Chapter 13

Virtual Environment Models

G. Drew Kessler
Lehigh University

(Contact information good until Sept, 1999)
Email: drewk@central.cis.upenn.edu
Phone: (215) 573-2815
Fax: (215) 898-0587
1. Introduction

The capabilities and limitations of users drive many of the design decisions of virtual environment technology. These considerations affect both the design of the hardware and the design of the software. To be interactive, a VE software application needs to constantly present the current “view” of a computer-generated world, and have that world react quickly to the user’s actions. To be convincing, the presentation must provide enough detail to make objects easily recognizable, and enough objects to give the user the sense of being in the world. To be useful, the environment must respond to the user, perhaps by changing a parameter of a visualization or a dimension of an object’s design, or by having objects respond as the real objects that they represent would respond.

An environment can be described as a geometry, which describes the spatial relationship of objects to the user or users, and the behavior of these objects. Objects may have a visual appearance, may make a sound, and may have an odor, taste or texture to be felt. Output devices for the far senses, sight, sound, and odor, have been easier to develop. Almost all VE applications provide a visual display, and many provide audio, although output for all of these senses has been shown to be effective (Dinh, 1999).

Given that a three-dimensional environment may occupy a large “space”, geometries that include many objects that constantly react to other objects and the user are need to give the full illusion that the user is in a realistic world. However, current technology is not capable of handling the amount of information and processing that this requires. In addition, creating all of the pieces of the world at such a level of detail would be a huge project. Luckily, many applications can be created with VE technology which do not require a full illusion. In addition, a somewhat realistic world can be quite convincing. It is still a challenge, though, to create and process the description of an environment, the environment’s model.

An environment model has at least three representations. One representation is how the model is described in a stored file, which may be designed to be readable and understandable by model builders. Another, similar,
representation is one that is accessed and modified by a running applications to effect complex behaviors that could not be described in the model itself. A final model is the internal representation that is optimized to constantly provide the information necessary to the output devices to render a visual model, provide 3D sound, and so on. At some point, the a model is translated from one representation to another. For example, a file format that provides a stored model is the Virtual Reality Modeling Language (VRML) (Ames, 1997). The Java3D package (Sowizral, 1997) provides a run-time model for Java programs. Parts of the Java3D run-time model can be explicitly “compiled” to produce an optimized internal model, at the cost of making modification of the environment model more difficult.

Some file representations are designed to translate easily to the internal representation, but do not easily translate to model that can be modified at run-time. Other file representations may translate easily to a model that is easy to modify at run-time, but require more processing to obtain an internal model that is optimized for interactive use. In the long run, a translation step from a file to an internal representation is cheap, as it occurs only once. On the other hand, if the run-time model is modified often, the translation from the run-time model to the internal model can become a critical performance bottleneck.

In this chapter, we will describe how an environment model is generally described in terms of geometric primitives and behaviors. This is often the model provided to a running application that can be modified appropriately. We then relate this representation to common file formats used to describe a virtual environment, and to an internal representation that provides optimizations for efficient presentation of the model. Finally, we describe how the user can be incorporated into the model to provide a clean interface between the user and the environment.

2. Geometry

For the most part, virtual environments are spatial in nature. An environment consists of properties associated with locations in three-space, such as sounds and appearance. A VE application, therefore, maintains a representation of spatial information, and attempts to organize that information so that arbitrary views can be
produced quickly and repetitively. Since visual perception is the primary channel used by sighted people to obtain information about the world around them, VE systems have focused on representing geometric models of visual objects, and their placement in the environment.

A virtual environment is usually described by low level primitives such as polygons, lines, and text, and by higher level primitives such as spheres, boxes, cones, polygonal meshes, polyhedra, curves, and curved surfaces. Current graphics hardware renders low level primitives, as well as polygonal meshes, very efficiently. However, many of the higher level primitives, such as sphere, curves, and curved surfaces, cannot be precisely rendered fast enough, and they are, instead, converted to a polygonal representation when they are added to the environment.

Although the visual appearance of an environment can be completely described by a set of polygons, lines, and text images, it is more useful to organize the geometry into separate groups that are related to each other. This network of geometries and relationships is known as a scene graph, and may contain more information than simply geometric primitives that are associated with nodes in the graph. In this section, we will describe the geometric primitives used in VE applications, how these primitives are organized into a scene graph, what relationships the scene graph provides between groups of primitives, and what additional information may be stored in the scene graph that affects the appearance of the environment it describes, such as materials and light sources.

2.1. Geometric primitives

Current computer graphics systems are optimized to display convex polygons and lines, and are capable of rendering millions of polygons per second to the screen. Therefore, geometries are decomposed into, or approximated by a set of polygons or lines. Text is often treated as a special primitive, although it can be crudely represented by a set of lines, a set of polygons, or more nicely represented by an image on a polygon, where parts of the polygon are made invisible by the image.

