Virtual Environments with Four or More Spatial Dimensions

Abstract

We describe methods for displaying complex, texture-mapped environments with four or more spatial dimensions that allow for real-time interaction. At any one moment in time, a three-dimensional cross section of the high-dimensional environment is rendered using techniques that have been implemented in OpenGL. The position and orientation of the user within the environment determine the 3-D cross section. A variety of interfaces can be used to control position and orientation in 4-D, including a mouse "freelook" interface for use with a computer monitor display, and an interface that uses a head-tracking system with three degrees of freedom and PINCH gloves in combination with a head-mounted display. The methods avoid the use of projections that require depth buffering in greater than three dimensions and can be used in conjunction with either 2-D or 3-D texture mapping. A computer graphic engine that displays 4-D virtual environments interactively uses these methods, as does a level editor and modeling program that can be used to create 4-D environments.

1 Introduction

The visualization of objects in high-dimensional space has long been the province of mathematicians and computer graphics specialists eager to visualize high-dimensional manifolds (Noll, 1967; Steiner & Burton, 1987; Banchoff, 1990; Feiner & Beshers, 1990; Hanson & Heng, 1992; Hanson, Munzner, & Francis, 1994). Most of this work has focused on techniques for the high-quality rendering of objects in four dimensions, and wonderful pictures and movies have resulted. These techniques use geometric projections that necessitate depth buffering in 4-D (Hollasch, 1991). The current limitations of computer graphics hardware relegate depth buffering in four or more dimensions to software, and the result is the noninteractive rendering of single pictures. Work that does provide an interactive interface for high-dimensional visualization has either relied on graphics supercomputers or has sacrificed visual quality by using wireframe or other simple models.

In an effort to provide both visual quality and interactivity, our technique for displaying and interacting with environments that have four or more spatial dimensions uses current 3-D computer graphics technology as much as possible. The principle of the technique is a simple one: if the observer of the high-dimensional environment does not know how to engage the interface to the higher dimension(s), then the user experiences the geometry of a normal 3-D environment. The principle is implemented by using observer position and orientation to determine a 3-D cross section of the high-dimensional world. Cross section determination does not require depth buffering, and the 3-D cross section is rendered using standard techniques of 3-D computer graphics, including veridical texture mapping.

In this paper, we describe methods for the interactive display of 4-D environments. These methods generalize readily to the display of environments with four or more dimensions. We then describe results with two programs. The first is a computer graphics engine that displays 4-D virtual environments interactively; it resembles a 3-D action game. The second is a level editor and modeling program that is used to create 4-D virtual environments. The engine and the level editor run well on a personal computer. Environments created using the techniques de-

Presence, Vol. 9, No. 6, December 2000, 616–631

Michael D'Zmura
Philippe Colantoni
Gregory Seyranian
Department of Cognitive Sciences
University of California, Irvine
Irvine, California USA

scribed herein were first presented by Seyranian and colleagues (1999) in work on human search and navigation in four-dimensional environments.

2 Methods

We describe an observer with a position and a viewing orientation in 4-D space, and objects in that space with boundaries that are determined by polyhedra. The position and orientation of the observer can be used to define a 3-D cross section of the 4-D space, and this cross section can be projected onto a 2-D image plane using standard techniques of computer graphics. The visual quality of rendered environments is improved by texture mapping. Objects in 4-D can be texture-mapped so that their geometric properties are conveyed faithfully and generally in 2-D projection; the implementation uses either 3-D or 2-D texture mapping. With these notions for the display of 4-D environments in hand, the generalization to the case of *n*D environments for values of $n \geq 4$ is then presented. Fundamental notions of 4-D vector spaces are reviewed briefly in Appendix A.

2.1 Methods for the Display of 4-D Environments

2.1.1 Geometric Primitives. The boundaries of 4-D objects may be approximated using polyhedra in much the same way that the boundaries of 3-D objects may be approximated using polygons. Each polyhedral boundary element has vertices that can be represented by 4-D vectors.

We define four polyhedra with four, five, six, and eight vertices, respectively, to be geometric primitives. These polyhedra are depicted in figure 1.

Eight cubes (regular octahedra) can be used to make a hypercube, which is perhaps the best-known 4-D object. To understand its construction, consider cubes of lesser dimension. (See table 1.) A 1-D interval, namely a line segment, is bounded by two 0-D points that specify its length. A 2-D interval (plane area) is bounded by four 1-D intervals, namely its four sides. If one supposes that this plane area is oriented along the X and Y axes, the area can be described by the product $[x_{\min}, x_{\max}] \times [y_{\min}, y_{\max}]$ of the intervals. A 3-D interval (rectangular volume) is bounded by six 2-D intervals, namely the six faces of a rectangular solid. The volume can again be

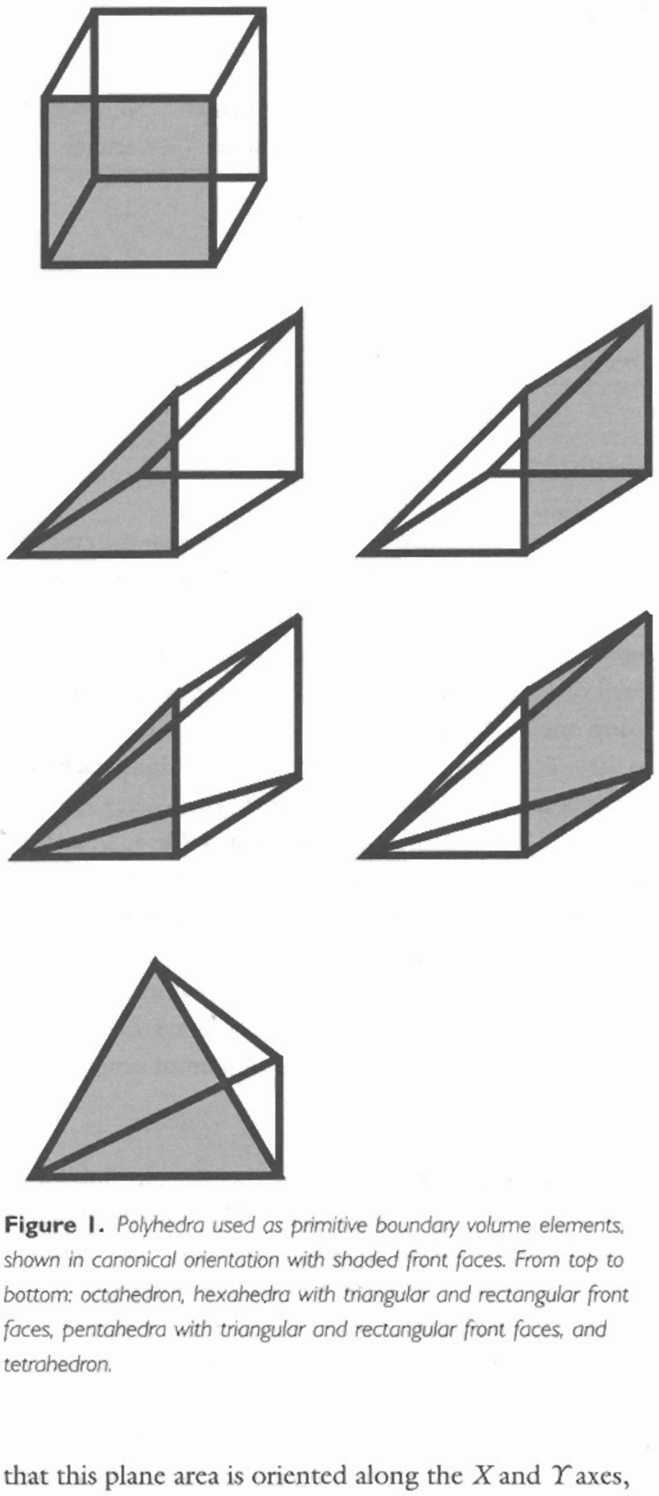


Figure 1. *Polyhedra used as primitive boundary volume elements, shown in canonical orientation with shaded front faces. From top to bottom: octahedron, hexahedra with triangular and rectangular front faces, pentahedra with triangular and rectangular front faces, and tetrahedron.*

Table I. *Cubes of various dimension*

Entity	Dimension	Number of boundary elements	Number of vertices
line segment	1	2	2
square	2	4	4
cube	3	6	8
hypercube	4	8	16
n-cube	*n*	2*n*	2^n

described by the product $[x_{min}, x_{max}] \times [y_{min}, y_{max}] \times [z_{min}, z_{max}]$ of the intervals.

