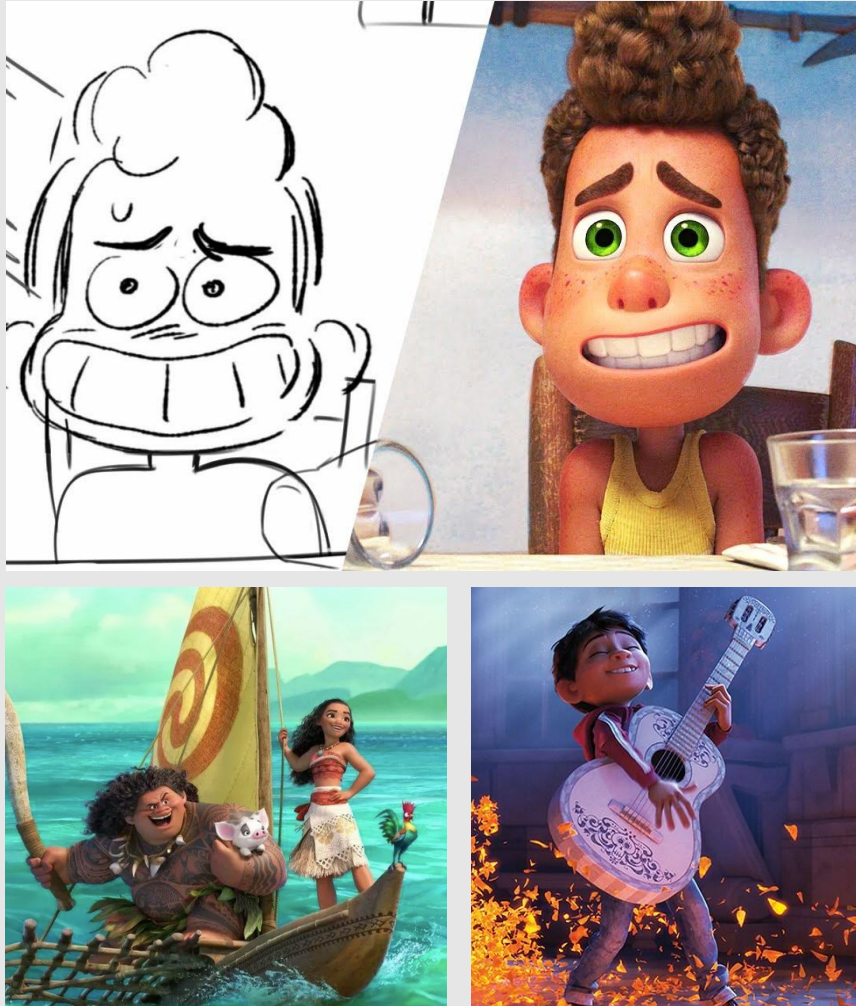


Splines & Kinematics

- Splines
- Forward Kinematics
- Inverse Kinematics

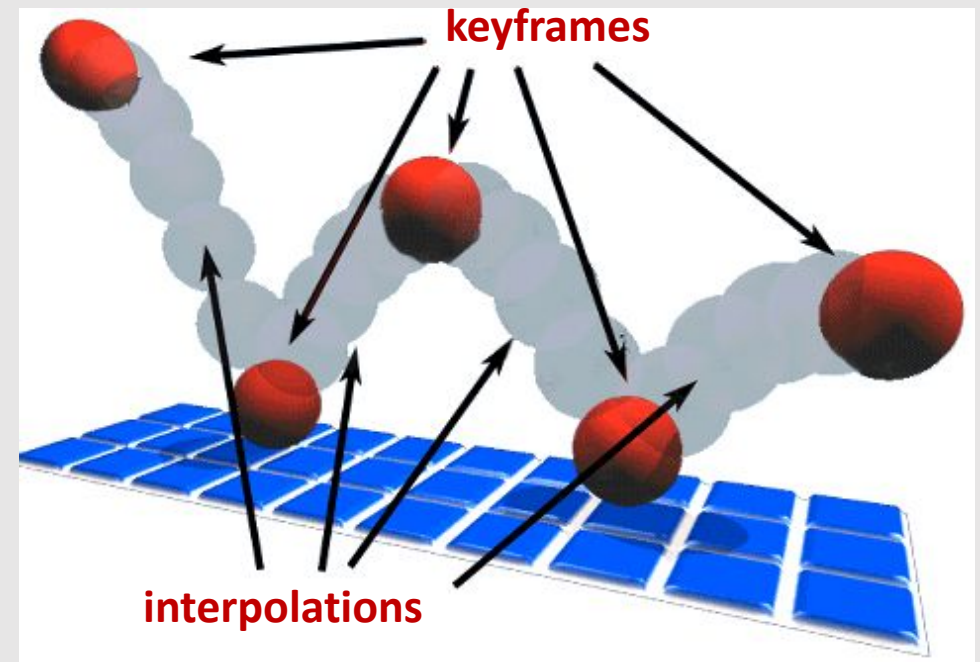
Recall: 3D Animation



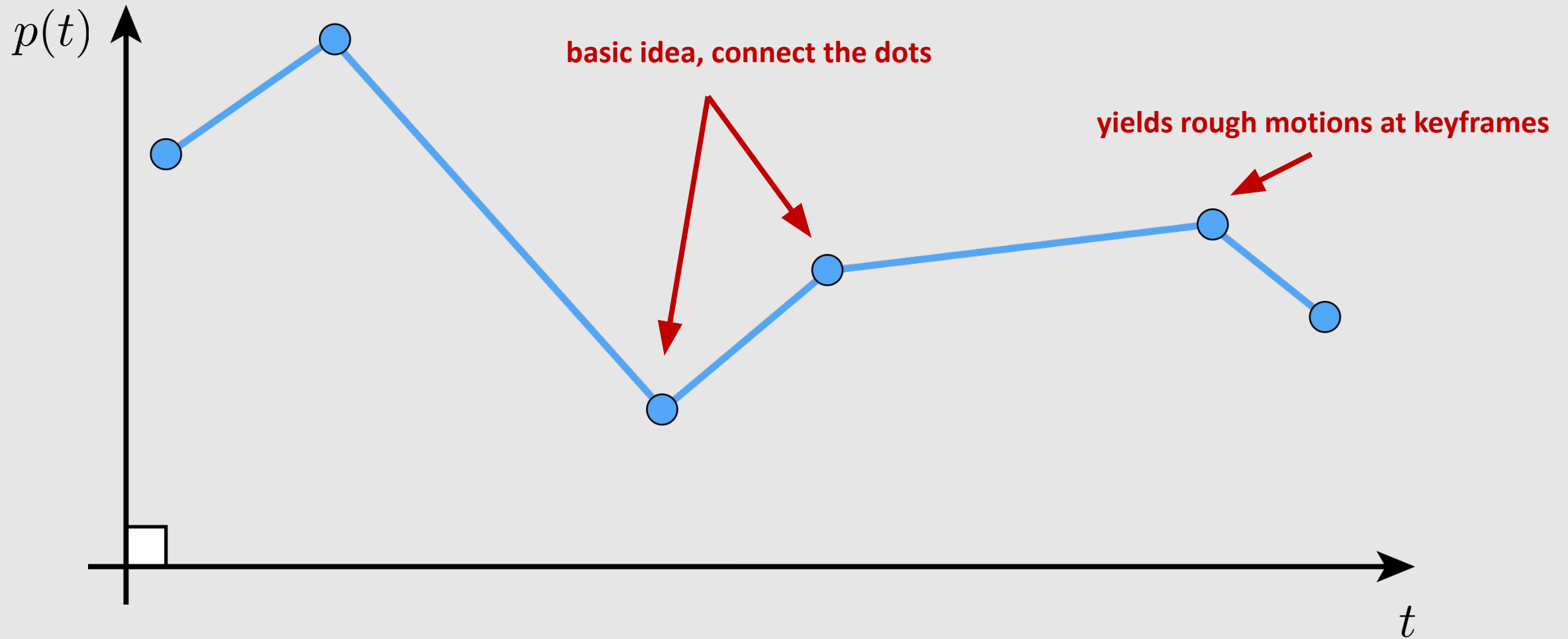
- Using meshes, materials, and rendering to produce 3D animated sequences
- Use a photorealistic renderer to make results photorealistic
- **Today:** No need to draw anything, computer takes care of everything
 - Set keyframes by hand
 - **Forward Kinematics**
 - **Inverse Kinematics**
 - Allow keyframes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

Keyframing

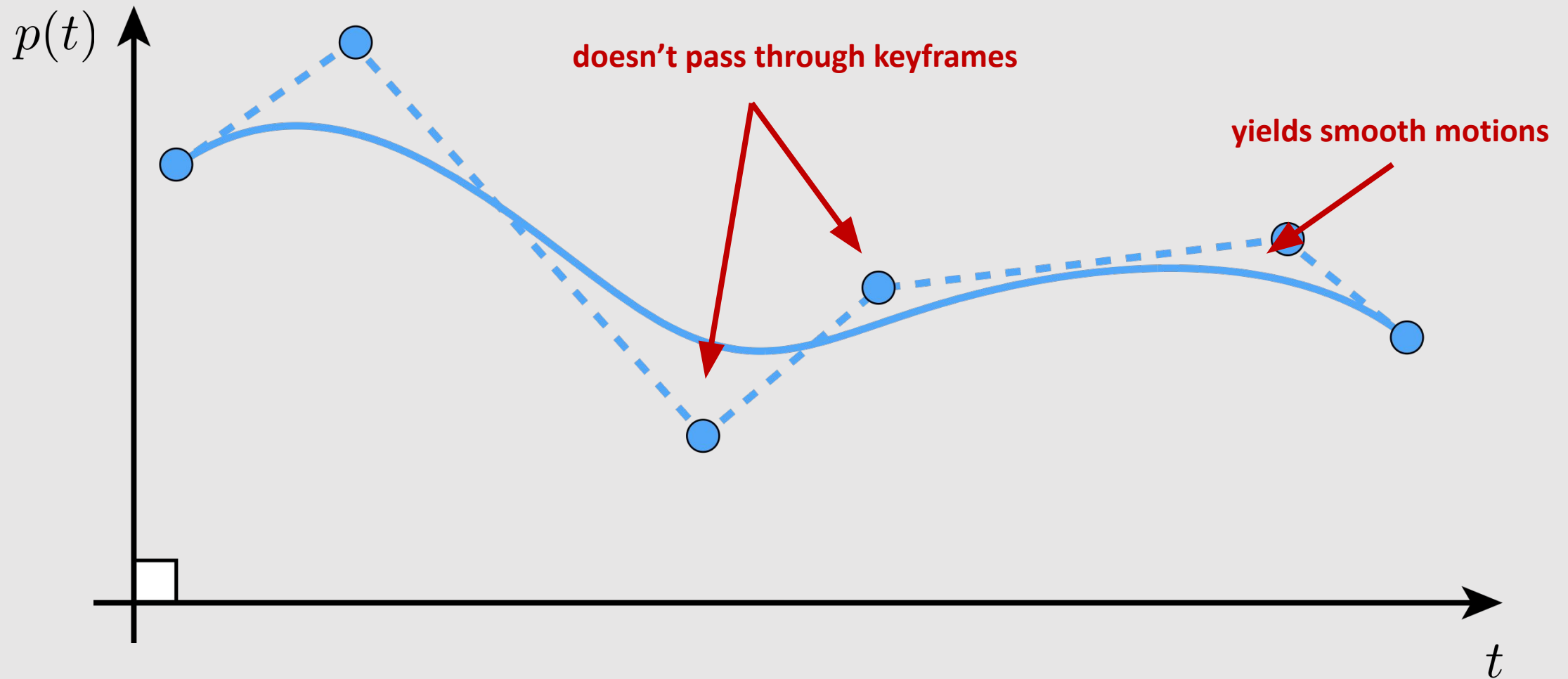
- Set keyframes at important locations in the animation
 - Have the computer interpolate the rest
- Can keyframe anything!
 - Color
 - Light intensity
 - Camera zoom
- **Problem:** how should data interpolate?
 - Linearly?
 - Along a curve/arc?



