Key Frame Interpolation via Splines and Quaternions

David Eberly
Geometric Tools, LLC
http://www.geometrictools.com/

Created: March 2, 1999
Last Modified: February 9, 2008

Contents

1 Introduction

This document illustrates how to interpolate key frames using Kochanek-Bartels splines and quaternion calculus. You should read the documents *Kochanek-Bartels Cubic Splines* (`KBSplines.pdf`) and *Quaternion Algebra and Calculus* (`Quaternions.pdf`) first. A computer implementation is provided for key frame interpolation where both position and orientation are interpolated via splines. A simple application to computer animation is also provided.

2 Squad using KB Splines

The *squad* construction for quaternions can be modified to support the ideas of Kochanek-Bartels splines. While those splines were defined in terms of positional quantities (an additive system), they are easily extended to quaternions (a multiplicative system). In this context, the splines are of the Catmull-Rom type where the derivatives at the control points are centered finite differences,

$$T_n = \frac{\log(q_n^{-1}q_{n+1}) + \log(q_{n-1}^{-1}q_n)}{2}. \tag{1}$$

For tension τ, continuity γ, and bias β, and for a single spline segment between q_n and q_{n+1}, the tangent at $t = 0$ (the "outgoing" tangent) is

$$T_n^0 = \frac{(1-\tau)(1-\gamma)(1-\beta)}{2}\log(q_n^{-1}q_{n+1}) + \frac{(1-\tau)(1+\gamma)(1+\beta)}{2}\log(q_{n-1}^{-1}q_n) \tag{2}$$

and the tangent at $t = 1$ (the "incoming" tangent) is

$$T_n^1 = \frac{(1-\tau)(1+\gamma)(1-\beta)}{2}\log(q_n^{-1}q_{n+1}) + \frac{(1-\tau)(1-\gamma)(1+\beta)}{2}\log(q_{n-1}^{-1}q_n). \tag{3}$$

The intermediate terms for a_n and b_n are constructed as earlier. The equation $T_n^0 = \log(q_n^{-1}q_{n+1}) + 2\log(q_n^{-1}a_n)$ leads to

$$a_n = q_n \exp\left(\frac{T_n^0 - \log(q_n^{-1}q_{n+1})}{2}\right) \tag{4}$$

and $T_n^1 = \log(q_{n-1}^{-1}q_n) - 2\log(q_n^{-1}b_n)$ leads to

$$b_n = q_n \exp\left(\frac{\log(q_{n-1}^{-1}q_n) - T_n^1}{2}\right). \tag{5}$$

Adjustments can be made to the tangents in Equations (2) and (3)) to take into account nonuniform sample times, just as in the interpolation of positional data. The multiplier of Equation (2) is $2\Delta_{n-1}/(\Delta_{n-1}+\Delta_n)$ and the multiplier of equation (3) is $2\Delta_n/(\Delta_{n-1}+\Delta_n)$ where $\Delta_n = s_{n+1} - s_n$ and s_n is the time value of the sample corresponding to q_n.

3 Implementation

The files referenced in this section are part of Wild Magic 2.x, which is downloadable from the Geometric Tools web site. They no longer exist in later versions of Wild Magic. For convenience, source code listings are at the end of this document.

3.1 Position Splines

The files `PosSpline.h` and `PosSpline.cpp` are straightforward implementations of the Kochanek-Bartels splines. The public interface is An application creates position key values and assigns tension, continuity, and bias per key. A sequence of interpolating polynomials and other data required by the interpolation are constructed from the keys. It is assumed that the key times are ordered, `key[i].t < key[i+1].t`, for all i. Since the interpolation requires boundary conditions, an application should choose `key[0]` and `key[numKeys-1]` appropriately. A reasonable choice is to set `key[0] = key[1]` and `key[numKeys-1] = key[numKeys-2]`.

A sample WinMain driver is provided in the source file. There are eight control points, each of the middle six which can be selected with the mouse. The tension, continuity, and bias parameters for the selected control point are displayed in the upper left corner of the window. The keys 't' and 'T' decrease and increase tension, respectively. The keys 'c' and 'C' control continuity and the keys 'b' and 'B' control bias. After each selection, a new spline is built and the spline is drawn.

3.2 Rotation Splines

The files CodeRotSpline.h and `RotSpline.cpp` implement rotations in axis-angle format and use squad with the Kochanek-Bartels extension. The interface is identical to that for position splines, except that the basic element of interpolation is a rotation which is represented by an axis of rotation and an angle of rotation about that axis. Use of the functions is the same as in the position case.

A sample WinMain driver is provided in the source file. There are eight control points, each of the middle six which can be selected with the mouse. The tension, continuity, and bias parameters for the selected control point are displayed in the upper left corner of the window. The keys 't' and 'T' decrease and increase tension, respectively. The keys 'c' and 'C' control continuity and the keys 'b' and 'B' control bias. After each selection, a new spline is built and the spline is drawn.

