The Michael Abrash
GRAPHICS PROGRAMMING COLUMNS

from the Dr. Dobb's Journal
Feb 1991 - Feb 1993

Contents

The polygon primeval	3	*(Feb 91)*
80x86 optimization: aim down the middle and pray	13	*(Mar 91)*
Fast convex polygons	23	*(Mar 91)*
The virtues of inexpensive approximation: the Edsun Continuous Edge Graphics DAC	34	*(Apr 91)*
Further ruminations of the Edsun CEG/DAC	41	*(May 91)*
Of songs, taxes, and the simplicity of complex polygons	51	*(Jun 91)*
Mode X: 256-color VGA magic	63	*(Jul 91)*
More undocumented 256-color VGA magic	75	*Aug 91)*
256-color VGA animation	86	*(Sep 91)*
The virtues of affordable technology: the Sierra Hicolor DAC	96	*(Oct 91)*
Antialiasing with the Sierra Hicolor DAC	107	*(Nov 91)*
Catching up	120	*(Dec 91)*
3-D animation	131	*(Jan 92)*
More 3-D animation	143	*(Feb 92)*
Fast 3-D animation: meet X-Sharp	153	*(Mar 92)*
Raw speed and more	166	*(Apr 92)*
Fast antialiasing	177	*(Jun 92)*
3-D shading	192	*(Jul 92)*
Color-modeling in 256-color mode	201	*(Aug 92)*
Pooh and the space station	210	*(Sep 92)*
How to shear a sheep, and other texture-mapping niceties	219	*(Oct 92)*
The good, the bad, and the run-sliced	228	*(Nov 92)*
Moving, faster lines, and page flipping	237	*(Dec 92)*
Yet another animation method	242	*(Jan 93)*
More dirty (dirtier?) rectangles	246	*(Feb 93)*

Journal: Dr. Dobb's Journal Feb 1991 v16 n2 p153(7)

The polygon primeval

Author: Abrash, Michael.
AttFile: GP-FEB91.ASC Source code listing.

Summary: The first in a series of columns develops routines to draw filled polygons. Filled polygons, which are essentially any closed form filled with pixels in a consistent color or pattern, are useful for a variety of graphics constructs. Polygons are divisible into convex, nonconvex and complex shapes. Filling a polygon is fundamentally a rasterization process involving drawing all of the horizontal lines within the polygon's boundaries. Problems that must be resolved include defining which pixels are within each polygon and ensuring that multiple polygons can fit together without overlapping. A line-tracing algorithm is modified so that it contains only pixels that are truly within the polygon. Rules and routines are developed to fit polygons together without overlapping and fill them.

"Give me but one firm spot on which to stand, and I will move the Earth. "

–Archimedes

Were Archimedes alive today, he might say, "Give me but one fast polygon-fill routine on which to call, and I will draw the Earth." Programmers often think of pixel drawing as being the basic graphics primitive, but filled polygons are equally fundamental and far more useful. Filled polygons can be used for constructs as diverse as a single pixel or a 3-D surface, and virtually everything in between.

I'll spend some time in this column developing routines to draw filled polygons and building more sophisticated graphics operations atop those routines, Sooner, rather than later, I'll get to 2-D manipulation and animation of polygonbased entities (with occasional diversions to Other graphics topics of interest), leading uP to an exploration of 3-D graphics. You can't get there from here without laying some groundwork, though, so this month I'll begin with the basics of filling a polygon. Next month, we'll see how to draw a polygon considerably faster. That will set the tone for this column: High-level exploration of a graphics topic first, followed by a speedy hardware-specific implementation for the IBM PCNGA combination, the most widely used graphics system around. Abstract, machine-independent graphics is a thing of beauty, but only by understanding graphics at all levels, including the hardware, can you boost performance into the realm of the sublime.

And slow computer graphics is scarcely worth the bother.

Filled Polygons

A polygon is simply a shape formed by lines laid end to end to form a continuous, closed path. A polygon is filled by setting all pixels within the polygon's boundaries to a color or pattern. For now, we'll work only with polygons filled with solid colors.

You can divide polygons into three categories: convex, nonconvex, and complex, as shown in Convex polygons include what you'd normally thingk of as " convex" and mole; as far as we're concerned, a convex polygon is on, for which any horizontal line drawn through the polygon encounters the right edge exactly once and the left edge exactly once, excluding horizontal and zero-length edge segments. Put another way, neither the right nor left edge of a convex polygon ever reverses direction from uP to down, or vice-versa. Also, the right and left edges of a convex polygon may not cross each other, although they may touch so long as the right edge never crosses over to the left side of the left edge. (Check out the second polygon drawn in Listing Three, page 166, which certainly isn't convex in the normal sense.) The boundaries of nonconvex polygons, on the other hand, can go in whatever

directions they please, so long as they never cross. Complex polygons can have any boundaries you might imagine, which makes for interesting problems in deciding which interior spaces to fill and which not. Each category is a superset of the previous one.

Why bother to distinguish between convex, nonconvex, and complex polygons? For peformance, especially when it comes to filling convex polygons. It's with filled convex polygons that we're going to start; they're widely useful and will serve well to introduce some of the subtler complexities of polygon drawing, not the least of which is the slippery concept of "inside." Which Side is inside? The basic principle of polygon filling is decomposing each polygon into a series of horizontal fines, one for each horizontal row of pixels, or scan line, within the polygon (a process I'll call scan conversion), and drawing the horizontal lines. I'll refer to the entire process as rasterization. Rasterization of convex polygons is easily done by starting at the top of the polygon and tracing down the left and right sides, one scan line (one vertical pixel) at a time, filling the extent between the two edges on each scan line, until the bottom of the polygon is reached. At first glance, rasterization does not seem to be particularly complicated, although it should be apparent that this simple approach is inadequate for nonconvex polygons.

There are a couple of complications, however. The lesser complication is how to rasterize the polygon efficiently, given that it's difficult to write fast code that simultaneously traces two edges and flus the space between them. The solution is decoupling the process of scan-converting the polygon into a list of horizontal lines from that of drawing the horizontal lines. One device-independent routine can trace along the two edges and build a list of the beginning and end coordinates of the polygon on each raster line. Then a second, device-specific routine can draw from the list after the entire polygon has been scanned. We'll see this in action shortly.

The second, greater complication arises because the definition of which pixels are "within" a polygon is a more complicated matter than you might imagine. You might think that scan-converting an edge of a polygon is analogous to drawing a line from one vertex to the next, but this is not so. A line by itself is a one-dimensional construct, and as such is approximated on a display by drawing the pixels nearest to the line on either side of the true line. A line serving as a polygon boundary, on the other hand, is part of a two-dimensional object. When filling a polygon, we want to draw the pixels within the polygon, but a standard vertex-to-vertex linedrawing algorithm will draw many pixels outside the polygon, as shown in Figure 2.

It's no crime to use standard lines to trace out a polygon, rather than drawing only interior pixels. In fact, there are certain advantages: For example, the edges of a filled polygon will match the edges of the same polygon drawn unfilled. Such polygons will look pretty much as they're supposed to, and all drawing on raster displays is, after all, only an approximation of an ideal.

There's one great drawback to tracing polygons with standard lines, however: Adjacent polygons won't fit together properly, as shown in Figure 3. If you use six equilateral triangles to make a hexagon, for example, the edges of the triangles will overlap when traced with standard lines, and more recently drawn triangles will wipe out portions of their predecessors. Worse still, odd color effects will show up along the polygon boundaries if exclusive-or drawing is used. Consequently, filling out to the boundary lines just won't do for drawing images composed of fitted together polygons. And because fitting polygons together is exactly what I have in mind, we need a different approach. How Do You Fit Polygons Together? How, then, do you fit polygons together? Very carefully. First, the line-tracing algorithm must be adjusted so that it selects only those pixels that are truly inside the polygon. This basically requires shifting a standard line-drawing algorithm horizontally by one half-pixel toward the polygon's interior. That leaves the issue of how to handle points that are exactly on the boundary, and points that lie at vertices, so that those points are drawn once and only once. To deal with that, we're going to adopt the following rules:

Points located exactly on nonhorizontal edges are drawn only if the interior of the polygon is directly to the right (left edges are drawn, right edges aren't). 9 Points located exactly on horizontal edges are drawn only if the interior of the polygon is directly below them (horizontal top edges are drawn, horizontal bottom edges aren't).

A vertex is drawn only if afl lines ending at that point meet the above conditions (no right or bottom edges end at that point).

All edges of a polygon except those that are flat tops or flat bottoms will be considered either right edges or left edges, regardless of slope. The left edge is the one that starts with the leftmost line down from the top of the polygon.

These rules ensure that no pixel is drawn more than once when adjacent polygons are fined, and that if polygons cover the full 360-degree range around a pixel, then that pixel will be drawn once and only once - just what we need in order to be able to fit filled polygons together seamlessly.

This sort of non-overlapping polygon filling isn't ideal for all purposes. Polygons are skewed toward the top and left edges, which not only introduces drawing error relative to the ideal polygon but also means that a filled polygon won't match the same polygon drawn unfilled. Narrow wedges and one-pixel-wide polygons will show up spottily. All in all, the choice of polygon-filling approach depends entirely on the ways in which the filled polygons will be used.

For our purposes, nonoverlapping polygons are the way to go, so let's have at them.

Non-overlapping Convex Polygons Made Easy

Without further ado, Listing One (page 164) contains a function, FillConvexPolygon, that accepts a list of points that describe a convex polygon, with the last point assumed to connect to the first, and scans it into a list of lines to flU, then passes that list to the function DrawHorizontalLineList Listing Two (page 165). Listing Three (page 166) is a sample program that calls FillConvexPolygon to draw polygons of various sorts, and Listing Four (page 166) is a header file included by the other listings.

Listing Two isn't particularly interesting; it merely draws each horizontal line in the passed-in list in the simplest possible way, one pixel at a time. NO, that doesn't make the pixel the fundamental primitive; next month I'll replace Listing Two with a much faster version that doesn't bother with individual pixels at all.)

Listing One is where the action is this month. Our goal is to scan out the left and right edges of each polygon so that all points inside and no points outside the polygon are drawn, and so that all points located exactly on the boundary are drawn only if they are not on right or bottom edges. That's precisely what Listing One does; here's how.

Listing one first finds the top and bottom of the polygon, then works out from the top point to find the two ends of the top edge. If the ends are at different locations, the top is flat, which has two implications. Firstly, it's easy to find the starting vertices and directions through the vertex list for the left and right edges. (To scan-convert them properly, we must first determine which edge is which.) Secondly, the top scan line of the polygon should be drawn without the rightmost pixel, because only the rightmost pixel of the horizontal edge that makes up the top scan line is part of a right edge.

if, on the other hand, the ends of the top edge are at the same location, then the top is pointed. In that case, the top scan line of the polygon isn't drawn; it's part of the right-edge line that starts at the top vertex. it's part of a left-edge line, too, but the right edge overrides.) When the top isn't flat, it's more difficult to tell in which direction through the vertex list the right and left edges go, because both edges start at the top vertex. The solution is to compare the slopes from the top vertex to ends of the two lines coming out of it in order to see which is leftmost. The calculations in Listing One involving the various deltas do this, using a slightly rearranged form of the equation: DeltaYN/DeltaXN > DeltaYP/DeltaXP

Once we know where the left edge starts in the vertex list, we can scan-convert it a line at a time until the bottom vertex is reached. Each point is stored as the starting X coordinate for the corresponding scan line in the list we'll pass to DrawHorizontalLineList. The nearest X coordinate

on each scan line that's on or to the right of the left edge is selected. The last point of each line making up the left edge isn't scan-converted, producing two desirable effects. First, it avoids drawing each vertex twice; two lines come into every vertex, but we want to scan-convert each vertex only once. Second, not scan-converting the last point of each line causes the bottom scan line of the polygon not to be drawn, as required by our rules. The first scan line of the polygon is also skipped if the top isn't flat.

Now we need to scan-convert the right edge into the ending X coordinate fields of the line list. This is performed in the same manner as for the left edge, except that every line in the right edge is moved one pixel to the left before being scan-converted. Why? We want the nearest point to the left of but not on the right edge, so that the right edge itself isn't drawn. As it happens, drawing the nearest point on or to the right of a line moved one pixel to the left is exactly the same as drawing the nearest point to the left of but not on that line in its original location. Sketch it out and you'll see what I mean.

Once the two edges are scan-converted, the whole line list is passed to DrawHorizontalLineList, and the polygon is drawn.

Finis. Oddball Cases Listing One handles zero-length segments (multiple vertices at the same location) by ignoring them, which will be useful down the road because scaled down polygons can end up with nearby vertices moved to the same location. Horizontal line segments are fine anywhere in a polygon, too. Basically, Listing One scan-converts between active edges (the edges that define the extent of the polygon on each scan line) and both horizontal and zero-length lines are non-active; neither advances to another scan line, so they don't affect the edges being scanned.

Book of the Month

The book of the month is the second edition of Foley and van Dam's classic Fundamentals of Interactive Computer Graphics, the inspiration and primary reference for much of the nonmachinespecific material I'll present in this column. The almost entirely rewritten new version, retitled Computer Graphics: Principles and Practice (AddisonWesley, 1990, $64.50), nearly doubles the size of the first tome, to a total of 1174 pages. You'll wish it were longer, too, because computer graphics has become such a broad field that even this massive book often merely touches on an area, providing the fundamental concepts, equations, and algorithms, and moves on. Still, just about everything you could want to know is in there somewhere. Truly a book to lose yourself in, and highly recommended.

Coming Up Next

This month's code merely demonstrates the principles of filling convex polygons, and is by no means fast. Next month, we'll spice things up by eliminating the floating point calculations and pixel-at-a-time drawing and tossing a little assembly language into the mix.

- - - -

```
#include <stdio.h>
#include <math.h>
#ifdef __TURBOC__
#include <alloc.h>
#else    /* MSC */
#include <malloc.h>
#endif
#include "polygon.h"

/* Advances the index by one vertex forward through the vertex list,
   wrapping at the end of the list */
#define INDEX_FORWARD(Index) \
   Index = (Index + 1) % VertexList->Length;

/* Advances the index by one vertex backward through the vertex list,
   wrapping at the start of the list */
#define INDEX_BACKWARD(Index) \
   Index = (Index - 1 + VertexList->Length) % VertexList->Length;

/* Advances the index by one vertex either forward or backward through
   the vertex list, wrapping at either end of the list */
#define INDEX_MOVE(Index,Direction)                                    \
   if (Direction > 0)                                                  \
      Index = (Index + 1) % VertexList->Length;                        \
   else                                                                \
      Index = (Index - 1 + VertexList->Length) % VertexList->Length;

extern void DrawHorizontalLineList(struct HLineList *, int);
static void ScanEdge(int, int, int, int, int, int, struct HLine **);

int FillConvexPolygon(struct PointListHeader * VertexList, int Color,
      int XOffset, int YOffset)
{
   int i, MinIndexL, MaxIndex, MinIndexR, SkipFirst, Temp;
   int MinPoint_Y, MaxPoint_Y, TopIsFlat, LeftEdgeDir;
   int NextIndex, CurrentIndex, PreviousIndex;
   int DeltaXN, DeltaYN, DeltaXP, DeltaYP;
   struct HLineList WorkingHLineList;
   struct HLine *EdgePointPtr;
   struct Point *VertexPtr;

   /* Point to the vertex list */
   VertexPtr = VertexList->PointPtr;

   /* Scan the list to find the top and bottom of the polygon */
   if (VertexList->Length == 0)
      return(1);  /* reject null polygons */
   MaxPoint_Y = MinPoint_Y = VertexPtr[MinIndexL = MaxIndex = 0].Y;
   for (i = 1; i < VertexList->Length; i++) {
      if (VertexPtr[i].Y < MinPoint_Y)
         MinPoint_Y = VertexPtr[MinIndexL = i].Y; /* new top */
      else if (VertexPtr[i].Y > MaxPoint_Y)
         MaxPoint_Y = VertexPtr[MaxIndex = i].Y; /* new bottom */
   }
   if (MinPoint_Y == MaxPoint_Y)
      return(1);  /* polygon is 0-height; avoid infinite loop below */

   /* Scan in ascending order to find the last top-edge point */
   MinIndexR = MinIndexL;
   while (VertexPtr[MinIndexR].Y == MinPoint_Y)
      INDEX_FORWARD(MinIndexR);
   INDEX_BACKWARD(MinIndexR); /* back up to last top-edge point */

   /* Now scan in descending order to find the first top-edge point */
   while (VertexPtr[MinIndexL].Y == MinPoint_Y)
      INDEX_BACKWARD(MinIndexL);
   INDEX_FORWARD(MinIndexL); /* back up to first top-edge point */

   /* Figure out which direction through the vertex list from the top
      vertex is the left edge and which is the right */
   LeftEdgeDir = -1; /* assume left edge runs down thru vertex list */
   if ((TopIsFlat = (VertexPtr[MinIndexL].X !=
         VertexPtr[MinIndexR].X) ? 1 : 0) == 1) {
      /* If the top is flat, just see which of the ends is leftmost */
```

```
   if (VertexPtr[MinIndexL].X > VertexPtr[MinIndexR].X) {
      LeftEdgeDir = 1;  /* left edge runs up through vertex list */
      Temp = MinIndexL;       /* swap the indices so MinIndexL   */
      MinIndexL = MinIndexR;  /* points to the start of the left */
      MinIndexR = Temp;       /* edge, similarly for MinIndexR   */
   }
} else {
   /* Point to the downward end of the first line of each of the
      two edges down from the top */
   NextIndex = MinIndexR;
   INDEX_FORWARD(NextIndex);
   PreviousIndex = MinIndexL;
   INDEX_BACKWARD(PreviousIndex);
   /* Calculate X and Y lengths from the top vertex to the end of
      the first line down each edge; use those to compare slopes
      and see which line is leftmost */
   DeltaXN = VertexPtr[NextIndex].X - VertexPtr[MinIndexL].X;
   DeltaYN = VertexPtr[NextIndex].Y - VertexPtr[MinIndexL].Y;
   DeltaXP = VertexPtr[PreviousIndex].X - VertexPtr[MinIndexL].X;
   DeltaYP = VertexPtr[PreviousIndex].Y - VertexPtr[MinIndexL].Y;
   if (((long)DeltaXN * DeltaYP - (long)DeltaYN * DeltaXP) < 0L) {
      LeftEdgeDir = 1;  /* left edge runs up through vertex list */
      Temp = MinIndexL;       /* swap the indices so MinIndexL   */
      MinIndexL = MinIndexR;  /* points to the start of the left */
      MinIndexR = Temp;       /* edge, similarly for MinIndexR   */
   }
}

/* Set the # of scan lines in the polygon, skipping the bottom edge
   and also skipping the top vertex if the top isn't flat because
   in that case the top vertex has a right edge component, and set
   the top scan line to draw, which is likewise the second line of
   the polygon unless the top is flat */
if ((WorkingHLineList.Length =
      MaxPoint_Y - MinPoint_Y - 1 + TopIsFlat) <= 0)
   return(1);  /* there's nothing to draw, so we're done */
WorkingHLineList.YStart = YOffset + MinPoint_Y + 1 - TopIsFlat;

/* Get memory in which to store the line list we generate */
if ((WorkingHLineList.HLinePtr =
      (struct HLine *) (malloc(sizeof(struct HLine) *
      WorkingHLineList.Length))) == NULL)
   return(0);  /* couldn't get memory for the line list */

/* Scan the left edge and store the boundary points in the list */
/* Initial pointer for storing scan converted left-edge coords */
EdgePointPtr = WorkingHLineList.HLinePtr;
/* Start from the top of the left edge */
PreviousIndex = CurrentIndex = MinIndexL;
/* Skip the first point of the first line unless the top is flat;
   if the top isn't flat, the top vertex is exactly on a right
   edge and isn't drawn */
SkipFirst = TopIsFlat ? 0 : 1;
/* Scan convert each line in the left edge from top to bottom */
do {
   INDEX_MOVE(CurrentIndex,LeftEdgeDir);
   ScanEdge(VertexPtr[PreviousIndex].X + XOffset,
         VertexPtr[PreviousIndex].Y,
         VertexPtr[CurrentIndex].X + XOffset,
         VertexPtr[CurrentIndex].Y, 1, SkipFirst, &EdgePointPtr);
   PreviousIndex = CurrentIndex;
   SkipFirst = 0; /* scan convert the first point from now on */
} while (CurrentIndex != MaxIndex);

/* Scan the right edge and store the boundary points in the list */
EdgePointPtr = WorkingHLineList.HLinePtr;
PreviousIndex = CurrentIndex = MinIndexR;
SkipFirst = TopIsFlat ? 0 : 1;
/* Scan convert the right edge, top to bottom. X coordinates are
   adjusted 1 to the left, effectively causing scan conversion of
   the nearest points to the left of but not exactly on the edge */
do {
   INDEX_MOVE(CurrentIndex,-LeftEdgeDir);
   ScanEdge(VertexPtr[PreviousIndex].X + XOffset - 1,
```

```
            VertexPtr[PreviousIndex].Y,
            VertexPtr[CurrentIndex].X + XOffset - 1,
            VertexPtr[CurrentIndex].Y, 0, SkipFirst, &EdgePointPtr);
      PreviousIndex = CurrentIndex;
      SkipFirst = 0; /* scan convert the first point from now on */
   } while (CurrentIndex != MaxIndex);

   /* Draw the line list representing the scan converted polygon */
   DrawHorizontalLineList(&WorkingHLineList, Color);

   /* Release the line list's memory and we're successfully done */
   free(WorkingHLineList.HLinePtr);
   return(1);
}

/* Scan converts an edge from (X1,Y1) to (X2,Y2), not including the
   point at (X2,Y2). This avoids overlapping the end of one line with
   the start of the next, and causes the bottom scan line of the
   polygon not to be drawn. If SkipFirst != 0, the point at (X1,Y1)
   isn't drawn. For each scan line, the pixel closest to the scanned
   line without being to the left of the scanned line is chosen */
static void ScanEdge(int X1, int Y1, int X2, int Y2, int SetXStart,
      int SkipFirst, struct HLine **EdgePointPtr)
{
   int Y, DeltaX, DeltaY;
   double InverseSlope;
   struct HLine *WorkingEdgePointPtr;

   /* Calculate X and Y lengths of the line and the inverse slope */
   DeltaX = X2 - X1;
   if ((DeltaY = Y2 - Y1) <= 0)
      return;     /* guard against 0-length and horizontal edges */
   InverseSlope = (double)DeltaX / (double)DeltaY;

   /* Store the X coordinate of the pixel closest to but not to the
      left of the line for each Y coordinate between Y1 and Y2, not
      including Y2 and also not including Y1 if SkipFirst != 0 */
   WorkingEdgePointPtr = *EdgePointPtr; /* avoid double dereference */
   for (Y = Y1 + SkipFirst; Y < Y2; Y++, WorkingEdgePointPtr++) {
      /* Store the X coordinate in the appropriate edge list */
      if (SetXStart == 1)
         WorkingEdgePointPtr->XStart =
               X1 + (int)(ceil((Y-Y1) * InverseSlope));
      else
         WorkingEdgePointPtr->XEnd =
               X1 + (int)(ceil((Y-Y1) * InverseSlope));
   }
   *EdgePointPtr = WorkingEdgePointPtr;   /* advance caller's ptr */
}
```

[LISTING TWO]

```
/* Draws all pixels in the list of horizontal lines passed in, in
   mode 13h, the VGA's 320x200 256-color mode. Uses a slow pixel-by-
   pixel approach, which does have the virtue of being easily ported
   to any environment. */

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

#define SCREEN_WIDTH    320
#define SCREEN_SEGMENT  0xA000

static void DrawPixel(int, int, int);

void DrawHorizontalLineList(struct HLineList * HLineListPtr,
      int Color)
{
   struct HLine *HLinePtr;
   int Y, X;
```

```
   /* Point to the XStart/XEnd descriptor for the first (top)
      horizontal line */
   HLinePtr = HLineListPtr->HLinePtr;
   /* Draw each horizontal line in turn, starting with the top one and
      advancing one line each time */
   for (Y = HLineListPtr->YStart; Y < (HLineListPtr->YStart +
         HLineListPtr->Length); Y++, HLinePtr++) {
      /* Draw each pixel in the current horizontal line in turn,
         starting with the leftmost one */
      for (X = HLinePtr->XStart; X <= HLinePtr->XEnd; X++)
         DrawPixel(X, Y, Color);
   }
}

/* Draws the pixel at (X, Y) in color Color in VGA mode 13h */
static void DrawPixel(int X, int Y, int Color) {
   unsigned char far *ScreenPtr;

#ifdef __TURBOC__
   ScreenPtr = MK_FP(SCREEN_SEGMENT, Y * SCREEN_WIDTH + X);
#else    /* MSC 5.0 */
   FP_SEG(ScreenPtr) = SCREEN_SEGMENT;
   FP_OFF(ScreenPtr) = Y * SCREEN_WIDTH + X;
#endif
   *ScreenPtr = (unsigned char)Color;
}
```

[LISTING THREE]

```
/* Sample program to exercise the polygon-filling routines. This code
   and all polygon-filling code has been tested with Turbo C 2.0 and
   Microsoft C 5.0 */

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

/* Draws the polygon described by the point list PointList in color
   Color with all vertices offset by (X,Y) */
#define DRAW_POLYGON(PointList,Color,X,Y)                    \
   Polygon.Length = sizeof(PointList)/sizeof(struct Point); \
   Polygon.PointPtr = PointList;                             \
   FillConvexPolygon(&Polygon, Color, X, Y);

void main(void);
extern int FillConvexPolygon(struct PointListHeader *, int, int, int);

void main() {
   int i, j;
   struct PointListHeader Polygon;
   static struct Point ScreenRectangle[] =
         {{0,0},{320,0},{320,200},{0,200}};
   static struct Point ConvexShape[] =
         {{0,0},{121,0},{320,0},{200,51},{301,51},{250,51},{319,143},
         {320,200},{22,200},{0,200},{50,180},{20,160},{50,140},
         {20,120},{50,100},{20,80},{50,60},{20,40},{50,20}};
   static struct Point Hexagon[] =
         {{90,-50},{0,-90},{-90,-50},{-90,50},{0,90},{90,50}};
   static struct Point Triangle1[] = {{30,0},{15,20},{0,0}};
   static struct Point Triangle2[] = {{30,20},{15,0},{0,20}};
   static struct Point Triangle3[] = {{0,20},{20,10},{0,0}};
   static struct Point Triangle4[] = {{20,20},{20,0},{0,10}};
   union REGS regset;

   /* Set the display to VGA mode 13h, 320x200 256-color mode */
   regset.x.ax = 0x0013;   /* AH = 0 selects mode set function,
                              AL = 0x13 selects mode 0x13
                              when set as parameters for INT 0x10 */
   int86(0x10, &regset, &regset);
```

```
   /* Clear the screen to cyan */
   DRAW_POLYGON(ScreenRectangle, 3, 0, 0);

   /* Draw an irregular shape that meets our definition of convex but
      is not convex by any normal description */
   DRAW_POLYGON(ConvexShape, 6, 0, 0);
   getch();    /* wait for a keypress */

   /* Draw adjacent triangles across the top half of the screen */
   for (j=0; j<=80; j+=20) {
      for (i=0; i<290; i += 30) {
         DRAW_POLYGON(Triangle1, 2, i, j);
         DRAW_POLYGON(Triangle2, 4, i+15, j);
      }
   }

   /* Draw adjacent triangles across the bottom half of the screen */
   for (j=100; j<=170; j+=20) {
      /* Do a row of pointing-right triangles */
      for (i=0; i<290; i += 20) {
         DRAW_POLYGON(Triangle3, 40, i, j);
      }
      /* Do a row of pointing-left triangles halfway between one row
         of pointing-right triangles and the next, to fit between */
      for (i=0; i<290; i += 20) {
         DRAW_POLYGON(Triangle4, 1, i, j+10);
      }
   }
   getch();    /* wait for a keypress */

   /* Finally, draw a series of concentric hexagons of approximately
      the same proportions in the center of the screen */
   for (i=0; i<16; i++) {
      DRAW_POLYGON(Hexagon, i, 160, 100);
      for (j=0; j<sizeof(Hexagon)/sizeof(struct Point); j++) {
         /* Advance each vertex toward the center */
         if (Hexagon[j].X != 0) {
            Hexagon[j].X -= Hexagon[j].X >= 0 ? 3 : -3;
            Hexagon[j].Y -= Hexagon[j].Y >= 0 ? 2 : -2;
         } else {
            Hexagon[j].Y -= Hexagon[j].Y >= 0 ? 3 : -3;
         }
      }
   }
   getch();    /* wait for a keypress */

   /* Return to text mode and exit */
   regset.x.ax = 0x0003;   /* AL = 3 selects 80x25 text mode */
   int86(0x10, &regset, &regset);
}
```

[LISTING FOUR]

```
/* POLYGON.H: Header file for polygon-filling code */

/* Describes a single point (used for a single vertex) */
struct Point {
   int X;   /* X coordinate */
   int Y;   /* Y coordinate */
};

/* 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 the 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 (used to describe a scan-converted polygon to the
   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 */
};
```

Journal: Dr. Dobb's Journal March 1991 v16 n3 p16(8)

80x86 optimization: aim down the middle and pray

Author: Abrash, Michael.
AttFile: 80X86.ASC Source code listing.

Summary: Optimizing code for 8088, 80286, 80386 and 80486 microprocessors is difficult because the chips use significantly different memory architectures and instruction execution times. Code cannot be optimized for the 80x86 family; rather, code must be designed to produce good performance on a range of systems or optimized for particular combinations of processors and memory. Programmers must avoid the unusual instructions supported by the 8088, which have lost their performance edge in subsequent chips. String instructions should be used but not relied upon. Registers should be used rather than memory operations. Branching is also slow for all four processors. Memory accesses should be aligned to improve performance. Generally, optimizing an 80486 requires exactly the opposite steps as optimizing an 8088.

Optimization

Picture this: You're an archer aiming at a target 100 feet away. A strong wind comes up and pushes each arrow to the left as it flies. Naturally, you compensate by aiming farther to the right. That's what it's like optimizing for the 8088; once you learn to compensate for the strong but steady effects of the prefetch queue and the 8-bit bus, you can continue merrily on your programming way.

Now the wind starts gusting unpredictably. There's no way to compensate, so you just aim for the bull's-eye and hope for the best. That's what it's like writing code for good performance across the entire 80x86 family, or even for the 286/386SX/386 heart of today's market. You just aim down the middle and pray.

The New World of the 80x86

In the beginning, the 8088 was king, and that was good. The optimization rules weren't obvious, but once you learned them, you could count on them serving you well on every computer out there.

Not so these days. There are four major processor types--8088, 80286, 80386, and 80486--with a bewildering array of memory architectures: cached (in several forms), page mode, static-column RAM, interleaved, and, of course, the 386SX, with its half-pint memory interface. The processors offer wildly differing instruction execution times, and memory architectures warp those times further by affecting the speed of instruction fetching and access to memory operands. Because actual performance is a complex interaction of instruction characteristics, instruction execution times, and memory access speed, the myriad processor-memory combinations out there make "exact performance" a meaningless term. A specific instruction sequence may run at a certain speed on a certain processor in a certain system, but that often says little about the performance of the same instructions on a different processor, or even on the same processor with a different memory system. The result: Precise optimization for the general PC market is a thing of the past. (We're talking about optimizing for speed here; optimizing for size is the same for all processors so long as you stick to 8088-compatible code.)

So there is no way to optimize performance ideally across the 80x86 family. An optimization that suits one processor beautifully is often a dog on another. Any 8088 programmer would instinctively replace:

```
        DEC  CX
        JNZ  LOOPTOP
```

with:

```
        LOOP  LOOPTOP
```

because LOOP is significantly faster on the 8088. LOOP is also faster on the 286. On the 386, however, LOOP is actually two cycles slower than DEC/JNZ. The pendulum swings still further on the 486, where LOOP is about twice as slow as DEC/JNZ--and, mind you, we're talking about what was originally perhaps the most obvious optimization in the entire 80x86 instruction set.

In short, there is no such thing as code that's truly optimized for the 80x86. Instead, code is either optimized for specific processor-memory combinations, or aimed down the middle, designed to produce good performance across a range of systems. Optimizing for the 80x86 family by aiming down the middle is quite different from optimizing for the 8088, but many PC programmers are inappropriately still applying the optimization lore they've learned over the years on the PC (or AT). The world has changed, and many of those old assumptions and tricks don't hold true anymore.

You will not love the new world of 80x86 optimization, which is less precise and offers fewer clever tricks than optimizing for the 8088 alone. Still, isn't it better to understand the forces affecting your code's performance out in the real world than to optimize for a single processor and hope for the best?

Better, yes. As much fun, no. Optimizing for the 8088 was just about as good as it gets. So it goes.

Optimization Rules for a New World

So, how do you go about writing fast code nowadays? One way is to write different versions of critical code for various processors and memory access speeds, selecting the best version at runtime. That's a great solution, but it requires an awful lot of knowledge and work.

An alternative is to optimize for one particular processor and settle for whatever performance you get on the others. This might make sense when the 8088 is the target processor because it certainly needs the optimization more than any other processor. However, 8088 optimization works poorly at the upper end of the 80x86 family.

Nowadays, though, most of us want to optimize for the 286 and 386 systems that dominate the market, or across all 80x86 processors, and that's a tough nut to crack. The 286 and 386 come in many configurations, and you can be sure, for example, that a 386SX, an interleaved 386, and a cached 386 have markedly different performance characteristics. There are, alas, no hard and fast optimization rules that apply across all these environments.

My own approach to 80x86 optimization has been to develop a set of general rules that serve reasonably well throughout the 80x86 line, especially the 286 and 386, and to select a specific processor (in my case a cached 386, for which cycle times tend to be accurate) to serve as the tiebreaker when optimization details vary from one processor to another. (Naturally, it's only worth bothering with these optimizations in critical code.) The rules I've developed are:

* Avoid accessing memory operands; use the registers to the max.

* Don't branch.

* Use string instructions, but don't go much out of your way to do so.

* Keep memory accesses to a minimum by avoiding memory operands and keeping instructions short.

* Align memory accesses.

* Forget about many of those clever 8088 optimizations, using oddball instructions such as DAA and XLAT, that you spent years learning.

Next I'll discuss each of these rules in turn in the context of 8088-compatible real mode, which is still the focus of the 80x86 world.

Later, I'll touch on protected mode.
Let's start by looking at the last--and most surprising--rule.

Kiss Those Tricks Goodbye

To skilled assembly language programmers, the 8088 is perhaps the most wonderful processor ever created, largely because the instruction set is packed with odd instructions that are worthless to compilers but can work miracles in the hands of clever assembly programmers. Unfortunately, each new generation of the 80x86 has rendered those odd instructions and marvelous tricks less desirable. As the execution time for the commonly used instruction ADD BX, 4 has gone down from four cycles (8088) to three cycles (286) to two cycles (386) to one cycle (486), the time for the less frequently used instruction CBW has gone from two cycles (8088 and 286) up to three cycles (386 and 486)!

Consider this ancient optimization for converting a binary digit to hex ASCII:

```
        ADD AL,90H
        DAA
        ADC AL,40H
        DAA
```

Now consider the standard alternative:

```
        ADD  AL,'0'
        CMP  AL,'9'
        JBE  HaveAscii
        ADD  AL,'A'-('9'+1)
HaveAscii:
```

As Figure 1 indicates, the standard code should be slower on an 8088 or 286, but faster on a 386 or a 486--and real-world tests confirm those results, as shown in Figure 2. (All "actual performance" timings in this article were performed with the Zen timer from Zen of Assembly Language, see "References" for details. The systems used for the tests were: 8088, standard 4.77 MHz PC XT; 80286, standard one-wait-state, 8 MHz PC AT; 386SX, 16 MHz noncached; 80386, 20 MHz externally cached with all instructions and data in external cache for all tests except Listings One and Two; 80486, 25 MHz internally cached, with all instructions and data in internal cache for all tests except Listings One and Two.)

In other words, this nifty, time-tested optimization is an anti-optimization on the 386 and 486.

Why is this? On the 386, DAA--a rarely used instruction--takes four cycles, and on the 486 it takes two cycles, in both cases twice as long as the more common instructions CMP and ADD; in contrast, on the 8088 all three instructions are equally fast at four cycles. Also, the instruction-fetching advantage that the 1-byte DAA provides on the 8088 means nothing on a cached 386.

Nor is this an isolated example. Most oddball instructions, from AAA to XCHG, have failed to keep pace with the core instructions--ADC, ADD, AND, CALL, CMP, DEC, INC, Jcc, JMP, LEA, MOV, OR, POP, PUSH, RET, SBB, SUB, TEST, and XOR--during the evolution from 8088 to 486. As we saw earlier, even LOOP lags behind on the 386 and 486. Check your favorite tricks for yourself; they might or might not hold up on the 386, but will most likely be liabilities on the 486. Sorry, but I just report the news, and the news is: Kiss most of those tricks goodbye as the 386 and 486 come to dominate the market. (This means that hand-optimization in assembly language yields less of a performance boost nowadays than it did when the 8088 was king; the improvement is certainly significant, but rarely in the 200-500 percent range anymore. Sic transit gloria mundi.) Most startling of all, string instructions lose much of their allure as we move away from the 8088, hitting bottom on the 486.

The 486: All the Rules Change

The 486 represents a fundamental break with 8088-style optimization. Virtually all the old rules fail on the 486, where, incredibly, a move to or from memory often takes just one cycle, but exchanging two registers takes three cycles. The nonbranching core instructions mentioned earlier take only one cycle on the 486 when operating on registers; MOV can, under most conditions, access memory in one cycle; and CALL and JMP take only three cycles, given a cache hit. However, noncore instructions take considerably longer. XLAT takes four cycles; even STC and CLC take two cycles each. The 486's highly asymmetric execution times heavily favor core instructions and defeat most pre-486 optimizations.

Core instructions do have a weakness on the 486. While 486 MOVs involving memory are remarkably fast, accessing memory for an operand to OR, ADD, or the like costs cycles. Even with the 8K internal cache, memory is not as fast as registers, except when MOV is used (and sometimes not even then), so registers are still preferred operands. (AND [BX],1 is fast, at only three cycles, but AND BX,1 takes only one cycle--three times as fast.)

OUT should be avoided whenever possible on the 486, and likewise for IN. OUT takes anywhere from 10 to 31 cycles, depending on processor mode and privileges, more than an order of magnitude slower than MOV. The lousy performance of OUT -- true on the 386 as well -- has important implications for graphics applications.

String instructions are so slow on the 486 that you should check cycle times before using any string instruction other than the always superior REP MOV's. For example, LODSB takes five cycles on the 486, but MOV AL,[SI]/INC SI takes only two cycles; likewise for STOSB and MOV [DI],AL/INC DI. Listing One (page 73) uses LODSB/STOSB to copy a string, converting lowercase to uppercase while copying; Listing Two (page 73) uses MOV/INC instead. Figure 3 summarizes the performance of the two routines on a variety of processors; note the diminishing effectiveness of string instructions on the newer processors. Think long and hard before using string instructions other than REP MOVS on the 486.

Optimization for the 486 is really a whole new ball game. When optimizing across the 80x86 family, the 486 will generally be the least of your worries because it is so much faster than the rest of the family; anything that runs adequately on any other processor will look terrific on the 486. Still, the future surely holds millions of 486s, so it wouldn't hurt to keep one eye on the 486 as you optimize.

String Instructions: Fading Stars

On the 8088, string instructions are so far superior to other instructions that it's worth going to great lengths to use them, but they lose much of that status on newer processors. One of the best things about string instructions on the 8088 is that they require little instruction fetching, because they're 1-byte instructions and because of the REP prefix; however, instruction fetching is less of a bottleneck on newer processors. String instructions also have superior cycle times on the 8088, but that advantage fades on the 286 and 386 as well.

On the 286, string instructions (when they do exactly what you need) are still clearly better than the alternatives. On the 386, however, some string instructions are, even under ideal circumstances, the best choice only by a whisker, if at all. For example, since Day One, clearing a buffer has been done with REP STOS. That's certainly faster than the looping MOV/ADD approach shown in Listing Three (page 73), but on the 386 and 486 it's no faster than the unrolled loop MOV/ADD approach of Listing Four (page 73), as shown in Figure 4. (Actually, in my tests REP STOS was a fraction of a cycle slower on the 386, and fractionally faster on the 486.) REP STOS is much easier to code and more compact, so it's still the approach of choice for buffer clearing--but it's not necessarily fastest on a 486 or fast-memory 386. This again demonstrates just how unreliable the old optimization rules are on the newer processors.

The point is not that you shouldn't use string instructions on the 386. REP MOVs is the best way to move data, and the other string instructions are compact and usually faster, especially on uncached systems. However, on the

386 it's no longer worth going to the trouble of juggling registers and reorganizing data structures to use string instructions. Furthermore, when you truly need maximum performance on the 386, check out nonstring instructions in unrolled loops. It goes against every lesson learned in a decade of 8088 programming, but avoiding string instructions sometimes pays on the 386.

The Siren Song of Memory Accesses

Finally, here's a rule that's constant from the 8088 to the 486: Use the registers. Avoid memory.

Don't be fooled by the much faster memory access times of the 286 and 386. The effective address calculation time of the 8088 is mostly gone, so MOV AX,[BX] takes only five cycles on the 286, and ADD [SI],DX takes only seven on the 386. That's so much faster than the 17 and 29 cycles, respectively, that they take on the 8088 that you might start thinking that memory is pretty much interchangeable with registers.

Think again. MOV AX,BX is still more than twice as fast as MOV AX,[BX] on the 286, and ADD SI,DX is more than three times as fast as ADD [SI],DX on the 386. Memory operands can also reduce performance by slowing instruction fetching. Memory is fast on the 286 and 386. Registers are faster. Use them as heavily as possible.

Don't Branch

Here's another rule that stays the same across the 80x86 family: Don't branch. Branching suffers on the 8088 from lengthy cycle counts and emptying the prefetch queue. Emptying the prefetch queue is a lesser but nonetheless real problem in the post-8088 world, and the cycle counts of branches are still killers. As Figure 4 indicates, it pays to eliminate branches by unrolling loops or using repeated string instructions.

Modern-Day Instruction Fetching

Instruction fetching is the bugbear of 8088 performance; the 8088 simply can't fetch instruction bytes as quickly as it can execute them, thanks to its undersized bus. Minimizing all memory accesses, including instruction fetches, is paramount on the 8088.

Instruction fetching is less of a problem nowadays. Figure 5 shows the maximum rates at which various processors can fetch instruction bytes; clearly, matters have improved considerably since the 8088, although instructions also execute in fewer cycles on the newer processors. Fetching problems can occur on any 80x86 processor, even the 486, but the only processors other than the 8088 that face major instruction fetching problems are the one-wait-state 286 and the 386SX, although uncached 386s may also outrun memory. However, the problems here are different from and less serious than with the 8088.

Consider: An 8088 executes a register ADD in three cycles, but requires eight cycles to fetch that instruction, a fetch/execute ratio of 2.67. A one-wait-state 286 requires three cycles to fetch a register ADD and executes it in two cycles, a ratio of 1.5. A 386SX can fetch a register ADD in two cycles, matching the execution time nicely, and a cached 386 can fetch two register ADDs in the two cycles it takes to execute just one. For register-only code--the sort of code critical loops should contain--the 386 generally runs flat out, and the 286 and 386SX usually (not always, but usually) outrun memory by only a little at worst. Greater fetching problems can arise when working with large instructions or instruction sequences that access memory nonstop, but those are uncommon in critical code. This is a welcome change from the 8088, where small, register-only instructions tend to suffer most from inadequate instruction fetching.

Also, uncached 386 systems often use memory architectures that provide zero-wait-state performance when memory is accessed sequentially. In register-only code, instruction fetches are the only memory accesses, so fetching proceeds at full speed when the registers are used heavily.

So, is instruction fetching a problem in the post-8088 world? Should instructions be kept short?

Yes. Smaller instructions can help considerably on the one-wait-state 286 and on the 386SX. Not as much as on the 8088, but it's still worth the trouble. Even a cached 386 can suffer from fetching problems, although that's fairly uncommon. For example, when several MOV WORD PTR [MemVar],0 instructions are executed in a row, as might happen when initializing memory variables, performance tends to fall far below rated speed, as shown in Figure 6. The particular problem with MOV WORD PTR [MemVar],0 is that it executes in just two (386) or three (286) cycles, yet has both an addressing displacement field and a constant field. This eats up memory bandwidth by requiring more instruction fetching. It also accesses memory, eating up still more bandwidth. We'll see this again, and worse, when we discuss protected mode.

Generally, though, post-8088 processors with fast memory systems and full-width buses run most instructions at pretty near their official cycle times; for these systems, optimization consists mostly of counting cycles. Slower memory or constricted buses (as in the 386SX) require that memory accesses (both instruction fetches and operand accesses) be minimized as well. Fortunately, the same sort of code--register only--meets both requirements.

Use the registers. Avoid constants. Avoid displacements. Don't branch. That's the big picture. Don't sweat the details.

Alignment: The Easy Optimization

The 286, 386SX, and 386 take twice as long to access memory words at odd addresses as at even addresses. The 386 takes twice as long to access memory dwords at addresses that aren't multiples of four as those that are. You should use ALIGN 2 to word align all word-sized data, and ALIGN 4 to dword align all data that's accessed as a dword operand, as in:

```
         ALIGN  4
 MemVar  dd  ?
         :
         MOV EAX,[MemVar]
```

Alignment also applies to code; you may want to word or dword align the starts of procedures, labels that can only be reached by branching, and the tops of loops. (Code alignment matters only at branch targets, because only the first instruction fetch after a branch can suffer from nonalignment.) Dword alignment of code is optimal, and will help on the 386 even in real mode, but word alignment will produce nearly as much improvement as dword alignment without wasting nearly as many bytes.

Alignment improves performance on many 80x86 systems without hindering it on any. Recommended.

Protected Mode

There are two sorts of protected mode, 16-bit and 32-bit. The primary optimization characteristic of 16-bit protected mode (OS/2 1.X, Rational DOS Extender) is that it takes an ungodly long time to load a segment register (for example, MOV ES,AX takes 17 cycles on a 286) so load segment registers as infrequently as possible in 16-bit protected mode.

Optimizing for 32-bit protected mode (OS/2 2.0, SCO Unix, Phar Lap DOS Extender) is another matter entirely. Typically, no segment loads are needed because of the flat address space. However, 32-bit protected mode code can be bulky, and that can slow instruction fetching. Constants and addressing displacements can be as large as 4 bytes each, and an extra byte, the SIB byte, is required whenever two 32-bit registers are used to address an operand or scaled addressing is used. So, for example, MOV DWORD PTR [MemVar],0 is a 10-byte instruction in 32-bit protected mode. The instruction is supposed to execute in two cycles, but even a 386 needs four to six cycles to fetch it, plus another two cycles to access memory; a few

such instructions in a row can empty the prefetch queue and slow performance considerably. The slowdown occurs more quickly and is more acute on a 386SX, which needs 14 cycles to perform the memory accesses for this nominally 2-cycle instruction.

Code can get even larger when 32-bit instructions are executed in 16-bit segments, adding prefix bytes. (Avoid prefix bytes if you can; they increase instruction size and can cost cycles.) Figure 7 shows actual versus nominal cycle times of multiple MOV DWORD PTR [EBX*4+MemVar],0 instructions running in a 16-bit segment. Although cache type (write-back, write-through) and main-memory write time also affect the performance of stores to memory, there is clearly a significant penalty for using several large (in this case, 13-byte) instructions in a row.

Fortunately, this is a worst case, easily avoided by keeping constants and displacements out of critical loops. For example, you should replace:

```
ADDLOOP: MOV  DWORD PTR BaseTable[EDX+EBX],0
         ADD  EBX,4
         DEC  ECX
         JNZ  ADDLOOP
```

with:

```
         LEA  EBX,BaseTable[EDX+EBX]
         SUB  EAX,EAX
ADDLOOP: MOV  [EBX],EAX
         ADD  EBX,4
         DEC  ECX
         JNZ  ADDLOOP
```

Better yet, use REP STOSD or unroll the loop!

Happily, register-only instructions are no larger in 32-bit protected mode than otherwise and run at or near their rated speed in 32-bit protected mode on all processors. All in all, in protected mode it's more important than ever to avoid large constants and displacements and to use the registers as much as possible.

Conclusion

Optimization across the 80x86 family isn't as precise as 8088 optimization, and it's a lot less fun, with fewer nifty tricks and less spectacular speed-ups. Still, familiarity with the basix 80x86 optimization rules can give you a decided advantage over programmers still laboring under the delusion that the 286, 386, and 486 are merely faster 8088s.

References

Abrash, Michael. Zen of Assembly Language. Glenview, Ill.: Scott, Foresman, 1990.

Barrenechea, Mark. "Peak Performance: On to the 486." Programmer's Journal, (November-December 1990).

Paterson, Tim. "Assembly Language Tricks of the Trade." Dr. Dobb's Journal (March 1990).

Turbo Assembler Quick Reference Guide. Borland International, 1990.

i486 Microprocessor Programmer's Reference Manual. Intel Corporation, 1989.

80386 Programmer's Reference Manual. Intel Corporation, 1986.

Microsystems Components Handbook: Microprocessors Volume I. Intel Corporation, 1985.

- -

80x86 OPTIMIZATION
by Michael Abrash

[LISTING ONE]

```
; Copies one string to another string, converting all characters to
; uppercase in the process, using a loop containing LODSB and STOSB.
; Adapted from Zen of Assembly Language, by Michael Abrash; not a
; standalone program, but designed to be used with the Zen timer from
; that book via the Zen timer's PZTIME.BAT batch file: ZTimerOn starts
; the clock, ZTimerOff stops it, and the test-bed program linked in by
; PZTIME.BAT starts the program, reports the results, and ends.

        jmp     Skip                    ;skip over data in CS and subroutine

SourceString  label  word               ;sample string to copy
        db      'This space intentionally left not blank',0
DestString    db     100 dup (?)        ;destination for copy

; Copies one zero-terminated string to another string,
; converting all characters to uppercase.
; Input: DS:SI = start of source string; DS:DI = start of destination buffer
; Output: none
; Registers altered: AX, BX, SI, DI, ES
; Direction flag cleared

CopyStringUpper:
        mov     ax,ds
        mov     es,ax           ;for STOS
        mov     bl,'a'          ;set up for fast register-register
        mov     bh,'z'          ; comparisons
        cld
StringUpperLoop:
        lodsb                   ;get next character and point to following character
        cmp     al,bl           ;below 'a'?
        jb      IsUpper         ;yes, not lowercase
        cmp     al,bh           ;above 'z'?
        ja      IsUpper         ;yes, not lowercase
        and     al,not 20h      ;is lowercase-make uppercase
IsUpper:
        stosb                   ;put character into new string and point to
                                ; following location
        and     al,al           ;is this the zero that marks end of the string?
        jnz     StringUpperLoop             ;no, do the next character
        ret

; Calls CopyStringUpper to copy & convert SourceString->DestString.
Skip:
        call    ZTimerOn                    ;start timing
        mov     si,offset SourceString      ;point SI to the string to copy from
        mov     di,offset DestString        ;point DI to the string to copy to
        call    CopyStringUpper             ;copy & convert to uppercase
        call    ZTimerOff                   ;stop timing
```

[LISTING TWO]

```
; Copies one string to another string, converting all characters to
; uppercase in the process, using no string instructions.
; Not a standalone program, but designed to be used with the Zen
; timer, as described in Listing 1.

        jmp     Skip                    ;skip over data in CS and subroutine

SourceString  label  word               ;sample string to copy
        db      'This space intentionally left not blank',0
DestString db 100 dup (?)               ;destination for copy
```

```
; Copies one zero-terminated string to another string,
; converting all characters to uppercase.
; Input: DS:SI = start of source string; DS:DI = start of destination string
; Output: none
; Registers altered: AL, BX, SI, DI

CopyStringUpper:
       mov    bl,'a'        ;set up for fast register-register
       mov    bh,'z'        ; comparisons
StringUpperLoop:
       mov    al,[si]       ;get the next character and
       inc    si            ; point to the following character
       cmp    al,bl         ;below 'a'?
       jb     IsUpper       ;yes, not lowercase
       cmp    al,bh         ;above 'z'?
       ja     IsUpper       ;yes, not lowercase
       and    al,not 20h    ;is lowercase-make uppercase

IsUpper:
       mov    [di],al       ;put the character into the new string and
       inc    di            ; point to the following location
       and    al,al         ;is this the zero that marks the end of the string?
       jnz    StringUpperLoop      ;no, do the next character
       ret

; Calls CopyStringUpper to copy & convert SourceString->DestString.
Skip:
       call   ZTimerOn
       mov    si,offset SourceString      ;point SI to the string to copy from
       mov    di,offset DestString        ;point DI to the string to copy to
       call   CopyStringUpper             ;copy & convert to uppercase
       call   ZTimerOff
```

[LISTING THREE]

```
; Clears a buffer using MOV/ADD in a loop.
; Not a standalone program, but designed to be used with the Zen
; timer, as described in Listing 1.

       mov    dx,2          ;repeat the test code twice, to make
                            ; sure it's in the cache (if there is one)
       mov    bx,dx         ;distance from the start of one word
                            ; to the start of the next
       sub    ax,ax         ;set buffer to zeroes
TestTwiceLoop:
       mov    cx,1024       ;clear 1024 words starting at address
       mov    di,8000h      ; DS:8000h (this is just unused memory
                            ; past the end of the program)
       call   ZTimerOn      ;start timing (resets timer to 0)
StoreLoop:
       mov    [di],ax       ;clear the current word
       add    di,bx         ;point to the next word
       dec    cx            ;count off words to clear until none
       jnz    StoreLoop     ; remain
       call   ZTimerOff     ;stop timing
       dec    dx            ;count off passes through test code
       jz     StoreDone     ;that was the second pass; we're done
       jmp    TestTwiceLoop ;that was first pass; do second pass with all
                            ; instructions and data in the cache
StoreDone:
```

[LISTING FOUR]

```
; Clears a buffer using MOV/ADD in an unrolled loop.
; Not a standalone program, but designed to be used with the Zen
; timer, as described in Listing 1.
```

```
        mov     dx,2            ;repeat the test code twice, to make
                                ; sure it's in the cache (if there is one)
        mov     bx,dx           ;distance from the start of one word
                                ; to the start of the next
        sub     ax,ax           ;set buffer to zeroes
TestTwiceLoop:
        mov     si,1024         ;clear 1024 words starting at address
        mov     di,8000h        ; DS:8000h (this is just unused memory
                                ; past the end of the program)
        call    ZTimerOn        ;start timing (resets timer to 0)
        mov     cl,4            ;divide the count of words to clear by
        shr     si,cl           ; 16, because we'll clear 16 words
                                ; each time through the loop
StoreLoop:
        REPT    16              ;clear 16 words in a row without looping
        mov     [di],ax         ;clear the current word
        add     di,bx           ;point to the next word
        ENDM
        dec     si              ;count off blocks of 16 words to clear
        jnz     StoreLoop       ; until none remain
        call    ZTimerOff       ;stop timing
        dec     dx              ;count off passes through test code
        jz      StoreDone       ;that was the second pass; we're done
        jmp     TestTwiceLoop   ;that was the first pass; do the second pass
                                ; with all instructions and data in the cache
StoreDone:
```

Journal: Dr. Dobb's Journal March 1991 v16 n3 p129(6)

Fast convex polygons

Author: Abrash, Michael.
AttFile: GP-MAR91.ASC Source code listing.

Summary: Assembly language programming can improve graphics performance and help programmers understand the function of their programs. Fast convex polygon filling can be performed by optimizing the existing polygon filling implementation. The drawing portion of the program is easily accelerated by using the 'memset' library function, which uses the REP STOS instruction. The code replacement yields a speed an order of magnitude faster than the old implementation. The polygon edge tracing procedures are slow because they use floating-point calculations, but integer calculations are both faster and more accurate. Modification of the drawing and edge tracing portions of the program increase performance almost 20 times. Assembly language can be used in place of C to improve speed even further.

Fast Convex Polygons

Last month, we explored the surprisingly intricate process of filling convex polygons. This month we're going to fill them an order of magnitude or so faster.

Two thoughts may occur to some of you at this point: "Oh, no, he's not going to get into assembly language and device dependent code, is he?" and, "Why bother with polygon filling--or, indeed, any drawing primitives--anyway? Isn't that what GUIs and third-party libraries are for?"

To which I answer, "Well, yes, I am," and, "If you have to ask, you've missed the magic of microcomputer programming." Actually, both questions ask the same thing, and that is: "Why should I, as a programmer, have any idea how my program actually works?"

Put that way, it sounds a little different, doesn't it?

GUIs, reusable code, portable code written entirely in high-level languages, and object-oriented programming are all the rage now, and promise to remain so for the foreseeable future. The thrust of this technology is to enhance the software development process by offloading as much responsibility as possible to other programmers, and by writing all remaining code in modular, generic form. This modular code then becomes a black box to be reused endlessly without another thought about what actually lies inside. GUIs also reduce development time by making many interface choices for you. That, in turn, makes it possible to create quickly and reliably programs that will be easy for new users to pick up, so software becomes easier to both produce and learn. This is, without question, a Good Thing.

The "black box" approach does not, however, necessarily cause the software itself to become faster, smaller, or more innovative; quite the opposite, I suspect. I'll reserve judgement on whether that is a good thing or not, but I'll make a prediction: In the short run, the aforementioned techniques will lead to noticeably larger, slower programs, as programmers understand less and less of what the key parts of their programs do and rely increasingly on general-purpose code written by other people. (In the long run, programs will be bigger and slower yet, but computers will be so fast and will have so much memory that no one will care.) Over time, PC programs will also come to be more similar to one another--and to programs running on other platforms, such as the Mac--as regards both user interface and performance.

Again, I am not saying that this is bad. It does, however, have major implications for the future nature of PC graphics programming, in ways that will directly affect the means by which many of you earn your livings. Not so very long from now, graphics programming--all programming, for that matter--will become mostly a matter of assembling in various ways components written by other people, and will cease to be the all-inclusively creative,

mind-bendingly complex pursuit it is today. (Using legally certified black boxes is, by the way, one direction in which the patent lawyers are leading us; legal considerations may be the final nail in the coffin of homegrown code.) For now, a PC programmer, to understand and even control every single thing that happens on a computer if you so desire, to realize any vision you may have. Take advantage of this unique window of opportunity to create some magic!

Neither does it hurt to understand what's involved in drawing, say, a filled polygon, even if you are using a GUI. You will better understand the performance implications of the available GUI functions, and you will be able to fill in any gaps in the functions provided. You may even find that you can out-perform the GUI on occasion by doing your own drawing into a system memory bitmap, then copying the result to the screen. You will also be able to understand why various quirks exist, and will be able to put them to good use. For example, X Window follows the polygon drawing rules described last month (although it's not obvious from the documentation); if you understood last month's discussion, you're in good shape to use polygons under X.

In short, even though it runs counter to current trends, it helps to understand how things work, especially when they're very visible parts of the software you develop. That said, let's learn more about filling convex polygons.

Fast Convex Polygon Filling

When last we left the topic of filling convex polygons, the implementation we had met all of our functional requirements. In particular, it met stringent rules that guaranteed that polygons would never overlap at shared edges, an important consideration when building polygon-based images. Unfortunately, the implementation was also slow as molasses. This month we'll work up polygon filling code that's fast enough to be truly usable.

Last month's polygon filling code involved three major tasks, each performed by a separate function: Tracing each polygon edge to generate a coordinate list (performed by the function ScanEdge); drawing the scanned-out horizontal lines that constitute the filled polygon (DrawHorizontalLineList); and characterizing the polygon and coordinating the tracing and drawing (FillConvexPolygon). The amount of time that the sample program from last month spent in each of these areas is shown in Table 1. As you can see, half the time was spent drawing and the other half was spent tracing the polygon edges (the time spent in FillConvexPolygon was relatively minuscule), so we have our choice of where to begin optimizing.

Fast Drawing

Let's start with drawing, which is easily sped up. The original code used a double-nested loop that called a draw-pixel function to plot each pixel in the polygon individually. That's a ridiculous approach in a graphics mode that offers linearly mapped memory, as does VGA mode 13h, the mode with which we're working. At the very least, we could point a far pointer to the left edge of each polygon scan line, then draw each pixel in that scan line in quick succession, using something along the lines of *ScrPtr++ = FillColor; inside a loop.

However, it seems silly to use a loop when the 80x86 has an instruction, REP STOS, that's uniquely suited to filling linear memory buffers. There's no way to use REP STOS directly in C code, but it's a good bet that the memset library function uses REP STOS, so you could greatly enhance performance by using memset to draw each scan line of the polygon in a single shot. That, however, is easier said than done. The memset function linked in from the library is tied to the memory model in use; in small (tiny, small, or medium) data models memset accepts only near pointers, so it can't be used to access screen memory. Consequently, a large (compact, large, or huge) data model must be used to allow memset to draw to display memory--a clear case of the tail wagging the dog. This is an excellent example of why, although it is possible to use C to do virtually anything, it's sometimes much simpler just to use a little assembly code and be done with it.

At any rate, Listing One (page 146) shows a version of DrawHorizontalLineList that uses memset to draw each scan line of the polygon in a single call. When linked to last month's test program, Listing One increases pure drawing speed (disregarding edge tracing and other nondrawing time) by more than an order of magnitude over last month's draw-pixel-based code. This despite the fact that Listing One requires a large (in this case, compact) data model. Listing One works fine with Turbo C++, but may not work with other compilers, for it relies on the aforementioned interaction between memset and the selected memory model.

At this point, I'd like to mention that benchmarks are notoriously unreliable; the results in Table 1 are accurate only for the test program, and then only when running on a particular system. Results could be vastly different if smaller, larger, or more complex polygons were drawn, or if a faster or slower computer/VGA combination were used. These factors notwithstanding, the test program does fill a variety of polygons of varying complexity sized from large to small and in between, and certainly the order of magnitude difference between Listing One and the old version of DrawHorizontalLineList is a clear indication of which code is superior.

Anyway, Listing One has the desired effect of vastly improving drawing time. There are cycles yet to be had in the drawing code, but as tracing polygon edges now takes 92 percent of the polygon filling time, it's logical to optimize the tracing code next.

Fast Edge Tracing

There's no secret as to why last month's ScanEdge was so slow: It used floating point calculations. One secret of fast graphics is using integer or fixed-point calculations, instead. (Sure, the floating point code would run faster if a math coprocessor were installed, but it would still be slower than the alternatives; besides, why require a math coprocessor when you don't have to?) Both integer and fixed-point calculations are fast. In many cases, fixed-point is faster, but integer calculations have one tremendous virtue: They're completely accurate. The tiny imprecision inherent in either fixed- or floating-point calculations can result in occasional pixels being one off from their proper location. This is no great tragedy, but after going to so much trouble to ensure that polygons don't overlap at common edges, why not get it exactly right?

In fact, when I tested out the integer edge tracing code by comparing an integer-based test image to one produced by floating point calculations, two pixels out of the whole screen differed, leading me to suspect a bug in the integer code. It turned out, however, that in those two cases, the floating point results were sufficiently imprecise to creep from just under an integer value to just over it, so that the ceil function returned a coordinate that was one too large. Floating point is very accurate--but it is not precise. Integer calculations, properly performed, are.

Listing Two (page 146) shows a C implementation of integer edge tracing. Vertical and diagonal lines, which are trivial to trace, are special-cased. Other lines are broken into two categories: Y-major (closer to vertical) and X-major (closer to horizontal). The handlers for the Y-major and X-major cases operate on the principle of similar triangles: The number of X pixels advanced per scan line is the same as the ratio of the X delta of the edge to the Y delta. Listing Two is more complex than the original floating point implementation, but not painfully so. In return for that complexity, Listing Two is more than 80 times faster--and, as just mentioned, it's actually more accurate than the floating point code.

You gotta love that integer arithmetic.

The Finishing Touch: Assembly Language

The C implementation is now nearly 20 times as fast as the original, good enough for most purposes. Still, it requires that a large data model be used (for memset), and it's certainly not the fastest possible code. The obvious next step is assembly language.

Listing Three (page 146) is an assembly language version of DrawHorizontalLineList. In actual use, it proved to be about 36 percent faster than Listing One; better than a poke in the eye, but just barely. There's more to these timing results than meets that eye, though. Display memory generally responds much more slowly than system memory, especially in 386 and 486 systems. That means that much of the time taken by Listing Three is actually spent waiting for display memory accesses to complete, with the processor forced to idle by wait states. If, instead, Listing Three drew to a local buffer in system memory or to a particularly fast VGA, the assembly implementation might well display a far more substantial advantage over the C code.

And indeed it does. When the test program is modified to draw to a local buffer, both the C and assembly language versions get 0.29 seconds faster, that being a measure of the time taken by display memory wait states. With those wait states factored out, the assembly language version of DrawHorizontalLineList becomes almost three times as fast as the C code.

There is a lesson here. An optimization has no fixed payoff; its value fluctuates according to the context in which it is used. There's relatively little benefit to further optimizing code that already spends half its time waiting for display memory; no matter how good your optimizations, you'll get only a two-times speedup at best, and generally much less than that. There is, on the other hand, potential for tremendous improvement when drawing to system memory, so if that's where most of your drawing will occur, optimizations such as Listing Three are well worth the effort.

Know the environments in which your code will run, and know where the cycles go in those environments.

Maximizing REP STOS

Listing Three doesn't take the easy way out and use REP STOSB to fill each scan line; instead, it uses REP STOSW to fill as many pixel pairs as possible via word-sized accesses, using STOSB only to do odd bytes. Word accesses to odd addresses are always split by the processor into 2-byte accesses. Such word accesses take twice as long as word accesses to even addresses, so Listing Three makes sure that all word accesses occur at even addresses, by performing a leading STOSB first if necessary.

Listing Three is another case in which it's worth knowing the environment in which your code will run. Extra code is required to perform aligned word-at-a-time filling, resulting in extra overhead. For very small or narrow polygons, that overhead might overwhelm the advantage of drawing a word at a time, making plain old REP STOSB faster.

Faster Edge Tracing

Finally, Listing Four (page 147) is an assembly language version of ScanEdge. Listing Four is a relatively straightforward translation from C to assembly, but is nonetheless almost twice as fast as Listing Two.

The version of ScanEdge in Listing Four could certainly be sped up still further by unrolling the loops. FillConvexPolygon, the overall coordination routine, hasn't even been converted to assembly language, so that could be sped up as well. I haven't bothered with these optimizations because all code other than DrawHorizontalLineList takes only 14 percent of the overall polygon filling time when drawing to display memory; the potential return on optimizing nondrawing code simply isn't great enough to justify the effort. Part of the value of a profiler is being able to tell when to stop optimizing; with Listings Three and Four in use, more than two-thirds of the time taken to draw polygons is spent waiting for display memory, so optimization is pretty much maxed out. However, further optimization might be worthwhile when drawing to system memory, where wait states are out of the picture and the nondrawing code takes a significant portion (46 percent) of the overall time.

Again, know where the cycles go.

By the way, note that all the versions of ScanEdge and FillConvexPolygon that we've looked at are adapter independent, and that the C code is also machine independent; all adapterspecific code is isolated in DrawHorizontalLineList. This makes it easy to add support for other graphics formats, such as the 8514/A, the XGA, or, for that matter, a non-PC system.

Books of the Month

Two books share honors this month. One isn't new, but is well worth having if you're a PC graphics programmer. Programmer's Guide to PC & PS/2 Video Systems, by Richard Wilton (Microsoft Press, 1987) is pretty much the standard graphics reference for the PC, covering all standards up through the VGA, including MCGA, EGA, Hercules, CGA, and MDA (but not 8514/A). For 8514/A programming, Graphics Programming for the 8514/A, by Jake Richter and Bud Smith (M&T Books, 1990), is good; which is lucky, because it's the only 8514/A book I know of!

- - - -

```
   point at (X2,Y2). If SkipFirst == 1, the point at (X1,Y1) isn't
   drawn; if SkipFirst == 0, it is. For each scan line, the pixel
   closest to the scanned edge without being to the left of the
   scanned edge is chosen. Uses an all-integer approach for speed and
   precision */

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

void ScanEdge(int X1, int Y1, int X2, int Y2, int SetXStart,
      int SkipFirst, struct HLine **EdgePointPtr)
{
   int Y, DeltaX, Height, Width, AdvanceAmt, ErrorTerm, i;
   int ErrorTermAdvance, XMajorAdvanceAmt;
   struct HLine *WorkingEdgePointPtr;

   WorkingEdgePointPtr = *EdgePointPtr; /* avoid double dereference */
   AdvanceAmt = ((DeltaX = X2 - X1) > 0) ? 1 : -1;
                           /* direction in which X moves (Y2 is
                              always > Y1, so Y always counts up) */

   if ((Height = Y2 - Y1) <= 0)  /* Y length of the edge */
      return;     /* guard against 0-length and horizontal edges */

   /* Figure out whether the edge is vertical, diagonal, X-major
      (mostly horizontal), or Y-major (mostly vertical) and handle
      appropriately */
   if ((Width = abs(DeltaX)) == 0) {
      /* The edge is vertical; special-case by just storing the same
         X coordinate for every scan line */
      /* Scan the edge for each scan line in turn */
      for (i = Height - SkipFirst; i-- > 0; WorkingEdgePointPtr++) {
         /* Store the X coordinate in the appropriate edge list */
         if (SetXStart == 1)
            WorkingEdgePointPtr->XStart = X1;
         else
            WorkingEdgePointPtr->XEnd = X1;
      }
   } else if (Width == Height) {
      /* The edge is diagonal; special-case by advancing the X
         coordinate 1 pixel for each scan line */
      if (SkipFirst) /* skip the first point if so indicated */
         X1 += AdvanceAmt; /* move 1 pixel to the left or right */
      /* Scan the edge for each scan line in turn */
      for (i = Height - SkipFirst; i-- > 0; WorkingEdgePointPtr++) {
         /* Store the X coordinate in the appropriate edge list */
         if (SetXStart == 1)
            WorkingEdgePointPtr->XStart = X1;
         else
            WorkingEdgePointPtr->XEnd = X1;
         X1 += AdvanceAmt; /* move 1 pixel to the left or right */
      }
   } else if (Height > Width) {
      /* Edge is closer to vertical than horizontal (Y-major) */
      if (DeltaX >= 0)
         ErrorTerm = 0; /* initial error term going left->right */
      else
         ErrorTerm = -Height + 1;   /* going right->left */
      if (SkipFirst) {   /* skip the first point if so indicated */
         /* Determine whether it's time for the X coord to advance */
         if ((ErrorTerm += Width) > 0) {
            X1 += AdvanceAmt; /* move 1 pixel to the left or right */
            ErrorTerm -= Height; /* advance ErrorTerm to next point */
         }
      }
      /* Scan the edge for each scan line in turn */
      for (i = Height - SkipFirst; i-- > 0; WorkingEdgePointPtr++) {
         /* Store the X coordinate in the appropriate edge list */
         if (SetXStart == 1)
            WorkingEdgePointPtr->XStart = X1;
         else
            WorkingEdgePointPtr->XEnd = X1;
         /* Determine whether it's time for the X coord to advance */
         if ((ErrorTerm += Width) > 0) {
```

```
            X1 += AdvanceAmt; /* move 1 pixel to the left or right */
            ErrorTerm -= Height; /* advance ErrorTerm to correspond */
         }
      }
   } else {
      /* Edge is closer to horizontal than vertical (X-major) */
      /* Minimum distance to advance X each time */
      XMajorAdvanceAmt = (Width / Height) * AdvanceAmt;
      /* Error term advance for deciding when to advance X 1 extra */
      ErrorTermAdvance = Width % Height;
      if (DeltaX >= 0)
         ErrorTerm = 0; /* initial error term going left->right */
      else
         ErrorTerm = -Height + 1;   /* going right->left */
      if (SkipFirst) {   /* skip the first point if so indicated */
         X1 += XMajorAdvanceAmt;    /* move X minimum distance */
         /* Determine whether it's time for X to advance one extra */
         if ((ErrorTerm += ErrorTermAdvance) > 0) {
            X1 += AdvanceAmt;       /* move X one more */
            ErrorTerm -= Height; /* advance ErrorTerm to correspond */
         }
      }
      /* Scan the edge for each scan line in turn */
      for (i = Height - SkipFirst; i-- > 0; WorkingEdgePointPtr++) {
         /* Store the X coordinate in the appropriate edge list */
         if (SetXStart == 1)
            WorkingEdgePointPtr->XStart = X1;
         else
            WorkingEdgePointPtr->XEnd = X1;
         X1 += XMajorAdvanceAmt;    /* move X minimum distance */
         /* Determine whether it's time for X to advance one extra */
         if ((ErrorTerm += ErrorTermAdvance) > 0) {
            X1 += AdvanceAmt;       /* move X one more */
            ErrorTerm -= Height; /* advance ErrorTerm to correspond */
         }
      }
   }

   *EdgePointPtr = WorkingEdgePointPtr;   /* advance caller's ptr */
}
```

[LISTING THREE]

```
; Draws all pixels in the list of horizontal lines passed in, in
; mode 13h, the VGA's 320x200 256-color mode. Uses REP STOS to fill
; each line.
; C near-callable as:
;     void DrawHorizontalLineList(struct HLineList * HLineListPtr,
;          int Color);
; All assembly code tested with TASM 2.0 and MASM 5.0

SCREEN_WIDTH  equ    320
SCREEN_SEGMENT       equ    0a000h

HLine  struc
XStart dw     ?      ;X coordinate of leftmost pixel in line
XEnd   dw     ?      ;X coordinate of rightmost pixel in line
HLine  ends

HLineList struc
Lngth  dw     ?      ;# of horizontal lines
YStart dw     ?      ;Y coordinate of topmost line
HLinePtr dw   ?      ;pointer to list of horz lines
HLineList ends

Parms  struc
               dw     2 dup(?)      ;return address & pushed BP
HLineListPtr   dw     ?             ;pointer to HLineList structure
Color          dw     ?             ;color with which to fill
Parms  ends
```

```
        .model small
        .code
        public _DrawHorizontalLineList
        align  2
_DrawHorizontalLineList     proc
        push   bp             ;preserve caller's stack frame
        mov    bp,sp          ;point to our stack frame
        push   si             ;preserve caller's register variables
        push   di
        cld                   ;make string instructions inc pointers

        mov    ax,SCREEN_SEGMENT
        mov    es,ax          ;point ES to display memory for REP STOS

        mov    si,[bp+HLineListPtr] ;point to the line list
        mov    ax,SCREEN_WIDTH      ;point to the start of the first scan
        mul    [si+YStart]          ; line in which to draw
        mov    dx,ax                ;ES:DX points to first scan line to
                                    ; draw
        mov    bx,[si+HLinePtr]     ;point to the XStart/XEnd descriptor
                                    ; for the first (top) horizontal line
        mov    si,[si+Lngth]        ;# of scan lines to draw
        and    si,si                ;are there any lines to draw?
        jz     FillDone             ;no, so we're done
        mov    al,byte ptr [bp+Color]      ;color with which to fill
        mov    ah,al                       ;duplicate color for STOSW
FillLoop:
        mov    di,[bx+XStart]              ;left edge of fill on this line
        mov    cx,[bx+XEnd]                ;right edge of fill
        sub    cx,di
        js     LineFillDone  ;skip if negative width
        inc    cx            ;width of fill on this line
        add    di,dx         ;offset of left edge of fill
        test   di,1          ;does fill start at an odd address?
        jz     MainFill      ;no
        stosb                ;yes, draw the odd leading byte to
                             ; word-align the rest of the fill
        dec    cx            ;count off the odd leading byte
        jz     LineFillDone  ;done if that was the only byte
MainFill:
        shr    cx,1          ;# of words in fill
        rep    stosw         ;fill as many words as possible
        adc    cx,cx         ;1 if there's an odd trailing byte to
                             ; do, 0 otherwise
        rep    stosb         ;fill any odd trailing byte
LineFillDone:
        add    bx,size Hline        ;point to the next line descriptor
        add    dx,SCREEN_WIDTH      ;point to the next scan line
        dec    si                   ;count off lines to fill
        jnz    FillLoop
FillDone:
        pop    di                   ;restore caller's register variables
        pop    si
        pop    bp                   ;restore caller's stack frame
        ret
_DrawHorizontalLineList     endp
        end
```

[LISTING FOUR]

```
; Scan converts an edge from (X1,Y1) to (X2,Y2), not including the
; point at (X2,Y2). If SkipFirst == 1, the point at (X1,Y1) isn't
; drawn; if SkipFirst == 0, it is. For each scan line, the pixel
; closest to the scanned edge without being to the left of the scanned
; edge is chosen. Uses an all-integer approach for speed & precision.
; C near-callable as:
;    void ScanEdge(int X1, int Y1, int X2, int Y2, int SetXStart,
;        int SkipFirst, struct HLine **EdgePointPtr);
; Edges must not go bottom to top; that is, Y1 must be <= Y2.
; Updates the pointer pointed to by EdgePointPtr to point to the next
;  free entry in the array of HLine structures.
```

```
HLine  struc
XStart dw     ?      ;X coordinate of leftmost pixel in scan line
XEnd   dw     ?      ;X coordinate of rightmost pixel in scan line
HLine  ends

Parms  struc
               dw     2 dup(?) ;return address & pushed BP
X1             dw     ?      ;X start coord of edge
Y1             dw     ?      ;Y start coord of edge
X2             dw     ?      ;X end coord of edge
Y2             dw     ?      ;Y end coord of edge
SetXStart      dw     ?      ;1 to set the XStart field of each
                             ; HLine struc, 0 to set XEnd
SkipFirst      dw     ?      ;1 to skip scanning the first point
                             ; of the edge, 0 to scan first point
EdgePointPtr   dw     ?      ;pointer to a pointer to the array of
                             ; HLine structures in which to store
                             ; the scanned X coordinates
Parms  ends

;Offsets from BP in stack frame of local variables.
AdvanceAmt       equ    -2
Height           equ    -4
LOCAL_SIZE       equ     4   ;total size of local variables

       .model small
       .code
       public _ScanEdge
       align  2
_ScanEdge      proc
       push   bp                   ;preserve caller's stack frame
       mov    bp,sp                ;point to our stack frame
       sub     sp,LOCAL_SIZE       ;allocate space for local variables
       push   si                   ;preserve caller's register variables
       push   di
       mov    di,[bp+EdgePointPtr]
       mov    di,[di]              ;point to the HLine array
       cmp    [bp+SetXStart],1     ;set the XStart field of each HLine
                                   ; struc?
       jz     HLinePtrSet          ;yes, DI points to the first XStart
       add    di,XEnd              ;no, point to the XEnd field of the
                                   ; first HLine struc
HLinePtrSet:
       mov    bx,[bp+Y2]
       sub    bx,[bp+Y1]           ;edge height
       jle    ToScanEdgeExit       ;guard against 0-length & horz edges
       mov    [bp+Height],bx       ;Height = Y2 - Y1
       sub    cx,cx                ;assume ErrorTerm starts at 0 (true if
                                   ; we're moving right as we draw)
       mov    dx,1                 ;assume AdvanceAmt = 1 (move right)
       mov    ax,[bp+X2]
       sub    ax,[bp+X1]           ;DeltaX = X2 - X1
       jzI    sVertical            ;it's a vertical edge--special case it
       jns    SetAdvanceAmt        ;DeltaX >= 0
       mov    cx,1                 ;DeltaX < 0 (move left as we draw)
       sub    cx,bx                ;ErrorTerm = -Height + 1
       neg    dx                   ;AdvanceAmt = -1 (move left)
       neg    ax                   ;Width = abs(DeltaX)
SetAdvanceAmt:
       mov    [bp+AdvanceAmt],dx
; Figure out whether the edge is diagonal, X-major (more horizontal),
; or Y-major (more vertical) and handle appropriately.
       cmp    ax,bx                ;if Width==Height, it's a diagonal edge
       jz     IsDiagonal           ;it's a diagonal edge--special case
       jb     YMajor               ;it's a Y-major (more vertical) edge
                                   ;it's an X-major (more horz) edge
       sub    dx,dx                ;prepare DX:AX (Width) for division
       div    bx                   ;Width/Height
                                   ;DX = error term advance per scan line
       mov    si,ax                ;SI = minimum # of pixels to advance X
                                   ; on each scan line
       test   [bp+AdvanceAmt],8000h  ;move left or right?
       jz     XMajorAdvanceAmtSet  ;right, already set
```

```
        neg     si                      ;left, negate the distance to advance
                                        ; on each scan line
XMajorAdvanceAmtSet:                    ;
        mov     ax,[bp+X1]              ;starting X coordinate
        cmp     [bp+SkipFirst],1        ;skip the first point?
        jz      XMajorSkipEntry         ;yes
XMajorLoop:
        mov     [di],ax                 ;store the current X value
        add     di,size Hline           ;point to the next HLine struc
XMajorSkipEntry:
        add     ax,si                   ;set X for the next scan line
        add     cx,dx                   ;advance error term
        jle     XMajorNoAdvance         ;not time for X coord to advance one
                                        ; extra
        add     ax,[bp+AdvanceAmt]      ;advance X coord one extra
        sub     cx,[bp+Height]          ;adjust error term back
XMajorNoAdvance:
        dec     bx                      ;count off this scan line
        jnz     XMajorLoop
        jmp     ScanEdgeDone
        align   2
ToScanEdgeExit:
        jmp     ScanEdgeExit
        align   2
IsVertical:
        mov     ax,[bp+X1]              ;starting (and only) X coordinate
        sub     bx,[bp+SkipFirst]       ;loop count = Height - SkipFirst
        jz      ScanEdgeExit            ;no scan lines left after skipping 1st
VerticalLoop:
        mov     [di],ax                 ;store the current X value
        add     di,size HLine           ;point to the next HLine struc
        dec     bx                      ;count off this scan line
        jnz     VerticalLoop
        jmp     ScanEdgeDone
        align   2
IsDiagonal:
        mov     ax,[bp+X1]              ;starting X coordinate
        cmp     [bp+SkipFirst],1        ;skip the first point?
        jz      DiagonalSkipEntry       ;yes
DiagonalLoop:
        mov     [di],ax                 ;store the current X value
        add     di,size Hline           ;point to the next HLine struc
DiagonalSkipEntry:
        add     ax,dx                   ;advance the X coordinate
        dec     bx                      ;count off this scan line
        jnz     DiagonalLoop
        jmp     ScanEdgeDone
        align   2
YMajor:
        push    bp                      ;preserve stack frame pointer
        mov     si,[bp+X1]              ;starting X coordinate
        cmp     [bp+SkipFirst],1        ;skip the first point?
        mov     bp,bx                   ;put Height in BP for error term calcs
        jz      YMajorSkipEntry         ;yes, skip the first point
YMajorLoop:
        mov     [di],si                 ;store the current X value
        add     di,size HLine           ;point to the next HLine struc
YMajorSkipEntry:
        add     cx,ax                   ;advance the error term
        jle     YMajorNoAdvance         ;not time for X coord to advance
        add     si,dx                   ;advance the X coordinate
        sub     cx,bp                   ;adjust error term back
YMajorNoAdvance:
        dec     bx                      ;count off this scan line
        jnz     YMajorLoop
        pop     bp                      ;restore stack frame pointer
ScanEdgeDone:
        cmp     [bp+SetXStart],1        ;were we working with XStart field?
        jz      UpdateHLinePtr          ;yes, DI points to the next XStart
        sub     di,XEnd                 ;no, point back to the XStart field
UpdateHLinePtr:
        mov     bx,[bp+EdgePointPtr]    ;point to pointer to HLine array
        mov     [bx],di                 ;update caller's HLine array pointer
ScanEdgeExit:
```

```
        pop     di              ;restore caller's register variables
        pop     si
        mov     sp,bp           ;deallocate local variables
        pop     bp              ;restore caller's stack frame
        ret
_ScanEdge       endp
        end
```

Journal: Dr. Dobb's Journal April 1991 v16 n4 p133(9)

The virtues of inexpensive approximation: the Edsun Continuous Edge Graphics DAC

Author: Abrash, Michael.

Summary: Edsun Laboratories Inc's Continuous Edge Graphics Digital-to-Analog Converter (DAC) features support for 8 bits-per-pixel (bpp) per color gun, compatibility with standard VGA DACs, capability to specify a pixel's color and capability for information to be embedded in the bitmap. The Continuous Edge Graphics DAC is complex to program and does not give users complete control over color, but its attributes are very substantial to any graphics programmer. The combination of the DAC with the VGA standard is likely to create its own high-end standard for computer graphics. The only problem with implementing the high-end graphics will be software support.