Linear Interpolation

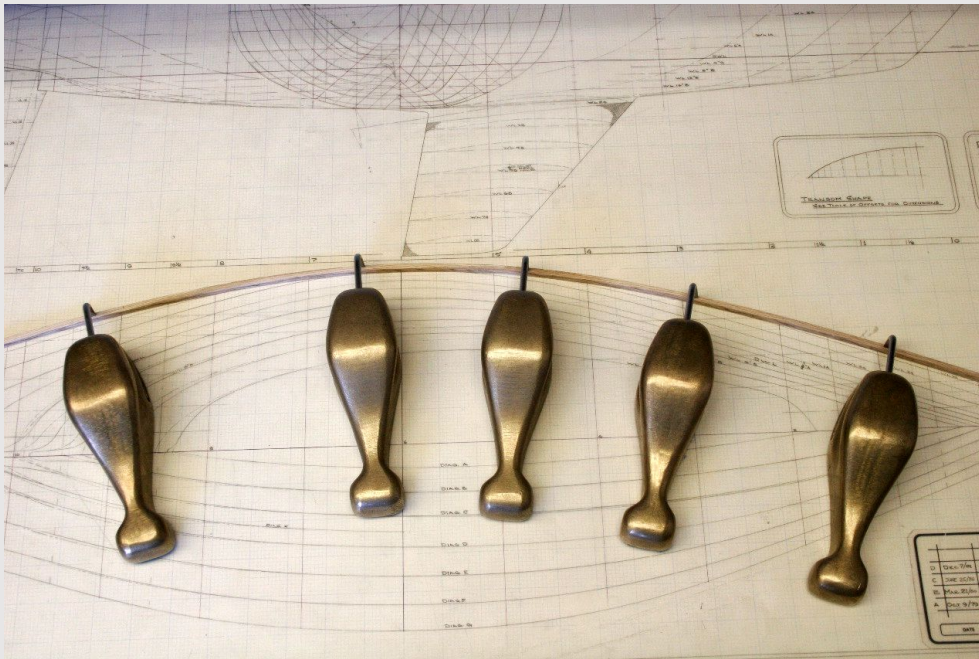


Piecewise Polynomial Interpolation

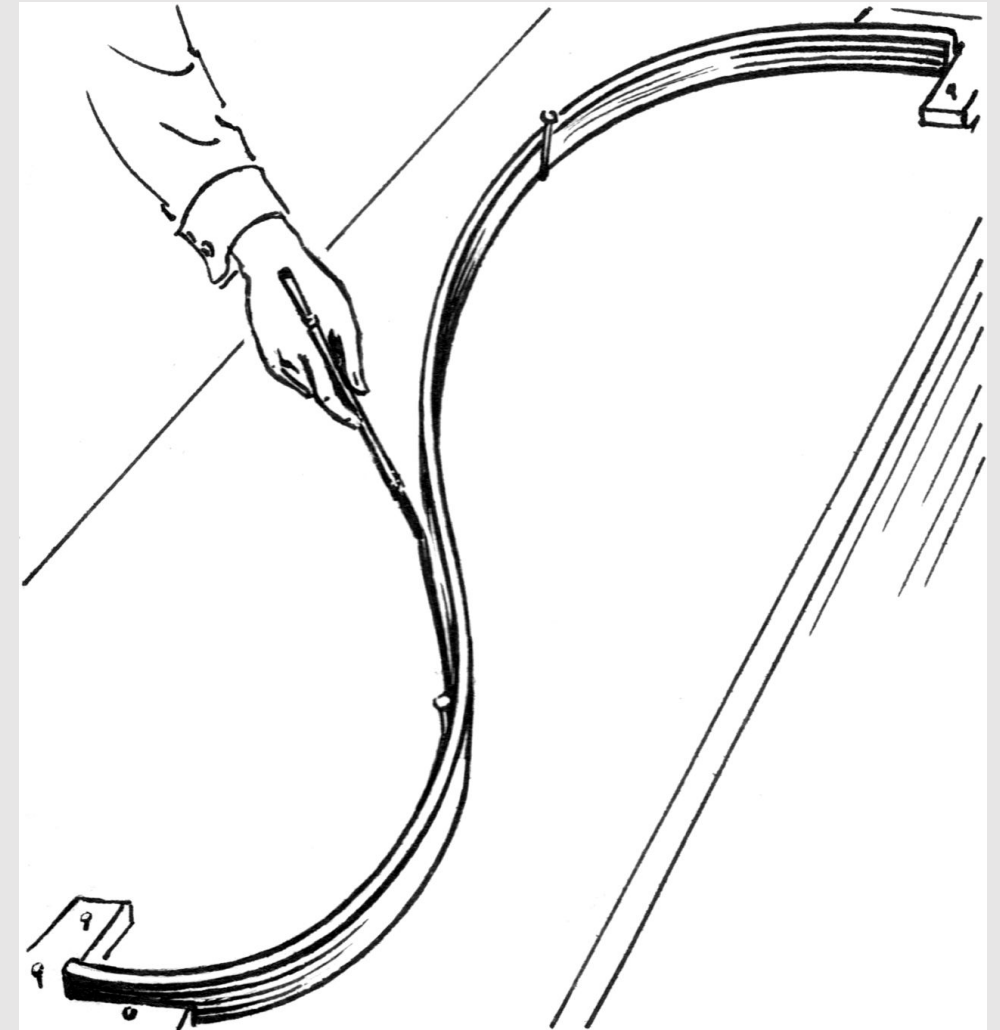


Splines

- Mathematical theory of interpolation arose from study of thin strips of wood or metal (“splines”) under various forces
 - **Splines** define interpolation



The Elastica: A Mathematical History (2008) Levin



Splines

- In this course, a spline is any piecewise cubic polynomial function

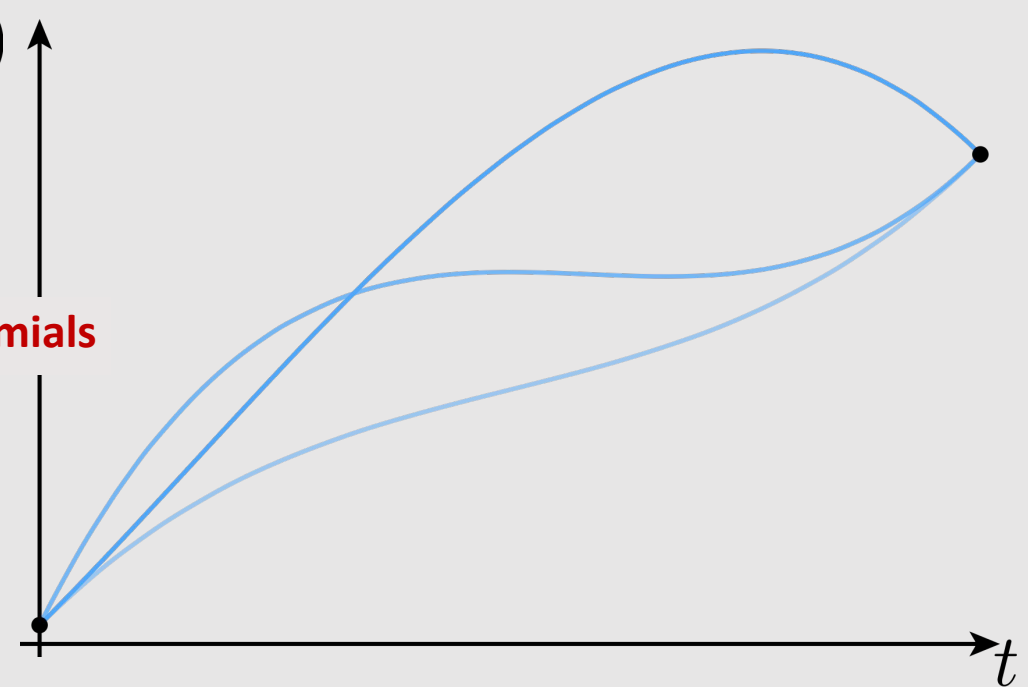
$$\text{for } t_i \leq t \leq t_{i+1}, f(t) = \sum_{j=1}^d c_i t^j =: p_i(t)$$

degree (points to d)
coefficients (points to c_i)
polynomials (points to $p_i(t)$)

- Common spline:** cubic polynomial:

$$p(t) = at^3 + bt^2 + ct + d$$

- 3 goals of splines:
 - Interpolation**
 - Continuity**
 - Locality**



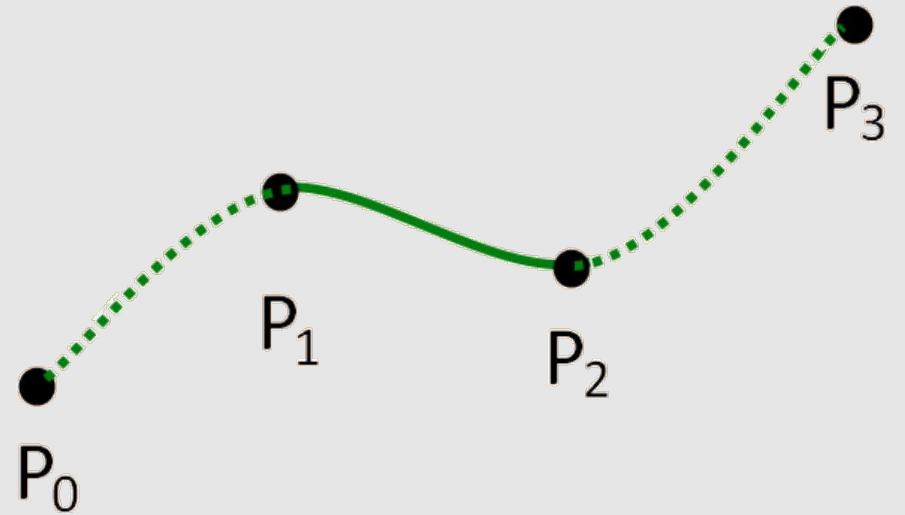
Many solutions!

Interpolation

- **Interpolation:** does the spline pass through the control points

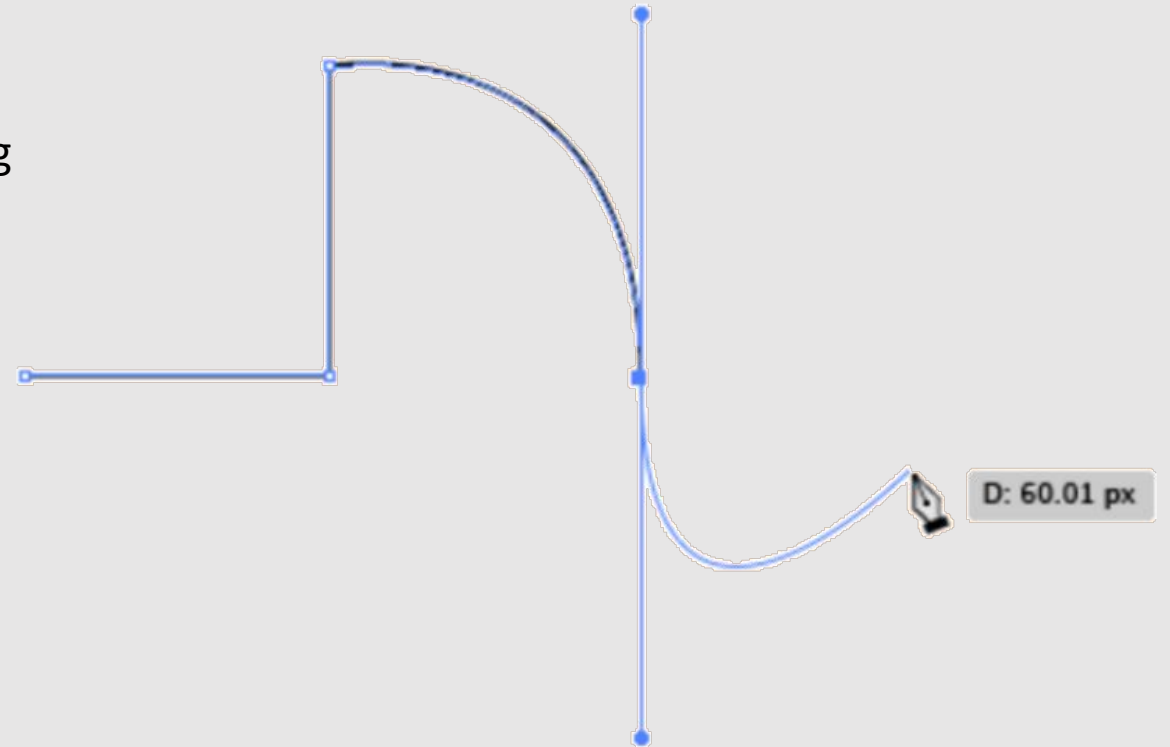
$$f(t_i) = f_i \quad \forall i$$

- For every keyframe f_i , there exists some time t_i where the interpolation of f equals the keyframe f_i



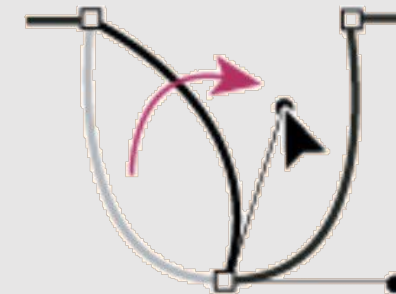
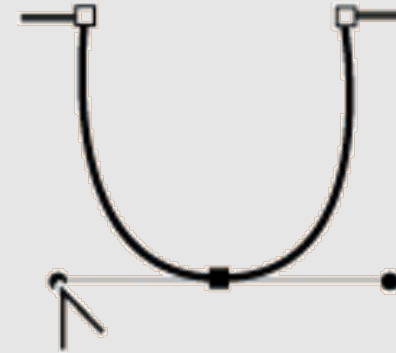
Continuity

- **Continuity:** Is the spline twice differentiable along control points [C2 continuity]
- 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 [C0 continuity]



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

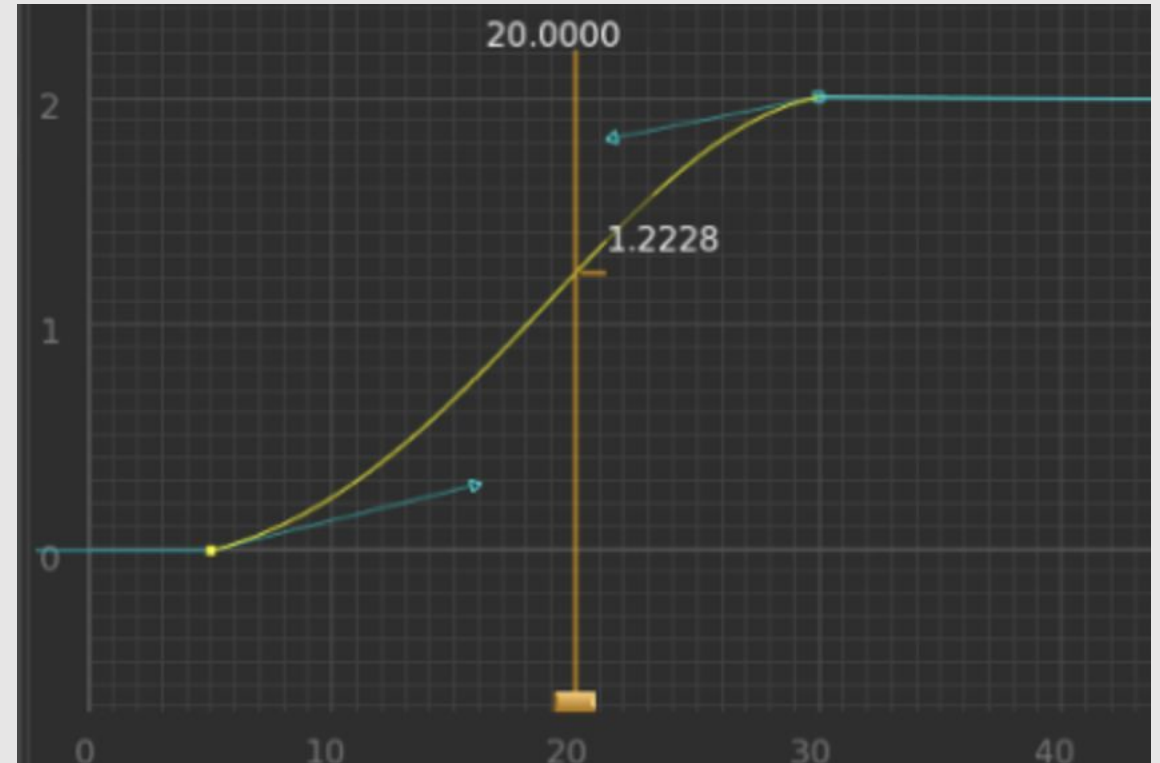
$$p(0) = p_0 \quad \Rightarrow d = p_0$$

