Simple Keyframe Animation

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License

Mike Bailey
mjb@cs.oregonstate.edu
Approaches to Animation

1. Motion Capture ("MoCap")

2. Using the laws of physics (we’ll use this in the spring-based motion)

3. Using functional (target-driven) animation (we’ll use this in collision avoidance)

4. Using keyframing
Keyframing involves creating certain key positions for the objects in the scene, and then the program later interpolating the animation frames in between the key frames.

In hand-drawn animation, the key frames were developed by the senior animators, and the in-between frames were developed by the junior animators.

In our case, you are going to be the senior animator, and the computer will do the in-betweening.

But, first we need to look into the mathematics of smooth curves . . .
Keyframing

But, first we need to look into the mathematics of smooth curves . . .

http://ieeexplore.ieee.org/ieee_pilot/articles/05/ttg2009050853/figures.html
Bézier Curves: the Derivation

One parametric line:

\[ P_{01} = (1-t)P_0 + tP_1 \]
Bézier Curves: the Derivation

Two parametric lines:

\[ P_{01} = (1 - t)P_0 + tP_1 \]
\[ P_{12} = (1 - t)P_1 + tP_2 \]
Bézier Curves: the Derivation

Two parametric lines, blended:

\[ P_{01} = (1-t)P_0 + tP_1 \]
\[ P_{12} = (1-t)P_1 + tP_2 \]

\[ P_{012} = (1-t)P_{01} + tP_{12} \]
\[ = (1-t)^2 P_0 + 2t(1-t)P_1 + t^2 P_2 \]
Bézier Curves: the Derivation

Three parametric lines, blended:

\[
P_{01} = (1-t)P_0 + tP_1 \\
P_{12} = (1-t)P_1 + tP_2 \\
P_{012} = (1-t)P_{01} + tP_{12} \\
= (1-t)^2 P_0 + 2t(1-t)P_1 + t^2 P_2
\]

\[
P_{123} = (1-t)P_{12} + tP_{23} \\
= (1-t)^2 P_1 + 2t(1-t)P_2 + t^2 P_3
\]
Bézier Curves: the Derivation

Three parametric lines, blended:

\[ P_{01} = (1 - t)P_0 + tP_1 \]
\[ P_{12} = (1 - t)P_1 + tP_2 \]

\[ P_{012} = (1 - t)P_{01} + tP_{12} \]
\[ = (1 - t)^2 P_0 + 2t(1 - t)P_1 + t^2 P_2 \]

\[ P_{123} = (1 - t)P_{12} + tP_{23} \]
\[ = (1 - t)^2 P_1 + 2t(1 - t)P_2 + t^2 P_3 \]

\[ P_{0123} = (1 - t)P_{012} + tP_{123} \]
\[ = (1 - t)^3 P_0 + 3t(1 - t)^2 P_1 + 3t^2 (1 - t)P_2 + t^3 P_3 \]
Bézier Curves: Drawing and Sculpting

\[ P(t) = (1 - t)^3 P_0 + 3t(1 - t)^2 P_1 + 3t^2(1 - t)P_2 + t^3 P_3 \]

\[ t = 0.0, 0.02, 0.04, 0.06, \ldots, 0.98, 1.0 \]
The General Form of Cubic Curves

\[ P(t) = A + Bt + Ct^2 + Dt^3 \]

In this form, you need to determine 4 quantities (A, B, C, D) in order to use the equation. That means you have to provide 4 pieces of information. In the Bézier curve, this happens by specifying the 4 points.

\[ P(t) = (1-t)^3 P_0 + 3t(1-t)^2 P_1 + 3t^2 (1-t)P_2 + t^3 P_3 \]

Rearranging gives:

\[
A = P_0 \\
B = -3P_0 + 3P_1 \\
C = 3P_0 - 6P_1 + 3P_2 \\
D = -P_0 + 3P_1 - 3P_2 + P_3
\]
Coons (also called Hermite) Cubic Curves

Another approach to specifying the 4 pieces of information would be to give a start point, an end point, a start parametric slope, and an end parametric slope.

If we do this, then the equation of the curve is:

\[ P = A + Bt + Ct^2 + Dt^3 \]

where:

\[ A = P_0 \]
\[ B = \dot{P}_0 \]
\[ C = -3P_0 + 3P_1 - 2\dot{P}_0 - \ddot{P}_1 \]
\[ D = 2P_0 - 2P_1 + \dot{P}_0 + \ddot{P}_1 \]
Now, Let’s Apply this to the Y Translation of a Keyframe Animation

To make this simple to use, our goal is to just specify the keyframe values, not the slopes. We will let the computer compute the slopes for us, which will result in being able to compute the in-between frames.
Many Professional Animation Packages Make You Sculpt the Slopes

