Simple Keyframe Animation

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

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.

Bézier Curves: the Derivation

One parametric line:

But, first we need to look into the mathematics of smooth curves . . .
Bézier Curves: the Derivation

Two parametric lines:

\[ P_{1} = (1-t)P_{0} + tP_{1} \]
\[ P_{12} = (1-t)P_{1} + tP_{2} \]

Bézier Curves: the Derivation

Two parametric lines, blended:

\[ P_{01} = (1-t)P_{0} + tP_{1} \]
\[ P_{12} = (1-t)P_{1} + tP_{2} \]
\[ P_{012} = (1-t)^2P_{0} + 2t(1-t)P_{1} + t^2P_{2} \]

Bézier Curves: the Derivation

Three parametric lines, blended:

\[ P_{0} = (1-t)P_{0} + tP_{1} \]
\[ P_{1} = (1-t)P_{1} + tP_{2} \]
\[ P_{01} = (1-t)P_{0} + tP_{1} \]
\[ P_{12} = (1-t)P_{1} + tP_{2} \]
\[ P_{012} = (1-t)^2P_{0} + 2t(1-t)P_{1} + t^2P_{2} \]

Bézier Curves: the Derivation

Three parametric lines, blended:

\[ P_{0} = (1-t)P_{0} + tP_{1} \]
\[ P_{1} = (1-t)P_{1} + tP_{2} \]
\[ P_{01} = (1-t)P_{0} + tP_{1} \]
\[ P_{12} = (1-t)P_{1} + tP_{2} \]
\[ P_{012} = (1-t)^3P_{0} + 3t(1-t)^2P_{1} + 3t^2(1-t)P_{2} + t^3P_{3} \]
Bézier Curves: Drawing and Sculpting

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

\( t = 0,.02,.04,.06,...,.98,1.0 \)

So How Do Smooth Curves Work in Computer Graphics?
A Small Amount of Input Change Results in a Large Amount of Output Change

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.

Rearranging gives A, B, C, and D for the Bézier curve:

\[ A = P_0 \]
\[ B = -3P_0 + 3P_1 \]
\[ C = 3P_0 - 6P_1 + 3P_2 \]
\[ D = -P_0 + 3P_1 - 3P_2 + P_3 \]

Another Approach: 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:

\[ A = P_0 \]
\[ B = P_1 \]
\[ C = -3P_0 + 3P_1 - 2P_0 - \dot{P}_1 \]
\[ D = 2P_0 - 2P_1 + \dot{P}_0 + \dot{P}_1 \]
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 ten result in being able to compute the in-between frames.

Many Professional Animation Packages Make You Sculpt the Slopes (but we won’t . . .)

Blender:

Getting the Two End Slopes

To get the slope at a keyframe point, draw a line between one keyframe back from that one and one keyframe ahead.
Do This Same Thing for the X, Y, and Z Translations and the X, Y, and Z Rotations

Instead of Key Frames, I Like Specifying Key Times Better

And, so, I created a C++ class to do it all for you

class Keytimes:
    void AddTimeValue( float time, float value );
    float GetFirstTime( );
    float GetLastTime( );
    int GetNumKeytimes( );
    float GetValue( float time );
    void PrintTimeValues( );

Instead of Key Frames, I Like Specifying Key Times Better

```c++
Keytimes Xpos;
int main( int argc, char *argv[] )
{
    Xpos.AddTimeValue( 0.0, 0.000 );
    Xpos.AddTimeValue( 2.0, 0.333 );
    Xpos.AddTimeValue( 1.0, 3.142 );
    Xpos.AddTimeValue( 0.5, 2.718 );
    fprintf( stderr, "%d time-value pairs:\n", Xpos.GetNumKeytimes( ) );
    Xpos.PrintTimeValues( );
    fprintf( stderr, "Time runs from %8.3f to %8.3f\n", Xpos.GetFirstTime( ), Xpos.GetLastTime( ) );
    for( float t = 0.; t <= 2.01; t += 0.1 )
    {
        float v = Xpos.GetValue( t );
        fprintf( stderr, "%8.3f\t%8.3f\n", t, v );
    }
}
```

Instead of Key Frames, I Like Specifying Key Times Better

<table>
<thead>
<tr>
<th>Time (s)</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>0.000</td>
</tr>
<tr>
<td>0.10</td>
<td>0.232</td>
</tr>
<tr>
<td>0.20</td>
<td>0.836</td>
</tr>
<tr>
<td>0.30</td>
<td>1.535</td>
</tr>
<tr>
<td>0.40</td>
<td>2.234</td>
</tr>
<tr>
<td>0.50</td>
<td>2.718</td>
</tr>
<tr>
<td>0.60</td>
<td>2.989</td>
</tr>
<tr>
<td>0.70</td>
<td>3.170</td>
</tr>
<tr>
<td>0.80</td>
<td>3.258</td>
</tr>
<tr>
<td>0.90</td>
<td>3.300</td>
</tr>
<tr>
<td>1.00</td>
<td>3.142</td>
</tr>
<tr>
<td>1.10</td>
<td>2.935</td>
</tr>
<tr>
<td>1.20</td>
<td>2.646</td>
</tr>
<tr>
<td>1.30</td>
<td>2.302</td>
</tr>
<tr>
<td>1.40</td>
<td>1.924</td>
</tr>
<tr>
<td>1.50</td>
<td>1.599</td>
</tr>
<tr>
<td>1.60</td>
<td>1.169</td>
</tr>
<tr>
<td>1.70</td>
<td>0.840</td>
</tr>
<tr>
<td>1.80</td>
<td>0.574</td>
</tr>
<tr>
<td>1.90</td>
<td>0.397</td>
</tr>
<tr>
<td>2.00</td>
<td>0.333</td>
</tr>
</tbody>
</table>
Using the System Clock in Animate() for Timing

```c
#if AnimationIsOn
{
  // # msec into the cycle (0 - MSEC-1):
  int msec = glutGet(GLUT_ELAPSED_TIME) % MSEC;
  // turn that into the current time:
  float NowTime = (float)msec / 1000.;
  // get the positions right now:
  NowX = Xpos.GetValue(NowTime);
  NowY = Ypos.GetValue(NowTime);
  NowZ = Zpos.GetValue(NowTime);
  NowRx = Rxpos.GetValue(NowTime);
  NowRy = Rypos.GetValue(NowTime);
  NowRz = Rzpos.GetValue(NowTime);

  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)