Spherical Skinning with Dual-Quaternions and QTangents

Ivo Zoltan Frey
Crytek R&D
Goals

#1 Improve performance by reducing the shader constant requirements for joint transformations
   30% shader constants reduction

#2 Reduce the memory footprint of skinned geometry
   22% vertex memory reduction
   29% for static geometry
Skinned Geometry
Goal #1

- Improve performance by reducing the shader constant requirements for joint transformations
  - Skinned geometry requires multiple passes
    - Motion Blur requires twice the transformations
  - The amount of required shader constants affects the performance of a single pass
Skinning with Quaternions

- ~30% less shader constants consumption compared to 4x3 packed matrices

- Quaternion Linear Skinning
  - Accumulated transformations don’t work for positions
    - Explosion of vertex instructions

- Quaternion Spherical Skinning [HEIJL04]
  - Extra vertex attribute required
  - Doesn’t handle well more than 2 influences per vertex

- Dual-Quaternion Skinning [KCO06] [KCZO08]
  - Increase in vertex instructions
Dual-Quaternion Skinning \[\text{[KSO06]}\ \text{[KCZO08]}\]

- Compared to Linear Skinning with matrices
  - Accumulation of transformations is faster
  - Applying the transformation is slower
  - With enough influences per vertex it becomes overall faster

- The reduction of shader constants was a win over the extra vertex instructions cost
From Linear to Spherical

- Geometry needs to be rigged differently
  - And you will still need your helper joints

- Riggers and Animators need to get used to it
  - Some will love it, others will hate it
    Most will keep changing their mind

- You might have to write skinning plug-ins for third party authoring software
  - Some recent authoring packages have adopted Dual-Quaternion Skinning out of the box
Goal #2

- Reduce the memory footprint of skinned geometry
  - We are now developing on consoles, every byte counts!
  - More compact vertex format will also lead to better performance
- Do not sacrifice quality in the process!
Tangent Frames

Tangent Frames were the biggest vertex attribute after our trivial memory optimizations.

In further optimizing them we need to ensure that:
- They keep begin efficiently transformed by Dual-Quaternions
- All our Normal Maps keep working as they are
About Tangent Frames

- Please make them **orthogonal**!

- If they are not, you are introducing **skewing**
  - You can’t use a transpose to invert the frame matrix
    - You need a **full matrix inversion**
  - This will also prevent you from using some compression techniques!
Compressed Matrix Format

Vertex attributes contain two of the frame’s vectors and a reflection scalar.

<table>
<thead>
<tr>
<th>Tangent</th>
<th>BiTangent</th>
<th>Reflection</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>y</td>
<td>z</td>
</tr>
<tr>
<td>x</td>
<td>y</td>
<td>z</td>
</tr>
<tr>
<td>s</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The third frame’s vector is rebuilt from a cross product of the given vectors and a multiplication with the reflection scalar.

\[ \text{normal} = \text{cross}(\text{tangent}, \text{biTangent}) \times s \]
Tangent Frames With Quaternions

Quaternion to Matrix conversion

\[
t = \text{transform}(q, \text{vec3}(1, 0, 0))
\]
\[
b = \text{transform}(q, \text{vec3}(0, 1, 0))
\]
\[
n = \text{transform}(q, \text{vec3}(0, 0, 1))
\]

Quaternions don’t natively contain reflection information
Bringing Reflection Into the Equation

Similarly to the compressed matrix format, we can introduce reflection with a scalar value

\[ t = \text{transform}(q, \text{vec3}(1, 0, 0)) \]
\[ b = \text{transform}(q, \text{vec3}(0, 1, 0)) \]
\[ n = \text{transform}(q, \text{vec3}(0, 0, 1)) \times s \]
# Tangent Frame Format Memory Comparison

## Compressed Matrix

<table>
<thead>
<tr>
<th>Tangent</th>
<th>BiTangent</th>
<th>Reflection</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>y</td>
<td>z</td>
</tr>
<tr>
<td>x</td>
<td>y</td>
<td>z</td>
</tr>
<tr>
<td>s</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

