Basic methods in Computer Animation

Lesson 02

Lesson 02 Outline

- Key-framing and parameter interpolation
- Skeleton Animation
- Forward and inverse Kinematics
- Procedural techniques
- Motion capture


Key-framing
and parameter interpolation

Key-frame base Animation

- Comes from traditional frame-based animations
- Trivial principle
  - Define object states (positions...) only in KEY frames
  - Let the computer calculate the in-between frames by interpolating state variables (positions...)
- Linterpolation types
  - Simple linear interpolation (insufficient in most scenarios)
  - Spline (cubic bezier) interpolation (commonly used)
  - Spherical (linear/bezier) interpolation (for quaternions)

Parameter interpolation

- Structure of key-frame: $F_i = (t_i , p_i)$
  - t_i: Time of i-th frame
  - p_i: Parameter value of i-th frame (position, color...)
- Problem
  - Having values in key-frames how to get reasonable values for in-between frames ?
- Solution
  - Given time t ($t_i < t < t_{i+1}$) and frames (F_i, F_{i+1})
  - Find parameter value $p = I(t, F_i, F_{i+1})$
  - Where I is some frame interpolation funtion
    - Nearest neighbor interpolation
    - Linear interpolation
    - Spline and many more

Parameter interpolation

- Frame Interpolation algorithm
  - Store key-frames F_i sorted by the time value ascending
  - Given time t, use binary search to find interval (F_i, F_{i+1})
  - Calculate parameter $p = I(t, F_i, F_{i+1})$ with interpolation
- Optimization
  - When time changes coherent: t' = t + dt where dt is small
  - Use interpolation evaluator instead of binary search
- Interpolation evaluator (similar to iterator)
  - Simple additional structure to store current key-frame and estimate next key-frame
  - Store intermediate results of previous interpolation
  - etc

Nearest neighbor interpolation

- Method:

  $p = I(t, F_i, F_{i+1}) = F_i$
- Pros
  - Very fast evaluation
  - Simple implementation
- Cons
  - Sharp discontinuities (non-smooth)
  - Non-physical motion
  - Visually distracting

Linear Interpolation

- Method (LERP)

$$\mathrm{p} = (1-s)\,\mathrm{p}_i + s\,\mathrm{p}_{i+1} \qquad where \qquad s = \frac{t - t_i}{t_{i+1} - t_i}$$

- Pros
  - Continuous motion
  - Fast and easy calculation and implementation
- Cons
  - Motion is only linear
  - Non-physical
  - First time derivation of motion is discontinuous

Cubic Bezier Interpolation

- N-th Bezier interpolation curve
  - Parameter s (0 <= s <= 1) is not time !!!

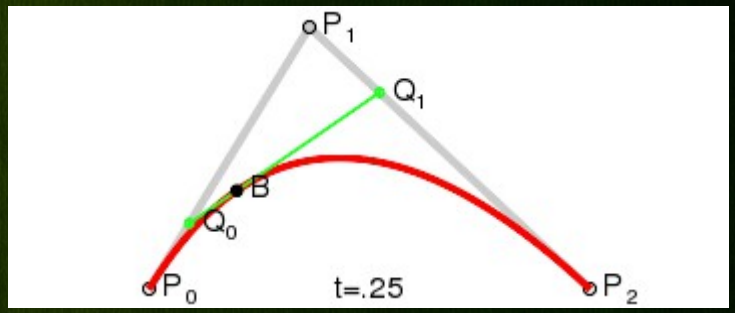
$$B^n(s) = \sum_{i=0}^{n} \binom{n}{i} (1-s)^{n-i} s^i \, \mathrm{p}_i$$

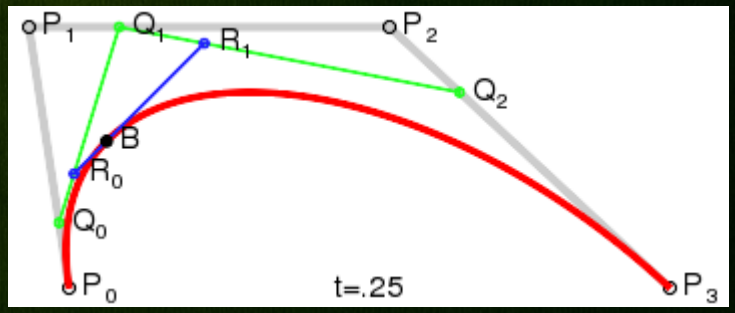
- Cubic (3-th) Bezier interpolation curve
  - Parameter s (0 <= s <= 1) is not time !!!

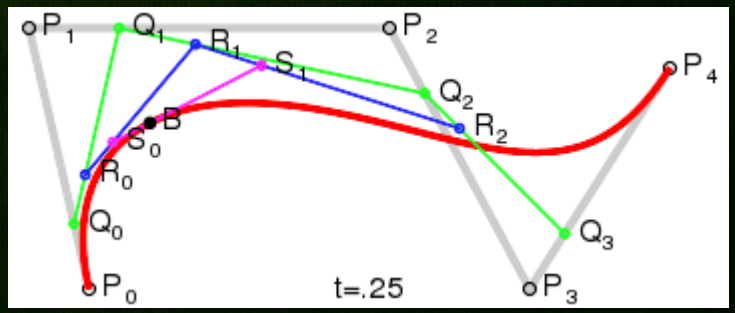
$$B^3(s) = (1-s)^3 \, \mathrm{p}_0 + (1-s)^2 s \, \mathrm{p}_1 + (1-s) s^2 \, \mathrm{p}_2 + s^3 \, \mathrm{p}_3$$

Cubic Bezier Interpolation

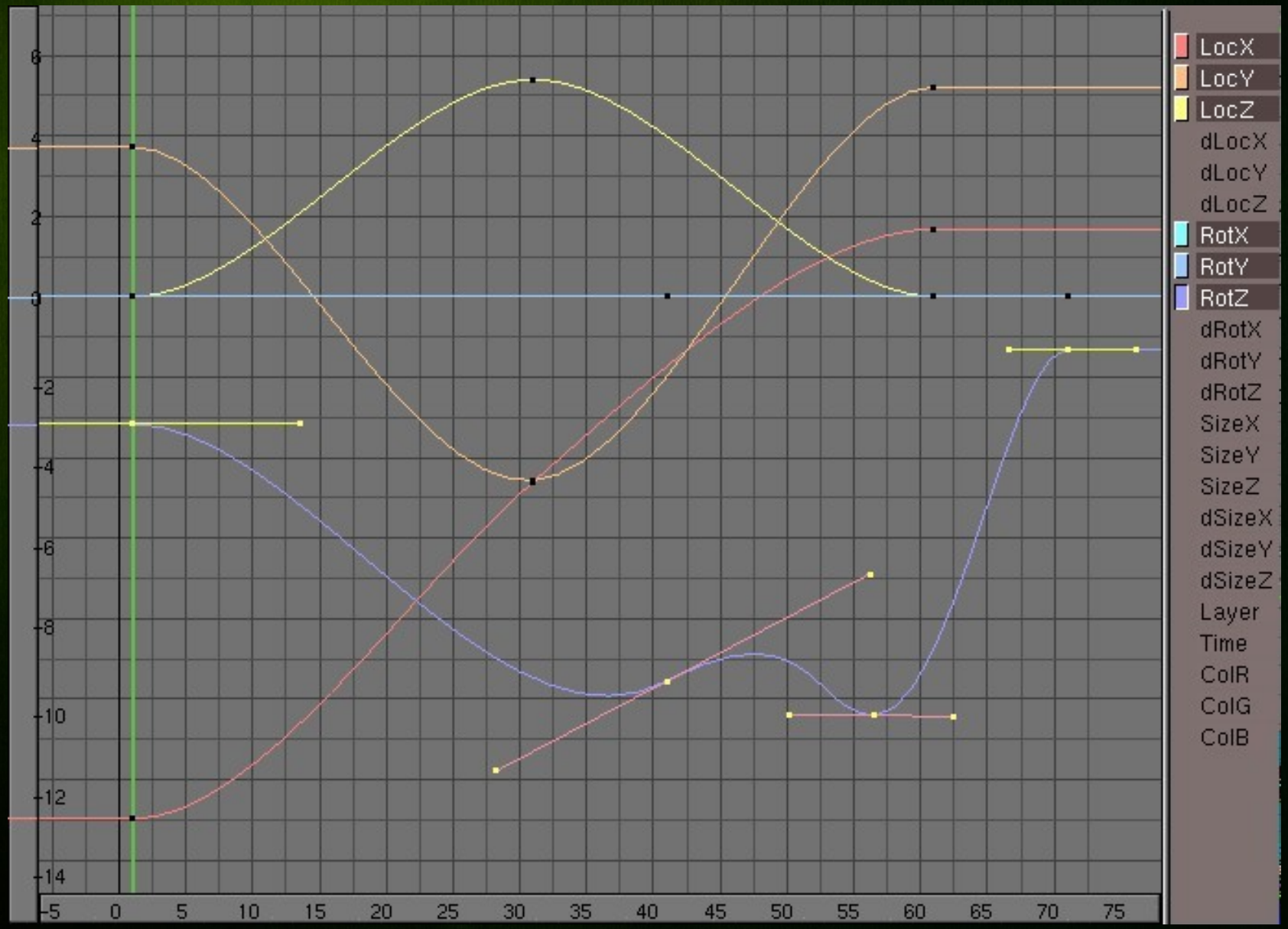
- Quadratic Bezier
  - Quadratic (s^2) equation
  - 3 control points
- Cubic Bezier
  - Cubic (s^3) equation
  - 4 control points
- Quartic Bezier
  - Quartic (s^4) equation
  - 5 control points



