Chapter 57

10,000 Freshly Sheared Sheep on the Screen

Chapter 57

The Critical Role of Experience in Implementing Fast, Smooth Texture Mapping

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 a p proved 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 10 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.

Visual Quality: A Black Hole ... Er, Art

Pleasing the eye with realtime computer animation is something less than a science, at least at the PC level, where there's a limited color palette and no time for antialiasing; in fact, sometimes it can be more than a little frustrating. As you may recall, in the previous chapter I implemented texture mapping in X-Sharp. There was plenty of experience involved there, some of which I didn't mention. 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 rotation; 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 simply 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 animation 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

In the previous chapter I added texture mapping to X-Sharp, but lacked space to explain some of its finer points. I'll pick up the thread now and cover some of those points here, and discuss the visual and performance enhancements that previous chapter's code needed—and which are now present in the version of X-Sharp in this chapter's subdirectory on the CD-ROM.

Back in Chapter 38, 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 result in imprecise pixel placement.

More than a year then passed between the time I wrote that statement and the time I implemented X-Sharp's texture mapper, during which time my long-term memory apparently suffered at least partial failure. When I went to implement texture mapping for the previous chapter, 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 be 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 not precise. 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, 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 fixed-point and all-integer polygon-filling approaches when drawing, as shown in Figure 57.1. Consequently, I ended up using an all-integer approach in X-Sharp for stepping through the destination polygon. However, I kept the fixed-point approach, which is faster and much simpler, for stepping through the source.

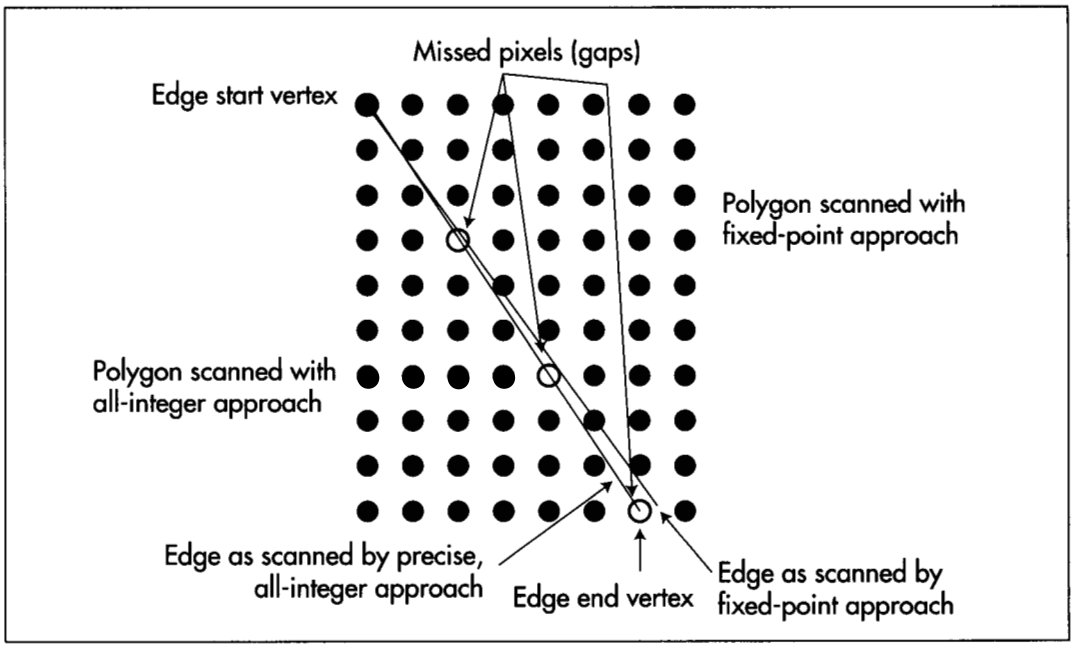
Why was it all right to mix approaches in this case? Precise pixel placement only matters when drawing; otherwise, we can get gaps, which are very visible. When selecting a pixel to copy from the source texture, however, the worst that happens is that we pick the source pixel next to the one we really 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: It's the difference between 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

The double-DDA texture-mapping code presented in the previous chapter worked adequately, but there were two things about it that left me less than satisfied. One flaw was performance; I'll address that shortly. The other flaw was the way textures shifted noticeably as the orientations of the polygons onto which they were mapped changed.

The previous chapter's code followed the standard polygon inside/outside rule for determining which pixels in the source texture map were to be mapped: Pixels that



Gaps caused by mixing fixed-point and all-integer math.
Figure 57.1

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 fine 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, the previous chapter's code truncated to get integer source coordinates. This, together with the orientation problem, meant that when a texture turned upside down, it slowed one new row and one new 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 57.1 is one solution to these problems. This code, which replaces the equivalently named function presented in the previous chapter (and, of course, is present in the X-Sharp archive in this chapter's subdirectory of the listings disk), 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.

LISTING 57.1 L57-1.C

```
/* Texture-map-draw the scan line between two edges. 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. */

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;
   }
}
```

Mapping Textures across Multiple Polygons

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 DEMO1 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 spaceships, or even human figures). Just map the desired texture onto the underlying polygonal framework of an object, and let double-DDA 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 what I presented in the previous chapter.

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 57.1.

Listing 57.2 is a high-performance assembly language implementation of Listing 57.1. Apart from the conversion to assembly language, this implementation improves performance by focusing on reducing inner loop bottlenecks. In fact, the whole of Listing 57.2 is nothing more than the inner loop for texture-mapped polygon drawing; Listing 57.2 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.

LISTING 57.2 L57-2.ASM

```
; 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
```

Within Listing 57.2, 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 57.1. Function calls are expensive operations, to be avoided when performance matters. Also, although Mode X (the undocumented 320×240 256-color 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 0. 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 0, 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. (Check out the next chapter, however, to see the best Map Mask optimization of all—setting it once and leaving it unchanged.)

In all, the overhead of drawing each pixel is reduced from a call to the set-pixel 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 57.2, speeding up the retrieval of pixels from the texture map. This operation looks very efficient in Listing 57.1, 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 57.2; ten instructions, with a maximum of two branches, replace all the texture calculations of Listing 57.1. Listing 57.2 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 57.2 much more complicated than Listing 57.1. Is it worth the trouble? Indeed it is. Listing 57.2 is more than twice as fast as Listing 57.1, 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?

I'm always interested in getting your feedback on and hearing about potential improvements to X-Sharp. Contact me through the publisher. There is no truth to the rumor that I can be reached under the alias "sheep-shearer," at least not for another 9,999 sheep.