Chapter 51

Sneakers in Space

Chapter 51

Using Backface Removal to Eliminate Hidden Surfaces

As I'm fond of pointing out, computer animation isn't a matter of mathematically exact modeling or raw technical prowess, but rather of fooling the eye and the mind. That's especially true for 3-D animation, where we're not only trying to convince viewers that they're seeing objects on a screen—when in truth that screen contains no objects at all, only gaggles of pixels—but we're also trying to create the illusion that the objects exist in three-space, possessing four dimensions (counting movement over time as a fourth dimension) of their own. To make this magic happen, we must provide cues for the eye not only to pick out boundaries, but also to detect depth, orientation, and motion. This involves perspective, shading, proper handling of hidden surfaces, and rapid and smooth screen updates; the whole deal is considerably more difficult to pull off on a PC than 2-D animation.

tip *In some senses, however, 3-D animation is easier than 2-D. Because there's more going on in 3-D animation, the eye and brain tend to make more assumptions, and so are more apt to see what they expect to see, rather than what's actually there.*

If you're piloting a (virtual) ship through a field of thousands of asteroids at high speed, you're unlikely to notice if the more distant asteroids occasionally seem to go right through each other, or if the topographic detail on the asteroids' surfaces sometimes shifts about a bit. You'll be busy viewing the asteroids in their primary role, as objects to be navigated around, and the mere presence of topographic detail will suffice; without being aware of it, you'll fill in the blanks. Your mind will see the topography peripherally, recognize it for what it is supposed to be, and, unless the landscape does something really obtrusive such as vanishing altogether or suddenly shooting a spike miles into space, you will see what you expect to see: a bunch of nicely detailed asteroids tumbling around you.

To what extent can you rely on the eye and mind to make up for imperfections in the 3-D animation process? In some areas, hardly at all; for example, jaggies crawling along edges stick out like red flags, and likewise for flicker. In other areas, though, the human perceptual system is more forgiving than you'd think. Consider this: At the end of *Return of the Jedi*, in the battle to end all battles around the Death Star, there is a sequence of about five seconds in which several spaceships are visible in the background. One of those spaceships (and it's not very far in the background, either) looks a bit unusual. What it looks like is a sneaker. In fact, it *is* a sneaker—but unless you know to look for it, you'll never notice it, because your mind is busy making simplifying assumptions about the complex scene it's seeing—and one of those assumptions is that medium-sized objects floating in space are spaceships, unless proven otherwise. (Thanks to Chris Hecker for pointing this out. I'd never have noticed the sneaker, myself, without being tipped off—which is, of course, the whole point.)

If it's good enough for George Lucas, it's good enough for us. And with that, let's resume our quest for realtime 3-D animation on the PC.

One-sided Polygons: Backface Removal

In the previous chapter, we implemented the basic polygon drawing pipeline, transforming a polygon all the way from its basic definition in object space, through the shared 3-D world space, and into the 3-D space as seen from the viewpoint, called *view space*. From view space, we performed a perspective projection to convert the polygon into screen space, then mapped the transformed and projected vertices to the nearest screen coordinates and filled the polygon. Armed with code that implemented this pipeline, we were able to watch as a polygon rotated about its Y axis, and were able to move the polygon around in space freely.

One of the drawbacks of the previous chapter's approach was that the polygon had two visible sides. Why is that a drawback? It isn't, necessarily, but in our case we want to use polygons to build solid objects with continuous surfaces, and in that context, only one side of a polygon is visible; the other side always faces the inside of the object, and can never be seen. It would save time and simplify the process of hidden surface removal if we could quickly and easily determine whether the inside or outside face of each polygon was facing us, so that we could draw each polygon only if it were visible (that is, had the outside face pointing toward the viewer). On average, half the polygons in an object could be instantly rejected by a test of this sort. Such testing of polygon visibility goes by a number of names in the literature, including backplane culling, backface removal, and assorted variations thereon; I'll refer to it as *backface removal.*

For a single convex polyhedron, removal of polygons that aren't facing the viewer would solve all hidden surface problems. In a convex polyhedron, any polygon facing the viewer can never be obscured by any other polygon in that polyhedron; this falls out of the definition of a convex polyhedron. Likewise, any polygon facing away from the viewer can never be visible. Therefore, in order to draw a convex polyhedron, if you draw all polygons facing toward the viewer but none facing away from the viewer, everything will work out properly, with no additional checking for overlap and hidden surfaces needed.

Unfortunately, backface removal completely solves the hidden surface problem for convex polyhedrons *only*, and only if there's a single convex polyhedron involved; when convex polyhedrons overlap, other methods must be used. Nonetheless, backface removal does instantly halve the number of polygons to be handled in rendering any particular scene. Backface removal can also speed hidden-surface handling if objects are built out of convex polyhedrons. In this chapter, though, we have only one convex polyhedron to deal with, so backface removal alone will do the trick.

Given that I've convinced you that backface removal would be a handy thing to have, how do we actually do it? A logical approach, often implemented in the PC literature, would be to calculate the plane equation for the plane in which the polygon lies, and see which way the normal (perpendicular) vector to the plane points. That works, but there's a more efficient way to calculate the normal to the polygon: as the cross-product of two of the polygon's edges.

