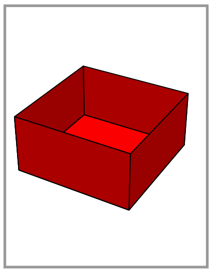
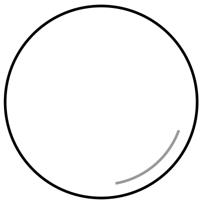
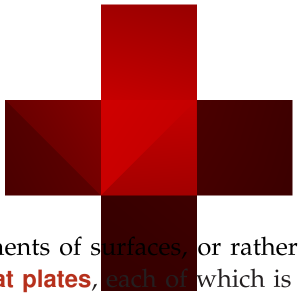


CHAPTER 14

Drawing surfaces in 3D

Only in mathematics books do spheres look like the thing on the left below, rather than the one on the right.

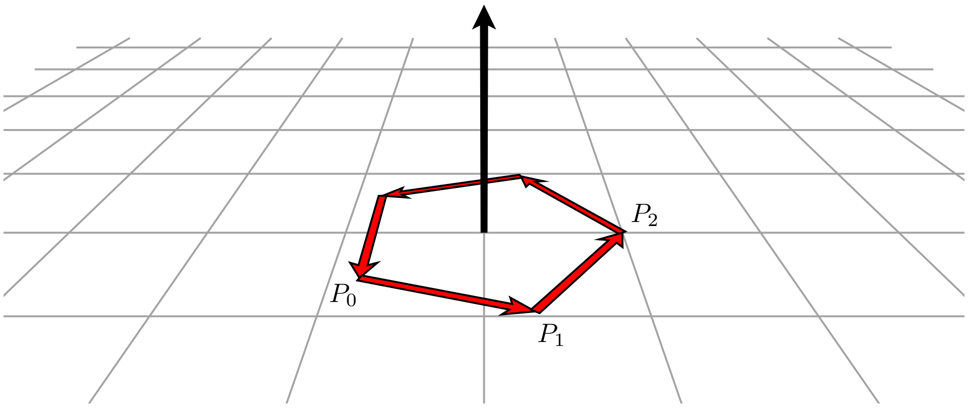
 

What your eye sees in reality are fragments of surfaces, or rather the light reflected from them. In computer graphics, a surface is an assembly of **flat plates**, each of which is a 2D polygon moved into location in space, together with a specification of one of its two sides. One difference between surfaces drawn by computer and those in the real world is that in the real world surfaces possess detail down to microscopic size, including the appearance of smooth curvature. Surfaces drawn by a computer can only be an approximation of these. Sometimes the plates making up a surface will have some extra data added to them to help make the illusion of reality stronger.

PostScript is not efficient enough to do very realistic 3D rendering. Among other things, it does not have access to specialized 3D hardware, and in particular has little comprehension of depth. But it is efficient enough to do a reasonable job on mathematical images.

14.1. Faces

A **surface fragment** or sometimes **face** will be an oriented polygon made up of 3D points all lying in some single plane. The orientation means a choice of **side** to the polygon—a top as opposed to a bottom, or an outside as opposed to an inside. The orientation in practice means that the vertices of the polygon are arranged in an array going around the edges of the polygon according to the right hand rule—so that if the right hand curls around in the direction the vertices are numbered, the thumb points towards the side chosen. In other words, if the polygon is on a plane in front of you and the vertices are arranged in a counter-clockwise direction, your eye is on the outside.



Associated to an oriented polygon in 3D is what I call its **normal function**, the unique linear function $Ax + By + Cz + D$ with these three properties:

- All points in the polygon lie on the plane $Ax + By + Cz + D = 0$;
- the outside of the face is where $Ax + By + Cz + D$ is positive;
- the length of the vector $[A, B, C]$ is 1.

The normal function can be computed from the oriented polygon each time it is to be drawn, but it is more efficient in designing a face to be drawn to compute the normal function once right when the array is determined, and then carry it along as part of the structure of the face. Suppose the vertices of the face are P_0, P_1, P_2, . . . We shall always assume that the edges of a face are **non-degenerate** in the sense that none of its edges have length 0, and that no two successive edges lie in a straight line. In practice this sometimes takes a little work to guarantee, as we shall see later on. At any rate, under this assumption the vector cross-product

$$(P_1 - P_0) \times (P_2 - P_1)$$

will be non-zero and perpendicular to the polygon, facing outwards. Let $[A, B, C]$ be this vector normalized by dividing it by its length. The value of $Ax + By + Cz$ will be the same on all vertices of the polygon, and if we set D to be the negative of this common value then $Ax + By + Cz + D$ will be the normal function.

In the simplest of our methods of drawing surfaces, a face will be by convention an array of two other arrays, the first being the oriented array of points making up the oriented polygon, the second an array of 4 numbers making up the normal function. Thus

```
[[[0 0 0] [1 0 0] [1 1 0] [0 1 0]] [0 0 1 0]]
```

is the face representing a unit square in the (x, y) plane facing out along the positive z-axis, whose normal function is z.

In the code in `ps3d` there is a procedure `normal-function` with one argument, an oriented array of 3D points, which returns the array `[A B C D]` corresponding to its normal function. It implements exactly the calculation described above. If the cross-product is 0, it returns an empty array, which is useful, as we shall see.

