Chapter 52

Fast 3-D Animation: Meet X-Sharp

Chapter 52

The First Iteration of a Generalized 3-D Animation Package

Across the lake from Vermont, a few miles into upstate New York, the Ausable River has carved out a fairly impressive gorge known as "Ausable Chasm." Impressive for the East, anyway; you might think of it as the poor man's Grand Canyon. Some time back, I did the tour with my wife and five-year-old, and it was fun, although I confess that I didn't loosen my grip on my daughter's hand until we were on the bus and headed for home; that gorge is deep, and the railings tend to be of the single-bar, rusted-out variety.

New Yorkers can drive straight to this wonder of nature, but Vermonters must take their cars across on the ferry; the alternative is driving three hours around the south end of Lake Champlain. No problem; the ferry ride is an hour well spent on a beautiful lake. Or, rather, no problem—once you're on the ferry. Getting to New York is easy, but, as we found out, the line of cars waiting to come back from Ausable Chasm gets lengthy about mid-afternoon. The ferry can hold only so many cars, and we wound up spending an unexpected hour exploring the wonders of the ferry docks. Not a big deal, with a good-natured kid and an entertaining mom; we got ice cream, explored the beach, looked through binoculars, and told stories. It was a fun break, actually, and before we knew it, the ferry was steaming back to pick us up.

A friend of mine, an elementary-school teacher, helped take 65 sixth graders to Ausable Chasm. Never mind the potential for trouble with 65 kids loose on a ferry.

Never mind what it was like trying to herd that group around a gorge that looks like it was designed to swallow children and small animals without a trace. The hard part was getting back to the docks and finding they'd have to wait an hour for the next ferry. As my friend put it, "Let me tell you, an hour is an eternity with 65 sixth graders screaming the song 'You Are My Sunshine.'"

Apart from reminding you how lucky you are to be working in a quiet, air-conditioned room, in front of a gently humming computer, free to think deep thoughts and eat Cheetos to your heart's content, this story provides a useful perspective on the malleable nature of time. An hour isn't just an hour—it can be forever, or it can be the wink of an eye. Just think of the last hour you spent working under a deadline; I bet it went past in a flash. Which is not to say, mind you, that I recommend working in a bus full of screaming kids in order to make time pass more slowly; there are quality issues here as well.

In our 3-D animation work so far, we've used floating-point arithmetic. Floating-point arithmetic—even with a floating-point processor but especially *without* one—is the microcomputer animation equivalent of working in a school bus: It takes forever to do anything, and you just *know* you're never going to accomplish as much as you want to. In this chapter, we'll address fixed-point arithmetic, which will give us an instant order-of-magnitude performance boost. We'll also give our 3-D animation code a much more powerful and extensible framework, making it easy to add new and different sorts of objects. Taken together, these alterations will let us start to do some really interesting real-time animation.

This Chapter's Demo Program

Three-dimensional animation is a complicated business, and it takes an astonishing amount of functionality just to get off the launching pad: page flipping, polygon filling, clipping, transformations, list management, and so forth. I've been building toward a critical mass of animation functionality over the course of this book, and this chapter's code builds on the code from no fewer than five previous chapters. The code that's required in order to link this chapter's animation demo program is the following:

- Listing 50.1 from Chapter 50 (draw clipped line list);
- 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. Note that the **struct** keywords in **FillConvexPolygon( )** must be removed to reflect the switch to typedefs in the animation header file.

As always, all required files are in this chapter's subdirectory on the CD-ROM.

LISTING 52.1 L52-1.C

```
/* 3-D animation program to rotate 12 cubes. Uses fixed point. All C code
   tested with Borland C++ in C compilation mode and the small model. */

#include <conio.h>
#include <dos.h>
#include "polygon.h"

/* 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;
static unsigned int PageStartOffsets[2] =
   {PAGE0_START_OFFSET,PAGE1_START_OFFSET};
int DisplayedPage, NonDisplayedPage;
int RecalcAllXforms = 1, NumObjects = 0;
Xform WorldViewXform;   /* initialized from floats */
/* pointers to objects */
Object *ObjectList[MAX_OBJECTS];

void main() {
   int Done = 0, i;
   Object *ObjectPtr;
   union REGS regset;

   InitializeFixedPoint(); /* set up fixed-point data */
   InitializeCubes();      /* set up cubes and add them to object list; other
                              objects would be initialized now, if there were any */
   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 {
      /* For each object, regenerate viewing info, if necessary */
      for (i=0; i<NumObjects; i++) {
         if ((ObjectPtr = ObjectList[i])->RecalcXform ||
               RecalcAllXforms) {
            ObjectPtr->RecalcFunc(ObjectPtr);
            ObjectPtr->RecalcXform = 0;
         }
      }
      RecalcAllXforms = 0;
      CurrentPageBase =    /* select other page for drawing to */
            PageStartOffsets[NonDisplayedPage = DisplayedPage ^ 1];
      /* For each object, clear the portion of the non-displayed page
         that was drawn to last time, then reset the erase extent */
      for (i=0; i<NumObjects; i++) {
         ObjectPtr = ObjectList[i];
         FillRectangleX(ObjectPtr->EraseRect[NonDisplayedPage].Left,
            ObjectPtr->EraseRect[NonDisplayedPage].Top,
            ObjectPtr->EraseRect[NonDisplayedPage].Right,
            ObjectPtr->EraseRect[NonDisplayedPage].Bottom,
            CurrentPageBase, 0);
         ObjectPtr->EraseRect[NonDisplayedPage].Left =
              ObjectPtr->EraseRect[NonDisplayedPage].Top = 0x7FFF;
         ObjectPtr->EraseRect[NonDisplayedPage].Right =
               ObjectPtr->EraseRect[NonDisplayedPage].Bottom = 0;
      }
      /* Draw all objects */
      for (i=0; i<NumObjects; i++)
         ObjectList[i]->DrawFunc(ObjectList[i]);
      /* Flip to display the page into which we just drew */
      ShowPage(PageStartOffsets[DisplayedPage = NonDisplayedPage]);
      /* Move and reorient each object */
      for (i=0; i<NumObjects; i++)
         ObjectList[i]->MoveFunc(ObjectList[i]);
      if (kbhit())
         if (getch() == 0x1B) Done = 1;   /* Esc to exit */
   } while (!Done);
   /* Return to text mode and exit */
   regset.x.ax = 0x0003;   /* AL = 3 selects 80x25 text mode */
   int86(0x10, &regset, &regset);
   exit(1);
}
```

