Interactive Computer Graphics: Lecture 16

Animation and Kinematics

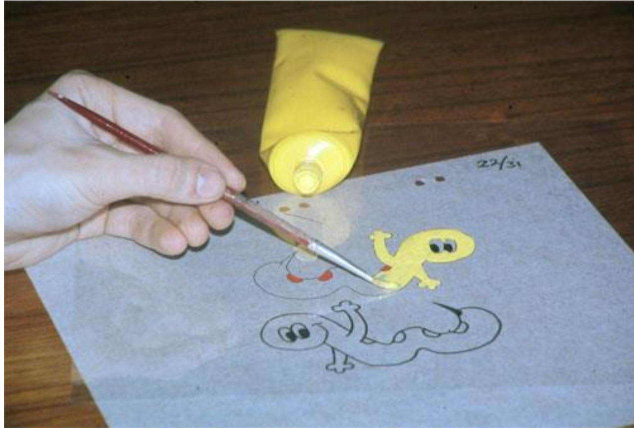
Some slides adopted from
Daniel Wagner, Michael Kenzel, TU-Graz
Duncan Gilles, Imperial
Seth Teller, MIT
Steve Rotenberg, UCSD

Animation of 3D models

In the early days physical models were altered frame by frame to create animation - eg King Kong 1933.

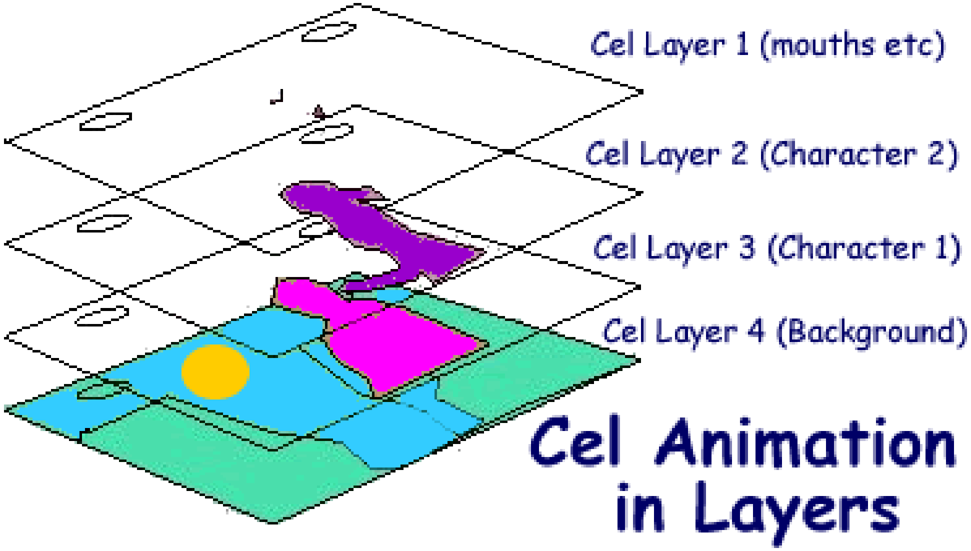
Computer support systems for animation began to appear in the late 1970, and the first computer generated 3D animated full length film was Toy Story (1995).

Conventional Animation



http://commons.wikimedia.org/

- Draw each frame of the animation
  - great control
  - tedious
- Reduce burden with cel animation
  - layer
  - keyframe
  - inbetween
  - ...



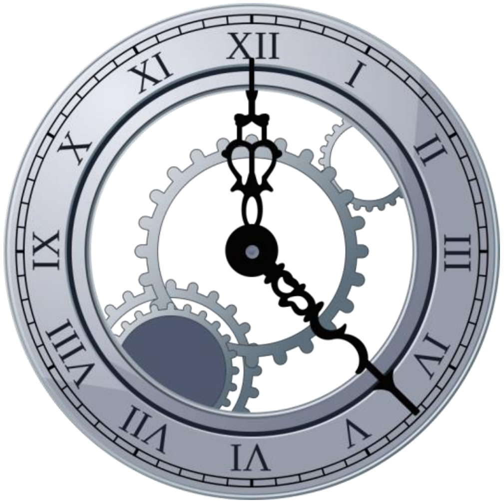
http://www.cybercomputing.co.uk/ICT/Design/celdesign.htm

Cel Animation

Sheila Graber via youtube

Computer-Assisted Animation

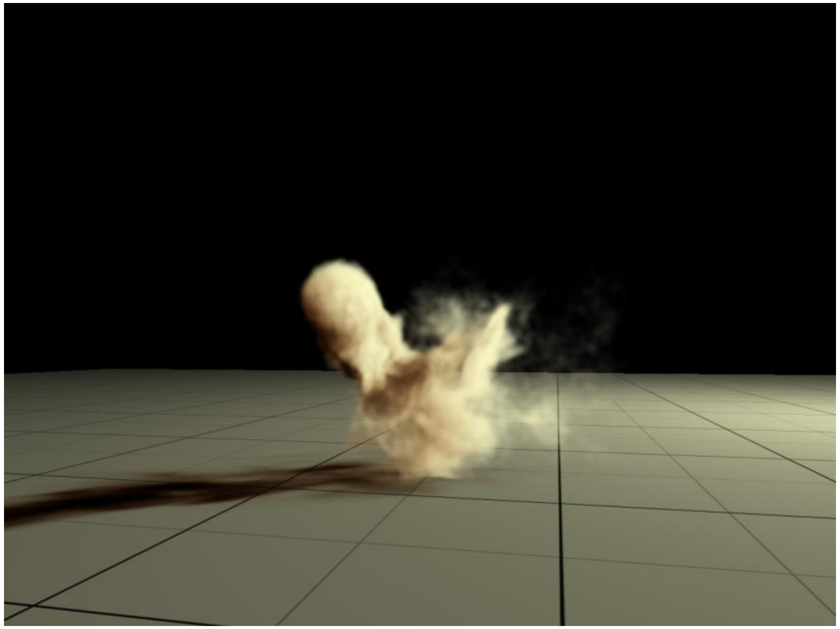
- Procedural animation
  - describes the motion algorithmically
  - express animation as a function
  of small number of parameters
  - Example: a clock with second, minute and hour hands
    - hands should rotate together
    - express the clock motions in terms of a “seconds” variable
    - the clock is animated by varying the seconds parameter



