CPU Animation

Introduction

The importance of real-time character animation has greatly increased in modern games. Animating meshes via ‘skinning’ can be performed on both a general purpose CPU and a more specialized GPU. However, to distribute the workload and maximize efficiency on a processor graphics system, skinning can be performed on the CPU and lighting and clipping computations can be performed on the GPU. One of the advantages of performing CPU skinning is that the animated post transformed data are readily available for collision detection. Also multicore and vectorization with Intel® Advanced Vector Extensions (Intel® AVX) can be used to improve performance of the skinning algorithm on an AVX capable CPU.

The objective of this sample is to provide best known methods to perform CPU skinning on an AVX capable processor graphics system. The sample compares performance of linear blend skinning algorithm on the CPU and GPU. There are 3 different skinning techniques implemented in the sample:

1. **CPUSkin Scalar**: This technique onloads the CPU and performs character animation by skinning the models on the CPU.
2. **CPUSkin AVX**: This technique optimizes the onloaded CPU skinning by using Intel AVX instructions.
3. **GPUSkin**: This technique implements GPU character animation by skinning the models in the vertex shader.

There is one dragon model whose mesh is skinned to perform one of 4 animations: namely a fly loop, a run loop, a glide loop, and a run fly run loop. The dragon model has 14,528 vertices and each vertex is influenced by a maximum of 4 bone matrices.

CPU skinning

This section provides more details on each of the CPU skinning techniques.

**CPUSkin Scalar:**

The animated position, normal and tangent for a vertex is computed by transforming the default bind-pose position, normal and tangent for that vertex by a set of weighted matrices that influences it. The weights reflect the extent of influence each bone matrix has on the final animated vertex. For example if M0, M1, M2 and M3 are the four bone matrices that influence vertex ‘v’ and W0, W1, W2 and W3 are the weights respectively then, the new animated position, normal and tangent is computed as shown above. $\text{Pos}_v, \text{Norm}_v \text{ and } \text{Tan}_v$ are the bind-pose position, normal, and tangent for vertex ‘v’.
\[ \text{animated} \text{Pos}_v = \text{Pos}_v \times W_0 \times M_0 + \text{Pos}_v \times W_1 \times M_1 + \text{Pos}_v \times W_2 \times M_2 + \text{Pos}_v \times W_3 \times M_3 \]
\[ \text{animatedNormal}_v = \text{Norm}_v \times W_0 \times M_0 + \text{Norm}_v \times W_1 \times M_1 + \text{Norm}_v \times W_2 \times M_2 + \text{Norm}_v \times W_3 \times M_3 \]
\[ \text{animatedTangent}_v = \text{Tan}_v \times W_0 \times M_0 + \text{Tan}_v \times W_1 \times M_1 + \text{Tan}_v \times W_2 \times M_2 + \text{Tan}_v \times W_3 \times M_3 \]

**CPUSkin AVX**

The CPU skinning algorithm was optimized by implementing three different methods using Intel AVX as listed below and method 3 which provided the best performance was retained. This method is an extension of the Intel® SSE implementation explained in (Waveren, 2005).

1. Use Intel AVX to animate 8 vertices at a time.
2. Use Intel AVX to animate 2 vertices at a time (implement the Intel SSE version on 2 vertices at a time).
3. Use Intel AVX to optimize the matrix transformation and work on only 1 vertex at a time.

The CPUSkin AVX technique uses Intel AVX to transform the default bind pose position, normal and tangent for each vertex with the weighted sum of matrices that influences the vertex. Intel AVX is used to vectorize the matrix transformation while computing the animated position as shown below. Similarly the animated normal and tangent is computed.

\[ \text{animated} \text{Pos}_v = \text{Pos}_v \times \left( W_0 \times M_0 + W_1 \times M_1 + W_2 \times M_2 + W_3 \times M_3 \right) \]
\[ \text{animated} \text{Pos}_v = \text{Pos}_v \times \text{WeightedMatrix} \]
where \( \text{WeightedMatrix} = \left( W_0 \times M_0 + W_1 \times M_1 + W_2 \times M_2 + W_3 \times M_3 \right) \)

Let \( \text{WeightedMatrix} = \begin{bmatrix} m_{00} & m_{01} & m_{02} & m_{03} \\ m_{10} & m_{11} & m_{12} & m_{13} \\ m_{20} & m_{21} & m_{22} & m_{23} \\ m_{30} & m_{31} & m_{32} & m_{33} \end{bmatrix} \)
Use AVX to vectorize the matrix multiply

