2001

Building an Interactive, Three-dimensional Virtual World

Raymond Mazza
Colby College

BUILDING AN INTERACTIVE, THREE-DIMENSIONAL VIRTUAL WORLD

by

RAYMOND MAZZA

Submitted in Partial Fulfillment of the Requirements of the
Senior Scholar Program

COLBY COLLEGE
2001

Acknowledgements

First, I would like to thank my advisor, Randy Jones, for the motivation he has provided me with throughout the year. Without Randy helping me set weekly goals for myself, I don't think I would have been able to accomplish nearly as much. Randy has also helped me put this paper together, as well as documents for the CCSCNE conference we attended to present my work.

I would also like to thank Clare Congdon for her constructive criticisms and assessments of my writings, which have been very helpful. Clare also provided great guidance this year in general as my academic advisor.

Thanks to Randy, Clare and Dale Skrien for being readers of my paper, and mentors throughout my college career.

Lastly, thanks to everyone else who has been interested in my work and stopped by the lab to ask questions or give me compliments and further motivation.

ABSTRACT

The movement of graphics and audio programming towards three dimensions is to better simulate the way we experience our world. In this project I looked to use methods for coming closer to such simulation via realistic graphics and sound combined with a natural interface.

I did most of my work on a Dell OptiPlex with an 800 MHz Pentium III processor and an NVIDIA GeForce 256 AGP Plus graphics accelerator — high end products in the consumer market as of April 2000. For graphics, I used OpenGL [1], an open-source, multi-platform set of graphics libraries that is relatively easy to use. I coded in C++.

The basic engine I first put together was a system to place objects in a scene and to navigate around the scene in real time. Once I accomplished this, I was able to investigate specific techniques for making parts of a scene more appealing.

The use of texture mapping (fitting images to geometric surfaces) makes one of the most incredible differences because it provides a means to add much detail to objects in the scenes. Furthermore, it saves time and geometry over representing the contents of the image in three dimensions. Textures can also be mapped to objects dynamically to make surfaces appear to move — this is useful when designing, say, the surface of a body of water. I used a technique of combining textures in a paint program to make transitions of textures appear less harsh or abrupt.

In order to keep a decent frame rate (around 40 frames per second), geometry has to be limited. Since detailed objects are more pleasing, I decided to have more complicated objects and smaller scenes. By connecting many of these smaller scenes, the user experiences a large, detailed world. I have also found that objects composed of many pieces at different depths are very pleasing to move around. I use 3D Studio Max [2] to model and alter objects, and 3D Exploration [3] to convert 3D object files to OpenGL code in C++.

By adding three-dimensional sound, the user can pay attention to the world with two senses as opposed to one. I added sounds using OpenAL [4] with some difficulty

and continuing trouble. OpenAL has support for attenuation, pitch control, reverberation, and Doppler effect.

I added interaction through two different methods: collision detection, and picking (OpenGL). Collision detection uses geometric approximations of objects' forms to detect when a moving object hits another, and is important to keep from walking through walls. Picking renders into a buffer, rather than to the screen, and provides useful information about the buffer's contents.

Life-like camera motion allows the user to feel more natural while navigating the world. Physics equations of motion controls falling of objects, including the camera. Other aesthetics I have added are alterations of lighting – to become darker during the "night" and lighter during the "day." I am also using blending to make objects such as water transparent, and to make distant objects fade in and out of the scene with a "fog". And I have worked on real-time reflections on flat and curved surfaces.

All these effects have brought this project far as I have had excellent results. Most of the techniques I used were successful. The world turned out to have a total of eleven interesting scenes. Although, there are many more possibilities for further scenes and features to incorporate. Some include different forms of optimization, real-time shadows, animation of complex objects, stereovision viewing with a headset, and head tracking.

Contents

CHAPTER 1 **Introduction** 1

CHAPTER 2 **Background and Terminology** 4

CHAPTER 3 **Creation of High Polygon Count Objects** 7

3.1 Introduction 7
3.2 The Procedural Approach 7
3.3 The 3-D Object Modeler Approach 9

CHAPTER 4 **Keeping it in Real Time** 12

4.1 Introduction 12
4.2 Optimizing Storage: Display Lists and Texture Objects 12
4.3 Face Culling 13
4.4 Dividing Scenes 14

CHAPTER 5 **Scene Unification** 17

5.1 Introduction 17
5.2 Transports 18
5.3 Transitions Situated Around Corners 18
5.4 Screen Snapshots and Transfers 19

CHAPTER 6 **Adding the Element of Time** 23

CHAPTER 7 **Visual Realism - Textures, Blending, Lighting, Fog** 28

7.1 Introduction 28
7.2 Textures 28
7.3 Dynamic Textures 36
7.4 Blending, Lighting, and Multi-Texturing 37
7.5 Sphere Mapping 40
7.6 Reflections on Flat Surfaces 43

7.7 Textures with Alpha Channels 45
7.8 Special Texturing Techniques 48
7.9 Fog Effects 50
7.10 Per-object fog 53
7.11 Multiple Transparencies 54
7.12 Fog and the Stencil Buffer 56

CHAPTER 8 **Terrain 58**

8.1 Introduction 58
8.2 Close Terrain 58
8.3 Distant Terrain 60

CHAPTER 9 **Particle Engine 63**

CHAPTER 10 **Boundlessness: Living Up to the Word "World" 68**

10.1 Introduction 68
10.2 Arrangement 68
10.3 Working with Emptiness 69

CHAPTER 11 **Navigation 70**

CHAPTER 12 **Collision Detection 72**

12.1 Introduction 72
12.2 Ray-Casting 73
12.3 Collision Cylinders 75
12.4 Collision Optimization 77

CHAPTER 13 **Collision Reaction 80**

13.1 Introduction 80
13.2 Multiple Subsequent Collisions 82
13.3 Ground Versus Walls 82

CHAPTER 14 **Audile Realism 84**

14.1 Introduction 84
14.2 False Starts 84
14.3 Working Sound 85

CHAPTER 15 **Real-Time Engine Manipulation 88**

CHAPTER 16 **Bugs, Obstacles, and Pitfalls 91**

16.1 Installations and Incompatibilities 91
16.2 On The Mechanics of Fog and Lighting 93
16.3 Collision Setup 94
16.4 3rd Party Bugs 94

CHAPTER 17 **Conclusions 96**

17.1 Results 96
17.2 Future Work 96

BIBLIOGRAPHY **Resources 102**

CHAPTER 1: Introduction

For the past academic year (2001) I have used recent technology and computer tools to design a virtual world that can be navigated fluidly in "real time". My goals were to make the world as realistic and visually pleasing as possible, as well as to keep the program running quickly enough for it to remain in real time.

The title of my project includes the adjective "interactive" because it is *the* defining characteristic for nearly every approach I took to designing the various parts of my world. The largest focus of interaction was on the "real-time" aspect of the world, as it would be in most other projects of this nature (especially with graphics).

When a graphical system does not include the real-time constraint, the product can be as complicated as necessary and can contain extremely intricate levels of detail. Real time has the objective of generating a storyline based on the user's control, whereas pre-rendered systems necessarily produce a static path through a scene, if they are even used for animation. (To "render" a scene is to draw it to the screen or an image file.) In such *non-interactive* systems, the user has very limited or no control, but the graphical presentation can be extremely realistic. If a scene takes 30 seconds or even an hour to render from one viewpoint, it is all right. A path through a non-real-time scene may take weeks to render. Movies commonly use pre-rendered 3D graphics to complement their live scenes. For example, in *The Truman Show* [24], some of the buildings only had first floors on the set, and computer graphics were used to fill in the higher floors seamlessly.

When planning my world, I constantly had to create a balance between complexity and speed. There is a limit to the number of polygons that may be rendered in a reasonable amount of time. Calculations on the computer's processor and graphics card, and the amount of memory required by the program also have limitations that must be addressed to keep response times quick enough.

I had two major goals for taking on such a task. First, my primary professional interests lie in the field of computer entertainment, so I wanted to produce an application in that area. What I had in mind was something I could show to anybody, regardless of computer science background, and get a response such as "Wow, that's really cool." My second goal was to take away from this project a new wealth of knowledge and experience that would allow me to continue entertainment programming, perhaps even professionally.

The main, overarching challenge was to develop a world that can be navigated in *real time*, and that remains realistic and aesthetic. The subsequent challenges, all of which will be described in this paper, include using textures, fog, blending, and lighting to create visual effects for water, sky, clouds, and reflections to increase realism with methods that can be managed "on the fly" for real time. For some realistic phenomena, I endeavored to build a particle engine that would be useful for snow and water systems, among others.

Modeling objects, large and small, and incorporating them into scenes served a tough task. Even the breaking up of the entire world into scenes that could be managed in real time was difficult, especially connecting them seamlessly to one another.

I had to also deal with empty space around scenes, and chose to do so by filling it with terrain, colored by height to produce realistic mountains and rivers. The terrain and other objects had to be optimized to require as little overhead as possible for the graphics card while still maintaining visual appeal.

By far the hardest of challenges was implementing *collision detection*, a means to keep objects moving through boundaries, for instance, to keep the person walking around the world from going through walls.

Some other challenges were adding forms of interaction with the mouse, use of three-dimensional sound, and adding the passage of time to the world, for realistic day-night transitions with synchronized lighting effects.

CHAPTER 2: Background and Terminology

My inspiration comes almost wholly from computer games, especially recent three-dimensional ones. These games include *Golden Eye* and *The Legend of Zelda, Ocarina of Time* for the Nintendo64 console system, and most notably *Unreal Tournament* [23] for Macintosh or Windows — a game with incredibly fast, smooth rendering, and a beautiful set of worlds. These games are put together by teams of programmers, coordinators, writers, and artists, and use the state-of-the-art graphics techniques to achieve wonderful effects. It is now common to see shadows, smoke, fire, explosions, realistic water, and even on occasion, reflections in newer games. The physics simulations, passage of time, weather phenomena, and realistic three-dimensional sound these games possess are great sources of motivation and are to be striven for.

In the remainder of this chapter I will provide some basic information about the mechanics of OpenGL [17] (the graphics libraries I used) and 3-D programming that will be necessary in order to follow the remaining chapters. Terms that are not defined in this section will be defined when they are first mentioned.

At the time of my proposal I had essentially no experience programming, or designing 3D graphics and sound. Nor had I ever used the programming language (C++) or graphics API (OpenGL, which I will explain in the following paragraphs) that were to soon become commonplace in my arsenal of tools. My experience was limited to a course on the design and implementation of computer games, a course that I took as a sophomore. This course dealt with a number of technical issues, but concentrated mostly on two-dimensional world representations.

Throughout the year I was continually picking up new abilities. I have become adept in C++ and OpenGL, as well as various sound and graphics related applications. I necessarily learned methods for creation, manipulation, interaction, and integration of 3D graphics and sound.

OpenGL is a set of libraries that allow the programming of graphics [17]. The basic constructs are points (vertices), lines, triangles, quads (rectangles), and n-sided polygons. The polygons can be filled with a color, or empty (wireframe). Each vertex of a polygon can have its own color, if desired, which creates a meld of colors from vertex to vertex. All the polygon coloring I mention uses 0.0 as the minimum and 1.0 as the maximum intensity of color. Each polygon can also have a normal — a vector that usually points away from the surface of the polygon, perpendicularly (but can point in any direction desired). Normals do not need to be specified for polygons when lighting is disabled. A small angle between a polygon's normal and a light in the scene means it will receive more light and be shaded less. Larger angles cause more shading. Normalizing a normal, or any other vector, reduces its length to one.

During explanations, I will refer to whomever the imaginary person is that is navigating the world as the "user" or the "viewer". If sound is involved, I may use the term "listener" instead. If I'm talking about the object that is in the scene at the viewer's location, I will be talking about the "camera" — the user's portal to this world.

Vertices are drawn with the OpenGL call "glVertex3f()" with coordinates for an X, Y, and Z position in the parentheses. OpenGL uses a standard three dimensional coordinate system, with the Y axis vertical, and the X and Z axes on the horizontal plane. The Z axis points towards the viewer. "Z value" is thus commonly the name used for a metric that describes how far an object is from the viewer. The Z buffer is a buffer which holds these values for every pixel that is present in a scene. The Z buffer is used for "depth testing", a function provided by OpenGL that looks at these values each time the screen is rendered and is able to draw pixels in the correct order. In other words, with depth testing enabled, if there is one building in front of another, they will get drawn in the correct order. Without depth testing, the back building may get drawn in front of the front one.

OpenGL's coordinate system can be navigated and modified with matrix operations. OpenGL has a few built in, including methods for transformation, rotation, and scale. Transformation moves around the coordinate system, rotation rotates the coordinate system, and scaling changes the size of the coordinate system. Object placement and sizing in scenes occurs with a series of such commands. Now we are ready to talk about object creation, which is the subject of the next chapter.

Having gone over some of the basics, we can now start to take a look at the issues I had and how I went about solving them. Keep in mind that every problem I had to solve was related to the universal problem of reducing polygon counts and computation while keeping the level of realism and aesthetics high.

CHAPTER 3: Creation of High Polygon Count Objects

3.1 Introduction

When I began programming objects (in this sense, a group of polygons that represent something — a tree or a wall) I quickly ran into a problem. Telling OpenGL to draw polygons is done with a series of vertex specifications, all typed in. For larger objects, this became tedious as it was taking a long time and I could not view the object as I was designing it. There are two approaches I will discuss that address this method. The first, is designing objects in a three-dimensional modeler (essentially a 3-D drawing and painting program) and converting the format to something usable by OpenGL. The second approach (the procedural approach) uses a set of functions that builds objects on the fly (each time the program draws to the screen) — which can be used for objects with repeated structure or some structure that can be described mathematically.

3.2 The Procedural Approach

For simple scenes composed of a few polygons and basic, pre-built objects like spheres and cylinders, coding by hand is probably the quicker method. Creating a room out of six rectangles and placing a sphere in the center would only take minutes.

But what if we wanted to add, say, a potted plant in one of the corners? If the plant has more than about twenty polygons (which is not many at all), it would be an extremely arduous task to sit down and type in, by hand, all the points into glVertex3f() statements. This is also especially difficult considering the lack of visualization hard coding in this manner provides. Typing a few polygons, executing the code to see if they are in the correct places, adding a few more and so on soon gets tedious.

The first scene I drew up on paper was a grove of trees encompassed by a wall, with a narrow opening at one end for an exit (see picture below).

Figure 3.1 An overhead view of the first scene I designed. I drew the circular portion of the walls procedurally (with math), I modeled the grass tufts by the stone and the ground in 3D Studio Max [3] and imported them, and the trees I downloaded from 3D Café.com [1] and imported them as well.

The walls alone were going to take me about fifteen polygons to construct, not something I wanted to input by hand. My solution was to draw the circular portion procedurally, using a loop and some basic geometrical properties of a circle. (A loop is a programming construct to do a series of similar commands one after the other.) I incremented a degree value in the loop such that new points were defined around the edge of a circle each pass, which were used to draw a series of rectangles. Then all I had to do was type in a few more rectangles for the entrance.

The benefits were twofold: this method was quicker than hard coding, and it allowed for simple alteration. By adjusting a few variables, the number of wall segments

could be adjusted to better approximate a circular enclosure, and the size of those segments could also be modified. In addition, this method was more intellectually exciting than typing in a bunch of numbers, rather mindlessly.

3.3 The 3-D Object Modeler Approach

Once the scene had walls, it needed some trees. Each tree would have a few hundred or so leaves, each leaf made of about five polygons for a total of over a few thousand polygons. I had two feasible options, hard coding certainly not one of them. I could draw a cylinder for the trunk, then make a method that draws one leaf. Using a series of controlled random transformations and rotations, I could draw these leaves around the trunk of the tree in a way we might expect them to be distributed on a real tree. This method would be considered a procedural approach, akin to the drawing of the enclosure.

One of the reasons I chose not to head in this direction was because each leaf would require a different set of translations and rotations to be put into place. After eight trees, that would be thousands upon thousands of matrix calculations. High levels of computation should be avoided because they can slow execution of the program, while one of my goals was to get the program running as quickly as possible. (A second reason I opted not to take the procedural approach was because I would have had to spend much of my time working on artistically perfecting the tree, something I did not wish to do when I could spend my time otherwise.)

I chose instead to make use of a tree object that I downloaded from 3DCafé.com [1] in Studio Max format. In order to use objects of other formats in my system, they had to be converted to OpenGL/C++ code, or a series of numbers that can be read in from a file to OpenGL/C++ code. I used an application called 3D Exploration [2] for such conversions. I created a *tree* object out of the resulting conversion. Now I just had to

translate the scene to the location where I wanted a tree, and told the *tree* object to draw itself with some scale to properly size it. This required no creation of polygons on my part, and saved much time (even though I had to make alterations to the generated code for it to work properly).

Importing models in this way also saved computational time. The entire object has its pieces all in place with proper coordinates, so no matrix calculations for rotation or translation need to be done to put it together.

Figure 3.2 Side view of one of the trees in my first scene. Each leaf is five triangles, the entire tree is made up of 4629 triangles.

Lastly for this scene I wanted to design a ground that was not so flat. I had one rectangle for the ground, and it looked too smooth. What I needed was to be able to see the object I was designing *while* designing it. This is where I first began using 3D Studio Max to create my own objects. Professional three dimensional modelers such as 3D studio max are incredibly useful and can greatly expedite object creation. I was able to

model objects in 3D Studio Max and work with characteristics of the model that OpenGL makes use of, such as face normals and texture mapping functions (which will be discussed in later sections). Using Studio Max I also designed some tufts of grass to put in the scene. Again, this was much quicker for me than typing in polygons by hand, and computationally quicker than procedurally creating these objects.

CHAPTER 4: Keeping it in Real Time

4.1 Introduction

The aspect of my project that introduces the most constraints is the ability to support graphical rendering and physical interaction in real time — as I have mentioned before the real-time aspect is what most of the project is based upon. Users can sit down and push a direction on the keypad and get virtually immediate visual response as the scene updates itself to account for the change. Without being able to accomplish this, the program would offer an altogether different type of experience, involving entirely different techniques for design, implementation, and use. This chapter covers many of the techniques I used to speed up the entire simulation in general, as opposed to specific types of effects that will be covered in later chapters.

4.2 Optimizing storage: Display Lists and Texture Objects

Display lists and texture objects are two common methods used to optimize storage of data [25]. A display list takes a series of OpenGL commands and stores them on the graphics card that is associated with the computer monitor (or display). In doing so, each time that portion of code needs to be used, it is already in the display's memory, ready to quickly dump the contents it will produce to the screen. All the large objects in the world use display lists, storing their thousands of points in the display's memory. The trees in the grove all use the same display list, so each time the screen gets drawn, it gets used many times, and reused in subsequent drawings. All model conversion through 3D Exploration [2] automatically utilized display lists, so I was able to take them for granted.

Texture Objects are another method to store frequently used information on the display adapter where they are needed. My use of them was transparent because it was built into the texture handler I was utilizing.

4.3 Face Culling

Face culling is a feature that allows reduction of polygon overhead by roughly a factor of two. Any polygon drawn to the screen has two faces — a front and a back. (In OpenGL, this is determined by the order the vertices are used to draw the polygon.) Face culling removes the back face from all designated polygons. A culled square would be visible from the front, but from the back would not be visible.

Figure 4.1 This is a view from underneath a scene that uses face culling. The floor and some walls are invisible because their back faces are facing us and thus culled, while all the ceilings and most of the walls are visible because their front faces are towards us.

Culling a few polygons makes no difference, but culling large numbers of polygons gave me a few frames per second increase in speed in some areas. I put it to use anywhere I was drawing many polygons that the viewer would only see from one

side — essentially all the ground, walls, and sky. If the user were to walk outside the boundaries of most terrain in the world, they would still be able to see the rest of the scene, because the back of the wall would not be visible to block the view. In Unreal Tournament [23] I put on a "walk through walls" code that allowed me to go outside the boundaries, and sure enough, the designers had culled all polygon faces possible. Although I used culling, it was not one of the major focuses of this project.

4.4 Dividing Scenes

Constructing a large-scale world inevitably requires many polygons and likely many textures and sounds as well. I realized early on, when developing my first scene, that it would be impossible to handle all the polygons for my whole world concurrently and retain an acceptable frame rate. Testing revealed that the graphics card I was using could handle around 40,000 triangles at a time, but no more, to keep above a threshold of about 30 frames per second (fps) with texturing and lighting enabled. (This threshold was a guideline, but by no means my definitive factor in deciding polygon count. Such things as special effects needed to be taken into account as well.) 30 Frames per second is good, while around 45-60 is very good. To keep above my frame rate threshold, I had to construct the world out of smaller pieces, which I refer to as "scenes". Each scene would have its own 40,000 triangle limit and would be handled individually by the graphics card while ignoring the rest of the scenes. In this way, a world with p triangles would require $p/40{,}000$ scenes. Most adventure type games are logically broken up this way. Even older, two-dimensional games like *Final Fantasy VI (Japanese Version)* [6] are typically broken up into various scenes that are filled with as much detail as possible without hitting some upper limit (polygon count in the 3D case).

There are many benefits and only a few negatives to this approach. The greatest benefit is achieving the expected outcome – being able to keep polygon count to a

maximum as well as retaining a high frame rate. Although this approach initially makes programming somewhat more difficult, in the long run the program becomes more modular and extensible, allowing for simpler addition of new scenes.

Each scene was then able to have its own objects, sounds, textures, and methods associated with it. The second scene I started designing was the path that the grove leads out to. Notice that this scene (below) contains no trees. None of the trees from the grove have to be drawn when viewing this scene, and since this scene has no trees, none of the tree objects even have to be loaded into memory, thus allowing more room for other objects. In the same way, there is a limit to the amount of texture memory a graphics card has. The second scene does not use any grass textures so they are not loaded, leaving room for the textures that are needed. Also, the sound card (if one is used, otherwise RAM is used) has limits to the amount of sound that can be loaded, another reason to keep sounds unique to a scene associated only with that particular scene.

