TERRAINOSAURUS

REALISTIC TERRAIN SYNTHESIS USING GENETIC ALGORITHMS

A Thesis

by

RYAN L. SAUNDERS

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

December 2006

Major Subject: Computer Science

TERRAINOSAURUS

REALISTIC TERRAIN SYNTHESIS USING GENETIC ALGORITHMS

A Thesis

by

RYAN L. SAUNDERS

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Approved by:

Chair of Committee,	John Keyser
Committee Members,	Glen Williams
	Donald House
Head of Department,	Valerie Taylor

December 2006

Major Subject: Computer Science

ABSTRACT

Terrainosaurus: Realistic Terrain Synthesis Using Genetic Algorithms. (December 2006)
Ryan L. Saunders, B.S., Texas A&M University
Chair of Advisory Committee: Dr. John Keyser

Synthetically generated terrain models are useful across a broad range of applications, including computer generated art & animation, virtual reality and gaming, and architecture. Existing algorithms for terrain generation suffer from a number of problems, especially that of being limited in the types of terrain that they can produce and of being difficult for the user to control. Typical applications of synthetic terrain have several factors in common: first, they require the generation of large regions of believable (though not necessarily physically correct) terrain features; and second, while real-time performance is often needed when *visualizing* the terrain, this is generally not the case when *generating* the terrain.

In this thesis, I present a new, design-by-example method for synthesizing terrain height fields. In this approach, the user designs the layout of the terrain by sketching out simple regions using a CAD-style interface, and specifies the desired terrain characteristics of each region by providing example height fields displaying these characteristics (these height fields will typically come from real-world GIS data sources). A height field matching the user's design is generated at several levels of detail, using a genetic algorithm to blend together chunks of elevation data from the example height fields in a visually plausible manner.

This method has the advantage of producing an unlimited diversity of reasonably realistic results, while requiring relatively little user effort and expertise. The guided randomization inherent in the genetic algorithm allows the algorithm to come up with novel arrangements of features, while still approximating user-specified constraints.

TABLE OF CONTENTS

LIST OF FIGURES

FIGURE Page

FIGURE Page

LIST OF TABLES

CHAPTER I

INTRODUCTION

Computer-rendered terrain is an important facet of most graphical applications that attempt to represent the natural world in some way, including CG-animated feature films, virtual reality systems, computer games, and art. The ability to synthesize large-scale, realistic terrain models is of considerable interest for many of these applications.

Fig. I.1. An Outdoor Scene from Halo 2. Modern graphics hardware is capable of rendering detailed terrain models in real time, as in this scene captured from *Halo 2* ([Bungie 2004]).

Much of the recent research in virtual terrain has centered around accelerating the visualization of large terrains to achieve interactive frame rates, resulting in a wide array of *level of detail (LOD) algorithms*, methods of increasing rendering speed by omitting details that are too far away or would be otherwise indiscernible to the viewer [Duchaineau et al. 1997] [Ulrich 2002] [Losasso & Hoppe 2004] [Li et al. 2003]. Recent, dramatic advances in the performance and capabilities of graphics acceleration hardware have enabled the interactive presentation of richly-detailed terrain models, and have sparked increased interest in

---

This thesis follows the style and format of the *Journal of the ACM*.

representing outdoor phenomena, as evidenced by the prominence of nature-related techniques at a recent *Game Developer's Conference* [Sanchez-Crespo 2002]. The visually stunning outdoor scenes in recent game titles, such as *Halo 2* [Bungie 2004] (Figure I.1), are solid evidence that believable terrain can be visualized in real-time with current technology.

Terrain generation, in contrast, has received comparatively little treatment in the literature. Fractal-based techniques are the most prevalent, because they are easily implemented, require relatively little processing time and human input, and yield at least mediocre results, allowing random, unique terrains to be generated rapidly. However, as several authors [Losasso & Hoppe 2004] [Pelton & Atkinson 2003] have noted, fractal methods for terrain generation are limited in the types of terrain they can simulate, and one generally has to resort to elevation maps digitized from the real world to get more interesting and believable terrain models.

In this thesis, I present *Terrainosaurus*, a new method of synthesizing realistic, heterogeneous (with respect to type of terrain) height fields at multiple levels of detail. This method departs from current industry-standard approaches to terrain generation in several important ways, most notably through the use of real-world terrain data as raw material, and of artificial intelligence methods to control the generation process.

There are a number of desirable characteristics for a terrain generation algorithm that could be optimized during its design, some of which are in conflict with one another. However, not all of these characteristics are equally important for all applications of terrain generation, and so, by selecting a more narrowly construed problem, it is possible to make some reasonable decisions as to which characteristics are of the highest priority. *Terrainosaurus* is aimed at the needs of "studio" users: artists, animators, simulation and video game designers, people who typically have high goals for realism and quality, have powerful computing resources at their disposal, and do *not* have real-time processing constraints. This means that, in order to be maximally useful to its target user base, *Terrainosaurus* should optimize the following characteristics:

- realism, such that the terrain models generated by it create a plausible illusion of the real world
- extensibility, such that new types of terrain can be added to its repertoire on an as-needed basis
- ease of use, such that a human user is not burdened with tedious detail or arcane controls

The motivations and goals of *Terrainosaurus* are covered in more depth in Chapter II.

The variant of the terrain generation problem most commonly addressed is that of generating a digital elevation map, in the form of a discrete height field: i.e., a rectangular grid of elevation values (Figure I.2). This formulation of the problem is popular because of a number of simplifications it makes:

- for every (x, y) coordinate pair within the bounds of the height field, there is precisely one corresponding elevation value; thus, the resulting surface is manifold everywhere but at the height field edges
- the horizontal spatial resolution is constant throughout the terrain, being pre-defined by the resolution of the rectangular grid

The advantages and disadvantages of of height fields, as well as those of several alternative representations for terrain, are discussed in Section III.1.2.

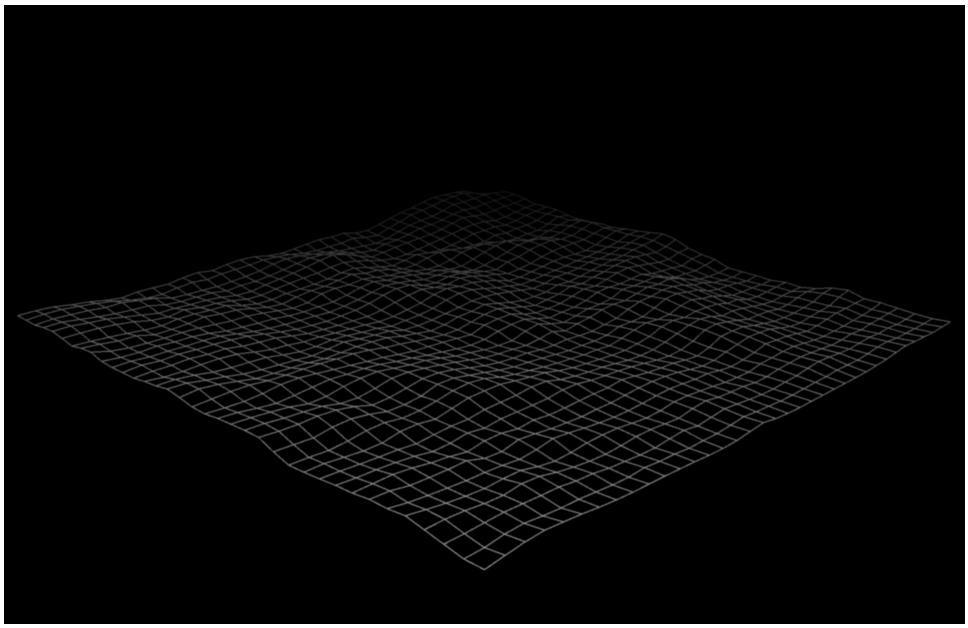

Fig. I.2. A Height Field. *Terrainosaurus* generates terrain in the form of *height fields*, rectangular grids containing one elevation value for each (x, y) coordinate pair.

To this problem of terrain height field generation, we apply genetic algorithms. Genetic algorithms are a class of techniques for solving difficult optimization problems using the metaphor of biological micro-evolution, and are discussed in further detail in Section III.3.1. This approach allows the graceful melding of competing goals, including design objectives specified by the user and realism constraints largely outside of the user's direct control.

From the perspective of a user, *Terrainosaurus* is composed of three distinct phases (Figure I.3):

- assembly of a library of *terrain types*, collections of GIS elevation data (or *terrain samples*) that possess similar characteristics and belong to the same category of terrain in the user's mind
- authoring of a 2D "map" describing the size, shape, and locations of one or more regions of terrain
- generation of a height field conforming to this map, with each region of terrain evincing similar characteristics to those displayed by the terrain samples belonging to the corresponding terrain type

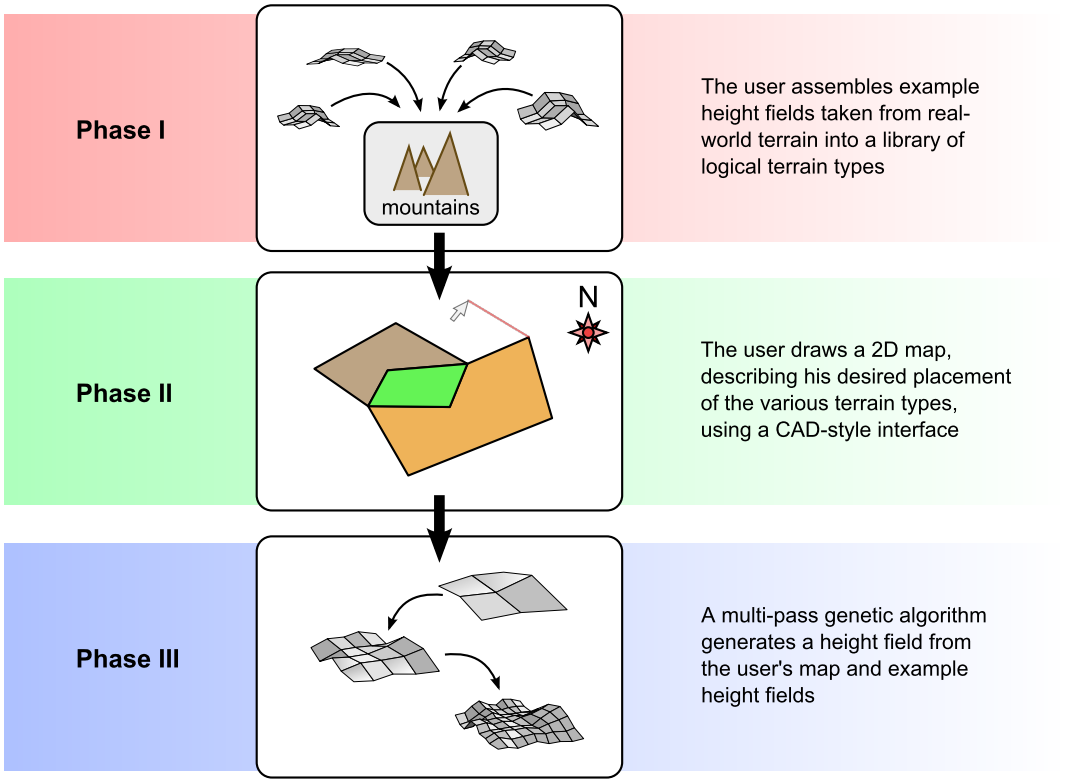

Fig. I.3. The Three Phases of *Terrainosaurus* .

The first of these phases, the construction of the terrain library (discussed in Section IV.2.1), is essentially a classification process. The user (or a third party) provides a number of sample height fields, presumably from GIS data sources, and sorts them into separate, logical terrain types. In this way, the user is able to construct a "palette" of terrain types, with which he will later create his map. In so doing, the user describes *by example* the characteristics he desires to have in the generated terrain.

At the moment, this process is entirely manual, and likely to be somewhat tedious, as it relies on the user to compare terrain samples visually and to determine terrain type membership on this basis. Furthermore, not all GIS elevation maps can be used as example terrains at the present time (the presence of bodies of water, for example, significantly alters the statistical characteristics of a height field, making it unsuitable as a reference example). In Chapter VII, I propose a number of areas for future work with the potential both to ease the burden on the user and to relax the restrictions on example height fields. Still, even in *Terrainosaurus*'s current state, this is not as cumbersome as it might seem, as this task need be performed only infrequently: once assembled, a terrain library can be reused indefinitely.

The second phase, the authoring of the map, is where the user will do most of his work, and is also where he has the most freedom to create. The user expresses his desired terrain configuration by sketching arbitrary

polygonal regions using a 2D CAD-style interface, and assigning each region a terrain type from the library constructed in the first phase.

Normally, the user will not want the boundaries between adjacent regions of terrain to be rigidly linear, so in addition to normal polygon editing operations, we provide a *boundary refinement* operation, which is a genetic-algorithm-controlled subdivision operation, replacing a straight boundary with a series of short linear segments, forming an irregular, less artificial-looking boundary connecting the same endpoints as the original (Figure I.4).

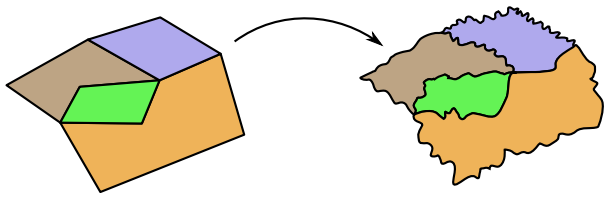

Fig. I.4. Boundary Refinement in the User's Map. In order to prevent artificial-looking linear boundaries from being apparent in the generated height field, *Terrainosaurus* provides a *boundary refinement* operation to subdivide the linear boundaries of the polygonal regions in the map into irregular boundaries.

The third phase, generation of the height field, is almost totally automated. The user selects a rectangular region of his map, and a target spatial resolution, and then launches the generation process. A genetic algorithm is applied repeatedly to generate successively higher resolution height fields (i.e., successively finer levels of detail). At each LOD, the task of the genetic algorithm is to find a plausible way of arranging small patches of height field data taken from the respective terrain types created by the user. The generated height field is deemed "good" insofar as each region is similar to the user-provided examples for the terrain type it is supposed to represent (Figure I.5).

The details of the *Terrainosaurus* algorithm are discussed in much greater depth in Chapter IV.

In contrast to many of the "quick and dirty" algorithms in common practice today, *Terrainosaurus* is somewhat more complex, and is correspondingly more difficult to implement. In the interest of illuminating the task of implementation somewhat, I discuss some of the design considerations, hurdles, and lessons learned from the prototype implementation in Chapter V. An individual interested in implementing *Terrainosaurus* may find this discussion useful, to avoid the problems we encountered during our research.

Conceptually, the main results of this work include:

- a new, genetic-algorithm-based method for generating terrain at multiple LODs without the use of fractals
- a terrain authoring paradigm for visually designing large-scale height fields that places a minimal burden on the user in terms of effort and domain knowledge
- a means of comparing terrain height fields for similarity, based on their features and statistical characteristics

In Chapter VI, I discuss the results produced by this work, as well as some of the significant problems and "wrong turns" encountered during the research. In the process of this research, we identified a number of promising avenues for further research, possibly leading to improvements in the quality of the generated height fields and streamlining of the user experience. These are discussed in Chapter VII.

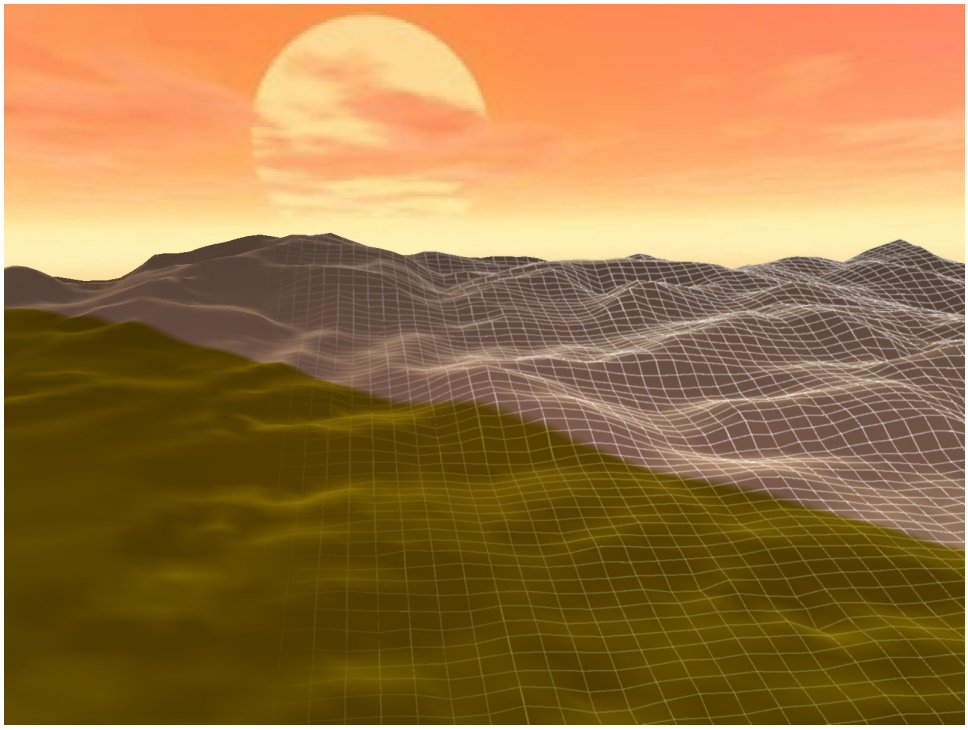

Fig. I.5. A Height Field Generated by *Terrainosaurus* . The generated terrain may be composed of heterogeneous terrain types arranged into arbitrary regions.

CHAPTER II

MOTIVATION

The goal of *Terrainosaurus* is to "build a better mousetrap", so to speak. Established terrain generation methodologies have their own respective strengths and weaknesses, which will be discussed in Section III.1.3. *Terrainosaurus* is an attempt to address the shortcomings of these existing methods, yielding a better way of generating artificial terrain for many applications.

In order to elucidate the motivation for something like *Terrainosaurus*, and to establish the context for the discussion that follows, it is worth spending time addressing some preliminary questions:

- Why generate terrain?
- What characteristics would an *ideal* terrain generation algorithm have?
- What should be the primary objectives for *Terrainosaurus*?

II.1. Applications of Terrain Generation

Before bothering any further with *how* to generate terrain, it is first helpful to establish *why* it is worth doing, and who the primary users of terrain generation algorithms are.

Terrain generation has applications in a number of fields, many of which are entertainment-related, though certainly not all are. Examples include:

- Computer-generated art & animation—both the commercial variety and "art for art's sake" often depict the natural world
- Video games & virtual reality (VR)—many virtual worlds involve at least some natural terrain. In addition to the entertainment uses of virtual worlds, they also have applications in military & non-military simulation
- Architectural rendering—while the focus of an architectural rendering is generally the buildings themselves, realistic terrain can help to provide the larger context for the buildings, contributing to the overall visual effect

II.2. Idealized Terrain Generation

As is the case in many problem domains, there are a number of traits that an ideal terrain generation algorithm would have, some of which are in tension with one another. Such an ideal algorithm is probably a figment of our collective imagination; nonetheless, having such an ideal in view provides a yardstick with which to evaluate the strengths and weaknesses of real algorithms.

While this is not an exhaustive list, such an imaginary, ideal algorithm would:

- require a low degree of human input
- permit a high degree of human control
- be completely intuitive to control

- be able to reproduce a wide variety of recognizable terrain types and features realistically, in believable relationship to one another
- produce models at arbitrary levels of detail
- run quickly enough to be used in real-time, dynamic applications
- be extensible to support new types of terrain

II.2.1. Requiring a Low Degree of Human Input

Digital terrain models typically involve large amounts of data. For example, a standard 10-meter USGS DEM (digital elevation model) [USGS 2003] contains more than 10,000 data points per square kilometer. If an artist has to place each point manually, this adds up to an enormous amount of painstaking work for the artist. Therefore, any terrain construction detail that can be automated without significantly limiting the artist's control over the result should yield an increase in productivity. An ideal terrain generation methodology would accept a command as simple as "give me a 10 km by 5 km area of desert-like terrain with a resolution of 10m per sample" and produce a believable result.

If the inputs required by an algorithm are sufficiently minimal, there may be a second benefit, in addition to the increase in artist productivity: the algorithm may be useful for generating unique terrains automatically (e.g., as a random world generator for a game).

II.2.2. Permitting a High Degree of Human Control

In obvious tension with the previous ideal is the goal of permitting the artist to exert an arbitrarily fine degree of control over the features and characteristics of the generated terrain. To give an artist full creative freedom requires that he be able to control the behavior of the terrain across all scales, from the macro-scale features (hills, valleys, mountains, etc.) to the micro-scale details (cracks, crevices, etc.), with any degree of localization, creating both global and local effects. An ideal algorithm would permit the artist to exert whatever amount of control he wishes, where he wishes, while intelligently filling in the details he doesn't care to specify directly.

II.2.3. Intuitively Controllable

If a user is required to gain arcane knowledge or specialized skills to use a tool effectively, that tool will have a correspondingly steep learning curve. An ideal terrain generation algorithm would be perfectly intuitive to use—a complete "black box" from the user's perspective, requiring no understanding whatsoever of the algorithm's innards. All of the inputs available to the user would be easily understood: even an inexperienced user would have a reasonable idea of how tweaking an input would affect the generated terrain.

II.2.4. Capable of Diverse, Believable Features & Terrain Types

A terrain generation algorithm that can reproduce only a narrow range of terrain types and features is inherently limited in its usefulness. An ideal algorithm would be able to create a wide diversity of terrain types (e.g., desert, mountains, plains) and features (e.g. ravines, riverbeds, volcanoes), both from the real world and from the imagination of the artist. Additionally, the transitions between different types of terrain

(e.g., from mountains to foothills) would be believable, and the placement of semantic features would make sense (waterfalls would pour into pools, rivers would always flow downhill).

II.2.5. Arbitrary Level of Detail

It is precisely because detailed terrain models involve such large quantities of data that continuous level of detail (CLOD) algorithms are necessary for their visualization, at least in the general case where the viewer is permitted to move about the terrain freely. However, full-detail terrain models are not always needed, the requisite level of detail being determined by their intended use. For example, a model meant to act as the setting for a computer-animated film needs highly realistic detail only in the areas near to which the action is to take place, and can use relatively simple geometry for distant terrain. Similarly, the level of detail required of the terrain in a flight simulation is significantly less than what is needed for ground-level action. An ideal terrain generation algorithm would permit the creation of terrain at multiple levels of detail, and the segmentation of large terrains into regions of differing levels of detail.

II.2.6. Fast Enough for Real-Time Applications

Many terrain-related algorithms are very slow, or else require a lengthy preprocessing phase [Ulrich 2002] [Torpy 2006] before the terrain can be used in a real-time application. Intuitively, we would expect all but the most simplistic terrain generation algorithms to fall into this category. Nonetheless, an algorithm that could do most or all of its processing on-the-fly would provide several important advantages, including:

- instantaneous feedback to the artist of the effects of a particular modification
- the ability to modify dynamically the terrain in fundamental and interesting ways (such as transforming a mountain into a crater, or rapidly eroding a riverbed into a deep gorge)
- savings in the amount of memory and disk storage required (if the full terrain can be (re)generated from a more compact set of parameters)
- the ability to create seamless, infinite worlds, with arbitrarily fine, dynamic level of detail

II.2.7. Extensible

Regardless of how many types of terrain a terrain generation tool can create, there will always be a user wanting something "just a little bit different"—the set of terrain types is ultimately as limitless as the human imagination. Thus, an ideal terrain generation algorithm would be extensible in some way, allowing new types of terrain to be introduced easily.

II.3. Goals

This idealized algorithm has set the standard of perfection fairly high, and *Terrainosaurus* most certainly will not possess all of the traits of our imaginary ideal. So when it becomes necessary to compromise on one trait in order to improve another, which traits should be preferred?

Different applications have different requirements, and the needs of the primary user community of terrain generation tools should be used to answer this question—if users demand realism and *Terrainosaurus* gives

them speedy generation but mediocre results...it will not be very popular. Looking back at the above list of terrain generation applications, one thing that most of them have in common is that they have time to spare: with the possible exception of some games & VR applications, none of them need to generate terrain at interactive speeds. Even in this last case, the terrain is normally generated once (either by the game designer, or when the program starts) and never modified afterward. In fact, most CLOD algorithms *depend* on the terrain being static). Thus, real-time performance, while nice, is not essential.

A second, significant observation is that terrain is, by its very nature, somewhat "sloppy": two mountains might have innumerable, minute differences between them, but if some fundamental relationships are intact, a human will perceive them as similar—the differences are inconsequential. The success of randomized algorithms in imitating natural phenomena can be largely attributed to this fact. Because of this, and because the terrain model being created is often quite large, a user of a terrain generation tool typically does not care to exercise a great deal of fine-scale control over the terrain. Therefore, when forced to choose between the competing goals of high controllability and ease of use, we should favor the latter.

In the design of *Terrainosaurus*, the goals considered to be most important are (in order of decreasing importance):

1. realism
2. extensibility
3. ease of use (intuitive control, with low input requirements)

By pursuing these as guiding objectives, we can expect that the resulting algorithm will be of maximal utility to the terrain generation community, particularly for non-real-time authoring of large-scale virtual environments.

CHAPTER III

BACKGROUND

In this chapter, I cover a number of topics that serve as necessary background material for understanding the *Terrainosaurus* algorithm. First, I survey the history and state-of-the-art in virtual terrain, and then highlight several previous works that are especially similar to *Terrainosaurus*. Finally, I touch upon several auxiliary topics that are directly relevant to *Terrainosaurus*.

III.1. Previous Work in Virtual Terrain

In order to understand the relationship of *Terrainosaurus* to the field of terrain generation, it is helpful to have a grasp of the scope of current terrain generation literature and praxis. To this end, in this section, I review the following topics:

- a brief discussion of the issues involved in mapping ellipsoid objects (e.g., the Earth)
- a survey of different structures for representing terrain, with their benefits and drawbacks
- a taxonomy of existing methods of generating terrain
- a discussion of level of detail issues in terrain generation
- a survey of existing terrain generation software
- pointers to terrain-related sites on the Internet

III.1.1. Geodetic Mapping

The Earth (and the other planets) are (roughly) ellipsoidal objects of enormous size. In most aspects of our everyday life, it suffices to think of the Earth simply as an infinite plane, completely flat. Only when we deal with a planet at larger scales (e.g., for global positioning and navigation) does it become necessary to account for the effects of planetary curvature.

Researchers in the cartographic and astronomic sciences have been devising schemes for dealing with these effects for a long time. With the rise of computing technology, the field of *Geographic Information Systems (GIS)*, the application of computers to mapping and understanding the Earth, has made great use of these schemes to produce digital maps of most of the Earth's surface. Still, GIS mapping is not without its difficulties, many of them due to attempting to "unwrap" the ellipsoidal planet surface to produce a planar representation of the geography; there appears to be no natural way of doing this. A full discussion of the difficulties of planetary mapping, and their work-arounds, is outside of the scope of this thesis, but to name a few:

- One way of producing a planar surface from an ellipsoidal surface is to map the ellipsoid using a spherical coordinate system, and then to create a rectangular map with the latitude and longitude as the two axes. The major problem with this approach is that distances become more and more distorted towards the poles. For a digital terrain model, this means that if the surface is sampled uniformly in the latitude/longitude coordinate system, the sample points will not be uniformly distributed across the surface of the sphere, but will be denser around the poles.

- Another way of producing a planar surface is to "slice" the ellipsoid like an orange and to flatten each slice of the map with a local planar projection. This results in less distortion, but causes discontinuities in the planar map and creates large areas in the map that do not correspond to any point on the ellipsoid.

For more information on geodetic mapping, projection & coordinate systems, and GIS in general, the USGS's website is a good place to start [USGS 2006].

III.1.2. Methods of Representing Terrain

One of the most fundamental decisions to be made when working with virtual terrain is that of how to *represent* the terrain. The choice of data structure will affect the set of available tools in our terrain generation "toolbox", and may also limit the kinds of terrain features that can be represented or the ways in which the terrain can be edited and used. Some questions that must be answered based on the needs of the application include:

- Does the terrain need to have infinite precision, such that it can be viewed at any arbitrary scale, or is it acceptable to have a finite, maximal resolution?
- Is it important to be able to represent terrain structures like caves and overhangs, in which multiple surfaces have the same horizontal coordinates, or will the terrain surface obey the vertical line test at every point?
- Are the effects of planetary curvature important, or is a "flat Earth" approximation good enough?
- Will the terrain surface need to be rendered and/or tested for object collisions in an efficient manner?

In light of these considerations, we can compare the merits and limitations of several alternative representations for terrain:

- height fields
- voxel grids
- non-uniform meshes
- analytic and fractal functions

III.1.2.1. Height Fields

The terrain representation most widely used at the present time is probably the *height field*. A height field represents a surface as a scalar function of two discrete variables, such that the horizontal coordinate pair (x, y) determines the elevation at that point. While there is nothing precluding the use of an infinite, continuous function or a non-rectangular domain, in usual practice, this function is discretized at regular intervals in x and y, with x and y valid over a finite, rectangular domain, the width and height of the height field (Equation III.1).

$$z = f(x, y) \begin{cases} x \in [0, w) \\ y \in [0, h) \end{cases} \qquad \text{(III.1)}$$

This formulation leads to the familiar implementation of the height field as a 2D array of scalar elevation values (Figure III.1).

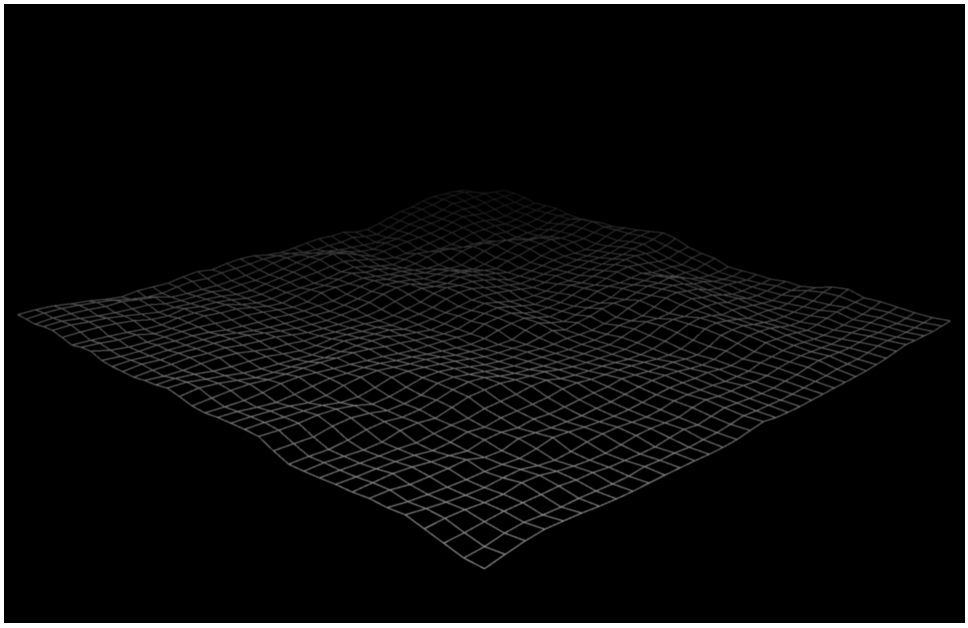

Fig. III.1. A Discrete Height Field.

Advantages of Height Fields

Once we observe that the discrete form of the height field is essentially the same thing as a greyscale image, this quickly leads us to the insight that computer vision and image processing techniques may be used to construct, modify, analyze, and compress terrain models represented as height fields (for example, a rough terrain could be made smoother by applying a standard Gaussian blur filter to it, and a height field can be stored using an image file format).

