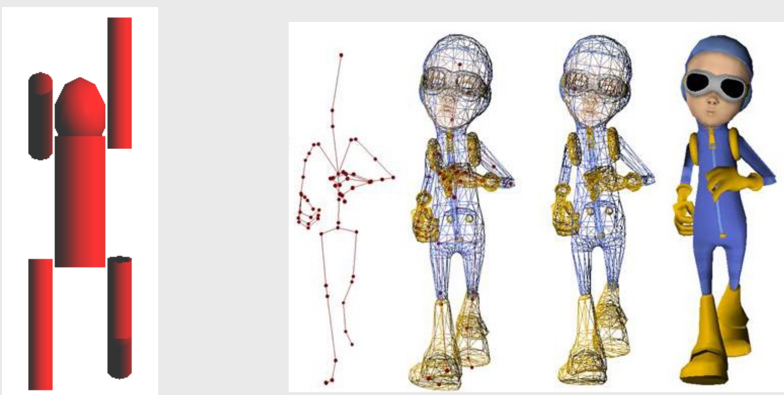
Comp 410/510

Computer Graphics
Spring 2023

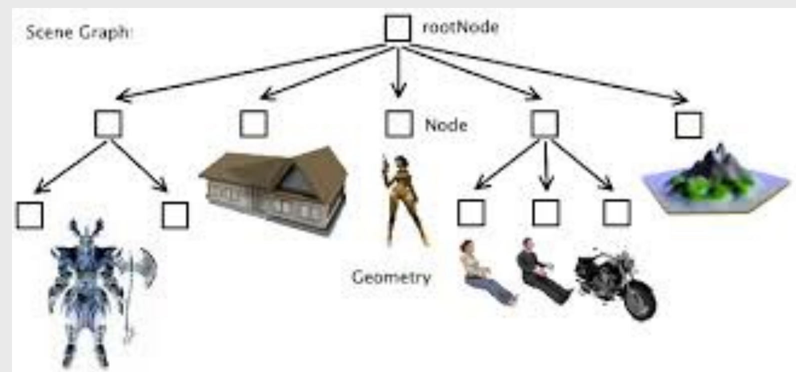
Hierarchical Modeling & Scene Graphs

Objective

- Introduce hierarchical models to represent object models and scenes in computer graphics

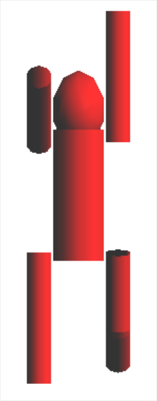


Articulated objects, e.g., robots, virtual characters

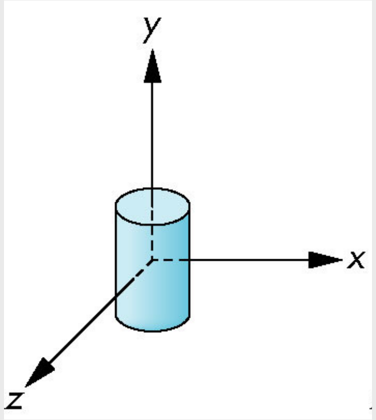


Scene graphs

Modeling Parts of an Hierarchical Model

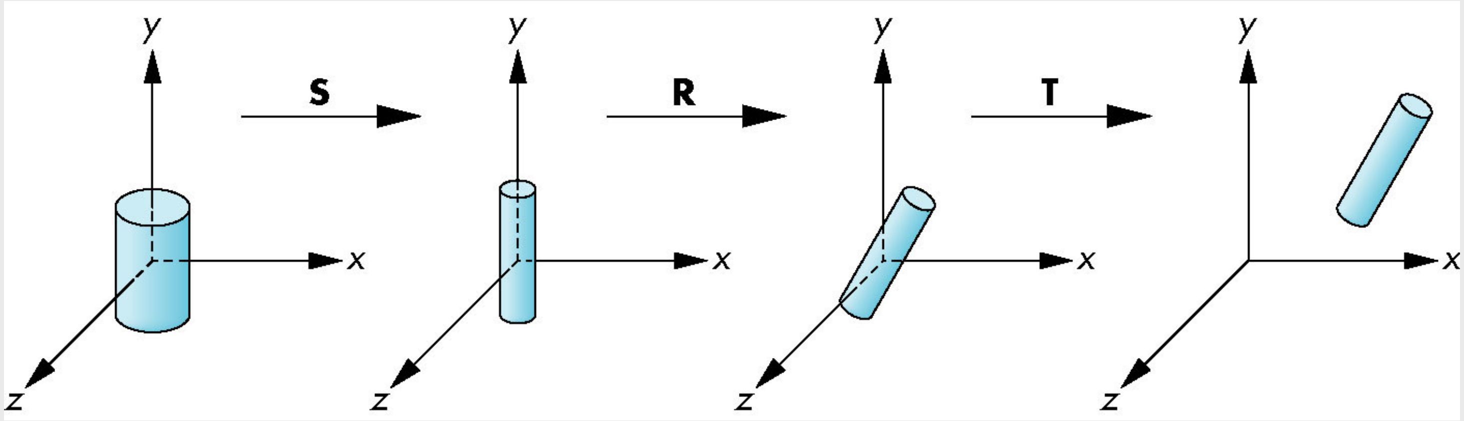


- Start with a prototype object (a symbol)



Modeling with Instance Transformation

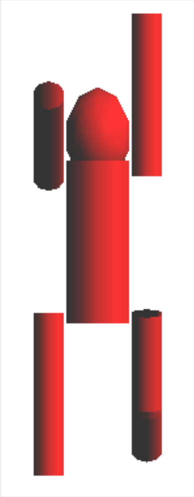
- Start with a prototype object (a symbol)
- Each appearance of the object in the scene is then an instance of the symbol
  - Must scale, orient, position
  - Defines instance transformation



Symbol-Instance Table

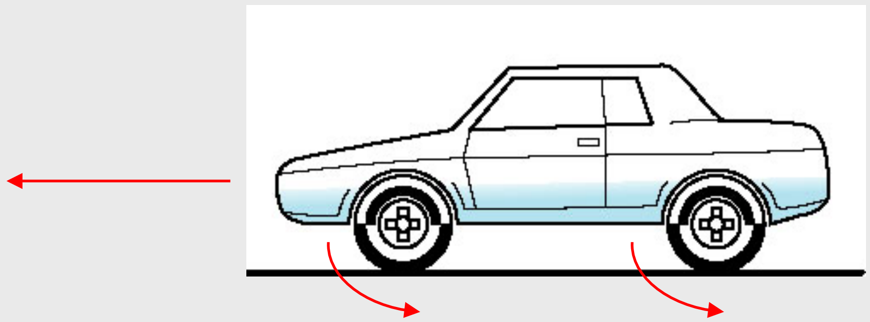
Can store the whole model by assigning a number to each symbol and storing the parameters for the instance transformation

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

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



- Speed of the car is actually determined by rotational speed of wheels or vice versa.