Geometric primitives are described by point coordinates in a three-dimensional coordinate system. By convention, the coordinate system is right-handed (when curling the fingers of the right hand from one positive axis to another, the out-stretched thumb points in the direction of the third positive axis). Whenever a group of primitives
describe an object that has a particular orientation (for example, a tree generally grows “upward”), convention assigns the Y coordinate axis to represent the “up” direction, the X coordinate axis to represent the “right” direction, and the Z axis to represent the “back” direction. This coordinate system is a remnant of models developed for display on a computer monitor, where the X axis is the horizontal axis of the monitor, and the Y axis is the vertical axis of the monitor. (If only monitors had been placed face up on or in a desk. Then the coordinate system convention might match the Z-up convention of mathematicians and physicists!)

Polygons, also known as faces, are described by a list of co-planar points in three space. Each pair of points on the list (and the pair containing the first and last points) defines an edge of the polygon. Graphics systems, however, are only designed to render convex polygons, and may just render triangles. Concave polygons or polygons with 4 or more edges are usually decomposed into triangles before they are rendered (often, a graphics system needs to told to spend the time to convert concave polygons correctly). Polygons are shown filled with some color or pattern, but are considered infinitely thin, which means that if they are view from the side, they will be invisible. Lines, however, are of a given width regardless of the viewing angle.

In addition to allowing for polygons and lines, modeling systems will generally allow users to create higher level geometries, such as boxes, spheres, cones, and cylinders. These shapes may be converted to polygons by the modeling system, or may be stored as their shapes, and must be converted to a polygonal representation by the rendering system. Maintaining the shape information provides the right level of accuracy for the model, where the quality of the representation can be adjusted based on its visibility, but provides an added burden to the rendering system. This trade off is also true for curves or curved surfaces. It should be noted, however, that current modeling packages are geared towards providing visually pleasing models, and therefore produce many more polygons than can be handled by an interactive rendering system. What’s worse, some of these packages do not allow the user to control the polygonalization algorithm to give a higher weight to less polygons than to higher accuracy.

An environment generally consists of a set of geometric objects, where each object is described by a set of solids or lines, and each solid is described by a set of surfaces, which, in turn, are described by a set of edge-connected polygons (see figure 1). The view point is assumed to always be outside of a solid. By making this
assumption, the rendering system can draw only polygons on surfaces that face the viewer, since surfaces that don’t face the viewer are on the other side of a solid, and are obscured by the surface of the solid facing the viewer. This technique is known as back-face culling. However, when the assumption fails (the view point moves inside of a solid), then the effect of back-face culling is that the surfaces disappear altogether (all surfaces are facing away from the view point). A rendering system may display an “inside” color instead, but this is expensive and rarely done.

![Figure 1: Polygonal surface](image)

In order for the back-face culling technique to work, polygons of a surface must have a “front” and a “back”. By convention, the order of the points that describe the polygon determines the front and back sides of the polygon: if the viewer is on the front side, then the points are given in counter-clockwise order. If the points are given in clockwise order from the point of view of the viewer, the viewer is on the back side of the polygon (as shown in figure 1).

Surfaces can be given as a set of polygons, but there are methods of representing surfaces that require less information, and therefore can be loaded into the application faster and transmitted faster from the application to the graphics rendering system. One such method represents a surface that is defined by a set of height values along a regular grid. An example of such a surface is terrain that contains no caves or out-croppings (all bumps go up, all holes go down). Not only does this method allow a surface to be described by the coordinates of the corners, the distance between grid points, and a height value for each grid point (roughly 1/3 the information needed if only points were used), but there are algorithms that can quickly approximate these surfaces with fewer polygons when the view is far away (Lindstrom, 1996).
A polygon mesh also describes a surface using less information than the same surface given as a set of distinct polygons. Graphics systems generally support meshes of triangles or quadrangles. A mesh is described by a list of polygons, where each successive polygon shares one edge completely with the previous polygon, and therefore shares two points. For example, a triangle mesh can be described by three points for the first triangle, then one more point for the next triangle, another point for the third triangle, and so one. A surface described in this manner uses roughly 1/3 fewer points to describe a surface that does not have the restrictions of the elevation map. In fact, an elevation map will likely be translated to a triangle mesh before it is rendered.

When a polyhedron is specified by a set of face polygons, each being described by a set of coordinates in three-space, many of the same coordinates are given more than once. This is a result of the common edges of polygons that, together, describe a surface. To avoid storing redundant information, the geometric model is very often given as a set of labelled coordinates, and a set of faces that list the labels of the coordinates that describe the face’s edges. This storage scheme is demonstrated in figure 2, which give the coordinates and faces of a 2x2x2 cube centered at the origin. Note that the order of the coordinates for each face are given in counter-clockwise order from the point of view of being outside of the cube. A similar strategy can be used to store coordinates of a list of line segments.

![Figure 2: Face and coordinate lists for a cube](image)

Graphics hardware vendors are quick to describe the performance of their display system in terms of how many polygons per second it can draw. However, this measurement does not tell the full story. Depending on the hardware architecture, a graphics system may be polygon-bound or fill-bound. If a system is fill-bound, then the
number of polygons that can be displayed to the screen is heavily dependent on how much screen space each polygon takes when rendered. In other words, the time taken to fill each polygon dominates the rendering time. If a system is polygon-bound, then the size of the polygon has a small effect on rendering time compared to changes to the number of polygons. This difference in architectures has a profound effect on how models should be designed and built for VE applications. If the VE application will use a polygon-bound system, then the model should contain as few polygons as possible, no matter how large they are individually. Surfaces that are mostly flat could be replaced by single polygons, for example. If the system used is fill-bound, then it is often a good idea to break up a large polygon (or, more precisely, one that will appear prominently on the display) into a set of smaller, co-planar polygons.

