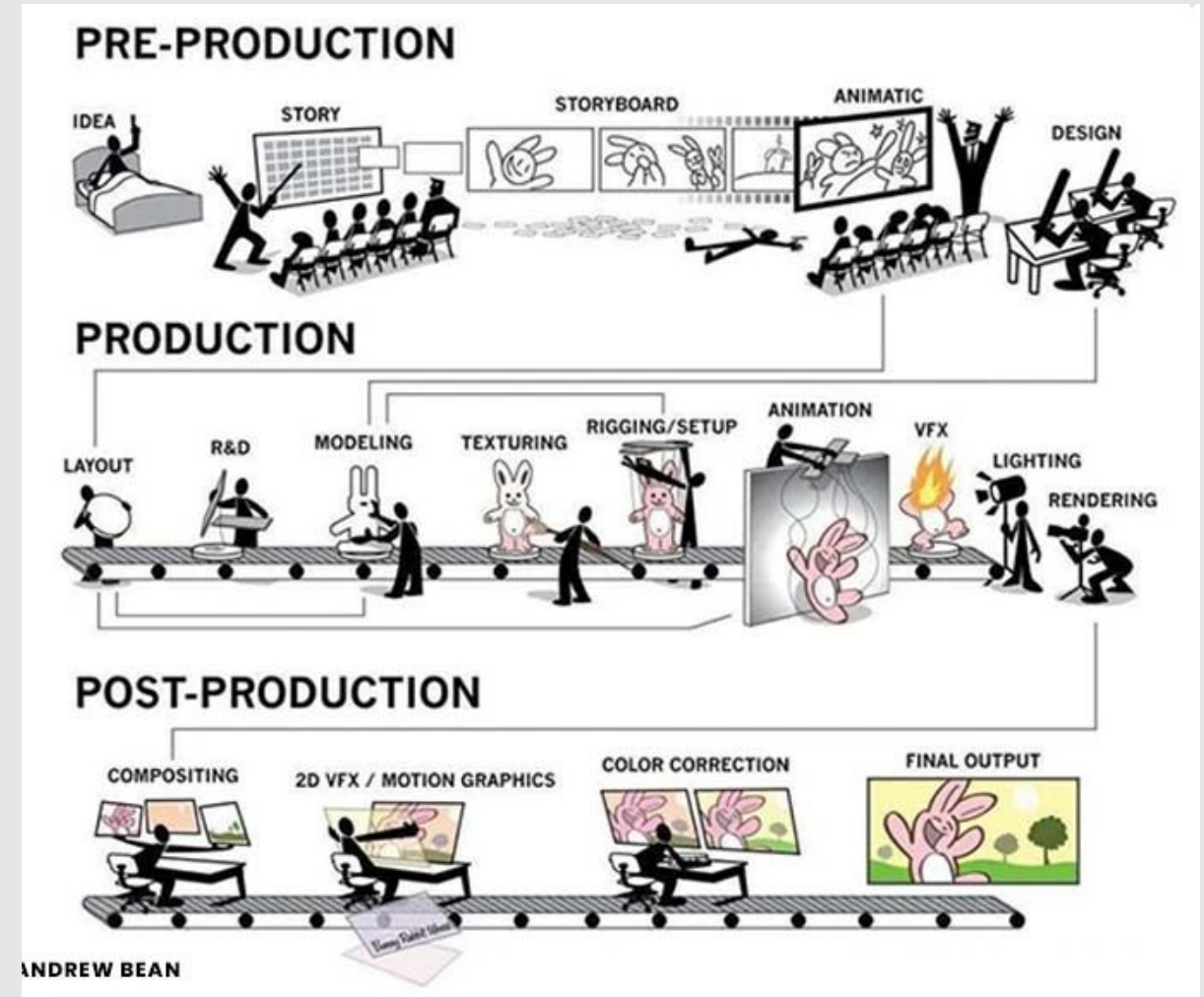


A4: Animation

Welcome to Animation

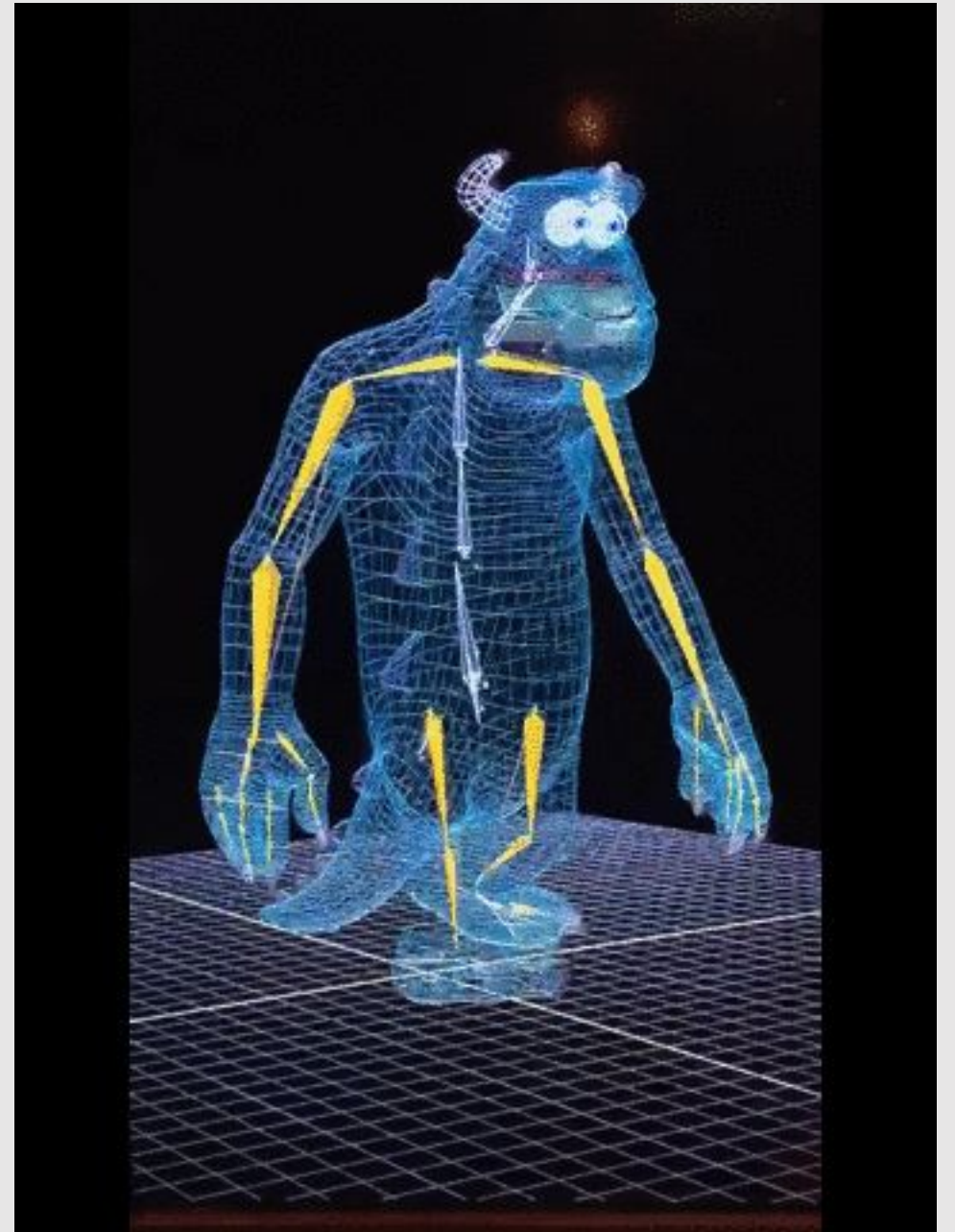
- Want to create photorealistic, fluid and exciting animations without drawing out every image
- Enter computer animation:
 - Create well-defined character models and meshes
 - Set keyframes using kinematics
 - Interpolate between keyframes with splines
 - Use a photorealistic renderer for final results



Real world applications...