The cross-product of two vectors is defined as the vector shown in Figure 51.1. One interesting property of the cross-product vector is that it is perpendicular to the plane in which the two original vectors lie. If we take the cross-product of the vectors that form two edges of a polygon, the result will be a vector perpendicular to the

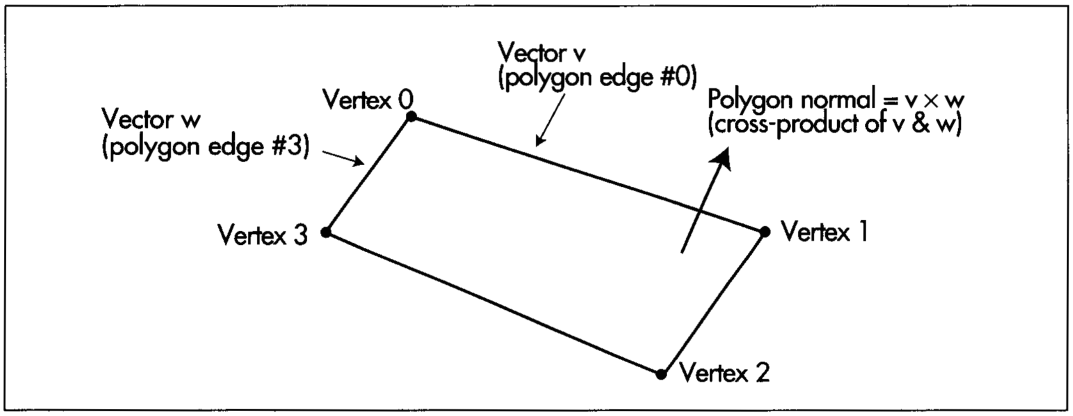
$$V = \begin{bmatrix} V_1 \\ V_2 \\ V_3 \end{bmatrix} \qquad W = \begin{bmatrix} W_1 \\ W_2 \\ W_3 \end{bmatrix} \qquad V \times W = \begin{bmatrix} V_2W_3 - V_3W_2 \\ V_3W_1 - V_1W_3 \\ V_1W_2 - V_2W_1 \end{bmatrix}$$

The cross-product of two vectors.
Figure 51.1

polygon; then, we'll know that the polygon is visible if and only if the cross-product vector points toward the viewer. We need one more thing to make the cross-product approach work, though. The cross-product can actually point either way, depending on which edges of the polygon we choose to work with and the order in which we evaluate them, so we must establish some conventions for defining polygons and evaluating the cross-product.

We'll define only convex polygons, with the vertices defined in clockwise order, as viewed from the outside; that is, if you're looking at the visible side of the polygon, the vertices will appear in the polygon definition in clockwise order. With those assumptions, the cross-product becomes a quick and easy indicator of polygon orientation with respect to the viewer; we'll calculate it as the cross-product of the first and last vectors in a polygon, as shown in Figure 51.2, and if it's pointing toward the viewer, we'll know that the polygon is visible. Actually, we don't even have to calculate the entire cross-product vector, because the Z component alone suffices to tell us which way the polygon is facing: positive Z means visible, negative Z means not. The Z component can be calculated very efficiently, with only two multiplies and a subtraction.

The question remains of the proper space in which to perform backface removal. There's a temptation to perform it in view space, which is, after all, the space defined with respect to the viewer, but view space is not a good choice. Screen space—the space in which perspective projection has been performed—is the best choice. The purpose of backface removal is to determine whether each polygon is visible to the viewer, and, despite its name, view space does not provide that information; unlike screen space, it does not reflect perspective effects.



Using the cross product to generate a polygon normal.
Figure 51.2

Backface removal may also be performed using the polygon vertices in screen coordinates, which are integers. This is less accurate than using the screen space coordinates, which are floating point, but is, by the same token, faster. In Listing 51.3, which we'll discuss shortly, backface removal is performed in screen coordinates in the interests of speed.

Backface removal, as implemented in Listing 51.3, will not work reliably if the polygon is not convex, if the vertices don't appear in clockwise order, if either the first or last edge in a polygon has zero length, or if the first and last edges are collinear. These latter two points are the reason it's preferable to work in screen space rather than screen coordinates (which suffer from rounding problems), speed considerations aside.

Backface Removal in Action

Listings 51.1 through 51.5 together form a program that rotates a solid cube in realtime under user control. Listing 51.1 is the main program; Listing 51.2 performs transformation and projection; Listing 51.3 performs backface removal and draws visible faces; Listing 51.4 concatenates incremental rotations to the object-to-world transformation matrix; Listing 51.5 is the general header file. Also required from previous chapters are: Listings 50.1 and 50.2 from Chapter 50 (draw clipped line list, matrix math functions); Listings 47.1 and 47.6 from Chapter 47, (Mode X mode set, rectangle fill); Listing 49.6 from Chapter 49; Listing 39.4 from Chapter 39 (polygon edge scan); and the **FillConvexPolygon()** function from Listing 38.1 from Chapter 38. All necessary modules, along with a project file, will be present in the subdirectory for this chapter on the listings diskette, whether they were presented in this chapter or some earlier chapter. This may crowd the listings diskette a little bit, but it will certainly reduce confusion!

