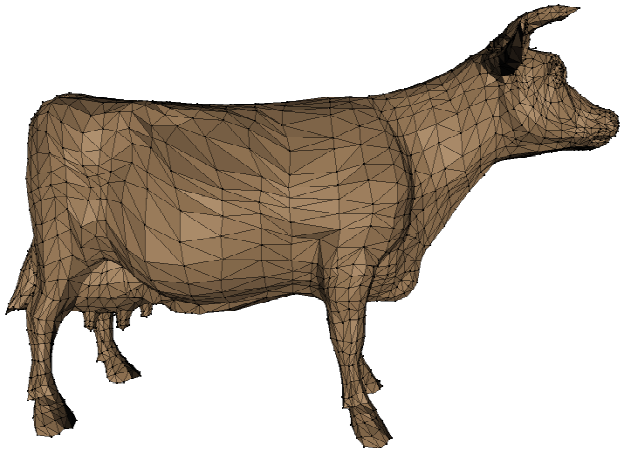
Where We Are: Static Models

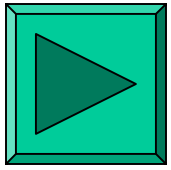
Our models are static sets of polygons



We can only move them via transformations

- translate, scale, rotate, or any other 4x4 matrix

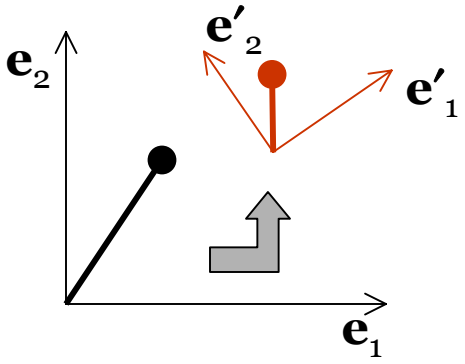
This does not allow us to simulate very realistic motion



So how do we fix this?

A Multitude of Spaces

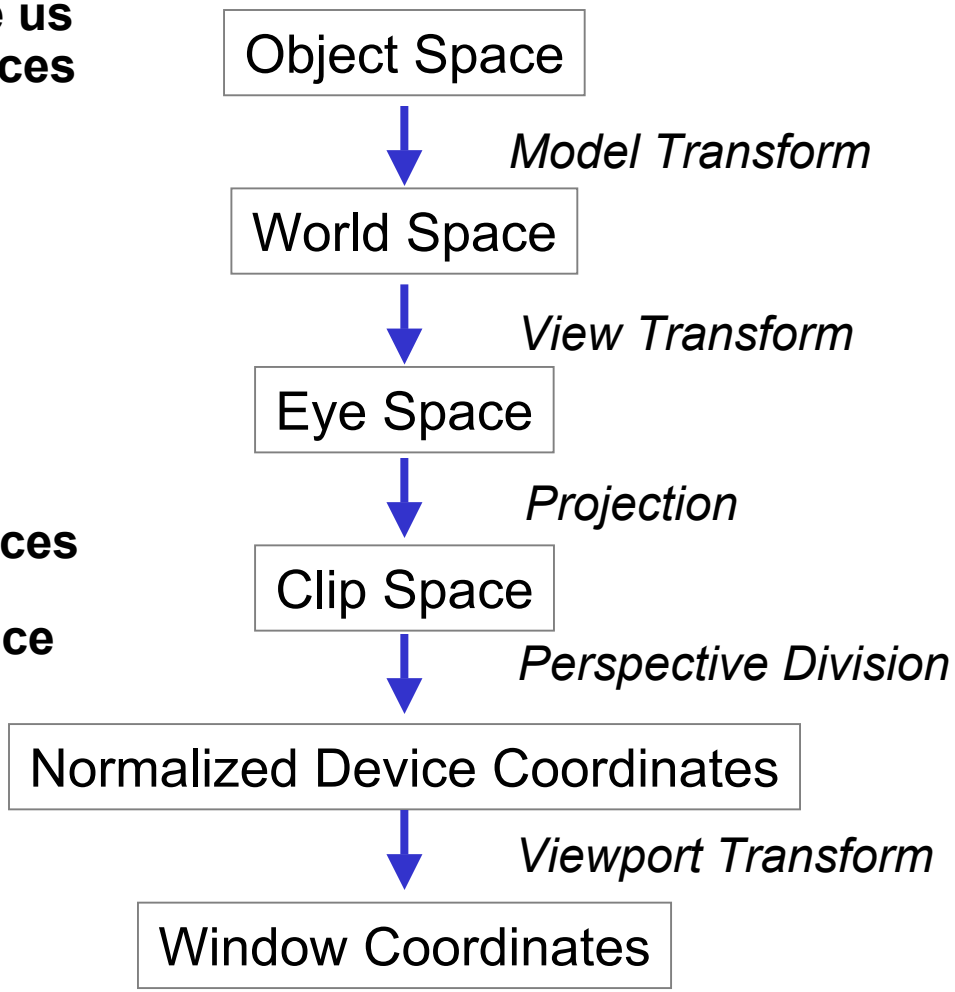
Recall that transformations move us between different coordinate spaces