2.2. The scene graph

One could represent the geometric model of a virtual environment with just a set of polygons, lines, and text. However, when a particular object, such as a chair, table, or vase, moved in the environment, then all of the coordinates of the polygons that represent the moving object would need to be changed, or transformed, in the same way. In addition, if the vase was conceptually “on top of” the table, the coordinates of the vase should move when the table moves. A more efficient representation of the model would group geometric primitives together into geometric objects, and would define “attachment” relationships between the objects.

Environment models, therefore, are commonly represented by a coordinate system graph, which is a directed acyclic graph. Each node of a coordinate system graph contains a set of geometric primitives, which usually represent a conceptual object or object part. The geometry of a node are described in terms of a local coordinate system. The edges of the graph define a transformation between the coordinate system of one node to the coordinate system of another node. Conceptually, the coordinate system of one node (the child) is contained in the coordinate system of another (the parent). If the parent’s coordinate system is transformed, the child’s coordinate system is transformed, as well (unbeknownst to the child), but if the child’s coordinate system is transformed, the parent’s coordinate system is unaffected, and the child’s transformation is visible in the parent’s coordinate system.
A coordinate system graph does not contain cycles because it would not be clear how a change in the transformation of an edge in the cycle should change the rest of the edges in the cycle. Should all of the transformations be changed equally, or should just one other edge change? If just one edge changes, which one? For example, if five marbles represented in a coordinate system graph were placed next to each other, and the graph contained edges between adjacent marbles to keep them spaced evenly, and an edge between the outer marbles to maintain a certain length for the marble line, how should the marbles move if the distance between the outer marbles changes? Or if the distance between two adjacent marbles changes? Should the marbles adjust to space out evenly again, or should just one other distance change?

Coordinate system transformations are described by a 4x4 matrix that operates on homogeneous coordinates, where a location in space is given by the 4-tuple (x, y, z, 1). A set of locations are transformed for one coordinate system to another by multiplying each location by the matrix transformation between the coordinate systems. Matrix transformations can be combined by multiplying them together, but matrix multiplications is not commutative, and a change in the order in which the transformations are given will change the total transformation (unless the transformations are all the identity matrix, which does nothing).

A matrix transformation in a coordinate system graph is often described as combination of rotations about the three major axis (X, Y, Z), changes in scale along the major axis, and a translation along the major axis. However, the transformation may include shears, mirrors, projections, or other affine transformations. Standard 3D matrix transformations and the use of these transformations in a coordinate system graph are described in more detail by Foley et al. (1996). Some systems or file formats (such as VRML) may describe the rotation component as a certain rotation around a given vector. This representation, known as a quaternion, describes a rotation more efficiently, and is better for interpolating between two rotations (smoothly changing from one rotation to another without experiencing gimbal lock). However, an extra step is required to incorporate a rotation in this form into a 4x4 matrix transformation (Watt 1992).

A coordinate system graph is usually represented as a tree, where the root node contains geometry in a “world” coordinate system, and the child nodes have local coordinate systems that related to the world coordinate system.
Therefore, points in a child node geometry can be transformed to the world coordinate system. Points further down a sub-tree of the root node can be transformed to their parent’s coordinate system, and then to its grand parent coordinate system, and so on until it is in the world coordinate system. Similarly, a point in the world coordinate system can be transformed to the local coordinate system of a descendant node by applying the inverse of each transformation on the path to that node. When rendering a scene for the display, geometry must be transformed to the world coordinate system, then transformed to a descendant node representing the coordinate system of the viewer, where the system’s origin is at the eye point.

2.3. Material properties

Surfaces in the real world have many properties that affect how they appear beyond their position and orientation. They may be rough or smooth. They may be dyed or painted with different colors, or may contain grains of material variation. They may have fine detail that are too expensive to model geometrically, such as the texture of skin or hair. At a great distance, even features like brick grooves, window sills, or door handles might be considered fine detail that is too expensive to model. These features can be generally described as the contribution each point on a face or line makes to the color of a pixel on the display, if the point is visible at all. In addition to the various properties of the face, this contribution will be affected by the light sources and global lighting model that is in effect for the environment.

