Optimized CPU-based Skinning for 3D Games

by Leigh Davies

Introduction

Lifelike 3D character models play an increasingly important role in many computer games. Organic models, such as people, are more complex to render than rigid bodies because the mesh that defines the shape of the model constantly changes as the model animates. This animating mesh is referred to as a 'skin' since it's influenced by the underlying structure of the object; 'skinning' is the process of animating this mesh. Traditionally done on the CPU, as model complexity increased, skinning has been done on the video card using vertex shader class hardware. However, there are advantages to performing skinning on the CPU, which this paper highlights. It also details an optimal way of CPU-based skinning using the floating point Streaming SIMD Extensions (SSE) instructions found on the Intel® Pentium® III processors and above. This optimized solution offers greater than double the performance of the initial C implementation, as well as a flexible and efficient alternative to vertex shader skinning. In addition, we will discuss how the addition of multi-threading support improves this optimized CPU skinning solution, as well as the nuances involved with multi-threading the skinning algorithm.

Background

Traditionally two methods are used to animate a skinned character mesh in a computer game. The first, called key-frame animation, involves saving the character in multiple poses, which are blended together as the model animates between the different positions. Key-frame animation is suitable for low polygon models and allows very fine control of the animation, however, on high-detail models it can be very memory intensive. The second method, palette matrix skinning, involves weighting the individual points of the character mesh to the bones of an underlying hierarchical skeleton. The advantage: just the skeletal animation needs to be saved, rather than the whole model in multiple positions.

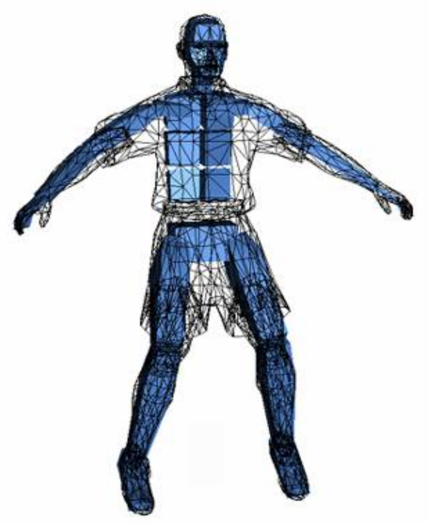


Figure 1 – Wire frame skinned character showing bone orientation.

Following are the properties of palette matrix skinning:

- For each pose a palette of matrices is created that represents the bone transformations.
- Each vertex in a skinned mesh is affected by one or more bones, depending upon a weight value.
- Usually weight values range between 0.0f and 1.0f. Depending on the distance from bone centre, this value denotes the amount of influence the bone's position t has on the final transformed vertex.
- Typically vertices have between 1 and 4 weights.

The two systems of animation are not mutually exclusive; frequently games will use a combination of both systems, with fine detail such as the face and hands animated using key-frames, and the remaining animations using matrix palette skinning. The remainder of this paper focuses on palette matrix skinning, given its advantages when dealing with the complex high polygon models of today's games.

Matrix Palette Skinning

Two main types of matrix palette skinning algorithms are used in computer games.

The first type stores the vertices to be used in the final mesh in bone-space; its properties include:

- Vertices those are stored relative to the bone that influences them. If the vertex is influenced by more than one bone it is stored multiple times, once per bone along with its weighting.

  Skinned Vertex = Pos1 * W1 * M1 + Pos2 * W2 * M2;

  M1 == World space transformation matrix for first bone of influence.
  Pos1 == Position of the vertex relative to the first bone.
  W1 == % weighting this bone has on the final position.
  M2 == World space transformation matrix for second bone of influence.
  Pos2 == Position of the vertex relative to the second bone.
  W2 == % weighting the second bone has on the final position.

- The animation hierarchy describes the orientation on the bones in world space.
- The CPU then transforms points by the animation hierarchy.
- The CPU then combines multiple weighted vertices back into a single vertex in world space.
- The model, which is now in world space, is submitted to DirectX or OpenGL in the same format as a static non-animating model. This enables use of hardware transformation and lighting when available.

System advantages include:
- Intuitive layout of data.
- A natural progression from un-skinned segmented animating models.

Disadvantages include:
- Data requirements grow rapidly as the envelope of the bones' influence grows and vertices become influenced by more bones.
- Cache usage is poor.
- This system is difficult to accelerate, as each vertex in the model's mesh is made up of several vertices, each stored in its own coordinate space.

The second system stores the vertices of the model in object space instead of world space, so that the points influenced by multiple bones need only to be stored once. The properties of this system include:

- The models vertices are exported in a default pose.
- An animation matrix hierarchy for this pose is exported.
- When the model is animated, the animation hierarchy describing the orientation on the bones in world space is multiplied by the inverse of animation matrix hierarchy for bind pose.

