Mathematics 308 — Geometry

Chapter 9. Drawing three dimensional objects

In this chapter we will see how to draw three dimensional objects with PostScript. The task will be made easier by a package of routines you can include in your programs.

1. What is a drawable object?

We shall first set up a number of conventions about exactly what we are drawing, and how we draw it. There are three stages to the process: (1) we work with a fixed object, specified once and for all in a given location; (2) we shall apply various rigid transformations to it; (3) we draw the transformed object on the page.

What is an object in this scheme? We shall follow nature, in a sense. What we generally see with our eyes is light reflected from various surfaces. Similarly here, we shall restrict ourselves to drawing objects which are made up of a number of faces, such as a cube or pyramid. A cube, for example, has six faces. The faces we allow will always be two-dimensional polygons. It will therefore by described almost completely by a sequence of its vertices, with successive vertices spanning an edge of the face. It will also be necessary to distinguish between the two sides of a face. In our programs, we shall specify a face therefore in two components: (1) a collection of points, the vertices of the polygon, and (2) a vector of unit length perpendicular to the face, pointing away from it in the direction we think of as the outside of the face. We shall include all these data in an array, according to the following conventions:

A **drawable object** is an array of faces.
A **face** is an array of two items, a polygon and a normal vector.
A **polygon** is an array of three dimensional vectors.
A **normal vector** is a vector of unit length perpendicular to the polygon.

A polygon can be any sequence of points in space, but if it is part of the face of a surface, then we want to think of it as closed. We adopt the convention that the array of points in a closed polygon has the same first and last point.

For example, we can represent a unit cube, centered at the origin and aligned with the axes, as an array of six faces. The top face will look like this:

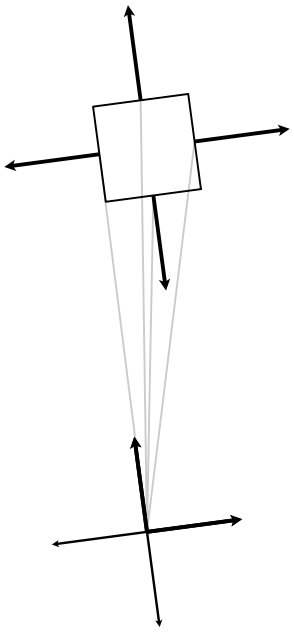
```
[
  [
    [-0.5 -0.5  0.5]
    [ 0.5 -0.5  0.5]
    [ 0.5  0.5  0.5]
    [-0.5  0.5  0.5]
    [-0.5 -0.5  0.5]
  ]
  [0 0 1]
]
```

We shall not worry about the order in which the faces of an object are to be listed, nor which point on a face we start with. Another point is that the procedures we shall use in drawing in 3D will be terribly redundant. The same point will be transformed several times, whereas careful programming might have gotten by with one transform. The consequent inefficiency will have a noticeable effect on the speed of our three dimensional drawing. But this is inescapable if our programs are to be readable.

2. How do we move it?

Suppose now that we have defined an object as a collection of faces. Suppose we want to move it, which means to apply a rigid transformation to it. Recall that a rigid transformation is an array **[M v]** of two items, a matrix M and a vector v. The matrix M is its *linear* component and v its *translation* component. If u is any vector we apply the transformation to u by calculating $Mu + v$. To move an object by a transformation A we apply it to each one of the points on its faces, and in addition we apply it to the normal vectors. However, the normal vector has to be treated a bit differently from the vertices. It represents a direction, not a location, and translation ought not to affect direction. Therefore, to transform a normal vector n by a rigid transformation **[M v]** we apply only the linear component. That is to say, we ignore the translation and just calculate Mn.

A picture might make it clearer why we only apply the linear component of a rigid transformation to the normal vector. Since the normal vector is essentially a direction, we can treat it as if it were a vector whose tail is at the origin.



3. A package of matrix manipulation procedures

Moving objects around in space requires an enormous number of transformations, hence an enormous number of matrix and vector calculations.