Snow castle

©Disney

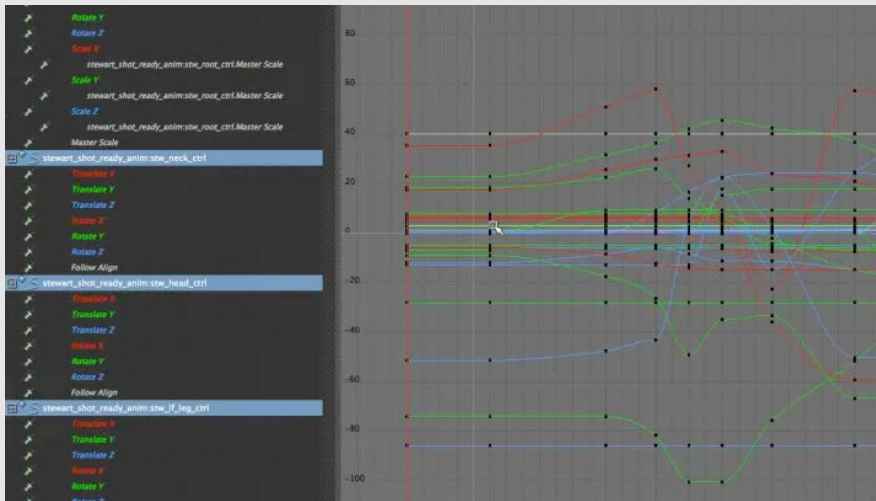
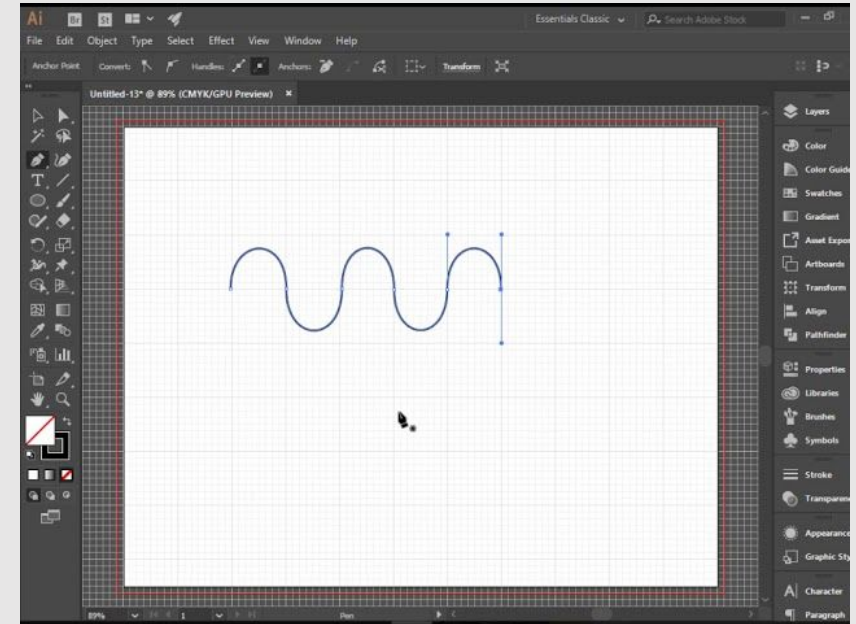


- Spline Interpolation
- Skeleton Kinematics

- Spline Interpolation
- Skeleton Kinematics

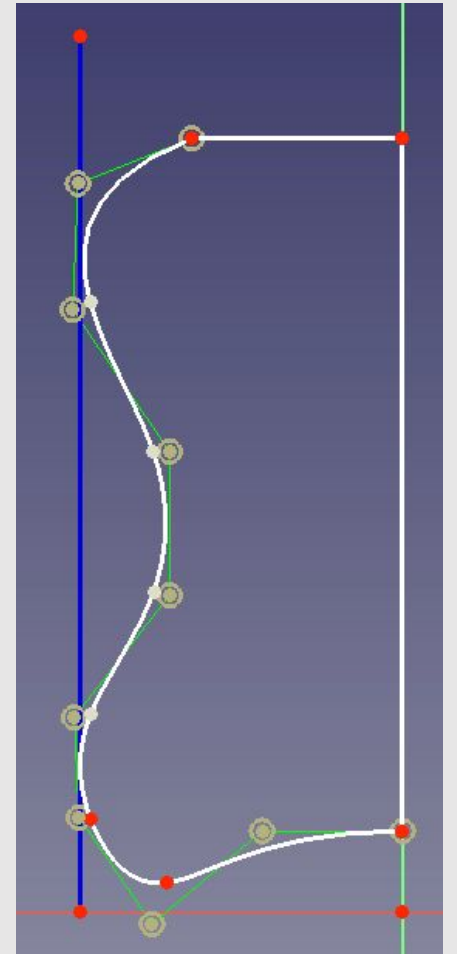
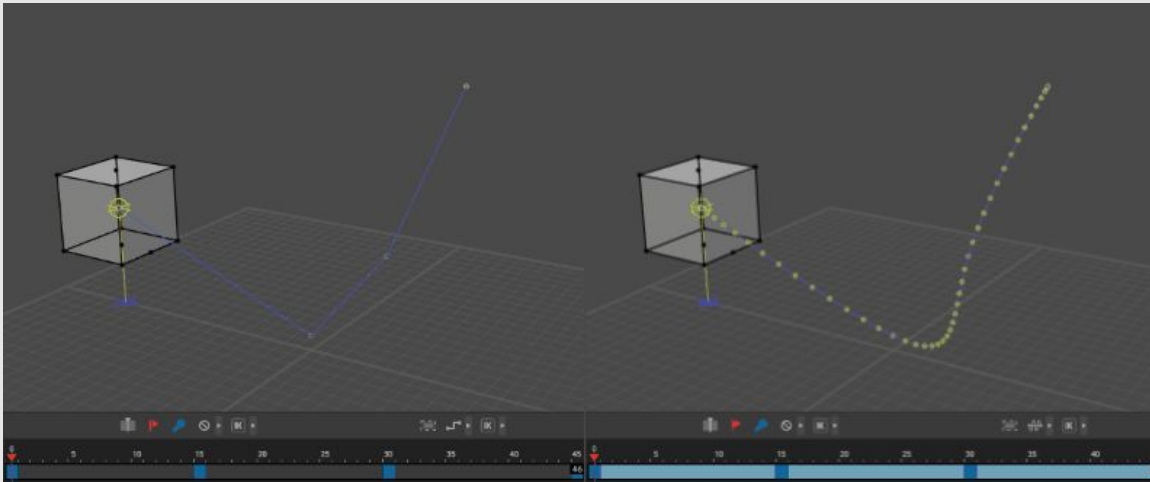
Splines Are Everywhere