The Virtues of Inexpensive Approximation: The Edsun Continuous Edge Graphics DAC

A while back, Hal Hardenbergh (of DTACK Grounded, neural net, and Offramp fame) was good enough to help me figure out how to draw roundish objects rapidly. Circles were no problem for Hal; neither were ellipses with horizontal or vertical major axes. The sticking point was tilted ellipses.

The problem wasn't that Hal didn't know how to draw tilted ellipses; the problem was drawing them fast. No matter how he approached the problem, he always ended up stuck with one small term that was very expensive (that is, slow) to calculate. And there matters stood for a while.

One day, Hal called and said, "I can draw tilted ovals very fast. These are not ellipses. However, they look so much like ellipses that it's hard to tell them apart." Hal had replaced the expensive term with an easily calculated but slightly less accurate term. The resulting ovals were no longer mathematically ellipses--but they were sure as heck close.

Sometimes you really do need genuine ellipses--but not very often. Computer graphics is the art of approximating the ideal in such a way that the eye and brain together fill in the gaps, and in that context, a tilted ellipse-like oval will almost always serve as well as a true ellipse -- better, if the oval can be drawn faster than the ellipse.

In other words, an inexpensive approximation often beats an expensive ideal.

Which brings us to the Continuous Edge Graphics Digital-to-Analog Converter (CEG/DAC), from Edsun Laboratories.

The Edsun Continuous Edge DAC

The CEG/DAC, which may well impact the state of IBM PC graphics every bit as much as Super VGA did, is a triumph of inexpensive approximation over expensive perfection. For about $15 added cost (in quantity; projected to drop to around $5 within a year), otherwise off-the-shelf VGAs can approach and sometimes even surpass 24-bit-per-pixel (bpp) display quality, with stunning results. It is literally impossible to watch Edsun's demo and not lust for a CEG/DAC. The CEG/DAC is being touted as an antialiasing device, about which I have my doubts. (It vastly expands the number of colors available in a single VGA frame, but in a sharply limited fashion. More about this later.) It might be more accurate to call it a dejagging palette expansion device, but whatever it is, it works.

The CEG/DAC is less than ideal in several respects. It's complex to program, it doesn't give you complete color control, and software that takes advantage of CEG/DAC features is often slower than normal VGA software at the same resolution. Nonetheless, for good and sufficient reasons which I will

explain shortly, I think that a year from now the high-end VGA standard will be CEG/DAC-plus-Super VGA, much as Super VGA is the standard today. Once the price of the CEG/DAC drops a little, a Super VGA manufacturer would be nuts to build an adapter without CEG/DAC, a user would be foolish to buy a Super VGA sans CEG/DAC, and a graphics programmer would be right out of his gourd not to at least investigate the technology. Photorealistic CEG/DAC images look amazingly better than standard 256 color images; lines look like they stepped off the Planet of the Vector Displays (okay, there's a little striping, but we're picking very small nits here); polygons blend together seamlessly; detail is much easier to pick out; and text looks terrific on the CEG/DAC.

To put it simply, CEG/DAC graphics Just Plain Look Great. And they're cheap. Ignore that at your own risk.

How It Works

The CEG/DAC combines the following four attributes, none of which seems particularly profound:

* Compatibility with standard VGA DACs

* Support for 8 bpp per color gun

* Capability for information to be embedded in the bitmap to reprogram the palette on-the-fly

* Capability for a pixel's color to be specified as a weighting of adjacent colors on the same scan line

Together, these seemingly innocuous features may set the next PC graphics standard.

VGA DAC Compatibility

Like any VGA DAC, the CEG/DAC sits at the end of the VGA adapter's pixel pipeline, accepting 8-bit pixel attribute values from the VGA chip, converting them into red, green, and blue pixel colors, and sending the appropriate RGB voltage levels to the monitor. Until a specific sequence of OUTs kicks it into CEG mode, the CEG/DAC performs this pixel conversion just like the VGA-standard Inmos DAC; each pixel attribute is used as a look-up index in an internal 256-entry palette table, and the looked-up RGB value is sent to the monitor. The CEG/DAC is Inmos-compatible right down to the pins, so manufacturers can put CEG/DACs on VGAs without altering board layouts, BIOSs, or manufacturing processes. This, in combination with the relatively low price, instantly converts the CEG/DAC from a high-end item with a niche market to a mass-market part. Why? Because the CEG/DAC is an inexpensive, no-fuss way for VGA manufacturers to distinguish and add value to their products, a valuable attribute indeed in a market where every VGA is beginning to look like every other, and prices are falling through the floor.

I predict that there will be CEG/DACs on several hundred thousand VGAs within a year even if no one ever uses it in CEG mode. In years past, "1024 X 768" sold a lot of Super VGAs, even though the 30Hz interlacing used to reach that resolution was capable of frying optic nerves in a matter of minutes. Likewise, it's of immense value to manufacturers to be able to use the CEG/DAC to claim "740,000 colors" and "2048 X 2048 effective resolution" (that is, that CEG/DAC displays at 1024 X 768 are equivalent to normal displays at 2048 X 2048, an interesting concept to which we'll return another time). Let's not carry the analogy too far, because the CEG/DAC is far more useful than was the 30Hz 1024 X 768 mode; the point is that the actual utility of the CEG/DAC is almost superfluous to its widespread use, which seems virtually assured. That, in turn, means that the CEG/DAC should quickly reach the critical mass at which it becomes worthwhile for mass-market software to support it routinely -- and that's the definition of a PC standard in my book.

The Standard of Comparison: 24 Bits per Pixel

In order to understand the importance and limitations of the remaining CEG/DAC features, we must first look at the traditional high-end standard for color graphics: 24-bpp color selection, yielding 16,777,216 independently selectable colors per pixel, enough to convince the eye that it's looking at a continuous color spectrum, or at least that portion of the spectrum that a CRT can display. The obvious benefit of 24 bpp is that the colors of the primitives that make up graphics displays -- points, lines, polygons, and so on -- can be independently selected from a virtually unlimited palette, giving the programmer a free hand in color selection. It's worth noting, however, that you'll never need all 16,000,000-plus colors simultaneously; there are, after all, fewer than a million pixels even at 1024 X 768 resolution. Then, too, primitives are rarely drawn in a random mix of colors; typically, each primitive is drawn in a single color or a mix of a few colors. It's likely that at most only a few hundred completely unrelated "main" colors will be required for a typical display, although thousands of variations on and combinations of those colors may also be required.

The less obvious benefit of 24 bpp is that it allows for excellent antialiasing. Raster graphics attempts to render an image of essentially infinite resolution on a medium of decidedly finite resolution--the computer screen. The generally inadequate pixel-by-pixel rate at which the image is sampled onto the screen can result in considerable error (aliasing) in rendering the image. The most notable symptom of aliasing is jagged primitive edges.

Antialiasing is the process of reducing the error of the displayed image relative to the original (ideal) image, in any of a number of ways, a topic to which we'll return in the future. For now, what's important to understand about antialiasing is that it creates attractive, smooth displays that are good representations of ideal images, and that it is generally performed by selecting the color of a given pixel as a function of the colors surrounding that pixel in the ideal image. As a very simple example, pixels that fall right on the boundary between a white polygon and a black polygon could be weighted 50 percent white and 50 percent black, as shown in Figure 1. While the resulting image has the same resolution as if antialiasing had not been performed, the eye perceives the gray pixels as blending into the adjacent polygons when displayed on the screen, and the edges appear to be smooth.

There are three points to be made here. First, 24 bpp is perfect for antialiasing because by making all colors available, it allows just the right color weighting to be selected for each pixel. Second, antialiasing does not involve selection of random pixel colors, but rather of pixel colors closely related to the colors of adjoining pixels. Antialiasing does not require new main colors, but rather variations on the main colors used to draw the primitives; there is considerable color coherence between neighboring pixels. Third, antialiasing need not be perfect to be useful. Merely eliminating jagged edges is a major step in the direction of visually appealing displays, regardless of the mathematical correctness of the procedure by which this is done.

24 bpp, though theoretically ideal, suffers from one major shortcoming: It's expensive. It requires both a great deal of memory (2.5 Mbyte for 1024 x 768 24-bpp), and hardware designed to handle the vast amounts of video data that must be pumped out to the screen. (It's worth noting, though, that relatively inexpensive 15 bit-per-pixel VGAs, built around the Tseng Labs ET4000 chip, should be available soon.) Performance also suffers from the need to manipulate larger bitmaps, and often from the increased demands that scanning all that video data places on memory bandwidth, as well. High-resolution adapters tend to be plagued by similar problems. The CEG/DAC doesn't provide either 24 bpp or higher resolution, but neither does it require any extra memory or additional hardware, and it places no additional demands on memory bandwidth.

8 bpp per Color Gun

The standard VGA DAC supports only 6 bpp per color gun, but in CEG mode the CEG/DAC provides a full 8 bpp per gun. This extra color information is, however, made available in the normal way, via the 256-entry palette look-up table (albeit with 24-bit RGB entries), so this feature alone doesn't increase the number of colors available at one time.

Reprogramming the Palette On-The-Fly

The CEG/DAC allows information embedded in the bitmap to reprogram the palette on-the-fly, a feature Edsun calls EDP, for Edsun Dynamic Palette. EDP expands the available color set by allowing the palette to be treated as a dynamic resource during the course of a frame; you can, if you want, have a completely different palette available for the hundredth scan line than you had for the first. However, EDP is not so useful as it may initially appear to be, because in order to reprogram the palette, at least six pixels in a row must display the same color, as follows:

The CEG/DAC normally accepts pixel values from the VGA and looks up the corresponding RGB values, just as a standard VGA DAC does. However, in CEG modes, certain pixel values are treated as commands rather than as pixel attributes; Table 1 shows the list of commands in Advanced-8 mode (more on this shortly). A portion of the palette is given over to these commands, so fewer pixel colors than normal can be selected directly through the palette; only 223 main pixel colors are available in Advanced-8 mode, and fewer still in other modes. Nonetheless, the total number of colors available in a single frame is vastly increased by the use of command values to derive new colors from the main colors and to change the palette on-the-fly. (The command-driven nature of the CEG/DAC is the reason a VGA adapter doesn't have to be changed if a CEG/DAC is installed. The bitmap is laid out in exactly the same way as usual; only the interpretation of some of the values in the bitmap changes, and the CEG/DAC does that interpretation. The VGA sends pixel values from the bitmap to the DAC just as it always does, without the faintest notion of the new meanings of some of the values in CEG mode.)

When a pixel set to the EDP command value is received by the DAC, the next four pixel values are taken to specify the new red, green, and blue values for the palette location to load, and the index of that location. While this is going on, the DAC has no new pixel information to display; the data stream from the VGA that should have provided the pixel values is instead providing the palette load values, as shown in Figure 2. The CEG/DAC compensates by displaying the last known pixel color for the duration of the palette load, so each EDP load results in six identical pixels in a row.

EDP loads can be handled without impacting the displayed image by sacrificing a portion of one or both sides of the bitmap to make a solid border; the EDP commands can be placed in this part of the bitmap without affecting the image, which is displayed in the remaining portion of the bitmap. Sacrificing five percent of the bitmap for a left edge border and five percent for a right border in 1024 x 768 mode allows 20 palette entries to be reprogrammed on every line. It would certainly be better if EDP commands could be executed during normal border time, but the VGA chip doesn't send information from the bitmap to the DAC at that time. A special VGA chip could easily be built that would allow more efficient palette loading -- but this way the CEG/DAC requires no modification to existing VGAs, and that characteristic is key to the success of the CEG/DAC.

Another way to handle EDP loads is to perform them in bitmap locations where there just happen to be at least six pixels in a row of the same color. This is obviously a complex proposition, especially if the screen is constantly changing. Using the CEG/DAC effectively is without a doubt a nontrivial programming exercise; the CEG/DAC makes high-quality graphics possible, but the bulk of the work in actually realizing such graphics falls on the software.

The quick-minded among you will no doubt note that EDP doesn't actually make any more colors simultaneously available; they're available in the same frame, but you're still limited to 223 main colors at any one time. True enough, but rarely will you need more than 200 main colors on a single scan line.

Pixel Weighting

The basic premise of pixel weighting is simple: Any pixel that's set to a pixel-weighting command value is drawn as the specified mix of the colors (not the attributes, but the final, looked-up, 24-bit colors) of the nearest pixels to the left and right. (See Table 1.) If either neighboring pixel is also a command rather than a color, the color of the nearest pixel that is a color is used. (Pixel weighting actually follows a more complicated set of

rules, particularly for line drawing and multipixel weighting sequences, as described in Edsun's CEG Level 3 Specification, but conceptually the above description will do.) This allows 32 weightings of colors, ranging in steps of 1/31 from 100 percent one color to 100 percent the other; oddly, this does not allow weightings of exactly 50-50. These weightings tend to be well suited to smoothing edges and boundaries, as illustrated by Figure 3. Similarly, pixel weighting tends to provide useful colors when color gradients are drawn.

Pixel weighting is truly the key to the CEG/DAC. The advertised 740,000 colors result from pixel weighting: 742,813 is the exact number, derived as each of the 223 main colors mixed in any of 30 ways with the 222 other main colors, divided by two because half the combinations are duplicates, plus the 223 unweighted main colors themselves. (The number of available colors is slightly higher if EDP is disabled, because an additional pixel value is then freed up for use as an attribute. If EDP is enabled, additional colors can be made available by loading the palette on-the-fly.) Pixel weighting doesn't actually allow you to select any arbitrary one out of 742,813 colors for any given pixel; in fact, you still have a choice of only one of at most 256 colors for any one pixel. However, if you can get by with 223 or fewer main colors (bearing in mind that EDP can change the main colors), the weightings available at any given pixel are useful for smoothing edges or shading, so you get a good portion (although by no means all) of the utility that 740,000 simultaneous colors would offer.

Pixel weighting is scarcely a perfect solution. Many antialiasing algorithms consider not only horizontally but also vertically adjacent pixels; pixel weighting takes no account of those pixels at all. In general, points at which three or more colors need to be taken into account aren't amenable to pixel weighting. What's more, because pixel weighting is left-to-right sequence dependent, it doesn't allow for proper weighting of the leftmost pixel on a scan line, and it can't always handle weighting sequences that are disrupted by intersecting primitives. (Edsun has, however, put in special cases that allow, for example, a near-vertical single-width line to cross a weighting sequence without disrupting the sequence. See the CEG Level 3 Specification for details on the special case mix rules, which are fairly complex.)

What pixel weighting amounts to is horizontal color mixing; as such, it provides many but not all of the colors needed for ideal antialiasing. (EDP can patch "holes" in the available color set, but only to a limited extent, because of the requirement for six consecutive pixels of the same color and because it must also be used to change main colors as necessary.) Nonetheless, CEG/DAC graphics look good even when mathematically correct antialiasing isn't possible; Lord knows, they do expand the color palette and get rid of the jaggies.

Like a stereo with the bass cranked up or an overbright TV, a CEG/DAC image can look terrific even if it isn't an ideal representation of the original image. I wouldn't call the CEG/DAC an "antialiasing" device in the classic sense, but what it does is useful in its own right; in fact, I expect new drawing and rendering approaches to spring up around it. To quote Foley and van Dam, from the Second Edition of Computer Graphics (page 646): "A simple or fast algorithm may be used if it produces attractive effects, even if no justification can be found in the laws of physics."

Pixel Weighting Details

There are actually three distinct CEG modes. The one described above is Advanced-8 mode, available in all 8-bpp modes. There's also Advanced-4 mode, used in 4-bpp display modes, which is similar but offers only seven or eight main colors (depending on whether EDP is enabled) and eight weightings. (See Table 2.) The final mode, Basic-8, also available in 8-bpp modes, is quite different. (See Table 3.) Instead of requiring that a pixel contain either a color attribute or a command, each Basic-8 pixel contains both a 4-bit color attribute and a one-of-eight pixel weighting; EDP is not available in Basic-8. The color attribute is mixed with an earlier pixel, possibly but not necessarily the preceding pixel. Because it is simpler and more flexible than Advanced-8, Basic-8 is useful for applications that need no more than 16 main colors.

Pros and Cons

It's unquestionably more difficult to program a CEG/DAC bitmap than a normal VGA or 24-bpp adapter. For general drawing with a full set of primitives, you must consider the surrounding state of the bitmap as you draw, because you may partially overwrite pixel weighting sequences already in the bitmap, producing unintended effects. Such is the cost of sequence-dependent bitmap encoding.

CEG/DAC code also often runs slower than standard VGA code at the same resolution, because there's more calculation to be performed for each pixel. However, CEG/DAC code is generally faster than equivalent code would be running on an adapter with 24 bpp or two to four times the resolution, because those approaches require that more bytes of display memory be manipulated per pixel or that many more pixels be drawn. Also, it's not necessary that you use pixel weightings and EDP everywhere on the screen; you may choose to use only the 223 main colors for most of the screen, and reserve CEG drawing for a logo, or for a few special icons.

On balance, the CEG/DAC involves considerable added software complexity, provides slower performance at a given resolution, and doesn't fit standard antialiasing, drawing, and rendering models particularly well. On the other hand, it makes wonderful graphics possible, and is faster and much, much cheaper than higher-resolution or higher-color displays. Basically, the burden with the CEG/DAC falls entirely on the programmer, who has to deal with the complex bitmap and must come up with drawing models suited to this odd chip. As far as the user is concerned, it looks good and costs little--a winner across the board.

And remember, users pay the freight.

Gamma Correction

Edsun has built into the CEG/DAC "gamma correction"--the process of generating a desired percentage of full brightness when mixing colors--making for accurate mixing, and saving programmers a major headache in calculating color mixes. Interestingly, gamma correction plus pixel weighting makes the CEG/DAC able to produce useful color components other than the 256 normally selectable for each color gun via the palette look-up table. If a 50-50 pixel weighting (actually, 16/31 to 15/31, which is as close as the CEG/ DAC can come to 50-50 because of the 1/31 pixel weighting step) is specified between a pixel with a red color component of 254 and another of 255, a red gamma-corrected component with a brightness halfway between the two will be sent to the monitor. In this respect, the CEG/DAC exceeds the color-generation capabilities of even 24-bpp adapters.

How to Develop for the CEG/DAC

To develop for the CEG/DAC, you'll need a VGA with a CEG/DAC, along with Edsun's basic documentation: Edsun Continuous Edge Graphics/ D-to-A Converters Data Sheet and Edsun Continuous Edge Graphics Level 3 Specification. You'll also want to get Edsun's CEG Software Development Kit (SDK), a $250 library of CEG-aware graphics functions--lines, ellipses, CEG control, and the like--and sample code. Applications developed with the SDK are royalty-free. SDK source code is available at no cost if you sign a nondisclosure and license agreement that allows you to distribute object code royalty-free.

The SDK is adequate to get started with CEG programming, but it isn't a complete graphics library; for instance, there's no support for clipping. Neither is the SDK particularly well optimized for performance; for example, a little probing with a debugger revealed that polygons are filled one dot at a time via calls to a draw-pixel subroutine. I found the SDK to be functional, but something of a chore to use, partly because the manual was of marginal assistance in dealing with the many unfamiliar CEG-specific functions; more explanatory and overview material would help greatly. The SDK is a good way to get started with CEG/DAC programming, and will no doubt improve with time (like the rest of us, it will take some time for Edsun's programmers to master CEG), but right now, don't expect too much from it. Personally, I'd get the SDK source code and treat that as no more than a

starting point. Clever programmers willing to take the time to understand the hardware and program the CEG bitmap directly will surely be able to work wonderful tricks with both display quality and performance.

The SDK comes with some utilities, including several to convert from Targa to CEG format and back; image manipulation tools are on the way for the next SDK revision. However, there are currently no tools to let you edit icons or prototype screens. That, I suspect, will change in a hurry; if you're looking for an interesting and potentially lucrative project, you could do worse than to build a first-rate CEG/DAC image editor.

Microsoft Windows CEG drivers will be available soon, so for Windows applications, the benefits of the CEG/DAC will be automatic with driver installation.

Implications

The CEG/DAC is an odd and complex technology, and it will be some time before its full potential is known. Areas for further research include maximizing performance, exploring the possibilities of the CEG/DAC's extremely fine color control, and allocating EDP loads in a changing bitmap, which seems to me a first-rate optimization problem.

Given that CEG/DAC programming makes for more attractive lines, edges, text, and images at little cost, I think that CEG/DAC-plus-Super VGA is likely to set the high-end mass-market standard, at least until XGA arrives in force, and possibly longer. CEG could well be applied to an XGA DAC clone, or, for that matter, to 15-bpp VGAs or non-PC systems. CEG/DAC-plus-Super VGA may dominate the market between vanilla VGA and Targa-level adapters until low-cost 24-bit-per-pixel graphics become available, a matter of a few years at least.

Is the ascendancy of the CEG/DAC a sure thing? Hardly. I do think that Super VGA manufacturers will put it into play; they can't afford not to. The problem is software: The chip is unfamiliar, limited in many respects, and just plain hard to program. Still, for all its imperfections, the CEG/DAC makes possible some very desirable graphics that were formerly flat-out impossible for mass-market software--and that sort of potential doesn't stay untapped in the PC arena for very long.

Journal: Dr. Dobb's Journal May 1991 v16 n5 p131(7)

Further ruminations of the Edsun CEG/DAC

Author: Abrash, Michael.
AttFile: GP-MAY91.ASC Source code listing.

Summary: Edsun Laboratories' CEG/DAC (Continuous Edge Graphics Digital-to-Analog Converter) (CEG/DAC) enables a VGA system to display graphics at near 24-bit-per-pixel (bpp) quality, but is better suited to static than dynamic images. The CEG/DAC is not a true 24-bpp device but achieves its performance by embedding information to reprogram the palette in the bitmap and pixel weighting, a process of specifying pixel colors as weighted mixes of adjacent pixels. Pixel weighting also works well for eliminating jaggedness in lines in static images but does not work well for performing temporal aliasing for sequences of animated images. Methods for working around CEG/DAC's limitations in generating dynamic images are discussed. Several routines demonstrate the visual capabilities of the technology.

If late, I find myself asking many questions, but finding few easy answers. How do lawyers manage to innovate without the incentive and protection of patents for their work? Why is this nation capable of mounting an awesome attack half a world away in just a couple of months, but unable to devise a coherent energy policy in ten years? Why does anyone buy a 386*387 machine when a 486 costs just about the same and is two to five times faster at floating point operations? I wonder about these things, I do. And, for lo these many weeks since last we spoke, I have wondered this: What is the place, in the grand scheme of things, of the Edsun Continuous Edge Graphics Digital-toAnalog-Converter (CEG/DAC)? What is it that this chip can and cannot do?

The CEG/DAC

As I discussed last month, the CEG/DAC is a VGA DAC substitute that makes a VGA capable of displaying graphics of near 24-bit-per-pixel (bpp) quality. This is accomplished via two features: Embedding in the bitmap information that reprograms the palette (EDP), and specifying pixel colors as mixes of the colors of neighboring pixels (pixel weighting). I stated last month that I think that Super VGA-plus-CEG/DAC may well be the next PC graphics standard. i still think that. The question I now have is: What sorts of graphics are the features of the CEG/DAC good for, out of the many things a PC programmer might wish for?

Were the CEG/DAC a true 24-bpp device, there'd be no question; it would be able to display any sort of graphics at all, limited only by the speed of the graphics code. However, the CEG/DAC is not a true 24-bpp device, and although sometimes it is a good substitute, sometimes it clearly is not-and often it is somewhere in between. I have come to think that the CEG/DAC is more useful for static images than for dynamic images (animation, real-time screen updates, and the like); in particular, the limitations of pixel weighting must be attended to carefully when animating.

For example, consider the simple task of drawing a line. Drawing Lines Through Space The standard technique for drawing lines rapidly is Bresenham's Algorithm. Bresenham-style lines suffer from one flaw: They tend to look jagged, because the line jumps abruptly from one pixel to the next.

The traditional cure for jaggles is antialiasing. Technically, antialiasing is the process of removing spurious signals resulting from undersampling, typically with a low-pass filter, but in the context of graphics, antialiasing has come to mean any process that helps eliminate jaggies. The CEG/DAC's pixel weighting feature is designed to do exactly that, by allowing you to mix the colors of horizontally neighboring pixels.

Pixel weighting works very well for smoothing jaggies in a single, static image. Run Listing Two (page 149), which when linked to Listing One (page 149) draws a rotating nonantialiased line floating point rather than Bresenham's is used for clarity and to save space, but the same points are

drawn), and press a key to freeze the image; you will see prominent jaggies along the length of the line. Now run Listing Three (page 150) linked to Listing One (if you have a CEG/DAC) or Listing Four (page 150) linked to Listing One (if you don't) to draw an antialiased line, and press a key to freeze the image. (Listing Three antialiases using CEG/DAC pixel weighting, as described later, and Listing Four emulates pixel weighting by setting up the VGA palette with 32 weighted mixes of the line and background colors.) The jaggies are gone, replaced with smooth edges. There are two sorts of aliasing in graphics, though. What we've just seen is spatial aliasing-aliasing over space, in this case over the pixels of the screen. There's also temporal aliasing, aliasing from one image to the next as animation is performed. When Listing Two is allowed to run freely, temporal aliasing is clearly visible in the form of jaggies crawling wildly along the edges of the line. It is this distracting and often hideous effect that drives the need for temporal antialiasing.

Drawing Lines Over Time

Pixel weighting doesn't serve particularly well for temporal antialiasing, for a number of reasons. The line animation performed by Listings Three and Four looks better than that of Listing Two, but there are still shifting patterns in the line, the line's brightness varies, and the line's motion appears less than perfectly smooth. There are several reasons for this, two of which have nothing to do with the CEG/DAC: The aspect ratio in 320 x 200 mode isn't 1: 1, and the fineness and accuracy of motion is limited by the use of integer coordinates. (Then again, pixel weighting is poorly suited to drawing based on fractional coordinates, because such drawing must often represent three or more color influences on a single pixel.) However, a number of other problems are directly related to the nature of the CEG/DAC.

For one thing, the line's brightness varies with angle. This is a side effect of the way the CEG/DAC handles Y-major lines (lines that are longer along the Y axis than the X axis). Normally, the CEG/DAC requires at least three pixels in a row to specify a single weighted pixel: Two colors to average between and a pixel weighting, which averages between the two colors, as shown in Figure 1. (I'm speaking of Advanced-8 mode here, the mode in which the maximum number of colors is available.) Requiring three or more pixels across for pixel weighting in this fashion would make it impossible to draw thin lines, especially in close proximity.

Edsun solved this by introducing a special case. If a pixel color is followed by a pixel mix and then by another color, the first color is taken to specify a line color, and the mix is taken to speci6, the weighting between the line color and the background color for the two pixels of the line, as shown in Figure 2. The mix itself is applied to the pixel that specified the line color, and the complement of the mix is applied to the pixel that specified the mix. in other words, a color-mix pair in the middle of a line of colors is taken to be a standalone unit specifying a cross-section of a two-pixel-wide line, and the mix is applied to spread the color across the two pixels in such a way that the intensity of the line color totals 100 percent. (Neither is this the only special case; CEG/DAC programming is not trivial.)

A total line intensity of 100 percent on each scan line sounds good, but in fact it is rarely correct. A vertical line should indeed have 100 percent intensity across each scan line. However, as the line tilts toward 45 degrees, the intensity per scan line should increase toward @2, because the line is covering more ground from one scan line to the next. With pixel weighting, however, the intensity per scan line remains the same, so the overall brightness of the line decreases.

Also, the width of a line drawn with 100 percent-sum pixel weighting varies with angle; horizontal and vertical lines are only one pixel wide, but all other lines spread across two pixels along the minor axis (the axis along which the line's length is less). The line's width, measured along the normal, also varies along the length of the line; this shows up as motion along the edges of the line as it turns, although it is more subdued than normal jaggies. The above caveats are also true of static images drawn with pixel weighting, but there they are not nearly as noticeable. I've said before that graphics is the art of fooling the eye and the brain. They're easily fooled for man things., for example, they're quite willing to interpret a sequence of slightly different images, displayed at a rate of 30 images per second, as motion. By the same token, however, disturbances in

the smooth progression of change over time stick out like red flags. Don't ask me why-, probably it was a handy trait to have when spotting prey or avoiding being prey was a full time job. At any rate, consistency from one frame to the next is essential.

Unfortunately, temporal consistency requires more than simply smoothing edges; it requires filtering the image so that spurious elements are removed that is, true antialiasing. With true antialiasing, a consistent representation of the image can be obtained from one frame to the next, no matter how the image is rotated or shifted, or what it is bordered by. I'd love to show you true antialiasing in action, but I'm well and truly running out of space; I promise to return to the topic soon.

True antialiasing requires that colors be settable as arbitrary mixes of the surrounding primitives in all directions. Pixel weighting does not make that possible. Therefore, except under special circumstances, pixel weighting is a dejagging rather than antialiasing tool, and is useful primarily for smoothing static images (where it surely does look good), rather than for smoothing dynamic images.

Further Limitations of Pixel Weighting There are other problems with dynamic pixel weighting. Programming is a cumbersome process, for starters. Consider Listing Three. For Y-major lines, the weighting between the two pixels on each scan line has to be calculated; while that can certainly be done without floating-point calculations, I do not see how it can be done without either an integer divide or a table search. Either way, lines drawn with pixel weighting are going to be considerably slower than Bresenham's lines.

X-major lines have their complications, too. Here the minor axis is vertical, so we must split the 100 percent brightness at each step between two scan lines. Unfortunately, the CEG/DAC is strictly a horizontally oriented device, so we are obliged to make sure that the line color precedes the mix sequence on each scan line, as shown in Figure 1. (The pixel establishing the line color actually displays in the previous pixel's color-, the line color doesn't show up until the first mix pixel, in the form of the specified mix.) There's another complication-, if any scan line has only one mix command, then that mix and the preceding color are interpreted as a special-case two pixel line mix, as described earlier, rather than a normal mix sequence, and the colors displayed for the pixels are quite different. These complications could have been handled as special cases in Listing Three; however, I decided that the code would be simpler and the operation of the CEG/ DAC would be clearer if I just drew all the mix pixels, then processed the buffer afterward to insert leading line color pixels and clean up one-mix sequences.

If you think all this sounds involved and slow, well, it is. it's by no means unmanageable, and certainly much more efficient implementations are possible, but CEG drawing does, nonetheless, generally require additional coding and execution time. Moreover, I didn't have to worry about handling intersections with other lines and graphics objects in Listing Three; normally, such intersections have to be checked for and potentially patched up, because they can generate unintended results by disrupting existing mix sequences. In addition, there are potential problems with left-side clipping of pixel weighting sequences.

There are, of course, workarounds for many of these limitations. If no rotation is performed and all moves are by an integral number of pixels, edge crawling largely vanishes. Careful construction of animation so that objects don't overlap can eliminate the complications of checking the bitmap for intersections. Also, many primitives, such as rectangle fills, don't require any antialiasing, and antialiased text can be predefined, then simply copied to the screen as needed.

In fact, predefining antialiased images and avoiding troublesome situations such as rotated polygons, inter sections with other objects, and clip ping may be keys to high-performance CEG/DAC code. When such condition are unavoidable, you may want to consider EDP (on-the-fly palette loading), which is probably the best bet for high color-content dynamic drawing with the CEG/DAC. (Basic-8 mode is simpler and faster when only a few colors are required.)

Consider this: EDP does not suffer from clipping problems. Neither must drawing that relies only on EDP concern itself with intersections; every

pixel is independent. The only remaining problem lies in making available all the colors needed for drawing. Suppose that in 640 x 480 256-color mode, you choose to allocate 160 pixels at the left and right edges of the screen (about 25 percent of the width of the screen) to static images, composed of mostly solid colors. EDP requires five pixels to load a single palette location, so about 30 colors could be changed per scan line during the static edges. Additional colors could be set in the active portion of the display, with the EDP commands embedded in areas of solid color.

Now, there would be 480 pixels in the active region across each scan line. 223 independent colors (the number available in Advanced-8 mode), with at least 30 changeable from line to line, could often serve quite nicely to draw whatever it is that those 480 pixels across are supposed to represent. No, it wouldn't be the same as drawing with 24 arbitrary bpp, but, properly structured.. it might not look a whole lot different; Listing Four, which uses only careful palette loading, antialiases just as well as the CEG hardware in Listing Three, and, with the help of EDP, would be considerably more flexible. (Not coincidentally, the photorealistic images in the Edsun demo have large blank borders.) Best of all, the only complications with EDP-based drawing would be figuring out which colors to change and where for maximum effect; the far greater complications of pixel weighting are avoided.

Of course, reliance on EDP is not a technique that a general-purpose driver can use; it's necessary that the application be CEG/DAC-aware. Long ago, all graphics applications were hardware aware; now quasi-hardware-independence, through general interfaces such as Windows and X Window System, is the mainstream. I suspect that the truly mind-blowing CEG/DAC applications will be throwbacks to that earlier era. in the end, what I'm saying is that if you want to do spectacular animation on the CEG/DAC, you might do well to ignore pixel weighting altogether, at least for images that must be generated in realtime or might intersect, and focus on EDP. That was obviously not Edsun's intent, for they didn't make it possible to enable only EDP; that's a shame, because then we'd have 255 main colors to work with.

Overall, I think the CEG/DAC is better suited to static images than dynamic; as the Edsun demo illustrates, the CEG/DAC is stunningly good at static displays, and as the examples in this column illustrate, there are significant complications and limitations associated with dynamic displays. One of the tricks is converting as much drawing as possible from dynamic to static; that's what predefining images and keeping them from intersecting is all about. Although the CEG/DAC is a powerful resource, in some respects it comes up short; it should be fascinating to watch as techniques evolve to work around its limitations.

Book of the Month

This month's book is The RenderMan Companion, by Steve Upstill (AddisonWesley, 1990). RenderMan is a comprehensive programming interface specification for 3-D graphics, implementations of the RenderMan interface have been the basis for stunning photorealistic imaging and special effects. (For background information on RenderMan, see Upstill's article "Photorealism in Computer Graphics," in DDJ, November 1988). Companion takes you on a wide-ranging tour of the RenderMan interface, with plenty of sample code and output; even if you never program RenderMan directly, this book provides worthwhile insight into the nature of 3-D rendering. At the very least, read the foreword, a brief history of computer graphics and the development of RenderMan; it provides a sense of the dizzying pace of progress in computer graphics, and of the people behind the wonders.

Which brings a thought to mind. I don't know how many, if any, of the techniques discussed in Companion and in the various standard graphics references are patented, but I've never seen a mention of such. How is it, I ask all you software patent proponents, that all this invention, publishing, and sharing of knowledge for the public benefit happened without the necessary" incentive and protection of patents? Think about it, that's all I ask.

- - - -

[LISTING ONE]

```
/* Animates a line rotating about its center. 180 frames are constructed
offscreen, one for each 1 degree rotation, and then copied to the screen to
produce animation. Edsun CEG antialiasing or emulated CEG antialiasing may
optionally be performed. To compile for Edsun CEG antialiasing, define
USE_CEG on the compiler command line (/DUSE_CEG for MSC, -DUSE_CEG for Turbo C)
and link this code with Listing 3. To compile for emulated CEG antialiasing,
define EMULATE_CEG on the compiler command line and link this code with
Listing 4. All C code tested with Microsoft C 5.0. Requires a large data model
(the compact model was used for testing). */

#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <math.h>
#include <dos.h>
#ifdef __TURBOC__
#include <mem.h>
#else
#include <memory.h>
#endif

/* Size of frames drawn to system memory, and size of the circle
   formed by the rotation of the line segment around its center */
#define FRAME_WIDTH     49
#define FRAME_HEIGHT    49
#define FRAME_CENTER_X  (FRAME_WIDTH/2)
#define FRAME_CENTER_Y  (FRAME_HEIGHT/2)
#define X_RADIUS        (FRAME_WIDTH/2-1)
#define Y_RADIUS        (FRAME_HEIGHT/2-1)
#define SCREEN_WIDTH    320
#define SCREEN_HEIGHT   200
#define SCREEN_SEGMENT  0xA000
#define X_OFFSET_ADJUST ((SCREEN_WIDTH/2) - FRAME_CENTER_X)
#define Y_OFFSET_ADJUST ((SCREEN_HEIGHT/2) - FRAME_CENTER_Y)
#define PI              3.141592

int main(void);
extern void DrawLine(unsigned char *,int,int,int,int,int,int,int);
extern int SetCEGMode(int);
extern void SetAAPalette(void);

int main() {
   unsigned char *FramePtr[180], *ScreenPtr, *TempPtr;
   int angle, X1, Y1, X2, Y2, temp, i;
   union REGS regset;

   /* Generate 180 frames, one for each 1 degree rotation of the line
      segment about its center; store the results in system memory */
   printf("Precalculating frames. Please wait...");
   /* First, allocate space for the frames */
   for (angle = 0; angle < 180; angle++) {
      if ((FramePtr[angle] = (unsigned char *)
            malloc(FRAME_WIDTH * FRAME_HEIGHT)) == NULL) {
         printf("Out of memory\n");
         return(1);
      }
      /* Clear the frame to black */
      memset(FramePtr[angle], 0, FRAME_WIDTH * FRAME_HEIGHT);
   }

   /* Generate the frames, one for each 1 degree rotation of the line
      segment about its center */
   for (angle = 0; angle < 180; angle++) {
      /* Calculate upper end of line as a counterclockwise rotation from right
         end of a horizontal line and lower end of line as a counterclockwise
         rotation from left end of a horizontal line */
      temp = cos((double)angle * PI / 180.0) * X_RADIUS + 0.5;
      X1 = FRAME_CENTER_X + temp;
      X2 = FRAME_CENTER_X - temp;
      temp = sin((double)angle * PI / 180.0) * Y_RADIUS + 0.5;
      Y1 = FRAME_CENTER_Y - temp;
```

```
      Y2 = FRAME_CENTER_Y + temp;
      /* Draw the line in white */
      DrawLine(FramePtr[angle], FRAME_WIDTH, FRAME_HEIGHT, X1, Y1,
            X2, Y2, 15);
   }

   /* Set to the standard 256-color VGA mode, mode 0x13, 320x200 */
   regset.x.ax = 0x0013; int86(0x10, &regset, &regset);

#ifdef USE_CEG
   /* Enable Advanced-8 CEG mode */
   if (!SetCEGMode(13)) {
   /* Restore text mode and we're done */
      regset.x.ax = 0x0003;
      int86(0x10, &regset, &regset);
      fprintf(stderr, "CEG/DAC not installed\n");
      return(1);     /* no CEG/DAC installed */
   }
#endif

#ifdef EMULATE_CEG
   /* Set the palette up for antialiasing */
   SetAAPalette();
#endif

   /* Draw the frames, at a rate of 1 frame per screen refresh
      interval, until a key is pressed */
   for (angle = 0;;) {
    do {
      /* Point to the destination area of display memory */
#ifdef __TURBOC__
      ScreenPtr = MK_FP(SCREEN_SEGMENT, (Y_OFFSET_ADJUST*SCREEN_WIDTH)
            + X_OFFSET_ADJUST);
#else
      FP_SEG(ScreenPtr) = SCREEN_SEGMENT;
      FP_OFF(ScreenPtr) =
            (Y_OFFSET_ADJUST * SCREEN_WIDTH) + X_OFFSET_ADJUST;
#endif
      /* Wait for the start of the vertical non-display portion of the
         frame */
      while (inp(0x3DA) & 0x08) ;
      while (!(inp(0x3DA) & 0x08)) ;
      /* Copy over the current frame, a scan line at a time */
      for (i = 0, TempPtr = FramePtr[angle]; i < FRAME_HEIGHT;
            i++, ScreenPtr += SCREEN_WIDTH, TempPtr += FRAME_WIDTH) {
         memcpy(ScreenPtr, TempPtr, FRAME_WIDTH);
      }
      angle = (angle + 1) % 180; /* wrap back every 180 frames */
    } while (!kbhit());
    if (getch() == 0x1B)    /* pause if a key was pressed */
      break;                /* exit if the key was Esc */
    getch();                /* wait for another key and resume */
   }

   /* Return the CEG/DAC to standard VGA operation by writing to
      palette location 223, restore text mode and we're done */
#ifdef USE_CEG
   while (inp(0x3DA) & 0x08) ;      /* wait for the start of */
   while (!(inp(0x3DA) & 0x08)) ;   /* vertical non-display */
   outp(0x3C8, 223); outp(0x3C9, 0); outp(0x3C9, 0); outp(0x3C9, 0);
#endif
   regset.x.ax = 0x0003; int86(0x10, &regset, &regset);
   return(0);
}
```

[LISTING TWO]

```
/* Draws a non-antialiased line from (X1,Y1) to (X2,Y2) into the buffer
pointed to by BufferPtr, of width BufferWidth and height BufferHeight. */

#include <dos.h>
#include <math.h>
#define SWAP(a,b) {temp = a; a = b; b = temp;}
```

```
void DrawLine(unsigned char *BufferPtr, int BufferWidth,
      int BufferHeight, int X1, int Y1, int X2, int Y2, int color)
{
   int X, Y, DeltaX, DeltaY, temp;
   double Slope, InverseSlope;

   /* Calculate the X and Y lengths of the line */
   DeltaX = X2 - X1;
   DeltaY = Y2 - Y1;

   /* Determine the major axis */
   if (abs(DeltaY) > abs(DeltaX)) {
      /* Y is the major axis */
      if (DeltaY < 0) {    /* make sure DeltaY is positive */
         SWAP(X1, X2);
         SWAP(Y1, Y2);
         DeltaX = -DeltaX;
         DeltaY = -DeltaY;
      }
      InverseSlope = (double)DeltaX / (double)DeltaY;
      /* Scan out the line, stepping along the Y axis one pixel at a
         time and calculating the corresponding X coordinates */
      for (Y = Y1; Y <= Y2; Y++) {
         X = X1 + (int)floor(((double)(Y - Y1) * InverseSlope) + 0.5);
         *(BufferPtr + BufferWidth * Y + X) = color;
      }
   } else {
      /* X is the major axis */
      if (DeltaX < 0) {    /* make sure DeltaX is positive */
         SWAP(X1, X2);
         SWAP(Y1, Y2);
         DeltaX = -DeltaX;
         DeltaY = -DeltaY;
      }
      Slope = (double)DeltaY / (double)DeltaX;
      /* Scan out the line, stepping along the X axis one pixel at a
         time and calculating the corresponding Y coordinates */
      for (X = X1; X <= X2; X++) {
         Y = Y1 + (int)floor(((double)(X - X1) * Slope) + 0.5);
         *(BufferPtr + BufferWidth * Y + X) = color;
      }
   }
}
```

[LISTING THREE]

```
/* Draws an Advanced-8 CEG-antialiased line from (X1,Y1) to (X2,Y2) into buffer
pointed to by BufferPtr, of width BufferWidth and height BufferHeight. */

#include <dos.h>
#include <math.h>
#define SWAP(a,b) {temp = a; a = b; b = temp;}

void DrawLine(unsigned char *BufferPtr, int BufferWidth,
      int BufferHeight, int X1, int Y1, int X2, int Y2, int color)
{
   int X, Y, DeltaX, DeltaY, temp, WeightingIndex, i, MixLength;
   double Slope, InverseSlope, XFloat, YFloat;

   /* Calculate X and Y lengths of the line */
   DeltaX = X2 - X1;
   DeltaY = Y2 - Y1;

   /* Determine the major axis */
   if (abs(DeltaY) > abs(DeltaX)) {
      /* Y is the major axis */
      if (DeltaY < 0) {    /* make sure DeltaY is positive */
         SWAP(X1, X2);
         SWAP(Y1, Y2);
         DeltaX = -DeltaX;
         DeltaY = -DeltaY;
      }
```

```
   InverseSlope = (double)DeltaX / (double)DeltaY;
   /* Scan out line, stepping along the Y axis 1 pixel at a time and
      calculating corresponding pair of X coordinates, 1 on each side of
      ideal line, with the mix between background color and line color at
      each of the two X coordinates proportional to proximity of line to
      that pixel, and with line color intensities of the two X coordinates
      summing to 100% */
   for (Y = Y1; Y <= Y2; Y++) {
      /* Exact X coordinate at this Y coordinate */
      XFloat = (double)X1 + ((double)(Y - Y1) * InverseSlope);
      /* Nearest X coordinate on or to the left of the line */
      X = (int)floor(XFloat);
      /* Draw the color to the left pixel */
      *(BufferPtr + BufferWidth * Y + X) = color;
      /* Draw the weighting index for the desired color mix for the
         left pixel to the right pixel; the CEG/DAC uses this to
         mix the color we just wrote with the background color and
         draw that for the left pixel, then uses the complementary
         mix to draw the right pixel (this pixel). Confusing, but
         that's the way the CEG/DAC works! */

      *(BufferPtr + BufferWidth * Y + X + 1) =
            (int)((XFloat - X) * 32.0) + 192;
   }
} else {
   /* X is the major axis */
   if (DeltaX < 0) {    /* make sure DeltaX is positive */
      SWAP(X1, X2);
      SWAP(Y1, Y2);
      DeltaX = -DeltaX;
      DeltaY = -DeltaY;
   }
   Slope = (double)DeltaY / (double)DeltaX;
   /* Scan out the line, stepping along the X axis one pixel at a
      time and calculating the corresponding pair of Y coordinates,
      one on each side of the ideal line, with the mix between the
      background color and the line color at each of the two Y
      coordinates proportional to the proximity of the line to that
      pixel, and with the line color intensities of the two Y
      coordinates summing to 100% */
   for (X = X1; X <= X2; X++) {
      /* Exact Y coordinate at this X coordinate */
      YFloat = (double)Y1 + ((double)(X - X1) * Slope);
      /* Nearest Y coordinate on or above the line */
      Y = (int)floor(YFloat);
      /* Calculate the weighting index for the percentage of this
         pixel that's on the top scan line of the pair this pixel
         is split between */
      WeightingIndex = (int)((YFloat - Y) * 32.0);
      /* Set the weighting for the top pixel */
      *(BufferPtr + BufferWidth * Y + X) =
            WeightingIndex + 192;
      /* Set the weighting for the bottom pixel, with the top and
         bottom weightings summing to 100% */
      *(BufferPtr + BufferWidth * (Y + 1) + X) =
            31 - WeightingIndex + 192;
   }

   /* Finally, post-process the buffer to put leading mix color
      bytes on mix sequences (so the weighting indexes have
      something to mix), and to turn one-pixel-wide sequences
      (artifacts of the above drawing approach, which will not
      display properly) into two-pixel-wide sequences by appending
      an additional mixed pixel 100% weighted to the background color */
   for (i = 0, MixLength = 0; i < (BufferWidth*BufferHeight); i++) {
      if (*(BufferPtr + i) != 0) {
         /* Part of a mix sequence; increment sequence length */
         MixLength++;
      } else {
         if (MixLength > 0) {
            /* Mix sequence just ended; set the line color to start
               the mix sequence */
               *(BufferPtr + i - MixLength - 1) = color;
            if (MixLength == 1)
```

```
                  /* This is a 1-long mix sequence; pad it to a 2-long
                     sequence with an all-background mixed pixel, so
                     the mix sequence will display properly, rather
                     than as a special 2-wide color/mix line case */
                  *(BufferPtr + i) = 0xDF;
               MixLength = 0;
            }
         }
      }
   }
}

/* Sets the desired CEG/DAC mode, enabling CEG graphics. Returns 1 for
   success, 0 if no CEG/DAC is installed. */
int SetCEGMode(int mode) {
   /* Wait for the start of the vertical non-display portion of the
      frame */
   while (inp(0x3DA) & 0x08) ;
   while (!(inp(0x3DA) & 0x08)) ;

   outp(0x3C7, 222);    /* write the CEG enable sequence */
   outp(0x3C9, 'C'); outp(0x3C9, 'E'); outp(0x3C9, 'G');
   outp(0x3C7, 222);
   outp(0x3C9, 'E'); outp(0x3C9, 'D'); outp(0x3C9, 'S');
   outp(0x3C7, 222);
   outp(0x3C9, 'U'); outp(0x3C9, 'N');
   outp(0x3C9, mode);   /* write the CEG mode */

   /* CEG mode should be enabled. Make sure this is a CEG/DAC */
   outp(0x3C6, 0xFF);   /* enable all DAC mask bits */
   if ((inp(0x3C6) & 0x70) == 0x70)
      return(0);  /* no version # bit is 0; this is not a CEG/DAC */
   else
      return(1);  /* this is a CEG/DAC, and it's ready to go */
}
```

[LISTING FOUR]

```
/* Draws an antialiased line from (X1,Y1) to (X2,Y2) into the buffer pointed
to by BufferPtr, of width BufferWidth and height BufferHeight. White on black
is the only supported color combination. */

#include <dos.h>
#include <math.h>
#define SWAP(a,b) {temp = a; a = b; b = temp;}

void DrawLine(unsigned char *BufferPtr, int BufferWidth,
      int BufferHeight, int X1, int Y1, int X2, int Y2, int color)
{
   int X, Y, DeltaX, DeltaY, temp, WeightingIndex, i, MixLength;
   double Slope, InverseSlope, XFloat, YFloat;

   /* Calculate X and Y lengths of the line */
   DeltaX = X2 - X1;
   DeltaY = Y2 - Y1;

   /* Determine the major axis */
   if (abs(DeltaY) > abs(DeltaX)) {
      /* Y is the major axis */
      if (DeltaY < 0) {    /* make sure DeltaY is positive */
         SWAP(X1, X2);
         SWAP(Y1, Y2);
         DeltaX = -DeltaX;
         DeltaY = -DeltaY;
      }
      InverseSlope = (double)DeltaX / (double)DeltaY;
      /* Scan out line, stepping along the Y axis one pixel at a time and
         calculating corresponding pair of X coordinates, 1 on each side of
         ideal line, with the mix between background color and line color at
         each of the two X coordinates proportional to proximity of line to
         that pixel, and with line color intensities of two X coordinates
         summing to 100% */
      for (Y = Y1; Y <= Y2; Y++) {
```

```
         /* Exact X coordinate at this Y coordinate */
         XFloat = (double)X1 + ((double)(Y - Y1) * InverseSlope);
         /* Nearest X coordinate on or to the left of the line */
         X = (int)floor(XFloat);
         /* Calculate the weighting index for the percentage of this
            pixel that's in the left column of the pair this pixel is
            split between */
         WeightingIndex = (int)((XFloat - X) * 32.0);
         /* Draw the left pixel with the desired color weighting */
         *(BufferPtr + BufferWidth * Y + X) = WeightingIndex + 16;
         /* Draw the right pixel with the complement of the left pixel
            color weighting, so that the line color intensities sum to 100% */
         *(BufferPtr + BufferWidth * Y + X + 1) =
               31 - WeightingIndex + 16;
      }
   } else {
      /* X is the major axis */
      if (DeltaX < 0) {    /* make sure DeltaX is positive */
         SWAP(X1, X2);
         SWAP(Y1, Y2);
         DeltaX = -DeltaX;
         DeltaY = -DeltaY;
      }
      /* Scan out the line, stepping along the X axis one pixel at a
         time and calculating the corresponding pair of Y coordinates,
         one on each side of the ideal line, with the mix between the
         background color and the line color at each of the two Y
         coordinates proportional to the proximity of the line to that
         pixel, and with the line color intensities of the two Y
         coordinates summing to 100% */
      Slope = (double)DeltaY / (double)DeltaX;
      for (X = X1; X <= X2; X++) {
         /* Exact Y coordinate at this X coordinate */
         YFloat = (double)Y1 + ((double)(X - X1) * Slope);
         /* Nearest Y coordinate on or above the line */
         Y = (int)floor(YFloat);
         /* Calculate the weighting index for the percentage of this
            pixel that's on the top scan line of the pair this pixel
            is split between */
         WeightingIndex = (int)((YFloat - Y) * 32.0);
         /* Draw the top pixel with the desired color weighting */
         *(BufferPtr + BufferWidth * Y + X) =
               WeightingIndex + 16;
         /* Draw the bottom pixel with the complement of the top pixel
            color weighting, so that the line color intensities sum to
            100% */
         *(BufferPtr + BufferWidth * (Y + 1) + X) =
               31 - WeightingIndex + 16;
      }
   }
}

/* Sets palette entries 16-47 for antialiasing, stepping from solid white to
solid black (increments of 1/31st.) Steps are corrected for a gamma of 2.3. */
void SetAAPalette() {
   union REGS regset;
   struct SREGS sregs;
   static unsigned char AASettings[32*3] = {
      63,63,63, 62,62,62, 61,61,61, 60,60,60, 59,59,59, 58,58,58,
      57,57,57, 56,56,56, 55,55,55, 54,54,54, 53,53,53, 52,52,52,
      51,51,51, 50,50,50, 49,49,49, 47,47,47, 46,46,46, 45,45,45,
      43,43,43, 42,42,42, 40,40,40, 39,39,39, 37,37,37, 35,35,35,
      33,33,33, 31,31,31, 28,28,28, 26,26,26, 23,23,23, 19,19,19,
      14,14,14,  0, 0, 0
   };

   regset.x.ax = 0x1012;
   regset.x.bx = 0x0010;
   regset.x.cx = 0x0020;
   regset.x.dx = (unsigned int) AASettings;
   segread(&sregs);
   sregs.es = sregs.ds;       /* point ES:DX to AASettings */
   int86x(0x10, &regset, &regset, &sregs);
}
```

Journal: Dr. Dobb's Journal June 1991 v16 n6 p139(6)

Of songs, taxes, and the simplicity of complex polygons

Author: Abrash, Michael.
AttFile: GP-JUN91.ASC C & Assembler source code.

Summary: Graphics programming techniques are presented that fill three types of arbitrary polygons: complex, meaning there are intersecting edges; convex, meaning a continuous curving line could touch all vertices in the order they are defined; and nonconvex, meaning there are no intersecting edges and the polygon is not complex. Complex is a superset of nonconvex, which itself is a superset of convex. Complex polygons require the slowest fill strategy, though that strategy will fill any type of polygon. Nonconvex polygons are filled with less sorting. Convex polygons can be filled the fastest by merely scanning the two main sides of the polygon. Polygon filling begins with the idea that intersections between all polygon edges and each scan line are identified and, and the spans between intersections are filled. A program listing implements the fill technique for complex polygons.

Every so often, my daughter asks me to sing her to sleep. (If you've ever heard me sing, this may cause you concern about either her hearing or her judgement, but love knows no bounds.) As any parent is well aware, singing a young child to sleep can easily take several hours, or until sunrise, which ever comes last. One night, running low on children's songs, I switched to a Beatles medley, and at long last her breathing became slow and regular. At the end, I softly sang "A Hard Day's Night," then quietly stood up to leave. As I tiptoed out, she said, in a voice not even faintly tinged with sleep, "Dad, what do they mean, 'working like a dog'? Chasing a stick? That doesn't make sense; people don't chase sticks."

That led us into a discussion of idioms, which made about as much sense to her as an explanation of quantum mechanics. Finally, I fell back on my standard explanation of the Universe, which is that a lot of the time it doesn't make sense.

As a general principle, that explanation holds up remarkably well. (In fact, having just done my taxes, I think Earth is actually run by blob-creatures from the planet Mrxx, who are helplessly doubled over with laughter at the ridiculous things they can make us do. "Let's make them get Social Security numbers for their pets next year!" they're saying right now, gasping for breath.) Occasionally, however, one has the rare pleasure of finding a corner of the Universe that makes sense, where everything fits together as if preordained.

Filling arbitrary polygons is such a case.

Filling Arbitrary Polygons

In the February column, I described three types of polygons: convex, nonconvex, and complex. The RenderMan Companion, which I mentioned last month, has an intuitive definition of convex: If a rubber band stretched around a polygon touches all vertices in the order they're defined, then the polygon is convex. If a polygon has intersecting edges, it's complex. If a polygon doesn't have intersecting edges but isn't convex, it's nonconvex. Nonconvex is a special case of complex, and convex is a special case of nonconvex. (Which, I'm well aware, makes nonconvex a lousy name--noncomplex would have been better--but I'm following X Window System nomenclature here.)

The reason for distinguishing between these three types of polygons is that the more specialized types can be filled with markedly faster approaches. Complex polygons require the slowest approach; however, that approach will serve to fill any polygon of any sort. Nonconvex polygons require less sorting, because edges never cross. Convex polygons can be filled fastest of all by simply scanning the two sides of the polygon, as we saw in March.

Before we dive into complex polygon filling, I'd like to point out that the code in this article, like all polygon filling code I've ever seen, requires that the caller describe the type of the polygon to be filled. Often, however, the caller doesn't know what type of polygon it's passing, or specifies complex for simplicity, because that will work for all polygons; in such a case, the polygon filler will use the slow complex-fill code even if the polygon is, in fact, a convex polygon.

Although I've never seen it mentioned anywhere, it is reasonably easy to determine whether a polygon specified as complex or nonconvex is actually convex. The best technique I've come up with is tracing around the polygon's boundary, counting the number of times that the boundary reverses X and Y directions. If the boundary reverses both directions no more than twice, the polygon is convex. Whether the faster drawing of convex polygons justifies the extra time required to count X and Y reversals depends on both the implementation and the number of polygons in any particular application which are specified as complex/nonconvex but are actually convex.

Active Edges

The basic premise of filling a complex polygon is that for a given scan line, we determine all intersections between the polygon's edges and that scan line and then fill the spans between the intersections, as shown in Figure 1. (Section 3.6 of Computer Graphics, second edition, by Foley and van Dam provides an overview of this and other aspects of polygon filling.) There are several rules that might be used to determine which spans are drawn and which aren't; we'll use the odd/even rule, which specifies that drawing turns on after odd-numbered intersections (first, third, and so on) and off after even-numbered intersections.

The question then becomes how we can most efficiently determine which edges cross each scan line and where. As it happens, there is a great deal of coherence from one scan line to the next in a polygon edge list, because each edge starts at a given Y coordinate and continues unbroken until it ends. In other words, edges don't leap about and stop and start randomly; the X coordinate of an edge at one scan line is a consistent delta from that edge's X coordinate at the last scan line, and that is consistent for the length of the line.

This allows us to reduce the number of edges that must be checked for intersection; on any given scan line, we only need to check for intersections with the currently active edges--edges that start on that scan line, plus all edges that start on earlier (above) scan lines and haven't ended yet--as shown in Figure 2. This suggests that we can proceed from the top scan line of the polygon to the bottom, keeping a running list of currently active edges--called the Active Edge Table (AET)--with the edges sorted in order of ascending X coordinate of intersection with the current scan line. Then we can simply fill each scan line in turn according to the list of active edges at that line.

Maintaining the AET from one scan line to the next involves three steps. First, we must add to the AET any edges that start on the current scan line, making sure to keep the AET X-sorted for efficient odd/even scanning. Second, we must remove edges that end on the current scan line. Third, we must advance the X coordinates of active edges with the same sort of error term-based, Bresenham's-like approach we used for convex polygons, again ensuring that the AET is X-sorted after advancing the edges.

Advancing the X coordinates is easy. For each edge, we'll store the current X coordinate and all required error term information, and we'll use that to advance the edge one scan line at a time; then we'll resort the AET by X coordinate as needed. Removing edges as they end is also easy; we'll just count down the length of each active edge on each scan line and remove an edge when its count reaches zero. Adding edges as their tops are encountered is a tad more complex. While there are a number of ways to do this, one particularly efficient approach is to start out by putting all the edges of the polygon, sorted by increasing Y coordinate, into a single list, called the Global Edge Table (GET). Then, as each scan line is encountered, all edges at the start of the GET that begin on the current scan line are moved to the AET; because the GET is Y-sorted, there's no need to search the entire GET. For still greater efficiency, edges in the GET that share common Y coordinates can be sorted by increasing X coordinate; this ensures that no

more than one pass through the AET per scan line is ever needed when adding new edges from the GET in such a way as to keep the AET sorted in ascending X order.

What form should the GET and AET take? Linked lists of edge structures, as shown in Figure 3. With linked lists, all that's required to move edges from the GET to the AET as they become active, sort the AET, and remove edges that have been fully drawn is the exchanging of a few pointers.

In summary, we'll initially store all the polygon edges in Y-primary/X-secondary sort order in the GET, complete with initial X and Y coordinates, error terms and error term adjustments, lengths, and directions of X movement for each edge. Once the GET is built, we'll do the following:

1. Set the current Y coordinate to the Y coordinate of the first edge in the GET.

2. Move all edges with the current Y coordinate from the GET to the AET, removing them from the GET and maintaining the X-sorted order of the AET.

3. Draw all odd-to-even spans in the AET at the current Y coordinate.

4. Count down the lengths of all edges in the AET, removing any edges that are done, and advancing the X coordinates of all remaining edges in the AET by one scan line.

5. Sort the AET in order of ascending X coordinate.

6. Advance the current Y coordinate by one scan line.

7. If either the AET or GET isn't empty, go to step 2.

That's really all there is to it. Compare Listing One (page 154) to the fast convex polygon filling code from March, and you'll see that, contrary to expectation, complex polygon filling is indeed one of the more sane and sensible corners of the universe.

Complex Polygon Filling: An Implementation

Listing One shows a function, FillPolygon, that fills polygons of all shapes. If CONVEX[underscore]FILL[underscore]LINKED is defined, then the fast convex fill code from March is linked in and used to draw convex polygons. Otherwise, convex polygons are handled as if they were complex. Nonconvex polygons are also handled as complex, although this is not necessary, as discussed shortly.

Listing One is a faithful implementation of the complex polygon filling approach just described, with separate functions corresponding to each of the tasks, such as building the GET and X-sorting the AET. Listing Two (page 156) provides the actual drawing code used to fill spans, built on a draw pixel routine that is the only hardware dependency in the C code. Listing Three (page 156) is the header file for the polygon filling code; note that it is an expanded version of the header file used by the fast convex polygon fill code from March. Listing Four (page 156) is a sample program that, when linked to Listings One and Two, demonstrates drawing polygons of various sorts.

Listing Four illustrates several interesting aspects of polygon filling. The first and third polygons drawn illustrate the operation of the odd/even fill rule. The second polygon drawn illustrates how holes can be created in seemingly solid objects; an edge runs from the outside of the rectangle to the inside, the edges comprising the hole are defined, and then the same edge is used to move back to the outside; because the edges join seamlessly, the rectangle appears to form a solid boundary around the hole.

The set of V-shaped polygons drawn by Listing Four demonstrate that polygons sharing common edges meet but do not overlap. This characteristic, which I discussed at length several months back, is not a trivial matter; it allows polygons to fit together without fear of overlapping or missed pixels. Listing One reflects a fairly complex rule for drawing pixels on polygon boundaries that I have devised. It's not essential that I detail that rule or its implementation in Listing One (which is fortunate, for I lack the

space to do so), but it's important that you know that it exists, and that, as a result, Listing One should always fill polygons so that common boundaries and vertices are drawn once and only once. This has the side-effect for any individual polygon of not drawing pixels that lie exactly on top or right boundaries or at certain vertices.

By the way, I have not seen polygon boundary filling handled this way elsewhere. The boundary filling approach in Foley and van Dam is similar, but seems to me to not draw all boundary and vertex pixels once and only once.

More On Active Edges

Edges of zero height--horizontal edges and edges defined by two vertices at the same location--never even make it into the GET in Listing One. A polygon edge of zero height can never be an active edge, because it can never intersect a scan line; it can only run along the scan line, and the span it runs along is defined not by that edge but by the edges that connect to its endpoints.

Performance Considerations

How fast is Listing One? When drawing triangles on a 20-MHz 386, it's less than one-fifth the speed of the fast convex polygon fill code. However, most of that time is spent drawing individual pixels; when Listing Two is replaced with the fast assembler line segment drawing code in Listing Five (page 157), performance improves by two and one-half times, to about half as fast as the fast convex fill code. Even after conversion to assembly in Listing five, DrawHorizontalLineSeg still takes more than half of the total execution time, and the remaining time is spread out fairly evenly over the various subroutines in Listing One. Consequently, there's no single place in which it's possible to greatly improve performance, and the maximum additional improvement that's possible is clearly considerably less than two times; for that reason, and because of space limitations, I'm not going to convert the rest of the code to assembly. However, when filling a polygon with a great many edges, and especially one with a great many active edges at one time, relatively more time would be spent traversing the linked lists. Then conversion to assembly (which actually lends itself very nicely to linked list processing) could pay off reasonably well.

The algorithm used to X-sort the AET is an interesting performance consideration. Listing One uses a bubble sort, usually a poor choice for performance. However, bubble sorts perform well when the data are already almost sorted, and because of the X coherence of edges from one scan line to another, that's generally the case with the AET. An insertion sort might be somewhat faster, depending on the state of the AET when any particular sort occurs, but a bubble sort will generally do just fine.

An insertion sort that scans backward through the AET from the current edge rather than forward from the start of the AET could be quite a bit faster, because edges rarely move more than one or two positions through the AET. However, scanning backward requires a doubly linked list, rather than the singly linked list used in Listing One. I've chosen to use a singly linked list partly to minimize memory requirements (double-linking requires an extra pointer field) and partly because supporting back links would complicate the code a good bit. The main reason, though, is that the potential rewards for the complications of back links and insertion sorting aren't great enough; profiling a variety of polygons reveals that less than ten percent of total time is spent sorting the AET. The potential 1-5 percent speedup gained by optimizing AET sorting just isn't worth it in any but the most demanding application -- a good example of the need to keep an overall perspective when comparing the theoretical characteristics of various approaches.

Nonconvex Polygons

Nonconvex polygons can be filled somewhat faster than complex polygons. Because edges never cross or switch positions with other edges once they're in the AET, the AET for a nonconvex polygon needs to be sorted only when new edges are added. In order for this to work, though, edges must be added to

the AET in strict left-to-right order. Complications arise when dealing with two edges that start at the same point, because slopes must be compared to determine which edge is leftmost. This is certainly doable, but because of space limitations and limited performance returns, I haven't implemented this in Listing One.

Coming Up

Next time, we may do some 256-color animation. Or we may poke into the innards of the new 15-bpp VGAs. Or perhaps we'll take a look at RenderMan. Who knows? If you have any preferences, by all means drop me a line.

- -

GRAPHICS PROGRAMMING COLUMN
by Michael Abrash

[LISTING ONE]

```
/* Color-fills an arbitrarily-shaped polygon described by VertexList.
If the first and last points in VertexList are not the same, the path
around the polygon is automatically closed. All vertices are offset
by (XOffset, YOffset). Returns 1 for success, 0 if memory allocation
failed. All C code tested with Turbo C++.
If the polygon shape is known in advance, speedier processing may be
enabled by specifying the shape as follows: "convex" - a rubber band
stretched around the polygon would touch every vertex in order;
"nonconvex" - the polygon is not self-intersecting, but need not be
convex; "complex" - the polygon may be self-intersecting, or, indeed,
any sort of polygon at all. Complex will work for all polygons; convex
is fastest. Undefined results will occur if convex is specified for a
nonconvex or complex polygon.
Define CONVEX_CODE_LINKED if the fast convex polygon filling code from
the February 1991 column is linked in. Otherwise, convex polygons are
handled by the complex polygon filling code.
Nonconvex is handled as complex in this implementation. See text for a
discussion of faster nonconvex handling */

#include <stdio.h>
#include <math.h>
#ifdef __TURBOC__
#include <alloc.h>
#else    /* MSC */
#include <malloc.h>
#endif
#include "polygon.h"

#define SWAP(a,b) {temp = a; a = b; b = temp;}

struct EdgeState {
   struct EdgeState *NextEdge;
   int X;
   int StartY;
   int WholePixelXMove;
   int XDirection;
   int ErrorTerm;
   int ErrorTermAdjUp;
   int ErrorTermAdjDown;
   int Count;
};

extern void DrawHorizontalLineSeg(int, int, int, int);
extern int FillConvexPolygon(struct PointListHeader *, int, int, int);
static void BuildGET(struct PointListHeader *, struct EdgeState *,
   int, int);
static void MoveXSortedToAET(int);
static void ScanOutAET(int, int);
static void AdvanceAET(void);
static void XSortAET(void);
```

```
/* Pointers to global edge table (GET) and active edge table (AET) */
static struct EdgeState *GETPtr, *AETPtr;

int FillPolygon(struct PointListHeader * VertexList, int Color,
      int PolygonShape, int XOffset, int YOffset)
{
   struct EdgeState *EdgeTableBuffer;
   int CurrentY;

#ifdef CONVEX_CODE_LINKED
   /* Pass convex polygons through to fast convex polygon filler */
   if (PolygonShape == CONVEX)
      return(FillConvexPolygon(VertexList, Color, XOffset, YOffset));
#endif

   /* It takes a minimum of 3 vertices to cause any pixels to be
      drawn; reject polygons that are guaranteed to be invisible */
   if (VertexList->Length < 3)
      return(1);
   /* Get enough memory to store the entire edge table */
   if ((EdgeTableBuffer =
         (struct EdgeState *) (malloc(sizeof(struct EdgeState) *
         VertexList->Length))) == NULL)
      return(0);  /* couldn't get memory for the edge table */
   /* Build the global edge table */
   BuildGET(VertexList, EdgeTableBuffer, XOffset, YOffset);
   /* Scan down through the polygon edges, one scan line at a time,
      so long as at least one edge remains in either the GET or AET */
   AETPtr = NULL;    /* initialize the active edge table to empty */
   CurrentY = GETPtr->StartY; /* start at the top polygon vertex */
   while ((GETPtr != NULL) || (AETPtr != NULL)) {
      MoveXSortedToAET(CurrentY);  /* update AET for this scan line */
      ScanOutAET(CurrentY, Color); /* draw this scan line from AET */
      AdvanceAET();                /* advance AET edges 1 scan line */
      XSortAET();                  /* resort on X */
      CurrentY++;                  /* advance to the next scan line */
   }
   /* Release the memory we've allocated and we're done */
   free(EdgeTableBuffer);
   return(1);
}

/* Creates a GET in the buffer pointed to by NextFreeEdgeStruc from
   the vertex list. Edge endpoints are flipped, if necessary, to
   guarantee all edges go top to bottom. The GET is sorted primarily
   by ascending Y start coordinate, and secondarily by ascending X
   start coordinate within edges with common Y coordinates */
static void BuildGET(struct PointListHeader * VertexList,
      struct EdgeState * NextFreeEdgeStruc, int XOffset, int YOffset)
{
   int i, StartX, StartY, EndX, EndY, DeltaY, DeltaX, Width, temp;
   struct EdgeState *NewEdgePtr;
   struct EdgeState *FollowingEdge, **FollowingEdgeLink;
   struct Point *VertexPtr;

   /* Scan through the vertex list and put all non-0-height edges into
      the GET, sorted by increasing Y start coordinate */
   VertexPtr = VertexList->PointPtr;   /* point to the vertex list */
   GETPtr = NULL;    /* initialize the global edge table to empty */
   for (i = 0; i < VertexList->Length; i++) {
      /* Calculate the edge height and width */
      StartX = VertexPtr[i].X + XOffset;
      StartY = VertexPtr[i].Y + YOffset;
      /* The edge runs from the current point to the previous one */
      if (i == 0) {
         /* Wrap back around to the end of the list */
         EndX = VertexPtr[VertexList->Length-1].X + XOffset;
         EndY = VertexPtr[VertexList->Length-1].Y + YOffset;
      } else {
         EndX = VertexPtr[i-1].X + XOffset;
         EndY = VertexPtr[i-1].Y + YOffset;
      }
      /* Make sure the edge runs top to bottom */
      if (StartY > EndY) {
```

```
            SWAP(StartX, EndX);
            SWAP(StartY, EndY);
         }
         /* Skip if this can't ever be an active edge (has 0 height) */
         if ((DeltaY = EndY - StartY) != 0) {
            /* Allocate space for this edge's info, and fill in the
               structure */
            NewEdgePtr = NextFreeEdgeStruc++;
            NewEdgePtr->XDirection =   /* direction in which X moves */
                  ((DeltaX = EndX - StartX) > 0) ? 1 : -1;
            Width = abs(DeltaX);
            NewEdgePtr->X = StartX;
            NewEdgePtr->StartY = StartY;
            NewEdgePtr->Count = DeltaY;
            NewEdgePtr->ErrorTermAdjDown = DeltaY;
            if (DeltaX >= 0)  /* initial error term going L->R */
               NewEdgePtr->ErrorTerm = 0;
            else              /* initial error term going R->L */
               NewEdgePtr->ErrorTerm = -DeltaY + 1;
            if (DeltaY >= Width) {     /* Y-major edge */
               NewEdgePtr->WholePixelXMove = 0;
               NewEdgePtr->ErrorTermAdjUp = Width;
            } else {                   /* X-major edge */
               NewEdgePtr->WholePixelXMove =
                     (Width / DeltaY) * NewEdgePtr->XDirection;
               NewEdgePtr->ErrorTermAdjUp = Width % DeltaY;
            }
            /* Link the new edge into the GET so that the edge list is
               still sorted by Y coordinate, and by X coordinate for all
               edges with the same Y coordinate */
            FollowingEdgeLink = &GETPtr;
            for (;;) {
               FollowingEdge = *FollowingEdgeLink;
               if ((FollowingEdge == NULL) ||
                     (FollowingEdge->StartY > StartY) ||
                     ((FollowingEdge->StartY == StartY) &&
                     (FollowingEdge->X >= StartX))) {
                  NewEdgePtr->NextEdge = FollowingEdge;
                  *FollowingEdgeLink = NewEdgePtr;
                  break;
               }
               FollowingEdgeLink = &FollowingEdge->NextEdge;
            }
         }
      }
   }

   /* Sorts all edges currently in the active edge table into ascending
      order of current X coordinates */
   static void XSortAET() {
      struct EdgeState *CurrentEdge, **CurrentEdgePtr, *TempEdge;
      int SwapOccurred;

      /* Scan through the AET and swap any adjacent edges for which the
         second edge is at a lower current X coord than the first edge.
         Repeat until no further swapping is needed */
      if (AETPtr != NULL) {
         do {
            SwapOccurred = 0;
            CurrentEdgePtr = &AETPtr;
            while ((CurrentEdge = *CurrentEdgePtr)->NextEdge != NULL) {
               if (CurrentEdge->X > CurrentEdge->NextEdge->X) {
                  /* The second edge has a lower X than the first;
                     swap them in the AET */
                  TempEdge = CurrentEdge->NextEdge->NextEdge;
                  *CurrentEdgePtr = CurrentEdge->NextEdge;
                  CurrentEdge->NextEdge->NextEdge = CurrentEdge;
                  CurrentEdge->NextEdge = TempEdge;
                  SwapOccurred = 1;
               }
               CurrentEdgePtr = &(*CurrentEdgePtr)->NextEdge;
            }
         } while (SwapOccurred != 0);
      }
```

```
}

/* Advances each edge in the AET by one scan line.
   Removes edges that have been fully scanned. */
static void AdvanceAET() {
   struct EdgeState *CurrentEdge, **CurrentEdgePtr;

   /* Count down and remove or advance each edge in the AET */
   CurrentEdgePtr = &AETPtr;
   while ((CurrentEdge = *CurrentEdgePtr) != NULL) {
      /* Count off one scan line for this edge */
      if ((--(CurrentEdge->Count)) == 0) {
         /* This edge is finished, so remove it from the AET */
         *CurrentEdgePtr = CurrentEdge->NextEdge;
      } else {
         /* Advance the edge's X coordinate by minimum move */
         CurrentEdge->X += CurrentEdge->WholePixelXMove;
         /* Determine whether it's time for X to advance one extra */
         if ((CurrentEdge->ErrorTerm +=
               CurrentEdge->ErrorTermAdjUp) > 0) {
            CurrentEdge->X += CurrentEdge->XDirection;
            CurrentEdge->ErrorTerm -= CurrentEdge->ErrorTermAdjDown;
         }
         CurrentEdgePtr = &CurrentEdge->NextEdge;
      }
   }
}

/* Moves all edges that start at the specified Y coordinate from the
   GET to the AET, maintaining the X sorting of the AET. */
static void MoveXSortedToAET(int YToMove) {
   struct EdgeState *AETEdge, **AETEdgePtr, *TempEdge;
   int CurrentX;

   /* The GET is Y sorted. Any edges that start at the desired Y
      coordinate will be first in the GET, so we'll move edges from
      the GET to AET until the first edge left in the GET is no longer
      at the desired Y coordinate. Also, the GET is X sorted within
      each Y coordinate, so each successive edge we add to the AET is
      guaranteed to belong later in the AET than the one just added */
   AETEdgePtr = &AETPtr;
   while ((GETPtr != NULL) && (GETPtr->StartY == YToMove)) {
      CurrentX = GETPtr->X;
      /* Link the new edge into the AET so that the AET is still
         sorted by X coordinate */
      for (;;) {
         AETEdge = *AETEdgePtr;
         if ((AETEdge == NULL) || (AETEdge->X >= CurrentX)) {
            TempEdge = GETPtr->NextEdge;
            *AETEdgePtr = GETPtr;  /* link the edge into the AET */
            GETPtr->NextEdge = AETEdge;
            AETEdgePtr = &GETPtr->NextEdge;
            GETPtr = TempEdge;   /* unlink the edge from the GET */
            break;
         } else {
            AETEdgePtr = &AETEdge->NextEdge;
         }
      }
   }
}

/* Fills the scan line described by the current AET at the specified Y
   coordinate in the specified color, using the odd/even fill rule */
static void ScanOutAET(int YToScan, int Color) {
   int LeftX;
   struct EdgeState *CurrentEdge;

   /* Scan through the AET, drawing line segments as each pair of edge
      crossings is encountered. The nearest pixel on or to the right
      of left edges is drawn, and the nearest pixel to the left of but
      not on right edges is drawn */
   CurrentEdge = AETPtr;
   while (CurrentEdge != NULL) {
      LeftX = CurrentEdge->X;
```

```
      CurrentEdge = CurrentEdge->NextEdge;
      DrawHorizontalLineSeg(YToScan, LeftX, CurrentEdge->X-1, Color);
      CurrentEdge = CurrentEdge->NextEdge;
   }
}
```

[LISTING TWO]

```
/* Draws all pixels in the horizontal line segment passed in, from
   (LeftX,Y) to (RightX,Y), in the specified color in mode 13h, the
   VGA's 320x200 256-color mode. Both LeftX and RightX are drawn. No
   drawing will take place if LeftX > RightX. */

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

#define SCREEN_WIDTH    320
#define SCREEN_SEGMENT  0xA000

static void DrawPixel(int, int, int);

void DrawHorizontalLineSeg(Y, LeftX, RightX, Color) {
   int X;

   /* Draw each pixel in the horizontal line segment, starting with
      the leftmost one */
   for (X = LeftX; X <= RightX; X++)
      DrawPixel(X, Y, Color);
}

/* Draws the pixel at (X, Y) in color Color in VGA mode 13h */
static void DrawPixel(int X, int Y, int Color) {
   unsigned char far *ScreenPtr;

#ifdef __TURBOC__
   ScreenPtr = MK_FP(SCREEN_SEGMENT, Y * SCREEN_WIDTH + X);
#else    /* MSC 5.0 */
   FP_SEG(ScreenPtr) = SCREEN_SEGMENT;
   FP_OFF(ScreenPtr) = Y * SCREEN_WIDTH + X;
#endif
   *ScreenPtr = (unsigned char) Color;
}
```

[LISTING THREE]

```
/* POLYGON.H: Header file for polygon-filling code */

#define CONVEX    0
#define NONCONVEX 1
#define COMPLEX   2

/* Describes a single point (used for a single vertex) */
struct Point {
   int X;   /* X coordinate */
   int Y;   /* Y coordinate */
};
/* Describes a series of points (used to store a list of vertices that
   describe a polygon; each vertex connects to the two adjacent
   vertices; 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 the beginning and ending X coordinates of a single
   horizontal line (used only by fast polygon fill code) */
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 (used to describe a scan-converted polygon to the
   low-level hardware-dependent drawing code) (used only by fast
   polygon fill code) */
struct HLineList {
   int Length;               /* # of horizontal lines */
   int YStart;               /* Y coordinate of topmost line */
   struct HLine * HLinePtr;  /* pointer to list of horz lines */
};
```

[LISTING FOUR]

```
/* Sample program to exercise the polygon-filling routines */

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

#define DRAW_POLYGON(PointList,Color,Shape,X,Y)             \
   Polygon.Length = sizeof(PointList)/sizeof(struct Point); \
   Polygon.PointPtr = PointList;                            \
   FillPolygon(&Polygon, Color, Shape, X, Y);

void main(void);
extern int FillPolygon(struct PointListHeader *, int, int, int, int);

void main() {
   int i, j;
   struct PointListHeader Polygon;
   static struct Point Polygon1[] =
         {{0,0},{100,150},{320,0},{0,200},{220,50},{320,200}};
   static struct Point Polygon2[] =
         {{0,0},{320,0},{320,200},{0,200},{0,0},{50,50},
          {270,50},{270,150},{50,150},{50,50}};
   static struct Point Polygon3[] =
         {{0,0},{10,0},{105,185},{260,30},{15,150},{5,150},{5,140},
          {260,5},{300,5},{300,15},{110,200},{100,200},{0,10}};
   static struct Point Polygon4[] =
         {{0,0},{30,-20},{30,0},{0,20},{-30,0},{-30,-20}};
   static struct Point Triangle1[] = {{30,0},{15,20},{0,0}};
   static struct Point Triangle2[] = {{30,20},{15,0},{0,20}};
   static struct Point Triangle3[] = {{0,20},{20,10},{0,0}};
   static struct Point Triangle4[] = {{20,20},{20,0},{0,10}};
   union REGS regset;

   /* Set the display to VGA mode 13h, 320x200 256-color mode */
   regset.x.ax = 0x0013;
   int86(0x10, &regset, &regset);

   /* Draw three complex polygons */
   DRAW_POLYGON(Polygon1, 15, COMPLEX, 0, 0);
   getch();    /* wait for a keypress */
   DRAW_POLYGON(Polygon2, 5, COMPLEX, 0, 0);
   getch();    /* wait for a keypress */
   DRAW_POLYGON(Polygon3, 3, COMPLEX, 0, 0);
   getch();    /* wait for a keypress */

   /* Draw some adjacent nonconvex polygons */
   for (i=0; i<5; i++) {
      for (j=0; j<8; j++) {
         DRAW_POLYGON(Polygon4, 16+i*8+j, NONCONVEX, 40+(i*60),
               30+(j*20));
      }
   }
   getch();    /* wait for a keypress */

   /* Draw adjacent triangles across the screen */
   for (j=0; j<=80; j+=20) {
```

```
      for (i=0; i<290; i += 30) {
         DRAW_POLYGON(Triangle1, 2, CONVEX, i, j);
         DRAW_POLYGON(Triangle2, 4, CONVEX, i+15, j);
      }
   }
   for (j=100; j<=170; j+=20) {
      /* Do a row of pointing-right triangles */
      for (i=0; i<290; i += 20) {
         DRAW_POLYGON(Triangle3, 40, CONVEX, i, j);
      }
      /* Do a row of pointing-left triangles halfway between one row
         of pointing-right triangles and the next, to fit between */
      for (i=0; i<290; i += 20) {
         DRAW_POLYGON(Triangle4, 1, CONVEX, i, j+10);
      }
   }
   getch();    /* wait for a keypress */

   /* Return to text mode and exit */
   regset.x.ax = 0x0003;
   int86(0x10, &regset, &regset);
}
```

[LISTING FIVE]

```
; Draws all pixels in the horizontal line segment passed in, from
;  (LeftX,Y) to (RightX,Y), in the specified color in mode 13h, the
;  VGA's 320x200 256-color mode. No drawing will take place if
;  LeftX > RightX. Tested with TASM 2.0
; C near-callable as:
;       void DrawHorizontalLineSeg(Y, LeftX, RightX, Color);

SCREEN_WIDTH    equ     320
SCREEN_SEGMENT  equ     0a000h

Parms   struc
        dw      2 dup(?) ;return address & pushed BP
Y       dw      ?       ;Y coordinate of line segment to draw
LeftX   dw      ?       ;left endpoint of the line segment
RightX  dw      ?       ;right endpoint of the line segment
Color   dw      ?       ;color in which to draw the line segment
Parms   ends

        .model small
        .code
        public _DrawHorizontalLineSeg
        align   2
_DrawHorizontalLineSeg  proc
        push    bp              ;preserve caller's stack frame
        mov     bp,sp           ;point to our stack frame
        push    di              ;preserve caller's register variable
        cld                     ;make string instructions inc pointers
        mov     ax,SCREEN_SEGMENT
        mov     es,ax           ;point ES to display memory
        mov     di,[bp+LeftX]
        mov     cx,[bp+RightX]
        sub     cx,di           ;width of line
        jl      DrawDone        ;RightX < LeftX; no drawing to do
        inc     cx              ;include both endpoints
        mov     ax,SCREEN_WIDTH
        mul     [bp+Y]          ;offset of scan line on which to draw
        add     di,ax           ;ES:DI points to start of line seg
        mov     al,byte ptr [bp+Color] ;color in which to draw
        mov     ah,al           ;put color in AH for STOSW
        shr     cx,1            ;# of words to fill
        rep     stosw           ;fill a word at a time
        adc     cx,cx
        rep     stosb           ;draw the odd byte, if any
DrawDone:
```

```
        pop     di              ;restore caller's register variable
        pop     bp              ;restore caller's stack frame
        ret
_DrawHorizontalLineSeg  endp
        end
```

Journal: Dr. Dobb's Journal July 1991 v16 n7 p133(7)

Mode X: 256-color VGA magic

Author: Abrash, Michael.
AttFile: GP-JUL91.ASC Source code listing.

