Lecture 4: Interpolation and blending

Computer Animation
Reminders

- Do you know your team members for the movie project?
- Did you start to work with your team members for the project idea and storyboard?
- Do you know when you have appointments in the mocap lab?
- Did you check which paper will you present and with whom?
- Did you start to think about your essay topic?
Content

- Cubic curves
- Keyframes & Channels
- Blending and State Machines
Cubic Curves
Polynomial Functions

- **Linear:** \( f(t) = at + b \)
- **Quadratic:** \( f(t) = at^2 + bt + c \)
- **Cubic:** \( f(t) = at^3 + bt^2 + ct + d \)
Beziers Curves

Beziers curves can be thought of as a higher order extension of linear interpolation.
There are lots of ways to formulate Bezier curves mathematically. Some of these include:

- **de Casteljau** (recursive linear interpolations)
- **Bernstein polynomials** (functions that define the influence of each control point as a function of $t$)
- **Cubic equations** (general cubic equation of $t$)
- **Matrix form**

We will briefly examine ALL of these!

In practice, **matrix form is the most useful in computer animation**, but the others are important for understanding.
Find the point $x$ on the curve as a function of parameter $t$: 
The de Casteljau algorithm describes the curve as a recursive series of linear interpolations.

This form is useful for providing an intuitive understanding of the geometry involved, but it is not the most efficient form.
de Casteljau Algorithm

\[ q_0 = \text{Lerp}(t, p_0, p_1) \]
\[ q_1 = \text{Lerp}(t, p_1, p_2) \]
\[ q_2 = \text{Lerp}(t, p_2, p_3) \]
de Casteljau Algorithm

\[
\mathbf{r}_0 = \text{Lerp}(t, \mathbf{q}_0, \mathbf{q}_1)
\]

\[
\mathbf{r}_1 = \text{Lerp}(t, \mathbf{q}_1, \mathbf{q}_2)
\]
The de Casteljau Algorithm

\[ x = \text{Lerp}(t, \mathbf{r}_0, \mathbf{r}_1) \]
Beziers Curve

https://www.khanacademy.org/partner-content/pixar/animate/parametric-curves/v/animation7
Recursive Linear Interpolation

\[ x = \text{Lerp}(t, r_0, r_1) \]

\[ r_0 = \text{Lerp}(t, q_0, q_1) \]

\[ r_1 = \text{Lerp}(t, q_1, q_2) \]

\[ q_0 = \text{Lerp}(t, p_0, p_1) \]

\[ q_1 = \text{Lerp}(t, p_1, p_2) \]

\[ q_2 = \text{Lerp}(t, p_2, p_3) \]

\[ \text{Lerp}(t, a, b) = (1 - t)a + tb \]
Expanding the Lerps

\[ q_0 = \text{Lerp}(t, p_0, p_1) = (1-t)p_0 + tp_1 \]
\[ q_1 = \text{Lerp}(t, p_1, p_2) = (1-t)p_1 + tp_2 \]
\[ q_2 = \text{Lerp}(t, p_2, p_3) = (1-t)p_2 + tp_3 \]

\[ r_0 = \text{Lerp}(t, q_0, q_1) = (1-t)((1-t)p_0 + tp_1) + t((1-t)p_1 + tp_2) \]
\[ r_1 = \text{Lerp}(t, q_1, q_2) = (1-t)((1-t)p_1 + tp_2) + t((1-t)p_2 + tp_3) \]

\[ x = \text{Lerp}(t, r_0, r_1) = (1-t)((1-t)((1-t)p_0 + tp_1) + t((1-t)p_1 + tp_2)) \]
\[ + t((1-t)((1-t)p_1 + tp_2) + t((1-t)p_2 + tp_3)) \]
Bernstein Polynomial Form

\[
x = (1-t)((1-t)((1-t)p_0 + tp_1) + t((1-t)p_1 + tp_2)) \\
+ t((1-t)((1-t)p_1 + tp_2) + t((1-t)p_2 + tp_3))
\]

\[
x = (1-t)^3 p_0 + 3(1-t)^2 tp_1 + 3(1-t)t^2 p_2 + t^3 p_3
\]

\[
x = (-t^3 + 3t^2 - 3t + 1)p_0 + (3t^3 - 6t^2 + 3t)p_1 \\
+ (-3t^3 + 3t^2)p_2 + (t^3)p_3
\]
Cubic Bernstein Polynomials

\[ x = \left(-t^3 + 3t^2 - 3t + 1\right)p_0 + \left(3t^3 - 6t^2 + 3t\right)p_1 + \left(-3t^3 + 3t^2\right)p_2 + \left(t^3\right)p_3 \]

\[ x = B_0^3(t)p_0 + B_1^3(t)p_1 + B_2^3(t)p_2 + B_3^3(t)p_3 \]

