Character Animation

Rigging: Use of *bones* to make a *skeleton* (*armature*)

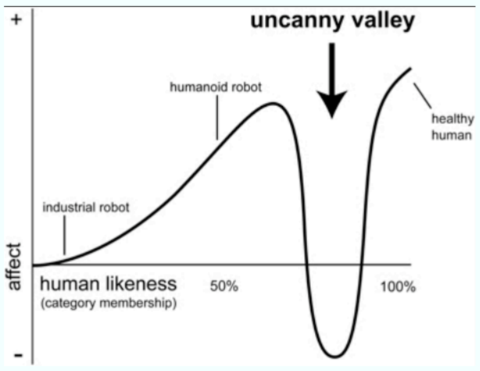
- Common to have one "root bone", other bones' positions are relative to it
  - muscles, especially facial ones, may be treated as variable-length bones
  - "tree of bones" – foot off shin off thigh off pelvis
  - for humanoid figures, it is commonplace to choose pelvis as root bone
  - but saddlesore gunfighter might suffer pointing problems if pelvis used as root
  - sometimes therefore a virtual bone is used as root – a floorplate say
  - termed a synthetic root bone (SRB) – "ground shadow" is common choice

Skinning: Association of mesh vertices with (generally multiple) bones
  - bones define an envelope of surrounding parts of mesh
  - as bones rotate relative to each other, mesh vertices must move too
  - interpolation is necessary

"Uncanny Valley"

In animation of human-like figures, greater realism gives increasingly satisfying result – but only up to a point. After that, people experience creepiness.

See "Ethan" in a previous assignment:



Ethan is really well modelled, but his impassive expression while attacking and being attacked do not fit well with his human-like movements and body shape. Things clearly non-human, like a gingerbread man, can be accepted more easily!

Copyright Image is shown to class here

Looping animations

- Many animations (walk, steam train, …) will loop through a series of poses
  - (but not all: dismemberment eg)
  - like keyframes: interpolation between poses will be required for smooth motion
    - many fewer poses then need to be created, less artist time required
- First and last poses of looping animation must then be identical
  - C0 – continuity of position
  - but if playing loop more than once, do not repeat both 1st and last each time
- Bone velocities at start and end should be identical
  - C1 – continuity of velocity (derivative of position)
- Bone accelerations probably need not be identical
  - C2 – continuity of acceleration (derivative of velocity) – probably not desired

Interpolation - a common theme

Vertices of skin mesh have their positions influenced by more than one bone

Skinning may involve deliberately placing some polygons and getting others positioned by formula

Poses between keyframes calculated by interpolation

- Rotational movement needs interpolation between keyframes
- Rotations and positions need to be cascaded through tree of bones

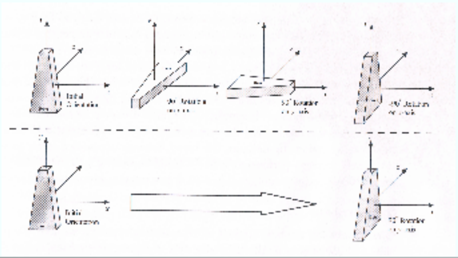
Colour shades might need to be interpolated

Interpolation as bones rotate: Euler angles

The start and end positions of a bone rotating in a joint are 3D angles
How to represent such angles? How to interpolate between such angles?

Euler angles give rotations about 3 axes, extrinsic (x y z) or intrinsic (pitch, roll, yaw)

- Order is important. Different toolsets may assume different orders ☹
- Spatially close angles may not be numerically close: difficulty for interpolation ☹
- Gimbal Lock problem arises: some orientations have multiple representations, also very fast changes of one angle may be needed near poles ☹



Multiple representations:
(Upper) Rotating 90° around x axis, then 90° around y axis, then 90° around z axis gives the same result as
(Lower) Rotating 0° around x axis, then 90° around y axis, then 0° around z axis
Note, the order is unchanged

Interpolation as bones rotate: Exponential Map

The "Exponential Map" gives another way to represent rotations in 3d

- A rotation is represented with 3 numbers, representing x, y and z components
- The ratio of x : y : z specifies a vector in 3D
- Its magnitude specifies, in radians, the clockwise rotation around that axis
- A zero-magnitude vector must be treated as describing no rotation at all

The space thus defined is non-euclidean

- Going infinitely far in a direction, you reach the same point infinitely many times
  - There are infinitely many ways to represent the same rotation ☹
  - e.g. π, 3π, 5π, 7π … so use just the semi-closed interval [0, 2π)

Interpolation between two rotations can be done using a technique that resembles expanding e^X function, but the result may involve unnatural twisting.

