# SOLACE: A Hierarchal Shader-Based Scenegraph Renderer

Joshua Shagam

3rd May 2003

# Contents

# List of Algorithms

# List of Figures

# List of Tables

# Chapter 1

# Design Goals

SOLACE (Stratified-ONSET[1] Labrynthine Actively-Computed Environment) is the end-user client portion of the Solace project[2], an ongoing general-purpose virtual reality environment project. One of the most vital portions of SOLACE, and definitely its most visible, is the 3D rendering engine, which has been in the planning stages since mid-1997 and under active development since September of 1999. During that time, the SOLACE renderer has been rewritten once, refactored twice, and is currently a general research platform for ongoing work in the area of highly-scalable realtime 3D rendering.

When designing a system as complex as a realtime 3D renderer, it is typical to have a number of goals in mind. Many of these design goals evolved in parallel with the SOLACE renderer, whereas many others have been an integral part from the beginning.

## 1.1 Platform Independence

Every current system which is capable of displaying realtime 3D graphics should be a possible compilation target for a modern 3D engine. However, not every current system has the same capabilities in terms of hardware, software, and high-level APIs. The design of a complex 3D engine must therefore accomodate the wide variety of interfaces, both at the system level and the user level.

### 1.1.1 Render Target Independence

Most modern graphics workstations have OpenGL available as a rasterization API, and so it's tempting to simply write all code as OpenGL. However, some hardware is not optimized for OpenGL, and instead is designed to be used most efficiently with Microsoft's DirectX API. Additionally, when we leave the realm of high-level workstation graphics and go into video game consoles (such as the Sony Playstation and Playstation 2 or the Nintendo 64 or GameCube) and other embedded devices (such as the Nokia N-Gage mobile phone), the landscape becomes quite tumultuous; platform-specific APIs, "graphics" chips which are really a CPU optimized for software rasterization (classic examples being the Amiga and Atari Jaguar, and current examples being the graphics chips used in the Nintendo GameCube, Sun's constantly-forthcoming MAJC architecture, and the programmable functionality in current workstation graphics chips such as the ATI Radeon and nVidia geForce series), and graphics chips which are programmed almost, but not quite, entirely unlike OpenGL (such as in the Sony Playstation).

Additionally, devices which are capable of displaying vector graphics very quickly but which aren't necessarily capable of full 3D rasterization must be considered. Volumetric vector displays, fast but non-3D-equipped high-performance X11 workstations, and direct neural interfaces are examples of where final render techniques might be totally different than OpenGL rendering.

The end result is that there must be an abstract separation between the rendering engine and the final display; as such, the underlying rasterization layer should be told what to render, not how to render it.

[1] Online Novel, SOLACE Environment Transport; the server portion of Solace

[2] http://www.cs.nmsu.edu/~joshagam/Solace/

### 1.1.2 Interface Independence

Another major pitfall in platform independence is relying on a single interface which severely limits the sorts of user interfaces which the renderer can communicate with.

On one end of the spectrum, tying a renderer to a single platform-specific display API, such as Xlib or WinAPI, makes it easy to port between different user interfaces within that platform, but difficult to port between different platforms.

On the other end of the spectrum, tying arenderer to a specific cross-platform display API, such as GTK, SDL, or wxWindows, makes it easy to port between different platforms, but difficult to port between different user interfaces within those platforms. For example, GTK and wxWindows are excellent choices for programs which use multiple windows, such as modelling programs or complex interfaces to rich 3D environments, but don't work for single-window or fullscreen interfaces. However, single-window-oriented display APIs such as SDL tie the renderer to only be used in single-window fullscreen-oriented applications, such as games, and become very cumbersome to use for modelling programs, which then must be adapted towards a single fullscreen view.

Additionally, most of the cross-platform display APIs are set up in such a way that not only is the renderer tied to the windowing API, but the renderer is further tied to a specific rendering API; for example, an SDL application must select *either* OpenGL *or* software rendering, and furthermore, platform-specific APIs (such as what is necessary on gaming consoles) are totally out of the question without adding bindings for those APIs to the display API. An extreme example of this is GLUT, which ties the application to using OpenGL with a very specific, inflexible event model.

Ideally, there should be a layer of abstraction on top of the display API separate from the rendering API.

### 1.1.3 Problem-Domain Independence

Furthermore, a renderer is usually designed with a specific problem domain in mind. Is it intended for an action game? An exploration game? A battlefield simulator? A cinematic rendering engine? An architectural walkthrough system?

By limiting a renderer to a specific problem domain, it is optimized for that domain. The tradeoff is that it cannot be used for other purposes if they should come to mind.

A renderer which can be used for *any* problem domain won't necessarily be the fastest or most efficient for any specific purpose, but it will have the huge advantage of being used for any purpose, with possibility for having its internals optimized for a specific purpose later.

The SOLACE renderer has been designed with interactive spaces in mind, with an emphasis on interactivity and depth of experience, but not on high framerates. Specifically, it uses a conceptual model similar to the classic MUD[3] design of "rooms" (which are not necessarily indoors) connected by exits, which maps directly to portal-based visibility. However, the functional unit of a "room" can be arbitrarily-large, being a vast expanse of desert, or an entire city block crawling with denizens, or even a bedroom cluttered with items randomly strewn about, so some intelligence in the visibility determination is required.

Due to the design goals, the SOLACE tradeoff is for quality and flexibility over frame rate (15fps is acceptible). However, the renderer itself is designed in such a way that it could still be used for lower-quality high-framerate "twitch-factor" applications, such as action games. Essentially, the SOLACE renderer leaves the quality/speed balancing decision up to the application designers and content creators; other applications using the SOLACE renderer don't have to make the same tradeoff as SOLACE.

## 1.2 Resource Utilization

Currently, there is a wide range of systems which are capable of displaying 3D graphics. Different systems have different strengths and weaknesses; some have amazing levels of fillrate but are amazingly slow at vertex processing. Others are amazingly fast at vertex processing but are limited in fillrate. Yet others are phenomenal in both regards, but many more are simply slow at everything.

Many "current" graphics chips have an incomplete set of hardware-implemented pixel operations. Memory is an issue, as well; some systems have as much as 1GB of texture memory, while others have as little as 4k, if

[3]Multi-User Dungeon, a text-based interactive roleplaying game

they even have any dedicated texture memory at all. Some systems can cache geometry within graphics memory; others cannot.

The end result is that a renderer which is to be used in a wide variety of modern systems must have a lot of flexibility in how it utilizes resources. On a vertex-bound system, it must be simple to simplify geometry to reduce the amount of vertex processing. On a memory-bound system, it must be possible to turn off features which require a lot of memory storage. On the other end of the spectrum, a gaming enthusiast's system should be able to make use of the large amounts of capability which they've paid good money for.

A modern renderer should scale from a low-end gaming console to a high-end CAD workstation, and to everywhere in between.

The SOLACE renderer takes the approach of using a geometry representation which can be arbitrarily-resampled so as to make maximum use of available vertex processing capability, while using a render pipeline which can be manipulated to make maximum use of pixel processing capability. These will be elaborated on later.

## 1.3 Artistic Expressiveness

In order to provide for ultimate artistic expressiveness within a 3D environment, many things must be considered.

