Chapter 54

3-D Shading

Chapter 54

Putting Realistic Surfaces on Animated 3-D Objects

At the end of the previous chapter, X-Sharp had just acquired basic hidden-surface capability, and performance had been vastly improved through the use of fixed-point arithmetic. In this chapter, 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 chapter (actually, it'll spill over into the next chapter), so let's get to it!

Support for Older Processors

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 54.1 is an improved version of FIXED.ASM, containing dual 386/8088 versions of **CosSin()**, **XformVec()**, and **ConcatXforms()**, as well as **FixedMul()** and **FixedDiv()**.

Given the new version of FIXED.ASM, with **USE386** set to 0, X-Sharp will now 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 between the low end and mainstream. 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 54.1 I've chosen not to implement a full 32×32 divide, but rather only a 32×16 divide. The reason is simple: performance. A 32×16 divide can be implemented on an 8088 with two **DIV** instructions, but a 32×32 divide takes a great deal more work, so far as I can see. (If anyone has a fast 32×32 divide, or has a faster way to handle signed multiplies and divides than the approach taken by Listing 54.1, please drop me a line care of the publisher.) 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 32×16 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 (that is, it 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()** is *not* a general-purpose 32×32 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 **Fixed-Div()** is designed to do just what X-Sharp needs, and no more, as quickly as possible.

