Raw Speed and More
The Naked Truth About Speed in 3-D Animation

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 teenagers 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 high school 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 stream-
ing 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 chapter’s topics: raw speed and hidden surfaces.

**Raw Speed, Part 1: 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 four chapters into development of the X-Sharp 3-D animation package. In realtime 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. In the previous chapter, we got up to a serviceable performance level by switching to fixed-point math, then implementing the fixed-point multiplication and division functions in assembly 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 53.1 is the module *FIXED.ASM* from this chapter’s iteration 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 altogether. *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.

**LISTING 53.1 FIXED.ASM**

; 386-specific fixed point routines.
; Tested with TASM

```assembly
ROUNDING_ON       equ 1  ; 1 for rounding, 0 for no rounding
ALIGNMENT         equ 2  ; no rounding is faster, rounding is more accurate
.model              small
.386               
.code
```

992 Chapter 53
; Multiplies two fixed-point values together.
; C near-callable as:
; Fixedpoint FixedMul(Fixedpoint M1, Fixedpoint M2);
; Fixedpoint FixedDiv(Fixedpoint Dividend, Fixedpoint Divisor);

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
    mov eax,[bp+M1]
    imul dword ptr [bp+M2] ;multiply
    if ROUNGING-ON
        add eax,8000h ;round by adding 2^-(-17)
        adc edx,0 ;whole part of result is in DX
    endif ;ROUNGING-ON
    shr eax,16 ;put the fractional part in AX
    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 ROUNGING-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
        rol eax,16 ;left 16 bits so that result will be
        mov dx,ax ;in EAX
        mov ax,[bp+Divisor]
        sub ax,ax ;clear low word of EAX
        mov ebx,dword ptr [bp+Divisor]
        and ebx,eax ;positive divisor?
        jns FDP2 ;yes
        dec cx ;mark it's a negative divisor
        neg ebx ;make divisor positive

    endif ;ROUNGING-ON
    push bp
    mov bp,sp
    ret

_FixedDiv endp

Raw Speed and More 993
FDP2: div ebx  ;divide
shr ebx,1  ;divisor/2, minus 1 if the divisor is
adc ebx,0 ; even
dec ebx
cmp ebx, edx
adc eax,0
and cx,cx
jz FDP3
neg eax

FDP3:
else : !ROUNDING_ON
mov edx,[bp+Dividend]
sub eax,eax
shrd eax,edx,16
sar edx, 16
idiv dword ptr [bp+Divisor]
define !ROUNDING_ON
shld edx,eax,16
endif
pop bp
ret

_FixedDiv endp

; Returns the sine and cosine of an angle.
; C near-callable as:
;   void CosSin(TAngle Angle, Fixedpoint *Cos, Fixedpoint *Sin);
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
jns CheckInRange

MakePos:
add bx,360*10
js MakePos
jmp short CheckInRange

MakeInRange:
sub bx,360*10
CheckInRange:
  cmp bx,360*10
  jg MakeInRange

994 Chapter 53
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:

```c
void XformVec(Xform WorkingXform, Fixedpoint *SourceVec,
               Fixedpoint *DestVec);
```

This assembly code is equivalent to this C code:

```c
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]
    imul dword ptr [bx] ;column 0 entry on this row
if ROUNDDNG_ON
    add eax,8000h ;xform entry times source X entry
    adc edx,0 ;round by adding 2^(-17)
endif ;ROUNDDNG_ON
    shrd eax,edx,16 ;shift the result back to 16.16 form
    mov ecx,eax
    mov eax,[si+soff+4] ;column 1 entry on this row
    imul dword ptr [bx+4] ;xform entry times source Y entry
if ROUNDDNG_ON
    add eax,8000h ;round by adding 2^(-17)
    adc edx,0 ;whole part of result is in DX
endif ;ROUNDDNG_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 ROUNDDNG_ON
    add eax,8000h ;round by adding 2^(-17)
    adc edx,0 ;whole part of result is in DX
```

996 Chapter 53
endif :ROUNDING_ON
    shr edx, 16
    mov [di+doff], ecx
    mov [di+doff], ecx
    add ecx, edx
    add ecx, [si+soff+12]
    add ecx, eax
    mov [di+doff], ecx
    add ecx, [si+soff+16]
    doff - doff + 4
    ENDM

:shift the result back to 16.16 form
:running total for this row
:add in translation
:save the result in the dest vector

pop di ;restore register variables
pop si
pop bp ;restore stack frame
ret
_XformVec endp

: Matrix multiplies SourceXforml by SourceXform2 and stores the
: result in DestXform. Multiplies a 4x4 matrix times a 4x4 matrix;
: each entry is a 16.16 form. 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 SourceXforml, 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(SourceXforml[i][0], SourceXform2[0][j]) +
:       FixedMul(SourceXforml[i][1], SourceXform2[1][j]) +
:       FixedMul(SourceXforml[i][2], SourceXform2[2][j]);
:   DestXform[i][3] =
:     FixedMul(SourceXforml[i][0], SourceXform2[0][3]) +
:     FixedMul(SourceXforml[i][1], SourceXform2[1][3]) +
:     FixedMul(SourceXforml[i][2], SourceXform2[2][3]) +
:     SourceXforml[i][3];
: }