The input quaternions have $z = 0$, so they lie on the 3D sphere which is the intersection of the 4D unit hypersphere and a hyperplane. The control points are drawn on the 2D screen by projecting out the w component (the remaining components are x and y). The quaternion spline curve is a curve on the 4D unit hypersphere. The curve that is drawn is a projection of the original by setting the w and z. It is not the case that the z components of the squad interpolation are 0. Moreover, the projection into 3D is not necessarily a curve on the 3D unit sphere. The projection into 2D screen space is just to give you an idea of the behavior of the interpolated curve in higher dimensions.

4 Application

A simple WinMain application to animation is provided in `KeyframeAnimation.cpp`. A stick figure representing a leg (hip joint, thigh, knee joint, calf, ankle joint, foot) is animated from a sequence of five key frames (first and last being the same frame, boundary conditions using repeated values at the end points). The thigh, calf, and foot are assumed to be fixed lengths. The hip location, hip angle (measured from the vertical axis), knee angle, and ankle angle are freely chosen parameters. The knee, ankle, and foot positions are determined by inverse kinematics.

More precisely, the free parameters are chosen with respect to the left-handed screen coordinates x and y. All z values are set to zero and all rotations are about the axis $(0, 0, 1)$. Let $\mathbf{H}$, $\mathbf{K}$, $\mathbf{A}$, and $\mathbf{F}$ be the hip, knee, ankle, and foot positions, respectively. Let θ be the angle formed by the leg $\mathbf{K} - \mathbf{H}$ with the y–axis (which points downwards). Let ϕ be the angle between $\mathbf{H} - \mathbf{K}$ and $\mathbf{A} - \mathbf{K}$. Let ψ be the angle between $\mathbf{K} - \mathbf{A}$ and $\mathbf{F} - \mathbf{A}$. Let $L_{hk} = |\mathbf{H} - \mathbf{K}|$, $L_{ka} = |\mathbf{K} - \mathbf{A}|$, and $L_{af} = |\mathbf{A} - \mathbf{F}|$ be the specified lengths of the thigh, calf, and foot, respectively. Some trigonometry will show that

$$\begin{aligned} \mathbf{K} &= \mathbf{H} + L_{hk}(\sin\theta, \cos\theta) \\ \mathbf{A} &= \mathbf{K} + (L_{ka}/L_{hk})R_{\phi}(\mathbf{H} - \mathbf{K}) \\ \mathbf{F} &= \mathbf{A} + (L_{af}/L_{ka})R_{\psi}(\mathbf{K} - \mathbf{A}) \end{aligned} \tag{6}$$

The angles ϕ and ψ are assumed to be acute. The rotation matrix R_{α} has first row $(\cos\alpha, \sin\alpha)$ and second row $(-\sin\alpha, \cos\alpha)$.

The application uses a position spline for $\mathbf{H}$ and rotation splines for θ, ϕ, and ψ. The key frames are initially drawn, but can be toggled py pressing 'k' and 'K'. Single-stepping through the animation is done by pressing 'a' and 'A'. The correction of the interpolated angles is used since the interpolated axis may be $(0, 0, -1)$. For applications which convert the interpolated axis and angle to a rotation matrix for use in a hierarchical transformation scheme, the flipped axis is irrelevant since the angle has a sign change at the same time.

5 Source Code

5.1 PosKey.h

```
#ifndef POSKEY_H
#define POSKEY_H

#include "WmlVector3.h"
using namespace Wml;

class PosKey
{
public:
    PosKey ();

    float& Time ();
    Vector3f& P ();
    float& Tension ();
```

```
    float& Continuity ();
    float& Bias ();

private:
    float m_fTime;
    Vector3f m_kP;

    // parameters are each in [-1,1]
    float m_fTension, m_fContinuity, m_fBias;
};

//----------------------------------------------------------------------------
inline float& PosKey::Time ()
{
    return m_fTime;
}
//----------------------------------------------------------------------------
inline Vector3f& PosKey::P ()
{
    return m_kP;
}
//----------------------------------------------------------------------------
inline float& PosKey::Tension ()
{
    return m_fTension;
}
//----------------------------------------------------------------------------
inline float& PosKey::Continuity ()
{
    return m_fContinuity;
}
//----------------------------------------------------------------------------
inline float& PosKey::Bias ()
{
    return m_fBias;
}
//----------------------------------------------------------------------------

#endif
```

5.2 PosKey.cpp

```
#include "PosKey.h"

//----------------------------------------------------------------------------
PosKey::PosKey ()
    :
    m_kP(Vector3f::ZERO)
{
    m_fTime = 0.0f;
    m_fTension = 0.0f;
```

```
    m_fContinuity = 0.0f;
    m_fBias = 0.0f;
}
//----------------------------------------------------------------------------
```

