Chapter 63

Floating-Point for Real-Time 3-D

Chapter 63

Knowing When to Hurl Conventional Math Wisdom Out the Window

In a crisis, sometimes it's best to go with the first solution that comes into your head—but not very often.

When I turned 16, my mother had an aging, three-cylinder Saab—not one of the sporty Saabs that appeared in the late '70s, but a blunt-nosed, ungainly little wagon that seated up to seven people in sardine-like comfort, with two of them perched on the gas tank. That was the car I learned to drive on, and the one I took whenever I wanted to go somewhere and my mother didn't need it.

My father's car, on the other hand, was a Volvo sedan, only a couple of years old and easily the classiest car my family had ever owned. To the best of my recollection, as of New Year's of my senior year, I had never driven that car. However, I was going to a New Year's party—in fact, I was going to chauffeur four other people—and for reasons lost in the mists of time, I was allowed to take the Volvo. So, one crystal clear, stunningly cold night, I picked up my passengers, who included Robin Viola, Kathy Smith, Jude Hawron...and Alan, whose last name I'll omit in case he wants to run for president someday.

The party was at Craig Alexander's house, way out in the middle of nowhere, and it was a good one. I heard Al Green for the first time, much beer was consumed (none by me, though), and around 2 a.m., we decided it was time to head home. So we piled into the Volvo, cranked the heat up to the max, and set off.

We had gone about five miles when I sensed Alan was trying to tell me something. As I turned toward him, he said, quite expressively, "BLEARGH!" and deposited a considerable volume of what had until recently been beer and chips into his lap.

Mind you, this wasn't just any car Alan was tossing his cookies in—it was my father's prized Volvo. My reactions were up to the task; without a moment's hesitation, I shouted, "Do it out the window! Open the window!" Alan obligingly rolled the window down and, with flawless aim, sent some more erstwhile beer and chips on its way.

And it was here that I learned that fast decisions are not necessarily good decisions. A second after the liquid flew out the window, there was a loud smacking sound, and a yelp from Robin, as the sodden mass hit the slipstream and splattered along the length of the car. At that point, I did what I should have done in the first place; I stopped the car so Alan could get out and finish being sick in peace, while I assessed the full dimensions of the disaster. Not only was the rear half of the car on the passenger side—including Robin's window, accounting for the yelp—covered, but the noxious substance had frozen solid. It looked like someone had melted an enormous candle, or possibly put cake frosting on the car.

The next morning, my father was remarkably good-natured about the whole thing, considering, although I don't remember ever actually driving the Volvo again. My penance consisted of cleaning the car, no small punishment considering that I had to take a hair dryer out to our unheated garage and melt and clean the gunk one small piece at a time.

One thing I learned from this debacle is to pull over very, very quickly if anyone shows signed of being ill, a bit of wisdom that has proven useful a suprising number of times over the years. More important, though, is the lesson that it almost always pays to take at least a few seconds to size up a crisis situation and choose an effective response, and that's served me well more times than I can count.

There's a surprisingly close analog to this in programming. Often, when faced with a problem in his or her code, a programmer's response is to come up with a solution as quickly as possible and immediately hack it in. For all but the simplest problems, though, there are side effects and design issues involved that should be thought through before any coding is done. I try to think of bugs and other problem situations as opportunities to reexamine how my code works, as well as chances to detect and correct structural defects I hadn't previously suspected; in fact, I'm often able to simplify code as I fix a bug, thanks to the understanding I gain in the process.

Taking that a step farther, it's useful to reexamine assumptions periodically even if no bugs are involved. You might be surprised at how quickly assumptions that once were completely valid can deteriorate.

For example, consider floating-point math.

Not Your Father's Floating-Point

Until last year, I had never done any serious floating-point (FP) optimization, for the perfectly good reason that FP math had never been fast enough for any of the code I needed to write. It was an article of faith that FP, while undeniably convenient, because of its automatic support for constant precision over an enormous range of magnitudes, was just not fast enough for real-time programming, so I, like pretty much everyone else doing 3-D, expended a lot of time and effort in making fixed-point do the job.

