Splines & Quaternions (splines.txt)

by
Jochen Wilhelmy
a.k.a.
digisnap
digisnap@cs.tu-berlin.de
06.08.1997

Introduction

This text describes the spline and quaternion mathematics which I used in the "Spotlight" 64Kb intro. It is compatible with Autodesk 3D Studio 4.0. The first section covers all one and three dimensional tracks. The second section describes the rotation track which has to be handled extra. All tracks are described with the parameters tension, bias and continuity. The text should guide you how to code this stuff. If you want more background knowledge you should look into this litarature:

[1] Computer Graphics Principles and Practice
    by Foley, van Dam, Feiner and Hughes
    ISBN 0-201-84840-6

[2] Shoemake, Ken, "Animating Rotation with Quaternion Curves"
    Proceedings of SIGGRAPH '85 (San Francisco, California, July 22-26, 1985)
    In Computer Graphics, 19(3), July 1985, ACM SIGGRAPH, New York
    (Pages 245-254)

Thanks to Admiral/Elyssis for "beta-testing" this text.

Tracks

A track consists of several keys specifying a certain property at a certain time. The property can be position, rotation, size and other things. A key contains a time (frame number), some parameters like tension, continuity and bias and a data field describing the property for the given time. In this section the data is a vector, which is one dimensional for tracks like roll track and three dimensional for the position and scale track. The keys will be called k, the vectors representing points will be called p. The point of the n'th key is therefore pn, the next one p(n+1).

You best understand the following explanations if you think of a two dimensional position track, because you can illustrate it with a pencil and a ruler on paper.

Since the time is continuous, you have to interpolate the gaps between the points. Draw some points on a paper. The simplest way to interpolate them is
to take a ruler and draw lines between the points. The formula would be

$$Q(t) = P1*(t) + P2*(1-t).$$

Assign P1 and P2 to two consecutive points in your track (pn and p(n+1)). t is a local time parameter running from 0.0 to 1.0. If T is the global time (frame number), then t = (T - T1)/(T2 - T1), where T1 is the frame number of the starting point and T2 is the frame number of the end point. T runs from T1 to T2. When it reaches T2, the end point becomes your new starting point and the next point in the track your new end point. This is equivalent to moving the ruler on the paper to the next two points. This way of interpolating does not give smooth paths, since the tangent vector of the path (the moving direction) changes suddenly when you pass a joint point. That's why splines are used instead.

The Hermite Spline

The Hermite spline is a curve between two points (P1 and P2) with given tangent vectors R1 and R2 at these points. If your time parameter t is running again from 0.0 to 1.0, you calculate a point on the spline like this:

$$Q(t) = P1*(2t^3-3t^2+1) + R1*(t^3-2t^2+t) + P2*(-2t^3+3t^2) + R2*(t^3-t^2).$$

To use this formula to interpolate your track, just look at four consecutive points in your track. Assign the second point to P1, the third to P2. The first and the last of the four points are used to calculate the tangent vectors R1 and R2. The parameters tension, bias and continuity control the direction and length of R1 and R2. But at first we neglect these parameters and take a position in the middle of the track, because at the start and end point of the track there is one point missing to calculate R1 and R2. pn is the point in the track whose tangent vector tn you want to find. Obviously you have to apply the following formula twice, once for P1 to find R1 and once for P2 to find R2:

$$tn = (p(n+1) - p(n-1))*0.5.$$

Parameters

As I mentioned before, the tangent vector tn of a point pn depends not only on the previous and next point, but also on the parameters which belong to the key kn, which contains pn. To simplify the formulas I introduce three vectors:

$$g1 = pn - p(n-1),$$
$$g2 = p(n+1) - pn,$$
$$g3 = g2 - g1.$$

Then a similar formula for tn is

$$tn = g1 + g3*0.5.$$

Draw the points p(n-1), pn and p(n+1) on a paper in a triangular form.

Attach straight line g1 at pn, that it points away from p(n-1). Then draw a straight line between the end point of g1 and p(n+1) to find g3 and draw a straight line between pn and the middle of g3 to find tn.