14.2. Polyhedra

A **polyhedron** in 3D is a collection of (flat) faces, each of which is a surface fragment whose orientation points outward, making up the boundary of a 3D region with inside and outside. Cubes, for examples, are polyhedra. According to our convention, a polyhedron will be an array of faces, where a face is what is prescribed in the previous section. The following, for example, defines a complete unit cube, centered at $(1/2, 1/2, 1/2)$ with edges aligned parallel to the coordinate axes:

```
/cube [
```

```
  [[[0 0 1] [1 0 1] [1 1 1] [0 1 1]] [0 0 1 -1]]
  [[[0 1 0] [1 1 0] [1 0 0] [0 0 0]] [0 0 -1 0]]
  [[[0 0 0] [0 0 1] [0 1 1] [0 1 0]] [0 -1 0 0]]
  [[[1 1 0] [1 1 1] [1 0 1] [1 0 0]] [1 0 0 -1]]
  [[[0 1 0] [0 1 1] [1 1 1] [1 1 0]] [0 1 0 -1]]
  [[[1 0 0] [1 0 1] [0 0 1] [0 0 0]] [0 -1 0 0]]
] def
```

Assembling cubes by hand takes some care in order to get all the orientations correctly, even for this simplest of polyhedra. The faces here come in pairs which one might call back and front; the array on a back face is almost the same as that on the front, but in reversed order and a shift in one coordinate. The face

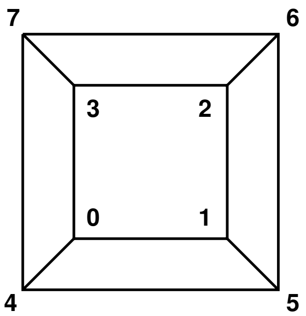
```
[[[0 0 1] [1 0 1] [1 1 1] [0 1 1]] [0 0 1 -1]]
```

is the front face of the cube where $z = 1$, and

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

is the back face corresponding to it, where $z = 0$. There are ways to use the symmetry of the cube to generate all its faces automatically, which will be explored in the Chapter on regular polyhedra. In fact, one rarely constructs a polyhedron without some sort of organizational scheme in mind to generate it.

One thing that will make polyhedra more efficient is to first write out in a single array all of its vertices, then refer to this array in listing the faces. It is also a good idea to record the the normal-functions of the faces before drawing begins. For the cube, for example, it would be better to include the following code in your program rather than the stuff written above:




```
/V [
  [0 0 0] [1 0 0] [1 1 0] [0 1 0]
  [0 0 1] [1 0 1] [1 1 1] [0 1 1]
] def

/cube [
  [[V 4 get V 5 get V 6 get V 7 get] dup normal-function]
  [[V 3 get V 2 get V 1 get V 0 get] dup normal-function]
  [[V 0 get V 4 get V 7 get V 3 get] dup normal-function]
  [[V 5 get V 1 get V 2 get V 6 get] dup normal-function]
  [[V 0 get V 1 get V 5 get V 4 get] dup normal-function]
  [[V 7 get V 6 get V 2 get V 3 get] dup normal-function]
] def
```

It's easier to use this numbering in seeing how to get the face arrays oriented correctly, too.

One interesting exercise is now to write out the arrays representing the triangles of the faces of a **regular tetrahedron**, which is a 3D figure with 4 vertices and 4 faces, all of which are equilateral triangles. The hard part is to find its vertices, and there are several ways to do this. The first is to take a suitable selection of 4 vertices from a cube—a selection of cube vertices with the property that any two are separated by a diagonal on a face of the cube. I'll leave that construction as an exercise without further comment. The trouble with it is that you aren't quite sure where you are, so to speak—the tetrahedron you get has an orientation in space that also requires some calculation to work conveniently with. More convenient ultimately, but more difficult to design in the first place, is one with a fixed radius and orientation, for example one such that

- the center is at $(0, 0, 0)$;
- the top is placed at $(0, 0, 1)$;
- this leaves us free to rotate the tetrahedron around the z-axis; but now fix it by specifying one vertex of the bottom is at $(0, y, z)$ for some suitable values of $y > 0$ and z.

Exercise 14.1. *Figure out what the vertices of this tetrahedron have to be in order to make all faces equilateral. Compose PostScript code as was done above for the cube.*

Exercise 14.2. *Draw a regular tetrahedron by hand, including vertex numbers, and write a code fragment analogous to that above.*

Exercise 14.3. *A* **regular octahedron** *has eight sides, all equilateral triangles. Find the vertices of some regular octahedron. Write PostScript code to draw it.*

14.3. Visibility for convex polyhedra

Usually when you draw objects in 3D they are solid—i.e. a face blocks off the faces behind it. Now in some circumstances this can be quite difficult to deal with, but there is a large class of objects for which it is easy—polyhedra which are **convex** and **closed**. 'Closed' means it has no holes. 'Convex' means it bulges out at all points, or at least never bulges in. For example, a sphere is convex, but the surface of a doughnut is not, since it has that hole in the middle. The difference, for drawing purposes, is this: if a closed object is not convex, then in some views you will have two faces pointing towards the eye, one at least partly obscured by the other. This means that you have to draw them in the correct order, so that the one behind is covered over by the one in front. If an object has holes, the plates making it up should be considered to have two sides, literally an in-side and an out-side, and it will not be convex. We'll look at the problems of drawing non-convex bodies later on.

A convex surface will lie entirely on one side of the plane spanned by each of its faces. (This is actually the technical definition of convexity.) To draw a convex surface, therefore, we can just check visibility of each face separately, without worrying about other faces. To check visibility for a single face, we have to check whether the eye lies in the region of visibility of that face. There is something subtle involved in this—in constructing a face we calculate its normal function, say $f = Ax + By + Cz + D$, and to check visibility for that original face we just have to evaluate that function on the eye. The eye is usefully expressed in homogeneous or 4D coordinates as $[x_e, y_e, z_e, w_e]$. To check visibility we therefore just check the condition

$$[x_e, y_e, z_e, w_e] \bullet [A, B, C, D] = Ax_e + By_e + Cz_e + Dw_e \geq 0 \ .$$

