PROCEDURAL GENERATION AND RENDERING OF LARGE-SCALE OPEN-WORLD ENVIRONMENTS

A Thesis

presented to

the Faculty of California Polytechnic State University,

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

by

Ian Dunn

December 2016

COMMITTEE MEMBERSHIP

TITLE:	Procedural Generation and Rendering of Large-Scale Open-World Environments
AUTHOR:	Ian Dunn
DATE SUBMITTED:	December 2016
COMMITTEE CHAIR:	Professor Zoë Wood, Ph.D. Department of Computer Science
COMMITTEE MEMBER:	Professor Franz Kurfess, Ph.D. Department of Computer Science
COMMITTEE MEMBER:	Professor Chris Lupo, Ph.D. Department of Computer Science

ABSTRACT

Procedural Generation and Rendering of Large-Scale Open-World Environments

Ian Dunn

Open-world video games give players a large environment to explore along with increased freedom to navigate and manipulate that environment. These requirements pose several problems that must be addressed by a game's graphics engine. Often there are a large number of visible objects, such as all of the trees in a forest, as well as objects comprised of large amounts of geometry, such as terrain. An open-world graphics engine must be able to render large environments at varying levels of detail and smoothly transition between detail levels to provide a believable experience. Often this involves finding a way to both store and generate the large amounts of geometry that represent the environment.

In this thesis we present a system for generating and rendering large exterior environments, with a focus on terrain and vegetation. We use a region-based procedural generation algorithm to create environments of varying types. This algorithm produces content that can be rendered at multiple levels of detail. The terrain is rendered volumetrically to support caves, overhangs, and cliffs, but is also rendered using heightmaps to allow for large view distances. Vegetation is implemented using procedurally generated meshes and impostors. The volumetric terrain is editable in real time, which limits our ability to pre-generate or cache large amounts of geometry, and also limits the number of assumptions we can make with regard to visibility.

We support a view distance of at least 25 miles in each direction, though distant objects are rendered at low resolution. The heightmap terrain used to achieve this view distance consists of over 360,000 triangles. Our system runs at 180 frames per second on commodity desktop hardware.

ACKNOWLEDGMENTS

Thanks to:

- Andrew Guenther, for uploading this template.
- My advisor, Dr. Wood, for listening to me talk about terrain generation for five years, and for all her help in making this thesis happen.
- The members of the graphics thesis group, for their insight and support: Lana Hodzic, Katie Davis, Audrey Waschura, Cody Thompson, and Kyle Piddington.
- My family for their love and encouragement: Linda, Terry, Trevor, and Mackenzie.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

Many of the top selling video games of all time are "open-world" games. *Minecraft*, *Grand Theft Auto V*, and *The Elder Scrolls V: Skyrim* are all good examples of this genre. One of their defining characteristics is that they offer players an expansive world to play and explore in. Those three titles alone have a combined total of over 190 million copies sold.

Creating such a world is not a simple task, however. It is not straightforward to write a game engine that can render rolling hills, towering mountains, and dense forests. Nor is it simple to create these exterior environments, either through the careful work of artists or the expensive application of procedural generation techniques.

1.1 A World Of Geometry

The primary source of difficulty in this field is the tremendous scale. Landscape scenes require incredible amounts of geometry to represent everything from the terrain itself to the trees and water. It is not be possible to store the entirety of the visible scene in a modern computer's memory.

Game developers attempting to render an open-world game must use techniques that not only load in different parts of the environment as they are needed, but also load those parts at differing detail levels.

1.2 Relic Engine

This paper describes Relic, a game engine designed to render large exterior environments. The Relic engine procedurally generates and renders terrain, vegetation, water, and atmospheric effects with a low-poly, untextured art style.

For terrain generation, we implement a system for picking between a variety of landscape generators using noise algorithms. This system determines not only the elevation and color of the terrain, but also parameters such as forestation and material properties. The generated content that can be produced and rendered at multiple levels of detail.

Relic renders terrain using a heightmap-based implementation for distant terrain and a novel voxel implementation for nearby terrain. The distant terrain system supports large view distances, up to 3000 miles in all directions. The use of a volumetric representation allows the terrain to support caves, overhangs, and cliffs, features which cannot be represented by a heightmap renderer. The volumetric terrain is also editable in real time.

Forests are rendered using a combination of three different rendering approaches. Terrain, forests, and other scene elements including the sky and water render with 25 mile view distance at a 2560x1440 resolution at 180 frames per second. See Figure 1.1 for an overview of the Relic landscape system.

Relic is written in just over 15,000 lines of C++ code and uses OpenGL to interface with graphics hardware.

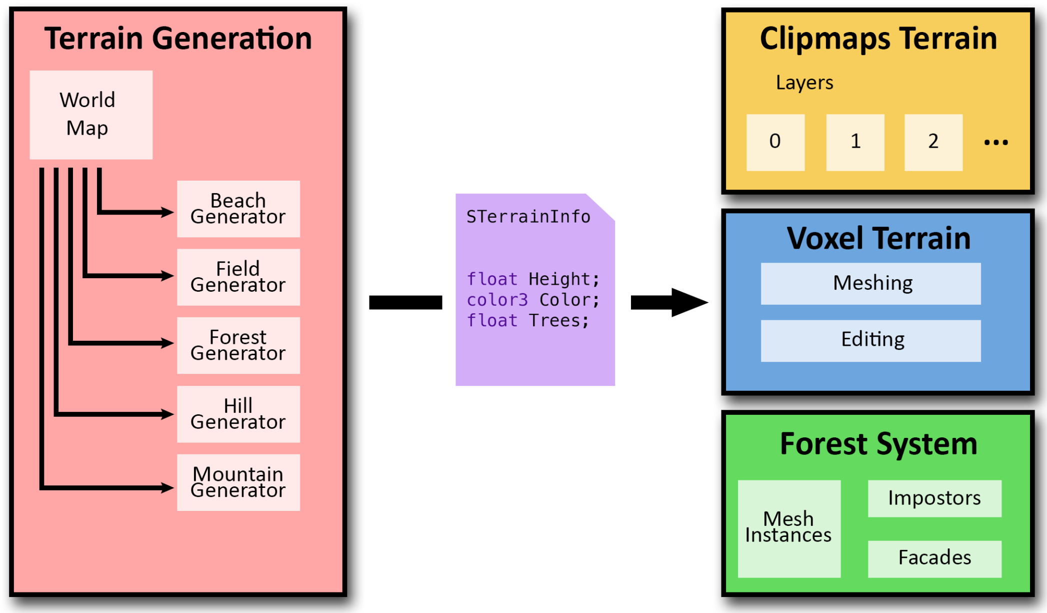

Figure 1.1: Overview of the presented system

1.3 Outline

In Chapter 2 we discuss some of the terrain rendering and generation techniques that form the basis for our system. Then, in Chapter 3 we compare some similar systems to the capabilities of Relic. In Chapters 4 and 5 we describe the contributions of this thesis. Chapter 6 contains the performance validation of each rendering system, and Chapter 7 outlines potential future improvements for the engine.

Chapter 2

BACKGROUND

Our system builds on a lot of existing technology. To lay the foundation of the thesis we first discuss some computer graphics fundamentals (Section 2.1). We then describe level-of-detail algorithms, a fundamental basis for most of the work in the system (Section 2.2). Next, we discuss some methods for generating and rendering terrain (Sections 2.4 and 2.3 respectively), including a look at voxel terrain in particular (Section 2.5). Finally, we give an overview of a rendering effect used to improve the appearance of our renders (Section 2.6).

2.1 The Many Els

Many computer graphics techniques involve breaking down complex objects into a large number of individual elements. As such, we need terminology to refer to these different types of elements.

Perhaps the most commonly known individual element is a pixel. The term pixel refers to a "**Pi**cture **El**ement". In this case, a picture has been broken down into its individual elements, pixels. In this paper, we will discuss pixels and other such elements including:

- Texels "**Tex**ture **El**ements"
- Surfels "**Surf**ace **El**ements"
- Voxels "**Vo**lume **El**ements"

2.2 Level of Detail

There are several limitations when it comes to rendering large scenes. In general, the large amount of geometry required to represent these scenes is at odds with the limited resources of the computer. The GPU is only able to render and shade a certain amount of geometry due to processor speed and number of available processors. Scene geometry must be loaded into main memory, then transferred to graphics memory, both of which have throughput maximums and storage maximums. The hardware interface also has some CPU overhead with regard to submitting draw commands, so we are limited in the number of draw calls we can make.

As such, in order to render large scenes with a large number of scene elements, we need to account for level of detail. 'Detail' in this context refers generally to the amount of geometry used to represent a scene element. Consider, for example, a tree. A high-detail representation of a tree would consist of geometry to represent branches, individual leaves, and the shape of the trunk. A low-detail representation of a tree might simply consist of a trunk-colored box and a leaf-colored cone. See Figure 2.1 for an example of high and low detail representations of a tree from our engine.

Smaller details like branches and individual leaves are no longer visible in the low-detail representation, but are represented more abstractly by the box and cone shapes. The advantage of the low-detail representation is that it requires fewer resources to draw.

In addition, it is also a bad idea to render distant scene elements at full-detail even if resources are available. The farther away a scene element is from the viewpoint, the smaller it is in terms of screen-space. Drawing primitives that are smaller in terms of screen-space can cause aliasing, a visual artifact. As such, low-detail representa-

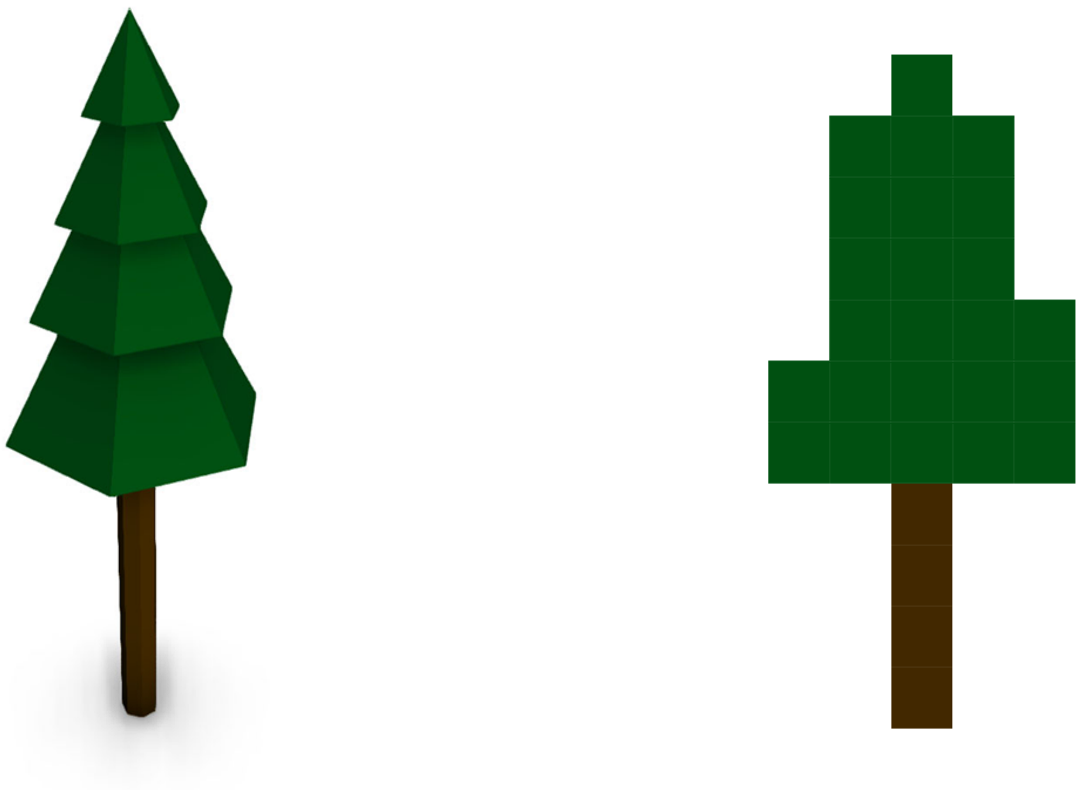

Figure 2.1: On the left is an example of a high-detail tree, on the right is a low-detail tree. While the low detail approximation may appear very crude, it is a reasonable substitution when the tree is so distant that it takes up only a few pixels of screenspace.

tions can sometimes appear better than their high-detail counterparts at large view distances.

Level-of-detail is commonly abbreviated LOD. A rendering system that is capable of representing scene elements in both high-detail and low-detail formats is referred to as a LOD rendering system. A LOD system must switch freely between these representations as the viewpoint changes to reduce the amount of geometry and draw calls required to render a large scene, while still representing nearby objects at full detail.

One common LOD system that is implemented in most graphics hardware is texture mipmapping [34].

2.3 Terrain Rendering

Terrain rendering is a prime candidate for using a level-of-detail algorithm because terrain consists of a large amount of geometry and is visible both up close and at large view distances. Over the years there have been many published techniques for terrain rendering [16] [4] [7] [12] [15] [33] [18] [10]. However, we will focus on some more modern techniques that serve as the basis for the terrain system presented in this paper.

2.3.1 Heightmaps

A common technique for simplifying terrain representation is the use of a heightmap. A heightmap is based on the assumption that terrain elevation is uniformly sampled, i.e. we know the height of the terrain at evenly spaced intervals. Instead of storing a set of 3D points to represent the terrain surface, we can instead create a generic 2D regular grid and store the height separately. This separate array of height values is

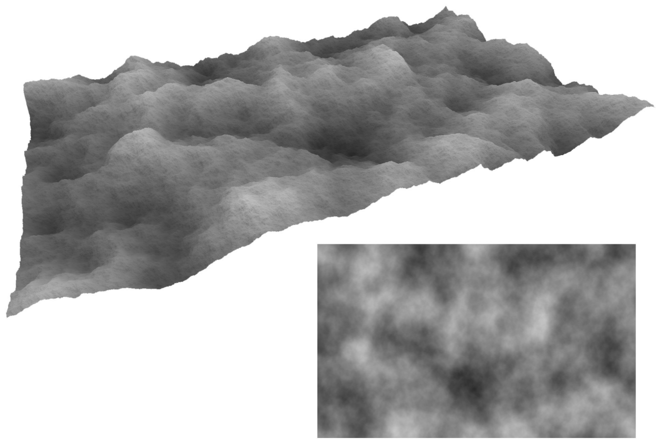
Figure 2.2: An example of a heightmap (below) and its respective terrain (above).

referred to as a heightmap. See Figure 2.2 for an example of a heightmap.

