# 10. Modeling and Hierarchy

# Overview

- Reading: ANG Ch. 10, except 10.9-10.11
  - Hierarchical Modeling I
  - Hierarchical Modeling II
  - Graphical Objects and Scene Graphs
- Sample Programs
  - Robot program
  - Figure program

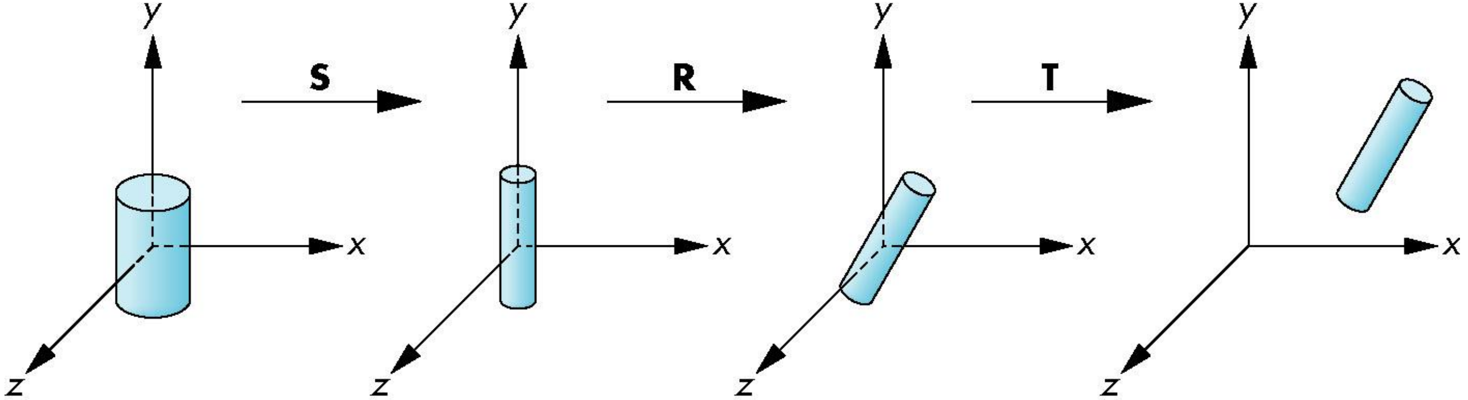
# Hierarchical Modeling I

# 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 a prototype object (a *symbol*)
- Each appearance of the object in the model is an *instance*
  - Must scale, orient, position
  - Defines instance transformation



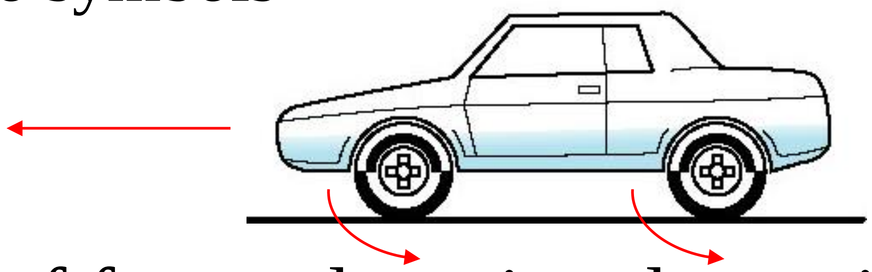
# Symbol-Instance Table

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

| Symbol | Scale | Rotate | Translate |
| --- | --- | --- | --- |
| 1 | $s_x, s_y, s_z$ | $\theta_x, \theta_y, \theta_z$ | $d_x, d_y, d_z$ |
| 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



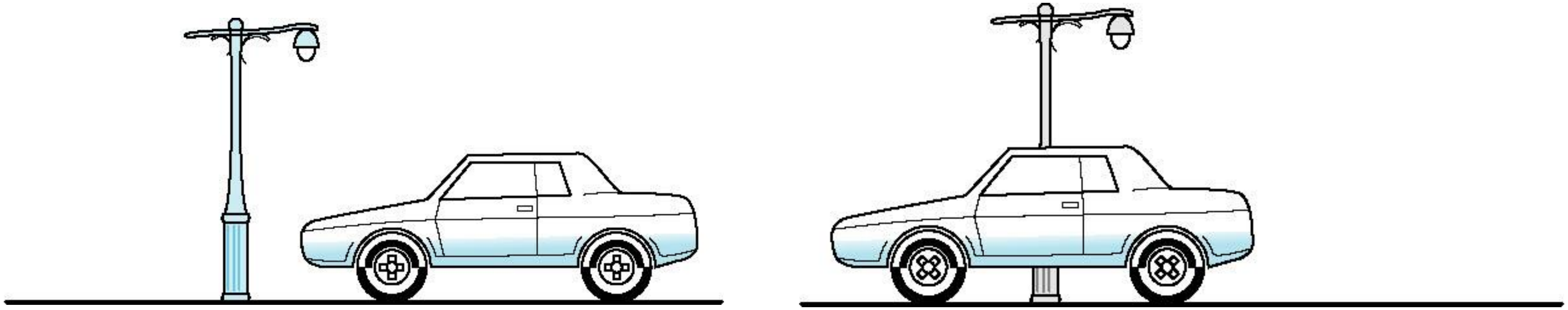
- Rate of forward motion determined by rotational speed of wheels

# Structure Through Function Calls

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

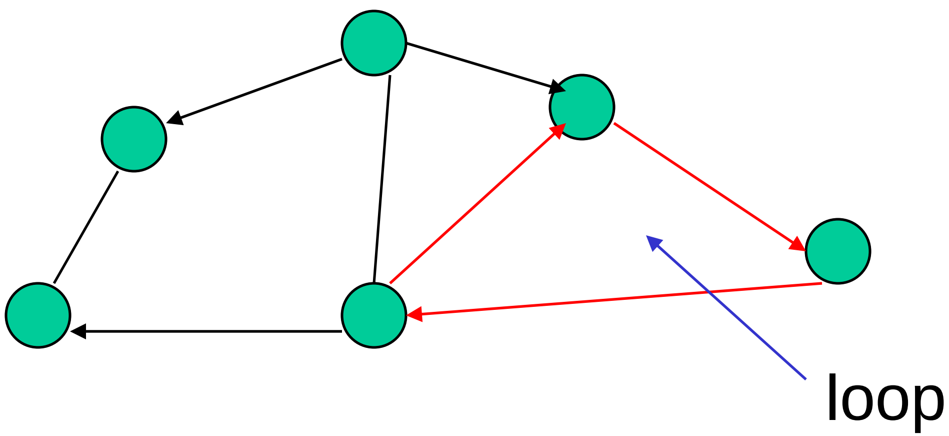
- Fails to show relationships well
- Look at problem using a graph

# Two-frame of Animation



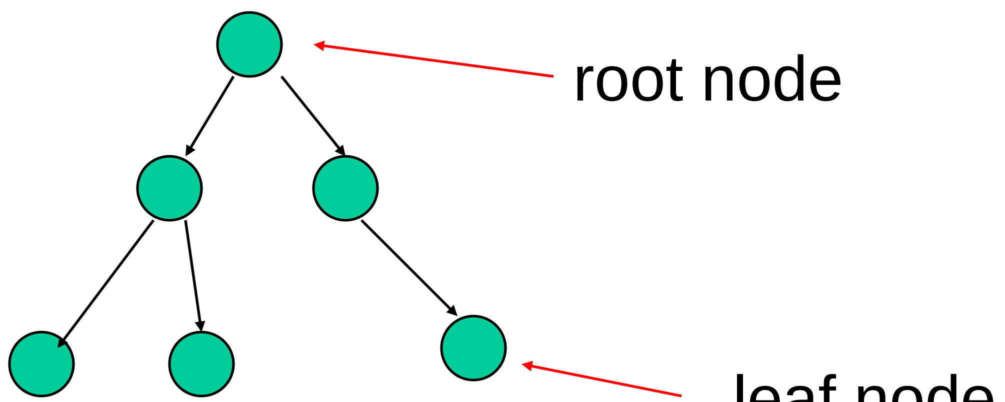
# Graphs

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

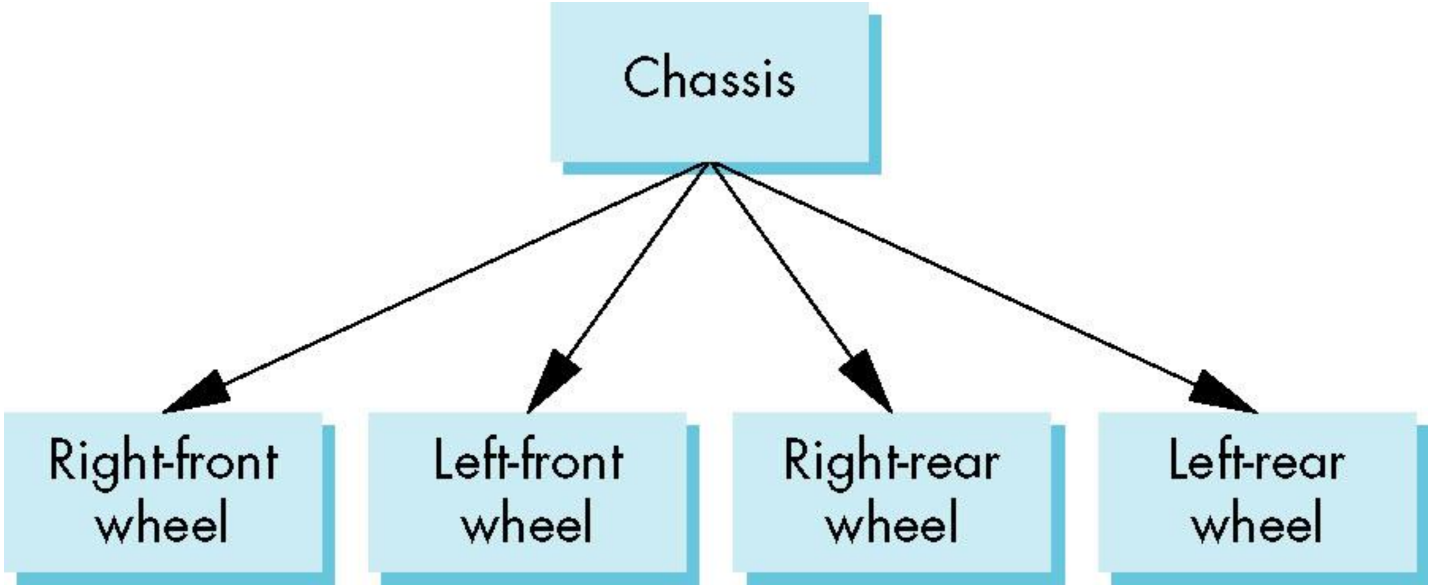


# Tree

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

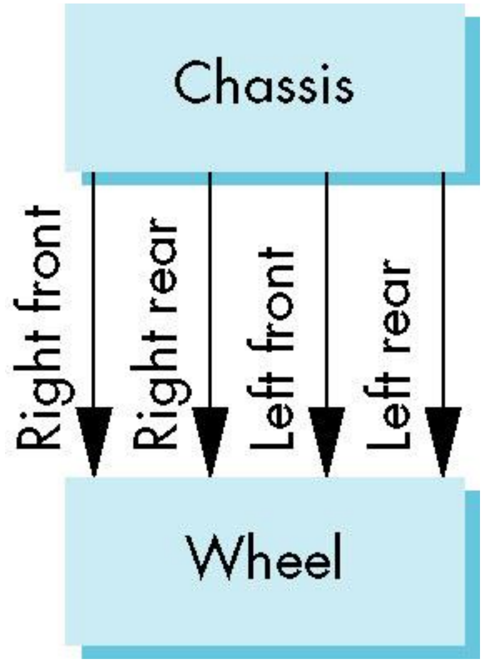


# Tree Model of Car



# DAG Model

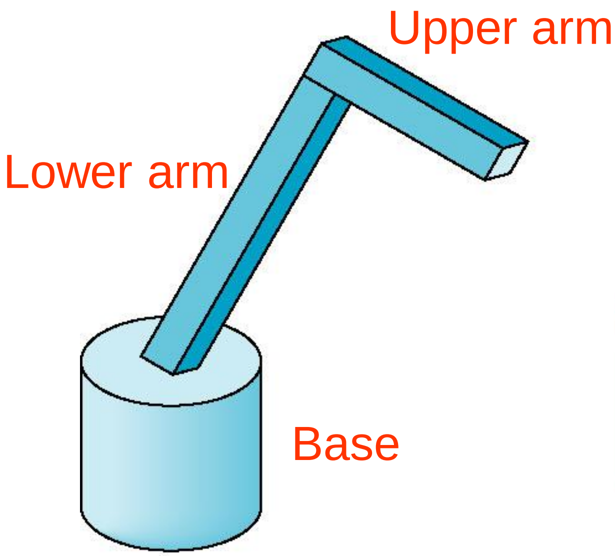
- If we use the fact that all the wheels are identical, we get a *directed acyclic graph*
  - Not much different than dealing with a tree