## Quaternion

<table>
<thead>
<tr>
<th>Quaternion</th>
<th>Reflection</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>y</td>
</tr>
<tr>
<td>z</td>
<td>w</td>
</tr>
<tr>
<td>s</td>
<td></td>
</tr>
</tbody>
</table>
Our Quaternion Properties

They are normalized

\[ \text{length}(q) = 1 \]

And they are sign invariant

\[ q = -q \]
We can compress a Quaternion down to three elements by making sure one of the them is greater than or equal to zero.

```cpp
if (q.w < 0)
    q = -q
```

We can then rebuild the missing element with

```cpp
q.w = sqrt(1 - dot(q.xyz, q.xyz))
```
Tangent Frame Format Memory Comparison

**Compressed Matrix**

<table>
<thead>
<tr>
<th>Tangent</th>
<th>BiTangent</th>
<th>Reflection</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>y</td>
<td>z</td>
</tr>
<tr>
<td>x</td>
<td>y</td>
<td>z</td>
</tr>
<tr>
<td></td>
<td></td>
<td>s</td>
</tr>
</tbody>
</table>

**Quaternion**

<table>
<thead>
<tr>
<th>Quaternion</th>
<th>Reflection</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>y</td>
</tr>
<tr>
<td>w</td>
<td>s</td>
</tr>
</tbody>
</table>

**Compressed Quaternion**

<table>
<thead>
<tr>
<th>Quaternion</th>
<th>Reflection</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>y</td>
</tr>
<tr>
<td>s</td>
<td></td>
</tr>
</tbody>
</table>
**Instruction Cost**

*Quaternion decompression*
5 mov, dp3, add, rsq, rcp

*Quaternion to Tangent and BiTangent*
6 add, mul, mad, mad, mad, mad

*Normal and Reflection computation*
3 mul, mad, mul

**Total**
11 for Tangent, BiTangent and Reflection
14 for full Tangent Frame
Avoiding Quaternion Compression

Isn't there a way to encode the reflection scalar in the Quaternion, instead of compressing it?

Remember, Quaternions are sign invariant

\[ q == -q \]

We can arbitrarily decide whether one of its elements has to be negative or positive!
Encoding Reflection

First we initialize the Quaternion by making sure $q.w$ is always positive

\[
\text{if } (q.w < 0) \\
q = -q
\]

If then we require reflection, we make $q.w$ negative by negating the entire Quaternion

\[
\text{if } (\text{reflection} < 0) \\
q = -q
\]
Decoding Reflection

All we have to do in order to decode our reflection scalar is to check for the sign of $q.w$

$$\text{reflection} = \begin{cases} -1 & \text{if } q.w < 0 \\ +1 & \text{otherwise} \end{cases}$$

As for the Quaternion itself, we can use it as it is!

$q = q$
Instruction Cost

Reflection decoding
2  slt, mad

Quaternion to Tangent and BiTangent
6  add, mul, mad, mad, mad, mad

Normal and Reflection computation
3  mul, mad, mul

Total
8 for Tangent, BiTangent and Reflection
11 for full Tangent Frame
Tangent Frame Transformation with Dual-Quaternion

**Quaternion-Vector transformation**

```c
float3x3 frame;
frame[0] = transform_quat_vec(
    skinningQuat, vertex.tangent.xyz);
frame[1] = transform_quat_vec(
    skinningQuat, vertex.biTangent.xyz);
frame[2] = cross(frame[0], frame[1]);
frame[2] *= vertex.tangent.w;
```

15 instructions

**Quaternion-Quaternion transformation**

```c
float4 q = transform_quat_quat(
    skinningQuat, vertex.qTangent);
float3x3 frame = quat_to_mat(q);
frame[2] *= vertex.qTangent.w < 0 ? -1 : +1;
```

16 instructions
QTangent Definition

A Quaternion of which the sign of the scalar element encodes the Reflection
Stress-Testing QTangents

By making sure we throw at it our most complex geometry!
Singularity Found!

