Abstract of "Modeling for the Plausible Emulation of Large Worlds" by Steven C. Dollins, Ph.D., Brown University, May 2002.

As interactive graphics applications such as driving- and flight-simulators and especially video games, advance towards ever more complex synthetic environments, numerous challenges arise in the modeling, storing, loading, simulating, animating and rendering of the large quantity of data involved. These challenges impose strict limitations upon the designers of typical interactive graphics applications, who, to maintain interactive display rates, must simplify the data that the computer processes in each frame.

The goal of this dissertation is to enable the authoring and real-time emulation of highly interactive, large-scale synthetic environments. In order to support very large worlds, we want to present the user with an approximation of both the geometry and behaviors of these worlds, computing only enough detail to emulate a plausible experience based on the limits of the user's time-varying perception, knowledge, and expectation of the environment. Our approaches for achieving this goal are on-the-fly semantic procedural modeling of a multi-resolution description for geometry and behaviors, an event-driven runtime for managing and displaying these at interactive rates, and a cache mechanism which organizes the geometry for efficient rendering on current graphics hardware.

Both as a modeling aid and as a means of data compression, we describe the world procedurally using stochastic subdivision techniques, generating geometric and behavioral detail only as needed by the application. Authored content, in a quadtree structure akin to a scene graph, provides the parameters that guide the procedural generation of the world. We present two new inter-related data structures, the quadtree cache and the quadtree dual, which manage the nested levels of detail of the procedurally generated terrain and object data. The novel interleaved update mechanism of the cache allows us to batch geometry into time-coherent pieces that are well-suited for real-time rendering.

Modeling for the Plausible Emulation of Large Worlds

by
Steven C. Dollins

B. S., Mathematics and Computer Science, University of Illinois, 1992
M. Sc., Computer Science, Brown University, 1999

A dissertation submitted in partial fulfillment of the
requirements for the Degree of Doctor of Philosophy
in the Department of Computer Science at Brown University

Providence, Rhode Island
May 2002

This dissertation by Steven C. Dollins is accepted in its present form by
the Department of Computer Science as satisfying the dissertation requirement
for the degree of Doctor of Philosophy.

Date ______________ ____
John F. Hughes, Director

Recommended to the Graduate Council

Date ______________ ____
Andries van Dam, Reader

Date ______________ ____
David Laidlaw, Reader

Approved by the Graduate Council

Date ______________ ____
Peder J. Estrup
Dean of the Graduate School and Research

Vita

Vitals

Steven C. Dollins was born on April 9, 1970 in Pittsburgh, PA. He attended Mount Lebanon High School. At the University of Illinois at Urbana-Champaign, he was the chair of the student ACM SIGGRAPH chapter for two years, leading the development of *Netspace*, one of the first networked, 3-D graphics based games for the PC.

Education

Ph.D. in Computer Science, May 2002.
Brown University, Providence, RI.

M.Sc. in Computer Science, May 1999.
Brown University, Providence, RI.

B.A. in Mathematics and Computer Science, May 1992.
University of Illinois at Urbana-Champaign, Champaign, IL.

Honors

Andries van Dam Fellowship, Brown University, 1998-99.

Graduate Research Fellowship, Brown University, 1992-93.

Acknowledgements

Thanks to Matthias Wloka for the initial ideas on adaptive, real-time computing. To Jeff White and Dan Goldstein for hours of discussion about event management and multi-resolution behaviors. To Jeff White, Dom Bhuphaibool, Andy Reiff for their collaboration in the exploration and implementation of multi-resolution behaviors. To Robert Zeleznik, Robert Duvall and Jeff Beall for early discussions on the design of languages for interactive systems. To Tim Rowley and Morgan McGuire for help and suggestions in programming OpenGL. To my advisors, John Hughes and Andy van Dam, for all their years of support. And thanks especially to my parents who taught me to appreciate the beauty and power of mathematics.

This work is supported in part by grants from SUN, NASA (grant ID# NAG 2-830), the NSF Science and Technology Center for Computer Graphics and Scientific Visualization (grant EIA 8920219), University and Graduate School fellowships from Brown University, as well as by the contributors to the Andries van Dam academic fellowship.

Contents

List of Figures

Chapter 1

Issues in Creating and Interacting with Large Worlds

1.1 General issues for large worlds

With the steady increase in computing and especially rendering power comes the ability for interactive applications to provide users with ever richer experiences. Simulators such as for driving [32], flight, war [38], even psychological phobia therapy [57] make use of virtual environments in which the goal is to emulate at least certain aspects of the real world. A variety of computer games, likewise, emulate the real world, though often taking more artistic liberties than do the simulators. Either type of application can enhance the user experience through greater geometric and animation complexity. Even with increases in computing power, though, authoring, managing, simulating, animating, and visualizing large worlds presents a variety of challenges.

1.1.1 Vast quantities of data

Many single player games make use of large worlds. Often, primarily for the purpose of data management, these worlds are broken into *levels*, each of which may have its own visual style composed of separate sets of resources such as geometric models, texture maps, sounds, or even behavioral models or game rules. Levels may be further decomposed into different sections through data structures such as binary space partition (BSP) [23] trees, as are used in id Software's DOOM series, which may be additionally wrapped by a cell-portal graph [37] as is DOOM's successor, the Quake series. Such decompositions work well for indoor environments such as those in the id Software games or for outdoor environments where there is a high degree of occlusion, for example, the dense forest and tight canyons in Bungie's Halo. The BSP-tree and cell-portal graph make use of occlusion to cull all but a small subset of the geometry in the level.

In more open outdoor environments, however, such as are typical of flight simulators, large scale role playing games, and outdoor sports games, the user is able to see potentially miles into the distance with at best ambiguous and frequently changing occlusion as the view moves throughout the world. The underlying shape of the terrain is most often represented as a polygonal mesh height field. In some cases, the height values may be chosen to lie most closely along specific surface details and stored in what is called a triangulated irregular network (TIN), but most often they are stored in a regular square grid. Height data can come from real geological data sources such as that collected by the USGS which is often used in real world simulators such as flight simulators or the military's SIMnet. In many games, however, the data is created by artists either using 3D modeling tools or procedural techniques.

In addition to a height-field (which merely describes the underlying shape of the landscape) an outdoor environment feels sterile unless it is also populated with objects such as rocks, grass, plants, trees, and even bugs, birds, and other animals. Since a single field of grass could easily have millions of individual blades, it is obvious that we can only manage and render such worlds reasonably by making use of level of detail tricks. The simplest and most commonly used trick is to avoid explicit object representations altogether and simply to color the ground polygons, either by setting their vertex colors or, more commonly, by texture mapping them to match the appearance of the objects. Fields of grass are mottled green or yellowish brown while rocky hillsides are brown to gray or even white on their snowy peaks.

Whereas a texture may be adequate for ground objects in flight simulators or (more generally) to represent objects at a distance, when ground objects are closer to the viewer they will need to stand free of the ground plane and, especially in the case of animals, be able to move about. Positioning thousands of trees and potentially millions of plants is a laborious process, but one which, as with defining the shape of the land, can also be automated as described in Chapter 2.

1.1.2 Hand modeling versus automatic generation

Most of the current 3D games make extensive use of pre-generated geometry. Many earlier games, on the other hand, created the game environment just before play began. The ASCII-based game Nethack, for instance, creates each level of a dungeon, populating it with monsters, when the player enters it. War simulation games such as Empire and Civilization generate their world maps while showing the opening credits. Pre-generating the geometry gives the game artists very precise control over the content, but the model takes time to create and space to store. Procedurally generated worlds, on the other hand, can give the player a much more varied experience with the ability to explore a new world with the start of each game.

Multi-player games also often take place in large worlds. Most obvious among these are the massively multiplayer (MMP) games such as the role-playing games Ultima Online [45], EverQuest [56], and Asheron's Call [58]. Others include communal virtual environments such as ActiveWorlds [1] in which users provide much of the content. In these environments, the large scale of the landscape is a necessity in that it spreads the users out, thus avoiding the technical problems presented by having too many users in too small an area. Since the worlds are persistent, the large scale also helps to provide the users with a variety of terrains and content as they explore the world.

With a persistent game, it makes little sense to generate a new world at the start of each player's session since all the participants need to share in a consistent environment – unless, however, the generator creates the same world each time. Typically, procedural algorithms generate their strings of pseudo-random values originating from a single seed value. As long as all of the players initialize their random number generators from the same seed, and as long as they utilize the stream of random values in the same way, they can create the same worlds.

Even in pre-modeled worlds, as the quantity of detail climbs in response to the hardware's ability to handle it, artists are making greater use of procedural tools. In some sense, artists have always used particular procedural tools off-line, such as fancy brushes in their paint programs. Most 3D modeling tools support a wide variety of plug-ins that automate the generation of fur, hair, trees, and landscapes. Likewise, as the mathematical sophistication of game engines increases, artists can, for instance, take advantage of higher order curved surfaces which are procedurally tessellated on the fly during run-time either by the game engine or even by the graphics driver or graphics processor. Such procedural tools help to magnify the artists' power while reducing both the time to create the models and the space to store them.

1.1.3 Rendering

Finely tessellated terrain can easily contain millions of sample points depending on the area covered and the resolution of the sampling. The resultant meshes are usually too large to render outright and consequently developers have explored a wide range of techniques to reduce the number of triangles in the meshes to one that adequately approximates the appearance of the land from a given viewpoint. Chapter 3 describes a number of such techniques and describes a new one using interleaved, nested levels of detail (LODs) which better adapts to current graphics hardware than do prior approaches.

1.2 Scene graphs and procedural modeling

Typical computer graphics applications contain some sort of scene description, frequently in the form of a directed acyclic graph called a *scene graph*. Internal nodes control grouping or attribute settings, such as color or position, and leaf nodes describe geometry. For each frame of an animation, a scene graph manager performs various operations on the scene graph, such as updating transformation nodes to move objects about or to add or remove objects and finally to render the scene into an image. For efficiency in rendering large scene graphs, systems may also include a variety of techniques for choosing only the necessary visible subset of the scene graph to render. Aliaga et al. [3] give an excellent summary of such techniques as applied to very large, mostly static geometry.

Procedural modeling refers to the description of a model or model element by code rather than by explicit data. This is a rather vague definition given that, at some level, any data must be interpreted by code within the graphics system in order to generate an image. In general, though, procedural modeling provides a higher level modeling element than the basic primitives of polygons and texture maps of use in current systems. Examples of procedural elements include the procedural shaders of Pixar's Renderman [14], algorithmically tessellated curved surfaces, L-system plant and ecosystem generators [16], and fractal landscapes [44].

Procedural modeling can also be applied to animation and generally refers to any animation that is calculated by the computer rather than explicitly laid out by a human. Such calculations might range anywhere from positioning aids via inverse kinematics through physics or biomechanical models through to behavioral or cognitive modeling. The distinction between what would be considered procedural and what would not is even fuzzier than it is for geometry. Keyframed animation based on strictly defined motion paths is not procedural. Yet nearly all interactive animations, such as those of game characters, have at least some procedural animation. In some sense, any code within the application that executes as a part of the frame loop can be thought of as procedural animation at some appropriate level. We would like to make the distinction, however, between procedural elements contained within the scene graph which are executed by a scene graph manager and traditional application code separate from the scene graph.

There are some key features of procedural modeling that make it of particular use for the emulation of large worlds. The first of these is what Alvy Ray Smith [55] refers to as "data amplification": a small amount of code is able to generate a large amount of scene data. This amplification works both as a modeling tool – we can easily generate larger quantities of model data automatically than an artist might have time to do – and as a data compression tool – the code is much smaller than the essentially infinite data that it can produce. A second advantage is that procedural modeling allows us to perform “lazy evaluation” in generating the data. Furthermore, whereas a traditional scene graph might need to store most of its world to some specific level of detail, we can produce both more and less detail for different parts of our world depending on how they are viewed by the user – distant hills may be generated only coarsely while the leaves on the bush nearby show fine detail.

Although many others have described procedural techniques for the generation of a variety of modeled objects, most of them are directed towards the creation of near photo-realistic environments and generate their models off-line rather than on-the-fly. One of the goals of this dissertation is to push the definition of procedural modeling to the extreme of encompassing the entire application within the scene graph, including both geometry and its driving behaviors. We will also present techniques for managing such an explicitly dynamic scene graph efficiently for both its animation and rendering.

1.3 Multi-resolution behaviors

Even after filtering down the number of objects in the world to those only in the relative vicinity of the viewer, fully animating the detailed motion of those objects can easily swamp even the most powerful of CPUs. As with representing geometry, we can filter down the types of motion made by the objects as well.

In general, most of the action in the world is out of view and even much of it that is in view is too far away to make out fine details. If a person is walking nearby, for instance, we may be able to see the subtle nuances of how the foot lands on the ground, angling over small pebbles, yet in the distance, the person may appear as little more than a slightly bobbing dot moving across the horizon. Computing the full motion for the distant hiker is obviously unnecessary. For distant or out of view behaviors, then, we can use approximations as long as we present the viewer with a plausible approximation that meets his or her expectations of the resulting motion.

On the other hand, we must also be sure to honor the appropriate constraints of the application. Games, for instance, have rules and in simplifying the animations, the runtime system must take those rules into account. We must either devote at least enough computing resources to this behavior or else replace it with a simpler one that can achieve the same apparent effect in order to meet the purpose of the application. Consequently, we would like to have a description both for animations and for general simulations that scale their details based on both perceptual and application importance.

Such descriptions will not work for all types of applications. Changing the level of detail of a simulation will necessarily change its outcome. For most games and even for most training simulations, however, the goal is more to present the user with a plausible experience than to compute an exact result. As such, we are interested in describing plausible emulations of large worlds rather than fully accurate simulations.

1.4 On-the-fly procedural generation

In sum, we would like to create large scale worlds on the fly, moving from the authoring of specific world geometry and objects towards that of parameterizable, multi-resolution procedures that can interact combinatorially to produce an abundance of novel content.

Such procedures are subject to a variety of constraints, both technical and semantic. Technical constraints include speed (they must be fast enough to evaluate during runtime) and locality (they must not depend on the results of neighboring behaviors which might not yet exist). Semantic constraints range from adhering to the rules of the game down to meeting the user's expectation of the basic laws of physics.

Since to meet these constraints the world must track the state of objects witnessed by the user, it is assumed that most behaviors will converge over time to a consistent default state. Over some appropriate time scale, the world will be able to forget and absorb state changes. If someone steps in a puddle, for instance, the ripples die out in a few seconds, whereas your name scratched on a stone might take months to wear away. In general, as is the case with all user interfaces, only explicitly author-defined significant events should be divergent or persistent.

Though its implementation is beyond the scope of this work, in the long term we would like a system which fully generally devotes computation power to behaviors that need it most as defined by

- perceptibility - visible, center-of-focus, moving, size/distance
- predictability - user expectations over time: motion frequency
- narrative/application relevance - author defined

and for which accuracy is necessary only to the extent that it satisfies the above.

1.5 Contributions

This dissertation presents a framework for on-the-fly procedural generation and runtime management of terrain and object data for large scale, natural environments. Specific contributions include:

- A description of the goals necessary for on-the-fly procedural generation of world data, especially motivating top-down semantic constraints.
- An efficient data structure, the nested LOD quad-tree cache, for dynamic tessellization of terrain optimized for current graphics hardware. The structure supports scaling of the resolution and discrete morphing between levels of detail. Updates to the cache are interleaved between the levels of the tree to amortize the cost of changes as the view moves throughout the world.
- The dual quad-tree structure for object management for the instantiation of objects, efficient proximity detection over multiple scales, and LOD management including grouping for efficient rendering.
- Discussion and examples of semantic, importance driven multi-resolution behaviors.

Chapter 2

Procedurally Modeling a World

2.1 Procedurally Modeled Terrain

By far the most common representation of terrains is the planar heightfield. Even for very large or distant views of the ground in which the curvature of the world is visible, a heightfield on the sphere is the most convenient representation. Sufficiently close to the ground, however, even on a sphere, a planar approximation with a regular rectangular coordinate system provides a compact and efficient mechanism for describing the shape of the terrain.

At its simplest, terrain can be described by a rectangular grid with a height z at each grid location (x, y). Procedural terrain-generating algorithms produce coherent noise, attempting to match the shape of real world terrain using a variety of different techniques, though only a few of these are able to run in real time on current hardware. This chapter looks at a number of algorithms and discusses their viability for on-the-fly, view-dependent generation of terrain.

2.2 Related Work

In his book *The Fractal Geometry of Nature* [39], Mandelbrot explores mathematical constructs that emulate the appearance of a variety of natural phenomena including plants, trees, coastlines, islands, terrain, rivers, and clouds. His work has inspired many others who have generalized his strict adherence to self-similar fractals to a wide range of natural terrain and object synthesis techniques.

The majority of landscape generation techniques generate their height values by summing successive layers of noise functions in a manner similar to Fourier synthesis. Each layer, or *octave*, is scaled in amplitude usually by $1/f$, where f is the frequency of the octave. The highest frequencies are cut off beyond a certain point either based on the practicality of the implementation – too high a frequency generates too much data – or based on the Nyquist limit – frequencies beyond the sampling rate of the display device only contribute noise or aliasing. When simulating a world, the low frequencies eventually stop following $1/f$ scaling and damp towards zero. There is, for instance, a highest peak and a deepest trench on Earth, so the low frequencies obviously do not grow unbounded. The low frequencies do still contribute, however, as suggested by the rise and fall of the continents and oceans, just with an amplitude bounded by gravity or the eroding effects of weather.

2.2.1 Subdivision

Subdivision surfaces were one of the more popular early techniques used to synthesize terrain. In *The Science of Fractal Images* [49, chapter 2], Saupe describes a number of techniques for generating 1-D and 2-D fractional Brownian motion, many of which were introduced by Fournier et al [22]. One of the fastest and simplest to implement of the methods described is midpoint displacement, a type of subdivision. Starting with a regular grid of random values, midpoint displacement generates a coherent noise signal by interpolating between neighboring grid points and then adding successively smaller random values to generate intermediate points. Such techniques were further developed by Lewis as generalized stochastic subdivision [33].

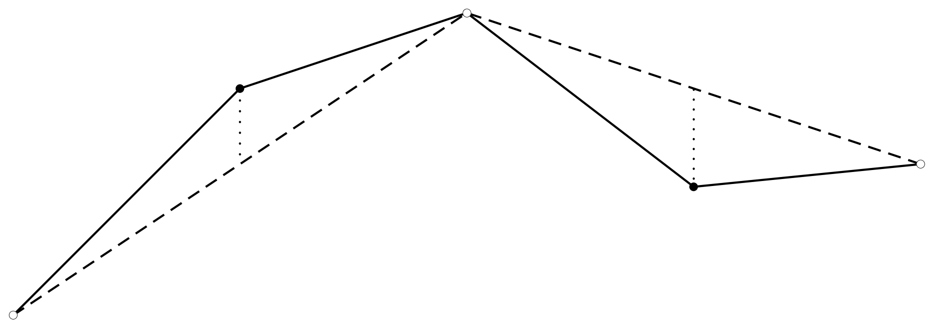

Figure 2.1: Stationary stochastic subdivision. The solid dots are computed by interpolating between the hollow dots and then adding a random displacement. The original (hollow) vertices are included in the next subdivision level.

