Xi'an Jiaotong-Liverpool University
西交利物浦大学

CPT205 Computer Graphics

Hierarchical Modelling

Lecture 08
2022-23

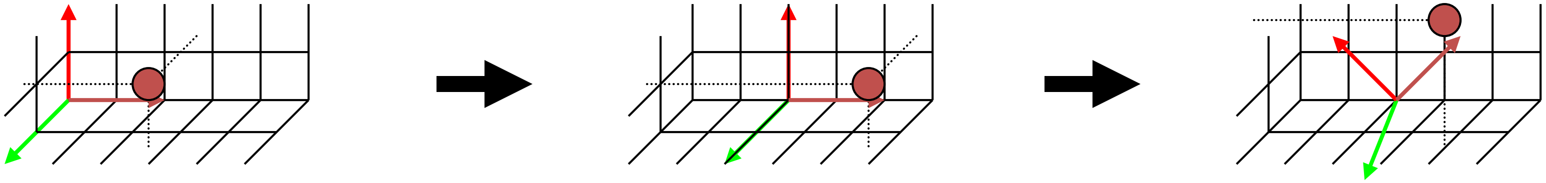
Yong Yue

Topics for today

- Local and world co-ordinate frames of reference
- Object transformations
- Linear modelling
    - Symbols
    - Instances
- Hierarchical modelling
    - Hierarchical trees
    - Articulated models
- Examples and code

Local and world frames of reference (1)

- We are used to defining points in space as (x,y,z). But what does that actually mean? Where is (0,0,0)?
- The actual truth is that there is no (0,0,0) in the real world. Objects are always defined *relative* to each other.
- We can *move* (0,0,0) and thus move all the points defined relative to that origin.



Local and world frames of reference (2)

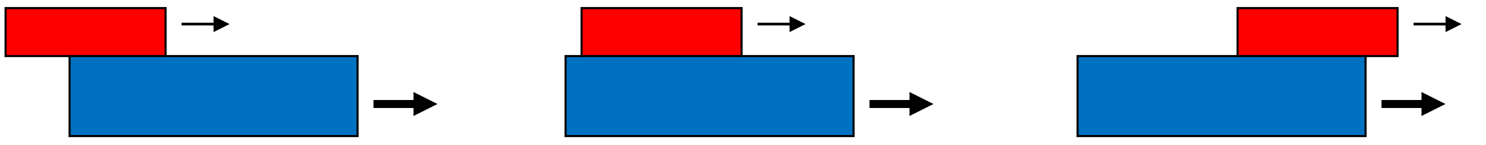
- The following terms are used interchangeably
  - *Local basis*
  - *Local transformation*
  - *Local / model frame of reference*
- Each of these refers to the location, in the greater world, of the (0,0,0) we are working with
  - They also include the concept of the current *local frame*, which is about the x, y, z directions.
  - By rotating the *local frame* of a coordinate system, we can rotate the world it describes.

What does the centre of the world mean?

- A *world frame of reference* is defined for a scene of objects.
- Each object has a *local frame of reference* which is relevant to the world frame.

Relative motion

- Relative motion - a motion takes place relative to a local origin.

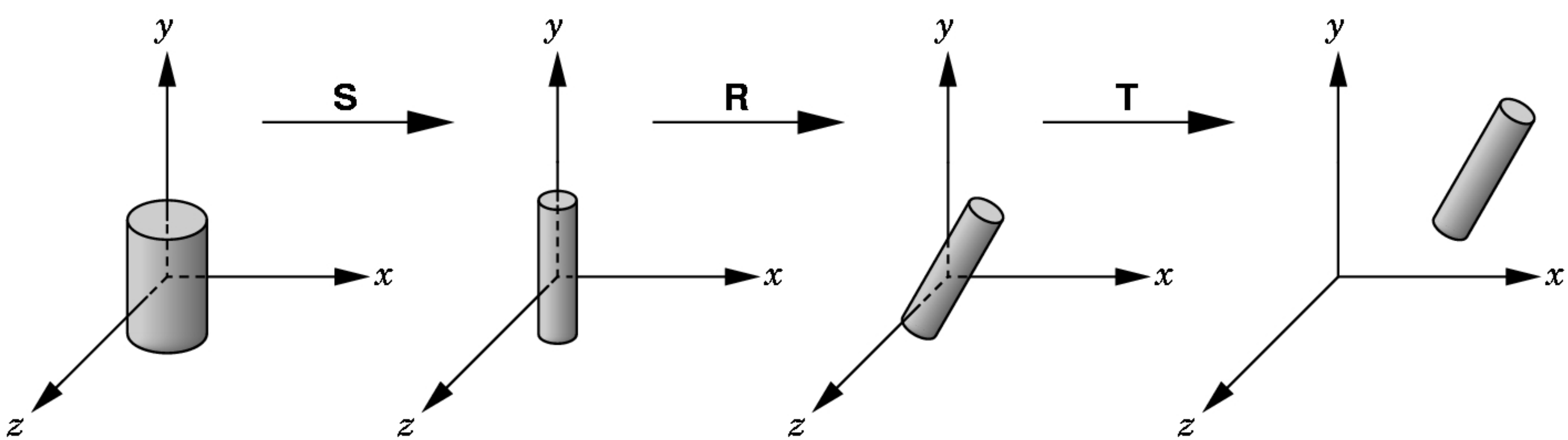


e.g. throwing a ball to a friend as you both ride in a train.

- The term *local origin* refers to the (0,0,0) that is chosen to measure the motion from.
- The local origin may be moving relative to some greater frame of reference.