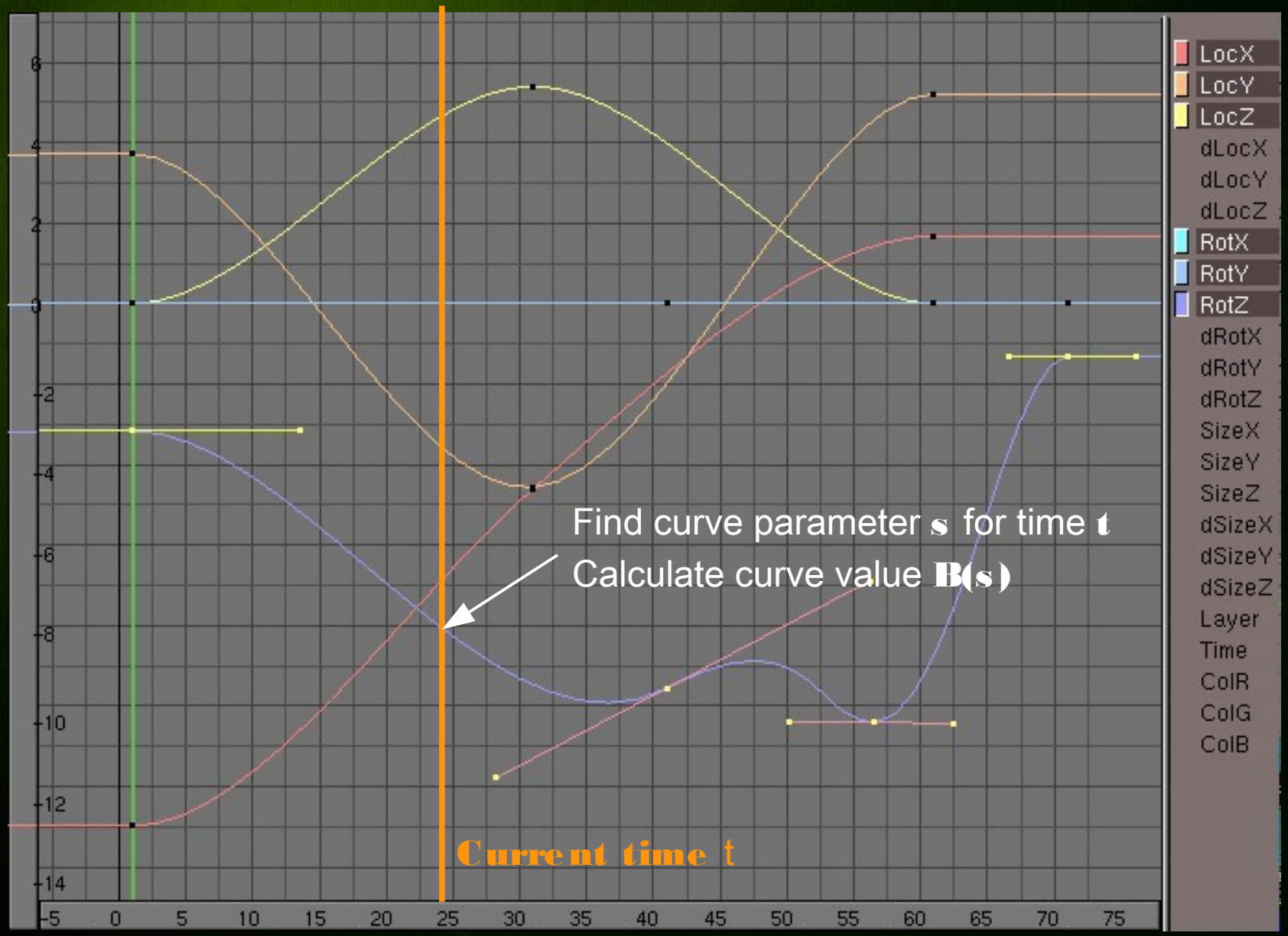




Bezier Interpolation in key-framing



Bezier Interpolation in key-framing



Curve interpolation in key-framing

- Curve parameter s is not time t (s != t)
- 1) For given time t find curve parameter s
  - No trivial analytic solution for cubic curves
  - Solve it numerically using binary search (can be slow)
- Optimization for (1)
  - Fix number of iterations, than use linear interpolation for s
  - Precompute values into cache, that use neighbor interpol.
- 2) With parameter s calculate curve value $\mathbf{B}(s)$
  - Evaluate parametric Bezier curve

Orientation in 3D

- Orientation in 3D has no natural representation
- There are more common definitions

- Orientation Matrix (Euler Angles)
- Orientation Axis and Angle
- Orientation Quaternion

- Each type has its pros/cons

Orientation Matrix (Euler Angles)

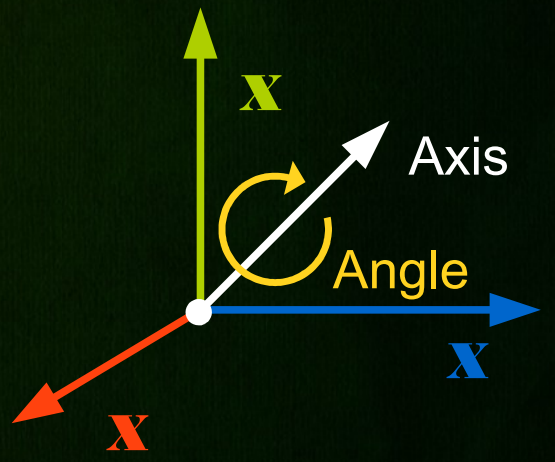
- Orientation is defined as 3 rotation angles (Ax, Ay, Az) around X-axis, Y-axis and Z-axis
- Orientation is represented as composition of 3 orthonormal rotation matrices (Rx, Ry, Rz) around (X, Y, Z) axes → R = Rx*Ry*Rz

$$\mathbf{R}_x(a_x)=\begin{pmatrix}1 & 0 & 0\\ 0 & +\cos(a_x) & -\sin(a_x)\\ 0 & +\sin(a_x) & +\cos(a_x)\end{pmatrix} \quad \mathbf{R}_y(a_y)=\begin{pmatrix}+\cos(a_y) & 0 & +\sin(a_y)\\ 0 & 1 & 0\\ -\sin(a_y) & 0 & +\cos(a_y)\end{pmatrix} \quad \mathbf{R}_z(a_z)=\begin{pmatrix}+\cos(a_z) & -\sin(a_z) & 0\\ +\sin(a_z) & +\cos(a_z) & 0\\ 0 & 0 & 1\end{pmatrix}$$

