

3D ENGINE FOR IMMERSIVE VIRTUAL ENVIRONMENTS

A Thesis

by

CHRISTOPHER DEAN ANDERSON

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

MASTER OF SCIENCE

December 2003

Major Subject: Visualization Sciences

3D ENGINE FOR IMMERSIVE VIRTUAL ENVIRONMENTS

A Thesis

by

CHRISTOPHER DEAN ANDERSON

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

MASTER OF SCIENCE

Approved as to style and content by:

Frederic I. Parke
(Chair of Committee)

Peter F. Stiller
(Member)

Donald H. House
(Member)

Phillip J. Tabb
(Head of Department)

December 2003

Major Subject: Visualization Sciences

ABSTRACT

3D Engine for Immersive Virtual Environments. (December 2003)

Christopher Dean Anderson, B.C.S., Southwest Texas State University

Chair of Advisory Committee: Dr. Frederic I. Parke

The purpose of this project is to develop a software framework, a 3D engine, which will generate images to be projected onto facets of a spatially immersive display (SID). The goal is to develop a software library to support the creation of images of specified 3D environments which are specific to the display geometries of a polyhedral class of SIDs. Part of this goal is developing auxiliary software to allow this library to be thoroughly tested. When properly working, the images being displayed on adjoining faces of the SID appear spatially and temporally consistent with one another, creating the illusion that the user is within a surrounding three-dimensional space.

This thesis is dedicated to my favorite person, the wonderful Miss Erin.

TABLE OF CONTENTS

	Page
ABSTRACT	iii
DEDICATION	iii
TABLE OF CONTENTS.....	v
LIST OF FIGURES	viii
1 INTRODUCTION	1
2 PRIOR WORK	5
3 PROBLEM STATEMENT AND METHODOLOGY.....	9
3.1 Project Goals	10
3.2 Developed Software Functions	11
3.3 Software Development Approach	12
4 IMPLEMENTATION.....	13
4.1 Implementation Approach	13
4.2 Model and Light Classes.....	15
4.2.1 Drawing Primitives.....	16
4.2.2 Lights	17
4.2.3 Surface Normals.....	19
4.2.4 Colors and Lighting Calculations	20
4.2.5 Per Vertex Colors	22
4.2.6 Texture Mapping.....	23
4.2.6.1 Basic Color Mapping	23
4.2.6.2 Reflection Mapping	24
4.2.7 Object Transformations	25
4.2.8 Object Parenting	26
4.2.9 Object Instancing.....	27
4.2.10 Object Files.....	28
4.2.11 Object Display.....	28
4.3 Scene Class.....	29
4.3.1 Camera Structure, Viewing Frustum.....	29
4.3.2 Background Color, Image and Skybox	31
4.3.3 Fog	33
4.3.4 Window Structure	34
4.3.5 Scene Files	35
4.4 Cave Class.....	35
4.4.1 Cave Geometry	37
4.4.2 Display Adjustment and Cave Files	38
4.4.3 Update Viewer Transforms and Projection Matrix.....	38

	Page
4.4.4 Environment Transformations.....	44
4.4.5 Drawing Borders.....	44
4.4.6 Visual Aids.....	45
4.5 Networking.....	45
4.6 Terrain Class.....	47
5 RESULTS.....	47
5.1 Cave Demo Application.....	48
5.2 Bubble Factory Project.....	50
6 CONCLUSION AND FUTURE DIRECTIONS.....	53
REFERENCES.....	55
APPENDIX A: INSTALLATION.....	57
APPENDIX B: USAGE.....	58
APPENDIX C: PROPRIETARY FILE FORMATS.....	61
C.1 Cave File Format.....	61
C.2 OBJ Extended 3D File Format.....	62
C.3 Scene File Format.....	63
APPENDIX D: CLASS AND STRUCTURE REQUIREMENTS.....	64
APPENDIX E: CLASSES.....	65
E.1 Cave Object Class.....	65
E.2 Image Class.....	67
E.3 Light Class.....	69
E.4 Model Class.....	70
E.5 Scene Class.....	74
E.6 Singly-linked List Class.....	76
E.7 Terrain Class.....	78
E.8 Texture Class.....	78
E.9 Length 3 Vector Class.....	79
APPENDIX F: STRUCTURES.....	86
F.1 Camera Structure.....	86
F.2 Fog State Structure.....	87
F.3 Window Structure.....	88
APPENDIX G: MISCELLANEOUS CODE.....	89
G.1 Color Type And Definitions.....	89
G.2 GLUT Environment.....	90
G.3 OpenGL Error Checking.....	91

	Page
G.4 Trigonometric Approximations.....	92
VITA.....	93

LIST OF FIGURES

FIGURE	Page
1 CAVELib setup with no network	5
2 CAVELib setup with network	6
3 WireGL / Chromium general setup	7
4 Scene data layout; the storage of virtual environments	14
5 Four drawing primitives are provided	16
6a Directional light on a terrain	18
6b Point light on the same terrain as figure 6a	18
7a Smooth shaded models have one surface normal per vertex	19
7b Flat shaded models have one surface normal per face	19
8a Diffuse only	20
8b Specular only, shininess 5	20
8c Diffuse plus specular, shininess 5	20
8d Specular only, shininess 15	20
8e Diffuse plus specular, shininess 15	20
9 Vertex colors make cheeks blush and give the eyes pupils	22
10a Blank geometry	24
10b Vertex texture coordinates	24
10c Textured geometry	24
11a Spherical environment map	25
11b Sphere with the environment map applied	25
12 The upper arm is the parent of the forearm, which is the parent of the hand, which is the parent of the thumb and fingers	26
13 Each egg is an instance of the same geometry but has an independent location, orientation, and scale	27
14a Angle of view is small (zoomed in)	30
14b Angle of view is large (zoomed out)	30
15 Symmetric viewing frustum	31

16	A skybox image is six images in one, each looking positive or negative along a Cartesian axis	32
17a	Linear function	34
FIGURE		Page
17b	Exponential function at 0.04 density	34
17c	Exponential function at 0.1 density	34
17d	Exponential squared at 0.04 density	34
17e	Exponential squared at 0.1 density	34
18a	Trapezoidal Icositetrahedron	37
18b	Rhombicuboctahedron	37
18c	Truncated Icosahedron (soccer ball)	37
19a	Scaled facet	38
19b	Horizontally skewed facet	38
20	The current side is the base side plus an index	39
21	A ray is shot from the viewer in the direction opposite the facet normal until it collides with the facet	40
22	rotaxis is orthogonal to vector [collpos – viewer] and forward	41
23	upward is rotated about rotaxis through rotation ₀ to determine upvector ₀ ..	41
24	upvector ₀ is rotated about the normal direction through rotation ₁ to determine upvector ₁	42
25	Facet vertices are rotated by rotation ₀ about rotaxis and by rotation ₁ about the normal direction	43
26	Near clipping distance is the distance from the viewer to the facet in the normal direction (collision ray)	43
27	On the TAMU system, each machine has an identical copy of the environment	46
28a	Height map	47
28b	Terrain generated from height map	47

1 INTRODUCTION

Plato introduces the fundamental cave allegory in Book VII of The Republic [8]. In his example, humans are imprisoned in a cave whose only opening is behind them. They are chained in such a way that they cannot turn around, so that all they can see is the back of the cave. Their only perception of objects is the shadows which are cast on the cave wall before them. Since they never see out of the cave, they do not perceive that anything exists outside of it. The images they see on the cave wall are their reality.

The act of communicating a proposed reality between groups of people has taken on innumerable expressions. Whether through writing, playacting, composing, painting, singing or teaching, people ranging from scientists to artists and from children to scholars have endeavored to engage other people and allow them to experience worlds which they have not yet known. Different forms of expression target different senses. A concerto may give a person the impression of being in a place she has never been, purely by sound. Theatrical productions may play to any of the five senses. The primary target of many mediums is vision. Movie theaters are a form of visual immersion. While an audience watches a film on a large movie screen, they see almost nothing but the screen. This gives them the impression that they are located in the environment where the story is taking place.

This thesis follows the style and format of *Computer Graphics*.

Due to the growth of computer imaging technology, new forms of visual expression are possible. Whatever we see in the real world can be simulated in a computer, and this virtual world can be viewed via a display device, such as a computer monitor. This idea of creating virtual objects or a virtual environment to represent real objects or a real environment has applications in many fields. Engineers can construct virtual prototypes and make corrections prior to building a real product. Astronomers can study virtual recreations of planets and galaxies. Movie makers can create virtual sets and even virtual characters to accomplish goals and tell stories that would not be feasible without the use of computer graphics.

Video games have become extremely popular, and most of them rely on the ability to give the game player the feeling that he is inside a fictitious environment. Game environments are often designed to simulate many aspects of the real world. They may consist of three-dimensional geometry which is illuminated and shaded to mimic real world lighting. Objects may be textured to give a high level of detail. Lights may cast shadows. A powerful aspect of making an environment believable is viewer interaction. Games allow a user to do many actions that she can relate to in the real world, such as walk, run, jump, swim, drive a vehicle, etc.

To engage a person with imagery, it is preferable that the person be free from distractions, so that they only see what is being presented to them. Immersing a person in a virtual environment is preferable for many visual interactive applications, but visual immersion has often been impractical due to the high cost of constructing immersive display systems. Immersive and semi-immersive displays have usually been driven by one or more high-end workstations [4], each of which cost tens of thousands of dollars. However, with the recent affordability of powerful graphics cards, high-speed network hardware, and

faster commodity a computer, paying a high price for an immersive display is no longer necessary [2]. Recent work has been done in the Visualization Lab at Texas A&M University to construct spatially immersive display systems composed of commodity hardware.

A spatially immersive display (SID) is one in which the display surfaces surround a user or users. Renderings of three-dimensional geometry are commonly generated to be shown on flat surfaces. That is, a digital spatial environment is projected from three dimensions into two dimensional images. For this reason, SID walls are typically flat. This is not realistic, in that 2D projections of a 3D environment can only approximate what a user would see if he were actually in the represented space. To best create the desired illusion, SID walls would be assembled in such a way that they approximate a surrounding sphere. This ideal display would have infinitely many facets, and each facet would be infinitely small, resulting in a perfect sphere. The simplest practical SID geometry is a cube. Although this is far from the ideal geometry, because of its simplicity it is by far the most common. The CAVEtm [9] is a cubic display system developed at the University of Illinois, and is the most common type of SID. Rarely are more than four of the six possible display walls utilized. Other SIDs, such as the Visionariumtm and Reality Centertm, use wrap-around (panoramic) techniques to project images of an environment onto a single curved screen, but they are not fully immersive. Head mounted displays (HMD) provide full immersion in that they allow the user to look in any direction. However, they generally do not provide peripheral vision.

This project presents the development of a software framework for the display of virtual environments on multi-faceted, spatially immersive displays. For this project a “3D engine” is considered to be the software which visually presents a virtual three-dimensional world to a user or users. The 3D engine presented in this discussion consists of an application-specific portion of code, and supporting libraries.

2 PRIOR WORK

The most widely used Application Programmer's Interface (API) for developing applications for immersive displays is CAVELib [1]. Initially designed and developed in the Electronic Visualization Laboratory at the University of Illinois, CAVELib is currently developed and distributed by VRCO Inc. CAVELib supports a wide variety of immersive and non-immersive displays, including but not limited to CAVEStm, RAVE'stm, ImmersaDeskstm, Visionariumstm, and RealityCenterstm. Designed primarily for display systems which run on high-end Silicon Graphics[®] systems, CAVELib may be configured to make use of multiple processors and multiple graphics pipes (see figure 1). The library supports a variety of controllers for user input, such as "wands" (3D mice). It also supports tracking sensors which may be used to render the environment relative to the position of the user's head, or to track the position of the 3D mouse.

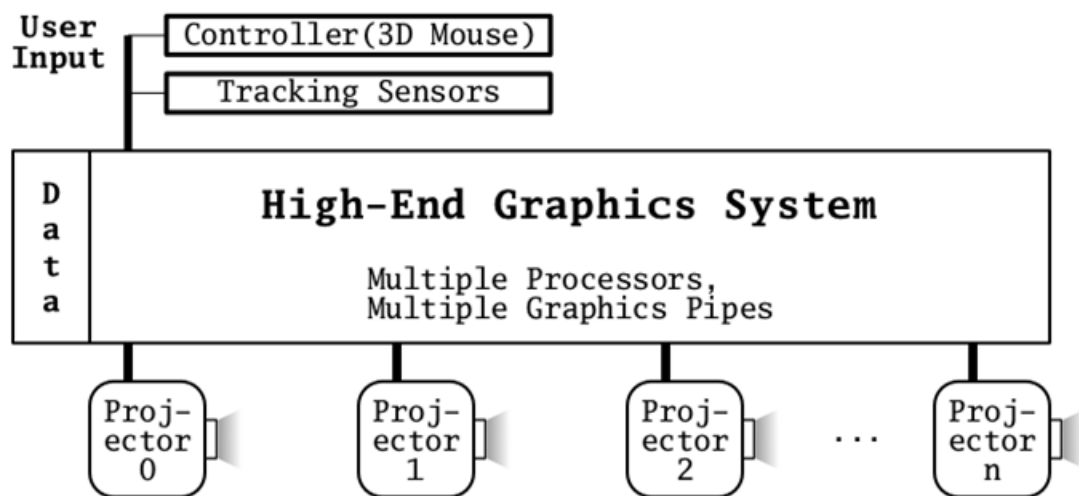


Figure 1: CAVELib setup with no network.

CAVELib has been extended to provide network support (see figure 2). Networking allows each of the display images to be rendered on a different machine. In this type of configuration, each machine has a copy of the data which describes the virtual world, and tracking and navigation data are shared between machines.

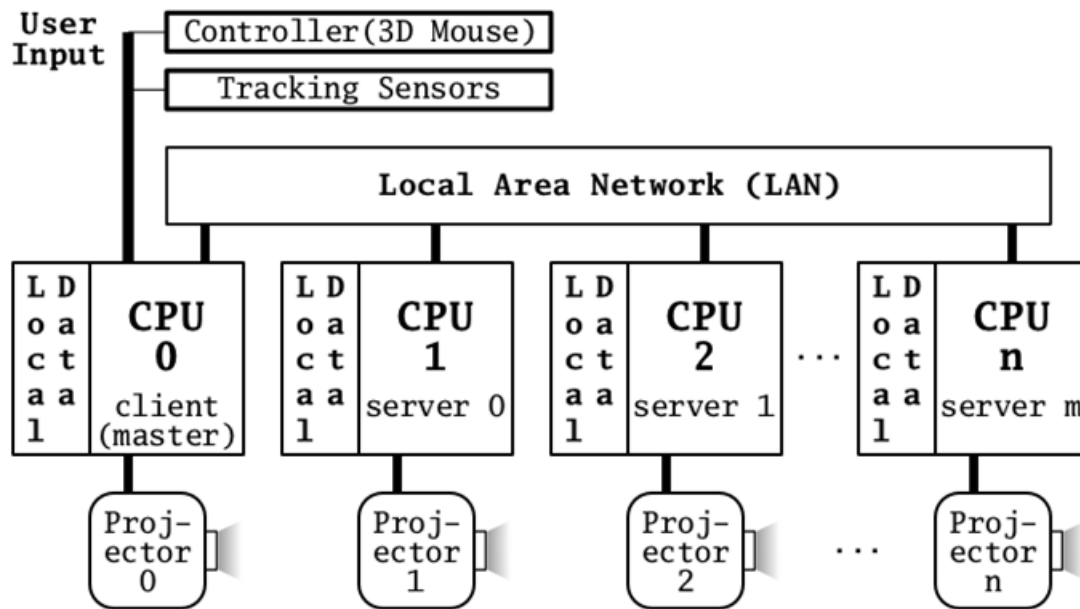


Figure 2: CAVELib setup with network.

WireGL, a research project at the Stanford University Computer Graphics Lab, developed software which allows graphics applications to render on clusters of commodity workstations [2,3]. Chromium, a software system that is derived from WireGL, provides rendering for multi-screen displays. In addition to providing the ability for applications to render on clusters of workstations, it has a robust way of handling distribution of graphics calls over networks [5]. WireGL and Chromium are implemented as an OpenGL driver. When the application makes OpenGL calls, the specialized driver distributes these graphics commands to servers across a network (see figure 3).