5.3 PosSpline.h

```
#ifndef POSSPLINE_H
#define POSSPLINE_H

// Kochanek-Bartels tension-continuity-bias spline interpolation for
// positional data.

#include "PosKey.h"

class PosSpline
{
public:
    // The keys should be sorted by increasing time.
    PosSpline (int iNumKeys, PosKey* akKey);
    ~PosSpline ();

    // The interpolators clamp the time to the range of times in the N input
    // keys, [key[0].time,key[N-1].time].
    Vector3f Position (float fTime);      // X(t)
    Vector3f Velocity (float fTime);      // X'(t)
    Vector3f Acceleration (float fTime);  // X"(t)

    // length of the spline
    float Length (float fTime);
    float TotalLength ();

    // Evaluate position and derivatives by specifying arc length s along the
    // spline.  If L is the total length of the curve, then 0 <= s <= L is
    // required.
    Vector3f PositionAL (float fS);
    Vector3f VelocityAL (float fS);
    Vector3f AccelerationAL (float fS);

private:
    class Poly
    {
    public:
        Vector3f Position (float fU);      // P(u)
        Vector3f Velocity (float fU);      // P'(u)
        Vector3f Acceleration (float fU);  // P"(u)
        float Speed (float fU);
        float Length (float fU);

        // Time interval on which polynomial is valid, tmin <= t <= tmax.
        // The normalized time is u = (t - tmin)/(tmax - tmin).  The inverse
```

```
        // range 1/(tmax-tmin) is computed once and stored to avoid having to
        // use divisions during interpolation.
        float m_fTMin, m_fTMax, m_fTInvRange;

        // P(u) = C0 + u*C1 + u^2*C2 + u^3*C3,  0 <= u <= 1
        Vector3f m_akC[4];

        // Legendre polynomial degree 5 for numerical integration
        static float ms_afModRoot[5];
        static float ms_afModCoeff[5];
    };

    void DoPolyLookup (float fTime, int& riI, float& rfU);

    int m_iNumPolys;
    Poly* m_akPoly;

    // support for arc length parameterization of spline
    void InvertIntegral (float fS, int& riI, float& rfU);
    float* m_afLength;
    float m_fTotalLength;
};

#endif
```