There are two distinct types of subdivision – stationary and non-stationary. Stationary subdivision, shown in figure 2.1, preserves the vertices and even some or all of the edges from one level to the next. Non-stationary subdivision, shown in figure 2.2, replaces the vertices (and edges) with new ones at the next level.

Mandelbrot argues [49, appendix A] that stationary subdivision can leave visible creases in the landscape due to the sharing of the edges from one subdivision level to the next. He, instead,

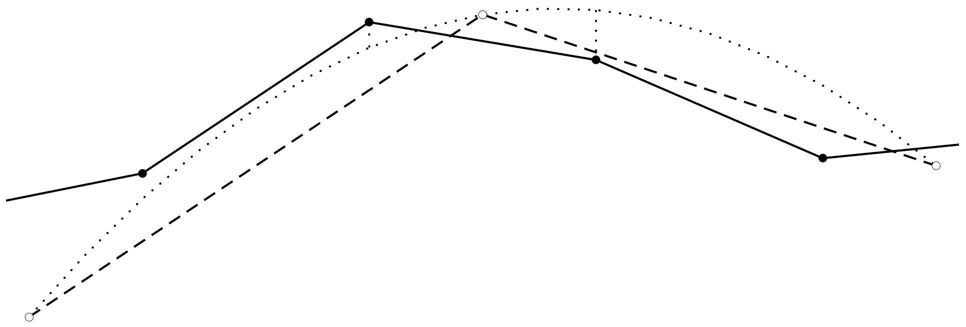

Figure 2.2: Non-stationary stochastic subdivision. The solid dots on either side of a given hollow dot are computed by quadratically interpolating between three hollow dots and then adding a random displacement. The original (hollow) vertices are not included the next subdivision level.

presents a number of non-stationary subdivision schemes based on triangles and hexagons designed to break up the edges between levels.

Subdivision has been generalized well beyond the simple grid to represent a wide variety of objects. In particular, Zorin [62] describes a generalization of the stochastic non-stationary or *dual* subdivision scheme that we will describe below.

2.2.2 Coherent noise

Perlin [51] suggests another technique for generating irregular surfaces through what has come to be known as *Perlin noise*. His technique is quite similar to that of midpoint subdivision in that random values at successively higher frequencies are summed together, except that in Perlin noise, the octaves are interpolated independently before they are summed. Peachey [18] gives an excellent summary of Perlin's and similar noise functions.

Each individual octave of Perlin noise is computed from a regular square lattice of random gradient vectors. Values of the noise function are zero at the lattice points and then computed by interpolating a cubic through the gradients for positions between lattice points. Since the function can have either zero or one zero-crossings between lattice points, the dominant wavelength of a given noise octave varies between one and one half of its lattice spacing.

To avoid visible artifacts of zero values at regular intervals, Perlin suggests sampling each noise octave as a slice through a one higher dimensional space. Thus, a 2-D noise texture map is computed from a 3-D lattice. Worley [18, chapter 3] further hides the regularities of the lattice by rotating each octave arbitrarily.

Computing the noise value for a point in 3-space entails an interpolation between eight gradient vectors from each octave's lattice. Perlin optimizes the generation of those vectors by hashing their coordinates into a precomputed table of random vectors. As a further simplification, he has suggested using simplicies instead of cubes – using, for instance, tetrahedra in 3-space – to reduce the number of lattice samples needed to just $n + 1$ instead of 2^n.

In addition to simply summing the successive octaves of the noise functions together, Perlin played with applying a number of different functions to the individual octaves before summing. Taking either the absolute value or the square of each octave, for instance, leads to a surface strewn through with sharp ridges.

In his Ph.D. thesis [43], Ken Musgrave uses a number of variations on Perlin noise to generate a number of types of natural terrain. One technique is to combine multiple noise functions, using one to parameterize another to vary the type of terrain throughout the overall landscape. Simply multiplying one noise function by another, for instance, generates a terrain which will be flat in regions where one of them is zero and rugged where they both are not. Multiplying the octaves individually allows for the smoothness to apply at different scales so that, for instance, there could be a smooth region on the side of a large mountain. Musgrave's work has been commercialized through Corel's Bryce [15] and Pandromeda's Mojoworld [47].

2.3 Discrete landscape features and objects

Although a continuous heightfield provides an efficient model approximating the shape of the terrain at a distance, it fails to capture the detail of objects upon the landscape up close. A heightfield cannot represent overhangs, nor, even more specifically, can it handle the intricacy of leaves hanging from a tree. Specific natural, organic, or man-made features require discrete representation in the form of explicit objects stored and rendered independently of the terrain heightfield.

2.3.1 Natural features

Many early games defined their world as a grid, assigning to each cell one of a number of fixed tiles describing the attributes for that cell. Terrain types were defined discretely, such as forest, desert, swamp, water, etc., each with differing values affecting, for instance, a player's speed or the types of objects or resource to be found in the cell.

Amburn, Grant, and Whitted [4] use interacting subdivision levels to merge objects such as roads or rivers with the terrain. At coarse levels, the terrain subdivision algorithm holds sway so that the road winds between the hills. At finer levels, the road's subdivision gradually masks that of the terrain making the ground under the road flat.

Prusinkiewicz and Hammel [52] models river systems using a system equivalent to Amburn et al.'s by choosing ground tiles matching the entry and exit points of a river as the ground subdivides. The river must maintain the constraint that it never flows uphill which is met simply by making it perfectly flat.

Musgrave [44] generates soil erosion runoff through a direct simulation of soil transfer over time on a grid of procedurally generated terrain. Though this method produces quite realistic models of terrain, it requires a global simulation which neither runs in real-time, nor allows for random access to the terrain.

Perbet and Cani's animating prairies [50] represent the ground with quad-trees and chooses the level of detail of animated grass based on distance to the viewer during run-time. The most distant ground is shown as a static texture. Closer to the viewer, the texture is animated to show wind motion. Closer still, it uses animated vertical texture-mapped polygons acting as slices through the prairie grass. Directly in front of the camera it uses individual geometry for the blades of grass.

For larger scale, discrete objects, Gardner [25] uses a clustering algorithm based on a thresholded noise function to distribute trees across a terrain.

Prusinkiewicz [53] models the growth of individual plants, including realistic simulation of chemical signals for branching and budding, using grammars called L-systems.

Deussen, Prusinkiewicz, et al. [16] model realistic ecosystems through analysis of geomorphology, moisture accumulation, and shadows to distribute plants of different types. Geomorphology (the shape of the land) determines the accumulation of moisture. Trees and large plants are placed first and smaller plants are generated statistically based on their preferred environment. Hundreds to thousands of plants are rendered by a raytracer and careful memory management is used to generate and store a reasonable number of instances of each type of plant without overwhelming available system memory. The rendering takes several minutes after scene generation so this system does not run in real-time.

Hammes [27] uses a subdivision scheme of predefined ecosystem tiles to render an environment in real-time. As with Deussen et al., Hammes' system starts with the underlying geomorphology, analyzing it for shape and available light. It then chooses one from a number of available ecosystem types that match the shape of the land. During subdivision, it chooses successively smaller tiles based on the ecosystem context of the parent tile, eventually choosing and positioning plants appropriate to each tile type.

2.3.2 Man-made objects

A Pattern Language: Towns, Buildings, Construction by Alexander [2] lays out a set of architectural patterns ranging from the design of spaces within homes, to the design of the house itself, up to the layout of town centers all with the intent to maximize the livability of the spaces.

For a walkthrough of Soda Hall at Berkeley [24], Funkhauser employed a simple expert system to automatically populate the building with furniture and artwork on the office walls, including appropriate positioning of the objects.

Parish and Müller[48] describe a system for laying out the streets and buildings of realistic looking cities. Roads branch from major traffic arteries and conform to hilly terrain.

2.4 On-the-fly Terrain Synthesis

Though a number of techniques exist for managing and rendering terrain and geometry during runtime, nearly all of the procedural techniques described above have, traditionally, been used to generate their data off-line. Hammes' [27] technique, one of the exceptions, is the one that this work most closely follows in terms of object generation, though even his terrain data derives from GIS databases.

Generating the world data on the fly subjects us to a number of constraints, not the least of which is speed. Although the variety of Perlin noise approaches that people have used produce some stunningly beautiful worlds, the computation involved is still a bit too slow for current hardware.

Another constraint of on-the-fly generation is that the generator must necessarily be local in its scope. We cannot, for instance, run a global waterflow and soil erosion simulation over the landscape since most of the world, including the potential source of any rivers, will not exist until we first visit it. On the other hand, we can make use of localized simulations, providing that they are not too computationally costly.

A final constraint, which applies especially to the application content is that of the semantics of the world. If, for example, we decide that we need a road connecting two distant cities, the terrain generator had better not introduce an uncrossable lake between them.

2.4.1 Terrain subdivision

For our implementation, we use a non-stationary stochastic quad-tree subdivision to generate the shape of the terrain. As processing power grows, however, Perlin noise could be substituted in just as well.

By this subdivision algorithm, the terrain shape is defined by the midpoints of the quad-tree cells. For each level of the subdivision, we derive the new child cell midpoints as a function of nine parent layer cells centered about the child cell's parent as shown in figure 2.3. The heights for other vertices in the mesh at any given level of detail, such as those at the corners of each quad-cell, are derived by interpolation through the nearby cell midpoints. At each successively finer layer of the subdivision, the previous level's cell midpoints, such as the center solid dot in the figure, will be replaced by newly interpolated corner vertices of that finer level.

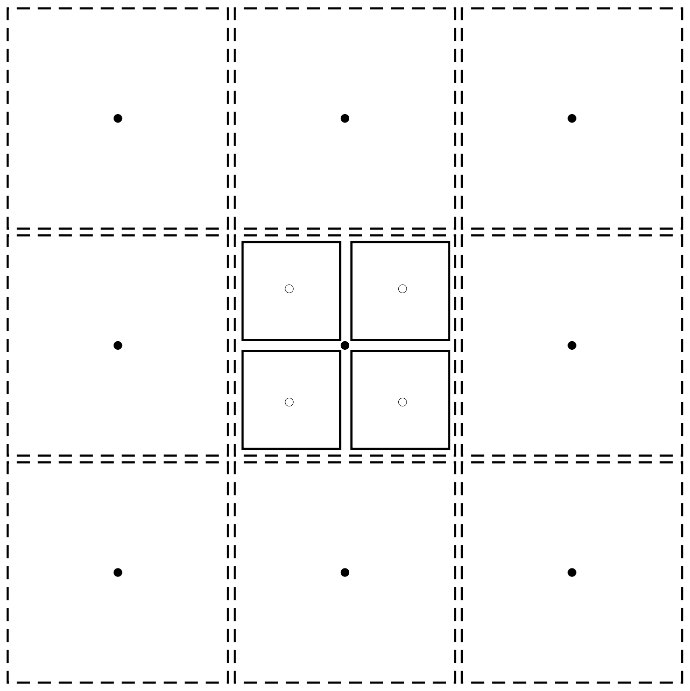

Figure 2.3: Quad-tree subdivision. The four hollow dots are computed by a bi-quadratic interpolation of the nine solid dots.

If we think of the center parent midpoint as the origin and scale to place the other parent midpoints at the integer points spanning from $(-1, -1)$ to $(1, 1)$, then the four child midpoints lie at $(\pm 1/4, \pm 1/4)$. To generate the four child cell midpoints (the hollow dots), we interpolate a bi-quadratic surface through the nine parent cell midpoints (the solid dots) and then randomly perturb their heights. Equation 2.1 gives the one dimensional quadratic function through points at $p_0 = -1$, $p_1 = 0$, and $p_2 = 1$ and equation 2.2 gives its evaluation at $-1/4$.

$$f(x) = -\frac{1}{2}p_0 x(1 - x) + p_1(1 + x)(1 - x) + \frac{1}{2}p_2(1 + x)x \tag{2.1}$$

$$f\left(-\frac{1}{4}\right) = \frac{1}{32}(5p_0 + 30p_1 - 3p_2) \tag{2.2}$$

If we apply this evaluation at $-1/4$ to itself three times, once for each of the three points, we can derive the full equation for the bi-quadratic interpolation evaluated at $(-1/4, -1/4)$ as shown in equation 2.3.

$$f\left(-\frac{1}{4}, -\frac{1}{4}\right) = \frac{1}{1024}\left(\begin{array}{rcrcr} 25p_{00} & + & 150p_{01} & - & 15p_{02} \\ +150p_{10} & + & 900p_{11} & - & 90p_{12} \\ -15p_{20} & - & 90p_{21} & + & 9p_{22} \end{array}\right) \tag{2.3}$$

By reflecting the coefficients about the *x*- and *y*-axes, we can derive the formulae for the three other child vertices as well. By storing these coefficients in an array, we can quickly compute the interpolated child values through a simple dot product with an array of the parent vertex values.

As was suggested by Musgrave, we can vary the roughness of the terrain by combining two or more noise functions. To this end, we compute a separate *roughness* noise value which scales the stochastic offset value before adding it to the interpolated heights. We can further scale the offset based on the geomorphology of the cell, smoothing out valleys and sharpening peaks, for instance.

2.4.2 Generating random values

To generate the stochastic offset used in the subdivision, each vertex needs a random number that we can derive independently from its neighbors. Typical pseudo-random number generators produce a stream of values, each derived as a function of one or more of the previous values, which are started off from some seed value. Such a generator will not suffice for our purposes, however, since we will most likely visit different parts of the world in different orders. What we need instead, is random values that derive from the two dimensional layout of our data. Perlin suggests one solution in the generation of his noise function which is to hash on the *x*- and *y*-coordinates of the vertex. In our case, since we need a different offset for each level of the subdivision, we could use a 3-D hash function, including the level as the third dimension.

As we generate the data for each cell, however, we will need more than just the one offset value. Rather, we will need a full stream of values per cell as we go on to generate all of the objects contained within that cell. Consequently, we need a random number generator that is relatively cheap to seed. Knuth [31] describes the linear congruential random number generator as one particularly efficient algorithm. It uses two large prime numbers, p_1 and p_2, to generate a new random value x_{i+1} from a previous value x_i using the wrapping properties of computer integer arithmetic as

$x_{i+1} = x_i * p_1 + p_2$. The prime values should be chosen to be near the maximum integer value for the storage being used. We can further generate multiple streams of random values from the same seed by using different prime numbers. Though quite fast to compute, one caution about the linear congruential generators is that their low order bits are not at all random. The lowest order bit, for instance, toggles between 0 and 1. Such generators are best used, therefore, by dividing them by MAX INT and treating them as a floating-point value between 0 and 1.

So, to generate our random offsets, we can derive an initial seed value for each cell by seeding a random generator with a combination of the *x*- and *y*-coordinates of the cell. From this initial seed, we can produce values as needed for the terrain generation, but we can also branch off of that stream using other prime values to produce independent streams as may be needed in the generation of objects.

2.4.3 Feature and object generation

To populate the landscape with objects, we follow the approaches used by Gardner [25], Deussen [16] and, especially, Hammes [27] in positioning objects based on the context of the geomorphology and ecosystem type of each cell of the quad-tree.

Amburn et al. [4] and Prusinkiewicz and Hammel [52] each suggest techniques for mixing other subdivision schemes into that of the terrain which we can use to produce roads and rivers. These ideas, however, have not yet been fully explored in this dissertation.

2.5 Textures and repeating noise

One of the disadvantages of producing the world data on the fly is that, on current hardware at least, we cannot yet afford to produce as much geometry as we need to color every pixel on the screen independently. We rely, instead, on triangle rendering interpolation to color the pixels between the vertices. When we render large triangles in the distance, therefore, if we set their colors based, for instance, on their altitude, the color interpolator might blend a triangle’s color between a vertex set to white in the snowy peaks and a vertex set to yellow on a sandy beach. The interpolation would then skip any colors we might prefer to have between them, such as gray for exposed rock beneath the snow line and brown and green for earth, trees and plants. The solution is to color the ground using texture maps.

2.5.1 Feature-based texture mapping

In pre-generated terrains, such texture maps are often either drawn by an artist or otherwise generated and then stored uniquely for the entire world. For large worlds, the map may be broken into pieces and then managed along with the terrain geometry. For our landscapes, which we are generating on the fly, a single global texture map will not suffice both because a single map cannot handle the scale we would like to cover and because we will not have generated the details necessary until we approach a given region. One possibility is to use ecosystem derived texture map tiles as suggested by Hammes [27]. Another is to map a texture not by the *x*- and *y*-coordinates of the terrain, but instead based, for instance, on the height and tilt of the land using the *z*-coordinate of the vertex and *z*-component of the surface normal. The interpolation of the triangle scan conversion will then produce all the colors for the altitudes between the vertices as we would expect. Figure 2.4 shows two images of the color variation as it varies with the tilt of the land.

2.5.2 Repeating noise texture generation

To provide the appearance of fine detail, such as the light and shadows of blades of grass or rough rock, it is typical to overlay the terrain with a repeating fine detail texture map at a scale much less than the one used to color the terrain. If the period of repetition is too small, however, we may be distracted by being able to see the repetition. If we can afford the extra textures in our hardware, however, we can extend the period by combining two or more repeating textures to create a non-repeating texture. Consider that if we blend together two textures, one scaled to a width of five and another to a width of six, they will only repeat at the least common multiple of their widths, in this case thirty. For 2-D textures, however, we can remove even that repetition if we rotate the textures arbitrarily with respect to one another. We can use a similar technique to produce cloud textures for the sky, even bringing them to life by translating the textures independently over time. Through a further combination of clamping and blending the texture colors we can animate the formation and dissipation of the clouds. Figure 2.5 shows the same repeating noise texture applied twice at two different orientations and scales to create a non-repeating cloudy sky.

Since we will not be generating these repeating textures on the fly, we can generate them at startup time and use a more sophisticated noise generator such as Perlin's. Creating a repeating texture can be tricky, however, since we must find a way to wrap the lattices of random values on which they are based. Such a wrapping is simple if we use axis-aligned grids, but is complicated when using the rotated lattices that we would prefer to hide the artifacts of the grid. For our lattice to tile, we must be able to map each of its vertices to a unique vertex within the unit square. Figure 2.6

shows a dotted lattice that has been rotated and scaled to align with the original lattice. A few of the vertices are labeled to show the repetition.

The rotated lattice can be defined by a combination rotation, scale, and translation matrix M (the lattices need not share the same origin as the ones in the picture do). Using modular arithmetic we can map a lattice vertex $\vec{v}$ from the transformed lattice to its associated vertex $\vec{v}_{\text{mod } \vec{1}}$ in the untransformed lattice's unit cell as shown in equation 2.4.

$$\vec{v}_{\text{mod } \vec{1}} = M * \left(\left(M^{-1} * \vec{v} \right) \mod \vec{1} \right) \tag{2.4}$$

In the 2-D plane, it is trivial to find a transformed orthogonal lattice that embeds in the integer lattice since any integer component vector (i, j) has an integer component perpendicular $(-j, i)$. In 3-space, however, this is not necessarily the case. Although such embeddings do exist, they can be tricky to find and do not exist even for most integer component vectors. On the other hand, since we are creating noise, we don't really need to use strictly orthogonal lattices. We just need a basis that is approximately orthogonal, making it much simpler to generate an embedding approximating a desirable scale by applying our desired scale and rotate transform to the unit axes and then picking nearby integer points to use as our basis vectors.

