REAL-TIME 4D ANIMATION ON A 3D GRAPHICS WORKSTATION [1]

Clifford M. Beshers
Steven K. Feiner

Department of Computer Science
Columbia University
New York, NY 10027

Beshers@CS.Columbia.Edu
Feiner@CS.Columbia.Edu

Abstract

We describe the general structure of a real-time four dimensional animation system and its implementation on a 3D graphics workstation with hardware support for transformations. Our focus is not on four dimensional geometry, but on techniques of exploiting the graphics hardware and software for this application. Real-time performance is achieved by employing the hardware matrix multiplier to do 4D modeling transformations. Gouraud shading is used to perform 4D depth cueing by relating the color intensity to a function of the fourth coordinate. An $O(n)$ software clipper is described that clips the 4D object between near and far clipping hyperplanes. A constant time method employing color map manipulation and 4D depth cueing is described as a useful high-speed approximation to the hyperplane clipping.

KEYWORDS: Four dimensional transformations, real-time animation, scientific visualization.

Introduction

Mathematicians have long been fascinated with the geometry of higher dimensional objects and spaces, attempting to understand and analyze extremely complicated and non-intuitive structures. Computer graphics provides an excellent means for exploring these objects visually. As early as 1967, Noll used a pen plotter was used to create images of hypercubes [Nol67]. In the mid 1970's, Banchoff [Ban78] explored four dimensional geometry in real-time using BUGS, a multiprocessor, microprogrammable vector graphics system with special microcode for 4D transformations [vDSH74]. With raster graphics, solid, shaded images became possible, but at the sacrifice of real-time speed [FSB82]. Modern raster workstations have again achieved real-time performance, now with three dimensional shaded images, and this paper describes how to exploit the architecture of typical 3D workstations to regain the real-time animation of four dimensional objects.

We describe the general structure and implementation of our four dimensional animation system and provide a small catalog of graphics techniques for exploring four dimensions. [2]

[1] This work supported in part by DARPA grant # N00039-84-C-0165 and an equipment grant from Hewlett-Packard Labs - AI University Grants Program.

Euclidean Four Space

A great deal is suggested by analogy, but while analogy is often a useful guide and stimulus, it provides no proofs, and may often lead one astray if not supplemented by logical reasoning.
[Som58, p. 2]

We begin by giving a brief description of four dimensional space and the hyperobjects residing there, relying heavily on analogy between two and three dimensions to motivate the jump from three to four, despite the warning above. More elegant and fuller descriptions can be found in [Abb52], [Gar77] and [Dew86]. Our description is intended to clarify the mathematics and organization used in the implementation of this program. For more complete mathematical descriptions see [Cox69], [Som58] and [Nol67], the last providing an especially useful and concise description of the mathematics required to implement four dimensional animation.

The mathematics of two and three dimensions generalizes to n dimensions quite simply, assuming a fourth axis perpendicular to each of the familiar three axes of three dimensions. One can imagine an infinity of parallel 3-spaces or hyperplanes all perpendicular to this fourth axis, just as we imagine an infinity of planes parallel to the XY plane and perpendicular to the Z axis. Points residing in 4-space are naturally represented as ordered 4-tuples, (x, y, z, w). It is important to realize that we are considering a fourth spatial dimension, not time as a fourth dimension, although it is interesting to consider a 3D object changing over time as if it were 4D.

[2] We have recently learned that Banchoff has independently developed a similar system [Ban87].

Because any object of dimension n cannot fit into a space of dimension $n-1$ (e.g., a cube cannot fit in the plane without being flattened), we need to reduce a 4D object to a 3D one using projection. Similarly, we need to further reduce the dimension of the 3D object by projecting it onto a 2-dimensional screen for viewing. The simplest projection to consider is orthographic. Given the plane $z = c$ as the image plane, the 3D to 2D orthographic projection of any point (x, y, z) is (x, y, c). Similarly, given a projection hyperplane defined by the equation $w = c$, the 4D to 3D orthographic projection of any point (x, y, z, w) is (x, y, z, c). Notice that, in terms of a computer implementation of orthographic projection, we need only ignore the last coordinate.