Summary: VGA's 320 x 240 256-color mode is most likely the single best mode of VGA, especially for animation. Features that make this mode so special include its 1:1 aspect ratio, which results in equal pixel spacing vertically and horizontally. Square pixels create the most attractive displays. In addition, mode X allows page flipping, which helps create smooth animation. Mode X pixels are processed in parallel, improving performance up to four times. However, the use of mode X is not widespread, since it is entirely undocumented. Only a very experienced VGA programmer would know that such a mode exists. The author provides mode set code, delineates the bitmap organization, and demonstrate how the basic write pixel and read pixel operations work.

There's a well-known Latin saying, in complexitate est opportunitas ("in complexity there is opportunity"), that must have been invented with the VGA in mind. Well, actually, it's not exactly well-known (I just thought of it this afternoon), but it should be. As evidence, witness the strange case of the VGA's 320 x 240 256-color mode, which is undeniably complex to program and isn't even documented by IBM -- but which is, nonetheless, perhaps the single best mode the VGA has to offer, especially for animation.

What Makes 320 x 240 Special?

Five features set the 320 x 240 256-color mode (which I'll call "mode X," befitting its mystery status in IBM's documentation) apart from other VGA modes. First, it has a 1:1 aspect ratio, resulting in equal pixel spacing horizontally and vertically (square pixels). Square pixels make for the most attractive displays, and avoid considerable programming effort that would otherwise be necessary to adjust graphics primitives and images to match the screen's pixel spacing. (For example, with square pixels, a circle can be drawn as a circle; otherwise, it must be drawn as an ellipse that corrects for the aspect ratio -- a slower, more complicated process.) In contrast, mode 13h, the only documented 256-color mode, provides a nonsquare 320 x 200 resolution.

Second, mode X allows page flipping, a prerequisite for the smoothest possible animation. Mode 13h does not allow page flipping, nor does mode 12h, the VGA's high-resolution 640 x 480 16-color mode.

Third, mode X allows the VGA's plane-oriented hardware to be used to process pixels in parallel, improving performance by up to four times over mode 13h.

Fourth, like mode 13h but unlike all other VGA modes, mode X is a byteper-pixel mode (each pixel is controlled by one byte in display memory), eliminating the slow read-before-write and bit-masking operations often required in 16-color modes. In addition to cutting the number of memory accesses in half, this is important because the memory caching schemes used by many VGA clones speed up writes more than reads.

Fifth, unlike mode 13h, mode X has plenty of offscreen memory free for image storage. This is particularly effective in conjunction with the use of the VGA's latches; together, the latches and the off-screen memory allow images to be copied to the screen four pixels at a time.

There's a sixth feature of mode X that's not so terrific: It's hard to program efficiently. If you've ever programmed a VGA 16-color mode directly, you know that VGA programming can be demanding; mode X is often as demanding as 16-color programming, and operates by a set of rules that turns everyting you've learned in 16-color mode sideways. Programming mode X is nothing like programming the nice, flat bitmap of mode 13h, or, for that matter, the flat, linear (albeit banked) bitmap used by 256-color SuperVGA modes. (I'd like to emphasize that mode X works on all VGAs, not just SuperVGAs.)

Many programmers I talk to love the flat bitmap model, and think that it's the ideal organization for display memory because it's so straightforward to program. Remember the saying I started this column with, though; the complexity of mode X truly is opportunity -- opportunity for the best combination of performance and appearance the VGA has to offer. If you do 256-color programming, especially if you use animation, you're missing the boat if you're not using mode X.

Although some developers have taken advantage of mode X, its use is certainly not widespread, being entirely undocumented; only an experienced VGA programmer would have the slightest inkling that it exists, and figuring out how to make it perform beyond the write pixel/read pixel level is no mean feat. I've never seen anything in print about it, and, in fact, the only articles I've seen about any of the undocumented 256-color modes were my own articles about the 320 x 200, 320 x 400, and 360 x 480 256-color modes in Programmer's Journal (January and September, 1989). (However, John Bridges has put code for a number of undocumented 256-color resolutions into the public domain, and I'd like to acknowledge the influence of his code on the mode set routine presented in this article.)

Given the tremendous advantages of 320 x 240 over the documented mode 13h, I'd very much like to get it into the hands of as many developers as possible, so I'm going to spend the next few columns exploring this odd but worthy mode. I'll provide mode set code, delineate the bitmap organization, and show how the basix write pixel and read pixel operations work. Then I'll move on to the magic stuff: rectangle fills, screen clears, scrolls, image copies, pixel inversion, and, yes, polygon fills (just a different driver), all blurry fast; hardware raster ops; and page flipping. In the end, I'll build a working animation program that showns many of the features of mode X in action.

The mode set code is the logical place to begin.

Selecting 320 x 240 256-Color Mode

We could, if we wished, write our own mode set code for mode X from scratch -- but why bother? Instead, we'll let the BIOS do most of the work by having it set up mode 13h, which we'll then turn into mode X by changing a few registers. Listing One (page 154) does exactly that.

After setting up mode 13h, Listing One alters the vertical counts and timings to select 480 visible scan lines. (There's no need to alter any horizontal values, because mode 13h and mode X both have 320-pixel horizontal resolutions.) The Maximum Scan Line register is programmed to double scan each line (that is, repeat each scan line twice), however, so we get an effective vertical resolution of 240 scan lines. It is, in fact, possible to get 400 or 480 independent scan lines in 256-color mode (see the aforementioned articles for details); however, 400-scan-line modes lack square pixels and can't support simultaneous offscreen memory and page flipping, and 480-scan-line modes lack page flipping altogether, due to memory constraints.

At the same time, Listing One programs the VGA's bitmap to a planar organization that is similar to that used by the 16-color modes, and utterly different from the linear bitmap of mode 13h. The bizarre bitmap organization of mode X is shown in Figure 1. The first pixel (the pixel at the upper left corner of the screen) is controlled by the byte at offset 0 in plane 0. (The one thing that mode X blessedly has in common with mode 13h is that each pixel is controlled by a single byte, eliminating the need to mask out individual bits of display memory.) The second pixel, immediately to the right of the first pixel, is controlled by the byte at offset 0 in plane 1. The third pixel comes from offset 0 in plane 2, and the fourth pixel from offset 0 in plane 3. Then the fifth pixel is controlled by the byte at offset 1 in plane 0, and that cycle continues, with each group of four pixels spread across the four planes at the same address. The offset M of pixel N in display memory is M = N/4, and the plane P of pixel N is P = N mod 4. For display memory writes, the plane is selected by setting bit P of the Map Mask register (Sequence Controller register 2) to 1 and all other bits to 0; for display memory reads, the plane is selected by setting the Read Map register (Graphics Controller register 4) to P.

It goes without saying that this one ugly bitmap organization, requiring a lot of overhead to manipulate a single pixel. The write pixel code shown in Listing Two (page 154) must determine the appropriate plane and perform a 16-bit OUT to select that plane for each pixel written, and likewise for the read pixel code shown in Listing Three (page 154). Calculating and mapping in a plane once for each pixel written is scarcely a recipe for performance.

That's all right, though, because most graphics software spends little time drawing individual pixels. I've provided the write and read pixel routines as basic primitives, and so you'll understand how the bitmap is organized, but the building blocks of high-performance graphics software are fills, copies, and bitblts, and it's here that mode X shines.

Designing From a Mode X Perspective

Listing Four (page 154) shows mode X rectangle fill code. The plane is selected for each pixel in turn, with drawing cycling from plane 0 to plane 3 then wrapping back to plane 0. This is the sort of code that stems from a write-pixel line of thinking; it reflects not a whit of the unique perspective that mode X demands, and although it looks reasonably efficient, it is in fact some of the slowest graphics code you will ever see. I've provided Listing Four partly for illustrative purposes, but mostly so we'll have a point of reference for the substantial speed-up that's possible with code that's designed from a mode X perspective.

The two major weaknesses of Listing Four both result from selecting the plane on a pixel by pixel basis. First, endless OUTs (which are particularly slow on 386s and 486s, often much slower than accesses to display memory) must be performed, and, second REP STOS can't be used. Listing Five (page 156) overcomes both these problems by tailoring the fill technique to the organization of display memory. Each plane is filled in its entirety in one burst before the next plane is processed, so only five OUTs are required in all, and REP STOS can indeed be used. (I've used REP STOSB in Listings Five and Six (page 156.) REP STOSW could be used and would improve performance on some 16-bit VGAs; however, REP STOSW requires extra overhead to set up, so it can be slower for small rectangles, especially on 8-bit VGAs. Doing an entire plane at a time can produce a "fading-in" effect for large images, because all columns for one plane are drawn before any columns for the next; if this is a problem, the four planes can be cycled through once for each scan line, rather than once for the entire rectangle.

Listing Five is 2.5 times faster than Listing Four at clearing the screen on a 20-MHz cached 386 with a Paradise VGA. Although Listing Five is slightly slower than an equivalent mode 13h fill routine would be, it's not grievously so. In general, performing plane-at-a-time operations can make almost any mode X operation, at the worst, nearly as fast as the same operation in mode 13h (although this sort of mode X programming is admittedly fairly complex). In this pursuit, it can help to organize data structures with mode X in mind. For example, icons could be prearranged in system memory with the pixels organized into four plane-oriented sets (or, again, in four sets per scan line to avoid a fading-in effect) to facilitate copying to the screen a plane at a time with REP MOVS.

Hardware Assist from an Unexpected Quarter

Listing Five illustrates the benefits of designing code from a mode X perspective; this is the software aspect of mode X optimization, which suffices to make mode X about as fast as mode 13h. That alone makes mode X an attractive mode, given its square pixels, page flipping, and offscreen memory, but superior performance would nonetheless be a pleasant addition to that list. Superior performance is indeed possible in mode X, although, oddly enough, it comes courtesy of the VGA's hardware, which was never designed to be used in 256-color modes.

All of the VGA's hardware assist features are available in mode X, although some are not particularly useful. The VGA hardware feature that's truly the key to mode X performance is the ability to process four planes' worth of data in parallel; this includes both the latches and the capability to fan data out to any or all planes. For rectangular fills, we'll just need to fan the data out to various planes, so I'll defer a discussion of other hardware

features until another column. (By the way, the ALUs, bit mask, and most other VGA hardware features are also available in mode 13h -- but parallel data processing is not.)

In planar modes, such as mode X, a byte written by the CPU to display memory may actually go to anywhere between zero and four planes, as shown in Figure 2. Each plane for which the setting of the corresponding bit in the Map Mask register is 1 receives the CPU data, and each plane for which the corresponding bit is 0 is not modified.

In 16-color modes, each plane contains one-quarter of each of eight pixel,s with the 4 bits of each pixel scanning all four planes. Not so in mode X. Look at Figure 1 again; each plane contains one pixel in its entirety, with four pixels at any given address, one per plane. Still, the Map Mask register does the same job in mode X as in 16-color modes; set it to 0Fh (all 1-bits), and all four planes will be written to by each CPU access. Thus, it would seem that up to four pixels could be set by a single mode X byte-sized write to display memory, potentially speeding up operations like rectangle fills by four times.

And, as it turns out, four-plane parallelism works quite nicely indeed. Listing Six is yet another rectangle-fill routine, this time using the Map Mask to set up to four pixels per STOS. The only trick to Listing Six is that any left or right edge that isn't aligned to a multiple-of-four pixel column (that is, a column at which one four-pixel set ends and the next begins) must be clipped via the Map Mask register, because not all pixels at the address containing the edge are modified. Performance is as expected; Listing Siz is nearly ten times faster at clearing the screen than Listing Four and just about four times faster than Listing Five -- and also about four times faster than the same rectangle fill in mode 13h. Understanding the bitmap organization and display hardware of mode X does indeed pay.

Just so you can see mode X in action, Listing Seven (page 158) is a sample program that selects mode X and draws a number of rectangles. Listing Seven links to any of the rectangle fill routines I've presented.

And now, I hope, you begin to see why I'm so fond of mode X. Next month, we'll continue with mode X by exploring the wonders that the latches and parallel plane hardware can work on scrolls, copies, blits, and pattern fills.

Notes From the Edsun Front

Comments coming my way indicate a great deal of programmer interest in the Edsun CEG/DAC, of which I wrote in April and May. However, everyone who has actually programmed the CEG/DAC complains about how hard it is; the results are nice, but the process of getting there is anything but. Nonetheless, programming the CEG/DAC is certainly a solvable problem, and whoever solves it best will come out looking mighty good. A fair analogy is writing active TSRs. Six years ago, TSR-writing was black magic, and Sidekick, primitive by today's standards, made a fortune. Today, any dope can choose from dozens of books and toolkits and make a rock-solid TSR in a few hours. As programmers develop better tools and a better understanding of the CEG/DAC, the grumbling will subside, and the software will take off. Another case of complexity providing opportunity.

Book of the Month

This month's book is Advanced Programmer's Guide to SuperVGAs, by Sutty and Blair (Brady, 1990, ISBN 0-13010455-8; $44.95). Pricey for softcover, but included in that price is a diskette of SuperVGA assembly code (which I have not tried out). This book is the single best guide I've seen to the Byzantine world of SuperVGA programming, where every one of dozens of VGA models has different mode numbers and banking schemes. Take it from someone who's waded through a slew of chip databooks and applications notes -- this book will save you a lot of time and aggravation if you have to program SuperVGAs directly.

Still, not everything I'd like to see is in there. For example, they cover only the Tseng Labs ET3000 chip, not the now widely used ET4000 that supports

15-bpp graphics. That's not the authors' fault, of course; it's a reflection
of the incredible diversity and rate of change in the SuperVGA arena.

Mode X. The Edsun CEG/DAC. SuperVGA programming. In complexitate est
opportunitas. Q.E.D.
- -

GRAPHICS PROGRAMMING COLUMN
by Michael Abrash

[LISTING ONE]

```
; Mode X (320x240, 256 colors) mode set routine. Works on all VGAs.
; C near-callable as:
;       void Set320x240Mode(void);
; Tested with TASM 2.0.
; Modified from public-domain mode set code by John Bridges.

SC_INDEX equ    03c4h   ;Sequence Controller Index
CRTC_INDEX equ  03d4h   ;CRT Controller Index
MISC_OUTPUT equ 03c2h   ;Miscellaneous Output register
SCREEN_SEG equ  0a000h  ;segment of display memory in mode X

        .model  small
        .data
; Index/data pairs for CRT Controller registers that differ between
; mode 13h and mode X.
CRTParms label  word
        dw      00d06h  ;vertical total
        dw      03e07h  ;overflow (bit 8 of vertical counts)
        dw      04109h  ;cell height (2 to double-scan)
        dw      0ea10h  ;v sync start
        dw      0ac11h  ;v sync end and protect cr0-cr7
        dw      0df12h  ;vertical displayed
        dw      00014h  ;turn off dword mode
        dw      0e715h  ;v blank start
        dw      00616h  ;v blank end
        dw      0e317h  ;turn on byte mode
CRT_PARM_LENGTH equ     (($-CRTParms)/2)

        .code
        public  _Set320x240Mode
_Set320x240Mode proc    near
        push    bp      ;preserve caller's stack frame
        push    si      ;preserve C register vars
        push    di      ; (don't count on BIOS preserving anything)

        mov     ax,13h  ;let the BIOS set standard 256-color
        int     10h     ; mode (320x200 linear)

        mov     dx,SC_INDEX
        mov     ax,0604h
        out     dx,ax   ;disable chain4 mode
        mov     ax,0100h
        out     dx,ax   ;synchronous reset while switching clocks

        mov     dx,MISC_OUTPUT
        mov     al,0e7h
        out     dx,al   ;select 28 MHz dot clock & 60 Hz scanning rate

        mov     dx,SC_INDEX
        mov     ax,0300h
        out     dx,ax   ;undo reset (restart sequencer)

        mov     dx,CRTC_INDEX ;reprogram the CRT Controller
        mov     al,11h  ;VSync End reg contains register write
        out     dx,al   ; protect bit
        inc     dx      ;CRT Controller Data register
        in      al,dx   ;get current VSync End register setting
        and     al,7fh  ;remove write protect on various
        out     dx,al   ; CRTC registers
```

```
        dec     dx      ;CRT Controller Index
        cld
        mov     si,offset CRTParms ;point to CRT parameter table
        mov     cx,CRT_PARM_LENGTH ;# of table entries
SetCRTParmsLoop:
        lodsw           ;get the next CRT Index/Data pair
        out     dx,ax   ;set the next CRT Index/Data pair
        loop    SetCRTParmsLoop

        mov     dx,SC_INDEX
        mov     ax,0f02h
        out     dx,ax   ;enable writes to all four planes
        mov     ax,SCREEN_SEG ;now clear all display memory, 8 pixels
        mov     es,ax         ; at a time
        sub     di,di   ;point ES:DI to display memory
        sub     ax,ax   ;clear to zero-value pixels
        mov     cx,8000h ;# of words in display memory
        rep     stosw   ;clear all of display memory

        pop     di      ;restore C register vars
        pop     si
        pop     bp      ;restore caller's stack frame
        ret
_Set320x240Mode endp
        end
```

[LISTING TWO]

```
; Mode X (320x240, 256 colors) write pixel routine. Works on all VGAs.
; No clipping is performed.
; C near-callable as:
;    void WritePixelX(int X, int Y, unsigned int PageBase, int Color);

SC_INDEX equ    03c4h   ;Sequence Controller Index
MAP_MASK equ    02h     ;index in SC of Map Mask register
SCREEN_SEG equ  0a000h  ;segment of display memory in mode X
SCREEN_WIDTH equ 80     ;width of screen in bytes from one scan line
                        ; to the next

parms   struc
        dw      2 dup (?) ;pushed BP and return address
X       dw      ?       ;X coordinate of pixel to draw
Y       dw      ?       ;Y coordinate of pixel to draw
PageBase dw     ?       ;base offset in display memory of page in
                        ; which to draw pixel
Color   dw      ?       ;color in which to draw pixel
parms   ends

        .model  small
        .code
        public  _WritePixelX
_WritePixelX    proc    near
        push    bp      ;preserve caller's stack frame
        mov     bp,sp   ;point to local stack frame

        mov     ax,SCREEN_WIDTH
        mul     [bp+Y]  ;offset of pixel's scan line in page
        mov     bx,[bp+X]
        shr     bx,1
        shr     bx,1    ;X/4 = offset of pixel in scan line
        add     bx,ax   ;offset of pixel in page
        add     bx,[bp+PageBase] ;offset of pixel in display memory
        mov     ax,SCREEN_SEG
        mov     es,ax   ;point ES:BX to the pixel's address

        mov     cl,byte ptr [bp+X]
        and     cl,011b ;CL = pixel's plane
        mov     ax,0100h + MAP_MASK ;AL = index in SC of Map Mask reg
        shl     ah,cl   ;set only the bit for the pixel's plane to 1
        mov     dx,SC_INDEX ;set the Map Mask to enable only the
```

```
        out     dx,ax       ; pixel's plane

        mov     al,byte ptr [bp+Color]
        mov     es:[bx],al  ;draw the pixel in the desired color

        pop     bp          ;restore caller's stack frame
        ret
_WritePixelX    endp
        end
```

[LISTING THREE]

```
; Mode X (320x240, 256 colors) read pixel routine. Works on all VGAs.
; No clipping is performed.
; C near-callable as:
;    unsigned int ReadPixelX(int X, int Y, unsigned int PageBase);

GC_INDEX equ    03ceh   ;Graphics Controller Index
READ_MAP equ    04h     ;index in GC of the Read Map register
SCREEN_SEG equ  0a000h  ;segment of display memory in mode X
SCREEN_WIDTH equ 80     ;width of screen in bytes from one scan line
                        ; to the next
parms   struc
        dw      2 dup (?) ;pushed BP and return address
X       dw      ?       ;X coordinate of pixel to read
Y       dw      ?       ;Y coordinate of pixel to read
PageBase dw     ?       ;base offset in display memory of page from
                        ; which to read pixel
parms   ends

        .model  small
        .code
        public  _ReadPixelX
_ReadPixelX     proc    near
        push    bp      ;preserve caller's stack frame
        mov     bp,sp   ;point to local stack frame

        mov     ax,SCREEN_WIDTH
        mul     [bp+Y]  ;offset of pixel's scan line in page
        mov     bx,[bp+X]
        shr     bx,1
        shr     bx,1    ;X/4 = offset of pixel in scan line
        add     bx,ax   ;offset of pixel in page
        add     bx,[bp+PageBase] ;offset of pixel in display memory
        mov     ax,SCREEN_SEG
        mov     es,ax   ;point ES:BX to the pixel's address

        mov     ah,byte ptr [bp+X]
        and     ah,011b ;AH = pixel's plane
        mov     al,READ_MAP ;AL = index in GC of the Read Map reg
        mov     dx,GC_INDEX ;set the Read Map to read the pixel's
        out     dx,ax       ; plane

        mov     al,es:[bx] ;read the pixel's color
        sub     ah,ah   ;convert it to an unsigned int

        pop     bp      ;restore caller's stack frame
        ret
_ReadPixelX     endp
        end
```

[LISTING FOUR]

```
; Mode X (320x240, 256 colors) rectangle fill routine. Works on all
; VGAs. Uses slow approach that selects the plane explicitly for each
```

```
; pixel. Fills up to but not including the column at EndX and the row
; at EndY. No clipping is performed.
; C near-callable as:
;    void FillRectangleX(int StartX, int StartY, int EndX, int EndY,
;       unsigned int PageBase, int Color);

SC_INDEX equ    03c4h   ;Sequence Controller Index
MAP_MASK equ    02h     ;index in SC of Map Mask register
SCREEN_SEG equ  0a000h  ;segment of display memory in mode X
SCREEN_WIDTH equ 80     ;width of screen in bytes from one scan line
                        ; to the next
parms   struc
        dw      2 dup (?) ;pushed BP and return address
StartX  dw      ?       ;X coordinate of upper left corner of rect
StartY  dw      ?       ;Y coordinate of upper left corner of rect
EndX    dw      ?       ;X coordinate of lower right corner of rect
                        ; (the row at EndX is not filled)
EndY    dw      ?       ;Y coordinate of lower right corner of rect
                        ; (the column at EndY is not filled)
PageBase dw     ?       ;base offset in display memory of page in
                        ; which to fill rectangle
Color   dw      ?       ;color in which to draw pixel
parms   ends

        .model  small
        .code
        public  _FillRectangleX
_FillRectangleX proc    near
        push    bp      ;preserve caller's stack frame
        mov     bp,sp   ;point to local stack frame
        push    si      ;preserve caller's register variables
        push    di

        mov     ax,SCREEN_WIDTH
        mul     [bp+StartY] ;offset in page of top rectangle scan line
        mov     di,[bp+StartX]
        shr     di,1
        shr     di,1    ;X/4 = offset of first rectangle pixel in scan
                        ; line
        add     di,ax   ;offset of first rectangle pixel in page
        add     di,[bp+PageBase] ;offset of first rectangle pixel in
                        ; display memory
        mov     ax,SCREEN_SEG
        mov     es,ax   ;point ES:DI to the first rectangle pixel's
                        ; address
        mov     dx,SC_INDEX ;set the Sequence Controller Index to
        mov     al,MAP_MASK ; point to the Map Mask register
        out     dx,al
        inc     dx      ;point DX to the SC Data register
        mov     cl,byte ptr [bp+StartX]
        and     cl,011b ;CL = first rectangle pixel's plane
        mov     al,01h
        shl     al,cl   ;set only the bit for the pixel's plane to 1
        mov     ah,byte ptr [bp+Color] ;color with which to fill
        mov     bx,[bp+EndY]
        sub     bx,[bp+StartY]  ;BX = height of rectangle
        jle     FillDone        ;skip if 0 or negative height
        mov     si,[bp+EndX]
        sub     si,[bp+StartX]  ;CX = width of rectangle
        jle     FillDone        ;skip if 0 or negative width
FillRowsLoop:
        push    ax      ;remember the plane mask for the left edge
        push    di      ;remember the start offset of the scan line
        mov     cx,si   ;set count of pixels in this scan line
FillScanLineLoop:
        out     dx,al   ;set the plane for this pixel
        mov     es:[di],ah ;draw the pixel
        shl     al,1    ;adjust the plane mask for the next pixel's
        and     al,01111b ; bit, modulo 4
        jnz     AddressSet ;advance address if we turned over from
        inc     di      ; plane 3 to plane 0
        mov     al,00001b ;set plane mask bit for plane 0
AddressSet:
        loop    FillScanLineLoop
```

```
        pop     di      ;retrieve the start offset of the scan line
        add     di,SCREEN_WIDTH ;point to the start of the next scan
                        ; line of the rectangle
        pop     ax      ;retrieve the plane mask for the left edge
        dec     bx      ;count down scan lines
        jnz     FillRowsLoop
FillDone:
        pop     di      ;restore caller's register variables
        pop     si
        pop     bp      ;restore caller's stack frame
        ret
_FillRectangleX endp
        end
```

[LISTING FIVE]

```
; Mode X (320x240, 256 colors) rectangle fill routine. Works on all
; VGAs. Uses medium-speed approach that selects each plane only once
; per rectangle; this results in a fade-in effect for large
; rectangles. Fills up to but not including the column at EndX and the
; row at EndY. No clipping is performed.
; C near-callable as:
;    void FillRectangleX(int StartX, int StartY, int EndX, int EndY,
;       unsigned int PageBase, int Color);

SC_INDEX equ    03c4h   ;Sequence Controller Index
MAP_MASK equ    02h     ;index in SC of Map Mask register
SCREEN_SEG equ  0a000h  ;segment of display memory in mode X
SCREEN_WIDTH equ 80     ;width of screen in bytes from one scan line
                        ; to the next
parms struc
        dw      2 dup (?) ;pushed BP and return address
StartX  dw      ?       ;X coordinate of upper left corner of rect
StartY  dw      ?       ;Y coordinate of upper left corner of rect
EndX    dw      ?       ;X coordinate of lower right corner of rect
                        ; (the row at EndX is not filled)
EndY    dw      ?       ;Y coordinate of lower right corner of rect
                        ; (the column at EndY is not filled)
PageBase dw     ?       ;base offset in display memory of page in
                        ; which to fill rectangle
Color   dw      ?       ;color in which to draw pixel
parms ends

StartOffset equ  -2     ;local storage for start offset of rectangle
Width    equ     -4     ;local storage for address width of rectangle
Height   equ     -6     ;local storage for height of rectangle
PlaneInfo equ    -8     ;local storage for plane # and plane mask
STACK_FRAME_SIZE equ 8

        .model  small
        .code
        public  _FillRectangleX
_FillRectangleX proc    near
        push    bp      ;preserve caller's stack frame
        mov     bp,sp   ;point to local stack frame
        sub     sp,STACK_FRAME_SIZE ;allocate space for local vars
        push    si      ;preserve caller's register variables
        push    di

        cld
        mov     ax,SCREEN_WIDTH
        mul     [bp+StartY] ;offset in page of top rectangle scan line
        mov     di,[bp+StartX]
        shr     di,1
        shr     di,1    ;X/4 = offset of first rectangle pixel in scan
                        ; line
        add     di,ax   ;offset of first rectangle pixel in page
        add     di,[bp+PageBase] ;offset of first rectangle pixel in
                        ; display memory
```

```
        mov     ax,SCREEN_SEG
        mov     es,ax   ;point ES:DI to the first rectangle pixel's
        mov     [bp+StartOffset],di ; address
        mov     dx,SC_INDEX ;set the Sequence Controller Index to
        mov     al,MAP_MASK ; point to the Map Mask register
        out     dx,al
        mov     bx,[bp+EndY]
        sub     bx,[bp+StartY]  ;BX = height of rectangle
        jle     FillDone        ;skip if 0 or negative height
        mov     [bp+Height],bx
        mov     dx,[bp+EndX]
        mov     cx,[bp+StartX]
        cmp     dx,cx
        jle     FillDone        ;skip if 0 or negative width
        dec     dx
        and     cx,not 011b
        sub     dx,cx
        shr     dx,1
        shr     dx,1
        inc     dx      ;# of addresses across rectangle to fill
        mov     [bp+Width],dx
        mov     word ptr [bp+PlaneInfo],0001h
                        ;lower byte = plane mask for plane 0,
                        ; upper byte = plane # for plane 0
FillPlanesLoop:
        mov     ax,word ptr [bp+PlaneInfo]
        mov     dx,SC_INDEX+1 ;point DX to the SC Data register
        out     dx,al   ;set the plane for this pixel
        mov     di,[bp+StartOffset] ;point ES:DI to rectangle start
        mov     dx,[bp+Width]
        mov     cl,byte ptr [bp+StartX]
        and     cl,011b ;plane # of first pixel in initial byte
        cmp     ah,cl   ;do we draw this plane in the initial byte?
        jae     InitAddrSet ;yes
        dec     dx      ;no, so skip the initial byte
        jz      FillLoopBottom ;skip this plane if no pixels in it
        inc     di
InitAddrSet:
        mov     cl,byte ptr [bp+EndX]
        dec     cl
        and     cl,011b ;plane # of last pixel in final byte
        cmp     ah,cl   ;do we draw this plane in the final byte?
        jbe     WidthSet ;yes
        dec     dx      ;no, so skip the final byte
        jz      FillLoopBottom ;skip this planes if no pixels in it
WidthSet:
        mov     si,SCREEN_WIDTH
        sub     si,dx   ;distance from end of one scan line to start
                        ; of next
        mov     bx,[bp+Height] ;# of lines to fill
        mov     al,byte ptr [bp+Color] ;color with which to fill
FillRowsLoop:
        mov     cx,dx   ;# of bytes across scan line
        rep     stosb   ;fill the scan line in this plane
        add     di,si   ;point to the start of the next scan
                        ; line of the rectangle
        dec     bx      ;count down scan lines
        jnz     FillRowsLoop
FillLoopBottom:
        mov     ax,word ptr [bp+PlaneInfo]
        shl     al,1    ;set the plane bit to the next plane
        inc     ah      ;increment the plane #
        mov     word ptr [bp+PlaneInfo],ax
        cmp     ah,4    ;have we done all planes?
        jnz     FillPlanesLoop ;continue if any more planes
FillDone:
        pop     di      ;restore caller's register variables
        pop     si
        mov     sp,bp   ;discard storage for local variables
        pop     bp      ;restore caller's stack frame
        ret
_FillRectangleX endp
        end
```

[LISTING SIX]

```
; Mode X (320x240, 256 colors) rectangle fill routine. Works on all
; VGAs. Uses fast approach that fans data out to up to four planes at
; once to draw up to four pixels at once. Fills up to but not
; including the column at EndX and the row at EndY. No clipping is
; performed.
; C near-callable as:
;    void FillRectangleX(int StartX, int StartY, int EndX, int EndY,
;       unsigned int PageBase, int Color);

SC_INDEX equ    03c4h   ;Sequence Controller Index
MAP_MASK equ    02h     ;index in SC of Map Mask register
SCREEN_SEG equ  0a000h  ;segment of display memory in mode X
SCREEN_WIDTH equ 80     ;width of screen in bytes from one scan line
                        ; to the next
parms   struc
        dw      2 dup (?) ;pushed BP and return address
StartX  dw      ?       ;X coordinate of upper left corner of rect
StartY  dw      ?       ;Y coordinate of upper left corner of rect
EndX    dw      ?       ;X coordinate of lower right corner of rect
                        ; (the row at EndX is not filled)
EndY    dw      ?       ;Y coordinate of lower right corner of rect
                        ; (the column at EndY is not filled)
PageBase dw     ?       ;base offset in display memory of page in
                        ; which to fill rectangle
Color   dw      ?       ;color in which to draw pixel
parms   ends

        .model  small
        .data
; Plane masks for clipping left and right edges of rectangle.
LeftClipPlaneMask       db      00fh,00eh,00ch,008h
RightClipPlaneMask      db      00fh,001h,003h,007h
        .code
        public  _FillRectangleX
_FillRectangleX proc    near
        push    bp      ;preserve caller's stack frame
        mov     bp,sp   ;point to local stack frame
        push    si      ;preserve caller's register variables
        push    di

        cld
        mov     ax,SCREEN_WIDTH
        mul     [bp+StartY] ;offset in page of top rectangle scan line
        mov     di,[bp+StartX]
        shr     di,1    ;X/4 = offset of first rectangle pixel in scan
        shr     di,1    ; line
        add     di,ax   ;offset of first rectangle pixel in page
        add     di,[bp+PageBase] ;offset of first rectangle pixel in
                        ; display memory
        mov     ax,SCREEN_SEG   ;point ES:DI to the first rectangle
        mov     es,ax           ; pixel's address
        mov     dx,SC_INDEX ;set the Sequence Controller Index to
        mov     al,MAP_MASK ; point to the Map Mask register
        out     dx,al
        inc     dx      ;point DX to the SC Data register
        mov     si,[bp+StartX]
        and     si,0003h                ;look up left edge plane mask
        mov     bh,LeftClipPlaneMask[si] ; to clip & put in BH
        mov     si,[bp+EndX]
        and     si,0003h                 ;look up right edge plane
        mov     bl,RightClipPlaneMask[si] ; mask to clip & put in BL

        mov     cx,[bp+EndX]    ;calculate # of addresses across rect
        mov     si,[bp+StartX]
        cmp     cx,si
        jle     FillDone        ;skip if 0 or negative width
        dec     cx
        and     si,not 011b
        sub     cx,si
        shr     cx,1
        shr     cx,1    ;# of addresses across rectangle to fill - 1
```

```
        jnz     MasksSet ;there's more than one byte to draw
        and     bh,bl   ;there's only one byte, so combine the left
                        ; and right edge clip masks
MasksSet:
        mov     si,[bp+EndY]
        sub     si,[bp+StartY]  ;BX = height of rectangle
        jle     FillDone        ;skip if 0 or negative height
        mov     ah,byte ptr [bp+Color] ;color with which to fill
        mov     bp,SCREEN_WIDTH ;stack frame isn't needed any more
        sub     bp,cx   ;distance from end of one scan line to start
        dec     bp      ; of next
FillRowsLoop:
        push    cx      ;remember width in addresses - 1
        mov     al,bh   ;put left-edge clip mask in AL
        out     dx,al   ;set the left-edge plane (clip) mask
        mov     al,ah   ;put color in AL
        stosb           ;draw the left edge
        dec     cx      ;count off left edge byte
        js      FillLoopBottom ;that's the only byte
        jz      DoRightEdge ;there are only two bytes
        mov     al,00fh ;middle addresses are drawn 4 pixels at a pop
        out     dx,al   ;set the middle pixel mask to no clip
        mov     al,ah   ;put color in AL
        rep     stosb   ;draw the middle addresses four pixels apiece
DoRightEdge:
        mov     al,bl   ;put right-edge clip mask in AL
        out     dx,al   ;set the right-edge plane (clip) mask
        mov     al,ah   ;put color in AL
        stosb           ;draw the right edge
FillLoopBottom:
        add     di,bp   ;point to the start of the next scan line of
                        ; the rectangle
        pop     cx      ;retrieve width in addresses - 1
        dec     si      ;count down scan lines
        jnz     FillRowsLoop
FillDone:
        pop     di      ;restore caller's register variables
        pop     si
        pop     bp      ;restore caller's stack frame
        ret
_FillRectangleX endp
        end
```

[LISTING SEVEN]

```
/* Program to demonstrate mode X (320x240, 256-colors) rectangle
   fill by drawing adjacent 20x20 rectangles in successive colors from
   0 on up across and down the screen */
#include <conio.h>
#include <dos.h>

void Set320x240Mode(void);
void FillRectangleX(int, int, int, int, unsigned int, int);

void main() {
   int i,j;
   union REGS regset;

   Set320x240Mode();
   FillRectangleX(0,0,320,240,0,0); /* clear the screen to black */
   for (j = 1; j < 220; j += 21) {
      for (i = 1; i < 300; i += 21) {
         FillRectangleX(i, j, i+20, j+20, 0, ((j/21*15)+i/21) & 0xFF);
      }
   }
   getch();
   regset.x.ax = 0x0003;   /* switch back to text mode and done */
   int86(0x10, &regset, &regset);
}
```

Journal: Dr. Dobb's Journal August 1991 v16 n8 p165(7)

More undocumented 256-color VGA magic

Author: Abrash, Michael.
AttFile: GP-AUG91.ASC Source code listing.

Summary: Programmers should remember that there are many subtle approaches to any problem and to keep the big picture in mind when implementing programs. Mode X is an undocumented 320 x 320 256-color mode of the VGA standard that supports page flipping, makes available off-screen memory, has square pixels, and permits users to increase performance by as much as four times by using the VGA's hardware. There are four latches in VGA, one for each plane of display memory, and these latches are used to copy data from one part of display memory to another. Latches are suitable for patterned fills and screen-to-screen copies, including scrolls. Four-pixel-wide patterns are extremely useful.

Every so often, a programming demon that I'd thought I'd forever laid to rest arises to haunt me once again. A minor example of this -- an imp, if you will -- is the use of "=" when I mean "==," which I've done all too often in the past, and am sure I'll do again. That's minor deviltry, though, compared to the considerably greater evils of one of my personal scourges, of which I was recently reminded anew: too-close attention to detail. Not seeing the forest for the trees. Looking low when I should have looked high. Missing the big picture, if you catch my drift.

Thoreau said it best: "Our life is frittered away by detail. . . . Simplify, simplify." That quote sprang to mind when I received a letter from Anton Treuenfels of Fridley, Minnesota, thanking me for clarifying the principles of filling adjacent convex polygons, as discussed in this column in February and March. Anton then went on the describe his own method for filling convex polygons.

Anton's approach had its virtues and drawbacks, foremost among the virtues being a simplicity Thoreau would have admired. For instance, in writing my polygon-filling code, I had spent quite some time trying to figure out the best way to identify which edge was the left edge and which the right, finally settling on comparing the slopes of the edges if the top of the polygon wasn't flat, and comparing the starting points of the edges if the top was flat. Anton simplified this tremendously by not bothering to figure out ahead of time which was the right edge of the polygon and which the left, instead scanning out the two edges in whatever order he found them and letting the low-level drawing code test, and if necessary swap, the end-points of each horizontal line of the fill, so that filling started at the leftmost edge. This is a little slower than my approach (although the difference is almost surely negligible), but it also makes quite a bit of code go away.

What that example, and others like it in Anton's letter, did was kick my mind into a mode that it hadn't -- but should have -- been in when I wrote the code, a mode in which I began to wonder, "How else can I simplify this code?"; what you might call Occam's Razor mode. You see, I created the convex polygon-drawing code by first writing pseudocode, then writing C code, and finally writing assembly code, and once the pseudocode was finished, I stopped thinking about the interactions of the various portions of the program. In other words, I became so absorbed in individual details that I forgot to consider the code as a whole. That was a mistake, and an embarrassing one for someone who constantly preaches that programmers should look at their code from a variety of perspectives. May my embarrassment be your enlightenment.

The point is not whether, in the final analysis, my code or Anton's code is better; both have their advantages. The point is that I was programming with half a deck because I was so fixated on the details of a single sort of implementation; I ended up with relatively hard-to-write, complex code, and missed out on many potentially useful optimizations by being so focused. It's a big world out there, and there are many subtle approaches to any problem, so relax and keep the big picture in mind as you implement your programs. Your code will likely be not only better, but also simpler. And

whenever you see me walking across hot coals in this column when there's an easier way to go, please, let me know!

Thanks, Anton.

Mode X Continued

Last month, I introduced you to what I call mode X, an undocumented 320 X 240 256-color mode of the VGA. Mode X is distinguished from mode 13h, the documented 320 X 200 256-color VGA mode, in that it supports page flipping, makes off-screen memory available, has square pixels, and, above all, lets you use the VGA's hardware to increase performance by as much as four times (at the cost of more complex and demanding programming, to be sure -- but end users care about results, not how hard the code was to write, and mode X delivers results in a big way). Last month we saw how the VGA's plane-oriented hardware can be used to speed solid fills. That's a nice technique, but this month we're going to move up to the big guns -- the latches.

The VGA has four latches, one for each plane of display memory. Each latch stores exactly one byte, and that byte is always the last byte read from the corresponding plane of display memory, as shown in Figure 1. Furthermore, whenever a given address in display memory is read, all four planes' bytes at that address are read and stored in the corresponding latches, regardless of which plane supplied the byte returned to the CPU (as determined by the Read Map register). As with so much else about the VGA, the above will make little sense to VGA neophytes, but the important point is this: By reading one display memory byte, 4 bytes --one from each plane -- can be loaded into the latches at once. Any or all of those 4 bytes can then be written anywhere in display memory with a single byte-sized write, as shown in Figure 2. The upshot is that the latches make it possible to copy data around from one part of display memory to another, 32 bits (four pixels) at a time -- four times as fast as normal. (Recall from last month that in mode X, pixels are stored one per byte, with four pixels in a row stored in successive planes at the same address, one pixel per plane.) However, any one latch can only be loaded from and written to the corresponding plane, so an individual latch can only work with every fourth pixel on the screen; the latch for plane 0 can work with pixels 0, 4, 8. . ., the latch for plane 1 with pixels 1, 5, 9. . ., and so on.

The latches aren't intended for use in 256-color mode -- they were designed to allow individual bits of display memory to be modified in 16-color mode -- but they are nonetheless very useful in mode X, particularly for patterned fills and screen-to-screen copies, including scrolls. Patterned filling is a good place to start, because patterns are widely used in windowing environments for desktops, window backgrounds, and scroll bars, and for textures and color dithering in drawing and game software.

Fast mode X fills with patterns that are four pixels in width can be performed by drawing the pattern once to the four pixels at any one address in display memory, reading that address to load the pattern into the latches, setting the Bit Mask register to 0 to specify that all bits drawn to display memory should come from the latches, and then performing the fill pretty much as we did last month, except that each line of the pattern must be loaded into the latches before the corresponding scan line on the screen is filled. Listings One and Two (page 181) together demonstrate a variety of fast mode X four-by-four pattern fills. (The mode set function called by Listing One is from last month's column.)

Four-pixel-wide patterns are more useful than you might imagine. There are actually [2.sup.128] possible patterns (16 pixels, each with [2.sup.8] possible colors); that set is certainly large enough for most color-dithering purposes, and includes many often-used patterns, such as halftones, diagonal stripes, and crosshatches.

Furthermore, eight-wide patterns, which are widely used, can be drawn with two passes, one for each half of the pattern; this principle can in fact be extended to patterns of arbitrary multiple-of-four widths. (Widths that aren't multiples of four are considerably more difficult to handle, because the latches are four pixels wide.)

Allocating Memory in Mode X

Listing Two raises some interesting questions about the allocation of display memory in mode X. In Listing Two, whenever a pattern is to be drawn, that pattern is first drawn in its entirety at the very end of display memory; the latches are then loaded from that copy of the pattern before each scan line of the actual fill is drawn. Why this double copying process, and why is the pattern stored in that particular area of display memory?

The double copying process is used because it's the easiest way to load the latches. Remember, there's no way to get information directly from the CPU to the ltches; the information must first be written to some location in display memory, because the latches can be loaded only from display memory. By writing the pattern to off-screen memory, we don't have to worry about interfering with whatever is currently displayed on the screen.

As for why the pattern is stored exactly where it is, that's part of a master memory allocation plan that will come to fruition next month when I implement a mode X animation program. Figure 3 shows this master plan; the first two pages of memory (each 76,800 pixels long, spanning 19,200 addresses -- that is, 19,200 pixel quadruplets -- in display memory) are reserved for page flipping, the next page of memory (also 76,800 pixels long) is reserved for storing the background (this is used to restore the holes left after images move), the last 16 pixels (four addresses) of display memory are reserved for the pattern buffer, and the remaining 31,728 pixels (7932 addresses) of display memory are free for storage of icons, images, temporary buffers, or whatever. This is an efficient organization for animation, but there are certainly many other possible setups. For example, you might choose to have a solidly-colored background, in which case you could dispense with the background page (instead using the solid rectangle fill routine to replace the background after images move), freeing up another 76,800 pixels of off-screen storage for images and buffers. You could even eliminate page-flipping altogether if you needed to free up a great deal of display memory. For example, with enough free display memory it is possible in mode X to create a virtual bitmap three times larger than the screen, with the screen becoming a scrolling window onto that larger bitmap. This technique has been used to good effect in a number of games, although I don't know if any of those games use mode X.

Copying Pixel Blocks Within Display Memory

Another fine use for the latches is copying pixels from one place in display memory to another. Whenever both the source and the destination share the same nibble alignment (that is, their start addresses modulo four are the same), it is not only possible but quite easy to use the latches to perform the copy four pixels at a time. Listing Three (page 182) shows a routine that copies via the latches. (When the source and destination do not share the same nibble alignment, the latches cannot be used, because the source and destination planes for any given pixel differ; in that case, you can set the Read Map register to select a source plane and the Map Mask register to select the corresponding destination plane, then copy all pixels in that plane; repeat for all four planes.)

Listing Three has an important limitation: It does not guarantee proper handling when the source and destination overlap, as in the case of a downward scroll, for example. Listing Three performs top-to-bottom, left-to-right copying. Downward scrolls require bottom-to-top copying; likewise, rightward horizontal scrolls require right-to-left copying. As it happens, my intended use for Listing Three is to copy images between off-screen memory and on-screen memory, and to save areas under pop-up menus and the like, so I don't really need overlap handling -- and I do really need to keep the size of this column down. However, you will surely want to add overlap handling if you plan to perform arbitrary scrolling and copying in display memory.

Now that we have a fast way to copy images around in display memory, we can draw icons and other images between two and four times faster than in mode 13h, depending on the speed of the VGA's display memory. (In case you're worried about the nibble-alignment limitation on fast copies, don't be; I'll address that fully next time, but the secret is to store all four possible rotations in off-screen memory, then select the correct one for each copy.) However, before our fast display memory-to-display memory copy routine can do

us any good, we must have a way to get pixel patterns from system memory into display memory, so that they can be copied with the fast copy routine.

Copying to Display Memory

The final piece of the puzzle is the system memory to display-memory-copy-routine shown in Listing Four (page 182). This routine assumes that pixels are stored in system memory in exactly the order in which they will ultimately appear on the screen; that is, in the same linear order that mode 13h uses. It would be more efficient to store all the pixels for one plane first, then all the pixels for the next plane, and so on for all four planes, because many OUTs could be avoided, but that would make images rather hard to create. And, while it is true that the speed of drawing images is, in general, often a critical performance factor, the speed of copying images from system memory to display memory is not particularly critical in mode X. Important images can be stored in off-screen memory and copied to the screen via the latches must faster than even the speediest system memory-to-display memory-copy-routine could manage.

I'm not going to present a routine to perform mode X copies from display memory to system memory, but such a routine would be a straightforward inverse of Listing Four.

Coming Up

Our Hero Risks Life, Limb,
and Word Count in a Thrilling Conclusion

Next month, I'll take all the model X tools we've developed, together with one more tool -- masked image copying -- and the remaining unexplored feature of mode X, page flipping, and build an animation application. I hope that when I'm done, you'll agree with me that mode X is the way to animate on the PC. I also hope that I can fit everything into one column; there are always so many interesting things to say that I have trouble keeping the size of these columns down, and mode X animation covers even more fertile ground than usual.

But, hey -- you've already heard about my programming demons; I'll spare you the writing demons. Besides, as I'm fond of saying, end users care about results, not how you produced them. For my writing, you folks are the end users -- and notice how remarkably little you care about how this magazine gets written and produced. You care that it shows up in your mailbox every month, and you care about how it got there. When you're a creator, the process matters. When you're a buyer, results are everything. All important. Sine qua non. The whole enchilada.

If you catch my drift.

Late Flash!

The Mode X mode set code in my July '91 column (Listing One, page 154) has a small -- but critical -- bug. On line 46, the value loaded into AL should be 0E3h, not 0E7h. Without this correction, the screen will roll on fixed-frequency (IBM 851X-style) monitors.

- -

GRAPHICS PROGRAMMING COLUMN
by Michael Abrash

[LISTING ONE]

```
/* Program to demonstrate mode X (320x240, 256 colors) patterned
   rectangle fills by filling the screen with adjacent 80x60
   rectangles in a variety of patterns. Tested with Borland C++
   2.0 in C compilation mode and the small model */
#include <conio.h>
```

```
#include <dos.h>

void Set320x240Mode(void);
void FillPatternX(int, int, int, int, unsigned int, char*);

/* 16 4x4 patterns */
static char Patt0[]={10,0,10,0,0,10,0,10,10,0,10,0,0,10,0,10};
static char Patt1[]={9,0,0,0,0,9,0,0,0,0,9,0,0,0,0,9};
static char Patt2[]={5,0,0,0,0,0,5,0,5,0,0,0,0,0,5,0};
static char Patt3[]={14,0,0,14,0,14,14,0,0,14,14,0,14,0,0,14};
static char Patt4[]={15,15,15,1,15,15,1,1,15,1,1,1,1,1,1,1};
static char Patt5[]={12,12,12,12,6,6,6,12,6,6,6,12,6,6,6,12};
static char Patt6[]={80,80,80,80,80,80,80,80,80,80,80,80,80,80,80,15};
static char Patt7[]={78,78,78,78,80,80,80,80,82,82,82,82,84,84,84,84};
static char Patt8[]={78,80,82,84,80,82,84,78,82,84,78,80,84,78,80,82};
static char Patt9[]={78,80,82,84,78,80,82,84,78,80,82,84,78,80,82,84};
static char Patt10[]={0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15};
static char Patt11[]={0,1,2,3,0,1,2,3,0,1,2,3,0,1,2,3};
static char Patt12[]={14,14,9,9,14,9,9,14,9,9,14,14,9,14,14,9};
static char Patt13[]={15,8,8,8,15,15,15,8,15,15,15,8,15,8,8,8};
static char Patt14[]={3,3,3,3,3,7,7,3,3,7,7,3,3,3,3,3};
static char Patt15[]={0,0,0,0,0,64,0,0,0,0,0,0,0,0,0,89};
/* Table of pointers to the 16 4x4 patterns with which to draw */
static char* PattTable[] = {Patt0,Patt1,Patt2,Patt3,Patt4,Patt5,Patt6,
      Patt7,Patt8,Patt9,Patt10,Patt11,Patt12,Patt13,Patt14,Patt15};
void main() {
   int i,j;
   union REGS regset;

   Set320x240Mode();
   for (j = 0; j < 4; j++) {
      for (i = 0; i < 4; i++) {
         FillPatternX(i*80,j*60,i*80+80,j*60+60,0,PattTable[j*4+i]);
      }
   }
   getch();
   regset.x.ax = 0x0003;   /* switch back to text mode and done */
   int86(0x10, &regset, &regset);
}
```

[LISTING TWO]

```
; Mode X (320x240, 256 colors) rectangle 4x4 pattern fill routine.
; Upper left corner of pattern is always aligned to a multiple-of-4
; row and column. Works on all VGAs. Uses approach of copying the
; pattern to off-screen display memory, then loading the latches with
; the pattern for each scan line and filling each scan line four
; pixels at a time. Fills up to but not including the column at EndX
; and the row at EndY. No clipping is performed. All ASM code tested
; with TASM 2. C near-callable as:
;    void FillPatternedX(int StartX, int StartY, int EndX, int EndY,
;       unsigned int PageBase, char* Pattern);

SC_INDEX equ    03c4h   ;Sequence Controller Index register port
MAP_MASK equ    02h     ;index in SC of Map Mask register
GC_INDEX equ    03ceh   ;Graphics Controller Index register port
BIT_MASK equ    08h     ;index in GC of Bit Mask register
PATTERN_BUFFER equ 0fffch ;offset in screen memory of the buffer used
                        ; to store each pattern during drawing
SCREEN_SEG equ  0a000h  ;segment of display memory in mode X
SCREEN_WIDTH equ 80     ;width of screen in addresses from one scan
                        ; line to the next
parms   struc
        dw      2 dup (?) ;pushed BP and return address
StartX  dw      ?       ;X coordinate of upper left corner of rect
StartY  dw      ?       ;Y coordinate of upper left corner of rect
EndX    dw      ?       ;X coordinate of lower right corner of rect
                        ; (the row at EndX is not filled)
EndY    dw      ?       ;Y coordinate of lower right corner of rect
```

```
                        ; (the column at EndY is not filled)
PageBase dw     ?       ;base offset in display memory of page in
                        ; which to fill rectangle
Pattern dw      ?       ;4x4 pattern with which to fill rectangle
parms   ends

NextScanOffset equ -2   ;local storage for distance from end of one
                        ; scan line to start of next
RectAddrWidth equ -4    ;local storage for address width of rectangle
Height   equ     -6     ;local storage for height of rectangle
STACK_FRAME_SIZE equ 6

        .model  small
        .data
; Plane masks for clipping left and right edges of rectangle.
LeftClipPlaneMask       db      00fh,00eh,00ch,008h
RightClipPlaneMask      db      00fh,001h,003h,007h
        .code
        public  _FillPatternX
_FillPatternX proc    near
        push    bp      ;preserve caller's stack frame
        mov     bp,sp   ;point to local stack frame
        sub     sp,STACK_FRAME_SIZE ;allocate space for local vars
        push    si      ;preserve caller's register variables
        push    di

        cld
        mov     ax,SCREEN_SEG   ;point ES to display memory
        mov     es,ax
                                ;copy pattern to display memory buffer
        mov     si,[bp+Pattern] ;point to pattern to fill with
        mov     di,PATTERN_BUFFER ;point ES:DI to pattern buffer
        mov     dx,SC_INDEX     ;point Sequence Controller Index to
        mov     al,MAP_MASK     ; Map Mask
        out     dx,al
        inc     dx              ;point to SC Data register
        mov     cx,4            ;4 pixel quadruplets in pattern
DownloadPatternLoop:
        mov     al,1            ;
        out     dx,al           ;select plane 0 for writes
        movsb                   ;copy over next plane 0 pattern pixel
        dec     di              ;stay at same address for next plane
        mov     al,2            ;
        out     dx,al           ;select plane 1 for writes
        movsb                   ;copy over next plane 1 pattern pixel
        dec     di              ;stay at same address for next plane
        mov     al,4            ;
        out     dx,al           ;select plane 2 for writes
        movsb                   ;copy over next plane 2 pattern pixel
        dec     di              ;stay at same address for next plane
        mov     al,8            ;
        out     dx,al           ;select plane 3 for writes
        movsb                   ;copy over next plane 3 pattern pixel
                                ; and advance address
        loop    DownloadPatternLoop

        mov     dx,GC_INDEX     ;set the bit mask to select all bits
        mov     ax,00000h+BIT_MASK ; from the latches and none from
        out     dx,ax           ; the CPU, so that we can write the
                                ; latch contents directly to memory
        mov     ax,[bp+StartY]  ;top rectangle scan line
        mov     si,ax
        and     si,011b         ;top rect scan line modulo 4
        add     si,PATTERN_BUFFER ;point to pattern scan line that
                                ; maps to top line of rect to draw
        mov     dx,SCREEN_WIDTH
        mul     dx      ;offset in page of top rectangle scan line
        mov     di,[bp+StartX]
        mov     bx,di
        shr     di,1    ;X/4 = offset of first rectangle pixel in scan
        shr     di,1    ; line
        add     di,ax   ;offset of first rectangle pixel in page
        add     di,[bp+PageBase] ;offset of first rectangle pixel in
                        ; display memory
```

```
        and     bx,0003h                ;look up left edge plane mask
        mov     ah,LeftClipPlaneMask[bx] ; to clip
        mov     bx,[bp+EndX]
        and     bx,0003h                 ;look up right edge plane
        mov     al,RightClipPlaneMask[bx] ; mask to clip
        mov     bx,ax                   ;put the masks in BX

        mov     cx,[bp+EndX]    ;calculate # of addresses across rect
        mov     ax,[bp+StartX]
        cmp     cx,ax
        jle     FillDone        ;skip if 0 or negative width
        dec     cx
        and     ax,not 011b
        sub     cx,ax
        shr     cx,1
        shr     cx,1    ;# of addresses across rectangle to fill - 1
        jnz     MasksSet ;there's more than one pixel to draw
        and     bh,bl   ;there's only one pixel, so combine the left
                        ; and right edge clip masks
MasksSet:
        mov     ax,[bp+EndY]
        sub     ax,[bp+StartY]  ;AX = height of rectangle
        jle     FillDone        ;skip if 0 or negative height
        mov     [bp+Height],ax
        mov     ax,SCREEN_WIDTH
        sub     ax,cx   ;distance from end of one scan line to start
        dec     ax      ; of next
        mov     [bp+NextScanOffset],ax
        mov     [bp+RectAddrWidth],cx ;remember width in addresses - 1
        mov     dx,SC_INDEX+1 ;point to Sequence Controller Data reg
                        ; (SC Index still points to Map Mask)
FillRowsLoop:
        mov     cx,[bp+RectAddrWidth] ;width across - 1
        mov     al,es:[si] ;read display memory to latch this scan
                        ; line's pattern
        inc     si      ;point to the next pattern scan line, wrapping
        jnz     short NoWrap ; back to the start of the pattern if
        sub     si,4    ; we've run off the end
NoWrap:
        mov     al,bh   ;put left-edge clip mask in AL
        out     dx,al   ;set the left-edge plane (clip) mask
        stosb           ;draw the left edge (pixels come from latches;
                        ; value written by CPU doesn't matter)
        dec     cx      ;count off left edge address
        js      FillLoopBottom ;that's the only address
        jz      DoRightEdge ;there are only two addresses
        mov     al,00fh ;middle addresses are drawn 4 pixels at a pop
        out     dx,al   ;set the middle pixel mask to no clip
        rep     stosb   ;draw the middle addresses four pixels apiece
                        ; (from latches; value written doesn't matter)
DoRightEdge:
        mov     al,bl   ;put right-edge clip mask in AL
        out     dx,al   ;set the right-edge plane (clip) mask
        stosb           ;draw the right edge (from latches; value
                        ; written doesn't matter)
FillLoopBottom:
        add     di,[bp+NextScanOffset] ;point to the start of the next scan
                                       ; line of the rectangle
        dec     word ptr [bp+Height] ;count down scan lines
        jnz     FillRowsLoop
FillDone:
        mov     dx,GC_INDEX+1 ;restore the bit mask to its default,
        mov     al,0ffh       ; which selects all bits from the CPU
        out     dx,al         ; and none from the latches (the GC
                              ; Index still points to Bit Mask)
        pop     di      ;restore caller's register variables
        pop     si
        mov     sp,bp   ;discard storage for local variables
        pop     bp      ;restore caller's stack frame
        ret
_FillPatternX endp
        end
```

[LISTING THREE]

```
; Mode X (320x240, 256 colors) display memory to display memory copy
; routine. Left edge of source rectangle modulo 4 must equal left edge
; of destination rectangle modulo 4. Works on all VGAs. Uses approach
; of reading 4 pixels at a time from the source into the latches, then
; writing the latches to the destination. Copies up to but not
; including the column at SourceEndX and the row at SourceEndY. No
; clipping is performed. Results are not guaranteed if the source and
; destination overlap. C near-callable as:
;    void CopyScreenToScreenX(int SourceStartX, int SourceStartY,
;       int SourceEndX, int SourceEndY, int DestStartX,
;       int DestStartY, unsigned int SourcePageBase,
;       unsigned int DestPageBase, int SourceBitmapWidth,
;       int DestBitmapWidth);

SC_INDEX equ    03c4h   ;Sequence Controller Index register port
MAP_MASK equ    02h     ;index in SC of Map Mask register
GC_INDEX equ    03ceh   ;Graphics Controller Index register port
BIT_MASK equ    08h     ;index in GC of Bit Mask register
SCREEN_SEG equ  0a000h  ;segment of display memory in mode X

parms   struc
        dw      2 dup (?) ;pushed BP and return address
SourceStartX dw ?       ;X coordinate of upper left corner of source
SourceStartY dw ?       ;Y coordinate of upper left corner of source
SourceEndX   dw ?       ;X coordinate of lower right corner of source
                        ; (the row at SourceEndX is not copied)
SourceEndY   dw ?       ;Y coordinate of lower right corner of source
                        ; (the column at SourceEndY is not copied)
DestStartX   dw ?       ;X coordinate of upper left corner of dest
DestStartY   dw ?       ;Y coordinate of upper left corner of dest
SourcePageBase dw ?     ;base offset in display memory of page in
                        ; which source resides
DestPageBase dw ?       ;base offset in display memory of page in
                        ; which dest resides
SourceBitmapWidth dw ?  ;# of pixels across source bitmap
                        ; (must be a multiple of 4)
DestBitmapWidth   dw ?  ;# of pixels across dest bitmap
                        ; (must be a multiple of 4)
parms   ends

SourceNextScanOffset equ -2   ;local storage for distance from end of
                              ; one source scan line to start of next
DestNextScanOffset equ -4    ;local storage for distance from end of
                              ; one dest scan line to start of next
RectAddrWidth equ -6    ;local storage for address width of rectangle
Height   equ     -8     ;local storage for height of rectangle
STACK_FRAME_SIZE equ 8

        .model  small
        .data
; Plane masks for clipping left and right edges of rectangle.
LeftClipPlaneMask       db      00fh,00eh,00ch,008h
RightClipPlaneMask      db      00fh,001h,003h,007h
        .code
        public  _CopyScreenToScreenX
_CopyScreenToScreenX proc    near
        push    bp      ;preserve caller's stack frame
        mov     bp,sp   ;point to local stack frame
        sub     sp,STACK_FRAME_SIZE ;allocate space for local vars
        push    si      ;preserve caller's register variables
        push    di
        push    ds

        cld
        mov     dx,GC_INDEX     ;set the bit mask to select all bits
        mov     ax,00000h+BIT_MASK ; from the latches and none from
        out     dx,ax           ; the CPU, so that we can write the
                                ; latch contents directly to memory
        mov     ax,SCREEN_SEG   ;point ES to display memory
        mov     es,ax
        mov     ax,[bp+DestBitmapWidth]
```

```
        shr     ax,1            ;convert to width in addresses
        shr     ax,1
        mul     [bp+DestStartY] ;top dest rect scan line
        mov     di,[bp+DestStartX]
        shr     di,1    ;X/4 = offset of first dest rect pixel in
        shr     di,1    ; scan line
        add     di,ax   ;offset of first dest rect pixel in page
        add     di,[bp+DestPageBase] ;offset of first dest rect pixel
                        ; in display memory
        mov     ax,[bp+SourceBitmapWidth]
        shr     ax,1            ;convert to width in addresses
        shr     ax,1
        mul     [bp+SourceStartY] ;top source rect scan line
        mov     si,[bp+SourceStartX]
        mov     bx,si
        shr     si,1    ;X/4 = offset of first source rect pixel in
        shr     si,1    ; scan line
        add     si,ax   ;offset of first source rect pixel in page
        add     si,[bp+SourcePageBase] ;offset of first source rect
                        ; pixel in display memory
        and     bx,0003h                ;look up left edge plane mask
        mov     ah,LeftClipPlaneMask[bx] ; to clip
        mov     bx,[bp+SourceEndX]
        and     bx,0003h                 ;look up right edge plane
        mov     al,RightClipPlaneMask[bx] ; mask to clip
        mov     bx,ax                   ;put the masks in BX

        mov     cx,[bp+SourceEndX]   ;calculate # of addresses across
        mov     ax,[bp+SourceStartX] ; rect
        cmp     cx,ax
        jle     CopyDone        ;skip if 0 or negative width
        dec     cx
        and     ax,not 011b
        sub     cx,ax
        shr     cx,1
        shr     cx,1    ;# of addresses across rectangle to copy - 1
        jnz     MasksSet ;there's more than one address to draw
        and     bh,bl   ;there's only one address, so combine the left
                        ; and right edge clip masks
MasksSet:
        mov     ax,[bp+SourceEndY]
        sub     ax,[bp+SourceStartY]  ;AX = height of rectangle
        jle     CopyDone        ;skip if 0 or negative height
        mov     [bp+Height],ax
        mov     ax,[bp+DestBitmapWidth]
        shr     ax,1            ;convert to width in addresses
        shr     ax,1
        sub     ax,cx   ;distance from end of one dest scan line to
        dec     ax      ; start of next
        mov     [bp+DestNextScanOffset],ax
        mov     ax,[bp+SourceBitmapWidth]
        shr     ax,1            ;convert to width in addresses
        shr     ax,1
        sub     ax,cx   ;distance from end of one source scan line to
        dec     ax      ; start of next
        mov     [bp+SourceNextScanOffset],ax
        mov     [bp+RectAddrWidth],cx ;remember width in addresses - 1
        mov     dx,SC_INDEX+1 ;point to Sequence Controller Data reg
                        ; (SC Index still points to Map Mask)
        mov     ax,es   ;DS=ES=screen segment for MOVS
        mov     ds,ax
CopyRowsLoop:
        mov     cx,[bp+RectAddrWidth] ;width across - 1
        mov     al,bh   ;put left-edge clip mask in AL
        out     dx,al   ;set the left-edge plane (clip) mask
        movsb           ;copy the left edge (pixels go through
                        ; latches)
        dec     cx      ;count off left edge address
        js      CopyLoopBottom ;that's the only address
        jz      DoRightEdge ;there are only two addresses
        mov     al,00fh ;middle addresses are drawn 4 pixels at a pop
        out     dx,al   ;set the middle pixel mask to no clip
        rep     movsb   ;draw the middle addresses four pixels apiece
                        ; (pixels copied through latches)
```

```
DoRightEdge:
        mov     al,bl   ;put right-edge clip mask in AL
        out     dx,al   ;set the right-edge plane (clip) mask
        movsb           ;draw the right edge (pixels copied through
                        ; latches)
CopyLoopBottom:
        add     si,[bp+SourceNextScanOffset] ;point to the start of
        add     di,[bp+DestNextScanOffset]  ; next source & dest lines
        dec     word ptr [bp+Height] ;count down scan lines
        jnz     CopyRowsLoop
CopyDone:
        mov     dx,GC_INDEX+1 ;restore the bit mask to its default,
        mov     al,0ffh         ; which selects all bits from the CPU
        out     dx,al           ; and none from the latches (the GC
                                ; Index still points to Bit Mask)
        pop     ds
        pop     di      ;restore caller's register variables
        pop     si
        mov     sp,bp   ;discard storage for local variables
        pop     bp      ;restore caller's stack frame
        ret
_CopyScreenToScreenX endp
        end
```

[LISTING FOUR]

```
; Mode X (320x240, 256 colors) system memory to display memory copy
; routine. Uses approach of changing the plane for each pixel copied;
; this is slower than copying all pixels in one plane, then all pixels
; in the next plane, and so on, but it is simpler; besides, images for
; which performance is critical should be stored in off-screen memory
; and copied to the screen via the latches. Copies up to but not
; including the column at SourceEndX and the row at SourceEndY. No
; clipping is performed. C near-callable as:
;    void CopySystemToScreenX(int SourceStartX, int SourceStartY,
;       int SourceEndX, int SourceEndY, int DestStartX,
;       int DestStartY, char* SourcePtr, unsigned int DestPageBase,
;       int SourceBitmapWidth, int DestBitmapWidth);

SC_INDEX equ    03c4h   ;Sequence Controller Index register port
MAP_MASK equ    02h     ;index in SC of Map Mask register
SCREEN_SEG equ  0a000h  ;segment of display memory in mode X

parms   struc
        dw      2 dup (?) ;pushed BP and return address
SourceStartX dw ?       ;X coordinate of upper left corner of source
SourceStartY dw ?       ;Y coordinate of upper left corner of source
SourceEndX   dw ?       ;X coordinate of lower right corner of source
                        ; (the row at EndX is not copied)
SourceEndY   dw ?       ;Y coordinate of lower right corner of source
                        ; (the column at EndY is not copied)
DestStartX   dw ?       ;X coordinate of upper left corner of dest
DestStartY   dw ?       ;Y coordinate of upper left corner of dest
SourcePtr    dw ?       ;pointer in DS to start of bitmap in which
                        ; source resides
DestPageBase dw ?       ;base offset in display memory of page in
                        ; which dest resides
SourceBitmapWidth dw ?  ;# of pixels across source bitmap
DestBitmapWidth   dw ?  ;# of pixels across dest bitmap
                        ; (must be a multiple of 4)
parms   ends

RectWidth equ   -2      ;local storage for width of rectangle
LeftMask equ    -4      ;local storage for left rect edge plane mask
STACK_FRAME_SIZE equ 4

        .model  small
        .code
        public  _CopySystemToScreenX
```

```
_CopySystemToScreenX proc    near
        push    bp      ;preserve caller's stack frame
        mov     bp,sp   ;point to local stack frame
        sub     sp,STACK_FRAME_SIZE ;allocate space for local vars
        push    si      ;preserve caller's register variables
        push    di

        cld
        mov     ax,SCREEN_SEG   ;point ES to display memory
        mov     es,ax
        mov     ax,[bp+SourceBitmapWidth]
        mul     [bp+SourceStartY] ;top source rect scan line
        add     ax,[bp+SourceStartX]
        add     ax,[bp+SourcePtr] ;offset of first source rect pixel
        mov     si,ax             ; in DS

        mov     ax,[bp+DestBitmapWidth]
        shr     ax,1            ;convert to width in addresses
        shr     ax,1
        mov     [bp+DestBitmapWidth],ax ;remember address width
        mul     [bp+DestStartY] ;top dest rect scan line
        mov     di,[bp+DestStartX]
        mov     cx,di
        shr     di,1    ;X/4 = offset of first dest rect pixel in
        shr     di,1    ; scan line
        add     di,ax   ;offset of first dest rect pixel in page
        add     di,[bp+DestPageBase] ;offset of first dest rect pixel
                        ; in display memory
        and     cl,011b ;CL = first dest pixel's plane
        mov     al,11h  ;upper nibble comes into play when plane wraps
                        ; from 3 back to 0
        shl     al,cl   ;set the bit for the first dest pixel's plane
        mov     [bp+LeftMask],al ; in each nibble to 1

        mov     cx,[bp+SourceEndX]   ;calculate # of pixels across
        sub     cx,[bp+SourceStartX] ; rect
        jle     CopyDone        ;skip if 0 or negative width
        mov     [bp+RectWidth],cx
        mov     bx,[bp+SourceEndY]
        sub     bx,[bp+SourceStartY]  ;BX = height of rectangle
        jle     CopyDone        ;skip if 0 or negative height
        mov     dx,SC_INDEX     ;point to SC Index register
        mov     al,MAP_MASK
        out     dx,al           ;point SC Index reg to the Map Mask
        inc     dx              ;point DX to SC Data reg
CopyRowsLoop:
        mov     ax,[bp+LeftMask]
        mov     cx,[bp+RectWidth]
        push    si      ;remember the start offset in the source
        push    di      ;remember the start offset in the dest
CopyScanLineLoop:
        out     dx,al           ;set the plane for this pixel
        movsb                   ;copy the pixel to the screen
        rol     al,1            ;set mask for next pixel's plane
        cmc                     ;advance destination address only when
        sbb     di,0            ; wrapping from plane 3 to plane 0
                                ; (else undo INC DI done by MOVSB)
        loop    CopyScanLineLoop
        pop     di      ;retrieve the dest start offset
        add     di,[bp+DestBitmapWidth] ;point to the start of the
                                        ; next scan line of the dest
        pop     si      ;retrieve the source start offset
        add     si,[bp+SourceBitmapWidth] ;point to the start of the
                                        ; next scan line of the source
        dec     bx      ;count down scan lines
        jnz     CopyRowsLoop
CopyDone:
        pop     di      ;restore caller's register variables
        pop     si
        mov     sp,bp   ;discard storage for local variables
        pop     bp      ;restore caller's stack frame
        ret
_CopySystemToScreenX endp
        end
```

Journal: Dr. Dobb's Journal Sept 1991 v16 n9 p127(7).

256-color VGA animation

Author: Abrash, Michael.
AttFile: GP-SEP91.ASC Source code listing.

Summary: The third in a series of articles on the undocumented 320-by-240-pixel, 256-color VGA mode (called 'X-mode' by the author) develops a routine for copying a nonrectangular image from system memory to display memory. The 'masked copying' routine employs a rectangular bitmap that acts as a mask that corresponds to the transmitted image. The bitmap controls which pixels in the display are set according to the image and which are left alone. Another routine enables masked copies of images in one section of display memory to be rapidly copied to another four at pixels at a time through selection of appropriate image and mask pairs in off-screen display memory. A compact but powerful animation application is developed that utilizes the various capabilities of the X-mode. The program is based on a page-flipping concept, with animation movement achieved through transposal of pixels through masked copying.