$$p(1) = p_1 \quad \Rightarrow a + b + c + d = p_1$$

$$p'(0) = u_0 \quad \Rightarrow c = u_0$$

$$p'(1) = u_1 \quad \Rightarrow 3a + 2b + c = u_1$$

- Gives us 4 constraints
 - Can be turned into 4 coefficients



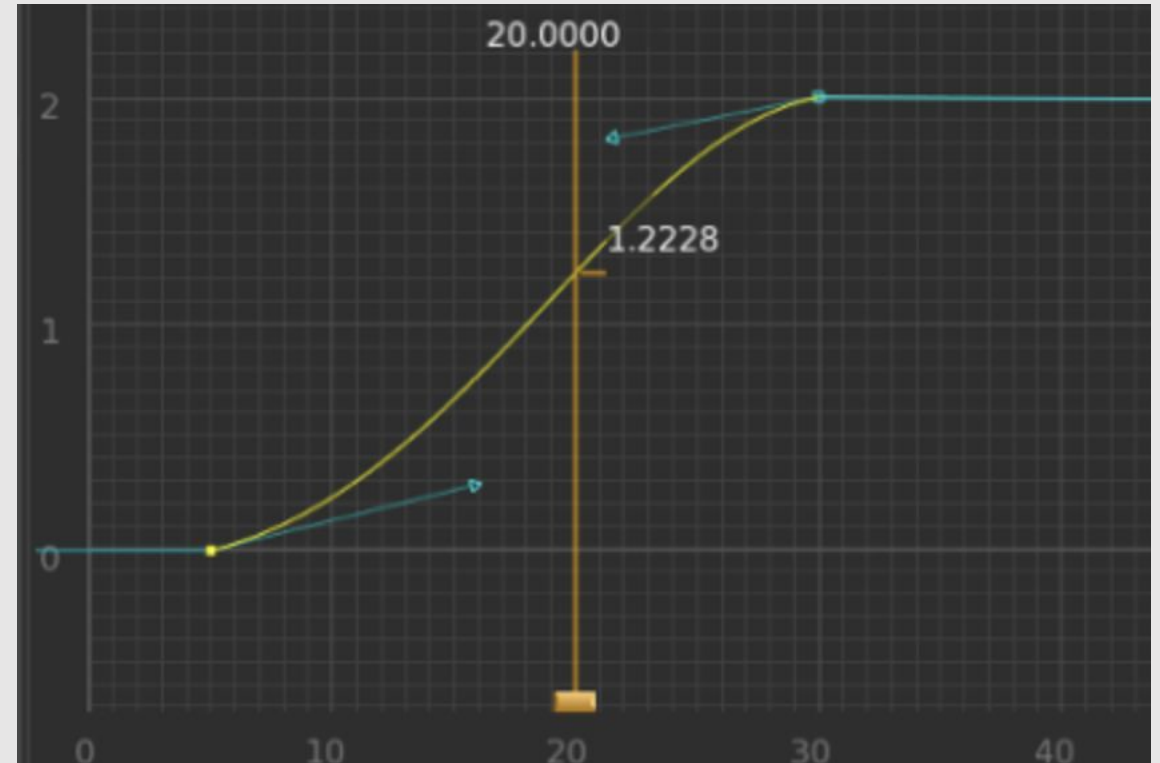
Piecewise Cubic Polynomial

- Can also write:

$$p(t) = at^3 + bt^2 + ct + d$$

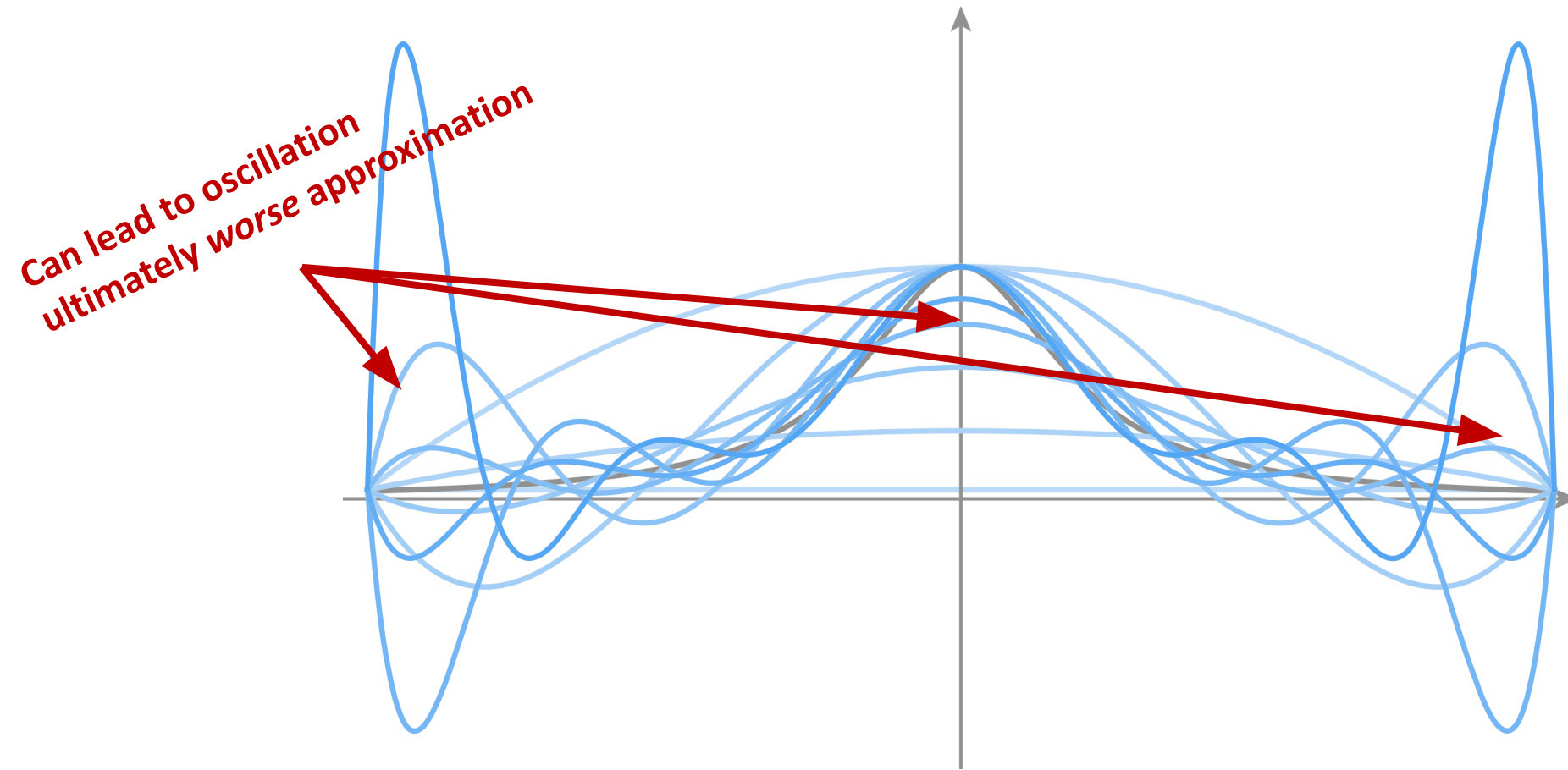
- As a linear system!

$$\begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 3 & 2 & 1 & 0 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} = \begin{bmatrix} p_0 \\ p_1 \\ u_0 \\ u_1 \end{bmatrix}$$



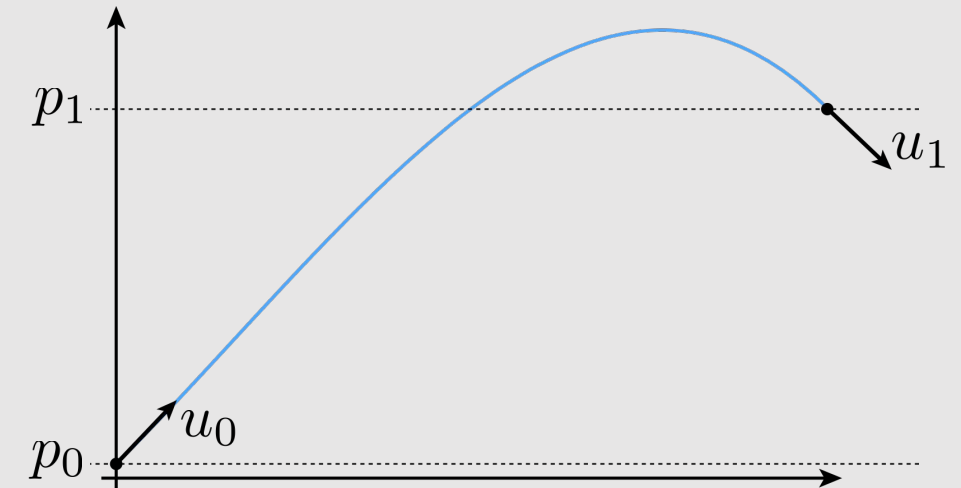
Runge Phenomenon

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



Natural Splines

- Build a spline out of piecewise cubic polynomials p_i
 - Each polynomial has range $t = [0,1]$
 - Keyframes agree at endpoints [C0 continuity]:
$$p_i(0) = f_i, \quad p_i(1) = f_{i+1}, \quad \forall i = 0, \dots, n-1$$
 - Tangents agree at endpoints [C1 continuity]:
$$p'_i(1) = p'_{i+1}(0), \quad \forall i = 0, \dots, n-2$$
 - Curvature agrees at endpoints [C2 continuity]:
$$p''_i(1) = p''_{i+1}(0), \quad \forall i = 0, \dots, n-2$$



- Total equations:
 - $2n + (n-1) + (n-1) = 4n - 2$
- Total DOFs:
 - $2n + n + n = 4n$
- Set curvature at endpoints to 0 and solve
$$p'_0(0) = 0, \quad p''_0(0) = 0$$

Natural Splines

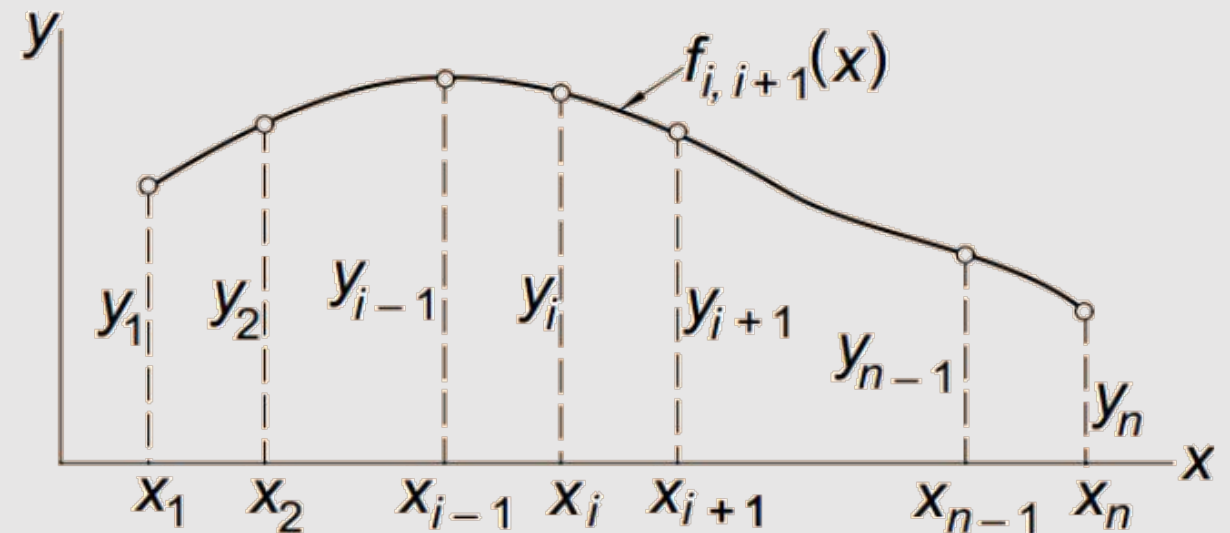
- ✓ **Interpolation:** by definition

$$p_i(0) = f_i, \quad p_i(1) = f_{i+1}, \quad \forall i = 0, \dots, n-1$$

- ✓ **Continuity:** by definition

$$p''_i(1) = p''_{i+1}(0), \quad \forall i = 0, \dots, n-2$$

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



Hermite/Bézier Splines

- Each cubic piece specified by endpoints and tangents

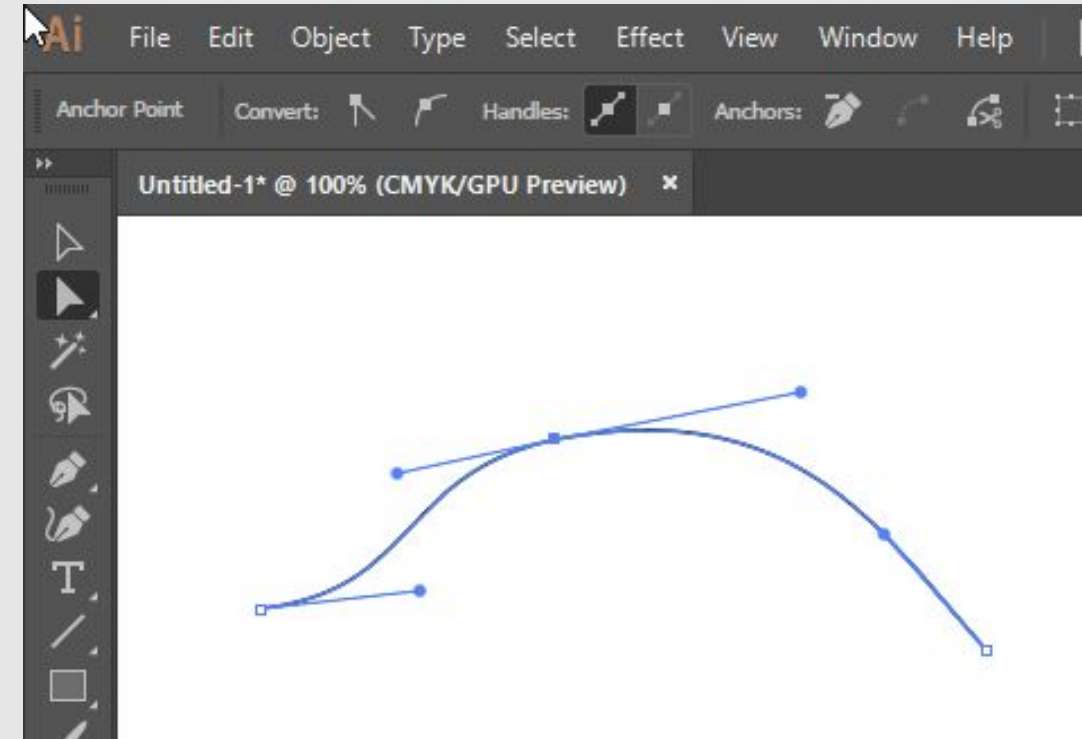
- Keyframes set at endpoints:

$$p_i(0) = f_i, \quad p_i(1) = f_{i+1}, \quad \forall i = 0, \dots, n-1$$

- Tangents set at endpoint:

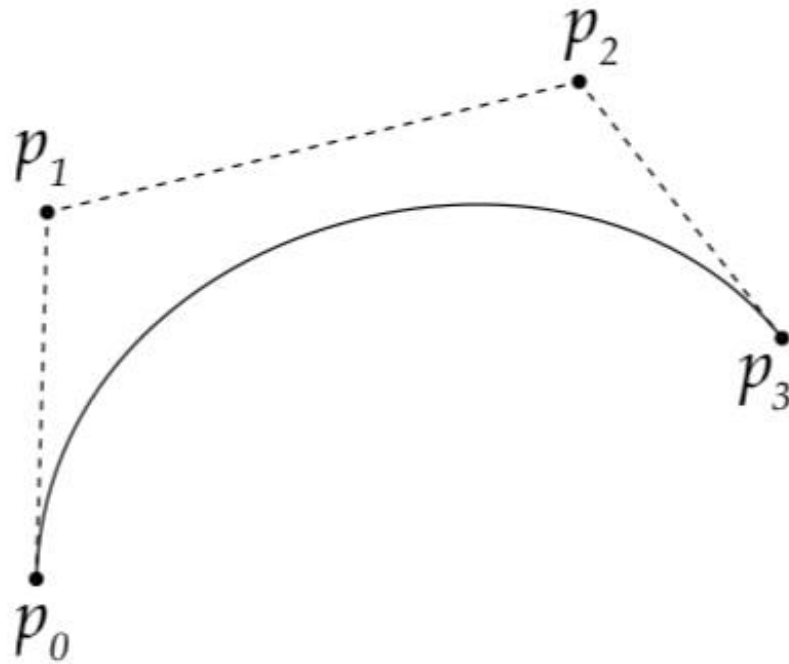
$$p'_i(0) = u_i, \quad p'_i(1) = u_{i+1}, \quad \forall i = 0, \dots, n-1$$

- Bézier splines specify keyframes and tangents
 - Natural splines specify just keyframes
 - Can get continuity if tangents are set equal
- Total equations:
 - $2n + 2n = 4n$
- Commonly used in vector art programs
 - Illustrator
 - Inkscape
 - SVGs

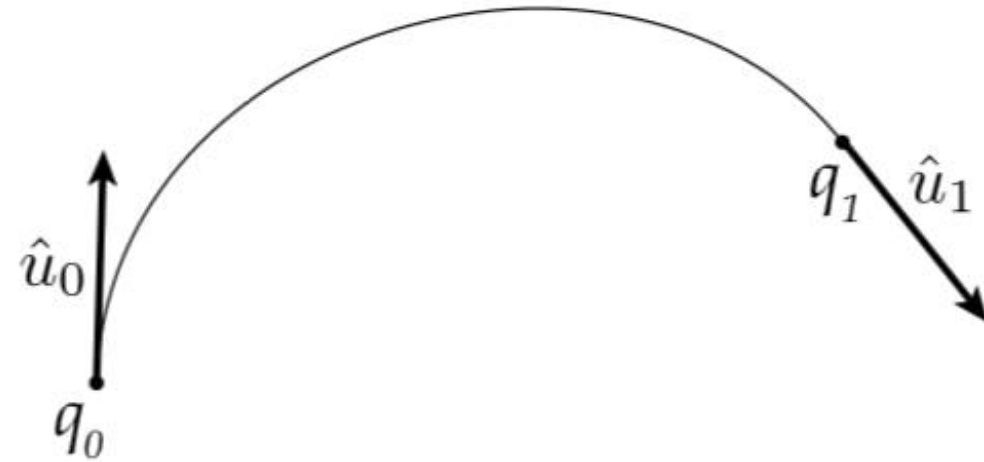


Hermite/Bézier Splines

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



cubic Bézier

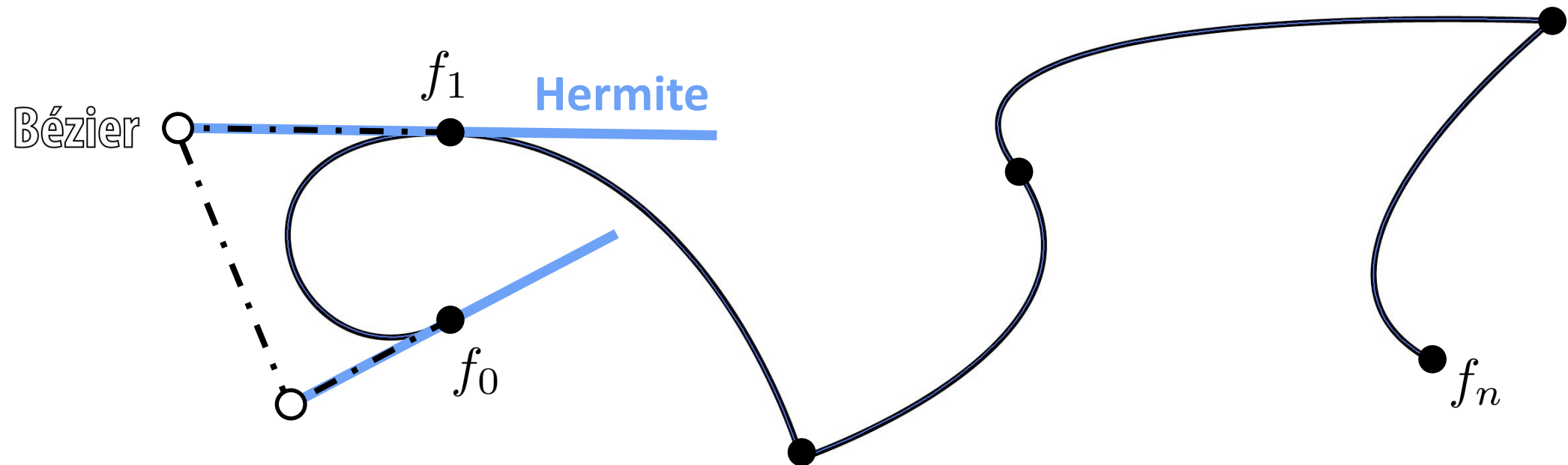


cubic Hermite

Same computation and properties! Just a different interface

Hermite/Bézier Splines

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



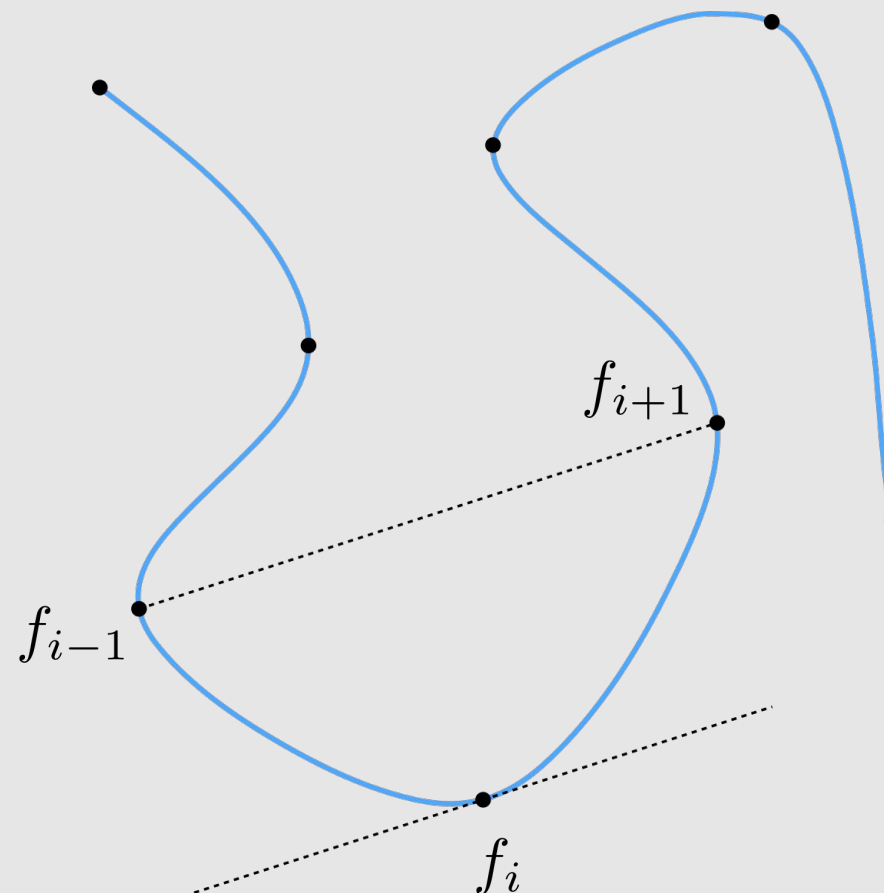
Same computation and properties! Just a different interface

Catmull-Rom Splines

- A specialized version of Hermite splines
 - Only 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 in motion capture
 - When we have tracking data, but no tangent data
 - Easy to generate tangent data



Catmull-Rom Splines

- Construct tangents via finite differences:

$$u_1 = \frac{p_2 - p_0}{t_2 - t_0} \quad u_2 = \frac{p_3 - p_1}{t_3 - t_1}$$

- Perform cubic interpolation on splines:

$$p(t) = h_{00}(t) * p_1 + h_{10}(t) * u_1 + h_{01}(t) * p_2 + h_{11}(t) * u_2$$

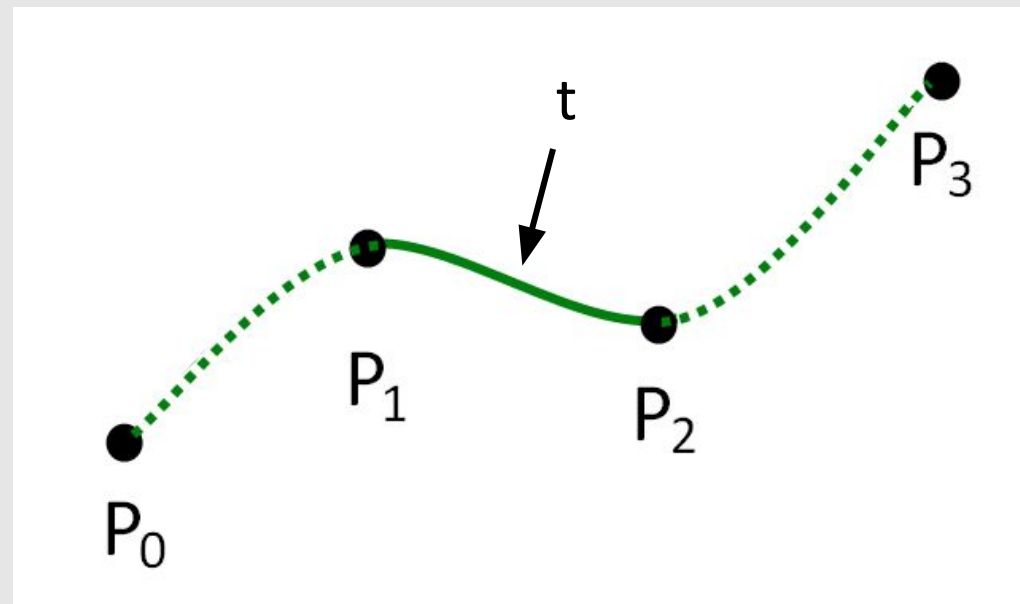
$$h_{00}(t) = 2t^3 - 3t^2 + 1$$

$$h_{10}(t) = t^3 - 2t^2 + t$$

$$h_{01}(t) = -2t^3 + 3t^2$$

$$h_{11}(t) = t^3 - t^2$$

Cubic interpolation is piecewise!
Must normalize t to range [0,1]



