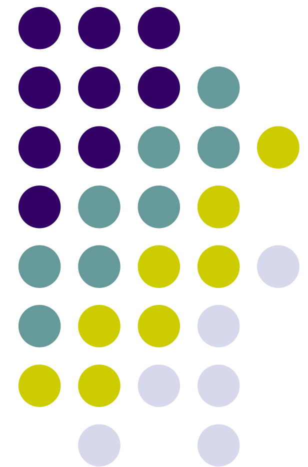
Computer Graphics
CS 543 – Lecture 8 (Part 1)
Hierarchical 3D Models

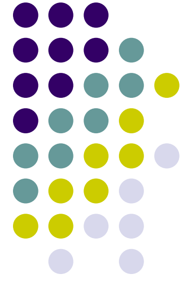
Prof Emmanuel Agu

Computer Science Dept.

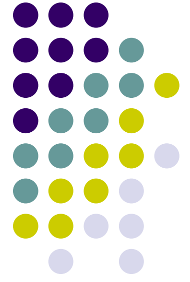
Worcester Polytechnic Institute (WPI)



Objectives

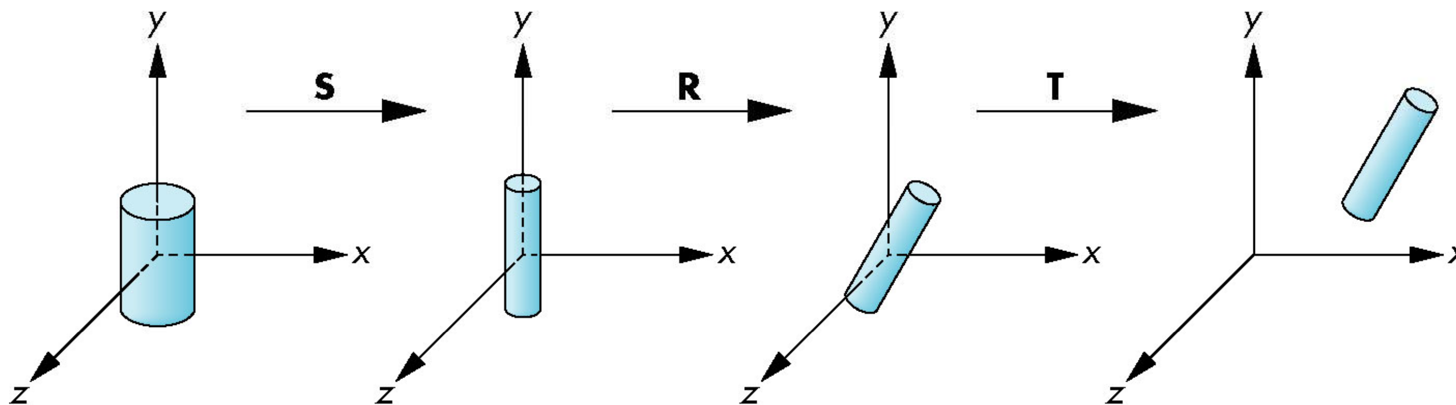


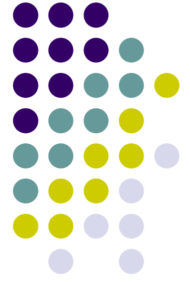
- Examine the limitations of linear modeling
  - Symbols and instances
- Introduce hierarchical models
  - Articulated models
  - Robots
- Introduce Tree and DAG models



Instance Transformation

- Start with unique object (a *symbol*)
- Each appearance of object in model is an *instance*
  - Must scale, orient, position
  - Defines instance transformation



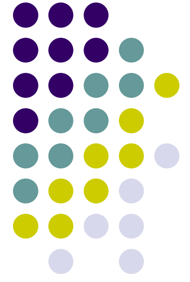


Symbol-Instance Table

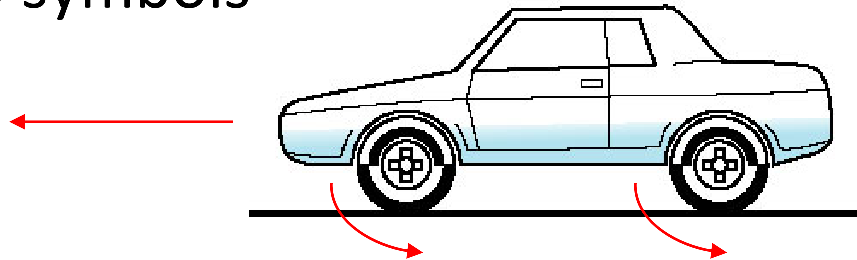
Can store a model by assigning number to each symbol and storing parameters for instance transformation

Symbol	Scale	Rotate	Translate
1	s_x, s_y, s_z	$\theta_x, \theta_y, \theta_z$	d_x, d_y, d_z
2			
3			
1			
1			
.			
.			

