Quaternions

Discovered by Sir William Hamilton, 1843, while on a walk in Dublin.
Legend says that he was so excited that he took out a knife and carved the equation into the stone of a bridge.
Good thing spray-paint hadn’t been invented yet…
Quaternions have 4 elements, one real and three complex:

A Quaternion can represent a point P like this:

A Quaternion can record a rotation transformation by an angle θ about an axis ħ like this:

Performing Rotations with Quaternions
A Quaternion can record a rotation transformation by an angle θ about an axis ħ like this:

Where:

Concatenated Quaternion Rotations are handled like this:

A Quaternion can represent a point P like this:

A rotated point, P' by one rotation is:

A rotated point, P' by multiple rotations is:

A Useful Concept: Spherical Linear Interpolation

Quaternions have 4 elements, one real and three complex:

By definition, we always force by making

And, by definition, we always force |q| = 1, by making

Where:

Note:

Complex conjugate:

Complex inverse:

Performing Rotations with Quaternions
A Quaternion can record a rotation transformation by an angle θ about an axis ħ like this:

Where:

Concatenated Quaternion Rotations are handled like this:

A Quaternion can represent a point P like this:

A rotated point, P' by one rotation is:

A rotated point, P' by multiple rotations is:
For this to be correct, A must be a unit vector.

Note that the sum of the Trace (diagonal) elements is: 

\[ 3c + (1 - c) = 1 + 2 \cos \theta \]

Letting \( t = \text{Trace}(R) = 1 + 2 \cos \theta \), then:

\[ \cos \theta = \frac{1}{2} (t - 1) \]
\[ \sin \theta = \sqrt{1 - \cos^2 \theta} \]

If we let

\[ d = \sqrt{a^2 + b^2 + c^2} \]

then

\[ \hat{n} = \left( \frac{a}{d}, \frac{b}{d}, \frac{c}{d} \right) \]

Note that the sum of the Trace (diagonal) elements is: 

\[ 3c + (1 - c) = 1 + 2 \cos \theta \]

Notes from Converting from a Quaternion to a Matrix

\[
R = \begin{bmatrix}
q_0^2 + q_1^2 - q_2^2 - q_3^2 & 2q_0q_1 - 2q_2q_3 & 2q_0q_2 + 2q_1q_3 \\
2q_0q_1 + 2q_2q_3 & q_0^2 - q_1^2 + q_2^2 - q_3^2 & 2q_0q_3 - 2q_1q_2 \\
2q_0q_2 - 2q_1q_3 & 2q_0q_3 + 2q_1q_2 & q_0^2 - q_1^2 - q_2^2 + q_3^2
\end{bmatrix}
\]

where:

\[
c = \cos \theta \\
n = \sin \theta
\]

Converting from a Matrix to a Quaternion

\[
R(Q) = \begin{bmatrix}
q_0^2 + q_1^2 - q_2^2 - q_3^2 & 2q_0q_1 - 2q_2q_3 & 2q_0q_2 + 2q_1q_3 \\
2q_0q_1 + 2q_2q_3 & q_0^2 - q_1^2 + q_2^2 - q_3^2 & 2q_0q_3 - 2q_1q_2 \\
2q_0q_2 - 2q_1q_3 & 2q_0q_3 + 2q_1q_2 & q_0^2 - q_1^2 - q_2^2 + q_3^2
\end{bmatrix}
\]

\[
R = \begin{bmatrix}
c + n^2(1 - c) & n_x n_y (1 - c) - n_z & n_x n_z (1 - c) + n_y \\
n_x n_y (1 - c) + n_z & c + n^2(1 - c) & n_y n_z (1 - c) - n_x \\
n_x n_z (1 - c) - n_y & n_y n_z (1 - c) + n_x & c + n^2(1 - c)
\end{bmatrix}
\]

where:

