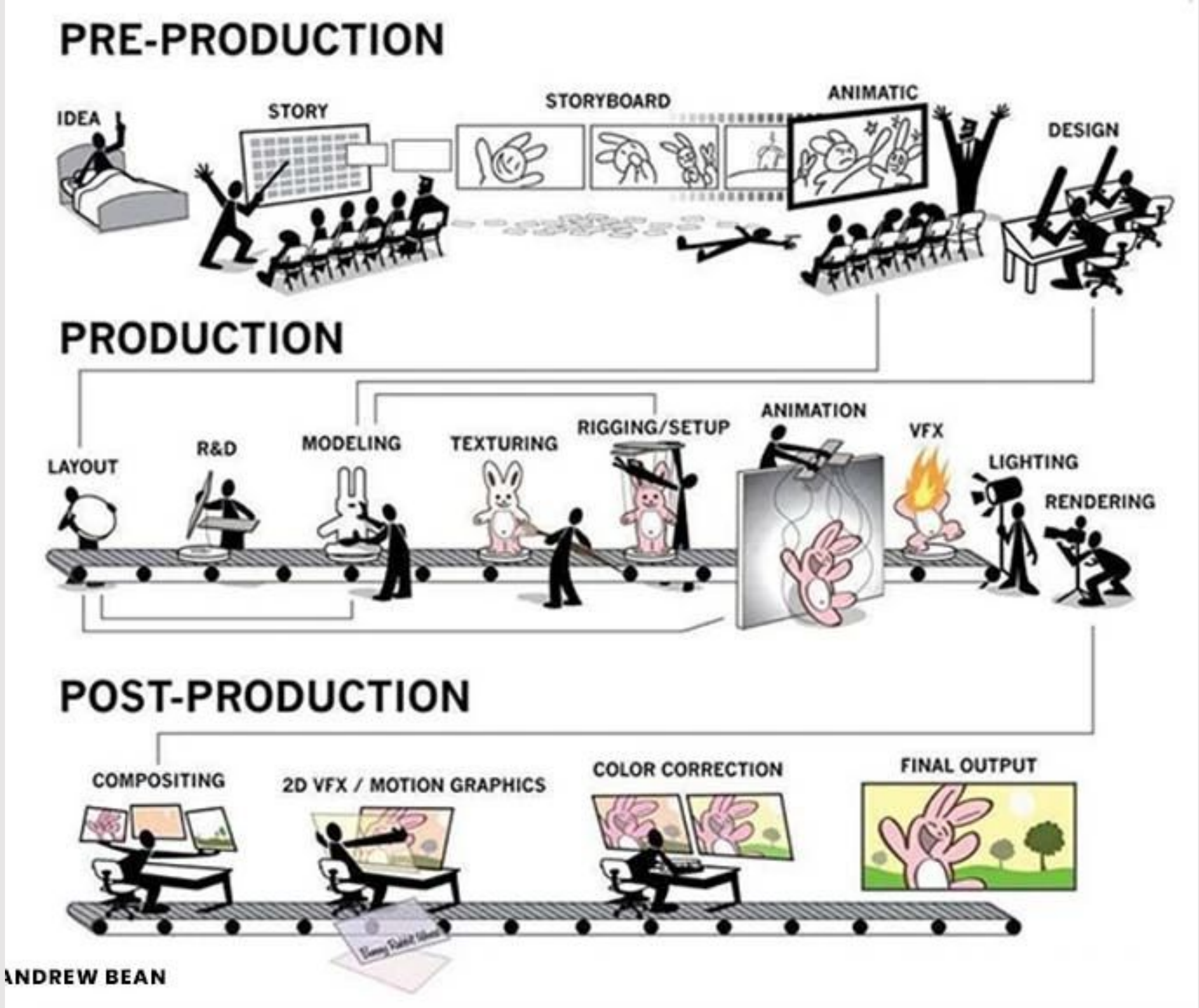
