Optimizing Vertex Linear Interpolation with the Intel® Pentium® III and Pentium® 4 Processors

By Wayne Coles
Introduction

Linear interpolation is a simple process of taking the position of a single vertex at two points in time and, as time passes, moving the current position between them. To perform this calculation, we require a value to signify the current time. This value ranges from 0 (where our vertex position will be at the start vertex) and 1 (where our vertex position will be at the end vertex). We can perform this calculation by multiplying the first vertex position by $1 - \text{time}$, the second vertex by $\text{time}$ and finally adding these two results together.

Thus:

$$\text{Current} = \text{start} \times (1.0f - \text{time}) + \text{end} \times \text{time}$$

This is called linear interpolation. The popular game series Quake* utilizes this method to improve the quality of its animations and reduce its memory requirement, allowing the game to only store key-frames of animation and linearly interpolate between them.

A naïve implementation of linear interpolation is shown in Example 1.

Example 1

```c
struct Vertex
{
    float vx, vy, vz;  // Vertex position
    float nx, ny, nz;  // Vertex normal
};

void Lerp( Vertex *pOut, Vertex *pStart, Vertex *pEnd, float fTime )
{
    float fMinus = 1.0f - fTime;  // Precalculate 1 - fTime

    for ( int nLoop = 0; nLoop < nVertices; nLoop++, pOut++, pEnd++, pStart++ )
    {
        pOut->vx = pStart->vx * fMinus + pEnd->vx * fTime;
        pOut->vy = pStart->vy * fMinus + pEnd->vy * fTime;
        pOut->vz = pStart->vz * fMinus + pEnd->vz * fTime;
        pOut->nx = pStart->nx * fMinus + pEnd->nx * fTime;
        pOut->ny = pStart->ny * fMinus + pEnd->ny * fTime;
        pOut->nz = pStart->nz * fMinus + pEnd->nz * fTime;
    }
}
```

Here we have interpolated the vertex normal as well as the vertex position. Technically this introduces errors into the resulting vertex as a linearly interpolated normal will not retain its unit length. We could renormalize after each interpolation, however normalization is quite an expensive operation and the errors introduced are typically small and unnoticeable (however this does depend on the function that the interpolated value serves).

Using the technique illustrated in Example 1 would certainly result in correctly interpolated values; however, the runtime performance leaves plenty of room for improvement.

SSE – An Introduction

SSE stands for Streaming SIMD Extensions, an instruction set supported by Intel Pentium® III and Pentium® 4 processors. The instruction set was explicitly created for use in accelerating multimedia processing, including 3D graphics computations. SIMD stands for Single Instruction Multiple Data, which means that we can give the processor one instruction and get multiple results back! SSE allows us to perform, say, a single multiply on up-to four values in one instruction. This could mean a huge boost in performance for our linear interpolation routine!

Intel introduced the SSE2 instruction set with the Pentium 4 processor. SSE2 extends the original SSE instruction set with support for packed double-precision floating-point values and for 128-bit packed integers. For the optimization efforts mentioned in this article, we only require single-precision floating point operations, which are provided with the SSE instruction set.

There are a few ways in which one can write SSE code. The IA-32 C++ classes allow the programmer to implement SIMD instructions using C++ classes. Alternatively, intrinsics offer a method of providing function-style operations that compile directly to assembly code. Finally, one can directly program the SSE code using Assembly language. We use Assembly language in this tutorial, as a matter of preference. If you are using Microsoft® Visual C++ 6 or earlier, you may need to upgrade the compiler for SIMD instruction support (visit Microsoft’s website for the processor pack upgrade) or install the latest version of the Intel® C++ compiler available at www.intel.com/software/products. Microsoft® Visual Studio .NET comes with all the necessary support, and is the development environment used to write this tutorial.
Notes to the reader

Intel documents generally comment their registers from the lowest value. In this article, we comment registers in register value order. We recommend the *Intel® Pentium® 4 and Intel® Xeon™ Processor Optimization Manual* available at http://www.intel.com/design/Pentium4/manuals/. Although the title suggests that it is only useful for targeting the Pentium 4 processor, it contains much information relevant to any SSE programming.

e.g.

