CS 6204 Character Animation

Skeleton Based Full Body Animation

Yong Cao

Virginia Tech

Objective

What are the technical details for implementing a skeleton based full body animation system?

- ***Typical Programming Structure of Animation Systems***
- ***Skeleton***
  - Joint / Bone, Hierarchy
  - DOF, Representation of Rotation
- ***Animation Data***
  - Poses and Channels
  - Interpolation

Structure of Animation System

```
while (not finished) {
    MoveEverything();
    DrawEverything();
}
```

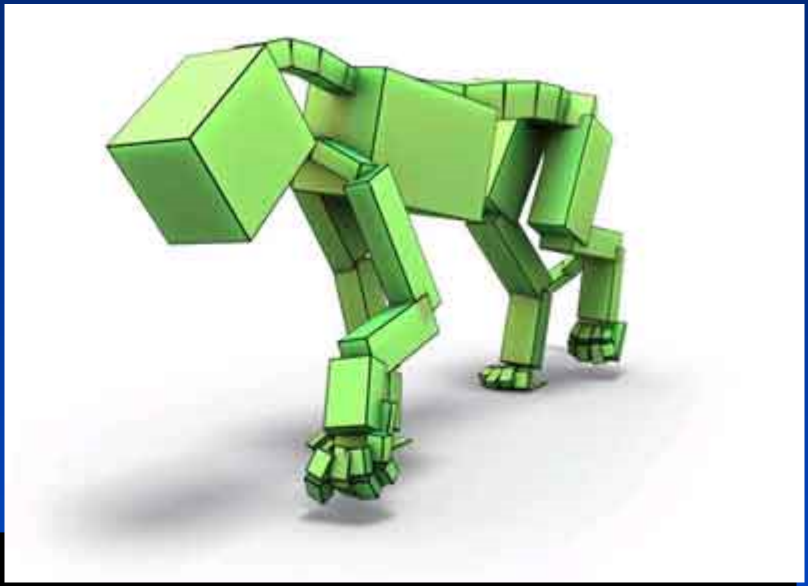
- Interactive vs. Non-Interactive
- Real Time vs. Non-Real Time

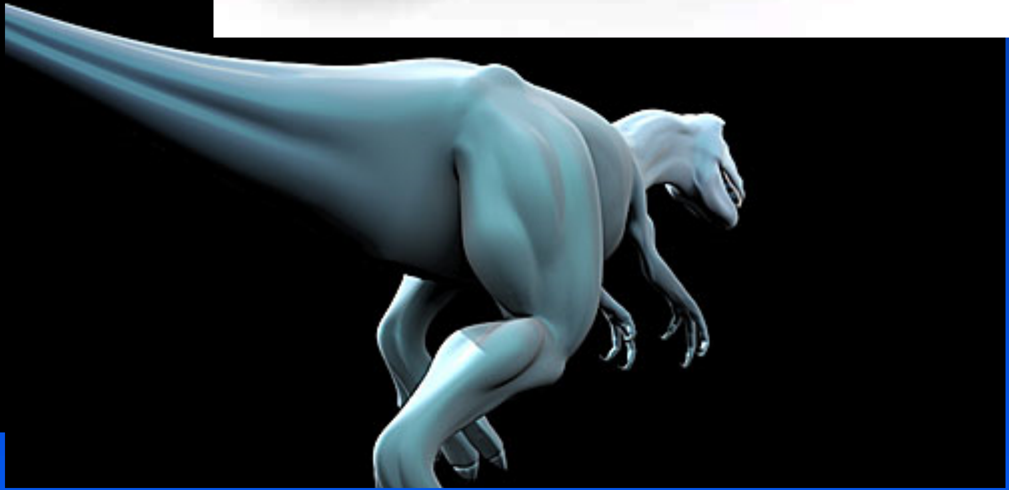
Structure of Animation System

```
while (not finished) {
    MoveEverything();
    DrawEverything();
}
```

- Can be implemented with Event-Driven design.
  - Such as Windows Message Passing
- OpenGL GLUT event handling library