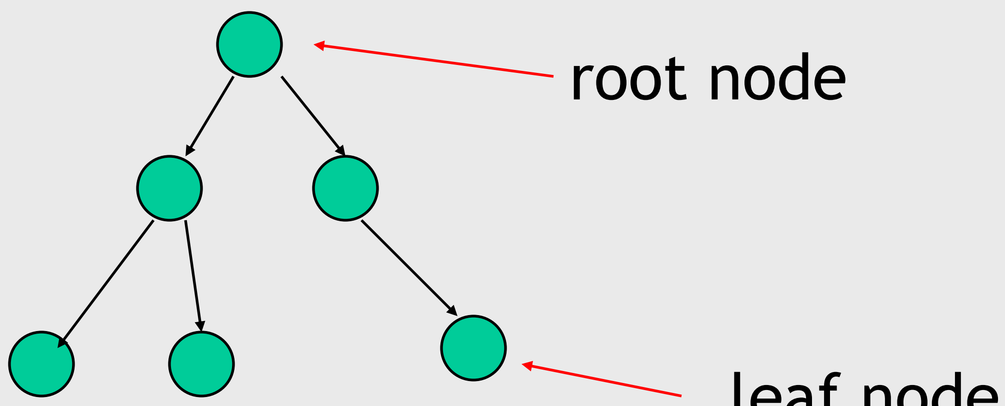
Structure through Function Calls

```
car()
{
    chassis(velocity);
    right_front_wheel(velocity);
    left_front_wheel(velocity);
    right_rear_wheel(velocity));
    left_rear_wheel(velocity);
}
```

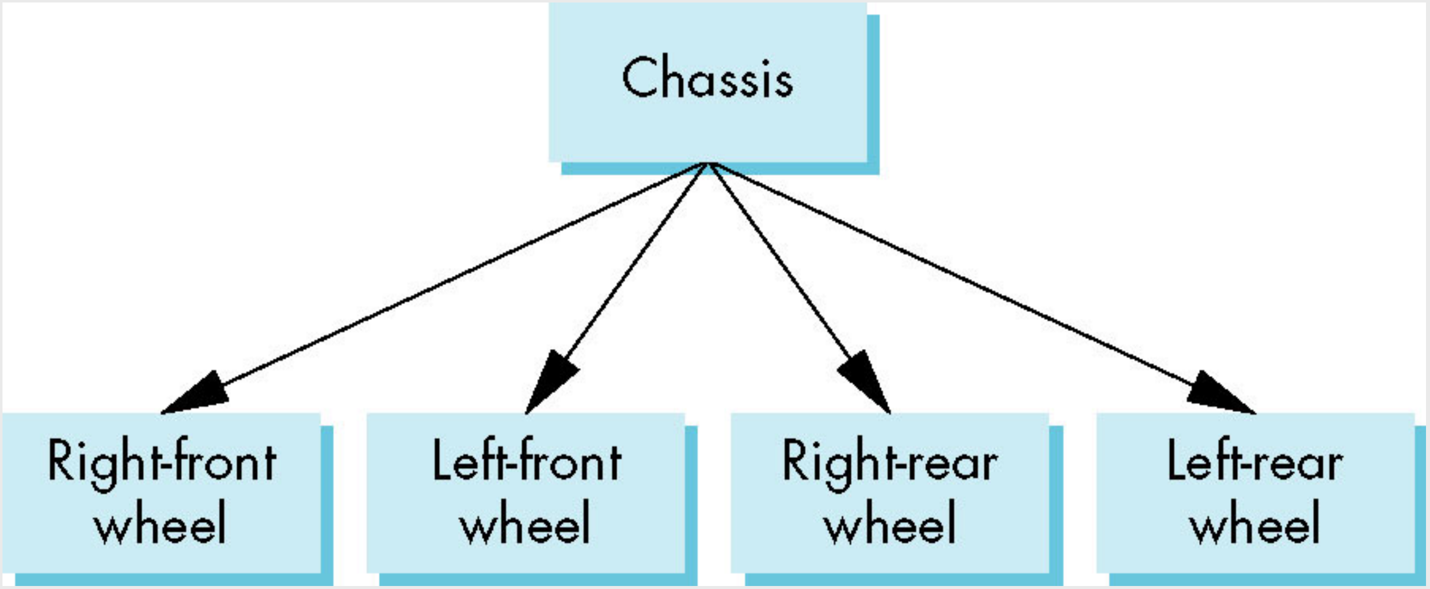
- Fails to show relationships
- What if we use a tree structure?

What is a Tree?

- A directed graph with no loop
  - Each node (except the root) has exactly one parent node
  - May have multiple children
  - Leaf or terminal nodes have no children

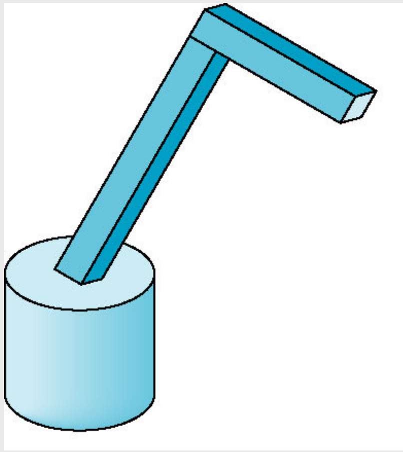


Tree Model of Car

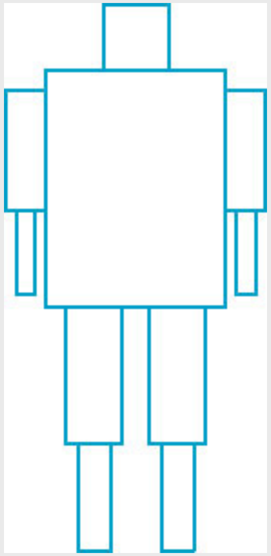


Articulated Models

- Parts are connected at joints
- Can specify state of the model by specifying all joint angles