Linear modelling (1)

- Start with a *symbol* (prototype)
- Each appearance of the object in the scene is an *instance*
  - We must scale, orient and position it to define the instance transformation
  - **M = T · R · S**



Linear modelling (2)

In OpenGL

- Set up appropriate transformations from the model frame (frame of symbols) to the world frame
- Apply it to the MODELVIEW matrix before executing the code

```
glMatrixMode(GL_MODELVIEW); // M = T·R·S
glLoadIdentity();
glTranslatef();
glRotatef();
glScalef();
glutSolidCylinder()          // or other symbol
```

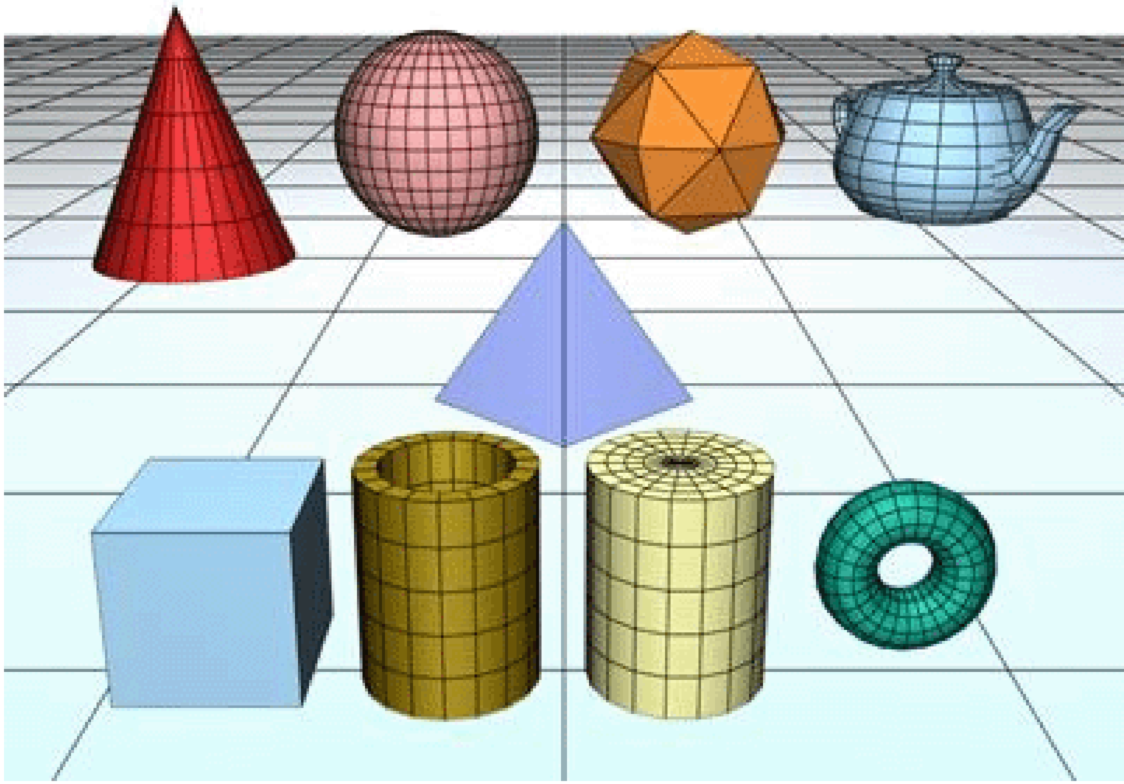
Linear modelling (3)

Example: generating a cylinder

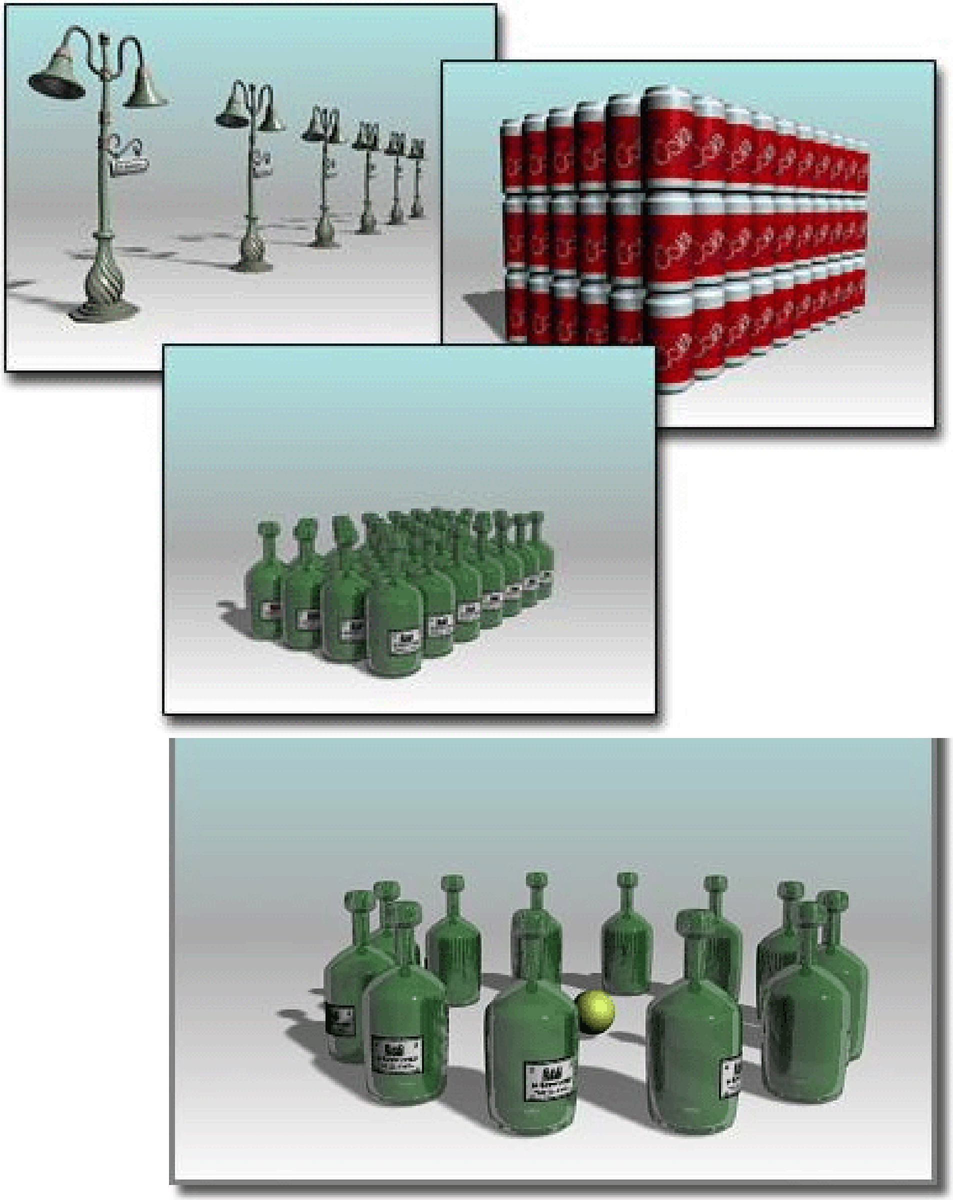
```
glBegin(GL_QUADS);
   For each A = Angles
   {
      glVertex3f(R*cos(A), R*sin(A), 0);
      glVertex3f(R*cos(A+DA), R*sin(A+DA), 0);
      glVertex3f(R*cos(A+DA), R*sin(A+DA), H);
      glVertex3f(R*cos(A), R*sin(A), H);
   }
glEnd();

// Make Polygons for Top/Bottom of cylinder
```

