Efficient Animation

Cem Cebenoyan

NVIDIA Corporation

cem@nvidia.com

Agenda

- Keyframe Interpolation Techniques
- Skinning Techniques
- Integration with Per-Pixel Lighting
- Optimization Issues

Keyframe Interpolation Idea

- Given a series of stored vertex positions, we blend between them depending on a parameter, most likely time.
- Can blend other attributes (like normals or texture space bases) in the same fashion.
- This blending can occur entirely on the GPU using vertex shaders and multiple streams.

Keyframe Interpolation

Linear Keyframe Interpolation

- The equation for linear keyframe interpolation is simply:

$$v_{final} = v_0 \cdot (1 - t) + v_1 \cdot t$$

- Where the final vertex position is a blend between vertex position 0 and vertex position 1 using the parameter t.

Linear Keyframe Interpolation

- In DirectX8, we can achieve this by setting the current two keyframes we’re blending between into separate streams, setting the blend parameter as a constant, and doing the blending in the shader using just two instructions:

```
mul r0, v0, c[BLEND_FACTOR].y
mad r0, v1, c[BLEND_FACTOR].x, r0
```

Keyframe Interpolation

- **The flexibility of vertex shaders allows us to use more complex blends in the vertex shader than just a simple linear interpolation.**
- **Contrast this with the fixed function tweening available in DirectX8 through the renderstate D3DRS_TWEENFACTOR, which is limited to a straight linear interpolation only.**

Hermite Spline Interpolation

- You can blend between keyframes using a subset of hermite splines known as Catmull-Rom splines.

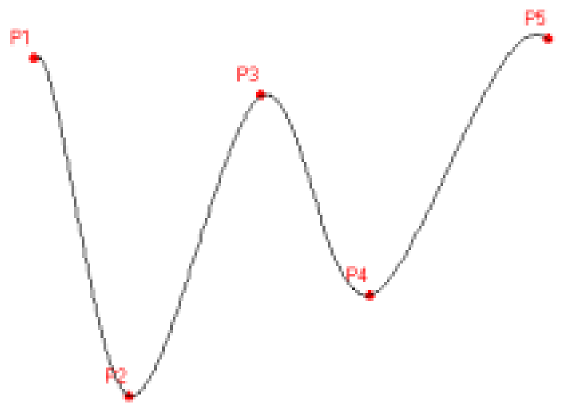
- Catmull-Rom splines are guaranteed to pass through the values they’re interpolating, making them ideal for keyframe interpolation.

Hermite Spline Interpolation

- Hermite spline interpolation uses the following scary equation:

$$v_{final} = h_0 \cdot v_i + h_1 \cdot (v_{i+1} - v_{i-1}) + h_2 \cdot (v_{i+2} - v_i) + h_3 \cdot v_{i+1}$$

- Example spline:



Hermite Spline Interpolation

- ***H*** **are the following cubic hermite basis functions:**

$$h_0 = 2t^3 - 3t^2 + 1$$

$$h_1 = -2t^3 + 3t^2$$

$$h_2 = t^3 - 2t^2 + t$$

$$h_3 = t^3 - t^2$$

- **Note that hermite splines are a function of *four* vertex positions, so they require us to use four separate streams, two positions before the current and two after.**

Hermite Spline Interpolation

- **Spline interpolation gives a smoother blend, but at a cost.**
- **Requires more instructions to do the blend in the vertex shader.**
- **Requires twice the vertex data to be fetched by the GPU.**
- **Can potentially reduce the memory requirements since you can have greater space between keyframes, and thus fewer keyframes overall.**
- **Watch for non-looping animations.**

Hermite Spline Interpolation

- Note that it may be worthwhile to store vertex attributes that don’t need to be interpolated, like texture coordinates, in separate streams. This is especially valuable for techniques that use many streams, like spline interpolation.
- Saves duplication of data in your vertex buffers.
- Saves extra data being fetched over the AGP bus.
- Can potentially be slower, though, due to stream inefficiencies. Test and find out.