openclipart.org

Computer-Assisted Animation

- Physically Based Animation
  - Assign physical properties to objects (masses, forces, inertial properties)
  - Simulate physics by solving equations
  - Realistic but difficult to control



Computer-Assisted Animation

- Motion Capture
  - Captures style, subtle nuances and realism
  - You must observe someone do something





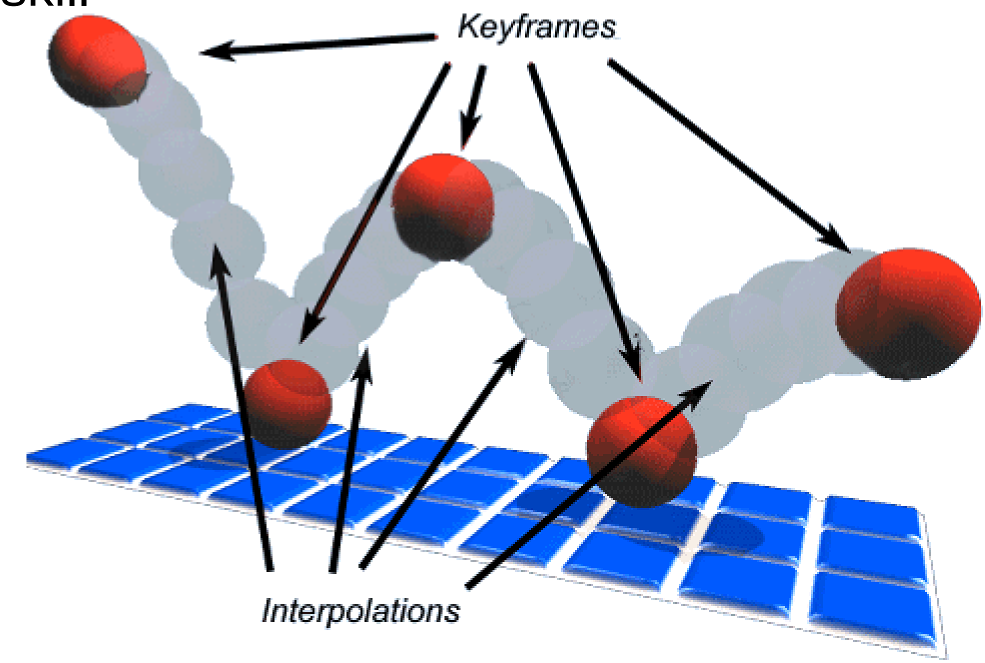
https://www.comp.nus.edu.sg/~leowwk/d-culture/d-culture.htm

Motion Capture

Full Sail University via youtube

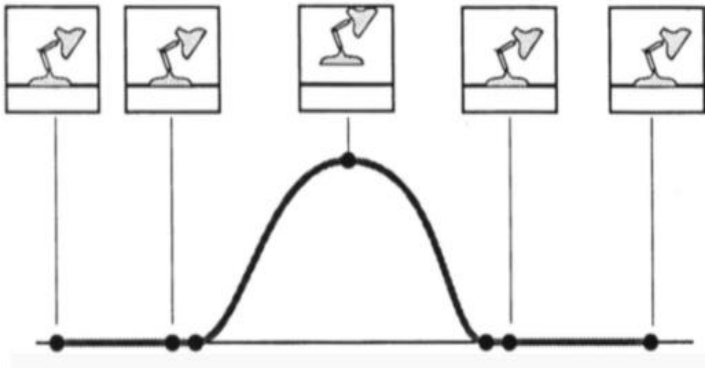
Computer-Assisted Animation

- Keyframing
  - automate the inbetweening
  - good control
  - less tedious
  - creating a good animation still requires considerable skill and talent



