Attention:

Chris Hecker
definition six, incorporated
checker@d6.com
http://www.d6.com/users/checker

PS. The email address at the end of the article is incorrect. Please use checker@d6.com for any correspondence.

Perspective Texture Mapping, Part V: It's About Time

Chris Hecker

Finally! The moment game developers have been waiting for. At long last, Chris Hecker unveils the denoue-ment of his texture mapping series. What a long, strange trip it has been.

Check the date on the *Game Developer* in your hands. Does April/May 1996 mean anything special? How about the same issue last year: April/May 1995? What, you have no idea what I'm talking about? That's understandable, since it was so long ago. I'll refresh your memory with a small quote from my column in that 1995 issue: "My next two articles should fix this lack of documentation, first by giving an easy-to-understand mathematical foundation…and sample code to implement the naive algorithm. In the next article, we'll speed it up to interactive performance."

Yes, you guessed it, the April/May 1995 issue contained the first installment of my "two-part" perspective texture mapping series. Now, a year later, we're finally going to finish the two part series with this issue, part five. True to my software engineering background, I can't estimate how long it will take me to do something to save my life. However, I feel the somewhat lengthy trip has been worth it.

A Long, Strange Trip

Unlike the first installment, this article will not be a journey through the elegant theory and math behind perspective texture mapping. Instead, this article will romp through the myriad optimization tricks and techniques we can use to squeeze every last bit of texture mapping performance from the Intel Pentium processor. While the rest of the series was platform independent, we have to turn to assembly language to get the most out of modern processors, and the Pentium is the market leader (and the CPU I know best). You'll still get a lot from reading this regardless of your chosen platform, but the code is specific to Intel. However, I got a new Macintosh clone from Power-Computing with a PowerPC 604 chip in it, so don't be surprised to see a PowerPC version of this texture mapper at some later date (as soon as I get used to the concept of 32 general purpose registers)!

To quickly bring us up to date, we decided to use a subdivided affine approximation to the true perspective curve (Behind the Screen, Dec./Jan. 1995). This approximation uses short linear spans with perspectively correct endpoints to approximate the rational perspective curve. I modified the `DrawScanLine` function to do the affine subdivision in C++ and the texture mapper got three times faster than the version with the divides. I uploaded a sample application that contains the texture mappers we developed so far. Of course, I was late in uploading it (see above comment about saving my life), and I apologize to anyone who looked and couldn't find it. The sample is up now though—see the end of this column for information on where to find it.

I can provide an overall outline for the optimizations we'll cover before going into detail. Our main optimizations will take advantage of the dual integer pipelines on the Pentium to implement a very fast fixed-point linear texture mapper for our affine spans, and we'll use the floating-point unit's ability to overlap execution (especially those costly perspective divides) with integer instructions to calculate the interpolation values for the next span as we render the current span.

Listing 1. The C++ Inner Loop

```
for(int Counter = 0;Counter < AffineLength;Counter++)
{
  int UInt = U>>16;
  int VInt = V>>16;

  *(pDestBits++) = *(pTextureBits + UInt +
     (VInt * TextureDeltaScan));

  U += DeltaU;
  V += DeltaV;
}
```

In addition, we'll pull a bunch of cheap tricks along the way to give it that extra kick.

Carry Me Away

Listing 1 shows the C++ linear inner loop for our **DrawScanLine** function. While this is better than our previous loops with their divides, it's still not great because it's doing a multiply and a bunch of shifts and adds for each pixel. If we look at why it's doing a multiply, we see it's calculating the source texture offset for each new coordinate, even though we're just doing a normal linear interpolation through the texture for each span. By definition, a linear interpolation increments by the same amount each step, so we can take advantage of this coherency to speed up our loop.

Let's ignore the V coordinate for the moment and see how we can take advantage of the U coordinate's coherency. As you can see from the loop, the U fixed-point variable is shifted down to extract its integer portion for each pixel, which is then added into the texture pointer. However, since we're linear, the integer portion of **DeltaU** is going to stay constant for the span—always incrementing the integer part of U by the same amount—so there's really no need to keep the integer portion of U around at all. If we think only of the U increments, we can keep the source pointer at the current texture pixel, and we can find the next texture pixel by just adding in the integer part of **DeltaU**—calculated outside the loop—to the pointer at each step. The only problem with this plan is the fractional part of U plus the fractional part of **DeltaU** will sometimes carry into the integer part of U. We need to know when this happens and add an extra step to our source pointer.

