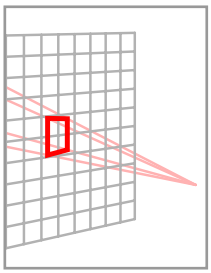
CHAPTER 13

PostScript in 3D



In this chapter I'll explain a 3D extension to PostScript that I call `ps3d`, which you can find in the file `ps3d.inc`. In order to make this extension available, just put a line `(ps3d.inc) run` at the top of your program, once you have downloaded `ps3d.inc`. The 3D graphics environment you get in this way is as close to the usual 2D one as I could make it, with a few unusual features I'll mention later.

The underlying computations involved in 3D drawing are much more intensive than that for 2D, and in view of that it was surprising to me that package `ps3d` has turned out to be acceptably efficient. As I have mentioned already, I have used the usual PostScript routines in 2D as a model, except that instead of being restricted to affine transformations, and therefore in this case to matrix arrays of size $3^2 + 3 = 12$, the underlying code works with arbitrary homogeneous 4×4 matrices, or arrays of size 16. There are several reasons for doing this, among them that it makes the final perspective rendering simpler. It also makes it possible to cast shadows easily. But the principal one, I have to confess, is mathematical simplicity. The disadvantage might be slowness, but although noticeable in some circumstances it doesn't seem to be a fatal problem. This whole package owes much to Jim Blinn's book **A Trip Down the Graphics Pipeline**, mostly for the rigorous use of homogeneous coordinates throughout. On the other hand, using normal functions instead of just normal vectors is something only a mathematician would suggest happily. The advantage of doing this is that non-orthogonal transformations can be handled, although this is perhaps only a theoretical advantage, since most coordinate changes are in fact orthogonal.

Still following graphics conventions (i.e. as opposed to mathematical ones) point-vectors are rows, and matrices are applied to them on the right. Affine functions $Ax + By + Cz + D$ will be expressed as column vectors. Matrices multiply them on the left, and evaluation of such a function is a matrix product

$$Ax + By + Cz + D = [\, x \quad y \quad z \quad 1 \,] \begin{bmatrix} A \\ B \\ C \\ D \end{bmatrix} .$$

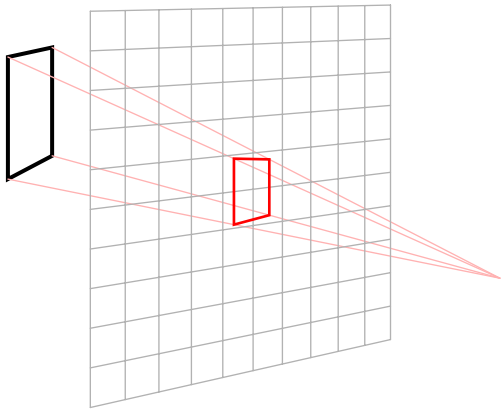
I recall that composition on the stack is the principal motivation for these conventions of order in PostScript programming. Thus $x\,S\,T$ conveniently applies S to x and then T. This conforms to the usual PostScript convention of applying operators to the object operated on so as to be ready to apply the next operator.

In this package, four dimensional directions are expressed in homogeneous coordinates as $[x, y, z, w]$. Such a point with $w \neq 0$ can be identified with the 3D point $(x/w, y/w, z/w)$, and one with $w = 0$ can be identified with a non-oriented direction in 3D. There is some inconsistency involved in this last point, because sometimes the package implicitly uses oriented directions.

