2012

Procedurally Generating Everything

Micah Caunter
The College of Wooster, mcaunter12@gmail.com

Procedurally Generating Everything

Independent Study Thesis

Presented in Partial Fulfillment of the Requirements for
the Degree Bachelor of Arts in the
Department of Computer Science at The College of Wooster

by
Micah Caunter

The College of Wooster
2012

Advised by:

Dr. Denise Byrnes

THE COLLEGE OF
WOOSTER

Abstract

This paper investigates using Perlin Noise in a three-dimensional environment. Perlin Noise has been previously used to simulate clouds, marble, wood, and other natural phenomena as well as to model virtual landscapes. The simulation created for the I.S. procedurally generates both static and animated textures and models virtual landscapes to create a virtual world. In order to keep up with the rapidly advancing graphics industry, the OpenGL API is utilized to generate the textures in real-time and to display the final product.

This work is dedicated to my parents, Pat and Donald, my brothers, Aaron and Chris, my sister-in-law, Mandy, and my step-father, Howard.

Acknowledgments

I would like to acknowledge Dr. Denise Byrnes for putting up with me for all these years and especially for all the editing she did and Dr. John Breitenbucher for putting together the Latex package for all the seniors to utilize for I.S.

Contents

List of Figures

List of Listings

CHAPTER 1

Introduction

Three-dimensional graphics are found everywhere; the start menu on a PC and the dock on a Mac, a character running around in a brand new video game, and the alien spaceship in a movie.

As recently as the 1970's, three-dimensional graphics were not mainstream. Computer usage was rare because few could afford the machines. Advancing the manufacturing of very large-scale-integrated (VLSI) chips for computer processors was the necessary step to bring costs down and allow the general public to become major users of computer systems.

Three-dimensional graphics have become much more realistic since their introduction in computers. During the 1960's, displayable graphics were limited to points, lines, and polygons defined by vertices and colors. These elements were not shaded based on light location, had no highlights based on the glossiness of the surface, and there was no way to simulate reflection or refraction [6].

In order to simulate realistic lighting, a lighting model was created. The model, known as the Phong Illumination Model or Phong Shading, was designed by Bui Tuong Phong in the mid 1970's. It accounts for ambient light (generic intensity of light not based on a single light source), diffuse reflectance (light that is created from a light source and is reflected from an object), specular reflectance (light reflectance

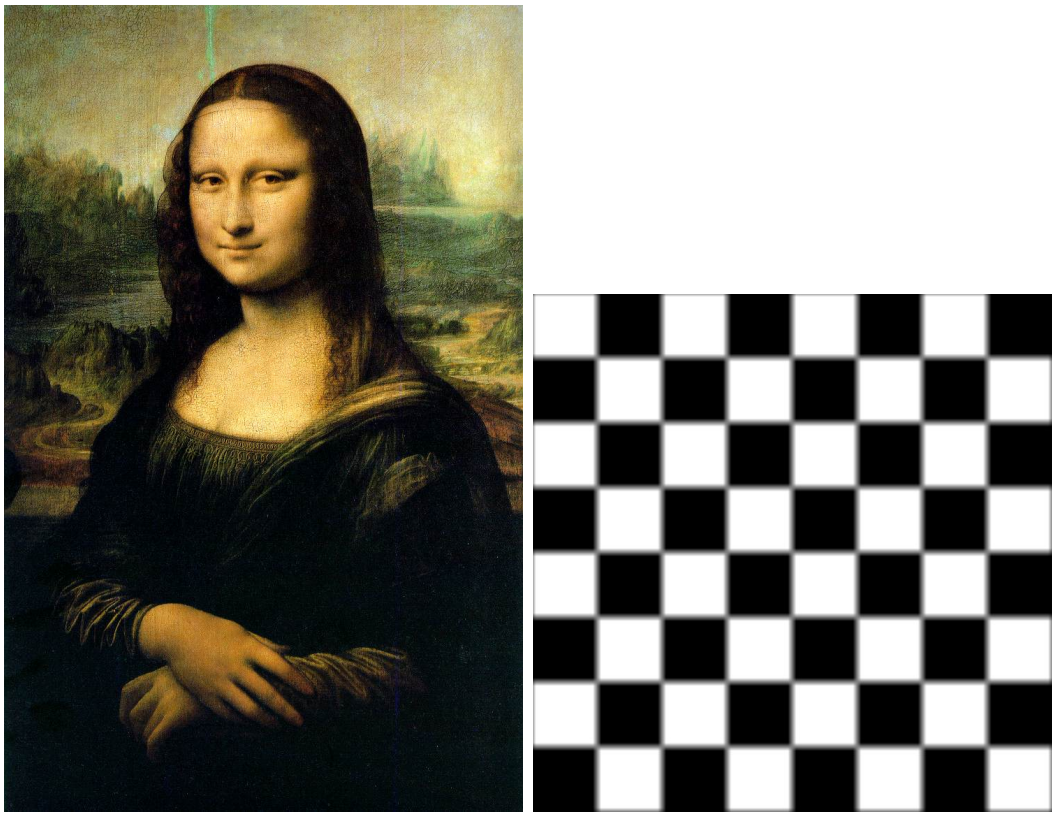
based on camera position and the shininess of an object), and light attenuation (light intensity based on the distance from the light source) [19].

One method of adding a pattern to a surface is to map a picture to the surface, a technique known as texture mapping. A texture is an image in one, two, or three dimensions that is mapped to a surface. A function is then applied to the image to map its coordinates onto the surface by scaling and stretching the image to fit the given surface [9].

James F. Blinn is responsible for extending Edwin Catmull's texture mapping algorithm for more realistic surface detailing. To add this realism, Blinn developed equations to calculate reflection and refraction based on the normal vectors of a surface. This allowed for surfaces to exhibit the environment around them rather than the specular highlight defined in the Phong Illumination Model [1].

Using textures poses a problem; how much memory is required to store textures? The size of a typical 256x256 image, stored in the JPEG format, requires from one to over 100 kilobytes, depending on its complexity. For an individual texture this size is minuscule because modern graphics cards contain a minimum of 256 megabytes of dedicated memory. But, a typical scene does not use a single texture mapped to a surface. Rather, a scene is usually composed of over 100,000 triangles and a multitude of textures to cover every surface. When the amount of data stored for a scene exceeds the amount of dedicated memory on the graphics card, the display rate slows to a crawl because the overflowing data must be stored in system memory.

Procedural generation helps alleviate this problem. Some images cannot be created by a mathematical function such as photographs or text (e.g. Figure 1.1(a)), but there are also many images such as wood, marble, and carpet (e.g. Figure 1.1(b)) that can be generated by functions. Procedural generation is used to create tile-able images, including the ones mentioned, using a variety of mathematically based functions. This allows functions to be called at runtime to generate the necessary textures.



(a) Mona Lisa [4] (b) Checkerboard

Figure 1.1: What Can and Cannot be Procedurally Generated

Using procedural generation functions to generate textures does not completely solve the problem of storing textures in memory, it only moves the texture storage off the hard drive. The functions must be implemented as programs that run on the graphics card in order to reduce the memory footprint required. These programs also have the ability to define more than one texture making them more memory efficient than using image files for the same purpose.

The procedural generation technique explored in this I.S. project is noise. Noise is generally considered something that is unwanted, but its randomness and unpredictability is what makes the textures look more natural and unique. It can be used to create a wide variety of natural phenomena including wood, marble, and clouds. The structure of these natural phenomena are easily identifiable, but because of their non-uniformity they require the randomness of noise in order to be generated.

Below is a brief overview of this I.S. project.

Chapter 2 examines the OpenGL API. The graphics pipeline for each version of OpenGL is examined by highlighting individual stages and how the pipeline has changed over time. The chapter also discusses major features introduced with each version of OpenGL.

Chapter 3 focuses on noise. Noise is examined in nature to justify its use in three-dimensional graphics before explaining it from a theoretical perspective. Several uses of noise are discussed, problems associated with mapping two-dimensional noise to a three-dimensional object, and the theory behind how a noise value is calculated.

Chapter 4 discusses the implementation of noise in C++ and GLSL. Two-dimensional and three-dimensional noise are implemented in a class for C++ and include examples of the resulting textures. Three-dimensional noise is also implemented in GLSL to demonstrate real-time texture generation and texturing complex three-dimensional objects.

Chapter 5 explores the applications of noise. It demonstrates using mathematical

functions to alter noise, simple textures based on interpolation between two colors, advanced textures that mimic natural phenomena, height map generation, and the generation of animated textures.

Chapter 6 builds on the applications of noise discussed in Chapter 5 in order to generate a virtual environment using only noise. The process is divided into multiple steps to highlight individual features as they are included in the application. At the end of the process the architecture of the application is analyzed to show its complexity.

CHAPTER 2

OpenGL

2.1 What is OpenGL?

OpenGL is currently the most widely used computer graphics API. Silicon Graphics originally created OpenGL in 1992 as an open graphics standard to replace IrisGL. In order for the standard to be open, Silicon Graphics created the OpenGL Architecture Review Board, a board that includes companies working together to guide the development of OpenGL. The open standard allowed OpenGL to flourish on a number of different platforms including Windows, Mac OS X, as well as Unix and Linux variants [12].

From a programmer's point of view, OpenGL makes development easier. It can be used with a wide variety of programming languages including C++, Java, and Python. It is scaleable to run on the fastest supercomputers, the smallest netbooks and even on smartphones. These advantages are why it is used to develop everything from medical imaging technology to CAD programs to game development [10].

2.2 OpenGL 1.X

The first versions of OpenGL, versions 1.0 to 1.5, were released over a long time period from 1992 to 2003, spanning many generations of graphics hardware. It was the first graphics library to offer multi-texturing support, a well documented implementation, and an open standard.

2.2.1 The Fixed-Function Pipeline

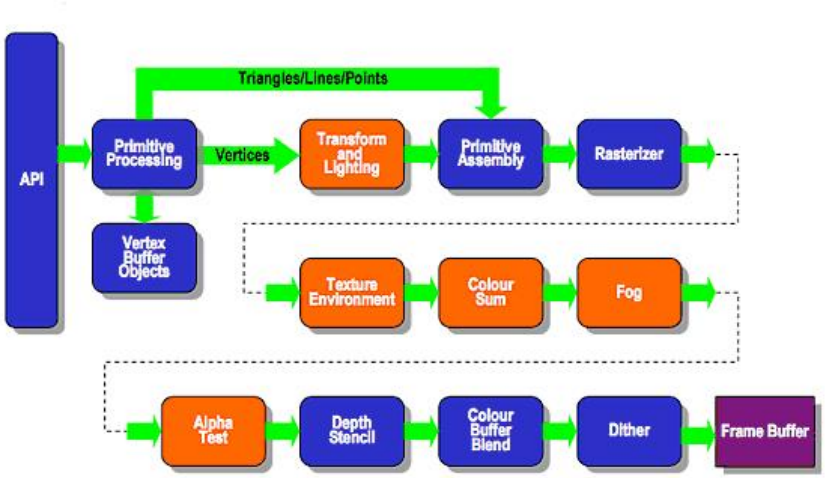


Figure 2.1: OpenGL Fixed-Function Pipeline [11]

OpenGL versions 1.0 to 1.5 use the process shown in Figure 2.1 for displaying graphics on screen. The next several paragraphs outline the process.

The first step in displaying graphics on screen is for the program to pass OpenGL commands to the API. These commands describe primitives such as triangles, lines, and points, vertices and their associated normal vectors, colors, materials, texture coordinates, and matrix operations. These commands are the basic building blocks of three-dimensional objects.

Next is the Primitive Processing stage. This is where all the commands passed to the API are organized (per primitive) for use further down the pipeline. Individual vertices and normal vectors are passed to the Transform and Lighting stage, but the primitives themselves skip this stage and go straight to Primitive Assembly.

In the first part of the Transform and Lighting stage, vertices are moved to their designated scene position using the Model-View Matrix. The Model-View Matrix is a 4x4 matrix based on the camera viewpoint that translates, rotates, and scales vertices into world space. Then the vertices are either flat shaded, one color per primitive, or Gouraud shaded where light intensity is calculated at each vertex using the normal vectors [8].

The Primitive Assembly stage reunites vertices from the Transform and Lighting stage with their associated primitives passed from the Primitive Processing stage. Then the primitives are assembled in world space before being transformed by the Projection Matrix to be contained within a 2x2x2 cube centered at the origin.

The Rasterizer stage converts the primitives from the Primitive Assembly stage into a two-dimensional image that can be displayed. Each point in the image corresponds to an integer coordinate of the area being drawn. These points, commonly called fragments, contain their assigned color, depth, and texture coordinates.

After rasterizing, the primitives enter the Texture Environment stage. This is where an image can be mapped onto a primitive. By using a fragment's texture coordinates, a color at those coordinates in an image is mapped to the fragment and thus modifies the fragment's original color.

The Color Sum stage is used to add the fragment color from the Texture Environment stage to a secondary color if one is defined, allowing a color to be applied to a fragment after texturing. This stage is always applied when lighting is enabled.

Another stage that operates on fragments generated by the Rasterizer is the Fog stage. The fog stage uses the depth of a fragment to fade out primitives based on

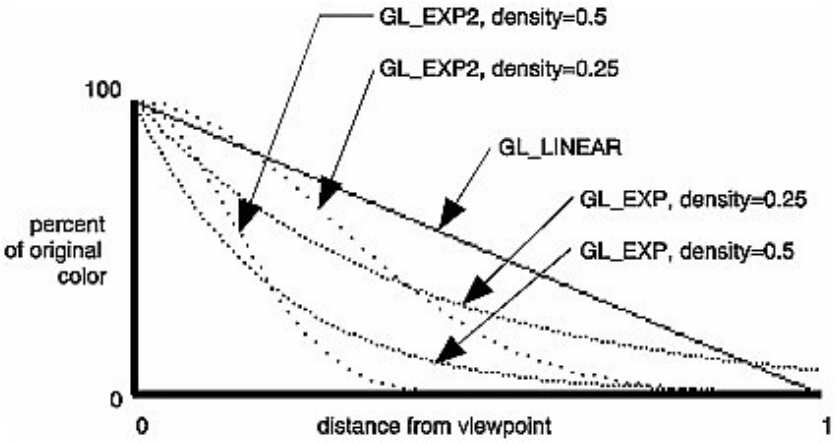


Figure 2.2: OpenGL Fog Functions [28]

their distance away from the camera. The color of the fog is defined by commands passed to the API and its intensity is defined by a linear or exponential function, also defined by commands to the API. The graph in Figure 2.2 shows the percentage of a fragment's original color in relation to the fog color.

The next step in the pipeline is the Alpha Test, which is used mainly for transparency. This step performs a logical comparison between the alpha value of a fragment and a reference value from the range [0, 1] depending on the specified test function. If the value returned is false then the fragment does not continue down the pipeline. When the Alpha Test is disabled the comparison always passes.

Before continuing on it needs to be made clear that the data being worked on in the OpenGL pipeline is stored in buffers. The buffer that holds all the colors to be displayed is the frame buffer. Other notable buffers are the stencil and depth buffers holding bit-masks and the depth of each fragment respectively.

The next stage is where the Stencil and Depth Buffer tests are carried out. The Stencil Buffer test is used for a logical comparison between the value in the stencil buffer at the fragment location and a reference value. If the test passes, the values

compared are bitwise-anded together. Otherwise the fragment is discarded from the pipeline. This is helpful in creating shadows and reflections.

The Depth Buffer test is used to make sure the incoming fragment is displayed if it passes a logical comparison with the depth value at the same location in the depth buffer. If the test fails, the fragment is discarded from the pipeline. However, if the test passes then the depth of the fragment is written to the depth buffer and it continues down the pipeline. This test allows primitives closest to the camera to be drawn.

Next is the Color Buffer Blend stage where colors previously stored in the frame buffer are blended with fragments passed in from the Depth Buffer test. The source value, the value of the incoming fragment, is blended with the destination value, the current value in the frame buffer. This stage is generally used to create the illusion of transparency.

The last stage before saving everything to the frame buffer is the Dithering stage. Because floating point values are used in the pipeline and the frame buffer only has a specific number of bits per color component, the extra accuracy must be removed before being saved to the frame buffer. When dithering is disabled, the extra accuracy is truncated and has no effect on the final color. However, when dithering is enabled it rounds the fragment color.

At the end of the pipeline everything is stored in the previously mentioned buffers; the frame buffer, depth buffer, and stencil buffer. The values stored in these buffers can be used by the pipeline if there is more data to be passed through. In order to display the data, another command is issued to send the frame buffer data to the screen.

2.2.2 Major Features

OpenGL 1.X brought several useful features to graphics rendering that had not been seen before. A few of the main features are discussed below including multi-texturing, array and buffer objects, and display lists.

2.2.2.1 Multi-Texturing

Multi-texturing is the process of using multiple textures for one primitive. In order to do this, a set of texture coordinates are supplied for each texture. The different textures are selected by setting the "active texture" so that the pipeline knows which set of texture coordinates to use. The Rasterizer is used to find the texture coordinates of each texture for each fragment. Then all the information is combined before going to the Color Sum stage.

While the benefit of multi-texturing may not immediately be evident, and may even seem slower than using one texture for a primitive, it is actually faster overall. The boost in speed comes from storing fewer textures because multiple textures can be combined to make new textures rather than storing every texture individually. Less overall textures means less time spent switching between textures and less time spent loading and unloading textures to and from memory.

2.2.2.2 Array and Buffer Objects

Array and buffer objects differ only in where they are stored. Array objects are stored in client memory (system memory) while buffer objects are stored in server memory (graphics card memory). This gives buffer objects a speed advantage because the data to be worked on already resides in graphics card memory.

These objects are used to store any data that can be converted to a vector format such as vertices, normal vectors, texture coordinates, and colors. All the data is specified per vertex in the order it is drawn for a primitive. Data held in a buffer

object can be of one type or it can be mixed, but mixed data must be uniformly organized.

2.2.2.3 Display Lists

A display list is a group of commands that are precompiled and stored in server memory (graphics card memory) so that an object or scene can be more easily reproduced with a single command. Rather than calling a function that uses CPU time to send all the object or scene data to the frame buffer, calling a display list is much faster.

Display lists can also be used hierarchically, or nested within one another. This makes sense when multiple objects have similar pieces such as a SUV and a truck. Both the truck and SUV have four wheels, but they have different chassis. This means that the display list for each vehicle has a unique display list for the vehicle's specific chassis, but the same display list is used for each of the wheels [22].

2.3 OpenGL 2.X

OpenGL 2.0 was released in 2004 and was the first major revision of OpenGL. The main addition was full support for vertex and fragment shaders that had previously only been available through extensions. An extension is an addition to the OpenGL API that is not part of the core features required for compatibility with the API. Extensions can be promoted to official features in later revisions of OpenGL, but they are not required to be implemented in hardware to be compatible with OpenGL.

2.3.1 The Programmable Pipeline

Figure 2.3 shows the pipeline used by OpenGL 2.0 and 2.1. When compared to the Fixed-Function Pipeline used by OpenGL 1.X, the Vertex Shader replaces the Transform and Lighting stage and the Fragment Shader replaces the Texture Environment,

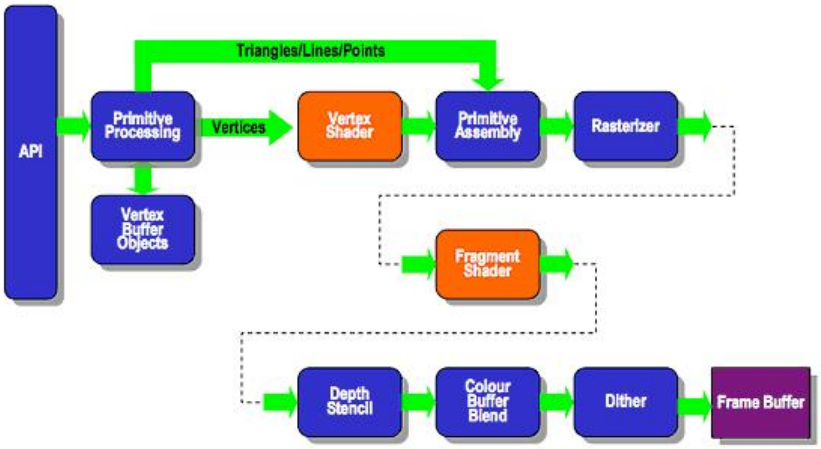


Figure 2.3: OpenGL Programmable Pipeline[11]

Color Sum, Fog, and Alpha Test stages. The replaced stages are still completely supported by the pipeline, but are not computed if programmable shaders are used. Rather than discuss the entire pipeline again, only the changes are discussed.

As stated previously, the Vertex Shader replaces the Transform and Lighting stage. The Vertex Shader operates on only one vertex at a time and has access to only the data for that vertex. Its main purpose is to perform the lighting calculations and move the vertex into world space just as the Transform and Lighting stage did. This stage can also define variables for its associated Fragment Shader to use later on in the pipeline. These variables are called varying, their value(s) are interpolated across the primitive similar to how colors are interpolated with Gouraud Shading.

After going through the Primitive Assembly stage, the Fragment Shader takes over. The Fragment Shader operates on one fragment at a time and has access to only the data for that fragment. Because it encompasses the Texture Environment, Color Sum, Fog, and Alpha Test stages, it takes on all those roles as well as applying functions of its own. For a fragment shader to be complete it must define a color for the fragment. If a depth value for the fragment isn't defined then the generic one from the fixed-function pipeline is used.

Other than these changes, the pipeline remains the same. The shader stages provide much more power than the fixed-function pipeline equivalents and are described in section 2.3.2.

2.3.2 Major Features

While vertex and fragment shaders provide most of the additional functionality in OpenGL 2.x, there are some other advancements worth noting; point sprites, non-power-of-two textures, and pixel buffer objects.

2.3.2.1 Vertex Shaders

As stated earlier, vertex shaders are used to replace the Transform and Lighting stage of the fixed-function pipeline. Varying variables, variables interpolated across a primitive based on values for each vertex, provide most of the power for vertex shaders. For example, Gouraud Shading calculates per vertex lighting by determining the color at each vertex and interpolating the color across the polygon. Using a varying variable, the normal vector can be interpolated across the primitive, moving the lighting calculation to the fragment shader, which allows the Phong Illumination Model to be used. The vertex position in space can also be changed before being moved into world coordinates. This allows deformation of a surface by changing the vertex coordinates, but there is limited usability of this feature because the normal vector cannot be recalculated by the vertex shader.

2.3.2.2 Fragment Shaders

Fragment shaders work with vertex shaders to extend the flexibility and power already provided. Phong Shading is only a small part of what can be done. Normal mapping, also known as bump-mapping, is the process of altering normal vectors. This used to be a per vertex operation, but can now be done with functions and textures using

the fragment shader. This allows for simulating a rough surface without the need for extra primitives. Building on this concept are relief and parallax mapping, both of which simulate occlusion, that is, hiding a section of the texture, as well as simulating a rough surface. A good example of this is a brick sidewalk where the mortar has been worn down between the bricks. Bump mapping allows for the surface of the bricks to be smoother and the mortar to be rougher. Relief and parallax mapping make the mortar appear to be at a different depth than the bricks.

2.3.2.3 Point Sprites

Point Sprites are used in particle systems, systems of particles used to simulate phenomena such as clouds, fire, and smoke. They allow GL-points to be textured rather than given a single color defined by a texture or a color defined by other OpenGL commands. GL-points are an OpenGL primitive that is represented by a square that is rotated towards the camera.

2.3.2.4 Non-Power-Of-Two Textures

Originally, texture dimensions were limited to powers of two such as 64x64 and 256x256. This addition to the library allows for textures of any dimension to be defined so long as they are still below the maximum texture size supported by the hardware.

2.3.2.5 Pixel Buffer Object

Before Pixel Buffer Objects were included in OpenGL, textures were stored in client memory (system memory) and all the operations involved were performed on the CPU. Now, textures can be stored in a similar way to vertices and normal vectors in buffer objects. This allows OpenGL to store the textures in server memory (graphics card memory) for faster access in the pipeline [23].

2.4 OpenGL 3.X

OpenGL 3.0 was released in 2008 making it the second major update to OpenGL since its release. With this release the main additions are geometry shaders and the deprecation model. Geometry shaders operate on entire primitives providing programmers with the ability to generate 3D models on the graphics card. The geometry shader uses transform feedback to provide better quality surface deformation when compared to vertex deformation in a vertex shader. The deprecation model was introduced in order to streamline the OpenGL API because it was growing rather large. Deprecated features are still available for use under the compatibility profile and the newer features are available under the core profile.