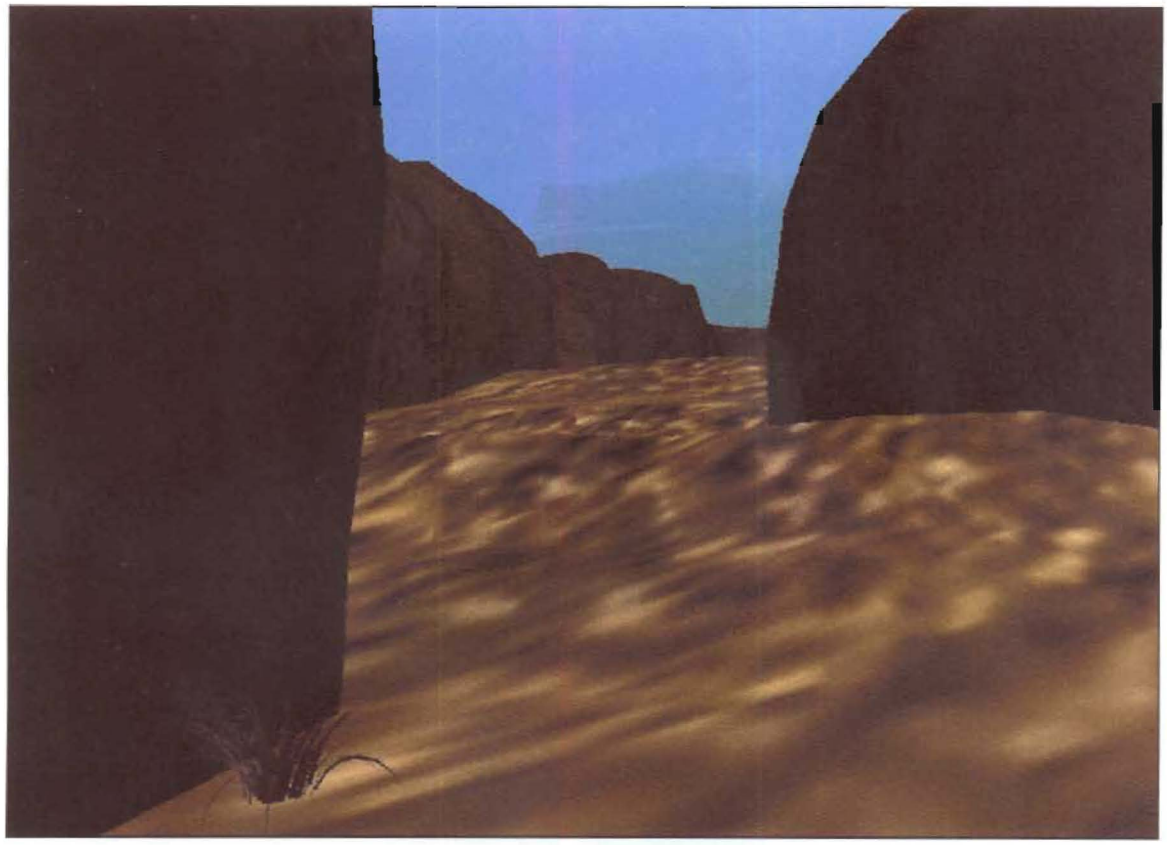


Figure 4.2 A view of the bumpy walls and ground, with barely distinguishable mountains in the background.

What this leaves is a way to keep very detailed scenes, with only the necessary associations, allowing for unification to a larger, more impressive whole. With scenes now separated from one another, one might wonder how this affects interaction with the world in real time. Moving from scene to scene will require a pause while the new scene loads. Depending on size, this may take anywhere from a fraction of a second, to a few seconds. All my scenes take only part of a second to load, not detracting from the real time aspect of the world. The longer someone has to wait, the more likely it is that they will notice this lag, which is not desired.

The only detail left to worry about in respect to scene transitions is how to unify all the scenes. I have devoted a separate chapter to discussing just this since it is a fair amount of information.

CHAPTER 5: Scene Unification

5.1 Introduction

Planning for scene unification should start, as I have come to realize the hard way, before each scene is created. I started planning early, but I did not take a thorough approach at the time. First I designed all the scenes, then I put them all together. This caused a belated discovery of certain facets of unification that I should have had in mind during scene design.

Transitions from one scene to the next must progress smoothly both visually, and technically in the coding. The simpler of the two to handle is the technical portion, if the scenes are programmed in a clean and modular fashion. All that needs to be done technically is, upon detecting the camera at the transition point (see chapter on collision detection), memory cleanup of the old scene, loading of the new scene, and switching execution to render and update the new scene.

The more difficult element of unification is the visual one. In the real world, it is not usual for a person to walk somewhere while looking straight ahead and not see a portion of the area in the distance. If a person is about to walk through a doorway, they see part of the room they are moving into. Even if there is a closed door, upon opening it some of the next room can be seen from the current one. For a virtual world to be more realistic, the same rules must apply — one scene must be visible from any scene directly connected to it (in the way common sense would dictate).

We cannot simply draw the entire connecting scene(s) adjacent to the current one, because that completely defeats the purpose of breaking the scenes up to begin with. I have investigated three valid approaches to handle this with minimal computational overhead, although none of them are necessarily quick to implement.

5.2 Transports

The first method to handle scene unification is to completely avoid such connections. Using something I call "transports" (conceptually a matter transporter, like in *Star Trek*) the user can be transported between scenes without seeing the new scene beforehand. Their use is restricted by genre, most likely to be found in science fiction worlds. Many multi-player games such as *Quake* and *Unreal Tournament* [23] use transports, especially for relocating to various places within the same scene. Visual connections can also be avoided by creating a world where an event sequence causes transitions, as opposed to approaching an adjacent scene. For example, in the game *Siphon Filter*, a special operative is set in one scene and must complete checkpoints such as disarming bombs or saving hostages. When all the checkpoints in one area are complete, game play stops, and soon the player is found in another scene with new checkpoints (possibly far from the first scene in *Siphon Filter's* world). This style would not have suited my needs, since all my scenes are directly neighboring other scenes.

5.3 Transitions Situated Around Corners

The second approach is to have scene transitions around corners, or situated in a way that only some minimum amount of the next scene is visible. This type of layout where the bulk of the scene lies around a corner can be seen in the overhead picture of the grove in chapter 3. This approach then takes the necessary polygons from that scene and attaches it to the current scene, making sure to trigger transportation before the camera gets close enough to see around the corner or to where the rest of the scene should be.

This method is appealing because it quite visually accurate. However, it is also very difficult (and does still require extra polygons). It is especially difficult in a world that uses *first-person perspective* viewing. In this perspective, the camera may be looking straight into the next scene, as opposed to a third-person perspective where the camera

may be at a slight angle to the ground or straight down and taking in less of the next scene.

5.4 Screen Snapshots and Transfers

The third approach is the one I chose to implement. I made sure to have most transitions situated around corners or in narrow paths to reduce the amount of inter-scene visibility. Then I took snapshots of the scenes from their entrance points, and "cut out" the sky using alpha values. An Alpha value is an optional component of an image (or polygon) that dictates a portion or level of transparency (see *Visual Realism – Textures with Alpha Channels*). I was then able to texture a rectangle located at the edge of the current scene with a snapshot of the next scene. Using alpha values, I made portions of these textures transparent, so the sky and background could be seen through the "cut out" parts.

Another reason to reduce the view of the next scene is that these textures are limited in detail compared to their three dimensional representation. In addition, these images are best viewed from the position at which they were taken. If part of the snapshot is a tree off in the distance, walking directly up to the texture reveals that the tree is not in the distance at all, but painted on a flat mural.

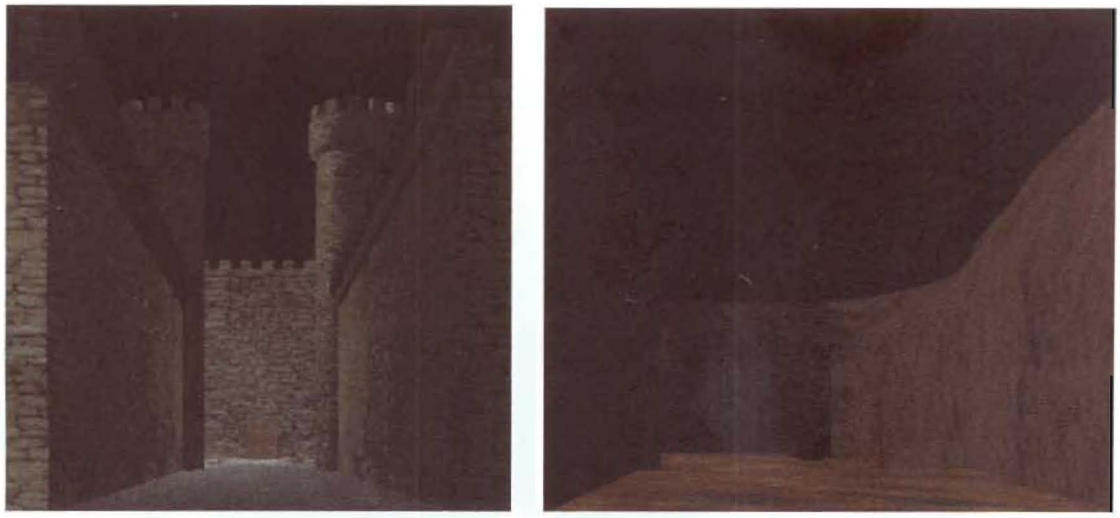


Figure 5.1 Two snapshots (from opposing scenes) used in the world for scene transitions. The black areas are the "cut out" areas.

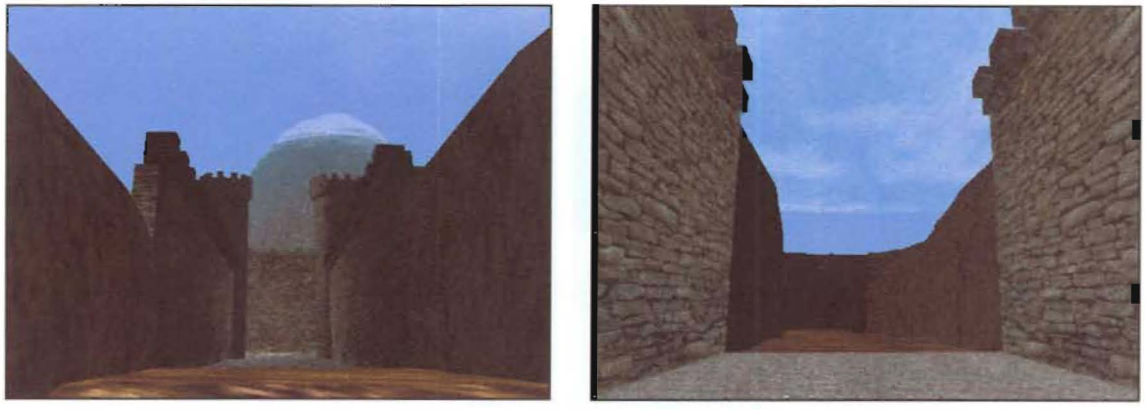


Figure 5.2 These two pictures show the transition snapshots actually used in the 3D scenes to create visual transitions. The black parts from the textures are transparent, showing the sky and mountains behind.

To address this issue, I decided on a policy of preventing the camera from getting too close to the snapshot. To enforce this, if the user brings the camera too close, they will automatically "transport" into the connecting scene. Such a transition requires a trigger point to move to the next scene outside of this limit, *and* an arrival point even further out for returning from the other scene. If the return point is too close to the transition's trigger — within the radius of the camera (see chapter on *Collision*

Detection) — then upon appearing in the scene, the trigger will immediately send the camera back to whence it came. Also, if the return point is too far away, it looks like you appear in the midst of the scene.

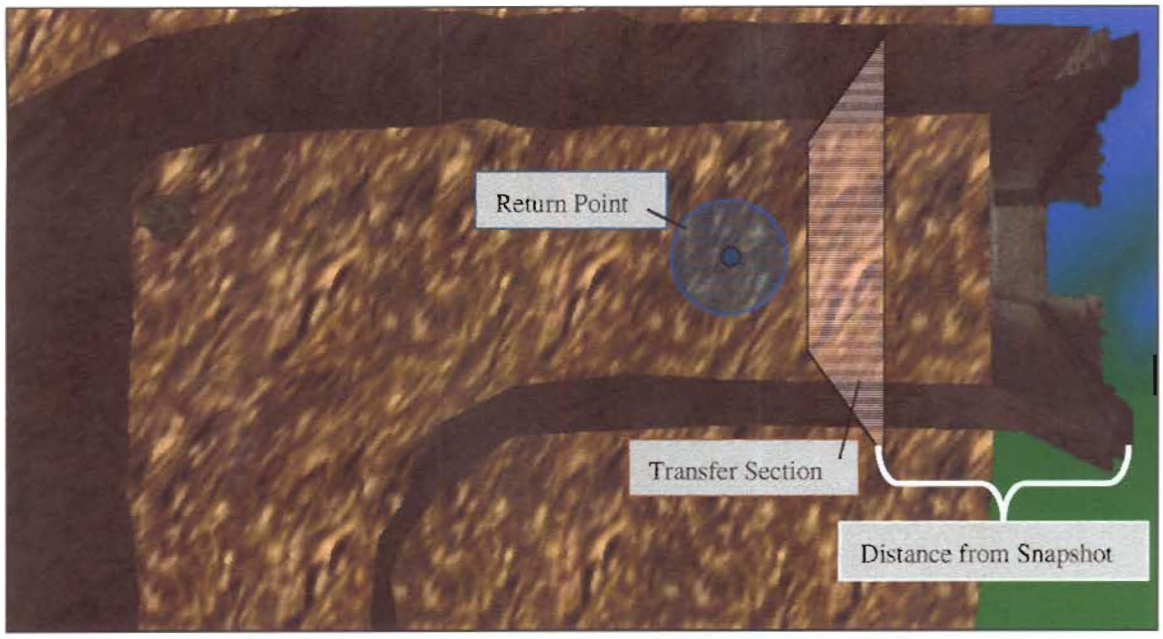


Figure 5.3 Overhead view illustrating the mechanics of a transfer section and return point in one scene.

Notice from the above diagram that this arrangement requires quite a bit of room. To accommodate this layout, I had to modify some of my scenes to stretch out their length near transitions.

Zelda for the Nintendo 64 console is a third person perspective game that employs an interesting technique similar to my method. The difference is when the player approaches the transition point, the camera freezes, and the character keeps walking towards and through the textured rectangle. However, the character is drawn after the texture with no depth testing comparison to it, so the character is seen on top of the

texture even when it is logically further away than the texture. This creates a great illusion of depth with the texture that cannot be created with my first person perspective.

CHAPTER 6: Adding the Element of Time

Something I thought would make the world even more realistic is if it changed with time akin to our world. With progressing time in our world trees grow, water flows, and day turns to night. By implementing a flow of time, I could at least create day-night transitions.

The element of time within a virtual world is a completely different topic than the aspect of executing in real time. Real time focuses on optimizations and design in such a way that user-world interaction can take place near instantaneously. The world actually having its own time, on the other hand, means being able to program characteristics of the world to change and be updated based on the flow of this time. As a matter of fact, the flow of time in the virtual world could be completely independent of the rate rendering occurs. It is possible to have one without the other, both, or neither. In a virtual world that does run in real time, adding time allows for more possibilities and creativity.

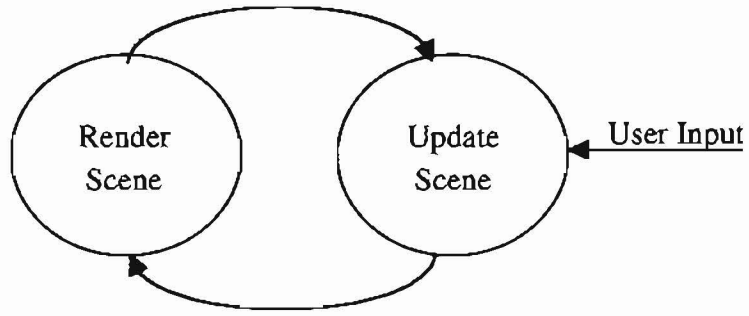


Figure 6.1
This figure illustrates the constant loop of scene rendering and updating. The update method receives user input.

In the virtual world, before everything is drawn, I call an *update* method that makes any necessary alterations from one time frame to the next. If the user is holding the forward button, the update method makes sure to move the camera in the scene such that it appears the viewer has moved forward. An update must be invoked every time a new animation frame is rendered, otherwise the view may appear slow or jumpy. If no updating is done, there is no reason to redraw the screen anyway, since it will be exactly the same (and hence wasting precious computations on the graphics card). Making updates in the update method once every time frame can provide time in the world that is dependent only on the combined speed of one update and one screen drawing.

I used this setup for a while, but it did not make much of a difference at the time because it was not critical to control the speed of the few updates I had. This may be completely acceptable depending upon the nature of the world, but the more changes that are taking place, the more likely it will be helpful to govern the frequency at which they take place. The one update that manifested this lack of control in my world was related to the vertical camera movement that simulates the position of a walking person's eyes through each step. Towards the middle of scenes the graphics card does more work, giving a slower frame and update rate – this is the rate on which I had based the movement. The camera appeared to move normally until approaching the very edge of a scene, when it would start to bob up and down very rapidly and unrealistically because the under-worked graphics card sped up rendering.

According to Game Architecture and Design [19] there are two approaches to *decouple* rendering from updating. One, called *semi-decoupling*, keeps the updates at a constant frequency, while the rendering occurs as quickly as possible. The other, *full-decoupling*, again renders as quickly as possible, but also has updates going as quickly as possible with some measure of time (perhaps the inverse of rendering frequency) to control the amount to update objects.

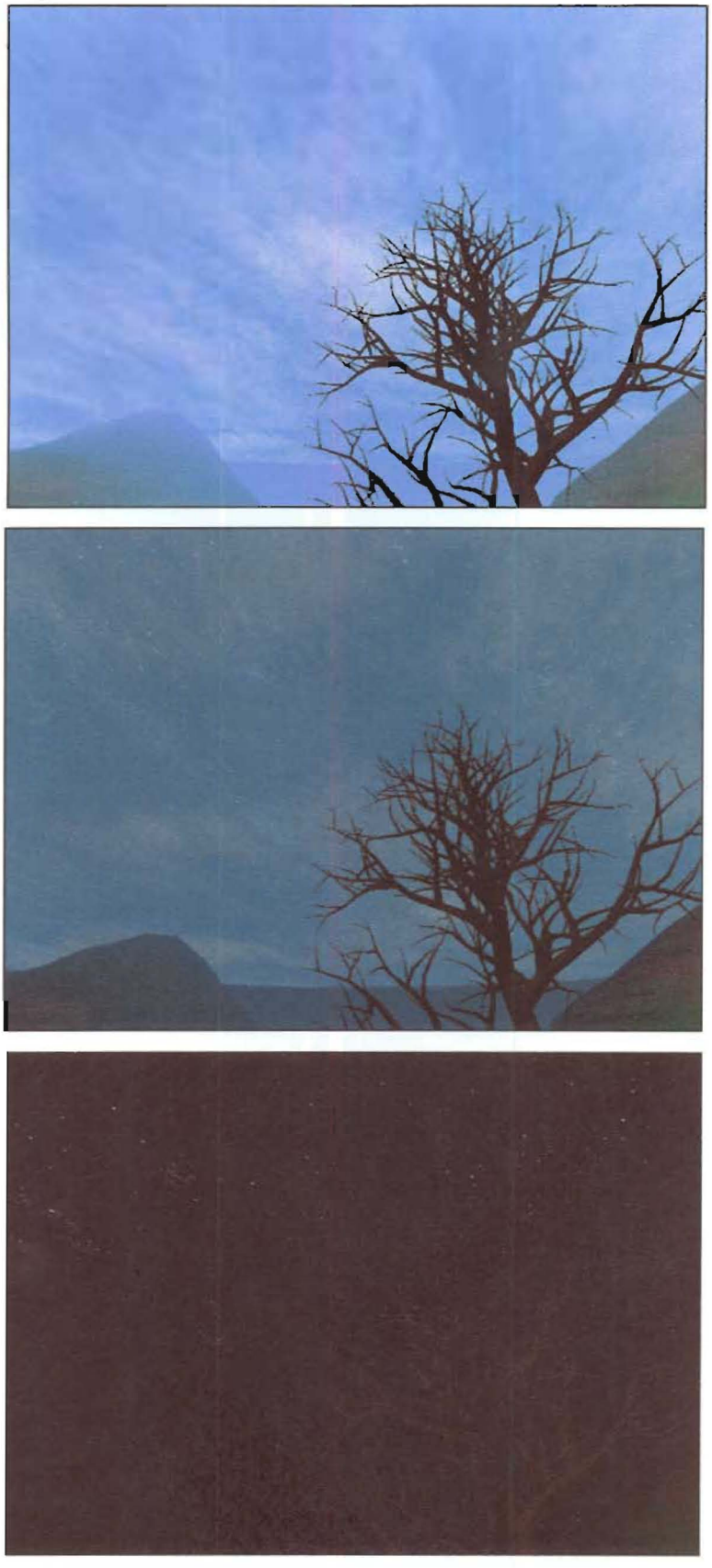
Interestingly, without researching these methods before hand, I employed a technique that is a hybrid of full and semi-decoupling. Rendering and updating follow each other sequentially as quickly as possible, and I let the update method know the current time when it executes. Time difference since the last update may be figured out, but that alone is not sufficient for everything. Say you were designing a simulation of a person throughout the day. If that person normally wakes up at 6:00 am, it would not be sufficient to know only the amount of time that has passed since last update — the actual time of day is needed.

Another example of how this is useful would be the operation of the sky in my world. What the sky looks like at any point during the program's execution is a function completely dependent on the current time, which is given to the update method as military time in decimal form (e.g. - exactly 1:30pm in my representation is 13.5000...). The sky goes through various transitions, most notably a light sky with clouds in the day to a dark, starry sky at night. If a sky update occurs at 6:21 am (6.35000...), the following conditional from my program, represented as pseudocode, gets executed:

```
if (military time > 4:30am AND military time <= 6:30am){
      // then this is sunrise, the sky needs to get lighter
      interval = (military time - 4:30am) / 2.0;
      // interval is now a number from 0.0 for the beginning of sunrise
      // to 1.0 at the end of sunrise.
      visibility of day sky = interval * maximum visibility;
      // this last line fades the day sky in from 4:30 to 6:30am
}
```

(Note: lines beginning with "//" denote a comment)

Figure 6.2 Day – Night transition



This is far less than everything that happens to the sky during update, but it helps illustrate the point that rather than having a finite set of states, updating based on current time provides the ability to produce a continuous spectrum of states. As the day sky fades in during the early morning hours, its opaqueness is determined by where in the interval between 4:30 am and 6:30 am the current time falls. At 4:30 am its opaqueness is multiplied by an "interval" factor of 0.0, giving it 0.0 opaqueness, so the night sky (which is always present, just slightly above the day sky) shows completely through. At 6:30 am the opaqueness is multiplied by a factor of 1.0 for a completely opaque and visible day sky, blocking out the night sky. All points in between are a blend of the day sky with the night sky behind it. All this happens seconds of the viewer's time — the virtual world's time can move at any speed; a full day in the world passes in only a few minutes of user time.

CHAPTER 7: Visual Realism — Textures, blending, lighting, fog

7.1 Introduction

Part of the original goal was to design a world that, although optimized to run in real time, remains "realistic and aesthetic" as well. Moving water, objects fading into the distance, windows, and reflections are all effects for added realism in the world that I will discuss in this chapter.

Texturing (applying images to surfaces), blending (transparencies), lighting, and fog effects are all features of OpenGL that I have spent time working with. Texturing is by far the most important tool of the four, but working well with all of them provides a means to produce wonderful visual realism in real time. Many of them go hand-in-hand for various techniques, and that is why I have included them all in the same chapter.

