recursive production rules is responsible for the actual 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).



Standard, periodic tiling **Aperiodic tiling**

Figure 5-16 **Example of standard tiling and aperiodic Wang tiling. From [STAM97]**

5.3.1.4 Texture Splatting

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 position-dependent weighted average of different material textures. These weights are assigned per height sample, typically using a texture for weight look-ups during rendering [BLOO00] [DEXT05]. See Figure 5-17. Typically, only a

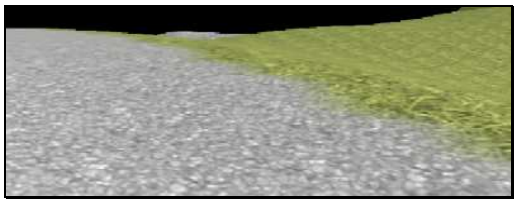


Figure 5-17 **Splat texturing. Compare to Figure 5-15. From [DEXT05]**

few materials are blended 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 (rectangular) 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.

Quad texturing is currently the algorithm used by most game engines, albeit with different numbers of supported textures, partitioning techniques and editing capabilities. For example, some engines support brushing weights onto the terrain, others support more automated methods, as described next.

5.3.1.5 Procedural Splatting

The splatting algorithm can be extended to calculate blend weights based on geometry. This simplifies texturing from manual per-sample weight assignments to tweaking parameters to the procedural calculation. Of course, the output of these calculations might be blended or overwritten by some manual value where more control is necessary. However, results obtained solely by procedural calculations can be quite believable for a good reason: the soil type, snow line and undergrowth in real terrains on earth are typically themselves influenced by the underlying geometry. Geometry-based inputs for procedural texturing that can easily be evaluated, like the local altitude and slope, are often found to be enough when combined with some randomness to create believable texturing. Other, more complex inputs can be used as inputs as well. In [HAMM01], more factors that influence the local sun, wind and rain conditions are used: 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. The procedural weight calculations can be done either off-line or on-line. For off-line calculation, the calculated

results still need to be stored explicitly. On-line calculation can be implemented either to be calculated by either the CPU or GPU during rendering.

The procedural techniques are typically based around layers of textures stacked on top of each other, with each layer determining its own opacity. This is somewhat intuitive, as snow, for example can be stacked upon grass, which, in turn, can be stacked on top of sand. The designer is expected to define these layers. One possible layer stack is depicted in Figure 5-18. The ordering of layers 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% opacity, the 'lower' layer will be completely covered, independently of the opacity setting of this lower layer. An example of a layered set of materials can be found in Figure 5-18. 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 opacity of child layers is multiplied by the opacity 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 opacity.

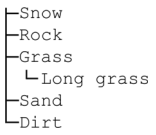


Figure 5-18 **Example of a user-defined material layer hierarchy**

5.3.2 Implementation Details

The previous section discussed many different texturing techniques. The exact available technique(s) depend(s) on the technology supported in game. Consequently, visual fidelity, storage requirements, editing flexibility and labor intensiveness offered in a (terrain) editor are dictated by the underlying texturing implementation.

A subset of texturing ideas and practices has been implemented in the testbed implemented for this research. The focus of this research lies with heightfield synthesis and editing. As there isn't one standard format in terrain texturing and the testbed isn't created around a single application (e.g. engine), creating texturing that would interoperate easily with other applications in a tool chain was not viable. Consequently,

the texturing output that is available in the testbed will not be interchangeable with other applications.

Still, the testbed texturing can be used to approach the result of terrain renderings in a target application by supporting a simple material editor, splatting techniques, controllable procedural texture weight assignment and the ability to load and save settings. This is important for several reasons. As texturing can visualize semantics (e.g. cliff at steep areas, snow on mountain peaks) and fake geometry detail (e.g. cracks in rock textures), even a rough approach of the desired texturing can give more context to the geometry. Also, good texturing and lighting results in an increased perception of depth, helping users to better interpret the rendered imagery.

5.3.2.1 Texture Splatting

At the implementation's basis lies a four-layer texture splat pixel shader that can be controlled from the material editor. Having exactly four layers is a compromise between rendering speed and flexibility, without requiring more complex and precalculation-intensive area partitioning techniques. The textures for each layer can be chosen freely. Examples of such layer textures are snow, rock, grass, sand and dirt images. The local weights of the different layers are a procedurally calculated mix of local height, slope and noise.

These weights are calculated in the terrain pixel shader itself, using a packed format of user parameters. Note that only the texture blend weights are calculated procedurally, not the textures themselves. As described in Section 5.2.2.3, the vertices that make up the terrain geometry are split into a shared XY component tile vertex buffers and unique Y component tile vertex buffers. To be able to calculate light influences and slope-dependent texturing for this implementation, the Y tile vertex buffers are appended with directional information. Directional vertex information is commonly represented as 3D 'normals', defined as normalized vectors that are perpendicular to the local mesh patch. However, in the case of heightfields, a 2D $(\frac{\delta y}{\delta x}, \frac{\delta y}{\delta z})$ gradient vector is more memory efficient. Each vertex is chosen to consist of one 4D vector: one height component, two gradient components and one spare component. In the terrain vertex shader, the gradient

is transformed into a 3D normal. During rasterization, this information is interpolated over the terrain's triangles and passed on to the terrain pixel shader. The pixel shader uses this normal for lighting calculations and slope-dependent texture weighting. A 'Y' vertex buffer is recalculated when either the LOD level changes or the terrain is locally modified. The gradients contained in this vertex buffer are calculated as follows:

$$(\frac{\delta y}{\delta x}, \frac{\delta y}{\delta z}) \approx (H_{x,y} - H_{x-1,y}, H_{x,y} - H_{x,y-1})$$

with H being the heightfield matrix. This will give only a rough approximation of the actual gradients, but it is also very fast. For the application of terrain editing, this tradeoff makes sense: terrain update speed is considered more important than rendering accuracy.

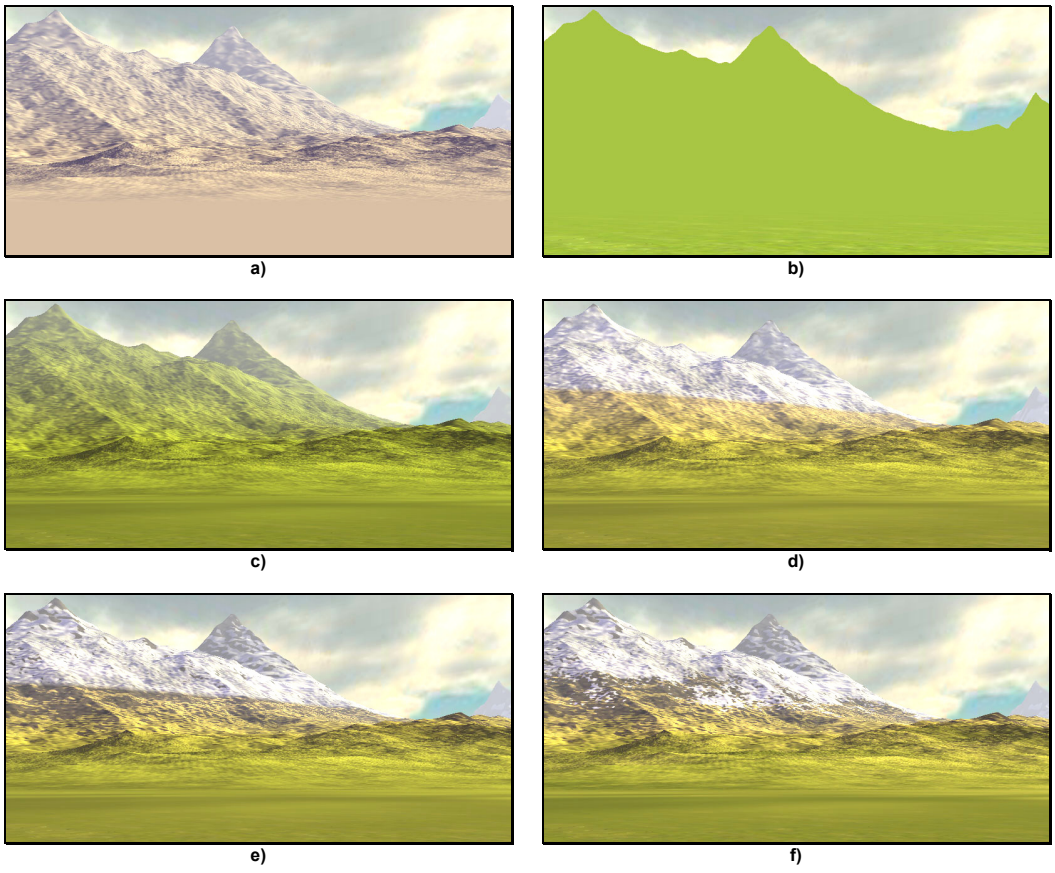
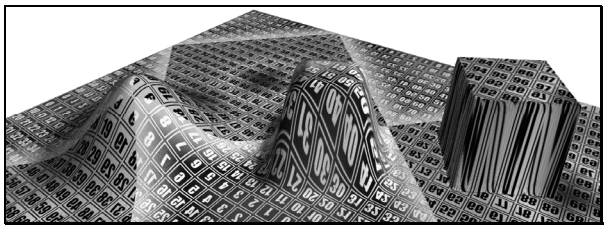


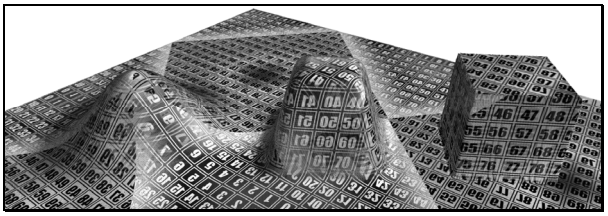
Figure 5-19 **Testbed heightfield texturing breakdown. a) Only lighting. b) Only one texture layer. c) Lighting and texture layer combined. d) Multiple lighted texture layers with height-dependent weight assignment. e) Height and slope-dependent assignment. f) Height, slope and noise-dependent assignment**

5.3.2.2 Texture Projection

As described earlier, texturing needs a projection to map 3D world positions to 2D texture coordinates. Traditionally, an orthographic vertical projection is used for heightfields. However, very steep terrain will then stretch textures that are vertically projected, as can be seen in Figure 5-20a. This effect can also be seen in many (slightly older) games. The vertical projection projects a pixel's 3D coordinates on the heightfield's horizontal XZ plane to get a texture's 2D UV coordinates.



a) Standard vertical projection



b) Normal-weighted X, Y & Z projection

Figure 5-20 Effects of standard and extended texture projections on differently sloped shapes. Please note that the diamond-shaped color gradients in the applied texture have no particular meaning

A more complex texturing technique has been implemented which can be chosen instead of the standard projection. This gives the user a tradeoff between performance and visual output. The extended texturing technique works by projecting the 3D coordinates not only on the XZ plane, but also on the XY and YZ planes. These three 2D coordinates are used to do three independent texture lookups in the pixel shader for each layer. Next, the looked-up texture colors are combined using a weighted sum:

$$N' = \begin{bmatrix} N_x N_x \\ N_y N_y \\ N_z N_z \end{bmatrix}$$

$$C_{texturing} = \begin{bmatrix} C_{yz} & C_{xz} & C_{xy} \end{bmatrix} \cdot \frac{N'}{\|N'\|}$$

where N is the local normal vector and C_{yz}, C_{xz} and C_{xy} are the three sampled texture RGB colors, represented as 3D column vectors. For flat terrain, this simplifies to $C = C_{xz}$,

as expected. For steeper terrain, the horizontal projections will start to dominate the blend, effectively hiding the stretched texturing of C_{xz}. Obviously, this method of texturing uses three times more texture look-ups in the pixel shader than the more standard vertical projection implementation and can have a significant effect on the frame rate. Instead of the component-wise multiplication in N', N' could also have been defined as a component-wise *abs(N)*. However, multiplication has the effect of increasing the relative weight of the most dominant axis and decreasing the relative weight of less dominant axes. This is desirable because blending too many textures together creates an overall muddled look. As can be seen in Figure 5-20b, using a blend of multiple projections effectively eliminates texture stretching. Note the smooth transitions in the leftmost shape.

5.3.2.3 Lighting

Obviously, texturing itself is not enough to produce (nearly) photo-realistic images; a lighting model must be applied as well. Sun light interacts with the atmosphere and terrain in a complex way. Complex models exist that approximate atmospheric scattering, soft (area) sky lights and interreflections on the terrain. Atmospheric scattering can be reasonably approximated in real time [ONEI05] [WENZ06]. Area light and interreflections are much more difficult to calculate and are currently still beyond the reach of hardware accelerated real-time graphics. Fairly exact models can, however, be precomputed using general techniques like radiosity solvers [GORA84]. Slightly less exact models exist that only consider direct lighting and occlusion. These precomputed results can be stored as direction-independent ambient occlusion factors or direction-dependent structures like horizon maps [MAX88], spherical aperture caps [OAT06] and spherical harmonics [SLOA02]. These data structures can then be used in real-time.

Obviously, for the purpose of terrain editing, it is considered a bad idea to calculate expensive lighting interaction in an elaborate preprocessing step every time the terrain is modified. And as photo-realistic rendering is less important during editing, it makes sense to use a simpler lighting model that does not need preprocessing. Consequently, a relatively inexpensive lighting model is proposed and has been implemented in the testbed.

Humans are very good at interpreting lighting and shading hints in images of 3D objects (e.g. terrains) to better understand their shape. Rendering virtual terrain without any lighting would create very flat looking imagery that is hard to interpret, especially when the terrain is textured with similar looking images. Compare Figure 5-19b and Figure 5-19c. Subtle differences in color and lighting, as well as atmospheric scattering, depth of field and fog convey the illusion of depth. The main purpose of the proposed lighting model is to create images that look as good as can be expected from simple, non-preprocessed data and show as much visual definition (i.e. illusion of depth) as possible.

a) Standard diffuse lighting: *s = 1*

b) Exaggerated lighting: *s = 10*

Figure 5-21 **Effects of standard and exaggerated slopes for lighting calculations. Note that differences are most visible at nearly flat areas**

As terrain receives light from both the sky and the sun in realistic situations, terrain that faces the sun should be lighter and more yellow and terrain that faces away from the sun will have a darker, more blue-grayish shade to it. One way of accomplish this effect to calculate the following:

$$\alpha = \max(N \bullet L, 0)$$
$$C_{light} = (1-\alpha)C_{sky} + \alpha C_{sun}$$
$$C_{combined} = C_{light} \cdot C_{texturing}$$

where $\underline{N}$ is the local normal unit vector and L is the unit-vector in the direction of the sun. $C_{texturing}$ is the color from the texturing step, as defined in Section 5.3.2.2. This calculation is done per pixel in the pixel shader. The formula is a variation on the standard ambient/diffuse directional light model. N can be calculated as follows:

$$N' = \left[s\frac{\partial y}{\partial x} \quad 1 \quad s\frac{\partial y}{\partial z} \right]$$
$$N = N' / \|N'\|$$

For standard lighting, s should be 1. However, when $s > 1$, slopes are considered to be s times steeper during lighting calculations. As the slopes will be exaggerated, so will the lighting output. There is no actual physical model that would support this idea, but using larger values for s can help to give the terrain more visual definition at nearly flat areas. See Figure 5-21. The testbed implementation offers an option to the user to set s.

One way of modeling the effect of atmospheric scattering and fog is as follows:

$$\alpha = e^{-f \cdot d^2}$$
$$C_{output} = (1-\alpha)C_{combined} + \alpha C_{fog}$$

Here, C_{fog} is the color of the sky in the distance and would normally be close or identical to C_{fog} but is offered as a separate variable for increased flexibility. f controls the density of the fog and d is the distance to the camera. More complex models do exist, but this more standard computer graphics implementation is fast and simply looks good enough for our purposes.

A simple way to increase the visual definition on a 3D image is set the light source (e.g. the sun) at a low angle, in a direction that is somewhat perpendicular to the camera's viewing direction. So to get the best lighting during editing, it should be easy to change the lighting conditions when the camera changes direction. For this reason, the user can change the sun's height and direction in the testbed by moving the mouse while holding only one key.

6 Heightfield Editing Techniques

This section presents a survey of techniques that are found in literature and used in current terrain editor applications. Simple, low-level brushes are discussed in Section 6.1. These are found mostly in interactive game editors. Examples of these low-level brushes are mouse-controlled local vertical heightfield pushing, pulling and leveling operations that operate at a specified location within a specified radius. Deforming terrain with brushes is intuitive, but as the brushes use simple shapes, it is difficult and time-consuming to create complex, natural scenes with these.

Terrain deforming simulations are discussed in Section 6.2. Also, procedural techniques are discussed in Section 6.3, followed by more specific 'building block' noise functions in Section 6.4. Procedural algorithms attempt to capture the complexity of natural terrain mathematically, while being faster to calculate than actual simulations. Some of the currently available level edit tools 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 with great ease. However, these tools only offer global, high-level parameters, which generate a whole terrain at once, making it hard to control exact placement of different desired landscape features (e.g. mountains and lakes) throughout the landscape. Typically, parameters are set, and the whole terrain is generated after waiting somewhere between a few seconds and a number of minutes. 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: the low-level brush tools. 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.

Sections 6.1 to 6.4 cover techniques and algorithms found in current applications. Section 6.5 and 6.6 offer several ideas to overcome the current difficulties described

above. This includes use of brushes for more complex operations, the use of layers and support for different blending operations. Many of these ideas are inspired or directly translated from other disciplines, like 2D image editing application. This was already hinted at in Section 2.2. Section 6.7 offers a short evaluation of the current and suggested techniques and discusses the need for efficient implementations of these techniques, to which a solution is explored in the remainder of this dissertation.

6.1 Low-level Brushes

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 that come with a game engine.

6.2 Simulation

This section discusses different simulation techniques found in literature and some terrain synthesis applications, capable of 'aging' or eroding an input terrain to create a new, more natural output terrain. These simulations are typically applied globally (i.e. on the whole terrain).

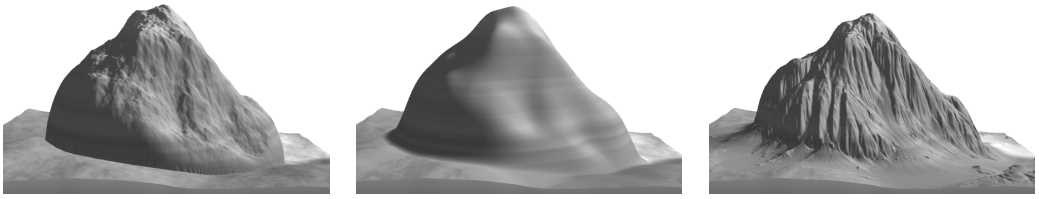


Figure 6-1 **Different types of erosion. Left to right: unaltered procedural heightfield, thermal erosion and fluvial erosion.**

These erosive simulations are only very crude approximations of the processes that occur in nature. However, impressive results are obtained with these algorithms. They are typically iterative, each iteration covering a time step. The covered 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 6-1.

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 the relative differences. The amount of material deposited is a fraction *c* times the height difference between the neighbors minus *T*. See Figure 6-2. In effect, a maximal slope is enforced after enough iterations are executed.

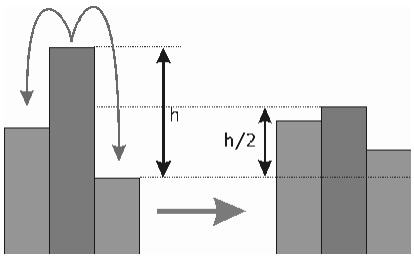


Figure 6-2 **Thermal erosion deposition with c = 0.5, T = 0. From [BENE01b]**

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.

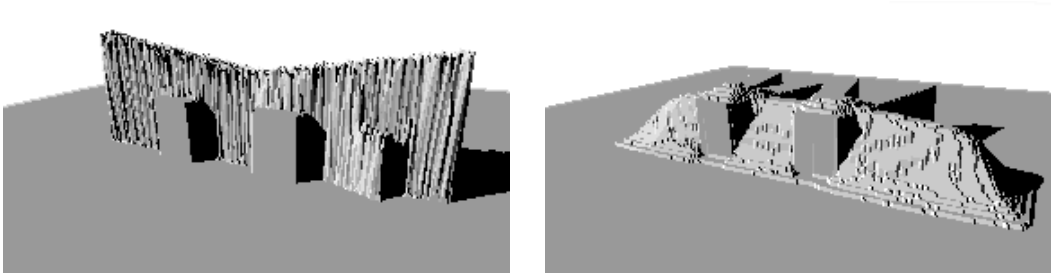


Figure 6-3 **Before (left) and after (right) erosion was applied to the letter W consisting of a hard material and a layer of soft material on top**

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 6-4. 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 6-3 shows a result that would be difficult to achieve with non-layered erosion.

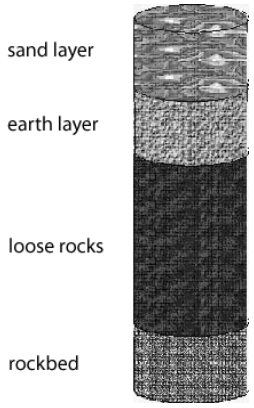


Figure 6-4 **Example of a layered core sample. From [BENE01a]**

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.

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

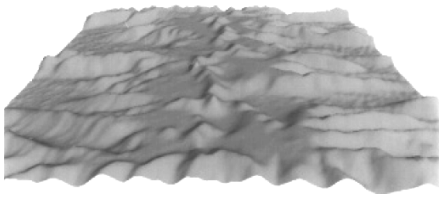


Figure 6-5 **Result of 100 iterations of fluvial water erosion. From [CHIB98]**

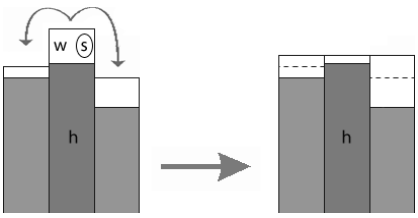


Figure 6-6 **Fluvial erosion water transfer**

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 6-5 for an example of a terrain created by this method.