2.4.1 Deprecation Model

The deprecation model is meant to streamline the OpenGL API. Deprecated features are available under the compatibility profile. The process of deprecating a feature is simply marking it for deprecation in one OpenGL version and then removing the feature from the core profile in a following OpenGL version.

The features targeted first were mainly from OpenGL 1.X. This includes the immediate-mode commands such as glVertex, glNormal, glColor, glBegin, glEnd, and a large number of other commands. This also removes display lists because they are precompiled sets of immediate-mode commands. These commands are superseded by server-side commands that use Vertex Buffer Objects.

Other notable features that are deprecated include operations that are replaced by Vertex and Fragment shaders, including the Transform and Lighting, Fog, Texture Environment, Alpha Test, and Color Sum stages. This was again, to move processing from client-side to server-side.

Because the Vertex Shader replaces the Transform and Lighting stage, the matrix stack and built-in lights are also deprecated. This makes it the programmer's responsibility to manage their own matrix stack and lights.

2.4.2 OpenGL 3.X Core Pipeline

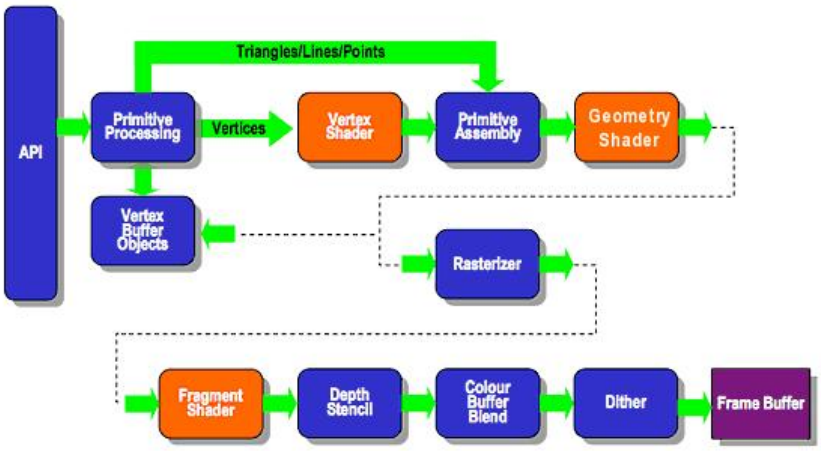


Figure 2.4: OpenGL 3.X Core Pipeline (Modified from [11])

Figure 2.4 shows the new pipeline structure for OpenGL 3.X. Nothing was removed from the OpenGL 2.1 pipeline. The only addition is the Geometry Shader.

The Geometry Shader receives specific primitive types from the Primitive Assembly stage. It takes any form of triangle, line or point such as a triangle fan or line strip, but it does not take other primitive types such as quads or polygons. In addition, there are also adjacency primitive types that allow the Geometry Shader to read data from vertices associated with primitives adjacent to the current primitive.

During operation, the Geometry Shader can choose to output the same primitive that is passed in, known as a pass-through shader, it can output multiple primitives, or it can choose to discard the primitive altogether. The primitives passed out of the Geometry Shader do not have to match the type of primitives passed in from Primitive Assembly.

There are two options for primitives once they leave the Geometry Shader. One option is to be passed to the Rasterizer and continue with the rest of the pipeline similar to previous versions of OpenGL. The other option is to use Transform Feedback and pass the primitives into a new Vertex Buffer Object to allow another set of shaders to work on the primitives before being displayed.

2.4.3 Major Features

Geometry Shaders are the main feature added with OpenGL 3.X, but another notable upgrade is newer versions of the GLSL shading language (1.3, 1.4, 1.5, 3.3).

2.4.3.1 Geometry Shaders

Section 2.4.2 briefly describes the process that a Geometry Shader goes through in the pipeline. The ability to completely discard primitives from the pipeline is an advanced form of culling. More important than that, though, is Transform Feedback.

For example, given a set of points in 3D space that represent normalized densities, a Geometry Shader could use those points to construct triangles using algorithms such as Marching Cubes or Marching Tetrahedrons. These triangles could be passed through the rest of the pipeline to be displayed, but they would have no lighting, textures, or colors associated with them. By using Transform Feedback, the triangles could be stored in a new Vertex Buffer Object and go through the pipeline again. The second time through, the Vertex Shader would transform the points to the proper scene position, the Geometry Shader would pass the primitives through to the Rasterizer, and the Fragment Shader would be free to do its job normally.

2.4.3.2 GLSL Language

With the addition of GLSL 1.3, Vertex and Fragment Shaders could finally use bitwise operations beyond comparisons for conditional statements. This could previously only

be used in GLSL 1.2 if an extension was loaded in the shader program [14]. This addition is necessary for the noise calculations focused on later in this paper.

GLSL 1.5 marked the addition of Geometry Shaders without the need for extensions to be loaded. GLSL 3.3 began to match the OpenGL and GLSL version with each new release, making it easier to keep track of the version numbers [24].

2.5 Conclusion

This chapter examines the most widely used computer graphics API, OpenGL. It discusses each major version of OpenGL by explaining how data is passed through the pipeline before being displayed. This chapter serves as the programming foundation for the effects examined in later chapters. Next, Chapter 3 forms the theoretical foundation of this I.S. by focusing on noise.

CHAPTER 3

Noise

3.1 What is Noise?

Noise, as it is defined by Eric Weisstein, is"an error that is superimposed on top of a true signal."[27] A common form of noise is the background static produced by speakers. Analog signal transmission to televisions produces noise as tearing or jaggedness across the on-screen picture as shown in Figure 3.1.

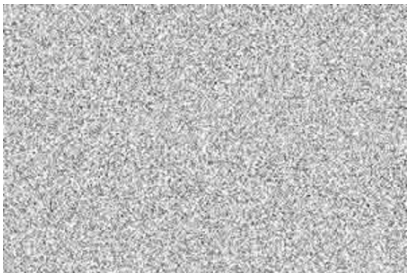


Figure 3.1: An Example of Non-Coherent Noise [26]

Both of the mentioned forms of noise are an annoyance when trying to enjoy the end product of music or the evening news. Not all noise is detrimental, but can actually be quite interesting as shown in Figure 3.2.

Figure 3.2 shows examples of noise that are found in nature. Examining the line between the landscape and the sky shows that it is not smooth, but rather bumpy and sharp when inspected more closely. The same can be said for the edges of the

Figure 3.2: Photo Taken at Garden of the Gods

clouds. They appear to be round and puffy from a distance, but again when examined at a closer level they have much more detail with no apparent pattern. The edge lines of trees against the landscape have the same design, a smooth shape from far away but far more jaggy detail up close. The edges of clouds, trees, and the landscape in Figure 3.2 are all examples of one dimensional noise.

3.2 Noise As a Mathematical Function

Mathematically, noise is a set of random values. The values are not truly random, but are seeded in order to reproduce the same set of values when requested. In order to produce these values, the noise function requires an integer seed and parameters based on the number of dimensions of the noise; it needs one parameter for one-dimensional noise, two parameters for two-dimensional noise, and so on. Figure 3.3 shows a graph of one-dimensional noise values.

Noise values by themselves are still not useful. It is necessary to interpolate between the noise values to establish a continuous function. Possibilities for this

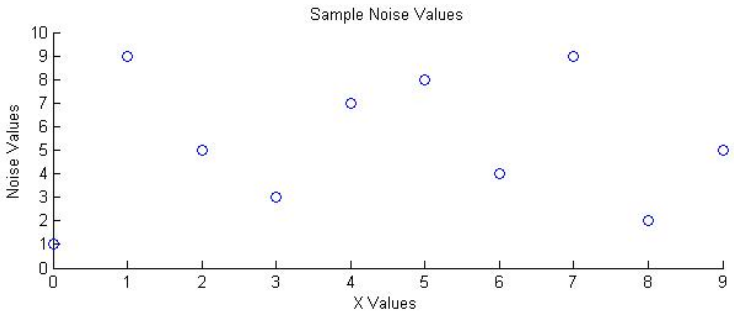


Figure 3.3: Graph of Sample Noise Values

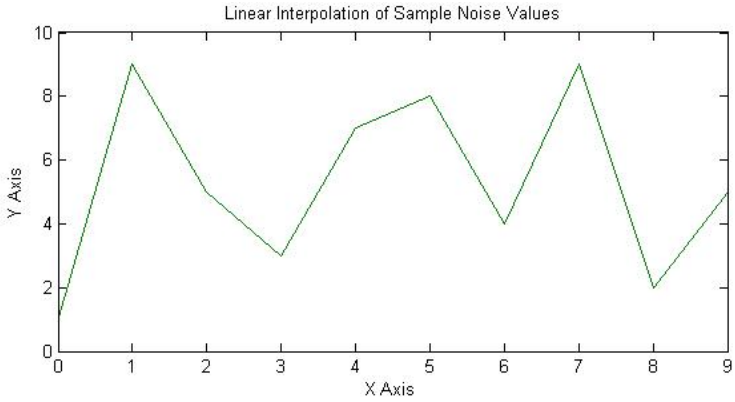
are limited to continuous functions such as odd-powered polynomials (linear, cubic, quintic, etc.) and the cosine function. Linear and cosine interpolation require only two points for interpolation making them the fastest computationally. For the noise values provided in Figure 3.3, the cubic interpolation requires four points in order to ensure a fully continuous function. The function must be differentiable across its entire domain causing it to be computationally inefficient in higher dimensions. Figure 3.4 demonstrates the differences between linear and cubic interpolation of the noise values from Figure 3.3.

Noise functions have an amplitude, frequency and wavelength similar to sine and cosine waves. By examining Figure 3.5 we can see that amplitude is defined to be half of the height of the wave. Wavelength is the length of one cycle of the wave, or more simply the distance from one peak or valley to another. The frequency of the wave is then defined to be the multiplicative inverse of wavelength.

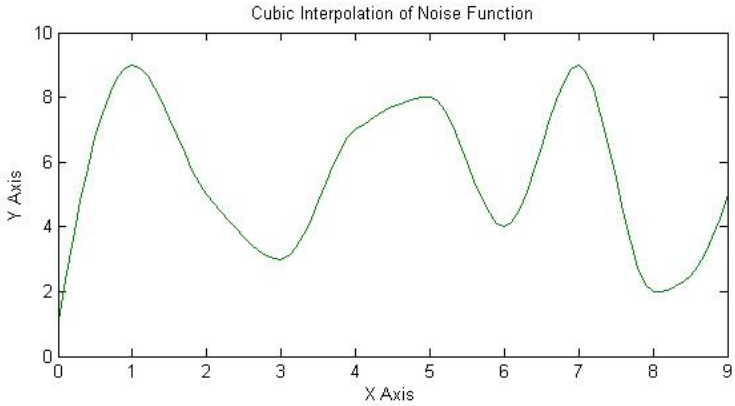
By examining Figure 3.6 we see that noise's amplitude is the difference between the maximum and minimum values generated by the function. Wavelength is the distance between two values passed to the function. Frequency is still defined to be the multiplicative inverse of the wavelength.

3.3 Perlin Noise

In its simplest form, Perlin Noise is a sum of noise functions into a single set of values.



(a) Linear Interpolation



(b) Cubic Interpolation

Figure 3.4: Methods of Interpolation

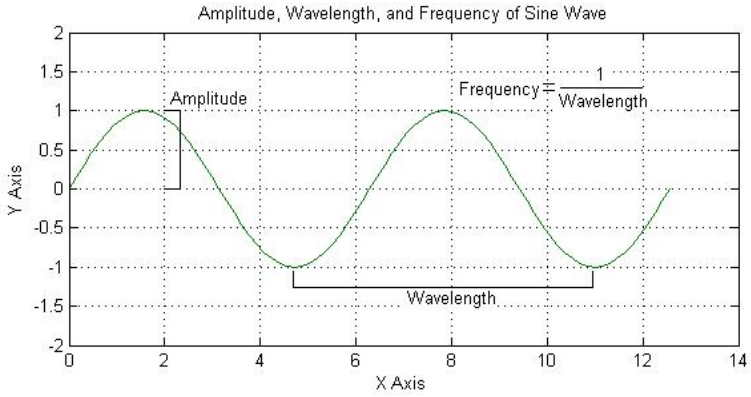


Figure 3.5: Amplitude, Wavelength, and Frequency of a Sine Wave

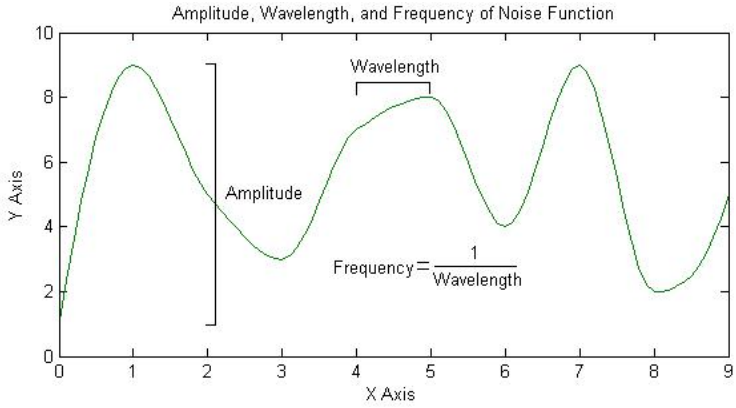
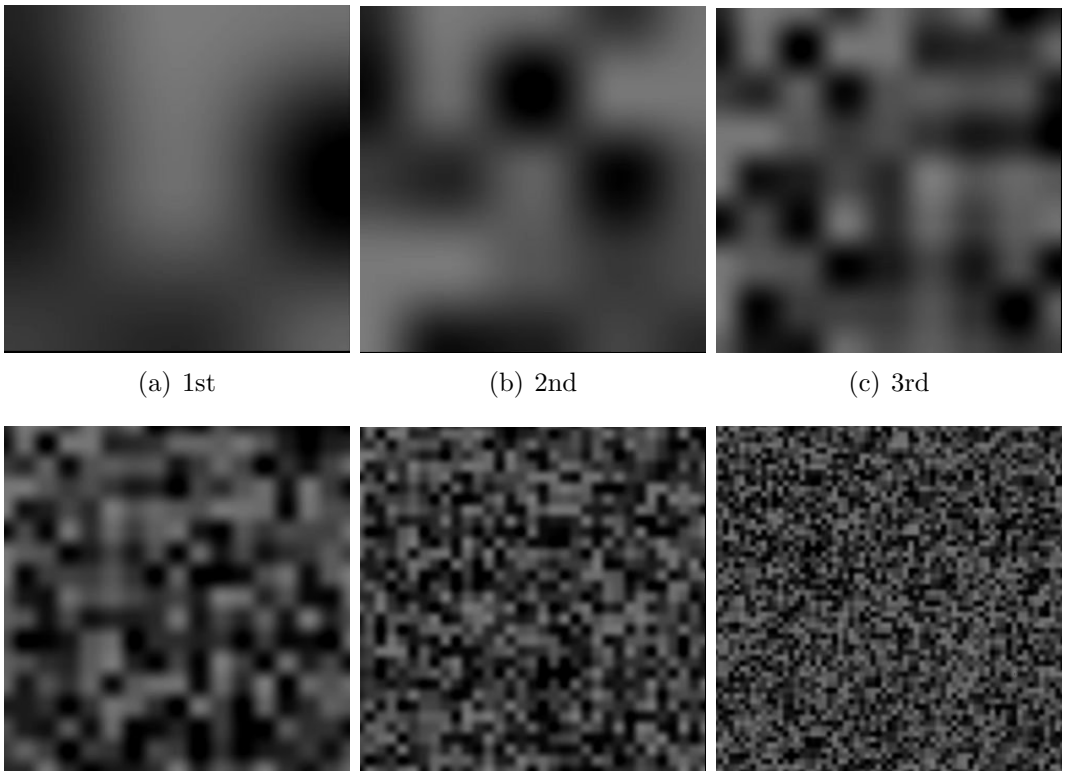


Figure 3.6: Amplitude, Wavelength, and Frequency of a Noise Function

By examining Figure 3.7 it can be seen that the frequency doubles for each subsequent picture from left to right causing each picture to become rougher, or "noisier." The amplitude is halved at the same time, but in order to show the differences in frequency, the values were normalized into the range [0, 1]. Because Perlin Noise and musical notes share the properties of amplitude and frequency, each of the pictures in Figure 3.7 is referred to as an octave.

Figure 3.8 is the final product of adding all the octaves together from Figure 3.7, resulting in an image that resembles a cloud. The first three octaves shown in Figure 3.7 are responsible for the main shape of the cloud while the latter three octaves are more responsible for the edge of the cloud. This is caused by the differences in amplitude and frequency of each octave.



(a) 1st (b) 2nd (c) 3rd

(d) 4th (e) 5th (f) 6th

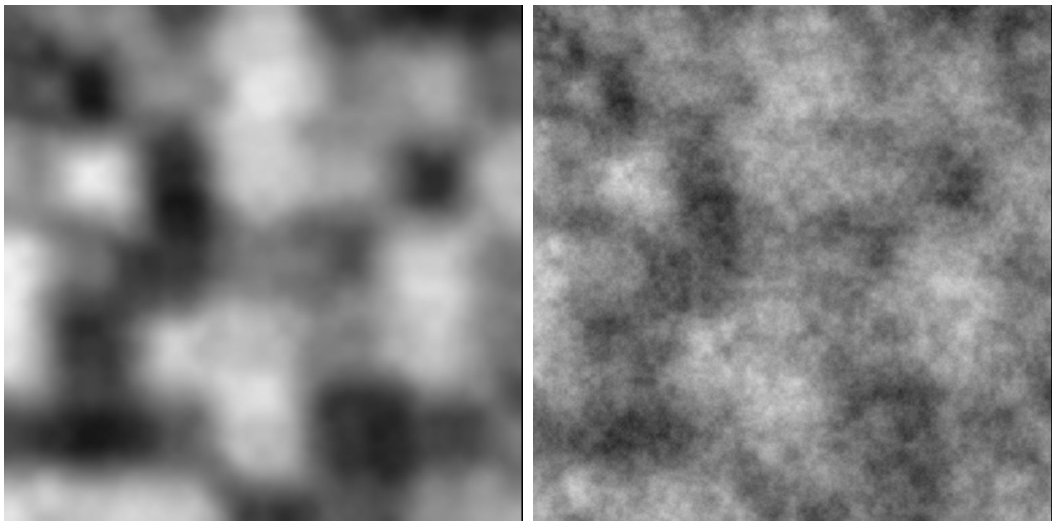
Figure 3.7: Octaves of a 2D Noise Function

Figure 3.8: Sum of Noise Functions

While Figure 3.8 does look like clouds, it can also be used as a height-map to define a landscape or as a texture. Textures generated using Perlin Noise can be specified infinitely, without repetition, making them much more suitable for a flat surface than a standard tile-able texture. The "infiniteness" of a non-repeating noise texture continues until the random number generator starts repeating values.

The definition of an octave may come from music, but there is no requirement to follow that rule explicitly. By definition, each octave's frequency is $2^{\text{octave number}}$ and the amplitude is $.5^{\text{octave number}}$. The .5 used in the amplitude equation is the persistence of a noise function. The persistence value is used to determine the roughness of the noise values and must be kept in the range (0,1) so that each subsequent octave affects the overall noise value less than the previous one. Lowering the persistence causes the resulting images to have a smoother output and conversely a higher persistence causes the resulting image to have a rougher output. This effect is shown in Figure 3.9.

In order to magnify a section of the texture and view it in more detail, a zoom variable is used. Zoom, by definition, means to look at something in closer detail, and in this case we are looking at the texture. Figure 3.10 shows different levels of zoom for two dimensions.



(a) Low Persistence (P=.3) (b) High Persistence (P=.7)

Figure 3.9: Differences in Persistence

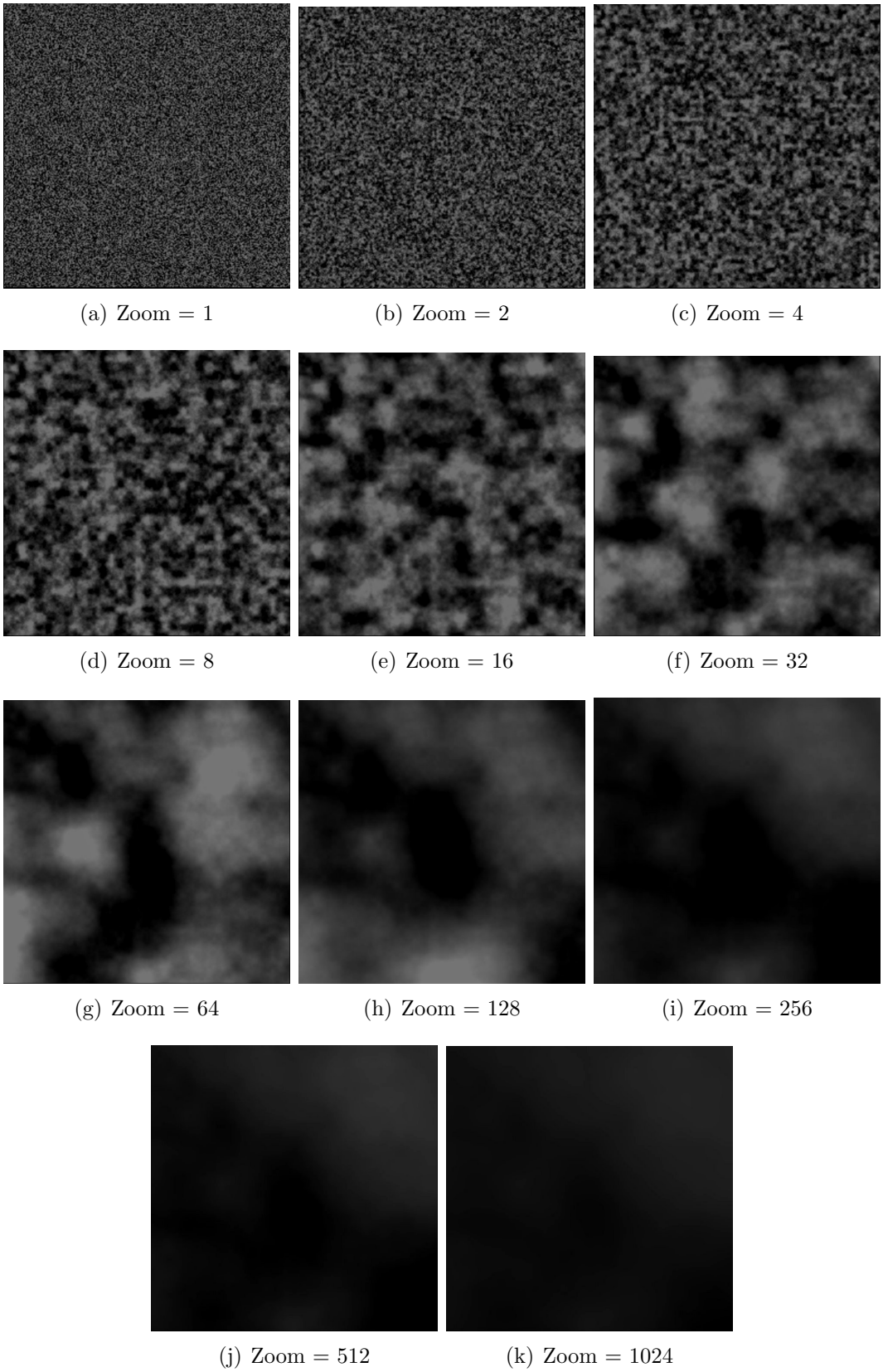
Notice that the texture never becomes blocky due to magnification. This is because the random number generator only takes integers as parameters. Noise values that are calculated between integer points are interpolated and this creates a limit on the level of detail at high zoom values. This is explained further in subsection 3.4 [5].

In three dimensions, Perlin Noise can take terrain, clouds, and texturing to another level. Figure 3.11 shows two examples of landscapes generated from two-dimensional Perlin Noise.

While both these terrains looks plausible, they are missing a few important geologic features. Cliffs and overhangs are impossible to generate because there cannot be two separate heights for the same set of parameters. Three dimensional noise adds these details.

As Figure 3.12 shows, Perlin Noise in three dimensions allows for any type of terrain to be created, from cliffs to overhangs to caves. This is because in three dimensions, the noise values are densities that determine the solidity of the object at that point.