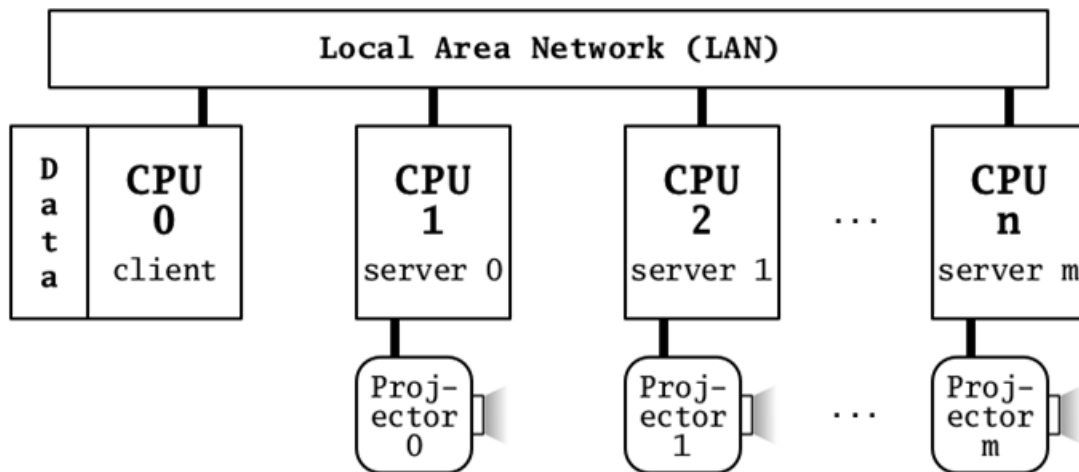


Figure 3: WireGL / Chromium general setup.

Though able to generate images for many types of displays, software such as CAVELib and Chromium are designed only to render rectangular images. SIDs having non-rectangular facets are not directly supported. Research and development has been and is still being conducted in the Visualization Laboratory at Texas A&M University (TAMU) to create SID systems which make

use of non-rectangular facets [7]. These provide better approximations to a sphere than other spatially immersive displays. Such systems have been constructed at TAMU using clusters of workstations. The focus of this thesis project is to develop supporting software that accommodates non-rectangular facet, polyhedral display surfaces.

3 PROBLEM STATEMENT AND METHODOLOGY

This project addresses the need for a software library that supports applications that generate images representing 3D environments for display on the facets of polyhedral spatially immersive displays. The software is not limited to specific display geometry. The geometry of the display facets may be any convex polyhedron. Unlike other immersive visualization software, facet geometry is not restricted to being rectangular or even quadrilateral. Each display facet may be any convex polygon. Triangles, pentagons, octagons and non-rectangular quadrilaterals are all valid facet shapes. Facets must be planar, however. The software is designed to work with commodity computers and projectors. This excludes specialized and possibly non-planar image projectors or optical systems.

The computational systems to be supported have been constructed as multiple networked nodes, where each node consists of a computer with high-performance display hardware, a projector, and one facet of a multi-facet display. By distributing graphics computation across nodes, one CPU per facet, a large number of nodes may be used with negligible speed loss. The number of facets is not a significant factor in determining overall system frame rate. The network communication protocol ensures that all nodes generate frames which correspond temporally to one another. The software allows interactive update of viewer position and orientation within the displayed virtual space.

3.1 Project Goals

The goal of this project is a tested software library that supports SID applications which:

- Generate images that are correctly oriented, clipped, and otherwise properly conformed for projection onto the facets of a class of polyhedral SIDs.
- Support interactive changes in position and orientation of the viewer. Changes in position affect the way geometry is projected into two dimensional images. Orientation changes affect the generation of stereo images.
- Support the networking of multiple computational graphics nodes so that all SID facet images can be displayed simultaneously and are temporally synchronized.
- Support stereo viewing via anaglyphic rendering.

Detailed user documentation has been developed as an aid to application development. This documentation is included as appendices to this thesis.

3.2 Developed Software Functions

The functions of the 3D engine software are as follows:

- Load data from files which defines the geometry, lights, and various display properties of the desired three-dimensional environment.
- Load data from files which specifies the geometry of the display facets and various attributes of the display system.
- Handle interactive user input from devices such as a keyboard and a mouse.
- Allow the geometric data to be manipulated by application software which makes use of the 3D engine.
- Render the resulting environment geometry using the OpenGL API [10].
- Generate images that conform to the specified display geometry, which are subsequently projected onto the corresponding display surfaces.
- Networked graphics nodes communicate utilizing the User Datagram Protocol (UDP). Viewer position and orientation with respect to the virtual world, and viewer position relative to the display geometry are sent by the client machine to each of the servers for every frame.
- Image generation for each graphics node is temporally synchronized with all other networked graphics nodes. Each server is contacted by the client every frame and the server returns the number of frames that it has rendered. Comparison of frame counts is used to maintain synchronization.

3.3 Software Development Approach

Modern, high quality software development practices were used to develop the software library. This software allows maximum flexibility in supporting application software, while maintaining high performance. The software was created using the C++ language, making use of its object-oriented features to keep various pieces of the software organized and intuitive. For portability, flexibility, and acceleration in rendering 3D graphics, the OpenGL API was chosen. It was supplemented with the OpenGL Utility Toolkit (GLUT). User tracking and navigation data are shared by graphics nodes over a local area network (LAN) via a basic UDP socket interface. Rendering non-rectangular images is achieved by rendering all pixels that lie outside the boundary of each facet as black. Stereo viewing is achieved by rendering images for the left and right eyes in disjoint color ranges and viewing these anaglyphs through glasses with appropriate color filters.

4 IMPLEMENTATION

Prior to creating software to support specialized display geometry, initial software was created to render a virtual world on a standard rectangular display. By previewing a virtual world on a simple single-screen display, we have a basis for judging how it should appear on more complex displays. We can judge how objects appear in terms of geometry and colors, and how they relate to one another in space. To accomplish this, a collection of libraries was created that facilitates the storage, manipulation, and display of the various types of data that specify the virtual world. Support was later added to facilitate displays having polyhedral geometry.

4.1 Implementation Approach

Consistent with the object-oriented software approach, which is supported by C++, the supporting libraries consist of a number of “classes.” Figure 4 depicts the data layout of virtual environments. A *model* class was created to store objects. Objects have geometry, which is defined in a local (object) coordinate system. Object geometry may be lines, triangles, quads, or other polygons. Objects may have surface normals used for shading. They may have material colors, vertex colors, and textures. Objects may be parented, and they may share geometry and textures. They may be read from Extended OBJ files (see Appendix C.2).

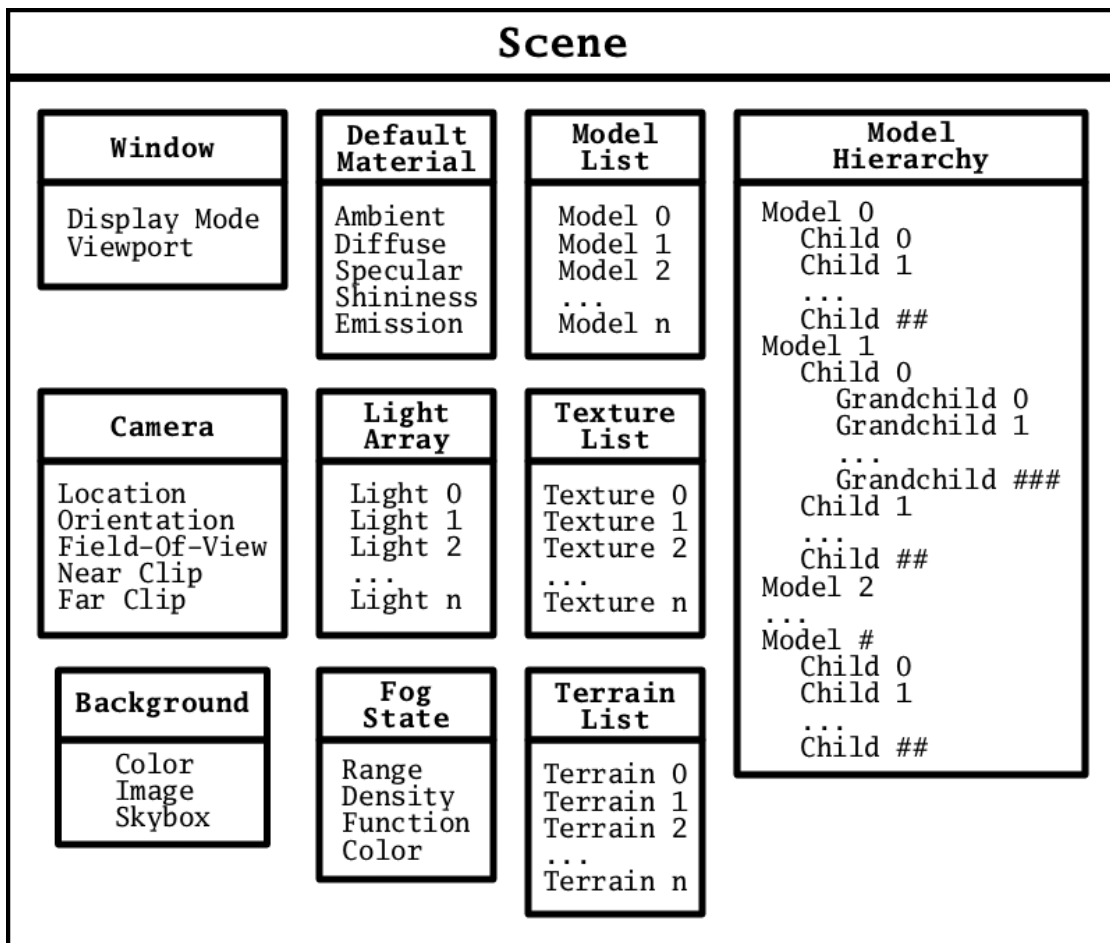


Figure 4: Scene data layout; the storage of virtual environments.

A *light* class was created to support lighting and shading. Lights have the following attributes: ambient color, diffuse color, and specular color. Lights may be directional or positional (point lights). Lights may be specified within scene files (see Appendix C.3).

A *scene* class was created to contain all elements that define a virtual world. A scene may have models, lights, terrains, and fog. A window structure defines the type of image buffer and viewport for rendering. A camera structure specifies the viewer location and orientation within the world. The camera also

specifies the viewing frustum by a field-of-view angle, near clipping distance, and far clipping distance. All scene elements and parameters may be specified within scene files.

A *cave* class was created to provide transformation and projection calculations for viewing through a specific facet of the specified display geometry. Output may be rotated, scaled, or skewed. The display geometry may be viewed from the outside, and other visual aids may be generated to help in understanding the current *cave* setup. Display geometry and parameters may be specified within *cave* files (see Appendix C.1). The projection calculations in the *cave* class override the field-of-view angle and near clipping distance in the camera structure.

The networking is done via UDP sockets. One machine, the client, contacts each of the other machines, the servers, and sends viewer location and orientation information each frame time. Each server sends a frame counter back to the client. The client ensures that all servers have the same frame counter that it does. This way all the display nodes are synchronized to within one frame time.

4.2 Model and Light Classes

The purpose of our virtual world is to give the illusion of a believable world. To do this, the world must consist of some collection of objects, and it must have some way to simulate the illumination of those objects in at least a semi-realistic way. The *model* class (see Appendix E.4) was created to store an object, including its geometry, its position and orientation, and various attributes to determine how the geometry is to be drawn.

4.2.1 Drawing Primitives

With the OpenGL API, objects are rendered as collections of geometric primitives. These primitives consist of individual vertices. Vertices may be interpreted either one at a time to define points (GL_POINTS), as pairs to define line segments (GL_LINES), as triplets to define triangles (GL_TRIANGLES), or four at a time to define quadrilaterals (GL_QUADS). Other primitives are also available.

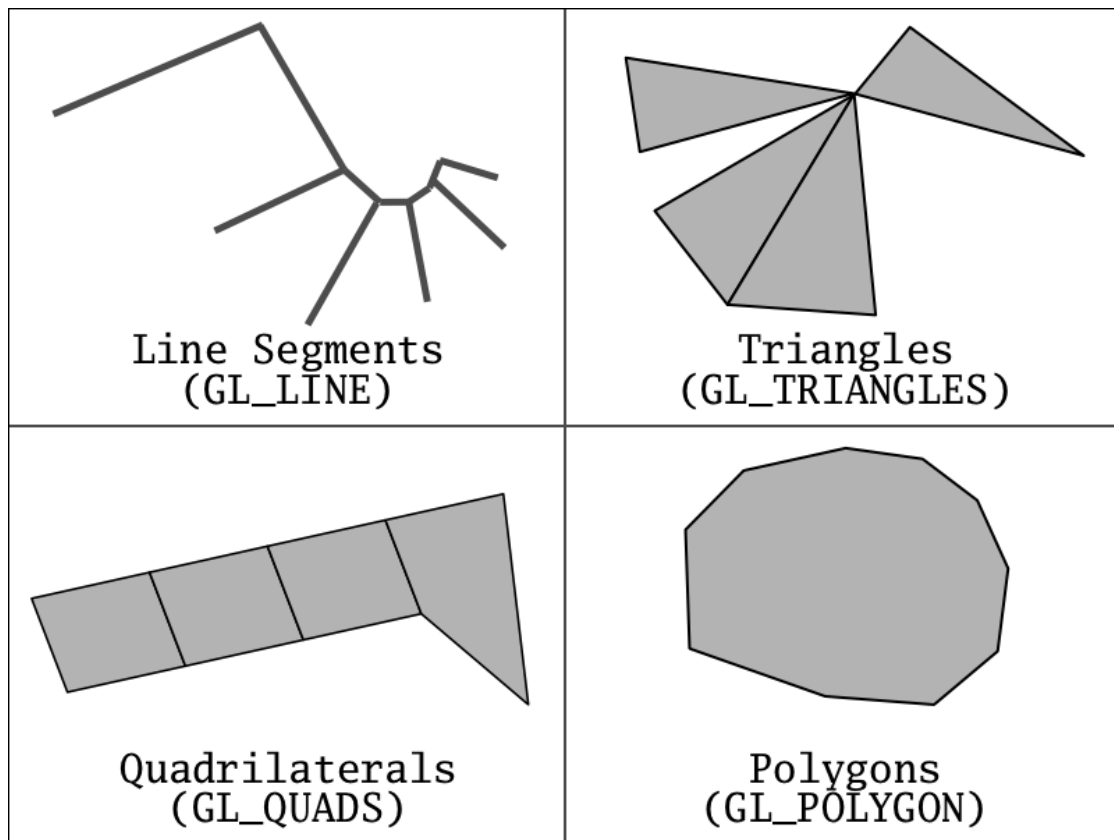


Figure 5: Four drawing primitives are provided.

Initially, only quadrilaterals (GL_QUADS) were used, as many models are created as a collection of adjacent quadrilaterals. Storing geometry with only a single primitive type was very restrictive. Support for triangles (GL_TRIANGLES), line segments (GL_LINES), and polygons having more than four sides (GL_POLYGON) was added (see figure 5). Throughout this discussion, when it does not matter which type of polygon (triangles, quads, more than four sides) is being considered, polygons will be referred to as “faces.” All faces are interpreted as having only one visible side. Vertices are assumed to specify faces in counter-clockwise order. If a face is oriented such that its vertices will be drawn in counter-clockwise order, it is front-facing, and is drawn. If its vertices are oriented in clockwise order from the point of view of the viewer, they represent a back-face, and are not drawn (culled).

4.2.2 Lights

Many of the attributes the *model* class stores relate to how OpenGL calculates colors at each vertex, and interpolates between vertex colors to find a color for each pixel. Many of these attributes are related to lighting. The *light* class (see Appendix E.3) was created to hold all of the attributes necessary to define a light source.

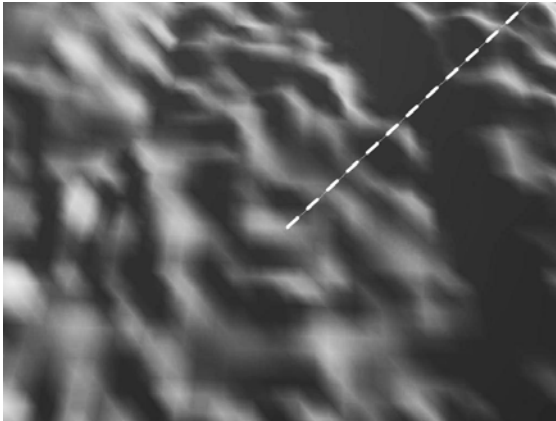


Figure 6a: Directional light on a terrain. Light direction is represented by a line.