This carry problem uncovers a major hole in C and C++ from the standpoint of integer optimizations: there's no carry bit. In other words, in assembly language, it's trivial to know when two added numbers overflow because the carry bit will be set, but in C++ there's no way to know without doing a bunch of cumbersome tests. So, in pseudocode, we want to do this:

```
UFrac += DeltaUFrac
pTexture += DeltaUInt + Carry
```

Where the variable **Carry** is set to 0 if the fractional addition didn't carry into the imaginary integer part (that we're not storing anymore) and is set to 1 if the addition did carry. This pseudocode is trivial in all the assembly languages I've ever seen, for example, in x86 assembly:

```
add             ebx,ecx
adc             esi,edx
```

Assuming the given registers contain the right values, the **adc** (add with carry) will add in the step and any carry from the previous addition. Implementing this code in C++ would be a mess.

That's it for the U coordinate, but we conveniently ignored the V coordinate because it's a good deal trickier. Like U, the V coordinate is linear for our span, so we can precalculate our increment and leave the integer part of V out of our loop. However, as you can see from Listing 1, the integer part of the V coordinate is scaled to step vertically in our texture bitmap. This doesn't present a problem for the normal V step, but when the fractional part of V carries into the integer part, the source pointer no longer steps by 1, it steps by the width of the texture bitmap. Not even assembly language has an instruction to add in an arbitrary number—like the **TextureDeltaScan** in Listing 1—on carry.

Quickly adding in the vertical source step on V's carry is where 99% of the programming brainpower is spent on linear texture mapping optimizations. I've seen about five or six ways of doing it myself, but by far the coolest, fastest, and most elegant way I've seen was invented by Michael Abrash. However, before I describe it, I'm going to address the optimization a lot of the experienced texture mappers in the audience think I'm going to use here.

If you go out on the Internet and look for affine texture mappers, you'll undoubtedly run into a lot of very optimized x86 code that only works with power-of-two source texture sizes, and specifically two to the eighth power (or 256-bytes wide for 8bpp textures), because if you keep your textures to a power-of-two width, you can very easily handle the V carry we're discussing using some special x86 instructions that operate on 8-bit portions of the full registers.

Let's run through an example, where our source texture is 256 by 256. We'll use the x86's **ebx** register, and its corresponding "byte registers," **bh** and **bl**. The byte registers are part of the 32 bit **ebx** register, and **bl** (b-low) is the lowest 8 bits—bit 0 through 7—and **bh** (b-high) is the next higher 8 bits—bit 8 through 15. If we keep the U coordinate in **bl** and the V coordinate in **bh**, we can use the following code to increment both U and V (assuming **ecx** and **edx** have the current **UFrac** and **VFrac**, respectively, and **esi** contains the texture pointer):

```
add             ecx,[DeltaUFrac]
adc             bl,[UIntStep]
add             edx,[DeltaVFrac]
adc             bh,[VIntStep]
add             esi,ebx
```

Notice the second **adc**. It adds in the carry from the **VFrac** addition, but I just got finished saying how this wouldn't

Listing 2. The x86 Asm Inner Loop

```
add     edx,[DeltaVFrac]        ; add in dVFrac

sbb     ebp,ebp                 ; store carry
mov     [edi],al                ; write pixel n

mov     al,[esi]                ; fetch pixel n+1
add     ecx,ebx                 ; add in dUFrac

adc     esi,[4*ebp + UVStepVCarry]      ; add in steps
```

work because V's carry needs to add in the width of the texture. The trick is that bh is actually already multiplied by the width of our texture—256—by virtue of its bit position in `ebx`. Neat, huh? Now, given such a cool trick, why wouldn't we use it? There are two reasons: first, restricting yourself to power-of-two textures isn't very flexible and is bad for cache coherency (see Behind the Screen, Oct./Nov. 1995). More importantly, with the Pentium Pro, this code will run slowly due to a new pipeline stall called the Partial Register Stall (PRS). The PRS happens when you modify one of the byte registers and then try to use the encompassing 32-bit register, much like the above code. The instruction `add esi,ebx` will stall for a very long time on the Pentium Pro. Why did Intel let this happen? I have no clue, although they say it will let them increase the clock speed more than if they had prevented the stall. Regardless, it's there, and we'll need to live with it.