Perspective projection from 4D to 3D is also possible. Given a 4D eye point on the w axis at distance d from the origin and a projection hyperplane $w = c$, the projection P' of a point $P = (x, y, z, w)$ is given by

$$P' = \left(\frac{(d-c)x}{d-w}, \frac{(d-c)y}{d-w}, \frac{(d-c)z}{d-w}, w\right) \qquad \text{[Nol67].}$$

This provides a limited viewing model that is sufficient for most purposes: the 4D eyepoint lies on the w axis; the viewing hyperplane is perpendicular to the w axis; parallel projection is always orthographic with the direction of projection coincident with the w axis; the perspective view volume is the 4D analog of a right pyramid.

We proceed to the modeling transformations, rotation, scaling and translation, in four dimensions. Just as in 3-space, four dimensional rotations occur in a plane. Four axes taken two at a time gives six planes of rotation, namely XY, YZ, XZ, XW, YW, and ZW. The rotation matrix M for any pair of axes a, b has entries: $M[i,i] = 1$ except for $M[a,a] = M[b,b] = cos(\theta)$ and $M[i,j] = 0$ except for $M[a,b] = -M[b,a] = sin(\theta)$, $i \neq j$. For example, a matrix for the YW rotation has the form

$$R = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & cos(\theta) & 0 & sin(\theta) \\ 0 & 0 & 1 & 0 \\ 0 & -sin(\theta) & 0 & cos(\theta) \end{pmatrix}.$$

Scaling matrices are just like 3D scaling matrices, with entries along the diagonal

$$S = \begin{pmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & S_w \end{pmatrix}.$$

Translation is also simply generalized, but requires a $5x5$ matrix

$$T = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ T_x & T_y & T_z & T_w & 1 \end{pmatrix}.$$

These modeling transformations are followed by projection from 4-space to 3-space, producing a 3D object, what we will call the *intermediate image*, which can then be rendered in the standard fashion with 3D modeling and viewing transformations. As the object is rotated in 4-space, this 3D intermediate object continually changes shape, just as the projection of a rotating cube on a plane continually changes, and it is partly this constant change that makes these objects so baffling to view. It is important to notice that these 3D modeling transformations can be thought of as image transformations of the intermediate object, i.e., fixing the 4D and changing the 3D modeling transformations produces different views of a static 3D object. Image rotations can be immensely enlightening, especially because these objects are unfamiliar and one often misinterprets their structure with a single view.

In summary, the 4D rendering pipeline consists of the following stages:

- Perform rotation, scaling and translation (modeling transforms) in 4-space.
- Project the object into 3-space to obtain the intermediate image.
- Apply any 3D modeling transforms.
- Apply 3D viewing transforms.

Transformation System

Our implementation has been geared for the increasingly common high performance 3D graphics workstations [Cla82], [ST86]. The important feature of these machines is the hardware matrix multiplier and an associated transformation stack of $4x4$ homogeneous matrices. The top matrix on the stack can be overwritten, or pre- or post-multiplied by another matrix. In normal use, all graphics operations such as $move(x, y, z)$ and $draw(x, y, z)$ are sent to the hardware pipeline, where all coordinates are multiplied by the top matrix, and then passed along to clippers, scalers, and finally to the scan conversion system which draws the primitives into the frame buffer. A good graphics library will make it possible for an application to multiply points by the top matrix without sending them down the pipeline. The result is that the programmer has access to a high speed $4x4$ matrix multiplier, which can multiply two matrices together and transform a large number of 4-coordinate points very quickly.

Recall that in three dimensions, rotation and scaling can be described with only a $3x3$ matrix, but that translation and perspective projection require the fourth row and column of a $4x4$ homogeneous matrix. To get translation in 4-space and perspective projection to 3-space with only matrix multiplication we would need a $5x5$ matrix. If we are willing to sacrifice 4D translation and agree to perform 4D to 3D perspective projection in software, then the remaining operations can be performed entirely with the hardware multiplier.

Thus, the program performs all transformations and projections in a two stage process. First, a $4x4$ matrix describing rotation and scaling in Euclidean 4-space is placed on the matrix stack, and all 4D vertices are multiplied by that matrix, producing a new set of coordinates that represents the object transformed in 4-space. This accomplishes the 4D modeling transformations described earlier.

Next, the vertices are projected from 4-space to 3-space, using either orthographic or perspective projection, producing the intermediate image. No work need be done for orthographic projection because of the simple 4D viewing model; the fourth coordinate is simply ignored. The resulting set of vertices, along with the edge and polygon descriptions, now describes a 3D object. A standard 3D modeling and viewing transformation is then placed on the stack, and the vertices and polygons sent down the pipeline to be drawn, using the pipeline as it was intended.