Vertices move through many spaces

We change spaces for convenience

- as with the view transform



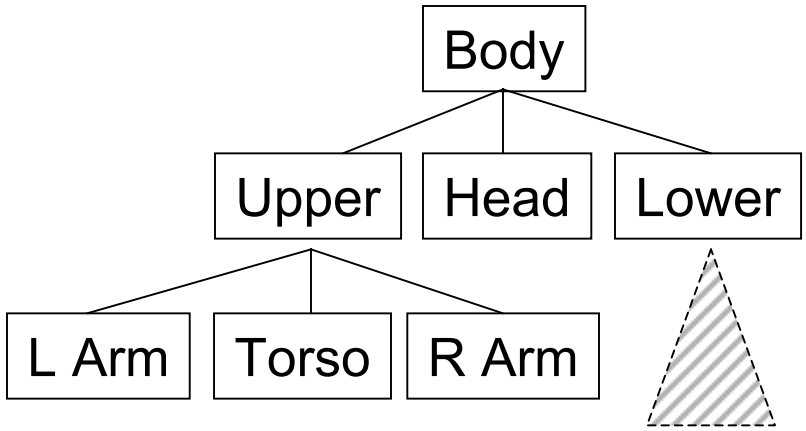
First Step: Break the thing Apart

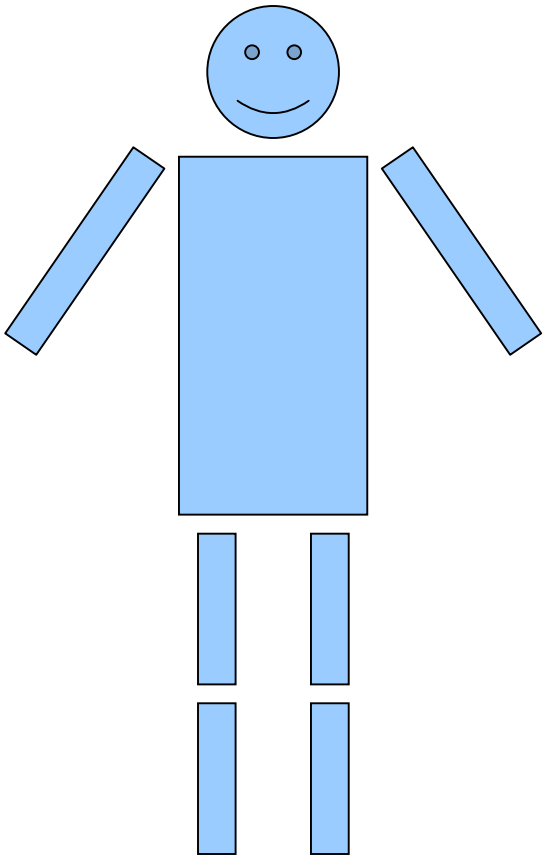
Model object as hierarchy of components

- each one can move
- and they move relative to parent

How do they move?

