UNIVERSITY of HOUSTON

DEPARTMENT OF COMPUTER SCIENCE

# COSC 4370 Fall 2023

## Interactive Computer Graphics

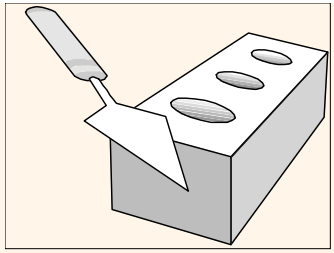
M & W 5:30 to 7:00 PM

Prof. Victoria Hilford

**PLEASE TURN your webcam ON**

**NO CHATTING during LECTURE**

prohibited.



# COSC 4370

## 5:30 to 7

## PLEASE

## LOG IN

## CANVAS

**Please close all other windows.**

# NEXT.

| | | |
|---|---|---|
| 10.25.2023 (W 5:30 to 7)<br>(19) | | **Lecture 10**<br>**(Modeling and Hierarchy)** |
| 10.30.2023 (M 5:30 to 7)<br>(20) | | **PROJECT 3** |
| 11.01.2023 (W 5:30 to 7)<br>(21) | | **EXAM 3 REVIEW** |
| 11.06.2023 (M 5:30 to 7)<br>(22) | | **EXAM 3** |

# COSC 4370 – Computer Graphics

## Lecture 10

## Modeling and Hierarchy
## Chapter 10

# Hierarchical Modeling

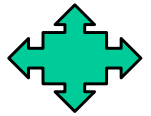
**Models are abstractions of the world both of the real world in which we live and of virtual worlds that we create with computers.**

In this chapter, we explore multiple approaches to developing and working with **Models of geometric objects.** We consider **Models** that use as building blocks a set of simple geometric objects: either the primitives supported by our graphics systems, or a set of user-defined objects built from these primitives. We extend the use of transformations from Chapter 4 to include hierarchical relationships among the objects.

**The techniques that we develop are appropriate for applications, such as robotics and figure animation, where the dynamic behavior of the objects is characterized by relationships among the parts of the model.** The notion of hierarchy is a powerful one and is an integral part of object-oriented methodologies.

**We extend our hierarchical models of objects to hierarchical Models of whole scenes, including cameras, lights, and material properties.** Such **Models** allow us to extend our graphics **API**s to more object-oriented systems and also give us insight into **using graphics over networks and distributed environments**, such as the **World WideWeb**.

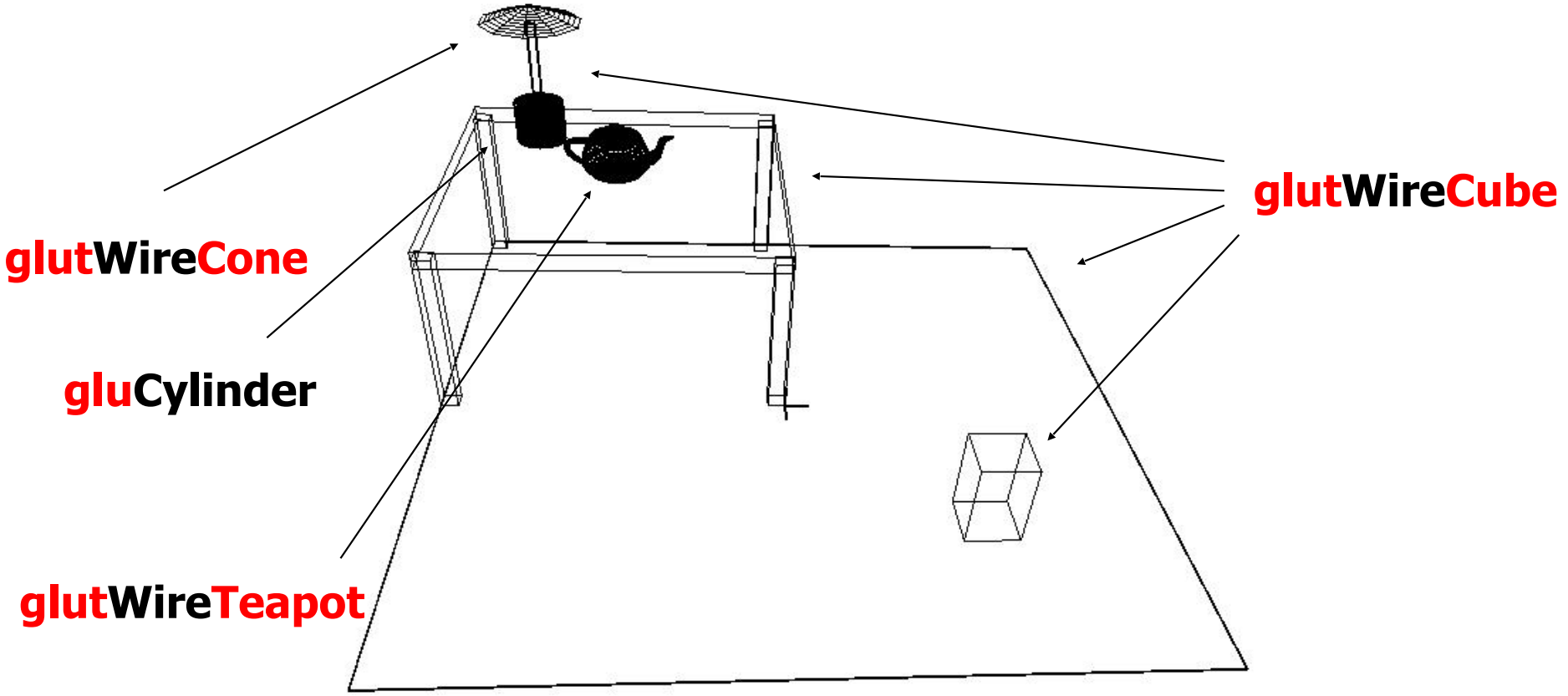
# Hierarchical Modeling I



# Using GLU/GLUT Objects

- From geometric primitives: point, line, polygon

GLU/GLUT provides very simple object primitives



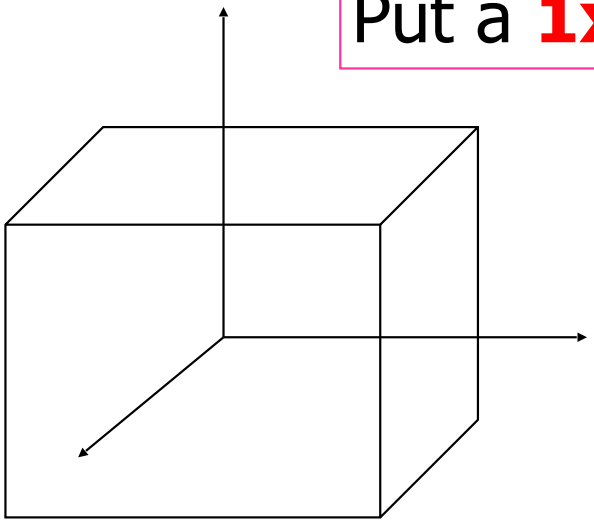
# glutWireCube(size)

Each glu/glut Object has its **default size, position, and orientation**

You need to perform **Modeling** transformation to make it right for you

**glutWireCube(1.0)** - 'wire' means wire frame

Put a **1x1x1 Cube** with its center at **World (0,0,0)**



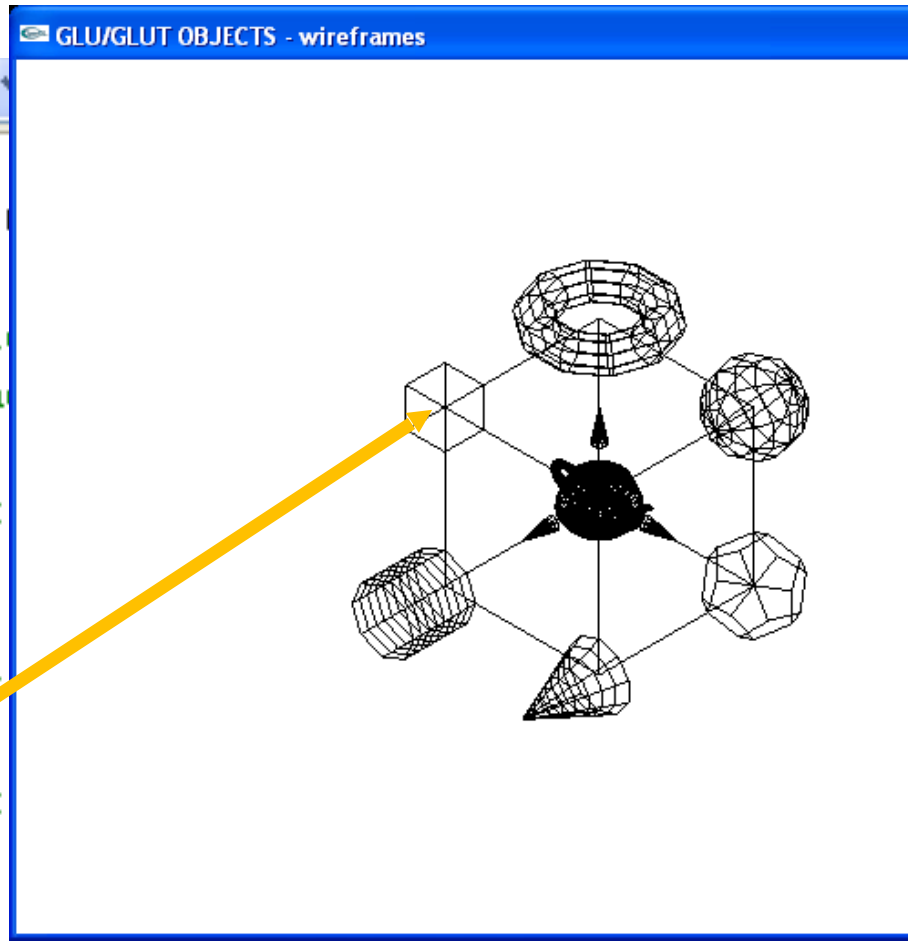
To create a **2 x 0.1 x 2** table top - need to call **glScalef(2, 0.1, 2)** before you call **glutWireCube(1.0)**

??????

# glutWireCube(size)

(Global Scope)

```
25      glLoadIdentity();
26      gluLookAt(2.0, 2.0, 2.0, 0.0, 0.0,
27
28      glClear(GL_COLOR_BUFFER_BIT); // cl
29      glColor3d(0,0,0); // draw black lin
30
31      axis(0.5);                    // z-ax
32      glPushMatrix();
33      glRotated(90, 0, 1.0, 0);
34      axis(0.5);                    // y-ax
35      glRotated(-90.0, 1, 0, 0);
36      axis(0.5);                    // z-ax
37      glPopMatrix();
38
39      glPushMatrix();
40      glTranslated(0.5, 0.5, 0.5); // big cube at (0.5, 0.5, 0.5)
41      glutWireCube(1.0);
42      glPopMatrix();
43
44      glPushMatrix();
45      glTranslated(1.0, 1.0, 0);    // sphere at (1, 1, 0)
```

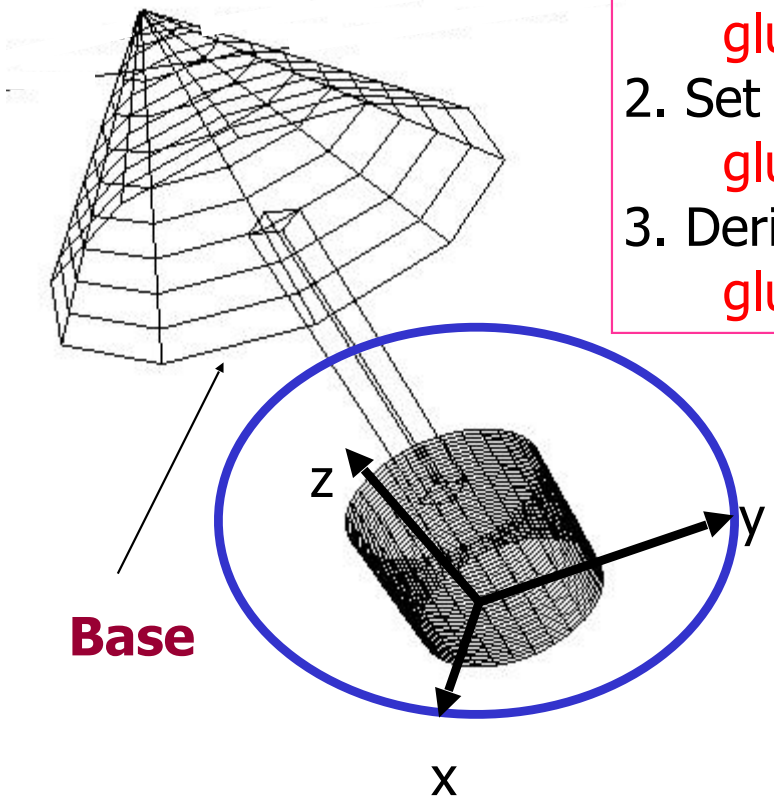


# gluCylinder()

- Three steps to create a **Cylinder**

**Sphere**, **Cylinder**, **Disk**, **Partial Disk**

1. Create a **GLU** quadric object
   glu**quadric**Obj *p = gluNewQuadric();
2. Set to wire frame mode
   glu**Quadric**DrawStyle(GLU__LINE);
3. Derive a **Cylinder** object from p
   glu**Cylinder**(p, base, top, height, slice, stacks)




**Base radius**

top **radius**

**height**

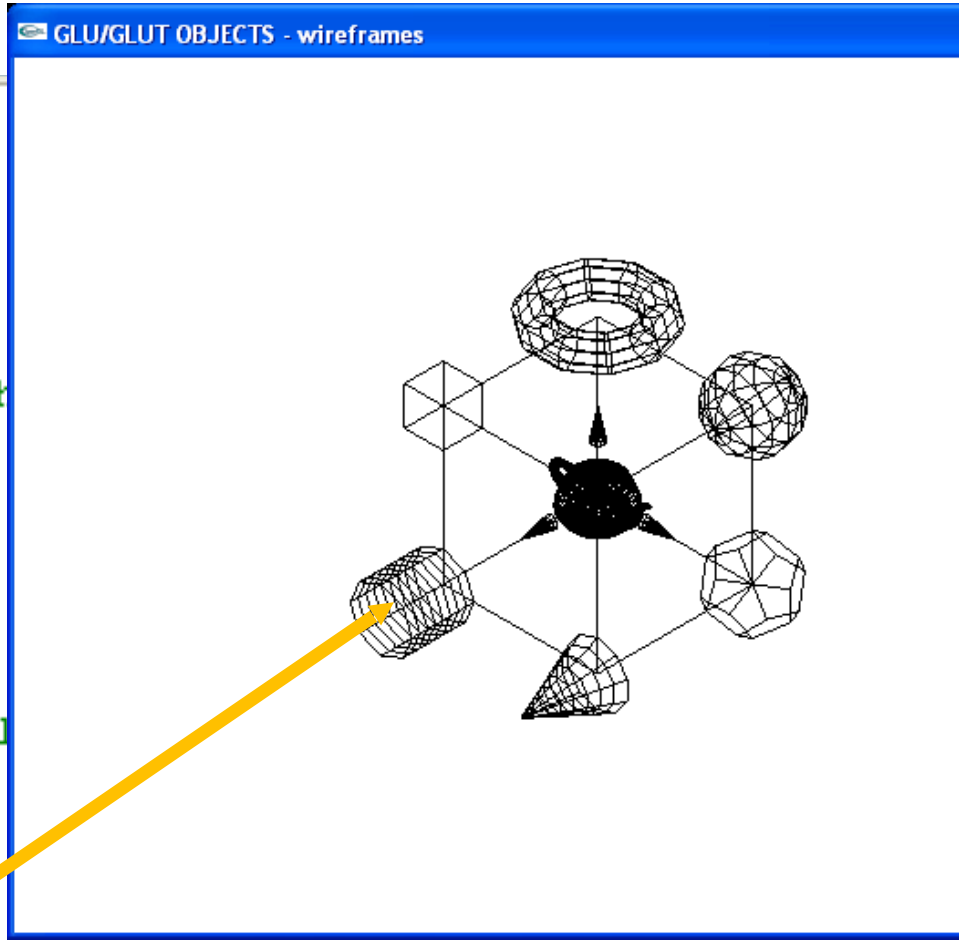
num. of vertical lines

num. of horizontal lines

**The default position is also with base at z = 0 plane**

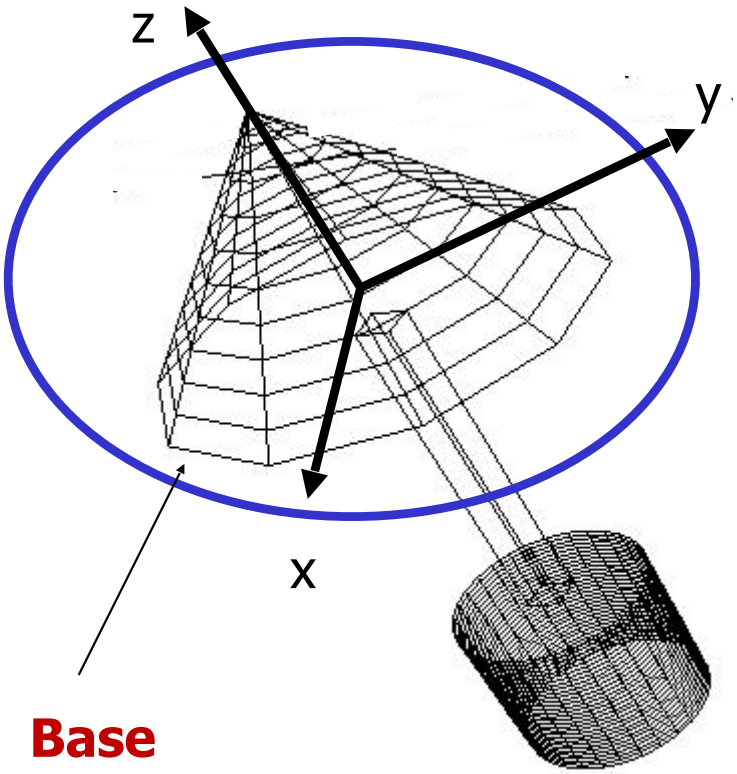
# gluCylinder()

```
61      glRotated(90.0, 1,0,0);
62      glutWireTorus(0.1, 0.3, 10,10);
63      glPopMatrix();
64
65      glPushMatrix();
66      glTranslated(1.0, 0 ,0); // dodecah
67      glScaled(0.15, 0.15, 0.15);
68      glutWireDodecahedron();
69      glPopMatrix();
70
71      glPushMatrix();
72      glTranslated(0, 1.0 ,1.0); // small
73      glutWireCube(0.25);
74      glPopMatrix();
75
76      glPushMatrix();
77      glTranslated(0, 0 ,1.0); // cylinder at (0,0,1)
78      GLUquadricObj * qobj;
79      qobj = gluNewQuadric();
80      gluQuadricDrawStyle(qobj,GLU_LINE);
81      gluCylinder(qobj, 0.2, 0.2, 0.4, 8,8);
82      glPopMatrix();
83      glFlush();
```



# glutWireCone()

Use glutWireCone and gluCylinder to make a lamp




glutWire**Cone**(**Base**, height, slices, stacks)

- A polygon approximation of a **Cone**.