Skinning Techniques

- There are a number of skinning techniques supported by DirectX8:
  - DirectX 7-style skinning
  - Fixed-function matrix palette skinning
  - Vertex shaders

DirectX7-style Skinning

- Specified through D3DRS_VERTEXBLEND.
- Not indexed, you get a limit to the number of bones *per call to DrawPrimitive()*.
- The limit on GeForce and GeForce2 is 2 bones, GeForce3 ups this to 4 bones, the API limit.

DirectX7-style Skinning

- Advantages
  - Very fast
- Disadvantages
  - Kills batching for models with a large number of bones.
  - Limited number of bones available.
  - Doesn’t integrate well with per-pixel lighting.

- Conclusion: use it when it’s sufficient, which may be rare due to the many limitations.

Fixed Function Matrix Palette Skinning

- New for DirectX8, allows you to specify up to four indices in your vertex format and index a palette of up to 256 matrices.
- Meant to address the number of bone limitations of the DirectX7-style of skinning.

FF Matrix Palette Skinning

- Advantages
  - Simple to use.
- Disadvantages
  - No current hardware supports it. And any hardware that will support it in the future will likely have much fewer than 256 matrices available.
  - The current software implementation in DirectX8 isn't optimal.
  - Doesn't integrate well with per-pixel lighting.
  - Limits you to four bones per vertex.
- Conclusion: not the most flexible or high performance skinning option.

Vertex Shader Matrix Palette Skinning

- Our choice: do matrix palette skinning in a vertex shader.
- Advantages:
  - Fully supported in hardware by GeForce3, and any future DirectX8 hardware.
  - Has a very fast software fallback through the DirectX8 runtime.
  - Supports a flexible number of bones per vertex.
  - Integrates well with per-pixel lighting.

VS Matrix Palette Skinning

- Disadvantages:
  - Potentially more complex, but we have sample code and NVLink to help.
  - Requires you to break up your mesh into sections influenced by a certain number of bones, but it shares this limitation with all hardware-accelerated skinning solutions.
- Conclusion: Vertex Shaders are the most flexible and efficient way to do skinning on the GPU.

The Address Register

- Vertex shader indexed matrix palette skinning relies on using the address register a0.x
- It allows you to offset into the constant memory a variable amount.
- Note that it can only be the destination of a mov instruction:

```
mul a0.x, r0, r1  // ERROR!
```

- Also, there is only an a0.x, no a0.y, a0.z, etc.

NVIDIA

VS Matrix Palette Skinning

- **Implementation details:**
- **Vertex declaration for two bones per vertex:**

```
DWORD dwDecl[] =
{
     D3DVSD_STREAM(0),
     D3DVSD_REG(0,  D3DVSDT_FLOAT3),     // position
     D3DVSD_REG(1,  D3DVSDT_FLOAT3),     // normal
     D3DVSD_REG(2,  D3DVSDT_FLOAT2),     // two weights
     D3DVSD_REG(3,  D3DVSDT_SHORT2),     // two indices
     D3DVSD_END()
};
```

VS Matrix Palette Skinning

- **Actual vertex shader code:**

```
//load first index
mov a0.x, V_INDICES.x

//transform position by first bone, store in r0
dp4 r0.x, V_POSITION, c[a0.x + CV_BONESTART + 0]
dp4 r0.y, V_POSITION, c[a0.x + CV_BONESTART + 1]
dp4 r0.z, V_POSITION, c[a0.x + CV_BONESTART + 2]

//transform normal by first bone, store in r2
dp3 r2.x, V_NORMAL, c[a0.x + CV_BONESTART + 0]
dp3 r2.y, V_NORMAL, c[a0.x + CV_BONESTART + 1]
dp3 r2.z, V_NORMAL, c[a0.x + CV_BONESTART + 2]
```