When moving from two to three dimensions, Figure 3.13 shows that the visual



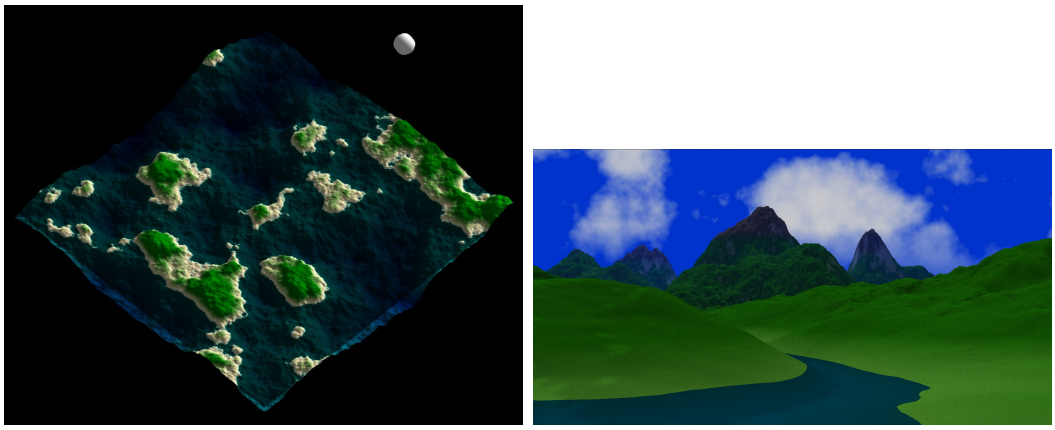
(a) Zoom = 1 (b) Zoom = 2 (c) Zoom = 4

(d) Zoom = 8 (e) Zoom = 16 (f) Zoom = 32

(g) Zoom = 64 (h) Zoom = 128 (i) Zoom = 256

(j) Zoom = 512 (k) Zoom = 1024

Figure 3.10: Zooming Into a Two-Dimensional Noise Function



(a) Island Landscape (b) Mountain Landscape

Figure 3.11: Landscapes Generated From Two-Dimensional Perlin Noise

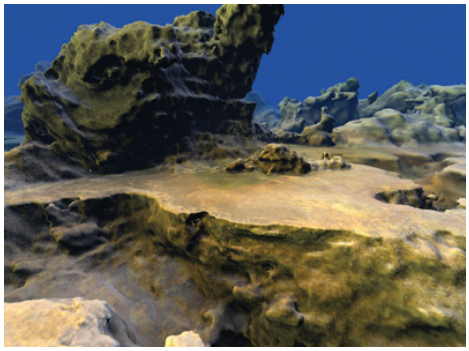
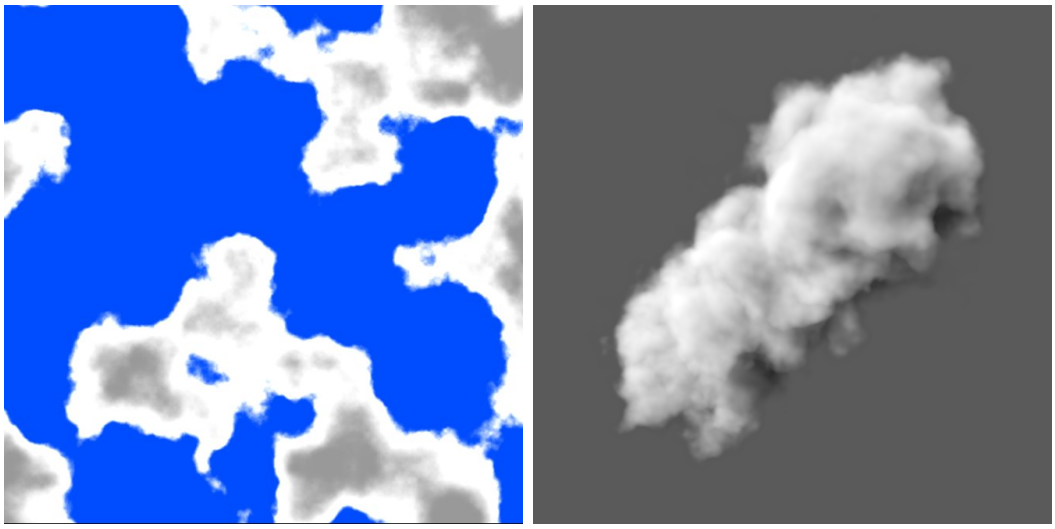


Figure 3.12: Three-Dimensional Noise Terrain [7]

quality of clouds improves. Two dimensions allows for the outline of a cloud to be formed, but in three dimensions the cloud can either have volume or the third parameter can be used to represent time to animate the clouds.

The sphere in Figure 3.14(b) is textured using the two dimensional noise texture in 3.14(a) that causes stretching at the poles and a seam where the two edges meet. The sphere in Figure 3.14(d), however, is textured using the three dimensional noise texture in Figure 3.14(c) that eliminates the stretching. To make this work, each point of the texture in Figure 3.14(c) is mapped to a point on the sphere when it is created. This can be applied to other objects as well, such as the torus, Utah Teapot, and cube shown in Figure 3.15.



(a) Two-Dimensional Clouds (b) Three-Dimensional Clouds [13]

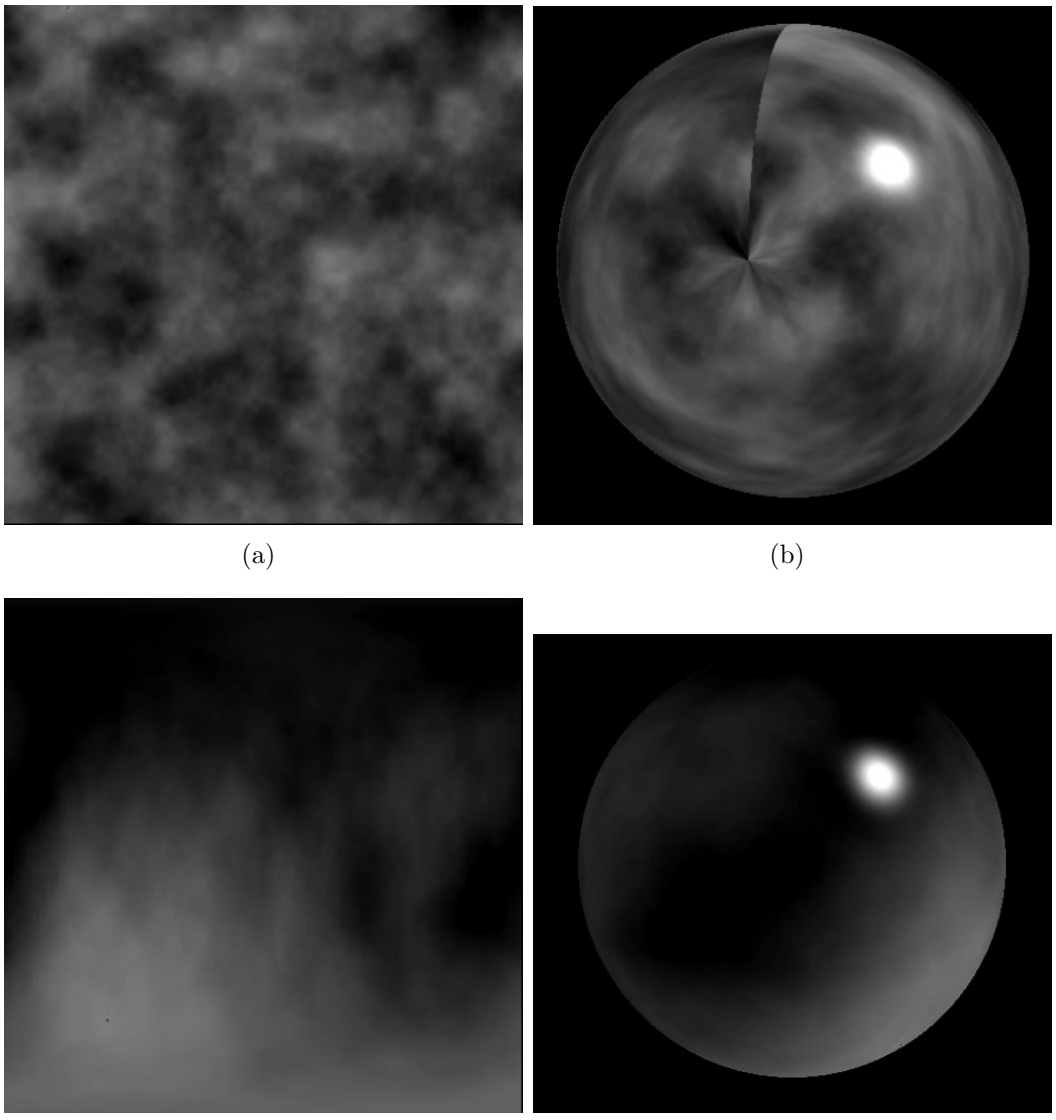
Figure 3.13: Noise Generated Clouds

3.4 Making Noise

So far, noise has been defined and showcased, but what about actually generating the noise?

One-dimensional noise only requires one parameter, commonly an x value. The function calculates the floor(x) and ceil(x) before passing those values to a pseudo-random number generator. The pseudo-random number generator returns values in the interval [-1,1]. The noise value for x is generated by interpolating between the noise values for floor(x) and ceil(x). Because the distance from floor(x) to ceil(x) is one, the interpolation function is applied to the distance, x - floor(x), to find the interpolation value between the noise values. Figure 3.16 shows this process graphically.

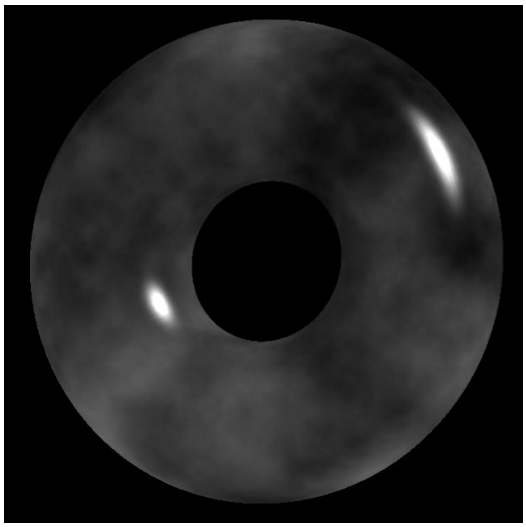
The whole process is computationally efficient because the algorithm has a constant runtime regardless of the x value or seed for the pseudo-random number generator. The most computationally intensive task is calculating the first interpolation value, but this is only done once. However, this benefit starts to diminish as noise is generated in higher dimensions.



(a) (b)

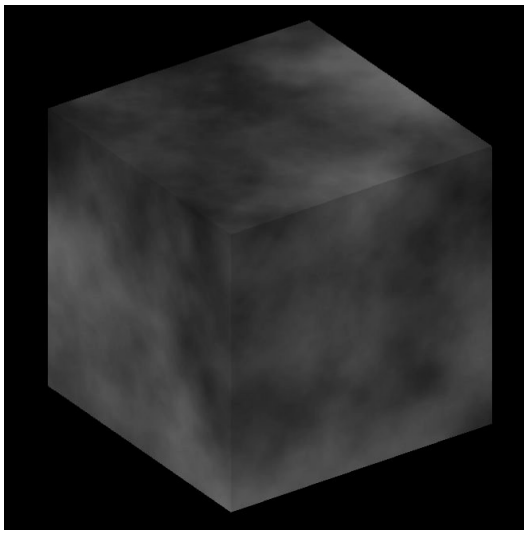
(c) (d)

Figure 3.14: Texture Mapping a Sphere



(a) Torus

(b) UtahTeapot



(c) Cube

Figure 3.15: Texture Mapping Other Three-Dimensional Objects

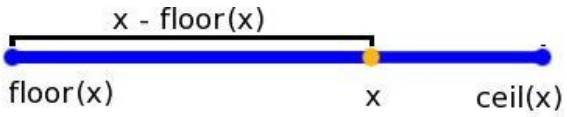


Figure 3.16: Calculating One-Dimensional Noise

In higher dimensions, interpolation between noise values is computed differently than was originally stated in subsection 3.2. In general, it requires four points to perform a cubic interpolation between the noise values. However, in Ken Perlin's original paper on Perlin Noise, he uses the polynomial $3t^2$ - $2t^3$ where t = x - floor(x) because its first derivative is zero at t = 0 and t = 1 allowing for cubic interpolation in almost the same time as linear or cosine interpolation [17]. In his paper about improving Perlin Noise, he found that his original function for interpolation wasn't correct because its second derivative was not zero at both t = 0 and t = 1 and he gives the polynomial $6t^5$ - $15t^4$ + $10t^3$ as a replacement [18].

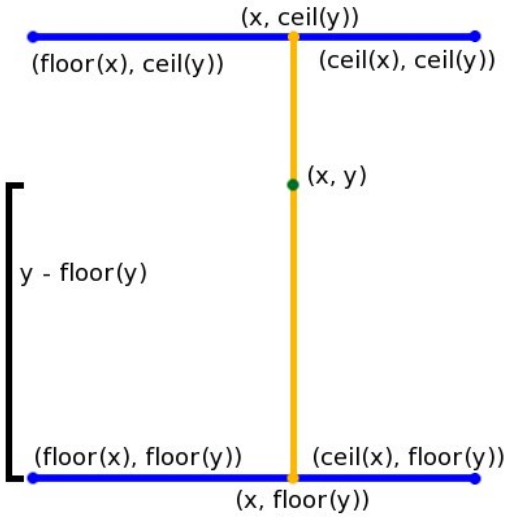


Figure 3.17: Calculating Two-Dimensional Noise

In two dimensions, two parameters are required; x and y. The floor(x), ceil(x), floor(y), and ceil(y) are calculated. Each (x, y) combination, noted by the blue points in Figure 3.17, are passed through the same pseudo-random number generator that is used for one-dimensional noise, causing noise values to be in the interval [-1, 1]. These values are first interpolated along the x-axis, noted by the blue lines on the top and

bottom of Figure 3.17. These values, noted by the yellow points, are then interpolated along the y-axis to produce the final noise value for (x, y) at the green point in Figure 3.17.

The number of operations required when moving from one to two dimensions nearly doubles. There are twice as many floors and ceilings performed and an additional two interpolations. Looking at the function in three dimensions is necessary to determine whether or not the complexity of the function increases exponentially or linearly with the number of dimensions.

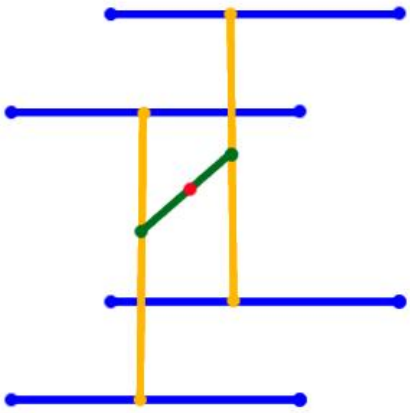


Figure 3.18: Calculating Three-Dimensional Noise

For three dimensions, three parameters are required; x, y, and z. The floor of each parameter is calculated, giving eight points for a unit cube. Each (x, y, z) combination is passed through the pseudo-random number generator, noted by the blue points in Figure 3.18. These values are first interpolated along the x-axis, noted by the blue lines. The x values are interpolated along the y-axis, values are the yellow points and interpolation is then performed along the yellow lines in Figure 3.18. Finally, the values are interpolated along the z-axis, noted by the green line, to generate the resulting noise value for (x, y, z) at the red point in Figure 3.18.

As was the case going from one to two dimensional noise, the number of calculations

nearly doubles going from two to three dimensions. This sets up a pattern to be used for a complexity analysis of the algorithm.

P = The number of dimensions

T = Computation time for one set of parameters

Number of floors = P

Number of points = 2^{P}

Number of interpolations = $2^{P - 1}$

$T = P + 2^{P} + 2^{P - 1}$

$T \approx 2^{P + 1}$

To be complete, the octaves are added into the equation. That gives all the necessary computations for one point.

θ = Number of octaves to calculate

$T = \theta * (P + 2^{P} + 2^{P - 1})$

The time for calculating noise in any number of dimensions is constant, but it is a large constant that is exponential for each dimension. It is important to remember that the function that calculates noise (as described above) is not the main cause of slow computational speeds. It is the set of points for which noise is calculated. For example, calculating noise for a line is linear because only a set of x values are used to calculate the noise values. A texture is quadratic, regardless of the surface it is applied to, because the texture itself is stored in two dimensions. The only cases that cause noise calculations to go into cubic runtime are when a three dimensional set of points is passed to the algorithm.

3.5 Conclusion

This chapter examines Perlin Noise on a theoretical level. Noise can be found both in nature and in electro-magnetic signals. Noise shares a number of properties with sine

and cosine functions, including amplitude, wavelength, and frequency, but it also has properties of its own including persistence, zoom, and octaves. Natural phenomena such as terrain and clouds can be created using noise. Generating noise is an efficient algorithm whose runtime is determined by the number of points for which noise values are generated. This leads us into Chapter 4 where noise is implemented in C++ and GLSL.

CHAPTER 4

Implementations of Perlin Noise

4.1 Overview

This chapter discusses the implementation of two-dimensional and three-dimensional Perlin Noise and rendering it as a texture or with GLSL shaders using OpenGL and C++. It is assumed that the reader can create an OpenGL window, render a texture to a primitive, and understands how to load a GLSL shader for use in a program.

4.2 Two-Dimensional Noise Implementation

Chapter 3 discussed the multiple inputs and functions required to generate a noise value. The main function requires a seed value, a zoom factor, and a persistence value before calculations are performed. For the main function to generate noise values it also needs inputs for the number of octaves and an (x, y) coordinate. The main function uses helper functions to generate the noise for each octave, to generate pseudo-random numbers, and to interpolate between values at each corner of the unit square. These requirements are captured in the following header file for a noise class.

Listing 4.1: Noise Class Header 2D

```
1 class Noise
```

```
2  {
3  public:
4          Noise(int seed, double zoom, double persistence);
5          double perlinNoise2D(int octaves, double x, double
               y);
6
7  private:
8          double noise2D(double x, double y);
9          double findNoise2D(double x, double y);
10         double interpolate(double a, double b, double x);
11
12         int seed;
13         double zoom;
14         double persistence;
15 };
```

The first function of the Noise class that needs to be implemented is the constructor. Its only job is to assign the seed, zoom, and persistence values based on the parameters passed in.

Listing 4.2: Noise Class - Constructor

```
1  Noise::Noise(int seed, double zoom, double persistence)
2  {
3          this->seed = seed;
4          this->zoom = zoom;
5          this->persistence = persistence;
6  }
```

The next function that needs to be implemented is the perlinNoise2D function. Its job is to sum each octave of noise and return the final noise value for the (x, y) coordinate. Each octave is calculated inside a for loop, adding each octave into the returnVal variable. Before the value is returned it is divided by maxH to normalize it in the range [-1, 1].

Listing 4.3: Noise Class - perlinNoise2D

```
1  double Noise::perlinNoise2D(int octaves, double x, double
       y)
2  {
```

```
3           double frequency, amplitude, returnVal = 0.0, maxH
                = 0.0;
4
5           for(int o = 0; o < octaves; o++)
6           {
7                   frequency = pow(2., o);
8                   amplitude = pow(persistence, o);
9                   maxH += amplitude;
10                  returnVal += (noise2D(x * frequency / zoom
                        , y * frequency / zoom)) * amplitude;
11          }
12
13          return (returnVal / maxH);
14  }
```

To calculate the noise value for each octave, perlinNoise2D calls noise2D. The function noise2D calculates noise values at each point of the unit square around the (x, y) coordinate passed to perlinNoise2D as shown in Figure 3.17. The function returns a noise value interpolated from the points of the unit square.

Listing 4.4: Noise Class - noise2D

```
1   double Noise::noise2D(double x, double y)
2   {
3           double floorX = floor(x);
4           double floorY= floor(y);
5
6           double s, t, u, v;
7
8           s = findNoise2D(floorX, floorY);
9           t = findNoise2D(floorX + 1,floorY);
10          u = findNoise2D(floorX, floorY + 1);
11          v = findNoise2D(floorX + 1, floorY + 1);
12
13          double int1 = interpolate(s, t, x - floorX);
14          double int2 = interpolate(u, v, x - floorX);
15
16          return interpolate(int1, int2, y - floorY);
17  }
```

Rather than calculating both the floor and ceiling of the x and y values, only the floors are stored. The ceiling is calculated by adding one to the floor when required

by the calls to findNoise2D. The variables s, t, u, and v each represent a corner of the unit square. The four calls to findNoise2D generate the noise values for each point of the unit square as shown in Figure 3.17. The first interpolation step is used to interpolate along x, for the corners of the square that have the same y-value. The value returned is from the final interpolation between int1 and int2 that store the interpolations along x.

The function findNoise2D generates pseudo-random numbers in the range [-1, 1]. While the random number generator provided by C++ could be used, it isn't as fast as the linear feedback shift register that is implemented in findNoise2D.

Listing 4.5: Noise Class - findNoise2D

```
1 double Noise::findNoise2D(double x, double y)
2 {
3         int n = (int)x + (int)y * seed;
4         n = (n << 13) ^ n;
5         long nn = (60493 * pow(n, 3) + 19990303 * n +
              1376312589) & 0x7fffffff;
6         return 1.0 - ((double)nn / 1073741824.0);
7 }
```

The first step in generating a random value is to combine the x and y portions of the coordinate and the seed value into a single integer. The value is then left shifted 13 bits and bitwise-xor'ed with itself. This value is then modified with a polynomial function and bitwise-and'ed with 0x7fffffff, resulting in a value in the range $[-(2^{30}),2^{30}]$. Each of the coefficients in the polynomial are prime numbers because it ensures maximum periodicity of the generated random value [15]. The return line transforms the value into the range [-1, 1] to make it usable by the other noise functions.

The last function implemented is the interpolation function. The one shown below uses the quintic function discussed in the previous chapter, but it can be interchanged with any of the other interpolation formulas mentioned.

Listing 4.6: Noise Class - interpolate

```
double Noise::interpolate(double a, double b, double x)
{
        //Cosine Interpolation Formulas
        //double ft=(x * 3.14159265358985);
        //double f=(1.0 - cos(ft)) *  0.5;

        //Linear Interpolation Formula
        //double f = x

        //Cubic Interpolation Formula
        //double f = 3 * pow(x, 2) - 2 * pow(x, 3);

        //Quintic Interpolation Formula
        double f = 6 * pow(x, 5) - 15 * pow(x, 4) + 10 *
            pow(x, 3);

        return a * (1.0 - f) + b * f;
}
```

Listing 4.6 concludes coding for the Noise class, but additional code must be developed to display the noise as a texture. The function makeNoiseTexture shown below is responsible for mapping noise values to each point in a texture.

Listing 4.7: 2D Noise Texture Creation

```
void makeNoiseTexture()
{
        GLfloat image[256][256][3];
        Noise myNoise(23467, 32, .5);
        double c;

        for(int i = 0 ; i < 256 ; i++)
        {
                for(int j = 0 ; j < 256 ; j++)
                {
                        c = myNoise.perlinNoise2D(8, i, j)
                            ;
                        c = (c + 1.)/2.;
                        image[i][j][0] = (GLfloat) c;
                        image[i][j][1] = (GLfloat) c;
                        image[i][j][2] = (GLfloat) c;
                }
        }
```

```
18
19              glEnable(GL_TEXTURE_2D);
20
21              glTexImage2D(GL_TEXTURE_2D, 0, 3, 256, 256, 0,
                    GL_RGB, GL_FLOAT, image);
22
23              glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,
                    GL_CLAMP);
24              glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
                    GL_CLAMP);
25              glTexParameterf(GL_TEXTURE_2D,
                    GL_TEXTURE_MAG_FILTER, GL_LINEAR);
26              glTexParameterf(GL_TEXTURE_2D,
                    GL_TEXTURE_MIN_FILTER, GL_LINEAR);
27  }
```

At the beginning of this function there are three variables declared; image, myNoise, and c. The array, image, is used to store the individual RGB values for each point of the texture. The instance of the Noise class, myNoise, is used to generate the noise values for the texture. The variable, c, is used for temporary storage of the generated noise values. The for loop goes through each point of the image array, calculates a noise value, transforms it to the range [0, 1], and uses that value for all three color channels. After the loop, texturing is enabled, the image array is given to OpenGL to use as the default active texture, and then parameters are set for texture wrapping and filtering.

The final step is to map the texture to a square primitive, which results in the image shown in Figure 4.1.

One last thing to note with two-dimensional noise is that the (x, y) coordinates need to be greater than or equal to zero. Figure 4.2 shows the result if negative values are used.