**Default position: its base at Z = 0 plane**
**Base**: the width of its base
height: the height of the cone
slices: the number of vertical lines used to make up the cone
stace: the number of horizontal lines used to make up the cone

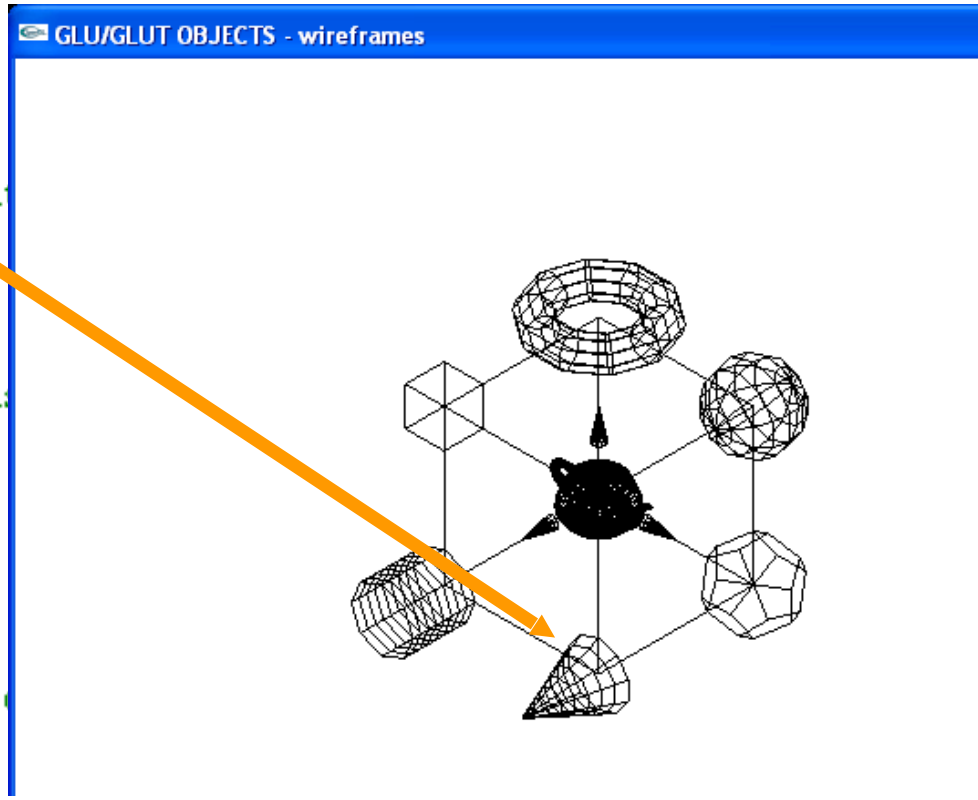
# glutWireCone()

Thread: Stack Frame:

GLUTObjects.cpp*

(Global Scope) displayWire()

```
46      glutWireSphere(0.25   10, 8);
47      glPopMatrix();
48
49      glPushMatrix();
50      glTranslated(1.0,0,1.0);     // cone at (1,0,1)
51      glutWireCone(0.2, 0.5, 10, 8);
52      glPopMatrix();
53
54      glPushMatrix();
55      glTranslated(1,1,1);
56      glutWireTeapot(0.2); // teapot a
57      glPopMatrix();
58
59      glPushMatrix();
60      glTranslated(0, 1.0 ,0); // torus
61      glRotated(90.0, 1,0,0);
62      glutWireTorus(0.1, 0.3, 10,10);
63      glPopMatrix();
64
65      glPushMatrix();
66      glTranslated(1.0, 0 ,0); // dode
67      glScaled(0.15, 0.15, 0.15);
```

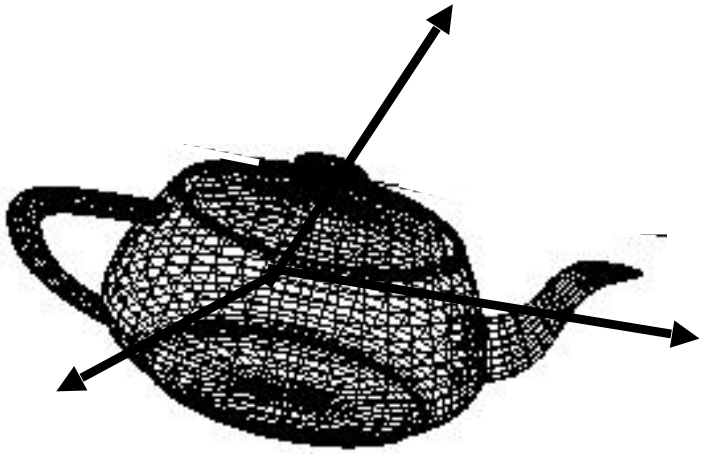


# glutWireTeapot()

The famous Utah Teapot has become an unofficial computer graphics mascot

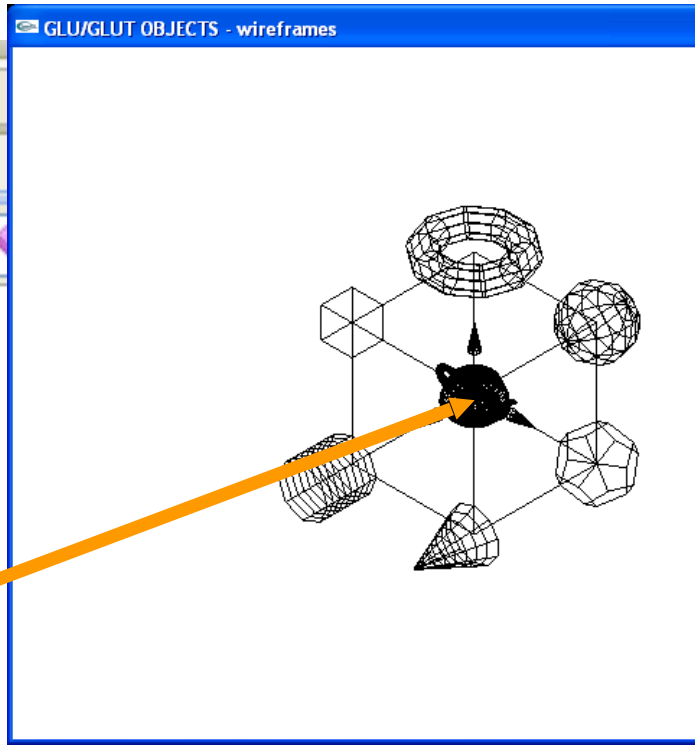
glutWireTeapot(0.5)

Create a **teapot** with size 0.5, and position its center at **(0,0,0)**



**Again, you need to apply transformations to position it at the right spot**

# glutWireTeapot()




GLUTObjects.cpp*

(Global Scope)

```
46      glutWireSphere(0.25, 10, 8);
47      glPopMatrix();
48
49      glPushMatrix();
50      glTranslated(1.0,0,1.0);     // cone at
51      glutWireCone(0.2, 0.5, 10, 8);
52      glPopMatrix();
53
54      glPushMatrix();
55      glTranslated(1,1,1);
56      glutWireTeapot(0.2); // teapot at (1,1,1)
57      glPopMatrix();
58
59      glPushMatrix();
60      glTranslated(0, 1.0 ,0); // torus at (0,1,0)
61      glRotated(90.0, 1,0,0);
62      glutWireTorus(0.1, 0.3, 10,10);
63      glPopMatrix();
64
65      glPushMatrix();
66      glTranslated(1.0, 0 ,0); // dodecahedron at (1,0,0)
67      glScaled(0.15, 0.15, 0.15);
```

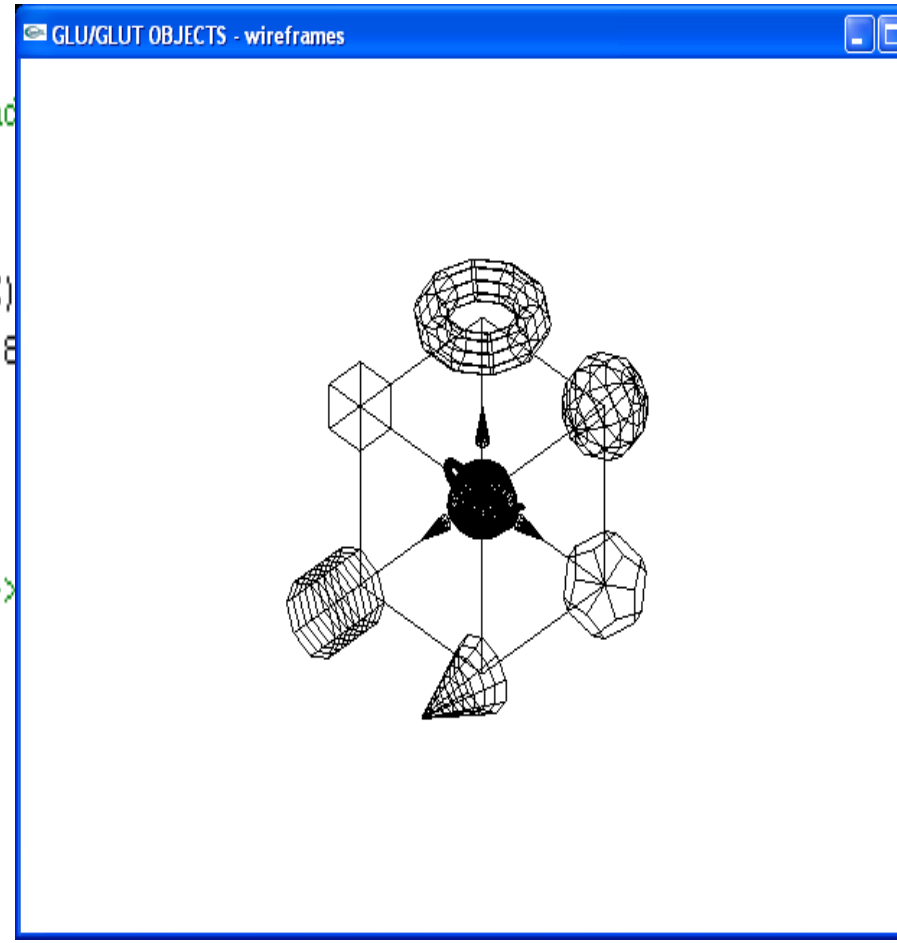

```
59      glPushMatrix();
60      glTranslated(0, 1.0 ,0); // torus at (0,1,0)
61      glRotated(90.0, 1,0,0);
62      glutWireTorus(0.1, 0.3, 10,10);
63      glPopMatrix();
64
65      glPushMatrix();
66      glTranslated(1.0, 0 ,0); // dodecahedron at (1,0,0)
67      glScaled(0.15, 0.15, 0.15);
68      glutWireDodecahedron();
69      glPopMatrix();
70
71      glPushMatrix();
72      glTranslated(0, 1.0 ,1.0); // small cube at (0,1,1)
73      glutWireCube(0.25);
74      glPopMatrix();
75
76      glPushMatrix();
77      glTranslated(0, 0 ,1.0); // cylinder at (0,0,1)
78      GLUquadricObj * qobj;
79      qobj = gluNewQuadric();
80      gluQuadricDrawStyle(qobj,GLU_LINE);
81      gluCylinder(qobj, 0.2, 0.2, 0.4, 8,8);
82      glPopMatrix();
83      glFlush();
84  }
```

Solution 'Lecture 10' (1 project)

Lecture 10

- Header Files
- Resource Files
- Source Files
  - GLUTObjects.cpp

```
69      glPushMatrix();
70      glTranslated(0, 1.0 ,1.0); // small cube at (0,1,1)
71      glutWireCube(0.25);
72      glPopMatrix();
73
74      glPushMatrix();
75      glTranslated(0, 0 ,1.0); // cylind
76      GLUquadricObj * qobj;
77      qobj = gluNewQuadric();
78      gluQuadricDrawStyle(qobj,GLU_LINE)
79      gluCylinder(qobj, 0.2, 0.2, 0.4, 8
80      glPopMatrix();
81      glFlush();
82  }
83  //<<<<<<<<<<<<<<<<<<<<<< main >>>>>>>>
84  void main(int argc, char **argv)
85  {
86      glutInit(&argc, argv);
87      glutInitDisplayMode(GLUT_SINGLE |
88      glutInitWindowSize(640,480);
89      glutInitWindowPosition(100, 100);
90      glutCreateWindow("GLU/GLUT OBJECTS - wireframes");
91      glutDisplayFunc(displayWire);
92      glClearColor(1.0f, 1.0f, 1.0f,0.0f);  // background is white
93      glViewport(0, 0, 640, 480);
94      glutMainLoop();
```

# Objectives

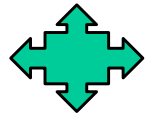
- **GLU/GLUT** objects

- Examine the limitations of **Linear** **Modeling**
  **Symbols** and **Instances**

- Introduce hierarchical models
  Articulated models
  Robots

- Introduce Tree and DAG models

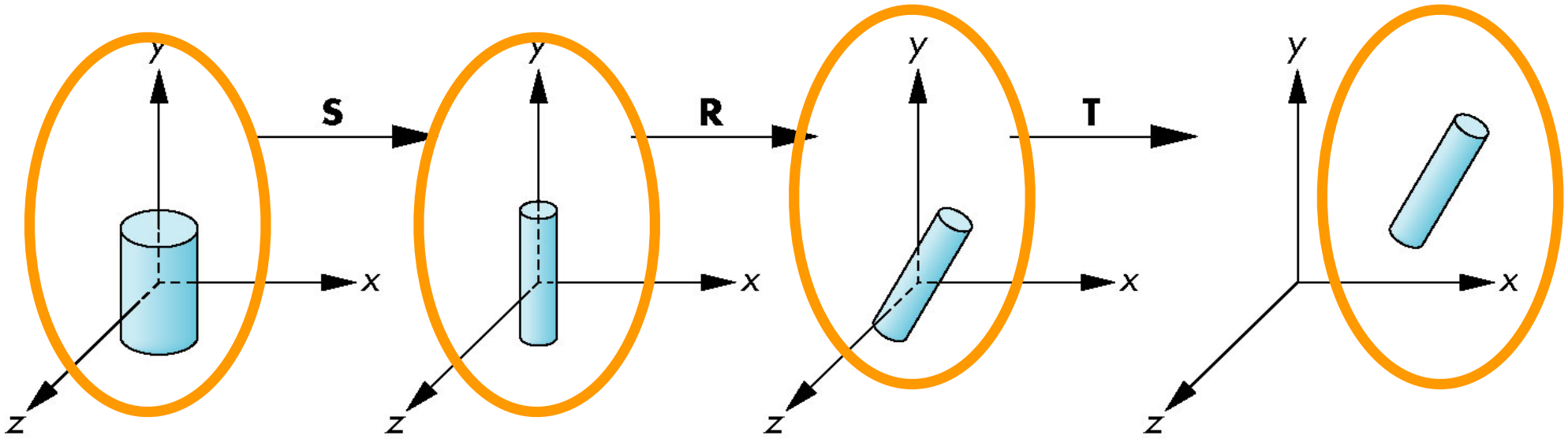
# Instance Transformation
(nonhierarchical model)

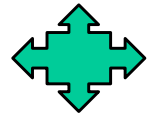


- Start with a prototype **Object** (a *symbol*)
- Each appearance of the **Object** in the **Model** is an ***instance***

  **Must scale, orient, position**

  Defines **instance transformation**



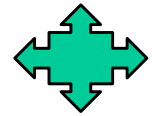


# Symbol-instance Table

Can **store a Model by assigning a number to each symbol** and storing the **parameters** for the **instance transformation**

| Symbol | Scale | Rotate | Translate |
|---|---|---|---|
| 1<br>2<br>3<br>1<br>1<br>.<br>. | $s_{x'}, s_{y'}, s_z$ | $\theta_{x'}, \theta_{y'}, \theta_z$ | $d_{x'}, d_{y'}, d_z$ |