But now in the course of making our program we have probably performed some transformations— for example, rotations—on the surface. This changes the points we draw, and changes also the normal function of the face. In principle we can recover the coordinates of the transformed face, and calculate also its new normal function. We can do this because one of the data structures in `ps3d` is the current 3D transform matrix T which transforms from the coordinates we are currently working with to the original default 3D coordinates. But there is something more efficient to do. As far as visibility is concerned, rotating an object in one direction is equivalent to rotating the eye in the opposite direction, and similarly moving an object away from you is equivalent to translating the eye away from the object in the opposite direction. In other words, if we want to check visibility of a face, we can *either* evaluate the visibility of the transformed face at the true eye, *or* evaluate the original normal function at the eye-equivalent obtained by applying the inverses of the coordinate changes. For the first method, we would apply the current 3D transform matrix T to the original face. But the `ps3d` mechanism also holds the inverse of this matrix T^{-1}. So applying T^{-1} to the eye will tell us where the 'virtual-eye' is at any moment. The matrix T is item 0 in the array you get by calling `cgfx3d`, and T^{-1} is item 1 in it. Therefore the code

```
/E get-eye cgfx3d 1 get transform3d def
```

defines `E` to be the virtual-eye; the procedure `get-virtual-eye` in `ps3d` does exactly the same. The command sequence `cim3d` is also a shorter replacement defined in `ps3d` for `cgfx3d 1 get`. Here is a fragment of program that, given the fragment above defining a cube and the usual 2D stuff setting the scale, will draw only the visible faces of a cube.

```
[0 0 4 1] set-eye
[1 1 1 ] 60 rotate3d
-0.5 -0.5 0 translate3d

/E get-virtual-eye def

cube {
  aload pop
  /f exch def % f = normal function
  /p exch def % p = array of vertices on the face
  f E dot-product 0 ge {
    newpath
     % move to last point first
    p p length 1 sub get aload pop moveto3d
    p {
      aload pop
      lineto3d
    } forall
    stroke
  } if
} forall
```

14.4. Shading

Checking visibility of faces will contribute to an illusion of solidity, especially if the object is shown in a sequence of successive positions in a kind of animation. Another technique for creating an illusion of solidity is that of shading faces according to where they are located relative to an imaginary light source.

The light source will usually be a direction in 3D, therefore a 4-vector whose last coordinate is 0. A light source straight overhead would be `[0 1 0 0]`, for example. Conventionally, a light source from overhead, slightly behind, and slightly to the left seems to be what the human eye is comfortable with, which would make the vector `[-0.25 1 0.25 0]`. It is best to normalize the light source so it has total length 1.

The shade of a face will then be a function of the angle between the light source and the vector perpendicular to the face. If it is $0°$ the face will be towards the light, and bright. If it is $180°$ it will be away from the light, and dark. The angle is a function of the dot-product of the two vectors, and in fact there is no need to work with the angle itself, just the dot-product.

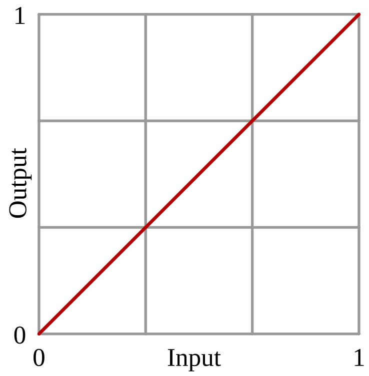
With our normalizations of the normal function and the light source, the dot-product will lie between -1 and 1. If -1 dark, if 1 bright. The simplest way to assign a shade is to let d be the dot-product, and assign a color $(d+1)/2$ in PostScript. So -1 becomes black, 1 white.

As with the eye, it is best to use a light-source-equivalent. So possible code for shading is this:

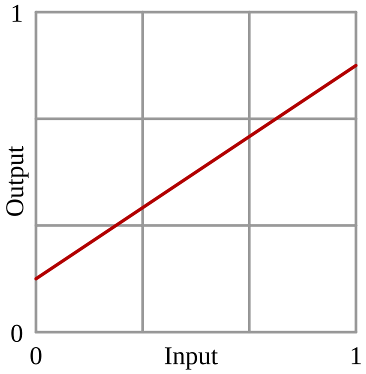
```
/light-source [-0.25 1 0.25 0] normalized def
/L light-source ctm3d 1 get transform3d def
```

```
cube {
  aload pop
  /f exch def  % f = normal function
  /p exch def  % p = array of vertices on the face
  f E dot-product 0 ge {
    newpath
     % move to last point first
    p p length 1 sub get aload pop moveto3d
    p {
      aload pop
      lineto3d
    } forall
     gsave
     /s L f dot-product 1 add 2 div def
     s setgray
     fill
     grestore
    stroke
  } if
} forall
```

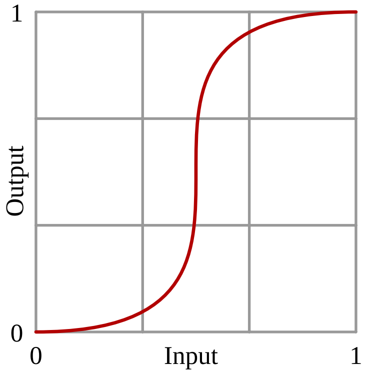
This is not yet ideal. Relying on a straight linear translation from dot-product to shade produces lighting that seems a bit harsh to the eye. For one thing, in the real world even the darkest places usually have a bit of reflected light from the environment, so the darkest shade allowable shouldn't actually be black. And often you won't want the brightest to be 1, which will make a face invisible against a white background. It is best to allow more control over the translation, to allow a more general function to do the job. We'll think of this in the following way: we first calculate the linear translation $(d+1)/2$ just as above, but then we fudge things a bit by translating the result, which lies between 0 and 1, to some other number between 0 and 1. We want to be able to specify a function from the interval $[0,1]$ to itself. The best way to understand what is going on is to look at the graph of our fudge function. We can put it all in a unit box. For example, here is the default translation, with no fudging (recall, we are looking at the translation after we have moved $[-1,1]$ into $[0,1]$):



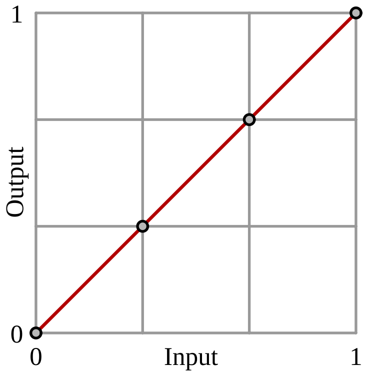
If we want to lighten the shadows and darken the bright spots, we want something like this:



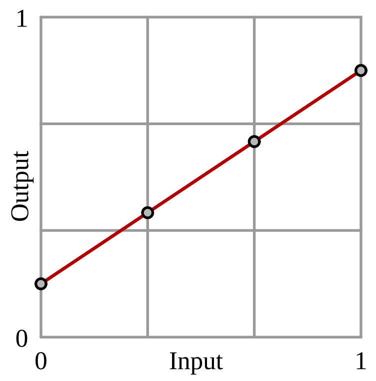
If we want to increase contrast, we want darks darker, lights lighter. If we want to increase contrast a lot, we want an S-shaped curve like this:



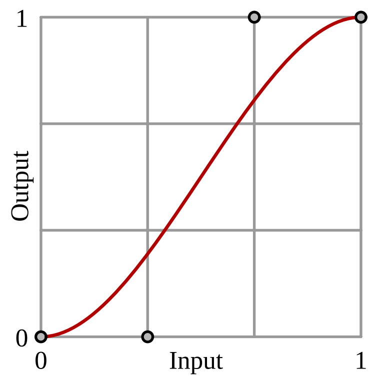
The default method of fudging I have included in `ps3d` doesn't allow the full range of these options. It uses a procedure I call `shade` to translate from the dot-product to a shade factor. There are two arguments to `shade`, a single number between -1 and 1 and an array of 4 numbers. The first and last numbers in the array are the minimum and maximum values of the shade factor, and the other two are more subtle parameters of the shade function, ones that determine control nodes $(1/3, s_1)$ and $(2/3, s_2)$ for the graph. For the default shading, without fudging, the array is `[0 1/3 2/3 1]`:



For an arbitrary straight line fudge function, choose the values in the array so that $(0, s_0)$, $(1/3, s_1)$, $(2/3, s_2)$, $(1, s_3)$ all lie in a straight line.

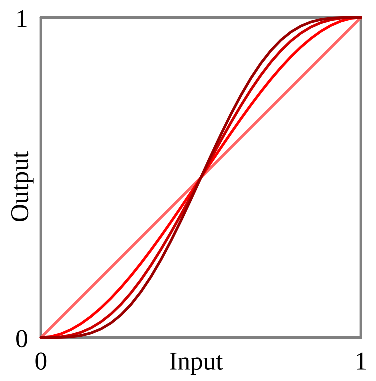


For non-linear fudging, the control values s_1 and s_2 are chosen to force the fudge graph to lie close to the points $(1/3, s_1)$ and $(2/3, s_2)$ without passing through them. There is a fault with this scheme, since the amount of contrast it can achieve is limited:



In order to avoid unwelcome curiousities, the array numbers should satisfy $0 \le s_0 \le s_1 \le s_2 \le 1$, but anything in that range should be acceptable.

For more control over shading, say with higher contrast, you can apply a Bernstein polynomial of degree higher than 3. The disadvantage of doing this is that it requires more computational effort, the advantage is that you can make arbitrarily strong contrast. Here, for example, are some fudge graphs you can get by using a cubic Bézier curve (i. e. Bernstein polynomial of degree 3), and then Bernstein polynomials of degree 5 and 7.



14.5. Smooth surfaces

In reality, most surfaces look essentially smooth, not at all like polyhedra. Nonetheless, computers can only approximate them, usually as polyhedra together with some extra data. For a few surfaces like spheres there are special ways to approximate them in this way, but for most surfaces the first step in drawing them is to **parametrize** them—to find maps from 2D into 3D that describe them.

Let's start with spheres. A sphere is determined completely by its center and its radius. In drawing, the center might as well be taken to be the origin, since `translate3d` can just move a sphere around. Different radii can be dealt with by `scale3d`, so I'll assume the radius to be $R = 1$.

A point on the surface of a sphere has **longitude** θ and **latitude** φ as its coordinates. Longitude measures the angular distance between it and a fixed meridian line, while latitude measures how far it is from an equator. If a point on a sphere of radius R has longitude θ and latitude φ then its (x, y, z) coordinates are

$$S(\theta, \varphi) = (R\cos\theta\cos\varphi, R\sin\theta\cos\varphi, R\sin\varphi) .$$

The parametrization must have the right orientation—these variables must look more or less like x and y everywhere on the surface, as it does here. The parametrization procedure has two arguments θ and φ, and returns the 3D point exhibited above.

```
 % longitude latitude -> point on unit sphere
/P { 1 dict begin
  /l exch def
  /L exch def
   % L=longitude l=latitude
  [L cos l cos mul
   L sin l cos mul
   l sin ]
end } def
```

We want now to use this parametrization to approximate the sphere by an assembly of flat plates. That's easy—a plate will be what you get when you map a coordinate rectangle into 3D. As θ ranges from 0 to 360 and φ from -90 to 90 (using degrees instead of radians, as is normal in PostScript) we cover the whole sphere. First we assemble the vertices in a double array. We use a procedure with a single argument N that returns an N by $2N$ array of grid points on the sphere, except that the poles are singletons. Each internal array is an array of points laid out along a parallel of latitude.

```
 % N -> latitudes in N+1 rows
/sphere-vertex { 1 dict begin
/N exch def

/dA 180 N div def
/dB 360 N div 2 div def

 % A = latitude
[
  [
    [0 0 -1]
  ]
/A -90 dA add def
N 1 sub {
  [
    /B 0 def
     % B = longitude
```

```
      2 N mul 1 add {
        B A P
        /B B dB add def
      } repeat
    ]
    /A A dA add def
    } repeat
    [
      [0 0 1]
    ]
  ]
  end } def
```

Finally, we assemble the vertices into faces.

```
   % N -> array of faces
  /sphere { 1 dict begin
  /N exch def

  /S N sphere-vertex def
   % S now is an array of vertices, arranged in latitudes
  [
     % the triangles at the south pole
    0 1 2 N mul 1 sub { /j exch def
      [
        [ S 0 get 0 get
          S 1 get j 1 add get
       S 1 get j get
        ] dup normal-function
      ]
    } for
     % the rectangular regions in the middle
    1 1 N 2 sub { /i exch def
      0 1 2 N mul 1 sub { /j exch def
        [
       [
       S i get j get
       S i get j 1 add get
            S i 1 add get j 1 add get
            S i 1 add get j get
          ] dup normal-function
        ]
      } for
    } for
     % the triangles at the north pole
    0 1 2 N mul 1 sub { /j exch def
      [
        [ S N 1 sub get j get
          S N 1 sub get j 1 add get
          S N get 0 get
        ] dup normal-function
      ]
    } for
```