### 1.3.1 Render Functions

First, the number of render operations which can be performed on a piece of geometry must be maximized. Whereas traditional 3D renderers simply provide a number of "boilerplate" render operations (e.g. "draw this opaque and lit" or "draw this transparent using alpha modulation"), it is only a matter of time before it becomes extremely cumbersome to define specific render operations and the parameters they require. A much more general and expressive solution is to use "shaders," where a render pass is defined with a simple language of rendering operations. However, generalized shaders are slow and not particularly suited to realtime rendering, while programmable pixel and vertex functionality is not sufficiently-robust in the vast majority of available graphics hardware.

The SOLACE renderer is designed to use shader scripts with flexibility in mind, using basic pixel operations. However, more advanced operations on the scene geometry itself will be implemented in a way which will be compatible with all forms of rendering, while taking advantage of hardware support when applicable.

Additionally, shader scripts are implemented in such a way that the shader complexity can be reduced in order to reduce the amount of needed fillrate, while also allowing for inefficient pixel operations to be performed in some other manner depending on the underlying display hardware.

### 1.3.2 Lighting

A much more subtle place where artistic expression takes place is in subtle "abuses" of rendering mechanisms. For example, it should be possible for an artist to model lighting effects into a mesh, by purposefully using surface normals which are not of unit length or which do not match the actual surface; for example, it might be useful for a mesh to be lit as though the light is to the side, for performing tangent-space normal mapping or other interesting special effects (such as fur or ephemeral surfaces).

Additionally, it should be possible for a surface normal to not be of unit length, causing certain vertices to be lit less than others, or even to "super-luminate" some vertices in order to draw attention to them (e.g. to simulate fluorescence). This issue has a major, if subtle, implication on the design of a renderer; if a scene representation allows scaling as a hierarchal transformation, then it must adjust surface normals accordingly (otherwise, an object which is scaled down will receive less light, and an object which is scaled up will be super-luminated). Many hierarchal renderers "solve" this issue by normalizing all surface normals at render time – but this removes the ability to non-uniformly illuminate an object! So, either the renderer must maintain scaling values separately from the transformation matrix stack, or it must simply not allow scaling as a hierarchal transformation, and require some other mechanism for scaling objects.

The SOLACE renderer takes the latter approach.

## 1.4 Zero Precomputation

Renderers which rely on precomputation inherently limit the amount of artistic expressiveness and problem-domain flexibility.

### 1.4.1 Visibility

If visibility information is predetermined, then an environment cannot behave in an unexpected way. Static visibility means there will never be a modification to a game which allows a rocket launcher to destroy a wall. Static visibility means that a door cannot be removed from its hinges. Static visibility means that the renderer needs to know ahead of time if a plane is going to knock down a skyscraper in a dense metropolitan area.

As stated before, the SOLACE renderer uses a hybrid approach of portals and dynamic spatial partitioning.

### 1.4.2 Lighting

If lighting is predetermined, then dramatic effects are severely limited. Lights cannot be turned on or off unless it is explicitly designed into the environment. If there is a power outage and all the lights go out, then someone cannot arbitrarily carry around a flashlight, at least not with realistic effects. If lighting is predetermined, then the eery glow of the mage's aura as he walks down the hallway does not look particularly good.

SOLACE leaves all lighting to the rasterization backend, with the intent that it will use either vertex lighting or projective shading approaches, using multipass dynamic shadowing techniques.

### 1.4.3 Mesh Detail

If mesh level of detail is predetermined, then the overall render quality is fixed to a small subset of available hardware. The 9,000-polygon banana with 5 fixed levels of detail still won't scale down to a system where a 200-polygon object is too much for a background element, and it won't scale up to a system where 18,000 polygons don't put a dent in performance. And if mesh level of detail is predetermined, then peeling the banana can become a difficult task.

The SOLACE renderer settles this by using a geometry image, essentially a high-resolution patch representation which may be arbitrarily-resampled as required to maintain an interactive framerate. A detailed description is given in section 2.2.

## 1.5 Ease of Content Creation

A good renderer will make it easy for non-artists to have at least some success in producing their own content, even if it involves putting together pre-fabricated clip-art into a new configuration. By using hierarchal modelling, rather than skeletal deformation, or by allowing for cut-and-paste operations from one piece of geometry to another, this makes the act of synthesizing new content from existing content much simpler.

Certain underlying designs can also significantly lower the barrier to content creation for unskilled users, while facilitating extreme levels of brilliance for skilled users. For example, simple geometry representations which can be modelled like clay or designed by specification are much more likely to be user-friendly than tediously-placed clouds of points which are arduously connected by polygons.

This is another place where the SOLACE renderer's use of geometry images becomes a clear advantage; although not yet implemented, there is already some design work on how to design a modeller specifically for geometry images which would allow for an object to be modelled with a series of simple deformations in a way which would be similar to modelling with clay.

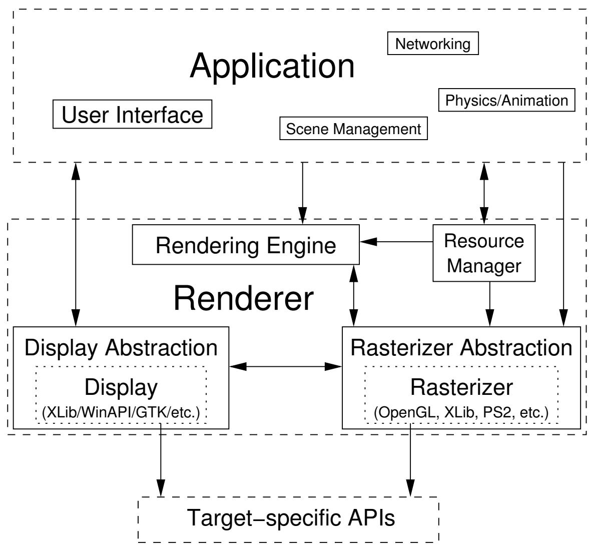

Figure 2.1: Data flow between major component groups

# Chapter 2

# High-Level Interface

## 2.1 API level

We maintain an abstraction between the application and the renderer, and functional abstractions within the renderer. We group the major functionality of the renderer into the rendering engine, the resource manager, the display abstraction, and the rasterizer abstraction. Their communication paths are depicted in figure 2.1.

The remainder of this chapter provides a conceptual overview of the responsibilities of each functional group.

### 2.1.1 Application

The application is the part which the user loads and works with directly. The application is where all high-level application logic resides, all of which boils down to loading and manipulating a 3D scene and telling the renderer to do its job, as far as the renderer is concerned.

Its outgoing communications within the renderer's framework are as follows:

- Resource manager: Whenever the application needs or no longer needs a resource (such as a mesh or texture), it tells the resource manager.
- Renderer: Whenever the application wants to render a frame, it tells the renderer which scene to render, with what viewpoint and rasterizer.

- Rasterizer: The application tells the rasterizer when to clear the display, which color planes to render to, and which part of the display to render in. Additionally, the application itself can tell the rasterizer to render shapes which are outside of the rendering engine's control, such as user interface elements.

- Display: The only outgoing communication to the display is the creation or deletion of the display targets themselves.

### 2.1.2 Resource Manager

The resource manager handles the allocation and deallocation of memory for resources (notably meshes and textures), and the garbage collection thereof.

When the application requests a resource, the resource manager checks to see if the resource is already loaded, in which case it gives the application a handle to the existing resource. Otherwise, it loads the resource (theoretically from disk, the Internet, or a procedural resource generator, though currently only the from-disk method is implemented). If the resource is not available, it tells the application, and if the resource is only partially-available (such as for a streaming Internet download) it queues the resource up for completion.