That article of faith was true up through the 486, but all the old assumptions are out the window on the Pentium, for three reasons: faster FP instructions, a pipelined floating-point unit (FPU), and the magic of a parallel FXCH. Taken together, these mean that FP addition and subtraction are nearly as fast as integer operations, and FP multiplication and division have the potential to be much faster—all with the range and precision advantages of FP. Better yet, the FPU has its own set of eight registers, so the use of floating-point can help relieve pressure on the x86's integer registers, as well.

One effect of all this is that with the Pentium, floating-point on the x86 has gone from being irrelevant to real-time 3-D to being a key element. Quake uses FP all the way down into the inner loop of the span rasterizer, performing several FP operations every 16 pixels.

Floating-point has not only become important for real-time 3-D on the PC, but will soon become even more crucial. Hardware accelerators will take care of texture mapping and will increase feasible scene complexity, meaning the CPU will do less bit-twiddling and will have far more vertices to transform and project, and far more motion physics and line-of-sight calculations and the like as well.

By way of getting you started with floating-point for real-time 3-D, in this chapter I'll examine the basics of Pentium FP optimization, then look at how some key mathematical techniques for 3-D—dot product, cross product, transformation, and projection—can be accelerated.

Pentium Floating-Point Optimization

I'm going to assume you're already familiar with x86 FP code in general; for additional information, check out Intel's *Pentium Processor User's Manual* (order #241430-001; 1-800-548-4725), a book that you should have if you're doing Pentium programming of any sort. I'd also recommend taking a look around http://www.intel.com.

I'm going to focus on six core instructions in this section: FLD, FST, FADD, FSUB, FMUL, and FDIV. First, let's look at cycle times for these instructions. FLD takes 1 cycle; the value is pushed onto the FP stack and ready for use on the next cycle. FST takes 2 cycles, although when storing to memory, there's a potential extra cycle that can be lost, as I'll describe shortly.

FDIV is a painfully slow instruction, taking 39 cycles at full precision and 33 cycles at double precision, which is the default precision for Visual C++ 2.0. While FDIV executes, the FPU is occupied, and can't process subsequent FP instructions until FDIV finishes. However, during the cycles while FDIV is executing (with the exception of the one cycle during which FDIV starts), the integer unit can simultaneously execute instructions other than IMUL. (IMUL uses the FPU, and can only overlap with FDIV for a few cycles.) Since the integer unit can execute two instructions per cycle, this means it's possible to have three instructions, an FDIV and two integer instructions, executing at the same time. That's exactly what happens, for example, during the second cycle of this code:

```
FDIV ST(0),ST(1)
ADD  EAX,ECX
INC  EDX
```

There's an important limitation, though; if the instruction stream following the FDIV reaches a FP instruction (or an IMUL), then that instruction and all subsequent instructions, both integer and FP, must wait to execute until FDIV has finished.

When a FADD, FSUB, or FMUL instruction is executed, it is 3 cycles before the result can be used by another instruction. (There's an exception: If the instruction that attempts to use the result is an FST to memory, there's an extra cycle lost, so it's 4 cycles from the start of an arithmetic instruction until an FST of that value can begin, so

```
FMUL ST(0),ST(1)
FST  [temp]
```

takes 6 cycles in all.) Again, it's possible to execute integer-unit instructions during the 2 (or 3, for FST) cycles after one of these FP instructions starts. There's a more exciting possibility here, though: Given properly structured code, the FPU is capable of averaging 1 cycle per FADD, FSUB, or FMUL. The secret is pipelining.

Pipelining, Latency, and Throughput

The Pentium's FPU is the first pipelined x86 FPU. *Pipelining* means that the FPU is capable of starting an instruction every cycle, and can simultaneously handle several instructions in various stages of completion. Only certain x86 FP instructions allow another instruction to start on the next cycle, though: FADD, FSUB, and FMUL are pipelined, but FST and FDIV are not. (FLD executes in a single cycle, so pipelining is not an issue.) Thus, in the code sequence

```
FADD1
FSUB
FADD2
FMUL
```

