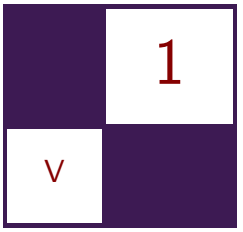


# Quaternions Revisited

Peter Sikachev, Vladimir Egorov,
and Sergey Makeev

## 1.1 Introduction

Quaternions have been extensively used in computer graphics in the last few years. One defines a quaternion as a hypercomplex number, $w + xi + yj + kz$, but in practice it is convenient to consider it to be a 4D vector $(x, y, z, w)$ with a special operation of multiplication defined for it. In 3D computer graphics, quaternions can be used to encode rotations and coordinate systems.

In this chapter we describe the experience of using quaternions in the MMO-RPG engine. In comparison with [Malyshau 12], we propose a quaternion interpolation solution that does not increase vertex count. Besides, we go deeper in detail regarding precision and performance issues. Finally, we strive to cover a broader range of problems, including normal mapping, skinning, instancing, morph targets, and nonuniform scale.

## 1.2 Quaternion Properties Overview

While strict wording may be found in the excellent book [Lengyel 11], we will summarize some key quaternion properties below. The quaternion $\mathbf{q}$ is called *normalized* if $\|\mathbf{q}\| = \sqrt{x^2 + y^2 + z^2 + w^2} = 1$. The geometric meaning of a normalized quaternion $(x, y, z, w)$ can be easily perceived if we re-write it as

$$(x, y, z, w) = \left(x' \sin\left(\frac{\alpha}{2}\right), y' \sin\left(\frac{\alpha}{2}\right), z' \sin\left(\frac{\alpha}{2}\right), \cos\left(\frac{\alpha}{2}\right)\right).$$

This quaternion encodes an $\alpha$ radians rotation around the axis (normalized vector) $(x', y', z')$. This notation immediately provides us with the following properties for quaternions:

- The quaternion $\mathbf{q}$ is equivalent to the quaternion $k\mathbf{q}$ where $k \in \mathbb{R}$.
- The normalized quaternions $(x, y, z, w)$ and $(-x, -y, -z, -w)$ are equivalent.
- The inverse rotation for the quaternion $(x, y, z, w)$ is denoted by the quaternion $(x, y, z, -w)$.

The angle between two normalized quaternions $\mathbf{q_1}$, $\mathbf{q_2}$ can be found as $\theta = 2 \arccos(q_1, q_2)$, where $(q_1, q_2)$ is a per-component dot product of $q_1$ and $q_2$. In this chapter all quaternions are implied to be normalized if not noted otherwise.

## 1.3 Quaternion Use Cases

In our engine we used quaternions for multiple purposes. Our goal was to replace bulky $3 \times 3$ rotation and tangent-space matrices throughout the entire engine. This affected the following engine components:

- normal mapping,
- generic transforms,
- instancing,
- skinning,
- morph targets.

For each of these cases we discuss the implications in the following individual sections.

## 1.4 Normal Mapping

Normal mapping was an initial reason to use quaternions engine-wide. Normal mapping requires one to define a so-called tangent space at each surface location: a coordinate system, which is defined by tangent, bi-tangent (often erroneously called bi-normal), and normal (TBN).

Usually, these basis vectors are defined per model vertex and then interpolated inside triangles, and this is where the problems arise. First, TBN occupies at least six interpolator channels (provided we reconstruct the third vector using a `cross` instruction and pack a handedness bit without wasting an extra interpolator channel). This may be alleviated by packing TBN, but it comes at a pixel shader additional cost.

Second, TBN might need orthonormalization multiple times throughout the pipeline. In the case of a model-view matrix containing a nonuniform scale, we

need to use inverse-transpose of the model-view matrix to transform TBN (tangents and bi-tangents use the same transform as normals). It is very expensive to invert a matrix in a shader, and in the case of a model-view inverse-transpose matrix pre-computation, we would need to pass this matrix through vertex attributes per instance in case of instancing. One possible hack is to re-use the original model-view matrix to transform TBN, but in this case we need to orthonormalize TBN afterwards to avoid lighting artifacts.