Intel manual commenting…

```
movaps xmm1, [ edi ]  // x1 x2 x3 x4
```

My Commenting…

```
movaps xmm1, [ edi ]  // x4 x3 x2 x1
```

Commenting registers in register value order allows one to see x4, x3 as the high double word of the register and x2, x1 as the low double word of the register. Finally, we’ve marked ‘unknown’ values in a register with a double dash "--" to signify that the value is unknown.

Detecting SSE Support

The first thing one must do is determine if the current machine supports the SSE instruction set. Fortunately for Windows* programmers, one can simply ask the operating system.

```c
BOOL IsProcessorFeaturePresent( DWORD ProcessorFeature );
```

If we pass in the parameter PF_XMMI_INSTRUCTIONS_AVAILABLE to this function and SSE is supported, the function will return TRUE. Those developing on a different OS, or those who prefer to ask the machine directly may determine whether SSE is supported using Assembly.

The Pentium® family of processors supports a specific command for detecting the presence of processor features called CPUID. To use this instruction, load the EAX register with the appropriate request code. The processor then fills the EAX, EBX, ECX and EDX registers with the requested information. To get supported feature information, including whether the processor supports SSE, the request code is 1. You can retrieve much more information using the CPUID instruction. To access the entire list of request codes for CPUID, download the IA-32 Intel Architecture Software Developer's Manual Volume 2: Instruction Set Reference from the Intel website.

When request code 1 is loaded to the EAX and the CPUID instruction issued, bit 25 of the EDX register will be 1 if the processor supports SSE, otherwise it will be 0.

```c
int CheckSSE()
{
    int iAvailable = 0;
    __asm
    {
        mov  eax, 1
        cpuid
        test edx, 0x02000000
        jz not_found
        mov [ iAvailable ], 1
    not_found:
    }
    return iAvailable;
}
```

AoS – Array of Structures

There are two major ways to use SSE. We’ll review the one that better suits the typical implementation of interpolation demonstrated above. First, we declared a structure to contain the data, and then we declared an array of this structure with enough entries to contain all the vertices. The SSE operations perform on four packed single-precision floating-point values at a time; however, the vertex position and normal are defined as two groups of three floats. This is easily fixed by using a homogenous representation, which simply means adding an extra component to our vectors, the w component, and assigning w a value of 1. Below is the modified structure:

```c
define Vertex
```
Although we’ve added an extra component (which by using our previous implementation would mean two extra multiply and additions) SIMD will perform four at a time so we actually achieve an overall saving on the number of operations performed. We’ll call our interpolation function LerpSSE and have the caller pass in the source vertex data, a target address to store the interpolated data and the interpolation value. We’ll interpolate from source A to source B, where an interpolation value of 0 will return source A and an interpolation value of 1 will return source B.

```c
void LerpSSE( Vertex *pTarget, Vertex *pSourceA, Vertex *pSourceB, float fLerp );
```

The first SSE instruction I’ll introduce is `movups`, which means ‘Move Unaligned Packed Single Precision’. This moves a set of four floating point values into a specified register. We’ll use this to get our vertex data in order to perform our interpolation. To start with, we’ll create a loop that loads our vertex data then immediately stores it into the target without performing any interpolation.

```c
void LerpSSE( Vertex *pTarget, Vertex *pSourceA, Vertex *pSourceB, float fLerp )
{
    __asm {
        mov ecx, nVertices * 2 // Vertex Count
        mov edi, pTarget  // Target address
        mov ebx, pSourceA  // Source A vertex data
        mov edx, pSourceB  // Source B vertex data

        sse_loop:
            // Load our source vertices into the SSE registers
            movups xmm0, XMMWORD PTR[ ebx ] // a3 a2 a1 a0
            movups xmm1, XMMWORD PTR[ edx ] // b3 b2 b1 b0

            // Apply interpolation...

            // Store our interpolated vertex here...
            movups XMMWORD PTR[ edi ], xmm0

            // Step to the next vertex...
            add edi, 16
            add ebx, 16
            add edx, 16

            // Loop for each vertex...
            dec ecx
            jne sse_loop
    }
}
```

Here we’re multiplying the vertex count by 2 because there are two sets of 4 float values we want to interpolate (the position and the normal). We then decrement the `ecx` register and loop while `ecx` is non-zero.