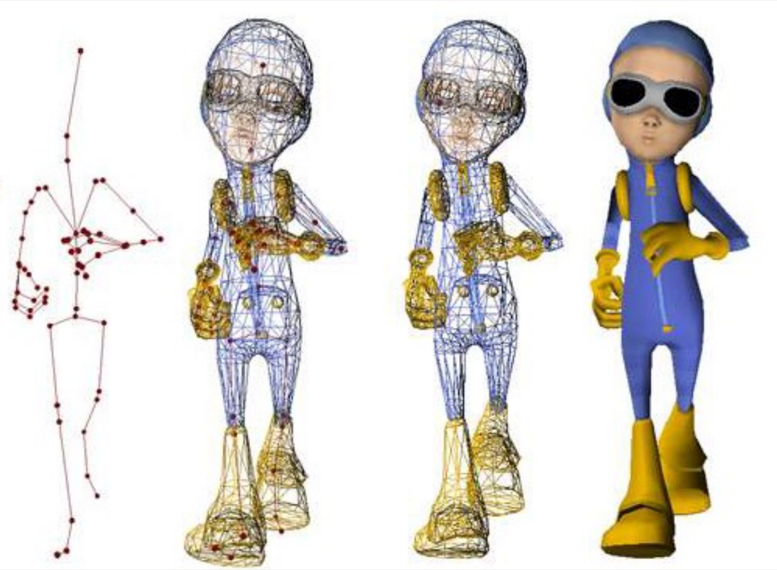
Robot arm



Humanoid

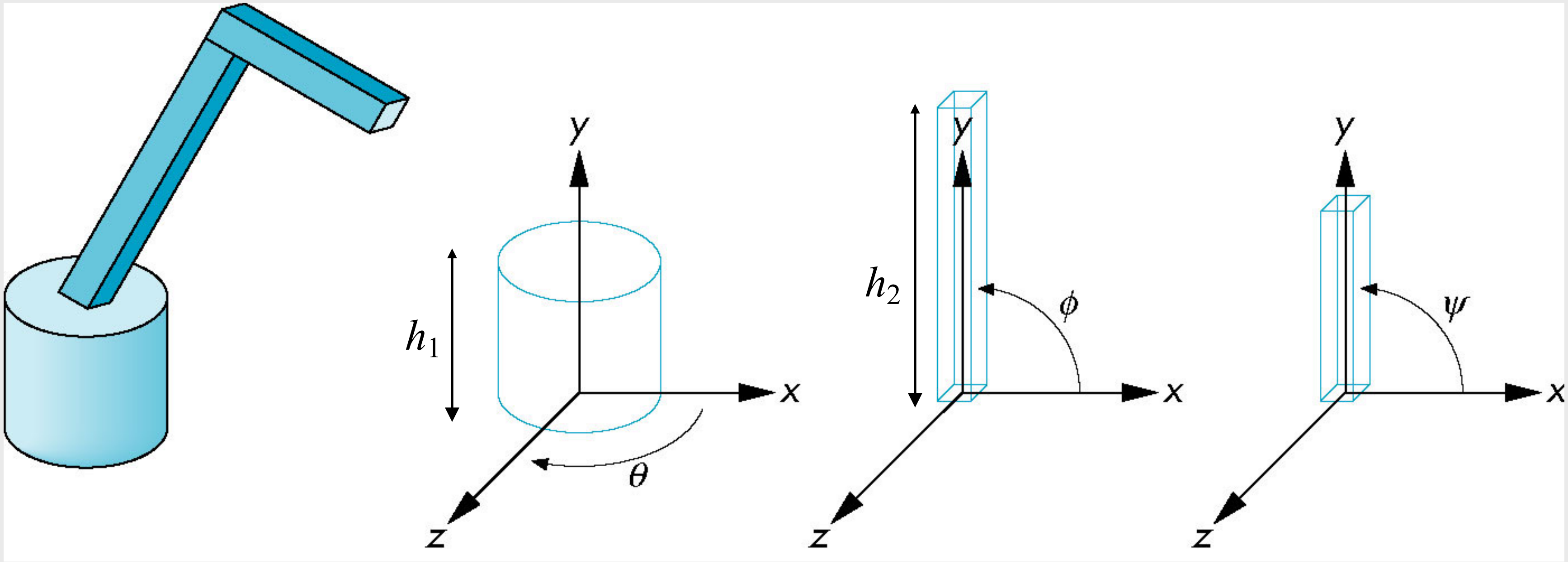
Articulated Models

- Parts are connected at joints
- Can specify state of the model by specifying all joint angles



Skeletal Animation

Robot Arm



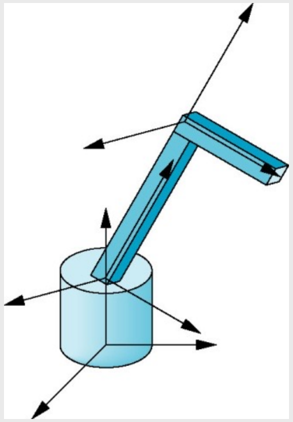
robot arm

parts in their own coordinate systems

We can build the robot arm via instance transformations

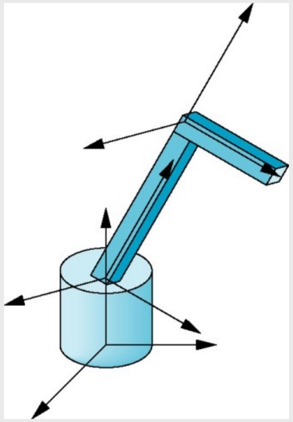
Hierarchical Relationships in Robot Arm

- 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
  - can independently rotate around connecting joint
- Upper arm attached to lower arm
  - Its position depends on rotation of both base and lower arm; must also translate relative to lower arm
  - can independently rotate around joint connecting to lower arm



Required Modelview Matrices

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



Recall: Thinking of Transformations

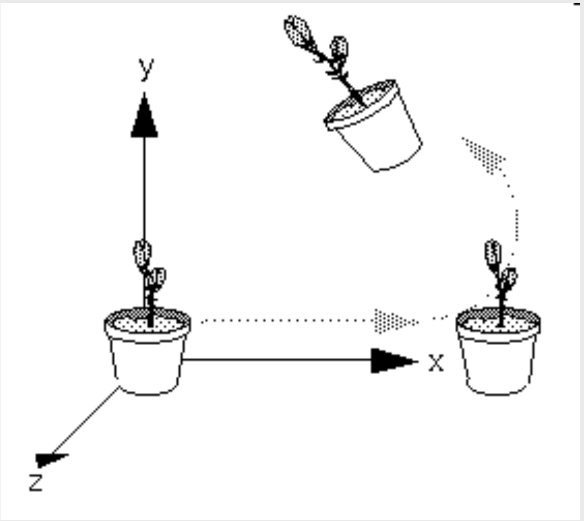
You can think transformations in two different ways:

Think in terms of a local **world** coordinate system; first rotate then translate.

```
Rotate(45.0, 0.0, 0.0, 1.0)*
Translate(5.0, 0.0, 0.0)
```

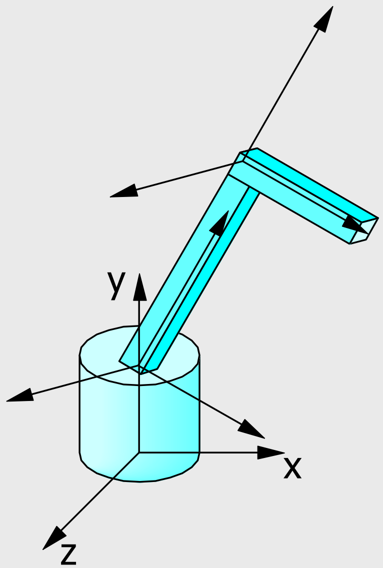
Think in terms of a grand, fixed, **camera** coordinate system; first translate then rotate.