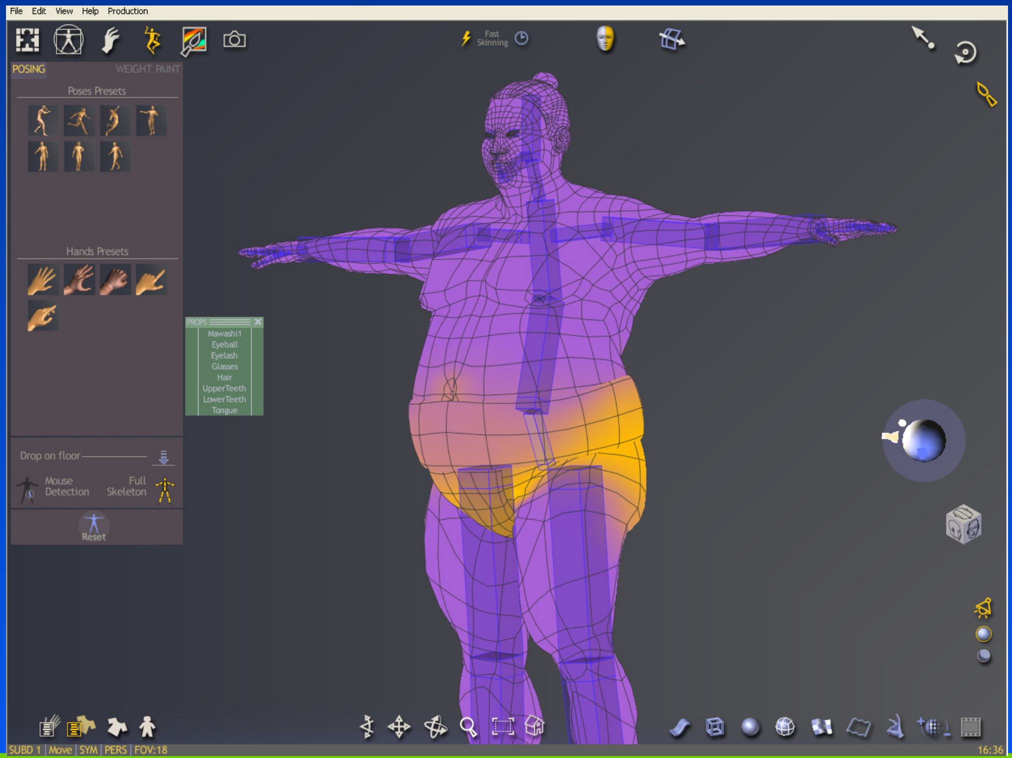
Animating a Character





Animating a Character

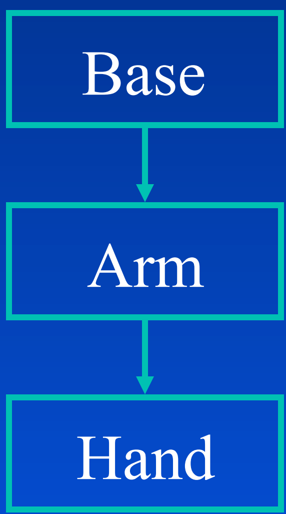
- ***Animating Mesh Vertices***
- ***Animating Vertex Groups***
  - How to group? **According to BONES / JOINTS**

