Lecture 2: Interpolation, Blending and Skinning

Computer Animation
In the last lecture

- Introduction to the course
- History of computer animation
- Animation Basics
  - Linear algebra for character animation (supplementary material)
- Forward Kinematics
- Orientation

- A video recording of the first lecture is on Teams. Slides are on the course website.
In this lecture

- Cubic curves
- Keyframes & Channels
- Animation Blending
- Skinning
Reminders

- Do you know your team members for the project and presentation? Please fill in the Google forms sent by email.

- Do you want to know papers to review and what paper to present?
  - The set of 22 papers was shared on a Gdrive in the email sent.
  - Select 1 paper for presentation and fill in the form with names of your co-presenters
  - Select also 6 papers (individually) that you want to read as we go (no need to tell the teachers in advance). You need to submit the paper review 1 day before the day the paper is presented.

- Please fill in the forms for groups till 3rd May Wednesday 23:59
  - After this date, we will assign you to groups.
Cubic Curves
Bezier Curves

- Bezier curves can be thought of as a higher order extension of linear interpolation.
Beziers Curve Formulation

- There are lots of ways to formulate Bezier curves mathematically. Some of these include:
  - de Castlejau (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
Paul de Casteljau (born 1930) is a French physicist and mathematician.

- In 1959, while working at Citroën, he developed an algorithm for evaluating calculations on a certain family of curves.
- Later it was formalized and popularized by engineer Pierre Bézier, and the curves called De Casteljau curve or Bézier curves.
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

\[ r_0 = \text{Lerp}(t, q_0, q_1) \]
\[ r_1 = \text{Lerp}(t, q_1, q_2) \]
de Casteljau Algorithm

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

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

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

\[ x = Lerp(t, r_0, r_1) = (1 - t)((1 - t)((1 - t)p_0 + tp_1) + t((1 - t)p_1 + tp_2)) + t((1 - t)((1 - t)p_1 + tp_2) + t((1 - t)p_2 + tp_3)) \]
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
Bernstein Polynomials

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

\[ B_0^2(t) = t^2 - 2t + 1 \]
\[ B_1^2(t) = -2t^2 + 2t \]
\[ B_2^2(t) = t^2 \]

\[ B_0^1(t) = -t + 1 \]
\[ B_1^1(t) = t \]

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

\[ \sum B_i^n(t) = 1 \]
Bernstein Polynomials

- Bernstein polynomial form of a Bezier curve:

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

\[ x(t) = \sum_{i=0}^{n} B^n_i(t)p_i \]
Bernstein Polynomials

- 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

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

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

\[ a = \left( -p_0 + 3p_1 - 3p_2 + p_3 \right) \]
\[ b = \left( 3p_0 - 6p_1 + 3p_2 \right) \]
\[ c = \left( -3p_0 + 3p_1 \right) \]
\[ d = (p_0) \]
Cubic Equation Form

- 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

\[ \mathbf{x} = \begin{bmatrix} t^3 & t^2 & t & 1 \end{bmatrix} \cdot \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} \mathbf{p}_0 \\ \mathbf{p}_1 \\ \mathbf{p}_2 \\ \mathbf{p}_3 \end{bmatrix} \]

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

\[
\mathbf{x} = t \cdot \mathbf{B}_{Bez} \cdot \mathbf{G}_{Bez}
\]

\[
\mathbf{x} = t \cdot \mathbf{C}
\]
Hermite Form

- 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

\( p_0 \)  \( v_0 \)  \( p_1 \)  \( v_1 \)
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 $\mathbf{x}(0)=p_0$, and at $t=1$, we want $\mathbf{x}(1)=p_1$.
- We want the derivative of the curve at $t=0$ to be $\mathbf{v}_0$, and $\mathbf{v}_1$ at $t=1$.