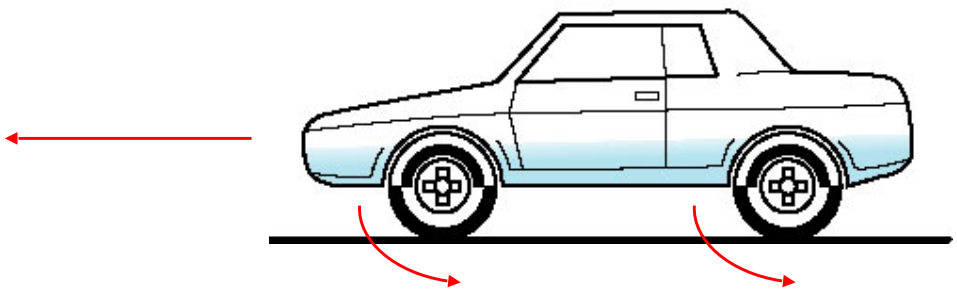
# Relationships in Car Model (nonhierarchical model)



Symbol-instance Table does not show relationships between parts of Model

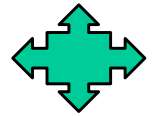
Consider Model of car

- **Chassis + 4 identical wheels**
- **Two symbols**



| 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 | | | |
| . | | | |
| . | | | |

# Structure Through Function Calls (speed & direction)



```
{
    float s; /* speed */
    float d[3] /* direction */
    float t /* time */

    /* determine speed and direction at time t*/

    draw_right_front_wheel(s,d);
    draw_left_front_wheel(s,d);
    draw_right_rear_wheel(s,d);
    draw_left_rear_wheel(s,d);
    draw_chassis(s,d);
}
```

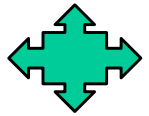
- Fails to show relationships well

- **Look at problem using a graph**

# Objectives

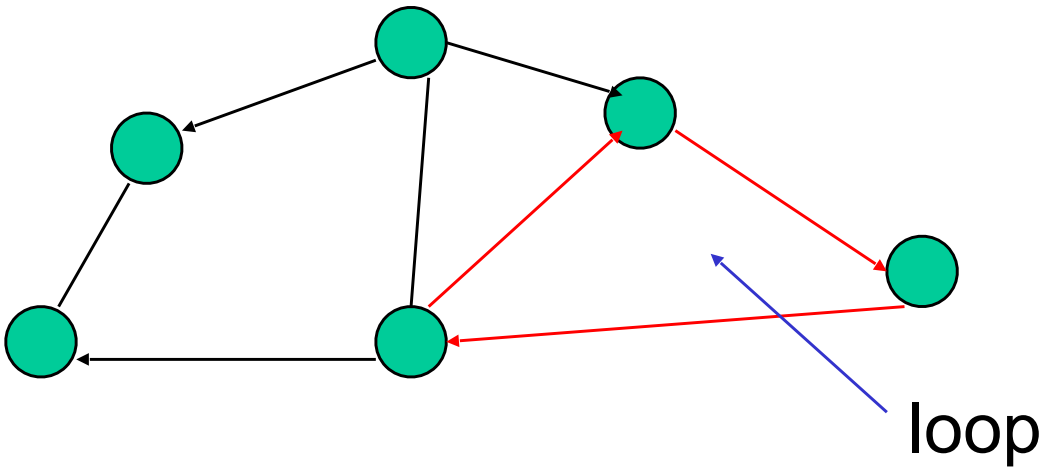
- **GLU/GLUT** objects
- Examine the limitations of linear modeling
  - Symbols and instances
- Introduce hierarchical models
  - Articulated models
  - Robots
- Introduce Tree and DAG **Models**

# Graphs
## (to represent relationships)



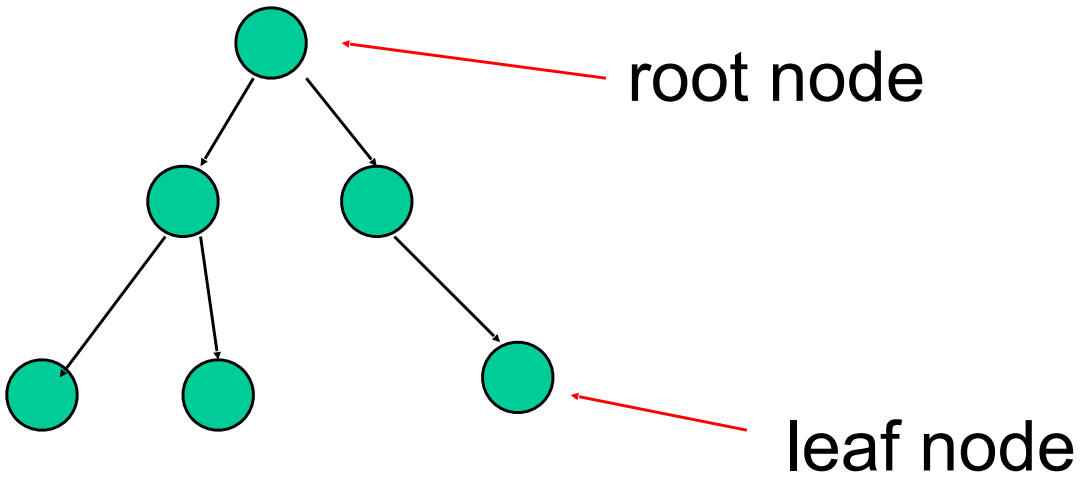
- Set of *nodes* and *edges (links)*
- Edge connects a pair of nodes

  Directed or undirected
- ***Cycle***: directed path that is a loop

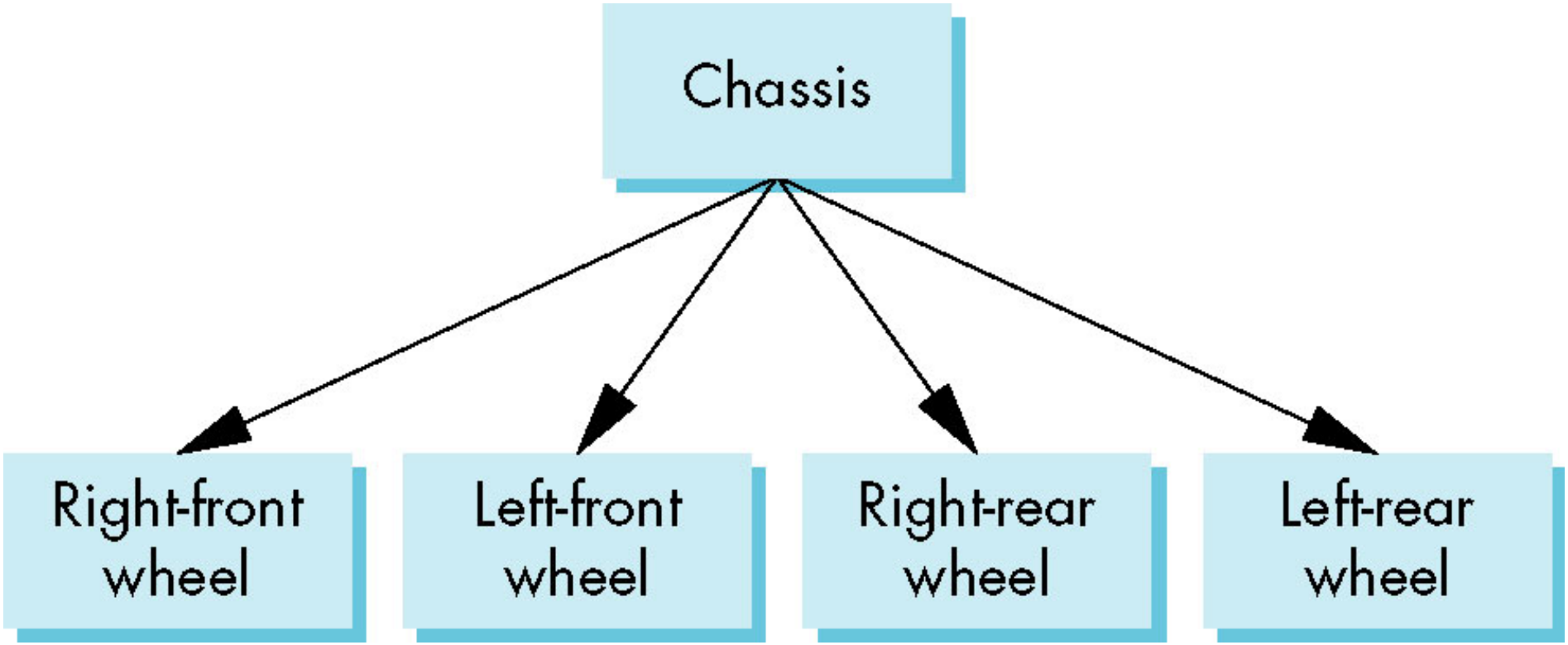


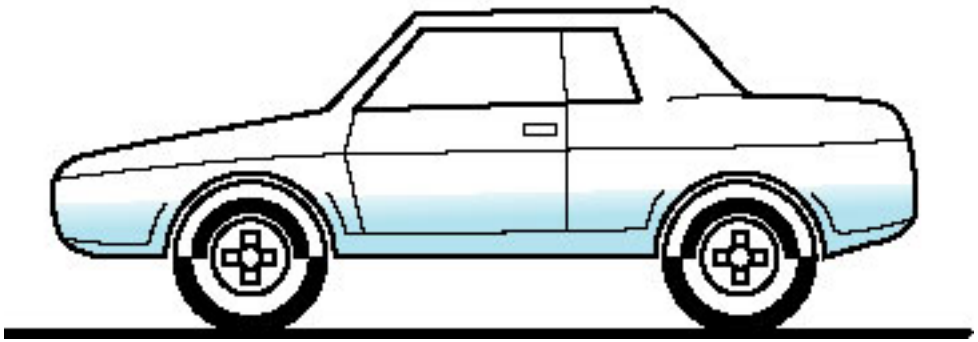
# Tree

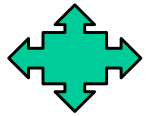
- **Graph** in which each **node** (except the root) has **exactly one parent node**
  - May have multiple children
  - Leaf or terminal **node**: no children



# Tree Model of Car
(hierarchical model)



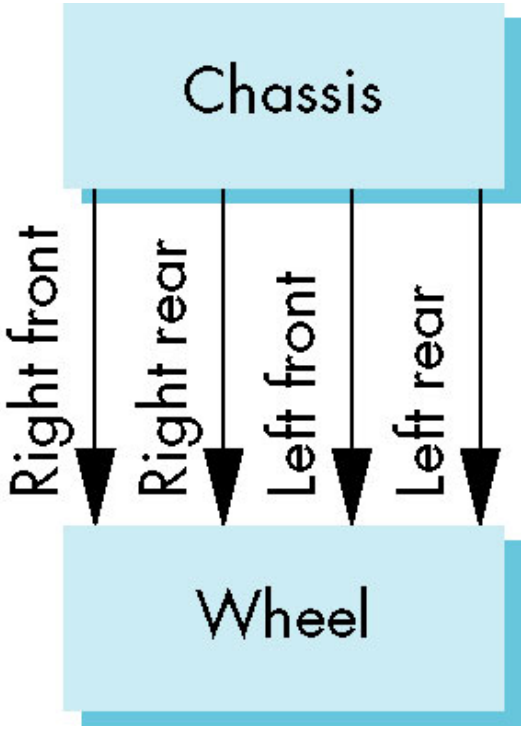





# DAG Model of Car

- If we use the fact that all the **wheels** are identical, we get a *Directed Acyclic Graph*

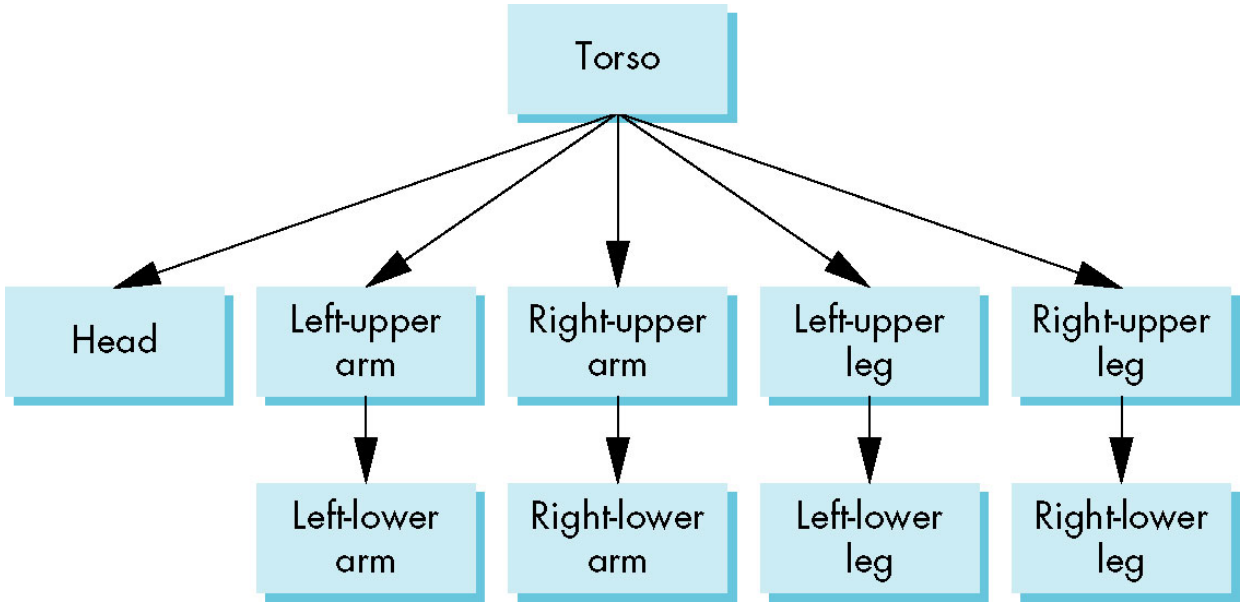
**Not much different than dealing with a tree**



# Modeling with Trees
(hierarchical model)

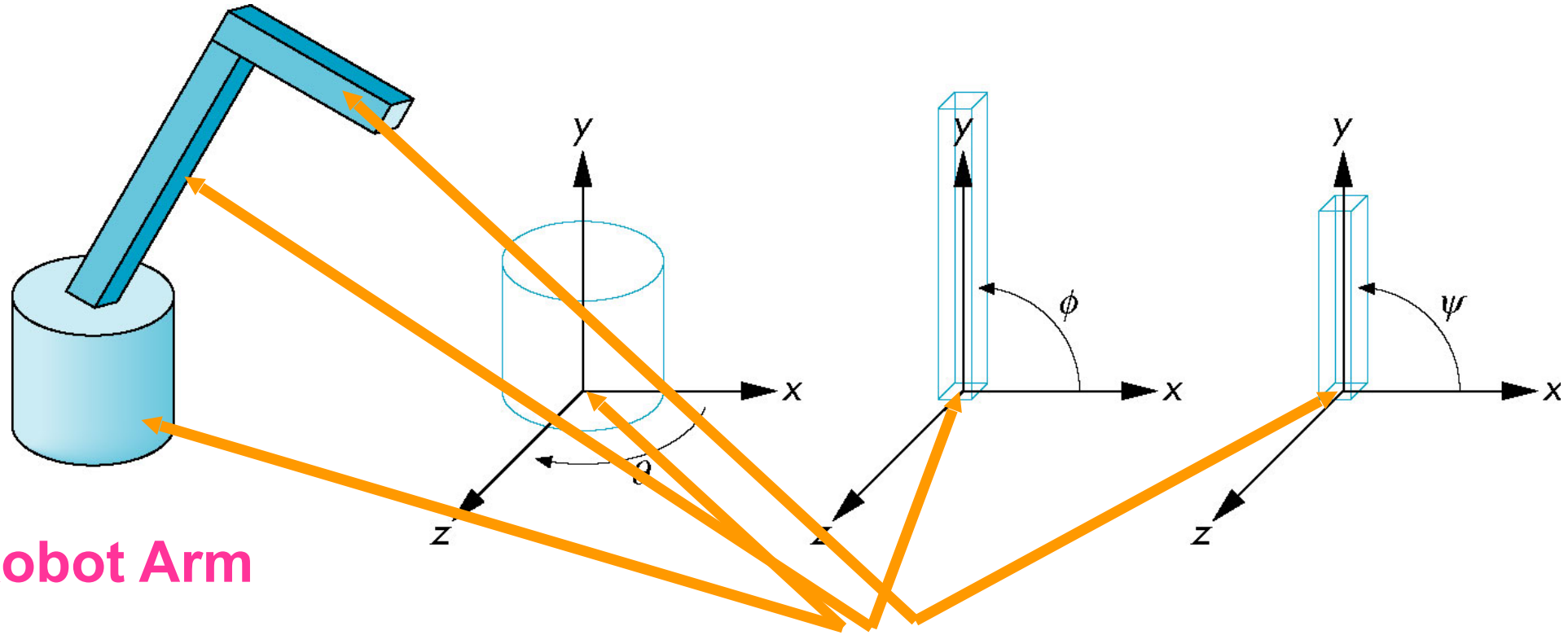
- Must decide what information to place in **Nodes** and what to put in **Edges**

- **Nodes**
  - **What to draw**
  - Pointers to Children




- **Edges**
  - May have information on **incremental changes to transformation matrices** (can also be store in **nodes**)