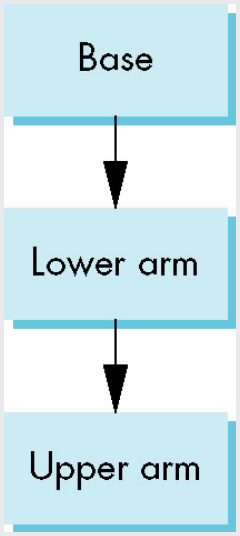
Better for understanding hierarchical models



An OpenGL Code for Robot

```
mat4 m; // modelview matrix
robot_arm()
{
    m = RotateY(theta);
    base(); // draw
    m *= Translate(0.0, h1, 0.0);
    m *= RotateZ(phi);
    lower_arm(); // draw
    m *= Translate(0.0, h2, 0.0);
    m *= RotateZ(psi);
    upper_arm(); // draw
}
```



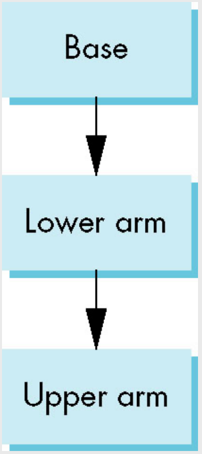


- Note that the code shows relationships between parts of model
  - Can change “look” of parts easily without altering relationships
- The modelview matrix for the upper arm is

$$\boldsymbol{M} = \boldsymbol{R}_b(\theta)\boldsymbol{T}_{la}(h_1)\,\boldsymbol{R}_{la}(\varphi)\,\boldsymbol{T}_{ua}(h_2)\,\boldsymbol{R}_{ua}(\psi)$$

- The code implements a tree traversal

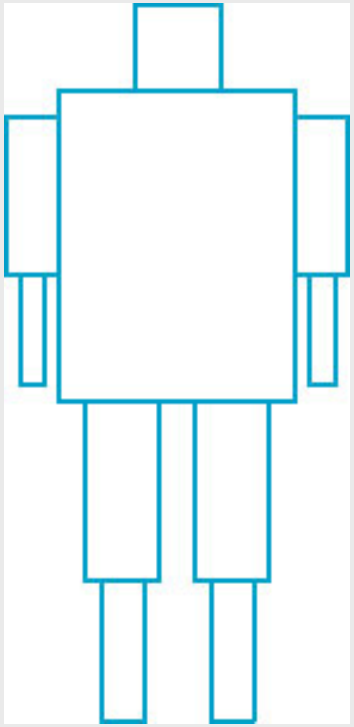
Generalization

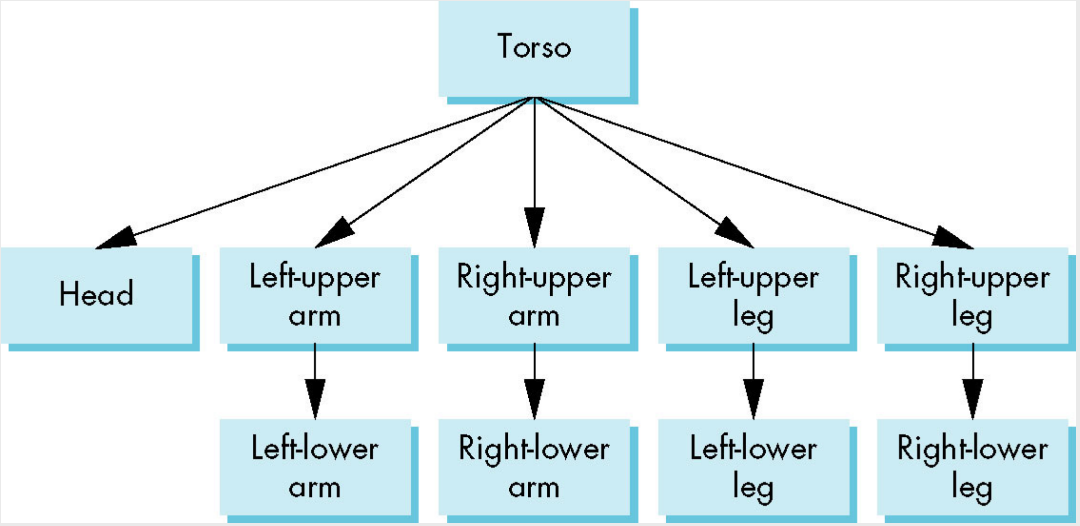


- May need to deal with multiple children
  - How do we represent a more general tree?
  - How do we traverse such a general data structure?

- Animation
  - How to use it dynamically?
  - Can we create and delete nodes (objects) during execution?