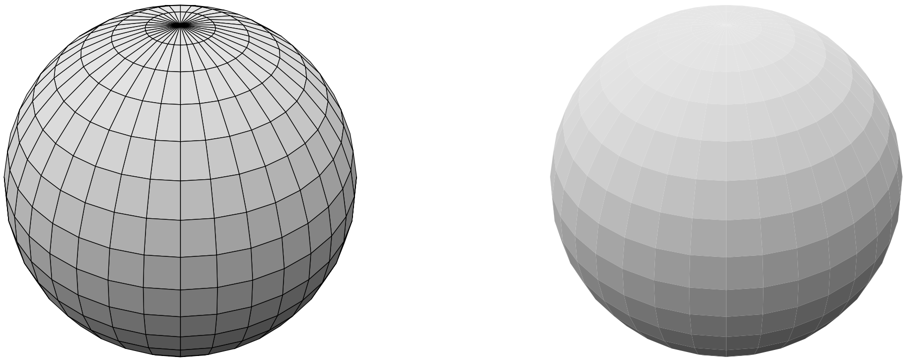
```
  ]
  end } def
```

To draw the regions on the sphere that we have constructed, we can just plug in the array of faces of a sphere where we dealt with the faces of a cube in earlier code.

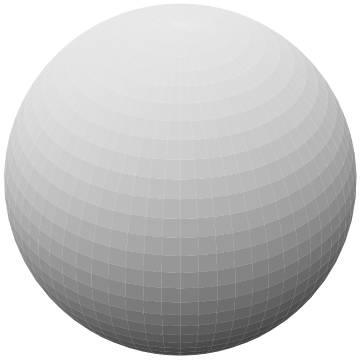
```
/S 18 sphere def
/E get-eye cim3d transform3d def
/L light-source cim3d transform3d def

S {
  aload pop
  /f exch def  % f = normal function
  /p exch def  % p = array of vertices on the face
  f E dot-product 0 ge {
    newpath
     % move to last point first
    p p length 1 sub get aload pop moveto3d
    p {
      aload pop
      lineto3d
    } forall
     gsave
     L f dot-product [ 0.2 0.2 0.9 0.9 ] shade
    setgray
     fill
     grestore
    stroke
  } if
} forall
```

Here is what we see with $N = 16$, in one version stroking the rectangle boundaries, in the other not.



In the following figure, $N = 32$. I would have liked to have included a picture with $N = 64$, but that would mean drawing—well, making an attempt to draw—roughly $8,192$ rectangles in 3D. I am afraid that my printer freezes up at the prospect, just contemplating this task. We'll see in the next section that this is unnecessary, anyway.



14.6. Smoother surfaces

Version 3 of PostScript introduced high quality shading. In earlier versions, if you wanted to show a gradient of color across a region you had to fill in all the separate colors in small regions, or maybe draw zillions of lines of different colors. This was almost always painful or slow or both. In the new scheme, you set colors at certain points and PostScript fills in between these points by interpolating the specified colors. There are several different methods for doing this—seven in all, to be exact—but we shall look here at one which nicely balances simplicity against quality, called **free-form Gouraud shading**. It fits in quite well with surface parametrization. There is one extra requirement now, however—the parametrization must specify not only location, but also the normal function at each point of the surface. We'll see why in a moment. At any rate, this is easy enough to obtain from the parametrization, by calculus. If the parametrization is

$$(s, t) \longmapsto (x(s, t), y(s, t), z(s, t))$$

then the vectors

$$[\partial x/\partial s, \partial y/\partial s, \partial z/\partial s] \,,\; [\partial x/\partial t, \partial y/\partial t, \partial z/\partial t]$$

span the tangent plane to the surface, and their cross product will be perpendicular to it and facing out—as long as the orientation of the parametrization is correct, so that (s, t) have the same orientation as (x, y) on the surface. Normalize this cross product to get $[A, B, C]$. For Gouraud shading, the parametrization should return this normal vector as well as the location. For the sphere, the following procedure will do, since the normal vector is the same as the location vector:

```
 % longitude latitude -> [ point on unit sphere, normal vector ]
/P { 1 dict begin
  /l exch def
  /L exch def
   % L=longitude l=latitude
  /x L cos l cos mul def
  /y L sin l cos mul def
  /z l sin def
  [[x y z] [x y z 0]]
end } def
```

The way PostScript does shading of any kind is through a special data structure called a **shading dictionary**. All you have to know is that the code that produces the shading looks like this:

```
<<
    /ShadingType 4
    /ColorSpace [ /DeviceGray ]
    /DataSource [ 0 x y g ...  ]
```

```
>>
shfill
```

or this:

```
<<
    /ShadingType 4
    /ColorSpace [ /DeviceRGB ]
    /DataSource [ 0 x y r g b ...  ]
>>
shfill
```

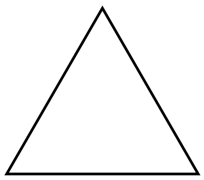
Here `shfill` is a command, a 'shading fill,' with the dictionary `<< ... >>` as argument. A dictionary is a list of **key** and **value** entries, associating a value with each key word listed. In the dictionary `/ShadingType 4` specifies what kind of a shading dictionary (free-form Gouraud) this is. Another kind that you might want to look into is Type 6, **Coons patch meshes**, which allows you to introduce curvature into your shaded fragments. These are all discussed in the section on patterns in the PostScript Reference Manual. The key `ColorSpace` specifies the type of colors to be used. The value `/DeviceGray` says that colors are specified by a single number between 0 and 1, specifying a shade of gray. Another possibility would be `/DeviceRGB`, indicating three numbers RGB. The `/DataSource` lists 2D points and colors to be interpolated between them. Here it is an array of 3×4 numbers, since there are three vertices in a triangular plate. Each of the 4 numbers is associated to a single vertex. The first of the 4 numbers is a 'magic number' that for us will always be 0. The next two are coordinates in the current user 2D coordinate system of the image of the vertex, and g the shade of gray associated to that vertex. If the color space is RGB then this single number becomes three color components. The point (x, y) is calculated from our 3D points in a way I'll explain in a moment.