\[
c = \cos \theta \\
n = \sin \theta
\]
Quaternions in GLM

```cpp
#include "glm/vec2.hpp"
#include "glm/vec3.hpp"
#include "glm/mat4x4.hpp"
#include "glm/gtc/matrix_transform.hpp"
#include "glm/gtc/matrix_inverse.hpp"
#include "gtc/quaternion.hpp"
#include "glm/gtx/quaternion.hpp"

glm::quat rot1 = glm::angleAxis(glm::radians(45.f), glm::vec3(0.707f, 0.707f, 0.));
glm::quat rot2 = glm::angleAxis(glm::radians(90.f), glm::vec3(1., 0., 0.,));
glm::quat combinedRots = rot2 * rot1;
glm::vec4 v = glm::vec4(1., 1., 1., 1.);
glm::vec4 vp = combinedRots * v;
glm::mat4 rotMatrix = glm::toMat4(combinedRots);
glm::vec4 vpp = rotMatrix * v; // same result as vp
```

Converting from a Quaternion to OpenGL

```cpp
glRotate3f(θ°, nx, ny, nz);
glm::rotate(glm::quat const & q, θR, glm::vec3( nx, ny, nz ) );
```

A Useful Concept: Spherical Linear Interpolation, where P and Q are Quaternions

Q(t) = \sin((1-t)\phi) P + \sin(\phi) Q, 0 ≤ t ≤ 1.

where:

\[ \cos \phi = P \cdot Q = (p_x q_x + p_y q_y + p_z q_z) \]

\[ \sin \phi = \sqrt{1 - \cos^2 \phi} \]

My Code (Quat.h, Quat.cpp)

```cpp
rotate( 
    float t, 
    const Quat& p, 
    const Quat& q ) 
{
    float angr; // angle between p and q in radians
    float c, s; // cosine and sine of the angle between p and q
    float cp, cq; // coefficients to multiply quaternions p and q
    Rotate r;
    // dot product to get the angle between p and q:
    c = p.s*q.s + p.vx*q.vx + p.vy*q.vy + p.vz*q.vz;
    angr = acos(c);
    // sine of that angle:
    s = sin(angr);
    // if the sine is 0., then p == q:
    if( s == 0. )
    {
        r = p;
        return r;
    }
    // do spherical interpolation:
    cp = sin((1.-t)*angr) / s;
    cq = sin(t * angr) / s;
    r = cp*p + cq*q;
    return r;
}
```

My Code (Quat.h, Quat.cpp)

```cpp
int main(int argc, char *argv[]) {
    Rotate r1 = Rotate(45.*D2R, 0., 0., 1.);
    Rotate r2 = Rotate(90.*D2R, 0., 0., 1.);
    Rotate r3 = Rotate(90.*D2R, 1., 0., 0.);
    Rotate r4 = r2 * r1;
    Rotate r5 = r3 * r2 * r1;
    fprintf(stderr, "r1       = %s
", r1.toString());
    fprintf(stderr, "r2       = %s
", r2.toString());
    fprintf(stderr, "r2*r1    = %s
", r4.toString());
    fprintf(stderr, "r3       = %s
", r3.toString());
    fprintf(stderr, "r3*r2*r1 = %s
", r5.toString());
    fprintf(stderr, "r3*r2*r1 matrix =
" );
    r5.printMatrix();
    Point p1 = Point(1., 1., 0.);
    Point p2 = r4 * p1;
    fprintf(stderr, "Original point    = %s
", p1.toString());
    fprintf(stderr, "Transformed point = %s
", p2.toString());
    // try interpolating from r1 to r5:
    const int N = 10;
    float dt = 1. / (float)(N - 1);
    float t = 0.;
    for( int i = 0; i < N; i++, t += dt )
    {
        Rotate r15 = Slerp(t, r1, r5);
        fprintf(stderr, "%2d   %5.3f   %s
", i, t, r15.toString());
    }
}
```

My Code (Quat.h, Quat.cpp)
A Really Good (i.e., Complete) Reference

Andrew Hanson, Visualizing Quaternions, Morgan-Kaufmann, 2006.