The appearance of a face under a given lighting configuration is generally described by properties of the face as a whole and properties of the points on the face. Properties given to the entire face include the color of the face under no light, the color response to a diffuse light, the color response to a specular reflection, and the transparency of the face. Properties of individual points on a face can be given by an image map, which contains color and transparency information for points on a two-dimensional grid. The color information of the image map may be combined with the face properties, or may over-ride them. When all is said and done, the material properties define a contribution of certain values of red, green, and blue to a pixel on the display, if the face is visible at that point on the display.
2.4. Lighting and shading

The appearance of a geometric primitive depends not only on its material properties, but also how light sources in the environment are positioned. An environment is assumed to have a certain amount of low-level ambient light, and the material definition given to a face describes the color to use for the interior of the face in the absence of any light sources. For each light source that does exist, a proportional amount of diffuse color, from the material definition, is added to particular pixel colors for parts of the face that are visible. The amount of diffuse color is dependent on the incident angle of the ray from the light source onto the face. Light sources can be given at infinity, where all light rays are parallel and travel in a certain direction, or can be given a position from which the rays emanate. Systems that render the display at interactive rates generally do not consider obstructions to light rays, and therefore, no shadows are shown. The left two screen-shots in figure 3 demonstrate the effect of a light source (shown in the top right corner) on a surface consisting of 6 triangles (the bottom image is shaded the same way, but has lines that show the triangles).

![Figure 3: Shading from light source and Gouraud shading](image)

In the images on the left of figure 3, each triangle was considered individually. If a group of faces are considered part of an approximation of single, curved surface, as in the images on the right of the figure, an additional technique can be used to shade the faces so that they appear curved rather than faceted. This technique requires that the normal to the curved surface at the vertex point of the face be given. For example, the normal at the center of the face group in the figure points directly out, rather than in the direction of the normal to any of the
individual faces. Using these vertex normals, the graphics system can determine the color of each point of the interior to the face by calculating the normal of the curved surface at that point and using that in the lighting calculation, rather than the normal of the face. In Gouraud shading, the most common shading algorithm using vertex normals, the normal at an interior point is a linear combination of the vertex normals. The two images on the right of the figure are rendered using Gouraud shading. Foley et al. (1996) describe this and other shading algorithms in more detail.

3. Behavior

A dynamic environment can be represented by a set of objects that have particular behaviors. These behaviors may be given as part of the stored model of the object, or may be defined by the execution of the application code, or some combination of the two. An object may have a set of behaviors that it follows depending on the context it finds itself in. These behaviors can be roughly categorized as environment-independent behaviors, which does not consider the current state of the other objects in the environment, and environment-dependent behaviors, that do consider other objects. Environment-independent behaviors include changes to the object that are solely based on the passing of time, or time-based behaviors, and changes that occur in a particular sequence, or scripted behavior. Environment-dependent behaviors include event driven behaviors, which respond to events initiated by users or other objects, and constraint maintenance behaviors, which react to changes of other objects to maintain defined constraints, such as attachment, gravity, or penetration limits.

- Environment-independent behavior
  - Time-based
  - Scripted
- Environment-dependent behavior
  - Event driven
  - Constraint maintenance

3.1. Time-based

An object’s behavior may describe a change in one of the object’s properties over time or at a certain point in time. Object properties that may be changed include the object’s position, orientation, color, and visibility. A
circling airplane may following a particular path as it flies around a town, where its position and orientation is determined by the amount of time that has passed from when it began flying. A distant mountain may change from a detailed model displayed during daylight hours, to a simpler model for dawn and dusk, to being completely invisible at night.

Time-based behaviors may be defined to use wall-clock time, or absolute time, which is the time frame of the user, or a time frame that is transformed from wall-clock time. Behaviors usually have a start and end time. If the behavior changes a property continually, then it defines how the property changes over the time between the start and end times. For convenience, the time values used to define such a change usually are 0 at the start time, and 1 at the end time. If a behavior should last, say, five minutes, then the behavior can be scaled appropriately.

Time frames can be used like one-dimensional coordinate system frames. A time frame can define a translation (time 0 to a start time) and a scale (time 1 to an end time). Every time frame can be related to the wall-clock time, or time frames can be organized hierarchically, like a coordinate system graph. For example, in the Mirage system (Tarlton, 1992), an activity consists of the change of an object’s property from a \( t=0 \) to a \( t=1 \) value. A single activity can be set to begin and end at certain times, or can be combined with other activities by becoming siblings of a parent activity. The relationship between a parent activity and its children may specify that the child activities occur simultaneously, or one after the other. The parent activity will have a relationship with wall-clock time that will have it occur at a certain time and pace. With this structure, an activity could flow at various speeds, and could even be paused or reversed!

For example, figure 4 shows an activity hierarchy where a “Rolling Wheel” activity consists of a “Rotating Wheel” activity and a “Moving Wheel” activity. These child activities may be combined for a coordinated motion by making no transformation from the parent time frame to the time frame of each child, as is show on the diagram on the right. Alternatively, the motions can be performed one after the other by transforming the 0 to 0.5 time values of the Rolling Wheel activity to the 0 to 1 time values of the Rotating Wheel activity, and transforming the 0.5 time values of the Rolling Wheel activity to the 0 to 1 time values of the Moving Wheel activity, as is show on
the diagram to the right. These two structures produce a rolling wheel and a wheel that rotates then slides, respectively.

