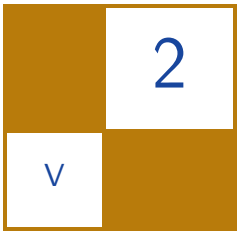

A Quaternion-Based Rendering Pipeline

Dzmitry Malyshau

2.1 Introduction

A matrix is the first thing every graphics developer faces when building an engine. It is a standard data object for representing 3D affine and projection transformations—covering most of the game developer's needs. Matrix operations have even been implemented in hardware used by a 3D graphics API like OpenGL. However, this is not the only way to represent transformations.

Quaternions were introduced by Sir William Rowan Hamilton in the middle of the nineteenth century, at a time when vector analysis did not exist. A quaternion is a hypercomplex 4D number of the form: $w + xi + yj + zk$. There are rules for quaternion multiplication, inversion, and normalization. Quaternions can effectively represent spatial rotations, by applying them to 3D vectors or when converted into 3×3 rotation matrices.

This chapter aims to explain pitfalls and advantages of the quaternion approach for graphics pipelines. There are many articles describing quaternion mathematics, such as [Void 03] and [Gruber 00], which we will not cover here, assuming that the reader is familiar with the basics. We will describe the KRI Engine (see [Malyshau 10a]) as a sample implementation of a complete quaternion-based rendering pipeline.

2.2 Spatial Data

Spatial transformation data combine *rotation*, *position*, and *scale*. Let's compare the most popular representation (i.e., a homogeneous matrix) with a new representation based on a quaternion (see Table 2.1).

We can conclude that quaternion representation is much more complex with regard to transformations in shaders, the manual perspective transform, and other

Type	Matrix	Quaternion (fixed handedness)
`struct Spatial` (*1)	`mat3 rotation_scale; vec3 position;`	`quaternion rotation; vec3 position; float scale;`
vectors to store (four-component)	three	two
understanding difficulty	easy	medium
interpolation flexibility (*2)	low	high
combining transforms cost	high	medium
applying transforms cost	low	medium
hardware support (*3)	high	medium
non-uniform scale support	yes	no
perspective transform	can be added easily, resulting in `mat4`	manual only

*1. Hereafter we use GLSL types (`vec3`,`vec4`,`mat3`,`mat4`,etc.).
*2. For the quaternion approach, you can use spherical linear interpolation (SLERP) or use dual-quaternion representation with little to no difficulty.
*3. Currently, there is no shading language with direct support of quaternions operations. The only exception is the normalization operator, which uses a fast built-in function for `vec4`. However, other operations are easily written via traditional built-in cross and dot vector products (see `quat_v.glsl`).

Table 2.1. Comparison of homogeneous matrix versus quaternion transformation.

issues, but that it provides definite benefits like interpolation flexibility and more efficient storage, which are critical for dynamic graphics scenes.

2.3 Handedness Bit

One of the major differences between rotation matrices and quaternions is the handedness property. Matrices operate freely between right-handed and left-handed coordinate systems. Quaternions do not have this property (always preserving the given handedness upon transformation), so we need to define the corresponding matrix handedness globally and store the actual value together with the quaternion itself. For an orthonormal matrix M, handedness is equal to the determinant and computed as follows:

$$\mathrm{H} = \mathrm{Handedness}(\mathrm{M}) = \det(\mathrm{M}) = ((\mathrm{Row}(0,\mathrm{M}) \times \mathrm{Row}(1,\mathrm{M})) \cdot \mathrm{Row}(2,\mathrm{M})). \tag{2.1}$$

The handedness of an orthonormal matrix can be either $+1$, in which case it is right-handed, or -1, in which case it is left-handed. Assuming we defined quaternions to correspond to right-handed matrices with regard to the vector rotation, we can implement the matrix conversion routines as well as the direct point transformation by a quaternion. Rotation matrix handedness is usually fixed in 3D editors (this applies to Blender and 3DsMax), so our initial assumption is not an issue. However, when a matrix is constructed from arbitrary basis vectors (e.g., tangent space), its handedness can also be arbitrary—and that is a problem.

The issue can be resolved by adding a bit of information to the quaternion: the handedness bit. The algorithm for the matrix processing in the quaternion approach is given in Algorithm 2.1.

Algorithm 2.1. (Calculating and applying the handedness bit.)