LISTING 51.1 L51-1.C

```
/* 3D animation program to view a cube as it rotates in Mode X. The viewpoint
   is fixed at the origin (0,0,0) of world space, looking in the direction of
   increasingly negative Z. A right-handed coordinate system is used throughout.
   All C code tested with Borland C++ in C compilation mode. */
#include <conio.h>
#include <dos.h>
#include <math.h>
#include "polygon.h"

#define ROTATION  (M_PI / 30.0)  /* rotate by 6 degrees at a time */

/* base offset of page to which to draw */
unsigned int CurrentPageBase = 0;
/* Clip rectangle; clips to the screen */
int ClipMinX=0, ClipMinY=0;
int ClipMaxX=SCREEN_WIDTH, ClipMaxY=SCREEN_HEIGHT;
/* Rectangle specifying extent to be erased in each page. */
struct Rect EraseRect[2] = { {0, 0, SCREEN_WIDTH, SCREEN_HEIGHT},
   {0, 0, SCREEN_WIDTH, SCREEN_HEIGHT} };
```

```
static unsigned int PageStartOffsets[2] =
   {PAGE0_START_OFFSET,PAGE1_START_OFFSET};
int DisplayedPage, NonDisplayedPage;
/* Transformation from cube's object space to world space. Initially
   set up to perform no rotation and to move the cube into world
   space -100 units away from the origin down the Z axis. Given the
   viewing point, -100 down the Z axis means 100 units away in the
   direction of view. The program dynamically changes both the
   translation and the rotation. */
static double CubeWorldXform[4][4] = {
   {1.0, 0.0, 0.0, 0.0},
   {0.0, 1.0, 0.0, 0.0},
   {0.0, 0.0, 1.0, -100.0},
   {0.0, 0.0, 0.0, 1.0} };
/* Transformation from world space into view space. Because in this
   application the view point is fixed at the origin of world space,
   looking down the Z axis in the direction of increasing Z, view space is
   identical to world space, and this is the identity matrix. */
static double WorldViewXform[4][4] = {
   {1.0, 0.0, 0.0, 0.0},
   {0.0, 1.0, 0.0, 0.0},
   {0.0, 0.0, 1.0, 0.0},
   {0.0, 0.0, 0.0, 1.0}
};
/* all vertices in the cube */
static struct Point3 CubeVerts[] = {
   {15,15,15,1},{15,15,-15,1},{15,-15,15,1},{15,-15,-15,1},
   {-15,15,15,1},{-15,15,-15,1},{-15,-15,15,1},{-15,-15,-15,1}};
/* vertices after transformation */
static struct Point3
      XformedCubeVerts[sizeof(CubeVerts)/sizeof(struct Point3)];
/* vertices after projection */
static struct Point3
      ProjectedCubeVerts[sizeof(CubeVerts)/sizeof(struct Point3)];
/* vertices in screen coordinates */
static struct Point
      ScreenCubeVerts[sizeof(CubeVerts)/sizeof(struct Point3)];
/* vertex indices for individual faces */
static int Face1[] = {1,3,2,0};
static int Face2[] = {5,7,3,1};
static int Face3[] = {4,5,1,0};
static int Face4[] = {3,7,6,2};
static int Face5[] = {5,4,6,7};
static int Face6[] = {0,2,6,4};
/* list of cube faces */
static struct Face CubeFaces[] = {{Face1,4,15},{Face2,4,14},
   {Face3,4,12},{Face4,4,11},{Face5,4,10},{Face6,4,9}};
/* master description for cube */
static struct Object Cube = {sizeof(CubeVerts)/sizeof(struct Point3),
   CubeVerts, XformedCubeVerts, ProjectedCubeVerts, ScreenCubeVerts,
   sizeof(CubeFaces)/sizeof(struct Face), CubeFaces};

void main() {
   int Done = 0, RecalcXform = 1;
   double WorkingXform[4][4];
   union REGS regset;

   /* Set up the initial transformation */
   Set320x240Mode(); /* set the screen to Mode X */
   ShowPage(PageStartOffsets[DisplayedPage = 0]);
```

```
/* Keep transforming the cube, drawing it to the undisplayed page,
   and flipping the page to show it */
do {
   /* Regenerate the object->view transformation and
      retransform/project if necessary */
   if (RecalcXform) {
      ConcatXforms(WorldViewXform, CubeWorldXform, WorkingXform);
      /* Transform and project all the vertices in the cube */
      XformAndProjectPoints(WorkingXform, &Cube);
      RecalcXform = 0;
   }
   CurrentPageBase =    /* select other page for drawing to */
         PageStartOffsets[NonDisplayedPage = DisplayedPage ^ 1];
   /* Clear the portion of the non-displayed page that was drawn
      to last time, then reset the erase extent */
   FillRectangleX(EraseRect[NonDisplayedPage].Left,
         EraseRect[NonDisplayedPage].Top,
         EraseRect[NonDisplayedPage].Right,
         EraseRect[NonDisplayedPage].Bottom, CurrentPageBase, 0);
   EraseRect[NonDisplayedPage].Left =
         EraseRect[NonDisplayedPage].Top = 0x7FFF;
   EraseRect[NonDisplayedPage].Right =
         EraseRect[NonDisplayedPage].Bottom = 0;
   /* Draw all visible faces of the cube */
   DrawVisibleFaces(&Cube);
   /* Flip to display the page into which we just drew */
   ShowPage(PageStartOffsets[DisplayedPage = NonDisplayedPage]);
   while (kbhit()) {
      switch (getch()) {
         case 0x1B:     /* Esc to exit */
            Done = 1; break;
         case 'A': case 'a':      /* away (-Z) */
            CubeWorldXform[2][3] -= 3.0; RecalcXform = 1; break;
         case 'T':      /* towards (+Z). Don't allow to get too */
         case 't':      /* close, so Z clipping isn't needed */
            if (CubeWorldXform[2][3] < -40.0) {
                  CubeWorldXform[2][3] += 3.0;
                  RecalcXform = 1;
            }
            break;
         case '4':         /* rotate clockwise around Y */
            AppendRotationY(CubeWorldXform, -ROTATION);
            RecalcXform=1; break;
         case '6':         /* rotate counterclockwise around Y */
            AppendRotationY(CubeWorldXform, ROTATION);
            RecalcXform=1; break;
         case '8':         /* rotate clockwise around X */
            AppendRotationX(CubeWorldXform, -ROTATION);
            RecalcXform=1; break;
         case '2':         /* rotate counterclockwise around X */
            AppendRotationX(CubeWorldXform, ROTATION);
            RecalcXform=1; break;
         case 0:     /* extended code */
            switch (getch()) {
               case 0x3B:  /* rotate counterclockwise around Z */
                  AppendRotationZ(CubeWorldXform, ROTATION);
                  RecalcXform=1; break;
               case 0x3C:  /* rotate clockwise around Z */
                  AppendRotationZ(CubeWorldXform, -ROTATION);
                  RecalcXform=1; break;
```

```
                    case 0x4B:  /* left (-X) */
                      CubeWorldXform[0][3] -= 3.0; RecalcXform=1; break;
                    case 0x4D:  /* right (+X) */
                      CubeWorldXform[0][3] += 3.0; RecalcXform=1; break;
                    case 0x48:  /* up (+Y) */
                      CubeWorldXform[1][3] += 3.0; RecalcXform=1; break;
                    case 0x50:  /* down (-Y) */
                      CubeWorldXform[1][3] -= 3.0; RecalcXform=1; break;
                    default:
                      break;
                 }
                 break;
              default:       /* any other key to pause */
                 getch(); break;
           }
        }
   } while (!Done);
   /* Return to text mode and exit */
   regset.x.ax = 0x0003;   /* AL = 3 selects 80x25 text mode */
   int86(0x10, &regset, &regset);
}
```