A second advantage of the height field is that its regular structure makes it possible to optimize operations like rendering, collision-detection and path-finding. The rendering of even very large height fields in real-time has been made feasible by the invention of a number of *continuous level of detail (CLOD) algorithms*, which render highly visible areas of the terrain with detailed geometry, using progressively simpler geometry for obscured or more distant parts of the terrain [Duchaineau et al. 1997] [Ulrich 2002] [Li et al. 2003] [Losasso & Hoppe 2004]. Collision detection, an expensive operation in the general case, can be done cheaply when one of the objects is a height field since, given an (x, y) location in the height field, only a few surrounding triangles need to be checked for collision.

A third advantage is that significant quantities of real-world terrain data are available in height field form, making it the representation of choice for working with GIS data.

Disadvantages of Height Fields

The most fundamental disadvantage of the height field is that, since surface elevation is a function of an (x, y) coordinate pair, there must be exactly one elevation for every pair of coordinates. Because of this, a height field is inherently unable to represent caves, overhangs, vertical surfaces, and other terrain structures in which multiple surfaces have the same horizontal coordinates (Figure III.2). In practice, this limitation is often inconsequential, since most natural terrain is fairly "well behaved" in this respect, and the exceptional cases can be handled by modeling overlapping structures as separate objects placed atop the terrain (though this solution does have an undesirable *ad hoc* quality to it).

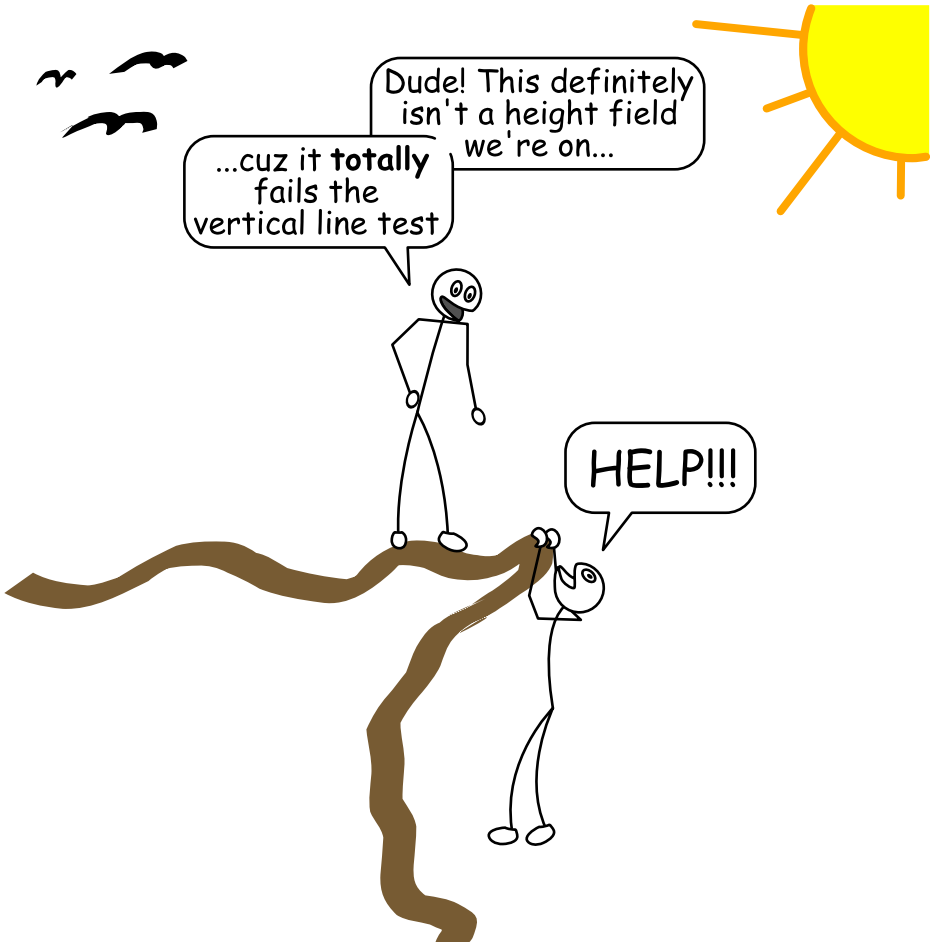

Fig. III.2. The Problem With Height Fields. One of the main disadvantages of height fields is that they are inherently limited in the kinds of features that they can capture; features such as overhangs and caves cannot be represented in a standard height field.

A second disadvantage of the height field, when used in its usual, discrete form, is that it has a finite, uniform resolution. This has two negative implications. First, because the resolution is *finite*, this places a firm upper bound on the degree of fine-scale detail that the height field can represent. As a result, when viewed from a too-close viewpoint, the terrain may appear blocky, featureless and unnatural. Second, because the resolution is *uniform* throughout the domain, a a height field cannot gracefully handle terrains having a highly variable local level of detail. If the resolution is chosen to match the average scale of the features in the terrain, then any finer-scale features will be simplified or eliminated; conversely, if the resolution is chosen to be high enough to capture the fine-scale features, areas containing only coarse features will also be captured at this same high resolution, an undesirable waste of space and processing time. Ideally, a terrain representation for terrain generation would either be infinite in resolution, or else would adaptively increase its resolution to accommodate the addition of fine scale details, rather than requiring an *a priori* decision about resolution.

A third disadvantage of the height field, if used to represent terrain on a planetary scale, is that rectangular height field patches do not map onto spheroid objects any better than rectangular image textures do. If the standard, two-pole spherical projection is used to map a height field onto a spherical planet, the density of height field points will be substantially greater in areas near the poles than at those near the equator.

III.1.2.2. Voxel Grids

Another possibility for representing terrain is the height field's 3D cousin, the *voxel grid*. A voxel grid is a discrete, three-dimensional grid of *voxels* (the volumetric equivalent of pixels), in which (in the simplest case) each voxel is either filled or not. By selectively filling voxels, one can create arbitrary 3D shapes.

Advantages of Voxel Grids

The main advantage that voxel grids have over height fields is that they are not constrained by the vertical line test, and can represent vertical surfaces, overhangs, etc.

Disadvantages of Voxel Grids

Voxel grids inherit most of the problems of height fields, and add several of their own. They share the height field's disadvantages of having a finite resolution, and of not handling planetary curvature gracefully. Rendering and collision detection are more expensive than with height fields, and more data must be pushed to the graphics card for the same amount of rendered horizontal area. Additionally, without the use of spatial subdivision techniques (such as octrees), voxel grids are typically very wasteful of memory (as large chunks of the grid are either completely empty or are buried deep underground).

III.1.2.3. Non-uniform Meshes

A more general way of handling terrain is to represent the surface of the terrain as an arbitrary mesh of 2D primitives (usually polygons, but sometimes quadric or cubic Bezier or NURBS patches) embedded in the 3D space. Such mesh surfaces are popular ways of modeling and animating characters and other objects, and there are numerous tools available for working with objects in this form.

A *triangular irregular network (TIN)* [Pajarola et al. 2002] is a special case of a non-uniform mesh. In a TIN, the surface is represented as a mesh of triangles derived from a set of 3D points (using, for example, the Delunay triangulation). TINs are useful ways of creating a mesh representation from real-world data—the points, called *mass points*, could be sampled directly from the terrain surface, or could be derived from the contours of a topographic map.

Advantages of Meshes

The main benefit of using meshes to model the terrain surface is that they are extremely general. The surface may have arbitrary geometry and topology (overhangs, caves, rock arches, etc.), and an artist working with meshes is free to model even the most bizarre and heterogeneous terrain structures using a single modeling paradigm.

A second advantage of meshes is that they naturally support variable level of detail, allowing more vertices in areas of sharp change and relatively few vertices in flat areas. As a result, a mesh structure can store some terrain models much more efficiently than regular grid methods, since it does not require a globally high resolution in order to achieve fine-scale features in a few places.

Furthermore, as most of the tools for computer modeling and animation support this paradigm, there is a significant user base that is already comfortable with manipulating objects represented in this way.

Disadvantages of Meshes

The main difficulty with using meshes for terrain generation is that it is not clear how to generate them automatically; even though terrain is almost always tessellated into some sort of polygonal representation before rendering, I am unaware of any terrain generation methods (other than manual sculpting) that work directly on a mesh representation (i.e., without the use of a more constrained, terrain-specific data structure).

III.1.2.4. Continuous Functions

A final possibility is to represent the terrain with some sort of analytic or fractal function. This approach is seldom used in practice, with the *MojoWorld* world generator being a rare, but impressive example [Pandromeda 2004].

Advantages of Continuous Functions

Continuous functions (both analytic and fractal) have the advantage of being viewable at any scale without losing resolution; viewed close-up, they do not look "faceted" as height fields tend to do. Beyond this, both analytic and fractal functions have their own advantages.

Certain classes of analytic functions offer mathematical advantages that make them friendly for rendering and/or collision detection/response. Many types of analytic functions are differentiable at most or all points, allowing precise derivatives to be calculated. Polynomial surfaces of low degree (quartic and below) have closed-form solutions, allowing ray/surface intersections to be calculated in a straightforward way.

Fractals offer a different advantage: unlike analytic functions, which typically become more and more linear when viewed at finer and finer scales, fractal functions continue to produce new details as they are evaluated

at progressively finer scales, meaning that a fractal terrain can have as much fine-scale detail as the display system can render.

Disadvantages of Continuous Functions

The main problem of using continuous functions is the difficulty of modeling with them. If a single, global function is used, it is difficult to know how to modify the function to achieve a certain local effect. A more usable approach is to compose a number of functions, each having a local area of effect (B-spline patches are example of this), though this has yet to replace polygon-based techniques as the dominant modeling paradigm.

The other drawback of continuous functions is the difficulty of rendering them. For a polygon-based rendering system, the function must be transformed into a form that the graphics hardware can process (e.g., triangles). For a ray tracing system, the first intersection of a ray with the surface must be evaluated. In either case, depending on the complexity and topology of the function, this process may be quite expensive.

III.1.3. A Taxonomy of Terrain Generation Methods

Terrain generation has not received nearly as much attention in the literature as terrain LOD rendering has, with relatively few innovations to date. The discussion of terrain generation in the computer science literature dates back at least as far as 1977, to a paper in which terrain created from simple mathematical functions is used to stage simulated military combat scenarios [Parry 1997]. Since that time, a variety of methods for producing terrain have been proposed, generally falling into four broad categories:

- GIS-based methods
- Sculpting methods
- Simulation methods
- Procedural methods

III.1.3.1. GIS-based Methods

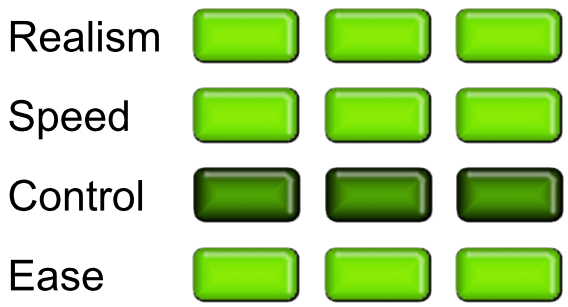

Fig. III.3. Typical Characteristics of GIS-based Methods. GIS-based methods are simple to implement, and borrow their realism directly from the real world, but are limited to the locations and resolutions for which real-world data is available.

The most obvious place to find natural-looking terrain is in nature itself. While not strictly a method of *generating* terrain, the use of GIS data is a simple and effective way of getting very realistic terrain models (Figure III.3). High-quality elevation maps (height fields) of the entire United States (and some other parts of the world) can be downloaded or ordered from a variety of GIS data providers (such as *Geo Community* [Qlinks 2006] and the *U.S. Geological Survey (USGS)* [USGS 2004]). These data are available in several formats (with older elevation maps available in the DEM format [USGS 2003] and more recent data available only in the more complex SDTS format [USGS 2003]) and at resolutions as high as 3 meters per sample in some cases (though resolutions of 10 and 30 meters per sample are much more commonplace).

GIS data is ideal when the overriding concern is realism, and is often the best or only option when an application needs to represent actual, real-world locations faithfully. Applications of this sort include mapping software [ESRI 2006] and simulations set in real-world locations [Electronic Arts 2003] [Microsoft Game Studios 2004]. Using GIS data, a high degree of realism can be achieved for relatively little human effort.

The major drawback to the use of GIS data is the constraint that it imposes: only real-world locations that are present in a mapping agency's database can be used. Thus, while little or no effort is required to *use* GIS data in an application, *finding* suitable data may be time-consuming or simply impossible. For many applications, this may unacceptably compromise artistic objectives (e.g., a movie director may want a particular arrangement of mountains and rivers) or other, more utilitarian goals (e.g., a particular military combat scenario might require a particular set of terrain features). A secondary disadvantage of GIS-based methods is the significant amount of space required to store large datasets. Finally, the level of detail available through GIS sources is limited—if a particular application requires finer-scale detail than that offered by the mapping agency, one will have to resort to other means for achieving it.

III.1.3.2. Sculpting Methods

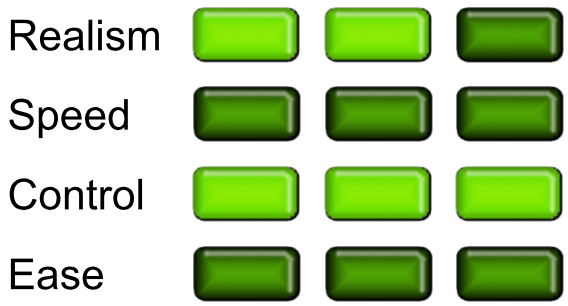

Fig. III.4. Typical Characteristics of Sculpting Methods. Sculpting methods allow the terrain author to construct virtually anything imaginable, but require significant effort and skill.

At the opposite end of the spectrum, in terms of human effort, is the use of artistic tools to "paint" or "sculpt" the features of the terrain (Figure III.4). In most cases, this means using computer-based modeling

and painting tools (such as *Adobe Photoshop*, *Bryce 3D*, or *Maya*) or specialized "level editors" (such as those shipped with the recent game titles *SimCity 4* [Electronic Arts 2003] and *Unreal Tournament 2004* [Epic Games 2004]). Less commonly, an artist might create a model using traditional, physical media and then digitize it using laser scanning techniques, as was done to create the terrain for the video game title *Trespasser* [Wyckoff 1999].

The primary advantage of sculpting methods is the enormous freedom given to the artist. Anything the artist can conceive, he can achieve, given sufficient skill and effort.

This strength is also its main drawback—achieving a desired result with this method typically requires a large investment of human time and effort, and the quality of the results is heavily dependent on the skill of the artist. As the size of the virtual environment increases, so does the cost of sculpting it, making this method of terrain generation less and less feasible as projects increase in scope.

III.1.3.3. Simulation Methods

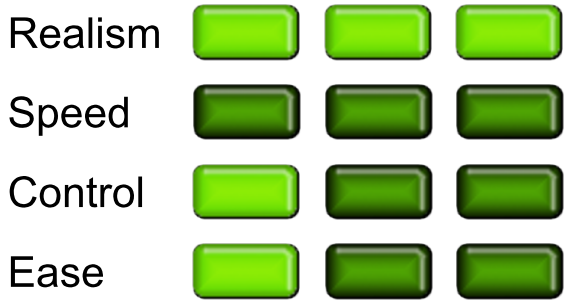

Fig. III.5. Typical Characteristics of Simulation Methods. Simulation methods have high potential for realism, but are computationally expensive and difficult to control.

Another family of terrain generation methods involves the evolution of terrain by simulating the effects of physical processes such as plate tectonics or erosion by wind or water [Kelley et al. 1988] [Musgrave et al. 1989] [Burke 1996] [von Werner 1996] (Figure III.5). Simulation techniques for terrain generation have been explored to a much smaller extent than other methods.

These methods have the potential to produce highly realistic results, to the extent that they accurately model the physical processes they are intended to simulate. They can also be very hands-off, requiring little input from the user.

One drawback of simulation methods is the amount of processing time required. In order for the simulation to produce realistic results, it must run at sufficiently fine resolutions in both space and time to adequately capture the effects of these processes; as the resolution of the simulation increases, so does the time required to run it. A second drawback of such methods is the relative lack of user control: since a simulation is intended to capture the effects of natural processes, the role of the human user is limited to setting up the

initial conditions for the simulation and kicking it off—achieving specific, user-specified effects may be difficult.

III.1.3.4. Procedural Methods

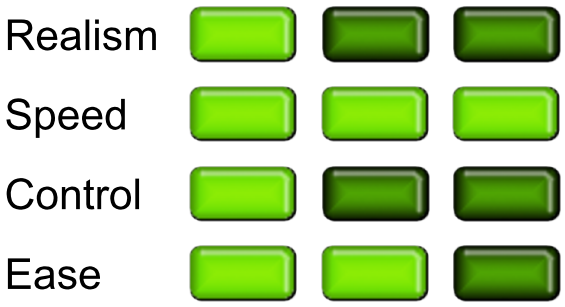

Fig. III.6. Typical Characteristics of Procedural Methods. Procedural methods are often simple to implement, relatively fast to run, somewhat difficult to control, and limited in realism.

The term "procedural methods" describes a broad family of terrain generation techniques whose unifying factor is that they produce a terrain model through the application of some sort of automatic "procedure" (Figure III.6). While the aforementioned simulation methods (Section III.1.3.3) fit this description, I consider them separately because, unlike true simulation methods, most procedural generation methods do not attempt to simulate any physical processes, and use techniques that are quite different from the numeric integration that is typical of simulations. In most cases, procedural generation methods are simply "hacks" that have been found to work acceptably for generating certain types of terrain and rely heavily on random number generation to produce irregular variations in the terrain surface. Some of the earliest discussions in the literature focus on procedural methods for approximating terrain [Parry 1997] [Marshall et al. 1980].

Some of the most popular procedural techniques are fractal in nature, such as the midpoint displacement method (MPD) [Fournier et al. 1982] and Perlin Noise [Perlin 1996]. These techniques exhibit the fractal property of self-similarity at different scales, and generally involve randomly perturbing the height values of the terrain by increasingly smaller amounts at increasingly finer scales. Prusinkiewicz & Hammel describe a method for generating mountains with an integrated river network that is entirely fractal in nature [Prusinkiewicz & Hammel 1993].

A similar class of procedural techniques in current practice is known as *collaging* or *faulting* methods [Burke 1996]. In these techniques, an irregular height field is created through the random superposition of simple shapes, such as spheres, cones, half-planes, or 2D trigonometric functions. After several hundred to several thousand iterations, the result is a random height field possessing similar curvature and smoothness characteristics to the primitive shape, but without any of the primitive shapes directly apparent in the terrain.

Less frequently used are a family of *spectral synthesis* techniques, which involve randomly or semi-randomly constructing a frequency-spectrum (Fourier- or wavelet-space) representation of a terrain and then performing the inverse transform to produce the spatial representation of the terrain [Pelton & Atkinson 2003] [Franke 2000].

Procedural methods for terrain generation are popular among artists and game designers [Fernandez 2006] [Woodhouse 2003], largely because they are easy to understand and implement. They also generate unique and (ideally) interesting results each time, and can run with little or no human input in many cases, making them useful for generating random game worlds. If used in this way, the application need not store any height field data on disk, since a new, unique terrain environment can be constructed each time the application runs.

The principal drawback of these methods is that they generally have no causal connection to the real-world terrain they try to emulate, instead bearing only an incidental resemblance (in other words, they only happen to *look* terrain-like rather than being somehow *derived from* the same principles and processes that cause the formation of real terrain). Because of this, most procedural methods are only useful for simulating a limited range of terrain types. In addition, since the procedural parameters do not typically correspond to real-world terrain characteristics (and are often rather unintuitive), achieving a desired effect is likely to be an exercise in trial and error.

III.1.4. Level of Detail Considerations

Another way of classifying terrain generation methods, essentially orthogonal to the taxonomy of the previous section (Section III.1.3), is with respect to how they handle level of detail. In terrain rendering, LOD considerations are important for handling large terrain models efficiently. In terrain *generation*, however, LOD acquires a whole new level of importance. In rendering, LOD may be thought of simply as an upper limit on the amount of detail to be shown; in generation, LOD has (or *should* have) an enormous impact on the shape and features of the generated terrain. Terrain features occur across a variety of scales, ranging from entire mountain ranges down to small cracks in the ground. As the LOD of a terrain is reduced, the finer details are lost, leaving only the larger features; because of this, we may consider those details that disappeared as *belonging to* the finer LOD, and we may design our algorithms to exploit this. At least three classes of generation methods can be distinguished with respect to LOD:

- methods that are LOD-agnostic, and generate the full spectrum of detail in one shot
- methods that generate detail at different scales, and combine them via superposition
- methods that work from coarse to fine, progressively introducing finer and finer modifications to the terrain

In the following sections, I describe these three classes of methods, concluding with a discussion of the applicability of fractals to terrain generation.

III.1.4.1. LOD-agnostic Methods

The most simplistic approach to LOD is, of course, to ignore it as much as possible. A number of generation methods follow this course, considering LOD only when initially selecting the resolution of the height field (or other structure).

Many of the procedural methods discussed in Section III.1.3.4 fit this description. Collaging/faulting methods [Burke 1996] are a good example: the resolution of the height field is fixed at the start of the algorithm, and the collaging process runs to completion without explicit consideration for the LOD of generated features. The features produced, whether large or small, are simply the consequence of the random overlapping of hundreds or thousands of simple shapes.

Another class of methods that is typically LOD-agnostic is the class of simulation methods discussed in Section III.1.3.3. Since simulation is usually done using numerical integration techniques on a regular grid structure, the initial choices of grid resolution and simulation time-step are normally the only points at which LOD considerations are consciously applied. Thereafter, the physical features of the terrain emerge through the approximation of physical processes, without further reference to LOD (though the initial choice of LOD will certainly affect the accuracy of the simulation).

These examples are sufficient to illustrate the principal weakness of most LOD-agnostic algorithms: because they do "everything at once", with many overlapping effects, the effects of any individual step in the generation process are difficult or impossible for the human user to distinguish, making the generation process inscrutable, unpredictable…and uncontrollable. As was mentioned in Section III.1.3.3, physically-based simulations have the drawback of being difficult for a user to guide towards a desired result. This is partially attributable to the LOD-agnostic nature of the simulation: a tweak to the initial conditions of the simulation may (or may not) produce the desired effect at a certain scale, but may also have undesired side-effects at larger or smaller scales. Non-physically-based methods of the LOD-agnostic sort share this characteristic of being opaque to the human user, and carry the additional liability of tending to generate physically incorrect shapes. It is difficult to see how such methods could ever rise above the level of simple "hacks" to become controllable, realistic terrain generators.

III.1.4.2. Superposition Methods

A second approach to LOD is to construct detail in separate "layers" of different scales, and then to combine these layers by adding them together (employing *superposition*, to borrow signal theory terminology). To understand the distinction between these methods and the following class (Section III.1.4.3), it is important to grasp that these different layers are not *themselves* the LODs of the terrain, but only the details belonging to a finite "slice" of the frequency spectrum. The actual LODs can be reconstituted by summing together all of the layers up to the requisite degree of fineness.

Spectral synthesis methods [Pelton & Atkinson 2003] [Franke 2000] take precisely this approach: by setting the values of the various frequency-domain coefficients, one can control the presence and importance of details at different scales. The recombination of these different scales into a spatial-domain height field is done by applying the appropriate inverse transform (Fourier or wavelet) to the frequency-domain representation.

Another example of a superposition method is Perlin Noise [Perlin 1996], in which random noise layers of different frequencies and amplitudes are added together; the choice of frequency and amplitude for each of the layers gives some measure of control over the overall characteristics of the resulting height field.

The practical success of Perlin Noise in computer graphics can be largely attributed to its versatility. Compared to the LOD-agnostic methods previously discussed (Section III.1.4.1), Perlin Noise (and similar superposition methods) allow a user to achieve a greater variety of effects, by allowing him to exert some degree of predictable control over the behavior of the surface at different scales.

III.1.4.3. Progressive Refinement Methods

A third approach to LOD in terrain generation is to refine the terrain surface progressively, starting from a coarse LOD and adding finer and finer details until the desired LOD is reached. These methods differ from those in the previous section (Section III.1.4.2) in that they construct new, fine-scale details as modifications to the coarser LODs.

These methods often work by recursive subdivision: given a complete, coarse LOD of the terrain, the next-finer LOD is constructed by subdividing the coarse LOD and, at the same time, introducing new, small-scale modifications to the terrain. The *Midpoint Displacement Method (MPD)* [Burke 1996] is, perhaps, the prototypical example of this. *Terrainosaurus* also takes a progressive refinement approach, though with a bit more sophistication in how it introduces new details.

A main advantage of progressive refinement methods over superposition methods is that they can make use of the previously constructed, coarse detail in deciding where and how to introduce the new, fine-scale detail. While no previously existing method of which I am aware exploits this at all (and *Terrainosaurus*, in its current incarnation, does so only to a limited degree), I believe that this family of methods holds the most promise for generating realistic terrain containing recognizable, coherent features, such as rivers, volcanoes, and ravines.

III.1.4.4. Fractals & Non-fractals

All terrain generation methods that are not agnostic with respect to LOD must decide how to handle the different LODs: what kinds of features will appear in each LOD, how large they will be, and how they will relate to one another. A physically correct solution to this problem is difficult (perhaps the "Holy Grail" of terrain generation), and as a result, many researchers and practitioners have turned to fractals as an approximate solution.

In his seminal work, *The Fractal Geometry of Nature* [Mandelbrot 1982], Benoit Mandelbrot observed that many objects in nature display the fractal characteristic of *self-similarity*, having essentially the same structure when viewed at a variety of scales. For example, the small-scale variations in elevation across a 1-meter-square piece of mountainous terrain might, if one zooms back far enough, happen to resemble the large-scale variations across the entire mountain range. This observation has been applied to great effect in computer graphics, with fractal algorithms being used as reasonable approximations of a number of natural structures, both regular (e.g., plants and trees) and irregular (e.g., terrain).

Randomized fractal algorithms are a convenient method of creating irregular shapes across an entire range of LODs. Because each successively finer LOD has a defined relationship to the one preceding it (modulo the effects of the random number generator), fractal algorithms offer effectively unlimited amounts of detail:

when rendering a fractal surface, one can evaluate the fractal to whatever degree of fineness the display system is able to show.

Still, as useful as the fractal approximation for terrain is, it cannot tell the whole story—not all types of terrain exhibit conveniently self-similar properties across all the levels of detail at which we experience them. To see that this must be the case, one need only consider the diverse array of physical forces that interact to create real-world terrain—plate tectonics, erosion due to wind & water, earthquakes, avalanches, floods & volcanic eruptions, dehydration & thermal expansion (to name a few)—it is intuitively obvious that each of these forces acts more noticeably at some scales than at others. Figure III.7 is a photograph, taken close-up, of cracked, dried mud formed at the bottom of a volcanic crater in the Death Valley area in California; if a fractal relationship held between this bit of terrain and its surrounding macro-environment, we would expect to find it in the context of an area characterized by broad, flat mesas, separated by narrow, severe ravines. As it turns out, this photo was taken at the bottom of a relatively shallow, sloping crater. Obviously, in this example, no fractal relationship holds between these two scales.

Fig. III.7. Cracked Mud. A photograph taken close-up of cracked mud. The physical process predominantly responsible for this effect does not manifest in the same way in the surrounding larger-scale environment. As a result, while a fractal relationship does appear to exist in the terrain in this photo, we would not expect this local effect to be co-reducible, along with its surrounding environment, to a single fractal function. (Photo courtesy of *www.freenaturepictures.com*.)

Therefore, since fractal relationships cannot be trusted to hold in the general case, any terrain generation algorithm that relies on fractal behavior is inherently limited—there will be types of terrain (probably *many* types of terrain) that it will be unable to reproduce. Any terrain generation algorithm that aims to support a broad range of terrain types at various scales must allow for the possibility of non-self-similar terrain.

This leads to the observation that multi-scale, *fractal* terrain generation algorithms are simply a special case of a more general class of multi-scale, *not-necessarily-fractal* terrain generation algorithms. Fractal algorithms answer the question, "What should the terrain look like at this scale?", with, "Just like it did at the larger scale, but smaller." This is a straightforward way of generating terrain across multiple LODs, but it is probably not sufficient for many types of terrain. Part of the goal of *Terrainosaurus* is to provide an alternative answer to this question, leading to a terrain generation methodology that is more general and controllable.

III.1.5. Existing Tools for Terrain Generation

While the list of terrain generation software is too large to be reviewed exhaustively in this section, it is worthwhile to make note of a representative sampling. Most of these applications permit the use of more than one of the above techniques as a way of mitigating the weaknesses of each.

III.1.5.1. Terragen

Terragen [Planetside 2006] is a well-known terrain generation and rendering tool, developed by Planetside Software, free for non-commercial use. It provides several fractal algorithms for generating terrains and also provides a simple set of sculpting tools for modifying the resultant height fields. It offers an integrated ray tracing engine including some very nice cloud, water, and atmospheric lighting effects, and has a significant artistic community.

III.1.5.2. MojoWorld

MojoWorld [Pandromeda 2004] is a set of applications for creating and exploring fractal worlds. *MojoWorld Transporter* is a free program for exploring the worlds created by the commercial product, *MojoWorld Generator*. Unlike most hybrid fractal/sculpting terrain generators, *MojoWorld* handles fractal objects in symbolic form, allowing entire worlds to be viewed at any level of detail and stored in relatively small files.

III.1.5.3. Bryce 3D

Bryce 3D [DAZ 2006] is a general-purpose 3D modeling and animation tool, but is perhaps best known for its terrain modeling capabilities. It offers a full set of painting/sculpting tools for generating and editing height-field-based terrain objects, including the ability to use Adobe *Photoshop* image filters.

III.1.5.4. World Machine

World Machine [Schmidt 2006] is a procedural height field generation tool, free for non-commercial use. Its most interesting aspect is that it treats the terrain creation process as composed of simple "device" primitives (e.g., 2D Perlin noise generator, Gaussian blur filter, etc.) which the user connects to form a "machine", a directed acyclic graph (DAG) that will produce the height field when evaluated. The main benefit of this approach over a more traditional fractal/sculpting approach is that, since the series of operations is preserved in the DAG, the user is free to tweak the parameters of any stage of the generation process, rather than having to commit to a particular filter size, random seed, etc. at the outset.

III.1.5.5. Erosion 3D

Erosion 3D [von Werner 1996] is an application for simulating the effects of water erosion on height field terrain. Developed at the Department of Soil Science and Water Protection of the Institut fÃ¼r Geographische Wissenschaften (Institute for Geographic Sciences) in Germany, *Erosion 3D* is intended primarily as an analysis tool, not as a terrain creation tool.

III.1.6. Other Sources for Terrain Information

There also exist a number of community hubs and information portals on the Internet that focus partially or exclusively on the synthesis and use of virtual terrain. These sites are good places to go to find mature, free software, tutorials, technical specifications, and links to other resources.

III.1.6.1. United States Geological Survey

http://www.usgs.gov

The *U.S. Geological Survey* [USGS 2006] is a U.S. government organization dedicated to geographic, environmental and biological research. The USGS maintains standards (i.e., file format specifications) for GIS data interchange, offers downloadable GIS data, and provides a wealth of GIS-related information.

III.1.6.2. GeoCommunity

http://www.geocomm.com

GeoCommunity [Qlinks 2006] is a portal site for the GIS community, and provides information, software, and freely downloadable GIS data for the entire United States, and some other parts of the world.

III.1.6.3. The Virtual Terrain Project

http://www.vterrain.org

The *Virtual Terrain Project (VTP)* [VTP 2006] is an online repository containing a wealth of information and free, cross-platform software, as well as a portal to additional resources. The stated goal of VTP is "to foster the creation of tools for easily constructing any part of the real world in interactive, 3D digital form."

III.1.6.4. Gamasutra

http://www.gamasutra.com

Gamasutra [Gamasutra 2006] is one of the premiere websites of the game developer community, tracking game industry news and providing tutorials on useful techniques for real-time interactive games, including techniques for generating and visualizing terrain.