Consider now a hypervolume centered on the origin and oriented along the *X*, *Y*, *Z* and *W* axes. The hypervolume can be described by the four-way product of its intervals and is bounded by eight rectangular solids. Each of these solids is found by holding fixed the value along one axis and allowing values along the other axes to vary. For instance, one finds a volume along each end of the *X* axis: the first is the volume $[y_{min}, y_{max}] \times [z_{min}, z_{max}] \times [w_{min}, w_{max}]$ located at value x_{min}, and the second is the same volume positioned at x_{max}. The other six volumes are found in a similar fashion.

If the sides of the hypervolume are all of equal length, then the hypervolume is a hypercube, and its boundary elements are each cubes. A hypercube is a 4-D box when viewed from outside and is a room of simple shape when viewed from inside.

2.1.2 Pipeline. Figure 2 shows the pipeline for geometric information processing. The boundary elements of the virtual environment are each represented by a list of vertex coordinates expressed in the global coordinate system. This information must be transformed so that the environment can be displayed using 3-D graphics technology. The user interface lets one control viewpoint position and orientation in the high-dimensional environment. Current position and orientation determine a 3-D cross section of the boundary elements to be displayed. The cross section determination is performed only on boundary elements that have not been culled in a viewpoint-dependent fashion. The intermediate result of cross section determination is a set of polygons that must be triangulated. Vertices of the resulting triangles are then transformed from global to local, viewer-centered coordinates for immediate rendering by the 3-D engine.

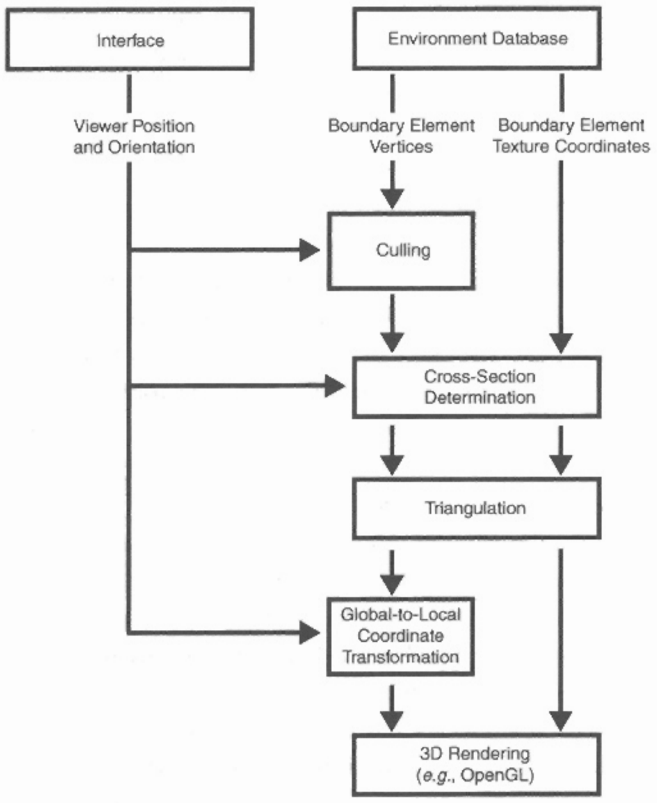

Interface
Environment Database
Viewer Position and Orientation
Boundary Element Vertices
Boundary Element Texture Coordinates
Culling
Cross-Section Determination
Triangulation
Global-to-Local Coordinate Transformation
3D Rendering (e.g., OpenGL)

Figure 2. *Pipeline for processing geometric information. See text for discussion.*

2.1.3 Observer-Dependent Cross Section. We now describe in detail the cross section determination. Observer position is represented by a 4-D vector $\mathbf{p} = [p_x\ p_y\ p_z\ p_w]^T$. Observer orientation is represented by a vector $\mathbf{q}$ of unit length or, equivalently, by a 4×4 rotation matrix $\mathbf{Q}$ that describes the rotation of some canonical orientation vector $\mathbf{q}'$ into the current orientation $\mathbf{q}$. The canonical orientation vector is simply a reference vector and may, for instance, correspond to an initial orientation for the observer such as $\mathbf{q}' = [0\ 0\ {-1}\ 0]^T$.

The computation of the 3-D cross section determined by the observer's position and orientation is best con-

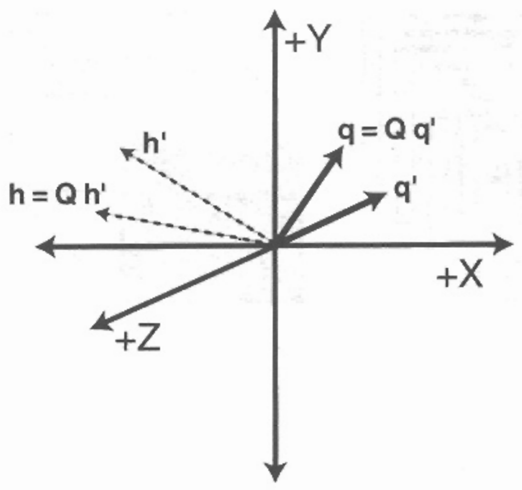


Figure 3. *The hidden axis and observer orientation. See text for discussion.*

ceived in terms of the axis that is perpendicular to the 3-D cross section. This "hidden axis" will change in response to observer motion. It is readily computed using the matrix **Q** that describes the observer's current orientation. In particular, if the coordinates of the unit vector $\mathbf{h}'$ along the hidden axis are known when the observer is in the canonical orientation $\mathbf{q}'$, then the matrix **Q** applied to $\mathbf{h}'$ will produce the current hidden axis **h**:

$$\mathbf{h} = \mathbf{Q}\,\mathbf{h}' . \tag{1}$$

For example, suppose that the observer is located at the origin of the global coordinate system and is facing in the direction of the $-Z$ half-axis (figure 3). To the right and left of the observer lie the $-X$ and $+X$ half-axes, above and below are the $+Y$ and $-Y$ half-axes, and forwards and backwards lie the $-Z$ and $+Z$ half-axes, respectively. The 3-D subspace spanned by the X, Y and Z axes is orthogonal to the hidden W axis, which is represented by the unit vector $\mathbf{h}' = [0\ 0\ 0\ 1]^{\mathrm{T}}$. If, at some later time, the orientation of the observer is represented by the rotation matrix **Q**, then the current hidden axis (viz. local W axis) is simply $\mathbf{Q}\,[0\ 0\ 0\ 1]^{\mathrm{T}}$, namely the fourth column of matrix **Q**.

The current value of the observer along the hidden axis **h** determines which cross section is taken perpendicular to **h**. The observer's position g along the hidden axis **h** is given by

$$g = \mathbf{p} \cdot \mathbf{h}. \tag{2}$$

The idea, then, is to display only the 3-D subspace that lies perpendicular to the current hidden axis and that has the same position along the hidden axis as the observer.

Our method for determining 3-D cross sections of polyhedra lying in four-space involves checking each edge of every polyhedron for intersection with the 3-D cross section. To detail this procedure, suppose that two vertices $\mathbf{v}_1$ and $\mathbf{v}_2$ of a polyhedron lie along an edge. Edge position may be parameterized by expressing a point **e** along it as a combination of the vertex $\mathbf{v}_1$ and the vertex $\mathbf{v}_2$:

$$\mathbf{e} = (1 - \alpha)\,\mathbf{v}_1 + \alpha\,\mathbf{v}_2\,,\ \alpha \in [0,1]\,. \tag{3}$$

For the point **e** to lie within the current 3-D cross section, its position along the hidden axis **h** must be identical to that of the observer:

$$\mathbf{e} \cdot \mathbf{h} = g. \tag{4}$$

By combining equations (3) and (4), one finds the value of parameter α for which the cross section intersects the edge:

$$\alpha = (g - \mathbf{v}_1 \cdot \mathbf{h}) \,/\, (\mathbf{v}_2 \cdot \mathbf{h} - \mathbf{v}_1 \cdot \mathbf{h}). \tag{5}$$

If the value of α is found to lie between 0 and 1, then the cross section intersects the edge. A zero-valued denominator corresponds to the situation in which the cross section is parallel to the edge.