Routines for this purpose are contained in a package **matrix3d.inc**, which is entirely concerned with matrix and vector manipulation in three dimensions. As usual, arguments are put onto the stack before the routine is called, and return values are left on the stack at exit.

Vectors in this scheme are arrays of three numbers **[x y z]**. Matrices are arrays of vectors, like this:

```
[
  [a00 a01 a02]
  [a10 a11 a12]
  [a20 a21 a22]
]
```

As suggested by the indexing above, the convention we adopt is that the vectors in a matrix are interpreted as its the rows.

Routine: **vector-add-3d**
Arguments: Two vectors u and v
Returns: The vector sum $u + v$

Routine: **dot-product-3d**
Arguments: Two vectors u and v
Returns: The dot product $u \bullet v$

Routine: **transform-3d**
Arguments: A matrix M and a vector v
Returns: The product Mv

Routine: **identity-3d**
Arguments: None
Returns: The 3×3 identity matrix I

Routine: **affine-transform-3d**
Arguments: $[M\ u]\ v$ where M is an 3×3 matrix, u and v vectors
Returns: $Mv + u$

Routine: **transpose-3d**
Arguments: A matrix A
Returns: ${}^tA$

Routine: **matrix-mul-3d**
Arguments: Two matrices A, B
Returns: AB

Routine: **vector-length-3d**
Arguments: A vector v
Returns: $\|v\|$

Routine: **normalize-3d**
Arguments: A vector v
Returns: $v/\|v\|$

Routine: **cross-product**
Arguments: Two vectors u and v
Returns: $u \times v$

Routine: **rotate-3d**
Arguments: A vector α, an angle θ, and a vector v
Returns: rotation of v by θ around α

Routine: **rotation-matrix-3d**
Arguments: A vector α and an angle θ
Returns: The matrix of the rotation by θ around α

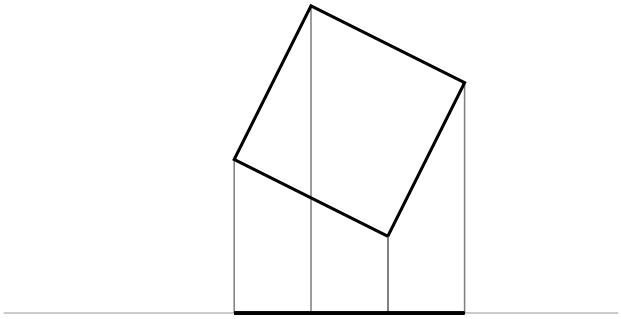
Routine: **vector-scale-3d**
Arguments: A vector v and a scalar c
Returns: cv

4. Converting three dimensions into two

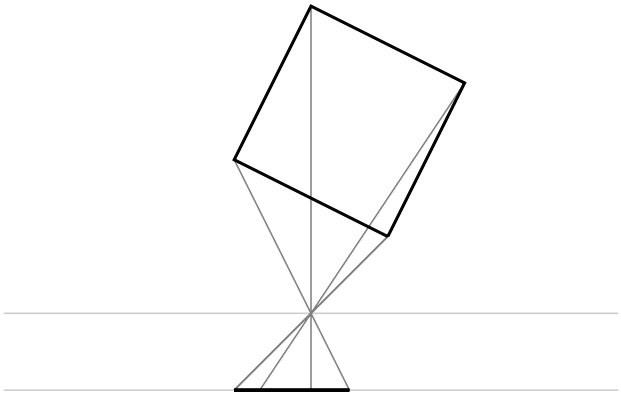
In order to draw a three dimensional object, you have to turn it into a two-dimensional object. In other words, you need to specify a way of transforming three dimensional points to two dimensional ones. There are two standard ways to do this, **projection** and **perspective**. We shall use only the simplest versions of either these.