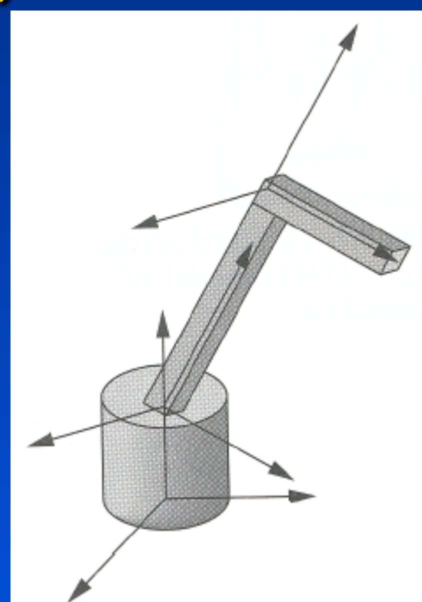


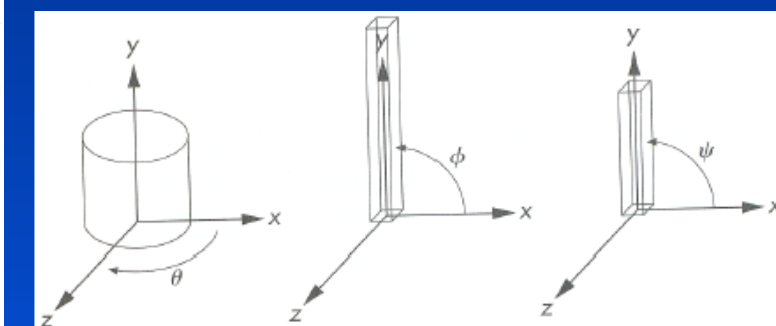
Articulated Figures with Joints

Parent joint VS Child joint

Global (character) Coordinate VS Local Coordinate

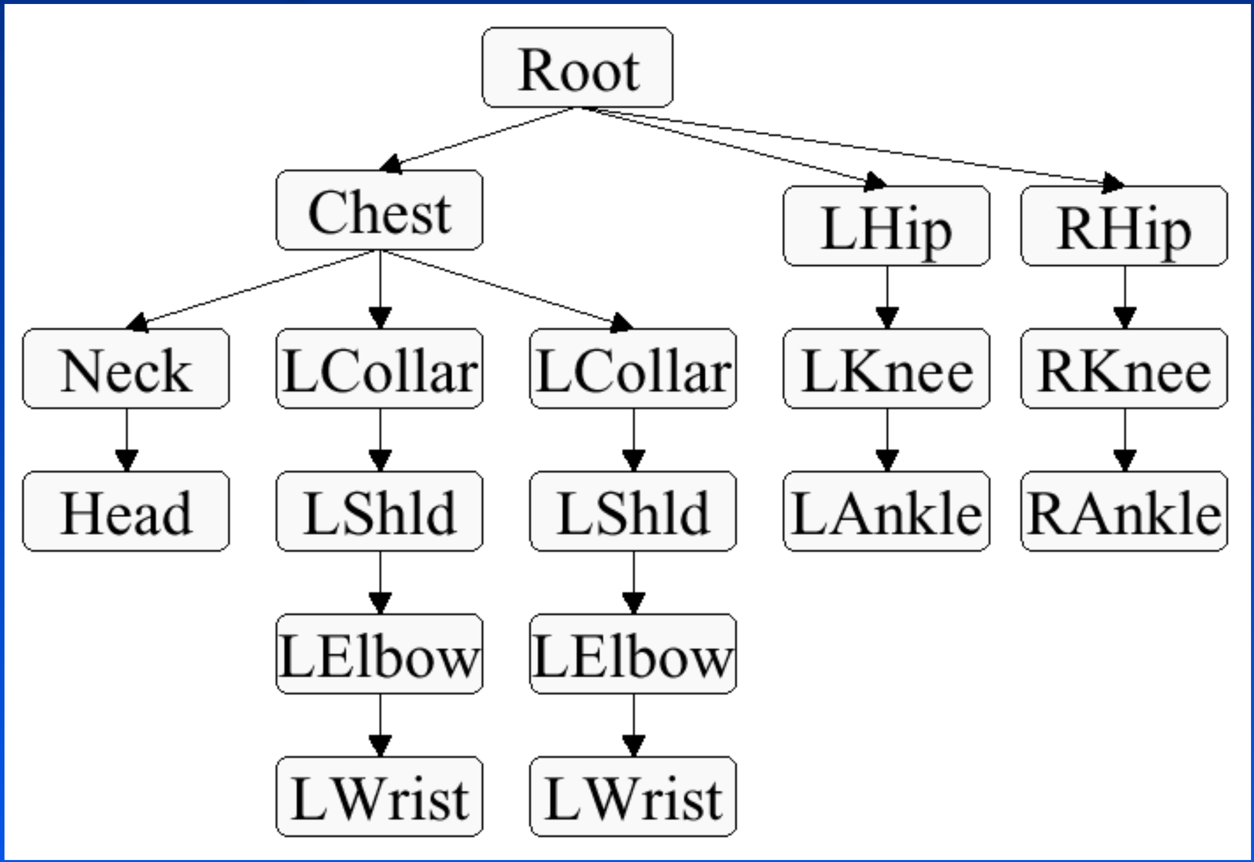


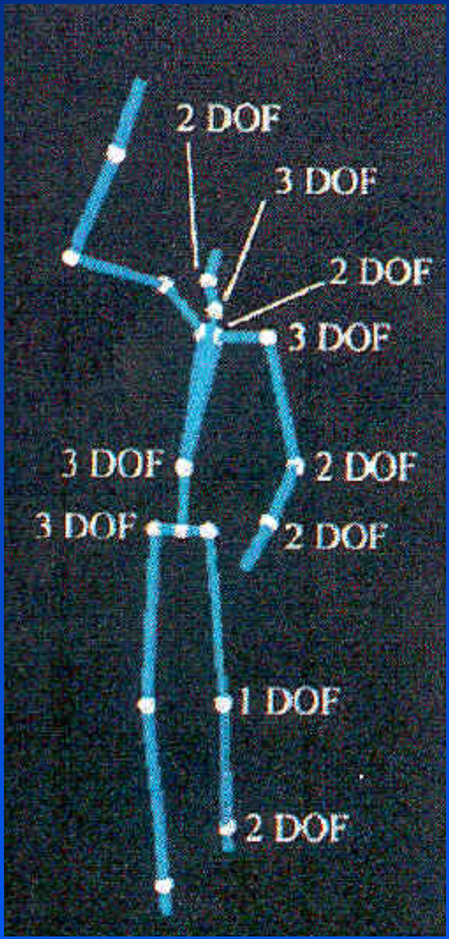




Joint Hierarchy and DOF

DOF: Degree of Freedom





Skeletons

Skeleton: A pose-able framework of joints arranged in a tree structure.

Joint: A joint allows relative movement within the skeleton. A rotation matrix.

Bone = Joint

Joints

Core Joint Data

- DOFs (N floats)
- Local matrix: **L**
- World matrix: **W**

Additional Data

- Joint offset vector: **r**
- DOF limits (min & max value per DOF)
- Type-specific data (rotation/translation axes, constants…)
- Tree data (pointers to children, siblings, parent…)

Animation Data

- *If a character has N DOFs, then a pose can be thought of as a point in N-dimensional pose space*

$$\mathbf{\Phi} = \begin{bmatrix} \phi_1 & \phi_2 & \ldots & \phi_N \end{bmatrix}$$

- *An animation can be thought of as a point moving through pose space, or alternately as a fixed curve in pose space*