\[
\begin{align*}
\mathbf{x}(0) &= p_0 = a0^3 + b0^2 + c0 + d = d \\
\mathbf{x}(1) &= p_1 = a1^3 + b1^2 + c1 + d = a + b + c + d \\
\mathbf{x}'(0) &= \mathbf{v}_0 = 3a0^2 + 2b0 + c = c \\
\mathbf{x}'(1) &= \mathbf{v}_1 = 3a1^2 + 2b1 + c = 3a + 2b + c
\end{align*}
\]
Hermite Curves

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

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

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

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

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

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

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

\[ x = t \cdot C \]
Hermite Curves

- 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:

\[ \mathbf{C} = \mathbf{B}_{\text{Bez}} \cdot \mathbf{G}_{\text{Bez}} = \mathbf{B}_{\text{Hrm}} \cdot \mathbf{G}_{\text{Hrm}} \]
Keyframes & Channels
Rigging and Animation

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

Animation System → pose → Rigging System → triangles → Renderer
Pose Space

- If a character has N DOFs, then a pose can be thought of as a point in N-dimensional pose space
  \[ \Phi = \begin{bmatrix} \phi_1 & \phi_2 & \ldots & \phi_N \end{bmatrix} \]

- 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

Time

Value

\[ \text{tmin} \quad \text{tmax} \]

Time

\[ \text{tmin} \quad \text{tmax} \]
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
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

Tangent Out

Keyframe (time,value)
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
class Keyframe {
    float Time;
    float Value;
    float TangentIn,TangentOut;
    char RuleIn,RuleOut;  // Tangent rules
    float A,B,C,D;       // Cubic coefficients
}

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

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

Common rules for Hermite tangents include:
- Flat (tangent = 0)
- Linear (tangent points to next/last key)
- Smooth (automatically adjust tangent for smooth results)
- Fixed (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)
\[ v_{1}^{in} = v_{1}^{out} = \frac{p_{2} - p_{0}}{t_{2} - t_{0}} \]
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)

\[ t_0 = 0 \quad \text{and} \quad t_1 = 1 \]
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)
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
Random Access

- 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.
animation {
    range [time_start] [time_end]
    numchannels [num]
    channel {
        extrapolate [extrap_in] [extrap_out]
        keys [numkeys] {
            [time] [value] [tangent_in] [tangent_out]
            ...
        }
    }
    channel ...
}
**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
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 Blending
Blending Overview

- 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

BLENDER

pose1 ...poseN

aux data

output pose
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).

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

\[
\mathbf{q}' = \text{Slerp}(t, \mathbf{q}_1, \mathbf{q}_2) = \frac{\sin((1-t)\theta)}{\sin \theta} \mathbf{q}_1 + \frac{\sin(t\theta)}{\sin \theta} \mathbf{q}_2
\]
We can also define some blenders for basic math operations:

- **ADD**: pose1 + pose2
- **SUBTRACT**: pose1 - pose2
- **SCALE**: f * pose1
Add & Subtract Blenders

- 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
  \[ \phi' = \phi_1 - \phi_2 \]
  \[ q^* = q_1 q_2^* \]
  \[ q^* = \begin{bmatrix} q_0 & -q_1 & -q_2 & -q_3 \end{bmatrix} \]
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

ADD
output pose
SCALE
SUBTRACT
look_right
default

walk

f

ADD
SCALE
SUBTRACT
Bilinear Blend

BILINEAR

output pose

pose1 pose2 pose3 pose4

s,t

DISSOLVE

output pose

s

pose1 pose2 pose3 pose4

DISSOLVE

DISSOLVE

pose1 pose2

s

pose3 pose4

s

t

DISSOLVE

output pose
Bilinear Blend

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

Blending example in Unity: [https://www.youtube.com/watch?v=HeHvlEYpRbM](https://www.youtube.com/watch?v=HeHvlEYpRbM)
Bilinear 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
Animation State Machines
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

[Diagram showing state transitions]

- State A
- State B
- State C
- State E
- State D

Events:
- EVENT1
- EVENT2
- EVENT3
- EVENT4
- EVENT5
- EVENT6
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...).
More Complex Jump

1. Stand
2. Jump Press
3. Stand to Crouch
4. Crouch
5. Float
6. Takeoff
7. Jump Release
8. Near Ground
9. Jump Release
10. Hop
11. Float
12. Land
Skinning
Skinning
With the smooth skin algorithm, a vertex can be attached to more than one joint with adjustable weights that control how much each joint affects it.