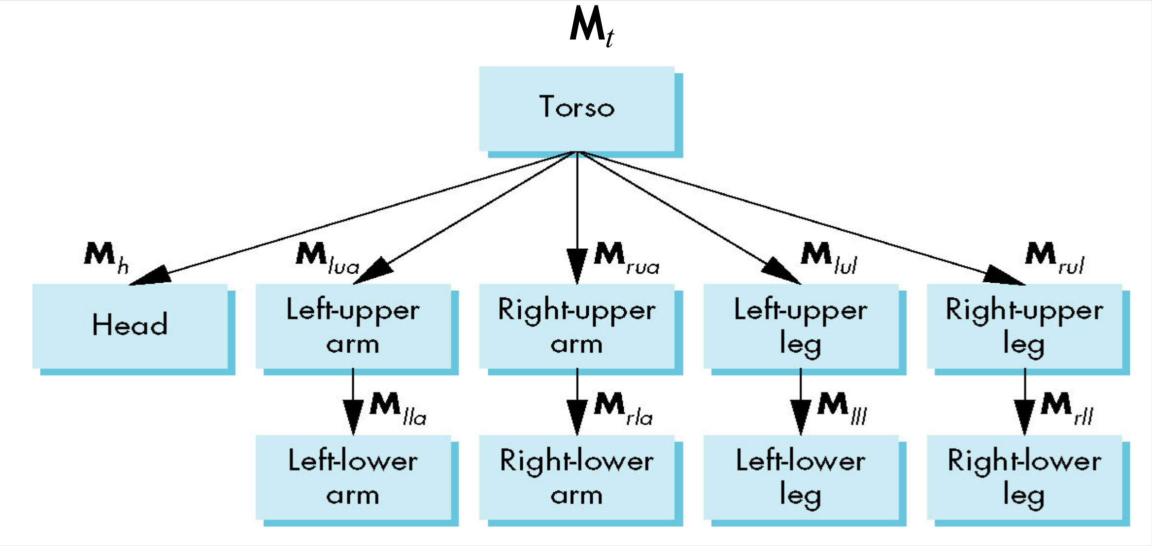
Humanoid Model





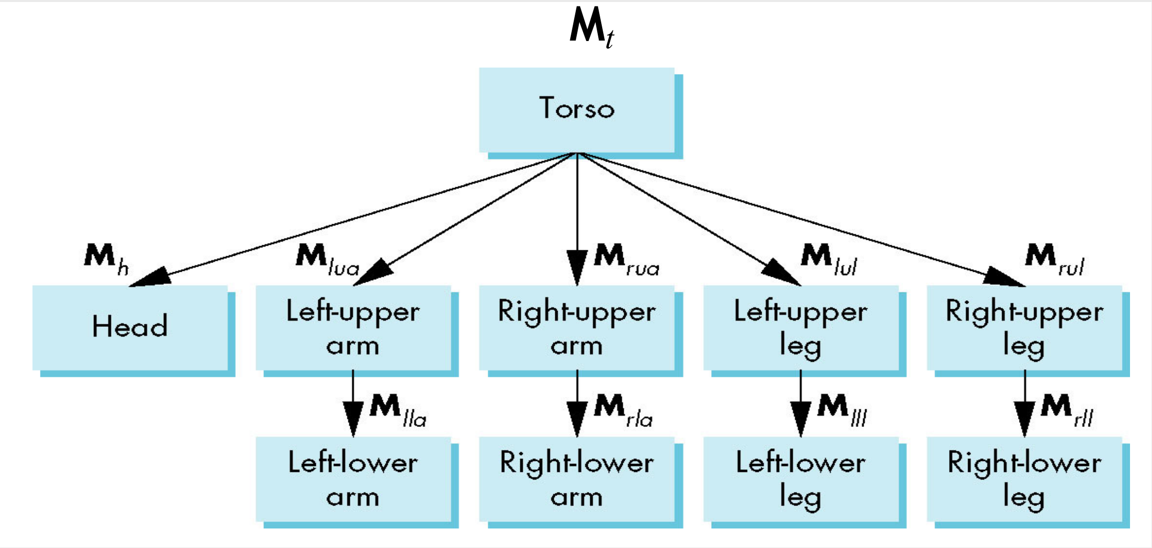
Building the Model

- Can build a simple implementation using quadrics: ellipsoids and cylinders
- Draw (render) model parts through functions such as
  - `torso()`
  - `left_upper_arm()`
- Matrices describe the pose of a node (part) with respect to its parent
  - M_{lla} positions left lower arm with respect to left upper arm



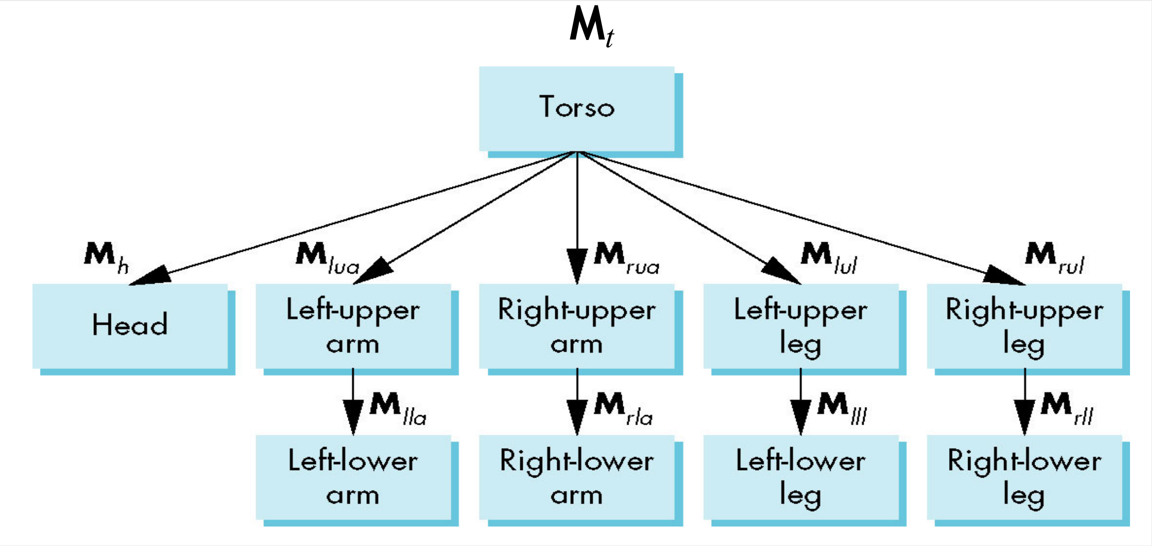
Display and Traversal

- The position of the model is determined by 11 joint angles (two for the head and one for each other part)
- Display of the tree can be thought of as a graph traversal
  - Visit each node once
  - Execute the display function at each node, that describes the part associated with the node
  - By applying the correct transformation matrix for position and orientation



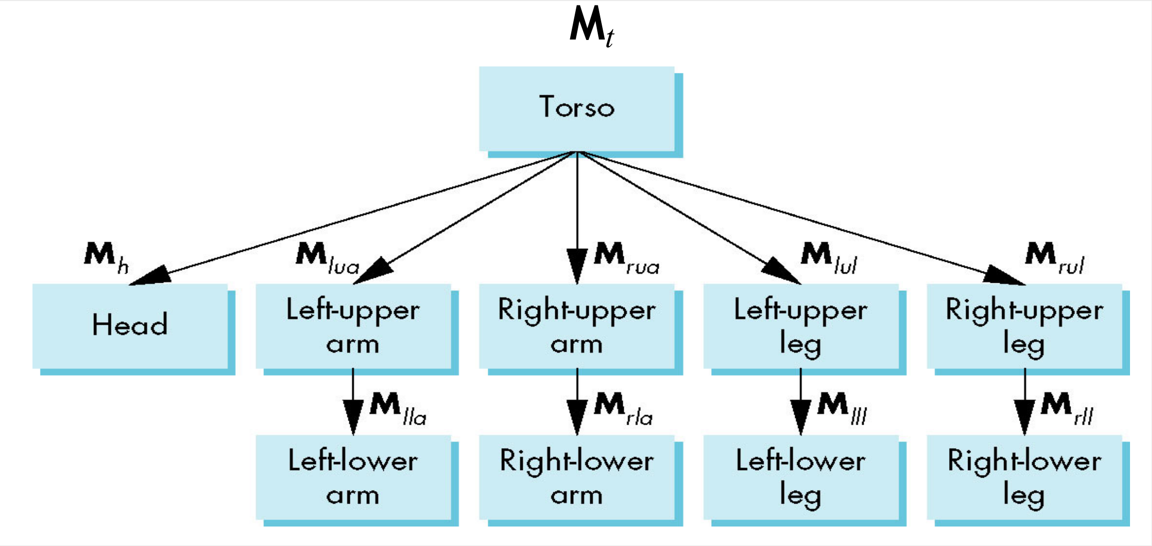
Transformation Matrices

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

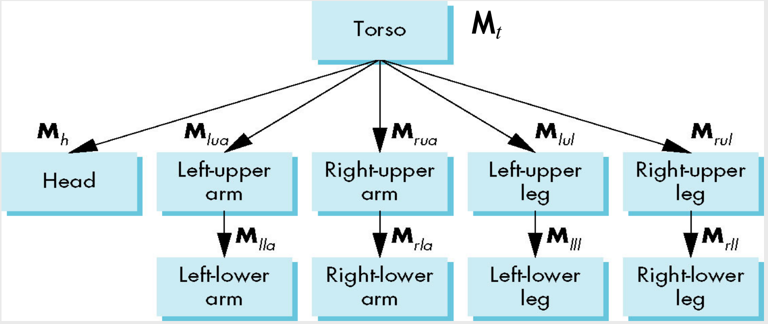


Stack-based Traversal

- Set model-view matrix to $M = M_t$ and draw torso
- Set model-view matrix to $M = M_t M_h$ and draw head
- For left-upper arm, we need M_t back to compute $M = M_t M_{lua}$
- Rather than recomputing $M_t M_{lua}$ from scratch or using an inverse matrix, we can use a matrix stack to store the current M as we traverse the tree
- For left-lower arm, we need $M = M_t M_{lua} M_{lla}$ and so on