Shading

If a workstation has hardware support for shaded polygons, the sticky question arises of how to shade these hyperobjects. One approach to shading hyperobjects is to view the intermediate image as a 3D object shaded by a light source. While this allows standard rendering algorithms to be used, the surface normals have to be continually recomputed, because the intermediate image is continually deforming as the hyperobject rotates in four dimensions.

Furthermore, in 4-space a hyperplane, and not a polygon, determines a unique perpendicular vector, so the surface normal of a polygon is not well-defined. Carey, et. al., [CBC87] take the interesting approach of shading hyperfaces with a 4D light source, taking advantage of the fact that the geometrical criteria used in simple lighting models, i.e., surface normals and inner-product of vectors, generalize to hyperfaces and 4D vectors.

We have adopted a different shading rule that provides a more direct visual correlation to the fourth dimension and which is better suited to these workstations, namely 4D depth cueing. While standard depth cueing relates the color intensity to the z value of a particular pixel, 4D depth cueing simply relates the intensity to the w value. This is most effective with Gouraud shading, mapping the w coordinate of each vertex into a colormap value. For example, if all 4D coordinates lie in the range $[wmin, wmax]$ at all times, then the expression $\frac{(w-wmin)}{(wmax-wmin)}$ maps all w values into the range $[0,1]$, suitable for a color value. With a monotonically decreasing intensity ramp, points closer in 4-space appear brighter while points further away appear dimmer. As objects rotate in 4-space, this technique can help the viewer track the action. See Figures 1 and 2 for 4D depth-cued images of hypercubes under orthographic and perspective projection respectively.

This is simply an image space variant of a well known technique for clarifying the fourth dimension. Banchoff [Ban78] describes a film with "water-level slicing", where a hyperplane intersection is in one color, everything above the hyperplane is another, and everything below is a third color.

Hyperplane Clipping

Just as slicing a 3D object with a plane produces surprising cross sections, so does slicing a hyperobject. Rather than finding the exact intersection of a 4D object with an infinitesimally thin hyperplane, we have implemented "near" and "far" clipping hyperplanes, analogous to those used commonly in 3D graphics. In general, an n dimensional space is divided into two halves by an $n-1$ dimensional space. In four dimensions a hyperplane, a single 3-space, divides 4-space into two halves, just as a plane divides 3-space and a line divides a plane. Using two hyperplanes perpendicular to the w axis, corresponding to the equations $w = near$ and $w = far$, allows for an efficient implementation. Given two points $P_0 = (x_0, y_0, z_0, w_0)$ and $P_1 = (x_1, y_1, z_1, w_1)$, the line segment $\overline{P_0P_1}$ can be written

$$P = (1-a)P_0 + aP_1, \qquad 0 \leq a \leq 1.$$

The choice of clipping hyperplanes allows the visibility of points to be determined simply with $w \geq near$ and $w \leq far$. If the endpoints are on opposite sides of a hyperplane $w = k$, then the value of a for the intersection point is defined by

$$a = \frac{(k - w_0)}{(w_1 - w_0)}$$

This is an adaptation of a standard parametric form used in many clipping algorithms, e.g., Sutherland and Hodgman [SH74].

We have implemented a simple software 4D "depth" clipper, that clips to the hyperplanes described above. Its performance turns out to be quite satisfactory. The clipper is derived from the Sutherland and Hodgman polygon clipping, chosen for its simplicity and extensibility [SH74].

The original algorithm clips against only one limiting plane at a time, doing each in turn, in a simple pipelining system; our version clips against one limiting hyperplane at a time. Thus, the modification is straightforward, replacing the clipping tests and intersection equations with the 4D equations described above. Furthermore, clipping against more general sets of hyperplanes can be implemented easily by adapting the tests and adding more stages to the clipping pipeline.

This clipper can be used in two different places in the rendering pipeline, either before or after the 4D modeling transformations. Although the two configurations are equivalent in that every image produced by one can be produced by the other, different effects occur as the object rotates or the clipping hyperplanes are moved.

If the clipper is placed before the 4D modeling transformation, the object will only be clipped in one orientation in 4-space, equivalent to clipping an object only with planes perpendicular to one axis in the object's coordinate system. To obtain slicing along an arbitrary axis, it is necessary to perform a one-time 4D rotation of the object to align the clipping axis with the w-axis. The resulting vertex set can be used instead of the original one to explore the effect of clipping hyperplanes along that axis.