Interpolation as bones rotate: normalised quaternions

Quaternions: represent a rotation as a 4-tuple <x, y, z, w>

- x y and z can be interpreted as a 3D vector, giving the axis of rotation;
- the length of that vector gives sin(θ/2) where θ is the angle of rotation (radians)
- w gives cos(θ/2)
- Since $\sin^2(X)+\cos^2(X)=1$, making $x^2+y^2+z^2+w^2=1$ is normalisation of quaternion

Ways to interpolate normalised quaternions

- lerp ≈ nlerp: weighted average of each component individually, normalise result
- slerp (spherical interpolation): calculate great circle between start & end
- log-quaternion lerp (aka exponential map interpolation): exotic & complex, described in 1998 paper "Practical parameterization of rotations using the exponential map"

Spherical Interpolation – "slerp"

The familiar "Mercator's Projection" used in flat maps of the world has faults:

- it exaggerates the sizes of areas near the poles such as Greenland, Antarctica
- it is misleading when considering the paths long-distance journeys should follow

Best paths follow a so-called "great circle", best seen on a globe.

Desirable properties of interpolation techniques

1. Interpolation should be along the shortest path ("least torque", least twisting)
   - exponential map interpolation doesn't achieve this
2. Interpolation should proceed at constant speed
   - nlerp doesn't achieve this
3. Order of blending three rotations should not matter
   - slerp doesn't achieve this

It is proven that no technique could ever achieve all three. What to sacrifice?

In practice, interpolation in games takes place between relatively close orientations.
When difference is less than 45° nlerp speed is almost indistinguishable from slerp

So nlerp, being simplest, is therefore preferred
ie 4/5 of the way from rotation [x1 y1 z1 w1] to rotation [x2 y2 z2 w2] is
[0.2*x1+0.8*x2 0.2*y1+0.8*y2 0.2*z1+0.8*z2 0.2*w1+0.8*w2]
(which must then be normalised)

Moving bones: kinematics

Forward kinematics (FK) involves applying a rotation to a bone, and cascading that rotation to all bones that are daughters in tree of bones

Inverse kinematics (IK) involves figuring out how bones need to be positioned to get an “end effector” (a robotics heritage term) into a particular position

IK may run into problems

- There may be no solution: object or floor may be out of reach
- Solution will usually have to be found iteratively rather than analytically:
    - technique of *cyclic coordinate descent*
    - when more than one bone involved, this takes time
- two-bones problem when nearly straight: esp if knees must not bend backwards eg
- loop of bones may exist:
    - eg rifle held in two hands, attach it to one, compensate with other

One-bone IK

Useful for several common purposes

- pointing character’s eyes
- pointing game’s camera
- getting character’s foot facing forward and level on ground

Generally quite trivial

- Find current orientation of relevant bone, and its desired orientation
- Calculate rotation as transform (matrix), apply it to current orientation
- But rotations can accumulate in undesirable fashion, eg for cameras with an “up”

So sometimes, alternatively, “start over”

- have single standard resting orientation
- calculate transform from that to desired orientation