Relationships in Car Model

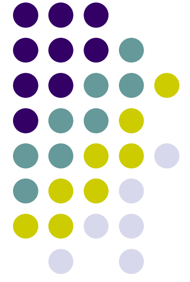


- Symbol-instance table does not show relationships between parts of model
- Consider model of car
  - Chassis + 4 identical wheels
  - Two symbols



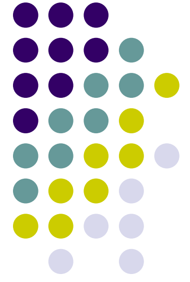
- Rate of forward motion determined by rotational speed of wheels

Structure Through Function Calls



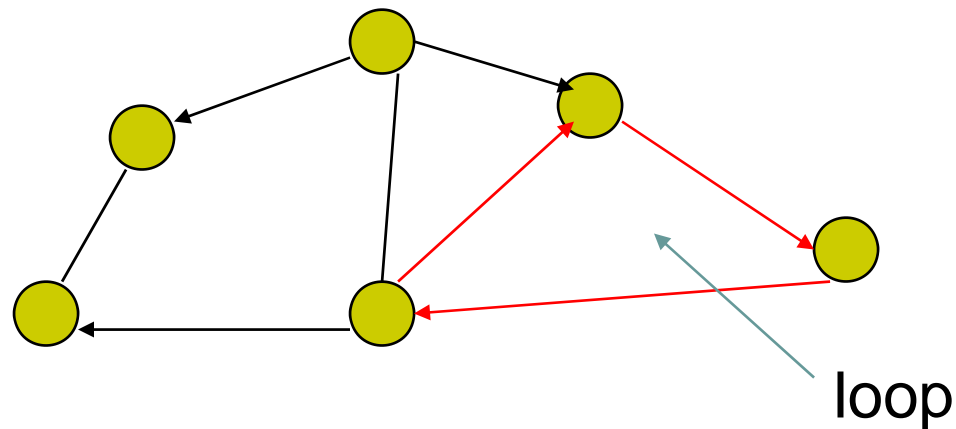
```
car(speed)
{
    chassis()
    wheel(right_front);
    wheel(left_front);
    wheel(right rear);
    wheel(left_rear);
}
```

- Fails to show relationships well
- Look at problem using a graph

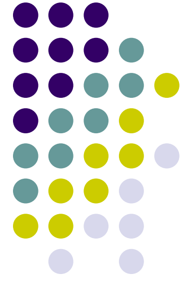


Graphs

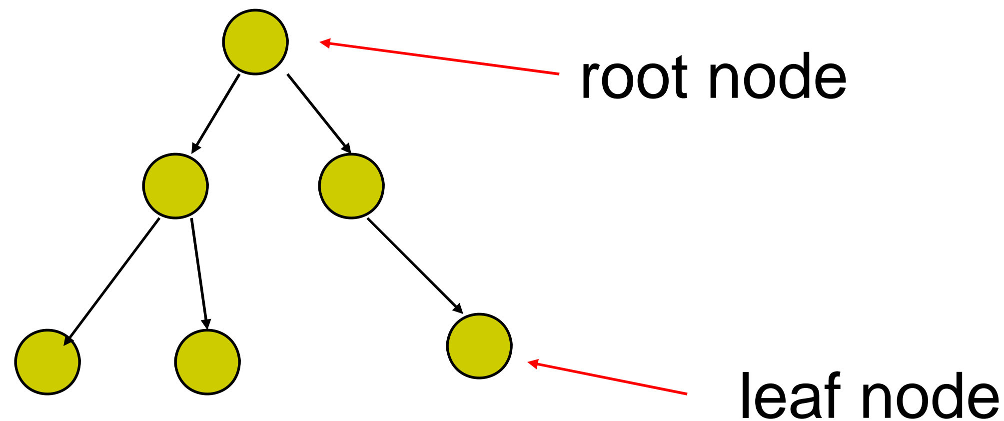
- Set of *nodes* and *edges (links)*
- Edge connects a pair of nodes
  - Directed or undirected
- *Cycle*: directed path that is a loop



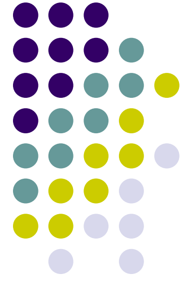
Tree

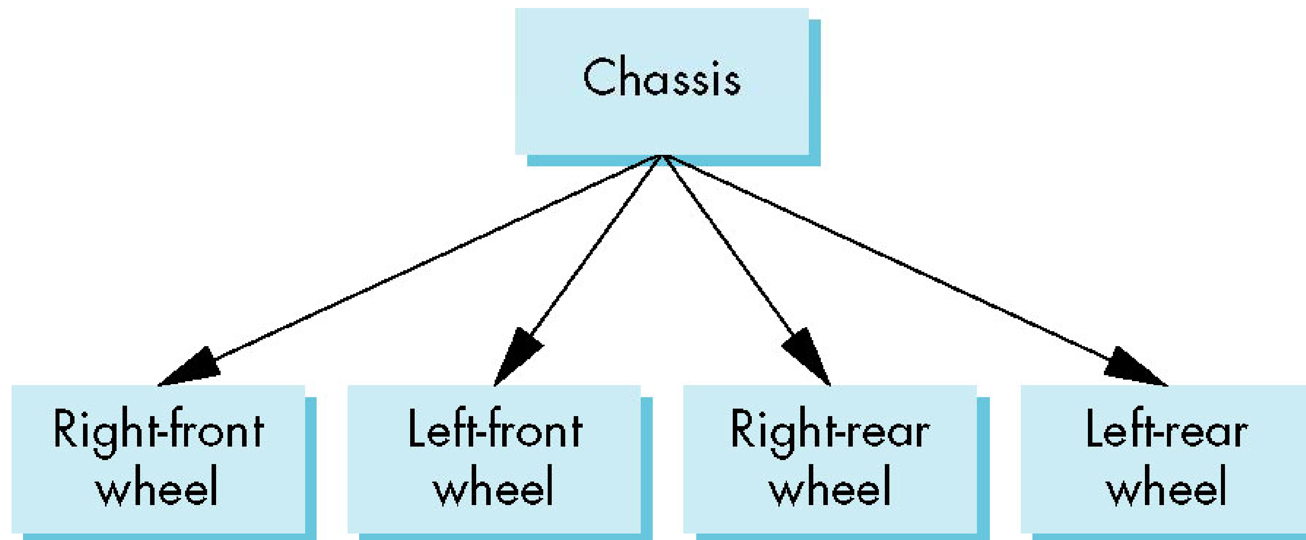


- Graph in which each node (except the root) has exactly one parent node
  - May have multiple children
  - Leaf or terminal node: no children