# Robot Arm
(hierarchical model)




Robot Arm

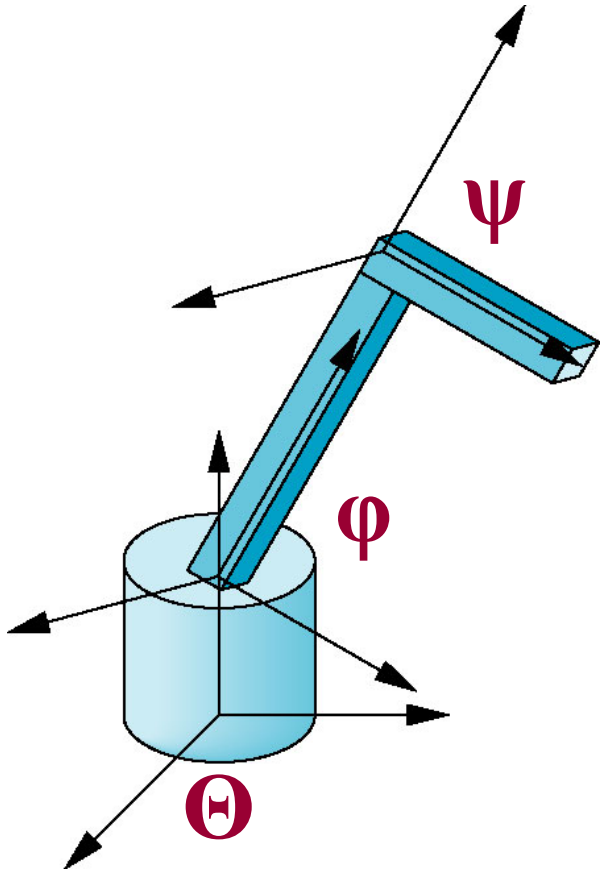
parts in their own coordinate systems

# Articulated Models

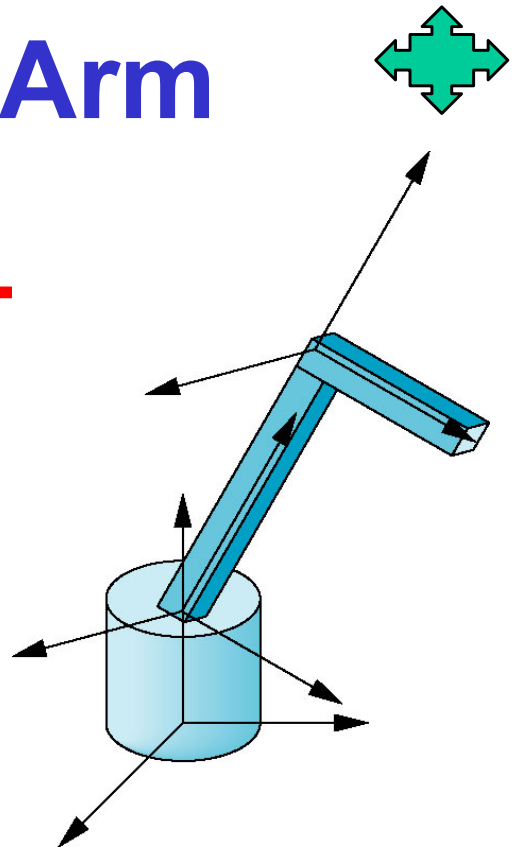
- **Robot Arm** is an example of an *Articulated* ***Model***

  **Parts** connected at joints

  Can specify **state of Model** by giving all joint **angles**




# Relationships in Robot Arm

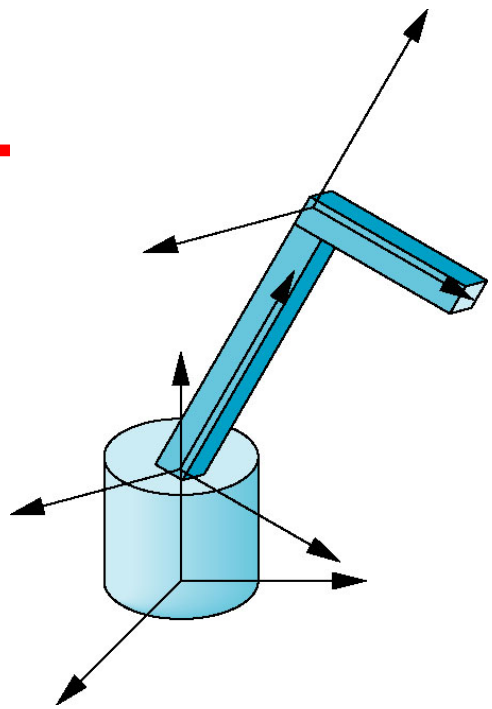



- **Base rotates independently**
  - **Single angle determines position**
- **Lower Arm attached to Base**
  - Its **position** depends on **rotation** of **Base**
  - **Must also translate relative to Base** and **rotate about connecting joint**
- **Upper Arm attached to Lower Arm**

ψ

  - Its **position** depends on both **Base** and **Lower Arm**
  - **Must translate relative to Lower Arm** and **rotate about joint connecting to Lower Arm**

# Required Matrices



- Rotation of **Base**: $\mathbf{R}_b$ $\Theta$

  Apply $\mathbf{M} = \mathbf{R}_b$ to **Base**

- Translate **Lower Arm** relative to **Base**: $\mathbf{T}_{lu}$

- Rotate **Lower Arm** around joint: $\mathbf{R}_{lu}$

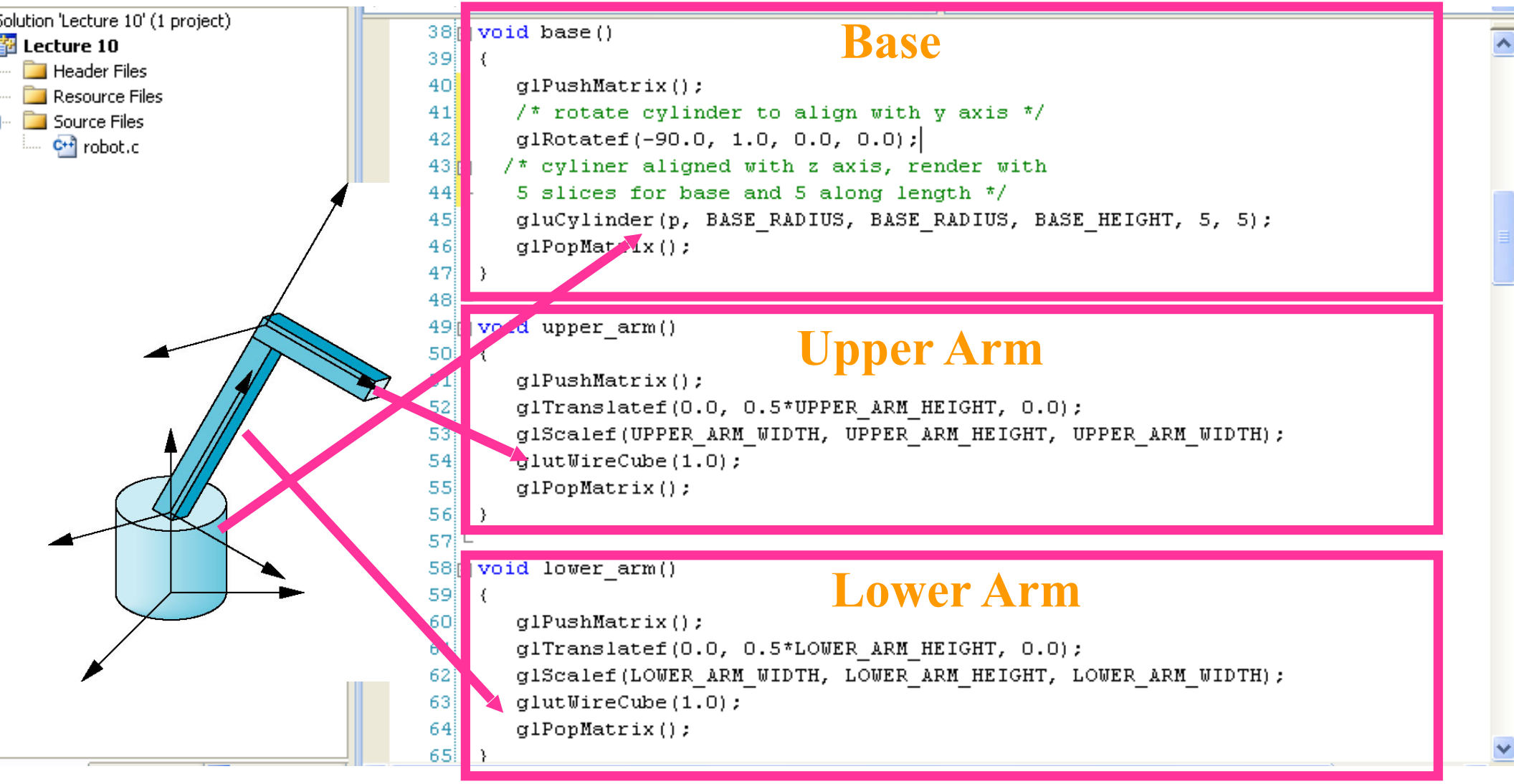
  Apply $\mathbf{M} = \mathbf{R}_b \mathbf{T}_{lu} \mathbf{R}_{lu}$ to lower arm

- Translate **Upper Arm** relative to **Lower Arm**: $\mathbf{T}_{uu}$ $\Psi$

- Rotate **Upper Arm** around joint: $\mathbf{R}_{uu}$

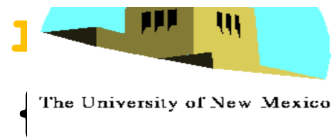
  Apply $\mathbf{M} = \mathbf{R}_b \mathbf{T}_{lu} \mathbf{R}_{lu} \mathbf{T}_{uu} \mathbf{R}_{uu}$ to **Upper Arm**

# Robot

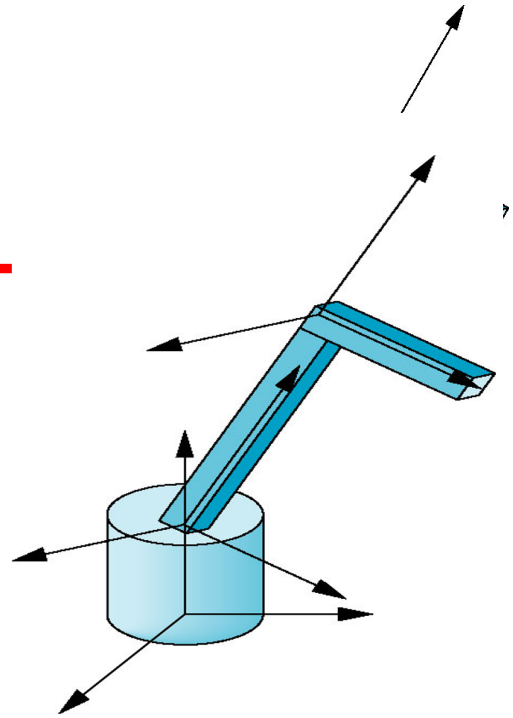



```
38  void base()
39  {
40      glPushMatrix();
41      /* rotate cylinder to align with y axis */
42      glRotatef(-90.0, 1.0, 0.0, 0.0);
43    /* cyliner aligned with z axis, render with
44      5 slices for base and 5 along length */
45      gluCylinder(p, BASE_RADIUS, BASE_RADIUS, BASE_HEIGHT, 5, 5);
46      glPopMatrix();
47  }
48
49  void upper_arm()
50  {
51      glPushMatrix();
52      glTranslatef(0.0, 0.5*UPPER_ARM_HEIGHT, 0.0);
53      glScalef(UPPER_ARM_WIDTH, UPPER_ARM_HEIGHT, UPPER_ARM_WIDTH);
54      glutWireCube(1.0);
55      glPopMatrix();
56  }
57
58  void lower_arm()
59  {
60      glPushMatrix();
61      glTranslatef(0.0, 0.5*LOWER_ARM_HEIGHT, 0.0);
62      glScalef(LOWER_ARM_WIDTH, LOWER_ARM_HEIGHT, LOWER_ARM_WIDTH);
63      glutWireCube(1.0);
64      glPopMatrix();
65  }
```

# OpenGL Code for Robot Arm




## Required Matrices

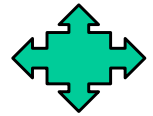


- Rotation of **Base**: $\mathbf{R}_b$ $\Theta$
  - Apply $\mathbf{M} = \mathbf{R}_b$ to **Base**
- Translate **Lower Arm** relative to **Base**: $\mathbf{T}_{lu}$
- Rotate **Lower Arm** around joint: $\mathbf{R}_{lu}$
  - Apply $\mathbf{M} = \mathbf{R}_b \mathbf{T}_{lu} \mathbf{R}_{lu}$ to lower arm
- Translate **Upper Arm** relative to **Lower Arm**: $\mathbf{T}_{uu}$ $\Psi$
- 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**

# Robot

```
1 project)
```

```
66
67  void display(void)
68  {
69
70     /* Accumulate ModelView Matrix as we traverse tree */
71      glClear(GL_COLOR_BUFFER_BIT);
72      glLoadIdentity();
73      glColor3f(1.0, 0.0, 0.0);
74
75      glRotatef(theta[0], 0.0, 1.0, 0.0);
76      base();
77
78      glTranslatef(0.0, BASE_HEIGHT, 0.0);
79      glRotatef(theta[1], 0.0, 0.0, 1.0);
80      lower_arm();
81
82      glTranslatef(0.0, LOWER_ARM_HEIGHT, 0.0);
83      glRotatef(theta[2], 0.0, 0.0, 1.0);
84      upper_arm();
85
86      glFlush();
87      glutSwapBuffers();
88  }
89
```



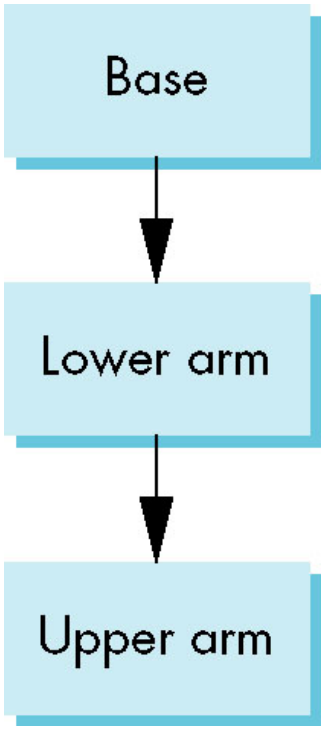
# Tree Model of Robot

Note code shows **relationships between Parts of Model**

Can change "look" of **Parts** easily without altering relationships

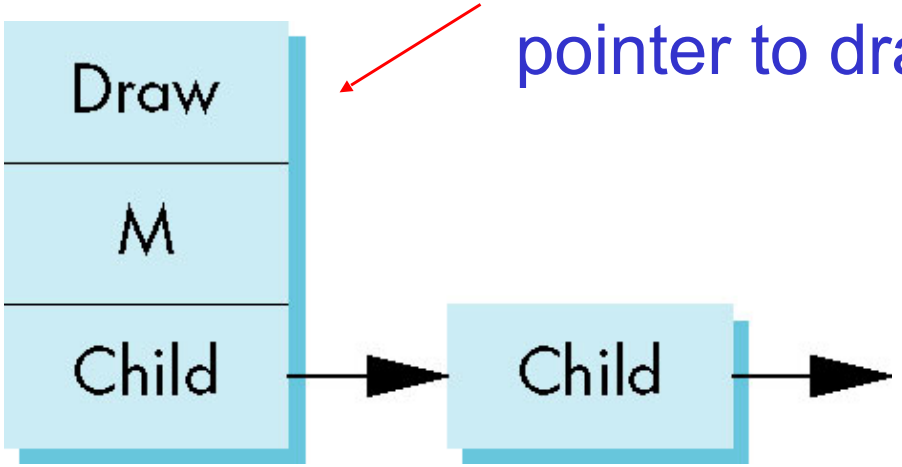
Simple example of Tree **Model**

Want ***general node structure*** for **nodes**



# Possible Node Structure

Code for drawing **Part** **or** pointer to drawing function

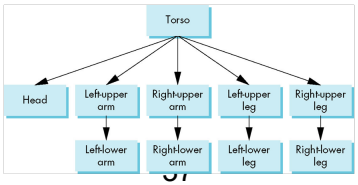



Linked **List** of pointers to **Children**

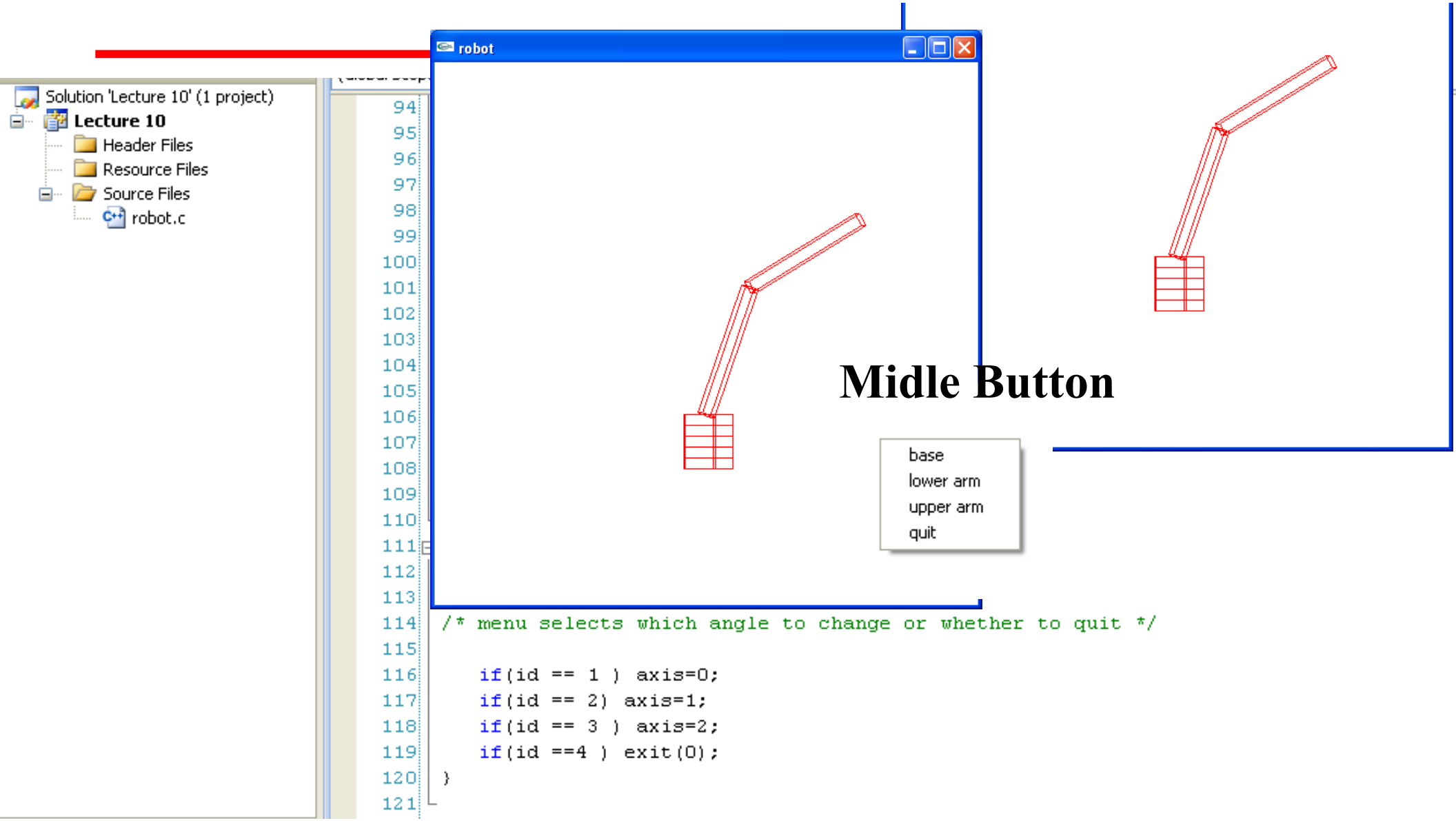
**Matrix** relating **Node** to **Parent**