http://www.erimez.com/

Keyframing



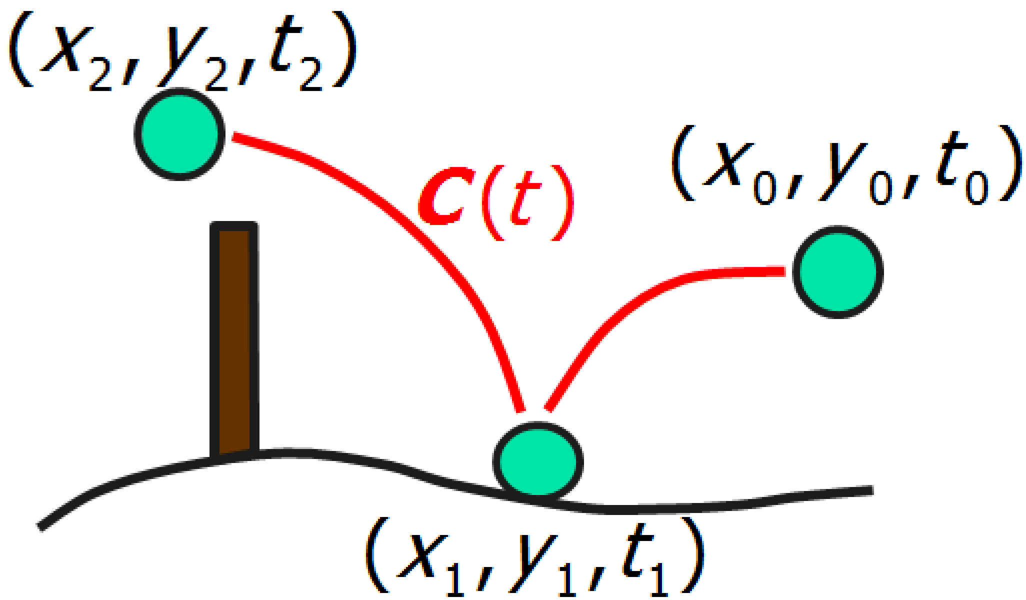
ACM © 1987 "Principles of traditional animation applied to 3D computer animation"

- Describe motion of objects as a function of time from a set of key object positions. In short, compute the inbetween frames.

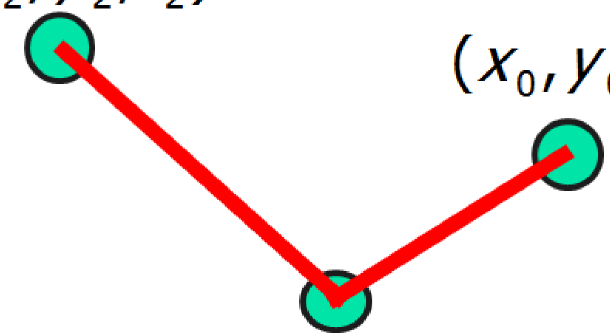
Keyframing

Given positions: $(x_i, y_i, t_i),\ i = 0, \ldots, n$

find curve $\boldsymbol{C}(t) = \begin{bmatrix} x(t) \\ y(t) \end{bmatrix}$ such that $\boldsymbol{C}(t_i) = \begin{bmatrix} x_i \\ y_i \end{bmatrix}$



Keyframing – Linear Interpolation

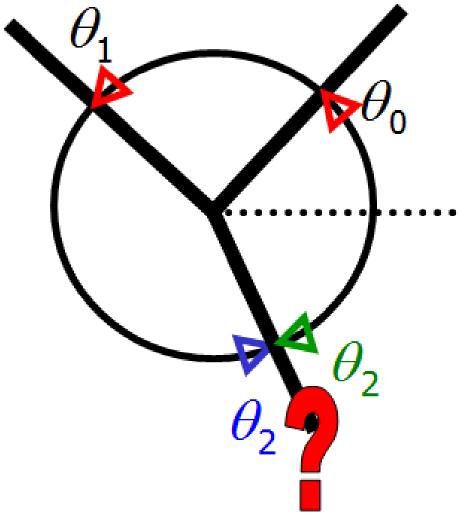


Simple problem: linear interpolation between first two points assuming $t_0=0$ and $t_1=1$: $x(t) = x_0(1-t) + x_1 t$

The x-coordinate for the complete curve in the figure:

$$x(t) = \begin{cases} \dfrac{t_1 - t}{t_1 - t_0} x_0 + \dfrac{t - t_0}{t_1 - t_0} x_1, & t \in [t_0, t_1) \\[2ex] \dfrac{t_2 - t}{t_2 - t_1} x_1 + \dfrac{t - t_1}{t_2 - t_1} x_2, & t \in [t_1, t_2] \end{cases}$$

Keyframing



- Polynomial interpolation
- Spline interpolation
- Interpolation of angles
  - is ambiguous!
  - Different measurements will produce different motion
- All methods have to interpolate usually 6 degrees of freedom + velocity and acceleration
- Common: interpolate each parameter (position, orientation, pitch, yaw, etc.) separately
- However, in 3D?

Interpolating Orientations in 3-D

- Quaternion Interpolation
- **L**inear int**erp**olation (lerp) of quaternion representation of orientations gives us something better:

$$\mathrm{lerp}\left(\mathbf{q}_0, \mathbf{q}_1, t\right) = \mathbf{q}(t) = \mathbf{q}_0\left(1 - t\right) + \mathbf{q}_1 t$$

- A quaternion can represent a rotation by an angle θ around a unit axis **a**:

$$\mathbf{q} = \begin{bmatrix} q_0 & q_1 & q_2 & q_3 \end{bmatrix}$$

$$\mathbf{q} = \left[\cos\frac{\theta}{2} \quad a_x \sin\frac{\theta}{2} \quad a_y \sin\frac{\theta}{2} \quad a_z \sin\frac{\theta}{2}\right]$$

Interpolating Orientations in 3-D

- To convert a quaternion to a rotation matrix:

$$\begin{bmatrix} 1-2q_2^2-2q_3^2 & 2q_1q_2+2q_0q_3 & 2q_1q_3-2q_0q_2 \\ 2q_1q_2-2q_0q_3 & 1-2q_1^2-2q_3^2 & 2q_2q_3+2q_0q_1 \\ 2q_1q_3+2q_0q_2 & 2q_2q_3-2q_0q_1 & 1-2q_1^2-2q_2^2 \end{bmatrix}$$