III.1.6.5. NeHe

http://www.nehe.gamedev.net

Neon Helium Productions (NeHe) [GameDev.net 2006] is another game developer community site, focused on largely on OpenGL, providing articles, tutorials and sample code. It has a number of tutorials on terrain rendering and generation.

III.2. Other Similar Works

In the course of reviewing the existing terrain generation literature, several papers were encountered that were especially similar to aspects of *Terrainosaurus* (in spirit, at least, if not in application).

III.2.1. *Procedural Modeling of Cities*

This paper [Parish & Müller 2001] by Parish & Müller describes *CityEngine*, a procedural approach to city generation (buildings, roads, etc.) using L-systems. The user provides raster maps of the land elevation, bodies of water, and population density, and *CityEngine* constructs a plausible road network and a set of buildings matching these maps, using L-system rules derived from studying actual cities.

CityEngine is similar to *Terrainosaurus* in a number of ways; in fact, parts of *Terrainosaurus*'s generation pipeline are inspired by that of *CityEngine*. In a sense, they are "sister systems", attacking different problems (terrain vs. cities), using different techniques (genetic algorithms vs. L-systems), but with the same objective: extensible, realistic synthesis of large-scale constructs, incorporating both human-designed layouts and realism constraints derived from real-world observations.

III.2.2. *Towards an Understanding of Landscape Scale and Structure*

This paper [Gallant & Hutchinson 1996] by Gallant & Hutchinson investigates the relationship between the resolution of a height field and the kinds of physical features that can be detected within that height field. As their analysis tool, they use a *positive wavelet decomposition* of the height field (intuitively, the inverse of the aforementioned collaging methods (Section III.1.3.4), although the positive wavelet decomposition does not technically have an inverse operation) to break it into features of different scales, from which they draw their conclusions.

While their objective is quite different (they are interested in hydrological analysis), their work bears a resemblance to *Terrainosaurus* in that they are seeking to decompose terrain height fields into features of different scales in order to characterize and understand them.

III.2.3. *SAR Surface Ice Cover Discrimination Using Distribution Matching*

This paper [Gill 2003] by Gill describes an algorithm for computer-aided discrimination between sea ice and open water from synthetic aperture radar (SAR) imagery. In this algorithm, a human identifies a region he believes to be "ice" and another he believes to be "water", and then the computer categorizes the rest of the image as "ice" or "water" using the Kolmogorov-Smimov (KS) and χ^2 statistical distribution matching tests.

This is similar to *Terrainosaurus* in that the statistical characteristics of human-identified terrain types are used to compare samples of terrain for similarity, though *Terrainosaurus* uses a different means of determining the degree of similarity.

III.2.4. *Flexible Generation and Lightweight View-Dependent Rendering of Terrain*

This technical report [Pelton & Atkinson 2003] by Pelton & Atkinson describes a spectral synthesis technique for terrain generation based on real-world GIS example data:

1. Compute a wavelet transform of the example height field
2. From this, construct a frequency histogram of the example height field
3. Generate a new wavelet-space representation of a height field by stochastically sampling the frequency histogram from the example height field
4. Create the spatial representation of the height field by performing the inverse wavelet transform

In this way, they are able to create new height fields similar to a reference height field. They note that their approach is successful at creating unique results with similar roughness characteristics to the original, but that it fails to capture semantic features, specifically riverbeds.

Their approach is similar to *Terrainosaurus* in that both derive new, unique terrain models using example GIS elevation maps.

III.3. Other Topics

Having surveyed the landscape of terrain-related literature, there still remain a few auxiliary topics that must be covered to set the stage adequately for the *Terrainosaurus* algorithm:

- genetic algorithms
- computer vision
- descriptive statistics

III.3.1. Genetic Algorithms

Unfortunately, many of the more interesting problems in computer science fall into the class of *NP hard* problems (for which there are no known polynomial-time solutions), meaning that they become intractably difficult to solve optimally as the size of the problem increases [Cormen et al. 2001]. In many cases, it is not feasible to find a perfect solution, and approximation techniques must be used. The *genetic algorithm (GA)* is one of these techniques.

Genetic algorithms are used in several places in *Terrainosaurus* as a way of replacing the role of human intelligence with a form of artificial intelligence. By pushing the burden of constructing a believable terrain model onto the computer, the human user is relieved of most of the work of terrain generation.

Conceptually, a genetic algorithm is a parallel search algorithm that tries to find better and better solutions to its problem through a process analogous to the Darwinian theory of biological micro-evolution. Single possible solutions within the solution space of the problem are called *chromosomes* (or, alternatively, *individuals*). A *fitness function* provides an evaluation of how good of a solution any particular chromosome is, expressed as a scalar value. As the GA iterates, new potential solutions to the problem are explored by taking the more "fit" of the existing chromosomes and recombining some of their sub-parts (their *genes*)

to create new chromosomes having characteristics of both "parents". By iteratively evolving a *population* of chromosomes (generally of some fixed size), the GA explores the solution space in a parallel fashion. Hopefully, after some reasonable number of iterations, the most "fit" of the chromosomes in the final population will be fairly close to an optimal solution to the problem [Obitko & Skavík 1999].

In order to adapt a GA to solve a particular problem, we must define three basic things:

- a modular, genetic encoding for describing candidate solutions
- crossover and mutation operators to work on this genetic representation
- a fitness function for evaluating the "goodness" of a particular candidate solution

The meaning and implementation of these things will vary according to the problem domain to which the GA is applied.

III.3.2. Computer Vision

Computer vision is the branch of computer science concerned with processing images to recover conceptual representations of the objects present within those images. Computer vision techniques are useful in a number of areas, especially robotics and augmented reality (AR) applications.

In *Terrainosaurus*, feature detection techniques are used in the analysis of terrain models, to find the location, size, and scale of geometric features of the terrain (such as edges, ridges, and peaks).

III.3.2.1. Single-scale Feature Detection

The normal method of feature detection is to design a *detector* for the desired feature, a function that, given a pixel location within the image, returns a scalar *response* indicating how much the pixels in that vicinity resemble the desired feature. The stronger the resemblance, the higher the response will be. This detector is then passed over every pixel in the image, and its response recorded. Pixels containing local maxima of the detector response are kept, and the other pixels are zeroed, a technique called *non-maximal suppression*. The result is a greyscale image with grey or white pixels wherever the feature was found and black elsewhere.

Feature detectors are highly sensitive to the scale of the feature being detected. For example, an edge detector optimized for finding sharp edges between regions will perform poorly if the edges in the image are diffuse (i.e., blurry), especially in the presence of noise. Thus, it is important to tune the detector to the scale of feature being sought.

III.3.2.2. Scale-space Feature Detection

Unfortunately, in many computer vision applications, the scales at which features will appear are unknown. If this is the case, then *scale-space* feature detection may be used to search for features across a range of scales [Lindeberg 1998,1] [Lindeberg 1998,2].

Scale-space feature detection is a relatively straightforward extension of single-scale feature detection, in which locally maximal detector responses are looked for, not only across the (x, y) domain of the image, but also across adjacent scales. Because the algorithm is independent of the actual feature detector matrix

used, virtually any single-scale detector may be adapted to work in scale-space, including "blob" detectors, "edge" detectors, "ridge" detectors, etc.

The major steps in scale-space feature detection are:

1. Generate the scale-space representation of the image by convolving the image with successively larger Gaussian blobs. This produces a 3-dimensional image "cube", in which the added dimension represents scale; as the scale parameter increases, fine-scale image details disappear, leaving only the larger-scale shapes.
2. Calculate the feature detector response at every pixel in the scale-space image.
3. Find the locations within the scale-space image at which the detector response is maximal. Finding the maxima in the x and y dimensions yields the locations of the detected features, while finding the scale at which each is maximized gives an estimate of the size of the feature.

In this way, features present in the image may be recovered and, additionally, classified according to the scale at which they appear. For example, when ridge detection is performed on an image of a crumpled piece of cloth, a tiny wrinkle in the fabric will yield a maximal detector response at a fine scale, and will completely disappear as the scale is increased. In contrast, a gentle fold in the fabric will give some small response at the finest scale, and a much higher response as the scale approaches that of the fold.

If the only benefit of scale-space detectors over single-scale detectors were that features are detectable at multiple scales, this would not be much of an improvement over single-scale detectors (since the same thing can be accomplished by running a single-scale detector multiple times, at several different scales), and not worth the additional cost in memory and computation time. However, since scale-space detectors are able to examine multiple scales *simultaneously*, a "long" feature (such as an edge or ridge) can be recognized as a single feature, even if it varies dramatically in scale along its length. In contrast, a multi-pass, single-scale detector would register multiple, partially overlapping features in this case, rather than a single feature of varying scale. As a result, a scale-space detector will report fewer features (and generally, more meaningful ones).

III.3.3. Descriptive Statistics

Descriptive statistics is the branch of statistics dealing with identifying patterns and trends within collected samples of data. In *Terrainosaurus*, measured statistics are used as a basis for comparing terrains for similarity: intuitively, the more similar the statistical characteristics of two terrains, the more "alike" they are. In particular, we are interested in the following four statistics:

- the sample mean
- the sample standard deviation
- the sample skewness
- the sample kurtosis excess

The following diagrams of these statistics depict distributions similar in shape to a Gaussian (normal) distribution. This is because some of these measures (the skewness and kurtosis excess) are defined with

reference to the Gaussian distribution, such that the Gaussian distribution has a value of zero. Thus, Gaussian-like distributions are convenient for conveying an intuitive sense of the statistic. This should not be taken to mean that these measures apply only to the Gaussian distribution; they are defined wholly in terms of raw moments and central moments of the distribution, and can be applied to any distribution. The degree to which these measures deviate from zero gives an indication of how non-Gaussian the distribution is.

III.3.3.1. The Sample Mean

The best-known statistic is the *sample mean*, or unweighted average (Equation III.2). Intuitively, this is the "center" value of a sample distribution (Figure III.8) [Weisstein 2004].

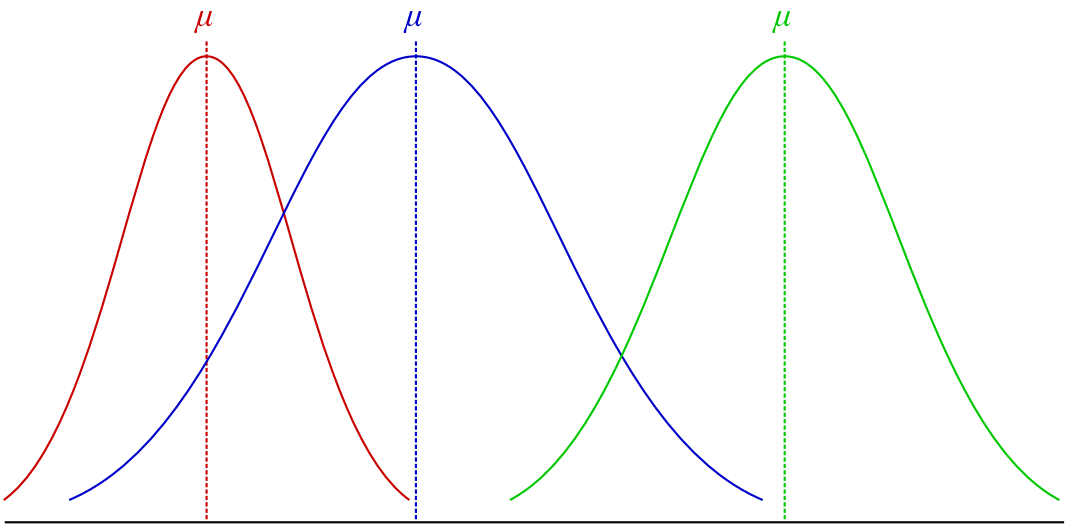

Fig. III.8. Mean. The mean of a statistical distribution is a first-order measure and is the value around which the distribution is centered.

$$\mu = \sum_{i=1}^{N} \frac{1}{x_i} \qquad \text{(III.2)}$$

III.3.3.2. The Sample Standard Deviation

A second useful statistic is the *sample standard deviation*, σ, a non-negative value measuring the degree of variability of a sample distribution (Equation III.3). A low standard deviation indicates that the sample values are tightly clustered about the mean (Figure III.9) [Weisstein 2003].

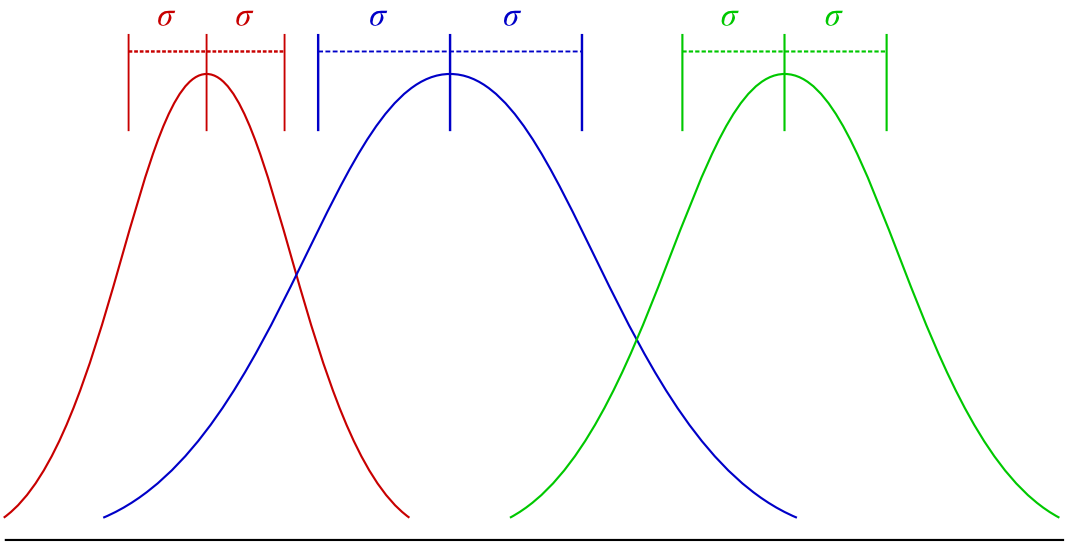

Fig. III.9. Standard Deviation. The standard deviation of a statistical distribution is a second-order measure describing the degree to which the distribution is "spread out".

$$\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^2} \qquad \text{(III.3)}$$

III.3.3.3. The Sample Skewness

A third, lesser-known statistic is the *sample skewness*, γ_1, which measures the degree of asymmetry of a sample distribution (Equation III.4). A skewness of zero indicates that the distribution is perfectly symmetrical about the mean, whereas a positive or negative skewness indicates that the distribution is skewed to the left or right of the mean, respectively (Figure III.10) [Weisstein 2005].

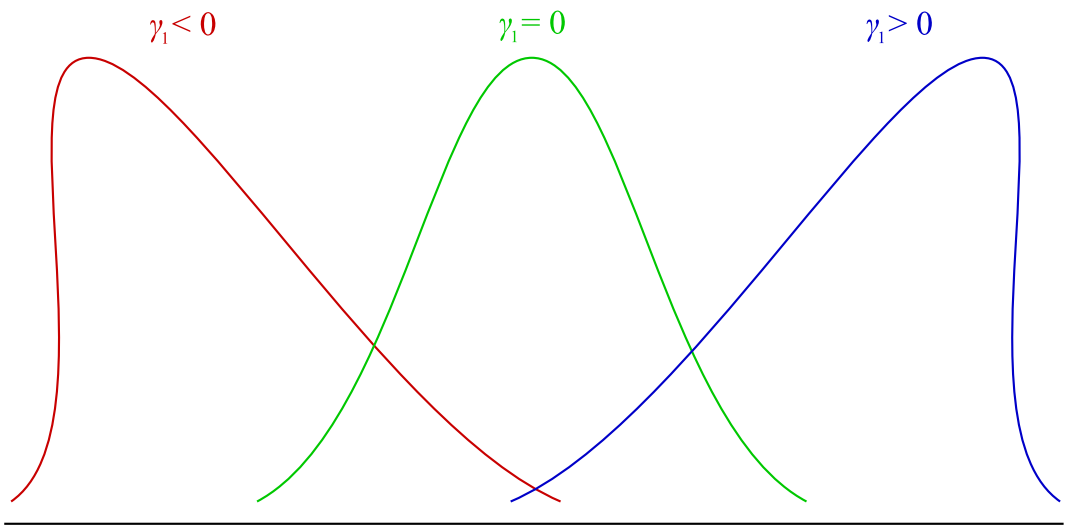

Fig. III.10. Skewness. The skewness of a statistical distribution is a third-order measure describing the amount of asymmetry of the distribution. Values less than zero indicate that the distribution is skewed to the left of the mean, and values greater than zero indicate that the distribution is skewed to the right.

$$\gamma_1 = \frac{k_3}{k_2^{3/2}} \qquad \text{(III.4)}$$

This formulation of the skewness is in terms of the second and third *k-statistics*. The first four k-statistics are given in Equation III.5 [Weisstein 2002].

$$k_1 = m_1$$

$$k_2 = \frac{n}{n-1} m_2$$

$$k_3 = \frac{n^2}{(n-1)(n-2)} m_3 \qquad \text{(III.5)}$$

$$k_4 = \frac{n^2\left[(n+1)m_4 - 3(n-1)m_2^2\right]}{(n-1)(n-2)(n-3)}$$

where n is the sample size and the m_r's are the r^{th} sample central moments, given by Equation III.6 [Weisstein 2003].

$$m_r = \frac{1}{n}\sum_{i=1}^{n} (x_i - \mu)^r \qquad \text{(III.6)}$$

III.3.3.4. The Sample Kurtosis Excess

A fourth statistic is the *sample kurtosis excess*, γ_2, often simply called the *sample kurtosis*, which measures the degree of peakedness of a sample distribution (Equation III.7). The normal distribution has a kurtosis of zero, whereas a more peaked distribution has a positive kurtosis and a flattened distribution has a negative kurtosis (Figure III.11) [Weisstein 2004].

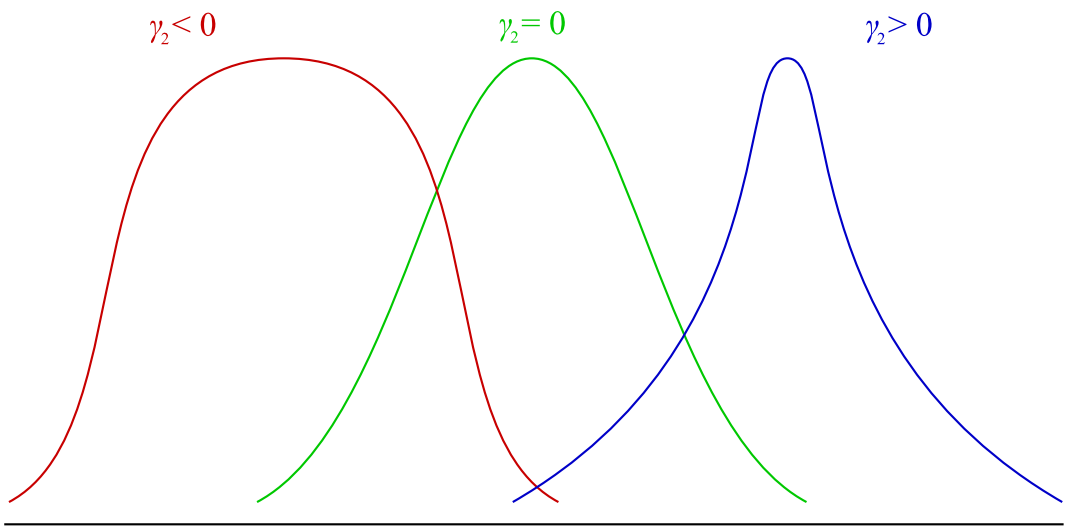

Fig. III.11. Kurtosis. The kurtosis excess of a statistical distribution is a fourth-order measure describing the amount of "peakedness" of the distribution. Values less than zero indicate that the distribution is more "flat" than a normal distribution, and values greater than zero indicate that the distribution is more peaked than a normal distribution.

$$\gamma_2 = \frac{k_4}{k_2^2} \qquad \text{(III.7)}$$

The k_i's are the second and fourth k-statistics, as defined above in Equation III.5.

CHAPTER IV

METHODS

In this chapter, I discuss the *Terrainosaurus* algorithm at a conceptual level, including its overall strategy and major steps, explaining the rationale for important design decisions. The gritty details of implementing the algorithm are glossed over here, and presented in detail in the chapter on implementation (Chapter V). These topics will be covered in the following order:

- prerequisites—design strategy and basic tools
- how the algorithm works from the user's perspective
- how the algorithm works from the computer's perspective

IV.1. Prerequisites

Before delving into the details of the algorithm, there are several things that must be nailed down first, outlining the basic strategies underlying *Terrainosaurus*:

- how ease of use will be ensured
- how the terrain generation problem will be attacked
- what data structure will be used to represent the terrain
- how level of detail will be addressed

IV.1.1. Usability Considerations

Since one of our major stated objectives is to create something easy to use, this aspect deserves specific consideration. While a full-scale user study would be helpful in this respect, this is outside the scope of this thesis. Nevertheless, there are basic user-interaction decisions we can make that will, *prima facie*, promote usability.

The crucial observation to make, in this regard, is that not all possible parameters and constraints for a terrain generation algorithm are of the same nature. For our purposes, it will suffice to distinguish between three types:

- those that are essentially *free*, meaning that the user can manipulate them for artistic purposes (within limits, of course), without damaging the apparent realism of the terrain—examples of these include the size of a mountain range, and the placement of lakes and mountain peaks
- those that are critical for maintaining the apparent realism of the terrain—examples of these include the characteristic fine-scale detail of different terrain types, and the smoothness of transitions between different terrain types
- everything else—the miscellaneous parameters that influence things like algorithm running time, memory usage and quality of results

In everything, our goal is for *Terrainosaurus* to be thoroughly intuitive. For each of these three types of parameters/constraints, a different method of user interaction is most appropriate.

IV.1.1.1. Visual Authoring

Those aspects of the terrain that can be freely manipulated for artistic effect without impacting the realism of the terrain belong under the control of the user. Since most of these are spatial and/or visual, a visual authoring environment is the obvious choice, allowing the author to interact with a visual representation of his work, and to refine it incrementally until it reaches his satisfaction. Human beings process visually-presented information quite efficiently, and we would expect this to be especially true of information that is already spatial in nature, such as terrain. Furthermore, the digital art community works almost exclusively with 2D and 3D visual authoring tools (e.g., *Photoshop*, *3D Studio*, *Maya*), and will be much more likely to adopt a tool that allows them to work in this fashion.

IV.1.1.2. Example-based Design

In contrast to the previous section, those aspects of the terrain that are crucial for maintaining its realism are not especially suitable candidates for user authoring. These are likely to be more highly constrained, and possibly in ways that are not well understood or would be tedious or impossible to replicate by hand (e.g., satisfying a certain statistical distribution). Such things are perhaps better left for the computer to solve, freeing the user to do what the computer cannot: imagine and create.

Still, the user needs to be able to control the characteristics of the generated terrain in *some* way. A *design by example* approach is a graceful method of specifying these characteristics: rather than requiring the user to comprehend the complexities that give a particular flavor of terrain its identity, we simply ask him to provide one or more examples that exhibit the characteristics he would like to see replicated in his terrain. In this way, the burden of reverse-engineering terrain is transferred off of the user and on to the software, and the algorithm becomes a true "black box" from the user's perspective, requiring almost no domain knowledge to be used.

Two additional benefits come from taking such an approach. First, the algorithm is easily extensible, since the user can simply add new examples of terrain to achieve new effects. Second, the algorithm can be "upgraded" completely transparently to the user—because of the simplicity of the interface, improvements can be made to the algorithm without the user having to learn to use new parameters, or even needing to know that anything has changed.

IV.1.1.3. Miscellaneous Controls

Naturally, there are other parameters and constraints that do not fit well into either of the above categories. If these values cannot be determined automatically, then other methods of presenting them to the user will be required, such as standard GUI controls (sliders, buttons, etc.).

Having additional, non-visual parameters for the user to set is not, in itself, a problem. The important thing is not for all parameters to be expressible visually, but for all parameters to be *intuitive*: the meaning of the parameter should be easily understood by a user who has no comprehension of the inner workings of the algorithm. As a result, the user should have a fair idea of what the effect of changing each parameter will be, even before the change is made.

One example of a non-visual parameter that is nonetheless intuitive is a "quality" parameter that adjusts a tradeoff between the amount of processing time spent and the realism of the result produced. Even though this tradeoff does not have an obvious visual meaning, humans already understand the concept of spending additional time working on a task to achieve a better result.

As an example of the sort of parameters that we want to *avoid* presenting to the user, consider the well-known Perlin Noise [Perlin 1996] function. While the usefulness of Perlin Noise is indisputable—it has been used to great effect for creating believable imitations of a wide variety of natural phenomena—it has a significant drawback: it requires a non-trivial amount of domain knowledge (or else brute force, trial-and-error experimentation) to achieve a desired result. To a novice user, the parameters (e.g., octaves, turbulence, persistence) are "magic": it is difficult to gain an intuition for the effects of tweaking one of them without first understanding how Perlin Noise works.

IV.1.2. Terrain Generation Strategy

Obviously, there is no unique "correct" answer to the problem of generating terrain; instead, there are infinitely many "good" answers, infinitely many more "bad" answers, and everything in between. Thus, the quest to make believable terrain can be viewed as a search over an infinitely large solution space, in which some solutions are quantitatively better than others.

A number of methods exist for finding good approximate solutions to problems for which an exhaustive search is infeasible or (as in the case of terrain generation) impossible. These methods vary in performance and generality. In *Terrainosaurus*, we have chosen to use a genetic algorithm for this purpose ([Obitko & Skavík 1999]). The major benefits of GAs (for our purposes) are:

- they are randomized, rather than deterministic; as a result, they are capable of some degree of innovation and can produce an effectively unlimited number of unique terrain models. Additionally, this randomness can enable a properly-tuned GA to escape from locally "best" solutions in order to find other, better solutions.
- they are extremely general: a GA treats its problem as a "black box", and need not understand the complex effects of the changes it makes, only caring about the results of the fitness evaluation. Because of this, a GA can unite a diversity of orthogonal or competing constraints: anything that can be incorporated into the fitness evaluation. This makes a GA an attractive development tool, as it leaves open a straightforward means of adding new constraints as they become necessary.

IV.1.3. Choice of Data Structure

The terrain data structure used in *Terrainosaurus* is the height field [1] (Section III.1.2.1). This decision makes sense in light of two facts:

1. most current, real-time applications of terrain use height fields

[1] Throughout the remainder of this thesis, the terms *height field* and *terrain* will be used synonymously, unless otherwise noted.

2. GIS elevation data is most widely available in this form; as a result, the data has *already* been sampled to a finite resolution and has lost its ability to resolve features like overhangs and caves. Thus, there is no further penalty for using height fields, and no additional benefit to using other representations that do not share these restrictions

IV.1.4. Level of Detail Strategy

From an LOD perspective, the terrain generation problem can be understood as the problem of constructing a height field having the following characteristics:

- at each LOD, appropriate, characteristic features are present in the terrain, in correct proportions
- the entire set of LODs is *coherent*, meaning that features present in coarse LODs continue to exist at the same locations in the finer LODs

As was stated in Section III.1.4.3, *Terrainosaurus* approaches terrain construction as a multi-LOD, progressive refinement process. As was also mentioned earlier (Section III.1.4.4), one of the more significant goals of *Terrainosaurus* is to offer a not-necessarily-fractal answer to the question of what the terrain ought to look like at any given LOD.

IV.2. The User's Perspective (What It Does)

Since ease-of-use is one of the primary objectives of our approach, it is worth taking some time to walk through how our approach looks from the perspective of a user. This will also serve to create the context for the more technical discussion to follow. Consider a hypothetical user, a set designer for a large computer animation studio. Among this user's routine tasks is the construction of outdoor, virtual worlds for the commercials, feature films, and video game content that his company develops.

From the user's perspective, terrain generation with *Terrainosaurus* involves three discrete phases:

- assembling a terrain library
- authoring a terrain map
- generating a height field

IV.2.1. Terrain Library Assembly

The user's first step is to assemble a *terrain library*, the "palette" with which he will later "paint" his terrain models. His task in this phase is to define the taxonomy of *terrain types* he wants to use. Terrain types are logical abstractions, semantic classes of terrain, as a human would think of them (things like "steep mountains", "sandy beach", "rocky desert", "plains", etc.) and are created by providing one or more *terrain samples*, example height fields representative of these classes. These examples will normally come from providers of real-world GIS data (e.g., [Qlinks 2006]), though they could potentially come from other sources as well. Once the user has populated his library with example terrains, *Terrainosaurus* analyzes the library (see Section IV.3.2) in order to identify unifying characteristics for each terrain type, measurable quantities that are similar across all of the example terrains within a single terrain type (and thus, which are likely to

contribute to the user's perception of those terrains as being related). These example terrains will also serve as the raw material for constructing new height fields in the third phase (see Section IV.3.3).

Because terrain types are correlative to human mental categories, they are effectively unlimited in number, and will vary according to the user. No matter how many terrain types are identified, there will always be someone who wants something just a bit different. At the same time, the choice of terrain types is not wholly arbitrary—it is important that example terrains for the same terrain type *are* truly similar in some way, or else the terrain analysis process will be hindered, because *Terrainosaurus* will be unable to identify meaningful unifying characteristics for that terrain type.

The user's terrain libraries can be as coarse- or as fine-grained as he wishes, depending on his needs, and he might have separate libraries for different purposes. For example, he might define many variations of "mountains" in order to finely control the kinds of features that appear throughout a mountainous terrain model, or he might have only a few, highly different, general-purpose terrain types for building more heterogeneous worlds.

While the construction of the terrain library is likely to be a bit tedious, it is something that will be done only infrequently, as once a terrain library has been assembled, it can be reused indefinitely, and extended incrementally as additional terrain types are needed. Section VII.5 discusses potential avenues for future research that might help alleviate this burden on the user.

IV.2.2. Map Design

The user's primary design task when using *Terrainosaurus* is to create the layout of his terrain. He does this visually, creating a 2D, vector-drawn map of the terrain using a CAD-style interface (Figure IV.1). By representing the map as a vector drawing rather than a raster image, we avoid committing to a particular resolution for the map, enabling the user to defer the decision of what LOD to generate until he actually generates it, allowing him to produce multiple LODs from a single map. The map is made up of one or more polygonal regions of terrain, which may be of arbitrary sizes and shapes. Each region is assigned one of the terrain types from the library. Through sketching out such regions and tweaking their shapes, the user can express the approximate layout he desires for his terrain, using a modeling paradigm that is well-established and intuitive.

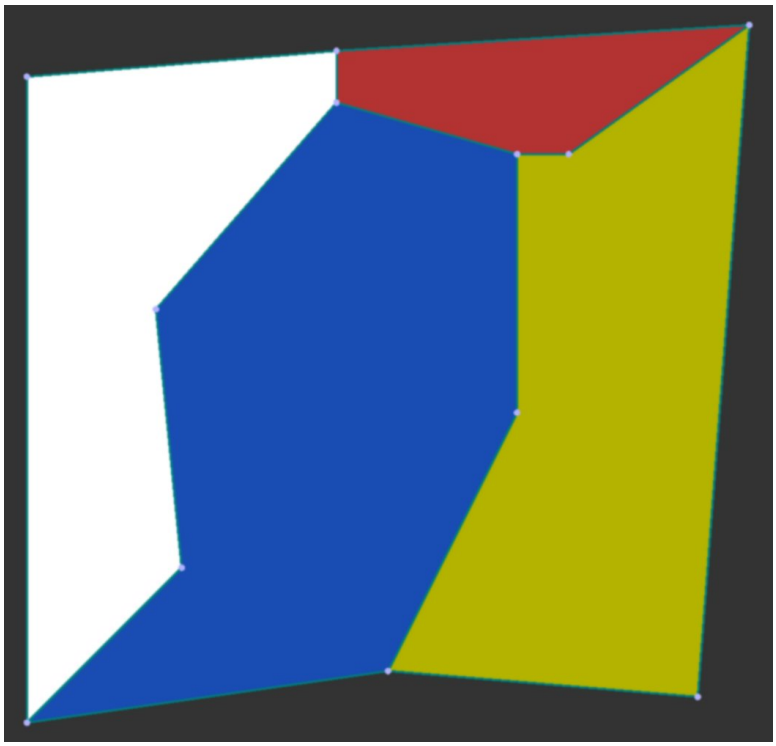

Fig. IV.1. The Map Authoring Interface. The user describes his desired terrain layout by authoring a 2D map, which specifies the location, size, and shape of regions of terrain, each of which may have its own terrain type. A CAD-style interface allows the user to accomplish this using a familiar and intuitive design paradigm.