The current 3-D cross section may intersect a given boundary element not at all, at just a single point, along a line segment, in a polygon, or in a volume. The only cases that we render are those that result in polygons. The primitives that we use give rise to planar polygons that are triangulated prior to transmission to the 3-D graphics engine.

2.1.4 Culling. Note that one may define the direction of the vector normal to each boundary element and thereby define front volumes and back volumes.

Volumes facing towards the back may be culled prior to cross section determination in the interests of efficiency. We implement back-volume culling by checking to see whether the normal **n**, applied to each vertex **v** of a boundary element, faces away from the observer at position **p**. If the inequality

$$(\mathbf{v} - \mathbf{p}) \cdot \mathbf{n} \geq 0 \qquad (6)$$

is true of each vertex, then the boundary element is ignored in further calculations. Note, in addition, that boundary elements that lie entirely behind the observer need not be subject to the cross section calculation. If the inequality

$$(\mathbf{v} - \mathbf{p}) \cdot \mathbf{q} \leq 0 \qquad (7)$$

holds true for each vertex **v** of a boundary element for observer position **p** and orientation **q**, then the element may safely be ignored in the cross section determination. A restriction on the field of view that can be expressed in terms of a half-field angle β can be used to eliminate a greater number of boundary elements. If the inequality

$$(\mathbf{v} - \mathbf{p}) \cdot \mathbf{q} \,/\, (\,|\mathbf{v} - \mathbf{p}|\; |\mathbf{q}|\,) \leq \cos \beta \qquad (8)$$

holds true for all vertices **v** of a boundary element, then the entire element lies outside the cone with half-angle β that describes the field of view. Back-volume culling and location behind are two among a number of visibility criteria that may be used to reduce the number of polyhedra examined in the cross section calculation.

2.1.5 Global to Local Coordinate Transformation. The coordinates of the visible polygons in the 3-D cross section are specified in the 4-D global coordinate system and must be converted to 3-D local coordinates for 3-D to 2-D rendering. The transformation of a polygon vertex, described in global coordinates by a 4-D vector **v**, into a polygon vertex described in viewer-centered coordinates by a 4-D vector **u**, is given by:

$$\mathbf{u} = \mathbf{Q}^{-1}(\mathbf{v} - \mathbf{p}). \qquad (9)$$

The 4-D vector **u** is converted into the desired 3-D vector **u'** by simply dropping its last *w*-coordinate entry.

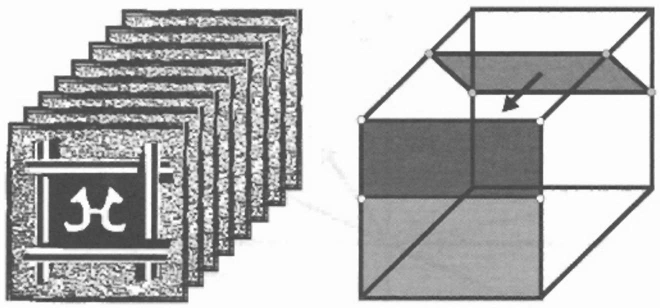


Figure 4. *Three-dimensional textures using two-dimensional texture-mapping. See text for discussion.*

These 3-D vectors **u'** are the vertices of triangles which can be sent directly to a standard 3-D graphics engine like OpenGL (OpenGL ARB et al., 1997), and which require no further geometric transformation.

2.1.6 Texture-Mapping. We texture-map the polygons in the current 3-D cross section of the 4-D environment to render them visible. Three-dimensional textures are needed because a polyhedron that serves as a boundary element in the 4-D environment is a boundary element at each and every one of its potentially visible points. Accordingly, texture must be defined at each point in its volume.

We have implemented the texture mapping of 3-D textures in two ways. The first uses 3-D texture mapping defined by SGI's 3-D texture-mapping extensions for OpenGL, on an Onyx2 with InfiniteReality graphics subsystem. Three-dimensional texture mapping is performed in hardware at high speed on this computer. The advantage of using true 3-D textures is that a 3-D texture may vary arbitrarily throughout the entire volume of a polyhedral boundary element. The disadvantage, however, is that hardware 3-D texture mapping is not yet a widely available feature.

Our second implementation uses 2-D texture mapping, which is a hardware function on inexpensive graphics cards for personal computers. The idea is to stack a 2-D image or texture to create a 3-D texture. (See figure 4.) To each vertex of a boundary element is associated a vector of 3-D texture coordinates. When computing the cross section of a particular boundary element, one also determines the cross section of its

3-D texture by using the parameters α (equation (5)) for each edge to interpolate between corresponding texture coordinates. This set of 3-D texture coordinates must then be projected onto two-space so that 2-D texture mapping may be performed. In our implementation, we have chosen to project texture coordinates in parallel onto the front faces of primitive boundary elements; these front faces are indicated in figure 1 by shading. Two versions of the pentahedron and the hexahedron are used: the first projects texture onto a front triangular face, and the other projects texture onto a front rectangular face.

2.2 Methods for the Display of *n*D Environments for $n \geq 4$

We turn now to the generalization of our display methods to *n*D space in cases in which $n \geq 4$. Observer position is represented in *n*D space by a vector $\mathbf{p} = [p_1 \ldots p_n]^T$. Observer orientation is represented by an *n*D vector $\mathbf{q}$ of unit length or by an $n \times n$ rotation matrix $\mathbf{Q}$ that rotates a canonical orientation vector $\mathbf{q}'$ into the current orientation $\mathbf{q}$. The rotation matrix may, again, be constructed as the composition of elementary rotations in planar subspaces.

The computation of the 3-D cross section involves a hidden subspace of dimension $m = n$-3. The hidden subspace lies orthogonal to the 3-D cross section; it changes in response to observer motion and is computed using the current rotation matrix. Suppose that the m unit vectors $\mathbf{h}'_1, \ldots, \mathbf{h}'_m$ provide a complete orthonormal basis for the *m*D hidden subspace when the observer is in canonical orientation $\mathbf{q}'$. One can form of these vectors an $n \times m$ matrix $\mathbf{H}'$ by placing the m vectors $\mathbf{h}'_1, \ldots, \mathbf{h}'_m$ into the columns of $\mathbf{H}'$. The current hidden subspace is then produced by applying the current rotation matrix $\mathbf{Q}$ to $\mathbf{H}'$:

$$\mathbf{H} = \mathbf{Q}\,\mathbf{H}' . \qquad (10)$$

The m columns $\mathbf{h}_1, \ldots, \mathbf{h}_m$ of matrix $\mathbf{H}$ provide a complete orthonormal basis for the current hidden subspace.

The observer's current position within the hidden subspace $\mathbf{H}$ determines the cross section. This position is represented by a row-vector $\mathbf{g} = [g_1 \ldots g_m]$ given by

$$\mathbf{g} = \mathbf{p}^T \cdot \mathbf{H} . \qquad (11)$$

The 3-D cross section of $(n-1)$-dimensional boundary elements involves checking each edge of each element for intersection. Suppose that a point $\mathbf{e}$ along the edge is given parametrically in terms of the edge's vertices $\mathbf{v}_1$ and $\mathbf{v}_2$, as in equation (3). For the point $\mathbf{e}$ to lie within the current cross section, its position within the hidden subspace must be identical to that of the observer:

$$\mathbf{e}^T \cdot \mathbf{H} = \mathbf{g} \qquad (12)$$

The value of parameter α for which the cross section intersects the edge must satisfy the m simultaneous equations:

$$\alpha = (g_i - \mathbf{v}_1 \cdot \mathbf{h}_i) \,/\, (\mathbf{v}_2 \cdot \mathbf{h}_i - \mathbf{v}_1 \cdot \mathbf{h}_i),\ i = 1, \ldots, m\,. \qquad (13)$$

If a single value for α satisfies the m equations (13) and that value lies between 0 and 1, then the cross section intersects the edge within the confines of the boundary element.