Traversal Code



Code	
`figure() {`	
`PushMatrix()`	save present model-view matrix
`torso();`	
`Rotate(…);`	update model-view matrix for head
`head();`	
`PopMatrix();`	recover original model-view matrix
`PushMatrix();`	save it again
`Translate(…);`	update model-view matrix for left upper arm
`Rotate(…);`	
`left_upper_arm();`	
`Translate(…);`	update model-view matrix for left lower arm
`Rotate(…);`	
`left_lower_arm();`	
`PopMatrix();`	recover original model-view matrix
`PushMatrix();`	save it again
`…`	rest of code
`}`	

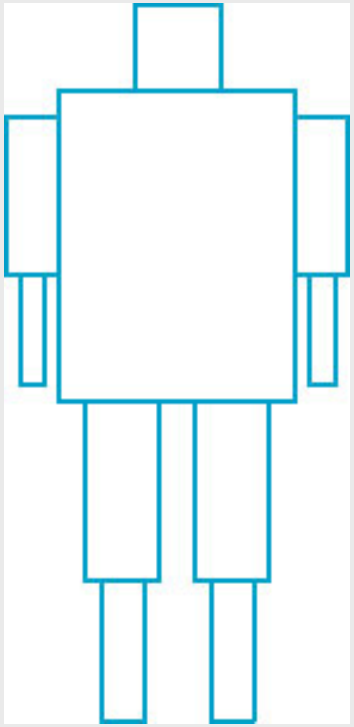
Attribute Changes

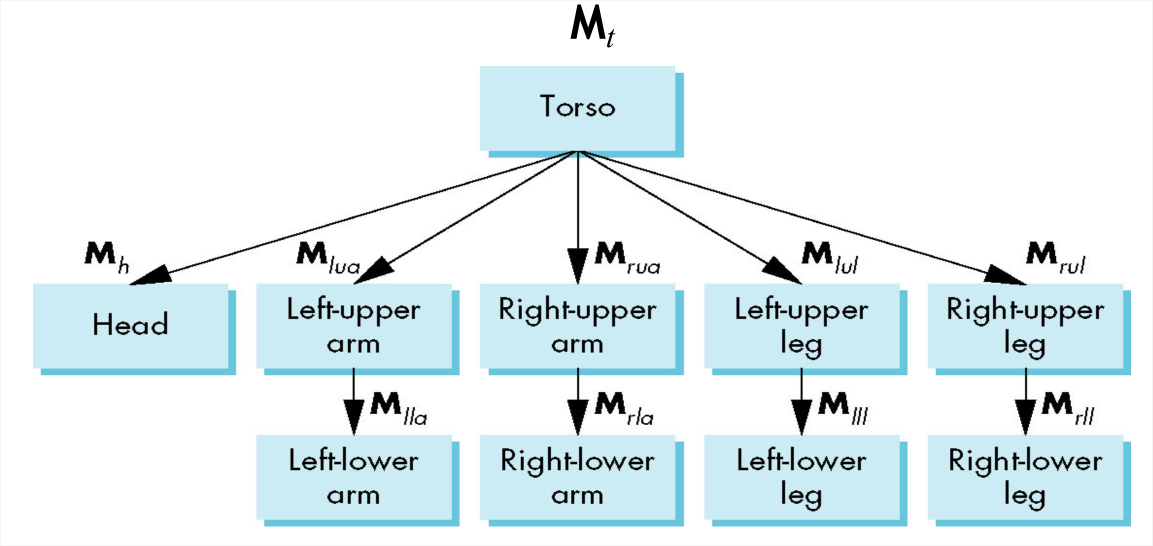
- Note that the sample code does not include any state changes
  - May also want to use **`PushAttrib`** and **`PopAttrib`** to protect against unexpected state changes that may affect later parts of the code
  - OpenGL has `glPushAttrib` and `glPopAttrib`
  - However, since OpenGL now has very few built-in state variables and everything is determined in the shaders, application programmer should implement `pushAttribute()` and `PopAttribute()` methods if necessary.

A More General Approach

- The previous code describes a particular tree and a particular traversal for a particular object
  - Note that the tree structure is not created physically
  - Can we develop a more general approach?
- Create explicitly a standard tree data structure to represent hierarchy
- Then render it with a universal traversal algorithm independent of the model

Humanoid Model





- We can convert animation of the humanoid model into a tree traversal problem.
- We physically create a tree that represents the object.
- Then use a generic traversal code to render it.

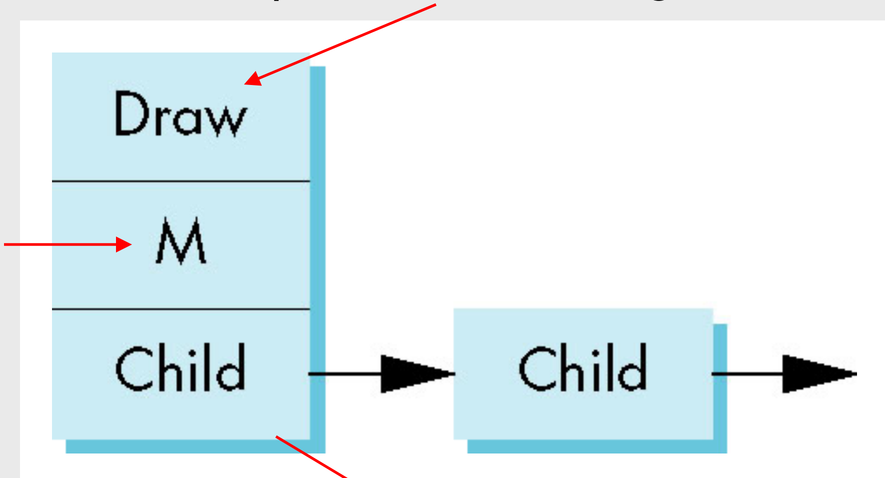
Modeling with Trees

- Must first decide what information to place in nodes
- Nodes will store
  - What to draw
  - Pointers to child nodes
  - Relative transformation matrices