There are 8 xmm registers named xmm0 through to xmm7, each of these 128bit registers can hold four floating point values.

To help visualize the next sections, I’ll introduce a simple diagram to show you the layout of an xmm register while performing single precision arithmetic:

<table>
<thead>
<tr>
<th>xmm register in single precision mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value4</td>
</tr>
</tbody>
</table>

As we already know, each xmm register contains four single precision values. Many SSE instructions use the contents of the destination register while generating the result, so descriptions of their operation generally contain phrases such as ’Moves from the destination register to the destination register’. Practicing with the instructions will help you understand them more fully.

### Interpolating

Now that we have our loop, we’re ready to perform the interpolation operation. In our loop we’ve loaded the vertex information into `XMM0` and `XMM1`. We’ll also need to move the interpolation values into some of the other registers. We’ll need a register filled with `fLerp` (which is...
the variable representing time) and another filled with \( 1 - f_{\text{Lerp}} \). We can then get these two values into our xmm registers using the SSE instruction `movss` (which means 'Move Scalar Single Precision'), which moves a single floating point value into the bottom of the xmm register. We do this for both the interpolation values and move them into separate registers. For this tutorial, I’ve chosen \( XMM6 \) and \( XMM7 \).

Let’s take a look at our routine’s setup code as it stands now:

```c
void LerpSSE( Vertex *pTarget, Vertex *pSourceA, Vertex *pSourceB, float fLerp )
{
    float fMinusLerp = 1.0f - fLerp;
    __asm {
        mov ecx, nVertices * 2 // Vertex Count
        mov edi, pTarget       // Target address
        mov ebx, pSourceA      // Source A vertex data
        mov edx, pSourceB      // Source B vertex data

        movss xmm6, [ fLerp     ] // -- -- -- fLerp
        movss xmm7, [ fMinusLerp ] // -- -- -- fMinusLerp

        sse_loop: // Main loop
    }
}
```

Our next problem is that these values only exist in the lower 32 bits of our xmm register (if you remember, each xmm register contains space for four packed single-precision floating-point values). We’ll need to copy them so that they fill the entire register. Fortunately there is an SSE instruction that we can use called `shufps` (which stands for ‘Shuffle Packed Single Precision’). This command can be a little confusing at first, but it does become clear with practice.

We need to provide a source and destination register as well as a select operand. The select operand makes the instructions versatile. You’ll need to understand binary numbers to fully grasp what the select operand does.

The select operand is an 8-bit value comprised of four 2-bit values. Each of these 2-bit values corresponds to an entry in the destination register. Bits 0 and 1 correspond to the bottom floating point value, bits 2 and 3 correspond to the second value, 4 and 5 to the third and finally bits 6 and 7 correspond to the fourth value in the xmm register.

A 2-bit value can hold a number between 0 and 3 which corresponds to an entry within a register. The first two entries in the select operand specify the values to take from the destination register to be stored in the first two values of the result. The second two entries in the select operand specify the values to take from the source register to be stored in the last two values of the result. The result is then stored in the destination register.

To help us formulate our select operand value, we can define a simple macro that allows us to specify the select operand in a more intuitive way.

```c
#define SHUF_SELECT( valuea, valueb, valuec, valued )  
    ( ( valuea ) << 6 ) | ( ( valueb ) << 4 ) |  
    ( ( valuec ) << 2 ) | ( valued )
```

Using this macro we can explicitly specify which values we want. The macro will then combine them together to form our `shufps` select operand.