- Not unique representation and “Gimbal Lock”
- Complicated decomposition (matrix → angles)

Rotation Axis and Angle

- Every rotation in 3D can be defined by its
  - Axis $\mathbf{u}=(x,y,z)$ (a direction that is left fixed by the rotation)
  - Angle $\mathbf{a}$ (the amount by which the rotation turns)



- Axis-Angle → Rotation Matrix

$$\mathbf{R}=\mathbf{P}+(\mathbf{I}-\mathbf{P})\cos(a)-\mathbf{Q}\sin(a)$$

$$\mathbf{P}=\begin{pmatrix} u_x u_x & u_x u_y & u_x u_z \\ u_y u_x & u_y u_y & u_y u_z \\ u_z u_x & u_z u_y & u_z u_z \end{pmatrix}=\mathbf{u}\,\mathbf{u}^{\mathrm{T}} \qquad \mathbf{I}=\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \qquad \mathbf{Q}=\begin{pmatrix} 0 & -u_z & +u_y \\ +u_z & 0 & -u_x \\ -u_y & +u_x & 0 \end{pmatrix}$$

Quaternions

- Similar to complex numbers
- Defined as: $q = w + x\mathbf{i} + y\mathbf{j} + z\mathbf{k}$ | $\mathbf{i}^2 = \mathbf{j}^2 = \mathbf{k}^2 = \mathbf{ijk} = -1$
- Add: $p+q = (w+w') + (x+x')\mathbf{i} + (y+y')\mathbf{j} + (z+z')\mathbf{k}$
- Multiply: $pq = (w + x\mathbf{i} + y\mathbf{j} + z\mathbf{k})(w' + x'\mathbf{i} + y'\mathbf{j} + z'\mathbf{k}) =$ $(ww'-xx'-yy'-zz') + (wx'+xw'+yz'+zy')\mathbf{i} + (wy'+xz'+yw'+zf')\mathbf{j} + (wz'+xy'+yx' + zw')\mathbf{k}$

- More info on wikipedia

Quaternions for spatial rotations

- Given unit rotation axis $\mathbf{u}$=(x,y,z) $|\mathbf{u}|$=1 and angle $\mathbf{a}$
- Define quaternion $\mathbf{q}$ as: $\mathbf{q} = \cos(a/2) + \mathbf{u} \sin(a/2)$
- Any vector $\mathbf{v}$ can be rotated around $\mathbf{u}$ by angle $\mathbf{a}$ as $\mathbf{v}' = \mathbf{q}\mathbf{v}\mathbf{q}^{-1}$ (quaternion rotation formula)
  - Here v = (a,b,c) represents quaternion $0 + a\mathbf{i} + b\mathbf{j} + c\mathbf{k}$
  - See proof in wikipedia – by converting into Rodrigues formula
- Rotation composition of $\mathbf{p}$ and $\mathbf{q}$ is r = $\mathbf{pq}$
  - $rvr^{-1} = (pq)v(pq)^{-1} = pqvq^{-1}p^{-1} = q(pvp^{-1})q^{-1}$
- Inverse rotation of $\mathbf{q}$ in $\mathbf{q}^{-1}$
  - $v = (q^{-1}q)v(q^{-1}q)^{-1} = q^{-1}qvqq^{-1} = q^{-1}(qvq^{-1})q$

Quaternions: definition

Definition: A quaternion is noted $q = s + v_x i + v_y j + v_z k$
with s, v_x, v_y, v_z: real numbers and i, j, k: imaginary numbers such that $i^2=j^2=k^2=-1$, $ij=-ji=k$, $jk=-kj=i$, $ki=-ik=j$.

- In condensed notation the quaternion can be expressed as $q = (s,\mathbf{v})$ where s is scalar part of q and $\mathbf{v}$ is the vector part with axes i, j, k.
- A unit quaternion ($|q_u| = 1$) can be noted as $q_u = (\cos(\theta), \sin(\theta)\,\mathbf{v})$
- The conjugate of quaternion $q = (s,\mathbf{v})$ is equal to $\bar{q} = (s,-\mathbf{v})$
- For a unit quaternion q_u we have

$$\bar{q}_u = q_u^{-1}$$

- In the context of orientation interpolation, quaternion angle 2θ of q_u can be interpreted as the rotation angle while the vector $\mathbf{v}$ is the rotation axis.

Spherical linear interpolation

- Given two unit vectors $\mathbf{v}_0$ and $\mathbf{v}_1$ and interpolation parameter t in (0,1) the slerp in defined as

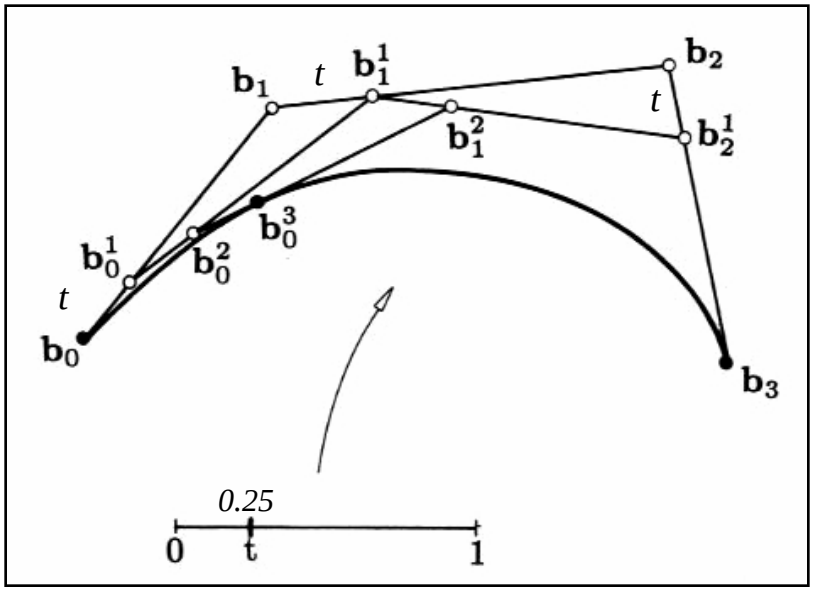
$$\text{slerp}(t, \mathbf{v_0}, \mathbf{v_1}) = \frac{\sin((1-t)a)}{\sin(a)}\mathbf{v_0} + \frac{\sin(ta)}{\sin(a)}\mathbf{v_1}$$

  - Where angle $\mathbf{a}$ = $\cos^{-1}(\mathbf{v}_0\mathbf{v}_1)$ is the angle between $\mathbf{v}_0$ and $\mathbf{v}_1$
- Applied to unit quaternions, slerp produces shortest rotation with constant angular velocity between orientations $\mathbf{q}_0$ and $\mathbf{q}_1$

