One man, one computer, one obsession...

Perone's Programming Pad

ANDROGRADE
BEAT ME UP TOO
BEAT ME UP
TECHNOPOLY
FFS
PROJECTS
CODE >
COLITIS >
GENEALOGY
RESUME
YOUTUBE
ABOUT ME (2006)
EMAIL ME
BONUS

Quaternion C++ Class

Quaternions are commonly used to represent orientations and rotations in 3d games. They are more efficient than using 3d matrices in number of operations, storage and quaternions also avoid gimbal lock. You can see the latest version of the entire math package at **GitHub** or visit my **downloadable code page**.

```
/****************************************
 * Quaternion class
 * By Will Perone
 * Original: 12-09-2003
 * Revised:  27-09-2003
 *           22-11-2003
 *           10-12-2003
 *           15-01-2004
 *           16-04-2004
 *           29-07-2011   added corrections from website,
 *           22-12-2011   added correction to *= operator, thanks Steve Rogers
 *
 * Notes:
 * if |q|=1 then q is a unit quaternion
 * if q=(0,v) then q is a pure quaternion
 * if |q|=1 then q conjugate = q inverse
 * if |q|=1 then q= [cos(angle), u*sin(angle)] where u is a unit vector
 * q and -q represent the same rotation
 * q*q.conjugate = (q.length_squared, 0)
 * ln(cos(theta),sin(theta)*v)= ln(e^(theta*v))= (0, theta*v)
 ****************************************/

#pragma once

#include "matrix4.h"
#include "assert.h"


struct quaternion
{
	union {
		struct {
			float    s; //!< the real component
			vector3f v; //!< the imaginary components
		};
		struct { float elem[4]; }; //! the raw elements of the quaternion
	};


	//! constructors
	quaternion() {}
	quaternion(float real, float x, float y, float z): s(real), v(x,y,z) {}
	quaternion(float real, const vector3f &i): s(real), v(i) {}

	//! from 3 euler angles
	quaternion(float theta_z, float theta_y, float theta_x)
	{
		float cos_z_2 = cosf(0.5*theta_z);
		float cos_y_2 = cosf(0.5*theta_y);
		float cos_x_2 = cosf(0.5*theta_x);

		float sin_z_2 = sinf(0.5*theta_z);
		float sin_y_2 = sinf(0.5*theta_y);
		float sin_x_2 = sinf(0.5*theta_x);
```

```
        // and now compute quaternion
        s   = cos_z_2*cos_y_2*cos_x_2 + sin_z_2*sin_y_2*sin_x_2;
        v.x = cos_z_2*cos_y_2*sin_x_2 - sin_z_2*sin_y_2*cos_x_2;
        v.y = cos_z_2*sin_y_2*cos_x_2 + sin_z_2*cos_y_2*sin_x_2;
        v.z = sin_z_2*cos_y_2*cos_x_2 - cos_z_2*sin_y_2*sin_x_2;

}

//! from 3 euler angles
quaternion(const vector3f &angles)
{
        float cos_z_2 = cosf(0.5*angles.z);
        float cos_y_2 = cosf(0.5*angles.y);
        float cos_x_2 = cosf(0.5*angles.x);

        float sin_z_2 = sinf(0.5*angles.z);
        float sin_y_2 = sinf(0.5*angles.y);
        float sin_x_2 = sinf(0.5*angles.x);

        // and now compute quaternion
        s   = cos_z_2*cos_y_2*cos_x_2 + sin_z_2*sin_y_2*sin_x_2;
        v.x = cos_z_2*cos_y_2*sin_x_2 - sin_z_2*sin_y_2*cos_x_2;
        v.y = cos_z_2*sin_y_2*cos_x_2 + sin_z_2*cos_y_2*sin_x_2;
        v.z = sin_z_2*cos_y_2*cos_x_2 - cos_z_2*sin_y_2*sin_x_2;
}

//! basic operations
quaternion &operator =(const quaternion &q)
{ s= q.s; v= q.v; return *this; }

const quaternion operator +(const quaternion &q) const
{ return quaternion(s+q.s, v+q.v); }

const quaternion operator -(const quaternion &q) const
{ return quaternion(s-q.s, v-q.v); }

const quaternion operator *(const quaternion &q) const
{       return quaternion(s*q.s - v*q.v,
                        v.y*q.v.z - v.z*q.v.y + s*q.v.x + v.x*q.s,
                        v.z*q.v.x - v.x*q.v.z + s*q.v.y + v.y*q.s,
                        v.x*q.v.y - v.y*q.v.x + s*q.v.z + v.z*q.s);
}

const quaternion operator /(const quaternion &q) const