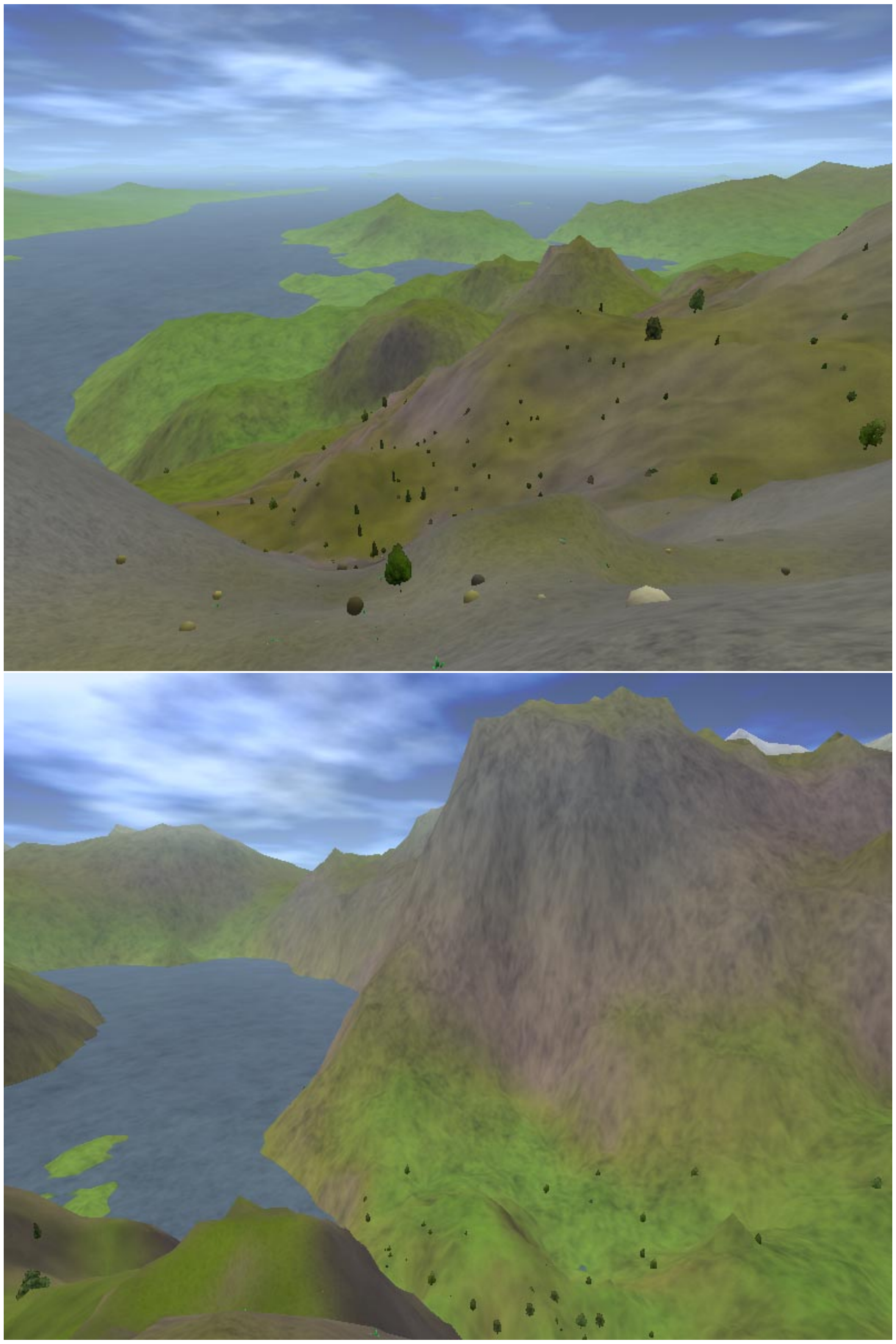

Figure 2.4: Two images of hills along the waterfront show the variety of the color map as it varies both with altitude and with the tilt of the land (indexing into the 2-D texture map using the z-coordinates of each of the vertex position and normal).

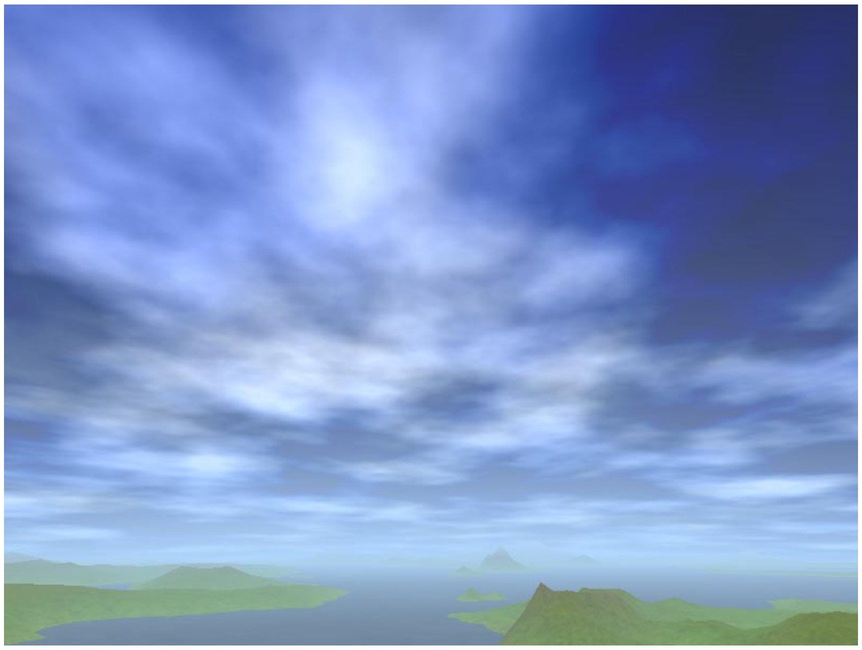

Figure 2.5: Two noise textures used to create non-repeating clouds in the sky.

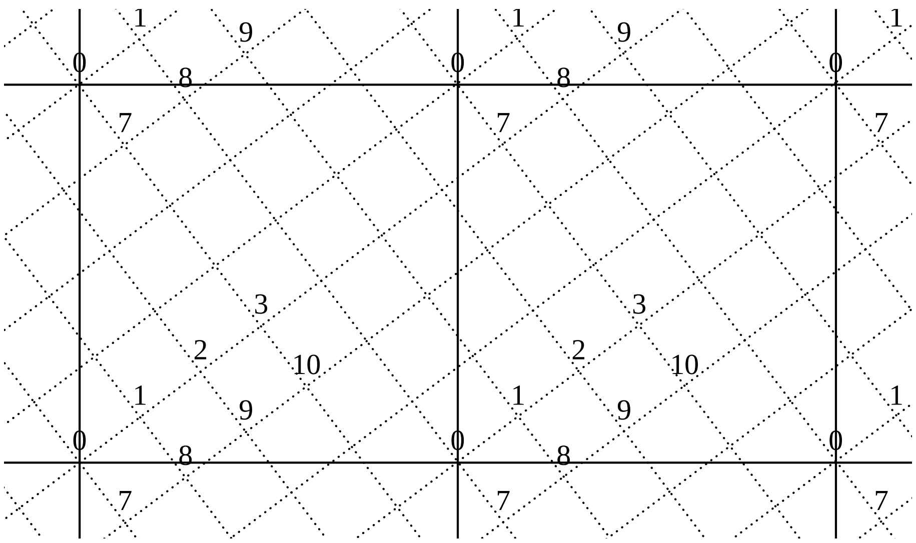

Figure 2.6: An embedded integer repeating noise lattice. The dotted lattice is embedded in the solid lattice through the basis vectors $e_0 = (3, 4)$ and $e_1 = (-4, 3)$.

Chapter 3

On-the-fly Adaptive Tessellation of Terrain

3.1 Dynamic, multi-resolution terrain meshing

Large terrains, such as those used in massively multi-player (MMP) games, might span hundreds of kilometers on a side. Geographic data of the Earth is available on the web at a variety of sites [26] [60] which provide elevation data at a resolution of ten or thirty meters between samples. Procedural generators can provide arbitrarily fine sampling resolutions. A one meter sampling of a 100km^2 patch of ground will have 10^5 samples on a side or 10^{10} sample points overall. Attempting to render a single mesh over all those points would be absurd on current graphics hardware. Nor would a brute-force rendering of all of the polygons of the mesh be particularly desirable without significant filtering to clamp the sampling frequencies to below that of the display resolution. The goal of dynamic, multi-resolution terrain meshing algorithms is to repeatedly provide an appropriate quality tessellation of the terrain based on the camera's position and orientation as it moves throughout the world.

3.2 Related work

Lindstrom et al. [34] describe the world using a quad-tree which is expanded and collapsed to maintain a uniform sampling of the world as seen through the virtual camera. A quad-tree cell is split based on its distance to the camera when the height difference between the original triangles and new vertex it introduces is large enough as projected onto the screen. As triangle splits are introduced to the mesh through expansion, the algorithm requires special processing to maintain the seam constraints between neighboring nodes. Specifically, we want neighboring cells to be within one level of detail of each other to avoid T-junctions in the mesh. When one polygon splits, that LOD change must propagate up through the tree, splitting neighboring cells as needed. To avoid sudden pops when a triangle splits, they use *vertex morphing* to animate the motion of the new vertex, based on its distance to the camera, between its position in the parent triangle and its final position.

With the ROAM algorithm [17], Duchaineau et al. introduced triangle bin-trees with an update mechanism that allows efficient maintenance of the LOD constraints during updates. ROAM refines the seam constraints between the levels of detail and makes use of a greedy algorithm to generate variable length triangle strips for efficient rendering. Figure 3.1 shows a succession of bin-tree triangle splits. Note that before the final vertex in the right-most image could be introduced, both of the triangles on either side had to be at the appropriate level of detail. We would not be able to introduce the hollow dotted vertex, for instance, before the large bottom triangle was first split as shown by the dotted line.

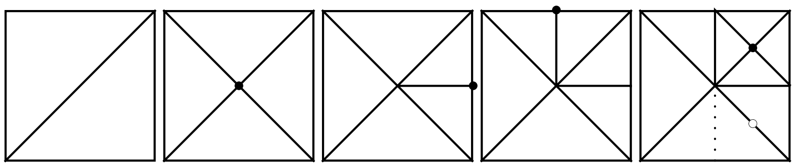

Figure 3.1: A succession of bin-tree triangle splits.

Pajarola [46] extended ROAM by using an array of quad-trees to allow for the paging in of data to support very large worlds. As with Lindstrom, however, he has to be careful to maintain continuity constraints between the different trees. He also introduces an elegant use of the Peano curve, shown in figure 3.2, to walk the trees to generate one long triangle strip, but which suffers from many degenerate triangles.

Most recently, Lindstrom and Pascucci [35] improved upon Lindstrom's earlier work with a compact data structure that allows for fast streaming of the data from disk or network with high coherency. They also introduce a nesting error metric that ensures continuity between the LOD levels by always triggering parent splits before the child splits.

Hoppe's view-dependent progressive meshes [28] allow the handling of irregular meshes. Irregular meshes are better able to conform to the shape of the land, but require a higher storage overhead and are less convenient for computing interactions between objects and the ground.

Dynamic meshing is also used for solving a variety of scientific calculations based on finite

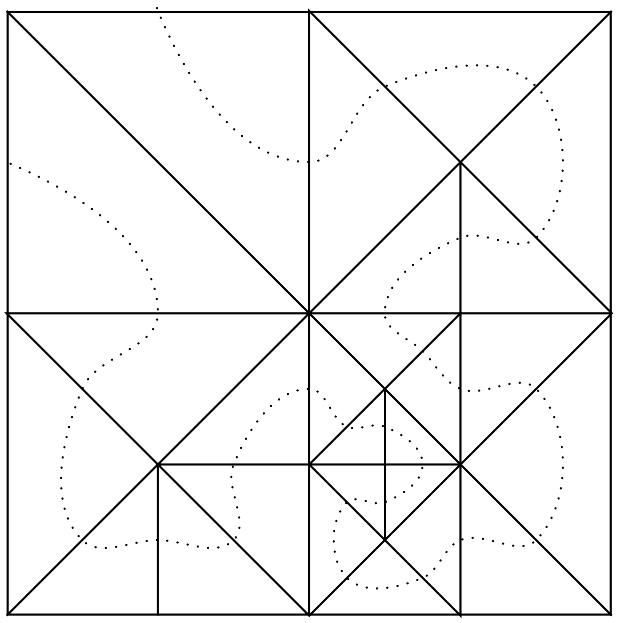

Figure 3.2: A bin-tree triangle strip using the Peano curve.

element simulation. Chatzi [10] describes an adaptive grid quite similar to the $\sqrt{2}$-subdivision scheme which we describe in section 3.6.1. Finite element meshing is used to support multi-scale computation, but does not emphasize the generation of view-dependent surface tessellations needed for terrain visualization.

3.2.1 Some analysis

Most of the grid-based dynamic meshing algorithms described above are stationary – they introduce new data between the older, lower frequency data vertices without making any change to those older points, that is, without adding to them any higher frequency information. Ideally, however, a given vertex should only represent the best approximation of the ground height for a particular sampling frequency of the terrain and not the exact height of the terrain at that point.

Consider, for instance, a low frequency vertex that lies at the bottom of a deep, but narrow, crevasse. At low frequency tessellations, before the crevasse is visible, the vertex should lie near the surface to approximate the low frequency shape of the terrain. Only when the sampling frequency is high enough to show the crevasse should the vertex position drop to that of the floor of the crevasse.

Likewise, surface attributes such as vertex normals will also change based on the sampling rate. A large, relatively flat field should have a vertex normal pointing near vertical at low frequencies, but may very well have high frequency ruts and ridges that perturb the normal for that same vertex at the fine scale.

Furthermore, consider that we are generating our data on the fly. All of these methods require a bottom up analysis of the pre-existing data to know what splits are needed by the children ahead of what is needed by the parent. We can't perform this analysis for the not yet generated data in our procedurally generated worlds.

Instead, we will introduce higher frequencies based strictly on the distance of the vertex to the viewer. Based on Nyquist sampling, we want to cut off high frequencies above a given point in order to avoid aliasing such as would be caused by sampling the vertex at the bottom of the crevasse as mentioned above. On the other hand, we would ideally like to include sufficiently high frequencies in the mesh that detail appears while it is still small on the screen.

Our trade-off is to sacrifice feature sensitivity in exchange for a finer overall tessellation in a form that maps more cleanly into the graphics hardware memory management.

3.2.2 Batching primitives to take best advantage of the hardware

ROAM and the algorithms that followed must walk their quad-trees every frame in which the camera moves, testing each edge and vertex for new split or join operations and updating the position of any vertices that are in the process of morphing between levels of detail. By choosing the walk path appropriately, the vertices can be visited in an order well suited to the immediate-mode rendering of triangle strips. Walking the entire tree every frame, chasing its numerous pointers to nodes scattered potentially throughout memory, however, is less than optimal. We would prefer instead to cache the triangle data in some compact, coherent form that is more efficient for reading by the graphics hardware and then only update that data as the levels of detail change in response to camera motion.

Graphics hardware is now sufficiently powerful that the bottleneck in rendering speed lies largely with the graphics bus. The best performance comes, therefore, from organizing the data in forms that are most optimized for transmitting it to the card. In current OpenGL, this means using indexed vertex arrays to render triangle strips. ROAM and the others have particular trouble using fixed size arrays, however, since they allow for the insertions and deletions of arbitrary vertices and triangles into the middle of strips. Furthermore, vertices that are morphing between levels of detail may change their attributes with every frame necessitating frequent re-writes of the vertex array data.

For best performance on current hardware, we would instead prefer a scheme for dynamically updating the terrain that stores the data in batches which can remain constant for long periods of time. Although we would also like to blend in changes between levels of detail, we want to avoid re-writing vertex data every frame. Instead, we will update batches of a fixed size all at once, blending LOD changes across them, and store the results in vertex arrays optimized for the graphics hardware.

3.3 Nested LOD Cache with Interleaved Updates

3.3.1 Quad-tree data storage

As in most grid-based terrain meshing algorithms, our primary data structure for storing the world data is the quad-tree. Since our procedurally generated world is arbitrarily large, however, the camera can easily move past the bounds of the root node. One possible solution is to grow the tree vertically, adding new root nodes as needed to act as the parent to store new sibling cells to our original root node. A simpler and somewhat more flexible approach is to use a 2-D hash table to store a forest of quad-trees such as is proposed by Pajarola [46]).

We can extend this 2-D grid view of the quad-trees to each of the lower levels as well, also viewing the trees as a stack of 2-D grids of cells. Each layer, from the root layer to leaves, represents a successively higher level of detail which contains finer scale objects and higher frequency information about the shape of the terrain. Within each layer, we can identify each cell by its index, (i, j), which is the coordinates of its lower left corner divided by the cell width. Where necessary, we can extend the index with a third component, k, representing the layer number, defining layer k as the one with a cell radius of 2^k.

3.3.2 View-dependent instantiation

Unlike the related work which adds and removes detail arbitrarily, during runtime, we wish to maintain a fixed number of cells around the camera at each level of detail. As shown in figure 3.3, we instantiate and store cells only from the middle of each of the parent levels of detail. Cells outside of the white regions which are far enough from the view to be too small to see have not yet been procedurally generated and therefore do no need to be stored.

A 1-D example, a binary tree, is shown in figure 3.4. Each node of the tree represents a region of terrain, halving in size with distance from the root. The node stores its geometric position computing its height as a random offset from an interpolation of its parents' positions as was described in chapter 2. The camera position in this example is represented by the vertical line which is just to the right of the root of the tree. Each layer of the tree has instantiated eight nodes, four to either side of the camera. To render the terrain, we generate a mesh through the leaves of the tree.

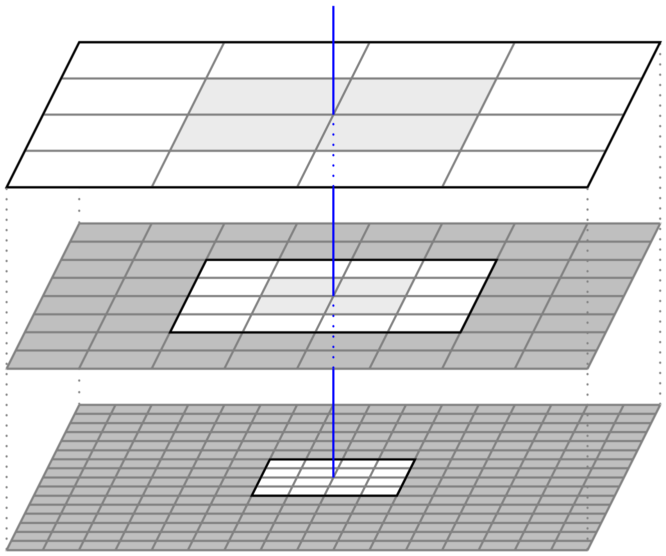
Figure 3.3: The visible cells of a quad-tree. We instantiate and store only a fixed number of cells around the current camera position, represented by the vertical blue line, at each level of detail.