Vertices rarely need to be attached to more than three joints.

Each vertex is transformed a few times and the results are blended.

The smooth skin algorithm has many other names: blended skin, skeletal subspace deformation (SSD), multi-matrix skin, matrix palette skinning...
The deformed vertex position is a weighted average:

\[ v' = w_1(v \cdot M_1) + w_2(v \cdot M_2) + \ldots + w_N(v \cdot M_N) \]

or

\[ v' = \sum w_i (v \cdot M_i) \]

where

\[ \sum w_i = 1 \]
To rig a skinned character, one must have a geometric skin mesh and a skeleton.

Usually, the skin is built in a relatively neutral pose, often in a comfortable standing pose.

The skeleton, however, might be built in more of a zero pose where the joints DOFs are assumed to be 0, causing a very stiff, straight pose.

To attach the skin to the skeleton, the skeleton must first be posed into a binding pose.

Once this is done, the vertices can be assigned to joints with appropriate weights.
Binding Matrices

- With rigid parts or simple skin, $\mathbf{v}$ can be defined local to the joint that transforms it.

- With smooth skin, several joints transform a vertex, but it can’t be defined local to all of them.

- Instead, we must first transform it to be local to the joint that will then transform it to the world.

- To do this, we use a binding matrix $\mathbf{B}$ for each joint that defines where the joint was when the skin was attached and premultiply its inverse with the world matrix:

$$
\mathbf{M}_i = \mathbf{B}_i^{-1} \cdot \mathbf{W}_i
$$
Overall skinning algorithm

- **Step 1:** Fit the skeleton into the mesh and find the matrix that defines where the joint was when the mesh was attached to the skeleton (Binding matrix)

- **Step 2:** Convert the position of vertices to the local coordinates of each bone by inverting the binding matrix

- **Step 3:** Move the joints to the new pose, compute the global position of the vertices using the new local-to-global matrix

- **Step 4:** Blend the results between bones
Example

- What is the position of point A after the elbow is bent 90 degrees?

- Assuming the weight for the upper arm is 0.8 and forearm is 0.2
- Assuming it is a point of the upper arm, the position is \((3,1)\)

- Assuming it is a point of the forearm, it is \((5,1)\)

- Adding the weights
  - \(0.8 \cdot (3,1) + 0.2 \cdot (5,1) = (3.4, 1)\)
How to decide the weights?

- Decide the mapping of the vertex to the bone

Rig Data Flow

- Input DOFs

\[ \Phi = \begin{bmatrix} \phi_1 & \phi_2 & \ldots & \phi_N \end{bmatrix} \]

- Rigging system (skeleton, skin...)

- Output renderable mesh (vertices, normals...)
Painting weights

https://www.youtube.com/watch?v=Ah-Jk7d3ok5
Algorithm Overview

Skin::Update() (view independent processing)
- Compute skinning matrix for each joint: $M = B^{-1} \cdot W$ (you can precompute and store $B^{-1}$ instead of $B$)
- Loop through vertices and compute blended position & normal

Skin::Draw() (view dependent processing)
- Loop through triangles and draw using world space positions & normal
Skinning Equations

- **Skeleton**

\[ L = L_{\text{joint}}(\phi_1, \phi_2, \ldots, \phi_N) \]
\[ W = L \cdot W_{\text{parent}} \]

- **Skinning**

\[ v' = \sum w_i v \cdot B_i^{-1} \cdot W_i \]
\[ n^* = \sum w_i n \cdot B_i^{-1} \cdot W_i \]
\[ n' = \frac{n^*}{|n^*|} \]
Limitations of Smooth Skin