Figure 4: Activity combinations

Individual activities may be specify an instantaneous change in an object’s property at the activity’s start time, or may specify a continuous change of a property based on the time value. The continuous change may be given as a function of the time value, or may simply be specified as an interpolation between a starting and ending property value. Different interpolation methods may be used depending on the property type. For example, a linear interpolation is often used for position changes, while a color change may be a linear interpolation of colors that have been transformed from RGB (red, green, blue) space to HSV (hue, saturation, value) space. A common
animation technique is to define particular property values, such as position and orientation, for objects at certain moments in time. A set of these values are known as a key frame, which define the start and ending values of interpolation activities that follow one another.

3.2. Scripted

A scripted behavior simply gives a sequence of changes to an objects, where a particular change may be a time-based behavior, but the behavior as a whole does not depend on the current time, or the rate that time passes in the environment. By not being time-based, scripted behaviors are at the mercy of the simulation rate to determine the speed through which a sequence is followed, which is why a time-based behavior is preferred, when it is possible to specify one.

3.3. Event driven

The behavior of an object may be specified as a particular reaction to an event that occurs in the environment. An event may be the result of a user’s action, such as pressing a button, or it may be the result of a singular occurrence in the environment, such as the collision or intersection of two objects, or it may be an incremental change to the property of another object, which may represent the application of a continuous change over the time between the previous and current simulation frames.

Events can be stored in a global event queue (which may actually be many queues), and dispatched to objects that have expressed an interested in them, or can be stored locally in an object’s out field, and routed to the in field of other objects when such a route has been set up. Generally, systems that are designed to support event response in application code use the event queue model, while systems that use only behaviors defined in a file description, such as VRML (Ames, 1997), will use event routes. However, the Java3D package (Sowizral, 1997) uses an event route network, which eases the incorporation of VRML models into its run-time system. The event route model is not as flexible as the event queue model, as a route must have a single source, while an event can come from
anywhere. It is not difficult for an object to listen for events from an event queue that come from one objects, but is it difficult for an object to set up routes from many objects.

3.4. Constraint maintenance

Depending on its complexity, a run-time system for VE applications may provide automatic behaviors for objects based on particular properties of the objects. These automatic behaviors can be classified as geometric constraint maintenance and physics-based motion.

A geometric constraint uses properties such as attachment and spatial relationships with other objects. Through the use of a coordinate system graph as part of a scene graph, a run-time system automatically maintains a spatial relationship between child and parent objects in the graph. The run-time system may also maintain a direct spatial relationship between two objects that are not a child or parent of the other in the scene graph. For example, objects that represent 3D interface components (buttons, menus, etc.) may be automatically arranged so that they are adjacent to each other, so that they do not overlap even if components are moved or scaled. In another example, the WALKEDIT system (Bukowski, 1995), which allows users to quickly position objects to construct rooms with furniture, books, and other items, enforces spatial constraints that come from abstract relationships, or associations. Picture frames are constrained to vertical surfaces, cups sit on horizontal surfaces such as desks or floors, and so on.

A run-time system can reasonably maintain a list of one-way constraints, where a change in one property results in a change in another property, but not vice-versa. For example, a change in the position of a parent node in the scene graph affects the position of the child node in the environment, but a change in the child node’s position does not affect the parent node’s position. It is much more difficult to support two-way constraints, or constraint dependences that contain cycles, as a solution must be found by solving a system of simultaneous equations, and it may be over or under-constrained (a solution does not exist, or many possible solutions exist).

A virtual environment can look more “realistic” if the objects in the environment follow the physical laws we are used to in the real world. Real world objects have mass, may be rigid or flexible, solid or fluid, will fall at a
constant acceleration when not supported, will move and twist when pushed by some force, and so on. However, simulating a world with many objects that obey such laws at interactive rates is a job beyond the ability of most computers in existence today. Therefore, current run-time systems generally do not try to enforce physical laws, but a few provide assistance to applications they wish to do so for a small subset of the objects in the environment. Objects may be given properties such as mass, velocity, angular velocity, acceleration, and angular momentum. In addition, some run-time systems can report when collisions occur between the geometries of two object, or a support library, such as I-COLLIDE (Cohen, 1995) can be used. It is almost always us to the application, however, to determine how the objects might change form or trajectory due to the collision.

4. File Formats

In the previous sections, we have defined the vocabulary with which virtual environments are described. The geometry and behavior of a VE application may given procedurally, using this vocabulary as a basis. A VE application may contain code that constructs geometry using a low level graphics library, such as OpenGL, or a high level library such as Java3D. The application may also contain a step-by-step procedure for how geometry changes over time, or as a reaction to the user’s actions.

A geometric model can generally be described more efficiently in a file that provides information about the final result, such as vertices, colors, and so on, rather than giving a procedure for building the geometry. A file format can be designed that is designed to describe geometry and spatial relationships between geometries, and therefore allow for clear and concise descriptions of a geometric model. It become the task of a file loader and the run-time system to translate the model from the stored representation to the run-time model, and ultimately to the internal model for efficient rendering.