Once again, the coordinates of the visible polygons in the 3-D cross section are specified in the *n*D global coordinate system and must be converted to 3-D local coordinates for 3-D to 2-D rendering. The transformation of an *n*D polygon vertex $\mathbf{v}$ into an *n*D polygon vertex $\mathbf{u}$ described in viewer-centered coordinates is given by:

$$\mathbf{u} = \mathbf{Q}^{-1}\,(\mathbf{v} - \mathbf{p}). \qquad (14)$$

The *n*D vector $\mathbf{u}$ is converted into the desired 3-D vector $\mathbf{u}$' by simply dropping its last n-3 entries.

Texture-mapping an *n*D environment generalizes the 4-D case. Suppose that the dimension of the actual texture to be used in hardware texture mapping is t (typically 2 or 3 but perhaps a larger number). The *n*D texture coordinates that correspond to a boundary element's vertices are used to determine texture coordinates for polygons that lie within the current 3-D cross section. The texture coordinates of the polygons are given by *n*D vectors that must be projected onto the lower-dimensional *t*D space. The simplest class among a wide variety of possible projections from *n*D to *t*D space are the parallel projections, and, of these, the simplest merely drops the last $n - t - 1$ coordinates.

3 Four-Dimensional Environment Creation and Display

We have developed a computer graphics engine and a level editor and modeling program, both of which use the methods described in the previous section for rendering 4-D environments. The software runs on PCs, workstations, and graphics supercomputers.

3.1 Computer Graphics Engine

The immediate purpose of the software development was to create tools for psychological experiments on search, navigation, and object recognition in high-dimensional spaces. We felt that one could learn best to get around in high-dimensional environments by interacting with them as freely as possible.

Towards this end, we equipped the engine with a motion model featuring 4-D collision detection, a subset-of-C scripting language, and code for multiple networked users based on TCP/IP. The software uses OpenGL for 3-D graphics rendering and implements visual effects such as transparency, lightmaps, fog, and mirrors. Many inexpensive graphics cards for personal computers have hardware for single-pass multitexturing, and we use this feature in our implementation to add lightmaps and other secondary textures to visible surfaces. The engine also includes a threaded, stack-based machine for interpreting scripts in a subset-of-C scripting language. The scripts are used to control elevators, platforms, sound effects, blinking lights, and other interactive elements of the 4-D environments. The software also provides for multiple-user experience using TCP/IP. At the moment, the visible representation of users is limited to a low-polyhedron-count, robot-like character that can adopt standing and flying postures and which possesses keyframes for walking.

3.1.1 Motion Model. We now detail the interface. The engine takes the *Y* axis to be oriented vertically along the direction of simulated gravity. The user can stand, walk, jump, swim, run on, or fly above a 3-D ground spanned by the *X*, *Z*, and *W* axes. Gravity pulls the user down the *Y* axis to the ground when the user jumps, falls, or sinks. Movement is controlled through either a mouse freelook interface or an interface for a head-mounted display. The interfaces are similar in that they let the user control three angles that determine orientation and direction of motion in 4-D space. The three angles describe user orientation in the *XZ*, *YZ*, and *ZW* planes, respectively. User orientation in the *XZ* plane defines a yaw θ_{xz} that corresponds to a rotation matrix

$$\mathbf{R}_{xz} = \begin{bmatrix} \cos\theta_{xz} & 0 & \sin\theta_{xz} & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta_{xz} & 0 & \cos\theta_{xz} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{15}$$

User orientation in the *ZW* plane defines a *zaw* θ_{zw} that corresponds to a rotation matrix

$$\mathbf{R}_{zw} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \cos\theta_{zw} & \sin\theta_{zw} \\ 0 & 0 & -\sin\theta_{zw} & \cos\theta_{zw} \end{bmatrix}. \tag{16}$$

These two matrices together determine a matrix $\mathbf{R}_h = \mathbf{R}_{zw}\,\mathbf{R}_{xz}$ that specifies the user's heading within the *XZW* subspace. The orientation of the viewpoint is further modulated by a pitch θ_{yz} that corresponds to a rotation matrix

$$\mathbf{R}_{yz} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta_{yz} & \sin\theta_{yz} & 0 \\ 0 & -\sin\theta_{yz} & \cos\theta_{yz} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{17}$$

The rotation matrix **Q** that defines the current orientation (equation (1)) is defined to be the product

$$\mathbf{Q} = \mathbf{R}_h\,\mathbf{R}_{yz} = \mathbf{R}_{zw}\,\mathbf{R}_{xz}\,\mathbf{R}_{yz}\,. \tag{18}$$

The roll of the observer in the *YZ* plane is not accessible directly through the interface: except in special cases, the user is always upright.

Figure 5 illustrates rotation in the *ZW* plane. The initial orientation of the observer is such that a hypersphere (center) and a double door (right) are visible (panel A). By changing orientation, the hypersphere and the door are made to disappear (frames B through D), and a new door is made to appear (frame H). Such a change in orientation is experienced as a smooth and controllable turning by a user of the engine software.

3.1.2 Freelook Interface. Using a mouse with three buttons, the freelook interface is implemented as follows. The left mouse button moves the user forward along the current heading, and the right mouse button moves the user backward. Motion of the mouse upwards and downwards (viz. away from and towards the user, respectively) controls pitch. Sideways motions of the mouse control either the yaw in the *XZ* plane (if the middle mouse button is not held down) or the zaw in the *ZW* plane (if the middle mouse button is held down). In particular, if the middle mouse button is not held down, then movement of the mouse to the left turns the user to the left, and movement to the right turns the user to the right. Likewise, when the middle button is held down, movement of the mouse to the left turns the user to the *nim* and movement to the right turns the user to the *bor*. (The terms *nim* and *bor* refer to zaws of negative and positive value, respectively, and are analogous to left and right.) Although it is possible to move and turn simultaneously with this interface, it does have a drawback: one cannot change yaw and zaw simultaneously.

3.1.3 Position-Tracking, PINCH Glove, and Head-Mounted Display Interface. We have implemented a similar interface for use with a head-mounted display (HMD). In our implementation, an Onyx2 workstation is used to drive an n-Vision HiRes HMD. Atop this HMD is an InterSense tracking cube with accelerometers that report 3-D orientation. A user wearing the HMD also wears Fakespace PINCH Gloves. Pinching the right index or the right ring finger and thumb moves the user forwards and backwards, respectively. The elevation of the head controls pitch. Turning the head left and right causes the viewpoint to turn left and right, if the right middle finger is not pinching the right thumb. If the right middle finger is pinching the thumb, turning the head left and right causes the viewpoint to turn *nim* and *bor*.

3.1.4 Collision Detection. Motion is constrained by the simulated solidity of floors, walls, ceilings, and objects in the environment. Attempted motion through such solids by the user is detected by an algorithm that checks whether the attempted motion would place a model representing the user partly or wholly within some foreign solid object. Although general collision detection in four dimensions can involve rather complicated algorithms (Latombe, 1991; Lin & Gottschalk, 1998), a simple ray-casting technique can handle tasks such as determining the height of the viewpoint position above the ground. From the position of the viewpoint, one casts a ray represented by a unit-length vector that passes through one or more boundary elements. The aims are to determine which of these boundary elements is closest, the point of intersection, and the distance of the point of intersection from the viewpoint.