Moreover, TBN becomes unnormalized and loses orthogonality during interpolation. Hence, orthogonalization and normalization also need to be performed in a pixel shader. In our experience, this takes a significant amount of ALUs in vertex and pixel shaders.

We tried several methods enabling normal mapping without TBN in our engine. [Mikkelsen 10] needs so-called derivative maps as an input, which can be obtained from normal maps. However, on our assets this method was capable of conveying much less detail than regular normal mapping, which was unacceptable for artistic reasons.

We also tried the method in [Schüler 06]. It implies additional ALU cost for differential chain rule application. However, the greatest limitation appeared to be pipeline issues: the TBN basis generated by Autodesk Maya did not coincide with one derived from the texture-coordinate gradient. This resulted in different parameterizations, making it impossible to reuse already existing normal maps.

Encoding TBN with quaternions allows us to overcome these problems. However, there are several problems with this approach. One is TBN matrix *handedness*, which we will not discuss here as it is covered in detail in [Malyshau 12].

The other problem is quaternion interpolation. Quaternion *spherical linear interpolation* (SLERP) produces correct results without a prior alignment. However, since vertex attribute interpolation is not programmable, we interpolate quaternions linearly, which causes issues. [Malyshau 12] proposes a solution that increases the original vertex count by 7.5% and polygon count by 0.14%. Below, we propose a solution, which works for any correctly UV-mapped model, that does not change vertex and polygon count at all. Let us first explain the ground for this problem.

Let us consider quaternions $\mathbf{q}_1 = (x_1, y_1, z_1, w_1)$ and $\mathbf{q}_2 = (x_2, y_2, z_2, w_2)$. If their dot product $(\mathbf{q}_1, \mathbf{q}_2)$ is positive, the quaternions will interpolate along the shortest arc; otherwise they interpolate along the longest. That being said, if we take a $-\mathbf{q}_1$ quaternion (instead of an equivalent $\mathbf{q}_1$), the interpolation arc will be reversed. For vertex normals inside the triangles, we will expect to interpolate along the shortest arc. Yet, for a closed model, that might not always be the case. A natural solution to this will be to *align* all quaternions at the model's neighboring vertices by multiplying one of them by $-1$ so that their dot product becomes positive. However, this will not solve the problem in most cases.

Let us consider a cylinder model as in Figure 1.1. We will take a quaternion $\mathbf{q} = (0, 0, 0, 1)$ at the point $A$ as a reference and start aligning all the quaternions