Catmull-Rom Splines

- Need to normalize time

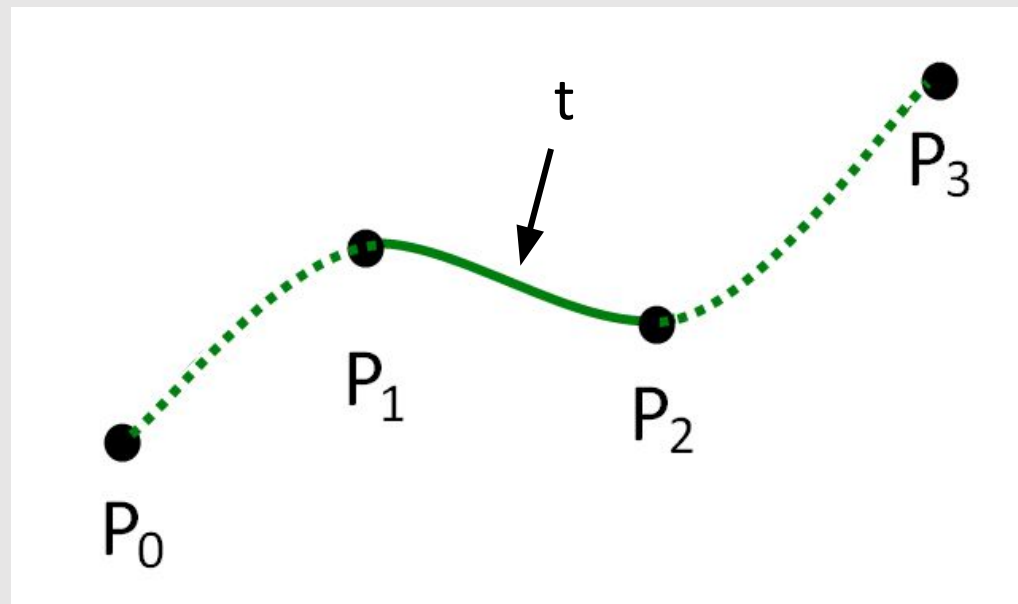
$$t_{norm} = \frac{t - t_1}{t_2 - t_1}$$

- Need to normalize tangents as well

$$u_1 = \frac{p_2 - p_0}{\frac{t_2 - t_1}{t_2 - t_1} - \frac{t_0 - t_1}{t_2 - t_1}}$$

$$u_1 = \frac{p_2 - p_0}{(t_2 - t_1) - (t_0 - t_1)} * (t_2 - t_1)$$

$$u_1 = \frac{p_2 - p_0}{t_2 - t_0} * (t_2 - t_1)$$



Hermite/Bézier/Catmull-Rom Splines

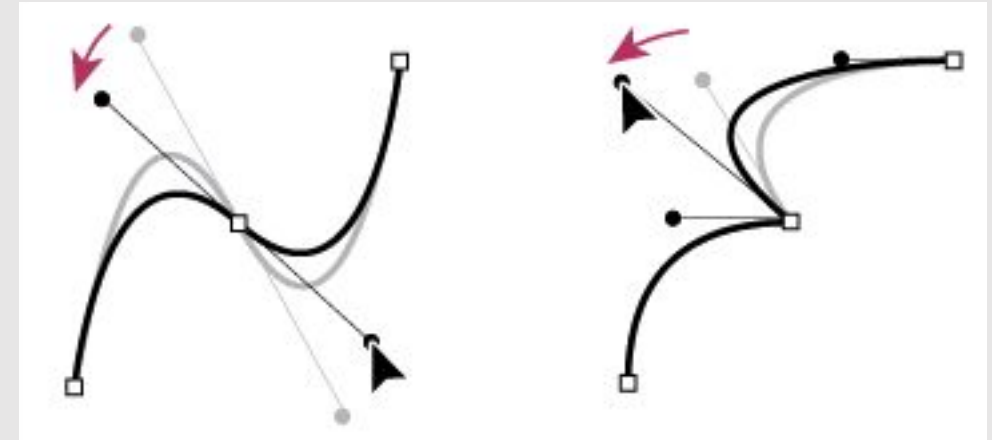
- **✓ Interpolation:** by definition

$$p_i(0) = f_i, \quad p_i(1) = f_{i+1}, \quad \forall i = 0, \dots, n-1$$

- **✗ Continuity:** Can produce splines that are not C2 (or even C1) continuous
 - Tangents do not need to be same values

$$p'_i(0) = u_i, \quad p'_i(1) = u_{i+1}, \quad \forall i = 0, \dots, n-1$$

- **✓ 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:

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

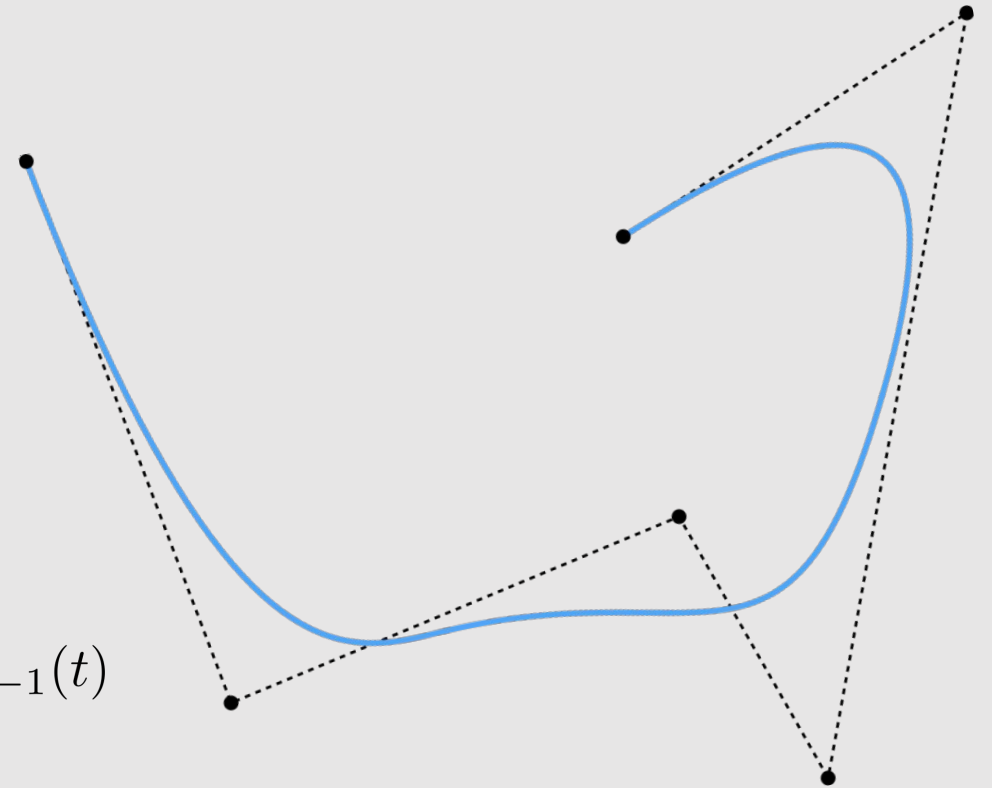
- And inductive condition:

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

- B-spline is a linear combination of bases:

$$f(t) := \sum_i a_i B_{i,d}$$

degree 



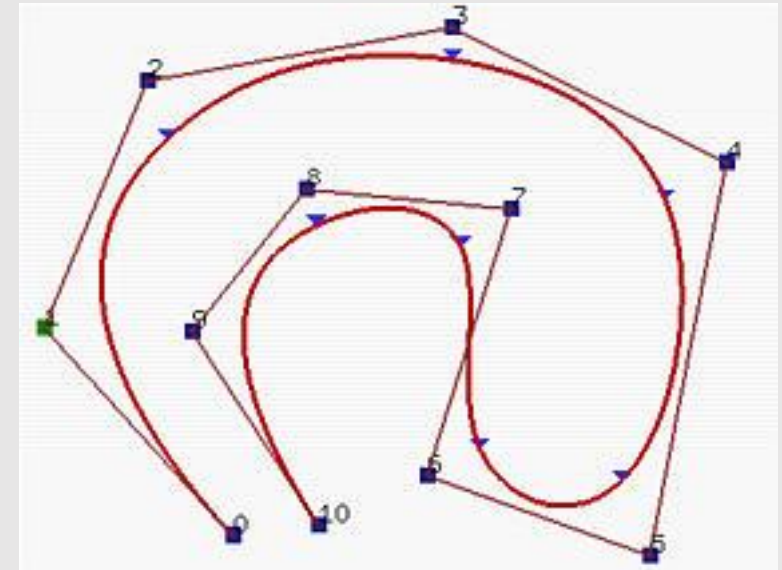
B-Splines

- **✗ Interpolation:** For higher degrees, splines do not pass through keyframes
- **✓ Continuity:** With higher degrees, bases are twice differentiable

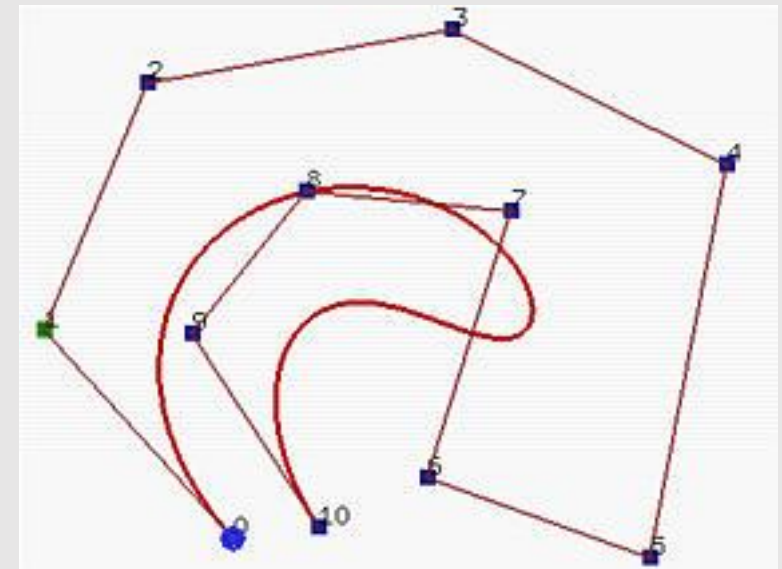
$$B_{i,k}(t) := \frac{t-t_i}{t_{i+k-1}-t_i} B_{i,k-1}(t) + \frac{t_{i+k}-t}{t_{i+k}-t_{i+1}} B_{i+1,k-1}(t)$$

- **✓ Locality:** B-spline bases are a function of the current and next bases only. Not a global system

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



[lower degree]



[higher degree]

Splines Review

	[Interpolation]	[Continuity]	[Locality]
Linear	✓	✗	✓
Natural	✓	✓	✗
Hermite	✓	✗	✓
Bezier	✓	✗	✓
Catmull-Rom	✓	✗	✓
B-Spline	✗	✓	✓

Splines Review

	[Interpolation]	[Continuity]	[Locality]
Linear	✓	✗	✓
Natural	✓	✓	✗
Hermite	✓	✗	✓
Bezier	✓	✗	✓
Catmull-Rom	✓	✗	✓
B-Spline	✗	✓	✓

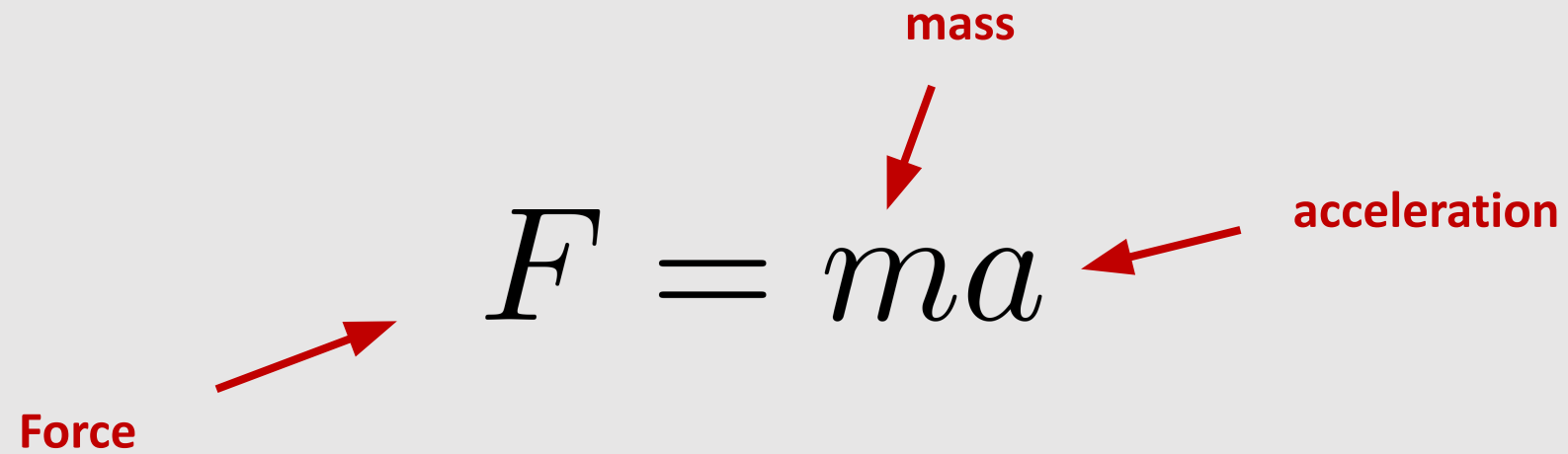
NO PERFECT SPLINE!

• ~~Splines~~

- Forward Kinematics
- Inverse Kinematics

We saw the rendering equation,
But what is the animation equation?

The Animation Equation



The diagram shows the equation $F = ma$ in a serif font. Three red arrows point to the variables: one from the word "Force" to F , one from the word "mass" to m , and one from the word "acceleration" to a .

$$F = ma$$

Force

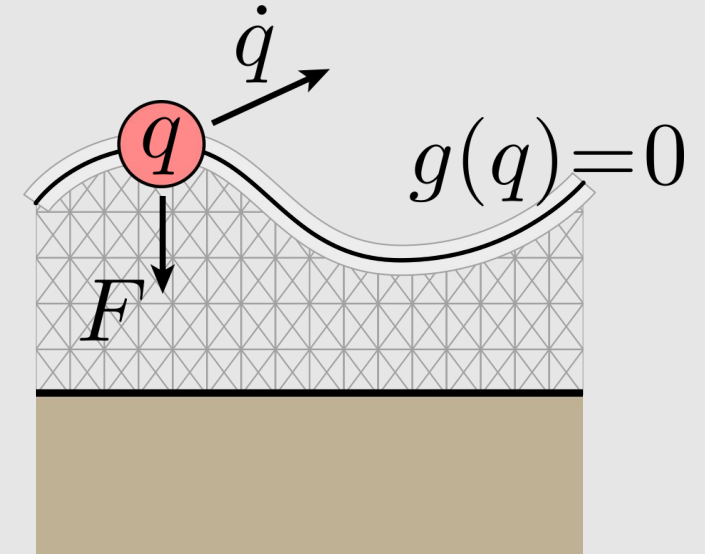
mass

acceleration

It's a little more complicated than just this...