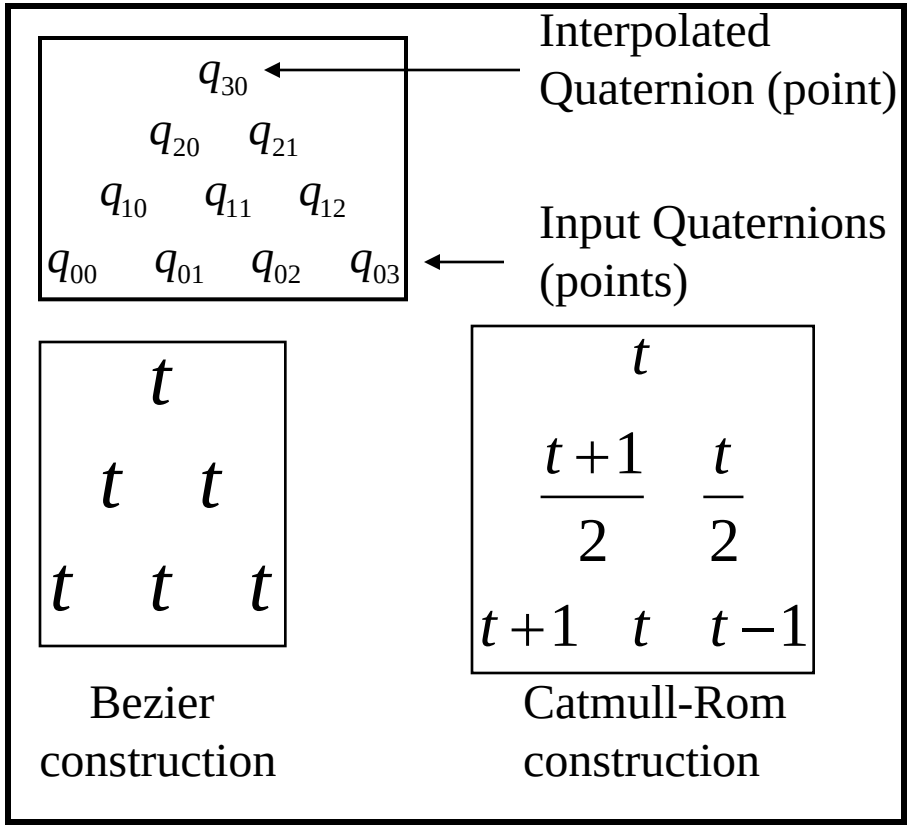
Quaternions: higher order interpolants

Spherical linear interpolation between more than two key orientations produces jerky, sharply changing motion across the keys. Higher order of continuity is required, e.g., spherical equivalent of the cubic spline.

A simple example of such construction is Catmull-Rom spline which passes through the key points and has C^1 continuity.



The de Casteljau algorithm: the point b^3_0 is obtained from repeated linear interpolation for $t=0.25$.



Catmull-Rom spherical interpolation

$$\text{slerp}(q_1, q_2, u) = \frac{\sin(1-u)\theta}{\sin\theta} q_1 + \frac{\sin u\theta}{\sin\theta} q_2, \text{ where } \cos\theta = q_1 \cdot q_2.$$

For example:

```
function qCatmullRom(
        q00, q01, q02, q03: quaternion; t: real
): quaternion;
        q10, q11, q12, q20, q21: quaternion;
begin
        q10 ← slerp(q00, q01, t + 1);
        q11 ← slerp(q01, q02, t);
        q12 ← slerp(q02, q03, t − 1);
        q20 ← slerp(q10, q11, (t + 1)/2);
        q21 ← slerp(q11, q12, t/2);
        return [slerp(q20, q21, t)];
endproc qCatmullRom
```

Quaternions ⟷ rotation matrices

In animation system each key is usually represented as a single orientation matrix. This sequence of matrices will be then converted into a sequence of quaternions. Interpolation between key quaternions is performed and this produces a sequence of in-between quaternions which are then converted back to rotation matrices. The matrices are then applied to the object.

A unit quaternion ***q = (W, (X,Y,Z))*** is equivalent to the matrix:

$$\begin{bmatrix} 1-2Y^2-2Z^2 & 2XY-2WZ & 2XZ+2WY & 0 \\ 2XY+2WZ & 1-2X^2-2Z^2 & 2YZ-2WX & 0 \\ 2XZ-2WY & 2YZ+2WX & 1-2X^2-2Y^2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

To convert from an orthogonal rotation matrix to a unit quaternion, we observe that if ***M = [m_{ij}]*** is the affine transformation in homogeneous form:

trace (M) = 4 - 4(X^2 + Y^2 + Z^2) = 4 W^2

and then ***X, Y, Z*** can be calculated as: ⟶

$$X = \frac{m_{32} - m_{23}}{4W},$$
$$Y = \frac{m_{13} - m_{31}}{4W},$$
$$Z = \frac{m_{21} - m_{12}}{4W}.$$

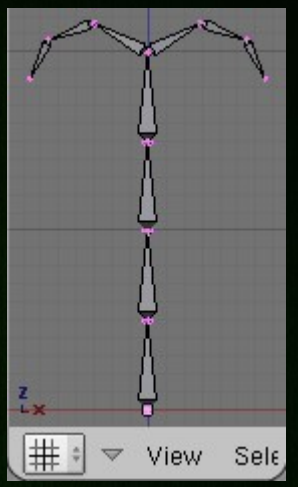

Skeleton
Skinning

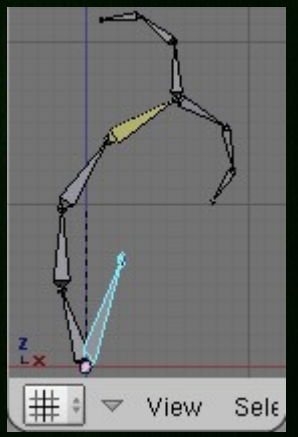
Skeleton Animation

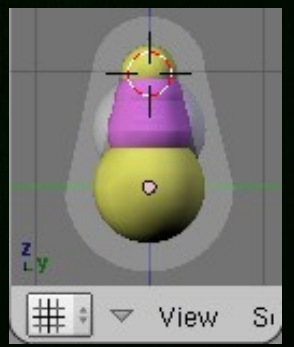
- Inspired by skeleton system of animals
- Basic work-flow
  - Create skeleton – connect bones into hierarchy - rigging
  - Create skin – usually a polygonal mesh of animal
  - Create vertex-bone weights - skinning
  - Animate skeleton using any animation technique - posing
- Skeleton is usually a articulated structure of bones
- Skinning weights define how much each vertex “belongs” to a given bone

Rigging skeleton

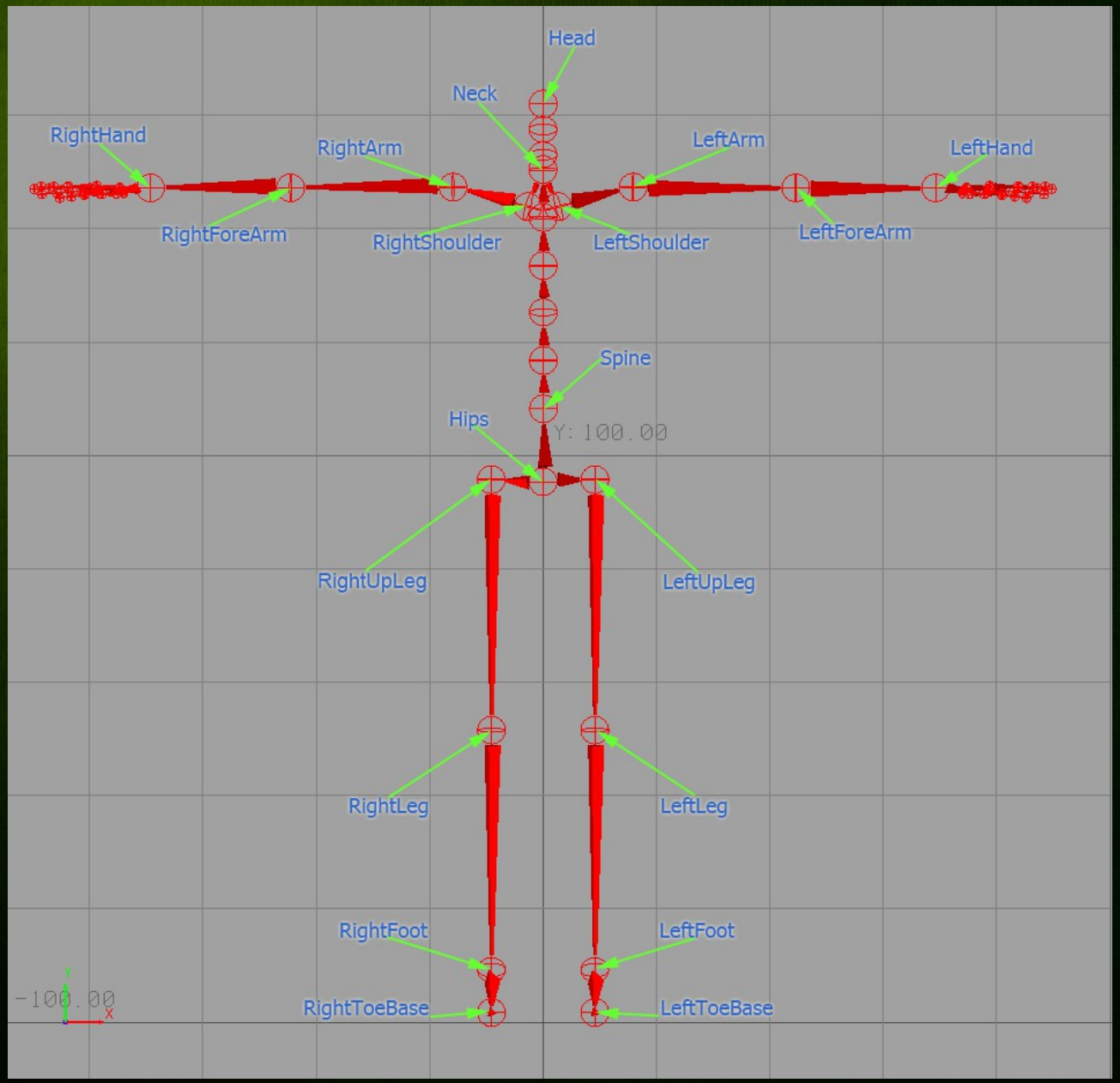
- Rigging
  - Create bone hierarchy – skeleton – in initial pose
- Bone definition
  - Name of bone
  - Reference to parent bone (none for root bone)
  - Set of child bone references (empty for leaves)
  - Local transformation (position, orientation, scale)
  - Length of bone (direction in local Z-axis)
  - Various translation/rotation limits (e.g. knee joint)
  - IK type – start / mid / end effector
  - Weighting type – (cylinder, capsule, sphere...)