**Modeling with Trees**

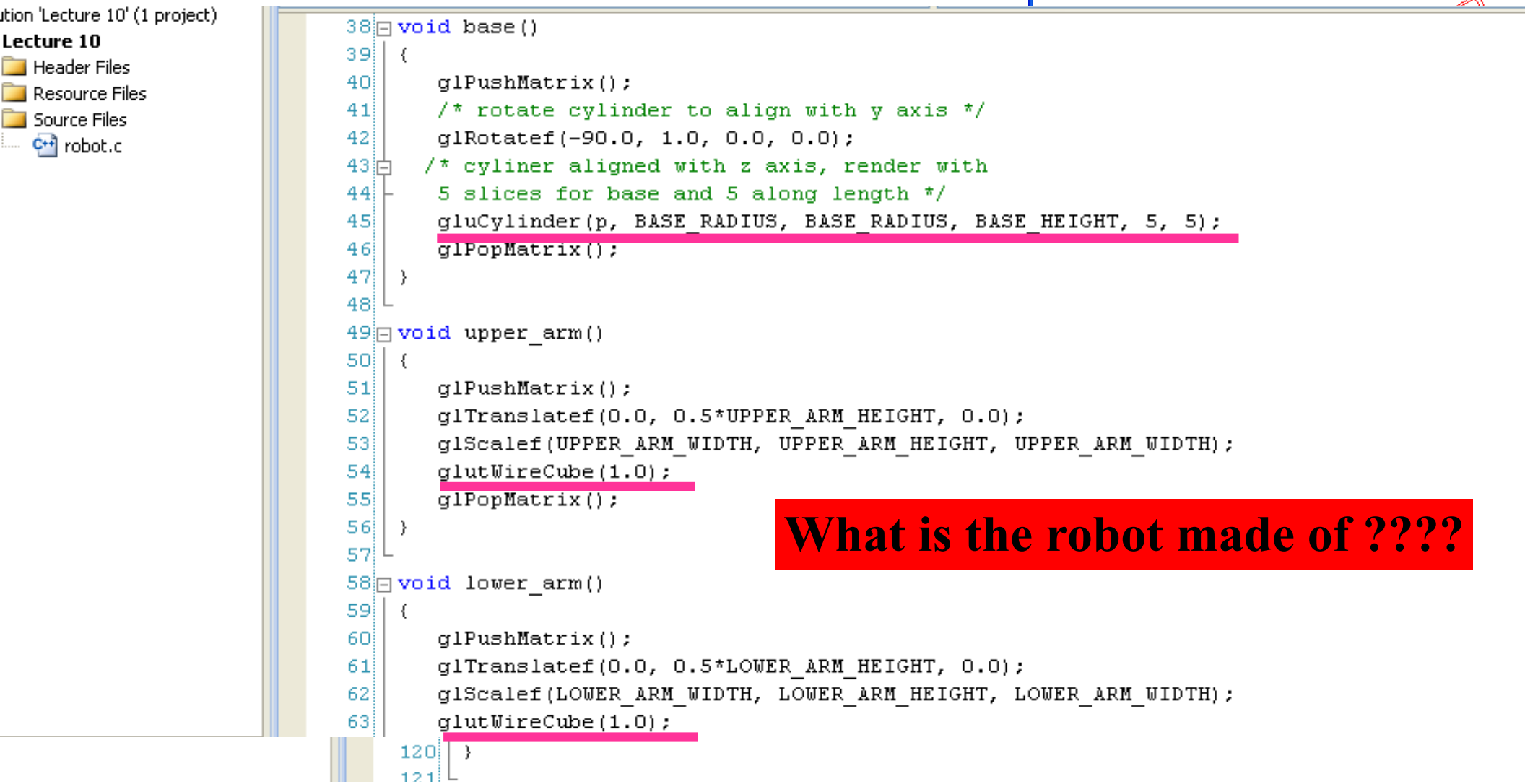
- Must decide what information to place in **Nodes** and what to put in **Edges**
- **Nodes**
  - **What to draw**
  - Pointers to Children
- **Edges**
  - May have information on **incremental changes to transformation matrices** (can also store in **nodes**)



# Robot Arm


Solution 'Lecture 10' (1 project)
Lecture 10
Header Files
Resource Files
Source Files
robot.c
robot
Midle Button
base
lower arm
upper arm
quit
/* menu selects which angle to change or whether to quit */
if(id == 1 ) axis=0;
if(id == 2) axis=1;
if(id == 3 ) axis=2;
if(id ==4 ) exit(0);
}

# Robot Arm


ution 'Lecture 10' (1 project)
Lecture 10
Header Files
Resource Files
Source Files
robot.c
38 void base()
39 {
40 glPushMatrix();
41 /* rotate cylinder to align with y axis */
42 glRotatef(-90.0, 1.0, 0.0, 0.0);
43 /* cyliner aligned with z axis, render with
44 5 slices for base and 5 along length */
45 gluCylinder(p, BASE_RADIUS, BASE_RADIUS, BASE_HEIGHT, 5, 5);
46 glPopMatrix();
47 }
48
49 void upper_arm()
50 {
51 glPushMatrix();
52 glTranslatef(0.0, 0.5*UPPER_ARM_HEIGHT, 0.0);
53 glScalef(UPPER_ARM_WIDTH, UPPER_ARM_HEIGHT, UPPER_ARM_WIDTH);
54 glutWireCube(1.0);
55 glPopMatrix();
56 }
57
58 void lower_arm()
59 {
60 glPushMatrix();
61 glTranslatef(0.0, 0.5*LOWER_ARM_HEIGHT, 0.0);
62 glScalef(LOWER_ARM_WIDTH, LOWER_ARM_HEIGHT, LOWER_ARM_WIDTH);
63 glutWireCube(1.0);
120 }
What is the robot made of ????

# Generalizations

## Need to deal with multiple Children

How do we represent a more General Tree?

How do we traverse such a Data Structure?

## Animation

How to use dynamically?

Can we create **and** delete Tree Nodes during execution?

# Hierarchical Modeling II

# Objectives

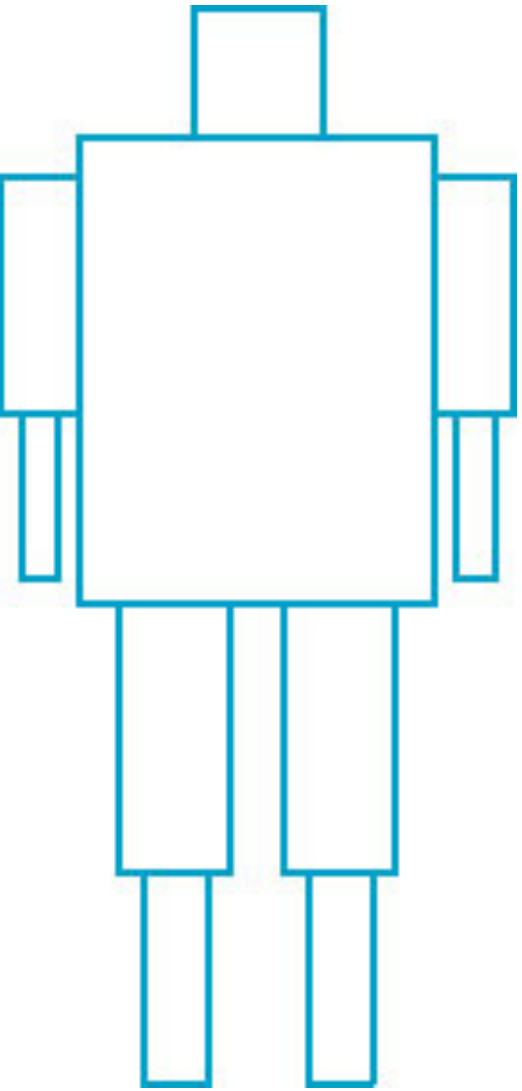
- Build a Tree-structured **Model** of a humanoid figure
- Examine various **Traversal strategies**
- Build a generalized Tree-**Model** structure that is independent of the **particular Model**

# Objectives

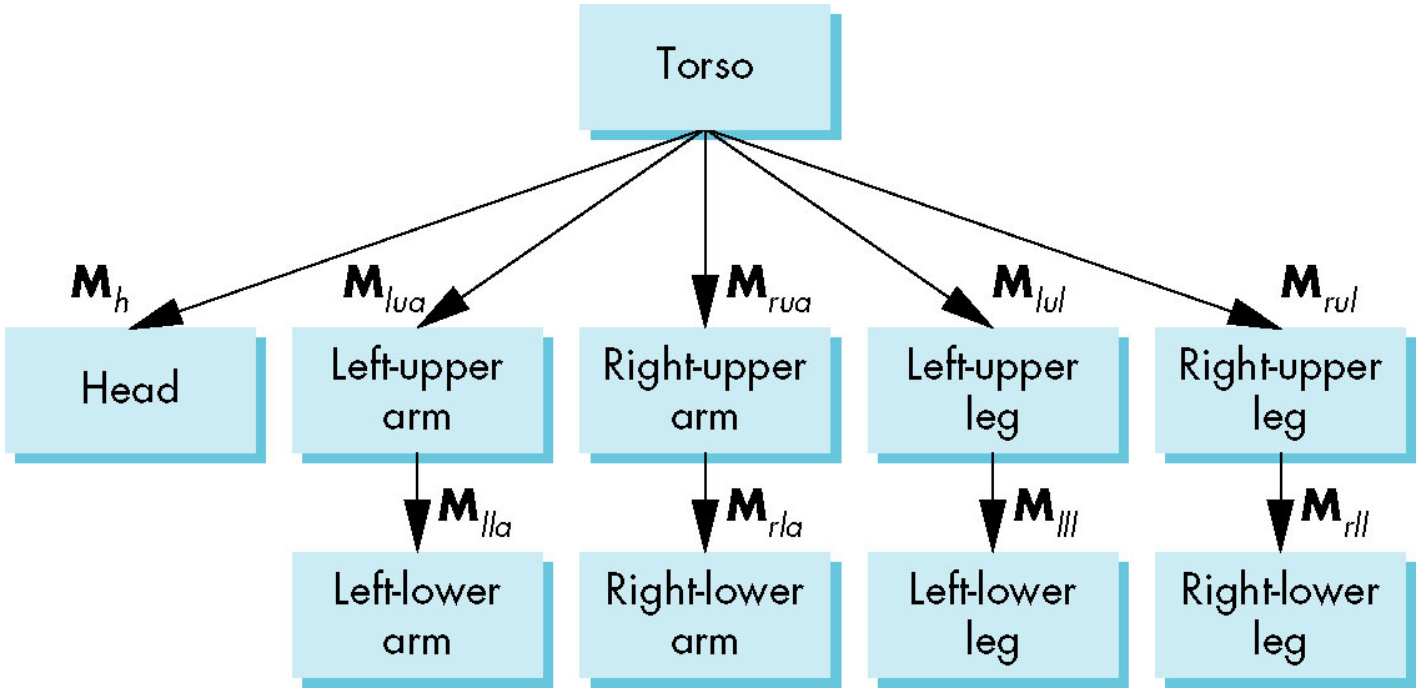
Build a Tree-structured **Model** of a humanoid figure

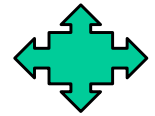
Examine various **traversal strategies**

Build a generalized tree-model structure that is independent of the particular model