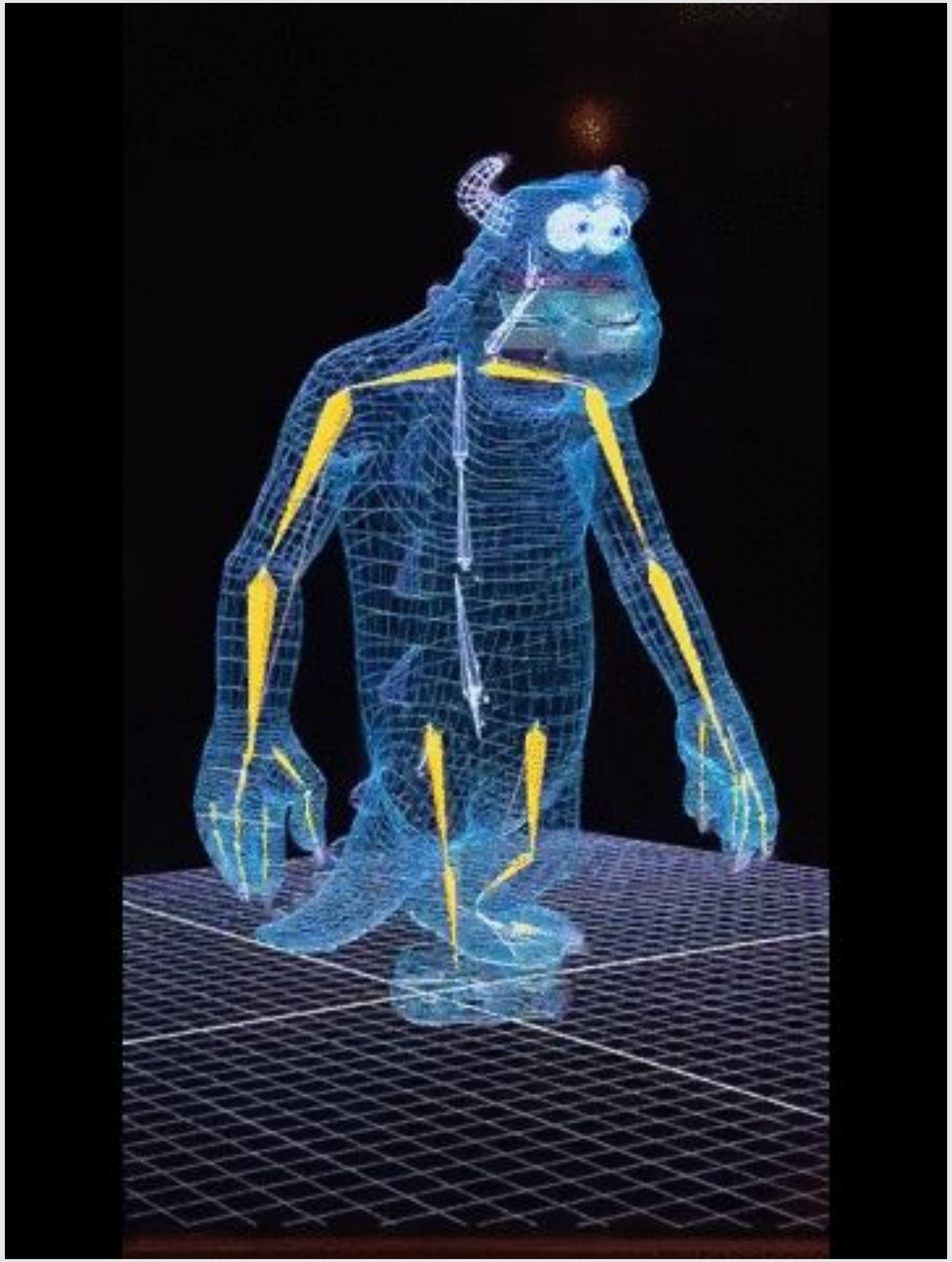
A4: Animation

