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

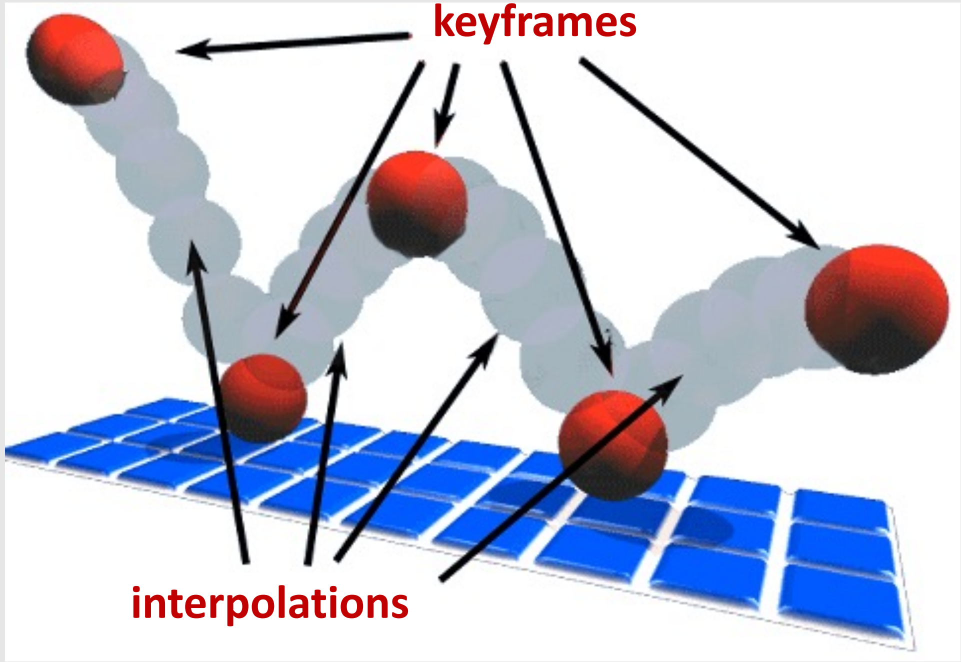
- **Today:** No need to draw anything, computer takes care of everything
    - Set keyframes by hand
        - **Forward Kinematics**
        - **Inverse Kinematics**
    - Allow keyframes to interpolate
        - **Splines**

Keyframing

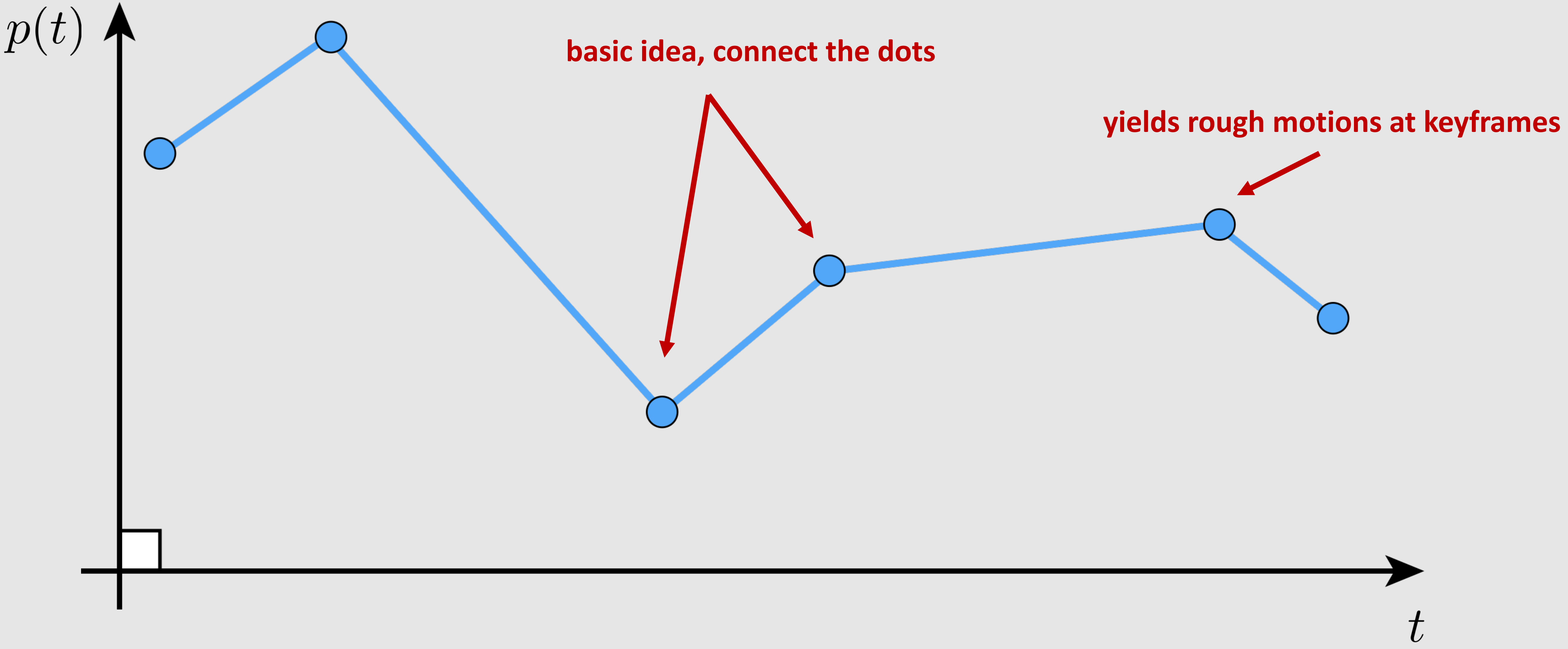
- Set keyframes at important locations in the animation
  - Have the computer interpolate the rest

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

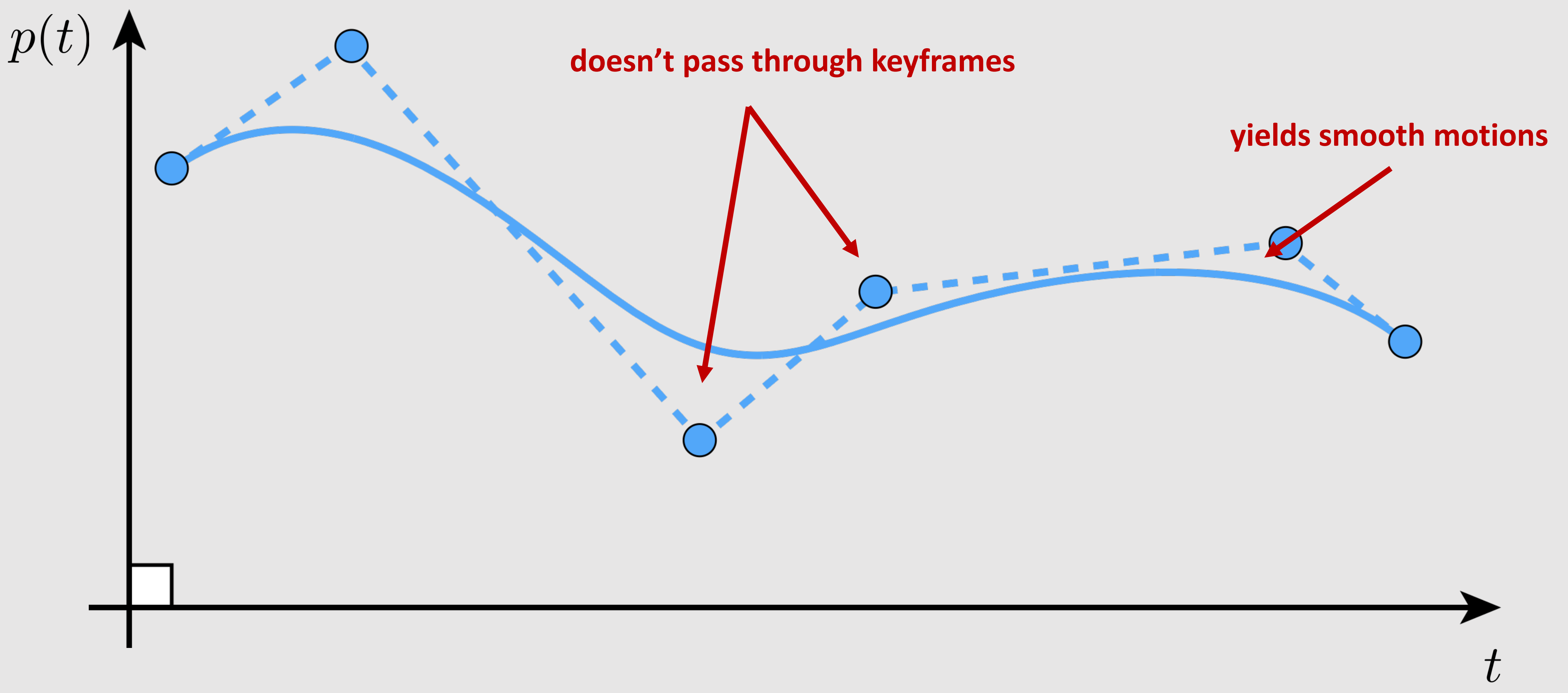
- **Problem:** how should data interpolate?
  - Linearly?
  - Along a curve/arc?