2.3.2 Geometry Clipmaps

Geometry clipmaps is a terrain rendering technique from 2004 that uses a series of nested regular grids to represent each detail level of the terrain [20]. A regular grid of terrain at maximum detail is centered around the viewpoint. Another regular grid of terrain at twice the scale (and thus half resolution) is centered around that inner grid, but with a hole cut out where the inner grid is located. Thus the second grid is clipped by the first grid (i.e. a clipmap). Consecutive grids are added in the same manner, each twice the scale and half the resolution of the previous. Each of these grids is referred to as a layer. The geometry clipmaps approach also includes a system for blending the low and high detail representations at layer boundaries, and for updating the heightmaps used by each layer. Heightmaps are updated toroidally so that small incremental updates are possible as the viewpoint moves. See Figures 2.3 and 2.4.

More recently, work has been done on improving the geometry clipmaps technique

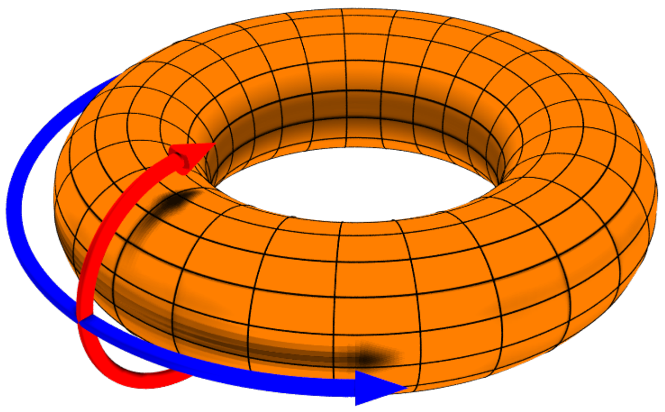

Figure 2.3: An interesting property of a toroid is that moving along either direction of gridlines will eventually lead you back to your starting point [9]. This concept is used to perform incremental updates to the heightmaps in geometry clipmaps.

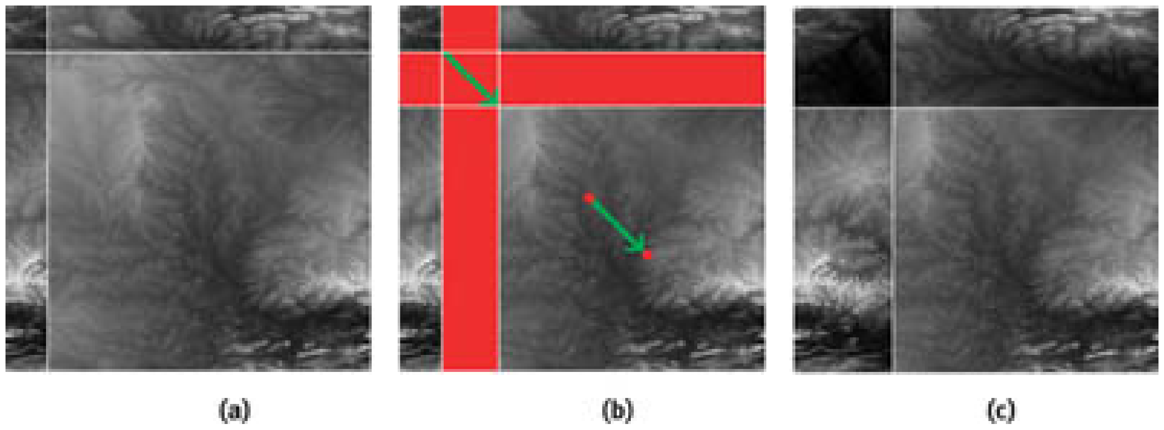

Figure 2.4: An example of a toroidal update for a geometry clipmaps heightmap [20]. Image (b) shows how the movement of the viewpoint causes the red region to be updated.

for use with modern rendering hardware, but the principles remain the same [1].

The system in this paper uses a modified version of geometry clipmaps, which is discussed in Section 5.2.

2.4 Terrain Generation

Before addressing the problem of how to render terrain we must first decide where our terrain data will come from.

There are many possible sources of terrain data. Acquired data from the real world is often available. The USGS provides satellite data for much of the world's surface. Some game engines also use terrain editing tools that allow artists to sculpt terrain manually.

It is also possible to implement algorithms that procedurally construct terrain. The advantage of using procedurally generated terrain is that it requires less time and effort than artistically creating a terrain, and it does not require much or any disk space (as opposed to using acquired data which can be very large).

Procedural terrain generation methodologies usually employ one or more of the follow techniques.

- Physical process simulations
- Fractal processes
- Noise algorithms

2.4.1 Physical process simulations

Real terrain is shaped by erosion, the gradual shaping of landscape caused by water, wind, and other forces. Some terrain generation techniques simulate these physical processes on a starter dataset to create realistic surfaces [14].

However, the nature of the physical simulation usually requires the entire world (or at least individual continents) to be generated at once. This means that the generation of very large worlds can be very expensive in terms of both processing time and memory requirements.

Physical process simulations are often used in fields where time and memory requirements are not as stringent, e.g. offline rendering for movies.

2.4.2 Fractal processes

Another technique for generating terrain is a fractal process, some algorithm that operates on geometry to add detail and which can be re-applied at smaller and smaller scale until a highly detailed surface results. One common form of fractal process is the Diamond-square Algorithm which sequentially subdivides and modulates a regular grid [22].

2.4.3 Noise Algorithms

Noise algorithms are similar to fractal processes but do not need to operate on a regular grid of fixed size [28]. The most typical way that noise algorithms are used is by generating a patch or formula for noise (typically in two dimensions, though sometimes more) and employing a technique called fractional Brownian motion to create a heightmap. The primary advantage of noise terrain generation over the previous two methods is that the generation of an individual elevation value is entirely independent of neighboring values. This means that a noise algorithm can be more easily parallelized, and applied more effectively for use with a LOD terrain rendering system.

Noise is generally a signal that varies randomly. Coherent noise is a special brand of noise that is more useful for generation purposes. As a random signal, it is reasonable to expect that a large change in domain results in a random change in output for a given noise function. Coherent noise has the property that for small changes in domain, only a small change in output will result.

Coherent Noise

There are two major types of coherent noise.

The most simple is value noise. Value noise is generated by calculating random numbers at fixed intervals and interpolating between the values.

Another type of coherent noise is gradient noise. Gradient noise is generated by calculating vectors at a fixed interval, then performing a dot product to calculate intermediary values instead of interpolating. It is a numerically similar process to value noise, but produces noise with more variance - that is, more detail in higher frequencies.

Fractal Noise

Fractal noise, or $1/f$ noise, is a technique for taking a noise function and adding detail at higher frequencies [28]. The general approach is:

1. Take a noise function as input
2. Double the frequency and halve the amplitude of the noise function, then add it to itself. This is called the second octave.
3. Double again the frequency and halve again the amplitude of the noise function, then add this too into the sum. This is the third octave.

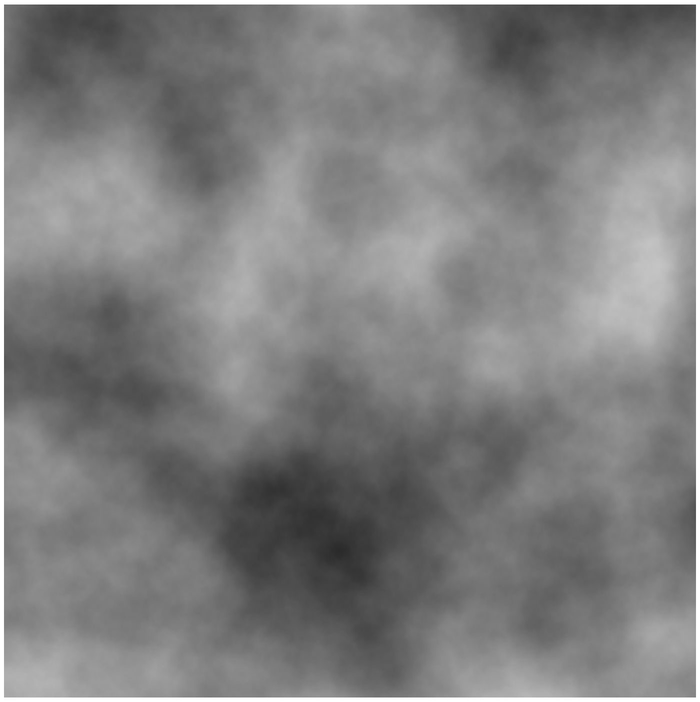

Figure 2.5: Fractal noise render

4. Repeat Step 3 for subsequent octaves until amplitude is close enough to zero that added octaves produce no change in the output, or until desired amount of detail is reached.

In general you do not need to exactly double the frequency or halve the amplitude each octave, but can fine tune these values e.g. by multiplying the frequency by 1.2 and the amplitude by 0.3. See Figure 2.5 for an example of fractal noise.

Ridged Noise

An additional step can be added before summing each individual octave. By first taking the negated absolute value of each generated noise value, a ridged version of fractional brownian motion can be created. This technique is commonly referred to as ridged fractal generator, or sometimes a turbulence generator [28]. In addition, some ridged fractal implementations use the value from each octave to scale the value

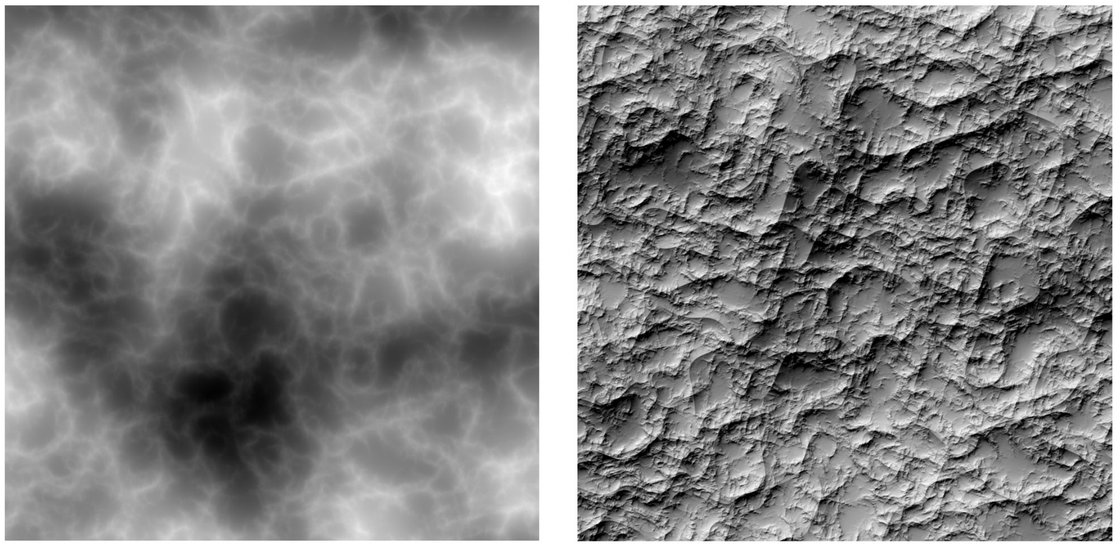

Figure 2.6: Ridged fractal noise render on the left, with a shaded render of the same noise pattern on the right.

from the next octave, to further emphasize the shape of ridge lines [2]. We use this approach in our ridged noise generator. See Figure 2.6 for an example of ridged noise.

Noise Distortion

Ridged noise works on by modifying the range of each noise function, but it is also possible to modify the domain in a similar fashion. Using two additional fractal generators, an offset vector can be produced for each sample of the height generator. This technique helps break up the homogeneous appearance of generated terrain by shrinking and expanding different areas. See Figure 2.7 for an example of distorted noise.

2.5 Voxels

A voxel is a three-dimensional analogue to a pixel. While an image can be represented by a 2D array of pixels, a volume can be represented by a 3D array of voxels - see

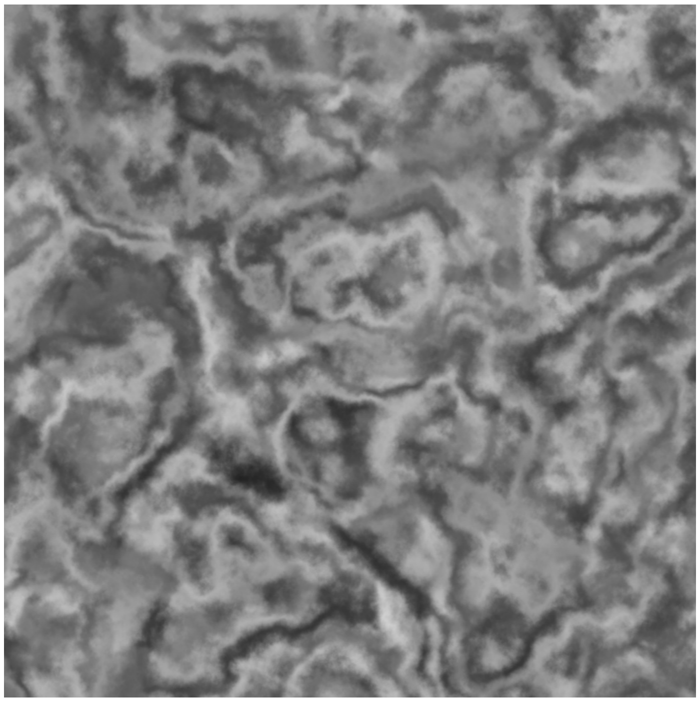

Figure 2.7: Distorted fractal noise render

Figure 2.8.

Voxels are sometimes used to represent and render 3D terrain. *Minecraft* is perhaps the most popular example of this technique, where the terrain and other objects are represented as voxels.

Voxels can be either converted to a polygon mesh for rendering (this is the technique used by *Minecraft*, or rendered more directly using ray-casting or other similar approaches. In Section 5.1 we discuss our approach to using voxels to represent and render terrain.

2.6 Screen-space Ambient Occlusion

Without shadows, it is sometimes difficult to understand the shape and spatial orientation of rendered objects. One way to add simple shadows to a scene is by accounting for the occlusion of ambient light caused by nearby objects.