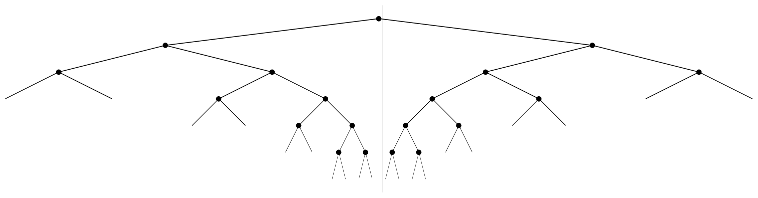
Figure 3.4: Centered binary tree. The camera position is represented by the vertical line.

As the camera moves through the scene, the runtime system instantiates new nodes in the tree ahead of the camera and removes those from behind, thus maintaining a relatively uniform frequency of visible information at every scale. Figure 3.5 shows a tree in which the finest level of the tree has shifted to the right as the camera moved.

For simplicity, parent nodes always instantiate all of their children, so a change to a layer of the tree occurs when the camera crosses the midpoint of the parent node cell. Figure 3.6 shows the level shifts as the camera crosses the next four successive cell midpoints. As can be seen in the figures, the finest layer of the tree updates with every other change to the tree. The next finest layer updates half as often as the finest and so on up the tree.

As is shown in the dashed circles, between two neighboring parent cell midpoints there are two child cell midpoints, at $1/4$ and $3/4$ of the distance between the parents, and some higher level cell midpoint at $1/2$. A child layer will be in one of three configurations with respect to its parent layer

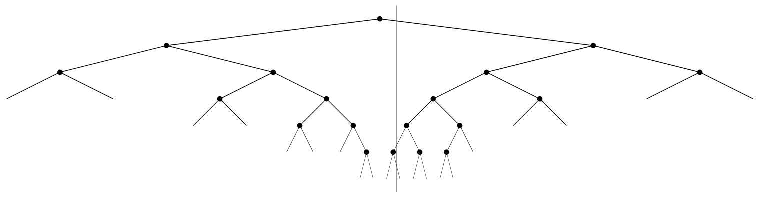

Figure 3.5: Shifted binary tree. The finest level of the tree has shifted as the camera moves to the right.

as the camera moves between the parent nodes. Between 0 and $1/4$ the child layer is to the left (as is the finest layer in the top image), between $1/4$ and $3/4$ the child is centered (middle two images), and between $3/4$ and 1, the child layer is to the right (bottom image). At $1/2$, some arbitrarily higher level shifts (as happens between the middle two images). When the parent layer shifts, the child layer switches from being centered to one side of the parent layer midpoint to being on the other side.

3.3.3 Caching the nodes

In order to instantiate new nodes in the tree using the procedural methods described in chapter 2, the runtime system splits a parent node, creating the children as a function of that node and its eight quad-cell neighbors. Ideally, we would like to be able to find these neighboring parent nodes in constant time. Using only the parent-child relationships in the tree, however, although a sibling to one side of a given node is only two pointer hops away (up to the parent and then down to the sibling), the neighboring cell to the other side may be an arbitrarily distant cousin. In particular, the greatest distance between two nodes in the tree is that between the leaves just to either side of the origin, ironically making the most common place for the viewer to be also the most expensive for finding cell neighbors.

One solution to the cost of finding the neighbors is to borrow the idea used to manage the multiple root nodes, namely to store the nodes of each layer in a 2-D hash table. Finding any arbitrary cell from the tree then becomes a constant time operation of hashing on the coordinates of that cell.

Another, even simpler solution, however, is to cache the pointers to the cells at each layer in simple 2-D arrays. Each time the runtime system shifts a layer of the tree, it simply updates the cache pointers for that layer. Although the overhead of updating the layer incurs a greater cost than that of adding just one row to the hash table, reading cells back from the 2-D array cache is much quicker than repeatedly hashing on the coordinates of the cells to read back from the hash table.

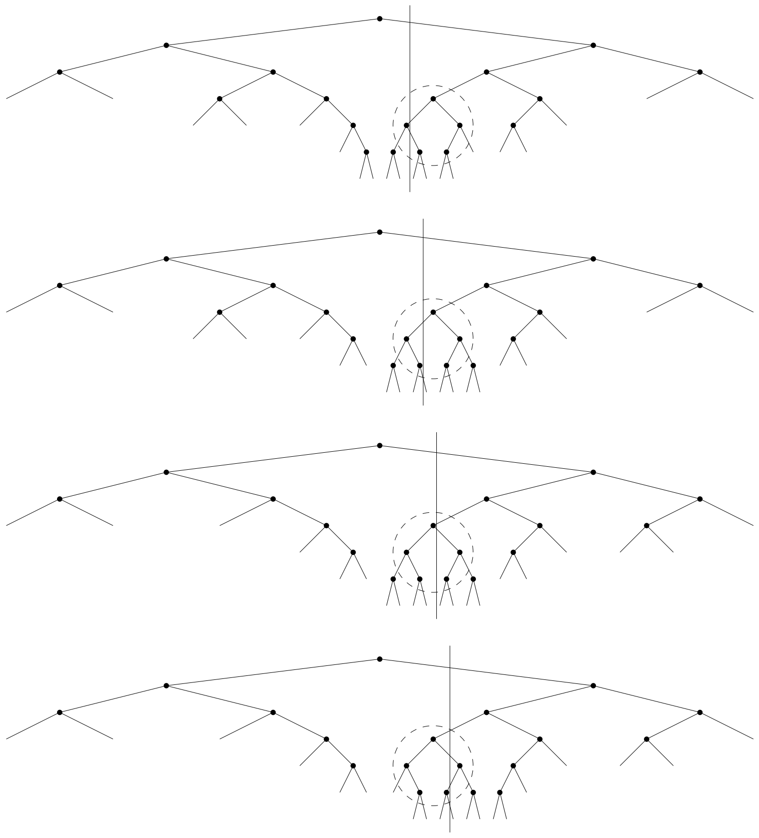

Figure 3.6: Four cache level shifts. From top to bottom, levels 2, 1, 3, and 1 shift as the camera moves from left to right (level 0 is the finest level).

3.3.4 Defining a continuous mesh

By dividing the terrain into separate levels of detail, we can tessellate each level independently of the others and then connect them with a seam of polygons stitching successive layers together. Each layer of the tessellation is composed of an *outer ring*, an *inner region*, and the *seam* between that layer and the subdivided cells which make up the *center*. Vertices for the outer and inner regions come strictly from this layer, while vertices in the seam must come from both this layer and its child. The finest layer has only a single region. Figure 3.7 shows each of these regions in a 1-D example.

We distinguish between the outer ring and the inner region of the tessellation because the inner region is forced to change configuration as the child layer moves with respect to the parent layer. Figure 3.8 shows all of the nine possible configurations for the inner region in 2-D. Note that despite the change in configuration, the inner region will always consist of the same number of cells. The outer rings and the seam always have the same shape, though, along with the interior region, their

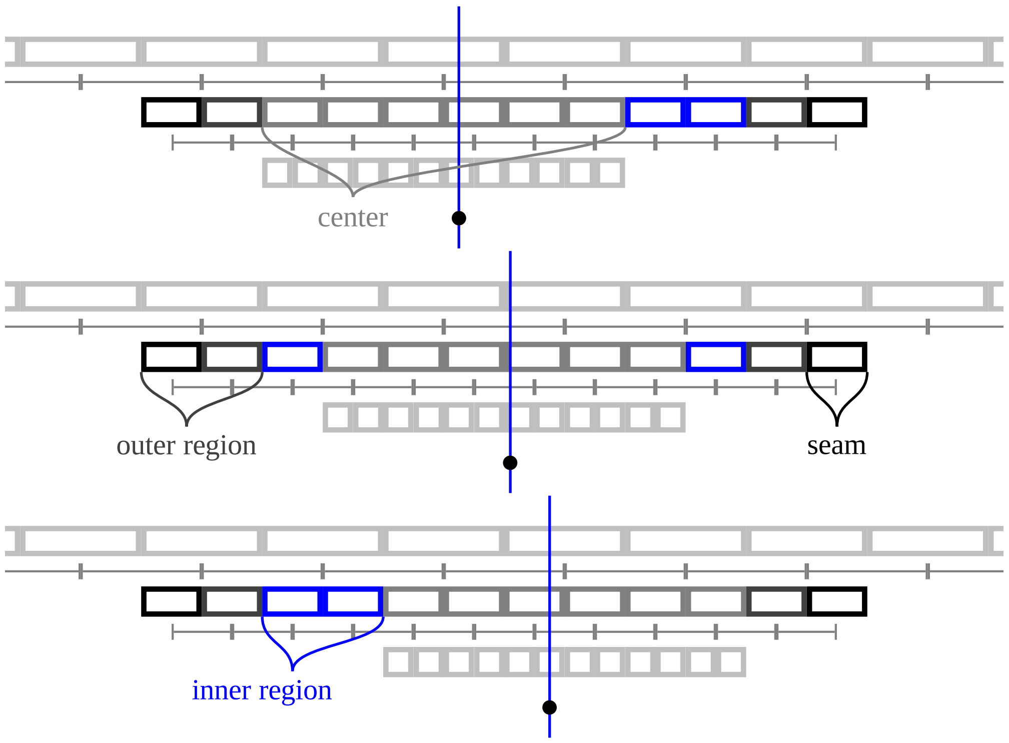

Figure 3.7: The different cache regions. The center cells, shown in gray, are those that have subdivided and have children. The outer cells of each cache layer, shown in dark, remain fixed as the child level moves. Between these two is the inner region, in blue, which changes configuration depending on the position of the child layer with respect to the parent. The outermost single cell acts as a seam between this level and its parent level.

underlying data changes as they slide across the surface of the terrain.

Figure 3.10 gives an example of the top half of a tessellation based on a cache width of sixteen cells. The heavy black lines border the outer ring, inner region, and seam respectively from the outside towards the middle. The outer dashed line marks the outer boundary of the middle of the three layers and the inner dashed line marks the inner boundary of the seam. In figure 3.9, the finest layer has shifted to the right leaving the inner region with two rows on the left and none on the right.

Remember that it is the midpoint of each cell that was computed by the procedural generation algorithms described in chapter 2. We compute the corner vertices of the cells by interpolating between the midpoints of the four surrounding cells. Since cells on the boundary of each layer have no neighbors on at least one side, we cannot compute all of their corners to tessellate them. Instead, these cells are used only to compute the corners of their neighbors towards the inside and are instead tessellated by the seam between this layer and the parent layer.

The fine dashed lines in figures 3.10 and 3.9 show merely one possible tessellation of each cell,

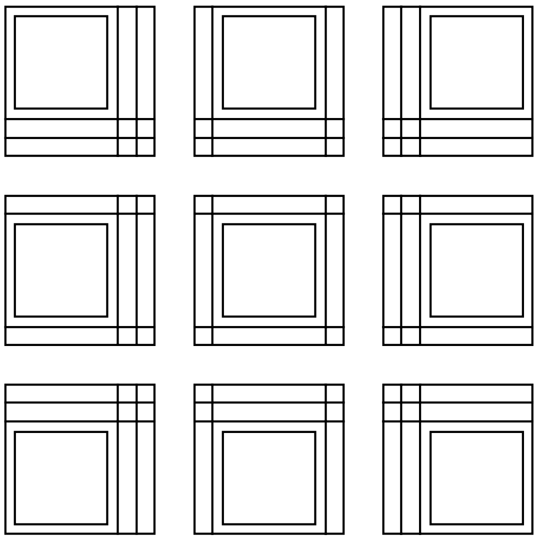

Figure 3.8: The nine possible configurations for the interior region.

in this case, using a triangle fan. Each layer of the mesh could be turned into triangle strips using a walk similar to the Peano-like curve construction of Pajarola[46]. We could also compute any number of other intermediate vertices through interpolation as we did for the cell corners which might allow for a more conventional triangle strip construction with fewer degenerate triangles. An advantage to sticking with the bin-tree tessellation, however, is that it allows us to preserve the triangle edges across the seams. Figure 3.11 shows the triangle fan tessellation.

3.3.5 Cache size

We can easily adjust the level of detail of the terrain mesh by varying the size of the cache, both in depth to change the minimum triangle size and in width to change the average triangle size. There are, however, a few constraints we must consider in choosing the width. First, we would, ideally, like the width of the cache and, consequently, the size of the vertex arrays holding the terrain data, to remain constant as the camera moves about. Second, we would like to keep the mesh approximately balanced in terms of the number of nodes on either side of the camera. Third, for simplicity both in the data structures and in the algorithms that manage them, we will always generate all the children of a node when we split it. Fourth, we would like to ensure that the necessary parent nodes always exist before they are needed to create a child node. Finally, fifth, there must be sufficient separation between the levels of detail to provide room for the seam that smoothly stitches them together.

The simplest way to meet the second constraint is to keep an equal number of subdivided nodes on either side of the camera, an even number in total, which, combined with the third constraint,

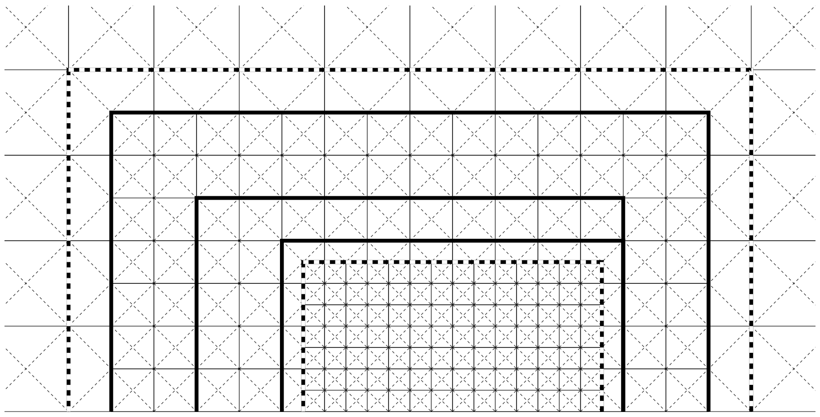

Figure 3.9: Terrain tessellation with the finest level shifted to the right.

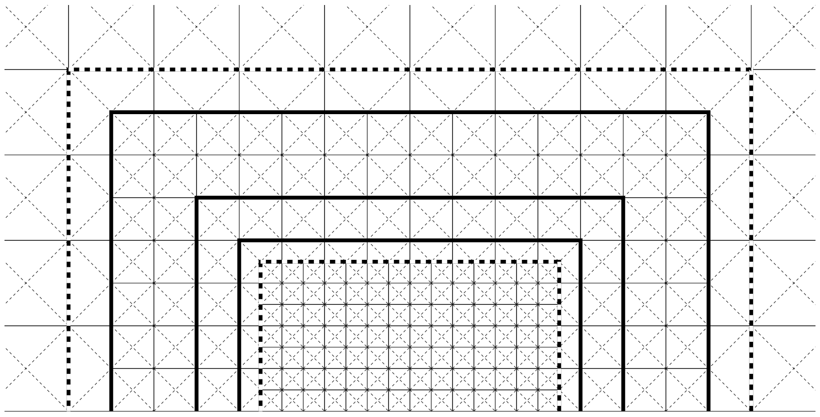

Figure 3.10: Terrain tessellation.

implies that the cache width must be a multiple of four. We can also think of this number as twice the *cache radius*. Half of the cache radius will be allocated to the child layers. Since the child level of detail must be able to move from side to side within the middle region of the parent layer, one cell of the cache radius must be allocated to the inner region. An additional cell must be allocated to the seam to meet constraint five. So, the minimum cache radius is four.

We can also think of the cache radius as twice the *inner radius*, which is the number of subdivided parent nodes to one side of the camera position. Since two nodes of the cache radius are allocated to the inner region and seam, the remaining nodes make up the outer ring, which has a width of two less than the inner radius. At the minimum cache radius of four, the inner radius is two and so the outer ring width would be zero. In general, though, we will set the inner radius much higher, such as eight or more, because we want much finer polygonization of the terrain and current graphics hardware can easily handle it.

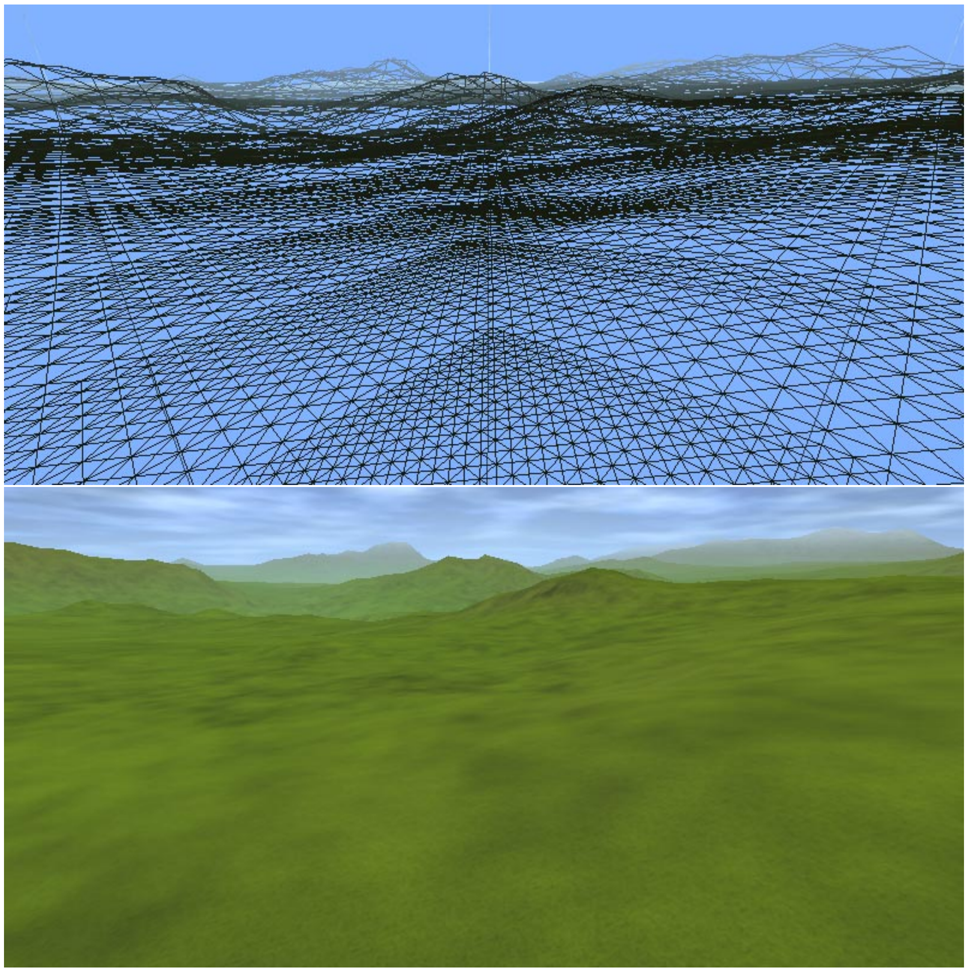

Figure 3.11: A plain leading up to hills in wireframe and in solid polygons.

3.3.6 Updating the cache as the view moves

How we choose to update the cache and vertex arrays as the view moves depends on the application. For bounded, relatively slow camera motion in which the camera will only cross one or two cell boundaries between frames, we can update the cache by incrementally removing and adding a single row from a single layer at a time. Applications that call for faster or more arbitrary camera motion will need a more general absolute positioning cache update.