Alternatively, when the object is rotated in 4-space before clipping, the clipping planes remain in a single orientation and the object is continually sliced in different ways, just as a rotating 3D object might be clipped in different ways by the traditional near and far clipping planes. See Figures 3, 4, and 5 for three different clippings of a hypercube.

The clipper is reasonably efficient, requiring 8 adds and 4 multiplies for every clipped point, and only one comparison for unclipped points. However, some performance degradation is seen, since the algorithm is $O(n)$ in terms of the number of vertices. We return to 4D depth cueing to find an efficient, constant-time method to approximate slicing. If the graphics architecture implements Gouraud shading by interpolating color map indices, then 4D depth cueing ensures that all pixels in a common hyperplane have the same color index. Note that the converse is not true, i.e., not all pixels with the same color index come from exactly the same hyperplane, but from a (hopefully) small range of hyperplanes along the clipping axis. Highlighting a particular color entry or band of entries causes a particular "thick" slice to become noticeable, although the parts of the object not in that band will still be visible. Often highlighting a band of entries will be necessary due to aliasing problems. For example, if the color indices range from 0 to 100, a polygon that covers this range but scans to a width of 10 pixels will have approximately a ten color value jump from pixel to pixel.

Thus, color map manipulation can be a useful tool, especially if employed as a "previewer" to locate quickly a particular slice before employing the more expensive clipping algorithm. See Figure 7 for a 4D object with a highlighted hyperplane slice.

Hidden Surfaces and Hollow Polygons

Contrary to our expectations, filled and shaded polygons with hidden surface removal had severe drawbacks when used to view hyperobjects. Two major problems occurred, first that 4D datasets often clash with the Z-buffer algorithm, and second that much of the interesting structure of the projection of a 4D object is obscured by the hidden surface elimination.

The workstations we use all employ a hardware Z-buffer for fast hidden surface removal. Though simple to use, the Z-buffer algorithm can alias badly when two or more polygons lie approximately in the same plane. Unfortunately, polygons that are disjoint in 4-space often project to polygons that are nearly identical in 3-space, so this problem occurs frequently. Short of adding checks to eliminate coplanar polygons, we have found no real-time solution to this problem.

The second problem, that of decreased visibility, is especially true for polytopes such as the hypercube; what was a fascinating line drawing becomes a dull set of quadrilaterals. Hyperplane clipping provides one means of exploring the interior of the 3D projection, but obviously this precludes viewing the *whole* structure. To retain the nice shading that comes with filled polygons, and yet not obscure the internal structure so markedly, we employ the simple technique of cutting the center out of the polygons.

For convex polygons, scaling down by s about the center of mass and then subtracting the result from the original polygon leaves a border of some width, if $0 < s < 1$. This hollow polygon is easily described by a sequence of quadrilaterals $(P_i, P_{i+1}, P'_{i+1}, P'_i)$ for all adjacent vertices i and $i + 1$, including the first and last pair. Adjusting the parameter s controls the border width, giving wire-frame at one extreme and solid polygons at the other. Intermediate values of s give shaded images so that both 3D and 4D depth perception is easy, and yet all facets of the object can be seen, as in Figures 1 and 2. If the workstation supports smooth-shaded lines, a wire-frame rendering can also have depth cueing. While this method is simpler than the hollow polygons, it retains the inherent ambiguity in wire-frame drawings that the depth ordering of two crossing lines is hard to determine visually.

Other Techniques

We mention briefly other graphics techniques which are useful in exploring four dimensions, but which are well documented elsewhere. Transparent polygons implemented by stippling, if available on the workstation, could be used instead of hollow polygons for adding clarity, but a finite number of stippling patterns limits the usefulness of this technique for complicated objects.

Stereo pairs, calculated from the intermediate image, can aid a viewers comprehension tremendously, especially when coupled with prodigious use of the 3D image transformations; these projections often have surprising characteristics when rotated in 3-space. Implementations on slower workstations can capitalize on this by generating several frames of 3D transformations for each 4D transformation, each additional 3D frame requiring only one pass through the hardware pipeline instead of two. This results in better response time and smoother animation.

Applications

Besides considering the regular polytopes, such as the hypercube pictured in Figures 1-5, we have also explored polytopes of importance in Information-Based Complexity Theory, one of which is pictured in Figures 6-8.

