# *HoloWeb*

***A Proposal for VRML II***

*Sun Microsystems*

*Sunday, Feb 4, 1996*

---

# Contents

---

# Chapter 1 - HoloWeb

## 1.1 The stuff that you'd expect to see first

HoloWeb is a binary file format and run-time API for real-time animated 3D worlds. HoloWeb files' native habitat is the Internet, where they are automatically connected together to form a seamless virtual universe; HoloWeb forms the foundation of 3D cyberspace in the truest sense of the term.

The existing Internet display metaphor is 2D textual pages with occasional attachments of sound or images. The navigation metaphor is jumping from node to node in a discrete tree.

The HoloWeb display metaphor is 3D cities of information. Within these virtual 3D cities, home pages are animated 3D buildings. The traversal metaphor is walking in 3D past or into these buildings. Related home buildings are geographically neighbored; to find all the major vendors of data base software, you walk down Data Base Street.

HoloWeb is not future technology, prototype authoring tools and run-time browsers are up and running in real-time today.

HoloWeb brings considerable new technology to the Internet, it advances the state of the art in several areas. These include:

- First significant use of Java as an integrated real-time 3D animation scripting language.
- Use of 3D geometry compression as a universal 3D geometry format, achieving 10 to 1 typical reductions in file size.
- Use of 768-bit 3D universe coordinates to effectively support *very* large virtual universes.
- Sophisticated viewing model supporting the gamut of today's and tomorrows display devices and

interfaces.

HoloWeb takes a RISC philosophy to graphics features. The only geometry types supported are compressed 3D dots, lines, and (textured) triangles, and text. A whole myriad of legacy application graphics features are simply not supported: dashed lines, annotation text, markers, model clipping, 2D subsets, etc. HoloWeb builds its low rendering semantics on a sub-set of industry standard OpenGL.

The rest of this document details the format and features of HoloWeb, spiced with a few dashes of philosophy as needed.

---

**Warning -** This is an early version of the HoloWeb specification. This specification mainly addresses issues of semantics independent from issues of syntax. The frequently mentioned binary and text syntax Appendices do not exist (except for extensive coverage of geometry compression). Several other VRML proposals exist which address the syntax issues in ways that are compatible with this specification.

---

**Caution -** Several areas are woefully incomplete. Many of these involve minor features, but others are larger issues that will need to be addressed soon. The intent of releasing this early specification is to expose many of the HoloWeb concepts, and start a dialog with other VRML developers.

---

# Chapter 2 - Introduction

## 2.1 Technical Introduction

**File format and API**

HoloWeb is a binary file format and run-time API for optimized real-time animated 3D worlds. It supports the use of Java as a run-time animation control language, and uses geometry compression as a standard method of transport. HoloWeb is optimized for use with efficient real-time 3D browsers, it is *not* just a geometry interchange format.

**Transports and displays pre-created geometry**

HoloWeb does *not* support complex modeling primitives. HoloWeb assumes that 3D authoring packages and digitizing services have created the complex 3D geometry of the future Web; HoloWeb is geared for efficient transport and display of pre-created geometry.

**Not a 3D browser**

HoloWeb is not a 3D browser, but a transport and interchange format for such browsers, and also specifies a run-time Java 3D API that such browsers must support. The philosophy was to leave as many implementation decisions as possible up to the run-time browser, while establishing several important policies to enable the creation of complex networked virtual universes.

**Multiple simultaneously active and viewable files**

HoloWeb is designed to support multiple HoloWeb files simultaneously active and viewable. Indeed one of the driving principles behind the HoloWeb design is to support virtual cities of information, where each building is a separate network connected file.

**Multi-participant shared virtual universe**

HoloWeb also has support for multi-user interaction within a shared virtual universe.

**Compressed binary file format**

The HoloWeb binary format is a sequence of byte-codes and embedded compressed data, similar to the Java byte-code format. An equivalent text format is also defined.

This document introduces the core concepts of HoloWeb, and documents their semantics.

# 2.2 HoloWeb Goals

A network interchange language for optimized 3D virtual worlds.

> **Goal:** Keep run-time browser requirements as simple as possible.
>
> **Goal:** Leave as many implementation details as possible up to the run-time browser.
>
> **Goal:** Minimize file size.
>
> **Non-Goal:** Authorable by humans using text editors.
>
> **Non-Goal:** HOLOWEB is *not* a modler interchange format, it supports *viewers.*
>
> **Goal:** Freely allow mixing of portions of virtual worlds authored separately.

## 2.2.1 768-bit Universal Coordinate system

HoloWeb defines a large virtual universe, several billion light years in size. Points within this universe are address by 768-bit xyz coordinates, or by more conventional floating point coordinates relative to a known local coordinate system (itself defined by using 768-bit coordinates). Within this space link objects connect to second level HoloWeb files or network links.

## 2.2.2 Java Programs as 3D Objects

Java programs are fundamentally intertwined with 3D geometry. In fact, each piece of java code has a spatial coordinate, and arguments to java programs are usually pointers to 3D objects within the universe. By having spatial coordinates, Java code objects can be spatially managed for access in the same way as other 3D objects.

# 2.3 Key HoloWeb Elements

**Geometry**

Just compressed geometry (triangles, textured triangles, lines, points). No cylinders, spheres, cones, NURBS, etc. are directly supported in the base definition. This is a directly consequence of HoloWeb's non-goal of being a model authoring language. However, Java can be used to extend the geometric repertoire at run time.

**Text**

HoloWeb supports 3D text objects.

**Rasters, texture maps**

Via WWW conventions, plus ...

**VR based viewing model**

HoloWeb leaves most details of the viewing model to the browser and the users' local display configuration. Thus the viewing model is de-coupled from HoloWeb applications, allowing the same applications to run without change on multiple different viewing configurations. The portions of the viewing model exposed by HoloWeb are defined to enable the support of the gamut of modern display devices, from standard CRT's to VR HMD's.

**Render semantics compatible with OpenGL**

HoloWeb defines a simple 3D rendering model, compatible with the industry standard OpenGL model. Further sub-setting conventions need to be defined for very low end platforms.

**Elemental Animation Operators**

Not all HoloWeb behavior needs to be programmed point by point in Java. A simple lower level set of animation basis functions (called *elemental animation operators*) are provided for both programmer and run-time efficiency. This includes simple piecewise curve functions of time of: orientation, position, orientation + position, scale, color, and opacity. Temporal loops support loops over arbitrary sub-objects; Movies support optimized loops over 2D images. Sounds support loops over audio data. Other elemental animation objects include "switch" functionality, and level of detail (LOD) objects.

**Java Control of Geometry and Elemental Animation Operators**

The fundamental unit of animation control in HoloWeb is the Java process. Individual Java processes orchestrate local animation sequencing via messages to geometry and elemental animation operators. In this way the control is localized with the objects being controlled. A typical HoloWeb applet may have dozens to hundreds of individual Java control processes, though only a handful will be active during a particular frame time.

**768-bit Universal Coordinate system**

High-resolution coordinates with 256-bit X, 256-bit Y, and 256-bit Z components.

Sub-files attached to 3D anchor points.

Support for simple collision detection.

**Audio & 3D sound**

Audio objects are part of the scene hierarchy, and can be processed as spatialized audio.

**Suggested 3D Browser Navigation interface**

Common hook points

"DOOM‘" mode for fast clients

"Myst‘" mode for slow clients

**Multi-platform support**

Local client decompresses geometry into appropriate local format.

Compiles Java behaviors into local machine code.

# 2.4 Organization of Document

This document has to describe HoloWeb at both the semantic and bit level. Issues of syntax have been intentionally separated from semantic details; thus the majority of this document concerns itself with semantics; issues of binary and text syntax have been regulated to appendices. A layered approach is taken, starting at the fundamental semantics and progressively refining.

**Object classes and data base organization**

Defines the fundamental HoloWeb object classes and data base organization, for both the transport format and run-time API.

**Coordinate systems and space**

Defines HoloWeb’s coordinate system semantics, geometric transformation semantics, and introduces the concept of high resolution 256-bit component coordinates: universe coordinates.

**HoloWeb view model**

Defines HoloWeb's 3D view model.

**Base Object Components**

Defines data elements common to all HoloWeb objects.

**Elemental Animation Operators**

Defines the semantics and components of HoloWeb's basic animation objects. First the abstract base class is defined, and then the specific operators.

**Avatars and Aspects**

Defines special objects for controlling the environment and the viewer.

**Java interface routines for controlling HoloWeb objects**

Defines the Java routines for controlling HoloWeb objects.

**3D Geometry Compression**

Defines HoloWeb's 3D geometry primitives and the geometry compression format. Also describes the process of compressing and decompressing 3D geometry.

**3D render semantics**

Defines additional details of the render semantics of HoloWeb objects.

**HoloWeb binary format**

Defines the bit level details of the HoloWeb file format.

---

# Chapter 3 - Object Classes and Data Base Organization

## 3.1 HoloWeb Object Classes

A HoloWeb file describes a directed graph of group nodes and leaf objects. It should be noted that much of this organization is for transport purposes; local browsers can re-organize the data however they choose, so long as the animation and display semantics are preserved.

In HoloWeb *all* objects have a location as part of their base structure. This includes Java behavior code, material and texture map references, prototype definitions, etc. The reason for this is that all semantics in HoloWeb is spatially scoped. This policy allows bounded computation on nearly unbounded virtual spaces to be formally specified.

Objects use their location information in different ways, in the discussion individual uses will be described for each object class.

All objects also have *annulation flags*. Values of this field contain "hints" or mandatory information about the object; a common example is TRANSFORM_CONSTANT, which indicates that the location information contained within the object itself will be constant for all time, and that a browser implementation is free to optimize it away.

**Group Nodes**

All non-leaf nodes are group nodes. The annotation flags field of group nodes can contain information to indicate their function in the hierarchy: as container for animation objects, grouping for data base organization purposes, etc. Most group node functions are handled by a single type of group node, appropriately named *group*. Beyond simple annotations, more complex extensions of group semantics is performed by indicating that the first child of a group is special, and contains additional group information. The general class of group extension nodes are *elemental animation operator* nodes.

Other types of group nodes exist. One is *HRoot* nodes; these are high resolution universal coordinate nodes. They are the only type of node that has no parent. HRoot nodes primary function is for rooting structure in the virtual world; most specialized group semantics do not apply to them. Two other types of group nodes are *Prototype* and *Instance* nodes, described below.

**Prototype, instance nodes**

A prototype node is head of a parameterized structure of nodes that can be later instanced by providing instance data for the parameters. Note that prototype nodes are *not* root nodes; prototype nodes have to have a parent node and a location in space. This location (along with the local scale value) define the sphere of influence of the prototype, outside of this sphere the prototype cannot be instanced.

**3D geometry**

The *geom* node contains 3D geometry, possibly compressed. The only geometry types supported are points, line, and triangles. Only triangles are subject to lighting. The semantics of this node is to cause the geometry to be displayed.

Several other specialized classes of leaf nodes support control information.

**Elemental animation operators**

One such special leaf node class are *elemental animation operator* nodes. If an elemental animation operator leaf node is present as the first element of a group node (and indicated in the annotation flags field of that group node), then the semantics of that group node is modified by the operator node. Elemental animation operators allow the contents of a group to be spun, flashed, etc. under control of the operator. Other functions include level of detail.

**Java processes**

Another specialized leaf node type is a *Java* node. This node expresses an instance of a Java process. An instance is parameterized by an argument list of back pointers to other nodes in the hierarchy that the Java code will send messages to. Java nodes generally receive messages by blocking on desired events, like an avatar approaching, a button being pressed, a time-out, etc.

**Aspects**

Another special class of leaf nodes are *aspects*. Aspect objects are handles on a number of environmental attributes, including light sources, background colors, textures, fog, clipping, etc.

**Avatar Souls**

One more special leaf node type is *avatar_soul*. Avatar soul objects are moveable viewing platforms that constrain the HoloWeb equivalent of the view camera. They are the primitive upon which programmable virtual vehicles can be built by applets.

A related object is the *TeleportTarget*. It is used to define "teleport" destination position/ orientation/ scale/center values.

**Image data, sound data, text data**

HoloWeb supports importation of non 3D geometry into the rendering semantics. 2D image data, in a variety of network formats, may be input either as a texture map definition, or as raster or sequence of rasters in space (the *movie* object.). Sound data import and acoustic rendering is supported by the *sound* object, text data import and rendering as special 3D geometry is supported by the *text* object.

**Links**

Another node is the *link* node. This is either a file or an Internet indirection to an additional HoloWeb file. This is *not* a TeleportTarget, rather it is a pointer to geometry and behavior that should be read in whenever an active avatar travels to within a specified radius of the link object's location in space. It is these link nodes that allow large virtual universes to be constructed across the Internet using HoloWeb.

# 3.2 HoloWeb Data Base Organization

As a general philosophy, HoloWeb strives to allow all node hierarchy information for a particular file to be completely defined at file load time. This allows browsers to convert "scene-graph" information into more optimized internal forms. To this end, all HoloWeb node annotation flag fields support the annotations of WRITE_DISABLED and READ_DISABLED. When found to be set, a browser knows that it can safely convert the node data into implementation specific forms.

One of the most important classes of optimization that most browsers will support is *culling*. In general, culling uses special cached data structures to rapidly mark portions of the node hierarchy as not needing any processing or display for a particular frame, thus vastly reducing the computation and render load. Specific forms of culling include *viewport culling*, where objects not within the viewers field of view need not be rendered. (These are objects to one side, above, below, or behind the viewer, or beyond the far clipping plane of the viewer.) Another form of culling is *occlusion culling*, where objects that are completely occluded by closer objects need not be rendered. There is also *execution culling*, where a Java node or elemental animation operator node can be determined to be too far away to effect the current display, and can be skipped from processing. Similar distance based culling can apply to light sources and sound sources.

Many of the architectural decisions made in the design of HoloWeb were made to support these and other forms of culling. For example, non-uniform scale transformations (in addition to a host of other problems) make impossible most viewport culling algorithms based on bounding spheres. Another decision to support culling is the near total lack of render attribute information other than at leaf nodes, other than transform attributes and color and opacity elemental animation operators. This allows a culler to safely by-pass non- visible portions of the node hierarchy without worrying about skipping over some required attribute setting. This also allows cullers to traverse those portions of the node hierarchy that are potentially visible in any order that suits the culler; it is explicitly not necessary for nodes to be visited in the same order that they appeared in the input file.

Sometimes to achieve a particular semantic action, it is impossible to avoid something that looks very much like dynamically re-arranging the node hierarchy. A common case is when an object needs to come under the influence of a controlling Java code or elemental animation object. HoloWeb has too mechanisms to support this short of dynamitic node attachment editing: Java threads and constraint objects.

Inside a Java thread, Java code can dynamically get a handle on HoloWeb objects that were not part of its original parameter list through a number of mechanisms. One of these are *proximity spheres*, sort of 3D fishing nets, where Java threads can get a handle on objects that come within the sphere, so long as they are not annotated READ_DISABLED. Once a Java thread has a pointer to a HoloWeb node, it can both send and receive messages to/from it. If the object is not annotated WRITE_DISABLED, the Java thread can modify many aspects of the objects internal state (subject to additional layers of WRITE_DISABLED protection on individual fields). Because it is the browser that obtains these handles for the Java thread in the first place, an optimizing browser can keep track of what objects might be effected by a particular Java thread. (There is a sub-issue about when the browser knows that the Java thread has relinquished the handle.)

The second mechanism allows elemental animation operators to attach and detach copies of their influence to objects dynamically. The mechanism here is called a *constraint*. An example of this would be an elevator or conveyer belt, which, while an object is within their field of influence, needs to assert some control over the objects location.

---

# Chapter 4 - Coordinate Systems and Space

# 4.1 Universe Coordinate System

**Space: the final frontier**

Double precision floating point, single precision floating point, or even fixed- point representing of three dimensional coordinates are sufficient to represent and display rich 3D scenes. Unfortunately, scenes are not worlds, let alone universes. If one ventures even a hundred miles from single precision (32-bit) floating point virtual world "0 0 0", representable points become quite quantized, to at very best a third of an inch (and much much more coarsely than that in practice).

If one wishes to "shrink" down to a small size (say the size of an IC transistor), even very near 0 0 0, the same problem arises. Such shrinkage very far from 0 0 0 is impossible.

If a large contiguous virtual universe is to be supported, then some form of higher resolution addressing is required. Thus the choice of 256-bit positional components for "high resolution" positions.

**HoloWeb Universe Coordinates**

A HoloWeb universe coordinate consists of three (signed, two's complement) 256-bit fixed point numbers, one each for X, Y, and Z. The fixed point is at bit 128, and the value 1.0 is defined to be exactly one centimeter. This is sufficient to describe a universe in excess of several hundred billion light years across, yet still define objects smaller than a proton (down to below the plank length). The table below shows how many bits are needed above or below the fixed point to represent the range of interesting physical dimensions.

| 2n centimeters | Unit |
|---|---|
| 93.933 | universe (20 billion light years) |
| 59.715 | light year |
| 43.766 | au |
| 23.602 | earth diameter |
| 17.296 | mile |
| 16.610 | km |
| 7.49 | human |
| 1.0 | cm |
| -13.287 | micron |
| -26.575 | angstrom |
| -36.837 | proton |
| -53.102 | electron |
| -108.931 | planck length |

A 256 bit fixed-point number also has the advantage of being able to directly represent nearly any reasonable single precision floating point value *exactly*.

Universe coordinates are only used in HoloWeb to embed more traditional floating point vworld coordinate systems within a much higher resolution substrate. In this way a visually seamless virtual universe of any conceivable size or scale can be created, and without worry about numerical accuracy.