So, given that we're not going to use the power-of-two texture trick, how do we write our code so it can carry an arbitrary value into the pointer when `VFrac` overflows? Enter Abrash's code snippet shown in Listing 2. This is the code for a pixel from the middle of an unrolled loop, so there's a bit of setup not shown here, but imagine this same snippet concatenated with itself a bunch of times. See if you can figure out how it works and then read on for the description of this tour de force of optimization.

Hit the Pipe

There are so many cool things about this code it's hard to know where to start describing it, but, since we were discussing the V carry, we'll start with how the code addresses that problem. The first half of the solution is these two instructions:

```
add     edx,[DeltaVFrac]
sbb     ebp,ebp
```

The first instruction adds in the fractional step as usual, but the second instruction saves the carry flag, using a neat trick involving the `sbb` (subtract with borrow) instruction. The `sbb` instruction is like the opposite of `adc`, it subtracts its source from its destination, but also subtracts the carry bit, so `sbb ebp,ebp` will subtract the `ebp` register from itself, giving 0 if there was no carry, or -1 if there was a carry. Thus, the carry bit from the `VFrac` addition is stored as a 0 or a -1 in `ebp`.

The second half of the solution comes with these instructions:

```
add     ecx,ebx
adc     esi,[4*ebp + UVStepVCarry]
```

The first instruction is the `UFrac` addition, so after it completes, the carry bit is set appropriately. The next instruction is where all the action occurs. It's an `adc`, so it adds in the carry from the `UFrac` addition as you'd expect. However, it's an `adc` from memory, and it uses a two `dword` array to accomplish its magic. `UVStepVCarry` is the address of the second `dword` in the array, and the 0 or -1 in `ebp` from the `VFrac` carry will select either the second `dword` if there was no V carry, or the first `dword` if there was a V carry (since 4 • -1 will subtact 4 bytes from the array address). The only thing left is to make sure the array has the appropriate steps in it, including the U and V integer steps and the V carry step for the first element in the array.

As if the basic operation wasn't good enough, the pipelining on this code is amazing as well. The order of instructions perfectly fills both pipes on the Pentium and manages to run the two additions from memory—both two-cycle instructions—in the Pentium U and V pipes at the same time (remember, the next pixel's code will come right after this pixel, so the `add edx` and the `adc esi` will run at the same time). The instructions are also far enough away from each other that there are no Address Generation Interlocks. Overall, it's a beautiful piece of code.

Walking and Chewing Gum

Regardless of how amazing our integer affine inner loop is, we'll still be slower than we need to be if we're waiting for the floating-point unit to calculate the perspective-corrected texture coordinates before starting each span. This is where the floating-point overlap I hinted about in the last issue enters in. Most modern processors can execute floating-point instructions at the same time as integer instructions, and some, like the Pentium, can execute multiple floating-point instructions at the same time. As an example of integer and floating-point overlap, the following code will take 36 cycles on a Pentium in double precision mode:

```
fdiv    [Number1]
fst     [Number2]
```

The division takes 33 cycles, the store takes two cycles, and there's a one-cycle stall for trying to store the result of the division right after it's completed. Guess how long the following code takes:

```
fdiv    [Number1]
rept 33
        add     ebx,ecx
        add     edx,eax
endm
fst     [Number2]
```

To guess correctly, you need to know that the `rept` macro repeats the contained code 33 times, so there are actually 66 instructions between the `fdiv` and the `fst`. This is actually a trick ques-

tion because this code takes the same 36 cycles as the first snippet, but we got to execute 66 integer instructions for free!

Well, we don't really get just any 66 instructions for free, but we do get 33 U and V pipe slots in which we can try to get some work done before using the result of the division. Some instructions, like integer multiplies, won't overlap with the floating-point unit, and you can't really do many other floating-point instructions at the same time as floating-point division, but we can start up the perspective divide for our next span and have it calculate as we're processing the current span, making it almost free.