LISTING 51.2 L51-2.C

```
/* Transforms all vertices in the specified object into view space, then
   perspective projects them to screen space and maps them to screen coordinates,
   storing the results in the object. */
#include <math.h>
#include "polygon.h"/

void XformAndProjectPoints(double Xform[4][4],
   struct Object * ObjectToXform)
{
   int i, NumPoints = ObjectToXform->NumVerts;
   struct Point3 * Points = ObjectToXform->VertexList;
   struct Point3 * XformedPoints = ObjectToXform->XformedVertexList;
   struct Point3 * ProjectedPoints = ObjectToXform->ProjectedVertexList;
   struct Point * ScreenPoints = ObjectToXform->ScreenVertexList;

   for (i=0; i<NumPoints; i++, Points++, XformedPoints++,
         ProjectedPoints++, ScreenPoints++) {
      /* Transform to view space */
      XformVec(Xform, (double *)Points, (double *)XformedPoints);
      /* Perspective-project to screen space */
      ProjectedPoints->X = XformedPoints->X / XformedPoints->Z *
            PROJECTION_RATIO * (SCREEN_WIDTH / 2.0);
      ProjectedPoints->Y = XformedPoints->Y / XformedPoints->Z *
            PROJECTION_RATIO * (SCREEN_WIDTH / 2.0);
      ProjectedPoints->Z = XformedPoints->Z;
      /* Convert to screen coordinates. The Y coord is negated to
         flip from increasing Y being up to increasing Y being down,
         as expected by the polygon filler. Add in half the screen
         width and height to center on the screen. */
      ScreenPoints->X = ((int) floor(ProjectedPoints->X + 0.5)) + SCREEN_WIDTH/2;
      ScreenPoints->Y = (-((int) floor(ProjectedPoints->Y + 0.5))) +
               SCREEN_HEIGHT/2;
   }
}
```