1. Given basis vectors, construct matrix M
2. Calculate handedness H, using Equation (2.1)
3. We have to make sure that M is right-handed before converting it to a quaternion. In order to achieve this we are going to multiply the first row of the matrix by the scalar H: Row(0,M) = Row(0,M) * H. This would not change the right-handed matrix (H == 1), but it would flip the handedness of a left-handed one (H == −1)
4. Transform to quaternion, Q = Quaternion(M)
5. Store (Q,H), instead of a matrix M
6. V′T(V) = Rotate(Q,V) * (H,1,1), negating x-coordinate of the transformation vertex V′

We give here some example Blender API Python code that converts a tangent space into the quaternion:

```
bitangent = normal.cross(tangent.normalized())        # derive the bitangent from the mean
                                                      # tangent and a normal
bitangent *= vertex.face.handedness                   # compensate for the opposite handedness
tangent = bitangent.cross(normal)                     # orthogonalize the tangent, defer
                                                      # handedness multiplication to the
                                                      # shader
tbn = mathutils.Matrix((tangent,bitangent,normal))    # composing ortho-normal right-handed
                                                      # tangent space matrix
vertex.quaternion = tbn.to_quaternion().normalized() # obtain the quaternion representing
                                                      # the vertex tangent space
```

2.4 Facts about Quaternions

Here we address some facts and myths about quaternions in an attempt to correct the common misunderstandings of their pros and cons and to provide needed information for using quaternions.

2.4.1 Gimbal Lock

According to Wikipedia, gimbal lock is the loss of one degree of freedom in a three-dimensional space that occurs when the axes of two of the three gimbals are driven into a parallel configuration, “locking” the system into rotation in a degenerate two-dimensional space. Gimbal lock is an attribute of the Euler angle representation of the rotation. Contrary to what some people believe, gimbal lock has nothing to do with either matrices or quaternions.

2.4.2 Unique State

There are always two quaternions (−Q, component-wise negative) that produce exactly the same rotation, thus representing the same state.

2.4.3 No Slerp in Hardware

Graphics hardware can linearly interpolate values passed from the vertex geometry into the fragment shader. One may conclude that quaternions should not be interpolated this way, because spherical linear interpolation is the only correct way to do that.

In fact, after normalizing the interpolated quaternion in the fragment shader we get normalized Lerp (or Nlerp), which is very close to Slerp. According to [Blow 04] it follows the same minimal curve; it just does not keep the constant velocity. So for 3D meshes, it is perfectly fine to use hardware for interpolation of quaternions.

It's important to note that Nlerp produces a larger error, the larger the angle is between quaternions as four-component vectors. While this may seem to be a problem, in fact the error is just four degrees for a 90-degree angle, according to calculations by [Kavan et. al 06]. In practice, we need this HW interpolation for vertex data, and your rendered mesh is not going to look smooth anyway if its normals differ by more than 90 degrees on a single face.

2.4.4 Hypercomplex Four-Dimensional Number

You don't need to know the complete mathematical background of quaternions in order to use them. Knowledge of the following operations is sufficient for graphics developers: inversion, multiplication, interpolation, and applying rotation to a vector.

The GLSL code for a sample rotation application is given below:

```
vec3 qrot(vec4 q, vec3 v)   {
    return v + 2.0*cross(q.xyz, cross(q.xyz,v) + q.w*v);
}
```

2.5 Tangent Space

Tangent space is an orthonormal coordinate system of a surface point, constructed from the tangent, bi-tangent, and normal. It is used for normal mapping as the basis for normals stored in the texture. Object-space normal maps suffer from reduced reusability across different meshes (and mesh parts—for texture tiling) in comparison to tangent-space normal maps. The latter also support skinning and morphing of the object that is normal-mapped; the only requirement is not to miss the tangent space while processing the vertex position. Quaternions are very effective for representing the tangent space. We will show this by comparing the traditional pipeline (via a tangent-space matrix, also known as TBN—an acronym for tangent, bi-tangent, normal) with a new pipeline based on quaternions.

2.5.1 TBN Pipeline

A 3D modeling program may provide both normal and tangent vectors, or just a normal vector for exporting. In the latter case, your engine has to generate tangents based on UV coordinates (exactly those used for normal mapping): in this case, the tangent is calculated as a surface parametrization following the orientation of the U-axis of the UV. Each vertex stores both normal and tangent resulting in two vectors or six floats.

In the case of direct Phong lighting calculations, we just need to transform a few vectors into the tangent space (namely the light vector and the view/half vector). Then, the GPU pipeline interpolates them between vertices. We evaluate the lighting using these two vectors and a surface normal fetched from a map. The tangent space matrix is constructed by simply crossing the normal with the tangent (their length and orthogonality can be ensured by the engine):

```
mat3 tbn = mat3(Tangent, cross(Normal,Tangent) ,Normal);
```

But there are scenarios where you need to interpolate the basis itself, for example to store it in textures for particles or hair to be generated from it afterwards. First, if you need to store the basis, it will occupy two textures, seriously affecting the storage requirements. Second, interpolated basis vectors are neither orthogonal (in general) nor do they have unit lengths. The construction of the TBN matrix is more costly in this case:

```
in vec3 Normal,Tangent;
vec3 n = normalize(Normal);
vec3 t = normalize(Tangent);
vec3 b = cross(n, t);
t = cross(b,n); //enforce orthogonality
mat3 tbn = mat3(t,b,n);
```

2.5.2 Quaternion Pipeline

The TBN matrix can be replaced by the pair: (quaternion, handedness). Each vertex stores a single quaternion and a handedness bit: This requires 1+ vectors or four floats and one bit. For example, the KRI engine stores handedness in the `Position.w` component, supplying an additional quaternion as `vec4` for TBN.