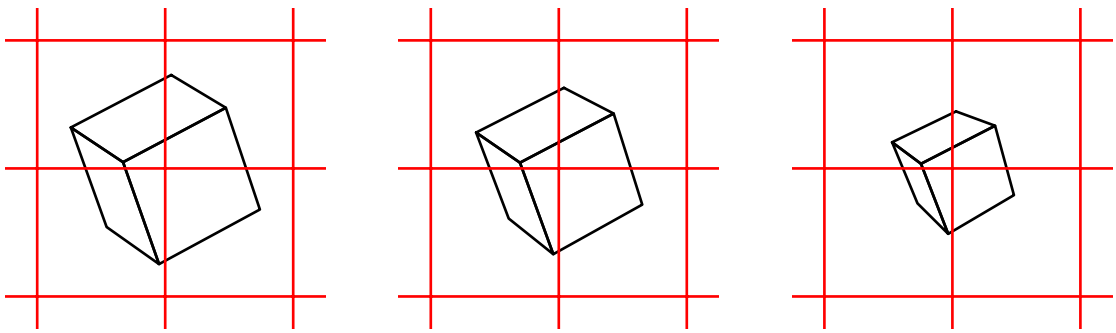
13.1. A survey of the package

In beginning 3D drawing, after loading the file `ps3d.inc`, the programmer has the option of choosing a position for his eye, by convention somewhere on the positive z-axis. He is then looking down towards the negative z-axis, and what he sees is the projection of objects upon his viewing screen, which is the plane $z = 0$. On this plane x and y are the usual 2D user coordinates. This projection of a point onto the viewing screen is the intersection of the line from that point to the observer's location with the plane $z = 0$. It is expected that most objects drawn will be located with $z < 0$.

If the programmer does not choose an eye location, the default mode of projection is orthogonal projection onto the plane $z = 0$, so that the eye is effectively at ∞ on the z-axis, the 4D point $[0, 0, 1, 0]$. The point of the convention that we are looking down (rather than up) the z-axis is to get the 3D orientation to be that of the right hand rule as well as to have the usual 2D coordinates interpreted naturally. I recall that, according to the right hand rule, if the right fist is curled so as to indicate how the positive x-axis is to be rotated towards the positive y-axis, then the extended thumb points towards positive values of z.



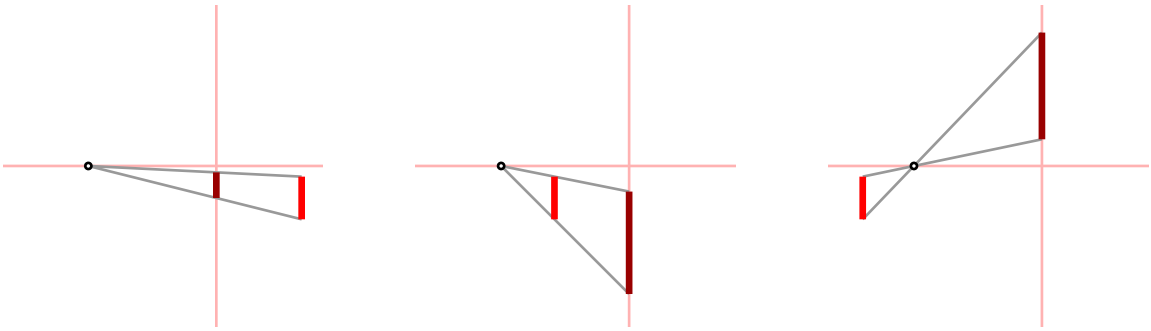
This setup has some paradoxical features. The following figures are identical except for the location of the eye, which is at $z = 27$, $z = 9$, and $z = 3$, respectively:



What's odd here is that as we move closer the cube becomes smaller! The explanation for this is that moving the eye in this model is not like moving your head back to change your view. When you move your head you move the whole eye, which includes both the lens at the front and the retina at the back. The distance from the lens to the retina—the **focal length** of the eye—does not change. Whereas in the graphics model I am using, moving the eye really means changing the focal length as well as the location of the eye. We are looking at the plane $z = 0$ as though it were a window onto which the world in the region $z \leq 0$ is projected, but as we move away our focal length is increased, too. In effect the left picture is taken with a telephoto lens, the right hand one with a wide angle lens. The two changes, location and focal length, compensate for each other, so that the grid, which is located on the plane $z = 0$, remains constant. This behavior might take getting used to. In the current version,

the eye location is set once at the very beginning of a program and should never be changed again. In some future version I'll make it possible to move the eye around like a camera, changing location and also zooming in and out.

There is another odd feature of these conventions. As I have already mentioned, normally the 3D objects drawn will lie in the region $z < 0$, beyond the viewing screen.



Normal viewing *Too close* *Behind the eye*

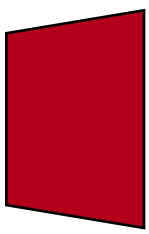
If the object lies between the eye and $z = 0$, it will be enlarged. If it lies behind the eye, it will be placed in an odd location. It is because of this unfamiliar behavior that it is best to draw only objects behind the plane $z = 0$.