\[ B_0^3(t) = -t^3 + 3t^2 - 3t + 1 \]
\[ B_1^3(t) = 3t^3 - 6t^2 + 3t \]
\[ B_2^3(t) = -3t^3 + 3t^2 \]
\[ B_3^3(t) = t^3 \]
Bernstein Polynomials

\[ f(t) \]

\[ B_0^3 \quad B_1^3 \quad B_2^3 \quad B_3^3 \]
Bernstein Polynomials

\[ B^3_0(t) = -t^3 + 3t^2 - 3t + 1 \]
\[ B^3_1(t) = 3t^3 - 6t^2 + 3t \]
\[ B^3_2(t) = -3t^3 + 3t^2 \]
\[ B^3_3(t) = t^3 \]

\[ B^2_0(t) = t^2 - 2t + 1 \]
\[ B^2_1(t) = -2t^2 + 2t \]
\[ B^2_2(t) = t^2 \]

\[ B^1_0(t) = -t + 1 \]
\[ B^1_1(t) = t \]

\[ B^n_i(t) = \binom{n}{i}(1-t)^{n-i}(t)^i \]
\[ \binom{n}{i} = \frac{n!}{i!(n-i)!} \]

\[ \sum B^n_i(t) = 1 \]
 Bernstein Polynomials

- Bernstein polynomial form of a Bezier curve:

\[
B_i^n(t) = \binom{n}{i} (1 - t)^{n-i} t^i
\]

\[
x(t) = \sum_{i=0}^{n} B_i^n(t) p_i
\]
We start with the de Casteljau algorithm, expand out the math, and group it into polynomial functions of \( t \) multiplied by points in the control mesh.

The generalization of this gives us the Bernstein form of the Bezier curve.

This gives us further understanding of what is happening in the curve:

- We can see the influence of each point in the control mesh as a function of \( t \).
- We see that the basis functions add up to 1, indicating that the Bezier curve is a convex average of the control points.
Cubic Equation Form

\[ x = (-t^3 + 3t^2 - 3t + 1)p_0 + (3t^3 - 6t^2 + 3t)p_1 \\
+ (-3t^3 + 3t^2)p_2 + (t^3)p_3 \]

\[ x = (-p_0 + 3p_1 - 3p_2 + p_3)t^3 + (3p_0 - 6p_1 + 3p_2)t^2 \\
+ (-3p_0 + 3p_1)t + (p_0)1 \]
Cubic Equation Form

\[ x = \left( -p_0 + 3p_1 - 3p_2 + p_3 \right)t^3 + \left( 3p_0 - 6p_1 + 3p_2 \right)t^2 + \left( -3p_0 + 3p_1 \right)t + (p_0) \]

\[ x = a t^3 + b t^2 + c t + d \]

\[ a = \left( -p_0 + 3p_1 - 3p_2 + p_3 \right) \]

\[ b = \left( 3p_0 - 6p_1 + 3p_2 \right) \]

\[ c = \left( -3p_0 + 3p_1 \right) \]

\[ d = (p_0) \]
If we regroup the equation by terms of exponents of $t$, we get it in the standard cubic form.

This form is very good for fast evaluation, as all of the constant terms $(a,b,c,d)$ can be precomputed.

The cubic equation form obscures the input geometry, but there is a one-to-one mapping between the two and so the geometry can always be extracted out of the cubic coefficients.
Cubic Matrix Form

\[ x = at^3 + bt^2 + ct + d \]

\[ a = (-p_0 + 3p_1 - 3p_2 + p_3) \]
\[ b = (3p_0 - 6p_1 + 3p_2) \]
\[ c = (-3p_0 + 3p_1) \]
\[ d = (p_0) \]

\[ x = \begin{bmatrix} t^3 & t^2 & t & 1 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} = \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} p_0 \\ p_1 \\ p_2 \\ p_3 \end{bmatrix} \]
Cubic Matrix Form

\[ \mathbf{x} = \begin{bmatrix} t^3 & t^2 & t & 1 \end{bmatrix} \cdot \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} p_0 \\ p_1 \\ p_2 \\ p_3 \end{bmatrix} \]

\[ \mathbf{x} = \begin{bmatrix} t^3 & t^2 & t & 1 \end{bmatrix} \cdot \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} p_{0x} & p_{0y} & p_{0z} \\ p_{1x} & p_{1y} & p_{1z} \\ p_{2x} & p_{2y} & p_{2z} \\ p_{3x} & p_{3y} & p_{3z} \end{bmatrix} \]
Matrix Form

\[ x = \begin{bmatrix} t^3 & t^2 & t & 1 \end{bmatrix} \cdot \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} p_{0x} & p_{0y} & p_{0z} \\ p_{1x} & p_{1y} & p_{1z} \\ p_{2x} & p_{2y} & p_{2z} \\ p_{3x} & p_{3y} & p_{3z} \end{bmatrix} \]

\[ x = t \cdot B_{Bez} \cdot G_{Bez} \]

\[ x = t \cdot C \]
Matrix Form

- We can rewrite the equations in matrix form.
- This gives us a compact notation and shows how different forms of cubic curves can be related.
- It also is a very efficient form as it can take advantage of existing 4x4 matrix hardware support...
Let’s look at an alternative way to describe a cubic curve.