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 6-6. 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)*.

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 6-1.

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 6-7. 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.

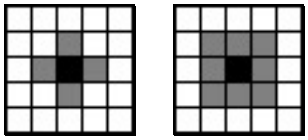


Figure 6-7 **Neighboring cells (grey) in the Von Neumann neighborhood (left) and Moore neighborhood (right)**

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 6-8. Although results are impressive, calculation time currently prohibits its use in interactive applications.

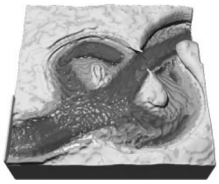


Figure 6-8 **Oxbow lake-like features carved out by water simulation in a terrain patch. From [BENE06]**

In Section 6.1, several inexpensive, low-level brushes were discussed. Although it would be possible to create complex terrain with these brushes, a user is greatly aided with brushes that create more complex and natural terrain as discussed in this section. The approaches discussed here are computationally quite expensive as geological processes are effectively simulated over time. In the next section, 'procedural' algorithms are discussed that also try to offer complex and natural results but, unlike the algorithms discussed here, can be evaluated much faster.

6.3 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.

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]. This section discusses procedural algorithms related to the generation of natural heightfields.

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.

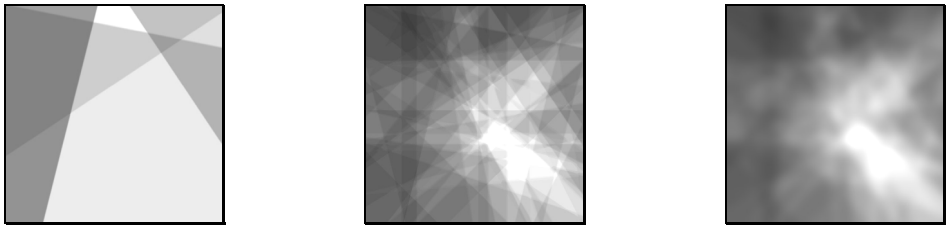
fBm surfaces do posses some features that visually distinguish them from real mountainous terrain. The increments of an fBm process have the property of being isotropic and stationary, creating terrain that is statically invariant under translation and rotation. This will result in terrain that looks 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.

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.

6.3.1 Poisson faulting

'Fault formation' and 'particle deposition' are two variants of Poisson faulting. Fault formation is introduced in [KRTE01] and is illustrated in Figure 6-9. 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.



After 4 iterations | After 64 iterations | After 64 iterations and filtering

Figure 6-9 **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 6-10. 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 6.2). 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.

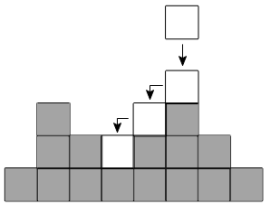


Figure 6-10 **Flow simulation in particle deposition**

6.3.2 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 Section 6.3.4 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 6-11.

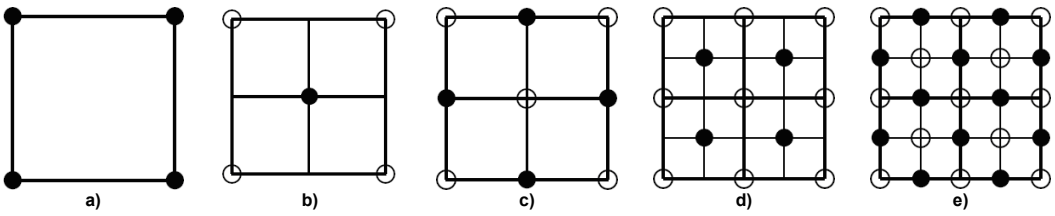


Figure 6-11 **Square-diamond midpoint displacement. b) and d) are intermediate results after applying the first phase. c) and e) applied phase 2. From [OLSE04]**

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.

6.3.3 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).

6.3.4 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 6-12.

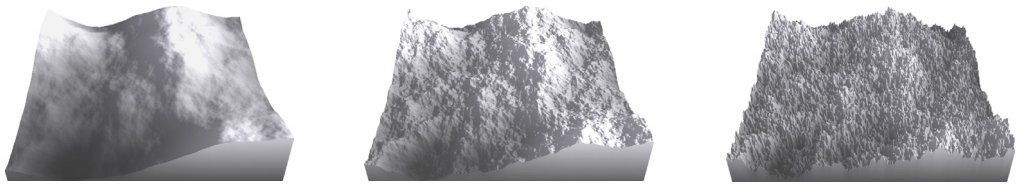
The actual noise function can be constructed in different ways, each having a different characteristic band-pass quality and construction speed. An overview of these functions is given in Section 6.4.

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:

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

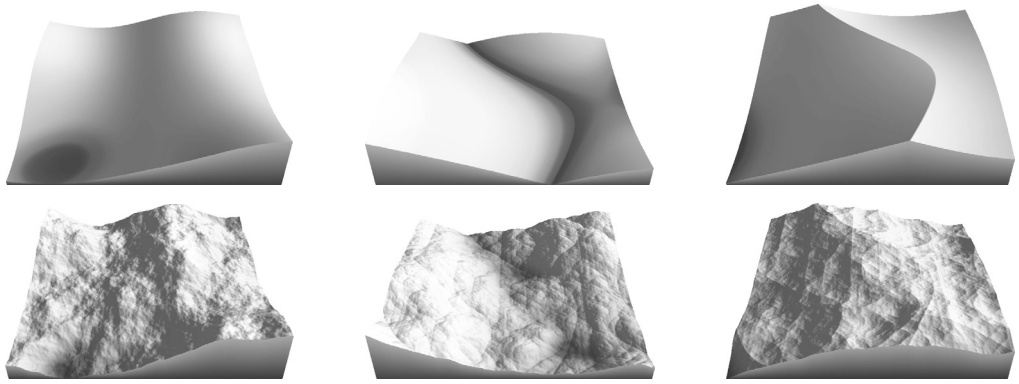
The turbulence function $T(n)$ [PERL89] is one of the first algorithms to explore the possibilities of this generalization by defining $T(n)$ as $abs(n)$. Taking the absolute value of $[-1, 1]$ noise folds it at each zero crossing, creating discontinuities and doubling the number of (positive) peaks. This creates more billowy, turbulent, cloud-like fractal landscapes. See Figure 6-13. 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.

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.



H = 1 : Df = 2, w = ¼√2 H = ½ : Df = 2½, w = ½ H = 0 : Df = 3, w = ½√2

Figure 6-12 **Heightfield of different fractal dimensions. Perlin noise**



fBm : T(n) = n Billowy : T(n) = abs(n) Ridged : T(n) = 1-abs(n)

Figure 6-13 **Heightfields with one octave (top row) and eight octaves (bottom row) of transformed noise. Perlin noise, H = ½**

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 6-14 for an example. In [EBER03, p. 498-506], different variants of these multifractal techniques are discussed in detail.

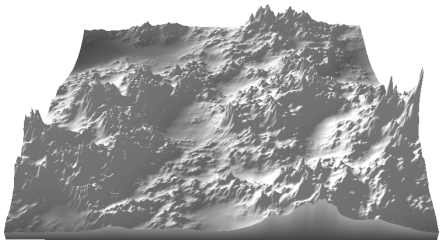


Figure 6-14 **Height-dependent high frequencies**

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.

6.3.5 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 6-15. 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.

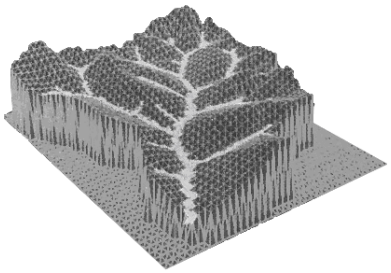
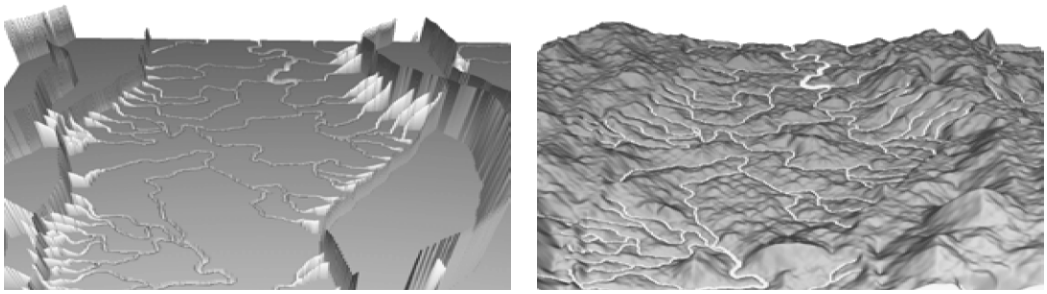


Figure 6-15 **Drainage network and fitted (non-fractal) surface. From [KELL88]**



Heightfield after ridge and water particle simulation **Heightfield after midpoint displacement**

Figure 6-16 **Fractal landscape with river network. From [BELH05]**

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 6-16.

6.3.6 Range and Domain Mapping

Another way to create more varying landscapes is to transform the output (or, range) 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 range mapping are the *bias* and *gain* [PERL89] functions:

$$bias_b(z) = z^{\log b / \log ½}$$

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

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 6-17 for examples of these functions, together with their effect on a heightfield.

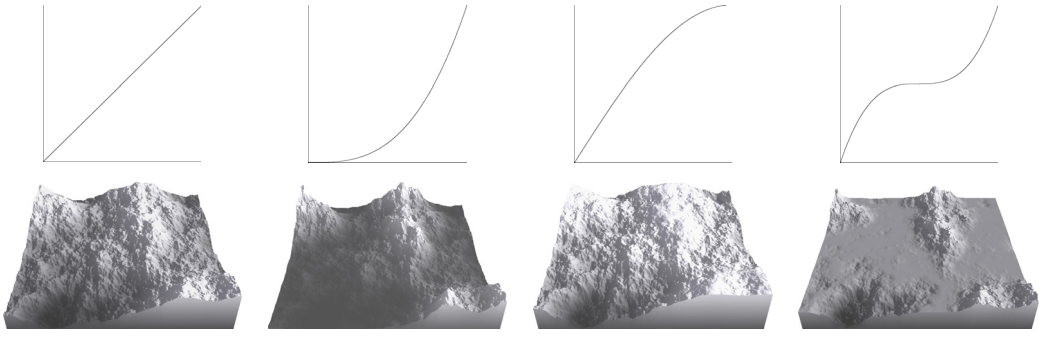


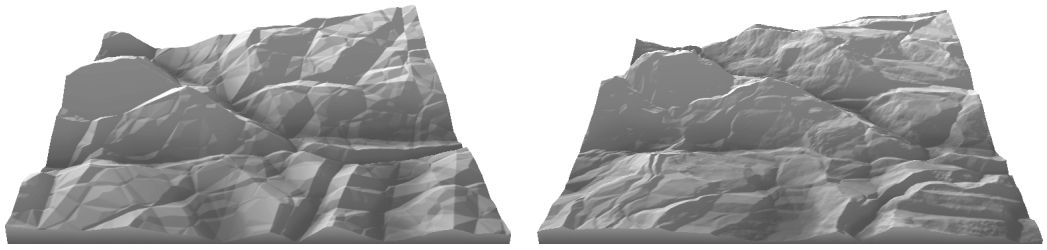
Figure 6-17 **Heightfields after post-processing. Perlin noise, H = ½. Top row: P mapping, bottom row: result**

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 6.4) 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 6-18. This is another example of cascading different functions to increase the visual complexity of the result.



Original heightfield without domain mapping **Heightfield with domain mapping**

Figure 6-18 **Voronoi heightfield without (left) and with (right) noise distorted input**

6.4 Noise functions

As described in the previous section, 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 section discusses different noise basis functions for use in noise synthesis-based terrain generation.

Fourier synthesis was already discussed in Section 6.3.3, 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 computationally intensive, making Fourier synthesis less practical than alternatives.

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 G table is also of size N and contains random numbers in the range $[-1, 1]^2$ for the 2D case. Then, a pseudo random value can be calculated by evaluating $G(H((x + H(y)) \bmod N))$ [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 computationally intensive but, depending on the application, can be worth the extra effort. See Figure 6-19. 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.

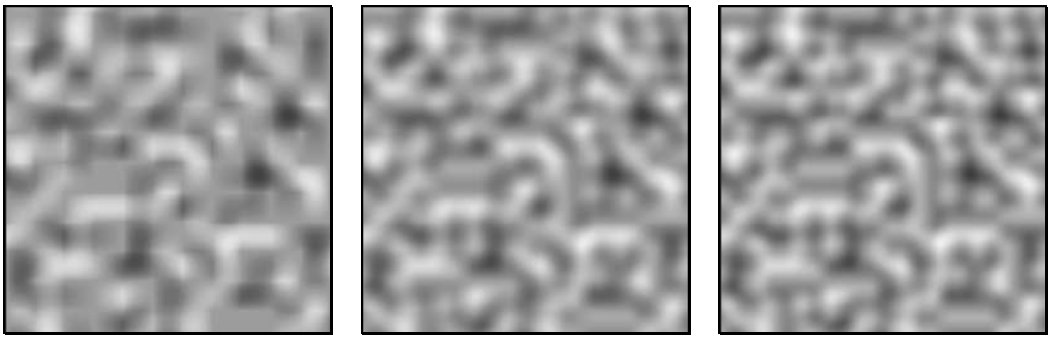


Figure 6-19 **Different gradient noise interpolation schemes**

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 [CATM74]. 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].

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.

Voronoi Noise

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 6-20 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 6-18.

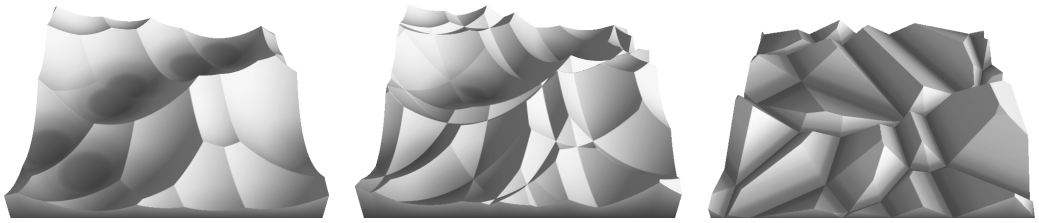


Figure 6-20 **Voronoi diagram 'noise'**

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

6.5 Complex Brushes

Low-level operations can be ideal when only small changes are needed. The user control offered by using the simple and unnatural brushes is large. Indeed, every type of terrain can be created with these tools by a good level designer given enough time. High-level procedural tools used in generation tools typically offer the opposite. Control is offered using only a number of parameters that can be tweaked. Output is reasonably natural without much effort from the user.

Obviously, tools that would fit somewhere between the global procedural terrain synthesis tools and the local brush operation tools or would mix best of both worlds would certainly find their use in the level design process. For example, 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. Like the low-level brush editing tools, a brush with a user-defined radius and falloff curve could be offered as a local interactive tool to apply procedural techniques locally and controllably, adjusting the terrain while brushing with simple mouse strokes. The parameters to the procedural

technique could then be made adjustable through the use of sliders and presets or could even be coupled to other sources (e.g. heightfields, noise generators, tablet stylus tilt).

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 a 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 section 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, which would globally amplify or fade away the effect. Also, more advanced, non-linear and order-dependent 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, which would be important as explained in Chapter 3. 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 both more memory intensive and computationally intensive, especially when many layers are used during editing. Collapsing one or more layers (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.

6.6 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 6.5, 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 some of algorithms discussed before, these techniques were developed as or inspired by image editing techniques, but can transparently be applied to heightfields.

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 6-21 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.

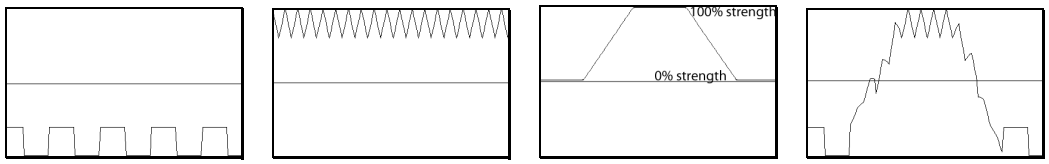


Figure 6-21 **Heightfields after post-processing. Perlin noise, H = ½. Top row: P mapping, bottom row: result**

In [BURT83], an image blending technique is introduced that uses Gaussian and Laplacian pyramid decompositions of the source and destination image to blend these differently for different frequencies or scales. Low-frequency signals are blended over a longer distance (i.e. feathered more), while higher-frequency signals are blended over shorter distances. This is an example of multi-resolution blending. Gaussian and Laplacian pyramids are discussed first, before the actual blending algorithm is described.

The Gaussian image decomposition 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 instead. 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 6-22.

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), often simply called 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 $N \times N$.

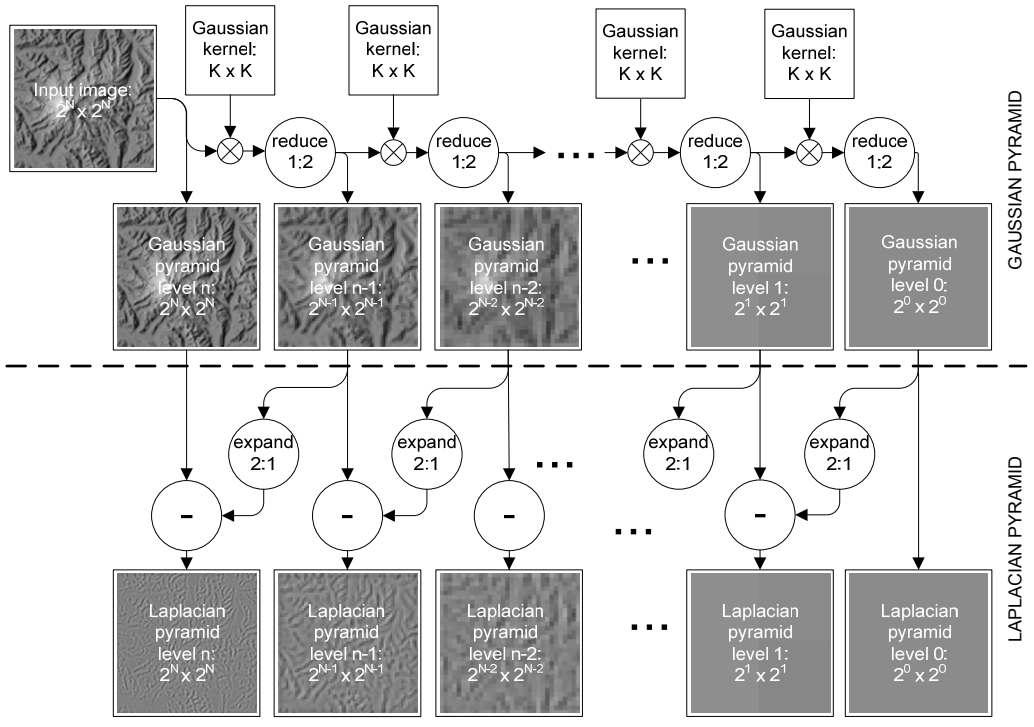


Figure 6-22 **Construction of the Gaussian and Laplacian image pyramid**

The actual multi-resolution blending 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 calculating 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 6-23 for the synthetic terrain cross section of Figure 6-21.

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.

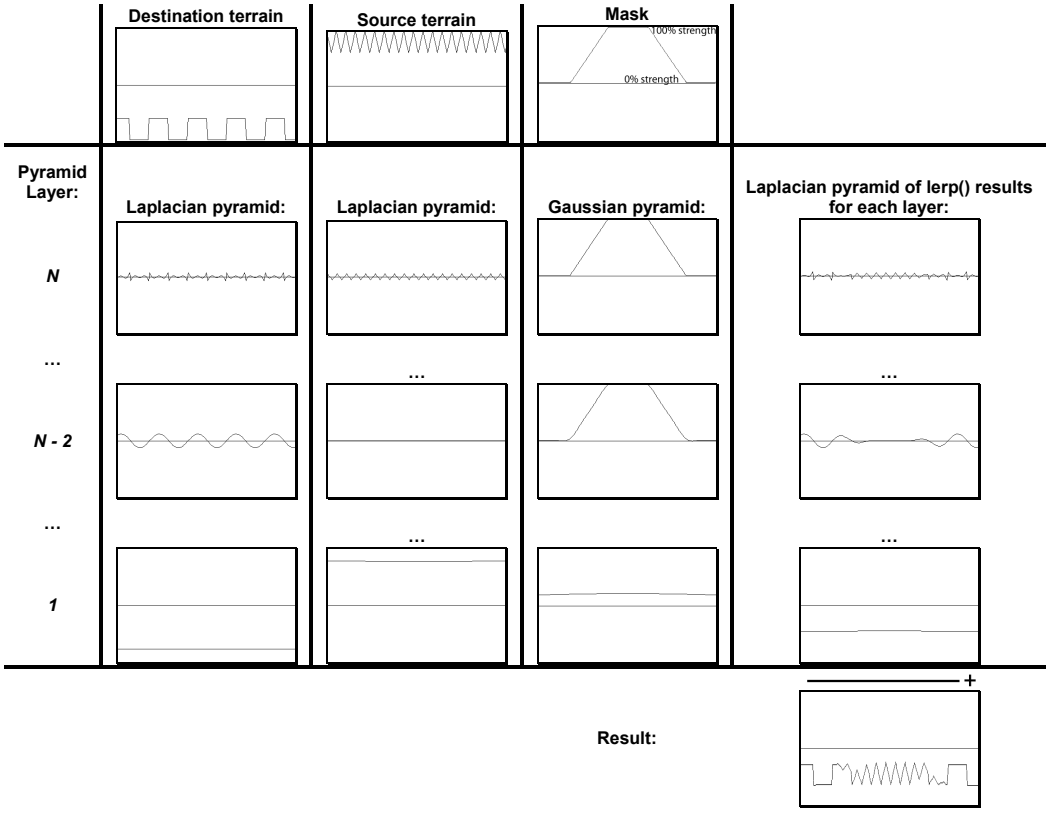


Figure 6-23 **Multi-resolution blending of a terrain heightfield cross section**

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 6-24 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 6-23 and Figure 6-24 this shows as a change of the mean height. This might or might not be appropriate for different situations.