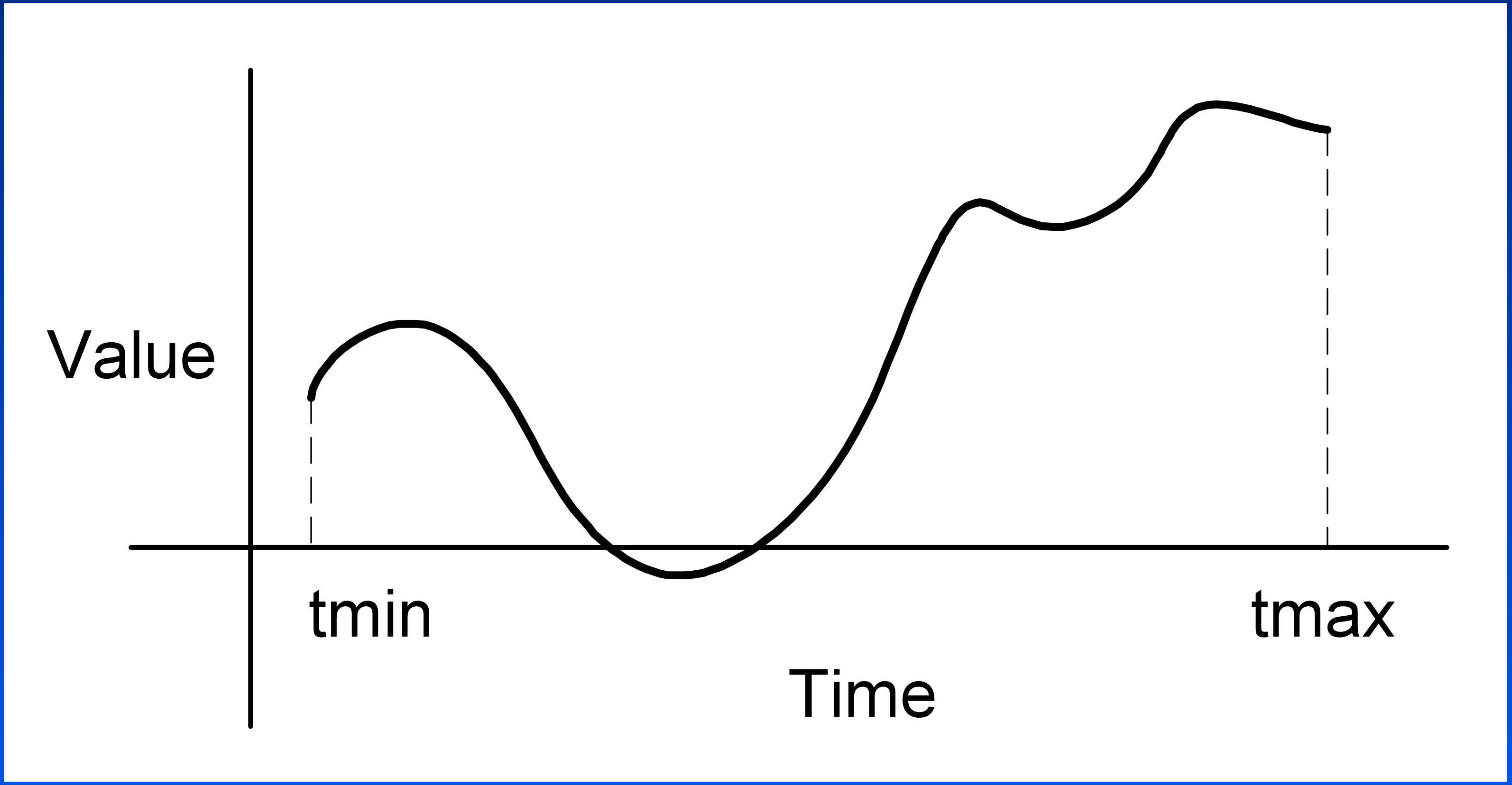
$$\mathbf{\Phi} = \mathbf{\Phi}(t)$$

Channels

- ***If the entire animation is an N-dimensional curve in pose space, we can separate that into N 1-dimensional curves, one for each DOF***

$$\phi_i = \phi_i(t)$$

Channels



Array of Channels

- ***An animation can be stored as an array of channels***

(NumDOFs x NumFrames)

Array of Poses

An alternative way to store an animation is as an array of poses

(NumFrames x NumDOFs)

Which is better, poses or channels?

Poses vs. Channels

Which is better?

It depends on your requirements.

The bottom line:

- Poses are faster
- Channels are far more flexible and can potentially use less memory

Poses vs. Channels

- ***Array of poses is great if you just need to play back some relatively simple animation and you need maximum performance.***
- ***Array of channels is essential if you want flexibility for an animation system or are interested in generality over raw performance***

Representing Rotation DOFs

Representing Orientations

Compared with Position, Orientation is not easy to represent:

Popular options:

- Euler angles
- Rotation vectors (axis/angle)
- 3x3 matrices
- Quaternions
- and more…

Euler’s Theorem

Euler’s Theorem: Any two independent orthonormal coordinate frames can be related by a sequence of rotations (not more than three) about coordinate axes, where no two successive rotations may be about the same axis.