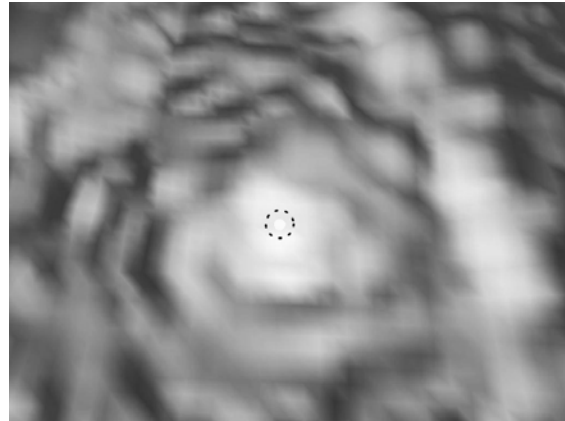


Figure 6b: Point light on the same terrain as figure 6a. Light location is represented by a small bright sphere.

Two types of lights; directional light sources and point light sources are supported. Directional lights “shine” on all geometry from the same direction (see figure 6a). Since they are defined as a direction and not a position, they may be thought of as being infinitely far away, and are often used to simulate sunlight. Point lights are positional. The direction from which they shine on geometry is determined by their position relative to that geometry in space (see figure 6b). They are also called omni-directional lights because they have no directional element. There is no constraint as to which direction they shine.

4.2.3 Surface Normals

The geometry of an object may consist of any number of surfaces. Surfaces are groups of contiguous faces. For the lighting calculations, OpenGL must know the surface normals of the objects. These are vectors that specify directions that are orthogonal to the surface. If an object is to be smooth-shaded, surface normals are specified at each vertex. Interpolation between computed vertex colors determines surface shading (see figure 7a). This gives the impression of a curved surface. In this case, the *model* class stores one normal vector per vertex. If an object is to be flat-shaded, meaning that each normal is consistent across a face and no interpolation is done, the *model* class stores one normal vector per face. Flat shading gives the impression of a surface consisting of flat faces (see figure 7b).

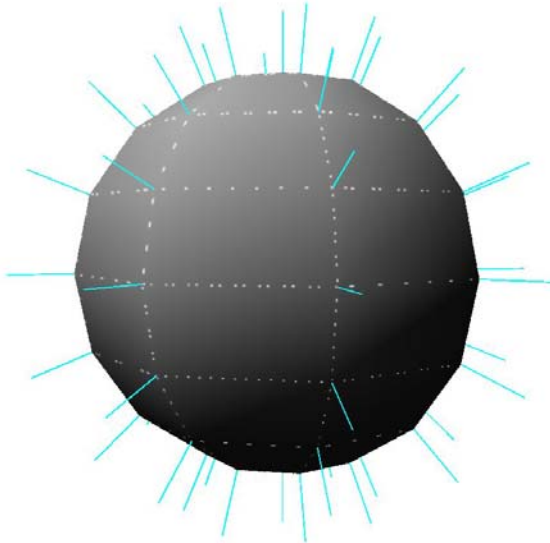


Figure 7a: Smooth shaded models have one surface normal per vertex.

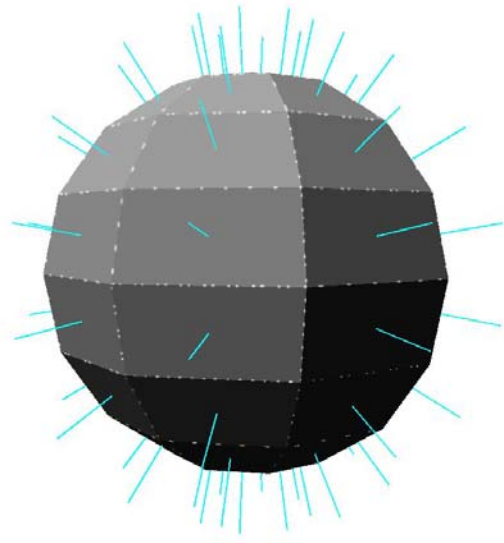


Figure 7b: Flat shaded models have one surface normal per face.

4.2.4 Colors and Lighting Calculations

Each light has the following color attributes which it contributes to the lighting calculations: ambient color, diffuse color, and specular color. Colors, unless otherwise specified, consist of red, green, blue, and alpha (RGBA) values. Alpha is a measure of opacity. In addition to these “light” colors, OpenGL uses “material” colors to calculate a color for each vertex. Although material colors may be specified per vertex, it is often acceptable to have material colors be consistent for an entire object, since it makes sense to treat most objects as consisting of a single material. To support this, the *model* class provides storage for the following material colors: emission, ambient, diffuse, and specular.



Figure 8a: Diffuse only.



Figure 8b: Specular only, shininess 5.

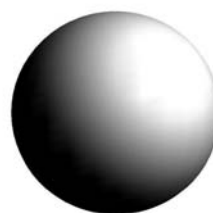


Figure 8c: Diffuse plus specular, shininess 5.

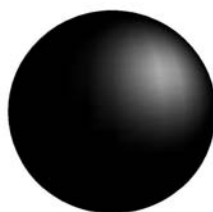


Figure 8d: Specular only, shininess 15.



Figure 8e: Diffuse plus specular, shininess 15.

In calculating color for a vertex, OpenGL starts with the material emission color, which is not dependent on lighting. Added to that is a global OpenGL ambient color which is scaled by the material ambient color. Added to that is the color determined by lighting calculations. This involves combining the light colors for each light source with the material colors of the object. Ambient color is not dependent on positions or directions, so each light's ambient color is simply scaled by the ambient material color.

Diffuse color is dependent on the angle between the light source direction vector from this vertex and the surface normal vector (see figure 8a). When this angle is zero (cosine of the angle is 1.0), the full amount of diffuse light color is used. When the angle between the surface normal and light direction is ninety degrees or more (cosine is less than or equal to zero), the diffuse light component is set to zero or black. Each light's diffuse color is scaled by the diffuse material color.

Specular color is dependent on the light direction vector, the surface normal vector, and the vector from the current vertex to the viewer (viewpoint). Consider shooting a ray from the viewer (your eyes), and reflecting it off of the object. The closer that reflected ray is to being parallel to the light direction vector, the brighter that vertex will be. This specular color component is then raised to a power which determines how quickly the color darkens (falls off) when the reflected ray gets further from being parallel to the light direction vector. This exponent is called shininess. The larger it is, the quicker the light "falls off" (see figures 8b – 8e). Conversely, a low shininess value causes the object to appear dull. This shininess value is also stored by the *model* class.

4.2.5 Per Vertex Colors

Although it is generally sufficient to have a single set of material colors per object, there are cases where it is useful to specify material color per vertex. For instance, consider a model of a human face that changes colors to represent changes in emotion. If this character's cheeks are to blush, a simple way to accomplish this is to increase the red components of the vertices which make up the cheek geometry (see figure 9). To support per vertex color, the *model* class provides storage for one color per vertex. Only one color array is provided because OpenGL provides only one color vertex array. This color array may be used to set any one of the material colors, or to set both diffuse and ambient colors.



Figure 9: Vertex colors make cheeks blush and give the eyes pupils.

4.2.6 Texture Mapping

Specifying material colors alone is often insufficient in giving a realistic representation of an object. In the real world, objects may have an essentially infinite level of detail. To specify colors at higher levels of detail, it is not sufficient to specify them per vertex. Colors must be specified between vertices. Visual detail is added to an object via texturing. OpenGL provides many options for texturing an object. Texturing is done using texture maps, which in OpenGL are one, two, or three dimensional images. One dimensional textures are quite limiting. Three dimensional (volume) textures tend to significantly increase render times. For most purposes, two dimensional (2D) maps are sufficient, and only they are supported for this project. A *texture* class (see Appendix E.8) was created to read and store 2D texture maps. The *model* class provides access to two types of texture maps, each of which is applied to surfaces in a specific way.

4.2.6.1 Basic Color Mapping

A basic color map may be used, which is a texture that gets mapped onto surfaces via 2D texture coordinates. For each vertex, a texture coordinate specifies the location in the texture map corresponding to that vertex (see figures 10a, 10b). For each pixel, a texture color is calculated by interpolating between texture coordinates of surrounding vertices (see figure 10c). After lighting calculations, the color of each pixel is multiplied by the pixel's color map value.

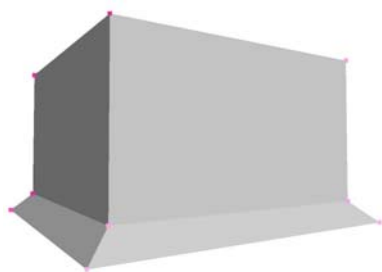


Figure 10a: Blank geometry.

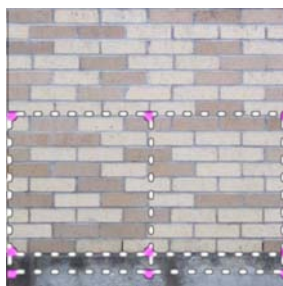


Figure 10b: Vertex texture coordinates.



Figure 10c: Textured geometry.

4.2.6.2 Reflection Mapping

A reflection map may be used to simulate surfaces reflecting their environment (see figure 11a). The reflection map is interpreted as a spherical environment map, which looks like a perfectly reflective sphere (see figure 11b). Take a ray from the viewer and reflect it off the object's surface at each vertex. The angle of reflection is determined by the surface normal. The direction of the reflected ray determines the reflection map coordinates. These coordinates change as the orientation of the viewer changes with respect to the object, and they are generated automatically by OpenGL. Spherical environment maps are usually created by making a picture of a highly reflective sphere. This may be accomplished with 3D rendering software or with an actual camera and reflective ball.

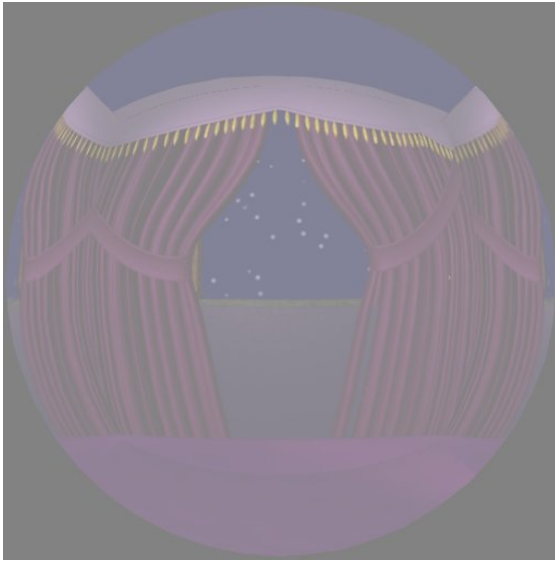


Figure 11a: Spherical environment map.

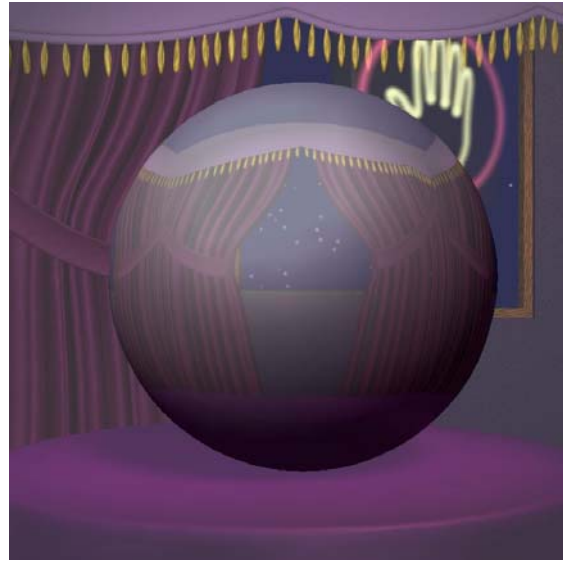


Figure 11b: Sphere with the environment map applied.

4.2.7 Object Transformations

Geometry is considered to be part of an object. It follows then that the geometry of each object is specified in its local coordinates. The local (object) coordinate system is relative to a global (world) coordinate system. The *model* class stores the following properties which define an object coordinate system: position, rotation, and scale. *Position* is simply an X-Y-Z point specifying the object's location in the global coordinate system. *Rotation* is actually three rotation angles, one for each axis (X, Y, Z), which specify the object's orientation in the global coordinate system. *Scale* is a number used to scale the object along all three local axes. *Scale* determines the object's size in the global coordinate system.

4.2.8 Object Parenting

While it is often sufficient to specify an object in world coordinates, it is often useful to specify a model relative to another model's coordinate system. This is called object parenting. For instance, if an arm is represented, a hand object could be parented to a forearm object, which would be parented to an upper-arm object, which would be relative to the world (see figure 12). So when the upper-arm rotates, the forearm and hand rotate accordingly. When the forearm rotates, it does so relative to the upper-arm, and the hand rotates accordingly. An object can only be transformed relative to one coordinate system, so an object may only have one parent. However, an object may have any number of "children." The hand, for example, might have five children which are its fingers. Each finger has only one parent, the hand (see figure 12). The *model* class supports parenting.

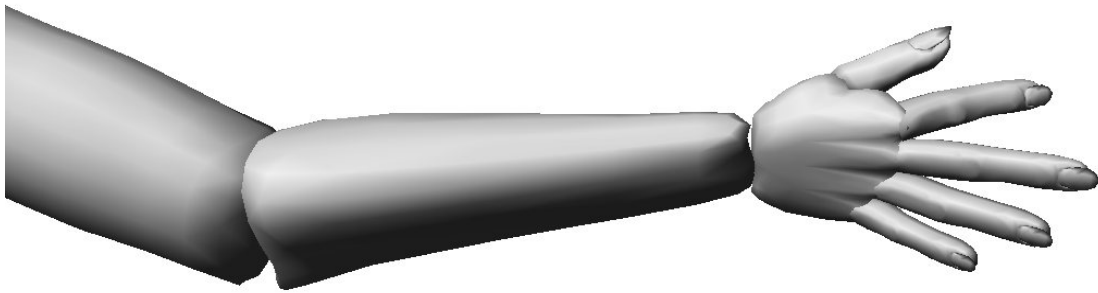


Figure 12: The upper arm is the parent of the forearm, which is the parent of the hand, which is the parent of the thumb and fingers.

4.2.9 Object Instancing

Geometry, which as previously stated is considered to be part of an object, is not necessarily unique for each object. For instance, consider a dozen eggs. Each egg is an object, meaning that it exists in its own coordinate system. Each egg certainly has a different global position, and differs a bit from the other eggs in orientation and scale (see figure 13). But it is reasonable to say that all eggs have essentially the same shape. For rendering to be efficient, each egg object can be stored as an instance of the same geometry. In the *model* class, instances share all vertex and face data. This includes vertex coordinates, normals, vertex colors, texture coordinates, lines, and faces.

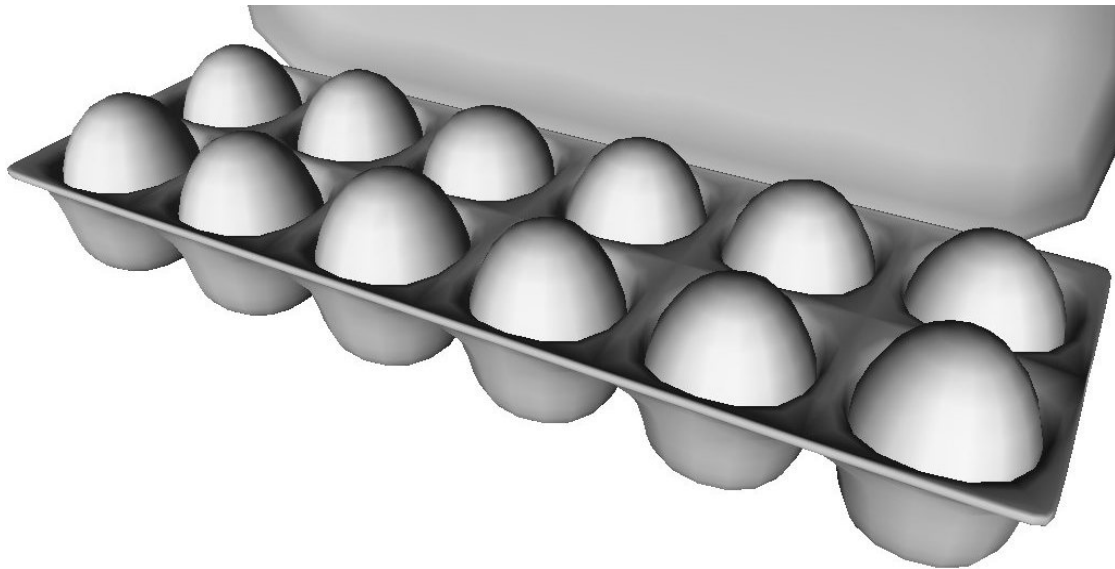


Figure 13: Each egg is an instance of the same geometry but has an independent location, orientation, and scale.