GRAPHICS PROGRAMMING COLUMN
by Michael Abrash

[LISTING ONE]

```
; Mode X (320x240, 256 colors) system memory-to-display memory masked copy
; routine. Not particularly fast; images for which performance is critical
; should be stored in off-screen memory and copied to screen via latches. Works
; on all VGAs. Copies up to but not including column at SourceEndX and row at
; SourceEndY. No clipping is performed. Mask and source image are both byte-
; per-pixel, and must be of same widths and reside at same coordinates in their
; respective bitmaps. Assembly code tested with TASM 2.0. C near-callable as:
;    void CopySystemToScreenMaskedX(int SourceStartX,
;       int SourceStartY, int SourceEndX, int SourceEndY,
;       int DestStartX, int DestStartY, char * SourcePtr,
;       unsigned int DestPageBase, int SourceBitmapWidth,
;       int DestBitmapWidth, char * MaskPtr);

SC_INDEX equ    03c4h   ;Sequence Controller Index register port
MAP_MASK equ    02h     ;index in SC of Map Mask register
SCREEN_SEG equ  0a000h  ;segment of display memory in mode X

parms   struc
        dw      2 dup (?) ;pushed BP and return address
SourceStartX dw ?       ;X coordinate of upper left corner of source
                        ; (source is in system memory)
SourceStartY dw ?       ;Y coordinate of upper left corner of source
SourceEndX   dw ?       ;X coordinate of lower right corner of source
                        ; (the column at EndX is not copied)
SourceEndY   dw ?       ;Y coordinate of lower right corner of source
                        ; (the row at EndY is not copied)
DestStartX   dw ?       ;X coordinate of upper left corner of dest
                        ; (destination is in display memory)
DestStartY   dw ?       ;Y coordinate of upper left corner of dest
SourcePtr    dw ?       ;pointer in DS to start of bitmap which source resides
DestPageBase dw ?       ;base offset in display memory of page in
                        ; which dest resides
SourceBitmapWidth dw ?  ;# of pixels across source bitmap (also must
                        ; be width across the mask)
DestBitmapWidth   dw ?  ;# of pixels across dest bitmap (must be multiple of 4)
MaskPtr      dw ?       ;pointer in DS to start of bitmap in which mask
                        ; resides (byte-per-pixel format, just like the source
                        ; image; 0-bytes mean don't copy corresponding source
                        ; pixel, 1-bytes mean do copy)
parms   ends
```

```
RectWidth equ   -2      ;local storage for width of rectangle
RectHeight equ  -4      ;local storage for height of rectangle
LeftMask equ    -6      ;local storage for left rect edge plane mask
STACK_FRAME_SIZE equ 6
        .model  small
        .code
        public  _CopySystemToScreenMaskedX
_CopySystemToScreenMaskedX proc    near
        push    bp      ;preserve caller's stack frame
        mov     bp,sp   ;point to local stack frame
        sub     sp,STACK_FRAME_SIZE ;allocate space for local vars
        push    si      ;preserve caller's register variables
        push    di

        mov     ax,SCREEN_SEG   ;point ES to display memory
        mov     es,ax
        mov     ax,[bp+SourceBitmapWidth]
        mul     [bp+SourceStartY] ;top source rect scan line
        add     ax,[bp+SourceStartX]
        mov     bx,ax
        add     ax,[bp+SourcePtr] ;offset of first source rect pixel
        mov     si,ax             ; in DS
        add     bx,[bp+MaskPtr] ;offset of first mask pixel in DS

        mov     ax,[bp+DestBitmapWidth]
        shr     ax,1            ;convert to width in addresses
        shr     ax,1
        mov     [bp+DestBitmapWidth],ax ;remember address width
        mul     [bp+DestStartY] ;top dest rect scan line
        mov     di,[bp+DestStartX]
        mov     cx,di
        shr     di,1    ;X/4 = offset of first dest rect pixel in
        shr     di,1    ; scan line
        add     di,ax   ;offset of first dest rect pixel in page
        add     di,[bp+DestPageBase] ;offset of first dest rect pixel
                        ; in display memory
        and     cl,011b ;CL = first dest pixel's plane
        mov     al,11h  ;upper nibble comes into play when plane wraps
                        ; from 3 back to 0
        shl     al,cl   ;set the bit for the first dest pixel's plane
        mov     [bp+LeftMask],al ; in each nibble to 1

        mov     ax,[bp+SourceEndX]   ;calculate # of pixels across
        sub     ax,[bp+SourceStartX] ; rect
        jle     CopyDone        ;skip if 0 or negative width
        mov     [bp+RectWidth],ax
        sub     word ptr [bp+SourceBitmapWidth],ax
                ;distance from end of one source scan line to start of next
        mov     ax,[bp+SourceEndY]
        sub     ax,[bp+SourceStartY] ;height of rectangle
        jle     CopyDone        ;skip if 0 or negative height
        mov     [bp+RectHeight],ax
        mov     dx,SC_INDEX     ;point to SC Index register
        mov     al,MAP_MASK
        out     dx,al           ;point SC Index reg to the Map Mask
        inc     dx              ;point DX to SC Data reg
CopyRowsLoop:
        mov     al,[bp+LeftMask]
        mov     cx,[bp+RectWidth]
        push    di      ;remember the start offset in the dest
CopyScanLineLoop:
        cmp     byte ptr [bx],0 ;is this pixel mask-enabled?
        jz      MaskOff         ;no, so don't draw it
                                ;yes, draw the pixel
        out     dx,al           ;set the plane for this pixel
        mov     ah,[si]         ;get the pixel from the source
        mov     es:[di],ah      ;copy the pixel to the screen
MaskOff:
        inc     bx              ;advance the mask pointer
        inc     si              ;advance the source pointer
        rol     al,1            ;set mask for next pixel's plane
        adc     di,0            ;advance destination address only when
                                ; wrapping from plane 3 to plane 0
        loop    CopyScanLineLoop
```

```
        pop     di              ;retrieve the dest start offset
        add     di,[bp+DestBitmapWidth] ;point to the start of the
                                        ; next scan line of the dest
        add     si,[bp+SourceBitmapWidth] ;point to the start of the
                                        ; next scan line of the source
        add     bx,[bp+SourceBitmapWidth] ;point to the start of the
                                        ; next scan line of the mask
        dec     word ptr [bp+RectHeight] ;count down scan lines
        jnz     CopyRowsLoop
CopyDone:
        pop     di      ;restore caller's register variables
        pop     si
        mov     sp,bp   ;discard storage for local variables
        pop     bp      ;restore caller's stack frame
        ret
_CopySystemToScreenMaskedX endp
        end
```

[LISTING TWO]

```
; Mode X (320x240, 256 colors) display memory to display memory masked copy
; routine. Works on all VGAs. Uses approach of reading 4 pixels at a time from
; source into latches, then writing latches to destination, using Map Mask
; register to perform masking. Copies up to but not including column at
; SourceEndX and row at SourceEndY. No clipping is performed. Results are not
; guaranteed if source and destination overlap. C near-callable as:
;    void CopyScreenToScreenMaskedX(int SourceStartX,
;       int SourceStartY, int SourceEndX, int SourceEndY,
;       int DestStartX, int DestStartY, MaskedImage * Source,
;       unsigned int DestPageBase, int DestBitmapWidth);

SC_INDEX equ    03c4h   ;Sequence Controller Index register port
MAP_MASK equ    02h     ;index in SC of Map Mask register
GC_INDEX equ    03ceh   ;Graphics Controller Index register port
BIT_MASK equ    08h     ;index in GC of Bit Mask register
SCREEN_SEG equ  0a000h  ;segment of display memory in mode X

parms   struc
        dw      2 dup (?) ;pushed BP and return address
SourceStartX dw ?       ;X coordinate of upper left corner of source
SourceStartY dw ?       ;Y coordinate of upper left corner of source
SourceEndX   dw ?       ;X coordinate of lower right corner of source
                        ; (the column at SourceEndX is not copied)
SourceEndY   dw ?       ;Y coordinate of lower right corner of source
                        ; (the row at SourceEndY is not copied)
DestStartX   dw ?       ;X coordinate of upper left corner of dest
DestStartY   dw ?       ;Y coordinate of upper left corner of dest
Source       dw ?       ;pointer to MaskedImage struct for source
                        ; which source resides
DestPageBase dw ?       ;base offset in display memory of page in
                        ; which dest resides
DestBitmapWidth   dw ?  ;# of pixels across dest bitmap (must be multiple of 4)
parms   ends

SourceNextScanOffset equ -2   ;local storage for distance from end of
                              ; one source scan line to start of next
DestNextScanOffset equ -4    ;local storage for distance from end of
                              ; one dest scan line to start of next
RectAddrWidth equ -6    ;local storage for address width of rectangle
RectHeight    equ -8    ;local storage for height of rectangle
SourceBitmapWidth equ -10 ;local storage for width of source bitmap
                        ; (in addresses)
STACK_FRAME_SIZE equ 10
MaskedImage     struc
 Alignments     dw  4 dup(?) ;pointers to AlignedMaskedImages for the
                             ; 4 possible destination image alignments
MaskedImage     ends
AlignedMaskedImage      struc
 ImageWidth     dw      ? ;image width in addresses (also mask width in bytes)
```

```
 ImagePtr       dw      ? ;offset of image bitmap in display memory
 MaskPtr        dw      ? ;pointer to mask bitmap in DS
AlignedMaskedImage      ends
        .model  small
        .code
        public  _CopyScreenToScreenMaskedX
_CopyScreenToScreenMaskedX proc    near
        push    bp      ;preserve caller's stack frame
        mov     bp,sp   ;point to local stack frame
        sub     sp,STACK_FRAME_SIZE ;allocate space for local vars
        push    si      ;preserve caller's register variables
        push    di

        cld
        mov     dx,GC_INDEX     ;set the bit mask to select all bits
        mov     ax,00000h+BIT_MASK ; from the latches and none from
        out     dx,ax           ; the CPU, so that we can write the
                                ; latch contents directly to memory
        mov     ax,SCREEN_SEG   ;point ES to display memory
        mov     es,ax
        mov     ax,[bp+DestBitmapWidth]
        shr     ax,1            ;convert to width in addresses
        shr     ax,1
        mul     [bp+DestStartY] ;top dest rect scan line
        mov     di,[bp+DestStartX]
        mov     si,di
        shr     di,1    ;X/4 = offset of first dest rect pixel in
        shr     di,1    ; scan line
        add     di,ax   ;offset of first dest rect pixel in page
        add     di,[bp+DestPageBase] ;offset of first dest rect pixel
                        ; in display memory. now look up the image that's
                        ; aligned to match left-edge alignment of destination
        and     si,3    ;DestStartX modulo 4
        mov     cx,si   ;set aside alignment for later
        shl     si,1    ;prepare for word look-up
        mov     bx,[bp+Source] ;point to source MaskedImage structure
        mov     bx,[bx+Alignments+si] ;point to AlignedMaskedImage
                        ; struc for current left edge alignment
        mov     ax,[bx+ImageWidth] ;image width in addresses
        mov     [bp+SourceBitmapWidth],ax ;remember image width in
                                          ; addresses
        mul     [bp+SourceStartY] ;top source rect scan line
        mov     si,[bp+SourceStartX]
        shr     si,1    ;X/4 = address of first source rect pixel in
        shr     si,1    ; scan line
        add     si,ax   ;offset of first source rect pixel in image
        mov     ax,si
        add     si,[bx+MaskPtr] ;point to mask offset of first mask pixel in DS
        mov     bx,[bx+ImagePtr] ;offset of first source rect pixel
        add     bx,ax            ; in display memory

        mov     ax,[bp+SourceStartX] ;calculate # of addresses across
        add     ax,cx                ; rect, shifting if necessary to
        add     cx,[bp+SourceEndX]   ; account for alignment
        cmp     cx,ax
        jle     CopyDone        ;skip if 0 or negative width
        add     cx,3
        and     ax,not 011b
        sub     cx,ax
        shr     cx,1
        shr     cx,1    ;# of addresses across rectangle to copy
        mov     ax,[bp+SourceEndY]
        sub     ax,[bp+SourceStartY]  ;AX = height of rectangle
        jle     CopyDone        ;skip if 0 or negative height
        mov     [bp+RectHeight],ax
        mov     ax,[bp+DestBitmapWidth]
        shr     ax,1            ;convert to width in addresses
        shr     ax,1
        sub     ax,cx ;distance from end of one dest scan line to start of next
        mov     [bp+DestNextScanOffset],ax
        mov     ax,[bp+SourceBitmapWidth] ;width in addresses
        sub     ax,cx ;distance from end of source scan line to start of next
        mov     [bp+SourceNextScanOffset],ax
        mov     [bp+RectAddrWidth],cx ;remember width in addresses
```

```
        mov     dx,SC_INDEX
        mov     al,MAP_MASK
        out     dx,al           ;point SC Index register to Map Mask
        inc     dx              ;point to SC Data register
CopyRowsLoop:
        mov     cx,[bp+RectAddrWidth] ;width across
CopyScanLineLoop:
        lodsb                   ;get the mask for this four-pixel set
                                ; and advance the mask pointer
        out     dx,al           ;set the mask
        mov     al,es:[bx]      ;load the latches with 4-pixel set from source
        mov     es:[di],al      ;copy the four-pixel set to the dest
        inc     bx              ;advance the source pointer
        inc     di              ;advance the destination pointer
        dec     cx              ;count off four-pixel sets
        jnz     CopyScanLineLoop

        mov     ax,[bp+SourceNextScanOffset]
        add     si,ax                       ;point to the start of
        add     bx,ax                       ; the next source, mask,
        add     di,[bp+DestNextScanOffset] ; and dest lines
        dec     word ptr [bp+RectHeight] ;count down scan lines
        jnz     CopyRowsLoop
CopyDone:
        mov     dx,GC_INDEX+1   ;restore the bit mask to its default,
        mov     al,0ffh         ; which selects all bits from the CPU
        out     dx,al           ; and none from the latches (the GC
                                ; Index still points to Bit Mask)
        pop     di      ;restore caller's register variables
        pop     si
        mov     sp,bp   ;discard storage for local variables
        pop     bp      ;restore caller's stack frame
        ret
_CopyScreenToScreenMaskedX endp
        end
```

[LISTING THREE]

```
/* Generates all four possible mode X image/mask alignments, stores image
alignments in display memory, allocates memory for and generates mask
alignments, and fills out an AlignedMaskedImage structure. Image and mask must
both be in byte-per-pixel form, and must both be of width ImageWidth. Mask
maps isomorphically (one to one) onto image, with each 0-byte in mask masking
off corresponding image pixel (causing it not to be drawn), and each non-0-byte
allowing corresponding image pixel to be drawn. Returns 0 if failure, or # of
display memory addresses (4-pixel sets) used if success. For simplicity,
allocated memory is not deallocated in case of failure. Compiled with
Borland C++ 2.0 in C compilation mode. */

#include <stdio.h>
#include <stdlib.h>
#include "maskim.h"

extern void CopySystemToScreenX(int, int, int, int, int, int, char *,
   unsigned int, int, int);
unsigned int CreateAlignedMaskedImage(MaskedImage * ImageToSet,
   unsigned int DispMemStart, char * Image, int ImageWidth,
   int ImageHeight, char * Mask)
{
   int Align, ScanLine, BitNum, Size, TempImageWidth;
   unsigned char MaskTemp;
   unsigned int DispMemOffset = DispMemStart;
   AlignedMaskedImage *WorkingAMImage;
   char *NewMaskPtr, *OldMaskPtr;
   /* Generate each of the four alignments in turn */
   for (Align = 0; Align < 4; Align++) {
      /* Allocate space for the AlignedMaskedImage struct for this
         alignment */
      if ((WorkingAMImage = ImageToSet->Alignments[Align] =
            malloc(sizeof(AlignedMaskedImage))) == NULL)
         return 0;
```

```
      WorkingAMImage->ImageWidth =
            (ImageWidth + Align + 3) / 4; /* width in 4-pixel sets */
      WorkingAMImage->ImagePtr = DispMemOffset; /* image dest */
      /* Download this alignment of the image */
      CopySystemToScreenX(0, 0, ImageWidth, ImageHeight, Align, 0,
            Image, DispMemOffset, ImageWidth,
            WorkingAMImage->ImageWidth * 4);
      /* Calculate the number of bytes needed to store the mask in
         nibble (Map Mask-ready) form, then allocate that space */
      Size = WorkingAMImage->ImageWidth * ImageHeight;
      if ((WorkingAMImage->MaskPtr = malloc(Size)) == NULL)
         return 0;
      /* Generate this nibble oriented (Map Mask-ready) alignment of
         the mask, one scan line at a time */
      OldMaskPtr = Mask;
      NewMaskPtr = WorkingAMImage->MaskPtr;
      for (ScanLine = 0; ScanLine < ImageHeight; ScanLine++) {
         BitNum = Align;
         MaskTemp = 0;
         TempImageWidth = ImageWidth;
         do {
            /* Set the mask bit for next pixel according to its alignment */
            MaskTemp |= (*OldMaskPtr++ != 0) << BitNum;
            if (++BitNum > 3) {
               *NewMaskPtr++ = MaskTemp;
               MaskTemp = BitNum = 0;
            }
         } while (--TempImageWidth);
         /* Set any partial final mask on this scan line */
         if (BitNum != 0) *NewMaskPtr++ = MaskTemp;
      }
      DispMemOffset += Size; /* mark off the space we just used */
   }
   return DispMemOffset - DispMemStart;
}
```

[LISTING FOUR]

```
/* MASKIM.H: structures used for storing and manipulating masked
   images */
/* Describes one alignment of a mask-image pair */
typedef struct {
   int ImageWidth; /* image width in addresses in display memory (also
                      mask width in bytes) */
   unsigned int ImagePtr; /* offset of image bitmap in display mem */
   char *MaskPtr;  /* pointer to mask bitmap */
} AlignedMaskedImage;
/* Describes all four alignments of a mask-image pair */
typedef struct {
   AlignedMaskedImage *Alignments[4]; /* ptrs to AlignedMaskedImage
                                         structs for four possible destination
                                         image alignments */
} MaskedImage;
```

[LISTING FIVE]

```
/* Sample mode X VGA animation program. Portions of this code first appeared
in PC Techniques. Compiled with Borland C++ 2.0 in C compilation mode. */

#include <stdio.h>
#include <conio.h>
#include <dos.h>
#include <math.h>
#include "maskim.h"

#define SCREEN_SEG         0xA000
#define SCREEN_WIDTH       320
#define SCREEN_HEIGHT      240
#define PAGE0_START_OFFSET 0
```

```
#define PAGE1_START_OFFSET (((long)SCREEN_HEIGHT*SCREEN_WIDTH)/4)
#define BG_START_OFFSET    (((long)SCREEN_HEIGHT*SCREEN_WIDTH*2)/4)
#define DOWNLOAD_START_OFFSET (((long)SCREEN_HEIGHT*SCREEN_WIDTH*3)/4)

static unsigned int PageStartOffsets[2] =
   {PAGE0_START_OFFSET,PAGE1_START_OFFSET};
static char GreenAndBrownPattern[] =
   {2,6,2,6, 6,2,6,2, 2,6,2,6, 6,2,6,2};
static char PineTreePattern[] = {2,2,2,2, 2,6,2,6, 2,2,6,2, 2,2,2,2};
static char BrickPattern[] = {6,6,7,6, 7,7,7,7, 7,6,6,6, 7,7,7,7,};
static char RoofPattern[] = {8,8,8,7, 7,7,7,7, 8,8,8,7, 8,8,8,7};

#define SMOKE_WIDTH  7
#define SMOKE_HEIGHT 7
static char SmokePixels[] = {
   0, 0,15,15,15, 0, 0,
   0, 7, 7,15,15,15, 0,
   8, 7, 7, 7,15,15,15,
   8, 7, 7, 7, 7,15,15,
   0, 8, 7, 7, 7, 7,15,
   0, 0, 8, 7, 7, 7, 0,
   0, 0, 0, 8, 8, 0, 0};
static char SmokeMask[] = {
   0, 0, 1, 1, 1, 0, 0,
   0, 1, 1, 1, 1, 1, 0,
   1, 1, 1, 1, 1, 1, 1,
   1, 1, 1, 1, 1, 1, 1,
   1, 1, 1, 1, 1, 1, 1,
   0, 1, 1, 1, 1, 1, 0,
   0, 0, 1, 1, 1, 0, 0};
#define KITE_WIDTH  10
#define KITE_HEIGHT 16
static char KitePixels[] = {
   0, 0, 0, 0,45, 0, 0, 0, 0, 0,
   0, 0, 0,46,46,46, 0, 0, 0, 0,
   0, 0,47,47,47,47,47, 0, 0, 0,
   0,48,48,48,48,48,48,48, 0, 0,
  49,49,49,49,49,49,49,49,49, 0,
   0,50,50,50,50,50,50,50, 0, 0,
   0,51,51,51,51,51,51,51, 0, 0,
   0, 0,52,52,52,52,52, 0, 0, 0,
   0, 0,53,53,53,53,53, 0, 0, 0,
   0, 0, 0,54,54,54, 0, 0, 0, 0,
   0, 0, 0,55,55,55, 0, 0, 0, 0,
   0, 0, 0, 0,58, 0, 0, 0, 0, 0,
   0, 0, 0, 0,59, 0, 0, 0, 0,66,
   0, 0, 0, 0,60, 0, 0,64, 0,65,
   0, 0, 0, 0, 0,61, 0, 0,64, 0,
   0, 0, 0, 0, 0, 0,62,63, 0,64};
static char KiteMask[] = {
   0, 0, 0, 0, 1, 0, 0, 0, 0, 0,
   0, 0, 0, 1, 1, 1, 0, 0, 0, 0,
   0, 0, 1, 1, 1, 1, 1, 0, 0, 0,
   0, 1, 1, 1, 1, 1, 1, 1, 0, 0,
   1, 1, 1, 1, 1, 1, 1, 1, 1, 0,
   0, 1, 1, 1, 1, 1, 1, 1, 0, 0,
   0, 1, 1, 1, 1, 1, 1, 1, 0, 0,
   0, 0, 1, 1, 1, 1, 1, 0, 0, 0,
   0, 0, 1, 1, 1, 1, 1, 0, 0, 0,
   0, 0, 0, 1, 1, 1, 0, 0, 0, 0,
   0, 0, 0, 1, 1, 1, 0, 0, 0, 0,
   0, 0, 0, 0, 1, 0, 0, 0, 0, 0,
   0, 0, 0, 0, 1, 0, 0, 0, 0, 1,
   0, 0, 0, 0, 1, 0, 0, 1, 0, 1,
   0, 0, 0, 0, 0, 1, 0, 0, 1, 0,
   0, 0, 0, 0, 0, 0, 1, 1, 0, 1};
static MaskedImage KiteImage;

#define NUM_OBJECTS  20
typedef struct {
   int X,Y,Width,Height,XDir,YDir,XOtherPage,YOtherPage;
   MaskedImage *Image;
} AnimatedObject;
AnimatedObject AnimatedObjects[] = {
```

```
   {  0,  0,KITE_WIDTH,KITE_HEIGHT, 1, 1,  0,  0,&KiteImage},
   { 10, 10,KITE_WIDTH,KITE_HEIGHT, 0, 1, 10, 10,&KiteImage},
   { 20, 20,KITE_WIDTH,KITE_HEIGHT,-1, 1, 20, 20,&KiteImage},
   { 30, 30,KITE_WIDTH,KITE_HEIGHT,-1,-1, 30, 30,&KiteImage},
   { 40, 40,KITE_WIDTH,KITE_HEIGHT, 1,-1, 40, 40,&KiteImage},
   { 50, 50,KITE_WIDTH,KITE_HEIGHT, 0,-1, 50, 50,&KiteImage},
   { 60, 60,KITE_WIDTH,KITE_HEIGHT, 1, 0, 60, 60,&KiteImage},
   { 70, 70,KITE_WIDTH,KITE_HEIGHT,-1, 0, 70, 70,&KiteImage},
   { 80, 80,KITE_WIDTH,KITE_HEIGHT, 1, 2, 80, 80,&KiteImage},
   { 90, 90,KITE_WIDTH,KITE_HEIGHT, 0, 2, 90, 90,&KiteImage},
   {100,100,KITE_WIDTH,KITE_HEIGHT,-1, 2,100,100,&KiteImage},
   {110,110,KITE_WIDTH,KITE_HEIGHT,-1,-2,110,110,&KiteImage},
   {120,120,KITE_WIDTH,KITE_HEIGHT, 1,-2,120,120,&KiteImage},
   {130,130,KITE_WIDTH,KITE_HEIGHT, 0,-2,130,130,&KiteImage},
   {140,140,KITE_WIDTH,KITE_HEIGHT, 2, 0,140,140,&KiteImage},
   {150,150,KITE_WIDTH,KITE_HEIGHT,-2, 0,150,150,&KiteImage},
   {160,160,KITE_WIDTH,KITE_HEIGHT, 2, 2,160,160,&KiteImage},
   {170,170,KITE_WIDTH,KITE_HEIGHT,-2, 2,170,170,&KiteImage},
   {180,180,KITE_WIDTH,KITE_HEIGHT,-2,-2,180,180,&KiteImage},
   {190,190,KITE_WIDTH,KITE_HEIGHT, 2,-2,190,190,&KiteImage},
};
void main(void);
void DrawBackground(unsigned int);
void MoveObject(AnimatedObject *);
extern void Set320x240Mode(void);
extern void FillRectangleX(int, int, int, int, unsigned int, int);
extern void FillPatternX(int, int, int, int, unsigned int, char*);
extern void CopySystemToScreenMaskedX(int, int, int, int, int, int,
   char *, unsigned int, int, int, char *);
extern void CopyScreenToScreenX(int, int, int, int, int, int,
   unsigned int, unsigned int, int, int);
extern unsigned int CreateAlignedMaskedImage(MaskedImage *,
   unsigned int, char *, int, int, char *);
extern void CopyScreenToScreenMaskedX(int, int, int, int, int, int,
   MaskedImage *, unsigned int, int);
extern void ShowPage(unsigned int);

void main()
{
   int DisplayedPage, NonDisplayedPage, Done, i;
   union REGS regset;
   Set320x240Mode();
   /* Download the kite image for fast copying later */
   if (CreateAlignedMaskedImage(&KiteImage, DOWNLOAD_START_OFFSET,
         KitePixels, KITE_WIDTH, KITE_HEIGHT, KiteMask) == 0) {
      regset.x.ax = 0x0003; int86(0x10, &regset, &regset);
      printf("Couldn't get memory\n"); exit();
   }
   /* Draw the background to the background page */
   DrawBackground(BG_START_OFFSET);
   /* Copy the background to both displayable pages */
   CopyScreenToScreenX(0, 0, SCREEN_WIDTH, SCREEN_HEIGHT, 0, 0,
         BG_START_OFFSET, PAGE0_START_OFFSET, SCREEN_WIDTH,
         SCREEN_WIDTH);
   CopyScreenToScreenX(0, 0, SCREEN_WIDTH, SCREEN_HEIGHT, 0, 0,
         BG_START_OFFSET, PAGE1_START_OFFSET, SCREEN_WIDTH,
         SCREEN_WIDTH);
   /* Move the objects and update their images in the nondisplayed
      page, then flip the page, until Esc is pressed */
   Done = DisplayedPage = 0;
   do {
      NonDisplayedPage = DisplayedPage ^ 1;
      /* Erase each object in nondisplayed page by copying block from
            background page at last location in that page */
      for (i=0; i<NUM_OBJECTS; i++) {
         CopyScreenToScreenX(AnimatedObjects[i].XOtherPage,
               AnimatedObjects[i].YOtherPage,
               AnimatedObjects[i].XOtherPage +
               AnimatedObjects[i].Width,
               AnimatedObjects[i].YOtherPage +
               AnimatedObjects[i].Height,
               AnimatedObjects[i].XOtherPage,
               AnimatedObjects[i].YOtherPage, BG_START_OFFSET,
               PageStartOffsets[NonDisplayedPage], SCREEN_WIDTH,
```

```
                SCREEN_WIDTH);
      }
      /* Move and draw each object in the nondisplayed page */
      for (i=0; i<NUM_OBJECTS; i++) {
         MoveObject(&AnimatedObjects[i]);
         /* Draw object into nondisplayed page at new location */
         CopyScreenToScreenMaskedX(0, 0, AnimatedObjects[i].Width,
               AnimatedObjects[i].Height, AnimatedObjects[i].X,
               AnimatedObjects[i].Y, AnimatedObjects[i].Image,
               PageStartOffsets[NonDisplayedPage], SCREEN_WIDTH);
      }
      /* Flip to the page into which we just drew */
      ShowPage(PageStartOffsets[DisplayedPage = NonDisplayedPage]);
      /* See if it's time to end */
      if (kbhit()) {
         if (getch() == 0x1B) Done = 1;   /* Esc to end */
      }
   } while (!Done);
   /* Restore text mode and done */
   regset.x.ax = 0x0003; int86(0x10, &regset, &regset);
}
void DrawBackground(unsigned int PageStart)
{
   int i,j,Temp;
   /* Fill the screen with cyan */
   FillRectangleX(0, 0, SCREEN_WIDTH, SCREEN_HEIGHT, PageStart, 11);
   /* Draw a green and brown rectangle to create a flat plain */
   FillPatternX(0, 160, SCREEN_WIDTH, SCREEN_HEIGHT, PageStart,
                                                   GreenAndBrownPattern);
   /* Draw blue water at the bottom of the screen */
   FillRectangleX(0, SCREEN_HEIGHT-30, SCREEN_WIDTH, SCREEN_HEIGHT,
                                                         PageStart, 1);
   /* Draw a brown mountain rising out of the plain */
   for (i=0; i<120; i++)
      FillRectangleX(SCREEN_WIDTH/2-30-i, 51+i, SCREEN_WIDTH/2-30+i+1,
                                                   51+i+1, PageStart, 6);
   /* Draw a yellow sun by overlapping rects of various shapes */
   for (i=0; i<=20; i++) {
      Temp = (int)(sqrt(20.0*20.0 - (float)i*(float)i) + 0.5);
      FillRectangleX(SCREEN_WIDTH-25-i, 30-Temp, SCREEN_WIDTH-25+i+1,
                                                30+Temp+1, PageStart, 14);
   }
   /* Draw green trees down the side of the mountain */
   for (i=10; i<90; i += 15)
      for (j=0; j<20; j++)
       FillPatternX(SCREEN_WIDTH/2+i-j/3-15, i+j+51,SCREEN_WIDTH/2+i+j/3-15+1,
                                      i+j+51+1, PageStart, PineTreePattern);
   /* Draw a house on the plain */
   FillPatternX(265, 150, 295, 170, PageStart, BrickPattern);
   FillPatternX(265, 130, 270, 150, PageStart, BrickPattern);
   for (i=0; i<12; i++)
      FillPatternX(280-i*2, 138+i, 280+i*2+1, 138+i+1, PageStart, RoofPattern);
   /* Finally, draw puffs of smoke rising from the chimney */
   for (i=0; i<4; i++)
      CopySystemToScreenMaskedX(0, 0, SMOKE_WIDTH, SMOKE_HEIGHT, 264,
        110-i*20, SmokePixels, PageStart, SMOKE_WIDTH,SCREEN_WIDTH, SmokeMask);
}
/* Move the specified object, bouncing at the edges of the screen and
   remembering where the object was before the move for erasing next time */
void MoveObject(AnimatedObject * ObjectToMove) {
   int X, Y;
   X = ObjectToMove->X + ObjectToMove->XDir;
   Y = ObjectToMove->Y + ObjectToMove->YDir;
   if ((X < 0) || (X > (SCREEN_WIDTH - ObjectToMove->Width))) {
      ObjectToMove->XDir = -ObjectToMove->XDir;
      X = ObjectToMove->X + ObjectToMove->XDir;
   }
   if ((Y < 0) || (Y > (SCREEN_HEIGHT - ObjectToMove->Height))) {
      ObjectToMove->YDir = -ObjectToMove->YDir;
      Y = ObjectToMove->Y + ObjectToMove->YDir;
   }
   /* Remember previous location for erasing purposes */
   ObjectToMove->XOtherPage = ObjectToMove->X;
   ObjectToMove->YOtherPage = ObjectToMove->Y;
```

```
   ObjectToMove->X = X; /* set new location */
   ObjectToMove->Y = Y;
}
```

[LISTING SIX]

```
; Shows the page at the specified offset in the bitmap. Page is displayed when
; this routine returns. This code first appeared in PC Techniques.
; C near-callable as: void ShowPage(unsigned int StartOffset);

INPUT_STATUS_1  equ     03dah   ;Input Status 1 register
CRTC_INDEX      equ     03d4h   ;CRT Controller Index reg
START_ADDRESS_HIGH equ  0ch     ;bitmap start address high byte
START_ADDRESS_LOW equ   0dh     ;bitmap start address low byte

ShowPageParms   struc
        dw      2 dup (?) ;pushed BP and return address
StartOffset dw  ?       ;offset in bitmap of page to display
ShowPageParms   ends
        .model  small
        .code
        public  _ShowPage
_ShowPage       proc    near
        push    bp      ;preserve caller's stack frame
        mov     bp,sp   ;point to local stack frame
; Wait for display enable to be active (status is active low), to be
; sure both halves of the start address will take in the same frame.
        mov     bl,START_ADDRESS_LOW        ;preload for fastest
        mov     bh,byte ptr StartOffset[bp] ; flipping once display
        mov     cl,START_ADDRESS_HIGH       ; enable is detected
        mov     ch,byte ptr StartOffset+1[bp]
        mov     dx,INPUT_STATUS_1
WaitDE:
        in      al,dx
        test    al,01h
        jnz     WaitDE  ;display enable is active low (0 = active)
; Set the start offset in display memory of the page to display.
        mov     dx,CRTC_INDEX
        mov     ax,bx
        out     dx,ax   ;start address low
        mov     ax,cx
        out     dx,ax   ;start address high
; Now wait for vertical sync, so the other page will be invisible when
; we start drawing to it.
        mov     dx,INPUT_STATUS_1
WaitVS:
        in      al,dx
        test    al,08h
        jz      WaitVS  ;vertical sync is active high (1 = active)
        pop     bp      ;restore caller's stack frame
        ret
_ShowPage       endp
        end
```

[LISTING SEVEN]

```
;;;<change this in Listing 3 from the August, 1991 column...>
;;;     mov     dx,SC_INDEX+1   ;point to Sequence Controller Data reg
;;;                             ; (SC Index still points to Map Mask)
;;;<...to this>
        mov     dx,SC_INDEX
        mov     al,MAP_MASK
        out     dx,al           ;point SC Index reg to Map Mask
        inc     dx              ;point to SC Data reg
```

Journal: Dr. Dobb's Journal Oct 1991 v16 n10 p155(6)

The virtues of affordable technology: the Sierra Hicolor DAC

Author: Abrash, Michael.
AttFile: GP-OCT91.ASC Source code listing.

Summary: Sierra Semiconductor Corp's Hicolor digital-to-analog converter (DAC) is easy to program and supports an 800x600, 32,768-color mode. The difference between the cost of the Hicolor DAC compared to a standard VGA DAC is less than $10 and some HiColor-based SuperVGA's are priced at less than $200. On the downside, the Hicolor DAC has some disadvantages. It needs twice as much memory for a given resolution as an equivalent CEG/DAC and graphics operations can take longer. The Hicolor DAC also does not perform gamma correction in hardware. It also does not include a built-in lookup table for correction of programmable gamma. However, the Hicolor DAC makes it possible for the first time in the VGA market to perform general antialiasing.

My, how quickly the PC world changes! Six months ago, I described the Edsun CEG/DAC as a triumph of inexpensive approximation. That chip was and is an ingenious bridge between SuperVGA and true color that requires no modifications to VGA chips or additional memory, yet achieves often-stunning results. Six months ago, the CEG/DAC was the only affordable path beyond SuperVGA.

Time and technology march on, and, in this case, technology has marched much the faster. I have on my desk a SuperVGA card, built around the Tseng Labs ET4000 VGA chip, 1 Mbyte of RAM, and the Sierra Semiconductor Hicolor DAC (digital-to-analog converter, the chip that converts pixel values from the VGA into analog signals for the monitor), that supports an 800x600, 32,768-color mode. The added cost of the Hicolor DAC over a standard VGA DAC (of which the Hi-color DAC is a fully compatible superset) to the board manufacturer is less than $10; I have already seen a Hicolor-based SuperVGA listed in Computer Shopper for under $200.

To those of us who remember buying IBM EGAs for $1000, there's a certain degree of unreality to the though of an 800x600 32K-color VGA for less than $200.

Understand, now, that I'm not talking about clever bitmap encoding or other tricky ways of boosting colore here. This is the real, 15-bpp, almost true-color McCoy, beautifully suited to imaging, antialiasing, and virtually any sort of high-color graphics you might imagine. The Hicolor DAC supports normal bitmaps that are just like 256-color bitmaps, except that each pixel is composed of 15 bits spread across 2 bytes. If you know how to program 800x600 256-color mode, you should have no trouble at all programming 800x600 32K-color mode; for the most part, just double the horizontal byte counts. (Lower-resolution 32K-color modes, such as 640x480, are available. No 1024x768 32K-color mode is supported, not due to any limitation of the Hicolor DAC, but because no VGA chip currently supports the 1.5 Mbyte of memory that mode requires. Expect that to change soon.) The 32K-color banking schemes are the same as in 256-color modes, except that there are half as many pixels in each bank. Even the complexities of the DAC's programmable palette go away in 32K-color mode, because there is no programmable palette.

And therein lies the strength of the Hicolor DAC: It's easy to program. Theoretically, the CEG/DAC can produce higher-color and more precise images using less display memory than the Hicolor DAC, because CEG color resolutions of 24-bpp and even higher are possible. Practically speaking, it's hard to write software -- especially real-time software -- that takes full advantage of the CEG/DAC's capabilities. On the other hand, it's very easy to extend existing 256-color SuperVGA code to support the Hicolor DAC, and although 32K colors is not the same as true color (24-bpp), it's close enough for most purposes, and astonishingly better than 256 colors. Digitized and rendered images look terrific on the Hicolor DAC, just as they do on the CEG/DAC -- and it's a lot easier and much faster to generate such images for the Hicolor DAC.

The Hicolor DAC has three disadvantages. First, it requires twice as much memory at a given resolution as does an equivalent 256-color or CEG/DAC mode. This is no longer a significant problem (apart from temporarily precluding a 1024x768 32K-color mode, as explained earlier); memory is cheap, and 1 Mbyte is becoming standard on SuperVGAs. Secondly, graphics operations can take considerably longer, simply because there are twice as many bytes of display memory to be dealt with; however, the latest generation of SuperVGAs provides for such fast memory access that 32K-color software will probably run faster than 256-color software did on the first generation of SuperVGAs. Finally, the Hicolor DAC neither performs gamma correction in hardware nor provides a built-in look-up tables to allow programmable gamma correction.

To refresh your memory, gamma correction is the process of compensating for the nonlinear response of pixel brightness to input voltage. A pixel with a green value of 60 is much more than twice as bright as a pixel of value 30. The Hicolor DAC's lack of built-in gamma correction puts the burden on software to perform the correction so that antialising will work properly, and images such as digitized photographs will display with the proper brightness. Software gamma correction is possible, but it's a time-consuming nuisance; it also decreases the effective color resolution of the Hicolor DAC for bright colors, because the bright colors supported by the Hicolor DAC are spaced relatively farther apart than the dim colors.

The lack of gamma correction is, however, a manageable annoyance. On balance, the Hicolor DAC is true to its heritage; a logical, inexpensive, and painless extension of SuperVGA. The obvious next steps are 1024x768 in 32K colors, and 800x600 with 24 bpp; heck, 4 Mbytes of display memory (eight 4-Mbit RAMs) would be enough for 1024x768 24-bpp with room to spare. In short, the Hicolor DAC appears to be squarely in the mainstream of VGA evolution. (Note that although most of the first generation of Hicolor boards are built around the ET4000, which has quietly and for good reason became the preeminent SuperVGA chip, the Hicolor DAC works with other VGA chips and will surely appear on SuperVGAs of all sorts in the near future.)

Does that mean that the Hicolor DAC will become a standard? Beats me. I'm out of the forecasting business; the world changes too fast. The CEG/DAC has a head start and is showing up in a number of systems, and who knows what else is in the pipeline? Still, programmers love the Hicolor DAC, and I would be astonished if there were not an installed base of at least 100,000 by the end of the year. Draw your own conclusions; but me, I can't wait to do some antialiased drawing on the Hicolor DAC (and I will, in this column, next month).

If the CEG/DAC is a triumph of inexpensive approxumation, the Hicolor DAC is a masterpiece of affordable technology. I'd have to call a 1-Mbyte Hicolor SuperVGA for around $200 the ultimate in graphics cost effectiveness at this moment -- but don't expect it to hold that title for more than six months. Things change fast in this industry; $200 true-color in a year, anyone?

Polygon Antialiasing

To my mind, the best thing about the Hicolor DAC is that, for the first time in the VGA market, it makes fast, general antialiasing possible -- and the readers of this column will soon see the fruits of that. You see, what I've been working toward in this column is real-time 3-D, perspective drawing on a standard PC, without the assistance of any expensive hardware. The object model I'll be using is polygon-based; hence the fast polygon fill code I've presented. With mode X (320x240, 256 colors, undocumented by IBM), we now have a fast, square-pixel, page-flipped, 256 colors, undocumented by IBM), we now have a fast, square-pixel, page-flipped, 256-color mode, the best that standard VGA has to offer. In this mode, it's possible to do not only real-time, polygon-based perspective drawing and animation, but also relatively sophisticated effects such as lighting sources, smooth shading, and hidden surface removal. That's everything we need for real-time e-d -- but things could still be better.

Pixel are so large in mode X that polygons have very visibly jagged edges. These jaggies are the results of the aliasing of which I spoke back in April and May; that is, distortion of the true image that results from undersampling at the low pixel rate of the screen. Jaggies are a serious problem; the whole point of real-time 3-D is to create the illusion of reality, but jaggies quickly destroy that illusion, particularly when they're

crawling along the edges of moving objects. More frequent sampling (higher resolution) helps, but not as much as you'd think. What's really needed is the ability to blend colors arbitrarily within a single pixel, the better to reflect the nature of the true image in the neighborhood of that pixel -- that is, antialiasing. The pixels are still as large as ever, but with the colors blended properly, the eye processes the screen as a continuous image, rather than as a collection of discrete pixels, and perceives the image at much higher resolution than the display actually supports.

There are many ways to antialias, some of them fast enough for real-time processing, and they can work wonders in improving image appearance -- but they all require a high degree of freedom in choosing colors. For many sorts of graphics, 256 simultaneous colors is fine, but it's not enough for generally useful antialiasing (although we will shortly see an interesting sort of special-case antialiasing with 256 colors). Therefore, the one element lacking in my quest for affordable real-time 3-D has been good antialiasing.

No longer. The Hicolor DAC provides plenty of colors (although I sure do with the software didn't have to do gamma correction!), and makes them available in a way that allows for efficient programming. In a couple of months, I'm going to start presenting 3-D code; initially, this code will be for mode X, but you can expect to see some antialiasing code for the Hicolor DAC soon.

256 Color Antialiasing

Next month, I'll explain how the Hicolor DAC works -- how to detect it, how to initialize it, the pixel format, banking, and so on -- and then I'll demonstrate Hicolor antialiasing. This month, I'm going to demonstrate antialiasing on a standard VGA, partly to introduce the uncomplicated but effective antialiasing technique that I'll use the next month, partly so you can see the improvement that even quick and dirty antialiasing produces, and partly to show the sorts of interesting things that can be done with the palette in 256-color mode.

I'm going to draw a cube in perspective. For reference, Listing One (page 173) draws the cube in mode 13h (320x200, 256 colors) using the standard polygon fill routine that I developed back in February and March. No, the perspective calculations aren't performed in Listing One; I just got the polygon vertices out of 3-D software that I'm developing and hardwired them into Listing One. Never fear, though; we'll get to true 3-D soon enough.

Listing One draws a serviceable cube, but the edges of the cube are very jagged. Imagine the cube spinning, and the jaggies rippling along its edges, and you'll see the full dimensions of the problem.

Listing Two (page 173) and Three (page 173) together draw the same cube, but with simple, unweighted antialiasing. The results are much better than Listing One; there's no question in my mind as to which cube I'd rather see in my graphics software.

The antialiasing technique used in Listing Two is straightforward. Each polygon is scanned out in the usual way, but at twice the screen's resolution both horizontally and vertically (which I'll call "double-resolution," although it produces four times as many pixels), with the double-resolution pixels drawn to a memory buffer, rather than directly to the screen. Then, after all the polygons have been drawn to the memory buffer, a second pass is performed; this pass looks at the colors stored in each set of four double-resolution pixels, and draws to the screen a single pixel that reflects the colors and intensities of the four double-resolution pixels that make it up, as shown in Figure 1. In other words, Listing Two temporarily draws the polygons at double resolution, then uses the extra information from the double-resolution bitmap to generate an image with an effective resolution considerably higher than the screen's actual 320x200 capabilities.

Two interesting tricks are employed in Listing Two. First, it would be best from the standpoint of speed, if the entire screen could be drawn to the double-resolution intermediate buffer in a single pass. Unfortunately, a buffer capable of holding one full 640x400 screen would be 64,000 or more bytes in size -- too much memory for most programs to spare. Consequently, Listing Two instead scans out the image just two double-resolution scan lines

(corresponding to one screen scan line) at a time. That is, the entire image is scanned once for every two double-resolution scan lines, and all information not concerning the two lines of current interest is thrown away. This banding is implemented in Listing Three, which accepts a full list of scan lines to draw, but actually draws only those lines within the current scan line band. Listing Three also draws to the intermediate buffer, rather than to the screen.

The polygon-scanning code from February was hard-wired to call the function DrawHorizontalLineList, which drew to the display; this is the polygon-drawing code called by Listing One. That was fine so long as there was only one possible drawing target, but now we have two possible targets -- the display (for nonantialiased drawing), and the intermediate buffer (for antialiased drawing). It's desirable to be able to mix the two, even within a single screen, because antialiased drawing looks better but nonantialiased is faster. Consequently, I have modified Listing One from February -- the function FillConvexPolygon -- to create FillCnvxPolyDrvr, which is the same as FillConvexPolygon, except that it accepts as a parameter the name of the function to be used to draw the scanned-out polygon. FillCnvxPolyDrvr is so similar to FillConvexPolygon that it's not worth taking up printed space to show it in its entirety; Listing Four (page 174) shows the differences between the two; and the new module will be available in its entirety as part of the code from this issue, under the name FILCNVXD.C.

The second interesting trick in Listing Two is the way in which the palette is stacked to allow unweighted antialiasing. Listing Two arranges the palette so that rather than 256 independent colors, we'll work with four-way combinations within each pixel of three independent colors (red, green, and blue), with each pixel accurately reflecting the intensities of each of the four color components that it contains. This allows fast and easy mapping from four double-resolution pixels to the single screen pixel to which they correspond. Figure 2 illustrates the mapping of subpixels (double-resolution pixels) through the palette to screen pixels. This palette organization converts mode 13h from a 256-color mode to a four-color antialiasing mode.

It's worth noting that many palette registers are set to identical values by Listin Two, because the values of sub-pixels matter, arrangements of these values do not. For example, the pixel values 0x01, 0x04, 0x10, and 0x40 all map to 25 percent blue. By using a table look-up to map sets of four double-resolution pixels to screen pixel values, more than half the palette could be freed up for drawing with other colors.

Unweighted Antialiasing: How Good?

Is the antialiasing used in Listing Two the finest possible antialiasing technique? It is not. It is an unweighted antialiasing technique, meaning that no accounting is made for how close to the center of a pixel a polygon edge might be. The edges are also biased a half-pixel or so in some cases, so registration with the underlying image isn't perfect. Nonetheless, the technique used in Listing Two produces attractive results, which is what really matters; all screen displays are approximations, and unweighted antialiasing is certainly good enough for PC animation applications. Unweighted antialiasing can also support good performance, although this is not the case in Listing Two and Three, where I have opted for clarity rather than performance. Increasing the number of lines drawn on each pass, or reducing the area processed to the smallest possible bounding rectangle would help improve performance, as, of course, would the use of assembly language. If there's room, I'll demonstrate some of these techniques next month.

For further information on antialiasing, you might check out the standard reference: Computer Graphics, by Foley and van Dam. Michael Covington's "Smooth Views," in the May, 1990 Byte, provides a short but meaty discussion of unweighted line antialiasing.

As relatively good as it looks, Listing Two is still watered-down antialiasing, even of the unweighted variety. For all our clever palette stacking, we have only five levels of each color component available; that's a far cry from the 32 levels of the Hicolor DAC, or the 256 levels of true color. The limiations of 256-color modes, even with the palette, are showing through.

Nexth month, 15-bpp antialiasing.

The Mode X Mode Set Bug, Revisited

Two months back, I added a last-minute note to this column describing a fix to the mode X mode set code that I presented in the July column. I'd like to describe how this bug slipped past me, as an illustration of why it's so difficult to write flawless software nowadays. The key is this: The PC world is so huge and diverse that it's a sure thing that someone, somewhere, will eventually get clobbered by even the most innocuous bug -- a bug that you might well not have found if you had spent the rest of your life doing nothing but beta testing. It's like the thought that 100 monkeys, typing endlessly, would eventually write the complete works of Shakespeare; there are 50,000,000 monkeys out there banging on keyboards and mousing around, and they will inevitably find any holes you leave in your software.

In writing the mode X mode set code, I started by modifying known-good code. I tried the final version of the code on both of my computers with five different VGAs, and I had other people test it out on their systems. In short, I put the code through all the hoops I had available, and then I sent it out to be beaten on by 100,000 DDJ readers. It took all of one day for someone to find a bug.

The code I started with used the VGA's 28-MHz clock. Mode X should have used the 25-MHz clock, a simple matter of setting bit 2 of the Miscellaneous Output register (3C2h) to 0 instead of 1.

Alas, I neglected to change that single bit, so frames were drawn at a faster rate than they should have been; however, both of my monitors are multifrequency types, and they automatically compensated for the faster frame rate. Consequently, my clock-selection bug was invisible and innocuous -- until all those monkeys started banging on it.

IBM makes only fixed-frequency VGA monitors, which require very specific frame rates; if they don't get what you've told them to expect, the image rolls -- and that's what the July mode X mode set code did on fixed-frequency monitors. The corrected version, shown in Listing Five (page 174), selects the 25-MHz clock, and works just fine on fixed-frequency monitors.

Why didn't I catch this bug? Neither I nor a single one of my testers had a fixed-frequency monitor! This nicely illustrates how difficult it is these days to test code in all the PC-compatible environments in which it might run. The problem is particularly severe for small developers, who can't afford to buy every model from every manufacturer; just imagine trying to test network-aware software in all possible configurations.

When people ask why software isn't bulletproof; why it crashes or doesn't coexist with certain programs; why PC clones aren't always compatible; why, in short, the myriad of irritations of using a PC exist -- this is a big part of the reason. That's just the price we pay for unfettered creativity and vast choice in the PC market.

Unfettered for the moment; but consider AT&T's patent on backing store, the "esoteric" idea of storing an obscured area of a window in a buffer so as to be able to redraw it quickly. It took me all of ten minutes to independently invent that one five years ago. Better yet, check out the letters to the editor in the July Programmer's Journal, about which I will say no more because it sets my teeth on edge. We'd better hope that no one patents "patterned tactile-pressure information input," that is, typing. Trust 50,000,000 monkeys to come up with a system as ridiculous as this.

- - - -

```
/* Draws the polygon described by the point list PointList in color
   Color with all vertices offset by (X,Y) */
#define DRAW_POLYGON(PointList,Color,X,Y)                    \
   Polygon.Length = sizeof(PointList)/sizeof(struct Point); \
   Polygon.PointPtr = PointList;                             \
   FillConvexPolygon(&Polygon, Color, X, Y);

void main(void);
extern int FillConvexPolygon(struct PointListHeader *, int, int, int);

/* Palette RGB settings to load the first four palette locations with
   black, pure blue, pure green, and pure red */
static char Palette[4*3] = {0, 0, 0,  0, 0, 63,  0, 63, 0,  63, 0, 0};

void main()
{
   struct PointListHeader Polygon;
   static struct Point Face0[] =
         {{198,138},{211,89},{169,44},{144,89}};
   static struct Point Face1[] =
         {{153,150},{198,138},{144,89},{105,113}};
   static struct Point Face2[] =
         {{169,44},{133,73},{105,113},{144,89}};
   union REGS regset;
   struct SREGS sregs;

   /* Set the display to VGA mode 13h, 320x200 256-color mode */
   regset.x.ax = 0x0013; int86(0x10, &regset, &regset);

   /* Set color 0 to black, color 1 to pure blue, color 2 to pure
      green, and color 3 to pure red */
   regset.x.ax = 0x1012;   /* load palette block BIOS function */
   regset.x.bx = 0;        /* start with palette register 0 */
   regset.x.cx = 4;        /* set four palette registers */
   regset.x.dx = (unsigned int) Palette;
   segread(&sregs);
   sregs.es = sregs.ds;       /* point ES:DX to Palette */
   int86x(0x10, &regset, &regset, &sregs);

   /* Draw the cube */
   DRAW_POLYGON(Face0, 3, 0, 0);
   DRAW_POLYGON(Face1, 2, 0, 0);
   DRAW_POLYGON(Face2, 1, 0, 0);
   getch();    /* wait for a keypress */

   /* Return to text mode and exit */
   regset.x.ax = 0x0003;   /* AL = 3 selects 80x25 text mode */
   int86(0x10, &regset, &regset);
}
```

[LISTING TWO]

```
/* Demonstrates unweighted antialiased drawing in 256 color mode.
   Tested with Borland C++ 2.0 in C mode in the small model. */

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

/* Draws the polygon described by the point list PointList in color
   Color, with all vertices offset by (X,Y), to ScanLineBuffer, at
   double horizontal and vertical resolution */
#define DRAW_POLYGON_DOUBLE_RES(PointList,Color,x,y)         \
   Polygon.Length = sizeof(PointList)/sizeof(struct Point); \
   Polygon.PointPtr = PointTemp;                             \
   /* Double all vertical & horizontal coordinates */        \
   for (k=0; k<sizeof(PointList)/sizeof(struct Point); k++) { \
```

```
      PointTemp[k].X = PointList[k].X * 2;                  \
      PointTemp[k].Y = PointList[k].Y * 2;                  \
   }                                                        \
   FillCnvxPolyDrvr(&Polygon, Color, x, y, DrawBandedList);

#define SCREEN_WIDTH 320
#define SCREEN_HEIGHT 200
#define SCREEN_SEGMENT 0xA000
#define SCAN_BAND_WIDTH (SCREEN_WIDTH*2)  /* # of double-res pixels
                                             across scan band */
#define BUFFER_SIZE  (SCREEN_WIDTH*2*2)   /* enough space for one scan
                                             line scanned out at double
                                             resolution horz and vert */
void main(void);
void DrawPixel(int, int, char);
int ColorComponent(int, int);
extern int FillCnvxPolyDrvr(struct PointListHeader *, int, int, int,
   void (*)());
extern void DrawBandedList(struct HLineList *, int);

/* Pointer to buffer in which double-res scanned data will reside */
unsigned char *ScanLineBuffer;
int ScanBandStart, ScanBandEnd;  /* top & bottom of each double-res
                                    band we'll draw to ScanLineBuffer */
int ScanBandWidth = SCAN_BAND_WIDTH;  /* # pixels across scan band */
static char Palette[256*3];

void main()
{
   int i, j, k;
   struct PointListHeader Polygon;
   struct Point PointTemp[4];
   static struct Point Face0[] =
         {{198,138},{211,89},{169,44},{144,89}};
   static struct Point Face1[] =
         {{153,150},{198,138},{144,89},{105,113}};
   static struct Point Face2[] =
         {{169,44},{133,73},{105,113},{144,89}};
   unsigned char Megapixel;
   union REGS regset;
   struct SREGS sregs;

   if ((ScanLineBuffer = malloc(BUFFER_SIZE)) == NULL) {
      printf("Couldn't get memory\n");
      exit(0);
   }

   /* Set the display to VGA mode 13h, 320x200 256-color mode */
   regset.x.ax = 0x0013; int86(0x10, &regset, &regset);

   /* Stack the palette for the desired megapixel effect, with each
      2-bit field representing 1 of 4 double-res pixels in one of four
      colors */
   for (i=0; i<256; i++) {
      Palette[i*3] = ColorComponent(i, 3);   /* red component */
      Palette[i*3+1] = ColorComponent(i, 2); /* green component */
      Palette[i*3+2] = ColorComponent(i, 1); /* blue component */
   }
   regset.x.ax = 0x1012;   /* load palette block BIOS function */
   regset.x.bx = 0;        /* start with palette register 0 */
   regset.x.cx = 256;      /* set all 256 palette registers */
   regset.x.dx = (unsigned int) Palette;
   segread(&sregs);
   sregs.es = sregs.ds;       /* point ES:DX to Palette */
   int86x(0x10, &regset, &regset, &sregs);

   /* Scan out the polygons at double resolution one screen scan line
      at a time (two double-res scan lines at a time) */
   for (i=0; i<SCREEN_HEIGHT; i++) {
      /* Set the band dimensions for this pass */
      ScanBandEnd = (ScanBandStart = i*2) + 1;
      /* Clear the drawing buffer */
      memset(ScanLineBuffer, 0, BUFFER_SIZE);
      /* Draw the current band of the cube to the scan line buffer */
```

```
      DRAW_POLYGON_DOUBLE_RES(Face0, 3, 0, 0);
      DRAW_POLYGON_DOUBLE_RES(Face1, 2, 0, 0);
      DRAW_POLYGON_DOUBLE_RES(Face2, 1, 0, 0);

      /* Coalesce the double-res pixels into normal screen pixels
         and draw them */
      for (j=0; j<SCREEN_WIDTH; j++) {
         Megapixel = (ScanLineBuffer[j*2] << 6) +
                     (ScanLineBuffer[j*2+1] << 4) +
                     (ScanLineBuffer[j*2+SCAN_BAND_WIDTH] << 2) +
                     (ScanLineBuffer[j*2+SCAN_BAND_WIDTH+1]);
         DrawPixel(j, i, Megapixel);
      }
   }

   getch();    /* wait for a keypress */

   /* Return to text mode and exit */
   regset.x.ax = 0x0003;   /* AL = 3 selects 80x25 text mode */
   int86(0x10, &regset, &regset);
}

/* Draws a pixel of color Color at (X,Y) in mode 13h */
void DrawPixel(int X, int Y, char Color)
{
   char far *ScreenPtr;

   ScreenPtr = (char far *)MK_FP(SCREEN_SEGMENT, Y*SCREEN_WIDTH+X);
   *ScreenPtr = Color;
}

/* Returns the gamma-corrected value representing the number of
   double-res pixels containing the specified color component in a
   megapixel with the specified value */
int ColorComponent(int Value, int Component)
{
   /* Palette settings for 0%, 25%, 50%, 75%, and 100% brightness,
      assuming a gamma value of 2.3 */
   static int GammaTable[] = {0, 34, 47, 56, 63};
   int i;

   /* Add up the number of double-res pixels of the specified color
      in a megapixel of this value */
   i = (((Value & 0x03) == Component) ? 1 : 0) +
       ((((Value >> 2) & 0x03) == Component) ? 1 : 0) +
       ((((Value >> 4) & 0x03) == Component) ? 1 : 0) +
       ((((Value >> 6) & 0x03) == Component) ? 1 : 0);
   /* Look up brightness of the specified color component in a
      megapixel of this value */
   return GammaTable[i];
}
```

[LISTING THREE]

```
/* Draws pixels from the list of horizontal lines passed in; drawing
   takes place only for scan lines between ScanBandStart and
   ScanBandEnd, inclusive; drawing goes to ScanLineBuffer, with
   the scan line at ScanBandStart mapping to the first scan line in
   ScanLineBuffer. Intended for use in unweighted antialiasing,
   whereby a polygon is scanned out into a buffer at a multiple of the
   screen's resolution, and then the scanned-out info in the buffer is
   grouped into megapixels that are mapped to the closest
   approximation the screen supports and drawn. Tested with Borland
   C++ 2.0 in C mode in the small model */

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

extern unsigned char *ScanLineBuffer;  /* drawing goes here */
```

```
extern int ScanBandStart, ScanBandEnd; /* limits of band to draw */
extern int ScanBandWidth;  /* # of pixels across scan band */

void DrawBandedList(struct HLineList * HLineListPtr, int Color)
{
   struct HLine *HLinePtr;
   int Length, Width, YStart = HLineListPtr->YStart;
   unsigned char *BufferPtr;

   /* Done if fully off the bottom or top of the band */
   if (YStart > ScanBandEnd) return;
   Length = HLineListPtr->Length;
   if ((YStart + Length) <= ScanBandStart) return;

   /* Point to the XStart/XEnd descriptor for the first (top)
      horizontal line */
   HLinePtr = HLineListPtr->HLinePtr;

   /* Confine drawing to the specified band */
   if (YStart < ScanBandStart) {
      /* Skip ahead to the start of the band */
      Length -= ScanBandStart - YStart;
      HLinePtr += ScanBandStart - YStart;
      YStart = ScanBandStart;
   }
   if (Length > (ScanBandEnd - YStart + 1))
      Length = ScanBandEnd - YStart + 1;

   /* Point to the start of the first scan line on which to draw */
   BufferPtr = ScanLineBuffer + (YStart - ScanBandStart) *
         ScanBandWidth;

   /* Draw each horizontal line within the band in turn, starting with
      the top one and advancing one line each time */
   while (Length-- > 0) {
      /* Draw the whole horizontal line if it has a positive width */
      if ((Width = HLinePtr->XEnd - HLinePtr->XStart + 1) > 0)
         memset(BufferPtr + HLinePtr->XStart, Color, Width);
      HLinePtr++;                /* point to next scan line X info */
      BufferPtr += ScanBandWidth; /* point to next scan line start */
   }
}
```

[LISTING FOUR]

```
/* The changes required to convert the function FillConvexPolygon,
   from Listing 1 in the Feb, 1991, column, into FillCnvxPolyDrvr.
   FillConvexPolygon was hardwired to call DrawHorizontalLineList to
   draw to the display; FillCnvxPolyDrvr is more flexible because it
   draws via the driver passed in as the DrawListFunc parameter */

/****** Delete this line ******/
extern void DrawHorizontalLineList(struct HLineList *, int);

/****** Change this... ******/
int FillConvexPolygon(struct PointListHeader * VertexList, int Color,
      int XOffset, int YOffset)
/****** ...to this ******/
int FillCnvxPolyDrvr(struct PointListHeader * VertexList, int Color,
      int XOffset, int YOffset, void (*DrawListFunc)())

/****** Change this... ******/
   DrawHorizontalLineList(&WorkingHLineList, Color);
/****** ...to this ******/
   (*DrawListFunc)(&WorkingHLineList, Color);
```

[LISTING FIVE]

```
; Mode X (320x240, 256 colors) mode set routine. Works on all VGAs.
; ***************************************************************
; * Revised 6/19/91 to select correct clock; fixes vertical roll *
; * problems on fixed-frequency (IBM 851X-type) monitors.        *
; ***************************************************************
; C near-callable as:    void Set320x240Mode(void);
; Tested with TASM 2.0.
; Modified from public-domain mode set code by John Bridges.

SC_INDEX equ    03c4h   ;Sequence Controller Index
CRTC_INDEX equ  03d4h   ;CRT Controller Index
MISC_OUTPUT equ 03c2h   ;Miscellaneous Output register
SCREEN_SEG equ  0a000h  ;segment of display memory in mode X

        .model  small
        .data
; Index/data pairs for CRT Controller registers that differ between
; mode 13h and mode X.
CRTParms label  word
        dw      00d06h  ;vertical total
        dw      03e07h  ;overflow (bit 8 of vertical counts)
        dw      04109h  ;cell height (2 to double-scan)
        dw      0ea10h  ;v sync start
        dw      0ac11h  ;v sync end and protect cr0-cr7
        dw      0df12h  ;vertical displayed
        dw      00014h  ;turn off dword mode
        dw      0e715h  ;v blank start
        dw      00616h  ;v blank end
        dw      0e317h  ;turn on byte mode
CRT_PARM_LENGTH equ     (($-CRTParms)/2)

        .code
        public  _Set320x240Mode
_Set320x240Mode proc    near
        push    bp      ;preserve caller's stack frame
        push    si      ;preserve C register vars
        push    di      ; (don't count on BIOS preserving anything)

        mov     ax,13h  ;let the BIOS set standard 256-color
        int     10h     ; mode (320x200 linear)

        mov     dx,SC_INDEX
        mov     ax,0604h
        out     dx,ax   ;disable chain4 mode
        mov     ax,0100h
        out     dx,ax   ;synchronous reset while setting Misc Output
                        ; for safety, even though clock unchanged
        mov     dx,MISC_OUTPUT
        mov     al,0e3h
        out     dx,al   ;select 25 MHz dot clock & 60 Hz scanning rate

        mov     dx,SC_INDEX
        mov     ax,0300h
        out     dx,ax   ;undo reset (restart sequencer)

        mov     dx,CRTC_INDEX ;reprogram the CRT Controller
        mov     al,11h  ;VSync End reg contains register write
        out     dx,al   ; protect bit
        inc     dx      ;CRT Controller Data register
        in      al,dx   ;get current VSync End register setting
        and     al,7fh  ;remove write protect on various
        out     dx,al   ; CRTC registers
        dec     dx      ;CRT Controller Index
        cld
        mov     si,offset CRTParms ;point to CRT parameter table
        mov     cx,CRT_PARM_LENGTH ;# of table entries
SetCRTParmsLoop:
        lodsw           ;get the next CRT Index/Data pair
        out     dx,ax   ;set the next CRT Index/Data pair
        loop    SetCRTParmsLoop
```

```
        mov     dx,SC_INDEX
        mov     ax,0f02h
        out     dx,ax   ;enable writes to all four planes
        mov     ax,SCREEN_SEG ;now clear all display memory, 8 pixels
        mov     es,ax         ; at a time
        sub     di,di   ;point ES:DI to display memory
        sub     ax,ax   ;clear to zero-value pixels
        mov     cx,8000h ;# of words in display memory
        rep     stosw   ;clear all of display memory

        pop     di      ;restore C register vars
        pop     si
        pop     bp      ;restore caller's stack frame
        ret
_Set320x240Mode endp
        end
```

Journal: Dr. Dobb's Journal Nov 1991 v16 n11 p123(7)

Antialiasing with the Sierra Hicolor DAC

Author: Abrash, Michael.
AttFile: GP-NOV91.ASC Source code listing.

Abstract: Sierra Semiconductor's Hicolor digital to analog converter provides classic, high quality antialiasing. Pixels in the Hicolor are stored in one word of display memory; each pixel on the screen corresponds to a group of sub-pixels in the memory buffer. Sub-pixels are averaged and the screen pixel is set to that level. In antialiasing, the image draws to the memory buffer at a multiple of the resolution that the screen supports. A sub-pixel resolution of twice normal along each axe has visible jagged edges but is better than nonantialiased images. Blending colors into a smooth gradient across a boundary allows the human eye to identify the boundary with better precision than the resolution of the screen.

There's an Italian saying, the gist of which is, "It need not be true, so long as it's well said." This strikes close to the essential truth of antialiasing: The image need not be accurate, so long as it looks like it is. You don't go to the trouble of antialiasing in order to get a mathematically precise representation of an image; you do it so the amazing eye/brain integrating/pattern matching system will see what you want it to see.

This is a particularly relevant thought at the moment, for we're smack in the middle of discussing the Sierra Hicolor DAC, which makes classic, high-quality antialiasing, of the sort that Targa boards have offered for years, available at mass-market prices. To recap, the Hicolor DAC extends SuperVGA to provide selection among enough colors for serious rendering and antialiasing; 32,768 simultaneous colors, to be exact. Although the Hicolor DAC falls short of the 24-bpp true color standard, you aren't likely to find a 24-bpp adapter priced in the $200-300 range.

Last month, we looked at simple, unweighted antialiasing in the context of the VGA's standard 256-color mode, performing antialiasing between exactly four colors--red, green, blue, and black--with five semi-independent levels of each of the three primary colors available. This month, we'll start off by discussing the basic Hicolor programming model, then we'll do the same sort of antialiasing as last month--but this time with 32 fully independent levels of each primary color and resolutions up to 800x600, which makes quite a difference indeed.

A Brief Primer on the Sierra Hicolor DAC

The operation of the Hicolor DAC in 32K-color mode is remarkably simple. First, the VGA must be set to a 256-color mode with twice the desired horizontal resolution; for example, a 1600x 600 256-color mode would be selected if 800x600 32K-color mode were desired. Then, the Hicolor DAC is set to high-color mode via the command register; in high-color mode, the Hicolor DAC takes each pair of 256-color pixels, joins them together into one 16-bit pixel, converts the red, green, and blue components (described shortly) directly to proportional analog values (no palette involved), and sends them to the monitor.

Ther is a serious problem here, however: There is no standard way for an application to select a high-color mode. It's not enough to set up the Hicolor DAC; the VGA must also be set to the appropriate double-resolution 256-color mode, and the sequence for doing that--especially selecting high-speed clocks-varies from VGA to VGA. There is no VESA mode number for high-color modes; there is a VESA programming guideline for high-color modes, and I'm looking into it, but it's certainly not as simple as a mode number. In any case, the VESA interface isn't always available.

Consequently, high-color mode selection is adapter-dependent. Fortunately, most of the Hicolor-based boards are built around the Tseng Labs ET4000 VGA

chip (ATI is the only exception I know of, which is why I'm focusing on ET4000-based Hicolor boards in this column), and Tseng provides a BIOS interface for high-color modes. (There's no guarantee that manufacturers using the ET4000 will follow the Tseng interface, but I suspect they will, as it's the closest thing to a standard at the mooment.) Unfortunately, when I run the ET4000 BIOS function that reports whether a Hicolor DAC is present on my Toshiba portable without a Hicolor board installed, it hangs my system, so it's not a good idea to rely on the BIOS functions alone.

My solution, shown in Listing One (page 146), is to first check for a Hicolor DAC and an ET4000 at the hardware level; if both are present, I call the BIOS (which is presumably at least not hostile to the Tseng BIOS high-color extensions at this point) to check for the availability of Hicolor modes, and finally, if all has gone well, to set the desired high-color mode. This is probably overkill, but at least this way you get three kinds of chip ID code to mix and match as you wish.

Programming the Hicolor DAC

The pixel format of the Hicolor DAC is straightforward: Each pixel is stored in one word of display memory, with the lowest 5 bits forming the blue component, the next 5 bits forming the green component, the next 5 bits forming the red component, and bit 15 ignored, as shown in Figure 1. Pixels start at even addresses. The bits within a word are organized Intel style, with the byte at the even address containing bits 7-0 (blue and part of green), and the byte at the odd address containing bits 158 (red and the rest of green).

Pixels proceed linearly for the length of the bitmap; the organization is the same as 256-color mode, except that each pixel takes up one word, rather than one byte. As in SuperVGA 256-color modes, the bitmap is too long to be addressed in the 64K video memory window, so banking must be used; again, the banking is just like 256-color or banking, except that each bank contains only half as many pixels; 32,768, to be exact.

On the ET4000, the Segment Select register at 3CDh controls banking, as shown in Figure 2. There are 16 banks, each spanning 64 Kbytes of the total 1-Mbyte bitmap. Banks can be selected separately for read and write to facilitate scrolling and screen-to-screen copies, although we won't need that today. Simple enough, but there's a catch: broken rasters.

Banks are 64K in length. If each Hicolor scan line is 1600 bytes long, then 65,536/1600=40 raster lines (lines 0-39) fit in bank 0--with 1536 bytes of the next line (line 40), also in the first bank. The last 64 bytes of line 40 are in bank 1, as shown in Figure 3, so the line is split by the bank boundary; hence the term "broken raster." Broken rasters crop up for other bank crossings as well, and make Hicolor programming somewhat slower (the extent depends heavily on the quality of the code) and considerably more complicated.

Broken rasters are not unique to Hicolor modes; 800x600 and 640x480 256-color modes also normally have broken rasters. However, there's a clever workaround for broken rasters in 256-color modes: Stretch the bitmap width to 1K pixels, via the Row Offset register, so that banks split between raster lines (although this works for 800x600 only if the VGA has 1 Mbyte of memory).

Sad to say, stretching the bitmap width to 1K pixels doesn't work in Hicolor mode. There's not enough memory to do it at 800x600 (at least until 2-Mbyte VGAs appear), but that's not the problem at 640x480. The problem is that a Row Offset register setting of 256 would be required to stretch the bitmap width to 1K pixels--and the Row Offset register only goes up to 255. I'm sure that when the VGA was being designed, 255 seemed like plenty, but then, 640K once seemed like pie in the sky.

The upshot is that Hicolor programming unavoidably requires handling broken rasters. That's a nuisance, but a manageable one; next, we'll see polygon fill code that deals with broken rasters.

Non-antialiased Hicolor Drawing

Listing Two (page 146) draws a perspective cube in 640x480 32K color mode, with help from the initialization code in Listing One, the DrawPixel-based low-level polygon fill code in Listing Three (page 146), and the header file in Listing Seven, page 149. (FILCNVXD.C from last month and Listing Four from the March column are also required.) Not surprisingly, the cube drawn by Listing Two looks a lot like the non-antialiased cube drawn last month, but isn't as jagged because the resolution is now much higher. Nonetheless, jaggies are still quite prominent, and they remain clearly visible at 800x600. (I've used 640x480 mode so that the code will work on fixed-frequency monitors, but Listing Two can be altered for 800x600 mode simply by changing the parameter passed to SetHCMode and the value of Bitmap WidthInBytes.)

Listing Two doesn't run very fast when linked to Listing Three; I suspect you'll like the low-level polygon fill code in Listing Four (page 146) much better. This code handles broken rasters reasonably efficiently, by checking for them at the beginning of each scan line, then splitting up the fill and banking appropriately whenever a bank crossing is detected.

Simple Unweighted Antialiasing

As the saying goes, you can never be too rich, too thin, or have too many colors available. Personally, I only buy one of those three assertions: You really can't have too many colors. Listings Five (page 148) and Six (page 149), together with Listings One, Three, and Seven, FILCNVXD.C from last month, and Listing Four from March, show why. This program draws the same cube, but this time employing the simple, unweighted antialiasing we used last month--and taking advantage of the full color range of the Hicolor DAC. The results are excellent: On my venerable NEC MultiSync, at a viewing distance of one foot, all but two of the edges look absolutely smooth, with not the slightest hint of jaggies, and the two imperfect edges show only slight ripples. At two feet, the cube looks perfect. The difference between the non-antialiased and antialiased cubes is astounding, considering that we're working with the same resolution in both cases.

A quick review of the simple antialiasing used this month and last: The image is drawn to a memory buffer at a multiple of the resolution that the actual screen supports. Each pixel on the screen maps to a group of hi-res pixels (subpixels), arranged in a square, in the memory buffer. The colors of the subpixels in each square are averaged, and the corresponding screen pixel is set to the average subpixel color.

There's not enough memory to scan out the entire image at high resolution (about 50K is required just to scan out one raster line at 4X resolution!), so Listing Six scans out just those pixels that lie in a specified band. (Each band corresponds to a single raster line in Listing Five.) Note that Listing Six draws 32-bit pixels to the memory buffer; this is true color, plus an extra byte for flexibility. Consequently, Listing Six is a general-purpose tool, and can be used with any sort of adapter, so long as the main program knows how to convert from true color to adapter-specific pixels. Listing Five does this by calculating the average intensity in each subpixel group of each of the three primary colors, in the range 0-255, then looking up the gamma corrected equivalent color value for the Hicolor DAC, mapped into the range 0-31.

A quick look at the gamma-corrected color mapping table in Listing Five shows why hardware gamma correction is sorely missed in the Hicolor DAC. The brightest half of the color range--from half intensity to full intensity--is spanned by only 9 of the Hicolor DAC's 32 color values. That means that for brighter colors, the Hicolor DAC effectively has only half the color resolution that you'd expect from 5 bits per color gun, and the resolution is even worse at the highest intensities.

I'd like to take a moment to emphasize that although Listing Five works with only the three primary colors, it could just as easily work with the thousands of colors that can be produced as mixes of the three primaries; there are none of the limitations of 256-color mode, and no special tricks (such as biasing the palette according to color frequency) need be used. Inevitably, though, proportionately fewer intermediate blends are available and hence antialiasing becomes less precise when there is less contrast

between colors; you're not going to be able to do much antialiasing between a pixel with a green true color value of 250 and another with a value of 255. This is where the lack of gamma correction and the difference between 15-bpp and true color become apparent.

Notes on the Antialiasing Implementation

Listing Five features user-selectable subpixel resolution (the multiple of the screen resolution at which the image should be drawn into the memory buffer). A subpixel resolution of two times normal along both axes (2X) looks much better than nonantialiased drawing, but still has visible jaggies. Subpixel resolution of 4X looks terrific, as mentioned earlier. Higher subpixel resolutions are, practically speaking, reserved for 386 protected mode, because they would require a buffer larger than 64K to hold the high-res equivalent of a single scan line.

On the downside, Listing Five is very slow, even though the conversion process from true color pixels to Hicolor pixels is limited to the bounding rectangle for the cube being drawn, thereby saving the time that was wasted last month drawing the empty space around the cube. It could easily be sped up by, say, an order of magnitude, in a number of ways. First, you could implement an ASM function that's the equivalent of memset, but stores longs (dwords) rather than chars (bytes). In the absence of any such C library function, Listing Six uses a loop with a pointer to a long, hardly a recipe for high performance.

Listing Five could also be sped up by doing the screen pixel construction from each square of subpixels in assembly language using pointers rather than array look-ups. It would also help to organize the screen pixel drawing more as a variant rectangle fill, instead of going through DrawPixel every time, so that the screen pointer doesn't have to be recalculated from scratch and the bank doesn't need to be calculated and set for every pixel. Clipping each polygon to the band before rather than after scanning it out would speed things up, as would building an edge list for the polygons once, ahead of time, then advancing it incrementally to scan out each band, rather than doing one complete scan of each polygon for each band. Bigger bands would help; drawing the whole image to the memory buffer in one burst, then converting the entire image to Hicolor pixels in a single operation, would be ideal, but would require a ridiculous amount of memory. (Would you believe, 31 megs for one full 800x600 Hicolor screen at 4X resolution?)

Finally, to alter Listing Five for 800x 600 Hicolor mode, change the parameter passed to SetHCMode, the value of BitmapWidthInBytes, and the value of SCREEN WIDTH.

Further Thoughts on Antialiasing

The banded true color approach of Listings Five and Six is easily extended to other antialiasing approaches. For example, you could, if you wished, average together all the subpixels not within a square, but rather within a circle of radius sqrt(2.0)*ResolutionMultiplier/2 around each pixel center. This approach is a little more complicated, but it has one great virtue: An image will be antialiased identically, regardless of its rotation.

Why is the shape of the subpixed area that's collected into a screen pixel important, when the maximum resolution we can actually draw with is the resolution of the screen? I'll quote William vanRyper, from the graphics.disp/vga conference on BIX:

If you anti-alias an edge on the screen, and let the eye-brain pick the edge somewhere the gradien between the object color and the background, you can adjust the placement of that perceptual edge by altering the ramp of the gradient. If the number of intermediate values you can choose among is greater then the number of gradient pixels you set (across the edge), you can adjust the position of the perceptucal edge in increments of less than a pixel. This means you can locate the antialiased object to sub-pixel precesion.

In other words, by using blends of color in a smooth, consistent gradient across a boundary, you can get the eye to pick out the boundary location with

a precision that's greater than the resolution of the screen. This is, of course, part and parcel of the wonderful eye/brain magic that allows color to substitute for resolution and makes antialiasing worthwhile.

Given that we can draw images with perceived resolution higher than the screen, consistency in subpixel placement is very important. Unfortunately, our simple square antialiasing does not produce the same results (a consistent color gradient) for an image rotated 45 degrees as it does for an unrotated image--but antialiasing based on a circular subpixel area does. So the shape of the subpixel area used for antialiasing matters because if it's not symmetric in all directions, boundaries will appear to wiggle as images roate, destroying the image of reality that antialiased animation strives to create.

On the other hand, if you're drawing only static images, use a square subpixel area for antialiasing; it's fast, easy, and looks just fine in that context. As I said at the outset, we're not seeking mathematical perfection here, just a good-looking display for the purpose at hand. If it looks good, it is good

- -

GRAPHICS PROGRAMMING COLUMN
by Michael Abrash

[LISTING ONE]

```
/* Looks for a Sierra Hicolor DAC; if one is present, puts the VGA into the
specified Hicolor (32K color) mode. Relies on the Tseng Labs ET4000 BIOS and
hardware; probably will not work on adapters built around other VGA chips.
Returns 1 for success, 0 for failure; failure can result from no Hicolor DAC,
too little display memory, or lack of an ET4000. Tested with Borland C++ 2.
0 in C mode in the small model. */

#include <dos.h>
#define DAC_MASK  0x3C6 /* DAC pixel mask reg address, also Sierra
                           command reg address when enabled */
#define DAC_WADDR 0x3C8  /* DAC write address reg address */

/* Mode selections: 0x2D=640x350; 0x2E=640x480; 0x2F=640x400; 0x30=800x600 */
int SetHCMode(int Mode) {
   int i, Temp1, Temp2, Temp3;
   union REGS regset;

   /* See if a Sierra SC1148X Hicolor DAC is present, by trying to
   program and then read back the DAC's command register. (Shouldn't be
   necessary when using the BIOS Get DAC Type function, but the BIOS function
   locks up some computers, so it's safer to check the hardware first) */
   inp(DAC_WADDR); /* reset the Sierra command reg enable sequence */
   for (i=0; i<4; i++) inp(DAC_MASK); /* enable command reg access */
   outp(DAC_MASK, 0x00); /* set command reg (if present) to 0x00, and
                            reset command reg enable sequence */
   outp(DAC_MASK, 0xFF); /* command reg access no longer enabled;
                            set pixel mask register to 0xFF */
   for (i=0; i<4; i++) inp(DAC_MASK); /* enable command reg access */
   /* If this is a Hicolor DAC, we should read back the 0 in the
      command reg; otherwise we get the 0xFF in the pixel mask reg */
   i = inp(DAC_MASK); inp(DAC_WADDR); /* reset enable sequence */
   if (i == 0xFF) return(0);

   /* Check for a Tseng Labs ET4000 by poking unique regs, (assumes
      VGA configured for color, w/CRTC addressing at 3D4/5) */
   outp(0x3BF, 3); outp(0x3D8, 0xA0);  /* unlock extended registers */
   /* Try toggling AC R16 bit 4 and seeing if it takes */
   inp(0x3DA); outp(0x3C0, 0x16 | 0x20);
   outp(0x3C0, ((Temp1 = inp(0x3C1)) | 0x10)); Temp2 = inp(0x3C1);
   outp(0x3C0, 0x16 | 0x20); outp(0x3C0, (inp(0x3C1) & ~0x10));
   Temp3 = inp(0x3C1); outp(0x3C0, 0x16 | 0x20);
   outp(0x3C0, Temp1);  /* restore original AC R16 setting */
   /* See if the bit toggled; if so, it's an ET3000 or ET4000 */
   if ((Temp3 & 0x10) || !(Temp2 & 0x10)) return(0);
   outp(0x3D4, 0x33); Temp1 = inp(0x3D5); /* get CRTC R33 setting */
   outp(0x3D5, 0x0A); Temp2 = inp(0x3D5); /* try writing to CRTC */
```

```
   outp(0x3D5, 0x05); Temp3 = inp(0x3D5); /*  R33 */
   outp(0x3D5, Temp1);  /* restore original CRTC R33 setting */
   /* If the register was writable, it's an ET4000 */
   if ((Temp3 != 0x05) || (Temp2 != 0x0A)) return(0);

   /* See if a Sierra SC1148X Hicolor DAC is present by querying the
      (presumably) ET4000-compatible BIOS. Not really necessary after
      the hardware check above, but generally more useful; in the
      future it will return information about other high-color DACs */
   regset.x.ax = 0x10F1;   /* Get DAC Type BIOS function # */
   int86(0x10, &regset, &regset); /* ask BIOS for the DAC type */
   if (regset.x.ax != 0x0010) return(0); /* function not supported */
   switch (regset.h.bl) {
      case 0:  return(0);  /* normal DAC (non-Hicolor) */
      case 1:  break;      /* Sierra SC1148X 15-bpp Hicolor DAC */
      default: return(0);  /* other high-color DAC */
   }

   /* Set Hicolor mode */
   regset.x.ax = 0x10F0;   /* Set High-Color Mode BIOS function # */
   regset.h.bl = Mode;     /* desired resolution */
   int86(0x10, &regset, &regset); /* have BIOS enable Hicolor mode */
   return (regset.x.ax == 0x0010); /* 1 for success, 0 for failure */
}
```

[LISTING TWO]

```
/* Demonstrates non-antialiased drawing in 640x480 Hicolor (32K color) mode on
an ET4000-based SuperVGA with a Sierra Hicolor DAC installed. Tested with
Borland C++ 2.0 in C mode in the small model. */

#include <conio.h>
#include <dos.h>
#include "polygon.h"
/* Draws the polygon described by the point list PointList in color
   Color, with all vertices offset by (x,y) */
#define DRAW_POLYGON(PointList,Color,x,y) {                 \
   Polygon.Length = sizeof(PointList)/sizeof(struct Point); \
   Polygon.PointPtr = PointList;                            \
   FillCnvxPolyDrvr(&Polygon, Color, x, y, DrawHCLineList);}

void main(void);
extern int SetHCMode(int);
extern int FillCnvxPolyDrvr(struct PointListHeader *, int, int, int,
   void (*)());
extern void DrawHCLineList(struct HLineList *, int);
int BitmapWidthInBytes = 640*2; /* # of bytes per raster line */

void main()
{
   struct PointListHeader Polygon;
   static struct Point Face0[] = {{396,276},{422,178},{338,88},{288,178}};
   static struct Point Face1[] = {{306,300},{396,276},{288,178},{210,226}};
   static struct Point Face2[] = {{338,88},{266,146},{210,226},{288,178}};
   union REGS regset;

   /* Attempt to enable 640x480 Hicolor mode */
   if (SetHCMode(0x2E) == 0)
      { printf("No Hicolor DAC detected\n"); exit(0); };

   /* Draw the cube */
   DRAW_POLYGON(Face0, 0x1F, 0, 0);       /* full-intensity blue */
   DRAW_POLYGON(Face1, 0x1F << 5, 0, 0);  /* full-intensity green */
   DRAW_POLYGON(Face2, 0x1F << 10, 0, 0); /* full-intensity red */
   getch();    /* wait for a keypress */

   /* Return to text mode and exit */
   regset.x.ax = 0x0003;   /* AL = 3 selects 80x25 text mode */
   int86(0x10, &regset, &regset);
}
```

[LISTING THREE]

```
/* Draws all pixels in the list of horizontal lines passed in, in Hicolor
(32K color) mode on an ET4000-based SuperVGA. Uses a slow pixel-by-pixel
approach. Tested with Borland C++ 2.0 in C mode in the small model. */

#include <dos.h>
#include "polygon.h"
#define SCREEN_SEGMENT     0xA000
#define GC_SEGMENT_SELECT  0x3CD

void DrawPixel(int, int, int);
extern int BitmapWidthInBytes; /* # of pixels per line */

void DrawHCLineList(struct HLineList * HLineListPtr,
      int Color)
{
   struct HLine *HLinePtr;
   int Y, X;

   /* Point to XStart/XEnd descriptor for the first (top) horizontal line */
   HLinePtr = HLineListPtr->HLinePtr;
   /* Draw each horizontal line in turn, starting with the top one and
      advancing one line each time */
   for (Y = HLineListPtr->YStart; Y < (HLineListPtr->YStart +
         HLineListPtr->Length); Y++, HLinePtr++) {
      /* Draw each pixel in the current horizontal line in turn,
         starting with the leftmost one */
      for (X = HLinePtr->XStart; X <= HLinePtr->XEnd; X++)
         DrawPixel(X, Y, Color);
   }
}

/* Draws the pixel at (X, Y) in color Color in Hicolor mode on an
   ET4000-based SuperVGA */
void DrawPixel(int X, int Y, int Color) {
   unsigned int far *ScreenPtr, Bank;
   unsigned long BitmapAddress;

   /* Full bitmap address of pixel, as measured from address 0 to
      address 0xFFFFF. (X << 1) because pixels are 2 bytes in size */
   BitmapAddress = (unsigned long) Y * BitmapWidthInBytes + (X << 1);
   /* Map in the proper bank. Bank # is upper word of bitmap addr */
   Bank = *(((unsigned int *)&BitmapAddress) + 1);
   /* Upper nibble is read bank #, lower nibble is write bank # */
   outp(GC_SEGMENT_SELECT, (Bank << 4) | Bank);
   /* Draw into the bank */
   FP_SEG(ScreenPtr) = SCREEN_SEGMENT;
   FP_OFF(ScreenPtr) = *((unsigned int *)&BitmapAddress);
   *ScreenPtr = (unsigned int)Color;
}
```

[LISTING FOUR]

```
; Draws all pixels in the list of horizontal lines passed in, in
; Hicolor (32K color) mode on an ET4000-based SuperVGA. Uses REP STOSW
; to fill each line. Tested with TASM 2.0. C near-callable as:
;      void DrawHCLineList(struct HLineList * HLineListPtr, int Color);

SCREEN_SEGMENT  equ     0a000h
GC_SEGMENT_SELECT equ   03cdh

HLine   struc
XStart  dw      ?       ;X coordinate of leftmost pixel in line
XEnd    dw      ?       ;X coordinate of rightmost pixel in line
HLine   ends

HLineList struc
Lngth   dw      ?       ;# of horizontal lines
```

```
YStart  dw      ?       ;Y coordinate of topmost line
HLinePtr dw     ?       ;pointer to list of horz lines
HLineList ends

Parms   struc
                dw      2 dup(?) ;return address & pushed BP
HLineListPtr    dw      ?       ;pointer to HLineList structure
Color           dw      ?       ;color with which to fill
Parms   ends

; Advances both the read and write windows to the next 64K bank.
; Note: Theoretically, a delay between IN and OUT may be needed under
; some circumstances to avoid accessing the VGA chip too quickly, but
; in actual practice, I haven't found any delay to be required.
INCREMENT_BANK  macro
        push    ax              ;preserve fill color
        push    dx              ;preserve scan line start pointer
        mov     dx,GC_SEGMENT_SELECT
        in      al,dx           ;get the current segment select
        add     al,11h          ;increment both the read & write banks
        out     dx,al           ;set the new bank #
        pop     dx              ;restore scan line start pointer
        pop     ax              ;restore fill color
        endm

        .model small
        .data
        extrn   _BitmapWidthInBytes:word
        .code
        public _DrawHCLineList
        align   2
_DrawHCLineList proc    near
        push    bp              ;preserve caller's stack frame
        mov     bp,sp           ;point to our stack frame
        push    si              ;preserve caller's register variables
        push    di
        cld                     ;make string instructions inc pointers
        mov     ax,SCREEN_SEGMENT
        mov     es,ax   ;point ES to display memory for REP STOS
        mov     si,[bp+HLineListPtr] ;point to the line list
        mov     ax,[_BitmapWidthInBytes] ;point to the start of the
        mul     [si+YStart]     ; first scan line on which to draw
        mov     di,ax           ;ES:DI points to first scan line to
        mov     al,dl           ; draw; AL is the initial bank #
                                ;upper nibble of AL is read bank #,
        mov     cl,4            ; lower nibble is write bank # (only
        shl     dl,cl           ; the write bank is really needed for
        or      al,dl           ; this module, but it's less confusing
                                ; to point both to the same place)
        mov     dx,GC_SEGMENT_SELECT
        out     dx,al           ;set the initial bank
        mov     dx,di           ;ES:DX points to first scan line
        mov     bx,[si+HLinePtr] ;point to the XStart/XEnd descriptor
                                ; for the first (top) horizontal line
        mov     si,[si+Lngth]   ;# of scan lines to draw
        and     si,si           ;are there any lines to draw?
        jz      FillDone        ;no, so we're done
        mov     ax,[bp+Color]   ;color with which to fill
        mov     bp,[_BitmapWidthInBytes] ;so we can keep everything
                                ; in registers inside the loop
                                ;***stack frame pointer destroyed!***
FillLoop:
        mov     di,[bx+XStart]  ;left edge of fill on this line
        mov     cx,[bx+XEnd]    ;right edge of fill
        sub     cx,di
        jl      LineFillDone    ;skip if negative width
        inc     cx              ;# of pixels to fill on this line
        add     di,di           ;*2 because pixels are 2 bytes in size
        add     dx,bp           ;do we cross a bank during this line?
        jnc     NormalFill      ;no
        jz      NormalFill      ;no
                                ;yes, there is a bank crossing on this
                                ; line; figure out where
        sub     dx,bp           ;point back to start of line
```

```
        add     di,dx           ;offset of left edge of fill
        jc      CrossBankBeforeFilling ;raster splits before the left
                                ; edge of fill
        add     cx,cx           ;fill width in bytes (pixels * 2)
        add     di,cx           ;do we split during the fill area?
        jnc     CrossBankAfterFilling ;raster splits after the right
        jz      CrossBankAfterFilling ; edge of fill
                                ;bank boundary falls within fill area;
                                ; draw in two parts, one in each bank
        sub     di,cx           ;point back to start of fill area
        neg     di              ;# of bytes left before split
        sub     cx,di           ;# of bytes to fill to the right of
                                ; the bank split
        push    cx              ;remember right-of-split fill width
        mov     cx,di           ;# of left-of-split bytes to fill
        shr     cx,1            ;# of left-of-split words to fill
        neg     di              ;offset at which to start filling
        rep     stosw           ;fill left-of-split portion of line
        pop     cx              ;get back right-of-split fill width
        shr     cx,1            ;# of right-of-split words to fill
                                ;advance to the next bank
        INCREMENT_BANK          ;point to the next bank (DI already
                                ; points to offset 0, as desired)
        rep     stosw           ;fill right-of-split portion of line
        add     dx,bp           ;point to the next scan line
        jmp     short CountDownLine ; (already advanced the bank)
;====================================================================
        align   2               ;dfill area is entirely to the left of
CrossBankAfterFilling:          ; the bank boundary
        sub     di,cx           ;point back to start of fill area
        shr     cx,1            ;CX = fill width in pixels
        jmp     short FillAndAdvance ;doesn't split until after the
                                ; fill area, so handle normally
;====================================================================
        align   2               ;fill area is entirely to the right of
CrossBankBeforeFilling:         ; the bank boundary
        INCREMENT_BANK          ;first, point to the next bank, where
                                ; the fill area resides
        rep     stosw           ;fill this scan line
        add     dx,bp           ;point to the next scan line
        jmp     short CountDownLine ; (already advanced the bank)
;====================================================================
        align   2               ;no bank boundary problems; just fill
NormalFill:                     ; normally
        sub     dx,bp           ;point back to start of line
        add     di,dx           ;offset of left edge of fill
FillAndAdvance:
        rep     stosw           ;fill this scan line
LineFillDone:
        add     dx,bp           ;point to the next scan line
        jnc     CountDownLine   ;didn't cross a bank boundary
        INCREMENT_BANK          ;did cross, so point to the next bank
CountDownLine:
        add     bx,size HLine   ;point to the next line descriptor
        dec     si              ;count off lines to fill
        jnz     FillLoop
FillDone:
        pop     di              ;restore caller's register variables
        pop     si
        pop     bp              ;restore caller's stack frame
        ret
;====================================================================
_DrawHCLineList endp
        end
```

[LISTING FIVE]

```
/* Demonstrates unweighted antialiased drawing in 640x480 Hicolor (32K color)
mode. Tested with Borland C++ 2.0 in C mode in the small model. */
```

```
#include <conio.h>
#include <dos.h>
#include <stdlib.h>
#include <string.h>
#include "polygon.h"
/* Draws the polygon described by the point list PointList in the
   color specified by RED, GREEN, AND BLUE, with all vertices
   offset by (x,y), to ScanLineBuffer, at ResMul multiple of
   horizontal and vertical resolution. The address of ColorTemp is
   cast to an int to satisfy the prototype for FillCnvxPolyDrvr; this
   trick will work only in a small data model */
#define DRAW_POLYGON_HIGH_RES(PointList,RED,GREEN,BLUE,x,y,ResMul) { \
   Polygon.Length = sizeof(PointList)/sizeof(struct Point);          \
   Polygon.PointPtr = PointTemp;                                     \
   /* Multiply all vertical & horizontal coordinates */              \
   for (k=0; k<sizeof(PointList)/sizeof(struct Point); k++) {        \
      PointTemp[k].X = PointList[k].X * ResMul;                      \
      PointTemp[k].Y = PointList[k].Y * ResMul;                      \
   }                                                                 \
   ColorTemp.Red=RED; ColorTemp.Green=GREEN; ColorTemp.Blue=BLUE;    \
   FillCnvxPolyDrvr(&Polygon, (int)&ColorTemp, x, y, DrawBandedList);}
#define SCREEN_WIDTH 640
#define SCREEN_SEGMENT 0xA000

void main(void);
extern void DrawPixel(int, int, char);
extern void DrawBandedList(struct HLineList *, struct RGB *);
extern int SetHCMode(int);

/* Table of gamma corrected mappings of linear color intensities in
   the range 0-255 to the nearest pixel values in the range 0-31,
   assuming a gamma of 2.3 */
static unsigned char ColorMappings[] = {
    0, 3, 4, 4, 5, 6, 6, 6, 7, 7, 8, 8, 8, 8, 9, 9, 9,10,10,10,
   10,10,11,11,11,11,11,12,12,12,12,12,13,13,13,13,13,13,14,14,
   14,14,14,14,14,15,15,15,15,15,15,15,16,16,16,16,16,16,16,16,
   17,17,17,17,17,17,17,17,17,18,18,18,18,18,18,18,18,18,19,19,
   19,19,19,19,19,19,19,20,20,20,20,20,20,20,20,20,20,20,20,21,
   21,21,21,21,21,21,21,21,21,21,22,22,22,22,22,22,22,22,22,22,
   22,22,22,23,23,23,23,23,23,23,23,23,23,23,23,24,24,24,24,24,
   24,24,24,24,24,24,24,24,24,25,25,25,25,25,25,25,25,25,25,25,
   25,25,25,26,26,26,26,26,26,26,26,26,26,26,26,26,26,26,26,27,27,
   27,27,27,27,27,27,27,27,27,27,27,27,27,28,28,28,28,28,28,
   28,28,28,28,28,28,28,28,28,28,28,29,29,29,29,29,29,29,29,29,
   29,29,29,29,29,29,29,30,30,30,30,30,30,30,30,30,30,30,30,30,
   30,30,30,30,30,30,31,31,31,31,31,31,31,31,31,31,31};
/* Pointer to buffer in which high-res scanned data will reside */
struct RGB *ScanLineBuffer;
int ScanBandStart, ScanBandEnd;  /* top & bottom of each high-res
                                    band we'll draw to ScanLineBuffer */
int ScanBandWidth;      /* # subpixels across each scan band */
int BitmapWidthInBytes = 640*2;  /* # of bytes per raster line in
                                    Hicolor VGA display memory */
void main()
{
   int i, j, k, m, Red, Green, Blue, jXRes, kXWidth;
   int SubpixelsPerMegapixel;
   unsigned int Megapixel, ResolutionMultiplier;
   long BufferSize;
   struct RGB ColorTemp;
   struct PointListHeader Polygon;
   struct Point PointTemp[4];
   static struct Point Face0[] =
         {{396,276},{422,178},{338,88},{288,178}};
   static struct Point Face1[] =
         {{306,300},{396,276},{288,178},{210,226}};
   static struct Point Face2[] =
         {{338,88},{266,146},{210,226},{288,178}};
   int LeftBound=210, RightBound=422, TopBound=88, BottomBound=300;
   union REGS regset;

   printf("Subpixel resolution multiplier:");
   scanf("%d", &ResolutionMultiplier);
   SubpixelsPerMegapixel = ResolutionMultiplier*ResolutionMultiplier;
```

```
   ScanBandWidth = SCREEN_WIDTH*ResolutionMultiplier;

   /* Get enough space for one scan line scanned out at high
      resolution horz and vert (each pixel is 4 bytes) */
   if ((BufferSize = (long)ScanBandWidth*4*ResolutionMultiplier) >
         0xFFFF) {
      printf("Band won't fit in one segment\n"); exit(0); }
   if ((ScanLineBuffer = malloc((int)BufferSize)) == NULL) {
      printf("Couldn't get memory\n"); exit(0); }

   /* Attempt to enable 640x480 Hicolor mode */
   if (SetHCMode(0x2E) == 0)
      { printf("No Hicolor DAC detected\n"); exit(0); };

   /* Scan out the polygons at high resolution one screen scan line at
      a time (ResolutionMultiplier high-res scan lines at a time) */
   for (i=TopBound; i<=BottomBound; i++) {
      /* Set the band dimensions for this pass */
      ScanBandEnd = (ScanBandStart = i*ResolutionMultiplier) +
            ResolutionMultiplier - 1;
      /* Clear the drawing buffer */
      memset(ScanLineBuffer, 0, BufferSize);
      /* Draw the current band of the cube to the scan line buffer */
      DRAW_POLYGON_HIGH_RES(Face0,0xFF,0,0,0,0,ResolutionMultiplier);
      DRAW_POLYGON_HIGH_RES(Face1,0,0xFF,0,0,0,ResolutionMultiplier);
      DRAW_POLYGON_HIGH_RES(Face2,0,0,0xFF,0,0,ResolutionMultiplier);

  /* Coalesce subpixels into normal screen pixels (megapixels) and draw them */
      for (j=LeftBound; j<=RightBound; j++) {
         jXRes = j*ResolutionMultiplier;
         /* For each screen pixel, sum all the corresponding
            subpixels, for each color component */
         for (k=Red=Green=Blue=0; k<ResolutionMultiplier; k++) {
            kXWidth = k*ScanBandWidth;
            for (m=0; m<ResolutionMultiplier; m++) {
               Red += ScanLineBuffer[jXRes+kXWidth+m].Red;
               Green += ScanLineBuffer[jXRes+kXWidth+m].Green;
               Blue += ScanLineBuffer[jXRes+kXWidth+m].Blue;
            }
         }
         /* Calc each color component's average brightness; convert
            that into a gamma corrected portion of a Hicolor pixel,
            then combine the colors into one Hicolor pixel */
         Red = ColorMappings[Red/SubpixelsPerMegapixel];
         Green = ColorMappings[Green/SubpixelsPerMegapixel];
         Blue = ColorMappings[Blue/SubpixelsPerMegapixel];
         Megapixel = (Red << 10) + (Green << 5) + Blue;
         DrawPixel(j, i, Megapixel);
      }
   }
   getch();    /* wait for a keypress */

   /* Return to text mode and exit */
   regset.x.ax = 0x0003;   /* AL = 3 selects 80x25 text mode */
   int86(0x10, &regset, &regset);
}
```

[LISTING SIX]

```
/* Draws pixels from the list of horizontal lines passed in, to a 32-bpp
buffer; drawing takes place only for scan lines between ScanBandStart and
ScanBandEnd, inclusive; drawing goes to ScanLineBuffer, with the scan line at
ScanBandStart mapping to the first scan line in ScanLineBuffer. Note that
Color here points to an RGB structure that maps directly to the buffer's pixel
format, rather than containing a 16-bit integer. Tested with Borland C++ 2.0
in C mode in the small model */

#include "polygon.h"
```

```
extern struct RGB *ScanLineBuffer;  /* drawing goes here */
extern int ScanBandStart, ScanBandEnd; /* limits of band to draw */
extern int ScanBandWidth;  /* # of subpixels across scan band */

void DrawBandedList(struct HLineList * HLineListPtr,
   struct RGB *Color)
{
   struct HLine *HLinePtr;
   int Length, Width, YStart = HLineListPtr->YStart, i;
   struct RGB *BufferPtr, *WorkingBufferPtr;

   /* Done if fully off the bottom or top of the band */
   if (YStart > ScanBandEnd) return;
   Length = HLineListPtr->Length;
   if ((YStart + Length) <= ScanBandStart) return;

   /* Point to XStart/XEnd descriptor for the first (top) horizontal line */
   HLinePtr = HLineListPtr->HLinePtr;

   /* Confine drawing to the specified band */
   if (YStart < ScanBandStart) {
      /* Skip ahead to the start of the band */
      Length -= ScanBandStart - YStart;
      HLinePtr += ScanBandStart - YStart;
      YStart = ScanBandStart;
   }
   if (Length > (ScanBandEnd - YStart + 1))
      Length = ScanBandEnd - YStart + 1;

   /* Point to the start of the first scan line on which to draw */
   BufferPtr = ScanLineBuffer + (YStart-ScanBandStart)*ScanBandWidth;

   /* Draw each horizontal line within the band in turn, starting with
      the top one and advancing one line each time */
   while (Length-- > 0) {
      /* Fill whole horiz line with Color if it has positive width */
      if ((Width = HLinePtr->XEnd - HLinePtr->XStart + 1) > 0) {
         WorkingBufferPtr = BufferPtr + HLinePtr->XStart;
         for (i = 0; i < Width; i++) *WorkingBufferPtr++ = *Color;
      }
      HLinePtr++;                /* point to next scan line X info */
      BufferPtr += ScanBandWidth; /* point to start of next line */
   }
}
```

[LISTING SEVEN]

```
/* POLYGON.H: Header file for polygon-filling code */

/* Describes a single point (used for a single vertex) */
struct Point {
   int X;   /* X coordinate */
   int Y;   /* Y coordinate */
};

/* 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 the 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 (used to
describe 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 */
};

/* Describes a color as an RGB triple, plus one byte for other info */
struct RGB { unsigned char Red, Green, Blue, Spare; };
```

Journal: Dr. Dobb's Journal Dec 1991 v16 n12 p131(8)

Catching up

Author: Abrash, Michael.
AttFile: GP-DEC91.ASC Antialiasing & Sierra Hicolor.