Referring to figure 6, a point within the subspace spanned by a boundary element has the form

$$\mathbf{c}_0 + \epsilon_1\mathbf{e}_1 + \epsilon_2\mathbf{e}_2 + \epsilon_3\mathbf{e}_3 \qquad (19)$$

where the parameters ϵ_1, ϵ_2, and ϵ_3 describe location within the subspace based on corner position $\mathbf{c}_0$ and spanned by edges $\mathbf{e}_1$, $\mathbf{e}_2$, and $\mathbf{e}_3$. If the ray $\mathbf{d}$ cast from viewpoint position $\mathbf{p}$ intersects that subspace, then the following equality holds for some value of the distance parameter δ:

$$\mathbf{p} + \delta\mathbf{d} = \mathbf{c}_0 + \epsilon_1\mathbf{e}_1 + \epsilon_2\mathbf{e}_2 + \epsilon_3\mathbf{e}_3\,. \qquad (20)$$

Forming the matrix $\mathbf{C}$ with columns $\mathbf{e}_1$, $\mathbf{e}_2$, $\mathbf{e}_3$, and -$\mathbf{d}$, respectively, one finds the parameters that describe the intersection as follows, when the matrix is invertible:

$$\mathbf{C}^{-1}(\mathbf{p} - \mathbf{c}_0) = [\epsilon_1\ \epsilon_2\ \epsilon_3\ \delta]^{\mathrm{T}}. \qquad (21)$$

Inequalities on the edge parameters inform one whether the intersection lies within the confines of the boundary element; these inequalities depend on the

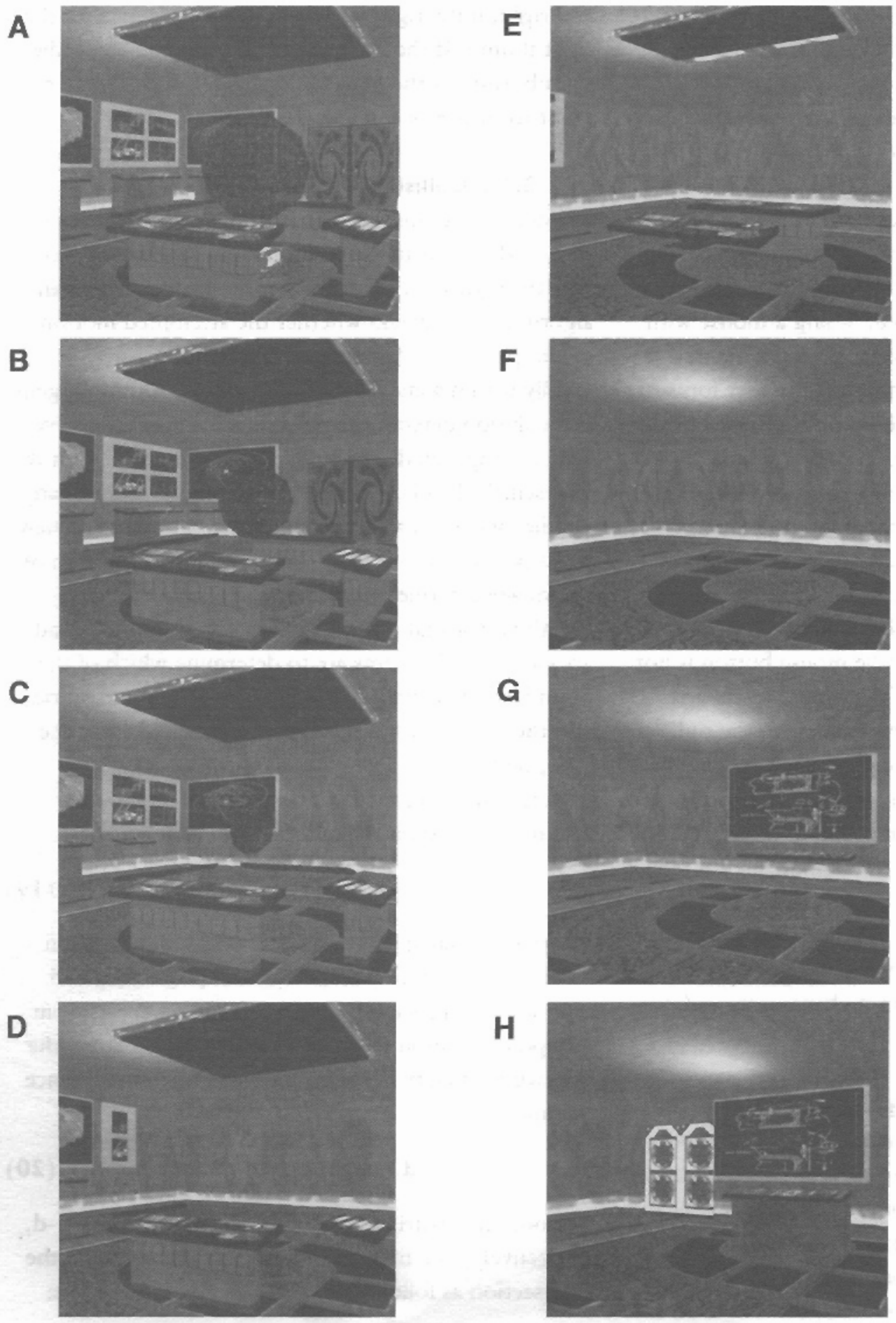


Figure 5. *A rotation in the ZW plane. The effects of changing zaw on view orientation are visible in these eight sequential frames labeled A through H. See text for discussion.*

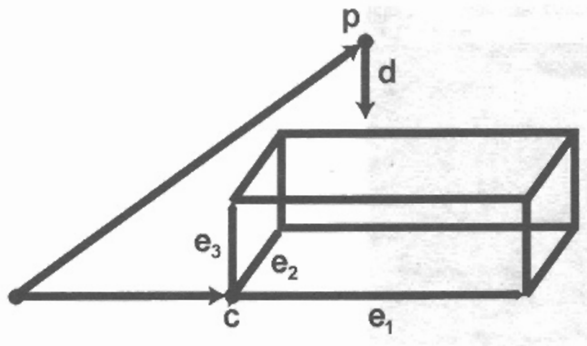


Figure 6. *Simple ray-casting for collision detection. See text for discussion.*

geometric primitive chosen (figure 1). One searches through all possible boundary elements for the one that produces the smallest distance δ. The generalization to higher dimensions is immediate.

3.2 Level Editing and Modeling

The editor provides a graphical user interface for 4-D model creation and level editing. Its features include a hierarchical object database; the simultaneous provision of both 2-D orthographic cross sections and fully-rendered 3-D views; an intuitive and flexible interface for texture-mapping; a full set of primitive objects; a hole editor; an extrusion editor; facilities for tesselation, tetrahedronization, and vertex editing; and multiprocessing support for cross section computation using POSIX threads.

Figure 7 shows a screen shot of the editor. The editor's primary window has four panels, each of which can be configured to provide a 2-D wireframe orthographic cross section (as at top left) or a 3-D perspective view (as at top right and bottom right). Any one of these panels can be toggled to provide a full-screen window. The 2-D orthographic cross sections can be provided in either the global coordinate system or in a coordinate system local to a selected object. In the latter case, only the selected object is depicted.

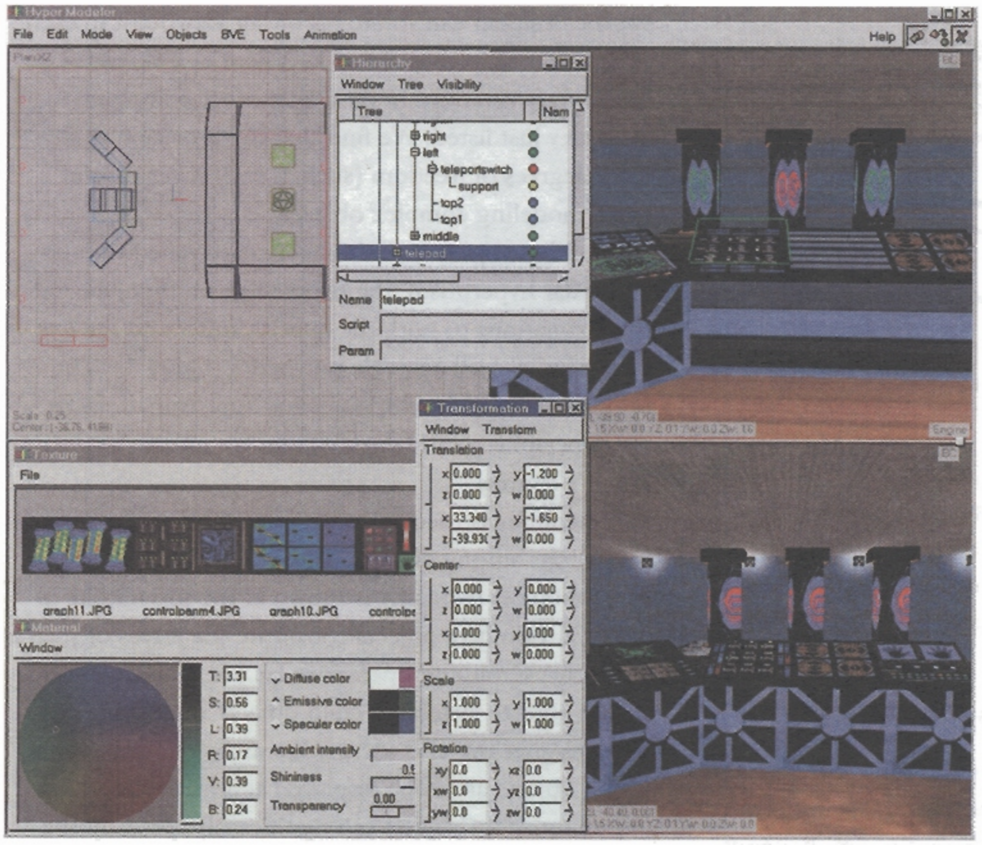
Designers can interact with the 3-D view in two ways. The first is through the interface described in section 3.1.1, and the second provides further degrees of freedom in viewpoint position and orientation. In particular, the designer can rotate his or her viewpoint in each of the six basic planes *XY*, *XZ*, *XW*, *YZ*, *YW*, and *ZW*. Elementary rotations in these planes are composed in the order just listed. We find that the provision of these extra degrees of freedom (such as roll) is very useful when modeling complex objects.