This is the drawing. Points of the track are marked with *, auxiliary points with o. Look at it in 50 lines text mode.

```
              o
             /|
          g1/ |
           /  |g3
          /   |
         /    |
      pn*---->|
         \  tn|
          \   |
         g2\  |
            \ |
             \|
  *           *
  p(n-1)      p(n+1)
```

It should help you to understand the influence of the parameters on the tangent vector tn. All parameters range from -1.0 to 1.0. The tension (T) parameter just scales the length of tn. The continuity (C) parameter moves the end point of tn on g3. The bias (B) parameter scales the lengths of g1 and g2. Our new formulas are:

```
g1 = (pn - p(n-1))*(1+B),
g2 = (p(n+1) - pn)*(1-B),
g3 = g2 - g1,
tangent case a: an = (g1 + g3*(0.5 + 0.5*C))*(1-T),
tangent case b: bn = (g1 + g3*(0.5 - 0.5*C))*(1-T).
```

For the incoming spline case a is valid, for the outgoing case b.

Start and end points

====================

The tangent vectors of the start and end points have to be calculated differently. For the start pont it is:

```
b0 = ((p1 - p0)*1.5 - a1*0.5)*(1-T),
```

where a1 is the tangent (case a) of p1, which depends on the parameters of the key k1. For the end point the formula is similar:

```
an = ((pn - p(n-1))*1.5 - b(n-1)*0.5)*(1-T).
```

The remaining case is if the track has only two keys.

Two key tracks

==============
If the track has only two keys, then the two tangents tn2 and tn1 for the only two points p0 and p1 are

```
b0 = (p1-p0)*(1-T),
a1 = (p1-p0)*(1-T).
```

The tension parameter is taken from key k0 to calculate b0 and from k1 to calculate a1.

Source example
==============
The following source comes from my 3DS file converter. It calculates the tangent vector for a given key number. This way I could handle all special cases in C++, while the assembler code in the intro only has to calculate the Hermite spline formula. I defined a n-dimensional vector class (tvec) with a copy constructor and the operators +, - and *. The key class (tkey) contains the following fields:

```
  int frame;
  float tension, continuity, bias;
  tclass * d;
```

In this case the d (data) field points to a vector. The variable called track is a collection (a class replacing an array) with the sub-field count to indicate how many objects (in this case keys) are in the collection. kn_1, kn and kn1 are pointers to a key. pn_1, pn and pn1 are pointers to a vector. gett(an,5) returns a5, gett(bn,0) returns b0.

```
enum {an,bn,point};

tvec ttrack::getv(int sel, int n) {
  tkey* kn_1,* kn,* kn1;
  tvec* pn_1,* pn,* pn1;

  kn = (tkey *) track[n];
  pn = (tvec *) kn->d;

  if (sel == point) return *pn;

  if (n == 0) {
    //first key
    kn1 = (tkey *) track[1];
    pn1 = (tvec *) kn1->d;

    if (track.count == 2) {
      //2 keys
      return (*pn1 - *pn)*(1.0 - kn->tension);
    };
    if (mode != 3) {
      //first key, no loop
      return ((*pn1 - *pn)*1.5 - getv(an,1)*0.5)*(1.0 - kn->tension);
    } else {
      //first key, loop
      kn_1= (tkey *) track[track.count-2];
    };
  } else if (n == track.count-1) {
    //last key
    kn_1 = (tkey *) track[n-1];
    pn_1 = (tvec *) kn_1->d;

    if (track.count == 2) {
      //2 keys
      return (*pn - *pn_1)*(1.0 - kn->tension);
    };
    if (mode != 3) {
      //last key, no loop
      return ((*pn - *pn_1)*1.5 - getv(bn,n-1)*0.5)*(1.0 - kn->tension);
    } else {
      //last key, loop
      kn1 = (tkey *) track[1];
    };
  } else {
    //middle keys
    kn_1= (tkey *) track[n-1];
    kn1 = (tkey *) track[n+1];
  };
  pn_1= (tvec *) kn_1->d;
  pn1 = (tvec *) kn1->d;

  float f = (sel == an) ? 0.5 : -0.5;

  tvec g1 = (*pn  - *pn_1)*(1.0 + kn->bias);
  tvec g2 = (*pn1 - *pn  )*(1.0 - kn->bias);

  return (g1 + (g2-g1)*(0.5 + f*kn->continuity))*(1.0 - kn->tension);
};
```