Skinned Vertex = Default Pos * W1 * M1 + Default Pos * W2 * M2;

Default Pos == Position of the vertex in object space.
M1 == Object space to world space transformation matrix for first bone of influence.
W1 == % weighting this bone has on the final position.
M2 == Object space to world space transformation matrix for second bone of influence.
W2 == % weighting the second bone has on the final position.

The advantages of this system include:

- Reduced memory footprint on models that have vertices weighted to numerous bones.
- Better cache usage when points are weighted to multiple bones.
- The data is stored in a format that allows acceleration on vertex shader-based video cards.

Disadvantages include:

- The animation stack that is generated in the game needs to be premultiplied by the inverse of the default pose.
- The matrix hierarchy used to transform the character no longer represents the bone positions and orientations making attachment of additional objects to the mesh more complex.

Why Do Skinning in Software?

With the second system suitable for hardware acceleration and the high theoretical vertex throughput of cutting edge GPUs, some may ask, “Why do skinning on the CPU?” Several compelling reasons exist:

- CPU skinning increases compatibility across a wide range of systems.
- The original hardware vertex shader specification used in DirectX* referred to as VS1.1 has a fixed limit of 96 instructions and can easily be exceeded when performing complex vertex operations.
- To utilize hardware in an efficient manner, the number of vertices that is processed in any one call needs to be maximised on current hardware (ATI 9800*, NVidia FX*). The batch sizes need to be between several hundred and several thousands; when skinning this can be hard to achieve due to the following:
    - The number of bones processed in any one call is restricted by vertex shader constant space. This means models that use more bones than can be stored in a single call require splitting into smaller batch sizes, reducing efficiency and increasing duplicate points.
    - The number of bones that influence each vertex must be static with each draw call; this can lead to wasted transformations on the GPU, as it needs to work to the worst-case scenario.

- Load balance: even where the GPU can transform the vertices faster than the CPU if the GPU is doing too much work while the CPU is sitting idle, moving work over the CPU can produce a performance increase.
- Many games require access to the post-transformed data for collision detection and shadow casting. Currently GPUs don't allow the data to be retrieved from the video card after it has been processed; therefore, allowing the CPU to do the first stage of the skinning can allow easy access to this data, while allowing the GPU to do the lighting and clipping calculations.
- The typical data formats for object space skinning allow the skinning to be optimized using SSE with very little work, giving surprisingly fast results. This allows models with 10,000+ polygons to be drawn at 60fps on a wide range of systems.
- A CPU skinned model can potentially benefit from GPU-accelerated lighting and clipping, using the fixed function transform and lighting hardware on many pre-vertex shader GPUs, which are still found on many systems.

Palette Skinning Algorithm

Following is a typical algorithm and data structure used to transform the vertices into world space on the CPU.

```
Class Mesh
{
        DWORD           m_WeightsPerVertex;
        DWORD           m_NumIndices;
        WORD*           m_Indices;
        DWORD           m_NumVertices;
        D3DXVector3*    m_pPositions;
        D3DXVector3*    m_pNormals;
        DWORD*          m_Diffuse;
        BYTE*           m_pBoneIndices;
        Float           m_pWeights;
}
```

Figure 2 – Typical data structure used to store the mesh.

For simplicity, assume that the number of weights affecting each point in the mesh is constant. Assuming a constant weight distribution allows for a direct comparison with hardware vertex shaders that inherently have this requirement, unless the model is drawn using multiple draw calls to split its vertices into groups organised around the number of number weights that affect the points. This is one area where skinning on the CPU has direct benefits compared with hardware systems, as there are much smaller overheads for processing the mesh in smaller batches that have been sorted to better reflect the properties of the individual points.

Below is the pseudo algorithm that uses the above data structure to transform the data. The algorithm is typical of those used in DirectX sample code and is efficient when m_WeightsPerVertex is a small value.

```
for(i=0;i<m_NumVertices;i++)
{
        Position.x = Position.y = Position.z = 0.0f;
        Normal.x = Normal.y = Normal.z = 0.0f;

        for(j=0;j<m_WeightsPerVertex;j++)
        {
                // Get weight and bone index.
                float Weight = m_pWeights[i*m_WeightsPerVertex+j];
                BYTE BoneIndex = m_pBoneIndices[i*m_WeightsPerVertex+j];

                // Multiply position by bone matrix, then weight and sum.
```

Figure 3 – Standard matrix palette skinning algorithm

The majority of the work done is the transform vertex position and transform vertex normal. These calculations are carried out once per bone, which influences the final point, therefore the amount of work done scales linearly with the increase in bones that affect the final vertex. Many modern lighting algorithms require that additional vectors need to be stored and transformed on a per-vertex basis, for effects such as tangent space per vertex bump mapping. The effect of this is per pixel lighting effects, which can become very expensive on animating skinned players and difficult to fit into the instruction space on first-generation GPU hardware.