For bounded, relatively slow camera motion, we can use an incremental update method. Since the cell boundaries of each level of detail intersect the midpoints of their child layers, the camera can move freely within each quadrant of a child cell without triggering any cache changes. Thus, we can detect cache changes at any level by computing the (i, j) coordinates of the camera at twice the resolution of the finest level, and then testing for a change from the previous frame. Given the assumption of relatively slow motion, we can shift the caches independently in the *x* and *y* directions at only the potential extra cost of generating one or two extra cells at the outside corner. For each step of the coordinate, we compute the level of the cache that changed by simple bit arithmetic on the coordinate, namely, the index of the lowest set bit is the layer that shifted. For positive motion, we test the bit on the prior value of the coordinate, whereas for negative motion, we test the bit on the next value of the coordinate.

After updating the cell pointers, we must update the vertex array data for the rendered mesh. When a given levels shifts, the seam with its child remains unchanged. The interior region changes from shifted to one side to the opposite configuration and is based on new vertex data. The exterior ring changes to new vertices. And finally, the parent level's seam and interior change.

For faster or arbitrary camera motion, when layers of the cache might shift multiple cells per frame or even be completely replaced by new cells, it can be more efficient to use an absolute update. Figure 3.12 shows, for instance, a cache layer which has shifted three cells to the right and two cells up.

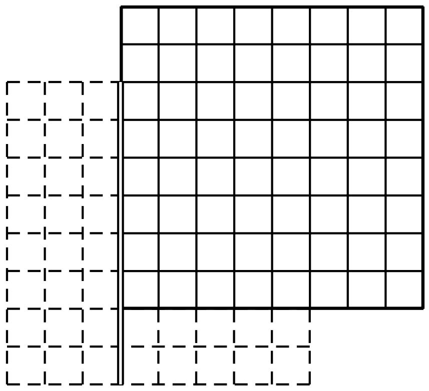

Figure 3.12: A shifted cache layer. The dashed cells in the two rectangular regions must be removed.

The absolute update algorithm consists of the following steps:

1. Compute the new cache depth based on the camera's height above the ground.
2. Remove any of the finest levels that are below the new depth.
3. For each level of cache, compute the new cache position coordinates.
4. For each level from the finest level to the coarsest, if the level changed position, remove the old cells. As shown in figure 3.12, the old cells can be grouped in two rectangles divided in the figure by the vertical double line.
5. For each level from the coarsest down to the new finest depth, if the level changed, update the cache pointers creating new cells from the parent level as needed. Also set the LOD for the cells

as defined in section 3.4.

6. Finally, for each of the rendered levels, update the vertex array data as follows:
    1. If this level changed, update the outer ring.
    2. If this level or its child changed, update the inner region.
    3. If the child changed, update the seam.

3.4 Discretized vertex morphing

Most view-dependent terrain tessellators use vertex morphing to blend in changes between the levels of detail. Each new vertex introduced to the mesh starts at the position that matches its triangles with the parent triangles they have replaced. The vertex then gradually moves into its appropriate position based on the distance to the camera. Each frame, all vertices in this transitory state have to compute their distance to the camera and update their rendering information potentially modifying data that we would prefer to have cached in the graphics system. As an alternative, the quad-tree cache provides us with a mechanism to discretize the vertex morphing, still providing a gradual introduction of triangle splits into the scene, but in a much more hardware friendly manner.

We can think of the outer ring of each cache layer as consisting of a set of square shaped rings, within each of which the cells are at approximately an equal distance from the viewer. We can set the LOD for each cell to a fractional value depending on which ring it lies in, ramping from zero and the outer ring to one at the innermost ring. Rather than morphing vertices every frame as ROAM does, we can morph them discretely using the ring LOD as the interpolating coefficient, changing the value only when the cell changes which ring it is in.

3.5 Hysteresis

A particular disadvantage of making discrete changes to the world is most evident when walking along one of the update boundaries, namely flickering. Though more costly and less hardware cache friendly, continuous LOD morphing does have the advantage that short back and forth motions will not cause the popping generated by repeatedly crossing a discrete update boundary. We can avoid such flickering, however, through the use of *hysteresis* by thickening each update boundary from a line into a region. We shift a given level of the cache only when we leave the update boundary region, but not when we enter it. Thus, when a cache layer shifts, it cannot flicker back to its previous position until the view moves all the way back across the boundary region and then leaves it from the other side.

We choose the size of the hysteresis regions proportional to the size of the cells in each layer, so one layer's update region has half the width of that of its parent. Since we want to make sure that the update regions do not overlap between the layers, at its largest extreme, a layer's update region can reach one third of the distance to the midpoint of its parent's update region with the parent's region reaching the other two thirds of the distance between them. To avoid contact between the regions, however, we can pull the region back to $1/4$ of the distance. Figure 3.13 shows the hysteresis regions around cell centers for several different cache layers.

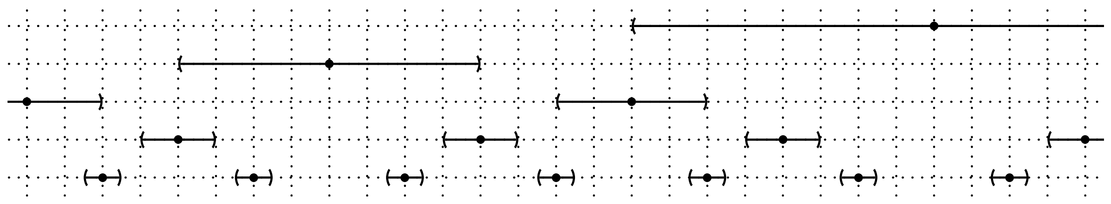

Figure 3.13: Update boundary hysteresis regions for several levels of detail.

3.6 Other subdivision schemes

Although a square grid provides a convenient subdivision framework for managing the terrain data, it is worth considering other tessellations of the plane, such as by triangles or hexagons. In doing so, however, we must keep in mind one of the key features of the quad-tree cache, namely that we can interleave the timing of the level updates as the view moves. That is, midpoints of the cells at different levels must not coincide along any of the primary axes of motion. When one level of the cache updates, we would like it to be the only level updating.

3.6.1 $\sqrt{2}$ subdivision

One particularly nice tessellation that meets our criteria is suggested by the triangle bin-trees as a simple extension to quad-trees. Rather than jumping immediately from one square cell to four with a linear scale factor of 2, we can tessellate the plane with 45-90-45 right triangles splitting each of them in half at the next level of subdivision for a linear scale factor of $\sqrt{2}$. Figure 3.14 shows the nested levels of detail for such a subdivision. Essentially, we have promoted the seam between our quad-tree levels into a formal level of detail.

One problem with this nesting, however, is that the different levels of detail touch at the corners, which leaves us without room to shift the finer levels within the coarser ones. The solution is to

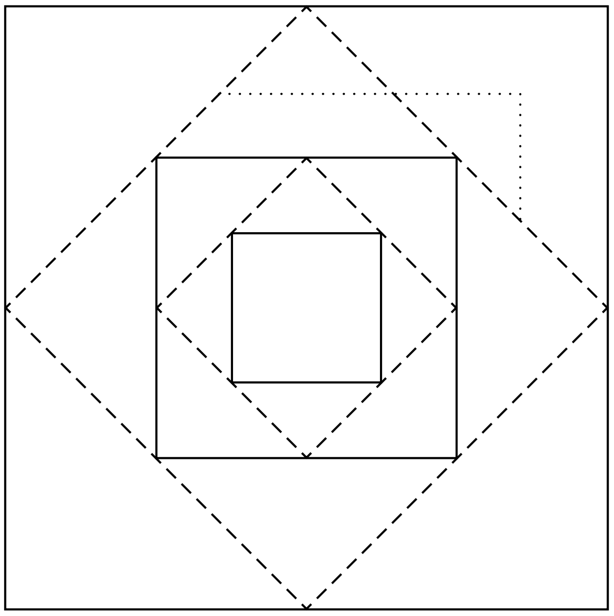

Figure 3.14: $\sqrt{2}$ subdivision hierarchy.

remove a number of cells from the corners of each level as is shown in figure 3.15. Rather than the nine relative positions a child level may be in with respect to its parent level as before, there are now only five: centered or shifted towards one of the four corners of the parent level. In the figure, the finest level is shifted towards the right and its parent level has shifted to the upper right.

As with the axis-aligned levels, updates to the rotated levels occur when the camera passes cell midpoints, or the diagonal lines connecting them as represented by the dashed lines in figure 3.14. The dashed lines in figure 3.15 represent the update bounds for each of the layers about the current camera position represented by the dot.

Figure 3.15 is merely one possible tessellation based on this scheme. Another less regular, though possibly more convenient, tessellation would use a full square cache for the axis-aligned layers and then an approximation to an axis-aligned square for the rotated layers. Consider, for instance, adding a few extra horizontal and vertical rows to the seams in the original quad-tree tessellation. In doing so, we must also add a few extra cells at the corners, so either the “width” of the cache would necessarily be a bit larger for the rotated layers than for the axis-aligned layers or the corner cells could be packed into the vacated edge cells using the memory packing scheme described by Lindstrom [35]. For the latter, consider in figure 3.14 storing the cells that would be in the triangle under the dotted lines in the upper right into the corner above the dotted line in the larger of the two rotated layers.

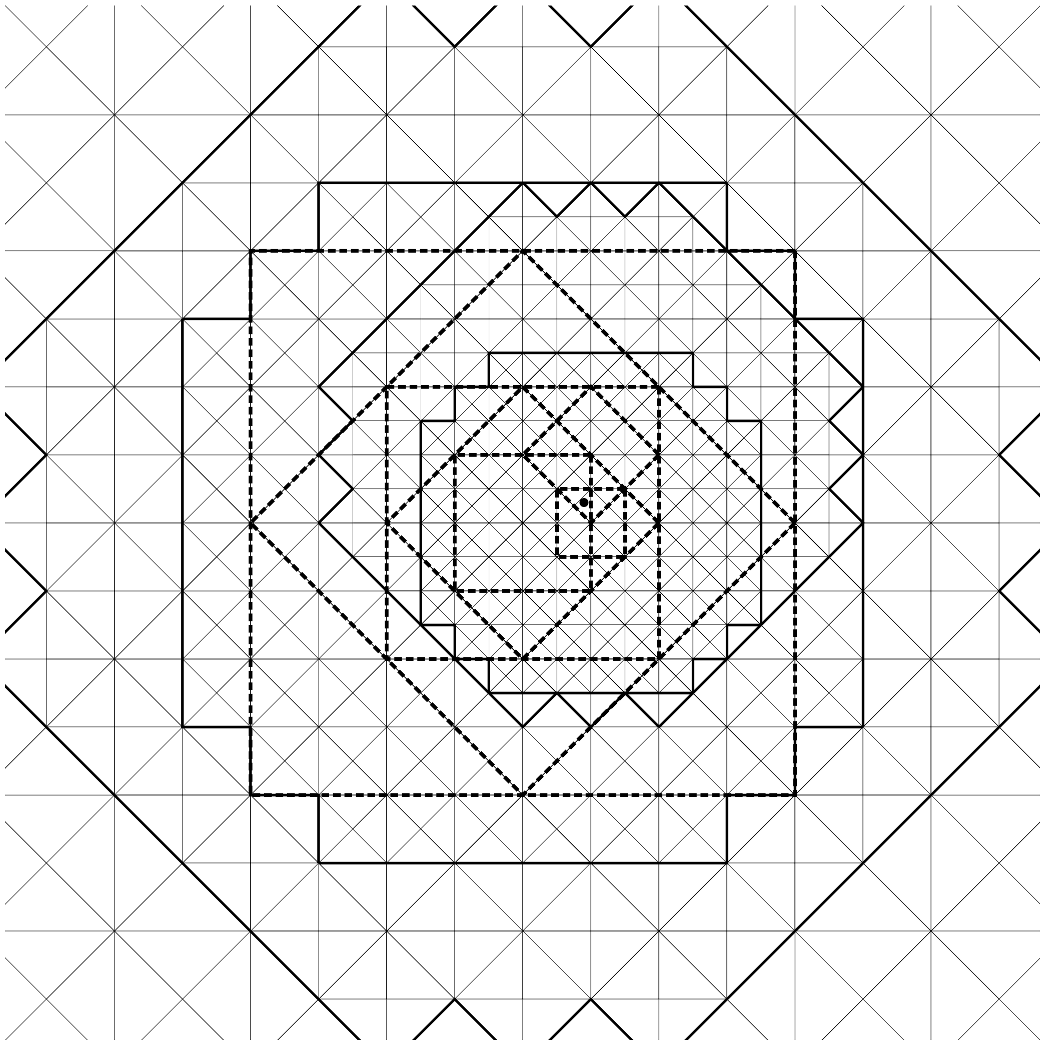

Figure 3.15: A $\sqrt{2}$ subdivision tessellation. The finest level is shifted to the right and its parent is shifted to the upper right. The camera position is represented by the black dot.

3.6.2 Triangle and hexagon subdivisions

Equilateral triangles make for a particularly attractive tessellation of the plane since they can more closely approximate a circle than the square grid. Figure 3.16 shows one possible planar tessellation using a hexagonal cache of triangles nested four to one. In this nesting, a child cache layer can be in one of seven positions with respect to its parent, either centered or shifted towards one of the six cache corners.

The dashed lines in the figure show the boundaries that trigger cache level changes for each of the three levels. Notice, however, that the upper left and upper right boundaries for the smallest

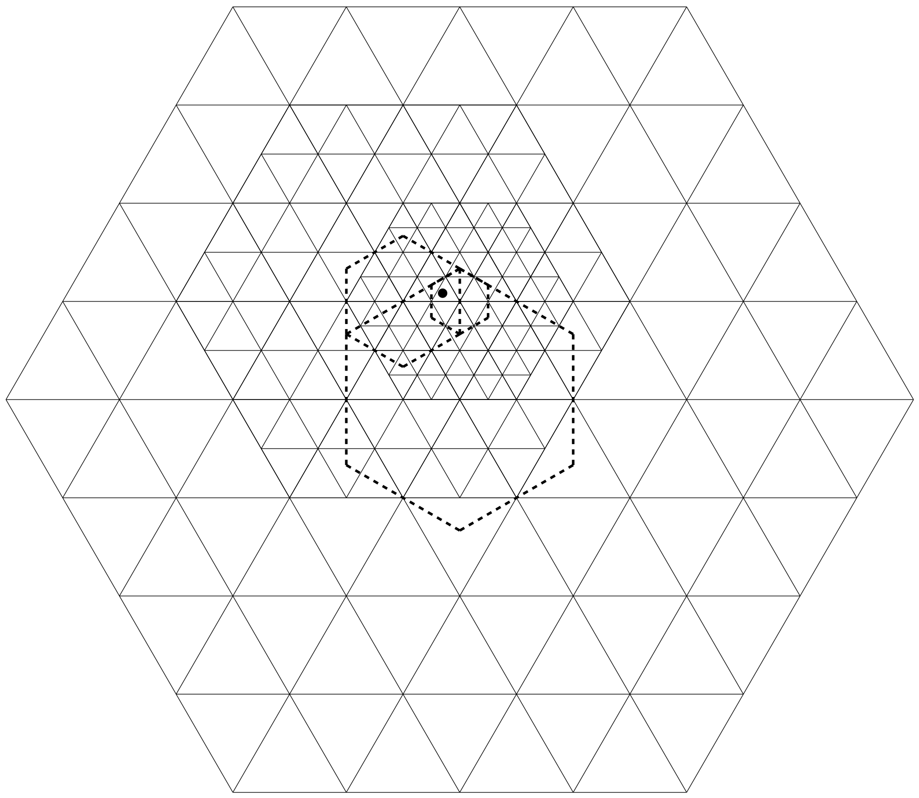

Figure 3.16: A problematic triangular subdivision of the plane.

and the largest levels coincide for this particular camera position. In fact, the corners of any of the coarser cache levels will be coincident with boundaries of every second finer cache level. Thus, crossing one of the coarsest cache boundaries near its corner can trigger a change in fully half of the total cache, leading to a simultaneous update of half of all of the world data. Consequently, this would not be a particularly desirable subdivision of the world.

3.7 Conclusions

The simple quad-tree subdivision is easy to manage and straight forward to implement with an axis-aligned square cache. $\sqrt{2}$ subdivision can be trickier, requiring a more rounded cache, but gives a more gradual transition between levels of detail. Other schemes may well be possible but care must be taken in their design to avoid coincident updates between the different levels of detail.

Chapter 4

Object Management with the Dual Quad-tree

4.1 General object management techniques

Nearly any natural world simulation, large or small, provides some form of physical simulation, including, in particular, collision detection. In its naive implementation, *n*-body collision detection has a complexity of O(n^2). Below, we discuss a number of more efficient collision detection algorithms which scale more appropriately to large worlds. Of particular interest is the ability to handle collision detection between objects of widely varying scales, such as, for instance, that between a flea and a dog. Mirtich's hierarchical hash tables directly support such scale differences. We improve upon its performance with the *dual quad-tree* structure and meld it with the procedural worlds we have described previously.

Large worlds have the additional task of instantiating and deleting data based on their position relative to the user. Dynamic objects present the additional challenge in managing their instantiation as the user encounters different areas of their domain. As we will show, the dual quad-tree presents an elegant solution to dynamic object management as well.

4.2 Related work

The the most general mechanism for solving the *n*-body collision (or proximity) detection problem is to divide space up into discrete cells. The simplest way to do this is to use a square grid to bin the objects together that are near one another. In 2-D, an object can be in up to four grid cells if it happens to overlap the cell corners and assuming that the object's size is less than the cell size. This size dependence is one of the restricting factors in using a grid. If the grid size is too large, there will be too many objects per cell. If the grid size is too small, objects will spend too much time changing cells. Furthermore, if the world is to contain objects of widely varying sizes, there may be no single grid spacing that is appropriately efficient for all objects.

I-COLLIDE [13] uses the sweep and prune algorithm which maintains lists of object extents, tracking overlaps in each dimension. When the projected extents of a pair of objects overlaps in all three dimensions, their axis-aligned bounding boxes intersect and the system tracks that pair for intersections based on exact object features. This technique can be quite expensive if objects cluster in any one dimension, such as occurs with the vertical when objects are all near the ground, though in that case the particular dimension can be removed. Another drawback is that the data structures are not particularly localized. The changes made to one object can affect other objects scattered throughout the world along each of the axes.

Hubbard [30] describes four dimensional frusta based on knowledge of potential object motion over time to bound the soonest time that two objects could possibly interact. By knowing the maximum acceleration an object may be subject to, we can bound the space throughout which an object can move within a given amount of time. This space forms a four dimensional horn, which, if cut off at a given time into the future, may be bound by a frustum which is easily tested for intersection with other frusta. Any collision detection between two objects can then be deferred until at least the time represented by the intersection of the frusta.

Mirtich [41] extends Hubbard's deferred tests by combining it with a multi-resolution spatial hashing. Objects are stored in layers of sparse 2-D arrays, implemented by hash tables, based on their size and in all higher layers. A cell of an array is allocated only as needed when an object enters the cell's bounds. The cells are stored in a hash table hashed on the (i, j) coordinates of each cell. As with the quad-trees described in Chapter 3, the cells nest perfectly from one scale to the next with four child cells in each parent cell in the 2-D case or eight cubic oct-tree cells per parent cell in 3-D.