Listing 3. Naive Adding

```
fld   [a1]        ; 1
fadd  [b1]        ; 3
fstp  [a1]        ; 2+1
fld   [c1]        ; 1
fadd  [d1]        ; 3
fstp  [c1]        ; 2+1
fld   [e1]        ; 1
fadd  [f1]        ; 3
fstp  [e1]        ; 2+1
```

Short Stack

The second floating-point technique I mentioned, executing multiple floating-point operations simultaneously, is slightly more convoluted. The Intel floating-point architecture is stack based, which means almost all the floating-point instructions will only operate on the top of the stack. This made it hard for Intel to pipeline the floating-point unit for the Pentium since all the instructions were vying for the same register—the floating-point top-of-stack register. So, instead of breaking all the existing floating-point code by making a bunch of new instructions to randomly access the floating-point registers, Intel decided to make it possible to move operands around on the stack very quickly. I actually wish they'd broken the code and made random register access easy, but Intel doesn't usually ask my opinion on these things, so I'll quickly describe the stack-based solution.

Listing 3 shows the obvious way to add some numbers together, along with the cycle counts for each instruction. It's implementing this C++ code:

```
a1 += b1; c1 += d1; e1 += f1;
```

The code executes in 21 cycles, including the three stalls (the 2+1 `fstp`

Listing 4. Quick Adding

```
fld   [a1]        ; 1
fadd  [b1]        ; 1
fld   [c1]        ; 1
fadd  [d1]        ; 1
fld   [e1]        ; 1
fadd  [f1]        ; 1
fxch  st(2)       ; 0
fstp  [a1]        ; 2
fstp  [c1]        ; 2
fstp  [e1]        ; 2
```

timings) for storing the results of an operation immediately following its completion. Listing 4 shows an alternate implementation of the same code, which executes in 12 cycles, or almost twice as fast. The instructions can pipeline if you don't access their results before the instruction is finished (the `fld` instruction pushes its operand onto the stack, and the previous `fadd` continues on its operand even as it's moved down one stack position). In our example, the `fadd`s take 3 cycles each in Listing 3, but only a single cycle each in Listing 4.

The second thing to notice is that the `fxch` instruction is free in Listing 4. This is Intel's offering to the angry God of Processor Architecture, who threatened to smite Intel dead if it didn't pipeline the floating-point unit. The almost-free `fxch` instruction makes it possible—not easy, just possible—to pipeline your floating-point code even though most of the instructions only operate on the top-of-stack register. Using `fxch`, you can move things around while they're still calculating, like the `e1+f1` addition in Listing 4. I called it "almost-free" because there are some restrictions you have to obey to keep it free; for example, the following instruction must be a floating-point operation, as it will stall a cycle if the following instruction is an integer operation. Intel's AP500 Application Note, available on their www.intel.com site and the Intel Architecture Labs CD, describes this technique in detail.

Finally

There are more tricks in the assembly texture mapper that deserve a mention, but they're all minor and I'm out of space. You can find them in the code itself. As with most assembly code, the texture mapper is way too long to include here in the magazine. You can, however, get it in the texture mapping archive on the *Game Developer* web site, on its ftp site (ftp.mfi.com/pub/gamedev/src), or on my homepage at http://ourworld.compuserve.com/homepages/checker.

How fast is it? Well, I must admit, I'm not finished optimizing it yet as I write this (again, see the comment about estimating how long it takes me to do something at the beginning of this article), but it's already two times as fast as the C++ subdividing affine texture mapper, and I hope to make it another two times faster by the time you read this and are able to pick up the code. It's currently drawing 4.5 million pixels per second on a Pentium 133, which is 5 times faster than our original texture mapper, and fast enough to do 70 frames per second at 320 by 200 if you're not doing anything else except texture mapping. That's definitely fast enough for a high-end 3D game, and I think we can safely say we've met the goals we set for ourselves at the beginning of this series, even if we did meet them a bit late. ■

You can e-mail Chris Hecker at checker@bix.com. Don't be surprised if it takes him a while to respond, although he'll assure you it will only take a second....