Let $\mathbf{Pos}_v = [x \ y \ z \ 1]$ and let $\mathbf{animatedPos}_v = [x' \ y' \ z' \ w']$

Substituting in

$\mathbf{animatedPos}_v = \mathbf{Pos}_v \times \mathbf{WeightedMatrix}$

$$
\begin{bmatrix}
x' \\
y' \\
z' \\
w'
\end{bmatrix} =
\begin{bmatrix}
x \\
y \\
z \\
1
\end{bmatrix} \times
\begin{bmatrix}
m_{00} & m_{01} & m_{02} & m_{03} \\
m_{10} & m_{11} & m_{12} & m_{13} \\
m_{20} & m_{21} & m_{22} & m_{23} \\
m_{30} & m_{31} & m_{32} & m_{33}
\end{bmatrix}
$$

$\mathbf{tmp} = 
\begin{bmatrix}
x \\
y \\
z \\
1
\end{bmatrix} \times
\begin{bmatrix}
m_{00} & m_{01} & m_{02} & m_{03} \\
m_{10} & m_{11} & m_{12} & m_{13} \\
m_{20} & m_{21} & m_{22} & m_{23} \\
m_{30} & m_{31} & m_{32} & m_{33}
\end{bmatrix} + 
\begin{bmatrix}
x \times m_{00} + y \times m_{10} \\
x \times m_{01} + y \times m_{11} \\
x \times m_{02} + y \times m_{12} \\
x \times m_{03} + y \times m_{13}
\end{bmatrix}
\begin{bmatrix}
z \times m_{20} + m_{30} \\
z \times m_{21} + m_{31} \\
z \times m_{22} + m_{32} \\
z \times m_{23} + m_{33}
\end{bmatrix}

Extract the 4 lower FPs and 4 higher FPs and add them together

$$
\begin{bmatrix}
x' \\
y' \\
z' \\
w'
\end{bmatrix} =
\begin{bmatrix}
x \times m_{00} + y \times m_{10} \\
x \times m_{01} + y \times m_{11} \\
x \times m_{02} + y \times m_{12} \\
x \times m_{03} + y \times m_{13}
\end{bmatrix} + 
\begin{bmatrix}
z \times m_{20} + m_{30} \\
z \times m_{21} + m_{31} \\
z \times m_{22} + m_{32} \\
z \times m_{23} + m_{33}
\end{bmatrix}
$$

**GPU Skinning**

This technique skins the models in the vertex shader on the GPU. Each instance of the model uses a unique set of bone transformation matrices. The set of matrices for all the instances are consolidated and sent to the GPU via a single ‘boneMatrix’ dynamic buffer. The boneMatrix buffer is updated each frame with the matrix information for each instance. The vertex buffer contains the default bind pose information. Employing instancing the vertex shader uses the bind pose information from the vertex buffer and the matrix information for each instance from the boneMatrix buffer and uniquely skins each model instance on the GPU.

**Optimizations for CPU skinning**

There are several optimizations implemented in the CPUAnimation sample.

- Multi-threading

  The sample uses the task manager created by (Minadakis, 2011) for multi-threading the CPU skinning update. The task manager is used as it abstracts and simplifies the task creation and
synchronization process. When multi-threading is enabled in the sample the task manager creates a set of tasks: one task for each dragon instance in the scene. Each task is responsible for completing the CPU skinning update for a single dragon instance in the scene.

- **Vectorizing with Intel AVX**
The CPU skinning algorithm is optimized by using Intel AVX compiler intrinsics. Please refer to CPU Skinning section for more details on how the CPU skinning algorithm is vectorized.

- **Pipelining /double buffering**
As the skinning calculations for each frame is independent of the previous frame, the CPU skinning update can be pipelined by using double buffers. Double buffering is used so that skinning update and rendering can be performed simultaneously. One of the buffers stores the animation information for the current frame and is used by the main thread for rendering. The other buffer stores animation information for the next frame and is updated by the animation thread.
When pipelining is enabled the user can choose how often the CPU skinning update should be performed. For example the user can choose to perform CPU skinning once every 3 frames. This means one of the buffers is mapped once every 3 frames for skinning update and in the meantime the other buffer is used to render the scene. At the end of the 3 frames the application makes sure that the skinning update is complete and swaps the two buffers so that the newly updated animation can be used to render the scene.