IV.2.2.1. Boundary Refinement

The use of polygons to represent regions in the terrain type map has the advantage of simplicity: operations on polygons are well-defined and well-understood, and provide a straightforward way for a user to work with the map. However, while linear shapes are useful tools for modeling, their linearity is also a drawback: since polygons are restricted to having straight edges, curved or irregular shapes may require large numbers of small segments to be adequately approximated. If the approximation is not fine enough for the particular application, the unrealistically straight edges will be visible. In our case, this means that if the length of a linear region boundary is large compared to the resolution of the generated height field, this boundary may be reflected visibly in the generated terrain as an unnaturally straight transition between two terrain types; the longer the boundary, the more noticeable this is likely to be. If the height field is colored or texture-mapped according to terrain type, this effect is greatly magnified (Figure IV.2). Therefore it is important that the boundaries between regions be of sufficiently fine scale.

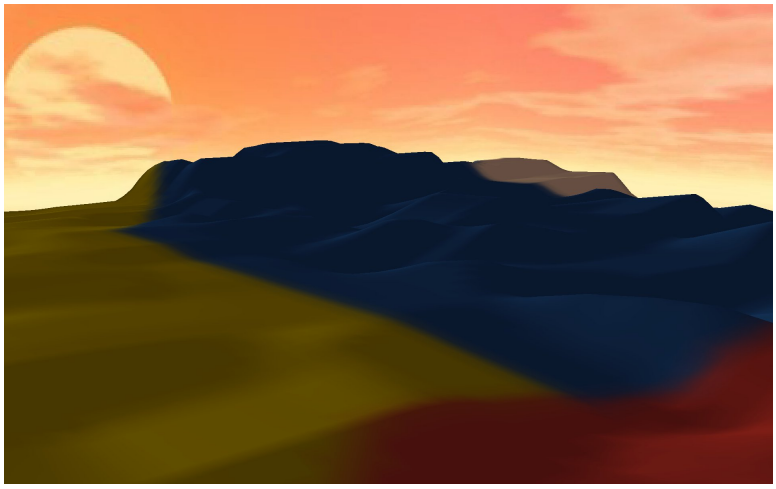

Fig. IV.2. Artifacts Resulting from Linear Region Boundaries. Seams between adjacent regions can appear as unnatural-looking artifacts in the generated height field if the region boundaries are long and linear. This effect is made all the more obvious by coloring or texture-mapping the regions.

Creating such fine-scale boundaries by hand would be quite tedious, and it is much to be preferred, from a user-experience standpoint, that this task be automated. We do this by providing a *boundary refinement* operation, which non-destructively replaces a straight boundary from the user's polygonal map with a new boundary that follows the same approximate path as the original, but with a meandering, irregular shape made up of many short segments (Figure IV.3). The user influences the shape of this boundary by adjusting a *smoothness* parameter. The resulting *refined boundary* will be used instead of the original linear boundary when the height field is constructed, producing a more believable, irregular transition between the regions (Figure IV.4). The gory details of this operation are described later, in Section IV.3.1. Using this operation, the user is enabled to sketch his map using rough, simple shapes, and then to fill in fine-scale boundary detail automatically. With this paradigm, even an inexperienced user should be able to create a simple but believable map in just a few minutes.

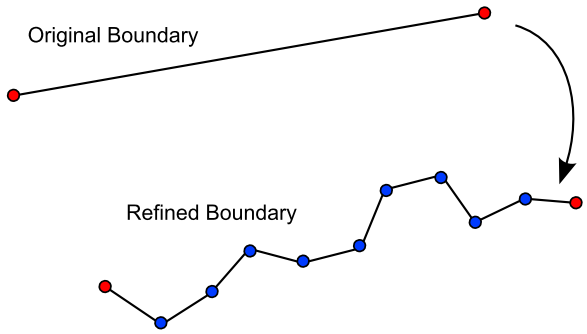

Fig. IV.3. The Boundary Refinement Operation. The boundary refinement operation replaces a long, linear segment of a region boundary with a string of short segments connecting the same endpoints.

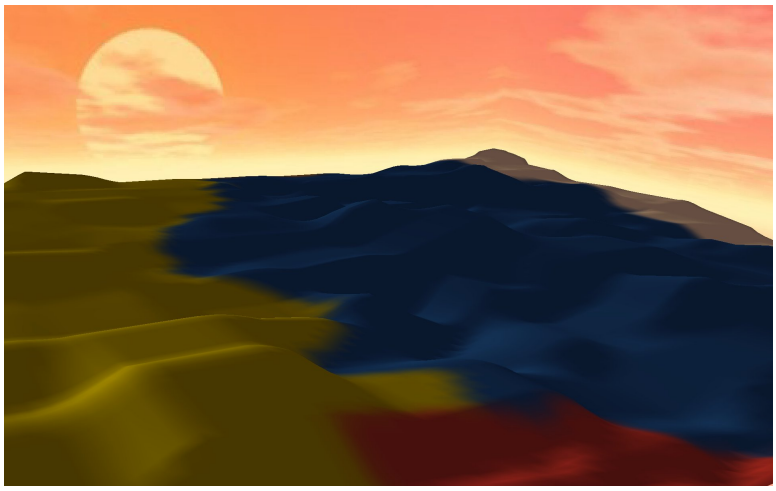

Fig. IV.4. A Refined Boundary Avoids Unnatural-looking Artifacts. With long, linear boundaries replaced by irregular, meandering boundaries, such boundary artifacts are no longer evident in the generated height field.

IV.2.3. Height Field Construction

Once satisfied with the map he has created, the user selects a rectangular chunk of the map, chooses the desired level of detail, and launches the height field generation algorithm. This third phase of the process is computationally expensive (and therefore, slow), but is also entirely automated: once started, no further input from the user is needed, and he can go on to other tasks.

In this phase, *Terrainosaurus* generates a height field at multiple LODs, beginning from the coarsest possible LOD and continuing up to the target LOD requested by the user. At each LOD, the height field is built such that the features present in each region resemble those of the corresponding terrain types at that LOD.

A secondary, optional by-product of generating each LOD is a rasterized version of the user's map, an image file with the same raster dimensions as the generated height field, in which each pixel represents the terrain type of the corresponding grid cell in the height field. This information could be loaded and used by another program to do terrain-type-based postprocessing, such as generating a texture map for the height field, or simulating erosion of the terrain surface using terrain-type-specific soil characteristics.

IV.3. The Computer's Perspective (How It Works)

Having covered the fundamental concepts in *Terrainosaurus* and outlined the general steps of the algorithm, all that remains to be done is to peer inside of the "black boxes" in the above outline, and fill in the details. In this section, I discuss the following aspects:

- the boundary refinement algorithm
- height field analysis & comparison
- the height field generation algorithm

IV.3.1. Boundary Refinement

The boundary refinement operation is the first algorithmic aspect of *Terrainosaurus* that I discuss, and also the first problem that we solve using a genetic algorithm.

IV.3.1.1. Overview

The boundary refinement operation is essentially a randomized subdivision operation. It takes as input a single line segment of arbitrary length and produces a piecewise-linear curve with the same starting and ending points as the original segment, but made up of N segments (connecting $N+1$ points), where N is proportional to the length of the original segment. This curve is then translated, rotated and scaled as needed to line it up with the original boundary's end points. Since we will ultimately be transforming the entire curve anyway, we can, without loss of generality, think of the original boundary as lying along the positive x axis, starting from the origin. All of the diagrams in this section reflect this convention.

A useful result of applying this operation to each boundary in the map is that the segment length throughout the entire map is roughly uniform (recall that N, the number of segments, was said to be proportional to the length of the original boundary). If this length is chosen to be small enough, relative to the resolution of the height field that will be generated, no straight boundaries between terrain types will be evident in the generated height field, nor in any texture maps applied to the height field.

The segment length (and thus, the number of segments) may be calculated from the resolution of the target height field, according to the spatial version of the Nyquist limit: considering the height field as sampling the boundary, we know that the height field cannot resolve boundary details finer than half its spatial resolution (Equation IV.1).

$$L \leq \frac{\text{cells}}{\text{meter}} \times 2 \qquad \text{(IV.1)}$$

However, since the generated boundary may need to be scaled somewhat to match the endpoints of the original boundary, we incorporate a "slop factor" of two, resulting in Equation IV.2.

$$L \leq \frac{\text{cells}}{\text{meter}} \qquad \text{(IV.2)}$$

The shape of the generated boundary is subject to the constraints imposed by the user. The user can control how sharp the angles between successive segments are allowed to be by adjusting the *smoothness* parameter, S, which can vary from 0.0 (very rough) to 1.0 (very smooth).

IV.3.1.2. Genetic Encoding

The first step in casting the boundary refinement problem as a genetic algorithm is to define a suitable genetic encoding for a boundary. In the encoding we selected, a chromosome has N genes, one for each segment in the resulting boundary, and each gene contains a real-valued angle in the range $[-\pi, \pi]$ indicating the *relative* change in direction of the corresponding segment with respect to the one preceding it (Figure IV.5). A positive angle indicates a turn to the counter-clockwise direction, while a negative angle indicates a turn to the clockwise direction. A differential angle of zero indicates that the segment is traveling in the same direction as the previous segment. The angle of the first segment is defined with respect to the *x* axis, since it has no preceding segment.

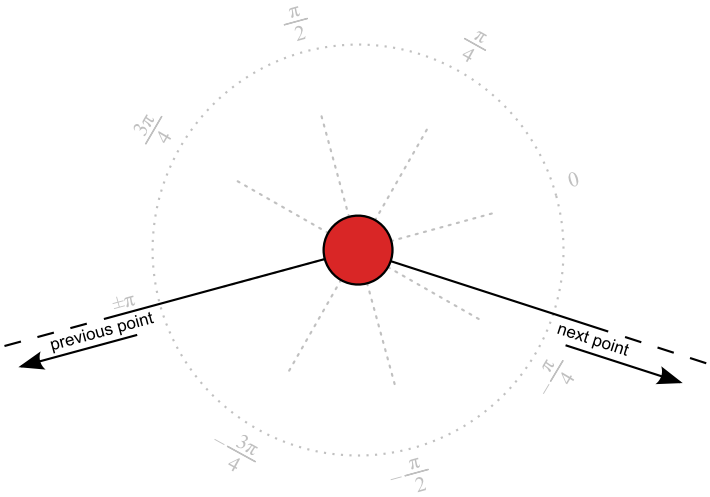

Fig. IV.5. The Encoding of a Gene in the Boundary GA. A boundary's genetic encoding is the sequence of relative angles between successive segments. An angle of zero indicates no change in direction.

An encoding such as this is advantageous in that it represents the shape of the boundary in terms of a *local* property of the curve (the differential angle). This is a convenient representation, since the property that we are trying to optimize (the curve smoothness) is itself a local property. Furthermore, the curve smoothness can be calculated using only the angles, without the need to convert to Cartesian coordinates, with the result that the fitness evaluation can be done quite cheaply, in terms of processing time.

Another important consequence of this encoding is that the relationship between the gene angle values and the 2D Cartesian points to which they are decoded depends *on the entire chromosome*: since each angle encoded in a gene is specified relative to the preceding line segment, a change in the angle of one gene will affect the location of not just the next point, but of *every* subsequent point. As a result, the end point of the boundary cannot easily be held fixed, since mutations earlier in the sequence will tend to move it. This is one reason that we must transform the curve at the end of the algorithm, rather than simply constructing the curve "in place".

A downside to this encoding is that it is possible for the boundary to double back on itself, which causes several problems. We prevent this by requiring that the *absolute* angle (i.e., the angle with respect to the *x* axis) cannot exceed a certain maximum angle at any point on the boundary (Figure IV.6). This is discussed in more detail in Section VI.1.

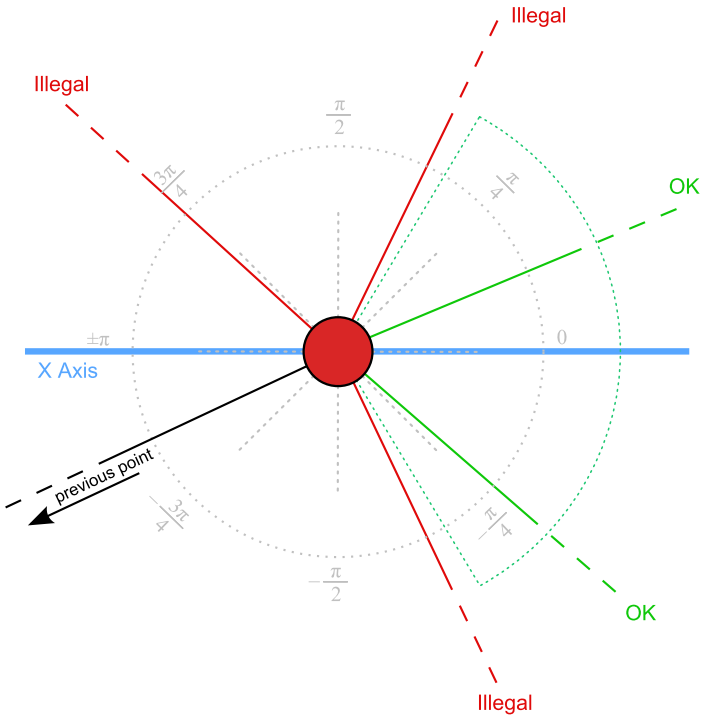

Fig. IV.6. The Absolute Angle Limit. A constraint is placed on the maximum absolute angle that a segment can have with respect to the reference axis, in order to prevent the boundary from doubling back and intersecting with itself.

IV.3.1.3. Genetic Operators

To mutate and cross-breed the chromosomes, we use the standard GA crossover and mutation operators. Mutating a gene corresponds to changing the degree of "bend" between two consecutive segments (Figure IV.7).

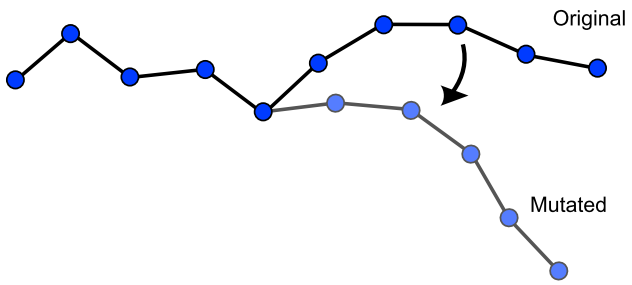

Fig. IV.7. The Boundary GA Mutation Operator. Mutation of a boundary segment's gene changes the amount of "bend" between that segment and its predecessor.

Crossing two chromosomes is the equivalent of cutting each boundary in the middle of one of the segments and exchanging the pieces (Figure IV.8).

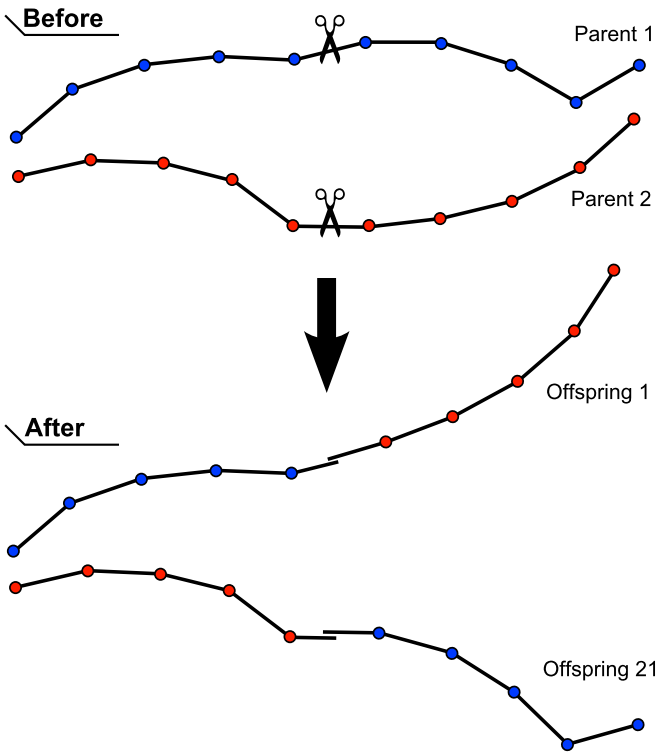

Fig. IV.8. The Boundary GA Crossover Operator.

IV.3.1.4. Fitness Evaluation

A chromosome is "fit" if the boundary curve it represents satisfies the smoothness constraint placed on it. The smoothness of a single gene is calculated according to Equation IV.3. When S is near 1, this equation is linear, favoring angles near zero and penalizing sharper angles. For lower values of S, the equation becomes more sinusoidal, favoring sharper angles (Figure IV.9). The constant 1.1 in this equation controls the horizontal offset of the sinusoid's peak, and is somewhat arbitrary.

$$F = \sin\left((\pi - \alpha)\,(1.1 - S) \right) \qquad \text{(IV.3)}$$

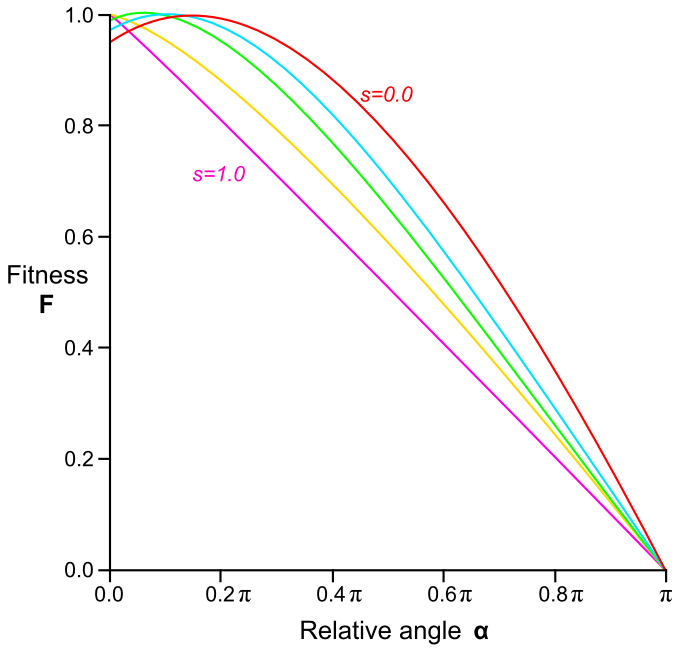

Fig. IV.9. The Smoothness Fitness Function for Several Values of S.

The fitness of the entire chromosome is then calculated as the mean fitness across all of the genes.

IV.3.1.5. Decoding the Result

Once the GA has completed, we have a "best" boundary, encoded as a series of floating-point, relative angle values. The final step in this operation is to decode this chromosome into a series of 2D points ($\mathbf{P}_0$ to $\mathbf{P}_N$) connecting the end points of the original boundary. This is done in a relatively straightforward manner:

1. The initial point $\mathbf{P}_0$ is placed at the origin.
2. Subsequent points $\mathbf{P}_1$ to $\mathbf{P}_N$ are calculated from the previous point, using the recurrence relation Equation IV.4.

$$\mathbf{P}_i = \mathbf{P}_{i-1} + \mathbf{P}\,(\cos\alpha_{i-1},\ \sin\alpha_{i-1}) \qquad \text{(IV.4)}$$

3. The end points of the generated curve will not, in general, line up with the end points of the original boundary. Therefore, we construct an affine transformation matrix that translates, rotates and scales the points such that the first and last points line up exactly with the start and end points, respectively, of the original boundary. This transform is constructed in the following way:

a. Let V_o be the vector connecting the start and end points of the original boundary, and V_r be similarly defined for the refined boundary.

b. The transform is the matrix $X = TRS$, with:

T	the 2D translation along the vector from the origin to the start point of the original boundary
R	the 2D rotation by the angle between V_o and V_r
S	the 2D scaling by the ratio $\frac{\lvert V_o \rvert}{\lvert V_r \rvert}$

At this point, the original boundary has been replaced with a suitable, refined version, and the operation is complete.

IV.3.2. Terrain Analysis & Comparison

The second algorithmic aspect I discuss is the way in which *Terrainosaurus* analyzes and compares height fields. The ability to recognize geometric and statistical similarities between height fields is central to *Terrainosaurus*'s design-by-example paradigm, allowing much of the labor of the terrain construction process to be transferred off of the user and onto the computer.

IV.3.2.1. Analyzing a Single Height Field

The ultimate goal of *Terrainosaurus* is the creation of terrain models that, to a human viewer, are recognizable, plausible reproductions of the kinds of terrain that the user supplied as inputs. Therefore, the central question that must be asked is this: what gives a "kind" of terrain its identity in the mind of the viewer? A comprehensive answer to this question would involve aspects of a number of disciplines, including geology, ecology, linguistics, and human cognition, and is certainly outside of the scope of this research. Nevertheless, for our purposes, we must arrive at a partial answer to this question, one that can be quantified in terms of height field geometry.

Intuitively, several ways of characterizing the geometry of a height field seem to be reasonable candidates for terrain type analysis. First, there are the elevations and slopes in the height field (i.e., the zeroth and first derivatives of the height field surface). Everyday experience tells us that different kinds of terrain often have very different elevation ranges and steepnesses; desert terrain, for example, is generally rather flat, while mountainous terrain can be extremely steep, even completely vertical in places. Furthermore, some kinds of terrain occur at characteristically different altitudes (it would be unusual indeed to find a sandy beach at 10,000 feet of elevation!). Empirical investigation of a number of example terrains indicated that the statistical distributions of elevation and slope tended to be similar between height fields of the same terrain type (Section VI.2.1).

Another way of quantitatively characterizing a height field is by the presence and size of certain identifiable "features": things like peaks, ridges, cliffs, rivers and gorges. Again, experience teaches us to expect

mountain ranges to have more pronounced peaks and ridges than most other kinds of terrain, and that plains areas contain primarily smooth, relatively flat ground. Empirical investigation also indicated that edge statistics tended to be similar within the same terrain type.

This list of height field characteristics is by no means exhaustive: a number of other measures seem worthy of future investigation as potential ways of terrain characterization and may ultimately turn out to be more effective for comparing terrains (several possibilities are discussed in Section VII.8). Even so, the characteristics already mentioned are sufficient starting material for a similarity function.

IV.3.2.2. Comparing a Height Field to a Reference Terrain Type

Having observed that real-world examples of the same terrain types appear to exhibit similarities in their statistical behavior, we need a way of quantifying this similarity. Conventional statistical hypothesis tests, such as χ^2, the KS test, and the t test were considered, but ultimately rejected as not adaptable enough. First of all, it is not enough just to test whether the means of two distributions come from the same population: the mean is an ambiguous indicator of terrain type (Gill remarks that the same is true when distinguishing between ice and sea water in SAR imagery [Gill 2003]), and also tells us nothing about the shape or roughness of the terrain. Second, we need to be able to accommodate *multiple* reference terrain samples; it is not clear how to adapt a KS test, for example, to use multiple reference distributions. Thus, a new kind of test is needed.

Towards this end, we introduce the concept of *Gaussian curve projection*, a technique for comparing unbounded scalar values to determine the *similarity* of a *test value* to one or more scalar *reference values*. With this as a building block, we define a *statistical distribution similarity measure* for comparing a test distribution (e.g., of elevation values) to one or more reference distributions. Finally, we define an *aggregate similarity measure* combining the distribution similarities to form an estimate of the overall similarity between a height field and one or more reference height fields (i.e., the height fields composing a terrain type).

Gaussian Curve Projection

Gaussian curve projection is a simple means of transforming an arbitrary scalar *test value* into a bounded *similarity measure*, given one or more scalar *reference values* against which to compare the test value. The test and reference values may be either bounded or unbounded, but the resulting similarity measure is guaranteed to fall in the range $[\,0,\,1\,]$, with values near one indicating "very similar" and values near zero indicating "very dissimilar".

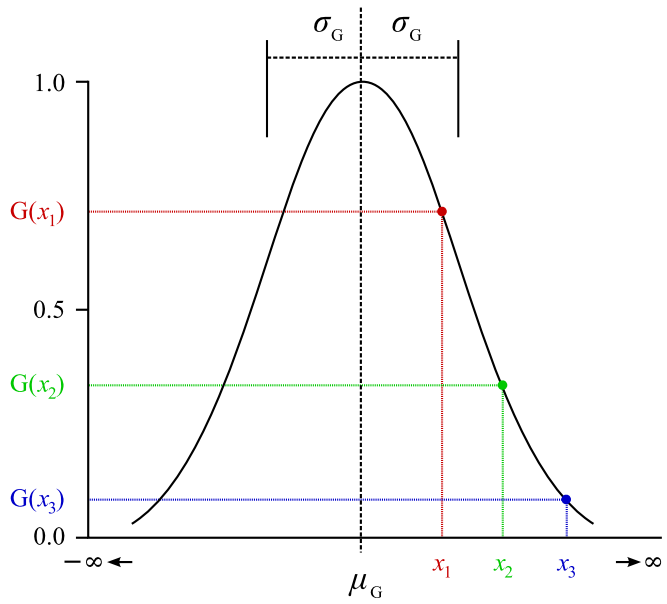

Fig. IV.10. Gaussian Curve Projection. Gaussian curve projection is a flexible technique for transforming bounded or unbounded values into the [0, 1] range, by evaluating a Gaussian function (with a suitable mean and standard deviation) at those values.

This is accomplished by constructing a Gaussian curve based on the reference values, and then finding the projection of the test value onto that curve (Figure IV.10). The curve has a height of one unit at its peak, has a center value μ_G equal to the mean of the reference values, and has a standard deviation σ_G chosen "appropriately" (how σ_G should be determined is discussed later). The resulting function (Equation IV.5) is capable of handling any scalar value, positive or negative, yields a value of one when evaluated at $x = \mu_G$, and yields values asymptotically approaching zero as x diverges from μ_G, with the rate of falloff governed by the value of σ_G.

$$G(x) = e^{\frac{(x-\mu_G)^2}{-2\sigma_G^2}} \qquad \text{(IV.5)}$$

The only missing piece in the above formulation is the determination of σ_G, which I have saved for last because it deserves a slightly longer discussion. This value behaves as a sort of tolerance, controlling how wide a range of values around μ_G is considered acceptable. There is no "one size fits all" formula for setting

this parameter; instead, the parameter must be set in a manner appropriate to the context in which Gaussian curve projection is being used. Here are some suggestions for determining σ_G:

- If the values being compared are bounded, σ_G may be chosen as some fraction of the total possible range of values.
- If the number of reference values is sufficiently large, then σ_G can be chosen to be the sample standard deviation of the reference values. This has the drawback that the resulting function will report some of the reference values themselves as having low similarity. This is likely not what is desired, leading to the next suggestion.
- If the number of reference values is sufficiently large, then σ_G can be chosen such that all reference values evaluate to a similarity value greater than or equal to some *baseline similarity value*, G_B (Figure IV.11). Given a choice of G_B, σ_G may be determined according to Equation IV.6, which is obtained by rearranging Equation IV.5. This guarantees that all reference values will be reported as having high similarity, and results in a larger value of σ_G than would be produced by the previous formulation.

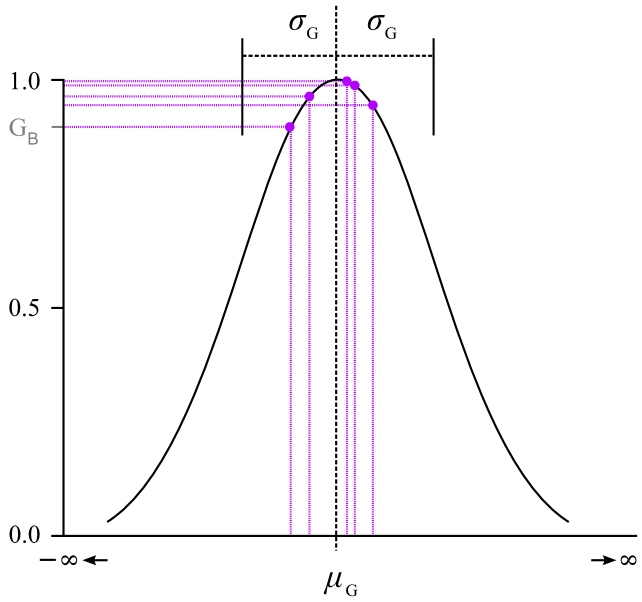

Fig. IV.11. Determining Sigma G for Multiple Reference Values Using a Baseline Similarity. When sufficient reference values are available, σ_G may be determined by choosing a baseline similarity value, G_B (in this example, 0.9), and selecting σ_G such that all reference values project to a similarity value greater than or equal to this baseline value.

$$\sigma_G = \max \sqrt{\frac{(x-\mu_G)^2}{-2ln(G_B)}} \qquad \text{(IV.6)}$$

- If the number of reference values is too small to derive a meaninful value of σ_G, but there are other, similar sets of reference values for which σ_G has already been determined, it may be possible to arrive at an acceptable σ_G by deriving it from the other σ_G's (e.g., by taking their mean).

These last two suggestions are what *Terrainosaurus* employs: when a suitably large set of example height fields (more than one) are available for any given terrain type, σ_G is calculated using Equation IV.6, with a G_B of 0.9; otherwise, σ_G cannot be determined from the terrain type, and instead, the mean σ_G of all the terrain types for which σ_G *could* be determined is used as a fallback.

Comparing Statistical Distributions

With this new tool in hand, we can now move on to comparing entire statistical distributions of things (elevations, slopes, etc.). To do this, we define an adaptive *distribution similarity measure* that compares a test distribution to the reference distributions on the basis of four statistical measures describing the distributions:

- the sample mean μ
- the sample standard deviation σ
- the sample skewness γ_1
- the sample kurtosis excess γ_2

In each of these statistics, the similarity of the test distribution to the reference distribution(s) is calculated using Gaussian curve projection as defined above. The combined distribution similarity is the weighted average of the resulting four individual similarity measures (Equation IV.7).

$$i \in \{\mu, \sigma, \gamma_1, \gamma_2\} \; : \mathrm{S_D} = \sum_{\forall i} w_i \mathrm{S}_i \qquad \text{(IV.7)}$$

The weights assigned to the four statistics (the w_i's) must sum to 1.0, in order to ensure that the combined distribution similarity measure stays within the $[0, 1]$ range, but they are not, in general, equal. As I alluded to earlier, this distribution similarity measure ($\mathrm{S_D}$) is *adaptive*, giving greater weight to individual statistics (the S_i's) in which the reference distributions are more unified and, similarly, giving lesser weight to those in which the reference distributions diverge. To understand the necessity of this adaptability, consider the case where a "hills" terrain type is defined with four example height fields having nearly identical distributions of elevation values, except that each has a significantly different mean elevation from the others (Figure IV.12).

