Splines & Kinematics
• Splines

• Forward Kinematics

• Inverse Kinematics
Recall: 3D Animation

- Using meshes, materials, and rendering to produce 3D animated sequences
- Use a photorealistic renderer to make results photorealistic
- **Today**: No need to draw anything, computer takes care of everything
  - Set keyframes by hand
    - **Forward Kinematics**
    - **Inverse Kinematics**
  - Allow keyframes to interpolate
    - **Splines**
Keyframing

• Set keyframes at important locations in the animation
  • Have the computer interpolate the rest

• Can keyframe anything!
  • Color
  • Light intensity
  • Camera zoom

• Problem: how should data interpolate?
  • Linearly?
  • Along a curve/arc?
Linear Interpolation

- basic idea, connect the dots
- yields rough motions at keyframes
Piecewise Polynomial Interpolation doesn’t pass through keyframes yields smooth motions
Mathematical theory of interpolation arose from study of thin strips of wood or metal ("splines") under various forces

- **Splines** help us define how interpolation should occur
• In this course, a spline is any piecewise cubic polynomial function

\[ f(t) = \sum_{j=1}^{d} c_i t^j =: p_i(t) \]

for \( t_i \leq t \leq t_{i+1} \)

• **Common spline:** cubic polynomial:

\[ p(t) = at^3 + bt^2 + ct + d \]

• 3 goals of splines:
  • Interpolation
  • Continuity
  • Locality

**Many solutions!**
• **Interpolation**: does the spline pass through the control points

\[ f(t_i) = f_i \quad \forall i \]

• For every keyframe \( f_i \), there exists some time \( t_i \) where the interpolation of \( f \) equals the keyframe \( f_i \)
**Continuity**: Continuity comes in levels. We check whether the spline and its derivatives are continuous at the control points.

- **C0 continuity** – keyframes are continuous
- **C1 continuity** – first derivative is continuous
- **C2 continuity** – second derivative is continuous

- By default, most applications will have keyframes connect
  - Gives us C0 continuity for free
Locality

- **Locality**: moving one control point does not modify the whole curve

- Important from a user perspective
  - Need to be able to make small, local changes to spline
Piecewise Cubic Polynomial

- So why piecewise cubic polynomials?

\[ p(t) = at^3 + bt^2 + ct + d \]

- Cubic polynomial coefficients can be broken down into their **keyframe and tangent components**
  - Animator specifies where a curve starts, ends, and the tangents at those points
    \[ p(0) = p_0 \quad \Rightarrow \quad d = p_0 \]
    \[ p(1) = p_1 \quad \Rightarrow \quad a + b + c + d = p_1 \]
    \[ p'(0) = u_0 \quad \Rightarrow \quad c = u_0 \]
    \[ p'(1) = u_1 \quad \Rightarrow \quad 3a + 2b + c = u_1 \]

- Gives us 4 constraints
  - Can be turned into 4 coefficients
Piecewise Cubic Polynomial

• Can also write:
  \[ p(t) = at^3 + bt^2 + ct + d \]

• As a linear system!
  \[
  \begin{bmatrix}
    0 & 0 & 0 & 1 \\
    1 & 1 & 1 & 1 \\
    0 & 0 & 1 & 0 \\
    3 & 2 & 1 & 0 
  \end{bmatrix}
  \begin{bmatrix}
    a \\
    b \\
    c \\
    d 
  \end{bmatrix}
  =
  \begin{bmatrix}
    p_0 \\
    p_1 \\
    u_0 \\
    u_1 
  \end{bmatrix}
  \]
Runge Phenomenon