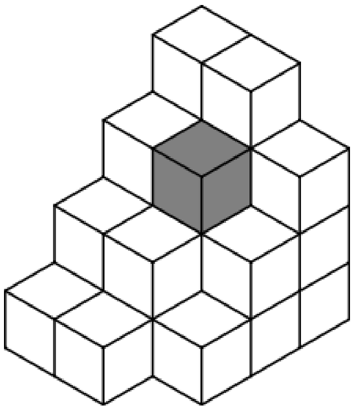

Figure 2.8: Voxels [8]

2.6.1 Ambient Occlusion

In rendering, ambient light is used to represent the indirect lighting of a scene caused by the natural reflection of light off of all objects. While direct light travels directly from the light to the object being shaded, ambient light comes from all directions. Ambient Occlusion is a technique for simulating this global illumination phenomenon by darkening surfaces whenever they are surrounded by many other nearby surfaces. In more technical terms, if a hemisphere aligned along the normal of a surfel contains many other surfel, that surfel is shaded darker than a surfel which is not surrounded.

2.6.2 Screen-space Ambient Occlusion

One simple and efficient way to approximate ambient occlusion in real time computer graphics is to use a technique called "Screen-space Ambient Occlusion" [32]. SSAO is a post-processing pass that operates on fragments, or pixels that have depth values associated with them. Instead of searching a hemisphere around each surfel in the scene, we instead search a hemisphere around each fragment looking for other fragments.

While this approach does not accurately account for any non-visible surface (including any object that is either occluded or off-screen), it is fast and straightforward to add to an existing rendering engine.

2.6.3 HBAO+

Horizon-based ambient occlusion (plus) is an improved and modern version of SSAO developed by Nvidia [25]. Nvidia provides HBAO+ as free software to game engine developers. As such, we choose to use HBAO+ in our engine instead of writing our own implementation of SSAO.

Chapter 3

RELATED WORKS

3.1 Terrain Generation

Some recent publications in the field of terrain generation have produced impressive landscapes. Génevaux et al. used a vector-based hydrology simulation with user-controllable parameters [14]. One interesting aspect to note is their novel approach to storing the terrain which focuses on a hierarchy of features, similar to constructive solid geometry. Zhou et al. implemented a system to stitch together samples from real-world DEM data, also with user-controllable parameters [35].

By comparison, the generation system Relic uses is much simpler, but it allows height values to be easily generated at arbitrary resolution.

3.2 Tree Rendering

Many published techniques for rendering forests involve ray casting or ray marching. Mantler et al. add tree rendering to a terrain renderer by ray marching on an appended tree heightmap [21].

Bruenton & Neyret use pre-rendered depth maps of tree models from many possible angles in addition to an algorithm for distant forest shading [5]. Pre-rendered depth maps are a more complex version of impostor rendering. The low-poly trees of Relic are more sufficiently represented by an impostor than photo-realistic trees, and impostors are far less intensive to both generate and render than depth maps.

3.3 Minecraft

Minecraft is an open-world game that uses voxels for representing and rendering terrain [29]. The voxel representation makes it possible to support caves and overhangs, as well as real-time editing. However, all terrain is represented at the same detail level, making it difficult to support large view distances. Additionally, *Minecraft* has a vertical range limited to 256 meters [23].

3.4 Voxels and Volumetric Terrain

Marching cubes is an algorithm for creating a polygonal mesh from volumetric data [19]. Unlike the voxel meshing used by *Minecraft*, Marching cubes can be used to create smooth surfaces because grid values are scalar distances, not boolean occupancy values. Relic's voxel algorithm operates on a boolean grid (which is more friendly for problems like in-game editing) but can still produce slanted surfaces.

The Transvoxel Algorithm is a technique for stitching together volumetric meshes so that level-of-detail techniques can be applied to volumetric terrain [17]. Relic avoids the need to apply LOD to volumetric terrain by using a heightmap representation for distant terrain.

Chapter 4

GENERATION

Our terrain generation algorithm creates a world with large continents that contain beaches, forests, hills, and mountains. The algorithm uses a low-frequency world map to determine continent outlines and region boundaries. For finely sampled terrain values, the values of the world map are used to select and influence custom generators for each region type. See Figures 4.1 and 4.2. The world map is discussed in Section 4.1 and the different region generators are discussed in 4.2

4.1 World Map

The terrain is generated by first establishing a low-frequency world map, then selecting from a series of different landscape generators based on general elevation. The world map uses a single distorted fractal noise layer with a box ramp to force oceans at all borders.

A low number of octaves is used (four, in our implementation) to guarantee a lack of high-frequency details. This makes it useful for generating features without having areas that swap between multiple region types unrealistically. Figure 4.2 shows an example of the smooth isolines generated by the world map.

4.2 Regions

For rendering, the terrain values need to be generated at 1 foot postings. We also need color values to shade the ground, and a forestation value to determine how to place trees.

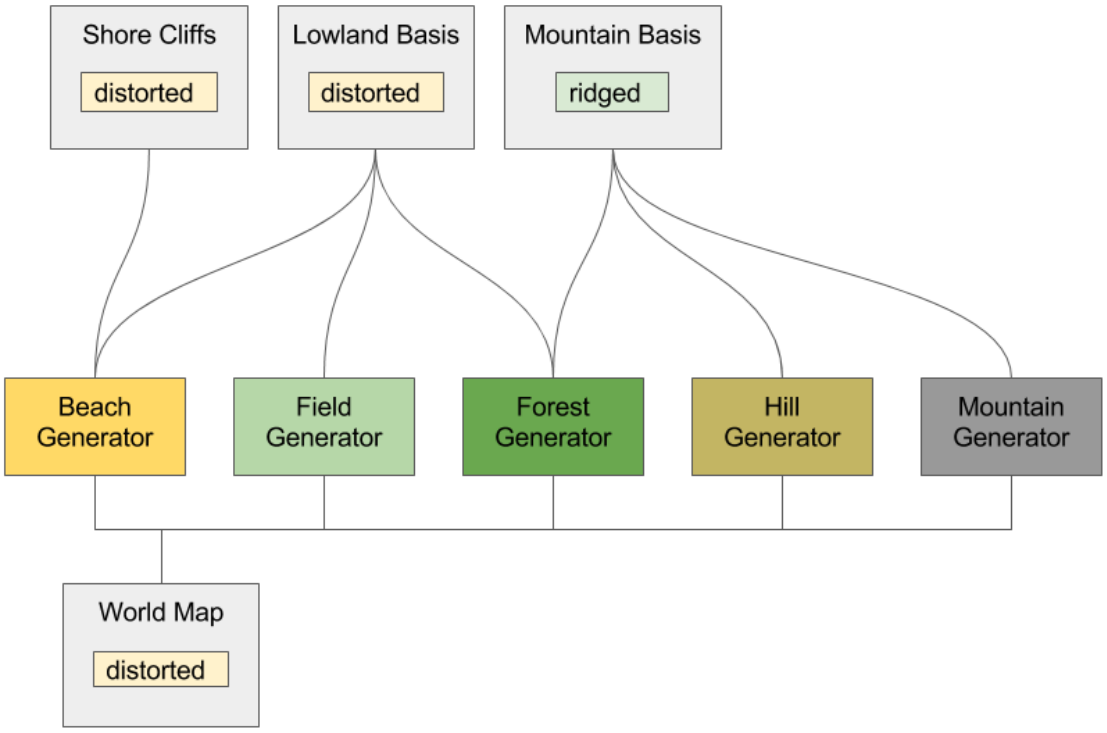

Figure 4.1: Terrain generation system overview, showing the type of noise generators used.

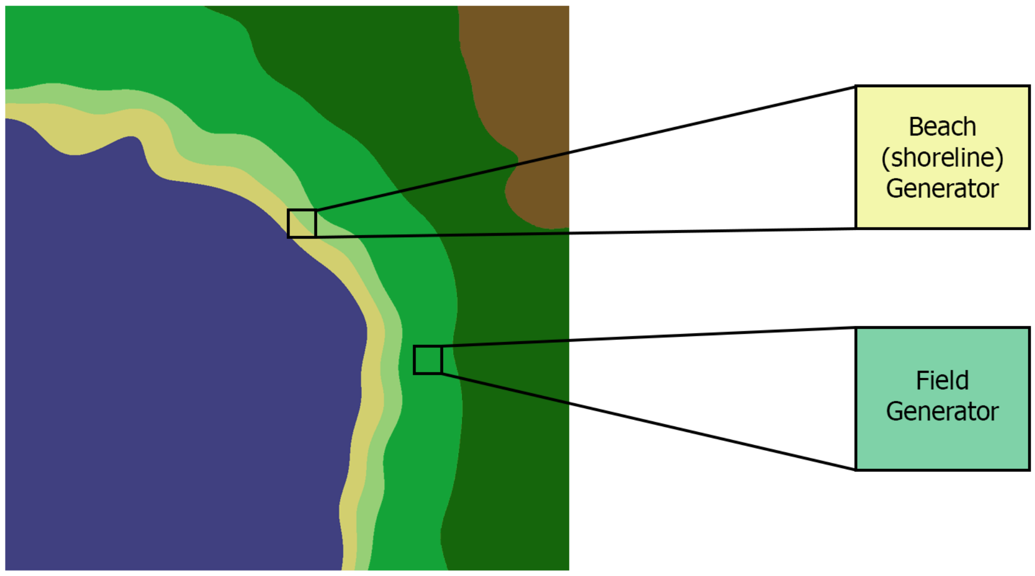

Figure 4.2: Example world map section and generator selection.

Region	Minimum	Maximum
Ocean	-1	0
Shore	0	0.05
Field	0.05	0.15
Forest	0.15	0.30
Hills	0.30	0.45
Mountain	0.45	1.0

Table 4.1: Region boundaries from the world map values.

Values from the world map are used to pick a particular set of terrain generation rules, referred to as a region. The return value from the world map generator has a theoretical range from -1 to 1, but most values tend to be between -0.7 and 0.7. This range is divided into regions with low values mapping to low elevation regions and high values mapping to high elevation regions. See Table 4.1 for the exact world map values used to determine regions.

The world map can usually be used to generate color values, i.e. the color from the image in 4.2 is used by each region generator. However, the Mountains generator in Section 4.2.3 does some additional coloring.

For each region, a "region delta" is calculated to indicate how far into a region the engine is currently generating data for. A region delta of 0.0 indicates the boundary with the next lower elevation region, 1.0 indicates the higher boundary, and 0.5 indicates the middle of the region.

The region types used by the engine are: Oceans, Beaches, Fields, Forests, Hills, and Mountains.

While each region generator is individually responsible for generating elevation values, some noise sources are shared between the different regions so that smooth transitions can be generated. As an example, the Oceans, Fields, and Forests generators all share a distorted fractal noise source.

4.2.1 Oceans and Fields

The oceans, fields, and forests generators all simply add some high-frequency noise to the world map value with some scaling. This generates small sandy hills below water for Oceans, small grassy hills for Fields, and tree-covered hills for Forests.

4.2.2 Beaches

Beaches serve as a transition between oceans and fields. Beaches may contain a cliff partway between the shoreline and the transition to the field region.

This cliff is generated by applying a vertical offset to any baseline value above a certain threshold. The offset is applied gradually over a small area so that the cliffs are not completely abrupt, but have a horizontal dimension of a few feet. As the baseline elevation approaches the transition to Fields, the offset is tapered off to give cliffs some additional prominence. See Figure 4.3.

The cliff generation offset function is as follows, where v is the map value:

$$shoreline(v) = \left\{ \begin{array}{ll} 0, & \text{for } 0 \leq x \leq \text{cliff\_start} \\ \text{cliff\_size} * \text{cliff\_falloff} * \text{cliff\_gain}, & \text{for cliff\_start} < x \leq 1 \end{array} \right\}$$

Where cliff_gain and cliff_falloff are both computed as follows, but clamped from 0 to 1:

$$\text{cliff\_gain} = \frac{v - \text{cliff\_start}}{\text{cliff\_range}}$$

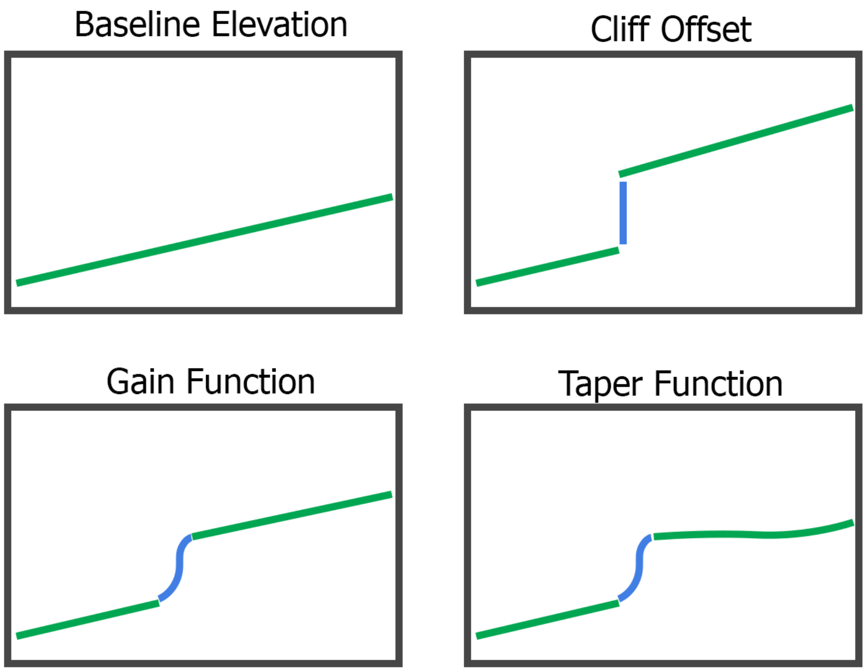

Figure 4.3: Beach cliff generation

$$\text{cliff\_falloff} = \frac{x - (\text{cliff\_start} + \text{cliff\_range})}{\text{falloff\_range}}$$

The exact size, shape, and shoreline distance of the cliffs are configured by the parameters cliff_start, cliff_range, cliff_size, and falloff_range. Relic uses additional noise layers to add variance to these parameters, and adds applies a gain function to both cliff_gain and cliff_falloff to create a smoother final appearance.

4.2.3 Forests, Hills, and Mountains

The Forests, Hills, and Mountains regions both use a custom version of a ridged fractal generator.