A file format might also allow simple, object-based behaviors to be clearly and concisely defined. However, complex behaviors and behaviors involving many objects are still best left to a programming language to describe the behavior procedurally. For example, the Alice system (Pausch, 1995) uses geometries that are described by model files, but object and environment behaviors are described using Python scripts.
Files that describe geometric models may be intended to be read and edited directly, such as those defined by the Virtual Reality Modeling Language (VRML) (Ames, 1997), or be intended to be created and modified by modeling applications, such as those defined by the Multigen OpenFlight format (MultiGen, Inc. 1998). A format that will be read and edited directly must define an organization of geometric primitives and components that match the designer’s conceptual model of the environment’s construction. In addition, it should use descriptive key words and allow components to be named. A model format that is not intended to be read directly can have a more compact form, although it may still be readable. For example, the Wavefront obj format defines a single geometric object using vertices, vertex normals, texture coordinates, faces, and face materials, each of which are defined using a short keyword (“v” for a vertex, “vn” for a vertex normal, etc.). A Wavefront material file supplements an obj file to provide the definition of face materials.

5. Internal Model

The internal model representation of the environment is designed to provide the rendering performance required for VE applications. The run-time representation of a geometric model could be passed directly to the low-level graphics library and audio libraries to present to the user, but the run-time representation is not likely to be the most efficient representation to use, especially for visual components. An internal representation, therefore, can be created that provides a more efficient transfer to the graphics hardware, and therefore allow for faster performance of the system.

The goal of almost all rendering optimizations is to reduce the amount of information passed from a software application to the rendering hardware. These optimizations can be divided into automatic and application-assisted optimizations. Automatic optimizations translate a run-time or stored model into a form that describes a geometry using less information, and computes extra information that allows for run-time selection of parts of the geometry that can be ignored. Automatic optimizations include primitive sorting for reduced state changes, display list generation, and culling faces for the render list based on the user’s viewpoint. Application-assisted optimizations include switching between different levels of detail and using billboarded geometries. These optimizations are
described in this section. Many of these are implemented by the IRIS Performer toolkit, described by Rohlf et al. (1994).

5.1. Primitive sorting

A graphics rendering system uses a current graphics state to determine how primitives will appear when they are rendered. The current graphics state includes the coordinate system, material properties, a texture image, and global properties such as fog and light sources. When describing the primitives of a model to be rendered, an application could specify all properties of the graphics state required to render the primitive correctly. However, more often that not, a primitive is part of a greater whole, and shares most of its properties with many other primitives. For example, a set of triangles that describe the surface of a hood for a car model will share the same material and lighting properties, and will share the same lighting properties with the triangles that describe the hub-cap. Repeating the current state for every polygon on a particular surface would be quite inefficient.

Systems that generate an internal model for rendering efficiency, such as the Performer toolkit, provide a framework which allows primitives to be sorted by their graphics state. Primitives that have the same material and global properties will appear together, while those with similar properties will be close to each other, and so on. This modified list will be used to describe the model to the renderer, and state changes will only be given when necessary. The end result of this optimization is that much of the redundancy of the model is removed.

The most optimal list of primitives would be a sorted list of all of the primitives. However, this list would need to be updated whenever one of an object’s properties changes. For example, since all of the primitives would need to have a common coordinate system, all of the primitives of an object would need to transformed to a common, world coordinate system, taking into account the coordinate system transformations between the object and its parent and ancestors. If the object were to move, however, though the change of one of these transformations, all of the primitives in the list that are part of the object would need to updated with a new position. This would be a book-keeping nightmare. Instead, lists of primitives that are sorted are usually limited to individual objects, as described in the next section. However, Durbin et al. (1995) describe a system that provides a list of primitives for
each object which is either just the primitives of the object, or one that includes the primitives of the object and its children, if the children haven’t changed their state in a while. In this system, when a child changes its state, and its primitives are part of a parent, the parent returns to its individual list until its children don’t change for a while. In this way, object trees that do not change often are rendered more efficiently (perhaps dramatically so, if the objects share many properties).

5.2. Display lists

Another method to reduce the transfer of geometry and properties changes through the graphics system is to, instead, create display lists representing the graphics commands required to create the geometry and to make property changes. A display list can be created and transferred once through the graphics system, then referred to by a unique id each time the commands should be executed by the graphics system. The graphics system may perform the sorting optimization. Display lists can contain the commands to create a geometric object, or may simply consists of the commands to set up a particular material with a texture image to be used for a variety of primitives.

Like a sorted primitive list, a display list is only useful as long as the properties of the primitives do not change. When one property of a single primitive changes, the list needs to be recreated, which be a time-consuming operation. For this reason, the Performer library does not use display lists for most primitives, instead using immediate mode, and optimizing as much as possible the transfer of graphics commands to the rendering system. For systems that use display lists and sorting, the command and primitive list can be re-generated whenever a property changes that makes that necessary, or can be generated under program control, for example by using the compile method for geometry objects in Java3D. The system, therefore, must catch changes to properties and perform a re-generation, or simply prevent the change unless the object is set to no longer be compiled.
5.3. **Face culling**

Two common methods to reducing the amount of information transferred to the rendering system involve omitting faces that cannot be seen by the user. One method, *back-face culling*, makes the assumption that face primitives define the surface of a solid object, and that the face will only be viewed from outside of the object. Therefore, the face is only rendered when the user’s view point is on the *front* side of the primitive, which is defined to be the side of the face for which the list of vertices is given in a counter-clockwise manor. If the viewer is on the back side of the face, it is not rendered. If the object is one that the viewer can be inside of, such as a room, the faces must be made to be two-sided, usually by having two faces that use the same vertices, but give the vertices in the opposite order. The faces may be offset to reflect the thickness of the boundary between inside and outside of the object (the thickness of the wall, for example). This technique is usually a feature of the low-level graphics library.