Figure 4.1 shows an example of four different objects in the hash tables. Objects are stored in each level with cells of width twice the object size and greater, and will lie in at most two cells per level along each dimension. When a moving object crosses a cell boundary entering a new cell at the object's scale, it tests for collisions against all the objects listed in that cell. When it enters cells at higher levels, it tests only against the objects in that cell of that scale, avoiding tests against the other objects of smaller scales also listed there but which lie in more distant cells at lower levels.

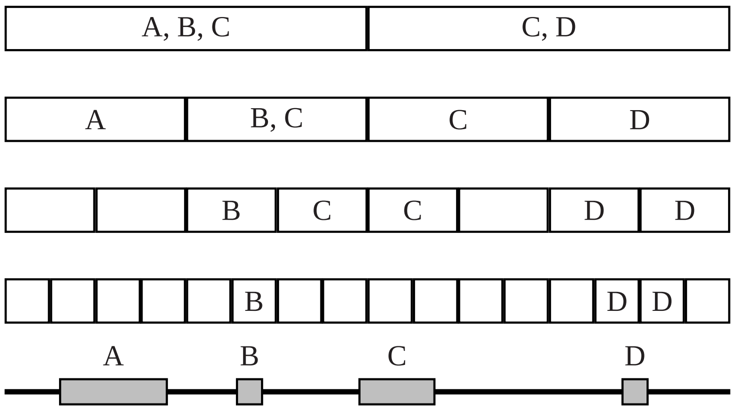

Figure 4.1: Hierarchical spatial hashing.

4.3 The Dual Quad-tree

One of the problems with quad-tree nesting, and likewise with Mirtich's hierarchical spatial hash tables, is that boundaries of the different levels of detail coincide. Objects C and D in figure 4.1, for instance, each span two cells not just as their level of the hierarchy, but also a number of levels above them. In particular, the origin shares a boundary with every one of the quad-tree layers making it the most expensive boundary for a moving object to cross.

The solution is to use the dual of each layer of the quad-tree which swaps the boundaries and the cells of the original quad-tree layers with each other. In particular, the origin, which was the boundary of quad-cells at all levels of the original tree, now becomes the center of a single cell at all levels of the dual.

We can use the dual quad-tree for proximity and collision detection, just as in Mirtich's spatial hashing. Because of the interleaving, cell boundaries occur twice as often, but objects lie in fewer cells. In particular, assuming that the objects are at most half the size of their base level in the hierarchy, an object will only overlap two cells in one level at a time. In figure 4.3, for instance, objects A, B, and C each overlap two cells in their base levels, but are fully contained by a single cell at all higher levels. Object D, is fully contained by a single cell at its base level, but spans a cell boundary at the third level.

4.3.1 Bounding an arbitrary region of space

An additional feature of the dual quad-tree is that there exists at some level a single dual-quad cell that fully contains any given bounded region of space. The same cannot be said of the quad-tree since, for instance, no single cell spans the origin. We can use this feature to manage the instantiation

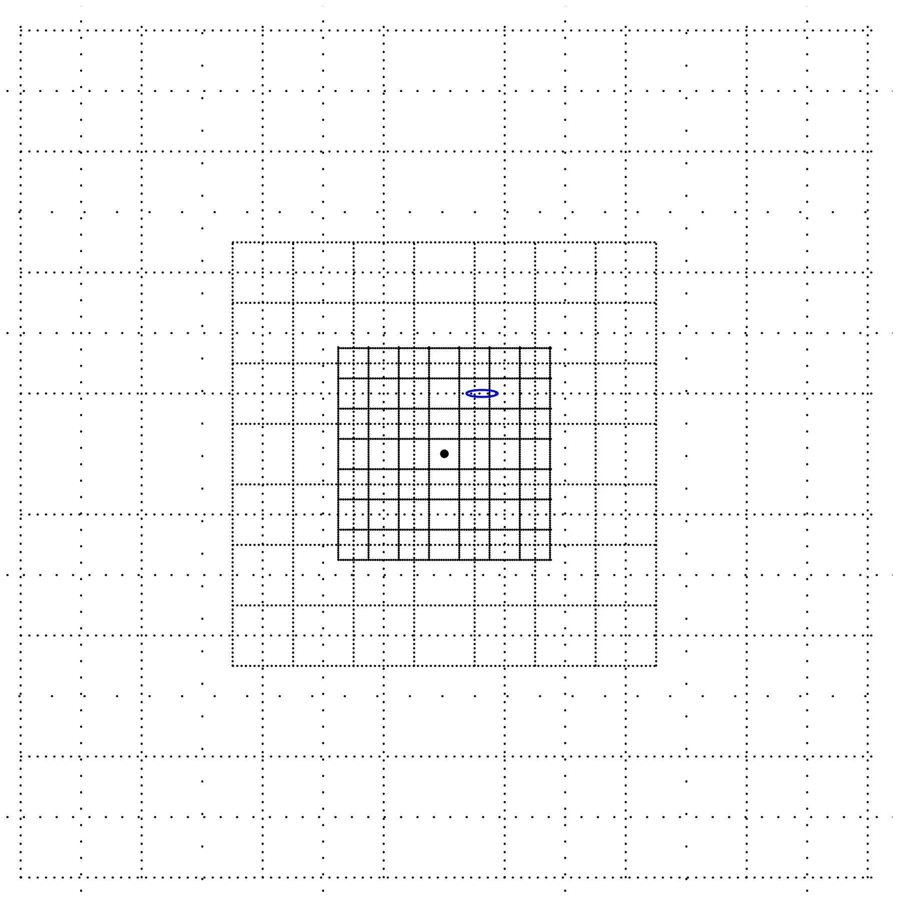

Figure 4.2: The dual quad-tree. The camera is centered at the origin. The region circled in blue is fully contained by a cell a level four.

of objects that roam over a given area as we will discuss below.

The blue region in figure 4.2 shows that an object of "size" 1 may need a cell of size 16 to completely contain it. We'll now explain why this factor of 16 is sufficient.

Let us consider a dual quad-tree whose x-divisions, at the smallest level, occur at odd integers, so that the "center" level-1 x-region is the interval $[-1, 1]$.

Suppose that U is a region of R^2 whose projection to each axis is contained within an interval of length $w > 0$.

Let $k = \max(\lfloor \log_2 w \rfloor, 1)$. We'll show that U is properly contained in some cell of level no greater than $k + 3$, thus showing that the factor of 16 is a tight bound.

First, replace U with a square of side $s < 2^{k+1}$ that contains U. If we can prove this square is properly contained in (by which we mean "contained in the interior of") a cell of level $k+3$, then U

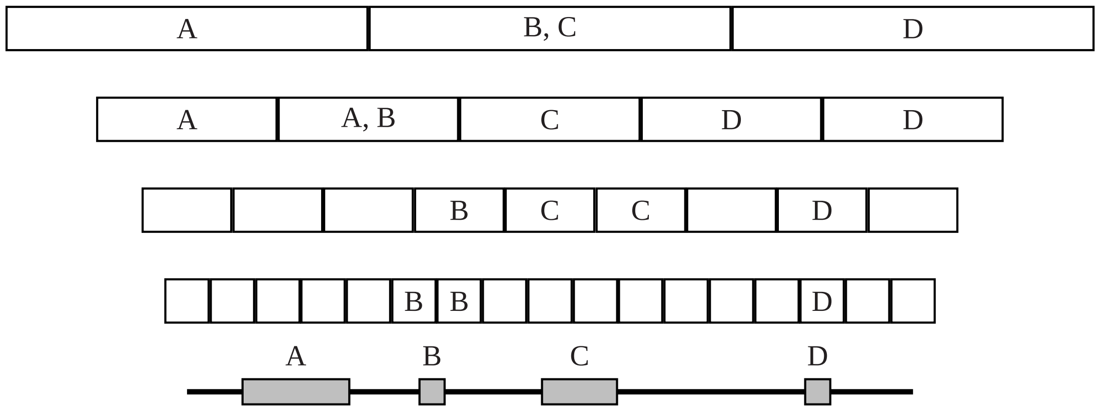

Figure 4.3: Dual quad-tree spatial hashing.

will evidently be properly contained in such a cell. Henceforth we'll use U to denote this square.

Second, by the self-similar nature of the problem, we can ignore all the cells of level smaller than $k + 1$; hence, by scaling everything down by a factor of 2^k, we will have $1 \leq w < 2$, and we need to show that any square of side less than 2 is properly contained in a cell whose side is no more than 16, i.e., a cell of level 4.

The dual quad-tree structure on the plane induces a dual binary-tree structure on each axis, like the one shown in figure 4.3.

We'll use the following simple observations to construct a proof.

- Let V denote the projection of U to the x-axis. If V contains no cell-division at some level i, then V is properly contained in some level-i cell of the x-axis dual binary tree. The same goes for W, the projection of U onto the y-axis.
- Because $w < 2$, and because cell-divisions in each axis occur at integer points, the projection V intersects at most two cell-divisions. Hence there are at most two levels of the x-axis dual binary tree at which V is not properly contained in a cell of that level. The same goes for W.
- Because the cell-divisions any cell above level 1 are separated from one another by at least 2 units, if V intersects two x-axis cell-divisions, then one of them is a level-1 division.
- If V is properly contained in an x-axis cell of level i, and W is properly contained in a y-axis cell of level i, then U is properly contained in the Cartesian product of these cells, which is a cell in the dual quad-tree at level i.

Now consider cells of the dual quad-tree of levels 1, 2, 3, and 4. We want to show that U is properly contained in at least one of these.

Consider the set L_V consisting of levels 1, 2, 3, or 4, for which V contains no cell-division at that level. Since V meets divisions of at most two levels, and *if* it meets subdivisions at two levels, at least one of them must be level-1, it's clear that L_V contains at least two of the numbers 2, 3, and 4. The same argument holds for L_W.

Since L_V and L_W both have at least two elements of the set $\{2, 3, 4\}$,it's clear that $L_V \cap L_W \neq$. Hence there is some level $\ell \in \{2, 3, 4\}$ such that V is properly contained in an x-cell of level ℓ, and W is properly contained in a y-cell of level ℓ. From this, we can conclude that U is properly contained in a dual quad-tree cell of level ℓ, and the proof is complete.

By a similar proof, we can see that a region of dimension d will be fully contained by a d-dimensional cell of level $d + 2$ or lower.

4.4 Adding the dual quad-tree to the nested LOD cache

We can introduce the dual quad-tree structure to the nested LOD cache discussed in chapter 3. Each dual cell is stored hanging off of the quad-tree cell to its upper right. That is, the quad-tree cell $(0, 0)$ which spans from the origin to $(2, 2)$ stores a pointer to the dual cell $(0, 0)$ which spans the origin from $(-1, -1)$ to $(1, 1)$. Dual cells are created only when all four of the overlapping quad-cells exist and so the left column and bottom row of quad-cells in the cache do not have dual cells.

4.5 Re-initializing cells and object state

When the view enters a cell containing only static objects, the procedural generator can instantiate them and set their attributes using the deterministic pseudo-random number generators seeded from the cell's seed. Objects with dynamic state are not quite as simple.

Chenney [12] reinitializes spatially bounded behaviors when they re-enter the view based on one of three possible levels of detail. The highest level is direct simulation, which requires full computation from a previously known state. The next is an approximating neural net which has been trained off-line. The third is a statistically chosen random initialization.

The choice of which method is used to re-initialize the state of a given system depends on how long it has been since the viewer last saw the system. For short glances away from the system, the viewer will still have a fresh memory of the position and motion of the various components of the system and would, for instance, be able to recognize if a ball that was falling at one point failed to strike the ground at a second glance a few moments later. Over time, however, the general chaos of most systems breaks down the viewers expectations for any particular configuration of the system, allowing it to be re-initialized either from a coarser approximate simulation or from a statistically chosen random configuration.

For the convenient management of objects in large scale worlds, we would like for all object behavior to be described such that any peculiarities of state that a viewer can observe will tend to converge back to statistically likely states. Once any out of view object has so converged, the run-time system is free to forget about that object and its state and let the procedural generators initialize it from scratch. In a sense, any signal the object's state once held has now decayed into noise.

4.5.1 Instantiating roaming entities

In many applications, there will be certain entities which are unique to a particular region of the world over which they roam. Consider, for instance, a lone stag that lives in a particular forest. The deer is much smaller than the forest and, ideally, should only be instantiated with some probability when a cell of the appropriate scale is introduced to the nested LOD cache. Yet if any appropriately sized cell of the forest has a chance of instantiating the deer, there is the possibility that multiple of them might do so at once.

We can solve the problem of multiple instancing, however, by storing instancing knowledge of the deer in the "owner" cell that contains the full area over which it roams. When any one of the smaller scale cells instances the deer, it registers that fact with the owner cell which will then block instancing of the deer by any other smaller cells. Thus, the smaller cells drive the instantiation of smaller objects but are able to share the impact of that instantiation with their neighbors through the owner cell.

4.6 Other considerations

The simulation and visual impact of objects may well be larger than the size of the object itself, thus causing the effects of that object to be visible at a much coarser level of detail than the one containing the object. Consider, for instance, object shadows, especially near dusk or dawn, which can stretch far across the landscape, much longer than the object itself. As the edge of the view cache moving through the world reaches the cell containing such an object, its shadow will suddenly pop into view much closer and, therefore, much larger than we would prefer.

Consequently, objects must be introduced based on the size of their area of influence. Such influences can be significantly difficult to recognize in general, since doing so requires the full understanding of an object's behavior. Consider, for instance, a person who presses the detonator for a nuclear explosion. Understanding the chain of consequences requires semantic synthesis to know what attributes are important, potentially at a much higher level of detail than the object itself would be visible. Such language analysis is beyond the scope of this dissertation.

4.7 Conclusions

In this chapter, we introduced the dual quad-tree as an efficient structure for detection of collisions between objects at a variety of scales. We showed further how the dual quad-tree may be used to manage the instantiation of objects that roam dynamically through an on-the-fly procedurally generated world.

Chapter 5

Multi-resolution Behaviors

In this chapter, we will introduce a number ideas about multi-resolution behaviors, that is about behaviors which scale both in their computational cost as well, consequently, as in the accuracy of their output. We will then present three proof-of-concept examples demonstrating applications of these ideas.

5.1 Event-driven behaviors

5.1.1 Definition of behaviors

Throughout this chapter, we will use the term *behavior* to refer to any arbitrary continuously time-varying value. Ideally, we would like to describe these behaviors by closed-form functions of time, but in general, doing so is not possible and we will settle for numeric approximations in computing their values. The continuity of the behaviors is broken by discrete, instantaneous events that operate on some subset of the behaviors that exist within the overall system. This definition is similar to that used by Elliott in SUN's TBAG architecture [20] and, successively, in Microsoft's DirectAnimation [40] and in Fran [19].

5.1.2 Type of events

Events are discrete changes that occur to one or more behaviors. An event could simply change the value of one or more behaviors, change the motion equations describing behaviors, create new or delete existing behaviors, or trigger other events. We group events into two classes: external and internal. *External* events are generated externally to the process being considered, such as by the user or from a network connection. *Internal* events, on the other hand, derive from some predicated condition being met, such as an object's height reaching zero triggering a collision event with the floor.

5.1.3 Problems with the naive approach

Many interactive graphics systems test every internal event predicate each frame, a mechanism which has a couple of disadvantages. The first is that the predicates are typically tested and evaluated in an arbitrary order. For accuracy, we want to handle the events in the chronological order in which they occur. If multiple events fire within a given frame, however, they will quite possibly be handled out of order. If any of these out of order events involve the same behaviors, they will generate incorrect results. The second disadvantage to testing all the events every frame is that likely only a few of them will have fired. Ideally, we would only evaluate an event predicate at the time that it fires.

5.1.4 Use *Discrete Event Simulation*

Mirtich [41, 42] solves both of these problems by evaluating the behaviors using Discrete Event Simulation (DES), which stores events in a priority queue independent of any particular frame rate. As each event is popped off the queue, the behaviors it involves are updated to the time of the event and then the results of the event are computed. External events are simply entered into the queue at the time that they occur. Internal events are represented in the queue by prediction events based on an estimate of the earliest time that the predicate might become true. Such a prediction requires an understanding or analysis of the behaviors involved in the event.

5.1.5 Reduce the number of events

Most behaviors will be subject to numerous potential interactions with other behaviors and will, therefore, have a long list of potential events. When an event changes the equations that define a behavior, many to all of its event predictions may have to be changed. The goal for the programmer must, therefore, be to specify as few events as possible for each behavior. In n-body collision detection, for example, each of n objects may collide with each of the others. This results in on the order of n^2 potential events. A more intelligent approach, however, recognizes that in order for a given object to collide with a distant one, it must first pass any objects in between the two. By testing only for collisions or passing of nearby neighbors, such as along the primary axes, we can turn n event tests per object into just a small constant per object. When two objects collide, we need only update the future collision event predictions that involve those two objects with their relatively few nearest neighbors.

5.1.6 A simple example

One simple example is that of a row of balls that are free to move without drag along the x-axis between two fixed endpoints. The balls are of differing sizes, have mass appropriate to their size, and obey the law of conservation of momentum when any two of them collide. In this simple case, the motion of each ball is linear and may easily be described by the closed form function of time: $x(t) = x_0 + v * t$. Given two such expressions, we can solve for the time t at which the two balls collide. If the balls are moving away from each other, this time will be in the past, otherwise the collision has yet to occur. Note that if anything changes the motion of either ball, such as a collision with a third ball, the initial event prediction will be incorrect and will need to be recomputed.

We begin our simulation by computing the collision times for each adjacent pair of balls and inserting all the future collision events into an event queue. At each frame of the animation, we first handle any events in the queue with a time less than the frame time. Each collision event stores the event time and pointers to the two balls. Since the collision will change the motion of both balls and, thereby, change the predicted time of their future collisions, we start by removing all such collisions from the queue. We can find these events in constant time by storing a list of each object's events on the object. As described above, to handle the collision, we first update the state of the two balls to that at the time of the collision and then compute the new velocities after the collision. Finally, we predict any future collisions with each ball's other neighbor and insert those predictions into the queue.

A 3-D example of such an event based physical simulation is shown in figure 5.1. In this case, the collision resolution event further computes the changes in the rotational velocities of the balls when they strike each other or the floor. Collisions between ballistic objects such as between the balls simplifies to a linear system since the objects are all subject to the same force, namely, gravity. Computing the collision time with a static object, such as with the three large fixed-position spheres in the middle of the box, requires the solving of a quartic. A simpler technique, as suggested by Mirtich, is to solve a simpler linear system, predicting instead the collision time with a plane that separates the two objects. At that predicted time, the event generates a new dividing plane between the objects and predicts a new time. When the distance between the objects is sufficiently close, they are defined as colliding and the system breaks the recursion by instead computing a collision response.