Instead of defining it with the 4 control points as a Bezier curve, we will define it with a position and a tangent (velocity) at both the start and end of the curve \((p_0, p_1, v_0, v_1)\).
Hermite Curve

\( \mathbf{v}_0 \)

\( \mathbf{p}_0 \)

\( \mathbf{v}_1 \)

\( \mathbf{p}_1 \)
Derivatives

- Finding the derivative (tangent) of a curve is easy:

\[ x = at^3 + bt^2 + ct + d \]

\[ \frac{dx}{dt} = 3at^2 + 2bt + c \]

\[ x = \begin{bmatrix} t^3 & t^2 & t & 1 \end{bmatrix} \cdot \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} \]

\[ \frac{dx}{dt} = \begin{bmatrix} 3t^2 & 2t & 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} \]
Hermite Curves

- We want the value of the curve at $t=0$ to be $x(0) = p_0$, and at $t=1$, we want $x(1) = p_1$.
- We want the derivative of the curve at $t=0$ to be $v_0$, and $v_1$ at $t=1$.

\[
\begin{align*}
\mathbf{x}(0) &= \mathbf{p}_0 = a0^3 + b0^2 + c0 + d = d \\
\mathbf{x}(1) &= \mathbf{p}_1 = a1^3 + b1^2 + c1 + d = a + b + c + d \\
\mathbf{x}'(0) &= \mathbf{v}_0 = 3a0^2 + 2b0 + c = c \\
\mathbf{x}'(1) &= \mathbf{v}_1 = 3a1^2 + 2b1 + c = 3a + 2b + c
\end{align*}
\]
Hermite Curves

\[ p_0 = d \]
\[ p_1 = a + b + c + d \]
\[ v_0 = c \]
\[ v_1 = 3a + 2b + c \]
Hermite Curves

\[ p_0 = d \]
\[ p_1 = a + b + c + d \]
\[ v_0 = c \]
\[ v_1 = 3a + 2b + c \]

\[
\begin{bmatrix}
  p_0 \\
  p_1 \\
  v_0 \\
  v_1 \\
\end{bmatrix} = \begin{bmatrix}
  0 & 0 & 0 & 1 \\
  1 & 1 & 1 & 1 \\
  0 & 0 & 1 & 0 \\
  3 & 2 & 1 & 0 \\
\end{bmatrix} \cdot \begin{bmatrix}
  a \\
  b \\
  c \\
  d \\
\end{bmatrix}
\]
Matrix Form of Hermite Curve

\[
\begin{bmatrix}
  a \\
  b \\
  c \\
  d
\end{bmatrix} = \begin{bmatrix}
  0 & 0 & 0 & 1 \\
  1 & 1 & 1 & 1 \\
  0 & 0 & 1 & 0 \\
  3 & 2 & 1 & 0
\end{bmatrix} \begin{bmatrix}
  0 \\
  1 \\
  1 \\
  0
\end{bmatrix}^{-1} \begin{bmatrix}
  p_0 \\
  p_1 \\
  v_0 \\
  v_1
\end{bmatrix}
\]

\[
\begin{bmatrix}
  a \\
  b \\
  c \\
  d
\end{bmatrix} = \begin{bmatrix}
  2 & -2 & 1 & 1 \\
  -3 & 3 & -2 & -1 \\
  0 & 0 & 1 & 0 \\
  1 & 0 & 0 & 0
\end{bmatrix} \begin{bmatrix}
  p_0 \\
  p_1 \\
  v_0 \\
  v_1
\end{bmatrix}
\]
Matrix Form of Hermite Curve

\[ x = \left[ t^3 \quad t^2 \quad t \quad 1 \right] \cdot \begin{bmatrix} 2 & -2 & 1 & 1 \\ -3 & 3 & -2 & -1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} p_{0x} & p_{0y} & p_{0z} \\ p_{1x} & p_{1y} & p_{1z} \\ v_{0x} & v_{0y} & v_{0z} \\ v_{1x} & v_{1y} & v_{1z} \end{bmatrix} \]

\[ x = t \cdot B_{Hrm} \cdot G_{Hrm} \]

\[ x = t \cdot C \]
The Hermite curve is another geometric way of defining a cubic curve.
We see that ultimately, it is another way of generating cubic coefficients.
We can also see that we can convert a Bezier form to a Hermite form with the following relationship:

\[ C = B_{Bez} \cdot G_{Bez} = B_{Hrm} \cdot G_{Hrm} \]
Rigging and Animation

Animation System

\[ \Phi = [\phi_1, \phi_2, \ldots, \phi_N] \]

Rigging System

renderer

pose

triangles
When we speak of an ‘animation’, we refer to the data required to pose a rig over some range of time.