Note that the normal is no longer needed—only the quaternion is provided by the exporter and expected to be present for the renderer. This change introduces a difficulty for the handling of objects with no UV coordinates, because we can't compute either tangents or quaternions for them. Hence, it will be impossible to compute even regular Phong lighting in contrast with the traditional approach where the normal vector is always available. This limitation can be alleviated by making all objects UV-mapped before exporting. Another solution is to have a hybrid pipeline that switches between quaternion-rich and normal-only inputs, but the implementation of such a dual pipeline is complex.

For direct Phong lighting calculations, we rotate vectors by our quaternion and apply the handedness bit as described in Algorithm 2.1. Computation-wise, it is not much more costly than the matrix pipeline, especially taking into account the general low cost of vertex shader computations (assuming that the number of fragments processed is significantly greater than the number of vertices, which should be the case for a properly designed engine).

In a more complex scenario, as in deferred lighting calculation, or baking the world-space surface basis into UV textures, the quaternion approach shows its full power. Storing the surface orientation requires only one texture (if you supply handedness in the position). Interpolation of the quaternion requires a special one-time condition to be met (an algorithm to accomplish this is presented below). At runtime an interpolated quaternion requires just a single normalization call:

```
in vec4 Orientation;
vec4 quat = normalize(Orientation);
```

Note that it is possible to not keep handedness as a vertex attribute (and pass it as a uniform instead) if one decides to split the mesh into two, each part containing all faces of the same handedness. In this case we would need to draw two meshes instead of one for each mesh consisting of faces with different handedness. This would save us a single floating-point operation in vertex attributes, but it would almost double the number of draw calls in a scene.

We can therefore conclude that quaternions require special handling: additional common routines in shaders, an additional preparation stage in the exporter (see Algorithm 2.2), and correct handedness application. But if they are handled properly, quaternions significantly reduce the bandwidth utilization and storage requirements. In advanced scenarios, they are extremely cheap computation-wise, compared to matrices.

2.6 Interpolation Problem with Quaternions

The interpolation problem originates from the fact that Q and $-$Q represent exactly the same orientation (Q is any quaternion). Hence, by interpolating between the same orientation represented with different signs, we produce a range of quaternions covering the hypersphere instead of just a constant value. In practice, this means that in a fragment shader the quaternion interpolated between vertex orientations will, in general, be incorrect.

The solution for this problem is a special preprocessing technique on the mesh, performed during the export stage. The technique ensures that dot(Q1,Q2)>=0 for each two orientations of vertices in the triangle, forcing all vertices of the triangle to be in the same four-dimensional hemisphere. The algorithm for processing each triangle is given in (Algorithm 2.2).

Algorithm 2.2. (Triangle preprocessing that guarantees correct quaternion interpolation.)

let Negative pair = a vertex pair, whose quaternions produce a negative dot product.
let N = number of negative pairs in the current triangle.
to Negate quaternion = multiply all its components by −1.
if N==0: OK, exit
if N==2: clone the common vertex (between two given negative pairs), negate the cloned quaternion; go to N==0 case
if N==3: clone any vertex, negating the quaternion; go to N==1 case
if N==1: divide the edge between a negative pair by inserting a vertex in the middle with attributes averaged from the pair (including quaternion); go to N==0 case.

As you can see, this procedure duplicates some vertices and even creates some faces. This doesn't look good at first, because we don't want to complicate meshes for drawing. However, the actual number of vertices added is quite small. For example, the test 10 K vertex mesh gains only 7.5% additional vertices and 0.14% additional faces.

While interpolation with quaternions is a complicated process, a solution is available preprocessing the data as described in Algorithm 2.2. Additionally, you may not need to interpolate the TBN of the vertices at all. For Phong lighting, for example, you can interpolate the light and camera vectors in tangent space. Interpolation of TBN quaternions (aside from being handy) is needed when lighting and normal extraction are separated (as for deferred techniques). When using this algorithm, the handedness becomes a property of a face as well as a property of a vertex, because each face now links vertices of the same handedness.