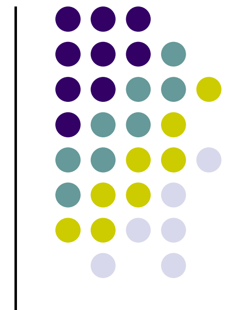


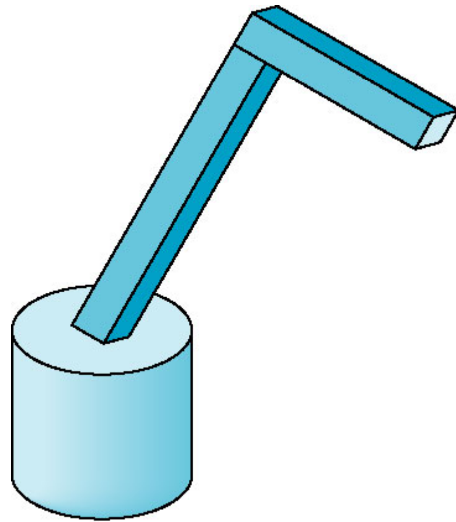
Tree Model of Car

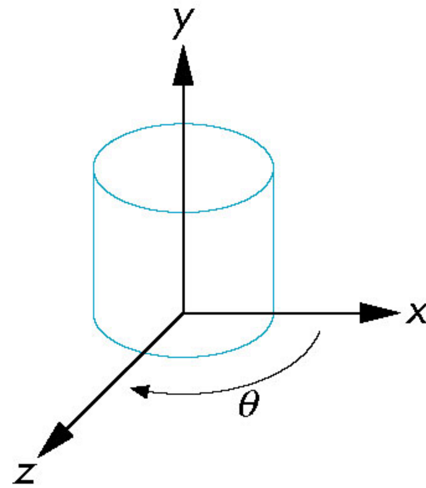


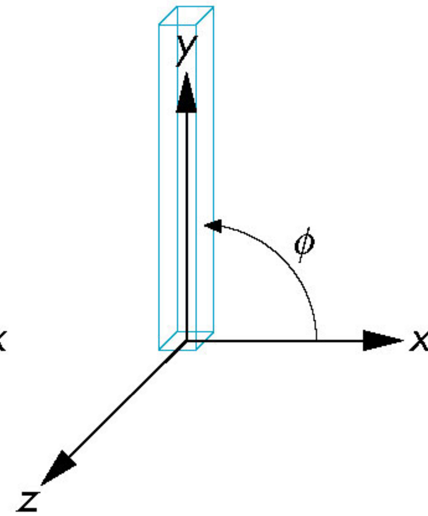


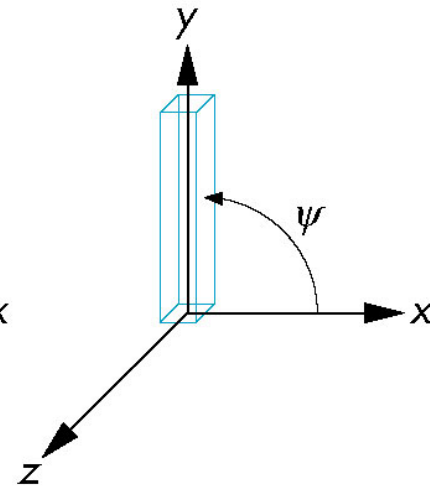
Robot Arm







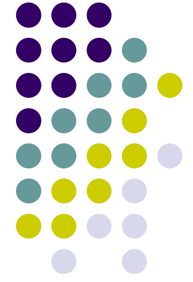




robot arm

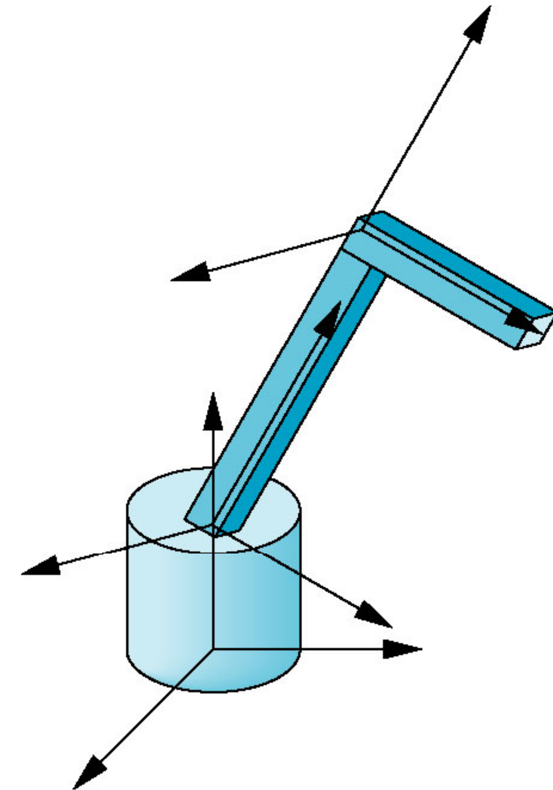
parts in their own coodinate systems

Articulated Models

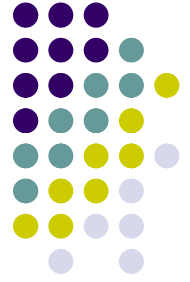


- Robot arm is example of *articulated model*
  - Parts connected at joints
  - Can specify state of model by