- with transformations





Introduce Transformation Nodes

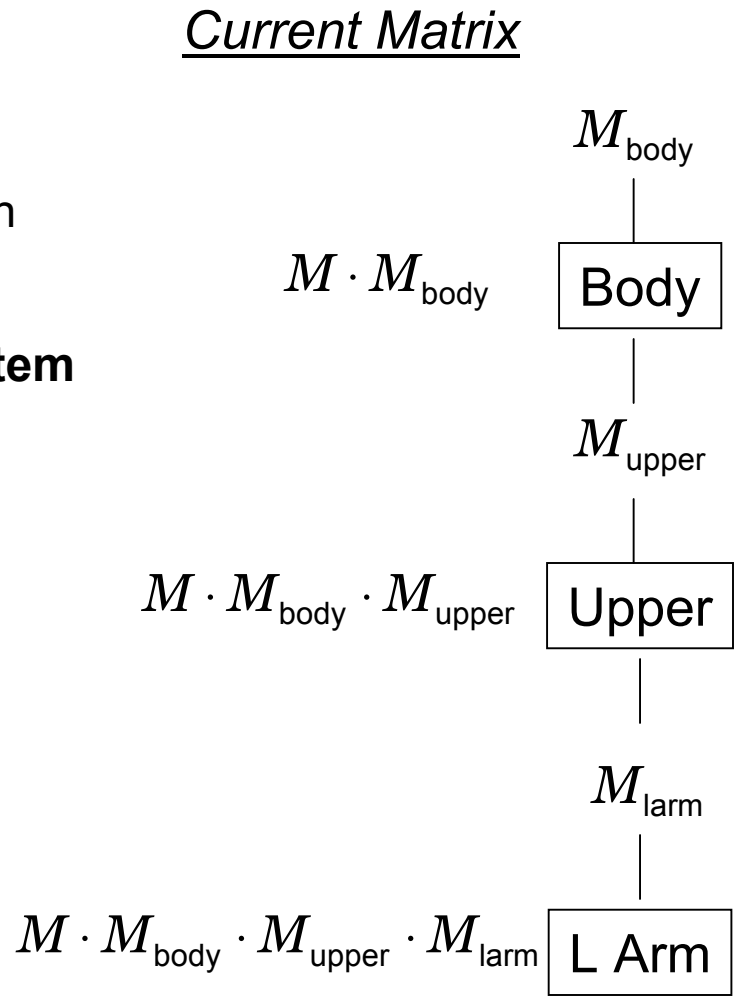
Traverse hierarchy during draw

- object nodes: draw them
- transform nodes
  - multiply into current matrix on way down
  - remove from current matrix on way up

Constantly changing local coordinate system

Allows changes at different scales

- apply rotation above “Body”
- vs. rotation above “L Arm”



Parameterizing Movement

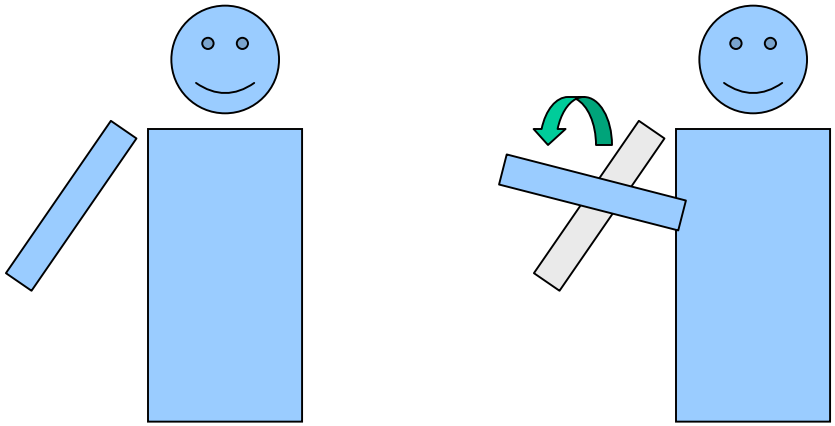
So how do we control the movement of parts?