- Smooth skin is very simple and quite fast, but its quality is limited

- The main problems are:
  - Joints tend to collapse as they bend more
  - Very difficult to get specific control
  - Unintuitive and difficult to edit

- Still, it is built in to most 3D animation packages

- If nothing else, it is a good baseline upon which more complex schemes can be built
Limitations of Smooth Skin

Skin collapse (bending) and twisting (candy wrapper) effect

Pose Space Deformation: A Unified Approach to Shape Interpolation and Skeleton-Driven Deformation, Lewis, Cordner and Fong, SIGGRAPH 2000
Why does it happen?

if it is a lower arm point

if it is a upper arm poir

preferred location
Why does it happen?

By linear interpolation

preferred location
Proposed to overcome the bending and candy wrapper effects
Currently, found in most 3D modelling tools, e.g. Maya, Blender

Geometric Skinning with Approximate Dual Quaternion Blending, Kavan Collins, Zara and O'Sullivan, Siggraph 2008
Supplementary material

Siggraph course on skinning (2014):
http://skinning.org/

Skinning: Real-time Shape Deformation
ACM SIGGRAPH 2014 Course
ACM SIGGRAPH Asia 2014 Invited Course
Symposium on Geometry Processing 2015 Invited Course
International Geometry Summit 2016 Invited Course

Alec Jacobson
Columbia University
Zhigang Deng
University of Houston
Ladislav Kavan
University of Pennsylvania
J.P. Lewis
Victoria University, Weta Digital

SIGGRAPH Asia lecturer: Yotam Gingold
George Mason University

Course Materials

- Part I: Direct methods (Ladislav Kavan)
  Course notes | Slides

- Part II: Automatic methods (Alec Jacobson)
  Course notes | Course notes (low resolution) | Slides | Slides (167MB .pptx with videos)

- Part III: Example-based methods (JP Lewis)
  Course notes
Another extension to the smooth skinning algorithm is to allow the vertices to be modeled at key values along the joints motion.

For an elbow, for example, one could model it straight, then model it fully bent.

These shapes are interpolated local to the bones before the skinning is applied.
To compute a blended vertex position:

\[ \mathbf{v}' = \mathbf{v}_{base} + \sum \phi_i \cdot (\mathbf{v}_i - \mathbf{v}_{base}) \]

The blended position is the base position plus a contribution from each target whose DOF value is greater than 0.

To blend the normals, we use a similar equation:

\[ \mathbf{n}' = \mathbf{n}_{base} + \sum \phi_i \cdot (\mathbf{n}_i - \mathbf{n}_{base}) \]

We won’t normalize them now, as that will happen later in the skinning phase.
Layered Approach

- We use a simple layered approach
  - Skeleton Kinematics
  - Shape Interpolation
  - Smooth Skinning

- Most character rigging systems are based on some sort of layered system approach combined with general purpose data flow to allow for customization
Free-Form Deformations

- FFDs are a class of deformations where a low detail control mesh is used to deform a higher detail skin.

- There are a lot of variations on FFDs based on the topology of the control mesh.

The original type of FFD uses a simple regular lattice placed around a region of space.

- The lattice is divided up into a regular grid.
- When the lattice points are then moved, they describe smooth deformation in their vicinity.
Lattice FFDs

- We start by defining the undeformed lattice space:

\[ x(s,t,u) = x_0 + s \cdot s_0 + t \cdot t_0 + u \cdot u_0 \]

\[ 0 \leq s \leq 1 \quad 0 \leq t \leq 1 \quad 0 \leq u \leq 1 \]

\[
M = \begin{bmatrix}
  s_x & s_y & s_z & 0 \\
  t_x & t_y & t_z & 0 \\
  u_x & u_y & u_z & 0 \\
  x_{0x} & x_{0y} & x_{0z} & 1
\end{bmatrix}
\]
Lattice FFDs

- We then define the number of sections in the 3 lattice dimensions:

\[1 \leq l, m, n \leq 3\]