**HoloWeb VWorld Coordinates**

Within a given vworld coordinate system, positions are expressed by three floating point numbers. Once again the default scale is centimeters, but this can be effected by scale changes in the modeling hierarchy.

*Coordinate System Conventions*

By default, the HoloWeb coordinate systems are right handed, with the orientation semantics that +Y is the local derivational up, +X is horizontal to the right, and +Z is directly toward the viewer.

Again, by default, HoloWeb coordinate systems are in units of centimeters.

# 4.2 Geometric Transforms

Transformations in HoloWeb are represented by a general orientation structure that contains a translation, represented by three double precision numbers; a rotation, represented by a (normalized) quaternion (four single precision numbers); and a (uniform) scale factor, represented by one double precision number. This combined structure is called a *gen_ori*, and is the type of the Transform field of all objects. Helping functions are defined within the scripting language to translate between numerous other transformation representations (4x4 matrices, etc.) (see Java interface routines chapter).

**Translation**

Three xyz values indicating the center of an object relative to the local vworld sub-coordinate system.

**Rotation**

A normalized quaternion [qx qy qz qw] representing the rotation by the angle of cos-1(qw) about the axis [qx qy qz]. The quaternion is assumed to be normalized, e.g. the sum of the squares of the components is unity. All built-in HoloWeb quaternion manipulation functions always return normalized quaternions.

**Scale**

A uniform scale of the object as it should appear in the local vworld (relative to its size in its own coordinates). Non uniform scales are explicitly disallowed for numerous performance reasons.

**Transform, gen_ori**

These last three types combined form a transform, sometimes refered to as a gen_ori structure. The order of application is the scale first, then the rotation, then the transformation.

## 4.3 Time

Representations of Time can also suffer from numerical accuracy problems. If one was to be a purest about it, then time values would also need 256-bit representations, in order to accurately represent a brief moment in the life of an electron anywhen from the big-bang to today. With appropriate time transforms, the now 1024-bit Universe Coordinates could serve as local anchors in space/time, letting mere floating point numbers represent local time.

However, this would be going too far (at least at present). Rather, representations of vworld time can be confined to double precision (64-bit) floating point numbers by two assumptions:

> 1) The time value of 0 is *not* the big bang; it is the local start time of the browser for stand-alone operation; if a standard time 0 is used, it is typically 00:00:00 UTC, Jan 1, 1970 (this is used for many networked applications).
>
> 2) Without time transforms, the smallest units of time of interest is usually not much than one millisecond. A 60Hz frame rate is 15 milliseconds, trackers and force feedback devices can require an order of magnitude better accuracy than this.

Combining these two, we see that from 1970 to 1996 requires 40 bits to represent all the milliseconds that have passed, beyond exact representability in single precision floating point. Indeed the 24-bit accuracy of single precision floating point runs out of milliseconds in four and a half hours (*some* 3D systems have been known to stay up longer than that).

Double precision, by contrast, has a few more bits, and won't run out of milliseconds until roughly the year 214,962 AD. Thus all vworld time values throughout HoloWeb are expressed using double precision floating point values. (Internally, browsers can use appropriate sized integers.)

---

# Chapter 5 - HoloWeb View Model

## 5.1 Viewing: separating application view control from display configuration

### 5.1.1 Introduction

As an application camera interface, the 4x4 viewing matrices of traditional 3D graphics packages are completely inadequate for the modern range of display devices. For example, if a display is headtracked (either for an external CRT or for a head mounted display (HMD)) the "camera" parameters *must* be dynamically computed, at least partially independent of whatever computation the application is doing.

Consider the traditional computer graphics view model parameter "field of view". With modern display devices, this is no longer a variable that can always be under the direct control of the application. For example, when HMDs are used, the field of view is effectively fixed by the HMD hardware and optics. If an application generates images using a markedly different field of view, the images are nearly un-viewable. Another important display configuration is that of head-tracked desktop or projection CRTs. Here the field of view varies as the viewer moves their head closer or further from the display screen. Once again if an application tries to force a specific field of view that is substantially different than the true dynamic value, then the display is nearly un-viewable. Indeed, nearly the only display configuration where the field of view can be forced to arbitrary values is the case of non head-tracked, non-stereo, external CRT displays ("flat-screen" mode).

What is needed is to separate the application’s need to control the virtual viewing platform from the end-user’s display configuration and calibration.

For these reasons, HoloWeb does not expose viewing transformations directly to the application. Rather, the application may specify a camera platform position, orientation, scale, and origin-mode.

Internally, browsers will support this by specifying the transformation between the viewer and the virtual world in two parts; a set of display configuration and calibration coordinate systems and transforms; and a virtual viewing platform coordinate system and transform.

While the majority of applications in the near-term may be satisfied with a viewing model that only supports flat-screen mode, HoloWeb’s support of a more general model allows the use of more sophisticated display configurations with no changes. It also ensures that HoloWeb applications initially targeted primarily at flat-screen mode will work properly in these other display environments as well, *with no changes to the applications*.

HoloWebs’ viewing model was designed in such a way that Browsers that only support simpler viewing configurations do not pay a performance penalty. This is another advantage of making the display configuration management the responsibility of the browser, not the application.

### 5.1.2 Viewer scale in the virtual world

One of the biggest differences between the traditional simple single camera model and the Virtual Reality head-tracked stereo camera model is the attention that needs to be paid to issues of scale. With a single camera, the scale of the viewer is not even semantically part of the camera model. But with head-tracked stereo, viewer scale is a fundamental parameter. One obvious way it comes into play is in specifying the distance between the left and right eyes (interpupillary distance). Here it is clear that the relative scale of the viewer with respect to the virtual world must be known to complete the view model. But just as importantly, relative head-movements of the viewer in the physical world must be scaled accurately and echoed in the virtual world for the visual perception to be correct. Once again, this can only be achieved if the relative scale between the virtual and physical world is known.

While most view policy decision can be made by the browser, sometime the applet may need to influence this policy. This shows up in HoloWeb’s interface for *clip* and *fog* objects to control distances of interest. Depending on the intent, it may make sense to specify these distances either in virtual world distances, or in physical world distances, this is supported by appropriate flags. For example, the front clipping plane is many times best specified as a small distance from the users head in the physical world, while the geometry may have been designed with specific assumptions about the maximum distance to the rear clipping plane in virtual world distance, and thus the rear clipping plane distance must be specified as a constant in virtual world coordinates. This second assignment is a function of the geometry content, and has to be specifiable by the applet independent from the browser.

### 5.1.3 Locating the viewer

There are other implications of the change to tracked viewers. Assume that a particular application wishes to display a virtual world in which the virtual floor appears at the same position as the real physical floor that the viewers are walking on. Consider two viewers, one five feet tall, the other six feet tall, both using HMD's. Clearly there is a calibration requirement that the system must know the real position of the physical floor relative to whatever head-tracking positional information is coming in. Furthermore, there is an application requirement that it wishes to fix portions of the virtual world (e.g., the virtual floor) at fixed relationships relative to the physical world, independent of where the viewer's head has gotten to at the moment.

But now consider a slightly different application, one in which a virtual airplane is being simulated, and that the airplane has a rather small glass canopy. Here the application does not care if the viewer is three feet tall or seven feet tall, or indeed if the viewer is standing or sitting (so long as they remain in the same posture). What the application wants is to make sure that the viewers virtual head nominal position is in the center of the canopy. Now the calibration constraint is for the system to know the nominal view head position (is the display configuration set up for sitting or standing? How tall is the present viewer?). The application requirement this time is to fix portions of the virtual world (e.g., the dynamic position/ orientation of the virtual airplane canopy) at a fixed relationship to the viewers nominal (calibrated, per viewer) head position, and does not really care where the physical floor is.

Consider finally a third application that wishes to display within the 3D equivalent of a window: a fixed volume in space. Just as 2D windows stay at fixed positions on the surface of the screen (until moved by the user), a 3D volume window would have a fixed position (and size) in space, stabilized to the physical world. That is, if currently your 3D stock quotes are appearing in a virtual cube three feet above the floor and slightly to the right of the viewer's (nominal) physical chair location, then that's where they should continue to appear, regardless of viewer head motion (until you specifically move the cube in space). The calibration requirement constraint is that the system must initially define this fixed volume relative to *something* in the physical world. The application requirement is that it wished to map a portion of the virtual world into this cube, independent from anything the viewer's head is up to.

These three examples illustrate the three ways that HoloWeb supports applications choosing how to specify constraints on the mapping of virtual world to the physical world. In all three cases, an explicitly specified point in the virtual world is constrained to match an un-ambiguously specified point in the physical world. The difference is in the choice of the tie point in the physical world.

**avatar feet**

The first example, of constraining a known point on the floor of the physical world to a known point in the virtual world is called *avatar_feet*. Here the un- ambiguous point in the physical world is the point on the floor directly below the point halfway between the physical viewers eyes, when the viewer is standing/ siting /lying at the nominal position in the physical world. This is called avatar feet because the point is at the feet of the viewer and the virtual avatar.

**avatar head**

The second example, of constraining a known point in the head of the physical viewer to a known point in the virtual world is called *avatar_head*. Here the unambiguous point in the physical world is the point halfway between the physical viewers eyes, when the viewer is standing/ siting /lying at the nominal position in the physical world. This is called avatar head because the point is inside the head of the viewer and the virtual avatar.

**avatar god's eye**

The third example, of constraining a known point in the physical world (near the viewer) to a known point in the virtual world is called *avatar_gods_eye*. Here the unambiguous point in the physical world nominal point in the physical world at a know offset from the nominal head or feet position of the viewer. This is called avatar gods eye because this mode is used to produce the traditional god's eye volume display.

Thus when an application wants to specify a new viewpoint to be applied to the viewer, the data given is a general orientation plus an integer specifying one of the three avatar modes described above. The AvatarSoul and the TeleportTarget nodes are just this collection of data.

---

# Chapter 6 - Base Object Components

## 6.1 Base Object Components

All HoloWeb objects are collections of data. The individual pieces of this data are usually referred to as components. The HoloWeb file format consists of a sequence of objects, and each object consists of a sequence of components. For the HoloWeb binary file format, the full specification of the composition of the components is required to be at the bit level. However, the organizational composition of different object types can be given at a higher level, by only referring to the types of components that make up a particular object type. The same composition applies both to HoloWeb's binary file format as well as HoloWeb's text file format.

Thus for clarity, HoloWeb objects will first be described in a more abstract manner, without reference to either text syntax issues, or the bit-level details. The formal, text and bit accurate specifications (will) appear in appendices.

These same components are also used in the run-time object representation, when a run-time object is accessible by one or more Java processes.

### 6.1.1 Generic Object Components

There are several object component types shared by different types of nodes. The semantics of these types are listed here for conciseness. The type names will be referred to in the individual object descriptions to follow.

**Flags**

Annotations associated with an object. Most annotations are hints to browsers on what optimizations can or might be made. Browsers are not required to make use of annotations (other than some like elemental animation operator child node present). From the authoring tool side, some annotations are merely hints (and thus can be guesses), but some others (like elemental animation operator child node present, and CONSTANT_TRANSFORM) must be true. Annotations load-time constants intended for the browser; they are not accessible at run-time by applications.

**Transform**

The transform component of this node, represented by a gen_ori object. The positional component is represented by three double precision floating point values for translation (xyz). The orientational component of this node's transform is represented as a normalized quaternion (four single precision floating point values).The scale component of this node's transform is represented by one double precision floating point value (the scale is therefore always uniform). Manipulatable by Java.

**Name**

An optional Unicode name associated with this object.

The next few sections describe some generic annotations, and then specifics of fields for group nodes and geometry nodes are given. The geometry node fields are previewed here so as to give an overall flavor for the general node details before digging into the details of rendering.

In later chapters, these three fields will sometimes be referred to as the BASE_OBJECT, when not showing all the detail again.

# 6.2 Annotations for Transform

Below are three annotations that apply to the transform field of all objects.

**CONSTANT_TRANSFORM**

Node includes a transform that will never change during run-time. Truly never should appear in optimized geometry.

**STATIC_TRANSFORM**

Node includes a transform that might be updated during run-time. (But not likely every frame. Need more specific stats?)

**DYNAMIC_TRANSFORM**

Node includes a transform that will be frequently updated during run-time.

## 6.3 Annotations for READ/WRITE DISABLING

Two annotations indicate the run-time mutability of an object (and, recursively, its children below it for group nodes). Many objects have additional object specific annotations to READ/WRITE disable certain individual fields of the object. The annotations below, by contrast, apply to *all* fields of the object.

**WRITE_DISABLED**

Indicates that this node, and the contents below it, can be restricted to read- only access by applets at run time. Thus browsers can know that none of the contents will be modified at run-time.

**READ_DISABLED**

Indicates that this node, and the contents below it, will be completely non- accessible to the applet at run-time, and the browser is free to optimize with this in mind. Specifically, not only does the initial data base not contain any pointers to this node or any below it (other than its immediate parent group node), run-time link creation (through capture boxes, constraints, pick events, etc.) cannot have any direct access to any of these nodes; the best they can do is get a pointer to the nearest ancestor that is not READ_DISABLED.

## 6.4 Annotations for Group Types

Below is a list of group node annotations. While some are mutuality inconsistent, others can appear together at the same Group node or HRoot node.

**UNION_CONST**

Indicates that the children are logically one bunch of geometry. They were not pre-combined due to some low level attribute incompatibility, such as different material attributes.

**CULL_SEP**

Indicates that the children should be view cull tested separately. Absence of this value indicates that the children should be cull tested together. As with many of these flags, this value is an advisory from the authoring system to the browser; browsers are not required to honor such requests; many will perform their own analysis of the culling strategy appropriate for a set of nodes.

**DB_ORG**

Grouping node only for upper level of data base. Almost always should also have CULL_SEP indicated.

**ELE_ANI_PRESENT**

Indicates that an elemental animation operator of some type is the first element of this group This annotation is set if a LOD node is present, as well.

**ELE_ANI_XFORM**

Elemental animation operator that will affect the group transform is the first element of this group.

**ELE_ANI_ATTR**

Elemental animation operator that will not affect the group transform is the first element of this group. But will effect color or opacity attribute.

**INSTANCE**

Instance object group node.

## 6.4.1 HRoot Object Components

HoloWeb HRoot objects have the following components:

Table 6-1 HROOT

| Object Component | Type of Component |
|---|---|
| Flags | HRoot specific annotation flags, integer valued |
| UniverseCoordinate | Absolute position represented by three 256-bit numbers |
| (Name) | (n 16-bit Unicode characters) |
| Children | The child nodes of this group object |

The UniverseCoordinates are absolute relative to the current file. They become translations when linked in by parent files.

HRoot nodes are the only node type that can (in fact, must) have no parent. group (and prototype) nodes must have a parent node; either another group node, or an HRoot node. In many cases, the only HRoot node needed would be a 000 HRoot relative to the local file. While the semantics could have been that any root group nodes are implicitly children of an implicit 0 0 0 HRoot node, a fully specified hierarchy is more explicit.

## 6.4.2 Group Object Components

HoloWeb group objects have the following components:

Table 6-2 GROUP

| Object Component | Type of Component |
|---|---|
| Flags | Group specific annotation flags, integer valued |
| Transform | General orientation value: position+rotation+scale |
| (Name) | (n 16-bit Unicode characters) |
| Children | The child nodes of this group object |

## 6.4.3 Geometry Object Basic Components

All HoloWeb geometry objects share several basic components. These are:

Table 6-3 GEOM_BASE_OBJECT

| Object Component | Type of Component |
|---|---|
| Flags | Geometry specific annotation flags, integer valued |
| Transform | General orientation value: position+rotation+scale |
| (Name) | (n 16-bit Unicode characters) |
| Color | RGB single precision floating point color |
| Opacity | One single precision floating point value |
| MaterialIndex | An 8-bit material index |
| TextureIndex | An 8-bit object texture index |
| Geometry | The specific geometry, in a variety of formats |

**Flags**

Geometry can contain a transform, the Flags field indicates the status of this transform (constant, static, dynamic). To have geometry without an associated transform, set the annotation field to CONSTANT_TRANSFORM, and the Transform to identity. Optimizing browsers will recognize this and ignore the transform.

In additional to the general WRITE_DISBLED and READ_DISABLED annotations, geometry supports two additional read/write disabled annotations specifically for the geometry field: GEOM_WRITE_DISABLED and GEOM_READ_DISABLED. The Geometry field is manipulatable by Java only of these flags are not set.

**Color**

The base color of the geometry defined by this node. Manipulatable by Java.

**Opacity**

The base opacity of the geometry defined by this node. Manipulatable by Java.

**MaterialIndex**

An indirect pointer to the material type for the geometry defined by this node. Manipulatable by Java.

**TextureIndex**

An indirect pointer to the texture for the geometry defined by this node. Manipulatable by Java.

**Geometry**

The geometry data for this node, in a variety of formats. Manipulation of the data by Java may be restricted by annotations; some formats may be quite a bit more time expensive to manipulate than others. The GEOM_WRITE_DISABLED and GEOM_READ_DISABLED annotations can block run-time manipulation to this individual field.

Further details of geometry semantics will be covered later in the Geometry chapter. The transport format of geometry is described in the chapter on geometry compression and in the binary and text file format appendices. Geometry rendering semantics also has its own chapter, as does (will) run-time manipulation and creation of geometry. Java messages to geometry are covered in the Java interface chapter.