The most frequently used commands in `ps3d.inc` are those which construct 3D paths—`moveto3d`, `lineto3d`, `closepath3d` and their relative versions. There are also 3D coordinate changes such as `translate3d`, `scale3d`, and `rotate3d`. Rotations require an axis as well as an angle as argument. As with PostScript in 2D, there are implicitly two 3D coordinate systems. One is the user's 3D coordinate system, and the other the default 3D system, which is also the user's system when 3D programming starts. Part of the 3D graphics environment keeps track of the transformation from the first to the second, by means of a 4×4 matrix. Thus when a program executes a command like `0 0 0 moveto3d` this transformation is applied to the 3D point $(0, 0, 0)$ to get its default 3D coordinates, and then the projection of this point upon the viewing plane is calculated. In this way, a 2D path is constructed as the 3D drawing proceeds. The transformation from user 3D to default 3D coordinates is affected by the 3D coordinate changes. As in 2D, coordinate changes affect directly the frame with respect to which the user coordinates are interpreted. Thus:

```
[0 0 3 1] set-eye

[0 1 0] -45 rotate3d

newpath
-0.5 -0.5 0 moveto3d
 0.5 -0.5 0 lineto3d
 0.5  0.5 0 lineto3d
-0.5  0.5 0 lineto3d
-0.5 -0.5 0 lineto3d
closepath
gsave
0.7 0 0 setrgbcolor
fill
grestore
stroke
```



Just as in ordinary 2D PostScript, we use a graphics stack to manage 3D graphics environments. The 3D graphics environment at any moment amounts to two homogeneous linear transformations T (4×4) and D (4×3), the first transforming user 3D coordinates to the default ones, and that determining the collapse from 3D onto the viewing screen. The graphics stack is an array `gstack3d` of some arbitrary height `gmax`—it happens to be 64 in `ps3d`. But the part of the stack that's used bounces around. The current height of the stack at any moment is `ght` and the current graphics data T and D are in `gstack3d[ght]`. The user will ordinarily not need to access the 3D graphics stack directly, because there are operators that access its components directly. The 3D graphics stack is managed with commands `gsave3d` and `grestore3d`. One use for the 3D graphics stack is to move different components of a figure independently, or even in linkage with other components. (This technique is explained quite nicely in Chapter 3 of Blinn's book.)

The 3D graphics environment really requires only T and D, but also, for efficiency and convenience, the inverse matrix T^{-1} is part of the stored data. Thus an item on the 3D graphics stack is an array of 3 matrices; T, T^{-1}, and D. The first component is called the **current 3D transformation matrix**. The effect of transforming coordinates by a matrix R (rotation, translation, scaling, projection, etc.) is to replace T by RT, T^{-1} by $T^{-1}R^{-1}$. The third component D is called the **display matrix**, used to transform homogeneous 4-vectors to homogeneous 3-vectors. At the moment the only display matrices used are those implementing perspective onto the plane $z = 0$ to an eye located in homogeneous space. Also at the moment it is set once and for all at the beginning of 3D drawing.

We shall see later why we want quick access to T^{-1}.

In Blinn's terms, 3D graphics involves a pipeline: a composition of coordinate changes from user coordinates to default 3D coordinates, then to the display plane (here $z = 0$), finally to the page. The last step is controlled by the usual 2D PostScript commands. The first two are managed in this package.

Loading `ps3d` (with the command `(ps3d.inc) run`) causes a file `matrix.inc` of 3D matrix procedures to be loaded as well. It also causes the variables `gstack3d`, `gmax`, `ght` that determine the 3D graphics environment to be defined. Initially, `ght` is 0, T and T^{-1} are the 4×4 identity matrices, and D amounts to orthogonal projection along the positive z-axis.

In addition `ps3d` defines variables `cpt3d` and `lm3d`, the **current point** and the **last point moved to**, 3D points expressed in default 3D coordinates. Normally, the user will have no need to know them.

The usual 3D vectors are identified with a 4D vector whose last coordinate is 1.

Next I'll explain the commands, grouped by subject.

13.2. The 3D graphics environment

	Arguments	Command	Return value
•	—	`gsave3d`	—

Puts a new copy of the current 3d environment on the graphics stack. It is extremely important to realize that manipulating the 3d graphics stack has no effect whatsoever on the 2D graphics state. And *vice versa*.

•	—	`grestore3d`	—

Restores the previous 3d graphics state.