- Splines are used in many parts of the animation pipeline
 - Can be used “literally” in designing assets
 - Or can also be used to describe motion of objects when animating
 - And way more...



What Are Splines?

- Splines are **piecewise functions** described by polynomials for each piece
- Think of them as several curves that are connected at their **endpoints**
- Each of these curves can be modified individually without affecting the other curves, as long as the endpoints are still connected



Hermite Splines

- Each cubic piece specified by endpoints and tangents

- Keyframes set at endpoints:

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

- Tangents set at endpoint:

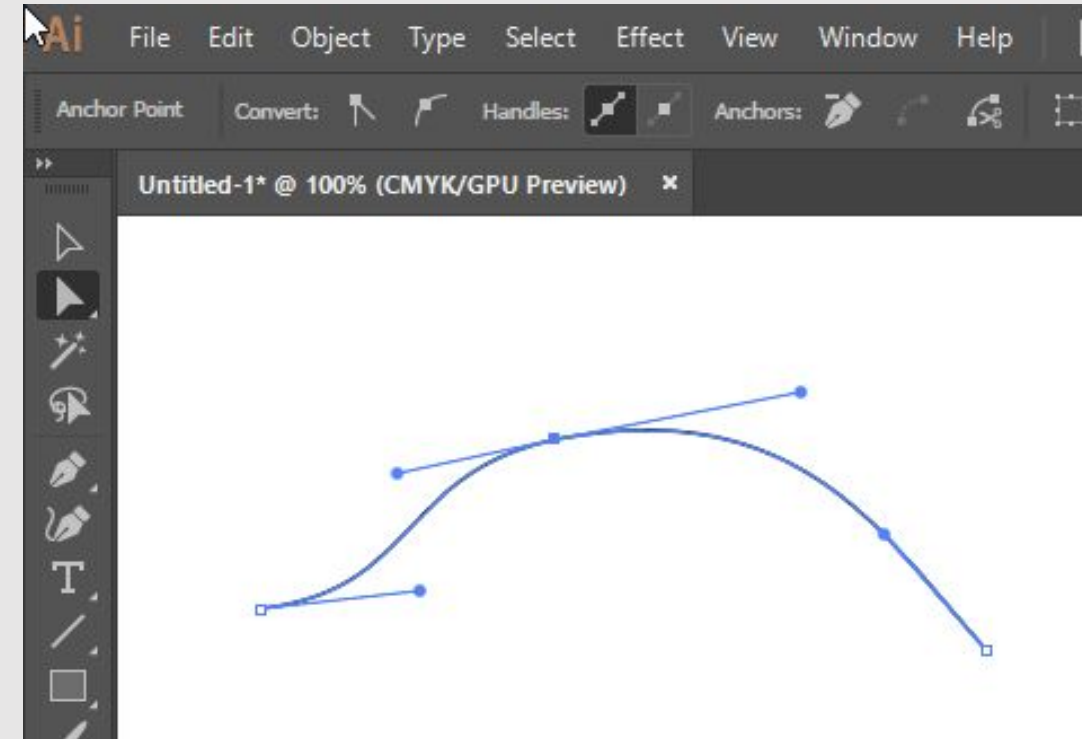
$$p'_i(0) = u_i, \quad p'_i(1) = u_{i+1}, \quad \forall i = 0, \dots, n-1$$