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…

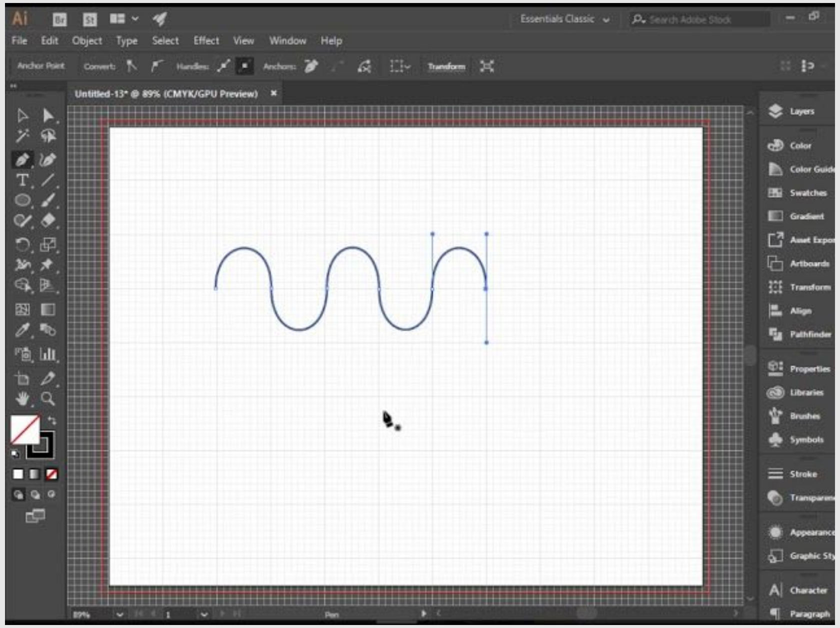


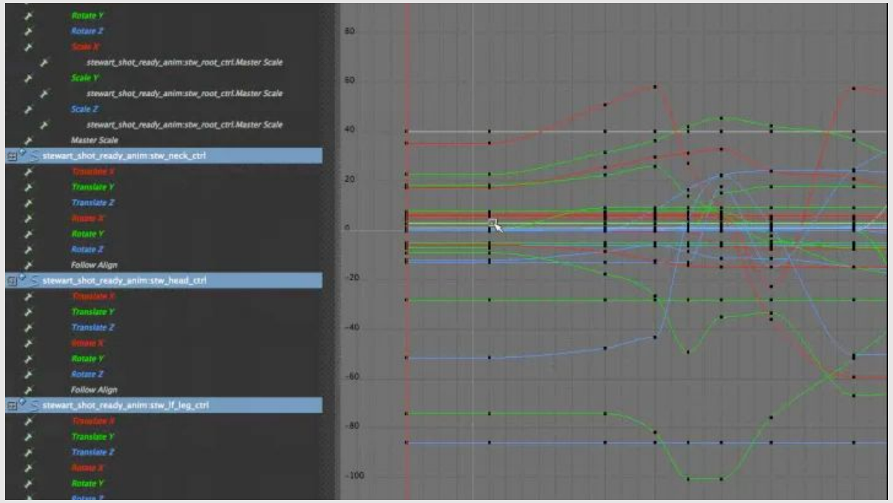
- Spline Interpolation
- Skeleton Kinematics
- Linear Blend Skinning
- Particle Simulation

- Spline Interpolation
- Skeleton Kinematics
- Linear Blend Skinning
- Particle Simulation

Splines Are Everywhere

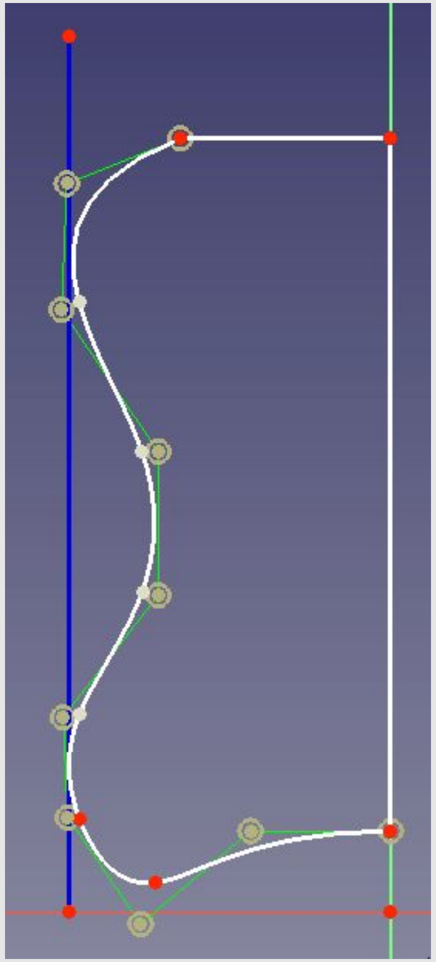
- Splines are used in many parts of the animation pipeline
  - Can be used “literally” in designing assets
  - Or can also be used to describe motion of objects when animating
  - Any way more…

