The package `ps3d.inc`

The package `ps3d` is rather complicated, but—considering how complicated—surprisingly efficient. I have used the usual PostScript routines in 2D as a model, except that in dealing with homogeneous coordinates I use 4×4 matrices. The principal advantage of this is that it allows perspective transformations such as shadow projection. The disadvantage is slowness, but it doesn't seem to be a fatal problem. This package owes much to Jim Blinn's book **A trip down the graphics pipeline**, mostly for the rigourous use of homogeneous coordinates throughout. On the other hands, using normal functions instead of just normal vectors is something only a mathematician would do 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 orthogonal.

Following graphics conventions (not mathematical ones) point-vectors are rows, and matrices are applied on the right. Function-vectors are columns, and matrices multiply them on the left. Composition on the stack is the motivation for this convention. Thus $x\,A\,B$ conveniently applies A to x and then B. 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.

The 3D graphics environment

Just as in ordinary 2D PostScript, we use a graphics stack to manage 3D coordinates. The graphics stack is an array `gstack3d` of some arbitrary height—here `gmax` is 64. The current height is `ght`, and the current graphics data are in `gstack3d[ght]`.

An item on the graphics stack is an array of 3 matrices; T, T^{-1}, and D. The 4×4 matrix T is the current matrix transforming the user's coordinates into the default ones. The effect of transforming coordinates bny 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 the 4×3 **display matrix**, used to transform homogeneous 4-vectors to homogeneous 3-vectors. At the moment the only display matrices allowed are those implementing perspective onto the plane $z = 0$ to an eye located in homogeneous space. Also at the moment no operation except an initial one affects it.

Why do we need T^{-1}? For matters affecting the appearance of surfaces—visibility and shading. The point x is outside the piece of surface S if and only if $x \cdot f_S \geq 0$, where f_S 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

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

from which we see that $f_{ST} = T^{-1} f_S$. We could calculate $T^{-1} f_S$ for each S but in practice we shall want to test 'visibility' for many bits of surface S 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.

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 PostScript commands. The first two are managed in this package.

Loading `ps3d` (with the command `(ps3d.inc) run`) also causes `matrix.inc` to be loaded.

Also, loading the `ps3d` package causes the variables `gstack3d`, `gmax`, `ght`, which determine the 3D graphics environment to be defined. Initially, ght is 0, T and T^{-1} are the 4×4 identity matrix, and D amounts to orthogonal projection along the positive z-axis. Note that in order to have orientation positive, the eye should be located where $z > 0$. The default display has the eye at ∞ on the z-axis, looking in the negative z-direction.

In addition `ps3d` defines variables `cpt3d` and `lm3d`, the **current point** and the **last point moved to**. These are both 3-vectors.

Normally, 3 vectors are identified with a 4-vector whose last coordinate is 1.

• `gsave3d`

Puts a new copy of the current 3d environment on the graphics stack. Manipulating the 3d graphics stack has no effect whatsoever on the 2D graphics state. *This is extremely important for animations created by turning pages!*

• `grestore3d`

Restores the previous 3d graphics state.

• `ctm3d`

Input: —
Output: the array T, T^{-1}, D.

• `currentpoint3d`

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

• `display-matrix`

Input: —
Output: D

• `set-eye`

Input: $[x\ y\ z\ w]$
Output: —

$$\begin{bmatrix} -z & 0 & 0 \\ 0 & -z & 0 \\ x & y & w \\ 0 & 0 & -z \end{bmatrix}$$

gives perspective onto point in $z = 0$ plane from $[x\ y\ z\ w]$

• `get-eye`

Input: —
Output: $[x\ y\ z\ w]$

Transformations

A coordinate transformation multiplies the current transformation matrix by the matrix of the transformation. The current matrix is

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

and the inverse T'.