Abstract: Various topics discussed in previous columns are re-examined. The author was criticized for using non-standard terminology to describe polygons in several columns. Three categories are defined in the X-Window System: complex, nonconvex and convex; each is a specialized subset of the preceding one. The X Window System names were used to describe the types of polygons that can be drawn with each of the polygon filling techniques, but the names do not accurately describe the diverse types of polygons that the techniques can draw. Turbo Debugger is one of the most popular debuggers in use, but it messes with the VGA's registers when it gets control, even when it is run on a monochrome screen and the application is run on a color screen. Turbo Debugger cannot debug page flipping, among other types of graphics; solutions to this problem are described.

It's been nigh on a year now since I started this column, and it's time to catch up on some interesting odds and ends that I've been unsuccessfully trying to squeeze in for the better part of that time. No, I haven't forgotten that I said I'd start in on 3-D animation this month, but it's been put off until next month. Now, calm down; I know I promised, it's just that these things take time to do properly; you wouldn't want me to start off prematurely and end up with, God forbid, slow 3-D animation, would you? Don't send nasty letters; I'll get to it next month, honest I will.

The funny thing of it is, 3-D perspective drawing is basically pretty easy. Shading (at least in its simpler aspects) is relatively easy, too. Even hidden surface handling isn't bad--given lots of memory and processor power. The memory is easy enough to come by in 386 protected mode, but the processor power isn't. We're talking major silicon here, Intel i860s or, better yet, Crays; 386s don't cut the mustard, so some sleight of hand is in order. Fixed point arithmetic can replace floating point. Table look-ups can stand in for excruciatingly slow sine and cosine functions. Fast, approximate antialiasing can be used instead of precise but slow techniques. The real trick, though, is some combination of restrictions and techniques that allows real-time hidden surface removal. There are at least a dozen ways to remove hidden surfaces, but the fast ones don't always work, and the ones that always work are slow. Workstations deal with this by using dedicated hardware to perform z-buffering, or by applying MIPS by the bushel to sorting, or the like. Because we can't do any of that, I'm still pondering the best way to wring real-time hidden surface handling out of a 386.

So, this month, catching up; next month, 3-D animation. If I bug out again next month, then write nasty letters. At least I'll know you care.

Nomenclature Blues

Bill Huber wrote to take me to task--and a well-deserved kick in the fanny it was, I might add--for my use of non-standard terminology in describing polygons in February, March, and June columns. The X-Window System defines three categories of polygons: complex, nonconvex, and convex. These three categories, each a specialized subset of the preceding category, not-so-coincidentally map quite nicely to three increasingly fast polygon filling techniques. Therefore, I used the XWS names to describe the sorts of polygons that can be drawn with each of the polygon filling techniques.

The problem is that those names don't accurately describe all the sorts of polygons that the techniques are capable of drawing. Convex polygons are those for which no interior angle is greater than 180 degrees. The "convex" drawing approach described in February and March actually handles a number of polygons that are not convex; in fact, it can draw any polygon through which no horizontal line can be drawn that intersects the boundary more than twice. (In other words, the boundary reverses Y direction exactly twice,

disregarding polygons that have degenerated into horizontal lines, which I'm going to ignore.) Bill was kind enough to send me the pages out of Computational Geometry, An Introduction (Springer-Verlag, 1988) that describe the correct terminology; such polygons are, in fact, "monotone with respect to a vertical line" (which unfortunately makes a rather long #define variable). Actually, to be a tad more precise, I'd call them "monotone with respect to a vertical line and simple," where "simple" means "not self-intersecting." Similarly, the polygon type I called "nonconvex" is actually "simple," and I suppose what I called "complex" should be referred to as "nonsimple," or maybe just "none of the above."

This may seem like nit-picking, but actually, it isn't; what it's really about is the tremendous importance of having a shared language. In one of his books, Richard Feynman describes having developed his own mathematical framework, complete with his own notation and terminology, in high school. When he got to college and started working with other people who were at his level, he suddenly understood that people can't share ideas effectively unless they speak the same language; otherwise, they waste a great deal of time on misunderstandings and explanation. Or, as Bill Huber put it, "You are free to adopt your own terminology when it suits your purposes well. But you risk losing or confusing those who could be among your most astute readers--those who already have been trained in the same or a related field." Ditto. Likewise. D'accord. and mea culpa; I shall endeavor to watch my language in the future.

Nomenclature in Action

Just to show you how much different proper description and interchange of ideas can be, consider the case of identifying convex polygons. Several months back, a nonfunctional method for identifying such polygons--checking for exactly two X direction changes and two Y direction changes around the perimeter of the polygon--crept into this column by accident. That method, as I have since noted, does not work. Still, a fast method of checking for convex polygons would be highly desirable, because such polygons can be drawn with the fast code from the March column, rather than the relatively slow, general-purpose code from the June column.

Now consider Bill's point that we're not limited to drawing convex polygons in our "convex fill" code, but can actually handle any simple polygon that's monotone with respect to a vertical line. Additionally, consider Anton Treuenfels's point, made back in the August column, that life gets simpler if we stop worrying about which edge of a polygon is the left edge and which is the right, and instead just scan out each raster line starting at whichever edge is left-most. Now, what do we have?

What we have is an approach passed along by Jim Kent, of Autodesk Animator fame. If we modify the low-level code to check which edge is left-most on each scan line and start drawing there, as just described, then we can handle any polygon that's monotone with respect to a vertical line regardless of whether the edges cross. (I'll call this "monotone-vertical" from now on; if anyone wants to correct that terminology, jump right in.) In other words, we can then handle nonsimple polygons that are monotone-vertical; self-intersection is no longer a problem. Then, we just scan around the polygon's perimeter looking for exactly two direction reversals along the Y axis only, and if that proves to be the case, we can handle the polygon at high speed. Figure 1 shows polygons that can be drawn by a monotone-vertical capable filler; Figure 2 shows some that cannot.

Listing One (page 149) shows code to test whether a polygon is appropriately monotone. This test lends itself beautifully to assembly language implementation, because it's basically nothing but pointers and conditionals; unfortunately, I don't have room for an assembly version this month, but translation from Listing One is pretty straight-forward, should you care to do so yourself. Listing Two and Three (page 149) are variants of the fast convex polygon fill code from March, modified to be able to handle all monotone-vertical polygons, including nonsimple ones; the edge-scanning code (Listing Four from March) remains the same, and so is not shown here. Listing Four (page 150) shows the changes needed to convert Listing One from June to employ the vertical-monotone detection test and use the fast vertical-monotone drawing code whenever possible; note that Listing Five from June is also required in order for this code to link. Listing Five (page 150) this month is the latest version of the polygon.h header file.

Is monotone-vertical polygon detection worth all this trouble? Under the right circumstances, you bet. In a situation where a great many polygons are being drawn, and the application either doesn't know whether they're monotone-vertical or has no way to tell the polygon filler that they are, performance can be increased considerably if most polygons are, in fact, monotone-vertical. This potential performance advantage is helped along by the surprising fact that Jim's test for monotone-vertical status is simpler and faster than my original, non-functional test for convexity.

See what accurate terminology and effective communiation can do?

Graphics Debugging with Turbo Debugger

I don't know what debugger you use, but I'd be willing to wager that more of you than not use the same one I do, Turbo Debugger. It's powerful, the interface is good (although arguably it lost some ease of use in the transition to mouse support and CUA compatibility), and it's obviously the debugger of choice if you're using a Borland compiler. I would be remiss, however, if I failed to warn you of a serious problem with TD when it comes to debugging graphics programs.

The problem, simply put, is that TD mucks about with the VGA's registers when it gets control, even when you're running TD on a monochrome screen and your app on a color screen, courtesy of the -do switch. Although TD has no business fooling around with the VGA's registers in that case, seeing as how it's not using the color screen, that's not the problem; the problem is that when TD resumes execution of the app, it doesn't put all the VGA's registers back the way it found them. Given that the VGA's registers are readable, this inteference is unnecessary; it's also unfortunate, because it makes it impossible to debug some kinds of graphics with TD without a second computer available.

What sorts of graphics can't TD debug? Page flipping, for one; when it gets control, TD seems to force the start address of the displayed portion of the bitmap back to 0, thereby displaying page 0 whether you like it or not. Setting VGA registers manually via the I/O feature in the CPU window is also intefered with; often, hand-entered register settings just don't stick. Mode X (320x240, 256 colors, as discussed in the July through September columns) is messed up quite royally at times; some of the nonstandard register settings required to create mode X are apparently undone. There may be other problems, but those I've mentioned are enough to limit TD's ability to debug many sorts of graphics, especially animation.

There is a solution, as it happens: Get another computer, run a serial link from your main to the second computer, and debug your programs running on that system via TDREMOTE. When running in remote mode, TD doesn't mess with any registers, and becomes an ideal graphics debugging tool. The downside, of course, is that you have to have a second computer; also, TDREMOTE is slower in almost every respect than TD.

Hi-Res VGA Page Flipping

This is one of those odd little items that might come in handy someday. The background is this: On a standard VGA, hi-res mode is mode 12h, which offers 640x480 resolution with 16 colors. That's a nice mode, with plenty of pixels, and square ones at that, but it lacks one thing--page flipping. The problem is that the mode 12h bitmap is 144 Kbytes in size, and the VGA has only 256 Kbytes total, too little memory for two of those monster mode 12h pages. With only one page, flipping is obviously out of the question, and without page flipping, top-flight, hi-res animation can't be implemented. The standard fallback is to use the EGA's hi-res mode, mode 10h (640x350, 16 colors) for page flipping, but this mode is less than ideal for a couple of reasons: It offers sharply lower vertical resolution, and it's lousy for handling scaled-up CGA graphics, because the vertical resolution is a fractional multiple--1.75 times, to be exact--of that of the CGA. CGA resolution may not seem important these days, but many images were originally created for the CGA, as were many graphics packages and games, and it's at least convenient to be able to handle CGA graphics easily.

There are a couple of interesting, if imperfect, solutions to the problem of hi-res page flipping. One is to use the split screen to enable page flipping only in the top two-thirds of the screen; see "VGA Split-Screen Animation," in the June, 1991 issue of PC Techniques, for details (and for details on the mechanics of page flipping, as well). This doesn't address the CGA problem, but it does yield square pixels and a full 640x480 screen resolution, although not all those pixels are flippable.

A second solution is to program the screen to a 640x400 mode. such a mode uses almost every byte of display memory (64,000 bytes, actually; you could add another few lines, if you really wanted to), and thereby provides the highest resolution possible on the VGA for a fully page-flipped display. It maps well to CGA resolutions, being either identical or double in both dimensions. As an added benefit, it offers an easy-on-the-eyes 70-Hz frame rate, as opposed to the 60 Hz that is the best that mode 12h can offer, due to the design of standard VGA monitors. Best of all, perhaps, is that 640x400 16-color mode is easy to set up.

The key to 640x400 mode is understanding that on a VGA, mode 10h (640x350) is, at heart, a 400-scan-line mode. What I mean by that is that in mode 10h, the Vertical Total register, which controls the total number of scan lines, both displayed and nondisplayed, is set to 447, exactly the same as in the VGA's text modes, which do in fact support 400 scan lines. A properly sized and centered display is achieved in mode 10h by setting the polarity of the sync pulses to tell the monitor to scan vertically at a faster rate (to make fewer lines fill the screen), by starting the overscan after 350 lines, and by setting the vertical sync and blanking pulses appropriately for the faster vertical scanning rate. Changing those settings is all that's required to turn mode 10h into a 640x400 mode, and that's easy to do, as illustrated by Listing Six (page 150), which provides mode set code for 640x400 mode.

In 640x400, 16-color mode, page 0 runs from offset 0 to offset 31,999 (7CFFh), and page 1 runs from offset 32,000 (7D00h) to 63,999 (0F9FFh). Page 1 is selected by programming the Start Address registers (CRTC registers 0Ch, the high 8 bits, and 0Dh, the low 8 bits) to 7D00h. Actually, because the low byte of the start address is 0 for both pages, you can page flip simply by writing 0 or 7Dh to the Start Address High register (CRTC register 0Ch); this has the benefit of eliminating a nasty class of potential synchronization bugs that can arise when both registers must be set. Listing Seven (page 150) illustrates simple 640x480 page flipping.

The 640x400 mode isn't exactly earth-shaking, but it can come in handy for page flipping and CGA emulation, and I'm sure that some of you will find it useful at one time or another.

Modifying VGA Registers

EGA registers are not readable. VGA registers are readable. This revelation will not come as news to most of you, but many programmers still insist on setting entire VGA registers even when they're modifying only selected bits, as if they were programming the EGA. This comes to mind because I recently received a query inquiring why write mode 1 (in which the contents of the latches are copied directly to display memory) didn't work in mode X. Actually, write mode 1 does work in mode X; it didn't work when this particular correspondent enabled it because he did so by writing the value 01h to the Graphics Mode register. As it happens, the write mode field is only one of several fields in that register, as shown in Figure 3. In 256-color modes, one of the other fields--bit 6, which enables 256-color pixel formatting--is not 0, and setting it to 0 messes the screen up quite thoroughly.

The correct way to set a field within a VGA register is, of course, to read the register, mask off the desired field, insert the desired setting, and write the result back to the register. In the case of setting the VGA to write mode 1, consult Example 1.

This approach is more of a nuisance than simply setting the whole register, but it's safer. It's also slower; for cases where you must set a field repeatedly, it might be worthwhile to read and mask the register once at the start, and save it in a variable, so that the value is readily available in memory and need not be repeatedly read from the port. This approach is especially attractive because INs are much slower than memory accesses on 386 and 486 machines.

Astute readers may wonder why I didn't put a delay sequence, such as JMP $+2, between the IN and OUT involving the same register. There are, after all, guidelines from IBM specifying that a certain period should be allowed to elapse before a second access to an I/O port is attempted, because not all devices can respond as rapidly as a 286 or faster chip can access a port. My answer is that while I can't guarantee that a delay isn't needed, I've never found a VGA that required one; I suspect that the delay specification has more to do with motherboard chips such as the timer, the interrupt controller, and the like, and I sure hate to waste the delay time if it's not necessary. However, I've never been able to find anyone with the definitive word on whether delays might ever be needed when accessing VGAs, so if you know the gospel truth, or if you know of a VGA/processor combo that does require delays, please drop me a line. You'd be doing a favor for a whole generation of graphics programmers who aren't sure whether they're skating on thin ice without delays.

- - - -

horizontal line drawn through the polygon at any point would cross exactly two active edges (neither horizontal lines nor zero-length edges count as active edges; both are acceptable anywhere in the polygon). Right & left edges may cross (polygons may be nonsimple). Polygons that are not convex according to this definition won't be drawn properly. (Yes, "convex" is a lousy name for this type of polygon, but it's convenient; use "monotone-vertical" if it makes you happier!)
****
NOTE: the low-level drawing routine, DrawHorizontalLineList, must be able to reverse the edges, if necessary to make the correct edge left edge. It must also expect right edge to be specified in +1 format (the X coordinate is 1 past highest coordinate to draw). In both respects, this differs from low-level drawing routines presented in earlier columns; changes are necessary to make it possible to draw nonsimple monotone-vertical polygons; that in turn makes it possible to use Jim Kent's test for monotone-vertical polygons.
****
Returns 1 for success, 0 if memory allocation failed */

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

/* Advances the index by one vertex forward through the vertex list,
wrapping at the end of the list */
#define INDEX_FORWARD(Index) \
   Index = (Index + 1) % VertexList->Length;

/* Advances the index by one vertex backward through the vertex list,
wrapping at the start of the list */
#define INDEX_BACKWARD(Index) \
   Index = (Index - 1 + VertexList->Length) % VertexList->Length;

/* Advances the index by one vertex either forward or backward through
the vertex list, wrapping at either end of the list */
#define INDEX_MOVE(Index,Direction)                                    \
   if (Direction > 0)                                                  \
      Index = (Index + 1) % VertexList->Length;                        \
   else                                                                \
      Index = (Index - 1 + VertexList->Length) % VertexList->Length;

extern void ScanEdge(int, int, int, int, int, int, struct HLine **);
extern void DrawHorizontalLineList(struct HLineList *, int);

int FillMonotoneVerticalPolygon(struct PointListHeader * VertexList,
      int Color, int XOffset, int YOffset)
{
   int i, MinIndex, MaxIndex, MinPoint_Y, MaxPoint_Y;
   int NextIndex, CurrentIndex, PreviousIndex;
   struct HLineList WorkingHLineList;
   struct HLine *EdgePointPtr;
   struct Point *VertexPtr;

   /* Point to the vertex list */
   VertexPtr = VertexList->PointPtr;

   /* Scan the list to find the top and bottom of the polygon */
   if (VertexList->Length == 0)
      return(1);  /* reject null polygons */
   MaxPoint_Y = MinPoint_Y = VertexPtr[MinIndex = MaxIndex = 0].Y;
   for (i = 1; i < VertexList->Length; i++) {
      if (VertexPtr[i].Y < MinPoint_Y)
         MinPoint_Y = VertexPtr[MinIndex = i].Y; /* new top */
      else if (VertexPtr[i].Y > MaxPoint_Y)
         MaxPoint_Y = VertexPtr[MaxIndex = i].Y; /* new bottom */
   }

   /* Set the # of scan lines in the polygon, skipping the bottom edge */
   if ((WorkingHLineList.Length = MaxPoint_Y - MinPoint_Y) <= 0)
      return(1);  /* there's nothing to draw, so we're done */
   WorkingHLineList.YStart = YOffset + MinPoint_Y;

   /* Get memory in which to store the line list we generate */
   if ((WorkingHLineList.HLinePtr =
```

```
         (struct HLine *) (malloc(sizeof(struct HLine) *
         WorkingHLineList.Length))) == NULL)
      return(0);  /* couldn't get memory for the line list */

   /* Scan the first edge and store the boundary points in the list */
   /* Initial pointer for storing scan converted first-edge coords */
   EdgePointPtr = WorkingHLineList.HLinePtr;
   /* Start from the top of the first edge */
   PreviousIndex = CurrentIndex = MinIndex;
   /* Scan convert each line in the first edge from top to bottom */
   do {
      INDEX_BACKWARD(CurrentIndex);
      ScanEdge(VertexPtr[PreviousIndex].X + XOffset,
            VertexPtr[PreviousIndex].Y,
            VertexPtr[CurrentIndex].X + XOffset,
            VertexPtr[CurrentIndex].Y, 1, 0, &EdgePointPtr);
      PreviousIndex = CurrentIndex;
   } while (CurrentIndex != MaxIndex);

   /* Scan the second edge and store the boundary points in the list */
   EdgePointPtr = WorkingHLineList.HLinePtr;
   PreviousIndex = CurrentIndex = MinIndex;
   /* Scan convert the second edge, top to bottom */
   do {
      INDEX_FORWARD(CurrentIndex);
      ScanEdge(VertexPtr[PreviousIndex].X + XOffset,
            VertexPtr[PreviousIndex].Y,
            VertexPtr[CurrentIndex].X + XOffset,
            VertexPtr[CurrentIndex].Y, 0, 0, &EdgePointPtr);
      PreviousIndex = CurrentIndex;
   } while (CurrentIndex != MaxIndex);

   /* Draw the line list representing the scan converted polygon */
   DrawHorizontalLineList(&WorkingHLineList, Color);

   /* Release the line list's memory and we're successfully done */
   free(WorkingHLineList.HLinePtr);
   return(1);
}
```

[LISTING THREE]

```
; Draws all pixels in list of horizontal lines passed in, in mode 13h, VGA's
; 320x200 256-color mode. Uses REP STOS to fill each line.
; ****************************************************************
; NOTE: is able to reverse the X coords for a scan line, if necessary to make
; XStart < XEnd. Expects whichever edge is rightmost on any scan line to be in
; +1 format; that is, XEnd is 1 greater than rightmost pixel to draw. if
; XStart == XEnd, nothing is drawn on that scan line.
; ****************************************************************
; C near-callable as:
;     void DrawHorizontalLineList(struct HLineList * HLineListPtr, int Color);
; All assembly code tested with TASM 2.0 and MASM 5.0

SCREEN_WIDTH    equ     320
SCREEN_SEGMENT  equ     0a000h

HLine   struc
XStart  dw      ?       ;X coordinate of leftmost pixel in line
XEnd    dw      ?       ;X coordinate of rightmost pixel in line
HLine   ends

HLineList struc
Lngth   dw      ?       ;# of horizontal lines
YStart  dw      ?       ;Y coordinate of topmost line
HLinePtr dw     ?       ;pointer to list of horz lines
HLineList ends

Parms   struc
                dw      2 dup(?) ;return address & pushed BP
HLineListPtr    dw      ?       ;pointer to HLineList structure
```

```
Color           dw      ?       ;color with which to fill
Parms   ends
        .model small
        .code
        public _DrawHorizontalLineList
        align   2
_DrawHorizontalLineList proc
        push    bp              ;preserve caller's stack frame
        mov     bp,sp           ;point to our stack frame
        push    si              ;preserve caller's register variables
        push    di
        cld                     ;make string instructions inc pointers

        mov     ax,SCREEN_SEGMENT
        mov     es,ax   ;point ES to display memory for REP STOS

        mov     si,[bp+HLineListPtr] ;point to the line list
        mov     ax,SCREEN_WIDTH ;point to the start of the first scan
        mul     [si+YStart]     ; line in which to draw
        mov     dx,ax           ;ES:DX points to first scan line to draw
        mov     bx,[si+HLinePtr] ;point to the XStart/XEnd descriptor
                                ; for the first (top) horizontal line
        mov     si,[si+Lngth]   ;# of scan lines to draw
        and     si,si           ;are there any lines to draw?
        jz      FillDone        ;no, so we're done
        mov     al,byte ptr [bp+Color] ;color with which to fill
        mov     ah,al           ;duplicate color for STOSW
FillLoop:
        mov     di,[bx+XStart]  ;left edge of fill on this line
        mov     cx,[bx+XEnd]    ;right edge of fill
        cmp     di,cx           ;is XStart > XEnd?
        jle     NoSwap          ;no, we're all set
        xchg    di,cx           ;yes, so swap edges
NoSwap:
        sub     cx,di           ;width of fill on this line
        jz      LineFillDone    ;skip if zero width
        add     di,dx           ;offset of left edge of fill
        test    di,1            ;does fill start at an odd address?
        jz      MainFill        ;no
        stosb                   ;yes, draw the odd leading byte to
                                ; word-align the rest of the fill
        dec     cx              ;count off the odd leading byte
        jz      LineFillDone    ;done if that was the only byte
MainFill:
        shr     cx,1            ;# of words in fill
        rep     stosw           ;fill as many words as possible
        adc     cx,cx           ;1 if there's an odd trailing byte to
                                ; do, 0 otherwise
        rep     stosb           ;fill any odd trailing byte
LineFillDone:
        add     bx,size HLine   ;point to the next line descriptor
        add     dx,SCREEN_WIDTH ;point to the next scan line
        dec     si              ;count off lines to fill
        jnz     FillLoop
FillDone:
        pop     di              ;restore caller's register variables
        pop     si
        pop     bp              ;restore caller's stack frame
        ret
_DrawHorizontalLineList endp
        end
```

[LISTING FOUR]

```
/*** Replace this... ***/
extern int FillConvexPolygon(struct PointListHeader *, int, int, int);

/*** ...with this... ***/
extern int FillMonotoneVerticalPolygon(struct PointListHeader *,
   int, int, int);
```

```
extern int PolygonIsMonotoneVertical(struct PointListHeader *);

/*** Replace this... ***/
#ifdef CONVEX_CODE_LINKED
   /* Pass convex polygons through to fast convex polygon filler */
   if (PolygonShape == CONVEX)
      return(FillConvexPolygon(VertexList, Color, XOffset, YOffset));
#endif

/*** ...with this... ***/
#ifdef CONVEX_CODE_LINKED
   /* Pass convex polygons through to fast convex polygon filler */
   if ((PolygonShape == CONVEX) ||
         PolygonIsMonotoneVertical(VertexList))
      return(FillMonotoneVerticalPolygon(VertexList, Color, XOffset,
            YOffset));
#endif
```

[LISTING FIVE]

```
/* POLYGON.H: Header file for polygon-filling code */

#define CONVEX    0
#define NONCONVEX 1
#define COMPLEX   2

/* Describes a single point (used for a single vertex) */
struct Point {
   int X;   /* X coordinate */
   int Y;   /* Y coordinate */
};

/* Describes 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
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 (used to
describe 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 */
};

/* Describes a color as an RGB triple, plus one byte for other info */
struct RGB { unsigned char Red, Green, Blue, Spare; };
```

[LISTING SIX]

```
/* Mode set routine for VGA 640x400 16-color mode. Tested with
   Borland C++ 2, in C compilation mode. */

#include <dos.h>

void Set640x400()
```

```
{
   union REGS regset;

   /* First, set to standard 640x350 mode (mode 10h) */
   regset.x.ax = 0x0010;
   int86(0x10, &regset, &regset);

   /* Modify the sync polarity bits (bits 7 & 6) of the
      Miscellaneous Output register (readable at 0x3CC, writable at
      0x3C2) to select the 400-scan-line vertical scanning rate */
   outp(0x3C2, ((inp(0x3CC) & 0x3F) | 0x40));

   /* Now, tweak the registers needed to convert the vertical
      timings from 350 to 400 scan lines */
   outpw(0x3D4, 0x9C10);   /* adjust the Vertical Sync Start register
                              for 400 scan lines */
   outpw(0x3D4, 0x8E11);   /* adjust the Vertical Sync End register
                              for 400 scan lines */
   outpw(0x3D4, 0x8F12);   /* adjust the Vertical Display End
                              register for 400 scan lines */
   outpw(0x3D4, 0x9615);   /* adjust the Vertical Blank Start
                              register for 400 scan lines */
   outpw(0x3D4, 0xB916);   /* adjust the Vertical Blank End register
                              for 400 scan lines */
}
```

[LISTING SEVEN]

```
/* Sample program to exercise VGA 640x400 16-color mode page flipping, by
drawing a horizontal line at the top of page 0 and another at bottom of page 1,
then flipping between them once every 30 frames. Tested with Borland C++ 2,
in C compilation mode. */

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

#define SCREEN_SEGMENT  0xA000
#define SCREEN_HEIGHT   400
#define SCREEN_WIDTH_IN_BYTES 80
#define INPUT_STATUS_1  0x3DA /* color-mode address of Input Status 1
                                 register */
/* The page start addresses must be even multiples of 256, because page
flipping is performed by changing only the upper start address byte */
#define PAGE_0_START 0
#define PAGE_1_START (400*SCREEN_WIDTH_IN_BYTES)

void main(void);
void Wait30Frames(void);
extern void Set640x400(void);

void main()
{
   int i;
   unsigned int far *ScreenPtr;
   union REGS regset;

   Set640x400();  /* set to 640x400 16-color mode */

   /* Point to first line of page 0 and draw a horizontal line across screen */
   FP_SEG(ScreenPtr) = SCREEN_SEGMENT;
   FP_OFF(ScreenPtr) = PAGE_0_START;
   for (i=0; i<(SCREEN_WIDTH_IN_BYTES/2); i++) *ScreenPtr++ = 0xFFFF;

   /* Point to last line of page 1 and draw a horizontal line across screen */
   FP_OFF(ScreenPtr) =
         PAGE_1_START + ((SCREEN_HEIGHT-1)*SCREEN_WIDTH_IN_BYTES);
   for (i=0; i<(SCREEN_WIDTH_IN_BYTES/2); i++) *ScreenPtr++ = 0xFFFF;

   /* Now flip pages once every 30 frames until a key is pressed */
   do {
```

```
      Wait30Frames();

      /* Flip to page 1 */
      outpw(0x3D4, 0x0C | ((PAGE_1_START >> 8) << 8));

      Wait30Frames();

      /* Flip to page 0 */
      outpw(0x3D4, 0x0C | ((PAGE_0_START >> 8) << 8));
   } while (kbhit() == 0);

   getch(); /* clear the key press */

   /* Return to text mode and exit */
   regset.x.ax = 0x0003;   /* AL = 3 selects 80x25 text mode */
   int86(0x10, &regset, &regset);
}

void Wait30Frames()
{
   int i;

   for (i=0; i<30; i++) {
      /* Wait until we're not in vertical sync, so we can catch leading edge */
      while ((inp(INPUT_STATUS_1) & 0x08) != 0) ;
      /* Wait until we are in vertical sync */
      while ((inp(INPUT_STATUS_1) & 0x08) == 0) ;
   }
}
```

Journal: Dr. Dobb's Journal Jan 1992 v17 n1 p121(7)

3-D animation

Author: Abrash, Michael.
AttFile: GP-JAN92.ASC Page flips & polygon fills in C.

Abstract: Tips and techniques for programming three-dimensional animation routines are presented. 3-D animation can be performed on the 80x86 architecture using a combination of drawing pipelines, projection, translation and rotation algorithms. Objects are built out of polygons that represent a surface; the polygon must be transformed from world space into view space, and the projection of the Z axis must follow a particular ratio in view space to accurately represent world space. Translation is the process of adding X, Y and Z offsets to a coordinate to move it through world space and actually requires nothing more than an addition for each axis. Rotation circularly moves coordinates around the origin in order to turn objects to the desired attitude before translating them into world space. A simple example program that displays and moves a polygon in 3-D space is included; source code and notes are provided.

When first I started programming micros, more than 11 years ago now, there wasn't much money in it, or visibility, or anything you could call a promising career. Sometimes, it was a way to accomplish things that would never have gotten done otherwise because minicomputer time cost too much; other times, it paid the rent; mostly, though, it was just for fun. Given free computer time for the first time in my life, I went wild, writing versions of all sorts of software I had seen on mainframes, in arcades, wherever. It was a wonderful way to learn how computers work: trial and error in an environment where nobody minded the errors, with no meter ticking.

Many sorts of software demanded no particular skills other than a quick mind and a willingness to experiment: Space Invaders, for instance, or full-screen operating system shells. Others, such as compilers, required a good deal of formal knowledge. Still others required not only knowledge but also more horsepower than I had available. The latter I filed away on my ever-growing wish list, and then forgot about for a while.

Three-dimensional animation was the most alluring of the areas I passed over long ago. The information needed to do rotation, projection, rendering, and the like was neither so well developed nor widely so available then as it is now, although, in truth, it seemed more intimidating than it ultimately proved to be. Even had I possessed the knowledge, though, it seems unlikely that I could have coaxed satisfactory 3-D animation out of a 4-MHz Z80 system with 160x72 monochrome graphics. In those days, 3-D was pretty much limited to outrageously expensive terminals attached to minis or mainframes.

Times change, and they seem to do so much faster in computer technology than in other parts of the universe. A 486 is capable of decent 3-D animation, owing to its integrated math coprocessor; not in the class of, say, an i860, but pretty good nonetheless. A 386 is less satisfactory, though; the 387 is no match for the 486's coprocessor, and most 386 systems lack coprocessors. However, all is not lost; 32-bit registers and built-in integer multiply and divide hardware make it possible to do some very interesting 3-D animation on a 386 with fixed-point arithmetic. Actually, it's possible to do a surprising amount of 3-D animation in real mode, and even on lesser 80x86 processors; in fact, the code in this article will perform real-time 3-D animation (admittedly very simple, but nonetheless real-time and 3-D) on a 286 without a 287, even though the code is written in real-mode C and uses floating-point arithmetic. In short, the potential for 3-D animation on the 80x86 family may be quite a bit greater than you think.

This month, we kick off an exploration of some of the sorts of 3-D animation that can be performed on the 80x86 family. Mind you, I'm talking about arbitrary 3-D animation, with all calculations and drawing performed on-the-fly; generating frames ahead of time and playing them back is an excellent technique, but I'm interested in seeing how far we can push purely real-time animation. Granted, we're not going to make it to the level of

Terminator 2, but we should have some fun nonetheless. The initial columns may seem pretty basic to those of you experienced with 3-D programming, and, at the same time, 3-D neophytes will inevitably be distressed at the amount of material I skip or skim over. That can't be helped, but at least there'll working code, the references mentioned later, and some explanation; that should be enough to start you on your way with 3-D.

Animating in three dimensions is a complex task, so this will be an ongoing topic, with later columns building on previous ones; even this firt 3-D column will rely on polygon fill and page-flip code from earlier columns. From time to time I'll skip to other topics, but I'll return to 3-D animation on a regular basis, because, to my mind, it's one of the most exciting things that can be done with a computer--and because, with today's hardware, it can be done.

References on 3-D Drawing

There are several good sources for information about 3-D graphics. Foley and van Dam's Computer Graphics: Principles and Practice (Second Edition, Addison-Wesley, 1990) provides a lengthy discussion of the topic and a great many references for further study. Unfortunately, this book is heavy going at times; a more approchable discussion is provided in Principles of Interactive Computer Graphics, by Newman and Sproull (McGraw-Hill, 1979). Although the latter book lacks the last decade's worth of graphics developments, it nonetheless provides a good overview of basic 3-D techniques, including many of the approaches likely to work well in real time on a PC.

A source that you may or may not find useful is the series of six books on C graphics by Lee Adams, as exemplified by High-Performance CAD Graphics in C (Windcrest/Tab, 1986). (I don't know if all six books discuss 3-D graphics, but the four I've seen do.) To be honest, this book has a number of problems, including: relatively little theory and explanation; incomplete and sometimes erroneous discussions of graphics hardware, use of nothing but global variables, with cryptic names like "array3" and "B21;" and--well, you get the idea. On the other hand, the book at least touches on a great many aspects of 3-D drawing, and there's a lot of C code to back that up. A number of people have spoken warmly to me of Adam's books as their introduction to 3-D graphics. I wouldn't recommend these books as your only 3-D references, but if you're just starting out, you might want to look at one and see if it helps you bridge the gap between the theory and implementation of 3-D graphics.

The 3-D Drawing Pipeline

Each 3-D object that we'll handle will be built out of polygons that represent the surface of the object. Figure 1 shows the stages a polygon goes through enroute to being drawn on the screen. (For the present, we'll avoid complications such as clipping, lighting, and shading.) First, the polygon is transformed from object space, the coordinate system the object is defined in, to world space, the coordinate system of the 3-D universe. Transformation may involve rotating, scaling, and moving the polygon. Fortunately, applying the desired transformation to each of the polygon vertices in an object is equivalent to transforming the polygon; in other words, transformation of a polygon is fully defined by transformation of its vertices, so it is not necessary to transform every point in a polygon, just the vertices. Likewise, transformation of all the polygon vertices in an object fully transforms the object.

Once the polygon is in world space, it must again be transformed, this time into view space, the space defined such that the viewpoint is at (0,0,0), looking down the Z axis, with the Y axis straight up and the X axis off to the right. Once in view space, the polygon can be perspective-projected to the screen, with the projected X and Y coordinates of the vertices finally being used to draw the polygon.

That's really all there is to basic 3-D drawing: transformation from object space to world space to view space to the screen. Next, we'll look at the mechanics of transformation.

One note: I'll use a purely right-handed convention for coordinate systems. Right-handed means that if you hold your right hand with your fingers curled and the thumb sticking out, the thumb points along the Z axis and the fingers point in the direction of rotation from the X axis to the Y axis, as shown in Figure 2. Rotations about an axis are counter-clockwise, as viewed looking down an axis toward the origin. The handedness of a coordinate system is just a convention, and left-handed would do equally well; however, right-handed is generally used for object and world space. Sometimes, the handedness is flipped for view space, so that increasing Z equals increasing distance from the viewer along the line of sight, but I have chosen not to do that here, to avoid confusion. Therefore, Z decreases as distance along the line of sight increases; a view space coordinate of (0,0-1000) is directly ahead, twice as far away as a coordinate of (0,0-500).

Projection

Working backward from the final image, we want to take the vertices of a polygon, as transformed into view space, and project them to 2-D coordinates on the screen, which, for projection purposes, is assumed to be centered on and perpendicular to the Z axis in view space, at some distance from the screen. We're after visual realism, so we'll want to do a perspective projection, in order that farther objects look smaller than nearer objects, and so that the field of view will widen with distance. This is done by scaling the X and Y coordinates of each point proportionately to the Z distance of the point from the viewer, a simple matter of similar triangles, as shown in Figure 3. It doesn't really matter how far down the Z axis the screen is assumed to be; what matters is the ratio of the distance of the screen from the viewpoint to the width of the screen. This ratio defines the rate of divergence of the viewing pyramid--the full field of view--and is used for performing all perspective projections. Once perspective projection has been performed, all that remains before calling the polygon filler is to convert the projected X and Y coordinates to integers, appropriately clipped and adjusted as necessary to center the origin on the screen or otherwise map the image into a window, if desired.

Translation

Translation means adding X, Y, and Z offsets to a coordinate in order to move it linearly through space. Translation is as simple as it seems; it requires nothing more than an addition for each axis. Translation is, for example, used to move objects from object space, in which the center of the object is typically the origin (0,0,0), into world space, where the object may be located anywhere.

Rotation

Rotation is the process of circularly moving coordinates around the origin. For our present purposes, it's necessary only to rotate objects about their centers in object space, so as to turn them to the desired attitude before translating them into world space.

Rotation of a point about an axis is accomplished by transforming it according to the formulas shown in Figure 4. These formulas map into the more generally useful matrix-multiplication forms also shown in Figure 4. Matrix representation is more useful for two reasons: First, it is possible to concatenate multiple rotations into a single matrix by multiplying them together in the desired order; that single matrix can then be used to perform the rotations more efficiently. Second, 3x3 rotation matrices can become the upper-left-hand portions of 4x4 matrices that also perform translation (and scaling as well, but we won't need scaling in the near future), as shown in Figure 5. A 4x4 matrix of this sort utilizes homogeneous coordinates; that's a topic way beyond this column, but, basically, homogeneous coordinates allow you to handle both rotations and translations with 4x4 matrices, thereby allowing the same code to work with either, and making it possible to concatenate a long series of rotations and translations into a single matrix that performs the same transformation as the sequence of rotations and transformations.

There's much more to be said about transformations and the supporting matrix math, but, in the interests of getting to working code this month, I'll leave that to be discussed as the need arises.

A Simple 3-D Example

At this point, we know enough to be able to put together a simple 3-D animation example. The example will do nothing more complicated than display a single polygon as it sits in 3-D space, rotating around the Y axis. To make things a little more interesting, we'll let the user move the polygon around in space with the arrow keys, and with the "A" (away), and "T" (toward) keys. The sample program requires two sorts of functionality: the ability to transform and project the polygon from object space onto the screen (3-D functionality), and the ability to draw the projected polygon (complete with clipping) and handle the other details of animation (2-D functionality).

Happily (and not coincidentally), we put together a nice 2-D animation framework back in the July, August, and September columns, so we don't much have to worry about non 3-D details. Basically, we'll use mode X (320x240, 256 colors, as dicussed in the above-mentioned columns), and we'll flip between two display pages, drawing to one while the other is displayed. One new 2-D element that we need is the ability to clip polygons; while we could avoid this for the moment by restricting the range of motion of the polygon so that it stays fully on the screen, certainly in the long run we'll want to be able to handle partially or fully clipped polygons. Listing One (page 140) is the low-level code for a mode X polygon filler that supports clipping. (The high-level polygon fill code is mode independent, and is the same as in the February and March columns, as noted further on.) The clipping is implemented at the low level, by trimming the Y extent of the scan line list up front, then clipping the X coordinates of each scan line in turn. This is not a particularly fast approach to clipping--ideally, the polygon would be clipped before it was scanned into a line list, avoiding potentially wasted scanning and eliminating the line-by-line X clipping--but it's much simpler, and, as we shall see, polygon filling performance is the least of our worries at the moment.

The other 2-D element we need is some way to erase the polygon at its old location before it's moved and redrawn. We'll do that by remembering the bounding rectangle of the polygon each time it's drawn, then erasing by clearing that area with a rectangle fill.

With the 2-D side of the picture well under control, we're ready to concentrate on the good stuff. Listings Two through Five are the sample 3-D animation program. Listing Two (page 140) provides matrix multiplication functions in a straightforward fashion. Listing Three (page 140) transforms, projects, and draws polygons. Listing Four (page 142) is the general header file for the program, and Listing Five (page 143) is the main animation program. Other modules required are: Listings One and Six from July (mode X mode set, rectangle fill); Listing Six from September (page 146); Listing Four from March (polygon edge scan); and the FillConvexPolygon() function from Listing One from February. To simplify matters, the archive file 3D-1.ARC, with all required modules and a make file, will be made available as part of the listings from this issue wherever DDJ code is posted.

Notes on the 3-D Animation Example

The sample program transforms the polygon's vertices from object space to world space to view space to the screen, as described earlier. In this case, world space and view space are congruent--we're looking right down the negative Z axis of world space--so the transformation matrix from world to view is the identity matrix; you might want to experiment with changing this matrix to change the viewpoint. The sample program uses 4x4 homogeneous coordinate matrices to perform transformations, as described above. Floating-point arithmetic is used for all 3-D calculations. Setting the translation from object space to world space is a simple matter of changing the appropriate entry in the fourth column of the object-to-world transformation matrix. Setting the rotation around the Y axis is almost as simple, requiring only the setting of the four matrix entries that control the Y rotation to the sines and cosines of the desired rotation. However,

rotations involving more than one axis require multiple rotation matrices, one for each axis rotated around; those matrices are then concatenated together to produce the object-to-world transformation. This area is trickier than it might initially appear to be; more in the near future.

The maximum translation along the Z axis is limited to -40; this keeps the polygon from extending past the viewpoint to positive Z coordinates. This would wreak havoc with the projection and 2-D clipping, and would require 3-D clipping, which is far more complicated than 2-D. We'll get to 3-D clipping someday, but, for now, it's much simpler just to limit all vertices to negative Z coordinates. The polygon does get mighty close to the viewpoint, though; run the program and use the "T" key to move the polygon as close as possible--the near vertex swinging past provides a striking sense of perspective.

The performance of Listing Five is, perhaps, surprisingly good, clocking in at 16 frames per second on a 20-MHz 386 with a VGA of average speed and no 387, although there is, of course, only one polygon being drawn, rather than the hundreds or thousands we'd ultimately like. What's far more interesting is where the execution time goes. Even though the program is working with only one polygon, 73 percent of the time goes for transformation and projection. An additional 7 percent is spent waiting to flip the screen. Only 20 percent of the total time is spent in all other activity--and only 2 percent is spent actually drawing polygons. Clearly, we'll want to tackle transformation and projection first when we look to speed things up. (Note, however, that a math coprocessor would considerably decrease the time taken by floating-point calculations.)

In Listing Three, when the extent of the bounding rectangle is calculated for later erasure purposes, that extent is clipped to the screen. This is due to the lack of clipping in the rectangle fill code from July; the problem would more appropriately be addressed by putting clipping into the fill code, but, unfortunately, I lack the space to do that here.

Finally, observe the jaggies crawling along the edges of the polygon as it rotates. This is temporal aliasing at its finest! It'll be a while before we address antialiasing, real-time antialiasing being decidedly nontrivial, but this should give you an idea of why antialiasing is so desirable.

Coming Up

Next time, we'll assign fronts and backs to polygons, and start drawing only those that are facing the viewer. That will enable us to handle convex polyhedrons, such as tetrahedrons and cubes. We'll also look at interactively controllable rotation, and at more complex rotations than the simple rotation around the Y axis that we did this month. If there's room, we'll try moving the viewpoint, and perhaps we'll even use fixed-point arithmetic to speed things up. If not, patience; we'll get to all that and more (shading, hidden surfaces, maybe even a little rendering and antialiasing) soon.

- - - -

```
SC_INDEX equ    03c4h   ;Sequence Controller Index
MAP_MASK equ    2       ;Map Mask register index in SC

HLine   struc
XStart  dw      ?       ;X coordinate of leftmost pixel in line
XEnd    dw      ?       ;X coordinate of rightmost pixel in line
HLine   ends

HLineList struc
Lngth   dw      ?       ;# of horizontal lines
YStart  dw      ?       ;Y coordinate of topmost line
HLinePtr dw     ?       ;pointer to list of horz lines
HLineList ends

Parms   struc
                dw      2 dup(?) ;return address & pushed BP
HLineListPtr    dw      ?       ;pointer to HLineList structure
Color           dw      ?       ;color with which to fill
Parms   ends
        .model small
        .data
        extrn   _CurrentPageBase:word,_ClipMinX:word
        extrn   _ClipMinY:word,_ClipMaxX:word,_ClipMaxY:word
; Plane masks for clipping left and right edges of rectangle.
LeftClipPlaneMask       db      00fh,00eh,00ch,008h
RightClipPlaneMask      db      001h,003h,007h,00fh
        .code
        align   2
ToFillDone:
        jmp     FillDone
        public _DrawHorizontalLineList
        align   2
_DrawHorizontalLineList proc
        push    bp              ;preserve caller's stack frame
        mov     bp,sp           ;point to our stack frame
        push    si              ;preserve caller's register variables
        push    di
        cld                     ;make string instructions inc pointers
        mov     dx,SC_INDEX
        mov     al,MAP_MASK
        out     dx,al           ;point SC Index to the Map Mask
        mov     ax,SCREEN_SEGMENT
        mov     es,ax   ;point ES to display memory for REP STOS
        mov     si,[bp+HLineListPtr] ;point to the line list
        mov     bx,[si+HLinePtr] ;point to the XStart/XEnd descriptor
                                ; for the first (top) horizontal line
        mov     cx,[si+YStart]  ;first scan line to draw
        mov     si,[si+Lngth]   ;# of scan lines to draw
        cmp     si,0            ;are there any lines to draw?
        jle     ToFillDone      ;no, so we're done
        cmp     cx,[_ClipMinY]  ;clipped at top?
        jge     MinYNotClipped  ;no
        neg     cx              ;yes, discard however many lines are
        add     cx,[_ClipMinY]  ; clipped
        sub     si,cx           ;that many fewer lines to draw
        jle     ToFillDone      ;no lines left to draw
        shl     cx,1            ;lines to skip*2
        shl     cx,1            ;lines to skip*4
        add     bx,cx           ;advance through the line list
        mov     cx,[_ClipMinY]  ;start at the top clip line
MinYNotClipped:
        mov     dx,si
        add     dx,cx           ;bottom row to draw + 1
        cmp     dx,[_ClipMaxY]  ;clipped at bottom?
        jle     MaxYNotClipped  ;no
        sub     dx,[_ClipMaxY]  ;# of lines to clip off the bottom
        sub     si,dx           ;# of lines left to draw
        jle     ToFillDone      ;all lines are clipped
MaxYNotClipped:
        mov     ax,SCREEN_WIDTH/4 ;point to the start of the first
        mul     cx              ; scan line on which to draw
        add     ax,[_CurrentPageBase] ;offset of first line
        mov     dx,ax           ;ES:DX points to first scan line to
                                ; draw
```

```
        mov     ah,byte ptr [bp+Color] ;color with which to fill
FillLoop:
        push    bx              ;remember line list location
        push    dx              ;remember offset of start of line
        push    si              ;remember # of lines to draw
        mov     di,[bx+XStart]  ;left edge of fill on this line
        cmp     di,[_ClipMinX]  ;clipped to left edge?
        jge     MinXNotClipped  ;no
        mov     di,[_ClipMinX]  ;yes, clip to the left edge
MinXNotClipped:
        mov     si,di
        mov     cx,[bx+XEnd]    ;right edge of fill
        cmp     cx,[_ClipMaxX]  ;clipped to right edge?
        jl      MaxXNotClipped  ;no
        mov     cx,[_ClipMaxX]  ;yes, clip to the right edge
        dec     cx
MaxXNotClipped:
        cmp     cx,di
        jl      LineFillDone    ;skip if negative width
        shr     di,1    ;X/4 = offset of first rect pixel in scan
        shr     di,1    ; line
        add     di,dx   ;offset of first rect pixel in display mem
        mov     dx,si   ;XStart
        and     si,0003h                ;look up left edge plane mask
        mov     bh,LeftClipPlaneMask[si] ; to clip & put in BH
        mov     si,cx
        and     si,0003h                  ;look up right edge plane
        mov     bl,RightClipPlaneMask[si] ; mask to clip & put in BL
        and     dx,not 011b     ;calculate # of addresses across rect
        sub     cx,dx
        shr     cx,1
        shr     cx,1    ;# of addresses across rectangle to fill - 1
        jnz     MasksSet ;there's more than one byte to draw
        and     bh,bl   ;there's only one byte, so combine the left
                        ; and right edge clip masks
MasksSet:
        mov     dx,SC_INDEX+1 ;already points to the Map Mask reg
FillRowsLoop:
        mov     al,bh   ;put left-edge clip mask in AL
        out     dx,al   ;set the left-edge plane (clip) mask
        mov     al,ah   ;put color in AL
        stosb           ;draw the left edge
        dec     cx      ;count off left edge byte
        js      FillLoopBottom ;that's the only byte
        jz      DoRightEdge ;there are only two bytes
        mov     al,00fh ;middle addresses are drawn 4 pixels at a pop
        out     dx,al   ;set the middle pixel mask to no clip
        mov     al,ah   ;put color in AL
        rep     stosb   ;draw the middle addresses four pixels apiece
DoRightEdge:
        mov     al,bl   ;put right-edge clip mask in AL
        out     dx,al   ;set the right-edge plane (clip) mask
        mov     al,ah   ;put color in AL
        stosb           ;draw the right edge
FillLoopBottom:
LineFillDone:
        pop     si              ;retrieve # of lines to draw
        pop     dx              ;retrieve offset of start of line
        pop     bx              ;retrieve line list location
        add     dx,SCREEN_WIDTH/4 ;point to start of next line
        add     bx,size HLine   ;point to the next line descriptor
        dec     si              ;count down lines
        jnz     FillLoop
FillDone:
        pop     di              ;restore caller's register variables
        pop     si
        pop     bp              ;restore caller's stack frame
        ret
_DrawHorizontalLineList endp
        end
```

[LISTING TWO]

```
/* Matrix arithmetic functions.
   Tested with Borland C++ 2.0 in the small model */

/* 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, as follows:
   --     --     -- --     -- --
   |       |     | 4 |     | 4 |
   |  4x4  |  X  | x |  =  | x |
   |       |     | 1 |     | 1 |
   --     --     -- --     -- -- */
void XformVec(double Xform[4][4], double * SourceVec,
   double * DestVec)
{
   int i,j;

   for (i=0; i<4; i++) {
      DestVec[i] = 0;
      for (j=0; j<4; j++)
         DestVec[i] += Xform[i][j] * SourceVec[j];
   }
}

/* Matrix multiplies SourceXform1 by SourceXform2 and stores the
   result in DestXform. Multiplies a 4x4 matrix times a 4x4 matrix;
   the result is a 4x4 matrix, as follows:
   --     --     --     --     --     --
   |       |     |       |     |       |
   |  4x4  |  X  |  4x4  |  =  |  4x4  |
   |       |     |       |     |       |
   --     --     --     --     --     -- */
void ConcatXforms(double SourceXform1[4][4], double SourceXform2[4][4],
   double DestXform[4][4])
{
   int i,j,k;

   for (i=0; i<4; i++) {
      for (j=0; j<4; j++) {
         DestXform[i][j] = 0;
         for (k=0; k<4; k++)
            DestXform[i][j] += SourceXform1[i][k] * SourceXform2[k][j];
      }
   }
}
```

[LISTING THREE]

```
/* Transforms convex polygon Poly (which has PolyLength vertices),
   performing the transformation according to Xform (which generally
   represents a transformation from object space through world space
   to view space), then projects the transformed polygon onto the
   screen and draws it in color Color. Also updates the extent of the
   rectangle (EraseRect) that's used to erase the screen later.
   Tested with Borland C++ 2.0 in the small model */
#include "polygon.h"

void XformAndProjectPoly(double Xform[4][4], struct Point3 * Poly,
   int PolyLength, int Color)
{
   int i;
   struct Point3 XformedPoly[MAX_POLY_LENGTH];
   struct Point ProjectedPoly[MAX_POLY_LENGTH];
   struct PointListHeader Polygon;

   /* Transform to view space, then project to the screen */
   for (i=0; i<PolyLength; i++) {
      /* Transform to view space */
```

```
      XformVec(Xform, (double *)&Poly[i], (double *)&XformedPoly[i]);
      /* Project the X & Y coordinates to the screen, rounding to the
         nearest integral coordinates. The Y coordinate is negated to
         flip from view space, where increasing Y is up, to screen
         space, where increasing Y is down. Add in half the screen
         width and height to center on the screen */
      ProjectedPoly[i].X = ((int) (XformedPoly[i].X/XformedPoly[i].Z *
            PROJECTION_RATIO*(SCREEN_WIDTH/2.0)+0.5))+SCREEN_WIDTH/2;
      ProjectedPoly[i].Y = ((int) (XformedPoly[i].Y/XformedPoly[i].Z *
            -1.0 * PROJECTION_RATIO * (SCREEN_WIDTH / 2.0) + 0.5)) +
            SCREEN_HEIGHT/2;
      /* Appropriately adjust the extent of the rectangle used to
         erase this page later */
         if (ProjectedPoly[i].X > EraseRect[NonDisplayedPage].Right)
          if (ProjectedPoly[i].X < SCREEN_WIDTH)
            EraseRect[NonDisplayedPage].Right = ProjectedPoly[i].X;
          else EraseRect[NonDisplayedPage].Right = SCREEN_WIDTH;
         if (ProjectedPoly[i].Y > EraseRect[NonDisplayedPage].Bottom)
          if (ProjectedPoly[i].Y < SCREEN_HEIGHT)
            EraseRect[NonDisplayedPage].Bottom = ProjectedPoly[i].Y;
          else EraseRect[NonDisplayedPage].Bottom = SCREEN_HEIGHT;
         if (ProjectedPoly[i].X < EraseRect[NonDisplayedPage].Left)
          if (ProjectedPoly[i].X > 0)
            EraseRect[NonDisplayedPage].Left = ProjectedPoly[i].X;
          else EraseRect[NonDisplayedPage].Left = 0;
         if (ProjectedPoly[i].Y < EraseRect[NonDisplayedPage].Top)
          if (ProjectedPoly[i].Y > 0)
            EraseRect[NonDisplayedPage].Top = ProjectedPoly[i].Y;
          else EraseRect[NonDisplayedPage].Top = 0;
   }
   /* Draw the polygon */
   DRAW_POLYGON(ProjectedPoly, PolyLength, Color, 0, 0);
}
```

[LISTING FOUR]

```
/* 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 the 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 (used to describe a scan-converted polygon to the
   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; };

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 int DisplayedPage, NonDisplayedPage;
extern struct Rect EraseRect[];
```

[LISTING FIVE]

```
/* Simple 3D drawing program to view a polygon as it rotates in
   mode X. View space is congruent with world space, with the
   viewpoint 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.
   Tested with Borland C++ 2.0 in the small model */
#include <conio.h>
#include <stdio.h>
#include <dos.h>
#include <math.h>
#include "polygon.h"
void main(void);

/* 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} };
/* Transformation from polygon's object space to world space.
   Initially set up to perform no rotation and to move the polygon
   into world space -140 units away from the origin down the Z axis.
   Given the viewing point, -140 down the Z axis means 140 units away
   straight ahead in the direction of view. The program dynamically
   changes the rotation and translation */
static double PolyWorldXform[4][4] = {
   {1.0, 0.0, 0.0, 0.0},
   {0.0, 1.0, 0.0, 0.0},
   {0.0, 0.0, 1.0, -140.0},
   {0.0, 0.0, 0.0, 1.0} };
```

```
/* Transformation from world space into view space. In this program,
   the view point is fixed at the origin of world space, looking down
   the Z axis in the direction of increasingly negative Z, so view
   space is identical to world space; 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}
};
static unsigned int PageStartOffsets[2] =
   {PAGE0_START_OFFSET,PAGE1_START_OFFSET};
int DisplayedPage, NonDisplayedPage;

void main() {
   int Done = 0;
   double WorkingXform[4][4];
   static struct Point3 TestPoly[] =
         {{-30,-15,0,1},{0,15,0,1},{10,-5,0,1}};
#define TEST_POLY_LENGTH   (sizeof(TestPoly)/sizeof(struct Point3))
   double Rotation = M_PI / 60.0; /* initial rotation = 3 degrees */
   union REGS regset;

   Set320x240Mode();
   ShowPage(PageStartOffsets[DisplayedPage = 0]);
   /* Keep rotating the polygon, drawing it to the undisplayed page,
      and flipping the page to show it */
   do {
      CurrentPageBase =    /* select other page for drawing to */
            PageStartOffsets[NonDisplayedPage = DisplayedPage ^ 1];
      /* Modify the object space to world space transformation matrix
         for the current rotation around the Y axis */
      PolyWorldXform[0][0] = PolyWorldXform[2][2] = cos(Rotation);
      PolyWorldXform[2][0] = -(PolyWorldXform[0][2] = sin(Rotation));
      /* Concatenate the object-to-world and world-to-view
         transformations to make a transformation matrix that will
         convert vertices from object space to view space in a single
         operation */
      ConcatXforms(WorldViewXform, PolyWorldXform, WorkingXform);
      /* 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;
      /* Transform the polygon, project it on the screen, draw it */
      XformAndProjectPoly(WorkingXform, TestPoly, TEST_POLY_LENGTH,9);
      /* Flip to display the page into which we just drew */
      ShowPage(PageStartOffsets[DisplayedPage = NonDisplayedPage]);
      /* Rotate 6 degrees farther around the Y axis */
      if ((Rotation += (M_PI/30.0)) >= (M_PI*2)) Rotation -= M_PI*2;
      if (kbhit()) {
         switch (getch()) {
            case 0x1B:     /* Esc to exit */
               Done = 1; break;
            case 'A': case 'a':      /* away (-Z) */
               PolyWorldXform[2][3] -= 3.0; break;
            case 'T':      /* towards (+Z). Don't allow to get too */
            case 't':      /* close, so Z clipping isn't needed */
               if (PolyWorldXform[2][3] < -40.0)
                     PolyWorldXform[2][3] += 3.0; break;
            case 0:     /* extended code */
               switch (getch()) {
                  case 0x4B:  /* left (-X) */
                     PolyWorldXform[0][3] -= 3.0; break;
                  case 0x4D:  /* right (+X) */
                     PolyWorldXform[0][3] += 3.0; break;
                  case 0x48:  /* up (+Y) */
                     PolyWorldXform[1][3] += 3.0; break;
                  case 0x50:  /* down (-Y) */
```

```
                    PolyWorldXform[1][3] -= 3.0; 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);
}
```

Journal: Dr. Dobb's Journal Feb 1992 v17 n2 p125(8)

More 3-D animation

Author: Abrash, Michael.
AttFile: 3D.EXE Self extracting archive.
GP-FEB92.ASC Source code listing.

Abstract: Computer animation, especially three-dimensional animation, is as much the art of illusion as a mathematical procedure. The programmer must compensate for such imperfections as 'jaggies' and flicker which the viewer would notice, but others are 'deleted' by the human brain which makes certain assumptions when viewing a scene. One-sided polygons, drawn as if they were not visible to the viewer in two-dimensional space, are discussed. Removing polygons that are not facing the viewer would solve all 'hidden surface' problems for a single convex polyhedron, but overlapping polyhedrons require other methods such as determining the cross-product of two vectors. Sample programs that perform backface removal, incremental transformation and other animation algorithms are presented.

As I'm fond of pointing out, computer animation isn't a matter of mathematically exaxt 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) of their own. To make this magic happen, we must provide cues for the eyes 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.

And yet, in some senses, 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 that 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 imperfetions 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, witout 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 request for real-time 3-D animation on the PC.

One-sided Polygons: Backface Removal

Last month, 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 last month's approach was that the polygon had two visible sides. Why is that a drawback? Well, it isn't, necessarily, but in our case we want to use polygons to build solid objects with continuous surfaces, and in the context, only one side of a polygon is ever 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 only for convex polyhedrons, and only if there's a single convex polyhederon involved; when convex polyhedrons overlap, other methods must be used. Nonetheless, backface removal does instantly halve the number of polygons to be handled in rednering any particular scene. Backface removal can also speed hidden-surface handling if objects are built out of convex polyhedrons, as we'll see in a future column. This month, though, we have only one convex polyhedron to deal wit, so backface removal alone will do the trick.

Give that I've convinced you that backface removal would be a handy thing to have, how do we actually do it? A it? A logical approach, often implemented in 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 1. One interesting property of the cross-product vector is that it is perpendicular to the plane in which the two original vector 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 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 2, and it 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.

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 Three, backface removal is performed in screen coordinates in the interest of speed.

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

Backface Removal in Action

Listings One through Five together form a program that rotates a solid cube in real time under user control. Listing One (page 147) is the main program; Listing Two (page 147) performs transformation and projection; Listing Three (page 147) performs backface removal and draws visible faces; Listing Four (page 148) concatenates incremental rotations to the object-to-world transformation matrix; Listing Five (page 150) is the general header file. Also required from previous columns are: Listings One and Two from last month (draw clipped line list, matrix math functions); Listings One and Six from July 1991, (mode X mode set, rectangle fill); Listing Six from September 1991; Listing Four from March 1991 (polygon edge scan); and the FillConvexPolygonO function from Listing One from February 1991. All necessary modules, along with a make file, will be available as part of the code from this issue.

The sample program places a cube, floating in three-space, under the complete control of the user. The arrow keys may be used to move the 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 ahve 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 3. 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 flaf 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 visible improvement, because there's only the one object, but the underlying mechanism is 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 387). 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 Four contains three functions; each concatenates an additional rotation around one of the three axes to an existing rotation. In order 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 last month). This cuts the number of floating-point multiplies from the 64 required for the multiplication of two 4X4 matrices to just 12, and floating point adds from 48 to 6.

Be aware that Listing Four 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 Four 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 precisionis 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.

A Note on Rounding Negative Numbers

Last month, I added 0.5 and truncated in order to round values from floating-point to integer format. In Listing Two this month, I've switched to adding 0.5 and using the floorO function. For positive values, the two approahces are equivalent; for negative values, only the floorO 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, in order 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 altogeher. 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.

Coming up: shading, hidden surfaces, and performance.

- -

GRAPHICS PROGRAMMING COLUMN
by Michael Abrash

[LISTING ONE]

```
/* 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++ 2.0 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 TWO]

```
/* 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 THREE]

```
/* 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);
      }
   }
}
```

[LISTING FOUR]

```
/* 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 FIVE]

```
/* 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[];
```

Journal: Dr. Dobb's Journal March 1992 v17 n3 p119(7)

Fast 3-D animation: meet X-Sharp

Author: Abrash, Michael.
AttFile: GP-MAR92.ASC Source code listing.

Abstract: Three elements come into play to enable 12 three-dimensional graphics cubes to rotate at an update rate of about 15 frames per second on a 20-MHz 386-based machine with a slow video graphics array. One of them is fixed-point arithmetic, which provides the graphics with an immediate order-of-magnitude performance boost. The second element is the use of the 386-based machine's 32-bit multiply and divide instructions. The third performance enhancement key is to maintain and operate on only the relevant sections of the transformation coordinates and matrices. The techniques are very powerful and can cause some problems. For example, since points outside of the 64K by 64K by 64K space cannot be handled, multiple matrix concatenations may turn out to be imprecise.

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. This summer, 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 grades 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, 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. This month, it's time for 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 animation. Unfortunately, they take a lot of code,

so I'll have to keep the text short. Therefore, without further ado, I give you real real-time animation.

Fixed Point, Native 386, and More

As of last month, we were at the point where we could rotate, move, and draw a solid cube in real time. This month's program, shown in Listings One through Ten (pages 134 through 138), 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 this month's performance are three. The first key is fixed-point arithmetic. In the last two months, we've 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 slow floating-point processor operations or excruciatingly slow floating-point emulator operations. Although the speed advantage of fixed-point varies depending on the operation, the processor, and whether 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. Real-mode C compilers, such as Borland C++ and Microsoft C, 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 Nine -- 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 (at least for the near future) 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 X 3 rotation matrix and a 3 X 1 translation vector, and coordinates are treated as 3 X 1 vectors. This saves a great many multiplications in the course of transforming each point.

Just for fun, I reimplemented the animation of Listings One through Ten with floating-point instructions. Together, the preceeding optimizations improve the performance of the entire animation -- including drawing time and overhead, 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 assembler 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 One through Ten, although the polygon fill code is of course mostly in assembler; 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 turbo-charge 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 I may just do that one of these months, especially if enough of you give me a hard time for taking the easy way out with 386 instructions), 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 X 64K X 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 now, but here are some brief thoughts. The working 64K X 64K X 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). 3-D clipping with a front clip plane of -1 or less can prevent divide overflow.

A New Animation Framework

Listings One through Ten represent not merely faster animation, but also a mostly complete, extensible, data-driven animation framework. Where earlier animation code was hardwired to demonstrate certain concepts, this month'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 Eight.

The animation framework is not yet complete. Movement is supported only along the Z axis, and then in a non-general fashion. More interesting movement isn't supported at this point because of the one gaping hole in the package: hidden-surface removal. Until this is implemented -- and it will be, soon -- 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 this month's code, anyway.

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() are costly. Much of this is correctable with a little carefully crafted assembly language and a lookup table; expect that soon. When all that is firmly in place, we'll take a look at the number of pixels being drawn versus the bandwidth of display memory; that'll give us an idea of how close we are to the theoretical limit of VGA animation. Probably not too close, even with those optimizations; a faster 2-D clipping approach and still faster polygon-fill code will most likely be in order. (Yes, Virginia, there is an even faster way to fill polygons!)

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

X-Sharp

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. There's no way all of that could fit in a single column, and in fact I've been building toward a critical mass of animation functionality since my very first column in DDJ. This month builds on code from no less than five previous columns. The code that's required in order to link this month's animation program is: Listing One from January (draw clipped line list); Listings One and Six from July 1991 (mode X mode set, rectangle fill);

Listing Six from September 1991; Listing Four in March 1991 (polygon edge scan); and the FillConvexPolygon() function from Listing One from February 1991. The struct keywords in FillConvexPolygon() must be removed to reflect the switch to typedefs in the animation header file.

This is the last time I'm going to list all the code needed to build the animation package, and I am not going to print every change to every module from now on. The code is simply getting too large to show every bit of it, and the scope is only going to grow as I add functions such as shading, 3-D clipping, and hidden surfaces; also, I'm going to be tinkering with stuff such as converting key code to assembler and modifying functions slightly as structures change, and I hate to take up valuable space in DDJ for what's basically fine-tuning and housekeeping.

I think we've reached the point where we can call this an ongoing project and give it a name. In the spirit of Al Stevens' wonderful D-Flat, I hereby dub the animation package X-Sharp. (X for mode X, sharp because who wants a flat animation package?) From now on, I will make the full source for X-Sharp available, complete with make files, online, and otherwise. It will be available in the file XSHARPn.ARC in the DDJ Forum on CompuServe and on M&T Online. Alternatively, you can send me a 360K or 720K formatted diskette and an addressed, stamped diskette mailer, care of DDJ (411 Borel Ave., San Mateo, CA 94403-3522), and I'll send you the latest copy of X-Sharp. There's no charge, but, in the spirit of Al Stevens's "careware," it'd be very much appreciated if you'd slip in a dollar or so for the folks at the Vermont Association for the Blind and Visually Impaired, who help the visually impaired build productive, self-sufficient lives, and are amazingly successful at it. Imagine for a moment trying to do your work if you lost your sight (and it can be done; I got a request for the text of my book Zen of Assembly Language in computer-readable form from a blind programmer the other day)--heck, imagine trying to cross the street--and I suspect you'll understand why it matters so much. As Al says, it's purely voluntary, but both you and I will feel good.

I'm available on a daily basis to discuss X-Sharp on M&T Online and Bix (user name MABRASH in both cases).