Tempting to use higher-degree polynomials to get higher-order continuity

Can lead to oscillation ultimately worse approximation
Natural Splines

- Can build a spline out of piecewise cubic polynomials \( p_i \)
  - Each polynomial extends from range \( t = [0,1] \)
    - Keyframes agree at endpoints [C0 continuity]:
      \[
p_i(0) = f_0, \quad p_{n-1}(1) = f_n, \quad p_i(1) = p_{i+1}(0) = f_{i+1}, \quad \forall i = 0, \ldots, n - 2
      \]
  - Tangents agree at endpoints [C1 continuity]:
    \[
p_i'(1) = p_{i+1}'(0), \quad \forall i = 0, \ldots, n - 2
    \]
  - Curvature agrees at endpoints [C2 continuity]:
    \[
p_i''(1) = p_{i+1}''(0), \quad \forall i = 0, \ldots, n - 2
    \]
- Total equations:
  - \( 2n + (n-1) + (n-1) = 4n - 2 \)
- Total DOFs:
  - \( 2n + n + n = 4n \)
- Set curvature at endpoints to 0 and solve
  \[
p_0''(0) = 0, \quad p_{n-1}''(1) = 0
  \]
Natural Splines

- **✓ Interpolation**: by definition
  \[ p_i(0) = f_i, \quad p_i(1) = f_{i+1}, \quad \forall i = 0, \ldots, n - 1 \]

- **✓ C2 Continuity**: by definition
  \[ p''_i(1) = p''_{i+1}(0), \quad \forall i = 0, \ldots, n - 2 \]

- **✗ Locality**: coefficients require us to solve a global linear system
  - Small modification to a keyframe requires resolving the entire system

\[ \begin{align*}
  &\quad y_1, y_2, y_{i-1}, y_i, y_{i+1}, y_{n-1}, y_n, \\
  &\quad x_1, x_2, x_{i-1}, x_i, x_{i+1}, x_{n-1}, x_n
\end{align*} \]
Hermite Splines

- Each cubic piece specified by endpoints and tangents
  - Keyframes set at endpoints:
    \[ p_i(0) = f_i, \quad p_i(1) = f_{i+1}, \quad \forall i = 0, ..., n - 1 \]
  - Tangents set at endpoint:
    \[ p'_i(0) = u_i, \quad p'_i(1) = u_{i+1}, \quad \forall i = 0, ..., n - 1 \]
- Natural splines specify just keyframes
  - Hermite splines specify keyframes and tangents
  - Can get continuity if tangents are set equal
- Total equations:
  - \( 2n + 2n = 4n \)
- Commonly used in vector art programs
  - Illustrator
  - Inkscape
  - SVGs
Hermite vs. Bézier Splines

Hermite curves specify keyframes and tangents, Bezier curves specify control points

Same computation and properties! Just a different interface
Hermite vs. Bézier Splines

Hermite curves specify keyframes and tangents, Bezier curves specify control points.

Same computation and properties! Just a different interface.
Catmull-Rom Splines

• A specialized version of Hermite splines
  • Only need to specify keyframes
  • Tangents computed as:

\[ u_i := \frac{f_{i+1} - f_{i-1}}{t_{i+1} - t_{i-1}} \]

• All the same properties of Hermite splines
• Commonly used to interpolate motion in computer animation
  • When we have tracking data, but not tangent data
  • Easy to generate tangent data
Hermite vs. Bézier vs. Catmull-Rom Splines

- **✓ Interpolation:** by definition
  \[ p_i(0) = f_i, \quad p_i(1) = f_{i+1}, \quad \forall i = 0, \ldots, n - 1 \]

- **✗ Continuity:** Can produce splines that are not C2 (or even C1) continuous
  - Tangents do not need to be same values
    \[ p'_i(0) = u_i, \quad p'_i(1) = u_{i+1}, \quad \forall i = 0, \ldots, n - 1 \]

- **✓ Locality:** each cubic polynomial is generated individually
  - Modifications can happen individually
  - Ease of use make it a prime candidate for vector applications
B-Splines

- Compute a weighted average of nearby keyframes when interpolating

- B-spline basis defined recursively, with base condition:

\[
B_{i,1}(t) := \begin{cases} 
1, & \text{if } t_i \leq t < t_{i+1} \\
0, & \text{otherwise}
\end{cases}
\]