This should include information to specify all necessary DOF values over the entire time range.

Sometimes, this is referred to as a ‘clip’ or even a ‘move’ (as ‘animation’ can be ambiguous).
If a character has N DOFs, then a pose can be thought of as a point in N-dimensional pose space

\[ \Phi = [\phi_1 \phi_2 \ldots \phi_N] \]

An animation can be thought of as a point moving through pose space, or alternately as a fixed curve in pose space

\[ \Phi = \Phi(t) \]

‘One-shot’ animations are an open curve, while ‘loop’ animations form a closed loop

Generally, we think of an individual ‘animation’ as being a continuous curve, but there’s no strict reason why we couldn’t have discontinuities (cuts)
If the entire animation is an N-dimensional curve in pose space, we can separate that into N 1-dimensional curves, one for each DOF

\[ \phi_i = \phi_i(t) \]

We call these ‘channels’

A channel will refer to pre-recorded or pre-animated data for a DOF, and does not refer to the more general case of a DOF changing over time (which includes physics, procedural animation...)

Channels
Channels

Value

tmin

tmax

Time
Array of Channels

- An animation can be stored as an array of channels.
- A simple means of storing a channel is as an array of regularly spaced samples in time.
- Using this idea, one can store an animation as a 2D array of floats (NumDOFs x NumFrames).
- However, if one wanted to use some other means of storing a channel, they could still store an animation as an array of channels, where each channel is responsible for storing data however it wants.
Array of Poses

- An alternative way to store an animation is as an array of poses

- This also forms a 2D array of floats (NumFrames x NumDOFs)

- Which is better, poses or channels?
Poses vs. Channels

- It depends on your requirements.

- The bottom line:
  - Poses are faster
  - Channels are far more flexible and can potentially use less memory
The array of poses method is about the fastest possible way to playback animation data.

A ‘pose’ (vector of floats) is exactly what one needs in order to pose a rig.

Data is contiguous in memory, and can all be directly accessed from one address.
Array of Channels

- As each channel is stored independently, they have the flexibility to take advantage of different storage options and maximize memory efficiency.

- Also, in an interactive editing situation, new channels can be independently created and manipulated.

- However, they need to be independently evaluated to access the ‘current frame’, which takes time and implies discontinuous memory access.
Channels

- As the array of poses method is very simple, there’s not much more to say about it.
- Therefore, we will concentrate on channels on their various storage and manipulation techniques.
Temporal Continuity

- Sometimes, we think of animations as having a particular frame rate (i.e., 30 fps)
- It’s often a better idea to think of them as being continuous in time and not tied to any particular rate. Some reasons include:
  - Film / NTSC / PAL conversion
  - On-the-fly manipulation (stretching/shrinking in time)
  - Motion blur
Animation Storage

- Regardless of whether one thinks of an animation as being continuous or as having discrete points, one must consider methods of storing animation data.

- Some of these methods may require some sort of temporal discretization, while others will not.

- Even when we do store a channel on frame increments, it’s still nice to think of it as a continuous function interpolating the time between frames.
There are several ways to store channels. Most approaches fall into either storing them in a ‘raw’ frame method, or as piecewise interpolating curves (keyframes).

A third alternative is as a user supplied expression, which is just an arbitrary math function.
Keyframe Channel

- A channel can be stored as a sequence of keyframes.
- Each keyframe has a time and a value and usually some information describing the tangents at that location.
- The curves of the individual spans between the keys are defined by 1-D interpolation (usually piecewise Hermite).
Keyframe Channel
Keyframe

Time

Value

Tangent In

Keyframe (time, value)

Tangent Out
Why Use Keyframes?

- Good user interface for adjusting curves
- Gives the user control over the value of the DOF and the velocity of the DOF
- Define a perfectly smooth function (if desired)
- Can offer good compression
- Video games may consider keyframes for compression purposes, even though they have a performance cost
Keyframed channels form the foundation for animating properties (DOFs) in many commercial animation systems.

Different systems use different variations on the exact math but most are based on some sort of cubic Hermite curves.
Keyframes can be generated automatically from sampled data such as motion capture.

This process is called ‘curve fitting’, as it involves finding curves that fit the data reasonably well.