VS Matrix Palette Skinning

```
//load second index
mov a0.x, V_INDICES.y

//transform position by second bone, store in r1
dp4 r1.x, V_POSITION, c[a0.x + CV_BONESTART + 0]
dp4 r1.y, V_POSITION, c[a0.x + CV_BONESTART + 1]
dp4 r1.z, V_POSITION, c[a0.x + CV_BONESTART + 2]

//transform normal by second bone, store in r3
dp3 r3.x, V_NORMAL, c[a0.x + CV_BONESTART + 0]
dp3 r3.y, V_NORMAL, c[a0.x + CV_BONESTART + 1]
dp3 r3.z, V_NORMAL, c[a0.x + CV_BONESTART + 2]
```

VS Matrix Palette Skinning

```
//blend between r0 and r1 -- the positions
mul r0, r0, V_WEIGHT.x
mad r1, r1, V_WEIGHT.y, r0

//blend between r2 and r3 – the normals
mul r2, r2, V_WEIGHT.x
mad r3, r3, V_WEIGHT.y, r7
```

- **Now, r1 contains the blended vertex position and r3 contains the blended normal. Just use them as usual…**

VS Matrix Palette Skinning

- Some notes:
  - Note that GeForce3 does not support the vertex attribute format D3DVSDT_UBYTE4, which is the most natural format for storing indices.
  - The previous sample used shorts to store the indices, which can take twice the memory.
  - Instead, it’s possible to use D3DVSDT_D3DCOLOR, which is a packed DWORD usually used to represent colors.
  - Note that these values will then have to be scaled by 255 in the vertex shader since they are converted to the range [0..1].
  - Memory / Speed tradeoff.

VS Matrix Palette Skinning -- Issues

- Limited constant memory – 96 4-vectors
- If your bone matrices are affine, it's not necessary to store a full 4x4 matrix per bone, a 4x3 will do.
- If you can avoid anisotropic scales in your bone matrices, you can avoid having to store inverse-transpose matrices for transforming vectors.
- So, 96 / 3 == 32 bones max, although realistically you'll probably only be able to store around 20-25.

VS Matrix Palette Skinning -- Issues

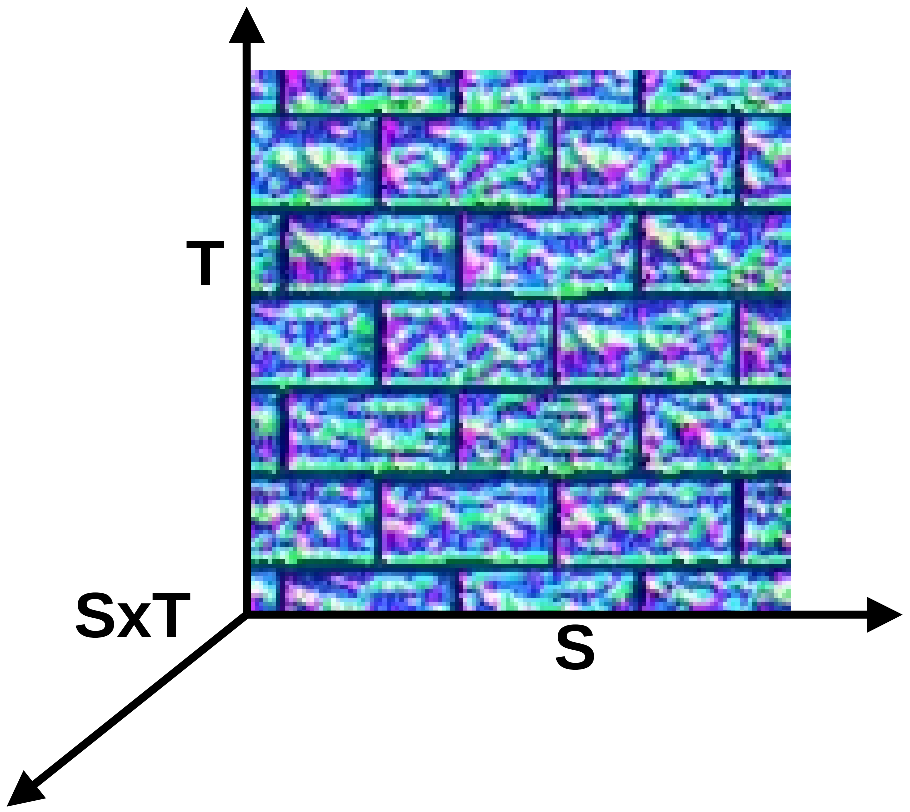
- Can use quaternions to increase number of possible bones in constant memory.
- One four vector to represent the represent the quaternion rotation.
- One three vector to represent translation.
- Total bones possible increases to 96 / 2 == 48!
- Transformation more expensive, though.