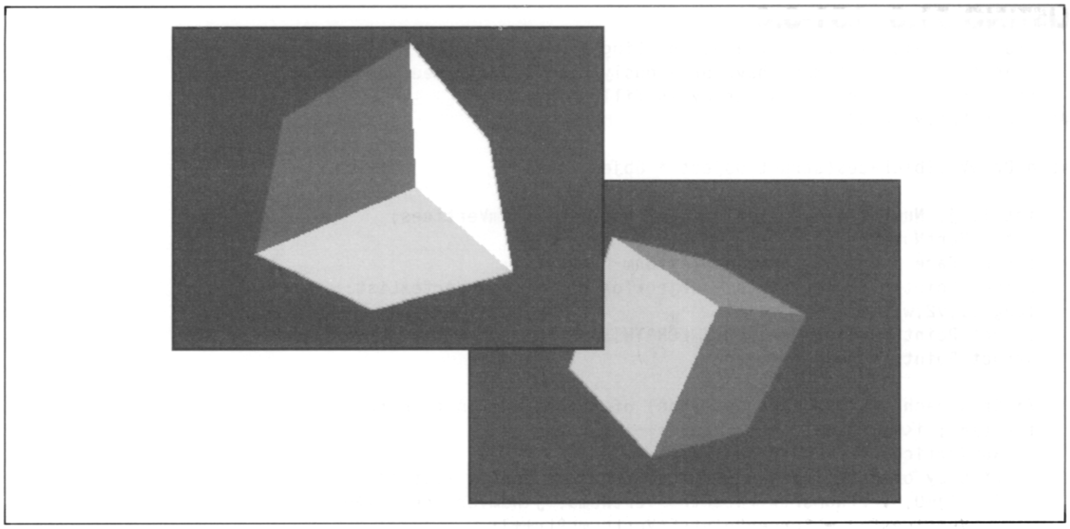
LISTING 51.3 L51-3.C

```
/* Draws all visible faces (faces pointing toward the viewer) in the specified
   object. The object must have previously been transformed and projected, so
   that the ScreenVertexList array is filled in. */
#include "polygon.h"

void DrawVisibleFaces(struct Object * ObjectToXform)
{
   int i, j, NumFaces = ObjectToXform->NumFaces, NumVertices;
   int * VertNumsPtr;
   struct Face * FacePtr = ObjectToXform->FaceList;
   struct Point * ScreenPoints = ObjectToXform->ScreenVertexList;
   long v1,v2,w1,w2;
   struct Point Vertices[MAX_POLY_LENGTH];
   struct PointListHeader Polygon;

   /* Draw each visible face (polygon) of the object in turn */
   for (i=0; i<NumFaces; i++, FacePtr++) {
      NumVertices = FacePtr->NumVerts;
      /* Copy over the face's vertices from the vertex list */
      for (j=0, VertNumsPtr=FacePtr->VertNums; j<NumVertices; j++)
         Vertices[j] = ScreenPoints[*VertNumsPtr++];
      /* Draw only if outside face showing (if the normal to the
         polygon points toward the viewer; that is, has a positive
         Z component) */
      v1 = Vertices[1].X - Vertices[0].X;
      w1 = Vertices[NumVertices-1].X - Vertices[0].X;
      v2 = Vertices[1].Y - Vertices[0].Y;
      w2 = Vertices[NumVertices-1].Y - Vertices[0].Y;
      if ((v1*w2 - v2*w1) > 0) {
         /* It is facing the screen, so draw */
         /* Appropriately adjust the extent of the rectangle used to
            erase this page later */
         for (j=0; j<NumVertices; j++) {
            if (Vertices[j].X > EraseRect[NonDisplayedPage].Right)
               if (Vertices[j].X < SCREEN_WIDTH)
                  EraseRect[NonDisplayedPage].Right = Vertices[j].X;
               else EraseRect[NonDisplayedPage].Right = SCREEN_WIDTH;
            if (Vertices[j].Y > EraseRect[NonDisplayedPage].Bottom)
               if (Vertices[j].Y < SCREEN_HEIGHT)
                  EraseRect[NonDisplayedPage].Bottom = Vertices[j].Y;
               else EraseRect[NonDisplayedPage].Bottom=SCREEN_HEIGHT;
            if (Vertices[j].X < EraseRect[NonDisplayedPage].Left)
               if (Vertices[j].X > 0)
                  EraseRect[NonDisplayedPage].Left = Vertices[j].X;
               else EraseRect[NonDisplayedPage].Left = 0;
            if (Vertices[j].Y < EraseRect[NonDisplayedPage].Top)
               if (Vertices[j].Y > 0)
                  EraseRect[NonDisplayedPage].Top = Vertices[j].Y;
               else EraseRect[NonDisplayedPage].Top = 0;
         }
         /* Draw the polygon */
         DRAW_POLYGON(Vertices, NumVertices, FacePtr->Color, 0, 0);
      }
   }
}
```