Linear Interpolation

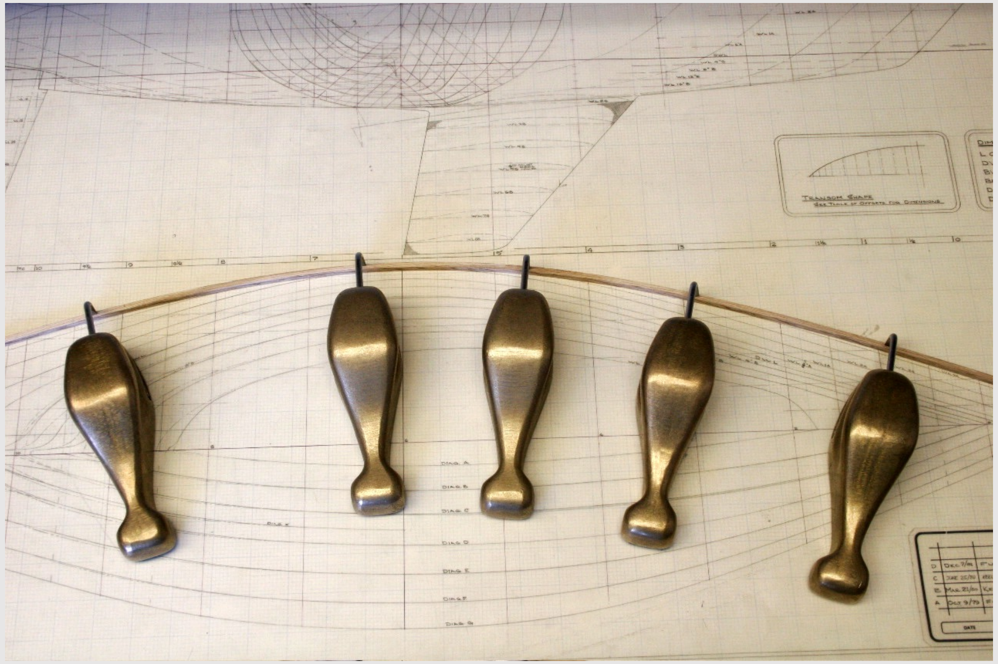


Piecewise Polynomial Interpolation

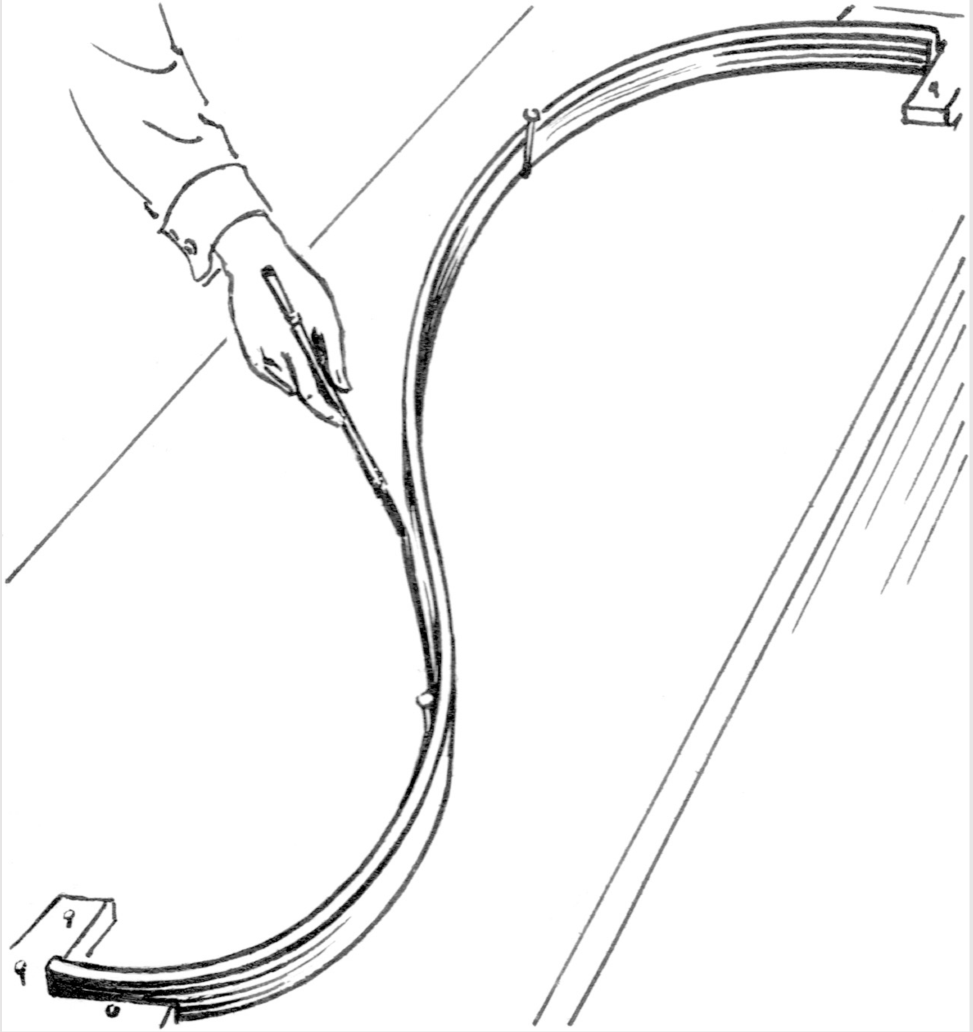


Splines

- Mathematical theory of interpolation arose from study of thin strips of wood or metal ("splines") under various forces
  - **Splines** help us define how interpolation should occur



The Elastica: A Mathematical History (2008) Levin



Splines

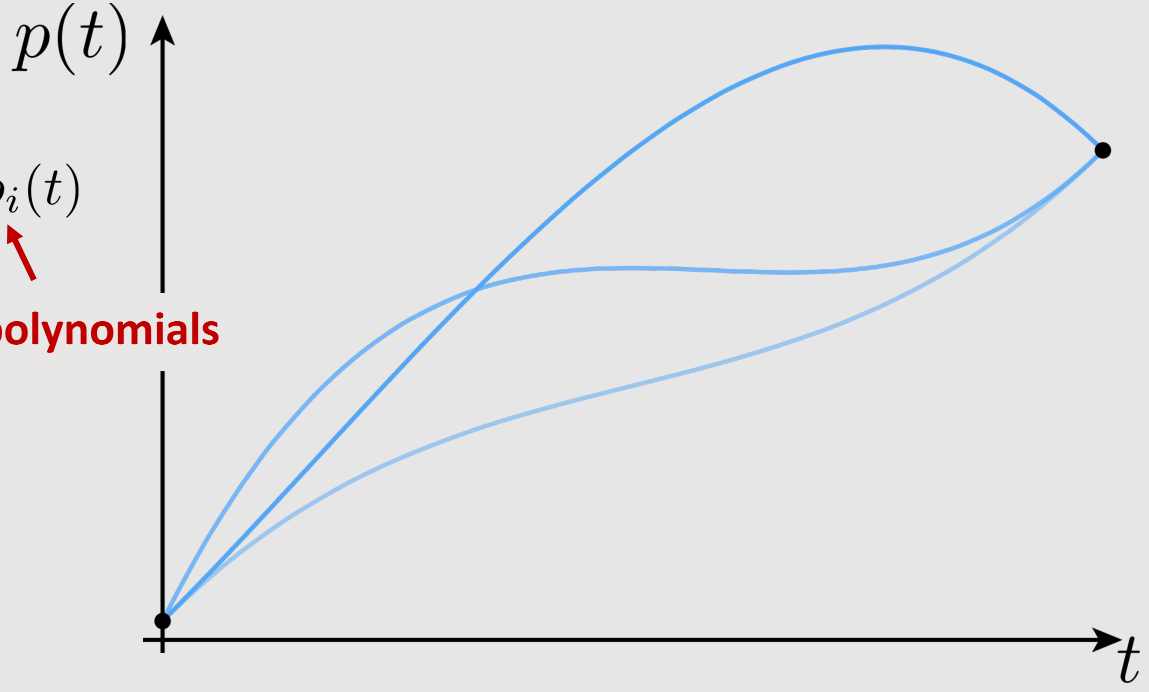
- In this course, a spline is any piecewise cubic polynomial function

degree

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

coefficients

polynomials



Many solutions!

- **Common spline:** cubic polynomial:

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

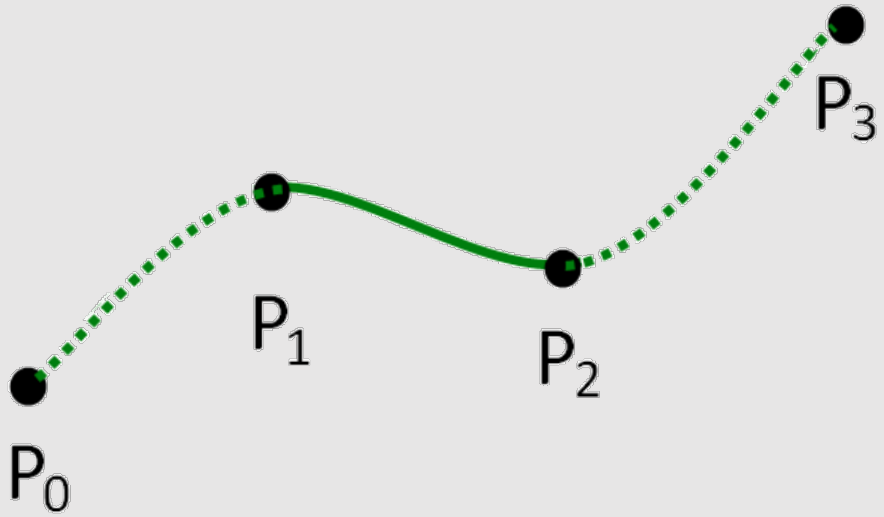
- 3 goals of splines:
  - Interpolation
  - Continuity
  - Locality

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

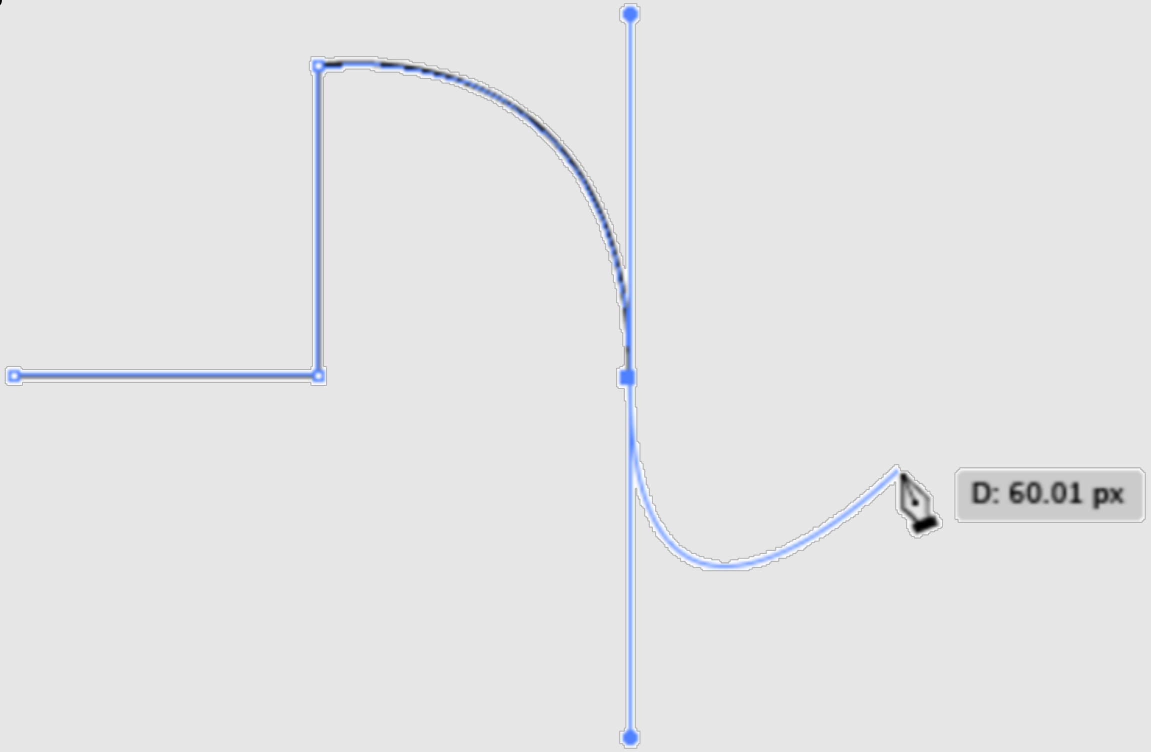


Continuity

- **Continuity:** Is the spline twice differentiable along control points

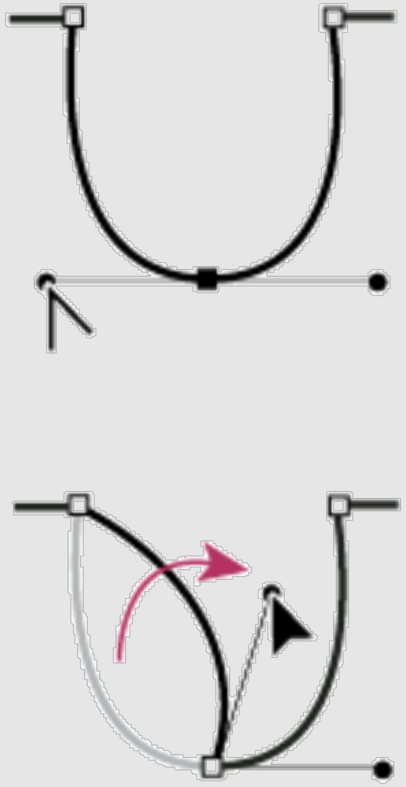
- C0 continuity – keyframes are continuous
- C1 continuity – first derivative is continuous
- C2 continuity – second derivative is continuous