VS Matrix Palette Skinning -- Issues

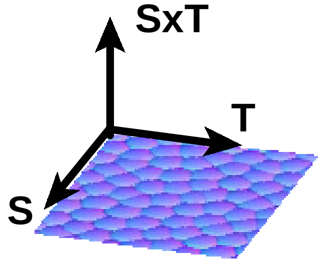
- **Which means you'll have to split up your mesh into sections influenced by a certain number of bones.**
- **Since bones are arranged spatially, vertices should fall into bins influenced by certain bones.**
- **Note that the D3DXMesh function ConvertToIndexedBlendedMesh() can do this for you.**

Texture Space Review

- **Per-pixel Lighting uses the idea of a local per-vertex texture space:**

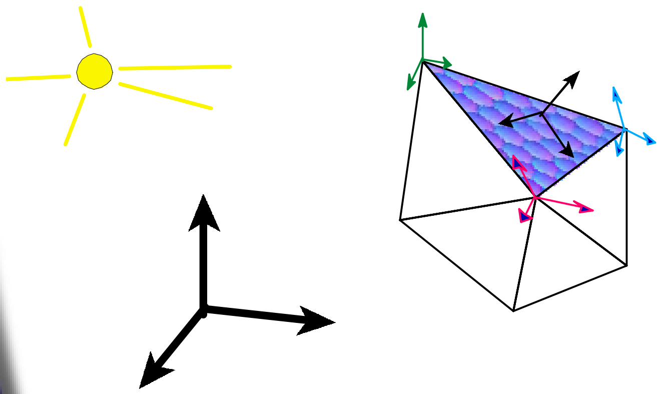


Texture Space Review



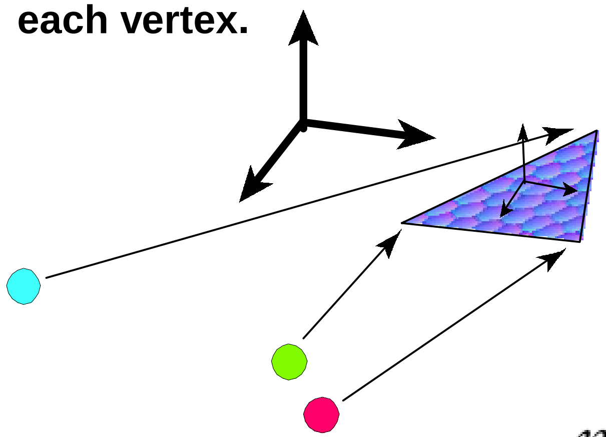
Normal Map – A flat plane in S,T direction

L and N are expressed in different coordinate systems



Solution = Rotate Light position into S,T,SxT space.

Result: New light position for each vertex.



Integration with Per-Pixel Lighting

- Per-pixel lighting usually stores a per-vertex texture space matrix to transform model space vectors into local texture space.
- However, when models distort (i.e. animate), these texture space vectors become invalid.
- The naïve solution to this is to regenerate the texture space matrices from the model data every time the model moves.
- This is quite CPU intensive and slow, however, especially for complex models.

A Better Way – Update the Bases

- The solution is to instead modify the existing bases, preferably in the vertex shader.