LISTING 52.2 L52-2.C

```
/* Transforms all vertices in the specified polygon-based object into view
   space, then perspective projects them to screen space and maps them to screen
   coordinates, storing results in the object. Recalculates object->view
   transformation because only if transform changes would we bother
   to retransform the vertices. */

#include <math.h>
#include "polygon.h"

void XformAndProjectPObject(PObject * ObjectToXform)
{
   int i, NumPoints = ObjectToXform->NumVerts;
   Point3 * Points = ObjectToXform->VertexList;
   Point3 * XformedPoints = ObjectToXform->XformedVertexList;
   Point3 * ProjectedPoints = ObjectToXform->ProjectedVertexList;
   Point * ScreenPoints = ObjectToXform->ScreenVertexList;

   /* Recalculate the object->view transform */
   ConcatXforms(WorldViewXform, ObjectToXform->XformToWorld,
                ObjectToXform->XformToView);
   /* Apply that new transformation and project the points */
   for (i=0; i<NumPoints; i++, Points++, XformedPoints++,
         ProjectedPoints++, ScreenPoints++) {
      /* Transform to view space */
      XformVec(ObjectToXform->XformToView, (Fixedpoint *) Points,
            (Fixedpoint *) XformedPoints);
      /* Perspective-project to screen space */
      ProjectedPoints->X =
            FixedMul(FixedDiv(XformedPoints->X, XformedPoints->Z),
            DOUBLE_TO_FIXED(PROJECTION_RATIO * (SCREEN_WIDTH/2)));
      ProjectedPoints->Y =
            FixedMul(FixedDiv(XformedPoints->Y, XformedPoints->Z),
            DOUBLE_TO_FIXED(PROJECTION_RATIO * (SCREEN_WIDTH/2)));
      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 polygon
         filler. Add in half the screen width and height to center on screen. */
      ScreenPoints->X = ((int) ((ProjectedPoints->X +
            DOUBLE_TO_FIXED(0.5)) >> 16)) + SCREEN_WIDTH/2;
      ScreenPoints->Y = (-((int) ((ProjectedPoints->Y +
            DOUBLE_TO_FIXED(0.5)) >> 16))) + SCREEN_HEIGHT/2;
   }
}
```

LISTING 52.3 L52-3.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 transformation in
   XformToChange, placing the result back into XformToChange. */
void AppendRotationX(Xform XformToChange, double Angle)
{
   Fixedpoint Temp10, Temp11, Temp12, Temp20, Temp21, Temp22;
   Fixedpoint CosTemp = DOUBLE_TO_FIXED(cos(Angle));
   Fixedpoint SinTemp = DOUBLE_TO_FIXED(sin(Angle));

   /* Calculate the new values of the six affected matrix entries */
   Temp10 = FixedMul(CosTemp, XformToChange[1][0]) +
         FixedMul(-SinTemp, XformToChange[2][0]);
   Temp11 = FixedMul(CosTemp, XformToChange[1][1]) +
         FixedMul(-SinTemp, XformToChange[2][1]);
   Temp12 = FixedMul(CosTemp, XformToChange[1][2]) +
         FixedMul(-SinTemp, XformToChange[2][2]);
   Temp20 = FixedMul(SinTemp, XformToChange[1][0]) +
         FixedMul(CosTemp, XformToChange[2][0]);
   Temp21 = FixedMul(SinTemp, XformToChange[1][1]) +
         FixedMul(CosTemp, XformToChange[2][1]);
   Temp22 = FixedMul(SinTemp, XformToChange[1][2]) +
         FixedMul(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 transformation in
   XformToChange, placing the result back into XformToChange. */
void AppendRotationY(Xform XformToChange, double Angle)
{
   Fixedpoint Temp00, Temp01, Temp02, Temp20, Temp21, Temp22;
   Fixedpoint CosTemp = DOUBLE_TO_FIXED(cos(Angle));
   Fixedpoint SinTemp = DOUBLE_TO_FIXED(sin(Angle));