Primitive objects include the hypertetrahedron, the triangular hyperprism, the hypercube, multipolyhedral approximations to both the hypercylinder and the hypersphere, as well as single polyhedra (figure 1). The resolutions of the multipolyhedral approximations to the hypersphere and hypercylinder are under the designer's control.

A hierarchy is used to control the inheritance of transformations among objects. The window that is used to control object hierarchy is shown at the top middle of figure 7. Primitive objects constitute the leaves of such a hierarchy. Grouping operations let one define complex, parent objects formed of child objects and primitives. The designer can also use the hierarchy editor to select which objects are to be displayed at any one time; hiding unnecessary objects can significantly speed up the display of large, complex environments.

The designer performs geometric transformations of objects using either a click-and-drag interface or a numerical transformation editor (pictured at the bottom center of figure 7). Transformations include translations, scalings, and rotations about arbitrary central points. Snap-to-grid functionality is provided with the click-and-drag interface. The numerical transformation editor lets one transform objects using either global coordinates or coordinates local to the current viewpoint.

The designer uses a hole editor to create doors, windows, and the like. Figure 8 shows a rectangular solid hole in an octahedral boundary element. The hole editor tessellates the original boundary element and its texture to accommodate the hole. This tessellation procedure uses the original boundary element to determine new boundary elements that occupy the original volume, minus the hole. In level construction, one often defines a solid wall to have both an inner face and an outer face. To create a hole that passes through such a wall, one creates corresponding holes in boundary ele-



right
left
teleportswitch
support
top2
top1
middle
telepad

Name telepad
Script
Param

Translation
x 0.000 y -1.200
z 0.000 w 0.000
x 33.340 y -1.650
z -39.93(w 0.000

Center
x 0.000 y 0.000
z 0.000 w 0.000
x 0.000 y 0.000
z 0.000 w 0.000

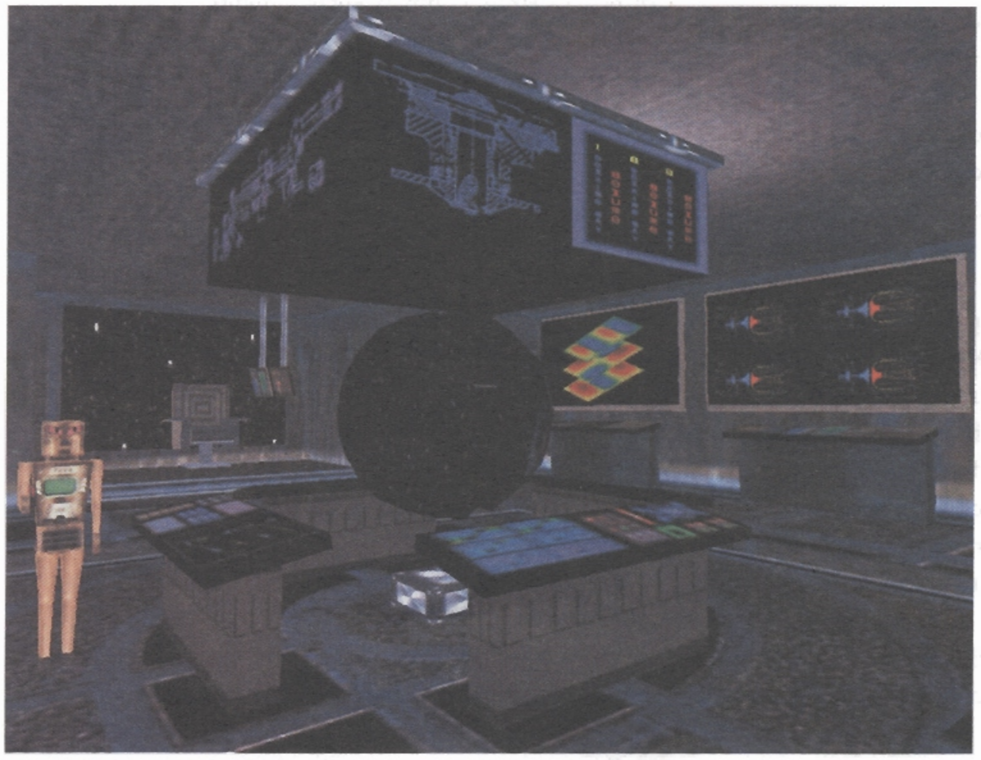
Scale
x 1.000 y 1.000
z 1.000 w 1.000

Rotation
xy 0.0 xz 0.0
xw 0.0 yz 0.0
yw 0.0 zw 0.0

graph11.JPG controlpanm4.JPG graph10.JPG controlpa

T: 3.31
S: 0.56
L: 0.39
R: 0.17
V: 0.39
B: 0.24

Diffuse color
Emissive color
Specular color
Ambient intensity
Shininess 0.5
Transparency 0.00



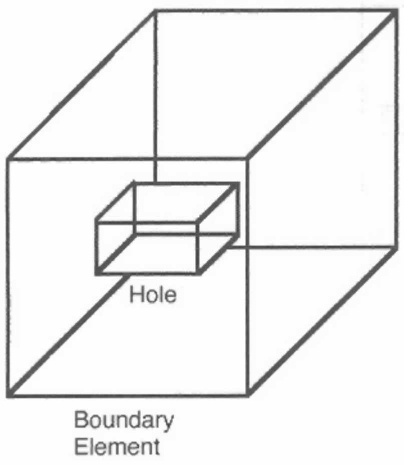


Figure 8. *Rectangular solid hole within an octahedral boundary element. See text for discussion.*

ments of both the inner and outer faces and joins the corresponding vertices of the holes appropriately.

The texture-mapping interface includes a texture palette for selection and a texture coordinate editor for positioning. The application of 2-D textures to 3-D boundary elements by the editor involves the selection of a privileged face. (See figure 1.) When the cross section of the boundary element is parallel to the privileged face, the 2-D texture is seen in the standard fashion. When the cross section is not parallel to the privileged face, then the 2-D texture is distorted by the projection indicated in figure 4. When 3-D texture maps are used, such distortions are not an issue. It is thus important for the designer of a 4-D environment that is texture-mapped using 2-D textures to choose one or more privileged subspaces in which texture mapping does not appear distorted.

The designer specifies the normal directions of the primitives, and this information is needed to perform back-volume culling (section 2.1.4).

For level design, the type of boundary element must be specified. These types include floor, wall, ceiling, water, and so on. Type information is needed for the engine to function properly. To help the designer with this process, the editor color-codes the boundary elements according to type. Further information required by the engine includes pointers from particular objects to scripts and the specification of object kinematic properties, which are specified through the hierarchy editor.