2.7 KRI Engine: An Example Application

The KRI engine's rendering pipeline works in the OpenGL 3 core context and does not contain matrices, but instead uses quaternions for both spatial transformations and tangent spaces.

2.7.1 Export

The exporter from Blender is written in Python. It computes a (normal, tangent) pair for each vertex, producing a (quaternion, handedness) pair. A special stage (Algorithm 2.2) duplicates a small number of vertices and faces in order to ensure the correct quaternion interpolation for the rendering. Transformation matrices for each spatial node are also converted into (quaternion, position, scale) form, ensuring uniform scale and uniform handedness across the scene.

2.7.2 Storage

Tangent space for vertices (as a quaternion) is stored as a `float vec4` attribute, and the handedness occupies the `Position.W` component. Spatial data uses the (quaternion, position, scale) form and is passed to the GL context as two vectors.

2.7.3 Skinning

Vertex data is skinned in a separate stage using transform feedback. Bone transformations are interpolated (on the CPU) and weighted (on the GPU) using a dual-quaternion representation (this is optional). Smaller spatial structure size allows the passing of 50% more bones in the shader uniform storage.

2.7.4 Render

A special GLSL object file (`quat_v.glsl`) is attached to each program before linking. It contains useful quaternion transformation functions that are missing from the GLSL specification. There are render components that use forward and deferred Phong lighting, as well as surface baking into UV textures (e.g., emitting particles/hair from the mesh surface, as used in [Malyshau 10b]).

An example of typical vertex transformation GLSL code (excluding projection) is given here:

```
uniform struct Spatial  {
    vec4 pos, rot;      // camera->world and model->world transforms
}s_cam, s_model;
// vertex attributes of position and quaternion
in vec4 at_vertex, at_quat;
// forward transform of a vertex (from quat_v.glsl)
vec3 trans_for(vec3,Spatial);
// inverse transform of a vertex (from quat_v.glsl)
vec3 trans_inv(vec3,Spatial);
. . .
vec3 v_world = trans_for(at_vertex.xyz, s_model); // world-space position
vec3 v_cam = trans_inv(v_world, s_cam);     // camera-space position
```

2.8 Conclusion

The OpenGL 3 core / ES 2.0 pipeline no longer pushes the programmer to use matrices. In this article, we showed some advantages of using quaternions and pointed out the complexities associated with them. We also used the KRI engine as a proof-of-concept for the idea of using quaternions as a full-scale matrix replacement in a real-world game engine scenario. We believe that quaternions will have a future much brighter than their past.

Bibliography

[Banks 94] Dadid C. Banks. "Illumination in Diverse Codimensions." In *Proceedings of SIGGRAPH '94, Computer Graphics Proceedings*. Annual Conference Series, edited by Andrew Glassner, pp. 327–334, New York: ACM Press, 1994.

[Blow 04] Jonathan Blow. "Understanding Slerp, Then Not Using It." *The Inner Product*. Available at http://number-none.com/product, Apr 2004.

[Gruber 00] Diana Gruber. "Do We Really Need Quaternions?" Game Dev.net. Available at http://www.gamedev.net/page/resources/_/do-we-really-need-quaternions-r1199, Sept 2000.

[Hast 05] Anders Hast. "Shading by Quaternion Interpolation." In WSCG (Short Papers), 53–56, 2005. Available at http://wscg.zcu.cz/wscg2005/Papers_2005/Short/B61-full.pdf.

[Kavan et. al 06] Ladislav Kavan, Steven Collins, Carol O'Sullivan, and Jiri Zara. "Dual Quaternions for Rigid Transformation Blending." Technical report TCD-CS-2006-46, Trinity College, Dublin, 2006.

[Malyshau 10a] Dzmitry Malyshau. "Quaternions." *KRI Engine*. Available at http://code.google.com/p/kri/wiki/Quaternions, 2010.

[Malyshau 10b] Dzmitry Malyshau. "Real-Time Dynamic Fur on the GPU." GameDev.net. Available at http://www.gamedev.net/page/resources/_//feature/fprogramming/real-time-dynamic-fur-on-the-gpu-r2774, Oct 2010.

[McMahon 03] Joe McMahon. "A (Mostly) Linear Algebraic Introduction to Quaternions." Program in Applied Mathematics, University of Arizona, Fall 2003.

[Svarovsky 00] J. Svarovsky. "Quaternions for Game Programming." In *Game Programming Gems*, edited by Mark DeLoura, pp. 195–299. Hingham, MA: Charles River Media, 2000.

[Void 03] Sobeit Void. "Quaternion Powers." GameDev.net. Available at http://www.gamedev.net/page/resources/_/reference/programming/math-and-physics/quaternions/quaternion-powers-r1095, Feb 2003.