- Saying a spline has continuity requires C2 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

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

- Animator specifies where a curve starts, ends, and the tangents at those points

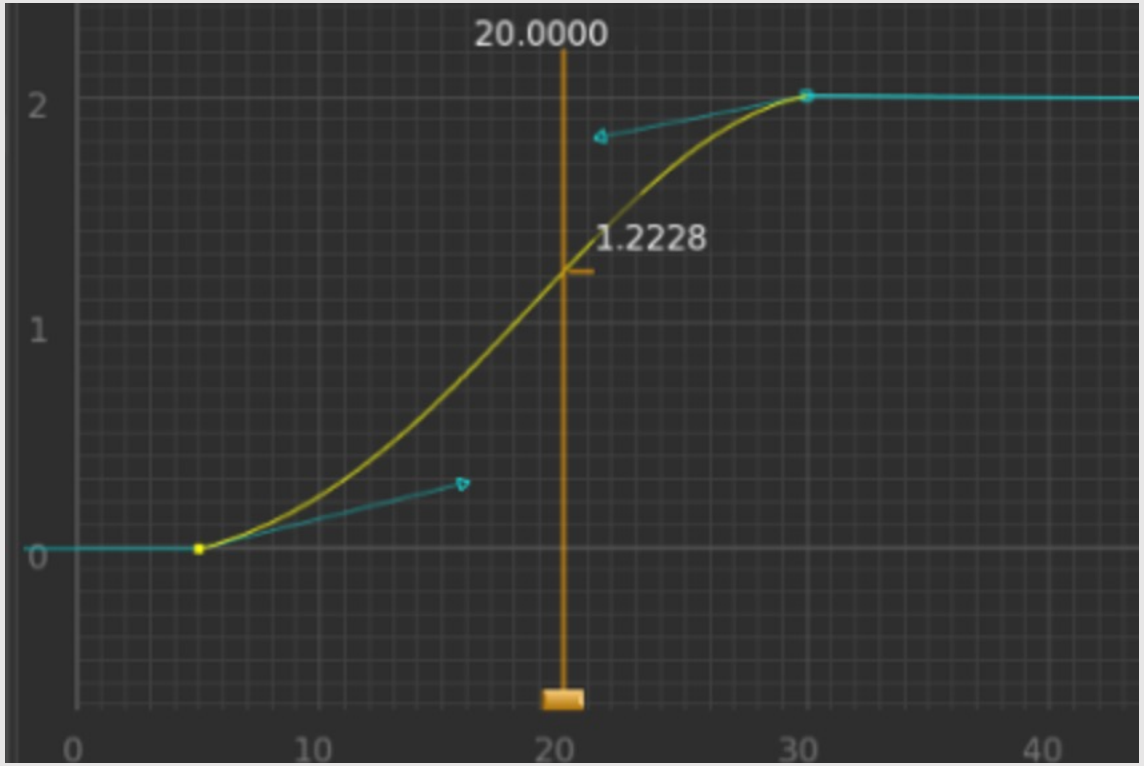
$$p(0) = p_0 \qquad \Rightarrow d = p_0$$

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

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

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

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



Piecewise Cubic Polynomial

- Can also write

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

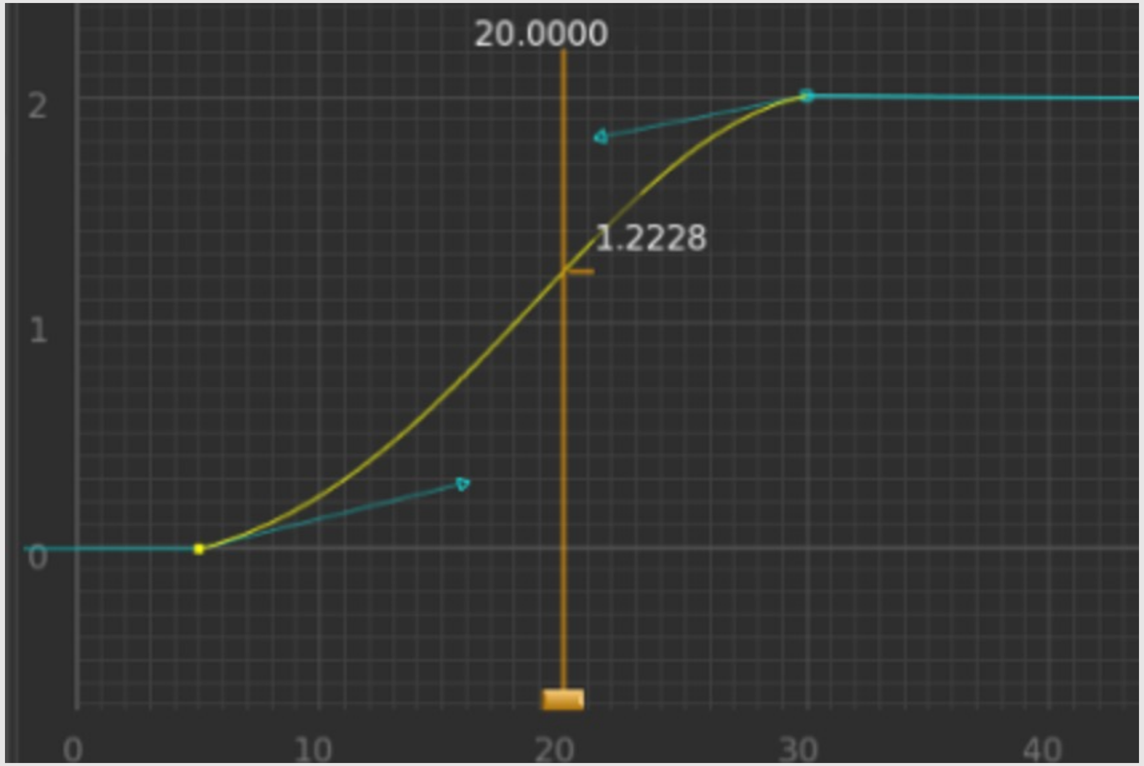
$$p(1) = p_1 \qquad \Rightarrow a + b + c + d = p_1$$

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

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

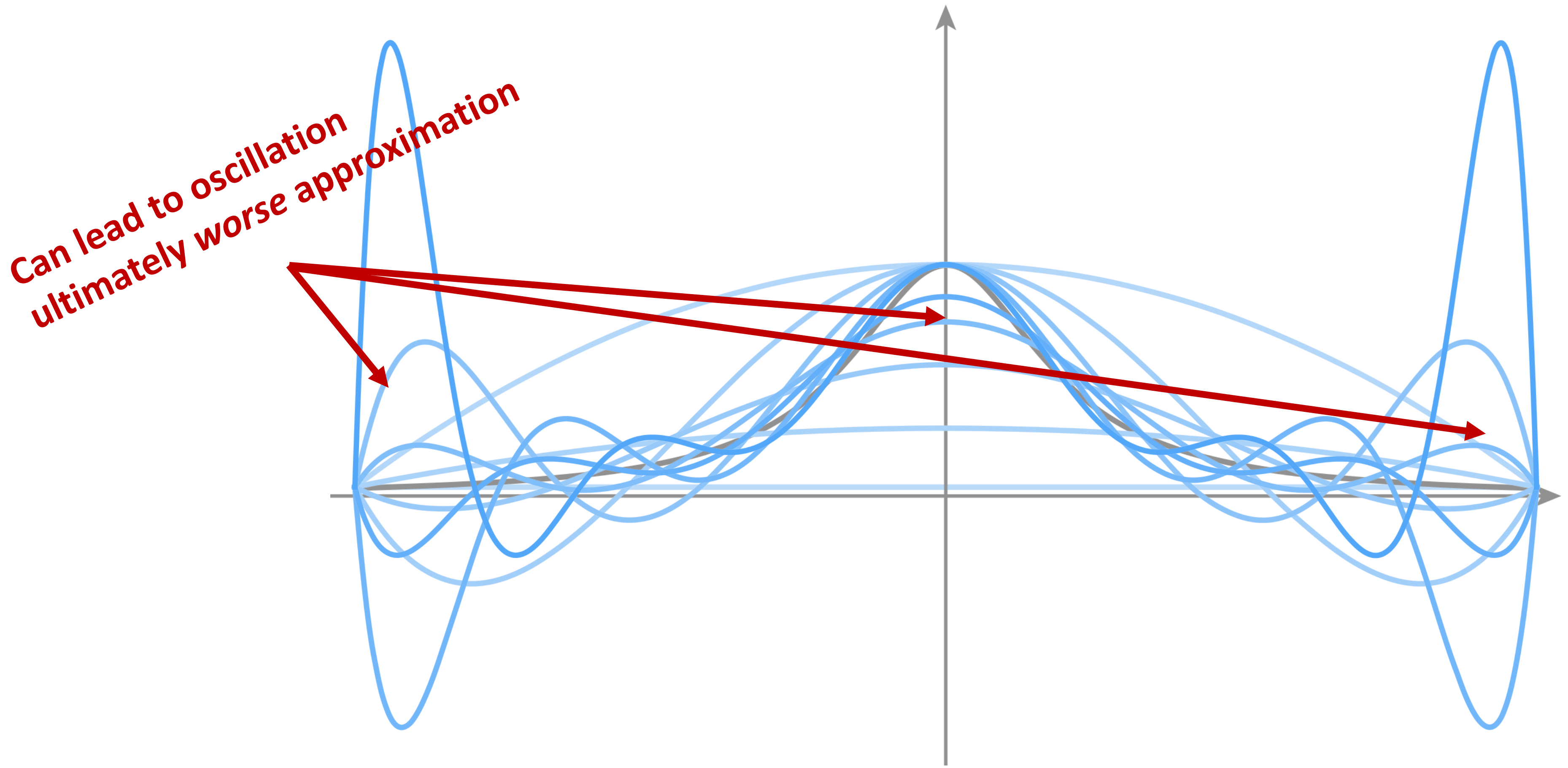
- as a linear system!