Rotation track
==============
Now we come to the rotation track, which is more difficult to understand and to code than the splines. In general, you have to perform some steps to come from a rotation track to the interpolated rotation matrix. The rotations in the rotation track are stored as relative rotations from one key to the next with a rotation angle and a rotation axis. You have to convert them to a quaternion and multiply each quaternion with the previous one to get an absolute rotation information. Then you interpolate the quaternions by the method described later. At the end you convert the interpolated rotation to a matrix.

Quaternions
===========
A quaternion is a set of four numbers. For some operations it is best to

treat them as a 4D-vector. The first is the dot product to determine the length. All quaternions which represent a rotation have the length 1, this means they lie on the four dimensional unit sphere. The second is an interpolation formula to find the shortest way from one quaternion to another on the surface of the unit sphere. For other operations treat them as a scalar and a 3D-vector. This is needed for the quaternion multiplication and conversion from angle and vector. The scalar part is called s and the vector part v. Therefore we have the following representations:

```
q = [x1,x2,x3,x4] = [s,v] = [w,(x,y,z)].
```

Conversion angle/axis to quaternion

The first step you have to do is to convert the angle and axis to a quaternion. The formula is

```
s = cos(angle/2.0),
v = axis * sin(angle/2.0).
```

Now you have to convert your rotation track from relative to absolute representation. Just multiply the quaternions, beginning with the second, like this:

```
Qn' = Q(n-1) x Qn,
```

where Q(n-1) is already absolute and 'x' is the quaternion multiplication

```
q1 x q2 = [s1,v1] x [s2,v2] = [(s1*s2 - v1*v2),(s1*v2 + s2*v1 + v1xv2)].
```

The 'v1*v2' term is the dot product of v1 and v2, the 'v1xv2' term is the vector product. The first quaternion does not need to be converted since it is the absolute initial rotation. Now you have a track consisting of quaternions representing absolute rotations.

Linear interpolation

Remember that the simplest way to interpolate a two dimensional position track was done by drawing straight lines between the points. The formula was:

```
Q(t) = P1*(t) + P2*(1-t).
```

Now we need the same thing for rotations. If you treat quaternions as 4D-vectors, all possible rotations lie on the four dimensional unit sphere. To come from one point to another you have to follow an arc on the surface of the sphere. This arc is the curve where the sphere intersects a plane through the two points and the origin. This is the formula for the Spherical Linear intERPolation (slerp):

```
slerp(q1,q2) = sin((1-t)*a)/sin(a) * q1  +  sin(t*a)/sin(a) * q2,
```

where t is the local time parameter running from 0.0 to 1.0 and a is the angle between q1 and q2. The angle is calculated with the standard dot product formula

```
cos(a) = q1 * q2,
```

where q1 and q2 have the length 1. If the angle is too small, you can use a normal linear interpolation to avoid a division by zero or wrong results. The slerp function is very important, so you should understand these examples:

```
slerp(q1,q2, 0.0) = q1,
slerp(q1,q2, 0.5) = a point half way from q1 to q2,
slerp(q1,q2, 1.0) = q2,
slerp(q1,q2,-1.0) = a point opposite to q2 relative to q1.
```

Splines on the sphere

Now we try to transfer all formulas from the spline section to this one. With the slerp function we already can do the equivalent to taking a pencil and a ruler and draw lines between points. But this is actually all we can do on the surface of the four dimensional sphere. Luckily Bezier found a method to draw splines only with a pencil and a ruler. The slerp function acts as our ruler. How to do this we see later, first we look at the formula of the Bezier spline. It depends on four points P1, P2 and C1, C2:

```
Q(t) = P1*(1-t)^3 + C1*3t*(1-t)^2 + C2*3t^2*(1-t) + P2*t^3.
```