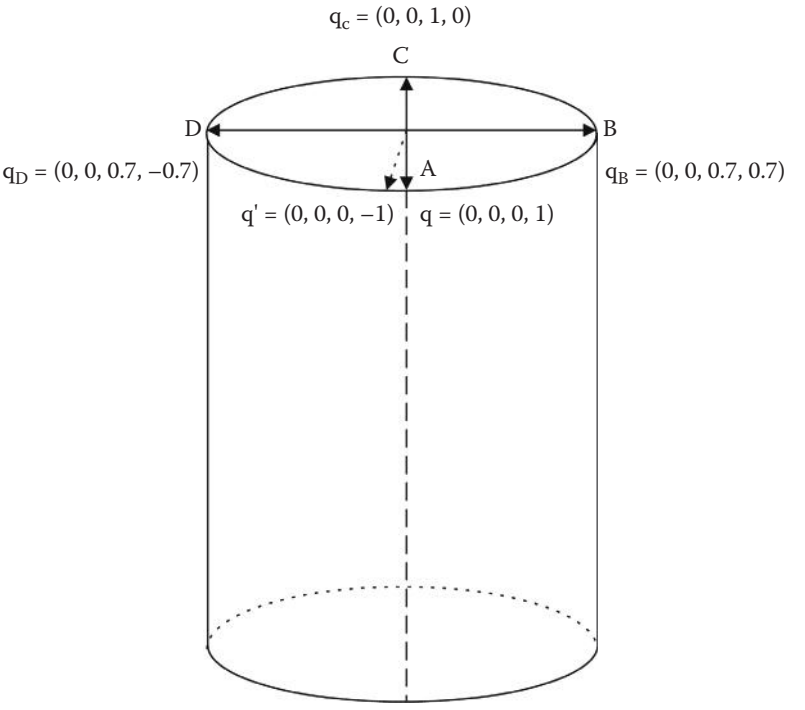


**Figure 1.1.** Aligning quaternions for the cylinder object.

on the cylinder surface one after another in the counterclockwise direction. As one can observe, when we return to point $A$ after a turn around, the respective quaternion at this point will be $\mathbf{q}' = (0, 0, 0, -1)$. This means that neighboring vertices at point $A$ will be *misaligned*, resulting in the long-arc quaternion interpolation as shown in Figure 1.2.

The geometrical justification for this effect lies in the quaternion nature. If we recall that a quaternion $\mathbf{q} = (x\sin(\frac{\alpha}{2}), y\sin(\frac{\alpha}{2}), z\sin(\frac{\alpha}{2}), \cos(\frac{\alpha}{2}))$, it becomes clear that while our turn around the angle $\alpha$ (and respective normal) made a complete 360°, which corresponds to the full period of the function, it only made a half-period (since we have $\frac{\alpha}{2}$ as an argument) for the quaternion.

In the general case, this problem is unsolvable without a topology alteration. However, for a real in-game UV-mapped asset, we will never have a situation when we can find a closed chain of adjacent vertices on the surface of a model so that a normal makes a 360° rotation when we traverse this chain. This is true because a closed mesh cannot be correctly UV-mapped so that a texture, unique at each point, can be applied to this mesh. If we get back to the cylinder sample and try to map a 2D texture to the cylindrical surface (Figure 1.3), this would mean that,

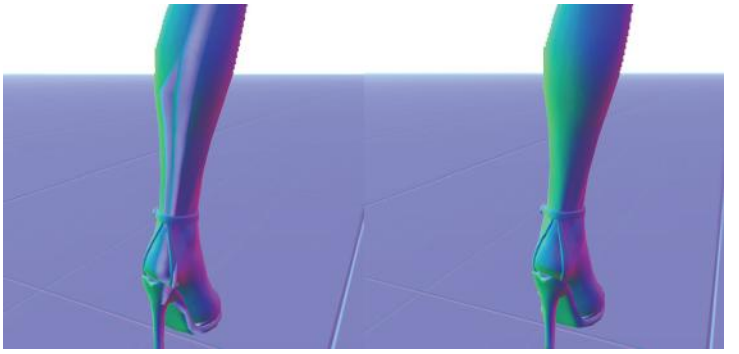

**Figure 1.2.** Normal rendering. Incorrectly interpolated quaternions (left) and correctly interpolated quaternions after alignment (right). [Image courtesy of Mail.Ru Group.]

at some point, a $u$-coordinate will change from 0 to 1 at the neighboring vertices, resulting in the whole texture $u$-range being mapped to the single triangle. To avoid this, artists duplicate the vertices of the model, having as a result vertices $\mathbf{v}$ and $\mathbf{v}'$ with the same position but different texture coordinates. This fact allows us to align quaternions at neighboring vertices without any further vertex duplication, which is described in Algorithm 1.1.