These shading routines are not unique to 3D drawing. The following code fragment draws a single triangle with colors black, red, and white at the vertices.

```
/A [ 0 3 sqrt 2 div ] def
/B [ -0.5 0 ] def
/C [ 0.5 0 ] def

/ds [
  0 A aload pop 1 0 0
  0 B aload pop 1 1 1
  0 C aload pop 0 0 0
] def

newpath
<<
  /ShadingType 4
  /ColorSpace [ /DeviceRGB ]
  /DataSource ds
>>
shfill
```



For a 3D object, the color at any point will be determined by the light source and the normal vector at that point. Thus the data structures needed for Gouraud shading are slightly different from the flat-plate scheme we used earlier. For one thing, *we now must use triangular plates instead of rectangular ones*. To use Gouraud shading to draw any 3D shape, the first step is to build it as a family of triangles, each with normal vectors at the vertices plus a normal function for the triangle itself. The normal vector for the triangle is used to test visibility, and the ones at the vertices are used to interpolate colors. We think of the surface to be drawn as an array of such colored triangles.

To be precise, a **vertex** P is here is an array `[[x y z] [A B C 0]]` where x, y, z are the 3D coordinates of P, $[A, B, C]$ the unit normal vector at P. This is the sort of structure to be returned from the parametrization procedure. A **triangle** is an array `[P Q R [A B C 0]]` where P, Q, R are vertices in this sense. A **surface** is an array of triangles. Thus

```
/T [
  P Q R
  [ P 0 get Q 0 get R 0 get ] normal-function
] def
```

defines a triangle `T`, if P, Q, R are vertices.

Usually these data will be constructed from a parametrization, but not always. We'll discuss how to do that efficiently in a moment.

Given an array `surface`, an array of triangles in this sense, here is how it is drawn:

```
surface {
   % [ P Q R normal ] now on stack
  aload pop
  /f exch def  % f = normal function
  /R exch def
  /Q exch def
  /P exch def
   % P, Q, R = vertex = [ pt + normal ]
  f E dot-product 0 ge {
    newpath
     % define grey tones for shading
     /sP L P 1 get dot-product 1 add 2 div def
     /sQ L Q 1 get dot-product 1 add 2 div def
     /sR L R 1 get dot-product 1 add 2 div def
     /ds [
       0 [ P 0 get aload pop 1 ] CTM transform3d render sP
       0 [ Q 0 get aload pop 1 ] CTM transform3d render sQ
       0 [ R 0 get aload pop 1 ] CTM transform3d render sR
     ] def

     newpath
     << /ShadingType 4
       /ColorSpace [ /DeviceGray ]
       /DataSource ds >>
    shfill
  } if
} forall
```

The line

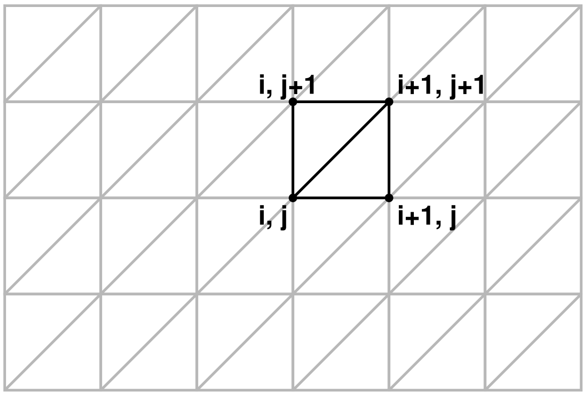
```
0 P 0 get transformto2d sP
```

has the 'magic number' 0; the 3D point P rendered to 2d; and then the shade at P. Here is what the figure looks like if `surface` is a sphere:

Smooth, eh? Shading in this figure has been done with a Bernstein polynomial of degree 5.

Most surfaces will be built from a parametrization by a rectangular array. Here is a sketch of the way things go, at least when we are drawing a surface covered by a rectangle *via* parametrization. Before we do any drawing at all, we do some preparation. (1) We write the parametrization function, which returns an array of two arrays, location plus normal function. (2) We build the $(M+1) \times (N+1)$ array of the object's 3D points plus normal function at those points, using the parametrization function. Here M and N are positive integers which we choose large enough to give an illusion of smoothness. Experimentation will probably be necessary to get them right. In effect we have now built a grid of size $M \times N$ covering the surface.

Then we build all the triangular faces we want to draw, using the grid as indicated here:



Here's code that does this:

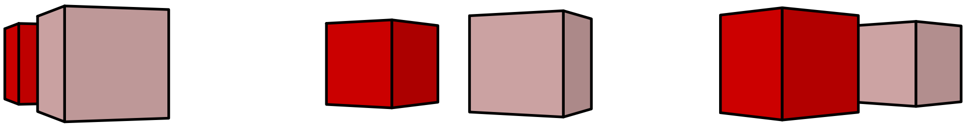
```
/sphere [
0 1 2 N mul 1 sub {
  /i exch def
  0 1 N 1 sub { /j exch def
    /P S i get j get def
    /Q S i 1 add get j get def
    /R S i 1 add get j 1 add get def
    /n [ P 0 get Q 0 get R 0 get ] normal-function def
    n length 0 ne {
        [ P Q R n ]
    } if
    /P S i get j get def
    /Q S i 1 add get j 1 add get def
    /R S i get j 1 add get def
```

```
        /n [ P 0 get Q 0 get R 0 get ] normal-function def
        n length 0 ne {
            [ P Q R n ]
        } if
      } for
    } for
    ] def
```

14.7. Abandoning convexity

Few scenes other than the very simplest, even in theoretical mathematical figures, consist of a single convex body. A simple visibility test by means of normal functions will no longer work.