It passes only through P1 and P2, C1 and C2 are control points. When we compare it with the Hermite spline, we can find the following relationship:

```
P1h = P1b                 => P1b = P1h
R1h = (C1b-P1b)*3         => C1b = P1h+R1h/3
R2h = (P2b-C2b)*3         => C2b = P2h-R2h/3
P2h = P2b                 => P2b = P2h
```

Since we can only construct Bezier splines on the surface of the sphere we have to convert the Hermite spline to a Bezier spline. Remember the straight lines g1, g2, g3 and the tangent tn. They were

```
g1 = pn - p(n-1),
g2 = p(n+1) - pn,
g3 = g2 - g1,
tn = g1 + g3*0.5.
```

These were relative quantities. On the sphere everything has to be absolute. Our new formulas become

```
g1 = slerp(qn,q(n-1),-1.0/3.0),
g2 = slerp(qn,q(n+1), 1.0/3.0),
g3 = slerp(g1,g2, 0.5),
control point case a: an = slerp(qn,g3,-1.0),
control point case b: bn = slerp(qn,g3, 1.0) = g3.
```

The new formula for g1 has a negative parameter, because the old g1 pointed away from p(n-1).The new g3 corresponds to pn + tn/3, an is the control point C2 for the incoming Bezier spline, bn is C1 for the outgoing one. bn is opposite to an. You should be able to construct an and bn on a paper now.

This is the drawing. Given points are marked with *, auxiliary points with o. Lines show the slerp functions running from the start point to the resulting point.

```
           g1
           o
          /|
         / |
  an o--*  o g3=bn
      qn\
         \
          o
          g2




*               *
q(n-1)          q(n+1)
```

Parameters
==========

Now we introduce the parameters tension (T), continuity (C) and bias (B) again. The formulas are:

```
g1 = slerp(pn,q(n-1),-(1+B)/3.0),
g2 = slerp(pn,q(n+1), (1-B)/3.0),

control point case a:
g3 = slerp(g1,g2,0.5 + 0.5*C),
an = slerp(pn,g3, (T-1)),

control point case b:
g3 = slerp(g1,g2,0.5 - 0.5*C),
bn = slerp(pn,g3,-(T-1)).
```

For the incoming spline case a is valid, for the outgoing case b.

Start and end points
====================

Now we want to calculate the control points an and bn at the start and end points. Here you have to decide if you want a version which is the equivalent to the other tracks or one which is compatible with Autodesk 3D Studio 4.0. First the equivalent version. The formula was

```
b0 = ((p1 - p0)*1.5 - a1*0.5)*(1-T),
```

for the tangent vector of the start point, which needs the tangent (case a) of the next point t(n+1). The formula for the first control point b0 of the quaternion curve is

```
b0 = slerp(qn,a1,(1-T)*0.5),
```

which I found graphically. It also needs the control point a1 which depends on the parameters of k1. The formula for the last control point is similar:

```
an = slerp(qn,b(n-1),(1-T)*0.5).
```

After some time comparing values I found that 3DS uses the following formulas:

```
b0 = slerp(q0,q1    ,(1-T)*(1+C*B)/3.0),
an = slerp(qn,q(n-1),(1-T)*(1-C*B)/3.0).
```

Two key tracks
==============
My version for the formulas:

```
b0 = slerp(q0,q1,(1-T)/3.0),
a1 = slerp(q1,q0,(1-T)/3.0).
```

The 3DS version is like the start and end points for tracks with more keys:

```
b0 = slerp(q0,q1,(1-T)*(1+C*B)/3.0),
a1 = slerp(q1,q0,(1-T)*(1-C*B)/3.0).
```