4.2.10 Object Files

The *model* class provides methods which read object models from files and write stored models to files. Initially, the Wavefront OBJ file format was chosen for its simplicity in storing geometry, for its readability, and for its compatibility with modeling software. Since some properties stored by the *model* class are not supported by the OBJ format, the format was extended into an “Extended OBJ” format, while keeping backwards compatibility with the original OBJ file specification. The format was extended to support parenting, instancing, and multiple texture maps. Backwards compatibility means that the *model* class will read the original OBJ format. Also, other software that reads OBJ files can also read “Extended OBJ” files if it simply ignores tags that are not part of the OBJ format. The *model* class does not interpret an “Extended OBJ” file as defining multiple objects, so each object file should only contain one object (see Appendix C.2). NOTE: While the *model* class stores material properties, they are not part of the “Extended OBJ” format.

4.2.11 Object Display

To properly display an object with the *model* class, the following must occur:

- 1) The proper OpenGL vertex arrays must be enabled, and corresponding array pointers must be assigned. These arrays include a vertex coordinate array, a surface normal array, a vertex color array, and a texture coordinate array.
- 2) The OpenGL material parameters must be set to the object’s material parameters. These are the object’s material colors and shininess.
- 3) The proper OpenGL texture objects must be selected. There may be a texture object for a color map and/or a reflection map.
- 4) Coordinate transformations must be done to position, orient, and scale the object relative to its parent coordinate system.
- 5) The primitives which constitute the object’s geometry must be drawn.

The *model* class provides methods to accomplish each of the above individually, and a *display* method which accomplishes all of the above. The *display* method is recursive. It will call itself once for each child of the current object. This means that when the *display* method is called with one *model* object, that object and all of its children will be displayed.

4.3 Scene Class

To represent a believable world, it is not only necessary to have detailed objects, but to have many of them. To facilitate environments having many objects and many lights, and to store the other properties necessary to specify a virtual world, a *scene* class was created. The *scene* class (see Appendix E.5) can store many objects, limited only by available memory. It can also store the maximum number of lights as specified by the OpenGL implementation. It can also maintain a texture list, which allows objects to share textures. A terrain list, which will be explained later, is also provided.

4.3.1 Camera Structure, Viewing Frustum

The *scene* class provides access to a *camera* structure (see Appendix F.1) which stores attributes that define the viewer within the virtual world. A *camera* has location and orientation. Changing the location and orientation of the *camera* based on user input allows the user to navigate the virtual world. A *camera* has a field-of-view angle. This can be thought of as a zoom factor, where narrow angles appear zoomed in (see figure 14a), and wide angles appear zoomed out (see figure 14b).



Figure 14a: Angle of view is small (zoomed in).

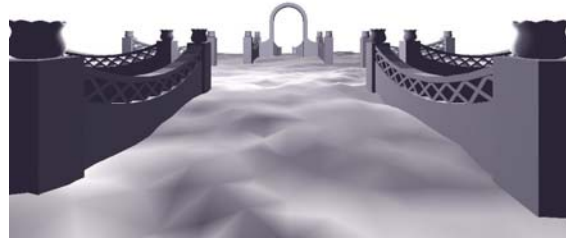


Figure 14b: Angle of view is large (zoomed out).

The *camera* structure also provides near and far clipping distances. Collectively, the field-of-view, near clipping distance, and far clipping distance define a viewing frustum (see figure 15). OpenGL only renders geometry within the defined viewing frustum. All geometry outside the frustum is clipped. A frustum is a shape like a truncated pyramid. The viewer (camera) is located at the apex of the pyramid. What the viewer sees is within the frustum. The distance from the viewer to the top of the frustum is the near clipping distance. The distance from the viewer to the bottom of the frustum is the far clipping distance.

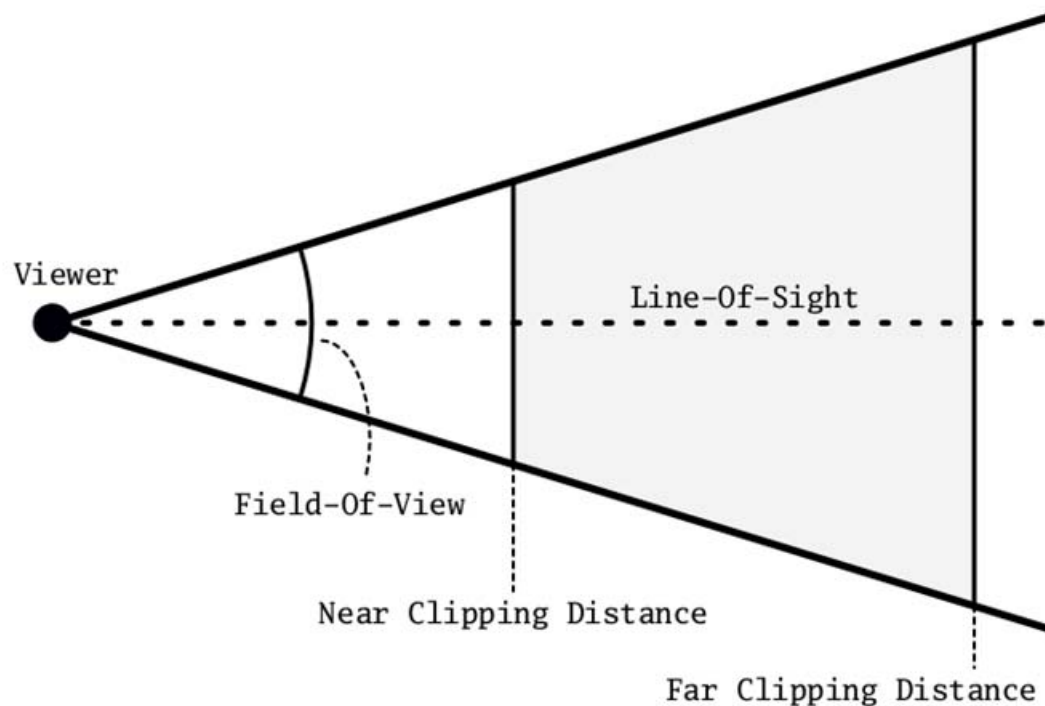


Figure 15: Symmetric viewing frustum.

4.3.2 Background Color, Image and Skybox

The *scene* class provides three ways of determining what is rendered behind all of the specified geometry. By default, the background is rendered black. The simplest way to change this is to set a different background color. Rather than a constant color, a background image may be loaded. However, this is not realistic, as the background image does not change as viewer orientation changes. The most realistic way to represent a background environment is to use a skybox. The *scene* class provides methods for loading and configuring a background image and a skybox.

A skybox is a cube that is rendered behind (surrounding) all other geometry. Each side of the cube is textured with an image. These environment images are created from the same point of view, looking in the six axial directions, each having a field-of-view of ninety degrees. The edges of the images should match, so the result is a contiguous backdrop which always appears infinitely far away.

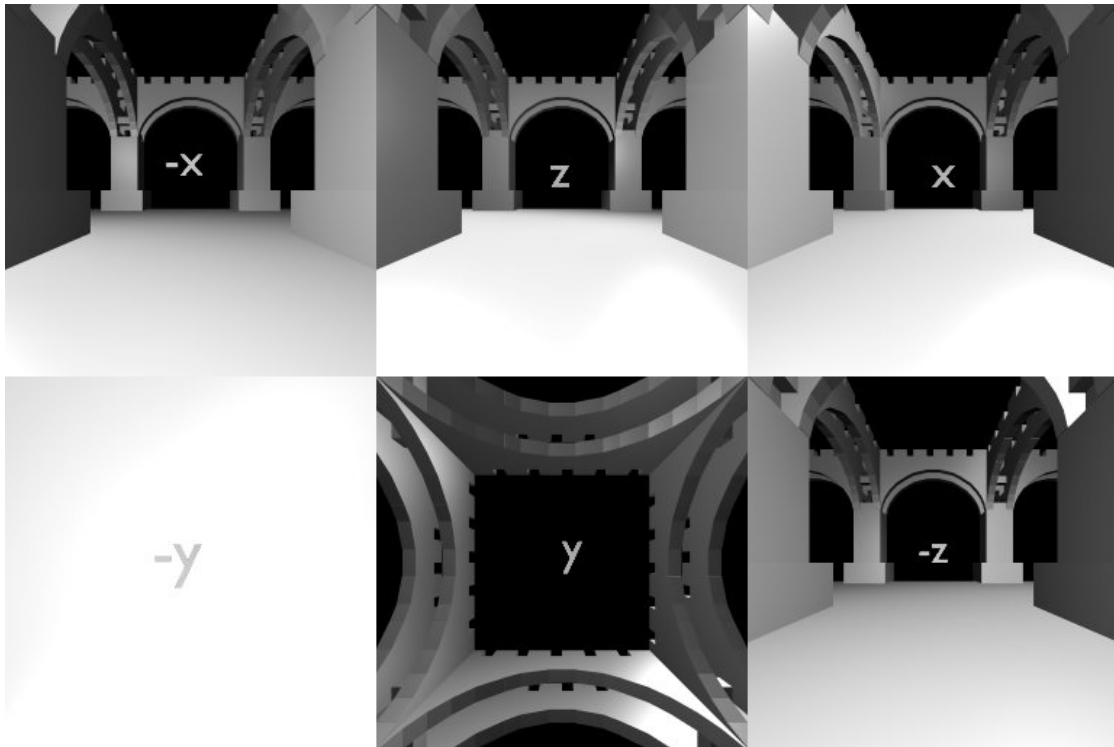


Figure 16: A skybox image is six images in one, each looking positive or negative along a Cartesian axis.

A skybox image is six images in one. Each image is a view along a Cartesian axis. Three views are along the axes in positive directions, and three are along the axes in negative directions. The skybox setup function expects these images to be oriented and ordered as shown in figure 16. The order is $-X$, $+Z$, $+X$ on top, and $-Y$, $+Y$, $-Z$ on bottom. The skybox is rendered such that these images are oriented along their respective axes in world coordinates.

4.3.3 Fog

The real world cannot be fully represented by objects, light sources, and nice backgrounds alone. Atmosphere plays a part as well. Fog or haze may cause a lack of transparency of the air. OpenGL provides a simple method for simulating fog, where each pixel color is combined with a fog color. The resulting amount of original pixel color versus fog color is determined by applying a fog function to the pixel's depth, which is stored in the depth buffer. The further away the pixel is, the more it will take on the color of the fog. The fog function may be linear, exponential, or exponential squared. With the linear fog function, the amount of fog color a pixel has is directly proportional to its depth (see figure 17a). With the exponential functions, the pixels do not take on the fog color at a constant rate. The resulting effect is that pixels appear to be enveloped in fog quicker than they do with the linear function.

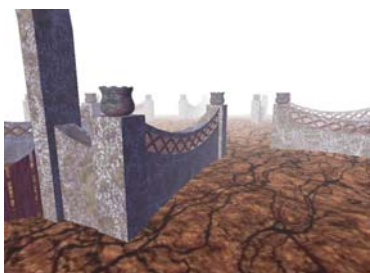


Figure 17a: Linear function.

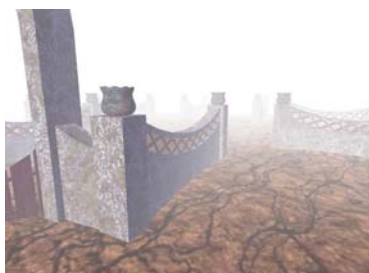


Figure 17b: Exponential function at 0.04 density.



Figure 17c: Exponential function at 0.1 density.

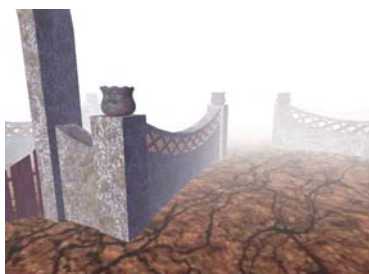


Figure 17d: Exponential squared at 0.04 density.



Figure 17e: Exponential squared at 0.1 density.

The *scene* class provides access to a *fog_state* (see Appendix F.2) structure, which stores the fog color, the distance from the viewer at which the fog function takes effect, the corresponding ending distance, the density of the fog, and the fog function. The density affects the steepness of the exponential functions, and therefore how quickly the fog “thickens” over the given distance (see figures 17b, 17c, 17d, and 17e).

4.3.4 Window Structure

The *scene* class provides access to a *window* structure (see Appendix F.3) which specifies what “window” the virtual world is rendered into. The *window* structure stores the viewport, which is the screen coordinate rectangle that OpenGL renders within. It also stores the display mode, which specifies what types of image buffers are used. Commonly, OpenGL programs render into an

RGBA color buffer and use a depth buffer (*z buffer*) for depth testing. Depth testing ensures that the correct geometry appears in front when geometry overlaps. Scenes such as those we are considering involve a lot of motion and should use double-buffering. With double-buffering, drawing is done to a “back” buffer. After the scene is drawn the buffers are swapped. The contents of the back buffer become the contents of the front buffer. The front buffer is the one displayed.

4.3.5 Scene Files

The *scene* class provides a method for reading scene data from a file. Object files to be imported may be specified in scene files. Variables for material properties may be set within scene files. Current (last read) material properties are applied to objects when they are loaded. Lights may be specified in scene files. Camera, background, fog, and window parameters may be specified within scene files.

As elements of the virtual world are imported by the *read* method they are stored within the *scene* object. Lights are imported into a *light* array. *Model* objects are added to a *model* list. *Terrain* objects are added to a *terrain* list. *Model* object textures are added to a *texture* list. The application from which the *read* method was called may then traverse each array. Changes may be made to the data, and element *display* methods may be called to draw the virtual world.

4.4 Cave Class

In this context, the term *cave* does not refer to the CAVEtm display developed at the University of Illinois. *Cave* is being used here in a more general sense to refer to polyhedral immersive display geometry and the *cave* class (see Appendix E.1) that has been developed to support it. After developing initial software that rendered a virtual world for a standard rectangular display, a *cave*

class was created to extend the software to render a virtual world as it appears for a particular specified facet of a polyhedral spatially immersive display (SID). To accomplish this, the viewer is rotated such that the viewer's line-of-sight is parallel to the normal vector of the specified facet. By default, the viewer is also rotated about the line-of-sight so that the longest side of the facet appears along the bottom of the viewport. For some geometry, this helps to maximize the facet area in the viewport. A viewing frustum is created such that the near clipping plane is coincident with the facet. The cave geometry is used to determine near clipping distances. No geometry inside the display geometry will be visible. By default the facet is rendered to the screen in a way that maximizes its screen area while maintaining its aspect ratio. Everything in the viewport that is inside the facet is rendered normally. Everything in the viewport that is outside the facet boundary is rendered as black. Display parameter defaults may be changed via the adjustments described in Section 4.4.2.

4.4.1 Cave Geometry

Geometry for the desired polyhedral display is imported as an OBJ file via the *model* class. This file is created using external software. Like any *model* class object, the display system geometry may consist of faces (facets) having any number of sides. Technically, this geometry may be anything, but limiting it to a convex polyhedron (see figures 18a, 18b, and 18c) with all facets facing inward ensures that each facet is visible from any position within the SID. If this limitation is not met, the rendered image for a display facet whose front (as described in Section 4.2.1) is not visible to the viewer is undefined.

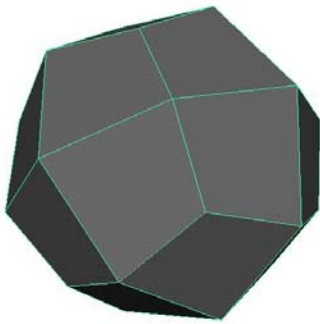


Figure 18a: Trapezoidal Icositetrahedron.

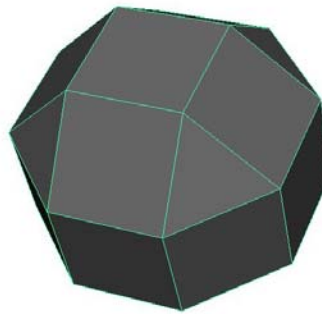


Figure 18b: Rhombicuboctahedron.