The blocky appearance of the image is caused by two's complement storage of integers in combination with the linear feedback shift method for generating random numbers. In two's complement, negative numbers are stored with a one in the left-most bit. When the left shift occurs during random number generation, this leading one

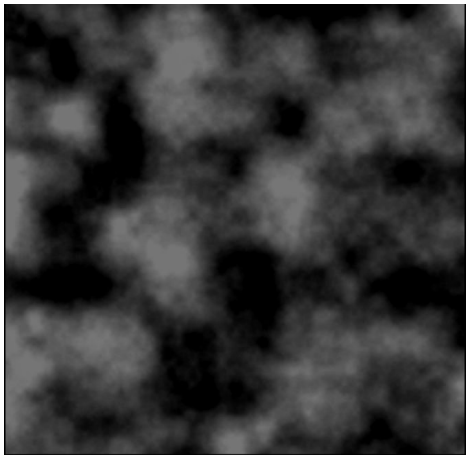


Figure 4.1: Final Product of Implementing 2D Noise

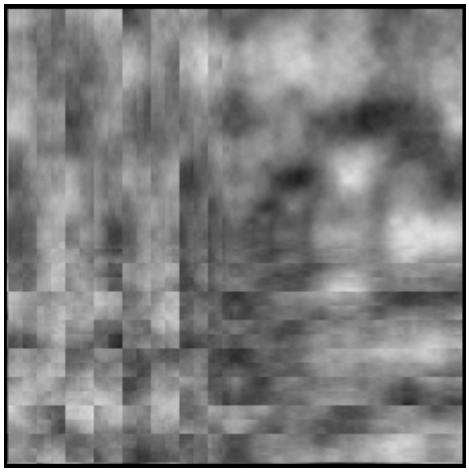


Figure 4.2: Issues With Negative Coordinates

is lost. This creates bars in the textures when one of the coordinate components is negative and a box pattern when both components are negative. These artifacts are shown in Figure 4.2.

4.3 Three-Dimensional Noise Implementation

Two-dimensional noise looks reasonable on a flat surface, but three-dimensional noise is necessary to avoid distortion on a curved surface as shown in Figure 3.14(b). Generating three-dimensional noise requires three functions to be added to the Noise class from the previous section.

Listing 4.8: Noise Class Header 2D and 3D

```
1  class Noise
2  {
3  public:
4          Noise(int seed, double zoom, double persistence);
5          double perlinNoise2D(int octaves, double x, double
               y);
6          double perlinNoise3D(int octaves, double x, double
               y, double z); //Added Function
7
8  private:
9          double noise2D(double x, double y);
10         double findNoise2D(double x, double y);
11         double noise3D(double x, double y, double z); //
               Added Function
12         double findNoise3D(double x, double y, double z);
               //Added Function
13         double interpolate(double a, double b, double x);
14
15         int seed;
16         double zoom;
17         double persistence;
18 };
```

The perlinNoise3D function is the same as perlinNoise2D except that it requires a z component for the third dimension.

Listing 4.9: Noise Class - perlinNoise3D

```
1  double Noise::perlinNoise3D(int octaves, double x, double
       y, double z)
2  {
3          double frequency, amplitude, returnVal = 0.0, maxH
               = 0.0;
4
5          for(int o = 0; o < octaves; o++)
6          {
7                  frequency = pow(2, o);
8                  amplitude = pow(persistence, o);
9                  maxH += amplitude;
10                 returnVal += (noise3D(x * frequency / zoom
                       , y * frequency / zoom, z * frequency /
                        zoom)) * amplitude;
11         }
12
```

```
13          return (returnVal / maxH);
14  }
```

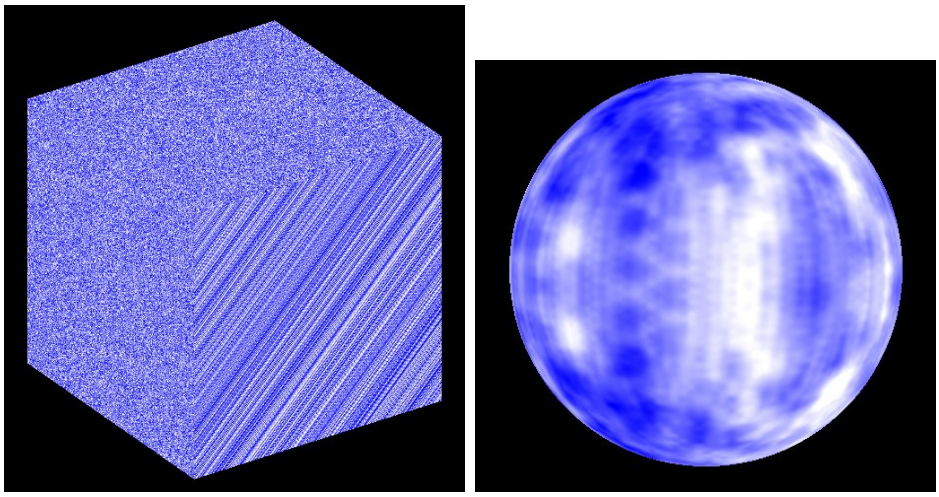
With the addition of the z component, a new function, noise3D, generates the noise value for each octave.

Listing 4.10: Noise Class - noise3D

```
1   double Noise::noise3D(double x, double y, double z)
2   {
3           double floorX = floor(x);
4           double floorY = floor(y);
5           double floorZ = floor(z);
6
7           double s, t, u, v, a, b, c, d;
8
9           s = findNoise3D(floorX, floorY, floorZ);
10          t = findNoise3D(floorX + 1, floorY, floorZ);
11          u = findNoise3D(floorX, floorY + 1, floorZ);
12          v = findNoise3D(floorX + 1, floorY + 1, floorZ);
13          a = findNoise3D(floorX, floorY, floorZ + 1);
14          b = findNoise3D(floorX + 1, floorY, floorZ + 1);
15          c = findNoise3D(floorX, floorY + 1, floorZ + 1);
16          d = findNoise3D(floorX + 1, floorY + 1, floorZ +
                1);
17
18          double int1 = interpolate(s, t, x - floorX);
19          double int2 = interpolate(u, v, x - floorX);
20          double int3 = interpolate(a, b, x - floorX);
21          double int4 = interpolate(c, d, x - floorX);
22
23          double int5 = interpolate(int1, int2, y - floorY);
24          double int6 = interpolate(int3, int4, y - floorY);
25
26          return interpolate(int5, int6, z - floorZ);
27  }
```

As discussed in section 3.4, noise is calculated at $2^{octaves}$ points and the number of interpolations is $2^{octaves}$ - 1. Functionally, the only difference between noise2D and noise3D is the number of components in the given coordinate.

The last function added to the Noise class is the findNoise3D function, which generates a random noise value at each point of a unit cube.



(a) Cube (b) Sphere

Figure 4.3: Incorrect Application of the Seed with 3-Dimensional Noise

Listing 4.11: Noise Class - findNoise3D

```
1 double Noise::findNoise3D(double x, double y, double z)
2 {
3         int n = (int)x * (seed + 5) + (int)y * (seed + 13)
              + (int)z * seed;
4         n = (n << 13) ^ n;
5         long nn = (60493 * pow(n, 3) + 19990303 * n +
              1376312589) & 0x7fffffff;
6         return 1.0 - ((double)nn / 1073741824.0);
7 }
```

The function findNoise3D combines each of the coordinate's components into a single number. Unlike findNoise2D, each component is multiplied by a different number based on the seed value. If the combination is performed as in findNoise2D, the result is two-dimensional noise. The effect can be seen in Figure 4.3 on a cube and a sphere.

Notice that two sides of the cube are of noise at a low zoom factor, but the third side shows streaking. As for the sphere, it has symmetry over its equator. Both of these issues are due to the x and y components being multiplied by the same seed value. Offsetting the seed value as shown in the code is one way around this problem. Another solution is to store three individual seed values, one each for x, y, and z.

Another difference between the two-dimensional and three-dimensional noise functions is how to demonstrate them with a texture. Two-dimensional noise is usually mapped to a flat surface, but three-dimensional noise can be mapped to any surface defined in three-dimensional space.

Listing 4.12: Spherical Noise Texture Creation

```
1  void makeSphereNoiseTexture(double radius)
2  {
3          GLfloat image[256][256][3];
4          Noise myNoise(23467, 32, .5);
5          double c, tempI, tempJ, passI, passJ, z;
6
7          for(int i = 0 ; i < 256 ; i++)
8          {
9                  tempI = -180 + 180.*((double) i / 256);
10
11                 for(int j = 0 ; j < 256 ; j++)
12                 {
13                         tempJ = 360. * ((double) j / 256);
14
15                         passI = radius * sin((tempJ) /
                               180. * M_PI) * sin((tempI) /
                               180. * M_PI) + radius;
16                         passJ = radius * cos((tempJ) /
                               180. * M_PI) * sin((tempI) /
                               180. * M_PI) + radius;
17                         z = radius * cos((tempI) / 180. *
                               M_PI) + radius;
18
19                         c = myNoise.perlinNoise3D(8, passI
                               , passJ, z);
20                         c = (c + 1.)/2.;
21                         image[i][j][0] = (GLfloat) c;
22                         image[i][j][1] = (GLfloat) c;
23                         image[i][j][2] = (GLfloat) c;
24                 }
25         }
26
27         //Texture Creation Parameters
28
29 }
```

Listing 4.12 generates a texture for a sphere. Building from the code for the

makeNoiseTexture function in listing 4.7, a few additions are made. There are an additional five variables; tempI and tempJ are angle values, passI and passJ are the x and y components of the (x, y, z) coordinate for a point on the sphere, and z is the z component. The radius is added to the coordinates to ensure that all the values are positive. Other than these changes, the makeSphereNoiseTexture function is the same as the makeNoiseTexture function.

The last step is to create the sphere by generating a hemisphere and displaying it twice. To aid in this process we can use a small helper class to contain both the vertex and texture coordinates.

Listing 4.13: Vertex Class

```
1  class Vertex
2  {
3  public:
4          double x, y, z;
5          double u, v;
6  };
```

By using the Vertex class, a single array can be used to hold all of the vertices for a triangle strip on the hemisphere to be displayed.

Listing 4.14: Sphere Creation

```
1  const int space = 1;
2  const int VertexCount = (90 / space) * (360 / space) * 4;
3  Vertex sphereVertices[VertexCount];
4
5  void CreateSphere (double radius)
6  {
7          int n = 0;
8
9          for(double i = 0; i < 90; i += space)
10         {
11                 for(double j = 0; j < 360; j += space)
12                 {
13                         sphereVertices[n].x = radius * sin
                               ((j) / 180. * M_PI) * sin((i) /
                                180. * M_PI);
```

```
14                  sphereVertices[n].y = radius * cos
                        ((j) / 180. * M_PI) * sin((i) /
                         180. * M_PI);
15                  sphereVertices[n].z = radius * cos
                        ((i) / 180. * M_PI);
16                  sphereVertices[n].v = (2 * i) /
                        360.;
17                  sphereVertices[n].u = (j) / 360.;
18                  n++;
19
20                  sphereVertices[n].x = radius * sin
                        ((j) / 180. * M_PI) * sin((i +
                        space) / 180. * M_PI);
21                  sphereVertices[n].y = radius * cos
                        ((j) / 180. * M_PI) * sin((i +
                        space) / 180. * M_PI);
22                  sphereVertices[n].z = radius * cos
                        ((i + space) / 180. * M_PI);
23                  sphereVertices[n].v = (2 * (i +
                        space)) / 360.;
24                  sphereVertices[n].u = (j) / 360.;
25                  n++;
26
27                  sphereVertices[n].x = radius * sin
                        ((j + space) / 180. * M_PI) *
                        sin((i) / 180. * M_PI);
28                  sphereVertices[n].y = radius * cos
                        ((j + space) / 180. * M_PI) *
                        sin((i) / 180. * M_PI);
29                  sphereVertices[n].z = radius * cos
                        ((i) / 180. * M_PI);
30                  sphereVertices[n].v = (2 * i) /
                        360.;
31                  sphereVertices[n].u = (j + space)
                        / 360.;
32                  n++;
33
34                  sphereVertices[n].x = radius * sin
                        ((j + space) / 180. * M_PI) *
                        sin((i + space) / 180. * M_PI);
35                  sphereVertices[n].y = radius * cos
                        ((j + space) / 180. * M_PI) *
                        sin((i + space) / 180. * M_PI);
36                  sphereVertices[n].z = radius * cos
                        ((i + space) / 180. * M_PI);
37                  sphereVertices[n].v = (2 * (i +
                        space)) / 360.;
```

```
38                              sphereVertices[n].u = (j + space)
                                    / 360.;
39                              n++;
40                      }
41              }
42      }
```

For simplicity, the sphereVertices array and the space and VertexCount variables are left as global variables. The variable, space, represents the space in degrees between vertices for the sphere. Because texture coordinates must lie in the range [0, 1], the same formula used to map each value of the noise texture to the sphere is used to calculate the texture coordinates for each vertex.

All that's left to do is display the sphere with the texture. The code shown below is placed inside an already existing display function.

Listing 4.15: Spherical Noise Texture Display

```
1  double length = 0.0;
2  glActiveTexture(GL_TEXTURE0); //Gets the active texture
3  glBegin (GL_TRIANGLE_STRIP);
4
5  for ( int i = 0; i <= VertexCount; i++)
6  {
7          glTexCoord2f (sphereVertices[i].u, sphereVertices[
               i].v);
8          length = sqrt(sphereVertices[i].x * sphereVertices
               [i].x + sphereVertices[i].y * sphereVertices[i
               ].y + sphereVertices[i].z * sphereVertices[i].z
               );
9          glNormal3f(sphereVertices[i].x / length,
               sphereVertices[i].y / length, -sphereVertices[i
               ].z / length);
10         glVertex3f (sphereVertices[i].x, sphereVertices[i
               ].y, -sphereVertices[i].z);
11 }
12
13 for ( int i = 0; i <= VertexCount; i++)
14 {
15         glTexCoord2f (sphereVertices[i].u, 1. -
               sphereVertices[i].v);
```

```
16          length = sqrt(sphereVertices[i].x * sphereVertices
                [i].x + sphereVertices[i].y * sphereVertices[i
                ].y + sphereVertices[i].z * sphereVertices[i].z
                );
17          glNormal3f(sphereVertices[i].x / length,
                sphereVertices[i].y / length, sphereVertices[i
                ].z / length);
18          glVertex3f (sphereVertices[i].x, sphereVertices[i
                ].y, sphereVertices[i].z);
19  }
20  glEnd();
```

When all the above code is implemented and the sphere's radius is set to eight then the output should look similar to the image in Figure 4.4. The view chosen is the top of the sphere.

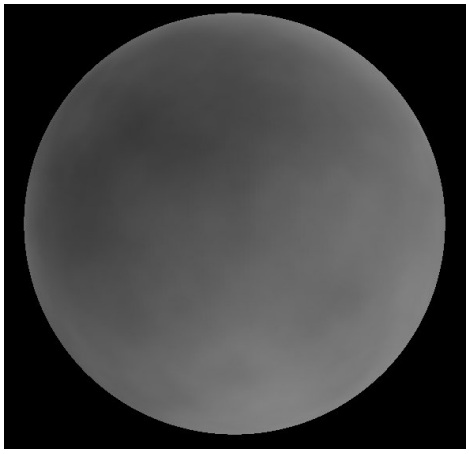


Figure 4.4: 3D Noise Mapped to a Sphere

4.4 Noise Implementation in GLSL

Experimentation with generating noise for larger texture sizes or a high number of octaves can easily show that the process is computationally expensive. Since no point of the texture depends on another, each point can be computed in parallel. Modern CPU's may have two, three, four, six, or more cores to process instructions in parallel, but there is another part of the computer that is built to run highly parallelized

operations; the graphics card. By using GLSL shaders, code can be written for the graphics card to run across hundreds of cores at the same time, thus generating noise at a much faster rate than the CPU. This allows noise to be animated and also provides the means for texturing a complex object with many polygons in real time.

The Noise class of the previous two sections can be used with a few syntax changes for GLSL. For the GLSL 1.20 implementation there are two sections to develop; the vertex shader and the fragment shader. Below is the vertex shader in listing 4.16.

Listing 4.16: GLSL - Noise.vs

```
varying vec4 vertexPosition;
varying vec3 vertexNormal;
varying vec3 lightPosition;

void main()
{
        vertexNormal = normalize(gl_NormalMatrix *
           gl_Normal);
        lightPosition = normalize(gl_LightSource[0].
           position.xyz);
        vertexPosition = gl_Vertex;
        gl_Position = ftransform();
}
```

The three varying variables are shared with the associated fragment shader. As they are labeled, vertexPosition is the vertex passed to OpenGL before being modified by the Modelview-Projection matrix, vertexNormal is the normal vector after being transformed by the Normal Matrix, and lightPosition is the normalized vector from the vertex to the light source. After assigning values to the varying variables, the ftransform function is applied to transform the vertex by the Modelview-Projection matrix.

Listing 4.17: GLSL - Noise.fs - Function Headers Uniform Variables and Varying Variables

```
#version 120
#extension GL_EXT_gpu_shader4: enable
```

```
3
4  varying vec3 lightPosition;
5  varying vec3 vertexNormal;
6  varying vec4 vertexPosition;
7
8  float perlinNoise3D(float x, float y, float z);
9  float noise3D(float x, float y, float z);
10 float findNoise3D(int x, int y, int z);
11 float interpolate(float a,float b,float x);
12
13 uniform ivec3 seed;
14 uniform int octaves;
15 uniform float zoom;
16 uniform float persistence;
```

The code in listing 4.17 is only a portion of the fragment shader for noise. The first two lines are used to set the GLSL version and enable an extension that allows for bitwise operations to be performed on integers. Without this ability, the linear feedback shift register code from subsection 4.3 causes a compile error. The next three lines are the varying variables passed in from the vertex shader. These values are interpolated by OpenGL using each vertex of the triangle where the fragment is located. Following the varying variables are the function declarations for noise. Finally, there are the uniform variables that are passed in from the main program.

Listing 4.18: GLSL - Noise.fs - perlinNoise3D

```
1  float perlinNoise3D(float x, float y, float z)
2  {
3          float frequency, amplitude, returnVal = 0.0, maxH
              = 0.0;
4
5          for(int o = 0; o < octaves; o++)
6          {
7                  frequency = pow(2, o);
8                  amplitude = pow(persistence, o);
9                  maxH += amplitude;
10                 returnVal += (noise3D(x * frequency / zoom
                       , y * frequency / zoom, z * frequency /
                        zoom)) * amplitude;
11         }
12
```

```
13          return (returnVal / maxH);
14  }
```

The perlinNoise3D function uses float instead of double type because double is not a recognized datatype in GLSL. Persistence, zoom, and the number of octaves are passed into the program as uniform variables so they are global in relation to perlinNoise3D.

Listing 4.19: GLSL - Noise.fs - noise3D

```
1   float noise3D(float x, float y, float z)
2   {
3           vec3 Pos = vec3(x, y, z);
4           ivec3 Flr = ivec3(floor(Pos));
5
6           float s, t, u, v, a, b, c, d;
7
8           s = findNoise3D(Flr.x, Flr.y, Flr.z);
9           t = findNoise3D(Flr.x + 1, Flr.y, Flr.z);
10          u = findNoise3D(Flr.x, Flr.y + 1, Flr.z);
11          v = findNoise3D(Flr.x + 1, Flr.y + 1, Flr.z);
12          a = findNoise3D(Flr.x, Flr.y, Flr.z + 1);
13          b = findNoise3D(Flr.x + 1, Flr.y, Flr.z + 1);
14          c = findNoise3D(Flr.x, Flr.y + 1, Flr.z + 1);
15          d = findNoise3D(Flr.x + 1, Flr.y + 1, Flr.z + 1);
16
17          float int1 = interpolate(s, t, fract(x));
18          float int2 = interpolate(u, v, fract(x));
19          float int3 = interpolate(a, b, fract(x));
20          float int4 = interpolate(c, d, fract(x));
21
22          float int5 = interpolate(int1, int2, fract(y));
23          float int6 = interpolate(int3, int4, fract(y));
24
25          return interpolate(int5, int6, fract(z));
26  }
```

The noise3D function has a few changes. By using the vec3 datatype, a three component vector of floating point numbers, there is less code to find the floor of the (x, y, z) coordinate. Using fract, which returns the fractional portion of a number, it is

no longer necessary to manually calculate x - floor(x). These changes take advantage of the GLSL built-in functions for a slight increase in speed.

Listing 4.20: GLSL - Noise.fs - findNoise3D

```
float findNoise3D(int x, int y, int z)
{
        int n = x * seed.x + y * seed.y + z * seed.z;
        n = (n << 13) ^ n;
        int nn = (60493 * (n * n * n) + 19990303 * n +
            1376312589) & 0x7fffffff;
        return 1.0 - (nn / 1073741824.0);
}
```

With this version of findNoise3D, there are separate seed values for the x, y, and z components. Except for that change, the function remains the same as the C++ version other than switching to the correct datatypes.

Listing 4.21: GLSL - Noise.fs - interpolate

```
float interpolate(float a, float b, float x)
{
        float f = 6 * pow(x, 5) - 15 * pow(x, 4) + 10 *
            pow(x, 3);
        return mix(a, b, f);
}
```

Similar to the noise3D function, the only changes to the interpolate function are to the datatypes and using GLSL built-in functions. The mix function is the built-in function for linear interpolation between two values. With all the noise functions rewritten for GLSL, the main function can be written.

Listing 4.22: GLSL - Noise.fs - main

```
void main()
{
        float c = perlinNoise3D(vertexPosition.x,
            vertexPosition.y, vertexPosition.z);
        c = (c + 1) / 2;
```

```
5
6           vec4 color = vec4(c, c, c, 1.);
7
8           vec4 ambientColor = color * (gl_LightSource[0].
               ambient + gl_LightModel.ambient);
9           vec4 diffuseColor = color * gl_LightSource[0].
               diffuse;
10          float diffuseValue = max(dot(vertexNormal,
               lightPosition), 0.0);
11
12          gl_FragColor = ambientColor + diffuseColor *
               diffuseValue;
13  }
```

The noise value for the fragment is calculated by using vertexPosition and assigned to c. c is then used to determine the color of the fragment before lighting is applied. The final fragment color is calculated using the Phong Illumination Model with the specular and attenuation components left out. In order to use this shader program it must be applied to an object.

Listing 4.23: Portion of Display Loop for GLSL

```
1  noiseShader.bind();
2  glUniform3i(glGetUniformLocation(noiseShader.id(), "seed")
       , 23467, 97, 21349);
3  glUniform1i(glGetUniformLocation(noiseShader.id(), "
       octaves"), 8);
4  glUniform1f(glGetUniformLocation(noiseShader.id(), "zoom")
       , 4);
5  glUniform1f(glGetUniformLocation(noiseShader.id(), "
       persistence"), .5);
6
7  glutSolidTeapot(6);
8  noiseShader.unbind();
```

In this example, the Utah Teapot included with GLUT is used. Using the same parameters as shown above for the uniform variables, the teapot should look similar to the image shown in Figure 4.5.

Figure 4.5: Utah Teapot Textured with Three-Dimensional Noise Using GLSL

4.5 Conclusion

This chapter examined the implementation of two-dimensional and three-dimensional noise using both C++ and GLSL. It discussed the problems associated with applying the seed improperly and when negative numbers are passed to the linear feedback shift register for random number generation. This chapter serves as a building block for Chapter 5, which focuses on using these implementations to create various textures and effects.

CHAPTER 5

Applications of Perlin Noise

5.1 Overview

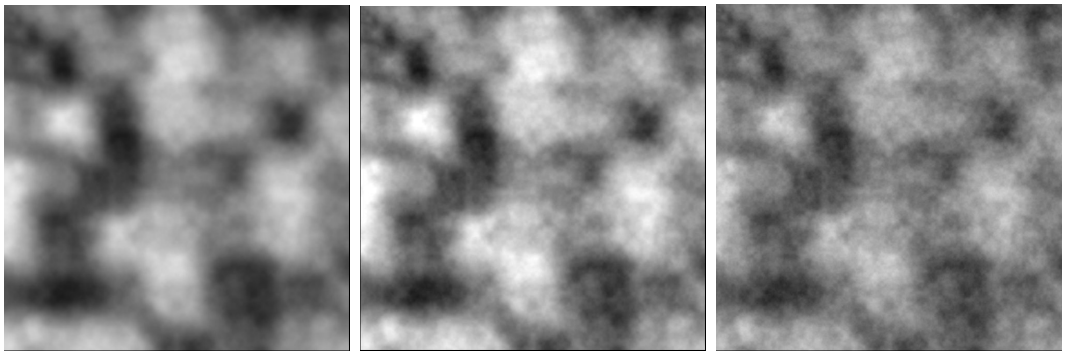
This chapter uses the implementations for noise from Chapter 4. This discussion focuses on the equations used to sum the noise values in order to create a variety of textures and effects.