CXparms struc
    dw 2 dup(?) ;return address & pushed BP
    dw ? ;pointer to first source xform matrix
    dw ? ;pointer to second source xform matrix
    dw ? ;pointer to destination xform matrix
CXparms ends

_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].SourceXforml ;SI points to xform1 matrix
    mov di,[bp].DestXform ;DI points to dest xform matrix

Raw Speed and More 997
roff=0
coff=0
REPT 3
REPT 3

mov eax,[si+roff]
imul dword ptr [bx+coff]
if ROUNGING_ON
add eax,8000h
adc edx,0
eendif :ROUNGING_ON
shrd eax,edx,16
mov ecx,eax

mov eax,[si+roff+4]
imul dword ptr [bx+coff+16]
if ROUNGING_ON
add eax,8000h
adc edx,0
eendif :ROUNGING_ON
shrd eax,edx,16
add ecx,eax

mov eax,[si+roff+8]
imul dword ptr [bx+coff+32]
if ROUNGING_ON
add eax,8000h
adc edx,0
eendif :ROUNGING_ON
shrd eax,edx,16
add ecx,eax

mov [di+coff+roff],ecx
coff=coff+4
ENDM

mov eax,[si+roff]
imul dword ptr [bx+coff]
if ROUNGING_ON
add eax,8000h
adc edx,0
eendif :ROUNGING_ON
shrd eax,edx,16
mov ecx,eax

mov eax,[si+roff+4]
imul dword ptr [bx+coff+16]
if ROUNGING_ON
add eax,8000h
adc edx,0
eendif :ROUNGING_ON
shrd eax,edx,16
add ecx,eax

mov eax,[si+roff+8]
imul dword ptr [bx+coff+32]

;row offset
;once for each row
;column offset
;once for each of the first 3 columns.
;assuming 0 as the bottom entry (no
;translation)
;column 0 entry on this row
;times row 0 entry in column

;round by adding 2^(-17)
;whole part of result is in DX

;shift the result back to 16.16 form
;set running total

;column 1 entry on this row
;times row 1 entry in col

;round by adding 2^-17
;whole part of result is in DX

;shift the result back to 16.16 form
;running total

;column 2 entry on this row
;times row 2 entry in col

;round by adding 2^(-17)
;whole part of result is in DX

;shift the result back to 16.16 form
;running total

;save the result in dest matrix
;point to next col in xform2 & dest

;now do the fourth column, assuming
;1 as the bottom entry, causing
;translation to be performed
;column 0 entry on this row
;times row 0 entry in column

;round by adding 2^(-17)
;whole part of result is in DX

;shift the result back to 16.16 form
;set running total

;column 1 entry on this row
;times row 1 entry in col

;round by adding 2^(-17)
;whole part of result is in DX

;shift the result back to 16.16 form
;running total

;column 2 entry on this row
;times row 2 entry in col

998  Chapter 53
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 cosine weren't major drags on performance, because the \texttt{sin()} and \texttt{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 \texttt{sin()} and \texttt{cos()} major drags, they were taking up over half the total execution time by themselves.

The solution is a lookup table. Listing 53.1 contains a function called \texttt{CosSin()} that calculates both the sine and cosine of an angle, via a lookup table. The function accepts angles 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 \texttt{CosSin()} for finer gradations yet. \texttt{GENCOS.C}, the program used to generate the lookup table (\texttt{COSTABLE.INC}), included in Listing 53.1, can be found in the XSHARP22 subdirectory on the listings diskette. \texttt{GENCOS.C} can generate a cosine table with any integral number of steps per degree.

\texttt{FIXED.ASM} (Listing 53.1) 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 assembly 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. What’s more, right now, objects can never pass behind or in front of each other. What that means is that 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; in this chapter, we’ll simply sort the objects on Z and leave it at that.)

This limited version of 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 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 53.1 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 53.2 shows X-Sharp file OLIST.C, which includes the key routines for depth sorting. 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 will proceed from back to front. The Z-sorting is done on the basis of the objects’ center points; a
**Why back-to-front sorting doesn't always work properly.**

**Figure 53.1**

center-point field 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.

**LISTING 53.2  OLIST.C**

```c
/* 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 = 0x7FFFFFFF;
    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++;

/* Sorts 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->NextObject = ObjectCmpPtr;
            ObjectPtr->PreviousObject = 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;
        }
    }
}

Rounding

FIXED.ASM contains the equate ROUNDDING_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, $2^{16}$; 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, the errors will eventually creep in, and reorthogonalization will become necessary at some point.

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

So far in our exploration of 3-D animation, 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 Colors array in INITBALL.C to match, but that’s a tiny detail; by and large, the process of generating a ball-shaped 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 Colors 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 someday have an interactive 3-D object editing tool and resource management setup. 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.