5.4 PosSpline.cpp

```
#include "PosSpline.h"

// Legendre polynomial information for Gaussian quadrature of speed on domain
// [0,u], 0 <= u <= 1.  The polynomial is degree 5.
float PosSpline::Poly::ms_afModRoot[5] =
{
    // Legendre roots mapped to (root+1)/2
    0.046910077f,
    0.230765345f,
    0.5f,
    0.769234655f,
    0.953089922f
};

float PosSpline::Poly::ms_afModCoeff[5] =
{
    // original coefficients divided by 2
    0.118463442f,
    0.239314335f,
    0.284444444f,
    0.239314335f,
    0.118463442f
};
```

```
//----------------------------------------------------------------------------
PosSpline::PosSpline (int iNumKeys, PosKey* akKey)
{
    assert( iNumKeys >= 4 );

    m_iNumPolys = iNumKeys-3;
    m_akPoly = new Poly[m_iNumPolys];

    for (int i0=0, i1=1, i2=2, i3=3; i0 < m_iNumPolys; i0++, i1++, i2++, i3++)
    {
        Vector3f kDiff10 = akKey[i1].P() - akKey[i0].P();
        Vector3f kDiff21 = akKey[i2].P() - akKey[i1].P();
        Vector3f kDiff32 = akKey[i3].P() - akKey[i2].P();

        // build multipliers at point P[i1]
        float fOmT0 = 1.0f - akKey[i1].Tension();
        float fOmC0 = 1.0f - akKey[i1].Continuity();
        float fOpC0 = 1.0f + akKey[i1].Continuity();
        float fOmB0 = 1.0f - akKey[i1].Bias();
        float fOpB0 = 1.0f + akKey[i1].Bias();
        float fAdj0 = 2.0f*(akKey[i2].Time() - akKey[i1].Time()) /
            (akKey[i2].Time() - akKey[i0].Time());
        float fOut0 = 0.5f*fAdj0*fOmT0*fOpC0*fOpB0;
        float fOut1 = 0.5f*fAdj0*fOmT0*fOmC0*fOmB0;

        // build outgoing tangent at P[i1]
        Vector3f kTOut = fOut1*kDiff21 + fOut0*kDiff10;

        // build multipliers at point P[i2]
        float fOmT1 = 1.0f - akKey[i2].Tension();
        float fOmC1 = 1.0f - akKey[i2].Continuity();
        float fOpC1 = 1.0f + akKey[i2].Continuity();
        float fOmB1 = 1.0f - akKey[i2].Bias();
        float fOpB1 = 1.0f + akKey[i2].Bias();
        float fAdj1 = 2.0f*(akKey[i2].Time() - akKey[i1].Time()) /
            (akKey[i3].Time() - akKey[i1].Time());
        float fIn0 = 0.5f*fAdj1*fOmT1*fOmC1*fOpB1;
        float fIn1 = 0.5f*fAdj1*fOmT1*fOpC1*fOmB1;

        // build incoming tangent at P[i2]
        Vector3f kTIn = fIn1*kDiff32 + fIn0*kDiff21;

        m_akPoly[i0].m_akC[0] = akKey[i1].P();
        m_akPoly[i0].m_akC[1] = kTOut;
        m_akPoly[i0].m_akC[2] = 3.0f*kDiff21 - 2.0f*kTOut - kTIn;
        m_akPoly[i0].m_akC[3] = -2.0f*kDiff21 + kTOut + kTIn;
        m_akPoly[i0].m_fTMin = akKey[i1].Time();
        m_akPoly[i0].m_fTMax = akKey[i2].Time();
        m_akPoly[i0].m_fTInvRange = 1.0f/(akKey[i2].Time()-akKey[i1].Time());
    }

    // compute arc lengths of polynomials and total length of spline
```

```
    m_afLength = new float[m_iNumPolys+1];
    m_afLength[0] = 0.0f;
    for (int i = 0; i < m_iNumPolys; i++)
    {
        // length of current polynomial
        float fPolyLength = m_akPoly[i].Length(1.0f);

        // total length of curve between poly[0] and poly[i+1]
        m_afLength[i+1] = m_afLength[i] + fPolyLength;
    }
    m_fTotalLength = m_afLength[m_iNumPolys];
}
//----------------------------------------------------------------------------
PosSpline::~PosSpline ()
{
    delete[] m_akPoly;
    delete[] m_afLength;
}
//----------------------------------------------------------------------------
void PosSpline::DoPolyLookup (float fTime, int& riI, float& rfU)
{
    // Lookup the polynomial that contains the input time in its domain of
    // evaluation.  Clamp to [tmin,tmax].

    if ( m_akPoly[0].m_fTMin < fTime )
    {
        if ( fTime < m_akPoly[m_iNumPolys-1].m_fTMax )
        {
            for (riI = 0; riI < m_iNumPolys; riI++)
            {
                if ( fTime < m_akPoly[riI].m_fTMax )
                    break;
            }
            rfU = (fTime-m_akPoly[riI].m_fTMin)*m_akPoly[riI].m_fTInvRange;
        }
        else
        {
            riI = m_iNumPolys-1;
            rfU = 1.0f;
        }
    }
    else
    {
        riI = 0;
        rfU = 0.0f;
    }
}
//----------------------------------------------------------------------------
Vector3f PosSpline::Position (float fTime)
{
    int i;
    float fU;
```

```
    DoPolyLookup(fTime,i,fU);
    return m_akPoly[i].Position(fU);
}
//----------------------------------------------------------------------------
Vector3f PosSpline::Velocity (float fTime)
{
    int i;
    float fU;
    DoPolyLookup(fTime,i,fU);
    return m_akPoly[i].Velocity(fU);
}
//----------------------------------------------------------------------------
Vector3f PosSpline::Acceleration (float fTime)
{
    int i;
    float fU;
    DoPolyLookup(fTime,i,fU);
    return m_akPoly[i].Acceleration(fU);
}
//----------------------------------------------------------------------------
float PosSpline::Length (float fTime)
{
    int i;
    float fU;
    DoPolyLookup(fTime,i,fU);
    return m_akPoly[i].Length(fU);
}
//----------------------------------------------------------------------------
float PosSpline::TotalLength ()
{
    return m_fTotalLength;
}
//----------------------------------------------------------------------------
void PosSpline::InvertIntegral (float fS, int& riI, float& rfU)
{
    // clamp s to [0,L] so that t in [tmin,tmax]
    if ( fS <= 0.0f )
    {
        riI = 0;
        rfU = 0.0f;
        return;
    }

    if ( fS >= m_fTotalLength )
    {
        riI = m_iNumPolys-1;
        rfU = 1.0f;
        return;
    }

    // determine which polynomial corresponds to s
    float fDist;
```

```
    for (riI = 0; riI < m_iNumPolys; riI++)
    {
        if ( fS <= m_afLength[riI+1] )
        {
            // distance along segment
            fDist = fS - m_afLength[riI];

            // initial guess for inverting the arc length integral
            rfU = fDist/(m_afLength[riI+1]-m_afLength[riI]);
            break;
        }
    }

    // use Newton's method to invert the arc length integral
    const float fTolerance = 1e-06f;
    const int iMax = 32;
    for (int i = 0; i < iMax; i++)
    {
        float fDiff = m_akPoly[riI].Length(rfU) - fDist;
        if ( Mathf::FAbs(fDiff) <= fTolerance )
            break;

        // assert: speed > 0
        rfU -= fDiff/m_akPoly[riI].Speed(rfU);
    }
}
//----------------------------------------------------------------------------
Vector3f PosSpline::PositionAL (float fS)
{
    int i;
    float fU;
    InvertIntegral(fS,i,fU);
    return m_akPoly[i].Position(fU);
}
//----------------------------------------------------------------------------
Vector3f PosSpline::VelocityAL (float fS)
{
    int i;
    float fU;
    InvertIntegral(fS,i,fU);
    return m_akPoly[i].Velocity(fU);
}
//----------------------------------------------------------------------------
Vector3f PosSpline::AccelerationAL (float fS)
{
    int i;
    float fU;
    InvertIntegral(fS,i,fU);
    return m_akPoly[i].Acceleration(fU);
}
//----------------------------------------------------------------------------
Vector3f PosSpline::Poly::Position (float fU)
```

```
{
    Vector3f kResult = m_akC[0]+fU*(m_akC[1]+fU*(m_akC[2]+fU*m_akC[3]));
    return kResult;
}
//----------------------------------------------------------------------------
Vector3f PosSpline::Poly::Velocity (float fU)
{
    Vector3f kResult = m_akC[1]+fU*(2.0f*m_akC[2]+3.0f*fU*m_akC[3]);
    return kResult;
}
//----------------------------------------------------------------------------
Vector3f PosSpline::Poly::Acceleration (float fU)
{
    Vector3f kResult = 2.0f*m_akC[2]+6.0f*fU*m_akC[3];
    return kResult;
}
//----------------------------------------------------------------------------
float PosSpline::Poly::Speed (float fU)
{
    return Velocity(fU).Length();
}
//----------------------------------------------------------------------------
float PosSpline::Poly::Length (float fU)
{
    // Need to transform domain [0,u] to [-1,1].  If 0 <= x <= u
    // and -1 <= t <= 1, then x = u*(t+1)/2.
    float fResult = 0.0f;
    for (int i = 0; i < 5; i++)
        fResult += ms_afModCoeff[i]*Speed(fU*ms_afModRoot[i]);
    fResult *= fU;

    return fResult;
}
//----------------------------------------------------------------------------
```