---

# Chapter 7 - Elemental Animation Operators

## 7.1 Elemental Animation Operators

The use of Java as an animation language eliminates the need for complex pre-programed object behavior. Any object behavior beyond the very simplest sort should be handled by a user written (or behavior library) Java program. However, even Java functions come with a certain amount of overhead. Furthermore, requiring users to always write behavior descriptions at the frame by frame level can be both cumbersome *and* inefficient. There is also the serious issue of how actions computed local at one processor can be shared by many more computers in a network in a time-effective fashion. Therefore a simple set of low level pre-programmed behaviors is included as a building basis for Java code. These are called *elemental animation operators*. These simple functions in many cases can handle the frame by frame details between "key frames"; Java code dictates what happens at the next level up. This is very similar to the one simplistic way of looking at how the human motor control functions are organized: the reptilian brain (elemental animation operators), the mammalian brain (Java), and the simian brain (user).

Five fundamental (virtual) physical properties are shared by nearly all objects: position, orientation, scale, color, and transparency. The elemental animation operators are just linear segment interpolations of a sequence of one of these values (and one combination value).

Details of how the interpolations are controlled will be covered shortly.

This in effect is a first order non-uniform spline of value control points for a property.

Of course the most fundamental property of (most) objects is their geometry. While usually it is too computational complex to interpolate in real-time between instances of geometry, a similar effect can be gained by switching in real-time between multiple different geometries. This is the elemental animation operator *tloop*. This operator can also be controlled to achieve the effect of what other systems call switch objects. (When general interpolation of geometry at run time is required, HoloWeb supports this through run-time access/update of geometry.)

A related elemental animation operator is *movie*, which switches in real-time between n individual 2D images (but displayed as a 2D plane embedded in 3D space). The *sound* operator does the same for audio data.

*tloop*: cycles among n individual objects

*movie*: cycles among n individual 2D images (embedded in 3D)

*audio*: cycles among sequential audio data (embedded in 3D)

### 7.1.1 Details of Interpolation

The control values for several of the elemental animation operators are specified by a sequence of alpha control value pairs:

$$(0.0, V_0),\ (\alpha_1, V_1),\ \ldots,\ (\alpha_{v-2}, V_{n-2}),\ (1.0, V_{n-1})$$

In the sequence, the first alpha value must be 0.0, the last must be 1.0, and all values in-between must be in the range of zero to one and monotonic increase. The control values are the type of the specific elemental animation operator. For positional animation, the control values would be xyz points, for orientation animation, the control values would be quaternions, etc.

This sequence defines a mapping from any value of alpha in the range zero to one to a value of the control value. The control value is defined to be the linear interpolation of the two control values given at the two sequence pairs whose alpha values bracket the input alpha value. Specifically if the input alpha lies between the values of ai, and ai+1, the control value is:

$$v' = v_i \cdot \left(1 - \frac{\alpha - \alpha_i}{\alpha_{i+1} - \alpha_i}\right) + v_{i+1} \cdot \frac{\alpha - \alpha_i}{\alpha_{i+1} - \alpha_i}$$

This is in effect a first order non-uniform spline.

Orientation in paths is specified by a quaternion at each control point. Interpolation of the orientation between two control points can be the linear interpolation of their two quaternions. For most cases of interest the values of the orientation are assumed to be close enough to each other to make the variance in different forms of orientation interpolation visually unimportant; thus the details of orientation interpolation may be left up to the browser implementation.

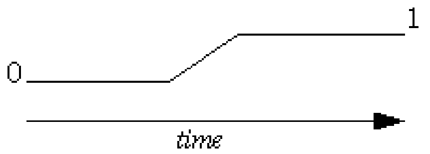
Elemental animation operators have parameters that map time values to alpha, as will be described in the next sections. Combining this with the alpha to value mapping, this is how real time is mapped to values.

The semantics of the alpha traversal depends upon how the control point alpha values were computed in the authoring system. One possibility would be for alpha to vary directly with the euclidean distance along the path (for positional animations). Another might corresponds to alpha reaching each control point at pre-computed times (rather like a train schedule).

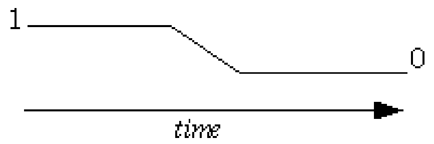
### 7.1.2 Alpha Sequencing Waveform

The mapping of time to alpha is controlled by several elemental animation operator attributes.When these parameters (and the alpha/value spline) are not being changed, the mapping from time to alpha is a *constant function of time*, and is well defined for all values of time. This means that elemental animation operators that do not otherwise communicate can none the less precisely coordinate their behaviors. Much of the Java behavior programming in HoloWeb involves Java threads responding to events and re-programming the behavior of the lower level elemental animation operators.

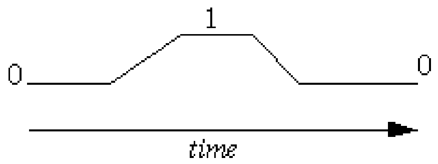
There are three one-shot and three cyclic behaviors for alpha:



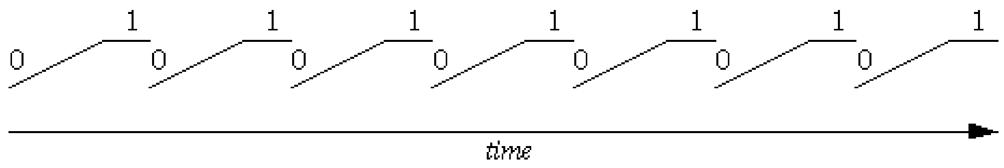

One-shot forward: At a specified point in time, alpha goes from 0 to 1 over a specified interval of time. At all times before the starting time, alpha is 0; at all times after the finish time, alpha is 1 (unless the elemental animation operator is reprogrammed (re-armed)).



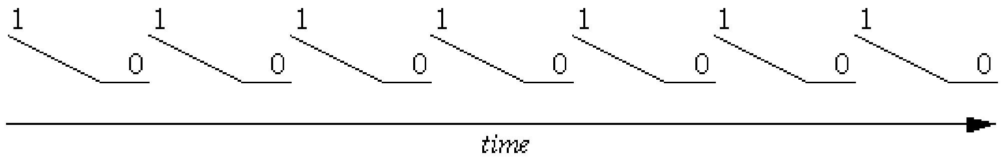

One-shot backward: At a specified point in time, alpha goes from 1 to 0 over a specified interval of time. At all times before the starting time, alpha is 1; at all times after the finish time, alpha is 0 (unless the elemental animation operator is reprogrammed (re-armed)).



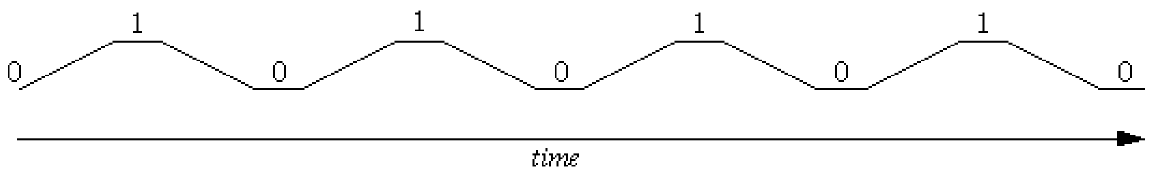

One-shot forward + backward: At a specified point in time, alpha goes from 0 to 1 over a specified interval of time. After a specified waiting interval, alpha then goes back from 1 to 0 over another specified interval of time. At all times before the starting time, alpha is 0; at all times after the finish time, alpha is also 0 (unless the elemental animation operator is reprogrammed (re-armed)).




Looped forward: Alpha goes from 0 to 1, waits a specified period of time, the alpha value then jumps back to 0 and then goes to 1 again. This action repeats for all time (unless the elemental animation operator is reprogrammed).




Looped backward: Alpha goes from 1 to 0, waits a specified period of time, the alpha value then jumps back to 1 and then goes to 0 again. This action repeats for all time (unless the elemental animation operator is reprogrammed).




Looped forward + backward: Alpha goes from 0 to 1, waits a specified period of time, then goes from 1 to 0, waits another period of time, then goes from 0 to 1 again. This action repeats for all time (unless the elemental animation operator is reprogrammed).

### 7.1.3 Specification of times

There are six time values that effect the overall timing of all elemental animation operators, though not all values effect all modes. A simplified definition of these are:

Start time: *time_zero* + *phase*

0 to 1 time: *forward_time*

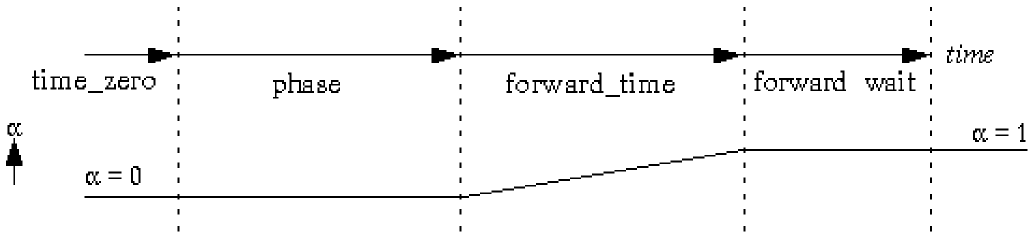
1 to 0 time: *backward_time*

period of waiting while alpha is at 1: *forward_wait*

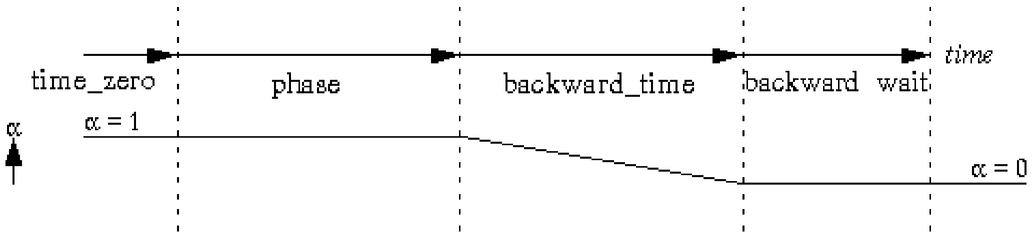
period of waiting while alpha is at 0: *backward_wait*

Different sequencing waveforms use different subsets of these values. The three diagrams below illustrate how each of the six types of sequencing waveforms use these timing values:

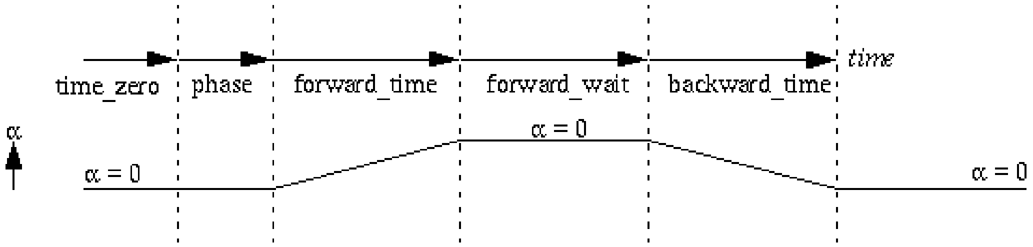
Sequencing waveform one-shot forward and looping forward:




Sequencing waveform one-shot backward and looping backward:




Sequencing waveform one-shot forward + backward and looping forward + backward:




## 7.1.4 Details of times

**time_zero and phase**

The starting time for elemental animation operations is specified by *two* values: *time_zero* and *phase*. time_zero is the nominal starting time (relative to virtual world time). phase is a specific offset from time_zero for individual variation. While both time_zero and phase are "local" instance variables of any elemental animation operator, typical use is for a Java function to simultaneously reset time_zero to the *same* value in several different ones ("synchronize watches"), but the operators will fire at their own individual relative timing (phase). If no such individual variation is desired, simply set phase to zero. Splitting the start time specification in this way obviates much of the need for local time generators.

**forward_time**

The virtual world time interval over which alpha varies from 0 to 1 is specified by the instance variable *forward_time* (in virtual seconds). If the alpha sequencing waveform is one-shot backward or looping backward, then forward time is *not* used (and alpha snaps instantaneously from 0 to 1 during looping backward).

**backward_time**

The virtual world time interval over which alpha varies from 1 to 0 is specified by the instance variable *backward_time* (in virtual seconds). If the alpha sequencing waveform is one-shot forward or looping forward, then backward time is *not* used (and alpha snaps instantaneously from 1 to 0 during looping forward).

**forward_wait**

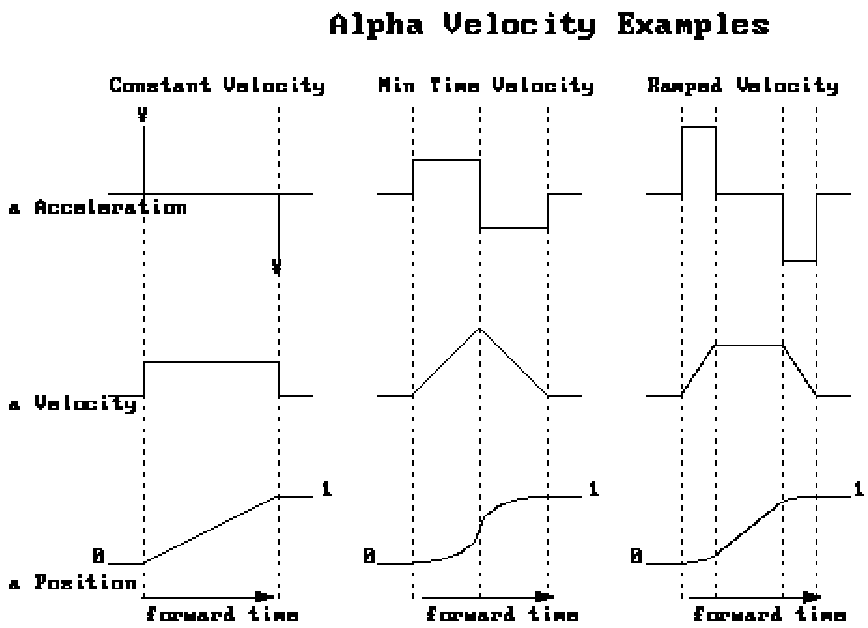
The virtual world time interval over which alpha stays at 1 for the alpha sequencing waveform one-shot forward + backward or looping forward or looping forward + backward is *forward_wait*. forward_wait is *not* used for the other three alpha sequencing waveforms.

**backward_wait**

The virtual world time interval over which alpha stays at 0 for the alpha sequencing waveform one-shot looping backward or looping forward + backward is *backward_wait*. backward_wait is *not* used for the other four alpha sequencing waveforms.

## 7.1.5 Acceleration of Alpha

A common technique used in animation is "slow-in, slow-out". This functionality is *not* built into HoloWeb, but can easily be achieved by an authoring package specifying the appropriate alpha value curve. Three typical cases are illustrated in the following diagram:




**Constant Velocity**

The constant velocity curve gets alpha from 0 to 1 at a constant velocity of alpha, and thus also get the elemental animation operator value controlled by alpha to traverse its range at a constant velocity. This case corresponds to an infinite initial and breaking acceleration of alpha. Thus is typically used for continuous motion: a constant rotation, looped fly-throughs of an object.

**Min Time**

The "Min Time" curve corresponds to a constant acceleration for half the total traversal time (and thus a ramped velocity), and then a constant deceleration for the other half of the time. For a physical system, this sort of acceleration behavior corresponds to a minimum time of traversal. This might be used for controlling a butterfly's wing flapping.

**Ramped Velocity**

The "Ramped Velocity" curve has initial period of acceleration, then coasts for a while, then a symmetrical period of de-acceleration. This might be used for a door opening.

Again, these curves are not build into HoloWeb, but can be created by specifying a few dozen alpha control point values in the controlling curve.

# 7.2 Elemental Object Components

All elemental animation operator objects share the parameters described in the previous section. Grouped together with the generic object fields, these are:

Table 7-1 ELEMENTAL_ANIMATION_OBJECT

| Object Component | Type of Component |
|---|---|
| Flags | Annotation flags |
| Transform | General orientation value: position+rotation+scale |
| (Name) | (n 16-bit Unicode characters) |
| Number of control points | An integer value |
| Time Zero | One double precision floating point value |
| Phase | One double precision floating point value |
| Forward Time | One double precision floating point value |
| Backward Time | One double precision floating point value |
| Forward Wait | One double precision floating point value |
| Backward Wait | One double precision floating point value |
| Sequencing waveform | An enum, one of six values |
| Array of control values | control value type |

By adjusting these parameters, Java objects can control many aspects of behavior and animation in a time efficient, net-work friendly way.

Two additional annotation values are supported: SPLINE_WRITE_DISABLED, and SPLINE_READ_DISABLED. While it is common for Java code to modify the other fields, there are many cases in which the control value spline is never accessed or modified. These annotations allow browsers to take advantage of this fact.

# 7.3 Elemental Animation Object Spline Principles

All the spline elemental animation operator objects operate in a similar fashion. Each contains an array of sample points. As alpha varies from 0 to 1, an operator will update its group transform (or attribute) to the appropriate interpolated value along its sample points. When alpha is 0, the value comes from the first sample point. When alpha is 1, the value comes from the last sample point. As described above, values of alpha interpolate along the control points are based upon the value of alpha pre stored at each control point.

Elemental animation operators affecting transform attributes can be viewed as modifying the transform field of the parent group node of the elemental animation operator.

Elemental animation operators on color and opacity conceptually modify theses fields of all non-write disabled geometry descendents of the group node. The value of the color or opacity should not be thought of as "living" at the elemental animation operator as a rendering attribute; rather it should be thought of as being immediately transmitted to (very typically pre-identified) child geometry nodes.

TLoop and LOD elemental animation operators switch which group child nodes will be displayed at a given frame.

Movie elemental animation operators only have an internal rendering effect, and do not effect any other child nodes of their parent group.

Sound elemental animation operators only have an internal audio rendering effect, and do not effect any other child nodes of their parent group.