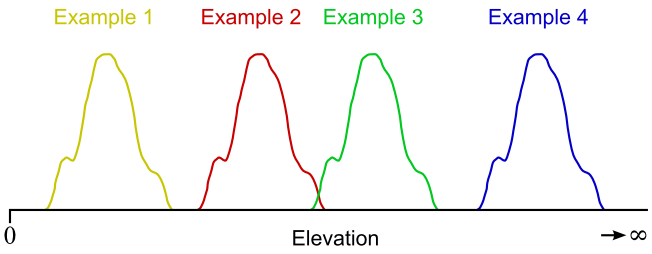

Fig. IV.12. The Need for Adaptability. Taking as an example the case where four distributions are identical, with the exception of their mean values, the need for an adaptive distribution similarity measure becomes apparent.

In this case, the σ_G for the standard deviation, skewness, and kurtosis are all extremely small, while that of the mean elevation is quite large. If, when combining the individual similarity measures to form the distribution similarity measure, we were to assign each measure a uniform weight of 0.25, this would have the undesirable consequence of giving undue merit to height fields whose elevation distributions are completely unlike those of the reference height fields but happen to have a mean elevation within the range spanned by the examples. Because the σ_G for the mean elevation is large, many height fields will have a mean elevation falling in the "good" zone, and would be rewarded for this with a minimum similarity score of approximately 0.25. As the divergence with respect to mean elevation increases, the σ_G grows to embrace more and more of the range of possible values, and the mean becomes less and less of a differentiator (Figure IV.13).

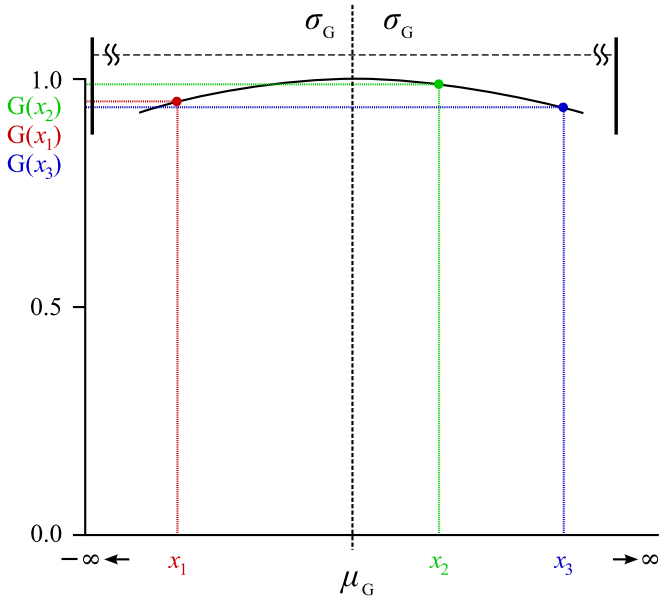

Fig. IV.13. A Useless Statistical Measure. As the reference distributions diverge further and further from one another, with respect to any particular statistic, that statistic becomes progressively more worthless for judging the similarity of a height field to that terrain type, because the Gaussian function for the individual statistics's similarity grows too wide to have any significant ability to differentiate.

Another way of stating this problem is to say that, as the σ_G for a statistic grows in magnitude, that statistic becomes less and less meaningful for answering the question "what gives this terrain type its identity?". In the extreme case of an enormous σ_G, the statistic is totally worthless (Figure IV.13). An obvious solution is to gradually ignore individual statistics as they become less useful: as the reference height fields disagree more strongly on a particular statistic, the σ_G for that statistic increases, and the corresponding w_i for that statistic

should decrease (with the others increasing proportionately to keep the sum of the w_i's at 1.0). Thus, for the example above, the mean would be assigned a weight near 0, and the other three statistics would be assigned weights near 0.33.

In order to determine appropriate values for the weights, we introduce another measure, the *agreement*. The agreement describes how successfully a particular statistic unites the reference distributions. We define the agreement as 1 minus the ratio of the curve variance to the variance of a Gaussian curve that spans the entire terrain library, or zero, if the variance of that statistic exceeds that of the library as a whole (Equation IV.8).

$$i \in \{\mu, \sigma, \gamma_1, \gamma_2\} : \mathrm{A}_i = \max\left(0,\ 1 - \left(\frac{\sigma_{\mathrm{G}}(i,\ tt)}{\sigma_{\mathrm{G}}(i,\ lib)}\right)^2\right) \qquad \text{(IV.8)}$$

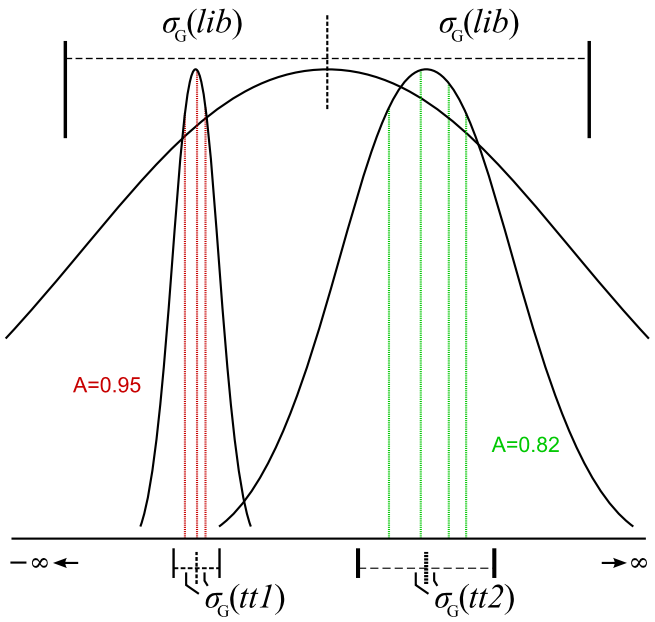

Fig. IV.14. Agreement. The agreement of a terrain type, with respect to an individual statistical measure, is a function of the ratio of that statistic's similarity curve variance to that of a curve spanning the whole library. Statistics that tightly cluster the example height fields (*tt1*) are more useful and exhibit a higher agreement than those in which the example height fields diverge (*tt2*).

This measure (Figure IV.14) gives an idea of how much confidence may be placed in the effectiveness of this statistic, and also gives us a way to set the w_i's in Equation IV.7: weights are chosen in proportion to the agreement values, normalized such that they sum to 1 (Equation IV.9).

$$i, j \in \{ \mu, \sigma, \gamma_1, \gamma_2 \} : w_i = \frac{A_i}{\sum_{\forall j} A_j} \qquad \text{(IV.9)}$$

Finally, we define the *distribution agreement* as the mean of the four individual agreement measures (Equation IV.10). This gives an estimate of the measure of significance that the distribution similarity has: if the example height fields are tightly clustered with respect to each of the four individual statistics, the distribution agreement will be very high. Conversely, if all four statistics are worthless, this value will be near zero.

$$i \in \{ \mu, \sigma, \gamma_1, \gamma_2 \} : A_D = \sum_{\forall i} \frac{A_i}{4} \qquad \text{(IV.10)}$$

Terrain Similarity Evaluation

Having defined a method of comparing the statistical distribution of a quantity between a single test height field and a set of reference height fields, only a small step further is required to be able to compare entire height fields on the basis of multiple such quantities. We define the *terrain type similarity* between a test height field and a set of reference height fields to be the weighted average of the distribution similarities for the following quantities (Equation IV.11):

- elevation
- slope
- edge scale
- edge length
- edge strength (detector response)

$$i \in \{ \forall \text{ distrib. } \} : S_{TT} = \sum_{\forall i} w_i S_i \qquad \text{(IV.11)}$$

As you might expect, the w_i's are defined to be proportional to the respective distribution agreements, as defined in Equation IV.10, normalized such that the w_i's sum to 1 (Equation IV.12).

$$i, j \in \{ \forall \text{ distrib.} \} : w_i = \frac{A_i}{\sum_{\forall j} A_j} \qquad \text{(IV.12)}$$

Just as the individual statistical agreements were used to attenuate the effect of a useless statistic, so also the distribution agreements diminish the effect of whole distributions that do not unify the reference height fields. For example, suppose that a terrain type composed of 5 example height fields displays a high degree of unanimity in the statistical distribution of slopes and elevations, but a lesser degree of correspondence in the distribution of edges. Due to the difference in agreement measures, when evaluating the similarity of a generated height field to the reference terrain type, *Terrainosaurus* will demand a high degree of conformity with respect to elevation and slope in order to give a high score, but will pay less attention to how the height field matches with respect to the other measures. Thus, characteristics in which the examples for a particular terrain type are strongly united will contribute more to the overall terrain similarity than will characteristics in which they diverge.

One advantage to this means of comparing height fields is that it is relatively immune to differences in height field size and shape: because the comparison is made on the basis of statistic characteristics, the height fields may be of differing sizes and shapes...even non-rectangular shapes. For *Terrainosaurus*, this is crucial, as the terrain regions in the user's map are highly unlikely to be rectangular. However, very small or thin regions are likely to perform less well than larger regions, both because of the smaller sample sizes that they represent, and because the smaller areas that they cover will inhibit the formation of longer features.

IV.3.3. Height Field Construction

The final algorithmic aspect that I discuss in depth is the height field construction step. This also employs a genetic algorithm, but to solve a somewhat harder problem than the boundary refinement problem (Section IV.3.1).

IV.3.3.1. Overview

The height field generation algorithm is, at its heart, a multi-scale image synthesis operation: the goal of this process is a height field (greyscale image) of the requested size and LOD, containing plausible imitations of the examples in the terrain library, arranged according to the user's map. At the beginning of this step, *Terrainosaurus* has the following inputs from the user (Figure IV.15):

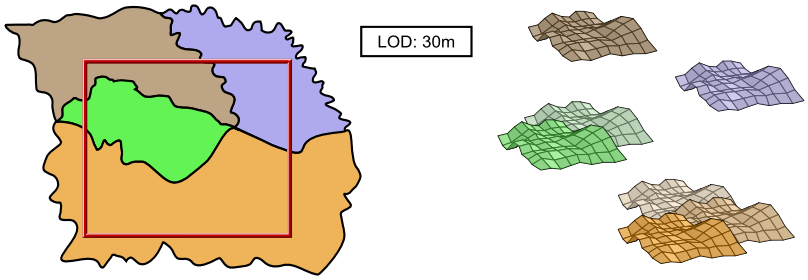

Fig. IV.15. Height Field GA Inputs.

- a 2D vector map, expressing the user's desired layout for the height field
- a world-space, rectangular "active" chunk of this map, indicating the particular area of the map for which a height field should be generated
- the target LOD to generate
- a library of terrain samples, serving as examples of what the terrain types referenced by the user's map should look like

The height field construction process proceeds in an iterative fashion, starting from the coarsest possible LOD and continuing until the target LOD has been reached. The coarsest LOD is constructed in a naïve fashion, by simply pasting together chunks of elevation data from the appropriate terrain types. Each subsequent LOD is constructed by the genetic algorithm, using the previous LOD as a rough "pattern" to follow.

The GA does not attempt to synthesize the height field from scratch; instead, it takes advantage of the fact that we already *have* realistic data for each terrain type...in the terrain library. Conceptually, the genetic algorithm searches for a way to blend together small chunks of terrain from the library, such that the resulting terrain "makes sense":

- each region of the height field resembles its corresponding terrain type
- the transition between regions at each boundary is smooth, without unrealistic discontinuities
- the entire height field has the same macro-scale shape as the previous LOD

The necessity of the first two of these constraints should be obvious: we want the terrain to look real. The reason for the third may require some further explanation. Recall that the LOD strategy used by *Terrainosaurus* is progressive refinement (Section III.1.4.3). During any run of the GA, earlier LODs have already established the macro-scale features of the terrain; the job of the current run of the GA is simply to add new, fine-scale detail, not to re-invent the coarse-scale structure.

IV.3.3.2. Analyzing the Map

The map the user created is a vector drawing. This makes a lot of sense from a user-interface standpoint, since we have no way of knowing *a priori* what LOD the user will want to generate (the user himself may

not know this, and even if he does have a particular LOD in mind, there is no guarantee he won't later decide that he needs a more detailed model). Representing the map with the (virtually) infinite resolution of a vector drawing allows us to defer this decision until the height field is actually generated.

However, since the height field we are going to generate is a raster object, it will be much more convenient to have the map in a similar form (i.e., a raster with the same dimensions as the height field). Furthermore, once the height field generation process begins, we *do* know the resolution of the height field we are to generate, so we can safely convert the map to a raster form, since the additional precision afforded by the vector representation is no longer useful. At the same time, we can do some extra analysis on the rasterized map that will be helpful later on. Specifically, for each grid cell in the rasterized map, the quantities we calculate are:

- the terrain type ID
- the distance to the nearest region boundary
- the unique ID for the enclosing terrain region

The first of these is simply the result of rasterizing the map. The second two are derived from the first, and require segmenting the map back into contiguous regions. It is worth noting that the regions found at this stage may not have a 1-to-1 correspondence with those in the vector-drawn map, for a variety of reasons:

- two adjacent regions with the same terrain type are indistinguishable from one another, and will be merged
- a region that is too tiny to cover any raster cells, or which falls outside of the active area of the map, will not appear in the rasterized map at all, and will be eliminated
- a concave region that only partially intersects the active area of the map may have multiple, distinct fragments that fall within the active area, in which case it will create multiple regions

IV.3.3.3. Creating the Initial LOD

Since the GA requires a "pattern" height field to follow when constructing the next LOD, we cannot begin the generation process without a base LOD with which to prime the GA. Obviously, this base LOD cannot be constructed by the GA, so some other way of creating a height field is needed.

An easy way of creating this base height field is to randomly select appropriately shaped chunks of elevation data from the corresponding LODs of the terrain samples in the library. The discontinuities that would otherwise exist at the edges between terrain types can be avoided by making the selected chunks a few pixels wider around the border and blending between chunks where they overlap. This can be done as an image compositing operation, by constructing an appropriately shaped alpha mask with an alpha value of 1 in the region interior, fading to 0 just outside the region boundaries (Figure IV.16). 5 pixels of overlap seems to work well.

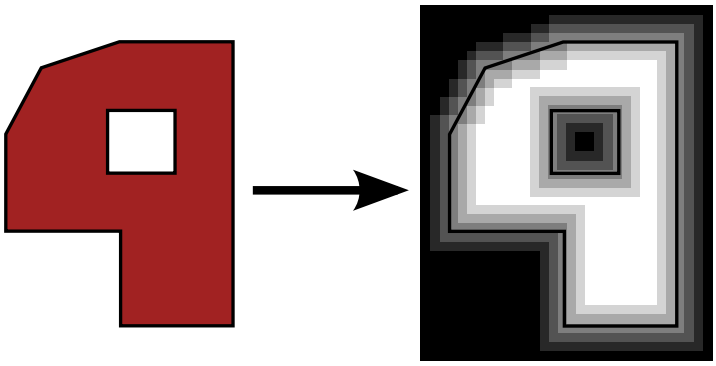

Fig. IV.16. Alpha Mask for Constructing the Base LOD. The base LOD is not constructed by the GA, but is instead created by combining chunks of raw data of the appropriate sizes and shapes, taken from the terrain library. These chunks are blended together, using an alpha mask with a linear falloff across the boundaries.

IV.3.3.4. Encoding & Decoding a Height Field

Once again, the problem must be expressed in a genetic encoding so that the GA can work on it. In this case, the thing that needs to be transformed into a set of genes is a height field.

Perhaps the simplest encoding would be for every pixel in the height field to be its own gene. We rejected this approach as being too fine-grained: a height field of any substantial size would contain thousands or millions of genes, and it is difficult to envision meaningful mutation operators to work on such an encoding.

In the encoding we selected, each gene represents a small, N×N-pixel chunk of the height field, and has a terrain type and a transformation associated with it. A gene does not directly contain the elevation data for its chunk of the height field; instead, it holds a pointer to one of the example terrains for its terrain type, and the (x, y) coordinates within that terrain from which to copy its source data. The transformation allows the source data to be rotated, translated and scaled before being blended into their target location (Figure IV.17).

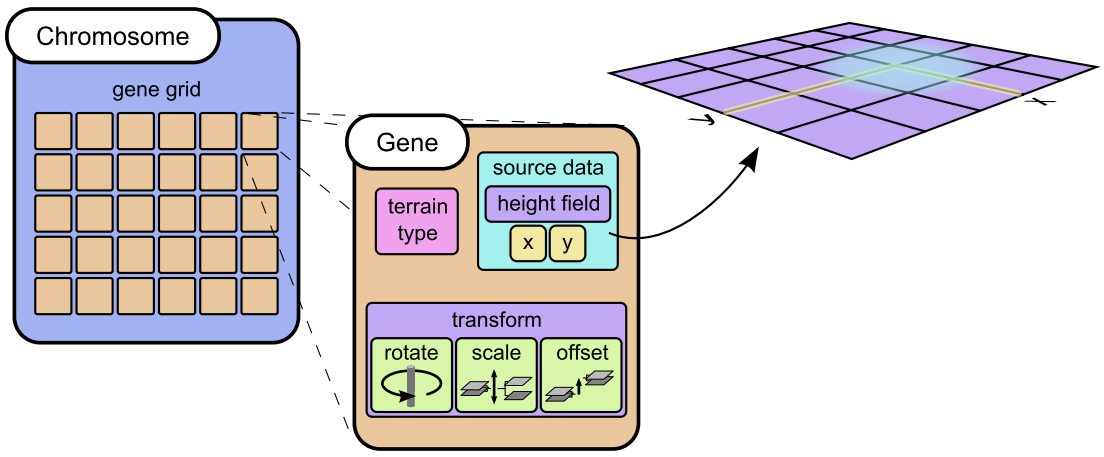

Fig. IV.17. The Encoding of a Gene in the Height Field GA.

A chromosome is a 2D grid of such genes, arranged such that each gene overlaps slightly with those on each side of it (Figure IV.18). To prevent unseemly seams from appearing between adjacent genes, we blend between adjacent genes using a 2D Gaussian blending function.

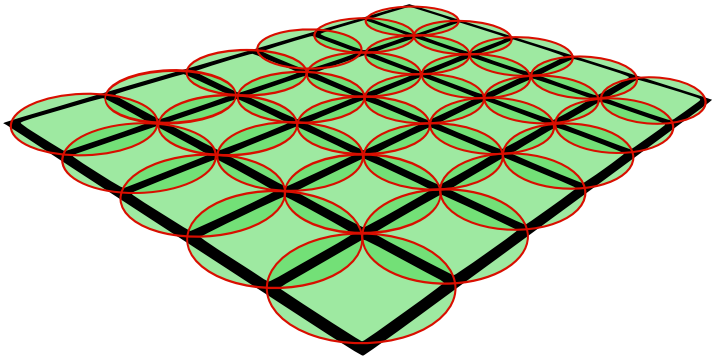

Fig. IV.18. The Gene Grid. A chromosome in the height field GA is a 2D grid of overlapping genes. Each gene has a local area of influence, within which it is responsible for determining elevation values of the height field. In the overlapping areas, two or more genes contribute to the height field elevations, with the areas of influence smoothly blended between them.

One benefit of this encoding is that it is relatively compact. By keeping only a *reference* to the source height field data within the gene, the chromosomes are able to be substantially smaller than they would be if they carried the pixel data internally (a single 1024×1024 height field stored in IEEE single-precision

floating-point format is four megabytes in size!). As a result, memory is not a significant factor in determining the GA population size (instead, processing time is the limiting factor).

A second, less obvious benefit is that it is computationally inexpensive. Since the gene transformation is only applied to the pixel data when the chromosome is decoded, mutation operations that only modify the transformation parameters can be very fast. Also, because no pixel data is stored in the gene, crossover operations are faster as well. Furthermore, many queries about the geometric characteristics of a gene (mean elevation, gradient, etc.) can be made quite inexpensive by precomputing these quantities for each of the reference terrain samples. Then, to query these properties for a gene, all that needs to be done is to look up the precomputed property for the gene's source data location, accounting for any transformation applied to the data.

Another not-so-obvious trait of this encoding is that it is a *lossless* encoding: it is possible to encode and decode a height field using this scheme, recovering the original height field exactly. It also is relatively robust against numerical drift: the transformation parameters for each gene can be tweaked indefinitely without corrupting the elevation data, since the transformation is only applied when the chromosome is decoded back into a height field, and it is always applied to the original data.

IV.3.3.5. Genetic Operators

With this encoding, a wide variety of genetic operators is possible; virtually any image processing operation is a candidate, though not all operations are equally reasonable. For example, a "vortex" transformation on a gene is not likely to improve the terrain configuration substantially, and in many cases would introduce unrealistic shapes to the generated height field. On the other hand, an image "rotation" transformation corresponds directly to a geometric rotation of the height field surface, and could be quite useful for rearranging the genes.

We use a varied set of genetic operators to operate on the chromosomes for the height field GA.

Rectangular Region Copy (crossover)	This crossover operator exchanges rectangular clusters of corresponding genes (i.e., genes with the same grid indices) between two chromosomes. The height and width of the copy rectangle are both selected randomly from the range $[\,1,\,N\,]$, thus, the number of copied genes falls into the range $\left[\,1,\,N^2\,\right]$. The location of the copy rectangle within the gene grid is randomly selected such that the entire rectangle is within the grid. This operator corresponds to an image copy operation.
Vertical Offset (mutation)	This mutation operator modifies the vertical offset component of a gene's transformation, effectively altering the mean elevation of the gene. It tends to transform the gene in the direction of the pattern height field's mean elevation in the vicinity of that gene. This operator is similar to an image brightness adjustment operation.

Vertical Scale (mutation)	This mutation operator modifies the vertical scale component of a gene's transformation, changing the elevation range of the gene without altering its mean elevation. It tends to transform the gene towards having the same elevation range as the pattern height field has in the vicinity of the gene. This operator corresponds to linearly stretching or compressing the contrast of an image.
Rotation (mutation)	This mutation operator modifies the rotation component of a gene's transformation, effectively rotating the contents of the gene about the horizontal center point of the gene. It tends to transform the gene toward having the same gradient direction as the pattern height field has in the vicinity of the gene. This operator corresponds directly to an image rotation operation.
Random Source Data Selection (mutation)	This mutation operator completely replaces the source data (i.e., the pointer to the source height field, and the coordinates within that height field) in a gene. The new source height field is randomly chosen from among the examples for the gene's terrain type, and the new source coordinates are randomly chosen from within that example.
Horizontal Offset (mutation)	This mutation operator modifies the coordinates within the source terrain sample from which the gene takes its data. Like the previous operator, this one has the effect of replacing the gene's contents with new data, but the effect is likely to be less drastic, as it keeps the same source height field and only picks new source coordinates within that height field.

I said of several of the mutation operators (those that modify the gene's transformation parameters) that they "tend to" adjust the transformation towards conformity with the pattern height field. This works by introducing a random change to the transformation parameter, drawn from a Gaussian distribution centered over the value that would conform the gene to the pattern (thus, this value has the highest probability of being chosen, but a nearby value may be chosen instead).

IV.3.3.6. Fitness Evaluation

Finally, we come to the crux of the matter: how does the GA discern between good height fields and bad ones? The fitness evaluation is separated into two aspects, and we calculate the overall fitness as a weighted combination of the two (Equation IV.13).

$$F = (1-\alpha)C_g + \alpha S_r \qquad \text{(IV.13)}$$

C_g geometric compatibility

S_r	regional terrain type similarity
α	a weighting coefficient, controlling how strongly each component affects the overall fitness; 0.5 was found experimentally to be a reasonable value

Geometric Compatibility

Geometric compatibility describes how well the height field encoded in the chromosome matches the "pattern" provided by the previous LOD. This is important for ensuring that the generated LOD is conforming to the macro-scale features constructed by the previous LODs. The compatibility of a chromosome can be estimated directly from the genetic representation, by comparing the mean elevation and the mean gradient over each gene's area of influence with those values for the corresponding areas of the pattern height field.

In comparing these geometric properties, we encounter the same difficulty that we did earlier, in our discussion of comparing statistical distributions: we are trying to compare two unbounded quantities, to get a compatibility measure in the bounded range $[\,0,\,1\,]$. We employ the same solution to this problem here as we did to the other instance: a Gaussian curve projection. In this case, the curve mean is the value of the pattern height field that we're trying to match, and the curve standard deviation is chosen to be one fourth the range for the gene's terrain type (e.g., for a "mountain" gene, the curve standard deviation would be one fourth of the elevation range for the "mountain" terrain type). Again, the choice of standard deviation is somewhat arbitrary, but this value seems to work well, allowing a moderate amount of disagreement between a gene and the pattern before the gene starts to be heavily penalized. The total compatibility of a gene is then calculated from Equation IV.14.

$$C_{g,i} = \frac{G(E_i) + G(M_i) + G(A_i)}{3} \qquad \text{(IV.14)}$$

$C_{g,i}$	overall compatibility for gene *i*
E_i	mean elevation over gene *i*
M_i	mean gradient magnitude (slope) over gene *i*
A_i	mean gradient angle over gene *i*

The aggregate compatibility of the entire chromosome (C_g) is then simply the mean of all of the $C_{g,i}$'s.

Regional Terrain Type Similarity

The second aspect of height field fitness is the regional terrain type similarity: how similar each region is to the examples that make up its corresponding terrain type, in terms of measurable characteristics.

The similarity of each individual region is calculated just as described in Section IV.3.2. For the aggregate similarity of the whole height field, however, rather than simply using the arithmetic mean of the individual regional similarities, we instead calculate the area-weighted mean, using the proportion of pixels within each region as the weight for that region's similarity. This reduces the impact that small regions have on the overall fitness of the terrain.

Localized Guidance of the Genetic Algorithm

As is the case for many things, the GA's great strength can also be a weakness. GAs can solve optimization problems in which the effects of changes are not well understood, precisely because the GA is agnostic about the internal interactions, only evaluating the outcomes. Unfortunately, this also means that a standard GA is rather "dumb": it does not take advantage of domain-specific knowledge that might help guide the GA more quickly in the direction of an optimal solution (or, at least, *away* from truly horrible solutions). When the chromosome size is large, the contribution of any individual gene to the overall fitness is highly diluted, thus an "error" in a gene may take a long time to be fixed, and convergence will be slow.

To address this, we modify the GA to retain additional information from the fitness evaluation, and we use this information to adjust the probability of a mutation occurring in a gene and also to influence the probability distribution function for choosing which mutation operator is invoked.

We do this in several places in the GA. At the region level, we retain the region similarity measure, using it to increase or decrease the mutation probabilities of the genes within that region. Similarly, at the gene level, we retain the individual compatibility components, as well as the overall compatibility measure. The probability of a gene being mutated is calculated using Equation IV.15:

$$P_i = P_m\left(\frac{1}{2} + \frac{1-S_r}{4} + \frac{1-C_{g,i}}{4}\right) \qquad \text{(IV.15)}$$

where P_m is the baseline probability of mutation. With this formulation, the baseline mutation probability still has a strong effect, but genes in highly dissimilar regions and genes that are highly incompatible with the underlying pattern are substantially more likely to be mutated.

Once the GA decides to mutate a particular gene, it still has to choose which mutation operator it will apply. An attractive feature of this enhancement to the GA is that smart and dumb mutation operators may be freely intermixed. A *smart* mutation operator is one that has a predictable relationship to some component of the fitness evaluation (e.g., E_i, the mean elevation of a gene, is directly affected by the "vertical offset" operator). In contrast, a *dumb* genetic operator produces unpredictable effects, or else cannot easily be related to any part of the fitness evaluation (e.g., the "pick new source location" operator). When deciding which operator to use, the GA modifies the mutation operator probability distribution function (PDF), giving higher probabilities to smart operators that are needed by the current gene and lower probabilities to smart operators that are not. Probabilities for dumb operators remain fixed.

One way to understand these changes to the GA is as additional, tighter feedback loops within the GA, essentially creating several child GAs within the height field generation GA, who are better equipped to handle particular aspects of the problem. One point that deserves to be mentioned is that, despite incorporating this additional guidance, the algorithm is still a *probabilistic* algorithm, not a deterministic one. While a speedier convergence rate is generally a good thing, we want to retain the randomness and diversity of the GA, so as not to lose the ability to escape local maxima in the solution space (i.e., terrains that are "OK" but not "great").

Cleaning Up the Final Result

A final improvement we can make concerns those few genes that are out-of-place at the end of the GA. Given enough evolution cycles, the GA is generally successful at bringing *most* of the genes into alignment, however, since it is a probabilistic algorithm, it is not unlikely that a handful of genes out of the dozens (or hundreds) in the chromosome might escape being brought into conformity with the rest. Such genes are especially noticeable if they have a substantially different elevation from the surrounding terrain; they look sort of like squarish "fingers" poking up from the ground (Figure IV.19).

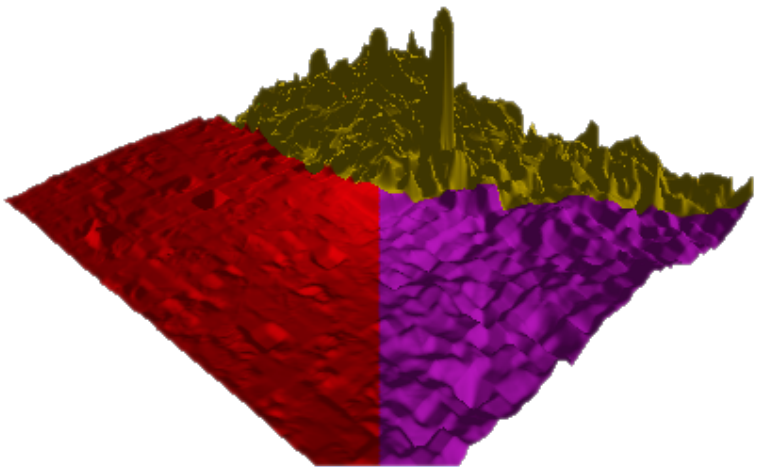

Fig. IV.19. Unaligned Genes Look Like "Fingers".

These remaining artifacts can be removed by applying a *conform* operator to these aberrant genes. The conform operator simply sets the gene's transformation parameters so as to give the gene a compatibility value $C_{g,i}$ of 1; in effect, it forces the gene to fit the pattern height field as closely as possible.

CHAPTER V

IMPLEMENTATION

In this chapter, I discuss some of the architectural features of the prototype implementation of *Terrainosaurus*, covering some of the more significant design decisions that were made and how some of the technical challenges were dealt with. In so doing, I hope to save other developers some of the difficulties I encountered during this research.

In this chapter, I present the following topics:

- the choice of development platform and technologies used
- a discussion of the application architecture
- suggested optimizations and simplifications

V.1. Technologies

The *Terrainosaurus* algorithm could be implemented using any number of programming languages and libraries, both commercial and free, proprietary and open-source. While these decisions are ultimately up to the programmer(s) implementing the algorithm, it may be productive to consider the decisions made in the design of the current implementation, and the reasons for them.

V.1.1. Development Platform

The current implementation of *Terrainosaurus* was developed in C++, with heavy reliance on the Standard Template Library (STL) and a number of the Boost libraries. C++ was selected for a number of reasons, including the following:

- C++ is a multi-paradigm programming language, allowing the developer a great deal of freedom in selecting the best approach to a particular problem. Object-oriented techniques, for example, are well-suited to implementing user interfaces, whereas generic programming techniques are appropriate for low-level utilities and complex algorithms.
- C++ gives the programmer a great deal of freedom in managing resources (such as memory), and typically does not incur the cost of compiler-generated run-time checks, meaning that a carefully written program can be very fast. Languages such as C# and Java provide nice additional features (run-time array bounds checking, garbage collection, etc.), but these come at the cost of run-time performance; thus, a well-written C++ program will always be faster than an equivalent program in C# or Java. Of course, the risk of foregoing these features is that bugs may be harder to track down, and as a result, it may take longer to develop the application.
- Good compilers for C++ exist for all major computing platforms, such as Microsoft's compiler for the Win32 platform and GCC for the many UNIX variant platforms.
- Third-party libraries, of both the commercial and free varieties, are widely available for C++, solving a diverse spectrum of problems.