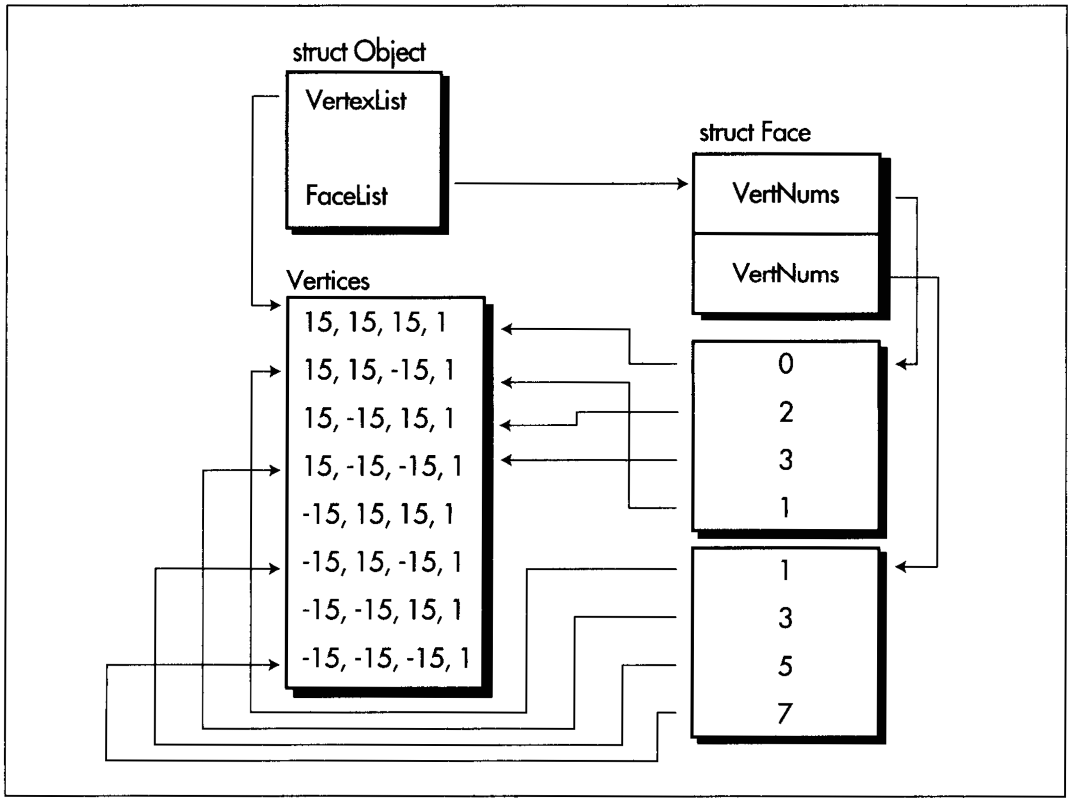
The sample program, as shown in Figure 51.3, places a cube, floating in three-space, under the complete control of the user. The arrow keys may be used to move the



Sample screens from the 3-D cube program.
Figure 51.3

cube left, right, up, and down, and the A and T keys may be used to move the cube away from or toward the viewer. The F1 and F2 keys perform rotation around the Z axis, the axis running from the viewer straight into the screen. The 4 and 6 keys perform rotation around the Y (vertical) axis, and the 2 and 8 keys perform rotation around the X axis, which runs horizontally across the screen; the latter four keys are most conveniently used by flipping the keypad to the numeric state.

The demo involves six polygons, one for each side of the cube. Each of the polygons must be transformed and projected, so it would seem that 24 vertices (four for each polygon) must be handled, but some steps have been taken to improve performance. All vertices for the object have been stored in a single list; the definition of each face contains not the vertices for that face themselves, but rather indexes into the object's vertex list, as shown in Figure 51.4. This reduces the number of vertices to be manipulated from 24 to 8, for there are, after all, only eight vertices in a cube, with three faces sharing each vertex. In this way, the transformation burden is lightened by two-thirds. Also, as mentioned earlier, backface removal is performed with integers, in screen coordinates, rather than with floating-point values in screen space. Finally, the **RecalcXForm** flag is set whenever the user changes the object-to-world transformation. Only when this flag is set is the full object-to-view transformation recalculated and the object's vertices transformed and projected again; otherwise, the values already stored within the object are reused. In the sample application, this brings no visual improvement, because there's only the one object, but the underlying mechanism is



The object data structure
Figure 51.4

sound: In a full-blown 3-D animation application, with multiple objects moving about the screen, it would help a great deal to flag which of the objects had moved with respect to the viewer, performing a new transformation and projection only for those that had.

With the above optimizations, the sample program is certainly adequately responsive on a 20 MHz 386 (sans 387; I'm sure it's wonderfully responsive with a math coprocessor). Still, it couldn't quite keep up with the keyboard when I modified it to read only one key each time through the loop—and we're talking about only eight vertices here. This indicates that we're already near the limit of animation complexity possible with our current approach. It's time to start rethinking that approach; over two-thirds of the overall time is spent in floating-point calculations, and it's there that we'll begin to attack the performance bottleneck we find ourselves up against.

Incremental Transformation

Listing 51.4 contains three functions; each concatenates an additional rotation around one of the three axes to an existing rotation. To improve performance, only the matrix entries that are affected in a rotation around each particular axis are recalculated (all but four of the entries in a single-axis rotation matrix are either 0 or 1, as shown in Chapter 50). This cuts the number of floating-point multiplies from the 64 required for the multiplication of two 4×4 matrices to just 12, and floating point adds from 48 to 6.