# 7.4 PPath Object Details

The ppath elemental animation operator object specifies a variable length array of alpha/position sample points. The component is:

Table 7-2 PPATH_OBJECT

| Object Component | Type of Component |
|---|---|
| Array of a,x,y,z points | Array[Four double precision floating point values] |

A ppath's control points define *relative* movement along their path. A ppath that is the first element of a group will cause the groups's contents to move in space offset to the path defined by the ppath's sample points as alpha varies from 0 to 1. The move is *relative*, in that a value of alpha of 0 causes no change in the groups contents position.

## 7.5 RPath Object Details

The rpath elemental animation operator object specifies a variable length array of alpha/rotation sample points. The component is:

Table 7-3 RPATH_OBJECT

| Object Component | Type of Component |
|---|---|
| Array of a,x,y,z,w points | Array[One double plus four single precision floating point values] |

A rpath that is the first element of a group will cause the groups's contents to change their orientation in space to the orientations defined by the rpath's sample points as alpha varies from 0 to 1. The orientations are also relative to the initial orientation of the other objects in the group.

A common rotation is a fixed rotation rate about a fixed axis. Browsers implementaions may discover this by analising the spline data, and if the spline is marked as non-writable, implement the rotation using a simpler rotation computation.

## 7.6 PRPath Object Details

The prpath elemental animation operator object specifies a variable length array of alpha/position/rotation sample points. The component is:

Table 7-4 PRPATH_OBJECT

| Object Component | Type of Component |
|---|---|
| Array of a, pos, quat points | Array[Four double and four single precision floating point values] |

A prpath that is the first element of a group will cause the groups's contents to change their (relative) position and orientation in space to the positions and orientations defined by the prpath's sample points as alpha varies from 0 to 1. The positions and orientations are relative to the initial positions and orientation of the other objects in the group.

## 7.7 Scaler Object Details

The scaler elemental animation operator object specifies a variable length array of alpha/scale sample points. The component is:

Table 7-5 SCALER_OBJECT

| Object Component | Type of Component |
|---|---|
| Array of a,scale points | Array[Two double precision floating point values] |

These are *relative* scales. The define scale multiples of the group object's current scale.The scale of the group object represents the effect of the scaler at a scale of 1.0. During operation, as alpha varies between 0 and 1, the scaler will scale the groups contents by relative scales that appear in the array between alpha of 0 and alpha of 1.

Note that the scale represented by the group object (scaler scale of 1.0) never has to occur during scaling, if the scaler scale of 1 is not a value in the array.

# 7.8 Colorer Object Components

The Colorer elemental animation operator object specifies a variable length array of alpha/color sample points. The component is:

Table 7-6 COLORER_OBJECT

| Object Component | Type of Component |
|---|---|
| Array of a,r,g,b points | Array[One double and three single precision floating point values] |

A colorer that is the first element of a group will cause the current_color for the groups other elements to vary along the spline of color values in the array.

# 7.9 Opacator Object Components

The Opacator elemental animation operator object specifies a variable length array of alpha/opacity sample points. The component is:

Table 7-7 OPACATOR_OBJECT

| Object Component | Type of Component |
|---|---|
| Array of a,opacity points | Array[One double and one single precision floating point values] |

A opacator that is the first element of a group will cause the current opacity for the groups other elements to vary along the spline of control opacity values in the array. An opacity of 0 (or very close to 0) effectively makes an object invisible; a browser implementation may notice that an opacator has been parked at a value of 0, and perform cull optimization on effected objects. This semantics has implications for picking and certain other behavior.

# 7.10 TLoop Object Semantics

The TLoop elemental animation operator object causes its parent group node to switch between its other n-1 children for display, as the alpha value of the TLoop object varies between 0 and 1. Specifically, for a given alpha value, the index (from 1 to number of children-1) is given by the equation:

index = 1 + floor(a*(number_of_children-1) + 0.49);

Notice that if the alpha interpolation is disabled, and the alpha value is directly set, the effect of a "switch" primitive is achieved.

# 7.11 Movie Object Semantics

The Movie elemental animation operator object specifies a variable length structure of sequential 2D pixel images in a variety of formats. The components are:

Table 7-8 MOVIE_OBJECT

| Object Component | Type of Component |
|---|---|
| Natural Frame Rate | Single precision floating point nominal inter frame time in seconds |
| 2D image sequence | Variable length structure of sequential 2D pixel images in a variety of formats |

The Movie elemental animation operator object causes the image sequence associated with this object to be displayed in space, as the alpha value of the Movie object varies between 0 and 1.

Most movie data has a natural frame rate. Movie objects retain this value, expressed as the time in seconds to play frames at their natural rate. If the forward time of a Movie object is set to the multiple of this time times the total number of frame (minus one) in the image sequence, natural playback is achieved. Some formats may not support playback at other than this natural rate.

Notice that if the alpha interpolation is disabled, and the alpha value is directly set, the effect of a seek to a frame operation is achieved. Note also that not all moving image formats may directly support frame specific seek.

Note also that while a movie object can be played backward by specifying one of the backward loop control values, not all moving image formats may directly support reverse playback.

# 7.12 Sound Object Semantics

The Sound elemental animation operator object specifies a variable length structure of sequential audio data in a variety of formats. The components are:

Table 7-9 SOUND_OBJECT

| Object Component | Type of Component |
|---|---|
| Natural Sample Rate | Single precision floating point nominal sample time. in seconds |
| Sound type | One of: Ambient, Cone, Point |
| Attentuation Factor | |
| Attentuation Distance | Single precision float attentuation distance |
| Volume | Relative sound volume, single precision float 0-1 |
| AudioData | Variable length structure of sequential audio data in a variety of formats |

The Sound elemental animation operator object causes the audio sequence associated with this object to be emitted in space (from the current vworld location of the sound object), as the alpha value of the Audio object varies between 0 and 1.

Most audio data has a natural play-back rate. Audio objects retain this value, expressed as the time in seconds to play samples at their natural rate. If the forward time of a Audio object is set to the multiple of this time times the total number of samples (minus one) in the audio sequence, natural playback is achieved. Some formats may not support playback at other than this natural rate.

Notice that if the alpha interpolation is disabled, and the alpha value is directly set, the effect of a seek to a time operation is achieved. Note also that not all audio formats may directly support frame specific seek.

Note also that while a audio object can be played backward by specifying one of the backward loop control values, not all audio formats may directly support reverse playback.

## 7.13 LOD Object Semantics

The LOD (level of detail) elemental animation operator object specifies a variable length array of distance thresholds. The components are:

Table 7-10 LOD_OBJECT

| Object Component | Type of Component |
|---|---|
| Array of distances | Array[One double precision floating point value] |

---

A LOD that is the first element of a group will cause its parent group node to switch between its other n-1 children for display, based on the current distance to the objects. This distance is compared with the monotonically decreasing values in the array of distances; the index of the first array distance value less than the object distance value is also used to select which child of the parent group to display. Index 0, which would have referred to the LOD node itself, instead indicates the threshold for no display at all of an object.

---

# Chapter 8 - Avatar Souls and Aspects

## 8.1 AvatarSouls: Handles on the viewer

An AvatarSoul object exists as a handle for controlling the viewer's view platform. The base_object Transform properties of an active AvatarSoul object directly control virtual half of the view transform.

The AvatarSoul object adds one additional components to the BASE_OBJECT:

Table 8-1 AVATAR_SOUL

| **Object Component** | **Type of Component** |
|---|---|
| avatar soul center | enum, one of avatar_feet, avatar_head, or avatar_gods_eye |

These three enums work as follows:

*avatar_soul_feet*: coordinates centered about the center of the AvatarSoul's feet.

*avatar_soul_head*: coordinates centered about the center of the AvatarSoul's eyes.

*avatar_soul_gods_eye*: coordinates centered about the center of the display window.

The position, orientation, and scale of the AvatarSoul object control those same values of the coexistence to vworld transform. The primary difference between the three AvatarSoul centers has to do with how the center of the transform is specified.

At any one time, a given user connected to a virtual world is connected ("jacked-in") to one specific AvatarSoul object within the virtual world. Any changes in the position, orientation, or scale of the connected AvatarSoul object directly relate to a similar change in the viewer's virtual viewing platform.

AvatarSouls are not user vehicle objects. Rather, they are a lower level primitive that allows an applet to create "plug-in" user vehicle. AvatarSouls are not TeleportTargets; rather they are the object that TeleportTargets effect.

The association of an active viewer with a particular AvatarSoul object is up to the browser. Some applets will not specify any AvatarSoul, and the browser should provide a default vehicle for the user to maneuver around the virtual world, or allow the user to employ a separate applet that *is* a virtual vehicle (plug-in Avatars). Other applets content will depend on exerting complete control on the AvatarSouls that the user inhabits while interacting with the applet (at least in the way the content authors intended). Here the applet will indicate to the browser which AvatarSouls and when it want the user connected to. (Details of this indication still need to be specified.)

If an applet wishes to completely control the path of an avatar through space, this can be achieved either by explicitly setting the AvatarSoul Transform field every frame from a Java thread, but for fixed paths this is more easily achieved by adding a dynamic constraint object between the AvatarSoul and a PRpath operator. At the end of the path, the applet would retract the constraint.

A more common case is when an applet wishes to only influence the position of an AvatarSoul, not completely specify it. Examples of this are elevators and conveyor belts. For an elevator, the applet only wishes to constrain the AvatarSoul.feet to be above where the elevator platform has risen/fallen too, unless the Avatar has moved off the elevator platform. Here the need is to form a dynamic constraint this only influences this aspect of Avatar position. One general solution would be for the applet to have a Java thread examine the AvatarSoul y-position every frame, modifying it as necessary. With complex constraints, this sort of per-frame Java code is probably the only general answer. However, for simpler situations variants of dynamic constraint objects on pre-specified components of objects's transformations may prove viable, so long as there is some uniformity between applets on applying these constraints (not always possible).

## 8.2 TeleportTarget: Places to Go

A TeleportTarget object exists as a handle for controlling where the viewer's view platform might be assigned to (discontinuately) go. The BASE_OBJECT Transform properties of aTeleportTarget object when applied to an active AvatarSoul directly control virtual half of the view transform.

The TelportTarget object adds one additional components to the BASE_OBJECT:

Table 8-2 AVATAR_OBJECT

| **Object Component** | **Type of Component** |
|---|---|
| TeleportTarget center | enum, one of avatar_feet, avatar_head, or avatar_gods_eye |

These three enums work the same as with the AvatarSoul object.

The semantics of teleporting a specified AvatarSoul object to a specified TelportTarget object is to for the contents of the TeleportTarget's Trasnform (position, rotation, scale), and center point type to relpace those fields within the AvatarSoul. However, this is a one-shot momentary constraint. If the AvatarSoul is under the control of some vehicle applet, the applet will take right over updating the AvatarSoul Transform, though starting from wherever it got to.

There are some complex issues involving repeated changes of scale. The approprate policy on how non-unity scale changes durring TeleportTarget assignment are to be handled can resolve these.

## 8.3 Aspects: Handles on the Environment

Several special pseudo objects exist to be used as local handles on local values of global rendering attributes. These handles are called aspects, in the sense that they are aspects of their global parent. Specifically these include:

*light*: handle on (one of 8) numbered light sources.

*sky*: handle on current background/fog color.

*background_ramp*: handle on current background color ramp.

*background_texture*: handle on current background texture.

*fog*: handle on fog control values.

*clip*: handle on clipping control values.

*fade*: handle additional orthogonal (constant fog level like) color fading. This is used to effect scene fade-cuts in a local environment. These have been found to be less jarring to the human visual system than fast cuts.

*light*: changing the color, position, orientation, or scale, of a light handle has the same effect on the numbered light source associated with that handle, if the handle is active. (Of course, for some light source types, some of these changes do not effect the lighting; the location of directional light sources does not effect the lighting.)

*sky*: changing the color of the sky handle changes the color of the background (and fog) if the handle is active. Changes to position, orientation, or scale have no effect, except on the handle object itself.

*fog*: The eight internal values of a fog aspect control the current depth cuing configuration. None of these values are directly tied to any base properties of a fog object.

*clip*: The six internal values of a clip aspect control the current clipping plane configuration. None of these values are directly tied to any base properties of a clip object.

Environment aspects control environment attributes only if the aspect is active. The semantics of activation and de-activation of aspects is controlled by environment proximity spheres, which will be described later.

## 8.4 Light Aspects: Handles on the Lights

There are eight light sources, numbered 0 through 7.

All light sources have the property of color.

All light sources have the property of being on (enabled), or off.

Light sources can be individually set to be of type ambient, directional, local, or spot.

Directional light sources have a property of direction.

Local and spot light sources have the properties of position and attenuation.

Spot light sources have properties of direction and cone angle, and an additional attenuation factor.

As was discussed, the properties of light sources are controlled by handles; virtual objects whose properties are linked to those of the light sources.

Table 8-3 LIGHT_OBJECT

| Object Component | Type of Component |
|---|---|
| Light Type | An enum of ambient, directional, local, spot |
| Angle | A single double precision floating point value |
| Attenuation 1 | A single double precision floating point value |
| Attenuation 2 | A single double precision floating point value |
| Exponent | A single double precision floating point value |

# 8.5 Sky Aspects: Handles on the background color

The sky object has one value of interest: its color. When a sky object is active, changes to its color will change the rendering background color (which is also the fog color)

The sky object adds no additional components to the BASE_OBJECT:

Table 8-4 SKY_OBJECT

| Object Component | Type of Component |
|---|---|

A particular sky object can have a constant color, which will only be in force when it is active. Another reasonable case would be to have a one-shot or continuous blinker controlling the sky object's color, in which case the sky color will in effect be controlled by the blinker. Finally, an application can directly set the sky color at run time through a Java call to change the color of the sky object. This is the *only* way that Java code can set the sky color; there is no direct global variable controlling the color of the background.

BACKGROUND_RAMP

BACKGROUND_TEXTURE

# 8.6 Fog Aspects: Handles on depth cuing

The fog object holds local environment control values for depth cuing and fog effects.

The fog object adds eight components to the BASE_OBJECT:

Table 8-5 FOG_OBJECT

| Object Component | Type of Component |
|---|---|
| fog front distance policy | enum, one of from_eye, from_screen |
| fog front world | enum, one of virtual_world, physical_world |
| fog front distance | One double precision floating point value |
| fog front scale | One single precision floating point value |
| fog rear distance policy | enum, one of virtual_world, physical_world |
| fog rear world | enum, one of virtual_world, physical_world |
| fog rear scale | One single precision floating point value |
| fog rear distance | One double precision floating point value |

The fog parameters loosely follow the PHIGS depth cueing model, where there are two inflection points in the fog distance/amount curve, here called front and rear. The two scale amounts are in the range of 0 to 1.

The distance policy indicates if a fog reference plane is to be specified relative to the viewer's dynamic eyepoint location, or specified relative to the fixed god's eye screen in space.

The world choice indicates wether the distance value will be specified in the physical world or the virtual world. (See the discussion in the viewing model chapter on why this world choice is important.)

The scale amount is the "level of fog", where 1.0 is no fog, and 0.0 is completely fogged out to the background color.

The distance value is the distance to the indicated fog reference plane in the world specified.

These three values are independently specified for both the front and rear fog reference planes.

A fog object can be constant, or have its values updated by application Java code. Note that like some other environment handles (e.g. light), none of the base object properties is tied directly to any of the fog effect values. Explicitly the fog color is not used, as the color of the fog is tied to the background color, which is controlled by another object.

Putting this another way, the fog is rarely animated. The fog object typically is used to represent a fixed depth cuing set-up, or as a handle for Java code to do whatever it wants. Note that the fog object is the *only* way that Java code can directly effect fog parameters.

# 8.7 Clip Aspects: Handles on the view clipping policies/clipping planes

The clip object holds local environment control values for the front and rear clipping planes.

The clip object adds six components to the BASE_OBJECT:

Table 8-6 CLIP_OBJECT

| Object Component | Type of Component |
|---|---|
| clip front distance policy | enum, one of from_eye, from_screen |
| clip front world | enum, one of virtual_world, physical_world |
| clip front distance | One double precision floating point value |
| clip rear distance policy | enum, one of virtual_world, physical_world |
| clip rear world | enum, one of virtual_world, physical_world |
| clip rear distnace | One double precision floating point value |

The distance policy indicates if a clipping plane is to be specified relative to the viewer's dynamic eyepoint location, or specified relative to the fixed god's eye screen in space.

The world choice indicates wether the distance value will be specified in the physical world or the virtual world. (See the discussion in the viewing model chapter on why this world choice is important.)

The distance value is the distance to the indicated clipping plane in the world specified.

These three values are independently specified for both the front and rear clipping planes.

A clip object can be constant, or have its values updated by application Java code. Note that like some other environment handles (e.g. light), none of the base object properties is tied directly to any of the clip effect values.

Putting this another way, the clip is rarely animated. The clip object typically is used to represent a fixed clipping plane set-up, or as a handle for Java code to do whatever it wants. Note that the clip object is the *only* way that Java code can directly effect clipping parameters.

# 8.8 Environment Proximity Sphere

An environment proximity sphere is a special node object to support the layered overriding control of environment aspects. When an environment proximity sphere object is the first object of a group, and the viewer crosses into the sphere defined by the radius of the environment object, any environment aspects within the rest of the group now becomes enabled and active, and override any environment aspects defined by environment proximity sphere objects outside of this one. Of course, any closer in environment proximity sphere objects may override this one's aspects. In other words, individual environment properties are spatially scoped by the nearest enclosing environment proximity sphere containing the viewer that defines a particular environmental attribute. If an attribute is not defined by any enclosing spheres, then that attribute defaults to a browser specific default (logically attached to the environment proximity sphere at infinity), who's value *is not* accessible to the applet. If an applet wants to control environmental attributes, it must put them in a finite environment proximity sphere.