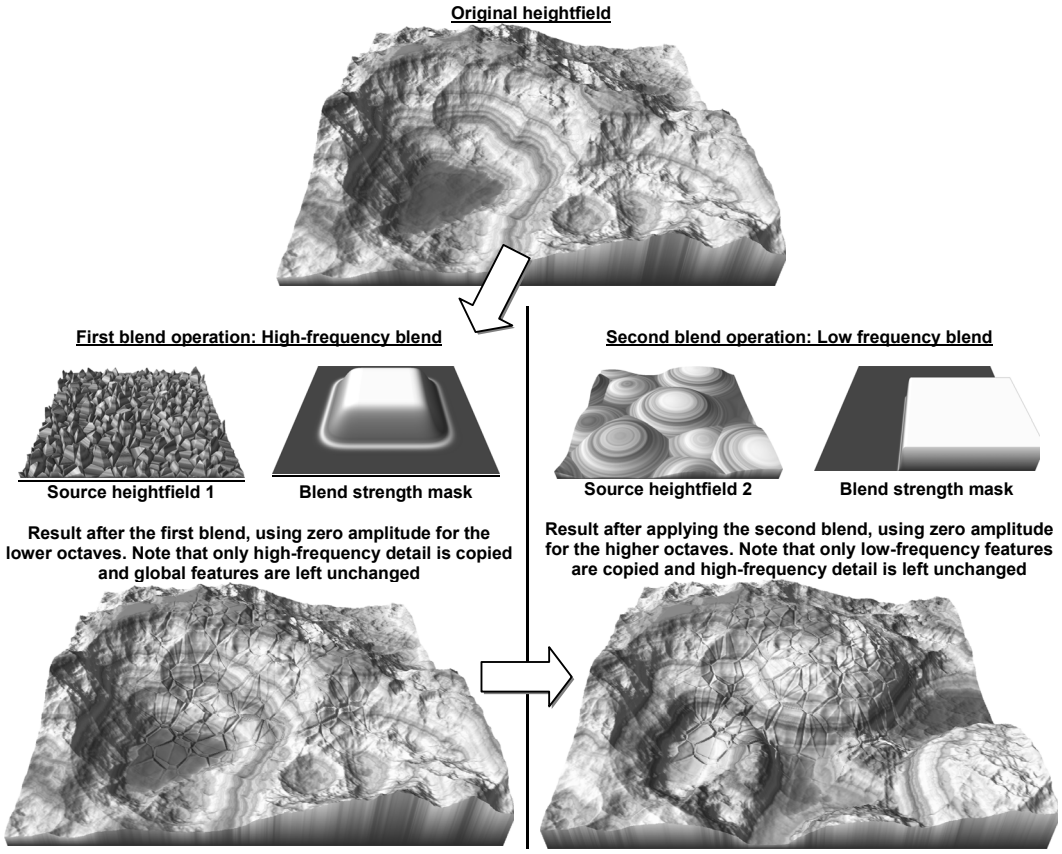


Figure 6-24 **Example of two differently weighted multi-resolution blending operations applied to a heightfield**

6.7 Undo

Although undo functionality is not an editing feature by itself, it is valuable to have during the process of editing. Especially when multiple undo and redo levels are made available, the user of such an editing application can experiment more freely with ideas and settings.

To undo an executed operation, the effect of the operation will need to be reversed. Likewise, to redo an operation, the effect must reappear exactly as it was before. This requires storing data structures for each operation that allow these two actions to be performed. The most compact way would be to define a functional inverse for each offered editing operation. Then redoing would consist of reapplying the operation with the same input parameters, while undoing would apply the inverse operation with these parameters. Sadly, not all operations have a true inverse function. Even simple push/pull brushes that would seem to be easily inversed (subtract some value instead of adding it) can cause problems in practice. For example, heightfields are typically clamped between some minimum and maximum value, internally represented as pure white and black, respectively. This means that all operations will need to clamp the output of an operation to this range. Therefore, even a simple add, followed by a range clamp is not guaranteed to have a one-on-one mapping and can therefore not always be reversed.

An alternative that is guaranteed to work would consist of simply explicitly storing all heightfield versions that were created up to the last operation. This would make it trivial to undo and redo an operation. However, naively keeping a copy of the full heightfield in memory after each operation has been executed would soon saturate all available memory. To limit memory consumption, all but the current heightfield version could be compressed. As most operations would not modify the complete heightfield, is makes sense to store only the areas that changed. For reasons explained in Chapter 7, it is beneficial to partition a heightfield into regularly-sized pages. Operations can then limit calculations to the affected pages. Consequently, heightfield pages that are not touched by an operation do not need to be stored to be undoable. Furthermore, pages that have been changed can be stored more efficiently by calculating the fore-and-after difference for each affected page, which could then be compressed by some lossless compression algorithm. A fast but effective algorithm was found to be the run-length encoding (RLE)

scheme. The above has been implemented in the testbed and easily reached compression ratios of 1:100 for typical editing operations like brushing on a large terrain.

6.8 Evaluation

Many different editing techniques found in current level and terrain editing/generation applications have been discussed in this section. On one hand, level editors often provide simple, low-level brushes, that offer much user control but also require much work and expertise to create natural-looking terrain. On the other hand, terrain synthesis applications offer simulation and procedural generation tools that can algorithmically generate one large terrain, only offering a limited number of algorithms to choose from and a number of global parameters for the user to tweak.

In an attempt to offer the best of both worlds, high-level brushes are suggested, that offer the control of brushes but allow creation of complex terrain like the global synthesis techniques. Another advantage of this mixing of options is that the time required to calculate the output of procedural synthesis or simulation techniques might decrease, as the user might often only brush over (and thus require evaluation of) relatively small areas. Support for simple brush strokes, together with user-editable settings and presets, provide a recognizable interface to users of Photoshop and other image editing applications. Brush-based range and domain mapping might assist designers in creating natural effects in an efficient way. These brush tools would all be fairly intuitive as their effect should be directly visible. Also, the types of parameters are intuitive and could be made consistent, or at least be loadable as user-created presets.

Offering the possibility to use Photoshop-like effect layers could further enhance the potential of any terrain editing tool. This would allow more order-independent creation of the terrain, thus better supporting the iterative design process described in Chapter 3. For example, sections of the terrain could be placed in different layers, letting the user translate or rotate the content of these layers separately. This way, mountain ranges could be moved in an instance, without destroying any detail or undoing any work.

This chapter has discussed many ideas and techniques. Creating an editor that offers all these different options would be a great asset. If, in addition, such an level editor

would also directly communicate or be integrated with the editor that is used for any other assets in a game level, the workflow and efficiency of a level designer would be greatly improved. But, as discussed in Chapter 3, for a (brush-based) iterative design system to be successful, performance is critical. A brush-based terrain editor with a large toolset could only be used to its full potential if and only if the brushes' influence can be calculated fast enough to make it function at interactive rates. As current procedural synthesis tools can take many minutes to generate a heightfield of reasonable resolution, this will require a flexible, but very fast pipeline. Even though a brush stroke might only affect a relatively small area of a terrain, a naive implementation of the discussed procedural algorithms would have difficulty with producing results at interactive rates. To improve this situation, either the algorithms themselves or the (hardware) efficiency of algorithms' implementations could be optimized. Unfortunately, it is often impossible to optimize the algorithms any further. However, a relatively new source of available computational power on PCs might be used to improve the efficiency of algorithm implementations: the graphics processor. The remainder of this dissertation is dedicated to this goal, exploring and explaining this idea, setting up a pipeline and implementing several brush-based algorithms as different test cases.

7 Parallel Processing

Editing algorithms designed for heightfield usually change many height samples per update. Although most editing algorithms would be quite simple, the total number of times these algorithms must be evaluated can be vast. As updates are viewed preferable in real-time or, at least, at interactive rates, it is important to get the most out of the available hardware. This is especially true for operations that have user-interactive brush strokes as their input, as this type of user input is most powerful and user friendly when feedback is instant during brush stroke activity. However, current applications have great difficulty in achieving this for larger areas. Procedural generation of a large terrain can easily take minutes to finish for complex algorithms. Furthermore, applications that support brush-based editing are typically only practical when small brushes are used, as this limits the amount of data to be modified in real-time. As today's personal computers are increasingly more capable of executing calculations in parallel through multi-core CPUs and graphics cards' GPUs, it makes sense to try to parallelize heightfield operations. This chapter discusses different technologies that are assumed to be available on target computers, and discusses the details of the heightfield operation pipeline that has been implemented in the testbed.

7.1 Multi-core CPU

Nowadays, many newer workstation PCs come with a double-core or even quad-core CPU. To get the most out of these systems, tasks must be threaded to be able to divide the load over the available cores. Threading an application often requires use of a threading library, substantial software design planning and, preferably, much knowledge and experience in the field of parallelism. Creating thread-safe, efficient applications has proved to be quite difficult in practice. Bugs are difficult to detect and reproduce, debugging is complicated and bottlenecks are harder to reduce. Even so, applications that require performance are forced to follow this trend, as multi-core systems are probably here to stay. Intel offers dual-core and quad-core technology in their Intel Core 2 and Intel Xeon product ranges. AMD offers dual-core and quad-core AMD Athlons and AMD Opterons.

Several software libraries and compiler extensions are available that aid in the creation of multi-threaded code. For example, OpenMP and Intel's Threading Building Blocks are multi-platform shared-memory extensions. Special language directives and routines assist the compiler in efficiently splitting low-level similar tasks (e.g. loop constructs) over the available processing units and offers typical parallel control structures like critical sections and barriers. This type of library is most valuable when similar or identical tasks must be executed for large amounts of data, leveraging data parallelism. Alternatively, libraries like MPI, POSIX Threads and Windows Threads offer control at a different level, by letting the caller create, pool, synchronize and destroy individual threads which may or may not be executed on the same computer, each with a separate task or routine. For example, a computer game could have separate threads for rendering, character behavior, route planning, physics simulations and resource loading. This type of parallelism is called task parallelism and is typically more bug-prone and requires more expertise. The many possible, less synchronized interactions between (many) different tasks typically makes task parallelism more difficult to comprehend, foresee and test than data parallelism.

Heightfield operations typically consist of doing the same calculations for large amounts of data. Executing these algorithms in parallel by assigning a different piece of affected terrain to each available core seems promising, as this would reduce the largest bottleneck in a typical terrain editor. This is a typical example of data parallelism, so libraries like OpenMP would best fit this type of application.

7.2 Graphics Programming Unit

Another possibility to increase performance on PCs is to offload calculations to the graphics card. Although typically only used for rendering, the graphics card can also be used for purposes other than rendering graphics due to recent advancements in power and flexibility. While the graphics pipeline used to be fixed in functionality, the support for pixel and vertex shaders opened the doorway to executing small, custom programs instead. Attracted by the theoretical peak performance and the high performance-per-dollar, both scientists and game programmers have found new ways to apply this computing power to new domains. Using the graphics cards' processors (i.e. GPUs) for purposes other than 3D scene rendering is often referred to as General-Purpose GPU (i.e.

GPGPU). Applications include fluid simulation, collision detection, molecule folding simulation and general-purpose sparse and dense matrix solvers, to name a few. See *http://www.gpgpu.org* for more examples and details. Another good reference is [PHAR05].

Even though multi-core CPUs become faster each year, the theoretical maximum computing power and memory throughput of graphics cards that can be found in today's off-the-shelf workstation PCs can be much higher than those of CPUs. To compare peak performance of three PCs of potential target users, see Table 7-1. The performance is measured in billion floating point operations per second (GFLOPS). For the high-end range of target PCs, the GPU can be well over one magnitude more powerful than the CPU. Note that the CPU/GPU comparison is not entirely fair, as the CPU results are actual synthetic peak measurements [SISOFT], while the GPU values are theoretical peak performances [COMPNV].

	CPU	GPU
PC1	Intel Pentium 4 530. 8.6 GFLOPS, 3.5 GB/s 2 MB L2 cache @ 7.3GB/s	NVIDIA 6600GT 4.0 GFLOPS, 16.0 GB/s
PC2	Intel Core 2 Duo Mobile T7200 12.6 GFLOPS, 3.5 GB/s, 4 MB L2 cache @ 25.5 GB/s	NVIDIA 7950 GTX 27.6 GFLOPS, 44.8 GB/s
PC3	Intel Core 2 Duo E6420 13.0 GFLOPS, 4.4 GB/s, 4 MB L2 cache @ 18.0 GB/s	NVIDIA 8800 GTS 624.0 GFLOPS, 64 GB/s

Table 7-1 **Target PC CPU and GPU specifications**

Another important factor in execution speed is memory access. As can be seen in Table 7-1 as well, the bandwidth between the GPU and video memory is considerably larger than the bandwidth between CPU and main memory. This high bandwidth is typically needed for the many texture reads per second when rendering detailed 3D scenes. CPUs generally have slower main memory access but have larger memory (L2) caches than GPUs, resulting in faster access if (and only if) the amount of processed data is small.

Current GPUs have the disadvantage of supporting only a limited set of instructions and are only capable of running algorithms that follow the stream programming paradigm. This is typically flexible enough for most graphics rasterization purposes, but limits the types of other algorithms they can execute efficiently. Stream processing is a simple model that allows the execution of a kernel in parallel over an input data stream to output a new data stream. The available computational units can apply the kernel independently from each other. For this reason, no explicit synchronization and

communication between units is required or supported in this model. Mapping this paradigm to GPUs is discussed in Section 7.3.

Processing heightfields is data-intensive. Typical heightfield operations apply very similar instructions to thousands or millions of height samples. This makes the large bandwidth available to the GPU very attractive, while the typically simple operations fit the streaming model quite naturally. As heightfield operation speed is a very important factor for the efficiency of a terrain editor user, exploring the possibilities to speed up operations by (partly) executing them on the GPU has become a large part of the research done for this dissertation.

7.3 GPU as Stream Processor

As GPUs were designed to render graphics, more general-purpose algorithm concepts will need to be mapped to supported GPU concepts and set up as rendering operations. Although the process of 'rendering' calculations sounds unusual at first, the translation of concepts is quite straightforward once one gets used to the idea. As hardware-accelerated graphics are rasterized to the video framebuffer or to off-screen textures, the output of any calculations must be made to fit in these textures. To map the straightforward concept of stream processing to the GPU, the GPU is set up to render 2D rectangles to an off-screen buffer (typically, an RTT texture in DirectX or an OpenGL FBO/PBO). The value (normally, a color) of each pixel in the rendered 2D rectangle can then be calculated by the GPU using custom pixel shader program. This pixel shader defines the actual operation, or stream kernel, and can be made to read from any input textures. The rendered rectangle represents the output stream. Likewise, one or more input textures can be bound as input streams. The input textures can be sampled at different locations within the same pixel shader program if desired. Note that, in this model, each pixel in the output rectangle can only depend on its input textures (and constants) but not on other values calculated as intermediate results during the rendering of other (e.g. neighboring) pixels.

Streaming model	<->	GPU
Input/Output Stream	<->	Input/Render Texture
Vectors	<->	1D texture or constant
Matrices	<->	2D texture or constants
Kernel	<->	Pixel shader
Computation	<->	Rasterization

Table 7-2 **Mapping the streaming model to the GPU**

The supported input and output texture formats are dependent on the vendor-specific hardware capabilities. Also, hardware-accelerated functionality like automatic mipmapping, bilinear and trilinear texture filtering and output blend modes might or might not be available, depending on the hardware capabilities and used texture format. This is one of the drawbacks of using current graphics hardware, as it might be necessary to create different optimal implementations of the desired algorithm to run on a wider range of hardware. Alternatively, a conservative implementation could be created to run on a wide range of hardware but might not use the available graphics hardware to their full potential. Not all algorithms can be run on GPUs. Limitations like shader code length, number of input textures and floating point precision have a considerable effect on the range of algorithms than can be executed on the GPU. Fortunately, with each new standard in shader models, the (minimally) supported list of features grows. Consequently, more complex algorithms can only be implemented (efficiently) with the latest shader models.

In today's games, rendered 3D scenes look best when rendered with Microsoft DirectX's Shader Model 3.0 or even 4.0. Similar shader standards are defined for the OpenGL API. These models define both optionally and minimally available hardware capabilities. An example of a minimally available capability is the supported instruction set used for the pixel and vertex shaders. Most modern computers support at least Shader Model 3.0. But to be able to run these games on older, low-end computers, separate code paths are usually written that support earlier and more limited shader models, at the cost of visual fidelity. As the computers that would potentially be used by level designers are not expected to be low-end, it is assumed that there will be support for at least Shader Model 3.0. This defines a sort of lower bound on the actual hardware capabilities that can be expected to be present.

7.3.1 Shader Languages

The GPU accepts small programs as pixel and vertex shaders. The shaders can be written in assembler code. However, several programming languages exist that are able to compile a more higher-level shader language to native machine code, simplifying shader program writing. DirectX natively offers the High-Level Shading Language (HLSL). OpenGL has its OpenGL Shading Language (GLSL). Both have defined multiple

models versions, with each newer version capable of expressing and compiling more complex programs. NVIDIA created a standard of its own called Cg, capable of compiling to different versions of both HLSL and GLSL. This simplifies the writing of shader programs on different platforms. These three languages are similar in language constructs and expression power. As these languages are designed to access low-level GPU constructs and the latest hardware capabilities directly, the generated machine code approaches the efficiency of manually written assembler code. Still, to use one of these languages to create a GPGPU application requires knowledge of graphics rendering, a manual setup of a GPGPU pipeline and resource management.

There are some even higher-level languages, offering a programming model that is more abstracted from the actual GPU hardware. Examples are Sh (*http://libsh.org/*) and BrookGPU (*http://graphics.stanford.edu/projects/brookgpu/*). Sh code is embedded inside native (CPU) C++ code. It uses meta programming, staged compiling and C++ stream models to offer an easy to use stream programming model that is easy to mix with standard C++ code. Although originally released under the LGPL license, newer versions are only released under a commercial license and go by the name of RapidMind Development Platform. Alternatively, BrookGPU is a stream programming language that comes with a separate Brook compiler that can generate C code from Brook stream programs. Both Sh and BrookGPU languages take care of the translation of stream operations to GPU shaders, the resource management and synchronization. This greatly simplifies stream programming on the GPU, at the cost of losing some efficiency and performance transparency.

The latest in GPU programming languages is NVIDIA's CUDA. Somewhere between the levels of the languages mentioned above, it offers direct access to the hardware architecture and capabilities of CUDA-capable graphics cards. CUDA offers an extended C language to create stream programs with, stream debugging capabilities, common stream operations and a stream math library. Although very promising, its use is currently limited to the high-end NVIDIA GeForce 8 and 9 series graphics cards.

When the GPU would be used to both execute as heightfield operations and render the heightfield in 3D, the resources must be shared between rendering and non-rendering tasks. To have the most control over resource sharing and resource management, the

higher-level languages like Sh and BrookGPU would be difficult to integrate. CUDA is also not an option, as not all target PCs are equipped with the required high-end NVIDIA graphics cards. As the Cg language is largely platform-independent and was already integrated with the testbed that has been based around the Ogre graphics engine, this language is the target language for all GPGPU efforts discussed in this dissertation. Consequently, a Cg GPGPU pipeline had to be set up in the testbed for this purpose.

7.4 GPU Pipeline

As an input stream must be translated to a texture for the GPU to be able to function as a stream processor, heightfields themselves must be translated to textures. As explained in Section 2.1, heightfields can be represented as grayscale images, with white being the maximum height and black being the minimum height. The obvious way of translating the concept of a heightfield to textures would be to create one large 2D texture and fill this with the height sample values. Sadly, current hardware does not allow the creation of arbitrarily large textures. Most of today's ATI and NVIDIA graphics cards have an upper bound on their supported texture size of 4096 x 4096 under the assumed DirectX 9 Shader Model 3.0. This means that heightfields larger than this size could not be stored in one texture.

Instead of representing the heightfield as one texture, the heightfield can be partitioned regularly into many smaller textures, called pages. As heightfield operations typically only change a subsection of the terrain that is currently worked on (e.g. brushed), only the parts of the heightfield that would be affected should minimally be present in video memory before the GPU can be used to execute an operation. Other heightfield pages do not necessarily have to be present. This leads to the idea of having many smaller textures that can be uploaded individually to the video memory when required, managed by a dedicated texture resource manager. This would also increase performance during shader execution, as current graphics hardware accesses smaller textures faster. As the amount of video memory is typically more limited than main memory and must be shared with any buffers required for rendering (Section 5.2.2), the full heightfield is stored in main memory. The video memory serves as a cache of recently used and to use textures. When the next heightfield operation requires a texture that is currently not available in video memory, it is uploaded. Likewise, updated pages are

copied back to the main memory when they are needed immediately for CPU operations, rendering or before texture eviction. Uploading and downloading pages is achieved by transferring the texture data over the graphics data bus. The AGP bus that was widely used until a few years ago would have created a serious bottleneck, as only 266 MB/s of data could be transferred from graphics card back to main memory. The newer de facto standard PCI Express x16 bus alleviates this bottleneck by supporting up to 4 GB/s in both directions [ATIP04]. Still, uploading and downloading data to video memory is relatively slow and, so, it is better to cache the heightfield textures in video memory for as long as possible. Only when available video memory becomes scarce, the least-recently used (LRU) texture is copied back to main memory, if not already done, and evicted from video memory. To this end, a custom memory manager could be written, but this functionality comes standard with DirectX's 'managed' textures. There, so-called shadow copies of the managed textures are kept in main memory at all times and are kept in sync with their respective video memory copy. OpenGL supports similar, but more driver-specific functionality.