E.g.

```c
// xmm0 = x4 x3 x2 x1
// xmm1 = y4 y3 y2 y1
shufps  xmm1, xmm0, SHUF_SELECT(  3, 2, 3, 2 )
// xmm1  = x4 x3 y4 y3
```

For our purposes, we need only take the first value and copy it to the rest of the values in the register. For this we can simply specify the register as the source and destination and pass in zero for the select operand (which always references the first value in our register). The final initialize code now looks like:

```c
void LerpSSE( Vertex *pTarget, Vertex *pSourceA, Vertex *pSourceB, float fLerp )
{
    float fMinusLerp = 1.0f - fLerp;
    __asm {
        mov ecx, nVertices * 2 // Vertex Count
        mov edi, pTarget       // Target address
        mov ebx, pSourceA      // Source A vertex data

        movss xmm6, [ fLerp     ] // -- -- -- fLerp
        movss xmm7, [ fMinusLerp ] // -- -- -- fMinusLerp

        sse_loop: // Main loop
    }
}
```
With the registers prepared, we can perform the interpolation. As before, we multiply the first vertex information by \( 1 - \text{time} \) and the second vertex information with \( \text{time} \), then add the results. The SSE instruction `mulps` ("Multiply Packed Single Precision") multiplies two xmm registers, and the `addps` ("Add Packed Single Precision") instruction adds two xmm registers (containing packed single-precision floating-point data). We use these two instructions to perform the interpolation:

```asm
// Apply interpolation...
mulps xmm0, xmm7  // sourceA * ( 1.0f – fLerp )
mulps xmm1, xmm6  // sourceB * fLerp
addps xmm0, xmm1  // sourceA + sourceB
```

This leaves us with the interpolated vertex coordinate in the \( XMM0 \) register, which we store in the destination address. Finally, we add the steps together to create our first SSE interpolation routine:

```c
void LerpSSE( Vertex *pTarget, Vertex *pSourceA, Vertex *pSourceB, float fLerp )
{
    float fMinusLerp = 1.0f – fLerp;

    __asm {
        mov ecx, nVertices * 2  // Vertex Count
        mov ed, pTarget  // Target address
        mov ebx, pSourceA  // Source A vertex data
        mov edx, pSourceB  // Source B vertex data

        movss xmm6, [ fLerp ]  // -- -- -- fLerp
        movss xmm7, [ fMinusLerp ] // -- -- -- fMinusLerp
        shufps xmm6, xmm6, 0  // fLerp fLerp fLerp fLerp
        shufps xmm7, xmm7, 0  // fMinusLerp fMinusLerp fMinusLerp fMinusLerp

        sse_loop:
            // Load our source vertices into the SSE registers
            movups xmm0, XMMWORD PTR[ ebx ]  // a3 a2 a1 a0
            movups xmm1, XMMWORD PTR[ edx ]  // b3 b2 b1 b0

            // Apply interpolation...
            mulps xmm0, xmm7  // sourceA * ( 1.0f – fLerp )
            mulps xmm1, xmm6  // sourceB * fLerp
            addps xmm0, xmm1  // sourceA + sourceB

            // Store our interpolated vertex here...
            movups XMMWORD PTR[ ed ], xmm0

            // Step to the next vertex...
            add edi, 16
            add ebx, 16
            add edx, 16

            // Loop for each vertex...
            dec ecx
            jne sse_loop
    };
}
```
Data Alignment

We have just made our linear interpolation faster, but we can make it faster still. The first and easiest optimization is to align the data. As we’ve already seen, the SSE instructions operate on four values at once. We can optimize the data access by aligning the data in memory on a four float boundary (16 bytes). If you’re using VC7 or VC6 with the processor pack, you may use the aligned_malloc function to perform the alignment. Alternatively, you may use the intrinsics_mm_malloc() and the corresponding_mm_free() for dynamic memory, or you may use__declspec(align(16)) for allocating aligned static memory.