# 8.9 Link Objects

Just-in-time loading of link objects (URL's). Programmable load/unload behavior: Load immediately, load on avatar proximity, load on Java message. Unload on proximity, unload on resource need-back by browser.

## 8.10 Proximity Spheres: triggering actions in 3D

Another central HoloWeb concept is that of "proximity spheres". A proximity sphere is a triggering device for Java animation code.

Proximity spheres are triggered by an active_avatar transiting their boundary (or, in some cases, the avatar hand).

A Java routine can sleep on a proximity sphere being crossed *into* or *out of*. Java can also test for *still inside*.

In addition to fast testing spheres, slower but more general convex surface proximity tests should be supported.

---

# Chapter 9 - More HoloWeb Objects

## 9.1 Additional HoloWeb Objects

In addition to the HoloWeb objects described in the previous chapters, and number of additional objects exist. This chapter describes some of those that are less well documented or defined. Most, but not all, of these have semantics similar to similar objects in other systems.

> Texture Objects
>
> Background Textures
>
> Collision Object: a low polygon count stunt double for collision detection for a more complex rendering polygon count object.
>
> Material Object. The usual suspect.
>
> I/O event objects and their routines. Proximity spheres and the various Java event blocking calls handle much of this, but more routines and specification is needed.
>
> Prototypes: A DAG of nodes with a Prototype node root defines a HoloWeb Prototype object. Within this DAG of nodes, individual fields of nodes can be marked as parameters. Most interesting discussion of Prototypes revolves around issues of syntax, not semantics.
>
> Instances: A reference to a Prototype object along with matching parameter instance information. All parameters must be fully instantiated; a new Prototype cannot be formed by passing a

Prototype as an instance variable to a Prototype object. Most interesting discussion of Instances revolves around issues of syntax, not semantics.

---

# Chapter 10 - Java Routine Interface

## 10.1 Java interface routines for controlling HoloWeb objects

At the object level, Java routines are just another HoloWeb object, complete with a 3D location. Java routines can have connections to other HoloWeb objects; this is how arguments are set up for Java routines.

The Java object itself is the Java code and a variable number of arguments:

Table 10-1 JAVA_OBJECT

| Object Component | Type of Component |
|---|---|
| Flags | annotation flags |
| Transform | General orientation value: position+rotation+scale |
| (Name) | (n 16-bit Unicode characters) |
| Java routine | Reference to Java routine |
| First routine argument | Reference to HoloWeb Node |
| Second routine argument | Reference to HoloWeb Node |
| . | . |
| . | . |
| . | . |
| Last routine argument | Reference to HoloWeb Node |

In the input file, in binary format the Java code is in Java byte code format; in text format, the Java routine is in text format.

## 10.2 HoloWeb Java Thread Model

Conceptually, all java objects in node hierarchy are independent concurrent threads. In actually, browsers are allowed to *execution cull* any threads who's geometric bounding sphere is outside the *locale* of the viewer currently. (The locale is a bounding sphere around the viewer at least as large in radius as the rear clipping plane is away from the eye point.) Note that the culling behavior is based on the location of the Java object, *not* the set of all its argument objects. This restricts the reach of control a java object can exsert over the virtual world; a java object on the other side of the galaxy cannot move your dog around. It is the responsibility of applications to specify the range of influence of any given java object. (Remember that a moving collection of nodes can carry it java control objects with it, moving their sphere of influence.)

Typically, most Java threads execute a small amount of code, and then block pending some specified event. As a result, most Java threads are in a blocked state nearly all of the time, and usual go right back to a blocked state after waking up for only one frame. Paraphrasing, a Java thread's life is an eternity of boredom, punctuated by moments of sheer terror.

While Java threads are blocked, plenty of animation is going on. The elemental animation operators loyally follow their most recent marching orders of their master Java threads, continuing spinning, blinking, moving their geometric charges. At some point in the future, some event, such as sleep timers, user interaction, collisions, etc., will re-awaken some controlling Java thread, whose execution will like cause new marching orders for the hapless foot soldiers of HoloWeb, the lowly elemental animation operators.

In some cases, partial ordering of the execution threads is desirable, and mechanisms to specify these should exist.

To support load management, a browser does not need to guarantee that every potentially executable Java thread will be executed every frame.

# 10.3 HoloWeb JavaThread Distributed Execution Model

Because of the arms length control of elemental animation by Java threads, distributed multi-user shared virtual environments are implicitly supported. A Java control thread need only execute on a single machine; when it sends a message to its slave elemental animation object, the message can be transparently cloned and sent to the replicated elemental animation objects on all the other interested machines. And because the elemental animation operators are closed functions of time, effectively identical behavior will be exhibited by all the participating machines (once the parameter change messages arrive).

This solution does not scale to cover arbitrary numbers of connected machines viewing the same state, nor does it solve the network latency scheme any more than any other predictor/interpolator based scheme. However, it does begin to address the issue of a general scripting mechanism supporting multi-user environments partially transparently for small numbers of low to medium latency connected machines.

## 10.3.1 Java structures for interface routines

The Java classes listed below describe the data objects required by the HoloWeb Java interface. The "HSJ" prefix stands for "HoloSketch Java", the first implementation of HoloWeb. The prefix probably should be changed, but for the moment the HoloWeb prefix of "HW" may be confused with hardware prefixes, and so is left alone for the moment.

```
/* Double precision 3D point object */

class HSJ_pt_d3d {
```

```
        double x, y, z;
}
/* Single precision 3D point object */

class HSJ_pt_f3d {
        double x, y, z;
}
/* Quaternion object */

class HSJ_quat {
        float x, y, z, w;
}
/* gen_ori transformation object */

class HSJ_gen_ori {
        double scale;
        HSJ_pt_d3d pos;
        HSJ_quat quat;
}
/* Single precision rgb color object */

class HSJ_color_rgb {
        float r, g, b;
}
/* Fog elemental animation operator attribute control object */

class HSJ_fog_attrs {
        int distance_policy;
        int world;
        double distance;
        float scale;
}
/* Clip elemental animation operator attribute control object */

class HSJ_clip_attrs {
        int distance_policy;
        int world;
        double distance;
}
```

### 10.3.2 Java interface routines for controlling generic objects

These routines check their arguments for validly. In general illegal operations are simply ignored, rather then attempting to rase exceptions. In some cases, the routine will return a success or failure indication.

```
public native double HSJGetPWorldTime();
```

Get the current vworld time value, expressed as a double precision floating point number indicating the number of seconds since the current invocation of the virtual universe, or shared network time.

```
public native void HSJOriAdd(int obj, float qx, float qy, float qz, float qw);
public native void HSJOriAddVec(int obj,HSJ_quat quat);
```

Add the specified quaternion rotation to the current orientation of the specified object.

```
public native void HSJPosAdd(int obj, double dx, double dy, double dz);
public native void HSJPosAddVec(int obj, HSJ_pt_d3d pos);
```

Set the position of the indicated object to the specified offset from its current position. The position will be relative the object's current vworld sub- coordinate system.

```
public native void HSJSetPos(int obj, double x, double y, double z);
public native void HSJSetPosVec(int obj, HSJ_pt_d3d pos);
```

Set the position of the indicated object to the xyz value passed. The position will be relative the object's current vworld sub-coordinate system.

```
public native void HSJSetOri(int obj, float qx, float qy, float qz, float qw);
public native void HSJSetOriVec(int obj, HSJ_quat);
```

Set the orientation of the indicated object to the quaternion value passed.

```
public native void HSJSetScale(int obj, double scale);
```

Set the scale of the indicated object to the indicated value.

```
public native void HSJSetRGB(int obj, float red, float green, float blue);
public native void HSJSetRGBVec(int obj, HSJ_color_rgb color);
```

Set the primary color of the indicated object to the indicated value.

```
public native void HSJSetOpacity(int obj, float opacity);
```

Set the primary opacity of the indicated object to the indicated value.

```
public native void HSJSetGori(int obj, double x, double y, double z, float qx, float
qz, float qw, double s);
public native void HSJSetGori(int obj, HSJ_gen_ori gori);
```

Set the generalized orientation of the specified object to the specified values: an xyz position, a quaternion orientation, and a uniform scale.

### 10.3.3 Java interface routines for controlling elemental animation operators

These routines check their arguments for validly. In general illegal operations are simply ignored, rather then attempting to rase exceptions. In some cases, the routine will return a success or failure indication.

```
public native void HSJStopAni(int obj);
public native void HSJStartAni(int obj);
```

These two routines cause all animation updates (including other Java code) attached to the specified object to cease/start again. This is *not* the same as pausing an object. This call should only be made when a section of animation will not be needed soon. Calls to these routines are automatically made whenever the viewer moves sufficiently far away from any animating object.

```
public native void HSJPauseAni(int obj, int pause, int recursive);
```

If the integer pause argument is 1, pause the indicated elemental animation object, in a way such that when it is un-paused it will continue changing right from wherever it was when it was paused.

If the pause argument was 0, restart the elemental animation object from where it was paused. This is achieved by updating the object's time_zero to be offset by exactly the length of time over which it was paused.

If the argument object is a group node, then its entire sub-graph will be traversed, and the pause will apply to all elemental animation objects found.

```
public native void HSJSetAniT0(int obj, double new_time_zero, int recursive);
```

Set the indicated objects (or recursive set of objects) time_zero attribute to the new time zero given, reflective to the current vworld time.

```
public native void HSJSetAniPhase(int obj, double new_phase, int recursive);
```

Set the indicated objects (or recursive set of objects) phase attribute to the new phase value.

```
public native void HSJSetAniLoop(int obj, int new_funct, int recursive);
```

Set the indicated objects (or recursive set of objects) animation loop attribute to a new value. Valid loop constants are: ELE_ANI_FORWARD, ELE_ANI_BACKWARD, and ELE_ANI_LOOP. These may be or-ed together to form the six legal looping combinations. All other combinations are invalid, and will not effect the current value of the animation object.

```
public native void HSJSetAniFBT(int obj, double new_forward_time, double new_backard
continuous, int recursive);
public native void HSJSetAniFBW(int obj, double new_forward_wait, double new_backard
continuous, int recursive);
```

Set the forward and backward time (or the forward and backward wait) to the new values given. If the continuous flag is zero, then the values are simply replaced. If the continuous flag is 1, then not only are the value set, but the time_zero value is offset such that even with the new attribute values, the value of alpha for the current frame will remain the same. This in effect allows one to change the speed of a particular elemental animation operator without causing any first order discontinuity in its visible operation.

```
public native void HSJSetAniAlpha(int obj, double new_alpha, int recursive);
```

Set the indicated objects alpha value directly. This doesn't really have any effect unless the object is paused or stoped, and even then will have no influence on what the object does after it is un-paused or un-stoped. Still, this is a good way to put a paused object into a known position. A common case is for obtaining switch behavior from a paused tloop object.

### 10.3.4 Java interface routines for controlling environment aspects

```
public native void SetLightType(int obj, int light_type);
```

Set the light type of the indicated light object.

```
public native void SetClipWorld(int obj, int front_clip_world);
```

Set the front clip world of the indicated clip object.

```
public native void SetFogWorld(int obj, int front_fog_world);
```

Set the front fog world of the indicated fog object.

... For all other slots of environmental objects.

### 10.3.5 Java interface routines for blocking on conditions

```
public native void HSJSleep(double time);
```

Block this Java thread until at least the specified number of seconds have gone by from the current time. If the time value is less than a frame time, zero, or less than zero, the process will still sleep until the next frame (e.g. won't continue during the current frame). HSJSleep(0.0) is the recommended way for a java thread to block until the next frame.

```
public native void HSJSleepHand(int obj, double dist);
public native void HSJSleepBody(int obj, double dist);
```

Sleep until the avatar's hand (wand/mouse), or until the avatar's body has approached the specified object to within the specified distance. (This is in effect an implicit proximity sphere.)

```
public native void HSJSleepAniEnd(int obj, int endpoint);
```

Sleep until the specified elemental animation operator has passed the next indicated endpoint (0 or 1). This is how a Java thread would "sleep until door finishes opening".

```
public native char HSJSleepKbd();
```

Sleep until the next time key is depressed on the keyboard. The value of the key pressed is the return value. *All* potentially active Java threads currently blocked on this event will get copies of the same value.

```
public native void HSJSleepSpaceBar();
```

Sleep until the next time the spacebar is depressed on the keyboard. This is mainly used for scripted slide presentations.

```
public native char HSJSleepHandKey();
```

Sleep until the next time key is depressed on the avatar's hand (wand/mouse). The value of the key pressed is the return value. (These keys are also called buttons on conventional mice.)

### 10.3.6 Java interface routines for controlling audio

This is an old (though implimented) 3D audio model. It should be replaced with a Java interface to the Sound object.

```
public native int PlaySound(String name, int loop, int deleteWhenDone, double dB);
public native int PlaySoundFromObject(int obj, String name, int loop, int deleteWhen
dB);
public native void StopSound(int sound);
public native void SetVolume(int sound, double dB);
```

```
public native void DeleteSound(int sound);
public native void DeleteSoundWhenFinished(int sound);
public native void LocateSoundWalls(double front, double left,double ceiling, double
double right, double ground);
```

### 10.3.7 Java interface routines for matrix, vector, and quaternion functions

Most of these routines check their input arguments for aliasing and do the right thing, e.g. vector_cross(a, b, a) will still compute the proper value.

```
public native int HSJMatEqual(double mat_in1[], double mat_in2[]);
```

Compare two 4x4 matrices for equality, element by element.

```
public native void HSJMatScale(double x_in, double y_in, double z_in, double mat_out
```

Create a 4x4 matrix with the specified non-uniform scales.

```
public native void HSJMatTrans(double x_in, double y_in, double z_in, double mat_out
```

Create a 4x4 matrix with the specified xyz translation.

```
public native void HSJMatScaleTrans(double x_in, double y_in, double z_in, double s_

                                    double mat_out[]);
```

Create a 4x4 matrix with the specified xyz translation and uniform scale.

```
public native void HSJMatTransScale(double s_in, double x_in, double y_in, double z_

                                    double mat_out[]);
```

Create a 4x4 matrix with the specified scaled xyz translation and uniform scale.

```
public native void HSJMatShearZ(double in_a, double in_b, double mat_out[]);
public native void HSJMatRotX(double in_theta, double mat_out[]);
```

Create a 4x4 matrix to rotate about the x axis by theta radians.

```
public native void HSJMatRotY(double in_theta, double mat_out[]);
```

Create a 4x4 matrix to rotate about the y axis by theta radians.

```
public native void HSJMatRotZ(double in_theta, double mat_out[]);
```

Create a 4x4 matrix to rotate about the z axis by theta radians.

```
public native void HSJMatTranspose(double mat_inout[]);
```

Transpose the passed matrix in place.

```
public native void HSJMatMulFpt(HSJ_pt_f3d pt_in, double  mat_in[], HSJ_pt_f3d pt_ou
```

Transform the input point by the matrix to the output point, single precision.

```
public native void HSJMatMulDpt(HSJ_pt_d3d pt_in, double  mat_in[], HSJ_pt_d3d pt_ou
```

Transform the input point by the matrix to the output point, double precision.

```
public native void HSJMatMulFnorm(HSJ_pt_f3d pt_in, double  mat_in[], HSJ_pt_f3d pt_
```

Transform the input normal by the matrix to the output normal, normalizing the result, single precision.

```
public native void HSJMatMulDnorm(HSJ_pt_d3d pt_in, double  mat_in[], HSJ_pt_d3d pt_
```

Transform the input normal by the matrix to the output normal, normalizing the result, double precision.

```
public native void HSJMatMulMat(double mat_in1[], double mat_in2[], double mat_out[]
```

Multiply the two matrices, producing the output matrix.

```
public native void HSJMatIdent(double mat_out[]);
```

Set the passed matrix to the idenity matrix.

public native void HSJMatDescale(double mat_in[], double mat_out[]);

Assuming that the input matrix has a uniform scale, return a version with the scale divided out.

```
public native void HSJFvecAdd(HSJ_pt_f3d in1, HSJ_pt_f3d in2, HSJ_pt_f3d out);
```

Add two single precision vectors, setting the third argument to the results.

```
public native void HSJFvecSub(HSJ_pt_f3d in1, HSJ_pt_f3d in2, HSJ_pt_f3d out);
```

Subtract two single precision vectors, setting the third argument to the results.

```
public native void HSJFvecSmul(HSJ_pt_f3d in1, double in2, HSJ_pt_f3d out);
```

Multiple the single precision vector by a double pericsion scaler value, returning a single precision vector in the third argument.

```
public native double HSJFvecDot(HSJ_pt_f3d in1, HSJ_pt_f3d in2);
```

Compute the dot product of two single precision vectors, returning the results as the double precision value of the function.

```
public native void HSJFvecCross(HSJ_pt_f3d in1, HSJ_pt_f3d in2, HSJ_pt_f3d out);
```

Compute the cross product of two single precision vectors, returning the results as a single precision vector in the third argument.

```
public native double HSJFvecLength(HSJ_pt_f3d in);
```

Compute the length of the single precision vector, returning the results as a double precision number as the return value of the function.

```
public native void HSJFvecNorm(HSJ_pt_f3d inout);
```

Normalize to unit length in place the single precision vector argument.

```
public native void HSJDvecAdd(HSJ_pt_d3d in1, HSJ_pt_d3d in2, HSJ_pt_d3d out);
```

Add two double precision vectors, setting the third argument to the results.

```
public native void HSJDvecSub(HSJ_pt_d3d in1, HSJ_pt_d3d in2, HSJ_pt_d3d out);
```