Linear modelling (4)

- Symbols (Primitives)
  Cone, Sphere, GeoSphere, Teapot, Box, Tube, Cylinder, Torus, etc.
- Copy
  Creates a completely separate clone from the original. Modifying one has no effect on the other.
- Instance
  Creates a completely interchangeable clone of the original. Modifying an instanced object is the same as modifying the original.



Linear modelling (5)



- Array: series of clones
  - Linear
    - Select object
    - Define axis
    - Define distance
    - Define number
  - Radial
    - Select object
    - Define axis
    - Define radius
    - Define number

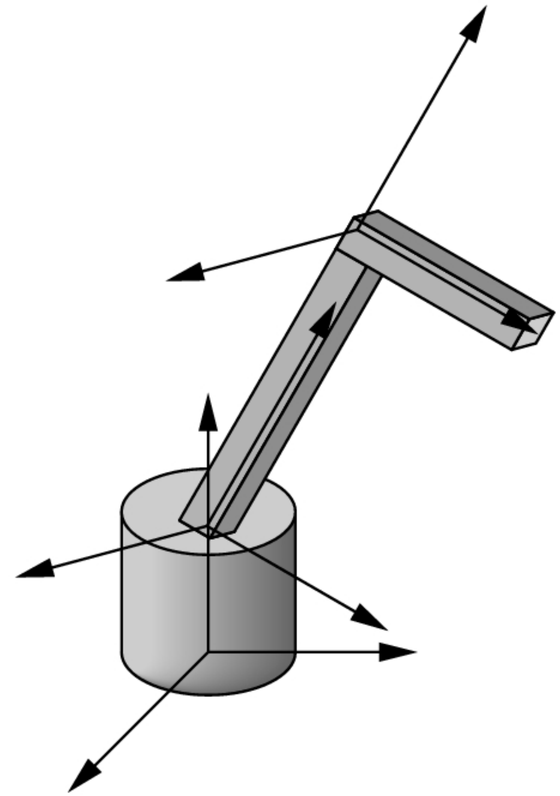
Linear modelling (6)

- Model stored in a table by
  - assigning a number to each symbol and
  - storing the parameters for the instance transformation
- Contains flat information
  but no information on the actual structure
- How to represent complex structures with constraints?
- Each part has its own model frame of co-ordinate system
  but no information of relationships
- How to manipulate with substructures?

Linear modelling (7)

Symbol	Scale	Rotate	Translate
1	s_x, s_y, s_z	$u_x,\ u_y,\ u_z$	d_x, d_y, d_z
2			
3			
1			
1			
.			
.			