   /* Calculate the new values of the six affected matrix entries */
   Temp00 = FixedMul(CosTemp, XformToChange[0][0]) +
         FixedMul(SinTemp, XformToChange[2][0]);
   Temp01 = FixedMul(CosTemp, XformToChange[0][1]) +
         FixedMul(SinTemp, XformToChange[2][1]);
   Temp02 = FixedMul(CosTemp, XformToChange[0][2]) +
         FixedMul(SinTemp, XformToChange[2][2]);
   Temp20 = FixedMul(-SinTemp, XformToChange[0][0]) +
         FixedMul( CosTemp, XformToChange[2][0]);
   Temp21 = FixedMul(-SinTemp, XformToChange[0][1]) +
         FixedMul(CosTemp, XformToChange[2][1]);
   Temp22 = FixedMul(-SinTemp, XformToChange[0][2]) +
         FixedMul(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 transformation in
   XformToChange, placing the result back into XformToChange. */
void AppendRotationZ(Xform XformToChange, double Angle)
{
   Fixedpoint Temp00, Temp01, Temp02, Temp10, Temp11, Temp12;
   Fixedpoint CosTemp = DOUBLE_TO_FIXED(cos(Angle));
   Fixedpoint SinTemp = DOUBLE_TO_FIXED(sin(Angle));

   /* Calculate the new values of the six affected matrix entries */
   Temp00 = FixedMul(CosTemp, XformToChange[0][0]) +
         FixedMul(-SinTemp, XformToChange[1][0]);
   Temp01 = FixedMul(CosTemp, XformToChange[0][1]) +
         FixedMul(-SinTemp, XformToChange[1][1]);
   Temp02 = FixedMul(CosTemp, XformToChange[0][2]) +
         FixedMul(-SinTemp, XformToChange[1][2]);
   Temp10 = FixedMul(SinTemp, XformToChange[0][0]) +
         FixedMul(CosTemp, XformToChange[1][0]);
   Temp11 = FixedMul(SinTemp, XformToChange[0][1]) +
         FixedMul(CosTemp, XformToChange[1][1]);
   Temp12 = FixedMul(SinTemp, XformToChange[0][2]) +
         FixedMul(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 52.4 L52-4.C

```
/* Fixed point matrix arithmetic functions. */

#include "polygon.h"

/* Matrix multiplies Xform by SourceVec, and stores the result in DestVec.
   Multiplies a 4x4 matrix times a 4x1 matrix; the result is a 4x1 matrix. Cheats
   by assuming the W coord is 1 and bottom row of matrix is 0 0 0 1, and doesn't
   bother to set the W coordinate of the destination. */
void XformVec(Xform WorkingXform, Fixedpoint *SourceVec,
   Fixedpoint *DestVec)
{
   int i;

   for (i=0; i<3; i++)
      DestVec[i] = FixedMul(WorkingXform[i][0], SourceVec[0]) +
            FixedMul(WorkingXform[i][1], SourceVec[1]) +
            FixedMul(WorkingXform[i][2], SourceVec[2]) +
            WorkingXform[i][3];   /* no need to multiply by W = 1 */
}

/* Matrix multiplies SourceXform1 by SourceXform2 and stores result in
   DestXform. Multiplies a 4x4 matrix times a 4x4 matrix; result is a 4x4 matrix.
   Cheats by assuming bottom row of each matrix is 0 0 0 1, and doesn't bother
   to set the bottom row of the destination. */
void ConcatXforms(Xform SourceXform1, Xform SourceXform2,
   Xform DestXform)
{
   int i, j;

   for (i=0; i<3; i++) {
      for (j=0; j<4; j++)
```

```
         DestXform[i][j] =
               FixedMul(SourceXform1[i][0], SourceXform2[0][j]) +
               FixedMul(SourceXform1[i][1], SourceXform2[1][j]) +
               FixedMul(SourceXform1[i][2], SourceXform2[2][j]) +
               SourceXform1[i][3];
   }
}
```

LISTING 52.5 L52-5.C

```
/* Set up basic data that needs to be in fixed point, to avoid data
   definition hassles. */

#include "polygon.h"

/* All vertices in the basic cube */
static IntPoint3 IntCubeVerts[NUM_CUBE_VERTS] = {
   {15,15,15},{15,15,-15},{15,-15,15},{15,-15,-15},
   {-15,15,15},{-15,15,-15},{-15,-15,15},{-15,-15,-15} };
/* Transformation from world space into view space (no transformation,
   currently) */
static int IntWorldViewXform[3][4] = {
   {1,0,0,0}, {0,1,0,0}, {0,0,1,0}};

void InitializeFixedPoint()
{
   int i, j;

   for (i=0; i<3; i++)
      for (j=0; j<4; j++)
         WorldViewXform[i][j] = INT_TO_FIXED(IntWorldViewXform[i][j]);
   for (i=0; i<NUM_CUBE_VERTS; i++) {
      CubeVerts[i].X = INT_TO_FIXED(IntCubeVerts[i].X);
      CubeVerts[i].Y = INT_TO_FIXED(IntCubeVerts[i].Y);
      CubeVerts[i].Z = INT_TO_FIXED(IntCubeVerts[i].Z);
   }
}
```