Subtract two double precision vectors, setting the third argument to the results.

```
public native void HSJDvecSmul(HSJ_pt_d3d in1, double in2, HSJ_pt_d3d out);
```

Multiple the double precision vector by a double pericsion scaler value, returning a double precision vector in the third argument.

```
public native double HSJDvecDot(HSJ_pt_d3d in1, HSJ_pt_d3d in2);
```

Compute the dot product of two double precision vectors, returning the results as the double precision value of the function.

```
public native void HSJDvecCross(HSJ_pt_d3d in1, HSJ_pt_d3d in2, HSJ_pt_d3d out);
```

Compute the cross product of two double precision vectors, returning the results as a double precision vector in the third argument.

```
public native double HSJDvecLength(HSJ_pt_d3d in);
```

Compute the length of the double precision vector, returning the results as a double precision number as the return value of the function.

```
public native void HSJDvecNorm(HSJ_pt_d3d inout);
```

Normalize to unit length in place the double precision vector argument.

```
public native void HSJMatInv2(double mat_in[], double mat_out[]);
```

Compute the inverse of the first passed matrix into the second matrix.

```
public native void HSJMatInv(double mat_inout[]);
```

Invert in place the passed matrix.

```
public native void HSJDmatToFmat(double dmat_in[], float fmat_out[]);
```

Copy the passed double precision matrix to the output single precision matrix.

```
public native void HSJFmatToDmat(float fmat_in[], double dmat_out[]);
```

Copy the passed single precision matrix to the output double precision matrix.

```
public native void HSJFmatToFmat(float fmat_in[], float fmat_out[]);
```

Copy the passed single precision matrix to the output single precision matrix.

```
public native void HSJDmatToDmat(double dmat_in[], double dmat_out[]);
```

Copy the passed double precision matrix to the output double precision matrix.

```
public native void HSJMatToQuat(double mat_in[], HSJ_quat quat_out);
```

Convert the passed double precision matrix to the output quaternion.

```
public native void HSJQuatToMat(HSJ_quat quat_in, double mat_out[]);
```

Convert the passed quaternion to the output double precision matrix.

```
public native void HSJQuatAdd(HSJ_quat quat_in1, HSJ_quat quat_in2, HSJ_quat quat_ou
```

Add the two passed quaternions, setting the third argument to the results.

```
public native void HSJQuatSub(HSJ_quat quat_in1, HSJ_quat quat_in2, HSJ_quat quat_ou
```

Subtract the two passed quaternions, setting the third argument to the results.

```
public native void HSJQuatInv(HSJ_quat quat_in, HSJ_quat quat_out);
```

Invert the first passed quaternion, setting the second argument to the results.

```
public native void HSJQuatSsub(HSJ_quat quat_in1, HSJ_quat quat_in2, HSJ_quat quat_o
```

Subtract the two passed quaternions, setting the third argument to the results.

```
public native double HSJMatToQuatCtrScale(double mat[], HSJ_quat q, HSJ_pt_d3d cente
scale);
```

Decompose the passed matrix into a rotational quaternions, a positional offset, and a scale.

```
public native double HSJMatToGenOri(double mat[], HSJ_gen_ori gori);
```

Decompose the passed matrix into a gen_ori structure.

```
public native double HSJGenOriToMat(HSJ_gen_ori gori, double mat[]);
```

Convert the gen_ori structure into a matrix.

```
public native void HSJQuatNorm(HSJ_quat q);
```

Normalize the passed quaternion in place.

```
public native void HSJQuatInterp(HSJ_quat q1, HSJ_quat q2, double alpha, HSJ_quat q3
```

Interpolate between the first two quaternions by the (0-1) alpha value, placing the resulting (normalized) quaternion into the fourth output argument.

---

# Chapter 11 - 3D Geometry

## 11.1 Geometry

HoloWeb only directly supports three types of geometry: points, lines, and triangles (and indirectly, text and image rasters). This section describes the general information content of HoloWeb geometry, in both its file representation, and its run-time representation.

**points**

Points are collections of vertex information; each point vertex contains a 3D location, and (optionally) an rgb color value. No form of points include normals; points are not subject to lighting.

**lines**

Lines are collections of vertex information and an indication of which pairs of vertices are connected to form lines. Each line vertex contains a 3D location, and (optionally) an rgb color value. No form of lines include normals; lines are not subject to lighting.

**triangles**

Triangles are collections of vertex information and an indication of which triples of vertices are connected to form triangles. Each vertex contains a 3D location, (optionally) a normal, and (optionally) an rgb color value *or* (optionally) a texture map coordinate. Triangles are always subject to lighting; to get the effect of no lighting, zero all material color values other than emissive.

**text**

Text object contains a variable length array of Unicode text, and appropriate font family and other fontology.

The representations of geometry for file transport vs. run-time script manipulation do not have to be the same, as the demands of each are quire different. The situation is akin to that of file transport vs. run-time script manipulation of 2D images. An encoding such as JPEG is well suited to efficient network transport, but is non-conducive to run-time manipulation in the extreme. HoloWeb's binary geometry compression format is exactly the same case.

Thus HoloWeb has two fairly different specific representations of geometry. The eight command language of HoloWeb's binary geometry compression format is described in the next chapter, and is primarily used for file and network transmission. The text file representation of geometry is primarily a cosmetic issue, and will not be addressed here.

For run-time representation of geometry, in many cases this is a moot point. If a geometry node (or any of its parent group nodes) has been marked READ_DISABLED WRITE_DISABLED, then the browser is free to convert the transmitted geometry into whatever optimized internal form it desires.

For the cases where run-time access / manipulation / creation of geometry is required, a form of un-compressed vertex representation is needed. For points, the format is variable length arrays of point vertices. For lines, the format is variable length arrays of line vertices, in which each vertex includes a move/draw bit. For triangles, the requirements are more complex, but some form of vertex array representation is likely.

No CSG or modeling operators are built into HoloWeb, although applications are free to define them in Java and pass their output in run-time geometric format.

The render semantics of these primitives is/will be covered in the render semantics section.

---

# Chapter 12 - 3D Geometry Compression

## 12.1 Geometry Compression Overview

HoloWeb's geometery compression format allows 3D geometery to be represented in an order of magnitude less space than most traditional 3D representations, with very little loss in object quality. The compression is achieved through several layer techniques.

**Compression**

First, the geometery to be compressed is converted into a generalized mesh form, which allows a triangle to be, on average, specified by 0.65 vertices.

Next the data for each vertex component of the geometry is converted to the most efficent representation format for its type, and then is quantized to as few bits as possible.

Now these quantized bits are differenced between successive vertices, and the results are modified-huffman encoded into self describing variable bit length data elements.

Finially, these variable length elements are strung together using HoloWeb's eight geometery commands into a finial compressed geometry block.

**Decompression**

Upon receipt in a Browser, compressed geometry blocks are decompressed into the local host's preferred geometry format by reversing the above process.

## 12.2 Chapter Organization

Before the bit details of the compression can be specified, several of the concepts used in geometry compression need elaboration. The first several sections are an expansion of the SIGGRAPH '95 paper on geometry compression.

**Generalized triangle strip**

This section is a refresher on the concept and semantics of a generalized triangle strip.

**Generalized triangle mesh**

This section introduces the concept and semantics of a generalized triangle mesh.

**Position representation and quantization**

This section describes the fixed point format used for 3D positional representation.

**Color representation and quantization**

This section describes the fixed point format used for color representation.

**Normal representation and quantization**

This section describes a novel folded table-based representation of surface normals, and the fixed point format of the resultant normals.

**Modified Huffman encoding**

This section describes the variant of Huffman delta encoding used for geometry compression.

**Geometry Compression Commands**

This section gives an overview of the eight geometry compression commands.

...

# 12.3 Generalized Triangle Strip

A generalized triangle strip is a generalization of the concept of a "zig-zag" and "start" triangle strip. A generalized triangle strip is a sequence of vertices in which each vertex contains a two bit replacement code. This replacement code defines how the present vertex is to be combined with previous vertices to form the next triangle. The replacement bits can also be thought of as a generalization of the "move/draw" bit used for lines.

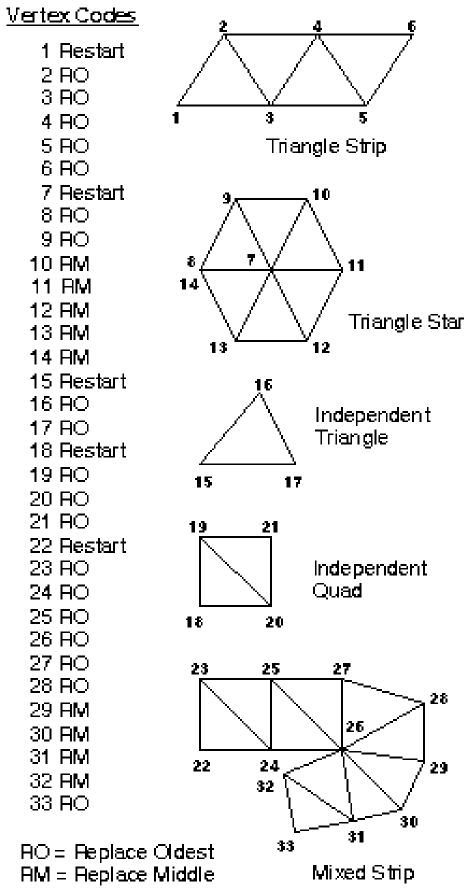



Figure 12-1: A Generalized Triangle Strip

A stack of the last three vertices used to form a triangle is kept. The three vertices are labeled oldest, middle, and newest. An incoming vertex of type *replace_oldest* causes the oldest vertex to be replaced by the middle, the middle to be replaced by the newest, and the incoming vertex becomes the newest. This corresponds to a PHIGS PLUS triangle strip (sometimes called a "zig-zag" strip). The replacement type replace_middle leaves the oldest vertex unchanged, replaces the middle vertex by the newest, and the incoming vertex becomes the newest. This corresponds to a triangle star or fan. The replacement type restart marks the oldest and middle vertices as invalid, and the incoming vertex becomes the newest. Generalized triangle strips must always start with this code. A triangle will be output only when a replacement operation results in three valid vertices. Restart corresponds to a "move" operation in polylines, and allows multiple unconnected variable-length triangle strips to be described by a single data structure passed in by the user, reducing the overhead. The generalized triangle strip’s ability to effectively change from "strip" to "star" mode in the middle of a strip allows more complex geometry to be represented compactly, and requires less input data bandwidth. The restart capability allows several pieces of disconnected geometry to be passed as one data block. Figure 12-1 shows a single generalized triangle strip, and the associated replacement codes.

Triangles are normalized such that the front face is always defined by a clockwise vertex order after transformation. To support this, there are two flavors of restart: *restart_clockwise* and *restart_counterclockwise*. The vertex order is reversed after every replace_oldest, but remains the same after every replace_middle.

# 12.4 Generalized Triangle Mesh

The first stage of geometry compression is to convert triangle data into an efficient linear strip form: the generalized triangle mesh. This is a near-optimal representation of triangle data, given fixed storage.

The existing concept of a generalized triangle strip structure allows for compact representation of geometry while maintaining a linear data structure. That is, the geometry can be extracted by a single monotonic scan over the vertex array data structure. This is very important for pipelined hardware implementations, a data format that requires random access back to main memory during processing is very problematic.

However, by confining itself to linear strips, the generalized triangle strip format leaves a potential factor of two (in space) on the table. Consider the geometry in Figure 12-2.

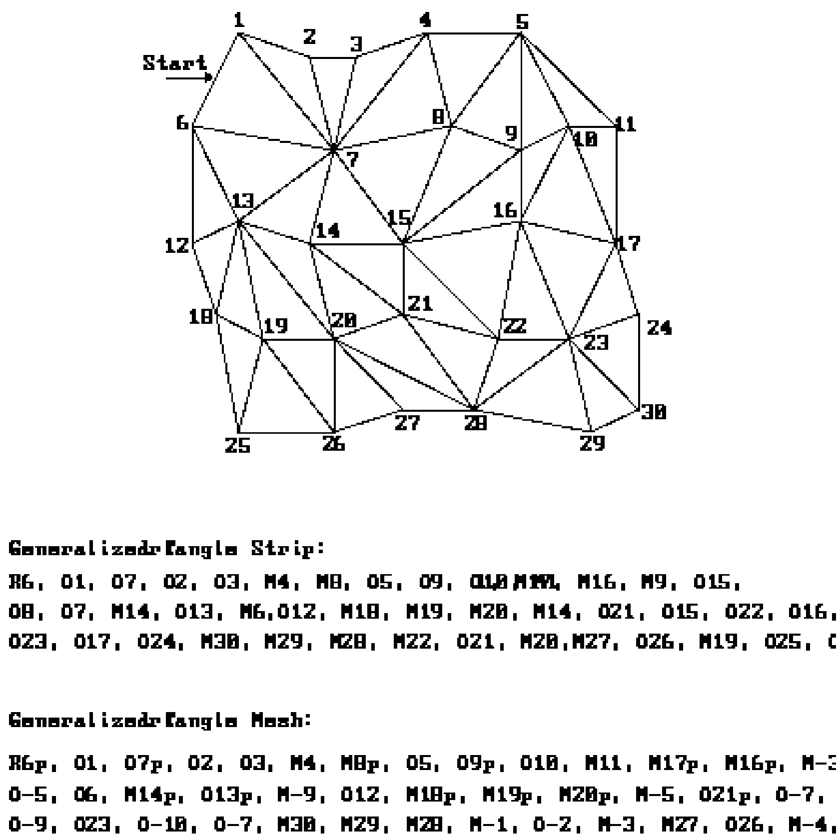



Figure 12-2

While it can be represented by one triangle strip, many of the interior vertices appear twice in the strip. This is inherent in any approach wishing to avoid references to old data. Some systems have tried using a simple regular mesh buffer to support re-use of old vertices, but there is a problem with this in practice: in general, geometry does not come in a perfectly regular rectangular mesh structure.

The generalized technique employed by geometry compression addresses this problem. Old vertices are explicitly pushed into a queue, and then explicitly referenced in the future when the old vertex is desired again. This fine control supports irregular meshes of nearly any shape. Any viable technique must recognize that storage is finite; thus the maximum queue length is fixed at 16, requiring a 4-bit index. We refer to this queue as the mesh buffer. The combination of generalized triangle strips and mesh buffer references is referred to as a generalized triangle mesh.

The fixed mesh buffer size requires all tessellators/re-strippers for compressed geometry to break up any runs longer than 16 unique references. Since geometry compression is not meant to be programmed directly at the user level, but rather by sophisticated tessellators/re-formatters, this is not too onerous a restriction. Sixteen old vertices allows up to 94% of the redundant geometry to avoid being re-specified.

Figure 12-2 also contains an example of a general mesh buffer representation of the surface geometry.

The language of geometry compression supports the four vertex replacement codes of generalized triangle strips (replace oldest, replace middle, restart clockwise, and restart counterclockwise), and adds another bit in each vertex header to indicate if this vertex should be pushed into the mesh buffer or not. The mesh buffer reference command has a 4-bit field to indicate which old vertex should be re-referenced, along with the 2-bit vertex replacement code. Mesh buffer reference commands do not contain a mesh buffer push bit; old vertices can only be recycled once.

Geometry rarely is comprised purely of positional data; generally a normal and/or color are also specified per vertex. Therefore, mesh buffer entries contain storage for all associated per-vertex information (specifically including normal and color). For maximum space efficiency, when a vertex is specified in the data stream, (per vertex) normal and/or color information should be directly bundled with the position information. This bundling is controlled by two state bits: bundle normals with vertices (bnv), and bundle colors with vertices (bcv). When a vertex is pushed into the mesh buffer, these bits control if its bundled normal and/or color are pushed as well. During a mesh buffer reference command, this process is reversed; the two bits specify if a normal and/or color should be inherited from the mesh buffer storage, or inherited from the current normal or current color. There are explicit commands for setting these two current values. An important exception to this rule occurs when an explicit "set current normal" command is followed by a mesh buffer reference, with the bnv state bit active. In this case, the former overrides the mesh buffer normal. This allows compact representation of hard edges in surface geometry. The analogous semantics are also defined for colors, allowing compact representation of hard edges in textures.

Two additional state bits control the interpretation of normals and colors when the stream of vertices is turned into triangles. The replicate normals over triangle (rnt) bit indicates that the normal in the final vertex that completes a triangle should be replicated over the entire triangle. The replicate colors over triangle (rct) bit is defined analogously.

# 12.5 Position Representation and Quantization

The 8-bit exponent of 32-bit IEEE floating-point numbers allows positions literally to span the known universe: from a scale of 100 billion light years, down to the radius of sub-atomic particles. However for any given tessellated object, the exponent is really specified just once by the current modeling matrix; within a given modeling space, the object geometry is effectively described with only the 24-bit fixed-point mantissa. Visually, in many cases far fewer bits are needed; thus the language of geometry compression supports variable quantization of position data down to as little a one bit. The maximum number of bits supported is at most 16 bits of precision per component of position.

We still assume that the position and scale of the local modeling spaces are specified by full 32-bit or 64-bit floating-point coordinates. If sufficient numerical care is taken, multiple such modeling spaces can be stitched together without cracks, forming seamless geometry coordinate systems with much greater than 16-bit positional precision.

Most geometry is local, so within the 16-bit (or less) modeling space (of each object), the delta difference between one vertex and the next in the generalized mesh buffer stream is very likely to be less than 16 bits in significance. Indeed one can histogram the bit length of neighboring position deltas in a batch of geometry, and based upon this histogram assign a variable length code to compactly represent the vertices. The typical coding used in many other similar situations is customized Huffman code; this is the case for geometry compression. The details of the coding of position deltas will be postponed until later, where they can be discussed in the context of color and normal delta coding as well.