Reading and writing simultaneously from/to the same texture is not supported on current graphics hardware. So, the input and output stream must differ for any stream operation. Also, DirectX managed textures cannot be written to directly. Only special textures called Render-To-Texture (RTT) or render target textures can be written to with a pixel shader. OpenGL offers framebuffer objects (FBO) and (the older and less efficient) pixelbuffer object (PBO) for this purpose. Consequently, when a page texture needs to be updated, the original page is set as input to the operation pixel shader which is then used to render a full rectangle to some RTT texture of the same size as the page. After the rectangle has been rendered, the result in the RTT texture can be copied over the original page texture again. As texture creation takes (some) time and pages are typically of identical size, it is better to reuse RTT textures between different operations or even between any different serially processed pages within the same operation.

The data flow is depicted in Figure 7-1. There, step 1 is executed when an operation is about to be executed, but the required input texture isn't yet available in video memory. This step is often not required, because when multiple operations are executed within the same area, the chances are that the required input textures are already cached in video memory. For step two, one or more available RTT texture are selected and used as output

stream container(s). After rendering, the output must be copied back to the managed page texture(s) (step 3). When the page is updated, a page's main memory copy is updated (step 4) when the data is needed on the CPU (e.g. for rendering of CPU operations) or when the video memory texture is about to be evicted when other textures need to be uploaded (step 1) and available memory is scarce.

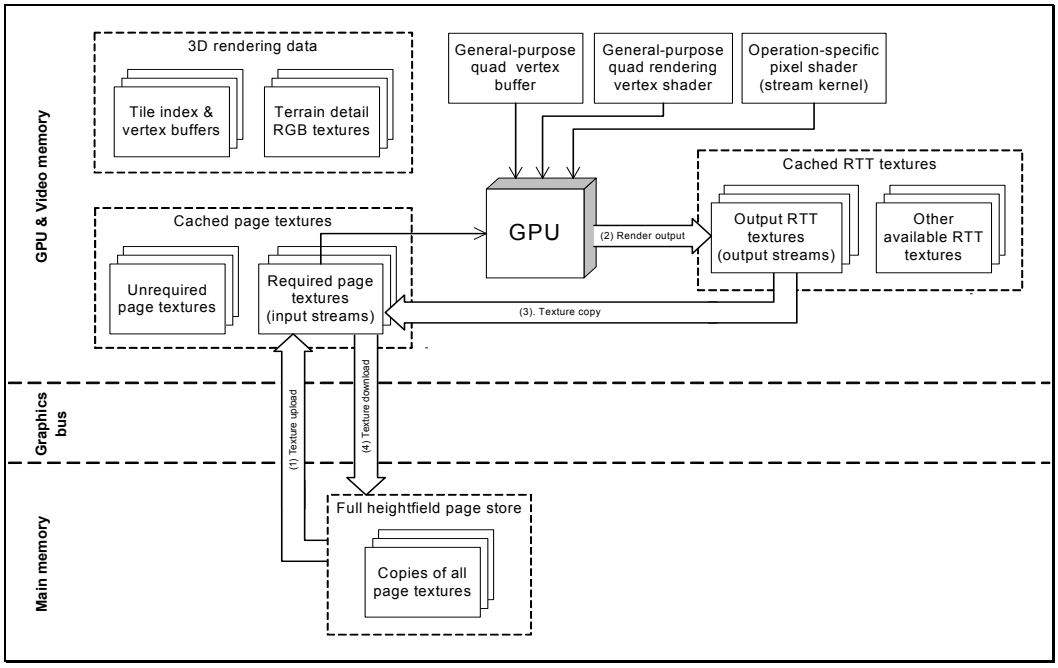


Figure 7-1 **Data flow diagram for the GPU operation pipeline running a single stream kernel**

This simple streaming model does not allow for any sharing of intermediate calculated data between different (e.g. neighboring) height samples. To support more complex operations that would require these interactions, this simple model can be split into a model that executes multiple, (serially) cascaded kernels. Then, intermediate result streams can be rendered and used as input to the next kernel in the cascade. This can also be used to execute iterative processes or split a kernel into a shared common part and an operation-specific part. To support cascaded kernels, step 1 and 2 in Figure 7-1 would be executed as normal for the first kernel in the cascade. As RTT textures can both be written to and read from (although not simultaneously), the RTT output texture filled by the first kernel can serve directly as input to the second kernel. Obviously, a different RTT output texture is required to render the output of the second kernel to. To extend this idea to cascades of more than two kernels, one could use one RTT texture per kernel. However, as each render operation is completed before the next is started, it is also

possible to use only two RTT textures. Then, one RTT is used to read from and the other is used to write to. After the render operation for some kernel has been completed, the role of the RTT textures is reversed and the next kernel in the cascade is executed and so on. This process is often referred to as buffer ping-ponging and is, obviously, more memory friendly. After the last kernel has been executed, the RTT texture that was last written to is copied back to the page texture according to step 3 and 4 in Figure 7-1.

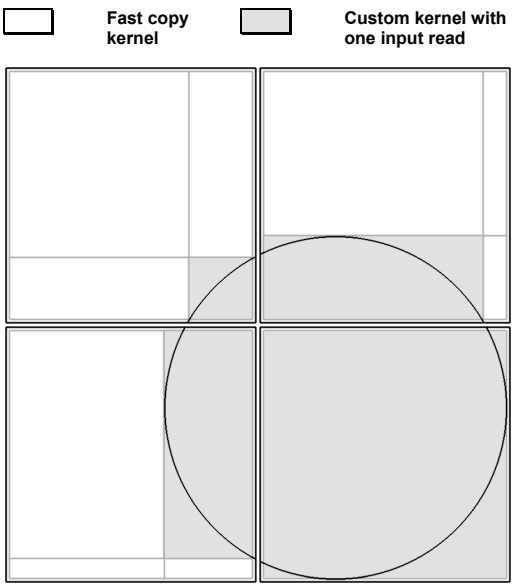


Figure 7-2 **Applying a circle-bound operation with a simple kernel, only reading its old value**

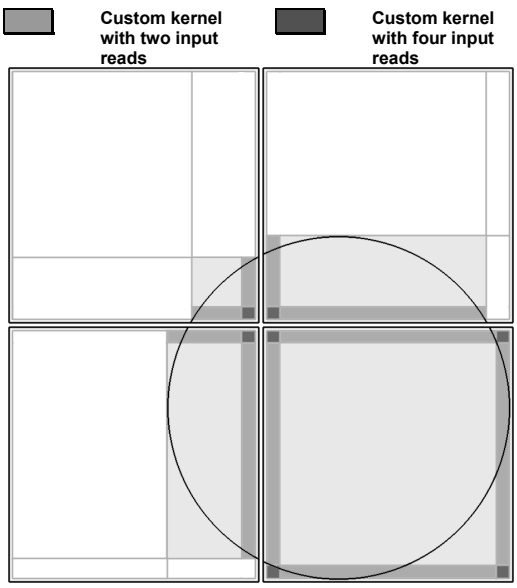


Figure 7-3 **Applying a circle-bound operation with a kernel that requires the old values of neighboring positions as well**

7.4.1 Render Rectangles

A local heightfield editing operation might only overlap partially with some page. For example, in Figure 7-2 and Figure 7-3, the four pages (the black-edged squares) are overlapped only partially by the circular (brush) editing operation. Imagine some 'blob' operation kernel that reads back each previous height value and adds a small value to this, based on the proximity of each height sample to the circle's center. Height samples on and outside the circle are left unchanged by this kernel. Then, the bounding rectangle of the circular area can easily be used to select which heightfield pages will be affected by this operation. These pages would then undergo the steps described in Figure 7-1, each rendering and updating a full page update using the kernel. This kernel could simply be

used to render the affected pages as a whole, as the influence of this specific kernel outside the circular area would be reduced to zero, thus simply copying the old values.

However, executing this circular operation kernel for the unaffected areas inside a page is typically more computationally intensive than simply copying these areas directly into the RTT texture using a fast copy kernel. As an optimization, the render rectangle used to render the updated page to the RTT texture can be split into multiple non-overlapping rectangles, together covering the full page seamlessly. Then, each of these rectangles can be rendered independently using any selected pixel shader. By taking the bounding rectangle of the total area that will be changed by some operation and intersecting this rectangle with some page's own full bounding rectangle, the affected rectangular subregion per page is calculated. This subregion forms the render rectangle that will be executed with the circular operation's kernel for some affected page. The page's full rectangle minus the affected subregion can be split into eight more rectangles: one per Moore neighbor (see Figure 6-7). When the affected subregion touches one or more edges of the page's own full rectangle, one or more of these eight rectangles will be degenerate (i.e. have zero area) and can be ignored. The non-degenerate rectangles can then be used as render rectangles with a fast, dedicated copy kernel to copy the regions of a page that are unaffected by the circular operation to the RTT texture. This is visualized in Figure 7-2. There, the circular brush touching the four pages will result in a total of four (shaded) render rectangles that will use the actual circular operation and nine non-degenerate (white) render rectangles that will use the dedicated copy kernel.

Special care is needed when some operation not only requires the previous height value of each sample, but also requires the previous values in its neighborhood. An example of such an operation is a blurring filter, limited to some circular shape. The problem with retrieving neighboring samples is that not all neighbors will necessarily lie within the same page. More precisely, when values must be read that are maximally K samples away in a $N \times N$ page with $2K < N$, the $(N - 2K) \times (N - 2K)$ samples that lie at least K samples from the page's edges will find all required values inside the same page. However, the samples that are less than K samples away from a page's border will miss some values in their $(2K+1) \times (2K+1)$ neighborhood. These K-border samples will need complementary lookups from neighboring pages to lookup their missing values. For

example, all *K*-border samples in Figure 7-3 are shaded in a darker grey, requiring two or more pages to read from.

Heightfield sample lookups must effectively be implemented as texture reads in the GPU pipeline model. Hence, the lookups for the *K*-border samples/pixels must be implemented as reads from multiple page textures. One way of implementing this would be to let the kernel's pixel shader conditionally determine which texture to use for each of the up to $(2K+1) \times (2K+1)$ sample reads per rendered *K*-border output sample. This, however, would be very slow on current hardware, as conditional branching is ill-supported. Although some GPUs can perform branching quite efficiently, many (namely NVIDIA) GPUs still perform branching by executing all code branches and then using the conditions to 'blend' between all calculated branch outputs.

To make this process more efficient, the operation's rectangle can be split into smaller rectangles that have similar properties concerning the pages required a the neighborhood lookup. More specifically, the rectangle that specifies a page's affected area (e.g. the shaded areas in Figure 7-2) is intersected with the following nine rectangles that cover the entire page: one $(N - 2K) \times (N - 2K)$ area (at the page's center) and the eight rectangles in the Moore neighborhood that cover the remaining parts of the page (which are *K* samples broad and/or high). Again, one or more of these resulting intersected rectangles can be degenerate. For example, only the lower-right page in Figure 7-3 contains all nine intersected areas as non-degenerate rectangles. Note that the areas that are not affected by the operation are not split. The degenerate rectangles can safely be discarded. The non-degenerate rectangles form areas of different amounts of required input textures. And so, part of the earlier mentioned branching is no longer required in the kernel. For example, the kernel that will be executed on the center rectangle is able to read all its $(2K+1) \times (2K+1)$ samples from one texture, ignoring all boundary conditions. Any rectangles that are formed from the horizontal and vertical Moore neighbors of this central rectangle will need to read from both the current page and exactly one neighboring page, which will be same for the complete rectangle. Any rectangles formed as the diagonal Moore neighbors will read from the current and three neighboring pages. These three cases (central rectangle, horizontal/vertical and diagonal Moore neighbor) are depicted in Figure 7-3 as different shades of grey within a page.

Still, the rectangles that require more than one page texture to look up values in their *(2K+1) x (2K+1)* neighborhood per sample need some way to do this efficiently. Current hardware supports different texture address modes, defining what will be returned when a texel (i.e. texture element) that lies outside the texture is looked up. One of these modes will return a custom color. By using this mode and setting the custom color to black (i.e. zeros), a lookup can be done in all bound input page textures and the summed result will be the height. This is done for every of the *(2K+1) x (2K+1)* sample positions. This is guaranteed to work because adding 'black' to any height value will leave the value unchanged and the areas covered by any input pages are guaranteed to be mutually exclusive and form one seamless heightfield. Note that the central rectangle will consequently look up only the value in the current page, all pixels in the horizontal/vertical Moore neighbor rectangles will sum the result of two lookups and all pixels in the diagonal Moore neighbor rectangles will sum the result of four lookups.

By splitting the affected area in different multi-texture-read regions and simply summing multiple texture reads together, no conditional branching is required, while still limiting the amount of required texture reads. Of course, performance is still best when K is only a small fraction of N. Also, values of K larger than $N/2$ are not supported by the technique described above. Hence, once a maximum of K is chosen, a minimum for N is easily deduced. However, there are a number of other factors that also influence the optimal size of N. Similar to the discussion in Section 5.2.2.3 on render tile size, have a small N increases the number of required render calls, thus increasing overhead. Also, a smaller N will result in relatively more multiple lookups per pixel when $K > 1$. Furthermore, when the choice for N results in pages that are relatively large when compared to typical user operations (e.g. used circular brush radii), larger unaffected areas will need to be rendered to the RTT texture and copied back to the page textures, increasing overhead as well. As the above techniques have been implemented in the testbed, experiments could be conducted and showed $N = 256$ to be a good compromise between the conflicting effects for somewhat smaller and computationally less intensive operations. $N = 512$ seemed to work best for larger and more complex operations.

A last optimization when buffer ping-ponging is used involves updating only the areas that could be different since the last render to an RTT texture. When an operation ping-pongs between RTT textures, only the area that could differ from what already is present

in some output RTT needs to be rendered. This is especially important when some operation is applied that would consist of many small updates within the same area. This optimization needs careful tracking of the contents of each RTT texture and the areas affected by its associated operation. When an RTT texture is found to contain a previous version of the page to be rendered, the page's render rectangles can be limited to (i.e. intersected with) the areas of that previous version that would be different.

The graphics library, driver and GPU work together to queue any render calls and asynchronously execute the first render call that has all its dependencies (e.g. input textures) available. While the GPU is rendering, the CPU is free to do other things. However, when the CPU requests some rendered or copied data from the graphics card while this information is not yet available, the CPU is temporarily stalled. For example, this could happen when the CPU requests an updated page in order to update the rendered 3D geometry, but step 1 through 4 in Figure 7-1 have not yet been fully completed for that page. As stalling would waste valuable CPU cycles, it is better to do something else on the CPU first after an operation render call has been dispatched and the result is needed. The testbed implemented this by queuing all page operations and interleaving one or a few page operation calls with actual 3D geometry render calls, and waiting one frame before the result of any sent page operation calls is requested again. This way, steps 1 through 4 from Figure 7-1 are given one frame the time to finish before being forced by any CPU request. By interleaving scene rendering and operation rendering, the influence of actual terrain editing on the frame rate is automatically limited. However, this might increase the total processing time unnecessarily on high-end machines, as the GPU might not be fully loaded with work at all times. Even so, the benefit of interleaving work and scene rendering on more low-end machines makes it a simple but valuable policy, as users on such machines would benefit greatly from a workable frame rate.

7.4.2 Texture Formats

As the graphics card is traditionally used for rendering color graphics, the standard graphics pipeline has been specialized for this purpose in many ways. For example, most shader instructions can work both on scalars and vectors of up to four components. These vector components are typically used as red, green and blue (and possible alpha) color

components or X, Y and Z (and possibly W) space components. When more components are needed, an array of scalars or vectors can often be used. Hardware support for this functionality is well standardized in the shader models. Also, practically all modern graphics cards that support Shader Model 3.0 can handle mathematical instructions that work on 16-bit (half) and 32-bit (single) precision floating point numbers. However, this level of standardization can currently not be expected for the supported input and output texture formats.

Hardware vendors are free to implement only a subset of the multitude of component and bit depth texture format combinations. Also, support for a texture format can only be offered partially. For example, the supported list of formats that can be rendered to is typically only a subset of the list of formats that can be read from. Furthermore, some formats might not support bilinear or trilinear filtering when read from, or automatic mipmap generation or blending with the previous texel (i.e. texture element) color when written to.

The most well supported format is called R8G8B8A8, offering four 8-bit components. Practically all modern hardware is capable of reading and writing to this format will full filtering and blending support. Other formats like one-, two- and four-component 16-bit integer, 16-bit floating point and 32-bit floating point formats are often also available for reading and (to a lesser extent) writing, but often with limited options for filtering and blending. For more details, see [NVID05] and [PERS07].

Not all formats would be usable for heightfields. As each distinct value in a texture represents a different height level, having too few bits can lead to artifacts. 8-bit integer heightfield will clearly show distinct levels, especially when the dynamic height range in world space is large. 32-bit heightfields are typically stored as single precision floating points normalized between 0 and 1, so only just over 23 bits (the mantissa) would be used effectively. 16-bit integer formats offers a compromise between accuracy and storage requirements. For most applications, the difference between 16-bit integer and 32-bit float heightfields would be unnoticeable, while the integer format would require only half the storage space. Another option is 16-bit half-precision floating point numbers, a format supported by many GPUs, but this would only offer just over 10 bits when using normalized values. Although better than the 8-bit integer format, some iso-level artifacts

can still be observed when this format is used. Consequently, the 16-bit integer format was found to offer the best tradeoff between quality and memory consumption.

The implemented testbed is able to import from and export to different formats at 8, 16 and 32 bit depths. However, only 16-bit integers are used internally for the reason described above, as well as to minimize conversion errors, as this is the format used by most other heightfield applications. Below, the implementation details of the testbed are discussed concerning used texture formats and its required workarounds.

When implementing heightfield operations on the GPU, the edited heightfields are best split into page textures, as described in Section 7.4. The most ideal situation to have 16-bit integer heightfields would be to use single-channel 16-bit integer textures, often referred to as SHORT16_L or SHORT16_R textures. This format is well supported by most modern graphics cards. However, several tests have been conducted with this format, and it has been found that some machines that supposedly did support it silently introduced iso-level artifacts. For example, when a 16-bit integer was read from a texture and written directly back to a new 16-bit integer texture, some height resolution was lost. It is assumed that these machines treated the 16-bit integers internally as 16-bit floats when read and (automatically) normalized between 0.0 and 1.0 during a texture lookup by a pixel shader, resulting in an effective resolution of about 10 bits (the mantissa). Although no documentation was found to support this assumption, it would explain the artifacts. As this format proved to be unreliable, a different format had to be used.

A usable alternative to the 16-bit integer textures would be 32-bit floating point textures, often called FLOAT32_R or FLOAT32_L textures. This format offers plenty of resolution (about 23 bits effectively). Although only 16 bits of resolution are required, having more resolution helps to prevent small accumulation errors due to repetitive rounding in the operations. However, this format would double the memory storage requirements and the used memory bandwidth. Also, it is less widely supported than 8 and 16-bit formats.

The format chosen for the testbed is the widely-supported multi-component format, called R5G6B5. This format has a red, a green and a blue component, totaling to 16-bits per pixel. By packing and unpacking these colors in the pixel shaders, a full 16-bit format

can be emulated. Also, by carefully choosing the way the format is packed on the GPU, the CPU can treat the 16-bit RGB colors simply as 16-bit integers when reading from and writing to the RGB pixels. The R5G6B5 texture components consist of 5-bit, 6-bit and 5-bit unsigned integers for the red, green and blue components, respectively. Even though the packing and unpacking causes overhead in the pixel shaders, is has been found to be roughly as fast as direct use of any available 32-bit floating point formats. This is probably due to the smaller bandwidth requirements. Obviously, the relative overhead of packing and unpacking will decrease for more complex shaders. Hence, operations should be combined in as less shaders/steps as possible. This also has the advantage of minimizing any rounding of intermediate results to the 16-bit storage resolution. So due to the smaller memory footprint, better support for this format and the roughly equal performance, it is preferred over the 32-bit floating point format. Another disadvantage of 32-bit floating points is that bilinear (and trilinear) access to textures and output blending is typically ill supported on current hardware. In contrast, R5G6B5 filtering and blending is generally supported. However, the packing algorithm renders this functionality useless. So this functionality must be emulated either way. For example, when bilinearly filtered texture access is desired, it must be emulated by blending 2 x 2 nearest-sample (unpacked) lookups together.

One problem with implementing bit packing on the GPU is the lack of integer support and bitwise operations. When the shader samples a texture, it will automatically rescale its values to exactly fit the floating point range [0.0, 1.0]. This means that the three components in a R5G6B5 texel are internally divided by 31, 63 and 31, respectively. Likewise, when any color is written to this format, the floating point color components are rescaled and rounded to fit in the 5-bit, 6-bit and 5-bit integer range again, respectively. This automatic scaling can result in inaccurate results when not accounted for. For the testbed, several packing and unpacking routines have been designed and implemented. The code in Figure 7-4 proved to be the most reliable under different hardware configurations. These unpack and pack routine is called by all operation pixel shaders every time a texture is read or written to, respectively. Components are laid out to match the little-endian CPU 16-bit unsigned integer format for transparent 16-bit integer CPU access. Note the floor(x) (i.e. round to largest integer smaller or equal to x) and round(x) (i.e. round to integer nearest to x) instructions. These were found to be

required to counter any input and output scaling inaccuracies. frac(*x*) calculates *x* – floor(*x*) and dot(*u*, *v*) calculates the dot product of two vectors.

```
float unpackFromR5G6B5 ( float3 pck ) {
    const float3 scales = { 8.0f, 1.0f / 8.0f, 1.0f / 256.0f };
    const float3 maxs   = { 31.0f, 63.0f, 31.0f };
    return dot ( round ( pck * maxs ), scales ) / 256.0f;
}

float3 packToR5G6B5 ( float value ) {
    const float3 scales = { 1.0f, 32.0f, 2048.0f };
    const float3 ranges = { 32.0f, 64.0f, 32.0f };
    const float3 maxs   = { 31.0f, 63.0f, 31.0f };
    return floor ( frac ( value * scales ) * ranges ) / maxs;
}
```

Figure 7-4 **R5G6B5 packing and unpacking Cg routines**

This concludes the general considerations and specific implementation details for the (GP)GPU pipeline. In Chapter 8, a number of specific kernels/pixel shaders are discussed that were implemented in the testbed to produce a range of different brushes for the user to work with.