Multibone IK by cyclic coordinate descent

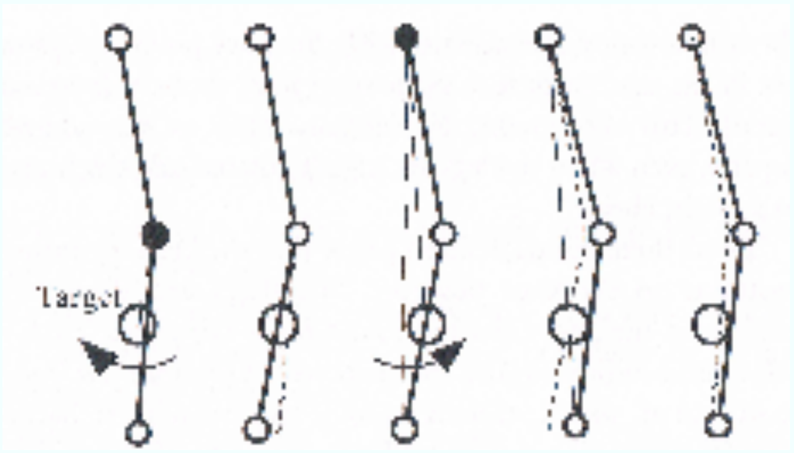
- Determine which joints in tree (from root bone to end effector) are allowed to move
  - spine eg may be required to be fixed when touching something with forefinger
- For each joint, find the rotation that minimises angle between end effector's position and its desired position
- Iterate until
  - reach limit on iterations, or
  - end effector's position is within acceptable tolerance

Iteration limit is very desirable since there may be no solution – out of reach!

Also, if joints have limits (eg knee cannot bend backwards), then just resetting it to straight may cause endless loop.

Cyclic coordinate descent has problem with 2-bone case when bones are nearly straight.

IK problem with two nearly straight bones



If it is desired to bring the ankle (lowest small circle) to the football (larger circle), repeatedly changing angles at the knee then hip then knee then hip … joints to minimise the then-remaining ankle-to-ball distance will take a very long time to converge on a solution.

Two-bone IK

This problem may be addressed cheaply and elegantly by using a geometric technique

Consider case of hip-knee-ankle
- Hip's position is known, and fixed
- Ankle's desired position is known
- Plane of knee joint is known
  - maybe variable, because of hip rotations in martial-arts kick, but still known
- Lengths of hip-to-knee and knee-to-ankle bones define spheres
- Equations will generally yield two possible solutions (or none)
  - if two, only one solution will respect the knee-forward constraint

Attachments

Characters may have other objects attached and moving with them – weapons eg
- Weapon may be treated as wholly separate object at all times
  - Bone in character bone tree serves to define position of root bone of weapon
- Weapon may be treated as part of character when toted
  - Character has two possible meshes: with weapon, without weapon
  - If weapon is dropped
    - instantly switch to without-weapon mesh
    - spawn new weapon object with its own separate mesh
    - let gravity take effect on it

Character may be an attachment to another object – helicopter, ledge, rescuer
- Position of character's root bone must be determined from the point of attachment
  - find position of hand
  - trace tree back to root bone to determine its position & orientation
  - apply transform to find positions & orientations of all other bones

Motion Extraction

The interplay of graphical animation and game logic brings problems of several kinds.

Generally, game logic is concerned with issues of where objects (incl characters) are, what space they occupy (for collisions), how fast they move, what state they are in.

Animations, by conventional wisdom, should not affect object locations, orientations

Nevertheless, it may be required that an object's game movement is coordinated somehow with its animation. Eg a walking character's feet should not slide around.

For this purpose, it may be necessary to use *motion extraction*

- to create poses which move the object's root bone, and
- modify them so that the root bone is (mostly) fixed, and
- to give the game logic access to the deltas to be applied to the poses

Linear Motion Extraction

- LME works well if there are no rotations or big changes in acceleration

- Determine position of root bone in first & last frames of animation
- Subtract appropriate proportion of total from each pose in animation
- Supply average linear velocity to game engine
- Keep *residual changes* with each pose of animation
  - the difference between proper position of root bone in the pose, and where the game logic will think it is when supplying proportion of overall movement

Extracting motion from an animation in this automatic way is better than requiring animator and programmer to agree on motion parameters. Changing the animation (to a higher ledge eg) will naturally cause game logic to adapt appropriately.

If a figure rotates in the course of an animation, LME will fail to detect any movement.

Composite Motion Extraction

Animations may involve rotations, or nonlinear overall motions.

- Movement around a running track is an example.
- If something happens to a character (killed, interrupted), with LME its game position will be all wrong

- CME involves
  - extracting the motion and orientation of root bone between each successive pose, rather than only start and end as in LME
  - encoding movement as relative to current orientation, rather than as absolute