- Total equations:

- $2n + 2n = 4n$

- Commonly used in vector art programs

- Illustrator
- Inkscape
- SVGs



Hermite Splines

Hermite form given by:

$$p(t) = h_{00}(t)p_0 + h_{10}(t)m_0 + h_{01}(t)p_1 + h_{11}(t)m_1$$

where:

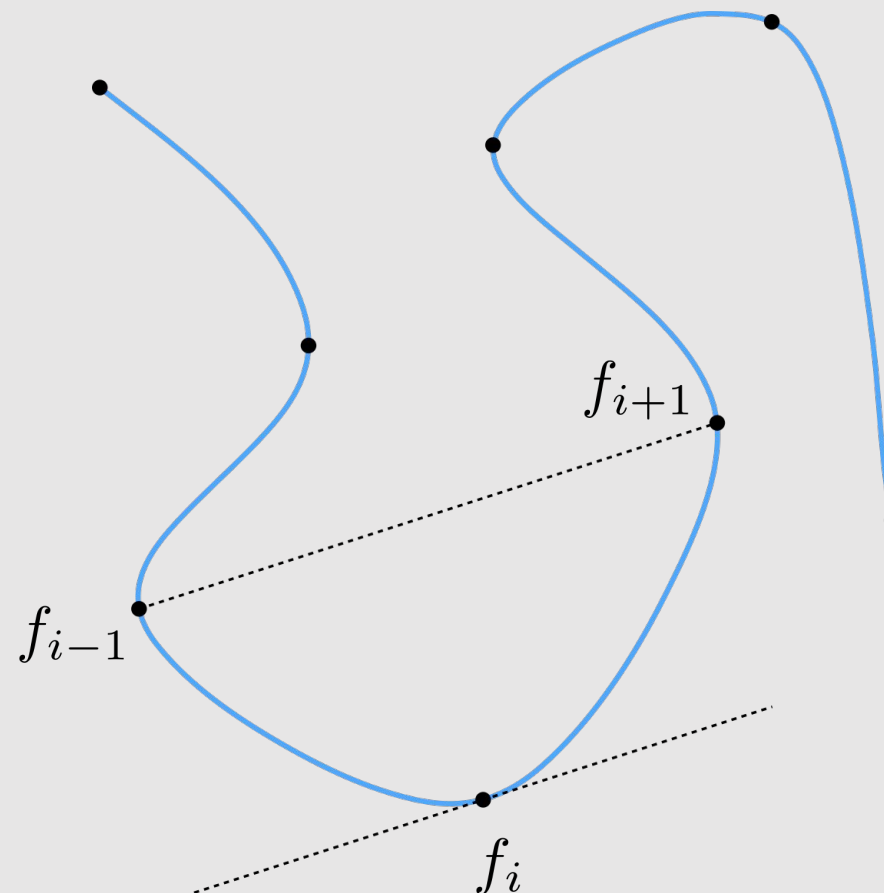
- p_0, p_1 are endpoints
- m_0, m_1 are endpoint tangents (derivatives)
- $h_{ij}(t)$ are cubic hermite basis functions

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:

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

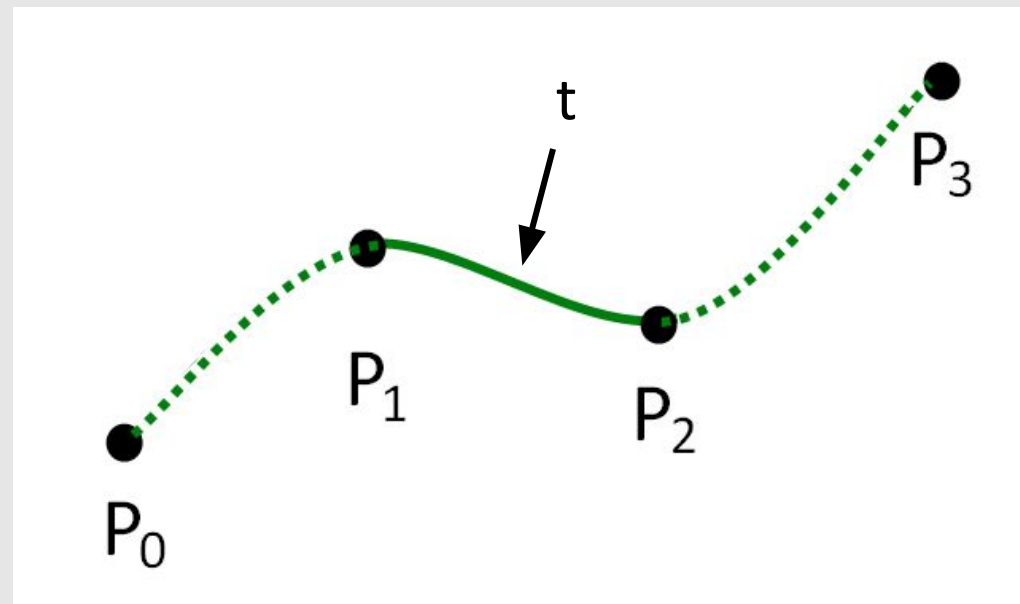
$$h_{00}(t) = 2t^3 - 3t^2 + 1$$

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

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

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

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



Catmull-Rom Splines

- Need to normalize time

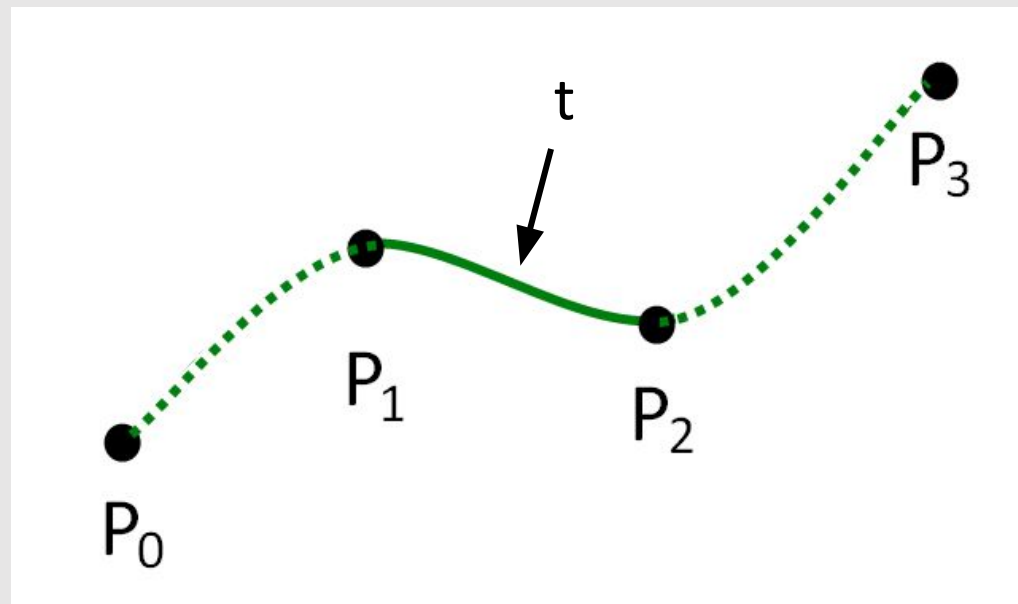
$$t_{norm} = \frac{t - t_1}{t_2 - t_1}$$

- Need to normalize tangents as well

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

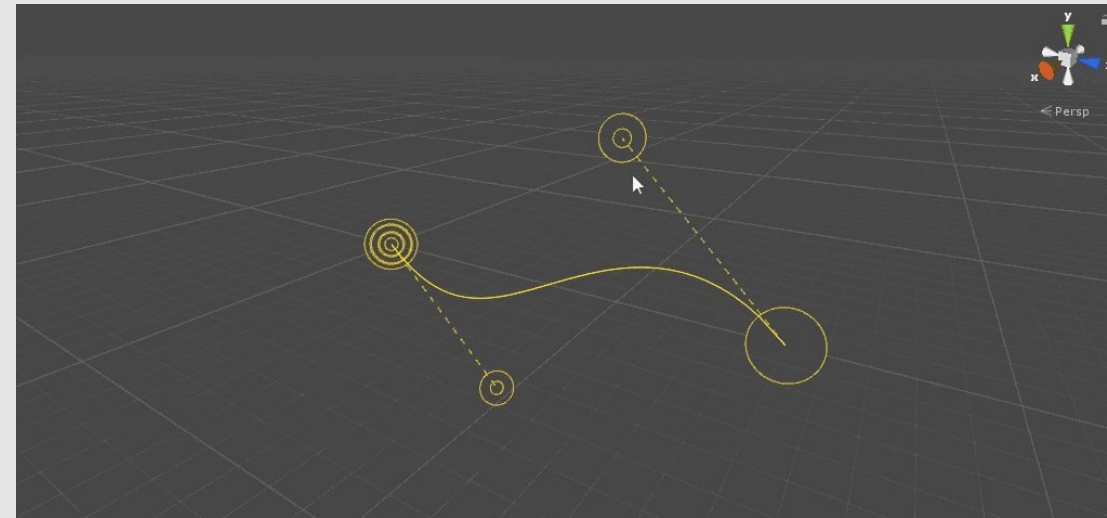
$$u_1 = \frac{p_2 - p_0}{(t_2 - t_1) - (t_0 - t_1)} * (t_2 - t_1)$$