When the application sends notification that a resource is no longer needed, the resource manager revokes the handle it gave to the application, and checks how many handles remain for the resource. If none remain, the resource manager then notifies the rasterizer and rendering engine that the resource no longer exists (so they can purge those resources from their internal caches), and deletes the resource from memory.

### 2.1.3 Rendering Engine

When the application tells the rendering engine to render a scene from a certain viewpoint, it does such, determining which parts of the scene hierarchy are visible and sends those to the rasterizer for rendering.

Although the rendering engine conceptually has the simplest task, it is actually the most complicated part of the renderer, and has to perform the most actual work.

### 2.1.4 Display

Currently, there is no windowing layer abstraction; instead, we just use plain XLib. This decision was made because XLib's API is available with hardware OpenGL acceleration on all of the current development platforms (Linux, SunOS, Mac OS X, and, to a limited extent, Microsoft Windows), and it is relatively simple to port from XLib to a more abstract display layer later, whereas porting from a more complex layer would become quite difficult.

Its outgoing communications are as follows:

- To the application, it provides vital user interaction data; mouse clicks, keyboard presses, pointer motion, window reconfiguration events, and so on.

- To the rasterizer it provides information about the physical display characteristics, notably the size of the displayed area.

- It communicates with the low-level platform APIs for window creation and deletion, display management, and so on.

### 2.1.5 Rasterizer

The rasterizer is simply told what to do by the rendering engine; the rendering engine typically sends it a transformation matrix and a series of shader scripts, and then it draws them as it's informed.

Its outgoing communication is simple but important. It tells the display abstraction when to flip display buffers as appropriate for the rasterization engine, and in some instances, it can provide visibility information to the rendering engine for occlusion culling purposes.

## 2.2 Artistic Level

The SOLACE renderer uses geometry images for all geometry representations. Although geometry images as formally described only allow for three-dimensional fixed-precision data [1], this implementation instead uses as many as 8 channels of data corresponding to surface coordinate $\langle x,y,z\rangle$, surface normal $\langle x',y',z'\rangle$, and texture co-ordinate $\langle s,t\rangle$. It is also worth mentioning that this implementation has used geometry images since the beginning in 1999, and it was the conception of a geometry image as a viable geometry representation which led to the onset of this renderer project.

Geometry images have a number of useful features which lend themselves well to a high-quality realtime renderer; notably, it is a simple matter to perform continuous level-of-detail (both for reducing and increasing the detail of a mesh), "morph"-type warping between arbitrarily-different meshes, and determining an optimal silhouette boundary from an arbitrary viewpoint for the purpose of generating shadow volumes and cartoon-style outlines [2].

### 2.2.1 Scene Hierarchy

### 2.2.2 Shader Scripts

# Chapter 3

# The Rendering Pipeline

## 3.1 Initial Setup

### 3.1.1 Scenegraph Traversal

### 3.1.2 Visibility Determination

## 3.2 Raster Driver

### 3.2.1 Shader Execution

### 3.2.2 Shadow Casting

# Chapter 4

# Performance Improvements

## 4.1 Continuous Level-Of-Detail

## 4.2 Displaylist Caching

## 4.3 Occlusion Culling

# Chapter 5

# Low-Level Interface

## 5.1 Revision Control

A number of the object classes implement a simple revision control interface for the purpose of quickly determining whether the data has changed since the last time some piece of code last accessed it. At a minimum, any object which implements revision control provides the following public interface:

- `void Touch()`: Register that this object has somehow changed
- `bool Changed(void *)`: Return true if this object has been changed since the last access by the specified globally-unique pointer
- `void Updated(void *)`: Register that the specified globally-unique pointer is now up-to-date with this object

## 5.2 Vector Library

In order to perform 3D calculations, it is desirable to have a library of 3D mathematics functions, specifically for linear algebra computations on vectors and matrices.

Although many libraries for linear algebra exist, they typically provide functionality intended for general linear algebra computations. However, for 3D viewing transformations, there are a number of constraints on the possible inputs:

- All vectors are treated as 4-tuples of $\langle x, y, z, w\rangle$ where $w$ is either 0 or 1, depending on the operation; as such, only $\langle x, y, z\rangle$ is actually stored and operated on (as $w$ only makes a difference when multiplying a matrix by a vector)
- All matrices are $4 \times 4$ in the form
$$\begin{bmatrix} R_{xx} & R_{xy} & R_{xz} & T_x \\ R_{yx} & R_{yy} & R_{yz} & T_y \\ R_{zx} & R_{zy} & R_{zz} & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$
and for our purposes, we can assume that the $R$ submatrix will be orthonormal; that is, all transformations encompass a non-shearing rotation and/or a translation. Scaling, perspective projection, and so on are not supported. (All perspective calculations will be performed by the underlying rasterizing engine.)

As such, the actual implementation of the vector library can take advantage of many shortcuts which reduce the complexity of an implementation.

### 5.2.1 3-Dimensional Vector (`vec3`)

The 3-dimensional vector class, vec3, is implemented as a struct with three public float members, x, y, and z, and provides the following operators:

- `vec3()`: Initialize the vector to default values; $\bar{v} = \langle 0,0,0 \rangle$
- `vec3(float,float,float)` (constructor: Initialize the vector to provided values; $\bar{v} = \langle x,y,z \rangle$
- `float &operator[](int)`: Provide indexed addressing to the struct; $\bar{v}_0 = \bar{v}_x$, $\bar{v}_1 = \bar{v}_y$, $\bar{v}_2 = \bar{v}_z$
- `bool operator==(vec3)`: Evaluates true if the right-hand side is equal to this, false otherwise
- `bool operator!=(vec3)`: Evaluates false if the right-hand side is equal to this, true otherwise
- `bool operator<(vec3)`: Provides a monotonic ordering for storage into an associative set or array (compares by x, then by y, then by z)
- `vec3 operator+(vec3)`: Return the summation of this vector with another; $\bar{v} + \bar{w}$
- `vec3 operator-()`: Return the negation of this vector; $-\bar{v}$
- `vec3 operator-(vec3)`: Return the subtraction of another vector from this one; $\bar{v} - \bar{w}$
- `float operator*(vec3)`: Return the dot product of this vector with another; $\bar{v} \cdot \bar{w}$
- `vec3 operator*(float)`: Return the multiplication of this vector with a scalar; $\bar{v} \cdot f$
- `vec3 operator/(float)`: Return the division of this vector by a scalar; $\frac{\bar{v}}{f}$
- `vec3 operator%(vec3)`: Return the cross product of this vector with another; $\bar{v} \times \bar{w}$

Additionally, the following global operators are provided by the header file:

- `vec3 operator*(float, vec3)`: Return the multiplication of a scalar by a vector; $f \cdot \bar{v}$
- `double abs(vec3)`: Return the magnitude of a vector; $\|\bar{v}\| = \sqrt{\bar{v} \cdot \bar{v}}$

### 5.2.2 Transformation Matrix (`Matrix`)

The `Matrix` class provides an array of 16 doubles stored in column-major format (for compatability with OpenGL), and the following operators (optimized for the constraints given previously):