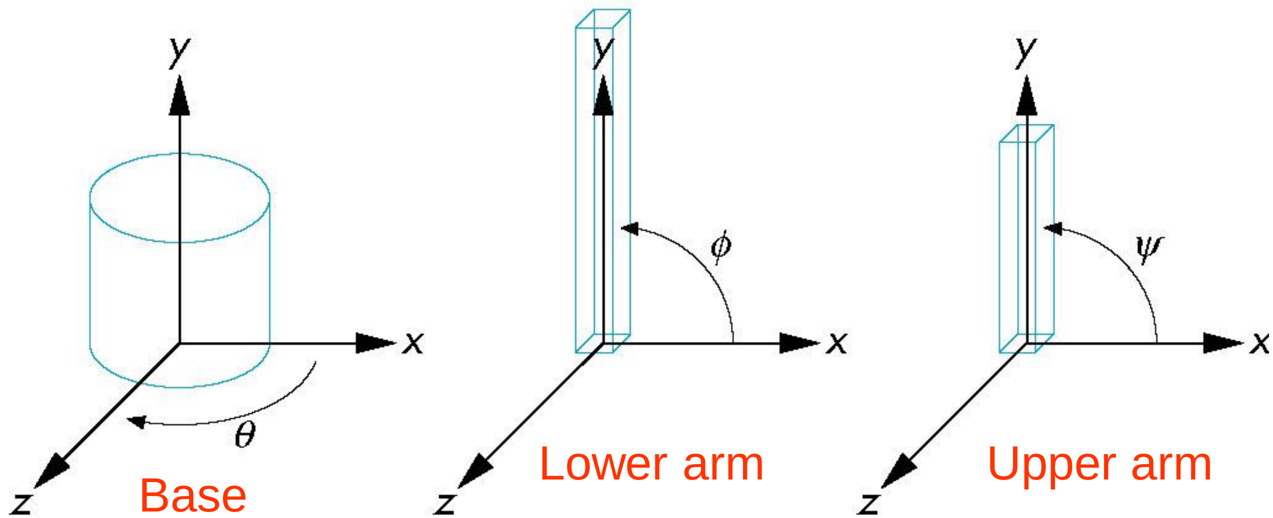
# Modeling with Trees

- 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



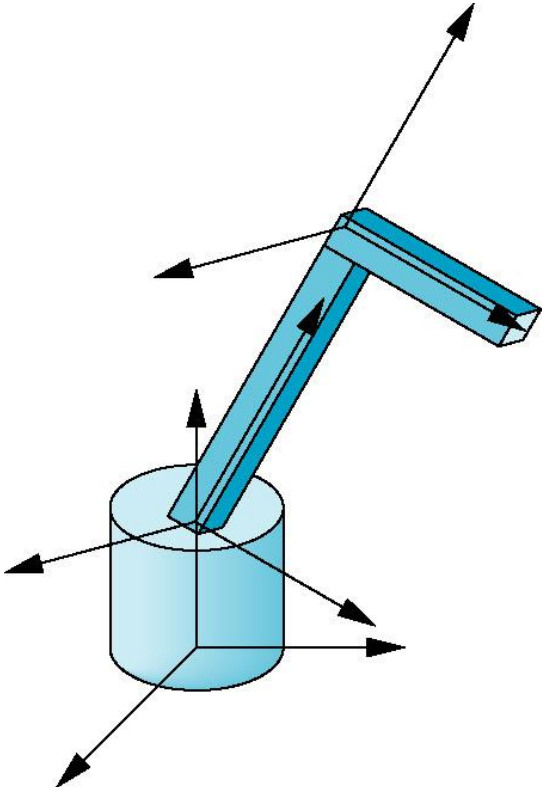

robot arm



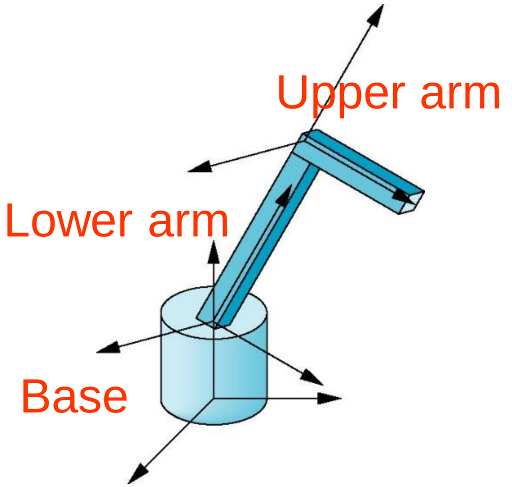

parts in their own coodinate systems

# 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

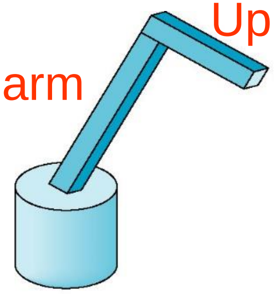


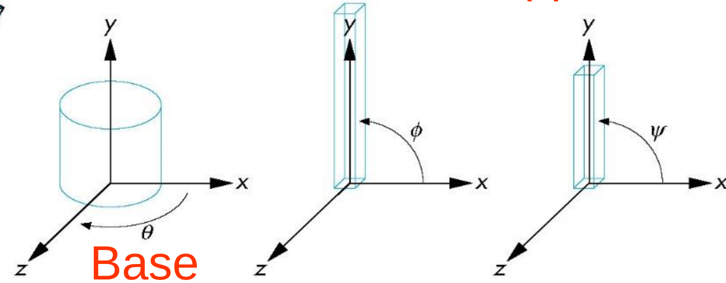
# Relationships in Robot Arm




- 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
  - Its position depends on both base and lower arm
  - Must translate relative to lower arm and rotate about joint connecting to lower arm

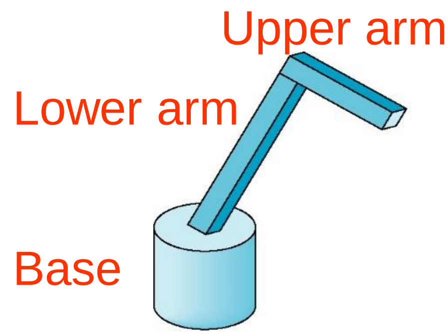
# Required Matrices

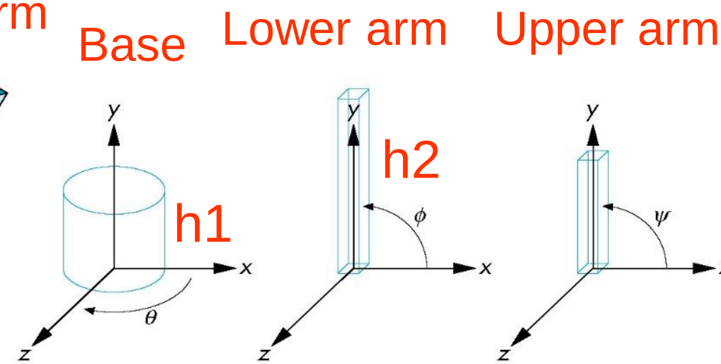


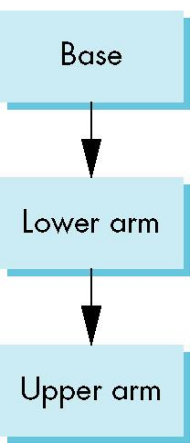




- Rotation of base: $\mathbf{R}_b$
  - Apply $\mathbf{M} = \mathbf{R}_b$ to base
- Translate lower arm relative to base: $\mathbf{T}_{lu}$
- Rotate lower arm around joint: $\mathbf{R}_{lu}$
  - Apply $\mathbf{M} = \mathbf{R}_b \mathbf{T}_{lu} \mathbf{R}_{lu}$ to lower arm
- Translate upper arm relative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

# OpenGL Code for Robot




```
robot_arm()
{
    glRotate(theta, 0.0, 1.0, 0.0);
    base();
    glTranslate(0.0, h1, 0.0);
    glRotate(phi, 0.0, 0.0 , 1.0 );
    lower_arm();
    glTranslate(0.0, h2, 0.0);
    glRotate(psi, 0.0, 0.0 , 1.0 );
    upper_arm();
}
```

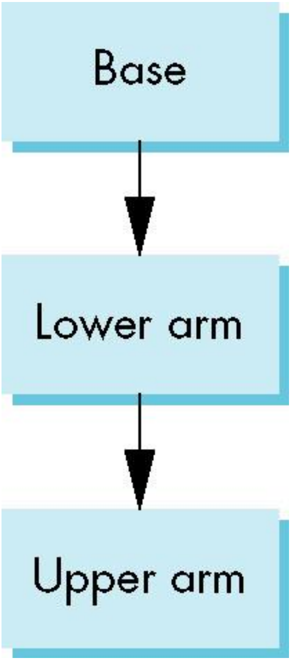
$\boxed{M = R_b}$

$\boxed{M = R_b T_{lu} R_{lu}}$

$\boxed{M = R_b T_{lu} R_{lu} T_{uu} R_{uu}}$

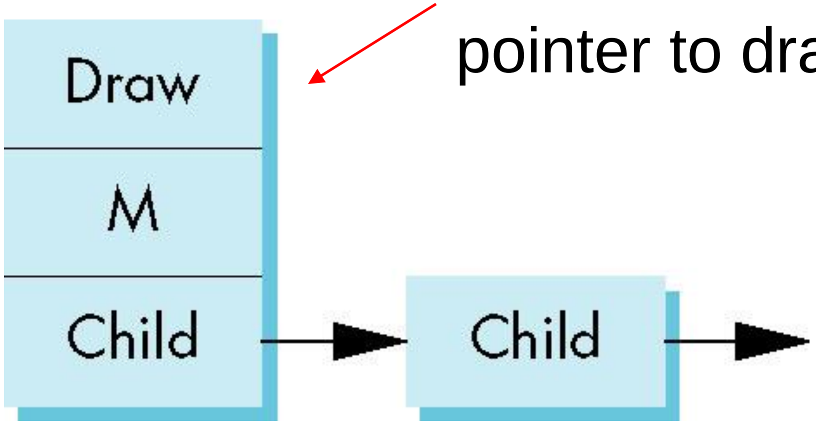
# Tree Model of Robot

- Note code shows relationships between parts of model
  - Can change "look" of parts easily without altering relationships
- Simple example of tree model
- Want a 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

# Generalizations

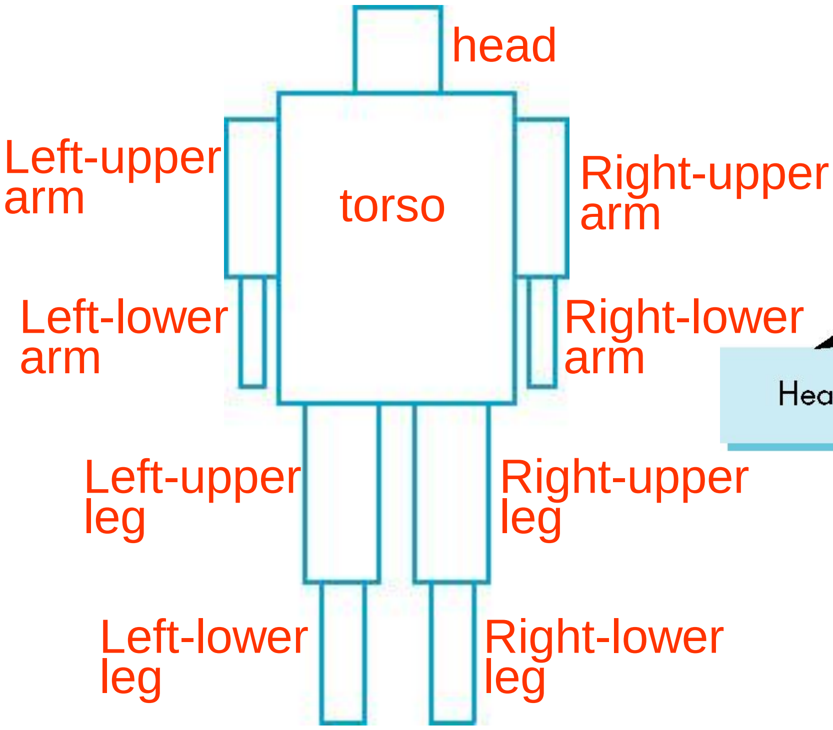
- Need to deal with multiple children
  - How do we represent a more general tree?
  - How do we traverse such a data structure?
- Animation
  - How to use dynamically?
  - Can we create and delete nodes during execution?

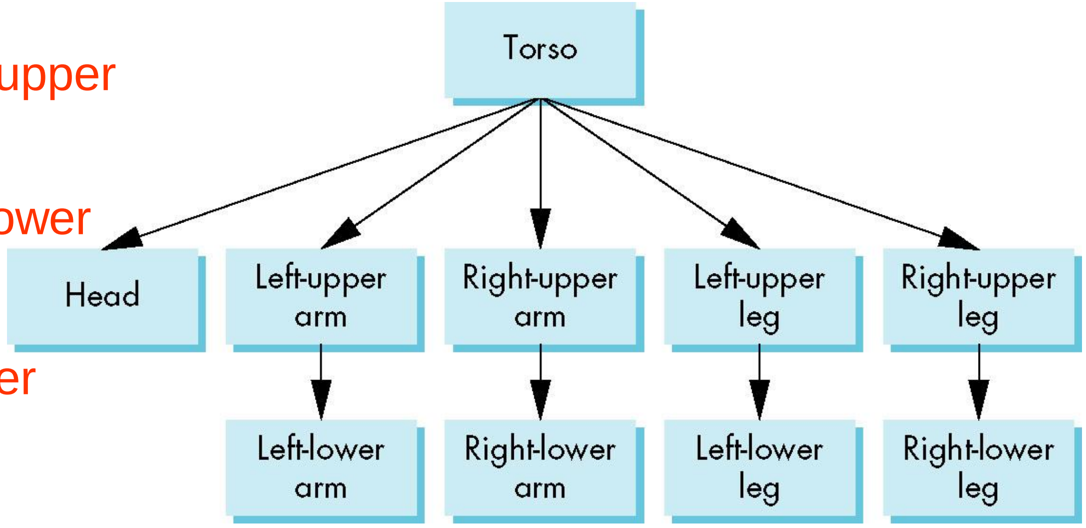
# Hierarchical Modeling II

# 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

# Humanoid Figure

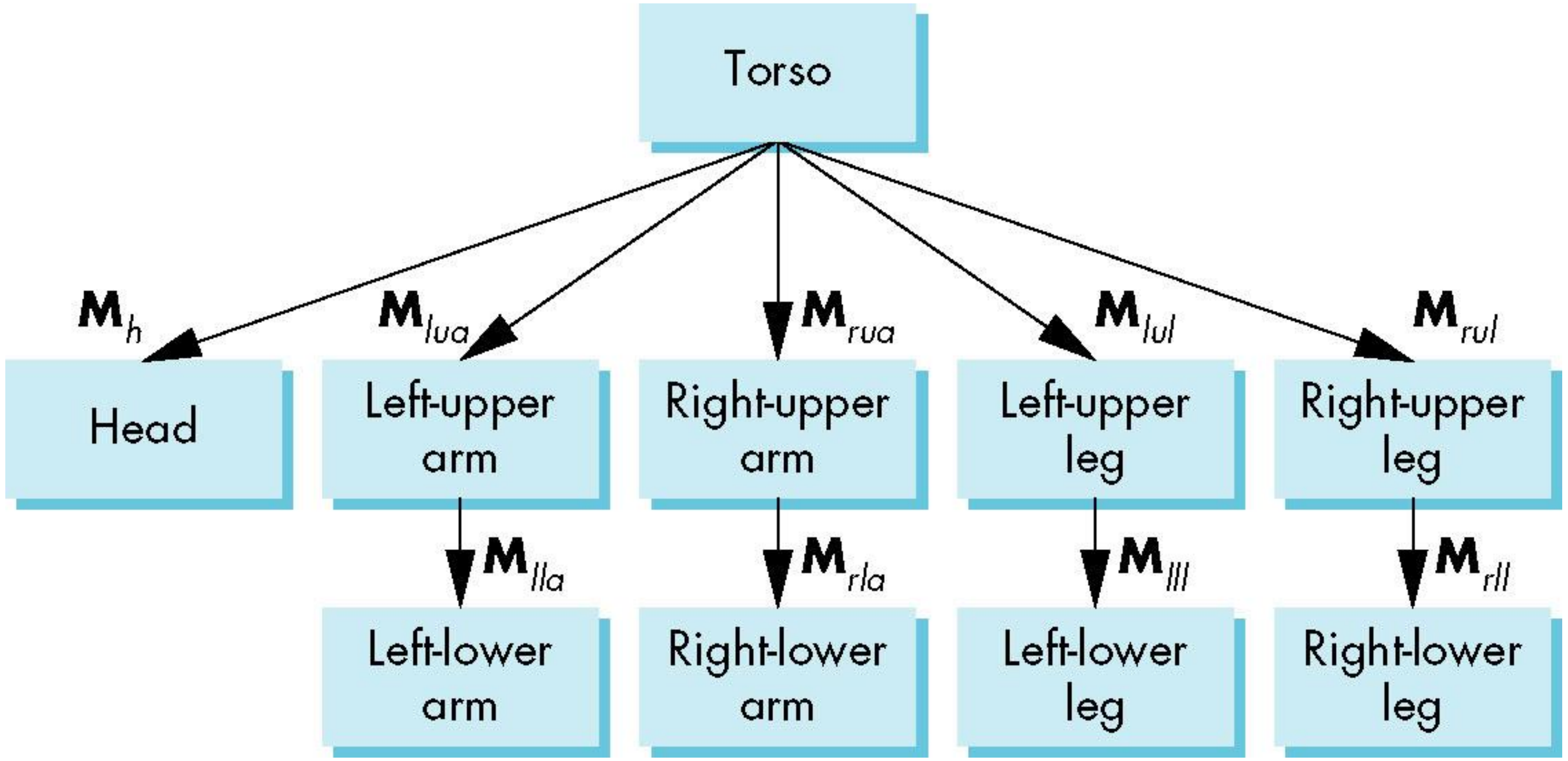


# Building the Model

- 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
  - $\mathbf{M}_{lla}$ positions left lower arm with respect to left upper arm