An Animation System

- Component of an animation system:
 - Object's mass
 - Object's position
 - 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)

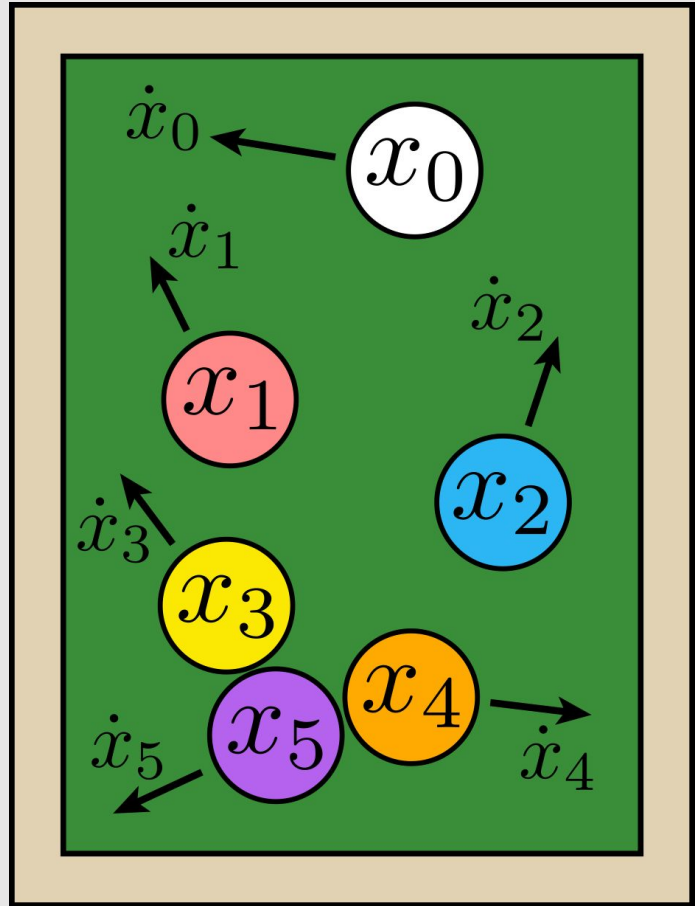


$$\dot{q} := \frac{d}{dt} q$$

$$g(q, \dot{q}, t) = 0$$

$$\ddot{q} = F/m$$

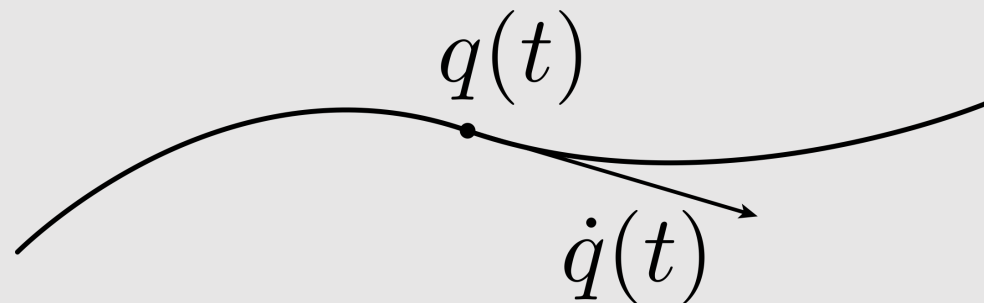
An Animation System



- Common to describe system with many moving pieces
 - **Ex:** a collection of billiard balls
 - Can collect into a single configuration:

$$q = (x_0, x_1, \dots, x_n)$$

- 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?



3D Animation in Unity (2020) Ing Jileček

Motion Capture

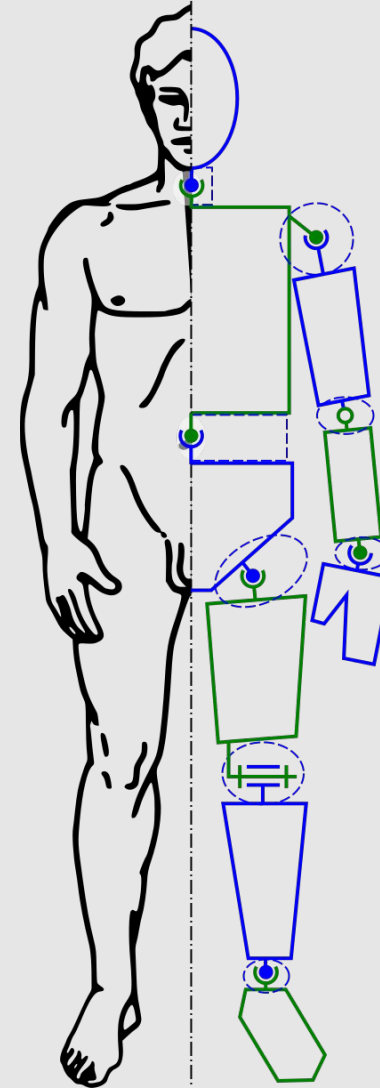
- Just take videos of real life poses
 - Map to character model
- Data can get very messy
 - Same idea as capturing a point cloud
- [+] Easy to understand
- [+] Capture real-life poses
- [-] Expensive to purchase
- [-] Very noisy data
- [-] Requires a lot of cleanup



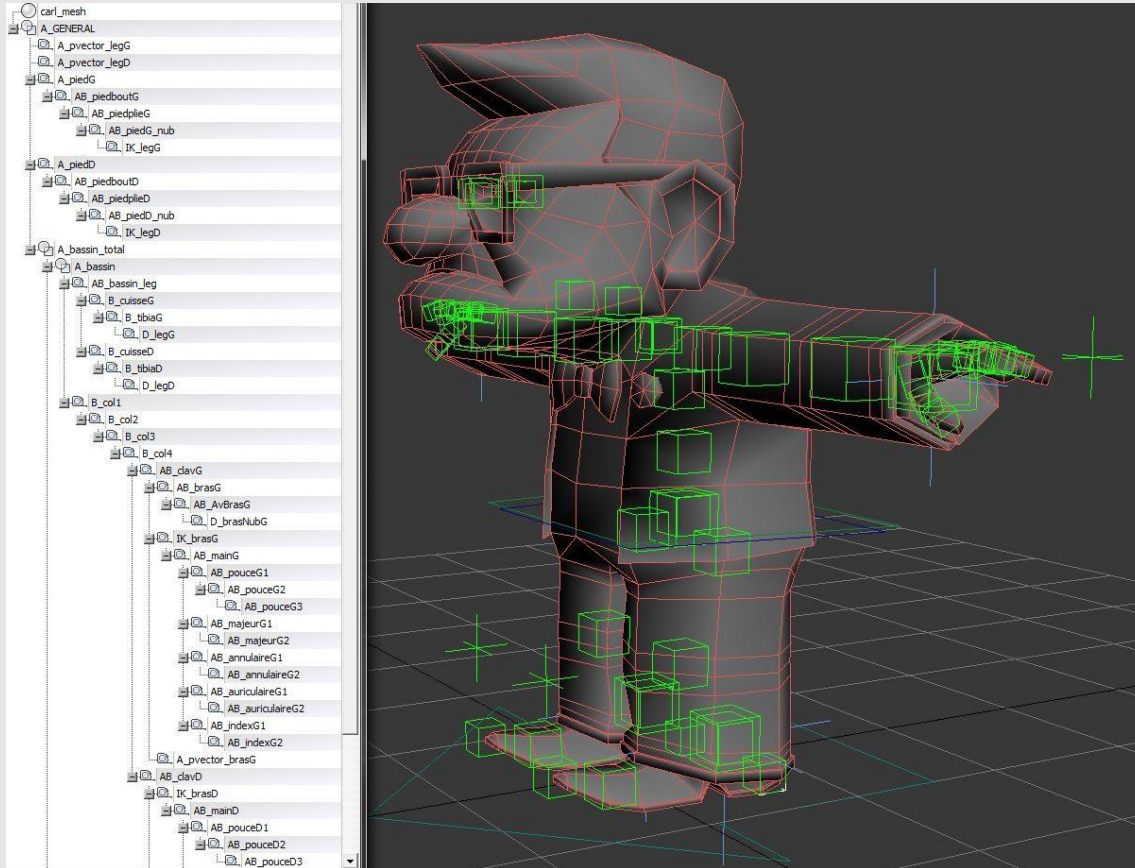
The Hobbit (2012) Peter Jackson

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)
 - Joints 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**!

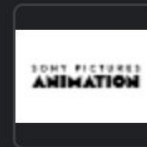


Character Rigging



Up (2009) Pixar

- 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



Rigging Artist
Sony Pictures Animation
Culver City, CA
via Greenhouse



Full-time

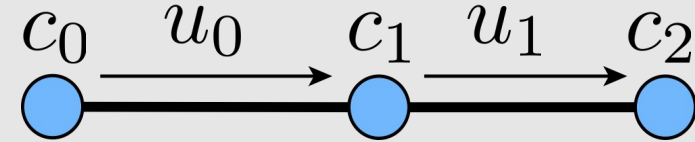


No degree mentioned

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)



- Elbow position p_1 computed as:

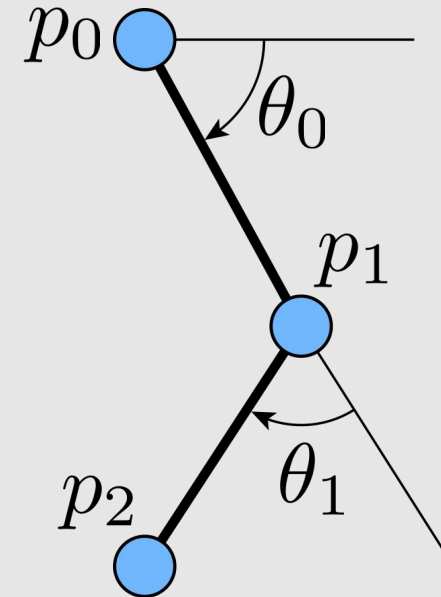
$$p_1 = p_0 + \begin{bmatrix} \cos \theta_0 & \sin \theta_0 \\ -\sin \theta_0 & \cos \theta_0 \end{bmatrix} u_0$$

- Can also be written as:

$$p_1 = p_0 + e^{i\theta_0} u_0$$

- New hand position p_2 computed as:

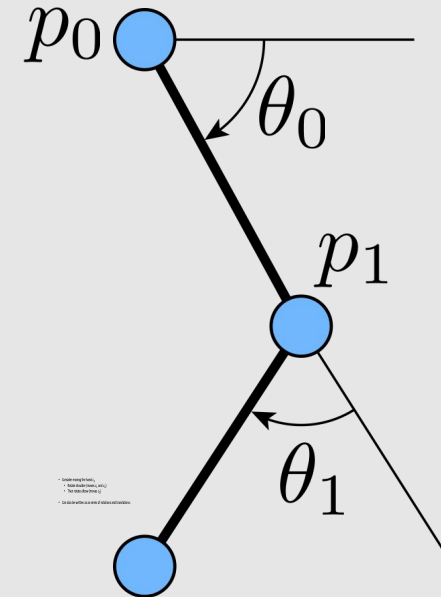
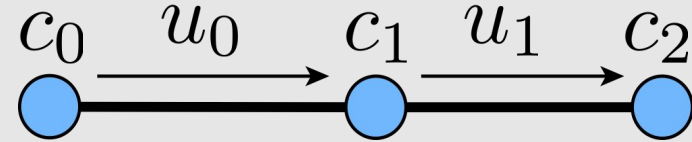
$$p_2 = p_0 + e^{i\theta_0} u_0 + e^{i\theta_0} e^{i\theta_1} u_1$$



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 as 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
 - Origin is often chest or space between rig's feet
- **Bone Space:** origin at the base point of the bone and its axes rotated by the bone (and its parents) rotations
 - **Bind Space:** no rotations, just translations
 - T-pose position

$$c_2 = T(u_1) T(u_0) c_0$$

- **Pose Space:** both rotations and translations
 - Model is "posed"

$$p_2 = T(u_1) R(\theta_1) T(u_0) R(\theta_0) p_0$$



[Bind Space]



[Pose Space]

Bind and Pose Spaces

- When rendering a model, need to bring all points into same coordinate space

- Bind-to-Local Transform:

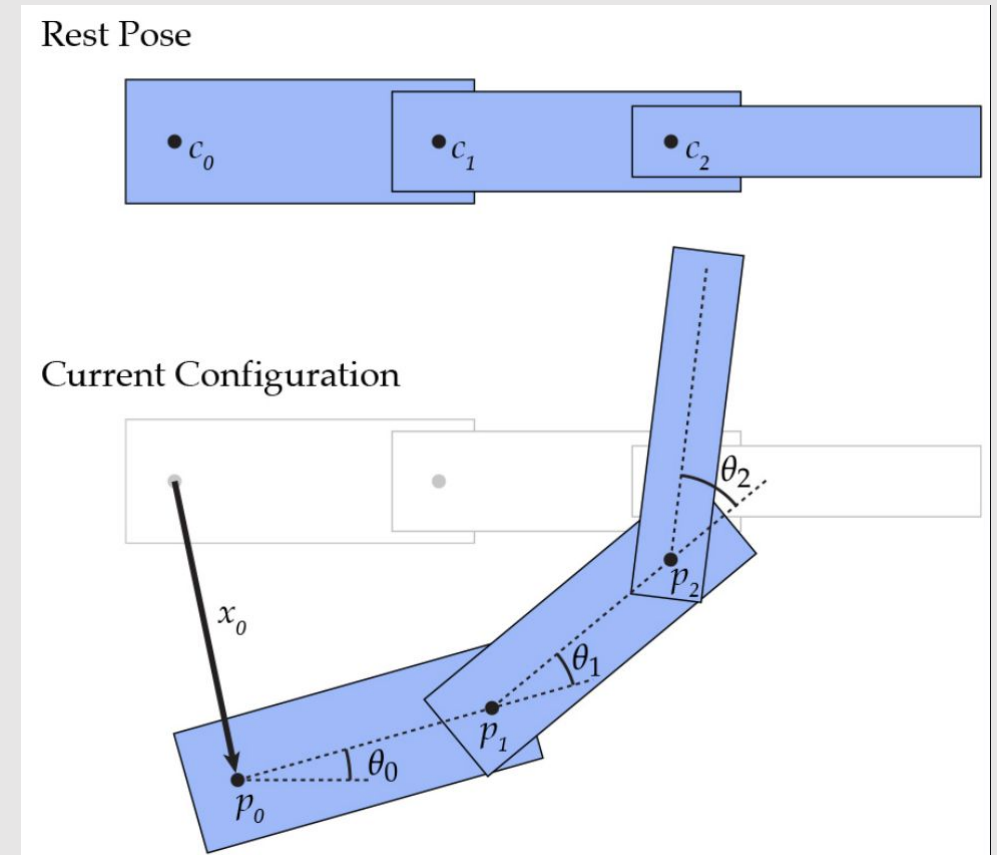
$$T_{2 \rightarrow 0} = T(p_1 - p_0) T(p_2 - p_1)$$

need to undo
p2's orientation

- Pose-to-Local Transform:

$$T_{2 \rightarrow 0} = R(\theta_0) T(p_1 - p_0) R(\theta_1) T(p_2 - p_1) R(\theta_2)$$

- Rotations and transformations saved as child-to-parent
- $(p_2 - p_1)$ and $(p_1 - p_0)$ are known as the **extent** of a joint
- Q:** how can we make these equations recursive?