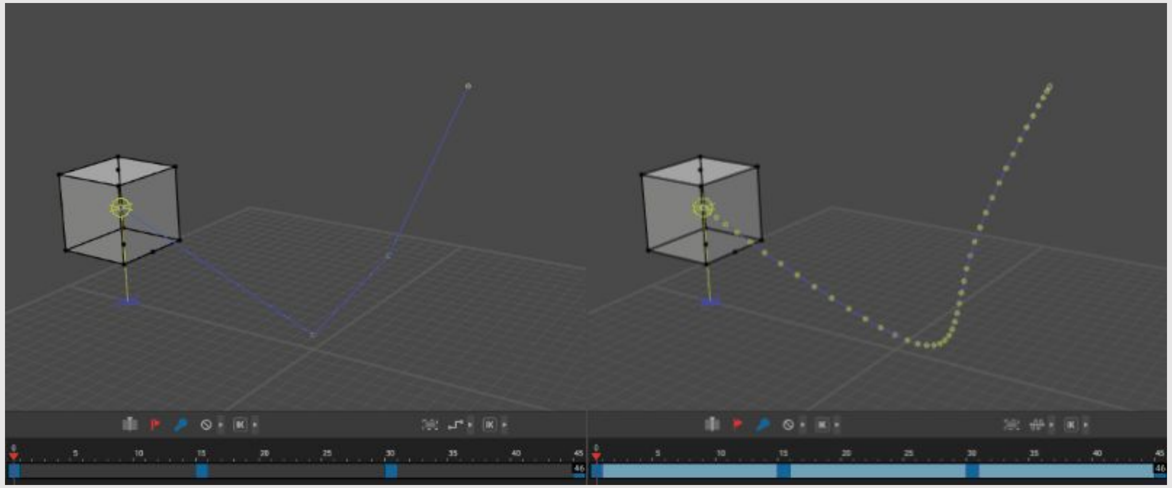




What Are Splines?

- 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



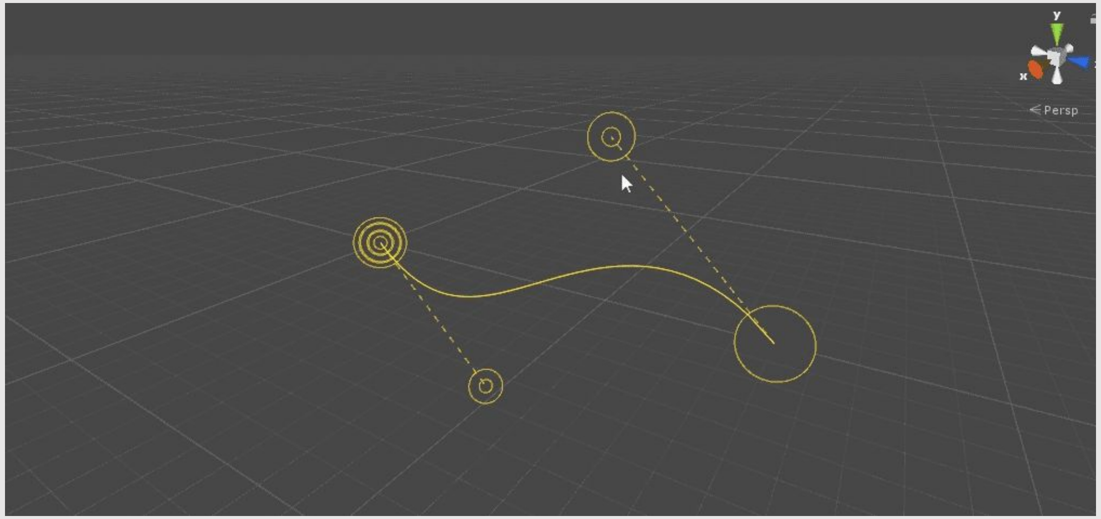


What Type of Spline Should We Use?

- Types of Splines:
  - Natural Spline
    - A series of piecewise cubics
  - Hermite Spline
    - Each piece is defined by its endpoints and the tangent to the curve
  - Bezier Spline
    - Similar to Hermite, except Bezier defines “control points” that change the curve instead of moving the tangent line
  - Catmull-Rom Spline
    - You specify keyframes (i.e. points that you want to go through) and you use a basic formula to compute the tangents
  - B-Spline
    - Define keyframes (like Cat-Rom) and take a weighted average of nearby keyframes to interpolate

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



What Type of Spline Should We Use?