# Tree with Matrices

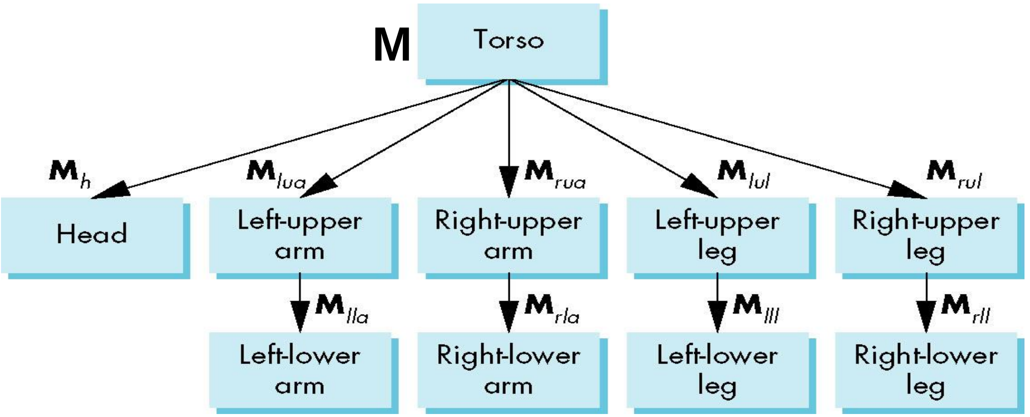


# Display and Traversal

- 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

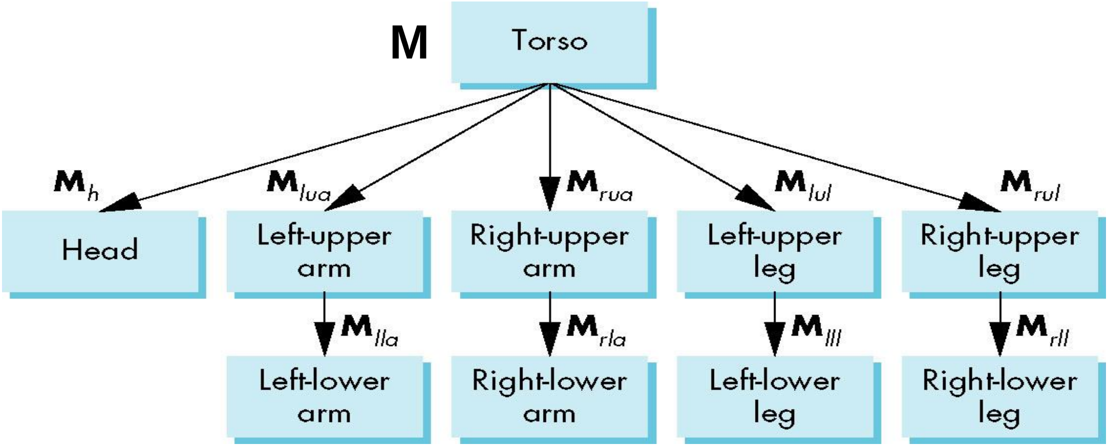
# Transformation Matrices

- 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



# Stack-based Traversal

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



# Traversal Code

```
figure() {
    glPushMatrix()
    torso();
    glRotate3f(…);
    head();
    glPopMatrix();
    glPushMatrix();
    glTranslate3f(…);
    glRotate3f(…);
    left_upper_arm();
    glPopMatrix();
    glPushMatrix();
```

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

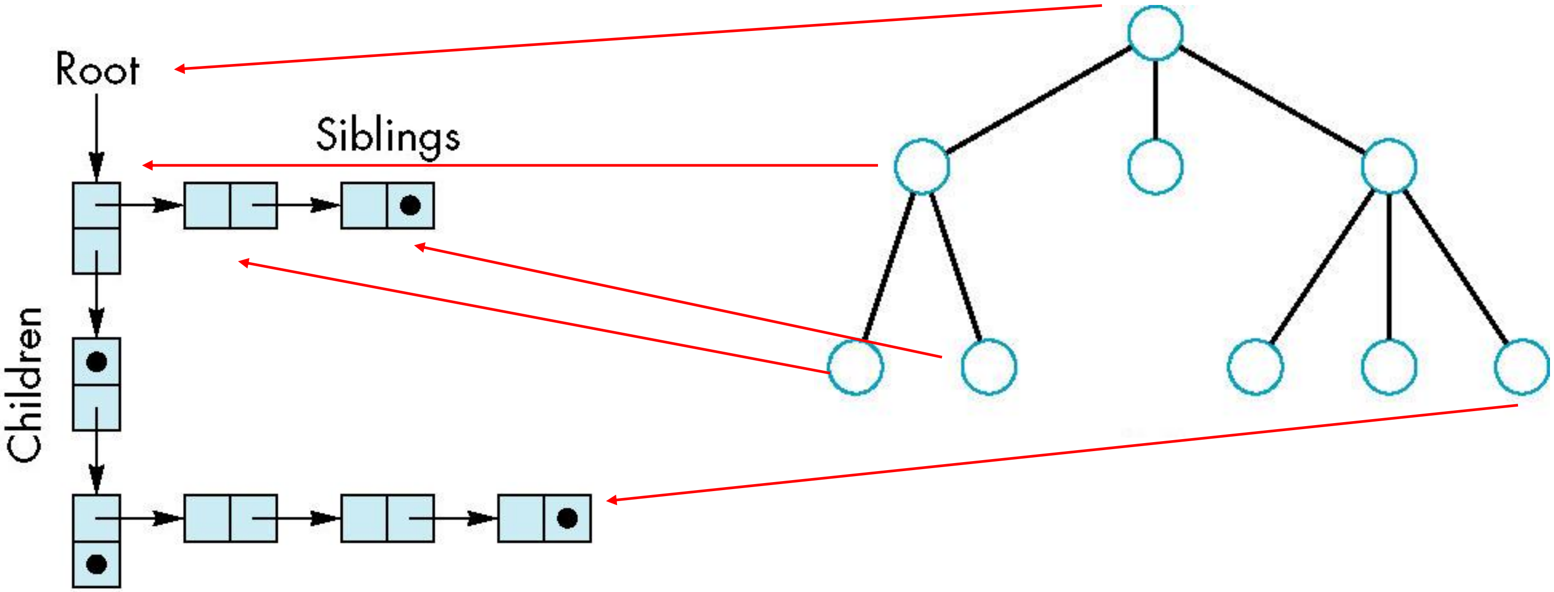
# Analysis

- 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

# General Tree Data Structure

- 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 is two pointers
  - Left: next node
  - Right: linked list of children

# Left-Child Right-Sibling Tree



# Tree node Structure

- At each node we need to store
  - Pointer to sibling
  - Pointer to child
  - Pointer to a function that draws the object represented by the node
  - Homogeneous coordinate matrix 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

# C Definition of treenode

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

# 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

# Preorder Traversal

```
void traverse(treenode *root)
{
  if(root == NULL) return;
  glPushMatrix();
  glMultMatrix(root->m);
  root->f();
  if(root->child != NULL)
     traverse(root->child);
  glPopMatrix();
  if(root->sibling != NULL)
     traverse(root->sibling);
}
```

# Notes

- We must save model-view matrix before multiplying it by node matrix
  - Updated matrix applies to children of node but not to siblings which contain their own matrices
- The traversal program applies to any left-child right-sibling tree
  - The particular tree is encoded in the definition of the individual nodes
- The order of traversal matters because of possible state changes in the functions

# Dynamic Trees

- If we use pointers, the structure can be dynamic

```
typedef treenode *tree_ptr;
tree_ptr torso_ptr;
torso_ptr = malloc(sizeof(treenode));
```

- Definition of nodes and traversal are essentially the same as before but we can add and delete nodes during execution

# 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

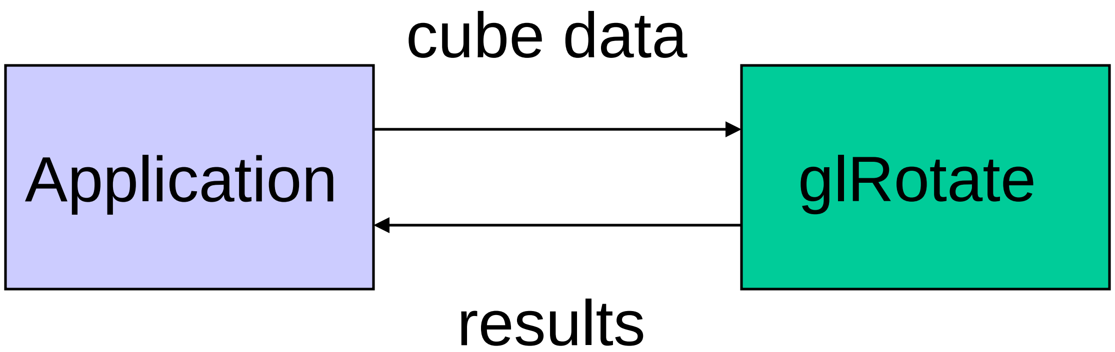
- When we define a geometric object in an application, upon execution of the code the object is passed through the pipeline
- It then disappears from the graphical system
- To redraw the object, either changed or the same, we must reexecute the code
- Display lists provide only a partial solution to 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++

- 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

- Suppose that we want to create a simple 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

- We would also like to have functions that act on the cube 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();`

# Building the Cube Object

```
class cube {
   public:
      float color[3];
      float matrix[4][4];
   // public methods

   private:

   // implementation

}
```

# 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

- 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);
```

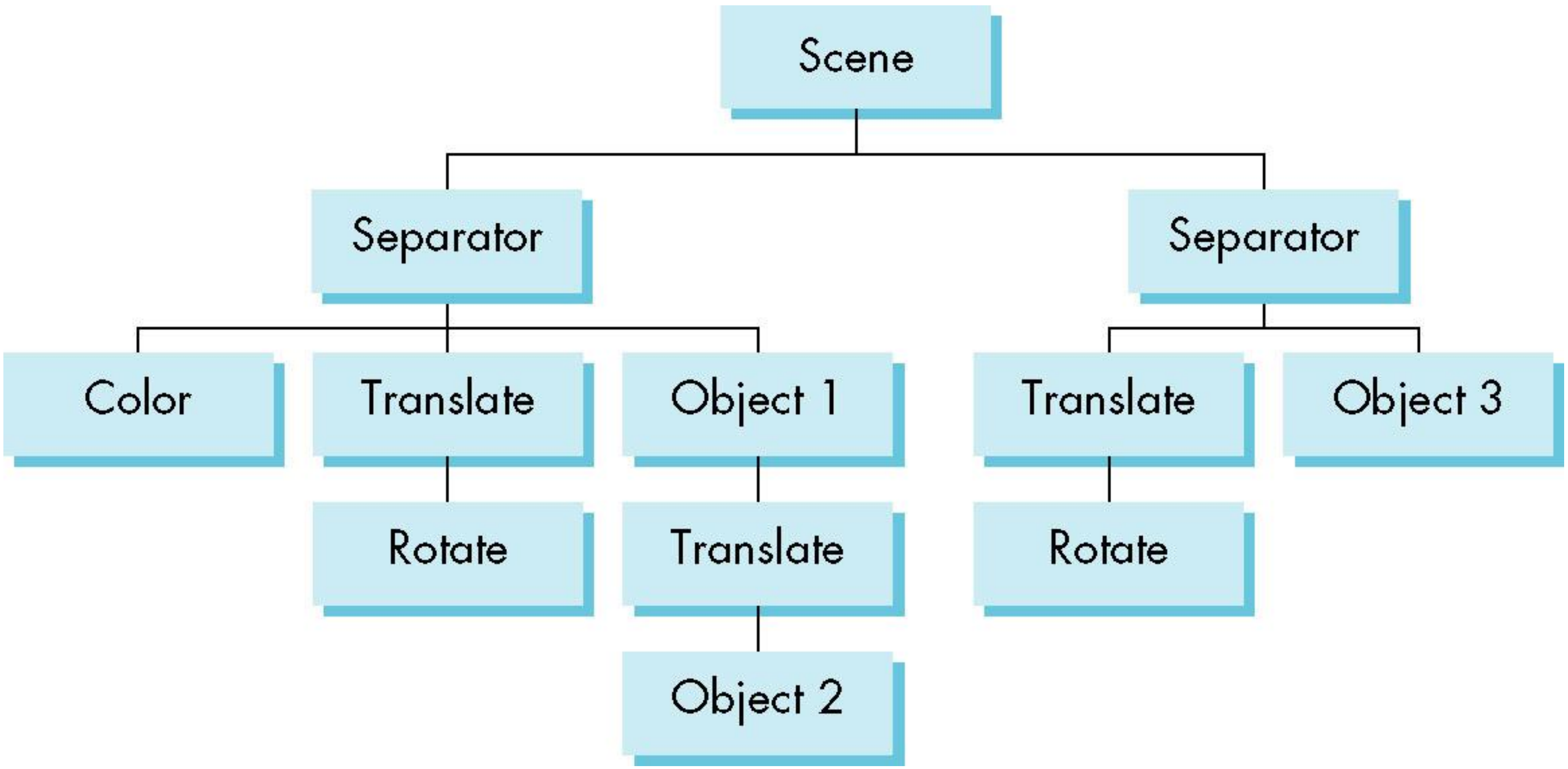
# Light Object

```
class light {     // match Phong model
  public:
   boolean type; //ortho or perspective
   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

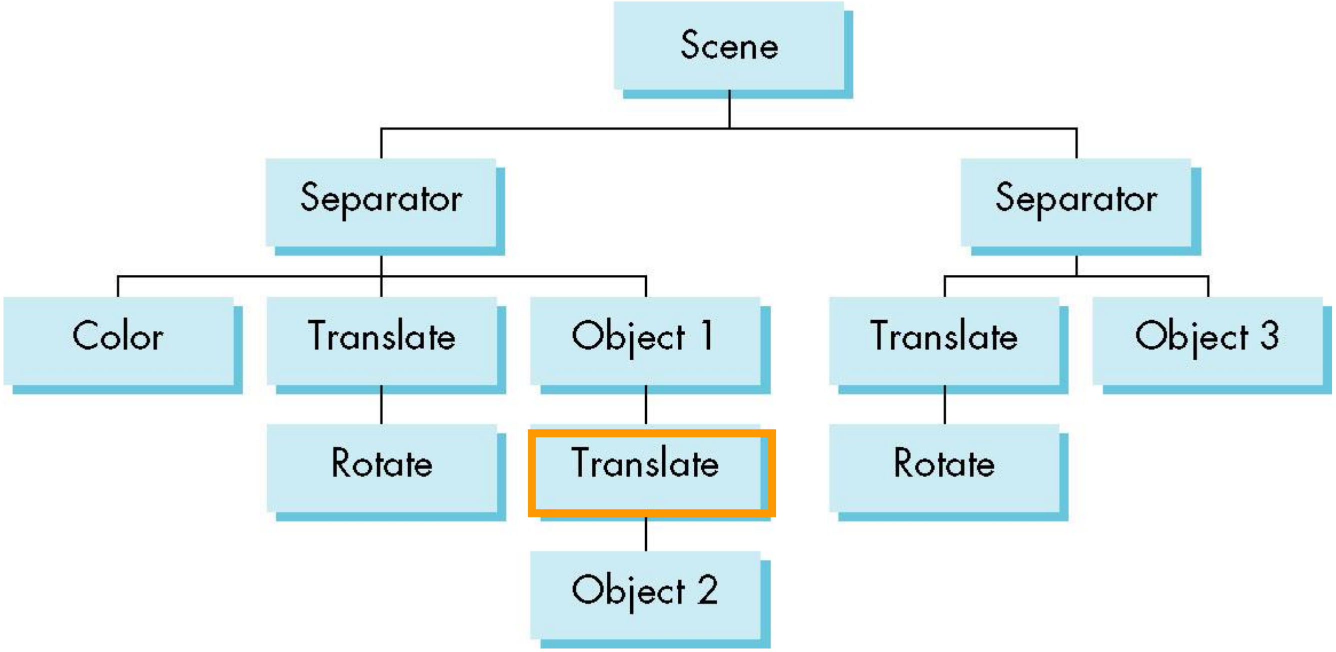
# Scene Graph



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 chages
  - 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

- 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
- <u>V</u>irtual <u>R</u>eality <u>M</u>arkup <u>L</u>anguage
    - Based on Inventor data base
    - Implemented with OpenGL