- `Matrix()`: Initialize the matrix to the identity
- `Matrix(double*)`: Initialize the matrix with the given array
- `double *data()`: Provide a pointer to the data
- `const double *cdata()`: Provide a const pointer to the data
- `double *operator[](int)`: Provide a pointer to the requested column of data
- `Matrix operator*(Matrix)`: Return the multiplication of this matrix with another (taking advantage of the fixed bottom row); $M \cdot N$
- `vec3 GetPos()`: Return the translation of this matrix;$\langle T_x, T_y, T_z \rangle$
- `vec3 operator*(vec3)`: Return this multiplied by a given vector where the homogenous coordinate is 1 (transform a location); $M \cdot \langle \bar{v}_x, \bar{v}_y, \bar{v}_z, 1 \rangle$
- `vec3 operator^(vec3)`: Return this multiplied by a given vector where the homogenous coordinate is 0 (transform a surface normal); $M \cdot \langle \bar{v}_x, \bar{v}_y, \bar{v}_z, 0 \rangle$

- `Matrix Transpose()`: Return the transpose of this matrix; $M^{\mathrm{T}}$; note that the resulting matrix can't be used with this vector library, and is only for compatability with libraries which require matrices to be provided in row-major (rather than column-major) format
- `Matrix Inverse()`: Return the mathematical inverse of this matrix using the algorithm in appendix B; $M^{-1}$

## 5.3 Configuration

### 5.3.1 Persistent Settings Registry (`Registry`)

The `Registry` object simply maintains a persistent registry of configuration information for all components and applications, using a simple key-value pair, stored as a simple flat text file. For simplicity of implementation, rather than deal with different data types, it simply treats everything as a plaintext string. It exposes the following interface:

- `Registry(string)`: Create an instance of a registry attached to the specified filename
- `~Registry()`: Save the registry to disk
- `void SetKey(string k, string v)`: Set the specified key to the specified value
- `void SetKeyFloat(string, float)`: Convert the specified float to a string and set the registry key accordingly
- `void SetKeyInt(string, int)`: Convert the specified int to a string and set the registry key accordingly
- `void SetKeyBool(string, bool)`: Convert the specified bool to a string and set the registry key accordingly
- `string GetKey(string)`: Fetch the specified key as a string
- `float GetKeyFloat(string)`: Fetch the specified key and convert to a float
- `int GetKeyInt(string)`: Fetch the specified key and convert to an int
- `bool GetKeyBool(string)`: Fetch the specified key and convert to a bool
- `bool KeyExists(string)`: Return whether the requested key is already in the registry

### 5.3.2 Per-object Configuration (`Settings`)

The `Settings` class is intended to add simple configuration to various object classes. For simplicify of implementation, it stores all settings as a key/value pair of string and float.

A `Setting` is a struct which simply contains the current value, and constraints for minimum, maximum, and granularity.

The `Settings` class, which is typically inherited by an object which uses it, exposes the following public interface:

- `Setting GetSetting(string)`: Return the setting structure for the given name
- `float SetSetting(string, float)`: Set the specified setting to the given value
- `string GetSettingDesc(string)`: Get the textual description of the given setting key
- `float GetFloatSetting(string)`: Get the value of the specified key
- `bool GetBoolSetting(string)`: Convert the value to a bool (threshold value 0.5) and return it
- `int GetIntSetting(string)`: Convert the value to an int (using rounding) and return it

- `bool ToggleBoolSetting(string)`: Helper function; equivalent to `(SetSetting(foo, !GetBoolSetting(foo)), GetBoolSetting(foo))`
- `set<string> ListSettings()`: Obtain a list of all settings values for this object
- `bool HasCapability()`: Return whether this object expresses a certain capability
- `float GetCapability()`: Return the numerical value of a capability
- `set<string> ListCapabilities()`: Obtain a list of all capabilities
- `void RegisterSettings(Registry *)`: Copy all settings to a `Registry` object
- `void ObtainSettings(Registry *)`: Copy all settings from a `Registry` object

Additionally, it exposes the following protected interface for the objects which inherit from this class:

- `float &NewSetting(string name, float min, float mx, float step, string description)`: Create a new setting with the specified characteristics, returning a reference to the actual value
- `void SetCapability(string name, float val)`: Register a capability with an optional value
- `void RemoveCapability(string name)`: Retract a capability

## 5.4 Object Geometry

### 5.4.1 Image CODEC (`Codec`)

The image CODEC is used for both saving (COmpressing) and loading (DECompressing) two-dimensional images with an arbitrary number of arbitrary-precision fixed-point channels. It provides a handful of algorithms for the compression, namely a non-progressive, lossless compressor, and a progressive, lossy compressor, based on a gradient pseudo-wavelet transform.

It provides the following public methods:

- `Codec()`: Initialize to safe default values
- `void SetSize(int w, int h, int c, int bpc)`: Set up the internal buffers for an image which is `w` by `h` elements big with `c` channels of data, with a data precision of `bpc` bits per channel
- `int Width()`: Return the current width of the image
- `int Height()`: Return the current height of the image
- `int NumChannels()`: Return the current number of data channels in the image
- `int Depth()`: Return the current data precision for the image
- `vector<int> &Channel(int)`: Access the specified data channel
- `vector<int> &operator[](int)`: Synonym for `Channel`, to treat the image as an array of channels at an API level
- `bool Load(int)`: Load an image from the given file descriptor, and return whether the operation was successful
- `bool Save(int, CodecType, void*)`: Save an image to the given file descriptor using the specified compression algorithm (defaulting to the gradient pseudo-wavelet), with an optional set of parameters to give to the compressor
- `bool Complete()`: Return whether a previous `Load` operation resulted in a complete image (opposed to a partial download)

Additionally, these are the private methods:

- `void WriteInt(gzFile, int n, int c)`: Write the lowest `c` bytes of the value `n` to the given compressed file handle
- `int ReadInt(gzFile, int c)`: Read a `c`-byte value from the given compressed file handle
- `void WriteValue(gzFile, int p)`: Write all channels of the `p`th pixel of the image
- `void ReadValue(gzFile, int p)`: Read all channels of the `p`th pixel of the image
- `void Decode_Flat(gzFile)`, `void Encode_Flat(gzFile)`: Functions to load from and save to a compressed file stream, using the lossless compressor
- `void Decode_Wavelet(gzFile, int w, int h)`, `void Encode_Wavelet(gzFile, int w, int h)`: Functions to load from and save to a compressed file stream, using the gradient pseudo-wavelet compressor, performed on the `w` by `h`-pixel subimage

### 5.4.2 Geometry Image (`Vertex`, `GeoImage`)

The `Vertex` class is simply a partitioned 8-dimensional vector which provides the representation of the surface element. Its members are `pos` and `nrm`, `vec3`s which provide the position and surface normal respectively, and `s` and `t`, floats which provide the texture coordinates.