- `translate3d`

Input: x, y, z
Output: —

The matrix transformation is

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

and the inverse transformation is

$$\begin{bmatrix} T'_0 & T'_4 & T'_8 & T'_{12} \\ T'_1 & T'_5 & T'_9 & T'_{13} \\ T'_2 & T'_6 & T'_{10} & T'_{14} \\ T'_3 & T'_7 & T'_{11} & T'_{15} \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -x & -y & -z & 1 \end{bmatrix}$$

- `rotate3d`

Input: axis α and angle A in degrees
Output: —

The matrix transformation is

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

and the inverse transformation is

$$\begin{bmatrix} T'_0 & T'_4 & T'_8 & T'_{12} \\ T'_1 & T'_5 & T'_9 & T'_{13} \\ T'_2 & T'_6 & T'_{10} & T'_{14} \\ T'_3 & T'_7 & T'_{11} & T'_{15} \end{bmatrix} \begin{bmatrix} R_0 & R_1 & R_2 & 0 \\ R_3 & R_4 & R_5 & 0 \\ R_6 & R_7 & R_8 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- `scale3d`

Input: a, b, c
Output: —

The matrix transformation 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 is

$$\begin{bmatrix} T'_0 & T'_4 & T'_8 & T'_{12} \\ T'_1 & T'_5 & T'_9 & T'_{13} \\ T'_2 & T'_6 & T'_{10} & T'_{14} \\ T'_3 & T'_7 & T'_{11} & T'_{15} \end{bmatrix} \begin{bmatrix} a^{-1} & 0 & 0 & 0 \\ 0 & b^{-1} & 0 & 0 \\ 0 & 0 & c^{-1} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

• `concat3d`

Input: matrix M
Output: —

It replaces T by M (extended) T.

The matrix transformation 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

$$\begin{bmatrix} T'_0 & T'_4 & T'_8 & T'_{12} \\ T'_1 & T'_5 & T'_9 & T'_{13} \\ T'_2 & T'_6 & T'_{10} & T'_{14} \\ T'_3 & T'_7 & T'_{11} & T'_{15} \end{bmatrix} \begin{bmatrix} M'_0 & M'_3 & M'_6 & 0 \\ M'_1 & M'_4 & M'_7 & 0 \\ M'_2 & M'_5 & M'_8 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

• `reflect3d`

• `affine-reflect3d`

Input: $[A\ B\ C\ D]$ and P
Output: —

The projective reflection.

• `plane-project`

Input: plane (A, B, C, D) and point P
Output: —

Projects from P onto the plane. The map here is $Q \mapsto f(P) * Q - f(Q)P$. It is idempotent.

It replaces T by M (extended) T.

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.

• `transform3d`

Input: v and M
Output: vM

• `dual-transform3d`

Input: M and v
Output: Mv

Drawing

• `moveto3d`

Input: x, y, z
Output: —

• `lineto3d`

Input: x, y, z
Output: —

• `rmoveto3d`

Input: x, y, z
Output: —

• `rlineto3d`

Input: x, y, z
Output: —

• `curveto3d`

Input: x_1, y_1, z_1, x_2, y_2, z_2, x_3, y_3, z_3
Output: —

• `closepath3d`

The principal advantage of using it is that it updates the current 3D point.

• `mkpath3d`

Input: [parameters], $/f$, t_0, t_1, N
Output:

Draws the path in N pieces using f as parametrization.

• `plainarrow3d`

Input: shaft width, arrow length

• `simple-convert`

Input: a 2D PostScript path
Output: —

This converts the path to a 3D path. One of the principal reasons to use it is to make fonts three-dimensional. (See 'flying fonts'.)

Surfaces

• `shade`

Input: $[A\ B\ C\ D\ ]$ and x
Output: —

The coordinates A, etc. are control values for a cubic function giving the shading weight—the array [0 1/3 2/3 1] would be neutral and [0 0 1 1] would be very strong contrast. The values of A and D are minimum and maximum. 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.

• `is-outside`

Input: normal function + eye
Output: true or false

Normally a face of a surface is a pair p and n, where p is a pair of 3D points and n is the normal function calculated for p (anti-clockwise orientation). Thus a face can be tested for visibility by unloading the face, then running through the array. Or unloaded, and an extra copy of n put on the stack to calculate shading if the face is visible.

- `normal-function`

Input: an array of 3D points
Output: 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.