5.5 RotKey.h

```
#ifndef ROTKEY_H
#define ROTKEY_H

#include "WmlQuaternion.h"
using namespace Wml;

class RotKey
{
public:
    RotKey ();

    float& Time ();
    Quaternionf& Q ();
    float& Tension ();
```

```
    float& Continuity ();
    float& Bias ();

private:
    float m_fTime;
    Quaternionf m_kQ;

    // parameters are each in [-1,1]
    float m_fTension, m_fContinuity, m_fBias;
};

//----------------------------------------------------------------------------
inline float& RotKey::Time ()
{
    return m_fTime;
}
//----------------------------------------------------------------------------
inline Quaternionf& RotKey::Q ()
{
    return m_kQ;
}
//----------------------------------------------------------------------------
inline float& RotKey::Tension ()
{
    return m_fTension;
}
//----------------------------------------------------------------------------
inline float& RotKey::Continuity ()
{
    return m_fContinuity;
}
//----------------------------------------------------------------------------
inline float& RotKey::Bias ()
{
    return m_fBias;
}
//----------------------------------------------------------------------------

#endif
```

5.6 RotKey.cpp

```
#include "RotKey.h"

//----------------------------------------------------------------------------
RotKey::RotKey ()
    :
    m_kQ(Quaternionf::IDENTITY)
{
    m_fTime = 0.0f;
    m_fTension = 0.0f;
```

```
    m_fContinuity = 0.0f;
    m_fBias = 0.0f;
}
//----------------------------------------------------------------------------
```

5.7 RotSpline.h

```
#ifndef ROTSPLINE_H
#define ROTSPLINE_H

// Kochanek-Bartels tension-continuity-bias spline interpolation adapted
// to quaternion interpolation.

#include "RotKey.h"

class RotSpline
{
public:
    RotSpline (int iNumKeys, RotKey* akKey);
    ~RotSpline ();

    Quaternionf Q (float fTime);

private:
    class SquadPoly
    {
    public:
        Quaternionf Q (float fU);

        // Time interval on which polynomial is valid, tmin <= t <= tmax.
        // The normalized time is u = (t - tmin)/(tmax - tmin).  The inverse
        // range 1/(tmax-tmin) is computed once and stored to avoid having to
        // use divisions during interpolation.
        float m_fTMin, m_fTMax, m_fTInvRange;

        // Q(u) = Squad(2u(1-u),Slerp(u,p,q),Slerp(u,a,b))
        Quaternionf m_kP, m_kA, m_kB, m_kQ;
    };

    int m_iNumPolys;
    SquadPoly* m_akPoly;
};

#endif
```