LISTING 54.1 FIXED.ASM

```
; Fixed point routines.
; Tested with TASM

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, however)
                                     ;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

;------------------------------------------------------------------------
; 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

if USE386

     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

else ;!USE386

    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   ax,word ptr CosTable[bx]          ;look up sine
    mov   dx,word ptr CosTable[bx+2]
    neg   bx                                ;sin(Angle) = cos(90-Angle)
    mov   cx,word ptr CosTable[bx+90*10*4+2] ;look up cosine
    mov   bx,word ptr CosTable[bx+90*10*4]
    jmp   CSDone

    align ALIGNMENT
Quadrant1:
    neg   bx
    add   bx,180*10                         ;convert to angle between 0 and 90
    shl   bx,2
    mov   ax,word ptr CosTable[bx]          ;look up cosine
    mov   dx,word ptr CosTable[bx+2]
    neg   dx                                ;negative in this quadrant
    neg   ax
    sbb   dx,0
    neg   bx                                ;sin(Angle) = cos(90-Angle)
    mov   cx,word ptr CosTable[bx+90*10*4+2] ;look up cosine
    mov   bx,word ptr CosTable[bx+90*10*4]
    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     ax,word ptr CosTable[bx]      ;look up cosine
        mov     dx,word ptr CosTable[bx+2]
        neg     bx                            ;sin(Angle) = cos(90-Angle)
        mov     cx,word ptr CosTable[90*10*4+bx+2] ;look up sine
        mov     bx,word ptr CosTable[90*10*4+bx]
        neg     cx                            ;negative in this quadrant
        neg     bx
        sbb     cx,0
        jmp     short CSDone

        align ALIGNMENT
Quadrant2:
        neg     bx
        add     bx,180*10                     ;convert to angle between 0 and 90
        shl     bx,2
        mov     ax,word ptr CosTable[bx]      ;look up cosine
        mov     dx,word ptr CosTable[bx+2]
        neg     dx                            ;negative in this quadrant
        neg     ax
        sbb     dx,0
        neg     bx                            ;sin(Angle) = cos(90-Angle)
        mov     cx,word ptr CosTable[90*10*4+bx+2] ;look up sine
        mov     bx,word ptr CosTable[90*10*4+bx]
        neg     cx                            ;negative in this quadrant
        neg     bx
        sbb     cx,0
CSDone:
        push    bx
        mov     bx,[bp].Cos
        mov     [bx],ax
        mov     [bx+2],dx
        mov     bx,[bp].Sin
        pop     ax
        mov     [bx],ax
        mov     [bx+2],cx

endif ;USE386

        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

; Macro for non-386 multiply. AX, BX, CX, DX destroyed.
FIXED_MUL   MACRO M1,M2
      local CheckSecondOperand,SaveSignStatus,FixedMulDone

                                        ;do four partial products and
                                        ; add them together, accumulating
                                        ; the result in CX:BX
                                        ;figure out signs, so we can use
                                        ; unsigned multiplies
      sub   cx,cx                       ;assume both operands positive
      mov   bx,word ptr [&M1&+2]
      and   bx,bx                       ;first operand negative?
      jns   CheckSecondOperand          ;no
      neg   bx                          ;yes, so negate first operand
      neg   word ptr [&M1&]
      sbb   bx,0
      mov   word ptr [&M1&+2],bx
      inc   cx                          ;mark that first operand is negative
CheckSecondOperand:
      mov   bx,word ptr [&M2&+2]
      and   bx,bx                       ;second operand negative?
      jns   SaveSignStatus              ;no
      neg   bx                          ;yes, so negate second operand
      neg   word ptr [&M2&]
      sbb   bx,0
      mov   word ptr [&M2&+2],bx
      xor   cx,1                        ;mark that second operand is negative
SaveSignStatus:
      push  cx                          ;remember sign of result; 1 if result
                                        ; negative, 0 if result nonnegative
      mov   ax,word ptr [&M1&+2]        ;high word times high word
      mul   word ptr [&M2&+2]
      mov   cx,ax       ;
                                        ;assume no overflow into DX
      mov   ax,word ptr [&M1&+2]        ;high word times low word
      mul   word ptr [&M2&]
      mov   bx,ax
      add   cx,dx
      mov   ax,word ptr [&M1&]          ;low word times high word
      mul   word ptr [&M2&+2]
      add   bx,ax
      adc   cx,dx
      mov   ax,word ptr [&M1&]          ;low word times low word
      mul   word ptr [&M2&]
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
      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:
      ENDM

      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

if USE386

      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 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
      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 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
      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 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
      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

else ;!USE386

     mov   si,[bp].WorkingXform          ;SI points to xform matrix
     mov   di,[bp].SourceVec             ;DI points to source vector
     mov   bx,[bp].DestVec               ;BX points to dest vector
     push  bp                            ;preserve stack frame pointer

soff=0
doff=0
     REPT 3                              ;do once each for dest X, Y, and Z
     push  bx                            ;remember dest vector pointer
     push  word ptr [si+soff+2]
     push  word ptr [si+soff]
     push  word ptr [di+2]
     push  word ptr [di]
     call  _FixedMul                     ;xform entry times source X entry
     add   sp,8                          ;clear parameters from stack
     mov   cx,ax ;set running total
     mov   bp,dx

     push  cx                            ;preserve low word of running total
     push  word ptr [si+soff+4+2]
     push  word ptr [si+soff+4]
     push  word ptr [di+4+2]
     push  word ptr [di+4]
     call  _FixedMul                     ;xform entry times source Y entry
     add   sp,8                          ;clear parameters from stack
     pop   cx                            ;restore low word of running total
     add   cx,ax                         ;running total for this row
     adc   bp,dx

     push  cx                            ;preserve low word of running total
     push  word ptr [si+soff+8+2]
     push  word ptr [si+soff+8]
     push  word ptr [di+8+2]
     push  word ptr [di+8]
     call  _FixedMul                     ;xform entry times source Z entry
     add   sp,8                          ;clear parameters from stack
     pop   cx                            ;restore low word of running total
     add   cx,ax                         ;running total for this row
     adc   bp,dx

     add   cx,[si+soff+12]               ;add in translation
     adc   bp,[si+soff+12+2]
     pop   bx                            ;restore dest vector pointer
     mov   [bx+doff],cx                  ;save the result in the dest vector
     mov   [bx+doff+2],bp
soff=soff+16
doff=doff+4
     ENDM

     pop   bp                            ;restore stack frame pointer

endif ;USE386

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

if USE386

           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 3                              ;once for each of the first 3 columns,