Be aware that Listing 51.4 performs an incremental rotation on top of whatever rotation is already in the matrix. The cube may already have been turned left, right, up, down, and sideways; regardless, Listing 51.4 just tacks the specified rotation onto whatever already exists. In this way, the object-to-world transformation matrix contains a history of all the rotations ever specified by the user, concatenated one after another onto the original matrix. Potential loss of precision is a problem associated with using such an approach to represent a very long concatenation of transformations, especially with fixed-point arithmetic; that's not a problem for us yet, but we'll run into it eventually.

LISTING 51.4 L51-4.C

```
/* Routines to perform incremental rotations around the three axes */
#include <math.h>
#include "polygon.h"

/* Concatenate a rotation by Angle around the X axis to the transformation in
   XformToChange, placing result back in XformToChange. */
   void AppendRotationX(double XformToChange[4][4], double Angle)
{
   double Temp10, Temp11, Temp12, Temp20, Temp21, Temp22;
   double CosTemp = cos(Angle), SinTemp = sin(Angle);
   /* Calculate the new values of the four affected matrix entries */
   Temp10 = CosTemp*XformToChange[1][0]+ -SinTemp*XformToChange[2][0];
   Temp11 = CosTemp*XformToChange[1][1]+ -SinTemp*XformToChange[2][1];
   Temp12 = CosTemp*XformToChange[1][2]+ -SinTemp*XformToChange[2][2];
   Temp20 = SinTemp*XformToChange[1][0]+ CosTemp*XformToChange[2][0];
   Temp21 = SinTemp*XformToChange[1][1]+ CosTemp*XformToChange[2][1];
   Temp22 = SinTemp*XformToChange[1][2]+ CosTemp*XformToChange[2][2];
   /* Put the results back into XformToChange */
   XformToChange[1][0] = Temp10; XformToChange[1][1] = Temp11;
   XformToChange[1][2] = Temp12; XformToChange[2][0] = Temp20;
   XformToChange[2][1] = Temp21; XformToChange[2][2] = Temp22;
}

/* Concatenate a rotation by Angle around the Y axis to the transformation in
   XformToChange, placing result back in XformToChange. */
   void AppendRotationY(double XformToChange[4][4], double Angle)
{
   double Temp00, Temp01, Temp02, Temp20, Temp21, Temp22;
   double CosTemp = cos(Angle), SinTemp = sin(Angle);
   /* Calculate the new values of the four affected matrix entries */
   Temp00 = CosTemp*XformToChange[0][0]+ SinTemp*XformToChange[2][0];
   Temp01 = CosTemp*XformToChange[0][1]+ SinTemp*XformToChange[2][1];
   Temp02 = CosTemp*XformToChange[0][2]+ SinTemp*XformToChange[2][2];
   Temp20 = -SinTemp*XformToChange[0][0]+ CosTemp*XformToChange[2][0];
   Temp21 = -SinTemp*XformToChange[0][1]+ CosTemp*XformToChange[2][1];
   Temp22 = -SinTemp*XformToChange[0][2]+ CosTemp*XformToChange[2][2];
   /* Put the results back into XformToChange */
   XformToChange[0][0] = Temp00; XformToChange[0][1] = Temp01;
   XformToChange[0][2] = Temp02; XformToChange[2][0] = Temp20;
   XformToChange[2][1] = Temp21; XformToChange[2][2] = Temp22;
}

/* Concatenate a rotation by Angle around the Z axis to the transformation in
   XformToChange, placing result back in XformToChange. */
   void AppendRotationZ(double XformToChange[4][4], double Angle)
{
   double Temp00, Temp01, Temp02, Temp10, Temp11, Temp12;
   double CosTemp = cos(Angle), SinTemp = sin(Angle);
   /* Calculate the new values of the four affected matrix entries */
   Temp00 = CosTemp*XformToChange[0][0]+ -SinTemp*XformToChange[1][0];
   Temp01 = CosTemp*XformToChange[0][1]+ -SinTemp*XformToChange[1][1];
   Temp02 = CosTemp*XformToChange[0][2]+ -SinTemp*XformToChange[1][2];
   Temp10 = SinTemp*XformToChange[0][0]+ CosTemp*XformToChange[1][0];
   Temp11 = SinTemp*XformToChange[0][1]+ CosTemp*XformToChange[1][1];
   Temp12 = SinTemp*XformToChange[0][2]+ CosTemp*XformToChange[1][2];
   /* Put the results back into XformToChange */
   XformToChange[0][0] = Temp00; XformToChange[0][1] = Temp01;
   XformToChange[0][2] = Temp02; XformToChange[1][0] = Temp10;
   XformToChange[1][1] = Temp11; XformToChange[1][2] = Temp12;
}
```