# 12.6 Color Representation and Quantization

We treat colors similar to positions, but with a smaller maximum accuracy. Thus rgba color data is first quantized to 12-bit unsigned fraction components. These are absolute linear reflectivity values, with 1.0 representing 100% reflectivity. An additional parameter allows color data effectively to be quantized to any amount less than 12 bits, i.e. the colors can all be within a 5- 5-5 rgb color space. (The a field is optional, controlled by the *color alpha present* (cap) state bit.) Note that this decision does not necessarily cause mock banding on the final rendered image; individual pixel colors are still interpolated between these quantized vertex colors, and also are subject to lighting.

The same delta coding is used for color components as is used for positions. Compression of color data is where geometry compression and traditional image compression face the most similar problem. However, many of the more advanced techniques for image compression were rejected for geometry color compression because of the difference in focus.

Image compression makes several assumptions about the viewing of the decompressed data that cannot be made for geometry compression. In image compression, it is known a priori that the pixels appear in a perfectly rectangular array, and that when viewed, each pixel subtends a narrow range of visual angles. In geometry compression, one has almost no idea what the relationship between the viewer and the rasterized geometry will be.

In image compression, it is known that the spatial frequency on the viewer's eyes of the displayed pixels is likely higher than the human visual system's color acuity. This is why colors are usually converted to yuv space, so that the uv color components can be represented at a lower spatial frequency than the y (intensity) component. Usually the digital bits representing the sub-sampled uv components are split up among two or more pixels. Geometry compression can't take advantage of this because the display scale of the geometry relative to the viewer's eye is not fixed. Also, given that compressed triangle vertices are connected to 4 - 8 or more other vertices in the generalized triangle mesh, there is no consistent way of sharing "half" the color information across vertices.

Similar arguments apply for the more sophisticated transforms used in traditional image compression, such as the discrete cosine transform. These transforms assume a regular (rectangular) sampling of pixel values, and require a large amount of random access during decompression.

## 12.7 Normal Representation and Quantization

Probably the most innovative concept in geometry compression is the method of compressing surface normals. Traditionally 96-bit normals (three 32-bit IEEE floating-point numbers) are used in calculations to determine 8-bit color intensities. 96 bits of information theoretically could be used to represent 296 different normals, spread evenly over the surface of a unit sphere. This is a normal every 2^-46 radians in any direction. Such angles are so exact that spreading out angles evenly in every direction from earth you could point out any rock on Mars with sub-centimeter accuracy.

But for normalized normals, the exponent bits are effectively un-used. Given the constraint $|N| = 1$, at least one of $N$x, $N$y, or $N$z *must* be in the range of 0.5 to 1.0. During rendering, this normal will be transformed by a composite modeling orientation matrix T: $N = N \, . \, T$.

Assuming the typical implementation in which lighting is performed in world coordinates, the view transform is not involved in the processing of normals. If the normals have been pre-normalized, then to avoid redundant re- normalization of the normals, the composite modeling transformation matrix T is typically pre-normalized to divide out any scale changes, and thus:

$$T_{0,0}^{2} + T_{1,0}^{2} + T_{2,0}^{2} = 1$$

, etc.

During the normal transformation, floating-point arithmetic hardware effectively truncates all additive arguments to the accuracy of the largest component. The result is that for a normalized normal, being transformed by a scale preserving modeling orientation matrix, in all but a few special cases, the numerical accuracy of the transformed normal value is reduced to no more than 24-bit fixed-point accuracy.

Even 24-bit normal components are still much higher in angular accuracy than the (repaired) Hubble space telescope. After empirical tests, it was determined that an angular density of 0.01 radians between normals gave results that were not visually distinguishable from finer representations. This works out to about 100,000 normals distributed over the unit sphere. In rectilinear space, these normals still require high accuracy of representation; we chose to use 16-bit components including one sign and one guard bit.

This still requires 48 bits to represent a normal. But since we are only interested in 100,000 specific normals, in theory a single 17-bit index could denote any of these normals. The next section shows how it is possible to take advantage of this observation.

### 12.7.1 Normal as Indices

The most obvious hardware implementation to convert an index of a normal on the unit sphere back into a *Nx Ny Nz* value, is by table look-up. The problem is the size of the table. Fortunately, there are several symmetry tricks that can be applied to vastly reduce the size of the table (by a factor of 48).

First, the unit sphere is symmetrical in the eight quadrants by sign bits. In other words, if we let three of the normal representation bits be the three sign bits of the xyz components of the normal, then we only need to find a way to represent one eighth of the unit sphere.

Second, each octant of the unit sphere can be split up into six identical pieces, by folding about the planes $x = y$, $x = x$, and $x = z$. (See Figure 12-3.) The six possible sextants are encoded with another three bits. Now only 1/48 of the sphere remains to be represented.

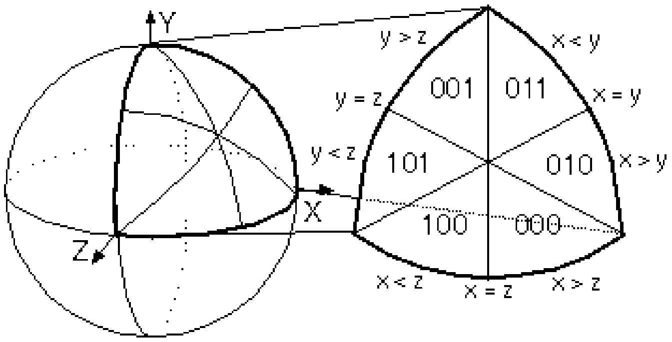



Figure 12-3. Encoding of the six sextants of each octant of a sphere

This reduces the 100,000 entry look-up table in size by a factor of 48, requiring only about 2,000 entries, small enough to fit into an on-chip ROM look-up table. This table needs 11 address bits to index into it, so including our previous two 3-bit fields, the result is a grand total of 17 bits for all three normal components.

Representing a finite set of unit normals is equivalent to positioning points on the surface of the unit sphere. While no perfectly equal angular density distribution exists for large numbers of points, many near-optimal distributions exist. Thus in theory one of these with the same sort of 48-way symmetry described above could be used for the decompression look-up table. However, several additional constraints mandate a different choice of encoding:

1) We desire a scalable density distribution. This is one in which zeroing more and more of the low order address bits to the table still results in fairly even density of normals on the unit sphere. Otherwise a different look-up table for every encoding density would be required.

2) We desire a delta-encodable distribution. Statistically, adjacent vertices in geometry will have normals that are nearby on the surface of the unit sphere. Nearby locations on the 2D space of the unit-sphere surface are most succinctly encoded by a 2D offset. We desire a distribution where such a metric exists.

3) Finally, while the computational cost of the normal encoding process is not too important, in general, distributions with lower encoding costs are preferred.

For all these reasons, we decided to use a regular grid in the angular space within one sextant as our distribution. Thus rather than a monolithic 11-bit index, all normals within a sextant are much more conveniently represented as two 6-bit orthogonal angular addresses, revising our grand total to 18-bits. Just as for positions and colors, if more quantization of normals is acceptable, then these 6-bit indices can be reduced to fewer bits, and thus absolute normals can be represented using anywhere from 18 to as few as 6 bits. But as will be seen, we can delta encode this space, further reducing the number of bits required for high quality representation of normals.

### 12.7.2 Normal Encoding Parameterization

Points on a unit radius sphere are parameterized by two angles, $\theta$ and $\phi$ using spherical coordinates. $\theta$ is the angle about the y axis; $\phi$ is the longitudinal angle from the y=0 plane. The mapping between rectangular and spherical coordinates is:

$$x = \cos\theta \cdot \cos\phi \qquad y = \sin\phi \qquad z = \sin\theta \cdot \cos\phi \quad (1)$$

Points on the sphere are folded first by octant, and then by sort order of xyz into one of six sextants. All the table encoding takes place in the positive octant, in the region bounded by the half spaces:

$$x \geq z \qquad z \geq y \qquad y \geq 0$$

This triangular-shaped patch runs from 0 to p/4 radians in $\theta$ , and from 0 to as much as 0.615479709 radians in $\phi : \phi_{max}$ .

Quantized angles are represented by two n-bit integers $\hat{\theta}_n$ and $\hat{\phi}_n$ , where n is in the range of 0 to 6. For a given n, the relationship between these indices $\theta$ and $\phi$ is

$$\theta(\hat{\theta}_n) = \operatorname{asin}\tan\left(\phi_{max} \cdot (n - \hat{\theta}_n) / 2^n\right)$$
$$\phi(\hat{\phi}_n) = \phi_{max} \cdot \hat{\phi}_n / 2^n \qquad (2)$$

These two equations show how values of $\hat{\theta}_n$ and $\hat{\phi}_n$ can be converted to spherical coordinates $\theta$ and $\phi$ , which in turn can be converted to rectilinear normal coordinate components via equation 1.

To reverse the process, e.g. to encode a given normal N into $\hat{\theta}_n$ and $\hat{\phi}_n$ , one cannot just invert equation 2. Instead, the N must be first folded into the canonical octant and sextant, resulting in N'. Then N' must be dotted with all quantized normals in the sextant. For a fixed n, the values of $\hat{\theta}_n$ and $\hat{\phi}_n$ that result in the largest (nearest unity) dot product define the proper encoding of N.

Now the complete bit format of absolute normals can be given. The uppermost three bits specify the octant, the next three bits the sextant, and finally two n- bit fields specify $\hat{\theta}_n$ and $\hat{\phi}_n$ . The 3-bit sextant field takes on one of six values, the binary codes for which are shown in Figure 12-2.

This discussion has ignored some details. In particular, the three normals at the corners of the canonical patch are multiply represented (6, 8, and 12 times). By employing the two unused values of the sextant field, these normals can be uniquely encoded as 26 special normals.

This representation of normals is amenable to delta encoding, at least within a sextant. (With some additional work, this can be extended to sextants that share a common edge.) The delta code between two normals is simply the difference in $\hat{\theta}_n$ and $\hat{\phi}_n$ : $\Delta\hat{\theta}_n$ and $\Delta\hat{\phi}_n$.

# 12.8 Modified Huffman Encoding

There are many techniques known for minimally representing variable-length bit fields. For geometry compression, we have chosen a variation of the conventional Huffman technique.

The Huffman compression algorithm takes in a set of symbols to be represented, along with frequency of occurrence statistics (histograms) of those symbols. From this, variable length, uniquely identifiable bit patterns are generated that allow these symbols to be represented with a near-minimum total number of bits, assuming that symbols do occur at the frequencies specified.

Many compression techniques, including JPEG, create unique symbols as tags to indicate the length of a variable-length data-field that follows. This data field is typically a specific-length delta value. Thus the final binary stream consists of (self-describing length) variable length tag symbols, each immediately followed by a data field whose length is associated with that unique tag symbol.

The binary format for geometry compression uses this technique to represent position, normal, and color data fields. For geometry compression, these <tag, data fields are immediately preceded by (a more conventional computer instruction set) op-code field. These fields, plus potential additional operand bits, are referred to as *geometry instructions* (see figure 3).

Traditionally, each value to be compressed is assigned its own associated label, e.g. an xyz delta position would be represented by three tag-value pairs. However, the delta xyz values are not uncorrelated, and we can get both a denser and simpler representation by taking advantage of this fact. In general, the xyz deltas statistically point equally in all directions in space. This means that if the number of bits to represent the largest of these deltas is n, then statistically the other two delta values require an average of n-1.4 bits for their representation. Thus we made the decision to use a single field-length tag to indicate the bit length of Dx, Dy, *and* Dz. This also means that we cannot take advantage of another Huffman technique that saves somewhat less than one more bit per component, but our bit savings by not having to specify two additional tag fields (for Dy and Dz) outweigh this. A single tag field also means that a hardware decompression engine can decompress all three fields in parallel, if desired.

Similar arguments hold for deltas of rgba values, and so here also a single field-length tag indicates the bit-length of the Dr, Dg, Db, and Da (if present) fields.

Both absolute and delta normals are also parameterized by a single value (n), which can be specified by a single tag.

We chose to limit the length of the Huffman tag field to the relatively small value of six bits. This was done to facilitate high-speed low-cost hardware implementations. (A 64-entry tag look-up table allows decoding of tags in one clock cycle.) Three such tables exist: one each for positions, normals, and colors. The tables contain the length of the tag field, the length of the data field(s), a data normalization coefficient, and an absolute/relative bit.

One additional complication was required to enable reasonable hardware implementations. As will be seen in a later section, all instruction are broken up into an eight-bit header, and a variable length body. Sufficient information is present in the header to determine the length of the body. But in order to give the hardware time to process the header information, the header of one instruction must be placed in the stream *before the body of the previous instruction*. Thus the sequence ... B0 H1B1 H2B2 H3 ... has to be encoded:

... H1 B0 H2 B1 H3 B2 ... .

## 12.9 Geometry Compression Commands

HoloWeb's geometry comprerssion protocoll defines eight commands to be used in specifying 3D geometry and certain affilited attributes. This section will give a brief overview of these commands and some of their semantics. More detail of these commands, including their bit lay-out, will be given in the next sections.

**vertex**

The primary command is *vertex*. A vertex command always specifies a 3D position, two generalized triangle strip replacement bits (*rep*), a mesh buffer push (*mbp*) bit, and may optionally specify a normal and/or a color (or texture map coordinate). The presence of normal or color data within a vertex command is controlled by two bundeling bits: *bnv* and *bcv*, respectively.

**normal, color**

There are also two stand alone commands for specifing normals and colors: *normal* and *color*. These commands may be freely intersperced with vertex commands, and semantically have (nearly) the same effect as normals or colors bundled directly with a normal.

Once a color or normal value is specified, either directly or bundled with a vertex command, that color or normal will remain in effect as the current color or normal until a new value is specified. In this fastion, for example, a constant material color may be specified to apply to s forthcoming sequence of non- color bundled vertices.

**set state**

The *set state* command update the value of the six state bits. Two of these bits are the normal and color bundeling bits, the other four will be described later.

**mesh buffer reference**

The *mesh buffer reference* command allows any of the sixteen vertices most recently pushed into the mesh buffer to be re-used in place of a vertex command at this point. Two vertex replacement bits are also present.

**set table**

The set *table command* allows a range of entries in one of the three huffman decompression tables all to be set to the same new value.

**pass through**

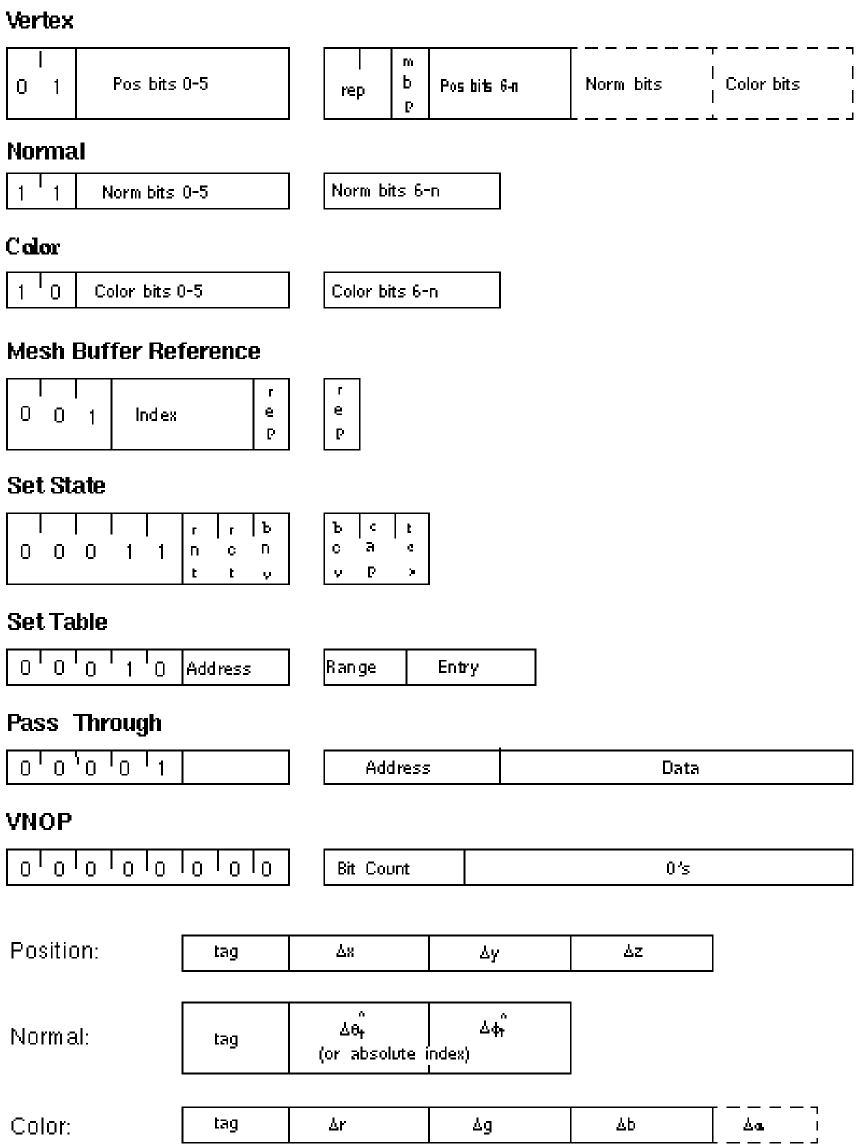
The pass through command allows other data to be embeded in the compression stream.