•	—	`cgfx3d`	$[T\ U\ D]$
•	—	`ctm3d`	T
•	—	`cim3d`	T^{-1}
•	—	`cdm3d`	D
•	—	`currentpoint3d`	$[x\ y\ z]$

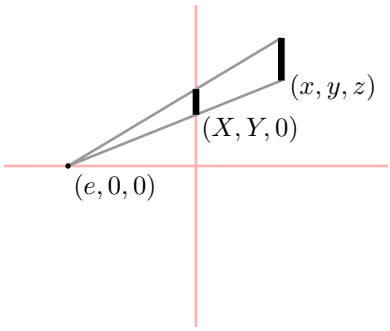
This returns the current point in user space, T^{-1} applied to `cpt3d`.

•	—	`display-matrix`	D

Returns a 4×3 homogeneous matrix.

- $[x\ y\ z\ w]$ `set-eye` —

Sets the eye at (x, y, z, w). In practice, currently it is assumed that $x = y = 0$. Here, as elsewhere in 3D drawing, $w = 0$ indicates a direction in space, whereas $(x, y, z, 1)$ indicates an ordinary 3D point.



The 3D to 2D projection takes a point (x, y, z) to the intersection (X, Y) of the line from that point to the eye with the plane $z = 0$. Assuming the eye is at (e_x, e_y, e_z) we can find it explicitly as

$$(1 - t)(e_x, e_y, e_z) + t(x, y, z)$$

where $(1 - t)e_z + tz = 0$. Therefore

$$\begin{aligned} t &= \frac{e_z}{e_z - z} \\ 1 - t &= \frac{-z}{e_z - z} \\ X = \frac{xe_z - ze_x}{e_z - z}, &\quad Y = \frac{ye_z - ze_y}{e - z}\ . \end{aligned}$$

If the eye is at $(0, 0, e)$ this becomes

$$X = \frac{ex}{e - z}, \quad Y = \frac{ey}{e - z}\ .$$

In terms of homogeneous coordinates it takes (x, y, z, w) to $(xe_z - ze_x, ye_z - ze_y, e_z w - ze_w)$. Equivalently, it sets the display matrix to be

$$\begin{bmatrix} e_z & 0 & 0 \\ 0 & e_z & 0 \\ -e_x & -e_y & -e_w \\ 0 & 0 & e_z \end{bmatrix}$$

- — `get-eye` $[x\ y\ z\ w]$
- — `get-virtual-eye` $[x\ y\ z\ w]$

This last will be explained later.

- $[x\ y\ z\ w]$ `render` $[X/W\ \ Y/W]$

This applies the display matrix to the original point, then turns the result $[X\ Y\ W]$ into a real 2D point in 2D user coordinates.

- $x\ y\ z$ `transform2d` $x'\ y'$

Transforms from user 3D coordinates first to default 3D coordinates, then to 2D user coordinates.

13.3. Coordinate transformations

A coordinate transformation multiplies the current transformation matrix by the matrix of the transformation. The current matrix is an array of 16 numbers, interpreted as

$$\begin{bmatrix} T_0 & T_4 & T_8 & T_{12} \\ T_1 & T_5 & T_9 & T_{13} \\ T_2 & T_6 & T_{10} & T_{14} \\ T_3 & T_7 & T_{11} & T_{15} \end{bmatrix}.$$

Let U be the inverse of T.

- $x\,y\,z$ `translate3d` —

The new current matrix becomes

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ x & y & z & 1 \end{bmatrix} \begin{bmatrix} T_0 & T_4 & T_8 & T_{12} \\ T_1 & T_5 & T_9 & T_{13} \\ T_2 & T_6 & T_{10} & T_{14} \\ T_3 & T_7 & T_{11} & T_{15} \end{bmatrix}$$

and the inverse transformation becomes

$$\begin{bmatrix} U_0 & U_4 & U_8 & U_{12} \\ U_1 & U_5 & U_9 & U_{13} \\ U_2 & U_6 & U_{10} & U_{14} \\ U_3 & U_7 & U_{11} & U_{15} \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -x & -y & -z & 1 \end{bmatrix}$$