This 4D object, which represents a unit ball in a Banach space, is used as a model of computational complexity. Specifically, the "radius" of 3D slices of this object, defined here as a geometrical measure of polygons and polyhedra, provide a measure of "adaption". This object has the rare property that the radius is larger for slices at two opposite extremities than for a slice from the middle; no object with a lower dimension has this property [KN88].

Figure 6 shows a 4D depth cued version of this unit ball. Figure 7 shows another view with a hyperplane intersection highlighted by color map manipulation. Figure 8 shows one interesting slice of this ball.

Work in progress involves studying a space-time model of animal and human skull growth using 3D skull data changing over time. The goal is to explore the geometry of growth curves and structure; the better the understanding of the growth process, the better techniques such as reconstructive surgery can be made [SM87].

Implementation

We have implemented this 4D animation system on the Hewlett-Packard 9000 Series 300 workstations with 98710 and 98721 hardware graphics accelerators, as well as the Silicon Graphics IRIS 1400 and 2400 workstations. Response for small objects, e.g., a hypercube, can be as fast as 10 frames per second, while larger objects with approximately 400 vertices can slow to a second or more per frame.

Conclusions

We have shown that real-time 4D animation is possible using real-time 3D graphics workstations. As expected, the implementation has proven useful in exploring both old and new 4D polytopes and surfaces, functioning as a mathematical educational tool, and as a research tool. Beyond simply studying pure geometry, we are now applying it to scientific data.

Acknowledgements

We would like to extend our thanks to Tom Rosenfeld who co-authored the original implementation, Linda Altounian and David Radvany who have worked on the most recent version, and to Mark Kon for his interest and applications.

References

[Abb52] Edwin A. Abbott. *Flatland.* Dover Publications, Inc., New York, 1952.

[Ban78] Thomas F. Banchoff. Computer animation and the geometry of surfaces in 3- and 4-space. In *Proceedings of the International Congress of Mathematicians*, pages 1005–1013, 1978.

[Ban87] Thomas F. Banchoff. Using computer graphics to explore the generation of surfaces in four dimensional space. Discovering the Fourth Dimension, published by Prime Computer, Inc., 1987.

[CBC87] Scott A. Carey, Robert P. Burton, and Douglas M. Campbell. Shades of a higher dimension. *Computer Graphics World*, 93–94, October 1987.

[Cla82] James H. Clark. The geometry engine: A VLSI geometry system for graphics. *Computer Graphics*, 16(3):127–133, July 1982.

[Cox69] H.S.M. Coxeter. *Introduction to Geometry.* John Wiley and Sons, Inc., New York, 2 edition, 1969.

[Dew86] A.K. Dewdney. Computer recreations. *Scientific American*, 14–23, April 1986.

[FSB82] Steven Feiner, David Salesin, and Thomas Banchoff. Dial: A diagrammatic animation language. *IEEE Computer Graphics and Applications*, 43–54, September 1982.

[Gar77] Martin Gardner. *Mathematical Carnival, Chapter 4: Hypercubes.* Vintage Books, New York, 1977.

[KN88] Mark Kon and Erich Novak. Bounds for the adaption problem in complexity theory. Department of Mathematics, Boston University, 1988.

[Nol67] Michael A. Noll. A computer technique for displaying n-dimensional hyperobjects. *Communications of the ACM*, 10(8):469–473, August 1967.

[SH74] Ivan E. Sutherland and Gary W. Hodgman. Reentrant polygon clipping. *Communications of the ACM*, 17(1):32–42, January 1974.

[SM87] Richard Skalak and Melvin Moss. The analysis of growth and form program project. 1987.

[Som58] D.M.Y. Sommerville. *An Introduction to the Geometry of N Dimensions.* Dover Publications, Inc., New York, 1 edition, 1958.

[ST86] Roger W. Swanson and Larry J. Thayer. A fast shaded-polygon renderer. In *SIGGRAPH '86 Proceedings, published as Computer Graphics*, pages 95–101, August 1986.

[vDSH74] A. van Dam, G. M. Stabler, and R. J. Harrington. Intelligent satellites for interactive graphics. In *Proceedings of the IEEE*, pages 483–492, April 1974.