8 GPU Editing

To test the potential benefit of today's GPU for the use in heightfield editing, several heightfield editing brushes/algorithms discussed in Chapter 6 are implemented on the GPU using the pipeline discussed in Chapter 7. Implemented brushes include the simple pull/push brush, a smoothing brush and several noise brushes.

8.1 Brush System

With a brush tool the user can draw a path with the mouse or tablet. The heightfield areas covered by the brushed path are modified in real-time. When the tablet is used, the stylus' pressure data could be used to influence the strength at different points on the path. To apply the operation, the brush path is discretized into instances. Each brush instance performs an actual terrain modification at its given position and strength on the path within a fixed brush shape (e.g. a circle). In the most basic form, each of these modifications would consist of one GPU operation, as described in the previous section. Obviously, a brush path is better approximated when more brush instances are used.

Some simpler terrain editing applications discretize a path by applying exactly one brush instance at the current mouse position for each frame. Each frame, the mouse position is used to shoot a ray from the camera onto the 3D terrain. Then, this 3D position is used to calculate the 2D position in heightfield sample space. Using exactly one instance per frame has the drawback of being frame rate dependent. While fast PCs might be able to draw continuous strokes with this system, slower PCs might leave large gaps between brush instances. Also, the instance density can change with the complexity of the operation type and the brush radius, as this typically changes the time required to apply a brush instance. See Figure 8-1.

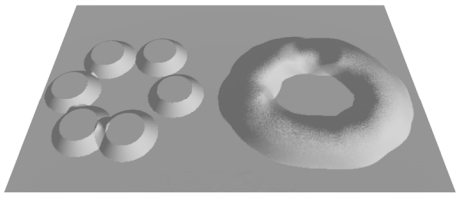


Figure 8-1 **Effect of applying a circular brush as the users follows a circular stroke. Left 'circle': applying one brush instance per frame on a low-end PC. Right 'circle': applying instances discretized from a continuous brush stroke spline**

A more robust and consistent solution is to derive a continuous brush path from the different mouse positions (in sample space) and then create brush instances on this path

with (roughly) equal spacing between them. To this end, the mouse positions are used to form piece-wise cubic Catmull-Rom splines [CATM74]. Then, the length of each spline piece (i.e. the Catmull-Rom spline defined by each set of four subsequent positions) is estimated by evaluating the spline piece at several points and summing the Euclidian distances between these evaluated points. These lengths are then used to calculate the parameter at which the next brush instance should be placed to be roughly some given distance away from the previous brush instance. See Figure 8-1 to compare results from this technique to the simpler one-instance-per-frame technique. Smaller spacing distances lead to more continuous results but take more time to calculate. Also, very small spacing distances can lead to small artifacts due to accumulated rounding errors of the very small changes per brush instance along the path. The testbed allows the users to influence the used spacing distance and combines this number with some simple heuristics based on the current brush radius, falloff and power to calculate the preferred distance.

Both this dissertation and the implemented testbed assume circular brush instances as a test case, but other shapes could be added easily. For example, a user-specified bitmap could be used instead. But for the sake of clarity, only circular brushes are discussed in here. These brushes have an outer radius (simply called the radius), a falloff distance (defined as the difference between the outer radius and some inner radius) and a falloff power. The radius, distances and positions are assumed to be measured in the heightfield's sample space throughout this section. The brush power influences the shape of the falloff area. A power of 1.0 results in a linear falloff ramp, at full strength at the inner radius and faded out completely at the outer radius. The formulas used in the testbed are given below.

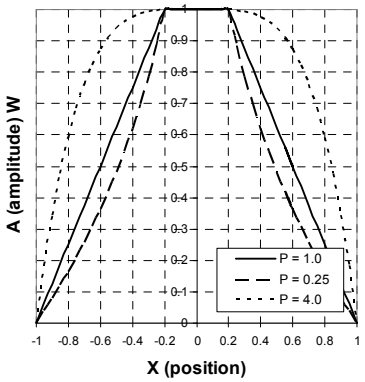


Figure 8-2 **Effect of power P on brush weight for R = 1, F = 0.8, d = 1, C = 0 (1D case)**

$$c_1 = \frac{1}{(R-F)^P - R^P}, \quad c_2 = \frac{P}{2}, \quad c_3 = -c_1 \cdot R^P$$

Eq. 8-1

$$A = d \cdot \min(\max(c_1 \cdot ((X-C) \bullet (X-C))^{C_2} + c_3, 0), 1)$$

In Eq. 8-1, C is (center) position of the brush instance, R is the radius with $R > 0$, F is the falloff with $0 > F > R$, P is the power and d is the maximum amplitude of the brush instance. Then, A calculates the weight or amplitude of the brush instance at any sample position X. See Figure 8-2. The constants c_1, c_2 and c_3 need only to be calculated every time either the radius, falloff or power changes. d is the user-specified brush strength divided by the brush instance spacing. This way, the combined effect strength of the operation is (largely) independent of the brush instance spacing.

The brush system outlined in this section forms the basis for all brushes implemented in the testbed. For the pull brush, the amplitude A is calculated and simply added to the previous height value for all affected samples for each brush instance. For the push brush A is simply subtracted instead of added to the previous height value. More complex brushes would use A as a weight factor to scale the output of other calculations.

Each brush instance results in a heightfield manipulation. When this is implemented naively, these manipulations will be sequentially executed as one operation and thus one render call per affected heightfield page by ping-ponging between two RTT textures (Section 7.4). Consequently, each brush instance in a brush stroke requires a full page to be rendered/copied, incurring considerable overhead. This overhead can be reduced in some cases by batching multiple brush instances together and executing this batch within the same render call. Whether and/or how this is possible depends on the class of the operation algorithm. Three classes are distinguished:

1. The algorithm f is linearly decomposable and the result independent of any order in which brush instances are applied. :

$$H'(x) = f(...(f(f(H, B_1, x), B_2, x)..., B_n, x) = f(H, B_1, x) + f(H, B_2, x) + ... + f(H, B_n, x)$$

2. The algorithm f is dependent on the brush instance order, and only depends on the local previous height value (and thus not on any neighbor (or farther) height values) to calculate each new height value:

$$H'(x) = f(...(f(f(H, B_1, x), B_2, x)..., B_n, x) = f(...(f(f(H(x), B_1), B_2)..., B_n)$$

3. All other algorithms.

Here, H denotes a heightfield as a function of position, x is some 2D position on the heightfield and B_i are brush instances (e.g. position and height) for $1 \leq i \leq n$. Algorithms (or, more specifically, the pixels shaders) for both the first and the second class can be optimized to batch multiple brush instances together. Algorithms of the first class can simply add together the weights (Eq. 8-1) for each of any batched brush instances before this combined weight is passed on atomically to the rest of the algorithm. Implemented brushes of this class include the pull/push brush and several noise brushes. Algorithms of the second class have a non-linear causal dependency. Still, brush instances can be batched and executed together by subsequently calculating the weight A (Eq. 8-1) and applying the change for each of the brush instances in a batch from within the same pixel shader. An example of an algorithm of this class would be a push/pull-like brush of which the added/subtracted (delta) height depends on the previous height value. See Section 8.2. Only algorithms of the third class cannot batch instances together in one call, as the results from the inter-pixel dependencies for each instance would need to be shared within one batched call, which is not supported in the stream programming model. A smoothing (i.e. blurring) brush would typically be of the third class.

Please note that the algorithms of the third class also require a slightly more complex architecture of the heightfield editing pipeline than depicted in Figure 7-1. The combination of the causality between brush instances and the dependencies between neighbors do not only allow instance batching, but also require special handling of the otherwise independent processing of pages. For example, when a single brush instance that covers two pages would cause an update of one page which would then be used as input to the other page, the result of the second page would be affected erroneously. This is because the same heightfield page store from Figure 7-1 is used both to read from and to write to. Therefore, brush instances that cover multiple pages should not write to the full heightfield page store directly, but should write to temporary buffers instead. When all affected pages are fully updated by a brush instance and stored in temporary buffers, these can safely be used to update the full page store.

Experiments have shown that a typical user stroke would require somewhere between 1 and 100 brush instances per page. To batch the brush instances, new brush instances are queued on each frame the user applies a brush stroke. This first-in first-out queue is then used to create batches. As each batch will be executed in a single render call using

one pixel shader, this pixel shader code must support the batch size. The brush instances can be evaluated by the pixel shader in a loop iterating over each instance in the batch. But as dynamic looping can be ill supported for Shader Model 3 graphic cards, the batch size is best made static, allowing the shader compiler to unroll the loop for efficiency. This means that the batch size would have a fixed upper limit, requiring larger queues to be spread over multiple sequential operations. But of course, the queue can also be smaller than the fixed batch capacity. Two different policies can be used to handle this case:

1. Wait for more instances to be queued until the shader's batch capacity is reached.

2. Batch the smaller queue now and set the brush instance weights (*d* on p. 111) of all unused batch slots to zero.

Although computationally more efficient, the first policy would require handling the case where no more brush instances are (likely) to be added in the near future. The second option wastes some computation power of the GPU but would typically finish sooner, as there is no need to wait. The size of the supported batch capacity must be chosen carefully. Smaller batch sizes will have more render and copy overhead. Larger batches might wait longer for more instances or waste more cycles executing dummy slots. After several experiments, the combination of a fixed batch size of 16 brush instances and the second policy was chosen for the testbed, as this proved to be the best compromise between flexibility, speed and overhead.

Another option would be to compile operation shaders for each possible batch size, selecting the shader with the batch capacity closest to the number of directly available queued instances. However, compiling a shader can take a noticeable amount of time, so compiling multiple versions of a shader should preferably be done once for each target machine and then cached between sessions for faster access. This idea has been tried out with automatic compilation of shaders for batches capacities of 1, 4, 16 and 64, but the gain in efficiency compared to the fixed-16 policy was found to be small and did not justify the added complexity to the pipeline and shader design. This method was therefore abandoned.

8.2 Push/Pull Brush

Using the brush system as defined in the previous section, it isn't difficult to create a push/pull brush. The pull brush can simply be implemented in a pixel shader by evaluating and adding the weights (Eq. 8-1) of all brush instances in a batch, doing a texture lookup from the previous page's texture, adding these numbers together and outputting this 'color'. The texture lookup and the color output must be unpacked and packed, respectively, as discussed in Section 7.4.2. The push brush is identical to the pull brush but uses a negative d (i.e. brush strength) in the calculation of each A (Eq. 8-1).

An extension to this simple brush would be to let the user specify a mask bitmap and modulate the brush instance's local strength with this bitmap. This could simply be accomplished by multiplying each calculated A with the (greyscale) color from an associated pixel in this bitmap. Internally, this bitmap would be loaded into video memory as a texture. This texture could also be rotated and scaled by the user, with or without controllable random variation per brush instance. This would require a texture lookup from within the pixel shader at a UV position that is transformed by a matrix (constant) that is assigned per brush instance.

Of course, variations on the weight formula can be tried out to get different results. For example, the previous formula for A (Eq. 8-1) could be modulated by $1-\min(\max((H(X)-C_{up})/R,0),1)$, assuming H, C_{up} and R are in the same units. C_{up} represents the previous height at the center of the brush instance. Calculating this value is best done on the CPU per brush instance and then made available to the shader by shader constants, along with the radius, power and position of each of the batched brush instances. When applying this modulation of A, a brush is created that functions somewhere between push/pulling and leveling (Section 6.1). This simple modification makes it easier to create ledges and walking trails on steep mountains, for example.

8.3 Perlin Noise

A number of noise brushes have been implemented for this dissertation. See Section 6.3 and 6.4 for a more general discussion on implementing noise functions and using them to generate mountainous effects. As base noise function, a seedable 2D Perlin

function has been implemented as a Cg pixel shader. This shader is loosely based on the work described in [GREE05]. See Figure 8-3 for the Cg implementation.

```
float perlinNoise(float2 p, float seed, uniform sampler2D texH, uniform sampler2D
texG)
{
  // Calculate 2D integer coordinates i and fraction f.
  float2 i = floor(p);
  float2 f = p - i;
  // Get weights from the coordinate fraction. Uses the quintic interpolator.
  float2 w = f * f * f * (f * (f * 6 - 15) + 10);
  float4 weights = float4(1, w.x, w.y, w.x * w.y);
  // Get the four randomly permutated indices from the noise lattice nearest to p
  // and offset these numbers with the seed number. The size of the permutation
  // texture is expected to be 256 x 256.
  float4 hash = tex2D(texH, i / 256) + seed / 256;
  // Permutate/hash the four offseted indices again and get the 2D gradient
  // for each of the four permutated coordinates-seed pairs. The size of the
  // permutated gradient texture is expected to be 256 x 256.
  float4 gradientLeft = tex2D(texG, hash.xy) * 2 - 1;
  float4 gradientRight = tex2D(texG, hash.zw) * 2 - 1;
  // Evaluate these four independent lattice gradients at p
  float nLeftTop = dot(gradientLeft.xy, f);
  float nRightTop = dot(gradientRight.xy, f + float2(-1,  0));
  float nLeftBottom = dot(gradientLeft.zw, f + float2( 0, -1));
  float nRightBottom = dot(gradientRight.zw, f + float2(-1, -1));
  // Bi-linearly blend between the gradients, using weights as blend factors.
  float4 grads = float4(nLeftTop, nRightTop - nLeftTop, nLeftBottom - nLeftTop,
                        nLeftTop - nRightTop - nLeftBottom + nRightBottom);
  float n = dot(grads, weights);
  // Return the noise value, normalized in the range [-1, 1]
  return n * 1.530734;
}
```

Figure 8-3 **2D Perlin noise Cg routine**

Two S x S textures are used as input to this routine: a fixed 2D permutation table *texH* and a fixed 2D gradient table *texG*. The S *x* S permutation texture stores four $H(\,(x' + H(y')\,)\ mod\ S\,)$ values, one for $(\lfloor x \rfloor, \lfloor y \rfloor)$, one for $(\lfloor x \rfloor, \lfloor y+1 \rfloor)$, one for $(\lfloor x+1 \rfloor, \lfloor y \rfloor)$ and one for $(\lfloor x+1 \rfloor, \lfloor y+1 \rfloor)$ as the texture's R, G, B and A 8-bit channel values, respectively. The resulting redundancy between color channels and neighboring texture pixels has the advantage of requiring only one texture lookup instead of four to retrieve a

hashed number for four neighboring noise lattice points. For the implementation, S was chosen to be 256. The original permutation table written by Ken Perlin was used to create *texH*. The S *x* S gradient texture *texG* stores ($G(H(i \bmod S) \bmod M); G(H(j \bmod S) \bmod M))$: two independent 2D gradients for the rehashed i and j values. M represents the number of different gradient coordinates, which is 8 for the implementation. Biasing the hashed output of tex$H$ by a *seed* number before it is used as input to *texG* allows the creation of S unique noise textures. The rehashing that is done by *texG* serves to improve the randomness of the output between seed numbers. See page 78 for more information on the tables G and H. Inspired by the improved 3D Perlin noise algorithm [PERL04], all gradients in G are coordinates that are made to lie on the unit circle: $(\cos(\frac{1+2k}{M}\pi);\sin(\frac{1+2k}{M}))$ for $k \in \mathbb{N}_0, k < M$. The angles are chosen to be off-axis to hide the regularity of the grid as much as possible. As S is chosen to be divisible by M, each angle will be represented in *texG* exactly S/M times. As *texG* and *texH* are stored in the R8G8B8A8 format (i.e. a four component 8-bit unsigned integer format), the components are limited to [0, 1], so the gradients' Cartesian 2D coordinates are scaled by 0.5 and then biased by 0.5 to map these coordinates to the allowed range before being stored in *texG*. The coordinates are mapped back to [-1, 1] by the shader code at the same lines *texG* is read in Figure 8-3. *gradientLeft.xy, gradientLeft.zw, gradientRight.xy* and *gradientRight.zw* each represent one unit-circle 2D vector for each of the four points on the noise lattice closest to the input position. These four vectors are interpreted as gradients and are bi-linearly interpolated by a quintic weight function in both directions to evaluate the noise function at *(x, y)* [PERL04]. See Figure 8-4. For a more mathematical treatment, see Section 8.3.3. The average output for this noise function so far is zero and its range is $\pm 0.5 \max_i(abs(G[i]_x) + abs(G[i]_y))$ with $G[i]$ being any of gradients in *texG*. Hence, scaling the noise function by $2.0/(\cos(\pi/8) + \sin(\pi/8)) \approx 1.530734$ will map the function symmetrically to [-1, 1] for the chosen set of gradients.

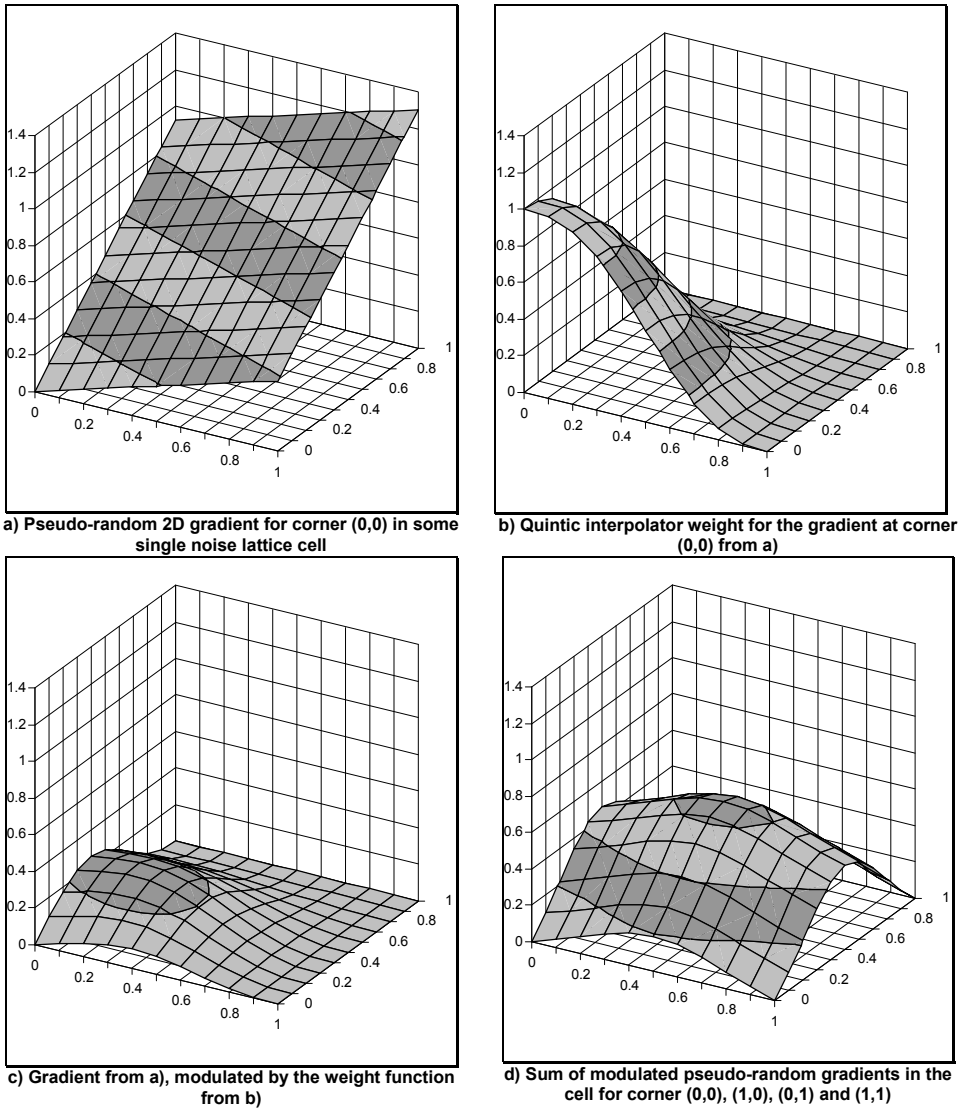


Figure 8-4 **Example of 2D Perlin noise construction. a) – c) Construction steps for one corner. d) Summed results for all four corners in a noise lattice cell**

8.3.1 Basic Turbulence

The Perlin noise function defines a band-limited pseudo-random 2D signal. To create mountain-like features with this function, noise signals with differently scaled coordinates and weights are added together, as described in Section 6.3.4. These summed Perlin noise functions are commonly referred to as a turbulence function. This effectively creates a noise signal with a frequency distribution that is controlled by the individual

noise signal weights and input scales. Each of these noise signals should be as independent from the other signals as possible. For this reason, each of the signals uses its own seed number. For the lacunarity (the relative coordinate scaling between subsequent noise signals), a value close to 2.0 was chosen. Good results were obtained with 1.92. The weights of the differently scaled noise signals can be influenced by the user but are typically smaller for higher-frequency signals.

In addition to summing differently weighed and scaled Perlin noise functions, variations of this function could be used instead. Two already discussed variations on the Perlin noise building block have been implemented: billowy and ridged noise. These noise variations are simply accomplished by summing over **`abs(perlinNoise(…))`** or **`1-abs(perlinNoise(…))`** instead of summing over **`perlinNoise(…)`**, respectively. See Section 6.3.4 and Figure 6-13.