A General Optimization for Complex Meshes

Optimizations should start at a high level, gradually working down to an instruction level when needed. The first optimization to the traditional skinning algorithm is to reduce the amount of work to be carried out on a per vertex basis,

Using a little algebra demonstrates that…

```
        Skinned Vertex = Pos * W1 * M1 + Pos * W2 * M2;
        Skinned Vertex = Pos * (W1 * M1 + W2 * M2);
        Skinned Vertex = Pos * Collapsed Matrix;
```

Figure 4

Computationally it is more efficient to perform the scalar matrix multiply and the matrix adds than to perform the original additional vector matrix multiply. Using the above algebra the original algorithm to transform a mesh can be rewritten as follows:

```
for(i=0;i<m_NumVertices;i++)
{
        Position.x = Position.y = Position.z = 0.0f;
        Normal.x = Normal.y = Normal.z = 0.0f;

        Memset(&MatrixPalette,0,sizeof(Matrix));
        for(j=0;j<m_WeightsPerVertex;j++)
        {
                // Get weight and bone index.
                float Weight = m_pWeights[i*m_WeightsPerVertex+j];
                BYTE BoneIndex = m_pBoneIndices[i*m_WeightsPerVertex+j];
```

Figure 5 – Matrix palette skinning algorithm using collapsed matrices

In the second algorithm the vertex position and normal need only be transformed once regardless of how many bones influence the final position. While this algorithm is computationally more expensive when only a single weight effects the final position, for greater than 2 weights per vertex this algorithm shows a significant reduction in the number of instructions compared to the original scheme. This general optimization for collapsing the matrices can be applied to both CPU skinning and GPU vertex shader skinning; this can be particularly important on GPU vertex shading with a limited number of instructions available in the shader. The advantages of collapsing the matrices increase as more weights effect the final position; in addition, any extra per-vertex calculations such as tangent space lighting calculations become much more efficient as their cost is no longer related to how many weights are affecting the point. In addition to the increased simplicity for complex meshes and lighting calculations, the second scheme benefits from its suitability for processing using the SIMD (Single Instruction Multiple Data) instruction set found on Intel Pentium III and above processors, as well as in vertex shaders.

Optimizing on the CPU Using SSE

SIMD instructions, introduced on the Intel Pentium III processor, are referred to as SSE. Additional instructions were made with subsequent architectures. Processors with Intel SSE support have 8 128-bit registers, and these registers can be used to store 4 single precision floating point numbers. SSE allows arithmetic, logical and load/store operations to be carried out on these 128-bit registers, which also allow all 4 floating point values to be processed using a single instruction. For maximum efficiency the data should be 16-byte aligned. One may program SSE instructions using inline Assembly or SSE Intrinsic, which allows the use of SSE Instructions directly from within C++ code. Intrinsic is supported by Visual Studio .Net and the Intel compiler.

When optimizing a skinning algorithm using SIMD, we use SSE’s ability to process up to 4 single precision floating point values, and rearrange the earlier algorithm to look like this.

```
for(i=0;i<m_NumVertices;i+=4)
{
            Collapse Matrix i;
            Collapse Matrix i+1;
            Collapse Matrix i+2;
            Collapse Matrix i+3;

            Rotate 4 Positions
            Rotate 4 Normals
            Normalise 4 Normals

            // Write the vertex out 4 vertices to the vertex buffer…
```

Figure 6 – Matrix palette skinning algorithm ordered for SSE.

SSE Attempt Number One
The first optimization attempt assumes an unfriendly SSE data layout. The structures used in the previous samples used an unaligned array of structures (AoS); we shall leave the data in this format and use a technique called “Gather & Scatter.” Its purpose: to pack the unaligned data into a 128-byte data type called __m128 which can be used by the intrinsic instructions. An example follows of how the gather function works:

This is calculation requires only six SSE instructions.

```
void LoadFourFloats(float* pIn0, float* pIn1, float* pIn2, float* pIn3, __m128* pOut)
{
            __m128 xmm0 = _mm_load_ss(pIn1);         // 0 0 0 pIn0
            __m128 xmm1 = _mm_load_ss(pIn2);         // 0 0 0 pIn1
            __m128 xmm2 = _mm_load_ss(pIn3);         // 0 0 0 pIn2
            __m128 xmm3 = _mm_load_ss(pIn4);         // 0 0 0 pIn3

            xmm0 = _mm_movelh_ps(xmm0, xmm2);   // 0 pIn2 0 pIn0
            xmm1 = _mm_shuffle_ps(xmm1, xmm3, _MM_SHUFFLE(0,1,0,1)); // pIn3 0 pIn1 0

            *pOut = _mm_or_ps(xmm0, xmm1); // pIn3 pIn2 pIn1 pIn0
}
```

Figure 7 – LoadFourFloats used to convert data into SSE-friendly formats.

A similar function called StoreFourFloats can be used to reverse the process and copy data from the SSE __m128 data type back into to unaligned data structures.