7.2 Textures

Texture mapping is the ability to take an image, and through some function, map it onto the face of a polygon. It is almost certainly the most appealing capability of current three-dimensional graphics APIs. To compare, look at the two screenshots below, the scene on the left with textures disabled, and the same scene on the right, but with textures enabled. Notice the incredible difference a few textures can make.

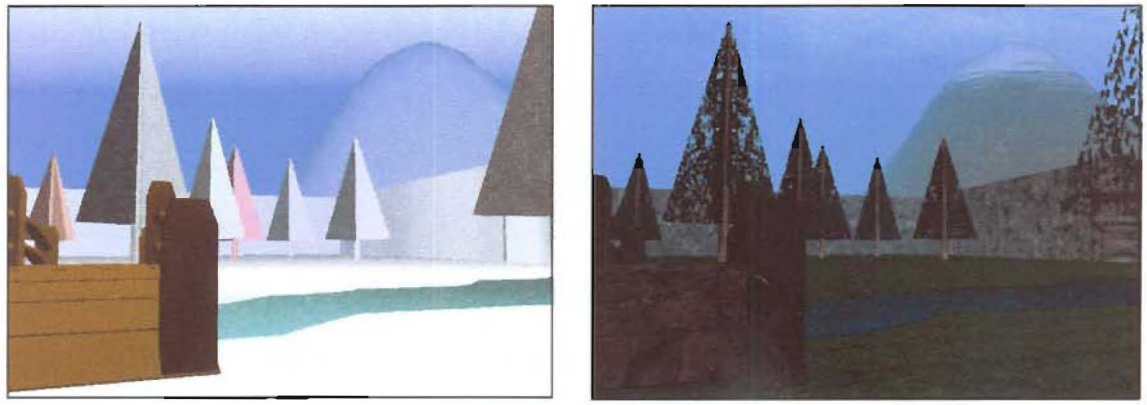


Figure 7.1 Left: "Flat Shading" in a scene with no textures. Right: The same scene with textures.

The first time I made the jump from a scene of flat colors to one coated with detailed textures, I was using built-in objects with pre-defined texture coordinates, including the famous teapot seen in many three dimensional demonstrations[8]. I realized that texture mapping would be a great tool, but it was not until I understood its inner workings that I became aware of its full potential (such as being able to move on an object).

One advantage of texture mapping is the amazing detail it can add to objects. Used correctly, it can save time and polygons (so they can be used elsewhere) over representing the contents of the image in three dimensions, and yield excellent visual results [21]. For example, to create a visual effect similar to that of the stone wall in my grove scene without using textures, the wall would have to be constructed out of three-dimensional stones with careful attention to coloring and placement. Although such a design may look good, this would be very time consuming and waste many polygons on an insignificant object that is not supposed to draw much focus (like most walls that merely are present to provide bounds for a scene).

The surfaces of most static objects in my world are covered with textures — the ground has grass textures, dirt, sand, and stone textures. Walls are usually some sort of

rock-like texture. Some trees have bark, wood constructs have a visible grain, and window grates are dark metal.

Only a few items — individual strands of grass, and distinct leaves — are able to make up for lack of texture with moderate polygon counts (along with somewhat expected uniformity of color). Each grass tuft seen is composed of a total of 912 colored triangles, which is a lot of detail for a relatively small object. Forgoing the adornment of texture on these tufts is thus acceptable. Polygons provide detail in depth of structure, whereas textures provide detail in a flat sense. A balance between the two is sometimes better than just making use of one or the other. For example, *The Legend of Zelda: Ocarina of Time* presents tufts of plant life using a few rectangles with moderately detailed textures mapped on them [7]. The textures have transparent sections. View the comparison below.

Figure 7.2
High-polygon-count grass tuft (right) made of 912 green triangles. Much depth of structure can be seen. Also, notice the triangles with faces pointing away from the light source are shaded.

Figure 7.3
Plain grass texture (left), essentially mapped on one flat polygon.

Figure 7.4
Plant (top-left) from *The Legend of Zelda: Ocarina of Time* viewed from the side at a slightly elevated angle. The plant is composed of six rectangles textured with the same image (bottom-left). The white portions are transparent. Viewed from directly above, the arrangement would look like the pattern on the bottom- right

An early idea I had was to use textures to create a virtual art gallery. This turned out to be quite simple in theory and in practice. The idea was to take image files of various works of art related to the theme of my world and "hang" them up in a hallway for real-time viewing in three-dimensions. One barrier was that textures can only be loaded in OpenGL with dimensions that are powers of two. Since this power of two proportionality usually does not match up to any artwork, I had to keep track of the true dimensions of the piece before loading, then scale the texture mapped rectangle in order to correct for this (with the glScalef() matrix operation). The following code demonstrates the scaling done for the John Waterhouse painting *Pandora's Box*, with relative dimensions 497 units wide to 875 units tall:

```
glScalef(1.0f, 0.72f, 1.0f); // scales the picture to square
glScalef(1.0f, 1.76f, 1.0f); // adjusts to original 497 x 875 ratio
glScalef(4.0f, 4.0f, 1.0f);  // quadruples the picture's width and height
```

The texture's (0,0) coordinates are mapped to the bottom left of the rectangle it is drawn on, and its (1,1) coordinates are mapped to the upper-right corner of the rectangle. Again, this is a simple technique that has attractive results.

Figure 7.5 *Pandora's Box* — a painting that hangs on the wall of the virtual art gallery.

There is another "cookie-cutter" method for storing textures that would allow maintenance of aspect ratio. This is done by taking the original image, and adding empty space to the dimensions in a paint program until they are powers of two. Then using texture coordinates that are the ratio of each original dimension to the new dimension, the image can be cut out, leaving the white space unused. I chose not to use this method because it required slightly more work (adding white space). However, this method is useful for storing many textures that can be retrieved by some mathematical function to specify a portion of the texture to return. A use for this would be the alphabet stored in one texture (see below).

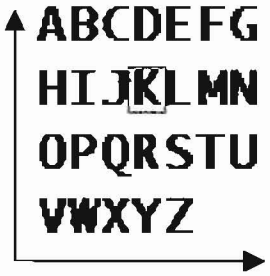
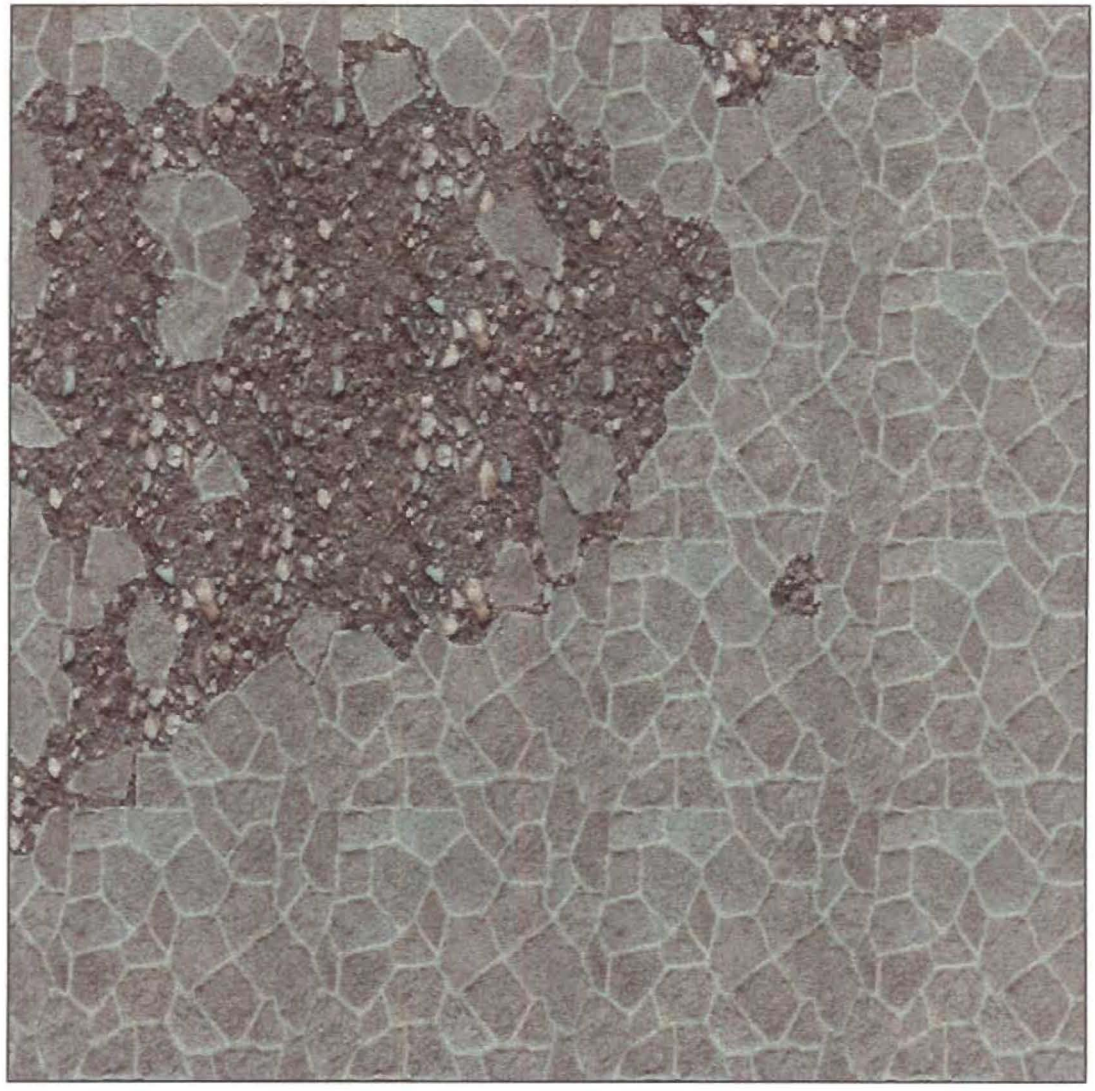


Figure 7.6
If an image of the alphabet is loaded as one texture, a mapping can be made from each desired letter to a particular portion of the texture to draw. The texture coordinates are (0,0) in the lower left, and (1,1) in the upper right corner. Letter number 11 (K) has coordinates of (3/7, 2/4) in the lower left and (4/7, 3/4) in the top right (K is four letters from the left in rows of 7, and the third letter from the bottom in columns of four).

In the castle courtyard I wanted to make the ground more interesting than it was with only one texture. There is no reason surfaces (a surface may be made of multiple polygons) should be limited to one texture. In the back-left corner of the courtyard, I wanted to have the stone floor break away to reveal the earth underneath it. This could be done by replacing the rectangle's texture in the corner with a dirt texture, but the transition between the two would be very sharp and obviously unnatural. Instead, I took the stone floor texture I was using and, in a graphics program (Paint Shop Pro), layered it on top of the dirt texture I desired. By cutting out portions of the top stone texture, removing some completely, and pushing and rotating other pieces, I was creating the natural transition I was looking for. When satisfied, I combined the layers into one image and loaded that as the texture for the far-left corner of the courtyard. Notice that when I was cutting pieces away, I had to make sure to leave any edges of the texture that would be adjacent to the normal texture of the ground undisturbed, so as to create the transition completely within the image I was engineering. This process is illustrated below.

Figure 7.7 The stone texture (top-left) was placed on top of the dirt and pebble texture (top-right) in a paint program. Then pieces of the top layer were cut away and rotated to produce the texture below.



This manual combining of images to produce one ad hoc texture works well, and can be applied in many situations. Unfortunately, it took too much time for me to work with it extensively. I did, however, take the idea one step further, and planned the whole outdoor ground by the art gallery to be one texture. There is a path of bricks leading from the door to and encompassing the tree, with intentional breaks to give the look of aging. I completely laid out the polygons of the scene beforehand; only after taking measurements was I able to use the paint program to design the texture in a way that it would fit correctly.

Figure 7.8 The grass, brick path, and tree soil is all one large texture engineered specifically for this

This technique is commonly seen in three-dimensional games to represent features that would normally be flat anyway, such as papers on the surface of a desk, seashells in sand, or worn paths on the ground similar to the path in my gallery scene. In Tony Hawk's Pro Skater 2, for example, scenes are littered with graffiti that is painted onto the texture of the background [22].

7.3 Dynamic Textures

Until now I have discussed objects in my world that all have constant texture coordinates — rocks, trees, and still items. This was not suitable enough, as I wanted some way to create the effect of flowing water or moving clouds. Desirable special effects can be created by a method of allowing the texture's mapping onto the object to change with respect to time, that I will refer to as *dynamic texture mapping*.

Peering into the well of the castle courtyard, one can see that the surface of the water appears to be constricting and expanding gradually. Also, in the large valley with the evergreen trees, the brook flows calmly through the scene. The polygons that the brook and the well water are represented with do not, themselves, move. The appearance of movement is created through using variable texture coordinates when mapping to the polygons. For example, the brook is drawn with its width's texture coordinates incremented each update, so the width of the image slides across the polygons that make up its surface.

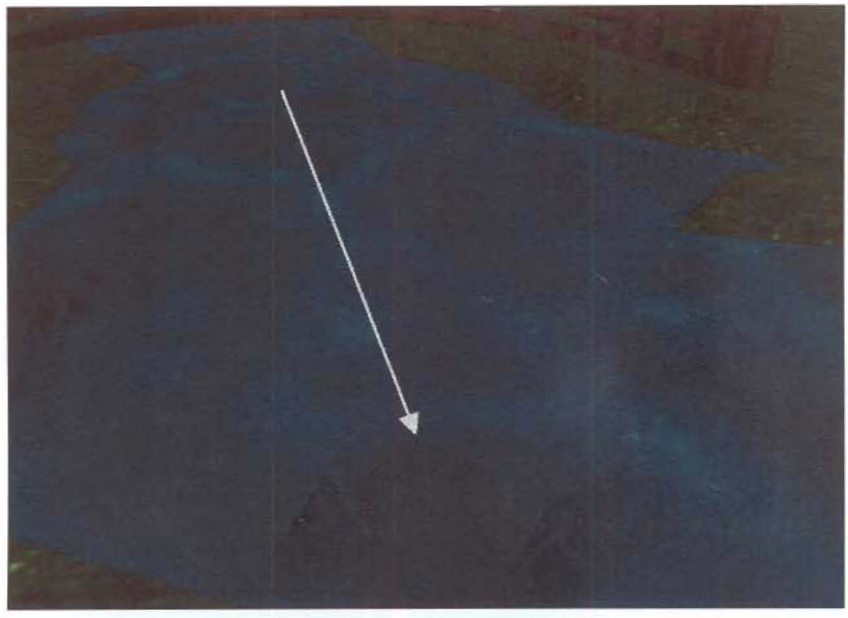


Figure 7.9 The arrow points in the direction the dynamic texture makes the water

I also did not want the water to be completely opaque, as it is not in nature. A feature of OpenGL called *blending* allows for polygons to be transparent, blending with any color values of polygons previously drawn behind them in each screen drawing. Blending provides a way to give objects a realistic watery or glassy appearance.

7.4 Blending, Lighting, and Multi-Texturing

The sky in my world makes use of blending to place multiple textures on one polygon (often referred to as *multi-texturing* [14]). By looking at my world's sky during the day, you will notice two layers of clouds, different sizes, moving in slightly different directions at different rates of speed. There are two sets of polygons, each with their own set of clouds. The cloud texture is the same, just used differently by each set of polygons (which I will refer to as *sky1* and *sky2*). *Sky2* is drawn first, completely opaque, with dynamic texture coordinates moving the clouds in one direction. *Sky1* is then drawn in exactly the same location, overlapping *Sky2*, but with an alpha value of 0.5, meaning 50%

opaque. *Sky1* also has texture coordinates that are half the value of *Sky2's*, meaning it only fits half as much of the texture onto itself. The texture therefore appears twice as large with bigger clouds, and looks to be closer to the viewer than the other set of smaller clouds. In addition, *sky1's* texture coordinates are incremented with a larger value, causing the clouds to move more quickly across the sky. All of this gives the impression of two layers of clouds, one much closer than the other, even though it is really a projection onto a flat plane.

For even more realistic effects, I wanted to take what I had done with the sky and modify it so that it would be bright and moderately cloudy during the day, but dark and starry during the night as well. Via the time dependence on the alpha values I discussed in chapter 6, I did just this. As the day gets later, both of the sky layers in the day slowly fade out to reveal a night sky placed slightly above. The process reverses at dawn. By encapsulating the world with a fading blue cylinder inside a stable black cylinder, the entire horizon and background became part of the night-day effect as well.

In my initial implementation, while the sky and background turned dark at night, the rest of the scene stayed just as bright as before. When viewing the sky during this transition, the illusion that other objects in the scene dim as well is created because this is what we expect to happen. Turning this from an illusion into a tangible reality was the next logical design step.

OpenGL provides up to eight lights for use at any one time. Lights provide shade for polygons based on their angle to the light (determined by a normal vector for each polygon or vertex).

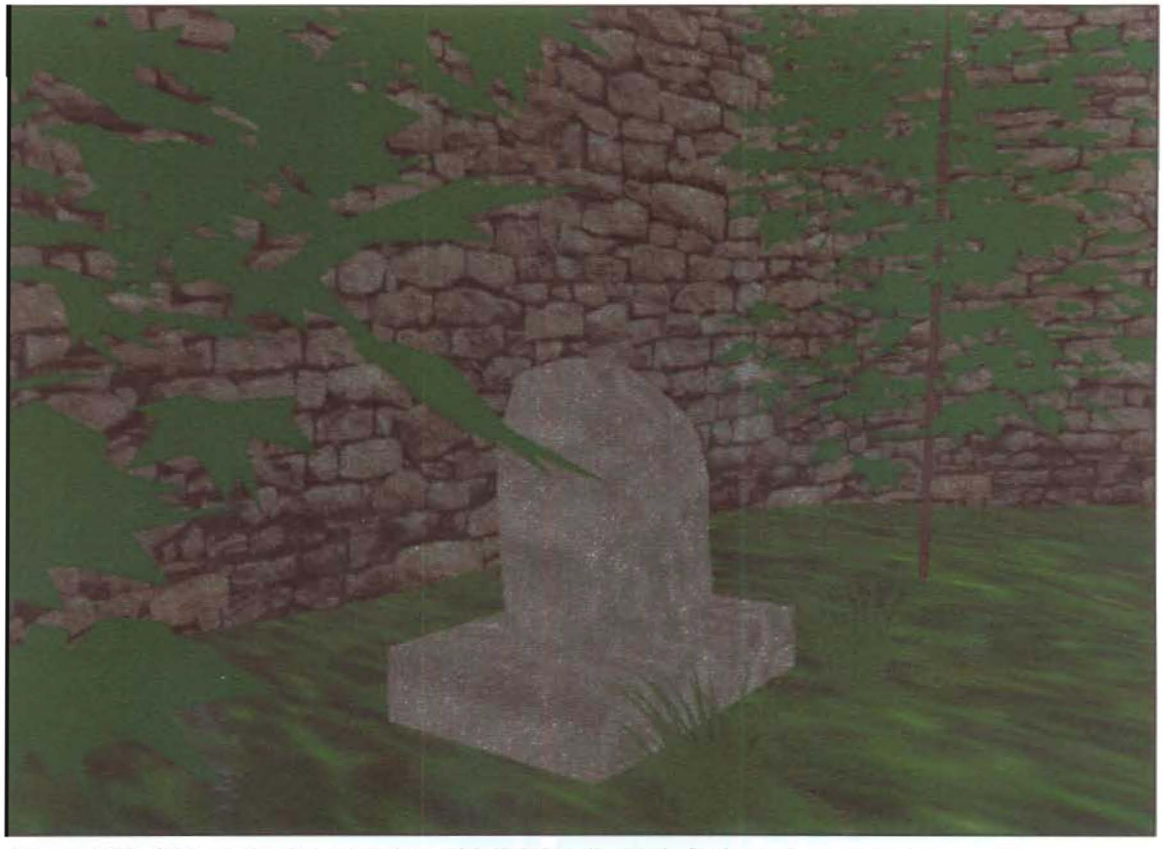


Figure 7.10 This screenshot was taken with lighting disabled. It almost hurts to look at.

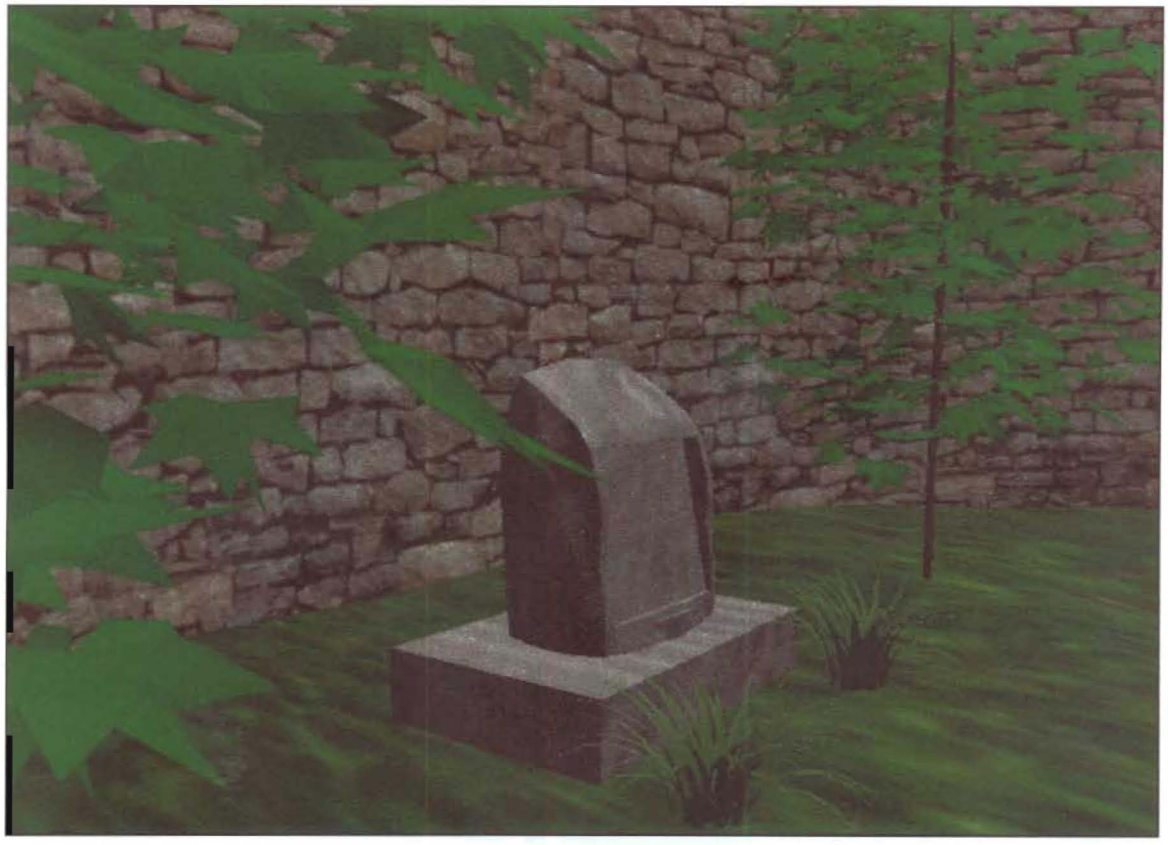


Figure 7.11 This picture shows the same scene with lighting enabled. Notice the shading that was not present before. There is much greater detail in the grass, stone, and leaves.