$FADD_1$ can start on cycle N, FSUB can start on cycle N+1, $FADD_2$ can start on cycle N+2, and FMUL can start on cycle N+3. At the start of cycle N+3, the result of $FADD_1$

is available in the destination operand, because it's been 3 cycles since the instruction started; FSUB is starting the final cycle of calculation; $FADD_2$ is starting its second cycle, with one cycle yet to go after this; and FMUL is about to be issued. Each of the instructions takes 3 cycles to produce a result from the time it starts, but because they're simultaneously processed at different pipeline stages, one instruction is issued and one instruction completes every cycle. Thus, the latency of these instructions—that is, the time until the result is available—is 3 cycles, but the throughput—the rate at which the FPU can start new instructions—is 1 cycle. An exception is that the FPU is capable of starting an FMUL only every 2 cycles, so between these two instructions

```
FMUL ST(1),ST(0)
FMUL ST(2),ST(0)
```

there's a 1-cycle stall, and the following three instructions execute just as fast as the above pair:

```
FMUL ST(1),ST(0)
FLD  ST(4)
FMUL ST(0),ST(1)
```

There's a caveat here, though: A FP instruction can't be issued until its operands are available. The FPU can reach a throughput of 1 cycle per instruction on this code

```
FADD ST(1),ST(0)
FLD  [temp]
FSUB ST(1),ST(0)
```

because neither the FLD nor the FSUB needs the result from the FADD. Consider, however

```
FADD ST(0),ST(2)
FSUB ST(0),ST(1)
```

where the ST(0) operand to FSUB is calculated by FADD. Here, FSUB can't start until FADD has completed, so there are 2 stall cycles between the two instructions. When dependencies like this occur, the FPU runs at latency rather than throughput speeds, and performance can drop by as much as two-thirds.

FXCH

One piece of the puzzle is still missing. Clearly, to get maximum throughput, we need to interleave FP instructions, such that at any one time ideally three instructions are in the pipeline at once. Further, these instructions must not depend on one another for operands. But ST(0) must always be one of the operands; worse, FLD can only push into ST(0), and FST can only store from ST(0). How, then, can we keep three independent instructions going?

The easy answer would be for Intel to change the FP registers from a stack to a set of independent registers. Since they couldn't do that, thanks to compatibility issues, they did the next best thing: They made the FXCH instruction, which swaps ST(0) and any other FP register, virtually free. In general, if FXCH is both preceded and followed by FP instructions, then it takes *no* cycles to execute. (Application Note 500, "Optimizations for Intel's 32-bit Processors," February 1994, available from http://www.intel.com, describes all the conditions under which FXCH is free.) This allows you to move the target of a pending operation from ST(0) to another register, at the same time bringing another register into ST(0) where it can be used, all at no cost. So, for example, we can start three multiplications, then use FXCH to swap back to start adding the results of the first two multiplications, without incurring any stalls, as shown in Listing 63.1.

LISTING 63.1 L63-1.ASM

```
; use of fxch to allow addition of first two; products to start while third
: multiplication finishes
        fld     [vec0+0]        ;starts & ends on cycle 0
        fmul    [vec1+0]        ;starts on cycle 1
        fld     [vec0+4]        ;starts & ends on cycle 2
        fmul    [vec1+4]        ;starts on cycle 3
        fld     [vec0+8]        ;starts & ends on cycle 4
        fmul    [vec1+8]        ;starts on cycle 5
        fxch    st(1)           ;no cost
        faddp   st(2),st(0)     ;starts on cycle 6
```

The Dot Product

Now we're ready to look at fast FP for common 3-D operations; we'll start by looking at how to speed up the dot product. As discussed in Chapter 30, the dot product is heavily used in 3-D to calculate cosines and to project points along vectors. The dot product is calculated as $d = u_1v_1 + u_2v_2 + u_3v_3$; with three loads, three multiplies, two adds, and a store, the theoretical minimum time for this calculation is 10 cycles.

Listing 63.2 shows a straightforward dot product implementation. This version loses 7 cycles to stalls. Listing 63.3 cuts the loss to 5 cycles by doing all three FMULs first, then using FXCH to set the third FXCH aside to complete while the results of the first two FMULs, which have completed, are added. Listing 43.3 still loses 50 percent to stalls, but unless some other code is available to be interleaved with the dot product code, that's all we can do to speed things up. Fortunately, dot products are often used in contexts where there's plenty of interleaving potential, as we'll see when we discuss transformation.

