Quaternion

```
double quat::Length() const {
	return sqrt(sqr(w)+sqr(v[0]) + sqr(v[1]) + sqr(v[2]));
}

void quat::Normalize() {
	double len = Length();
	assert(IsNotZero(len));

	w /= len;
	v[0] /= len;
	v[1] /= len;
	v[2] /= len;
}

quat quat::conj(void) const {
	return quat(w, -v[0], -v[1], -v[2]);
}

quat quat::inverse(void) const{
	quat temp = conj();

	double lengthSqr = sqr(Length());
	assert(lengthSqr);

	temp.w /= lengthSqr;
	temp.v[0] /= lengthSqr;
	temp.v[1] /= lengthSqr;
	temp.v[2] /= lengthSqr;

	return temp;
}

double quat::scale(void) const {
	return Length();
}

double quat::angle(void) const {
	quat temp(w,v);
	temp.Normalize();

	return 2 * acos(temp.w);
}

double quat::cosTheta(void) const {
	return cos(angle()/2);
}

double quat::sinTheta(void) const {
	return sin(angle()/2);
}

vector quat::unitV(void) const {
	vector temp = v;

	temp.Normalize();

	return temp;
}
```

Quaternion

```
double quat::angleU(void) const {
	return 2 * acos(w);
}

double quat::cosThetaU(void) const {
	return cos(angleU()/2);
}

double quat::sinThetaU(void) const {
	return sin(angleU()/2);
}

quat &quat::operator+=(const quat &r){
	w += r.w;
	v += r.v;

	return *this;
}

quat &quat::operator-=(const quat &r){
	w -= r.w;
	v -= r.v;

	return *this;
}

quat &quat::operator*=(const quat &r){
	w = (w * r.w) - Dot(v,r.v);
	v = (r.w * v) + (w * r.v) + Cross(v, r.v);

	return *this;
}

quat &quat::operator/=(const quat &r){
    quat rQ(r);
    rQ.inverse();
    return *this * rQ;

	return *this;
}

vector quat::rotate(vector vR) const {
	quat vQ(0,vR);
	quat conjugate(*this);
	conjugate.conj();

	return (*this * vQ * conjugate).v;
}

void quat::QuatFromAngleAxis(double a[4]){
	double sin_a = sin( a[0] / 2 );

	w = cos( a[0] / 2 );
	v[0] = a[1] * sin_a;
	v[1] = a[2] * sin_a;
	v[2] = a[3] * sin_a;
}
```

Quaternion

```
void quat::QuatFromAngleAxis(double theta, vector V){
	double sin_a = sin( theta / 2 );

	w = cos( theta / 2 );
	v[0] = V[0] * sin_a;
	v[1] = V[1] * sin_a;
	v[2] = V[2] * sin_a;
}

void quat::QuatToAngleAxis(double a[4]) const {
	a[0] = acos(w);

	double sin_a = 1 / sin( a[0] );

	a[0] *= 2;

	a[1] = v[0] * sin_a;
	a[2] = v[1] * sin_a;
	a[3] = v[2] * sin_a;
}

void quat::QuatToAngleAxis(double &theta, vector &V) const{
	theta = acos(w);

	double sin_a = 1 / sin( theta );

	theta *= 2;

	V[0] = v[0] / sin_a;
	V[1] = v[1] / sin_a;
	V[2] = v[2] / sin_a;
}

void quat::QuatFromEulerAngles(double x, double y, double z){
	QuatFromAngleAxis(z, vector(0,0,1));
	quat zQ(w,v);
	QuatFromAngleAxis(y, vector(0,1,0));
	quat yQ(w,v);
	QuatFromAngleAxis(x, vector(1,0,0));
	quat xQ(w,v);



	quat finalQ = xQ * yQ * zQ;

	w = finalQ.w;
	v = finalQ.v;
}

quat operator-(const quat &q){
	return quat(-q.w, -q.v[0], -q.v[1], -q.v[2]);
}

quat operator+(const quat &q1, const quat &q2){
	return quat(q1.w + q2.w, q1.v[0] + q2.v[0], q1.v[1] + q2.v[1], q1.v[2] + q2.v[2]);
}
```