LISTING 52.6 L52-6.C

```
/* Rotates and moves a polygon-based object around the three axes.
   Movement is implemented only along the Z axis currently. */

#include "polygon.h"

void RotateAndMovePObject(PObject * ObjectToMove)
{
   if (--ObjectToMove->RDelayCount == 0) {   /* rotate */
      ObjectToMove->RDelayCount = ObjectToMove->RDelayCountBase;
      if (ObjectToMove->Rotate.RotateX != 0.0)
         AppendRotationX(ObjectToMove->XformToWorld,
               ObjectToMove->Rotate.RotateX);
      if (ObjectToMove->Rotate.RotateY != 0.0)
         AppendRotationY(ObjectToMove->XformToWorld,
               ObjectToMove->Rotate.RotateY);
      if (ObjectToMove->Rotate.RotateZ != 0.0)
         AppendRotationZ(ObjectToMove->XformToWorld,
               ObjectToMove->Rotate.RotateZ);
      ObjectToMove->RecalcXform = 1;
   }
   /* Move in Z, checking for bouncing and stopping */
   if (--ObjectToMove->MDelayCount == 0) {
      ObjectToMove->MDelayCount = ObjectToMove->MDelayCountBase;
      ObjectToMove->XformToWorld[2][3] += ObjectToMove->Move.MoveZ;
      if (ObjectToMove->XformToWorld[2][3]>ObjectToMove->Move.MaxZ)
         ObjectToMove->Move.MoveZ = 0; /* stop if close enough */
      ObjectToMove->RecalcXform = 1;
   }
}
```

LISTING 52.7 L52-7.C

```
/* Draws all visible faces in specified polygon-based object. Object must have
   previously been transformed and projected, so that ScreenVertexList array is
   filled in. */

#include "polygon.h"

void DrawPObject(PObject * ObjectToXform)
{
   int i, j, NumFaces = ObjectToXform->NumFaces, NumVertices;
   int * VertNumsPtr;
   Face * FacePtr = ObjectToXform->FaceList;
   Point * ScreenPoints = ObjectToXform->ScreenVertexList;
   long v1, v2, w1, w2;
   Point Vertices[MAX_POLY_LENGTH];
   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 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 object later */
         for (j=0; j<NumVertices; j++) {
            if (Vertices[j].X >
                  ObjectToXform->EraseRect[NonDisplayedPage].Right)
               if (Vertices[j].X < SCREEN_WIDTH)
                  ObjectToXform->EraseRect[NonDisplayedPage].Right =
                        Vertices[j].X;
               else ObjectToXform->EraseRect[NonDisplayedPage].Right =
                     SCREEN_WIDTH;
            if (Vertices[j].Y >
                  ObjectToXform->EraseRect[NonDisplayedPage].Bottom)
               if (Vertices[j].Y < SCREEN_HEIGHT)
                  ObjectToXform->EraseRect[NonDisplayedPage].Bottom =
                        Vertices[j].Y;
               else ObjectToXform->EraseRect[NonDisplayedPage].Bottom=
                     SCREEN_HEIGHT;
            if (Vertices[j].X <
                  ObjectToXform->EraseRect[NonDisplayedPage].Left)
               if (Vertices[j].X > 0)
                  ObjectToXform->EraseRect[NonDisplayedPage].Left =
                        Vertices[j].X;
               else ObjectToXform->EraseRect[NonDisplayedPage].Left=0;
            if (Vertices[j].Y <
                  ObjectToXform->EraseRect[NonDisplayedPage].Top)
               if (Vertices[j].Y > 0)
                  ObjectToXform->EraseRect[NonDisplayedPage].Top =
                        Vertices[j].Y;
               else ObjectToXform->EraseRect[NonDisplayedPage].Top=0;
         }
         /* Draw the polygon */
         DRAW_POLYGON(Vertices, NumVertices, FacePtr->Color, 0, 0);
      }
   }
}
```