  giving all joint angles

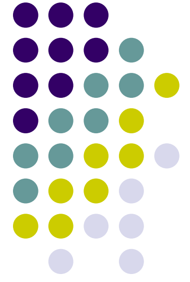


Required Matrices

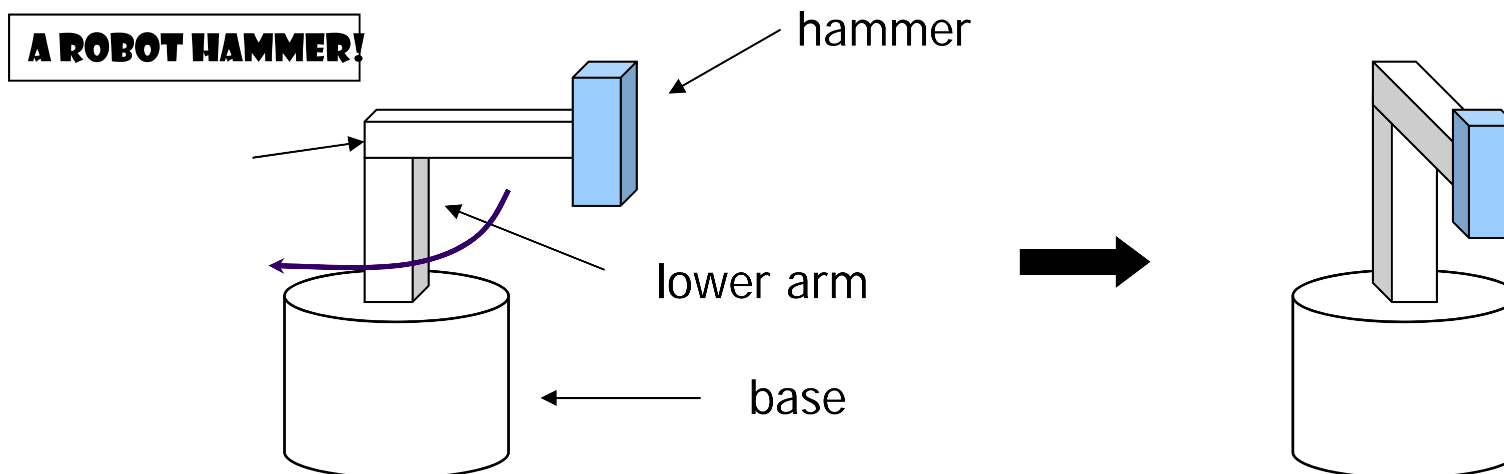


- Rotation of base: $\mathbf{R}_b$
  - Apply $\mathbf{M} = \mathbf{R}_b$ to base
- Translate lower arm relative to base: $\mathbf{T}_{lu}$
- Rotate lower arm around joint: $\mathbf{R}_{lu}$
  - Apply $\mathbf{M} = \mathbf{R}_b \mathbf{T}_{lu} \mathbf{R}_{lu}$ to lower arm
- Translate upper arm relative to upper arm: $\mathbf{T}_{uu}$
- Rotate upper arm around joint: $\mathbf{R}_{uu}$
  - Apply $\mathbf{M} = \mathbf{R}_b \mathbf{T}_{lu} \mathbf{R}_{lu} \mathbf{T}_{uu} \mathbf{R}_{uu}$ to upper arm

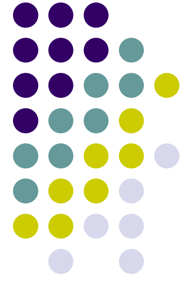
Hierarchical Transforms



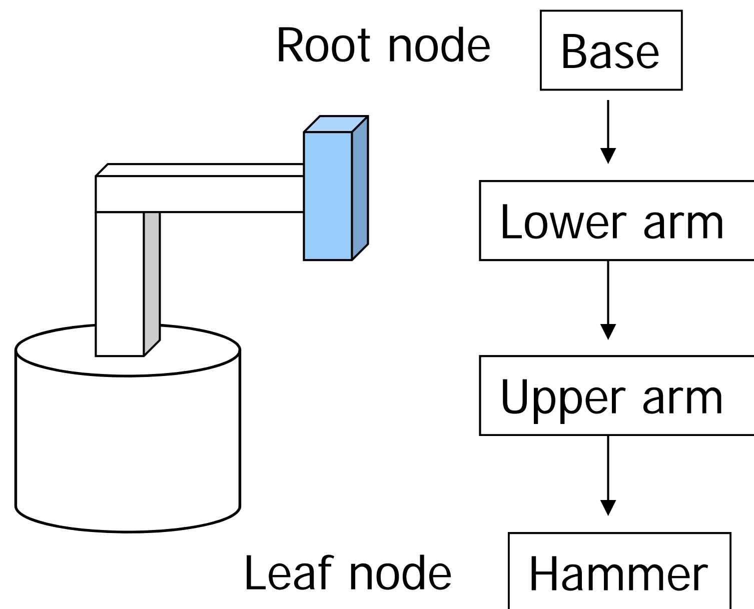
- Robot arm: Many small parts
- Attributes (position, orientation, etc) depend on each other



Hierarchical Transforms



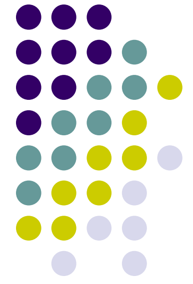
- Object dependency description using tree structure