5.8 RotSpline.cpp

```
#include "RotSpline.h"
```

```
//----------------------------------------------------------------------------
RotSpline::RotSpline (int iNumKeys, RotKey* akKey)
{
    assert( iNumKeys >= 4 );

    m_iNumPolys = iNumKeys-3;
    m_akPoly = new SquadPoly[m_iNumPolys];

    // Consecutive quaterions should form an acute angle.  Changing sign on
    // a quaternion does not change the rotation it represents.
    int i;
    for (i = 1; i < iNumKeys; i++)
    {
        if ( akKey[i].Q().Dot(akKey[i-1].Q()) < 0.0f )
            akKey[i].Q() = -akKey[i].Q();
    }

    for (int i0=0,i1=1,i2=2,i3=3; i0<m_iNumPolys; i0++,i1++,i2++,i3++)
    {
        Quaternionf kQ0 = akKey[i0].Q();
        Quaternionf kQ1 = akKey[i1].Q();
        Quaternionf kQ2 = akKey[i2].Q();
        Quaternionf kQ3 = akKey[i3].Q();

        Quaternionf kLog10 = (kQ0.Conjugate()*kQ1).Log();
        Quaternionf kLog21 = (kQ1.Conjugate()*kQ2).Log();
        Quaternionf kLog32 = (kQ2.Conjugate()*kQ3).Log();

        // build multipliers at q[i1]
        float fOmT0 = 1.0f - akKey[i1].Tension();
        float fOmC0 = 1.0f - akKey[i1].Continuity();
        float fOpC0 = 1.0f + akKey[i1].Continuity();
        float fOmB0 = 1.0f - akKey[i1].Bias();
        float fOpB0 = 1.0f + akKey[i1].Bias();
        float fAdj0 = 2.0f*(akKey[i2].Time() - akKey[i1].Time()) /
            (akKey[i2].Time() - akKey[i0].Time());
        float fOut0 = 0.5f*fAdj0*fOmT0*fOpC0*fOpB0;
        float fOut1 = 0.5f*fAdj0*fOmT0*fOmC0*fOmB0;

        // build outgoing tangent at q[i1]
        Quaternionf kTOut = fOut1*kLog21 + fOut0*kLog10;

        // build multipliers at q[i2]
        float fOmT1 = 1.0f - akKey[i2].Tension();
        float fOmC1 = 1.0f - akKey[i2].Continuity();
        float fOpC1 = 1.0f + akKey[i2].Continuity();
        float fOmB1 = 1.0f - akKey[i2].Bias();
        float fOpB1 = 1.0f + akKey[i2].Bias();
        float fAdj1 = 2.0f*(akKey[i2].Time() - akKey[i1].Time()) /
            (akKey[i3].Time() - akKey[i1].Time());
        float fIn0 = 0.5f*fAdj1*fOmT1*fOmC1*fOpB1;
        float fIn1 = 0.5f*fAdj1*fOmT1*fOpC1*fOmB1;
```

```
        // build incoming tangent at q[i2]
        Quaternionf kTIn = fIn1*kLog32 + fIn0*kLog21;

        m_akPoly[i0].m_kP = kQ1;
        m_akPoly[i0].m_kQ = kQ2;
        m_akPoly[i0].m_kA = kQ1*((0.5f*(kTOut-kLog21)).Exp());
        m_akPoly[i0].m_kB = kQ2*((0.5f*(kLog21-kTIn)).Exp());
        m_akPoly[i0].m_fTMin = akKey[i1].Time();
        m_akPoly[i0].m_fTMax = akKey[i2].Time();
        m_akPoly[i0].m_fTInvRange = 1.0f/(akKey[i2].Time()-akKey[i1].Time());
    }
}
//----------------------------------------------------------------------------
RotSpline::~RotSpline ()
{
    delete[] m_akPoly;
}
//----------------------------------------------------------------------------
Quaternionf RotSpline::Q (float fTime)
{
    // find the interpolating polynomial (clamping used, modify for looping)
    int i;
    float fU;

    if ( m_akPoly[0].m_fTMin < fTime )
    {
        if ( fTime < m_akPoly[m_iNumPolys-1].m_fTMax )
        {
            for (i = 0; i < m_iNumPolys; i++)
            {
                if ( fTime < m_akPoly[i].m_fTMax )
                    break;
            }
            fU = (fTime-m_akPoly[i].m_fTMin)*m_akPoly[i].m_fTInvRange;
        }
        else
        {
            i = m_iNumPolys-1;
            fU = 1.0f;
        }
    }
    else
    {
        i = 0;
        fU = 0.0f;
    }

    return m_akPoly[i].Q(fU);
}
//----------------------------------------------------------------------------
Quaternionf RotSpline::SquadPoly::Q (float fU)
```

```
{
    Quaternionf kSquad = Quaternionf::Squad(fU,m_kP,m_kA,m_kB,m_kQ);
    return kSquad;
}
//----------------------------------------------------------------------------
```

5.9 KeyframeAnimation.h

```
#ifndef KEYFRAMEANIMATION_H
#define KEYFRAMEANIMATION_H

#include "WmlApplication2.h"
#include "PosSpline.h"
#include "RotSpline.h"

class KeyframeAnimation : public Application2
{
public:
    KeyframeAnimation ();
    virtual ~KeyframeAnimation ();

    virtual bool OnInitialize ();
    virtual void OnTerminate ();
    virtual void OnDisplay ();
    virtual void OnKeyDown (unsigned char ucKey, int iX, int iY);

protected:
    class Leg
    {
    public:
        // specified
        Vector3f m_kH;
        float m_fHKLength, m_fKALength, m_fAFLength;
        float m_fHAngle, m_fKAngle, m_fAAngle;

        // derived
        Vector3f m_kK, m_kA, m_kF;
    };

    void InverseKinematics (Leg& rkLeg);

    const int m_iNumKeys;
    const float m_fHKLength, m_fKALength, m_fAFLength;
    const float m_fTMin, m_fTMax, m_fDt;

    Leg* m_akLeg;
    PosSpline* m_pkPosSpline;
    RotSpline* m_pkHRotSpline;
    RotSpline* m_pkKRotSpline;
    RotSpline* m_pkARotSpline;
    float m_fTime;
```

```
    bool m_bDrawKeys;
};

#endif
```