To date, our experience in designing 4-D environments with the editor underlines the importance of thinking in one more dimension than usual. For instance, one might start designing an environment by specifying a floor plan. However, a standard floor plan with markings in a plane area does not help with 4-D design. Four-dimensional floors are volumetric, and a proper floor plan consists of markings within a volume.

Figure 9 shows how three floor plans drawn in the *XZ*, *ZW*, and *XW* planes may be combined to provide a volumetric floor plan in four dimensions. Each of the 2-D floor plans indicates the positions of four waist-high control panels that surround a central hypersphere. When combined, these provide a total of six control panels (two along each of the *X*, *Z*, and *W* axes) that surround the central hypersphere.

Figure 10 shows a 3-D cross section of a spaceship helm, created using the editor and viewed by the engine. Four waist-high control panels are seen to surround a hypersphere with starfield texture mapping. A 2-D floor plan of this cross section would indicate the four control panels and the hypersphere. As suggested by figure 9, there are two further control panels that surround the hypersphere, and these are aligned along the axis orthogonal to the cross section viewed in figure 10. A

Figure 7. *Window layout of the level editor and modeling program. The top left panel shows a 2-D floor plan of part of a 4-D environment that includes a teleport chamber, which is rendered in 3-D from user-controlled viewpoints in the top right and bottom right panels. The two 3-D views at right reveal different aspects of the 4-D scene; note that the teleports at top include two that are blue-green, and those at bottom are all blue-red. Also visible are the model hierarchy editor (top center), the material editor (bottom left), texture palette (middle left), and numerical transformation editor (bottom middle).*

Figure 10. *Ship's helm with hyperspherical starfield display viewed with the engine. Screenshot taken from an IBM-compatible personal computer with an nVidia Riva TNT 3-D graphics card. A low-polyhedron-count, 4-D character is visible at left. Hyper software may be downloaded from www.cvr.uci.edu/dzmura.*

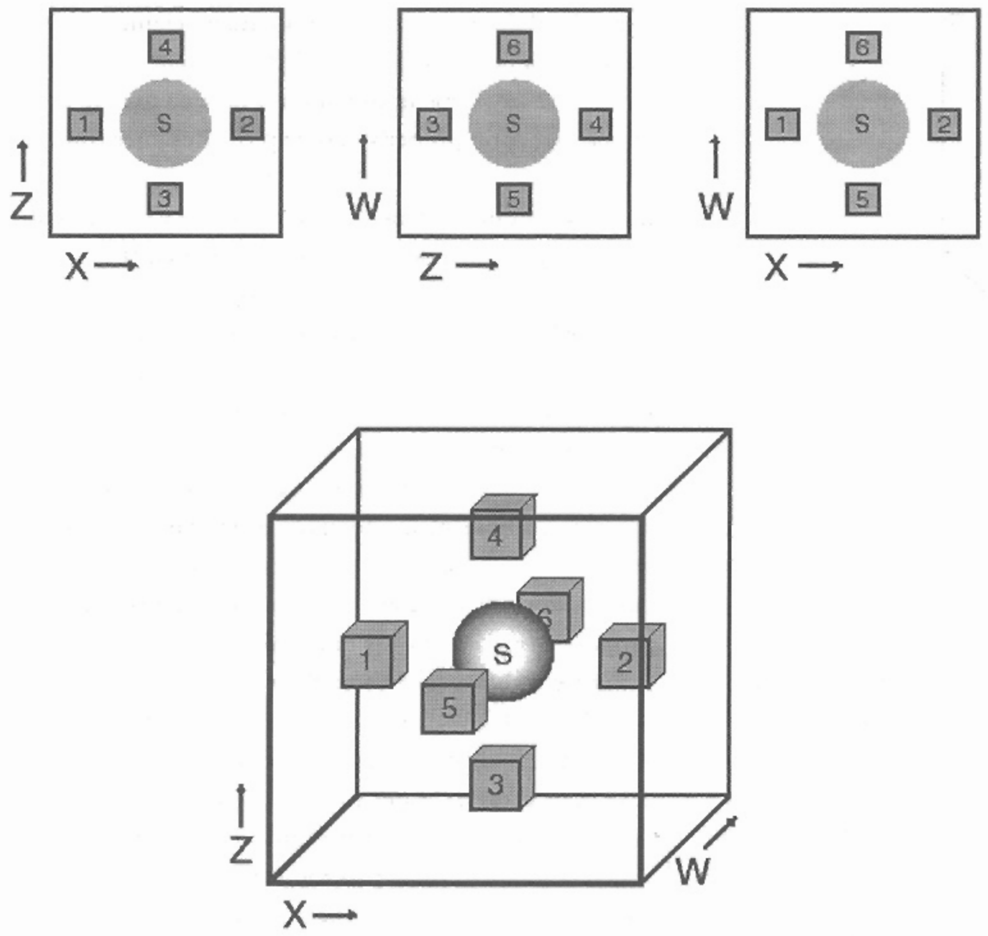


Figure 9. *Two-dimensional floor plans in the XZ, ZW, and XW planes (above) and their joint representation as a volumetric floor plan (below). Six "control panels" (labeled 1 through 6) surround a central hypersphere S. Figure 10 shows a rendered view. See text for discussion.*

volumetric floor plan can capture this information and is useful when designing 4-D levels.

4 Conclusions

We have presented methods for displaying environments with four or more spatial dimensions. We have also described software that uses these methods to create high-dimensional environments and to display them. The computations involved can be conducted rapidly enough on a present-day IBM-compatible personal computer for a user to interact in real time with fully textured high-dimensional environments.

Rather than working with equations and various parametric forms of manifolds to be rendered, we have chosen to represent boundaries using a set of geometric primitives. This is the sort of representation that has proven so successful in 3-D computer graphics. The primary drawback of this choice is that such a representation can provide only an approximation to curved boundaries. Of course, with increasingly small and increasingly numerous polyhedra, one can create an arbitrarily good approximation to any curved boundary in four dimensions. Yet increasing the number of polyhedra to be rendered reduces the rate at which frames can be rendered.

We find that a room like the helm (figure 10), which has just over 1,600 boundary elements, can be rendered by the engine in 32-bit color at nearly thirty frames per second at a spatial resolution of 1024 × 768 using a personal computer with an Intel Celeron 500 MHz CPU and an nVidia Ultra TNT2 3-D graphics card. All of the geometric calculations are conducted by the CPU. As depicted in figure 2, the geometric primitives are subject to a frame-by-frame calculation that determines their cross section with a particular 3-D subspace. Triangles in the cross section and their texture coordinates are then sent to the OpenGL (or other) graphics pipeline. As noted in Appendix A, the homogeneous transformations that are needed in geometric calculations pertaining to 4-D environments are represented by 5-D matrices. Acceleration of 5-D matrix operations has the potential to speed the geometric calculations enormously.

The use of the software goes beyond the research in human perception and navigation for which it was originally designed. Beyond potential applications in education and entertainment, the software can be used to visualize events in spacetime. Three-dimensional solid objects that change and interact as a function of time can be extruded appropriately to create hyperobjects in four spatial dimensions. With our methods, the events can then be viewed from an arbitrary position and orientation in spacetime in an interactive and immersive fashion. One need no longer be confined to watching a "movie" of a sequence of events in which the time axis is the hidden axis. Rather, one can view events from arbitrary perspectives in spacetime.

Fascination with the fourth dimension dates to the nineteenth century and has given rise to many popular accounts of dimensionality over the years, some of them more mathematical in nature (Hilbert & Cohn-Vossen, 1952) and some of them less so (Hinton, 1904; Abbott, 1952; Manning, 1960; Rucker, 1984). A recurring theme is the question of how beings of low dimension interact with beings and objects in higher dimensions. For instance, Abbott's *Flatland* (1952) concerns itself with the interaction of two- and three-dimensional beings.

With our rendering method, it occurs that if the observer of the high-dimensional environment does not know how to engage the interface to the higher dimension(s), then the observer will experience the geometry of a normal 3-D environment. By analogy, may it perhaps be the case that we already live our lives in an environment with four or more noncompactified spatial dimensions, but simply do not know how to work the interface?