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



Three Main Spline Properties

- Interpolation:
 - Does the spline pass through the control points you specified?
- Continuity:
 - C0: Are the keyframes continuous?
 - C1: Are the first derivatives continuous?
 - C2: Are the second derivatives continuous?
- Locality:
 - Changing one point / part of the curve does not change the entire curve

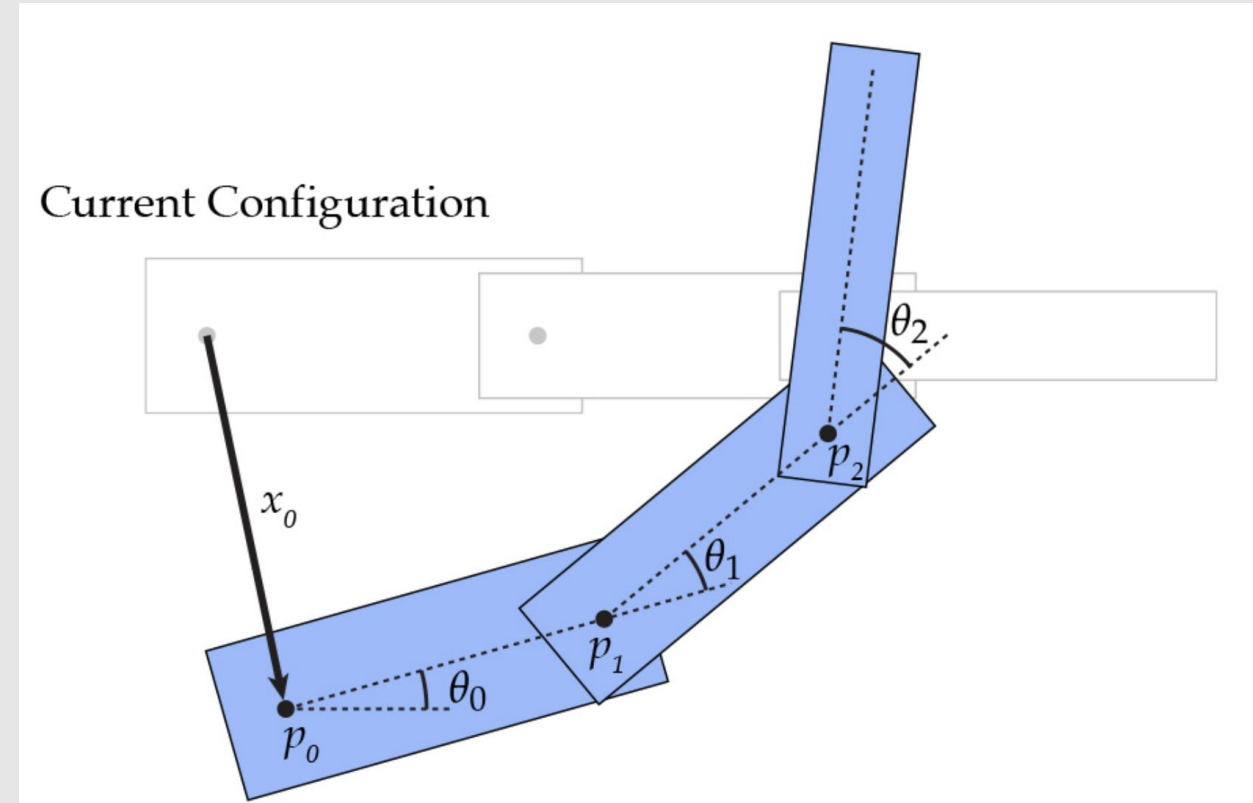


- Spline Interpolation
- **Skeleton Kinematics**

Forward Kinematics

FK is all about calculating the overall transformation for a bone. The core idea is that the transformation applied to a **parent joint** is also applied to all of its **children** joints. This results in a *chain* of transformations.