We have all these transformation nodes

- 4x4 matrix = 16 coefficients
- obviously don't want to specify manually!

Could just specify sets of rotate, translate, scale

- much handier
- but still lacks one crucial thing — things might fall apart



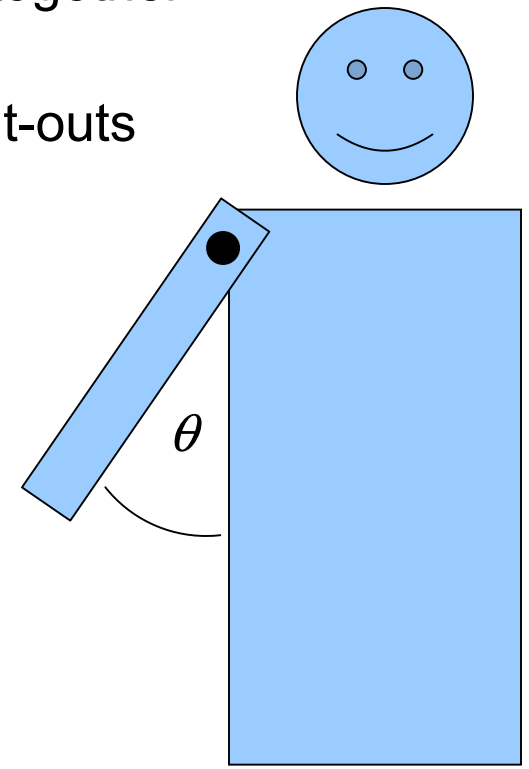
Parameterizing Movement

We want to link things with joints

Example for a shoulder

- specify corresponding points that remain together
- provide a joint angle parameter
- just like building people with cardboard cut-outs

Construct transform from parameters



Building Hierarchies

We need some good way to build them

- manual manipulation — select objects and hit “Group” button
- graph layout — draw graphs directly
- textual description of hierarchy
- write scripts to generate hierarchy

And we need to manipulate the transformations

- type in rotate, scale, translate values
- attach parameters to GUI elements (e.g., sliders)
- write little procedural controllers
  - for instance, “increment angle every 1.3 seconds”
  - acts sort of like a motor

Matrix Stacks

Instead of a current matrix, we need a matrix stack

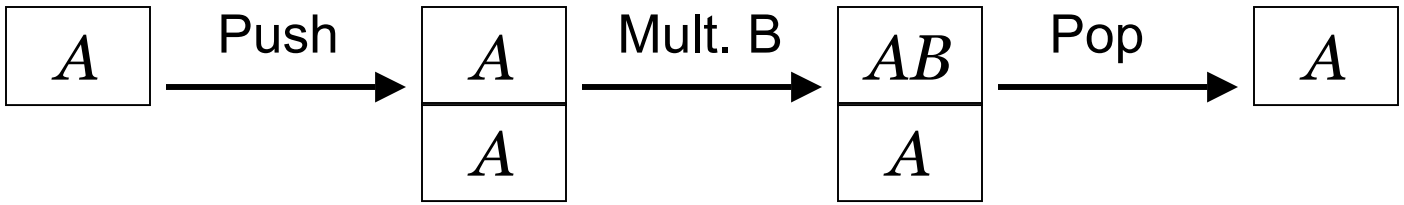
- current matrix is just the top of the stack

Stack operations

- PUSH — duplicate matrix on top — glPushMatrix()
- POP — remove matrix on top — glPopMatrix()

When traversing hierarchies

- PUSH on entering transformation node
- multiply transformation into current matrix
- descend to children
- POP when returning up the tree