The `GeoImage` class implements a geometry image as a two-dimensional array of vertices with optional topology hints for edge wraparound (indicating whether the mesh is flat, cylindrical, or toroidal in nature), and provides the following public methods:

- `GeoImage()`: Initialize an empty geometry image
- `GeoImage(int, int)`: Initialize a blank geometry image with the specified size
- `GeoImage(GeoImage)`: Initialize a geometry image as a copy of another geometry image
- `operator=(GeoImage)`: Replace the current geometry image with a copy of another geometry image
- `~GeoImage()`: Free the resources taken by this geometry image
- `unsigned int Bytes()`: Return how many bytes of memory this image is really taking
- `void SetSize(int, int)`: Change the image buffer's size without preserving the data
- `int Height()`, `int Width()`: Return the dimensions of this image
- `Vertex *operator[](int)`: Return a pointer to the requested row of data
- `float BoundRadius()`: Return the radius of a bounding sphere positioned at the origin of the geometry image's local coordinate system
- `void BoundPoints(vec3&, vec3&)`: Return the corner points of the AABB of the geometry image
- `void Resize(GeoImage&, float h, float w, FilterType)`: Resize this geometry image, storing the result in the specified other image, using the specified image filter (one of nearest-neighbor, bilinear, or a trilinear approximation of supersampled)
- `GeoImage Resize(float h, float w, FilterType)`: Resize this geometry image, returning the result as a new geometry image
- `void Renormalize(bool)`: Ensure that all surface elements' surface normals are of unit length; if the parameter is true, also ensure that all surface elements' surface normals are outward-facing and conform to the overall surface geometry

- `void Silhouette(vec3, ShadowVolume&, bool cap, bool front)`: Determine the silhouette boundary of this geometry image from the given location in the local coordinate system and store it in the given shadow volume structure; if `cap` is set, also store the endcap polygons which contribute to the shadow volume, and if `front` is set, build the volume using front-facing polygons
- `bool Load(string)`: Load a geometry image from the specified file on disk, replacing the current image. Return whether the load was successful
- `bool Save(string, CodecType)`: Save this geometry image to disk with the specified filename. Return whether the save was successful
- `bool Complete()`: Return whether a previous `Load` operation resulted in a complete file load
- `void SetLODConst()`: Set the projected mesh density for calculating mesh resampling weight
- `float GetLODConst()`: Return the stored relative mesh density
- `float CalcLOD(float dist, int sh, float fov)`: Calculate the resampling factor for this geometry image, `dist` units away from a camera rendered to a `sh`-pixel-tall display with a camera with a `fov`-degree viewing angle
- `float GuessLODConst()`: Try to guess the overall projected mesh density based on a heuristic
- `bool MipMap()`: Reduce the mesh by one MIP-map level using a $2 \times 2$ tent filter
- `GeoImage operator*(float)`: Create a new geometry image by scaling this geometry image by the specified factor
- `GeoImage operator+(GeoImage)`: Create a new geometry image by displacing this geometry image's surface elements by the specified geometry image
- `bool GetWrapH()`, `bool GetWrapV()`: Return the hints for edge wrapping
- `void SetWrapH(bool)`, `void SetWrapV(bool)`: Set the hints for edge wrapping
- Revision control, as per section 5.1

Additionally, `GeoImage` provides one private method, `Copy(GeoImage)`, which copies the specified geometry image's data into this image.

There are also two global methods for geometry images:

- `GeoImage Combine(GeoImage m1, GeoImage m2, float f1, float f2, float nh, float nw, FilterType)`: Morph between two geometry images, using the specified weighting factors, and optional target image size (if unspecified, the image size will be determined by weighing the inputs' sizes by the weighting factors), using the specified filter type for the resampling
- `GeoImage operator*(float, GeoImage)`: Equivalent to `GeoImage::operator*(float)`

### 5.4.3 Texture Map (`Texture`)

A texture map is a 2-dimensional array of image data, with as few as 1 or as many as 4 channels, with their mappings provided in table 5.1. For compatability with OpenGL, the image is stored from bottom to top.

The `Texture` class provides the following public methods:

- `Texture()`: Initialize an empty texture
- `Texture(int w, int h, int c)`: Initialize a blank texture with the given size and number of channels
- `~Texture()`: Free the allocated memory
- `Texture(Texture)`: Copy constructor

Table 5.1: Mappings of channel count to texture format

| Channels | Format |
|---|---|
| 1 | Luminance |
| 2 | Luminance, Alpha |
| 3 | Red, Green, Blue |
| 4 | Red, Green, Blue, Alpha |

- `operator=(Texture)`: Copy operator
- `bool Load(string)`, `bool Save(string, CodecType)`: File manipulation; see `GeoImage`, above
- `bool Load(FILE*)`, `bool Load(int)`: Load from an open file stream or file descriptor
- `bool SavePPM(string)`: Save the current texture in PPM[1] format (in top-to-bottom order)
- `unsigned char *Data()`: Get a pointer to the internal data
- `void SetSize(int w, int h, int c)`: Change the size of the allocated image, destroying the existing data
- `int Width()`, `int Height()`, `int Channels()`: Return the current image size
- `void Scale(int w, int h, bool filter)`: Resize the image, preserving the existing data, optionally with a bilinear filter
- `bool GetFilter()`: Return whether the renderer is allowed to apply a filter to this texture
- `void SetFilter(bool)`: Set whether the renderer is allowed to apply a filter to this texture
- `bool GetMipMap()`: Return whether the renderer is allowed to apply a supersampled reduction filter to this texture
- `bool SetMipMap(bool)`: Set whether the renderer is allowed to apply a supersampled reduction filter to this texture
- Revision control, as per section 5.1

### 5.4.4 Lighting Material (`Color`, `Material`)

A `Color` is simply a 4-tuple of $\langle red, green, blue, alpha \rangle$ stored as public floating-point members `r`, `g`, `b` and `a`, respectively. It also provides the following methods:

- `Color()`: Initialize to solid black; $\langle 0,0,0,1 \rangle$
- `Color(float,float,float,float)`: Initialize to the specified color, defaulting to 1 for alpha if unspecified
- `float *GetGLColor(float *)`: Convert this color to an OpenGL-format color buffer
- `bool operator==(Color)`: Return whether these colors are identical
- `bool operator!=(Color)`: Return whether these colors are different
- `Color operator+(Color)`: Return the summation of these two colors
- `Color operator+=(Color)`: Add the specified color to this color, and return the result

[1]Portable Pixmap, a simple file format for command-line image manipulation in UNIX

**Algorithm 1** Mathematical operations on materials

```
Material eggshell(Color(1,1,0.9), Color(0.01,0.01,0.01), 3); // mostly-matte off-white
Material redPlastic(Color(1,0,0), Color(1,1,1), 50); // shiny red
Material combined = eggshell*0.3 + redPlastic*0.7; // 70% shiny red plastic, 30% eggshell
```

- `Color operator-()`: Return the negation of this color
- `Color operator-(Color)`: Return the subtraction of the specified color from this one
- `Color operator*(Color)`: Return this color modulated by the specified one
- `Color operator*(float)`: Return this color modulated by the specified value
- `float operator^(Color)`: Treat these colors as 4-dimensional vectors and take the dot product

A `Material` is a collection of parameters regarding how a surface responds to light. It has the following components:

- `Color ambient`: How the material modulates ambient light
- `Color diffuse`: How the material modulates diffuse reflections
- `Color specular`: How the material modulates specular reflections
- `float shininess`: How polished (“shiny”) the material is
- `Color emission`: How much light the material gives off

It also provides the following methods:

- `Material()`: Initialize to the default material (shiny white plastic)
- `Material(Color, Color, float, Color, Color)`: Initialize to a material with the following parameters, respectively; diffuse, specular (defaults to white), shininess (defaults to 20), ambient (defaults to diffuse), emission (defaults to none)
- `Material operator*(float)`: Modulate the parameters of the material by the specified amount
- `Material operator+(Material)`: Return the addition of the parameters of two materials

The multiplication and addition methods are provided for simply combining two materials; for example, see algorithm 1.