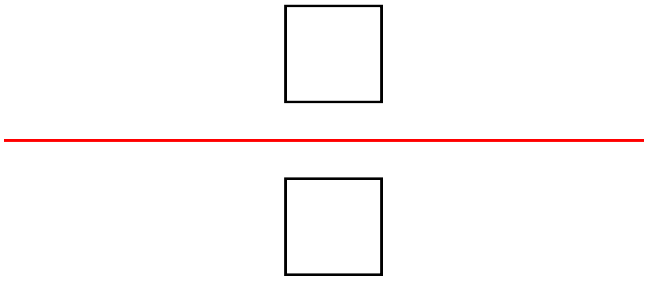
The basic problem can be demonstrated with a simple example. Suppose that you want to draw not just one but two cubes. In certain situations, one of them will hide part or even all of the other from view.



The basic idea is to draw the one that is farthest away first, and then the nearest one, because PostScript paints over what it draws. Indeed, this strategy is called the **painter's algorithm**. Furthermore, if you are doing an animation, in effect moving your eye around the pair of cubes, which one is farthest away and which one is nearest will change, so you must decide dynamically as the scene changes which is to be drawn first. In addition, if you have a large number of objects to draw, you will have to make these dynamical choices efficiently. These all seem like impossible demands, right?

Drawing complicated scenes in much more complicated in 3D than in 2D—and more interesting, since some real ideas are required. Much high-end 3D drawing, for example in video games or movies, relies on a pixel-by-pixel treatment. the pixels in hardware designed for this purpose incorporate a **depth** coordinate—that is to say, depth with respect to the plane of the screen—and pixels are colored in the order of their depth, so that close pixels are painted after far ones. This hardware option is unavailable to PostScript, which is essentially device-independent. The PostScript program itself must therefore be responsible for keeping track of depth. The standard method for doing this is to use a **binary space partition**.

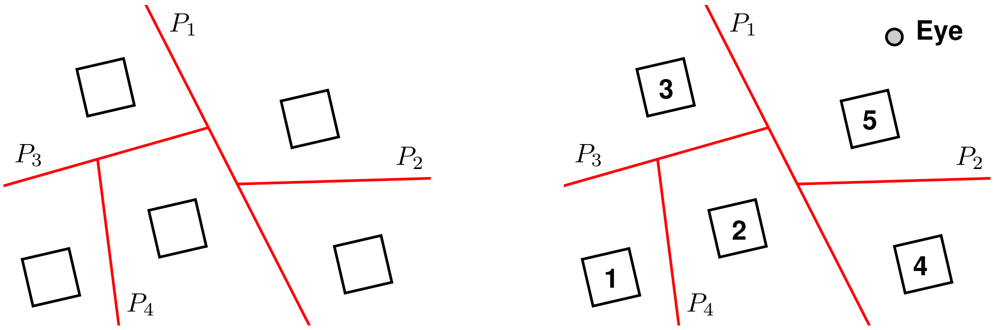
In drawing the two cubes, for example, the trick is to place a plane between the two cubes, and use that to keep track of how the eye is related to the cubes. The cube to draw first is the one on the side of this plane opposite from the eye.



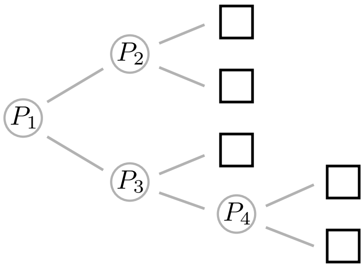
In other words, we divide space into two parts by a plane. Each side of this plane contains one of the cubes. The question of nearest and farthest is determined by determining which side of this plain the eye (or, in practice, what I have called the **virtual eye**) is on.

This division of space into two components by means of this plane is about the simplest example of a binary (two-fold) partition of space. If there are more than two convex objects to be drawn, the idea is to partition all of space by a plane separating one group of objects from the other, then partition each of the half-spaces if necessary by its intersection with a plane, etc. until each of the pieces that space has been chopped up into contains exactly one of the objects to be drawn.

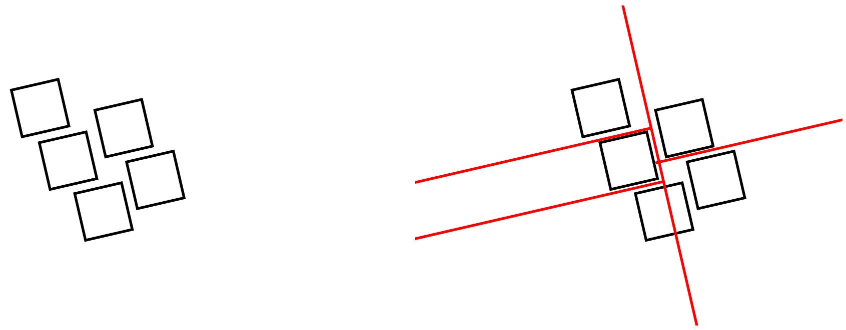
This partition is stored in a way that makes it easy and rapid to decide where the eye lies in relation to these partitioning planes. The important thing is that the construction of the partition itself, which takes up a relatively large amount of time, can be accomplished before any drawing at all. The amount of work involved in this is roughly proportional to n^2 if there are n objects to be drawn. The drawing itself turns out to be proportional to n, which is generally much smaller.



The basic idea, illustrated on the left, is to first partition all of space into two pieces by a single plane (here P_1), then partition each of the remaining components by its intersection with a plane (P_2 and P_3), etc. When using this partition to draw, in this example the program first decides what side of P_1 the eye lies on, and then recursively looks at each side in turn. The right hand image shows the order of drawing with the eye at upper right. The structure necessary to do this drawing is a binary tree whose nodes are the separating planes and whose leaves the objects to be drawn.

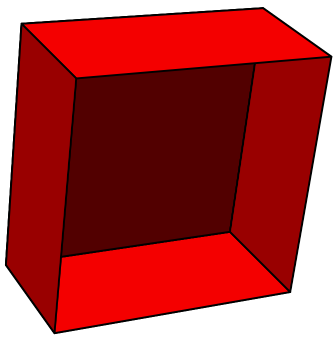


This strategy of partitioning by half-spaces might work easily, or it might not. Sometimes some extra work is involved. As the figure on the left below illustrates, the given figures might have to be chopped up. This is acceptable, since what we are drawing are surface fragments, and a plane cuts a surface fragment into smaller fragments. But we definitely have to allow for this chopping.