5.10 KeyframeAnimation.cpp

```
#include "KeyframeAnimation.h"

KeyframeAnimation g_kTheApp;

//----------------------------------------------------------------------------
KeyframeAnimation::KeyframeAnimation ()
    :
    Application2("KeyframeAnimation",0,0,256,256,ColorRGB(1.0f,1.0f,1.0f)),
    m_iNumKeys(7),
    m_fHKLength(70.0f),
    m_fKALength(70.0f),
    m_fAFLength(15.0f),
    m_fTMin(0.0f),
    m_fTMax((float)(m_iNumKeys-3)),
    m_fDt(0.01f*(m_fTMax-m_fTMin))
{
    m_pkPosSpline = NULL;
    m_pkHRotSpline = NULL;
    m_pkKRotSpline = NULL;
    m_pkARotSpline = NULL;
    m_fTime = m_fTMin;
    m_bDrawKeys = true;
}
//----------------------------------------------------------------------------
KeyframeAnimation::~KeyframeAnimation ()
{
}
//----------------------------------------------------------------------------
bool KeyframeAnimation::OnInitialize ()
{
    if ( !Application2::OnInitialize() )
        return false;

    // keyframe data
    Vector3f akPos[7] =
    {
        Vector3f(50.0f,34.0f,0.0f),
        Vector3f(50.0f,34.0f,0.0f),
        Vector3f(80.0f,33.0f,0.0f),
        Vector3f(110.0f,33.0f,0.0f),
        Vector3f(140.0f,34.0f,0.0f),
        Vector3f(207.0f,34.0f,0.0f),
        Vector3f(207.0f,34.0f,0.0f)
    };
```

```
float afHAngle[7] =
{
    0.0f, 0.0f, 0.392699f, 0.589049f, 0.490874f, 0.0f, 0.0f
};

float afKAngle[7] =
{
    2.748894f, 2.748894f, 1.963495f, 2.356194f, 2.748894f, 2.748894f,
    2.748894f
};

float afAAngle[7] =
{
    -1.178097f, -1.178097f, -1.178097f, -1.570796f, -1.963495f,
    -1.178097f, -1.178097f
};

PosKey akHPos[7];
RotKey akHRot[7], akKRot[7], akARot[7];
m_akLeg = new Leg[7];

int i;
for (i = 0; i < m_iNumKeys; i++)
{
    float fTime = (float)(i-1);
    akHPos[i].Time() = fTime;
    akHPos[i].P() = akPos[i];
    akHRot[i].Time() = fTime;
    akHRot[i].Q().FromAxisAngle(Vector3f::UNIT_Z,afHAngle[i]);
    akKRot[i].Time() = fTime;
    akKRot[i].Q().FromAxisAngle(Vector3f::UNIT_Z,afKAngle[i]);
    akARot[i].Time() = fTime;
    akARot[i].Q().FromAxisAngle(Vector3f::UNIT_Z,afAAngle[i]);

    m_akLeg[i].m_fHKLength = m_fHKLength;
    m_akLeg[i].m_fKALength = m_fKALength;
    m_akLeg[i].m_fAFLength = m_fAFLength;
    m_akLeg[i].m_kH = akHPos[i].P();
    m_akLeg[i].m_fHAngle = afHAngle[i];
    m_akLeg[i].m_fKAngle = afKAngle[i];
    m_akLeg[i].m_fAAngle = afAAngle[i];
    InverseKinematics(m_akLeg[i]);
}

m_pkPosSpline = new PosSpline(m_iNumKeys,akHPos);
m_pkHRotSpline = new RotSpline(m_iNumKeys,akHRot);
m_pkKRotSpline = new RotSpline(m_iNumKeys,akKRot);
m_pkARotSpline = new RotSpline(m_iNumKeys,akARot);

OnDisplay();
return true;
```

```
}
//----------------------------------------------------------------------------
void KeyframeAnimation::OnTerminate ()
{
    delete[] m_akLeg;
    delete m_pkPosSpline;
    delete m_pkHRotSpline;
    delete m_pkKRotSpline;
    delete m_pkARotSpline;
    Application2::OnTerminate();
}
//----------------------------------------------------------------------------
void KeyframeAnimation::OnDisplay ()
{
    ClearScreen();

    int iX0, iY0, iX1, iY1;

    if ( m_bDrawKeys )
    {
        // draw key frames
        for (int i = 0; i < m_iNumKeys; i++)
        {
            iX0 = (int)m_akLeg[i].m_kH.X();
            iY0 = (int)m_akLeg[i].m_kH.Y();
            iX1 = (int)m_akLeg[i].m_kK.X();
            iY1 = (int)m_akLeg[i].m_kK.Y();
            DrawLine(iX0,iY0,iX1,iY1,Color(0,0,0));

            iX0 = iX1;
            iY0 = iY1;
            iX1 = (int)m_akLeg[i].m_kA.X();
            iY1 = (int)m_akLeg[i].m_kA.Y();