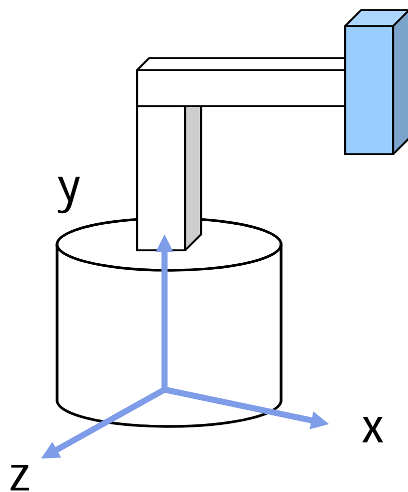
Object position and orientation can be affected by its parent, grand-parent, grand-grand-parent … nodes

Hierarchical representation is known as Scene Graph

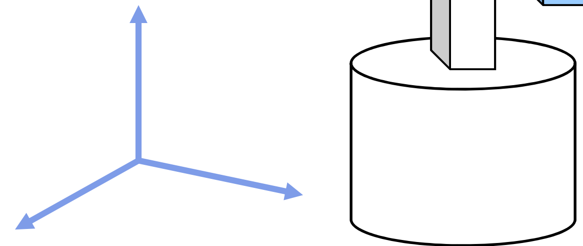
Transformations



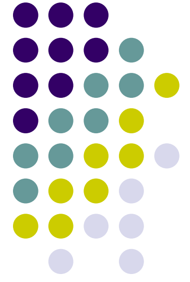
- Two ways to specify transformations:
  - (1) Absolute transformation: each part of the object is transformed independently relative to the origin



Translate the base by (5,0,0);
Translate the lower arm by (5,0,0);
Translate the upper arm by (5,0,0);
…

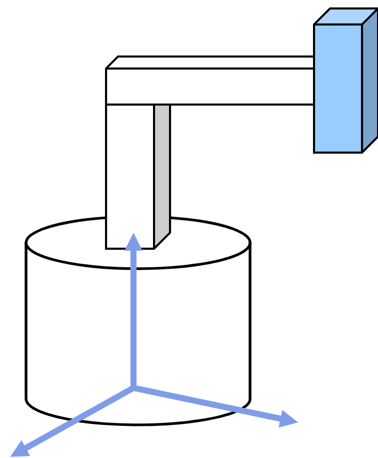


Relative Transformation

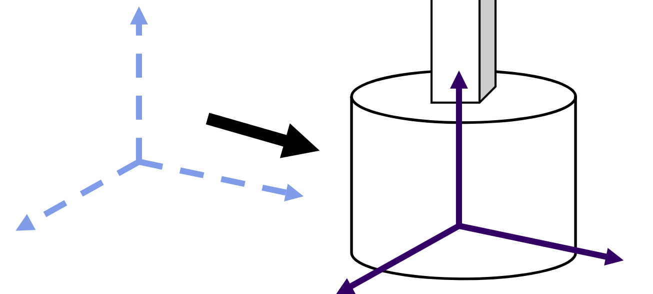


A better (and easier) way:

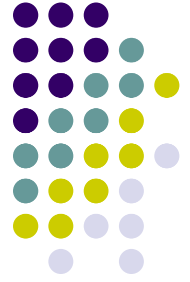
(2) Relative transformation: Specify the transformation for each object relative to its parent



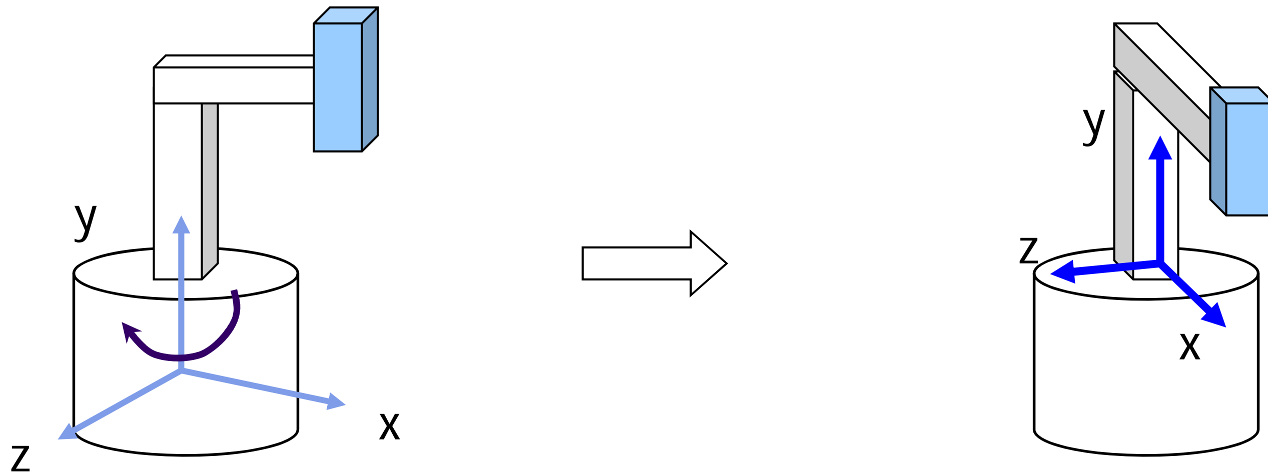
Step 1: Translate base and its descendants by (5,0,0);