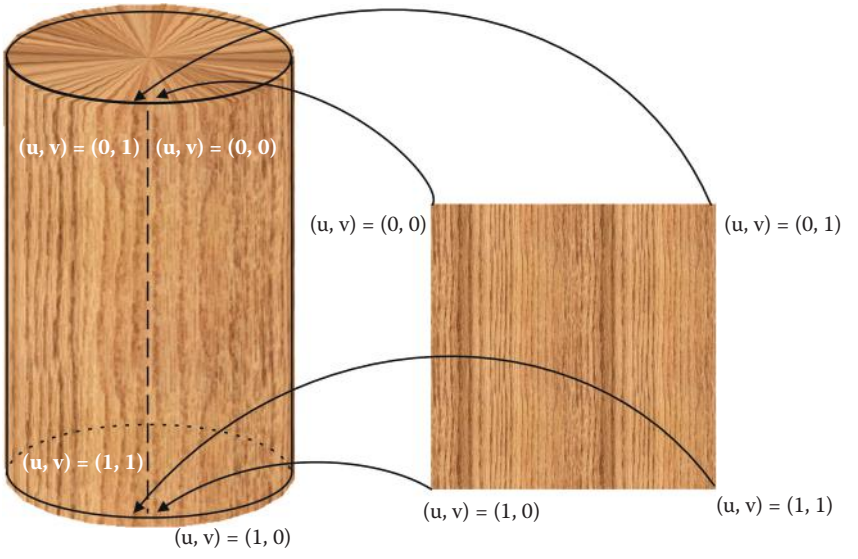


**Figure 1.3.** Textured cylinder object.

**Require:** Initialize all vertex flags as *non-traversed*
1: **while** *non-traversed* vertex exists **do**
2: select any *non-traversed* vertex as **q**
3: set **q** flag as *traversed*
4: **for** every *non-traversed* vertex $\mathbf{q}'$ sharing edge with **q** **do**
5: set $\mathbf{q}'$ flag as *traversed*
6: align $\mathbf{q}'$ to **q**
7: repeat recursively from step 4 for $\mathbf{q}:=\mathbf{q}'$
8: **end for**
9: **end while**

**Algorithm 1.1.** Quaternion alignment algorithm for UV-mapped meshes.

## 1.5 Generic Transforms and Instancing

If we compare encoding transformations with quaternions and with matrices, quaternions will have several advantages. First, a transformation matrix needs $4 \times 3$ values to encode position, rotation and scale, while with *SQTs* (scale-quaternion-translation) we need only eight values provided we assume the scale to be uniform. Second, reverse rotation comes virtually for free in the case of quaternions, while for matrices this is a quite costly operation. Finally, multiplying two $3 \times 3$ rotation matrices will result in 45 scalar operations (additions and multiplications), while quaternions can be multiplied using a total of 28 scalar operations.

Therefore, we used quaternions in our engine as widely as possible. However, we have experienced several caveats with them.

The first one comes from the fact that a normalized quaternion **q** being multiplied by itself several times (i.e., $\mathbf{q}^n$) becomes significantly unnormalized. For example, an incremental rotation of an object at a same angle may lead to this. A straightforward solution to this is normalizing a quaternion multiplication product, but this significantly reduces efficiency, as normalization is usually a relatively slow operation. An alternative solution will be keeping a normalization assertion check for a debug version of an application and eliminating cases of all possible multiplications of a quaternion on itself.

Another limitation for quaternions is nonuniform scale. In *Skyforge* we needed a nonuniform scale for a family of effects classified as "channeling." Figure 1.4 shows an example of a channeling effect. The particular feature of this family of effects is that an effect is stretched between two arbitrary points in space. Hence, a nonuniform scale is needed to be applied to them.

While a nonuniform scale may co-exist with quaternions, we have faced several limitations, which should be taken into consideration at early production stages. They come from the fact that we effectively decouple scale from rotation (in contrast to conventional $3 \times 3$ scale-rotation matrices).

**Figure 1.4.** Channeling spell being cast. [Image courtesy of Mail.Ru Group.]

First, only *axis-aligned nonuniform* scale may be encoded. Figure 1.5 shows the difference between encoding the same transform with an SQT and a scale-rotation matrix. While a single scale-rotation transform performed by a scale-rotation matrix can also encode only an axis-aligned nonuniform scale, a superposition of these transforms can encode a non-axis-aligned scale, while a superposition of SQTs will always keep all scales axis-aligned. This means, for instance, that a box will always remain a box after any SQT transformations (keeping straight angles around its corners), but this is not a case for an arbitrary scale-rotation matrices superposition. In our experience, we did not have a strong need for such transformations. However, this can slightly increase the mesh count, as, for instance, different rhombi can no longer be encoded with a single mesh with different transformations applied.

Second, with SQTs all scale transformations are done in the *object space*. For instance, if you rotate an object at a 90° angle in the $xy$-plane and then scale along the $x$-axis, it will be scaled along the $y$-axis instead, as shown in Figure 1.6. In terms of channeling effects, this imposed a restriction on the content. All objects that a channeling effect consists of should be aligned along the major axis of the effect; otherwise, they will be stretched orthogonally to the effect direction.