The first use of SSE is to collapse the matrices into a single 3x4 column major matrix; the translation into Column major is useful for the later vertex multiplication.

```
void CollapseMat(float* pM1, float* pM2, float W1, float W2,__m128* pR)
{
            __m128 xmm0,xmm1,xmm2,xmm3,xmm4,xmm5,xmm6;

            // Load and propagate the matrix weight 1.
            xmm0 = _mm_load_ss(&W1);
            xmm0 = _mm_shuffle_ps(xmm0,xmm0,0);

            // Load matrix 1, this loads an unaligned row major matrix
            LoadFourFloats(&pM1[0],&pM1[4],&pM1[8],&pM1[12],&xmm1);
            LoadFourFloats(&pM1[1],&pM1[5],&pM1[9],&pM1[13],&xmm2);
            LoadFourFloats(&pM1[2],&pM1[6],&pM1[10],&pM1[14],&xmm3);
```

Figure 8 – SSE instructions needed to collapse matrices

As we traverse the array of vertices four at a time, we use the LoadFourFloats function to rearrange the data so that batches of 4 X positions and 4 Y Positions are loaded into a single 128Bit SSE register. The Reshuffle function gathers data from the collapsed matrices, allowing the following calculation:

```
R1.x = M1.00 * P1.x + M1.10 * P1.y + M1.20 * P1.z + M1.30;
R2.x = M2.00 * P2.x + M2.10 * P2.y + M2.20 * P2.z + M2.30;
R3.x = M3.00 * P3.x + M3.10 * P3.y + M3.20 * P3.z + M3.30;
R4.x = M4.00 * P4.x + M4.10 * P4.y + M4.20 * P4.z + M4.30;
```

Figure 9 – Desired format for fast transformation of four vertices by the collapsed matrices.

```
 Void HD_4Vec4Mat(__m128* pM, float* pP1, float* pP2, float* pP3,  float* pP4,)
{

            // Load the input position components [POSITION].
            LoadFourFloats(&pP1[0],&pP2[0],&pP3[0],&pP4[0],&xmm0);
            LoadFourFloats(&pP1[1],&pP2[1],&pP3[1],&pP4[1],&xmm1);
            LoadFourFloats(&pP1[2],&pP2[2],&pP3[2],&pP4[2],&xmm2);

            ////////////////////////////////////////////////////////////////////////
            // Do the X's.

            // Load 1st column of each bone matrix.
```

Figure 10 – Transformation of four packed vertices by the collapsed matrix stack.

Using the reshuffled matrix data, the same is done for the vertex normals but without adding the transitional component. The final stage of re-normalizing the world space vertex works easily; with the X's, Y's and Z's already grouped, normalising four vertex normals requires just nine instructions.

```
xmm3 = _mm_mul_ps(xmm0, xmm0);      // X*X
xmm4 = _mm_mul_ps(xmm1, xmm1);      // Y*Y
xmm5 = _mm_mul_ps(xmm2, xmm2);      // Z*Z
xmm3 = _mm_add_ps(xmm3, xmm4);      // X*X + Y*Y
xmm3 = _mm_add_ps(xmm3, xmm5);      // X*X + Y*Y + Z*Z
xmm3 = _mm_rsqrt_ps(xmm3);          // 1 / sqrt(X*X + Y*Y +
Z*Z)
xmm0 = _mm_mul_ps(xmm0, xmm3);      // RecipLength * X
xmm1 = _mm_mul_ps(xmm1, xmm3);      // RecipLength * Y
xmm2 = _mm_mul_ps(xmm2, xmm3);      // RecipLength * Z
```

Figure 11 – Normalization routine for four packed normals.

SSE attempt number one increased speed 80% compared to the C implementation. (Figure 15 shows the exact frame rates and the configuration of the machine used for this test.)

SSE Attempt Number Two
For the second attempt, rather than start with the data in an AoS and convert on the fly to an SSE-friendly structure,we pre-process the data into a format referred to as a Structure of Arrays (SoA). This provides the following benefits:

- Data is guaranteed to be 16-byte aligned, allowing faster load and stores.
- Data is sequential in memory, therefore the Intel® Pentium® 4 processor hardware prefetch can begin streaming the vertex data into the caches.
- The SSE routines need not gather four X's from four separate structures, removing the need for "Gather" functions.

Drawbacks to the system include:

- Data arranged in this way is slightly counter-intuitive, and less programmer friendly
- Pre-processing is required if data was loaded in a traditional AoS format.