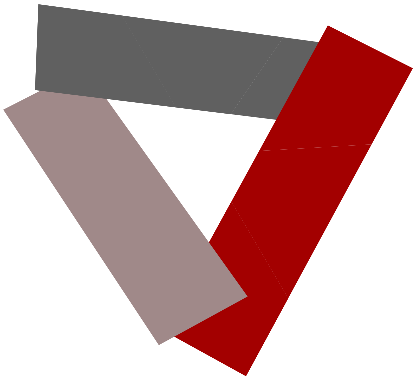


There are special ways to draw special surfaces, but a binary space partition can be used to draw any collection of surface fragments from any perspective. To be precise, a binary space partition is a tree, as I have mentioned, with two kinds of nodes, branches and leaves, and it is defined in an essentially recursive manner. A leaf is simply a surface fragment. A branch is in essence a separating plane, and in practice this plane is chosen to be the support of one of the surface fragments. So it becomes a surface fragment together with the two binary space partitions of the half spaces it determines, conventionally called left and right. If $f \geq 0$ is the defining equation of the separating plane, then all the leaves of the left partition lie entirely in the region $f \leq 0$ and all those on the right lie in $f \geq 0$. More completely, any leaf lies in the intersection of all of the half spaces associated to the branches containing it, which is of course a convex set. The leaves of a binary space partition are drawn recursively. Each node ν of the partition is examined; if the eye lies, say, in the region $f_\nu \geq 0$ then all the leaves in the left partition are drawn, followed by the fragment associated to the node itself, followed in turn by the right partition. In constructing the partition associated to a collection of fragments, a fragment is chosen at random from the collection and all the other fragments split into the halves determined by its support. These are collected into left and right collections, and the partitions associated to each of these is then constructed. This is allowable since the chosen surface fragment itself is excluded from both left and right, and the complexity of each partition construction is decreased at each step.

The results can be quite pleasant, although complicated figures require a long time for PostScript to draw.



Certain configurations illustrate that in some configurations there is absolutely no way to guarantee back-to-front drawing without splitting up fragments, since there is no way to order the original fragments.



Another application of the splitting algorithm used in binary space partitioning is to chop away things in 3D that lie behind the eye, thus eliminating the weird effects mentioned in another chapter.

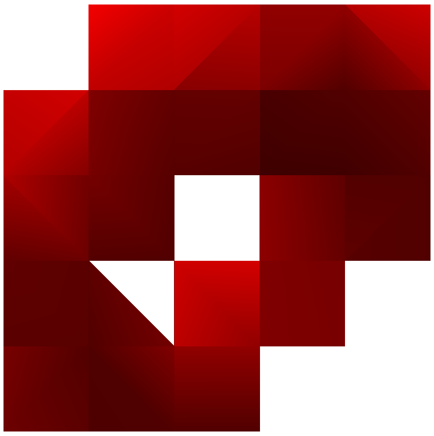
14.8. Summary

Drawing surfaces is quite complicated, and it is probably a good idea if I summarize the process here, at least in the most common cases. I'll work here with the most complicated case, where we are interested in using `shfill`. It should be easy enough to adapt what I say to the easier case that doesn't use it.

- I'll assume the surface can be parametrized by a map $f(s, t)$ from 2D to 3D. The first step is to write the **parametrization function** `f` that has an array `[s t]` of length two as argument and returns the 4-item array `[ x y z n ]`, where n is the unit normal vector to the surface at (x, y, z), the point parametrized by (s, t). I'll call such an array a **vertex**.
- Build an array of vertices over the range you want to look at. Usually this will be correspond to a rectangle in parameter space.
- Assemble the plates of the surface into an array. These can be triangles, rectangles, . . . It is best—i.e. most efficient—to build the largest flat plates possible. For example, on a sphere you build the rectangles laid out by longitude and latitude. Be careful about singular points of the parametrization, such as the poles of the parametrization of the sphere by latitude and longitude, where the rectangles collapse to triangles.
- Decide on a color scheme. Usually just a single color.
- Pick a light source and a shading scheme.
- If you're dealing with a single convex object, there is nothing more to be done except to draw it and shade it, testing visibility with the normal function for plates.
- Otherwise, make up a BSP structure. Then draw. In my code, the procedure that builds the BSP has as one of ist arguments an interpolation routine that constructs the normal vector at a point on a segment between two other vertices. This is needed because constructing the BSP tree occasionally requires that the plates it starts with get split into smaller plates.

14.9. Code

PostScript routines for building binary space partitions can be found in the file `bsp.inc`, and sample usage in `triad.ps`, `box.ps`, and `doughnut.ps`. The last illustrates a complete construction of a smooth surface. Well, maybe not that smooth—note the straight line segments on the boundary of the doughnut. This can be fixed by only the more sophisticated shading using Coons patches.



The parametrization of the doughnut (or **torus**, which my Latin dictionary translates as 'cushion') is

$$(s,t) \longmapsto (\,(R + r\cos t)\cos s, (R + r\cos t)\sin s, r\sin t\,)$$

where R and r are the two radii involved. The easiest way to see this is to start with the circle in the (x,z) plane of radius r at center $(R,0)$ and rotate it around the z-axis in 3D.

In implementing binary space partitioning for drawing fragments, some extra care has to be taken to avoid nasty floating point problems. The main difficulty is in splitting, because if two fragments are almost parallel and one is split by the other, the result may vary wildly depending on floating point errors in the calculation. In practice in drawing 3D figures a lot of fragments will have exactly the same normal function, which can cause extremely bad effects without precautions. One good thing to do is to assign all these fragments *exactly* the same normal functions—i.e. exactly the same array. Another is not to split any fragment by a plane approximately parallel to it, although this is a costly and somewhat arbitrary test.

References

1. M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf, **Computational geometry—algorithms and applications**, Springer Verlag, 1991. Chapter 12 is all about binary space partitions.