I designed my scenes to have the first of these be the default light, the "sun" that is intense enough to illuminate an entire scene. By reserving this light for the sun, I was able to synchronize the changing sky with this particular light using a procedure resembling that of the fluctuating translucence of the day sky.

7.5 Sphere Mapping

Something else I was looking to somehow add to the world was a method of real-time light reflections. One method of doing so with minimal computational overhead is known as *sphere mapping*.

Sphere mapping is a mode of texture mapping that takes a texture and projects it onto a specified surface as if the surface were a completely reflective sphere. The "rays of light" from the texture are projected parallel, as if from infinitely far away, and with an infinite focus [15]. The effect this gives is the appearance of the surface reflecting that texture. It is a quick approximation for reflections when you know what will actually be reflected, although it is not nearly as accurate as ray tracing. Ray tracing simulates many light rays projected from the eye of the observer, and uses accurate physics to determine the destination of each ray. This is computationally expensive, but produces very realistic images. Since this cannot be done in real time on consumer hardware, sphere mapping can be used to approximate the reflection. Sphere mapping is especially well suited for curved surfaces, on which humans cannot notice odd reflection nearly as easily as on flat surfaces.

Games commonly use sphere mapping (or similar functions) to give surfaces a shiny appearance without intention of reflecting a portion of the scene. In *The Legend of Zelda, Majora's Mask* a small mound of gold dust is sphere mapped with a golden-yellow texture with lighter and darker areas. When viewed from different angles, the appearance of portions of the mound change from lighter to darker gold giving the surface a shiny appearance. Glass window panes and beakers in *GoldenEye* are also mapped with light white and gray textures in this manner, combined with blending, to make the glass look as if it is reflecting light at various angles.

Figure 7.12 Sphere mapping on a sphere for feigned reflection.

I used sphere mapping to create a reflection on a hemisphere lodged in the side of a wall. To do this properly, I positioned the camera where the sphere was to be located, and snapped a screenshot for the texture, effectively capturing the "light" that would be incident on the sphere. Then I applied it to the sphere and lodged it in the wall so only half could be seen. More than half would reveal that it was reflecting the same image no matter where it is viewed from. To make it more interesting and add the illusion that the reflection updates in real-time, I took a picture from the same location at night, and applied it to a slightly smaller sphere which I placed inside the day sphere. Synchronously fading the day sphere with the day-night transition reveals the night sphere underneath, but it is so close in size to the day sphere that it appears to be the same one, and thus changing reflection with its surroundings.

7.6 Reflections on Flat Surfaces

At this point in time (2001), it is impossible to create reflections through physically real methods (i.e. – ray tracing) in real time on consumer hardware. That is one reason, from my experience, reflections are often not seen in real time. Unreal Tournament [23], Tony Hawk's Pro Skater 2, and Perfect Dark all have some reflections on flat surfaces, but use them sparingly. I discovered the preferred method, or "trick", for creating the illusion of a real reflection from the NeHe tutorials [11]. It is relatively simple, and produces visually accurate reflections in real time.

The "trick" to doing this is only a few steps:

1. Determine the plane the reflective surface will lie on.
2. Take any objects that will be reflected, and produce a way to draw them with their coordinates flipped over this plane. If the plane were the Y=0 plane, all this would entail is changing the sign of all the Y coordinates of the object. Do the same for the normal vectors.
3. Draw the normal objects, with the normal lighting.
4. Draw the "reflected" objects, using the stencil buffer if necessary to cut out the portion of the screen to draw to that corresponds only to the reflective surface. To be as accurate as possible (but this is not necessary) reflect all the light positions as well before drawing these objects. (The stencil buffer allows drawing to take place only in specified portions of the screen. For a more in-depth explanation, see chapter 7.12 *Fog and the Stencil Buffer.*)
5. Lastly, blend the reflective surface in its place, so the reflective objects can be seen partially through it.

I employed this technique to create a reflection on a floor, and it came out very nice (see the images below). Note that I did not have to use the stencil buffer because the viewer cannot move to any position so as to see the reflected objects from anywhere except through the floor.

Figure 7.13 Reflections in the floor of a potted tree and window (above), and a wooden chair (below).

Figure 7.14 This is not an image accidentally inserted upside-down, but rather the same scene viewed from underneath the floor, looking upwards. The objects are completely mirrored beneath the floor.

7.7 Textures with Alpha Channels

Each pixel of a texture can be defined to have a red, green, blue, and an alpha component to determine exactly how that pixel is displayed on-screen. The alpha channel, like blending, designates how transparent a pixel should be. Great things can be done just by designating certain pixels to be completely transparent. Imagine telling OpenGL to draw ten green triangles within a rectangular boundary in two dimensions as in the image seen below.

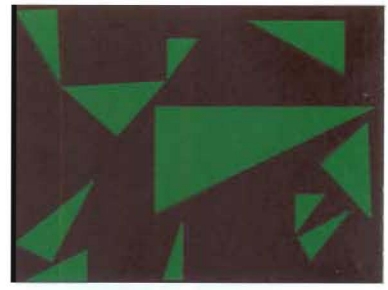


Figure 7.15 The black portion of this image can be specified as transparent.

To get the same effect, you could alternately take this image of all the triangles with the black areas designated as transparent, and map that onto a rectangle. This reduces polygon count and saves programming time.

In the world I used these textures in two places. The trees in the valley have foliage that is mapped onto four large triangles for each tree. They do not look nearly as pretty as the very leafy trees, but use less than $1/100^{th}$ the number of polygons to draw. Had a texture artist designed the foliage rather than myself, it would look much better, and still only need a few polygons.

A better looking example is the set of metal window grates in the gallery.

Figure 7.16 This window grate is a texture with the alpha values mapped onto a rectangle.

They are each one rectangle mapped with a texture with alpha channels.

The related challenge for me was getting such textures to load. OpenGL does not perform any image loading functions, that is left up to the user. I had a third party package for loading TGA format images, but it did not support alpha channels. Without knowing the first thing about file formats, I took it upon myself to tailor the package for my needs. I ended up taking arrays of 24-bit pixels for each image (1byte/8 bits each for red, green, and blue channels), and resizing the array to accommodate the same number of pixels with 32-bit size. I inserted another 8 bits for an alpha value on the end of each pixel while making sure to slide the other pixels back in the array, unharmed. After doing this for each pixel, the array becomes completely full. To make use of the new accommodating texture (represented as an array of pixels), I designed a function that would take a red, green, and blue value as input, and turn that composite color completely transparent by iterating through the pixel array searching for matches. From what I have seen, it is common to use black as this color. For the trees, I used a paint program to design the texture, and made all the desired see-through portions black (RGB components 0-0-0). Then I loaded the texture, specifying color components of 0-0-0 to be transparent.

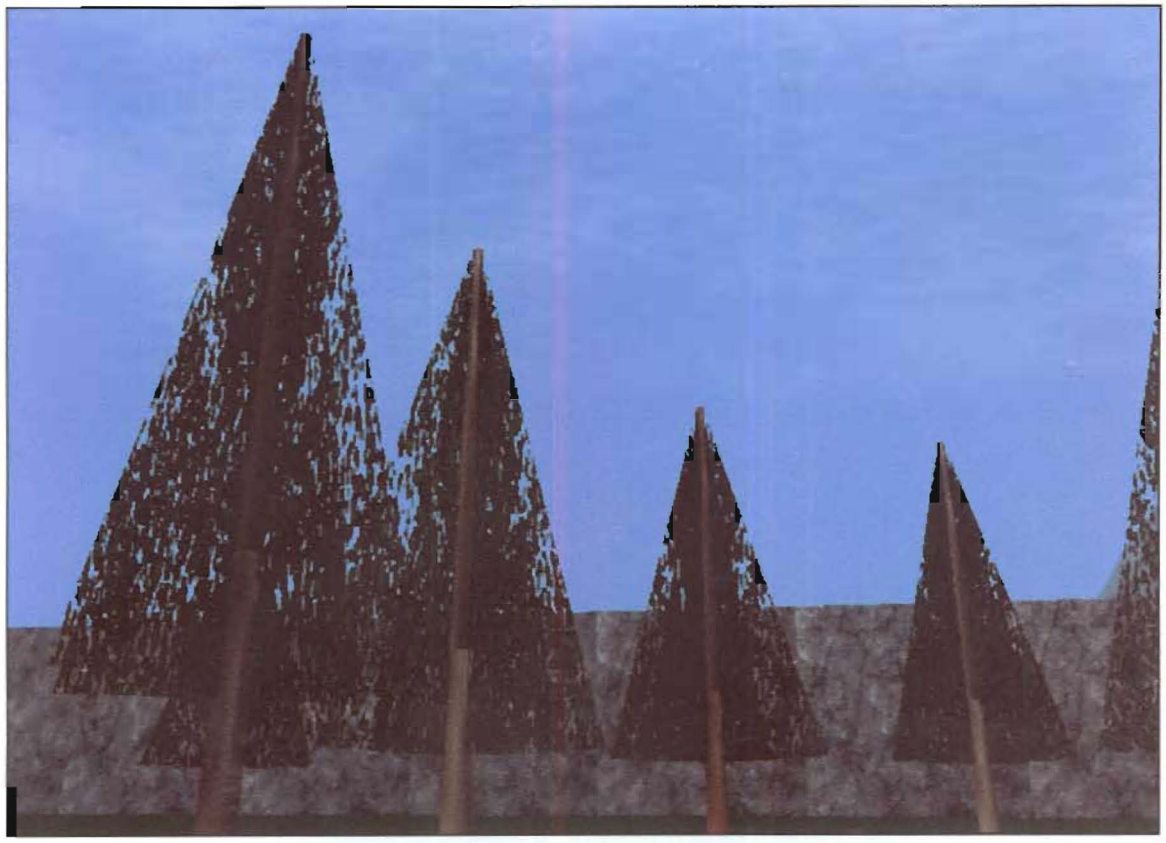


Figure 7.17 Evergreen trees I made with alpha-channeled textures for transparent sections.

7.8 Special Texturing Techniques

Through a fairly simple combination of dynamic texturing and wireframe polygons, I was able to create a great effect in the horizon scene. The ground and walls fade away to reveal a wireframe structure which looks like it has flashes of light travelling up the wires and shooting across the grid. The idea for this scene came from the movie *The 13th Floor*. I was able to recreate it with the following texture of my own creation:

Figure 7.18 Texture used to create light flashes in wireframe mountains (below).

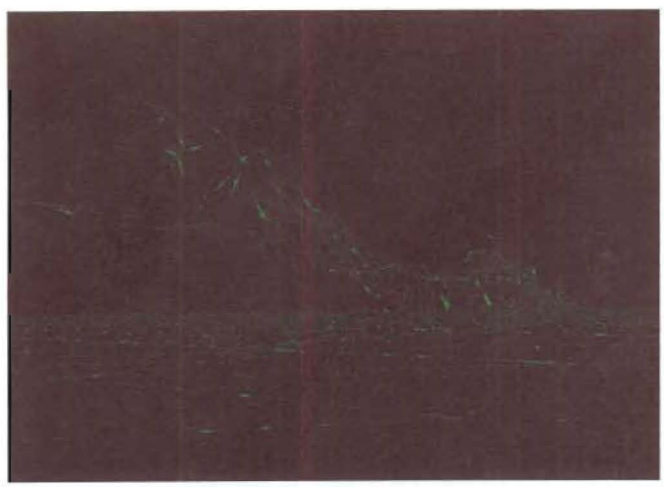


Figure 7.19 Wireframe mountains seem to surge with green energy by dynamically mapping the above texture to the wire structure.

In wireframe mode while texturing is enabled, the wires display whatever portion of texture falls on them at any given moment. As the above texture traverses the grid, the bright lines cross the wires less than the darker areas of the texture. The slight angles of the brighter lines are to reduce these intersections to points, which move up and down wires as the image slides across the structure. I discovered this technique quite by chance when I happened to be viewing the world in wireframe while looking at the starry night sky of the world. The texture of starry sky was flashing across the sky's wire grid as the

brighter stars passed. I quickly adapted this to the green flashing wires --- which turned out to be one of my favorite displays.

7.9 Fog Effects

Once I had some of the scenes laid out, I wanted to be able to do some interesting things with them. One of which was hiding distant portions from the viewer so as to encourage exploration to those areas. The solution was *fog*.

OpenGL and other modern 3D graphics APIs like DirectX [18] and Sony's Emotion Engine [20] for Playstation2 provide a feature referred to as *fog*. Fog is a way to take a set of polygons, and blend their colors with a specified fog color, with a magnitude that is functionally dependent on Z value (the distance from the origin, which is where a scene is always viewed from). It acts essentially the same way as real fog — the further away we are from some object, the more fog there is between us and that object to obscure our view of it. Except in OpenGL we get to control all sorts of parameters that govern the way this fog acts. Fog comes in any color or density that the programmer can imagine, and actually has more uses than just to have fog in a scene. Two other values that control the fog are *min* and *max* values, stating where the fog begins and ends, respectively. Until *min*, there is no fog effect, and past *max*, the fog color takes precedence over everything, fogging out anything beyond.

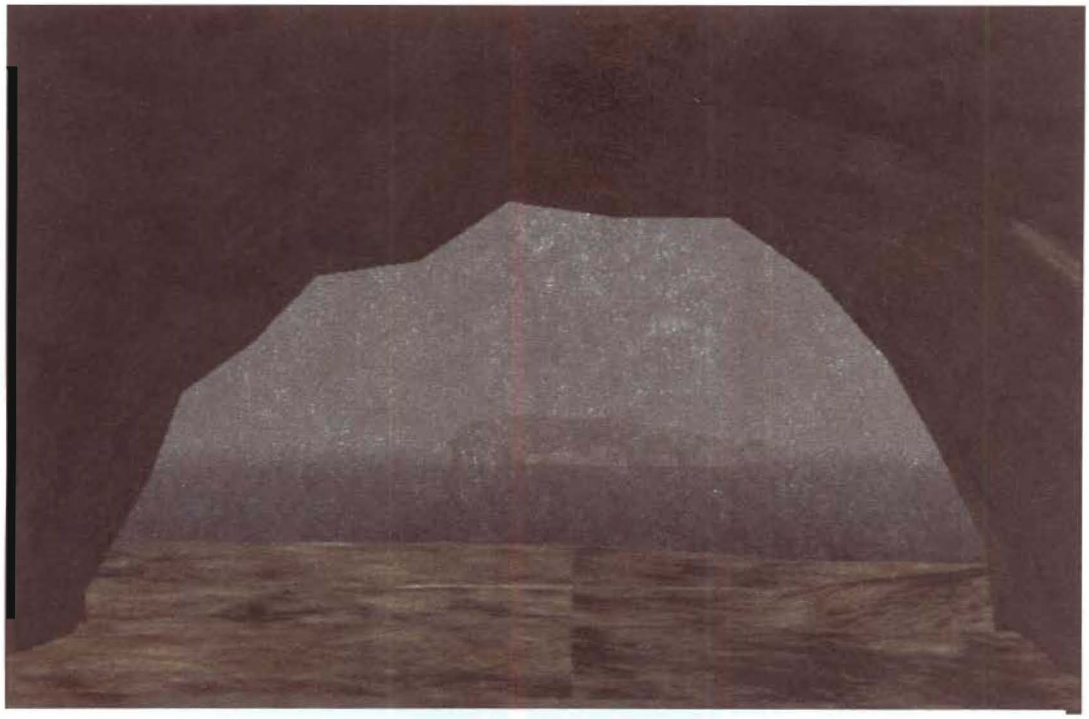


Figure 7.20 Foggy entrance to the underground lake scene in the world.

One settable parameter in OpenGL is a distance beyond which it will not draw anything. I read in the OpenGL SuperBible[25] that one common use for fog is creating a gentle means for objects to fade in and out of a scene at this maximal drawing distance. By setting the fog color to the color of the sky, and the fog end distance to less than the drawing threshold, objects do not get harshly sliced away. However, this is only an issue in very large, open-ended terrain areas. Before there were any mountains in the background of my scenes, the only section large enough for me to use this effect was the valley.

Once I implemented the time-variable sky that changed from blue during the day to black at night, the static-color fog needed modification to account for this. What was happening at night was trees and terrain in the distance would be fog blended to a light blue color, but disconcertingly against a black sky. I needed a way to figure out what the effective color of the sky during this transition was. Since the background was black I

was able to calculate the effective red, green, and blue channels of the sky by multiplying the blue of the front layer by its alpha value each time the scene was updated:

```
effectiveRed   = skyRedChannel   * skyAlpha;
effectiveGreen = skyGreenChannel * skyAlpha;
effectiveBlue  = skyBlueChannel  * skyAlpha;
```

When the alpha value is at its maximum of 1.0 during the day, the effective values make up the blue color of the front layer of sky, and when at a minimum of 0.0 at night, all the color channels scale to 0.0 for black.

Figure 7.21 Fog effects used to simulate light attenuation from viewer position.

Thus, at night, the fog serves as a simulation of light attenuation from the viewer's position. In the underground passage scene (above) I took this to an extreme and used black fog with a short maximum distance, such that the viewer can only see a small percentage of the surrounding scene at any time. It looks as if the user is walking with a weak, non-directional light (like a torch, minus the flickering).

7.10 Per-object fog

There was a point when the mountainous terrain in the scenes ended abruptly. The mountains are all one piece, and they are incomplete on the edge — they have no backside because no viewer could see them from that angle without breaking rules set by my program. (Viewed from afar, their shape somewhat resembles a cutout of an egg crate, with cuts going through the peaks of the crate.) The troughs in the terrain are colored blue for rivers, and they inelegantly "flow"" into thin air at the terrain's edge.

My solution to dealing with this was giving the mountains, and sometimes just the mountains (depending on the scene) a fog value. Fog does not have to be applied equally to an entire scene. If fog is enabled, the current fog values at the drawing of an object are used, allowing for per-object fog, or even per-polygon fog. This became very useful for me, because often I did not want certain objects to fade in the distance as quickly as others, or even at all.

To improve the aesthetics, I applied fog to the terrain (on a scene by scene basis) with a maximum value that is no greater than the minimum distance the viewer can come to an edge of the terrain.

```
glFogf(GL_FOG_START, 100.0f);  // set the start distance of the fog
glFogf(GL_FOG_END, 450.0f);    // 450 is always closer than terrain's edge
glEnable(GL_FOG);                    // enable fog (just for the terrain)
. . .                                // other OpenGL state changes
glPushMatrix();                // store current modelview matrix state
    glTranslatef(0.0,50,0.0);  // position the terrain
    glScalef(1000.0f, 500.0f, 1000.0f);     // scale up the terrain
    terrain->drawList();                    // draw the terrain
glPopMatrix();                 // restore the modelview matrix state
. . .                          // more commands
glDisable(GL_FOG);             // so fog does not effect things drawn after
```

This code is from the gallery scene. Notice that the maximum fog distance, GL_FOG_END, is set to 450 units. The camera is never able to get closer than 450 units

in this scene to any edge of the terrain. The terrain thus fades off into the distance before the edge, and the viewer is not aware of any such sharp or sudden cutoff.

7.11 Multiple Transparencies

After fixing this problem with terrain, I started looking for a method to draw more realistic water than I had been using, which was just a transparent sheet with a water texture applied to it. It would appear more realistic if the water got darker with deepness. Water absorbs more light the deeper it becomes, making it more difficult to see at lower depths. With just transparent polygons, every depth underneath is proportionally as visible as without the water.

The first solution I came up with involved using multiple water layers blended together. Each layer would "absorb" more light, essentially making each level of depth less the color of non-water elements, and more the color of the water itself. The well in the courtyard scene displays this method of attenuation by depth. The top sections of the well are easier to see and distinguish details in than layers further down. While this approach is conceptually straightforward, there are finer points of the implementation that warrant discussion.

First, OpenGL only blends transparent pixels with pixels that have already been drawn. In this way it makes sense to draw anything transparent at the end of the drawing section of code. If I were to draw all the water layers and then draw the surrounding well, the well would not be visible through the water at any level at all.

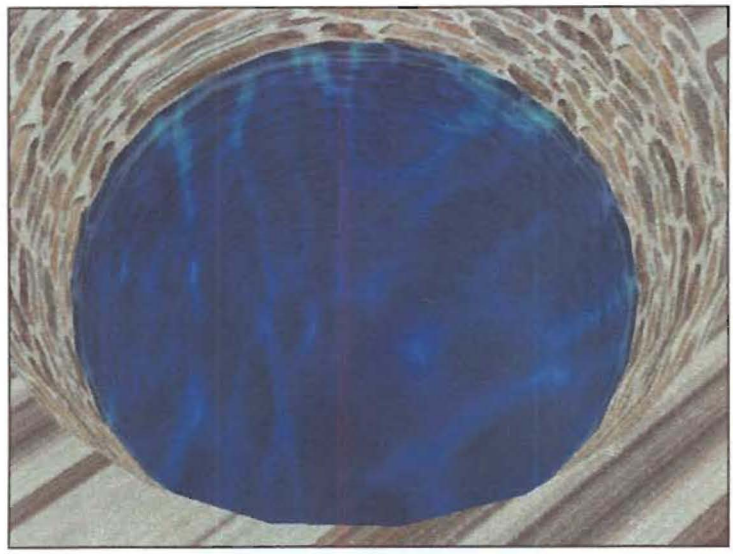


Figure 7.22 A view looking down the well, drawn with multiple transparent layers.

This also implies that the individual water layers must be sorted according to depth, and drawn from the bottom up. Since the viewer will always be situated so as to see the top layer first, then the second, third, et cetera, the layers can have a static drawing order. However, complications can arise if the static assumption can be violated. Consider an example scene where two transparent glasses are sitting on a table. From one angle, the first glass can be seen through the second, and from another, the second glass can be seen through the first. In these two cases, the order of drawing needs to be determined dynamically by distance to the camera — depth-sorting the transparent items and drawing in order from furthest to closest.

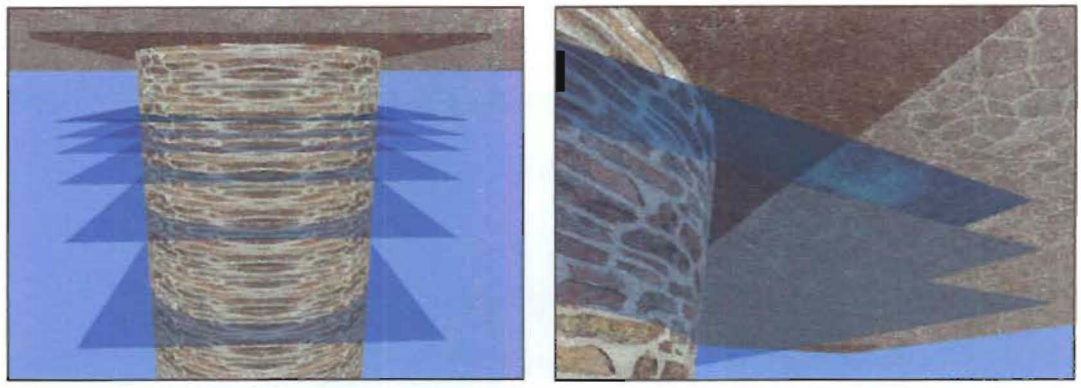


Figure 7.23 Left: the sequence of layers shown spaced further apart towards the bottom. Right: a close-up of the top three layers, showing only the top one needs to have a water texture. The rest are just transparent.