### 5.4.5 Shader Scripting (`Shader`, `ShaderSet`, `ShaderGroup`)

The `Shader` structure implements a single operation of the shader scripting language as described in section 2.2.2. It provides the following public members:

- `GeoImage *mesh`: A handle to the geometry image to render; Default: none
- `Functor *functor`: A handle to the geometry transform to perform (not yet implemented); Default: none
- `Material mat`: The light reflection model; Default: shiny white plastic
- `Texture *txt`: A handle to the texture map to use; Default: none
- `BlendFunc blendf`: The per-pixel operation for this shader; `Opaque`, `Replace`, `Add`, `Blend`, `Mul`, `Haze`, `Filter`, `Subtract`; Default: `Opaque`

- `RenderFunc renderf`: Which form of the geometry to render; `Solid`, `Wire`, `Outline`, `Shadow`, `ShadowWire`; Default: `Solid`
- `TestFunc alphaf`: The boolean alpha test function for accepting or rejecting a pixel; `Always`, `Never`, `Less`, `Greater`; Default: `Always`
- `float alphat`: The threshold value for an `alphaf` of `Less` or `Greater`; Default: 0.5
- `FaceCull faces`: Which set of faces to render; `Front`, `Back`, `All`; Default: `Front`
- `float thick`: Thickness parameter for line-drawing `RenderFunc`s; Default: 1
- `bool smooth`: Whether to interpolate lighting calculations; Default: true
- `bool lit`: Whether to light the surface (if false, use the diffuse material color); Defuault: true
- `bool depth`: Whether to occlude objects drawn behind this pass at a later time; Default: true

The `Shader` interface also provides two other object classes, `ShaderSet` (a shader script for a single pass) and `ShaderGroup` (a group of shader scripts for all passes).

## 5.5 Resource Management

The resource management portion of the engine implements a reference-counting mechanism for loading resources into memory, and acts as middleware between the application and the actual resource loading.

### 5.5.1 Resource Manager (`Resman`)

The resource manager internally stores two resource allocators (one for meshes, and one for textures), a set of all rendering engines and rasterizers, and a mapping from a resource's handle to its originating URL. The resource manager itself is mostly just a wrapper around the individual resource allocators.

It provides the following public methods (where *`Resource`* indicates either `Texture` or `GeoImage`):

- `Resman()`: Initializes the resource manager.
- `void AddRenderer(Renderer *)`: Register a rendering engine
- `void RemoveRenderer(Renderer *)`: Deregister a rendering engine
- `const RendererSet& GetRenderers()`: Return a list of all registered rendering engines
- `void AddDriver(Driver *)`: Register a rasterizer
- `void RemoveDriver(Driver *)`: Deregister a rasterizer
- `const RendererSet& GetRenderers()`: Return a list of all registered rasterizers
- *`Resource`* `*Load`*`Resource`*`(URL, bool)`: Load the resource at the specified URL, optionally forcing a reload
- *`Resource`* `*AddRef(`*`Resource`*` *)`: Attach a handle to a resource
- *`Resource`* `*Add`*`Resource`*`(URL)`: Get a handle to the resource at the specified URL; a synonym for `AddRef(Load`*`Resource`*`(...))`
- `bool DelRef(`*`Resource`*` *)`: Delete a resource handle and return whether this caused the resource to be purged
- `int QueryRef(`*`Resource`*` *)`: Return the number of handles there are to the specified resource
- `void Reload(URL)`: Force the resource manager to reload and update the resource at the specified URL

- `URL GetUrl(`*`Resource`*` *)`: Return the URL which a resource was loaded from
- `void DelRefs(ShaderSet)`: Delete all resource handles held by a shader script
- `void DelRefs(Shader)`: Delete all resource handles held by a single shader part
- `void Purge(Node *)`: Notify all rendering engines that a scenegraph node has been deleted. (This is provided as a helper function to the application, even though it doesn't fit in with the purpose of this module, simply because the resource manager already has the code necessary for this purpose.)

This module also implements the following private method:

- `PurgeNotify(`*`Resource`*` *)`: Notify all drivers and renderers of a resource's purge, and erase the stored URL mapping for the handle

### 5.5.2 Resource Allocator (`ResmanGC`)

The resource allocator is the module which actually handles the loading and garbage collecting of the different resource types. It is a template class with template parameter `XX`, which specifies the kind of resource which it serves as an allocator for. It maintains an internal reference count for each resource type, and provides the following interface:

- `ResmanGC()`: Initialize internal state
- `void SetResName(string)`: Set the English name of the resource it's managing
- `~ResmanGC()`: Make sure that all of its managed resources are properly disposed of, printing a warning for any resources which are still in memory (indicating a memory leak in the application)
- `XX *Load(URL, bool)`: Actually perform the loading operation on the given URL for the resource manager
- `XX *AddRef(XX *)`: Actually perform the duplication of a handle
- `bool DelRef(XX *)`: Actually perform the deletion of a handle
- `int QueryRef(XX *)`: Actually perform the querying of handle counts
- `void Pump()`: Work on previously incomplete loads of a resource (for progressive resource loading)

## 5.6 Scene Hierarchy

### 5.6.1 Spatial Partitioning (`Octree`)

The `Octree` class implements dynamic spatial partitioning using a dynamic octree[3]. It provides the following public methods:

- `Octree(Octree *)`: Create a new octree node with the specified parent (which defaults to `NULL`)
- `~Octree()`: Destroy this octree node and its associated hierarchy
- `void GetExtents(vec3&, vec3&, bool)`: Get the AABB

### 5.6.2 Scene Node (`Node`)

A `Node` is a single node within the scene hierarchy. It consists of a base geometry image, configuration about its location, orientation, and visibility, a named set of child objects, and the necessary information to work with a dynamic spatial partitioning.

It provides the following public methods:

- Initialization
  - `Node()`: Initialize an empty node
  - `Node(GeoImage *)`: Initialize an empty node and set the base geometry image
  - `~Node()`: Clean up
- Geometry
  - `float Radius(bool)`: Return the radius of the origin-centered bounding sphere which contains either just this node (if the parameter is false) or this node and all its children (if the parameter is true)
  - `void SetBaseGeoImage(GeoImage *)`: Set the base geometry image for this node
  - `GeoImage *GetBaseGeoImage()`: Get the base geometry image
- Configuration
  - `void SetPos(vec3)`, `vec3 GetPos()`: Set/get the origin of this object within its parent's coordinate space
  - `void SetPoint(vec3)`, `vec3 GetPoint()`: Set/get the direction which the object should point in within the parent's coordinate space
  - `void SetUp(vec3)`, `vec3 GetUp()`: Set/get the direction which the object should vertically orient towards within the parent's coordinate space
  - `void SetRot(vec3)`: Helper function to `SetUp()` and `SetPoint()` based on a HPR vector
  - `void SetVisible(bool)`, `bool GetVisible()`: Set/get whether this object should be rendered
  - `void SetShadows(bool)`, `bool GetShadows()`: Set/get whether this object should cast a shadow
  - `void SetActive(bool)`, `bool GetActive()`: Set/get whether this object's hierarchy should be rendered
  - `void SetBillboard(bool)`, `bool GetBillboard()`: Set/get whether this object should always be oriented towards the viewpoint
  - `Matrix TransMatrix()`: Compute the transformation matrix given this object's orientation as per `SetUp()` and `SetPoint()`
- Rendering:
  - `void AddShader(Shader::Pass, Shader)`: Add the specified shader operation to the specified pass
  - `ShaderSet &GetShader(Shader::Pass)`: Get the entire shader script for the specified pass
  - `ShaderGroup &GetShaders()`: Get all shader scripts for all passes
  - `bool HasShader(Shader::Pass)`: Return whether this object has a shader script for the specified pass
- Hierarchy:
  - `void AddChild(string, Node *)`: Add this node as a child object with the specified name
  - `Node * const &operator[](string)`: Get a handle to the child of a given name