5.2 Varying Persistence Levels

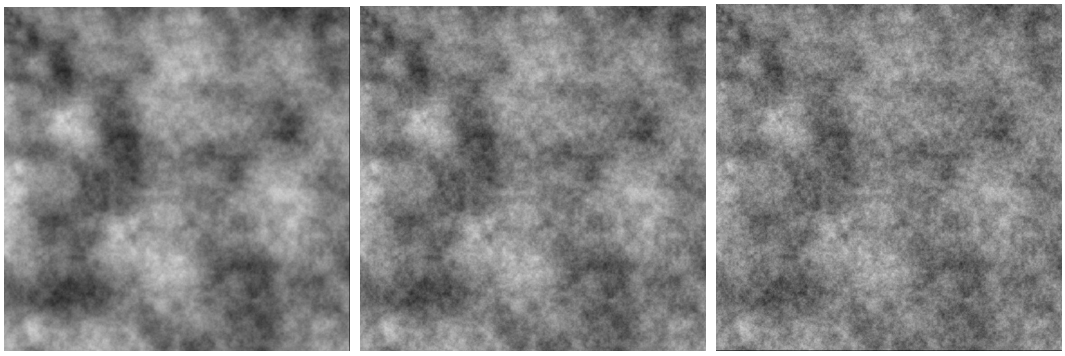
One of the simpler ways of changing how a noise function looks is by varying the persistence level. Different levels of persistence are shown in Figure 5.1.

The lower persistence levels (P = .1, .2, .3) are given their shape from the lower octaves because of the miniscule amplitude at higher octaves. The middle persistence levels (P = .4, .5, .6) are less blocky because the amplitude of the higher octaves is large enough to have an effect. The higher persistence levels (P = .7, .8, .9) leave the blockiness behind and begin to resemble non-coherent noise because each successive octave has an amplitude that is nearly as much as the previous octave.

(a) P = .1 (b) P = .2 (c) P = .3



(d) P = .4 (e) P = .5 (f) P = .6



(g) P = .7 (h) P = .8 (i) P = .9

Figure 5.1: Differences in Persistence

5.3 Applying Mathematical Functions

Another way to change the look of a noise function is to apply an equation such as cosine, tangent, or absolute value. Figure 5.2 shows this effect by using the Noise class implementation from subsection 4.2. The texture dimension is changed from 256x256 to 768x768 for higher quality and the zoom factor is increased to 128 so that differences are more noticeable.

Looking at Figure 5.2(b), the image looks washed out compared to Figure 5.2(a). Because of where values for cosine lie on the unit circle, the noise values are transformed from the range [0, 1] to [.54, 1]. This greatly increases the brightness of the image.

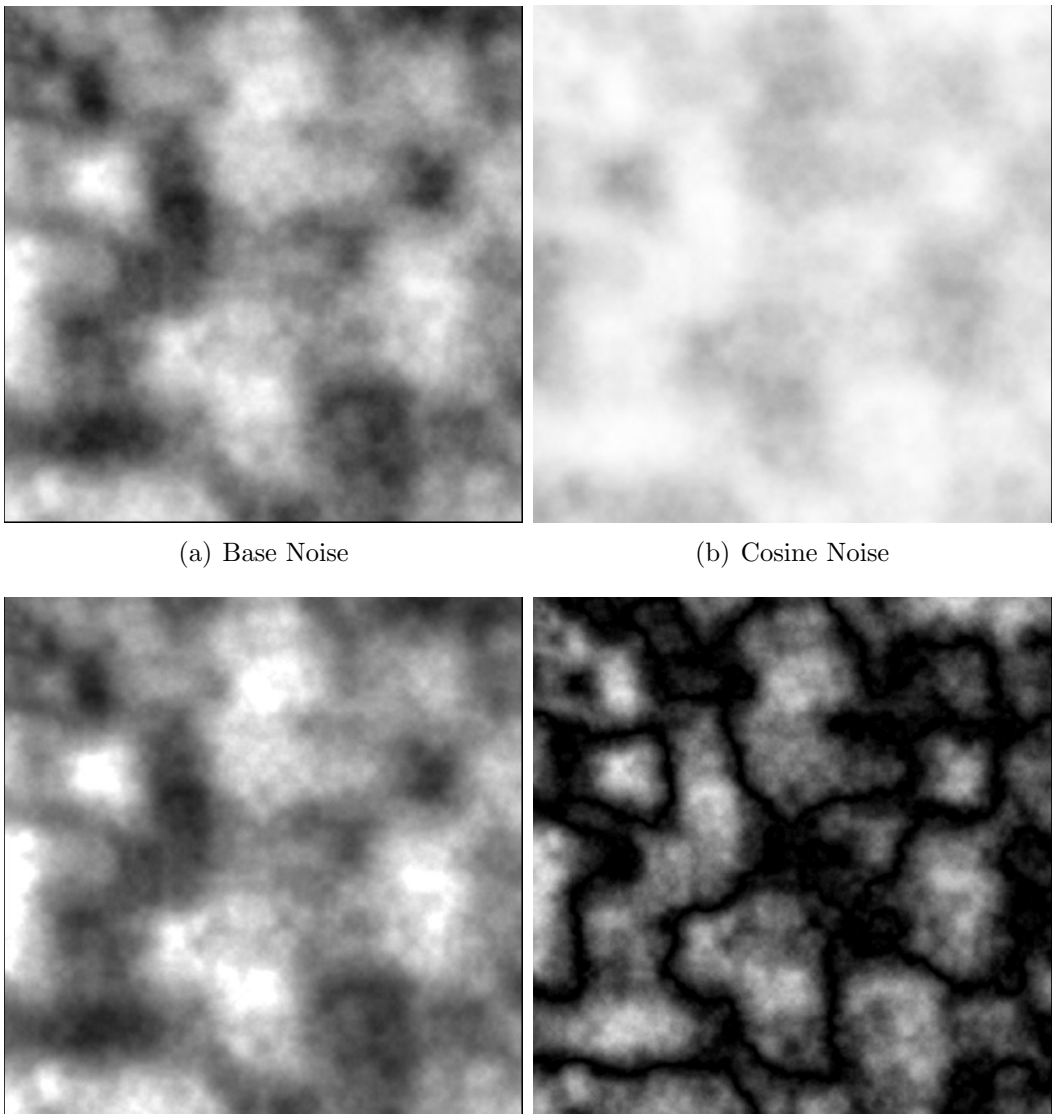
In Figure 5.2(c), the image is a higher contrast version of Figure 5.2(a). This is again because of where values lie on the unit circle. The noise values are transformed from the range [0, 1] to [0, 1.56]. RGB values lie in the range [0, 1], so the values in the range [1, 1.56] are clamped to one and the difference between the values lying in the range [0, 1] after the transformation become more pronounced.

Because the absolute value function has no effect on values already lying in the range [0, 1], the noise values were left in the range [-1, 1] to be transformed to the range [0, 1]. Pronounced lines are displayed in Figure 5.2(d) where the original values were closest to .5.

5.4 Adding Color

Greyscale textures are useful for analysis of different aspects of noise functions such as zoom level and persistence, but new effects come into play when colors are used.

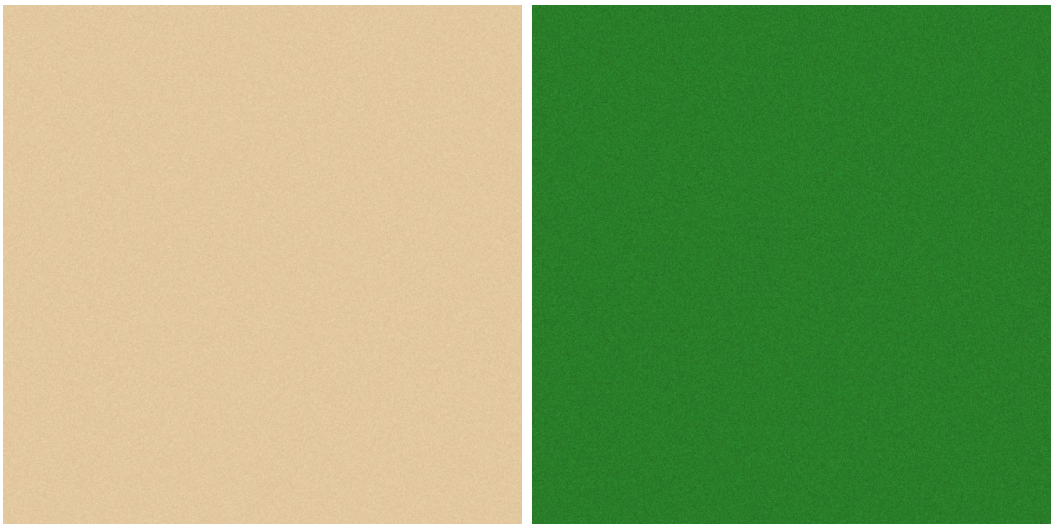
The sand and grass textures shown in Figure 5.3 are simple examples of what can be done with noise. Any two colors can easily be blended using the interpolation function of the Noise class.



(a) Base Noise (b) Cosine Noise

(c) Tangent Noise (d) Absolute Value Noise

Figure 5.2: Applying Mathematical Functions to Noise



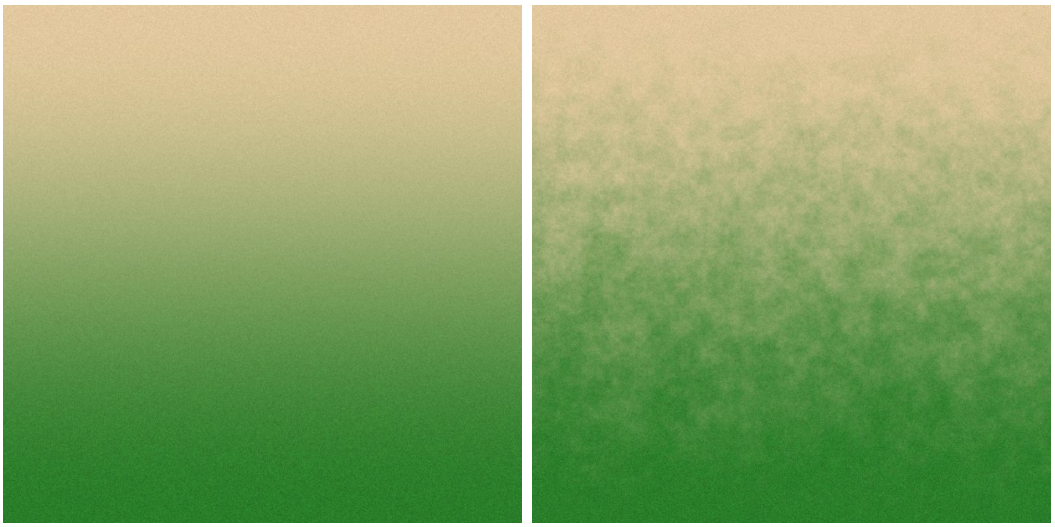
(a) Sand (Seed: 1234976, Zoom: .125, Persistence: .9)

(b) Grass (Seed: 2354583, Zoom: 1, Persistence: .7)

Figure 5.3: Simple Noise Textures

Listing 5.1: 2D Grass Texture Creation

```
1  Noise grass(2354583, 1, .7);
2
3  void makeGrassTexture()
4  {
5          GLfloat image[768][768][3];
6
7          float lightGreen[3] = {50. / 255., 156. / 255.,
               50. / 255.};
8          float darkGreen[3] = {34. / 255., 96. / 255., 34.
               / 255.};
9
10         double c;
11
12         for(int i = 0; i < 768; i++)
13         {
14                 for(int j = 0; j < 768; j++)
15                 {
16                         c = (1. + grass.perlinNoise2D(8, i
                               , j)) / 2.;
17                         image[i][j][0]= (GLfloat)
                               interpolate(lightGreen[0],
                               darkGreen[0], c);
18                         image[i][j][1]= (GLfloat)
                               interpolate(lightGreen[1],
                               darkGreen[1], c);
```



(a) Linear Blending (b) Noise Blending

Figure 5.4: Texture Blending Comparison

```
19                                  image[i][j][2]= (GLfloat)
                                        interpolate(lightGreen[2],
                                        darkGreen[2], c);
20                              }
21                      }
22
23                      //Texture Creation Parameters
24              }
```

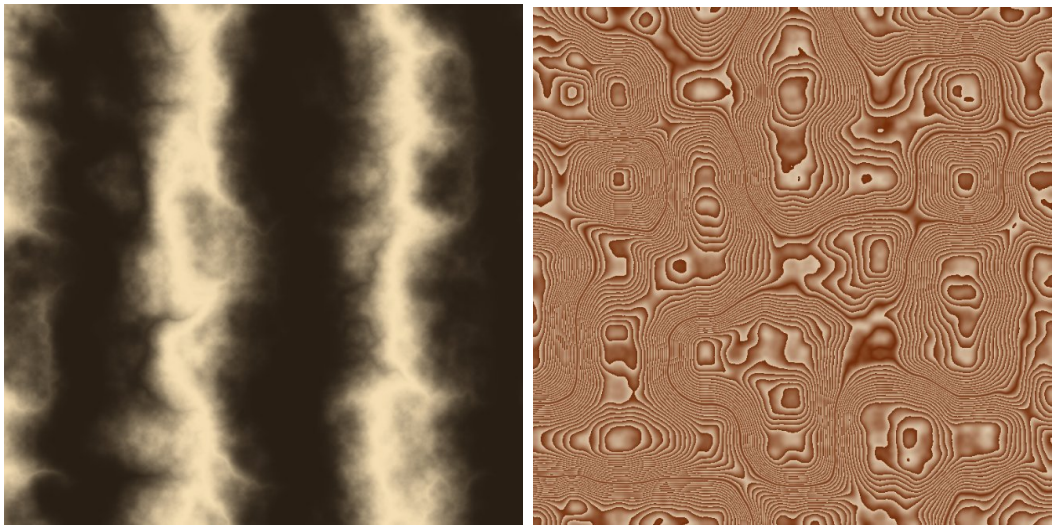
The code segment of listing 5.1 blends between the two colors of green using the noise value, c, to interpolate between the colors. The texture for grass looks far better than a flat green color, but using noise values for interpolation doesn't need to stop with colors for a single texture.

Figure 5.4 compares generic linear blending to noise blending of the sand and grass textures shown in Figure 5.3. When going from grass to sand in real life there isn't a linear blend between the two, so it makes sense to use blending based on noise for a higher degree of realism.

Listing 5.2: Noise Blending Between Grass and Sand

```
1 Noise sand(1234976, .125, .9);
```

```
2  Noise grass(2354583, 1, .7);
3  Noise mix(4567345, 32, .8);
4
5  void noiseBlendTexture()
6  {
7          GLfloat image[768][768][3];
8
9          float lightBrown[3] = {245. / 255., 222. /
               255.,179. / 255.};
10         float darkBrown[3] = {210. / 255.,180. / 255.,140.
               / 255.};
11         float lightGreen[3] = {50. / 255., 156. / 255.,
               50. / 255.};
12         float darkGreen[3] = {34. / 255., 96. / 255., 34.
               / 255.};
13
14         double c, x, myI, myJ, a, b;
15
16         for(int i = 0; i < 768; i++)
17         {
18                 for(int j = 0; j < 768; j++)
19                 {
20                         myI = i, myJ = j;
21
22                         c = (1. + sand.perlinNoise2D(8,
                               myI, myJ)) / 2.;
23                         x = (1. + grass.perlinNoise2D(8,
                               myI, myJ)) / 2.;
24                         a = clamp(myI / 768. + .2 * (mix.
                               perlinNoise2D(8, myI, myJ)),
                               0., 1.);
25
26                         image[i][j][0] = (GLfloat)
                               interpolate(interpolate(
                               lightBrown[0], darkBrown[0], c)
                               , interpolate(lightGreen[0],
                               darkGreen[0], x), a);
27                         image[i][j][1] = (GLfloat)
                               interpolate(interpolate(
                               lightBrown[1], darkBrown[1], c)
                               , interpolate(lightGreen[1],
                               darkGreen[1], x), a);
28                         image[i][j][2] = (GLfloat)
                               interpolate(interpolate(
                               lightBrown[2], darkBrown[2], c)
                               , interpolate(lightGreen[2],
                               darkGreen[2], x), a);
```



(a) Marble (b) Wood

Figure 5.5: Advanced Noise Texturing

```
                    }
            }

            //Texture Creation Parameters
}
```

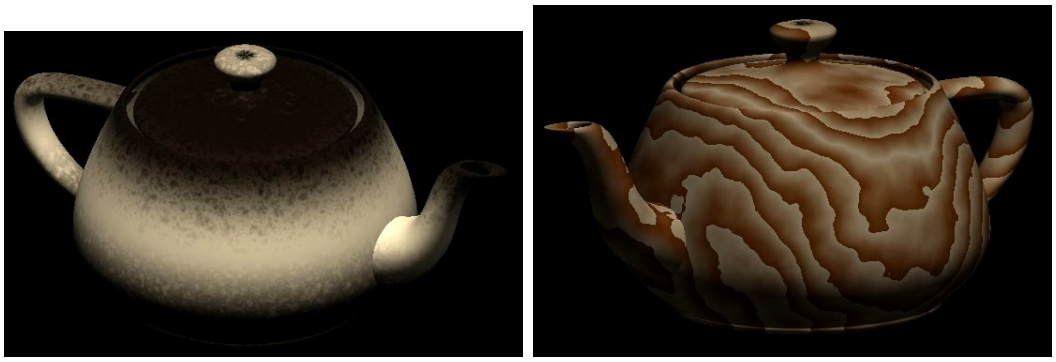
To implement the noise-based blending technique, another instance of the Noise class is used to offset the linear blend value.

The sand and grass textures previously displayed are simple examples of the kinds of textures noise produces. Figure 5.5 displays more advanced textures made with noise.

Figure 5.5(a) is a marble design that is created by using the absolute value of the noise function as an offset to a sine function based on the j value (horizontal placement in the texture). The image in Figure 5.5(b) is a wood design created by multiplying the absolute value of the noise function with a constant offset and then taking the fractional part of the result.

Listing 5.3: Marble and Wood Texture Creation

```
//Marble Design
```



(a) Marble (b) Wood

Figure 5.6: Advanced GLSL Noise Texturing

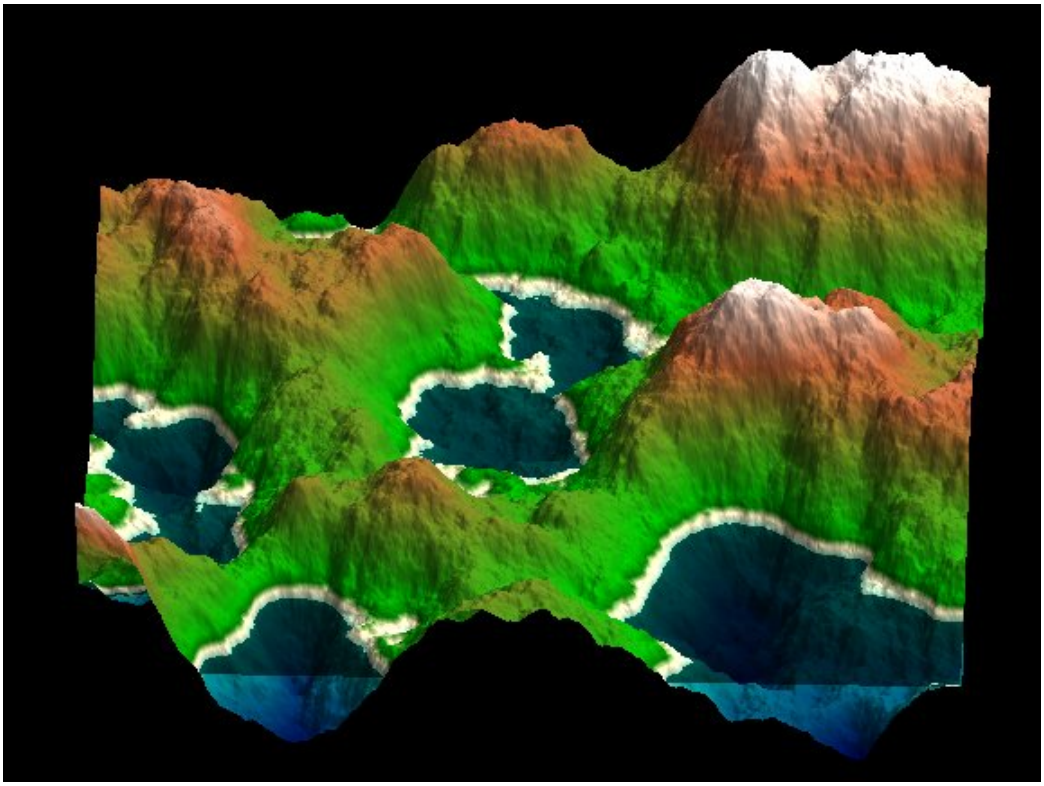
```
Noise myNoise(23467, 128, .5);
c = sin(j / 96. + fabs(myNoise.perlinNoise2D(8, i, j)));

//Wood Design
Noise myNoise(23467, 128, .25);
c = fabs(myNoise.perlinNoise2D(8, i, j)) * 20;
c -= int(c);
```

The functions for these textures can also be implemented in GLSL for use with more interesting objects as shown in Figure 5.6 where they are applied to a Utah Teapot.

5.5 Height Maps

Moving away from texturing for a moment, another application of noise is generating height maps to create virtual terrains. Nothing extra needs to be done with the Noise class to generate a height map because the heights are determined by passing the x and z components to generate the noise value used by the y value for the height.

Figure 5.7 shows the end result of using noise to generate height maps. Figure 5.7(a) was created using only one noise function causing features at different heights to look similar. Figure 5.7(b) uses multiple noise functions to create a variety of geographical features at different elevation levels.



(a) Single Noise Function

(b) Multiple Noise Functions

Figure 5.7: Terrain From Height Maps

5.6 Animated Textures

One of the main advantages of texturing using GLSL is the ability to create animated textures. Examples are clouds in the background (shown in Figure 5.7(b)), fire, and water.

The first step in making an animated texture is to add a uniform variable for time. When the time variable is updated, the noise function generates a different value for a fragment. This allows for two different approaches to animating the noise; making time another dimension of the noise function or adding it to the values already being passed to the noise function.

Listing 5.4: How To Animate Noise

```
1 // Using another dimension
2 float c = perlinNoise4D(vertexPosition.x, vertexPosition.y
    , vertexPosition.z, time);
3
4 // Adding time into existing values
5 float c = perlinNoise3D(vertexPosition.x + time,
    vertexPosition.y + time, vertexPosition.z + time);
```

As shown in subsection 3.4, increasing the number of dimensions of a noise function doubles the constant time needed to calculate a single noise value. In the case of three-dimensional noise, we move from interpolating a cube to a four-dimensional hypercube. Simply adding the time value to the coordinates in three-dimensions is no better because the image remains static. The solution to this is to make different octaves of the noise function move in different directions by using directional vectors. The most efficient way to implement this is to hard code the vectors into the shader to avoid running into the limit of uniform variables that can be passed from the main program.

Listing 5.5: GLSL - Noise Animation

```
1  uniform float time;
2
3  float perlinNoise3D(float x, float y, float z)
4  {
5          float frequency, amplitude, returnVal = 0.0, maxH
               = 0.0;
6
7          vec3 dir1 = time * vec3(1, 0, 0);
8          vec3 dir2 = time * vec3(.5, .25, .25);
9          vec3 dir3 = time * vec3(0, .5, .5);
10         vec3 dir4 = time * vec3(.33, 2., 1.);
11
12
13         for(int o = 0; o < octaves; o++)
14         {
15                 frequency = pow(2, o);
16                 amplitude = pow(persistence, o);
17                 maxH += amplitude;
18                 if(o < .25 * octaves)
19                         returnVal += (noise3D(x *
                              frequency / zoom + dir1.x, y *
                              frequency / zoom + dir1.y, z *
                              frequency / zoom + dir1.z)) *
                              amplitude;
20                 else if(o < .5 * octaves)
21                         returnVal += (noise3D(x *
                              frequency / zoom + dir2.x, y *
                              frequency / zoom + dir2.y, z *
                              frequency / zoom + dir2.z)) *
                              amplitude;
22                 else if(o < .75 * octaves)
23                         returnVal += (noise3D(x *
                              frequency / zoom + dir3.x, y *
                              frequency / zoom + dir3.y, z *
                              frequency / zoom + dir3.z)) *
                              amplitude;
24                 else
25                         returnVal += (noise3D(x *
                              frequency / zoom + dir4.x, y *
                              frequency / zoom + dir4.y, z *
                              frequency / zoom + dir4.z)) *
                              amplitude;
26         }
27
28         return (returnVal / maxH);
29 }
```

In the perlinNoise3D function of listing 5.5, every quarter of the total number of octaves has its own direction vector. The uniform variable, time, is multiplied with each of the direction vectors so that the offset is different when the time is updated by the main program.