For instanced objects, encoding rotation with quaternions worked straightforwardly. We provide more details on this in Section 1.8.

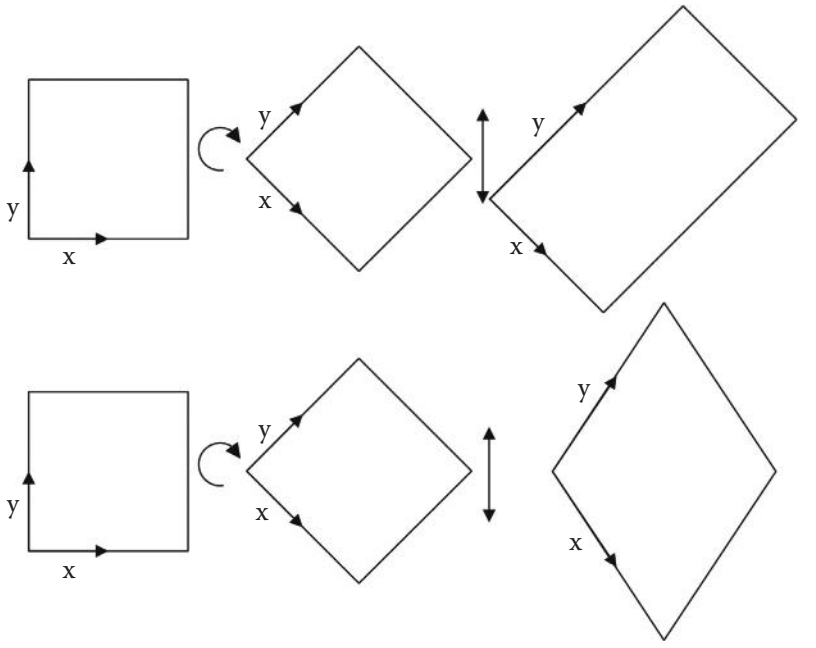


**Figure 1.5.** A box being rotated and scaled using SQT (top) and scale-rotation matrix (bottom). Object $xy$-axis is shown.

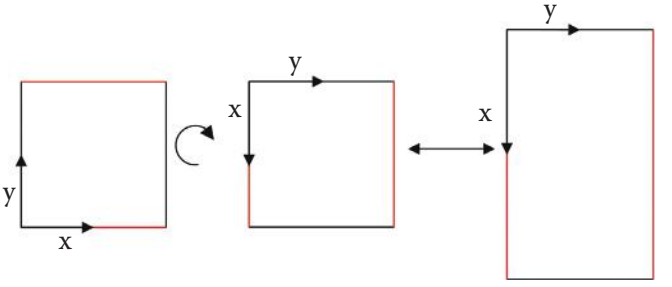


**Figure 1.6.** 90° rotation and scaling with SQT.

## 1.6 Skinning

In our engine we pack bone data for skinning into a texture. There are several reasons for this. First, as the game is supposed to run on a DirectX 9 compatible hardware, only 256 constants are available in the general case. Second, we wanted to enable character instancing: in the case of storing skinning data in a single texture, we need to store only a texture offset per instance. Thus, we

**Figure 1.7.** Matrix palette skinning (left) and direct quaternion blending (right). [Image courtesy of Mail.Ru Group.]

pack all bones of all characters in a single 2D texture, the format of which is discussed below. In fact, we use this texture as a "large constant buffer" which we dynamically address using an instance ID, in order to make skinning work for many characters on DirectX 9 compatible hardware.

A straightforward solution for using quaternions in skinning would be to interpolate quaternions, corresponding to different bones, *before* applying rotation; in the same way as matrices are interpolated in the *matrix palette skinning* approach, as proposed in [Hejl 04]. However, our experience shows that this method produces incorrect results when a vertex is skinned to several bones, which have significant rotation from a bind pose, as shown in Figure 1.7.

The mathematical reason for this lies in the fact that quaternions *cannot be correctly blended*. Provided quaternions can be very effectively *interpolated*, this sounds a bit controversial; therefore we provide a proof below.