- $[x\,y\,z]\,A$ `rotate3d` —

The arguments are axis and angle. The new current transformation matrix becomes

$$\begin{bmatrix} R_0 & R_3 & R_6 & 0 \\ R_1 & R_4 & R_7 & 0 \\ R_2 & R_5 & R_8 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} T_0 & T_4 & T_8 & T_{12} \\ T_1 & T_5 & T_9 & T_{13} \\ T_2 & T_6 & T_{10} & T_{14} \\ T_3 & T_7 & T_{11} & T_{15} \end{bmatrix}$$

and the inverse transformation

$$\begin{bmatrix} U_0 & U_4 & U_8 & U_{12} \\ U_1 & U_5 & U_9 & U_{13} \\ U_2 & U_6 & U_{10} & U_{14} \\ U_3 & U_7 & U_{11} & U_{15} \end{bmatrix} \begin{bmatrix} R_0 & R_1 & R_2 & 0 \\ R_3 & R_4 & R_5 & 0 \\ R_6 & R_7 & R_8 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

since the inverse of a rotation matrix is its transpose.

- $a\,b\,c$ `scale3d` —

The new 3D transformation matrix is

$$\begin{bmatrix} a & 0 & 0 & 0 \\ 0 & b & 0 & 0 \\ 0 & 0 & c & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} T_0 & T_4 & T_8 & T_{12} \\ T_1 & T_5 & T_9 & T_{13} \\ T_2 & T_6 & T_{10} & T_{14} \\ T_3 & T_7 & T_{11} & T_{15} \end{bmatrix}$$

and the inverse transformation

$$\begin{bmatrix} U_0 & U_4 & U_8 & U_{12} \\ U_1 & U_5 & U_9 & U_{13} \\ U_2 & U_6 & U_{10} & U_{14} \\ U_3 & U_7 & U_{11} & U_{15} \end{bmatrix} \begin{bmatrix} a^{-1} & 0 & 0 & 0 \\ 0 & b^{-1} & 0 & 0 \\ 0 & 0 & c^{-1} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- matrix `concat3d` —

The argument is a 3×3 matrix M, i.e. an array of 9 numbers. This replaces T by M (extended) T.

The new 3D transformation matrix is

$$\begin{bmatrix} M_0 & M_3 & M_6 & 0 \\ M_1 & M_4 & M_7 & 0 \\ M_2 & M_5 & M_8 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} T_0 & T_4 & T_8 & T_{12} \\ T_1 & T_5 & T_9 & T_{13} \\ T_2 & T_6 & T_{10} & T_{14} \\ T_3 & T_7 & T_{11} & T_{15} \end{bmatrix}$$

and the inverse transformation is

$$\begin{bmatrix} U_0 & U_4 & U_8 & U_{12} \\ U_1 & U_5 & U_9 & U_{13} \\ U_2 & U_6 & U_{10} & U_{14} \\ U_3 & U_7 & U_{11} & U_{15} \end{bmatrix} \begin{bmatrix} M'_0 & M'_3 & M'_6 & 0 \\ M'_1 & M'_4 & M'_7 & 0 \\ M'_2 & M'_5 & M'_8 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- $[A\,B\,C\,D]\,P$ `plane-project` —

Projects from the 3D homogeneous point P onto a plane. After this change, which is not invertible, paths are projected onto the plane. Explicitly, the map here is $Q \mapsto f(P)\,Q - f(Q)\,P$.

The matrix transformation is

$$\begin{bmatrix} f(P) - AP_0 & -AP_1 & -AP_2 & -AP_3 \\ -BP_0 & f(P) - BP_1 & -BP_2 & -BP_3 \\ -CP_0 & -CP_1 & f(P) - CP_2 & -CP_3 \\ -DP_0 & -DP_1 & -DP_2 & f(P) - DP_3 \end{bmatrix} \begin{bmatrix} T_0 & T_4 & T_8 & T_{12} \\ T_1 & T_5 & T_9 & T_{13} \\ T_2 & T_6 & T_{10} & T_{14} \\ T_3 & T_7 & T_{11} & T_{15} \end{bmatrix}.$$

The inverse transformation is invalid.

- $v = [x\,y\,z\,w]\,M$ `transform3d` vM
- $M\,f$ `dual-transform3d` Mf

13.4. Drawing

- $x\,y\,z$ `moveto3d` —
- $x\,y\,z$ `lineto3d` —
- $x\,y\,z$ `rmoveto3d` —
- $x\,y\,z$ `rlineto3d` —
- $x_1\,y_1\,z_1\ldots$ `curveto3d` —

The input is nine numbers.