- - - -

```
   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 TWO]

```
/* 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 THREE]

```
/* 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 FOUR]

```
/* 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 FIVE]

```
/* 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 SIX]

```
/* 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 SEVEN]

```
/* 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 EIGHT]

```
/* 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 NINE]

```
; 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 3.0.
        .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 TEN]

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

Journal: Dr. Dobb's Journal April 1992 v17 n4 p139(7)

Raw speed and more

Author: Abrash, Michael.
AttFile: GP-APR92.ASC Source code listing.

Abstract: X-Sharp three-dimensional (3-D) software has assembly language added for performance improvement, and for matrix math functions. Several functions can be eliminated by converting them to pure assembly language, such as the function to multiply a matrix by a vector, and the function to concatenate matrices. Depth sorting is useful for handling hidden surfaces, however it does not address the issue of nonconvex objects. Convex polyhedrons are addressed. Fast microcomputer-based animation is not seriously affected by problems with depth sorting, problems that can affect photo-realistic rendering to a greater extent. Features that are still to be added to X-Sharp include shading, antialiasing, support for non-convex objects, 3-D clipping, texture mapping and performance improvement.

As usual, there's an awful lot to cover this month, so I'll have to make this quick. That's a shame, because this absolutely true story is even lovelier in the long version, as told by--well, I promised I wouldn't even hint at who he is, for reasons that will soon be obvious.

Years ago, this friend of mine--let's call him Bert--went to Hawaii with three other fellows to celebrate their graduation from high school. This was an unchaperoned trip, and they behaved pretty much as responsibly as you'd expect four teenages to behave, which is to say, not; there's a story about a rental car that, to this day, Bert can't bring himself to tell. They had a good time, though, save for one thing: no girls.

By and by, they met a group of girls by the pool, but the boys couldn't get past the hi-howya-doin stage, so they retired to their hotel room to plot a better approach. This being the early '70s, and them being slightly tipsy teenagers with raging hormones and the effective combined IQ of four eggplants, it took them no time at all to come up with a brilliant plan: streaking. The girls had mentioned their room number, so the boys piled into the elevator, pushed the button for the girls' floor, shucked their clothes as fast as they could, and sprinted to the girls' door. They knocked on the door and ran on down the hall. As the girls opened their door, Bert and his crew raced past, toward the elevator, laughing hysterically.

Bert was by far the fastest of them all. He whisked between the elevator doors just as they started to close; by the time his friends got there, it was too late, and the doors slid shut in their faces. As the elevator began to move, Bert could hear the frantic pounding of six fists thudding on the closed doors. As Bert stood among the clothes littering the elevator floor, the thought of his friends stuck in the hall, naked as jaybirds, was just too much, and he doubled over with helpless laughter, tears streaming down his face. The universe had blessed him with one of those exceedingly rare moments of perfect timing and execution.

The universe wasn't done with Bert quite yet, though. He was still contorted with laughter--and still quite thoroughly undressed--when the elevator doors opened again. On the lobby.

And with that, we come to this month's topics, raw speed and hidden surfaces.

Raw Speed, Part I: Assembly Language

I would like to state, here and for the record, that I am not an assembly language fanatic. Frankly, I prefer programming in C; assembly language is hard work, and I can get a whole lot more done with fewer hassles in C. However, I am a performance fanatic, performance being defined as having programs be as nimble as possible in those areas where the user wants fast response. And, in the course of pursuing performance, there are times when a little assembly language goes a long way.

We're now in the fourth month of working on the X-Sharp #-D animation package. In real-time animation, performance is sine qua non--Latin for "Make it fast or find another line of work"--so some judiciously applied assembly language is in order. Last month, we got up to a serviceable performance level by switching to fixed-point math, then implementing the fixed-point multiplication and division functions in assembler in order to take advantage of the 386's 32-bit capabilities. There's another area of the program that fairly cries out for assembly language: matrix math. The function to multiply a matrix by a vector (XformVec()) and the function to concatenate matrices (ConcatXforms()) both loop heavily around calls to FixedMul(); a lot of calling and looping can be eliminated by converting these functions to pure assembly language.

Listing One, page 157, is the module FIXED.ASM from the current version of X-Sharp, with XformVec() and ConcatXforms() implemented in assembly language. The code is heavily optimized, to the extent of completely unrolling the loops via macros so that looping is eliminated althogether. FIXED.ASM is highly effective; the time taken for matrix math is now down to the point where it's a fairly minor component of execution time, representing less than ten percent of the total. It's time to turn our optimization sights elsewhere.

Raw Speed, Part II: Look it Up

It's a funny thing about Turbo Profiler: Time spent in the Borland C++ 80x87 emulator doesn't show up directly anywhere that I can see in the timing results. The only way to detect it is by way of the line that reports what percent of total time is represented by all the areas that were profiled; if you're profiling all areas, whatever's not explicitly accounted for seems to be the floating-point emulator time. This quirk fooled me for a while, leading me to think sine and and cosine weren't major drags on performance, because the sin() and cos() functions spend most of their time in the emulator, and that time doesn't show up in Turbo Profiler's statistics on those functions. Once I figured out what was going on, it turned out that not only were sin() and cos() major drags, they were taking up over half the total execution time by themselves.

The solution is a lookup table. Listing One contains a function called CosSin() that calculates both the sine and cosine of an angle, via a lookup table. The function accepts agnles in tenths of degrees; I decided to use tenths of degrees rather than radians because that way it's always possible to look up the sine and cosine of the exact angle requested, rather than approximating, as would be required with radians. Tenths of degrees should be fine enough control for most purposes; if not, it's easy to alter CosSin() for finer gradations yet. GENCOS.C, the program used to generate the lookup table (COSTABLE.INC), included in Listing One, can be found in the X-Sharp archive. GENCOS.C can generate a cosine table with any integral number of steps per degree.

FIXED.ASM speeds X-Sharp up quite a bit, and it changes the performance balance a great deal. When we started out with 3-D animation, calculation time was the dragon we faced; more than 90 percent of the total time was spent doing matrix and projection math. Additional optimizations in the area of math could still be made (using 32-bit multiplies in the backface-removal code, for example), but fixed-point math, the sine and cosine lookup, and selective assembler optimizations have done a pretty good job already. The bulk of the time taken by X-Sharp is now spent drawing polygons, drawing rectangles (to erase objects), and waiting for the page to flip. In other words, we've slain the dragon of 3-D math, or at least wounded it grievously; now we're back to the dragon of polygon filling. We'll address faster polygon filling soon, but for the moment, we have more than enough horsepower to have some fun with. First, though, we need one more feature: hidden surfaces.

Hidden Surfaces

So far, we've made a number of simplifying assumptions in order to get the animation to look good; for example, all objects must currently be convex polyhedrons. We'll deal with that one down the road a little, but first we have to address a still more fundamental limitation, which is that right now,

objects can never pass behind or in front of each other. In short, it's time to have a look at hidden surfaces.

There are a passel of ways to do hidden surfaces. Way off at one end (the slow end) of the spectrum is z-buffering, whereby each pixel of each polygon is checked as it's drawn to see whether it's the frontmost version of the pixel at those coordinates. At the other end is the technique of simply drawing the objects in back-to-front order, so that nearer objects are drawn on top of farther objects. The latter approach, depth sorting, is the one we'll take today. (Actually, true depth sorting involves detecting and resolving possible ambiguities when objects overlap in z; simply sorting on z, which we'll be doing this month, is more precisely known as the "painter's algorithm.")

Depth sorting is fast but less than perfect. For one thing, it doesn't address the issue of nonconvex objects, so we'll have to stick with convex polyhedrons for now. For another, there's the question of what part of each object to use as the sorting key; the nearest point, the center, and the farthest point are all possibilities--and, whichever point is used, depth sorting doesn't handle some overlap cases properly. Figure 1 illustrates one case in which back-to-front sorting doesn't work, regardless of what point is used as the sorting key.

For photo-realistic rendering, these are serious problems.

For fast PC-based animation, however, they're manageable. Choose objects that aren't too elongated; arrange their paths of travel so they don't intersect in problematic ways; and, if they do overlap incorrectly, trust that the glitch will be lost in the speed of the animation and the complexity of the screen.

Listing Two, page 159, shows the key routines for depth sorting, from the X-Sharp file OLIST.C. Objects are now stored in a linked list. The initial, empty list, created by InitializeObjectList(), consists of a sentinel entry at either end, one at the farthest possible z coordinate, and one at the nearest. New entries are inserted by AddObject() in z-sorted order. Each time the objects are moved, before they're drawn at their new locations, SortObjects() is called to z-sort the object list, so that drawing wil proceed from back to front. The z-sorting is done on the basis of the objects' center points; a center-point filed has been added to the object structure to support this, and the center point for each object is now transformed along with the vertices. That's really all there is to depth sorting--and now we can have objects that overlap in x and y.

I wish I could reproduce the lastes X-Sharp demonstration program here; I really do. It's a lot of fun, with 11 spinning cubes and a whirling faceted ball bouncing back and forth between the screen and deep space at varying speeds. The number of faces has nearly doubled since last month, to 138, and the number of vertices is up to 150, but thanks to FIXED.ASM, the animation is snappier than ever, and of course there's a much-enhanced sense of depth, thanks to the visual cues made possible by depth sorting. We've reached the point of genuinely dynamic animation on a 386, even with a slowpoke VGA; my daughter thinks it's a little scary having those things hurtling toward her so fast. On a 486 with a fast (Tseng ET-4000) VGA, the animation is too fast; the illusion of reality is broken. We should all have such problems.

I'd like to list the demo program in its entirety, but I can't. As I explained last month, X-Sharp is way too large, over 2000 lines now; even the changes since last month would blow my page budget out of the water. Instead, I've made the full source available in the file XSHARPn.ARC in the DDJ Forum on CompuServe, on M&T Online, and in the graphic.disp conference on Bix. Alternatively, you can send me a 360K or 720K formatted diskette and an addressed, stamped diskette mailer, care of DDJ, 411 Borel Ave., San Mateo, CA 94402, and I'll send you the latest copy of X-Sharp. There's no charge, but it'd be very much appreciated if you'd slip in a dollar or so to help out the folks at the Vermont Association for the Blind and Visually Impaired. It's not every day that you can make a difference so easily--and, believe me, their work does make a difference.

I'm available on a daily basis to discuss X-Sharp on M&T Online and Bix (user name mabrash in both cases).

Rounding

FIXED.ASM containes the equate ROUNDING[underscore]ON. When this equate is 1, the results of multiplications and divisions are rounded to the nearest fixed-point values; when it's 0, the results are truncated. The difference between the results produced by the two approaches is, at most, [2.sup.-16]; you wouldn't think that would make much difference, now, would you? But it does. When the animation is run with rounding disabled, the cubes start to distort visibly after a few minutes, and after a few minutes more they look like they've been run over. In contrast, I've never seen any significant distortion with rounding on, even after a half-hour or so. I think the difference with rounding is not that it's so much more accurate, but rather that the errors are evenly distributed; with truncation, the errors are biased, and biased errors become very visible when they're applied to right-angle objects. Even with rounding, though, errors will eventually creep in, and reorthogonalization will become necessary at some point. We'll have a look at that soon.

The performance cost of rounding is small, and the benefits are highly visible. Still, truncation errors become significant only when they accumulate over time, as, for example, when rotation matrices are repeatedly concatenated over the course of many transformations. Some time could be saved by rounding only in such cases. For example, division is performed only in the course of projection, and the results do not accumulate over time, so it would be reasonable to disable rounding for division.

Having a Ball

For three months, we've had nothing to look at but triangles and cubes. It's time for something a little more visually appealing, so the demonstration program now features a 72-sided ball. What's particularly interesting about this ball is that it's created by the GENBALL.C program in the BALL subdirectory of X-Sharp, and both the size of the ball and the number of bands of faces are programmable. GENBALL.C spits out to a file all the arrays of vertices and faces needed to create the ball, ready for inclusion in INITBALL.C. True, if you change the number of bands, you must change the Color array in INITBALL.C to match, but that's a tiny detail; by and large, the process of generating a ballshaped object is now automated. In fact, we're not limited to ball-shaped objects; substitute a different vertex and face generation program for GENBALL.C, and you can make whatever convex polyhedron you want; again, all you have to do is change the Color array correspondingly. You can easily create multiple versions of the base object, too; INITCUBE.C is an example of this, creating 11 different cubes.

What we have here is the first glimmer of an object-editing system. GENBALL.C is the prototype for object definition, and INITBALL.C is the prototype for general-purpose object instantiation. Certainly, it would be nice to have an interactive 3-D object editing tool and resource management setup, and perhaps we'll do that someday. We have our hands full with the drawing end of things at the moment, though, and for now it's enough to be able to create objects in a semiautomated way.

Eating Crow and Other Delicacies

Back in November, I said, "Hicolor programming unavoidably requires handling broken rasters" (Hicolor mode features 32K colors, by means of the Sierra Hicolor DAC combined with a Hicolor-capable Super-VGA.) Any absolute assertion on my part seems to be the cue for readers to come swarming out of the woodworks with examples to the contrary. In this case, M&T Online user rfredrick was quick to point out that setting the bitmap width--th offset from the start of one line to the start of the next--to 1928 bytes in 640X480 Hicolor mode eliminates broken rasters (displayed lines that span banks). A bitmap width of 1928 is selected by setting the Row Offset register (CRTC register 13h) to 241, like so: outpw(0x3d4, 0x13 \ (241 << 8));. Once the width is set, all you have to do is use 1928 for the offset from one row to the next, rather than 1280, and you're all set. Actually, the breaks are still there, but they can be ignored because they happen off to the right of the displayed portion of the bitmap. To get the Hicolor drawing code from the November 1991 column to support 1928-wide Hicolor bitmaps, just change BitmapWidthInBytes from 640*2 to 1928.

rfrederick credits this information to the folks at Everex. Thanks to all involved for passing it along.

Coming Up

There are a lot of wonderful things to add to X-Sharp, including shading of many sorts, antialiasing, support for non-convex objects, faster polygon drawing and clipping, 3-D clipping, texture mapping, still better performance--you get the idea. Personally, I'm excited as all get-out about this stuff, and I'll certainly cover more of it in the near future. Graphics is a broad field, though, and you folks have diverse interests; I'd like to hear what you're interested in seeing in this space. More 3-D animation? More animation of other types? Programming techniques for PC graphics hardware, such as 24-bpp VGAs, the S3 Super-VGA accelerator, and XGA? JPEG? Graphics operations such as seed fills and fast (and I do mean fast!) line drawing? Color mapping to a 256-color palette? Dithering? Something else?

It's a big list, because this is a big--and tremendously exciting--field. So long as there's code to be written for a topic (after all, this is the Graphics Programming column), I'm game. Let me know what you'd like to see.

- - - -

```
FDparms ends
        align   ALIGNMENT
        public  _FixedDiv
_FixedDiv       proc    near
        push    bp
        mov     bp,sp

if ROUNDING_ON
        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 half as
        adc     eax,0           ; 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:
else ;!ROUNDING_ON
        mov     edx,[bp+Dividend]
        sub     eax,eax
        shrd    eax,edx,16      ;position so that result ends up
        sar     edx,16          ; in EAX
        idiv    dword ptr [bp+Divisor]
endif ;ROUNDING_ON
        shld    edx,eax,16      ;whole part of result in DX;
                                ; fractional part is already in AX
        pop     bp
        ret
_FixedDiv       endp
;=============================================================================
; Returns the sine and cosine of an angle.
; C near-callable as:
;       void CosSin(TAngle Angle, Fixedpoint *Cos, Fixedpoint *);

        align   ALIGNMENT
CosTable label dword
        include costable.inc

SCparms struc
        dw      2 dup(?)        ;return address & pushed BP
Angle   dw      ?               ;angle to calculate sine & cosine for
Cos     dw      ?               ;pointer to cos destination
Sin     dw      ?               ;pointer to sin destination
SCparms ends

        align   ALIGNMENT
        public _CosSin
_CosSin proc    near
        push    bp              ;preserve stack frame
        mov     bp,sp           ;set up local stack frame

        mov     bx,[bp].Angle
        and     bx,bx           ;make sure angle's between 0 and 2*pi
        jns     CheckInRange
MakePos:                        ;less than 0, so make it positive
```

```
        add     bx,360*10
        js      MakePos
        jmp     short CheckInRange

        align   ALIGNMENT
MakeInRange:                    ;make sure angle is no more than 2*pi
        sub     bx,360*10
CheckInRange:
        cmp     bx,360*10
        jg      MakeInRange

        cmp     bx,180*10       ;figure out which quadrant
        ja      BottomHalf      ;quadrant 2 or 3
        cmp     bx,90*10        ;quadrant 0 or 1
        ja      Quadrant1
                                ;quadrant 0
        shl     bx,2
        mov     eax,CosTable[bx] ;look up sine
        neg     bx              ;sin(Angle) = cos(90-Angle)
        mov     edx,CosTable[bx+90*10*4] ;look up cosine
        jmp     short CSDone

        align   ALIGNMENT
Quadrant1:
        neg     bx
        add     bx,180*10       ;convert to angle between 0 and 90
        shl     bx,2
        mov     eax,CosTable[bx] ;look up cosine
        neg     eax             ;negative in this quadrant
        neg     bx              ;sin(Angle) = cos(90-Angle)
        mov     edx,CosTable[bx+90*10*4] ;look up cosine
        jmp     short CSDone

        align   ALIGNMENT
BottomHalf:                     ;quadrant 2 or 3
        neg     bx
        add     bx,360*10       ;convert to angle between 0 and 180
        cmp     bx,90*10        ;quadrant 2 or 3
        ja      Quadrant2
                                ;quadrant 3
        shl     bx,2
        mov     eax,CosTable[bx] ;look up cosine
        neg     bx              ;sin(Angle) = cos(90-Angle)
        mov     edx,CosTable[90*10*4+bx] ;look up sine
        neg     edx             ;negative in this quadrant
        jmp     short CSDone

        align   ALIGNMENT
Quadrant2:
        neg     bx
        add     bx,180*10       ;convert to angle between 0 and 90
        shl     bx,2
        mov     eax,CosTable[bx] ;look up cosine
        neg     eax             ;negative in this quadrant
        neg     bx              ;sin(Angle) = cos(90-Angle)
        mov     edx,CosTable[90*10*4+bx] ;look up sine
        neg     edx             ;negative in this quadrant
CSDone:
        mov     bx,[bp].Cos
        mov     [bx],eax
        mov     bx,[bp].Sin
        mov     [bx],edx

        pop     bp              ;restore stack frame
        ret
_CosSin endp
;======================================================================
; 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 the
; bottom row of the matrix is 0 0 0 1, and doesn't bother to set
; the W coordinate of the destination.
; C near-callable as:
;       void XformVec(Xform WorkingXform, Fixedpoint *SourceVec,
```

```
;               Fixedpoint *DestVec);
;
; This assembly code is equivalent to this C code:
;   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 */

XVparms struc
        dw      2 dup(?)        ;return address & pushed BP
WorkingXform dw ?               ;pointer to transform matrix
SourceVec dw    ?               ;pointer to source vector
DestVec dw      ?               ;pointer to destination vector
XVparms ends

        align   ALIGNMENT
        public _XformVec
_XformVec       proc    near
        push    bp              ;preserve stack frame
        mov     bp,sp           ;set up local stack frame
        push    si              ;preserve register variables
        push    di

        mov     si,[bp].WorkingXform ;SI points to xform matrix
        mov     bx,[bp].SourceVec    ;BX points to source vector
        mov     di,[bp].DestVec      ;DI points to dest vector

soff=0
doff=0
        REPT 3                  ;do once each for dest X, Y, and Z
        mov     eax,[si+soff]   ;column 0 entry on this row
        imul    dword ptr [bx]  ;xform entry times source X entry
if ROUNDING_ON
        add     eax,8000h       ;round by adding 2^(-17)
        adc     edx,0           ;whole part of result is in DX
endif ;ROUNDING_ON
        shrd    eax,edx,16      ;shift the result back to 16.16 form
        mov     ecx,eax         ;set running total

        mov     eax,[si+soff+4] ;column 1 entry on this row
        imul    dword ptr [bx+4] ;xform entry times source Y entry
if ROUNDING_ON
        add     eax,8000h       ;round by adding 2^(-17)
        adc     edx,0           ;whole part of result is in DX
endif ;ROUNDING_ON
        shrd    eax,edx,16      ;shift the result back to 16.16 form
        add     ecx,eax         ;running total for this row

        mov     eax,[si+soff+8] ;column 2 entry on this row
        imul    dword ptr [bx+8] ;xform entry times source Z entry
if ROUNDING_ON
        add     eax,8000h       ;round by adding 2^(-17)
        adc     edx,0           ;whole part of result is in DX
endif ;ROUNDING_ON
        shrd    eax,edx,16      ;shift the result back to 16.16 form
        add     ecx,eax         ;running total for this row

        add     ecx,[si+soff+12] ;add in translation
        mov     [di+doff],ecx   ;save the result in the dest vector
soff=soff+16
doff=doff+4
        ENDM

        pop     di              ;restore register variables
        pop     si
        pop     bp              ;restore stack frame
        ret
_XformVec       endp
;======================================================================
; Matrix multiplies SourceXform1 by SourceXform2 and stores the
; result in DestXform. Multiplies a 4x4 matrix times a 4x4 matrix;
```

```
; the result is a 4x4 matrix. Cheats by assuming the bottom row of
; each matrix is 0 0 0 1, and doesn't bother to set the bottom row
; of the destination.
; C near-callable as:
;       void ConcatXforms(Xform SourceXform1, Xform SourceXform2,
;               Xform DestXform)
;
; This assembly code is equivalent to this C code:
;   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];
;   }

CXparms struc
        dw      2 dup(?)        ;return address & pushed BP
SourceXform1 dw ?               ;pointer to first source xform matrix
SourceXform2 dw ?               ;pointer to second source xform matrix
DestXform    dw ?               ;pointer to destination xform matrix
CXparms ends

        align   ALIGNMENT
        public _ConcatXforms
_ConcatXforms   proc    near
        push    bp              ;preserve stack frame
        mov     bp,sp           ;set up local stack frame
        push    si              ;preserve register variables
        push    di

        mov     bx,[bp].SourceXform2 ;BX points to xform2 matrix
        mov     si,[bp].SourceXform1 ;SI points to xform1 matrix
        mov     di,[bp].DestXform    ;DI points to dest xform matrix

roff=0                          ;row offset
        REPT 3                  ;once for each row
coff=0                          ;column offset
        REPT  4                 ;once for each column
        mov     eax,[si+roff]   ;column 0 entry on this row
        imul    dword ptr [bx+coff] ;times row 0 entry in column
if ROUNDING_ON
        add     eax,8000h       ;round by adding 2^(-17)
        adc     edx,0           ;whole part of result is in DX
endif ;ROUNDING_ON
        shrd    eax,edx,16      ;shift the result back to 16.16 form
        mov     ecx,eax         ;set running total

        mov     eax,[si+roff+4] ;column 1 entry on this row
        imul    dword ptr [bx+coff+16] ;times row 1 entry in col
if ROUNDING_ON
        add     eax,8000h       ;round by adding 2^(-17)
        adc     edx,0           ;whole part of result is in DX
endif ;ROUNDING_ON
        shrd    eax,edx,16      ;shift the result back to 16.16 form
        add     ecx,eax         ;running total

        mov     eax,[si+roff+8] ;column 2 entry on this row
        imul    dword ptr [bx+coff+32] ;times row 2 entry in col
if ROUNDING_ON
        add     eax,8000h       ;round by adding 2^(-17)
        adc     edx,0           ;whole part of result is in DX
endif ;ROUNDING_ON
        shrd    eax,edx,16      ;shift the result back to 16.16 form
        add     ecx,eax         ;running total

        add     ecx,[si+roff+12] ;add in translation

        mov     [di+coff+roff],ecx ;save the result in dest matrix
coff=coff+4                     ;point to next col in xform2 & dest
        ENDM
```

```
roff=roff+16                    ;point to next col in xform2 & dest
        ENDM

        pop     di              ;restore register variables
        pop     si
        pop     bp              ;restore stack frame
        ret
_ConcatXforms   endp
        end
```

[LISTING TWO]

```
/* Object list-related functions. */
#include <stdio.h>
#include "polygon.h"

/* Set up the empty object list, with sentinels at both ends to
  terminate searches */
void InitializeObjectList()
{
   ObjectListStart.NextObject = &ObjectListEnd;
   ObjectListStart.PreviousObject = NULL;
   ObjectListStart.CenterInView.Z = INT_TO_FIXED(-32768);
   ObjectListEnd.NextObject = NULL;
   ObjectListEnd.PreviousObject = &ObjectListStart;
   ObjectListEnd.CenterInView.Z = 0x7FFFFFFFL;
   NumObjects = 0;
}

/* Adds an object to the object list, sorted by center Z coord. */
void AddObject(Object *ObjectPtr)
{
   Object *ObjectListPtr = ObjectListStart.NextObject;

   /* Find the insertion point. Guaranteed to terminate because of
      the end sentinel */
   while (ObjectPtr->CenterInView.Z > ObjectListPtr->CenterInView.Z) {
      ObjectListPtr = ObjectListPtr->NextObject;
   }

   /* Link in the new object */
   ObjectListPtr->PreviousObject->NextObject = ObjectPtr;
   ObjectPtr->NextObject = ObjectListPtr;
   ObjectPtr->PreviousObject = ObjectListPtr->PreviousObject;
   ObjectListPtr->PreviousObject = ObjectPtr;
   NumObjects++;
}

/* Resorts the objects in order of ascending center Z coordinate in view space,
   by moving each object in turn to the correct position in the object list. */
void SortObjects()
{
   int i;
   Object *ObjectPtr, *ObjectCmpPtr, *NextObjectPtr;

   /* Start checking with the second object */
   ObjectCmpPtr = ObjectListStart.NextObject;
   ObjectPtr = ObjectCmpPtr->NextObject;
   for (i=1; i<NumObjects; i++) {
      /* See if we need to move backward through the list */
      if (ObjectPtr->CenterInView.Z < ObjectCmpPtr->CenterInView.Z) {
         /* Remember where to resume sorting with the next object */
         NextObjectPtr = ObjectPtr->NextObject;
         /* Yes, move backward until we find the proper insertion
            point. Termination guaranteed because of start sentinel */
         do {
            ObjectCmpPtr = ObjectCmpPtr->PreviousObject;
         } while (ObjectPtr->CenterInView.Z <
               ObjectCmpPtr->CenterInView.Z);
```

```
            /* Now move the object to its new location */
            /* Unlink the object at the old location */
            ObjectPtr->PreviousObject->NextObject =
                  ObjectPtr->NextObject;
            ObjectPtr->NextObject->PreviousObject =
                  ObjectPtr->PreviousObject;

            /* Link in the object at the new location */
            ObjectCmpPtr->NextObject->PreviousObject = ObjectPtr;
            ObjectPtr->PreviousObject = ObjectCmpPtr;
            ObjectPtr->NextObject = ObjectCmpPtr->NextObject;
            ObjectCmpPtr->NextObject = ObjectPtr;

            /* Advance to the next object to sort */
            ObjectCmpPtr = NextObjectPtr->PreviousObject;
            ObjectPtr = NextObjectPtr;
         } else {
            /* Advance to the next object to sort */
            ObjectCmpPtr = ObjectPtr;
            ObjectPtr = ObjectPtr->NextObject;
         }
      }
   }
```

Journal: Dr. Dobb's Journal June 1992 v17 n6 p139(7)

Fast antialiasing

Author: Abrash, Michael.
AttFile: GP-JUN92.ASC Source code listing.

Abstract: Advanced computer graphics techniques require powerful computers and dedicated systems to handle the output, but utility microcomputers based on Intel's 80386 and 80486 microprocessors become burdened when trying to compute such graphics features as anti aliasing. Anti aliasing involves smoothing lines and edges to remove the jagged appearance. Anti aliasing makes computer images more attractive and aids in making technical drawings more accurate. An algorithm developed by Xiaolin Wu describing line anti aliasing is described. The algorithm brackets either side of a line's pixel with pixels of differing values of color. The sum intensity of the bordering pixels equals the intensity of the line's pixel.

The thought first popped into my head as I unenthusiastically picked through the salad bar at a local "family" restaurant, trying to decide whether the meatballs, the fried clams, or the lasagna was likely to shorten my life the least. I decided on the chicken in mystery sauce.

The thought recurred when my daughter asked, "Dad, is that fried chicken?"

"I don't think so," I said. "I think it's stewed chicken."

"It looks like fried chicken."

"Maybe it's fried, stewed chicken," my wife volunteered hopefully. I took a bite. It was, indeed, fried, stewed chicken. I can now, unhesitatingly and without reservation, recommend that you avoid fried, stewed chicken at all costs.

The thought I had was as follows: This is not good food. Not a profound thought, but it raises an interesting question: Why was I eating in this restaurant? The answer, to borrow a phrase from E.F. Schumacher, is appropriate technology. For a family on a budget, with a small child, tired of starting at each other over the kitchen table, this was a perfect place to eat. It was cheap, it had greasy food and ice cream, no one cared if children dropped things or talked loudly or walked around, and most important of all, it wasn't home. So what if the food was lousy? Good food was a luxury, a bonus; everything on the above list was necessary. A family restaurant was the appropriate dining-out technology, given the parameters within which we had to work.

When I read through SIGGRAPH proceedings and other state-of-the-art computer-graphics material, all too often I feel like I'm dining at a four-star restaurant with two-year-old triplets and an empty wallet. We're talking incredibly inappropriate technology for PC graphics here. Sure, I say to myself as I read about an antialiasing technique, that sounds wonderful--if I had 24-bpp color, and dedicated hardware to do the processing, and all day to wait to generate one image. Yes, I think, that is a good way to do hidden surface removal--in a system with hardware z-buffering. Most of the stuff in Computer Graphics is riveting, but, alas, pretty much useless on PCs. When an 80x86 has to do all the work, speed becomes the overriding parameter, especially for real-time graphics.

Literature that's applicable to fast PC graphics is hard enough to find, but what we'd really like is above-average image quality combined with terrific speed, and there's almost no literature of that sort around. There is some, though, and you folks are right on top of it. For example, alert reader Michael Chaplin, of San Diego, wrote to suggest that I might enjoy the line-antialiasing algorithm presented in Xiaolin Wu's article, "An Efficient Antialiasing Technique," in the July 1991 issue of Computer Graphics. Michael was dead-on right. This is a great algorithm, combining excellent antialiased line quality with speed that's close to that of non-antialiased Bresenham's line drawing. This is the sort of algorithm that makes you want to go out and write a wireframe animation program, just so you can see how

good those smooth lines look in motion. Wu antialiasing is a wonderful example of what can be accomplished on inexpensive, mass-market hardware with the proper programming perspective. In sort, it's a splendid example of appropriate technology for PCs.

Wu Antialiasing

To recap briefly, antialiasing is the process of smoothing lines and edges so that they appear less jagged. Antialiasing is partly an aesthetic issue, because it makes images more attractive. It's partly an accuracy issue, because it makes it possible to position and draw images with effectively more precision than the resolution of the display. Finally, it's partly a flat-out necessity, to avoid the horrible, crawling, jagged edges of temporal aliasing when performing animation.

The basic premise of Wu antialiasing is almost ridiculously simple: As the algorithm steps one pixed unit at a time along the major (longer) axis of a line, it draws the two pixels bracketing the line along the minor acis at each point. Each of the two bracketing pixels is drawn with a weighted fraction of the full intensity of the drawing color, with the weighting for each pixel equal to one minus the pixel's distance along the minor axis from the ideal line. Figure 1 illustrates this concept.

The intensities of the two pixels that bracket the line are selected so that they always sum to exactly 1; that is, to the intensity of one fully illuminated pixel of the drawing color. The presence of aggregate full-pixel intensity means that at each step, the line has the same brightness it would have if a single pixel were drawn at precisely the correct location. Moreover, thanks to the distribution of the intensity weighting, that brightness is centered at the ideal line. Not coincidentally, a line drawn with pixel pairs of aggregate single-pixel intensity, centered on the ideal line, is perceived by the eye not as a jagged collection of pixel pairs, but as a smooth line centered on the ideal line. Thus, by weighting the bracketing pixels properly at each step, we can readily produce what looks like a smooth line at precisely the right location, rather than the jagged pattern of line segments that non-antialiased line-drawing algorithms such as Bresenham's trace out.

You might expect that the implementation of Wu antialiasing would fall into two distinct areas: tracing out the line (that is, finding the appropriate pixel pairs to draw) and calculating the appropriate weightings for each pixel pair. Not so, however. The weighting calculations involved only a few shifts, XORs, and adds; for all practical purposes, tracing and weighting are rolled into one step--and a very fast step it is. How fast is it? On a 33-MHz 486 with a fast VGA, a good but not maxed-out assembler implementation of Wu antialiasing draws a more than respectable 5000 150-pixel-long vectors per second. That's especially impressive considering that about 1,500,000 actual pixels are drawn per second, meaning that Wu antialiasing is drawing at over 50 percent of the maximum memory bandwidth--and hence more than half the faster theoretically possible drawing speed--of an AT-bus VGA. In short, Wu antialiasing is about as fast an antialiased line approach as you could ever hope to find for the VGA.

Tracing and Intensity in One

Horizontal, vertical, and diagonal lines do not require Wu antialiasing because they pass through the center of every pixel they meet; such lines can be drawn with fast, special-case code. For all other cases, Wu lines are traced out one step at a time along the major axis by means of a simple, fixed-point algorithm. The move along the minor axis with respect to a one-pixel move along the major axis (the line slope for lines with slopes less than 1, 1/slope for lines with slopes greater than 1) is calculated with a single integer divide. This value, called the "error adjust," is stored as a fixed-point fraction, in 0.16 format (that is, all bits are fractional, and the decimal point is just to the left of bit 15). An error accumulator, also in 0.16 format, is initialized to 0. Then the first pixel is drawn; no weighting is needed, because the line intersects its endpoints exactly.

Now the error adjust is added to the error accumulator. The error accumulator indicates how far between pixels the line has progressed along

the minor axis at any given step; when the error accumulator turns over, it's time to advance one pixel along the minor axis. At each step along the line, the major-axis coordinate advances by one pixel. The two bracketing pixels to draw are simply the two pixels nearest the line along the minor axis. For instance, if X is the current major-axis coordinate and Y is the current minor-axis coordinate, then the two pixels to be drawn are (X,Y) and (X,Y + 1). In short, the derivation of the pixels at which to draw involves nothing more complicated than advancing one pixel along the major axis, adding the error adjust to the error accumulator, and advancing one pixel along the minor axis when the error accumulator turns over.

So far, nothing special; but now we come to the true wonder of Wu antialiasing. We know which pair of pixels to draw at each step along the line, but we also need to generate the two proper intensities, which must be inversely proportional to distance from the ideal line and sum to 1, and that's a potentially time-consuming operation. Let's assume, however, that the number of possible intensity levels to be used for weighting is the value NumLevels = [2.sup.n] for some integer n, with the minimum weighting (0 percent intensity) being the value [2.sup.n-1], and the maximum weighting (100 percent intensity) being the value 0. Given that, lo and behold, the most significant n bits of the error accumulator select the proper intensity value for one element of the pixel pair, as shown in Figure 2. Better yet, [2.sup.n-1] minus the intensity of the first pixel selects the intensity of the other pixel in the pair, because the intensities of the two pixels must sum to 1; as it happens, this result can be obtained simply by flipping the n least-significant bits of the first pixel's value. All this works because what the error accumulator accumulates is precisely the ideal line's current distance between the two bracketing pixels.

The intensity calculations take longer to describe than they do to perform. All that's involved is a shift of the error accumulator to right-justify the desired intensity weighting bits, and then an XOR to flip the least-significant n bits of the first pixel's value in order to generate the second pixel's value. Listing One (page 154) illustrates just how efficient Wu antialiasing is; the intensity calculations take only three statements; and the entire Wu line-drawing loop is only nine statements long. Of course, a single C statement can hide a great deal of complexity, but Listing Six (page 156), the inner loop of an assembler implementation, shows that only 15 instructions are required per step along the major axis--and the number of instructions could be reduced to ten by special-casing and loop unrolling. Make no mistake about it, Wu antialiasing is fast.

Sample Wu Antialiasing

The true test of any antialiasing technique is how good it looks, so lt's have a look at Wu antialiasing in action. Listing One is a C implementation of Wu antialiasing. Listing Two (page 155) is a sample program that draws a variety of Wu-antialiasing lines, followed by non-antialiased lines, for comparison. Listing Three (page 156) contains DrawPixel() and SetMode() functions for mode 13h, the VGA's 320x200 256-color mode. Finally, Listing Four (page 156) is a simple, non-antialiased line-drawing routine. Link these four listings together and run the resulting program to see both Wu-antialiased and non-antialiased lines.

Listing One isn't particularly fast, because it calls DrawPixel() for each pixel. On the other hand, DrawPixel() makes it easy to try out Wu antialiasing in a variety of modes; just adapt the code in Listing Three for the 256-color mode you want to support. For example, Listing Five (page 156) shows code to draw Wu-antialiased lines in 640x480 256-color mode on a Super-VGA built around the Tseng Labs ET4000 chip with at least 512K of display memory installed. It's well worth checking out Wu antialiasing at 640x480. Although antialiased lines look much smoother than normal lines at 320 x 200 resolution, they're far from perfect, because the pixels are so big that the eye can't blend them properly. At 640x480, however, Wu-antialiased lines look fabulous; from a couple of feet away, they look as straight and smooth as if they were drawn with a ruler.

Listing One requires that the DAC palette be set up so that a NumLevel-long block of palette entries contains linearly decreasing intensities of the drawing color. The size of the block is programmable, but must be a power of two. The more intensity levels, the better. Wu says that 32 intensities is enough; on my system, eight and even four levels looked pretty good. I found

that gamma correction, which gives linearly spaced intensity steps, improved antialiasing quality significantly. Fortunately, we can program the palette with gamma-corrected values, so our drawing code doesnt's have to do eny extra work.

Listing One isn't very fast, so I implemented Wu antialiasing in assembler, hard-coded for mode 13h. The inner loop of the assmbler code is shown in Listing Six; the full assembler routine will be available as part of the code archive from this issue of DDJ. (See "Availability" on page 3.) High-speed graphics code and fast VGAs go together like peanut butter and jelly, which is to say very well indeed; the assembler implementation ran more than twice as fast on my 486 after I ran the SETBUS utility from last month to put the VGA into 16-bit mode. Enough said!

Notes on Wu Antialiasing

Wu antialiasing can be applied to any curve for which it's possible to calculate at each step the positions and intensities of two bracketing pixels, although the implementation will generally be nowhere near as efficient as it is for lines. However, Wu's anticle in Computer Graphics does describe an efficient algorithm for drawing antialiased circles. Wu also describes a technique for antialiasing solids, such as filled circles anf polygons. Wu's approach biases the edges of filled objects outward. Although this is no good for adjacent polygons of the sort used in rendering, it's certainly possible to design a more accurate polygon-antialiasing approach around Wu's basic weighting technique. The results would not be quite so good as more sophisticated antialiasing techniques, but they woudl be much faster.

In general, in fact, the results obtained by Wu antialiasing are only so-so, by theoretical measures. Wu antialiasing amounts to a simple box filter placed over a step approximation of a line, and that process introduces a good deal of deviation from the ideal. On the other hand, Wu notes that even a 10 percent error in intensity doesn't lead to noticeable loss of image quality, and for Wu-antialiased lines up to 1K pixels in length, the error is under 10 percent. If it looks good, it is good--and it looks good. With a 16-bit error accumulator, fixed-point inaccuracy becomes a problem for Wu-antialiased lines longer than 1K. For such lines, you should switch to using 32-bit error values, which would let you handle lines of any practical length.

In the listings, I have chosen to truncate, rather than round, the error-adjust value. This increases the intensity error of the line but guarantees that fixed-point inaccuracy won't cause the minor axis to advance past the endpoint. Over-running the endpoint would result in the drawing of pixels outside the line's bounding box, and potentially even in an attempt to access pixels off the edge of the bitmap.

Finally, I should mention tha, as published, Wu's algorithm draws lines symmetrically, from both ends at once. I haven't done this for a number of reasons, not least of which is that symmetric drawing is an inefficient way to draw lines that span banks on banked Super-VGAs. Banking aside, however, symmetric drawing is potentially faster, because it eliminates half of all calculations; in so doing, it cuts intensity error in half, as well.

Matrix Orthogonalization Made Easy

Peter Brooks of MicroMind passed along the following nifty idea. Fixed-point rotation matrices tend to drift from the proper values over the course of repeated concatenations, due to fixed-point error, with the colums gradually ceasing to be mutually perpendicular (orthogonal). It is essential that rotation matrices remain orthogonal, however, in order to perform nondistorting rotations. One way to deal with this is to periodically recalculate one of the column as the cross-product of the other two, which works because the cross-product of two vectors is an orthogonal vector.

Peter's insight was: Why not recalculate one colum as the cross-product of the other two every time you recalculate a rotation matrix? In other words, do the matrix multiplication to generate two columns of the result matrix, then just calculate the third column as the cross-product of the other two.

The row or column calculated as the cross-product should be rotated regularly. Not only does this guarantee an orthogonal matrix at all times, but it also turns out to be faster, because it takes fewer operations to calculate a cross-product than to recalculate a matrix column. (All of the above applies equally well if you substitute "row" for "column.")

This sounds like a great idea to me. My only question is whether one should alternate between rows and columns, in order to distribute error more evenly. Any thoughts, readers?

Next Time

We didn't quite make it back to X-Sharp this month, although the topics were certainly related. Next month, for sure. In the meantime, keep those excellent suggestions coming. And stay the heck away from fried, stewed chicken.

- -

GRAPHICS PROGRAMMING COLUMN
by Michael Abrash

[LISTING ONE]

```
/* Function to draw an antialiased line from (X0,Y0) to (X1,Y1), using an
 * antialiasing approach published by Xiaolin Wu in the July 1991 issue of
 * Computer Graphics. Requires that the palette be set up so that there
 * are NumLevels intensity levels of the desired drawing color, starting at
 * color BaseColor (100% intensity) and followed by (NumLevels-1) levels of
 * evenly decreasing intensity, with color (BaseColor+NumLevels-1) being 0%
 * intensity of the desired drawing color (black). This code is suitable for
 * use at screen resolutions, with lines typically no more than 1K long; for
 * longer lines, 32-bit error arithmetic must be used to avoid problems with
 * fixed-point inaccuracy. No clipping is performed in DrawWuLine; it must be
 * performed either at a higher level or in the DrawPixel function.
 * Tested with Borland C++ 3.0 in C compilation mode and the small model.
 */
extern void DrawPixel(int, int, int);

/* Wu antialiased line drawer.
 * (X0,Y0),(X1,Y1) = line to draw
 * BaseColor = color # of first color in block used for antialiasing, the
 *          100% intensity version of the drawing color
 * NumLevels = size of color block, with BaseColor+NumLevels-1 being the
 *          0% intensity version of the drawing color
 * IntensityBits = log base 2 of NumLevels; the # of bits used to describe
 *          the intensity of the drawing color. 2**IntensityBits==NumLevels
 */
void DrawWuLine(int X0, int Y0, int X1, int Y1, int BaseColor, int NumLevels,
   unsigned int IntensityBits)
{
   unsigned int IntensityShift, ErrorAdj, ErrorAcc;
   unsigned int ErrorAccTemp, Weighting, WeightingComplementMask;
   int DeltaX, DeltaY, Temp, XDir;

   /* Make sure the line runs top to bottom */
   if (Y0 > Y1) {
      Temp = Y0; Y0 = Y1; Y1 = Temp;
      Temp = X0; X0 = X1; X1 = Temp;
   }
   /* Draw the initial pixel, which is always exactly intersected by
      the line and so needs no weighting */
   DrawPixel(X0, Y0, BaseColor);

   if ((DeltaX = X1 - X0) >= 0) {
      XDir = 1;
   } else {
      XDir = -1;
      DeltaX = -DeltaX; /* make DeltaX positive */
   }
   /* Special-case horizontal, vertical, and diagonal lines, which
```

```
   require no weighting because they go right through the center of
   every pixel */
if ((DeltaY = Y1 - Y0) == 0) {
   /* Horizontal line */
   while (DeltaX-- != 0) {
      X0 += XDir;
      DrawPixel(X0, Y0, BaseColor);
   }
   return;
}
if (DeltaX == 0) {
   /* Vertical line */
   do {
      Y0++;
      DrawPixel(X0, Y0, BaseColor);
   } while (--DeltaY != 0);
   return;
}
if (DeltaX == DeltaY) {
   /* Diagonal line */
   do {
      X0 += XDir;
      Y0++;
      DrawPixel(X0, Y0, BaseColor);
   } while (--DeltaY != 0);
   return;
}
/* Line is not horizontal, diagonal, or vertical */
ErrorAcc = 0;  /* initialize the line error accumulator to 0 */
/* # of bits by which to shift ErrorAcc to get intensity level */
IntensityShift = 16 - IntensityBits;
/* Mask used to flip all bits in an intensity weighting, producing the
   result (1 - intensity weighting) */
WeightingComplementMask = NumLevels - 1;
/* Is this an X-major or Y-major line? */
if (DeltaY > DeltaX) {
   /* Y-major line; calculate 16-bit fixed-point fractional part of a
      pixel that X advances each time Y advances 1 pixel, truncating the
      result so that we won't overrun the endpoint along the X axis */
   ErrorAdj = ((unsigned long) DeltaX << 16) / (unsigned long) DeltaY;
   /* Draw all pixels other than the first and last */
   while (--DeltaY) {
      ErrorAccTemp = ErrorAcc;   /* remember currrent accumulated error */
      ErrorAcc += ErrorAdj;      /* calculate error for next pixel */
      if (ErrorAcc <= ErrorAccTemp) {
         /* The error accumulator turned over, so advance the X coord */
         X0 += XDir;
      }
      Y0++; /* Y-major, so always advance Y */
      /* The IntensityBits most significant bits of ErrorAcc give us the
         intensity weighting for this pixel, and the complement of the
         weighting for the paired pixel */
      Weighting = ErrorAcc >> IntensityShift;
      DrawPixel(X0, Y0, BaseColor + Weighting);
      DrawPixel(X0 + XDir, Y0,
            BaseColor + (Weighting ^ WeightingComplementMask));
   }
   /* Draw the final pixel, which is always exactly intersected by the line
      and so needs no weighting */
   DrawPixel(X1, Y1, BaseColor);
   return;
}
/* It's an X-major line; calculate 16-bit fixed-point fractional part of a
   pixel that Y advances each time X advances 1 pixel, truncating the
   result to avoid overrunning the endpoint along the X axis */
ErrorAdj = ((unsigned long) DeltaY << 16) / (unsigned long) DeltaX;
/* Draw all pixels other than the first and last */
while (--DeltaX) {
   ErrorAccTemp = ErrorAcc;   /* remember currrent accumulated error */
   ErrorAcc += ErrorAdj;      /* calculate error for next pixel */
   if (ErrorAcc <= ErrorAccTemp) {
      /* The error accumulator turned over, so advance the Y coord */
      Y0++;
   }
```

```
      X0 += XDir; /* X-major, so always advance X */
      /* The IntensityBits most significant bits of ErrorAcc give us the
         intensity weighting for this pixel, and the complement of the
         weighting for the paired pixel */
      Weighting = ErrorAcc >> IntensityShift;
      DrawPixel(X0, Y0, BaseColor + Weighting);
      DrawPixel(X0, Y0 + 1,
            BaseColor + (Weighting ^ WeightingComplementMask));
   }
   /* Draw the final pixel, which is always exactly intersected by the line
      and so needs no weighting */
   DrawPixel(X1, Y1, BaseColor);
}
```

[LISTING TWO]

```
/* Sample line-drawing program to demonstrate Wu antialiasing. Also draws
 * non-antialiased lines for comparison.
 * Tested with Borland C++ 3.0 in C compilation mode and the small model.
 */
#include <dos.h>
#include <conio.h>

void SetPalette(struct WuColor *);
extern void DrawWuLine(int, int, int, int, int, int, unsigned int);
extern void DrawLine(int, int, int, int, int);
extern void SetMode(void);
extern int ScreenWidthInPixels;  /* screen dimension globals */
extern int ScreenHeightInPixels;

#define NUM_WU_COLORS 2 /* # of colors we'll do antialiased drawing with */
struct WuColor {        /* describes one color used for antialiasing */
   int BaseColor;       /* # of start of palette intensity block in DAC */
   int NumLevels;       /* # of intensity levels */
   int IntensityBits;   /* IntensityBits == log2 NumLevels */
   int MaxRed;          /* red component of color at full intensity */
   int MaxGreen;        /* green component of color at full intensity */
   int MaxBlue;         /* blue component of color at full intensity */
};
enum {WU_BLUE=0, WU_WHITE=1};             /* drawing colors */
struct WuColor WuColors[NUM_WU_COLORS] =  /* blue and white */
    {{192, 32, 5, 0, 0, 0x3F}, {224, 32, 5, 0x3F, 0x3F, 0x3F}};

void main()
{
   int CurrentColor, i;
   union REGS regset;

   /* Draw Wu-antialiased lines in all directions */
   SetMode();
   SetPalette(WuColors);
   for (i=5; i<ScreenWidthInPixels; i += 10) {
      DrawWuLine(ScreenWidthInPixels/2-ScreenWidthInPixels/10+i/5,
            ScreenHeightInPixels/5, i, ScreenHeightInPixels-1,
            WuColors[WU_BLUE].BaseColor, WuColors[WU_BLUE].NumLevels,
            WuColors[WU_BLUE].IntensityBits);
   }
   for (i=0; i<ScreenHeightInPixels; i += 10) {
      DrawWuLine(ScreenWidthInPixels/2-ScreenWidthInPixels/10, i/5, 0, i,
            WuColors[WU_BLUE].BaseColor, WuColors[WU_BLUE].NumLevels,
            WuColors[WU_BLUE].IntensityBits);
   }
   for (i=0; i<ScreenHeightInPixels; i += 10) {
      DrawWuLine(ScreenWidthInPixels/2+ScreenWidthInPixels/10, i/5,
            ScreenWidthInPixels-1, i, WuColors[WU_BLUE].BaseColor,
            WuColors[WU_BLUE].NumLevels, WuColors[WU_BLUE].IntensityBits);
   }
   for (i=0; i<ScreenWidthInPixels; i += 10) {
      DrawWuLine(ScreenWidthInPixels/2-ScreenWidthInPixels/10+i/5,
            ScreenHeightInPixels, i, 0, WuColors[WU_WHITE].BaseColor,
            WuColors[WU_WHITE].NumLevels,
            WuColors[WU_WHITE].IntensityBits);
   }
```

```
   getch();                /* wait for a key press */

   /* Now clear the screen and draw non-antialiased lines */
   SetMode();
   SetPalette(WuColors);
   for (i=0; i<ScreenWidthInPixels; i += 10) {
      DrawLine(ScreenWidthInPixels/2-ScreenWidthInPixels/10+i/5,
            ScreenHeightInPixels/5, i, ScreenHeightInPixels-1,
            WuColors[WU_BLUE].BaseColor);
   }
   for (i=0; i<ScreenHeightInPixels; i += 10) {
      DrawLine(ScreenWidthInPixels/2-ScreenWidthInPixels/10, i/5, 0, i,
            WuColors[WU_BLUE].BaseColor);
   }
   for (i=0; i<ScreenHeightInPixels; i += 10) {
      DrawLine(ScreenWidthInPixels/2+ScreenWidthInPixels/10, i/5,
            ScreenWidthInPixels-1, i, WuColors[WU_BLUE].BaseColor);
   }
   for (i=0; i<ScreenWidthInPixels; i += 10) {
      DrawLine(ScreenWidthInPixels/2-ScreenWidthInPixels/10+i/5,
            ScreenHeightInPixels, i, 0, WuColors[WU_WHITE].BaseColor);
   }
   getch();                /* wait for a key press */

   regset.x.ax = 0x0003;   /* AL = 3 selects 80x25 text mode */
   int86(0x10, &regset, &regset);   /* return to text mode */
}

/* Sets up the palette for antialiasing with the specified colors.
 * Intensity steps for each color are scaled from the full desired intensity
 * of the red, green, and blue components for that color down to 0%
 * intensity; each step is rounded to the nearest integer. Colors are
 * corrected for a gamma of 2.3. The values that the palette is programmed
 * with are hardwired for the VGA's 6 bit per color DAC.
 */
void SetPalette(struct WuColor * WColors)
{
   int i, j;
   union REGS regset;
   struct SREGS sregset;
   static unsigned char PaletteBlock[256][3];   /* 256 RGB entries */
   /* Gamma-corrected DAC color components for 64 linear levels from 0% to
      100% intensity */
   static unsigned char GammaTable[] = {
      0, 10, 14, 17, 19, 21, 23, 24, 26, 27, 28, 29, 31, 32, 33, 34,
      35, 36, 37, 37, 38, 39, 40, 41, 41, 42, 43, 44, 44, 45, 46, 46,
      47, 48, 48, 49, 49, 50, 51, 51, 52, 52, 53, 53, 54, 54, 55, 55,
      56, 56, 57, 57, 58, 58, 59, 59, 60, 60, 61, 61, 62, 62, 63, 63};

   for (i=0; i<NUM_WU_COLORS; i++) {
      for (j=0; j<WColors[i].NumLevels; j++) {
         PaletteBlock[j][0] = GammaTable[((double)WColors[i].MaxRed * (1.0 -
               (double)j / (double)(WColors[i].NumLevels - 1))) + 0.5];
         PaletteBlock[j][1] = GammaTable[((double)WColors[i].MaxGreen * (1.0 -
               (double)j / (double)(WColors[i].NumLevels - 1))) + 0.5];
         PaletteBlock[j][2] = GammaTable[((double)WColors[i].MaxBlue * (1.0 -
               (double)j / (double)(WColors[i].NumLevels - 1))) + 0.5];
      }
      /* Now set up the palette to do Wu antialiasing for this color */
      regset.x.ax = 0x1012;   /* set block of DAC registers function */
      regset.x.bx = WColors[i].BaseColor;   /* first DAC location to load */
      regset.x.cx = WColors[i].NumLevels;   /* # of DAC locations to load */
      regset.x.dx = (unsigned int)PaletteBlock; /* offset of array from which
                                                   to load RGB settings */
      sregset.es = _DS; /* segment of array from which to load settings */
      int86x(0x10, &regset, &regset, &sregset); /* load the palette block */
   }
}
```

[LISTING THREE]

```
/* VGA mode 13h pixel-drawing and mode set functions.
 * Tested with Borland C++ 3.0 in C compilation mode and the small model.
 */
#include <dos.h>

/* Screen dimension globals, used in main program to scale. */
int ScreenWidthInPixels = 320;
int ScreenHeightInPixels = 200;

/* Mode 13h draw pixel function. */
void DrawPixel(int X, int Y, int Color)
{
#define SCREEN_SEGMENT  0xA000
   unsigned char far *ScreenPtr;

   FP_SEG(ScreenPtr) = SCREEN_SEGMENT;
   FP_OFF(ScreenPtr) = (unsigned int) Y * ScreenWidthInPixels + X;
   *ScreenPtr = Color;
}

/* Mode 13h mode-set function. */
void SetMode()
{
   union REGS regset;

   /* Set to 320x200 256-color graphics mode */
   regset.x.ax = 0x0013;
   int86(0x10, &regset, &regset);
}
```

[LISTING FOUR]

```
/* Function to draw a non-antialiased line from (X0,Y0) to (X1,Y1), using a
 * simple fixed-point error accumulation approach.
 * Tested with Borland C++ 3.0 in C compilation mode and the small model.
 */
extern void DrawPixel(int, int, int);

/* Non-antialiased line drawer.
 * (X0,Y0),(X1,Y1) = line to draw, Color = color in which to draw
 */
void DrawLine(int X0, int Y0, int X1, int Y1, int Color)
{
   unsigned long ErrorAcc, ErrorAdj;
   int DeltaX, DeltaY, XDir, Temp;

   /* Make sure the line runs top to bottom */
   if (Y0 > Y1) {
      Temp = Y0; Y0 = Y1; Y1 = Temp;
      Temp = X0; X0 = X1; X1 = Temp;
   }
   DrawPixel(X0, Y0, Color);  /* draw the initial pixel */
   if ((DeltaX = X1 - X0) >= 0) {
      XDir = 1;
   } else {
      XDir = -1;
      DeltaX = -DeltaX; /* make DeltaX positive */
   }
   if ((DeltaY = Y1 - Y0) == 0)  /* done if only one point in the line */
      if (DeltaX == 0) return;

   ErrorAcc = 0x8000;   /* initialize line error accumulator to .5, so we can
                           advance when we get halfway to the next pixel */
   /* Is this an X-major or Y-major line? */
   if (DeltaY > DeltaX) {
      /* Y-major line; calculate 16-bit fixed-point fractional part of a
         pixel that X advances each time Y advances 1 pixel */
      ErrorAdj = ((((unsigned long)DeltaX << 17) / (unsigned long)DeltaY) +
            1) >> 1;
```

```
      /* Draw all pixels between the first and last */
      do {
         ErrorAcc += ErrorAdj;      /* calculate error for this pixel */
         if (ErrorAcc & ~0xFFFFL) {
            /* The error accumulator turned over, so advance the X coord */
            X0 += XDir;
            ErrorAcc &= 0xFFFFL;    /* clear integer part of result */
         }
         Y0++;                      /* Y-major, so always advance Y */
         DrawPixel(X0, Y0, Color);
      } while (--DeltaY);
      return;
   }
   /* It's an X-major line; calculate 16-bit fixed-point fractional part of a
      pixel that Y advances each time X advances 1 pixel */
   ErrorAdj = ((((unsigned long)DeltaY << 17) / (unsigned long)DeltaX) +
         1) >> 1;
   /* Draw all remaining pixels */
   do {
      ErrorAcc += ErrorAdj;      /* calculate error for this pixel */
      if (ErrorAcc & ~0xFFFFL) {
         /* The error accumulator turned over, so advance the Y coord */
         Y0++;
         ErrorAcc &= 0xFFFFL;    /* clear integer part of result */
      }
      X0 += XDir;                /* X-major, so always advance X */
      DrawPixel(X0, Y0, Color);
   } while (--DeltaX);
}
```

[LISTING FIVE]

```
/* Mode set and pixel-drawing functions for the 640x480 256-color mode of
 * Tseng Labs ET4000-based SuperVGAs.
 * Tested with Borland C++ 3.0 in C compilation mode and the small model.
 */
#include <dos.h>

/* Screen dimension globals, used in main program to scale */
int ScreenWidthInPixels = 640;
int ScreenHeightInPixels = 480;

/* ET4000 640x480 256-color draw pixel function. */
void DrawPixel(int X, int Y, int Color)
{
#define SCREEN_SEGMENT        0xA000
#define GC_SEGMENT_SELECT     0x3CD /* ET4000 segment (bank) select reg */
   unsigned char far *ScreenPtr;
   unsigned int Bank;
   unsigned long BitmapAddress;

   /* Full bitmap address of pixel, as measured from address 0 to 0xFFFFF */
   BitmapAddress = (unsigned long) Y * ScreenWidthInPixels + X;
   /* Bank # is upper word of bitmap addr */
   Bank = BitmapAddress >> 16;
   /* Upper nibble is read bank #, lower nibble is write bank # */
   outp(GC_SEGMENT_SELECT, (Bank << 4) | Bank);
   /* Draw into the bank */
   FP_SEG(ScreenPtr) = SCREEN_SEGMENT;
   FP_OFF(ScreenPtr) = (unsigned int) BitmapAddress;
   *ScreenPtr = Color;
}

/* ET4000 640x480 256-color mode-set function. */
void SetMode()
{
   union REGS regset;

   /* Set to 640x480 256-color graphics mode */
   regset.x.ax = 0x002E;
   int86(0x10, &regset, &regset);
}
```

[LISTING SIX]

```
; Inner loop for drawing Y-major lines from Wu-antialiased line drawer.
YMajorLoop:
        add     dx,bp                   ;calculate error for next pixel
        jnc     NoXAdvance              ;not time to step in X yet
                                        ;the error accumulator turned over,
                                        ; so advance the X coord
        add     si,bx                   ;add XDir to the pixel pointer
NoXAdvance:
        add     si,SCREEN_WIDTH_IN_BYTES ;Y-major, so always advance Y
; The IntensityBits most significant bits of ErrorAcc give us the intensity
; weighting for this pixel, and the complement of the weighting for the
; paired pixel.
        mov     ah,dh   ;msb of ErrorAcc
        shr     ah,cl   ;Weighting = ErrorAcc >> IntensityShift;
        add     ah,al   ;BaseColor + Weighting
        mov     [si],ah ;DrawPixel(X, Y, BaseColor + Weighting);
        mov     ah,dh   ;msb of ErrorAcc
        shr     ah,cl   ;Weighting = ErrorAcc >> IntensityShift;
        xor     ah,ch   ;Weighting ^ WeightingComplementMask
        add     ah,al   ;BaseColor + (Weighting ^ WeightingComplementMask)
        mov     [si+bx],ah ;DrawPixel(X+XDir, Y,
                        ; BaseColor + (Weighting ^ WeightingComplementMask));
        dec     di      ;--DeltaY
        jnz     YMajorLoop
```

[WU ANTIALIASING]

```
; C near-callable function to draw an antialiased line from
; (X0,Y0) to (X1,Y1), in mode 13h, the VGA's standard 320x200 256-color
; mode. Uses an antialiasing approach published by Xiaolin Wu in the July
; 1991 issue of Computer Graphics. Requires that the palette be set up so
; that there are NumLevels intensity levels of the desired drawing color,
; starting at color BaseColor (100% intensity) and followed by (NumLevels-1)
; levels of evenly decreasing intensity, with color (BaseColor+NumLevels-1)
; being 0% intensity of the desired drawing color (black). No clipping is
; performed in DrawWuLine. Handles a maximum of 256 intensity levels per
; antialiased color. This code is suitable for use at screen resolutions,
; with lines typically no more than 1K long; for longer lines, 32-bit error
; arithmetic must be used to avoid problems with fixed-point inaccuracy.
; Tested with TASM 3.0.
;
; C near-callable as:
;       void DrawWuLine(int X0, int Y0, int X1, int Y1, int BaseColor,
;               int NumLevels, unsigned int IntensityBits);

SCREEN_WIDTH_IN_BYTES equ 320      ;# of bytes from the start of one scan line
                                   ; to the start of the next
SCREEN_SEGMENT       equ    0a000h ;segment in which screen memory resides

; Parameters passed in stack frame.
parms  struc
       dw     2 dup (?) ;pushed BP and return address
X0     dw     ?       ;X coordinate of line start point
Y0     dw     ?       ;Y coordinate of line start point
X1     dw     ?       ;X coordinate of line end point
Y1     dw     ?       ;Y coordinate of line end point
BaseColor dw  ?       ;color # of first color in block used for
                      ; antialiasing, the 100% intensity version of the
                      ; drawing color
NumLevels dw  ?       ;size of color block, with BaseColor+NumLevels-1
                      ; being the 0% intensity version of the drawing color
                      ; (maximum NumLevels = 256)
IntensityBits dw ?    ;log base 2 of NumLevels; the # of bits used to
                      ; describe the intensity of the drawing color.
                      ; 2**IntensityBits==NumLevels
                      ; (maximum IntensityBits = 8)
parms  ends

       .model small
```

```
        .code
; Screen dimension globals, used in main program to scale.
_ScreenWidthInPixels    dw      320
_ScreenHeightInPixels   dw      200

        .code
        public _DrawWuLine
_DrawWuLine proc near
        push    bp      ;preserve caller's stack frame
        mov     bp,sp   ;point to local stack frame
        push    si      ;preserve C's register variables
        push    di
        push    ds      ;preserve C's default data segment
        cld             ;make string instructions increment their pointers

; Make sure the line runs top to bottom.
        mov     si,[bp].X0
        mov     ax,[bp].Y0
        cmp     ax,[bp].Y1      ;swap endpoints if necessary to ensure that
        jna     NoSwap          ; Y0 <= Y1
        xchg    [bp].Y1,ax
        mov     [bp].Y0,ax
        xchg    [bp].X1,si
        mov     [bp].X0,si
NoSwap:

; Draw the initial pixel, which is always exactly intersected by the line
; and so needs no weighting.
        mov     dx,SCREEN_SEGMENT
        mov     ds,dx           ;point DS to the screen segment
        mov     dx,SCREEN_WIDTH_IN_BYTES
        mul     dx              ;Y0 * SCREEN_WIDTH_IN_BYTES yields the offset
                                ; of the start of the row start the initial
                                ; pixel is on
        add     si,ax           ;point DS:SI to the initial pixel
        mov     al,byte ptr [bp].BaseColor ;color with which to draw
        mov     [si],al                 ;draw the initial pixel

        mov     bx,1            ;XDir = 1; assume DeltaX >= 0
        mov     cx,[bp].X1
        sub     cx,[bp].X0      ;DeltaX; is it >= 1?
        jns     DeltaXSet       ;yes, move left->right, all set
                                ;no, move right->left
        neg     cx              ;make DeltaX positive
        neg     bx              ;XDir = -1
DeltaXSet:

; Special-case horizontal, vertical, and diagonal lines, which require no
; weighting because they go right through the center of every pixel.
        mov     dx,[bp].Y1
        sub     dx,[bp].Y0      ;DeltaY; is it 0?
        jnz     NotHorz                 ;no, not horizontal
                                ;yes, is horizontal, special case
        and     bx,bx           ;draw from left->right?
        jns     DoHorz          ;yes, all set
        std                     ;no, draw right->left
DoHorz:
        lea     di,[bx+si]      ;point DI to next pixel to draw
        mov     ax,ds
        mov     es,ax           ;point ES:DI to next pixel to draw
        mov     al,byte ptr [bp].BaseColor ;color with which to draw
                                ;CX = DeltaX at this point
        rep     stosb           ;draw the rest of the horizontal line
        cld                     ;restore default direction flag
        jmp     Done            ;and we're done

        align   2
NotHorz:
        and     cx,cx           ;is DeltaX 0?
        jnz     NotVert                 ;no, not a vertical line
                                ;yes, is vertical, special case
        mov     al,byte ptr [bp].BaseColor ;color with which to draw
VertLoop:
        add     si,SCREEN_WIDTH_IN_BYTES ;point to next pixel to draw
```

```
        mov     [si],al             ;draw the next pixel
        dec     dx              ;--DeltaY
        jnz     VertLoop
        jmp     Done            ;and we're done

        align   2
NotVert:
        cmp     cx,dx           ;DeltaX == DeltaY?
        jnz     NotDiag                 ;no, not diagonal
                                ;yes, is diagonal, special case
        mov     al,byte ptr [bp].BaseColor ;color with which to draw
DiagLoop:
        lea     si,[si+SCREEN_WIDTH_IN_BYTES+bx]
                                ;advance to next pixel to draw by
                                ; incrementing Y and adding XDir to X
        mov     [si],al                 ;draw the next pixel
        dec     dx              ;--DeltaY
        jnz     DiagLoop
        jmp     Done            ;and we're done

; Line is not horizontal, diagonal, or vertical.
        align   2
NotDiag:
; Is this an X-major or Y-major line?
        cmp     dx,cx
        jb      XMajor                  ;it's X-major

; It's a Y-major line. Calculate the 16-bit fixed-point fractional part of a
; pixel that X advances each time Y advances 1 pixel, truncating the result
; to avoid overrunning the endpoint along the X axis.
        xchg    dx,cx           ;DX = DeltaX, CX = DeltaY
        sub     ax,ax           ;make DeltaX 16.16 fixed-point value in DX:AX
        div     cx              ;AX = (DeltaX << 16) / DeltaY. Won't overflow
                                ; because DeltaX < DeltaY
        mov     di,cx           ;DI = DeltaY (loop count)
        sub     si,bx           ;back up the start X by 1, as explained below
        mov     dx,-1           ;initialize the line error accumulator to -1,
                                ; so that it will turn over immediately and
                                ; advance X to the start X. This is necessary
                                ; properly to bias error sums of 0 to mean
                                ; "advance next time" rather than "advance
                                ; this time," so that the final error sum can
                                ; never cause drawing to overrun the final X
                                ; coordinate (works in conjunction with
                                ; truncating ErrorAdj, to make sure X can't
                                ; overrun)
        mov     cx,8                    ;CL = # of bits by which to shift
        sub     cx,[bp].IntensityBits       ; ErrorAcc to get intensity level (8
                                ; instead of 16 because we work only
                                ; with the high byte of ErrorAcc)
        mov     ch,byte ptr [bp].NumLevels ;mask used to flip all bits in an
        dec     ch                      ; intensity weighting, producing
                                        ; result (1 - intensity weighting)
        mov     bp,BaseColor[bp]        ;***stack frame not available***
                                ;***from now on                 ***
        xchg    bp,ax                   ;BP = ErrorAdj, AL = BaseColor,
                                ; AH = scratch register

; Draw all remaining pixels.
YMajorLoop:
        add     dx,bp                   ;calculate error for next pixel
         jnc      NoXAdvance                  ;not time to step in X yet
                                              ;the error accumulator turned over,
                                              ; so advance the X coord
          add     si,bx                       ;add XDir to the pixel pointer
NoXAdvance:
          add   si,SCREEN_WIDTH_IN_BYTES ;Y-major, so always advance Y

; The IntensityBits most significant bits of ErrorAcc give us the intensity
; weighting for this pixel, and the complement of the weighting for the
; paired pixel.
        mov     ah,dh  ;msb of ErrorAcc
        shr     ah,cl  ;Weighting = ErrorAcc >> IntensityShift;
        add     ah,al  ;BaseColor + Weighting
```

```
        mov     [si],ah         ;DrawPixel(X, Y, BaseColor + Weighting);
        mov     ah,dh   ;msb of ErrorAcc
        shr     ah,cl   ;Weighting = ErrorAcc >> IntensityShift;
        xor     ah,ch   ;Weighting ^ WeightingComplementMask
        add     ah,al   ;BaseColor + (Weighting ^ WeightingComplementMask)
        mov     [si+bx],ah ;DrawPixel(X+XDir, Y,
                        ; BaseColor + (Weighting ^ WeightingComplementMask));
        dec     di      ;--DeltaY
        jnz     YMajorLoop
        jmp     Done    ;we're done with this line

; It's an X-major line.
        align   2
XMajor:
; Calculate the 16-bit fixed-point fractional part of a pixel that Y advances
; each time X advances 1 pixel, truncating the result to avoid overrunning
; the endpoint along the X axis.
        sub     ax,ax           ;make DeltaY 16.16 fixed-point value in DX:AX
        div     cx              ;AX = (DeltaY << 16) / Deltax. Won't overflow
                                ; because DeltaY < DeltaX
        mov     di,cx           ;DI = DeltaX (loop count)
        sub     si,SCREEN_WIDTH_IN_BYTES ;back up the start X by 1, as
                                ; explained below
        mov     dx,-1           ;initialize the line error accumulator to -1,
                                ; so that it will turn over immediately and
                                ; advance Y to the start Y. This is necessary
                                ; properly to bias error sums of 0 to mean
                                ; "advance next time" rather than "advance
                                ; this time," so that the final error sum can
                                ; never cause drawing to overrun the final Y
                                ; coordinate (works in conjunction with
                                ; truncating ErrorAdj, to make sure Y can't
                                ; overrun)
        mov     cx,8                            ;CL = # of bits by which to shift
        sub     cx,[bp].IntensityBits           ; ErrorAcc to get intensity level (8
                                                ; instead of 16 because we work only
                                                ; with the high byte of ErrorAcc)
        mov     ch,byte ptr [bp].NumLevels  ;mask used to flip all bits in an
        dec     ch                              ; intensity weighting, producing
                                                ; result (1 - intensity weighting)
        mov     bp,BaseColor[bp]        ;***stack frame not available***
                                        ;***from now on              ***
        xchg    bp,ax                   ;BP = ErrorAdj, AL = BaseColor,
                                        ; AH = scratch register
; Draw all remaining pixels.
XMajorLoop:
        add     dx,bp                   ;calculate error for next pixel
        jnc     NoYAdvance              ;not time to step in Y yet
                                        ;the error accumulator turned over,
                                        ; so advance the Y coord
        add     si,SCREEN_WIDTH_IN_BYTES ;advance Y
NoYAdvance:
         add    si,bx           ;X-major, so add XDir to the pixel pointer

; The IntensityBits most significant bits of ErrorAcc give us the intensity
; weighting for this pixel, and the complement of the weighting for the
; paired pixel.
        mov     ah,dh   ;msb of ErrorAcc
        shr     ah,cl   ;Weighting = ErrorAcc >> IntensityShift;
        add     ah,al   ;BaseColor + Weighting
        mov     [si],ah         ;DrawPixel(X, Y, BaseColor + Weighting);
        mov     ah,dh   ;msb of ErrorAcc
        shr     ah,cl   ;Weighting = ErrorAcc >> IntensityShift;
        xor     ah,ch   ;Weighting ^ WeightingComplementMask
        add     ah,al   ;BaseColor + (Weighting ^ WeightingComplementMask)
        mov     [si+SCREEN_WIDTH_IN_BYTES],ah
                ;DrawPixel(X, Y+SCREEN_WIDTH_IN_BYTES,
                ; BaseColor + (Weighting ^ WeightingComplementMask));
        dec     di      ;--DeltaX
        jnz     XMajorLoop

Done:                   ;we're done with this line
        pop     ds      ;restore C's default data segment
        pop     di      ;restore C's register variables
```

```
        pop     si
        pop     bp      ;restore caller's stack frame
        ret             ;done
_DrawWuLine endp
        end
```

Journal: Dr. Dobb's Journal July 1992 v17 n7 p133(7)

3-D shading

Author: Abrash, Michael.
AttFile: GP-JUL92.ASC Source code.

Abstract: The development of a real-time animation package called X-Sharp is discussed, with special attention paid to three-dimensional shading algorithms. X-Sharp's initial implementation runs only on 80386 and higher processors because it uses 32-bit multiply and divide instructions. A new version that supports the 8088 and 80286 processors is described; it uses 32 x 16 instead of 32 x 32 divides. Three-dimensional shading may be either ambient or diffuse; both use the same basic model, with each surface assigned a reflectivity between 1 and 0. Ambient shading expresses intensity as the minimum of a function. Diffuse shading is more complex because the effective intensity of directed light depends on the angle at which it strikes a surface. Details of shading implementation are discussed, along with graphics debugging.

This month, we return to X-Sharp, the real-time animation package that we started developing in January. When last we saw X-Sharp, it had just acquired basic hidden-surface capability, and performance had been vastly improved through the use of fixed-point arithmetic. This month, we're going to add quite a bit more: support for 8088 and 80286 PCs, a general color model, and shading. That's an awful lot to cover in one column (actually, it'll spill over into the next column), so let's get to it!

Sub-386 Support

To date, X-Sharp has run on only the 386 and 486, because it uses 32-bit multiply and divide instructions that sub-386 processors don't support. I chose 32-bit instructions for two reasons: They're much faster for 16.16 fixed-point arithmetic than any approach that works on the 8088 and 286; and they're much easier to implement than any other approach. In short, I was after maximum performance, and I was perhaps just a little lazy.

I should have known better than to try to sneak this one by you. The most common feedback I've gotten on X-Sharp is that I should make it support the 8088 and 286. Well, I can take a hint as well as the next guy. Listing One (page 150) contains the latest versions of the FixedMul[unkeyable] and FixedDiv[unkeyable] functions, in a form that can be conditionally assembled to use either 386-specific or generic 8088 instructions. The complete new version of FIXED.ASM, containing dual 386/8088 versions of CosSin[unkeyable], Xform Vec[unkeyable], and ConcatX-forms[unkeyable], as well as FixedMul[unkeyable] and FixedDiv[unkeyable], can be found in the full X-Sharp archive, the availability of which is described further on.

Given the new version of FIXED.ASM, with USE386 set to O, X-Sharp will run on any processor. That's not to say that it will run fast on any processor, or at least not as fast as it used to. The switch to 8088 instructions makes X-Sharp's fixed-point calculations about 2.5 times slower overall. Since a PC is perhaps 40 times slower than a 486/33, we're talking about a hundred-times speed difference from the low end to the high end. A 486/33 can animate a 72-sided ball, complete with shading (as discussed later), at 60 frames per second (fps), with plenty of cycles to spare; an 8-MHz AT can animate the same ball at about 6 fps. Clearly, the level of animation an application uses must be tailored to the available CPU horsepower.

The implementation of a 32-bit multiply using 8088 instructions is a simple matter of adding together four partial products. A 32-bit divide is not so simple, however. In fact, in Listing One I've chosen not to implement a full 32x32 divide, but rather only a 32x16 divide. The reason is simple: performance. A 32x16 divide can be implemented on an 8088 with two DIV instructions, but a 32x32 divide takes a great deal more work, so far as I can see. (If anyone has a fast 32x32 divide, or has a faster way to handle signed multiplies and divides than the approach taken by Listing One, please drop me a line.) In X-Sharp, division is used only to divide either X or Y

by Z in the process of projecting from view space to screen space, so the cost of using a 32x16 divide is merely some inaccuracy in calculating screen coordinates, especially when objects get very close to the Z=0 plane. This error is not cumulative (doesn't carry over to later frames), and in my experience doesn't cause noticeable image degradation; therefore, given the already slow performance of the 8088 and 286, I've opted for performance over precision.

At any rate, please keep in mind that the non-386 version of FixedDiv[unkeyable] is not a general-purpose 32x32 fixed-point division routine. In fact, it will generate a divide-by-zero error if passed a fixed-point divisor between -1 and 1. As I've explained, the non-386 version of FixedDiv[unkeyable] is designed to do just what X-Sharp needs, and no more, as quickly as possible.

Shading

So far, the polygons out of which our animated objects have been built have had colors of fixed intensities. For example, a face of a cube might be blue, or green, or white, but whatever color it is, that color never brightens or dims. Fixed colors are easy to implement, but they don't make for very realistic animation. In the real world, the intensity of the color of a surface varies depending on how brightly it is illuminated. The ability to simulate the illumination of a surface, or shading, is the next feature we'll add to X-Sharp.

The overall shading of an object is the sum of several types of shading components. Ambient shading is illumination by what you might think of as background light, light that's coming from all directions; all surfaces are equally illuminated by ambient light, regardless of their orientation. Directed lighting, producing diffuse shading, is illumination from one or more specific light sources. Directed light has a specific direction, and the angle at which it strikes a surface determines how brightly it lights that surface. Specular reflection is the tendency of a surface to reflect light in a mirrorlike fashion. There are other sorts of shading components, including transparency and atmospheric effects, but the ambient- and diffuse-shading components are all we're going to deal with for a while, with specular shading not too far in the future.

Ambient Shading

The basic model for both ambient and diffuse shading is a simple one. Each surface has a reflectivity between 0 and 1, where 0 means all light is absorbed and 1 means all light is reflected. A certain amount of light energy strikes each surface. The energy (intensity) of the light is expressed such that if light of intensity 1 strikes a surface with reflectivity 1, then the brightest possible shading is displayed for that surface. Complicating this somewhat is the need to support color; we do this by separating reflectance and shading into three components each--red, green, and blue--and calculating the shading for each color component separately for each surface.

Given an ambient-light red intensity of [IA.sub.red] and a surface red reflectance [R.sub.red'], the displayed red ambient shading for that surface, as a fraction of the maximum red intensity, is simply min([IA.sub.red] x [R.sub.red'] 1). The green and blue color components are handled similarly. That's really all there is to ambient shading, although of course we must design some way to map displayed color components into the available palette of colors, which I'll discuss next month. Ambient shading isn't the whole shading picture, though. In fact, scenes tend to look pretty bland without diffuse shading.

Diffuse Shading

Diffuse shading is more complicated than ambient shading, because the effective intensity of directed light falling on a surface depends on the angle at which it strikes the surface. According to Lambert's law, the light energy from a directed light source striking a surface is proportional to the cosine of the angle at which it strikes the surface, with the angle measured

relative to a vector perpendicular to the polygon (a polygon normal), as shown in Figure 1. If the red intensity of directed light is [ID.sub.red,] the red reflectance of the surface is [R.sub.red,] and the angle between the incoming directed light and the surface's normal is [unkeyable], then the displayed red diffuse shading for that surface, as a fraction of the largest possible red intensity, is min ([ID.sub.red] X [R.sub.red] X cos[unkeyable], 1).

That's easy enough to calculate--but seemingly slow. Determining the cosine of an angle can be sped up with a table lookup, but there's also the task of figuring out the angle, and, all in all, it doesn't seem that diffuse shading is going to be speedy enough for our purposes. Consider this, however: According to the properties of the dot product (denoted by the operator "[unkeyable]", as shown in Figure 2), cos [(unkeyable)] = (v[unkeyable]w)/[v]x[w]), where v and w are vectors, [unkeyable] is the angle between v and w, and [v] is the length of v. Suppose, now, that v and w are unit vectors; that is, vectors exactly one unit long. Then the above equation reduces to cos[unkeyable] =v[unkeyable]w. In other words, we can calculate the cosine between N, the unit-normal vector (one-unit-long perpendicular vector) of a polygon, and L', the reverse of a unit vector describing the direction of a light source, with just three multiples and two adds. (The reason the light-direction vector must be reversed is explained later.) Once we have that, we can easily calculate the red diffuse shading from a directed light source as min([ID.sub.red]) x [R.sub.red] x (L'[unkeyable]N), 1) and likewise for the green and blue color components.

The overall red shading for each polygon can be calculated by summing the ambient-shading red component with the diffuse-shading component from each light source, as in min(([IA.sub.red] X [R.sub.red] + ([ID.sub.red0] x [R.sub.red] x ([L.sub.0'][unkeyable]N)) + ([ID.sub.red] x [R.sub.red]x([L.sub.1'][unkeyable]N))+...), 1) where [ID.sub.red0] and [L.sub.0'] are the red intensity and the reversed unit-direction vector, respectively, for spotlight 0. Listing Two, page 152, shows the X-Sharp module DRAWPOBJ.C, which performs ambient and diffuse shading. Toward the bottom, you will find the code that performs shading exactly as described by the above equation, first calculating the ambient red, green, and blue shadings, then summing that with the diffuse red, green, and blue shadings generated by each directed light source.

Shading: Implementation Details

In order to calculate the cosine of the angle between an incoming light source and a polygon's unit normal, we must first have the polygon's unit normal. This could be calculated by generating a cross-product on two polygon edges to generate a normal, then calculating the normal's length and scaling to produce a unit normal. Unfortunately, that would require taking a square root, so it's not a desirable course of action. Instead, I've made a change to X-Sharp's polygon format. Now, the first vertex in a shaded polygon's vertex list is the end-point of a unit normal that starts at the second point in the polygon's vertex list, as shown in Figure 3. The first point isn't one of the polygon's vertices, but is used only to generate a unit normal. The second point, however, is a polygon vertex. Calculating the difference vector between the first and second points yields the polygon's unit normal. Adding a unit-normal endpoint to each polygon isn't free; each of those endpoints has to be transformed, along with the rest of the vertices, and that takes time. Still, it's faster than calculating a unit normal for each polygon from scratch.

We also need a unit vector for each directed light source. The directed light sources I've implemented in X-Sharp are spotlights; that is, they're considered to be point light sources that are infinitely far away. This allows the simplifying assumption that all light rays from a spotlight are parallel and of equal intensity throughout the displayed universe, so each spotlight can be represented with a single unit vector and a single intensity. The only trick is that in order to calculate the desired cos([unkeyable]) between the polygon unit normal and a spotlight's unit vector, the direction of the spotlight's unit vector must be reversed, as shown in Figure 4. This is necessary because the dot product implicitly places vectors with their start points at the same location when it's used to calculate the cosine of the angle between two vectors. The light vector is incoming to the polygon surface, and the unit normal is outbound, so only by reversing one vector or the other will we get the cosine of the desired angle.

Given the two unit vectors, it's a piece of cake to calculate intensities, as shown in Listing Two. The sample program DEMO1, in the X-Sharp archive (built by running K1.BAT), puts the shading code to work displaying a rotating ball with ambient lighting and three spot lighting sources that the user can turn on and off. What you'll see when you run DEMO1 is that the shading is very good--face colors change very smoothly indeed--so long as only green lighting sources are on. However, if you combine spotlight two, which is blue, with any other light source, polygon colors will start to shift abruptly and unevenly. As configured in the demo, the palette supports a wide range of shading intensities for a pure version of any one of the three primary colors, but a very limited number of intensity steps (four, in this case) for each color component when two or more primary colors are mixed. While this situation can be improved, it is fundamentally a result of the restricted capabilities of the 256-color palette, and there is only so much that can be done without a larger color set. Next month, I'll talk about some ways to improve the quality of 256-color shading.

Shading problems pretty much vanish in 15-bpp or better modes, such as the 32K-color mode supported by the Sierra Hicolor DAC. I've designed X-Sharp's color model looking forward to this emerging generation of highly color-capable adapters, and DEMO1 gives you a taste of how terrific shading will look on such adapters.

More on colors next month.

Where to get X-Sharp

The full source for X-Sharp is available electronically as XSHARPn.ARC in the DDJ Forum on CompuServe, on M&T Online (see "Availability," page 3), and in the graphic.disp conference on Bix (as a ZIP file). Alternatively, you can send me a 360K or 720K formatted diskette and an addressed, stamped diskette mailer, care of DDJ, 411 Borel Ave., San Mateo, CA 94402, and I'll send you the latest copy of X-Sharp. There's no charge, but it'd be very much appreciated if you'd slip in a dollar or so to help out the folks at the Vermont Association for the Blind and Visually Impaired. Your response so far has been great. I just took a bundle of checks and money over to VABVI this week, and they were very happy indeed. Thanks!

I'm available on a daily basis to discuss X-Sharp on M&T Online and Bix (user name mabrash in both cases.)

Graphics Debugging Update

A while back, I lamented that Turbo Debugger had some annoying quirks when it came to debugging graphics in a dual-monitor set-up (using the -do switch). However, most of these quirks can be worked around by combining two of TD's display-handling modes. As I reported, TD blocks manual access (performed via the I/O menu in the CPU window) to some of the VGA's registers, and sometimes alters the page flipping and other registers when break-points occur. As it turns out, however, all that only happens in Smart display mode. If you use None display mode, TD doesn't touch any VGA registers at all. The drawback to using None mode is that in None mode, for some reason, program output via DOS functions goes to the debugging screen rather than the target screen. However, you can actively switch between Smart and None mode via the Options menu, so a reasonably complete solution is to start out in Smart mode, then switch to None mode when you encounter problems in manually accessing the VGA's registers from TD, or if you think TD is otherwise interfering with the state of the VGA.

Recommended Reading

Andrew Glassner's Graphics Gems (Academic Press, 1990) is an oddly enjoyable book. Odd, because there's no overall coherency to the book; it's a collection of more than 100 largely unrelated contributions by various authors on a hodgepodge of graphics subjects. Enjoyable, because it's that rarest sort of graphics programming book: one that you can open at random and start reading for fun. A good example of the nature of Graphics Gems is a chapter on mapping RGB colors into a 4-bit color space; this chapter features somewhat arcane theory, an interesting perspective on color space,

and a fast technique for RGB mapping in 16-color modes. On balance the chapter is a little uneven, but useful, informative, and interesting--a description that would serve well for Graphics Gems as a whole, as well.

- -

GRAPHICS PROGRAMMING COLUMN
by Michael Abrash

[LISTING ONE]

```
; Fixed point multiply and divide routines. Tested with TASM 3.0.
USE386      equ 1   ;1 for 386-specific opcodes, 0 for 8088 opcodes
MUL_ROUNDING_ON equ 1   ;1 for rounding on multiplies, 0 for no
                                ; rounding. Not rounding is faster, rounding is
                                ; more accurate and generally a good idea
DIV_ROUNDING_ON equ 0   ;1 for rounding on divides, 0 for no rounding.
                                ; Not rounding is faster, rounding is more
                                ; accurate, but because division is only
                                ; performed to project to the screen, rounding
                                ; quotients generally isn't necessary
ALIGNMENT   equ 2
    .model small
    .386
    .code
;=======================================================================
; Multiplies two fixed-point values together. C near-callable as:
;   Fixedpoint FixedMul(Fixedpoint M1, Fixedpoint M2);
FMparms struc
    dw  2 dup(?)    ;return address & pushed BP
M1  dd  ?
M2  dd  ?
FMparms ends
    align   ALIGNMENT
    public  _FixedMul
_FixedMul       proc    near
        push    bp
        mov     bp,sp
if USE386
        mov     eax,[bp+M1]
        imul    dword ptr [bp+M2] ;multiply
if MUL_ROUNDING_ON
        add     eax,8000h       ;round by adding 2^(-17)
        adc     edx,0           ;whole part of result is in DX
endif ;MUL_ROUNDING_ON
        shr     eax,16          ;put the fractional part in AX
else    ;!USE386
                ;do four partial products and add them
                ; together, accumulating the result in CX:BX
    push    si      ;preserve C register variables
    push    di
               ;figure out signs, so we can use unsigned multiplies
    sub cx,cx       ;assume both operands positive
    mov ax,word ptr [bp+M1+2]
    mov si,word ptr [bp+M1]
    and ax,ax       ;first operand negative?
    jns CheckSecondOperand ;no
    neg ax      ;yes, so negate first operand
    neg si
    sbb ax,0
    inc cx      ;mark that first operand is negative
CheckSecondOperand:
    mov bx,word ptr [bp+M2+2]
    mov di,word ptr [bp+M2]
    and bx,bx       ;second operand negative?
    jns SaveSignStatus  ;no
    neg bx      ;yes, so negate second operand
    neg di
    sbb bx,0
    xor cx,1        ;mark that second operand is negative
SaveSignStatus:
    push    cx      ;remember sign of result; 1 if result
                ; negative, 0 if result nonnegative
```