The other method to reduce the amount of faces rendered involves the higher-level representation of the objects in the environment. Objects are checked for an intersection or containment in the user’s view volume frustum, which defines the volume of the environment that the user would see if there were not obstructing objects. This volume is defined by the position of the user’s view point and viewing window in the environment, and the near and far clipping planes. If an object is found to be outside of the viewing frustum, it is simply not rendered at all, saving the rendering system the effort of determining that each face of the object falls outside of the rendered display.

This method depends on a simple geometric representation of each object that is a good approximation of the volume the object occupies. Common representations include spheres and boxes. It is easy to check for intersections between spheres and other spheres or plane-based volumes, but they are often contain much more empty space than space occupied by the object. To be easy to manipulate, spheres must be given in a single coordinate system, as non-uniform scale and shear transformations will change the spheres into non-aligned ellipsoids, which are much more difficult to work with. Box volumes usually have less empty space. A box volume
can all be aligned to a single world axis for easier intersection testing, or aligned with the local coordinate system of the object it represents, for a more accurate approximation.

Since a scene graph defines a hierarchy of objects, this hierarchy can be exploited for more effective view volume culling. The parent of group of child objects might have a bounding volume that contains the geometry of the parent and the geometries of the group. Therefore, if the parent object’s bounding volume is outside of the viewing volume, so is all of its children, and that whole sub-tree can be omitted from the rendering of the current frame. If the environment is sparsely populated by objects, then a bounding volume hierarchy that may be more effective can be constructed separately from the scene graph based on grouping the closest objects, then the closest groups together, until one group contains all objects.

5.4. Level of detail switching

The amount of detail necessary for an object’s description depends on how much detail the user can possibly notice. If the object is far away, or in a foggy or low-light environment, the user will not notice much detail, while if the object is near by on a clear day, most detail will be perceived. A method to reducing the amount of information transferred to the rendering system involves selecting a object geometry from a list of representations, each of a different levels of detail, which has the least detail but will appear very similar to the object at its full detail. That object is used, potentially saving the wasted effort of rendering detail geometry that would not be noticed. This method of omitting detail can work for all presentation mediums involving the distance senses. Sights, sounds, and smells diminish in intensity and detail based on distance or other factors. In general, the application must supply the different levels of detail for an object, and give some measure of how it compares to a full detail model. A common approach, as seen in the LOD node of the VRML file format, is to associate a representation with a range of distances from the user’s eye point for which that representation should be used. A system, such as Performer, can also automatically determine when an object’s geometry is so small that it does not occupy enough of the screen to be noticed, and therefore can be omitted completely.
5.5. **Billboarded geometry**

A common technique that is used to reduced the amount of detail needed for a geometric model is to use an image containing a picture of the detail. This technique works well for distant objects, as the absence of depth to parts of the model is not noticed. However, this technique alone will not work for objects that can be viewed from all directions, such as trees or clouds. Another technique can be used, in addition to an image, to give the appearance of depth to a model, if the model can be assumed to appear the same from all directions. This technique is known as billboarding. The image is placed on a flat polygon, but the polygon automatically always faces the viewer, or always orients to the viewer while keeping one axis aligned (such as the “up” axis for standing objects). This technique can be combined with changing the image based on the rotated angle to give an approximation of an object that is not symmetric around one or more axes. These techniques require that the model be defined to take advantage of the feature of the run-time system that maintains a simple orientation constraint of an object with the viewing position.

6. **Including the User**

The main difference between 3D virtual environment applications and other desktop computer applications is that the user has a presence in the environment. The user has a position and orientation within a 3D geometric model, and views the environment from that position, and may receive sounds and smells, based on that position in the environment. The user’s view can be displayed on a desktop monitor or workbench (which may be vertical or horizontal), on a hand-held display, or on a head-mounted display (HMD). The user’s position and orientation may be controlled by keyboard or mouse input, or from input from six degree of freedom tracking devices (that give a position and orientation of a “tracker” in reference to a “transmitter” device placed in the user’s workspace). Tracking devices may also be used to determine the relative position of the user’s hands, feet, torso, etc., as needed by the application.

The following sections describe how the user can be included in the environment model, how tracking device information can be incorporated into the user model, and finally, how the user model can be used to generate the
parameters for generating the user’s view on whatever display device that is being used. Examples of the
techniques presented come from the Simple Virtual Environment (SVE) library (Kessler), and other systems.