Fitting algorithms allow the user to specify tolerances that define the acceptable quality of the fit.

This allows two way conversion between keyframe and raw formats, although the data might get slightly distorted with each translation.
class Keyframe {
    float Time;
    float Value;
    float TangentIn, TangentOut;
    char RuleIn, RuleOut;  // Tangent rules
    float A, B, C, D;     // Cubic coefficients
}
Tangent Rules

- Rather than store explicit numbers for tangents, it is often more convenient to store a ‘rule’ and recomputes the actual tangent as necessary.

- Usually, separate rules are stored for the incoming and outgoing tangents.

- Common rules for Hermite tangents include:
  - Flat  \( (\text{tangent} = 0) \)
  - Linear  \( (\text{tangent} \text{ points to next/last key}) \)
  - Smooth  \( (\text{automatically adjust tangent for smooth results}) \)
  - Fixed  \( (\text{user can arbitrarily specify a value}) \)

- Remember that the tangent equals the rate of change of the DOF (or the velocity).

- Note: ‘v’ for tangents (velocity).
Flat tangents are particularly useful for making ‘slow in’ and ‘slow out’ motions (acceleration from a stop and deceleration to a stop)
Linear Tangents

\[ v_0^{out} = v_1^{in} = \frac{p_1 - p_0}{t_1 - t_0} \]
\( v_1^{in} = v_1^{out} = \frac{p_2 - p_0}{t_2 - t_0} \)
Occasionally, one comes across the ‘step’ tangent rule

This is a special case that just forces the entire span to a constant

This requires hacking the cubic coefficients \((a=b=c=0, \ d=p_0)\)
The two main computations a keyframe channel needs to perform are:

- Compute tangents from rules
- Compute cubic coefficients from tangents & other data
Cubic Coefficients

- Keyframes are stored in order of their time.
- Between every two successive keyframes is a *span* of a cubic curve.
- The span is defined by the value of the two keyframes and the outgoing tangent of the first and incoming tangent of the second.
- Those 4 values are multiplied by the Hermite basis matrix and converted to cubic coefficients for the span.
- For simplicity, the coefficients can be stored in the first keyframe for each span.
Hermite Curve (1D)

$p_0$ to $p_1$ with $t_0=0$ and $t_1=1$
Matrix Form of Hermite Curve

\[ f(t) = \begin{bmatrix} t^3 & t^2 & t & 1 \end{bmatrix} \cdot \begin{bmatrix} 2 & -2 & 1 & 1 \\ -3 & 3 & -2 & -1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} p_0 \\ p_1 \\ v_0 \\ v_1 \end{bmatrix} \]

\[ f(t) = t \cdot B_{Hrm} \cdot g_{Hrm} \]
\[ f(t) = t \cdot c \]

- Remember, this assumes that \( t \) varies from 0 to 1
If $t_0$ is the time at the first key and $t_1$ is the time of the second key, a linear interpolation of those times by parameter $u$ would be:

$$t = \text{Lerp}(u, t_0, t_1) = (1-u)t_0 + ut_1$$

The inverse of this operation gives us:

$$u = \text{InvLerp}(t, t_0, t_1) = \frac{t - t_0}{t_1 - t_0}$$

This gives us a $0...1$ value on the span where we now will evaluate the cubic equation.
To evaluate the cubic equation for a span, we must first turn our time $t$ into a $0..1$ value for the span (we’ll call this parameter $u$)

$$u = \text{InvLerp}(t, t_0, t_1) = \frac{t - t_0}{t_1 - t_0}$$

$$x = au^3 + bu^2 + cu + d = d + u(c + u(b + u(a)))$$
Extrapolation Modes

- Channels can specify ‘extrapolation modes’ to define how the curve is extrapolated before $t_{\text{min}}$ and after $t_{\text{max}}$

- Usually, separate extrapolation modes can be set for before and after the actual data

- Common choices:
  - Constant value (hold first/last key value)
  - Linear (use tangent at first/last key)
  - Cyclic (repeat the entire channel)
  - Cyclic Offset (repeat with value offset)
  - Bounce (repeat alternating backwards & forwards)
Extrapolation

- Flat:

- Linear:
Extrapolation

- Cyclic:

- Cyclic Offset:
Extrapolation

- Bounce:
The main runtime function for a channel is something like:

```c
float Channel::Evaluate(float time);
```

This function will be called many times...

For an input time $t$, there are 4 cases to consider:

- $t$ is before the first key (use extrapolation)
- $t$ is after the last key (use extrapolation)
- $t$ falls exactly on some key (return key value)
- $t$ falls between two keys (evaluate cubic equation)
The Channel::Evaluate function needs to be very efficient, as it is called many times while playing back animations.