                                               ; assuming 0 as the bottom entry (no
                                               ; translation)
           mov     eax,[si+roff]               ;column 0 entry on this row
           imul    dword ptr [bx+coff]         ;times row 0 entry in column
```

```
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
        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 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
        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 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
        shrd    eax,edx,16                  ;shift the result back to 16.16 form
        add     ecx,eax                     ;running total

        mov     [di+coff+roff],ecx          ;save the result in dest matrix
coff=coff+4                                 ;point to next col in xform2 & dest
        ENDM

                                            ;now do the fourth column, assuming
                                            ; 1 as the bottom entry, causing
                                            ; translation to be performed
        mov     eax,[si+roff]               ;column 0 entry on this row
        imul    dword ptr [bx+coff]         ;times row 0 entry in column
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
        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 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
        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 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
        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

roff=roff+16                                    ;point to next col in xform2 & dest
        ENDM

else ;!USE386

        mov     di,[bp].SourceXform2            ;DI points to xform2 matrix
        mov     si,[bp].SourceXform1            ;SI points to xform1 matrix
        mov     bx,[bp].DestXform               ;BX points to dest xform matrix
        push    bp                              ;preserve stack frame pointer

roff=0                                          ;row offset
        REPT 3                                  ;once for each row
coff=0                                          ;column offset
        REPT 3                                  ;once for each of the first 3 columns,
                                                ; assuming 0 as the bottom entry (no
                                                ; translation)
        push    bx                              ;remember dest vector pointer
        push    word ptr [si+roff+2]
        push    word ptr [si+roff]
        push    word ptr [di+coff+2]
        push    word ptr [di+coff]
        call    _FixedMul                       ;column 0 entry on this row times row 0
                                                ; entry in column
        add     sp,8                            ;clear parameters from stack
        mov     cx,ax   ;set running total
        mov     bp,dx

        push    cx                              ;preserve low word of running total
        push    word ptr [si+roff+4+2]
        push    word ptr [si+roff+4]
        push    word ptr [di+coff+16+2]
        push    word ptr [di+coff+16]
        call    _FixedMul                       ;column 1 entry on this row times row 1
                                                ; entry in column
        add     sp,8                            ;clear parameters from stack
        pop     cx                              ;restore low word of running total
        add     cx,ax                           ;running total for this row
        adc     bp,dx

        push    cx                              ;preserve low word of running total
        push    word ptr [si+roff+8+2]
        push    word ptr [si+roff+8]
        push    word ptr [di+coff+32+2]
        push    word ptr [di+coff+32]
        call    _FixedMul                       ;column 1 entry on this row times row 1
                                                ; entry in column
        add     sp,8                            ;clear parameters from stack
        pop     cx                              ;restore low word of running total
        add     cx,ax                           ;running total for this row
        adc     bp,dx

        pop     bx                              ;restore DestXForm pointer
        mov     [bx+coff+roff],cx               ;save the result in dest matrix
        mov     [bx+coff+roff+2],bp
```

```
coff=coff+4                                          ;point to next col in xform2 & dest
         ENDM
                                                     ;now do the fourth column, assuming
                                                     ; 1 as the bottom entry, causing
                                                     ; translation to be performed
         push     bx                                 ;remember dest vector pointer
         push     word ptr [si+roff+2]
         push     word ptr [si+roff]
         push     word ptr [di+coff+2]
         push     word ptr [di+coff]
         call     _FixedMul                          ;column 0 entry on this row times row 0
                                                     ; entry in column
         add      sp,8                               ;clear parameters from stack
         mov      cx,ax    ;set running total
         mov      bp,dx

         push     cx                                 ;preserve low word of running total
         push     word ptr [si+roff+4+2]
         push     word ptr [si+roff+4]
         push     word ptr [di+coff+16+2]
         push     word ptr [di+coff+16]
         call     _FixedMul                          ;column 1 entry on this row times row 1
                                                     ; entry in column
         add      sp,8                               ;clear parameters from stack
         pop      cx                                 ;restore low word of running total
         add      cx,ax                              ;running total for this row
         adc      bp,dx

         push     cx                                 ;preserve low word of running total
         push     word ptr [si+roff+8+2]
         push     word ptr [si+roff+8]
         push     word ptr [di+coff+32+2]
         push     word ptr [di+coff+32]
         call     _FixedMul                          ;column 1 entry on this row times row 1
                                                     ; entry in column
         add      sp,8                               ;clear parameters from stack
         pop      cx                                 ;restore low word of running total
         add      cx,ax                              ;running total for this row
         adc      bp,dx

         add      cx,[si+roff+12]                    ;add in translation
         add      bp,[si+roff+12+2]

         pop      bx                                 ;restore DestXForm pointer
         mov      [bx+coff+roff],cx                  ;save the result in dest matrix
         mov      [bx+coff+roff+2],bp
coff=coff+4                                          ;point to next col in xform2 & dest

roff=roff+16                                         ;point to next col in xform2 & dest
         ENDM

         pop      bp                                 ;restore stack frame pointer

endif ;USE386

         pop      di                                 ;restore register variables
         pop      si
         pop      bp                                 ;restore stack frame
         ret
_ConcatXforms     endp
         end
```