- And inductive condition:

\[
B_{i,k}(t) := \frac{t-t_i}{t_{i+k-1}-t_i} B_{i,k-1}(t) + \frac{t_{i+k}-t}{t_{i+k}-t_{i+1}} B_{i+1,k-1}(t)
\]

- B-spline is a linear combination of bases:

\[
f(t) := \sum_i a_i B_{i,d}
\]
B-Splines

- **Interpolation**: For higher degrees, splines do not pass through keyframes

- **Continuity**: With higher degrees, bases are twice differentiable

\[
B_{i,k}(t) := \frac{t-t_i}{t_{i+k-1}-t_i} B_{i,k-1}(t) + \frac{t_{i+k}-t}{t_{i+k}-t_{i+1}} B_{i+1,k-1}(t)
\]

- **Locality**: B-spline bases are a function of the current and next bases

\[
B_{i,k}(t) := \frac{t-t_i}{t_{i+k-1}-t_i} B_{i,k-1}(t) + \frac{t_{i+k}-t}{t_{i+k}-t_{i+1}} B_{i+1,k-1}(t)
\]
B-Splines

- **Incorrect Interpolation:** For higher degrees, splines do not pass through keyframes

- **Correct Continuity:** With higher degrees, bases are twice differentiable

\[
B_{i,k}(t) := \frac{t-t_i}{t_{i+k-1}-t_i} B_{i,k-1}(t) + \frac{t_{i+k}-t}{t_{i+k-1}-t_{i+1}} B_{i+1,k-1}(t)
\]

- **Correct Locality:** B-spline bases are a function of the current and next bases

\[
B_{i,k}(t) := \frac{t-t_i}{t_{i+k-1}-t_i} B_{i,k-1}(t) + \frac{t_{i+k}-t}{t_{i+k-1}-t_{i+1}} B_{i+1,k-1}(t)
\]
**Splines Review**

<table>
<thead>
<tr>
<th></th>
<th>[ Interpolation ]</th>
<th>[ Continuity ]</th>
<th>[ Locality ]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>Natural</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>Hermite</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>Bezier</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>Catmull-Rom</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>B-Spline</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
### Splines Review

<table>
<thead>
<tr>
<th></th>
<th>[ Interpolation ]</th>
<th>[ Continuity ]</th>
<th>[ Locality ]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td>Natural</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td>Hermite</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td>Bezier</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td>Catmull-Rom</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td>B-Spline</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

**NO PERFECT SPLINE!**
• Splines

• Forward Kinematics

• Inverse Kinematics
We saw the rendering equation,  
But what is the animation equation?
The Animation Equation

\[ F = ma \]

It’s a little more complicated than just this...
An Animation System

- Component of an animation system:
  - Object’s mass
  - Object’s configuration
  - Object’s velocity
  - Object’s acceleration
  - Forces acting on object
  - Set of constraints

- Configuration $q(t)$ is time dependent
  - Can use splines to interpolate control points (keyframes)

\[
\begin{align*}
\dot{q} &:= \frac{d}{dt} q \\
g(q, \dot{q}, t) &= 0 \\
\ddot{q} &= \frac{F}{m}
\end{align*}
\]
An Animation System

- Common to describe system with many moving pieces
  - Ex: a collection of billiard balls
  - Can collect into a single configuration:
    \[ q = (x_0, x_1, \ldots, x_n) \]
- Naturally maps to the way we actually solve equations on a computer
  - All variables stacked into a vector and handed to a solver
Character Animation

• Configuration of a character is the configuration of all their individual joints

• Keyframes save poses of characters
  • **Goal:** use splines to interpolate between poses of a character
    • Natural splines
    • Hermite splines
    • B-splines

• **Problem:** what is an efficient, user-friendly way of setting character poses?
Motion Capture