The ideal SSE-friendly data layout is 16-byte aligned sequential data. From the previous sample, we process X's separately to Y's and Z's, so an array of pre-gathered X's is perfect. Therefore we store the position and normal information in the following structure:

```
typedef struct _SSEVertexData
{
        __m128* pX;
        __m128* pY;
        __m128* pZ;
        __m128* pNX;
                                    __m128* pNY;
        __m128* pNZ;
}SSEVertexData;
```

Figure 12 – SSE-friendly mesh structure

In addition to the mesh information we also convert the matrix palette into a more SSE-friendly format:

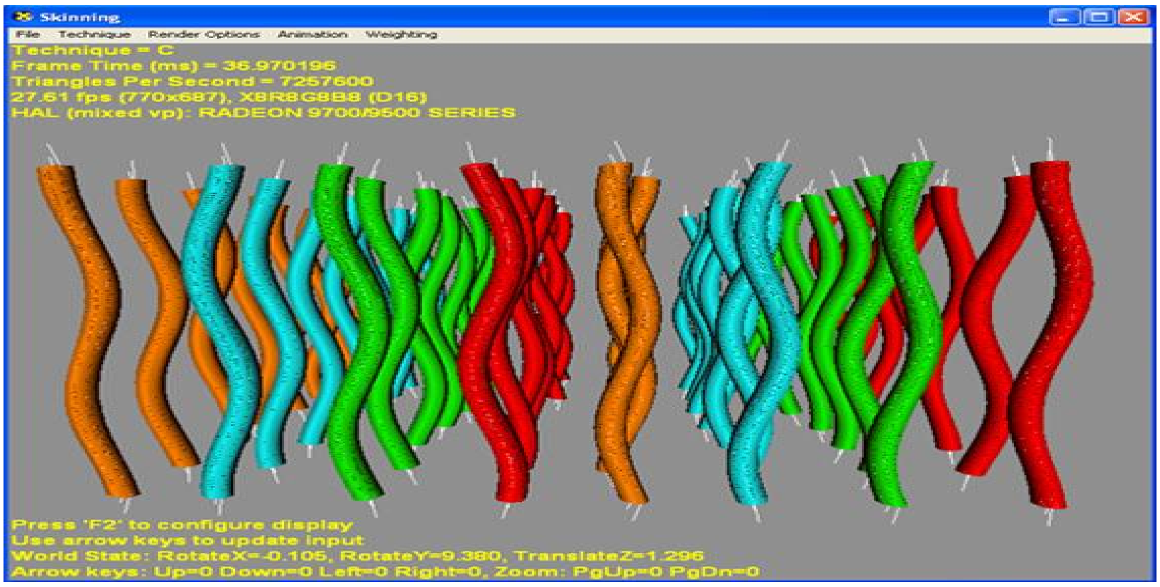
```
typedef struct _SSEMatrixData
{
            __m128 M[3];
}SSEMatrixData;
```

Figure 13 – SSE-friendly matrix data type.

Each matrix in the palette is converted to this format once per frame using the same reordering of data that was used in the original CollapseMat function. This provides significant saving over the approach in SSE 1 which converted between one and four matrices per vertex. SSE attempt number two increased speed by approximately 20% compared to the previous version, using unfriendly data. (Figure 15 shows details of this test.)

Demonstration Application

Based on the DirectX 9 API, the application shown in Figure 2 is based on the standard framework, and can compare the original algorithm along with the two SSE attempts. It also allows the data to be transformed using a DirectX vertex shader that runs in both Software and Hardware accelerated modes. The application creates 64 cylinders, each containing over 2000 vertices that have been weighted to 30 individual bones. The amount of weighing assigned to the vertices can be adjusted. With each cylinder drawn in a single DrawIndexedPrimitive call, the application is not limiting DirectX due to sub-optimal batch sizes. The application was compiled using the Intel Compiler V8.0.



```
Technique = C
Frame Time (ms) = 36.970196
Triangles Per Second = 7257600
27.61 fps (770x687), X8R8G8B8 (D16)
HAL (mixed vp): RADEON 9700/9500 SERIES
```

```
Press 'F2' to configure display
Use arrow keys to update input
World State: RotateX=-0.105, RotateY=9.380, TranslateZ=1.296
Arrow keys: Up=0 Down=0 Left=0 Right=0, Zoom: PgUp=0 PgDn=0
```

Figure 14 –Skinning Comparison Application

Controls while running the demo include:

- Arrow keys to control object rotation
- "Page Up" and "Page Down" keys allow the user to zone in and out.
- The F2 key configures the display properties.
- The menu allows the user to enable/disable the following:
    - Wire Frame mode, which can greatly affect frame rate on some hardware.
    - Display Bones, which again can affect frame rate as the bones are drawn using wireframe.
    - Select the amount of weighting used on the cylinders.
    - Select the current skinning mode.
    - Modify the type of animation performed on the cylinders.
    - Enable multi-threading of the animation update (discussed later in this paper.)

Figure 15 shows the results obtained from this application during a sample run.

	SW vertex shader	C	SSE	SSE (Friendly Data)	HW vertex shader
Frames Per Second	41	45	82	98	146
miliseconds	23	21	12	10	67

Figure 15 – Results of Skinning Comparison Application

System Config 3.0Ghz Intel® Pentium® 4 processor with Hyper-Threading Technology; 512MB RAM; ATI Radeon 9800 Pro; Processor Config BIOS switchable (SP, HT)*
Performance tests and ratings are measured using specific computer systems and/or components and reflect the approximate performance of Intel products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance.