To use one of the turbulence functions as a brush, the output of the turbulence evaluation is modulated with the same formula and system used for the push/pull brush (see Section 8.2). This allows the turbulence function to be painted with using circular brushes, (pseudo-)randomly adding and subtracting heights. The noise seed and the input coordinates to the noise function during the evaluation of a brush instance can be defined in several ways:

1. **User-selected seed number & input coordinates relative to the brush instance's center.** Different noise brush instances will replicate the same noise signal inside the brush area, only translating relatively to each other. The user can change the random noise signal as a whole by changing the seed number.

2. **User-selected seed number & input coordinates in absolute world space.** The noise signal is implicitly defined for the whole terrain, independent of the position of the brush instances. The brush instances only effectively influence the local quantity of noise that is added or subtracted. The user can change the terrain noise signal as a whole by changing the seed number.

3. **Seed number randomly chosen at beginning of each brush stroke & input coordinates in absolute world space**. Because the seed number will change with each stroke, the results are unpredictable. But as a complete stroke will use the same seed number, results of individual brush instances are consistent with each other.

4. **Seed number is randomly chosen for each individual brush instance**. The noise signal is completely random and will be inconsistent between brush instances, making dominant features for a specific noise type stand out less. When brush instances overlap (which is typically the case), it is somewhat like adding many uncorrelated noise signals, together approaching a random signal with the Gaussian distribution, as expected from the central limit theorem.

As a test case, option 2 has been implemented in the testbed. Good and intuitive noise brushes were achieved with this system. However, each of these options would probably have its preferred applications. Hence, it might be best to offer all options to the user and let the user decide which to use for a specific task or effect.

8.3.2 Quilez Noise

A relatively new and unknown procedural noise/turbulence variation has been described in [QUIL08], an article by I. Quilez. It described an algorithm that extends the Perlin turbulence function in an attempt to create less uniform and more natural results. In the article, impressive imagery accompanies the description of a code snippet. However, an implementation of this (pseudo-)code resulted in lesser-quality output. After some experimentation, similar results were achieved, but also showed the described theory to be erroneous. The author claimed to describe the analytic derivative of the Perlin noise function, that could be used to distort the Perlin turbulence (i.e. summed Perlin noise function, see Section 6.3.4) in interesting ways. This proved to be incorrect, as the function the derivative was derived for did not produce Perlin noise, but behaved more like a simple value lattice noise function (see Section 6.4). Even so, when this false derivative was used to distort the turbulence function of summed (correct) Perlin noise signals, convincing results were achieved. For example, see Figure 8-5.

Note the spatial variation in smoothness and detail, which creates a less statistically uniform and more natural landscape.

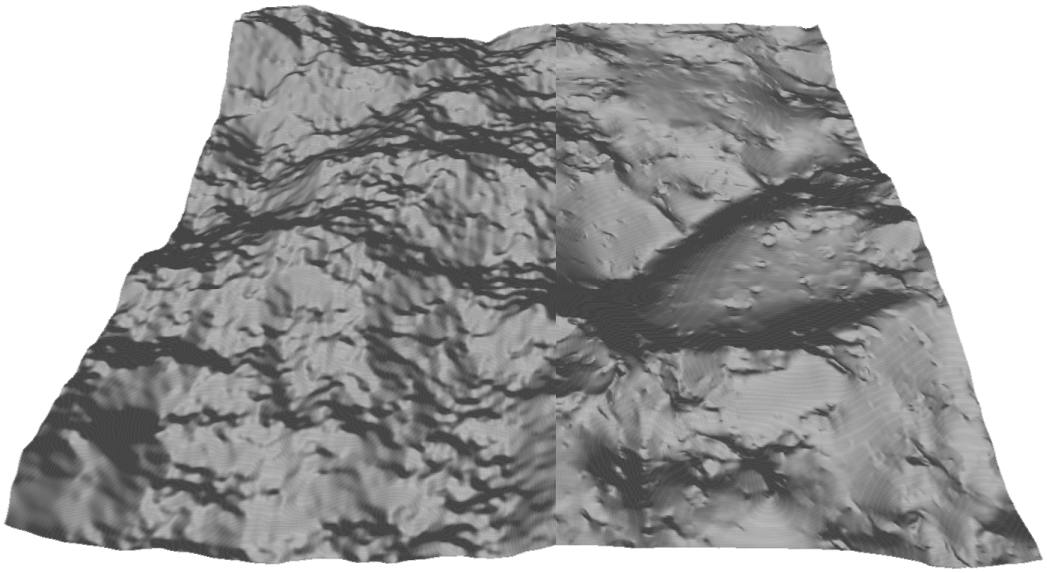


Figure 8-5 **Comparison of standard Perlin turbulence (left) and 'Quilez' turbulence (right) using the same Perlin noise input**

This noise/turbulence variation defines a complex transform $T(n, x, y)$ function:

$$T^{L}_{L_{\min}}(n,x,y) = \frac{n}{\left\| \left(\sum_{l=L_{\min}}^{L} \frac{\partial N}{\partial \lambda^l x}(\lambda^l x, \lambda^l y), \sum_{l=L_{\min}}^{L} \frac{\partial N}{\partial \lambda^l y}(\lambda^l x, \lambda^l y) \right) \right\|^2}$$

which is then used to transform the Perlin noise function $N(x, y)$ before it is summed in the turbulence function:

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

Starting from the coarsest (i.e. L_{min}) noise scale, the T function calculates the squared length of the summed (false) 2D gradients/derivatives of the noise function and divides the noise signal input n by this squared length. Hence, a large gradient in a noise octave L will decrease the noise signals for this noise scale and all subsequent finer noise scales or octaves. Consequently, areas surrounding local peaks and valleys in some noise scale will be relatively smooth compared to the peaks themselves. Combined with the fact that T does not actually use correct derivatives of N but of some other noisy but consistent signal, new patterns emerge.

Similar to the noise and turbulence function discussed in Section 8.3, this function can be modulated with the A function (Eq. 8-1) to limit the noise to the area contained by individual brush instances. Also, seed number and input coordinates can be controlled and determined as described in Section 8.3.

8.3.3 Erosive Noise

Inspired by the idea of using the derivative of the noise function to influence the turbulence function, this section introduces a novel noise warping algorithm that has been designed specifically to fake the effect of fluvial erosion. See Figure 6-1.

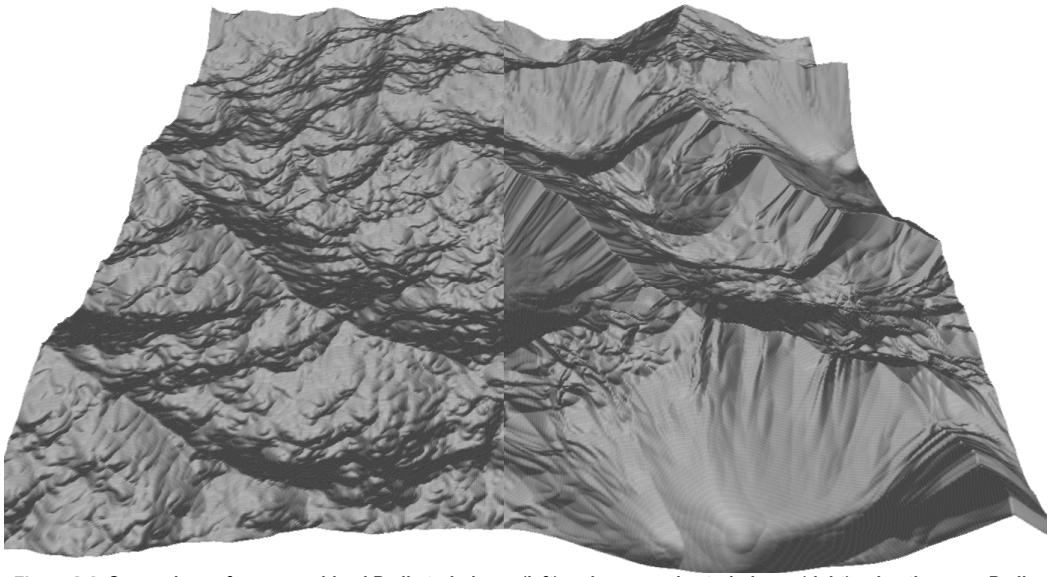


Figure 8-6 **Comparison of common ridged Perlin turbulence (left) and new erosive turbulence (right) using the same Perlin noise input**

As fluvial erosion simulation was found to be quite slow on high resolution heightfields, having a noise function that is able to approximate some of the fluvial features is a great asset.

This algorithm is basically still a Perlin noise-summing turbulence function. However, it does not only define a (range) transform function for each summed noise octave, but it also influences the positions at which the noise signal of the subsequent noise octaves are evaluated (i.e. uses an octave-dependent domain transform).

The 2D derivative over the horizontal plane of the Perlin noise signal, or $\left(\frac{\partial N}{\partial x}(x,y),\frac{\partial N}{\partial y}(x,y)\right)$, can be interpreted as the gradient defining the local steepest up-hill direction. When this gradient is used to offset the input position at which the Perlin noise is evaluated for subsequent octaves in the turbulence functions, noise bumps and dents will become elongated in the locally steepest direction and become compressed near peaks and valleys. This elongation of features somewhat approximates the erosive effect of rainfall, carving out small down-hill gulleys into mountains. Note that noise octaves are expected to be evaluated from coarsest to finest, as before. Only this way will large (mountain) features influence the creation of smaller (gulley) features. In addition to the gradient-based input offsetting, both the range and domain transform are made dependent on the intermediate total height accumulated during coarse-to-fine octave noise evaluation in such a way that valleys will become smoother than peaks.

As the required (correct) derivative of the Perlin noise function is not known to have been described in any literature, this is done here. Note that only the Perlin noise function in two dimensions is considered here. To get the derivative of the Perlin noise signal, the noise algorithm must first be stated as a mathematical function. Defining the noise function as $N(p)$ where $p \in \Re^2$, the following helper definitions and shorthand notations are defined.

$$w(t) = 6t^5 - 15t^4 + 10t^3$$
$$i = (\lfloor p_x \rfloor, \lfloor p_y \rfloor)$$
$$f = p - i$$

$$w_x = w(f_x)$$
$$w_y = w(f_y)$$
$$G_{jk} = G(i_x + j, i_y + k)$$

Note that G_{00}, G_{10}, G_{01}, G_{11} denote the four gradients in the Perlin gradient lattice nearest to p. The hashing required to get a gradient from a position on the noise lattice is assumed to be implicit. For more details, see Section 8.3. The function $N(p)$ can now be defined as follows:

$$N_{left,top} = G_{00} \bullet (f_x, f_y) = G_{00x} f_x + G_{00y} f_y$$
$$N_{right,top} = G_{10} \bullet (f_x - 1, f_y) = G_{10x} f_x + G_{10y} f_y - G_{10x}$$
$$N_{left,bottom} = G_{01} \bullet (f_x, f_y - 1) = G_{01x} f_x + G_{01y} f_y - G_{01y}$$
$$N_{right,bottom} = G_{11} \bullet (f_x - 1, f_y - 1) = G_{11x} f_x + G_{11y} f_y - G_{11x} - G_{11y}$$

$$N_{top} = N_{left,top} + (N_{right,top} - N_{left,top}) w(f_x)$$
$$N_{top} = (G_{00x}) f_x + (G_{00y}) f_y + (-G_{10x}) w_x + (-G_{00x} + G_{10x}) f_x w_x + (-G_{00y} + G_{10y}) f_y w_x$$

$$N_{bottom} = N_{left,bottom} + (N_{right,bottom} - N_{left,bottom}) w(f_x)$$
$$N_{bottom} = -G_{01y} + (G_{01x}) f_x + (G_{01y}) f_y + (G_{01y} - G_{11x} - G_{11y}) w_x + (-G_{01x} + G_{11x}) f_x w_x + (-G_{01y} + G_{11y}) f_y w_x$$

$$N = N_{top} + (N_{bottom} - N_{top}) w(f_y)$$
$$N = \begin{pmatrix} (G_{00x}) f_x + (G_{00y}) f_y + (-G_{10x}) w_x + (-G_{00x} + G_{10x}) f_x w_x + (-G_{00y} + G_{10y}) f_y w_x + \\ -G_{01y} w_y + (-G_{00x} + G_{01x}) f_x w_y + (-G_{00y} + G_{01y}) f_y w_y + (G_{01y} + G_{10x} - G_{11x} - G_{11y}) w_x w_y + \\ (G_{00x} - G_{01x} - G_{10x} + G_{11x}) f_x w_x w_y + (G_{00y} - G_{01y} - G_{10y} + G_{11y}) f_y w_x w_y \end{pmatrix}$$

The derivative of this function can be calculated as follows:

$$\frac{\partial f_x}{\partial x} = \frac{\partial f_y}{\partial y} = 1, \quad \frac{\partial f_y}{\partial x} = \frac{\partial f_x}{\partial y} = 0, \quad \frac{\partial w(f_y)}{\partial x} = \frac{\partial w(f_x)}{\partial y} = 0,$$
$$\frac{\partial w(t)}{\partial t} = 30t^4 - 60t^3 + 30t^2$$
$$\frac{\partial t w(t)}{\partial t} = 36t^5 - 75t^4 + 40t^3$$
$$\frac{\partial N}{\partial x} = \begin{pmatrix} (G_{00x} + (G_{01x} - G_{00x}) w_y) + \\ (-G_{10x} + (G_{10y} - G_{00y}) f_y + (G_{01y} - G_{11x} - G_{11y} + G_{10x} + (G_{00y} - G_{01y} + G_{11y} - G_{10y}) f_y) w_y) \frac{\partial w(f_x)}{\partial x} + \\ (G_{10x} - G_{00x} + (G_{00x} - G_{01x} + G_{11x} - G_{10x}) w_y) \frac{\partial f_x w(f_x)}{\partial x} \end{pmatrix}$$
$$\frac{\partial N}{\partial y} = \begin{pmatrix} (G_{00y} + (G_{10y} - G_{00y}) w_x) + \\ (-G_{01y} + (G_{01x} - G_{00x}) f_x + (G_{10x} - G_{11y} - G_{11x} + G_{01y} + (G_{00x} - G_{10x} + G_{11x} - G_{01x}) f_x) w_x) \frac{\partial w(f_y)}{\partial y} + \\ (G_{01y} - G_{00y} + (G_{00y} - G_{10y} + G_{11y} - G_{01y}) w_x) \frac{\partial f_y w(f_y)}{\partial y} \end{pmatrix}$$

Note that $\frac{\partial w(f_x)}{\partial x} = \frac{\partial w(x)}{\partial x} = \frac{\partial w(t)}{\partial t}$ and $\frac{\partial f_x w(f_x)}{\partial x} = \frac{\partial x w(x)}{\partial x} = \frac{\partial t w(t)}{\partial t}$ for $i_x \le p_x < i_x + 1$. However, due to the way i_x is defined and the gradient lattice is used, this will be true for all i_x and thus all p_x, covering the complete domain. Obviously, this

reasoning also holds for the *y* direction. As explained in Section 8.3, the noise function is best normalized by dividing the evaluated output by the theoretically largest output, which can be determined from the set of used gradients. But when the noise function is scaled by a constant, the derivatives should be scaled by this constant as well.

Now that the derivative has been determined, it can be used to manipulate the turbulence function. As this erosion type is visually most evident in mountainous terrain, the algorithm is based around the ridged Perlin function instead of the standard Perlin noise function. Hence, $N'(x, y) = 1 - \text{abs}(N(x, y))$. Therefore,

$$\frac{\partial N'(x,y)}{\partial x} = \begin{pmatrix} -\frac{\partial N(x,y)}{\partial x} \Big| N(x,y) \geq 0 \\ \frac{\partial N(x,y)}{\partial x} \Big| N(x,y) < 0 \end{pmatrix} \quad \text{and} \quad \frac{\partial N'(x,y)}{\partial y} = \begin{pmatrix} -\frac{\partial N(x,y)}{\partial y} \Big| N(x,y) \geq 0 \\ \frac{\partial N(x,y)}{\partial y} \Big| N(x,y) < 0 \end{pmatrix}$$

This piecewise derivative was found difficult to get good results with. Hence, a rough but continuous approximation to the derivative is recommended in this case instead:

$$\frac{\partial N'(x,y)}{\partial x} \approx -N(x,y)\frac{\partial N(x,y)}{\partial x} \quad \text{and} \quad \frac{\partial N'(x,y)}{\partial y} \approx -N(x,y)\frac{\partial N(x,y)}{\partial y}$$

Because $-1 \leq N(x,y) \leq 1$, the absolute value of the approximation will always be somewhat smaller than the exact solution. Moreover, it will approximate zero at peaks in $N'(x,y)$ instead of returning a discontinuous output.

For the fluvial noise type, the following turbulence-like function was designed:

$$S^L_{L_{\min}}(x,y) = S^{L-1}_{L_{\min}}(x,y) \cdot \beta \min(1, H^{L-1}_{L_{\min}}(x,y)) \quad \text{for L>L}_{\min},\ S^L_{L_{\min}}(x,y) = 1 \text{ otherwise}$$

$$D^L_{L_{\min}}(x,y) = \sum_{l=L_{\min}}^{L} S^l_{L_{\min}} w^l \cdot -N(x,y)\left(\frac{\partial N(x,y)}{\partial x}, \frac{\partial N(x,y)}{\partial y}\right)$$

$$H^{L_{\max}}_{L_{\min}}(x,y) = \sum_{l=L_{\min}}^{L_{\max}} S^l_{L_{\min}} w^l (1 - abs(N(\lambda^l(x + \alpha D^l_{L_{\min},x}(x,y)), \lambda^l(y + \alpha D^l_{L_{\min},y}(x,y)))))$$

Here, $N(x, y)$ is the original 2D Perlin noise function described above. To create rougher terrain at the peaks than in the valleys, $S^L_{L_{\min}}(x,y)$ defines a factor modulating

the octave amplitudes and input distortion, using the intermediate result of coarser noise features to weigh the finer features. $D^{L}_{L_{\min}}(x,y)$ denotes the amount of distortion added to the input coordinate of $N(x, y)$. This function sums over the approximated derivatives of $N'(x, y)$, weighted by w^l and $S^{L}_{L_{\min}}(x,y)$, somewhat similar to a standard turbulence function. α and β are constants that can be used to tweak the result. Good results were achieved with $\alpha = 0.15$ and $\beta = 1.1$. This function was largely developed by means of experimentation in the implemented testbed. To this end, the function was implemented as another operation pixel shader, which could be edited and reloaded while the testbed was running. This greatly shortened the iterative development and test cycles. Naturally, other functions can be designed iteratively this way as well.

Again, the output from this function can be weighed by the A function (Eq. 8-1) to limit its effects to the brushed area. Also, the previously described options for noise seeding and different input coordinates can be applied as well.

8.3.4 Distorted Noise

In addition to the previously described noise/turbulence types, another type of noise can be used orthogonally. The idea is that, like domain warping, a second noise source can distort the input to the primary noise function to create even more complex procedural results. So, for example, instead of the standard Perlin turbulence function

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

the turbulence function becomes

$$H^{L_{\max}}_{L_{\min}}(x,y) = \sum_{l=L_{\min}}^{L_{\max}} w^l T(N(\lambda^l x + kN_1(\lambda^l x, \lambda^l y), \lambda^l y + kN_2(\lambda^l x, \lambda^l y)))$$

Another possibility would be to use the following turbulence function.

$$D_{1,L_{\min}}^{L_{\max}}(x,y) = \sum_{l=L_{\min}}^{L_{\max}} w_d^l T(N_1(\lambda_d^l x, \lambda_d^l y))$$

$$D_{2,L_{\min}}^{L_{\max}}(x,y) = \sum_{l=L_{\min}}^{L_{\max}} w_d^l T(N_2(\lambda_d^l x, \lambda_d^l y))$$

$$H_{L_{\min}}^{L_{\max}}(x,y) = \sum_{l=L_{\min}}^{L_{\max}} w^l T(N(\lambda^l(x + w^l D_{1,L_{\min}}^{L_{\max}}(x,y)), \lambda^l(y + w^l D_{2,L_{\min}}^{L_{\max}}(x,y))))$$

Both algorithms would require three times the amount of noise function evaluations. However, the two algorithms result in a different type of distortion. The first algorithm distorts each octave of H independently, the second algorithm uses the same (but scaled) earlier calculated distortion to each of the octaves in H. Consequently, the first algorithm distorts the octaves with noise frequencies similar to the octave's own frequency band and the second algorithm uses all frequencies to distort each octave in H. Both algorithms can aid in the creation of more natural terrain, but especially the second algorithm really adds unique features to the output. As the calculation of this distortion is relatively expensive (as it more than triples the amount of required computations when compared to standard turbulence), only the second algorithm has been found to justify the added cost. Hence, only the latter type of noise distortion has been implemented in the testbed and is available for all discussed turbulence types (Perlin, ridged, billowy, Quilez and erosive).

Note that $N(x, y)$, $N_1(x, y)$ and $N_2(x, y)$ should be uncorrelated noise functions. For this purpose, the same Perlin noise function can be evaluated using different seed numbers. Like the primary noise types, the distortion noises D_1 and D_2 can also be easily adapted to get billowy and ridged distortion by using a different $T(n)$ function. In the testbed, the user is able to influence the distortion noise type (e.g. Perlin, ridged, billowy), the distortion weights w_d and the octave scales λ_d.

This distortive domain warping results in a natural, swirly distortion over the horizontal plane. This can break up regularity and uniformity. For example, see Figure 8-7 to compare the effect of the input distortion to the algorithm discussed in the previous section. Note how (almost) straight lines become swirly, complex curves.