**Sample Usage**
Figure 1 shows a screenshot of the CPUAnimation sample. The sample loads with 50 dragon instances, with each instance performing one of the 3 animations. Use the combo box to choose from one of the 3 ‘Skinning Techniques’.

Use the ‘Instances’ slider to vary the number of dragons in the scene. The maximum number of dragons in the scene is set to 200.

The ‘Enable Pipeline’ checkbox enables / disables the ability to use double buffering when performing the CPU skinning update. Pipelining can be enabled only when multi-threading is enabled in the sample.

Use the ‘Frames CPU Skin spans’ slider to choose how often the CPU skinning updates should be performed. If the ‘Frames CPU Skin spans’ slider is set to 2 the CPU skinning update is performed once every 2 frames. When varying the ‘Frames CPU Skin spans’ slider make sure the vertices are skinned on the CPU at least ~30 times per second.

The ‘Animate’ checkbox enables / disables animation in the scene.

The ‘Multi-Threading’ checkbox enables /disables Intel® TBB optimizations and the ‘Wireframe’ checkbox enables / disables wireframe fill mode in the scene.
The total number of vertices skinned on the CPU/GPU, the time taken to complete a skinning update on the CPU and the frame time minus the time spent on the draw call for CPU skinning are displayed in the top left corner of the screen.

**Performance**

The performance for the CPUAnimation sample is measured on a 2.2GHz Intel microarchitecture code name Sandy Bridge-based system which has 4 cores / 8 threads. The time is measured in milliseconds. Skinning time is the time taken to complete a single CPU skinning update for all the dragons in the scene and Frame time is the time taken to complete skinning and rendering a single frame.

Table 1 shows the performance for CPU skinning when the application is run in single threaded mode and Table 2 shows the performance when the application is run in multi-threaded pipeline enabled mode with CPU skinning performed once every frame.
### Table 1: Skinning performance in single threaded mode

<table>
<thead>
<tr>
<th>#of dragons</th>
<th># of vertices Skinned</th>
<th>CPU Skin Scalar</th>
<th>CPU Skin AVX</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Skinning time (ms)</td>
<td>Skinning time (ms)</td>
</tr>
<tr>
<td>1</td>
<td>14,528</td>
<td>0.87</td>
<td>0.26</td>
</tr>
<tr>
<td>10</td>
<td>145,280</td>
<td>8.53</td>
<td>2.55</td>
</tr>
<tr>
<td>50</td>
<td>726,400</td>
<td>41.05</td>
<td>12.26</td>
</tr>
<tr>
<td>100</td>
<td>1,452,800</td>
<td>81.56</td>
<td>24.21</td>
</tr>
<tr>
<td>150</td>
<td>2,179,200</td>
<td>123.00</td>
<td>35.07</td>
</tr>
<tr>
<td>200</td>
<td>2,905,600</td>
<td>156.81</td>
<td>46.85</td>
</tr>
</tbody>
</table>

### Table 2: Skinning performance in multithreaded, pipeline enabled mode with CPU skinning performed once every frame

<table>
<thead>
<tr>
<th>#of dragons</th>
<th># of vertices Skinned</th>
<th>CPU Skin Scalar</th>
<th>CPU Skin AVX</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Skinning time (ms)</td>
<td>Skinning time (ms)</td>
</tr>
<tr>
<td>1</td>
<td>14,528</td>
<td>0.88</td>
<td>0.28</td>
</tr>
<tr>
<td>10</td>
<td>145,280</td>
<td>3.02</td>
<td>0.96</td>
</tr>
<tr>
<td>50</td>
<td>726,400</td>
<td>11.77</td>
<td>4.1</td>
</tr>
<tr>
<td>100</td>
<td>1,452,800</td>
<td>23.64</td>
<td>7.42</td>
</tr>
<tr>
<td>150</td>
<td>2,179,200</td>
<td>34.88</td>
<td>11.37</td>
</tr>
<tr>
<td>200</td>
<td>2,905,600</td>
<td>46.62</td>
<td>15.4</td>
</tr>
</tbody>
</table>

From Table 1 and Table 2 we can see that for CPU Skin Scalar there is a performance gain of 180% – 250% and for CPU Skin AVX there is performance gain of 165% – 230% depending upon the number of vertices skinned on the CPU in the scene.

Figure 2 and Figure 3 compares the performance for single threaded and multithreaded pipeline enabled modes for CPU Skin Scalar and CPU Skin AVX techniques respectively.