# Open Scene Graph

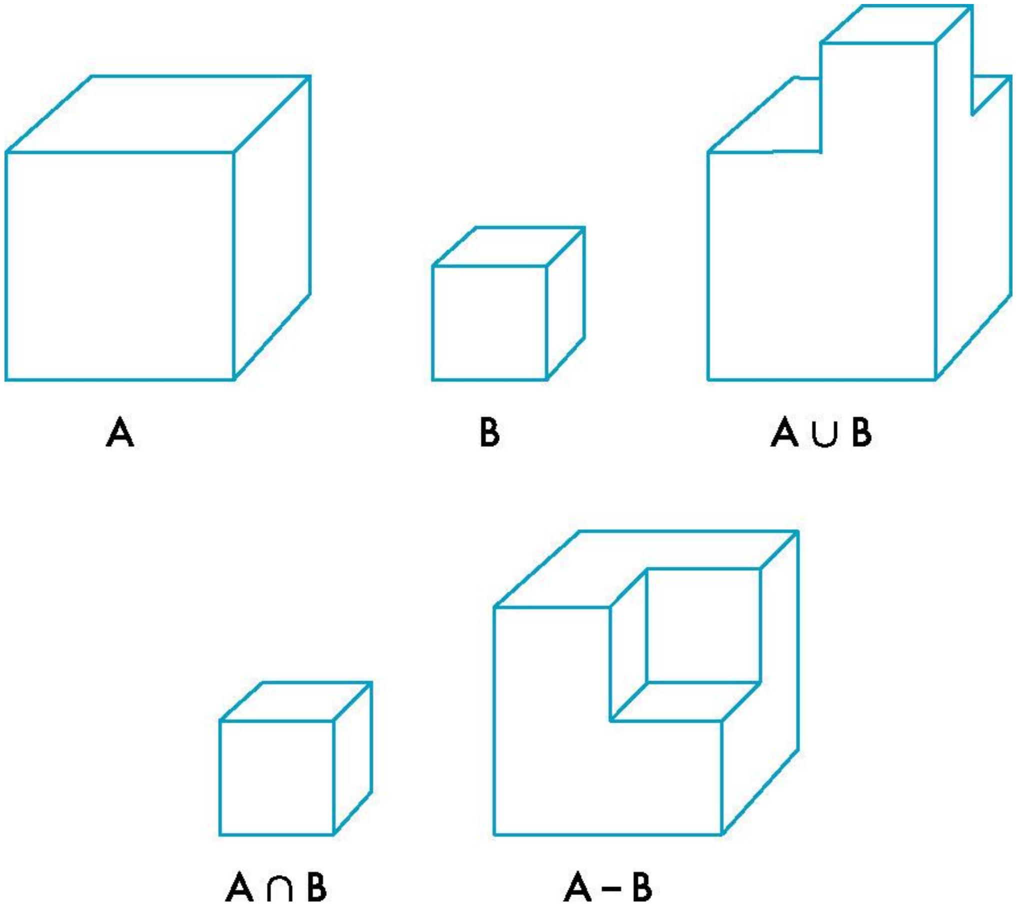
- Supports very complex geometries by adding occulusion culling in first path
- Supports translucently through a second pass that sorts the geometry
- First two passes yield a geometry list that is rendered by the pipeline in a third pass

# Other Tree Structures

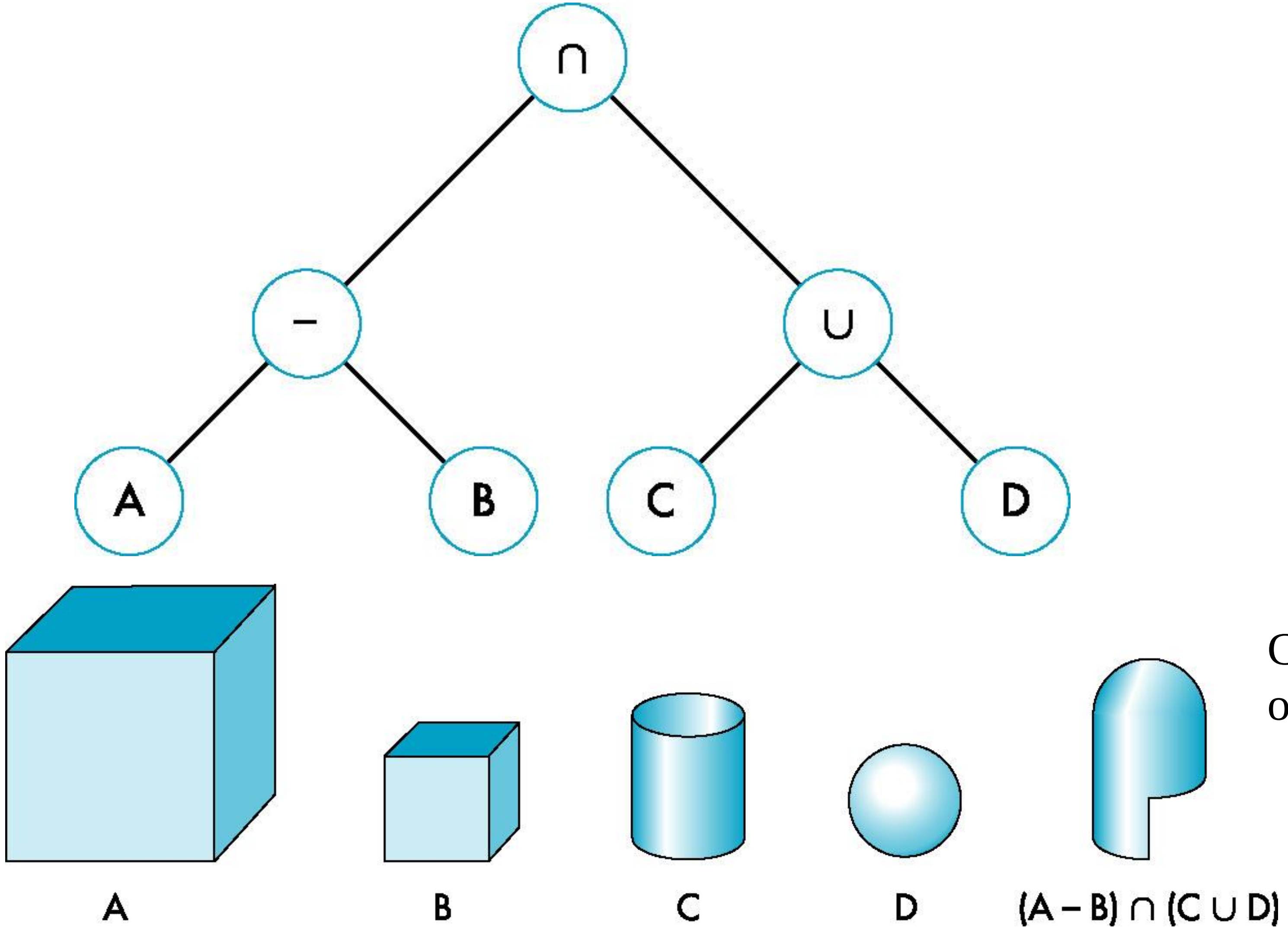
- Constructive Solid Geometry (CSG) Trees
- Binary Spatial-Partition (BSP) Trees
- Quadtrees and Octrees