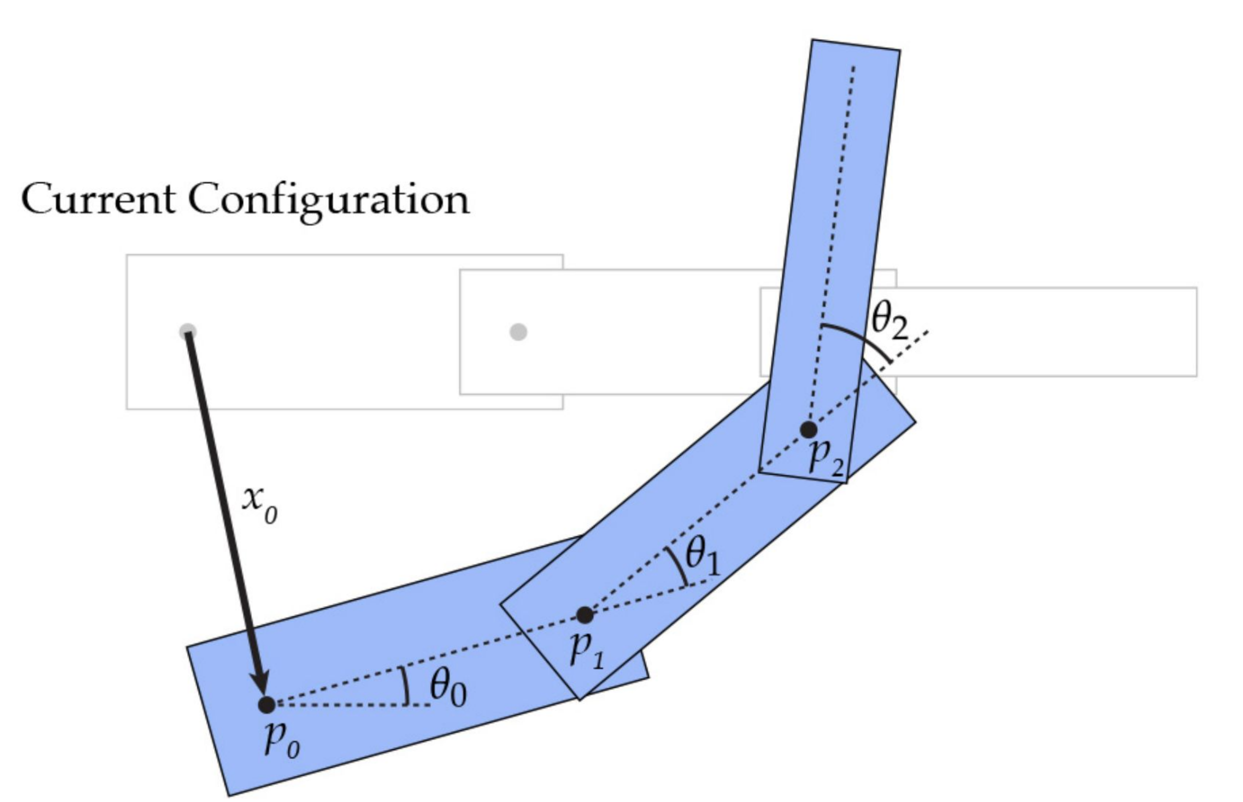
	[Interpolation]	[Continuity]	[Locality]
Linear	✓	✗	✓
Natural	✓	✓	✗
Hermite	✓	✗	✓
Bezier	✓	✗	✓
Catmull-Rom	✓	✗	✓
B-Spline	✗	✓	✓

- Spline Interpolation
- Skeleton Kinematics
- Linear Blend Skinning
- Particle Simulation

Forward Kinematics

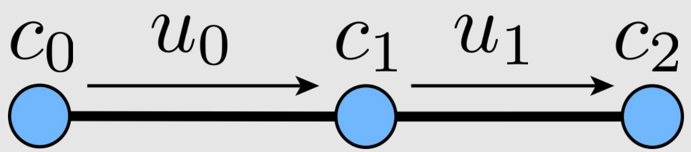
Idea: transformation applied to parent joint is also applied to the children joint

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



A Note About Spaces

these will be flipped



- Bind-to-Local:

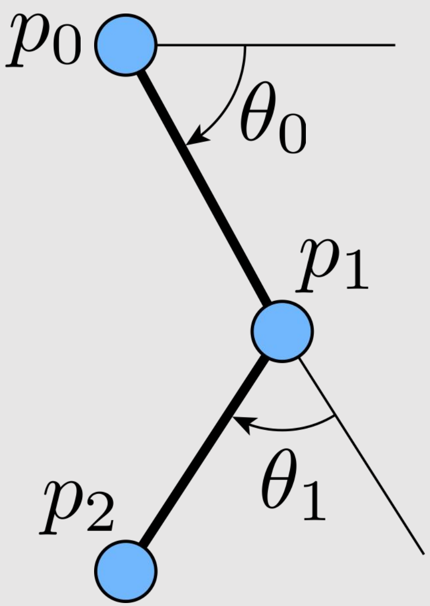
$$c_0 = T(u_0)\,T(u_1)\,c_2$$

need to undo p2's orientation

- 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

Inverse Kinematics

Idea: move the skeleton towards target point using gradient descent

$$X_{k+1} = X_k - \tau \nabla f \qquad f(\theta(t)) = \frac{1}{2}|p(\theta(t)) - q|^2$$

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

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

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

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

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

$p =$ **[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)**

[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

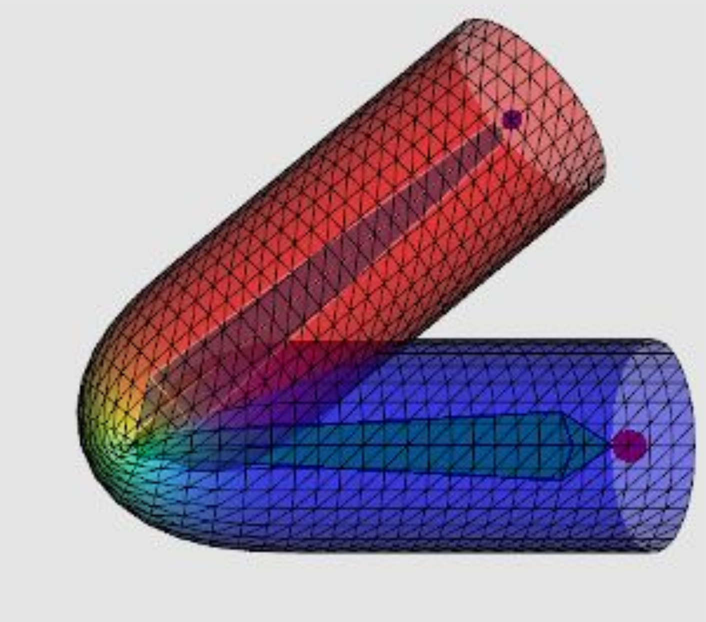
Steps:

- Call `gradient_in_current_pose()` to compute d loss / d 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`

- Spline Interpolation
- Skeleton Kinematics
- Linear Blend Skinning
- Particle Simulation

Linear Blend Skinning

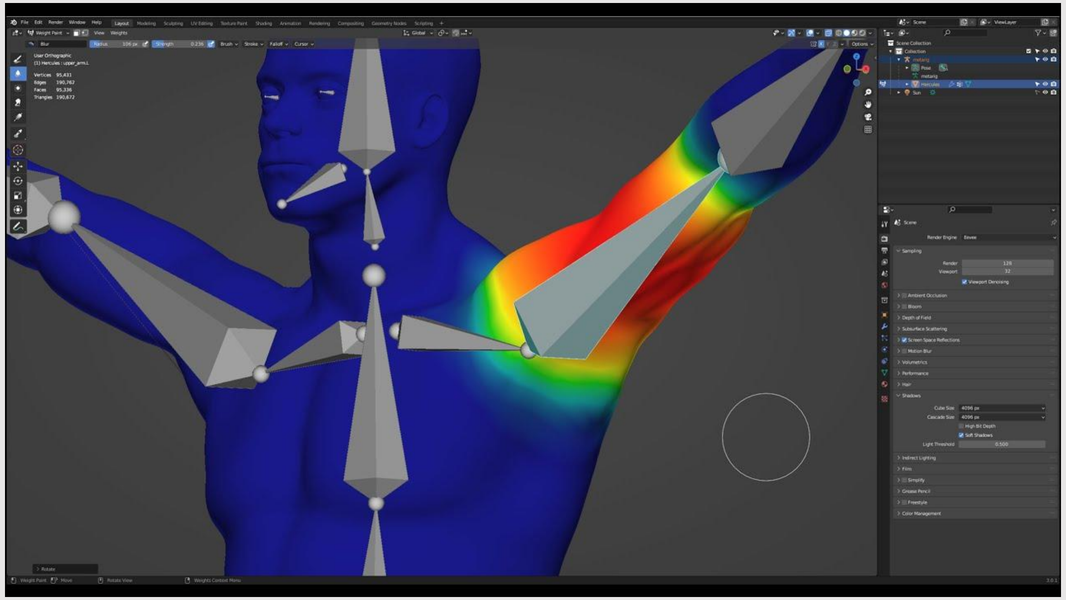
Motivation: We control how much the mesh geometry moves as the bones rotate by assigning each vertex a "weight" per bone