Interpolating Orientations in 3-D

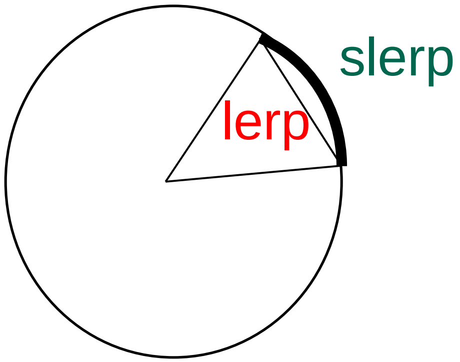
- Linear interpolation of Quaternions:
- If we want to do a linear interpolation between two points **a** and **b** in normal space

$$\operatorname{lerp}(\mathbf{q}_0, \mathbf{q}_1, t) = \mathbf{q}(t) = \mathbf{q}_0(1-t) + \mathbf{q}_1 t$$

- where t ranges from 0 to 1
- Note that the Lerp operation can be thought of as a weighted average (convex)

Interpolating Orientations in 3-D

- If we want to interpolate between two points on a sphere (or hypersphere), we don't just want to Lerp between them
- Instead, we will travel across the surface of the sphere by following a 'great arc'



Interpolating Orientations in 3-D

- We define the spherical linear interpolation (slerp) of two unit vectors in N dimensional space as:

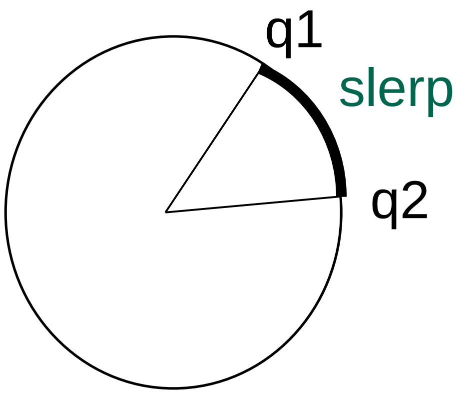
$$Slerp(t, \mathbf{a}, \mathbf{b}) = \frac{\sin\left((1-t)\theta\right)}{\sin\theta}\mathbf{a} + \frac{\sin\left(t\theta\right)}{\sin\theta}\mathbf{b}$$

$$where: \theta = \cos^{-1}\left(\mathbf{a} \cdot \mathbf{b}\right)$$

Interpolating Orientations in 3-D

- Remember that there are two redundant vectors in quaternion space for every unique orientation in 3D space
- What is the difference between

  Slerp(t,**q1**,**q2**)   and   Slerp(t,-**q1**,**q2**) ?



Interpolating Orientations in 3-D

- Remember that there are two redundant vectors in quaternion space for every unique orientation in 3D space
- What is the difference between

  Slerp(t,**q1**,**q2**) and Slerp(t,-**q1**,**q2**) ?

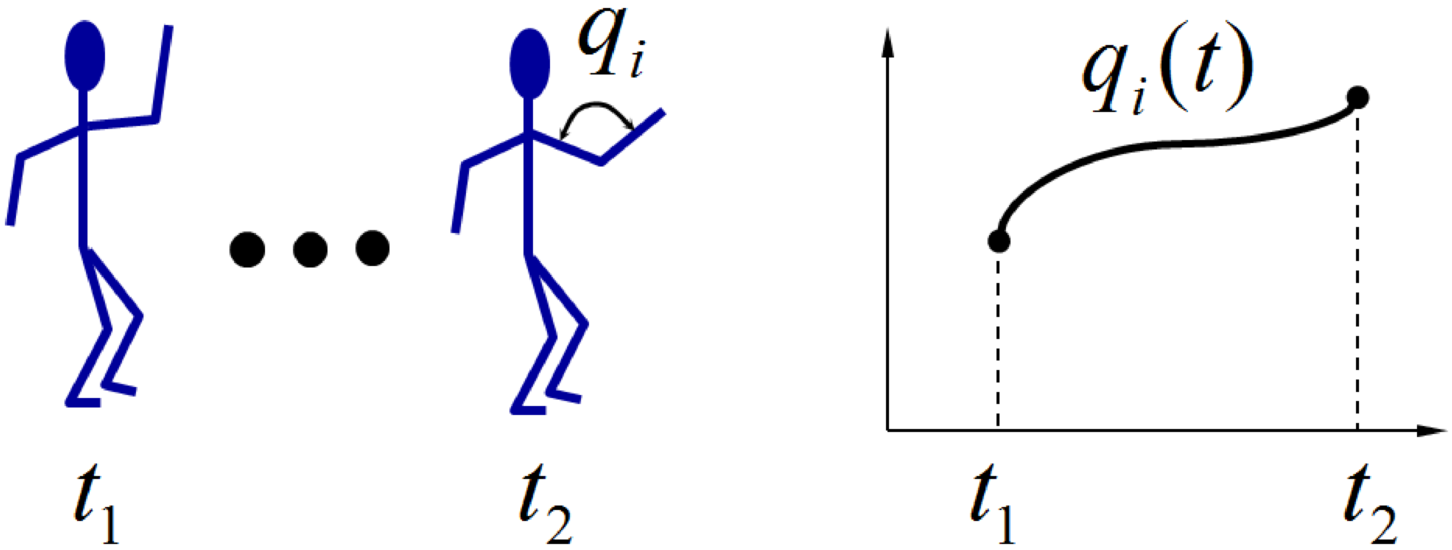
- One of these will travel less than 90 degrees while the other will travel more than 90 degrees across the sphere
- This corresponds to rotating the ‘short way’ or the ‘long way’
- Usually, we want to take the short way, so we negate one of them if their dot product is < 0