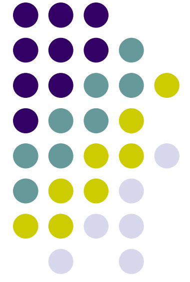
Relative Transformation



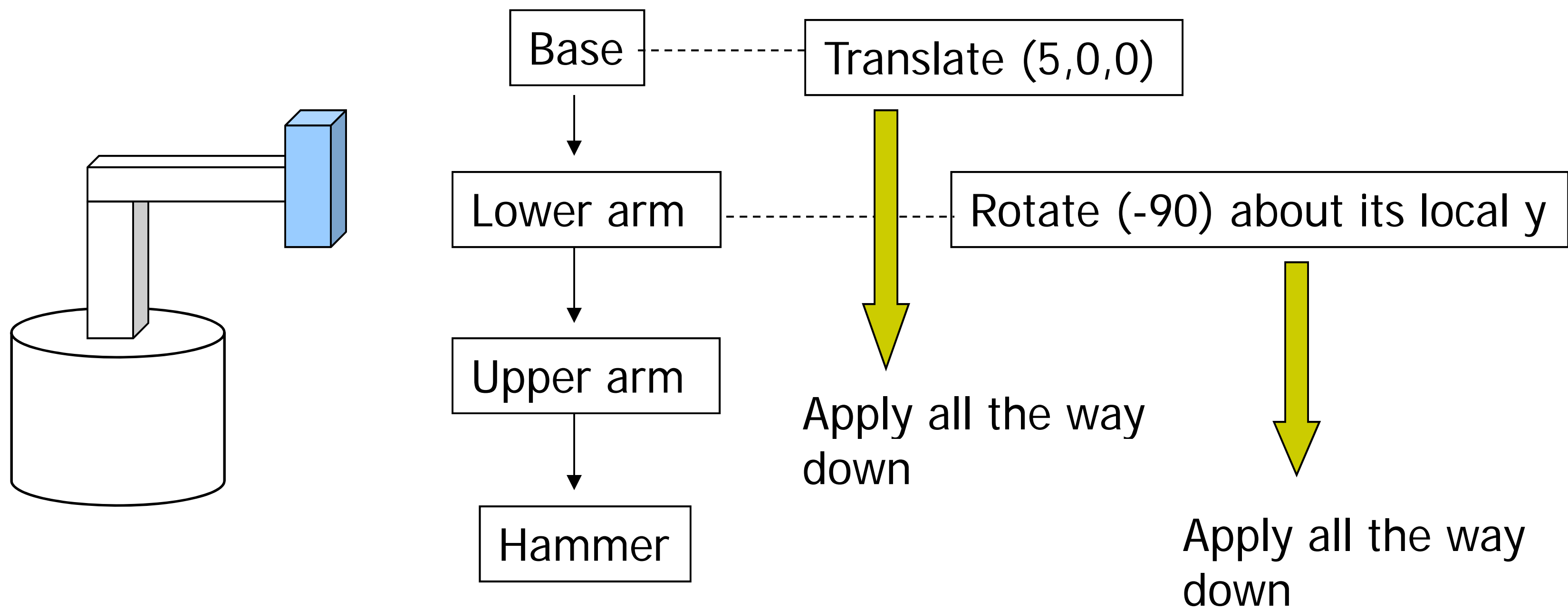
Step 2: Rotate the lower arm and all its descendants relative to the base's local y axis by -90 degree



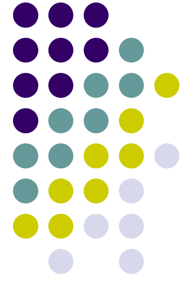
Relative Transformation



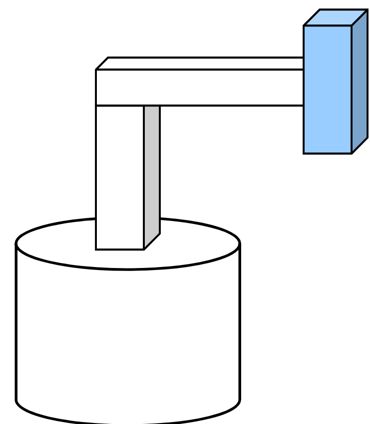
- Represent relative transformation using scene graph

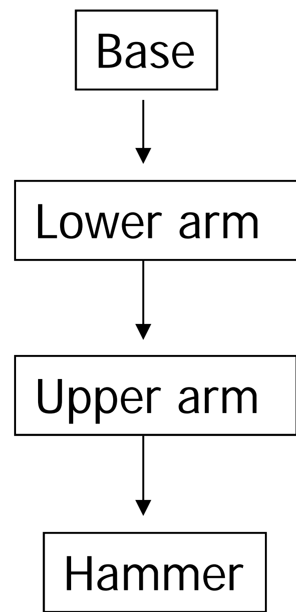


Hierarchical Transforms Using OpenGL



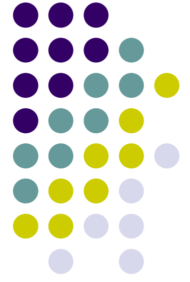
- Translate base and all its descendants by (5,0,0)
- Rotate lower arm and its descendants by -90 degree about local y





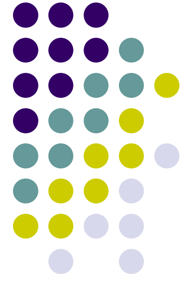

```
ctm = LoadIdentity();

… // setup your camera

ctm *= Translatef(5,0,0);

Draw_base();

ctm *= Rotatef(-90, 0, 1, 0);

Draw_lower _arm();
Draw_upper_arm();
Draw_hammer();
```