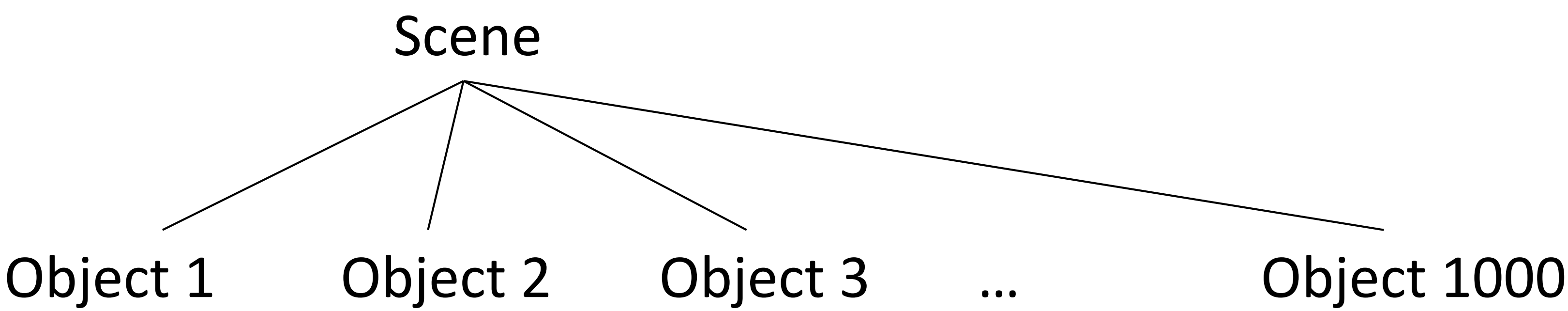
Linear model table



Model with constraints

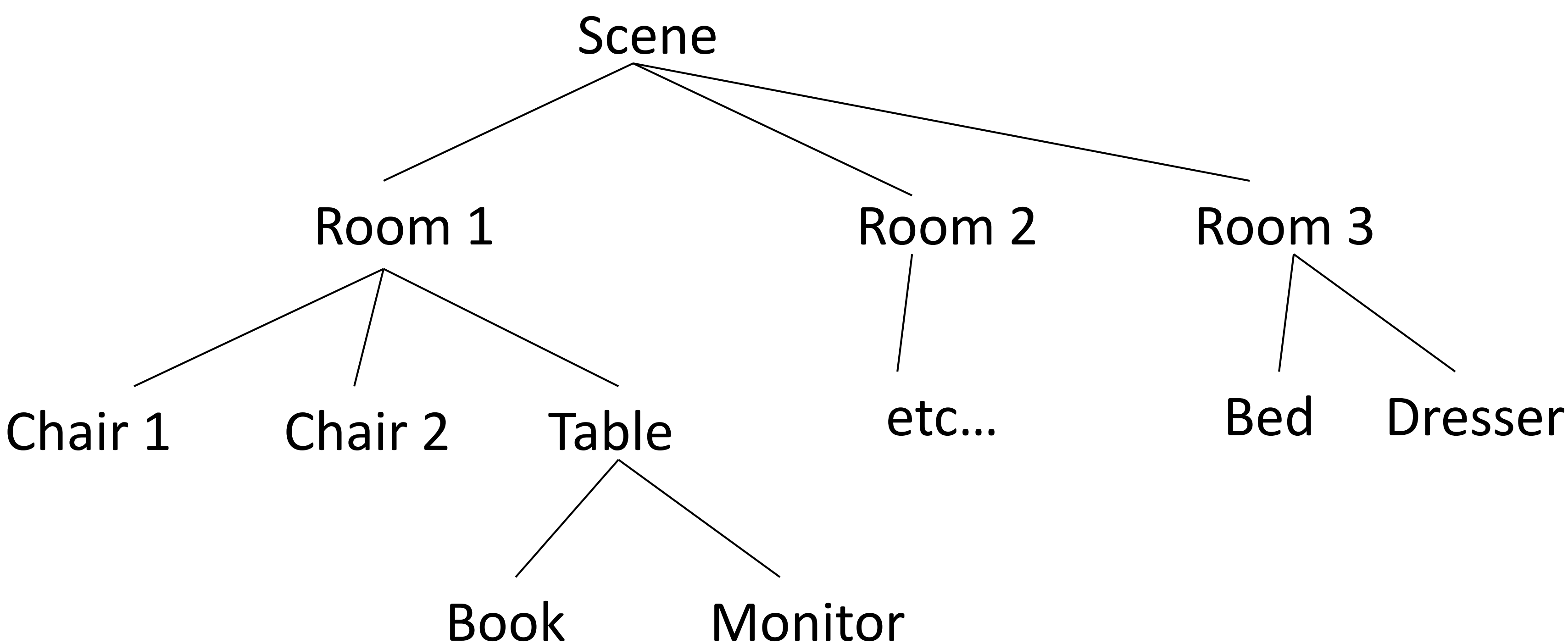
Scene hierarchy (1)

If a scene contains 1000 objects, we might think of a simple organisation like this.



Scene hierarchy (2)

We could also have a hierarchical grouping like this.

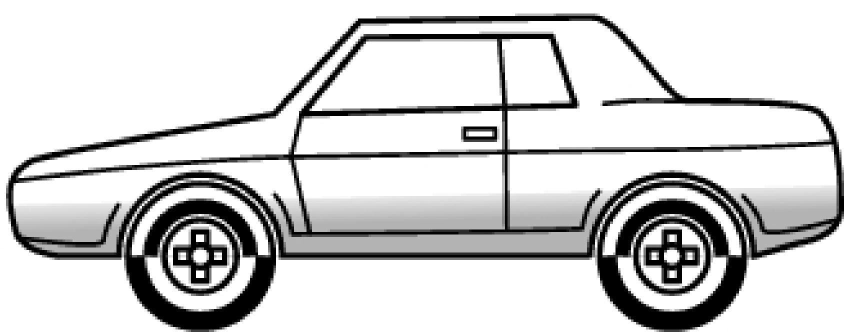


Scene hierarchy (3)

- In a scene, some objects may be grouped together in some way. For example, an *articulated* figure may contain several rigid components connected together in a specified fashion.
  - several objects sitting on a tray that is being carried around
  - a bunch of moons and planets orbiting around in a solar system
  - a hotel with 200 rooms, each room containing a bed, table, chairs, etc.
- In each of these cases, the placement of objects is described more easily when we consider their locations relative to each other.