Another finer point to address when drawing the water layers in the well is that, at larger distances apart from neighboring layers, layers can easily be distinguished by the viewer, and the water will not look as realistic as with just the top layer alone. When looking into a lake or a glass of water, we expect to see just one distinct transition where air meets water. Unfortunately, I could not just draw hoards of layers infinitesimally close to each other, because OpenGL has to do calculations for each. I found that after about seven or so transparent layers, frame rate drops to poor levels. To work around this, I drew the first two layers very close together, and each subsequent set further apart. This works well because lower layers are more obscured from higher ones, and they are therefore less noticeable as distinct layers.

7.12 Fog and the Stencil Buffer

Another approach to creating realistic water can be implemented with fog and the stencil buffer. The stencil buffer [25] is a buffer that controls the portions of the screen that are drawn to (hence its name). In other words, if you wanted to draw to the top-left corner of the screen, you could fill the corresponding locations in the stencil buffer with

one value (for "draw") and the rest with another (for "don't draw). Then every subsequent drawing when stenciling is enabled would only output to the top-left corner of the screen. Unfortunately I have not had time to code this method, but I have solidly worked out the logistics.

In the courtyard scene, say I wanted to fade the well water relative to depth in a smoother fashion than with multiple transparent layers. This could be done by applying a fog to everything under the level of the water, with perhaps a deep blue, almost black color. However, unlike normal fog effects, we do not want this fog to apply to the entire visible scene, just to the water in the well. The stencil buffer must be manipulated to single out this part of the scene. Since there is no terrain in the courtyard below water level *except* that covered by the water itself, we would only have to set up a plane equation of `Y = Water Level` and give it to the stencil buffer. Having the proper area singled out for fog is the harder part. Next the fog must be controlled. A depth of two units in the water should be just as obscure if the viewer were standing at the water's edge as if standing far away. The fog start and end attributes must be updated relative to the viewer's distance from the water such that the fog always starts at the distance the viewer is from the water's edge, with some constant difference between fog start and fog end.

This will create the smooth light attenuation we are looking for. A caveat is that this is a better approximation for a body of water with a smaller localized surface area. This is because at the edge of the water, water closer to the top will still be darker further away than at that point. However, since water tends to reflect more light at greater angles of incidence to the surface, we expect to see less in this situation anyway. The effect is worth its flaws, and is less expensive for the graphics processor than blending multiple layers as previously discussed (blending with fog is only done once, not multiple times).

CHAPTER 8: Terrain

8.1 Introduction

In this chapter I will discuss two types of terrain (landforms) that are different because of their use in scenes and the methods by which they were created. An example of distant terrain is the set of mountains and water in the background of most of the outdoor scenes, which I used to fill the emptiness beyond the scene. Close terrain is all other terrain that can be closely viewed, such as walls, ground, and rocks.

8.2 Close Terrain

Rarely does one walk outside and see landforms that have completely flat faces in nature. Good textures help a great deal, but a geometrically flat surface still looks flat. I made it a point to use good textures and dedicate the use of many polygons in some of the more enclosed scenes to create pleasing and realistic terrain. By using higher levels of polygons, surfaces can be given a bumpy appearance rather than a plain, flat one. One feature of "good textures" is high contrast (e.g. — splotches of light and dark) to suggest shadows due to indentations and protrusions.

The path leading up to the castle is a very nice example of high terrain polygon counts, versus the lower polygon count walls in the valley. The path was a prime choice for high polygon count land because the focus of the scene is on the land itself. There are no special non-terrain objects to draw attention. I formed the walls and ground in 3D Studio Max using large sheets of connected rectangles. At first I tried to apply a noise function to create some bumps in the walls, but at high enough levels to perceive a difference from flatness the effects were too angular. I ended up repeatedly selecting semi-random groups of vertices, and pushing and pulling them until the terrain was shaped in a way I thought looked realistic and had a moderately bumpy surface. I used a

texture generation function to map a sandstone texture to the surface of the path. Although automatic texture generation does not always give good results (it works more often when perfection is not necessary), it worked well in this case.

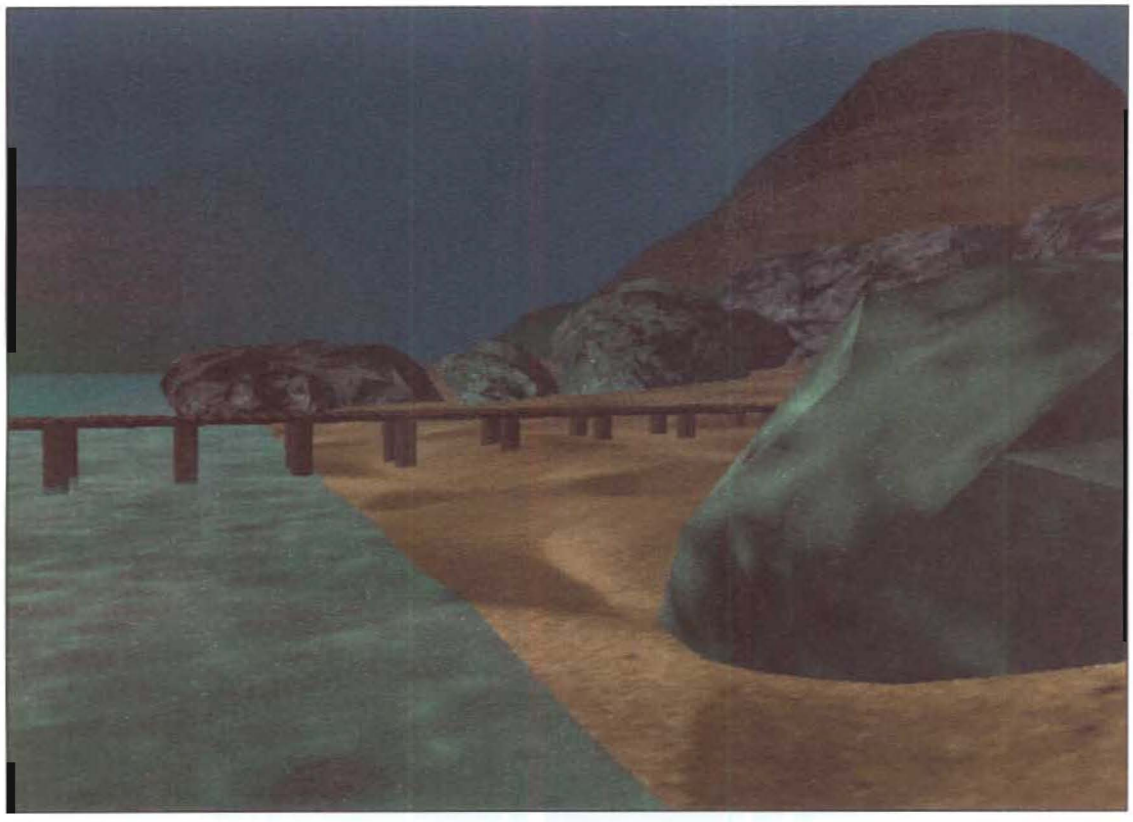


Figure 8.1 Rocks on the beach were made in 3D Studio Max from spheres with noise and stretching functions, and then having spherical texture mapping coordinates applied.

I created the rock in the path and the rocks on the beach (above) as spheres in 3D Studio Max, and then I again applied noise and stretch functions. In Studio Max I also applied multiple texture mapping functions provided, including shrink-wrap, spherical, and cylindrical. The spherical mapping produced better looking results than the other functions. I made the tufts of grass in Studio Max by creating four triangle-shaped blades, and applying a bend modifier to them at different angles. By duplicating them and positioning them around a circle, the tuft took shape.

8.3 Distant Terrain

Before I had included any distant terrain in my world, I had noticed a nagging feeling of emptiness while navigating some of the outdoor scenes. With no features beyond the immediate scene, the viewer feels as if the scene exists in a vast expanse of nothingness. And it did. To alleviate this sense, I filled the void with what I call "distant terrain" because it looms far in the distance.

Using 3D Studio Max, I started to model terrain with a sheet of 7200 connected triangles. This is slightly more than the number of polygons in one of the leafy tree models in the world, and for the amount of each outdoor scene this landscape would fill up, that's an acceptable number. To continue, I pushed and pulled vertices (like pushing and pulling portions of a sheet of silly putty) and sloped sets of triangles until I had what I thought looked like good landscape.

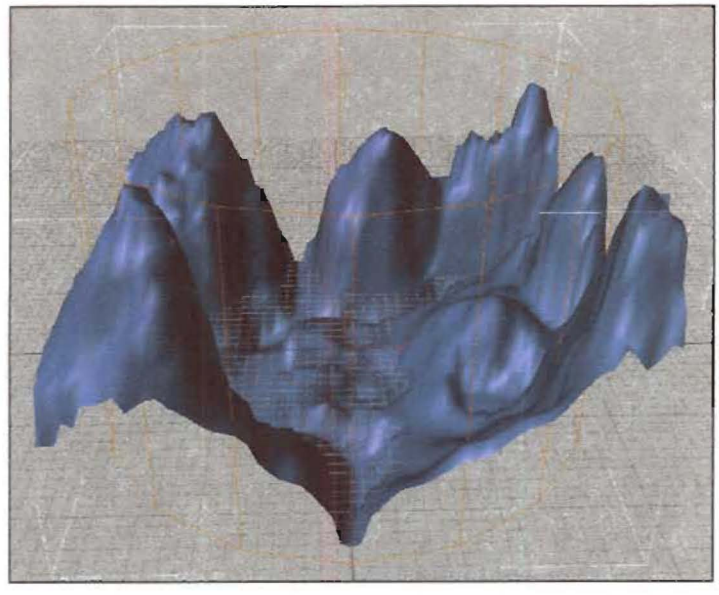


Figure 8.2 The distant terrain landform I modeled in 3D Studio Max. Notice the orange cylinder encompassing the mountains — this is the shape of the texture mapping function used.

The method I used to apply a realistic looking texture to this terrain is interesting. A common means for texturing terrain is to create some function that textures according to a function of height, similar to the way a topological map is colored. For example, the highest points are white for snow. Slightly lower than snow, but above the tree line, is brown. Different levels of greens represent areas around and above sea level for grass and trees, and blues are used for water at the lowest points. Using Paint Shop Pro, I designed my own texture with those levels of color :

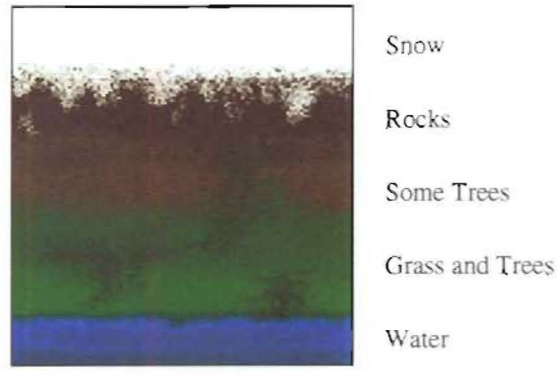


Figure 8.3 The texture I created for the terrain mapping, and the various heights corresponding to the type of land feature each represents.

I applied this texture cylindrically to the terrain using a function built into Studio Max. This essentially takes the texture and wraps it into a tube around the structure, applying the texture to the portions of the model it meets with.

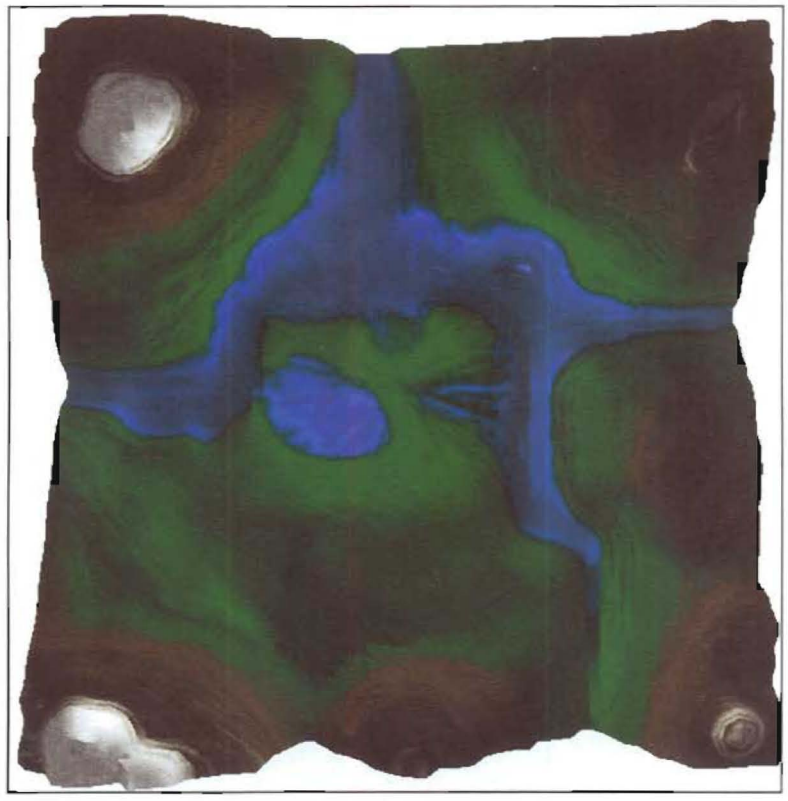
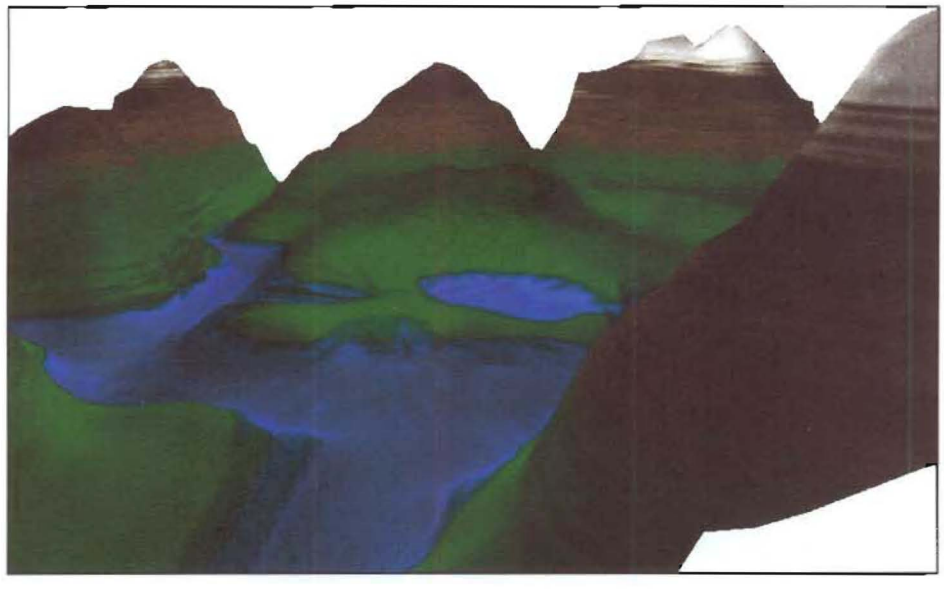


Figure 8.4 Overhead view of the terrain (above) and side view (below).



CHAPTER 9: Particle Engine

Up to a point in the world, I had no real dynamic structures. At most, textures on polygons were moving, but no actual polygons were changing with time. I decided to try implementing some realistic natural effects by manipulating polygons.

Many phenomena in nature consist of a structure that is a building block for a much larger structure. Some examples are rain, snow, water, clouds, dust, and fog. A rainstorm consists of individual drops of rain, as a snowstorm consists of many individual snow flakes. Fog, clouds, and water all are made up of water particles in different densities. A particle engine is an object that controls a set of building blocks to produce a larger, dynamic phenomenon. Although I have already developed clouds and water with dynamic textures, a particle engine would allow me to bring the phenomena out of the plane, and really into 3D. Dynamically textured water could be paired with a particle engine for something like splashes in the water.

I got the basic idea from a tutorial on a website [15], and decided to implement my own. First, a particle for my purposes had to be a polygon or set of polygons that act according to certain rules, most specifically, physical laws of motion. The attributes of my particles are position, velocity, acceleration, color, texture, size, lifetime, decay rate, and whether active or not. Each time a particle is updated, its velocity is incremented by its acceleration due to gravity (and gravity alone, minimizing calculations by singling out the most important factor), and its position is incremented by its velocity. In this way, snow can fall, water can be spouted upwards and fall back to the earth, or haze can just hang in midair.

Say we were to create a particle engine with ten thousand small particles that have a blue, circular texture and a negative acceleration in the vertical direction (acceleration downwards, towards the ground). The engine would then draw each of these ten

thousand blue particles falling at the same rate from the starting point. We want to be able to spread the particles out somewhat, so I defined functions to set initial position and velocity to random values within a given range. Now say we used this to spread the particles out over ten units, and vary initial velocity slightly. We would then see the particles fall over a space of ten units slightly apart from each other. But they keep falling and eventually can't be seen any more.

In order to keep particles flowing, we need to regenerate particles that are no longer of use, like those that fall through the ground. This is why all the particles have a lifetime, starting with some random value between 0.0 and 1.0. On update, all particles have their life decreased by their decay value. Upon reaching 0.0 life, a particle is given full life (1.0) again along with its initial position and velocity. I added an alternative condition for regeneration upon reaching a specified plane. In my valley scene I have snow falling, and once it passes through the plane Y=0, it is a waste of resources to wait for those particles to die, since they cannot be seen. Instead, they immediately regenerate on passing through this plane. To make the transition from alive to dead less visibly abrupt, the life of the particle (conveniently between 0.0 and 1.0) can be used as the alpha channel, so the particle fades away as it decays.

I also created functions to randomize size within bounds, color, and decay. With all this functionality, I have imitated snowfall, water spouts, and small waterfalls fairly convincingly.

Figure 9.1 The Particle Engine used to create a fountain spouting water.

One problem I had was that I was only drawing the particles as circles in one plane, and they became less visible as the viewer moved closer to this plane from the side. My first intention was to use a technique called *billboarding* [5] where a polygon is rotated so that its surface is always facing the viewer. Many games, especially early ones like Castle Wolfenstein, use this technique [4] for smaller items that are close in the scene, or for objects that save needed space by being represented as a series of flat polygons rather than a large set of polygons in three dimensions. Objects with radial symmetry (in real life) are especially suitable for billboarding because they should look approximately the same from any angle, anyway. Applying three matrix operations per particle (one for rotation in each dimension) is not feasible for thousands of particles, as this slows the frame rate down to unacceptable levels.

Another solution, which I eventually chose, is to provide the option to draw a particle as three orthogonal rectangles (see circled portions of wireframe image below). It uses three times as many polygons as the previous particles that were only being drawn in one plane, but these newer particles could be seen well from any angle due to their structure. This approach was also much better than actually drawing the particles as

spheres, which would require at least six times as many polygons per particle to get an approximation of roundness inherent in particles such as water droplets.

Using the particle engine for an entire large body of water is out of the question because the rendering limit of about 10,000 particles (to keep a decent frame rate) would need to be far exceeded. Using large, dynamically texture mapped polygons is still better in this case, but for more detailed simulations, the particle engine is useful.

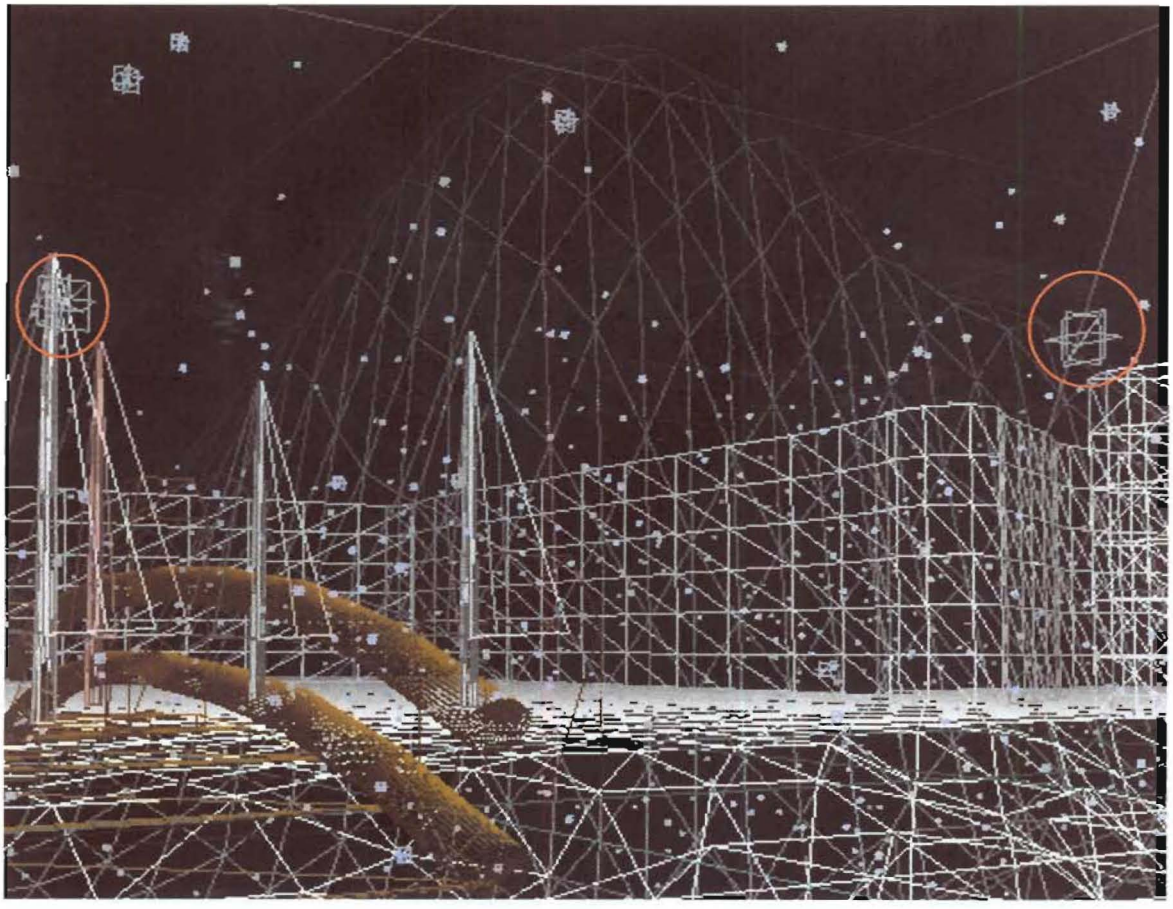


Figure 9.2 Part of a scene drawn in wireframe, with the particle engine simulating snow. Notice the two closer snow particles (circled in red), and how each has polygons on the X-Y, X-Z, and Y-Z planes, for viewing from any angle.