Another means of implementing *Terrainosaurus* that was considered is as an extension to a general-purpose numeric processing application, such as *Matlab* or its free cousin, *Octave*. These programs provide native implementations of mathematical constructs such as vectors and matrices, have a (limited) graphical user interface, and have an impressive array of add-on "toolboxes" providing additional functionality, including statistical analysis, image processing, and pattern analysis. Because *Matlab* is an interpreted language it is especially useful for rapid prototyping of algorithms.

Although it was considered, *Matlab* was ultimately rejected as a development platform, for two main reasons.

1. *Matlab* is a commercial product; if implemented as a *Matlab* toolbox, *Terrainosaurus* would only be usable by persons having access to a copy of *Matlab*. A person interested in modeling terrain is not especially likely to be a *Matlab* user.
2. *Matlab*'s programming language does not provide sufficient facilities for modularization, type safety, and code reuse to make development of a medium- to large-scale application feasible.

V.1.2. Graphics API

OpenGL was chosen as the 3D rendering API, because of its cross-platform availability and familiarity. Another immediate-mode rendering API (e.g., *Direct3D*) could have been used equivalently.

Another possibility for displaying the results is to implement an interface to one of the commercially available modeling and rendering systems (e.g., *Maya*, *3D Studio*). Then, rather than being rendered directly, the results would be used to instantiate objects in the modeling system's scene graph. In this way, one could get high-quality rendering support "for free".

V.2. Supporting Libraries

While almost any needed functionality could, in principle, be implemented directly in C++, this is wasted effort when there already exist mature, freely available C/C++ libraries and tools providing good solutions for these problems. In order to simplify the development process, a third-party library was used wherever possible, as long as the following things were true of it:

1. The library is fairly mature, providing a robust, full-featured solution to the problem domain it addresses. Incomplete and alpha-quality libraries are not desirable.
2. The library has adequate documentation, and is under active development/maintenance. This gives some degree of confidence that the developers of the library are committed to its continued existence and improvement.
3. The library is cross-platform, and does not have dependencies on proprietary libraries. This keeps *Terrainosaurus* from being bound to a particular operating system variant.
4. The library is distributed under fairly liberal licensing terms. A "do (mostly) whatever you want" style license, such as the Apache License or the LGPL is preferable, but a GPL'd library was considered acceptable if it is also available under a commercial license. This leaves open the possibility of future commercial development.

V.2.1. File Parsing

For small-scale projects, it may be sufficient to hard-code configuration constants and algorithm parameters directly into the source code. An application of any significant size and complexity, though, generally needs to be able to accept configurable parameters from data files (and ideally, in a robust way, so that a malformed data file does not cause the application to crash). Besides adding to the overall quality of the application, having the ability to read configurable parameters from a file can also drastically cut down on the amount of time needed to tune a complex algorithm, by eliminating the edit/compile/run cycle.

Writing robust file parsing code can be both difficult and tedious, so some sort of higher-level solution is desirable. *ANTLR* [Parr 2006] is one such tool, allowing a developer to write a description of the grammar for his file format and generating Java, C#, C++, or Python code for parsing files in that format. The generated code is human readable (unlike that produced by other tools like yacc), and generates fairly good error messages for malformed files. Once familiar with *ANTLR*, a developer can modify or extend the format of the file with very little effort.

V.2.2. Fourier Transform

Frequency-domain signal analysis generally implies calculating the discrete Fourier transform (DFT) of a spatial-domain or time-domain signal (this is needed in *Terrainosaurus* for speeding up the feature detection step, as described in Section V.4.2.1). However, writing an efficient discrete Fourier transform (DFT) implementation is tricky indeed—a research area in its own right. Fortunately, a very complete and highly optimized C library exists for performing many variations on the DFT, called *FFTW* (the "Fastest Fourier Transform in the West") [Friggo & Johnson 2003]. *FFTW* is available from the Massachusetts Institute of Technology under both the GPL and a commercial license.

V.3. Application Architecture

A suggested general architecture for an implementation of *Terrainosaurus* is pictured in Figure V.1. To a large degree, this is a straightforward reflection of the concepts described in Chapter IV.

In this section, I discuss the following aspects of this suggested architecture in greater depth:

- how LOD is expressed in the architecture
- inputs and outputs of the algorithm
- the data structures
- the user interface
- implementing the genetic algorithms

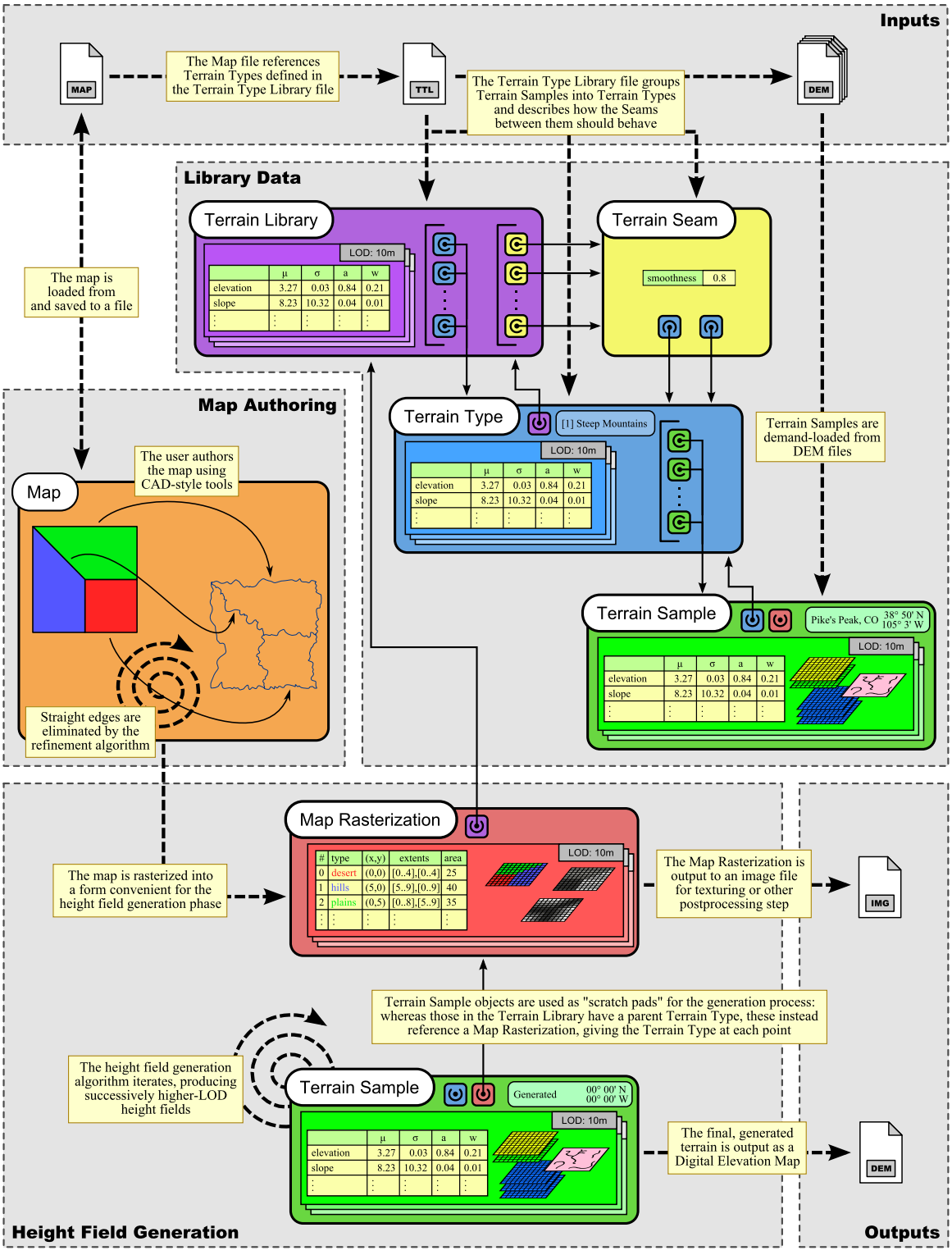

Fig. V.1. A Suggested Application Architecture.

V.3.1. LOD Handling

Level of detail is a concept fundamental to *Terrainosaurus*, and is important for a number of things. As discussed in Chapter IV, LOD is central to the height field generation algorithm: starting from a coarse LOD, the algorithm builds progressively finer-scale height fields, until reaching the user's desired LOD. In each iteration, the algorithm uses real-world elevation data of that LOD, either taken directly from GIS data available at that LOD, or else resampled from GIS data at a different LOD. LOD is also important in other, less obvious ways. For example, the desired LOD tells us how finely the map boundaries must be subdivided in order for there not to be any straight boundaries in the resulting height field. Also, when rendering a height field, it is necessary to know the LOD: this specifies how to scale the x and y dimensions of the height field in order to be in correct proportion to the vertical axis.

Because we are working with USGS elevation data, the choice of LODs has already been made for us, to a large degree. USGS DEMs are commonly available in resolutions of 1/9 arc-second (3 1/3 meters per sample), 1/3 arc-second (10 meters per sample) and 1 arc-second (30 meters per sample). Because of this, a power-of-three relationship between LODs is most convenient, as it requires the least amount of resampling, because the standard-resolution DEMs can be used directly. Following this power-of-three relationship, coarser LODs can be derived with resolutions of 90m, 270m, 810m (Figure V.2). Further resolutions, such as 2.4km and coarser, are less useful because standard USGS 10m and 30m DEMs become unusably small when resampled to such coarse resolutions (around 4x4 samples).

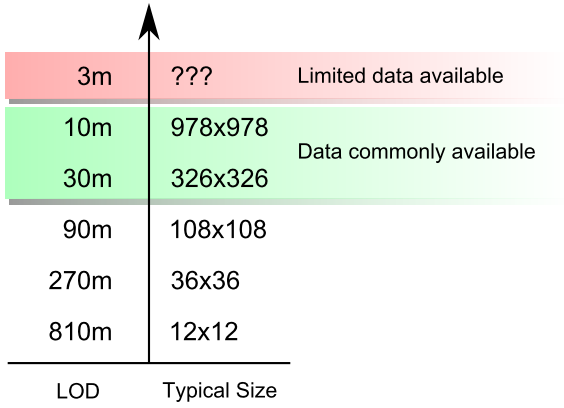

Fig. V.2. Choice of Levels of Detail. USGS digital elevation maps come in a range of LODs, with a power-of-three relationship between successive resolutions. This scheme can be extended to include additional resolutions for which data is not typically available (in these cases, the standard-resolution maps must be resampled).

Because LOD is so ubiquitous throughout *Terrainosaurus*, it is natural that its data structures would directly support multiple levels of detail. The diagrams in this chapter depict the multi-LOD components of the data structures as in Figure V.3.

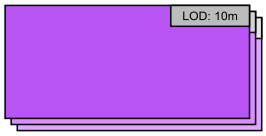

Fig. V.3. Multi-LOD Objects. Most of the data structures suggested for *Terrainosaurus* have at least some attributes that are LOD-specific.

V.3.2. Inputs & Outputs

Figure V.1 depicts the inputs and outputs of an implementation of *Terrainosaurus*. Assuming that *Terrainosaurus* is not embedded in the context of a larger application, this generally implies that *Terrainosaurus* is reading/writing files from/to machine storage (e.g., the computer's hard drive).

The input and output files include:

- the terrain type library (TTL) file—this describes the user's classification of example height fields into a taxonomy of terrain types and is read to determine what elevation map files should be loaded (see Figure V.5 in Section V.3.3.1 for a suggested format for this file)
- terrain type map (MAP) files—these contain the map designs authored by the user, and are both read and written by *Terrainosaurus* (see Figure V.10 in Section V.3.3.5 for a suggested format for these files)
- digital elevation map (DEM) files—these contain height field elevation data, and are read to load the example height fields, and written to save the height fields generated by *Terrainosaurus* (see Section V.3.3.3 for a discussion of file formats for height fields)
- image (IMG) files—these encode the terrain type of each point in a height field as a pixel color and are generated as a by-product of the map-rasterization process; they may be used for terrain-type-based postprocessing of the generated height fields

V.3.3. Suggested Data Structures

The way in which data is organized is of great importance in the design of most kinds of software. A poorly structured data model will adversely impact the design of the rest of the system and may be difficult to change once the rest of the system has been built. As an aid to future implementors, I offer the following suggested organization of data structures.

V.3.3.1. Terrain Library

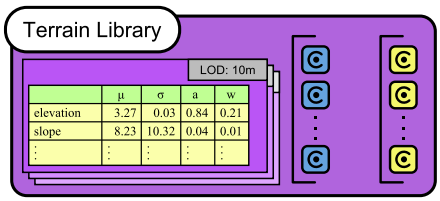

Fig. V.4. The Terrain Library Data Structure.

The Terrain Library structure (Figure V.4) is primarily a container for the other data structures, but it also holds aggregate statistics for the entire library of height fields. The important components of the Terrain Library are:

- a set of Terrain Type objects representing the various types of terrain defined by the user
- a set of Terrain Seam objects representing the properties of the seams between each possible pair of Terrain Types
- terrain statistics aggregated from all of the Terrain Samples in the library; these are used to establish the significance of the agreement between the Terrain Samples of a Terrain Type
- similarity parameters aggregated from all of the Terrain Types in the library; these are used as a fallback when a Terrain Type does not have enough Terrain Samples to calculate its own similarity parameters

Because the Terrain Library, once constructed, is essentially static, it makes sense to store this information in a file, and to load it at application startup. A simple way to accomplish this is with a file format resembling the Windows .ini format, essentially a human-readable list of key/value pairs grouped into sections (Figure V.5).

```
# A Terrain Type entry
[Terrain Type: California_Coast_Hills]
  color = <0.6, 0.6, 0.3, 1.0>
  sample = "35120e8 - Cypress Mountain, CA"
  sample = "35120f8 - Lime Mountain, CA"
  sample = "35121f1 - Pebblestone Shut-in, CA"
  sample = "33116b7 - Mesa Grande, CA"
.
.
# A Terrain Seam entry
[Terrain Seam: Colorado_Small_Mountains & Colorado_Large_Mountains]
  smoothness = 0.3
.
.
# Terrain statistics aggregated across the whole library for an LOD
[Aggregate: LOD_30m]

  # Whole-library variances, used to calculate agreement
  elevation_mean_variance     = 8.18938e+008f
  elevation_stddev_variance   = 2.14651e+006f
  elevation_skewness_variance = 11.5058f
  elevation_kurtosis_variance = 85.087f
.
.
  # Whole-library averages, used when a terrain type has
  # insufficient samples to calculate its own values
  default_elevation_mean_variance     = 2.48967e+006f
  default_elevation_stddev_variance   = 68969.8f
  default_elevation_skewness_variance = 1.74482f
  default_elevation_kurtosis_variance = 4.5054f
.
.
```

Fig. V.5. An Example Terrain Type Library (.ttl) File.

V.3.3.2. Terrain Type

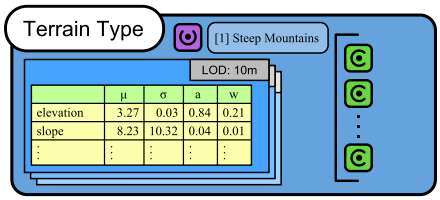

Fig. V.6. The Terrain Type Data Structure.

A Terrain Type (Figure V.6) represents a single, conceptual type of terrain, having one or more concrete examples. Its important components are:

- a name, describing the terrain type in a way that is meaningful to the user (e.g., "Mountains")
- an integer ID, uniquely identifying this Terrain Type within the parent Terrain Library; this ID is used by other data structures to reference this Terrain Type
- a set of Terrain Sample objects, the example terrains that make up this Terrain Type
- terrain statistics aggregated from all the Terrain Samples belonging to this Terrain Type; these are used to calculate the similarity parameters used when measuring how "like" this Terrain Type a generated height field is

V.3.3.3. Terrain Sample

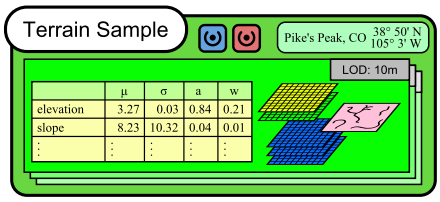

Fig. V.7. The Terrain Sample Data Structure.

The Terrain Sample class (Figure V.7) represents a single, rectangular chunk of terrain, and can be thought of as an enhanced height field. It serves two similar, but distinct functions:

1. it represents the example GIS terrains within the Terrain Library, in which case it has a parent Terrain Type
2. it represents the under-construction terrains, in which case it has a Map Rasterization and does *not* have a parent Terrain Type

The important components of a Terrain Sample are:

- if this is an example terrain in the terrain library, a reference to the parent Terrain Type
- if this is a height field being generated by *Terrainosaurus*, a reference to a Map Rasterization describing where the terrain regions are located and what the terrain type is at each point in the height field
- a rectangular grid of height field elevations (pictured above in yellow)
- a rectangular grid of 2D vectors representing the gradient at each point in the height field (pictured above in green)
- lists of features (peaks, edges, ridges, etc.) located within the height field; each feature contains one or more 4-dimensional points, giving the (x, y) location within the height field, the scale at which the detector gave the strongest response, and the value of that detector response
- calculated statistics for each region in the height field (encoded in the associated Map Rasterization); these statistics need to be evaluated separately for each region because each region must be evaluated against its corresponding Terrain Type
- raster objects containing windowed statistical measurements of the Terrain Sample, such as the mean elevation, mean gradient and minimum/maximum elevation, calculated over the N×N neighborhood surrounding each cell of the raster; these are precalculated when the Terrain Sample is studied, and are during the height field construction GA to perform efficient queries of the geometry of individual genes (pictured above in blue)

The vast majority of the data processed by *Terrainosaurus* comes from terrain elevation maps. Height field data are commonly found in either the DEM (Digital Elevation Map) format [USGS 2003] or the SDTS (Spatial Data Transfer Standard) format [USGS 2003], with newer data available only in the SDTS format. The DEM format is an ASCII text format with fixed-length records, a relatively simple format, but also rather bulky—a typical 30m DEM is larger than a megabyte. SDTS, in contrast, is a binary file format, and is much more compact, but also much more complicated, as SDTS is the USGS's "Swiss Army Knife" format, capable of storing a wide range of raster and vector map data.

While the preceding discussion may sound disheartening, the developer of terrain processing software actually has quite a bit of latitude in selecting a terrain file format. This is because, while GIS data sources typically use only the aforementioned formats, there exist public-domain utilities for converting between these formats and a wide array of raster formats, including TIFF, Targa, raw XYZ coordinates, POV (the *POV-Ray* ray tracer file format), and *AutoCAD* DXF. The *Virtual Terrain Project* [VTP 2006] also describes an additional, terrain-specific format, the Binary Terrain (BT) format, which also offers better compression than ASCII formats.

In the current implementation of *Terrainosaurus*, we chose to implement a parser for the DEM format, because DEM is easy to read and because PC environments typically have plenty of hard drive space, so the data expansion is not a huge problem. Image file formats are not ideal for storing terrain data because, in order to encode a height field into such a format, the elevations must be scaled to fit within the range of legal pixel values (often $[0, 1]$ or $[0, 255]$; doing so loses the absolute scale of the data, and makes it impossible to compare elevations between different height fields.

One important "gotcha" to be wary of, when using DEM or SDTS data, is that the real-world tiles covered by the data in an elevation map are often non-rectangular, due to the curvature of the Earth (Section III.1.1). As a result, the actual, valid region of a DEM or SDTS height field may be a trapezoid (or even an arbitrary quadrilateral, depending on the mapping coordinate system used); grid cells outside of this valid region will be marked "void". A simple solution for dealing with this is to trim some number of pixels off of each edge (30 pixels seems to be sufficient), thereby ensuring a rectangular valid region.

V.3.3.4. Terrain Seam

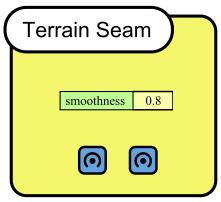

Fig. V.8. The Terrain Seam Data Structure.

The Terrain Seam class (Figure V.8) represents characteristics of the boundary between two different Terrain Types (two adjacent regions of the same Terrain Type are considered to be a single, contiguous region; thus, a Terrain Type cannot have a boundary with itself). The components of a Terrain Seam are:

- a scalar in the range $[\,0,\,1\,]$ indicating the target smoothness for that boundary type
- references to the two Terrain Types this Terrain Seam separates

V.3.3.5. Map

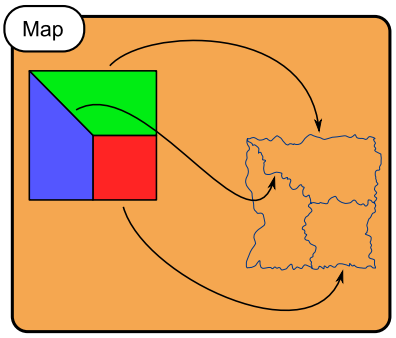

Fig. V.9. The Map Data Structure.

The Map class (Figure V.9) represents a user-authored, vector-drawn Terrain Type map.

The Map can be implemented as a 2-dimensional connected polygonal mesh (using, for example, a variation of the winged-edge [Baumgart 1975] data structure): polygonal regions in the map correspond to mesh faces, and boundaries between regions correspond to mesh edges. The usefulness of this approach is that conventional polygon-modeling tools and techniques may be used to author the map. Furthermore, most 3D modeling packages already contain robust tools for editing polygon meshes, possibly simplifying implementation of *Terrainosaurus* as part of such a package.

In order to use a polygon mesh structure to represent the map, it must be augmented with some additional data fields. Instead of conventional polygon mesh attributes like 3D positional coordinates, texture coordinates, vertex colors and normals, the map mesh needs the following attributes:

- 2D positional coordinates for each vertex in the map
- the Terrain Type for each region (face) in the map
- the sequence of 2D points representing the boundary refinement for each edge in the map

Because the Map is implemented as a topologically connected polygon mesh, it must also be stored this map in a format that preserves this information. A simple way to do this is with a trimmed-down version of the Wavefront OBJ file format [Wavefront 1995]. The only record types of the OBJ format that are needed are a 2D version of the vertex record ('v'), a terrain type declaration ('tt', analogous to the material declaration), and a face record ('f') (see Figure V.10).

```
# Map vertices
v -700.0 -700.0     # 1
v -400.0 -400.0     # 2
.
.
# Map regions (faces)
tt California_Coast_Hills
f 1 2 3 4 5 6
.
.
```

Fig. V.10. An Example Terrain Type Map File. One way of persisting the user's terrain type map is with a variation on the ubiquitous Wavefront OBJ file format.

V.3.3.6. Map Rasterization

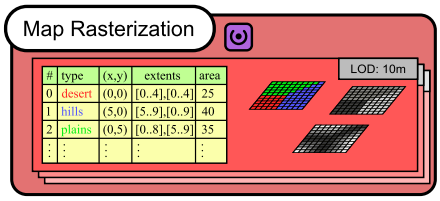

Fig. V.11. The Map Rasterization Data Structure.

The Map Rasterization class (Figure V.11) is a raster version of a Map, and is usually associated with one or more Terrain Samples of the same dimensions. Besides being useful during the height field generation process, as described in Section IV.3.3.2, the Map Rasterization is also useful for exporting a representation of the map in a form useful for downstream processing of the height field: the raster of terrain type IDs can be saved in a conventional image format (PNG or Targa, for example), and used to do further computation, such as assigning texture coordinates to the terrain, or populating the terrain with trees, rocks, or other objects. The components of a Map Rasterization are:

- a raster object containing the integer terrain type ID for each grid cell
- a raster object containing the integer region ID for the region enclosing each grid cell
- a raster object containing the scalar distance from the center of the grid cell to the nearest region boundary; this is used to generate the alpha masks for creating the coarsest LOD to prime the generation process (Section IV.3.3.3)
- the number of distinct regions present in the Map Rasterization
- the Terrain Type, pixel-area, axis-aligned bounding box of each region, and a pixel located inside of that region (needed to flood-fill the region)

V.3.4. Suggested User Interface

Ideally, a graphical user interface for *Terrainosaurus* should allow the user to view, navigate and edit the terrain type map in an intuitive fashion, and to invoke the height field generation GA, and to visualize the results. At a minimum, the interface should support the following operations:

- load map
- save map
- create region
- delete region
- move vertex
- set terrain type
- refine boundary

- select region to generate
- save terrain

The prototype implementation of *Terrainosaurus* accomplishes these operations with two user interface windows: a map editor window and a terrain viewer window.

The map editor window (Figure V.12) allows the user to select regions, boundaries, and vertices by clicking on them or "lasso selecting" them with the mouse. When selecting individual objects or adding regions to the map, the mouse pointer will snap to nearby vertices and edges, making it possible to create connected polygons. Loading, saving and editing operations are triggered by keyboard commands.

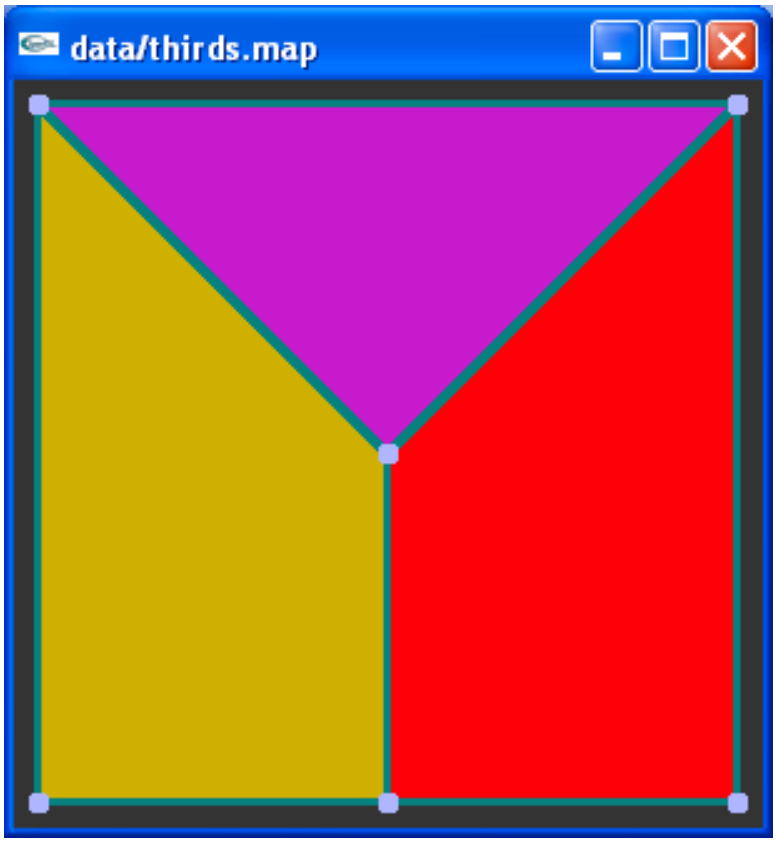

Fig. V.12. The Map Editor Window. The map editor window allows the user to view, navigate and edit a terrain type map using polygon modeling operations.

The terrain viewer window (Figure V.13) is a 3D height field viewer, allowing the user to explore the generated height field (or one of the example height fields) in 3D. One minor "trick" that deserves mentioning is that, when displaying a height field, the height field geometry (or the camera) should be offset in the vertical direction by the mean elevation of the height field. This is necessary in order to have the height

field in the viewport; if this transformation is not done, the height field surface may be located far above or below the camera in the 3D space, making it difficult to find.

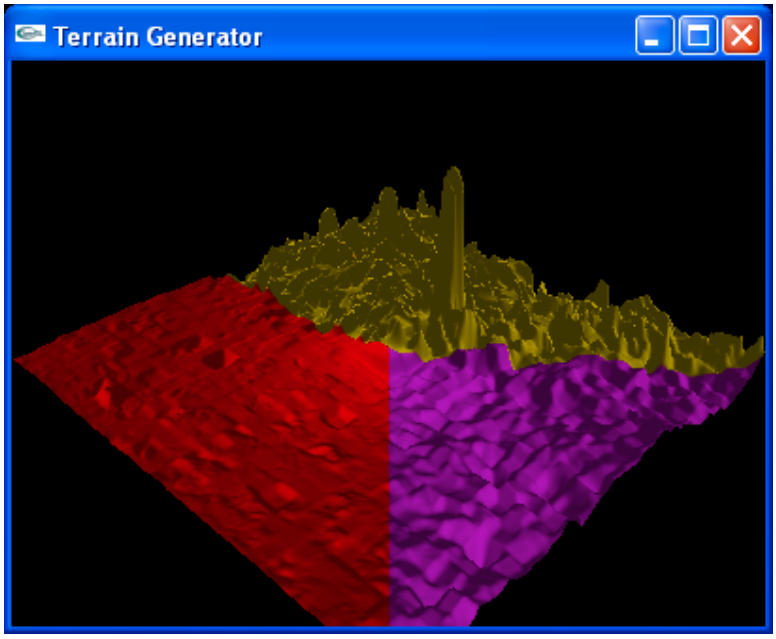

Fig. V.13. The Terrain Viewer Window. The terrain viewer window allows the user to view the generated terrain at different levels of detail and from any angle.

V.3.5. Implementing the Genetic Algorithms

Because GAs are used in multiple places in *Terrainosaurus*, it is advantageous to write the bulk of the GA code in a reusable manner and then to specialize it as needed for each of the GAs. To do this, however, is slightly more complicated than writing reusable library functions, since the parts of the algorithm that need to be specialized are embedded deep within the algorithm (i.e., the reused functionality is the over-arching algorithm, not the low-level functions and objects, as is normally the case when creating a reusable code library).

V.3.5.1. A Generic GA Framework

This pattern of interaction between the reusable and application-specific parts of the code is sometimes referred to *inversion of control*, and implementations of this principle are commonly called *frameworks*. Frameworks can be implemented in a structured programming language through the use of callback functions (the GLU polygon tessellator code is a small example of this), but object-oriented and generic programming constructs (inheritance, polymorphism and templates) make framework implementation much simpler and cleaner. The usual method of creating a framework is for the *hotspots* (application-specific parts of the framework) to be implemented as abstract interfaces, and the rest of the framework to be implemented in

terms of these abstract objects. Then, to specialize the framework for a particular application, all that a developer needs to do is to create application-specific objects to fit into those hotspots.

A suggested way of decomposing a GA as a framework is as follows. To use the framework to solve a particular problem, the developer would subclass some or all of the following classes to implement problem-specific functionality.

Genetic Algorithm	the top-level object of the framework. This object has a number of parameters specifying the overall behavior of the algorithm, such as the population size, number of evolution cycles, and the probabilities of mutation and crossover. In addition to setting these parameters, the developer must add one or more specialized Operator objects to the GA. Once configured, the GA is launched by calling its run() function.
Chromosome	the representation of an individual solution in the algorithm. This class should contain some number of Genes, and the specialized Operators should be written to work on it.
Gene	an atomic sub-part of a Chromosome. Mutation and crossover operators work on these.
Initialization Operator	an operator for performing some arbitrary initialization on a Chromosome. These are called to initialize new Chromosomes as they are introduced into the population (e.g., to replace those that were killed off in the previous evolution cycle).
Crossover Operator	an operator for performing some sort of exchange of genetic material between two Chromosomes. The probability of a crossover operator being invoked on a pair of Chromosome is controlled by the Genetic Algorithm's *crossover probability* parameter.
Mutation Operator	an operator for performing some arbitrary mutation on a Gene. The probability of a mutation operator being invoked on a gene is controlled by the Genetic Algorithm's *mutation probability* parameter.
Fitness Operator	an operator for calculating a fitness value (a scalar in the range $[0, 1]$) for a Chromosome.

A Genetic Algorithm must have *at least* one initialization operator and one fitness operator, but it may have as many operators of each type as desired, and may optionally assign a weight to each registered operator. For the initialization, crossover, and mutation operators, these weights are used to construct a cumulative probability distribution function, which is then used to select which of the available operators is used (with probabilistic preference given to operators with higher weights). In the case of the fitness operators, *each* registered operator is invoked on the chromosome, and the weights are used to determine how significant each fitness component is to the overall fitness calculation.