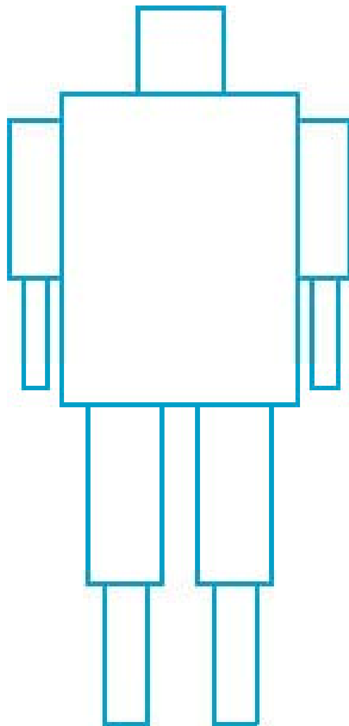
OpenGL Code for Robot

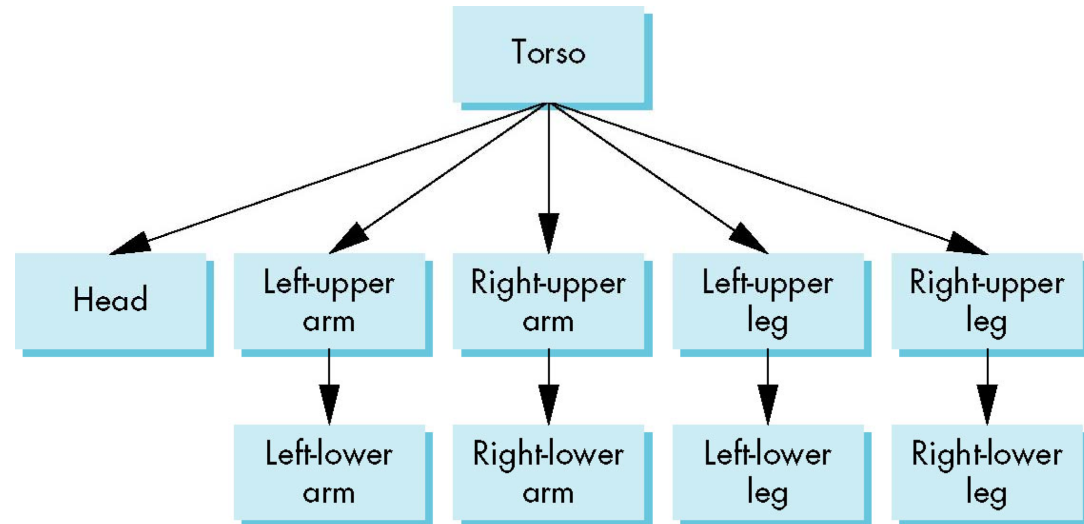


```
mat4 ctm;
robot_arm()
{
    ctm = RotateY(theta);
    base();
    ctm *= Translate(0.0, h1, 0.0);
    ctm *= RotateZ(phi);
    lower_arm();
    ctm *= Translate(0.0, h2, 0.0);
    ctm *= RotateZ(psi);
    upper_arm();
}
```

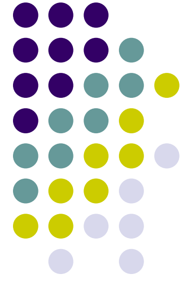
Humanoid Figure





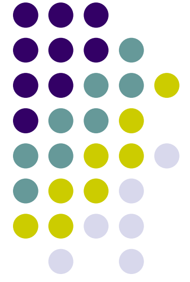


Building the Model



- Can build model using simple shapes
- Access parts through functions
  - `torso()`
  - `left_upper_arm()`
- Matrices describe position of node with respect to its parent
  - $\mathbf{M}_{\mathrm{lla}}$ positions left lower leg with respect to left upper arm

Tree with Matrices



Torso

$\mathbf{M}_h$ $\mathbf{M}_{lua}$ $\mathbf{M}_{rua}$ $\mathbf{M}_{lul}$ $\mathbf{M}_{rul}$

Head

Left-upper arm

Right-upper arm

Left-upper leg

Right-upper leg

$\mathbf{M}_{lla}$ $\mathbf{M}_{rla}$ $\mathbf{M}_{lll}$ $\mathbf{M}_{rll}$

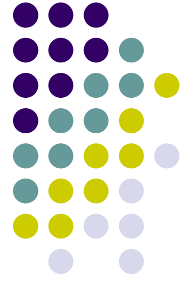
Left-lower arm

Right-lower arm

Left-lower leg

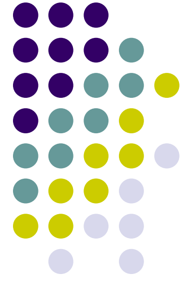
Right-lower leg

Transformation Matrices



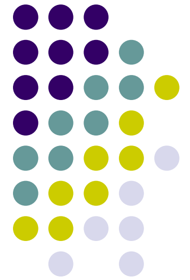
- There are 10 relevant matrices
  - $\mathbf{M}$ positions and orients entire figure through the torso which is the root node
  - $\mathbf{M}_h$ positions head with respect to torso
  - $\mathbf{M}_{lua}$, $\mathbf{M}_{rua}$, $\mathbf{M}_{lul}$, $\mathbf{M}_{rul}$ position arms and legs with respect to torso
  - $\mathbf{M}_{lla}$, $\mathbf{M}_{rla}$, $\mathbf{M}_{lll}$, $\mathbf{M}_{rll}$ position lower parts of limbs with respect to corresponding upper limbs

glPushMatrix and glPopMatrix



- Two important calls:
  - PushMatrix( ): Save current modelview matrix in stack
  - PopMatrix( ): restore transform matrix to what it was before PushMatrix( )

