Chapter 10. A 3D extension of PostScript

Here is a list of commands included in the file **ps3d.inc**. You should install this file in your working directory and include it in a program with the sequence **(ps3d.inc) run** whenever you want to use these commands.

We need to understand a bit about the environment this package uses. Whatever we draw in 3D has to be rendered onto a 2D page. The basic mechanism for doing this will be to project from a location onto the plane $z = 0$ through a projection into a fixed point on the positive z-axis called the **origin**. This point can actually be at infinity, in which case the projection is just along lines parallel to the z-axis. The way we are doing things, weare assuming a **right-handed coordinate system**, looking down the **negative** z-axis, with the x and y axes in the usual place and facing in the usual direction. (This is in contrast to many computer graphic systems, in which you are looking up the positive z axis. But then you must either be working in a left-handed coordinate system or change the way the x and y axes point. The computer graphics people made a mistake, because the whole rest of the world applies the right hand rule for calculating orientation, and has done so for several hundred years. This is a more confusing mistake than the way in which many systems work with row rather than column vectors.)

The origin will thus always be at a point $(0, 0, a)$ with $a > 0$, or at infinity in the direction of the positive z-axis. Both cases can be handled in a single manner by working with **homogeneous coordinates**. We have already seen a suggestion of this in 2D, where we explained some of PostScript's conventions by embedding each 2D point (x, y) into 3D by putting it at $(x, y, 1)$. The new scheme is a variation of this: we will first of all embed 3D points (x, y, z) into 4D by placing it at $(x, y, z, 1)$. But it will turn out to be extremely useful to think of arbitrary points (x, y, z, w) in 4D as relating to points in 3D. We do this by assuming that two points (x, y, z, w) and (x_*, y_*, z_*, w_*) are equivalent if one is just a non-zero multiple of the other. So if $w \neq 0$ the point (x, y, z, w) will be equivalent to the 'ordinary' 3D point $(x/w, y/w, z/w, 1)$. But if $w = 0$ then the 'point' it corresponds to will be a kind of point at infinity, in the direction of (x, y, z). This may seem like an unnecessary complication at this moment, but being able to work with homogeneos coordinates and allowing us to think of directions as a generalization of points will allow an enormous simplification in calculations. You'll see some examples later on.

One immediate consequence is that just as in 2D, we can think of affine transformations in 3D as coming from 4×4 matrices of a special sort:

$$\begin{matrix} a_{1,1} & a_{1,2} & a_{1,3} & x_1 \\ a_{2,1} & a_{2,2} & a_{2,3} & x_2 \\ a_{3,1} & a_{3,2} & a_{3,3} & x_3 \\ 0 & 0 & 0 & 1 \end{matrix}$$

Here the matrix A is the **linear component** and the column vector x the **translation component**.

• The command **set-display** establishes where the origin is. It has as its only argument the 4D location of your origin. Your choices are restricted to **[0 0 a 1]** or **[0 0 1 0]**. If $a < 0$ you will get peculiar effects (about like standing on your head). If $a = 0$ you will be in serious trouble. *There is no check on the value of a, so it is your responsibilty to get it right.* The reason for the name of the command is that it sets the way in which 3D objects are displayed in 2D. You should set the display very near the beginning of any program in which you are drawing in 3D. But if you don't set it explicitly the origin will be **[0 0 1 0]** by default.

• The command **ctm3d** places on the stack the current 4×4 transformation matrix which converts your 3D user coordinates to the starting system before projecting them onto the (x, y)-plane. But there is a small trick—it will returnanarray of two 4×4 matrices. One of them is the transformation matrix, the other its inverse. So the get the current one you must put **ctm3D 0 get**. To get its inverse you put **ctm3d 1 get**. What you get from either is an array of size 16 on the stack, from which you read off the **rows** of the matrix:

$$\begin{matrix} t_0 & t_1 & t_2 & t_3 \\ t_4 & t_5 & t_6 & t_7 \\ t_8 & t_9 & t_{10} & t_{11} \\ t_{12} & t_{13} & t_{14} & t_{15} \end{matrix}$$

Why 16 elements, when in 2D a PostScript matrix has only 6 rather than 9 elements? Because the transformations we want to apply include perspective projection onto planes, which is not a linear map, but can be obtained from linear transformations on homogeneous coordinates.

• **origin** returns the current origin, an array of 4 elements.

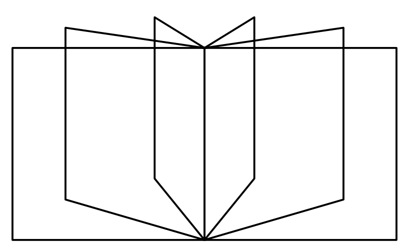
• **moveto3d** has three arguments, and starts a 3D path.

• **lineto3d** is similar.

• **curveto3d** has nine arguments, or three 3D points. You use these in combination with the ordinary commands **newpath**, **stroke** etc. to draw.

Here, for example, is a complete program which draws a rotating square:

```
%!

72 dup scale
0.01 setlinewidth
1 setlinecap
1 setlinejoin

(ps3d.inc) run

[0 0 5 1] set-display

% makes a unit square

/mksquare {
0 0 0 moveto3d
1 0 0 lineto3d
1 1 0 lineto3d
0 1 0 lineto3d
closepath
} def

0 -2 0 translate3d

6 {
  newpath
  mksquare
  stroke
  [0 1 0] 36 rotate3d
} repeat
```



I have placed a grid in front of it so you can measure things. You can see already that drawing in 3D is more complicated than it is in 2D simply because there is more room (an extra dimension!) in which to draw.

• We can change 3D coordinates with simple commands **translate3d** and **scale3d**. We can also rotate coordinates with a command **rotate3d**. It has two arguments, an array of three elements giving the axis of rotation and its orientation, and an angle. To go with these we have commands **gsave3d** and **grestore3d** which save and restore 3D coordinate systems. These commands have nothing to do with the 2D versions. You should realize that in processing the drawing commands in 3D there are three stages, which are essentially independent of each other: (1) application of the 3D transform matrix; (2) flattening points in 3D to ones in the (x, y)-plane; (3) drawing points in this plane on your screen. Each step is essentially independent of the other. For example **scale3d** will have no effect on line widths. In animations, you must still invoke **gsave/grestore** on each page, but what you do about **gsave3d/grestore3d** depends on what effect you want to create.

• **`transform3d`** has two arguments, a 4D vector v and an array of 16 elements representing a 4×4 matrix T , and returns Tv. **`dual-transform3d`** calculates vT, assuming v is a row vector.

• There are a few utilities **`dot-product`** and **`length3d`** with 3D vectors as arguments. Also **`normalize`** which returns a unit vector in the same direction.

• Finally, there is **`plane-project`**. It has two arguments, an array of four elements **`[A B C D]`** representing a plane $Ax + By + Cz + D = 0$ and a point **`[x y z w]`**. It changes the coordinate system by smashing everything down onto the plane according to perspective projection through the point. This is a simple way to create shadows, with the point as light source.

• There are also procedures **`mkpath3d`** to draw 3D parametrized curves, and **`shade`** to calculate shading. Also some routines to test visibility of surfaces, and to construct a surface as a collection of polygons from its parametrization.. I'll explain those later.