Hyper-Threading Technology

Hyper-Threading Technology (HT Technology) was introduced to the Intel Pentium 4 processor. A HT Technology enabled PC exposes a second "logical" CPU within a single processor. Multiple processing tasks are completed more quickly on a system with HT Technology by executing two or more threads at the same time. From the point of view of both the operating system and the user, multiple tasks can be processed as if two actual processors were at work. To achieve maximum performance gains through Hyper-Threading, one must understand how HT Technology works.

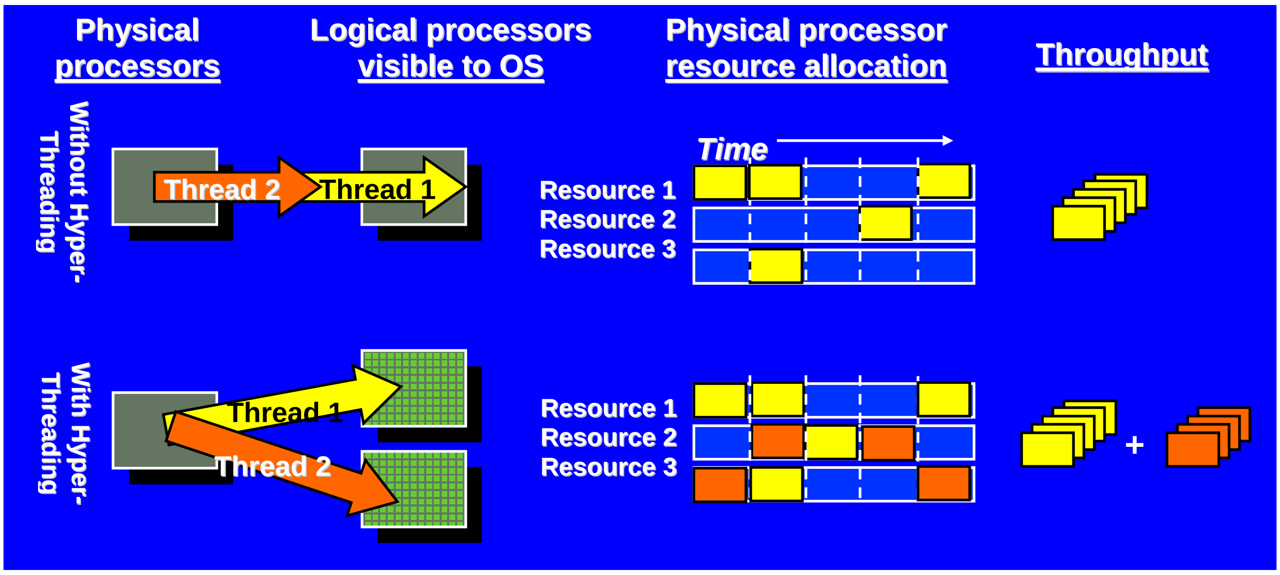


Figure 16: How Hyper-Threading Technology Works

Figure 16 highlights one of the biggest bottlenecks in a traditional system without HT Technology: at any one time the total throughput of the system is limited by dependencies between the different resources. On a system with HT Technology the total throughput is increased by allowing multiple threads to access the processor resources.

To increase the performance of the skinning application, we need to split the character-drawing work across multiple threads. We could create the multiple threads using win32 but OpenMP* is also an alternative.

What is OpenMP?

The OpenMP Application Program Interface (API) supports multi-platform shared-memory parallel programming in C/C++ and FORTRAN on a variety of architectures. OpenMP is jointly defined by a group of computer hardware and software vendors, andin the consumer games arena it is currently supported by Intel compiler. OpenMP allows easy multithreading, ideally WITHOUT changing the original C/C++ code. (See www.OpenMP.org for samples and details of the OpenMP specification.)

There are three components of the OpenMP complier extension:

- #pragma's (compiler directives), the– most important
- API and Runtime Library
- Environment Variables

Benefits include the provision of:

- An easy way to use HT Technology
- A portable and standardized method
- Dedicated profiling tools

OpenMP works on a Fork-Join model. At startup the *main* thread creates a *team* of additional threads. Whenever the application enters a parallel region declared using **#pragma omp** the statements that are enclosed by the parallel region construct are then executed in parallel among the various team threads. They synchronize at the end of the parallel region construct, leaving only the master thread active.