Let us consider Figure 1.8. A quaternion $\mathbf{q}_2$ defines a $180 - \epsilon^\circ$ rotation in an xy-plane and $\mathbf{q}_3$ corresponds to a $180 + \epsilon^\circ$ rotation. If we blend quaternions $\mathbf{q}_1$, $\mathbf{q}_2$ and $\mathbf{q}_1$, $\mathbf{q}_3$ with weights $\lambda_1 = \lambda_2 = \lambda_3 = 0.5$, resulting quaternions $\mathbf{q}_4$ and $\mathbf{q}_5$ will point in significantly different directions if we use shortest-arc interpolation for blending. Taking $\epsilon$ infinitely small proves that the blending result of $\mathbf{q}_1$ and $\mathbf{q}_2$ ($\mathbf{q}_3$) is not *continuous*.

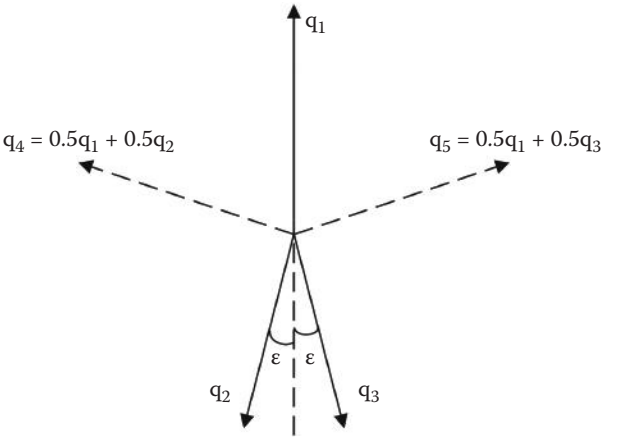


**Figure 1.8.** Discontinuity of blending two quaternions.

Furthermore, blending three or more quaternions is not *commutative*. Let us consider Figure 1.9. Quaternions $\mathbf{q}_1$, $\mathbf{q}_2$, and $\mathbf{q}_3$ specify 0°, 140°, and 220° rotations in the $xy$-plane, respectively. Let us set quaternion weights to $\lambda_1 = \lambda_2 = \lambda_3 = \frac{1}{3}$. The case where we blend quaternions in order $\mathbf{q}_4 = (\lambda_1\mathbf{q}_1 + \lambda_2\mathbf{q}_2) + \lambda_3\mathbf{q}_3$ will differ from the result when we blend them in order $\mathbf{q}_5 = (\lambda_1\mathbf{q}_1 + \lambda_3\mathbf{q}_3) + \lambda_2\mathbf{q}_2$.

Having said that, we do not blend quaternions directly. Instead, we blend final transformed vertex positions. This, obviously, slightly increases ALU count in the shader code, but we benefit from a reduced skinning texture size and fewer vertex texture fetches, as shown in Section 1.8.

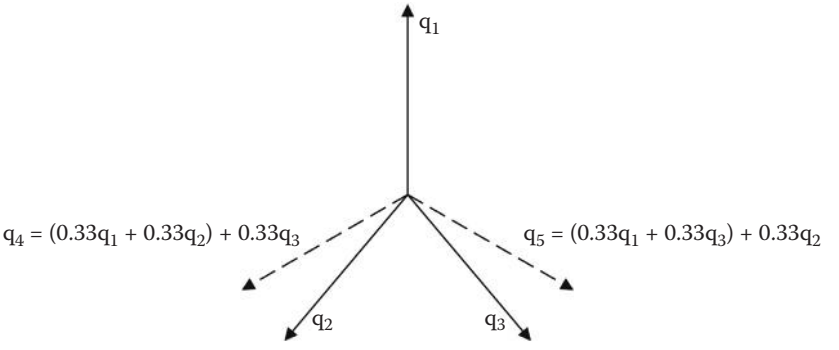


**Figure 1.9.** Ambiguity of blending three quaternions.

**Figure 1.10.** Original character (left) and two extreme morph targets (center, right). [Image courtesy of Mail.Ru Group.]