# Humanoid Figure



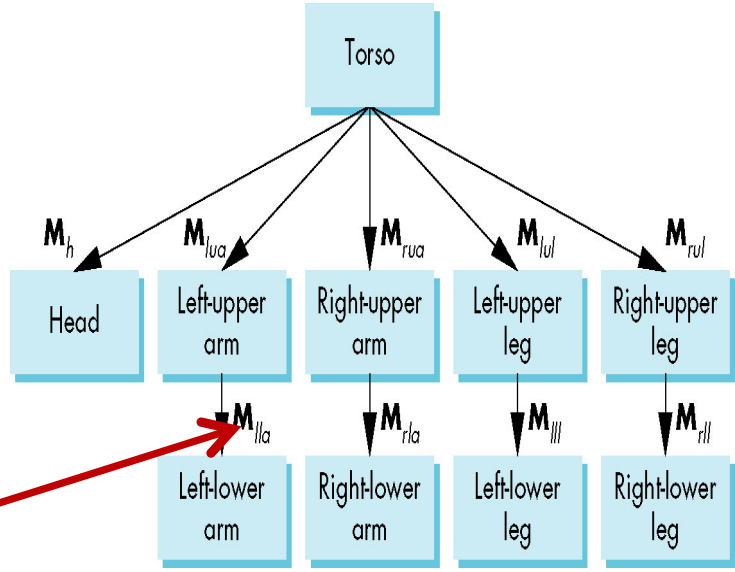



# Building the Model

- Can build a simple implementation using quadrics: ellipsoids and cylinders

- Access **Parts** through functions

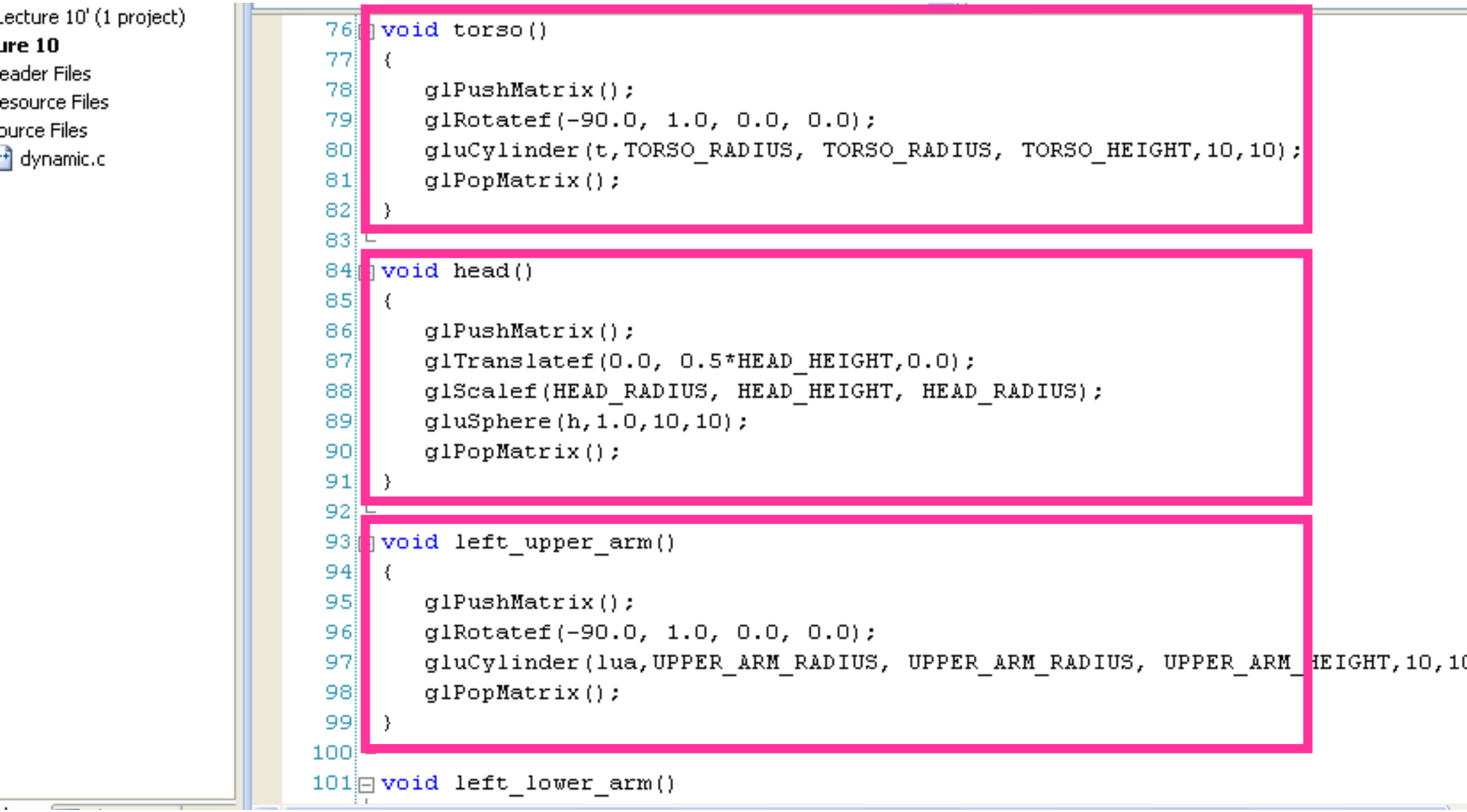
```
torso()
left_upper_arm()
```



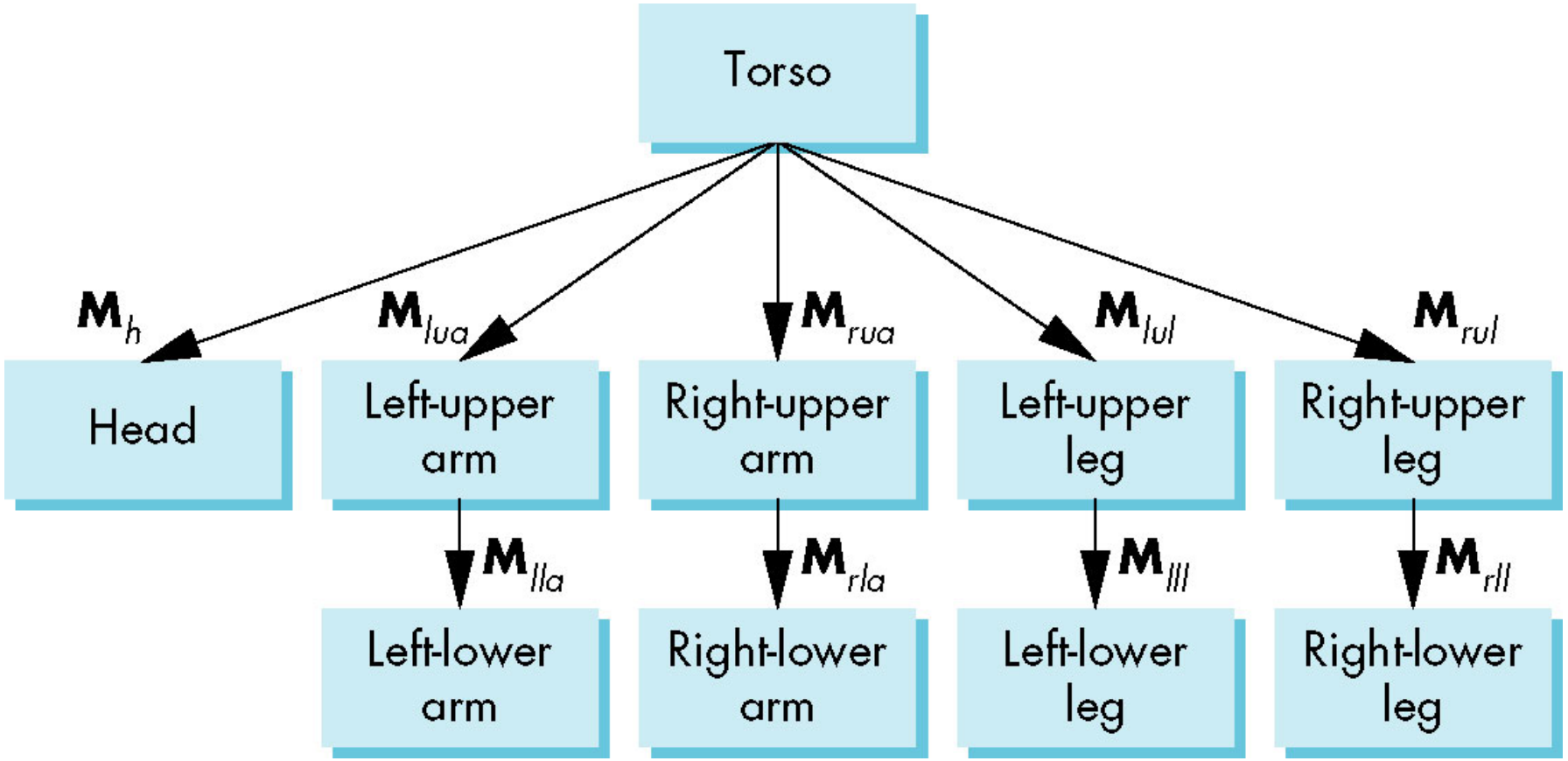

- **Matrices** describe position of **node** with respect to its **parent**

  $M_{lla}$ positions **left lower arm** with respect to **left upper arm**

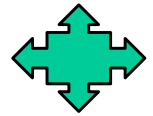
# Building the Model with Cylinder



# Tree with Matrices
(pre-order traversal)



# Objectives



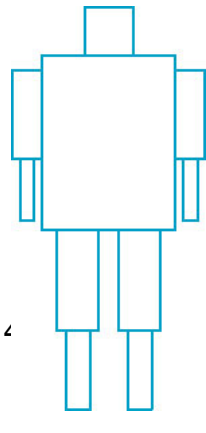
- Build a tree-structured model of a humanoid figure
- Examine various **Traversal strategies**
- Build a generalized tree-model structure that is independent of the particular model

# Display and Traversal

The position of the Figure is determined by 11 joint angles (two for the head and one for each other part)

**Display of the Tree** requires a ***graph Traversal***

- **Visit each node once**
- **Display function at each node that describes the part associated with the node, applying the correct transformation matrix for position and orientation**



# Transformation Matrices

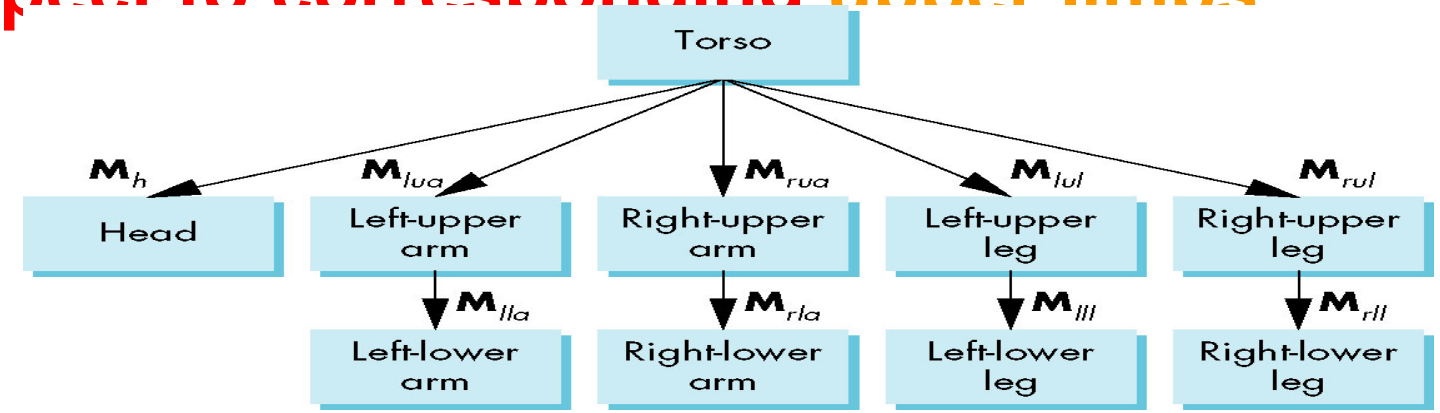
- There are 10 relevant **Matrices**

  **M positions and orients entire Figure through the torso which is the Root node**

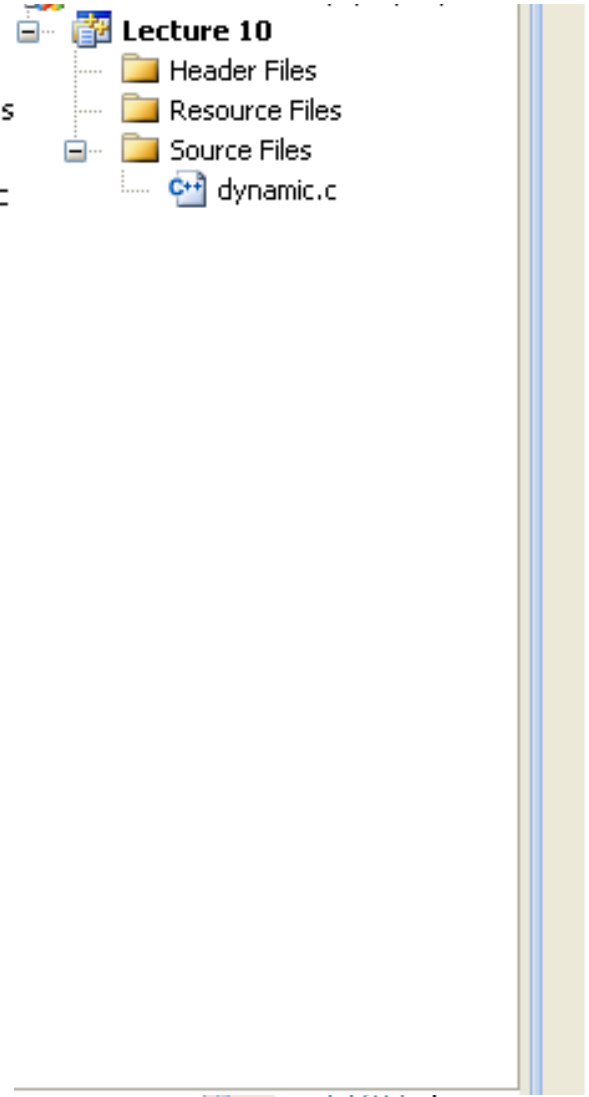
  $\mathbf{M}_h$ **positions head with respect to torso**

  $\mathbf{M}_{lua}$, $\mathbf{M}_{rua}$, $\mathbf{M}_{lul}$, $\mathbf{M}_{rul}$ **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**



# Display and Traversal



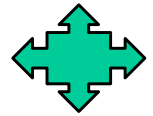

```
55
56  double size=1.0;
57
58  t_ptr torso_ptr, head_ptr, lua_ptr, rua_ptr, lll_ptr, rll_ptr,
59                  lla_ptr, rla_ptr, rul_ptr, lul_ptr;
60
61  void traverse(t_ptr root)
62  {
63     if(root==NULL) return;
64
65     glPushMatrix();
66     glMultMatrixf(root->m);
67     root->f();
68
69     if(root->child!=NULL) traverse(root->child);
70
71     glPopMatrix();
72
73     if(root->sibling!=NULL) traverse(root->sibling);
74  }
75
76  void torso()
77  {
78     glPushMatrix();
79     glRotatef(-90.0, 1.0, 0.0, 0.0);
80     gluCylinder(t,TORSO_RADIUS, TORSO_RADIUS, TORSO_HEIGHT,10,10);
```

Traversal ??????

root L R – preOrder

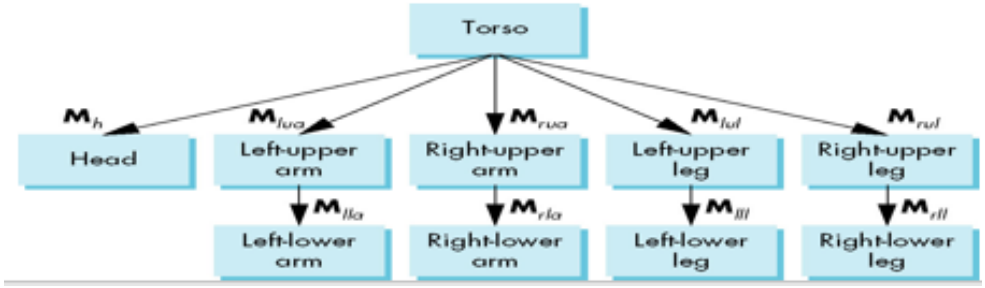
```
173     if(btn==GLUT_LEFT_BUTTON && state == GLUT_DOWN)
```

# 1. Stack-based Traversal (explicitly traversal)

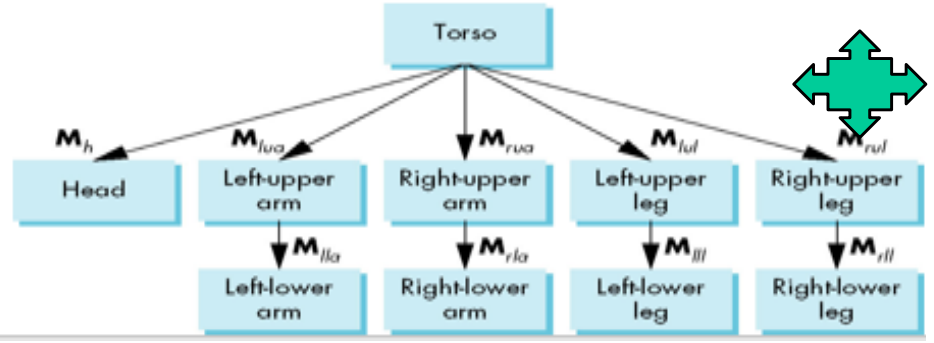


- Set **model-view matrix** to $\mathbf{M}$ and **draw torso**
- Set **model-view matrix** to $\mathbf{M}\mathbf{M}_h$ and **draw head**
- For left-upper arm need $\mathbf{M}\mathbf{M}_{lua}$ and so on

- Rather than recomputing $\mathbf{M}\mathbf{M}_{lua}$ 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**

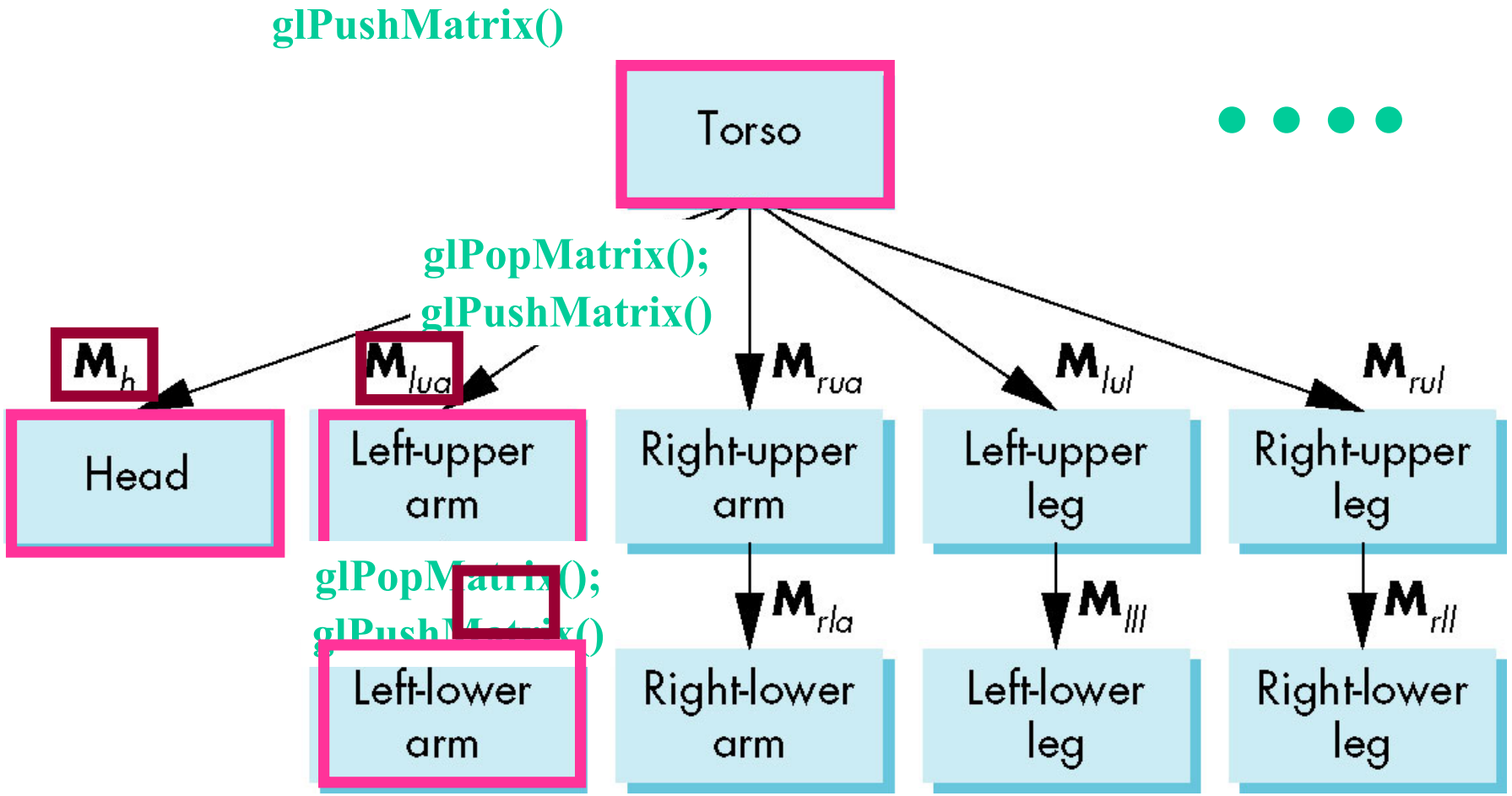


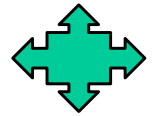
# Traversal Code




```
figure()
{
     glPushMatrix();
        torso();
        glTranslate();
        glRotate3():
        head();
     glPopMatrix();
     glPushMatrix();
        glTranslate();
        glRotate3();
        left_upper_leg();
        glTranslate();
        glRotate3();
        left_lower_leg();
     glPopMatrix();
     glPushMatrix();
        glTranslate();
        glRotate3();
        right_upper_leg();
     glPopMatrix();
     glPushMatrix();
     .
     .
     .
```