A Possible Node Structure

Code for drawing part or
pointer to drawing function

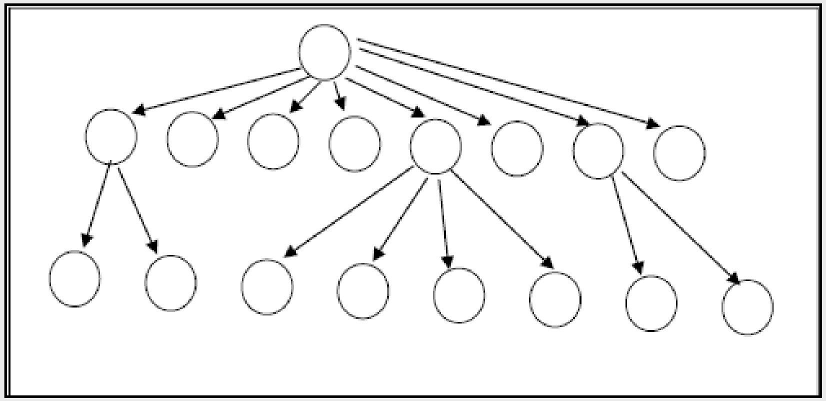
Matrix relating the
node to its parent

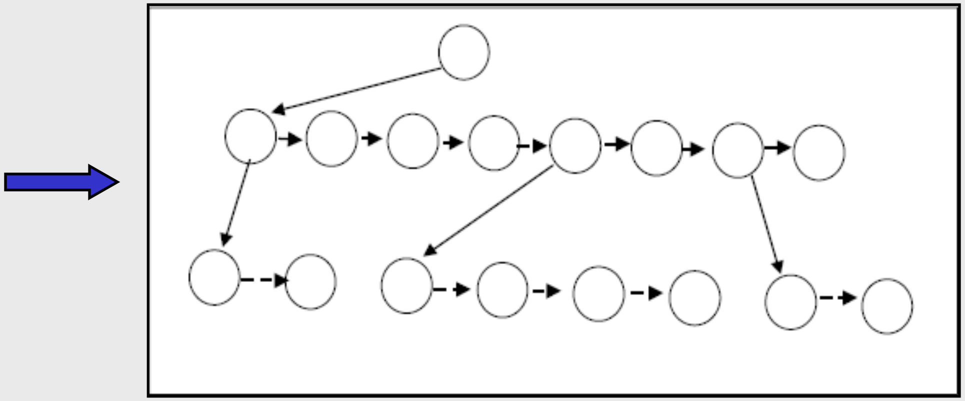


Pointer to linked list of children

General Tree Data Structure

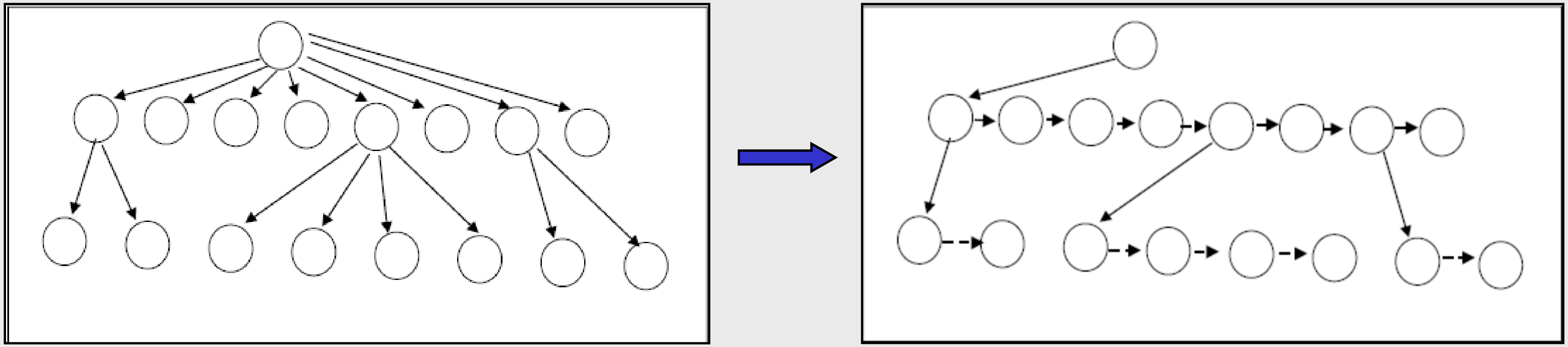
- Need a data structure to represent a tree and also an algorithm to traverse the tree
- We can use a left-child right-sibling structure
  - Uses linked lists
  - Each node in data structure has two pointers
  - **Left**: to the first child of the node
  - **Right**: to the linked list of siblings





Node Structure

- At each node we need to store
  - Pointer to sibling
  - Pointer to child
  - A function that draws the object represented by the node
  - Transformation matrix to multiply on the right of the current model-view matrix
    - Represents changes going from parent to node



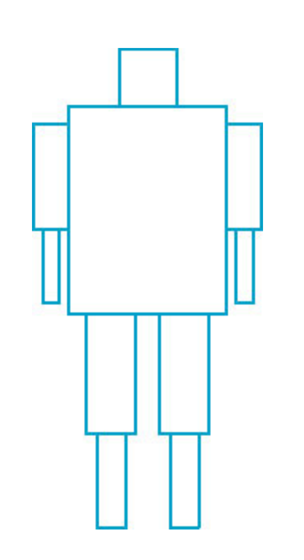
C/C++ Definition of node

```
typedef struct node
{
   Glfloat m[16];
   void (*render)();
   struct node *sibling;
   struct node *child;
} node;
```

Pointers

Back to humanoid example

- The position of model is determined by 11 joint angles stored in **`theta[11]`**
- Animate by changing the angles and redisplaying
- Form the required modelview matrices using **`Rotate`** and **`Translate`** functions



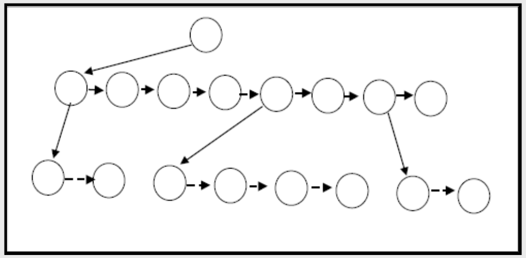
Defining torso and head nodes

```
node torso_node, head_node, lua_node, … ;

torso_node.m = RotateY(theta[0]);
torso_node.render = torso_draw; /* torso_draw() draws torso */
torso_node.sibling = NULL;
torso_node.child = &head_node;

head_node.m = Translate(0.0, TORSO_HEIGHT+0.5*HEAD_HEIGHT,
  0.0)*RotateX(theta[1])*RotateY(theta[2]);
head_node.render = head_draw;
head_node.sibling = &lua_node;
head_node.child = NULL;
```

Could be better written in C++!

(see also the provided code)

Preorder Traversal (depth-first)



```
void traverse(node *root)
{
   if(root==NULL) return;
   mvstack.push(model_view);
   model_view = model_view*root->m;
   root->render();
   if(root->child!=NULL)
        traverse(root->child);
   model_view = mvstack.pop();
   if(root->sibling!=NULL)
        traverse(root->sibling);
}
```

Generic and universal!

Can traverse and render any left-child right-sibling structure

Rendering is achieved by

```
traverse(&torso_node)
```

Notes

- We must save modelview matrix before multiplying it by node matrix
- The updated matrix applies to children of a node but not to siblings which contain their own matrices
- The traversal program applies to any left-child right-sibling tree
- The order of traversal matters because of possible state changes in the functions; thus you may need to use some **`PushAttrib`** and **`PopAttrib`** functions

Dynamic Trees

- If we use pointers, the structure can be dynamic

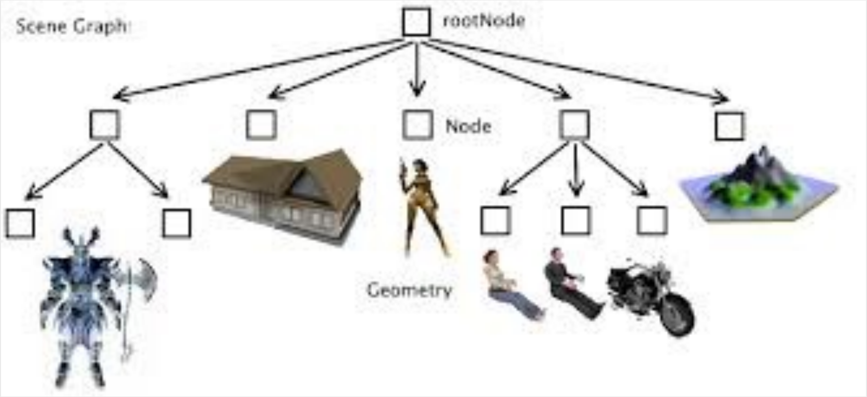
```
node* torso_ptr;
torso_ptr = malloc(sizeof(node));
…
free(torso_ptr);
```