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 mirror-like 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 in X-Sharp.

Ambient Shading

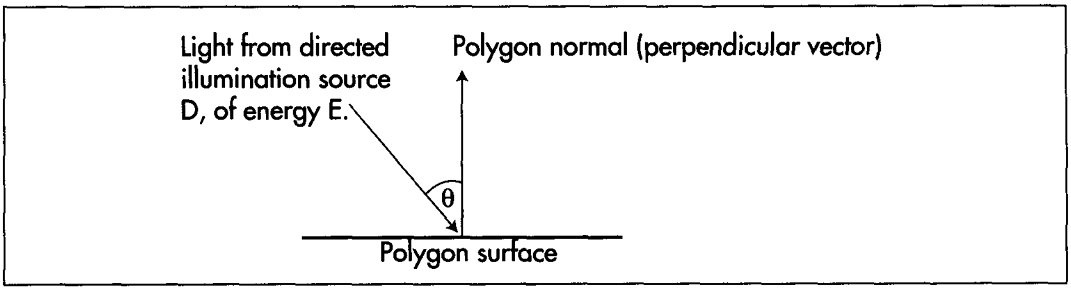
The basic model for both ambient and diffuse shading is a simple one. Each surface has a reflectivity between 0 and 1, where 0 means all light is absorbed and 1 means all light is reflected. A certain amount of light energy strikes each surface. The energy (intensity) of the light is expressed such that 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_{red} and a surface red reflectance R_{red}, the displayed red ambient shading for that surface, as a fraction of the maximum red intensity, is simply $\min(IA_{red} \times R_{red}, 1)$. 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; I'll do that in the next chapter. Ambient shading isn't the whole shading picture, though. In fact, scenes tend to look pretty bland without diffuse shading.

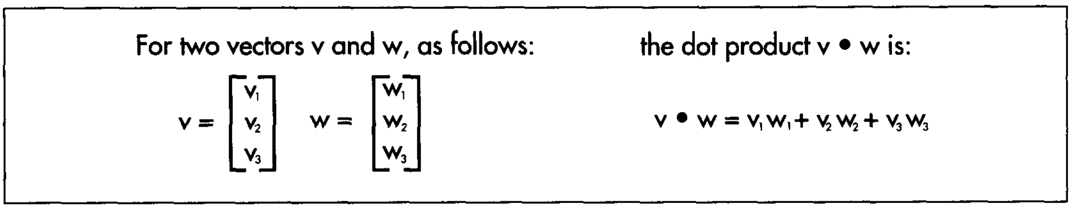
Diffuse Shading

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 54.1. If the red intensity of directed light is ID_{red}, the red reflectance of the surface is R_{red}, and the angle between the incoming directed light and the surface's normal is theta, then the displayed red diffuse shading for that surface, as a fraction of the largest possible red intensity, is min $(ID_{red} \times R_{red} \times \cos(\theta), 1)$.

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 "•", as shown in Figure 54.2), $\cos(\theta)=(v \bullet w) / |v| \times |w|$), where v and w are vectors, θ 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 $\cos(\theta)=v \bullet w$. 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 multiplies and two adds. (I'll explain why the light-direction vector must be reversed later.) Once we have that, we can easily calculate the red



Illumination by a directed light source.
Figure 54.1



The dot product of two vectors.
Figure 54.2

diffuse shading from a directed light source as $\min(ID_{red} \times R_{red} \times (L' \bullet N), 1)$ 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 $\min((IA_{red} \times R_{red}) + (ID_{red0} \times R_{red} \times (L_0' \bullet N)) + (ID_{red1} \times R_{red} \times (L_1' \bullet N)) + \ldots, 1)$ where ID_{red0} and L_0' are the red intensity and the reversed unit-direction vector, respectively, for spotlight 0. Listing 54.2 shows the X-Sharp module DRAWPOBJ.C, which performs ambient and diffuse shading. Toward the end, 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.