Implementing Transformation Nodes

Be careful about transformation order

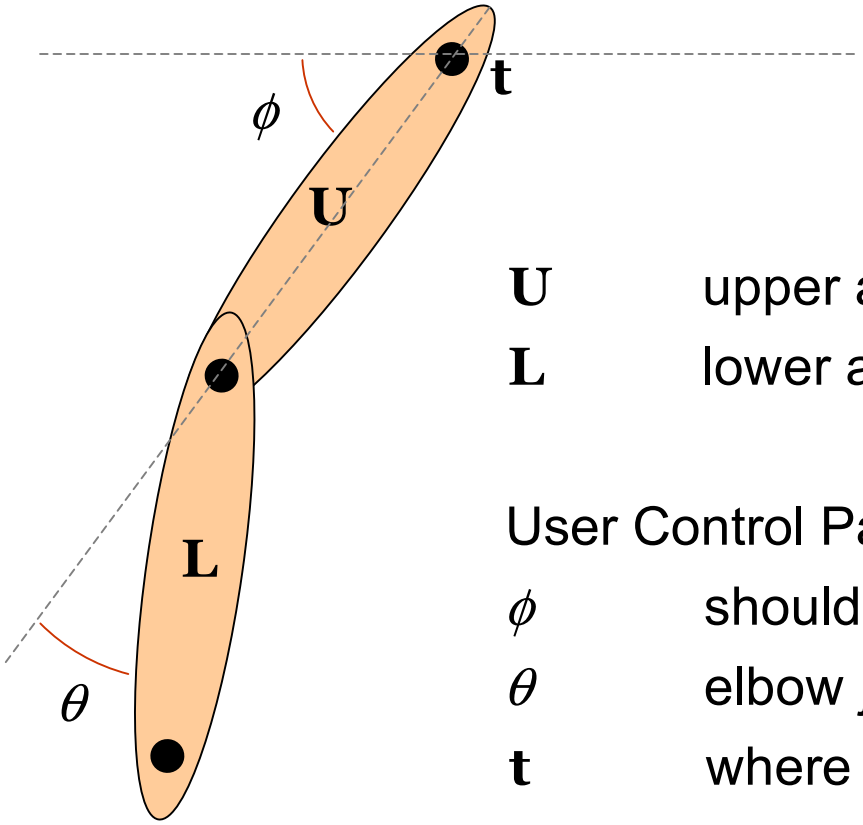
- (usually) want to scale before rotation
- (usually) want to rotate before translation

Simple 2-D Case:

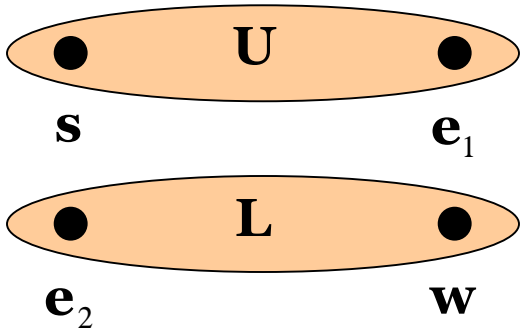
```
glPushMatrix();
glTranslatef(dx, dy);        // Further translation
glTranslatef(cx, cy);        // Back to center
glRotatef(angle, 0, 0, 1);
glScalef(s, t);
glTranslatef(-cx,-cy);       // Center to origin
… descend to children …
glPopMatrix();
```

Example: Building a Simple Arm

We want to model this arm as a hierarchy.



Out of these basic components.



U	upper arm
L	lower arm (forearm)

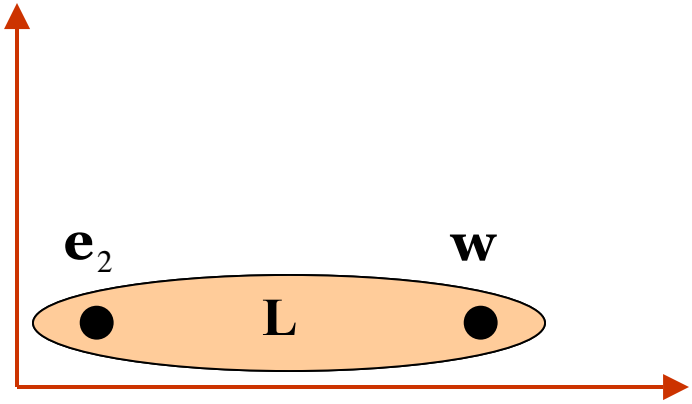
User Control Parameters:

ϕ	shoulder joint angle
θ	elbow joint angle
t	where shoulder meets torso

Positioning the Forearm

Initially: segments in same position

$\mathbf{s}$	shoulder joint location
$\mathbf{e}_1, \mathbf{e}_2$	elbow joint locations
$\mathbf{w}$	wrist joint location

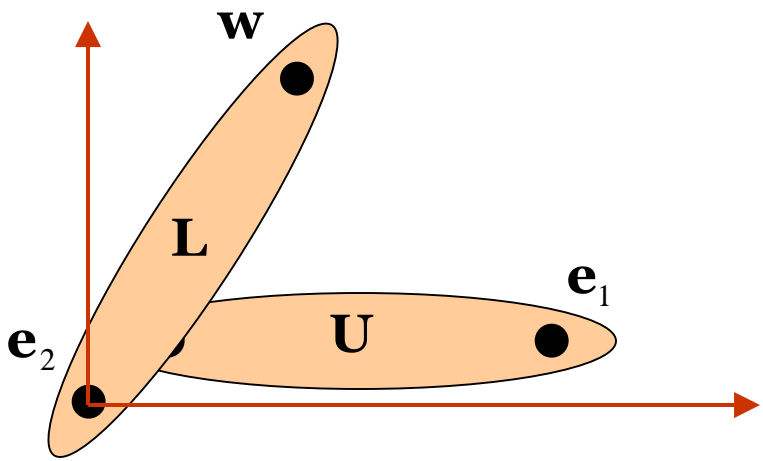


First: perform elbow rotation

- translate elbow joint to origin
- rotate by given angle

(1) translate($-\mathbf{e}_2$)

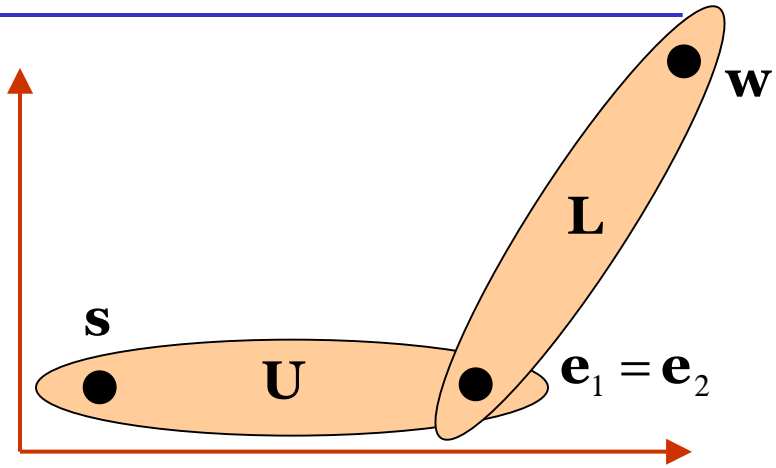
(2) rotate(θ)



Attaching Forearm to Upper Arm

Second: align corresponding elbows

(3) translate($\mathbf{e}_1$)