From Table 2 we can see that there is a performance gain of 190% – 220% from CPU Skin Scalar to CPU Skin AVX technique depending upon the number of vertices skinned on the CPU in the scene.

Figure 4 compares the performance of CPU Skin Scalar and CPU Skin AVX for the multithreaded pipeline enabled mode.

Table 3 shows the frame time for CPU and GPU skinning techniques. Frame time for the CPU skinning techniques was measured when the sample was run in a multithreaded pipeline enabled mode. Irrespective of the technique, skinning was performed once every frame.
Figure 2: Compares skinning time performance of CPU Skin Scalar technique in single threaded and multithreaded pipeline enabled mode.

Figure 3: Compares skinning time performance of CPU Skin AVX technique in single threaded and multithreaded pipeline enabled modes.
From Table 3 we can see that there is a performance gain of 17% – 160% from GPU Skin technique to CPU Skin AVX technique depending upon the number of vertices skinned in the scene.

Figure 5 compare the frame time performance for the 3 skinning techniques.

![Skinning time for Scalar vs. AVX](image)

Table 3: Frame time performance for multithreaded pipeline enabled mode with CPU skinning performed once every frame
Table 4: Frame time performance for multithreaded pipeline enabled mode with CPU skinning performed once every ‘N’ frames

<table>
<thead>
<tr>
<th># of dragons</th>
<th># of vertices Skinned</th>
<th>CPUSkin Scalar</th>
<th>CPU Skin AVX</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Frames CPU Skin spans</td>
<td>Frame Time</td>
</tr>
<tr>
<td>1</td>
<td>14,528</td>
<td>10</td>
<td>1.65</td>
</tr>
<tr>
<td>10</td>
<td>145,280</td>
<td>10</td>
<td>2.83</td>
</tr>
<tr>
<td>50</td>
<td>726,400</td>
<td>3</td>
<td>8.88</td>
</tr>
<tr>
<td>100</td>
<td>1,452,800</td>
<td>1</td>
<td>23.29</td>
</tr>
<tr>
<td>150</td>
<td>2,179,200</td>
<td>1</td>
<td>34.28</td>
</tr>
<tr>
<td>200</td>
<td>2,905,600</td>
<td>1</td>
<td>45.67</td>
</tr>
</tbody>
</table>

Table 4 shows the time taken to complete a single frame in multithreaded pipeline enabled mode when CPU skinning is performed once every ‘N’ frames by setting ‘Frames CPU Skin spans’ slider to a value > 1.
There are certain cases in which CPU skinning has to be performed once every frame so that there are at least ~30 animation updates per second. For such cases Table 4 has the same value as in Table 3.

From Table 3 and Table 4 we can see that for CPU Skin Scalar there is a performance gain of 17% – 27% and for CPU Skin AVX there is a performance gain of 10% – 14% depending upon the number of dragons in the scene.

Figure 6 and Figure 7 compares the frame time performance when CPU skinning is performed once every frame and when it’s performed once every ‘N’ frames.

![Frame time for CPU Skin Scalar](image)

*Figure 6: Compares performance of CPU Skin Scalar technique when skinning is performed once every frame and when skinning is performed once every ‘N’ frames*

**Conclusions**
- From Table 3 and Figure 6 we can see that CPU Skinning is clearly the winner as compared to GPU skinning on an IVB system.

**Future Work**
There is an opportunity to implement more optimizations for CPU skinning as listed below:
• Reorder the vertices so that all vertices that use the same set of matrices for skinning are contiguous in the list. By doing so, we can avoid reloading the matrices multiple times.
• Once the Intel® Advanced Vector Extensions 2 (Intel® AVX2) instruction set is released, we can use fused multiply adds in the Intel AVX optimization code.

**Figure 7:** Compares performance of CPU Skin Scalar technique when skinning in performed once every frame and when skinning is performed once every 'N' frames

<table>
<thead>
<tr>
<th>Time in milliseconds</th>
<th>Number of vertices skinned on the CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.63</td>
<td>14528</td>
</tr>
<tr>
<td>1.64</td>
<td>145280</td>
</tr>
<tr>
<td>2.72</td>
<td>726400</td>
</tr>
<tr>
<td>8.34</td>
<td>1452800</td>
</tr>
<tr>
<td>16.92</td>
<td>2179200</td>
</tr>
<tr>
<td>25.16</td>
<td>2905600</td>
</tr>
<tr>
<td>33.41</td>
<td></td>
</tr>
</tbody>
</table>

**CPU Skin AVX** (lower is better)

**References**