PopMatrix and PushMatrix Illustration in Stack



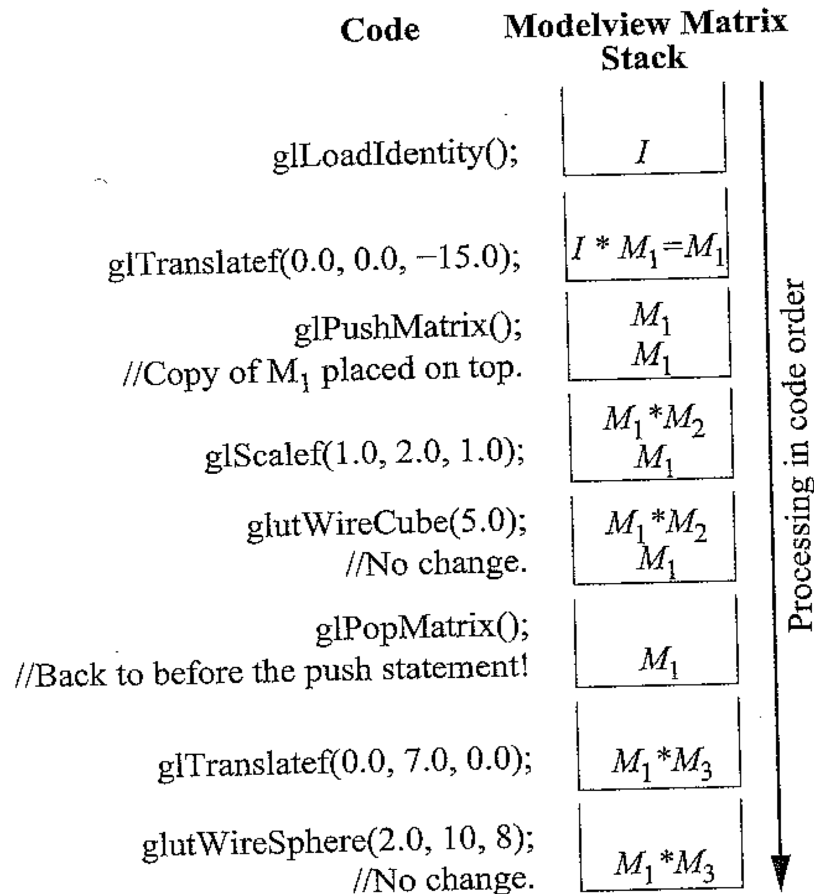
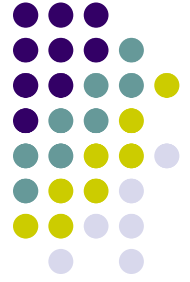


Figure 4.19: Transitions of the modelview matrix stack.

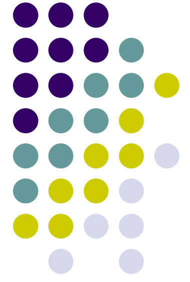
Ref: **Computer Graphics Through OpenGL by Guha**

Stack-based Traversal



- Set model-view matrix to $\mathbf{M}$ and draw torso
- Set model-view matrix to $\mathbf{MM}_{\mathrm{h}}$ and draw head
- For left-upper arm need $\mathbf{MM}_{\mathrm{lua}}$ and so on
- Rather than recomputing $\mathbf{MM}_{\mathrm{lua}}$ from scratch or using an inverse matrix, we can use the matrix stack to store $\mathbf{M}$ and other matrices as we traverse the tree

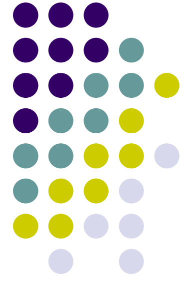
Traversal Code



```
figure() {
    PushMatrix()
    torso();
    Rotate (…);
    head();
    PopMatrix();
    PushMatrix();
    Translate(…);
    Rotate(…);
    left_upper_arm();
    PopMatrix();
    PushMatrix();
```

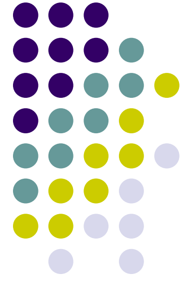
- save present model-view matrix
- update model-view matrix for head
- recover original model-view matrix
- save it again
- update model-view matrix for left upper arm
- recover and save original model-view matrix again
- rest of code

Scene Graph



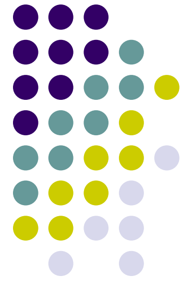
- Scene
  - Separator
    - Color
    - Translate
      - Rotate
    - Object 1
      - Translate
        - Object 2
  - Separator
    - Translate
      - Rotate
    - Object 3

Preorder Traversal



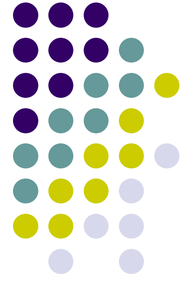
```
PushAttrib
PushMatrix
Color
Translate
Rotate
Object1
Translate
Object2
PopMatrix
PopAttrib
...
```

Inventor and Java3D

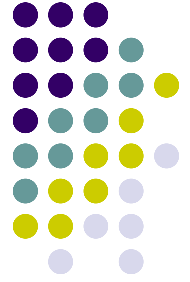


- Inventor and Java3D provide a scene graph API
- Scene graphs can also be described by a file (text or binary)
  - Implementation independent way of transporting scenes
  - Supported by scene graph APIs
- However, primitives supported should match capabilities of graphics systems
  - Hence most scene graph APIs are built on top of OpenGL or DirectX (for PCs)

VRML



- Want to have a scene graph that can be used over the World Wide Web
- Need links to other sites to support distributed data bases
- Virtual Reality Markup Language
  - Based on Inventor data base
  - Implemented with OpenGL



References

- Angel and Shreiner, Interactive Computer Graphics (6th edition), Chapter 8