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



Runge Phenomenon

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



Natural Splines

- 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]:

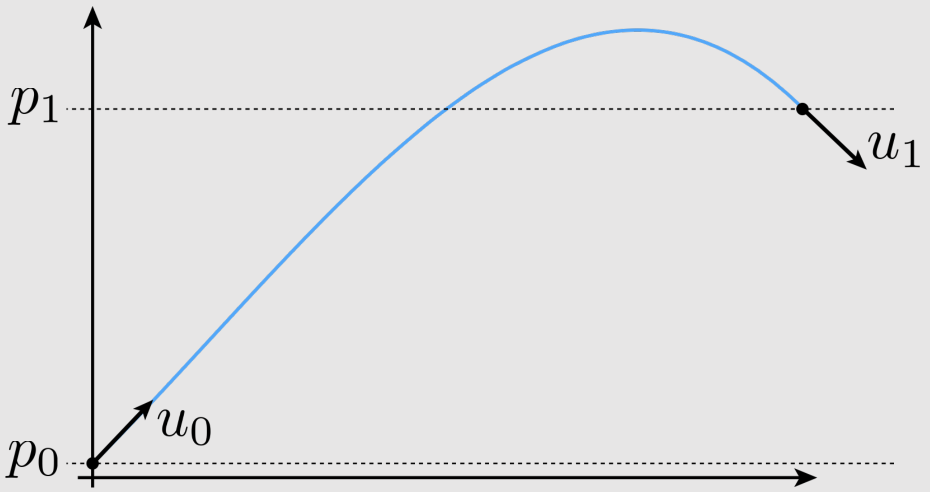
$$p_i(t_i) = f_i, \qquad p_i(t_{i+1}) = f_{i+1}, \qquad \forall i = 0, \dots, n-1$$

    - Tangents agree at endpoints [C1 continuity]:

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

    - Curvature agrees at endpoints [C2 continuity]:

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



- Total equations:
  - 2n + (n-1) + (n-1) = 4n – 2
- Total DOFs:
  - 2n + n + n = 4n
- Set curvature at endpoints to 0 and solve

$$p'_0(t_0) = 0, \qquad p''_0(t_{i+1}) = 0$$

Natural Splines

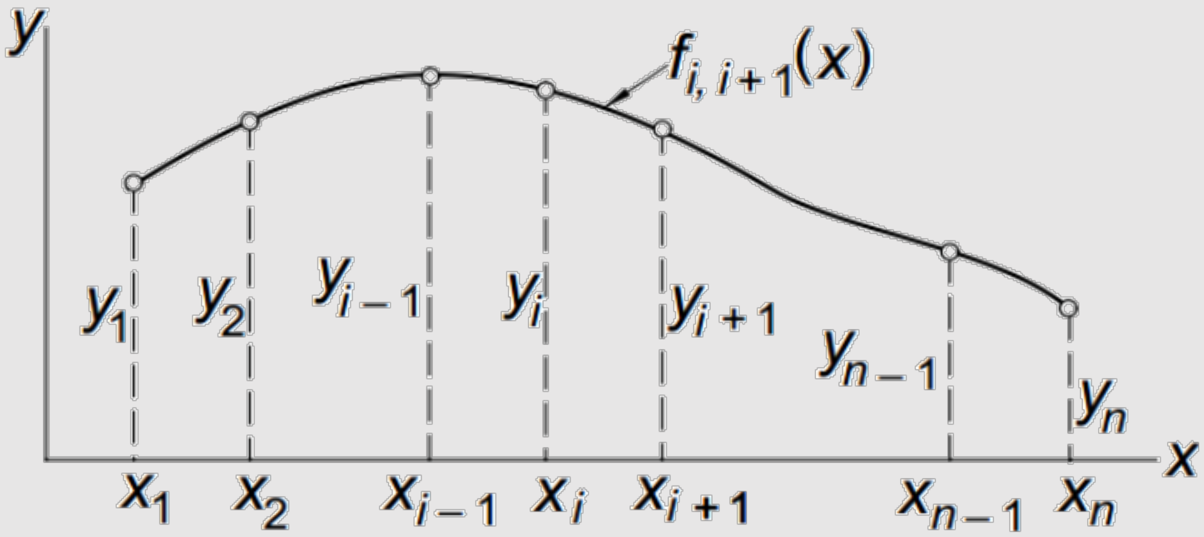
- ✓ **Interpolation:** by definition

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

- ✓ **Continuity:** by definition

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

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



Hermite/Bézier Splines

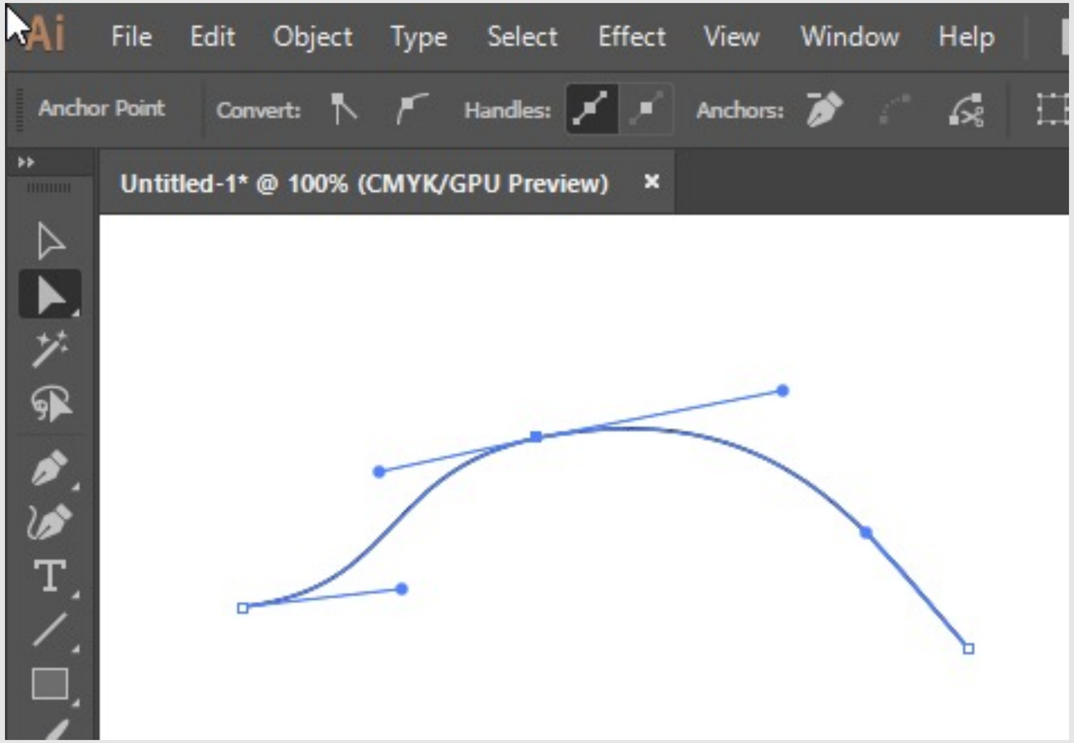
- Each cubic piece specified by endpoints and tangents
  - Keyframes set at endpoints:

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

  - Tangents set at endpoint:

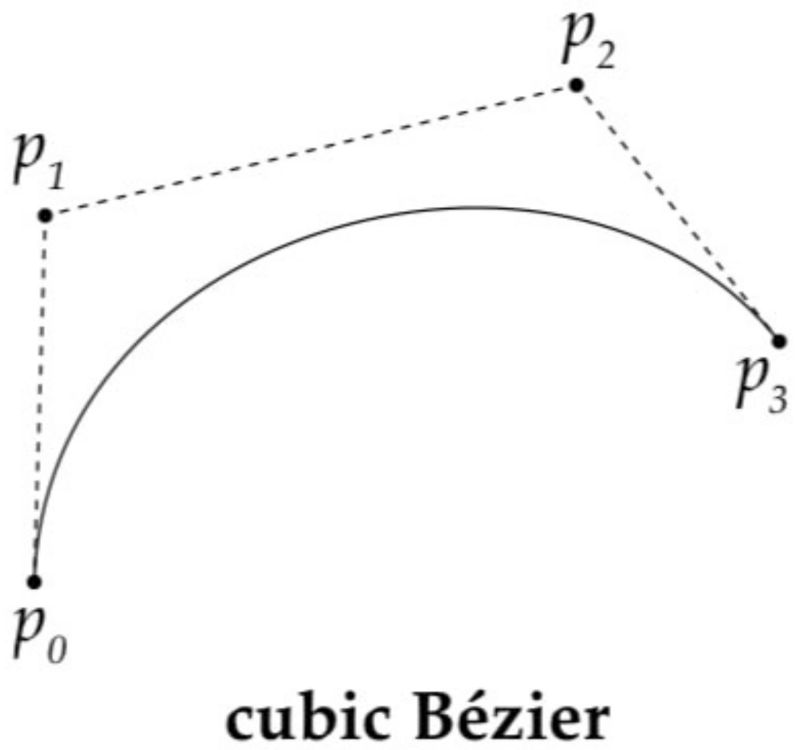
$$p'_i(t_i) = u_{i,} \qquad p'_i(t_{i+1}) = u_{i,+1}, \qquad \forall i = 0, \dots, n-1$$

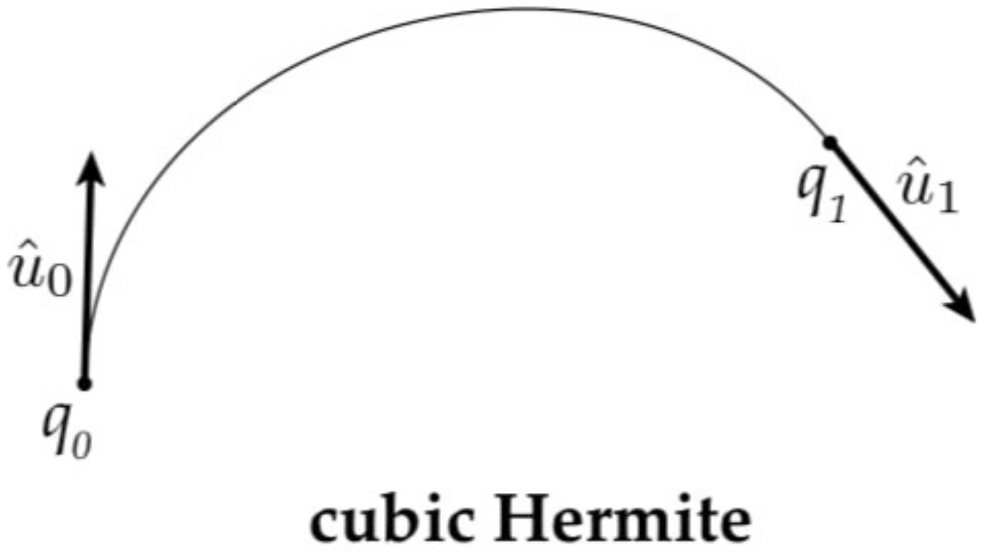
- Natural splines specify just keyframes
  - Bezier splines specify keyframes and tangents
  - Can get continuity if tangents are set equal
- Total equations:
  - 2n + 2n = 4n
- Commonly used in vector art programs
  - Illustrator
  - Inkscape
  - SVGs