  - `const Children &GetChildren()`: Directly access this node's immediate descendents in the hierarchy
  - `bool RemoveChild(string)`: Remove the specified child from the hierarchy, and return whether it was successful
  - `Node *GetChild(string)`: Return a handle to a child if it exists, NULL if it doesn't
  - `unsigned int NumChildren()`: Return the number of immediate children

- Visibility:
  - `Octree *GetOctree()`: Get the root of this node's spatial partitioning octree
  - `Octree *GetOctreePos()`: Get the octree node which this scene node is situated within
  - `void SetOctreePos(Octree *)`: Notify this scene node of which octree node it is situated in

- Revision control, as per section 5.1, with the following additions:
  - `void TouchChildren()`: Call `Touch()` on all of this object's children within the scene hierarchy

`Node` also implements the following private methods:

- `void Init(GeoImage *)`: Initialize the values of this object, with the specified base geometry image
- `void SetParent(Node *)`: Set the parent object within the scene hierarchy
- `void UpdateRadius(float)`: Recalculate the bounding radius; if a value is given, use the maximum of the current radius and the value (for updating a single child), otherwise use the maximum of the current radius and all of the childrens' radii
- `void UpdateChild(Node *, vec3 pos, vec3 rad)`: Update the specified child, which used to be at the given position and had the given radius
- `void UpdateParent(vec3 pos, float rad)`: Update our parent of a change in position or radius, using the specified old values

### 5.6.3 Lights (`Light`)

A `Light` is a data structure which specifies a point in space from which illumination should be cast, and the properties of the light which it emits. It provides the following public data members:

- `bool active`: Whether this light should actually illuminate; Default: true
- `bool shadows`: Whether this light should cast shadows; Default; true
- `vec3 pos`: The position within the current room's coordinate space; Default: $\langle 0,0,0\rangle$
- `Color color`: The color emitted by this light; Default: white
- `Color ambient`: The per-light ambient light (which can still be shadowed); Default: none
- `Color globalAmbient`: The contribution to the global ambient light made by this light; Default: none
- `vec3 falloff`: The equation for calculating the rate at which this light's contribution falls off with respect to the distance $r$, where the equation for the attenuation factor $F(r)$ given a vector $v$ is encoded as $\frac{1}{F(r)} = v_x + v_y r + v_z r^2$; Default: $\langle 1,0,0.1\rangle \Rightarrow \frac{1}{F(r)} = 1 + 0.01r^2$, an aesthetically-pleasing balance between constant attenuation and reality's inverse-square law
- `float shadowlen`: The length which shadow volumes will be extruded to

- `vec3 pointat`: The location in the room's coordinate system which this light will be directed towards; Default: $\langle 0,0,0\rangle$
- `float angle`: The angular field of the spotlight, in degrees; Default: 180
- `float focus`: The focal intensity of the spotlight, such that 0 is uniform coverage and $\infty$ is an infinitely-small pinprick; Default: 0

Additionally, there are two methods for dealing with this light's shadow-pass shader:

- `void AddShader(Shader)`: Add a shader operation to this light's shadow pass
- `ShaderSet &GetShader()`: Get the complete shader script for this light's shadow pass

### 5.6.4 Rooms (`Room`)

A `Room` is a special `Node` which can be used for the root of a scene hierarchy. It inherits from `Node`, and adds the following functionality:

- `void AddLight(string, Light *)`: Add a light to this room with a given name
- `LightSet GetLights()`: Get all of the lights in this room
- `bool RemoveLight(string)`: Remove the light of the given name, and return whether the operation was successful
- `Light *&GetLight(string)`: Get a specific light
- `void SetAmbient(Color)`, `Color GetAmbient()`: Set/get the global ambient light for this room

### 5.6.5 Viewpoint (`Camera`)

The `Camera` object is a data structure which encapsulates the minimal amount of information needed to render a 3-dimensional scene in perspective. It contains the following public members:

- `vec3 pos`: The vantage point of the camera within the global coordinate system; Default: $\langle 0,0,0\rangle$
- `vec3 lookat`: The point towards which the camera looks within the global coordinate system; Default: $\langle 0,0,-1\rangle$
- `vec3 up`: The direction which the camera tries to most closely match its vertical axis to; Default: $\langle 0,1,0\rangle$
- `float fov`: The vertical field of view of the camera; Default: 60
- `float aspect`: The pixel aspect ratio of the final display; Default: 1 (square pixels)
- `float hith`: The furthest distance away from the camera which should be considered visible; Default: $10^6$
- `float yon`: The closest distance away from the camera which should be considered visible; Default: 0.1

## 5.7 Renderer

### 5.7.1 Rendering Engine (`Renderer`)

The `Renderer` object inherits from `Settings`, and provides a simple interface for performing the complex task of rendering the scene.

Its public interface is as follows:

- `Renderer()`: Configure the inherited `Settings` object with the configuration variables described in section D.1, and create two internal `Visibility` objects, one for camera visibility and one for light visibility

- `~Renderer()`: Delete the visibility structures
- `void Start(Driver *)`: Start rendering to the specified rasterizer
- `void Render(Room *, Camera)`: Render the specified scene hierarchy from the given viewpoint
- `void Finish()`: Finish any rendering which has been deferred to the end (e.g. the blend pass)
- `float Stat(string)`: Get an accumulated statistics value
- `void Purge(GeoImage *)`: Purge the specified mesh from all internal caches
- `void Purge(Texture *)`: Purge the specified texture from all internal caches
- `void Purge(Node *)`: Purge the specified scene node from all internal caches

The private interface is much more complex:

- `void PushMatrix(Matrix)`: Push the given transformation matrix onto the internal matrix stack
- `void PushMatrix(Node *)`: Determine the transformation matrix for the specified scene node, multiply it by the current transformation matrix, and push the result onto the stack
- `void PopMatrix()`: Pop the topmost transformation matrix off of the stack
- `vec3 Project(vec3)`: Apply the current transformation to a point
- `vec3 Unproject(vec3)`: Reverse the current transformation from a point
- `GeoImage *GetGeoImage(Node *, Shader &, Matrix *, float detail, float maxDetail, float maxPolys)`: Given a node, a shader part, and various characteristics for a resampling, determine the level of detail reduction for a mesh and return the resampled mesh, which is cached if possible
- void OcclusionPass(PrevisPass, Occlusion *, bool deferLight): Render the given previous-visibility pass and add its objects to the given occlusion buffer, optionally with the

#### 5.7.1.1 Supplemental object classes

- `DeferredNode`: A scene node which has been deferred for later; a pair of transformation matrix and `Node*`
- `DeferredPass`: A collection of `DeferredNode`s, sorted by distance from the camera
- `PrevisPass`: An unsorted collection of transformation matrices and `Node*`s for storing the temporal coherence of visibility determination
- `PrevisSet`: A collection of all `PrevisPass`es for all currently-active `Rooms`
- `LightContext`: A collection of active lights as given by a `Room`
- `LODcache`: A cache of resampled mesh data