6.1. User Model

The user’s position and orientation in the environment could be given simply in terms of the coordinate system of the root of the environment’s coordinate system graph, or the world coordinate system. Using this method, however, it would be difficult to have the user follow the path of objects in the environment, such as cars or airplanes, or be in objects such as rooms. VE systems, therefore, usually provide a special node in the scene graph that represents the user. This node can be attached as a child to any other node in the graph, and therefore follow that node as it moves. For example, the Java3D system (Sowizral, 1997) provides a ViewPlatform object node that can be placed anywhere in the scene graph.

However, using one special node to represent the user makes it difficult to have other body parts in the scene besides the user’s eye, such as hands and feet. An application may want to have a geometric representation of these body parts to provide ways for the user to interact with objects in the environment. In the environment model of the SVE run-time system, the user is represented by a tree of objects that may or may not have a geometric representation. The root of the tree is a “user” object, which serves as the connection point for child objects such as “head” and “hand”. The “head” object has “eye” child objects, and “hand” objects may have child objects representing fingers. Other objects can be added to this user model as needed. These objects can be controlled by keyboard or mouse input, or may be controlled by tracking devices, as described in the next section. Regardless of how they are controlled, though, the interaction of the user with the environment can be defined in terms of the movements of the objects in the user model. In addition, this model allows for a variety of display configurations, as described in section 6.3.
6.2. **Incorporating Tracking Information**

Tracking devices generally report the position and orientation of a “tracker” in relationship to a particular reference frame. If the tracker is the end of a set of mechanical linkages, then the reference frame is the base of the linkages. If the tracker is an electromagnetic device, then the reference frame is the “transmitter” part of the device (often placed on the desk or hung above the user).

Information from the tracking device can be used in raw form, or adjusted by a given offset to describe where the reference frame would appear in the environment if it had a geometric representation. This information can be used to determine the position of the viewer, or be used to determine how the user interacts with the scene. However, this method requires keeping the reference frames of the tracking devices configured correctly as the environment changes.

In the SVE library, tracking information is introduced into the user model to control a body part (or any other object in the environment) by adding two objects. One object is used to represent the reference frame itself, and it is set to be the child of the controlled object’s parent. The other new object represents the tracker, and is placed as a child of the reference frame object. The controlled object is then set to be the child of the tracker object. Its coordinate system transformation is set to the raw tracking information each frame. By initializing the coordinate system transform of the reference frame object correctly, the controlled object will move around its previous parent correctly. If a correction needs to be made for how the tracker is attached to the body part it is tracking (it is 90 degrees off, perhaps), then a correction can be place in the transformation from the tracker to the controlled object.

6.3. **Viewing Model**

For each frame generated by the VE application for the user’s display, the run-time system determines the user’s eye point and viewing direction, and then uses viewing parameters specific to the display to render the frame. A common approach to specifying the viewing parameters is to use a camera model, as in Java3D, which is described in detail by Robinett (1995). The camera model assumes that the viewer is looking towards the center of
the display, and uses a given field of view and aspect ratio to describe the horizontal and vertical angle made by planes passing through the display window’s edges and the eye point.

The camera model works well for head-mounted displays, and desktop displays that do not consider movement of the eye. However, this representation does not support the “fish-tank” display configuration (Deering, 1992), where the window to the virtual world remains stationary within the user’s workspace as it represents a computer monitor, projection screen, or desktop display; or hand-held displays (Rekimoto, 1997), where the window to the virtual world is rendered to a hand-held display that moves with the user’s hand motions.

The SVE system supports all of these display configurations by using its user model and by introducing an object that represents the view plane. A view in the SVE system is defined by an object representing the eye point (or two objects representing the different eyes for stereo images), and object representing the view plane, the width and height of the window in the view plane’s coordinate system, and the root of the environment model being viewed. The relationship between the view plane and the user model is depicted in figure 5.

The *View plane* object is the representation of the projection plane for the default view of the environment. Other display configuration parameters give the extents of the window on the plane. For HMD configurations, the view plane object is positioned to correspond to the optical projection plane of the HMD (with window extent values that provide the correct vertical and horizontal field of view angles for the HMD), and is attached to the *Head* object. For fish-tank displays, the view plane represents the actual monitor screen in the real world configuration. Therefore, the view plane does not move with the user’s head, but stays stationary in the user’s reference frame. Thus, the view plane object is attached to the *User* or *Workspace* object. For fish-tank displays, it is important that the view plane be positioned correctly so that the position difference, or disparity, between left and right views of an object are correct (objects at the view plane should have no disparity, while objects further in front of or behind the view plane should have larger disparities).
Figure 5: Viewing frustum for various display configurations
7. Summary

Virtual environment applications present 3D environments to a user that is located in the environment. In this chapter, we have described how the environment is defined in terms of three representations. One representation defines the model in a file or set of files. The format of these files are designed to provide a concise, geometrically organized description of geometric primitives and simple behaviors. Another representation is used by the application at run-time to modify the environment, effect behavior of the objects in the environment, and to make the environment react to user interactions. The final environment representation is an internal representation that, when possible, distills the description of the environment to the information that is strictly necessary, and provides that information to a rendering system to produce the image for the display. Many of the techniques described in this chapter are given in terms of a visual result, but they could also be adapted to the presentation for other senses, such as sound and smells.

8. References