# CSG Trees

- Set operators: ∩, ∪, –

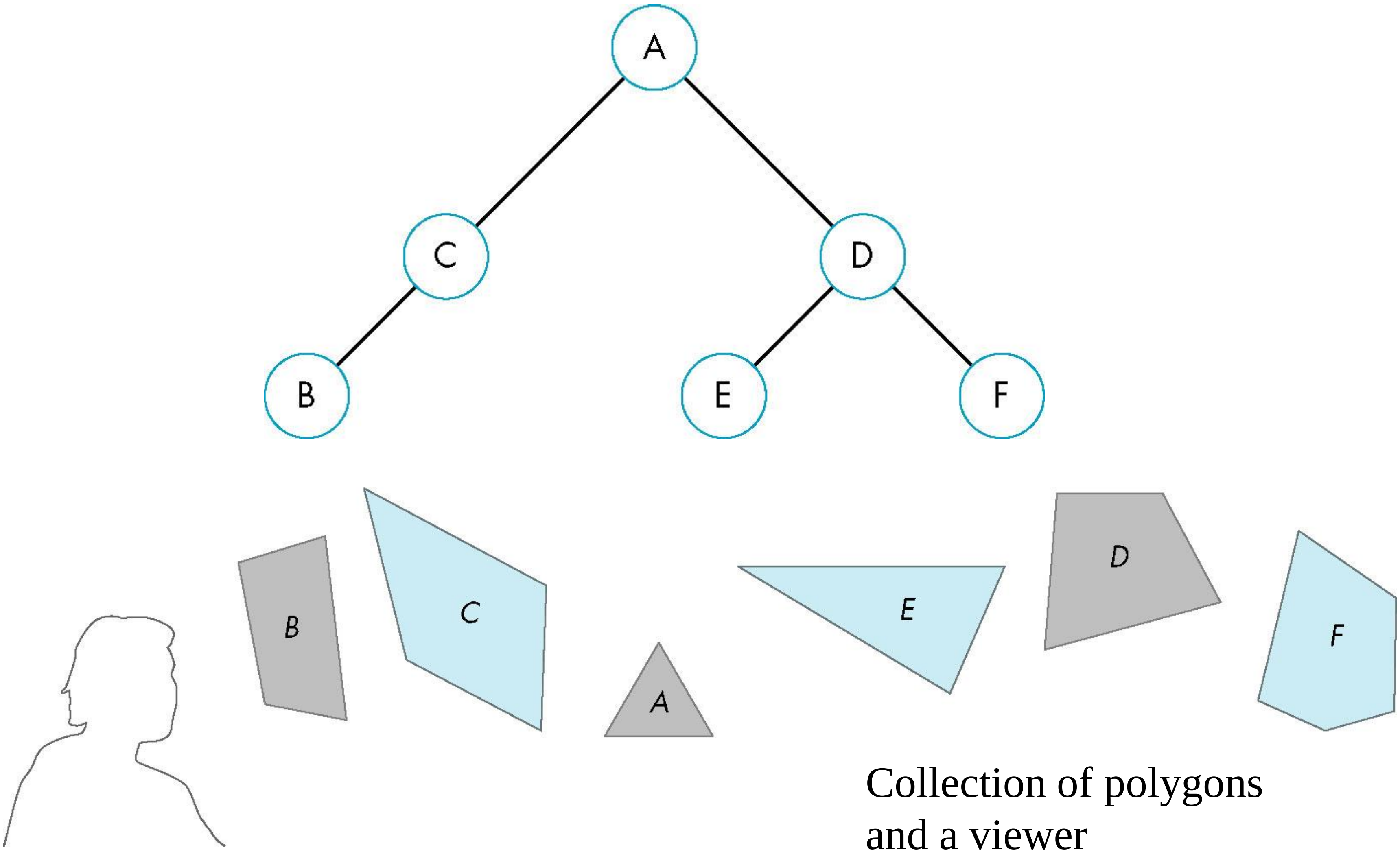


# CSG Trees



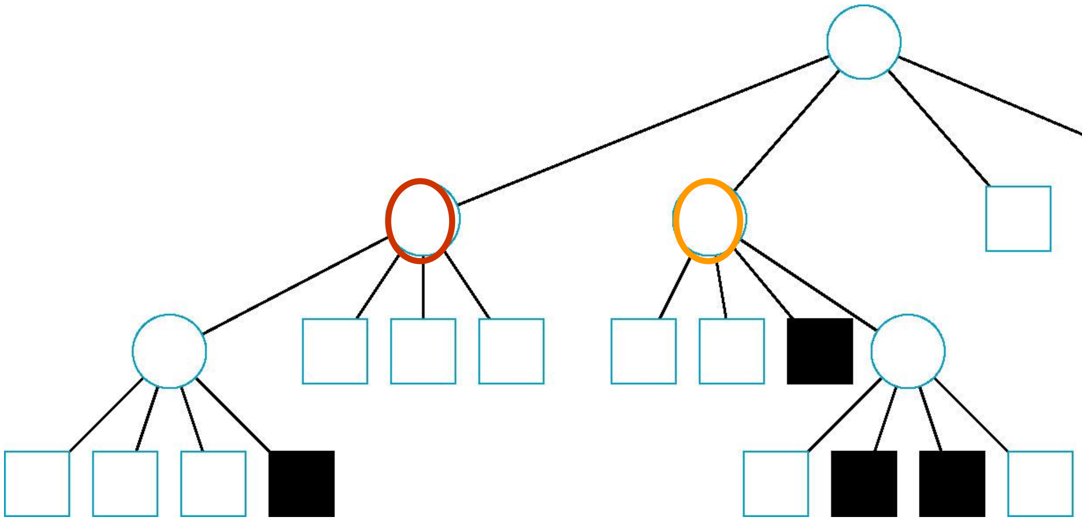

CSG object

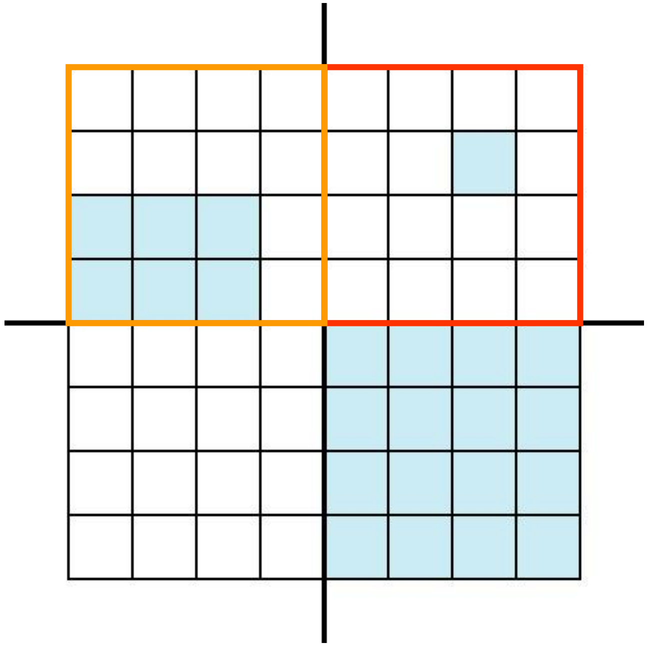
# BSP Trees




Collection of polygons
and a viewer

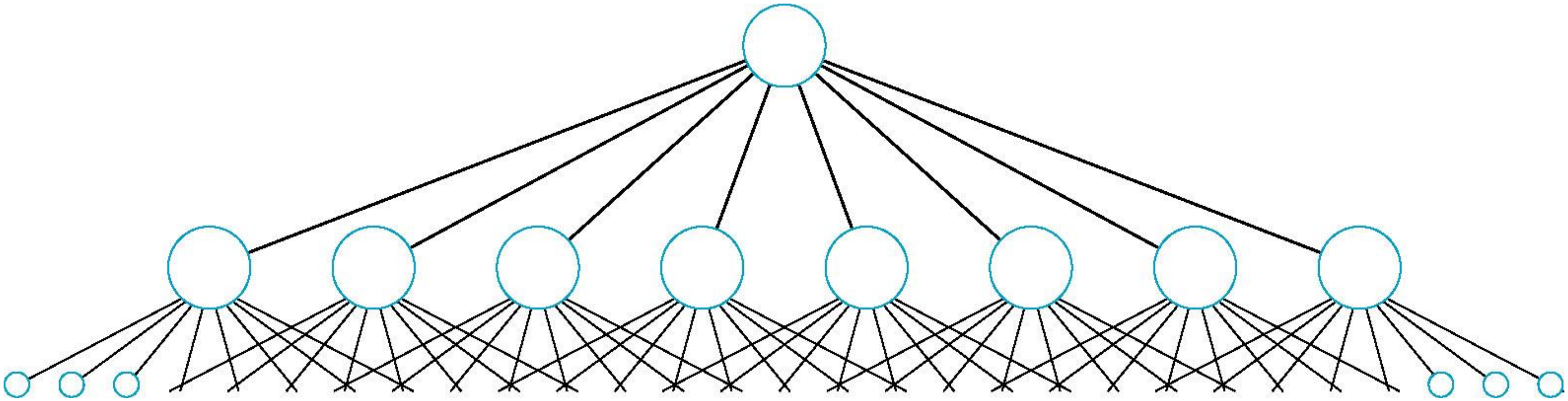
# Quadtrees



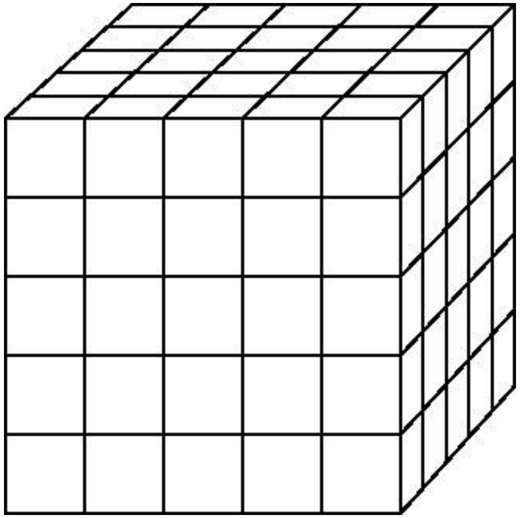


First subdivision of space

# Octrees

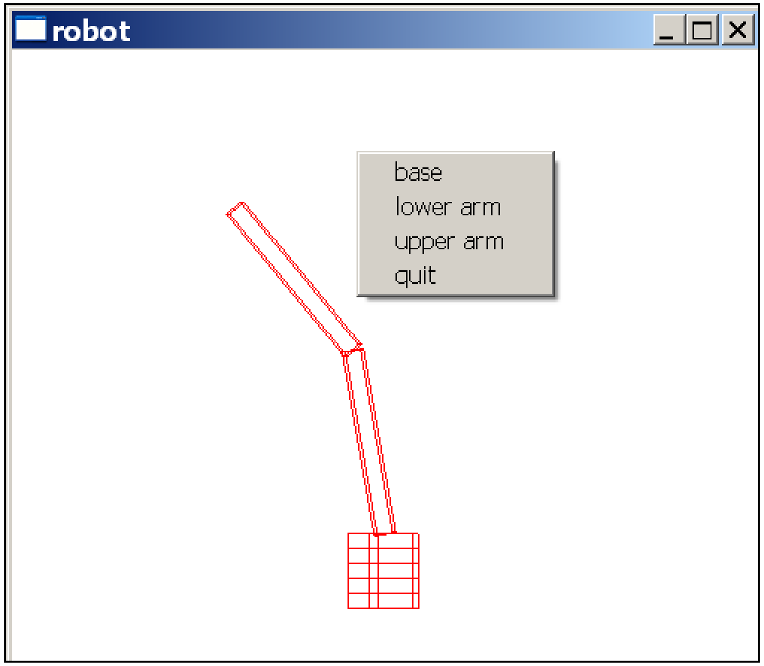


Volume data set

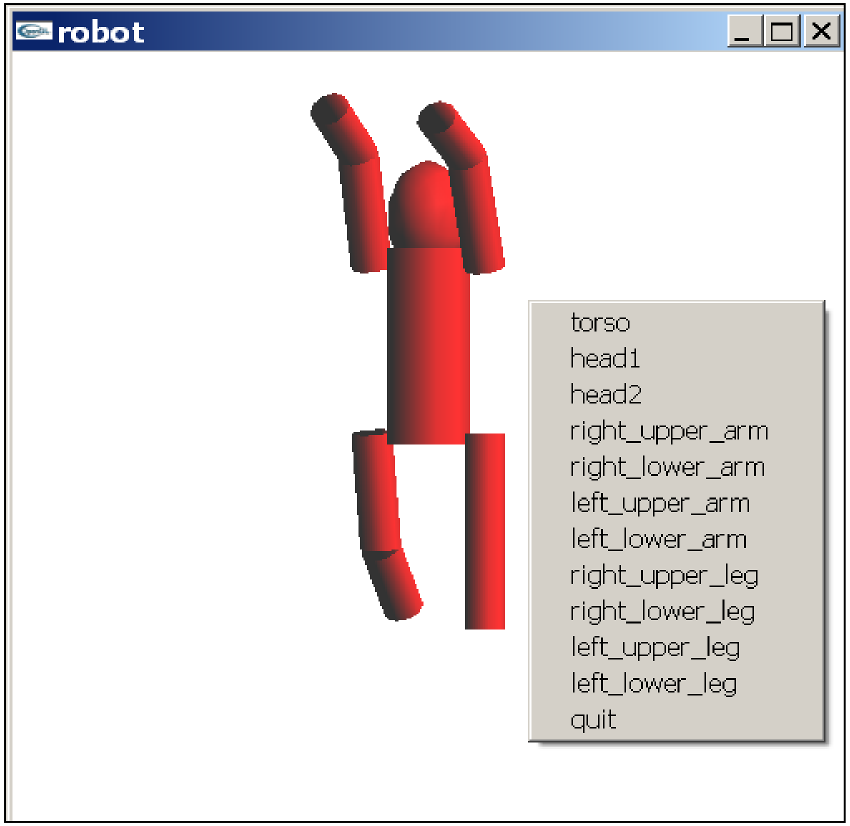


# Sample Programs

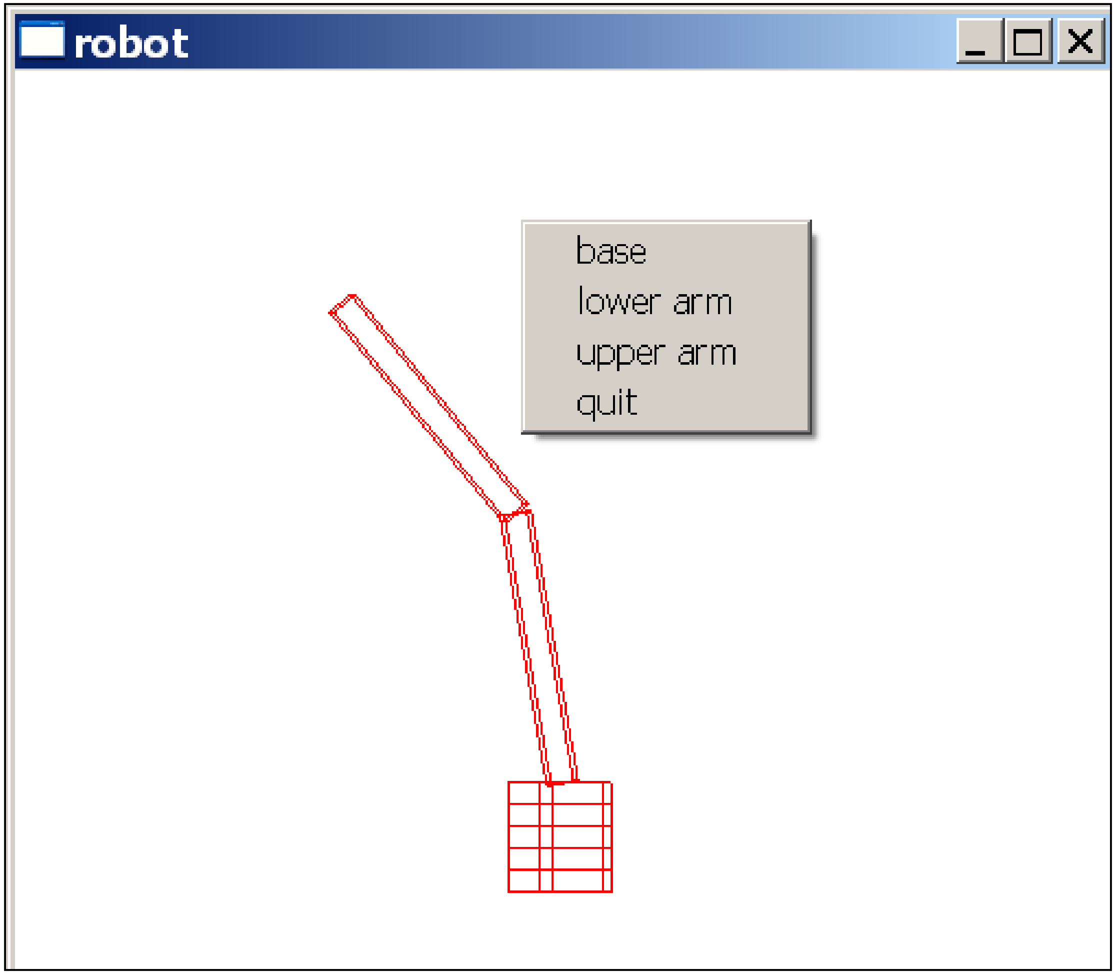
- Robot program
  - robot.c




- Figure program
  - figure.c

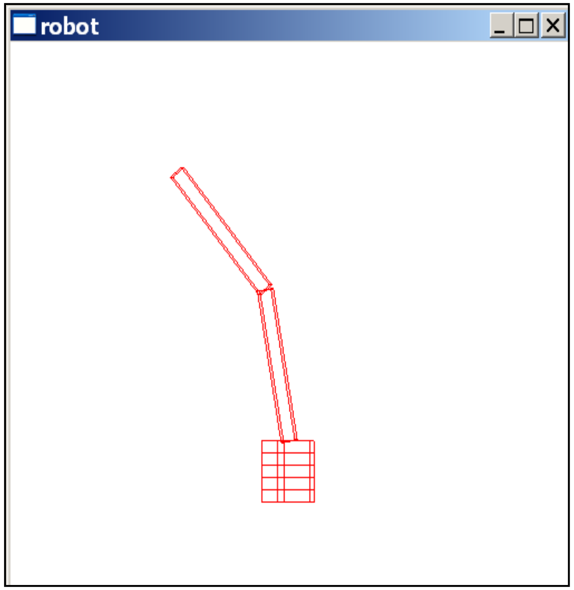


# Robot Program



```
/* Robot program (Chapter 8). Cylinder for base,
scaled cube for arms */
/* Shows use of instance transformation to define parts
(symbols) */
/* The cylinder is a quadric object from the GLU library
*/
/* The cube is also obtained from GLU */

#ifdef __APPLE__
#include <GLUT/glut.h>
#else
#include <GL/glut.h>
#endif

#include <stdlib.h>
/* Let's start using #defines so we can better
interpret the constants (and change them) */
```

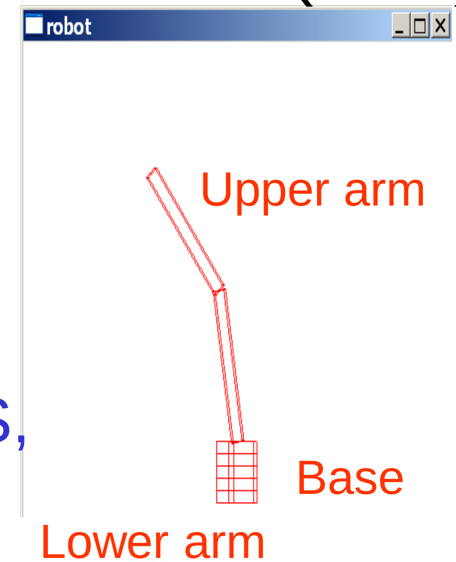
# robot.c (3/11)

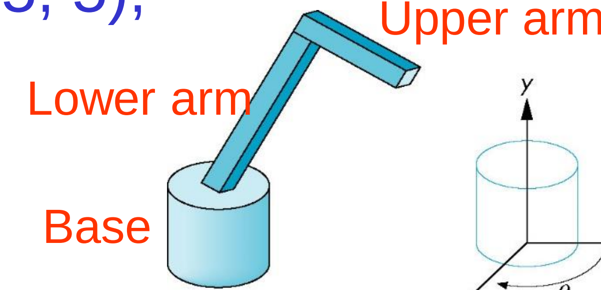


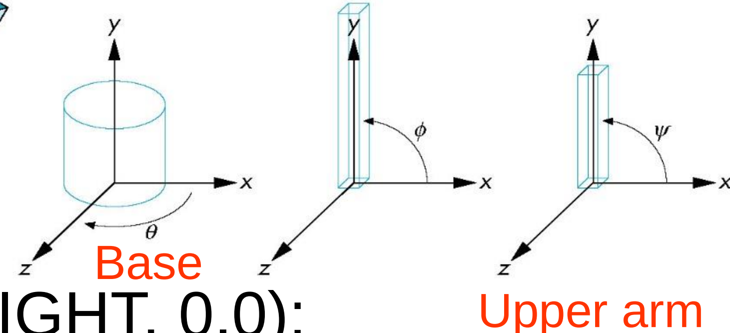

```
#define BASE_HEIGHT 2.0
#define BASE_RADIUS 1.0
#define LOWER_ARM_HEIGHT 5.0
#define LOWER_ARM_WIDTH 0.5
#define UPPER_ARM_HEIGHT 5.0
#define UPPER_ARM_WIDTH 0.5

typedef float point[3];

static GLfloat theta[] = {0.0,0.0,0.0};
static GLint axis = 0;
GLUquadricObj  *p; /* pointer to quadric object */
/* Define the three parts */
/* Note use of push/pop to return modelview matrix
to its state before functions were entered and use
rotation, translation, and scaling to create instances
of symbols (cube and cylinder */
```

# robot.c (4/11)

```
void base()
{   glPushMatrix();
/* rotate cylinder to align with y axis */
   glRotatef(-90.0, 1.0, 0.0, 0.0);
/* cyliner aligned with z axis, render with
   5 slices for base and 5 along length */
   gluCylinder(p, BASE_RADIUS, BASE_RADIUS,
             BASE_HEIGHT, 5, 5);
   glPopMatrix();
}
void upper_arm()
{  glPushMatrix();
   glTranslatef(0.0, 0.5*UPPER_ARM_HEIGHT, 0.0);
   glScalef(UPPER_ARM_WIDTH, UPPER_ARM_HEIGHT,
              UPPER_ARM_WIDTH);
   glutWireCube(1.0);
   glPopMatrix();
}
```