- Just take videos of real life poses
  - Map to character model

- Data can get very messy
  - Same idea as capturing a point cloud

- [+] Easy to understand
- [+] Capture real-life poses
- [-] Expensive to purchase
- [-] Very noisy data
- [-] Requires a lot of cleanup

The Hobbit (2012) Peter Jackson
The Human Rig

• Many systems well-described by a kinematic chain
  • Collection of rigid bodies, connected by joints
  • Joints have various behaviors
    • Ball (shoulder)
    • Piston (neck)
    • Hinge (elbow)
  • Also have constraints (e.g., range of angles)
    • Human neck can’t rotate around fully
    • Owl necks can!
  • Hierarchical structure (body → leg → foot)

• In animation, often called a character rig
  • Character rigs are scene graphs!
Character Rigging

- Character rigging is a separate job from character modeling and character animation
  - Focuses on:
    - Optimal joint placement
    - Joint angle extent
    - Joint hierarchy

- Not all human rigs are the same!
  - Depends on character model proportions/movements
How do we animate a rig?
Forward Kinematics

- Consider moving the hand $c_2$
  - Rotate shoulder (moves $c_1$ and $c_2$)
  - Then rotate elbow (moves $c_2$)

- New elbow position $p_1$ computed as:
  \[ p_1 = p_0 + \begin{bmatrix} \cos \theta_0 & \sin \theta_0 \\ -\sin \theta_0 & \cos \theta_0 \end{bmatrix} u_0 \]
  - Can also be written as:
    \[ p_1 = p_0 + e^{i\theta_0} u_0 \]

- New hand position $p_2$ computed as:
  \[ p_2 = p_0 + e^{i\theta_0} u_0 + e^{i\theta_0} e^{i\theta_1} u_1 \]
Forward Kinematics

• Consider moving the hand $c_2$
  • Rotate shoulder (moves $c_1$ and $c_2$)
  • Then rotate elbow (moves $c_2$)

• Can also be written as a series of rotations and translations:

$$p_2 = T(u_1) R(\theta_1) T(u_0) R(\theta_0) p_0$$
A Note About Spaces

- **World Space**: absolute coordinate space
- **Local Space**: the model’s space
  - Often use the rig’s center as the origin
- **Bone Space**: For a given bone $i$, the origin is the bone’s base point and the axes are rotated by its rotations and all the parent rotations before it
  - **Bind Space**: a form of Bone Space, but no rotations, just translations
    - Think of Bind Space as the model in T-pose position with no rotations applied, just the offsets
      \[ c_2 = T(u_1) T(u_0) c_0 \]
  - **Pose Space**: a form of Bone Space, with both rotations and translations applied
    - Think of it as the model that is posed with rotations
      \[ p_2 = T(u_1) R(\theta_1) T(u_0) R(\theta_0) p_0 \]
A Note About Spaces

• In Scotty3D, we give you points in either Bind or Pose Space, and you need to compute the transformation to Local Space
  • Just involves flipping the order of matrices
    • Bind-to-Local:
      \[ c_0 = T(u_0) \cdot T(u_1) \cdot c_2 \]
    • Pose-to-Local:
      \[ p_0 = R(\theta_0) \cdot T(u_0) \cdot R(\theta_1) \cdot T(u_1) \cdot R(\theta_2) \cdot p_2 \]
  
• Rotations and transformations will be saved as child-to-parent
  • No need to invert

\[ \frac{15}{462/662} \]
Forward Kinematics

• [+] Computationally efficient
• [+] Easy interface to work with
• [+] Explicit control over every joint
• [-] Produces rigid animations
• [-] Hard to model real-world motions
• [-] Requires more keyframes

• Results often look robot-like

Big Hero 6 (2014) Disney
Linear Blend Skinning

- Vertices track with bones
  - Known as blend skinning

- For each vertex $i$, compute weights $w_{ij}$ for each bone $j$
  - Weights are normalized for each vertex
    \[
    \sum_j w_{ij} = 1
    \]