For the first two octaves of noise, a standard gradient noise sample is used instead of the ridged version. This helps reduce the sharpness of some peaks and hides some unnatural artifacts that sometimes occur. Figure 4.4 shows the first few octaves of

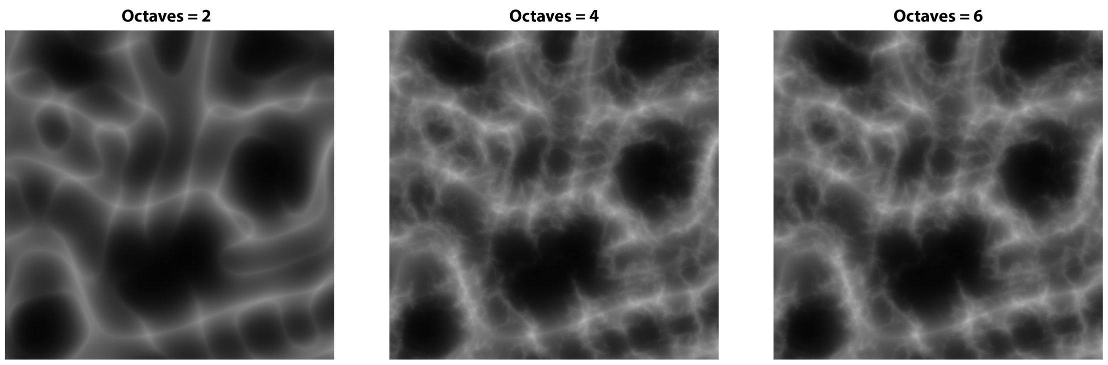

Figure 4.4: Two, Four, and Six octaves of a standard ridged fractal generator without our modification of using a standard fractal noise source for the first two octaves.

a ridged fractal generator without this modifications. The ridge lines established by the second octave have a large effect on the resulting image even as more octaves are added. The smoothness of these lines creates circular artifacts in the resulting terrain. Figure 4.5 shows the first few octaves of a ridged fractal generator with this modification. The effect is similar to lowering the amplitude and increasing the frequency of a normal ridged fractal generator but with large-scale variation preserved.

In addition, a hilliness parameter is added to help smoothly transition between mountains and hills. When hilliness is 0, the standard absolute value is used. At hilliness of 1, the input value is squared instead to produce a parabolic shape. Hilliness values between 0 and 1 interpolate between these two shapes.

At the low edge of Hill regions, a hilliness value of 1 is used along with low overall amplitude. These same parameters are used at the high edge of the Forest region, and create smooth rolling hills of medium height. In the Mountain region, hilliness falls to 0 and amplitude increases to create sharp ridge lines and towering peaks.

Mountain regions are colored by adding snow coverage over the region color specified by the world map. Snow is added whenever the angle between the normal of the surface and the up vector is smaller than some value. For high elevations, a large

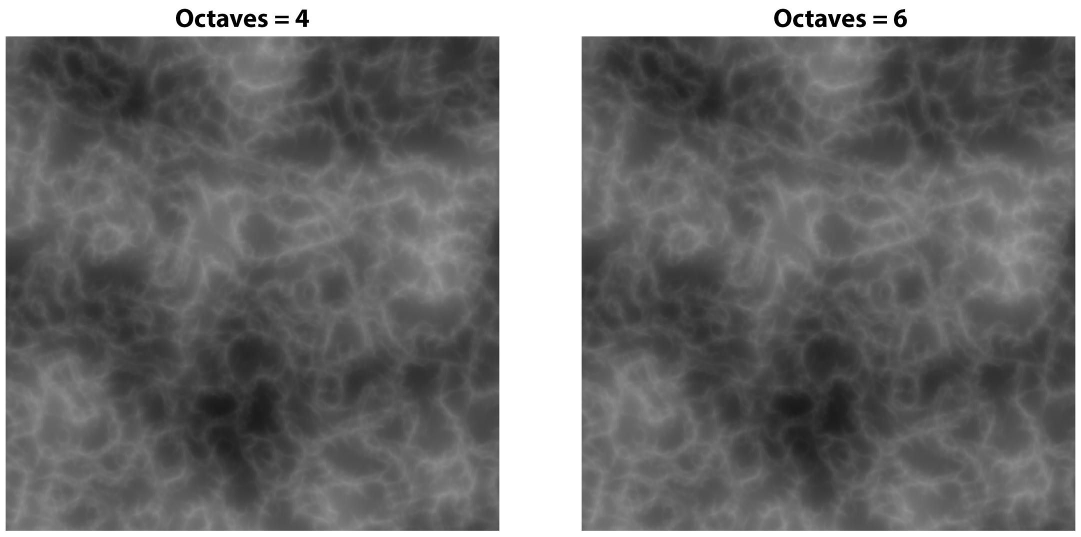

Figure 4.5: Four and Six octaves of the ridged fractal generator used by Relic, where a standard fractal noise source is used for the first two octaves.

angle is used, and at low elevations a small angle is used.

4.3 System Overview

The use of a world map generator defines at a large scale what type of geometry to generate, including continent outlines and region boundaries. Values generated by the world map are used to pick a particular region generator and transition smoothly between each region. The regions, in order of elevation, are Oceans, Beaches, Fields, Forests, Hills, and Mountains. Oceans and Fields have simple geometry - just some low amplitude noise to create small hills and variations. Beaches serve as a ramp between Ocean and Field regions, with a cliff generated partway up the shore. Forests, Hills, and Mountains use a custom ridged fractal generator to create ridged mountaintops and rolling hills.

Chapter 5

RENDERING

Our landscape rendering system consists of three primary components - a nearby terrain renderer, a distant terrain renderer, and a forest renderer.

The nearby terrain renderer uses a meshing algorithm influenced by voxel terrain systems (Section 5.1). This mesh generation is expensive but is necessary for meaningful player interaction and is limited to a very narrow region around the viewpoint.

Far terrain is rendered using a modified implementation of geometry clipmaps (Section 5.2). The nature of the clipmaps algorithm makes it possible to extend the terrain view distance substantially with minimal overhead.

Our forest system uses mesh instances, impostors, and a similar representation called facades to render vast forests (Section 5.4). The system is designed to render a large number of individual trees with the highest performance possible.

In addition to these three primary components, our system also renders water and other additional environment aspects, discussed in Sections 5.5 and 5.6.

5.1 Nearby Terrain

Nearby terrain is rendered using a pseudo-voxel representation designed to allow for sloped surfaces while maintaining intuitive live editing capabilities. A voxel algorithm that uses uniform voxel size is simple to implement and manage but is poorly suited to represent sloped surfaces. Fully volumetric terrain implementations allow for arbitrary slopes but are less intuitive for manipulation by players and navigation by AI. We utilize a pseudo-voxel system that allows for diagonal faces in addition to solid

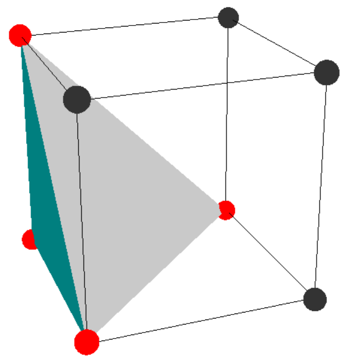

Figure 5.1: Red spheres indicate a full sub-voxel, grey indicates an empty sub-voxel.

voxels. This system allows for simple grid-based editing and simple physics calculations, while allowing sloped polygonal faces that are more aesthetically pleasing than simple voxels.

5.1.1 Voxel algorithm

In principle, the system works by breaking down each individual voxel into eight sub-voxels (one for each corner of the cube) and generating diagonal faces based on which of these sub-voxels are occupied. However, certain sub-voxel configurations are considered degenerate and are automatically trimmed to a lesser configuration. For example, in our system any pseudo-voxel with only a single sub-voxel occupied is automatically trimmed to an empty voxel. In this case the trimming occurs because there is no diagonal face to represent just a single corner of a voxel. Figure 5.1 shows the pseudo-voxel with the fewest possible sub-voxels, four.

The system can therefore trivially trim any pseudo-voxel configuration with fewer than four sub-voxels.

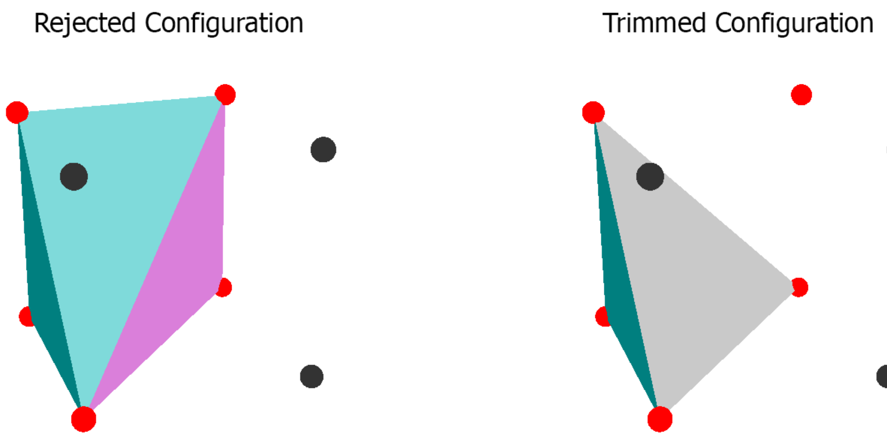

Figure 5.2: Red spheres indicate a full sub-voxel, grey indicates an empty sub-voxel. The image on the left shows a possible triangulation for this sub-voxel configuration. However, our algorithm rejects this configuration and instead generates the triangulation on the right.

The system also trims certain pseudo-voxel configurations to a simpler configuration. See Figure 5.2 for an example.

This trimming is performed to preserve smooth slopes of generated terrain. See Figures 5.3 and 5.4 for a comparison between untrimmed and trimmed sub-voxel configurations.

5.1.2 Face Generation

Triangles are generated for each pseudo-voxel configuration using lookup tables for efficiency and simplicity. The largest lookup table is 12 kilobytes, so the memory impact of using the lookup tables is minimal. Triangles are divided into two categories: interior and exterior.

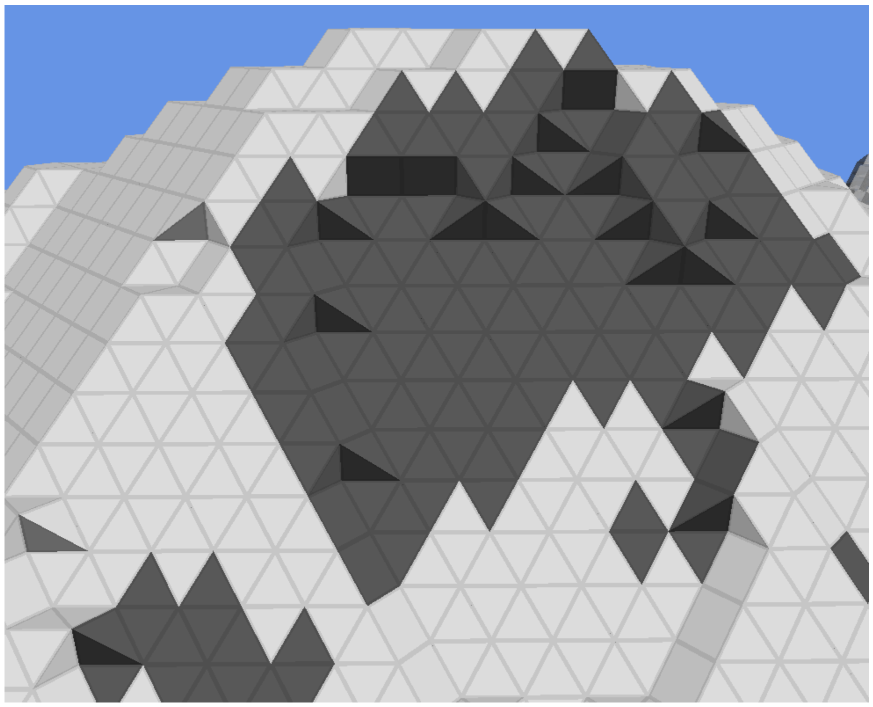

Figure 5.3: Voxel triangulation of a mountaintop without sub-voxel trimming enabled.

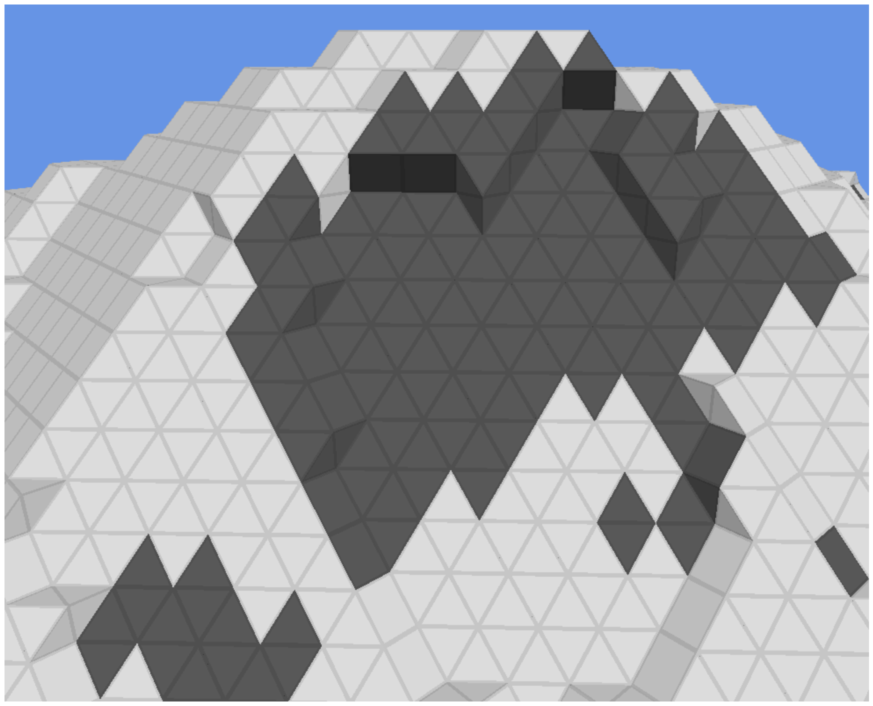

Figure 5.4: Voxel triangulation of a mountaintop with sub-voxel trimming enabled.