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

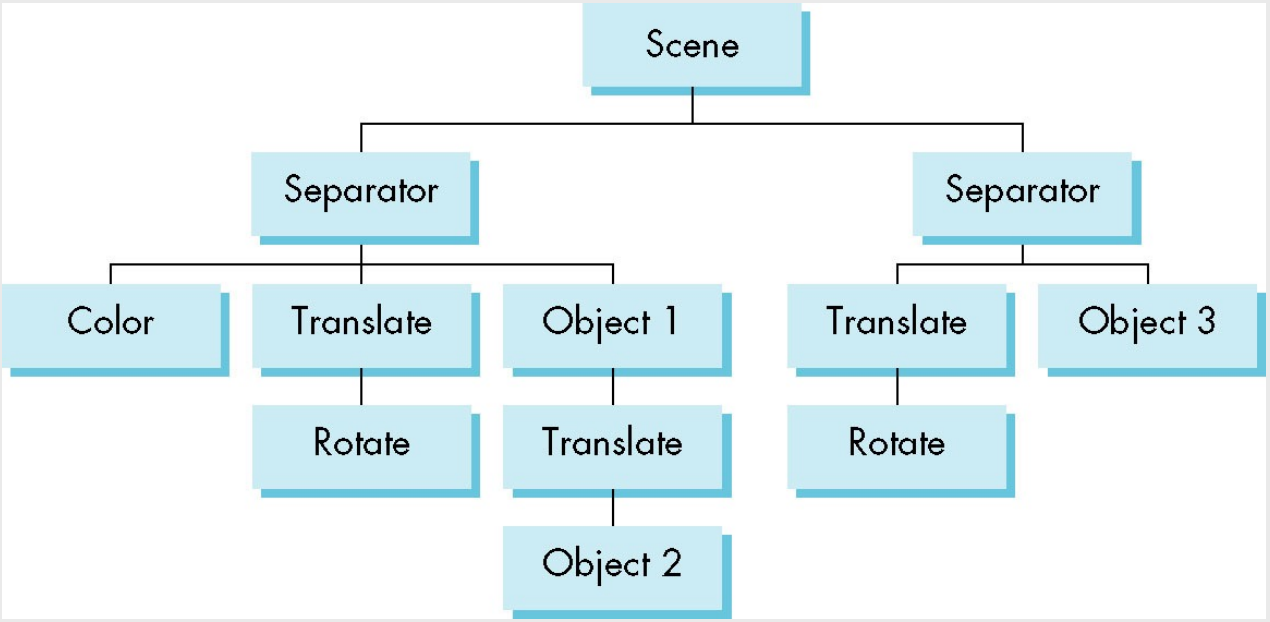
Scene Graphs

Scene Graphs

- In computer graphics, such tree structures are also commonly used to describe graphical scenes composed of many objects
- Called as “scene graphs”
- Used in Java3D, OpenInventor, VRML, X3D, Open Scene Graph, etc.



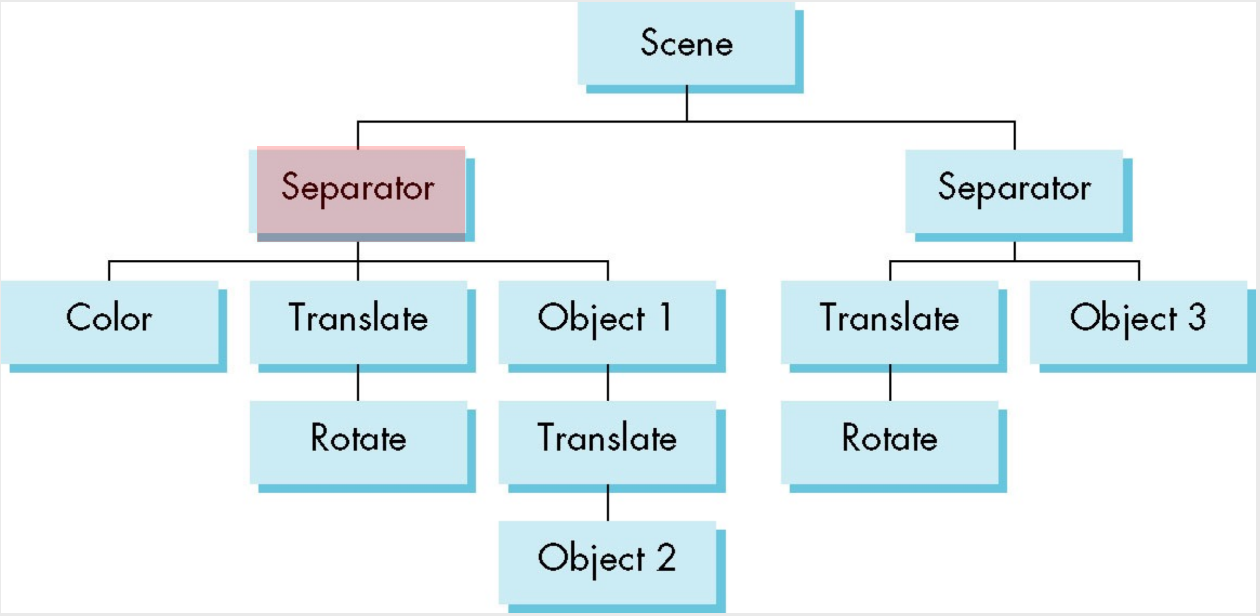
Scene Graph Structure



Scene graphs usually include also other geometrical and non-geometrical components of a scene such as materials, camera, lights, etc.

Preorder Depth-First Traversal

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



Scene graphs can be rendered using a universal traversal algorithm

- Need separator nodes to isolate state changes
- Otherwise, state changes can propagate
- Equivalent to OpenGL Push/Pop

Open Inventor and Java3D

- Java3D, Open Inventor and Open Scene Graph each provides a scene graph API
- Primitives supported by APIs should match capabilities of graphics systems
  - Hence most scene graph APIs are built on top of OpenGL or DirectX
- Scene graphs can also be described by a file (ASCII or binary such as VRML, X3D)
  - Implementation independent way of transporting scenes
  - Can easily be visualized by scene graph APIs

VRML (superceded by X3D)

- Describes a scene graph that can be used over the World Wide Web
- Virtual Reality Markup Language (VRML)
  - Based on Open Inventor data structure
  - Open Inventor is built on top of OpenGL
- Example:

```
#VRML V1.0 ascii

Separator {
     Translation{
          translation 2.25 0 0
     }
     Material {
          emissiveColor    1 0 0
     }
     Cube {}
}
```