Hermite/Bézier Splines

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

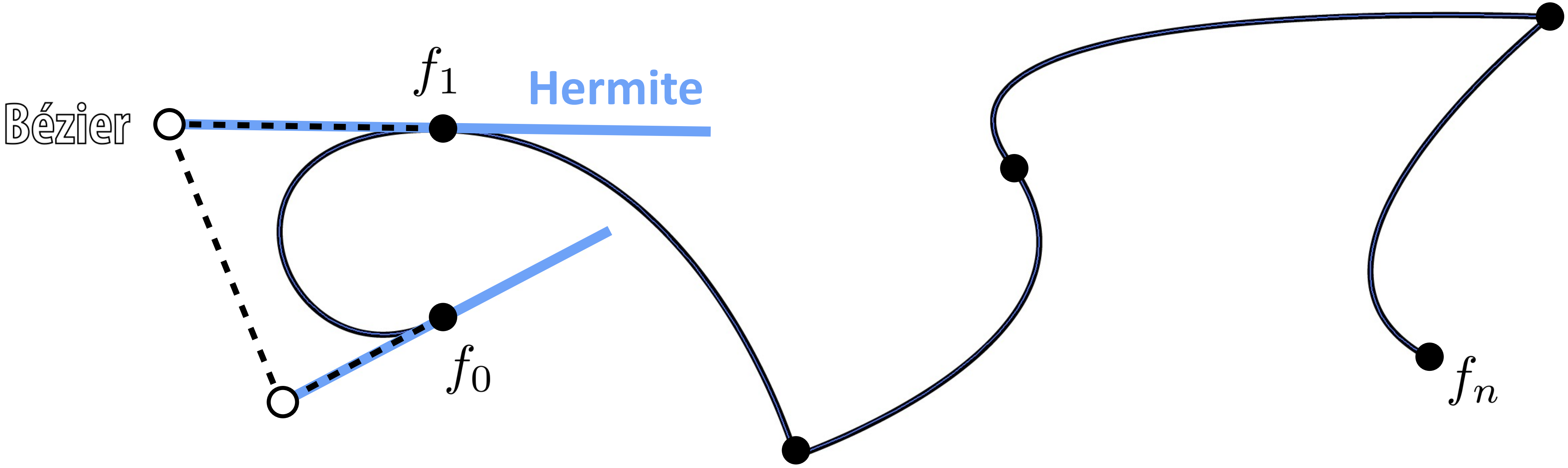




Same computation and properties! Just a different interface

Hermite/Bézier Splines

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



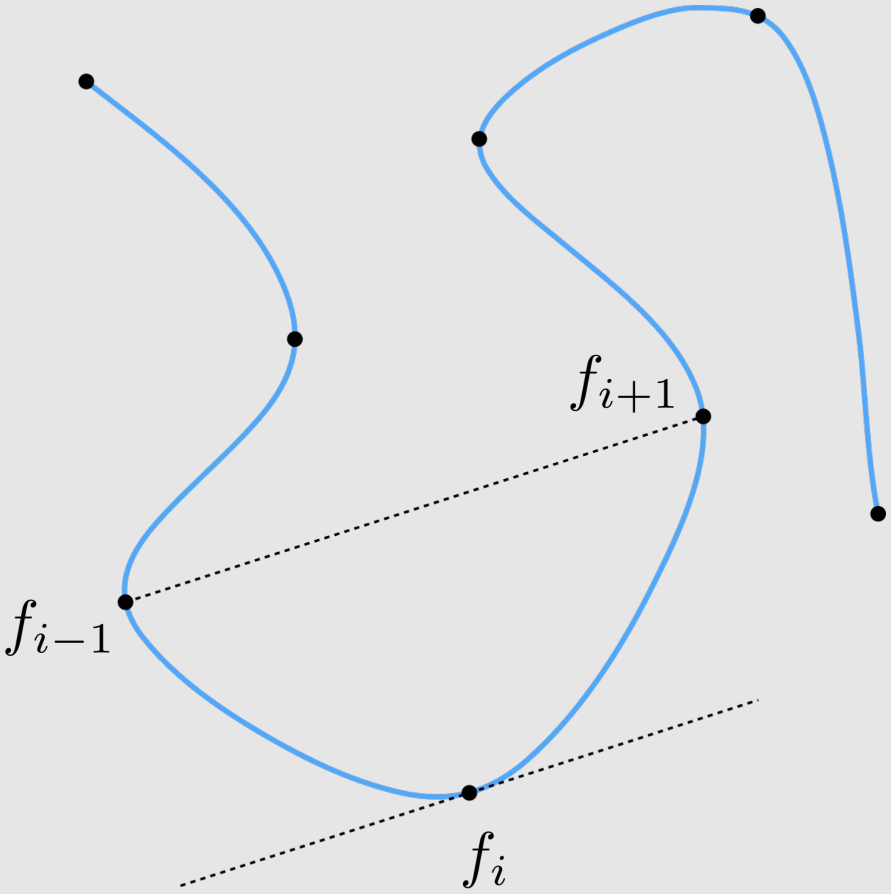
Same computation and properties! Just a different interface

Catmull-Rom Splines

- A specialized version of Hermite splines
    - Only 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/Bézier/Catmull-Rom Splines

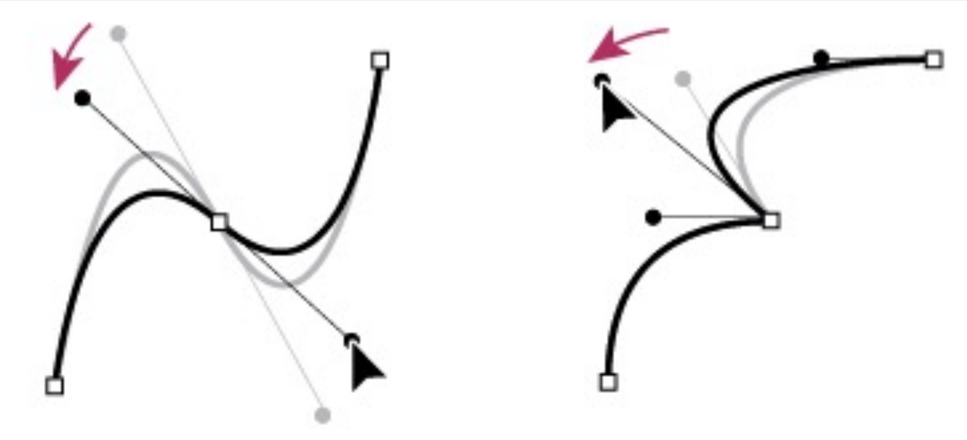
- ✓ **Interpolation:** by definition

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

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

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

- ✓ **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:

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

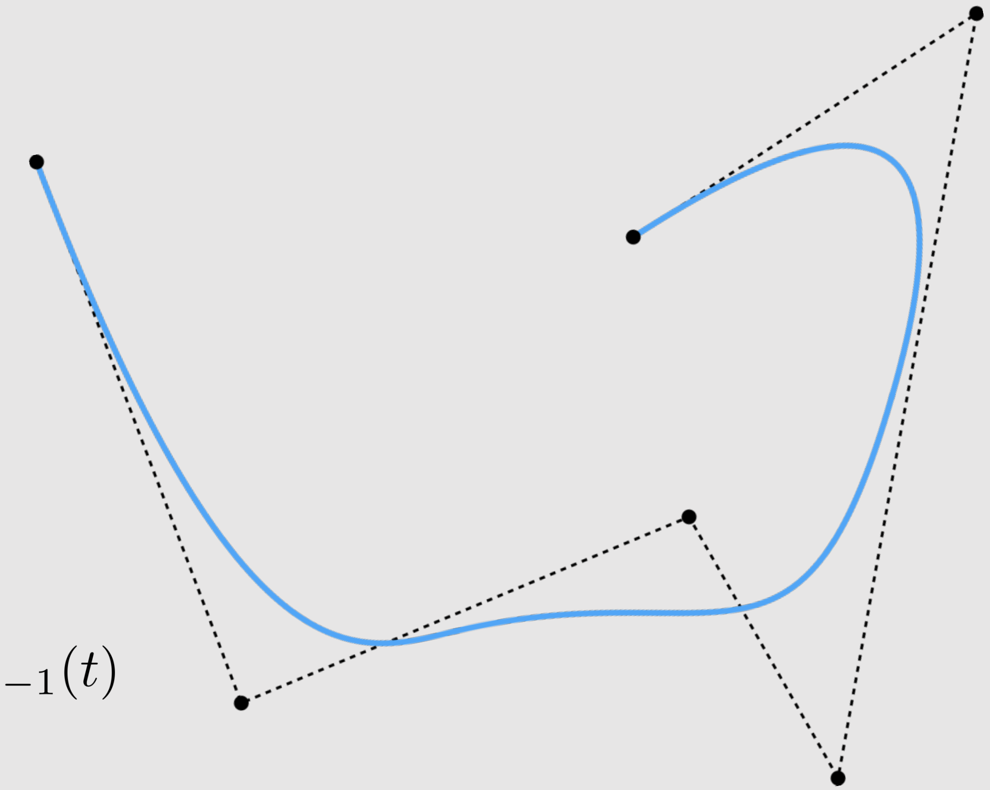
- And inductive condition:

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

- B-spline is a linear combination of bases:

degree

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



B-Splines

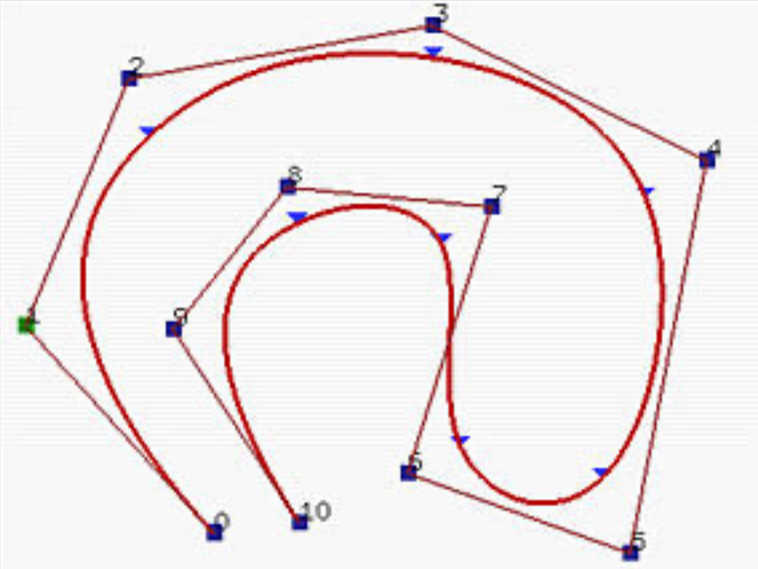
- ✗ **Interpolation:** For higher degrees, splines do not pass through keyframes

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

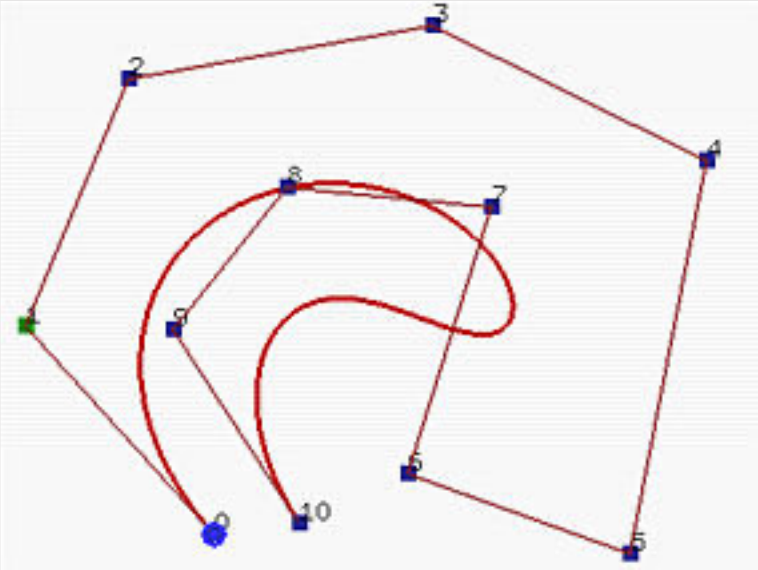
$$B_{i,k}(t) := \frac{t - t_i}{t_{i+k-1} - t_i} B_{i,k-1}(t) + \frac{t_{i+k} - t}{t_{i+k} - t_{i+1}} B_{i+1,k-1}(t)$$