Interpolating Orientations in 3-D

- We can construct Bezier curves on the 4D hypersphere by following the exact same procedure using Slerp instead of Lerp
- It's a good idea to flip (negate) the input quaternions as necessary in order to make it go the 'short way'
- There are other, more sophisticated curve interpolation algorithms that can be applied to a hypersphere
  - Interpolate several key poses
  - Additional control over angular velocity, angular acceleration, smoothness…

Articulated Models

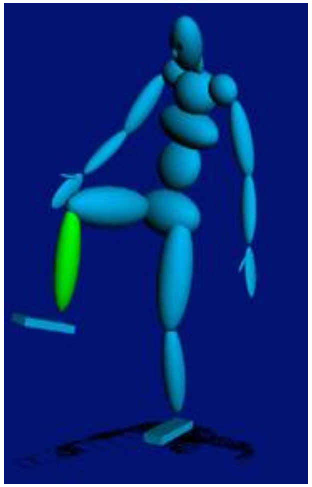
- **Articulated models**:
  - rigid parts
  - connected by joints
- They can be animated by specifying the joint angles (or other display parameters) as functions of time.



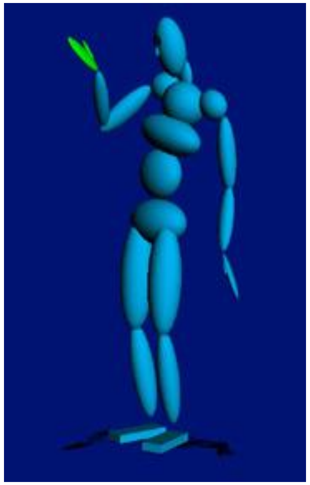
Forward Kinematics

- Describes the positions of the skeleton parts as a function of the joint angles.

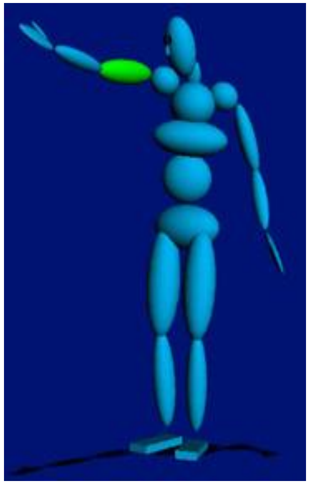
1 DOF: knee



2 DOF: wrist

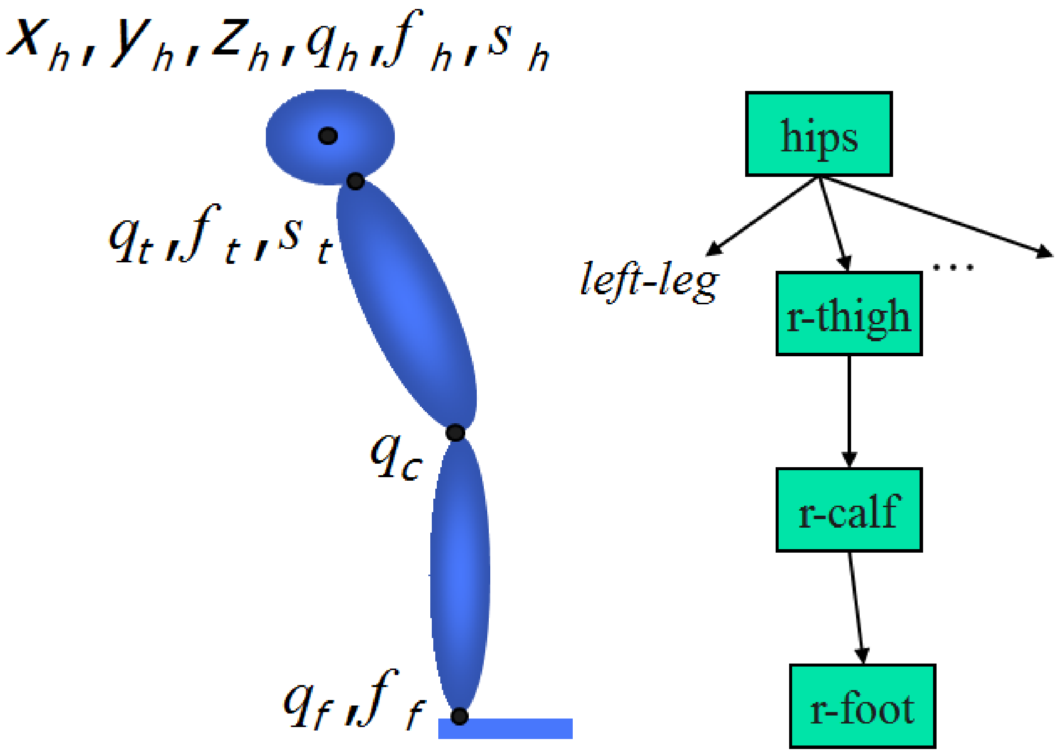


3 DOF: arm



Forward Kinematics

- Each bone transformation described relative to the parent in the hierarchy:



Forward Kinematics

- Transformation matrix for a sensor/effecter $\mathbf{v}_s$ is a matrix composition of all joint transformation between the sensor/effecter and the root of the hierarchy.