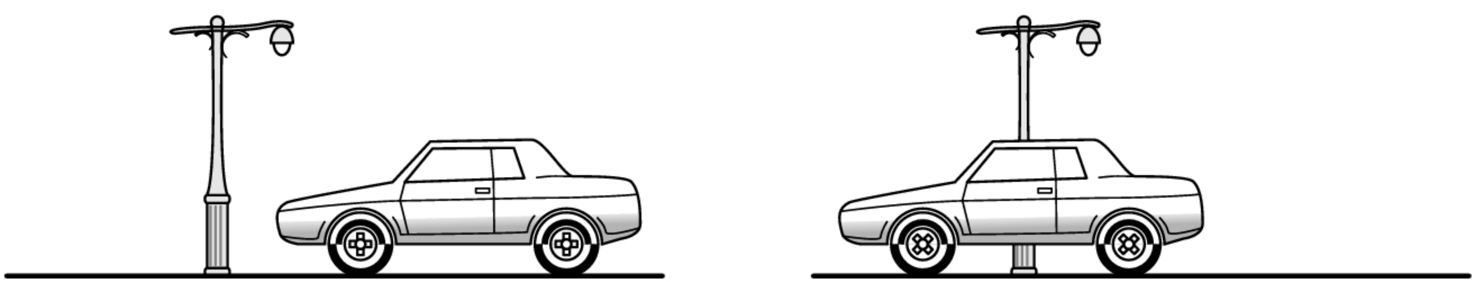
Hierarchical models – a car (1)

- Consider the model of a car
  - Chassis + 4 identical wheels
  - Two symbols
- Speed of the car is actually determined by the rotational speed of wheels or vice versa.



Hierarchical models – a car (2)

```
void main ( );
{   float s = ...;              // speed
    float d[3] = {...};         // direction
    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);
}   // WE DO NOT WANT THIS!
```



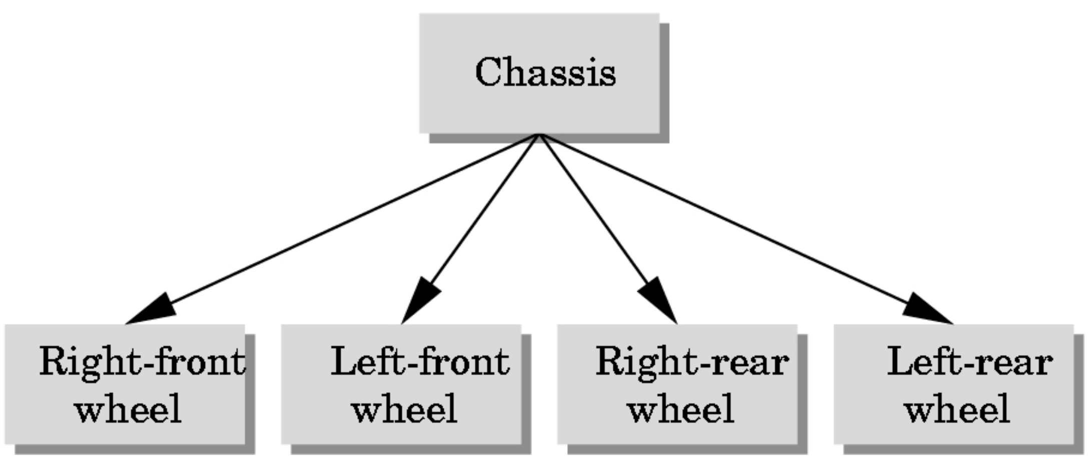
Two frames of reference for animation

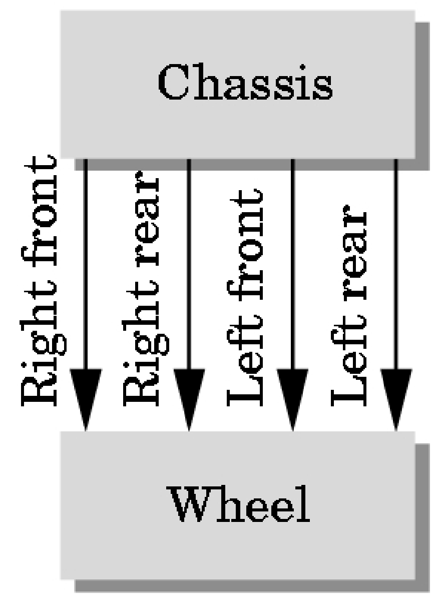
Hierarchical tree (1)

- It is very common in computer graphics to define a complex scene in some sort of hierarchical fashion.
- The individual objects are grouped into a hierarchy that is represented by a tree structure (upside down tree).
  - Each moving part is a single *node* in the tree.
  - The node at the top is the *root node.*
  - Each node (except the root) has exactly one *parent* node which is directly above it.
  - A node may have multiple *children* below it.
  - Nodes with the same parent are called *siblings.*
  - Nodes at the bottom of the tree with no children are called *leaf nodes.*

Hierarchical tree (2)

- Direct Acyclic Graph (DAG) stores a position of each wheel.
- Trees and DAGs – hierarchical methods express the relationships.