Base Transform

- Need to account for base transforms

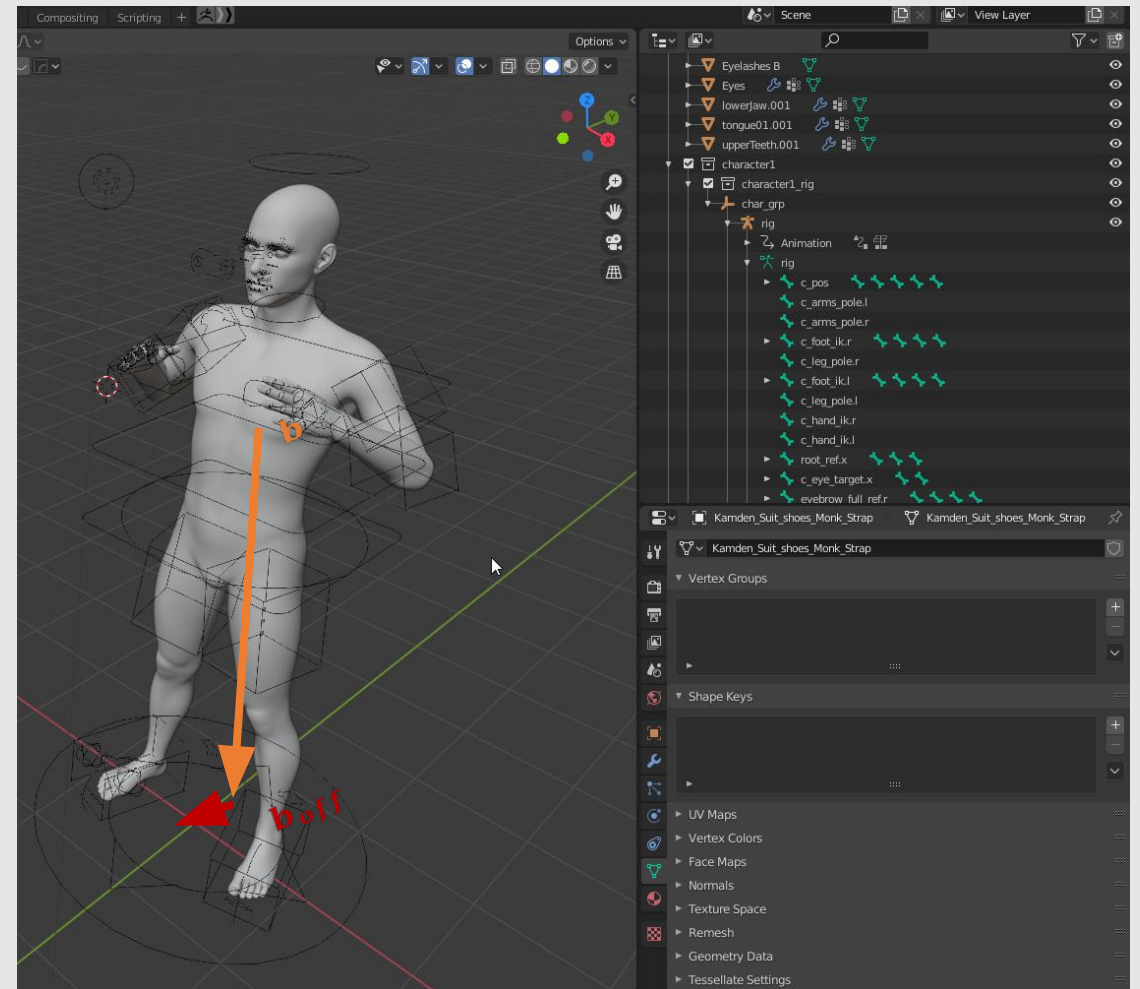
- Bind-to-Local Transform:

$$T_{2 \rightarrow b} = T(b) T(p_1 - p_0) T(p_2 - p_1)$$

- Pose-to-Local Transform:

$$T_{2 \rightarrow b} = T(b + b_{off}) R(\theta_0) T(p_1 - p_0) \dots$$

- Base transform b is the base point of the model
- Base transform offset b_{off} is the translational offset when the model is animated
 - Think moving the entire model!



Some very confused guy on the r/BlenderHelp subreddit (2020) Blender

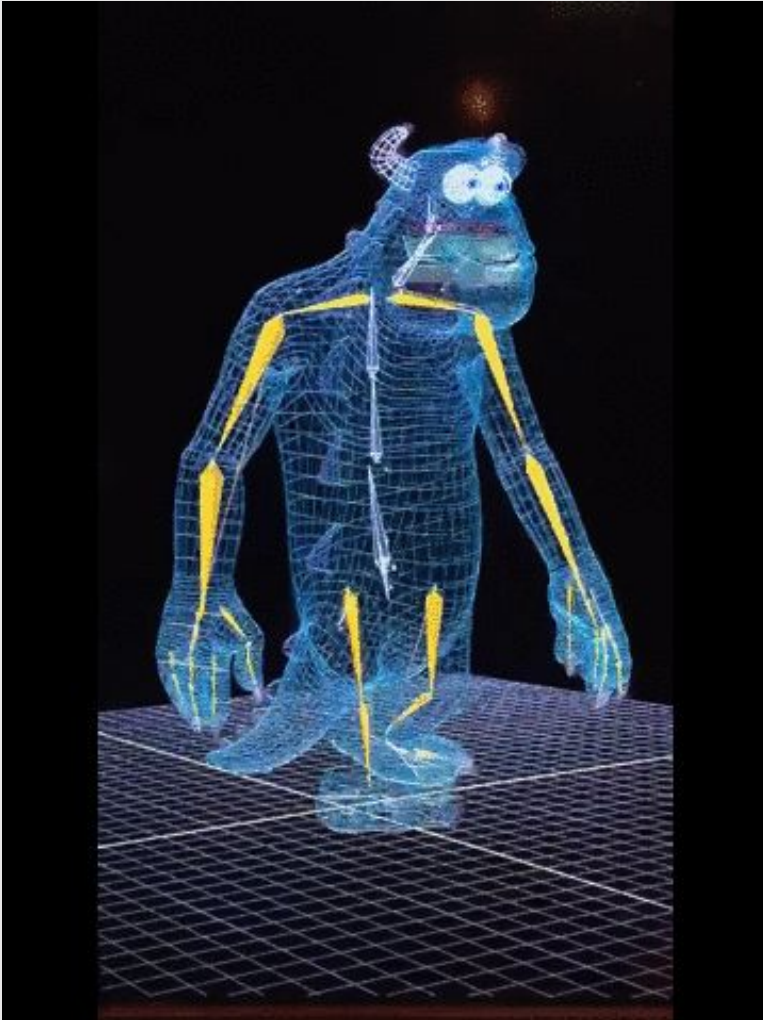
Forward Kinematics

- [+] 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



Monster's Inc (2001) Pixar

- 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

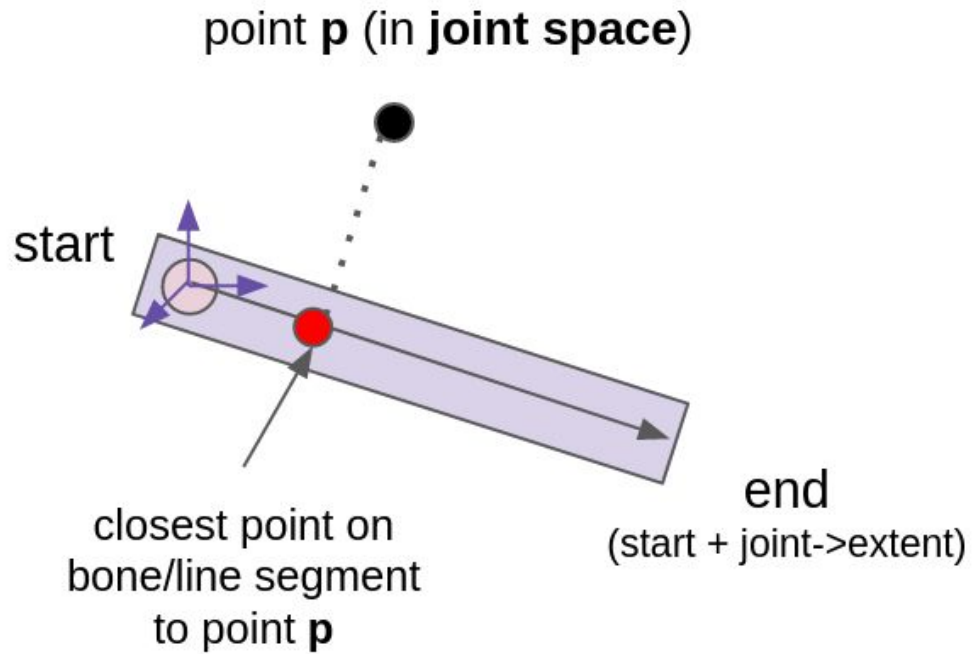
$$\sum_j w_{ij} = 1$$

- 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 :)

Computing Weights



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

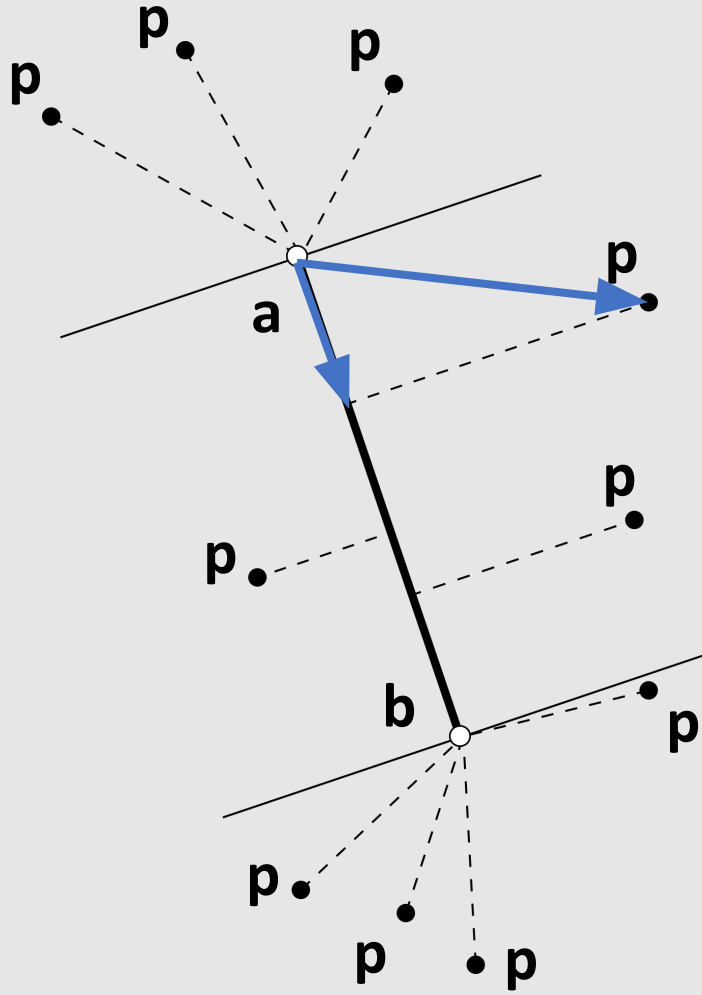
$$\hat{w}_{ij} = \frac{\max(0, r - d_{ij})}{r}$$

Why do we need r ?