Exterior Faces

Exterior triangles are the triangles that would normally be generated by a simple voxel system - the faces of a cube. Exterior triangles are checked against each neighboring pseudo-voxel for visibility. In the trivial case, a completely solid voxel surrounded entirely by solid voxels produces no triangles, since all exterior triangles are occluded by the exterior faces of each neighboring voxel.

Interior faces

Interior faces are the triangles that are unique to Relic's voxel system, as opposed to an ordinary voxel system that only generates exterior faces. Interior faces have some form of diagonal slope see Figure 5.5 for a demonstration of each type of interior triangle that Relic generates.

5.1.3 Rendering

The world is divided into chunks for triangulation and rendering purposes. Each chunk is a 32x32x32 volume of voxels. Relic generates voxels at a resolution of one voxel per foot, so each chunk is also 32x32x32 feet in dimension. The size of voxel chunks is further discussed in Section 6.2.

One triangle mesh is generated per chunk. Each chunk tracks its neighbors in all six face directions so that occluded exterior faces can always be accurately detected. This means that a ring of non-visible chunks must be loaded around all visible chunks, since no visible chunk can have an unloaded neighbor.

In order to determine which chunks are visible, a large sphere is placed around the camera and any chunk that intersects this sphere is loaded and rendered. As the camera moves, chunks that leave this sphere are unloaded and chunks that enter it

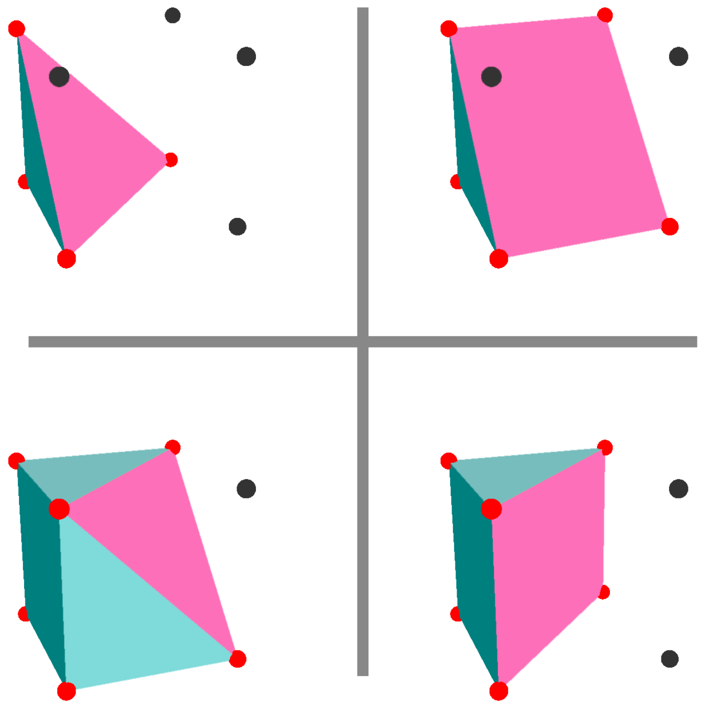

Figure 5.5: Example of the four types of interior face. Each pink face is an interior face. Note that this is just one possible orientation of each class of interior face, but each class can be oriented in either four or eight directions. (The left two classes have eight orientations and the right two have four).

are loaded.

Screen-space ambient occlusion is used to help visual understanding of the voxel terrain shape.

5.2 Far Terrain

The voxel terrain described in the previous section is ideal for editability and representation of features such as overhangs and caves, but it is very expensive to both generate and render. In order to maximize the view distance of the Relic world, far terrain is rendered using a flat-shaded implementation of geometry clipmaps with a few modifications.

5.2.1 Pre-calculated Index Buffer

The original geometry clipmaps implementation recalculates index buffers pre-frame so that each layer can grow and expand organically. This makes it possible to seamlessly transition to lower-resolution terrain when the viewpoint moves rapidly.

However, our system locks the size of each layer so that the index buffers can be pre-calculated. This was found to significantly reduce CPU overhead. This means that while the original geometry clipmaps algorithm would simply show less high-frequency detail when the viewpoint moves rapidly, our system slows down with increased load times when the viewpoint moves rapidly. However, our clipmaps implementation is part of an engine designed for a game, where player movement speed will inherently be limited by a physics simulation. In addition, rapid movement of the viewpoint will require all other systems (voxel terrain, forests, physics, etc.) to stop and load new data. As such, allowing the geometry clipmaps algorithm to continue running in this scenario would not be helpful.

5.2.2 Normal Calculation

Since our terrain is flat-shaded for a polygonal effect, the per-quad normals provided by a normal map must be inaccurately applied to two triangles. We also found that normal calculation from our procedural generation system incurred significant CPU overhead. Accurate normal calculation in transition regions was also difficult.

Our implementation uses a per-triangle normal calculation in a geometry shader so that accurate normals are always calculated for each triangle. This saves significant CPU time as normals don't need to be pre-calculated or sent to the GPU normal map. The addition of the geometry shader was found to not have a significant decrease in rendering performance.

5.3 Terrain Blending

In order to use both the near and far terrain representations, Relic implements a blend region between the two systems. In general, the near terrain is simply rendered on top of the far terrain (ignoring the actual depth of either geometry). However, we implement a blend region between the two representations so that the transition is smoother, and there is an additional caveat that must be addressed.

Consider the side-profile view of both terrain systems shown in 5.6. In the red regions, only the near terrain is visible, so no additional work needs to be done. In the purple regions, both terrains are visible, but our choice of rendering the near terrain on top will ensure that only the near terrain is visible. The blue region is the problem area. From this viewpoint, there is a dip in the near terrain geometry but the smoother far terrain representation cuts through this gap. At this distance we only want to see the near representation. Figure 5.7 shows how this effect appears in-engine. Relic solves this problem by simply culling any fragments from the far

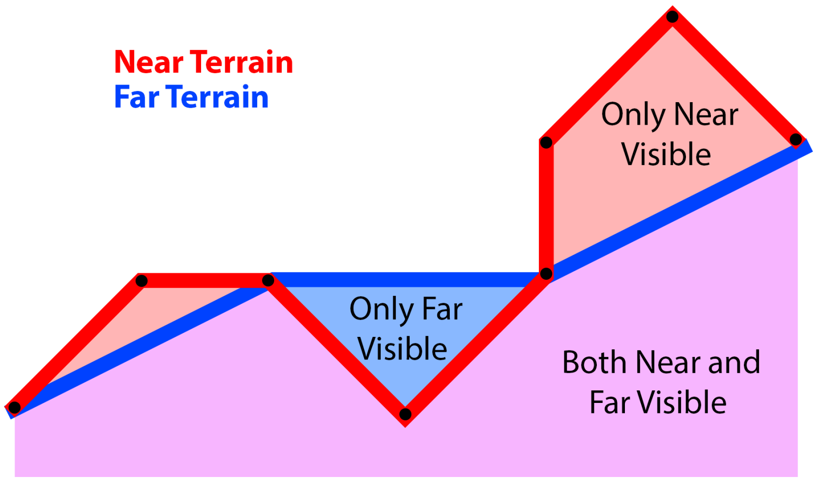

Figure 5.6: In this image the red line represents the shape of the near (voxel) terrain and the blue line represents the shape of the far (clipmaps) terrain. The light blue shaded region is an area where the far terrain might show above the near terrain from this side view. Relic solves this problem by culling all nearby fragments from the far terrain render.

terrain representation that are closer than a given threshold.

Relic also implements a blend between the two terrain representations. This blend is implemented in a post-processing pass that uses separate frame buffers for each terrain representation. The algorithm operates on individual fragments. If both systems are visible at a given fragment, and the distance to the fragment is near the maximum distance of the near terrain representation, a blend of the two systems is

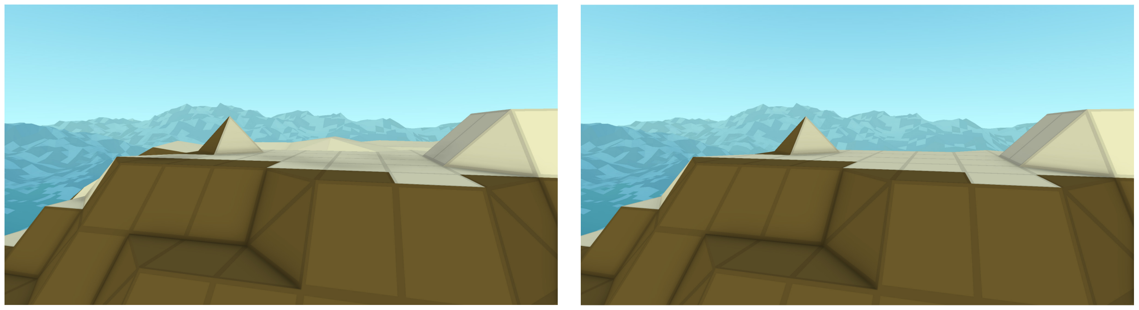

Figure 5.7: Example of the effect described in Figure 5.6 shown in-engine. In the left image, the far terrain geometry is visible above the near terrain. In the right image, it is culled properly.

used. For all other fragments, the near terrain fragment is selected if it is visible, otherwise the far terrain fragment is selected.

5.3.1 Terrain Summary

Terrain is rendered using both a voxel representation and a clipmap representation. The voxel representation supports editing and 3D terrain features such as cliffs, overhangs, and caves. However, the voxel system is costly to generate and render. A geometry clipmaps implementation is used to support a large terrain view distance beyond where the voxel representation ends. The blend pass described in Section 5.3 is used to merge these two terrain representations.

5.4 Forests

Our vegetation system currently supports rendering a large number of low-poly spruce trees, but could be expanded to support other types of vegetation. The system uses three levels of detail: mesh instances, impostors, and mesh facades.

Trees are rendered in chunks similar to the chunk system used by nearby terrain. Each chunk represents an 80x80 foot area of forested terrain. A chunk contains at most 100 tree meshes or impostors, or is represented by a single facade mesh.

5.4.1 Mesh Instances

The mesh instances layer simply uses instance rendering to draw the tree model in several places.

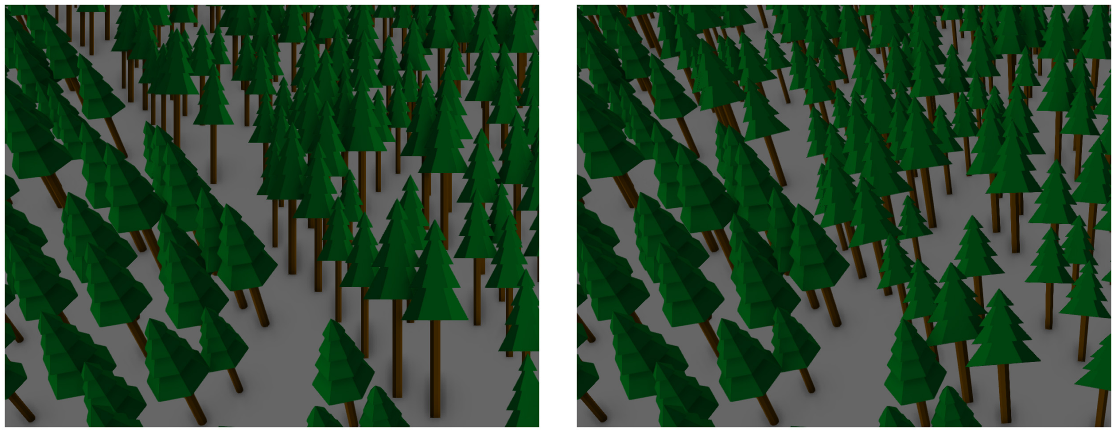

Figure 5.8: In the left image, tree impostors are rendered on spherical billboards. At this elevated viewpoint, the billboard effect is clearly visible when compared to the tree mesh instances in the lower left corner of the image. The right image shows the same scene as rendered by the Relic engine, using cylindrical billboards (where only the Y axis rotates).

5.4.2 Impostors

The impostors layer draws a billboard for each tree in the group. During initialization, the colors and normals of the tree model are rendered to textures for use in drawing each impostor. The impostors are locked in their X and Z axis rotation so that the base of the impostor always rests on the ground, and to reduce artifacts when the camera is high above or below the impostor. See Figure 5.8 for a comparison between the cylindrical billboards we use and the alternative spherical billboard. The rotation around the Y-axis is used to rotate the normals of the impostor so that lighting is accurate for impostors in all directions.

5.4.3 Mesh Facades

The final layer uses a simple mesh to represent a group of trees. The appearance of this mesh is quite simplistic but at large view distances it is sufficient to represent a group of trees. See Figure 5.9 for a comparison between mesh facades up close and

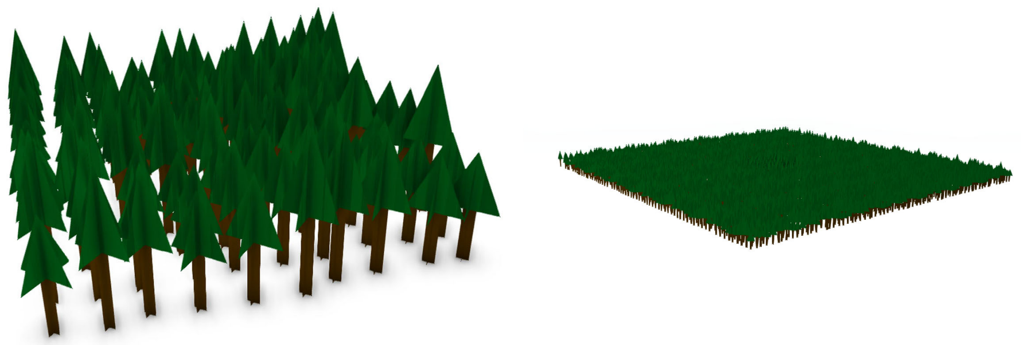

Figure 5.9: On the left, a tree facade is shown in detail. The image on the right is many tree facades at great distance.

Representation	Distance
Mesh Instance	2
Impostor	16
Facade	32

Table 5.1: Tree view distances expressed in chunk widths.

at a distance.

5.4.4 Rendering

Chunks within a certain distance are rendered as meshes. Chunks within a farther distance are rendered as impostors. Finally, any chunk beyond the impostor distance is rendered as a facade.

The system therefore has three configurable tree distance parameters. We found that the values in Table 5.1 worked well. Increasing any view distance reduces the visual artifacts from using a low-detail representation, but decreases performance.