```
    push    ax      ;remember high word of M1
    mul bx      ;high word M1 times high word M2
    mov cx,ax       ;accumulate result in CX:BX (BX not used until
                            ; next operation) assume no overflow into DX
    mov ax,si       ;low word M1 times high word M2
    mul bx
    mov bx,ax
    add cx,dx       ;accumulate result in CX:BX
    pop ax      ;retrieve high word of M1
    mul di      ;high word M1 times low word M2
    add bx,ax
    adc cx,dx       ;accumulate result in CX:BX
    mov ax,si       ;low word M1 times low word M2
    mul di
if MUL_ROUNDING_ON
        add     ax,8000h    ;round by adding 2^(-17)
    adc bx,dx
else ;!MUL_ROUNDING_ON
    add bx,dx       ;don't round
endif ;MUL_ROUNDING_ON
    adc cx,0        ;accumulate result in CX:BX
    mov dx,cx
    mov ax,bx
    pop cx
    and cx,cx       ;is the result negative?
    jz  FixedMulDone    ;no, we're all set
    neg dx      ;yes, so negate DX:AX
    neg ax
    sbb dx,0
FixedMulDone:
    pop di      ;restore C register variables
    pop si
endif   ;USE386
        pop     bp
        ret
_FixedMul       endp
;============================================================================
; Divides one fixed-point value by another. C near-callable as:
;   Fixedpoint FixedDiv(Fixedpoint Dividend, Fixedpoint Divisor);
FDparms struc
    dw  2 dup(?)    ;return address & pushed BP
Dividend dd ?
Divisor  dd ?
FDparms ends
    align   ALIGNMENT
    public  _FixedDiv
_FixedDiv       proc    near
        push    bp
        mov     bp,sp
if USE386
if DIV_ROUNDING_ON
        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 the 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:
else ;!DIV_ROUNDING_ON
    mov edx,[bp+Dividend]
    sub eax,eax
    shrd    eax,edx,16  ;position so that result ends up
    sar edx,16      ; in EAX
    idiv    dword ptr [bp+Divisor]
endif ;DIV_ROUNDING_ON
    shld    edx,eax,16  ;whole part of result in DX;
                    ; fractional part is already in AX
else ;!USE386
;NOTE!!! Non-386 division uses a 32-bit dividend but only the upper 16 bits
; of the divisor; in other words, only the integer part of the divisor is
; used. This is done so that the division can be accomplished with two fast
; hardware divides instead of a slow software implementation, and is (in my
; opinion) acceptable because division is only used to project points to the
; screen (normally, the divisor is a Z coordinate), so there's no cumulative
; error, although there will be some error in pixel placement (the magnitude
; of the error is less the farther away from the Z=0 plane objects are). This
; is *not* a general-purpose divide, though; if the divisor is less than 1,
; for instance, a divide-by-zero error will result! For this reason, non-386
; projection can't be performed for points closer to the viewpoint than Z=1.
                ;figure out signs, so we can use
                ; unsigned divisions
    sub cx,cx       ;assume both operands positive
    mov ax,word ptr [bp+Dividend+2]
    and ax,ax       ;first operand negative?
    jns CheckSecondOperandD ;no
    neg ax      ;yes, so negate first operand
    neg word ptr [bp+Dividend]
    sbb ax,0
    inc cx      ;mark that first operand is negative
CheckSecondOperandD:
    mov bx,word ptr [bp+Divisor+2]
    and bx,bx       ;second operand negative?
    jns SaveSignStatusD ;no
    neg bx      ;yes, so negate second operand
    neg word ptr [bp+Divisor]
    sbb bx,0
    xor cx,1        ;mark that second operand is negative
SaveSignStatusD:
    push    cx      ;remember sign of result; 1 if result
                ; negative, 0 if result nonnegative
    sub dx,dx       ;put Dividend+2 (integer part) in DX:AX
    div bx      ;first half of 32/16 division, integer part
                ; divided by integer part
    mov cx,ax       ;set aside integer part of result
    mov ax,word ptr [bp+Dividend] ;concatenate the fractional part of
                ; the dividend to the remainder (fractional
                ; part) of the result from dividing the
                ; integer part of the dividend
    div bx      ;second half of 32/16 division
if DIV_ROUNDING_ON EQ 0
    shr bx,1        ;divisor/2, minus 1 if the divisor is
    adc bx,0        ; even
    dec bx
    cmp bx,dx       ;set Carry if the remainder is at least
    adc ax,0        ; half as large as the divisor, then
    adc cx,0        ; use that to round up if necessary
endif ;DIV_ROUNDING_ON
    mov dx,cx       ;absolute value of result in DX:AX
    pop cx
    and cx,cx           ;is the result negative?
    jz  FixedDivDone    ;no, we're all set
    neg dx              ;yes, so negate DX:AX
    neg ax
    sbb dx,0
FixedDivDone:
endif ;USE386
    pop bp
```

```
        ret
_FixedDiv       endp
    end
```

[LISTING TWO]

```
/* Draws all visible faces in the specified polygon-based object. The object
   must have previously been transformed and projected, so that all vertex
   arrays are filled in. Ambient and diffuse shading are supported. */
#include "polygon.h"

void DrawPObject(PObject * ObjectToXform)
{
   int i, j, NumFaces = ObjectToXform->NumFaces, NumVertices;
   int * VertNumsPtr, Spot;
   Face * FacePtr = ObjectToXform->FaceList;
   Point * ScreenPoints = ObjectToXform->ScreenVertexList;
   PointListHeader Polygon;
   Fixedpoint Diffusion;
   ModelColor ColorTemp;
   ModelIntensity IntensityTemp;
   Point3 UnitNormal, *NormalStartpoint, *NormalEndpoint;
   long v1, v2, w1, w2;
   Point Vertices[MAX_POLY_LENGTH];

   /* Draw each visible face (polygon) of the object in turn */
   for (i=0; i<NumFaces; i++, FacePtr++) {
      /* Remember where we can find the start and end of the polygon's
         unit normal in view space, and skip over the unit normal endpoint
         entry. The end and start points of the unit normal to the polygon
         must be the first and second entries in the polgyon's vertex list.
         Note that the second point is also an active polygon vertex */
      VertNumsPtr = FacePtr->VertNums;
      NormalEndpoint = &ObjectToXform->XformedVertexList[*VertNumsPtr++];
      NormalStartpoint = &ObjectToXform->XformedVertexList[*VertNumsPtr];
      /* Copy over the face's vertices from the vertex list */
      NumVertices = FacePtr->NumVerts;
      for (j=0; j<NumVertices; j++)
         Vertices[j] = ScreenPoints[*VertNumsPtr++];
      /* Draw only if outside face showing (if the normal to the polygon
         in screen coordinates 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 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;
         }
         /* See if there's any shading */
            if (FacePtr->ShadingType == 0) {
            /* No shading in effect, so just draw */
            DRAW_POLYGON(Vertices, NumVertices, FacePtr->ColorIndex, 0, 0);
         } else {
            /* Handle shading */
            /* Do ambient shading, if enabled */
            if (AmbientOn && (FacePtr->ShadingType & AMBIENT_SHADING)) {
               /* Use the ambient shading component */
               IntensityTemp = AmbientIntensity;
            } else {
               SET_INTENSITY(IntensityTemp, 0, 0, 0);
            }
            /* Do diffuse shading, if enabled */
            if (FacePtr->ShadingType & DIFFUSE_SHADING) {
               /* Calculate the unit normal for this polygon, for use in dot
                  products */
               UnitNormal.X = NormalEndpoint->X - NormalStartpoint->X;
               UnitNormal.Y = NormalEndpoint->Y - NormalStartpoint->Y;
               UnitNormal.Z = NormalEndpoint->Z - NormalStartpoint->Z;
               /* Calculate the diffuse shading component for each active
                  spotlight */
               for (Spot=0; Spot<MAX_SPOTS; Spot++) {
                  if (SpotOn[Spot] != 0) {
                     /* Spot is on, so sum, for each color component, the
                        intensity, accounting for the angle of the light rays
                        relative to the orientation of the polygon */
                     /* Calculate cosine of angle between the light and the
                        polygon normal; skip if spot is shining from behind
                        the polygon */
                     if ((Diffusion = DOT_PRODUCT(SpotDirectionView[Spot],
                           UnitNormal)) > 0) {
                        IntensityTemp.Red +=
                              FixedMul(SpotIntensity[Spot].Red, Diffusion);
                        IntensityTemp.Green +=
                              FixedMul(SpotIntensity[Spot].Green, Diffusion);
                        IntensityTemp.Blue +=
                              FixedMul(SpotIntensity[Spot].Blue, Diffusion);
                     }
                  }
               }
            }
            /* Convert the drawing color to the desired fraction of the
               brightest possible color */
            IntensityAdjustColor(&ColorTemp, &FacePtr->FullColor,
                  &IntensityTemp);
            /* Draw with the cumulative shading, converting from the general
               color representation to the best-match color index */
            DRAW_POLYGON(Vertices, NumVertices,
                  ModelColorToColorIndex(&ColorTemp), 0, 0);
         }
      }
   }
}
```

Journal: Dr. Dobb's Journal August 1992 v17 n8 p149(7)

Color-modeling in 256-color mode

Author: Abrash, Michael.
AttFile: GP-AUG92.ASC Source code listing.
XSHRP21.ZIP X-Sharp library for 3D graphics.

Abstract: When developing computer graphics software, it is necessary to define a set of colors that fit within the 256-color limit of the VGA display standard. The first step is to select a working color model. RGB is a good choice for modeling colors that are generated by light sources. Colors are defined in terms of a triplet code that specifies one of 256 levels of intensity of red, blue and green. While there are over 16 million possible combinations of color, the programmer must define a palette of 256 colors for use with VGA. One method of defining a palette is to program an application that utilizes a subset of RGB space, biasing the the VGA palette towards that subspace. Another method is to define colors by utilizing only two primaries. Programmers can also use one part of the palette for a range of colors and use the rest of the palette for fixed colors of non-shaded objects. While 256-color VGA presents limits, careful programming around these limitations can yield nearly 24-bit color quality.

Lately, my daughter has wanted some fairly sophisticated books read to her. Wind in the Willows. Little House on the Prairie. Pretty heady stuff for a six year old, and sometimes I wonder how much of it she really understands. As an experiment, during today's reading I stopped whenever I came to a word I thought she might not know, and asked her what it meant. One such word was "mulling."

"Do you know what 'mulling' means?" I asked.

She thought about it for a while, then said, "Pondering."

"Very good!" I said, more than a little surprised.

She smiled and said, "But, Dad, how do you know that I know what 'pondering' means?"

"Okay," I said, "What does 'pondering' means?"

"Mulling," she said.

What does this anecdote tell us about the universe in which we live? Well, it certainly indicates that this universe is inhabited by at least one comedian and one good straight man. Beyond that, though, it can be construed as a parable about the difficulty of defining things properly; for example, consider the complications inherent in the definition of color on a 256-color display adapter such as the VGA. Coincidentally, VGA color modeling just happens to be this month's topic, and the place to start is with color modeling in general.

A Color Model

We've been developing X-Sharp, a realtime 3-D animation package, for several months now. Last month, we added illumination sources and shading; that addition makes it necessary for us to have a general-purpose color model, so that we can display the gradations of color intensity necessary to render illuminated surfaces properly. In other words, when a bright light is shining straight at a green surface, we need to be able to display bright green, and as that light dims or tilts to strike the surface at a shallower angle, we need to be able to display progressively dimmer shades of green.

The first thing to do is to select a color model in which to perform our shading calculations, the dot product-based stuff I discussed last month. The approach we'll take is to select an ideal representation of the full color space and do our calculations there, as if we really could display

every possible color; only as a final step will we map each desired color into the limited 256-color set of the VGA, or the color range of whatever adapter we happen to be working with. There are a number of color models that we might choose to work with, but I'm going to go with the one that's both most familiar and, in my opinion, simplest: RGB (red, green, blue). In the RGB model, a given color is modeled as the mix of specific fractions of full intensities of each of the three color primaries. For example, the brightest possible pure blue is 0.0*R, 0.0*G, 1.0*B. Half-bright cyan is 0.0*R, 0.5*G, 0.5*B. Quarter-bright gray is 0.25*R, 0.25*G, 0.25*B. You can think of RGB color space as being a cube, as shown in Figure 1, with any particular color lying somewhere inside or on the cube.

RGB is good for modeling colors generated by light sources, because red, green, and blue are the additive primaries; that is, all other colors can be generated by mixing red, green, and blue light sources. They're also the primaries for color computer displays, and the RGB model maps beautifully onto the display capabilities of 15- and 24-bpp display adapters, which tend to represent pixels as RGB combinations in display memory.

How, then, are RGB colors represented in X-Sharp? Each color is represented as an RGB triplet, with eight bits each of red, green, and blue resolution, using the structure shown in Listing One (page 166). That is, each color is described by three color components--on each for red, green, and blue--and each primary color component is represented by eight bits. Zero intensity of a color component is represented by the value O, and full intensity is represented by the value 255. This gives us 256 levels of each primary color component, and a total of 16,772,216 possible colors.

Holy cow! Isn't 16,000,000-plus colors a bit of overkill?

Actually, no, it isn't. At the eighth Annual Computer Graphics Show in New York, this past January, Sheldon Linker, of Linker Systems, related an interesting tale about color perception research at the Jet Propulsion Lab back in the '70s. The JPL color research folks had the capability to print more than 50,000,000 distinct and very precise colors on paper. As a test, they tried printing our words in various colors, with each word printed on a background that differed by only one color index from the word's color. No one expected the human eye to be able to differentiate between two colors, out of 50,000,000-plus, that were so similar. It turned out, through, that everyone could read the words with no trouble at all; the human eye is surprisingly sensitive to color gradations, and also happens to be wonderful at detecting edges.

When the JPL team went to test the eye's sensitivity to color on the screen, they found that only about 16,000,000 colors could be distinguished, because the color-sensing mechanism of the human eye is more compatible with reflective sources such as paper and ink than with emissive sources such as CRTs. Still, the human eye can distinguish about 16,000,000 colors on the screen. That's not so hard to believe, if you think about it; the eye senses each primary color separately, so we're really only talking about detecting 256 levels of intensity per primary here. It's the brain that does the amazing part; the 16,000,000-plus color capability actually comes not from extraordinary sensitivity in the eye, but rather from the brain's ability to distinguish between all the mixes of 256 levels of each of three primaries.

So it's perfectly reasonable to maintain 24 bits of color resolution, and X-Sharp represents colors internally as ideal, device-independent 24-bit RGB triplets. All shading calculations are performed on these triplets, with 24-bit color precision. It's only after the final 24-bit RGB drawing color is calculated that the display adapter's color capabilities come into play, as the X-Sharp function ModelColorToColorIndex() is called to map the desired RGB color to the closest match the adapter is capable of displaying. Of course, that mapping is adapter dependent. On a 24-bpp device, it's pretty obvious how the internal RGB color format maps to displayed pixel colors: directly. On VGAs with 15-bpp Sierra Hicolor DACs, the mapping is equally simple, with the five upper bits of each color component mapping straight to display pixels. But how on earth do we map those 16,000,000-plus RGB colors into the 256-color space of a standard VGA?

This is the "color definition" problem I mentioned at the start of the column. The VGA palette is arbitrarily programmable to any set of 256 colors, with each color defined by six bits each of red, green, and blue

intensity. In X-Sharp, the function InitializePalette() can be customized to set up the palette however we wish; this gives us nearly complete flexibility in defining the working color set. Even with infinite flexibility, however, 256 out of 16,000,000 or so possible colors is a pretty puny selection. It's easy to set up the palette to give yourself a good selection of just blue intensities, or of just greens; but for general color modeling there's simply not enough palette to go around.

One way to deal with the limited simultaneous color capabilities of the VGA is to build an application that uses only a subset of RGB space, then bias the VGA's palette toward that subspace. This is the approach used in the DEMO1 sample program in X-Sharp; Listings Two and Three (page 166) show the versions of InitializePalette() and ModelColorToColorIndex() that set up and perform the color mapping for DEMO1. In DEMO1, three-quarters of the palette is set up with 64 intensity levels of each of the three pure primary colors (red, green, and blue), and then most drawing is done with only pure primary colors. The resulting rendering quality is very good, because there are so many levels of each primary.

The downside is that this excellent quality is available for only three colors: red, green, and blue. What about all the other colors that are mixes of the primaries, like, say, cyan or yellow, to say nothing of gray? In the DEMO1 color model, any RGB color that is not a pure primary is mapped into a 2-2-2 RGB space that the remaining quarter of the VGA's palette is set up to display; that is, there are exactly two bits of precision for each color component, or 64 general RGB colors in all. This is genuinely lousy color resolution, being only 1/64th of the resolution we really need for each color component. In this model, a staggering 262,144 colors from the 24-bit RGB cube map to each color in the 2-2-2 VGA palette. The results are not impressive; the colors of mixed-primary surfaces jump abruptly, badly damaging the illusion of real illumination. To see how poor a 2-2-2 RGB selection can look, run DEMO1, and press the '2' key to turn on spotlight 2, the blue spotlight. Because the ambient lighting is green, turning on the blue spotlight causes mixed-primary colors to be displayed--and the result looks terrible, because there just isn't enough color resolution. Unfortunately. 2-2-2 RGB is close to the best general color resolution the VGA can display.

Another approach would be to set up the palette with reasonably good mixes of two primaries but no mixes of three primaries, then use only two-primary colors in your applications (no grays or whites or other three-primary mixes). Or you could choose to shade only selected objects, using part of the palette for a good range of the colors of those objects, and reserving the rest of the palette for the fixed colors of the other, nonshaded objects. Jim Kent, author of Autodesk Animator, suggests dynamically adjusting the palette to the needs of each frame, for example by allocating the colors for each frame on a first-come, first-served basis. That wouldn't be trivial to do in real time, but it would make for extremely efficient use of the palette.

The sad truth is that the VGA's 256-color palette is an inadequate resource for general RGB shading. The good news is that clever workarounds can make VGA graphics look nearly as good as 24-bpp graphics; the burden falls on you, the programmer, to design your applications and color mapping to compensate for the VGA's limitations. To experiment with a different 256-color model in X-Sharp, just change InitializePalette() to set up the desired palette and ModelColorToColorIndex() to map 24-bit RGB triplets into the palette you've set up. It's that simple, and the results can be striking indeed.

Where to Get X-Sharp

The full source for X-Sharp is available in the file XSHP n.ARC in the DDJ Forum on CompuServe, and as XSHARPn .ZIP in both the programming/graphics conference on M&T Online and the graphic.disp conference on Bix. Alternatively, you can send me a 360K or 720K formatted diskette and an addressed, stamped diskette mailer, care of X-Sharp, DDJ, 411 Borel Ave., San Mateo, CA 94402, and I'll send you the latest copy of X-Sharp. There's no charge, but it'd be very much appreciated if you'd slip in a dollar or so to help out the folks at the Vermont Association for the Blind and Visually Impaired.

I'm available on a daily basis to discuss X-Sharp on M&T Online and Bix (user name mabrash in both cases).

Fast VGA Text

This next item comes from The BitMan. (That's how he asked to be described; don't ask me why.) The BitMan passed along a nifty application of the VGA's under-appreciated write mode 3 that is, under the proper circumstances, the fastest possible way to draw text in any 16-color VGA mode.

The task at hand is illustrated by Figure 2. We want to draw what's known as solid text, in which the effect is the same as if the cell around each character was drawn in the background color, and then each character was drawn on top of the background box. (This is in contrast to transparent text, where each character is drawn in the foreground color without disturbing the background.) Assume that each character fits in an eight-wide cell (as is the case with the standard VGA fonts), and that we're drawing text at byte-aligned locations in display memory.

Solid text is useful for drawing menus, text areas, and the like; basically, it can be used whenever you want to display text on a solid-color background. The obvious way to implement solid text is to fill the rectangle representing the background box, then draw transparent text on top of the background box. However, there are two problems with doing solid text this way. First, there's some flicker, because for a little while the box is there but the text hasn't yet arrived. More important is that the background-followed-by-foreground approach accesses display memory three times for each byte of font data: once to draw the background box, once to read display memory to load the latches, and once to actually draw the font pattern. Display memory is incredibly slow, so we'd like to reduce the number of accesses as much as possible. With The BitMan's approach, we can reduce the number of accesses to just one per font byte, and eliminate flicker, too.

The keys to fast solid text are the latches and write mode 3. The latches, as you may recall from earlier discussions in this column, are four internal VGA registers that hold the last bytes read from the VGA's four planes; every read from VGA memory loads the latches with the values stored at that display memory address across the four planes. Whenever a write is performed to VGA memory, the latches can provide some, none, or all of the bits written to memory, depending on the bit mask, which selects between the latched data and the drawing data on a bit-by-bit basis. The latches solve half our problem; we can fill the latches with the background color, then use them to draw the background box. The trick now is drawing the text pixels in the foreground color at the same time.

This is where it gets a little complicated. In write mode 3 (which incidentally is not available on the EGA), each byte value that the CPU writes to the VGA does not get written to display memory. Instead, it turns into the bit mask. (Actually, it's ANDed with the Bit Mask register, and the result becomes the bit mask, but we'll leave the Bit Mask register set to 0xFF, so the CPU value will become the bit mask.) The bit mask selects, on a bit-by-bit basis, between the data in the latches for each plane (the previously loaded background color, in this case) and the foreground color. Where does the foreground color come from, if not from the CPU? From the Set/Reset register, as shown in Figure 3. Thus, each byte written by the CPU (font data, presumably) selects foreground or background color for each of eight pixels, all done with a single write to display memory.

I know this sounds pretty esoteric, but think of it this way. The latches hold the background color in a form suitable for writing eight background pixels (one full byte) at a pop. Write mode 3 allows each CPU byte to punch holes in the background color provided by the latches, holes through which the foreground color from the Set/Reset register can flow. The result is that a single write draws exactly the combination of foreground and background pixels described by each font byte written by the CPU. It may help to look at Listing Four (page 167), which shows The BitMan's technique in action. And yes, this technique is absolutely worth the trouble; it's about three times faster than the fill-then-draw approach described above, and about twice as fast as transparent text. So far as I know, there is no faster way to draw text on a VGA.

It's important to note that The BitMan's technique only works on full bytes of display memory. There's no way to clip to finer precision; the background color will inevitably flood all of the eight destination pixels that aren't selected as foreground pixels. This makes The BitMan's technique most

suitable for monospaced fonts with characters that are multiples of eight pixels in width, and for drawing to byte-aligned addresses; the technique can be used in other situations, but is considerably more difficult to apply.

At this point, some of you are no doubt nodding your heads and saying, "Yes, I see how that would work." Others are probably muttering, "Well, heck, I knew that; tell me something new." Then there are the rest of you, the VGA neophytes, the ones with glazed eyes, who think this technique sounds interesting, but understand maybe 30 percent of what you just read. Where can you turn for help?

Read on.

Useful VGA Reading

For years, I've recommended Richard Wilton's Programmer's Guide to PC and PS/2 Video Systems (Microsoft Press, 1987, $24.95, ISBN 1-55615-103-9) as a VGA-programming reference, not because it's perfect, but because it was the only VGA book I knew of that was good enough to be useful. I've added another book to my good-enough-to-get list: Programmer's Guide to the EGA and VGA Cards, Second Edition, by Richard Ferraro (Addison-Wesley, 1990, $29.95, ISBN 0-201-57025-4). This 1000-plus page tome has a wide variety of valuable VGA information, ranging from registers to BIOS functions to the specifics of seven manufacturers' Super-VGA implementations, and it has plenty of good figures. This is, without question, a useful book. However, it is not (sigh), the ultimate VGA reference I've awaited for five years. The book is not error-free, especially regrettable in a second edition; for example, the polarity of the bits in the Color Don't Care register are reversed in the discussion of read mode 1. Also, although write mode 3 and the latches are covered, they're discussed in considerably less detail than I'd like to see; you'd have a tough time figuring out why The BitMan's technique works from this book alone. And surely Ferraro knows that you have to read display memory to load the latches before the bit mask can do its job of protecting selected pixels within a destination byte, because he shows line-drawing code that does just that. Still, he keeps saying that the bit mask keeps destination pixels from being modified, as if that happens even if you don't read display memory first. Nonetheless, I've found this book useful when I've reached for it, and I've found myself reaching for it increasingly often, and that's the real test of any reference. If you're a PC graphics programmer, you should probably have this book on your shelf.

- - - -

```
   -------------------------
   |1  0 |     Green       |
   -------------------------
    7  6  5  4  3  2  1  0

   -------------------------
   |1  1 |      Blue       |
   -------------------------
    7  6  5  4  3  2  1  0

   Colors are gamma corrected for a gamma of 2.3 to provide approximately
   even intensity steps on the screen.
*/

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

static unsigned char Gamma4Levels[] = { 0, 39, 53, 63 };
static unsigned char Gamma64Levels[] = {
    0, 10, 14, 17, 19, 21, 23, 24, 26, 27, 28, 29, 31, 32, 33, 34,
   35, 36, 37, 37, 38, 39, 40, 41, 41, 42, 43, 44, 44, 45, 46, 46,
   47, 48, 48, 49, 49, 50, 51, 51, 52, 52, 53, 53, 54, 54, 55, 55,
   56, 56, 57, 57, 58, 58, 59, 59, 60, 60, 61, 61, 62, 62, 63, 63,
};

static unsigned char PaletteBlock[256][3];   /* 256 RGB entries */

void InitializePalette()
{
   int Red, Green, Blue, Index;
   union REGS regset;
   struct SREGS sregset;

   for (Red=0; Red<4; Red++) {
      for (Green=0; Green<4; Green++) {
         for (Blue=0; Blue<4; Blue++) {
            Index = (Red<<4)+(Green<<2)+Blue;
            PaletteBlock[Index][0] = Gamma4Levels[Red];
            PaletteBlock[Index][1] = Gamma4Levels[Green];
            PaletteBlock[Index][2] = Gamma4Levels[Blue];
         }
      }
   }

   for (Red=0; Red<64; Red++) {
      PaletteBlock[64+Red][0] = Gamma64Levels[Red];
      PaletteBlock[64+Red][1] = 0;
      PaletteBlock[64+Red][2] = 0;
   }

   for (Green=0; Green<64; Green++) {
      PaletteBlock[128+Green][0] = 0;
      PaletteBlock[128+Green][1] = Gamma64Levels[Green];
      PaletteBlock[128+Green][2] = 0;
   }

   for (Blue=0; Blue<64; Blue++) {
      PaletteBlock[192+Blue][0] = 0;
      PaletteBlock[192+Blue][1] = 0;
      PaletteBlock[192+Blue][2] = Gamma64Levels[Blue];
   }

   /* Now set up the palette */
   regset.x.ax = 0x1012;   /* set block of DAC registers function */
   regset.x.bx = 0;        /* first DAC location to load */
   regset.x.cx = 256;      /* # of DAC locations to load */
   regset.x.dx = (unsigned int)PaletteBlock; /* offset of array from which
                                                to load RGB settings */
   sregset.es = _DS; /* segment of array from which to load settings */
   int86x(0x10, &regset, &regset, &sregset); /* load the palette block */
}
```

[LISTING THREE]

```
/* Converts a model color (a color in the RGB color cube, in the current
   color model) to a color index for mode X. Pure primary colors are
   special-cased, and everything else is handled by a 2-2-2 model. */
int ModelColorToColorIndex(ModelColor * Color)
{
   if (Color->Red == 0) {
      if (Color->Green == 0) {
         /* Pure blue */
         return(192+(Color->Blue >> 2));
      } else if (Color->Blue == 0) {
         /* Pure green */
         return(128+(Color->Green >> 2));
      }
   } else if ((Color->Green == 0) && (Color->Blue == 0)) {
      /* Pure red */
      return(64+(Color->Red >> 2));
   }
   /* Multi-color mix; look up the index with the two most significant bits
      of each color component */
   return(((Color->Red & 0xC0) >> 2) | ((Color->Green & 0xC0) >> 4) |
         ((Color->Blue & 0xC0) >> 6));
}
```

[LISTING FOUR]

```
; Demonstrates drawing solid text on the VGA, using The BitMan's write mode
; 3-based, one-pass technique. Tested with TASM 3.0 and MASM 5.1.

CHAR_HEIGHT equ 8   ;# of scan lines per character (must be <256)
SCREEN_HEIGHT   equ 480 ;# of scan lines per screen
SCREEN_SEGMENT  equ 0a000h  ;where screen memory is
FG_COLOR    equ 14  ;text color
BG_COLOR    equ 1   ;background box color
GC_INDEX    equ 3ceh    ;Graphics Controller (GC) Index reg I/O port
SET_RESET   equ 0   ;Set/Reset register index in GC
G_MODE      equ 5   ;Graphics Mode register index in GC
BIT_MASK    equ 8   ;Bit Mask register index in GC

    .model  small
    .stack  200h
    .data
Line    dw  ?   ;current line #
CharHeight dw   ?   ;# of scan lines in each character (must be <256)
MaxLines dw ?   ;max # of scan lines of text that will fit on screen
LineWidthBytes dw ? ;offset from one scan line to the next
FontPtr dd  ?   ;pointer to font with which to draw
SampleString    label   byte
    db  'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
    db  'abcdefghijklmnopqrstuvwxyz'
    db  '0123456789!@#$%^&*(),<.>/?;:',0

    .code
start:
    mov ax,@data
    mov ds,ax

    mov ax,12h
    int 10h ;select 640x480 16-color mode

    mov ah,11h  ;BIOS character generator function
    mov al,30h  ;BIOS get font pointer subfunction
    mov bh,3    ;get 8x8 ROM font subsubfunction
    int 10h ;get the pointer to the BIOS 8x8 font
    mov word ptr [FontPtr],bp
    mov word ptr [FontPtr+2],es

    mov bx,CHAR_HEIGHT
    mov [CharHeight],bx ;# of scan lines per character
    mov ax,SCREEN_HEIGHT
```

```
    sub dx,dx
    div bx
    mul bx      ;max # of full scan lines of text that
    mov [MaxLines],ax   ; will fit on the screen

    mov ah,0fh          ;BIOS video status function
    int 10h                     ;get # of columns (bytes) per row
    mov al,ah           ;convert byte columns variable in
    sub ah,ah           ; AH to word in AX
    mov [LineWidthBytes],ax ;width of scan line in bytes
                                ;now draw the text
        sub     bx,bx
    mov [Line],bx   ;start at scan line 0
LineLoop:
    sub ax,ax       ;start at column 0; must be a multiple of 8
    mov ch,FG_COLOR ;color in which to draw text
    mov cl,BG_COLOR ;color in which to draw background box
    mov si,offset SampleString ;text to draw
    call    DrawTextString  ;draw the sample text
    mov bx,[Line]
    add bx,[CharHeight] ;# of next scan line to draw on
    mov [Line],bx
    cmp bx,[MaxLines]   ;done yet?
    jb  LineLoop    ;not yet

    mov ah,7
    int 21h     ;wait for a key press, without echo

    mov ax,03h
    int 10h ;back to text mode

    mov ah,4ch
    int 21h ;exit to DOS

; Draws a text string.
; Input: AX = X coordinate at which to draw upper left corner of first char
;    BX = Y coordinate at which to draw upper left corner of first char
;    CH = foreground (text) color
;    CL = background (box) color
;    DS:SI = pointer to string to draw, zero terminated
;    CharHeight must be set to the height of each character
;    FontPtr must be set to the font with which to draw
;        LineWidthBytes must be set to the scan line width in bytes
; Don't count on any registers other than DS, SS, and SP being preserved.
; The X coordinate is truncated to a multiple of 8. Characters are
; assumed to be 8 pixels wide.
    align   2
DrawTextString  proc    near
    cld
    shr ax,1        ;byte address of starting X within scan line
    shr ax,1
    shr ax,1
    mov di,ax
    mov ax,[LineWidthBytes]
    mul bx          ;start offset of initial scan line
    add di,ax       ;start offset of initial byte
    mov ax,SCREEN_SEGMENT
    mov es,ax       ;ES:DI = offset of initial character's
                    ; first scan line
                    ;set up the VGA's hardware so that we can
                    ; fill the latches with the background color
    mov dx,GC_INDEX
    mov ax,(0ffh SHL 8) + BIT_MASK
    out dx,ax   ;set Bit Mask register to 0xFF (that's the
                ; default, but I'm doing this just to make sure
                ; you understand that Bit Mask register and
                ; CPU data are ANDed in write mode 3)
    mov ax,(003h SHL 8) + G_MODE
    out dx,ax       ;select write mode 3
    mov ah,cl       ;background color
    mov al,SET_RESET
    out dx,ax       ;set the drawing color to background color
    mov byte ptr es:[0ffffh],0ffh ;write 8 pixels of the background
                    ; color to unused offscreen memory
```

```
    mov cl,es:[0ffffh]  ;read the background color back into the
                ; latches; the latches are now filled with
                ; the background color. The value in CL
                ; doesn't matter, we just needed a target
                ; for the read, so we could load the latches
    mov ah,ch   ;foreground color
    out dx,ax   ;set the Set/Reset (drawing) color to the
                ; foreground color
                ;we're ready to draw!
DrawTextLoop:
    lodsb           ;next character to draw
    and al,al       ;end of string?
    jz  DrawTextDone    ;yes
    push    ds      ;remember string's segment
    push    si      ;remember offset of next character in string
    push    di      ;remember drawing offset
                ;load these variables before we wipe out DS
    mov dx,[LineWidthBytes] ;offset from one line to next
    dec dx      ;compensate for STOSB
        mov     cx,[CharHeight] ;
        mul     cl              ;offset of character in font table
    lds si,[FontPtr]            ;point to font table
    add si,ax                   ;point to start of character to draw
                                ;the following loop should be unrolled for
                                ; maximum performance!
DrawCharLoop:   ;draw all lines of the character
    movsb       ;get the next byte of the character and draw
                ; character; data is ANDed with Bit Mask
                ; register to become bit mask, and selects
                ; between latch (containing the background
                ; color) and Set/Reset register (containing
                ; foreground color)
    add di,dx   ;point to next line of destination
    loop    DrawCharLoop

    pop di      ;retrieve initial drawing offset
    inc di      ;drawing offset for next char
    pop si      ;retrieve offset of next character in string
    pop ds      ;retrieve string's segment
    jmp DrawTextLoop    ;draw next character, if any

    align   2
DrawTextDone:           ;restore the Graphics Mode register to its
                ; default state of write mode 0
    mov dx,GC_INDEX
    mov ax,(000h SHL 8) + G_MODE
    out dx,ax       ;select write mode 0
    ret
DrawTextString  endp
    end start
```

Journal: Dr. Dobb's Journal Sept 1992 v17 n9 p141(7)

Pooh and the space station

Author: Abrash, Michael.
AttFile: GP-SEP92.ASC Source code listing.
XSHARP21.ZIP X-Sharp library for 3-D graphics.

Abstract: Texture mapping is accomplished by mapping an image into the surface of a polygon that has been transformed in the process of three-dimensional drawing. Rapid texture mapping can be accomplished by quickly identifying the value for each pixel in the destination polygon. The pixel values of the polygon are found by mapping each destination pixel in the transformed polygon back to its bitmapped image by using a reverse transformation. The backwards transformation is used in order to avoid gaps in the final image. If the destination polygon is rotated in two directions, each transformed vertex should be mapped to the corresponding vertex in the bitmap. The rapid texture mapping method uses a two-stage digital differential analyzer (DDA) approach. An example of the texture mapping method using the likeness of Winnie the Pooh is presented.

So, here's where Winnie the Pooh lives: in a space station orbiting Saturn. No, really; I have it straight from my daughter, and a six year old wouldn't make up something that important, would she? One day she wondered aloud, "Where is the Hundred Acre Wood, exactly?" and before I could give one of those boring parental responses about how it was imaginary--but A.A. Milne probably imagined it to be somewhere near London--my daughter announced that the Hundred Acre Wood was in a space station orbiting Saturn, and there you have it.

As it turns out, that's a very good location for the Hundred Acre Wood, leading to many exciting adventures for Pooh and Piglet. Consider the time they went down to the Jupiter gravity level (we're talking centrifugal force here; the station is spinning, of course) and nearly turned into pancakes of the Pooh and Piglet varieties, respectively. Or the time they drifted out into the free-fall area at the core and had to be rescued by humans with wings strapped on (a tip of the hat to Robert Heinlein here). Or the time they were caught up by the current in the river through the Wood and drifted for weeks around the circumference of the station, meeting many cultures and finding many adventures along the way. (Yes, Riverworld; no one said the stories you tell your children need to be purely original, just interesting.)

(If you think Pooh and Piglet in a space station is a tad peculiar, then I won't even mention Karla, the woman who invented agriculture, medicine, sanitation, reading and writing, peace, and just about everything else while traveling the length of the Americas with her mountain lion during the last Ice Age; or the Mars Cats and their trip in suspended animation to the Lesser Magellenic Cloud and beyond; or most assuredly Little Whale, the baby Universe Whale that is naughty enough to eat inhabited universes. But I digress.)

Anyway, I bring up Pooh and the space station because a great many people have asked me to discuss fast texture mapping. Texture mapping is the process of mapping an image (in our case, a bitmap) onto the surface of a polygon that's been transformed in the process of 3-D drawing. Up to this point, each polygon we've drawn in X-Sharp (the 3-D animation package we've been developing in this column) has been a single, solid color. Recently, we added the ability to shade polygons according to lighting, but each polygon was still a single color. Thus, in order to produce any sort of intricate design, a great many tiny polygons would have to be drawn. That would be very slow, so we need another approach. One such approach is texture mapping; that is, mapping the bitmap containing the desired image onto the pixels contained within the transformed polygon. Done properly, this should make it possible to change X-Sharp's output from a bland collection of monocolor facets to a lively, detailed, and much more realistic scene.

"What sort of scene?" you may well ask. This is where Pooh and the space station come in. When I sat down to think of a sample texture-mapping application, it occurred to me that the shaded ball we added to X-Sharp

recently looked at least a bit like a spinning, spherical space station, and that the single unshaded, yellow polygon looked somewhat like a window in the space station, and it might be a nice example if someone were standing in the window ....

The rest is history.

Principles of Quick-and-Dirty Texture Mapping

The key to our texture-mapping approach will be quickly determining what pixel value to draw for each pixel in the transformed destination polygon. These polygon pixel values will be determined by mapping each destination pixel in the transformed polygon back to the image bitmap, via a reverse transformation, and seeing what color resides at the corresponding location in the image bitmap, as shown in Figure 1. It might seem more intuitive to map pixels the other way, from the image bitmap to the transformed polygon, but in fact it's crucial that the mapping proceed backward from the destination, in order to avoid gaps in the final image. With the approach of finding the right value for each destination pixel in turn, via a backward mapping, there's no way we can miss any destination pixels. On the other hand, with the forward-mapping method, some destination pixels may be skipped or double-drawn, because this is not necessarily a one-to-one or onto mapping. Although we're not going to take advantage of it now, mapping back to the source makes it possible to average several neighboring image pixels together to calculate the value for each destination pixel; that is, to antialias the image. This can greatly improve texture quality, although it is slower.

Mapping Textures Made Easy

To understand how we're going to map textures, consider Figure 2, which maps a bitmapped image directly onto an untransformed polygon. Here, we simply map the origin of the polygon's object coordinate system somewhere within the image, then map the vertices to the corresponding image pixels. (For simplicity, I'll assume in this discussion that the polygon's coordinate system--its object coordinate system--is in units of pixels, but scaling images to polygons is eminently doable. This will become clearer when we look at mapping images into transformed polygons, below.) Mapping the image to the polygon is then a simple matter of stepping one scan line at a time in both the image and the polygon, each time advancing the X coordinates of the edges according to the slopes of the lines, just as is normally done when filling a polygon. Since the polygon is untransformed, the stepping is identical in both the image and the polygon, and the pixel mapping is one-to-one, so the appropriate part of each scan line of the image can simply be block copied to the destination.

Now matters get more complicated. What if the destination polygon is rotated in two dimensions? We no longer have a neat direct mapping from image scan lines to destination polygon scan lines. We still want to draw across each destination scan line, but the proper source pixels for each destination scan line may now track across the source bitmap at an angle, as shown in Figure 3. What to do?

The solution is remarkably simple. We'll just map each transformed vertex to the corresponding vertex in the bitmap; this is easy, because the vertices are at the same indices in the original and transformed vertex lists. Each time we select a new edge to scan for the destination polygon, we'll select the corresponding edge in the source bitmap, as well. Then--and this is crucial--each time we step a destination edge one scan line, we'll step the corresponding source image edge an equivalent amount.

Ah, but what is an "equivalent amount?" Think of it this way. If a destination edge is 100 scan lines high, so it will be stepped 100 times, then we'll divide the SourceXWidth and SourceYHeight lengths of the source edge by 100, and add those amounts to the source edge's coordinates each time the destination is stepped one scan line. Put another way, we have, as usual, arranged things so that in the destination polygon we step DestYHeight times, where DestYHeight is the height of the destination edge. The above approach arranges to step the source image edge DestYHeight times too, to match what the destination is doing, as shown in Figure 4.

Now we're able to track the coordinates of the polygon edges through the source image in tandem with the destination edges. Stepping across each destination scan line uses precisely the same technique. In the destination, we step DestXWidth times across each scan line of the polygon, once for each pixel on the scan line. (DestXWidth is the horizontal distance between the two edges being scanned on any given scan line.) To match this, we divide SourceXWidth and SourceYHeight (the lengths of the scan line in the source image, as determined by the source edge points we've been tracking, as described above) by the width of the destination scan line, DestXWidth, to produce SourceXStep and SourceYStep. Then, we just step DestXWidth times, adding SourceXStep and SourceYStep to SourceX and SourceY each time, and choose the nearest image pixel to (SourceX,SourceY) to copy to (DestX,DestY). (Note that the names used above, such as "SourceXWidth," are used for descriptive purposes, and don't necessarily correspond to the actual variable names used in Listing Two.)

That's a workable approach for 2-D rotated polygons--but what about 3-D rotated polygons, where the visible dimensions of the polygon can vary with 3-D rotation and perspective projection? First, I'd like to make it clear that texture mapping takes place from the source image to the destination polygon after the destination polygon is projected to the screen. That is, the image will be mapped after the destination polygon is in its final, drawable form. Given that, it should be apparent that the above approach automatically compensates for all changes in the dimensions of a polygon. You see, this approach divides source edges and scan lines into however many steps the destination polygon requires. If the destination polygon is much narrower than the source polygon, as a result of 3-D rotation and perspective projection, we just end up taking bigger steps through the source image and skipping a lot of source image pixels, as shown in Figure 5. The upshot is that the above approach handles all transformations and projections effortlessly. It could also be used to scale source images up to fit in larger polygons; all that's needed is a list of where the polygon's untransformed vertices map into the source image, and everything else happens automatically. In fact, mapping from any polygonal area of a bitmap to any destination polygon will work, given only that the two polygons have the same number of vertices.

Notes on DDA Texture Mapping

That's all there is to quick-and-dirty texture mapping. This technique basically uses a two-stage digital differential analyzer (DDA) approach to step through the appropriate part of the source image in tandem with the normal scan-line stepping through the destination polygon, so I'll call it "DDA texture mapping." It's worth noting that there is no need for any trigonometric functions at all, and only two divides are required per scan line.

This isn't a perfect approach, of course. For one thing, it isn't anywhere near as fast as drawing solid polygons; the speed is more comparable to drawing each polygon as a series of lines. Also, the DDA approach results in far from perfect image quality, since source pixels may be skipped or selected twice. I trust, however, that you can see how easy it would be to improve image quality by antialiasing with the DDA approach. For example, we could simply average the four surrounding pixels as we did for simple, unweighted antialiasing in this column last year. Or, we could take a Wu antialiasing approach (see my June column) and average the two bracketing pixels along each axis according to proximity. If we had cycles to waste (which, given that this is real-time animation on a PC, we don't), we could improve image quality by putting the source pixels through a low-pass filter sized in X and Y according to the ratio of the source and destination dimensions (that is, how much the destination is scaled up or down from the source).

Finally, I'd like to point out that this sort of DDA texture mapping is display-hardware dependent, because the bitmap for each image must be compatible with the number of bits per pixel in the destination. That's actually a fairly serious issue. One of the nice things about X-Sharp's polygon orientation is that, until now, the only display dependent part of X-Sharp has been the transformation from RGB color space to the adapter's color space. Compensation for aspect ratio, resolution, and the like all happens automatically in the course of projection. Still, we need the ability to display detailed surfaces, and it's hard to conceive of a fast way

to do so that's totally hardware independent. (If you know of one, drop me a line.)

For now, all we need is fast texture mapping of adequate quality, which the straightforward, non-antialiased DDA approach supplies. I'm sure there are many other fast approaches, and, as I've said, there are certainly better-looking approaches, but DDA texture mapping works well, given the constraints of the PC's horsepower. Next, we'll look at code that performs DDA texture mapping. First, though, I'd like to take a moment to thank Jim Kent, author of Autodesk Animator and a frequent contributor to this column, for getting me started with the DDA approach.

Fast Texture Mapping: An Implementation

As you might expect, I've implemented DDA texture mapping in X-Sharp. Listing One (page 164) shows the new header files and defines, and Listing Two (page 164) shows the actual texture-mapped polygon drawer. The set-pixel routine that Listing Two calls is a slight modification of the mode X set-pixel routine from my June 1991 column. In addition, INITBALL.C has been modified to create three texture-mapped polygons and define the texture bitmaps, and modifications have been made to allow the user to flip the axis of rotation. In short, you will need the complete X-Sharp archive (available as described below) to see texture mapping in action, but Listings One and Two are the actual texture mapping code in its entirety.

There's a lot I'd like to say about DDA texture mapping, and about the use of it in the DEMO1 program in X-Sharp, but I'll have to save most of it for next month because I'm running out of space. Here's the big thing: DDA texture mapping looks best on fast-moving surfaces, where the eye doesn't have time to pick nits with the shearing and aliasing that's an inevitable by-product of such a crude approach. Get the X-Sharp archive, compile DEMO1, and run it. The initial display looks okay, but certainly not great, because the rotational speed is so slow. Now press the S key a few times to speed up the rotation and flip between different rotation axes. I think you'll be amazed at how much better DDA texture mapping looks at high speed. This technique would be great for mapping textures onto hurtling asteroids or jets, but would come up short for slow, finely detailed movements.

No matter how you slice it, DDA texture mapping beats boring, single-color polygons nine ways to Sunday. The big downside is that it's much slower than a normal polygon fill; move the ball close to the screen in DEMO1, and watch things slow down when one of those big texture maps comes around. Of course, that's partly because the code is all in C; if I get a chance, I'll convert to assembler and turbocharge the DDA texture mapping for next time, and maybe speed up the general polygon- and rectangle-fill code, too. I'll try to get to that next month, along with a further discussion of texture mapping and attending to some rough spots that remain in the DDA texture mapping implementation, most notably in the area of exactly which texture pixels map to which destination pixels as a polygon rotates.

And, in case you're curious, yes, there is a bear in DEMO1. I wouldn't say he looks much like a Pooh-type bear, but he's a bear nonetheless. He does tend to look a little startled when you flip the ball around so that he's zipping by on his head, but, heck, you would too in the same situation. And remember, when you buy the next VGA megahit, Bears in Space, you saw it here first.

Where to Get X-Sharp

The full source for X-Sharp is available in the file XSHRP n.ZIP in the DDJ Forum on CompuServe and XSHARP n.ZIP in the programming/graphics conference on M&T Online and the graphic.disp conference on Bix. (XSHARP20 is the first version that includes texture mapping.) Alternatively, you can send me a 360K or 720K formatted diskette and an addressed, stamped diskette mailer, care of DDJ, 411 Borel Ave., San Mateo, CA 94402, and I'll send you the latest copy of X-Sharp. There's no charge, but it'd be very much appreciated if you'd slip in a dollar or so to help out the folks at the Vermont Association for the Blind and Visually Impaired.

I'm available on a daily basis to discuss X-Sharp on M&T Online and Bix (user name mabrash in both cases).

By the way, folks, I very much appreciate both your generous contributions to the Vermont Association for the Blind and your letters. However, I can't--honest-to-god, can't--write back to all of you. Not if I ever want to get any work done, anyway, and I have a family to feed. When I have time, I try to answer those of you who have written with questions, but there are no guarantees, especially given the flood of letters that have materialized in the wake of X-Sharp. You'll stand a lot better chance of getting a reply on MCI Mail, Bix, or M&T Online. Mind you, I do enjoy getting your letters, and the feedback and suggestions are great; it's just the responding part that's a problem. So keep those cards and letters (and e-mail messages) coming!

- -

GRAPHICS PROGRAMMING COLUMN
by Michael Abrash

[LISTING ONE]

```
/* New header file entries related to texture-mapped polygons */
/* Draws the polygon described by the point list PointList with a bitmap
   texture mapped onto it */
#define DRAW_TEXTURED_POLYGON(PointList,NumPoints,TexVerts,TexMap) \
   Polygon.Length = NumPoints; Polygon.PointPtr = PointList;       \
   DrawTexturedPolygon(&Polygon, TexVerts, TexMap);
#define FIXED_TO_INT(FixedVal) ((int) (FixedVal >> 16))
#define ROUND_FIXED_TO_INT(FixedVal) \
   ((int) ((FixedVal + DOUBLE_TO_FIXED(0.5)) >> 16))
/* Retrieves specified pixel from specified image bitmap of specified width. */
#define GET_IMAGE_PIXEL(TexMapBits, TexMapWidth, X, Y) \
   TexMapBits[(Y * TexMapWidth) + X]
/* Masks to mark shading types in Face structure */
#define NO_SHADING      0x0000
#define AMBIENT_SHADING 0x0001
#define DIFFUSE_SHADING 0x0002
#define TEXTURE_MAPPED_SHADING 0x0004
/* Describes a texture map */
typedef struct {
   int TexMapWidth;  /* texture map width in bytes */
   char *TexMapBits; /* pointer to texture bitmap */
} TextureMap;

/* Structure describing one face of an object (one polygon) */
typedef struct {
   int * VertNums;   /* pointer to list of indexes of this polygon's vertices
                        in the object's vertex list. The first two indexes
                        must select end and start points, respectively, of this
                        polygon's unit normal vector. Second point should also
                        be an active polygon vertex */
   int NumVerts;     /* # of verts in face, not including the initial
                        vertex, which must be the end of a unit normal vector
                        that starts at the second index in VertNums */
   int ColorIndex;   /* direct palette index; used only for non-shaded faces */
   ModelColor FullColor; /* polygon's color */
   int ShadingType;  /* none, ambient, diffuse, texture mapped, etc. */
   TextureMap * TexMap; /* pointer to bitmap for texture mapping, if any */
   Point * TexVerts; /* pointer to list of this polygon's vertices, in
                        TextureMap coordinates. Index n must map to index
                        n + 1 in VertNums, (the + 1 is to skip over the unit
                        normal endpoint in VertNums) */
} Face;
extern void DrawTexturedPolygon(PointListHeader *, Point *, TextureMap *);
```

[LISTING TWO]

```
/* Draws a bitmap, mapped to a convex polygon (draws a texture-mapped polygon.)
   "Convex" means that every horizontal line drawn through the polygon at any
   point would cross exactly two active edges (neither horizontal lines nor
   zero-length edges count as active edges; both are acceptable anywhere in
   the polygon), and that the right & left edges never cross. Nonconvex
   polygons won't be drawn properly. Can't fail. */
#include <stdio.h>
#include <math.h>
#include "polygon.h"
/* Describes the current location and stepping, in both the source and
   the destination, of an edge */
typedef struct {
   int Direction;    /* through edge list; 1 for a right edge (forward through
                        vertex list), -1 for a left edge (backward
                        through vertex list) */
   int RemainingScans;  /* height left to scan out in dest */
   int CurrentEnd;      /* vertex # of end of current edge */
   Fixedpoint SourceX;     /* current X location in source for this edge */
   Fixedpoint SourceY;     /* current Y location in source for this edge */
   Fixedpoint SourceStepX; /* X step in source for Y step in dest of 1 */
   Fixedpoint SourceStepY; /* Y step in source for Y step in dest of 1 */
                        /* variables used for all-integer Bresenham's-type
                           X stepping through the dest, needed for precise
                           pixel placement to avoid gaps */
   int DestX;           /* current X location in dest for this edge */
   int DestXIntStep;    /* whole part of dest X step per scan-line Y step */
   int DestXDirection;  /* -1 or 1 to indicate way X steps (left/right) */
   int DestXErrTerm;    /* current error term for dest X stepping */
   int DestXAdjUp;      /* amount to add to error term per scan line move */
   int DestXAdjDown;    /* amount to subtract from error term when the
                           error term turns over */
} EdgeScan;
int StepEdge(EdgeScan *);
int SetUpEdge(EdgeScan *, int);
void ScanOutLine(EdgeScan *, EdgeScan *);
int GetImagePixel(char *, int, int, int);
/* Statics to save time that would otherwise pass them to subroutines. */
static int MaxVert, NumVerts, DestY;
static Point * VertexPtr;
static Point * TexVertsPtr;
static char * TexMapBits;
static int TexMapWidth;
/* Draws a texture-mapped polygon, given a list of destination polygon
   vertices, a list of corresponding source texture polygon vertices, and a
   pointer to the source texture's descriptor. */
void DrawTexturedPolygon(PointListHeader * Polygon, Point * TexVerts,
   TextureMap * TexMap)
{
   int MinY, MaxY, MinVert, i;
   EdgeScan LeftEdge, RightEdge;
   NumVerts = Polygon->Length;
   VertexPtr = Polygon->PointPtr;
   TexVertsPtr = TexVerts;
   TexMapBits = TexMap->TexMapBits;
   TexMapWidth = TexMap->TexMapWidth;
   /* Nothing to draw if less than 3 vertices */
   if (NumVerts < 3) {
      return;
   }
   /* Scan through the destination polygon vertices and find the top of the
      left and right edges, taking advantage of our knowledge that vertices run
      in a clockwise direction (else this polygon wouldn't be visible due to
      backface removal) */
   MinY = 32767;
   MaxY = -32768;
   for (i=0; i<NumVerts; i++) {
      if (VertexPtr[i].Y < MinY) {
         MinY = VertexPtr[i].Y;
         MinVert = i;
      }
      if (VertexPtr[i].Y > MaxY) {
```

```
            MaxY = VertexPtr[i].Y;
            MaxVert = i;
         }
      }
      /* Reject flat (0-pixel-high) polygons */
      if (MinY >= MaxY) {
         return;
      }
      /* The destination Y coordinate is not edge specific; it applies to
         both edges, since we always step Y by 1 */
      DestY = MinY;
      /* Set up to scan the initial left and right edges of the source and
         destination polygons. We always step the destination polygon edges
         by one in Y, so calculate the corresponding destination X step for
         each edge, and then the corresponding source image X and Y steps */
      LeftEdge.Direction = -1;   /* set up left edge first */
      SetUpEdge(&LeftEdge, MinVert);
      RightEdge.Direction = 1;   /* set up right edge */
      SetUpEdge(&RightEdge, MinVert);
      /* Step down destination edges one scan line at a time. At each scan
         line, find the corresponding edge points in the source image. Scan
         between the edge points in the source, drawing the corresponding
         pixels across the current scan line in the destination polygon. (We
         know which way the left and right edges run through the vertex list
         because visible (non-backface-culled) polygons always have the vertices
         in clockwise order as seen from the viewpoint) */
      for (;;) {
         /* Done if off bottom of clip rectangle */
         if (DestY >= ClipMaxY) {
            return;
         }
         /* Draw only if inside Y bounds of clip rectangle */
         if (DestY >= ClipMinY) {
            /* Draw the scan line between the two current edges */
            ScanOutLine(&LeftEdge, &RightEdge);
         }
         /* Advance the source and destination polygon edges, ending if we've
            scanned all the way to the bottom of the polygon */
         if (!StepEdge(&LeftEdge)) {
            break;
         }
         if (!StepEdge(&RightEdge)) {
            break;
         }
         DestY++;
      }
   }
   /* Steps an edge one scan line in the destination, and the corresponding
      distance in the source. If an edge runs out, starts a new edge if there
      is one. Returns 1 for success, or 0 if there are no more edges to scan. */
   int StepEdge(EdgeScan * Edge)
   {
      /* Count off the scan line we stepped last time; if this edge is
         finished, try to start another one */
      if (--Edge->RemainingScans == 0) {
         /* Set up the next edge; done if there is no next edge */
         if (SetUpEdge(Edge, Edge->CurrentEnd) == 0) {
            return(0);  /* no more edges; done drawing polygon */
         }
         return(1);     /* all set to draw the new edge */
      }
      /* Step the current source edge */
      Edge->SourceX += Edge->SourceStepX;
      Edge->SourceY += Edge->SourceStepY;
      /* Step dest X with Bresenham-style variables, to get precise dest pixel
         placement and avoid gaps */
      Edge->DestX += Edge->DestXIntStep;  /* whole pixel step */
      /* Do error term stuff for fractional pixel X step handling */
      if ((Edge->DestXErrTerm += Edge->DestXAdjUp) > 0) {
         Edge->DestX += Edge->DestXDirection;
         Edge->DestXErrTerm -= Edge->DestXAdjDown;
      }
      return(1);
   }
```

```
/* Sets up an edge to be scanned; the edge starts at StartVert and proceeds
   in direction Edge->Direction through the vertex list. Edge->Direction must
   be set prior to call; -1 to scan a left edge (backward through the vertex
   list), 1 to scan a right edge (forward through the vertex list).
   Automatically skips over 0-height edges. Returns 1 for success, or 0 if
   there are no more edges to scan. */
int SetUpEdge(EdgeScan * Edge, int StartVert)
{
   int NextVert, DestXWidth;
   Fixedpoint DestYHeight;
   for (;;) {
      /* Done if this edge starts at the bottom vertex */
      if (StartVert == MaxVert) {
         return(0);
      }
      /* Advance to the next vertex, wrapping if we run off the start or end
         of the vertex list */
      NextVert = StartVert + Edge->Direction;
      if (NextVert >= NumVerts) {
         NextVert = 0;
      } else if (NextVert < 0) {
         NextVert = NumVerts - 1;
      }
      /* Calculate the variables for this edge and done if this is not a
         zero-height edge */
      if ((Edge->RemainingScans =
            VertexPtr[NextVert].Y - VertexPtr[StartVert].Y) != 0) {
         DestYHeight = INT_TO_FIXED(Edge->RemainingScans);
         Edge->CurrentEnd = NextVert;
         Edge->SourceX = INT_TO_FIXED(TexVertsPtr[StartVert].X);
         Edge->SourceY = INT_TO_FIXED(TexVertsPtr[StartVert].Y);
         Edge->SourceStepX = FixedDiv(INT_TO_FIXED(TexVertsPtr[NextVert].X) -
               Edge->SourceX, DestYHeight);
         Edge->SourceStepY = FixedDiv(INT_TO_FIXED(TexVertsPtr[NextVert].Y) -
               Edge->SourceY, DestYHeight);
         /* Set up Bresenham-style variables for dest X stepping */
         Edge->DestX = VertexPtr[StartVert].X;
         if ((DestXWidth =
               (VertexPtr[NextVert].X - VertexPtr[StartVert].X)) < 0) {
            /* Set up for drawing right to left */
            Edge->DestXDirection = -1;
            DestXWidth = -DestXWidth;
            Edge->DestXErrTerm = 1 - Edge->RemainingScans;
            Edge->DestXIntStep = -(DestXWidth / Edge->RemainingScans);
         } else {
            /* Set up for drawing left to right */
            Edge->DestXDirection = 1;
            Edge->DestXErrTerm = 0;
            Edge->DestXIntStep = DestXWidth / Edge->RemainingScans;
         }
         Edge->DestXAdjUp = DestXWidth % Edge->RemainingScans;
         Edge->DestXAdjDown = Edge->RemainingScans;
         return(1);  /* success */
      }
      StartVert = NextVert;   /* keep looking for a non-0-height edge */
   }
}
/* Texture-map-draw the scan line between two edges. */
void ScanOutLine(EdgeScan * LeftEdge, EdgeScan * RightEdge)
{
   Fixedpoint SourceX = LeftEdge->SourceX;
   Fixedpoint SourceY = LeftEdge->SourceY;
   int DestX = LeftEdge->DestX;
   int DestXMax = RightEdge->DestX;
   Fixedpoint DestWidth;
   Fixedpoint SourceXStep, SourceYStep;
   /* Nothing to do if fully X clipped */
   if ((DestXMax <= ClipMinX) || (DestX >= ClipMaxX)) {
      return;
   }
   if ((DestXMax - DestX) <= 0) {
      return;  /* nothing to draw */
   }
```

```
   /* Width of destination scan line, for scaling. Note: because this is an
      integer-based scaling, it can have a total error of as much as nearly
      one pixel. For more precise scaling, also maintain a fixed-point DestX
      in each edge, and use it for scaling. If this is done, it will also
      be necessary to nudge the source start coordinates to the right by an
      amount corresponding to the distance from the the real (fixed-point)
      DestX and the first pixel (at an integer X) to be drawn) */
   DestWidth = INT_TO_FIXED(DestXMax - DestX);
   /* Calculate source steps that correspond to each dest X step (across
      the scan line) */
   SourceXStep = FixedDiv(RightEdge->SourceX - SourceX, DestWidth);
   SourceYStep = FixedDiv(RightEdge->SourceY - SourceY, DestWidth);
   /* Clip right edge if necessary */
   if (DestXMax > ClipMaxX) {
      DestXMax = ClipMaxX;
   }
   /* Clip left edge if necssary */
   if (DestX < ClipMinX) {
      SourceX += SourceXStep * (ClipMinX - DestX);
      SourceY += SourceYStep * (ClipMinX - DestX);
      DestX = ClipMinX;
   }
   /* Scan across the destination scan line, updating the source image
      position accordingly */
   for (; DestX<DestXMax; DestX++) {
      /* Get currently mapped pixel out of image and draw it to screen */
      WritePixelX(DestX, DestY,
            GET_IMAGE_PIXEL(TexMapBits, TexMapWidth,
            FIXED_TO_INT(SourceX), FIXED_TO_INT(SourceY)) );
      /* Point to the next source pixel */
      SourceX += SourceXStep;
      SourceY += SourceYStep;
   }
}
```

Journal: Dr. Dobb's Journal Oct 1992 v17 n10 p151(7)

How to shear a sheep, and other texture-mapping niceties

Author: Abrash, Michael.
AttFile: GP-OCT92.ASC Source code listing.
XSHARP21.ZIP X-Sharp library for 3D graphics.

Abstract: Experience is important in graphics programming because performance is so important, and performance programming is largely based on experience. A real-time three-dimensional animation program is developed; animation depends on adequate speed to be convincing, but too much speed can destroy the illusion. The shifting of images must be tuned so that the images are in the 'sweet spot' of apparent motion, in which the eye ignores the jumping and aliasing. The importance of experience in programming is shown in the refinements made over a period of time to the X-Sharp 3-D animation package, to which texture mapping was added. Enhancements to the texture-mapping code presented in a previous column are examined, and the code is listed; how to get X-Sharp is described.

I recently spent an hour or so learning how to shear a sheep. Among other things, I learned--in great detail-about the importance of selecting the proper comb for your shears, heard about the man who holds the world's record for sheep sheared in a day (more than 600, if memory serves), and discovered, Lord help me, the many and varied ways in which the New Zealand Sheep Shearing Board improves the approved sheep-shearing method every year. The fellow giving the presentation did his best, but let's face it, sheep just aren't very interesting. If you have children, you'll know why I was there; if you don't, there's no use explaining.

The chap doing the shearing did say one thing that stuck with me, although it may not sound particularly profound. (Actually, it sounds pretty silly, but bear with me.) He said, "You don't get really good at sheep shearing for ten years, or 10,000 sheep." I'll buy that. In fact, to extend that morsel of wisdom to the greater, non-ovine-centric universe, it actually takes a good chunk of experience before you get good at anything worthwhile--especially graphics, for a couple of reasons. First, performance matters a lot in graphics, and performance programming is largely a matter of experience. You can't speed up PC graphics simply by looking in a book for a better algorithm; you have to understand the code C compilers generate, assembly language optimization, VGA hardware, and the performance implications of various graphics-programming approaches and algorithms. Second, computer graphics is a matter of illusion, of convincing the eye to see what you want it to see, and that's very much a black art based on experience.

This month, experience figures into our current subject, real-time 3-D animation, in several ways. Stay tuned.

Visual Quality: A Black Hole...Er, Art

Pleasing the eye with real-time computer animation is something less than a science, at least at the PC level, where there's no time for antialiasing and a limited color palette; in fact, sometimes it can be more than a little frustrating. For example, last month I implemented texture mapping in X-Sharp, the 3-D animation package that's an on-going project in this column. My first implementation was disappointing; the texture maps shimmied and sheared badly, like a loosely affiliated flock of pixels, each marching to its own drummer. Then I added a control key to speed up the routine; what a difference! The aliasing problems were still there, but with the faster rotation, the pixels moved too quickly for the eye to pick up on the aliasing; the rotating texture maps, and the rotating ball as a whole, crossed the threshold into being accepted by the eye as a viewed object, rather than a collection of pixels.

The obvious lesson here is that adequate speed is important to convincing animation. There's another, less obvious side to this lesson though. I'd been running the texture-mapping demo on a 20-MHz 386 with a slow VGA when I

discovered the beneficial effects of greater speed. When, some time later, I ran the demo on a 33-MHz 486 with a fast VGA, I found that the faster rotation was too fast! The ball spun so rapidly that the eye couldn't blend successive images together into continuous motion, much like watching a badly flickering movie.

So the second lesson is that either too little or too much speed can destroy the illusion. Unless you're antialiasing, you need to tune the shifting of your images so that they're in the "sweet spot" of apparent motion, in which the eye is willing to ignore the jumping and aliasing, and blend the images together into continuous motion. Only experience can give you a feel for that sweet spot.

Fixed-point Arithmetic, Redux

Last month, I added texture mapping to X-Sharp, but lacked space to explain some of the liner points. This month, I'll cover some of those points, and discuss the visual and performance enhancements I've added since last month.

In the very lust installment of this column, I spent a good bit of time explaining exactly which pixels were inside a polygon and which were outside, and how to draw those pixels accordingly. This was important, I said, because only with a precise, consistent way of defining inside and outside would it be possible to draw adjacent polygons without either overlap or gaps between them.

As a corollary, I added that only an all-integer, edge-stepping approach would do for polygon filling. Fixed-point arithmetic, although alluring for speed and ease of use, would be unacceptable because round-off error would resuit in imprecise pixel placement.

More than a year then passed, during which time my long-term memory apparently suffered at least partial failure. When I went to implement texture mapping last month, I decided that since transformed destination vertices can fall at fractional pixel locations, the cleanest way to do the texture mapping would be to use fixed-point coordinates for both the source texture and the destination screen polygon. That way, there would a minimum of distortion as the polygon rotated and moved. Theoretically, that made sense; but there was one small problem: gaps between polygons.

Yes, folks, I had ignored the voice of experience (my own voice, at that.) at my own peril. You can be assured I will not forget this particular lesson again: Fixed-point arithmetic is notprecise. That's not to say that it's impossible to use fixed-point for drawing polygons; if all adjacent edges share common start and end vertices and common edges are always stepped in the same direction, then all polygons should share the same fixed-point imprecision, and edges should fit properly (although polygons may not include exactly the right pixels). What you absolutely cannot do is mix fixedpoint and all-integer polygon-filling approaches when drawing, as shown in Figure 1. Consequently, I ended up using an all-integer approach in X-Sharp for stepping through the destination polygon. However, I kept the fixedpoint approach, which is faster and much simpler, for stepping through the source. Why was it all right to mix approaches in this case? Precise pixel placement only matters when drawing, because otherwise we can get gaps, which are very visible. When selecting a plxel to copy from the source texture, however, the worst that happens is that we pick the source pixel next to the one we really want, causing the mapped texture to appear to have shifted by one pixel at the corresponding destination pixel; given all the aliasing and shearing already going on in the texture-mapping process, a one-pixel mapping error is insignificant.

Experience again: knowing which flaws (like small texture shifts) can reasonably be ignored, and which (like those that produce gaps between polygons) must be avoided at all costs.

Texture Mapping: Orientation Independence

Last month's double-DDA texture-mapping code worked adequately, but there were two things about it that left me less than satisfied. One flaw was performance; that's addressed shortly. The other flaw was the way textures

shifted noticeably as the orientations of the polygons they were mapped onto changed..

Last month's code followed the standard polygon inside/outside rule for determining which pixels in the source texture map were to be mapped: Pixels that mapped exactly to the left and top destination edges were considered to be inside, and pixels that mapped exactly to the right and bottom destination edges were considered to be outside. That's me for filling polygons, but when copying texture maps, it causes different edges of the texture map to be omitted, depending on the destination orientation, because different edges of the texture map correspond to the right and bottom destination edges, depending on the current rotation. Also, last month's code truncated to get integer source coordinates. This, together with the orientation problem, meant that when a texture turned upside down, it gained one extra row and one extra column of pixels from the next row and column of the texture map. This asymmetry was quite visible, and not at all the desired effect.

Listing One (page 164) is one solution to these problems. This code, which replaces the equivalently named function from last month, makes no attempt to follow the standard polygon inside/outside rules when mapping the source. Instead, it advances a half-step into the texture map before drawing the first pixel, so pixels along all edges are half included. Rounding rather than truncation to texture-map coordinates is also performed. The result is that the texture map stays pretty much centered within the destination polygon as the destination rotates, with a much-reduced level of orientation-dependent asymmetry.

Mapping Textures Across Multiple Poly One of the truly nifty things about double-DDA texture mapping is that it is not limited to mapping a texture onto a single polygon. A single texture can be mapped across any number of adjacent polygons simply by having polygons that share vertices in 3-space also share vertices in the texture map. In fact, the demonstration program DEMO 1 in the X-Sharp archive maps a single texture across two polygons; this is the blue-on-green pattern that stretches across two panels of the spinning ball. This capability makes it easy to produce polygon-based objects with complex surfaces (such as banding and insignia on a spaceship). Just map the desired texture onto the underlying polygonal framework of an object, and let doubleDDA texture mapping do the rest.

Fast Texture Mapping

Of course, there's a problem with mapping a texture across many polygons: Texture mapping is slow. If you run DEMO1 and move the ball up close to the screen, you'll see that the ball slows considerably whenever a texture swings around into view. To some extent that can't be helped, because each pixel of a texture-mapped polygon has to be calculated and drawn independently. Nonetheless, we can certainly improve the performance of texture mapping a good deal over last month.

By and large, there are two keys to improving PC graphics performance. The first--no surprise--is assembly language. The second, without which assembly language is far less effective, is understanding exactly where the cycles go in inner loops; in our case, that means understanding where the bottlenecks are in Listing One.

Listing Two (page 164) is a high-performance assembly language implementation of Listing One. Apart from the conversion to assembly language, this implementation improves performance by focusing on reducing innerloop bottlenecks. In fact, the whole of Listing Two is nothing more than the inner loop for texture-mapped polygon drawing; Listing Two is only the code to draw a single scan line. Most of the work in drawing a texture-mapped polygon comes in scanning out individual lines, though, so this is the appropriate place to optimize.

Within Listing Two, all the important optimization is in the loop that draws across each destination scan line, near the end of the listing. One optimization is elimination of the call to the set-pixel routine used to draw each pixel in Listing One. Function calls are expensive operations, to be avoided when performance matters. Also, although mode X (the undocumented 320x240 256-co1or VGA mode X-Sharp runs in) doesn't lend itself well to pixel-oriented operations like line drawing or texture mapping, the inner loop has been set up to minimize mode X's overhead. A rotating plane mask is

maintained in AL, with DX pointing to the Map Mask register; thus, only a rotate and an OUT are required to select the plane to which to write, cycling from plane 0 through plane 3 and wrapping back to O. Better yet, because we know that we're simply stepping horizontally across the destination scan line, we can use a clever optimization to both step the destination and reduce the overhead of maintaining the mask. Two copies of the current plane mask are maintained, one in each nibble of AL. (The Map Mask register pays attention only to the lower nibble.) Then, when one copy rotates out of the lower nibble, the other copy rotates into the lower nibble and is ready to be used. This approach eliminates the need to test for the mask wrapping from plane 3 to plane O, all the more so because a carry is generated when wrapping occurs, and that carry can be added to DI to advance the screen pointer.

In all, the overhead of drawing each pixel is reduced from a call to the setpixel routine and full calculation of the screen address and plane mask to five instructions and no branches. This is an excellent example of converting full, from-scratch calculations to incremental processing, whereby only information that has changed since the last operation (the plane mask moving one pixel, for example) is recalculated.

Incremental processing and knowing where the cycles go are both important in the final optimization in Listing Two, speeding up the retrieval of pixels from the texture map. This operation looks very efficient in Listing One, consisting of only two adds and the macro GET_ IMAGE_PIXEL. However, those adds are fixed-point adds, so they take four instructions apiece, and the macro hides not only conversion from fixed-point to integer, but also a time-consuming multiplication. Incremental approaches are excellent at avoiding multiplication, because cumulative additions can often replace multiplication. That's the case with stepping through the source texture in Listing Two; ten instructions, with a maximum of two branches, replace all the texture calculations of Listing One. Listing Two simply detects when the fractional part of the source x or y coordinate turns over and advances the source texture pointer accordingly.

As you might expect, all this optimization is pretty hard to implement, and makes Listing Two much more complicated than Listing One. Is it worth the trouble? Indeed it is. Listing Two is more than twice as fast as Listing One, and the difference is very noticeable when large, texture-mapped areas are animated. Whether more than doubling performance is significant is a matter of opinion, I suppose, but imagine that you're in William Gibson's Neuromancer, trying to crack a corporate database. Which texture-mapping routine would you rather have interfacing you to Cyberspace?

Where to Get X-Sharp

The full source for X-Sharp is available in the file XSHRPn. ZIP in the DDJ Forum on CompuServe, and as XSHARPn .ZIP in the programming/graphics conference on M&T Online and the graphic.disp conference on Bix. (XSHARP21 is the first version that includes fast, assembly language texture mapping.) Alternatively, you can send me a 360K or 720K formatted diskette and an addressed, stamped diskette mailer, care of DDJ, 411 Borel Ave., San Mateo, CA 94402; and I'll send you the latest copy of X-Sharp. There's no charge, but it'd be very much appreciated if you'd slip in a dollar or so to help out the folks at the Vermont Association for the Blind and Visually Impaired.

I'm available on a daily basis to discuss X-Sharp on M&T Online and Bix; my user name is mabrash in both cases. There is no truth to the rumor that I can be reached under the alias "sheep-shearer," at least not for another 9999 sheep.

- - - -

```
void ScanOutLine(EdgeScan * LeftEdge, EdgeScan * RightEdge)
{
   Fixedpoint SourceX;
   Fixedpoint SourceY;
   int DestX = LeftEdge->DestX;
   int DestXMax = RightEdge->DestX;
   Fixedpoint DestWidth;
   Fixedpoint SourceStepX, SourceStepY;

   /* Nothing to do if fully X clipped */
   if ((DestXMax <= ClipMinX) || (DestX >= ClipMaxX)) {
      return;
   }

   if ((DestXMax - DestX) <= 0) {
      return;  /* nothing to draw */
   }
   SourceX = LeftEdge->SourceX;
   SourceY = LeftEdge->SourceY;

   /* Width of destination scan line, for scaling. Note: because this is an
      integer-based scaling, it can have a total error of as much as nearly
      one pixel. For more precise scaling, also maintain a fixed-point DestX
      in each edge, and use it for scaling. If this is done, it will also
      be necessary to nudge the source start coordinates to the right by an
      amount corresponding to the distance from the the real (fixed-point)
      DestX and the first pixel (at an integer X) to be drawn) */
   DestWidth = INT_TO_FIXED(DestXMax - DestX);

   /* Calculate source steps that correspond to each dest X step (across
      the scan line) */
   SourceStepX = FixedDiv(RightEdge->SourceX - SourceX, DestWidth);
   SourceStepY = FixedDiv(RightEdge->SourceY - SourceY, DestWidth);

   /* Advance 1/2 step in the stepping direction, to space scanned pixels
      evenly between the left and right edges. (There's a slight inaccuracy
      in dividing negative numbers by 2 by shifting rather than dividing,
      but the inaccuracy is in the least significant bit, and we'll just
      live with it.) */
   SourceX += SourceStepX >> 1;
   SourceY += SourceStepY >> 1;

   /* Clip right edge if necssary */
   if (DestXMax > ClipMaxX)
      DestXMax = ClipMaxX;

   /* Clip left edge if necssary */
   if (DestX < ClipMinX) {
      SourceX += FixedMul(SourceStepX, INT_TO_FIXED(ClipMinX - DestX));
      SourceY += FixedMul(SourceStepY, INT_TO_FIXED(ClipMinX - DestX));
      DestX = ClipMinX;
   }
   /* Scan across the destination scan line, updating the source image
      position accordingly */
   for (; DestX<DestXMax; DestX++) {
      /* Get the currently mapped pixel out of the image and draw it to
         the screen */
      WritePixelX(DestX, DestY,
            GET_IMAGE_PIXEL(TexMapBits, TexMapWidth,
            ROUND_FIXED_TO_INT(SourceX), ROUND_FIXED_TO_INT(SourceY)) );
      /* Point to the next source pixel */
      SourceX += SourceStepX;
      SourceY += SourceStepY;
   }
}
```

[LISTING TWO]

```
; Draws all pixels in the specified scan line, with the pixel colors
; taken from the specified texture map.  Uses approach of pre-stepping
; 1/2 pixel into the source image and rounding to the nearest source
```

```
; pixel at each step, so that texture maps will appear reasonably similar
; at all angles.  This routine is specific to 320-pixel-wide planar
; (non-Chain4) 256-color modes, such as mode X, which is a planar
; (non-chain4) 256-color mode with a resolution of 320x240.
; C near-callable as:
;   void ScanOutLine(EdgeScan * LeftEdge, EdgeScan * RightEdge);
; Tested with TASM 3.0.