First we must specify our viewing position. In all cases, we are going to think of the computer screen or the page we are drawing on—the viewing plane—as the (x, y) plane, with the usual orientation. Since we want the right hand rule to hold, we must to assume that the positive z-axis is coming out perpendicularly from this plane towards us. We are looking down the negative z-axis, therefore, and we shall assume that the only part of space visible to us is the region $z < 0$.

Projection is simple. A point (x, y, z) is simple projected down onto the plane $z = 0$. Ignoring the redundant z coordinate, we get the point (x, y).



Perspective is more interesting. Here the idea is to imitate an eye or a pin-hole camera.



The camera has a lens and a back plate. Light goes through the lens, and strikes the back plate. Place the lens at the origin space, and the back plate on the plane $z = 1$. The image of the point (x, y, z) is the point on the line through the points $(0, 0, 0)$ and (x, y, z) which lies on the plane $z = 1$. That line, which we can call the **line of sight** to (x, y, z), is made up of all scalar multiples (sx, sy, sz). We can solve $sz = 1$ to get $s = 1/z$. The image point is then $(x/z, y/z, 1)$.

There is one problem with this scheme. The image on the back plate of the camera is inverted from the image we see ourselves. This is awkward, and instead of using the camera back plate as a model for perspective imaging, we imagine a window laid down between us and the view, in the plane $z = -1$. Our formula for perspective is therefore

$$(x, y, z) \mapsto (-x/z, -y/z) \ .$$

We can incorporate these two techniques in PostScript procedures:

```
% u

/projection {
1 dict begin
/v exch def
[v 0 get v 1 get]
end
} def

% v

/perspective {
2 dict begin
/v exch def
/z v 2 get neg def
[v 0 get z div v 1 get z div]
end
} def
```

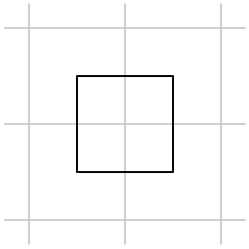
5. Visibility

There are two standard styles of drawing thre dimensional figures. One is to draw all of the edges of all of its faces. This called drawing its **frame**. The other is to draw just the faces that are visible to the observer—i.e. drawing it as if it were solid.

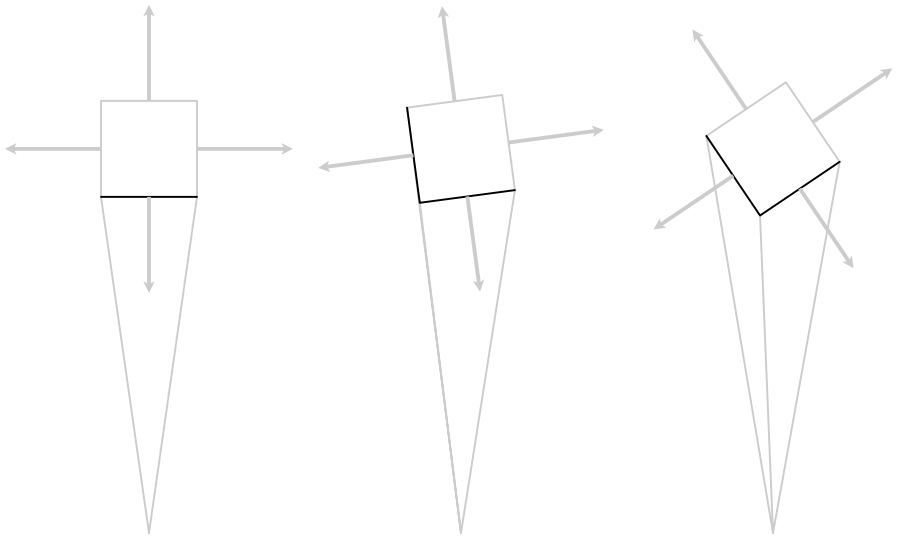
The question of when faces in 3D are visible may become clearer if we try to explain it with a model of what happens in 2D. We'll look in detail only for perspective; the case of projection is much easier.