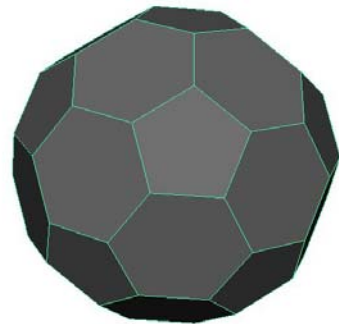


Figure 18c: Truncated Icosahedron (soccer ball).

4.4.2 Display Adjustment and Cave Files

The *cave* class has variables which are adjusted to accommodate different projectors and screens. The current facet index is set. The side of the facet that is oriented along the bottom of the screen is specified. The output may be scaled (see figure 19a) horizontally and vertically. It may also be skewed (see figure 19b) horizontally and vertically. The output may also be rotated. Each of these attributes and the name of the OBJ file representing the display geometry may be specified in a *cave* file. The *cave* class provides a method for reading *cave* files (see Appendix C.1).

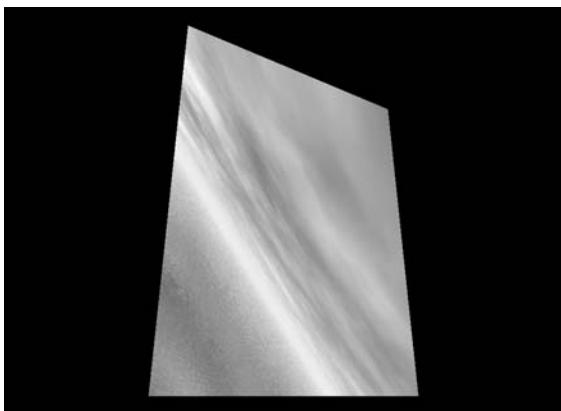


Figure 19a: Scaled facet.

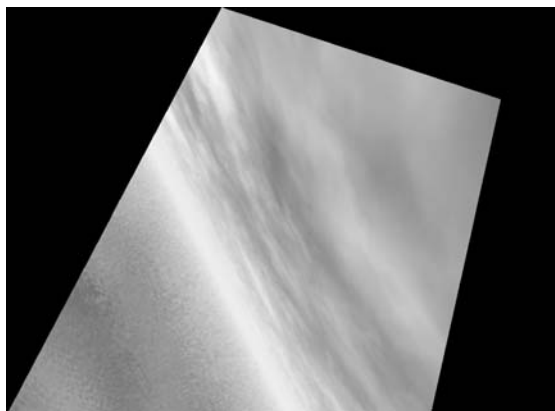


Figure 19b: Horizontally skewed facet.

4.4.3 Update Viewer Transforms and Projection Matrix

Interactive viewer tracking is supported, in that the viewer position and orientation relative to the display geometry may be changed by the application for each frame time. Because of this, some of the *cave* class variables are recalculated every frame via the *update* method. The default viewer position is the origin of the display geometry. For stereo viewing, the viewer position is offset

for each eye. For each new frame time, the actions described in the following lists occur.

If the facet index has changed from the previous frame:

1. The number of edges of the facet is determined.
2. New “working” vertices are allocated. Working vertices are a copy of the facet vertices that will be transformed into screen space and used to set up the perspective matrix.
3. The “base” facet edge is set to be the longest facet edge.

Regardless of a change in the facet index, the following actions are done for each frame:

1. The current facet edge is set to be the base facet side plus the current side index. This is the facet edge which by default is rendered along the bottom of the viewport. See figure 20.

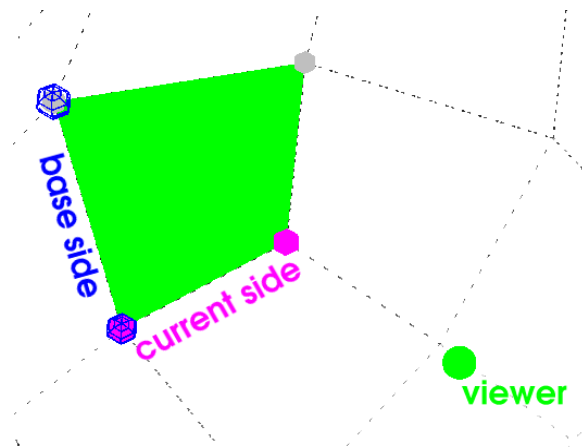


Figure 20: The **current side** is the **base side** plus an index. In this case the index is one.

2. The working vertices are set to be the facet vertices relative to the viewer position. See figure 20.

3. Using a ray from the viewer position in the direction opposite to the facet normal vector, the point where the ray collides with the facet, **collpos**, is calculated. See figure 21.

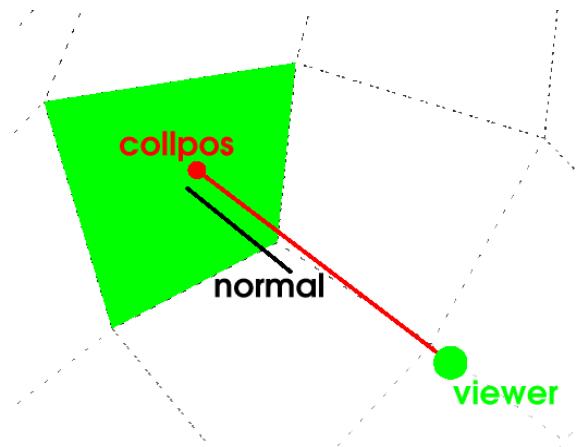


Figure 21: A ray is shot from the **viewer** in the direction opposite the facet **normal** until it collides with the facet. Call that **collpos**.

4. A rotation axis vector **rotaxis** is calculated to be orthogonal to the collision vector and the **forward** $[0,0,-1]$ vector (right handed world coordinates). If **rotaxis** is invalid (not-a-number), which will occur if the collision ray (described above) is parallel to the **forward** vector, it is set to the **upward** $[0,1,0]$ vector (world coordinates). See figure 22.
5. **rotation₀** is the angle between the negated facet normal and the **forward** vector. See figure 22.

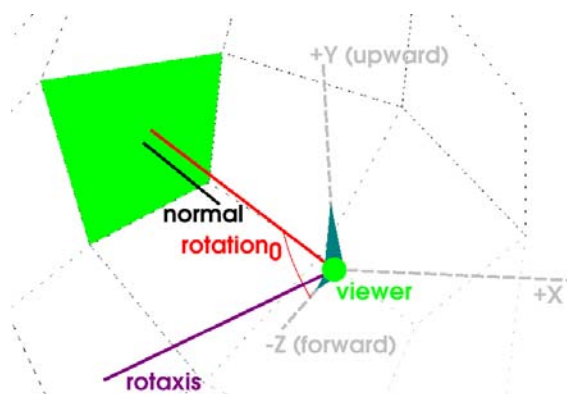


Figure 22: **rotaxis** is orthogonal to vector **[collpos – viewer]** and **forward**. **rotation₀** is the angle between the negated **normal** (collision direction) and **forward**.

6. **upvector₀** is the **upward** vector rotated through **rotation₀** about **rotaxis**. See figure 23.

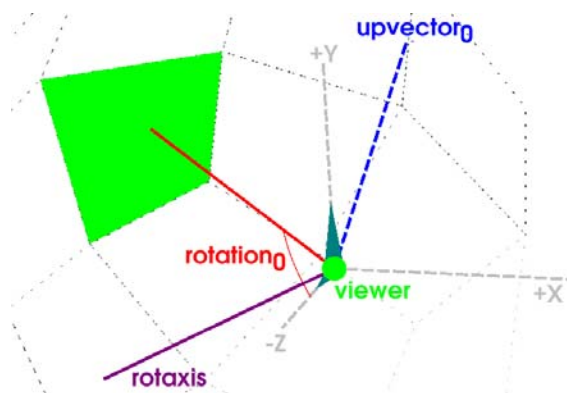


Figure 23: **upward** is rotated about **rotaxis** through **rotation₀** to determine **upvector₀**.

7. **ortho** is a vector orthogonal to the current facet edge and the facet normal. See figure 24.

8. **rotation₁** is the angle between **upvector₀** and **ortho**, plus an optional cave class **Z-rotation** angle. See figure 24.
9. **upvector₁** is set to **upvector₀** rotated through **rotation₁** about the facet normal direction. If **upvector₁** is not parallel to **ortho**, **rotation₁** is negated and **upvector₁** is recalculated. See figure 24.

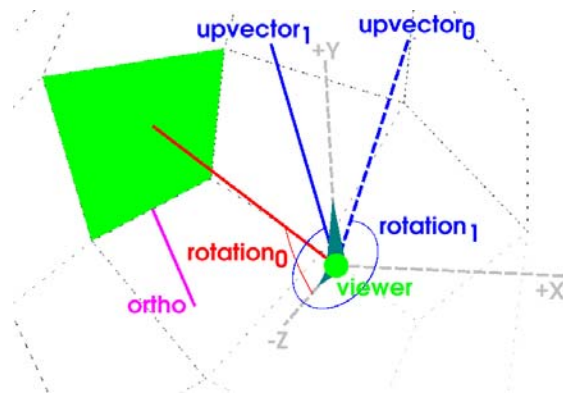


Figure 24: **upvector₀** is rotated about the **normal** direction through **rotation₁** to determine **upvector₁**.

10. The working vertices are rotated by **rotation₁** about the facet normal and by **rotation₀** about **rotaxis**. This results in the facet defined by the working vertices being co-planar to the display window. See figure 25.

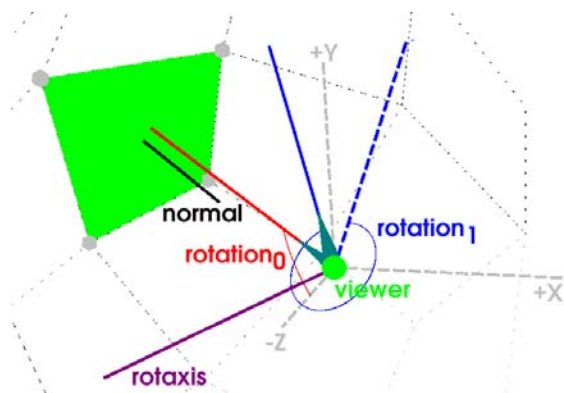


Figure 25: Facet vertices are rotated by **rotation₀** about **rotaxis** and by **rotation₁** about the **normal** direction. Conversely, the **viewer** is depicted as rotated relative to the facet.

If *cave mode* is on, then...

11. The near clipping distance in the *camera* structure is set to the distance from the viewer to **collpos**. See figure 26.

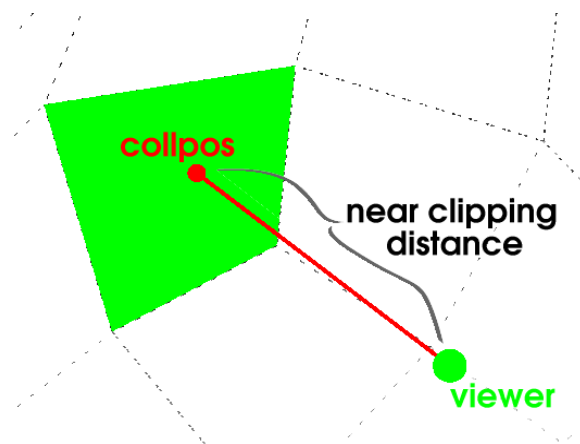


Figure 26: Near clipping distance is the distance from the viewer to the facet in the **normal** direction (collision ray).

12. The variables which define the viewing frustum are set so that the entire facet will be visible on screen, and that it will utilize the maximum

available screen area, while keeping the facet's aspect ratio relative to the viewport aspect ratio. The current near and far clipping distances are used.

13. The frustum is scaled horizontally and vertically with the current scaling values.

Set the projection matrix...

14. The projection matrix is flipped horizontally to accommodate back-projection.
15. The projection matrix is skewed via a skew matrix which is generated from horizontal and vertical skew factors.

4.4.4 Environment Transformations

For a standard rectangular display, the *camera* structure (Section 4.3.1) is used to position and orient the viewer relative to the virtual world. When generating images for a specified display (cave mode), the *camera* is used to position and orient the display geometry relative to the virtual world. Then the viewer is translated and rotated relative to the display geometry. The *translation* method translates the viewer relative to the origin of the display geometry. The *rotation* method rotates the viewer as described in Section 4.4.3.

4.4.5 Drawing Borders

After transformations are applied and the projection matrix is set, the world that is visible through the facet is drawn, and the rest of the viewport is black. To accomplish this, two approaches were tested. One approach uses arbitrary clipping planes. Clipping planes were defined so that only the geometry visible through the facet was drawn, and everything outside the facet boundary was clipped. While accomplishing the goal, this resulted in significantly increased render times and unacceptable rendering artifacts at the facet boundary. In the second, preferred approach, the virtual world is drawn to the entire viewport.

Then a black border is drawn to blacken all pixels outside the facet boundary.

4.4.6 Visual Aids

To understand how the viewer is oriented with respect to the display geometry, especially to see which facet the viewer is looking through, it is helpful to view the display geometry from outside. To support this, the *cave* class includes a *testing* method which draws the following visual references: the display geometry, the current facet, the facet normal, a cone representing the viewer, and the vectors that are set by the *update* method. This *testing* method is used to visualize the viewer relative to the *cave* geometry while viewing them within the virtual world rendered on a standard rectangular display.

4.5 Networking

To support interactive changes in position and orientation of the viewer, including position relative to the world and position relative to the display, data is communicated between graphics nodes which are connected via a local area network (LAN). When the software is initiated, one node is designated the client and is given the network hostname or IP address of each of the other graphics nodes (see figure 27). All non-client nodes are designated servers, and every frame time they receive packets of navigational data from the client. The client sends a data packet to each server and waits for a response from each server before it continues. The response from the servers is a data packet containing a frame counter. If a server's frame counter does not match the client's frame counter, the client generates an error. This insures that all node displays are never more than one frame time out of synchronization.

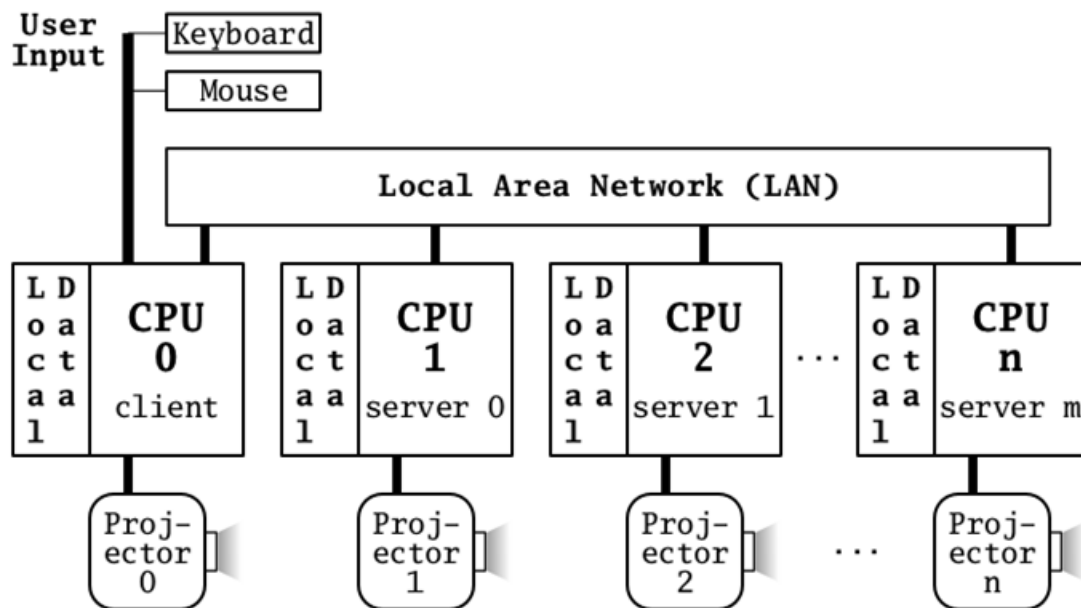


Figure 27: On the TAMU system, each machine has an identical copy of the environment. The client sends navigational and frame number data to the servers every frame time.

The servers expect data packets to be of a specific size. If a server receives a packet of an incorrect size, it dies. So when the client exits, it kills all servers by sending them packets which are smaller than the expected packet size.