{
        quaternion p(q);
        p.invert();
        return *this * p;
}

const quaternion operator *(float scale) const
{ return quaternion(s*scale,v*scale); }

const quaternion operator /(float scale) const
{ return quaternion(s/scale,v/scale); }

const quaternion operator -() const
{ return quaternion(-s, -v); }

const quaternion &operator +=(const quaternion &q)
{ v+=q.v; s+=q.s; return *this; }

const quaternion &operator -=(const quaternion &q)
{ v-=q.v; s-=q.s; return *this; }

const quaternion &operator *=(const quaternion &q)
{
        float x= v.x, y= v.y, z= v.z, sn= s*q.s - v*q.v;
        v.x= y*q.v.z - z*q.v.y + s*q.v.x + x*q.s;
        v.y= z*q.v.x - x*q.v.z + s*q.v.y + y*q.s;
```

```
        v.z= x*q.v.y - y*q.v.x + s*q.v.z + z*q.s;
        s= sn;
        return *this;
}

const quaternion &operator *= (float scale)
{ v*=scale; s*=scale; return *this; }

const quaternion &operator /= (float scale)
{ v/=scale; s/=scale; return *this; }


//! gets the length of this quaternion
float length() const
{ return (float)sqrt(s*s + v*v); }

//! gets the squared length of this quaternion
float length_squared() const
{ return (float)(s*s + v*v); }

//! normalizes this quaternion
void normalize()
{ *this/=length(); }

//! returns the normalized version of this quaternion
quaternion normalized() const
{ return  *this/length(); }

//! computes the conjugate of this quaternion
void conjugate()
{ v=-v; }

//! inverts this quaternion
void invert()
{ conjugate(); *this/=length_squared(); }

//! returns the logarithm of a quaternion = v*a where q = [cos(a),v*sin(a)]
quaternion log() const
{
        float a = (float)acos(s);
        float sina = (float)sin(a);
        quaternion ret;

        ret.s = 0;
        if (sina > 0)
        {
                ret.v.x = a*v.x/sina;
                ret.v.y = a*v.y/sina;
                ret.v.z = a*v.z/sina;
        } else {
                ret.v.x= ret.v.y= ret.v.z= 0;
        }
        return ret;
}

//! returns e^quaternion = exp(v*a) = [cos(a),vsin(a)]
quaternion exp() const
{
        float a = (float)v.length();
        float sina = (float)sin(a);
        float cosa = (float)cos(a);
        quaternion ret;

        ret.s = cosa;
        if (a > 0)
        {
                ret.v.x = sina * v.x / a;
                ret.v.y = sina * v.y / a;
                ret.v.z = sina * v.z / a;
        } else {
```

```
            ret.v.x = ret.v.y = ret.v.z = 0;
        }
        return ret;
    }

    //! casting to a 4x4 isomorphic matrix for right multiplication with vector
    operator matrix4() const
    {
        return matrix4(s,  -v.x, -v.y,-v.z,
                    v.x,  s,  -v.z, v.y,
                    v.y, v.z,    s,-v.x,
                    v.z,-v.y,  v.x,   s);
    }

    //! casting to 3x3 rotation matrix
    operator matrix3() const
    {
        Assert(length() > 0.9999 && length() < 1.0001, "quaternion is not normalized");
        return matrix3(1-2*(v.y*v.y+v.z*v.z), 2*(v.x*v.y-s*v.z),   2*(v.x*v.z+s*v.y),
                    2*(v.x*v.y+s*v.z),   1-2*(v.x*v.x+v.z*v.z), 2*(v.y*v.z-s*v.x),
                    2*(v.x*v.z-s*v.y),   2*(v.y*v.z+s*v.x),   1-2*(v.x*v.x+v.y*v.y));
    }

    //! computes the dot product of 2 quaternions
    static inline float dot(const quaternion &q1, const quaternion &q2)
    { return q1.v*q2.v + q1.s*q2.s; }

    //! linear quaternion interpolation
    static quaternion lerp(const quaternion &q1, const quaternion &q2, float t)
    { return (q1*(1-t) + q2*t).normalized(); }

    //! spherical linear interpolation
    static quaternion slerp(const quaternion &q1, const quaternion &q2, float t)