Now that the basis for animated noise is written it can be used to implement functions for clouds, fire, and water. The first step in recreating any natural phenomenon is to study pictures of the feature to be implemented.

Looking at Figure 5.8, the color of the clouds is determined by two main factors; thickness and whether or not sunlight is bouncing off the cloud. While checking to see if sunlight bounces off a point of a cloud requires a normal vector, thickness can be implemented using only noise values.

Three colors are needed to implement noise-based clouds; sky color, dark cloud color, and light cloud color. Noise values are used to determine if the fragment color is of the sky or a blend between the light and dark cloud colors.

Listing 5.6: GLSL - Animated Clouds

```
1  void main()
2  {
3          float c = perlinNoise3D(vertexPosition.x,
               vertexPosition.y, vertexPosition.z);
4          c = (c + 1) / 2;
5
6          vec4 sky = vec4(0, .25, 1, 1);
7          vec4 cloudDark = vec4(.5, .5, .5, 1);
8          vec4 cloudLight = vec4(.9, .9, .9, 1);
9          vec4 color;
10
11         if(c < .5)
12                 color = sky;
13         else if(c < .55)
14                 color = interpolate(sky, cloudLight, 20 *
                       (c - .5));
15         else if(c < .75)
16                 color = interpolate(cloudLight, cloudDark,
                        5 * (c - .55));
17         else
```

(a)

(b)

Figure 5.8: Pictures of Real Clouds

```
18                  color = cloudDark;
19
20              vec4 ambientColor = color * (gl_LightSource[0].
                    ambient + gl_LightModel.ambient);
21              vec4 diffuseColor = color * gl_LightSource[0].
                    diffuse;
22              float diffuseValue = max(dot(vertexNormal,
                    lightPosition), 0.0);
23
24              gl_FragColor = ambientColor + diffuseColor *
                    diffuseValue;
25  }
```

Instead of using c to determine the greyscale value, c determines the percentage of each color used and to interpolate between these colors. In order to ensure proper blending between each color, the x parameter of each interpolate statement needs to be changed so that it spans the range [0, 1]. Implementing the above code with the animated noise code shown previously yields something similar to Figure 5.9.

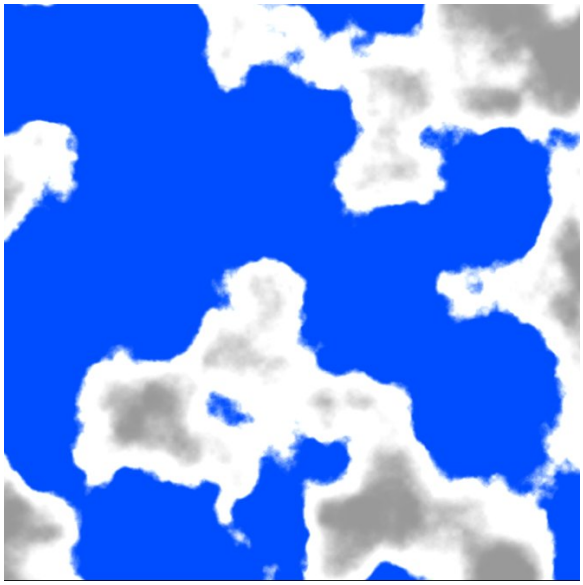


Figure 5.9: Snapshot of Animated Clouds

Moving onto fire, the process again starts with a picture of actual fire before emulating it with noise.

While Figure 5.10 only displays fire in a fireplace, there are obviously many other kinds of fire depending on the type and size of the fuel source used.

Figure 5.10: Fire in a Fireplace [25]

Taking a closer look at Figure 5.10, there are three colors of the fire as well as a background color that need to be blended. The flames are cooler the farther they are from the source and this is modeled with a temperature gradient. Then because heat rises, the direction vectors need to point in an upwards direction.

Listing 5.7: GLSL - Animated Fire

```
1  //Direction Vectors
2  vec3 dir1 = time * vec3(0, -1, 0);
3  vec3 dir2 = time * vec3(.5, -1, .25);
4  vec3 dir3 = time * vec3(.66, -1, .5);
5  vec3 dir4 = time * vec3(.35, -1, .9);
6
7  void main()
8  {
9          float temperature = (distance(vertexPosition.y,
               10.) + .5) / 20.;
10
11         vec4 color;
12         vec4 lightOrange = vec4(252. / 255., 238. / 255.,
               209. / 255., 1);
13         vec4 middleOrange = vec4(247. / 255., 209. / 255.,
               76. / 255., 1);
14         vec4 darkOrange = vec4(133. / 255. ,39. / 255., 3.
               / 255., 1);
15         vec4 background = vec4(0, 0, 0, 1);
16
```

```
17          float c = perlinNoise3D(vertexPosition.x,
                vertexPosition.y, vertexPosition.z);
18
19          float d = perlinNoise3D(vertexPosition.y,
                vertexPosition.x, vertexPosition.z);
20
21          temperature = clamp(temperature + c * .35, 0., 1.)
                ;
22
23          if(temperature < .25)
24                  color = background;
25          else if(temperature < .45)
26                  color = interpolate(background, darkOrange
                        , clamp(5. * (temperature - .25) + .1 *
                         d, 0., 1.));
27          else if(temperature < .65)
28                  color = interpolate(darkOrange,
                        middleOrange, clamp(5. * (temperature -
                         .45) + .1 * d, 0., 1.));
29          else if(temperature < .9)
30                  color = interpolate(middleOrange,
                        lightOrange, clamp(4. * (temperature -
                        .65) + .1 * d, 0., 1.));
31          else
32                  color = lightOrange;
33
34          vec4 ambientColor = color * (gl_LightSource[0].
                ambient + gl_LightModel.ambient);
35          vec4 diffuseColor = color * gl_LightSource[0].
                diffuse;
36          float diffuseValue = max(dot(vertexNormal,
                lightPosition), 0.0);
37
38          gl_FragColor = ambientColor + diffuseColor *
                diffuseValue;
39  }
```

The direction vectors used for fire are similar to the vectors used for basic animated noise or clouds, but a fire vector's highest magnitude component is the y component to make sure that the flames rise. Temperature is defined to be the distance from y = 10 and is then offset by a noise function so that there isn't a constant gradient of color from the bottom to the top. The x and y components are then swapped to generate another noise value that is used for the noise blending as shown in Figure

5.4(b). Temperature and blending values are all clamped to the range [0, 1] to make sure that the color values generated have RGB values in the same range. Figure 5.11 shows the result with the same seed values used for the clouds, but with octaves set to seven, zoom set to three, and persistence set to .55.

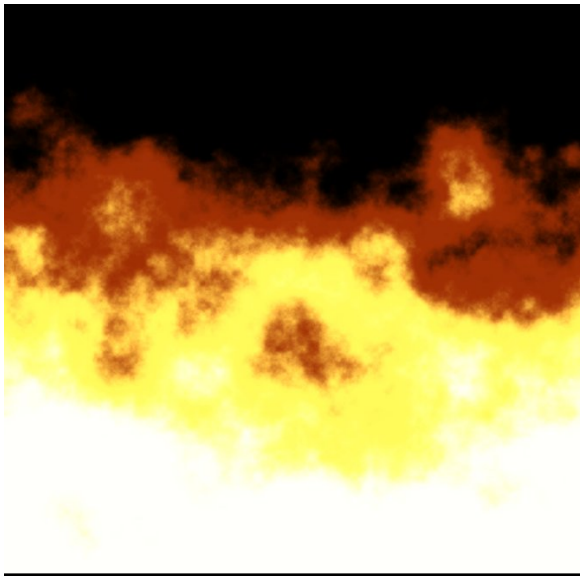


Figure 5.11: Animated Fire

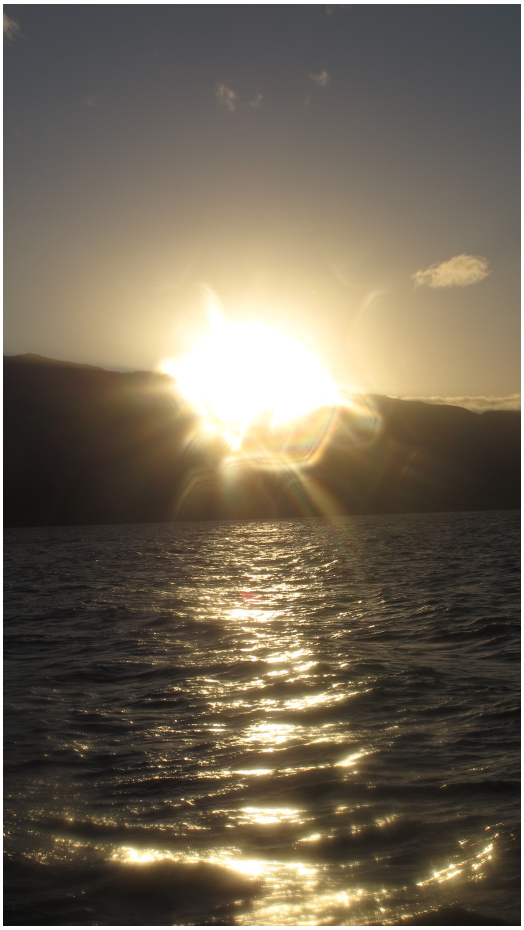
Compared to clouds and fire, it is much more difficult to model water with noise. Large bodies of water such as lakes and oceans have waves due to the moon's orbit around the earth, the waves have foam bubbles, and the color can be clear or opaque depending on the water depth. Examples are shown in Figure 5.12.

The larger waves are fairly regular, similar to a slightly offset sine or cosine function. The smaller ripples look noisier than the large waves. The foam appears mostly at the shoreline, but is also apparent where the waves crash. The water color blends from a blue to a blue-green tint before becoming clear at the shoreline.

Listing 5.8: GLSL - Animated Water

```
1 //Direction Vectors
2 vec3 dir1 = time * vec3(0, 1, 0);
3 vec3 dir2 = time * vec3(.1, .7, .1);
4 vec3 dir3 = time * vec3(-.3, .5, .1);
5 vec3 dir4 = time * vec3(1.1, .1, -1.1);
```

(a)



(b)

Figure 5.12: Pictures of the Pacific Ocean

```
6
7  void main()
8  {
9          float shoreDistance = (distance(vertexPosition.y,
               10.) + .5) / 20.;
10
11         vec4 darkBlue = vec4(0, 0, .33,1.);
12         vec4 lightBlue = vec4(0, .5, .25, 1.);
13         vec4 seaFoam = vec4(.5, .9, .9, 1.);
14         vec4 sand = vec4(235. / 255., 212. / 255., 169. /
               255.,1.);
15         vec4 color;
16
17         float c = perlinNoise3D(vertexPosition.x,
               vertexPosition.y, vertexPosition.z);
18         c = sin(vertexPosition.y + time +  abs( c ));
19         c = abs( c );
20
21         if(shoreDistance < .5)
22         {
23                 if(c < .05)
24                         color = seaFoam;
25                 else if(c < .07)
26                         color = interpolate(seaFoam,
                           darkBlue, 50. * (c - .05));
27                 else
28                         color = darkBlue;
29         }
30
31         else if(shoreDistance < .75)
32         {
33                 if(c < .05)
34                         color = seaFoam;
35                 else if(c < .07)
36                         color = interpolate(seaFoam,
                           interpolate(darkBlue, lightBlue
                           , 4. * (shoreDistance - .5))
                           ,50. * (c - .05));
37                 else
38                         color = interpolate(darkBlue,
                           lightBlue, 4. * (shoreDistance
                           - .5));
39         }
40
41         else
42         {
43                 if(c < .05)
```

```
44                          color = interpolate(seaFoam, sand,
                                4. * (shoreDistance - .75));
45                  else if(c < .07)
46                          color = interpolate(seaFoam,
                               interpolate(lightBlue, sand, 4.
                                * (shoreDistance - .75)),50. *
                                (c - .05));
47                  else
48                          color = interpolate(lightBlue,
                               sand, 4. * (shoreDistance -
                               .75));
49          }
50
51          vec4 ambientColor = color * (gl_LightSource[0].
               ambient + gl_LightModel.ambient);
52          vec4 diffuseColor = color * gl_LightSource[0].
               diffuse;
53          float diffuseValue = max(dot(vertexNormal,
               lightPosition), 0.0);
54
55          gl_FragColor = ambientColor + diffuseColor *
               diffuseValue;
56  }
```

The direction vectors used by perlinNoise3D are adjusted to capture the general direction that the waves travel. A gradient is used to determine deepness of the water so that the different blues can be blended as they approach the shoreline. A sine function is used to produce the periodic waves moving towards the shore and offset by noise similar to the marble feature. The outermost if statements determine the depth of the water so that the correct color is used depending on distance to the shoreline. The nested if statements are based on the c value to determine where the waves are located. Figure 5.13 shows the result using the above code with octaves set to four, zoom set to three, and persistence set to .3.

5.7 Conclusion

This chapter examined the various applications of noise in a graphical environment. It can be used to create static textures such as grass, sand, marble, and wood. Blending

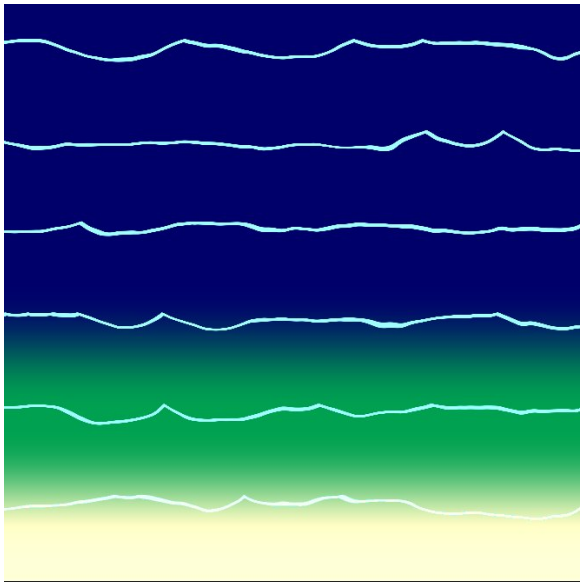


Figure 5.13: Animated Water

between textures using noise values looks more realistic than a linear blend. Virtual terrains can be created by generating height maps. Natural phenomena such as clouds, water, and fire can be emulated using noise. This leads into Chapter 6, which pulls everything together to create a procedurally generated world.

CHAPTER 6

Procedurally Generating Everything

6.1 Overview

This chapter uses the implementations in Chapters 4 and 5 to procedurally generate a virtual world. The discussion focuses on the development of the application as various features are added. Code for each step is separated by folders on the CD.

6.2 Creating A Virtual World

6.2.1 Step 1

The first step in developing a graphical application is to specify a window area where the graphics are rendered. While it is a trivial step, it is still important because without the window nothing else can be done. Figure 6.1 shows the window that is used to display the virtual world.

6.2.2 Step 2

The next step is rendering a simple terrain as shown in Figure 6.2.

The Noise class from Chapter 4 is used to create the height map for the landscape. Normal averaging is used to calculate the normal vectors at each vertex to minimize

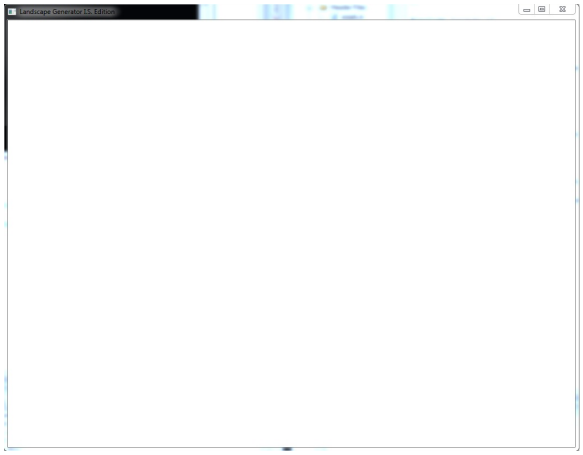


Figure 6.1: A Blank Window

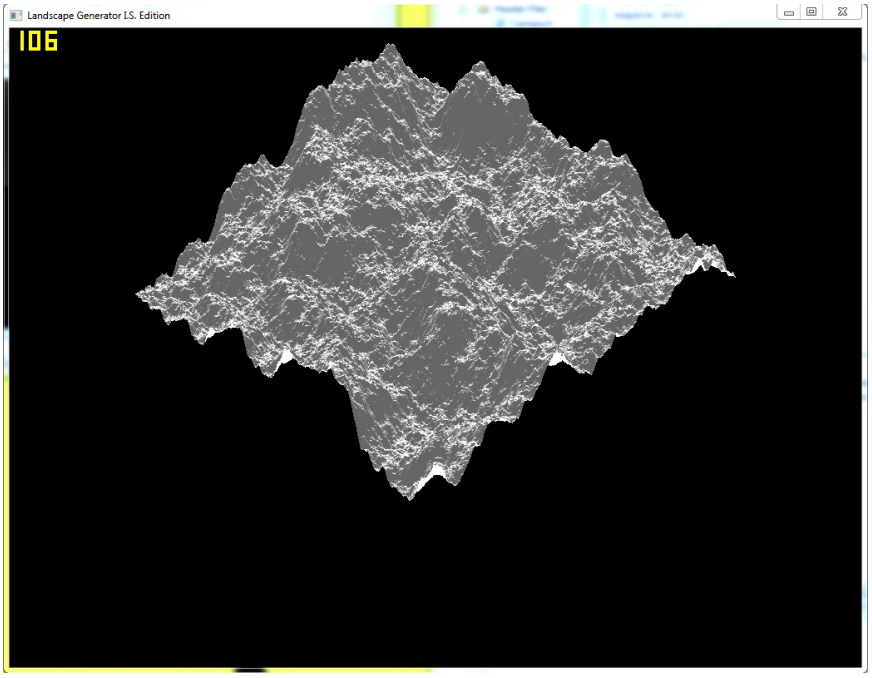


Figure 6.2: A Simple Terrain

the visibility of hard edges. A shader program is used to calculate per-fragment lighting using the Phong Illumination Model.

6.2.3 Step 3

The results of step 2 produce a landscape that looks like a crinkled piece of paper. Figure 6.3 shows the landscape with colors based on the height of the landscape.

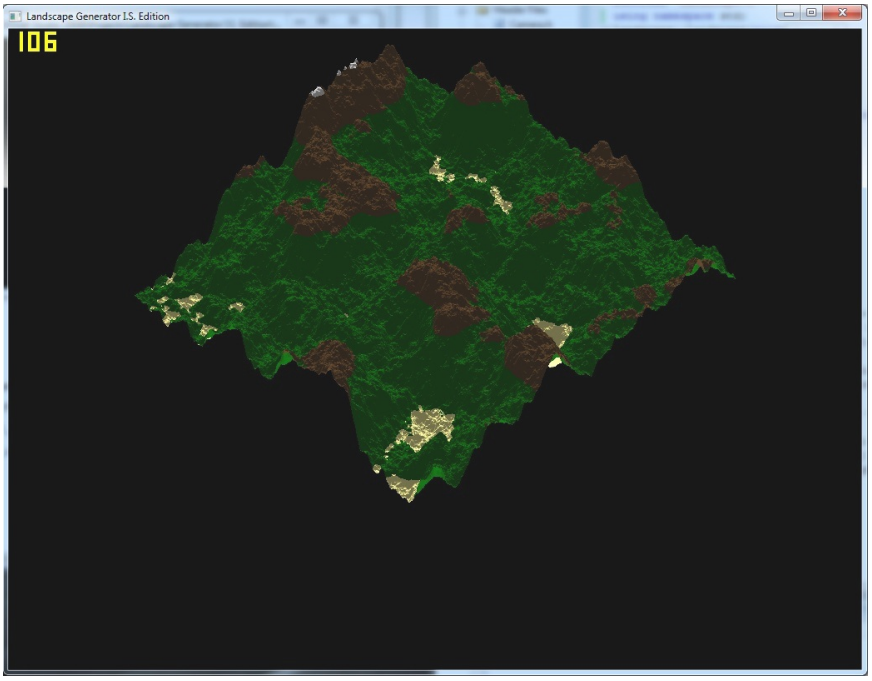


Figure 6.3: Adding Some Color

With the addition of landscape coloring, water also needs to be added as shown in Figure 6.4.

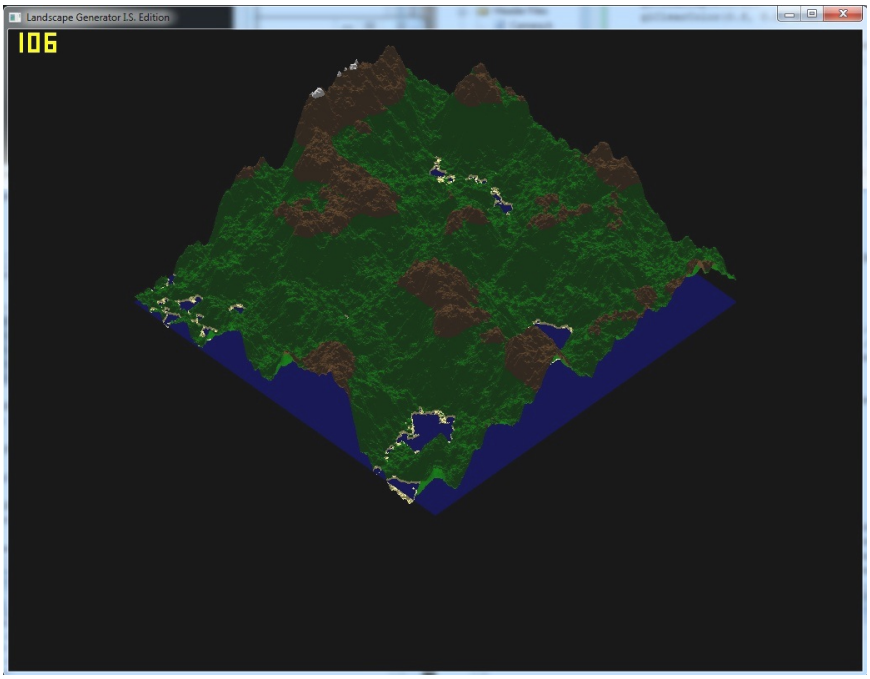


Figure 6.4: Adding A Water Level

The water is made up of two triangles forming a square. This is computationally

efficient, but looks unrealistic because the water can be seen both above and below the landscape.

Using an overhead view is one way to view the landscape, but it can also be viewed in first person as shown in Figure 6.5.