Complete skeleton example



Skinning skeleton

- Matrix palette skinning technique
- Each vertex of mesh (skin) has a small set of skinning weights $\mathbf{W} = (\mathbf{w}_0, \mathbf{w}_1, \mathbf{w}_2, \mathbf{w}_3)$
- Each weight $\mathbf{w}_i$ belongs to one (close) bone $\mathbf{B}_i$ with a world transformation matrix $\mathbf{M}_i$
- The final vertex transformation is

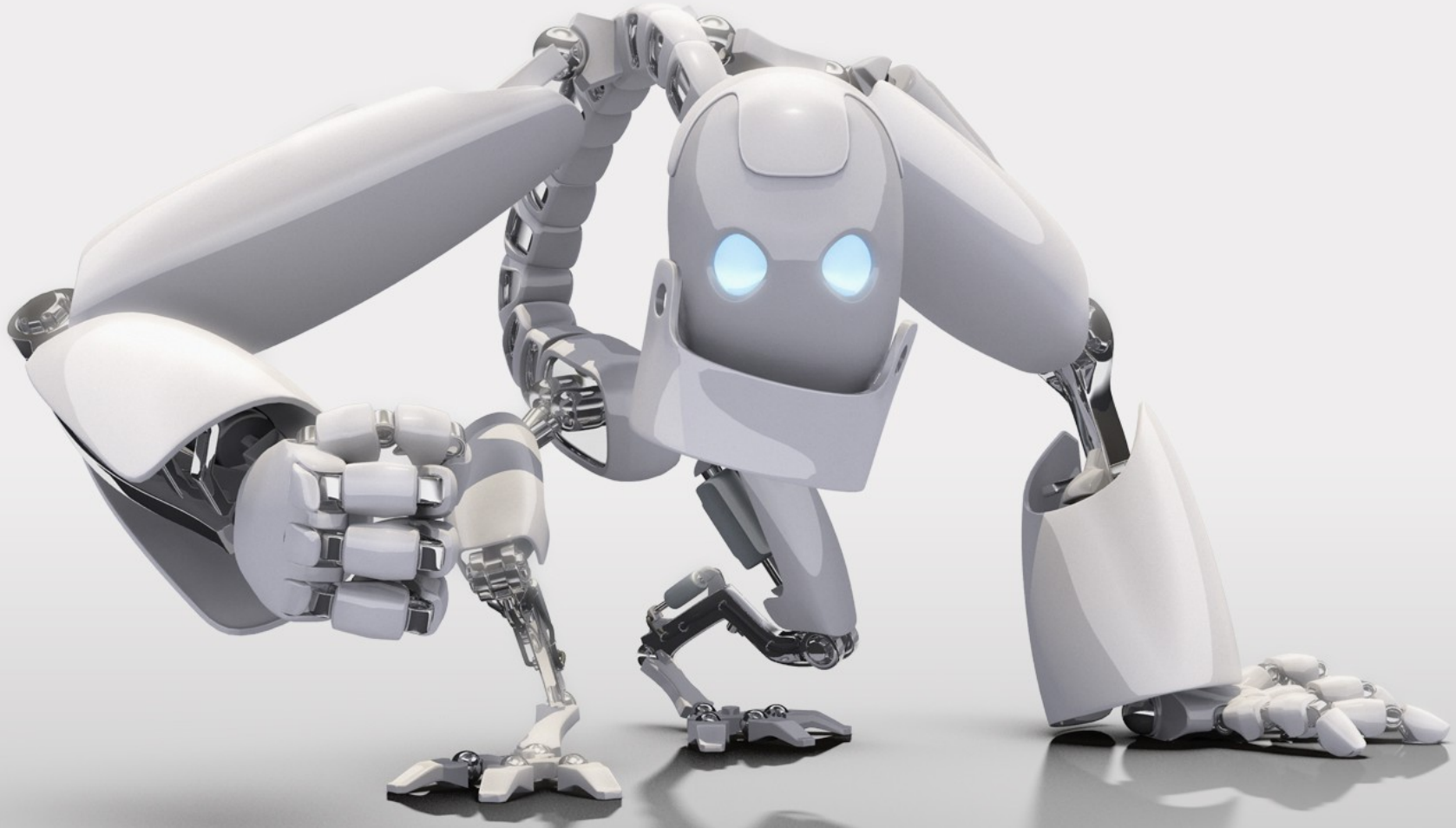
$$\mathbf{v}'_i = \frac{1}{\sum_{k=0}^{n} w_k} \sum_{k=0}^{n} w_k \mathbf{M}_k \mathbf{v}_i$$

Skin, Skeleton and Weights

Posing skeleton

- Only bone transformations are animated
- Any animation technique can be used
- World transformation Q of each bone is composed recursively from parent transform
- $Q_i = M_i Q_{i-1} = M_{i-1} M_{i-2} \ldots M_0$
  - where Q_i is parent of Q_{i+1}
- For leaf bones $Q_i = M_i$