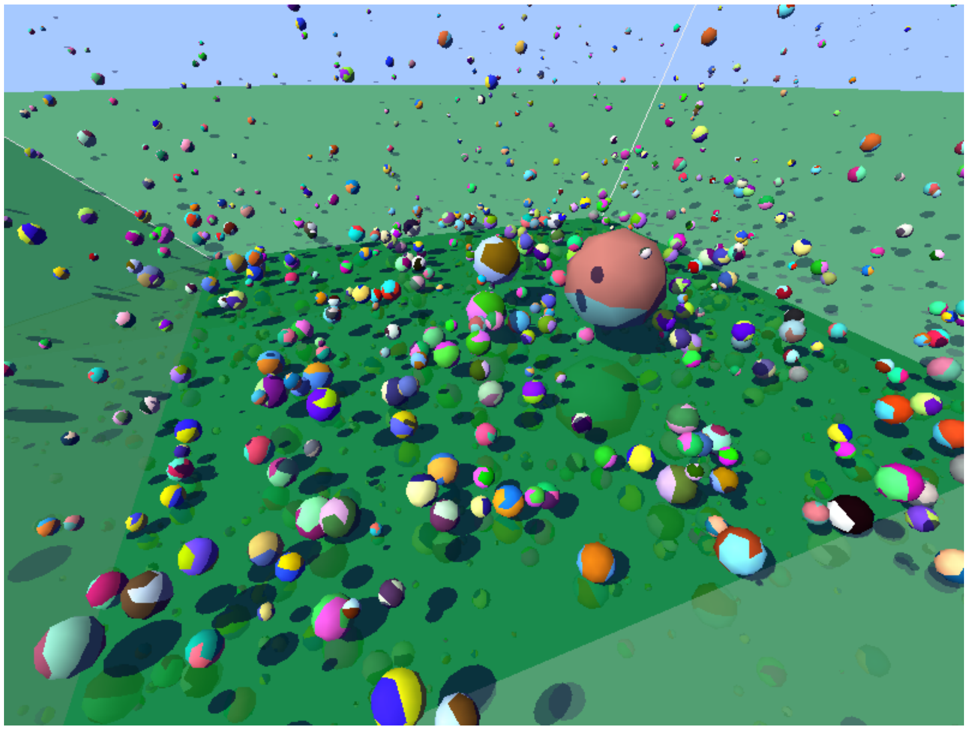

Figure 5.1: 1000 balls in an event-based physical simulation.

5.1.7 Event-based computation

Event-based algorithms have been applied by Basch and Guibas to a variety of computational geometric problems in what they refer to as Kinetic Data Structures [6].

Event-based computations must be certain to make use of effective thresholds on the time intervals used between events. Consider, for instance, Zeno's paradox in which an arrow fired at a target must first reach half the distance, and then half again, and then half again. If each of these distances were associated with the firing of an event, the arrow would eventually stall as more time was spent on the administrative overhead of processing the event that was spent on advancing the state of the arrow. Mirtich[41] handled such conditions in his impulse physics simulations when, for instance, a ball was barely bouncing off of a surface, by detecting the small forces involved and switching the system to one of resting contact.

We have a similar situation in rendered systems which are bound by a discrete frame rate. Our threshold in this case lies not just with the simulation itself, but also in our ability to perceive it and the quality of that perception. Small vibrations, for instance, can result in aliasing when our sampling of them falls below the Nyquist limit. Our thresholds must, therefore, also apply to the rendering frame rate, switching us, perhaps, to a blurred geometric model to represent the rapid motion between different states. In effect, we do not want to push events through the system faster than their results can be used.

5.2 Higher level and multi-resolution event-based behaviors

Higher level behaviors are often described by finite state machines which group collections of continuous behaviors into nodes and use discrete events as the arcs between them. Yet more complicated behaviors may be described by hierarchical FSMs [8] in which each state contains a full FSM of its own.

We introduce the idea of multi-resolution behaviors as those that provide multiple representations for such a hierarchically bound state. Such representations take on two distinct forms akin to the multi-resolution representations of objects, namely those that make discrete changes between different behavior models and those than scale a single behavior continuously. A discrete change is one that wholly replaces components of the behavior with alternate representations, such as switching between components in a state machine. A more continuous change may, for instance, vary the time step used in an integration, providing a less accurate, but quicker to compute behavioral result. Carlson and Hodgins provide an example of the use of multiple discrete behaviors [9] in a simulation of a game between kangaroo-like robots.

Both the discrete and continuous LOD changes to a behavior are *qualitative*, that is, they change the quality of the resultant simulation. We can also make use of *qualitative* simplification which occurs when we vary the quantity of data on which the behavior must operate, typically through limiting the number of events it must process. Both qualitative and quantitative simplification are frequently used in the rendering of objects. View culling is a quantitative simplification which helps to keep unnecessary geometry out of the pipeline. Likewise, geometric simplification is a form a qualitative simplification.

As with geometric LOD, we can use multi-resolution behaviors along with event culling to accelerate the processing of behaviors of objects in our world.

5.2.1 *Qualitative* behavior simplification

Figure 5.2 shows two views of a multi-resolution simulation of monsters roaming through a building, designed in collaboration with White[61]. The behavior of the monsters is chosen discretely based on their visibility by the user. In the image on the right, the gray rooms are each at least partly visible to the user, so the monsters they contain compute their full simulation including navigating around one another. Monsters out of view in the black rooms, however, can approximate their behaviors, including skipping the collision avoidance behavior altogether. Monsters at an intermediate stage, such as those in the top room which is visible to the player but quite distant, may approximate their behaviors as long as they adhere to what little the user is able from that distance. When one of the darkened rooms comes into view, the monsters snap to a plausible position along their motion paths, meeting the expectations of the viewer that they not be colliding.

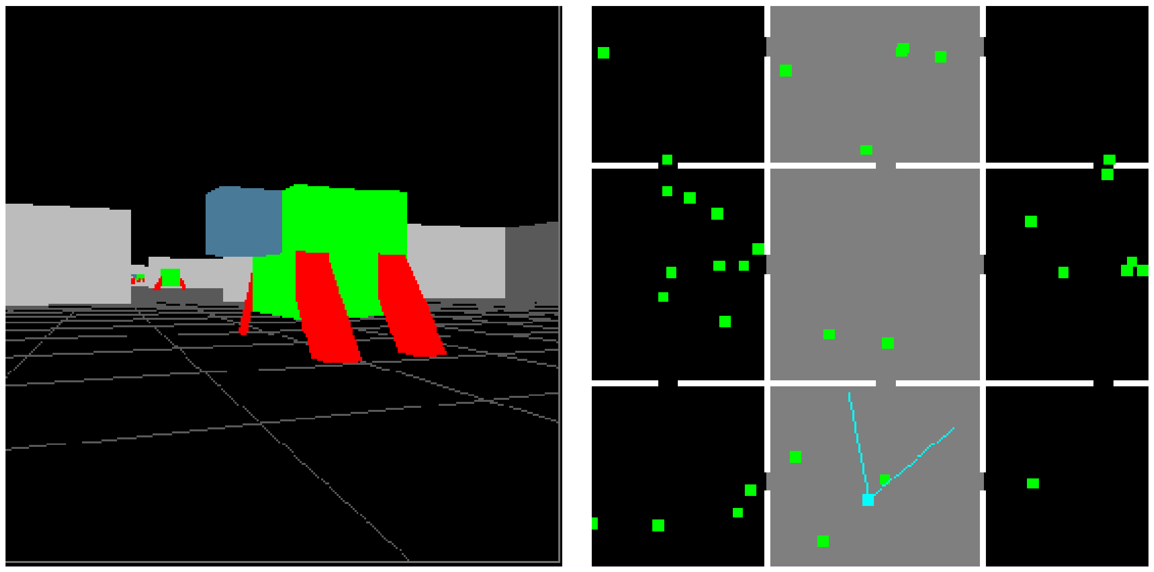
Figure 5.2: A multi-resolution simulation of monsters in a building. Monsters in the visible rooms in gray simulate at full resolution including collision avoidance. Those in the black rooms approximate their behavior, snapping to a plausible state when the room comes into view.

5.2.2 LOD of behaviors through *quantitative* data simplification

As an exploration of quantitative simplification applied to the tools used in artificial intelligence, we developed, in collaboration with Reiff, a multi-scale path planning engine [54]. Using the quad-tree structure of Chapter 3, this engine derives path maps through a crowded plain filled with obstacles. Through the hierarchical structure of the tree, objects of different scales choose their routes through the plain based on their ability to find clear passage. The engine supports the dynamic addition and removal of objects at all scales such as happens in our view-dependent, semantically synthesized world when the user moves about.

Behavior simplification occurs when there are fewer events for an object to process. Far from the user, where small objects have been removed, an entity will have, for instance, fewer collisions to avoid. In trade-off, however, it must adapt its interpretation of the terrain appropriately to best approximate its reaction to the small items which would otherwise be there. For instance, when scrambling up a rocky hillside, an entity would need to approximate the roughness of the ground when small rocks have been culled because the viewer is too far away to see them.

5.3 Plausible emulation – qualitative simplification

5.3.1 Plausible physics

Both Hughes and Barzel [5] and, later, Chenney [11] have argued that the clean and mathematically precise physics that are often simulated in computer applications is not appropriately representative of the real world. Actual balls bouncing on a surface, for instance, are subject to the minute flaws it contains which subtly distort their trajectory after impact. Imperfections on the surface of the balls will, likewise, affect their flight path as they move through the air.

Rather than insisting that physical computation be fully numerically precise, therefore, we can define our more appropriate goal as that of providing the viewer with a plausible emulation of the physical process. That is, we can and should use approximations for the physics which is designed to best meet the expectations of the user, yet which, to save on computation, go no further.

5.3.2 Multi-resolution ballroom dance floor emulation

A full system that ties together many of the ideas presented here was designed in collaboration with Bhuphaibool[7] which provides a simultaneous multi-resolution representation of geometry, animation, simulation and behavior through the emulation of a crowded ballroom dance floor.

Figure 5.3 shows two views of approximately 100 pairs of dancers performing the box step. Nearby dancers have fully articulated and animated bodies showing the expected contact with the floor and compute an accurate collision avoidance through the choice of dance moves appropriate to avoid contact with other dancers. As the distance increases between the couple and the viewer, the geometry is simplified by removing the more subtle joint motions such as at the ankle and then, eventually, the knee, replacing the separate pieces of transformed geometry with fewer rigid parts. As the geometry simplifies, the animation also adapts, such as that for the hips, to compensate and provide a visually expected result. The behavior is likewise simplified with distance until, for the most distant dancers, whose geometry has been reduced to a vertical cylinder, collision avoidance is removed altogether. As an exception to the simplification as tied to distance to the view, dancers who choose the twirl move have their detail level temporarily raised since this more dramatic motion is more likely to draw the viewer’s attention.

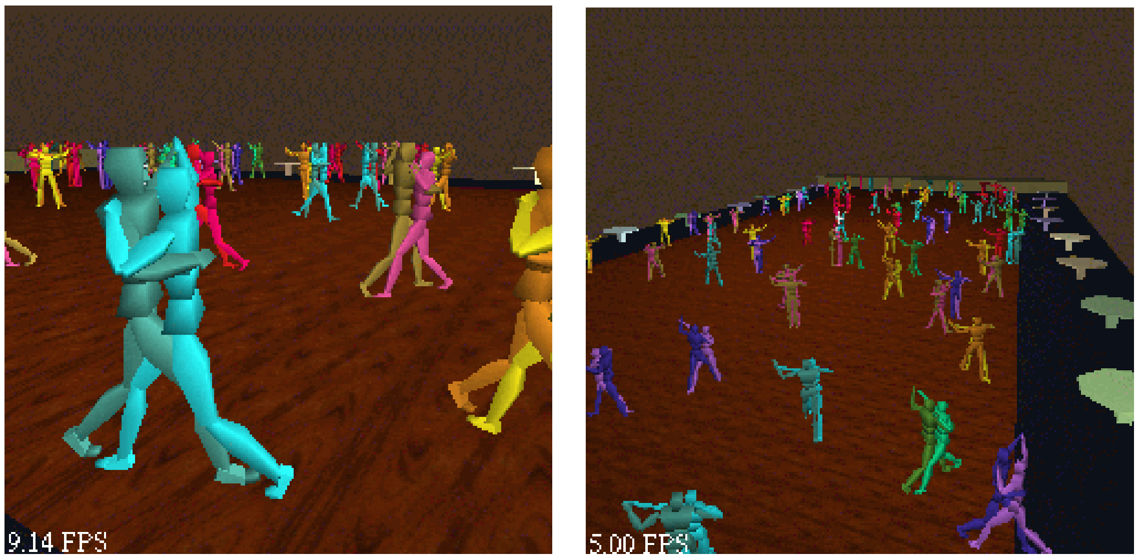

Figure 5.3: Two images of a multi-resolution ballroom dance simulation.

5.4 General object LOD

There are numerous techniques that have been used to provide geometric levels of detail for objects, including adaptive tessellations [59], progressive meshes [29], and wavelet compression [36].

Funkhauser [24] defines object *importance* as a criterion for choosing the level of detail for an object. An object's importance can be defined by a wide range of sources. The most obvious, and certainly most relevant for rendering, is its visual attributes. One visual attribute is how large the object projects onto the screen, that is, its size divided by its distance to the viewer. Larger objects in screen space will need a higher level of detail to avoid artifacts. On the other hand, an object that is passing quickly across the screen may blur out much of its detail and could suffice with a lower level of detail. On the other hand, an object moving slowly enough not to blur, but moving in contrast to a still background will most likely draw attention to itself calling for an increased level of detail. Likewise, any object with rendering attributes, be they color, brightness, shape, shininess, etc. which stand out from the background or other objects in the scene will draw the viewer's eye. Funkhauser's system weighs the benefits of rendering an object with increased LOD against the costs such rendering entails.

We can extend this idea of importance throughout an object's behavior as was done in the case of the twirling ballroom dancers above. Ideally, such importance would be measured using perceptual techniques [21] and tracing back through the behavioral definition to define a measure of importance on the triggering events. By thus defining the behavior of the objects in the world, the runtime system can simplify, either away or to approximations, those events that result in little visual or otherwise experiential impact on the user of the system.

Chapter 6

Conclusions

6.1 Results

All of the landscape images in this dissertation were generated from a square sixteen layer quad-tree cache with an inner radius of ten, for a cache size of 40x40 or 1600 cells per layer and 25,600 cells in full. The finest layer, numbered 0, has a cell diameter of one meter, so layer *i* has a cell radius of 2^i. The rendered mesh skips the finest layer, but uses the next twelve layers for a total mesh radius of $2^{12} \times 20 = 81,920$ meters at the middle of the square edge to approximately 116km at the corners.

6.2 Contributions

Though the general problem of specifying a declarative, event-based language for the expression of behaviors amenable to multi-resolution analysis and simplification remains a challenging open problem, this dissertation presents a framework for their efficient use. The multi-resolution behaviors prototyped in chapter 5 each provide an adaptive level of detail that scales sub-linearly with the apparent size of the world.

The *dual quad-tree* structure presented in chapter 4 provides and efficient multi-scale collision or proximity detection algorithm. With its separation of the geometry by size it meshes well with the view-dependent generation used for the world synthesis which, likewise, adds and removes objects at specific scale ranges. We can further use its attribute of providing an efficient coverage of any bounded region to manage the probabilistic instantiation of unique dynamic entities throughout the world.

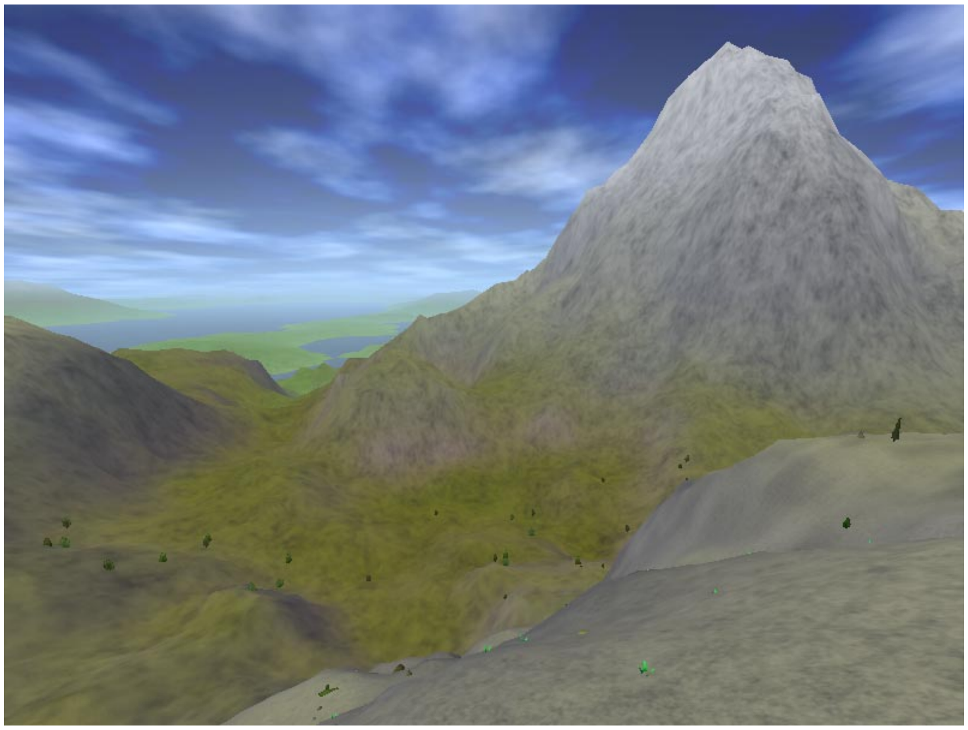

Figure 6.1: A mountain. The height generating noise function is scaled by a second roughness noise function leading to the variety of terrains ranging from the flatter plains in the background to the rough peak seen here.

In chapter 3, we presented the *nested LOD quad-tree cache* which triggers the instantiation of and stores a subset of the globally defined quad-tree that, through semantic synthesis, implicitly stores the world data. Through the cache, we can compute a simple, efficient polygonal tessellation of the terrain. The interleaved update mechanism allows us to batch the geometric data into time-coherent, fixed size pieces ideal for efficient rendering with vertex arrays while still providing a gradual, though discrete form of fractional level of detail.

6.3 Future work

The design of the data structures presented throughout this dissertation for managing world data were motivated by our thinking about the locality constraints imposed by on-the-fly semantic modeling of behaviors. What remains to be designed, however, is a description and implementation of a language that can synthesize such behaviors based on their semantics.

Although the proof-of-concept implementations of multi-resolution behaviors demonstrate their utility in large worlds, they have not yet been implemented as a part of the larger on-the-fly procedurally generated world management data structures introduced through chapters 3 and 4. Such an integration would make use of the quad-tree cells outside of the nested LOD cache to preserve simulation state information for any sufficiently semantically important objects they contain, such that the runtime system could re-initialize them appropriately if they come back into view.

We intend, further, to explore the management of the on-the-fly instantiation of dynamic objects, maintaining the appropriate history of motion paths necessary to match the user's memory of objects seen. For instance, if the user passes a slow moving object, moves out of visual range of it, and then stops, he would expect the object eventually to catch back up with him with at least some degree a probability allowing for the appropriate possibility that the object might also have stopped or turned. On the other hand, through the accumulation of possible events affecting an object's behavior, we can assume that any specifically witnessed object state may eventually decay into noise and be forgotten. Though Chenney explored these ideas for the behaviors of systems confined to a bounded space [11], the dynamic cell allocation of our quad-trees should allow for the arbitrary tracking of fully mobile entities such as is done on a small scale for cars in the Iowa driving simulator [32].