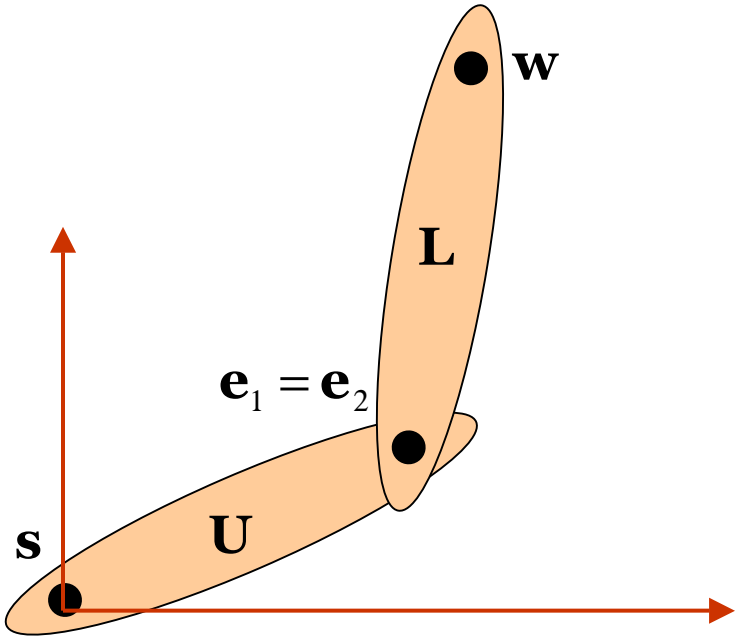


Third: perform shoulder rotation

- must operate on *whole* arm

(4) translate($-\mathbf{s}$)

(5) rotate(ϕ)

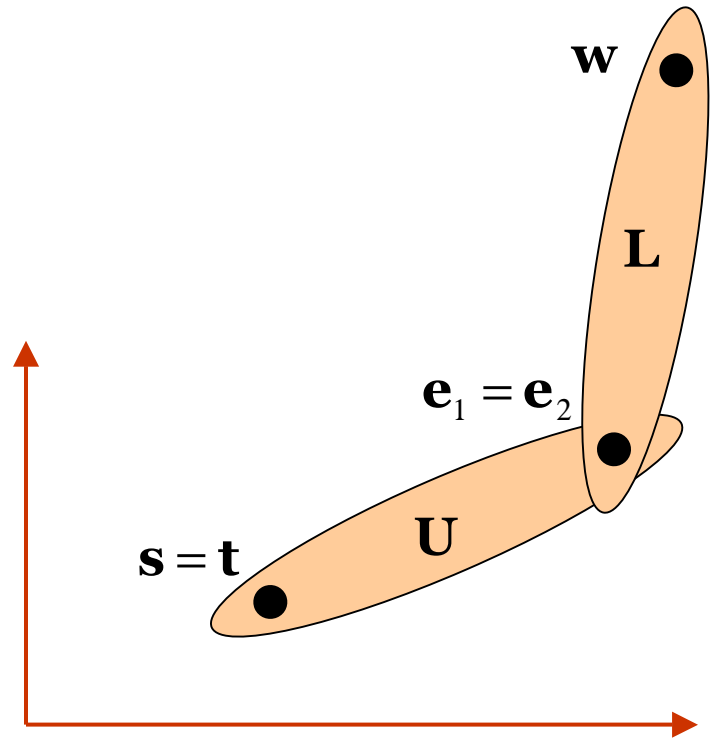


Placing the Shoulder

Fourth: put shoulder in right spot

(6) translate($\mathbf{t}$)

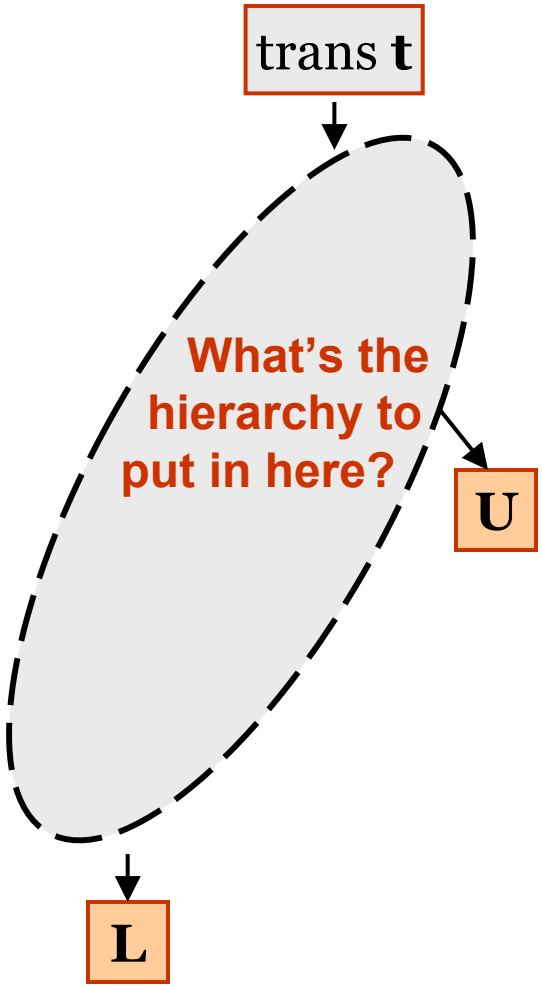
And we're done!