LISTING 52.8 L52-8.C

```
/* Initializes the cubes and adds them to the object list. */

#include <stdlib.h>
#include <math.h>
#include "polygon.h"

#define ROT_6  (M_PI / 30.0)       /* rotate 6 degrees at a time */
#define ROT_3  (M_PI / 60.0)       /* rotate 3 degrees at a time */
#define ROT_2  (M_PI / 90.0)       /* rotate 2 degrees at a time */
#define NUM_CUBES 12               /* # of cubes */

Point3 CubeVerts[NUM_CUBE_VERTS]; /* set elsewhere, from floats */
/* vertex indices for individual cube 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};
static int *VertNumList[]={Face1, Face2, Face3, Face4, Face5, Face6};
static int VertsInFace[]={ sizeof(Face1)/sizeof(int),
   sizeof(Face2)/sizeof(int), sizeof(Face3)/sizeof(int),
   sizeof(Face4)/sizeof(int), sizeof(Face5)/sizeof(int),
   sizeof(Face6)/sizeof(int) };
/* X, Y, Z rotations for cubes */
static RotateControl InitialRotate[NUM_CUBES] = {
   {0.0,ROT_6,ROT_6},{ROT_3,0.0,ROT_3},{ROT_3,ROT_3,0.0},
   {ROT_3,-ROT_3,0.0},{-ROT_3,ROT_2,0.0},{-ROT_6,-ROT_3,0.0},
   {ROT_3,0.0,-ROT_6},{-ROT_2,0.0,ROT_3},{-ROT_3,0.0,-ROT_3},
   {0.0,ROT_2,-ROT_2},{0.0,-ROT_3,ROT_3},{0.0,-ROT_6,-ROT_6},};
static MoveControl InitialMove[NUM_CUBES] = {
   {0,0,80,0,0,0,0,0,-350},{0,0,80,0,0,0,0,0,-350},
   {0,0,80,0,0,0,0,0,-350},{0,0,80,0,0,0,0,0,-350},
   {0,0,80,0,0,0,0,0,-350},{0,0,80,0,0,0,0,0,-350},
   {0,0,80,0,0,0,0,0,-350},{0,0,80,0,0,0,0,0,-350},
   {0,0,80,0,0,0,0,0,-350},{0,0,80,0,0,0,0,0,-350},
   {0,0,80,0,0,0,0,0,-350},{0,0,80,0,0,0,0,0,-350}, };
/* face colors for various cubes */
static int Colors[NUM_CUBES][NUM_CUBE_FACES] = {
   {15,14,12,11,10,9},{1,2,3,4,5,6},{35,37,39,41,43,45},
   {47,49,51,53,55,57},{59,61,63,65,67,69},{71,73,75,77,79,81},
   {83,85,87,89,91,93},{95,97,99,101,103,105},
   {107,109,111,113,115,117},{119,121,123,125,127,129},
   {131,133,135,137,139,141},{143,145,147,149,151,153} };
/* starting coordinates for cubes in world space */
static int CubeStartCoords[NUM_CUBES][3] = {
   {100,0,-6000},  {100,70,-6000}, {100,-70,-6000}, {33,0,-6000},
   {33,70,-6000},  {33,-70,-6000}, {-33,0,-6000},   {-33,70,-6000},
   {-33,-70,-6000},{-100,0,-6000}, {-100,70,-6000}, {-100,-70,-6000}};
/* delay counts (speed control) for cubes */
static int InitRDelayCounts[NUM_CUBES] = {1,2,1,2,1,1,1,1,1,2,1,1};
static int BaseRDelayCounts[NUM_CUBES] = {1,2,1,2,2,1,1,1,2,2,2,1};
static int InitMDelayCounts[NUM_CUBES] = {1,1,1,1,1,1,1,1,1,1,1,1};
static int BaseMDelayCounts[NUM_CUBES] = {1,1,1,1,1,1,1,1,1,1,1,1};

void InitializeCubes()
{
   int i, j, k;
   PObject *WorkingCube;

   for (i=0; i<NUM_CUBES; i++) {
      if ((WorkingCube = malloc(sizeof(PObject))) == NULL) {
         printf("Couldn't get memory\n"); exit(1); }
      WorkingCube->DrawFunc = DrawPObject;
      WorkingCube->RecalcFunc = XformAndProjectPObject;
      WorkingCube->MoveFunc = RotateAndMovePObject;
      WorkingCube->RecalcXform = 1;
      for (k=0; k<2; k++) {
         WorkingCube->EraseRect[k].Left =
            WorkingCube->EraseRect[k].Top = 0x7FFF;
         WorkingCube->EraseRect[k].Right = 0;
         WorkingCube->EraseRect[k].Bottom = 0;
      }
      WorkingCube->RDelayCount = InitRDelayCounts[i];
      WorkingCube->RDelayCountBase = BaseRDelayCounts[i];
      WorkingCube->MDelayCount = InitMDelayCounts[i];
      WorkingCube->MDelayCountBase = BaseMDelayCounts[i];
      /* Set the object->world xform to none */
      for (j=0; j<3; j++)
         for (k=0; k<4; k++)
            WorkingCube->XformToWorld[j][k] = INT_TO_FIXED(0);
      WorkingCube->XformToWorld[0][0] =
         WorkingCube->XformToWorld[1][1] =
         WorkingCube->XformToWorld[2][2] =
         WorkingCube->XformToWorld[3][3] = INT_TO_FIXED(1);
      /* Set the initial location */
      for (j=0; j<3; j++) WorkingCube->XformToWorld[j][3] =
            INT_TO_FIXED(CubeStartCoords[i][j]);
      WorkingCube->NumVerts = NUM_CUBE_VERTS;
      WorkingCube->VertexList = CubeVerts;
      WorkingCube->NumFaces = NUM_CUBE_FACES;
      WorkingCube->Rotate = InitialRotate[i];
      WorkingCube->Move.MoveX = INT_TO_FIXED(InitialMove[i].MoveX);
      WorkingCube->Move.MoveY = INT_TO_FIXED(InitialMove[i].MoveY);
      WorkingCube->Move.MoveZ = INT_TO_FIXED(InitialMove[i].MoveZ);
      WorkingCube->Move.MinX = INT_TO_FIXED(InitialMove[i].MinX);
      WorkingCube->Move.MinY = INT_TO_FIXED(InitialMove[i].MinY);
      WorkingCube->Move.MinZ = INT_TO_FIXED(InitialMove[i].MinZ);
      WorkingCube->Move.MaxX = INT_TO_FIXED(InitialMove[i].MaxX);
      WorkingCube->Move.MaxY = INT_TO_FIXED(InitialMove[i].MaxY);
      WorkingCube->Move.MaxZ = INT_TO_FIXED(InitialMove[i].MaxZ);
      if ((WorkingCube->XformedVertexList =
            malloc(NUM_CUBE_VERTS*sizeof(Point3))) == NULL) {
         printf("Couldn't get memory\n"); exit(1); }
      if ((WorkingCube->ProjectedVertexList =
            malloc(NUM_CUBE_VERTS*sizeof(Point3))) == NULL) {
         printf("Couldn't get memory\n"); exit(1); }
      if ((WorkingCube->ScreenVertexList =
            malloc(NUM_CUBE_VERTS*sizeof(Point))) == NULL) {
         printf("Couldn't get memory\n"); exit(1); }
      if ((WorkingCube->FaceList =
            malloc(NUM_CUBE_FACES*sizeof(Face))) == NULL) {
         printf("Couldn't get memory\n"); exit(1); }
      /* Initialize the faces */
      for (j=0; j<NUM_CUBE_FACES; j++) {
         WorkingCube->FaceList[j].VertNums = VertNumList[j];
         WorkingCube->FaceList[j].NumVerts = VertsInFace[j];
         WorkingCube->FaceList[j].Color = Colors[i][j];
      }
      ObjectList[NumObjects++] = (Object *)WorkingCube;
   }
}
```