Posing skinned skeleton



Forward and inverse Kinematics

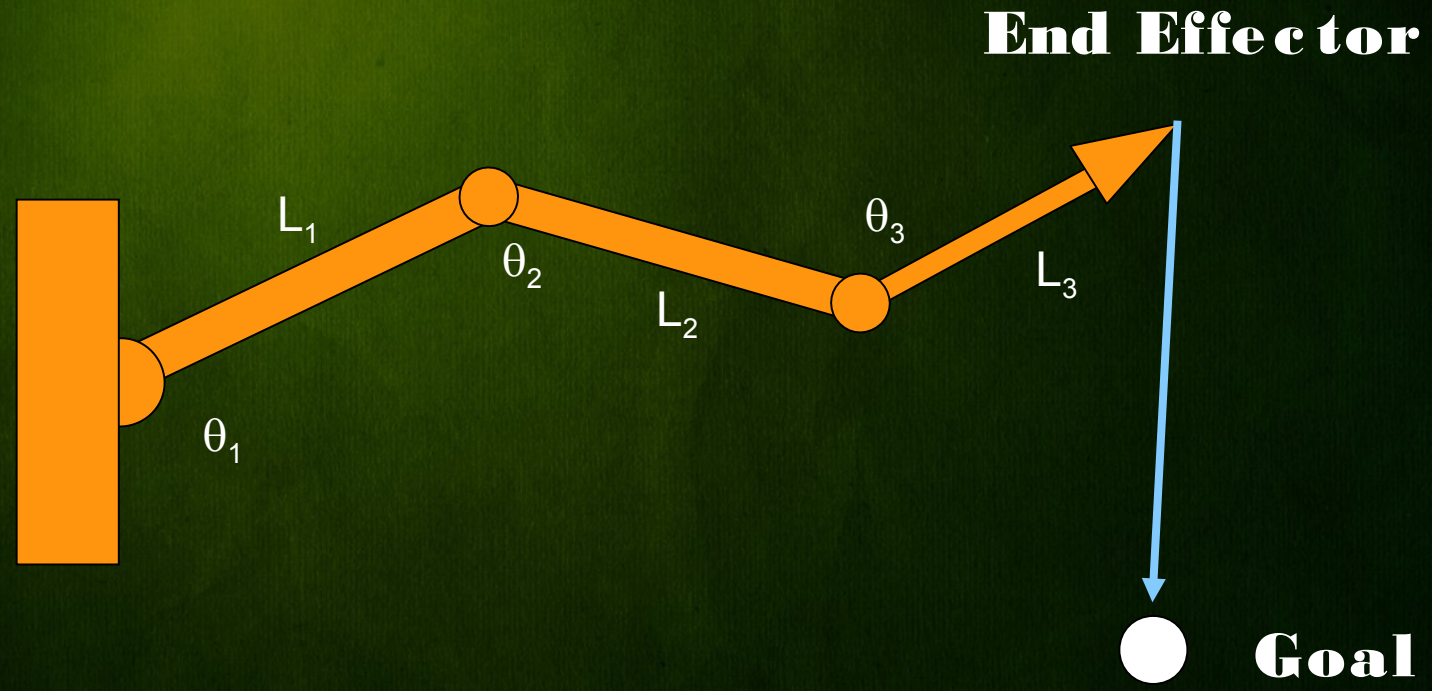
Forward Kinematics

- Forward (direct) kinematics
  - Put objects into transformation hierarchy
  - Animate each transformation directly (eg by key-framing)
  - Problem: Figure wants to reach a cup on a table by hand, but how to interpolate transformations to get natural motion ?

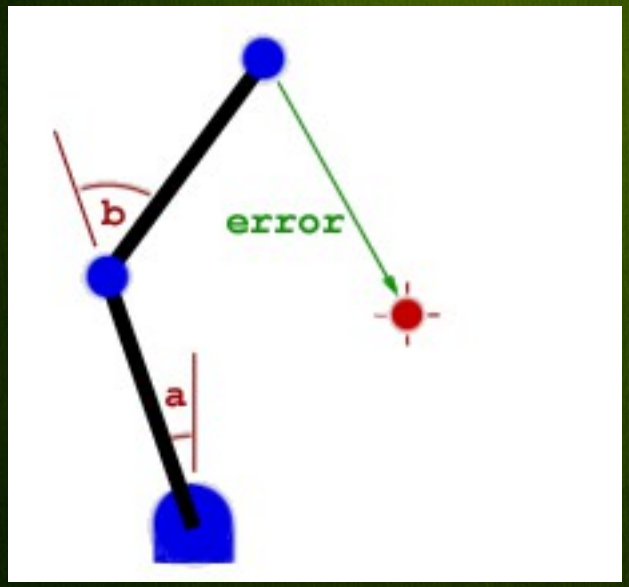
Inverse Kinematics

- Overall strategy
  - Set goal configuration of end effector
  - Calculate interior bone angles
- **Analytical solutions:** when linkage is simple enough, directly calculate joint angles in configuration that satisfies goal
- **Numerical solutions:** complex linkages. At each time slice, determine joint movements that take you in direction of goal position