There are two main components to the evaluation:

- Find the proper span
- Evaluate the cubic equation for the span
To evaluate a channel at some arbitrary time \( t \), we need to first find the proper span of the channel and then evaluate its equation.

As the keyframes are irregularly spaced, this means we have to search for the right one.

If the keyframes are stored as a linked list, there is little we can do except walk through the list looking for the right span.

If they are stored in an array, we can use a binary search, which should do reasonably well.
If a character is playing back an animation, then it will be accessing the channel data sequentially.

Doing a binary search for each channel evaluation for each frame is not efficient for this.

If we keep track of the most recently accessed key for each channel, then it is extremely likely that the next access will require either the same key or the very next one.

This makes sequential access of keyframes potentially very fast.
Robustness

- The **channel should always return some reasonable value regardless of what time t was passed in**
  - If there are no keys in the channel, it should just return 0
  - If there is just 1 key, it should return the value of that key
  - If there are more than 1 key, it should evaluate the curve or use an extrapolation rule if t is outside of the range
  - At a minimum, the ‘constant’ extrapolation rule should be used, which just returns the value of the first (or last) key if t is before (or after) the keyframe range

- When creating new keys or modifying the time of a key, one needs to verify that its time stays between the key before and after it
animation {
    range [time_start] [time_end]
    numchannels [num]
    channel {
        extrapolate [extrap_in] [extrap_out]
        keys [numkeys] {
            [time] [value] [tangent_in] [tangent_out]
            ...
        }
    }
    channel ...
}
The first 3 channels will be the root translation (x,y,z)

After that, there will be 3 rotational channels for every joint (x,y,z) in the same order that the joints are listed in the .skel file
Anim classes

- Suggested classes:
  - Keyframe: stores time, value, tangents, cubics...
  - Channel: stores an array (or list) of Keyframes
  - Animation: stores an array of Channels
  - Player: stores pointer to an animation & pointer to skeleton. Keeps track of time, accesses animation data & poses the skeleton.

- Optional:
  - Rig: simple container for a skeleton, skin, and morphs
  - Pose: array of floats (or just use stl vector)
  - ChannelEditor: it’s always nice to separate editor classes from the data that they edit
Blending & State Machines
Now that we understand how to manipulate animation data, we can edit and play back simple animation.

The subject of blending and sequencing encompasses a higher level of animation playback, involving constructing the final pose out of a combination of various inputs.

We will limit today’s discussion to encompass only pre-stored animation (channel) data as the ultimate input. But it is also possible to combine with procedural animation.
Most areas of computer animation have been pioneered by the research and special effects industries.

Blending and sequencing, however, is one area where video games have made a lot of real progress in this area towards achieving interactively controllable and AI characters in complex environments...

The special effects industry is using some game related technology more and more (battle scenes in Lord of the Rings...):
http://www.massivesoftware.com/about.html
Animation Playback
Poses

- A pose is an array of values that maps to a rig.

- If the rig contains only simple independent DOFs, the pose can just be an array of floats.

- If the rig contains quaternions or other complex coupled DOFs, they may require special handling by higher level code.

- Therefore, for generality, we will assume that a pose contains both an array of $M \geq 0$ floats and an additional array of $N \geq 0$ quaternions.

$$\Phi = [\phi_0 \ldots \phi_{M-1} \quad q_0 \ldots q_{N-1}]$$
Remember that the AnimationClip stores an array of channels for a particular animation (or it could store the data as an array of poses...)

This should be treated as constant data, especially in situations where multiple animating characters may simultaneously need to access the animation (at different time values)

For playback, animation is accessed as a pose. Evaluation requires looping through each channel.

class AnimationClip {
    void Evaluate(float time, Pose &p);
};
We need something that ‘plays’ an animation. We will call it an animation *player*.

At it’s simplest, an animation player would store a `AnimationClip*`, `Rig*`, and a float time.

As an active component, it would require some sort of `Update()` function.

This update would increment the time, evaluate the animation, and then pose the rig.
class AnimationPlayer {
    float Time;
    AnimationClip *Anim;
    Pose P;
    public:
    void SetClip(AnimationClip &clip);
    const Pose &GetPose();
    void Update();
};
Animation Player

- A simple player just needs to increment the Time and access the new pose once per frame.

- The first question that comes up though, is what to do when it gets to the end of the animation clip?
  - Loop back to start
  - Hold on last frame
  - Deactivate itself... (return 0 pose?)
  - Send a message...
Some features we may want to add for a more versatile animation player include:

- Variable playback rate
- Play backwards (& deal with hitting the beginning)
- Pause

It’s kinda like a DVD player...
As we will use players and static poses as basic components in our blending discussion, we will make a notation for them:

- **look_right**
  - static pose

- **walk**
  - current pose (Animation Player)
Animation Blending
We can define blending operations that affect poses.