LISTING 52.9 L52-9.ASM

```
; 386-specific fixed point multiply and divide.
;
; C near-callable as: Fixedpoint FixedMul(Fixedpoint M1, Fixedpoint M2);
;                 Fixedpoint FixedDiv(Fixedpoint Dividend, Fixedpoint Divisor);
;
; Tested with TASM
;
        .model small
        .386
        .code
        public  _FixedMul,_FixedDiv
; Multiplies two fixed-point values together.
FMparms struc
        dw      2 dup(?)          ;return address & pushed BP
M1      dd      ?
M2      dd      ?
FMparms ends
        align   2
_FixedMul       proc    near
        push    bp
        mov     bp,sp
        mov     eax,[bp+M1]
        imul    dword ptr [bp+M2] ;multiply
        add     eax,8000h         ;round by adding 2^(-16)
        adc     edx,0             ;whole part of result is in DX
        shr     eax,16            ;put the fractional part in AX
        pop     bp
        ret
_FixedMul       endp
; Divides one fixed-point value by another.
FDparms struc
         dw     2 dup(?)          ;return address & pushed BP
Dividend dd     ?
Divisor  dd     ?
FDparms ends
        align   2
```

```
_FixedDiv       proc    near
        push    bp
        mov     bp,sp
        sub     cx,cx           ;assume positive result
        mov     eax,[bp+Dividend]
        and     eax,eax         ;positive dividend?
        jns     FDP1            ;yes
        inc     cx              ;mark it's a negative dividend
        neg     eax             ;make the dividend positive
FDP1:   sub     edx,edx         ;make it a 64-bit dividend, then shift
                                ; left 16 bits so that result will be in EAX
        rol     eax,16          ;put fractional part of dividend in
                                ; high word of EAX
        mov     dx,ax           ;put whole part of dividend in DX
        sub     ax,ax           ;clear low word of EAX
        mov     ebx,dword ptr [bp+Divisor]
        and     ebx,ebx         ;positive divisor?
        jns     FDP2            ;yes
        dec     cx              ;mark it's a negative divisor
        neg     ebx             ;make divisor positive
FDP2:   div     ebx             ;divide
        shr     ebx,1           ;divisor/2, minus 1 if the divisor is
        adc     ebx,0           ; even
        dec     ebx
        cmp     ebx,edx         ;set Carry if remainder is at least
        adc     eax,0           ; half as large as the divisor, then
                                ; use that to round up if necessary
        and     cx,cx           ;should the result be made negative?
        jz      FDP3            ;no
        neg     eax             ;yes, negate it
FDP3:   mov     edx,eax         ;return result in DX:AX; fractional
                                ; part is already in AX
        shr     edx,16          ;whole part of result in DX
        pop     bp
        ret
_FixedDiv       endp
        end
```