- And then set the initial lattice positions: \(\mathbf{p}_{ijk}\)

\[
\mathbf{p}_{ijk} = \mathbf{x}_0 + \frac{i}{l} \cdot \mathbf{s}_0 + \frac{j}{m} \cdot \mathbf{t}_0 + \frac{k}{n} \cdot \mathbf{u}_0
\]
Lattice FFDs

- To deform a point $x$, we first find the $(s, t, u)$ coordinates:

$$
\begin{bmatrix}
  s & t & u & 1
\end{bmatrix} = x \cdot \mathbf{M}^{-1}
$$

- Then deform that into world space (similar to Bezier curves and Bernstein polynomials):

$$
x_{ffd} = \sum_{i=0}^{l} \binom{l}{i} (1-s)^{l-i} s^i \cdot \left( \sum_{j=0}^{m} \binom{m}{j} (1-t)^{m-j} t^j \cdot \left( \sum_{k=0}^{n} \binom{n}{k} (1-u)^{n-k} u^k \cdot p_{ijk} \right) \right)
$$
The concept of FFDs was later extended to allow an arbitrary topology control volume to be used.
Another type of deformation allows the user to place lines or curves within a skin.

When the lines or curves are moved, they distort the space around them.

Multiple lines & curves can be placed near each other and will properly interact.

The motion of the skin is based on the motion of the underlying muscle and bones. Therefore, in an anatomical simulation, the tissue beneath the skin must be accounted for.

One can model the bones, muscle, and skin tissue as deformable bodies and then use physical simulation to compute their motion.

Various approaches exist ranging from simple approximations using basic primitives to detailed anatomical simulations.
Skín & Muscle Simulation

- Bones are essentially rigid
- Muscles occupy almost all of the space between bone & skin
- Although they can change shape, muscles have essentially constant volume
- The rest of the space between the bone & skin is filled with fat & connective tissues
- Skin is connected to fatty tissue and can usually slide freely over muscle
- Skin is anisotropic as wrinkles tend to have specific orientations
Some simplified anatomical models use ellipsoids to model bones and muscles

Muscles are attached to bones, sometimes with tendons as well.

The muscles contract in a volume preserving way, thus getting wider as they get shorter.
Complex musculature can be built up from lots of simple primitives
Detailed Anatomical Models

- One can also do detailed simulations that accurately model bone & muscle geometry, as well as physical properties

- e.g. Visible Human Project
Body Scanning

- Data input has become an important issue for the various types of data used in computer graphics.

- Examples:
  - Geometry: Laser scanners
  - Motion: Optical motion capture
  - Materials: Gonioreflectometer
  - Faces: Computer vision

- Recently, people have been researching techniques for directly scanning human bodies and skin deformations.
Practical approaches tend to use either a 3D model scanner (like a laser) or a 2D image based approach (computer vision)

The skin is scanned at various key poses and some sort of 3D model is constructed.

Some techniques attempt to fit this back onto a standardized mesh, so that all poses share the same topology. This is difficult, but it makes the interpolation process much easier.

Other techniques interpolate between different topologies. This is difficult also.

Prototypical Bodies

http://graphics.cs.cmu.edu/projects/muscle/
http://dyna.is.tue.mpg.de/

Siggraph course on “Learning body shapes in motion”, 2016
On 3\textsuperscript{rd} May Wednesday, you will have a mocap tutorial in groups. An email will be sent by Kazi for the organization.

No lecture on 5\textsuperscript{th} May (Liberation Day).

On May 10\textsuperscript{th}, I will talk about facial animation and deep learning.
What to do after this lecture

- Find your groups, meet your team members
- Select your paper(s) and start reading
- Attend the mocap tutorial and start making yourself familiar with skills/tools needed for your project
- Enjoy the weekend! 😊
References/Supplementary Material

- Siggraph course on skinning (2014): http://skinning.org/

- Siggraph course on “Learning body shapes in motion”, 2016
  https://dl.acm.org/doi/10.1145/2897826.2927326

- Some of the slides of this lecture are based on the Computer Animation course at the University of California San Diego.