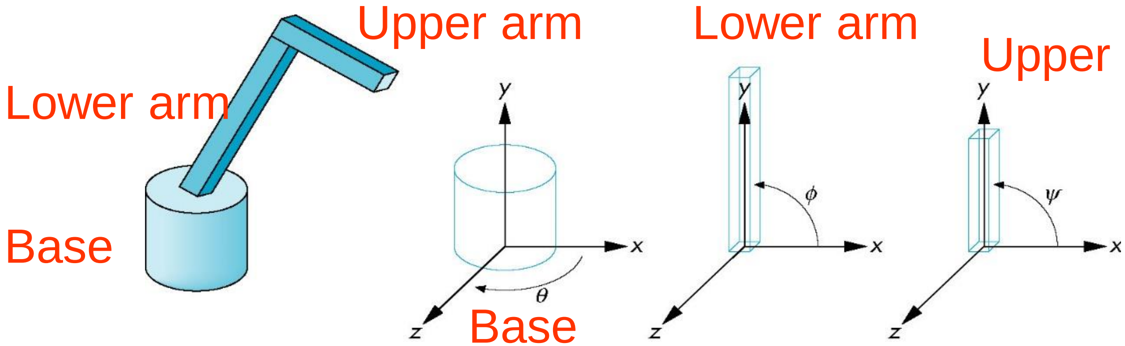


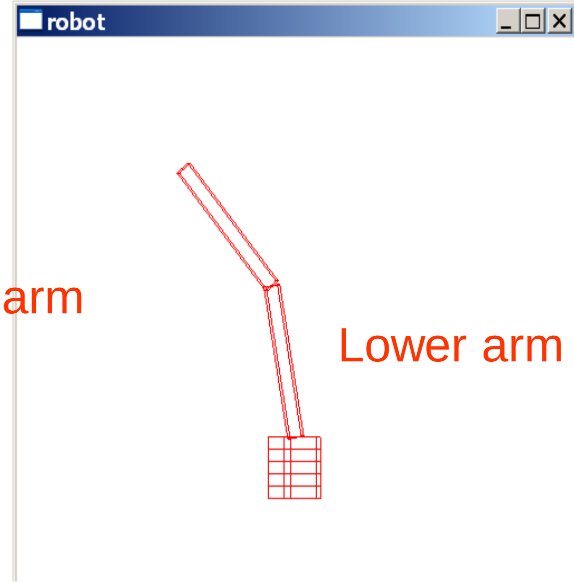



# robot.c (5/11)

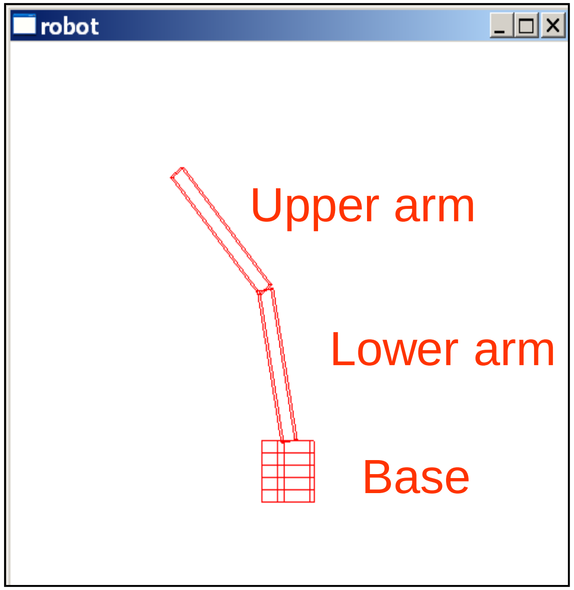
```
void lower_arm()
{
  glPushMatrix();

  glTranslatef(0.0, 0.5*LOWER_ARM_HEIGHT, 0.0);
  glScalef(LOWER_ARM_WIDTH,
              LOWER_ARM_HEIGHT, LOWER_ARM_WIDTH);
  glutWireCube(1.0);

  glPopMatrix();
}
```

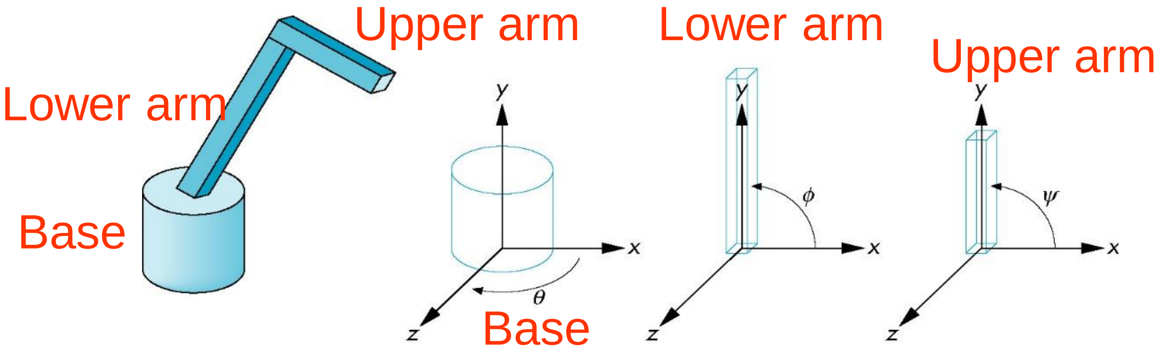


# robot.c (6/11)

```
void display(void)
{
/* Accumulate ModelView Matrix as we traverse tree */
    glClear(GL_COLOR_BUFFER_BIT);
    glLoadIdentity();
    glColor3f(1.0, 0.0, 0.0);
    glRotatef(theta[0], 0.0, 1.0, 0.0);
    base();
    glTranslatef(0.0, BASE_HEIGHT, 0.0);
    glRotatef(theta[1], 0.0, 0.0, 1.0);
    lower_arm();
    glTranslatef(0.0, LOWER_ARM_HEIGHT, 0.0);
    glRotatef(theta[2], 0.0, 0.0, 1.0);
    upper_arm();
    glFlush();
    glutSwapBuffers();
}
```



```
void mouse(int btn, int state, int x, int y)
{
/* left button increase joint angle, right button decreases it */
        if(btn==GLUT_LEFT_BUTTON &&
          state == GLUT_DOWN)
      {
      theta[axis] += 5.0;
      if( theta[axis] > 360.0 ) theta[axis] -= 360.0;
      }
        if(btn==GLUT_RIGHT_BUTTON &&
           state == GLUT_DOWN)
      {
      theta[axis] -= 5.0;
      if( theta[axis] < 360.0 ) theta[axis] += 360.0;
      }
      display();
}
```

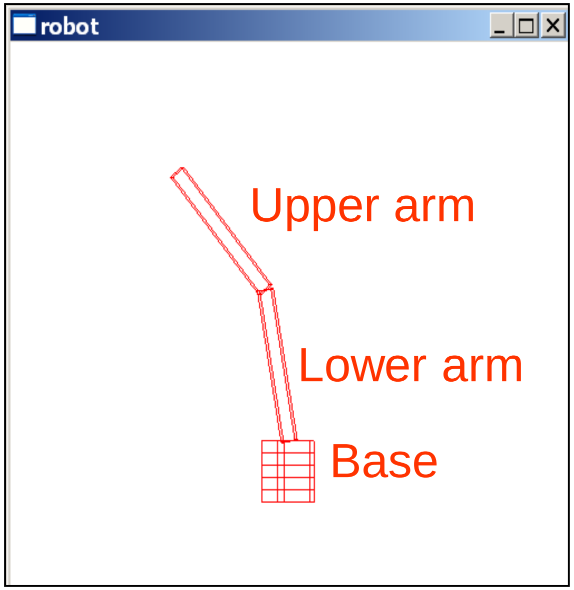
```
void menu(int id)
{

/* 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.c (9/11)

```
void
myReshape(int w, int h)
{
  glViewport(0, 0, w, h);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  if (w <= h)
     glOrtho(-10.0, 10.0, -5.0 * (GLfloat) h / (GLfloat) w,
        15.0 * (GLfloat) h / (GLfloat) w, -10.0, 10.0);
  else
     glOrtho(-10.0 * (GLfloat) w / (GLfloat) h,
        10.0 * (GLfloat) w / (GLfloat) h, -5.0, 15.0, -10.0, 10.0);
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
}
```

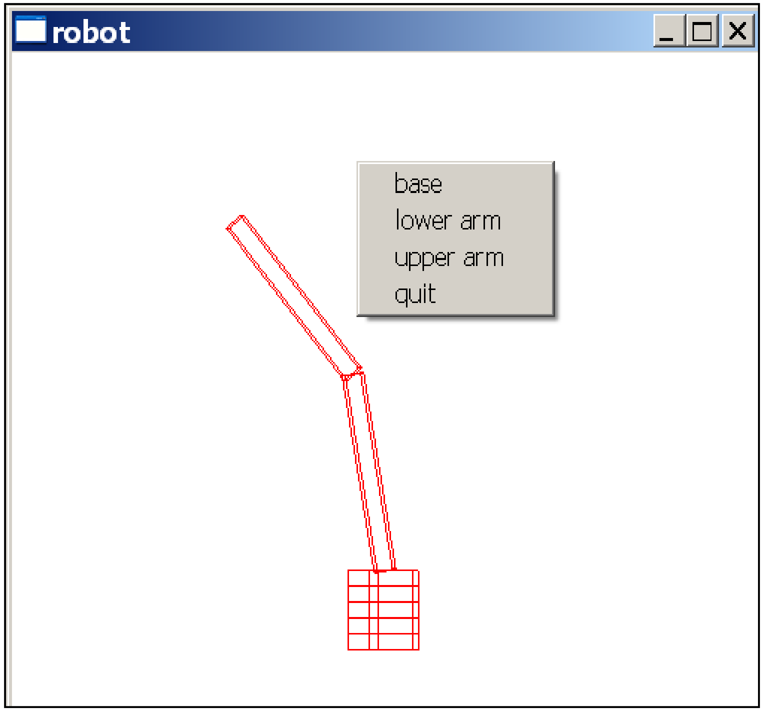
## robot.c (10/11)

```
void myinit()
{
  glClearColor(1.0, 1.0, 1.0, 1.0);
  glColor3f(1.0, 0.0, 0.0);

  p=gluNewQuadric(); /* allocate quadric object */
  gluQuadricDrawStyle(p, GLU_LINE);
     /* render it as wireframe */
}
```



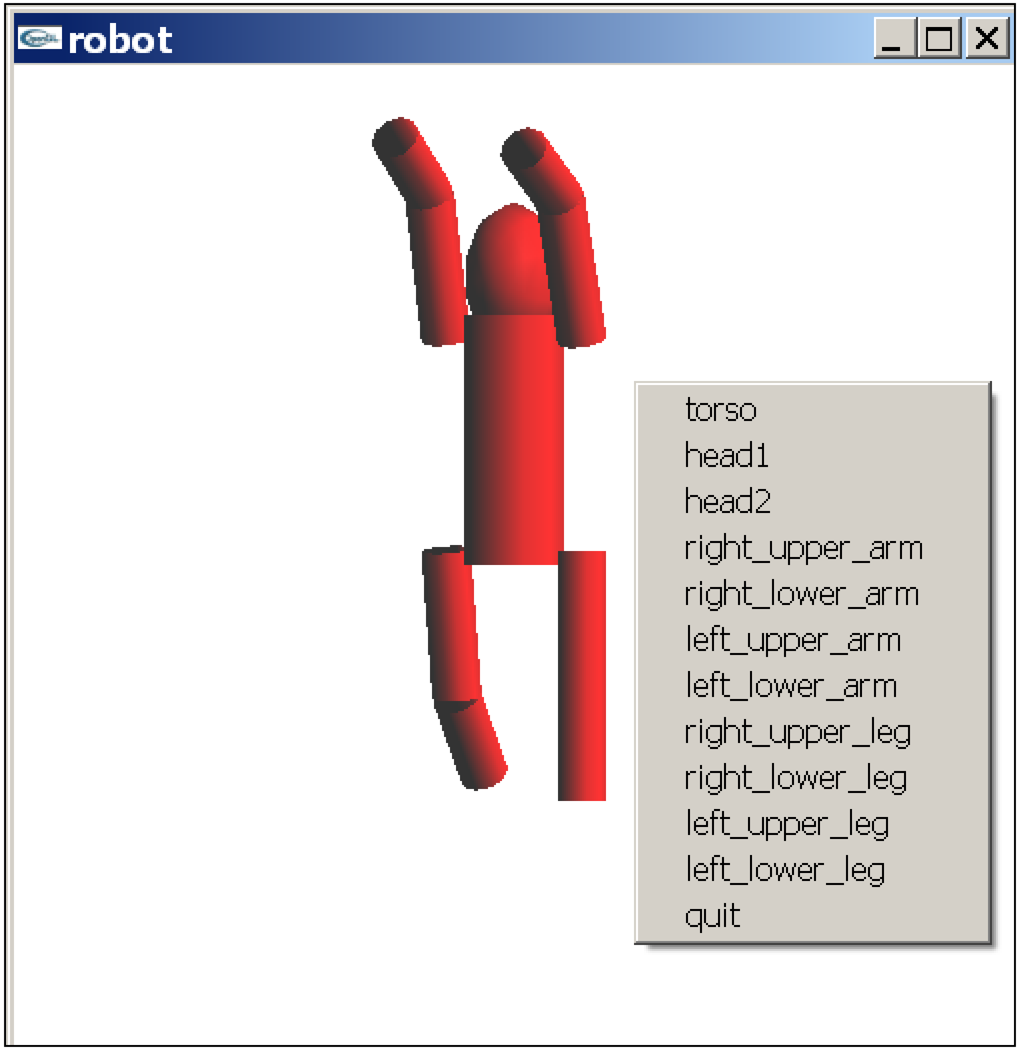
# robot.c (11/11)

```
void main(int argc, char **argv)
{   glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB |
                        GLUT_DEPTH);
    glutInitWindowSize(500, 500);
    glutCreateWindow("robot");
    myinit();
    glutReshapeFunc(myReshape);
    glutDisplayFunc(display);
    glutMouseFunc(mouse);
    glutCreateMenu(menu);
    glutAddMenuEntry("base", 1);
    glutAddMenuEntry("lower arm", 2);
    glutAddMenuEntry("upper arm", 3);
    glutAddMenuEntry("quit", 4);
    glutAttachMenu(GLUT_MIDDLE_BUTTON);
    glutMainLoop();
}
```



# Figure program

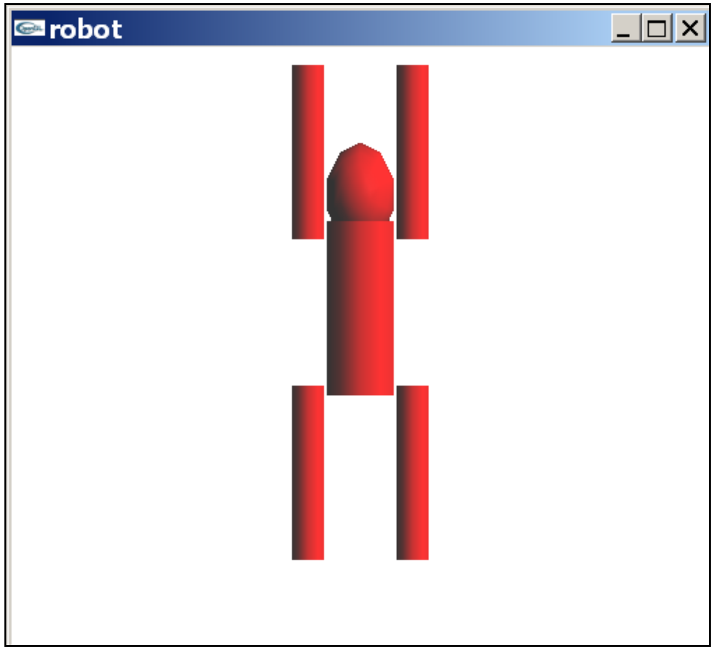
figure.c 1/22



```
/* Interactive Figure Program from Chapter 8 using cylinders (quadrics) */
/* Style similar to robot program but here we must traverse tree to display */
/* Cylinders are displayed as filled and light/material properties */
/* are set as in sphere approximation program */

#include <stdlib.h>

#ifdef __APPLE__
#include <GLUT/glut.h>
#else
#include <GL/glut.h>
#endif
```

figure.c 3/22

```
#define TORSO_HEIGHT 5.0
#define UPPER_ARM_HEIGHT  3.0
#define LOWER_ARM_HEIGHT 2.0
#define UPPER_LEG_RADIUS   0.5
#define LOWER_LEG_RADIUS  0.5
#define LOWER_LEG_HEIGHT  2.0
#define UPPER_LEG_HEIGHT   3.0
#define UPPER_LEG_RADIUS   0.5
#define TORSO_RADIUS 1.0
#define UPPER_ARM_RADIUS   0.5
#define LOWER_ARM_RADIUS  0.5
#define HEAD_HEIGHT 1.5
#define HEAD_RADIUS 1.0
```



```
typedef float point[3];

static GLfloat theta[11] = {0.0,0.0,0.0,0.0,0.0,0.0,0.0,
          180.0,0.0,180.0,0.0}; /* initial joint angles */
static GLint angle = 2;