Inverse Kinematics Scenario

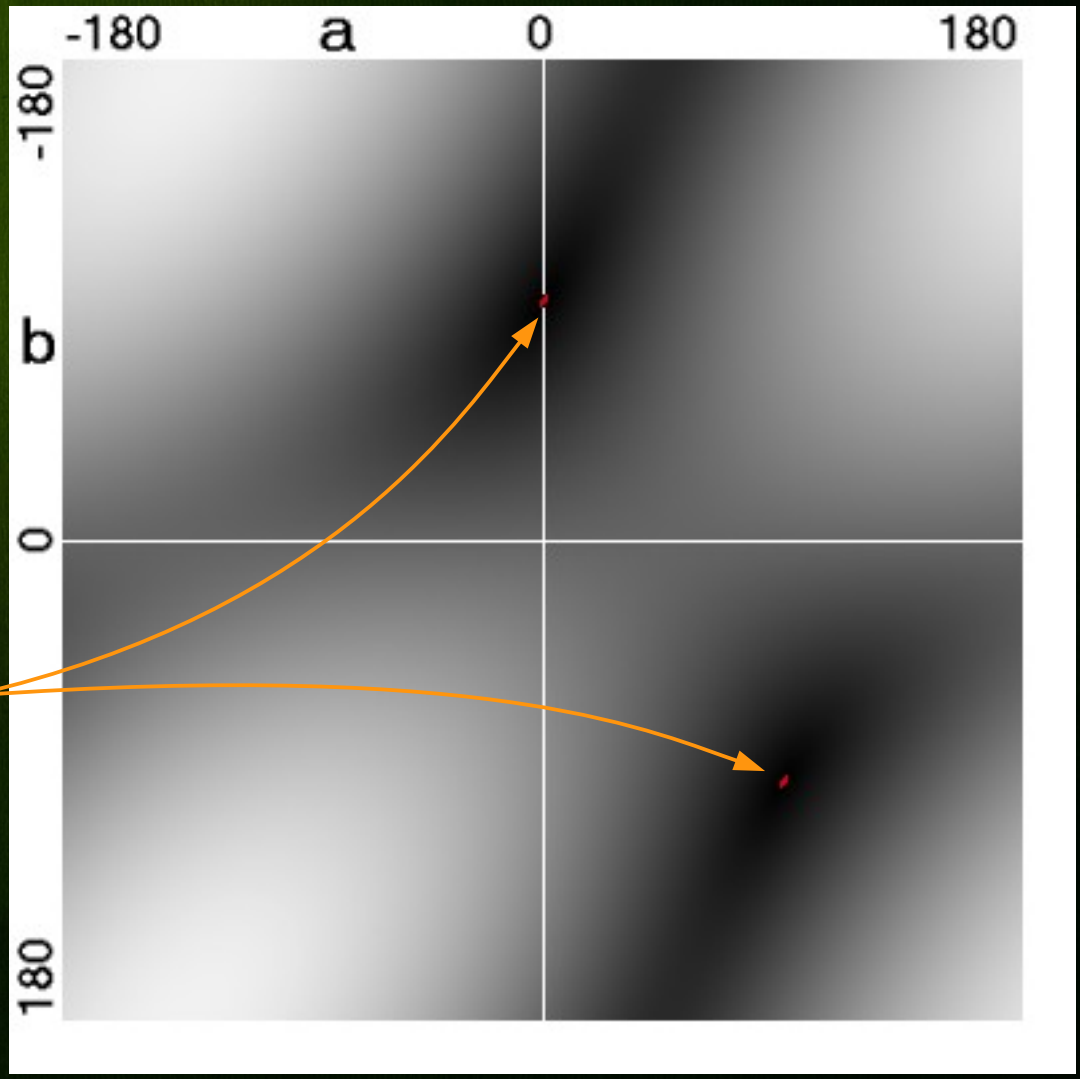


Inverse Kinematics - Minimization



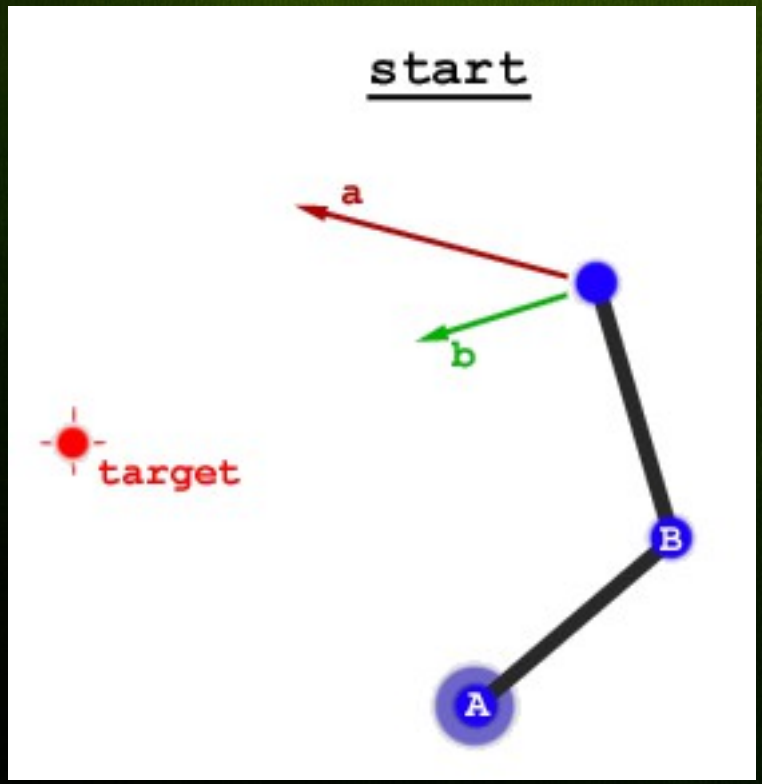
Solution = Minimum error

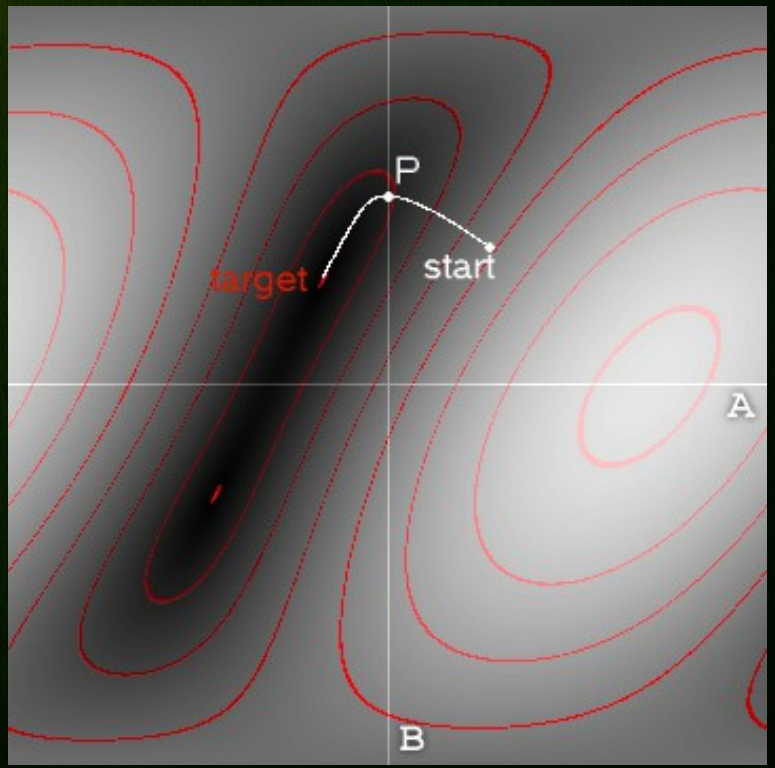
Any algorithm you can think of, for finding the lowest points on graphs can be used for Inverse Kinematics.



Inverse Kinematics - Minimization

- Simple gradient minimization – find better configuration gradient of angles





IK – Gradient by Measurement

- Pseudo-code: (for 2d, 2 angles)

  distance = GetDistance(a,b)

  while (Distance > 0.1) {
  - da = GetDistance(a+1,b) – GetDistance(a-1,b);
  - da = GetDistance(a,b+1) – GetDistance(a,b-1);
  - a -= da; b -= db;
  - Distance = GetDistance(a,b)

  }
- GetDistance(a,b){
  - Move joints using angles a and b, than return |target - tip|

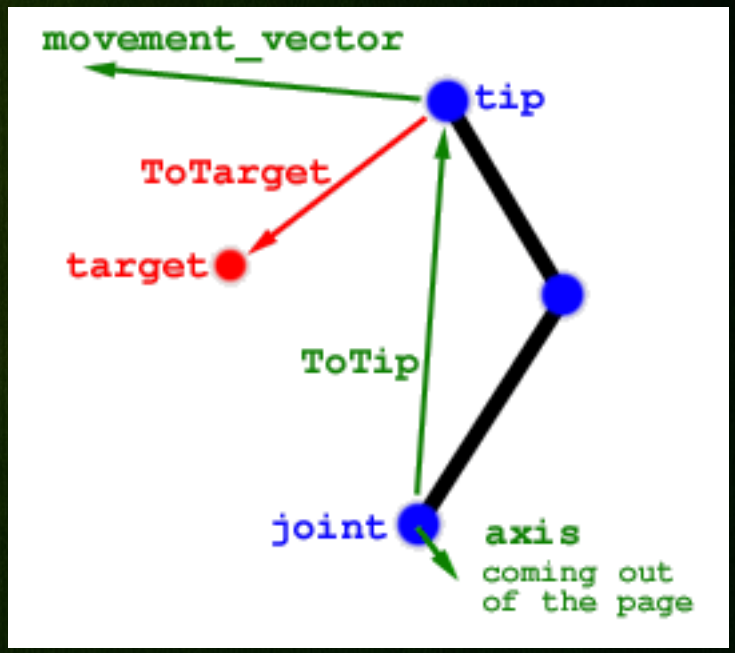
  }

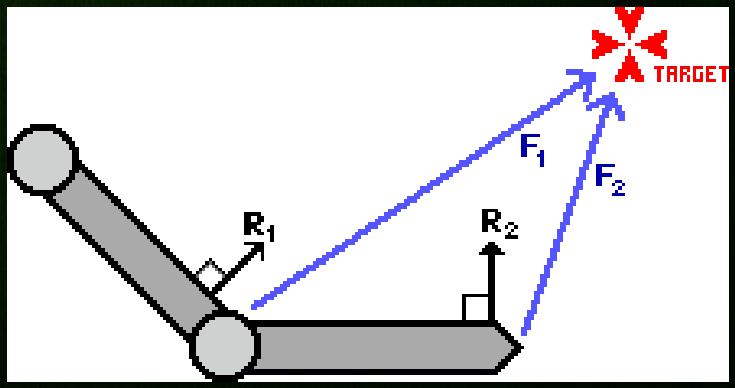
IK – Gradient by Calculation

- Pseudo-code:

  for each joint {
  - if 3d: axis = joint rotation axis
  - if 2d: axis = (0,0,1)
  - toTip = tip – jointCenter
  - toTarget = target – tip
  - moveDir = cross(toTip, axis)
  - gradient = dot(moveDir, toTarget)
  - alpha -= gradient