LISTING 52.10 POLYGON.H

```
/* POLYGON.H: Header file for polygon-filling code, also includes
   a number of useful items for 3-D animation. */

#define MAX_OBJECTS  100   /* max simultaneous # objects supported */
#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)
#define NUM_CUBE_VERTS 8              /* # of vertices per cube */
#define NUM_CUBE_FACES 6              /* # of faces per cube */
/* 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);
```

```
#define INT_TO_FIXED(x) (((long)(int)x) << 16)
#define DOUBLE_TO_FIXED(x) ((long) (x * 65536.0 + 0.5))

typedef long Fixedpoint;
typedef Fixedpoint Xform[3][4];
/* Describes a single 2D point */
typedef struct { int X; int Y; } Point;
/* Describes a single 3D point in homogeneous coordinates; the W
   coordinate isn't present, though; assumed to be 1 and implied */
typedef struct { Fixedpoint X, Y, Z; } Point3;
typedef struct { int X; int Y; int Z; } IntPoint3;
/* 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; last vertex is assumed to connect to first) */
typedef struct { int Length; Point * PointPtr; } PointListHeader;
/* Describes the beginning and ending X coordinates of a single
   horizontal line */
typedef struct { int XStart; int XEnd; } HLine;
/* Describes a Length-long series of horizontal lines, all assumed to
   be on contiguous scan lines starting at YStart and proceeding
   downward (used to describe a scan-converted polygon to the
   low-level hardware-dependent drawing code). */
typedef struct { int Length; int YStart; HLine * HLinePtr;} HLineList;
typedef struct { int Left, Top, Right, Bottom; } Rect;
/* structure describing one face of an object (one polygon) */
typedef struct { int * VertNums; int NumVerts; int Color; }  Face;
typedef struct { double RotateX, RotateY, RotateZ; } RotateControl;
typedef struct { Fixedpoint MoveX, MoveY, MoveZ, MinX, MinY, MinZ,
   MaxX, MaxY, MaxZ; } MoveControl;
/* fields common to every object */
#define BASE_OBJECT                                             \
   void (*DrawFunc)();     /* draws object */                   \
   void (*RecalcFunc)();   /* prepares object for drawing */    \
   void (*MoveFunc)();     /* moves object */                   \
   int RecalcXform;        /* 1 to indicate need to recalc */   \
   Rect EraseRect[2];      /* rectangle to erase in each page */
/* basic object */
typedef struct { BASE_OBJECT } Object;
/* structure describing a polygon-based object */
typedef struct {
   BASE_OBJECT
   int RDelayCount, RDelayCountBase; /* controls rotation speed */
   int MDelayCount, MDelayCountBase; /* controls movement speed */
   Xform XformToWorld;        /* transform from object->world space */
   Xform XformToView;         /* transform from object->view space */
   RotateControl Rotate;      /* controls rotation change over time */
   MoveControl Move;          /* controls object movement over time */
   int NumVerts;              /* # vertices in VertexList */
   Point3 * VertexList;       /* untransformed vertices */
   Point3 * XformedVertexList;   /* transformed into view space */
   Point3 * ProjectedVertexList; /* projected into screen space */
   Point * ScreenVertexList;     /* converted to screen coordinates */
   int NumFaces;              /* # of faces in object */
   Face * FaceList;           /* pointer to face info */
} PObject;

extern void XformVec(Xform, Fixedpoint *, Fixedpoint *);
extern void ConcatXforms(Xform, Xform, Xform);
extern int FillConvexPolygon(PointListHeader *, int, int, int);
extern void Set320x240Mode(void);
extern void ShowPage(unsigned int);
extern void FillRectangleX(int, int, int, int, unsigned int, int);
extern void XformAndProjectPObject(PObject *);
extern void DrawPObject(PObject *);
extern void AppendRotationX(Xform, double);
extern void AppendRotationY(Xform, double);
extern void AppendRotationZ(Xform, double);
extern near Fixedpoint FixedMul(Fixedpoint, Fixedpoint);
extern near Fixedpoint FixedDiv(Fixedpoint, Fixedpoint);
extern void InitializeFixedPoint(void);
extern void RotateAndMovePObject(PObject *);
extern void InitializeCubes(void);
extern int DisplayedPage, NonDisplayedPage, RecalcAllXforms;
extern int NumObjects;
extern Xform WorldViewXform;
extern Object *ObjectList[];
extern Point3 CubeVerts[];
```

A New Animation Framework: X-Sharp

Listings 52.1 through 52.10 shown earlier represent not merely faster animation in library form, but also a nearly complete, extensible, data-driven animation framework. Whereas much of the earlier animation code I've presented in this book was hardwired to demonstrate certain concepts, this chapter's code is intended to serve as the basis for a solid animation package. Objects are stored, in their entirety, in customizable structures; new structures can be devised for new sorts of objects. Drawing, preparing for drawing, and moving are all vectored functions, so that variations such as shading or texturing, or even radically different sorts of graphics objects, such as scaled bitmaps, could be supported. The cube initialization is entirely data driven; more or different cubes, or other sorts of convex polyhedrons, could be added by simply changing the initialization data in Listing 52.8.

Somewhere along the way in writing the material that became this section of the book, I realized that I had a generally useful animation package by the tail and gave it a name: X-Sharp. (*X* for Mode X, *sharp* because good animation looks sharp, and, well, who would want a flat animation package?)

Note that the X-Sharp library as presented in this chapter (and, indeed, in this book) is not a fully complete 3-D library. Movement is supported only along the Z axis in this chapter's version, and then in a non-general fashion. More interesting movement isn't supported at this point because of one of the two missing features in X-Sharp: hidden-surface removal. (The other missing feature is general 3-D clipping.) Without hidden surface removal, nothing can safely overlap. It would actually be easy enough to perform hidden-surface removal by keeping the cubes in different Z bands and drawing them back to front, but this gets into sorting and list issues, and is not a complete solution—and I've crammed as much as will fit into one chapter's code, anyway.

I'm working toward a goal in this last section of the book, and there are many lessons to be learned and stories to be told along the way. So as X-Sharp grows, you'll find its evolving implementations in the chapter subdirectories on the listings diskette. This chapter's subdirectory, for example, contains the self-extracting archive file XSHARP14.EXE, (to extract its contents you simply run it as though it were a program) and the code in that archive is the code I'm speaking of specifically in this chapter, with all the limitations mentioned above. Chapter 53's subdirectory, however, contains the file XSHARP15.EXE, which is the next step in the evolution of X-Sharp, and it is the version that I'll be specifically talking about in that chapter. Later chapters will have their own implementations in their respective chapter subdirectories, in files of the form XSHARPxx.EXE, where xx is an ascending number indicating the version. The final and most recent X-Sharp version will be present in its own subdirectory called XSHARP22. If you're intending to use X-Sharp in a real project, use the most recent version to be sure that you avail yourself of all new features and bug fixes.