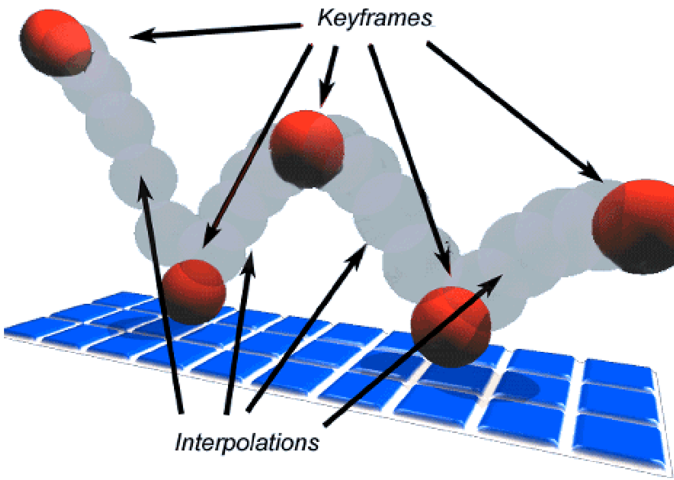
- **Kinematics**
- Describes the positions of the body parts as a function of the joint angles.
- **Dynamics**
- Describes the positions of the body parts as a function of the applied forces.

Inverse Kinematics

- Forward Kinematics
  - Given the skeleton parameters (position of the root and the joint angles) ***p*** and the position of the sensor/effecter in local coordinates v_s, what is the position of the sensor in the world coordinates v_w.
  - Not too hard, we can solve it by evaluating $\mathbf{S}(\mathbf{p})v_s$
- Inverse Kinematics
  - Given the position of the sensor/effecter in local coordinates v_s and the position of the sensor in the world coordinates v_w, what are the skeleton parameters ***p***.
  - Much harder requires solving the inverse of

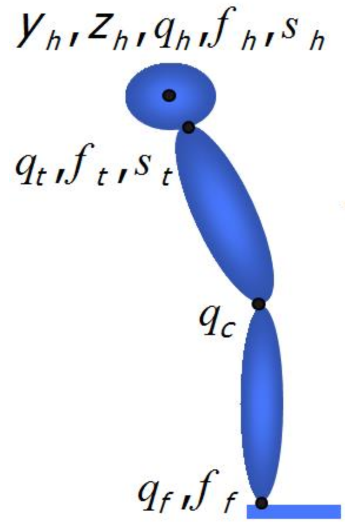
  the non-linear function $\mathbf{S}(\mathbf{p})$
  - We can solve it by root-finding $\mathbf{p}$? such that $\mathbf{S}(\mathbf{p})v_s - \mathrm{v_w} = 0$
  - We can solve it by optimization $\underset{\mathbf{p}}{\text{minimize}} \left(\mathbf{S}(\mathbf{p})v_s - \mathrm{v_w}\right)^2$

Animation + Kinematics + Model?