LISTING 63.2 1L63-2.ASM

```
; unoptimized dot product; 17 cycles
        fld     [vec0+0]    ;starts & ends on cycle 0
        fmul    [vec1+0]    ;starts on cycle 1
        fld     [vec0+4]    ;starts & ends on cycle 2
        fmul    [vec1+4]    ;starts on cycle 3
```

```
        fld     [vec0+8]    ;starts & ends on cycle 4
        fmul    [vec1+8]    ;starts on cycle 5
                            ;stalls for cycles 6-7
        faddp   st(1),st(0) ;starts on cycle 8
                            ;stalls for cycles 9-10
        faddp   st(1),st(0) ;starts on cycle 11
                            ;stalls for cycles 12-14
        fstp    [dot]       ;starts on cycle 15,
                            ; ends on cycle 16
```

LISTING 63.3 L63-3.ASM

```
;optimized dot product; 15 cycles
        fld     [vec0+0]    ;starts & ends on cycle 0
        fmul    [vec1+0]    ;starts on cycle 1
        fld     [vec0+4]    ;starts & ends on cycle 2
        fmul    [vec1+4]    ;starts on cycle 3
        fld     [vec0+8]    ;starts & ends on cycle 4
        fmul    [vec1+8]    ;starts on cycle 5
        fxch    st(1)       ;no cost
        faddp   st(2),st(0) ;starts on cycle 6
                            ;stalls for cycles 7-8
        faddp   st(1),st(0) ;starts on cycle 9
                            ;stalls for cycles 10-12
        fstp    [dot]       ;starts on cycle 13,
                            ; ends on cycle 14
```

The Cross Product

When last we looked at the cross product, we found that it's handy for generating a vector that's normal to two other vectors. The cross product is calculated as $[u_2v_3\text{-}u_3v_2\ u_3v_1\text{-}u_1v_3\ u_1v_2\text{-}u_2v_1]$. The theoretical minimum cycle count for the cross product is 21 cycles. Listing 63.4 shows a straightforward implementation that calculates each component of the result separately, losing 15 cycles to stalls.

LISTING 63.4 L63-4.ASM

```
;unoptimized cross product; 36 cycles
        fld     [vec0+4]     ;starts & ends on cycle 0
        fmul    [vec1+8]     ;starts on cycle 1
        fld     [vec0+8]     ;starts & ends on cycle 2
        fmul    [vec1+4]     ;starts on cycle 3
                             ;stalls for cycles 4-5
        fsubrp  st(1),st(0)  ;starts on cycle 6
                             ;stalls for cycles 7-9
        fstp    [vec2+0]     ;starts on cycle 10,
                             ; ends on cycle 11
        fld     [vec0+8]     ;starts & ends on cycle 12
        fmul    [vec1+0]     ;starts on cycle 13
        fld     [vec0+0]     ;starts & ends on cycle 14
        fmul    [vec1+8]     ;starts on cycle 15
                             ;stalls for cycles 16-17
        fsubrp  st(1),st(0)  ;starts on cycle 18
                             ;stalls for cycles 19-21
        fstp    [vec2+4]     ;starts on cycle 22,
                             ; ends on cycle 23
```

```
fld     [vec0+0]     ;starts & ends on cycle 24
fmul    [vec1+4]     ;starts on cycle 25
fld     [vec0+4]     ;starts & ends on cycle 26
fmul    [vec1+0]     ;starts on cycle 27
                     ;stalls for cycles 28-29
fsubrp  st(1),st(0)  ;starts on cycle 30
                     ;stalls for cycles 31-33
fstp    [vec2+8]     ;starts on cycle 34,
                     ; ends on cycle 35
```

We couldn't get rid of many of the stalls in the dot product code because with six inputs and one output, it was impossible to interleave all the operations. However, the cross product, with three outputs, is much more amenable to optimization. In fact, three is the magic number; because we have three calculation streams and the latency of FADD, FSUB, and FMUL is 3 cycles, we can eliminate almost every single stall in the cross-product calculation, as shown in Listing 63.5. Listing 63.5 loses only one cycle to a stall, the cycle before the first FST; the relevant FSUB has just finished on the preceding cycle, so we run into the extra cycle of latency associated with FST. Listing 63.5 is more than 60 percent faster than Listing 63.4, a striking illustration of the power of properly managing the Pentium's FP pipeline.