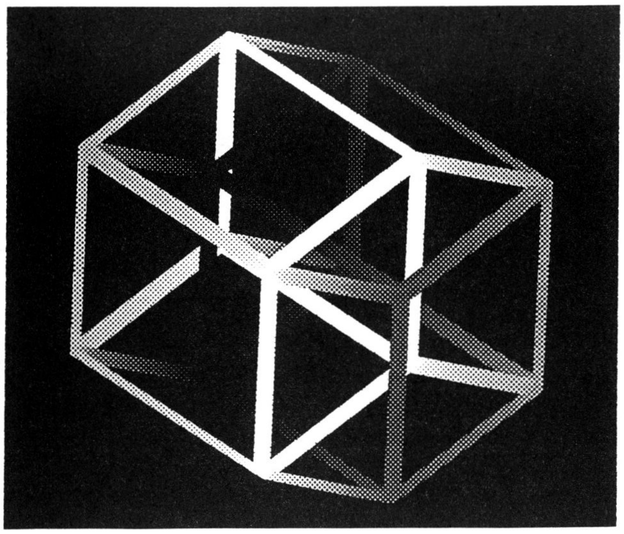
Figure 1

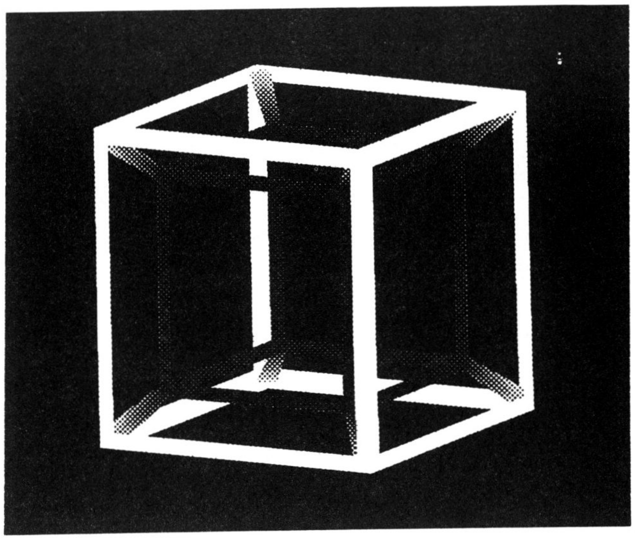
Figure 2

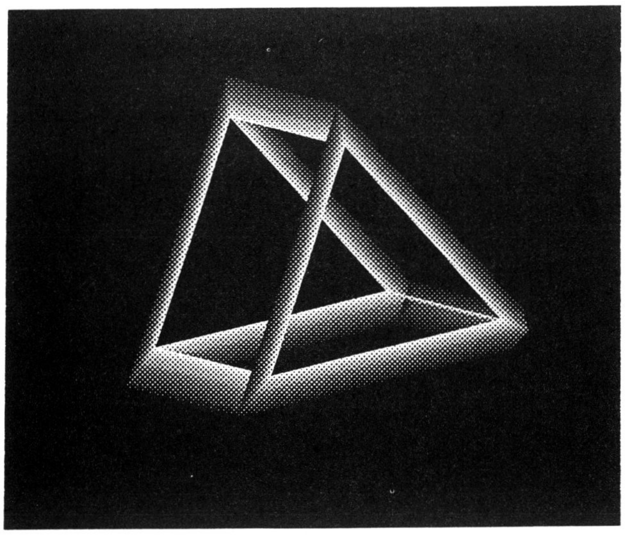
Figure 3

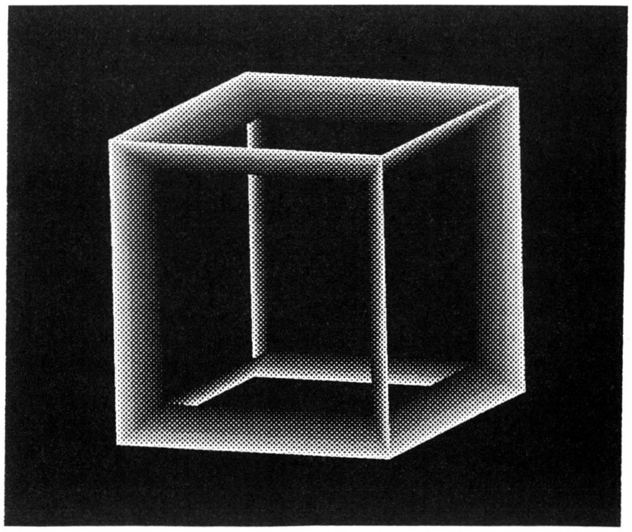
Figure 4