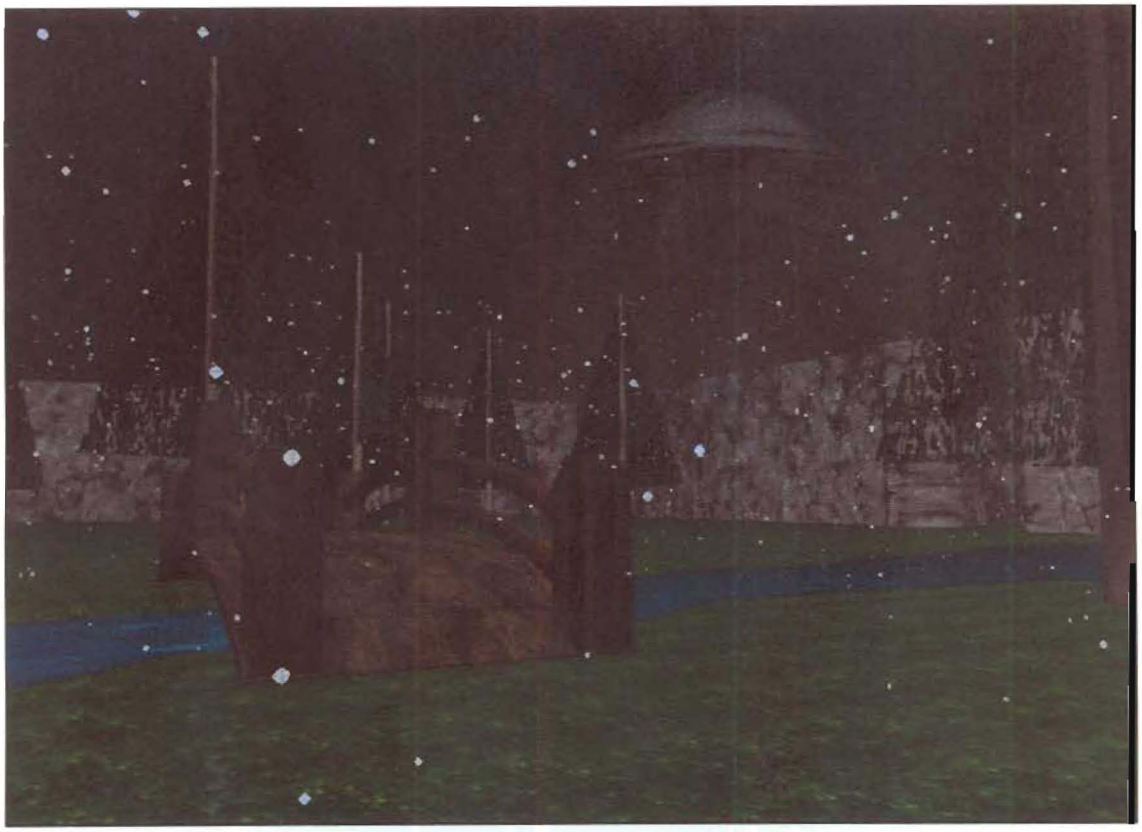


Figure 9.3 The particle engine used to create 10,000 constant snow particles that recycle themselves upon hitting the ground.

CHAPTER 10: Boundlessness: Living up to the word "World"

10.1 Introduction

"Building a world" is quite a phrase to live up to indeed. I could not imaging exploring our earth to the extent that I have seen every portion of it. "World" is used with a connotation of this sort of boundlessness, a quality I attempted to introduce into my world through aesthetics and size rather than any particular programming methods.

The number of explorable scenes present was the most important aspect of design in respect to vastness. The world has a total of ten different areas. The goal was to have at least ten, and am glad I was able to achieve this many, although the more the better. Ten was enough to allow variation within the scenes themselves, creating strong diversity in the world with a beach, mountain paths, a castle and art gallery, a grove of trees, underground sections, bodies of water, and the end of the programmed world.

10.2 Arrangement

The arrangement of scenes plays a very important role in giving the user a feeling of expansiveness. The *worst* possible organization of scenes would be to have the first scene connected to one scene, that to another, all the way to the last scene in a completely linear fashion. The user needs opportunity to make decisions regarding where to explore. By choosing to go one direction and temporarily leaving another untouched, the hinting feeling of vastness begins to grow. Linearly, this is not possible.

The underground passage, mountain path, and valley scene all present these decisions. With a limited number of scenes, they cannot all be connected reasonably this way, and there are dead ends that effectively put a cap on the world. It is conceivable that all the scenes *could* have more than two connections to the other scenes, but that would ruin the mental map one creates when exploring.

10.3 Working with Emptiness

Adding the surrounding terrain of mountains and rivers to the outdoor scenes was one of the best oppositions to confinement. Explorable or not, the terrain took a vast emptiness, and turned it around into a view suggesting a large, terrestrial land. The high-altitude gallery's outdoor portion gives the best outlook onto the land, especially if the viewer jumps up onto the stone wall at the edge of the yard.

I specifically created a beach because oceans carry with them an air of awesome immensity. We are used to looking at an ocean and striving to comprehend where it leads and what may be "on the other side". In my world it hopefully prompts such questioning, as the user does not know that I did not program something to be on a far off shore somewhere.

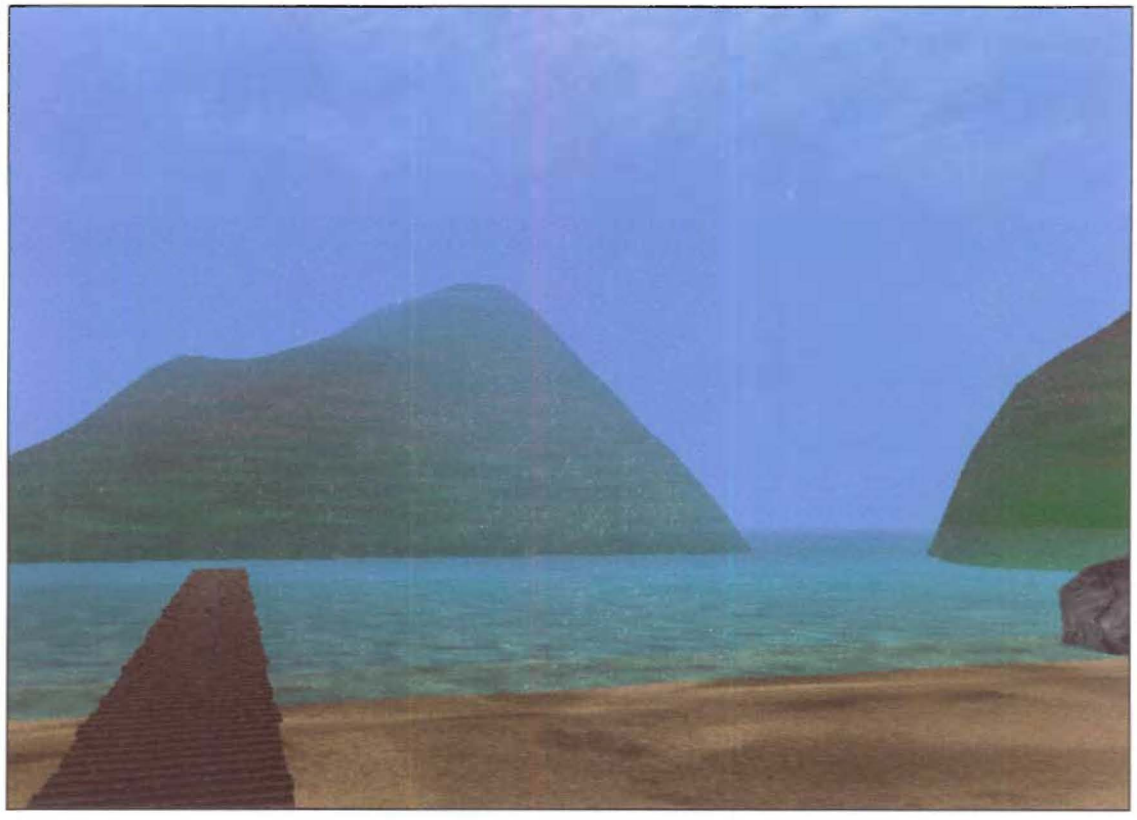


Figure 10.1 A view off into the distance on the beach in the world.

CHAPTER 11: Navigation

Once there is a scene in place, with some ground and rocks or trees, there is the problem of actually being able to move around it to look at everything. It is not a feature automatic or built in to OpenGL, so I needed to figure out a way to draw things to the screen in positions such that it looked like the viewer were able to move.

One of the first OpenGL programs I wrote was very simple, displaying a red wireframe cube on the screen. I added interactions that would change the position of the cube based on key input, to give the appearance of moving towards or away from the cube. I did this by including an updateable variable in the Z coordinate of the cube's position:

```
glTranslatef(0.0, 0.0, -10 + variable);  // get to the correct spot
glutWireCube(1.0); // draw a cube there of side length 1
```

The third field in the *translate* function tells OpenGL to move back along the Z axis 10 units plus some variable amount that depends on user input. The cube is then drawn there.

After this simple experiment, what I wanted was to draw multiple objects and manipulate them such that it looked like the user was actually walking around. Now, in OpenGL the scene is always drawn from the origin. So to get this movement effect I could just add some variable term to each dimension of the object's position similar to the way I handled the cube. But then when the user walks around, the world would only be viewable from one angle, straight down the negative Z axis. To account for this in the same manner I would have added variables to a rotation term for each object as well. Unfortunately, this approach quickly becomes tedious and messy with variables all over the place.

To take care of the problem in a cleaner fashion, I used a common approach where the entire scene is rotated and then translated at once. My program keeps track of the amount of distance from the origin and rotation past zero degrees to represent the view based on input. Before drawing each scene, the rotations are applied, once for each dimension, then the translation is applied. All the values that are used are the opposite sign of their stored values — to move the "camera" ahead 5 units, we are actually instead moving the whole scene *back* 5 units. And by keeping the effects of the translations and rotations on the current transformation matrix while everything in the scene is drawn, it will look as if the "camera" has moved, when it actually is the entire scene.

At this point I was then able to navigate scenes and look at my objects from any point or angle (excluding roll – analogous to a side-head-tilting movement). As exciting as this may have been, the experience of moving around was like navigating a completely empty space — I could walk through walls and I could move backwards off the edge of the scene and watch it grow smaller and fade away. This brings up the much more complicated topics of collision detection and collision reaction, which I discuss in the next chapters.

CHAPTER 12: Collision Detection

12.1 Introduction

Before collision detection was implemented, I had my scenes all laid out and they looked great — but there were some big problems. When walking around, it was possible to move right through walls and objects. If the viewer happened to try walking down a hill, the hill would simply drop away underneath, requiring an explicit set of key commands for floating up and down. I needed some way to know exactly where the boundaries were in each scene, so that the viewer could react to them automatically.

Collision detection is the ability to be notified when two objects paths intersect in space-time. It is a means to connect the viewer (and other objects) to the world in more than the superficial sense of being restricted to looking at the scenery. Collision detection has an application wherever you want to know when two objects hit one another. In a car racing game it would be used to see when cars crash into one another and to keep them from falling through the ground or driving through walls. In a soccer game, players would not be able to kick the ball without collision detection.

Collision detection is no trivial matter — at least one month of my allotted time overall was spent researching and implementing collision detection. Entire fields of research are dedicated to this topic alone. As a matter of fact, when I was searching for information on collision detection, much of it was too complicated to get involved with. I also attempted to use two different third party collision detection packages, but due to complexity and lack of time to spend understanding other people's interfaces, I did not use either. Instead, I decided to save many hours and much frustration, and at the same time learn a great deal, by implementing my own version.

12.2 Ray-Casting

One of the foundations of collision detection within three dimensional graphics is a mathematical concept known as *ray-casting*. Ray-casting has the ability to account for the world's three spatial dimensions. An online tutorial at flipcode.com [13] helped me get started dealing with ray-casting. The concept is fairly easy to understand: you take an initial point in 3-space of a moving object and a destination point in the same space at some later time, then find the mathematical equation of the ray formed by moving from one to the other. Now, iterate through all collision polygons and test to see if the ray intersects with the planes that these polygons lie on. Each time the ray intersects with one of these planes, check to see if the point of intersection with the plane is actually within the bounds of that polygon. If so, then there was a collision.

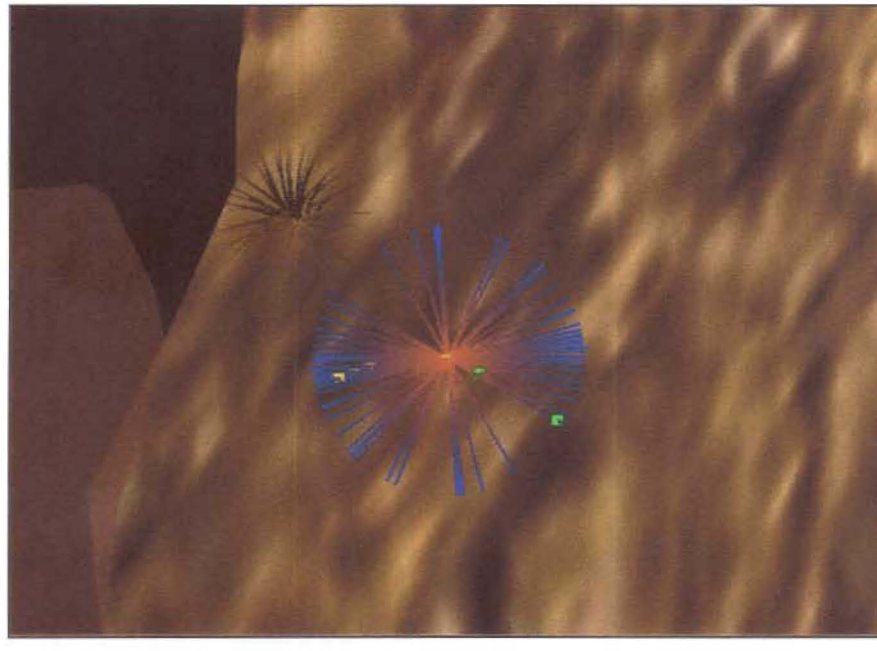


Figure 12.1 This picture shows ray-casting from the center of the cylindrical boundary surrounding the camera to its edges in the directions of planes that collision polygons lie on. Green and yellow points show where intersections with these planes occur. None of these intersections are actually with polygons, just their planes.

This required the implementation of some basic geometric structures. I wrote the representation for a 3D Point, which is equivalent to a vector and has the same fundamental operations. Then I wrote a 3D Plane representation, which uses a 3D Point for its normal vector, and I wrote a 3D Polygon representation that uses 3D Points as its vertices. Some sample code I use in my project to determine an intersection point using ray-casting is below.

```
Point3D Plane3D::RayIntersect(Point3D start, Point3D finish){
   float t;  // point in the ray where the intersection  occurs
   Point3D ray(finish.x, finish.y, finish.z); // duplicate the finish point
   ray.subtract(&start);
   float num   = Point3D::DotProduct(normal, &start) + distance;
   float denom = Point3D::DotProduct(normal, &ray);
   if(denom == 0.0) {
      t = 1.0f;
   }else{
      t = num/denom;
   }
   ray.scale(-t);      // scale ray by -t, because of backwards coord. system
   ray.add(&start);    // add the starting point to it,
   return ray; // holds point of intersection  now
}
```

Just finding out if one point intersects with a polygon is only the beginning to collision detection. We don't want to represent the viewer as a point — for if we did the viewer would fall to the surface of the ground and feel like the size of a pea. Also, the viewer would be able to get within the minimum drawing distance of objects, which is 0.5 units. Any pixels within these bounds do not get drawn. If the camera is allowed to get within 0.5 units of a wall, that portion of the wall will disappear to reveal anything that may or may not be behind. For this reason, the camera must be associated with a collision *volume*.

12.3 Collision Cylinders

The collision geometry I use most often in the world next to polygons, is the cylinder. I chose to use cylinders because they are quite symmetrical, easy to represent, and can be used to give curved surfaces accurate collision behavior. In order to avoid excessive and time-consuming calculation, I restricted cylinder space to all upright cylinders (whose caps remain in the horizontal plane).

The three polygon-cylinder cases supported by my collision manager (so far) are when:

1. The center of a cylinder moves through a polygon.
2. A vertical cylinder intersects with a vertical polygon, with better approximation for polygons with closer to vertical edges.
3. A vertical cylinder intersects with a horizontal polygon, with greater approximation as angle of elevation increases for non-horizontal polygons.

The three cases are checked in the order presented. If any test finds there is a collision, the other tests do not happen. Case (1) simply tests to see if the center of the given cylinder passes through the polygon by casting a ray from the initial position of the cylinder's center to its final position.

If case (1) is not true, case (2) is checked. First, the vertical region of space where the cylinder and polygon overlap is found. If there is none, there is no intersection. Otherwise, the test continues by taking the midpoint of the vertical range of overlap, and casting a ray from that vertical point in the horizontal center of the cylinder along the polygon's positive and negative normal (could be on either side of the polygon). If either of these rays intersect the polygon, there is a collision. Also, to account for the case of a cylinder being outside the vertical boundaries of the polygon, the closest point of the polygon in the horizontal (X-Z) plane is found, and a ray is cast from the center of the cylinder through that polygon's X-Z point and the cylinder's Y point.

Case (3) is subsequently checked if cases (1) and (2) were negative. It takes the point on the edge of the top cap of the cylinder in the cylinder's direction of motion, and casts a ray from this point to the same point on the cylinder's bottom cap. An intersection is a collision.

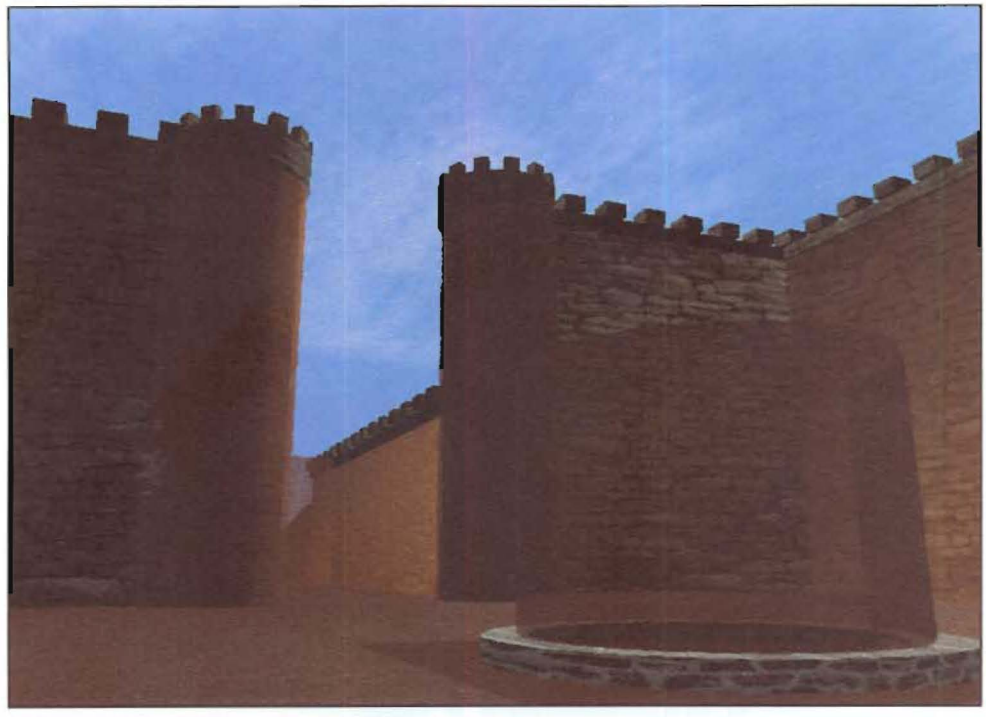


Figure 12.2 This screenshot shows collision boundaries in a translucent red color. Notice the flat polygons on the floor and walls, and the cylindrical boundaries around the turrets and well. Notice with the well that something need not be drawn for a boundary to exist; in the well's case, it is the absence of a floor that requires a boundary, so the viewer cannot fall down the well.

The three cases combined with the orientation restrictions of the collision objects provide an effective means to detect cylinder-polygon collisions. Keep in mind, that is only one set of tests, that took over 100 lines of *compact*, well-thought-out *C++* code to implement. The collision manager provides other sets of tests including polygon-sphere, cylinder-cylinder, et cetera. One problem I had was that cylinders colliding with a point where two polygons met in a corner sometimes got the cylinder stuck along one of the polygons if the point were an angle of 270 degrees (where the cylinder is within that

angle). Since vertical cylinder-cylinder collisions do not have any flaws, I began sealing such points with cylinders.

12.4 Collision Optimization

Some forms of optimization are necessary for collision detection, because it is not feasible to check for a collision between every existent polygon. This would be n(n-1)/2 checks (where n is the total number of polygons), which takes $0(n^2)$ time to complete. For one of my normal scenes, this would be well over a billion checks. This must all be done within a miniscule fraction of a second to be useful, and I would not want my program to have that dependency. Forms of optimization exist that reduce the set of total polygons for the collision manager to handle, and that reduce the number of polygons within that set that have to be checked against one another.

It is logical to break up the collision objects that move from the objects that do not, because objects that do not move will not have a chance to collide with each other. My collision manager has a separate container for each of these objects. The dynamic objects are checked against one another, then checked against all the static objects for collision. This partition saves many calculations since many more static elements (like walls and floors) will generally exist than dynamic ones (like a camera or a bird).