High-dimensional environments differ from our own in the way that lights and sounds would work. For instance, a point source of light in four dimensions presumably radiates in all directions and so causes visible lights and shadows within a 3-D subspace. To a creature within the 3-D subspace, these lights and shadows would almost never have a visible cause. Likewise, one should be able to hear 4-D events, even if they lie outside the visible subspace. We have witnessed no such effects in our daily experience.

Much of the promise of virtual reality lies in the ability to experience environments that could not possibly exist. The categories of perception identified by Kant (1791) include space, time, and causality—all of which are leading candidates for virtual tampering. With the present 4-D environments, we have made significant headway on the category space. We look forward to future environments that also violate time and causality in consistent and useful ways.

Acknowledgments

We thank Scott Richman for his helpful comments and Barb Krug for her digital artwork. This work was supported by NIH EY10014, NSF DB9724595, and a fellowship to Colantoni from the Région Rhône-Alpes, France.

References

Abbott, E. A. (1952). *Flatland*. New York: Dover.

Banchoff, T. F. (1990). *Beyond the Third Dimension: Geometry, Computer Graphics, and Higher Dimensions.* New York: Scientific American Library.

Feiner, S., & Beshers, C. (1990). Visualizing n-dimensional virtual worlds with n-vision. *Computer Graphics, 24*(2), 3–38.

Hanson, A. J., & Heng, P. A. (1992). Illuminating the fourth dimension. *IEEE Computer Graphics and Applications* (July), 54–62.

Hanson, A. J., Munzner, T., & Francis, G. (1994). Interactive methods for visualizable geometry. *IEEE Computer, 27*(7), 73–83.

Hilbert, D., & Cohn-Vossen, S. (1952). *Geometry and the imagination*. (P. Nemenyi, Trans.). New York: Chelsea.

Hinton, C. H. (1904). *The fourth dimension*. London: Swann Sonnenschein.

Hollasch, S. (1991). *Four-space visualization of 4-D objects*. Unpublished master's thesis, Arizona State University.

Kant, I. (1791/1956). *Kritik der reinen vernunft*. Hamburg: Felix Meiner.

Latombe, J.-C. (1991). *Robot motion planning*. Boston: Kluwer.

Lin, M., & Gottschalk, S. (1998). Collision detection between geometric models: a survey. *Proceedings of IMA Conference on Mathematics of Surfaces*, http://citeseer.nj.nec.com/Lin98collision.html.

Manning, H. P., (Ed.). (1960). *The fourth dimension simply explained*. New York: Dover.

Noll, M. A. (1967). A computer technique for display n-dimensional hyperobjects. *Communications of the ACM, 10*(8), 469–473.

OpenGL ARB, Woo, M., Neider, J., & Davis, T. (1997). *OpenGL programming guide* (2nd ed.) Reading, MA: Addison-Wesley.

Rucker, R. (1984). *The fourth dimension: Toward a geometry of higher reality*. Boston: Houghton-Mifflin.

Seyranian, G. D., Krug, B., Richman, S., & D'Zmura, M. (1999). Search and navigation in four-dimensional environments. *Investigative Ophthalmology and Visual Science, 40*(4), S801.

Steiner, K. V., & Burton, R. P. (1987). Hidden volumes: The 4th dimension. *Computer Graphics World* (February), 71–74.

Appendix A: Mathematical Preliminaries

We use vectors and matrices to represent positions, orientations, and transformations in the virtual environments. Unit vectors along four mutually perpendicular axes provide a complete orthonormal basis for 4-D space. A 4-D column vector **v** has entries that specify its positions along the axes:

$$\mathbf{v} = [v_1\ v_2\ v_3\ v_4]^T , \qquad (A1)$$

(in which the superscript T denotes transpose).

Operations such as dot product and the determination of length work in the expected manner. To illustrate the dot product, suppose that unit vectors **x**, **y**, **z** and **w** along the X, Y, Z, and W axes, respectively, provide an orthonormal basis. One can then use the dot product to find the x coordinate v_x of vector **v** in that basis:

$$v_x = \mathbf{v} \cdot \mathbf{x} = v_1 x_1 + v_2 x_2 + v_3 x_3 + v_4 x_4 . \qquad (A2)$$

The y, z, and w coordinates are found in a similar fashion:

$$v_y = \mathbf{v} \cdot \mathbf{y},\ v_z = \mathbf{v} \cdot \mathbf{z} \text{ and } v_w = \mathbf{v} \cdot \mathbf{w} .$$

Calculating the length $|\mathbf{v}|$ of vector **v** generalizes in a simple way the calculation in three dimensions:

$$\begin{aligned} |\mathbf{v}| &= (v_x v_x + v_y v_y + v_z v_z + v_w v_w)^{1/2} \\ &= (\mathbf{v} \cdot \mathbf{v})^{1/2} . \end{aligned} \qquad (A3)$$

Finally, one can find the angle θ between two vectors **v** and **w** as follows:

$$\theta = \cos^{-1} ((\mathbf{v} \cdot \mathbf{w})/(|\mathbf{v}|\ |\mathbf{w}|)) . \qquad (A4)$$

Two vectors **v** and **w** are perpendicular to each other if their dot product has value zero:

$$\mathbf{v} \cdot \mathbf{w} = 0. \qquad (A5)$$

In the case of a zero-valued dot product, the angle θ between the two vectors is 90 degrees or $\pi/2$ radians, because $\cos^{-1}(0) = \pi/2$.

The normal **n** to a set of three vectors $\mathbf{v}_1$, $\mathbf{v}_2$ and $\mathbf{v}_3$, in which no one of the three may be expressed as a linear combination of the other two, can be expressed using the dot product:

$$\mathbf{v}_1 \cdot \mathbf{n} = 0,\ \mathbf{v}_2 \cdot \mathbf{n} = 0,\ \mathbf{v}_3 \cdot \mathbf{n} = 0. \qquad (A6)$$

A nonzero normal vector that satisfies these equations may be found by calculating the determinant of a 4×4 matrix with $\mathbf{v}_1$, $\mathbf{v}_2$, and $\mathbf{v}_3$, and the unknown **n** as its rows. The **n** so determined may be normalized to have unit length, and this unit-length normal vector is unique up to sign.

A translation or shift of position in 4-D space can be specified by a 4-D vector **t** which is added to a vector **a** to provide a transformed vector **b**:

$$\mathbf{b} = \mathbf{a} + \mathbf{t}. \tag{A7}$$

Scaling in 4-D space can be specified by a 4 × 4 diagonal matrix **S** with entries along the diagonal that specify the scale factor along the corresponding axis:

$$\mathbf{b} = \mathbf{S}\,\mathbf{a}. \tag{A8}$$

One can represent a rotation in 4-D space by a 4 × 4 orthogonal matrix; such a matrix **R** satisfies the equation $\mathbf{R}\,\mathbf{R}^T = \mathbf{I}$. We take such rotations to act on the left of a vector **a** to provide a rotated vector **b**:

$$\mathbf{b} = \mathbf{R}\,\mathbf{a}. \tag{A9}$$

The simplest rotations in 4-D space are those that occur within a single plane or 2-D subspace. For instance, rotating the *XY* plane by angle θ about the *Z* and *W* axes can be represented by a matrix $\mathbf{R}_{xy}$ with form:

$$\mathbf{R}_{xy} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{A10}$$

Similar matrices embed two-dimensional rotations within planes spanned by any two of the basis axes in 4-D space; choosing any two of the four possible axes provides six possible 2-D rotations.

Transformations in four-space can be represented using homogeneous transformations in five-space. Suppose that one has a vector **a** that, when subject to a transformation represented by a 4 × 4 matrix **M** acting on the left and then translated by vector **t**, provides a vector **b**:

$$\mathbf{b} = \mathbf{M}\,\mathbf{a} + \mathbf{t}. \tag{A11}$$

One can embed this transformation in five-space using vector $[a_x\ a_y\ a_z\ a_w\ 1]^T$ and a 5 × 5 matrix which is formed by placing the matrix **M** in the first four rows and columns, placing the vector **t** in the first four entries of the fifth column, and placing the vector [0 0 0 0 1] in the last row. Multiplying the two produces a 5-D vector with the form $[b_x\ b_y\ b_z\ b_w\ 1]^T$.