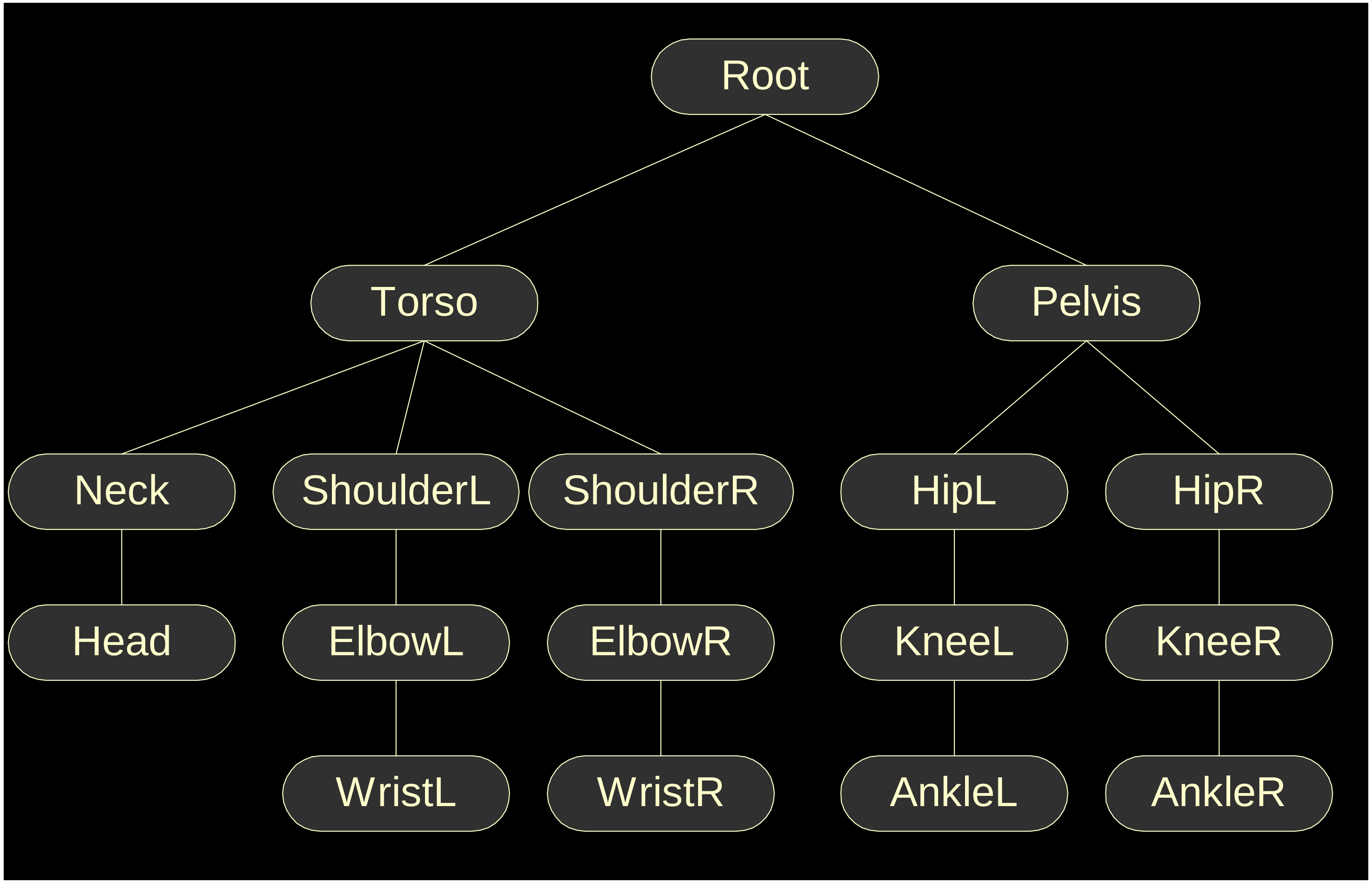




Articulated model

- An articulated model is an example of a hierarchical model consisting of rigid parts and connecting joints.
- The moving parts can be arranged into a tree data structure if we choose some particular piece as the ‘root’.
- For an articulated model (like a biped character), we usually choose the root to be somewhere near the centre of the torso.
- Each joint in the figure has specific allowable *degrees of freedom* (*DOFs*) that define the range of possible poses for the model.

Articulated model – biped character



Hierarchical transformations

- Each *node* in the tree represents an object that has a matrix describing its location and a model describing its geometry.
- When a node up in the tree moves its matrix,
  - it takes its children with it (in other words, rotating a character's shoulder joint will cause the elbow, wrist, and fingers to move as well).
  - so child nodes inherit transformations from their parent node.
- Each node in the tree stores a *local matrix* which is its transformation *relative to its parent.*
- To compute a node's *world space matrix*, we need to concatenate its local matrix with its parent's world matrix: $\mathbf{M}_{\mathbf{world}} = \mathbf{M}_{\mathbf{parent}} \cdot \mathbf{M}_{\mathbf{local}}$

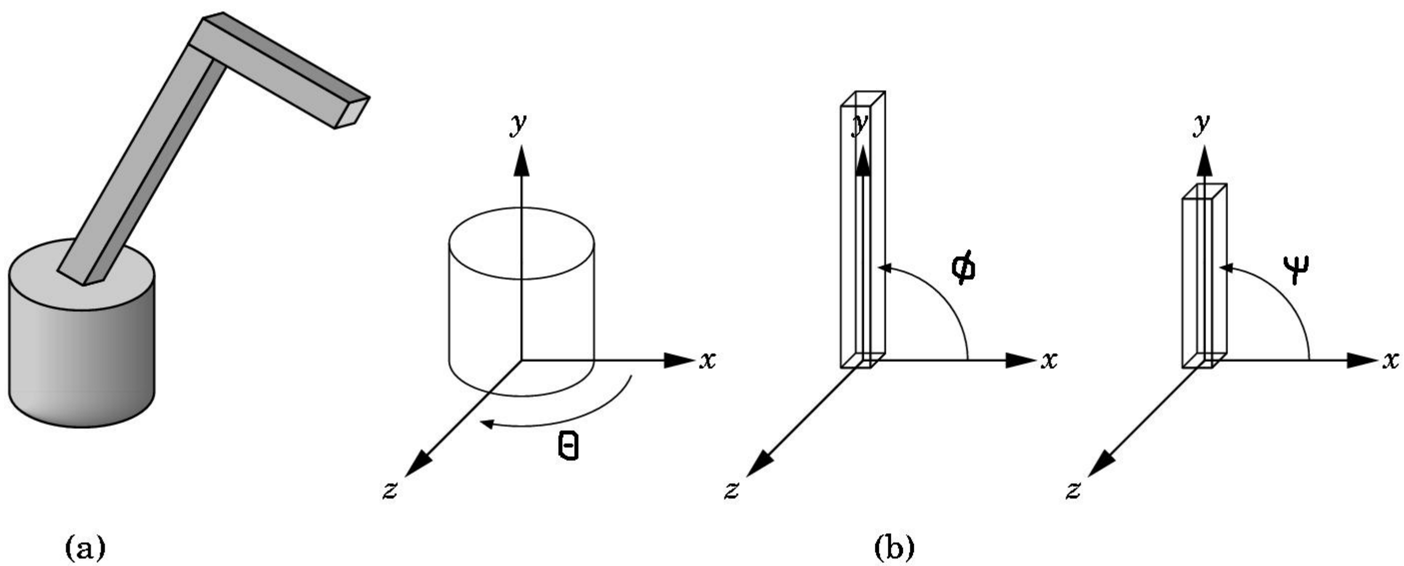
Recursive traversal and OpenGL matrix stacks