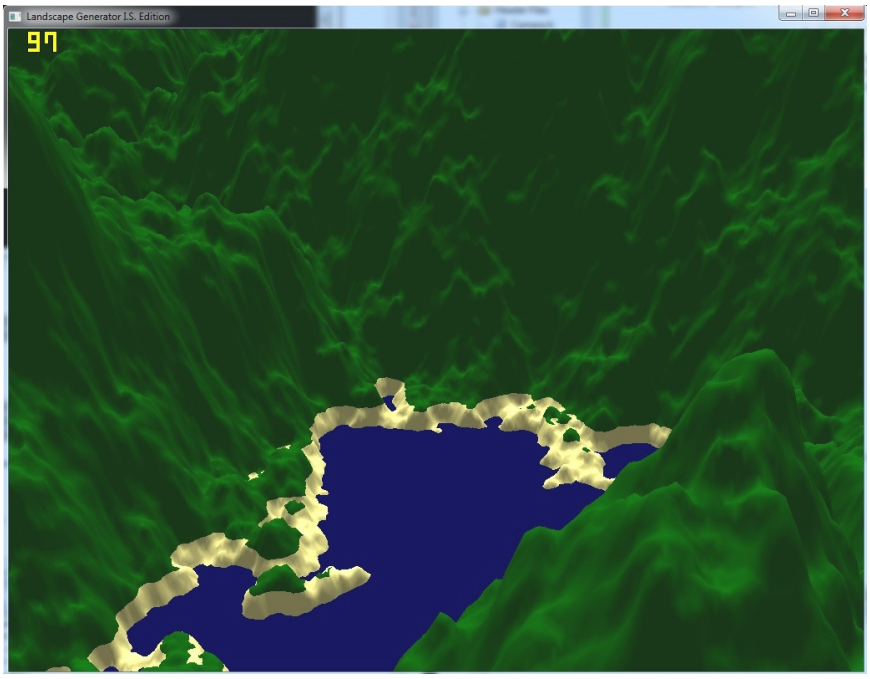


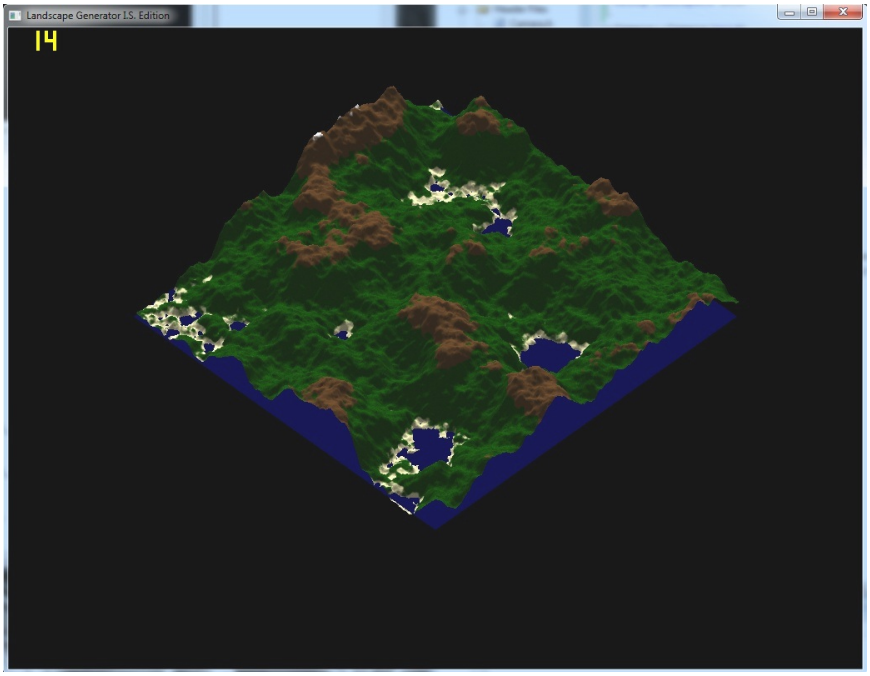
Figure 6.5: First Person Viewing

6.2.4 Step 4

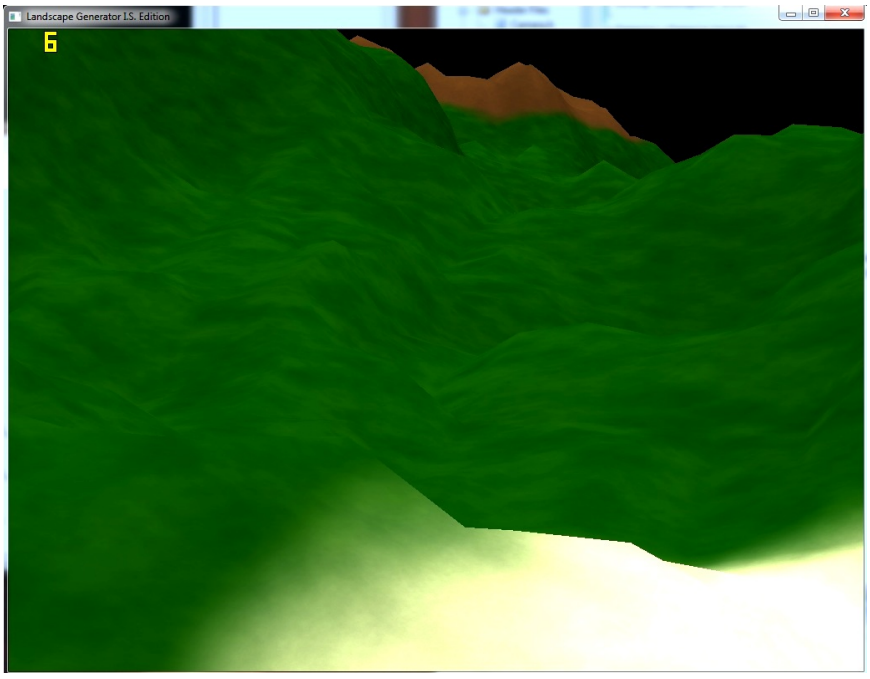
Examining the landscape in first person shows two problems; there is a hard line between different colors and that line is at a specific height. Figure 6.6 shows both the orthogonal and first person views of the landscape after fixing these issues.

This step varies the color using a method similar to the one discussed in subsection 5.4, but is based on three-dimensional noise. To get rid of the hard line between sections of color, linear blending based on the landscape height is used.

Without a sky for the background there is nothing but black beyond the edge of the landscape. Figure 6.7 shows the addition of a sky with clouds.



(a)



(b)

Figure 6.6: Textured Colors and Linear Interpolation

Figure 6.7: A Sky with Clouds

6.2.5 Step 5

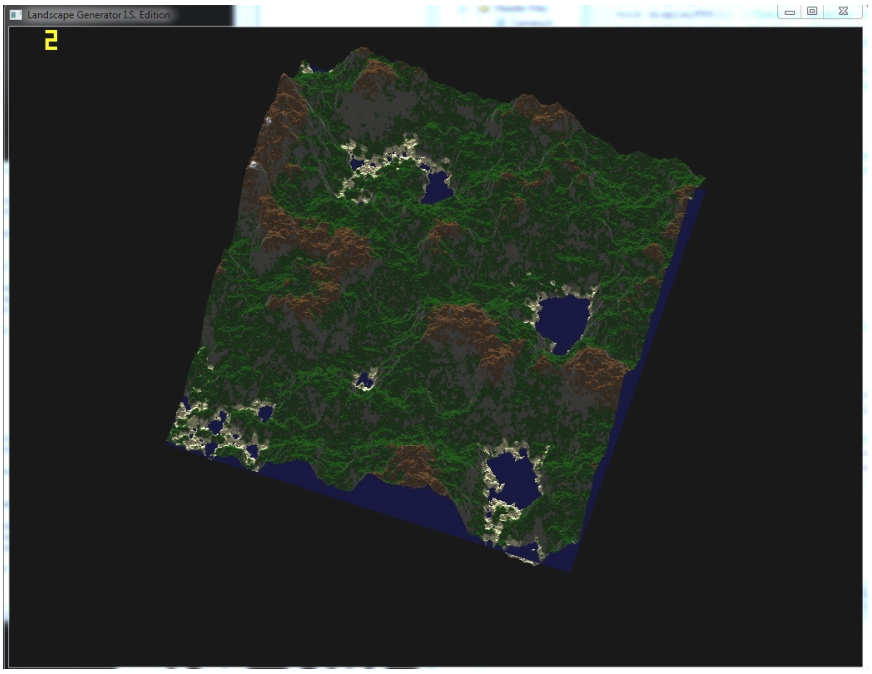
The algorithm used to color the landscape looks reasonable, but Figure 6.8 shows a more advanced algorithm that also takes the slope of the surface into consideration. The interpolation between height levels is also offset by noise to add to the realism.

Another feature that can be improved on is the appearance of the water. In its current state the water has a single color no matter where the observer or light source is located. Bump mapping is used to simulate ripples in the water as shown in Figure 6.9.

6.2.6 Step 6

As stated in subsection 5.5, using multiple noise functions allows for more variety in the geographical features of a landscape. Figure 6.10 shows a landscape generated using four noise functions.

The first function uses a low persistence at a high zoom level to define the main shape of the landscape. Two more noise functions at higher persistence levels and



(a)

(b)

Figure 6.8: Slope Texturing and Noisy Interpolation

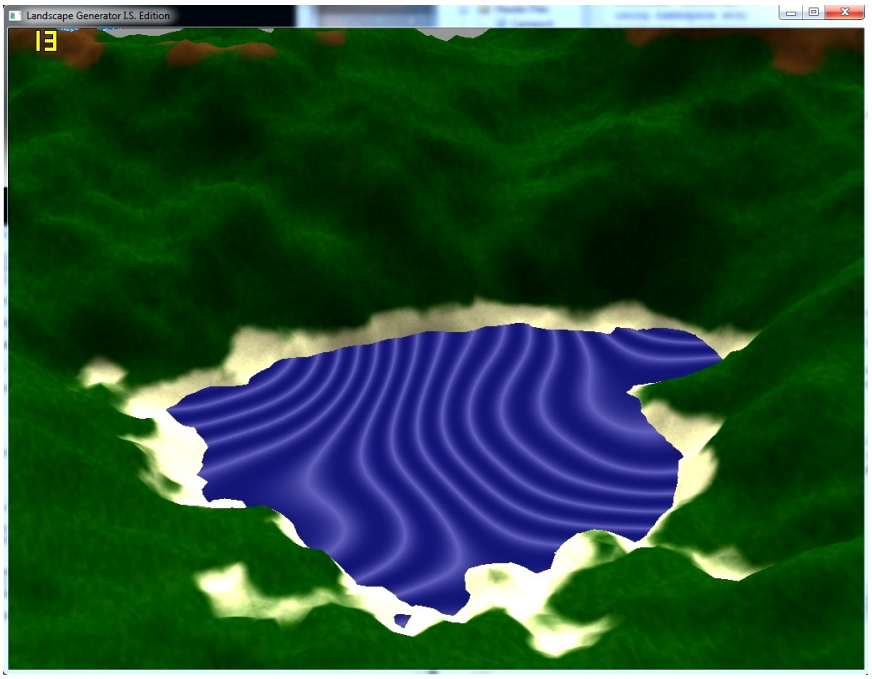


Figure 6.9: Bump Mapping Water

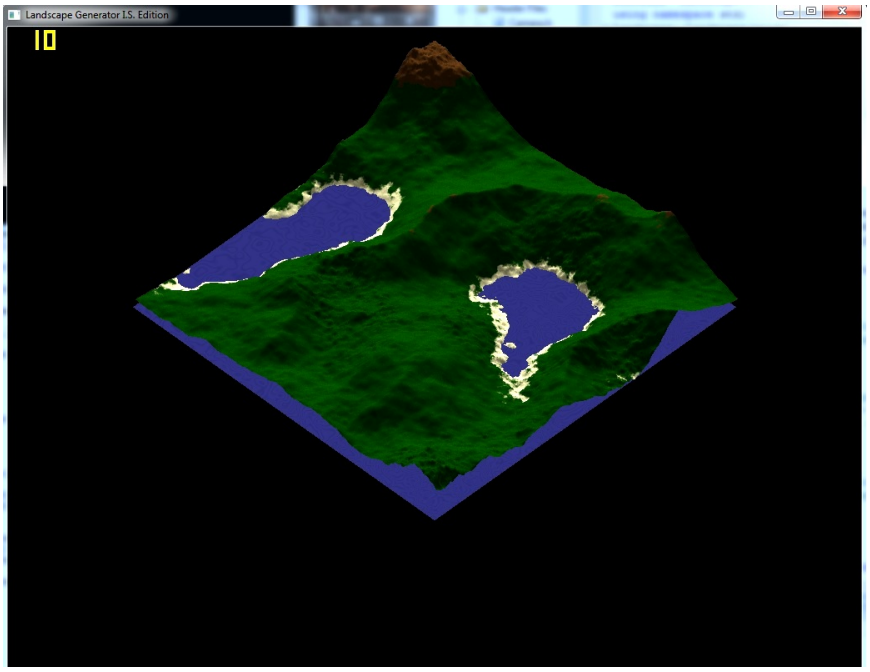


Figure 6.10: Multiple Noise Landscape

lower zoom levels are used to add bumpiness to the landscape. The last noise function is used to determine how much of the higher persistence functions are added to the first noise function so that the bumpiness is not uniform over the entire landscape.

The next feature added is a day/night cycle. This includes a moving light source,

changing the light properties based on time of day, and changing the colors used for the sky and clouds. Figure 6.11 shows the sky at night and dusk.

6.2.7 Step 7

Step 6 allows for a day and night cycle, but a change in time requires a light source. Figure 6.12 shows the sun behind a large cloud.

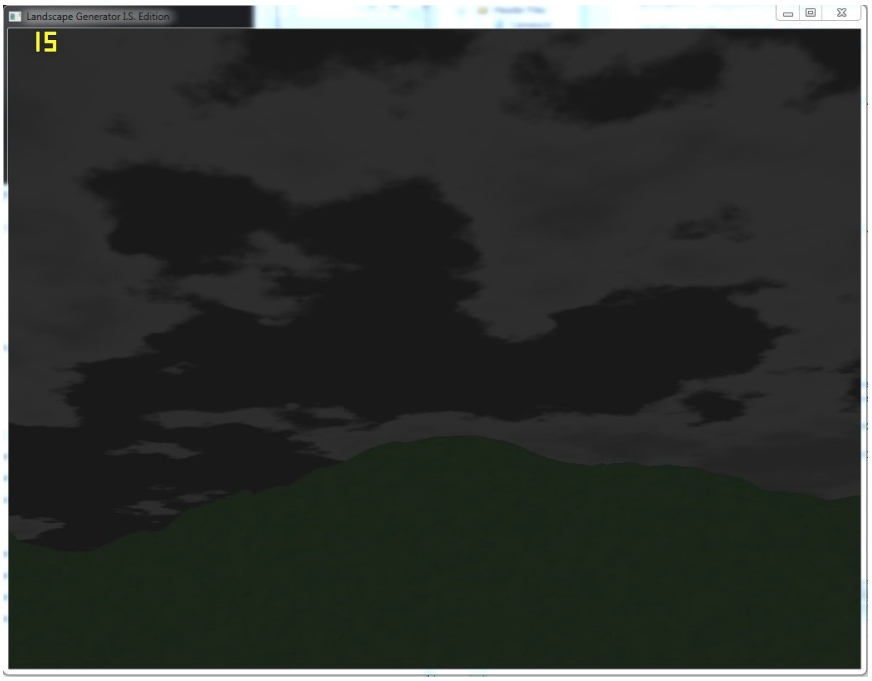
Another detail that has been overlooked so far is the shape of the landscape. Multiple noise functions give the landscape a more varied look, but it is still only a square section of land.

Figure 6.13 shows a sphere with its vertices deformed by three-dimensional noise. The deformed sphere is a much better representation of the moon than of a landscape, but that is also due to the current shader program selecting colors based on the y-values of the vertices. For the world to have the same color scheme as the flat landscape it must be based on distance from the center of the world as shown in Figure 6.14.

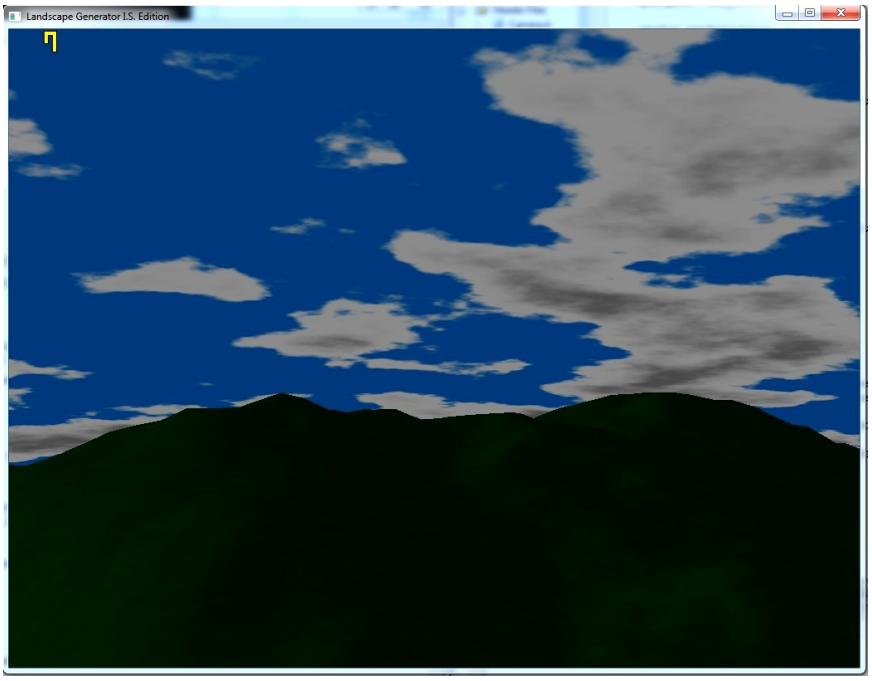
6.2.8 Step 8

Every landscape generated so far is stored in a single vertex buffer allowing for the landscape to be drawn with a single set of OpenGL commands. The landscapes generated so far have been relatively small and can be easily displayed in real time. Larger landscapes, however, are displayed at less than optimal speeds. To address this problem, the landscape must be separated into sections. Figure 6.15 shows a "flat" landscape that is made of 2x2 sections.

A common problem with connecting multiple sections of a landscape together is that the normal vectors at the edges of a section don't match up with those of an adjacent section. This is fixed by increasing the dimensions of the height map by two in each direction, but using only the points not on the edge of the section to create



(a) Night



(b) Dusk

Figure 6.11: Addition of Day/Night Cycle

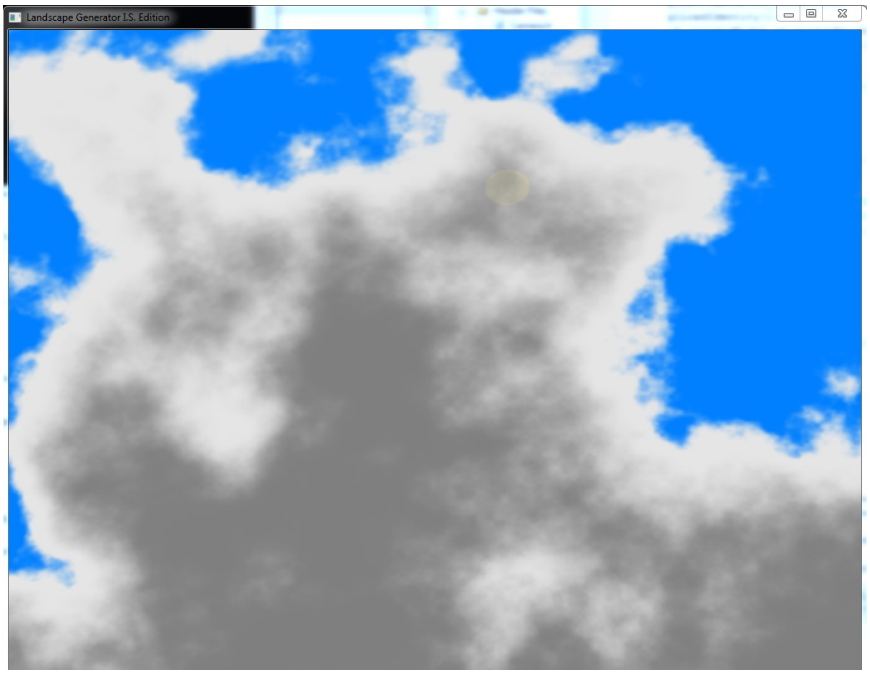


Figure 6.12: Sun Behind The Clouds

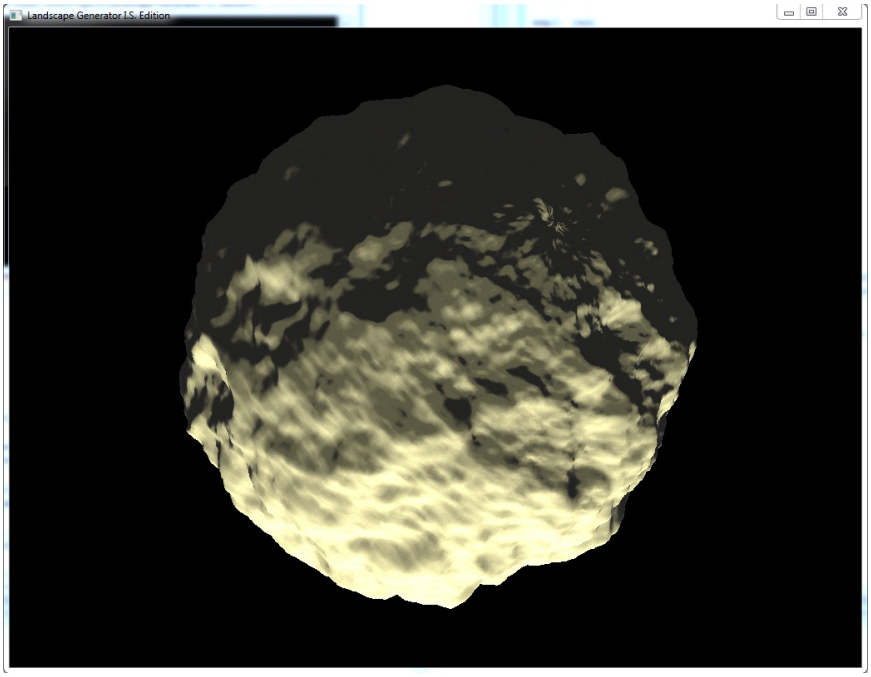


Figure 6.13: The Moon

the visible height map. This ensures that when the normal averaging occurs every point that is displayed has its normal vector averaged from six triangles.

The multiple section concept can be extended to the world as well. Because a sphere is generated using the same code shown in subsection **??**, triangles are much

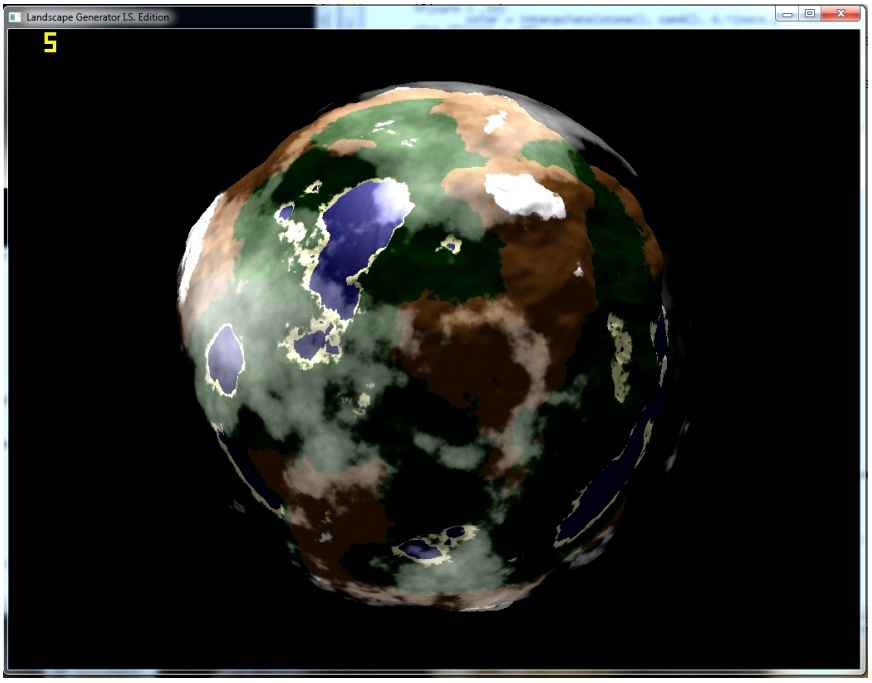


Figure 6.14: The First World

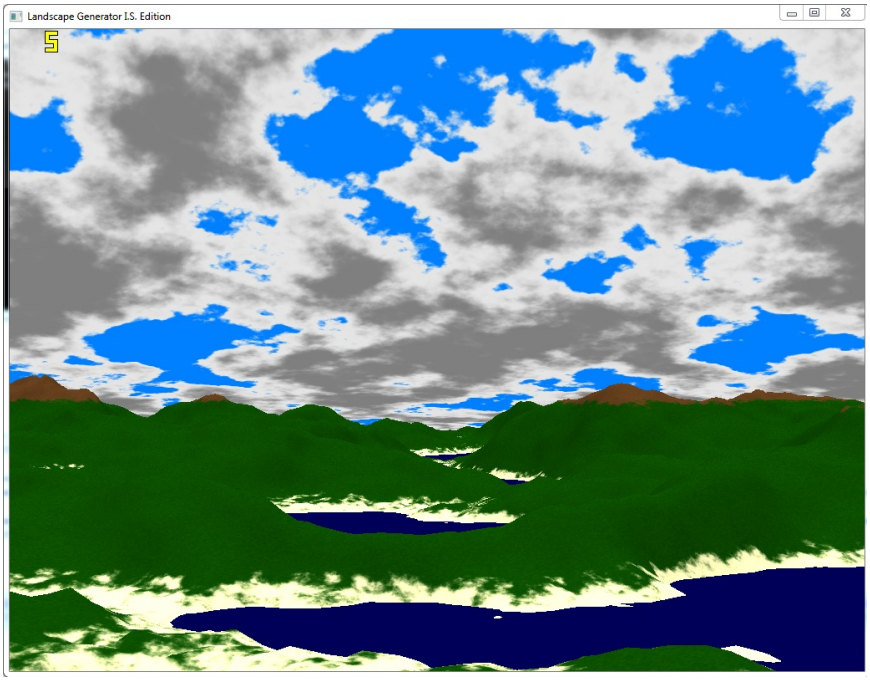


Figure 6.15: Flat Sectioned Landscape

smaller near the poles than they are near the equator as shown in Figure 6.16. This is partially remedied by doubling the number of sections that go horizontally around the sphere compared to the number of sections that go from pole to pole. The idea for this remedy is based on latitude and longitude lines. Latitudes range from -90° to

90° and longitudes range from -180° to 180°. Figure 6.17 shows a world made with 2x4 sections.