http://www.erimez.com/

+

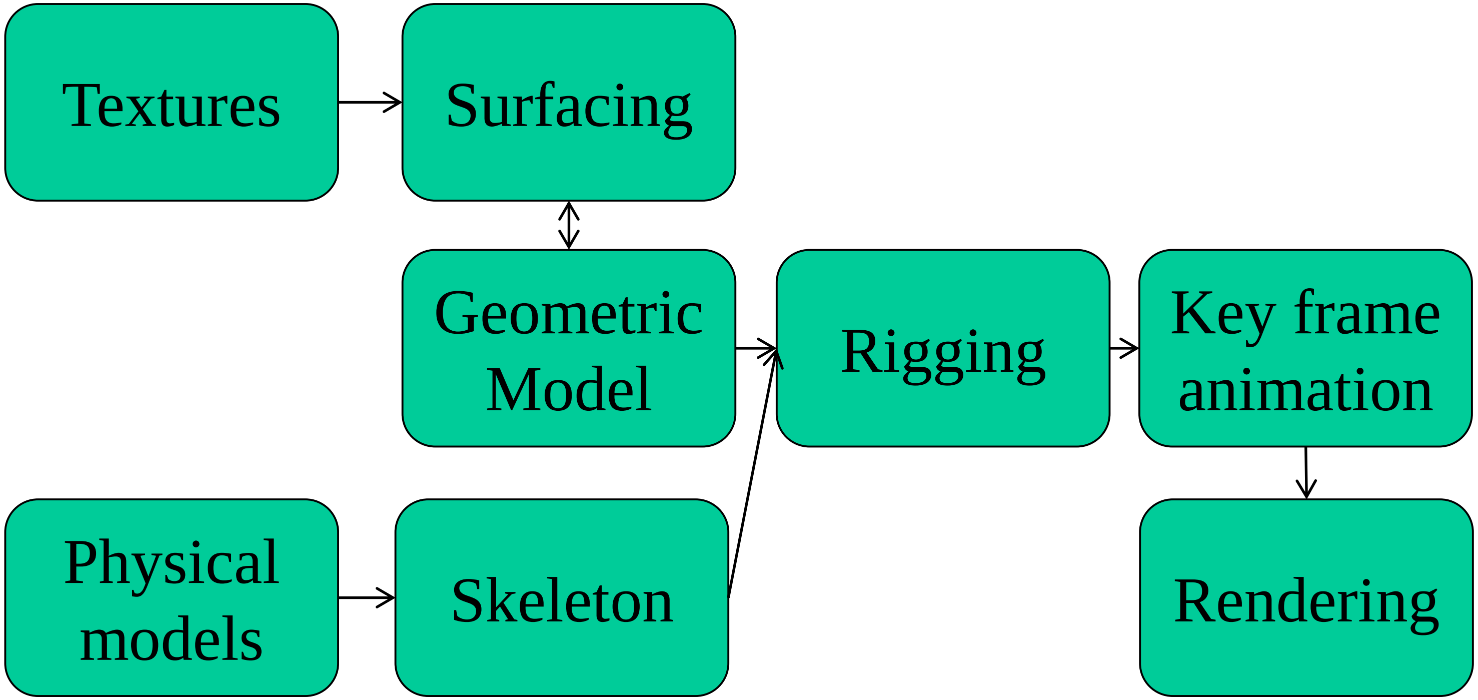


http://docs.unity3d.com/

+

Production process

- A lot of manual work!



Skin

- Robots and mechanical creatures can usually be rendered with rigid parts and don’t require a smooth skin
- To render rigid parts, each part is transformed by its joint matrix independently
- In this situation, every vertex of the character’s geometry is transformed by exactly one matrix

$$\mathbf{v}' = \mathbf{v} \cdot \mathbf{W}$$

where v is defined in joint’s local space

Simple Skin

- A simple improvement for low-medium quality characters is to rigidly bind a skin to the skeleton. This means that every vertex of the continuous skin mesh is attached to a joint.
- In this method, as with rigid parts, every vertex is transformed exactly once and should therefore have similar performance to rendering with rigid parts.

$$\mathbf{v}' = \mathbf{v} \cdot \mathbf{W}$$

Smooth Skin

- With the smooth skin algorithm, a vertex can be attached to more than one joint with adjustable weights that control how much each joint affects it
- Vertices rarely need to be attached to more than three joints
- Each vertex is transformed a few times and the results are blended
- The smooth skin algorithm has many other names: blended skin, skeletal subspace deformation (SSD), multi-matrix skin, matrix palette skinning…

Smooth Skin

- The deformed vertex position is a weighted sum:

$$\mathbf{v}' = w_1(\mathbf{v}\cdot\mathbf{M}_1) + w_2(\mathbf{v}\cdot\mathbf{M}_2) + \ldots w_N(\mathbf{v}\cdot\mathbf{M}_N)$$

$$or$$

$$\mathbf{v}' = \sum w_i(\mathbf{v}\cdot\mathbf{M}_i)$$

$$where$$

$$\sum w_i = 1$$

Smooth Skin

- Binding Matrices:
- With rigid parts or simple skin, v can be defined local to the joint that transforms it
- With smooth skin, several joints transform a vertex, but it can’t be defined local to all of them
- Instead, we must first transform it to be local to the joint that will then transform it to the world
- To do this, we use a binding matrix **B** for each joint that defines where the joint was when the skin was attached and pre multiply its inverse with the world matrix:

$$\mathbf{M}_i = \mathbf{B}_i^{-1} \cdot \mathbf{W}_i$$

Smooth Skin

- Normals:
- To compute shading, we need to transform the normals to world space also
- Because the normal is a direction vector, we don't want it to get the translation from the matrix, so we only need to multiply the normal by the upper 3x3 portion of the matrix
- For a normal bound to only one joint:

$$\mathbf{n}' = \mathbf{n} \cdot \mathbf{W}$$

Smooth Skin

Skin::Update() (view independent processing)

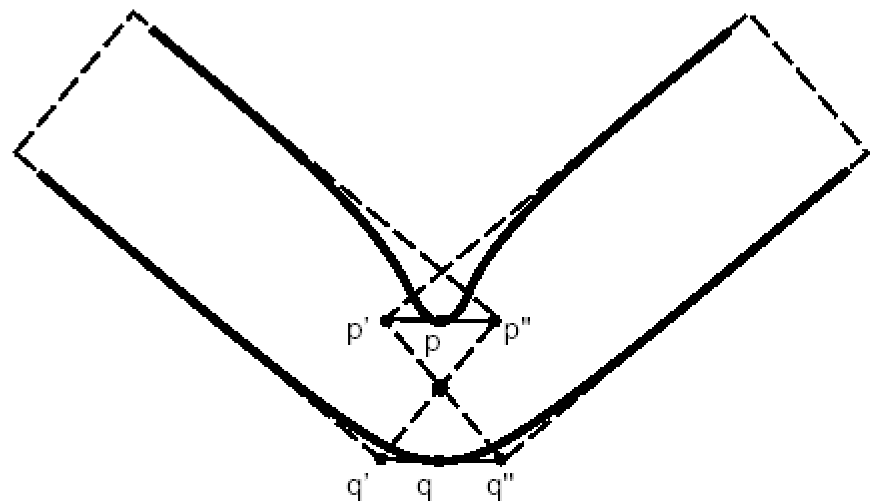
- Compute skinning matrix for each joint: $M=B^{-1}\cdot W$ (you can precompute and store B^{-1} instead of B)
- Loop through vertices and compute blended position & normal

Skin::Draw() (view dependent processing)

- Set matrix state to Identity (world)
- Loop through triangles and draw using world space positions & normals