  }
- Force based algorithm





Procedural Animation

L-Systems
Fractals

L-Systems

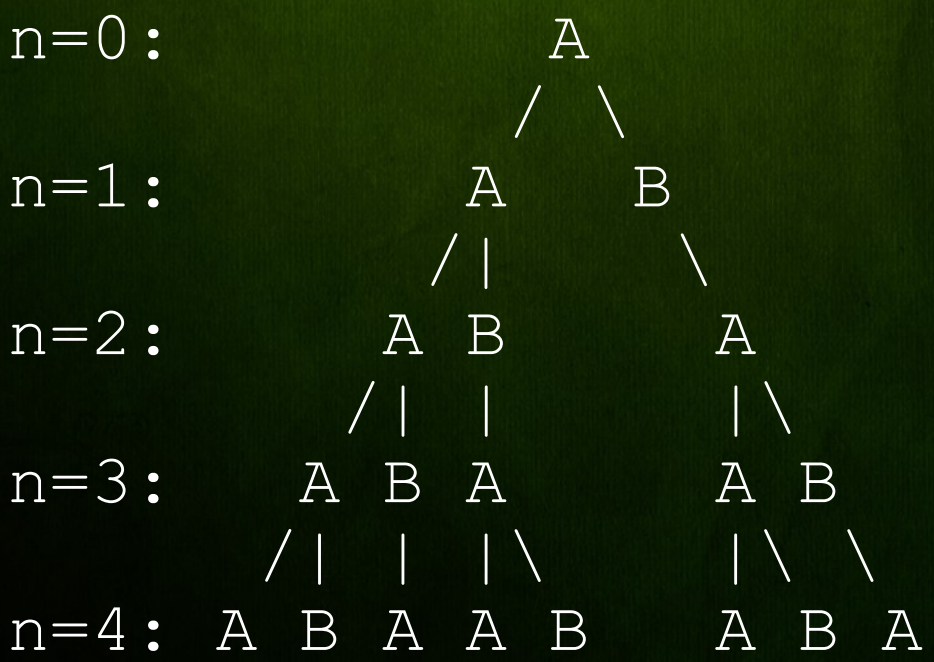
- Lindenmayer system (L-system) is a **parallel rewriting system** (formal grammar)
  - Most famously used to model the growth processes of plants
- Formal definition: L = (N,T,S,P)
  - L – L-system is a 4-tuple
  - N – Set of non-terminal letters (big letters)
  - T – Set of terminal letters (small letters)
  - P – Set of production rules
- Production Rule:
- Non-terminal → (non)terminal string
- Various sub-types exists (original: D0L system)

Example 1: Algae

- Lindenmayer's original L-system
  - for modelling the growth of algae.
- L = ({A,B}, {}, A, {(A → AB), (B → A)}
- Grammar results:
  - n = 0 : A
  - n = 1 : AB
  - n = 2 : ABA
  - n = 3 : ABAAB
  - n = 4 : ABAABABA
  - n = 5 : ABAABABAABAAB
  - n = 6 : ABAABABAABAABABAABABA
  - n = 7 : ABAABABAABAABABAABABAABAABABAABAAB

Example 1: Algae

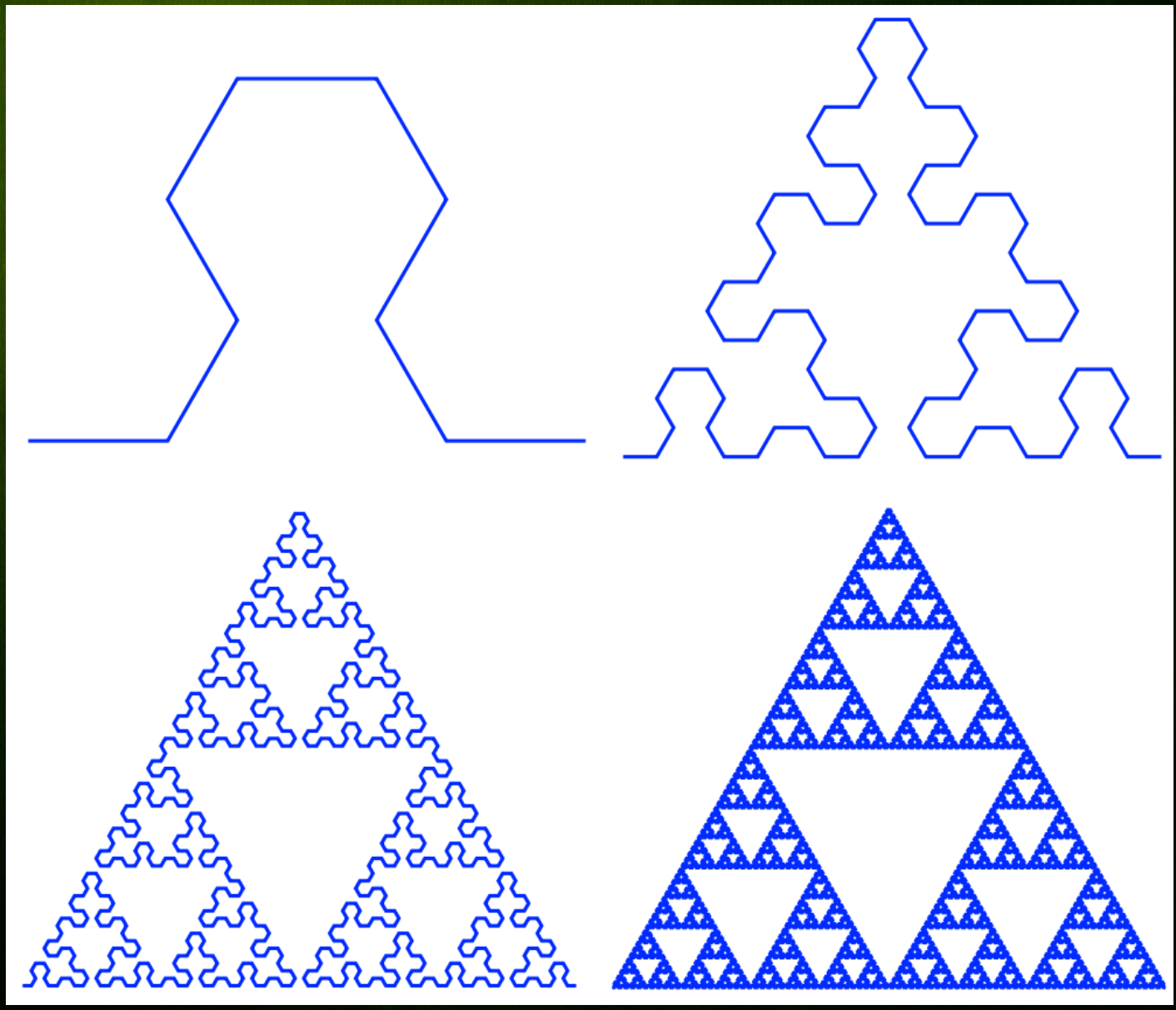
- n=0: start (axiom/initiator)
- n=1: the initial single A spawned into AB by rule (A → AB), rule (B → A) couldn't be applied
- n=2: former string AB with all rules applied, A spawned into AB again, former B turned into A
- n=3: note all A's producing a copy of themselves in the first place, then a B, which turns ...
- n=4: ... into an A one generation later, starting to spawn/repeat/recurse then



Example: Sierpinski triangle

- L = ({A, B}, {+, -}, A, {(A → B-A-B), (B → A+B+A)}
- Parameters: (angle = 60°)
- A, B: both mean "draw forward",
- +: means "turn left by angle" (turtle graphics)
- -: means "turn right by angle" (turtle graphics)
- The angle changes sign at each iteration so that the base of the triangular shapes are always in the bottom (they would be in the top and bottom, alternatively, otherwise)

Example: Sierpinski triangle



Motion Capture

Motion Capture

- Inspired by Rotoscoping, capturing frames by cameras
- Marker-based work-flow
  - Attach reflex markers on key parts of actors body (knees...)
  - Create skeleton and assign marker points
  - Capture video-sequence of moving actor (multiple cameras)
  - Use image based techniques to find 3d position of markers
  - Animate the skeleton by the reconstructed path data
- Pros: faster, simpler, more precise
- Cons: Marker retouching, complex motion = many markers


the
End
that was enough...