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

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



[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

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

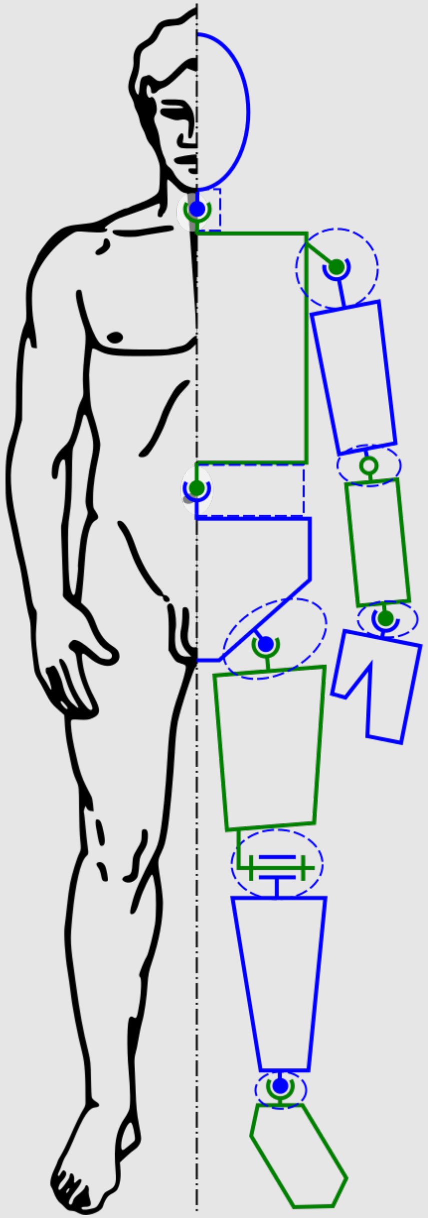
- Just take videos of real life poses
  - Map to character model
- Data can get very messy
  - Same idea as capturing a point cloud
- [+] Easy to understand
- [+] Capture real-life poses
- [-] Expensive to purchase
- [-] Very noisy data
- [-] Requires a lot of cleanup

The Hobbit (2012) Peter Jackson

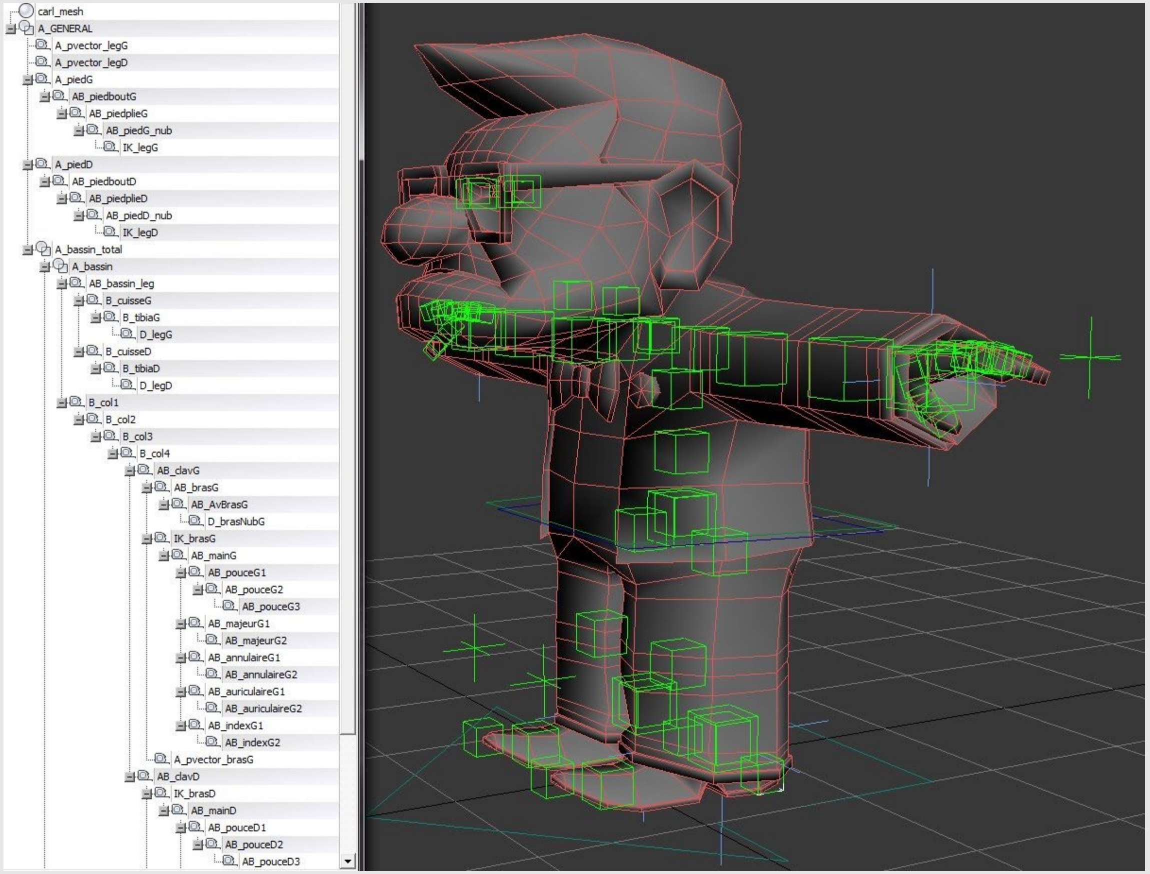
The Human Rig

- Many systems well-described by a kinematic chain (easier to specify constraints)
  - Collection of rigid bodies, connected by joints
  - Joints have various behaviors
    - Ball (shoulder)
    - 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!**



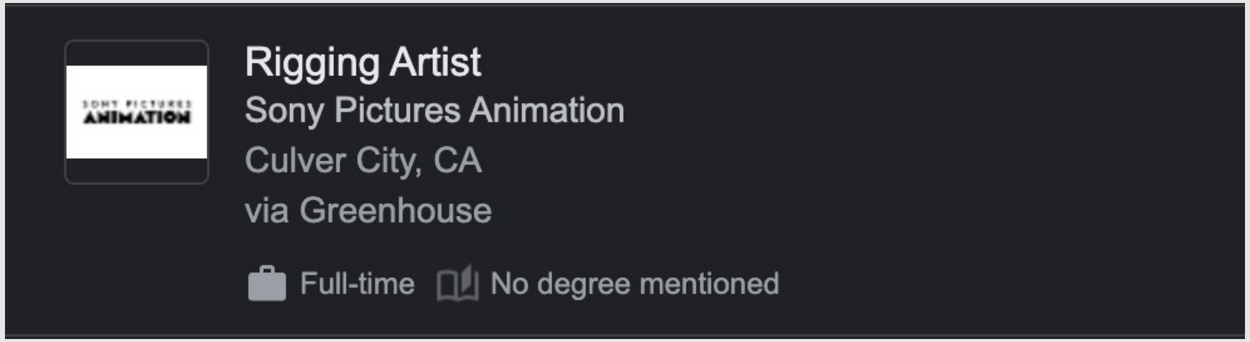
Character Rigging



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



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:

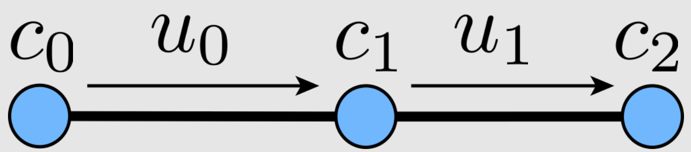
$$p_1 = p_0 + \begin{bmatrix} \cos\theta_0 & \sin\theta_0 \\ -\sin\theta_0 & \cos\theta_0 \end{bmatrix} u_0$$

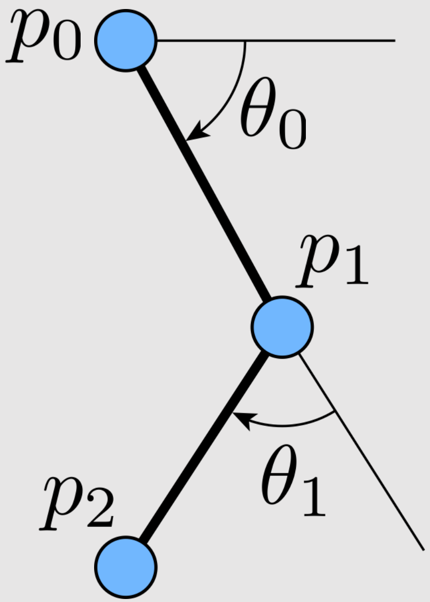
  - Can also be written as:

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

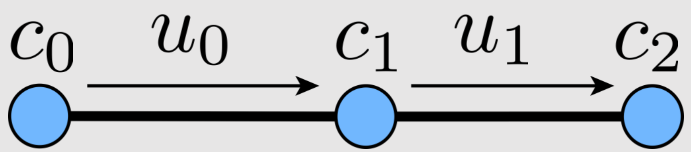
- New hand position p_2 computed as:

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





Forward Kinematics



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

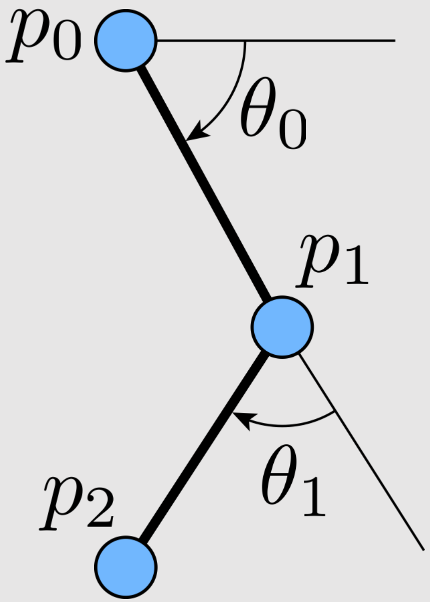
- If we view it as coordinate space transformation, this can also be written as:

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

$$p_1 = T(p_0)\, R(\theta_0) T(u_0) [0, 0, 0]^T$$

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

$$p_2 = T(p_0)\, R(\theta_0)\, T(u_0)\, R(\theta_1) T(u_1) [0, 0, 0]^T$$



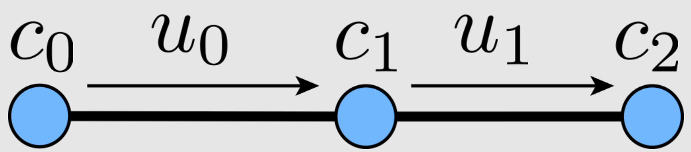
A Note About Spaces

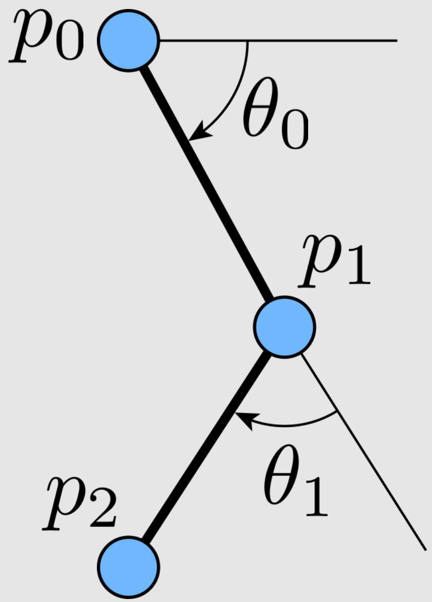
- **World Space:** absolute coordinate space
- **(Skeleton) Local Space**: the model's space
  - Often use the rig's center as the origin
- **Bone Space:** For a given bone i, the origin is the bone's base point and the axes are rotated by its rotations and all the parent rotations before it
  - **Bind 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

$$c_2 = T(p_0)\,T(u_0)\,T(u_1)[0,0,0]^T = B\,[0,0,0]^T$$

  - **Pose Space:** a form of Bone Space, with both rotations and translations applied
    - Think of it as the model that is posed with rotations