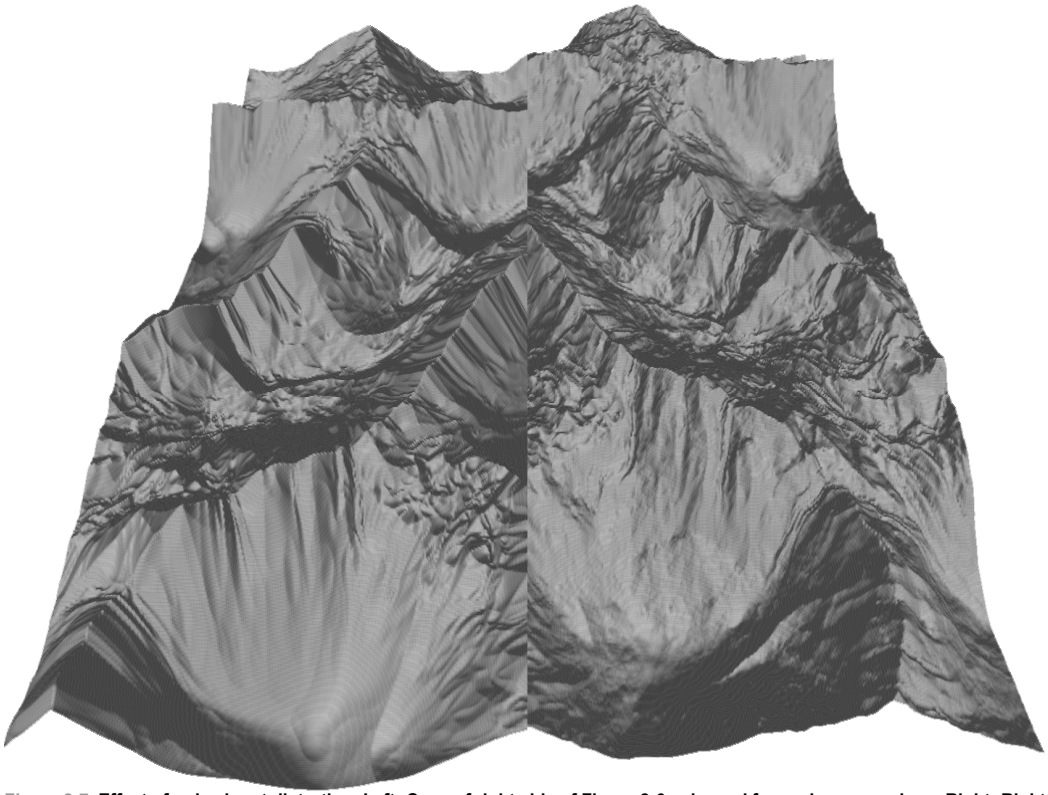


Figure 8-7 **Effect of noise input distortion. Left: Copy of right side of Figure 8-6, mirrored for easier comparison. Right: Right side of Figure 8-6, but distorted by the technique discussed in this section**

8.3.5 Directional Noise

Terrain features in nature are generally not uniformly distributed and direction invariant. The most common turbulence functions, however, do create very statistically uniform and isotropic virtual terrain. The more complex discussed turbulence functions each try to hide these unnatural properties in a different way. Quilez noise varies the weights of the individual octaves in a complex way to an attempt to limit uniformity of the local terrain roughness (see Section 8.3.2). The fluvial noise type from Section 8.3.3 creates a random signal that is dependent on the local slope direction, resulting in less isotropic terrain. The last noise variation discussed in this dissertation and implemented in the testbed is directional noise.

Like distorted noise (Section 8.3.4), this variation is another way to distort the input to the previously discussed noise types in order to create new effects and can be used in

combination with other domain warping techniques. This type of domain warping is more user-controlled and less procedural than the distorted noise variation in that is does not create more natural terrain by itself, but in the hands of the user, it can aid in the creation of interesting anisotropic features. This variation requires the local direction for each brush instance in a brush stroke to be calculated. The direction per brush instance is then used as input to a function in the operation's pixel shader that stretches and compresses the input coordinates used for a turbulence function. The used turbulence function could be any of the algorithms discussed in previous sections.

To implement this effect, the direction of the brush path must be calculated at each of the brush instances in the path. Calculating the local brush direction from the smooth spline path could be calculated analytically, by a central differences method or simply by normalizing the difference between the positions of subsequent brush instances. This direction can then be rotated by some user-specified angle a (e.g. 0º or 90º). The rotated 2D direction vector d for each brush instance is then normalized, multiplied by the scalar k, and sent to the pixel shader by means of shader constants, along with each brush instance's position and weight. k is calculated from the user-specified stretch ratio s. This ratio specifies the size of the noise features in the rotated direction, relative to the size of the noise features in the perpendicular direction. Please note that noise features become stretched when the noise input coordinates are compressed. k is calculated from s as follows:

$$k = \sqrt{1 - \frac{1}{s}}, \quad s \geq 1$$

Then, a coordinate p can be compressed to p' for each pixel in the operation pixel shader with minimum effort:

$$p' = p - d(p \bullet d)$$

This p' coordinate is then used instead of p as input to a noise function. When these brush direction-dependent coordinates are used, the noise features are stretched s times in the rotated brush direction and are left unstretched in the perpendicular direction.

The directional noise can be combined with the distorted noise discussed in the previous section by simple adding independently calculated noise to the x and y components of p'. This will still create elongated features in a user-controlled direction, but will add more natural and complex variation to it. An example of good use of this combination would be to set the rotation to 90º and paint along a mountain ridged with a large brush. This would create elongated features that look somewhat similar to the fluvial noise discussion in the previous section. When the rotation is set to 0º, it could be used to apply a brush stroke from mountain top to valley along the steepest slope to get even more features that would, for example, supposedly be carved out by water streams. See Figure 8-8 for examples of the effect of this brush with different rotations.

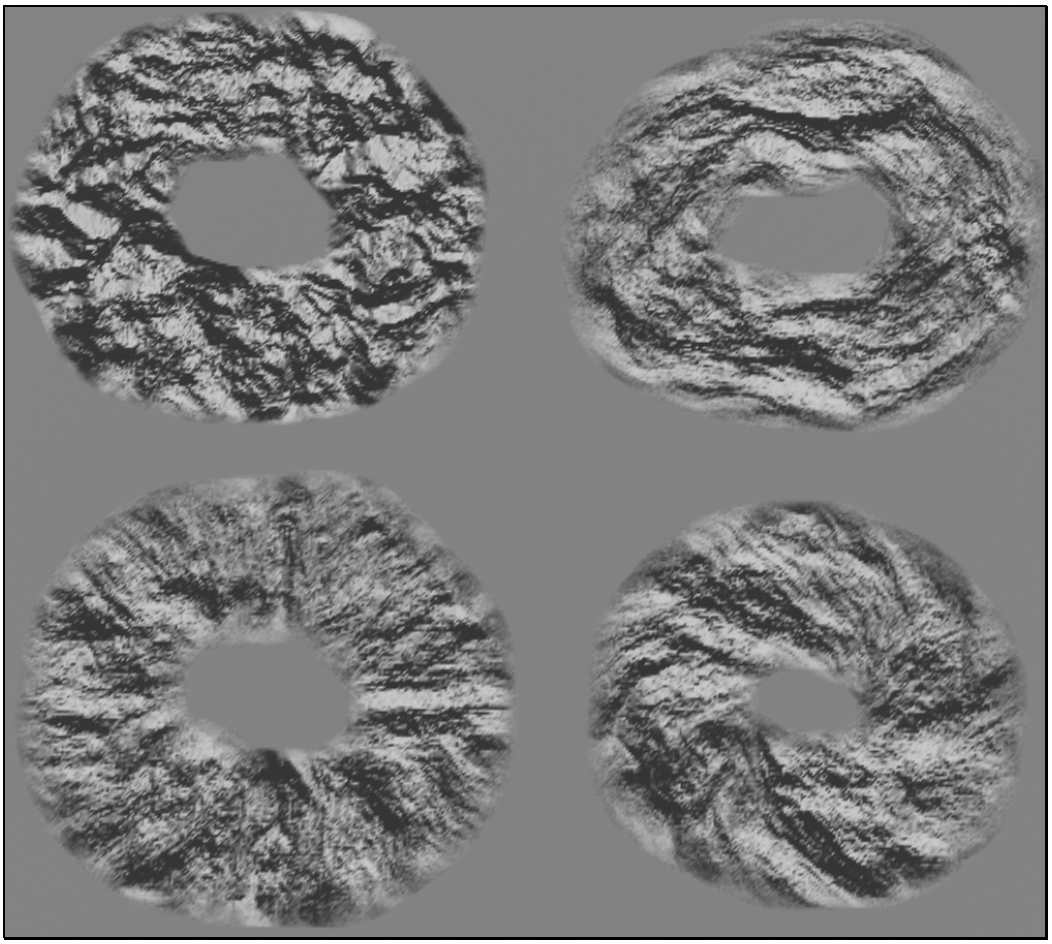


Figure 8-8 **Top view of terrain edited by four circular brush strokes of directional+distorted ridged Perlin noise brush, each with different brush settings. Top left stroke: *s = 1* (no stretching). Top right stroke: *s = 10, a = 0º* (features stretched along stroke). Bottom left stroke: *s = 10, a = 90º* (features stretched perpendicularly to stroke). Bottom right stroke: *s = 10, a = 45º* (stretching at an angle relative to stroke).**

8.4 Smoothing

The last brush that has been implemented for the GPU pipeline is a smoothing brush. In this context, smoothing is considered to be equivalent to blurring. A naïve but correct implementation would be to apply a blurring kernel for each brush instance to the heightfield. The kernel could be a 2D Gaussian(-like) blur filter with all height samples inside the brush radius sampled by the filter tabs. Consequently, the filter would update the heightfield by reading and processing many neighboring height samples for each sample covered by a brush instance. As explained in Section 8.1, this type of algorithm cannot batch brush instances together into a single render call. This means that each brush instance will need to 'ping-pong' between buffers, increasing the overhead due to the many required writes and read-backs. Early experiments revealed that performance was dramatically lower than other implemented brushes. Even the elaborate and complex calculations required for the many noise variations discussed earlier were much faster to finish than the blurring filter, as those brushes were able to batch brush instances together.

One way of improving performance would be to perform an extra layer of indirection. For example, the user could be allowed to draw a mask using a special mask brush. This mask could, for example, be drawn on top of the standard 3D terrain texturing and would be red where brushed and would be transparent otherwise. This mask could be stored in a separate 'maskfield' and be packed, partitioned and paged exactly like a heightfield. Once the mask has been created, the user could then select the smoothing operation. This operation would work similarly to other brush operations in that it needs a pixel shader to update the individual affected heightfield pages. But unlike other brush operations, its pixel shader would use the maskfield as its input to get the local operation strength or weight from, instead of calculating the weight based on the distance to the brush center for each brush instance. In other words, the individual brush instances are combined together into a single input, which is then used to calculate the smoothing all at once. This way, all causal dependencies between the brush instances in a stroke are effectively collapsed. The result of this technique would differ from the earlier discussed naïve technique, and it would not update the heightfield during the actual application of a mouse stroke, but the blurred result would be many times faster to calculate.

To speed up the blurring process when using large filter sizes, the Gaussian pyramid technique discussed in Section 6.6 could be used. In [STRE06], this method has been implemented for the GPU. For large blur filters, the number of samples needed for this method would be much less than the number of samples needed for the brute-force approach. However, it would also require more textures and streams to be coordinated. A less correct, but faster approach of applying a blurring filter has been implemented in the testbed. Even though this approach is only an approximation of a true blurring kernel, results are both fast and convincing. The approach has been inspired by the lattice interpolation technique found in the Perlin noise algorithm. This technique shows very little evidence of the regular lattice grid in the final result and has a fairly clean frequency response. Similar to this idea, the implemented blurring effect uses only a relatively small amount of tabs on a virtual lattice of height samples, which are then interpolated to create a smooth 'blurred' surface from. The lattice height samples are simply samples from the input heightfield sampled at regular intervals. Therefore, the interval (i.e. the distance between the lattice points) is somewhat related to a true blurring filter radius. Unlike the Perlin interpolating technique that interpolates gradients of the closest 2 x 2 neighbors, this implementation interpolates the actual heights of the closest 4 x 4 lattice neighbors for any point inside the affected area to create a smooth 'blurred' surface. These lattice control points are interpolated by a uniform bicubic B-spline blend function. This separable 2D B-spline defines a weight for each of the 16 control point height values using the following formula [CATM78]:

$$w_{ij}(x,y) = \left(B_i \begin{bmatrix} x^3 \\ x^2 \\ x \\ 1 \end{bmatrix} \right) \cdot \left(\mathrm{B_j} \begin{bmatrix} y^3 \\ y^2 \\ y \\ 1 \end{bmatrix} \right) \quad \text{with} \quad B = \frac{1}{6} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 0 & 4 \\ -3 & 3 & 3 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix}, \quad 0.0 \le x, y \le 1.0, \quad 0 \le i, j \le 3$$

Here, B is the standard uniform cubic B-spline blend matrix. $w_{ij}(x, y)$ calculates the weight of control point height value P_{ij} for the surface evaluation at position (x, y). Please note that the i and j subscript in B_i *and* B_j denote the selection of one of the four rows of B. The (x, y) coordinate denotes the position inside a lattice cell, is always relative to P_{11} and lies in the range $[0, 1]^2$. See Figure 8-9. Hence, the height at the local (x, y) coordinate can be calculated as follows:

$$H(x,y) = \sum_{i=0}^{3}\sum_{j=0}^{3} w_{ij}(x,y) \cdot P_{ij}$$

As each point on the sample lattice can be several height samples apart, the brushed area will be sparsely sampled. This is faster to calculate as only 16 samples are required per 'blurred' output sample, and makes the kernel easier to scale in size because the distance between lattice points can be varied at will. For example, visually pleasing results were still achieved when only 1 in 8^2 heightfield samples were on the sampling lattice, requiring only one sixtyfourth of the heightfield samples to be read.

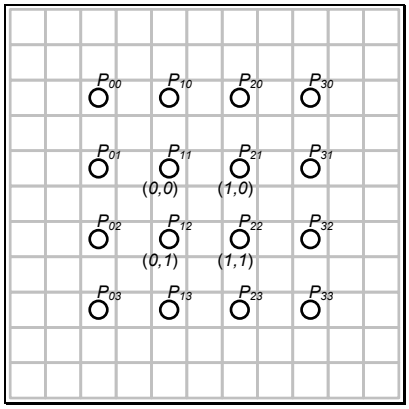


Figure 8-9 **Interpolation lattice of 4 x 4 samples with a lattice spacing of 2**

Again, to use this operation as a brush, the A function (Eq. 8-1) can be used as a blend factor to interpolate between the old unblurred input and the output of the blurring function discussed above. This will limit the effect of the blurring kernel to the brushed area and also offers the user control over the amount of applied blurring.

This concludes the details of the GPU brushes that have been implemented for the testbed. Even though many more techniques and variations that have been discussed in Chapter 6 could have been implemented for the GPU pipeline, the set of implemented algorithms shows the viability of using the graphics processor as a way to speed up different types of calculations. Also, novel techniques which proved to be valuable to the users have been described in detail. The next section discusses, amongst other things, the practical results and considerations of many of the implemented techniques in more detail.

9 Assessment

This chapter will assess the features that were implemented in the testbed editor. The testbed that has been created as part of this research is fitted with several of the techniques discussed in this dissertation. It allows its users to view and edit heightfields using a number of simple and complex brushes, and to import and export heightfield from/to several formats. The suggested GPU pipeline is used to design a number of simple and more complex brushes, in addition to a number of single-thread CPU reference brushes. This chapter assesses the benefits and disadvantages of the discussed techniques in practice and discusses any problems and considerations that were come across during its development.

The first chapters are mainly aimed at analysis requirements and different ideas and methods that could be used to improve heightfield editing in general. The combination of complex procedural/simulation algorithms and the more user-controllable brushing is identified as a powerful mix. For their application, a number of different brushes have been tested. To apply the brushes, a brush path spline algorithm is described and implemented. This created a consistent result, independent from frame rate, brush size and brush type. Even though this is a standard feature in, for example, Adobe Photoshop, it was a considerable improvement over many terrain editing applications. Another feature that users have found to be inadequate in many terrain editing applications is camera navigation. In addition to the more standard W, A, S, D key up/left/down/right camera movement and mouse-controlled camera rotation, the testbed enables the user to navigate as if the rendered image of the terrain is a 2D picture. Panning and trucking (left/right and up/down) camera movement is accomplished by drag-and-dropping any point on the screen: The 3D camera position is adjusted is such a way the originally picked 3D (terrain) position under the mouse pointer will be kept under the mouse pointer during the mouse dragging in real time. Due to the perspective in the rendered image, the amount of absolute 3D horizontal and vertical translation is automatically made to depend on the distance between the picked point and the camera. This prevents the usual disadvantages with fixed-speed key control. Likewise, camera dolly (in/out) movement can be accomplished by picking any point on the screen with the mouse and then 'dragging' to or from this point at will. Again, the amount of movement is made to

increase with distance to the picked point. User feedback verified that this combination of controls combined both navigation speed and precision and felt quite intuitive.

Tests with brushes that naïvely apply low-level single-threaded algorithms showed that modern computers are powerful enough for these to be executed at interactive rates without the need for complex optimizations and parallelism techniques, as long as the used brush radius is relatively small. However, when using reasonably large brush sizes (for example, covering 100000+ height samples), performance did degrade noticeably. For procedural synthesis brushes to be considered working at interactive rates, the maximum practical brush size was reduced even further. All brush strokes are applied internally by working through a queue of brush instances. Hence, operations that are too demanding to be handled in real-time result in delayed terrain updates. Consequently, the naïve implementation resulted in updates that lagged or trailed (far) behind the user's current brush stroke activity when too large or too complex brushes are used.

To optimize the execution speed of brush operations, the graphics cards that are typically available in target PCs were used. The implemented GPU pipeline allowed access to their computational power and large bandwidth. It was found that optimized GPU brushes were roughly up to one magnitude faster than their single-threaded CPU counterparts on the more high-end target PCs and offered about the same performance on low-end target platforms. But as the GPU pipeline causes some extra overhead and asynchronous delays, its full potential was only realized for operations that affected relatively large areas. Also, different heightfield operation algorithms required different pipeline features and could not all be implemented with the same efficiency. Furthermore, several workarounds were needed to get the results working at the bit-depth required for heightfields, causing an additional performance penalty. But as graphics hardware flexibility and standards are improving in a rapid pace, more efficient use of hardware will soon be made possible in the form of more powerful programming constructs and common support for currently optional and exotic features.

As GPUs are currently still specifically created to render graphics, more general concepts were needed to be mapped to render-specific concepts. Also, GPU program debugging tools are currently still relatively primitive. Both factors hindered development and contributed to the fact that writing single-threaded operations took only

a fraction of the time required for the development of the GPU pipeline and its operations. Also, GPU processing is difficult to make reliable under all situations and all (expected) target platforms. For example, uncommon situations like a 'lost device' must be handled properly. This is a DirectX state that results in suddenly corrupted data in video memory and can be caused by external factors like switching to another 3D application. Also, because the support of many capabilities of graphics hardware can differ per vendor and generation, it is difficult to create software that copes with this efficiently. Typically, either non-optimal but safer programs are created, or different code paths are developed for different platforms. Even then, it is difficult to say whether the program will behave as expected on all targeted platforms, unless explicitly tested. For example, some hardware-specific features and limitations were found to lack official documentation. Also, a tested target platform would sporadically produce erroneous output when some brushes were executed on the GPU. After extensive testing, the problem was found not to lie in the testbed but either in the display driver or the actual hardware itself. Even so, standards, and with them the minimally available hardware capabilities, are improving. Each new generation of graphics hardware is more flexible and more powerful, making the discussed drawbacks, workarounds and limitations a temporary inconvenience.

Overall, the testbed was found to be a valuable step in the good direction. However, the choice for having both the research and testbed independent from any specific game engine technology has limited the potential of the testbed in a real production environment. As stated in the Chapter 3, tools used in iterative design need tight integration with the complete production tool chain to reduce overheads as much as possible. Even though the testbed contains quite powerful features and could currently be used in a tool chain by importing and exporting heightfields, the process of importing and exporting from/to other applications will cause a significant overhead. For example, exporting a large heightfield to a file from the testbed can take a few seconds. However, some engines then would take between a few seconds and a few minutes to import the heightfield file. It would then be very inefficient to export the heightfield from some game engine editor, import it into the testbed, make some minor modification, export it and import it into the game engine editor again. To improve this situation, the communication needs to be faster and/or the number of switches between different applications needs to decrease. An example of the first would be to develop a thick

client/thin server system. The game engine-specific thin server would then serve to offer direct memory access to (only) the heightfield area that needs to be edited. The testbed would be the thick client, capable of communicating with any available engine server. A reversed client/server architecture could also work. Then, the actual editing would be done in the game engine's own (client) editor and any actual heightfield operations will be delegated to the testbed server. The render and UI code of the testbed server would then be made obsolete. A third option would be to integrate the testbed operation code completely with an existing game engine editor. Although this would require the most (game-specific) code refactoring, it would also allow for the most game-specific optimizations.

The second and third option would both have additional benefits over the current implementation and the first option. Even though the terrain texturing can be made to resemble the texturing in the final game, the results are probably not exactly the same. Also, editing terrain without context of other objects in a level (e.g. buildings, trees and a water line) adds to the difficulty of using an external terrain(-only) editor. Of course, certain objects could be made importable into the terrain editor to add context, but it would be much more precise and user-friendly to edit terrain in its full context and thus in the game engine's own editor. The above second and third option do exactly that. Note that full integration with a game engine and editor would also imply direct sharing of resources. Especially GPU resources (e.g. textures) might be hard to share between a complete game engine and the memory greedy GPU terrain operations. More (game engine-specific) research would be needed here.

Another difficulty that arose during the development of the testbed was memory consumption. Considerable effort has been spent on minimizing memory requirements and preventing memory leaks. Still, long editing sessions resulted in depletion of available virtual memory, which is limited to 2 GB on Microsoft Windows 32-bit platforms. This depletion was caused by the offered unlimited number of undo levels. Obviously, limiting the undo depth would result in a quick fix but might hinder the user. A better approach would be to use a 64-bit OS or create a custom virtual memory system, capable of virtually addressing a much larger data store, asynchronously and transparently swapping data to and from disk. This would even become more important

when support for heightfield layers would be offered, as this is expected to increase memory consumption considerably.