# Tree with Matrices





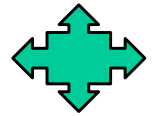
# Analysis

- The code describes a **particular Tree** and a **particular traversal strategy**
  - Can we **develop a more general approach**?
- Note that the sample code **does not include state changes, such as changes to colors**
  - May also want to use `glPushAttrib` and `glPopAttrib` to protect against **unexpected state changes** affecting later parts of the code

# Objectives

- Build a tree-structured model of a humanoid figure
- Examine various **traversal strategies**
- Build a **Generalized Tree Model structure** that is independent of the **particular Model**

# 2. General Tree Data Structure (traversal)



- Need a data structure to **represent Tree** and an **algorithm to traverse the tree**

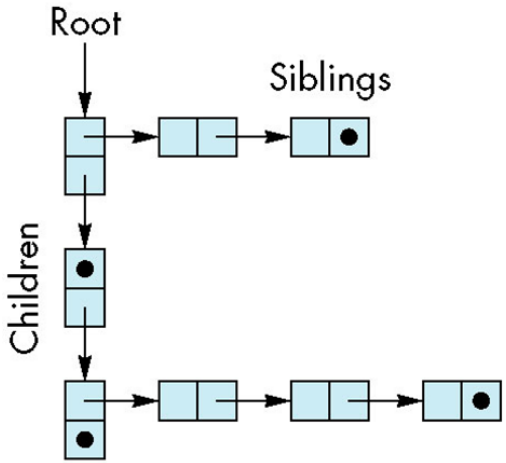
- We will use a ***left-child right sibling*** structure

  **Uses Linked Lists**

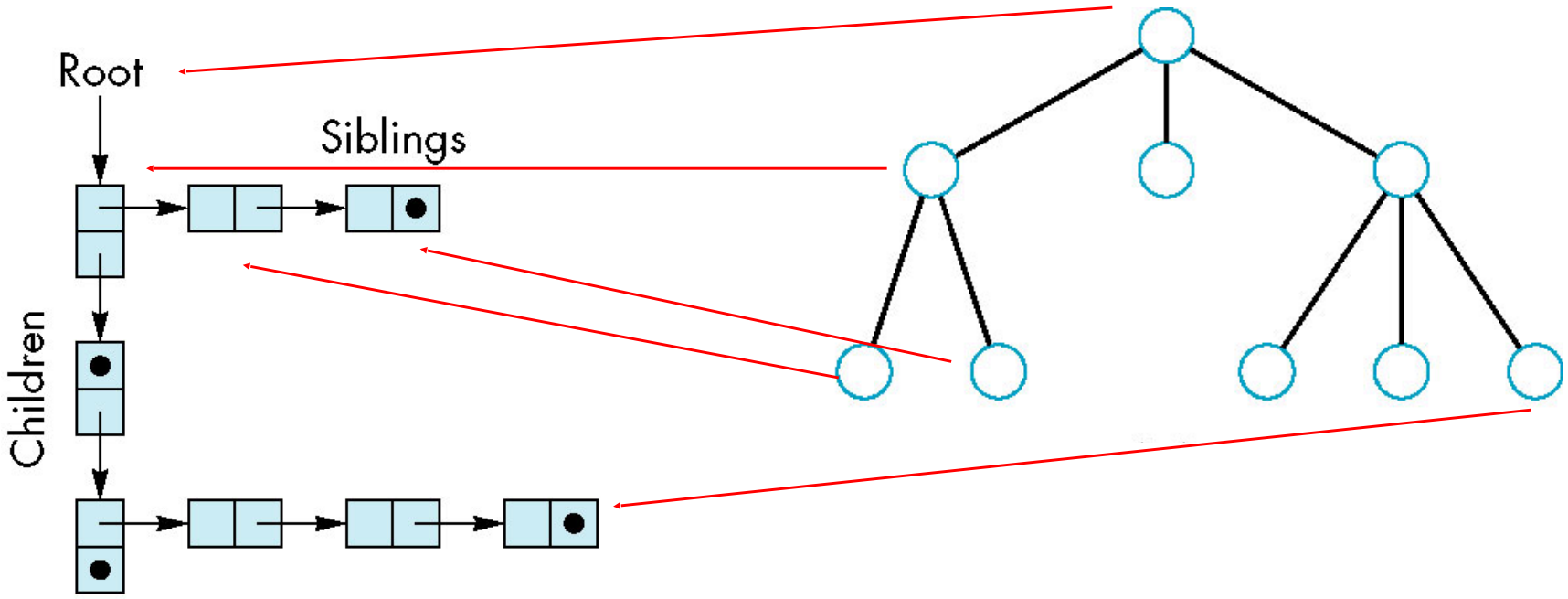
  **Each node in data structure has two pointers**

  **Left: next node**

  **Right: Linked List of children**

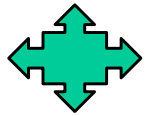


# Left-Child Right-Sibling Tree



# Tree node Structure

```
typedef struct treenode
{
   GLfloat m[16];
   void (*f)();
   struct treenode *sibling;
   struct treenode *child;
} treenode;
```



At each **node** we need to store

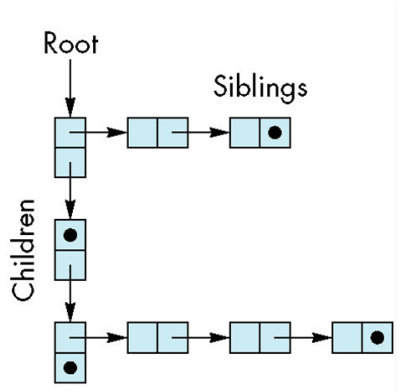
Pointer to a **function f that draws the Object represented by the node**

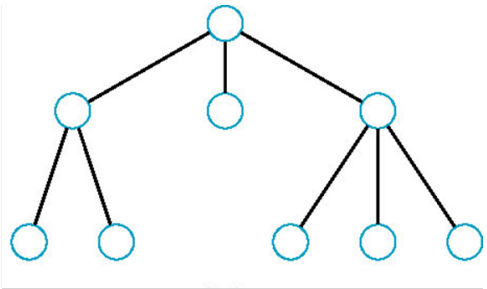
Homogeneous coordinate matrix **m** to multiply on the right of the current **Model-View Matrix**

- Represents **changes** going from **parent** to **node**
- In OpenGL this matrix is a 1D array storing matrix by columns

Pointer to **sibling**

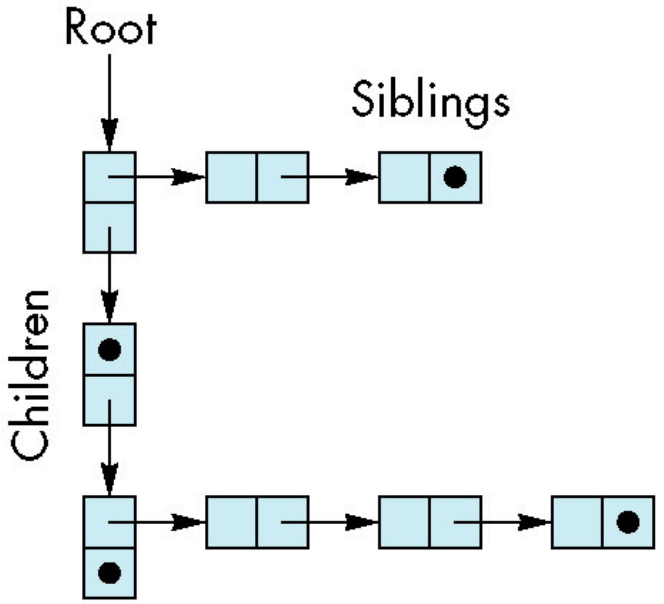
Pointer to **child**



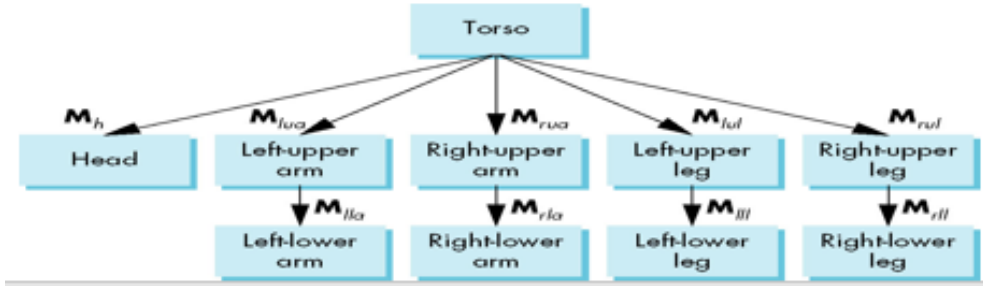
# C Definition of treenode

```
typedef struct treenode
{
   GLfloat m[16];
   void (*f)();
   struct treenode *sibling;
   struct treenode *child;
} treenode;
```




```
typedef struct treenode
{
     GLfloat m[16];
     void (*f)();          // draw
     struct treenode *sibling;
     struct treenode *child;
} treenode;
```

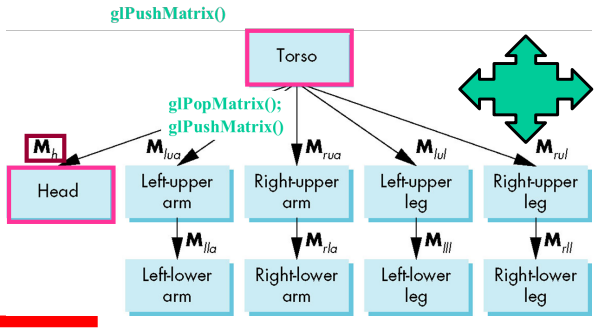
# C Definition of treenode




```
34  void left_upper_arm();
35  void right_upper_arm();
36  void left_upper_leg();
37  void right_upper_leg();
38
39  typedef float point[3];
40
41
42  typedef struct treenode
43  {
44
45    GLfloat m[16];
46    void (*f)();
47    struct treenode *sibling;
48    struct treenode *child;
49
50  } treenode, *t_ptr;
51
52
53  static GLfloat theta[11] = {0.0,0.0,0.0,0.0,0.0,0.0,0.0,
54                180.0,0.0,180.0,0.0}; /* initial joint angles */
55  static GLint angle = 2;
56
57  GLUquadricObj *t, *h, *lua, *lla, *rua, *rla, *lll, *rll, *rul, *lul;
```

```
typedef struct treenode
{
     GLfloat m[16];
     void (*f)();      // draw
     struct treenode *sibling;
     struct treenode *child;
} treenode;
```

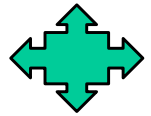
# Defining the torso node



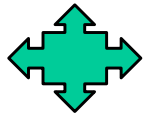

```
treenode torso_node, head_node, lua_node, … ;

/* use OpenGL functions to form matrix */
glLoadIdentity();
glRotatef(theta[0], 0.0, 1.0, 0.0);

/* move model-view matrix to m */
glGetFloatv(GL_MODELVIEW_MATRIX, torso_node.m)

torso_node.f = torso; /* torso() draws torso */
torso_node.sibling = NULL;
torso_node.child = &head_node;
```

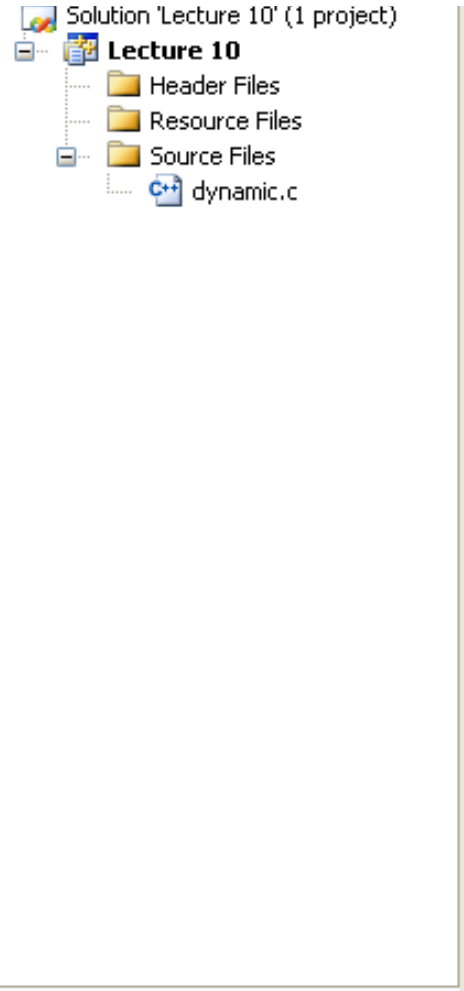
```
typedef struct treenode
{
     GLfloat m[16];
     void (*f)();      // draw
     struct treenode *sibling;
     struct treenode *child;
} treenode;
```

# Defining the torso node

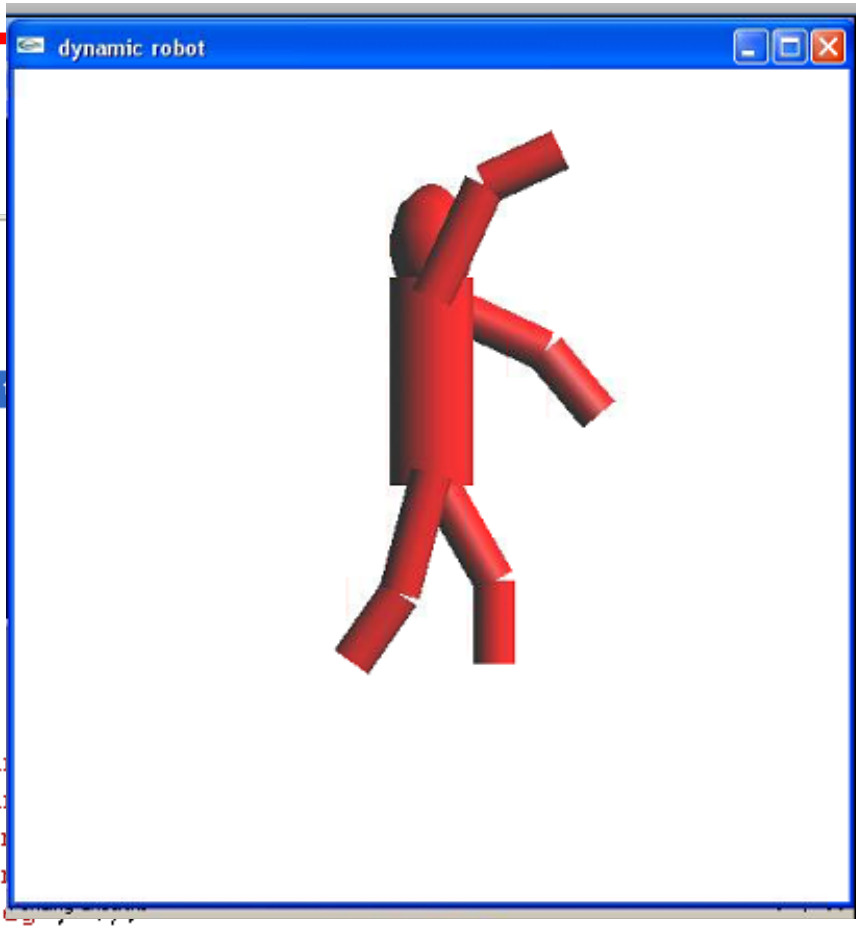


```
185     }
186     glPushMatrix();
187     switch(angle)
188     {
189     case 0 :
190         glLoadIdentity();
191         glRotatef(theta[0], 0.0, 1.0, 0.0);
192         glGetFloatv(GL_MODELVIEW_MATRIX,torso_ptr->m);
193         break;
194
195     case 1 : case 2 :
196         glLoadIdentity();
197         glTranslatef(0.0, TORSO_HEIGHT+0.5*HEAD_HEIGHT, 0.0);
198         glRotatef(theta[1], 1.0, 0.0, 0.0);
199         glRotatef(theta[2], 0.0, 1.0, 0.0);
200         glTranslatef(0.0, -0.5*HEAD_HEIGHT, 0.0);
201         glGetFloatv(GL_MODELVIEW_MATRIX,head_ptr->m);
202         break;
203
204     case 3 :
205         glLoadIdentity();
206         glTranslatef(-(TORSO_RADIUS+UPPER_ARM_RADIUS), 0.9*TORSO_HEIGHT, 0.0);
207         glRotatef(theta[3], 1.0, 0.0, 0.0);
208         glGetFloatv(GL_MODELVIEW_MATRIX,lua_ptr->m);
209         break;
210
```