Three Keys to Realtime Animation Performance

As of the previous chapter, we were at the point where we could rotate, move, and draw a solid cube in real time. Not too shabby...but the code I'm presenting in this chapter goes a bit further, rotating 12 solid cubes at an update rate of about 15 frames per second (fps) on a 20 MHz 386 with a slow VGA. That's 12 transformation matrices, 72 polygons, and 96 vertices being handled in real time; not Star Wars, granted, but a giant step beyond a single cube. Run the program if you get a chance; you may be surprised at just how effective this level of animation is. I'd like to point out, in case anyone missed it, that this is fully *general* 3-D. I'm not using any shortcuts or tricks, like prestoring coordinates or pregenerating bitmaps; if you were to feed in different rotations or vertices, the animation would change accordingly.

The keys to the performance increase manifested in this chapter's code are three. The first key is fixed-point arithmetic. In the previous two chapters, we worked with floating-point coordinates and transformation matrices. Those values are now stored as 32-bit fixed-point numbers, in the form 16.16 (16 bits of whole number, 16 bits of fraction). 32-bit fixed-point numbers allow sufficient precision for 3-D animation, but can be manipulated with fast integer operations, rather than by slow floating-point processor operations or excruciatingly slow floating-point emulator operations. Although the speed advantage of fixed-point varies depending on the operation, on the processor, and on whether or not a coprocessor is present, fixed-point multiplication can be as much as 100 times faster than the emulated floating-point equivalent. (I'd like to take a moment to thank Chris Hecker for his invaluable input in this area.)

The second performance key is the use of the 386's native 32-bit multiply and divide instructions. C compilers operating in real mode call library routines to perform multiplications and divisions involving 32-bit values, and those library functions are fairly slow, especially for division. On a 386, 32-bit multiplication and division can be handled with the bit of code in Listing 52.9—and most of even that code is only for rounding.

The third performance key is maintaining and operating on only the relevant portions of transformation matrices and coordinates. The bottom row of every transformation matrix we'll use (in this book) is [0 0 0 1], so why bother using or recalculating it when concatenating transforms and transforming points? Likewise for the fourth element of a 3-D vector in homogeneous coordinates, which is always 1. Basically, transformation matrices are treated as consisting of a 3×3 rotation matrix and a 3×1 translation vector, and coordinates are treated as 3×1 vectors. This saves a great many multiplications in the course of transforming each point.

Just for fun, I reimplemented the animation of Listings 52.1 through 52.10 with floating-point instructions. Together, the preceeding optimizations improve the performance of the entire animation—including drawing time and overhead, and not just math—by more than ten times over the code that uses the floating-point emulator. Amazing what one can accomplish with a few dozen lines of assembly and a switch in number format, isn't it? Note that no assembly code other than the native 386 multiply and divide is used in Listings 52.1 through 52.10, although the polygon fill code is of course mostly in assembly; we've achieved 12 cubes animated at 15 fps while doing the 3-D work almost entirely in Borland C++, and we're *still* doing sine and cosine via the floating-point emulator. Happily, we're still nowhere near the upper limit on the animation potential of the PC.

Drawbacks

The techniques we've used to turbocharge 3-D animation are very powerful, but there's a dark side to them as well. Obviously, native 386 instructions won't work on 8088 and 286 machines. That's rectifiable; equivalent multiplication and division routines could be implemented for real mode and performance would still be reasonable. It sure is nice to be able to plug in a 32-bit **IMUL** or **DIV** and be done with it, though. More importantly, 32-bit fixed-point arithmetic has limitations in range and accuracy. Points outside a 64K×64K×64K space can't be handled, imprecision tends to creep in over the course of multiple matrix concatenations, and it's quite possible to generate the dreaded divide by 0 interrupt if Z coordinates with absolute values less than one are used.

I don't have space to discuss these issues in detail, but here are some brief thoughts: The working 64K×64K×64K fixed-point space can be paged into a larger virtual space. Imprecision of a pixel or two rarely matters in terms of display quality, and deterioration of concatenated rotations can be corrected by restoring orthogonality, for example by periodically calculating one row of the matrix as the cross-product of the other two (forcing it to be perpendicular to both). Alternatively, transformations can be calculated from scratch each time an object or the viewer moves, so there's no chance for cumulative error. 3-D clipping with a front clip plane of -1 or less can prevent divide overflow.

Where the Time Goes

The distribution of execution time in the animation code is no longer wildly biased toward transformation, but sine and cosine are certainly still sucking up cycles. Likewise, the overhead in the calls to **FixedMul()** and **FixedDiv()** is costly. Much of this is correctable with a little carefully crafted assembly language and a lookup table; I'll provide that shortly.

Regardless, with this chapter we have made the critical jump to a usable level of performance and a serviceable general-purpose framework. From here on out, it's the fun stuff.