LISTING 63.5 L63-5.ASM

```
;optimized cross product; 22 cycles
        fld       [vec0+4]    ;starts & ends on cycle 0
        fmul      [vec1+8]    ;starts on cycle 1
        fld       [vec0+8]    ;starts & ends on cycle 2
        fmul      [vec1+0]    ;starts on cycle 3
        fld       [vec0+0]    ;starts & ends on cycle 4
        fmul      [vec1+4]    ;starts on cycle 5
        fld       [vec0+8]    ;starts & ends on cycle 6
        fmul      [vec1+4]    ;starts on cycle 7
        fld       [vec0+0]    ;starts & ends on cycle 8
        fmul      [vec1+8]    ;starts on cycle 9
        fld       [vec0+4]    ;starts & ends on cycle 10
        fmul      [vec1+0]    ;starts on cycle 11
        fxch      st(2)       ;no cost
        fsubrp    st(5),st(0) ;starts on cycle 12
        fsubrp    st(3),st(0) ;starts on cycle 13
        fsubrp    st(1),st(0) ;starts on cycle 14
        fxch      st(2)       ;no cost
                              ;stalls for cycle 15
        fstp      [vec2+0]    ;starts on cycle 16,
                              ; ends on cycle 17
        fstp      [vec2+4]    ;starts on cycle 18,
                              ; ends on cycle 19
        fstp      [vec2+8]    ;starts on cycle 20,
                              ; ends on cycle 21
```

Transformation

Transforming a point, for example from worldspace to viewspace, is one of the most heavily used FP operations in realtime 3-D. Conceptually, transformation is nothing more than three dot products and three additions, as I will discuss in Chapter 61.

(Note that I'm talking about a subset of a general 4×4 transformation matrix, where the fourth row is always implicitly [0 0 0 1]. This limited form suffices for common transformations, and does 25 percent less work than a full 4×4 transformation.)

Transformation is calculated as:

$$\begin{vmatrix} v_1 \\ v_2 \\ v_3 \\ 1 \end{vmatrix} = \begin{vmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \\ m_{31} & m_{32} & m_{33} & m_{34} \\ 0 & 0 & 0 & 1 \end{vmatrix} \begin{vmatrix} u_1 \\ u_2 \\ u_3 \\ 1 \end{vmatrix}$$

or

$$v_1 = m_{11}u_1 + m_{12}u_2 + m_{13}u_3 + m_{14}$$
$$v_2 = m_{21}u_1 + m_{22}u_2 + m_{23}u_3 + m_{24}$$
$$v_3 = m_{31}u_1 + m_{32}u_2 + m_{33}u_3 + m_{34}.$$

When it comes to implementation, however, transformation is quite different from three separate dot products and additions, because once again the magic number *three* is involved. Three separate dot products and additions would take 60 cycles if each were calculated using the unoptimized dot-product code of Listing 63.2, and would take 54 cycles if done one after the other using the faster dot-product code of Listing 63.3, in each case followed by the a final addition per dot product.

When fully interleaved, however, only a single cycle is lost (again to the extra cycle of FST latency), and the cycle count drops to 34, as shown in Listing 63.6. This means that on a 100 MHz Pentium, it's theoretically possible to do nearly 3,000,000 transforms per second, although that's a purely hypothetical number, due to cache effects and set-up costs. Still, more than 1,000,000 transforms per second is certainly feasible; at a frame rate of 30 Hz, that's an impressive 30,000 transforms per frame.