V.3.5.2. The Boundary Refinement GA

The boundary refinement GA is relatively straightforward to implement, with each gene containing only a relative angle from the previous segment and the absolute angle with respect to the x axis. The operators defined on this GA are:

Init: *Random Angle*	an initialization operator that populates a chromosome with random angles in each gene (subject to the max absolute angle constraint)
Init: *Straight Line*	an initialization operator that populates a chromosome with an angle of zero between consecutive segments
Cross: *Splice Subsequences*	a crossover operator that chooses a split-point and exchanges the subsequences following that point between two chromosomes
Mutate: *Random Bend*	a mutation operator that introduces a random change to the angle in a single gene
Fitness: *Smoothness*	a fitness operator that evaluates the fitness of a chromosome according to the scheme described in Section IV.3.1.4

The boundary refinement GA itself has only one additional parameter beyond the standard parameters belonging the the Genetic Algorithm framework:

max absolute angle	the maximum allowed deviation from the x axis; see Section VI.1.1 in Chapter VI for further discussion of the problems and implications associated with this.

V.3.5.3. The Height Field Generation GA

The height field GA is a bit more complicated than that for the boundary refinement. The Chromosome for this GA contains a 2D grid of Genes, each of which contains pointers to its source Terrain Sample and Terrain Type, as well as a set of transformation parameters. Also, the Chromosome retains the results of the last fitness evaluation (the regions' similarity and gene compatibility measurements), and uses these values to bias the operator probabilities towards operators that are more likely to be helpful (Section IV.3.3.6).

The operators defined for the height field GA are:

Init: *Random Source Data*	an initialization operator that initializes each Gene with a randomly selected source Terrain Sample from the appropriate Terrain Type, and randomly chosen (x, y) coordinates within that Terrain Sample from which to get its elevation data.
Cross: *Swap Rectangular Region*	a crossover operator that swaps rectangular clusters of Genes between two Chromosomes
Mutate: *Reset Transform*	a mutation operator that resets the transformation parameters in a Gene.

Mutate: *Vertical Offset*	a mutation operator that adjusts the mean elevation of a Gene.
Mutate: *Vertical Scale*	a mutation operator that adjusts the vertical range spanned by a Gene.
Mutate: *Vertical Rotate*	a mutation operator that adjusts the rotation around the vertical applied to the elevation values for a Gene.
Mutate: *Horizontal Translate*	a mutation operator that adjusts the (x, y) coordinates used for the source height field data for a Gene.
Fitness: *Gene Compatibility*	a fitness operator that evaluates the similarity between the approximate geometric shape of the elevations controlled by each Gene and the corresponding area of the "pattern" height field from the previous LOD.
Fitness: *Region Similarity*	a fitness operator that evaulates the similarity between each region of generated terrain and the corresponding terrain type that that region is supposed to emulate.

Also, the height field GA has several additional parameters beyond the standard parameters belonging the the Genetic Algorithm framework:

gene size	the width/height (in pixels) of a single Gene—larger values result in fewer Genes being required to cover the entire height field, but also permit less fine-scale modification to the height field; values around 16 pixels seem to work well, at least for lower resolution height fields (up to 90m).
overlap factor	the percentage of linear overlap between adjacent Genes (see Figure IV.18); this controls how much blending occurs between adjacent Genes—a value of zero implies no blending, and would result in discontinuities at gene boundaries; values around 20% seem to work well.
max crossover width	the width of the largest rectangular chunk of Genes that will be swapped during a single crossover operation; if this value is N, this implies that, at most, N^2 Genes will be swapped.
max vertical scale	the maximum factor by which to scale a Gene's elevation values during a single mutation; this controls how drastic of a change the GA is allowed to make.
max vertical offset	the maximum amount by which to change a Gene's mean elevation during a single mutation; this controls how drastic of a change the GA is allowed to make.

V.4. Optimizations & Simplifications

Terrainosaurus is a computationally expensive algorithm; as such, anything that can be done to increase its efficiency is a welcome improvement. Furthermore, it will be easier to implement if existing technologies can be leveraged to solve some of its sub-problems. Toward these ends, I offer several ideas for optimizing and/or simplifying the implementation of the algorithm that were used in implementing the prototype. Further suggestions for optimizing the process that have not yet been explored are discussed in Chapter VII.

V.4.1. Caching the Analysis of Library Terrain Samples

The height field analysis step is, by far, the most expensive part of the algorithm. While this cost cannot be completely eliminated (as each generated height field must be evaluated for fitness), it is at least possible to avoid repeatedly analyzing those height fields belonging to the Terrain Library, since these are essentially static. One way to do this is to dump the results of analyzing an LOD of a height field into a file as soon as it is analyzed. Then, whenever that LOD of that particular height field is needed thereafter, if the dump file is newer than the .dem file from which it was generated, the analysis can be skipped, and the previously calculated results just rehydrated from the file.

In fact, if the dump file is created as a *binary* dump of the Terrain Sample's data structures, the analysis results can be loaded very quickly, even more quickly than loading the original .dem file that they originally came from! This can be attributed to the rather bad compression of the .dem file format, and to the cost of parsing ASCII text into numeric data, which is avoided by reading and writing as binary. One word of warning though: during active development, it is easy to change a data structure, while forgetting the impact that this will have on the dump files. Be sure to verify that your dump files are the size your data structures expect them to be.

V.4.2. Optimizing the Feature Detection Step

Feature detection is the most computationally expensive part of the height field fitness evaluation—speeding this up will result in a significant reduction in overall execution time. Several things can be done to accelerate this step.

V.4.2.1. Do Convolution in the Frequency Domain

The first step in scale-space feature detection is to generate the scale-space representation of height field (height field), which requires convolving the image with Gaussian filters of various sizes. As the size of the filter increases, the convolution becomes more and more computationally expensive to perform in the spatial domain. Fortunately, because of the properties of the Fourier transform, the expensive spatial-domain operation of convolving two images is equivalent to performing ordinary, element-wise multiplication of their frequency domain representations (i.e., their Fourier transforms). The computational cost of this multiplication does not increase as the filter size grows. Therefore, if the amount of convolution to be done is large enough, the computational savings of doing this convolution in the frequency domain will more than offset the expense of performing the forward and inverse Fourier transforms, for a net increase in speed.

V.4.2.2. Save & Reuse Computations

At the risk of stating the obvious, one way of reducing the expense of feature detection is to cache the results of computations rather than recomputing them each time they are needed. Feature detectors typically use first, second, and third, or even higher partial derivatives of the image to compute their response, and these derivatives occur multiple times in the evaluation. Because of this, a significant speed-up can be realized (at the cost of higher memory usage) by creating additional rasters to cache the various derivatives. Furthermore, different detectors often have some of their sub-computations in common; thus, the overall cost of doing both edge and ridge detection can be reduced by keeping the intermediate results from the edge detector and reusing them for the ridge detector.

V.4.2.3. Limit the Number of Scales

The cost of feature detection is proportional to the number of scales being searched. Thus, a good way to limit the expense is to reduce the number of scales. Because of the known, power-of-three relationship between successive LODs of the terrain, it may be possible (or even preferable) to limit the scales searched by the feature detection step to a small number. How significantly this will affect the performance of the GA fitness function is not clear, and is an area for future research (Section VII.11).

V.4.3. Optimizing the Computation of Windowed Statistics

In the "studying" phase of analysis for a Terrain Sample, several statistics are calculated over N×N cell neighborhoods around each cell of the height field. Computing these quantities is very similar to performing convolution with an N×N pixel filter. Just like many filters, these operations are *separable*, meaning that they can be done more efficiently by being evaluated as two sequential 1-dimensional operations: first the statistic is evaluated across the x-coordinate, and then across the y-coordinate of the result.

V.4.4. Simplifying Rasterization of the Map

Generating the Map Rasterization from the Map is a somewhat difficult problem, primarily because, especially with the addition of the refined boundaries, the polygons that make up the map tend to have many, many edges, and to be highly non-convex.

Fortunately, this problem has already been solved. The *GL Utilities (GLU) Library* includes a tessellator for transforming (possibly self-intersecting) non-convex polygons into triangles, which can be rendered easily. Then, to create the Map Rasterization, we need only render these triangles, with an appropriately chosen viewport, and then to read back the rendered pixels from the framebuffer. For this to work, the color with which each triangle is rendered needs to encode the terrain type ID for its corresponding region. Also, it is important to *disable* lighting, antialiasing and alpha blending, so that the rendering system does not interpolate colors (thus destroying this encoding).

Another potential solution, if using *OpenGL* is impossible or undesirable, is to rasterize the boundaries between regions, and then to use a flood-fill algorithm to fill in the regions.

CHAPTER VI

RESULTS & DISCUSSION

In this chapter, I will discuss the results achieved with *Terrainosaurus*, both successes and problems, and with running times and generated images. The sequence of topics will parallel that of the previous chapters:

- the boundary refinement algorithm
- the terrain comparison algorithm
- the height field construction algorithm

VI.1. Boundary Refinement

The boundary refinement operation offers the user a simple means of creating irregular boundaries between regions of terrain, without having to draw every bend in the curve by hand (Figure VI.1, Figure VI.2). In many (if not most) cases, this amount of control is sufficient for the user's needs. It is also a very fast computation (refining the boundaries for an entire map is virtually instantaneous from the user's point of view, using 20 evolution cycles and a population size of 5 for the GA).

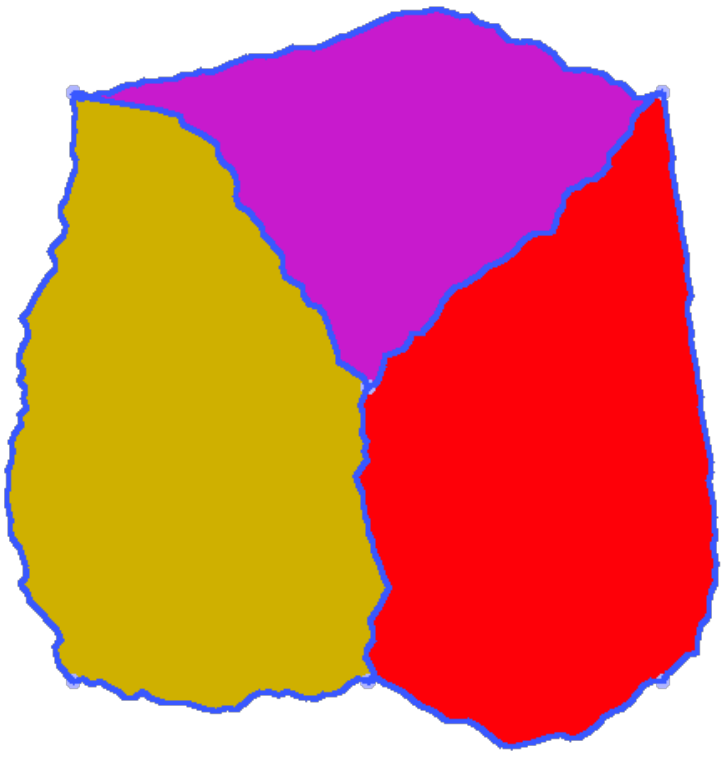

Fig. VI.1. Map Boundaries Refined With S = 0.9.

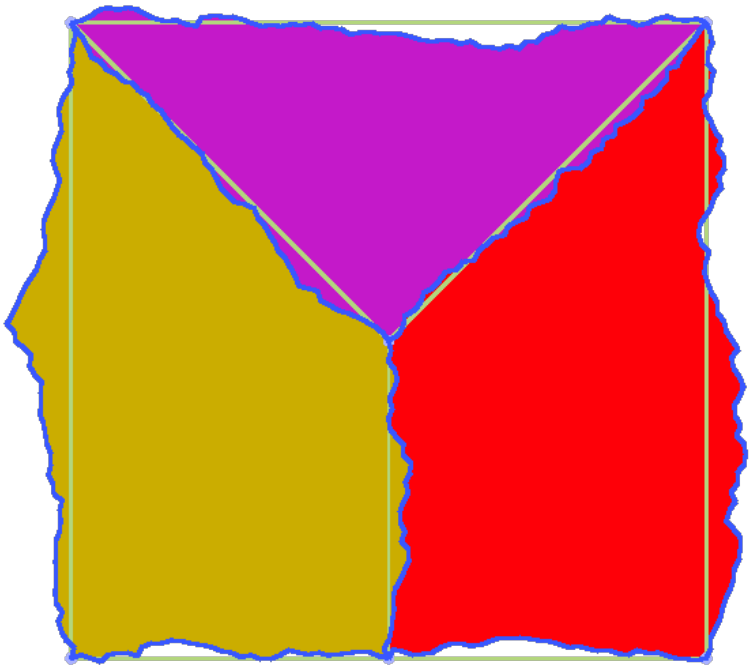

Fig. VI.2. Map Boundaries Refined With S = 0.1.

There are, however, several aspects that deserve some additional discussion.

VI.1.1. The Accumulated Angle Constraint

One item of interest regarding this operation is the global constraint we impose, that the absolute angle is not allowed to exceed a certain threshold at any point. This was found to be necessary in order to force the boundary to make progress in the direction of the end point. Without this constraint it is very possible for the curve to double back on itself, especially if the smoothness parameter is low (thus allowing sharper turns with each segment). This is problematic for a number of reasons.

First of all, it is easy for the curve to intersect itself, producing loops in the boundary. We consider this behavior to be undesirable, since it effectively creates additional regions, and it is not completely clear which of the two adjacent terrain types should fill the new regions (Figure VI.3).

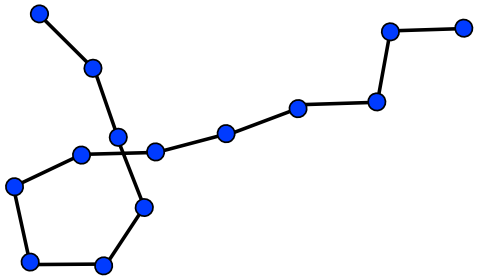

Fig. VI.3. A Self-intersecting Boundary. If no constraints are placed on the boundary GA, it can generate boundaries that loop back on themselves.

Second, it is possible for the generated start and end points to be very near to each other (i.e., the boundary wanders far away, but ends up back near the place from which it started). Such a boundary is a very poor approximation of the original. Worse still, a generated boundary with this characteristic will have to be scaled enormously in order to get the start and end points to line up with the original boundary. The result of this is that the boundary will be magnified to absurd proportions, as compared to the rest of the map, and will probably also intersect several nearby boundaries (Figure VI.4).

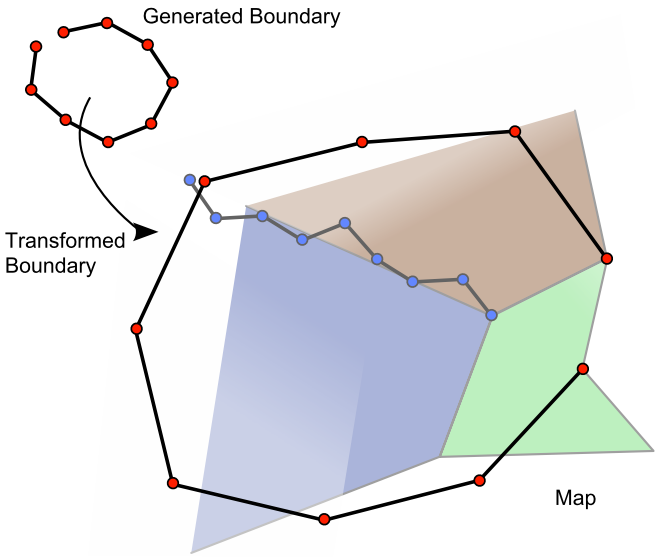

Fig. VI.4. A Badly Scaled, Backtracking Boundary. If a boundary's end point is too close to its start, then the scale factor required to place the generated end points on top of the original end points is huge, causing the boundary to be scaled to absurd proportions.

A third problem, related to the previous two, is that as the refined boundary becomes less linear, the scale factor needed to bring the end points into alignment increases. As the scale factor increases, so do the lengths of the line segments that make up the boundary, thereby thwarting one of the objectives of the boundary refinement operation (i.e., to keep the lengths of individual segments small).

These problems are all prevented or minimized by the global constraint on accumulated angle. Unfortunately, this simplification also excludes certain, valid boundary shapes, in much the same way that height fields cannot represent certain, valid terrain features. Thus, it would be nice to be able to lift this restriction, but in order to do this, we would need some other way of avoiding the aforementioned problems; this is an area for future work.

VI.1.2. Smoothness and Level of Detail

Another topic worth discussing is the effect of the smoothness parameter. In order to keep the curve well-behaved, both locally and at larger scales, we evaluate its fitness at several levels of detail, using the same smoothness value. Thus, the generated curves will display similar behavior at several scales (i.e., they are fractal-like). As a result, the operation is somewhat limited in the kinds of boundaries it can produce: rough, meandering boundaries and smooth, straight boundaries are both possible, but smooth, meandering boundaries are not (since this would imply sharper turns at larger scales and softer turns at finer scales). This behavior could be made more controllable with the introduction of additional smoothness parameters for coarser levels of detail, though the additional benefit might not be worth the added complexity.

VI.1.3. Additional Constraints

As mentioned previously, one of the benefits of a genetic algorithm is its flexibility. While the only constraint currently imposed on the generated boundaries is that they have a user-specified, characteristic smoothness, it would be relatively straightforward to incorporate additional constraints into the fitness evaluation, such as:

- In addition to matching the locations of the endpoints of the original boundary, the refined boundary should also match specific angles at the endpoints. This would make it possible to eliminate sharp "corners" from regions.
- The refined boundary should not intersect other, nearby boundaries.
- The refined boundary should not have any self-intersections, nor should it end near to where it began. This would help to address the problems discussed in the previous section about the global constraint on the accumulated angle.
- The refined boundary should remain within a user-defined "envelope". This would provide the user with additional control over the shape of the boundary.

VI.2. Terrain Library Analysis

VI.2.1. Empirical Analysis

In order to validate the claim that the statistics used are suitable for establishing similarity, we examined 56 terrains taken from 7 US states. The terrains were grouped into 18 terrain types, the smallest of which

contained only one example, and the largest of which contained 8 examples. Classification was done based on visual inspection by one user.

Visual comparison of histograms in *Matlab* revealed a high degree of similarity between the histograms of the same terrain type, when compared to those of other terrain types. In most cases, the mean values were fairly close, and the histogram had the same approximate shape (the example in Figure VI.5 is fairly typical). Because of this, we conclude that these statistics *are* meaningful in evaluating similarity.

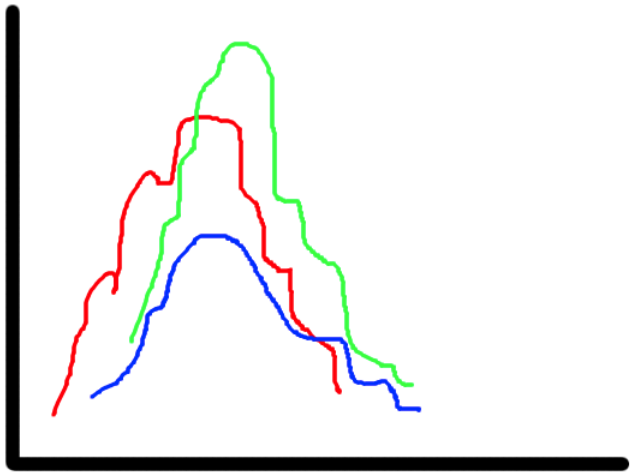

Fig. VI.5. Elevation Histograms from the California Coast Hills. The elevation histograms for several samples of similar terrain, taken from a region of southern California. The overall histogram shapes correspond fairly well to one another.

The terrain library also scored well against the similarity function. Of course, every terrain sample received a self-fitness score above 90%—this is true by definition, so it means nothing. What is significant is that agreement measurements were also high, typically above 90% for most measurements. In contrast, when a sample of steep, Wyoming mountains was "misclassified" among samples of Florida flatland, many of the agreement measurements dropped from 97%–99% to under 60% (and in once case, nearly to 0%). This agrees with conclusions drawn from visual inspection of the histograms.

VI.2.2. The Similarity Function

The most difficult part of *Terrainosaurus* was constructing an effective terrain type similarity measurement. While there is certainly room for improvement, the function described in Section IV.3.2 does a reasonably good job of favoring more realistic terrain, at least with the terrain types used to test it (e.g., Figure VI.6, Figure VI.7, Figure VI.8, Figure VI.9). This success can be attributed to several desirable characteristics:

- it scores all of the reference height fields highly (similarity of 90% or better) without over-fitting the data; this allows it to generalize effectively in order to accept new data
- it is able to detect when a particular measurement is useless for evaluating a particular terrain type and ignore that measurement; furthermore, it can also detect when its overall discriminating power is weak due to bad input data
- it is not sensitive to any similarity between terrain samples belonging to different terrain types; this allows it to be tolerant of overlapping (or even identical) terrain types created by the user

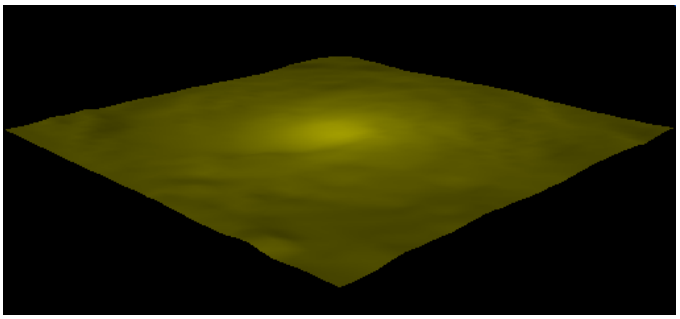

Fig. VI.6. A Reference Height Field from Florida.

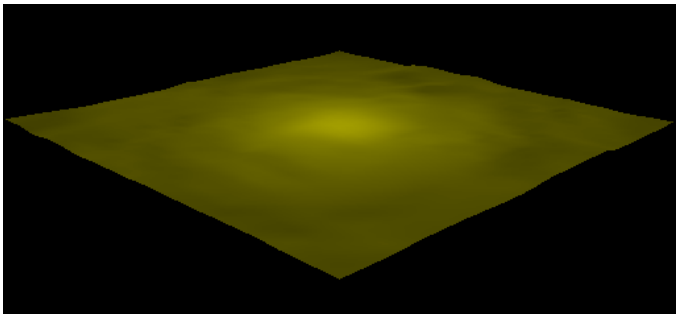

Fig. VI.7. A Generated Height Field Based on the Florida Reference.

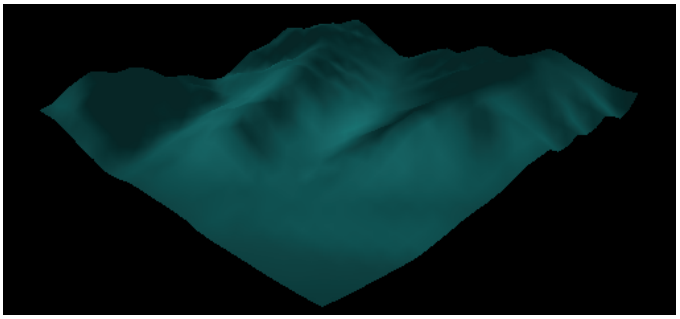

Fig. VI.8. A Reference Height Field from Washington.

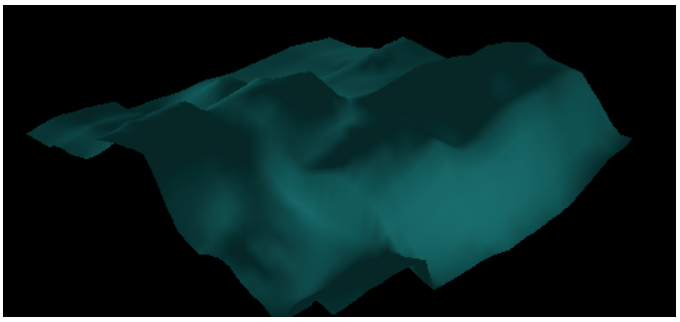

Fig. VI.9. A Generated Height Field Based on the Washington Reference.

VI.2.2.1. On the Importance of Classifying Well

The user's only job in the terrain library analysis process is to segregate the example terrains into meaningful terrain types. It is not important for these terrain types to be *disjoint* with respect to one another (i.e., no overlap in measured statistics); to require this of the user would be overly taxing, and is not reflective of the real world either (many diverse terrain types share similar mean elevations, for example). One of the strengths of this algorithm is that the user is allowed to create as many, finely-distinguished terrain types as he needs to suit his purposes.

It *is* important, however, that each terrain type be *coherent*. The similarity function is designed to adapt itself to whatever patterns it is able to discover within the examples given to it—if the examples given to it are essentially unrelated, the function will be unable to learn anything meaningful, and will, accordingly, produce garbage. When the similarity function described in this paper was originally implemented, it produced initially poor results; this turned out to be the result of several badly-classified examples.

A related issue that also poses problems for the similarity analysis is the presence of *multiple* terrain types within a single terrain sample, such as a lake in the midst of mountainous terrain, or mountains trailing off into a flat plain. These situations can yield statistical distributions that are uncharacteristic of *any* of the terrain types involved, often significantly skewed or multi-modal. Because *Terrainosaurus* does not currently detect these conditions, such example terrains cannot be used.

Addressing these issues would go a long way towards improving the overall user experience in this phase, and is an area for future work (Section VII.5).

VI.2.2.2. Things that Didn't Work

In arriving at this similarity function, a number of things were attempted that turned out not to work effectively. I mention a few of them here in the hopes of providing guidance and inspiration to future researchers in this area: guidance—that the dead-ends of the past need not be revisited—and inspiration—that one of these failed ideas might be the seed that one day sprouts into an idea that *does* work.

Direct Height Field to Height Field Comparison

Early in the research process, we attempted to use the RMS (root mean square) difference between the generated height field and a reference example as an estimate of their similarity. This was never intended to be the final similarity function, but only a temporary, partial solution. Just the same, it is instructive to consider the problems from which it suffers, as similar pixel-wise approaches will likely have many of the same failings:

- it is quite sensitive to the precise placement of features in the terrain; as such, it is too restrictive to be the basis for a real similarity function, as it cannot even relate similar *example* terrains to one another; consider comparing a sloping terrain to its mirror image—this would receive a low similarity score even though it is comparing a terrain *to itself*!
- it is not clear how to generalize a pixel-by-pixel comparison such as this to handle multiple example terrains in a terrain type
- it is extremely sensitive to the mean elevation of the terrains: a large difference in mean elevations will inordinately penalize two otherwise very similar terrains
- it is not even clear how to compare two rectangular terrains of different dimensions, much less two non-rectangular regions
- as the RMS difference between two terrains is effectively unbounded, this value cannot, by itself serve as a similarity measure, and it is unclear how to adequately transform this unbounded value into a bounded value (this is only an issue if fitness-proportional selection is used in the GA; an unbounded fitness function can still work if tournament selection is used in the GA)

Comparing the Fourier Transform

Another idea we attempted was to compare the Fourier transforms of terrains. This gets around the problem of the previous idea, that the height fields needed to be of the same size, since the Fourier transforms can be resampled to the same size and compared directly. However, attempting to compare the FFTs magnitudes of apparently similar terrains did not yield promising results, and so we abandoned this approach.

Linear Pattern Analysis

As a third approach, we tried to apply standard pattern analysis tools to discover automatically the relationships between examples of the same terrain type. The main difficulty motivating this approach was that of identifying which characteristics of a set of terrains are most important. In order to choose which terrain chromosomes to keep and which to "recycle" during the GA, a good means of ranking them is needed...but given the wide variety of possible characteristics that could be used, it is hard to see which should be given preference (or whether all should be weighted equally).

Fisher's linear discriminant (FLD) [Gutierrez-Osuna 2004] is a common pattern recognition technique for *dimensionality reduction*, in which a large number of characteristics can be projected down to a smaller set. FLD chooses the projection that maximizes separation between the different classes of data (i.e., in our case, the terrain types). It also calculates a separability ratio, which can be used as a relative measurement of how well a set of characteristics separates the classes. Using this separability ratio, we thought to find

an optimal set of characteristics (and weights for those characteristics) with which to compare and evaluate terrain height fields.

While this approach initially showed some promise, it turned out to have some serious problems. The fatal flaw with using FLD is that it solves the wrong problem: it produces the combination of characteristics that *show the biggest difference* between the terrain types in the library. This has the effect of ignoring characteristics that all terrain types have in common, even if those characteristics turn out to be important for producing that terrain type. What we actually want is the set of characteristics that *most strongly characterize* each terrain type—this set might be different for each terrain type. Given a terrain height field, FLD would be useful for helping to answer the question "To which of terrain type does it most likely belong?", whereas the *real* question we want to answer is "How much like its reference terrain type is it?".

A strange consequence of using a similarity function based on FLD is that the similarity between two height fields of the same terrain type cannot be determined *except through opposition to every other terrain type*. This is wildly counter-intuitive: adding new terrain types or adding new examples to an existing terrain type should have no effect on the *other* terrain types.

Related to this, in order for an FLD-based similarity function to work, all terrain types must be significantly different from one another. This places an extra burden on the user: he must ensure, not only that the examples in each terrain type are similar to each other, but also that they are different from all of the other terrain types. Furthermore, such a similarity function is conceptually opposite to what is really wanted: a classifier-based system attempts to maximize separability; that is, *differences* between terrain types are made more important than the *similarities* within a terrain type.

A final problem with FLD is that it requires a significant number of examples in each class to work (if there are too few samples, a matrix becomes singular and thus cannot be inverted). As the number of characteristics under consideration increases, the requisite number of examples increases as well. Because of this, FLD is not a good candidate for a similarity fitness function: it doesn't work at all with too few samples.

Histogram Aggregation

A fourth approach, which bears a stronger resemblance to the similarity function we ultimately used, but still turned out to be fatally flawed was to create one giant, terrain-type-wide *super histogram* (Figure VI.10) for each statistic (e.g., elevation, slope) used in the comparison. These would be computed by adding together the individual histograms from each sample in the terrain type, and normalizing the super histogram to have an area of 1. In theory this composite histogram would be smoother than those of the individual examples, and would better represent the terrain type as a whole. Then, to evaluate the similarity of a generated height field to this terrain type, one need only calculate the RMS difference between the height field's normalized histogram and the terrain type's super histogram.

While this idea led to the similarity function described earlier, it has several problems. The first problem is the difficulty of combining histograms with different bucket sizes, though this might be solved by resampling the histogram. The more serious problem is that, unless the distribution means are very close, the resulting super histogram will be multi-modal. Obviously, such a distribution will not compare well to the (usually) unimodal distributions that produced it.

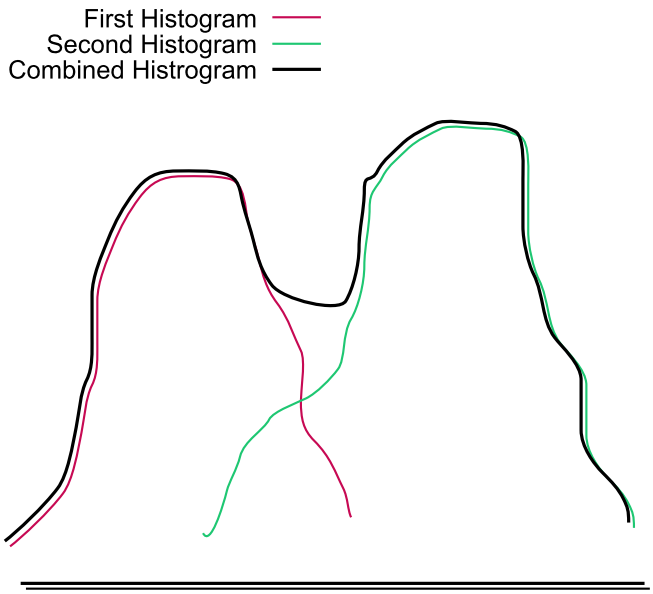

Fig. VI.10. A "Super Histogram". Concatenating the histograms of multiple terrain samples turned out to be a fatally flawed approach.