- To compute all of the world matrices in the scene, we can traverse the tree in a *depth-first traversal.*
- As each node is traversed, its world space matrix is computed.
- By the time a node is traversed, it is guaranteed that its parent's world matrix is available.
- The GL matrix stack is set up to facilitate the rendering of hierarchical scenes.
- While traversing the tree, we can call **`glPushMatrix()`** when going down a level, and **`glPopMatrix()`** when coming back up.

Articulated model – robot arm (1)

The robot arm is another example of articulated model.

- Parts are connected at joints.
- We can specify state of model by specifying all joint angles.



Robot arm — Parts in their own frames of reference

Articulated model – robot arm (2)

- Base rotates independently
  - Single angle determines position
- Lower arm attached to the base
  - Its position depends on the rotation of the base
  - It must also translate relative to the base and rotate around the connecting joint
- Upper arm attached to lower arm
  - Its position depends on both the base and lower arm
  - It must translate relative to the lower arm and rotate around the joint connecting to the lower arm

Articulated model – robot arm (3)

- Rotate the base: $\mathbf{R_b}$
  Apply $\mathbf{M_{b\text{-}w} = R_b}$ to the base
- Translate the lower arm relative to the base: $\mathbf{T_{la}}$
- Rotate the lower arm around the joint: $\mathbf{R_{la}}$
  Apply $\mathbf{M_{la\text{-}w} = R_b \cdot T_{la} \cdot R_{la}}$ to the lower arm
- Translate the upper arm relative to the lower arm: $\mathbf{T_{ua}}$
- Rotate the upper arm around the joint: $\mathbf{R_{ua}}$
  Apply $\mathbf{M_{ua\text{-}w} = R_b \cdot T_{la} \cdot R_{la} \cdot T_{ua} \cdot R_{ua}}$ to the upper arm

Articulated model – robot arm (4)

- Each of the 3 parts has 1 degree of freedom – described by a joint angle between them.

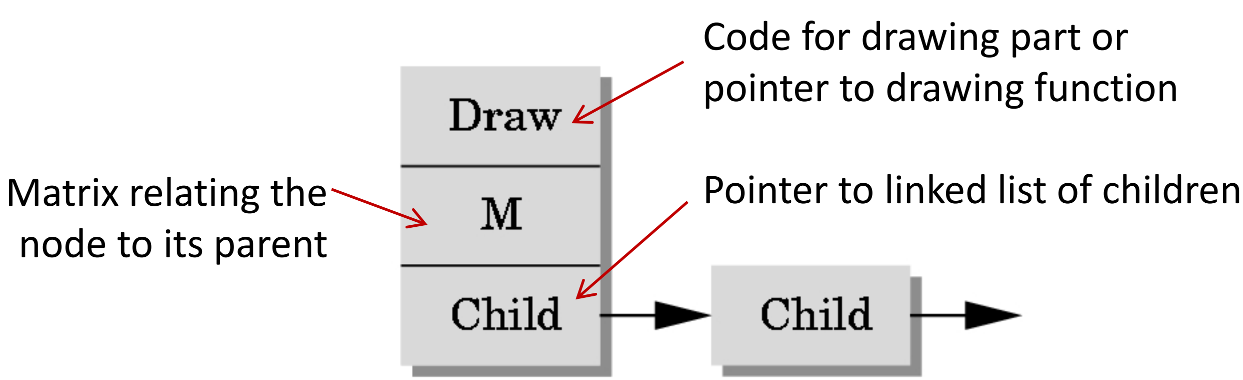
```
void display()
{   glRotatef(theta, 0.0, 1.0, 0.0);
    base();
    glTranslatef(0.0, h1, 0.0);
    glRotatef(phi, 0.0, 0.0, 1.0);
    lower_arm();
    glTranslatef(0.0, h2, 0.0);
    glRotatef(psi, 0.0, 0.0, 1.0);
    upper_arm();
}
```

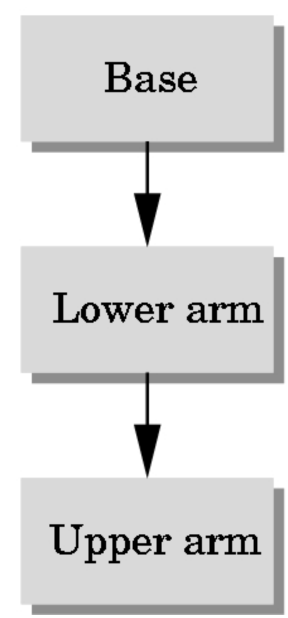
- The code shows relationships between the parts of the model. The appearance can change easily without altering the relationships.
- The MODELVIEW matrix for the upper arm is
  $\mathbf{M_{ua\text{-}w} = R_b(\theta) \cdot T_{la}(h1) \cdot R_{la}(\phi) \cdot T_{ua}(h2) \cdot R_{ua}(\psi)}$

Articulated model – robot arm (5)