LISTING 63.6 L63-6.ASM

```
;optimized transformation: 34 cycles
      fld    [vec0+0]       ;starts & ends on cycle 0
      fmul   [matrix+0]     ;starts on cycle 1
      fld    [vec0+0]       ;starts & ends on cycle 2
      fmul   [matrix+16]    ;starts on cycle 3
      fld    [vec0+0]       ;starts & ends on cycle 4
      fmul   [matrix+32]    ;starts on cycle 5
      fld    [vec0+4]       ;starts & ends on cycle 6
      fmul   [matrix+4]     ;starts on cycle 7
      fld    [vec0+4]       ;starts & ends on cycle 8
      fmul   [matrix+20]    ;starts on cycle 9
      fld    [vec0+4]       ;starts & ends on cycle 10
      fmul   [matrix+36]    ;starts on cycle 11
      fxch   st(2)          ;no cost
      faddp  st(5),st(0)    ;starts on cycle 12
      faddp  st(3),st(0)    ;starts on cycle 13
      faddp  st(1),st(0)    ;starts on cycle 14
      fld    [vec0+8]       ;starts & ends on cycle 15
```

```
fmul    [matrix+8]      ;starts on cycle 16
fld     [vec0+8]        ;starts & ends on cycle 17
fmul    [matrix+24]     ;starts on cycle 18
fld     [vec0+8]        ;starts & ends on cycle 19
fmul    [matrix+40]     ;starts on cycle 20
fxch    st(2)           ;no cost
faddp   st(5),st(0)     ;starts on cycle 21
faddp   st(3),st(0)     ;starts on cycle 22
faddp   st(1),st(0)     ;starts on cycle 23
fxch    st(2)           ;no cost
fadd    [matrix+12]     ;starts on cycle 24
fxch    st(1)           ;starts on cycle 25
fadd    [matrix+28]     ;starts on cycle 26
fxch    st(2)           ;no cost
fadd    [matrix+44]     ;starts on cycle 27
fxch    st(1)           ;no cost
fstp    [vec1+0]        ;starts on cycle 28,
                        ; ends on cycle 29
fstp    [vec1+8]        ;starts on cycle 30,
                        ; ends on cycle 31
fstp    [vec1+4]        ;starts on cycle 32,
                        ; ends on cycle 33
```

Projection

The final optimization we'll look at is projection to screenspace. Projection itself is basically nothing more than a divide (to get 1/z), followed by two multiplies (to get x/z and y/z), so there wouldn't seem to be much in the way of FP optimization possibilities there. However, remember that although FDIV has a latency of up to 39 cycles, it can overlap with integer instructions for all but one of those cycles. That means that if we can find enough independent integer work to do before we need the 1/z result, we can effectively reduce the cost of the FDIV to one cycle. Projection by itself doesn't offer much with which to overlap, but other work such as clamping, window-relative adjustments, or 2-D clipping could be interleaved with the FDIV for the next point.

Another dramatic speed-up is possible by setting the precision of the FPU down to single precision via FLDCW, thereby cutting the time FDIV takes to a mere 19 cycles. I don't have the space to discuss reduced precision in detail in this book, but be aware that along with potentially greater performance, it carries certain risks, as well. The reduced precision, which affects FADD, FSUB, FMUL, FDIV, and FSQRT, can cause subtle differences from the results you'd get using compiler defaults. If you use reduced precision, you should be on the alert for precision-related problems, such as clipped values that vary more than you'd expect from the precise clip point, or the need for using larger epsilons in comparisons for point-on-plane tests.

Rounding Control

Another useful area that I can note only in passing here is that of leaving the FPU in a particular rounding mode while performing bulk operations of some sort. For

example, conversion to int via the FIST instruction requires that the FPU be in chop mode. Unfortunately, the FLDCW instruction must be used to get the FPU into and out of chop mode, and each FLDCW takes 7 cycles, meaning that compilers often take at least 14 cycles for each float->int conversion. In assembly, you can just set the rounding state (or, likewise, the precision, for faster FDIVs) once at the start of the loop, and save all those FLDCW cycles each time through the loop. This is even more true for **ceil()**, which many compilers implement as horrendously inefficient subroutines, even though there are rounding modes for both **ceil()** and **floor()**. Again, though, be aware that results of FP calculations will be subtly different from compiler default behavior while chop, ceil, or floor mode is in effect.

A final note: There are some speed-ups to be had by manipulating FP variables with integer instructions. Check out Chris Hecker's column in the February/March 1996 issue of *Game Developer* for details.

A Farewell to 3-D Fixed-Point

As with most optimizations, there are both benefits and hazards to floating-point acceleration, especially pedal-to-the-metal optimizations such as the last few I've mentioned. Nonetheless, I've found floating-point to be generally both more robust and easier to use than fixed-point even with those maximum optimizations. Now that floating-point is fast enough for real time, I don't expect to be doing a whole lot of fixed-point 3-D math from here on out.

And I won't miss it a bit.