**vnop**

The variable length no-operation command allows the compression bit-stream to be padded by a specified number of bits. This allows portions of the compression data to be 32-bit alligned when desired.

# 12.10 Bit layout of Geometry Decompression Commands

The following figure shows the bit level lay out of the eight geometry decompression commands. Each command has a unique opcode, and then some (possible variable) number of arguments. The actual bit-length of many of the components may vary, and if so, a unique (dynamic) huffman tag at the very start of any variable length argument delimits the size of the argument.



## 12.11 Semantics of Geometry Decompression Commands

Thes next few sections present the formal semantics of the geometry decompression commands.

### 12.11.1 Header, Body to Variable Length Command

The ...

### 12.11.2 Variable Length Command to Command

The ...

### 12.11.3 Delta position to position

**absolute_position(x, y, z):**

```
cur_x  x, cur_y  y, cur_z  z
```

**relative_position**(Dx, Dy, Dz):

```
cur_x  cur_x + Dx, cur_y  cur_y + Dy, cur_z  cur_z + Dz
```

### 12.11.4 Delta color to color

**absolute_color(r, g, b {,** a}):

```
cur_x  x, cur_y  y, cur_z  z, {cur_a  a }
```

**relative_color**(Dr, Dg, Dg {, Da}):

```
cur_x  cur_x + Dx, cur_y  cur_y + Dy, cur_z  cur_z + Dz,
{cur_a  cur_a + Da }
```

### 12.11.5 Encoded delta normal to encoded normal

```
State: cur_oct, cur_sex, cur_u, cur_v
```

**absolute_normal(oct, sex, u, v):**

```
cur_oct  oct, cur_sex  sex, cur_u  u, cur_v  v,
```

**relative_normal**(Du, Dv):

```
cur_u  cur_u + Du, cur_v  cur_v + Dv,
if (cur_u < 0)
        cur_u  ~cur_u, cur_sex  flip_u[cur_sex]
```

```
else if (cur_v < 0)
        cur_v  ~cur_v, cur_oct  cur_oct <xor flip_v[cur_sex]
else if (cur_u + cur_v  64)
        cur_u  64 - cur_u, cur_v  64 - cur_v,
        cur_sex  flip_uv[cur_sex]
```

### 12.11.6 encoded normal to rectliner normal

```
nx  norms[v,u].nx,    ny  norms[v,u].ny,    nz  norms[v,u].nz,
if (cur_sex & 4) t  nx, nx  nz, nz  t
if (cur_sex & 2) t  ny, ny  nz, nz  t
if (cur_sex & 1) t  nx, nx  ny, ny  t
if (cur_oct & 1) nz  -nz
if (cur_oct & 2) ny  -ny
if (cur_oct & 4) nx  -nx
```

## 12.12 Semantics of Vertices

The formal seamntics of the compression format is best described by a state description of the decompression process. It must be emphasied that these state descriptions are given to illustrate the formal semantics, not an efficent implementation.

### 12.12.1 Command to Vertex

This section describes the state change semantics caused by each command to generate the next output vertex, prior to assembly into triangles. The internal state consists of the six mode bits, a current normal and current color, normal_overide and color_overide bits, the sixteen mesh buffer vertices, and the current mesh index.

#### normal(n):

```
current_normal  n, normal_overide  1
```

#### color(c):

```
current_color  c, color_overide  1
```

#### vertex(rep, mbp, p {, n} {, c}):

```
current_position  p,
if (bnv) current_normal  n,
if (bcv) current_color  c,
output_vertex(rep, current_position, current_normal, current_color)
if (mbp) mesh_buffer[oldest_mesh_index].position  p
if (mbp && bnv) mesh_buffer[mesh_index].normal  n
if (mbp && bcv) mesh_buffer[mesh_index].color  c
if (mbp) mesh_index  (mesh_index+1) & 15
normal_overide  0, color_overide  0
```

#### mesh buffer reference(rep, i):

```
current_position
                        mesh_buffer[(mesh_index - i) & 15].position
if (bnv && !normal_overide)
```

```
    current_normal mesh_buffer[(mesh_index - i) & 15].normal
if (bcv && !color_overide)
    current_color  mesh_buffer[(mesh_index - i) & 15].color
output_vertex(rep, current_position, current_normal, current_color)
```

**set state(new_bnv, new_bcv, new_cap, new_rnt, new_rct, new_tex):**

```
bnv  new_bnv,
bcv  new_bcv,
cap  new_cap,
rnt  new_rnt,
rct  new_rct,
tex  new_tex
```

**set table(address, range, entry):**

```
    ...
```

**pass through(data):**

```
    (null)
```

**vnop(length):**

```
    (null)
```

### 12.12.2 Vertex to Intermediate Triangle

This section describes the formal semantics of assemblying vertices with replacement commands into nearly finished triangles: the semantics of generalized triangle strips.

**output_vertex(restart clockwise, newv):**

```
newest  newv, number_of_vertices  1, ccw = 0
```

**output_vertex(restart counterclockwise, newv):**

```
newest  newv, number_of_vertices  1, ccw = 1
```

**output_vertex(replace_middle, newv):**

```
if (number_of_vertices < 2)
        midlest  newest, newest  newv, number_of_vertices++
else if (number_of_vertices < 3)
        oldest  midlest, midlest  newest, newest  newv,
        number_of_vertices++,
        intermediate_triangle(ccw, oldest, midlest, newest)
else if (number_of_vertices == 3)
        midlest  newest, newest  newv,
        intermediate_triangle(ccw, oldest, midlest, newest)
```

**output_vertex(replace_oldest, newv):**

```
if (number_of_vertices < 2)
        midlest  newest, newest  newv, number_of_vertices++
```

```
else if (number_of_vertices < 3)
        oldest  midlest, midlest  newest, newest  newv,
        number_of_vertices++,
        intermediate_triangle(ccw, oldest, midlest, newest)
else if (number_of_vertices == 3)
        oldest  midlest, midlest  newest, newest  newv,
        ccw = 1 - ccw,
        intermediate_triangle(ccw, oldest, midlest, newest)
```

### 12.12.3 Intermediate Triangle to Final Triangle

The final stage is to take into account the current rnt and rct mode bits settings. These control the seamtics of the normal and color vertex data: is such data specific to its parent vertex, or should it be replicated across its parent triangle? The semantics of the counterclockwise bit also can be expressed here, thus the final triangles can always be assumed to be front facing when their vertices appear in counterclockwise order.

**intermediate_triangle(ccw, v1, v2, v3):**

```
if (!rnt && !rct && ccw)
    final_triangle(v1.position, v1.normal, v1.color,
                                        v2.position, v2.normal, v2.color,
                                        v3.position, v3.normal, v3.color)
else if (!rnt && !rct && !ccw)
    final_triangle(v2.position, v2.normal, v2.color,
                                        v1.position, v1.normal, v1.color,
                                        v3.position, v3.normal, v3.color)
else if (rnt && !rct && ccw)
    final_triangle(v1.position, v3.normal, v1.color,
                                        v2.position, v3.normal, v2.color,
                                        v3.position, v3.normal, v3.color)
else if (rnt && !rct && !ccw)
    final_triangle(v2.position, v3.normal, v2.color,
                                        v1.position, v3.normal, v1.color,
                                        v3.position, v3.normal, v3.color)
else if (!rnt && rct && ccw)
    final_triangle(v1.position, v1.normal, v3.color,
                                        v2.position, v2.normal, v3.color,
                                        v3.position, v3.normal, v3.color)
else if (!rnt && rct && !ccw)
    final_triangle(v2.position, v2.normal, v3.color,
                                        v1.position, v1.normal, v3.color,
                                        v3.position, v3.normal, v3.color)
else if (rnt && rct && ccw)
    final_triangle(v1.position, v3.normal, v3.color,
                                        v2.position, v3.normal, v3.color,
                                        v3.position, v3.normal, v3.color)
else if (rnt && rct && !ccw)
    final_triangle(v2.position, v3.normal, v3.color,
                                        v1.position, v3.normal, v3.color,
                                        v3.position, v3.normal, v3.color)
```

## 12.13 Outline of Geometry Process

HoloWeb only formally defines the geometry compression format and the decompression semantics. Authoring tools are free to empoly whatever geometry compression algorithms they choose so long as the results aheards to the specifications described in the previous sections.

However, in order to further document the semantics of the geometry compression format, and overview of one particular geometry compression algorithm is given here.

### 12.13.1 Compressing Geometry Data

Group geometry to be compressed into separate ridged objects. Typically such objects will be individually culled during rendering, so one should not too extensively join objects prior to compression. In optimized systems, the granularity of object splitting will be computed by a algorithm that takes culling optimization into account.

### 12.13.2 Convert to Generalized Mesh Format

Once a group of geometry has been identified, it next is converted into generalized mesh format.

The next step is the quantization of the positions, colors, normals, and/or texture map coordinates of the geometry. All these quantizations can be varied within the geometry, but for simplicity a single fixed quantization of each will be assumed here.

### 12.13.3 Position

*Normalize the position data*

The containing bounding box for the object is computed. This is the minimal box such that all geometry vertices are contained within it. The vertices are then all normalized to be contained within this bounding box by first subtracting the xyz location of the bounding box center from the vertex xyz; and then dividing all the xyz vertex values by the half length of the longest side of the bounding box. Thus all normalized positions with be within the 1 unit cube. A constant matrix transform corresponding to an offset to the center of the bounding box, and an inverse scale by the half length of the longest side of the bounding box is created as a prolog for the geometry data.

*Quantize the position data*

Assuming that position data is to be quantized to n bits, each vertex position component should be multiplied by the value of 2n, and then rounded to the nearest integer.

### 12.13.4 Normals

*Normalize the normals*

Each normal should be normalized to unit length.

*Quantize the xyz components of the normal to 14 bits accuracy*

Each normal component should be multiplied by 214, then rounded to the nearest integer, and then converted back to floating point representation and divided back by 214.

*Fold the normal to the positive octant*

If an xyz component of the normal is negative, invert it, and save the original sign bits as a three bit octant value:

```
oct = 0;
if(nx < 0.0) oct |= 4, nx = -nx
if(ny < 0.0) oct |= 2, ny = -ny
if(nz < 0.0) oct |= 1, nz = -nz
```

*Fold the normal to the nx nz ny sextant*

Check (in exactly the following order):

```
sex = 0;
if (nx < ny) t = nx, nx = ny, ny = t, sex |= 1
if (nz < ny) t = ny, ny = nz, nz = t, sex |= 2
if (nx < nz) t = nx, nx = nz, nz = t, sex |= 4
```

*Match the nearest quantize normal representation*

Take the dot product of the normal with each of the quantized reference normals in the table for the specified number of quantized normal bits. That uv normal index for the reference normal that gives the greatest (nearest unity) dot product result is the new quantized normal representation

(along with the octant and sextant representation)

*Check for special normals*

...

## 12.13.5 Colors

The colors are assumed to be in a 0.0 to 1.0 representation to begin with.

*Quantize the color values*

Assuming that color data is to be quantized to n bits, each vertex color component (r g b and optionally a) should be multiplied by the value of 2n, and then rounded to the nearest integer.

Texture map coordinates may appear in place of color components within the compression stream, as controlled by the *tex* state bit. If 2D texture mapping is desired, then the uv texture coordinate values take the place of the rg color components in the compression stream (b is still present but disposed of).

## 12.13.6 Collect Delta Code Statistics

Make a pass in generalized mesh order through all vertices in the geometry. For each successive pair of vertices, compute the difference between their component values, compute the bit-length of this (signed) difference, and histogram this bit length. Specifically for each component type:

## 12.13.7 Position Delta Code Statistics

Compute Dx, Dy, and Dz. Determine which of these has the greatest magnitude. Compute the number of bits for this component, including one sign bit. This is the length to be histogrammed for positions.

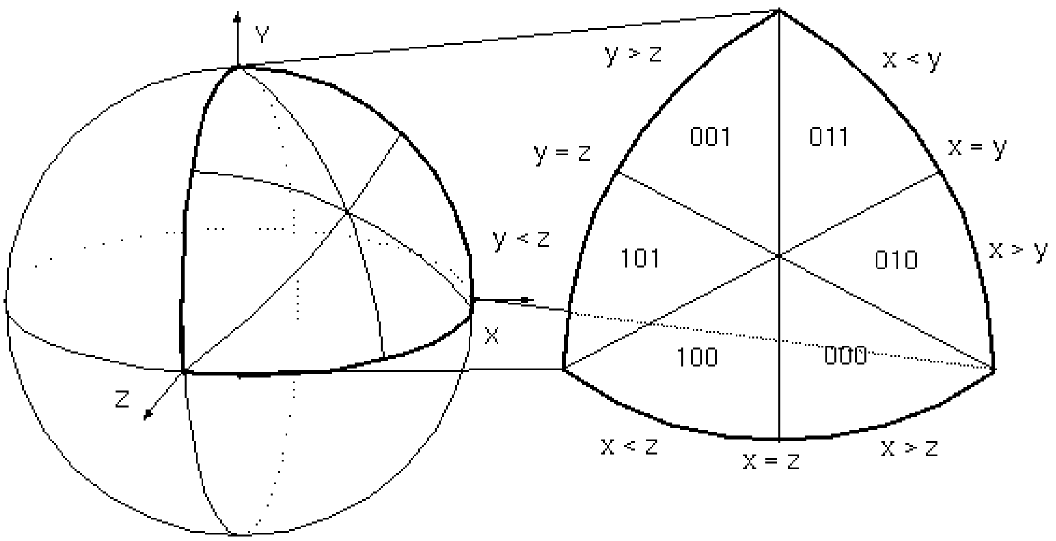
## 12.13.8 Color Delta Code Statistics

Compute Dr, Dg, Db, and Da (if present). Determine which of these has the greatest magnitude. Compute the number of bits for this component, including one sign bit. This is the length to be histogrammed for colors.

## 12.13.9 Normal Delta Code Statistics

For a given pair of normals, check to see if they have the same octant and sextant. If so, compute Du and Dv. Determine which of these has the greatest magnitude. Compute the number of bits for this component, including one sign bit. This is the length to be histogrammed for this pair of normals.

If the normals have different sextants and/or octants, check to see if their sextants share and edge. Depending on what type of edge their share, the delta including the change in edges is encoded in one of three ways: u + Du < 0, v + Dv < 0, and u + Du + v + Dv > 64. Each case is discussed in a paragraph below. The sextant numbers are from the binary codes shown in the figure below.




Sextants 0 and 4, 1 and 5, and 2 and 3 share the u = 0 edge. When crossing this boundary, Du becomes ~ - last_u. This will generate a negative cur_u value during decompression, which causes the decompressor to invert cur_u and look up the new sextant in a table.

Sextants 0 and 2, 1 and 3, and 4 and 5 share the u + v = 64 edge. Du becomes 64 - u - last_u and Dv becomes 64 - v - last_v. When cur_u + cur_v > 64, the decompressor sets cur_u = 64 - cur_u and cur_v = 64 - cur_v, and a table look up determines the new sextant.

Each sextant shares the v = 0 edge with its corresponding sextant in another octant. When in sextants 1 or 5, the normal moves across the x-axis, across the y-axis for sextants 0 or 4, and across the z-axis for sextants 2 or 3. Dv becomes ~v - last_v. The decompressor inverts a negative cur_v and performs a table look up for a mask to exclusive-or with the current octant value.

Otherwise the normals can not be delta encoded, and so the second (target) normal must be represented by an absolute reference to its three octant, three sextant, and 2 n-bit u v addresses. This is the length to be histogrammed for this pair of normals.

### 12.13.10 Assign Huffman Tags

Encode data into variable bit length decompression commands

One can use an algorithm similar to the one used by the JPEG image compression standard. The main differences are how codes are reassigned when their lengths exceed the maximum code length and how the data bits are encoded in the compressed data stream.

The frequencies of the data lengths are used as leaf nodes in a binary tree. The algorithm used to generate the tree places the less frequent codes deeper in the tree. After the tree is built, the traversal path to a leaf node becomes its Huffman code and the depth in the tree becomes its code length.

Codes generated with a length greater than six, the maximum code length, must be shortened. These nodes are merged with more frequent nodes by increasing the number of sign bits included with the smaller data length.

---

# Chapter 13 - 3D Render Semantics

## 13.1 3D Render Semantics

Defines additional details of the render semantics of HoloWeb objects.

---

# Chapter 14 - HoloWeb Example Scenes

## 14.1 HoloWeb Example Scenes

Several simple examples of the use of HoloWeb objects in the construction of virtual worlds are given.

Example Java code:

```
void door(int obj1)
{
    while (true) {
        HSJSleepKbd();
        HSJSetAniT0(obj1, 0.0, 1);
        HSJSleep(4.1);
    }
}
```

This routine door has exactly one input argument: a reference to a one-shot elemental animation object that when fired will cause a door to open. The Java routine will wait for a keyboard event (by sleeping), and then fire off the door opening sequence.

Notice that it does not matter to the Java routine wither the door rotates open via a orientation spline, or slides away into the wall, ceiling, or floor. The Java code here purely exists to start and stop action.

If the door was to make a sound as it opened, this also would be the responsibility of the Java routine. Right after resetting time_zero on the door object, the Java routine would make a call to initiate the desired sound.

# 14.2 Sample Browser semantics: HoloView

While HoloWeb is not define a complete 3D browser, it mandates many semantics of HoloWeb compliant browsers. This chapter illustrates the use of HoloWeb in building a 3D browser, by describing the features of the sample HoloWeb browser, HoloView.

# 14.3 HoloWeb binary format

Defines the bit level details of the HoloWeb binary file format, beyond those for compressed geometry, described in an earler chapter.

# 14.4 HoloWeb text format

Defines the details of the HoloWeb text file format.