The network code does not exist as a network class or library. Due to its simplicity, it has been implemented within the main application code. The network code may be changed or removed by the application developer.

4.6 Terrain Class

Initially, viewer navigation was very simple. The viewer was allowed to walk (translate) on a plane and rotate, which gives the impression of floating and looking in controllable directions while staying at a fixed height. To facilitate applications in which the user is given the impression of moving across an uneven surface, a *terrain* class (see Appendix E.7) was created. It allows a grayscale image to be imported and used as a height map. A height map is an image whose pixel values are used to determine surface elevation (see figure 28a). Lighter pixels (larger values) correspond to greater height. A square surface grid, having one vertex for each pixel in the height map, is generated. Each pixel of the image is multiplied by a height factor to determine the height (Y-location) of its corresponding vertex of the grid (see figure 28b). As the viewer walks/moves across the grid, the vertices of the grid cell that the viewer is above are interpolated with respect to the horizontal viewer position and a new height is calculated. The viewer position is set based on this *terrain* height, so the viewer moves up and down following the terrain elevation.

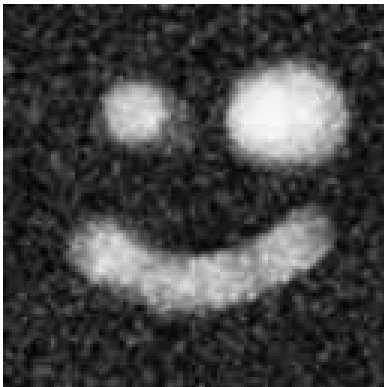


Figure 28a: Height map

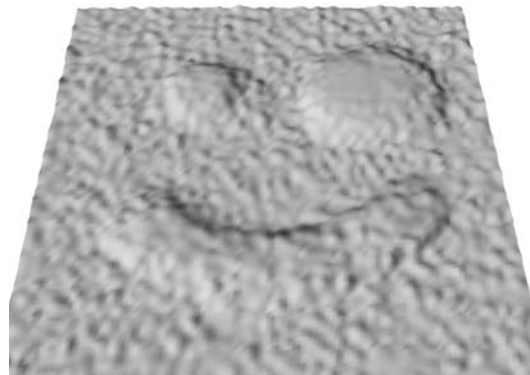


Figure 28b: Terrain generated from height map.

5 RESULTS

Two applications have been developed which make use of the software framework of libraries and network code described above. A demonstration application has been in constant development. Its purpose is to thoroughly test each portion of code that has been developed for this project. A second application was developed to provide specific interaction between the user and the environment.

5.1 Cave Demo Application

The Cave Demo application has been updated throughout the development of the software. With the addition of each object class or data structure, the application has been updated to test the new code. The Demo application also provides display-specific networking as described in Section 4.5, and supports anaglyphic stereo capability.

This application is able to read and display all of the defined elements of a sample virtual environment. At start-up, it processes command-line arguments and reads all valid specified files. For each OBJ (or Extended OBJ) file, it calls the *read* method of the *model* class. For each scene file, it calls the *read* method of the *scene* class. And for each cave file, it calls the *read* method of the *cave* class. All *model* objects are subsequently organized into a hierarchy determined by parent-child relationships. This allows each object to be transformed by the coordinate system of its parent. At each frame time, the sample application performs the following display tasks:

1. Draw the background color, background image, or skybox.
2. Information is updated for each enabled light in the *light* array. This means that all enabled lights are applied to all geometry in the virtual world.
3. The *display* method of the *model* class is called for each object at the

base of the *model* hierarchy; that is, each object that has no parent. Since the *display* method is applied recursively to parented objects, all objects in the hierarchy are drawn.

4. The *display* method of the *terrain* class is called for each object in the *terrain* list.

The Demo application allows the user to navigate the virtual environment using a mouse and/or keyboard. Pressing mouse buttons or keyboard arrow keys translates the viewer in left, right, forward, and backward directions (NOTE: To work properly, this requires a three-button mouse). Two buttons or keys may be held simultaneously to move the user diagonally. Moving the mouse changes the orientation of the viewer. Forward and backward mouse movements tilt the viewer down and up, respectively. Left and right mouse movement pan the viewer. The user may navigate vertically with the plus and minus keys on the number pad of the keyboard. However, any change in vertical location (Z-position) of the viewer is lost when the viewer is moved horizontally if its height is being determined by a *terrain*.

The Demo application provides anaglyphic stereo rendering. Two viewer eye positions are defined, equidistant to the left and right of the non-stereo viewer position. All geometry in the virtual world except for the background is drawn twice, once for each eye position. Drawing for the left eye position is masked so that only the red color channel is drawn to. Drawing for the right eye position is masked so that only the cyan (blue and green) color channels are drawn to. An effect of depth is achieved when this anaglyphic rendering is viewed with red-cyan filtered glasses.

The Demo application includes keyboard commands which allow the user to change various display parameters. These changes include toggling surface

normal display, changing drawing modes (filled polygons, wireframe, vertex points), turning off smooth shading, toggling lighting, etc. When generating images which conform to a specified display facet (cave mode), commands are available to scale and skew output, change facets, etc. Commands are available for toggling anaglyphic stereo display mode, and for setting the distance of the stereo eye positions from the standard viewer position. These and other commands are documented in Appendix B.

The Demo application can be run in standard rectangular display mode, or in cave display mode. “Cave mode” means that the display geometry is determined by input from an OBJ file, which is specified within a cave file. The application may be run on a single machine, or on a network of graphics nodes (see figure 26).

Several object models and scene files have been created to test this application. In addition, two cave files for two different display facet geometries have been tested. The first test was on a three-facet section of a surrounding twenty-four facet polyhedron. The second test was on a three-facet section of a ten-facet surrounding “cylindrical” prism.

5.2 Bubble Factory Project

The Bubble Factory Project was created for the Texas A&M Visualization program’s “Time Based Media” course. The goal was to create an underwater environment containing machinery that a user could interact with. Development for the Bubble Factory Project began with specific goals for user interaction that are more complex than what the Demo application provides. The user input was to seem as “hands-on” as possible using a mouse and keyboard. This meant that if a user were to “walk” up to a machine with a handle, he could then move the mouse in a forward (up) motion, and then in a backward (down) motion to

simulate a hand reaching up and pulling down. The grasping and releasing of the handle by the hand might be accomplished by pressing and releasing a button.

The project code began with a copy of the then current Cave Demo Application. So the Bubble Factory application incorporated most of the functionality described in Section 5.1. It loads lights and geometry via scene files and OBJ files. It goes through display steps which are similar to those of the Demo application: draw the background, update OpenGL for each light, and draw each *model* object.

One of the requirements of an environment with interactive machinery was that pieces of machines needed to move relative to other pieces. Having everything in global coordinates was insufficient. This was the reason for upgrading the *model* class to support parenting where each object may have any number of children, and where parent-child hierarchies may be arbitrarily deep.

An aesthetic requirement of the Bubble Factory Project was that the user feels as if she were walking on a rugged surface. The environment was to have a hilly terrain, so the viewer would not be limited to a constant elevation. The simplistic navigation of the Demo application did not provide this kind of adaptation to landscape, so the *terrain* class was created. The Bubble Plant application, and eventually the Demo application were updated to calculate the viewer height relative to a *terrain*, as described in the Terrain Class section.

It was observed that some machinery has many parts which have essentially the same geometry, such as bolts and rods. To make the storage and display of such objects more efficient, the *model* class was updated to allow object instancing. The Bubble Factory itself is a building in the virtual environment

which has a control rod. When the handle of the control rod is “pulled down,” the Factory generates a bubble which floats out of a stack on top. The application stores one default bubble object. Each time the Bubble Factory’s control handle is pulled down, a new instance of the default bubble object is generated.

The Bubble Factory Project was not designed to make use of multiple graphics nodes. It may be run over a network because it incorporates the same networking capability as the sample application. This means that user navigational data is shared across the network, but updates to the virtual environment only occur on one machine, the client. Future work might include support for updating copies of the virtual environment across the network.

6 CONCLUSION AND FUTURE DIRECTIONS

The goals of this project have been accomplished. Using the developed libraries, applications were created that generate images that are properly conformed for projection onto facets of polyhedral spatially immersive displays. The software supports interactive changes in viewer position and orientation. It supports the networking and synchronization of an arbitrary number of graphics nodes. Anaglyphic stereo viewing was added to the Cave Demo application. Detailed documentation was written, and is available as appendices to this document.

The work done in this thesis could be extended as follows:

- The developed software framework could be used to support a variety of applications. Feedback from this would then determine what changes and additions should be made to support a wider range of applications.
- A better approach could be devised for calibrating and adjusting the display parameters to compensate for non-linearity and distortion in projection systems.
- Support interactive manipulation of the virtual environment. This may include allowing the user to interactively edit geometry, change shading properties, change texturing, change lighting properties, adjust fog attributes, or make any other changes to the virtual environment which need to be consistent across all of the networked nodes.
- Provide support for active, time-sequential stereo viewing. This might be accomplished by updating the software to generate images for left-eye and right-eye sequentially, and by viewing the display through shutter glasses which are synchronized with the display refresh. This would be difficult if not impossible to achieve with multiple displays, as the refresh of each projector would need to be synchronized.

- Support passive stereo viewing. This could be accomplished by using two projectors per facet with polarizing filters, or by using one projector with a time sequencing polarizing filter per facet. Resulting images would be viewed through polarized glasses.
- Extend support for a wider range of interactive devices such as joysticks/gamepads. This would involve upgrading the software to determine the type of interactive devices being used and act accordingly.
- Support user motion tracking using systems such as the Ascension Flock Of Birds [6]. Utilizing a motion tracking system would allow for interactive changes of the user position and/or orientation in relation to the display geometry. The result of user position tracking would be geometrically correct projection of the scene geometry into an image for each facet, regardless of the user's location within the SID space. The result of orientation tracking would be correctly generated stereographic images, regardless of the direction the user is looking.
- Implement support for physically-based simulation. Interaction with an environment, such as hitting walls, falling off edges, or doing anything which causes an object to react in a physically-based way, help the user feel as if he is in a tangible world.
- Provide more robust network error handling.
- Support cast shadows.

REFERENCES

- [1] *CAVELib Users Manual*. VRCO Inc., October 1999.
<http://www.gup.uni-linz.ac.at/vrcave/CAVELibManual.pdf>

- [2] Humphreys, G., I. Buck, M. Eldridge and P. Hanrahan. "Distributed Rendering for Scalable Displays," *Proceedings SC2000: High Performance Networking and Computing*, pp. 60, November 2000.

- [3] Humphreys, G., M. Eldridge, I. Buck, G. Stoll, M. Everett and P. Hanrahan. "WireGL: A Scalable Graphics System for Clusters," *Proceedings SIGGRAPH 2001*, pp. 129—140, August 2001.

- [4] Humphreys, G. and P. Hanrahan. "A Distributed Graphics System for Large Tiled Displays," *Proceedings IEEE Visualization*, pp. 215—224, October 1999.

- [5] Humphreys, G., M. Houston, R. Ng, R. Frank, S. Ahern, P.D. Kirchner and J.T. Klosowski. "Chromium: A Stream-Processing Framework for Interactive Rendering on Clusters," *Proceedings SIGGRAPH 2002*, pp. 693-702, July 2002.

- [6] *Motion Star: Installation and Operations Guide*. Ascension Technology, 1996.

- [7] Parke, F.I. "Next Generation Immersive Visualization Environments," *Proceedings Sigradi 2002*, pp. 163—166, May 2002.

- [8] Plato. B. Jowett, Trans. *The Republic*, chapter 7, pp. 514a—520a, The Colonial Press, 1901.

- [9] Schönbrunner, O. "Human-Computer Interface in the CAVE," *Proceedings 4th Central European Seminar on Computer Graphics (CESCG)*, pp. 11—20, May 2000.

- [10] Woo, M., J. Neider, T. Davis and D. Shreiner. *OpenGL Programming Guide*, 3rd ed., Addison-Wesley, 1999.

APPENDIX A

INSTALLATION

There's not much as far as installation, but here are simple instructions on how to decompress and build the software, assuming you have a gzipped tar archive.

3Dengine (3Deng for Linux)
Author: Chris Anderson

BUILD INSTRUCTIONS

Decompress a gzipped archive...
extract(x), unzip(z), with verbose(v) output, file(f) matching
3Deng*.tar.gz:
> tar xzvf 3Deng*.tar.gz

Change to the base 3Dengine directory:
> cd 3Dengine

Remove old executables and objects:
> make clean

Compile code:
> make

If no errors occur during the make, a '3Deng' executable is created.

CHANGE APPLICATION

The main application code is pointed to by src/main.cpp. It is a symbolic link:

main.cpp -> ../app_src/cave.cpp

To change which application is compiled, change main.cpp:
> cd src
> ln -sf ../app_src/another_app.cpp main.cpp
> cd ..

Then recompile:
> make clean
> make

APPENDIX B

USAGE

Here is how to run the default cave application, including its various runtime commands.

```
3Deng [ --server | --server=hostname | --server=IP_address ] \
      [ cavefile(.cave) ] [ scenefiles(.sce) ] [ modelfiles(.obj) ]
```

DESCRIPTION

3Deng is a fullscreen OpenGL application. All of the inputs are optional. This document describes 3Deng when compiled as the Cave Demo application.

`--server` Indicates that the application is a server, which means that it receives navigational information from a client which is responsible for connecting to it over the network.

`--server=IP`

`--server=host` Indicates that the application is a client, which connects to a server at the given hostname or IP address. A client should have one of these parameters per server to which it connects.

`cavefile (.cave)` Cave configuration files are used to set cave attributes and to specify a cave model.

`modelfiles (.obj)` Model files specify models to be loaded into a scene, aside from those in the scene files.

`scenefiles (.sce)` Scene files specify a scene, including lights, models, and terrains. They can set the field-of-view, the viewport, the fog state, etc.

GENERAL RUNTIME COMMANDS

`Ctrl-c` toggle control mode

The idea of control mode is that there are two sets of input. One set is the input that sets display properties and provides display adjustment. The second set is the user input as specified by a particular application. When control mode is on, the following display adjustment commands are available. It is on/true by default.

```

c          toggle cave mode

          When cave mode is on/true, the scene is rendered from the
          point of view of the viewer looking through a cave facet.
          When cave mode is off/false, the scene is rendered for a
          standard rectangular display.

F          toggle fog
s          toggle smooth shading
n          toggle surface normal display
Ctrl-d     toggle light token display
Ctrl-l     toggle lighting
p          toggle pause
t          toggle stereo mode

f          draw surfaces as filled polygons (default)
P          draw surfaces as vertex points
w          draw surfaces as lines(wireframe)

arrow keys translate the viewer
mouse buttons translate the viewer
mouse movement rotates viewer

Esc        exit program
          --client exit invokes exit on each server

```

CAVE ADJUSTMENT

```

M < > ?    Xscale      (default = 1)
m , . /    Yscale      (default = 1)
K L : "    Xskew (default = 0)
k l ; '    Yskew (default = 0)

          For the above commands, the far left adjustments
          ( M M K k ) make large decrements. The near left commands
          ( < , L l ) make small decrements. The far right commands
          ( ? / " ' ) make large increments. The near right commands
          ( > . : ; ) make small increments.

PgUp       increment cave facet index number
PgDown     decrement cave facet index number

          Facet indeces are from zero to one less than the total
          number of facets.

S          increment base side of cave facet

Home       rotate long facet side -> bottom
End        rotate long facet side -> top (default)

```

STEREO ADJUSTMENT

{ [] } eye spacing

As with cave scaling and skewing above, the outer keys are large adjustments; the inner keys are small adjustments.

APPENDIX C

PROPRIETARY FILE FORMATS

C.1 Cave File Format

```
#comment

model model_file(.obj)  # cave geometry

facet[%u]                # index of facet
                        # Default: 0

side[%u]                 # index of side of facet
                        # Default: 0

Xscale[%f]               # horizontal field-of-view scale
                        # Default: 1

Yscale[%f]               # vertical field-of-view scale
                        # Default: 1

Xskew[%f]                # degree of horizontal skew
                        # Default: 0

Yskew[%f]                # degree of vertical skew
                        # Default: 0

Zrotation[%f]            # rotation about line-of-sight
                        # Default: 0
```

3D ENGINE NOTES

All of the above are optional. Multiple cave files may be used in succession. The only data in the cave geometry OBJ file that is used is the vertices and faces. No other transformation or display tags are considered. Zrotation cannot be set until a model has been specified. Cave files must end in ".cave".