In this situation, a location matrix is a pair (M, v) where M is a 2×2 rotation matrix and v a 2D vector. We want to draw convex polygons as seen from the origin. To implement perspective on a point, we draw a line from it to the origin and intersect it with the line $y = -1$.

As an example, we might as well look at a square, because all the problems occur there already. The basic square we start with is centred at the origin, 1 unit on a side.



Then we translate it and rotate it around its centre. As we do this, various sides of the square become 'invisible', as seen from the origin.



How can we tell whether a side is invisible or not? As the pictures above indicate, a side becomes invisible just when the line from the origin to a point on that side (the **line of sight** to that point) becomes parallel to the side. A better way to formulate this is to say that *a side becomes just visible when the dot product of the position vector and a normal to that side becomes* 0. All of the side is visible if the normal vector points roughly towards the origin, and invisible if away from the origin. In the first case the dot product is less than or equal to 0, and in the second greater than 0.

We therefore have to compare the transformed normal vector to the transformed position vector. The only slightly tricky point is that the two vectors must be transformed somewhat differently. The position vector is transformed by the whole location matrix, translation and all. But the normal vector really measures the **direction** perpendicular to the transformed face, and in calculating how it is transformed we only apply the matrix component of the location matrix. In other words, we are really interested in the normal vector translated back along the position vector to the origin.

We therefore have the following procedure for deciding whether a side of a transformed figure is visible from the origin or not:

- *We calculate the position vector* r *of one of the vertices of the side. We do this by applying a 2D location matrix to the corresponding corner of the original side.*
- *Then we apply the* **linear component** *of the location matrix to the vector which is normal to the corresponding side of the original square, to get the normal vector* n *to the transformed side.*
- *If* $n \bullet r \leq 0$ *the side is visible.*

For projection, the criterion is much simpler: the face is invisible when the z component of n is negative.

Here are PostScript procedures to to determine visibility of f transformed by A..

```
% A f

/is-visible-in-projection {
3 dict begin
/f exch def
/A exch def
/n A 0 get f 1 get transform-3d def
n 2 get 0 lt
end
```

```
} def

% A f

/is-visible-in-perspective {
2 dict begin
/f exch def
/A exch def
  A f 0 get 0 get affine-transform-3d
  A 0 get f 1 get transform-3d
dot-product-3d 0 lt
end
} def
```

There is one extra remark to make. *It does not matter which point on the face we choose as the position vector!* If P and Q are two points on the face, then the segment PQ lies along the edge and is perpendicular to the normal vector. Hence

$$(OP - OQ) \bullet \nu = 0, \qquad OP \bullet \nu = OQ \bullet \nu \ .$$

6. Shading

One way to make pictures of three dimensional objects more realistic, even more comprehensible to the eye, is to simulate shading. The idea is that we have a light source somewhere, and that a face is brighter if it is turned towards this light than if it is turned away from it. It turns out that we get the most easily interpreted effects if we put the light source up and to the left. We shall specify its direction as a vector of unit length, say $\ell = (-1, 1, 0.5)$. To decide how light a face is, compare its normal vector n of the face with the direction of the light, and more particularly measure the angle between the two. We therefore take the brightness to be a function of the dot-product $d = \ell \bullet n$, which is the cosine of this angle. This ranges between -1 and 1. It is -1 if turned away, 1 if turned towards. We want therefore to choose a possible shade of grey—a number between 0 (black) and 1 (white)—for every number between -1 and 1. One acceptable function is $(1 + d)^2/4$.

```
% normal

/shading {
2 dict begin
/n exch def
/d n light-source dot-product-3d def
1 d add dup mul 4 div
end
} def
```

Exercise 6.1. *Graph the function* $(1 + d)^2/4$ *between* $d = -1$ *and* 1. *Why is it suitable for shading? What's wrong with* $(1 + d)/2$*?*

These last procedure are contained in the single file **`projection.inc`**.