GLUquadricObj *t, *h, *lua, *lla, *rua, *rla, *lll, *rll, *rul, *lul;

double size=1.0;

void torso()
{
  glPushMatrix();
  glRotatef(-90.0, 1.0, 0.0, 0.0);
  gluCylinder(t,TORSO_RADIUS, TORSO_RADIUS,
                   TORSO_HEIGHT,10,10);
  glPopMatrix();
}
```

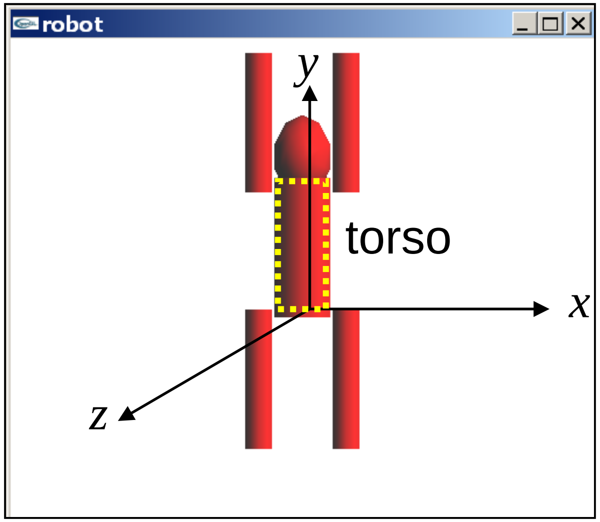


figure.c 5/22

```
void head()
{
  glPushMatrix();
  glTranslatef(0.0, 0.5*HEAD_HEIGHT,0.0);
  glScalef(HEAD_RADIUS, HEAD_HEIGHT, HEAD_RADIUS);
  gluSphere(h,1.0,10,10);
  glPopMatrix();
}

void left_upper_arm()
{
  glPushMatrix();
  glRotatef(-90.0, 1.0, 0.0, 0.0);
  gluCylinder(lua,UPPER_ARM_RADIUS,
         UPPER_ARM_RADIUS, UPPER_ARM_HEIGHT,10,10);
  glPopMatrix();
}
```

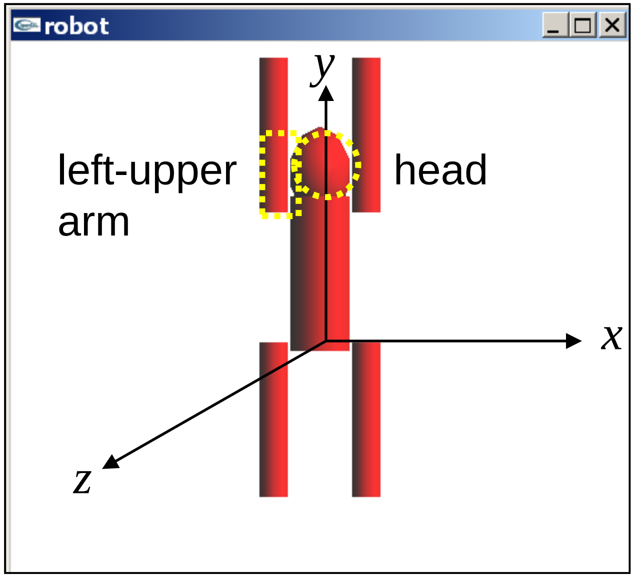


figure.c 6/22

```
void left_lower_arm()
{
  glPushMatrix();
  glRotatef(-90.0, 1.0, 0.0, 0.0);
  gluCylinder(lla,LOWER_ARM_RADIUS,
        LOWER_ARM_RADIUS, LOWER_ARM_HEIGHT,10,10);
  glPopMatrix();
}

void right_upper_arm()
{
  glPushMatrix();
  glRotatef(-90.0, 1.0, 0.0, 0.0);
  gluCylinder(rua,UPPER_ARM_RADIUS,
        UPPER_ARM_RADIUS, UPPER_ARM_HEIGHT,10,10);
  glPopMatrix();
}
```

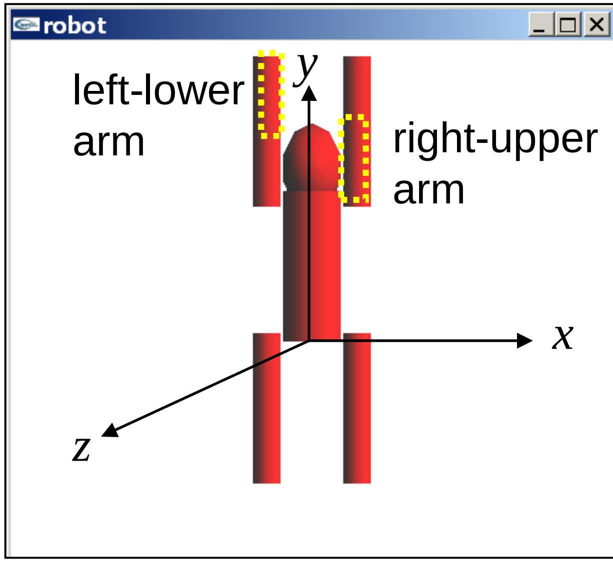


figure.c 7/22

```
void right_lower_arm()
{
  glPushMatrix();
  glRotatef(-90.0, 1.0, 0.0, 0.0);
  gluCylinder(rla,LOWER_ARM_RADIUS,
        LOWER_ARM_RADIUS, LOWER_ARM_HEIGHT,10,10);
  glPopMatrix();
}

void left_upper_leg()
{
  glPushMatrix();
  glRotatef(-90.0, 1.0, 0.0, 0.0);
  gluCylinder(lul,UPPER_LEG_RADIUS,
        UPPER_LEG_RADIUS, UPPER_LEG_HEIGHT,10,10);
  glPopMatrix();
}
```

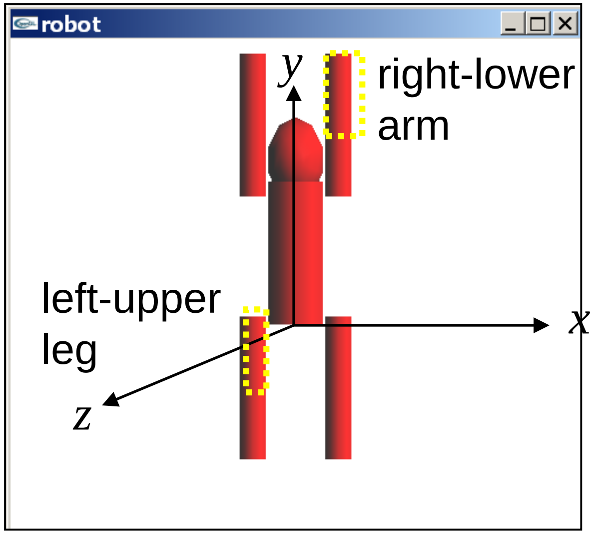


figure.c 8/22

```
void left_lower_leg()
{
  glPushMatrix();
  glRotatef(-90.0, 1.0, 0.0, 0.0);
  gluCylinder(lll,LOWER_LEG_RADIUS,
        LOWER_LEG_RADIUS, LOWER_LEG_HEIGHT,10,10);
  glPopMatrix();
}

void right_upper_leg()
{
  glPushMatrix();
  glRotatef(-90.0, 1.0, 0.0, 0.0);
  gluCylinder(rul,UPPER_LEG_RADIUS,
        UPPER_LEG_RADIUS, UPPER_LEG_HEIGHT,10,10);
  glPopMatrix();
}
```

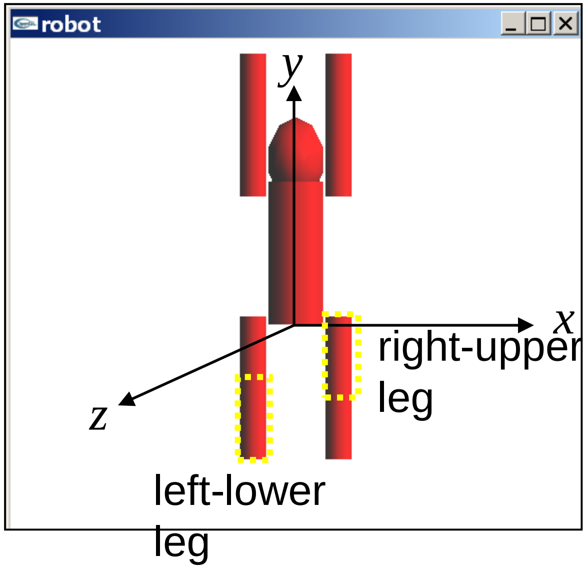
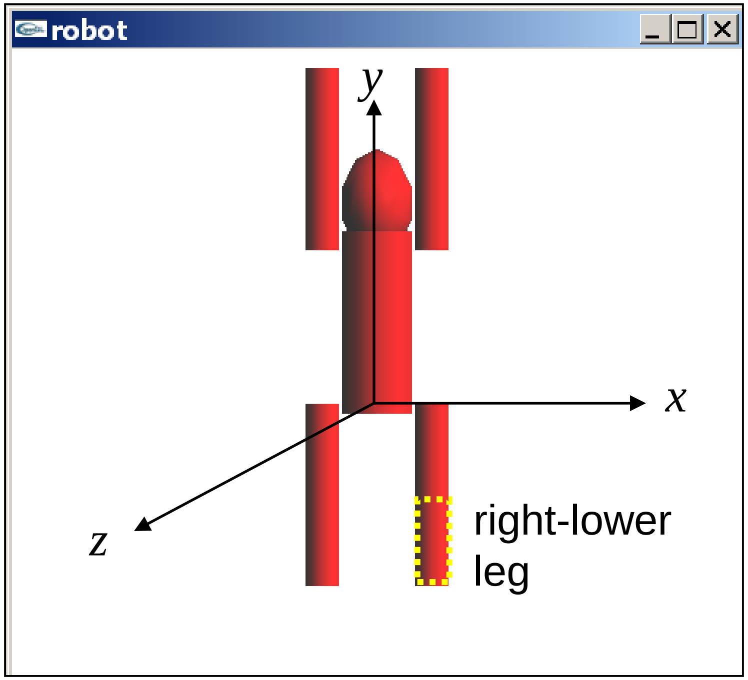
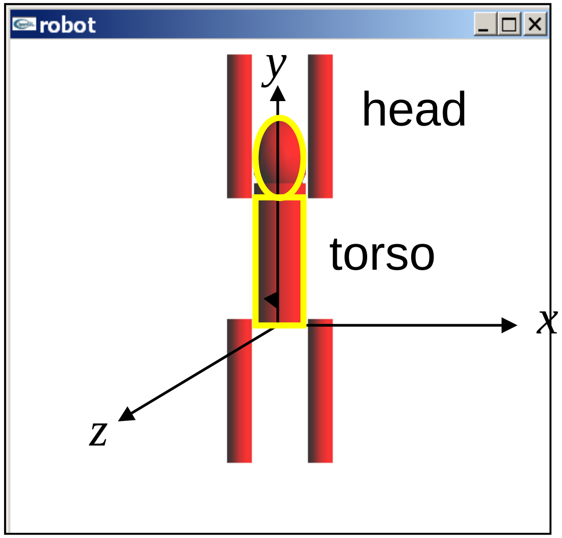


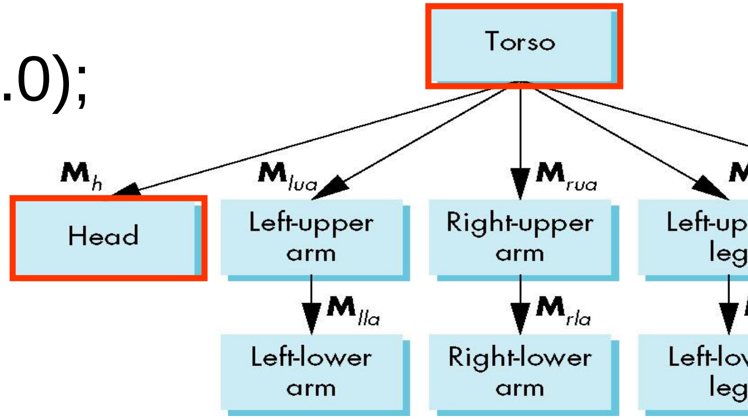
figure.c 9/22

```
void right_lower_leg()
{
  glPushMatrix();

  glRotatef(-90.0, 1.0, 0.0, 0.0);
  gluCylinder(rll,LOWER_LEG_RADIUS,
      LOWER_LEG_RADIUS, LOWER_LEG_HEIGHT,10,10);

  glPopMatrix();
}
```

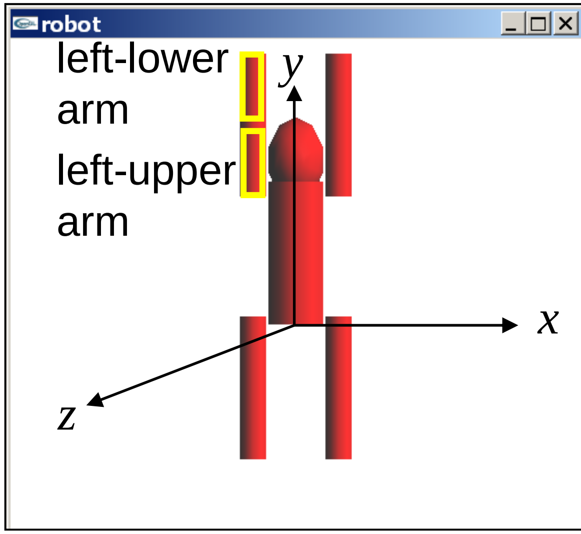


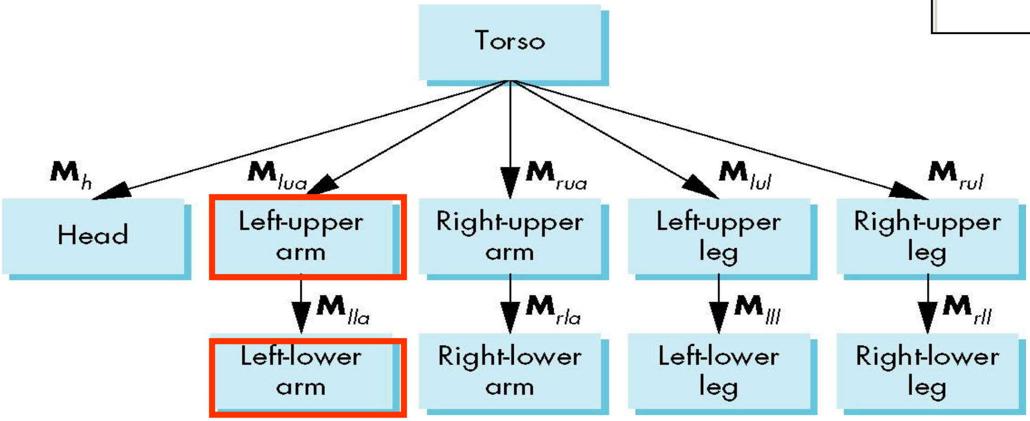
```
void  display(void)
{
 glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
 glLoadIdentity();
 glColor3f(1.0, 0.0, 0.0);

 glRotatef(theta[0], 0.0, 1.0, 0.0);
 torso();
 glPushMatrix();

 glTranslatef(0.0, TORSO_HEIGHT+0.5*HEAD_HEIGHT, 0.0);
 glRotatef(theta[1], 1.0, 0.0, 0.0);
 glRotatef(theta[2], 0.0, 1.0, 0.0);
 glTranslatef(0.0, -0.5*HEAD_HEIGHT, 0.0);
 head();

 glPopMatrix();
```



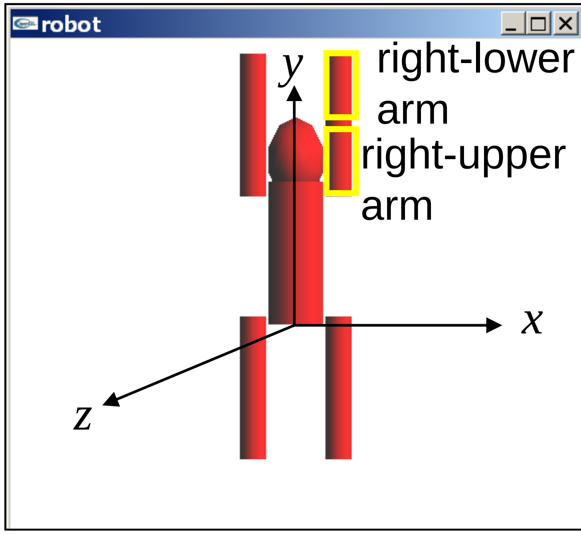



```
glPushMatrix();