    {
        quaternion q3;
        float dot = quaternion::dot(q1, q2);

        /*  dot = cos(theta)
            if (dot < 0), q1 and q2 are more than 90 degrees apart,
            so we can invert one to reduce spinning */
        if (dot < 0)
        {
            dot = -dot;
            q3 = -q2;
        } else q3 = q2;

        if (dot < 0.95f)
        {
            float angle = acosf(dot);
            return (q1*sinf(angle*(1-t)) + q3*sinf(angle*t))/sinf(angle);
        } else // if the angle is small, use linear interpolation
            return lerp(q1,q3,t);
    }

    //! This version of slerp, used by squad, does not check for theta > 90.
    static quaternion slerpNoInvert(const quaternion &q1, const quaternion &q2, float t)
    {
        float dot = quaternion::dot(q1, q2);

        if (dot > -0.95f && dot < 0.95f)
        {
            float angle = acosf(dot);
            return (q1*sinf(angle*(1-t)) + q2*sinf(angle*t))/sinf(angle);
        } else  // if the angle is small, use linear interpolation
            return lerp(q1,q2,t);
    }

    //! spherical cubic interpolation
    static quaternion squad(const quaternion &q1,const quaternion &q2,const quaternion &a,const quaternion &b,float t)
    {
```

```
    quaternion c= slerpNoInvert(q1,q2,t),
               d= slerpNoInvert(a,b,t);
    return slerpNoInvert(c,d,2*t*(1-t));
}

//! Shoemake-Bezier interpolation using De Castlejau algorithm
static quaternion bezier(const quaternion &q1,const quaternion &q2,const quaternion &a,const quaternion &b,float t)
{
    // level 1
    quaternion q11= slerpNoInvert(q1,a,t),
               q12= slerpNoInvert(a,b,t),
               q13= slerpNoInvert(b,q2,t);
    // level 2 and 3
    return slerpNoInvert(slerpNoInvert(q11,q12,t), slerpNoInvert(q12,q13,t), t);
}

//! Given 3 quaternions, qn-1,qn and qn+1, calculate a control point to be used in spline interpolation
static quaternion spline(const quaternion &qnm1,const quaternion &qn,const quaternion &qnp1)
{
    quaternion qni(qn.s, -qn.v);
    return qn * (( (qni*qnm1).log()+(qni*qnp1).log() )/-4).exp();
}

//! converts from a normalized axis - angle pair rotation to a quaternion
static inline quaternion from_axis_angle(const vector3f &axis, float angle)
{ return quaternion(cosf(angle/2), axis*sinf(angle/2)); }

//! returns the axis and angle of this unit quaternion
void to_axis_angle(vector3f &axis, float &angle) const
{
    angle = acosf(s);

    // pre-compute to save time
    float sinf_theta_inv = 1.0/sinf(angle);

    // now the vector
    axis.x = v.x*sinf_theta_inv;
    axis.y = v.y*sinf_theta_inv;
    axis.z = v.z*sinf_theta_inv;

    // multiply by 2
    angle*=2;
}

//! rotates v by this quaternion (quaternion must be unit)
vector3f rotate(const vector3f &v)
{
    quaternion V(0, v);
    quaternion conjugate(*this);
    conjugate.conjugate();
    return (*this * V * conjugate).v;
}

//! returns the euler angles from a rotation quaternion
vector3f euler_angles(bool homogenous=true) const
{
    float sqw = s*s;
    float sqx = v.x*v.x;
    float sqy = v.y*v.y;
    float sqz = v.z*v.z;

    vector3f euler;
    if (homogenous) {
        euler.x = atan2f(2.f * (v.x*v.y + v.z*s), sqx - sqy - sqz + sqw);
        euler.y = asinf(-2.f * (v.x*v.z - v.y*s));
        euler.z = atan2f(2.f * (v.y*v.z + v.x*s), -sqx - sqy + sqz + sqw);
    } else {
        euler.x = atan2f(2.f * (v.z*v.y + v.x*s), 1 - 2*(sqx + sqy));
        euler.y = asinf(-2.f * (v.x*v.z - v.y*s));
        euler.z = atan2f(2.f * (v.x*v.y + v.z*s), 1 - 2*(sqy + sqz));
```

```
        }
        return euler;
    }
};
```