Source example
==============
This time the data field in the key class holds a pointer to a quaternion. qn_1, qn and qn1 are pointers to a quaternion. The quaternion class has a slerp-method. This version is still not 100% compatible, small differences occur when the bias parameters are other than 0. But I think there is no real bug in it. The commented code lines are my versions of the start and end point calculatoins.

```
tquaternion ttrack::getq(int sel, int n) {
  tkey* kn_1,* kn,* kn1;
  tquaternion* qn_1,* qn,* qn1;

  kn = (tkey *) track[n];
  qn = (tquaternion *) kn->d;
  if (sel == point) return *qn;

  if (n == 0) {
    //first key
    kn1 = (tkey *) track[1];
    qn1 = (tquaternion *) kn1->d;

    //2 keys
//  if (track.count == 2) return qn->slerp(*qn1,(1.0 - kn->tension)/3.0);

    if (mode != 3 || track.count <= 2) {
      //first key, no loop
//    return qn->slerp(getq(an,1),(1.0 - kn->tension)*0.5);
      return qn->slerp(*qn1,(1.0 - kn->tension)*(1.0 + kn->continuity*kn-
>bias)/3.0);
    } else {
      //first key, loop
      kn_1= (tkey *) track[track.count-2];
    };
  } else if (n == track.count-1) {
    //last key
    kn_1 = (tkey *) track[n-1];
    qn_1 = (tquaternion *) kn_1->d;

    //2 keys
//  if (track.count == 2) return qn->slerp(*qn_1,(1.0 - kn->tension)/3.0);

    if (mode != 3 || track.count <= 2) {
      //last key, no loop
//    return qn->slerp(getq(bn,n-1),(1.0 - kn->tension)*0.5);
      return qn->slerp(*qn_1,(1.0 - kn->tension)*(1.0 - kn->continuity*kn-
>bias)/3.0);
    } else {
      //last key, loop
      kn1 = (tkey *) track[1];
    };
  } else {
    //middle keys
    kn_1= (tkey *) track[n-1];
    kn1 = (tkey *) track[n+1];
  };
  qn_1= (tquaternion *) kn_1->d;
  qn1 = (tquaternion *) kn1->d;

  float f = (sel == an) ? 1.0 : -1.0;

  tquaternion g1 = qn->slerp(*qn_1,-(1.0 + kn->bias)/3.0);
  tquaternion g2 = qn->slerp(*qn1 , (1.0 - kn->bias)/3.0);

  return qn->slerp(g1.slerp(g2,0.5 + f*0.5*kn->continuity),f*(kn->tension -
1.0));
```

```
};
```

Interpolation

=============

This time having the control points does not mean we have everything we need. We finally have to draw a spline on the surface of the four dimensional sphere. This can be done using the slerp function again, because you can construct Bezier splines with a pencil and a ruler. We just need some temporary quaternions to do this:

```
q0 = slerp(qn,bn,t),
q1 = slerp(bn,a(n+1),t),
q2 = slerp(a(n+1),q(n+1),t),

q0 = slerp(q0,q1,t),
q1 = slerp(q1,q2,t),

q0 = slerp(q0,q1,t).
```

Read in [2] more about this. q0 is now your final quaternion. Now you have to convert it to a matrix. If q0 = [w,(x,y,z)] then the rotation matrix M is:

```
     ( 1-2yy-2zz        2xy+2wz          2xz-2wy  )
M =  (  2xy-2wz        1-2xx-2zz         2yz+2wx  )
     (  2xz+2wy         2yz-2wx         1-2xx-2yy )
```

Debugging

=========

Since you can't see if everything works fine, you should convert the interpolated quaternions back to a angle/axis representation to compare them with 3DS. If you have defined one key at frame 0 and the next at frame 10, then select key 5 and show the key info of the object. 3DS will then show the interpolated values for frame 5. If you interpolate qn and q(n+1), the resulting quaternion is q0. To convert it back, simply calculate

```
q = qn^(-1) x q0,
```

where 'x' is the quaternion multiplication and qn^(-1) is the inverse of qn. If qn = [s,v], then

```
qn^(-1) = [s,-v].
```

This is the simplified formula for unit quaternions. Now convert the quaternion q to angle/axis:

```
angle = 2.0*arccos(s),
axis = v/sqrt(1-s^2) = v/sin(angle/2.0).
```

You have to avoid divisions by zero of course.

Conclusion

A complete track system is not very easy to implement. I still have not
found a formula for the ease to and ease from parameters yet. If you know
them or have found any errors in this text, please mail me.