Blender:
The “Y vs. Frame” Curve Looks Like This
Computing the End Slopes for the Y Translation

\[
\frac{dY}{dF_0} = \frac{Y_1 - Y_{-1}}{F_1 - F_{-1}}
\]
\[
\frac{dY}{dF_1} = \frac{Y_2 - Y_0}{F_2 - F_0}
\]
\[
\frac{dF}{dt}_{0-1} = \frac{F_1 - F_0}{1.0.} = F_1 - F_0
\]

\[
\dot{Y}_0 = \frac{dY}{dt}_0 = \frac{dY}{dF_0} \frac{dF}{dt}_{0-1}
\]
\[
\dot{Y}_1 = \frac{dY}{dt}_1 = \frac{dY}{dF_1} \frac{dF}{dt}_{0-1}
\]

\[A = Y_0\]
\[B = \dot{Y}_0\]
\[C = -3Y_0 + 3Y_1 - 2\dot{Y}_0 - \ddot{Y}_1\]
\[D = 2Y_0 - 2Y_1 + \dot{Y}_0 + \ddot{Y}_1\]

\[Y = A + Bt + Ct^2 + Dt^3\]
Getting the Two End Slopes

\[ \frac{dY}{dt \_i} = \frac{dY}{dF} \frac{dF}{dt \_i \rightarrow i+1} \]

\[ \frac{dY}{dt \_i+1} = \frac{dY}{dF \_i+1} \frac{dF}{dt \_i \rightarrow i+1} \]
Getting the Two End Slopes

\[
\frac{dY}{dF_i} = \frac{Y_{i+1} - Y_{i-1}}{F_{i+1} - F_{i-1}} = \frac{Y_{i+2} - Y_i}{F_{i+2} - F_i}
\]

\[
\frac{dF}{dt}_{i \rightarrow i+1} = \frac{F_{i+1} - F_i}{1.0.}
\]
Getting the Two End Slopes

\[
\frac{dY}{dF_i} = \frac{Y_{i+1} - Y_{i-1}}{F_{i+1} - F_{i-1}} = \frac{650 - 650}{450 - 200} = \frac{650 - 400}{525 - 300}
\]

\[
\frac{dF}{dt}_{i \rightarrow i+1} = \frac{F_{i+1} - F_i}{1.0} = F_{i+1} - F_i = 450 - 300 = 150
\]
Getting the Two End Slopes

\[
\frac{dY}{dt}_{i} = \frac{dY}{dF_i} \frac{dF}{dt}_{i \rightarrow i+1} = \frac{0}{250} \times 150 = 0
\]

\[
\frac{dY}{dt}_{i+1} = \frac{dY}{dF_{i+1}} \frac{dF}{dt}_{i \rightarrow i+1} = \frac{250}{225} \times 150 = 166.7
\]
Do This Same Thing for the X, Y, and Z Translations and the X, Y, and Z Rotations
Using the System Clock in Animate( ) for Timing

if( AnimationIsOn )
{
    // # msec into the cycle ( 0 - MSEC-1 ):
    int msec = glutGet( GLUT_ELAPSED_TIME ) % MSEC;

    // turn that into the current frame number:
    NowFrame = (int) ( (float)MAXFRAME * (float)msec / (float)MSEC );

    // look through the keyframes and figure out which two keyframes this is between:
    for( int i = 0; i < maxKeyframes - 1; i++ )
    {
        if( ????? )
        {
            KeyFrameBefore = ?????;
            KeyFrameAfter = ?????;
            break;
        }
    }

    // get the t (0.-1.) for frame NowFrame in the interval between the i
    // th keyframe and the (i+1)st ketframe:
    NowT = ?????

Using the System Clock in Animate( ) for Timing

// determine the A, B, C, and D for all the interpolation curves in that interval:

Ax, Bx, Cx, Dx = ?????
Ay, By, Cy, Dy = ?????
...

// use the coefficients and t to compute the current transformation (and other) parameters:

NowX = Ax + Bx*NowT + Cx*NowT*NowT + Dx*NowT*NowT*NowT;
NowY = Ay + By*NowT + Cy*NowT*NowT + Dy*NowT*NowT*NowT;
...

// post a redisplay to use those parameters:

}

In Display( ):

glPushMatrix( );
    glTranslatef( NowX, NowY, NowZ );
glRotatef( NowRx, 1., 0., 0. );
glRotatef( NowRy, 0., 1., 0. );
glRotatef( NowRz, 0., 0., 1. );
    << Draw the moving object >>

glPopMatrix( );
A Final Word

If you ever do this “for real”, *quaternions* are a better way to do the rotations.

Quaternions are essentially 4D complex numbers, the details of which are beyond the scope of this class. They do a smoother job of rotations because they deal with an angle and an axis of rotation, rather than 3 angles about the principle axes, which is somewhat arbitrary.

See, for example, [http://en.wikipedia.org/wiki/Quaternion](http://en.wikipedia.org/wiki/Quaternion)