C.2 OBJ Extended 3D File Format

```
#comment

v  %f %f %f # vertex coordinate
vt %f %f    # 2D texture coordinate
vn %f %f %f # vertex normal
vc %i %i %i %i    # RGBA vertex color

l v/vt/vn v/vt/vn
    Specify a line segment with vertex indices.
    Indices for vt and vn are optional.

f v/vt/vn v/vt/vn v/vt/vn [v/vt/vn]
    Specify a 3 or 4 vertex face with vertex indices.
    Indices for vt and vn are optional.

o object_name
parent parent_object_name
instanceof source_object_name

position %f %f %f
rotation %f %f %f
scale    %f %f %f

usemap texture_file [color|reflect|bump]
    If neither color, reflect, or bump is
    specified, texture maps should be
    interpreted in said order(first color,
    then reflect, then bump).
```

3D ENGINE NOTES

Textures may be raw PPM, raw PGM, JPEG, or TARGA(raw or RLE).
 Uniform scale only.
 Bump textures are not yet implemented.
 vc alpha values are not currently used.
 Model files must end in ".obj".

C.3 Scene File Format

```
#comment

viewport position[%d,%d] resolution[%d,%d]
projection fov[%f] nearclip[%f] farclip[%f]

eyespacing[%lf]          # stereo, distance between eyes
                          # Default: 0.04

bgcolor[%i,%i,%i]        # background color
background texture_file # background flat image
skybox 3X2_texture_file # six piece environment map

fog start[%f] end[%f] density[%f] color[%i,%i,%i] function[exp | exp2 |
linear]

mapmin[%f,%f,%f]         # minimum/maximum "geographic" location for
mapmax[%f,%f,%f]         # viewer position and light adjustment

position[%f,%f,%f]       # initial viewer position
rotation[%f,%f,%f]       # initial viewer orientation

positionchange[%f]       # viewer movement speed factor
rotationchange[%f]       # viewer rotation speed factor

light position[%f,%f,%f,%f] ambient[%i,%i,%i] diffuse[%i,%i,%i]
specular[%i,%i,%i]

# Material State #
emission[%i,%i,%i]
ambient[%i,%i,%i]
diffuse[%i,%i,%i]
specular[%i,%i,%i]
shininess[%f]

import model_file(.obj) # import a single model(.obj)
importdir directory    # import all model(.obj) files in directory
terrain gray_texture_file amplitude[%f] position[%f,%f,%f]
```

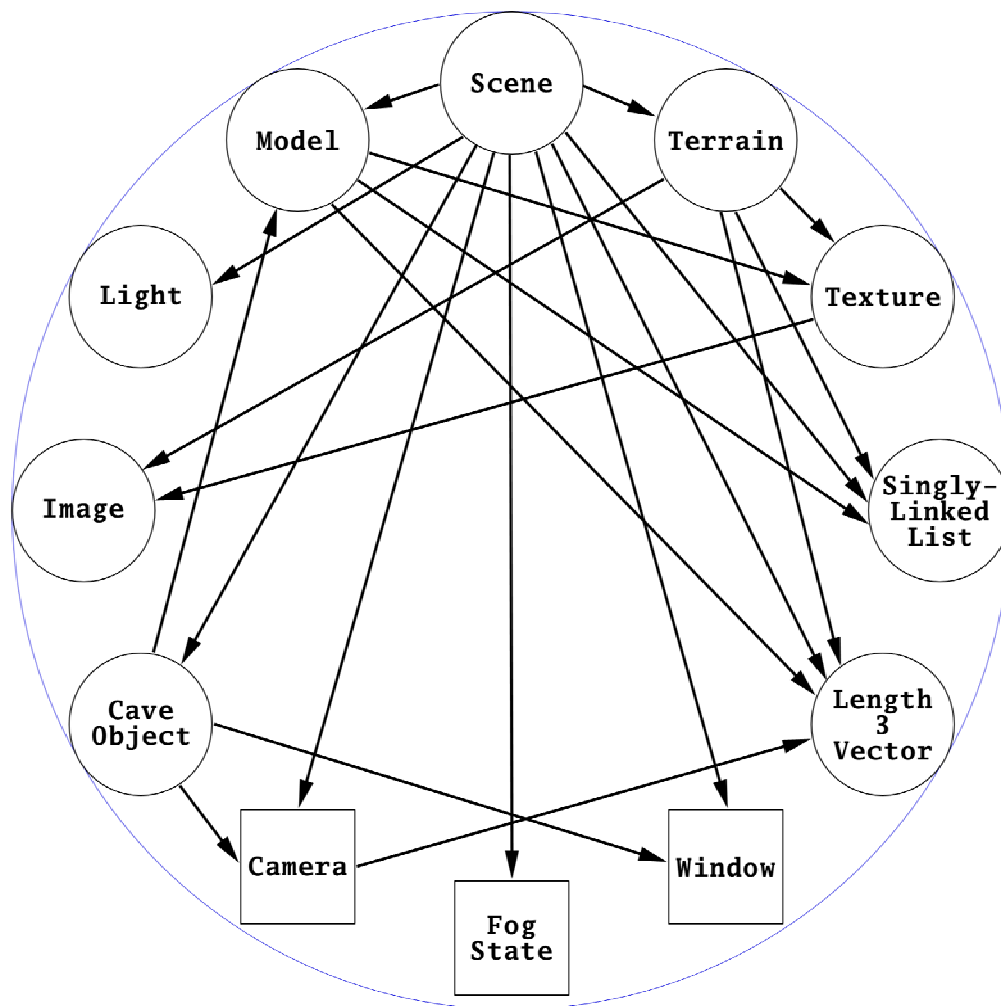
3D ENGINE NOTES

Any attributes may be left off and will be set to default values. Attributes may be specified in any order. Textures may be raw PPM, raw PGM, JPEG, or TARGA(raw or RLE). Scene files must end in ".sce".

APPENDIX D

CLASS AND STRUCTURE REQUIREMENTS

Classes are circled. Structures are squared. Arrows denote which class/struct each class/struct requires. That is, class A requires (points to) class B.



APPENDIX E

CLASSES

E.1 Cave Object Class

Files: cave.h, cave.cpp

Class cave_object stores cave attributes and provides methods to read cave files, and to update, draw, and test a cave.

State

bool	mode	cave mode flag, signifies whether cave rendering (transforms and drawing) is currently enabled Default: false
model	*mdl	pointer to cave geometry
uint	*facetptr	pointer to indeces to the vertices of the currentfacet
uint	facet	index to the current facet
uint	nfacets	number of facets contained in 'mdl'
uint	nsides	number of sides of current facet
uint	plus_side	indexes current side of current facet relative to the long side
float	Xscale	scales the output image horizontally Default: 1
float	Yscale	scales the output image vertically Default: 1
float	Xskew	angle, in degrees, of horizontal skew Default: 0
float	Yskew	angle, in degrees, of vertical skew Default: 0

Methods

```

cave_object()

    Set default values as listed above. Pointers are set to
    NULL. Unless specified above, numbers are set to zero.

void dump(char* append="")

    Dump cave information, most of which can be used in a
    cave(.cave) file, then print the given append string.

int read(char *pathname)

    Read a cave(.cave) file and sets any attributes which it
    specifies.

    Return zero upon success, less-than-zero upon failure.

void update(window *win, camera *cam)

    Update the projection matrix and variables which are used
    for cave transformations. Set the camera structure's
    nearclip based on the distance from the viewer position to
    the current facet, and use the new nearclip and the given
    farclip to set up the viewing frustum. The window
    structure is needed to get the window/screen aspect ratio.

void translation()

    Translate based on the viewer position relative to the
    cave. This should be applied to all objects in the scene
    whose location can change.

void rotation()

    Rotate such that the line-of-sight is orthogonal to the
    current facet. This should be applied to all objects in
    the scene whose orientation can change.

void draw()

    Draw the cave border. This should be the last drawing
    function called, and should be called without depth
    testing.

void testing()

    Draw visual aids: wireframe cave, filled current facet,
    facet normal, cone representing the viewer, etc.
    Note: This should only be called when cave mode is
    off(false).

```

E.2 Image Class

Files: image.h, image.cpp

Class image stores an image and provides methods for reading, writing, and manipulating images. It requires libjpeg, which is installed by default with most Linux distributions. More info on libjpeg may be found [here](#). The following formats are supported:

raw PGM	read/write
raw PPM	read/write
JPEG(RGB or gray)	read/write
raw TARGA(gray, RGB or RGBA)	read
rle TARGA(gray, RGB or RGBA)	read

State

uint width	image width in pixels
uint height	image height in pixels
uint depth	number of color channels(bytes) per pixel. Though the I/O methods only use depths [1, 4], the image manipulation methods handle arbitrary depths.
ulong size	total bytes needed to store the image: width by height by depth
byte *pixmap	pointer to pixel map

Methods

```

    image()
        Set numbers to zero, pointer to NULL.

    ~image()
        Free pixel map memory.

void dump(FILE *OUT=stdout)
        Dump resolution and size information to the given file
        pointer.

image&
    operator=(const image &img)
        Allocate a new pixel map and copy all data.
        Return a reference to the current image.
```

```

void  fliph()

    Flip image horizontally.

void  flipv()

    Flip image vertically.

image*
    conform(uint width, uint height, char crop_align=0)

    Crop and scale to the given width and height.
    Crop alignment may be 'c'(center), 'l'(left), 'r'(right),
    't'(top), or 'b'(bottom), or zero(no cropping).

    Return a pointer to the new image.

int   read_PNM(FILE *IN)

    Read a raw PGM(P5) or raw PPM(P6) image from the given file
    pointer.

    Return zero upon success, less-than-zero upon failure.

int   write_PNM(FILE *OUT)

    Write a raw PGM(P5) if depth is 1, or a raw PPM(P6) if
    depth is 3, to the given file pointer.

    Return zero upon success, less-than-zero upon failure.

int   read_JPEG(FILE *IN)

    Read a JPEG file, which may be RGB or grayscale, from the
    given file pointer.

    Return zero upon success, less-than-zero upon failure.

int   write_JPEG(FILE *OUT, int quality=75)

    Write a JPEG file, which may be RGB or grayscale, at the
    given quality, to the given file pointer.

    Return zero upon success, less-than-zero upon failure.

```

```
int    read_TARGA(FILE *IN)
```

Read a TARGA image from the given file pointer.
The following TARGA types are supported:

```

2      raw    RGB(A)
3      raw    grayscale
10     rle    RGB(A)
11     rle    grayscale
```

Return zero upon success, less-than-zero upon failure.

```
int    read(char *path)
```

Read any type supported by the above read functions by
calling the appropriate one, depending on the suffix(pgm,
ppm, jpg, tga) of the given path.

Return zero upon success, less-than-zero upon failure.

E.3 Light Class

File: light.h

Class light stores light attributes including position/direction and
colors and provides a method to enable a light.

State

```
bool    on          flag for whether or not light is on
                        Default: false
```

```
Glenum
    ID              OpenGL number of the light
```

```
float position[4]
                position of the light in homogeneous
                coordinates (x,y,z,w). Generally the fourth value is
                zero or one. If it is zero, the light is a
                directional light. Otherwise it is a point light.
```

```
Color ambient    ambient RGBA intensity of the light
```

```
Color diffuse     diffuse RGBA intensity of the light
```

```
Color specular    specular RGBA intensity of the light
```

Methods

```
light()
```

Set the default 'on' to false.

```
void enable()
```

Enable the light and update the graphics API with its color states.

See OpenGL function `glLight`.

E.4 Model Class

Files: `model.h`, `model.cpp`

Class `model` reads and writes OBJ-like files, and provides methods used to display models.

State

<code>char *filename</code>	path to the OBJ-like file the model was read from
<code>char *name</code>	name of the model, either from the filename or as specified within the file
<code>model *parent</code>	pointer to the parent model, or NULL if it has no parent
<code>model *instanceof</code>	pointer to the model that the current model is an instance of, or NULL if the current model is not an instance
<code>slist *children</code>	pointer to a list of children models, or NULL if the current model has no children
<code>bool smooth</code>	true if the model has smooth-shaded normals (one normal per vertex), or false if the model has flat-shaded normals (one normal per face). Default: false
<code>bool forcedisplay</code>	true if the object should always be displayed, ignoring the visible flag. Default: false

bool	visible	true if the object is visible and should therefore be displayed. Default: true
vectd	position	position(X, Y, Z) of the object Default [0, 0, 0]
vectd	rotation	Euler rotation angles(X, Y, Z), in degrees Default [0, 0, 0]
scalar	scale	uniform scale Default: 1
vectd	*verts	pointer vertex coordinates
vectd	*norms	pointer normals
texture	*colormap	color texture map pointer
texture	*reflectmap	reflection map pointer
texture	*bumpmap	bump map pointer. Bump mapping is not currently implemented.
uint	displaylist	integer name of the OpenGL display list. Default: 0
uint	**lines	pointer to indices for lines
uint	**tris	pointer to indices for triangles
uint	**quads	pointer to indices for quad faces
uint	**polys	pointer to indices for polygon faces
uint	nverts	number of vertices
uint	nlines	number of lines
uint	ntris	number of triangles
uint	nquads	number of quad faces
uint	npolys	number of polygon faces
float	**texcoords	pointer to texture coordinates


```

byte  *npolyverts      array of numbers of polygon vertices, one for
                        each polygon

byte  **colors          pointer to bytes representing RGBA colors for
                        each vertex

byte  displaymask       bit mask specifying how to interpret the display
                        list. The display mask indicates what sorts of
                        commands a display list contains, and may be a
                        bitwise combination of the following bit masks:

                                TRANSFORM_LIST      coordinate transform commands
                                MATERIAL_LIST       material state settings
                                TEXTURE_LIST        texture state settings
                                DRAW_LIST           drawing commands
                                or
                                FULL_LIST           all of the above

                        Default: 0

Color ambient           ambient RGBA reflectance of the material
                        Default [1, 1, 1, 1] (white)

Color diffuse           diffuse RGBA reflectance of the material
                        Default [1, 1, 1, 1] (white)

Color specular          specular RGBA reflectance of the material
                        Default [1, 1, 1, 1] (white)

Color emission          emissive RGBA reflectance of the material
                        Default [1, 1, 1, 1] (white)

float shininess         specular falloff of the material
                        Default: 0

```

Methods

```

model()

        Set default values as listed above. Pointers are set to
        NULL. Unless specified above, numbers are set to zero.

void dump(char *str="")

        Dump various information from the model's state.

```

```

int  read(char *pathname, slist *modellist=NULL, slist
        *texturelist=NULL)

        Read a model(.obj) file, add it to 'modellist' if given,
        and add its texture(s) to 'texturelist' if given.
        Return zero upon success, less-than-zero upon failure.

int  write(char *pathname)

        Write a model(.obj) file.
        Return zero upon success, less-than-zero upon failure.

void  material_settings()

        Set the OpenGL state according to the material properties
        of the model.

void  texture_settings(GLint color=GL_MODULATE, GLint
        reflection=GL_DECAL, GLint bump=GL_MODULATE)

        Set the OpenGL state according to the texture properties of
        the model, and specify how textures are to be used via
        three texture function parameters. For more information on
        the available texture functions, see the OpenGL function
        glTexEnv.

void  transforms()

        Translate, rotate, and scale.

void  draw_elements()

        Draw elements(lines, triangles, quads, polys) of a model.

void  draw_norms(float normsize)

        Draw normals of a model, and recursively draw normals of
        its children.

void  set_array_states()

        Set states of vertex arrays prior to drawing a model.
        Possible arrays to set are the vertex array, normal
        array(if smooth shaded), texture coordinate array(if color
        texture mapped), and color array(if model has vertex
        colors).

void  display()

        Display a model, and recursively display its children.
        display() calls transforms(), texture_settings(),
        material_settings(), and draw_elements().

```

```
void displaylist_setup(GLenum listmode=GL_COMPILE)
```

Set up a display list for a model, depending on the model's displaymask.

```
~model()
```

free all dynamically allocated memory

E.5 Scene Class

Files: scene.h, scene.cpp

Class scene encapsulates a lot of 3Dengine data. It provides a method for reading scene files, and methods for setting up background images and environment cubes(skyboxes). A method for writing scene files will be added eventually.