A blend operation takes one or more poses as input and generates one pose as output.

In addition, it may take some auxiliary data as input (control parameters, etc.)
Generic Blend Operation

pose1 \rightarrow \text{BLENDER} \rightarrow \text{aux data} \rightarrow \text{output pose} \rightarrow \ldots \text{poseN}
Cross Dissolve

- Perhaps the most common and useful pose blend operation is the ‘cross dissolve’

- Also known as: Lerp (linear interpolation), blend, dissolve…

- The cross dissolve blender takes two poses as input and an additional float as the blend factor (0…1)
The two poses are basically just interpolated

The DOF values can use Lerp, but the quaternions should use the ‘Slerp’ operation (spherical linear interpolate)

\[
\phi' = \text{Lerp}(t, \phi_1, \phi_2) = (1 - t)\phi_1 + t\phi_2
\]

\[
q' = \text{Slerp}(t, q_1, q_2) = \frac{\sin((1 - t)\theta)}{\sin \theta} q_1 + \frac{\sin(t\theta)}{\sin \theta} q_2
\]
If a DOF represents an angle, we may want to have the interpolation check for crossing the +180 / -180 boundary

\[
\text{if } (\phi_1 - \phi_2 > 180^\circ) \quad \phi' = \text{Lerp}(t, \phi_1 - 360^\circ, \phi_2) \\
\text{else if } (\phi_2 - \phi_1 > 180^\circ) \quad \phi' = \text{Lerp}(t, \phi_1, \phi_2 - 360^\circ) \\
\text{else } \phi' = \text{Lerp}(t, \phi_1, \phi_2)
\]

Unfortunately, this complicates the concept of a DOF (and a pose) a bit more. Now we must also consider that some DOFs behave in different ways than others.
Also, for quaternions, we may wish to force the interpolation to go the ‘short way’:

\[
if (q_1 \cdot q_2 < 0) \quad q' = \text{Slerp}(t, -q_1, q_2)
\]

\[
else \quad q' = \text{Slerp}(t, q_1, q_2)
\]
Consider a situation where we want a character to blend from a stand animation to a walk animation.
Cross Dissolve: Stand to Walk

- We could have two independent animations playing (stand & walk) and then gradually ramp the ‘t’ value from 0 to 1

- If the transition is reasonably quick (say <0.5 second), it might look OK

- Note: this is just a simple example of a dissolve and not necessarily the best way to make a character start walking...
Cross Dissolve: Walk to Run

- Blending from a walk to a run requires some additional consideration...
We want to make sure that the walk and run are in phase when we blend between them.

One could animate them in a consistent way so that the two clips both start at the same phase.

But, let’s assume they aren’t in sync...

Instead, we’ll just store an offset for each clip that marks some synchronization point (say at the time when the left foot hits the ground).

We’ll call these offsets $o_{\text{walk}}$ and $o_{\text{run}}$. 
Let’s assume that \( f \) is our dissolve factor (0…1) where \( f=0 \) implies walking and \( f=1 \) implies running.

The resulting velocity that the character should move is simply:

\[
v' = \text{Lerp}(f, v_{\text{walk}}, v_{\text{run}})
\]

To get the animations to stay in phase, however, we need to adjust the speeds that they are playing back.

This means that when we’re halfway between walk and run, the walk will need to be sped up and the run will need to be slowed down.
Cross Dissolve: Walk to Run

- As we are sure that we want the two to stay in phase, we can just lock them together.

- For example, we will just say that if \( t_{walk} \) is the current time of the walk animation, then \( t_{run} \) should be:

\[
t_{run} = \text{mod}\left( (t_{walk} - o_{walk} + d_{walk}) \frac{d_{run}}{d_{walk}} + o_{run}, d_{run} \right)
\]
We can also define some blenders for basic math operations:

- **ADD**
  
  \[ \text{ADD}: \text{pose}_1 + \text{pose}_2 \]

- **SUBTRACT**
  
  \[ \text{SUBTRACT}: \text{pose}_1 - \text{pose}_2 \]

- **SCALE**
  
  \[ \text{SCALE}: f \times \text{pose}_1 \]
Basic Math Blend Operations

- It's not always obvious how to define consistent behaviors between independent DOFs and quaternions.
- Quaternion addition and subtraction don’t really give an expected result.
- Addition of orientations implies that you start with the first orientation and then you do a rotation from there that corresponds to how the second orientation is rotated from neutral.
- This behavior is more like quaternion multiplication (although quaternion multiplication is not commutative).
A reasonable behavior for an add blender could be:

\[ \phi' = \phi_1 + \phi_2 \]
\[ q' = q_1 q_2 \]

For subtraction, we could multiply by the conjugate of the quaternion

\[ \phi' = \phi_1 - \phi_2 \]
\[ q' = q_1 q_2^* \]
\[ q^* = [q_0, -q_1, -q_2, -q_3] \]
As we want our quaternions to stay unit length, we don’t really want to scale them.