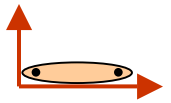
Important things to notice

- presents limited control knobs
- automatically handle interconnection
  - e.g., elbow joint

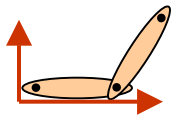
Exercise: Converting to Hierarchy



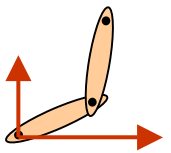
(1) translate($-\mathbf{e}_2$)
(2) rotate(θ)



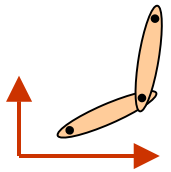
(3) translate($\mathbf{e}_1$)



(4) translate($-\mathbf{s}$)
(5) rotate(ϕ)



(6) translate($\mathbf{t}$)



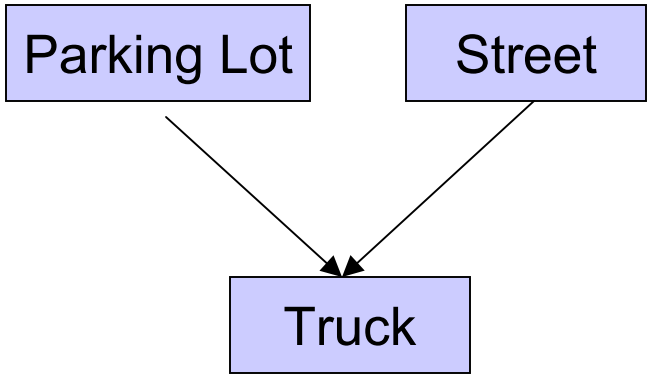
Scene Graphs

This idea can be extended to the whole scene

- collect every object into a single hierarchy

Provides several nice advantages

- natural way of defining bounding volumes for culling
- can instance same model in many places
  - but graph is no longer a tree
  - it's a more general DAG
- can introduce new node types also
  - light nodes
  - material nodes



OpenGL State Stack

In OpenGL, you can also push/pop state variables

- glPushAttrib(…)
- glPopAttrib()

Pass to glPushAttrib() a bitfield describing what to push

- GL_ALL_ATTRIB_BITS
- GL_ENABLE_BIT — everything set by glEnable
- GL_LIGHTING_BIT — light position, colors, materials, …
- GL_CURRENT_BIT — current color, normal, …
- and several others

How to Generate Animation?

We can create & parameterize models now

- design the geometry
- set up a bunch of control knobs (e.g., joint angles)

But how do we animate these models

- don’t want to manually tweak transformation parameters

We’ll specify parameters as functions of time

- but we need to do this conveniently
  - no writing out explicit polynomial functions

Recall How We Display Animation

We create animated behavior just like movie projectors

- display a sequence of still images in rapid succession
- creates the illusion of continuous motion
- typically want 30 frames/sec, and definitely higher than 10

Given some parameterized hierarchical model

- for every frame, we calculate the correct parameter values
- we set our hierarchy control knobs appropriately
- and we draw the scene in its current state

We’re going to need to figure out how to control parameters

- key-framing — specify poses and automatically interpolate
- physically-based motion — simulate physical laws