- This works for both matrix palette skinning and keyframe interpolation, provided you’re using vertex shaders.

VS Matrix Palette Skinning

- For each axis of Texture Space-- usually called the S, T, and SxT vectors– skin the vectors in the exact same fashion as you did the vertex normals.
- Note that you can skip skinning the SxT vector and instead derive it in the vertex shader using the cross product, which is only two instructions. Can be much cheaper if you have many bones per vertex.
- You can also often just use the vertex normal as the SxT vector.

Keyframe Interpolation

- Create keyframes for the S, T, and SxT vectors as well (or, once again, derive the SxT vector in the shader)
- Linearly interpolate between the S(0) and S(1) using the keyframe weight from 0 to 1

  (1 – Weight) S0 + (Weight) * S1
- Now Normalize the result
  - To handle scaled or stretched textures
  - Rescale by the linearly interpolated length of the two keyframe vectors

    NormalizedVector *=

    (1 – Weight) * LengthOf(S0) +

    (Weight)   * LengthOf(S1)

Keyframe Interpolation

- The normalizing of the vector approximates a SLERP
- The rescaling ensures that any stretching or scaling in the textures is preserved
  - especially important if morphing
- Note that these techniques may not be necessary if you’re using a more sophisticated interpolation technique like hermite interpolation.

Optimization Issues

- With skinning, the number of instructions in your vertex shaders grows quickly.
- If you’re doing simple vertex lighting, the skinning cost is at least 9 instructions per bone.
- For per-pixel lighting with skinning the bases, the cost grows to at least 16 instructions per bone.
- So it’s 64 instructions to do *just the skinning* with 4 bones and per-pixel lighting, not to mention putting the light into texture space, renormalizations, projection, etc.

Optimization Issues

- If the number of bones influencing individual vertices varies a great deal from vertex to vertex, bin your vertices into groups influenced by a certain number of bones.
- So, if you have a character with vertices in the face influenced by 4 bones each, while vertices on the torso are influenced by 2 bones each, don’t render them with the same call to DrawPrimitive(), even if the total number of verts fits in the constant memory.
- Instead, render each section with two different vertex shaders.

More Optimization Issues

- Note that running the ‘optimal’ vertex shader isn’t always a win.
- If you’re entirely fill or memory bound, it may be easiest to just render everything with the same vertex shader, since the length of your shader isn’t directly influencing your performance in these cases.
- Test and see!

What About the CPU?

- While the hardware can run through a vertex shader much faster than even the fastest CPU can, there may be situations where skinning on the CPU is still preferable.
- If you're doing multiple passes over your characters, the vertex shader has to be run for each pass, there's no persistence.
- On the CPU, one can simply skin the vertices once and send the pre-skinned vertices to the hardware for each pass.
- The CPU can also start transforming the next frame's vertices while the GPU is rasterizing, improving parallelism.

What About the CPU?

- Also, if you’re completely bound by fill or memory, the GPU gets into a situation where the TnL unit is sitting idle while waiting on the rasterizer.
- Using the CPU can potentially improve performance in this case as the CPU can run through the vertex buffer without any stalls at all.
- Note that certain operations are much faster on the GPU:
  - Vector normalization
  - Exponents for specular lighting
  - Cross-products

More Uses For CPU

- Any time an exact skinned position is necessary.
  - Stencil shadows / silhouette extraction
  - Exact collision (for maybe a fighting game)

- Extremely long vertex shaders also may benefit CPU.

- Conclusion: the GPU is extremely good at what it does, but isn’t the solution to every single problem. The goal is to make your games run as fast as possible looking as good as possible.

One Idea For Balancing CPU/GPU

- Use the CPU to do the skinning and generate a keyframe every 5 or so frames.
- Then use the GPU to blend between these keyframes for the intermediate frames, using linear or more complex blending.
- Could strike a great balance between CPU and GPU, especially for multi-pass.

Questions…

?

Cem Cebenoyan

cem@nvidia.com

www.nvidia.com/Developer

nVIDIA