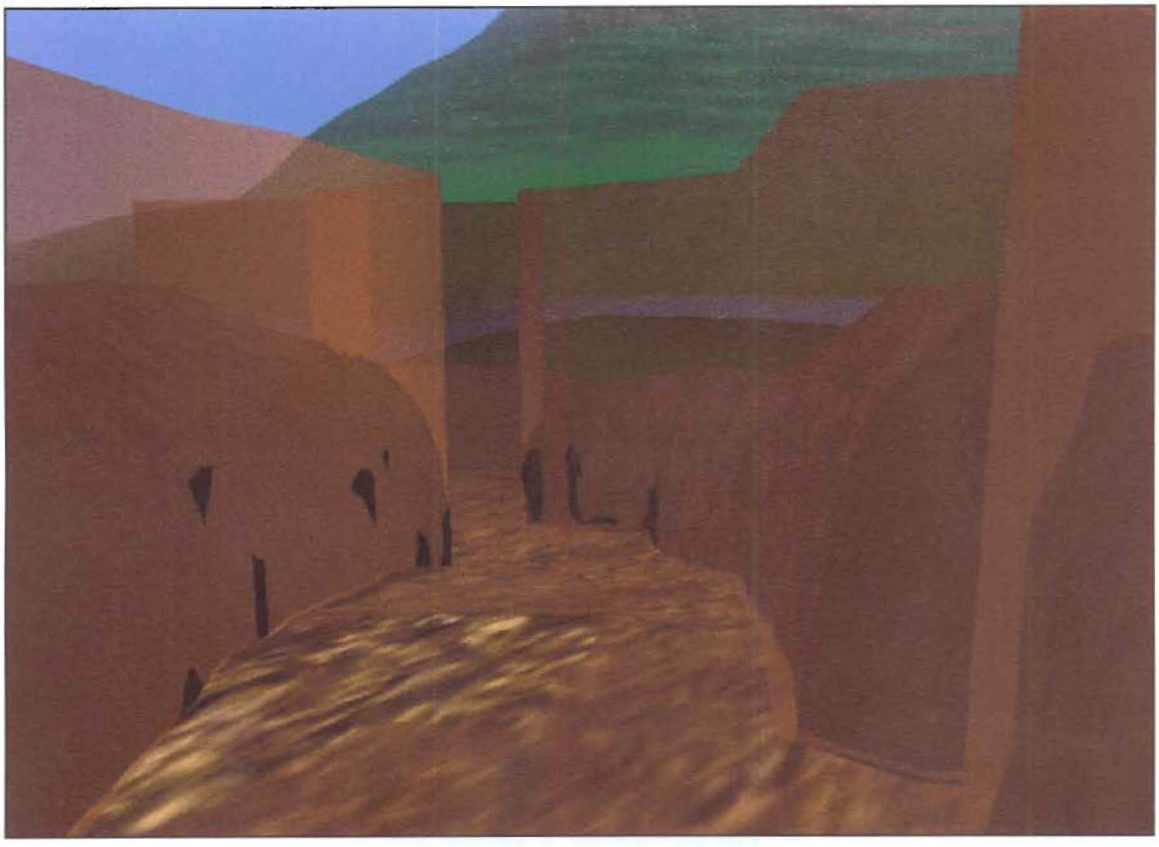


Figure 12.3 All the red collision polygons approximating the shape of the wall to further reduce the set of collision polygons, as opposed to using every polygon that makes up the wall as a boundary.

Once I had all the constructs that could collide with one another (cylinders, polygons, et cetera), I was able to use them to approximate all the surfaces the camera might collide with in every scene (seen above). By approximating these surfaces as opposed to using those surfaces, I was reducing the set of collision polygons. There are two reasons why it would not have been a good idea for me to filter all scene polygons through a function to create collision polygons. The first is that if a scene has 45,000 *static* polygons, and just the camera, it would take hundreds of thousands of calls to functions that make many calculations to test all possibilities. Adding more dynamic polygons makes this number of calculations much larger. I was generally able to make

these approximations with twenty to seventy polygons and cylinders — a much better number than tens of thousands.

I would like to briefly discuss another form of collision optimization that I did not have the need to use nor the time to implement, although it is very interesting and quite elegant. It uses a construct called an *octree* [10], which is a tree where each node has eight children. They are used by taking the three-dimensional space in a scene, as a cube, and breaking it up into eight smaller cubes each of equal dimensions. Cubes are recursively broken down until some stopping criterion is met, such as a minimum size or polygon content. All collision objects are added to the leaf of the tree that they fall into (possibly more than one). Since collisions are spatially dependent and only objects in the same leaf are close, we know that only objects falling within the same leaf could possibly collide with one another. Thus the working set is reduced greatly, and quickly, in a graceful way. I think implementing octrees would be a great extension to the project, as they are useful in other methods of optimization as well.

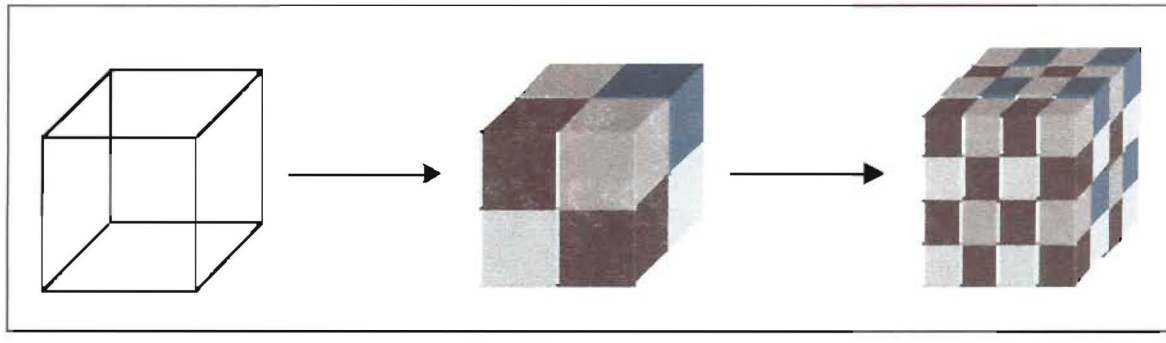


Figure 12.4 Illustration of an octree partition

CHAPTER 13: Collision Reaction

13.1 Introduction

Being able to detect a collision is great, but not enough in itself. Something actually needs to be done when two things hit each other. For the camera specifically, we want to at least stop it from moving through the wall. By keeping a starting position and a destination position as part of the camera's information for each update, if the camera collides between these two points, it is sufficient to set its destination back to its starting point. In that way, if the camera collides with any wall, it is stopped in its tracks.

There are two problems with just stopping the camera's motion completely. First, it is less frustrating when moving around if the camera were actually to slide along walls, rather than being stopped and having to back up or turn around to continue moving. Second, if there is a gravity that keeps the camera on the ground, it would never be able to move at all because it would be constantly colliding with the ground. I had this problem at first, so I had to set gravity to zero while I coped with the situation.

After drawing many pictures and diagrams, I figured out that to slide an object along a collision barrier, two crucial pieces of information are needed. These pieces are the path of the object's attempted travel, and the normal of the collision surface at the point of collision. The path is known, since it was used to detect a collision. The normals are an inherent piece of polygons, and those polygons only need be queried. Sphere normals can be found by taking the ray from the center of the sphere to the point of collision and normalizing it (reducing its length to 1). Lastly, (vertical) cylinders also have a function to return a normal, which is the ray from its center to the point of collision, *with the same vertical coordinate*, normalized.

By taking the dot product of the path and the normal, a distance d is yielded that is the length of the path along the normal. This essentially collapses the path onto the

normal, but the value *d* has no direction (it is a scalar value). Now, the amount to remove is obtained by scaling the normal by *d* (this result has a direction). Then the produced vector can be subtracted from the destination of the moving object before collision reaction to attain the new, altered destination. This is illustrated below.

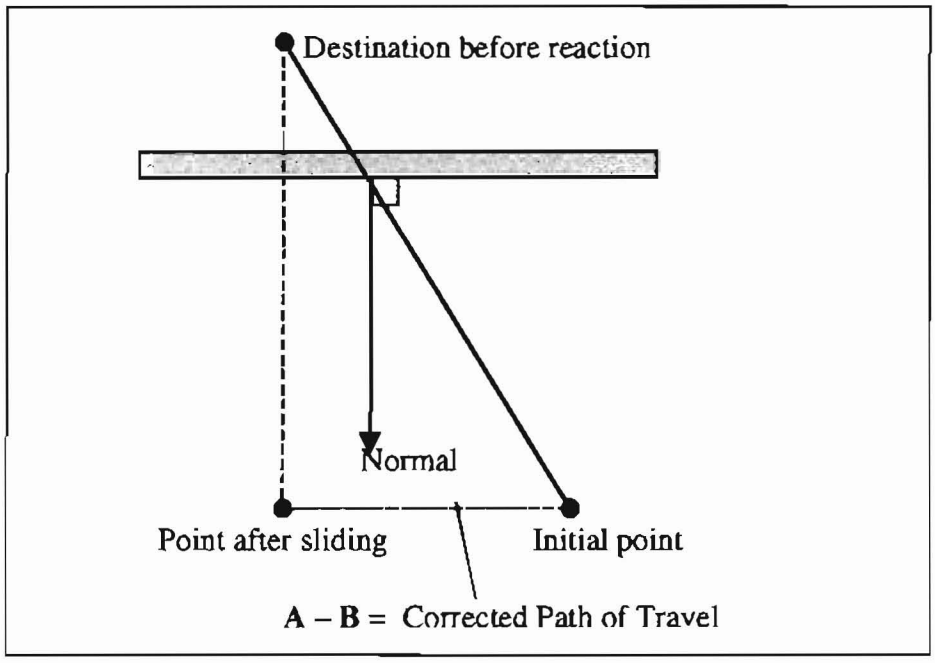


Figure 13.1 The mechanism for sliding along a collision boundary.

Notice that even though there are right triangles and similar triangles involved in the diagrams, trigonometry alone is not enough to solve the problem. Vectors need be used because more than just magnitudes, we are dealing with direction in two (or three) dimensions. Also, although in the diagram it appears to be taking place far from the edge of the surface, this is generally not the case. In real time, the distances involved in traveling from one point to the next are minute since they span all of a few hundredths of a second. The distance is therefore not noticeable. Be aware, however, that this sliding technique is used uniquely in navigation. If we were to try simulating a ball being

thrown at the same surface, we would want to instead reverse its velocity and flip over the surface's normal for an elastic collision.

13.2 Multiple Subsequent Collisions

It is possible that a situation would occur, most likely in a corner, where a collision with one polygon would slide the viewer past the boundary of the other polygon. Someone would easily be able to then "slide" out of the boundaries of a scene, and literally fall off the edge of the world. Gravity would relentlessly pull the viewer downward into a never ending pit of emptiness — a situation that I did not want to let happen.

To prevent escaping boundaries via sliding, I added additional collision handling as a follow up to any initial collision. The collision manager executes a loop for each moving object that breaks only when that object's path does not collide with anything at all. There is also an alternate condition which breaks upon some maximum number of collisions so as to avoid an infinite loop, although this has not been necessary.

13.3 Ground versus Walls

At this point the viewer had pleasing reactions when moving along walls, and when gravity was enabled, the viewer was able to walk along the ground freely. A problem I did not foresee was related to sloping ground. I built a test ramp in the castle courtyard scene to test the completeness of my algorithms, which, of course, were proven incomplete by the ramp.

I was pleased to see that I could navigate the viewer up the ramp and fall off the end to the ground, without going through the ground. The problem was that standing still on the ramp revealed that the surface appeared to have no friction. The force due to gravity was constantly trying to pull the camera slightly below the surface, and the

surface was sliding it back along the normal. The net vector pointed down the ramp with a magnitude proportional to the slope of the surface.

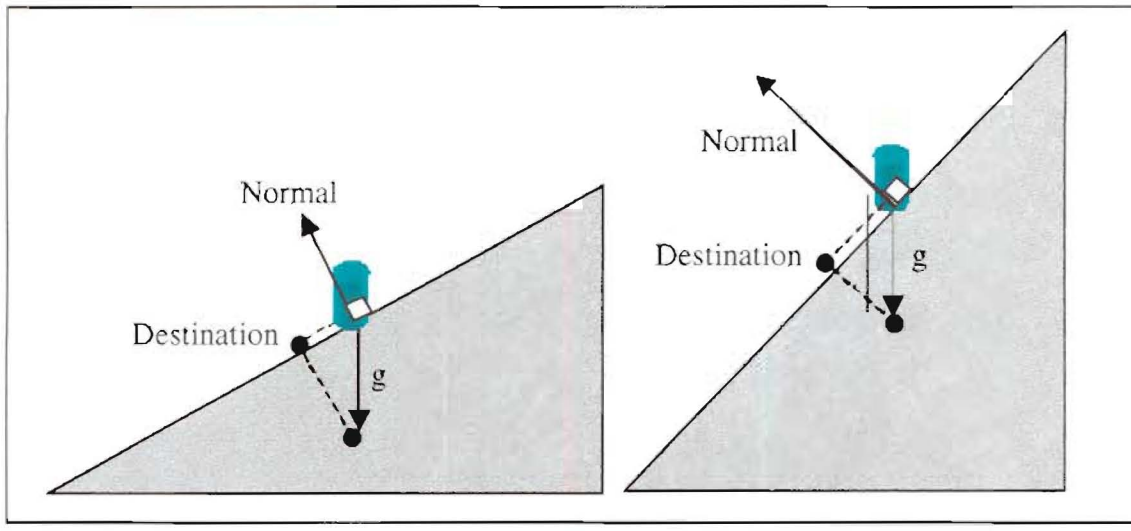


Figure 13.2 The viewer sliding down two differently sloped surfaces due to gravity.

It was clear that I either had to add some friction to the surface, or more plausibly, make some sort of technical distinction between walls and ground, which is the option I opted for. From my Polygon representation, I derived the Ground representation and the Wall representation. The camera continued to handle collisions with walls without modifications.

I had to change the camera's reaction to ground collisions so that it would still be able to move, act properly with gravity, and not slide down sloped faces. So rather than sliding along the normal, the camera's collision cylinder casts a ray from the center of its base along the vector of gravity to find a point of intersection. The component of this point along the vector of gravity is taken, incremented by a one-hundred-thousandth of a percent (in the direction opposing gravity), and replaces the camera's component with this new one. Thus the camera is placed ever so slightly above the ground, retaining its component of motion along the ground to allow for sliding.

CHAPTER 14: Audile Realism

14.1 Introduction

Originally I had not considered putting sound in my project, and had not even mentioned it in my proposal. But sound, even a random distant bird call, breaks the monotony of silence and causes the viewer to focus on the world with two senses instead of one. Requiring attention of another sense draws the user into a more unified world that is all the more realistic. As I realized this I became inclined to include audio as an integral feature of the world.

14.2 False Starts

There was much trouble in the process of including audio. I examined two suitable options for three dimensional sound APIs: Microsoft DirectSound, and OpenAL [16]. OpenAL (for Open Audio Language) is OpenGL's equivalent in audio — multi-platform, state-based, and generally similar in dialect. OpenAL, however, is at a premature stage, copyrighted in 2000. DirectSound is part of Microsoft's larger suite of programming libraries, DirectX. I chose to work with DirectSound since it has, ostensibly, a sturdier foothold being on version 7.0 as opposed to OpenAL's 1.0 version.

This chimera of mine lasted only a few days when I became increasingly frustrated with the sheer amount of code necessary to load just one sound and play it. The tutorial on Microsoft's website had about seven pages of code with frequent abstruse and unexplained code statements. Upon attempting to compile the files, I received errors when theoretically I should not have. All this was just for a dimensionless sound, the ordeal doubtlessly more difficult for three-dimensional sound. I pondered what I had learned, then promptly threw in the towel and moved on to OpenAL.

A sample project the designers of OpenAL sent me was relatively easy to understand. Every piece of code had a clear meaning, and there was far less set up to get OpenAL functioning. It took me quite a while to install the correct files in order to use OpenAL, and I still do not have the optimal setup. I discuss this further and relate it to a more general problem in a later chapter *Bugs, Obstacles, and Pitfalls*.

14.3 Working Sound

Once I was able to get sounds playing, my first intention was to synchronize sounds with individual footsteps (the low points in the up-down "head" movement of the camera). Using a sound editor I broke up a clip of a person walking through leaves into five separate footsteps. One of the five is randomly chosen and played each step at the foot of the viewer, providing the viewer with a more human feeling interface, as opposed to a rigid camera gliding silently through the air. If I had time, I would have liked to take this idea further. I wanted to give each section of ground a "material" property that describes what type of ground it is, such as snow, grass, water, stone, or gravel. Over the continuous collision the camera's bounding cylinder has with the grounds, it could record the current material underfoot. Then the stepping sound could be changed depending on the type of material. *Tony Hawk's Pro Skater 2* is a popular skateboarding game in 3D that gains much from utilizing such methods. The sound the skateboard produces rolling around changes masterfully when skating from one material to another. Blacktop, wooden ramps, rails, and even corrugated metal have sounds that match exquisitely, and were probably taken from real skating clips.

Next, I decided to add sounds that insinuated the presence of other life in the world. I chose crows because they are birds that are common most places, and people are usually used to hearing them. I divided up some crow calls I had found into four different ones, and placed the four in random distant locations in the outdoor scenes. By

having distant crow sounds, I hoped users would get the feeling that since the crows could not be seen, but could in fact be heard, that there were actually crows *somewhere else* in the world, subliminally conveying that somewhere else actually exists and the world is larger than the explorable sections. There is also the possibility that the user would hear the crow sounds and not register it as something special due to their commonality in our world, and thus feel somewhat more at ease. Note that the speed of sounds is in no way affected by the element of time in the world. A crow will sound just the same if days pass in seconds or hours in our world. The *frequency* at which individual sounds occur may change if it is dependent on time (for example, the occurrence of the sound of a plane flying overhead every virtual-world-hour will change with the rate of virtual-world-time). Its pitch, wavelength, and amplitude will not, as they are only dependent on position and velocity relative to the listener.

The other worldly sound present in outdoor scenes is wind. A constant wind-blowing that reminds the listener of the elements, and fits in with the moving clouds above. There are functions available in OpenAL to set the minimum and maximum gain of a sounds, however not in the version I was using. The min and max gain could be set to the same value to keep a sound as completely ambient, with the same volume at any location in the scene. This is the way I wanted to handle the wind, since there is no effective point source for wind. Since these functions were out of grasp, I had to reset the position of the wind to be the same distance relative to the viewer at all times. Problems occurred when I set the location to be the exact location of the listener, however. The screen got all jumpy and no sounds would play, as if some computational task were eating all the program's resources. Just accepting this as something that causes problems, I avoided positioning sources at the listener from then on.

The last sound found in the world is a pervading static. Most of the crows and footsteps have a short burst of static each time OpenAL plays one of them. I was unable

to chop these parts off in an editor, and became unsure whether this static was a byproduct of the audio editing program itself or OpenAL. Either way, it was noticeable, and mildly annoying, since I am not used to hearing my shoes spout white noise with each step. My fix was to add a continuous static at low enough volume that the listener is unaware, yet still masks the aggravating static accompanying regular sounds.

CHAPTER 15: Real-Time Engine Manipulation

A "3D Engine" is basically a set of functions and procedures that are used in a controlled way to display three dimensional graphics. It almost always connotes real-time viewing. I had to have the ability to toggle features of my 3D engine while moving around the world. Most commonly, I needed to turn collision detection off to get to forbidden parts of a scene for debugging purposes (in which case I also had to turn gravity off prior, so I would not plummet through the ground). I have left the various key toggles in place so users can actually view the world under different circumstances.

Fog can be toggled. In the valley scene, and any other scene with the mountains in view this makes a significant difference.

Lighting can be toggled. The differences are incredible in some areas, especially the grove. The leafy trees and the tomb have much more definition when their polygons are shaded with respect to face normals. The grass also gains much definition from lighting.

Textures may be enabled and disabled. Textures are altered by the color of the polygon faces they are applied to. Rarely it was useful to color the polygons anything other than bright white. Disabling textures shows the flat polygons and their colors.

A feature called *depth testing* may be toggled. Depth testing utilizes the Z-buffer, comparing Z values of the current item being drawn with values in the same location of the buffer, allowing pixels to be drawn to the screen if they are spatially in front of all other pixels. Turning off depth testing draws polygons to the screen in the order they are given to OpenGL. If the viewer is inside a solid black cube, but an orange cylinder is drawn after the cube with no depth testing, the cube would be visible.

Gravity can be toggled, allowing the viewer to hover any height above the ground.

Collision detection can be toggled for travel through walls and floors. Without collision, there is no way to detect when the camera hits a scene transition point. If gravity is still on, the camera will fall through the ground. The collision boundaries can also be drawn as light-red, semi-transparent polygons. The scene, independent of the collision boundaries by be toggled as well.

Users can view camera orientation, position, and current frame rate in the title bar of the window if desired.

Figure 15.1 Below are all the keys used and their corresponding affect when the world is running.

"8" — Move forward (on number pad)

"5" — Move Backward (on number pad)

"4" — Turn Left (on number pad)

"6" — Turn Right (on number pad)

"7" — Sidestep Left (on number pad)

"9" — Sidestep Right (on number pad)

"A" — Look Down

"Z" — Look Up

Up Arrow — Float up when gravity is disabled

Down Arrow — Float down when gravity is disabled

Hold "Tab" — Doubles speed of movement

Hold "Caps Lock" — Extremely quick movement

"P" — Toggle position display

"O" — (O as in olive) Toggle orientation display

"F" — Toggle frames per second and day counter display

"M" — Toggle "Mouselook" so the mouse can be used to change orientation vertically and horizontally.

"S" — Toggle drawing the scene (use with the boundary toggle to see only boundaries)

"B" — Toggle view of collision boundaries

"C" — Toggle Collision Detection

"G" — Toggle Gravity

"Space Bar" — Jump

"L" — Toggle Lighting

"Y" — Toggle Fog Effects

"W" — Toggle Wireframe Mode

"D" — Toggle Depth Testing

"T" — Toggle Textures

"X" — Toggle Cross-Hairs (an orange square that stays in the center of the screen)

"E" — Toggle Particle Engine (use in valley to toggle snow, and in the underground lake to toggle the water spout).

"RETURN" — This returns everything to normal. Use when you fall off the edge of the scene, get stuck, or can't see anything.

CHAPTER 16: Bugs, Obstacles, and Pitfalls

16.1 Installations and Incompatibilities

In my ideal world, computers would be completely set up for me. Every necessary piece of software would be installed, and one-hundred percent bug-free. This includes all the programming tools and libraries — they would all come ready to use, newest version and compiler independent, with clear descriptions of the constructs available and how they are to be used. Then I could sit down and do my thing; I could design and program. There would be no time wasted on false starts, no conflicting versions or missing pieces. And, ideally, I would make no errors along the way.

Unfortunately, such a world is far from existence. Throughout the project I have encountered many bugs and numerous obstacles that impeded my progress. The larger ones are mainly related to using third party code and compatibility.

To begin with, when I finally started implementing scene transitions, I began getting inconsistent crashes. I narrowed the location of the bug down to loading of textures for new scenes. Using the debugger, I pinpointed the exact line causing the error. It happened to be in the texture loader files that are quite popular among the OpenGL community. The constructor (function that creates the image) for the TGA image format was using a conditional statement to incorrectly determine whether to free memory associated with the image. It did not make sense to me why memory would be freed on construction, so I circumvented the freeing in the constructor. The results were that creating textures no longer trampled on memory in other locations. This also fixed an uncanny problem I was having where an ID field of the fifth collision polygon added to the collision manager would get corrupted. Up to that point, I had to uncomfortably add a "decoy" polygon to the collision manager so none of the good ones would get disturbed. It turns out its memory was usually part of the memory freed by the image

loader file. This bug took multiple days of searching to finally understand and fix, since the program did not crash at the same point every time.