Allocating the vertex data with aligned_malloc guarantees that the vertex data is aligned on a 16 byte memory address. One can load the data into an xmm register using a faster alternative command to movups, called movaps (‘Move Aligned Packed Single Precision’) but the alternative requires the data to be aligned to a 16-byte boundary. This doesn’t immediately look to be a major alteration, however, using aligned data helps the processor perform its memory access much faster, which translates to faster interpolation code. Using movaps on unaligned memory will cause an interrupt, and the OS will likely display an illegal operation error.

SoA Structure of Arrays

We have now written our interpolation code using the SSE instruction set. It’s not bad but we can still improve upon it. As mentioned earlier, we have an extra float in our structures whose only purpose is to pad the vertex information to a multiple of 16 bytes, leaving us with one superfluous result. We can resolve this by discarding the AoS formatted vertex data and re-designing the data to use a structure of arrays. Here is the new structure definition:

```c
struct Vertex
{
    float x[ nVertices * 2 ];
    float y[ nVertices * 2 ];
    float z[ nVertices * 2 ];
};
```

Here we have a structure that contains three arrays: one for each element within a vertex. For our implementation, each array contains the vertex element followed by the normal element, repeated along the length of the array (so two elements per vertex). We must still pad these arrays to a multiple of 16 bytes, but this means we will receive, at worst, three unwanted results per array. Even with this padding, we see a significant improvement on the AoS approach, which yields one unwanted result (per vertex) but zero unwanted results per array at best. Since our vertex data has two elements per vertex entry, this is reduced to only two unwanted results at worst.

To support this data format, I modified the routine to accept a pointer to an array of floats instead of a pointer to vertex data. Thus, the routine calls the function for each array rather than calling the entire structure at once. We’ll also need an element count with which to loop. We’ll accept the number of total entries in the array and divide it by four, which represents the number of array elements (not the number of vertices). Since the array contains the vertex position followed by the vertex normal, the number of elements will equal the number of vertices multiplied by two. We divide by four, since the SSE code will operate on four values at once.

```c
void LerpSSE( float *pTarget, float *pSourceA, float *pSourceB, float fLerp, int nElements )
{
    float fMinusLerp = 1.0f – fLerp;

    __asm {
        mov ecx, [ nElements ] // Element Counter
        shr ecx, 2   // Divide by 4
        mov edi, pTarget  // Target address
        mov ebx, pSourceA  // Source A vertex data
        mov edx, pSourceB  // Source B vertex data

        movss xmm6, [ fLerp ] // -- -- -- fLerp
        movss xmm7, [ fMinusLerp ] // -- -- -- fMinusLerp
        shufps xmm6, xmm6, 0 // fLerp fLerp fLerp fLerp
        shufps xmm7, xmm7, 0 // fMinusLerp fMinusLerp fMinusLerp fMinusLerp

        sse_loop:
            // Load our source vertices into the SSE registers
            movaps xmm0, XMMWORD PTR[ ebx ] // a3 a2 a1 a0
            movaps xmm1, XMMWORD PTR[ edx ] // b3 b2 b1 b0

            // Apply interpolation...
            mulsps xmm0, xmm7 // sourceA * ( 1.0f - fLerp )
            mulsps xmm1, xmm6 // sourceB * fLerp
            addsps xmm0, xmm1 // sourceA + sourceB
```
movaps  XMMWORD PTR[ edi ], xmm0

add    edi, 16
add    ebx, 16
add    edx, 16

// Loop for each vertex...
dec    ecx
jne    sse_loop
}

The shr instruction binary shifts a register to the right by the specified number of bits. Shifting a value to the right by 2 bits is the same as dividing by four, only quicker.