- Weights average transforms of each bone to compute posed vertex position $v'_i$ from bind vertex $v_i$
  \[
  v'_i = \sum_j (w_{ij} P_j B_j^{-1}) v_i
  \]
  - $P_j$ is bone $j$’s bone-to-pose transform
  - $B_j$ is bone $j$’s bone-to-bind transform
    - It should type-check : )

Monster’s Inc (2001) Pixar
Computing Weights

- $r$ is the radius of the bone
- $d_{ij}$ is the distance between $v_i$ and its closest projection onto the bone

$$\hat{w}_{ij} = \max(0, r - d_{ij})$$

- Make sure to normalize weights

$$w_{ij} = \frac{\hat{w}_{ij}}{\sum_j \hat{w}_{ij}}$$

Why do we need $r$?
Review: Closest Point on a Line Segment

Compute the vector $\mathbf{p}$ from the line base $\mathbf{a}$ along the line

$$\langle \mathbf{p} - \mathbf{a}, \mathbf{b} - \mathbf{a} \rangle$$

Normalize to get a time

$$t = \frac{\langle \mathbf{p} - \mathbf{a}, \mathbf{b} - \mathbf{a} \rangle}{\langle \mathbf{b} - \mathbf{a}, \mathbf{b} - \mathbf{a} \rangle}$$

Clip time to range $[0,1]$ and interpolate

$$\mathbf{a} + (\mathbf{b} - \mathbf{a})t$$
Weight Painting

- Computer animation applications allow you to specify weights on your own
  - Known as **weight painting**
- UI uses color to illustrate magnitude of each vertex/bone pair
- Part of the rigging pipeline

Blender (2021) Ton Roosendaal
• Splines

• Forward Kinematics

• Inverse Kinematics
How Humans Move

• We don’t think about the movement of each individual joint
  • Instead, we think about a part of our body, and where we want it to go
    • Our body solves for the correct movements
    • Ex: hand moves to reach a doorknob

• No unique solution
  • Many ways to catch a ball

• What if our rig behaved a similar way...
Inverse Kinematics

• Identify a bone on the rig \( i \) and a handle \( h \) that it should reach for
  • Can try to satisfy multiple targets \((i, h)\)

• Loss function \( f(q) \) for rig configuration \( q \) is:

\[
f(q) = \sum_{(i,h)} \frac{1}{2} |p_i(q) - h|^2
\]

  • Where \( p_i(q) \) is the position of the end of bone \( i \)

• **Goal:** compute the gradient \( \nabla f(q) \)
  • Gradient represents how changing each joint will change the loss function
  • Apply gradient descent with some timestep \( \tau \):

\[
q = q - \tau \nabla f(q)
\]
Inverse Kinematic Gradient

\[
\frac{df}{d\theta_k} = \frac{d}{d\theta_k} \sum_{(i,h)} \frac{1}{2} |p_i(q) - h|^2
\]

Take gradient with respect to function

\[
\frac{df}{d\theta_k} = \sum_{(i,h)} (p_i(q) - h) \frac{dp_i}{d\theta_k}
\]

Expand \( p_i \) into transformations. Each rotation in 3D is axis-aligned

\[
\frac{dp_i}{d\theta_k} = \frac{d}{d\theta_k} \left[ \prod_{j=0,i-1} R(\theta_j^z) R(\theta_j^y) R(\theta_j^x) T(u_j) \right] R(\theta_i^z) R(\theta_i^y) R(\theta_i^x) u_i
\]

Gradient breaks down into 3 parts:

\[
\frac{dp_i}{d\theta_k} = [ \text{linear transformation} ] \frac{d}{d\theta_k} [ \text{derivative} ] [ \text{transformed point} ]
\]
Inverse Kinematic Gradient

\[ \frac{dp_i}{d\theta^y_k} = ??? \]