In any case, scaling a quaternion has no effect on the resulting orientation!

Instead, we can think of scaling as moving towards or away from 0 (i.e., scaling by a number less than 1 brings us closer to 0, scaling by >1 takes us away from 0…)

Therefore, we could define the scale blender as:

\[ \phi' = f\phi_1 \]

\[ q' = Slerp(f, [1 \ 0 \ 0 \ 0], q_1) \]
As an example of math blending operations, consider a character that walks and turns.

One approach to achieving this is to have an underlying walk animation and some body turn on top of it.

We make a static ‘look_right’ pose and a static ‘default’ pose.

The subtraction gives us the difference between look_right and default.

If we scale this and then add it on top of the underlying walk animation. The scale we use can be based on how hard the character is turning (-1...1).
Math Operations: Body Turn

- `walk`
- `f`
- `look_right` ➔ `SUBTRACT` ➔ `SCALE` ➔ `ADD` ➔ `output pose`
- `default`
We can also speed this up by precomputing the subtraction and making a combined add/scale blender.
**Bilinear Blend**

- **BILINEAR**
  - Input: \( s, t \)
  - Output: \( \text{output pose} \)

- **DISSOLVE**
  - Input: \( s \)
  - Output: \( \text{output pose} \)
Bilinear Blend

Bilinear blend is an extension to the cross dissolve that takes four input poses and two interpolation parameters $s$ & $t$

https://www.youtube.com/watch?v=HeHvIEYpRbM
Bilinear Blend

- Bilinear (and trilinear...) blends can be useful for a wide range of applications
- As one example, consider a video game character who has to aim a weapon
- The character must be able to stand still and aim at any object within +/- 135 degrees to the side to side and +/- 45 degrees up and down
- An animator can supply key poses at 45 degree increments in both directions
- Then, for any desired angle, we can find the right four targets and do a bilinear blend
We can also have a blender that combines poses in different ways.

For example, we might want to treat the upper body separately from the lower body, or treat each limb separately, etc.

We can use different blenders for each body section and then combine them into a final pose.

This also implies that we can use smaller pose vectors in each body section to save computations and memory.

The actual combine operation could just have lookup tables that map index values of the incoming poses to index values of the final pose.
Mirroring animations across the x=0 plane can be an effective way to save memory and complexity.

It requires that a character is symmetrical (or close enough...)

Like the combine blender, mirroring requires some sort of table as input that describes how to mirror each DOF.

Different DOFs will need different treatment:

- **Translation**: \( x' = -x, \ y' = y, \ z' = z \)
- **Rotation**: \( x' = x, \ y' = -y, \ z' = -z \)
- **Quaternion**: \( q' = [q_0, \ q_1, \ -q_2, \ -q_3] \)

Also, DOFs on the right need to be swapped with DOFs on the left.
Multi-Track Blending

- One can also think of an animation blending system as being similar to a multi-track audio (or video) editing system.

- Different animations (or poses) can be placed in different ‘tracks’ and each track could have some additional controls and custom behaviors.
Animation State Machines
Blending is great for combining a few motions, but it does not address the issue of sequencing different animations over time.

For this, we will use a state machine.

We will define the state machine as a connected graph of states and transitions.

At any time, exactly one of the states is the current state.

Transitions are assumed to happen instantaneously.
State Machines

- **state_A**
  - EVENT1 to **state_B**
  - EVENT2 to **state_C**

- **state_B**
  - EVENT3 to **state_D**

- **state_C**
  - EVENT4 to **state_B**
  - EVENT5 to **state_E**

- **state_D**
  - EVENT6 to **state_E**
State Machines

- In the context of animation sequencing, we think of states as representing individual animation clips and transitions being triggered by some sort of event.

- An event might come from some internal logic or some external input (button press...).
Consider a simple state machine where a character jumps upon receiving a JUMP_PRESS message.
More Complex Jump

- **stand**
  - JUMP_PRESS
  - **stand2crouch**
    - JUMP_RELEASE
    - **crouch**
      - JUMP_RELEASE
      - **takeoff**
        - NEAR_GROUND
        - **land**
    - **hop**
      - JUMP_RELEASE
      - **float**

stand {JUMP_PRESS stand2crouch }
stand2crouch {
    JUMP_RELEASE hop
    END crouch }
crouch {JUMP_RELEASE takeoff }
takeoff {END float }
hop {END float }
float {NEAR_GROUND land }
land {END stand }
Creating State Machines

- Typing in text
- Graphical state machine editor
- Automated state machine generation (motion graphing)