VI.3. Height Field Construction

The height field construction algorithm is able to create a reasonably good imitation of the example terrains given to it, providing a computationally-expensive, but low effort means of generating terrain.

VI.3.1. Performance of the GA

The height field GA is, without a doubt, the most computationally intensive part of the application. As the LOD increases, so does the computation time (see Table VI.1).

Table VI.1. Height field generation running times.

LOD	**Time (s)**
270m	40 s
90m	270 s
30m	2070 s

These numbers are for one run on a 1.2 GHz Pentium III with 256 MB of RAM. The running time ratios between successive LODs are unsurprising: each successive LOD takes between 6 and 8 times as long as the one before. The GA itself (ignoring the fitness function) is linear in the number of genes in a chromosome. Similarly,the fitness function is linear in the number of pixels in the height field. Since each successive LOD is approximately 9 times larger than the previous, it makes sense that the running times would scale similarly.

Obviously, scalability is an issue with this algorithm. As it is now, the 30m LOD is at least reachable. Unfortunately, the 10m LOD can be expected to take a 6 to 8 times as long again. Before the algorithm can be pushed to these higher LODs, it must be accelerated somehow, whether via a GPU implementation, or some other form of parallelism.

VI.3.2. Successfulness of the GA

From the figures in this section, it is obvious that the algorithm works fairly well, at least at the 270m (Figure VI.11) and 90m (Figure VI.12) LODs. The terrain types bear a significant resemblance to their reference examples, and the transitions between regions work well too. Unfortunately, when going to the 30m LOD (Figure VI.13), this does not hold true. According to several GA researchers with whom I spoke at a recent conference on evolutionary programming, as the size of the problem increases, the GA population size should increase accordingly. This is problematic, as the running time for the 30m LOD is already quite large.

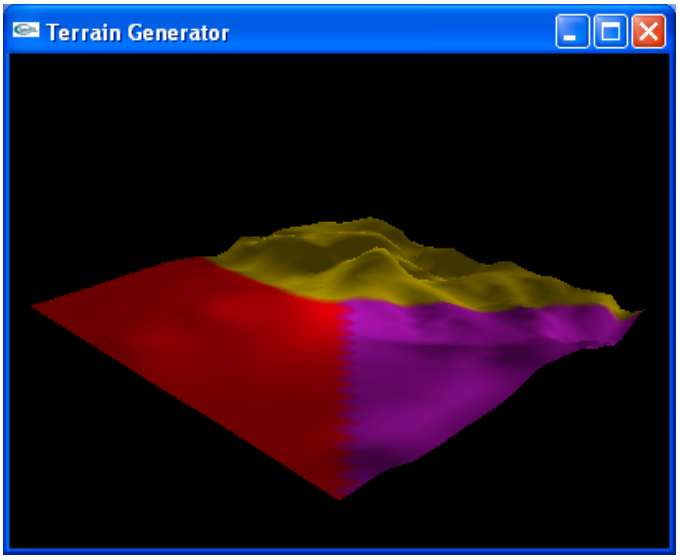

Fig. VI.11. "Thirds" at 270m. At the 270m resolution, the terrain types display characteristically different elevation patterns.

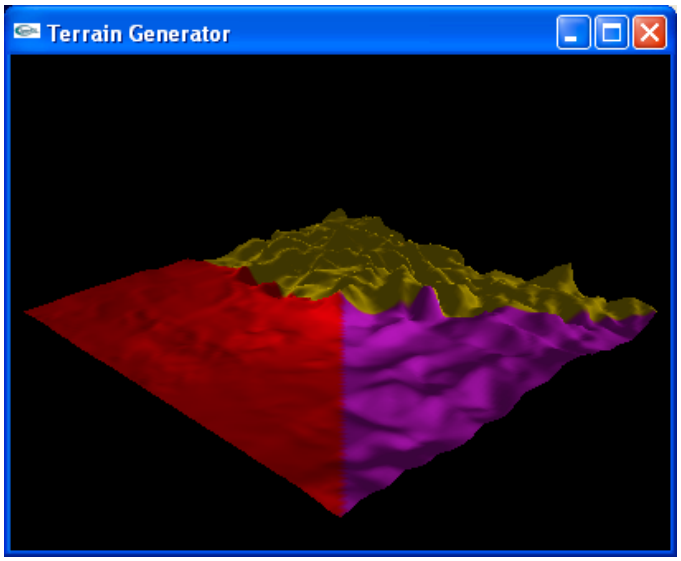

Fig. VI.12. "Thirds" at 90m. At the 90m resolution, the terrain types still look fairly reasonable, though it would be nice to see larger, more coherent features in the hilly terrain (yellow).

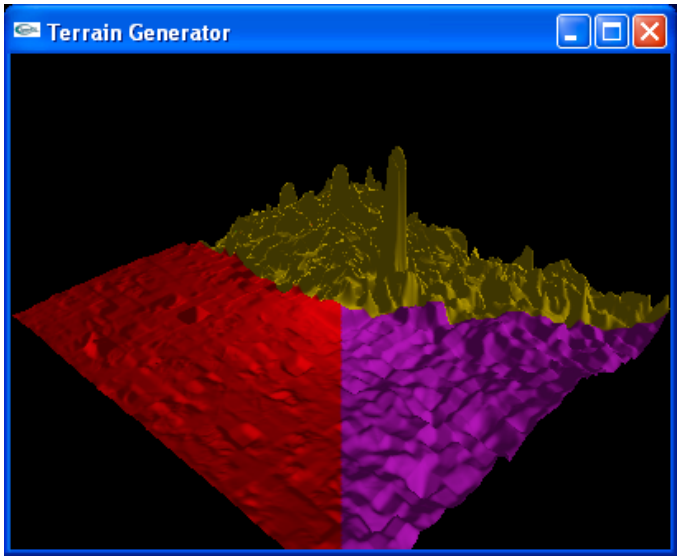

Fig. VI.13. "Thirds" at 30m. At the 30m resolution, the size of the terrain has exceeded the ability of the GA to bring it all together.

Further research into tuning the GA parameters is likely to yield better results; still, there are some other ways in which the construction process could be improved as well. Probably the most noticeable flaw in the 30m height field (Figure VI.13) is the "tiling" effect visible in the flatter areas. This effect can also be seen in the mountainous, yellow area, though it is a bit less glaring due to the already rugged terrain in that area.

This effect is the result of the genes not aligning well with one another. One possible way of dealing with this issue would be to subdivide the genes into smaller patches (2×2 or 3×3 would be a good start) for the purpose of calculating gene compatibility. This would yield a more accurate estimate of the shape of a gene, and would hopefully lead to finding genes that naturally fit well to the pattern height field, which would reduce or eliminate the tiling effect.

CHAPTER VII

FUTURE WORK

While the results achieved with this method are respectable, numerous extensions and improvements are possible, offering greater performance, a higher degree of realism, greater ease of use, and/or additional control over the generated terrain.

VII.1. User Study

One of the main objectives of this research has been to develop a method of generating terrain that is easy and intuitive for a user to use. In pursuit of this goal, I have had to make a number of judgments as to the user-friendliness of particular aspects, armed primarily with my own intuition as a user and creator of graphically-oriented software. While I have some degree of experience using painting, illustration, modeling and animation software, I am unquestionably more technically adept than the average user of such software. Furthermore, as the programmer of *Terrainosaurus*, I have the luxury/handicap of knowing all the details of its implementation. Hence, it would be beneficial to evaluate the effectiveness of *Terrainosaurus* with groups of professional 3D artists, architects, and simulation/game designers to see how effectively they are able to use it, what features they like, and what features they find to be either constraining or conspicuously absent.

VII.2. Placement of Features

In *Terrainosaurus*, the user's control over the design of his terrain is region-based: all user modifications to the shape of the terrain are made through creating regions of various shapes and sizes, and assigning terrain types to them. If the user wants a finer degree of control over one area of the map, he creates smaller regions and/or a finer taxonomy of terrain types. This works well enough for exercising fine control over the size and shape of regions, and the spatial relationships between them, but is less effective at guaranteeing the presence of particular features (e.g., rivers, volcanoes, cliffs etc.) in the places the user wants them.

A very useful extension to this method would be to allow the user to specify the geometric attributes of particular features (e.g., the path of a river, the location and elevation of a mountain peak, etc.) during the design phase. In the generation phase, these feature specifications would impose additional, localized constraints for the GA to meet, and would influence or override the shape of the generated terrain in that area. Such an extension would provide the user a whole new level of control over the generation process, with no necessary increase in complexity (if the user doesn't care about the placement of specific features, he can simply not use that extension; then the terrain generation process is no different than described above).

There are at least two challenges in creating such an extension. First, the definition of what constitutes a "feature" in this context is a bit hazy. How large of an area around a user-placed mountain peak should be considered "the mountain"? And what sort of data structure should be used to represent the "feature"? One possibility is to let features be areas (not necessarily rectangular) of height field data isolated from the input terrain samples. This has the benefit of providing a natural relationship between the identified features and

the terrain types in which they occur, but it is not clear how this would encompass features like rivers (since they can be of arbitrary length).

Besides the difficulty of defining and representing distinct features, there is also the problem of maintaining physically correct relationships in the vicinity of these features. For example, if the user places a cliff or a waterfall in a particular location, then it becomes necessary to ensure that the area "above" that cliff/waterfall has a higher elevation than the ground "below" the feature. Or, to give another example, when placing rivers, it may be necessary to create lakes or cut gorges in order to maintain the constraint that a river can never flow uphill.

Both of these challenges suggest that terrain synthesis using individually placed features may add a significant amount of complexity to the generation process. A good first step toward understanding this problem might be to identify a representative set of features (mountains, rivers, gorges, waterfalls, alluvial fans, cliffs, etc.) and to analyze the geometric "environment" in which each feature can exist. This might give some insight into how to represent a feature and how to incorporate its constraints into the generation process.

VII.3. Automatic Map Construction

In the usual case, a user of *Terrainosaurus* will want to exercise control over the general layout of the terrain, so leaving the layout of the terrain type map in the hands of the user makes a lot of sense. Nevertheless, there are cases in which it would be useful to be able to create a plausible map automatically, such as a game engine creating random worlds, or an artist looking for inspiration. Therefore, some sort of higher-level mechanism for automatic map construction (possibly another GA), could be a useful extension.

VII.4. Automatic Generation of Textures & Objects

As lovely as the terrains generated by *Terrainosaurus* are, they are a bit lacking in visual realism without appropriate textures and objects. Without these sorts of visual cues, the illusion of "the real world" will never be complete. Thus, an important companion task to the generation of the landscape is the synthesis and placement of realistic textures and objects.

Although the generated terrain model could be textured in a number of ways, ideally we would like to be able to texture the generated model automatically, using the geometry of the height field and the terrain-type map to guide the process. A genetic algorithm approach similar to that employed for generating the height field might be very successful at generating believable textures. Taking this idea a bit further, the placement of three-dimensional objects (such as natural objects like rocks and trees, or man-made objects like bridges and houses) could also be automated. By taking the geometry and terrain type into account, more realistic results could be achieved (e.g., trees should not be placed above the treeline elevation, houses built on swamp land could be built on stilts).

VII.5. Computer-aided Terrain Classification & Segmentation

Currently, the most tedious and error-prone part of this approach is the construction of the terrain type library: the process is completely dependent on a human to classify terrain samples correctly, and also

requires the person doing the classification to avoid samples containing mixtures of terrain types (such as lakes, seashore, etc.).

A useful extension would be to incorporate the feature analysis into this earlier stage as well. The computer could analyze new terrain samples as they are being added, compare them to those already present in the library, and inform the user of how similar it is to those already in the library, and how a particular classification of the sample would affect the aggregate statistics that control the similarity function (i.e., would adding this terrain to the "mountains" category significantly diminish the agreement between "mountains" examples?). This information could help the user to make a better decision in classifying the sample.

Ideally, the terrain library would be able to admit any height field as a terrain sample. As discussed in Section VI.2.2.1, terrain samples containing multiple terrain types, are currently unusable. In order to make them usable, the terrain library needs to handle non-rectangular sub-regions of the height field, allowing a terrain sample to be segmented into its constitutent terrain types. Some of this can be done automatically (since a body of water has constant elevation, the computer should be able to segment water from non-water quite easily), but segmenting other types of terrain would require fuzzy boundaries and more sophisticated segmentation algorithms. Perhaps an adaptation of the user-initiated classification procedure described in Gill's paper on ice classification [Gill 2003] would be effective for this.

VII.6. Terrain Type Interpolation

Some types of terrain occur only near certain other types. For example, it would be unusual to find sandy terrain in the middle of a grassy plain (unless it's a golf course). On the other hand, sandy beaches are ubiquitous near the ocean. Terrain types such as this could be viewed as *transitionary* types and could be introduced automatically near the user-generated regions of the neighboring terrain type. So, for example, the user could create a region of "plains" adjacent to "mountains" and the height field construction algorithm would introduce some "mountain foothills" around the boundary between them.

VII.7. More Intelligent Construction of the Base LOD

One weakness of *Terrainosaurus*, in the normal case where multiple terrain types are present in the map, is that the algorithm for constructing the coarsest LOD is rather naive: a simple copy-and-paste operation with blending near the seams to prevent sharp drops. This can have a disconcerting, unrealistic effect when the mean elevations are significantly different. A more intelligent means of constructing this initial LOD could eliminate this problem.

VII.8. Enhanced Similarity Function

The current fitness function evaluates generated terrain regions for similarity to the reference terrain type by comparing a few of the more obvious characteristics of terrain. There must be other characteristics that could be incorporated to give additional discriminating power to the similarity function and thus improve the quality of the generated terrains. Some ideas for future investigation are:

- the spatial "density" of features—how close together they typically occur, how "clumped" they are

- the distance to the terrain type boundary—how far from the transition between terrain types certain features typically occur (e.g., we would expect a mountain peak of any significant size to occur towards the interior of a mountainous region)
- "directionality" of features—to what extent certain features exhibit the same directional tendency; this might be necessary to reproduce terrains strongly affected by wind erosion, for example
- frequency-spectrum information—though direct comparison of Fourier transform coefficients did not yield useful results (Section VI.2.2.2), it may be that a more sophisticated frequency analysis would do so
- higher-order derivatives of the surface—how the curvature varies across the terrain surface

VII.9. Cross-LOD Analysis

The iterative, multi-LOD height field generation approach used in *Terrainosaurus* is based on the observation that different features become visible at different scales. Hence, it is reasonable to consider them as "belonging" to different LODs and generate them accordingly. No attempt has been made, however, to look for relationships between features at different LODs. It might be, for example, that fine-scale ridges tend to occur nearby and perpendicular to larger-scale ridges in certain types of mountain ranges. It seems likely that there would be many relationships of this sort that could be exploited to achieve more believable terrain models, though how to discover and apply these relationships is unclear.

VII.10. Enhanced Mutation & Crossover Operators

In conjunction with a better terrain similarity function, it would also be nice to have some smarter mutation and crossover operators. One way in which the operators might be made more intelligent is with regard to the formation of features in the terrain. Rather than naively copying rectangular regions, a crossover operator could copy contiguous genes that span an identified feature. Similarly, a mutation operator might push the source coordinates for a gene closer to or further from a feature detected in the source height field, in an attempt to find more plausible source material, given the state of features forming in the generated height field.

VII.11. Performance Improvements

Unfortunately, the height field generation phase is a bit on the slow side. This is partially attributable to the use of a genetic algorithm, but there are also other areas of the process that contribute to its slowness.

The use of a genetic algorithm in the height field generation process has certain benefits, such as the enormous amount of flexibility it affords, but is not without its drawbacks. The most obvious of these is its runtime complexity, as discussed in Section VI.3.1. It might be possible to achieve similar results using a different, more efficient optimization algorithm (though doing so might preclude the implementation of some of the other improvements described above).

The most CPU-intensive part of the fitness analysis is the feature detection step; scale-space feature detection is a rather expensive operation. A useful topic for future research would be to investigate how the diagnostic power of the feature detector changes as the range and number of scales searched decreases. Because the terrain construction process focuses at each step on generating detail at a particular LOD (the detail at coarser

LODs is already essentially fixed, and the detail at finer LODs does not exist yet), it might even be that a single-scale detector could perform as well as or better than a multi-scale detector.

Another aspect of the algorithm with a lot of potential for optimization is its inherent parallelizability. A large proportion of the computations performed during the height field generation algorithm are done once per element (whether an "element" is a pixel, a height field cell, or a gene), with relatively few conditional branches and data dependencies, and so might be able to benefit from a symmetric multiprocessing system, or better yet, implementation on a modern GPU. The large memory sizes and programmability features of recent GPU architectures suggest that it might be possible to run large parts of the construction algorithm entirely on the GPU.

CHAPTER VIII

CONCLUSION

Terrain generation is a topic of interest to practitioners in a number of fields, some of which are entertainment- and art-related, with others being more utilitarian in nature. This topic has received significantly less treatment in the literature than has the related topic of real-time terrain visualization. Terrain generation methods currently in practice are usually fractal in nature, and are often difficult to control.

Terrainosaurus is a new, user-friendly method of generating an effectively unlimited diversity of 3D terrain models. It differs from current state-of-the-art methods for terrain generation in a number of ways, in particular:

- it uses user-provided, real-world elevation data as raw material. Because of this, a user can extend the capabilities of the system without changing the fundamental algorithm, simply by adding examples of new types of terrain.
- it follows a user-centric design paradigm, where types of terrain are described by example, and the desired arrangement of these terrain types is specified in an intuitive manner (i.e., by drawing a map). In this paradigm, the user is freed from nearly all of the manual labor that characterizes manual "sculpting" methods for height field creation, and is not required to learn esoteric skills in order to get a desired result, as in many procedural methods.
- it uses artificial intelligence techniques, specifically genetic algorithms, to generate a height field approximating the user's design and manifesting the appropriate terrain characteristics in each distinct region. To my knowledge, this is the first significant attempt to apply artificial intelligence techniques to the problem of terrain generation.

The second contribution of *Terrainosaurus* is a paradigm for terrain generation in which the user is freed from almost all of the burden of constructing the terrain, while at the same time still retaining some control over the shape of the terrain. All of the inputs expected of the user are intuitive to grasp. Furthermore, it is not necessary for the user to understand how *Terrainosaurus* works in order to use it effectively. These characteristics will become increasingly important in the future as the scale of virtual worlds continues to increase—development teams simply will not have the time to design the terrain manually.

The third contribution of *Terrainosaurus* is a new method of performing approximate comparisons of terrain height fields using a variation on statistical distribution matching. This comparison method is adaptive, is not highly sensitive to differences in the size or shape of the terrains being compared, and has a built-in measure of how well it is performing in any given case.

Terrainosaurus, while not as trivial to implement as more simplistic algorithms, could be of enormous use in a studio authoring environment, especially when integrated with 3D modeling tools for manual fine-tuning.

Towards this end, it would be useful to implement this algorithm as a plugin for one or more currently popular 3D modeling packages.

Obviously, terrain generation is not yet a solved problem. The positive results produced by *Terrainosaurus* suggest that even more promising results will emerge with additional research. It is my belief that future research in this field ought to pursue a course similar to *Terrainosaurus*'s—the use of artificial intelligence techniques seems a promising road (and perhaps the *most* promising) for doing terrain generation in a way that is both realistic and user-friendly.

REFERENCES

[Baumgart 1975] Baumgart, B. Winged Edge Polyhedron Representation for Computer Vision. In *Proceedings of the 1975 National Computer Conference*. AFIPS Press. Arlington, VA. 589–596.

[Bungie 2004] Bungie Studios. 2004. Halo 2 *(software)*. Microsoft Corporation. Redmond, WA.

[Burke 1996] Burke, C. 1996. Generating Terrain. http://www.geocities.com/Area51/6902/terrain.html.

[Cormen et al. 2001] Cormen, T., Leiserson, C., Rivest, R., and Stein, C. 2001. Introduction to Algorithms, 2nd Ed.. MIT Press. Cambridge, MA.

[DAZ 2006] DAZ Productions. 2006. Bryce 3D *(software)*. DAZ Productions. Draper, UT. http://www.daz3d.com/program/bryce/.

[Duchaineau et al. 1997] Duchaineau, M., Wolinsky, M., Sigeti, D., Miller, M., Aldrich, C., and Mineev-Weinstein, M. 1997. ROAMing Terrain: Real-time Optimally Adapting Meshes. In *Proceedings of IEEE Visualization '97*. IEEE Computer Society Press. Los Alamitos, CA. 81–88.

[Electronic Arts 2003] Electronic Arts. 2003. SimCity 4 *(software)*. Electronic Arts.

[Epic Games 2004] Epic Games, Inc. 2004. Unreal Tournament 2004 *(software)*. Atari, Inc. Lyon, France. http://www.unrealtournament.com/.

[ESRI 2006] ESRI. 2006. ArcGIS *(software)*. ESRI. Redlands, CA. http://www.esri.com/software/arcgis/.

[Fernandez 2006] Fernandez, A. 2006. Lighthouse 3D - Terrain Tutorial. http://www.lighthouse3d.com/opengl/terrain/index.php3.

[Fournier et al. 1982] Fournier, A., Fussell, D., and Carpenter, L. 1982. Computer Rendering of Stochastic Models. In *Communications of the ACM*. Vol. 25. No. 6. ACM Press. New York, NY. 371–384.

[Franke 2000] Franke, S. 2000. Spectral Synthesis Noise for Creating Terrain. In *GameDev.net*. GameDev.net. http://www.gamedev.net/reference/articles/article900.asp.

[Friggo & Johnson 2003] Friggo, M. and Johnson, S. 2003. FFTW: The Fastest Fourier Transform in the West, Version 3 *(software)*. Massachusetts Institute of Technology. Cambridge, MS. http://www.fftw.org/.

[Gallant & Hutchinson 1996] Gallant, J. and Hutchinson, M. Towards an Understanding of Landscape Scale and Structure. In *Proceedings of the Third International Conference/Workshop on Integrating GIS and Environmental Modeling*. National Center for Geographic Information and Analysis. Santa Barbara, CA.

[Gamasutra 2006] Gamasutra. 2006. Gamasutra: The Art & Business of Making Games. CMP Media LLC. San Francisco, CA. http://www.gamasutra.com/.

[GameDev.net 2006] 2006. NeHe Productions. GameDev.net. http://nehe.gamedev.net/.

[Gill 2003] Gill, R. SAR Surface Ice Cover Discrimination Using Distribution Matching. In *Proceedings of POLinSAR 2003*.

[Gutierrez-Osuna 2004] Gutierrez-Osuna, R. 2004. Lecture Notes: Course in Pattern Recognition. PRISM Group at Texas A&M University. College Station, TX. http://research.cs.tamu.edu/prism/lectures.htm.

[Kelley et al. 1988] Kelley, A., Malin, M., and Nielson, G. 1988. Terrain Simulation Using a Model of Stream Erosion. In *Proceedings of SIGGRAPH '88*. ACM Press. New York, NY. 263–268.

[Lindeberg 1998,1] Lindeberg, T. 1998. Feature Detection with Automatic Scale Selection. In *International Journal of Computer Vision*. Vol. 30. No. 2. Springer. New York, NY. 77–116. ftp://ftp.nada.kth.se/CVAP/reports/cvap198.pdf.

[Lindeberg 1998,2] Lindeberg, T. 1998. Edge Detection and Ridge Detection with Automatic Scale Selection. In *International Journal of Computer Vision*. Vol. 30. No. 2. Springer. New York, NY. 117–154. ftp://ftp.nada.kth.se/CVAP/reports/cvap191.pdf.

[Li et al. 2003] Li, S., Liu, X., and Wu, E. 2003. Feature-based Visibility-driven CLOD for Terrain. In *Proceedings of Pacific Graphics 2003*. IEEE Computer Society Press. Los Alamitos, CA. 313–322.

[Losasso & Hoppe 2004] Losasso, F. and Hoppe, H. 2004. Geometry Clipmaps: Terrain Rendering Using Nested Regular Grids. In *Proceedings of SIGGRAPH 2004*. ACM Press. New York, NY. 769–776.

[Mandelbrot 1982] Mandelbrot, B. 1982. The Fractal Geometry of Nature. W. H. Freeman. New York, NY.

[Marshall et al. 1980] Marshall, R., Wilson, R., and Carlson, W. 1980. Procedure Models for Generating Three-Dimensional Terrain. In *Proceedings of SIGGRAPH '80*. ACM Press. New York, NY. 154–162.

[Microsoft Game Studios 2004] Microsoft Game Studios. 2004. Flight Simulator 2004 *(software)*. Microsoft Corporation. Redmond, WA. http://www.microsoft.com/games/flightsimulator/.

[Musgrave et al. 1989] Musgrave, F., Kolb, C., and Mace, R. 1989. The Synthesis and Rendering of Eroded Fractal Terrains. In *Proceedings of SIGGRAPH '89*. ACM Press. New York, NY. 41–50.

[Obitko & Skavík 1999] Obitko, M. and Skavík, P. 1999. Visualization of Genetic Algorithms in a Learning Environment. In *Proceedings of Spring Conference on Computer Graphics '99*. Comenius University. Bratislava, Slovakia. 101–106.

[Pajarola et al. 2002] Pajarola, R., Antonijuan, M., and Lario, R. QuadTIN: Quadress-based Triangulated Irregular Networks. In *Proceedings of IEEE Visualization 2002*. IEEE Computer Society Press. Los Alamitos, CA.

[Pandromeda 2004] Pandromeda, Inc. 2004. MojoWorld *(software)*. Pandromeda, Inc. New Creek, WV. http://www.pandromeda.com/.

[Parish & Müller 2001] Parish, Y. and Müller, P. 2001. Procedural Modeling of Cities. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*. ACM Press. New York, NY. 301–308.

[Parr 2006] Parr, T. 2006. ANTLR: ANother Tool for Language Recognition *(software)*. http://www.antlr.org/.

[Parry 1997] Parry, S. 1997. The Generation and Use of Parameterized Terrain in Land Combat Simulation. In *Proceedings of Winter Simulation Conference '77*. 422–431.

[Pelton & Atkinson 2003] Pelton, B. and Atkinson, D. 2003. Flexible Generation and Lightweight View-Dependent Rendering of Terrain. *School of Engineering Technical Report COEN-2003-01-22*. Santa Clara University, Department of Computer Engineering. Santa Clara, CA.

[Perlin 1996] Perlin, K. An Image Synthesizer. In *Proceedings of SIGGRAPH '85*. ACM Press. New York, NY. 287–296.

[Planetside 2006] Planetside Software. 2006. Terragen *(software)*. Planetside Software. Ellesmere Port, Cheshire, UK. http://www.planetside.co.uk/terragen/.

[Prusinkiewicz & Hammel 1993] Prusinkiewicz, P. and Hammel, M. 1993. A Fractal Model of Mountains with Rivers. In *Proceedings of Graphics Interface '93*. 174–180.

[Qlinks 2006] Qlinks Media Group. 2006. Geo Community Website. Qlinks Media Group. Niceville, FL. http://www.geocomm.com/.

[Sanchez-Crespo 2002] Sanchez-Crespo, D. 2002. Science Imitates Nature at GDC. In *Gamasutra.com*. CMP Media LLC. San Francisco, CA. http://www.gamasutra.com/gdc2002/features/nature/nature_01.htm.

[Schmidt 2006] Schmidt, S. 2006. World Machine *(software)*. http://www.world-machine.com/.

[Torpy 2006] Torpy, A. 2006. L3DT *(software)*. Bundysoft. http://www.bundysoft.com/L3DT/.

[Ulrich 2002] Ulrich, T. 2002. Rendering Massive Terrains Using Chunked Level of Detail Control. In *Proceedings of SIGGRAPH 2001*. http://cvs.sourceforge.net/viewcvs.py/*checkout*/tu-testbed/tu-testbed/docs/sig-notes.pdf?rev=HEAD.

[USGS 2003] USGS. 2003. The SDTS Document. United States Geological Survey. Reston, VA. http://rockyweb.cr.usgs.gov/nmpstds/demstds.html.

[USGS 2003] USGS. 2003. National Mapping Program Standards. United States Geological Survey. Reston, VA. http://rockyweb.cr.usgs.gov/nmpstds/demstds.html.

[USGS 2004] USGS. 2004. USGS Seamless Data Distribution. United States Geological Survey. Reston, VA. http://gisdata.usgs.net/Website/Seamless/viewer.php.

[USGS 2006] USGS. 2006. USGS Website. United States Geological Survey. Reston, VA. http://www.usgs.gov/.

[von Werner 1996] von Werner, M. 1996. Erosion 3D *(software)*. Institut für Geographische Wissenschaften. Berlin, Germany. http://www.geog.fu-berlin.de/~erosion/.

[VTP 2006] VTP. 2006. The Virtual Terrain Project *(software)*. VTP. http://www.vterrain.org/.

[Wavefront 1995] Wavefront Technologies. 1995. The OBJ File Format Specification. http://www.martinreddy.net/gfx/3d/OBJ.spec.

[Weisstein 2004] Weisstein, E. 2004. Sample Mean. In *MathWorld—A Wolfram Web Resource*. Wolfram Research, Inc. Champaign, IL. http://mathworld.wolfram.com/SampleMean.html.

[Weisstein 2003] Weisstein, E. 2003. Standard Deviation. In *MathWorld—A Wolfram Web Resource*. Wolfram Research, Inc. Champaign, IL. http://mathworld.wolfram.com/StandardDeviation.html.

[Weisstein 2005] Weisstein, E. 2005. Skewness. In *MathWorld—A Wolfram Web Resource*. Wolfram Research, Inc. Champaign, IL. http://mathworld.wolfram.com/Skewness.html.

[Weisstein 2004] Weisstein, E. 2004. Kurtosis. In *MathWorld—A Wolfram Web Resource*. Wolfram Research, Inc. Champaign, IL. http://mathworld.wolfram.com/Kurtosis.html.

[Weisstein 2002] Weisstein, E. 2002. k-Statistic. In *MathWorld—A Wolfram Web Resource*. Wolfram Research, Inc. Champaign, IL. http://mathworld.wolfram.com/k-Statistic.html.

[Weisstein 2003] Weisstein, E. 2003. Sample Central Moment. In *MathWorld—A Wolfram Web Resource*. Wolfram Research, Inc. Champaign, IL. http://mathworld.wolfram.com/SampleCentralMoment.html.

[Woodhouse 2003] Woodhouse, F. 2003. Terrain Generation Using Fluid Simulation. In *GameDev.net*. GameDev.net. http://www.gamedev.net/reference/articles/article2001.asp.

[Wyckoff 1999] Wyckoff, R. 1999. Postmortem: DreamWorks Interactive's Trespasser. In *Gamasutra.com*. CMP Media LLC. San Francisco, CA. http://www.gamasutra.com/features/19990514/trespasser_01.htm.

VITA

Contact Information

Ryan L. Saunders may be reached by mail at PO Box 753, Bellevue, WA 98009, or by email at saunders@aggienetwork.com.

Education

I received a Bachelor of Science in Computer Engineering from Texas A&M University in 2002. I continued on to pursue my Master of Science also at Texas A&M, graduating in December of 2006.

Professional Experience

My professional experience includes:

- Hewlett Packard (Richardson, TX) Intern—FORTRAN compiler team
- Dynetics, Inc. (Huntsville, AL), Intern—Industrial automation division
- Self-employed (College Station, TX), Independent Contractor—Dynamic website development
- Microsoft Corporation (Redmond, WA), Software Development Engineer—Microsoft Office (current)

Publications

In the course of my Master's degree studies, I was co-author of a paper, *Terrain Generation Using Genetic Algorithms*, which was accepted to the *Genetic and Evolutionary Computation Conference (GECCO) 2005*.