5.5 Water

Our water system uses the geometry of our far terrain (geometry clipmaps) implementation for simplicity.

Instead of offsetting each vertex by a heightmap value, however, we used summed Gerstner waves in the vertex shader. This provides both a vertical offset and a normal to use for rendering.

The result is expensive, but simple and looks reasonable.

5.6 Environment

The system renders a billboard to represent the sun and a sky sphere with vertex colors to represent the sky.

Lighting for all other objects in the scene uses three directional lights: one for direct sunlight, one for scene reflection of sunlight, and one for sky light. The sunlight has high bright white color and points from the sun position towards the scene. The scene reflection light has dim white color and points in the opposite direction of the sunlight, with the Y component clamped to zero. The sky light points directly downward (negative Y) and has soft blue color.

Our system also implements a simple atmospheric scattering simulation by applying fog to each rendered fragment based on scene depth. The fog color interpolates between dark blue for near fragments and the sky color for far fragments.

5.7 Depth Buffer Precision

Our system supports very large view distances. Setting the far plane at sufficient distance for our scene significantly degrades the performance of the depth buffer,

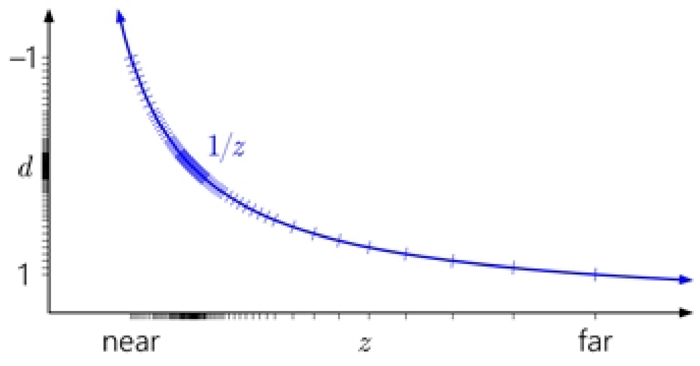

Figure 5.10: Mapping of Z values to depth values [31]. The tick marks on the blue line indicate the precision of the depth buffer.

even with 32 bits of precision. Insufficient precision in the depth buffer results in Z-fighting on distant elements and especially noise in the ambient occlusion effect.

One problem is that the mapping of Z values to depth values has a reciprocal shape, such that most of the depth buffer precision is alloted for nearby fragments (See Figure 5.10). To improve depth buffer performance for far distances, we use the inverted depth buffer trick [31]. By using a DirectX compatibility feature of OpenGL, we can use a depth buffer ranging from zero to one instead of negative one to one. Then, by storing depths reversed from the conventional direction, we can utilize the inherent distribution of floating point precision to even out the distribution of depth values in our scene. The conventional depth direction has near values at zero and far values at one. However, floating point precision is higher for values closer to zero. By storing near depths at one and far depths at zero, the additional floating point precision creates a pseudo-logarithmic distribution of depth values.

This results in a significant reduction in depth precision artifacts.

5.8 Rendering Summary

The primary rendering components of Relic are a nearby terrain renderer, a distant terrain renderer, and a forest renderer.

The nearby terrain renderer uses a voxel representation that is meshed using our novel sub-voxel system. Far terrain is rendered using geometry clipmaps with some modifications.

Relic's forest system uses mesh instances, impostors, and facade meshes to render forests with a large view distance. Finally, Relic includes a system to render water, a sky sphere, and atmospheric scattering.

Chapter 6

RESULTS

We have presented a system for generating and rendering large landscape scenes, supporting mountains, hills, forests, and lakes. This system supports large view distances while simultaneously supporting local in-game editing using a voxel system. As shown in Section 6.1, you can see the worlds generated are visually appealing and run at reasonable rates. Sections 6.2, 6.3, and 6.4 discuss the performance considerations of the voxel, clipmaps, and forests systems respectively.

6.1 Full System Screenshots

The following screenshots were rendered at around 180 frames per second with a 2560x1440 resolution. The test machine was a desktop computer with an Intel i7-5820K 3.3GHz CPU, a Nvidia GTX 980 GPU, and 16 GB of RAM.

Figures 6.1 and 6.2 show a view of a forest region with mountain regions in the distance. Figures 6.3 and 6.4 show a mountain region. Figure 6.5 shows the same view of the previous mountain screenshot with a different sun position. Figure 6.6 shows a lake and figure 6.7 shows a shore cliff.

6.2 Voxels

The size of voxel chunks affects rendering performance and the cost of modification. Voxel size is measured in voxels, which in Relic maps one-to-one with world units, or feet.

Since modifying the terrain requires the entire chunk mesh to be reconstructed,

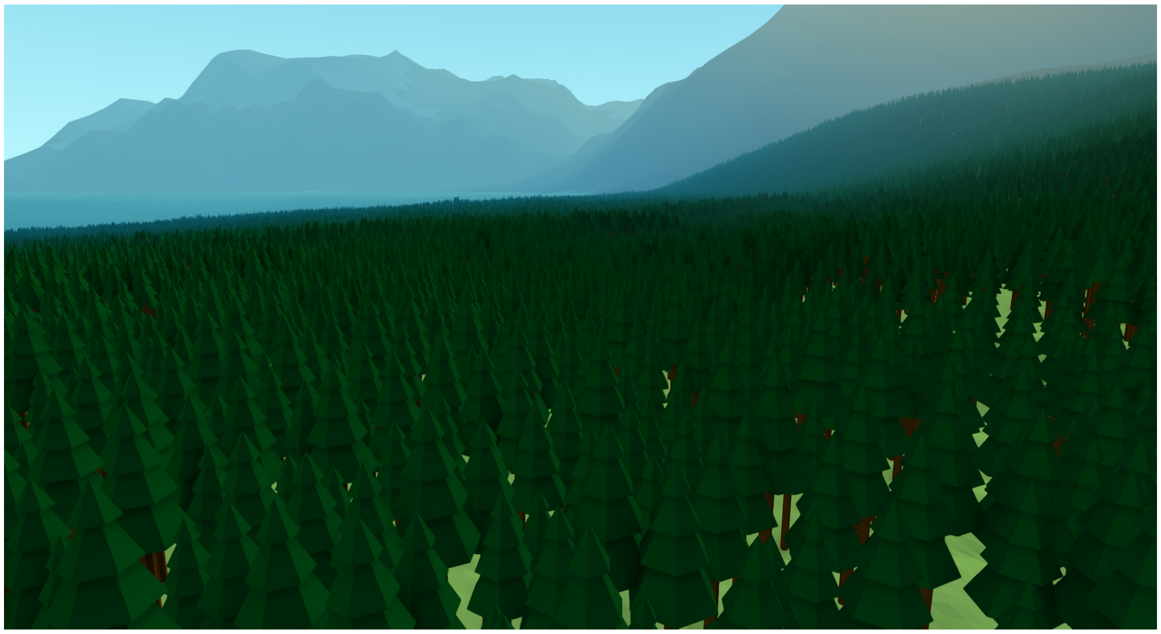

Figure 6.1: Forest region.

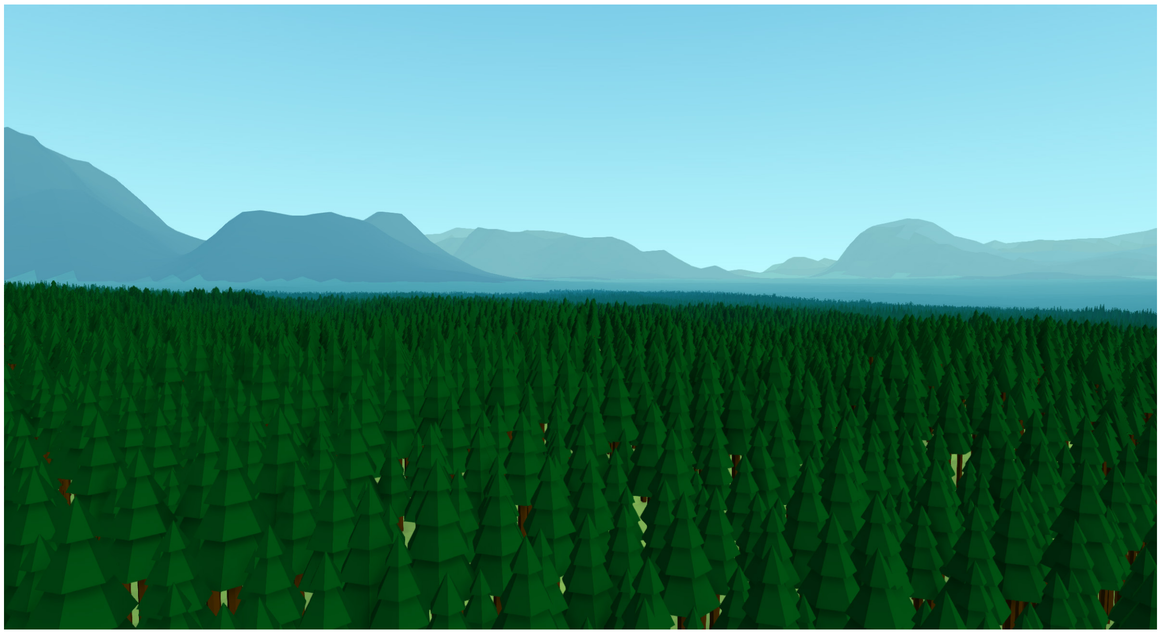

Figure 6.2: Another view of a forest region.

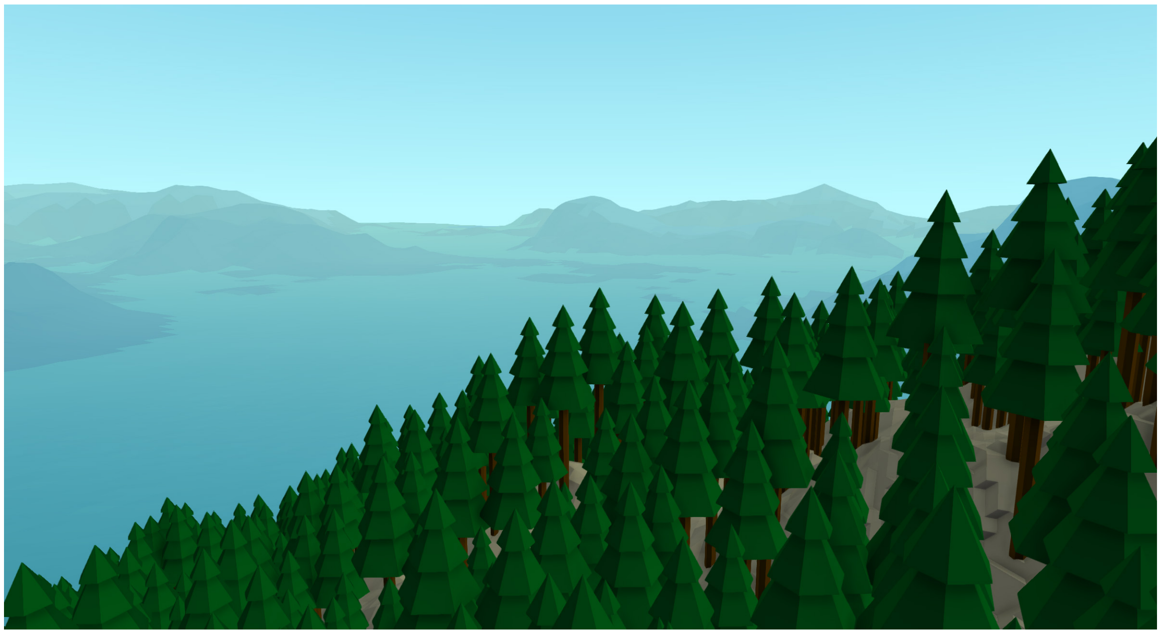

Figure 6.3: Mountain region.

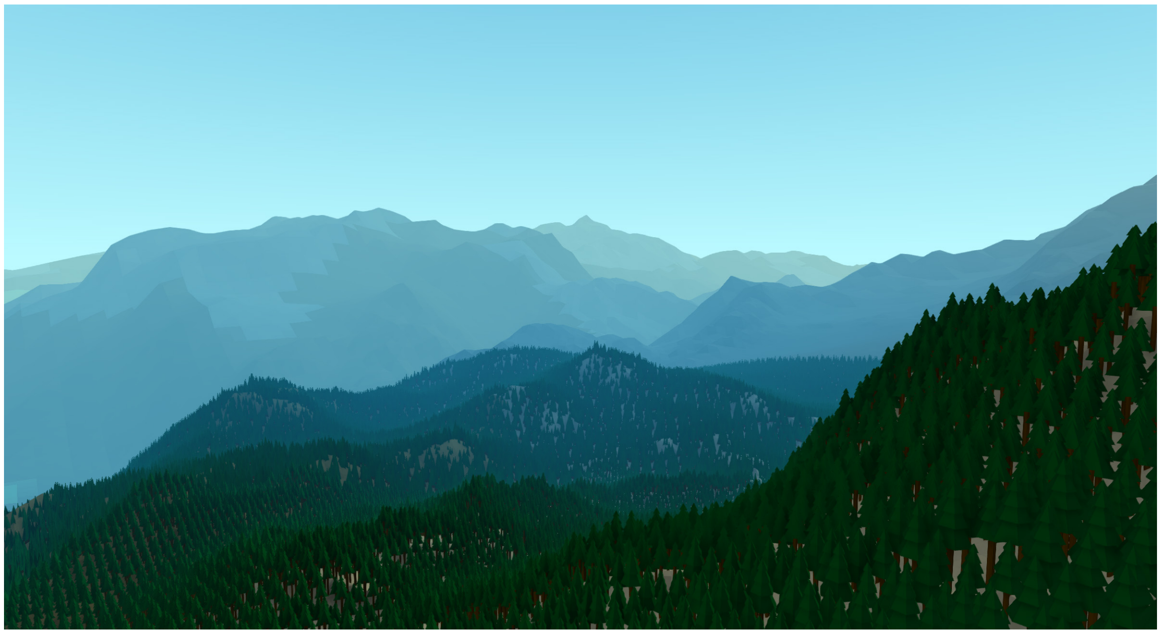

Figure 6.4: Mountain region with distant trees.