SC_INDEX equ    03c4h   ;Sequence Controller Index
MAP_MASK equ    02h     ;index in SC of Map Mask register
SCREEN_SEG equ  0a000h  ;segment of display memory in mode X
SCREEN_WIDTH equ 80     ;width of screen in bytes from one scan line
                        ; to the next

        .model  small
        .data
        extrn   _TexMapBits:word, _TexMapWidth:word, _DestY:word
        extrn   _CurrentPageBase:word, _ClipMinX:word
        extrn   _ClipMinY:word, _ClipMaxX:word, _ClipMaxY:word

; Describes the current location and stepping, in both the source and
; the destination, of an edge. Mirrors structure in DRAWTEXP.C.
EdgeScan struc
Direction dw    ?       ;through edge list; 1 for a right edge (forward
                        ; through vertex list), -1 for a left edge (backward
                        ; through vertex list)
RemainingScans dw ?     ;height left to scan out in dest
CurrentEnd dw   ?       ;vertex # of end of current edge
SourceX dd      ?       ;X location in source for this edge
SourceY dd      ?       ;Y location in source for this edge
SourceStepX dd  ?       ;X step in source for Y step in dest of 1
SourceStepY dd  ?       ;Y step in source for Y step in dest of 1
                        ;variables used for all-integer Bresenham's-type
                        ; X stepping through the dest, needed for precise
                        ; pixel placement to avoid gaps
DestX   dw      ?       ;current X location in dest for this edge
DestXIntStep dw ?       ;whole part of dest X step per scan-line Y step
DestXDirection dw ?     ;-1 or 1 to indicate which way X steps (left/right)
DestXErrTerm dw ?       ;current error term for dest X stepping
DestXAdjUp dw   ?       ;amount to add to error term per scan line move
DestXAdjDown dw ?       ;amount to subtract from error term when the
                        ; error term turns over
EdgeScan ends

Parms   struc
          dw      2 dup(?) ;return address & pushed BP
LeftEdge  dw      ?        ;pointer to EdgeScan structure for left edge
RightEdge dw      ?        ;pointer to EdgeScan structure for right edge
Parms   ends

;Offsets from BP in stack frame of local variables.
lSourceX        equ     -4      ;current X coordinate in source image
lSourceY        equ     -8      ;current Y coordinate in source image
lSourceStepX    equ     -12     ;X step in source image for X dest step of 1
lSourceStepY    equ     -16     ;Y step in source image for X dest step of 1
lXAdvanceByOne  equ     -18     ;used to step source pointer 1 pixel
                                ; incrementally in X
lXBaseAdvance   equ     -20     ;use to step source pointer minimum number of
                                ; pixels incrementally in X
lYAdvanceByOne  equ     -22     ;used to step source pointer 1 pixel
                                ; incrementally in Y
lYBaseAdvance   equ     -24     ;use to step source pointer minimum number of
                                ; pixels incrementally in Y
LOCAL_SIZE      equ     24      ;total size of local variables
        .code
        extrn   _FixedMul:near, _FixedDiv:near
        align   2
ToScanDone:
        jmp     ScanDone
        public  _ScanOutLine
        align   2
_ScanOutLine    proc    near
        push    bp              ;preserve caller's stack frame
        mov     bp,sp           ;point to our stack frame
```

```
        sub     sp,LOCAL_SIZE   ;allocate space for local variables
        push    si              ;preserve caller's register variables
        push    di
; Nothing to do if destination is fully X clipped.
        mov     di,[bp].RightEdge
        mov     si,[di].DestX
        cmp     si,[_ClipMinX]
        jle     ToScanDone      ;right edge is to left of clip rect, so done
        mov     bx,[bp].LeftEdge
        mov     dx,[bx].DestX
        cmp     dx,[_ClipMaxX]
        jge     ToScanDone      ;left edge is to right of clip rect, so done
        sub     si,dx           ;destination fill width
        jle     ToScanDone      ;null or negative full width, so done

        mov     ax,word ptr [bx].SourceX        ;initial source X coordinate
        mov     word ptr [bp].lSourceX,ax
        mov     ax,word ptr [bx].SourceX+2
        mov     word ptr [bp].lSourceX+2,ax

        mov     ax,word ptr [bx].SourceY        ;initial source Y coordinate
        mov     word ptr [bp].lSourceY,ax
        mov     ax,word ptr [bx].SourceY+2
        mov     word ptr [bp].lSourceY+2,ax
; Calculate source steps that correspond to each 1-pixel destination X step
; (across the destination scan line).
        push    si              ;push dest X width, in fixedpoint form
        sub     ax,ax
        push    ax              ;push 0 as fractional part of dest X width
        mov     ax,word ptr [di].SourceX
        sub     ax,word ptr [bp].lSourceX       ;low word of source X width
        mov     dx,word ptr [di].SourceX+2
        sbb     dx,word ptr [bp].lSourceX+2     ;high word of source X width
        push    dx              ;push source X width, in fixedpoint form
        push    ax
        call    _FixedDiv       ;scale source X width to dest X width
        add     sp,8            ;clear parameters from stack
        mov     word ptr [bp].lSourceStepX,ax   ;remember source X step for
        mov     word ptr [bp].lSourceStepX+2,dx ; 1-pixel destination X step
        mov     cx,1            ;assume source X advances non-negative
        and     dx,dx           ;which way does source X advance?
        jns     SourceXNonNeg   ;non-negative
        neg     cx              ;negative
        cmp     ax,0            ;is the whole step exactly an integer?
        jz      SourceXNonNeg   ;yes
        inc     dx              ;no, truncate to integer in the direction of
                                ; 0, because otherwise we'll end up with a
                                ; whole step of 1-too-large magnitude
SourceXNonNeg:
        mov     [bp].lXAdvanceByOne,cx  ;amount to add to source pointer to
                                        ; move by one in X
        mov     [bp].lXBaseAdvance,dx   ;minimum amount to add to source
                                        ; pointer to advance in X each time
                                        ; the dest advances one in X
        push    si              ;push dest Y height, in fixedpoint form
        sub     ax,ax
        push    ax              ;push 0 as fractional part of dest Y height
        mov     ax,word ptr [di].SourceY
        sub     ax,word ptr [bp].lSourceY       ;low word of source Y height
        mov     dx,word ptr [di].SourceY+2
        sbb     dx,word ptr [bp].lSourceY+2     ;high word of source Y height
        push    dx              ;push source Y height, in fixedpoint form
        push    ax
        call    _FixedDiv       ;scale source Y height to dest X width
        add     sp,8            ;clear parameters from stack
        mov     word ptr [bp].lSourceStepY,ax   ;remember source Y step for
        mov     word ptr [bp].lSourceStepY+2,dx ; 1-pixel destination X step
        mov     cx,[_TexMapWidth] ;assume source Y advances non-negative
        and     dx,dx           ;which way does source Y advance?
        jns     SourceYNonNeg   ;non-negative
        neg     cx              ;negative
        cmp     ax,0            ;is the whole step exactly an integer?
        jz      SourceYNonNeg   ;yes
        inc     dx              ;no, truncate to integer in the direction of
```

```
                                ; 0, because otherwise we'll end up with a
                                ; whole step of 1-too-large magnitude
SourceYNonNeg:
        mov     [bp].lYAdvanceByOne,cx  ;amount to add to source pointer to
                                        ; move by one in Y
        mov     ax,[_TexMapWidth]       ;minimum distance skipped in source
        imul    dx                      ; image bitmap when Y steps (ignoring
        mov     [bp].lYBaseAdvance,ax   ; carry from the fractional part)
; Advance 1/2 step in the stepping direction, to space scanned pixels evenly
; between the left and right edges. (There's a slight inaccuracy in dividing
; negative numbers by 2 by shifting rather than dividing, but the inaccuracy
; is in the least significant bit, and we'll just live with it.)
        mov     ax,word ptr [bp].lSourceStepX
        mov     dx,word ptr [bp].lSourceStepX+2
        sar     dx,1
        rcr     ax,1
        add     word ptr [bp].lSourceX,ax
        adc     word ptr [bp].lSourceX+2,dx

        mov     ax,word ptr [bp].lSourceStepY
        mov     dx,word ptr [bp].lSourceStepY+2
        sar     dx,1
        rcr     ax,1
        add     word ptr [bp].lSourceY,ax
        adc     word ptr [bp].lSourceY+2,dx
; Clip right edge if necessary.
        mov     si,[di].DestX
        cmp     si,[_ClipMaxX]
        jl      RightEdgeClipped
        mov     si,[_ClipMaxX]
RightEdgeClipped:
; Clip left edge if necssary
        mov     bx,[bp].LeftEdge
        mov     di,[bx].DestX
        cmp     di,[_ClipMinX]
        jge     LeftEdgeClipped
; Left clipping is necessary; advance the source accordingly
        neg     di
        add     di,[_ClipMinX]  ;ClipMinX - DestX
                                ;first, advance the source in X
        push    di              ;push ClipMinX - DestX, in fixedpoint form
        sub     ax,ax
        push    ax              ;push 0 as fractional part of ClipMinX-DestX
        push    word ptr [bp].lSourceStepX+2
        push    word ptr [bp].lSourceStepX
        call    _FixedMul       ;total source X stepping in clipped area
        add     sp,8            ;clear parameters from stack
        add     word ptr [bp].lSourceX,ax   ;step the source X past clipping
        adc     word ptr [bp].lSourceX+2,dx
                                ;now advance the source in Y
        push    di              ;push ClipMinX - DestX, in fixedpoint form
        sub     ax,ax
        push    ax              ;push 0 as fractional part of ClipMinX-DestX
        push    word ptr [bp].lSourceStepY+2
        push    word ptr [bp].lSourceStepY
        call    _FixedMul       ;total source Y stepping in clipped area
        add     sp,8            ;clear parameters from stack
        add     word ptr [bp].lSourceY,ax   ;step the source Y past clipping
        adc     word ptr [bp].lSourceY+2,dx
        mov     di,[_ClipMinX]  ;start X coordinate in dest after clipping
LeftEdgeClipped:
; Calculate actual clipped destination drawing width.
        sub     si,di
; Scan across the destination scan line, updating the source image position
; accordingly.
; Point to the initial source image pixel, adding 0.5 to both X and Y so that
; we can truncate to integers from now on but effectively get rounding.
        add     word ptr [bp].lSourceY,8000h    ;add 0.5
        mov     ax,word ptr [bp].lSourceY+2
        adc     ax,0
        mul     [_TexMapWidth]   ;initial scan line in source image
        add     word ptr [bp].lSourceX,8000h    ;add 0.5
        mov     bx,word ptr [bp].lSourceX+2 ;offset into source scan line
        adc     bx,ax            ;initial source offset in source image
```

```
        add     bx,[_TexMapBits] ;DS:BX points to the initial image pixel
; Point to initial destination pixel.
        mov     ax,SCREEN_SEG
        mov     es,ax
        mov     ax,SCREEN_WIDTH
        mul     [_DestY] ;offset of initial dest scan line
        mov     cx,di   ;initial destination X
        shr     di,1
        shr     di,1    ;X/4 = offset of pixel in scan line
        add     di,ax   ;offset of pixel in page
        add     di,[_CurrentPageBase] ;offset of pixel in display memory
                        ;ES:DI now points to the first destination pixel

        and     cl,011b ;CL = pixel's plane
        mov     al,MAP_MASK
        mov     dx,SC_INDEX
        out     dx,al   ;point the SC Index register to the Map Mask
        mov     al,11h  ;one plane bit in each nibble, so we'll get carry
                        ; automatically when going from plane 3 to plane 0
        shl     al,cl   ;set the bit for the first pixel's plane to 1
; If source X step is negative, change over to working with non-negative
; values.
        cmp     word ptr [bp].lXAdvanceByOne,0
        jge     SXStepSet
        neg     word ptr [bp].lSourceStepX
        not     word ptr [bp].lSourceX
SXStepSet:
; If source Y step is negative, change over to working with non-negative
; values.
        cmp     word ptr [bp].lYAdvanceByOne,0
        jge     SYStepSet
        neg     word ptr [bp].lSourceStepY
        not     word ptr [bp].lSourceY
SYStepSet:
; At this point:
;       AL = initial pixel's plane mask
;       BX = pointer to initial image pixel
;       SI = # of pixels to fill
;       DI = pointer to initial destination pixel
        mov     dx,SC_INDEX+1   ;point to SC Data; Index points to Map Mask
TexScanLoop:
; Set the Map Mask for this pixel's plane, then draw the pixel.
        out     dx,al
        mov     ah,[bx]         ;get image pixel
        mov     es:[di],ah      ;set image pixel
; Point to the next source pixel.
        add     bx,[bp].lXBaseAdvance   ;advance the minimum # of pixels in X
        mov     cx,word ptr [bp].lSourceStepX
        add     word ptr [bp].lSourceX,cx ;step the source X fractional part
        jnc     NoExtraXAdvance           ;didn't turn over; no extra advance
        add     bx,[bp].lXAdvanceByOne    ;did turn over; advance X one extra
NoExtraXAdvance:
        add     bx,[bp].lYBaseAdvance   ;advance the minimum # of pixels in Y
        mov     cx,word ptr [bp].lSourceStepY
        add     word ptr [bp].lSourceY,cx ;step the source Y fractional part
        jnc     NoExtraYAdvance           ;didn't turn over; no extra advance
        add     bx,[bp].lYAdvanceByOne    ;did turn over; advance Y one extra
NoExtraYAdvance:
; Point to the next destination pixel, by cycling to the next plane, and
; advancing to the next address if the plane wraps from 3 to 0.
        rol     al,1
        adc     di,0
; Continue if there are any more dest pixels to draw.
        dec     si
        jnz     TexScanLoop
ScanDone:
        pop     di              ;restore caller's register variables
        pop     si
        mov     sp,bp           ;deallocate local variables
        pop     bp              ;restore caller's stack frame
        ret
_ScanOutLine    endp
        end
```

Journal: Dr. Dobb's Journal Nov 1992 v17 n11 p171(6)

The good, the bad, and the run-sliced

Author: Abrash, Michael
Attached: GP-NOV92.ASC Source code listing.
XSHARP21.ZIP Library for 3-D animation.

Abstract: Programmers developing graphics applications should never believe that they have developed the fastest possible code; others often have better ideas. Performance-intensive applications should never have code that performs the same calculation more than once. Programmers have a duty to implement the desired functions while enabling the hardware to work as efficiently as possible. Bresenham's run-length slice algorithm is a good model for efficient programming. The approach implemented by the algorithm steps one pixel at a time along the major axis, while an integer term indicating how close the line is to advancing halfway to the next major pixel along the minor axis is maintained. The algorithm provides an automatic decision-making structure that allows infrequent decisions to be made as quickly as possible.

Years ago, I worked at a company that asked me to write blazingly fast line-drawing code for an AutoCAD driver. I implemented the basic Bresenham's line-drawing algorithm; streamlined it as much as possible; special-cased horizontal, diagonal, and vertical lines; broke out separate, optimized routines for lines in each octant; and massively unrolled the loops. When I was done, I had line drawing down to a mere five or six instructions per pixel, and I handed the code over to the AutoCAD driver person, content in the knowledge that I had pushed the theoretical limits of the Bresenham's algorithm on the 80x86 architecture, and that this was as fast as line drawing could get on a PC. That feeling lasted for about a week, until Dave Miller, who these days is a Windows display-driver whiz at Engenious Solutions, casually mentioned Bresenham's faster run-length slice line-drawing algorithm.

Remember Bill Murray's safety tip in Ghostbusters? It goes something like this. Harold Ramis tells the Ghostbusters not to cross the beams of the antighost guns. "Why?" Murray asks.

"It would be bad," Ramis says.

Murray says, "I'm fuzzy on the whole good/bad thing. What exactly do you mean by 'bad'?" It turns out that what Ramis means by bad is basically the destruction of the universe.

"Important safety tip," Murray comments dryly.

I learned two important safety tips from my line-drawing experience; neither involves the possible destruction of the universe, so far as I know, but they are nonetheless worth keeping in mind. First, never, never, never think you've written the fastest possible code. Odds are, you haven't. Run your code past another good programmer, and he or she will probably say, "But why don't you do this?" and you'll realize that you could indeed do that, and your code would then be faster. Or relax and come back to your code later, and you may well see another, faster approach. There are a million ways to implement code for any task, and you can almost always find a faster way if you need to.

Second, when performance matters, never have your code perform the same calculation more than once. This sounds obvious, but it's astonishing how often it's ignored. For example, consider the snippet of code in Example 1. Here, the programmer knows which way the line is going before the main loop begins--but nonetheless performs that test every time through the loop, when calculating the address of the next pixel. Far better to perform the test only once, outside the loop, as shown in Example 2.

Think of it this way. A program is a state machine. It takes a set of inputs and produces a corresponding set of outputs by passing through a

set of states. Your primary job as a programmer is to implement the desired state machine. Your additional job as a performance programmer is to minimize the lengths of the paths through the state machine. This means performing as many tests and calculations as possible outside the loops, so that the loops themselves can do as little work--pass through as few states--as possible.

Which brings us full circle to Bresenham's run-length slice line-drawing algorithm, which just happens to be an excellent example of a minimized state machine. In case you're fuzzy on the good/bad performance thing, that's "good"--as in fast.

Run-length Slice Fundamentals

First off, I have a confession to make: I'm not sure that the algorithm I'll discuss is actually, precisely Bresenham's run-length slice algorithm. It's been a long time since I read about this algorithm; in the intervening years, I've misplaced Bresenham's article, and I was unable to locate it in time for this column. (Vermont libraries leave something to be desired in the high-tech area.) As a result, I had to derive the algorithm from scratch, which was admittedly more fun than reading about it, and also ensured that I understood it inside and out. The upshot is that what I discuss may or may not be Bresenham's run-length slice algorithm--but it surely is fast.

The place to begin understanding the run-length slice algorithm is the standard Bresenham's line-drawing algorithm. (I discussed the standard Bresenham's algorithm at length in the May 1989 issue of the now-defunct Programmer's Journal.) The basis of the standard approach is stepping one pixel at a time along the major axis (the longer dimension of the line), while maintaining an integer error term that indicates at each major-axis step how close the line is to advancing halfway to the next pixel along the minor axis. Figure 1 illustrates standard Bresenham's line drawing. The key point here is that a calculation and a test are performed once for each step along the major axis.

The run-length slice algorithm rotates matters 90 degrees, with salubrious results. The basis of the run-length slice algorithm is stepping one pixel at a time along the minor axis (the shorter dimension), while maintaining an integer error term indicating how close the line is to advancing an extra pixel along the major axis, as illustrated by Figure 2.

Consider this: When you're called upon to draw a line with an X-dimension of 35 and a Y-dimension of 10, you have a great deal of information available, some of which is ignored by standard Bresenham's. In particular, because the slope is between 1/3 and 1/4, you know that every single run--a run being a set of pixels at the same minor-axis coordinate--must be either three or four pixels long. No other length is possible, as shown in Figure 3 (apart from the first and last runs, which are special cases that I'll discuss shortly). Therefore, for this line, there's no need to perform an error-term calculation and test for each pixel. Instead, we can just perform one test per run, to see whether the run is three or four pixels long, thereby eliminating about 70 percent of the calculations in drawing this line.

Take a moment to let the idea behind run-length slice drawing soak in. Periodic decisions must be made to control pixel placement. The key to speed is to make those decisions as infrequently and quickly as possible. Of course, it will work to make a decision at each pixel--that's standard Bresenham's. However, most of those per-pixel decisions are redundant, and in fact we have enough information before we begin to know which are the redundant decisions. Run-length slice drawing is exactly equivalent to standard Bresenham's, but it pares the decision-making process down to a minimum. It's some-what analogous to the difference between finding the greatest common divisor of two numbers using Euclid's algorithm and finding it by trying every possible divisor. Both approaches produce the desired result, but that which takes maximum advantage of the available information and minimizes redundant work is preferable.

Run-length Slice Implementation

We know that for any line, a given run will always be one of two possible lengths. How, though, do we know which length to select? Surprisingly, this is easy to determine. For the following discussion, assume that we have a slope of 1/3.5, so that X is the major axis; however, the discussion also applies to Y-major lines, with X and Y reversed.

The minimum possible length for any run in an X-major line is int(XDelta/YDelta), where XDelta is the X-dimension of the line and YDelta is the Y-dimension. The maximum possible length is int(XDelta/YDelta) + 1. The trick, then, is knowing which of these two lengths to select for each run. To see how we can make this selection, refer to Figure 4. For each one-pixel step along the minor axis (Y, in this case), we advance at least three pixels. The full advance distance along X (the major axis) is actually three-plus pixels, because there is also a fractional portion to the advance along X for a single-pixel Y step. This fractional advance is the key to deciding when to add an extra pixel to a run. The fraction indicates what portion of an extra pixel we advance along X (the major axis) during each run. If we keep a running sum of the fractional parts, we have a measure of how close we are to needing an extra pixel; when the fractional sum reaches 1, it's time to add an extra pixel to the current run. Then we can subtract 1 from the running sum (because we just advanced one pixel), and continue on.

Practically speaking, however, we can't work with fractions because floating-point arithmetic is slow and fixed-point arithmetic is imprecise. Therefore, we take a cue from standard Bresenham's and scale all the error-term calculations up so that we can work with integers. The fractional X (major axis) advance per one-pixel Y (minor axis) advance is the fractional portion of XDelta/YDelta. This value is exactly equivalent to (XDelta % YDelta)/YDelta. We'll scale this up by multiplying it by YDelta*2, so that the amount by which we adjust the error term up for each one-pixel minor-axis advance is (XDelta % YDelta)*2.

We'll similarly scale up the one pixel by which we adjust the error term down after it turns over, so our downward error-term adjustment is YDelta*2. Therefore, before drawing each run, we'll add (XDelta % YDelta)*2 to the error term. If the error term turns over (reaches one full pixel), then we'll lengthen the run by 1, and subtract YDelta*2 from the error term. (All values are multiplied by 2 so that the initial error term, which involves a 0.5 term, can be scaled up to an integer, as discussed below.)

This is not a complicated process, involving only integer addition and subtraction and a single test, and it lends itself to many and varied optimizations. For example, you could break out hardwired optimizations for drawing each possible pair of run lengths. For the aforementioned line with a slope of 1/3.5, for example, you could have one routine hardwired to blast in a run of three pixels as quickly as possible, and another hardwired to blast in a run of four pixels. These routines would ideally have no looping, but rather just a series of instructions customized to draw the desired number of pixels at maximum speed. Each routine would know that the only possibilities for the length of the next run would be three and four, so they could increment the error term, then jump directly to the appropriate one of the two routines depending on whether the error term turned over. Properly implemented, it should be possible to reduce the average per-run overhead of line drawing to less than one branch, with only two additions and two tests (the number of runs must also be counted down), plus a subtraction half the time. On a 486, this amounts to something on the order of 150 nanoseconds of overhead per pixel, exclusive of the time required to actually write the pixel to display memory.

That's good.

Run-length Slice Details

A couple of run-length slice implementation details yet remain. First is the matter of how error-term turnover is detected. This is done in much

the same way as it is with standard Bresenham's: The error term is initialized to a value equivalent to -1 pixel and incremented for each step; when the error term reaches 0, we've advanced one full pixel along the major axis, and it's time to add an extra pixel to the current run. This means that we only have to test the sign of the error term after advancing it to determine whether or not to add an extra pixel to each run.

The second and more difficult detail is balancing the runs so that they're centered around the ideal line, and therefore draw the same pixels that standard Bresenham's would draw. If we just drew full-length runs from the start, we'd end up with an unbalanced line, as shown in Figure 5. Instead, we have to split the initial pixel plus one full run as evenly as possible between the first and last runs of the line, and adjust the initial error term appropriately for the initial half-run.

The initial error term is simply one-half of the normal fractional advance along the major axis, because the initial step is only one-half pixel along the minor axis. This half-step gets us exactly halfway between the initial pixel and the next pixel along the minor axis. All the error-term adjusts are scaled up by two times precisely so that we can scale up this halved error term for the initial run by two times, and thereby make it an integer.

The other trick here is that if an odd number of pixels are allocated between the first and last partial runs, we'll end up with an odd pixel, since we are unable to draw a half-pixel. This odd pixel is accounted for by adding half a pixel to the error term.

That's all there is to run-length slice line drawing; the partial first and last runs are the only tricky part. Listing One (page 190) is a run-length slice implementation in C. This is not an optimized implementation, nor is it meant to be; this listing is provided so that you can see how the run-length slice algorithm works. Next month, I'll move on to an optimized version, but this month's listing will make it much easier to grasp the principles of run-length slice drawing, and to understand next month's code.

Notwithstanding that it's not optimized, Listing One is reasonably fast. If you run Listing Two (page 191), a sample line-drawing program that you can use to test-drive Listing One, you may be as surprised as I was at how quickly the screen fills with vectors, considering that Listing One is entirely in C and has some redundant divides. Or perhaps you won't be surprised, in which case I suggest you check back next month.

Next Time

Next month, I'll switch to assembly language and speed up run-length slice lines considerably. I'll also spend some time discussing the limitations of run-length slice drawing, and I'll look at possible further optimizations. After that, perhaps we'll have a look at seed fills, or more 3-D animation, or some new 2-D animation topics--or maybe something completely different. Your suggestions are, as always, welcome.

- - - -

```
void DrawHorizontalRun(char far **ScreenPtr, int XAdvance, int RunLength,
                       int Color);
void DrawVerticalRun(char far **ScreenPtr, int XAdvance, int RunLength,
                       int Color);
/* Draws a line between the specified endpoints in color Color. */
void LineDraw(int XStart, int YStart, int XEnd, int YEnd, int Color)
{
   int Temp, AdjUp, AdjDown, ErrorTerm, XAdvance, XDelta, YDelta;
   int WholeStep, InitialPixelCount, FinalPixelCount, i, RunLength;
   char far *ScreenPtr;

   /* We'll always draw top to bottom, to reduce the number of cases we have to
   handle, and to make lines between the same endpoints draw the same pixels */
   if (YStart > YEnd) {
      Temp = YStart;
      YStart = YEnd;
      YEnd = Temp;
      Temp = XStart;
      XStart = XEnd;
      XEnd = Temp;
   }
   /* Point to the bitmap address first pixel to draw */
   ScreenPtr = MK_FP(SCREEN_SEGMENT, YStart * SCREEN_WIDTH + XStart);

   /* Figure out whether we're going left or right, and how far we're
      going horizontally */
   if ((XDelta = XEnd - XStart) < 0)
   {
      XAdvance = -1;
      XDelta = -XDelta;
   }
   else
   {
      XAdvance = 1;
   }
   /* Figure out how far we're going vertically */
   YDelta = YEnd - YStart;

   /* Special-case horizontal, vertical, and diagonal lines, for speed
      and to avoid nasty boundary conditions and division by 0 */
   if (XDelta == 0)
   {
      /* Vertical line */
      for (i=0; i<=YDelta; i++)
      {
         *ScreenPtr = Color;
         ScreenPtr += SCREEN_WIDTH;
      }
      return;
   }
   if (YDelta == 0)
   {
      /* Horizontal line */
      for (i=0; i<=XDelta; i++)
      {
         *ScreenPtr = Color;
         ScreenPtr += XAdvance;
      }
      return;
   }
   if (XDelta == YDelta)
   {
      /* Diagonal line */
      for (i=0; i<=XDelta; i++)
      {
         *ScreenPtr = Color;
         ScreenPtr += XAdvance + SCREEN_WIDTH;
      }
      return;
   }

   /* Determine whether the line is X or Y major, and handle accordingly */
   if (XDelta >= YDelta)
   {
```

```
   /* X major line */
   /* Minimum # of pixels in a run in this line */
   WholeStep = XDelta / YDelta;

   /* Error term adjust each time Y steps by 1; used to tell when one
      extra pixel should be drawn as part of a run, to account for
      fractional steps along the X axis per 1-pixel steps along Y */
   AdjUp = (XDelta % YDelta) * 2;

   /* Error term adjust when the error term turns over, used to factor
      out the X step made at that time */
   AdjDown = YDelta * 2;

   /* Initial error term; reflects an initial step of 0.5 along the Y
      axis */
   ErrorTerm = (XDelta % YDelta) - (YDelta * 2);

   /* The initial and last runs are partial, because Y advances only 0.5
      for these runs, rather than 1. Divide one full run, plus the
      initial pixel, between the initial and last runs */
   InitialPixelCount = (WholeStep / 2) + 1;
   FinalPixelCount = InitialPixelCount;

   /* If the basic run length is even and there's no fractional
      advance, we have one pixel that could go to either the initial
      or last partial run, which we'll arbitrarily allocate to the
      last run */
   if ((AdjUp == 0) && ((WholeStep & 0x01) == 0))
   {
      InitialPixelCount--;
   }
  /* If there're an odd number of pixels per run, we have 1 pixel that can't
  be allocated to either the initial or last partial run, so we'll add 0.5
  to error term so this pixel will be handled by the normal full-run loop */
   if ((WholeStep & 0x01) != 0)
   {
      ErrorTerm += YDelta;
   }
   /* Draw the first, partial run of pixels */
   DrawHorizontalRun(&ScreenPtr, XAdvance, InitialPixelCount, Color);
   /* Draw all full runs */
   for (i=0; i<(YDelta-1); i++)
   {
      RunLength = WholeStep;  /* run is at least this long */
      /* Advance the error term and add an extra pixel if the error
         term so indicates */
      if ((ErrorTerm += AdjUp) > 0)
      {
         RunLength++;
         ErrorTerm -= AdjDown;   /* reset the error term */
      }
      /* Draw this scan line's run */
      DrawHorizontalRun(&ScreenPtr, XAdvance, RunLength, Color);
   }
   /* Draw the final run of pixels */
   DrawHorizontalRun(&ScreenPtr, XAdvance, FinalPixelCount, Color);
   return;
}
else
{
   /* Y major line */

   /* Minimum # of pixels in a run in this line */
   WholeStep = YDelta / XDelta;

   /* Error term adjust each time X steps by 1; used to tell when 1 extra
      pixel should be drawn as part of a run, to account for
      fractional steps along the Y axis per 1-pixel steps along X */
   AdjUp = (YDelta % XDelta) * 2;

   /* Error term adjust when the error term turns over, used to factor
      out the Y step made at that time */
   AdjDown = XDelta * 2;
```

```
      /* Initial error term; reflects initial step of 0.5 along the X axis */
      ErrorTerm = (YDelta % XDelta) - (XDelta * 2);

      /* The initial and last runs are partial, because X advances only 0.5
         for these runs, rather than 1. Divide one full run, plus the
         initial pixel, between the initial and last runs */
      InitialPixelCount = (WholeStep / 2) + 1;
      FinalPixelCount = InitialPixelCount;

      /* If the basic run length is even and there's no fractional advance, we
         have 1 pixel that could go to either the initial or last partial run,
         which we'll arbitrarily allocate to the last run */
      if ((AdjUp == 0) && ((WholeStep & 0x01) == 0))
      {
         InitialPixelCount--;
      }
      /* If there are an odd number of pixels per run, we have one pixel
         that can't be allocated to either the initial or last partial
         run, so we'll add 0.5 to the error term so this pixel will be
         handled by the normal full-run loop */
      if ((WholeStep & 0x01) != 0)
      {
         ErrorTerm += XDelta;
      }
      /* Draw the first, partial run of pixels */
      DrawVerticalRun(&ScreenPtr, XAdvance, InitialPixelCount, Color);

      /* Draw all full runs */
      for (i=0; i<(XDelta-1); i++)
      {
         RunLength = WholeStep;  /* run is at least this long */
         /* Advance the error term and add an extra pixel if the error
            term so indicates */
         if ((ErrorTerm += AdjUp) > 0)
         {
            RunLength++;
            ErrorTerm -= AdjDown;   /* reset the error term */
         }
         /* Draw this scan line's run */
         DrawVerticalRun(&ScreenPtr, XAdvance, RunLength, Color);
      }
      /* Draw the final run of pixels */
      DrawVerticalRun(&ScreenPtr, XAdvance, FinalPixelCount, Color);
      return;
   }
}
/* Draws a horizontal run of pixels, then advances the bitmap pointer to
   the first pixel of the next run. */
void DrawHorizontalRun(char far **ScreenPtr, int XAdvance,
   int RunLength, int Color)
{
   int i;
   char far *WorkingScreenPtr = *ScreenPtr;

   for (i=0; i<RunLength; i++)
   {
      *WorkingScreenPtr = Color;
      WorkingScreenPtr += XAdvance;
   }
   /* Advance to the next scan line */
   WorkingScreenPtr += SCREEN_WIDTH;
   *ScreenPtr = WorkingScreenPtr;
}
/* Draws a vertical run of pixels, then advances the bitmap pointer to
   the first pixel of the next run. */
void DrawVerticalRun(char far **ScreenPtr, int XAdvance,
   int RunLength, int Color)
{
   int i;
   char far *WorkingScreenPtr = *ScreenPtr;

   for (i=0; i<RunLength; i++)
   {
      *WorkingScreenPtr = Color;
```

```
      WorkingScreenPtr += SCREEN_WIDTH;
   }
   /* Advance to the next column */
   WorkingScreenPtr += XAdvance;
   *ScreenPtr = WorkingScreenPtr;
}
```

[LISTING TWO]

```
/* Sample line-drawing program. Adapted from code that appeared in the
   _On Graphics_ column in Programmer's Journal. Tested with
   Borland C++ 3.0 in the small model. */

#include <dos.h>

#define GRAPHICS_MODE   0x13
#define TEXT_MODE       0x03
#define BIOS_VIDEO_INT  0x10
#define X_MAX           320      /* working screen width */
#define Y_MAX           200      /* working screen height */

extern void LineDraw(int XStart, int YStart, int XEnd, int YEnd, int Color);

/* Subroutine to draw a rectangle full of vectors, of the specified
 * length and color, around the specified rectangle center.  */
void VectorsUp(XCenter, YCenter, XLength, YLength, Color)
int XCenter, YCenter;   /* center of rectangle to fill */
int XLength, YLength;   /* distance from center to edge of rectangle */
int Color;              /* color to draw lines in */
{
   int WorkingX, WorkingY;

   /* Lines from center to top of rectangle */
   WorkingX = XCenter - XLength;
   WorkingY = YCenter - YLength;
   for ( ; WorkingX < ( XCenter + XLength ); WorkingX++ )
   {
      LineDraw(XCenter, YCenter, WorkingX, WorkingY, Color);
   }
   /* Lines from center to right of rectangle */
   WorkingX = XCenter + XLength - 1;
   WorkingY = YCenter - YLength;
   for ( ; WorkingY < ( YCenter + YLength ); WorkingY++ )
   {
      LineDraw(XCenter, YCenter, WorkingX, WorkingY, Color);
   }
   /* Lines from center to bottom of rectangle */
   WorkingX = XCenter + XLength - 1;
   WorkingY = YCenter + YLength - 1;
   for ( ; WorkingX >= ( XCenter - XLength ); WorkingX-- )
   {
      LineDraw(XCenter, YCenter, WorkingX, WorkingY, Color);
   }
   /* Lines from center to left of rectangle */
   WorkingX = XCenter - XLength;
   WorkingY = YCenter + YLength - 1;
   for ( ; WorkingY >= ( YCenter - YLength ); WorkingY-- )
   {
      LineDraw(XCenter, YCenter, WorkingX, WorkingY, Color);
   }
}
/* Sample program to draw four rectangles full of lines.  */
int main()
{
   union REGS regs;

   /* Set graphics mode */
   regs.x.ax = GRAPHICS_MODE;
   int86(BIOS_VIDEO_INT, &regs, &regs);

   /* Draw each of four rectangles full of vectors */
```

```
   VectorsUp(X_MAX / 4, Y_MAX / 4, X_MAX / 4, Y_MAX / 4, 1);
   VectorsUp(X_MAX * 3 / 4, Y_MAX / 4, X_MAX / 4, Y_MAX / 4, 2);
   VectorsUp(X_MAX / 4, Y_MAX * 3 / 4, X_MAX / 4, Y_MAX / 4, 3);
   VectorsUp(X_MAX * 3 / 4, Y_MAX * 3 / 4, X_MAX / 4, Y_MAX / 4, 4);

   /* Wait for a key to be pressed */
   getch();

   /* Back to text mode */
   regs.x.ax = TEXT_MODE;
   int86(BIOS_VIDEO_INT, &regs, &regs);
}
```

Example 1:

```
for (i=0; i<RunLength; i++)
{
   *WorkingScreenPtr = Color;
   if (XDelta > 0)
   {
      WorkingScreenPtr++;
   }
   else
   {
      WorkingScreenPtr--;
   }
}
```

Example 2:

```
if (XDelta > 0)
{
   for (i=0; i<RunLength; i++)
   {
      *WorkingScreenPtr++ = Color;
   }
}
else
{
   for (i=0; i<RunLength; i++)
   {
      *WorkingScreenPtr-- = Color;
   }
}
```

Journal: Dr. Dobb's Journal Dec 1992 v17 n12 p143(5)

Moving, faster lines, and page flipping

Author: Abrash, Michael

Abstract: Graphics programming can be optimized, however if the program is running well it may be better not to try and change it. Serge Mathieu of Concepteva Inc has a useful twist on animation. Serge's system is a combination of page flipping and dirty-rectangle animation: set the start address to display page (), draw to page 1, set the start address to display page 1, wait for leading edge of vertical sync, copy from page 1 to page 0, set the address to display page 0, now identical to page 1, wait for leading edge of vertical sync. This method makes it necessary to maintain only page and eliminates the complications involved with maintaining two separate pages.

As I write this, the wife, the kid, and I are in the throes of yet another lightning-quick transcontinental move, this time to Redmond to work for You Know Who. Moving is never fun, but what makes it worse for us is the pets. Getting them into kennels and to the airport is hard; there's always the possibility that they might not be allowed to fly because of the weather; and, worst of all, they might not make it. Animals don't usually end up injured or dead, but it does happen.

In a (not notably successful) effort to cheer me up about the prospect of shipping my animals, a friend told me the following story, which he swears actually happened to a friend of his. I don't know--to me, it has the sound of an urban legend, which is to say it makes a good story, but you can never track down the person it really happened to; it's always a friend of a friend. But maybe it is true, and anyway, it's a good story.

This friend of a friend (henceforth referred to as FOF), worked in an air-freight terminal. Consequently, he handled a lot of animals, which was fine by him, because he liked animals; in fact, he had quite a few cats at home.

You can imagine his dismay when, one day, he took a kennel off the plane to find that the cat it carried was quite thoroughly dead. (No, it wasn't resting; this cat was bloody deceased.)

FOF knew how upset the owner would be, and came up with a plan to make everything better. At home, he had a cat of the same size, shape, and markings. He would substitute that cat, and since all cats treat all humans with equal disdain, the owner would never know the difference, and would never suffer the trauma of the loss of her cat. So FOF drove home, got his cat, put it in the kennel, and waited for the owner to show up--at which point, she took one look at the kennel and said, "This isn't my cat. My cat is dead."

As it turned out, she had shipped her recently deceased feline home to be buried. History does not record how FOF dug himself out of this one.

Okay, but what's the point? The point is, if it isn't broken, don't fix it. And if it is broken, maybe that's all right, too. Which brings us, neat as a pin, to the topic of drawing lines in a serious hurry.

Fast Run-length Slice Line Drawing

Last month, we examined the principles of run-length slice line drawing, which draws lines a run rather than a pixel at a time, a run being a series of pixels along the major (longer) axis. I concluded by promising a fast assembler version for this month. Listing One (page 159) is the promised code, in a form that's plug-compatible with the C code from last month.

Your first question is likely to be the following: Just how fast is Listing One? Is it optimized to the hilt, or just pretty fast? The quick answer is: It's fast. Listing One draws lines at a rate of nearly 1 million pixels per second on my 486/33, and is capable of still faster drawing, as I'll discuss shortly. (The heavily optimized AutoCAD line-drawing code that I mentioned last month drew 150,000 pixels per second on an EGA in a 386/16, and I thought I had died and gone to Heaven. Such is progress.) The full answer is a more complicated one, and ties in to the principle that if it is broken, maybe that's okay--and to the principle of looking before you leap, also known as profiling before you optimize.

When I went to speed up run-length slice lines, I initially manually converted the C code from last month into assembler. Then I streamlined the register usage and used REP STOS wherever possible. Listing One is that code. At that point, line drawing was surely faster, although I didn't know exactly how much faster. Equally surely, there were significant optimizations yet to be made, and I was itching to get on to them, for they were a lot more interesting than a basic C-to-assembler port.

Ego intervened at this point, however. I wanted to know how much of a speed-up I had already gotten, so I timed the performance of the C code vs. the assembler code. To my horror, I found that I had not gotten even a two-times improvement! I couldn't understand how that could be--the C code was decidedly unoptimized--until I hit on the idea of measuring the maximum memory speed of the VGA to which I was drawing.

Bingo. The Paradise VGA in my 486/33 is fast for a single display-memory write, because it buffers the data, lets the CPU go on its merry way, and finishes the write when display memory is ready. However, the maximum rate at which data can be written to the adapter turns out to be no more than one byte every microsecond. Put another way, you can only write one byte to this adapter every 33 clock cycles on a 486/33. Therefore, no matter how fast I made the line-drawing code, it could never draw more than 1,000,000 pixels per second in 256-color mode in my system. The C code was already drawing at about half that rate, so the potential speed-up for the assembler code was limited to a maximum of two times, which is pretty close to what Listing One did, in fact, achieve. When I compared the C and assembler implementations drawing to normal system (nondisplay) memory, I found that the assembler code was actually four times as fast as the C code.

In fact, Listing One draws lines at about 92 percent of the maximum possible rate in my system--that is, it draws very nearly as fast as the VGA hardware will allow. All the optimization in the world would get me less than 10 percent faster line drawing--and that only if I eliminated all overhead, an unlikely proposition at best. The code isn't fully optimized, but so what?

Now it's true that faster line-drawing code would likely be more beneficial on faster VGAs, especially local-bus VGAs, and in slower systems. For that reason, I'll list a variety of potential optimizations to Listing One. On the other hand, it's also true that Listing One is capable of drawing lines at a rate of 2.2 million pixels per second on a 486/33, given fast enough VGA memory, so it should be able to drive almost any non-local-bus VGA at nearly full speed. In short, Listing One is very fast, and, in many systems, further optimization is basically a waste of time.

Profile before you optimize.

Further Optimizations

Following is a quick tour of some of the many possible further optimizations to Listing One.

The run-handling loops could be unrolled more than the current two times. However, bear in mind that a two-times unrolling gets more than half the maximum unrolling benefit with less overhead than a more heavily unrolled loop.

BX could be freed up in the Y-major code by breaking out separate loops for X advances of 1 and -1. DX could be freed up by using AH as the counter for the run loops, although this would limit the maximum line length that could be handled. The freed registers could be used to keep more of the whole-step and error variables in registers. Alternatively, the freed registers could be used to implement more esoteric approaches like unrolling the Y-major inner loop; such unrolling could take advantage of the knowledge that only two run lengths are possible for any given line. Strangely enough, on the 486 it might also be worth unrolling the X-major inner loop, which consists of REP STOSB, because of the slow start-up time of REP relative to the speed of branching on that processor.

Special code could be implemented for lines with integral slopes, because all runs are exactly the same length in such lines. Also, the X-major code could try to write an aligned word at a time to display memory whenever possible; this would improve the maximum possible performance on some 16-bit VGAs.

One weakness of Listing One is that for lines with slopes between 0.5 and 2, the average run length is less than two, rendering run-length slicing ineffective. This can be remedied by viewing lines in that range as being composed of diagonal, rather than horizontal or vertical runs. I haven't space to discuss this, but it's not very complicated, and it guarantees a minimum run length of 2. That renders run drawing considerably more efficient, and makes techniques such as unrolling the inner run-drawing loops more attractive.

Finally, be aware that run-length slice drawing is best for long lines, because it has more and slower setup than standard Bresenham's, including a divide. Run-length slice is great for 100-pixel lines, but not necessarily for 20-pixel lines, and it's a sure thing that it's not terrific for 3-pixel lines. Both approaches will work, but if line-drawing performance is critical, whether you'll want to use run-length slice or standard Bresenham's depends on the typical lengths of the lines you'll be drawing. For lines of widely varying lengths, you might want to implement both approaches, and choose the best one for each line, depending on the line length--assuming, of course, that your display memory is fast enough and your application demanding enough to make that level of optimization worthwhile.

An Interesting Twist on Page Flipping

I've spent a fair amount of time exploring various ways to do animation. (See, for example, my July, August, and September 1991 DDJ columns, as well as those in the January through April 1992 issues.) I thought I had pegged all the possible ways to do animation: exclusive-ORing; simply drawing and erasing objects; drawing objects with a blank fringe to erase them at their old locations as they're drawn; page flipping; and, finally, drawing to local memory and copying the dirty (modified) rectangles to the screen.

To my surprise, someone threw me an interesting and useful twist on animation the other day, a cross between page flipping and dirty-rectangle animation. That someone was Serge Mathieu of Concepteva Inc., in Rosemere, Quebec, who informed me that he designs everything "from a game 'point de vue'."

In normal page flipping, you display one page while you update the other page. Then you display the new page while you update the other. This works fine, but the need to keep two pages current can make for a lot of bookkeeping and possibly extra drawing, especially in applications where only some of the objects are redrawn each time.

Serge didn't care to do all that bookkeeping in his animation applications, so he came up with the following approach (which I've reworded, amplified, and slightly modified):

1. Set the start address to display page 0.

2. Draw to page 1.

3. Set the start address to display page 1 (the newly drawn page), then wait for the leading edge of vertical sync, at which point the page has flipped and it's safe to modify page 0.

4. Copy, via the latches, from page 1 to page 0 the areas that changed from the last screen to the current one.

5. Set the start address to display page 0, which is now identical to page 1, then wait for the leading edge of vertical sync, at which point the page has flipped and it's safe to modify page 1.

6. Go to step 2.

The great benefit of Serge's approach is that the only page that is ever actually drawn to (as opposed to block-copied to) is page 1. Only one page needs to be maintained, and the complications of maintaining two separate pages vanish entirely. The performance of Serge's approach may be better or worse than standard page flipping, depending on whether a lot of extra work is required to maintain two pages or not. My guess is that Serge's approach will usually be slower, owing to the considerable amount of display-memory copying involved, and also to the double page-flip per frame. There's no doubt, however, that Serge's approach is simpler, and the resultant display quality is every bit as good as standard page flipping. Given page flipping's fair degree of complication, this approach is a valuable tool, especially for less-experienced animation programmers.

An interesting variation on Serge's approach doesn't page flip or wait for vertical sync:

1. Set the start address to display page 0.

2. Draw to page 1.

3. Copy, via the latches, the areas that changed from the last screen to the current one from page 1 to page 0.

4. Go to step 2.

This approach totally eliminates page flipping, which can consume a great deal of time. The downside is that images may shear for one frame if they're only partially copied when the raster beam reaches them. This approach is basically a standard dirty-rectangle approach, except that the drawing buffer is stored in display memory, rather than in system memory. Whether this technique is faster than drawing to system memory depends on whether the benefit you get from the VGA's hardware, such as the Bit Mask, the ALUs, and especially the latches (for copying the dirty rectangles) is sufficient to outweigh the extra display-memory accesses involved in drawing, since display memory is notoriously slow.

Finally, I'd like to point out that in any scheme that involves changing the display-memory start address, a clever trick can potentially reduce the time spent waiting for pages to flip. Normally, it's necessary to wait for display enable to be active, then set the two start address registers, and finally wait for vertical sync to be active, so you know the new start address has taken effect. The start-address registers must never be set around the time vertical sync is active (the new start address is accepted at either the start or end of vertical sync on the EGAs and VGAs I'm familiar with), because it would then be possible to load a half-changed start address (one register loaded, the other not yet loaded), and the screen would jump for a frame. Avoiding this condition is the motivation for waiting for display enable, because display enable is active only when vertical sync is not active and will not become active for a long while.

Suppose, however, that you arrange your page start addresses so that they both have a low-byte value of 0 (page 0 starts at 0000h, and page 1 starts at 8000h, for example). Page flipping can then be done simply by setting the new high byte of the start address, then waiting for the leading edge of vertical sync. This eliminates the need to wait for display enable (the two bytes of the start address can never be mismatched); page flipping will often involve less waiting, because display enable becomes inactive long before vertical sync becomes active.

Using the above approach reclaims all the time between the end of display enable and the start of vertical sync for doing useful work. (The steps I've given for Serge's animation approach assume that the single-byte approach is in use; that's why display enable is never waited for.)

Thanks

I took another bundle of reader contributions from the X-Sharp careware over to the Vermont Association for the Blind the other day. They were very grateful. Thanks to all of you who have helped so far.

Journal: Dr. Dobb's Journal Jan 1993 v18 n1 p127(6)

Yet another animation method

Author: Abrash, Michael

Abstract: A graphics technique called dirty-rectangle animation can overcome many of the performance problems encountered with VGA display monitors. The technique is called dirty because the graphics drawn using the technique do not match those that appear on-screen. Rather than draw directly to the screen, images are drawn and stored in off-screen or non-display memory. The list of the bounding rectangles for the drawn-to areas are the dirty rectangles. These bounding rectangles are transferred to the screen once all drawing and redrawing is complete. Drawing and redrawing directly to the screen creates excessive flicker and reduces the visual presentation's quality. Dirty-rectangle animation improves image presentation because only the final pixel representation appears on-screen. The technique is also faster because it limits the amount of interaction with VGA hardware.

As documented last month, we brought our pets with us when we moved out here to Seattle. At about the same time, our Golden Retriever, Sam, observed his third birthday. Sam is relatively intelligent, in the sense that he is clearly smarter than a Banana Slug, although if he were in the same room with Jeff Duntemann's dogs Mr. Byte and Chewy, there's a reasonable chance that he would mistake them for something edible (a category that includes rocks, socks, and a surprising number of things too disgusting to mention), and Jeff would have to find a new source of openings for his column.

But that's not important now. What is important is that--and I am not making this up--this morning I managed to find the one pair of socks Sam hadn't chewed holes in. And what's even more important is that after we moved and Sam turned three, he calmed down amazingly. We had been waiting for this magic transformation since Sam turned one, the age at which most puppies turn into normal dogs who lie around a lot, waking up to eat their Science Diet (motto, "The dog food that costs more than the average neurosurgeon makes in a year") before licking themselves and going back to sleep. When Sam turned one and remained hopelessly out of control we said, "Goldens take two years to calm down," as if we had a clue. When he turned two and remained undeniably Sam we said, "Any day now." By the time he turned three, we were reduced to figuring that it was only about seven more years until he expired, at which point we might be able to take all the fur he had shed in his lifetime and weave ourselves some clothes without holes in them, or quite possibly a house.

But miracle of miracles, we moved, and Sam instantly turned into the dog we thought we'd gotten when we forked over $500--calm, sweet, and obedient. Weeks went by, and Sam was, if anything, better than ever. Clearly, the change was permanent.

And then we took Sam to the vet for his annual check-up and found that he had an ear infection. Thanks to the wonders of modern animal medicine, a $5 bottle of liquid restored his health in just two days. And with his health, we got, as a bonus, the old Sam. You see, Sam hadn't changed. He was just tired from being sick. Now he once again joyously knocks down any stranger who makes the mistake of glancing in his direction, and will, quite possibly, be booked any day now on suspicion of homicide by licking.

Okay, you give up. What exactly does this have to do with graphics? I'm glad you asked. The lesson to be learned from Sam The Dog With A Brain The Size Of A Walnut is that while things may look like they've changed, in fact they often haven't. Take VGA performance. If you buy a 486 with a Super-VGA, you'll get performance that knocks your socks off, especially if you run Windows. Things are liable to be so fast that you'll figure the Super-VGA has to deserve some of the credit. Well, maybe it does if it's a local-bus VGA. But maybe it doesn't, even if it

is local bus--and it certainly doesn't if it's an ISA-bus VGA, because no ISA-bus VGA can run faster than about 300 nanoseconds per access, and VGAs capable of that speed have been common for at least a couple of years now. Your 486 VGA system is fast almost entirely because it has a 486 in it. (486 systems with accelerators such as the ATI Ultra or Diamond Stealth are another story altogether.) Underneath it all, the VGA is still painfully slow--and if you have an old VGA or IBM's original PS/2 motherboard VGA, it's incredibly slow. The fastest ISA-bus VGA around is two to twenty times slower than system memory, and the slowest VGA around is as much as 100 times slower. In the old days, the rule was, "Display memory is slow, and should be avoided." Nowadays, the rule is, "Display memory is not quite so slow, but should still be avoided."

So, as I say, sometimes things don't change. Of course, sometimes they do change. For example, in just 49 dog years, I fully expect to own at least one pair of underwear without a single hole in it. Which brings us, deus ex machina and the creek don't rise, to yet another animation method: dirty-rectangle animation.

VGA Access Times

Actually, before we get to dirty rectangles, I'd like to take you through a quick refresher on VGA memory and I/O access times. I want to do this partly because the slow access times of the VGA make dirty-rectangle animation particularly attractive, and partly as a public service, because even I was shocked by the results of some I/O performance tests I recently ran.

Table 1 shows the results of the aforementioned I/O performance tests, as run on two 486/33 Super-VGA systems under the Phar Lap 386!DOS-Extender. (The systems and VGAs are unnamed because this is a not-very-scientific spot test, and I don't want to unfairly malign, say, a VGA whose only sin is being plugged into a lousy motherboard, or vice versa.) Under Phar Lap, 32-bit protected-mode apps run with full I/O privileges, meaning that the OUTs I measured had the best official cycle times possible on the 486: 10 cycles. OUT takes 16 cycles in real mode on a 486, and a mind-boggling 30 cycles in protected mode if running without full I/O privileges (as is normally the case for protected-mode applications). Basically, I/O is just plain slow on a 486.

Slow as 30 or even 10 cycles for an OUT is, one could only wish that VGA I/O was actually that fast. The fastest OUT in Table 1 is 26 cycles, and the slowest is 126--this for an operation that's supposed to take 10 cycles. To put this in context, MUL takes only 13 to 42 cycles, and a normal MOV to or from system memory takes exactly one cycle on the 486. In short, OUTs to VGAs are as much as 100 times slower than normal memory accesses, and are generally two to four times slower than display memory accesses, although there are exceptions.

Of course, VGA display memory has its own performance problems. The fastest ISA-bus VGA can, at best, support sustained write times of about 10 cycles per word-sized write; 15 or 20 cycles is more common, even for relatively fast Super-VGAs; the worst case I've seen is 65 cycles per byte. However, intermittent writes, mixed with a lot of register- and cache-only code, can effectively execute in one cycle because the VGA and the 486 coprocess. Display memory reads tend to take longer, because coprocessing isn't possible--one microsecond is a reasonable rule of thumb for VGA reads, although there's considerable variation. So VGA memory tends not to be as bad as VGA I/O, but Lord knows it isn't good.

In conclusion, OUTs, in general, are lousy on the 486 (and to think they only took three cycles on the 286!). OUTs to VGAs are particularly lousy. Display memory performance is pretty poor, especially for reads. The conclusions are obvious, I would hope. Structure your graphics code, and, in general, all 486 code, to avoid OUTs. For graphics, this especially means using write mode 3 rather than the bit-mask register. When you must use the bit mask, arrange drawing so that you can set the bit mask once, then do a lot of drawing with that mask. For example, draw a whole edge at once, then the middle, then the other edge, rather than setting the bit mask several times on each scan line to draw the edge and middle bytes together. Don't read from display memory if you don't have to. Write each pixel once and only once.

It is indeed a strange concept: The key to fast graphics is staying away from the graphics adapter as much as possible.

Dirty-rectangle Animation

The relative slowness of VGA hardware is part of the appeal of the technique that I call "dirty-rectangle" animation, in which a complete copy of the contents of display memory is maintained in off-screen system (nondisplay) memory. All drawing is done to this system buffer. As offscreen drawing is done, a list is maintained of the bounding rectangles for the drawn-to areas; these are the "dirty" rectangles, dirty in the sense that they do not match the contents of the screen. After all drawing for a frame is completed, all the dirty rectangles for that frame are copied to the screen in a burst, and then the cycle of off-screen drawing begins again.

Why, exactly, would we want to go through all this complication, rather than simply drawing to the screen in the first place? The reason is visual quality. If we were to do all our drawing directly to the screen, there'd be a lot of flicker as objects were erased and then redrawn. Similarly, overlapped drawing done with the painter's algorithm (in which farther objects are drawn first, so that nearer objects obscure them) would flicker as farther objects were visible for short periods. With dirty-rectangle animation, only the finished pixels for any given frame ever appear on the screen; intermediate results are never visible. Figure 1 illustrates the visual problems associated with drawing directly to the screen; Figure 2 shows how dirty-rectangle animation solves these problems.

Well, then, if we want good visual quality, why not use page flipping? For one thing, not all adapters and modes support page flipping. The CGA and MCGA don't, and neither do the VGA's 640x480 16-color or 320x200 256-color modes, or many Super-VGA modes. In contrast, all adapters support dirty-rectangle animation. Another advantage of dirty-rectangle animation is that it's generally faster. While it may seem strange that it would be faster to draw off screen and then copy the result to the screen, that is often the case, because dirty-rectangle animation usually reduces the number of times the VGA's hardware needs to be touched, especially in 256-color modes. This reduction comes about because when dirty rectangles are erased, it's done in system memory, not in display memory, and since most objects move a good deal less than their full width (that is, the new and old positions overlap), display memory is written to fewer times than with page flipping. (In 16-color modes, this is not necessarily the case, because of the parallelism obtained from the VGA's planar hardware.) Also, read/modify/write operations are performed in fast system memory rather than slow display memory, so display memory rarely needs to be read. This is particularly good because display memory is generally even slower for reads than for writes.

Also, page flipping wastes a good deal of time waiting for the page to flip at the end of the frame. Dirty-rectangle animation never needs to wait for anything because partially drawn images are never present in display memory. Actually, in one sense, partially drawn images are sometimes present because it's possibly for a rectangle to be partially drawn when the scanning raster beam reaches that part of the screen. This causes the rectangle to appear partially drawn for one frame, producing a phenomenon I call "shearing." Fortunately, shearing tends not to be particularly distracting, especially for fairly small images, but it can be a problem when copying large areas. This is one area in which dirty-rectangle animation falls short of page flipping, because page flipping has perfect display quality, never showing anything other than a completely finished frame. Similarly, dirty-rectangle copying may take two or more frame times to finish, so even if shearing doesn't happen, it's still possible to have the images in the various dirty rectangles show up nonsimultaneously. In my experience, this latter phenomenon is not a serious problem, but do be aware of it.

Dirty Rectangles in Action

Listing One (page 140) demonstrates dirty-rectangle animation. This is a very simple implementation, in several respects. For one thing, it's written entirely in C, and animation fairly cries out for assembly language. For another thing, it uses far pointers, which C often handles with less than optimal efficiency, especially because I haven't used library functions to copy and fill memory. (I did this so the code would work in any memory model.) Also, Listing One doesn't attempt to coalesce rectangles so as to perform a minimum number of display-memory accesses; instead, it copies each dirty rectangle to the screen, even if it overlaps with another rectangle, so some pixels get copied multiple times. Listing One runs pretty well, considering all of its failings; on my 486/33, ten 11x11 images animate at a very respectable clip.

One point I'd like to make is that although the system-memory buffer in Listing One has exactly the same dimensions as the screen bitmap, that's not a requirement, and there are some good reasons not to make the two the same size. For example, if the system buffer is bigger than the screen, it's possible to pan the visible area around the system buffer. Or, alternatively, the system buffer can be just the size of a desired window, representing a window into a larger, virtual buffer. We could then draw the desired portion of the virtual bitmap into the system-memory buffer, then copy the buffer to the screen, and the effect will be of having panned the window to the new location.

Another argument in favor of a small viewing window is that it restricts the amount of display memory actually drawn to. Restricting the display memory used for animation reduces the total number of display-memory accesses, which in turn boosts overall performance; it also improves the performance and appearance of panning, in which the whole window has to be redrawn or copied. If you keep a close watch, you'll notice that many high-performance animation games similarly restrict their full-featured animation area to a relatively small region. Often, it's hard to tell that this is the case, because the animation region is surrounded by flashy digitized graphics and by items such as scoreboards and status screens, but look closely and see if the animation region in your favorite game isn't smaller than you thought.

Next month, I'll put the important parts of dirty-rectangle animation into assembler, and I'll coalesce dirty rectangles to minimize display-memory accesses--and maybe, just maybe, I'll do some panning. Then we'll see what kind of stuff dirty-rectangle animation is really made of.

3-D Reading

As anyone who's been following this column for a while knows, I'm keenly interested in 3-D graphics. Thus, it is with considerable pleasure that I'm able to report that Programming in 3 Dimensions: 3-D Graphics, Ray Tracing, and Animation by Christopher D. Watkins and Larry Sharp (M&T Books, 1992) is good stuff. There's a fair amount of theory, and lots of 3-D implementation, from modeling and scenes to ray tracing and finally, animation. The animation is the precomputed, playback kind, of the Autodesk Animator sort, and while it lacks the on-the-fly flexibility of the real-time animation we've developed in this column, my oh my, it does look good. If you get this book, I strongly suggest you get the disk as well; in which case, run ANIMATE.EXE, with BOUNCE as the input file, and marvel that you now have, in source form, all the software needed to implement that animation. Ten years ago, I'll bet you couldn't have produced this level of fully rendered, real-time playback animation for less than $50,000 in hardware and software; now, a couple of thousand will easily do the trick. What a great time this is to be a programmer! Recommended.

Journal: Dr. Dobb's Journal Feb 1993 v18 n2 p127(6)

More dirty (dirtier?) rectangles

Author: Abrash, Michael
Attached: GP-FEB93.ASC Source code listing.

Abstract: Dirty-rectangle animation is a method of animation programming that results in good visual quality without the video hardware and memory necessary for page flipping. The dirty-rectangle method can add further advantages with the use of low-level drawing routines written in assembly language to increase performance. Masking an object before drawing it allows pixels to penetrate only through holes in the mask. Masked images, which are sometimes called sprites, make animation more realistic. Drawing masked images can be slow, but speed can be increased with the use of assembly language. Another good animation programming technique is internal animation, where the images themselves instead of their positions are changed. Other recommended techniques are discussed.

Programming is, by and large, a linear process. One statement or instruction follows another, in predictable sequences, with tiny building blocks strung together to make a custom state machine. As programmers, we grow adept at this sort of idealized linear thinking, which is, of course, A Good Thing. Still, it's important to keep in mind there's a large chunk of the human mind that doesn't work in a linear fashion.

I've written elsewhere about the virtues of nonlinear/right-brian/lateral/what-have-you thinking in solving tough programming problems, such as debugging or optimization, but it bears repeating. The mind can be an awesome pattern-matching and extrapolation tool, if you let it. For example, the other day I was grinding my way through a particularly difficult bit of debugging. The code had been written by someone else, and, to my mind, there's nothing worse than debugging someone eles's code; there's always the nasty feeling that you don't quite know what's going on. The overall operation of this code wouldn't come clear in my head, no matter how long I started at it, leaving me with a rising sense of frustration and a determination not to quite until I got this bug.

In the midst of this, a coworker poked his head through the door and told me he had somthing I had to listen to. Reluctantly, I went to his office, whereupon he played a tape of what is surely one of the most bizarre 911 calls in history. No doubt some of you have heard this tape, which I will briefly described as involving a deer destroying the interior of a car and bitting a man in the neck. Perhaps you found it funny, perhaps not--but as for me, it hit me exactly right. I started laughing helplessly, tears rolling down my face. When I went back to work--presto!--the pieces of the debugging puzzle had come together in my head, and the work went quickly and easily.

Obviously, my mind needed a break from linear, left-brain, push-it-out thinking, so it could do the sort of integrating work it does so well--but that it's rarely willing to do under conscious control. It was exactly this sort of thinking I had in mind when I titled my book Zen of Assembly Language. (Although I must admit that few people seem to have gotten the connection, and I've had to field a lot of questions about whether I'm a Zen disciple. I'm not--more of a Dave Barry disciple. If you don't know who Dave Barry is, you should; he's good for your right brain.) Give your mind a break once in a while, and I'll bet you'll find you're more productive.

We're strange thinking machines, but we're the best ones yet invented, and it's worth learning how to tap our full potential. And with that, it's back to dirty-rectangle animation.

Dirty-rectangle Animation, Continued

Last month, we got our feet wet with dirty-rectangle animation. This technique is an alternative to page flipping that's capable of producing animation of very high visual quality, without any help at all from video hardware, and without the need for any extra, nondisplayed video memory. This makes dirty-rectangle animation more widely usable than page flipping, because many adapters don't support page flipping. Dirty-rectangle animation also tends to be simpler to implement than page flipping, because there's only one bitmap to keep track of. A final advantage of dirty-rectangle animation is that it's potentially somewhat faster than page flipping, because display-memory accesses can theoretically be reduced to exactly one access for each pixel that changes from one frame to the next.

The speed advantage of dirty-rectangle animation was entirely theoretical last month, because the implementation was completely in C, and because no attempt was made to minimize display-memory accesses. The visual quality of last month's animation was also less than ideal, for reasons we'll explore shortly. The code in Listings One (page 142) and Two (page 144) addresses the shortcomings of last month's code.

Listing Two implements the low-level drawing routines in assembly language, which boosts performance a good deal. For maximum performance, it would be worthwhile to convert mor of Listing One into assembler, so a call isn't required for each animated image, and overall performance could be improved by streamlining the C code, but Listing Two goes a long way toward boosting animation speed. This program now supports snappy animation of 15 images (as opposed to 10 last month), and the images are two pixels wider this month. That level of performance is all the more impressive considering that this month I've converted the code from using rectangular images to using masked images.

Masked Images

Masked images are rendered by drawing an object's pixels through a mask; pixels are actually drawn only where the mask specifies that drawing is allowed. This makes it possible to draw nonrectangular objects that don't improperly interfere with each other when they overlap. Masked images also make it possible to have transparent areas (windows) within objects. Masked images produce far more realistic animation than do rectangular images, and therefore are more desirable. Unfortunately, masked images are also considerably slower to draw; however, a good assembly language implementation can go a long way toward making masked images draw rapidly enough, as illustrated by this month's code. (Masked images are also known as "sprites;" some video hardware supports sprites directly, but on the IBM PC it's necessary to handle sprites in software.)

Masked images make it possible to render scenes so a given image convincingly appears to be in front of or behind other images; that is, so images are displayed in z-order (by distance). By consistently drawing images that are supposed to be farther away before drawing nearer images, the nearer images will appear in front of the other images, and because masked images draw only precisely the correct pixels (as opposed to blank pixels in the bounding rectangle), there's no interference between overlapping images to destroy the illusion.

Internal Animation

I've added another feature essential to producing convincing animation: internal animation, the process of changing the appearance of a given object over time, as distinguished from changing the location of a given object. Internal animation makes images look active and alive. I've implemented the simplest possible form of internal animation in Listing One--alternation between two images--but even this level of internal animation greatly improves the feel of the overall animation. You could easily increase the number of images cycled through, simply by increasing Internal-AnimateMax for a given entity. You could also implement more complex image-selection logic to proudce more interesting and less

predictable internal-animation effects, such as jumping, ducking, running, and the like.

Dirty-rectangle Management

As mentioned above, dirty-rectangle animation makes it possible to access display memory a minimum number of times. Last month's code didn't do any of that; instead, it copied every dirty rectangle to the screen, regardless of overlap between rectangles. This month's code goes to the other extreme, taking great pains never to draw overlapped portions of rectangles more than once. This is accomplished by checking for overlap whenever a rectangle is to be added to the dirty list. When overlap with an existing rectangle is detected, the new rectangle is reduced to between zero and four nonoverlapping rectangles. Those rectangles are then again considered for addition to the dirty list, and may again be reduced, if additional overlap is detected.

A good deal of code is required to generate a fully nonoverlapped dirty list. Is it worth it? It certainly can be, but in Listing One, it probably isn't. For one thing, you'd need bigger, heavily overlapped objects for this approach to pay off big. Besides, this program is mostly in C, and spends a lot of time doing things other than actually accessing dispaly memory. It also takes a fair amount of time just to generate the nonoverlapped list; the overhead of all the looping, intersecting, and calling required to generate the list eats up a lot of the benefits of accessing display memory less often. Nonetheless, fully nonoverlapped drawing can be useful under the right circumstances, and I've implemented it in Listing One so you'll have something to refer to should you decide to try this route.

There are a couple of additional techniques you might try if you want to wring maximum performance out of dirty-rectangle animation. You could try coalescing rectangles as you generate the dirty-rectangle list. That is, you could detect pairs of rectangles that can be joined together into larger rectangles, so that fewer, larger rectangles would have to be copied. This would boost the efficiency of the low-level copying code, albeit at the cost of some cycles in the dirty-list management code.

You might also try taking advantage of the natural coherence of animated graphics screens. In particular, because the rectangle used to erase an image at its old location often overlaps the rectangle within which the image resides at its new location, you could simply directly generate the two or three nonoverlapped rectangles required to copy both the erase rectangle and the new-image rectangle for any single moving image. The calculation of these rectangles could be very efficient, given that you know in advance the direction of motion of your images. Handling this particular overlap case would eliminate most overlapped drawing, at a minimal cost; you might then decide to ignore overlapped drawing between different images, which tends to be both less common and more expensive to identify and handle.

Drawing Order and Visual Quality

A final note on dirty-rectangle animation concerns the quality of the displayed screen image. Last month, we simply stuffed dirty rectangles in a list in the order they became dirty, and then copied all of the rectangles in that same order. Unfortunately, this caused all of the erase rectangles to be copied first, followed by all of the rectangles of the images at their new locations. Consequently, there was a significant delay between the appearance of the erase rectangle for a given image and the appearance of the new rectangle. A byproduct was the fact that a partially complete--part old, part new--image was visible long enough to be noticed. In short, although the pixels ended up correct, they were in an intermediate, incorrect state for a sufficient period of time to make the animation look wrong.

This violated a fundamental rule of animation: No pixel should ever be perceptible in an incorrect state. To correct the problem, this month I've sorted the dirty rectangles by Y coordinate, and secondarily by X coordinate. This means the screen updates from the top down, and from

left to right, so the several nonoverlapping rectangles copied to draw a given image should be drawn nearly simultaneously. Run last month's code and then this month's; you'll see quite a difference in appearance.

Avoid the trap of thinking animation is merely a matter of drawing the right pixels, one after another. Animation is the art of drawing the right pixels at the right times so the eye and brain see what you want them to see. Animation is a lot more challenging than merely cranking out pixels, and it sure as heck isn't a purely linear process.

Until We Meet Again

It's been two years since my first graphics column for DDJ. In that time, I've learned a great deal about graphics, partly from my research for the column, but mostly from those of you around the world. For instance, I got two letters from the Ukraine last week. (One was accompanied by a manuscript--in Russian. Two important tips: Send your manuscripts to the DDJ editorial offices, not to me, and write them in English.) You are a remarkably inquisitive and sharing lot, and it's hard to imagine how I could have enjoyed writing this column any more than I have.

Unfortunately, this will be my last column, for now at least. Other responsibilities and challenges beckon, and I've covered much of what I set out to share two years ago (six years ago, if you count my graphics columns in the late Programmer's Journal). I hope the world is a little better because of the interchange of ideas, information, and even a bit of humor that went on in this space. I know I'm richer for having written and corresponded with many of you.

I'm not vanishing off the face of the Earth, of course; we'll meet in these pages again. Until then, thanks for your support and sharing. In particular, thanks for your support of the Careware effort; you've helped change many lives for the better. I'd also like to thank the DDJ staff, especially Monica Berg and Tami Zemel, for their unfailing support and good humor.

Au revoir.

- -

GRAPHICS PROGRAMMING COLUMN
by Michael Abrash

[LISTING ONE]

```
/* Sample simple dirty-rectangle animation program, partially optimized and
featuring internal animation, masked images (sprites), and nonoverlapping dirty
rectangle copying. Tested with Borland C++ 3.0 in the small model. */

#include <stdlib.h>
#include <conio.h>
#include <alloc.h>
#include <memory.h>
#include <dos.h>

/* Comment out to disable overlap elimination in the dirty rectangle list. */
#define CHECK_OVERLAP 1
#define SCREEN_WIDTH  320
#define SCREEN_HEIGHT 200
#define SCREEN_SEGMENT 0xA000

/* Describes a dirty rectangle */
typedef struct {
   void *Next;    /* pointer to next node in linked dirty rect list */
   int Top;
   int Left;
   int Right;
   int Bottom;
} DirtyRectangle;
```

```
/* Describes an animated object */
typedef struct {
   int X;            /* upper left corner in virtual bitmap */
   int Y;
   int XDirection;   /* direction and distance of movement */
   int YDirection;
   int InternalAnimateCount; /* tracking internal animation state */
   int InternalAnimateMax;   /* maximum internal animation state */
} Entity;
/* Storage used for dirty rectangles */
#define MAX_DIRTY_RECTANGLES  100
int NumDirtyRectangles;
DirtyRectangle DirtyRectangles[MAX_DIRTY_RECTANGLES];
/* Head/tail of dirty rectangle list */
DirtyRectangle DirtyHead;
/* If set to 1, ignore dirty rectangle list and copy the whole screen. */
int DrawWholeScreen = 0;
/* Pixels and masks for the two internally animated versions of the image
   we'll animate */
#define IMAGE_WIDTH  13
#define IMAGE_HEIGHT 11
char ImagePixels0[] = {
   0, 0, 0, 9, 9, 9, 9, 9, 0, 0, 0, 0, 0,
   0, 0, 9, 9, 9, 9, 9, 9, 9, 0, 0, 0, 0,
   0, 9, 9, 0, 0,14,14,14, 9, 9, 0, 0, 0,
   9, 9, 0, 0, 0, 0,14,14,14, 9, 9, 0, 0,
   9, 9, 0, 0, 0, 0,14,14,14, 9, 9, 0, 0,
   9, 9,14, 0, 0,14,14,14,14, 9, 9, 0, 0,
   9, 9,14,14,14,14,14,14,14, 9, 9, 0, 0,
   9, 9,14,14,14,14,14,14,14, 9, 9, 0, 0,
   0, 9, 9,14,14,14,14,14, 9, 9, 0, 0, 0,
   0, 0, 9, 9, 9, 9, 9, 9, 9, 0, 0, 0, 0,
   0, 0, 0, 9, 9, 9, 9, 9, 0, 0, 0, 0, 0,
};
char ImageMask0[] = {
   0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0,
   0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0,
   0, 1, 1, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0,
   1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0,
   1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0,
   1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0,
   1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0,
   1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0,
   0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0,
   0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0,
   0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0,
};
char ImagePixels1[] = {
   0, 0, 0, 9, 9, 9, 9, 9, 0, 0, 0, 0, 9,
   0, 0, 9, 9, 9, 9, 9, 9, 9, 0, 9, 9, 9,
   0, 9, 9, 0, 0,14,14,14, 9, 9, 9, 9, 0,
   9, 9, 0, 0, 0, 0,14,14,14, 0, 0, 0, 0,
   9, 9, 0, 0, 0, 0,14,14, 0, 0, 0, 0, 0,
   9, 9,14, 0, 0,14,14,14, 0, 0, 0, 0, 0,
   9, 9,14,14,14,14,14,14, 0, 0, 0, 0, 0,
   9, 9,14,14,14,14,14,14,14, 0, 0, 0, 0,
   0, 9, 9,14,14,14,14,14, 9, 9, 9, 9, 0,
   0, 0, 9, 9, 9, 9, 9, 9, 9, 0, 9, 9, 9,
   0, 0, 0, 9, 9, 9, 9, 9, 0, 0, 0, 9, 9,
};
char ImageMask1[] = {
   0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 0, 1,
   0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1,
   0, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0,
   1, 1, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0,
   1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0,
   1, 1, 1, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0,
   1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0,
   1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0,
   0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0,
   0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1,
   0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 1, 1,
};
```

```
/* Pointers to pixel and mask data for various internally animated
   versions of our animated image. */
char * ImagePixelArray[] = {ImagePixels0, ImagePixels1};
char * ImageMaskArray[] = {ImageMask0, ImageMask1};
/* Animated entities */
#define NUM_ENTITIES 15
Entity Entities[NUM_ENTITIES];
/* Pointer to system buffer into which we'll draw */
char far *SystemBufferPtr;
/* Pointer to screen */
char far *ScreenPtr;
void EraseEntities(void);
void CopyDirtyRectanglesToScreen(void);
void DrawEntities(void);
void AddDirtyRect(Entity *, int, int);
void DrawMasked(char far *, char *, char *, int, int, int);
void FillRect(char far *, int, int, int, int);
void CopyRect(char far *, char far *, int, int, int, int);
void main()
{
   int i, XTemp, YTemp;
   unsigned int TempCount;
   char far *TempPtr;
   union REGS regs;
   /* Allocate memory for the system buffer into which we'll draw */
   if (!(SystemBufferPtr = farmalloc((unsigned int)SCREEN_WIDTH*
         SCREEN_HEIGHT))) {
      printf("Couldn't get memory\n");
      exit(1);
   }
   /* Clear the system buffer */
   TempPtr = SystemBufferPtr;
   for (TempCount = ((unsigned)SCREEN_WIDTH*SCREEN_HEIGHT); TempCount--; ) {
      *TempPtr++ = 0;
   }
   /* Point to the screen */
   ScreenPtr = MK_FP(SCREEN_SEGMENT, 0);
   /* Set up the entities we'll animate, at random locations */
   randomize();
   for (i = 0; i < NUM_ENTITIES; i++) {
      Entities[i].X = random(SCREEN_WIDTH - IMAGE_WIDTH);
      Entities[i].Y = random(SCREEN_HEIGHT - IMAGE_HEIGHT);
      Entities[i].XDirection = 1;
      Entities[i].YDirection = -1;
      Entities[i].InternalAnimateCount = i & 1;
      Entities[i].InternalAnimateMax = 2;
   }
   /* Set the dirty rectangle list to empty, and set up the head/tail node
      as a sentinel */
   NumDirtyRectangles = 0;
   DirtyHead.Next = &DirtyHead;
   DirtyHead.Top = 0x7FFF;
   DirtyHead.Left= 0x7FFF;
   DirtyHead.Bottom = 0x7FFF;
   DirtyHead.Right = 0x7FFF;
   /* Set 320x200 256-color graphics mode */
   regs.x.ax = 0x0013;
   int86(0x10, &regs, &regs);
   /* Loop and draw until a key is pressed */
   do {
      /* Draw the entities to the system buffer at their current locations,
         updating the dirty rectangle list */
      DrawEntities();
      /* Draw the dirty rectangles, or the whole system buffer if
         appropriate */
      CopyDirtyRectanglesToScreen();
      /* Reset the dirty rectangle list to empty */
      NumDirtyRectangles = 0;
      DirtyHead.Next = &DirtyHead;
      /* Erase the entities in the system buffer at their old locations,
         updating the dirty rectangle list */
      EraseEntities();
      /* Move the entities, bouncing off the edges of the screen */
      for (i = 0; i < NUM_ENTITIES; i++) {
```

```
         XTemp = Entities[i].X + Entities[i].XDirection;
         YTemp = Entities[i].Y + Entities[i].YDirection;
         if ((XTemp < 0) || ((XTemp + IMAGE_WIDTH) > SCREEN_WIDTH)) {
            Entities[i].XDirection = -Entities[i].XDirection;
            XTemp = Entities[i].X + Entities[i].XDirection;
         }
         if ((YTemp < 0) || ((YTemp + IMAGE_HEIGHT) > SCREEN_HEIGHT)) {
            Entities[i].YDirection = -Entities[i].YDirection;
            YTemp = Entities[i].Y + Entities[i].YDirection;
         }
         Entities[i].X = XTemp;
         Entities[i].Y = YTemp;
      }
   } while (!kbhit());
   getch();    /* clear the keypress */

   /* Back to text mode */
   regs.x.ax = 0x0003;
   int86(0x10, &regs, &regs);
}
/* Draw entities at their current locations, updating dirty rectangle list. */
void DrawEntities()
{
   int i;
   char far *RowPtrBuffer;
   char *TempPtrImage;
   char *TempPtrMask;
    Entity *EntityPtr;

   for (i = 0, EntityPtr = Entities; i < NUM_ENTITIES; i++, EntityPtr++) {
      /* Remember the dirty rectangle info for this entity */
      AddDirtyRect(EntityPtr, IMAGE_HEIGHT, IMAGE_WIDTH);
      /* Point to the destination in the system buffer */
      RowPtrBuffer = SystemBufferPtr + (EntityPtr->Y * SCREEN_WIDTH) +
            EntityPtr->X;
      /* Advance the image animation pointer */
      if (++EntityPtr->InternalAnimateCount >=
            EntityPtr->InternalAnimateMax) {
         EntityPtr->InternalAnimateCount = 0;
      }
      /* Point to the image and mask to draw */
      TempPtrImage = ImagePixelArray[EntityPtr->InternalAnimateCount];
      TempPtrMask = ImageMaskArray[EntityPtr->InternalAnimateCount];
      DrawMasked(RowPtrBuffer, TempPtrImage, TempPtrMask, IMAGE_HEIGHT,
               IMAGE_WIDTH, SCREEN_WIDTH);
   }
}
/* Copy the dirty rectangles, or the whole system buffer if appropriate,
   to the screen. */
void CopyDirtyRectanglesToScreen()
{
   int i, RectWidth, RectHeight;
   unsigned int Offset;
   DirtyRectangle * DirtyPtr;
   if (DrawWholeScreen) {
      /* Just copy the whole buffer to the screen */
      DrawWholeScreen = 0;
      CopyRect(ScreenPtr, SystemBufferPtr, SCREEN_HEIGHT, SCREEN_WIDTH,
               SCREEN_WIDTH, SCREEN_WIDTH);
   } else {
      /* Copy only the dirty rectangles, in the YX-sorted order in which
         they're linked */
      DirtyPtr = DirtyHead.Next;
      for (i = 0; i < NumDirtyRectangles; i++) {
         /* Offset in both system buffer and screen of image */
         Offset = (unsigned int) (DirtyPtr->Top * SCREEN_WIDTH) +
               DirtyPtr->Left;
         /* Dimensions of dirty rectangle */
         RectWidth = DirtyPtr->Right - DirtyPtr->Left;
         RectHeight = DirtyPtr->Bottom - DirtyPtr->Top;
         /* Copy a dirty rectangle */
         CopyRect(ScreenPtr + Offset, SystemBufferPtr + Offset,
               RectHeight, RectWidth, SCREEN_WIDTH, SCREEN_WIDTH);
```

```
         /* Point to the next dirty rectangle */
         DirtyPtr = DirtyPtr->Next;
      }
   }
}
/* Erase the entities in the system buffer at their current locations,
   updating the dirty rectangle list. */
void EraseEntities()
{
   int i;
   char far *RowPtr;
   for (i = 0; i < NUM_ENTITIES; i++) {
      /* Remember the dirty rectangle info for this entity */
      AddDirtyRect(&Entities[i], IMAGE_HEIGHT, IMAGE_WIDTH);
      /* Point to the destination in the system buffer */
      RowPtr = SystemBufferPtr + (Entities[i].Y * SCREEN_WIDTH) +
            Entities[i].X;
      /* Clear the rectangle */
      FillRect(RowPtr, IMAGE_HEIGHT, IMAGE_WIDTH, SCREEN_WIDTH, 0);
   }
}
/* Add a dirty rectangle to the list. The list is maintained in top-to-bottom,
left-to-right (YX sorted) order, with no pixel ever included twice, to minimize
the number of display memory accesses and to avoid screen artifacts resulting
from a large time interval between erasure and redraw for a given object or for
adjacent objects. The technique used is to check for overlap between the
rectangle and all rectangles already in the list. If no overlap is found, the
rectangle is added to the list. If overlap is found, the rectangle is broken
into nonoverlapping pieces, and the pieces are added to the list by recursive
calls to this function. */
void AddDirtyRect(Entity * pEntity, int ImageHeight, int ImageWidth)
{
   DirtyRectangle * DirtyPtr;
   DirtyRectangle * TempPtr;
   Entity TempEntity;
   int i;
   if (NumDirtyRectangles >= MAX_DIRTY_RECTANGLES) {
      /* Too many dirty rectangles; just redraw the whole screen */
      DrawWholeScreen = 1;
      return;
   }
   /* Remember this dirty rectangle. Break up if necessary to avoid
      overlap with rectangles already in the list, then add whatever
      rectangles are left, in YX sorted order */
#ifdef CHECK_OVERLAP
   /* Check for overlap with existing rectangles */
   TempPtr = DirtyHead.Next;
   for (i = 0; i < NumDirtyRectangles; i++, TempPtr = TempPtr->Next) {
      if ((TempPtr->Left < (pEntity->X + ImageWidth)) &&
          (TempPtr->Right > pEntity->X) &&
          (TempPtr->Top < (pEntity->Y + ImageHeight)) &&
          (TempPtr->Bottom > pEntity->Y)) {

         /* We've found an overlapping rectangle. Calculate the
            rectangles, if any, remaining after subtracting out the
            overlapped areas, and add them to the dirty list */
         /* Check for a nonoverlapped left portion */
         if (TempPtr->Left > pEntity->X) {
            /* There's definitely a nonoverlapped portion at the left; add
               it, but only to at most the top and bottom of the overlapping
               rect; top and bottom strips are taken care of below */
            TempEntity.X = pEntity->X;
            TempEntity.Y = max(pEntity->Y, TempPtr->Top);
            AddDirtyRect(&TempEntity,
                  min(pEntity->Y + ImageHeight, TempPtr->Bottom) -
                  TempEntity.Y,
                        TempPtr->Left - pEntity->X);
         }
         /* Check for a nonoverlapped right portion */
         if (TempPtr->Right < (pEntity->X + ImageWidth)) {
            /* There's definitely a nonoverlapped portion at the right; add
               it, but only to at most the top and bottom of the overlapping
               rect; top and bottom strips are taken care of below */
            TempEntity.X = TempPtr->Right;
```

```
            TempEntity.Y = max(pEntity->Y, TempPtr->Top);
            AddDirtyRect(&TempEntity,
                  min(pEntity->Y + ImageHeight, TempPtr->Bottom) -
                  TempEntity.Y,
                  (pEntity->X + ImageWidth) - TempPtr->Right);
         }
         /* Check for a nonoverlapped top portion */
         if (TempPtr->Top > pEntity->Y) {
            /* There's a top portion that's not overlapped */
            TempEntity.X = pEntity->X;
            TempEntity.Y = pEntity->Y;
            AddDirtyRect(&TempEntity, TempPtr->Top - pEntity->Y, ImageWidth);
         }
         /* Check for a nonoverlapped bottom portion */
         if (TempPtr->Bottom < (pEntity->Y + ImageHeight)) {
            /* There's a bottom portion that's not overlapped */
            TempEntity.X = pEntity->X;
            TempEntity.Y = TempPtr->Bottom;
            AddDirtyRect(&TempEntity,
                  (pEntity->Y + ImageHeight) - TempPtr->Bottom, ImageWidth);
         }
         /* We've added all non-overlapped portions to the dirty list */
         return;
      }
   }
#endif /* CHECK_OVERLAP */
   /* There's no overlap with any existing rectangle, so we can just
      add this rectangle as-is */
   /* Find the YX-sorted insertion point. Searches will always terminate,
      because the head/tail rectangle is set to the maximum values */
   TempPtr = &DirtyHead;
   while (((DirtyRectangle *)TempPtr->Next)->Top < pEntity->Y) {
      TempPtr = TempPtr->Next;
   }
   while ((((DirtyRectangle *)TempPtr->Next)->Top == pEntity->Y) &&
            (((DirtyRectangle *)TempPtr->Next)->Left < pEntity->X)) {
      TempPtr = TempPtr->Next;
   }
   /* Set the rectangle and actually add it to the dirty list */
   DirtyPtr = &DirtyRectangles[NumDirtyRectangles++];
   DirtyPtr->Left = pEntity->X;
   DirtyPtr->Top = pEntity->Y;
   DirtyPtr->Right = pEntity->X + ImageWidth;
   DirtyPtr->Bottom = pEntity->Y + ImageHeight;
   DirtyPtr->Next = TempPtr->Next;
   TempPtr->Next = DirtyPtr;
}
```

[LISTING TWO]

```
; Assembly language helper routines for dirty rectangle animation. Tested with
; TASM 3.0. Fills a rectangle in the specified buffer. C-callable as:
;  void FillRect(char far * BufferPtr, int RectHeight, int RectWidth,
;                int BufferWidth, int Color);
        .model  small
        .code
parms   struc
                dw      ?       ;pushed BP
                dw      ?       ;pushed return address
BufferPtr       dd      ?       ;far pointer to buffer in which to fill
RectHeight      dw      ?       ;height of rectangle to fill
RectWidth       dw      ?       ;width of rectangle to fill
BufferWidth     dw      ?       ;width of buffer in which to fill
Color           dw      ?       ;color with which to fill
parms   ends
        public  _FillRect
_FillRect   proc  near
        cld
        push    bp
        mov     bp,sp
        push    di
```

```
        les     di,[bp+BufferPtr]
        mov     dx,[bp+RectHeight]
        mov     bx,[bp+BufferWidth]
        sub     bx,[bp+RectWidth]       ;distance from end of one dest scan
                                        ; to start of next
        mov     al,byte ptr [bp+Color]
        mov     ah,al                   ;double the color for REP STOSW
RowLoop:
        mov     cx,[bp+RectWidth]
        shr     cx,1
        rep     stosw
        adc     cx,cx
        rep     stosb
        add     di,bx                   ;point to next scan to fill
        dec     dx                      ;count down rows to fill
        jnz     RowLoop

        pop     di
        pop     bp
        ret
_FillRect   endp

; Draws a masked image (a sprite) to the specified buffer. C-callable as:
;     void DrawMasked(char far * BufferPtr, char * Pixels, char * Mask,
;                     int ImageHeight, int ImageWidth, int BufferWidth);
parms2  struc
                dw      ?       ;pushed BP
                dw      ?       ;pushed return address
BufferPtr2      dd      ?       ;far pointer to buffer in which to draw
Pixels          dw      ?       ;pointer to image pixels
Mask            dw      ?       ;pointer to image mask
ImageHeight     dw      ?       ;height of image to draw
ImageWidth      dw      ?       ;width of image to draw
BufferWidth2    dw      ?       ;width of buffer in which to draw
parms2  ends
        public  _DrawMasked
_DrawMasked     proc    near
        cld
        push    bp
        mov     bp,sp
        push    si
        push    di

        les     di,[bp+BufferPtr2]
        mov     si,[bp+Mask]
        mov     bx,[bp+Pixels]
        mov     dx,[bp+ImageHeight]
        mov     ax,[bp+BufferWidth2]
        sub     ax,[bp+ImageWidth]      ;distance from end of one dest scan
        mov     [bp+BufferWidth2],ax    ; to start of next
RowLoop2:
        mov     cx,[bp+ImageWidth]
ColumnLoop:
        lodsb                           ;get the next mask byte
        and     al,al                   ;draw this pixel?
        jz      SkipPixel               ;no
        mov     al,[bx]                 ;yes, draw the pixel
        mov     es:[di],al
SkipPixel:
        inc     bx                      ;point to next source pixel
        inc     di                      ;point to next dest pixel
        dec     cx
        jnz     ColumnLoop
        add     di,[bp+BufferWidth2]    ;point to next scan to fill
        dec     dx                      ;count down rows to fill
        jnz     RowLoop2

        pop     di
        pop     si
        pop     bp
        ret
_DrawMasked     endp
```

```
; Copies a rectangle from one buffer to another. C-callable as:
;     void CopyRect(DestBufferPtr, SrcBufferPtr, CopyHeight, CopyWidth,
;                   DestBufferWidth, SrcBufferWidth);

parms3  struc
                dw      ?       ;pushed BP
                dw      ?       ;pushed return address
DestBufferPtr   dd      ?       ;far pointer to buffer to which to copy
SrcBufferPtr    dd      ?       ;far pointer to buffer from which to copy
CopyHeight      dw      ?       ;height of rect to copy
CopyWidth       dw      ?       ;width of rect to copy
DestBufferWidth dw      ?       ;width of buffer to which to copy
SrcBufferWidth  dw      ?       ;width of buffer from which to copy
parms3  ends
        public  _CopyRect
_CopyRect       proc    near
        cld
        push    bp
        mov     bp,sp
        push    si
        push    di
        push    ds

        les     di,[bp+DestBufferPtr]
        lds     si,[bp+SrcBufferPtr]
        mov     dx,[bp+CopyHeight]
        mov     bx,[bp+DestBufferWidth] ;distance from end of one dest scan
        sub     bx,[bp+CopyWidth]       ; of copy to the next
        mov     ax,[bp+SrcBufferWidth]  ;distance from end of one source scan
        sub     ax,[bp+CopyWidth]       ; of copy to the next
RowLoop3:
        mov     cx,[bp+CopyWidth]       ;# of bytes to copy
        shr     cx,1
        rep     movsw                   ;copy as many words as possible
        adc     cx,cx
        rep     movsb                   ;copy odd byte, if any
        add     si,ax                   ;point to next source scan line
        add     di,bx                   ;point to next dest scan line
        dec     dx                      ;count down rows to fill
        jnz     RowLoop3

        pop     ds
        pop     di
        pop     si
        pop     bp
        ret
_CopyRect       endp
        end
```

This document was converted to PostScript™ by D. Bragin (s570959@tfh-berlin.de) Dec 20, 1994
NOT FOR SALE