### Data Swizzling

Using the SoA format results in an optimal data submission for the SSE code. Unfortunately, most uses for the vertex data (such as supplying it to a 3D graphics card for rendering) require data in the more standard AoS format. Somewhere along the line we need to convert from SoA back to AoS, a re-arrangement called data swizzling. First, we can write this code in C just to get started; the following routine will do this for us:

```c
void SwizzleSoA( AOSVertex *pTarget, SOAVertex &source, int nVertices )
{
    int iIndex = 0; // Used to index the SOA structure...

    for ( int nLoop = 0; nLoop < nVertices; nLoop++, iIndex += 2 )
    {
        // Swizzle vertex position...
        pTarget[ nLoop ].vx = source.x[ iIndex ];
        pTarget[ nLoop ].vy = source.y[ iIndex ];
        pTarget[ nLoop ].vz = source.z[ iIndex ];

        // Swizzle vertex normal...
        pTarget[ nLoop ].nx = source.x[ iIndex + 1 ];
        pTarget[ nLoop ].ny = source.y[ iIndex + 1 ];
        pTarget[ nLoop ].nz = source.z[ iIndex + 1 ];
    }
}
```

Even after swizzling, the optimized interpolation code still outperforms the original C interpolation routine because SSE is simply that much faster. However, we can optimize the swizzle function itself and get a further boost.

Fortunately, SSE provides instructions to aid in data swizzling. While swizzling data, you need to monitor what values are contained in the registers. A preferred method of keeping track is to comment each swizzle routine with the data you expect the registers to contain.

We'll need to use two additional SSE instructions, unpcklps and unpckhps to combine data from the low or high double words of two registers. The enclosed diagrams show what happens when they are executed.

**unpcklps – Unpack Low Packed Single Precision**

This instruction takes two registers and combines the two low values.

*Value1* from the destination register is stored in *Value1* of the destination register.  
*Value2* from the destination register is stored in *Value3* of the destination register.  
*Value1* from the source register is stored in *Value2* of the destination register.  
*Value2* from the source register is stored in *Value4* of the destination register.  
The source register remains unchanged.
unpcklps instruction operation diagram

<table>
<thead>
<tr>
<th>Dest 4</th>
<th>Dest 3</th>
<th>Dest 2</th>
<th>Dest 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source 4</td>
<td>Source 3</td>
<td>Source 2</td>
<td>Source 1</td>
</tr>
<tr>
<td>Source 2</td>
<td>Dest 2</td>
<td>Source 1</td>
<td>Dest 1</td>
</tr>
</tbody>
</table>

unpckhps — Unpack High Packed Single Precision

This instruction takes two registers and combines the two high values.

*Value4* from the destination register is stored in *Value3* of the destination register.
*Value3* from the destination register is stored in *Value1* of the destination register.
*Value4* from the source register is stored in *Value4* of the destination register.
*Value3* from the source register is stored in *Value2* of the destination register.

The source register remains unchanged.

With these instructions, we can now tackle our swizzle routine:

```c
struct AOSVertex
{
    float  vx, vy, vz;  // Vertex position
    float  nx, ny, nz;  // Vertex normal
};

struct SOAVertex
{
    float  x[ nVertices * 2 ];
    float  y[ nVertices * 2 ];
    float  z[ nVertices * 2 ];
};

void SwizzleSOA( AOSVertex *pTarget, SOAVertex *pSource, int nVertices )
```

As always we need to prepare our registers for the task at hand. We’ll use *EDI* again as our target pointer and *ECX* as our loop counter. Since we’re handling four floats at a time, we can swizzle two vertices for each loop, so we’ll need to divide our vertex count by two (make sure you pass in a multiple of two vertices as well). Finally, we need to retrieve the array pointers from our source data. Let’s take a quick look at our setup code now:

```asm
    __asm
    {
        mov ecx, [ nVertices ]
        mov edi, [ pTarget ]
        mov ebx, [ pSource ]

        mov edx, [ ebx ] // Retrieve x array pointer
        mov eax, [ ebx + 4 ] // Retrieve y array pointer
        mov ebx, [ ebx + 8 ] // Retrieve z array pointer

        shr ecx, 1 // Divide ecx by 2

        swizzle_loop:
```
First load the current vertex information into the xmm registers:

```assembly
movaps xmm0, [ edx ]  // x4 x3 x2 x1
movaps xmm1, [ eax ]  // y4 y3 y2 y1
movaps xmm2, [ ebx ]  // z4 z3 z2 z1
```

Now we’ll perform the complete swizzle operation for both vertices and then store them back to memory at the end.