State

window	
*win	window pointer
camera	
*cam	camara pointer
Color bgcolor	background color Default [0, 0, 0, 0] (black)
Color ambient	ambient material color which is applied to newly imported models Default [.2, .2, .2, 1] (dark gray)
Color diffuse	diffuse material color which is applied to newly imported models Default [.8, .8, .8, 1] (light gray)
Color specular	specular material color which is applied to newly imported models Default [0, 0, 0, 0] (black)
Color emission	emissive material color which is applied to newly imported models Default [0, 0, 0, 0] (black)
float shininess	specular falloff which is applied to newly imported models Default: 1

```

vectd mapmin      minumum XYZ coordinates of the scene
                  Default: -1000
                  Element range: double precision floating point
                  numbers

vectd mapmax      maximum XYZ coordinates of the scene
                  Default: 1000
                  Element range: double precision floating point
                  numbers

double
    speed          speed factor for changes in camera position
                  Default: 25
                  Suggested range [1, 100]

double
    rotspeed       speed factor for changes in camera rotation
                  Default: 0.25
                  Suggested range [0, 1]

double
    rotaccel       acceleration factor for changes in camera rotation
                  Default: 7.5
                  Suggested range [1, 20]

double
    eyespacing     distance between eyes for stereo rendering
                  Default: 0.04
                  Suggested range [.01, .1]

model *background
                  pointer to a model consisting of a textured quad to
                  be used as a static background image

model *skybox      pointer to a model consisting of a textured cube to
                  be use as an environment cube

sllist
    *terrainlist   list off terrains contained in the scene

sllist
    *modellist      list of models contained in the scene

sllist
    *hierarchy     list of models without parents contained in the scene

sllist
    *texturelist   list of textures contained in the scene

light lights[GL_MAX_LIGHTS]
                  array of lights

```

```
fog_state
    *fog          fog state structure
```

Methods

```
    scene()

        Set default values as listed above.  Pointers are set to
        NULL.

void  read(FILE *ENV)

        Read a scene(.sce) file from the given file pointer.

void  write(FILE *SCE)

        Write a scene(.sce) file to the given file pointer.
        --not implemented at the moment :(

void  background_setup()

        Setup a background flat, compiling it into a displaylist.

int   skybox_setup(cave_object *cave)

        Setup a skybox(env cube), compiling it into a displaylist.

        Return zero upon success, less-than-zero upon failure.
```

E.6 Singly-linked List Class

File: sllist.h

Template class sllist provides dynamic storage for any type of elements.

State

```
T      *ptr      pointer to the stored element
sllist *next      pointer to the next node in the list
```

Methods

```
sllist()

        Set pointers to NULL.
```

```

unsigned int
    size()

        Return the number of items in the list.

T*    head()

        Return a pointer to the first element, or NULL if the list
        is empty.

T*    get(int target)

        Return the item indexed from first of list by target.
        Note: step() is faster for retrieving each item
        sequentially.

T*    step(int stride=1)

        Return the item indexed from previous step call by stride.
        Note: step contains a static variable, so don't "mix" step
        calls! Use step(0) to rewind.

int    count(T *key)

        Return the number of instances of key in the list.

int    push(T *newptr)

        Push a new node with a pointer to newptr onto the list.

        Return the number of items in the list.

int    insert(T *afterme, T *newptr)

        Insert a new node with a pointer to newptr into the list
        after the first node which points to afterme, if there is
        one.

        Return the number of items in the list.

T*    pop()

        Pop the top node off the list.

        Return a pointer to the top node, or NULL if the list is
        empty.

```

```
int    remove(T *bad, int deletes=-1)
```

Remove nodes pointing to bad element. If deletes is non-negative, then it specifies the maximum number of nodes to be deleted. Otherwise all nodes matching bad are deleted.

Return the number of nodes deleted.

E.7 Terrain Class

Files: terrain.h, terrain.cpp

Class terrain provides methods to read a grayscale image and display it as a terrain by interpreting it as a heightmap. Its state and methods are very similar to class model.

WARNING: This class is weak and displays very slowly. Avoid using it.

E.8 Texture Class

Files: texture.h, texture.cpp

Class texture provides a method for reading an image and storing it in a new 2D texture object.

State

```
char *filename    path to the image file
uint  ID          texture object ID number
```

Methods

```
texture()
```

Set default values.

```
void dump(char *append="")
```

Dump texture state, followed by the append string.

```
uint read_2D(char *filename)
```

Read an image file into a new texture object.

Return ID upon success, zero upon failure.

E.9 Length 3 Vector Class

File: vect.h

Template class vect<> provides logical, arithmetic, and assignment operations, as well as other vector math functions for manipulating vectors consisting of three floating point elements. It is advisable to use the type defines 'vectf' and 'vectd' instead of using vect<> directly.

Type Definitions

```
typedef vect      vectf
typedef vect      vectd
```

Defines

```
PI          = 3.1415926535897
SMALLNUM    = 1.0e-6
BIGNUM      = 1.0e6
```

Global Constants

```
static const int  X = 0
static const int  Y = 1
static const int  Z = 2
```

Methods - constructors, set, access operator, and dump

```
vect()
    Initialize to [0,0,0].

vect(scalar s)
    Initialize to [s,s,s].

vect(scalar x, scalar y, scalar z)
    Initialize to [x,y,z].

vect(const vect &v)
    Initialize to v.
```



```
const
vect&      set(scalar x, scalar y, scalar z)
```

Set vector to [x,y,z].

Return vector.

```
scalar&
operator[](int i)
```

Return vector element i.

```
void dump(char *append="")
```

Dump [vector] followed by append.

Methods - component-wise scalar & vector arithmetic

```
vect operator+(scalar s, const vect &v)
vect operator+(scalar s)
```

Return vector $s + v$ or $v + s$.

```
vect operator-(scalar s, const vect &v)
vect operator-(scalar s)
```

Return vector $s - v$ or $v - s$.

```
vect operator*(scalar s, const vect &v)
vect operator*(scalar s)
```

Return vector $s * v$ or $v * s$.

```
vect operator/(scalar s, const vect &v)
vect operator/(scalar s)
```

Return vector s / v or v / s .

Methods - component-wise scalar & vector arithmetic assignment

```
const
vect& operator =(scalar s)
```

Set vector to [s,s,s] and return it.

```
const
vect& operator+=(scalar s)
```

Set vector to $v + s$ and return it.

```
const
vect& operator-=(scalar s)
```

Set vector to $v - s$ and return it.

```
const
vect& operator*=(scalar s)
```

Set vector to $v * s$ and return it.

```
const
vect& operator/=(scalar s)
```

Set vector to v / s and return it.

Methods - component-wise vector arithmetic

```
vect operator-()
```

Return vector with components negated.

```
vect operator+(const vect &v)
```

Return $v_1 + v_2$.

```
vect operator-(const vect &v)
```

Return $v_1 - v_2$.

```
vect operator^(const vect &v)
```

Return $v_1 \wedge v_2$ (multiply component-wise).

```
vect operator/(const vect &v)
```

Return v_1 / v_2 .

Methods - dot product and cross product

```
scalar
operator*(const vect &v)
```

Return $v_1 \cdot v_2$.

```
vect operator%(const vect &v)
```

Return $v_1 \times v_2$.

Methods - vector arithmetic assignment

```
const
vect& operator =(const vect &v)
```

Assign vector to v and return vector.

```
const
vect& operator+=(const vect &v)
```

Assign vector to v1 + v2 and return vector.

```
const
vect& operator-=(const vect &v)
```

Assign vector to v1 - v2 and return vector.

```
const
vect& operator^=(const vect &v)
```

Assign vector to v1 ^ v2 and return vector.

```
const
vect& operator/=(const vect &v)
```

Assign vector to v1 / v2 and return vector.

```
const
vect& operator%=(const vect &v)
```

Assign vector to v1 % v2 and return vector.

Methods - relational

```
bool operator==(const vect &v)
```

Return true if v1 equals v2, else return false.

```
bool operator!=(const vect &v)
```

Return true if v1 does not equal v2, else return false.

```
bool operator< (const vect &v)
```

Return true if each element of v1 is less than each respective element of v2, else return false.

```
bool operator<=(const vect &v)
```

Return true if each element of v1 is less than or equal to each respective element of v2, else return false.

```
bool operator> (const vect &v)
```

Return true if each element of v1 is greater than each respective element of v2, else return false.

```
bool operator>=(const vect &v)
```

Return true if each element of v1 is greater than or equal to each respective element of v2, else return false.

```
int parallel(vect B)
```

Return 1 if vector A is parallel to vector B and they point in the same direction. Return -1 if they are parallel but point in opposing directions. Return zero if they are not parallel.

Methods

```
void swap(vect &v1, vect &v2)
```

Swap two vectors.

```
vect minimum(const vect &v1, const vect &v2)
```

Return a vector having the minimum X-Y-Z components of the two.

```
vect minimum(vect *v, unsigned long i)
```

Return a vector having the minimum X-Y-Z components of all vectors in an array.

```
vect maximum(const vect &v1, const vect &v2)
```

Return a vector having the maximum X-Y-Z components of the two.

```
vect maximum(vect *v, unsigned long i)
```

Return a vector having the maximum X-Y-Z components of all vectors in an array.

```
vect average(const vect &v1, const vect &v2)
```

Return a vector having the average X-Y-Z components of the two.

```
vect average(vect *v, unsigned long i)
```

Return a vector having the average X-Y-Z components of all vectors in an array.

```

vect  abs(const vect &v)

    Return a vector having the absolute value X-Y-Z components
    of v.

const vect& abs(vect *v)

    Set v to its own absolute value and return it.

vect  sqr(const vect &v)

    Return a vector consisting of the components of v, squared.

const vect& sqr(vect *v)

    Square the components of v and return it.

scalar
    M(const vect &v)

    Return the magnitude of v.

scalar
    Msqr(const vect &v)

    Return the magnitude of v, squared.

vect  N(vect v)

    Return the normal of v.

const
vect& N(vect *v)

    Normalize v and return it.

scalar
    D(vect v1, vect v2)

    Return the distance between two vectors.

scalar
    Dsqr(vect v1, vect v2){

    Return the distance between two vectors, squared.

scalar
    angle_between(vect v1, vect v2)

    Return the angle between two vectors, in degrees.

```

```
void quat_mult(scalar *dstw, vect *dstv,  
               scalar w0,vect v0,scalar w1,vect v1)
```

Multiply quaternions $w_0 + v_0$ and $w_1 + v_1$, and store the new quaternion in $dstw + dstv$.

```
vect rotate(vect v, scalar a, vect axis)
```

Rotate v by " a " degrees about axis.
Note: Axis must be normalized.

APPENDIX F

STRUCTURES

F.1 Camera Structure

File: camera.h

Structure camera stores location and orientation of a viewer, as well as field-of-view and clipping attributes which define a viewing frustum.

vectd position	location of camera Default [0,0,0]
vectd rotation	orientation of camera - Euler rotation angles (X, Y, Z), in degrees Default [0,0,0]
float fov	field of view in degrees, unused in cave mode Default: 60
float nearclip	distance to the near clipping plane Default: 0.1
float farclip	distance to the far clipping plane Default: 1000

F.2 Fog State Structure

File: fog.h

Structure fog_state stores parameters of distance, density, function, and color of fog.

```
double
    start          GL_FOG_START - starting distance of the fog

double
    end            GL_FOG_END - ending distance of the fog

double
    density        GL_FOG_DENSITY - density of the fog

GLint mode         GL_FOG_MODE - fog function: GL_EXP, GL_EXP2, or
                    GL_LINEAR

Color fcolor       GL_FOG_COLOR - fog color
```

See OpenGL function glFog.

F.3 Window Structure

File: window.h

Structure window stores position and size of a window/viewport, the window's ID number, and a bit mask specifying it's display mode.

Guint

 ID window ID number as returned by the windowing system.
 See GLUT function glutCreateWindow.

Guint

 displaymode bit mask for display and depth buffer configuration.
 See GLUT function glutInitDisplayMode.

Guint

 xpos horizontal origin of viewport

Guint

 ypos vertical origin of viewport
 See GLUT function glutInitWindowPosition.

Guint

 width width of viewport

Guint

 height height of viewport

See GLUT function glutInitWindowSize.

APPENDIX G

MISCELLANEOUS CODE

G.1 Color Type And Definitions

File: colors.h

Color Type Definition

```
typedef float Color [4]
```

Type Color is an array of 4 floats. They are meant to be interpreted as red, blue, green, alpha, with color ranges normalized from zero to one.

Constant Color Definitions

Aqua	0.00, 1.00, 1.00, 1
Black	0.00, 0.00, 0.00, 1
Blue	0.00, 0.00, 1.00, 1
Fuchsia	1.00, 0.00, 1.00, 1
Gray	0.50, 0.50, 0.50, 1
Green	0.00, 0.50, 0.00, 1
Lime	0.00, 1.00, 0.00, 1
Maroon	0.50, 0.00, 0.00, 1
Navy	0.00, 0.00, 0.50, 1
Olive	0.50, 0.50, 0.00, 1
Purple	0.50, 0.00, 0.50, 1
Red	1.00, 0.00, 0.00, 1
Silver	0.75, 0.75, 0.75, 1
Teal	0.00, 0.50, 0.50, 1
White	1.00, 1.00, 1.00, 1
Yellow	1.00, 1.00, 0.00, 1
Light	0.90, 0.90, 0.90, 1
Dark	0.10, 0.10, 0.10, 1
Clear	0.00, 0.00, 0.00, 0

G.2 GLUT Environment

Files: env_glut.h, env_glut.cpp

This is an effort to abstract GLUT calls out of the main application code. The following handler functions must be defined externally:

```
void  visibility(int state)
void  reshape(int w, int h)
void  display()
void  idle()
void  mouse(int button, int state, int x, int y)
void  motion(int x, int y)
void  passive(int x, int y)
void  keydown(unsigned char key, int x, int y)
void  keyup(unsigned char key, int x, int y)
void  specialdown(int key, int x, int y)
void  specialup(int key, int x, int y)
```

Defines

```
LEFT_ARROW      = 100
UP_ARROW        = 101
RIGHT_ARROW     = 102
DOWN_ARROW      = 103

DOUBLE_RGBA_DEPTH = GLUT_DOUBLE | GLUT_RGBA | GLUT_DEPTH
WinsysInit(x, y)  = glutInit(x, y)
InitDisplayMode(mode)
                  = glutInitDisplayMode(mode)
WarpPointer(x, y) = glutWarpPointer(x, y)
SwapBuffers()    = glutSwapBuffers()
GetWindowWidth() = glutGet(GLUT_WINDOW_WIDTH)
GetWindowHeight() = glutGet(GLUT_WINDOW_HEIGHT)
HideCursor()     = glutSetCursor(GLUT_CURSOR_NONE)
ShowCursor()     = glutSetCursor(GLUT_CURSOR_INHERIT)
KeyRepeatOn()    = glutSetKeyRepeat(GLUT_KEY_REPEAT_ON)
KeyRepeatOff()   = glutSetKeyRepeat(GLUT_KEY_REPEAT_OFF)
MainLoop()       = glutMainLoop()
EnterGameMode()  = glutEnterGameMode()
LeaveGameMode()   = glutLeaveGameMode()
```

Functions

```
void  non_display()

        Don't display anything.

void  SetHandlers()

        Set all callback handlers to default functions.
```

```
void Pause()

    Nullify most callback handlers and set the display callback
    to non_display().

void Unpause()

    Set all callback handlers to default functions--
    SetHandlers().
```

G.3 OpenGL Error Checking

Files: errorcheck.h, errorcheck.cpp

If OpenGL has thrown an error, errorcheck prints the error code and optional message, then exits the program.

Function

```
void errorcheck(const char msg[]=NULL)
```

G.4 Trigonometric Approximations

Files: trig_approximations.h, trig_approximations.cpp

These functions provide fast access to approximations of the desired trigonometric values. Angles are in degrees.

Defines

PI = 3.1415926535897
SMALLNUM = 1.0e-6

Functions

void TrigApproxInit()

TrigApproxInit() must be called prior to using the other functions or macros. It

double Sin(double angle)

Return sine approximations.

double Cos(double angle)

Return cosine approximations.

double Tan(double angle)

Return tangent approximations.

Macros

Csc(angle) = (1.0/Sin(angle))

Return cosecant approximations.

Sec(angle) = (1.0/Cos(angle))

Return secant approximations.

Cot(angle) = (1.0/Tan(angle))

Return cotangent approximations.

VITA

Christopher Dean Anderson
418 Oak Springs Dr.
Seguin, TX 78155

Bachelor of Science in Computer Science
Southwest Texas State University
May 2000

Master of Science in Visualization Sciences
Texas A&M University
December 2003