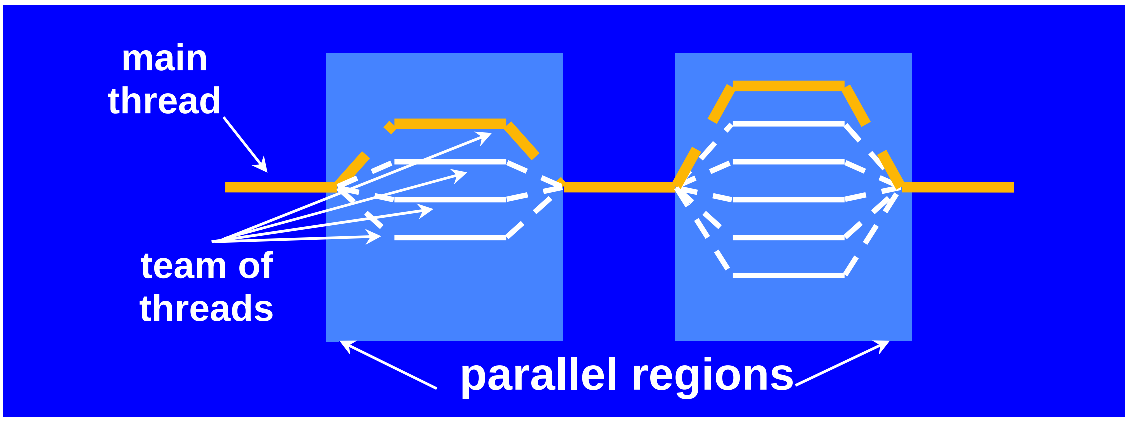


Figure 17: Fork-Join Parallelism

Data Level Parallelism
OpenMP is designed to allow two types of parallel programming: the first, data level parallelism, splits the work being done on a set of data (in our case the vertices of the model between the available processors). A sample of how this would work follows.

```
#pragma omp parallel for
        for(i=0;i<m_NumVertices;i++)
        {
                // Skin the vertices.
        }
```

Unfortunately, this technique did not improve performance, because:

- When tested on a dual processor system only a minimal performance increase occurred. This, when combined with the Intel VTune™ Performance Analyzer profile of the application, showed that level 2 cache misses are high, a good indication that the application is limited by memory bandwidth.
- SSE units are well utilized. Referring back to Figure 16, if one resource is already nearly fully utilized, then threading code that requires similar resources will not increase the total throughput of the system.

Task Level Parallelism
The second type is task level parallelism, used in the skinning application. Rather than splitting the work done on the vertices across the available processors, we perform different tasks in parallel. For the skinning demo, calculations for the next frame's animation were processed in parallel while calculating the current frames vertices.

The pragma to do this is shown below:

```
#pragma omp parallel sections
{
        #pragma omp section
                m_pCylinder->UpdateAnim(m_fTime + m_fElapsedTime);
        #pragma omp section
                m_pCylinder->UpdateSkin();
}
```

When splitting the work across multiple threads, one important rule is that the data being modified by one thread should *not* be used by the other thread unless synchronization pragmas are used to protect from race conditions. For the matrix palette skinning sample, the matrix palettes were double buffered to allow task level parallelism to take place.

Figure 18 shows the results obtained from this application during a sample run:

	Sine Wave	Perlin Noise 10 Octaves	Perlin Noise 20 Octaves
Single Threaded	98.6	80.6	66.7
Multi- Threaded	103.2	95.26	87
Percentage Increase	4.6%	18.1%	30.4%

Figure 18 – Results of Skinning comparison Application, where the application was performing SSE friendly skinning during all of the tests.

The skin update overlaps the following frames animation update. Generating the next animation frame using simple sine waves to animate the matrices produces a slight performance increase. If we change to using a more mathematically complex operation, just as Perlin noise, to animate the matrices we start to see much improved performance. In a real-world application the additional work performed to generate the animation might be the result of decompressing packed animation data, or blending multiple animation frames or even applying inverse kinematics to the animation hierarchy. The reason for the improved threading performance in the more complex animation system is the load balance between the two parallel tasks. The largest performance increases occur in multi-threading when performing two tasks, in parallel, that take a similar amount of time to complete.

Thread Profiler
Greater multi-threading performance gains can be achieved when the tasks running in parallel are well balanced (Figure 18). To see if this applies in an application, a new type of profiling tool is needed. The Intel thread profiler is an application that plugs into the Intel® VTune™ Performance Analyzer and can profile the effects of threading on a program. The thread profiler shows the amount of parallelism an application is achieving, and any potential threading overheads. Version 2.0 of the profiler added the ability to profile win32 threads in addition to applications written using OpenMP. The thread profile allows the user to obtain information on the load balance of the parallel regions, and any overheads associated with running the code in parallel.

Figure 19 shows the output of the multi-threaded sine wave animation run, with the transformation done using SSE-friendly data. Figure 20 shows similar data from a run conducted with animation generated using 20 Octaves of Perlin noise.