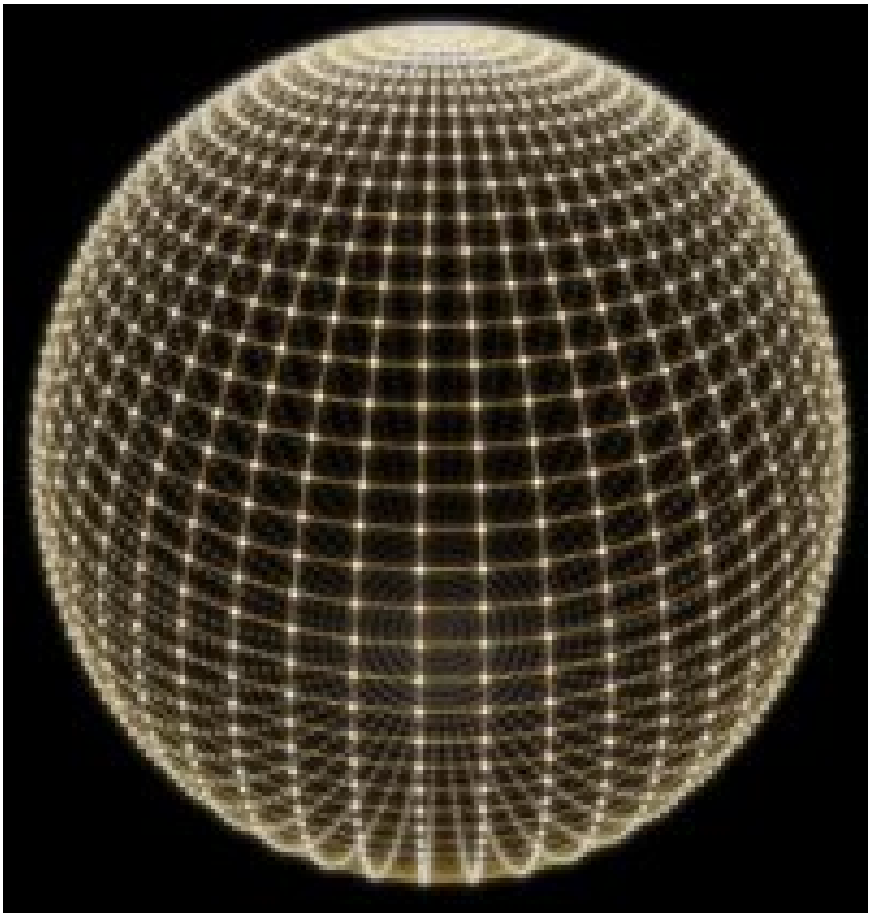


Figure 6.16: UV Mapped Sphere [20]

6.2.9 Final Product

Adding features incrementally does not mean that the application look as though it was built in an ad hoc manner, but this is not the case. Figure 6.18 shows a UML Class diagram of the Landscape Generator program.

Beginning with the heart of the diagram, the VertD and VertF classes are containers for (x,y,z) coordinates consisting of doubles and floats respectively.

Below VertF in the diagram is the Noise class from Chapter 4. It generates noise values for the program based on the seed values, persistence and zoom factor.

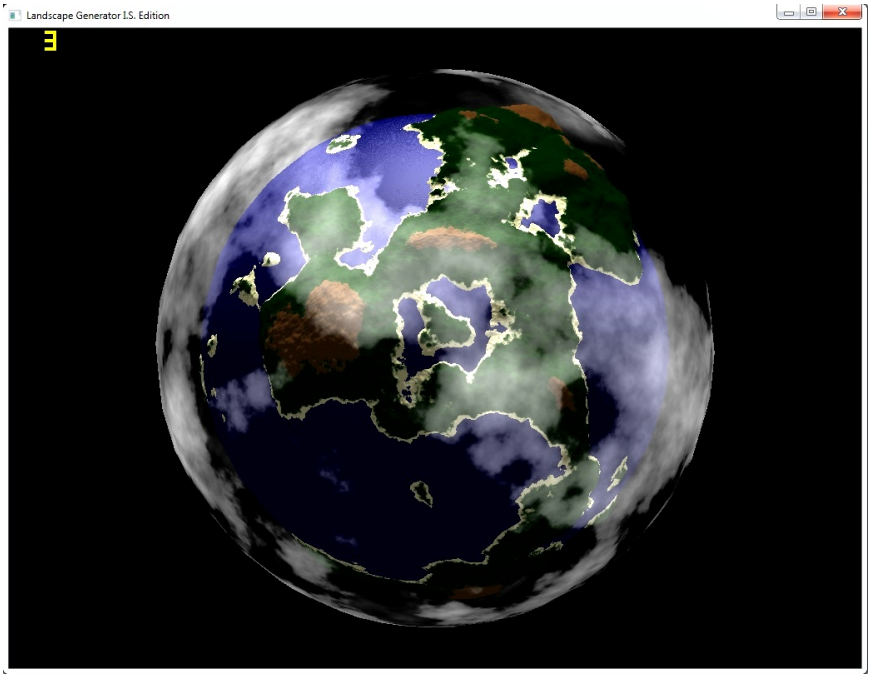


Figure 6.17: Spherical Sectioned Landscape

The Landscape and SphericalTerrain classes create the height map for a flat landscape and spherical landscape respectively. They are called to display the landscape during the program's main loop.

Connected to Landscape and SphericalTerrain are the SectionLandscape and SphericalSection classes. Functionally, these classes are similar in that they store arrays of Landscape's and SphericalTerrain's and calculate the dimensions for each individual section.

The Camera class also uses the VertD class. It is responsible for storing the camera's location and the point where the camera is aimed.

The myGL class is used by the Landscape and SphericalTerrain classes previously mentioned as well as the Skybox and Shader classes. It provides OpenGL 2.1 functionality because Windows requires function pointers to all OpenGL commands that are not included in OpenGL 1.1.

The Shader class is used to load, compile and run shader programs. It activates a shader during the display process and then deactivates it when it is not in use.

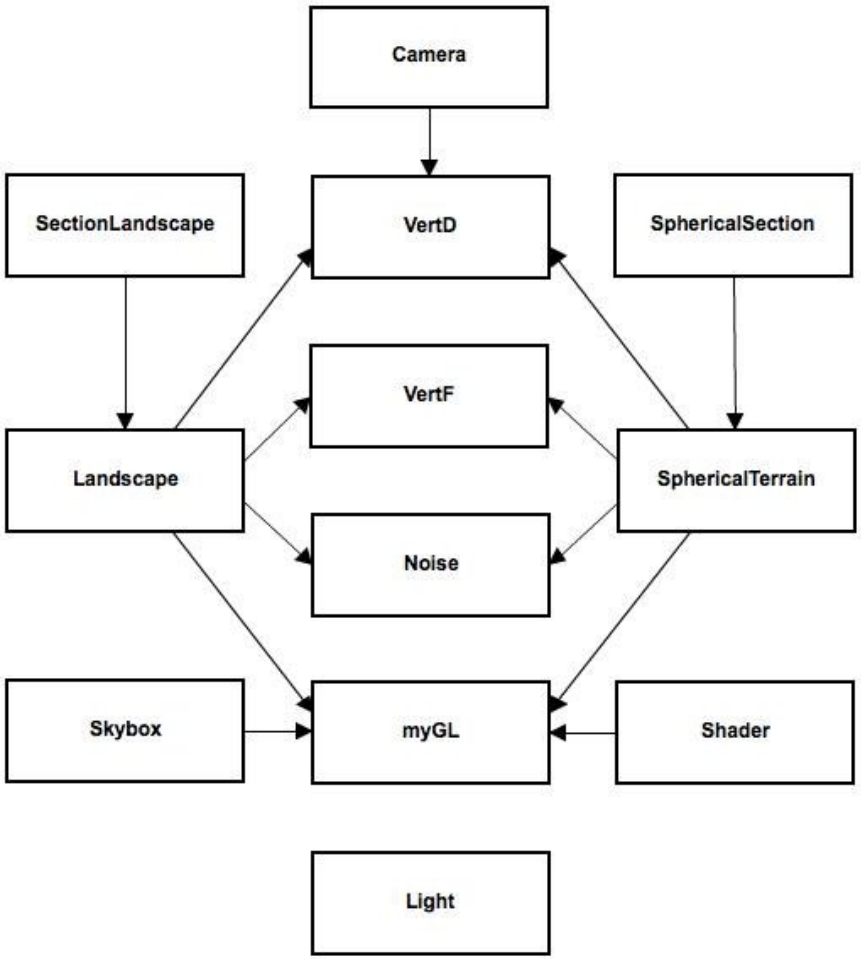


Figure 6.18: Landscape Generator UML Class Diagram

The Skybox class is used to display a sphere of a specific radius. This is used to display the clouds in perspective viewing and to display an orthographic viewing of the world as well as the water surface of the world.

The Light class is not connected to any other class. It is a container for the ambient, diffuse, and specular colors of a light source as well as its position.

In addition to the classes in Figure 6.18, the application uses six shader programs, three each for the flat landscape and spherical landscape. They are the terrain shader, water shader, and cloud shader.

The terrain shader is responsible for coloring the landscape based on height. It calculates two noise values, one is used for blending between two colors for a ground

material such as a lighter and darker green for the grass, and the other is to offset blending between layers such as sand and grass.

The water shader is responsible for bump mapping the water layer. It calculates two noise values that are used to offset the z coordinate of the normal vector to create the ripple effect.

The cloud shader is slightly different for the two types of landscape. In the flat landscape it is responsible for creating the clouds based on a noise value that blends between the sky and clouds. It also changes the colors of the sky and clouds based on the time of day and makes a spot for the sun that is blended with the sky. For the spherical landscape the shader is responsible for creating the clouds based on a noise value and determining an alpha value so that the cloud layer can be blended with the water and landscape layers for a transparent effect.

6.2.10 Conclusion

This chapter examines the process of developing a program that procedurally generates a landscape. It starts by creating a first window and ends with a landscape that has bump mapped water and clouds generated in real-time. The chapter also outlines the organization of the program into separate classes and shader programs.

CHAPTER 7

Conclusion

7.1 Future Work

As it goes with any research project, more could have been done with additional time. Below are some ideas that could be explored and other improvements that could be made to the applications presented in this I.S.

7.1.1 Applications

There are seven texturing techniques discussed in Chapter 5. The sand and grass texture is created by blending two different colors using noise. Blending between three colors could possibly produce a more realistic effect. The marble texture exhibited banding, but the bands are straight. To be more realistic, the bands could be skewed with another noise function. The wood texture uses only specific colors of brown, but there are many different kinds of wood that can be simulated with a simple variance of colors. The cloud and fire effects could be more realistic if noise was applied to a particle system. The water animation simulates waves coming to shore, but the waves themselves are never broken or staggered.

7.1.2 Landscape Generator

The landscape generator application could be enhanced in the following ways.

The rendering speed for the landscape could be faster if sections of the scene not visible are culled prior to moving down the pipeline. Figure 7.1 shows an example of the current implementation on top and the improved culling algorithm on the bottom. Changing the level of detail of a visible section based on its distance from the camera would allow for a larger view distance to be used because fewer vertices would need to travel through the graphics pipeline. An example of this is shown in Figure 7.2. The checkerboard squares closer to the camera take up much more area than the squares farther away from the camera.

There are also a few ways that the landscape could have been made more realistic on a small scale. The water should be transparent based on its height above the landscape. Fogging out sections based on their distance away from the camera would hide the appearance of sections as they come within the view radius. The landscape is a height map with a water surface. Adding blades of grass, trees, or any type of vegetation would have improved the realism.

On a larger scale, the cloud layer around the spherical world looks out of place without the blue glow around the world similar to the one shown in Figure 7.3. The clouds do not lighten or darken based on the light position. The coloring of the landscape left a little bit to be desired as well. Having the entire planet covered in grass and water looks interesting, but adding gradients for temperature and humidity levels would make it much more realistic.

7.1.3 OpenGL 3

In Chapter 2, OpenGL's history and functionality are scrutinized because the API is used to display the effects of this I.S. However, all of the effects shown can be implemented using OpenGL 2.1. OpenGL 3 included a new shader type, the geometry

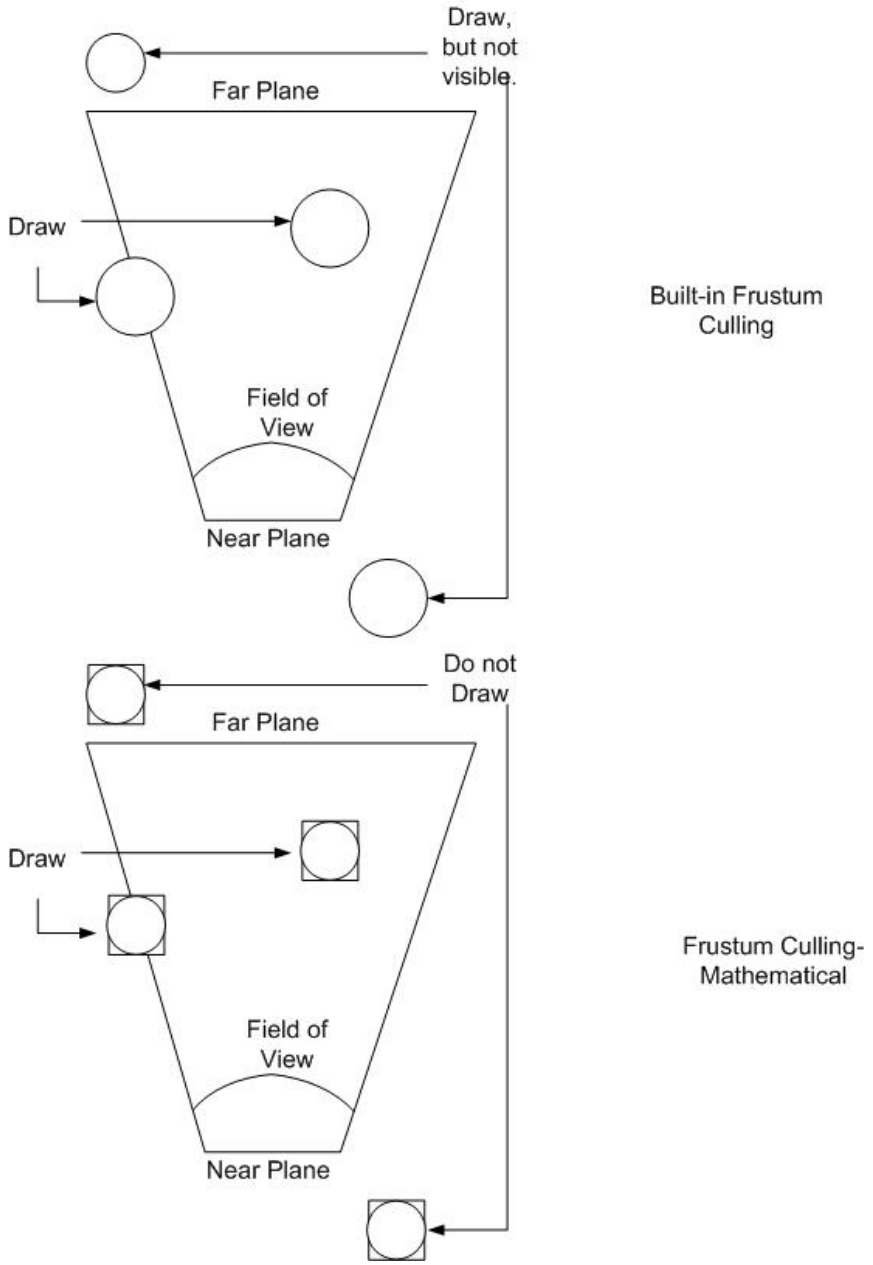


Figure 7.1: View-Based Culling [21]

shader, which opened up a number of other possibilities. For example, a set of points passed to a geometry shader can be used to calculate the triangles and normal vectors. This can be performed on the graphics card in real-time rather than on the CPU as part of the initialization phase of the application. Another possibility would be to deform the water layer using noise. The bump mapping looks semi-realistic until

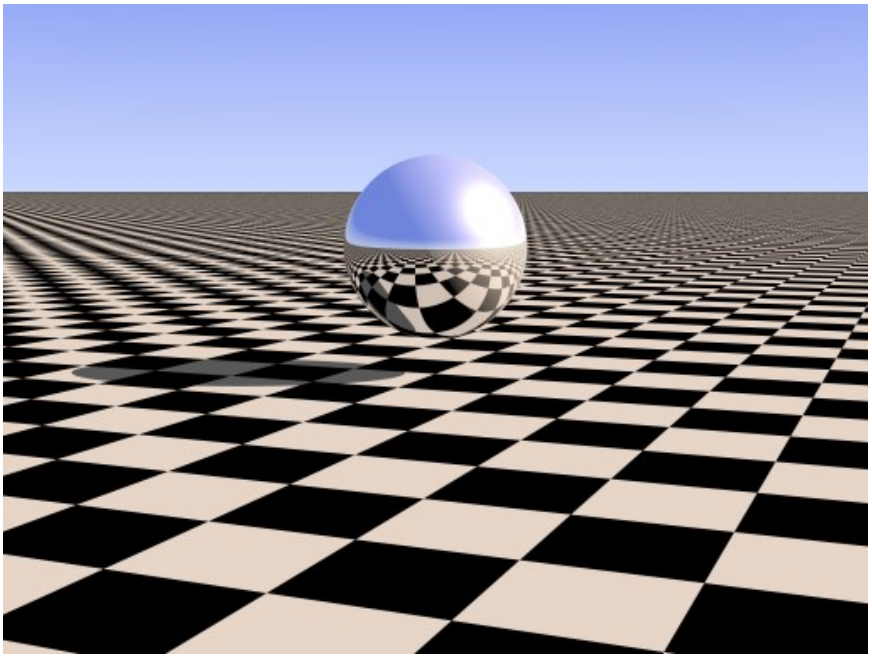


Figure 7.2: Less Detail at Distance [2]

Figure 7.3: The Earth [16]

it meets the edge of the landscape where it's extremely noticeable that the water's surface is still completely flat.

7.2 Summary and Conclusion

Chapters 5 and 6 showcased a variety of effects including textures for marble, wood, and animated fire and the procedural landscape generator. The development of these products required research and preliminary programming to be done first. Understanding a graphics API, graphics card programming, and Perlin Noise were essential components of this research.

One of the necessary requirements for this I.S. is the ability to display images onscreen. OpenGL, the most widely used computer graphics API, was used for this aspect. The API handles the entire graphics pipeline from defining vertices and normal vectors to displaying the data onscreen. It also allows the use of shader programs, making the Phong Illumination Model computable in real-time.

Just as important as being able to display the final products are the calculations done to create them, and without Perlin Noise this wouldn't have been possible. Noise functions are similar to sine and cosine waves in that they have definitions of amplitude, wavelength, and frequency. Noise functions also have properties of their own including persistence, zoom level, and the number of octaves.

In order for noise functions to be used, an entire class was implemented for noise generation and discussed in Chapter 4. This implementation generates noise in C++. Slight changes in the code written for the class allowed for all the necessary functions to be translated into GLSL code for the OpenGL shader programs.

Creating the implementations of noise was the turning point required to start work on the final products. Grass and sand were both created using the C++ implementation and were also used to demonstrate noise-blending for a more realistic texture blending

technique. Wood and marble were created using the C++ implementation for a flat surface and the GLSL implementation for the Utah teapot. Clouds, fire, and water were all created and animated using the GLSL implementation on a flat surface.

Chapter 6 covers the process of creating a procedurally generated landscape using noise. It utilizes clouds, grass, sand, noise-blending, and multiple noise functions to create the height map demonstrated in Chapter 5, but it goes even further. Textures for stone, snow, and dirt are generated. Water is bump mapped to create the illusion of ripples along the surface. A sphere is deformed using noise to create a world.

The application of procedural generation with noise does not stop with the textures and landscapes made using the final products of this I.S. Wood grain can be simulated with high frequency noise that is stretched in one direction. The rough surface of brick and stone can be created by using high persistence noise at a low zoom level. The possibilities are endless.

References

1. James F. Blinn and Martin E. Newell. Texture and reflection in computer generated images. *Communications of the ACM*, 19(10), October 1976. 2

2. Julian M Bucknall. Ray tracing image from june 2010's pcplus, June 2010. URL http://www.imagico.de/pov/pict/sphere2.jpg. x, 99

3. Wayne E. Carlson. A historical timeline of computer graphics and animation, September 2004. URL https://design.osu.edu/carlson/history/timeline.html.

4. Leonardo da Vinci. Mona lisa, June 2006. URL http://www.ibiblio.org/wm/paint/auth/vinci/joconde/. vii, 3

5. Hugo Elias. Perlin noise, December 1998. URL http://freespace.virgin.net/hugo.elias/models/m_perlin.htm. 27

6. James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics Principles and Practice*. Addison Wesley Publishing Company, second edition in c edition, 1996. 1

7. Ryan Geiss. Gpu gems 3: Chapter 1. generating complex procedural terrains using the gpu, July 2009. URL http://developer.nvidia.com/node/158. viii, 29

8. H. Gouraud. Continuous shading of curved surfaces. *IEEE Transactions on Computers*, 20(6), 1971. 8

9. Paul S. Heckbert. Survey of texture mapping. *IEEE Computer Graphics and Applications*, November 1986. 2

10. Khronos Group Inc. Opengl - the industry's foundation for high performance graphics, 2011. 6

11. Khronos Group Inc. Opengl es 2x - the standard for embedded accelerated 3d graphics, February 2012. URL http://www.khronos.org/opengles/2_X/. vii, 7, 13, 17

12. Khronos Group Inc. The history of opengl, 2012. URL http://www.khronos.org/kite/ocw/mod/page/view.php?id=2. 6

13. Peter Kutz. Computer graphics by peter kutz, 2011. URL http://peterkutz.com/computergraphics/. viii, 30

14. Barthold Lichtenbelt and Pat Brown. Ext gpu shader4. Technical report, NVIDIA, 2008. 19

15. Maxim. Pseudo random number generation using linear feedback shift registers, June 2010. URL http://www.maxim-ic.com/app-notes/index.mvp/id/4400. 40

16. NASA. Nasa visible earth: The blue marble, February 2002. URL http://visibleearth.nasa.gov/view.php?id=57723. x, 99

17. Ken Perlin. An image synthesizer. *Computer Graphics*, 19(3), 1988. 33

18. Ken Perlin. Improving noise. *Computer Graphics*, 35(3), 2002. 33

19. Bui Tuong Phong. Illumination for computer generated pictures. *Communications of the ACM*, 18(6), 1975. 2

20. Iñigo Quilez. Patched sphere, 2008. URL http://www.iquilezles.org/www/articles/patchedsphere/patchedsphere.htm. x, 92

21. roecode. Xna framework gameengine development. (part 13, occlusion culling and frustum culling), February 2008. URL http://roecode.wordpress.com/2008/02/18/xna-framework-gameengine-development-part-13-occlusion-culling-and-frustum-culli x, 98

22. Mark Segal and Kurt Akeley. The opengl graphics system: A specification (version 1.5). Technical report, Silicon Graphics, Inc., 2003. 12

23. Mark Segal and Kurt Akeley. The opengl graphics system: A specification (version 2.1 - july 30,2006). Technical report, Silicon Graphics, Inc., 2006. 15

24. Mark Segal and Kurt Akeley. The opengl graphics system: A specification (version 3.3 (core profile) - march 11, 2010). Technical report, Kronos Group Inc., 2010. 19

25. Larry Stritch. Celebrating wildflowers - plant of the week - plants of the winter solstice:, 2011. URL http://www.fs.fed.us/wildflowers/plant-of-the-week/winter_solstice.shtml. ix, 73

26. Ulysses. Analog tv is no more, June 2009. URL http://ulyssesonline.com/2009/06/12/analog-tv-is-no-more/. vii, 20

27. Eric W. Weisstein. Noise, 2012. URL http://mathworld.wolfram.com/Noise.html. 20

28. Mason Woo, Jackie Neider, and Tom Davis. *OpenGL Programming Guide*. Addison-Wesley Publishing Company, 1997. vii, 9