Weapons Artist

FF FFFFFF
FF FFFFFF
FF FFFFFF
UUUUUUUU
UUUUUUU
Singularity Found!

At times the most complex cases pass, while the simplest fail!
Singularity

Our singularities manifest themselves when the Quaternion’s scalar element is equal to zero

\[
\begin{bmatrix}
-1, & 0, & 0 \\
0, & -1, & 0 \\
0, & 0, & 1 \\
\end{bmatrix}
\]

\[
0, \ 0, \ 1, \ 0
\]

This means the Tangent Frame’s surface is perpendicular to one of the identity’s axis
Floating-Point Standards

- So what happens when the Quaternion’s scalar element is 0?

- The IEEE Standard for Floating-Point Arithmetic does differentiate between -0 and +0, so we should be fine!

- However GPUs don’t exactly always comply to this standard, at times for good reasons
GPUs Floating-Point “Standards”

- GPUs allow vertex attributes to be specified as integers representing normalized unit scalars.
- They are then resolved into Floating-Point values.
- Integers don’t differentiate between -0 and +0, thus this information is lost in the process.
Handling Singularities

- In order to use integers to encode reflection, we need to ensure that $q \cdot w$ is never zero.

- When we find $q \cdot w$ to be zero, we need to apply a bias.
Defining Our Bias Constant

We define our bias constant as the smallest value that will satisfy $q \cdot w \neq 0$

If we are using an integer format, this value is given by

$$bias = \frac{1}{2^{\text{BITS}-1} - 1}$$
Applying the Bias Constant

We need to apply our bias for each Quaternion satisfying $q.w < \text{bias}$, and while doing so we make sure our Quaternion stays normalized.

```c
if (q.w < bias)
{
    q.xyz *= sqrt(1 - bias*bias)
    q.w = bias
}
```
QTangents with Skinned Geometry

- From **36 bytes** to **28 bytes** per vertex
- ~22% memory saved
- No overhead with Dual-Quatennion Skinning
- ~8 instruction overhead with Linear Skinning

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Position</td>
<td>4 float16</td>
<td>8 bytes</td>
</tr>
<tr>
<td>TexCoord</td>
<td>2 float16</td>
<td>4 bytes</td>
</tr>
<tr>
<td>Tangent</td>
<td>4 int16</td>
<td>8 bytes</td>
</tr>
<tr>
<td>BiTangent</td>
<td>4 int16</td>
<td>8 bytes</td>
</tr>
<tr>
<td>SkinIndices</td>
<td>4 uint8</td>
<td>4 bytes</td>
</tr>
<tr>
<td>SkinWeights</td>
<td>4 uint8</td>
<td>4 bytes</td>
</tr>
</tbody>
</table>
QTangents with Static Geometry

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Position</td>
<td>4 float16</td>
<td>8 bytes</td>
</tr>
<tr>
<td>TexCoord</td>
<td>2 float16</td>
<td>4 bytes</td>
</tr>
<tr>
<td>Tangent</td>
<td>4 int16</td>
<td>8 bytes</td>
</tr>
<tr>
<td>BiTangent</td>
<td>4 int16</td>
<td>8 bytes</td>
</tr>
</tbody>
</table>

- From 28 bytes to 20 bytes per vertex
- ~29% memory saved
- ~8 instruction overhead

71%
Future Developments

- Quaternions across polygons
  - Interpolating Quaternions across polygons and making use of them at the pixel level
- Quaternions in G-Buffers
  - Encoding the whole Tangent Frame instead of just Normals
  - Can open doors to more Deferred techniques
    - Anisotropic Shading
    - Directional blur along Tangents
Special Thanks

- Ivo Herzeg, Michael Kopietz, Sven Van Soom, Tiago Sousa, Ury Zhilinsky
- Chris Kay, Andreas Kessissoglou, Mathias Lindner, Helder Pinto, Peter Söderbaum
- Crytek
References

[HEIJL04] Heijl, J., "Hardware Skinning with Quaternions", Game Programming Gems 4, 2004


Questions?

ivof@crytek.com