LISTING 54.2 DRAWPOBJ.C

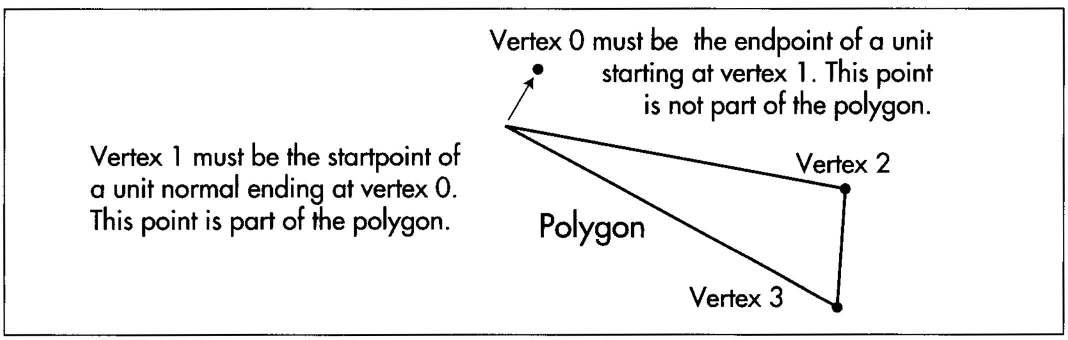
```
/* Draws all visible faces in the specified polygon-based object. The object
   must have previously been transformed and projected, so that all vertex
   arrays are filled in. Ambient and diffuse shading are supported. */
#include "polygon.h"

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 */
```

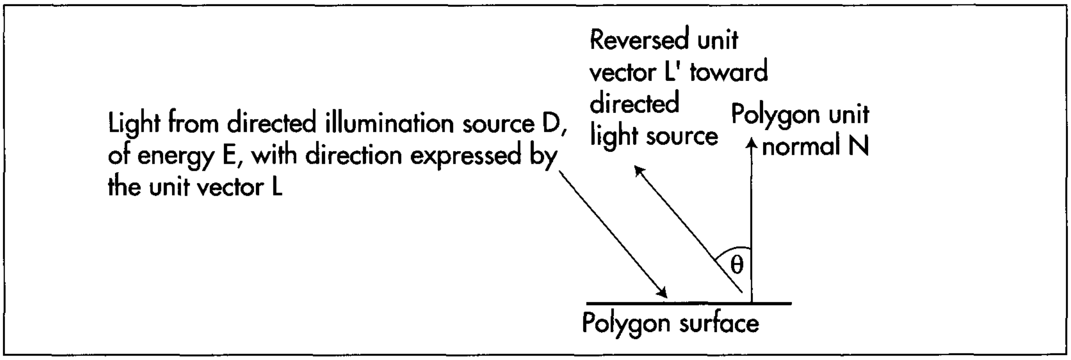
```
                if ((Diffusion = DOT_PRODUCT(SpotDirectionView[Spot],
                      UnitNormal)) > 0) {
                   IntensityTemp.Red +=
                         FixedMul(SpotIntensity[Spot].Red, Diffusion);
                   IntensityTemp.Green +=
                         FixedMul(SpotIntensity[Spot].Green, 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);
    }
  }
 }
}
```

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 54.3. The first point isn't one of the polygon's vertices, but is used only to generate a unit normal. The



The unit normal in the polygon data structure.
Figure 54.3



The reversed light source vector.
Figure 54.4

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 end-points 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(theta) 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 54.4. 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 54.2. The sample program DEMO1, in the X-Sharp archive on the listings disk (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. In the next chapter, I'll talk about some ways to improve the quality of 256-color shading.