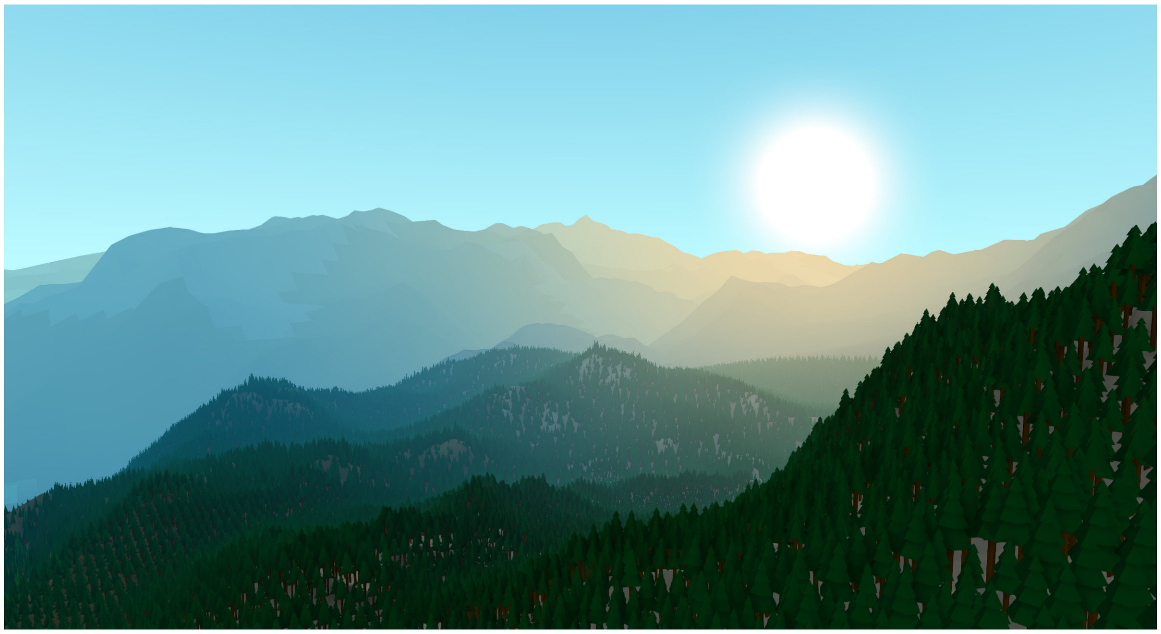

Figure 6.5: Same view as the previous screenshot, with different sun position.

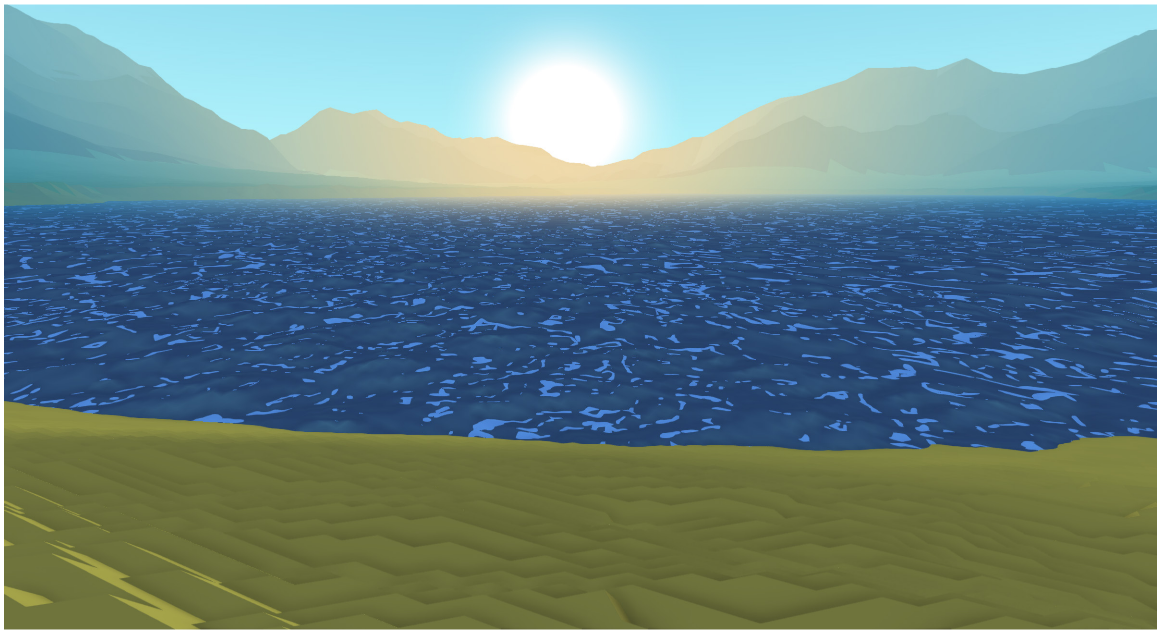

Figure 6.6: A lake.

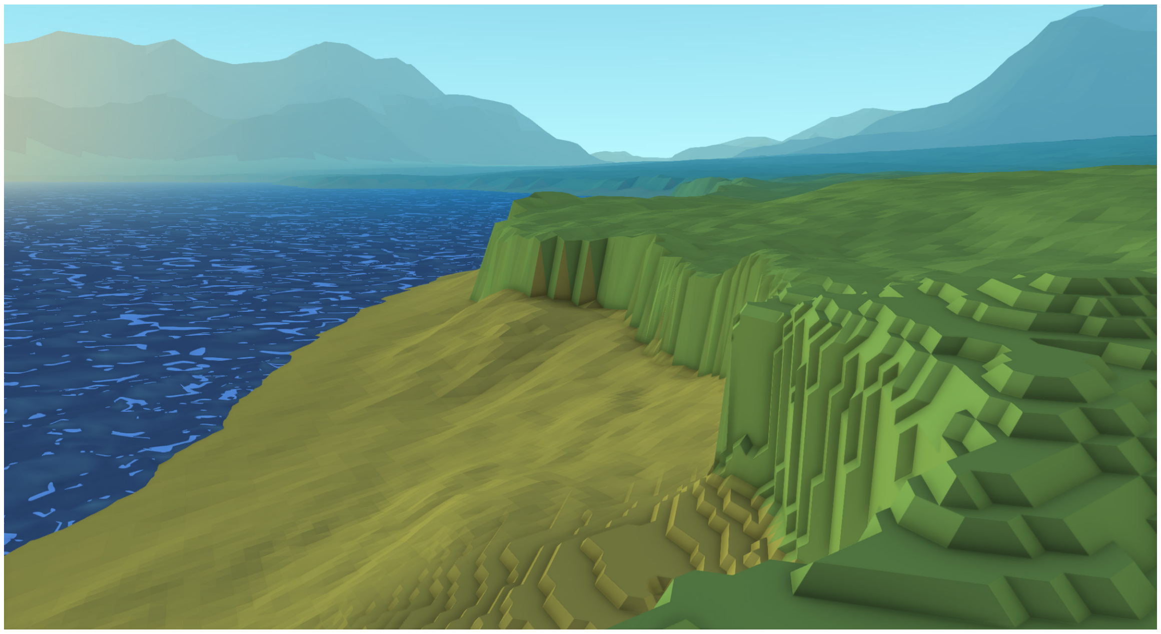

Figure 6.7: Shoreline cliff.

it is ideal to have smaller chunks. However, larger chunk sizes reduce the number of draw calls.

Figure 6.8 shows the relationship between chunk size and render performance for a fixed view distance. Since the same amount of geometry is rendered for each chunk size tested, the decrease in performance at low chunk size can be attributed to draw calls and other engine overhead. Figure 6.9 shows the modification cost for each chunk size. These results indicate that that a chunk size of 32 or 64 is a good compromise between render performance and modification cost.

6.3 Clipmaps

One major motivation for using geometry clipmaps is the exponential relationship between layer count and view distance. Figures 6.10 and 6.11 show the relationship between clipmap layers, view distance, and frame time in milliseconds for a scene containing only the clipmaps terrain and a skybox.

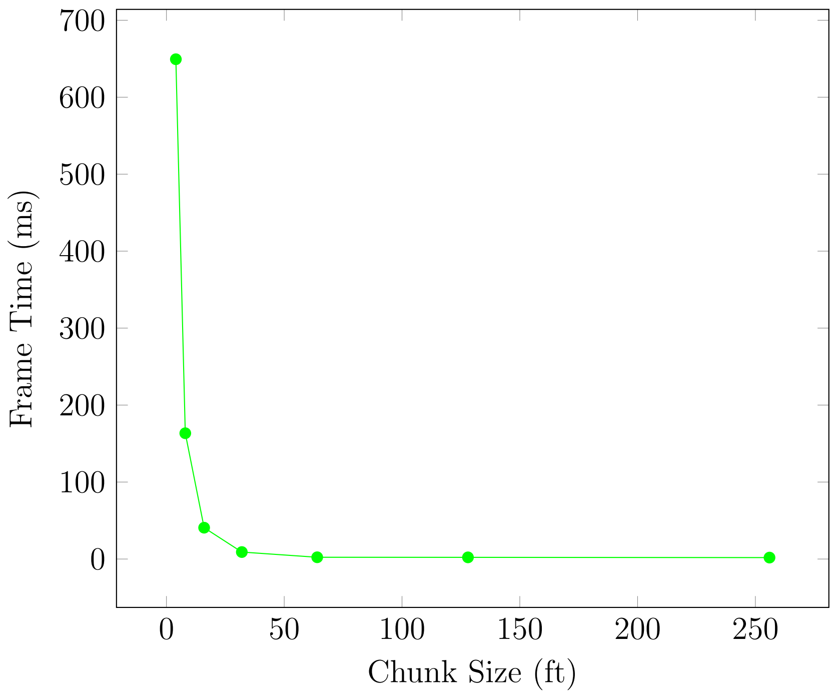

Figure 6.8: Voxel terrain rendering performance for different chunk sizes. For each chunk size, view distance is set to 512 feet.

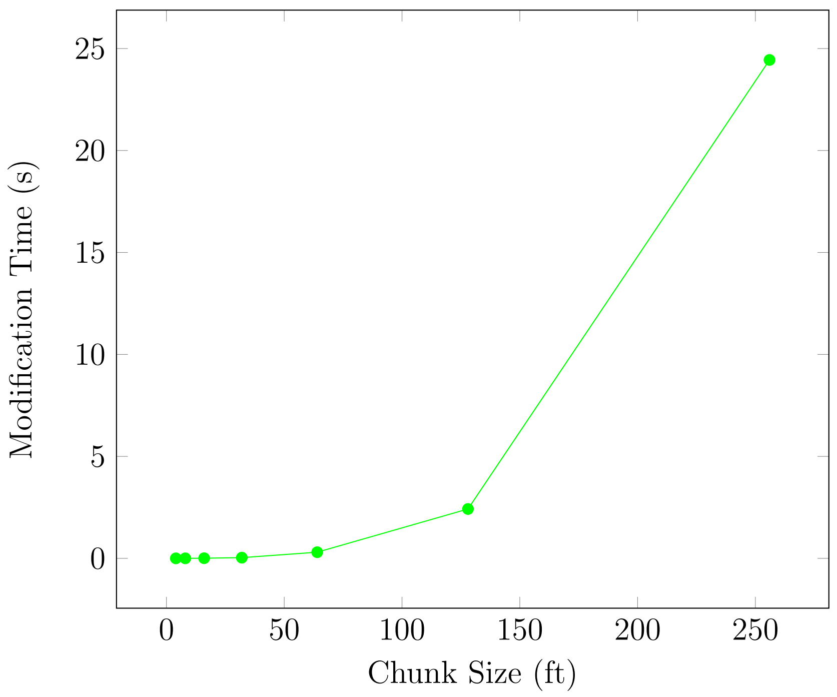

Figure 6.9: Graph of modification cost (i.e. digging) for different chunk sizes.

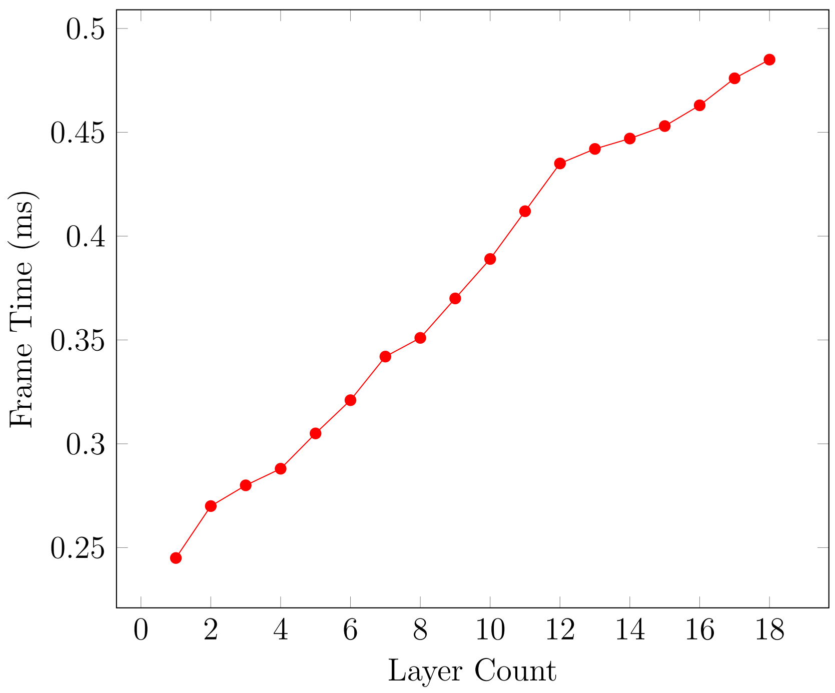

Figure 6.10: Geometry clipmaps render time for a given number of clipmap layers. Note that the frame-time is always sub-millisecond, even for a large number of layers.