Linear Blend Skinning

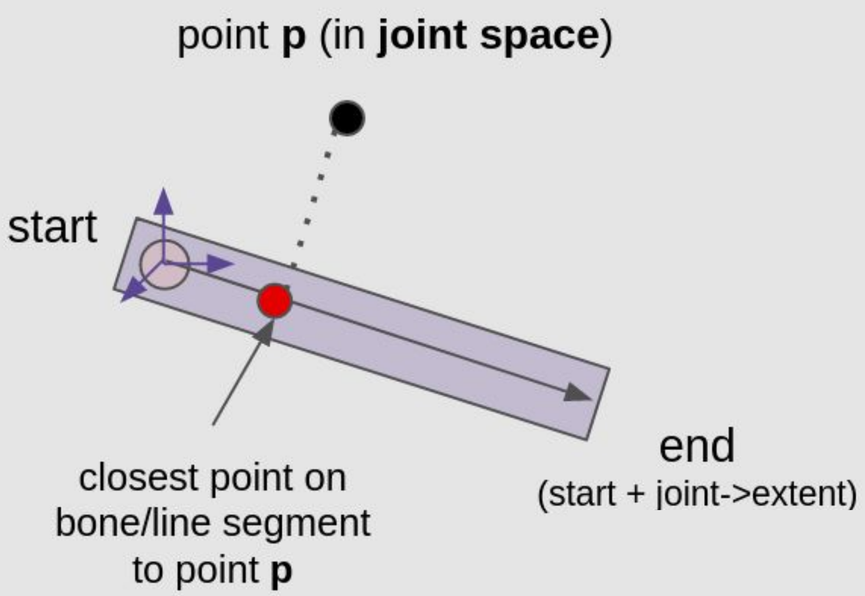
There are many ways one can assign these weights

- Manually - assign weights by having an artist “paint” the weights with a 3D program
- Automatically/Algorithmically
  - One method is Linear Blend Skinning, where we assign weights inversely proportional to the distance between the vertex and the bones



Linear Blend Skinning

Weights assigned via inverse distance from vertex to the bone (represented by a line segment) up to a max distance define by bone::radius



For weight of vertex i with bone j,
and distance between the two given by d:

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

Note : need to normalize per vertex so all weights add to one!

Linear Blend Skinning

New vertex positions are thus a weighted sum of transformations under the bone transformations

- Transformations are from bind space (B) to pose space (P)

$$v_i' = (\sum_i w_{ij} P_j B_j^{-1}) v_i$$

- Spline Interpolation
- Skeleton Kinematics
- Linear Blend Skinning
- Particle Simulation

Particles in Scotty3D

- Non-self-interacting, physics-simulated, spherical particles that interact with the rest of the scene
- Can use basic physics to simulate where a particle will be in the scene at a given time

$$F = ma$$

$$a = \frac{dv}{dt}$$

$$\frac{dv}{dt} = \frac{F}{m}$$

$$\frac{dx}{dt} = v$$

Particles in Scotty3D

- Much easier to update position/velocity at small time-steps then continuously over a large time period
- Forward Euler
  - Can be unstable and does not conserve energy in a system – but that's ok for now

$$x_{t + \Delta t} = x_t + v_t \cdot \Delta T$$

$$v_{t + \Delta t} = v_t + a \cdot \Delta T$$

Particle Collisions

- Can use Scotty3D's ray tracing (that you implemented!) to detect if particle collides with the scene within a timestep
  - Assume all particles collide elastically (i.e. a particle's velocity should be the same before and after a collision, with its direction reflected based on the surface normal)

  - Create a ray based on the particle's position and velocity
  - If the particles hits the scene within the current timestep
    - Reflect the velocity
  - Update velocity and position based on current timestep
  - Repeat until entire timestep is consumed

Particle Collisions

- **If the particles hits the scene within the current timestep**
  - Reflect the velocity

- Note this is not as simple as checking if the ray hits something in the scene