First we mix the low x and y values together. We’ll need to use the high values later so we’ll use xmm3 as a temporary register for this operation. We’ll copy the x coordinates into xmm3, then mix them with the y coordinates using `unpcklps`:

```assembly
movaps xmm3, xmm0  // x4 x3 x2 x1
unpcklps xmm3, xmm1  // y2 x2 y1 x1
```

We then copy the z positions into our target registers (xmm4 and xmm5):

```assembly
movaps xmm4, xmm2  // z4 z3 z2 z1
movaps xmm5, xmm2  // z4 z3 z2 z1
```

We can now use the `shufps` instruction to combine our registers into our target format:

```assembly
shufps xmm4, xmm3, SHUF_SELECT( 1, 0, 0, 0 )  // y1 x1 z1 z1
shufps xmm5, xmm3, SHUF_SELECT( 3, 2, 1, 1 )  // y2 x2 z2 z2
```

This is actually our swizzled data ready for storage in memory. We’ll explain the strange layout in the registers shortly. The second vertex swizzle is similar to the first, except that we unpack the two high values (instead of the low values), and we won’t need the coordinates again so we can reuse the position registers, saving us a copy instruction:

```assembly
unpckhps xmm0, xmm1  // y4 x4 y3 x3
movaps xmm6, xmm2  // z4 z3 z2 z1
shufps xmm6, xmm0, SHUF_SELECT( 1, 0, 2, 2 )  // y3 x3 z3 z3
shufps xmm2, xmm0, SHUF_SELECT( 2, 1, 3, 3 )  // y4 x4 z4 z4
```

We’ve now swizzled two sets of vertex data (position and normal). All that remains is to store this data back to memory and loop around for the rest of the vertices.

```assembly
// Store position & normal for vertex 1
movhps [ edi ], xmm4  // Store y1 x1
movss [ edi + 8 ], xmm4  // Store z1
movhps [ edi + 12 ], xmm5  // Store y2 x2
movss [ edi + 20 ], xmm5  // Store z2

// Store position & normal for vertex 2
movhps [ edi + 24 ], xmm6  // Store y3 x3
movss [ edi + 32 ], xmm6  // Store z3
movhps [ edi + 36 ], xmm2  // Store y4 x4
movss [ edi + 44 ], xmm2  // Store z4
```

Unfortunately, our target isn’t a set of four floats, so we had to store the upper two values (`movhps`) and the lowest value (`movss`) separately, instead of using the faster `movaps` to store our swizzled data.

To loop around, we simply need to increase our target and source pointers, decrement our loop counter and repeat until we reach zero:

```assembly
add edi, 48  // Skip two vertices
add edx, 16
add eax, 16
add ebx, 16

dec ecx
jnz swizzle_loop
```
Epilogue
Increasing processor speeds have led to the introduction of more complex algorithms that provide games with a more dynamic player environment, resulting in a decline in the use of vertex linear interpolation for animations. Additionally, vertex processing in hardware is increasingly replacing the need for software interpolation. However, most users do not have hardware vertex processors, so it’s important to optimize your software interpolation to support this gaming segment.

Furthermore, these optimization techniques apply equally to linear interpolation, which has many uses besides vertex interpolation.

The primary goal of this article is to introduce you to the worlds of SSE and assembly code and to provide you with the techniques to your code for deployment on Intel’s Pentium® III and Pentium 4 processors.

Links
http://developer.intel.com Intel’s premier site for hardware developers.
http://www.intel.com/IDS Intel’s premier site for web and software developers.

Recommended Reading
IA-32 Intel Architecture Software Developer’s Manual Volume 2: Instruction Set Reference


Acknowledgments
The author thanks Intel employees Sara Sarmiento and Kim Pallister for their assistance.

About the Author
Wayne Coles is a professional game developer from England, with a background in 6502, Z80 and 68000 assembly language. He spends most of his time coding in C++ and is a big fan of both Intel and Microsoft. He can be contacted by email at: nfactorial@btopenworld.com.