- $x_1\,y_1\,z_1\ldots$ `rcurveto3d` —
- — `closepath3d` —

It updates the current 3D point.

- [see below] `mkpath3d` —

The arguments are t_0, t_1, N, an array of parameters, and the function `/f`. This function has two arguments, the array of parameters and a single variable t. It returns an array made up of (1) a 3D point (x, y, z) and (2) a velocity vector $x', y', z')$. This procedure draws the path in N pieces using f as parametrization.

- — `2d-path-convert` —

This converts the current 2D path to a 3D path according to the current 3D environment. Thus

```
[0 0 5 1] set-eye

[0 1 0] 45 rotate3d

newpath
0 0 moveto
(ABC) true charpath
2d-path-convert
gsave
0.7 0 0.1 setrgbcolor
fill
grestore
stroke
```

13.5. Surfaces

I'll deal with surface rendering in much more detail in the next chapter, but make a few preliminary remarks here.

Surfaces are most conveniently interpreted to be an assembly of flat plates, maybe very small ones to be sure. To ensure an illusion of depth in 3D pictures, they are often shaded according to the position of some imaginary light source. In mathematical graphics we are not interested in realistic rendering, which may even clutter up a good diagram with irrelevant junk, and can settle for simple tricks. Still, a few things are important for the right illusion.

- x $[a_0\, a_1]$ `lshade` —
- x $[a_0\, a_1\, a_2\, a_3]$ `shade` —
- x $[a_0 \ldots\, a_n]$ `bshade` —

The number x lies in $[-1, 1]$, and is usually the result of the calculation of the dot-product of normal vector and light direction. It is shifted to $[0, 1]$ and then the weighting function is applied.

The coordinates a_0, etc. are control values for a Bernstein polynomial determining the shading weight. The values of a_0 and a_n are minimum and maximum. The first two are duplicated by the last one, but they are significantly more efficient. For `shade` itself, $[0\ 1/3\ 2/3\ 1]$ would be neutral and $[0\ 0\ 1\ 1]$ would be relatively strong contrast. These things are discussed in detail in the next chapter.

- array of 3D points `normal-function` $[A, B, C, D]$

Returns the normal function $Ax + By + Cz + D$ which is 0 on the points of the array, and increasing in the direction determined by the right-hand rule.

It is in dealing with matters affecting the appearance of surfaces—visibility and shading—that we need T^{-1}. The point x is outside the piece of surface Σ if and only if $x \cdot f_\Sigma \geq 0$, where f_Σ is the normal function associated to S. We often want to know similarly if x is outside the transformed surface ST. This happens if and only if xT^{-1} is outside S, or

$$xT^{-1} \cdot f_\Sigma = x \cdot T^{-1} f_\Sigma \geq 0\ ,$$

from which we see that $f_{\Sigma T} = T^{-1} f_\Sigma$. We could calculate $T^{-1} f_\Sigma$ for each Σ but in practice we shall want to test 'visibility' for many bits of surface Σ and only one x (the 'eye', say). So we must be ready to calculate $T^{-1}x$. This, and similar calculations involving the light source, motivate the extra matrix T^{-1} on the graphics stack. The command `get-virtual-eye` returns this point.

13.6. Code

The `ps3d` package is in `ps3d.inc`. This incorporates a collection of generic matrix routines, imported from `matrix.inc`. This package is documented separately in `matrix.pdf`. Sample 3D drawing is contained in the files `cube-frame.ps`, `cube-solid.ps`, `cube-shaded.ps`, and `cube-shadow.ps`. The last two of these use techniques discussed in the next chapter.

References

1. Modern perspective was discovered in the early 15th century, and early accounts are still of interest. Leon Alberti wrote the first treatise on it in 1435, and what he said cannot be bettered for succinctness: '. . . whoever looks at a picture sees a certain cross section of a visual pyramid.' (This is quoted from p. 209 of **A Documentary History of Art**, edited and translated by Elizabeth Holt, published by Doubleday Anchor Books.)

2. Jim Blinn, **Jim Blinn's Corner—A Trip Down the Graphics Pipeline**, Morgan Kaufmann, San Francisco, 1996. Chapter 8 of this book examines interesting questions about allowing not only a change in 3D coordinates, but also ways in which the display matrix can be changed to conform to different ways of viewing 3D space. In mathematical graphics, this isn't so important as it is in non-mathematical graphics, where one often has to track a moving object.