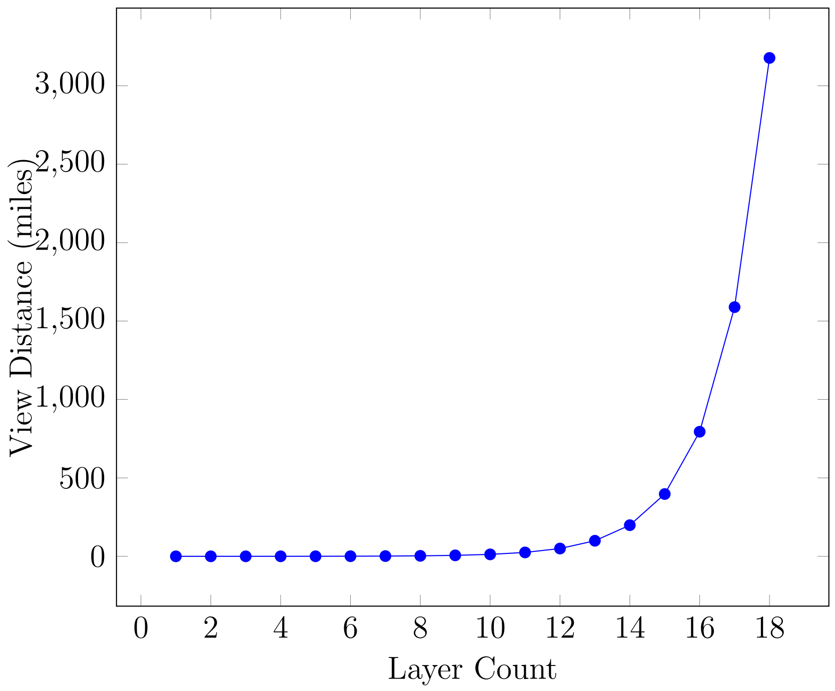

Figure 6.11: Geometry clipmaps view distance and for a given number of clipmap layers.

There is a linear relationship between frame time and layer count, but an exponential relationship between view distance and layer count. For a relatively minor increase in frame time an absurd view distance of over 3000 miles can be achieved. For reference, the largest possible view distance on earth is around 300 miles, between two mountains in South America. [11]

Larger individual layer sizes increase view distance and reduce polygon screen-space at the cost of performance. The addition of a geometry shader normal calculation simplifies the terrain generation process and improves visual quality, but incurs a small performance hit. We found a frame time increase of about 0.01 ms when using 11 layers.

6.4 Vegetation

The far layers of the vegetation system are rough visual approximations of the closeup tree meshes, but are significantly less costly for rendering. Figure 6.12 shows the cost of rendering a large number of each type of representation.

Increasing the size of tree groups improves rendering performance by reducing the number of draw calls, but also causes larger CPU lag spikes when a new group of trees has to be generated. Figure 6.13 shows the render performance for each group size.

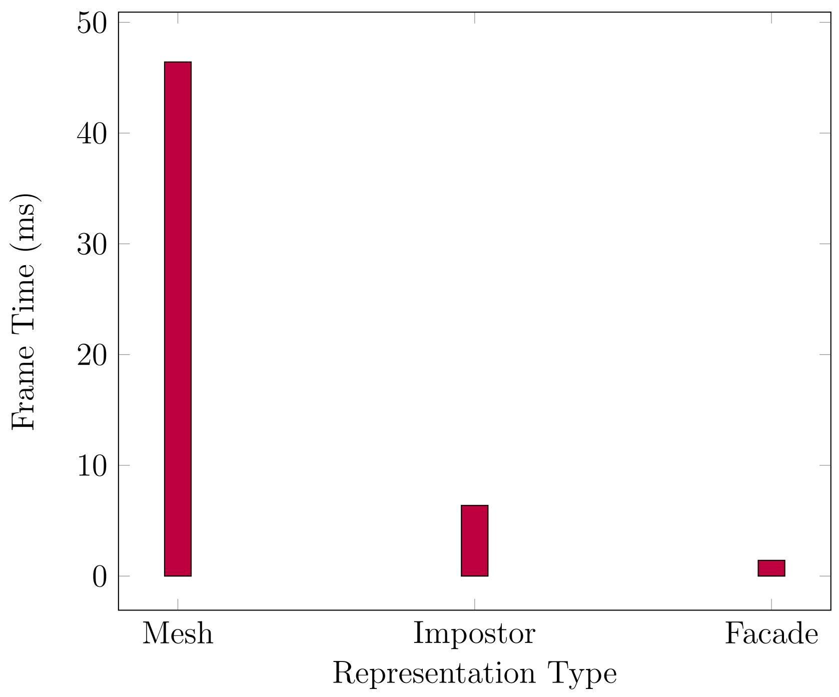

Figure 6.12: Tree performance vs. tree representation type, view distance set to 1280 feet.

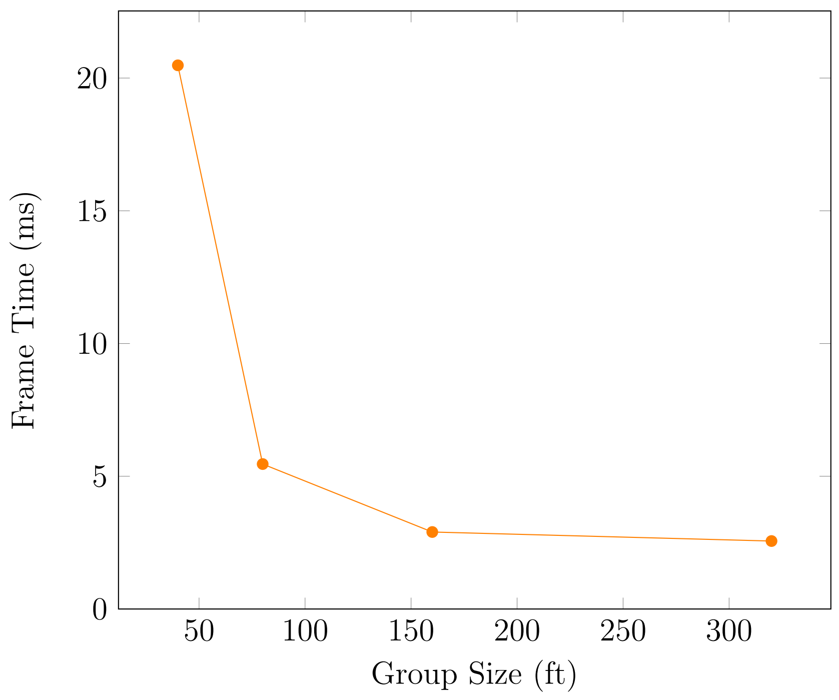

Figure 6.13: Tree performance vs. tree group size.

Chapter 7

FUTURE WORK

The system introduced supports multiple terrain types and render attributes, however the major limitations at this time are the diversity of the generated terrain, the lack of advanced lighting calculations, and the view distance of the forest rendering system.

7.1 Improved Terrain

Our presented terrain system produces vast mountain ranges, lakes, and continents. However, only a few terrain types are supported and at the highest resolution (the resolution the player primarily experiences) a lot of the terrain lacks interesting features. In the future we would like to expand our terrain generation system using some of the techniques from Génevaux et al. to add sophisticated water systems and more small-scale terrain features [14].

7.2 Terrain Lighting

Besides the HBAO+ post-processing pass applied to the whole scene, no shadows are currently calculated. The scene is inherently difficult to shadow due to the large view scales. A cascaded shadow map system with robust view frustum culling could be applied to the geometry clipmap layers but the performance hit on rendering may be significant.

One option to improve rendering performance is horizon-based shadowing. Horizon-base shadowing pre-calculates horizon values for each pixel in a heightmap. These horizon values are then compared against sun elevation to determine whether a fragment is in shadow.

While calculating horizon values for all clipmap layers may be expensive, it might be possible to calculate horizon values for e.g. every fourth layer and re-use the low resolution horizontal values in high resolution layers.

7.2.1 Accurate Scattering

Our current scattering system is a very rough approximation of real scattering. There are open source solutions with more accurate simulations of the Rayleigh and Mie scattering which could be used [3].

It might also be better to use a non-photo-realistic color-ramp scattering simulation such as the one used by Firewatch [24].

7.2.2 Volumetric Lighting

Nvidia Gameworks, which Relic uses for screen-space ambient occlusion, also has an implementation of volumetric lighting that could be used for more accurate sunset and sunrise effects [26].

7.3 Vegetation

While the mesh facade rendering system is cheap, it is a very rough visual approximation of distance forests. It is also not sufficiently cheap to render at significant scale. As such, the current system has a much shorter view distance for vegetation than it does for terrain.

Using a GPU ray-cast simulation of forests could produce much larger view distances and improved visual approximation. [21]

7.4 Water

Our water system uses the shape of geometry clipmaps for simplicity but this causes a lot of water vertices to be rendered off-screen. Using a grid projected from screen-space is one way to efficiently render water at different view scales [6].

BIBLIOGRAPHY

[1] A. Asirvatham and H. Hoppe. Terrain rendering using gpu-based geometry clipmaps.
`http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter02.html`.

[2] J. Bevins. Ridgedmulti class reference. `http://libnoise.sourceforge.net/docs/classnoise_1_1module_1_1RidgedMulti.html`. Accessed: 2016/12/16.

[3] E. Bruneton and F. Neyret. Precomputed Atmospheric Scattering. *Computer Graphics Forum*, 27(4):1079–1086, June 2008.

[4] E. Bruneton and F. Neyret. Real-time rendering and editing of vector-based terrains. *Computer Graphics Forum*, 27(2):311–320, Apr. 2008.

[5] E. Bruneton and F. Neyret. Real-time realistic rendering and lighting of forests. *Computer Graphics Forum*, 31(2pt1):373–382, May 2012.

[6] E. Bruneton, F. Neyret, and N. Holzschuch. Real-time Realistic Ocean Lighting using Seamless Transitions from Geometry to BRDF. *Computer Graphics Forum*, 29(2):487–496, May 2010. EUROGRAPHICS 2010 (full paper) - Session Rendering I.

[7] P. Cignoni, F. Ganovelli, E. Gobbetti, F. Marton, F. Ponchio, and R. Scopigno. Planet–sized batched dynamic adaptive meshes (p-bdam). In *Proceedings IEEE Visualization*, pages 147–155, Conference held in Seattle, WA, USA, October 2003. IEEE Computer Society Press.

[8] W. Commons. Voxels.svg.
`https://commons.wikimedia.org/wiki/File:Voxels.svg`, 2006. Accessed: 2016/12/16.

[9] DaveBurke. Toroidal_coord.png. `https://en.wikipedia.org/wiki/File:Toroidal_coord.png`, 2006. Accessed: 2016/12/16.

[10] W. H. de Boer. Fast terrain rendering using geometrical mipmapping, 2000.

[11] J. de Ferrant. Longest lines of sight. `http://www.viewfinderpanoramas.org/panoramas.html#longlines`.

[12] W. Evans, D. Kirkpatrick, and G. Townsend. Right triangular irregular networks. Technical report, University of Arizona, Tucson, AZ, USA, 1997.

[13] M. Finch. Effective water simulation from physical models. `http://http.developer.nvidia.com/GPUGems/gpugems_ch01.html`.

[14] J.-D. Génevaux, E. Galin, E. Guérin, A. Peytavie, and B. Beneš. Terrain generation using procedural models based on hydrology. *ACM Trans. Graph.*, 32(4):143:1–143:13, July 2013.

[15] H. Hakl and L. Van Zijl. Diamond terrain algorithm. *South African Computer Journal*, December 2002.

[16] B. D. Larsen and N. J. Christensen. Real-time terrain rendering using smooth hardware optimized level of detail. *Journal of WSCG*, 11(2):282–9, feb 2003. WSCG'2003: 11th International Conference in Central Europe on Computer Graphics, Visualization and Digital Interactive Media.

[17] E. S. Lengyel. *Voxel-based Terrain for Real-time Virtual Simulations.* PhD thesis, University of California at Davis, Davis, CA, USA, 2010. AAI3404919.

[18] P. Lindstrom, D. Koller, W. Ribarsky, L. F. Hodges, N. Faust, and G. A. Turner. Real-time, continuous level of detail rendering of height fields. In *Proceedings of the 23rd Annual Conference on Computer Graphics and*

Interactive Techniques, SIGGRAPH '96, pages 109–118, New York, NY, USA, 1996. ACM.

[19] W. E. Lorensen and H. E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. *SIGGRAPH Comput. Graph.*, 21(4):163–169, Aug. 1987.

[20] F. Losasso and H. Hoppe. Geometry clipmaps: Terrain rendering using nested regular grids. *ACM Trans. Graph.*, 23(3):769–776, Aug. 2004.

[21] S. Mantler and S. Jeschke. Interactive landscape visualization using gpu ray casting. In *Proceedings of the 4th International Conference on Computer Graphics and Interactive Techniques in Australasia and Southeast Asia*, GRAPHITE '06, pages 117–126, New York, NY, USA, 2006. ACM.

[22] G. S. P. Miller. The definition and rendering of terrain maps. *SIGGRAPH Comput. Graph.*, 20(4):39–48, Aug. 1986.

[23] Altitude. `http://minecraft.gamepedia.com/Altitude`. Accessed: 2016/12/16.

[24] J. Ng. The art of firewatch. `https://www.youtube.com/watch?v=ZYnS3kKTcGg`. Accessed: 2016/12/16.

[25] Hbao+. `http://www.geforce.com/hardware/technology/hbao-plus`. Accessed: 2016/12/16.

[26] Nvidia volumetric lighting. `http://www.geforce.com/hardware/technology/hbao-plus`. Accessed: 2016/12/16.

[27] A. Patel. Polygonal map generation for games.

http://www-cs-students.stanford.edu/~amitp/game-programming/polygon-map-generation/. Accessed: 2016/12/16.

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

[29] M. Persson. Terrain generation, part 1. http://notch.tumblr.com/post/3746989361/terrain-generation-part-1. Accessed: 2016/12/16.

[30] I. Quilez. outdoors lighting. http://www.iquilezles.org/www/articles/outdoorslighting/outdoorslighting.htm, 2013. Accessed: 2016/12/16.

[31] N. Reed. Depth precision visualized. https://developer.nvidia.com/content/depth-precision-visualized.

[32] P. Shanmugam and O. Arikan. Hardware accelerated ambient occlusion techniques on gpus. In *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games*, I3D '07, pages 73–80, New York, NY, USA, 2007. ACM.

[33] M. White. Real-time optimally adapting meshes: Terrain visualization in games. *International Journal of Computer Games Technology*, 2008.

[34] L. Williams. Pyramidal parametrics. *SIGGRAPH Comput. Graph.*, 17(3):1–11, July 1983.

[35] H. Zhou, J. Sun, G. Turk, and J. M. Rehg. Terrain synthesis from digital elevation models. *IEEE Transactions on Visualization and Computer Graphics*, 13(4):834–848, July/August 2007.