LISTING 51.5 POLYGON.H

```
/* POLYGON.H: Header file for polygon-filling code, also includes a number of
   useful items for 3D animation. */
#define MAX_POLY_LENGTH 4  /* four vertices is the max per poly */
#define SCREEN_WIDTH 320
#define SCREEN_HEIGHT 240
#define PAGE0_START_OFFSET 0
#define PAGE1_START_OFFSET (((long)SCREEN_HEIGHT*SCREEN_WIDTH)/4)
/* Ratio: distance from viewpoint to projection plane / width of projection
   plane. Defines the width of the field of view. Lower absolute values = wider
   fields of view; higher values = narrower. */
#define PROJECTION_RATIO   -2.0 /* negative because visible Z coordinates are negative */
/* Draws the polygon described by the point list PointList in color Color with
   all vertices offset by (X,Y) */
#define DRAW_POLYGON(PointList,NumPoints,Color,X,Y)          \
   Polygon.Length = NumPoints; Polygon.PointPtr = PointList; \
   FillConvexPolygon(&Polygon, Color, X, Y);
/* Describes a single 2D point */
struct Point {
   int X;   /* X coordinate */
   int Y;   /* Y coordinate */
};
/* Describes a single 3D point in homogeneous coordinates */
struct Point3 {
   double X;   /* X coordinate */
   double Y;   /* Y coordinate */
   double Z;   /* Z coordinate */
   double W;
};
```

```
/* Describes a series of points (used to store a list of vertices that
   describe a polygon; each vertex is assumed to connect to the two adjacent
   vertices, and the last vertex is assumed to connect to the first) */
   struct PointListHeader {
   int Length;                /* # of points */
   struct Point * PointPtr;   /* pointer to list of points */
};
/* Describes beginning and ending X coordinates of a single horizontal line */
struct HLine {
   int XStart; /* X coordinate of leftmost pixel in line */
   int XEnd;   /* X coordinate of rightmost pixel in line */
};
/* Describes a Length-long series of horizontal lines, all assumed to be on
   contiguous scan lines starting at YStart and proceeding downward (describes
   a scan-converted polygon to low-level hardware-dependent drawing code) */
struct HLineList {
   int Length;                /* # of horizontal lines */
   int YStart;                /* Y coordinate of topmost line */
   struct HLine * HLinePtr;   /* pointer to list of horz lines */
};
struct Rect { int Left, Top, Right, Bottom; };
/* Structure describing one face of an object (one polygon) */
struct Face {
   int * VertNums;   /* pointer to vertex ptrs */
   int NumVerts;     /* # of vertices */
   int Color;        /* polygon color */
};
/* Structure describing an object */
struct Object {
   int NumVerts;
   struct Point3 * VertexList;
   struct Point3 * XformedVertexList;
   struct Point3 * ProjectedVertexList;
   struct Point * ScreenVertexList;
   int NumFaces;
   struct Face * FaceList;
};
extern void XformVec(double Xform[4][4], double * SourceVec, double * DestVec);
extern void ConcatXforms(double SourceXform1[4][4],
   double SourceXform2[4][4], double DestXform[4][4]);
extern void XformAndProjectPoly(double Xform[4][4],
   struct Point3 * Poly, int PolyLength, int Color);
extern int FillConvexPolygon(struct PointListHeader *, int, int, int);
extern void Set320x240Mode(void);
extern void ShowPage(unsigned int StartOffset);
extern void FillRectangleX(int StartX, int StartY, int EndX,
   int EndY, unsigned int PageBase, int Color);
extern void XformAndProjectPoints(double Xform[4][4],struct Object * ObjectToXform);
extern void DrawVisibleFaces(struct Object * ObjectToXform);
extern void AppendRotationX(double XformToChange[4][4], double Angle);
extern void AppendRotationY(double XformToChange[4][4], double Angle);
extern void AppendRotationZ(double XformToChange[4][4], double Angle);
extern int DisplayedPage, NonDisplayedPage;
extern struct Rect EraseRect[];
```

A Note on Rounding Negative Numbers

In the previous chapter, I added 0.5 and truncated in order to round values from floating-point to integer format. Here, in Listing 51.2, I've switched to adding 0.5

and using the **floor()** function. For positive values, the two approaches are equivalent; for negative values, only the **floor()** approach works properly.

Object Representation

Each object consists of a list of vertices and a list of faces, with the vertices of each face defined by pointers into the vertex list; this allows each vertex to be transformed exactly once, even though several faces may share a single vertex. Each object contains the vertices not only in their original, untransformed state, but in three other forms as well: transformed to view space, transformed and projected to screen space, and converted to screen coordinates. Earlier, we saw that it can be convenient to store the screen coordinates within the object, so that if the object hasn't moved with respect to the viewer, it can be redrawn without the need for recalculation, but why bother storing the view and screen space forms of the vertices as well?

The screen space vertices are useful for some sorts of hidden surface removal. For example, to determine whether two polygons overlap as seen by the viewer, you must first know how they look to the viewer, accounting for perspective; screen space provides that information. (So do the final screen coordinates, but with less accuracy, and without any Z information.) The view space vertices are useful for collision and proximity detection; screen space can't be used here, because objects are distorted by the perspective projection into screen space. World space would serve as well as view space for collision detection, but because it's possible to transform directly from object space to view space with a single matrix, it's often preferable to skip over world space. It's not mandatory that vertices be stored for all these different spaces, but the coordinates in all those spaces have to be calculated as intermediate steps anyway, so we might as well keep them around for those occasions when they're needed.