A smaller issue related to textures was the conversion from 3D Studio Max to OpenGL. Texture coordinates generated in 3D Studio Max would not always carry over correctly through 3D Exploration to OpenGL. Specifically, the box-mapping function had problems. In 3D Studio Max I would apply a texture to an object as if it were being wrapped around a brick, and only two opposite faces would end up correctly mapped in OpenGL. The rest would have streaks of texture as pixels were dragged along the surface. This can be seen in the section with the stone structures that the viewer can walk under. These were all box-mapped. I did not detect the problem until the scene was in OpenGL, at which point is would have taken more time than it was worth to go back and change all the boxes to individual rectangles and map all six faces of all the blocks individually with a planar function.

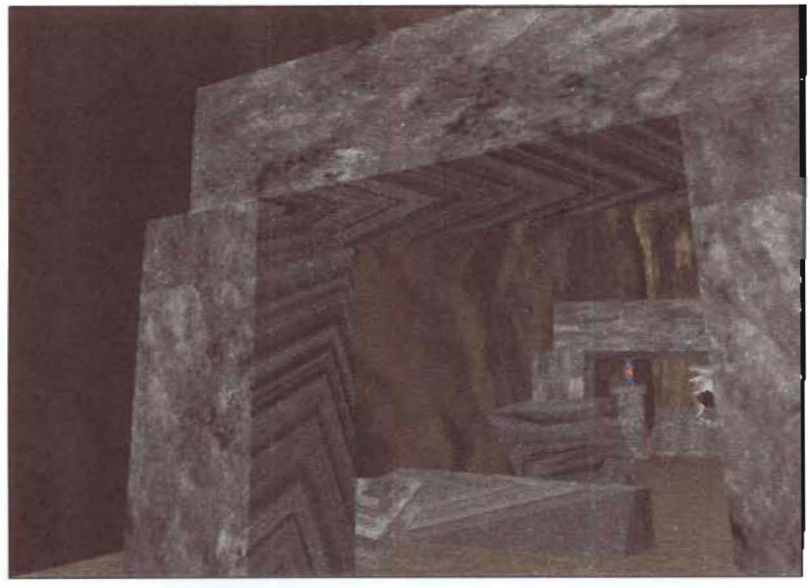


Figure 16.1 Streaking seen on the inside and underside of blocks imported from 3D Studio Max.

16.2 On The Mechanics of Fog and Lighting

The first time I tried to make use of fog was in my underground lake scene. I did not want the viewer to be able to see the far end of the cave until making it most of the way inside. I applied a simple linear, gray fog function to the whole scene. It worked well except that the water was not behaving as expected. The water did not seem to be shading enough at some times, and more than necessary at other times. It was also shading the entire (sheet) of water the same, rather than getting foggier with distance. I eventually realized by standing in the middle of the water and spinning around, that the amount the sheet of water was shaded depended on the camera's angle in the horizontal plane. At 0, 90, 180, and 270 degrees, the fog was at a minimum and directly in between each of those adjacent values was a maximum. At the minima, the camera's line of sight was perpendicular to one of the water's edges, whereas the maximums were facing corners. What I came away with was that since the water was just one large rectangle (where the rest of the cave was made of smaller triangles) the whole thing was getting shaded the same — more when the camera was facing a corner since the corners were further away than any edge. It also did not help that I was standing in the middle of the polygon that was being shaded, either.

This was a simple lesson for me early in the project, before I had designed all of my scenes. From then on, I tessellated the polygons in my scene to be smaller than a certain size, no larger than about three meters on either side (for rectangles). The smaller the polygons are, the smoother fog looks. *Tessellation* is the term for breaking one polygon up into a larger number of smaller polygons. Lighting also works in a similar way, giving closer approximations to real world lighting as the amount of tessellation approaches infinity — of course, you do not want to tessellate until the frame rate drops to sloth-like speeds, but rather find a suitable balance.

16.3 Collision Setup

One task that became tedious after a while was collecting points for collision polygons. I had no method to feed polygons into that would know how to break them down into representative polygons. Such a method would also be hard pressed to give meanings such as "ground" or "wall" to the polygons, so I had to do all that myself. I drew cross-hairs on the screen so I could pinpoint locations, then typed the points into various arrays that I used to create the polygons. It would have been nice not to have to spend so much time doing this, but I would have spent more time doing it any other way, and it had to get done.

16.4 3rd Party Bugs

Now the largest problem of them all: trying to use third party software (aside from the TGA loader that worked in the end). Many times I attempted to use packages for collision detection, alpha-texture image loaders, DirectX wrappers, and OpenAL that just did not work. Usually, I found that libraries my development environment needed to have installed to work with these packages were not put together for CodeWarrior. Libraries are specific to the compiler, and there are more common ones than CodeWarrior that I found most libraries were made for, such as Visual C++. If I could not find the necessary libraries, I tried installing the files they were created from, but often they required other files I did not have. I corresponded with individuals designated to assist programmers such as myself, but to little avail.

The one instance I was helped well was when I was trying to install OpenAL. I informed one of the key figures of errors I was getting, and got the reply "we did an update of the Windows code last week, but discovered a few days ago that the include files ... were the wrong versions." Eventually, after much drudgery I will not discuss here, I was able to get OpenAL to compile — a huge success. However, it was soon to

be outdated by a newer version. On the unfortunate side of things, the version I have still seems to be unstable.

CHAPTER 17: Conclusions

17.1 Results

I am very pleased with the distance this project has come, considering I was a complete novice at the beginning of the year. Towards the beginning, it took two weeks just to get everything installed correctly and write a program that drew a red square. One of the first 3-D objects I used was the standard teapot [8], but it looked "funny". It took me a while to figure out that depth testing was not activated, something that I would be able to notice in a second now. I went from taking over a month to design one scene to creating more than one per week at times. My productiveness sped up immensely throughout the year.

To briefly recap the major work I have done: scene design, layout, object modeling, object importing, texturing, dynamic texturing, multitexturing, texture transitions, texture special effects, terrain mapping, particle engine design for snow and water systems, reflections feigned on curved surfaces, accurate reflections on flat surfaces, collision detection, pleasing collision reaction, basic inclusion of physics of motion, interaction through picking, a realistic interface, work with 3-D sound, optimization, fog and lighting effects, and inclusion of time with day-night transitions. All work that was done with one goal in mind — *to create a realistic and aesthetic world that can be navigated in real time.* It is a goal I think I definitely achieved.

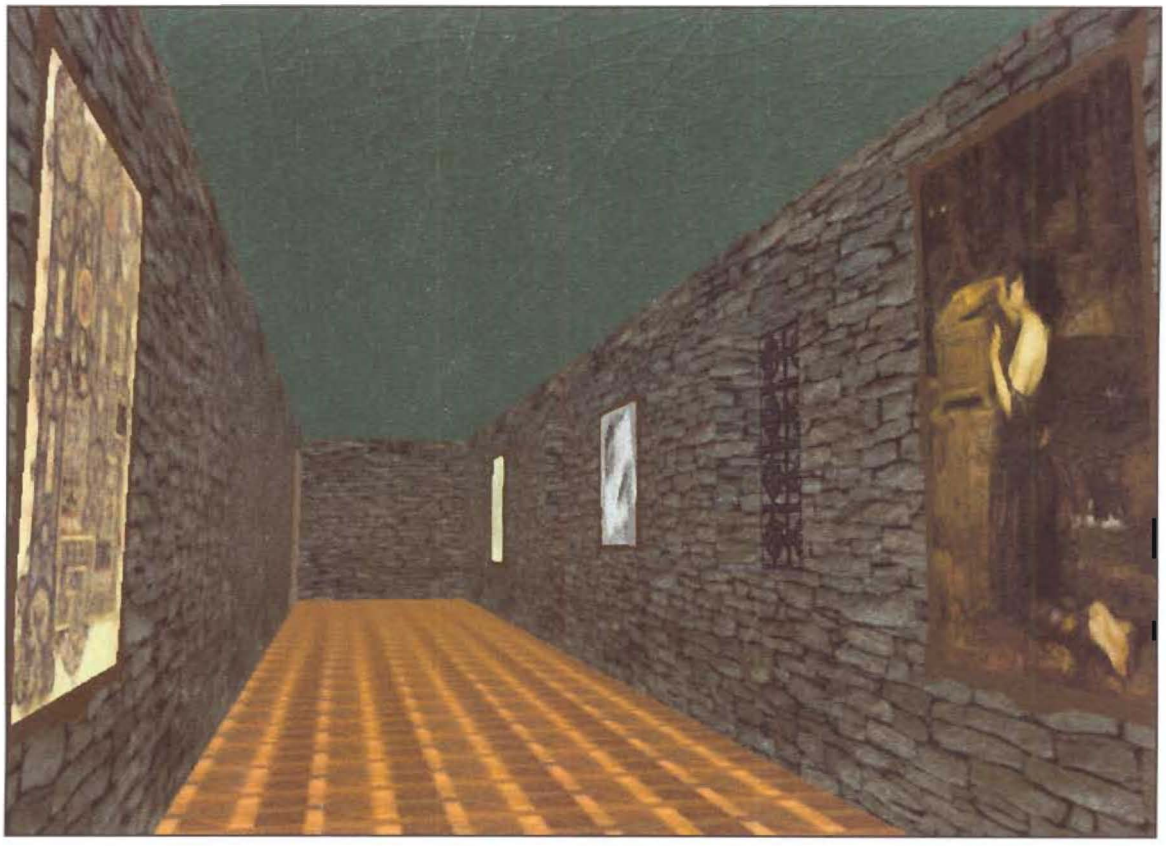
17.2 Future Work

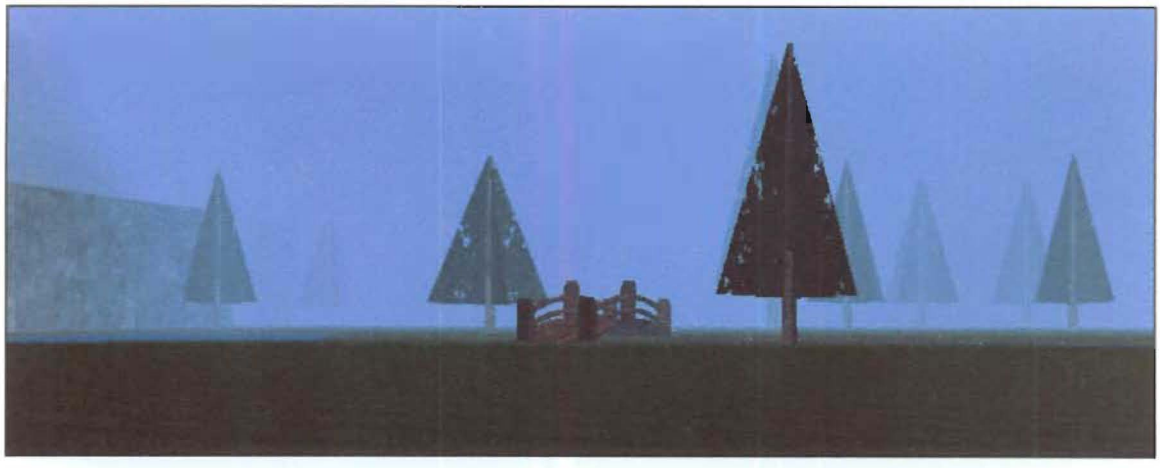
Inevitably, this project can be extended in many ways. This section is aimed towards anyone who may want to continue work on this project, and the possible areas to focus on.

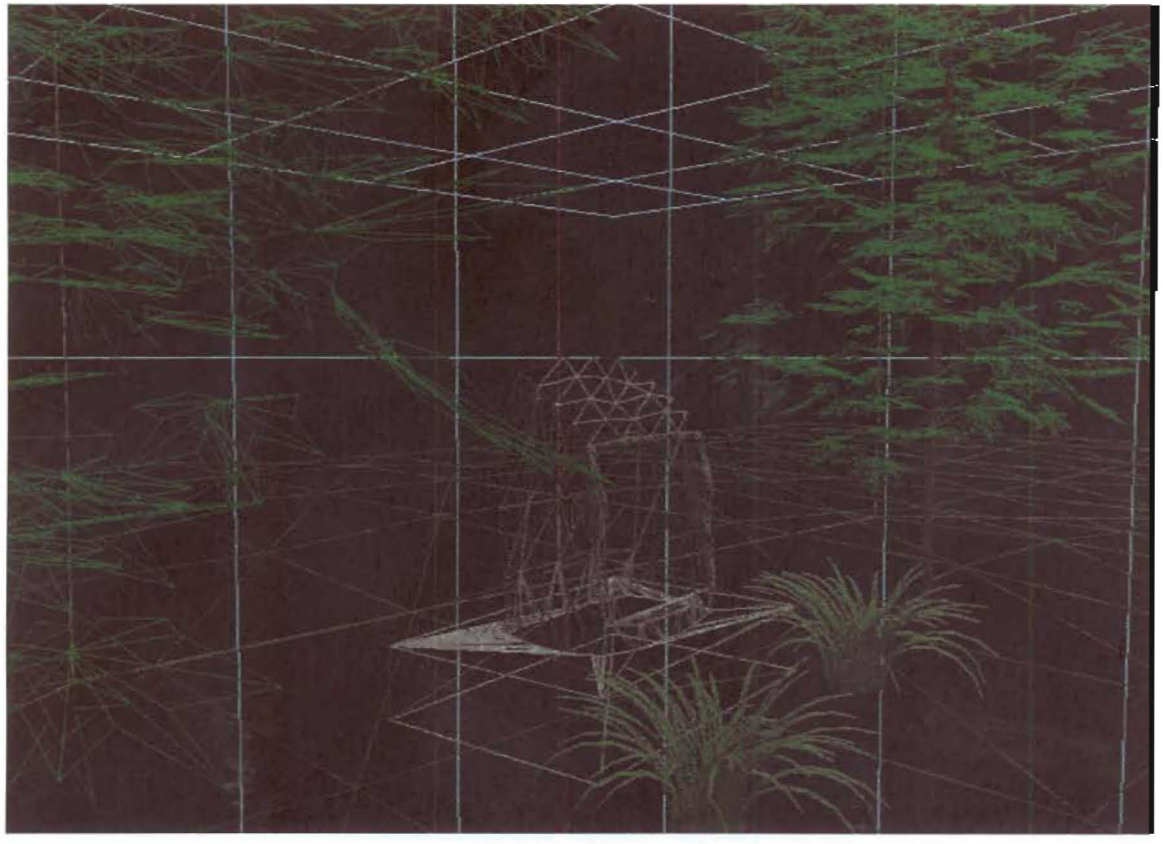
- **Sound:** Installing a newer, robust version of OpenAL, as it is quickly maturing. With it in place, use sounds included with the project and others to fill the world with realistic sounds. I suggest having the sound of the viewer's footsteps change when walking on different surfaces; I also suggest switching from crow sounds during the day to crickets and owls at night (both of which I have sounds for).
- **Code Organization:** I could not, in the interest of time, focus primarily on the structure and neatness of my code. There is definitely room to make it more object-oriented and somewhat cleaner. I am sure there are files with duplication of code, and these could be fixed. One of my obstacles in this respect was being new to the C++ language, so this may be better suited for someone with prior C++ experience.
- **Animation:** Adding complex animations, like a creature that has moving legs and a head that can walk around the world, or a flying bird.
- **Optimization:** Implement "Octrees" [10]. They are supposedly wonderful for optimization of collision detection, and culling portions of scenes that do not need to be drawn, allowing for larger, more detailed scenes with more interaction.
- **Shadows:** Real-time shadows on more than just one plane. Ray-casting is used in some methods for shadows, which will make this easier since I have implemented ray-casting already for collision detection.
- **Collision Detection:** The current methods for collision detection are good enough, but have room for extension. Create a function for automatically approximating surfaces and turning them into collision boundaries. Or try MathEngine's collision detection package that is included in the files with this project. I personally got more out of implementing it myself, however, and I suggest the same.
- **Stereo Vision:** It would be very cool to modify the world to render into stereo buffers and have it output to a stereo vision visor, with a head tracker to control viewer orientation.

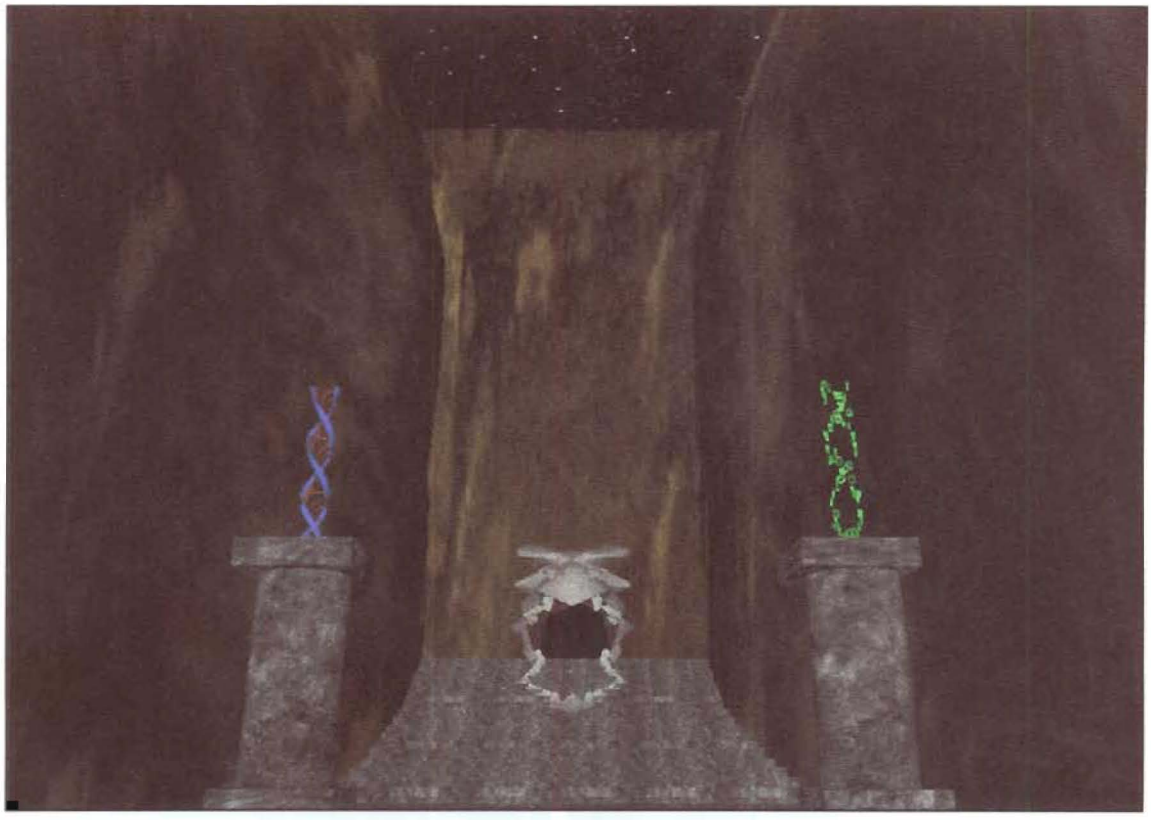
- **Purpose/Story:** Use the world for something: experimentation, scientific visualization, or a game — give it a purpose or a story besides that of understanding the mechanics of real-time 3-D graphics programming.

Each of these would be a great amendment to the current state of the project. Anyone who works with this project or a similar one long enough to get past the initial learning curve will discover just how fun real-time, three-dimensional graphics programming can be.









Resources

[1] 3D CAFÉ'S FREE 3D MODELS MESHES. Online. Internet. [April 19, 2001]. Available WWW: http://www.3dcafe.com/asp/meshes.asp

[2] 3D Exploration. Online. Internet. [December 14, 2000]. Available WWW: http://www.xdsoft.com/explorer/

[3] 3D Studio MAX Release 3. Online. Internet. [December 14, 2000]. Available WWW: http://www2.discreet.com/products/d_products.html?prod=3dsmax

[4] 3D Techniques Used In Games. Online. Internet. [May 2, 2001]. Available WWW: http://www-stud.fh-regensburg.de/~kuo32652/englisch/gamegraphic.html#billboarding

[5] Billboarding by Nate Miller. Online. Internet. [May 2, 2001]. Available WWW: http://nate.scuzzy.net/docs/billboard.html

[6] Final Fantasy VI — Features. Online. Internet. [April 18, 2001]. Available WWW: http://www.squaresoft.com/web/games/anthology/FFVI/features.html

[7] Garden Games – Zelda64. Online. Internet. [April 18, 2001]. Available WWW: http://europa.spaceports.com/~layout/zelda64/screenshots.html

[8] The History of The Teapot. Online. Internet. [April 18, 2001]. Available WWW: http://web2.iadfw.net/sjbaker1/software/teapot.html

[9] Introduction to DirectX Graphics. Online. Internet. [December 14, 2000]. Available WWW: http://msdn.microsoft.com/library/psdk/directx/dx8_c/hh/directx8_c/_dx_introduction_to_directx_graphics_graphics.htm

[10] Introduction to Octrees. Online. Internet. [May 1, 2001]. Available WWW: http://www.flipcode.com/tutorials/tut_octrees.shtml

[11] Jeff Molofee's OpenGL Windows Tutorial #27. Online. Internet. [May 1, 2001]. Available WWW: http://nehe.gamedev.net/tutorials/lesson27.asp

[12] Johnsonbaugh, Richard; Kalin, Martin. Object-Oriented Programming in C++, Second Edition. Prentice Hall, 1999.

[13] Miller, Kurt. Collision Detection. Online. Internet. [April 19, 2001]. Available WWW: http://www.flipcode.com/tutorials/tut_collision.shtml

[14] Multitexture. Online. Internet. [April 23, 2001]. Available WWW: http://reality.sgi.com/blythe/sig99/advanced99/notes/node60.html

[15] Neon Helium Productions. Online. Internet. [April 16, 2000]. Available WWW: http://nehe.gamedev.net/opengl.asp

[16] OpenAL.org. Online. Internet. [December 14, 2000]. Available WWW: http://www.openal.org

[17] OpenGL.org. Online. Internet. [December 14, 2000]. Available WWW: http://www.opengl.org

[18] Range-Based Fog. Online. Internet. [April 23, 2001]. Available WWW: http://msdn.microsoft.com/library/psdk/directx/DX8_C/hh/directx8_c/_dx_range_based_fog_graphics.htm

[19] Rollings, Andrew; Morris, Dave. Game Architecture and Design. Coriolis Group, 2000.

[20] Sony Playstation2 Specification. Online. Internet. [April 23, 2001]. Available WWW: http://www.e-scapegames.co.uk/playstation2.htm

[21] Texture Mapping. Online. Internet. [April 19, 2001]. Available WWW: http://www.futuretech.vuurwerk.nl/tex.html

[22] Tony Hawk 2 – Pictures Page 3. Online. Internet. [May 3, 2001]. Available WWW: http://videogames.about.com/games/videogames/library/blthawk2dcpics3.htm

[23] Unreal Tournament. Online. Internet. [May 1, 2001]. Available WWW: http://www.unrealtournament.com/

[24] Weir, Peter. The Truman Show. DVD. Paramount Studio. 1999.

[25] Wright Jr., Richard. OpenGL SuperBible, Second Edition. Waite Press, 1999.