- Make sure to normalize weights

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

Review: Closest Point on a Line Segment



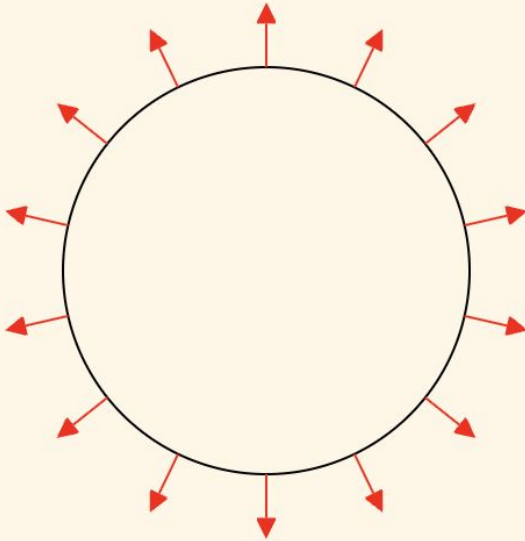
Compute the vector p from the line base a along the line

Normalize to get a time

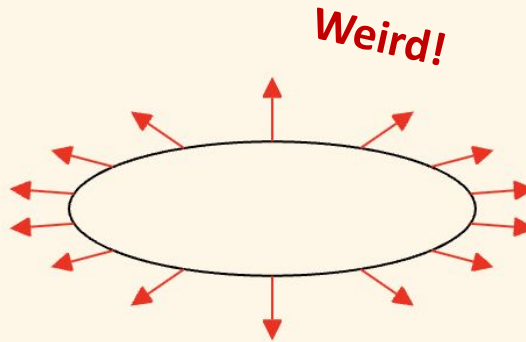
Clip time to range $[0,1]$ and interpolate

Linear Blend Normals

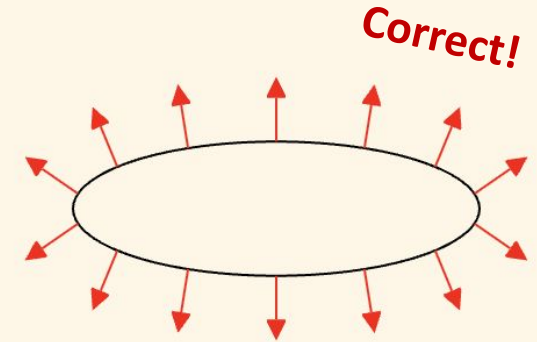
Let our skinning matrix $M_i = \sum_j (w_{ij} P_j B_j^{-1})$



$$\begin{aligned} v'_i &= v_i \\ n'_i &= n_i \end{aligned}$$



$$\begin{aligned} v'_i &= M_i v_i \\ n'_i &= M_i n_i \end{aligned}$$



$$\begin{aligned} v'_i &= M_i v_i \\ n'_i &= (M_i^{-1})^T n_i \end{aligned}$$

inverse transpose

**Derivation: <https://paroj.github.io/gltut/Illumination/Tut09%20Normal%20Transformation.html>

Linear Blend Normals

Let our skinning matrix $M_i = \sum_j (w_{ij} P_j B_j^{-1})$

Polar decomposition gives:

$$M_i = R_{i1} * S_i * R_{i2}$$

Invert scale only:

$$M'_i = R_{i1} * S_i^{-1} * R_{i2}$$

Apply inverse and transpose to all elements:

$$M'_i = (R_{i1}^{-1})^T * (S_i^{-1})^T * (R_{i2}^{-1})^T$$

$$(R_{i2}^{-1})^T = R_{i2}$$

$$(S_i)^T = S_i$$

Take out transpose:

$$M'_i = (R_{i2}^{-1} * S_i^{-1} * R_{i1}^{-1})^T$$

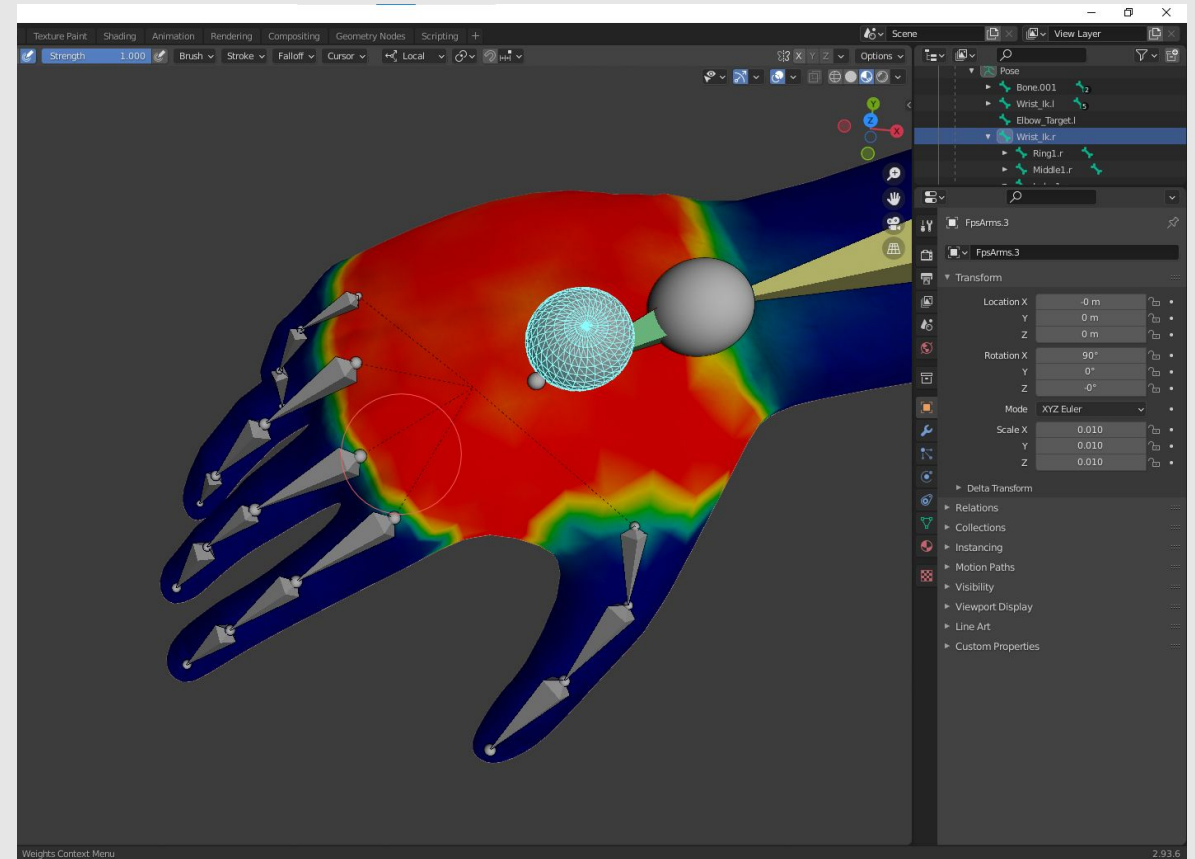
Take out inverse:

$$M'_i = ((R_{i1} * S_i * R_{i2})^{-1})^T = (M_i^{-1})^T$$

inverse transpose

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



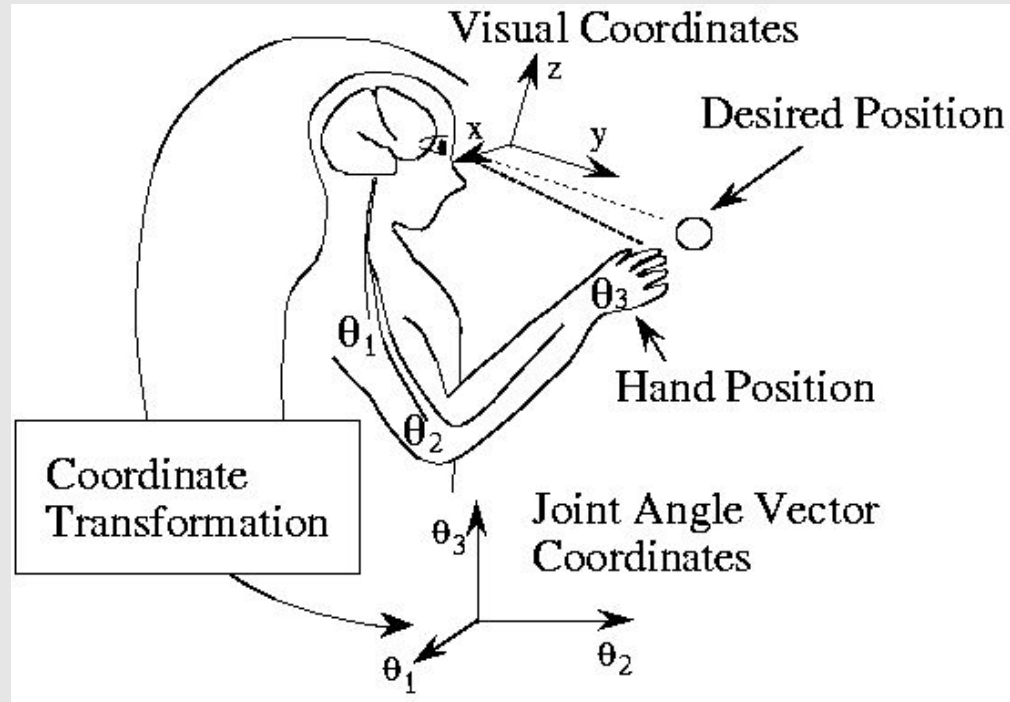
Blender (2021) Ton Roosendaal

- ~~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
 - 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...

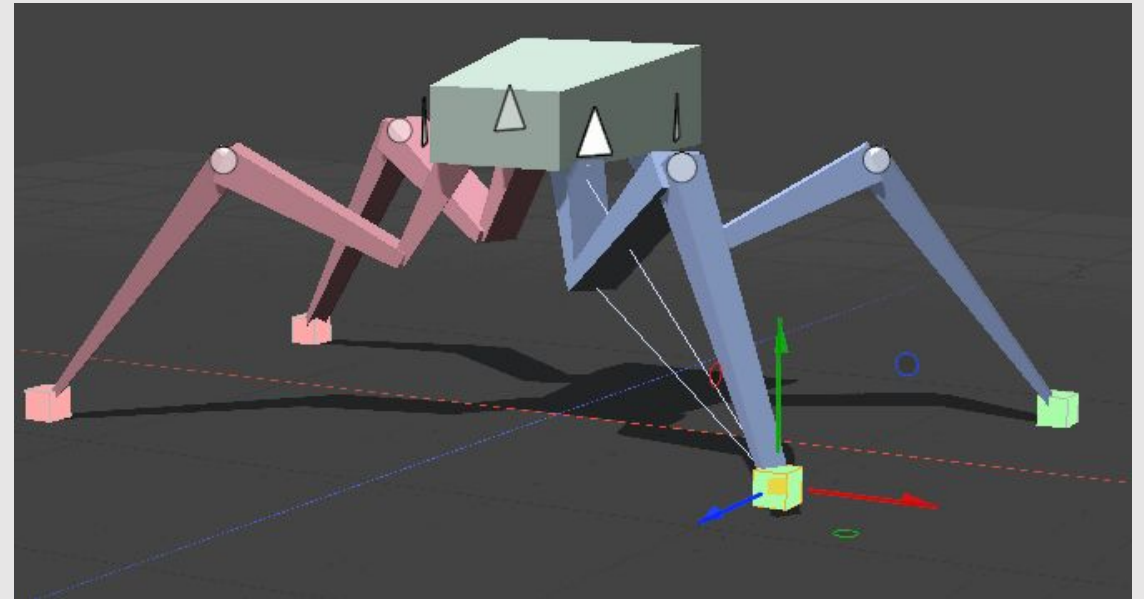
Inverse Kinematics

- 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 τ :

$$q = q - \tau \nabla f(q)$$



Foundry (2020) Foundry Hub

Inverse Kinematic Gradient

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

Take gradient with respect to function

$$\frac{df}{d\theta_k^y} = \sum_{(i,h)} (p_i(q) - h) \frac{dp_i}{d\theta_k^y}$$

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

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

Gradient breaks down into 3 parts:

$$\frac{dp_i}{d\theta_k^y} = \underbrace{R(\theta_0^z) R(\theta_0^y) R(\theta_0^x) T(u_0) \dots R(\theta_k^z)}_{\text{[linear transformation]}} \underbrace{\frac{d}{d\theta_k^y} R(\theta_k^y)}_{\text{[derivative]}} \underbrace{R(\theta_k^x) T(u_i) \dots R(\theta_i^z) R(\theta_i^y) R(\theta_i^x) u_i}_{\text{[transformed point]}}$$

Inverse Kinematic Gradient

$$\frac{dp_i}{d\theta_k^y} = ???$$

Fun fact: by transforming the axis of rotation and base point to local coordinates, Then the derivative of the rotation $R(\theta_k^y)$ by amount θ_k^y around axis y and center r of point p becomes:

$$\frac{dp_i}{d\theta_k^y} = y \times (p - r)$$

constant for a
given handle


$$p = [\text{linear transformation}] [R(\theta_k^y)] [\text{transformed point}]$$

specific to the
current joint


$$r = [\text{linear transformation}'] [0,0,0]$$


$$y = ([\text{linear transformation}'] [R(\theta_k^z)]) . \text{rotate}(\theta_k^y)$$

[linear transformation'] = all rotations and transformations up to, but not including the kth bone

Inverse Kinematic Gradient

- All joints that come before joint k contribute to the movement of joint k
 - **Example:** moving your shoulder moves your hand
- Compute how every joint prior to joint k affects the movement of joint k
 - Gives us a gradient for each joint in range $[0 - k]$

$$\nabla f_k^y = (p_i(q) - h) \cdot [y_k \times (p_i(q) - r_k)]$$

$$\nabla f_{k-1}^y = (p_i(q) - h) \cdot [y_{k-1} \times (p_i(q) - r_{k-1})]$$

$$\nabla f_{k-2}^y = (p_i(q) - h) \cdot [y_{k-2} \times (p_i(q) - r_{k-2})]$$

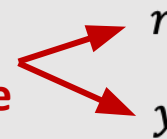
...

$$\nabla f_0^y = (p_i(q) - h) \cdot [y_0 \times (p_i(q) - r_0)]$$

constant for a
given handle



specific to the
current joint



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!

$$\nabla f_k^y += (p_i(q) - h) \cdot [y_k \times (p_i(q) - r_k)]$$

$$\nabla f_{k-1}^y += (p_i(q) - h) \cdot [y_{k-1} \times (p_i(q) - r_{k-1})]$$

$$\nabla f_{k-2}^y += (p_i(q) - h) \cdot [y_{k-2} \times (p_i(q) - r_{k-2})]$$

...

$$\nabla f_0^y += (p_i(q) - h) \cdot [y_0 \times (p_i(q) - r_0)]$$

Inverse Kinematic Gradient

```
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

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

```
vec3 gradient_in_current_pose() {  
  
    for (auto &handle : handles) {  
  
        Vec3 h = handle.target;  
        Vec3 p = // TODO: compute output point  
  
        Hint: look at  
        Skeleton::Bone::compute_rotation_axes  
        Mat4::angle_axis  
        Mat4::rotate  
  
        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 ∇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
- **Q:** What is a good step size τ ?