29 Comments

Your Name
In the Quaternion constructor, shouldn't the divide by half be on the inside of the trig? i.e., sin(theta *.5f)? 61 21

Will
good catch. 20 0

rsyndrome
I think operator*= doesn't work properly as you need to store previously a copy of v. Indeed new y and z components may be invalid since v.x has already been altered. 3 0

Will
another good catch, thanks! 8 0

Abtin
Hey, 0 54

I think there are some problem in computing euler angle from quaternion. I changed it to code below and it worked. All came from wikipedia page:
http://en.wikipedia.org/wiki/Conversion_between_quaternions_and_Euler_angles

euler.x = atan2f(2.f * (v.z*v.y + v.x*s), 1 - 2* (sqx + sqy));
euler.y = asinf(-2.f * (v.x*v.z - v.y*s));
euler.z = atan2f(2.f * (v.x*v.y + v.z*s), 1 - 2* (sqy + sqz));

jon
what's the point of having non-member inline functions static? An inline function has internal linkage, so static does nothing. 1 0

Will
They are member functions. 0 0

Dan *2011/04/06*
Thanks for this. It helped me do a c++ assignment designing user controlled bezier curves using quaternions. 0 0

luneart *2011/04/22*
@ Abtin: 0 0
look at the url you gave !
your way is the inhomogeneous way, Will used the homogeneous one.

Hans *2011/06/26*
You have not applied the above corrections in your downloadable package yet. 0 0

Will *2011/06/29*
Thanks Hans, didn't think people were still downloading my package; I'll correct that. 0 0

Susie *2011/07/08* **Contact Me**
Great hammer of Thor, that is powerfully hefulpl! 0 0

Nelle *2011/07/09* **Contact Me**
This ifnormaiton is off the hizool! 11 0

E *2011/07/21*
I think some revision control is in order. The version shown on this page is different from the ones in your downloadable zip files. Also the last revision date (16-04-2004) must be wrong given the dates of the comments above. 0 0

Will *2011/07/22*
Yes I know, I should get with the times and put it in some sort of public repo. 0 0

Will *2011/07/29* **Contact Me**
Thank for all your feedback, I've updated the class above and the source code zip file with the latest corrections. 0 0

Jake *2011/09/09* **Contact Me**
I can't seem to compile this in ubuntu. Is it portable? 0 0

Will *2011/09/09*
Could be a difference in how gcc interprets some stuff like the templates, I haven't tried compiling it in gcc, 0 0

feedback would be great.

Jake *2011/09/09* **Contact Me** 0 0

here are some examples of errors it spit out. The full list is too long to reproduce here. First it complains that a bunch of files are missing. But that's easy to fix, because they're just capitalized differently e.g.

#include "matrix3.h"

but the file is called Matrix3.h . More difficult to fix are errors like

vector3.h: In member function 'void vector3<T>::operator()(T, T, T)':
vector3.h:53: error: 'x' was not declared in this scope

or

utility.h:137: error: ISO C++ forbids declaration of 'interp_lin' with no type

Jake *2011/09/09* **Contact Me** 0 0

If you want, I can give you the whole error list

Will *2011/09/09* **Contact Me** 0 0

send me the whole error list to my email, I'll take a look. It's been some years since I worked on this codebase but might find something.

Will *2011/12/23* 0 0

Just a note, I've updated the Quaternion class with a correction to the *= operator. The downloadable zip file has also been updated.

Colin *2012/04/01* 13 0

Your awesome. Your math code has saved me a large amount of time. I worship the ground you've walked upon.

Clemens *2012/05/05* 0 0

Thank you so very much for this comprehensive take on Quaternions. You are very awesome indeed, Sir!

Jon *2012/09/27* 0 0

Yes - this is great work. Did you build a unit test suite for this (as a born sceptic I like to see proof everything works as it should). If you didn't, no worries, and if I get round to it I will send you.

Will *2012/09/27* **Contact Me** 0 0

Thanks for the complement! No I never built a unit test for it, I made it so long ago... Though for something complicated like Quaternions that is probably a good idea. If you come up with one let me know!

Will *2012/10/23* 5 0

This code is now available on my GitHub at https://github.com/MegaManSE/willperone/tree/master/Math

foakleys sale *2013/04/19* **Contact Me** 0 10

The project refused to implement weak cryptography, even where the RFCs required it. Their position was that the RFCs had unfortunately been subverted into including weak methods, but there was still no excuse for actually implementing those. Among the things rejected were null encryption, single DES, and Oakley Group 1. This did not generally lead to interoperation problems, even though those were the only required algorithms in the RFCs. Almost everyone implemented the more secure Triple DES and groups two and five, so almost everyone could talk to FreeS/WAN. Some users wanted single DES; the project explicitly refused to provide any assistance for that. foakleys sale

sergii *2013/05/23* **Contact Me** 0 0

should not exp(q) also multiply by exp(Re(q)) ?

I beleive for
q = [s, v], were s= Re(q) and v = Im(q)
the formula is:
exp(q) = exp(s)*(cos(|v|) + sin(|v|)*v/|v|);

Your Name | Email (optional) <- for private contact

Comment

Comment