Euler Angles

Any orientation can be represented by 3 numbers

A sequence of rotations around principle axes is called an Euler Angle Sequence

Assuming we limit ourselves to 3 rotations without successive rotations about the same axis, we could use any of the following 12 sequences:

XYZ	*XZY*	*XYX*	*XZX*
YXZ	*YZX*	*YXY*	*YZY*
ZXY	*ZYX*	*ZXZ*	*ZYZ*

Euler Angles

This gives us 12 redundant ways to store an orientation using Euler angles

Different industries use different conventions for handling Euler angles (or no conventions)

Euler Angles to Matrix Conversion

To build a matrix from a set of Euler angles, we just multiply a sequence of rotation matrices together:

$$\mathbf{R}_z \cdot \mathbf{R}_y \cdot \mathbf{R}_x = \begin{bmatrix} c_z & -s_z & 0 \\ s_z & c_z & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} c_y & 0 & s_y \\ 0 & 1 & 0 \\ -s_y & 0 & c_y \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 \\ 0 & c_x & -s_x \\ 0 & s_x & c_x \end{bmatrix}$$

$$= \begin{bmatrix} c_y c_z & s_x s_y c_z - c_x s_z & c_x s_y c_z + s_x s_z \\ c_y s_z & s_x s_y s_z + c_x c_z & c_x s_y s_z - s_x c_z \\ -s_y & s_x c_y & c_x c_y \end{bmatrix}$$

Euler Angle Order

As matrix multiplication is not commutative, the order of operations is important

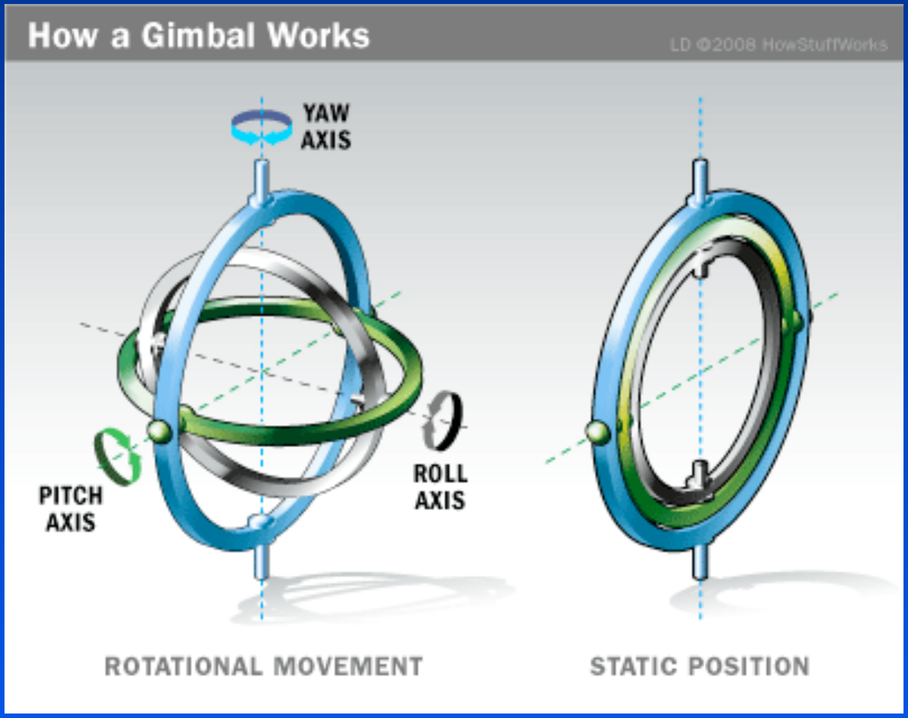
Using Euler Angles

- *To use Euler angles, one must choose which of the 12 representations they want*
- *There may be some practical differences between them and the best sequence may depend on what exactly you are trying to accomplish*

Gimbal Lock

One potential problem that they can suffer from is 'gimbal lock'

This results when two axes effectively line up, resulting in a temporary loss of a degree of freedom



Interpolating Euler Angles

One can simply interpolate between the three values independently

Interpolating near the ‘poles’ can be problematic

Note: when interpolating angles, remember to check for crossing the +180/-180 degree boundaries

Euler Angles

Euler angles are used in a lot of applications, but they tend to require some rather arbitrary decisions

They also do not interpolate in a consistent way (but this isn't always bad)

They can suffer from Gimbal lock and related problems

There is no simple way to concatenate rotations

Conversion to/from a matrix requires several trigonometry operations

They are compact (requiring only 3 numbers)