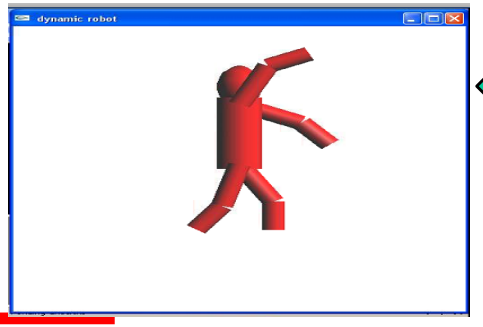





```
433  {
434      glutInit(&argc, argv);
435      glutInitDisplayMode(GLUT_DOUBLE
436      glutInitWindowSize(500, 500);
437      glutCreateWindow(" dynamic robo
438      myinit();
439      glutReshapeFunc(myReshape);
440      glutDisplayFunc(display);
441      glutMouseFunc(mouse);
442
443      glutCreateMenu(menu);
444      glutAddMenuEntry("torso", 0);
445      glutAddMenuEntry("head1", 1);
446      glutAddMenuEntry("head2", 2);
447      glutAddMenuEntry("right_upper_a
448      glutAddMenuEntry("right_lower_a
449      glutAddMenuEntry("left_upper_ar
450      glutAddMenuEntry("left_lower_ar
451      glutAddMenuEntry("right_upper_le
452      glutAddMenuEntry("right_lower_leg", 8);
453      glutAddMenuEntry("left_upper_leg", 9);
454      glutAddMenuEntry("left_lower_leg", 10);
455      glutAddMenuEntry("quit", 11);
456      glutAttachMenu(GLUT_MIDDLE_BUTTON);
457
458      glutMainLoop();
```



# Notes



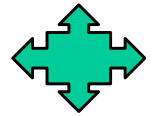

The **position** of **Figure** is determined by 11 joint angles stored in **`theta[11]`**

**Animate** by changing the **angles** and redisplaying

We form the required **Matrices** using **`glRotate`** and **`glTranslate`**

- More efficient than software
- Because the **matrix** is formed in **model-view Matrix**, we may want to **first push original model-view Matrix on Matrix stack**

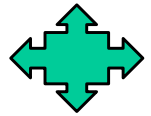
# Graphical Objects and Scene Graphs



# Objectives

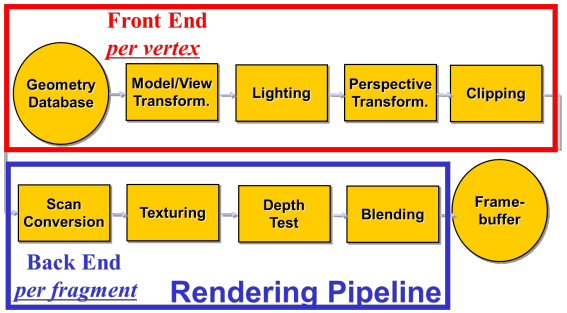
- Introduce **Graphical Objects**
- Generalize the notion of **Objects** to include **lights**, **cameras**, **attributes**
- Introduce **Scene Graphs**

# Limitations of Immediate Mode Graphics



When we define a **geometric Object** in an application, **upon execution of the code the Object** is passed through the pipeline

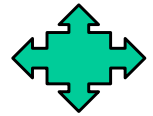
**It then** disappears from the graphical system




To **redraw the Object**, either changed or the same, we must **re-execute the code**

**Display lists** provide only a partial solution to this problem

# OpenGL and Objects



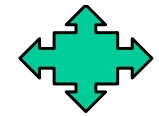
**OpenGL** lacks an **Object Orientation**

Consider, for example, a **green Sphere**

- We can **Model the Sphere with polygons** or use **OpenGL quadrics**
- Its **color** is determined by the **OpenGL** State and is not a property of the **Object**

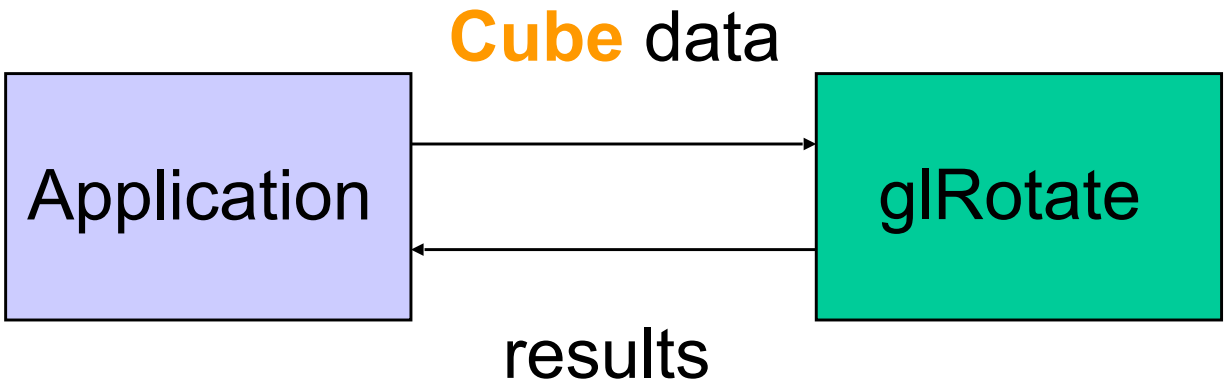
Defies our notion of a **physical Object**

We can try to **build better Objects in code using Object Oriented Languages/Techniques**



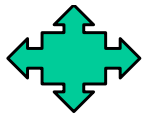
# Imperative Programming Model

Example: **rotate** a **Cube**




The **rotation** function must know how the **Cube** is represented

- **Vertex list**
- **Edge list**



# Object Oriented Programming Model

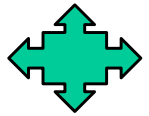
In this Model, the representation is stored with the object


The application sends a *message* to the object

The object contains functions (*methods*) which allow it to transform itself

# C/C++

```
struct  cube
{
   float color[3];
   float matrix[4][4];

/* implementation goes here */

}
```



- Can try to use **C structs** to **build objects**

- **C++** provides better support
  - Use **class** construct
  - Can hide implementation using **public**, **private**, and **protected members** in a **class**

**Can also use friend designation to allow classes to access each other**

# Cube object
(geometric object)

```
struct  cube
{
   float color[3];
   float matrix[4][4];

/* implementation goes here */

}
```

Suppose that we want to create a **Cube object** that we can **scale, orient, position** and **set its color** directly through code such as

```
cube mycube;
mycube.color[0]=1.0;
mycube.color[1]= mycube.color[2]=0.0;
mycube.matrix[0][0]=.....
```

# Cube object Functions

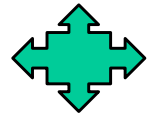
```
struct  cube
{
   float color[3];
   float matrix[4][4];

/* implementation goes here */

}
```

- We would also like to have **functions** that act on the Cube object such as

```
mycube.translate(1.0, 0.0,0.0);
mycube.rotate(theta, 1.0, 0.0, 0.0);
setcolor(mycube, 1.0, 0.0, 0.0);
```

- We also need a way of **displaying** the **Cube**

```
mycube.render();
```



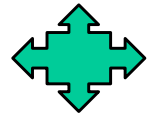
# The Implementation

- Can use any implementation in the **private part** such as a **vertex** list

- The **private part** has access to **public members** and the implementation of **class methods** can use any implementation without making it visible

- **Render method** is tricky, but it will invoke the standard **OpenGL** drawing functions such as `glVertex`

# Other objects
## (object-oriented graphics system)



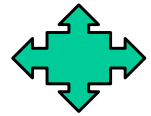
- Other **objects** have **geometric** aspects
  - Cameras
  - Light sources

- But we should be able to have **nongeometric objects** too
  - Materials
  - Colors
  - Transformations (matrices)

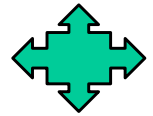
# Application Code

```
cube mycube;
material plastic;
mycube.setMaterial(plastic);

camera frontView;
frontView.position(x ,y, z);
```

# Light object
(geometric object)



```
class light {      // match Phong model
  private:
   boolean type;
   boolean near;
   float position[3];
   float orientation[3];
   float specular[3];
   float diffuse[3];
   float ambient[3];
}
```



# Scene Descriptions

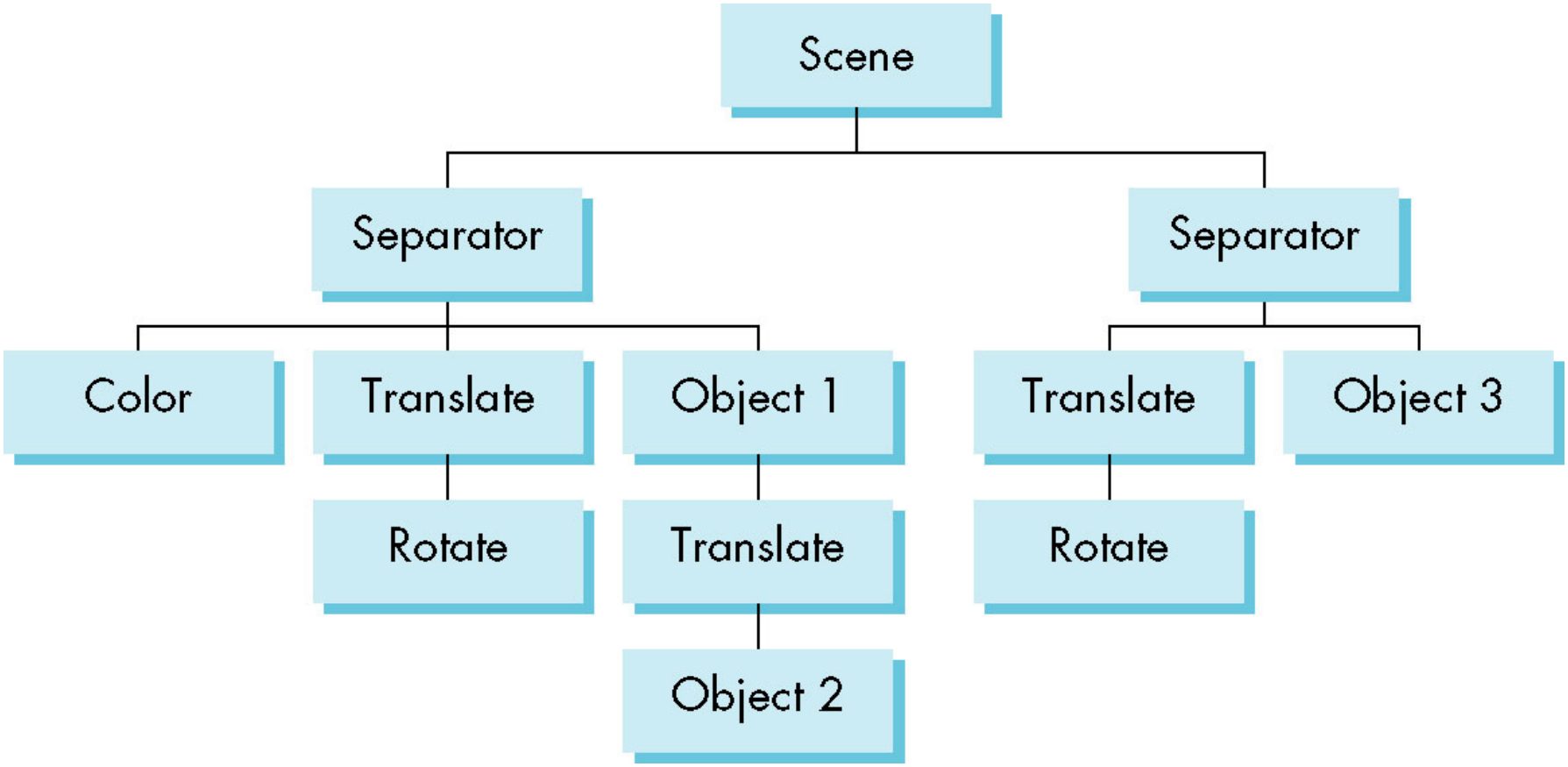
If we recall **Figure Model**, we saw that

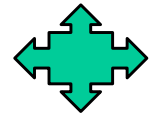
- We could **describe model** either by Tree or by equivalent code
- We could write a generic **Traversal** to **display**

If we can represent all the elements of a scene (**cameras, lights, materials, geometry**) as **C++ objects**, we should be able to show them in a Tree

- **Render scene by traversing this Tree**

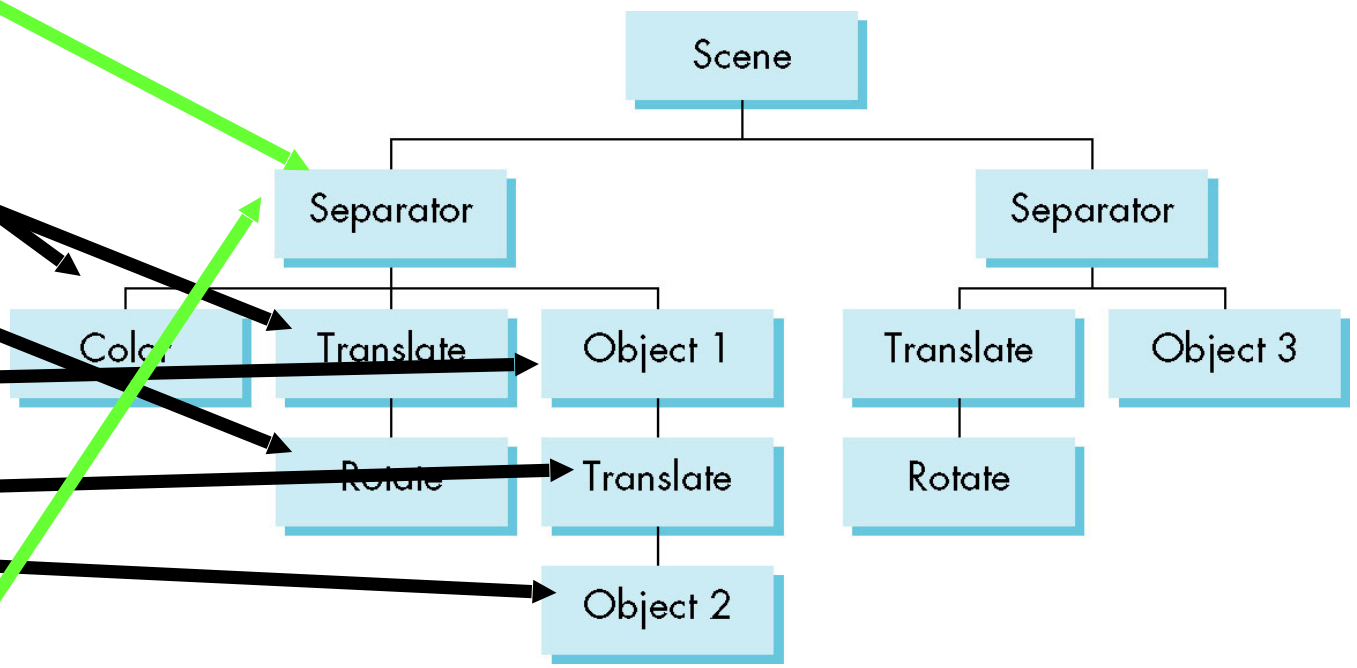
# Scene Graph


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



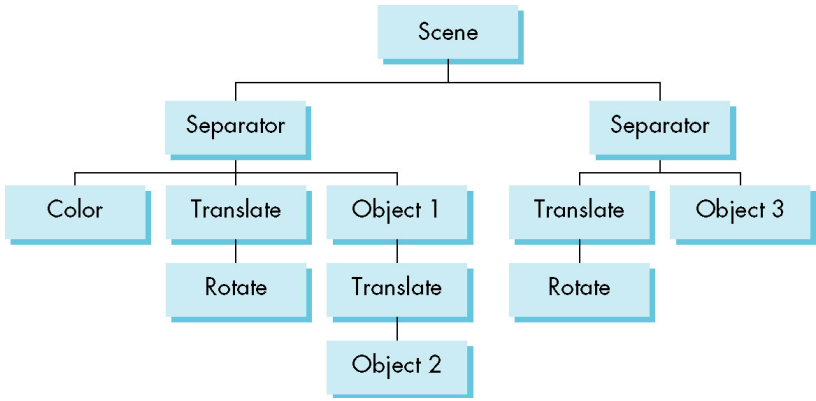
# PreOrder Traversal

```
glPushAttrib
glPushMatrix
glColor
glTranslate
glRotate
object1
glTranslate
object2
glPopMatrix
glPopAttrib
...
```



# Group Nodes

- Necessary to isolate **state changes**
  - Equivalent to **OpenGL** Push/Pop
- Note that as with the **Figure Model**
  - We can write a **universal traversal algorithm**
  - **The order of traversal can matter**
    - If we do not use the **group node**, **state changes** can persist



# Inventor and Java3D

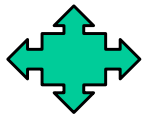
nventor and Java3D provide a **Scene Graph APIs**

**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**

http://www.ocnus.com/models/

http://www.cs.iupui.edu/~aharris/mm/texture/texCar.wrl

http://www.cs.iupui.edu/~aharris/webDesign/vrml/plane.wrl

http://www.cs.iupui.edu/~aharris/mm/figure/figure.wrl

# NEXT.

| | | |
|---|---|---|
| 10.30.2023 (M 5:30 to 7)<br>(20) | | PROJECT 3 |
| 11.01.2023 (W 5:30 to 7)<br>(21) | | EXAM 3 REVIEW |
| 11.06.2023 (M 5:30 to 7)<br>(22) | | EXAM 3 |

**At 6:45 PM.**

# End Class 19

**VH, Download Attendance Report**
**Rename it:**
**10.25.2023 Attendance Report FINAL**

**VH, upload Lecture 10 to CANVAS.**