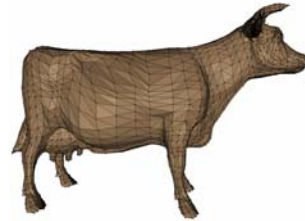


## Static Models

Our models are static sets of polygons



We can only move them via transformations

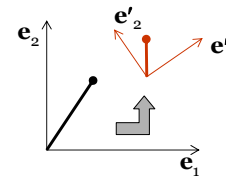
- translate, scale, rotate, or any other 4x4 matrix

This does not allow us to simulate very realistic motion

So how do we fix this?

## A Multitude of Spaces

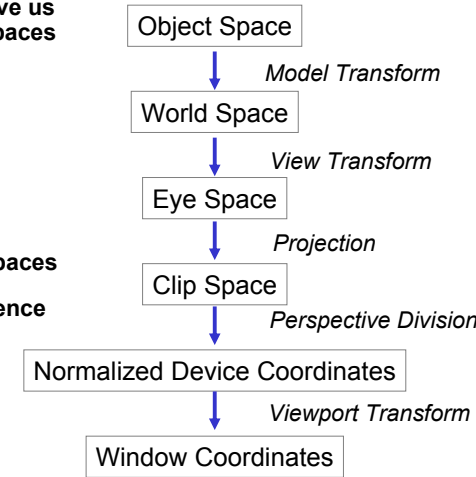
Recall that transformations move us between different coordinate spaces



Vertices move through many spaces

We change spaces for convenience

- as with the view transform



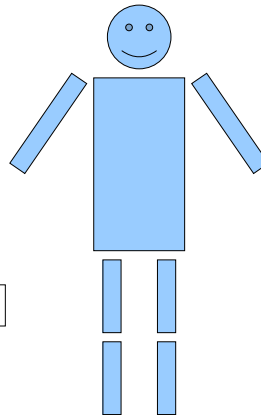
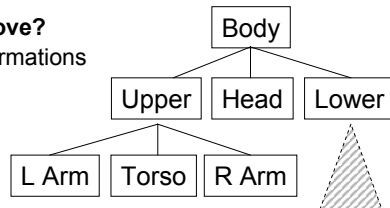
## First Step: Break the thing Apart

Model object as hierarchy of components

- each object node has a local coordinate system
- each component defined relative to parent
- each one can move
- and they move relative to parent

How do they move?

- with transformations



## Introduce Transformation Nodes

Traverse hierarchy during draw

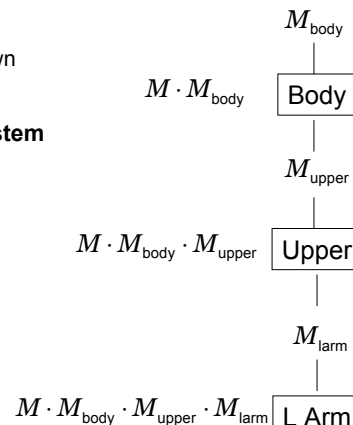
- object nodes: draw them
- transform nodes
  - multiply into current matrix on way down
  - remove from current matrix on way up

Constantly changing local coordinate system

Allows changes at different scales

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

Current Matrix



## Parameterizing Movement

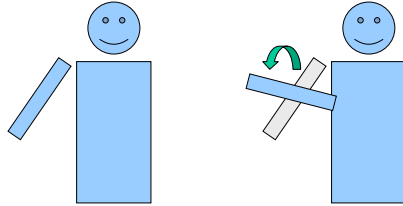
So how do we control the movement of parts?

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



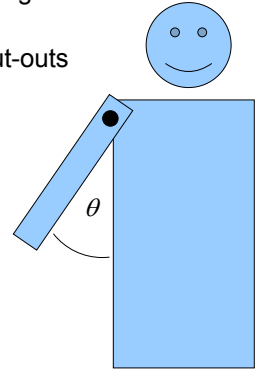
## Parameterizing Movement

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

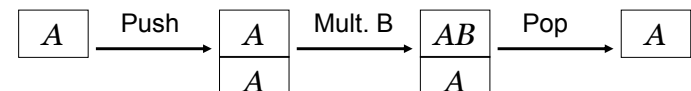
Stack operations

- PUSH — duplicate matrix on top
- POP — remove matrix on top

`glPushMatrix()`  
`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:

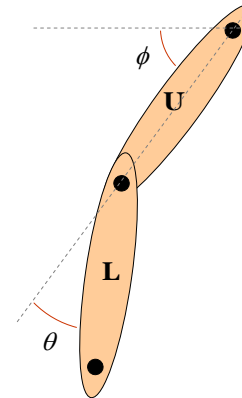
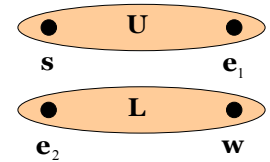
```

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)

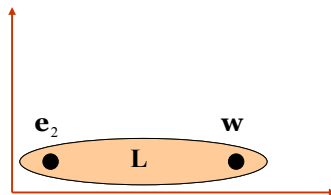
User Control Parameters:

- $\phi$  shoulder joint angle
- $\theta$  elbow joint angle
- $t$  where shoulder meets torso

## Positioning the Forearm

Initially: segments in same position

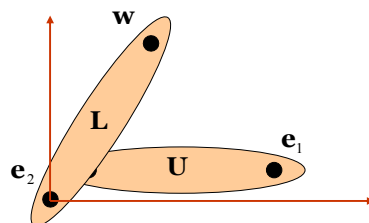
- $s$  shoulder joint location
- $e_1, e_2$  elbow joint locations
- $w$  wrist joint location



First: perform elbow rotation

- translate elbow joint to origin
- rotate by given angle

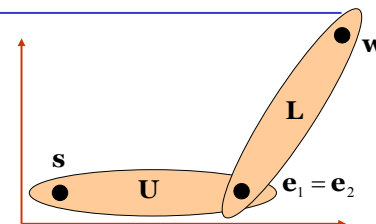
- (1)  $\text{translate}(-e_2)$
- (2)  $\text{rotate}(\theta)$



## Attaching Forearm to Upper Arm

Second: align corresponding elbows

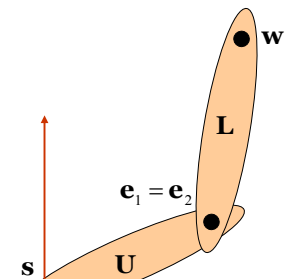
- (3)  $\text{translate}(e_1)$



Third: perform shoulder rotation

- must operate on *whole* arm

- (4)  $\text{translate}(-s)$
- (5)  $\text{rotate}(\phi)$



## Placing the Shoulder

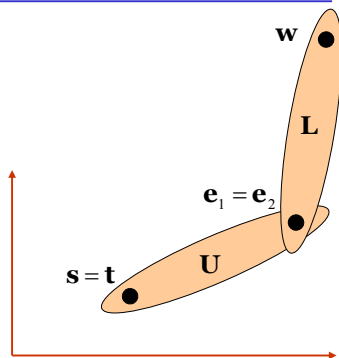
Fourth: put shoulder in right spot

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

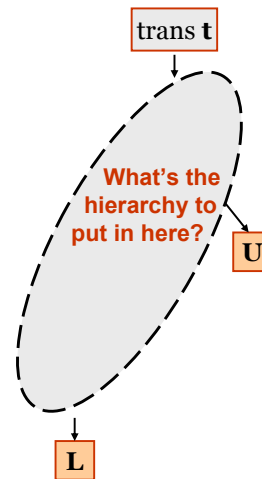
And we're done!

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( $\theta$ )



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



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



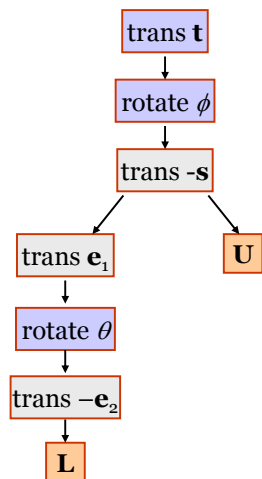
(5) rotate( $\phi$ )



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



## Converting to Transformation Hierarchy



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



(2) rotate( $\theta$ )



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



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



(5) rotate( $\phi$ )



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



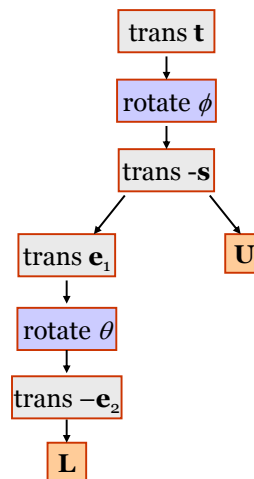
## Notable Properties of Transform Hierarchy

Geometry is always at the leaves

- internal nodes are transforms

There are 2 types of transforms

- **structural** (shown in gray)
  - fixed at design time
  - keeps things together
- **control knobs** (shown in blue)
  - variable parameters
  - controlled by user



## Scene Graphs

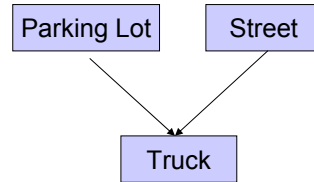
---

### 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