glTranslatef(-(TORSO_RADIUS+UPPER_ARM_RADIUS),
             0.9*TORSO_HEIGHT, 0.0);
glRotatef(theta[3], 1.0, 0.0, 0.0);
left_upper_arm();

glTranslatef(0.0, UPPER_ARM_HEIGHT, 0.0);
glRotatef(theta[4], 1.0, 0.0, 0.0);
left_lower_arm();

glPopMatrix();
```



```
glPushMatrix();

glTranslatef(TORSO_RADIUS+UPPER_ARM_RADIUS,
             0.9*TORSO_HEIGHT, 0.0);
glRotatef(theta[5], 1.0, 0.0, 0.0);
right_upper_arm();

glTranslatef(0.0, UPPER_ARM_HEIGHT, 0.0);
glRotatef(theta[6], 1.0, 0.0, 0.0);
right_lower_arm();

glPopMatrix();
```



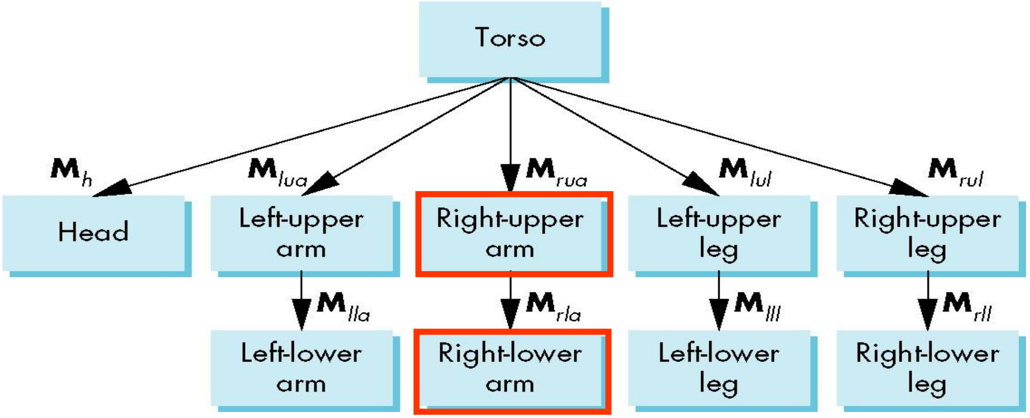
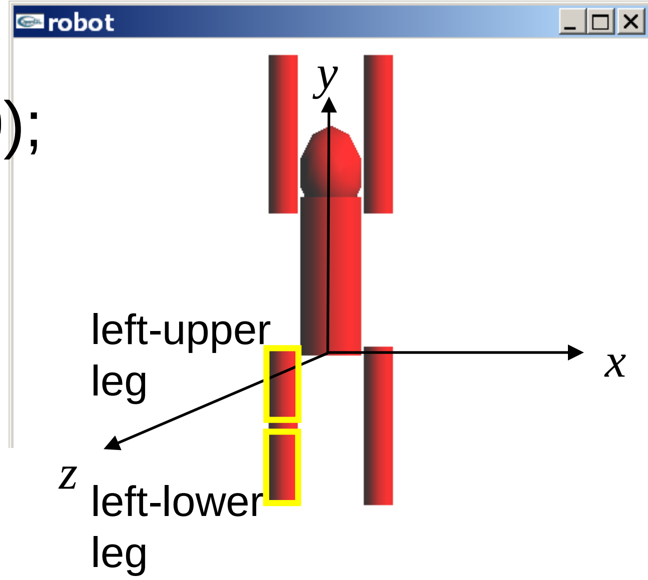
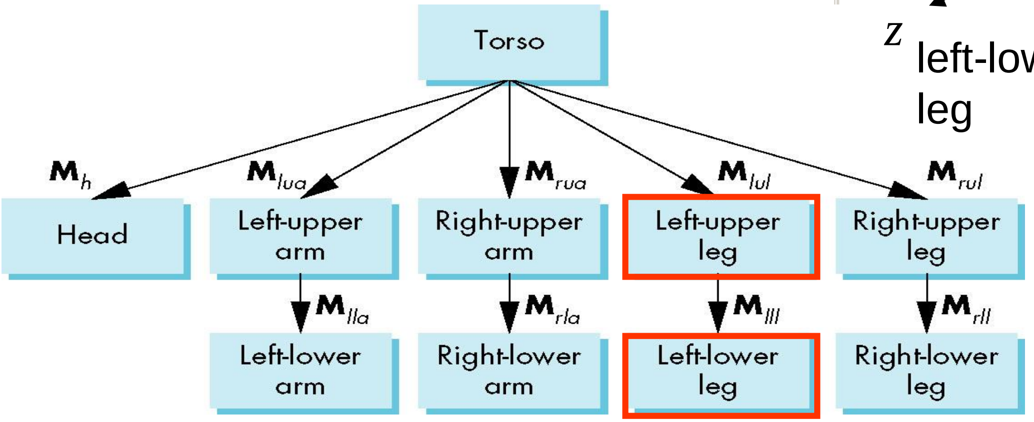



figure.c 13/22

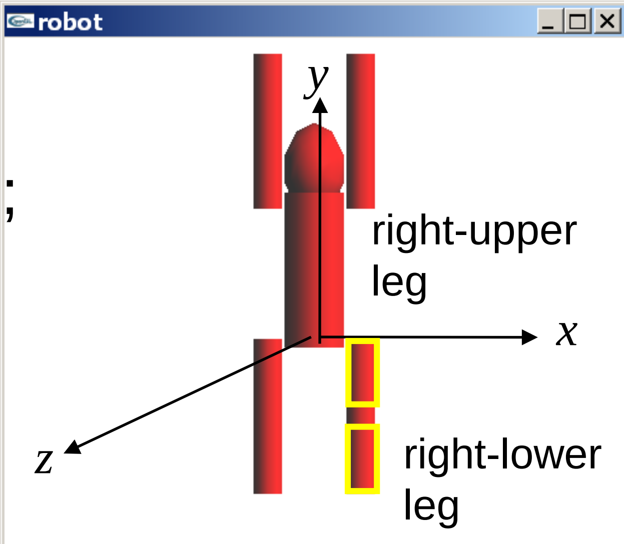
```
glPushMatrix();

glTranslatef(-(TORSO_RADIUS+UPPER_LEG_RADIUS),
                    0.1*UPPER_LEG_HEIGHT, 0.0);
glRotatef(theta[7], 1.0, 0.0, 0.0); // initial value:theta[7]=180.0
left_upper_leg();

glTranslatef(0.0, UPPER_LEG_HEIGHT, 0.0);
glRotatef(theta[8], 1.0, 0.0, 0.0);
left_lower_leg();

glPopMatrix();
```



```
glPushMatrix();
glTranslatef(TORSO_RADIUS+UPPER_LEG_RADIUS,
             0.1*UPPER_LEG_HEIGHT, 0.0);
glRotatef(theta[9], 1.0, 0.0, 0.0); // initial value: theta[9]=180.0
right_upper_leg();

glTranslatef(0.0, UPPER_LEG_HEIGHT, 0.0);
glRotatef(theta[10], 1.0, 0.0, 0.0);
right_lower_leg();

glPopMatrix();
glFlush();
glutSwapBuffers();
}
```





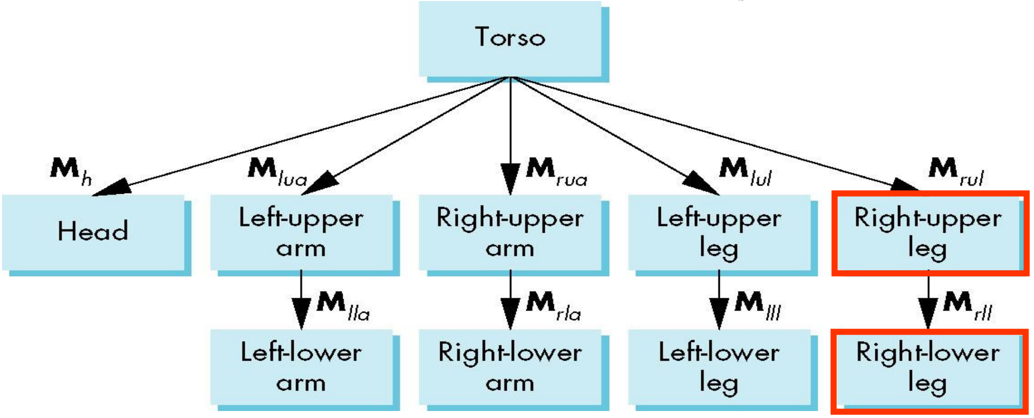



figure.c 15/22

```
void mouse(int btn, int state, int x, int y)
{
  if(btn==GLUT_LEFT_BUTTON && state == GLUT_DOWN)
   {
     theta[angle] += 5.0;
     if( theta[angle] > 360.0 ) theta[angle] -= 360.0;
    }
  if(btn==GLUT_RIGHT_BUTTON && state == GLUT_DOWN)
   {
     theta[angle] -= 5.0;
     if( theta[angle] < 360.0 ) theta[angle] += 360.0;
    }
  display();
}
```

```
void menu(int id)
{
   if(id <11 ) angle=id;
   if(id ==11 ) exit(0);
}
void  myReshape(int w, int h)
{   glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    if (w <= h)
        glOrtho(-10.0, 10.0, -10.0 * (GLfloat) h / (GLfloat) w,
                    10.0 * (GLfloat) h / (GLfloat) w, -10.0, 10.0);
    else
        glOrtho(-10.0 * (GLfloat) w / (GLfloat) h,
                10.0 * (GLfloat) w / (GLfloat) h, 0.0, 10.0, -10.0, 10.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}
```

figure.c 17/22

```
void myinit()
{
    GLfloat mat_specular[]={1.0, 1.0, 1.0, 1.0};
    GLfloat mat_diffuse[]={1.0, 1.0, 1.0, 1.0};
    GLfloat mat_ambient[]={1.0, 1.0, 1.0, 1.0};
    GLfloat mat_shininess={100.0};
    GLfloat light_ambient[]={0.0, 0.0, 0.0, 1.0};
    GLfloat light_diffuse[]={1.0, 0.0, 0.0, 1.0};
    GLfloat light_specular[]={1.0, 1.0, 1.0, 1.0};
    GLfloat light_position[]={10.0, 10.0, 10.0, 0.0};

    glLightfv(GL_LIGHT0, GL_POSITION, light_position);
    glLightfv(GL_LIGHT0, GL_AMBIENT, light_ambient);
    glLightfv(GL_LIGHT0, GL_DIFFUSE, light_diffuse);
    glLightfv(GL_LIGHT0, GL_SPECULAR, light_specular);
```

```
glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
glMaterialfv(GL_FRONT, GL_AMBIENT, mat_ambient);
glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
glMaterialf(GL_FRONT, GL_SHININESS, mat_shininess);

glShadeModel(GL_SMOOTH);
glEnable(GL_LIGHTING);
glEnable(GL_LIGHT0);
glDepthFunc(GL_LEQUAL);
glEnable(GL_DEPTH_TEST);

glClearColor(1.0, 1.0, 1.0, 1.0);
glColor3f(1.0, 0.0, 0.0);
```

figure.c 19/22

```
/* allocate quadrics with filled drawing style */
    h=gluNewQuadric();
    gluQuadricDrawStyle(h, GLU_FILL);
    t=gluNewQuadric();
    gluQuadricDrawStyle(t, GLU_FILL);
    lua=gluNewQuadric();
    gluQuadricDrawStyle(lua, GLU_FILL);
    lla=gluNewQuadric();
    gluQuadricDrawStyle(lla, GLU_FILL);
```

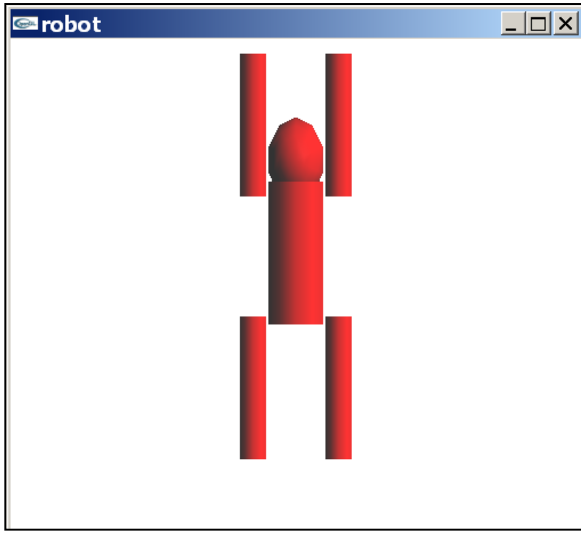


figure.c 20/22

```
    rua=gluNewQuadric();
    gluQuadricDrawStyle(rua, GLU_FILL);
    rla=gluNewQuadric();
    gluQuadricDrawStyle(rla, GLU_FILL);
    lul=gluNewQuadric();
    gluQuadricDrawStyle(lul, GLU_FILL);
    lll=gluNewQuadric();
    gluQuadricDrawStyle(lll, GLU_FILL);
    rul=gluNewQuadric();
    gluQuadricDrawStyle(rul, GLU_FILL);
    rll=gluNewQuadric();
    gluQuadricDrawStyle(rll, GLU_FILL);
}
```

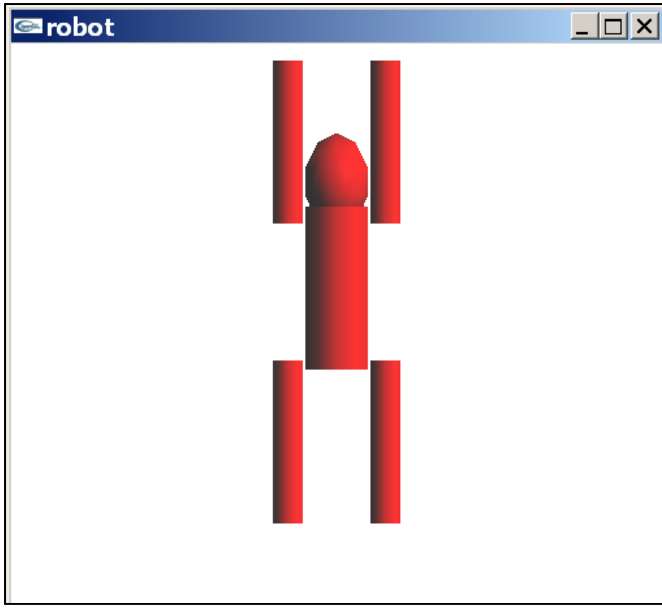


figure.c 21/22

```
void main(int argc, char **argv)
{
   glutInit(&argc, argv);
   glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB |
                               GLUT_DEPTH);
   glutInitWindowSize(500, 500);
   glutCreateWindow("robot");
   myinit();
   glutReshapeFunc(myReshape);
   glutDisplayFunc(display);
   glutMouseFunc(mouse);
```

figure.c 22/22

```
glutCreateMenu(menu);
 glutAddMenuEntry("torso", 0);
 glutAddMenuEntry("head1", 1);
 glutAddMenuEntry("head2", 2);
 glutAddMenuEntry("right_upper_arm", 3);
 glutAddMenuEntry("right_lower_arm", 4);
 glutAddMenuEntry("left_upper_arm", 5);
 glutAddMenuEntry("left_lower_arm", 6);
 glutAddMenuEntry("right_upper_leg", 7);
 glutAddMenuEntry("right_lower_leg", 8);
 glutAddMenuEntry("left_upper_leg", 9);
 glutAddMenuEntry("left_lower_leg", 10);
 glutAddMenuEntry("quit", 11);
 glutAttachMenu(GLUT_MIDDLE_BUTTON);

 glutMainLoop();
}
```