- “**Bind**” position: the rotation should be zero
- “**Posed**” position: take into account the “pose” of the joint (euler angle)



Bind vs Posed Position

Feature	Bind Position (Bj)	Posed Position (Pj)
Rotation	Should be zero (pose = (0,0,0)).	Takes into account the pose (Euler angle).
Transform	Pure translation by the parent's extent	Rotation (XYZ order) followed by translation by the parent's extent
Implementation	<code>Skeleton::bind_pose</code>	<code>Skeleton::current_pose</code>

A Note About Spaces

- Bind-to-Local:

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

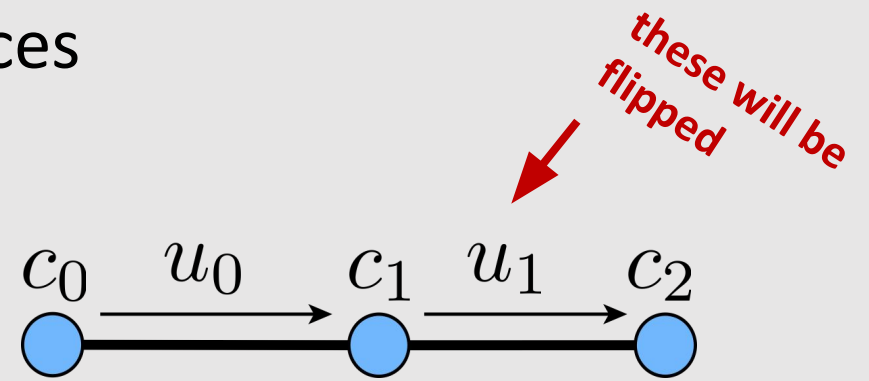
- Pose-to-Local:

$$p_0 = R(\theta_0) T(u_0) R(\theta_1) T(u_1) R(\theta_2) p_2$$

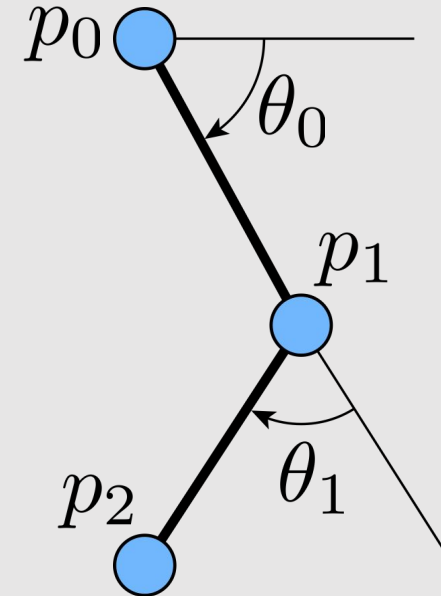
we give you `compute_rotation_axes`
which can be used to calculate this rotation
matrix

Rotations and transformations will be saved as
child-to-parent

- No need to invert



need to undo
 p_2 's orientation



Inverse Kinematics

Idea: move the skeleton towards target point using gradient descent

$$X_{k+1} = X_k - \tau \nabla f$$

$$f(\theta(t)) = \frac{1}{2} |p(\theta(t)) - q|^2$$

Procedure:

- Skeleton::gradient_in_current_pose
 - first calculate the jacobian via $(J_\theta)_i = \vec{r} \times \vec{p}$
 - then approximate gradient using jacobian via $\nabla_\theta f \approx \alpha J_\theta^T (p(\theta) - q)$
- Skeleton::solve_ik
 - Use gradient descent to calculate the joints' pose at the next time step
 - See intro to optimization lecture

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

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

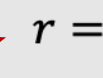
Fun fact: by transforming the axis of rotation and base point to local coordinates, Then the derivative of the rotation $R(\theta_k^y)$ by amount θ_k^y around axis y and center r of point p becomes:

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

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

```
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 Descent

Steps:

- Call `gradient_in_current_pose()` to compute $d \text{ loss} / d \text{ pose}$
- Update positions of all the bones by the computed gradients
- Loop through each handle and calculate the loss
 - loss is $\sum_{(i,h)} \frac{1}{2} |p_i(q) - h|^2$
- If at a local minimum (e.g., gradient is near-zero), return 'true'
- If run through all steps, return 'false'