**Fun fact:** by transforming the axis of rotation and base point to local coordinates, then the derivative of the rotation \( R(\theta^y_k) \) by amount \( \theta^y_k \) around axis \( y \) and center \( r \) of point \( p \) becomes:

\[ \frac{dp_i}{d\theta^y_k} = y \times (p - r) \]

\( p = \text{[ linear transformation]} [R(\theta^y_k)] \text{[ transformed point]} \)

\( r = \text{[ linear transformation’]} [0,0,0] \)

\( y = ([\text{ linear transformation’}] [R(\theta^z_k)]) \cdot \text{rotate}(\theta^y_k) \)

\([\text{ linear transformation’}]=\text{all rotations and transformations up to, but not including the kth bone}\)
Inverse Kinematic Gradient

• Note: all joints that come before joint $k$ can also contribute to the movement of joint $k$
  • **Example:** moving your shoulder moves your hand

• Need to also compute how every joint prior to joint $k$ affects the movement of joint $k$
  • Gives us a gradient for each joint in range [0 - $k$]

\[
\nabla f_k^y = (p_i(q) - h) \cdot [y_k \times (p_i(q) - r_k)] \\
\nabla f_{k-1}^y = (p_i(q) - h) \cdot [y_{k-1} \times (p_i(q) - r_{k-1})] \\
\nabla f_{k-2}^y = (p_i(q) - h) \cdot [y_{k-2} \times (p_i(q) - r_{k-2})] \\
... \\
\nabla f_0^y = (p_i(q) - h) \cdot [y_0 \times (p_i(q) - r_0)]
\n\]
Inverse Kinematic Gradient

- Each joint $k$ will have its own vector gradient
  \[ \frac{df}{d\theta_k} = \langle \frac{df}{d\theta_k^x}, \frac{df}{d\theta_k^y}, \frac{df}{d\theta_k^z} \rangle \]
  - Same process for computing each component, just use $x_k$, $y_k$, or $z_k$
- What if we have multiple target pairs $(i, h)$?
  - Gradient becomes a sum!

\[ \nabla f_k^y = (p_i(q) - h) \cdot [y_k \times (p_i(q) - r_k)] \]
\[ \nabla f_{k-1}^y = (p_i(q) - h) \cdot [y_{k-1} \times (p_i(q) - r_{k-1})] \]
\[ \nabla f_{k-2}^y = (p_i(q) - h) \cdot [y_{k-2} \times (p_i(q) - r_{k-2})] \]
\[ \vdots \]
\[ \nabla f_0^y = (p_i(q) - h) \cdot [y_0 \times (p_i(q) - r_0)] \]
Inverse Kinematic Gradient

```cpp
vec3 gradient_in_current_pose() {
    for (auto &handle : handles) {
        Vec3 h = handle.target;
        Vec3 p = // TODO: compute output point

        // walk up the kinematic chain
        for (BoneIndex b = handle.bone; b < bones.size(); b = bones[b].parent) {
            Bone const &bone = bones[b];
            Mat4 xf = // TODO: compute [linear transform’]

            Vec3 r = xf * Vec3{0.0f, 0.0f, 0.0f};

            Vec3 x = // TODO: compute bone’s x-axis in local space
            Vec3 y = // TODO: compute bone’s y-axis in local space
            Vec3 z = // TODO: compute bone’s z-axis in local space

            gradient[b].x += dot(cross(x, p - r), p - h);
            gradient[b].y += dot(cross(y, p - r), p - h);
            gradient[b].z += dot(cross(z, p - r), p - h);
        }
    }
}
```
Inverse Kinematic Gradient

• How do we apply the gradient?
  • Iterate through each joint $j$ and apply $\nabla f_j$
  • Make sure to clear all gradients after each step!
    $$\theta_j = \theta_j - \tau \nabla f_j$$

• Recompute the loss function
    $$f(q) = \sum_{(i,h)} \frac{1}{2} |p_i(q) - h|^2$$

• If loss is lower than some threshold, terminate
  • Otherwise continue until max steps exceeded

my optimizer

HE MAKE AN BIG STEPPY