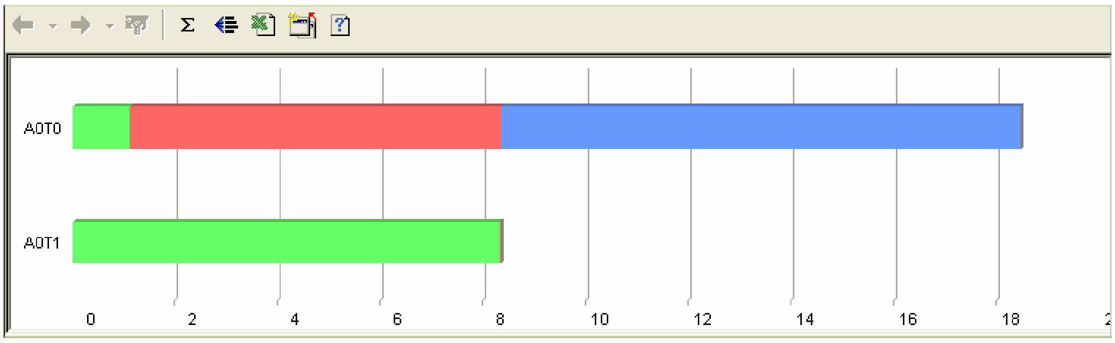


Figure 19, Thread Profile output for sine wave-generated animation

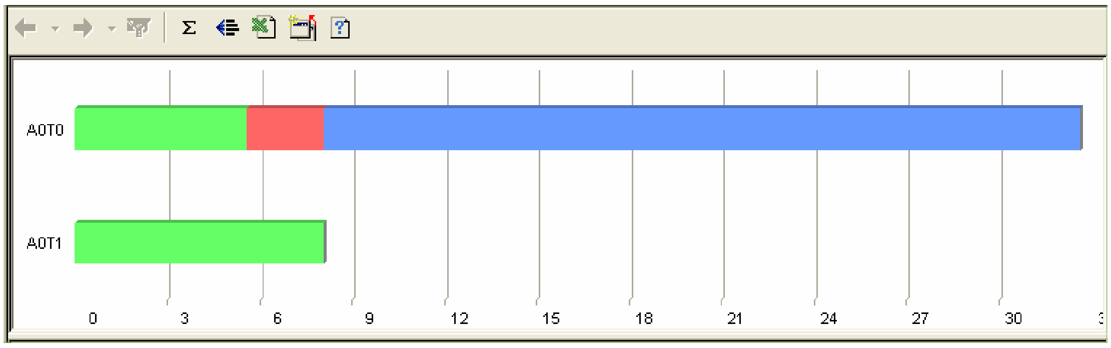


Figure 20, Thread Profile output for Perlin noise-generated animation

With the red area in the image denoting the load imbalance, Figure 19 clearly shows that when a sine wave is used to generate the animation, the work load imbalance between the two parallel regions in the code is much greater than in the second example, using the Perlin noise.

Conclusion

The hardware vertex shaders offer the fastest method of performing matrix palette skinning, but the data-friendly SSE version offers a significant improvement over standard C implementations of skinning algorithms. In the case where hardware T&L is available (such as Geforce2* and Geforce4MX* class hardware), the second SSE version is more than twice as fast as a software vertex shader, because the application uses the hardware lighting and clipping provided by the GPU. Even on the high-end GPU with vertex processing, a good case may exist for using a SSE skinning technique if one requires access to the transformed data, for silhouette generation for stencil shadow casting.

While SSE provided a way to optimize the existing palette skinning algorithm using data level parallelism, Hyper-Threading Technology is best suited to a higher-level, task level parallelism. Multi-threading load imbalance and resource contention can give unexpected results, and one must try different combinations of load-balancing to find the best solution. You should seriously consider threading objects in a game, as the potential gains will continue to increase on future hardware.
Taking a coarse-grained approach to threading is frequently the key to getting the biggest improvement from Hyper-Threading Technology, as it reduces resource contention and allows for a greater degree of parallelism with the application.

Download source code

Future Work

DirectX 9c supports an additional extension to the Vertex Shader model referred to as VS3.0, removing the need for the output of the vertex shader to be tied to a FVF vertex format and possibly providing an alternative to writing SSE routines in order to maximise utilisation of the CPU.

Although the skinned cylinders used in the demo application demonstrate well the principles of matrix palette skinning, they differ from typical real world data in one area: they have even weight distribution over all their points. Expanding the application to allow the distribution of weights to vary within a single model would be more realistic, allowing the CPU skinning systems to make use of their greater ability to dynamically change how they process the data based on its properties within a single model.

The CPU skinning algorithms use only the basic SSE instruction set. Further research into using the addition instructions available under SSE3, such as horizontal add within a 128-bit register, might provide additional improvements to the CPU skinning algorithm.

Acknowledgments
I would like to give special thanks to Jon Story and Aaron Coday for much of the initial research into optimising skinning; they laid much of the groundwork for this paper.

References

- Intel® Software College
- MSDN, Streaming SIMD Extensions (SSE).
- Official OpenMP Web site
- Threading Methodology: Principles and Practice
- Intel Threading Tools and OpenMP

About the Author

Leigh Davies is an application engineer with Intel's Developer Relations Division. Prior to joining Intel he worked for six years as a senior PC programmer at a UK-based games company specializing in writing DirectX applications.