            DrawLine(iX0,iY0,iX1,iY1,Color(0,0,0));

            iX0 = iX1;
            iY0 = iY1;
            iX1 = (int)m_akLeg[i].m_kF.X();
            iY1 = (int)m_akLeg[i].m_kF.Y();
            DrawLine(iX0,iY0,iX1,iY1,Color(0,0,0));
        }
    }

    // interpolate key frames
    Leg kInterp;
    float fAngle;
    Vector3f kAxis;

    kInterp.m_fHKLength = m_fHKLength;
    kInterp.m_fKALength = m_fKALength;
    kInterp.m_fAFLength = m_fAFLength;
    kInterp.m_kH =  m_pkPosSpline->Position(m_fTime);
```

```
    m_pkHRotSpline->Q(m_fTime).ToAxisAngle(kAxis,fAngle);
    kInterp.m_fHAngle = ( kAxis.Z() > 0.0f ? fAngle : -fAngle );

    m_pkKRotSpline->Q(m_fTime).ToAxisAngle(kAxis,fAngle);
    kInterp.m_fKAngle = ( kAxis.Z() > 0.0f ? fAngle : -fAngle );

    m_pkARotSpline->Q(m_fTime).ToAxisAngle(kAxis,fAngle);
    kInterp.m_fAAngle = ( kAxis.Z() > 0.0f ? fAngle : -fAngle );

    InverseKinematics(kInterp);

    // draw interpolated key frame
    iX0 = (int)kInterp.m_kH.X();
    iY0 = (int)kInterp.m_kH.Y();
    iX1 = (int)kInterp.m_kK.X();
    iY1 = (int)kInterp.m_kK.Y();
    DrawLine(iX0,iY0,iX1,iY1,Color(255,0,0));

    iX0 = iX1;
    iY0 = iY1;
    iX1 = (int)kInterp.m_kA.X();
    iY1 = (int)kInterp.m_kA.Y();
    DrawLine(iX0,iY0,iX1,iY1,Color(255,0,0));

    iX0 = iX1;
    iY0 = iY1;
    iX1 = (int)kInterp.m_kF.X();
    iY1 = (int)kInterp.m_kF.Y();
    DrawLine(iX0,iY0,iX1,iY1,Color(255,0,0));

    Application2::OnDisplay();
}
//----------------------------------------------------------------------------
void KeyframeAnimation::OnKeyDown (unsigned char ucKey, int, int)
{
    if ( ucKey == 'q' || ucKey == 'Q' || ucKey == KEY_ESCAPE )
    {
        RequestTermination();
        return;
    }

    switch ( ucKey )
    {
    case 'k':
    case 'K':
        // toggle drawing of key frames
        m_bDrawKeys = !m_bDrawKeys;
        OnDisplay();
        break;
    case 'b':
    case 'B':
```

```
        // single step backwards through animation
        m_fTime -= m_fDt;
        if ( m_fTime < m_fTMin )
            m_fTime = m_fTMax;
        OnDisplay();
        break;
    case 'f':
    case 'F':
        // single step forwards through animation
        m_fTime += m_fDt;
        if ( m_fTime > m_fTMax )
            m_fTime = m_fTMin;
        OnDisplay();
        break;
    }
}
//----------------------------------------------------------------------------
void KeyframeAnimation::InverseKinematics (Leg& rkLeg)
{
    rkLeg.m_kK.X() = rkLeg.m_kH.X() +
        rkLeg.m_fHKLength*Mathf::Sin(rkLeg.m_fHAngle);
    rkLeg.m_kK.Y() = rkLeg.m_kH.Y() +
        rkLeg.m_fHKLength*Mathf::Cos(rkLeg.m_fHAngle);
    rkLeg.m_kK.Z() = 0.0f;

    float fCos = Mathf::Cos(rkLeg.m_fKAngle);
    float fSin = Mathf::Sin(rkLeg.m_fKAngle);
    float fInvLength = 1.0f/rkLeg.m_fHKLength;
    float fDx = (rkLeg.m_kH.X()-rkLeg.m_kK.X())*fInvLength;
    float fDy = (rkLeg.m_kH.Y()-rkLeg.m_kK.Y())*fInvLength;
    rkLeg.m_kA.X() = rkLeg.m_kK.X()+rkLeg.m_fKALength*( fCos*fDx+fSin*fDy);
    rkLeg.m_kA.Y() = rkLeg.m_kK.Y()+rkLeg.m_fKALength*(-fSin*fDx+fCos*fDy);
    rkLeg.m_kA.Z() = 0;

    fCos = Mathf::Cos(rkLeg.m_fAAngle);
    fSin = Mathf::Sin(rkLeg.m_fAAngle);
    fInvLength = 1.0f/rkLeg.m_fKALength;
    fDx = (rkLeg.m_kK.X()-rkLeg.m_kA.X())*fInvLength;
    fDy = (rkLeg.m_kK.Y()-rkLeg.m_kA.Y())*fInvLength;
    rkLeg.m_kF.X() = rkLeg.m_kA.X()+rkLeg.m_fAFLength*( fCos*fDx+fSin*fDy);
    rkLeg.m_kF.Y() = rkLeg.m_kA.Y()+rkLeg.m_fAFLength*(-fSin*fDx+fCos*fDy);
    rkLeg.m_kF.Z() = 0.0f;
}
//----------------------------------------------------------------------------
```