$$p_2 = T(p_0)\,R(\theta_0)\,T(u_0)\,R(\theta_1)T(u_1)[0,0,0]^T = P\,[0,0,0]^T$$

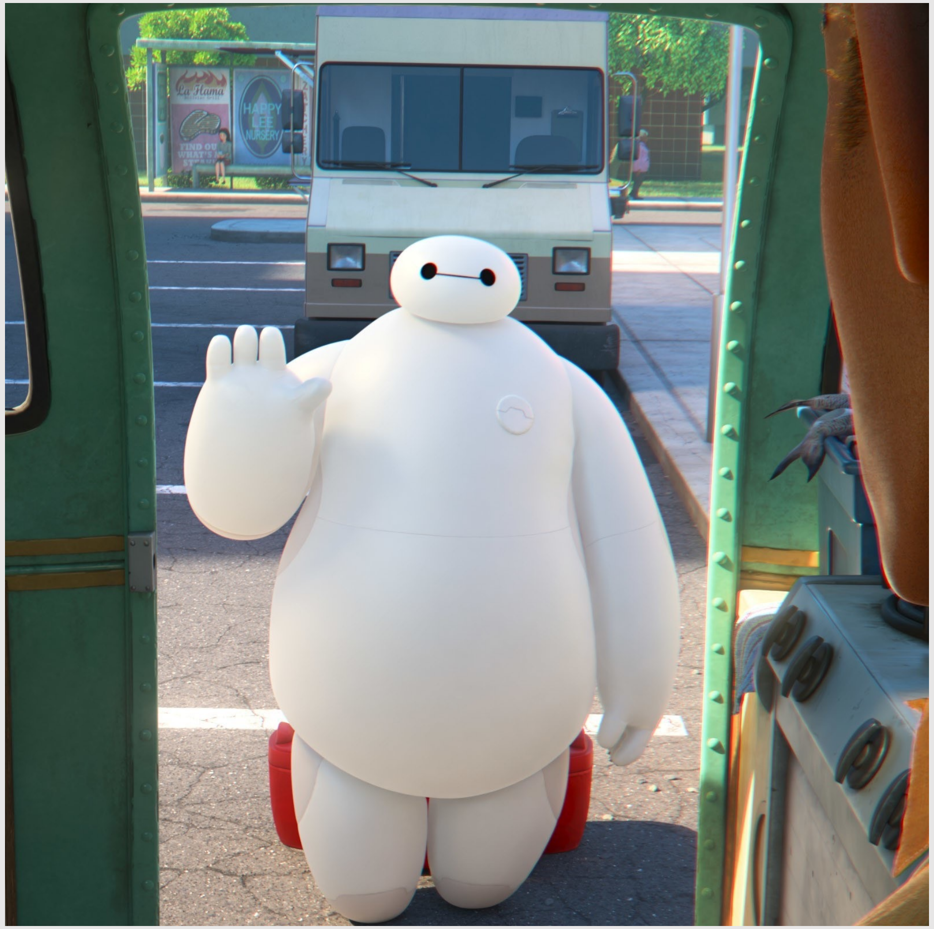




Forward Kinematics

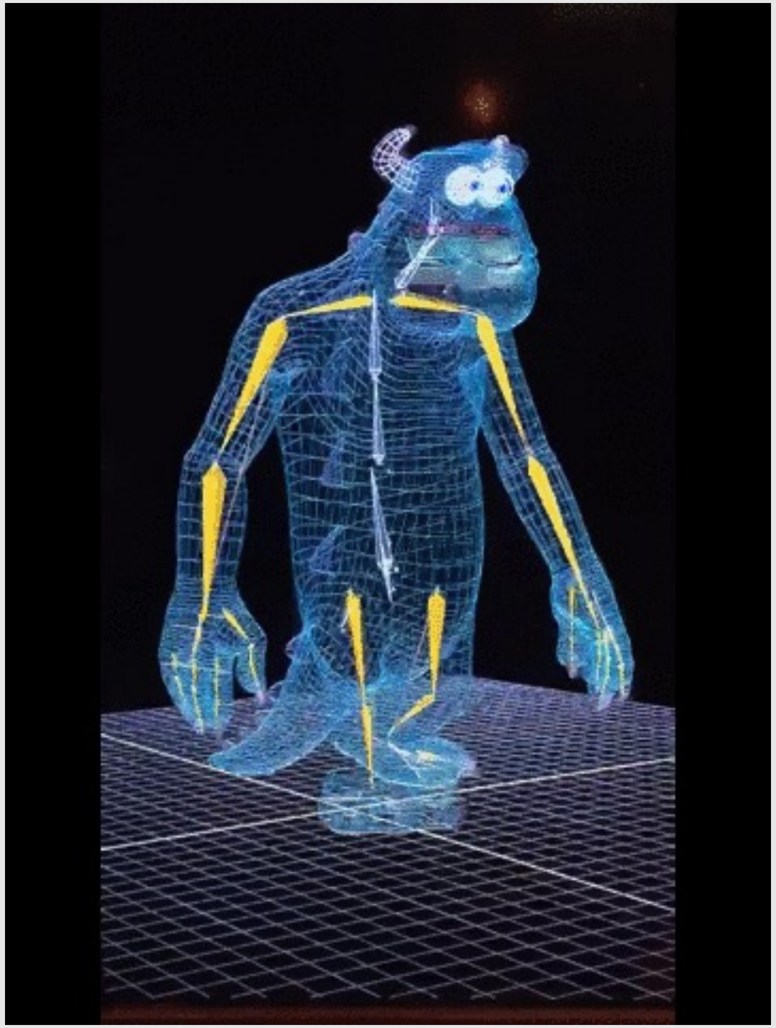
- [+] Computationally efficient
- [+] Easy interface to work with
- [+] Explicit control over every joint
- [-] Produces rigid animations
- [-] Hard to model real-world motions
- [-] Requires more keyframes

- 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

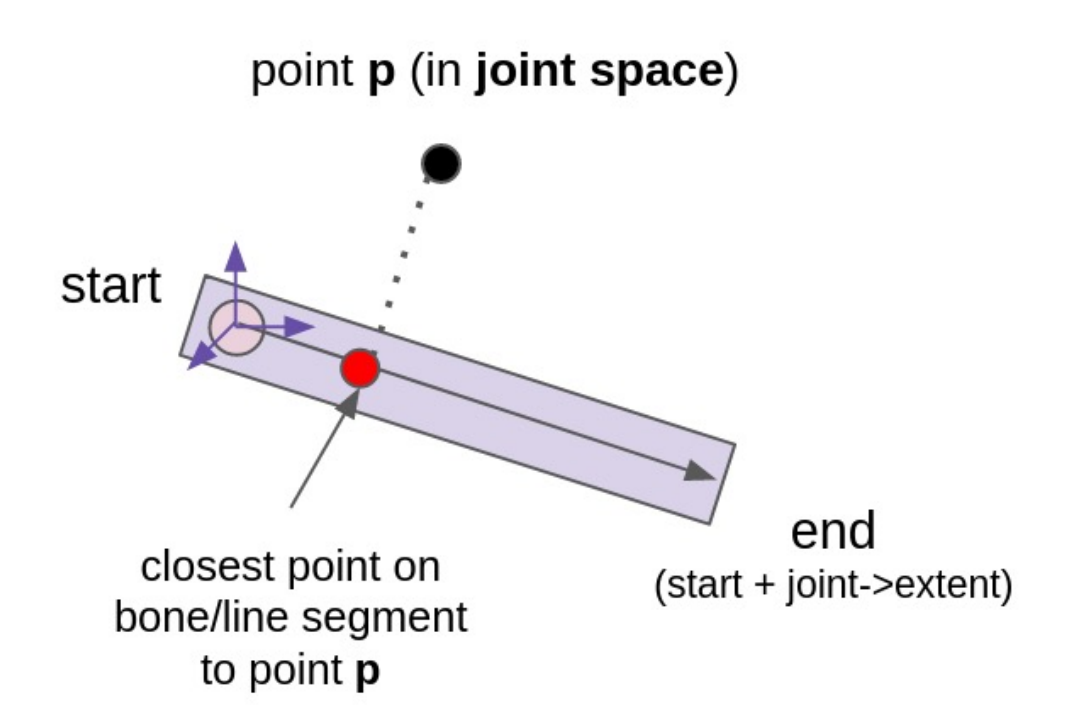
$$\sum_j w_{ij} = 1$$

  - 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

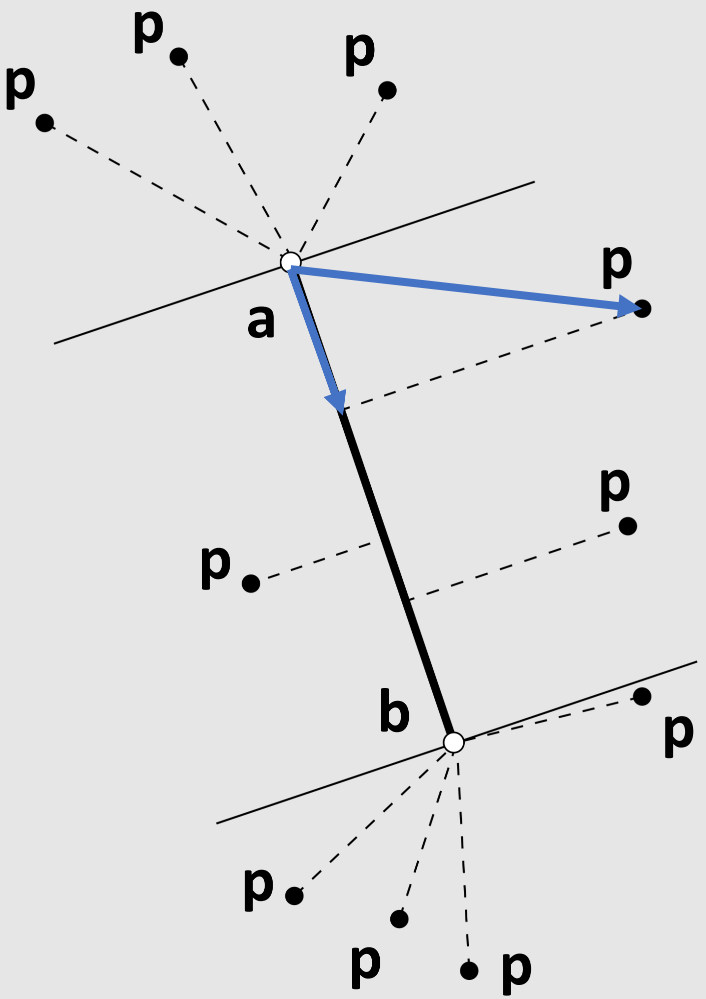
$$\hat{w}_{ij} = \frac{\max(0, r - d_{ij})}{r}$$