Quaternion

```
quat operator-(const quat &q1, const quat &q2){
	return quat(q1.w - q2.w, q1.v[0] - q2.v[0], q1.v[1] - q2.v[1], q1.v[2] - q2.v[2]);
}

quat operator*(const quat &q1, const quat &q2){
	vector temp = (q2.w * q1.v) + (q1.w * q2.v) + Cross(q1.v, q2.v);

	return quat((q1.w * q2.w) - Dot(q1.v,q2.v), temp[0], temp[1], temp[2]);
}

quat operator/(const quat &q1, const quat &q2){
    quat rQ(q2);
    rQ.inverse();

    return q1 * rQ;
}

double dot(const quat &a, const quat &b){
	return a.w*b.w + a.v[0]*b.v[0] + a.v[1]*b.v[1] + a.v[2]*b.v[2];
}

ostream &operator<<(ostream &output, const quat &q){
	output << "(" << q.w << ", <" << q.v[0] << ", " << q.v[1] << ", " << q.v[2] <<
		">)";
	return output;
}

void QuatInterp::SortKeys(void){
	bool found;

	do
	{
		found = false;
		for (int i = nKeys - 1; i > 0; i--)
		{
			if (keys[i-1].t > keys[i].t)
			{
				QuatKey tmp = keys[i];
				keys[i] = keys[i-1];
				keys[i-1] = tmp;
				found = true;
			}
		}
	} while (found == true);
}

int QuatInterp::FindInterval(double t){
	int i = 0;

	while (i + 1 < nKeys && keys[i + 1].t < t){
		i++;
	}

	return i;
}
```

Quaternion

```
QuatInterp *QuatInterp::LoadQuatInterpolator(Loader &input)
{
	char type[256];
	QuatInterp *result = NULL;

	input.ReadToken(type);
	if (strcmp(type, "slerp") == 0)
		result = new slerp;
	else if (strcmp(type, "bezier") == 0)
		result = new bezierQuat;
	else
		input.Error(type, "Unknown quaternion interpolator type");

	result->Load(input);
	return result;
}

void QuatInterp::Load(Loader &input)
{
	char token[256], tmp[256];

	if (!input.ReadInt(nKeys))
		input.Error("Interpolator object must begin with a key count");

	input.ReadToken(token);
	for (int i = 0; i < nKeys; i++)
	{
		if (strcmp(token, "key") != 0)
			input.Error("Each key must begin with the keyword 'key'");

		if (!input.ReadDouble(keys[i].t))
			input.Error("Key %d must begin with a time", i);

		do
		{
			input.ReadToken(token);

			if (strncmp(token, "val", 3) == 0)
			{
				input.ReadToken(tmp);
				if (strncmp(tmp, "quat", 4) == 0)
				{
					if (!input.ReadDouble(keys[i].q.w))
						input.Error("A quaternion key value must begin
							with a scalar 'w' value");

					for (int j = 0; j < 3; j++)
						if (!input.ReadDouble(keys[i].q.v[j]))
							input.Error("Not enough coordinates for
								quaternion key vector");

					keys[i].q.Normalize();
				}
				else if (strncmp(tmp, "angleaxis", 9) == 0)
				{
					double angle;
					vector v;
```

```
                                if (!input.ReadDouble(angle))
                                        input.Error("An angle-axis value for a
                                                quaternion key must begin with an
                                                angle\n");

                                for (int j = 0; j < 3; j++)
                                        if (!input.ReadDouble(v[j]))
                                                input.Error("Not enough coordinates for
                                                        quaternion key vector");
                                keys[i].q.QuatFromAngleAxis(angle * PI / 180.0, v);

                        }
                        else if (strncmp(tmp, "euler", 5) == 0)
                        {
                                vector v;

                                for (int j = 0; j < 3; j++)
                                {
                                        if (!input.ReadDouble(v[j]))
                                                v[j] = 0.0;
                                }

                                keys[i].q.QuatFromEulerAngles(v[0] * PI / 180.0, v[1] *
                                        PI / 180.0, v[2] * PI / 180.0);

                        }
                        else input.Error("Unknown quaternion key value type: %s\n",
                                tmp);
                }
        } while (strcmp(token, "key") != 0 && strcmp(token, "end") != 0);

        if (strcmp(token, "end") == 0 && i != nKeys - 1)
                input.Error("End reached before all keys were found");
    }
}

quat slerp::Evaluate(double t)
{
    int interval = FindInterval(t);
    double s = ( t - keys[interval].t ) / ( keys[interval+1].t + keys[interval].t );

    quat q0 = keys[interval].q;
     quat q1 = keys[interval + 1].q;

    quat temp;

    double dotQ = dot(q0,q1);

    if (dotQ < 0){
        dotQ = -dotQ;
        temp = -q1;
    }else{
        temp = q1;
    }
    double angle = acos(dotQ);

    return ((q0*(sin(angle*(1-s)) )+ (temp*sin(angle*s))/sin(angle)));
}
```