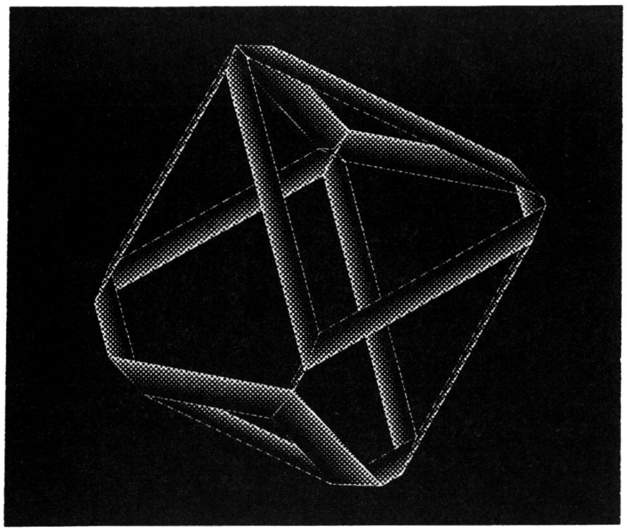

Figure 5

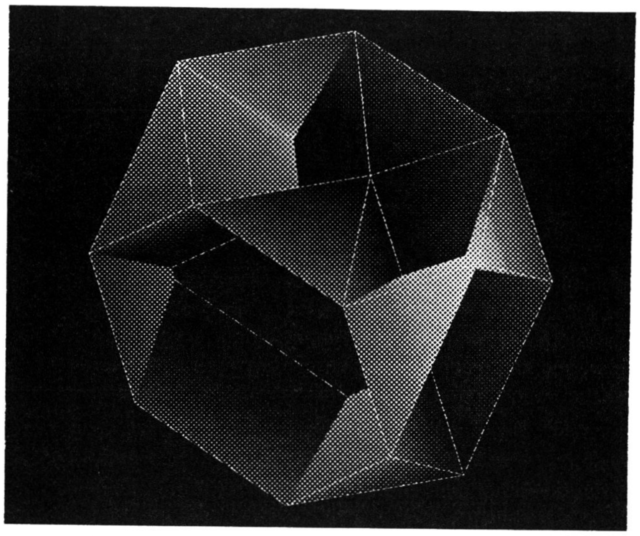

Figure 6

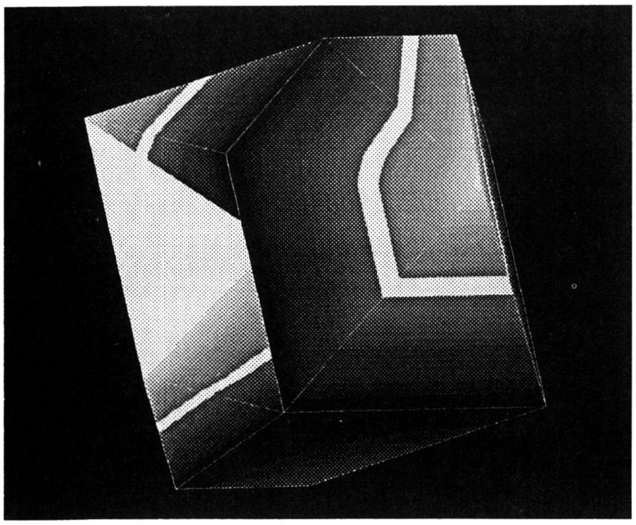

Figure 7

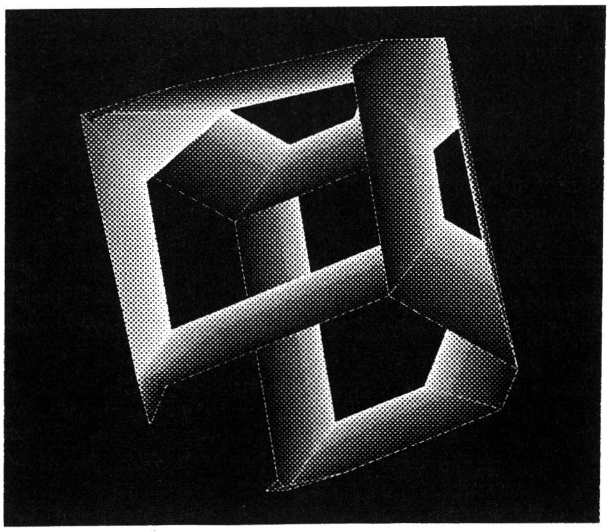

Figure 8