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

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

Normalize to get a time

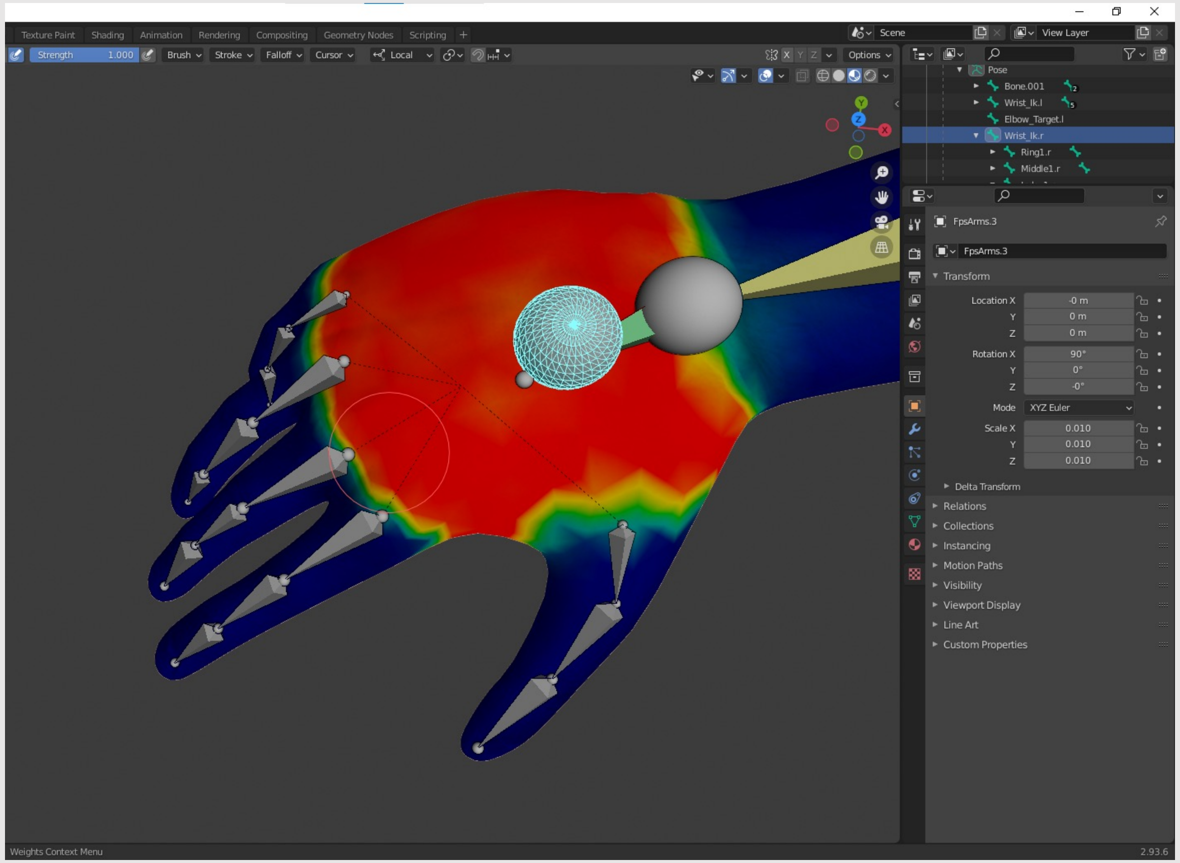
$$t = \frac{\langle \mathbf{p} - \mathbf{a}, \mathbf{b} - \mathbf{a} \rangle}{\langle \mathbf{b} - \mathbf{a}, \mathbf{b} - \mathbf{a} \rangle}$$

Clip time to range [0,1]and interpolate

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

Weight Painting

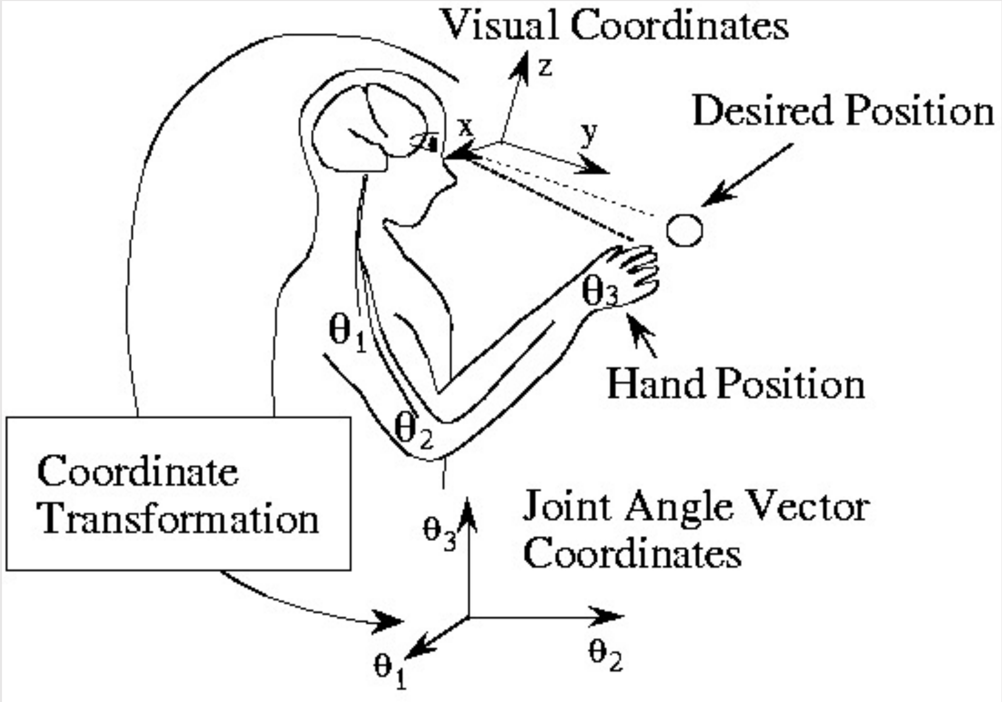
- Computer animation applications also allow you to specify weights on your own
  - Known as **weight painting**
- UI uses color to illustrate magnitude of each vertex/bone pair
- Part of the rigging pipeline
- To obtain smoothly varying weights, can use Laplacian to smooth the field
  - (Recall Special Topics #2)



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…

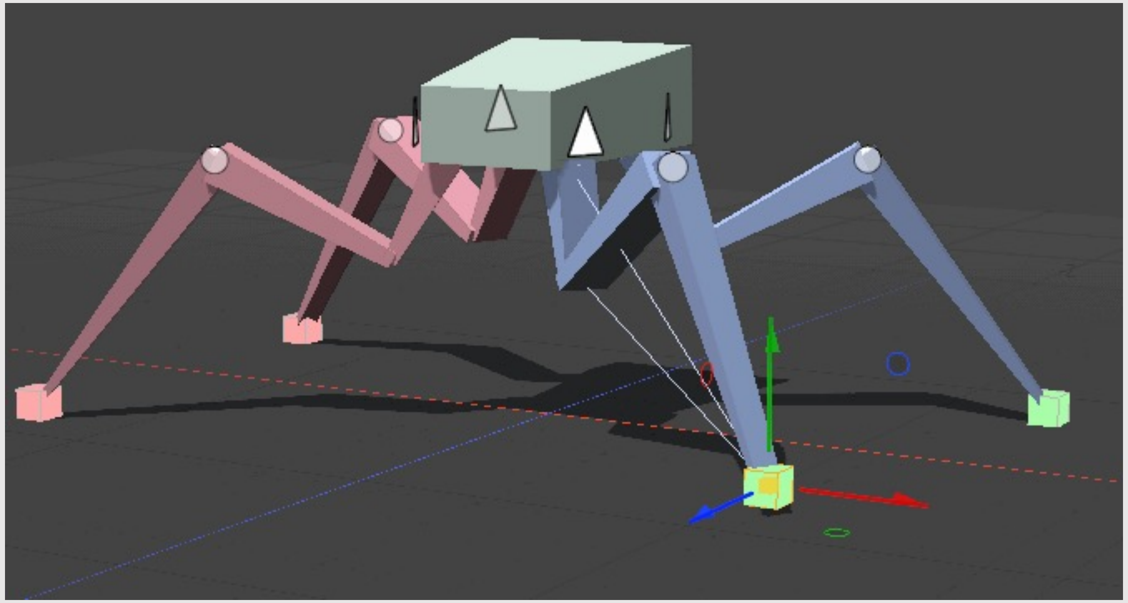
Inverse Kinematics

- 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 step size 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

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

Gradient breaks down into 3 parts:

$$\frac{dp_i}{d\theta_k^y} = R(\theta_0^z)R\left(\theta_0^y\right)R(\theta_0^x)T(u_0)\ldots R(\theta_k^z)\frac{d}{d\theta_k^y}R\left(\theta_k^y\right)R(\theta_k^x)T(u_i)\ldots R(\theta_i^z)R\left(\theta_i^y\right)R(\theta_i^x)u_i$$

[linear transformation] [derivative] [transformed point]

Inverse Kinematic Gradient

To calculate this derivative:

$$\frac{dp_i}{d\theta_k^y} = R(\theta_0^z)R\left(\theta_0^y\right)R(\theta_0^x)T(u_0)\ldots R(\theta_k^z)\frac{d}{d\theta_k^y}R\left(\theta_k^y\right)R(\theta_k^x)T(u_i)\ldots R(\theta_i^z)R\left(\theta_i^y\right)R(\theta_i^x)u_i$$

[linear transformation] [derivative] [transformed point]

Option 1: directly differentiating the rotation matrix:

$$R(\theta_k^y) = \begin{bmatrix} \cos(\theta_k^y) & 0 & \sin(\theta_k^y) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\theta_k^y) & 0 & \cos(\theta_k^y) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\frac{d}{d\theta_k^y}R(\theta_k^y) = \begin{bmatrix} -\sin(\theta_k^y) & 0 & \cos(\theta_k^y) & 0 \\ 0 & 0 & 0 & 0 \\ -\cos(\theta_k^y) & 0 & -\sin(\theta_k^y) & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Inverse Kinematic Gradient

Option 2: use geometric intuition

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:

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

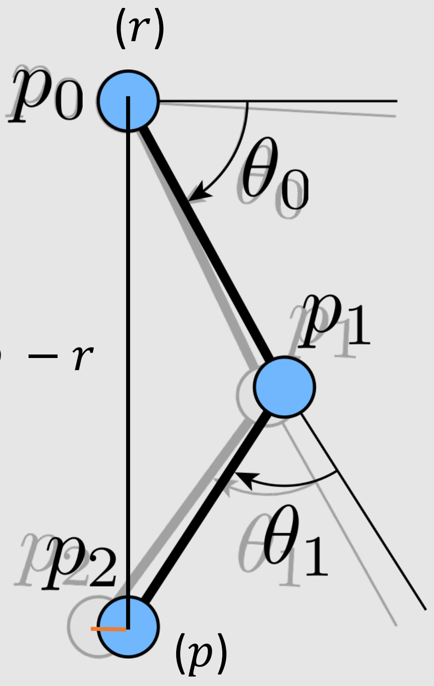
constant for a given handle

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

specific to the current joint

$r =$ **[linear transformation'] [0,0,0]**

$y =$ **([linear transformation'] $[R(\theta_k^z)]$).rotate(θ_k^y)**



y is pointing out of the screen

δp is perpendicular to $p - r$

[linear transformation'] = 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]

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

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

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

$$\ldots$$

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

constant for a given handle

p

r

y

specific to the current joint

Inverse Kinematic Gradient

- Each joint k will have its own vector gradient $\frac{df}{d\theta_k} = < \frac{df}{d\theta_k^x}, \frac{df}{d\theta_k^y}, \frac{df}{d\theta_k^z} >$
  - 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!

$$\nabla f_k^y = \sum_{i,h} (p_i(q) - h) \cdot [y_k \times (p_i(q) - r_k)]$$
$$\nabla f_{k-1}^y = \sum_{i,h} (p_i(q) - h) \cdot [y_{k-1} \times (p_i(q) - r_{k-1})]$$
$$\nabla f_{k-2}^y = \sum_{i,h} (p_i(q) - h) \cdot [y_{k-2} \times (p_i(q) - r_{k-2})]$$
$$\dots$$
$$\nabla f_0^y = \sum_{i,h} (p_i(q) - h) \cdot [y_0 \times (p_i(q) - r_0)]$$

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

- 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