### 5.7.2 Visibility Determination (`Visibility`)

### 5.7.3 Occlusion Buffer (`Occlusion`)

#### 5.7.3.1 OpenGL Hardware-Assisted Occlusion (`OcclusionGL`)

## 5.8 Rasterizer (`Driver`)

### 5.8.1 OpenGL Backend (`DriverGL`)

### 5.8.2 XLib Backend (`DriverXLib`)

## 5.9 Application

### 5.9.1 Basic Rendering Test (`testRender`)

### 5.9.2 Command processor (`Command`)

# Chapter 6

# Future Work

virtualized lighting improvements
- anisotropic mesh filtering
- implement mesh functors
- implement display layer
- rewrite image handling
- shadow mapping instead of shadow volumes
- rework configuration layer

# Appendix A

# Glossary

| | |
|---|---|
| **AABB** | Axis-oriented Bounding Box; a rectangular region of space which is oriented parallel to the coordinate system's axes, defined by its two extremal corner points |
| **billboard** | An object which is always oriented towards the camera; usually a flat surface with a texture painted on it to give the impression of a detailed object |
| **CODEC** | COmpressor/DECompressor; a pair of algorithms used to make a compact (and sometimes approximate) representation of a chunk of data, and to reconstitute the original data |
| **HPR** | Heading/Pitch/Roll; a means of orienting an object based on relative angular measures |
| **octree** | A recursive data structure formed by partitioning a region of space into 8 pieces by cutting it along each axis plane; the 3-dimensional equivalent of a binary tree |
| **orthonormal** | A transformation where the basis vectors are orthogonal (perpendicular to each other) and normal (of unit length) |
| **scenegraph** | The graph structure formed by the linkages between objects in a hierarchally-defined scene |
| **shader** | A general term for a programmable rendering architecture |

# Appendix B

# Fast Orthonormal Matrix Inversion

Given a matrix $M$, its inverse $M^{-1}$ is such that $M \cdot M^{-1} = I$. Because computing $M^{-1}$ for a generalized matrix is both time-consuming and not necessarily defined, it is normally desirable to avoid finding the inverse of a matrix, which creates severe problems for certain operations which are performed repeatedly in a modern 3D engine (for example, back-projecting the camera location into an object's local coordinate space). The traditional solution is to simply maintain the inverse of the transformation matrix in a separate stack by performing the inverse transformation.

However, orthonormal transformation matrices have a number of convenient properties which make calculating the inverse a simple and fast operation which is significantly faster than performing the inverse transformation, and only has to be applied when the inverse is actually required.

In order to perform this operation, we first decompose the orthonormal transformation matrix $M$ into a partitioned matrix comprised of its rotation part $R$ and its translation part $T$:

$$M = \begin{bmatrix} R & T \\ 0 & I \end{bmatrix}$$

Similarly, we decompose $M^{-1}$:

$$M^{-1} = \begin{bmatrix} R' & T' \\ 0 & I \end{bmatrix}$$

Given these decompositions, we can restate the relationship between $M$ and $M^{-1}$ as

$$\begin{aligned} M \cdot M^{-1} &= I \\ \begin{bmatrix} R & T \\ 0 & I \end{bmatrix} \begin{bmatrix} R' & T' \\ 0 & I \end{bmatrix} &= \begin{bmatrix} I & 0 \\ 0 & I \end{bmatrix} \\ \begin{bmatrix} R \cdot R' & R \cdot T' + T \\ 0 & I \end{bmatrix} &= \begin{bmatrix} I & 0 \\ 0 & I \end{bmatrix} \end{aligned}$$

Thus, to solve the inverse equation, we must only solve two simple equations,

$$\begin{aligned} R \cdot R' &= I \\ R \cdot T' + T &= 0 \end{aligned}$$

The first equation is quite simple. As $R \cdot R' = I$, $R' = R^{-1}$. As $R$ is an orthonormal rotation, it means that the vector defined by row $i$ will be either orthogonal to the vector defined by column $j$ (if $i \neq j$) or equal (if $i = j$). Furthermore, the magnitude of a row or column vector will always be 1. Thus, $R \cdot R^{\mathrm{T}} = I$, and therefore $R' = R^{-1} = R^{\mathrm{T}}$.

The second equation is nearly as simple to solve with simple algebra, making use of the relationship above;

$$\begin{aligned} R \cdot T' + T &= 0 \\ R \cdot T' &= -T \\ T' &= -T \cdot R^{-1} \\ T' &= -T \cdot R^{\mathrm{T}} \end{aligned}$$

**Algorithm 2** Calculating the inverse of an orthonormal transformation matrix

```
Matrix Matrix::Inverse() const {
    Matrix ret;
    double *r = ret.data();
    const double *a = cdata();

    // Calculate inverse of rotation part
    r[ 0] = a[ 0];
    r[ 1] = a[ 4];
    r[ 2] = a[ 8];

    r[ 4] = a[ 1];
    r[ 5] = a[ 5];
    r[ 6] = a[ 9];

    r[ 8] = a[ 2];
    r[ 9] = a[ 6];
    r[10] = a[10];

    // Calculate inverse of translation part
    r[12] = -(r[ 0]*a[12] + r[ 4]*a[13] + r[ 8]*a[14]);
    r[13] = -(r[ 1]*a[12] + r[ 5]*a[13] + r[ 9]*a[14]);
    r[14] = -(r[ 2]*a[12] + r[ 6]*a[13] + r[10]*a[14]);

    return ret;
}
```

Finally, we roll these two computations back together into a single partitioned matrix:

$$M^{-1} = \begin{bmatrix} R^{\mathrm{T}} & -T \cdot R^{\mathrm{T}} \\ 0 & I \end{bmatrix}$$

The actual implementation is quite simple; $R^{\mathrm{T}}$ requires only 9 assignments, and $-T \cdot R^{\mathrm{T}}$requires only 9 multiplies, 6 additions, 3 negations, and 3 assignments. The final implementation is given in algorithm 2.

From this derivation, it is simple to extend the calculation to orthogonal transformations which may involve isometric scaling (i.e. it is scaled equally in all directions); assuming that $R$ is truly orthogonal and isometric, then $R \cdot R^{\mathrm{T}} = \alpha I$ where $\alpha$ can be determined by taking the dot product of a matching row and column of $R$ (for example, the first row by the first column). From there, we derive:

$$\begin{aligned} R \cdot R^{\mathrm{T}} &= \alpha \cdot I \\ R^{\mathrm{T}} &= \alpha \cdot R^{-1} \\ R^{-1} &= \frac{R^{\mathrm{T}}}{\alpha} \end{aligned}$$

Thus, adding isometric scaling to the matrix inverse requires finding $\frac{1}{\alpha}$ (which involves 3 multiplies, 2 adds, 1 division, and 1 assignment) and multiplying $R^{\mathrm{T}}$by it (which involves 9 multiplies).

# Appendix C

# Command Set

What follows is a list of commands for the simple command processor (`Command` object).

# Appendix D

# Renderer and Driver Settings

## D.1 Renderer

## D.2 Driver

### D.2.1 DriverGL

### D.2.2 DriverXLib

# Bibliography

[1] Xianfeng Gu, Steven J. Gortler, and Hugues Hoppe. Geometry images. *ACM SIGGRAPH 2002*, pages 355–361, July 2002.

[2] Joshua Shagam and Joseph Pfeiffer Jr. Complex geometry tasks made simple with geometry images. *Publication pending*.

[3] Joshua Shagam and Joseph Pfeiffer Jr. Dynamic spatial partitioning for real-time visibility determination. *Publication pending*.