Smooth Skin

- Smooth skin is very simple and quite fast, but its quality is limited
- The main problems are:
  - Joints tend to collapse as they bend more
  - Very difficult to get specific control
  - Unintuitive and difficult to edit
- Still, it is built in to most 3D animation packages and has support in both OpenGL and Direct3D
- If nothing else, it is a good baseline upon which more complex schemes can be built


p'
p
p"
q'
q
q"

Smooth Skin

- Improvements
- Bone links
  - extra joints inserted in the skeleton to assist with the skinning
- Shape Interpolation
  - allow the verts to be modeled at key values along the joints motion
  - For an elbow, for example, one could model it straight, then model it fully bent

Rigging Process

- To rig a skinned character, one must have a geometric skin mesh and a skeleton
- Usually, the skin is built in a relatively neutral pose, often in a comfortable standing pose
- The skeleton, however, might be built in more of a ‘zero’ pose where are joints DOFs are assumed to be 0, causing a very stiff, straight pose
- To attach the skin to the skeleton, the skeleton must first be posed into a binding pose
- Once this is done, the verts can be assigned to joints with appropriate weights

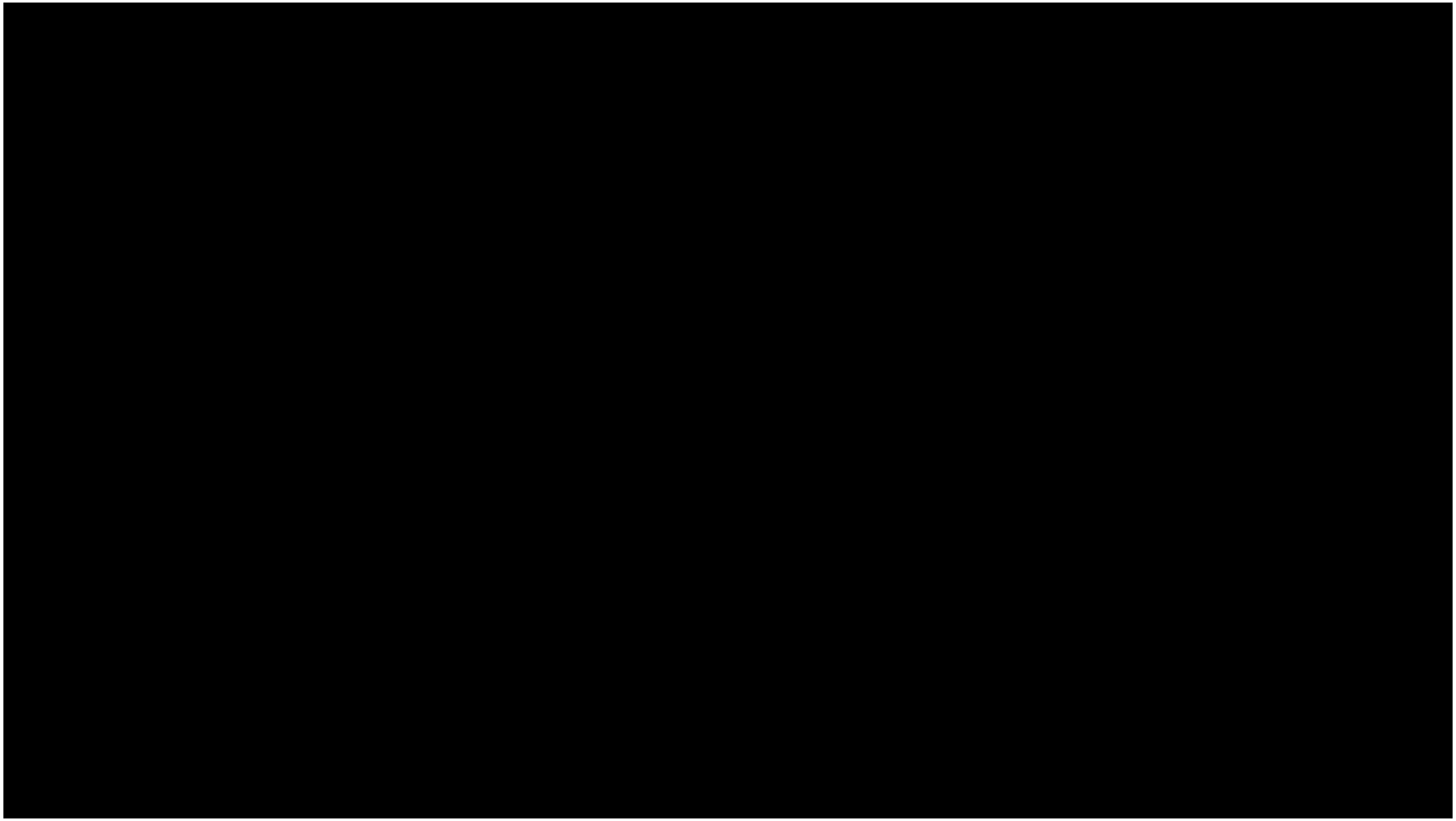
Skin Binding

- Attaching a skin to a skeleton is not a trivial problem and usually requires automated tools combined with extensive interactive tuning
- Binding algorithms typically involve heuristic approaches
- Some general approaches:
  - Containment
  - Point-to-line mapping
  - Delaunay tetrahedralization

Animation in practise

Mike Pickton via youtube

Production process in practice



https://www.youtube.com/watch?v=OM58kkZpTNw

Questions?