## 1.7 Morph Targets

As *Skyforge* is an MMORPG game, we want to give players as many opportunites to customize characters as possible. One of the most interesting customizations enables a player to change body proportions of his or her character in a continuous fashion. This is implemented via interpolation of each vertex between two so called *morph targets*, i.e., separate vertex streams. An example of such an interpolation is shown in Figure 1.10.

Regarding quaternions, only a few considerations should be kept in mind for morph targets. First, meshes, corresponding to different targets between which we are interpolating, should have the same topology and *uv*-mapping. This obviously includes handedness of initial TBN in vertices. Second, after we convert TBNs to quaternions, we should align quaternions, corresponding to the same vertex in different targets. This can be obtained naturally, if we start Algorithm 1.1 for both targets at the same vertex, as meshes share the same topology.

## 1.8 Quaternion Format

In this section we discuss how we store quaternions in memory and how we pack them when sending to GPU. Table 1.1 shows layouts we use for quaternions in our engine.

The most interesting case is a vertex quaternion. Our experiments have shown that 8-bit precision is enough for per-vertex quaternions. However, since TBNs can have different handedness, we also need to store a handedness bit. We did not want to spend another channel on it, so we came up with two solutions.

| Case | Channels | | | | |
|---|---|---|---|---|---|
| Vertex Quaternion | $x$ (int8) | $y$ (int8) | $z$ (int8) | $h$ (1b) | $w$ (7b) |
| Model Transform | $x$ (float32) | $y$ (float32) | $z$ (float32) | $w$ (float32) | |
| Instance Data | $x$ (int16) | $y$ (int16) | $z$ (int16) | $w$ (0, calculated) | |
| Bone Transform | $x$ (float16) | $y$ (float16) | $z$ (float16) | $w$ (float16) | |

**Table 1.1.** Quaternion packing for different scenarios. $h$ stands for quaternion handedness.

First, we could store in the same byte a handedness bit and a sign bit for $w$ and then reconstruct $w$ in the shader as we operate onto normalized quaternions. We abandoned this approach, as unpacking $w$ would lead to the `sqrt` instruction, which is costly and should be avoided in the shader.

Instead, we packed a handedness into the first bit and packed $w$ into the remaining seven last bits (as an unsigned integer). It turned out that even seven bits is enough in the case of vertex quaternion. Furthermore, this approach resulted in a very fast unpacking, as shown in Listing 1.1. A nice property of this packing is that it interpolates correctly in all cases of vertex quaternions and morph targets. We implicitly use the fact that it is never a case in practice to interpolate quaternions with different handedness. In this case, the first bit of quaternions to be interpolated would always be the same, resulting in correct interpolation of low seven bits.

For model transforms, as we operate them on the CPU side, we store the quaternion values in `float32` format. To pack a quaternion of instance rotation, we opted for `int16` for several reasons. First, we wanted to keep instance vertex layout as small as possible, and 32-bit formats are definitely an overkill for a quaternion in terms of precision. Quaternion's values are limited to the $[-1, 1]$ domain, so we need only fixed-point precision. Unfortunately, 8-bit formats were not enough as they provide around 1–2° angle steps. While this was enough for a vertex quaternion, in the case of encoding an instance's position, such a

```
float UnpackFirstBitFromByte( float argument )
{
  return saturate((argument * 255.0f - 127.5f) * 100.0f);
}

float UnpackLow7BitsFromByte( float firstBit, float argument )
{
  return (argument * 255.0f - 128.0f * firstBit) / 127.0f;
}
```

**Listing 1.1.** Vertex quaternion unpacking.

```
inline uint16 Float2fp16( float x )
{
 uint32 dwFloat = *((uint32 *)&x);
 uint32 dwMantissa = dwFloat & 0x7fffff;
 int32 iExp = (int)((dwFloat>>23) & 0xff) - (int)0x70;
 uint32 dwSign = dwFloat>>31;


 int result = ( (dwSign<<15)
  | (((uint32)(max(iExp, 0)))<<10)
  | (dwMantissa>>13) );
 result = result & 0xFFFF;
 return (uint16)result;
}
```

**Listing 1.2.** `float32` to `float16` fast conversion.

large angle step resulted in non-smooth rotation and difficulties in aligning static instanced geometry for map designers.