Through the design and integration of a language for semantic synthesis of multi-resolution behaviors with the nested LOD cache, we eventually intend to demonstrate an on-the-fly procedurally generated fully dynamic, interactive virtual world.

And finally, as an implementation issue, the scale of the lands we produce is effectively bound by the machine precision of floating-point numbers. When the camera moves sufficiently far from the origin, say 10,000km or so, the resolution of the numbers breaks down such that the position of neighboring polygon vertices is able to map to the same value. This can lead to errors both in the texture mapping and in the geometry itself as shown in figure 6.3. We can address this problem by storing a higher precision translation for each cache level, keeping the geometry instead in that more local reference frame. Simulation between objects at different levels will always have to take the potentially differing translations into account. Likewise, texture coordinates for the ground detail textures and for the sky map will also have to map cleanly between the different coordinate systems which places restrictions on their variability.

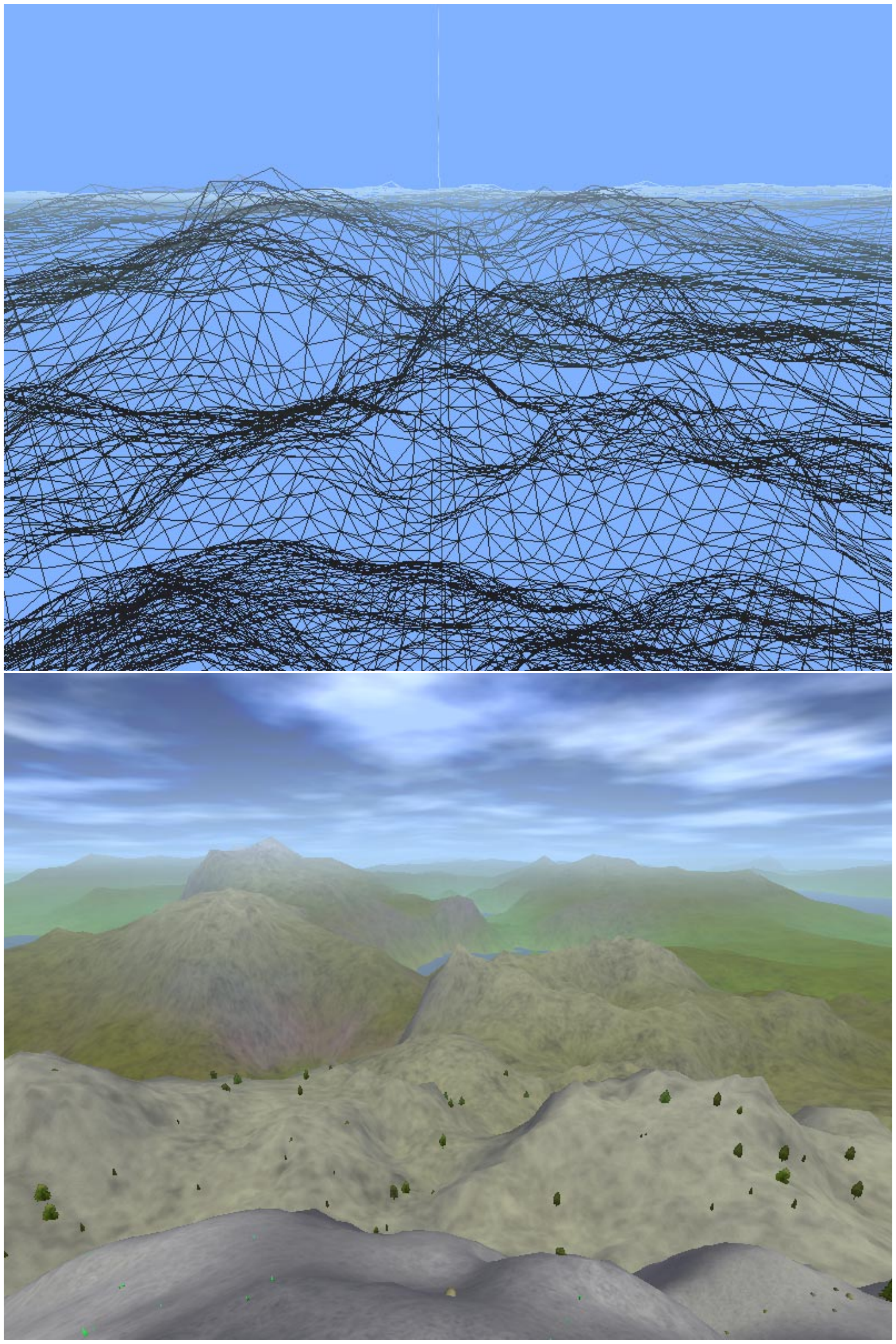

Figure 6.2: Rugged mountains in wireframe and in solid polygons. In wireframe, we can see the approximate visual uniformity of the sampling at all distance from the viewpoint. Though subject to artifacts in areas of high curvature, the efficiency of the uniform sampling allows a higher overall terrain tessellation.

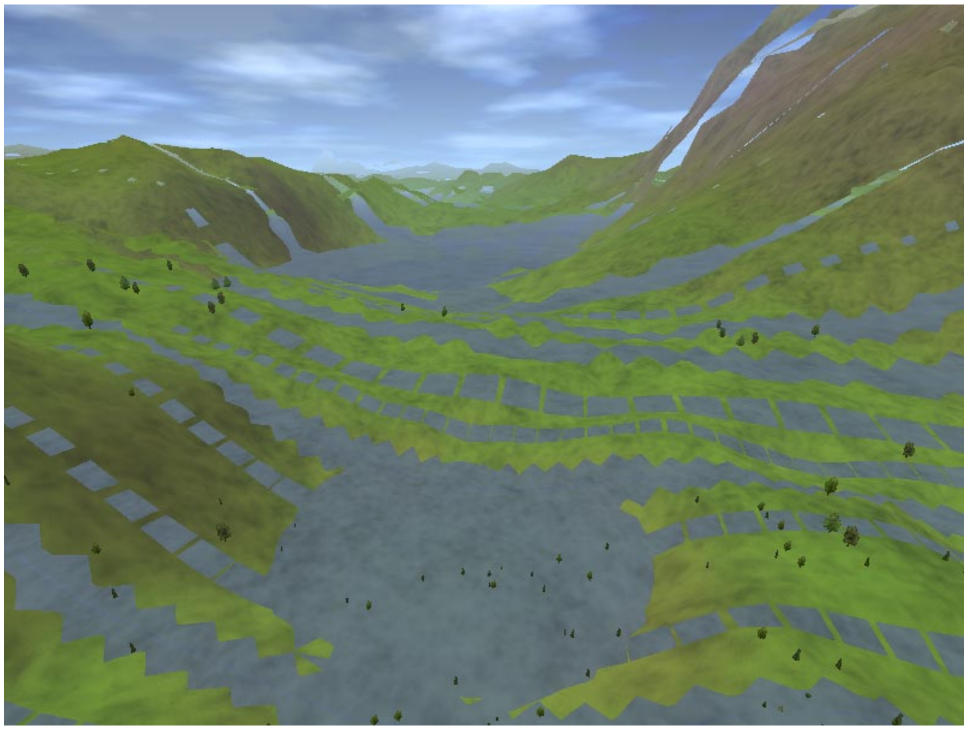

Figure 6.3: Far from the origin. Floating-point numbers begin to fail us when the camera is 10,000km from the origin. This problem can be solved by using a higher resolution coordinate system to reposition the origin such as to that of each of the root quad-tree cells.

Bibliography

[1] Activeworlds Corp. *Active Worlds*, 1999. Commercial online virtual environment. `<http://www.activeworlds.com/>`.

[2] Christopher Alexander, Sara Ishikawa, and Murray Silverstein. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, 1977.

[3] Daniel Aliaga, Jon Cohen, Andrew Wilson, Eric Baker, Hansong Zhang, Carl Erikson, Kenny Hoff, Tom Hudson, Wolfgang Stürzlinger, Rui Bastos, Mary Whitton, Fred Brooks, and Dinesh Manocha. MMR: An Interactive Massive Model Rendering System Using Geometric and Image-Based Acceleration. In *Symposium on Interactive 3D Graphics*, pages 199–206, 1999.

[4] Phil Amburn, Eric Grant, and Turner Whitted. Managing Geometric Complexity with Enhanced Procedural Models. *Computer Graphics*, 20(4):189–195, 1986.

[5] Ronen Barzel, John F. Hughes, and Daniel N. Wood. Plausible Motion Simulation for Computer Graphics Animation. In *Computer Animation and Simulation '96*, pages 183–197, 1996.

[6] Basch, Guibas, and Hershberger. Data Structures for Mobile Data. In *SODA: ACM-SIAM Symposium on Discrete Algorithms (A Conference on Theoretical and Experimental Analysis of Discrete Algorithms)*, 1997.

[7] Dom Bhuphaibool. Multi-resolution Animation and Behavior in Densely Populated Scenes. Master's thesis, Brown, 1999.

[8] Bruce M. Blumberg and Tinsley A. Galyean. Multi-Level Direction of Autonomous Creatures for Real-Time Virtual Environments. *Computer Graphics*, 29(Annual Conference Series):47–54, 1995.

[9] Deborah A. Carlson and Jessica K. Hodgins. Simulation Levels of Detail for Real-time Animation. In Wayne A. Davis, Marilyn Mantei, and R. Victor Klassen, editors, *Graphics Interface '97*, pages 1–8. Canadian Human-Computer Communications Society, 1997.

[10] V. Chatzi and F. Preparata. Integer-coordinate crystalline meshes. In *Proceedings of the Swiss Conference on CAD/CAM*, 1999.

[11] Stephen Chenney and D. A. Forsyth. Sampling Plausible Solutions to Multi-Body Constraint Problems. In Kurt Akeley, editor, *Siggraph 2000, Computer Graphics Proceedings*, pages 219–228. ACM Press / ACM SIGGRAPH / Addison Wesley Longman, 2000.

[12] Stephen Chenney and David Forsyth. View-Dependent Culling of Dynamic Systems in Virtual Environments. In *Symposium on Interactive 3D Graphics*, pages 55–58, 1997.

[13] Jonathan D. Cohen, Ming C. Lin, Dinesh Manocha, and Madhav K. Ponamgi. I-COLLIDE: An Interactive and Exact Collision Detection System for Large-Scale Environments. In *Symposium on Interactive 3D Graphics*, pages 189–196, 218, 1995.

[14] R. Cook. Shade Trees. In Hank Christiansen, editor, *Computer Graphics (SIGGRAPH '84 Proceedings)*, volume 18, pages 223–231, 1984.

[15] Corel Corporation. *Bryce 5*. Commercial product.

[16] Oliver Deussen, Pat Hanrahan, Bernd Lintermann, Radomír Měch, Matt Pharr, and Przemyslaw Prusinkiewicz. Realistic Modeling and Rendering of Plant Ecosystems. *Computer Graphics*, 32(Annual Conference Series):275–286, 1998.

[17] Mark A. Duchaineau, Murray Wolinsky, David E. Sigeti, Mark C. Miller, Charles Aldrich, and Mark B. Mineev-Weinstein. ROAMing Terrain: Real-time Optimally Adapting Meshes. In *IEEE Visualization '97*, 1997.

[18] David S. Ebert, F. Kenton Musgrave, Darwyn Peachey, Ken Perlin, and Steven Worley. *Texturing & Modeling: A Procedural Approach*. AP Professional, second edition, 1998.

[19] Conal Elliott and Paul Hudak. Functional Reactive Animation. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '97)*, volume 32(8), pages 263–273, 1997.

[20] Conal Elliott, Greg Schechter, Ricky Yeung, and Salim Abi-Ezzi. TBAG: A High Level Framework for Interactive, Animated 3D Graphics Applications. *Computer Graphics*, 28(Annual Conference Series):421–434, 1994.

[21] James A. Ferwerda, Sumanta N. Pattanaik, Peter Shirley, and Donald P. Greenberg. A Model of Visual Adaptation for Realistic Image Synthesis. *Computer Graphics*, 30(Annual Conference Series):249–258, 1996.

[22] Alain Fournier, Don Fussell, and Loren Carpenter. Computer rendering of stochastic models. *Communications of the ACM*, 25:371–384, June 1982.

[23] H. Fuchs, Z. Kedem, and B. Naylor. On Visible Surface Generation by a Priori Tree Structures. In *Computer Graphics (SIGGRAPH '80 Proceedings)*, volume 14, pages 124–133, 1980.

[24] Thomas A. Funkhouser and Carlo H. Séquin. Adaptive Display Algorithm for Interactive Frame Rates During Visualization of Complex Virtual Environments. *Computer Graphics*, 27(Annual Conference Series):247–254, 1993.

[25] Geoffrey Y. Gardner. Simulation of Natural Scenes Using Textured Quadric Surfaces. In Hank Christiansen, editor, *Computer Graphics (SIGGRAPH '84 Proceedings)*, volume 18, pages 11–20, 1984.

[26] GIS Data Depot. *Free GIS Data*. ThinkBurst Media, Inc. 10 April 2002 `<http://www.gisdatadepot.com/>`.

[27] Johan Hammes. Modeling of Ecosystems as a Data Source for Real-time Terrain Rendering. In C.Y. Westort, editor, *Digital Earth Moving, First International Symposium, DEM 2001*, pages 98–111, 2001.

[28] Hughes Hoppe. View-Dependent Refinement of Progressive Meshes. *Computer Graphics*, 31(Annual Conference Series):189–198, 1997.

[29] Hugues Hoppe. Progressive Meshes. *Computer Graphics*, 30(Annual Conference Series):99–108, 1996.

[30] Philip M. Hubbard. Collision Detection for Interactive Graphics Applications. *IEEE Transactions on Visualization and Computer Graphics*, 1(3):218–230, 1995.

[31] Donald E. Knuth. *The Art of Computer Programming: Volume 2, Seminumerical Algorithms*. Addison Wesley Longman, third edition, 1998.

[32] Jon G. Kuhl, Douglas Evans, Yiannis E. Papelis, Richard Romano, and Ginger Watson. The Iowa Driving Simulator: An Immersive Research Environment. *IEEE Computer*, 28(7):35–41, 1995.

[33] J. P. Lewis. Generalized Stochastic Subdivision. *ACM Transactions on Graphics*, 6(3):167–190, 1987.

[34] P. Lindstrom, D. Koller, W. Ribarsky, L. Hodges, N. Faust, and G. Turner. Real-Time Continuous Level of Detail Rendering of Height Fields. *Proceedings of SIGGRAPH '96*, pages 109–118, 1996.

[35] P. Lindstrom and V. Pascucci. Visualization of Large Terrains Made Easy. In *Proceedings of IEEE Visualization 2001*, 2001.

[36] Michael Lounsbery, Tony D. DeRose, and Joe Warren. Multiresolution Analysis for Surfaces of Arbitrary Topological Type. *ACM Transactions on Graphics*, 16(1):34–73, 1997.

[37] David Luebke and Chris Georges. Portals and Mirrors: Simple, Fast Evaluation of Potentially Visible Sets. In *Symposium on Interactive 3D Graphics*, pages 105–106, 1995.

[38] M. R. Macedonia, M. J. Zyda, D. R. Pratt, P. T. Barham, and S. Zeswitz. NPSNET: A Network Software Architecture for Large-Scale Virtual Environment. *Presence*, 3(4):265–287, 1994.

[39] Benoit B. Mandelbrot. *The Fractal Geometry of Nature*. W. H. Freeman and Company, 1977.

[40] Microsoft. *DirectAnimation*, 1997. In the Microsoft DirectX web page. `<http://www.microsoft.com/directx>`.

[41] Brian Mirtich. *Impulse-based Dynamic Simulation of Rigid Body Systems*. PhD thesis, University of California at Berkeley, 1996.

[42] Brian Mirtich. Timewarp Rigid Body Simulation. In Kurt Akeley, editor, *Siggraph 2000, Computer Graphics Proceedings*, pages 193–200. ACM Press / ACM SIGGRAPH / Addison Wesley Longman, 2000.

[43] F. Kenton Musgrave. *Methods for Realistic Landscape Imaging*. PhD thesis, Yale, 1993.

[44] F. Kenton Musgrave, Craig E. Kolb, and Robert S. Mace. The Synthesis and Rendering of Eroded Fractal Terrains. *Computer Graphics*, 23(3):41–50, 1989.

[45] Origin. *UltimaOnline*, 1997. Commercial online game. `<http://www.uo.com/>`.

[46] Renato B. Pajarola. Large Scale Terrain Visualization Using The Restricted Quadtree Triangulation. In David Ebert, Hans Hagen, and Holly Rushmeier, editors, *IEEE Visualization '98*, pages 19–26, 1998.

[47] Pandromeda, Inc. *Mojoworld 1.0*, 2001. Commercial product.

[48] Yoav I H Parish and Pascal Müller. Porcedural Modeling of Cities. *Computer Graphics*, 35(Annual Conference Series):301–308, 2001.

[49] Heinz-Otto Peitgen and Dietmar Saupe, editors. *The Science of Fractal Images*. Springer-Verlag, 1988.

[50] Frank Perbet and Marie-Paule Cani. Animating Prairies in Real-Time. In *Symposium on Interactive 3D Graphics*, 2001.

[51] Ken Perlin. An Image Synthesizer. In *Computer Graphics (SIGGRAPH '85 Proceedings)*, volume 19, pages 287–296, 1985.

[52] Przemyslaw Prusinkiewicz and Mark Hammel. A Fractal Model of Mountains with Rivers. In *Proceedings of Graphics Interface '93*, pages 174–180, 1993. uses subdivision.

[53] Przemyslaw Prusinkiewicz, Mark James, and Radomír Měch. Synthetic Topiary. *Computer Graphics*, 28(Annual Conference Series):351–358, 1994.

[54] Andrew Reiff. Path Planning in Large-Scale, Dynamic Environments. Master's thesis, Brown, 2001.

[55] Alvy Ray Smith. Plants, Fractals, and Formal Languages. In Hank Christiansen, editor, *Computer Graphics (SIGGRAPH '84 Proceedings)*, volume 18, pages 1–10, 1984.

[56] Sony Online Entertainment. *EverQuest*, 1999. Commercial online game. `<http://www.everquest.com/>`.

[57] Dorothy Strickland, Larry Hodges, Max North, and Suzanne Weghorst. Overcoming Phobias by Virtual Exposure. *Communications of the ACM*, 40(8):34–39, August 1997.

[58] Turbine Entertainment Software. *Asheron's Call*. Commercial online game. `<http://www.microsoft.com/games/zone/asheronscall/default.asp>`.

[59] Greg Turk. Re-tiling polygonal surfaces. *Computer Graphics*, 26(2):55–64, 1992.

[60] *USGS National Geospatial Data Clearinghouse - General Information*. U.S. Geological Survey. 10 April 2002 `<http://nsdi.usgs.gov/>`.

[61] Jeff White. Improving Behavior Efficiency in Virtual Worlds. Master's thesis, Brown, 1998.

[62] Denis Zorin and Peter Schröder. A Unified Framework for Primal/Dual Quadrilateral Subdivision Schemes. *Computer Aided Geometric Design*, 18(5):429–454, 2001.