Even though development of the GPU pipeline was cumbersome, the resulting fast evaluation of complex procedural brushes allowed results to be viewed almost instantly even for large brushes once this pipeline was in place, which is a considerable improvement over current typical editor applications. This did not only assist in the rapid creation of complex terrain by users, it also helped in the design of new procedural variations as the results could be inspected and adjusted in little time. This was further aided by the implemented capability to reload and recompile an operation pixel shader at run-time. Hence, a text editor could be used to experiment with shader code, which could be used instantly in a running editor. This feedback was most valuable for algorithms that are 'tweaked' for visual quality instead of mathematical correctness.

10 Conclusions

The previous chapter discussed the experimental results and practical considerations. This chapter covers interesting fields for future research, followed by the dissertation's concluding remarks, summarizing all discussed topics and findings.

10.1 Future work

A number of areas for future work might extend the capabilities of the suggested and implemented techniques. Some of these have already been mentioned in Chapter 9. Most notably, the workflow of the terrain editing process itself is improved by the techniques proposed in this report and the testbed. However, terrain editing is only one aspect of level design in a production pipeline. As the testbed editor is currently a stand-alone tool, it could be improved by better integrating this editor with the other tools in the pipeline. Different client/server and full integration approaches could be implemented, each with different tradeoffs between execution speed, conversion speed, rendered context and the reusability between game projects. In contrast to the work presented in this dissertation, integration would require exact knowledge of and dependencies on the specific tools and game engine used, including their implementation details.

A number of brushes have been developed using the fast GPU editing pipeline. The results obtained with these brushes have proven the viability and the potential of this approach. An interesting area of future work would be the development of efficient parallel implementations of additional common and novel brush types to even better support the user. Of course, other suggested techniques like the layers and blending techniques could be developed using this pipeline as well. A last interesting area of future work is the development of a set of tools to improve the typically ill-supported creation of unnatural but realistic, man-made terrain features like canals and roads.

10.2 Concluding remarks

The creation of large outdoor terrains, or more specifically heightfields, that are simultaneously functional and realistic for use in computer game levels is currently

hindered by the lack of applications able to rapidly create realistic but controllable features. To this end, many suggestions, recommendations, ideas and experimental results have been compiled in this dissertation.

Firstly, analysis of the iterative design paradigm, as embodied in many creative design tasks including level design, has led to the recognition of the need for simultaneously more controllable and powerful tools than is offered by currently available applications. Suggested improvements include support for a toolset of powerful but controllable procedural brushes, blendable layers, a powerful undo system, friendlier user interfaces, presets to accommodate for various levels of expertise, and continuity and reduction of the edit-and-evaluate cycle time. Some of these suggestions are inspired by techniques found in image editing applications like Adobe Photoshop. These techniques would be fairly easy to translate to the domain of heightfields, as both images and heightfields are essentially matrices of values. Moreover, such features will be instantly recognized by users, shortening the learning curve.

Secondly, techniques to render terrain in real-time have been surveyed. Not only do terrain render algorithms have a direct impact on the frame rate during viewing, they also differ in the time required to present updates to any edited area. Hence, different render algorithms have been discussed, of which one has been optimized in several ways to further improve render and update speed. A texturing method has been discussed that efficiently hides texture stretching on more sloped terrain features. Although this method has not been previously found in literature, it is probably similar to what is offered by some of the best modern game engines. A customizable layered texture splatting technique has been created and described that allows for complex texturing effects, capable of approaching results presented by most modern game engines.

Thirdly, editing algorithms and techniques found in literature and in practice, which would potentially be useful as tools in a level editor, have been described in detail, including their merits and disadvantages. These algorithms have been classified into three categories: low-level brush editing, global (erosive) simulations and global parameterized procedural synthesis. Each of these has its advantages and disadvantages in terms of realism, flexibility and required processing time. Low-level tools work best when precise control is required. High-level simulation and synthesis tools attempt to

create realistic terrain while requiring little effort from the user. This, however, limits the usefulness to a user when more precise control over the created features is required. Also, the complexity of the global algorithms often leads to poor performance. To offer the user a level of control that lies between low-level simplistic but precise tools and high-level global simulation and synthesis techniques, a brush-based procedural approach is suggested. In practice, this also has the advantage of typically limiting the affected area of complex operations to a relatively small area, decreasing the required processing time. Even so, applying complex brushes at interactive rates proved challenging.

Fourthly, as typical heightfield operations were found to consist of applying the same algorithm to large amounts of data elements, it made sense to investigate ways of exploiting the parallelism offered in modern PCs to improve editing at interactive or even real-time rates. Instead of focusing on multi-core CPUs, a specialized pipeline was developed for the typically even more powerful graphics processor, or GPU, found in modern PCs, to execute terrain modification algorithms with. Exploiting the highly parallel GPU for purposes other than rendering is a relatively new approach, only made possibly by recent advancements in its programmability. The developed pipeline was then used to improve the performance of several brushes, offering a number of low-level and procedural techniques. To get the most out of the available hardware, a number of optimizations have been applied, including minimization of the number and size of the areas updated by complex operations, and combining the effect of multiple brush operation instances together in a single render call.

Lastly, valuable progress has been made in improving procedural techniques. Even though commonly used techniques do not create repetitious terrain, they do often tend to synthesize features that look too similar to each other to be considered very realistic. Two novel variations have been described that improve this situation. First, an algorithm has been derived that excels in creating eroded mountainous terrain with statistically different features in the created valleys, tops and slopes. Secondly, a more user-controlled variation has been described that is specifically designed to be used as a brush. This variation creates features that are more dependent on the actual user brush strokes and can therefore create more natural variation.

Summarizing, the research results described in this dissertations represent a considerable step towards simultaneously improving quality, speed and control of the tools offered to game level designers. Discussed techniques for this include currently available tools, ideas from other disciplines and novel algorithms. Experiments showed that even complex algorithms can be offered as interactive tools on today's hardware when parallelism is exploited. Therefore, achieving the ultimate goal of integrating these techniques within one single application can justly be expected to bring about improvement of the iterative workflow, enhancement of user control and simplification of the creation of realistic terrain features.

Bibliography

[ASIR05] A. ASIRVATHAM, H. HOPPE. Terrain Rendering Using GPU-Based Geometry Clipmaps. In *GPU Gems 2*. R. Fernando, Ed., Addison Wesley, 27-45. 2005.

[ATIP04] Delivering the next generation of video performance with PCI Express. ATI white paper. 2004. *http://ati.amd.com/technology/pciexpress/PCIEWP.pdf.*

[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.

[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.*

[BOER00] W. DE BOER. Fast Terrain Rendering Using Geometrical MipMapping. 2000. *http://www.flipcode.com/archives/article_geomipmaps.pdf.*

[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.

[CATM74] E. CATMULL, R. ROM. A class of local interpolating splines. In *Computer Aided Geometric Design*, R.E. Barnhill and R.F. Reisenfeld, Eds. Academic Press, New York, 317–326. 1974.

[CATM78] E. CATMULL, J. CLARK. Recursively generated B-spline surfaces on arbitrary topological meshes. Computer-Aided Design. IPC Business Press, vol. 10, no. 6, 350-355. Nov. 1978.

[COMPNV] Comparison of Nvidia graphics processing units. *http://en.wikipedia.org/wiki/Comparison_of_NVIDIA_Graphics_Processing_Units*

[COOL69] J. COOLEY, P. LEWIS, P. WELCH. The finite Fourier transform. In *IEEE Transactions on Audio and Electroacoustics, vo*l.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.

[DACH06a] C. DACHSBACHER. Interactive Terrain Rendering: Towards Realism with Procedural Models and Graphics Hardware. Thesis. University of Erlangen–Nürnberg. 2006.

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

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

[DUCH97] M. DUCHAINEAU, M. WOLINSKY, D.E. SIGETI, M.C. MILLER, C. ALDRICH, M.B. MINEEV-WEINSTEIN. ROAMing Terrain: Real-time Optimally Adapting Meshes. In Proceedings of the IEEE Visualization '97. Los Alamitos, CA, USA. IEEE Computer Society Press, 81-88. 1997.

[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.

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

[FERN03] R. FERNANDO, M.K. KILGARD. The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics. Addison-Wesley. 2003.

[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.

[GORA84] G.M. GORAL, K.E. TORRANCE, D.P. GREENBERG, B. BATTAILLE. Modeling the interaction of light between diffuse surfaces. ACM SIGGRAPH Computer Graphics, vol. 18, no. 3, 213-222. Jul. 1984.

[GREE05] S. GREEN. Implementing Improved Perlin Noise. In *GPU Gems 2*. R. Fernando, Ed., Addison Wesley. 409-416. 2005.

[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.

[HOPP98] H. HOPPE, Smooth view-dependent level-of-detail control and its application to terrain rendering. In *Proceedings of the IEEE Visualization '98*. Los Alamitos, CA, USA, 1998. IEEE Computer Society Press, 35-42, 1998.

[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.

[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.

[LIND96] P. LINDSTROM, D. KOLLER, W. RIBARSKY, L. HODGES, N. FAUST, G. TURNER, A. Real-time, continuous level of detail rendering of height fields. In *Proceedings of the SIGGRAPH '96*. ACM Press, New York, NY, USA, 109-118. 1996.

[LOSA04] F. LOSASSO, H. HOPPE. Geometry clipmaps: terrain rendering using nested regular grids. In *ACM Transactions on Graphics (TOG),* vol. 23, no. 3, 769-776. Aug. 2004.

[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.

[MAX88] N.L. MAX. Horizon mapping: shadows for bump-mapped surfaced. In *The Visual Computer*, vol. 4, no. 2, 109-117. March 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.

[NVID05] NVIDIA GPU Progeramming Guide 2.4.0. NVIDIA Corporation. *http://developer.download.nvidia.com/GPU_Programming_Guide/GPU_Programming_Guide.pdf.*

[OAT06] C. OAT, P. SANDER. Ambient aperture lighting. ACM SIGGRAPH 2006 Course Notes, Advanced real-time rendering in 3D graphics and games session, 143-152. 2006.

[OGRE3D] Ogre 3D open source graphics engine. *http://www.ogre3d.org*.

[OIS] Object Oriented Input System. *http://sourceforge.net/projects/wgois/.*

[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.

[ONEI05] S. O'NEIL. Accurate Atmospheric Scattering. In *GPU Gems 2*. R. Fernando, Ed., Addison Wesley, 253-268. 2005.

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

[PHAR05] M. PHARR et al. GPU Gems 2. R. Fernando, Ed., Addison Wesley. 2005.

[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.

[PERS07] E. PERSSON. ATI Radeon HD2000 Programming Guide. *http://www.atitech.com/developer/SDK/AMD_SDK_Samples_May2007/Documentations/ATI_Radeon_HD_2000_programming_guide.pdf*

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

[QUIL08] I. QUILEZ. More noise. Online article. *http://rgba.scenesp.org/iq/computer/articles/morenoise/morenoise.htm.* 2008.

[RÖTT98] S. RÖTTGER., W. HEIDRICH, P. SLUSALLEK, H.P. SEIDEL. Real-Time Generation of Continuous Levels of Detail for Height Fields. In *Proceedings of WSCG '98*, 315-322. 1998.

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

[SISOFT] SiSoftware Sandra XII. *http://www.sisoftware.net/*

[STRE06] N. STRENGERT, M. KRAUS, T. ERTL. Pyramid methods in GPU-based image processing. In *Proceedings of the Vision, Modeling, and Visualization 2006 (VMV2006).* 169-176. 2006.

[SLOA02] P.-P. SLOAN, J. KAUTZ, J. SNYDER. Precomputed Radiance Transfer for Real-Time Rendering in Dynamic, Low-Frequency Lighting Environments. In *ACM Transactions on Graphics (TOG),* vol. 21, no. 3, 527-536. Jul. 2002.

[SNOO01] G. SNOOK. Simplified Terrain Using Interlocking Tiles. In *Game Programming Gems 2, M. Deloura, Ed.,* Charles River Media, Inc. 377-383. 2001.

[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.

[ULRI02] T. ULRICH. Super-size it! Scaling up to Massive Virtual Worlds. ACM SIGGRAPH 2002 Course #35 Notes, 75-85. 2002.

[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.

[WENZ06] C. WENZEL. Real-Time Atmospheric Effects in Games. ACM SIGGRAPH 2006 Course Notes, Advanced real-time rendering in 3D graphics and games session, 113-128. 2006.

[WINTAB] WinTab tablet interface. *http://www.wacomeng.com/devsupport/ibmpc/downloads.html.*

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

[WXWIDG] wxWidgets cross-platform GUI library. *http://www.wxwidgets.org/.*

Table of Figures

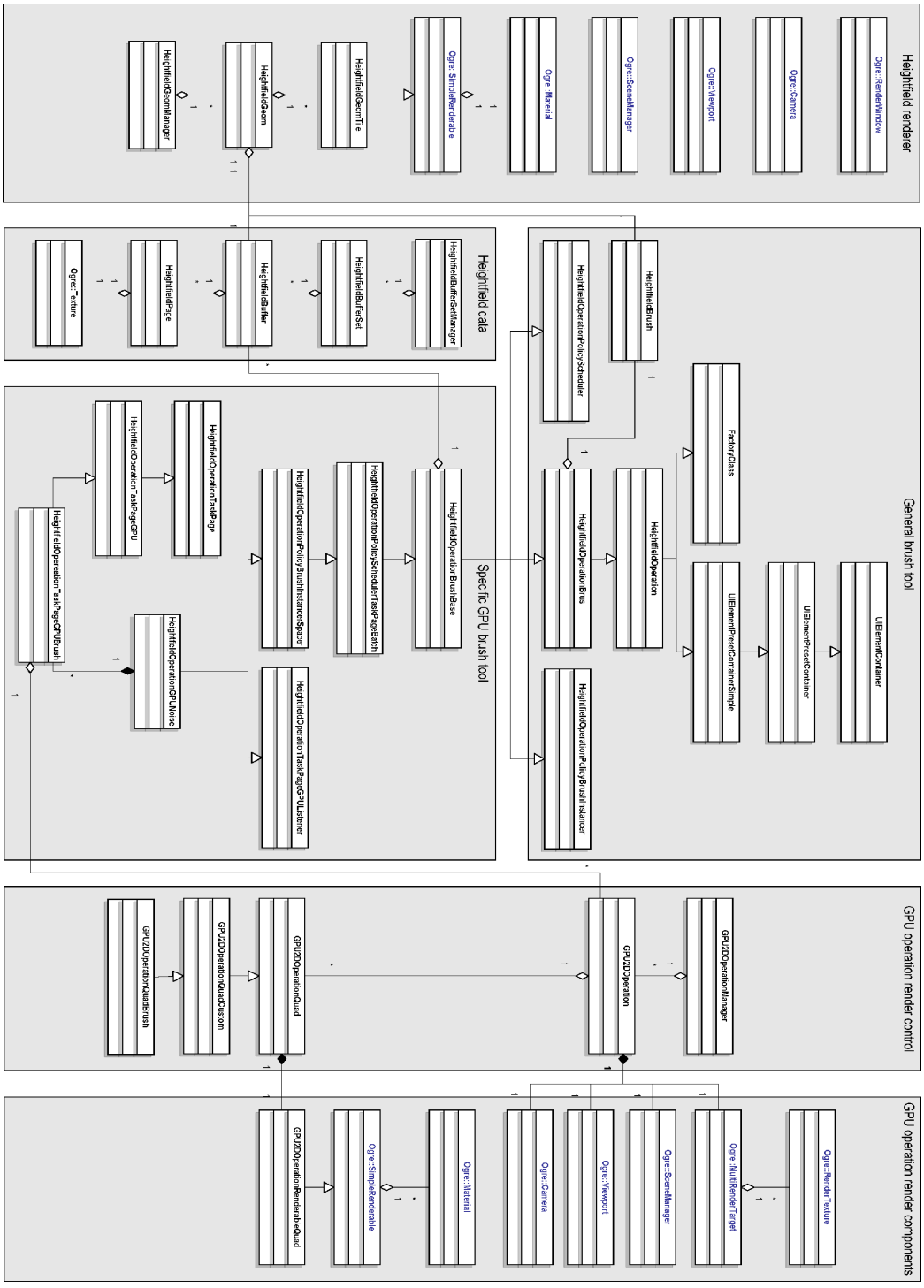
Table of Tables

Appendix A. UML class diagram testbed

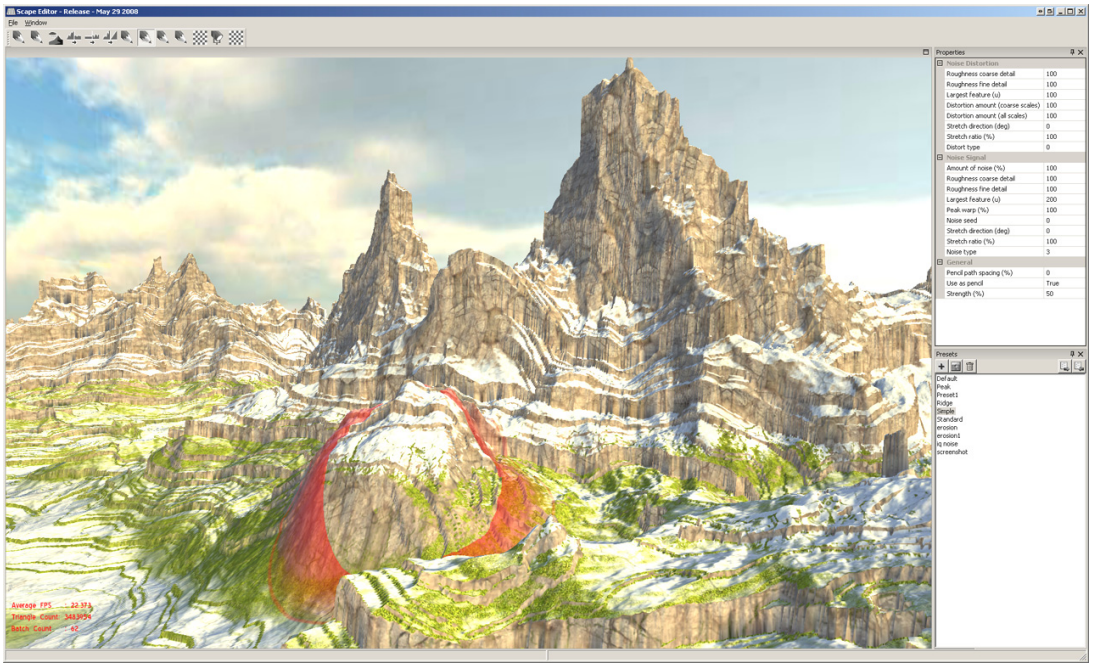
UML class overview diagram of the implemented testbed. See Appendix B for details on the classes that are related to GPU processing.

Appendix B. UML class diagram GPU editing

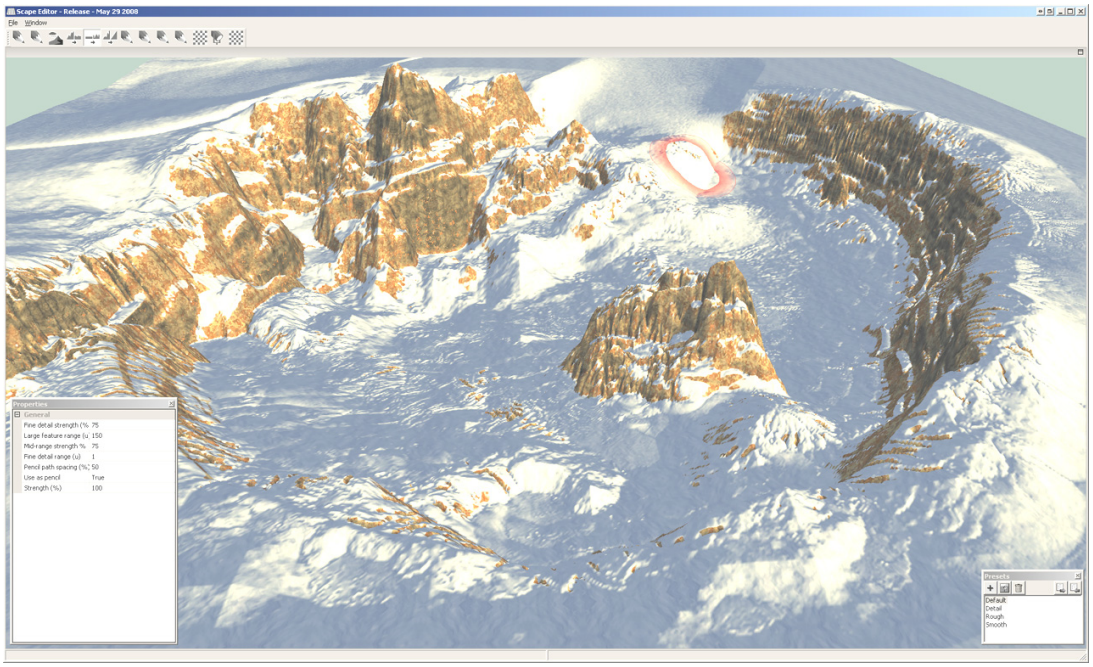
UML class overview diagram of classes related to GPU editing. The inheritance tree is specific for the noise brush 'HeightfieldOperationGPUNoise' and could be different for other brushes.



Appendix C. Testbed editor screenshots



Example of procedural synthesis result in testbed editor. The terrain outside the red brush area was created completely procedurally on the GPU by a custom experimental shader program that performs several noise evaluations and performs various range and domain warps



Example of terrain made by user-controlled brush strokes in testbed editor. Applied brushes include a simple push/pull brush, a turbulence brush and an erosion simulation brush