Finally, we experimented with quaternion packing for skinning. Initially, we stored a skinning bone $3 \times 3$ scale-rotation matrix and position (12 values in total) in a `float32` RGBA texture. Therefore, we needed three vertex texture fetches per bone and we wasted 25% of the skinning texture for better alignment, resulting in four `float32` RGBA texels per bone. After switching to quaternions, we used SQT encoding with a uniform scale: this resulted in eight values in total. That allowed us to store a single bone information only in two texels, thus making two vertex texture fetches per bone. As we packed skinning values in a texture, the format of different SQT components had to stay the same. Scale and transform needed a floating-point format; this is why we picked `float16` (a.k.a. `half`). The only issue we tackled in packing was a low speed of a standard DirectX fp16 packing function, which resulted in significant CPU stalls. To address this, we used a fast packing method similar to [Mittring 08]. However, we enhanced this method, making it work for all domain values, unlike the original one. The resulting code is shown in Listing 1.2.

## 1.9 Comparison

After the transition to quaternions had been done, we made a comparison, shown in Table 1.2. As could be observed, using quaternions significantly reduces memory footprint. In the case of normal mapping, the number of `nrm`, `rsq`, and `rcp` instructions is also decreased, which provides better performance increase than one could be expected from the raw ALUs figures. In the case of skinning and instancing, instruction count increases, but in our experience, ALUs have not been a bottleneck in vertex shaders.

| Case | Matrix/TBN | | | | Quaternion | | | |
|---|---|---|---|---|---|---|---|---|
| Measured Value | ALUs | | Memory | | ALUs | | Memory | |
| Pipeline Stage | VS | PS | VS | VS→PS | VS | PS | VS | VS→PS |
| Normal Mapping | 15 | 12 | 7 | 7 | 13 | 12 | 4 | 5 |
| Instancing (Scale+Rot) | 15 | - | 20 | - | 21 | - | 12 | - |
| Skinning | 33 | - | 64 | - | 71 | - | 16 | - |

**Table 1.2.** Comparison of $3 \times 3$ rotation matrices/TBN and quaternions performance (in ALUs) and memory footprint (vertex attributes (TBN, instance data)/vertex texture fetch (skinning) size in bytes (VS)/interpolator channels count (VS→PS)).

## 1.10 Conclusion

In this chapter we tried to fully cover our experience with quaternions. We have used quaternions throughout the whole engine, significantly reducing memory and bandwidth costs. There are, however, certain pitfalls, which we described, that we hope will not prevent quaternions from replacing matrices in modern engines.

## 1.11 Acknowledgements

The authors would like to thank Victor Surkov for helping with the illustrations for this chapter.

## Bibliography

[Hejl 04] Jim Hejl. "Hardware Skinning with Quaternions." In *Game Programming Gems 4*, edited by Andrew Kirmse, pp. 487–495. Boston: Cengage Learning, 2004.

[Lengyel 11] Eric Lengyel. *Mathematics for 3D Game Programming and Computer Graphics*, Third edition. Boston: Cengage Learning PTR, 2011.

[Malyshau 12] Dzmitry Malyshau. "A Quaternion-Based Rendering Pipeline." In *GPU Pro 3*, edited by Wolfgang Engel, pp. 265–273. Boca Raton, FL: A K Peters/CRC Press, 2012.

[Mikkelsen 10] Morten S. Mikkelsen. "Bump Mapping Unparametrized Surfaces on the GPU." *Journal of Graphics, GPU, and Game Tools* 1 (2010), 49–61.

[Mittring 08] Martin Mittring. "Advanced Virtual Texture Topics." In *ACM SIGGRAPH 2008 Games*, pp. 23–51. New York: ACM, 2008.

[Schüler 06] Christian Schüler. "Normal Mapping without Precomputed Tangents." In *ShaderX5: Advanced Rendering Techniques*, edited by Wolfgang Engel, pp. 131–140. Boston: Cengage Learning, 2006.