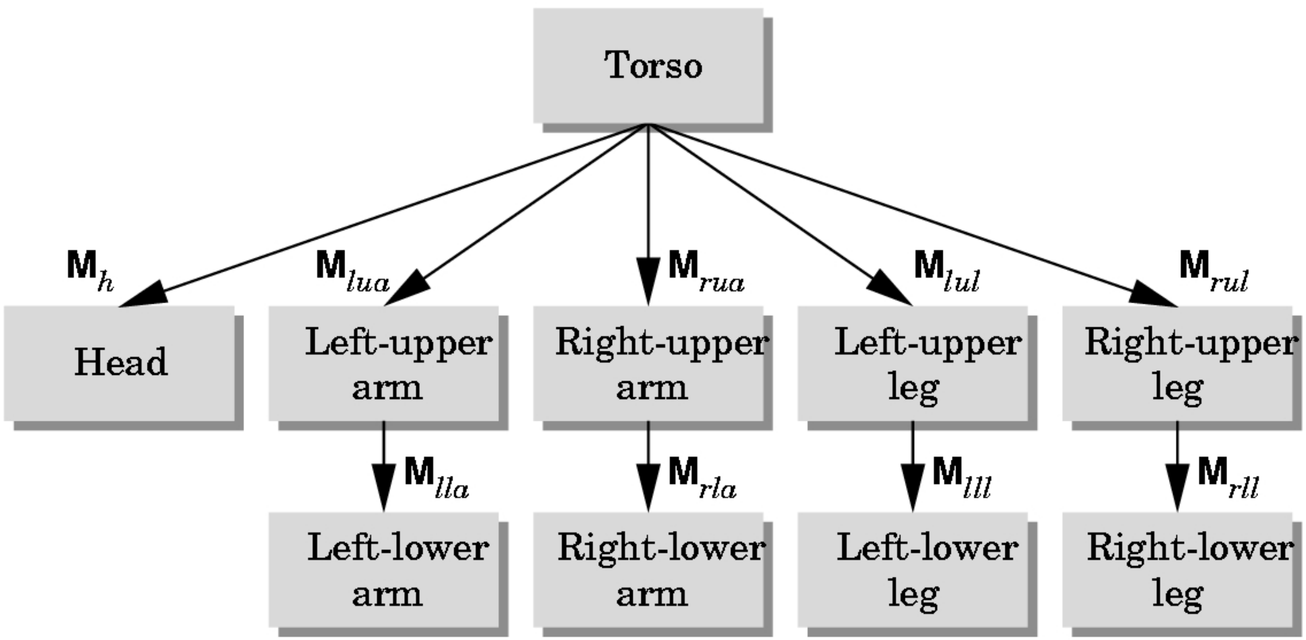
If information is stored in the nodes (not in edges), each node must store at least:

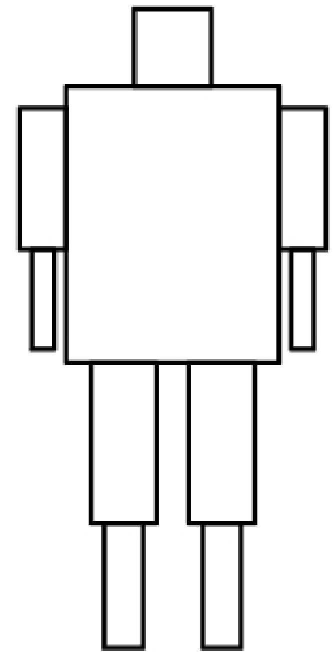
- A pointer to a function that draws the object represented by the node.
- A matrix that positions, orients and scales the object of the node relative to the node's parent (including its children).
- A pointer to its children.





A humanoid model





A humanoid model – building the model

- We can build a simple implementation using quadrics: ellipsoids and cylinders.
- Access parts through functions such as

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

- Matrices describe the position of a node with respect to its parent.

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

A humanoid model – traversal and display

- The position of the figure is determined by 11 joint angles.
- 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, applying the correct transformation matrix for position and orientation.

A humanoid model – transformation matrices

There are 10 relevant matrices.

- $\mathbf{M_t}$ positions and orients the entire figure through the torso which is the root node.
- $\mathbf{M_h}$ positions the head with respect to the torso.
- $\mathbf{M_{lua}}$, $\mathbf{M_{rua}}$, $\mathbf{M_{lul}}$, $\mathbf{M_{rul}}$ position the arms and legs with respect to the torso.
- $\mathbf{M_{lla}}$, $\mathbf{M_{rla}}$, $\mathbf{M_{lll}}$, $\mathbf{M_{rll}}$ position the lower parts of the limbs with respect to the corresponding upper limbs (parents).

A humanoid model – tree and traversal

- All matrices are incremental and any traversal algorithm can be used (depth-first or breadth-first). We can traverse from the left to right and depth first.
- Explicit traversal in the code is performed, using stacks to store required matrices and attributes.
- Recursive traversal code is simpler, and the storage of matrices and attributes is made implicitly.

A humanoid model – stack-based traversal

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

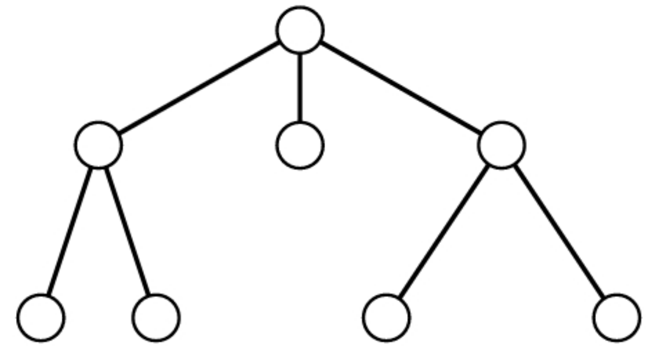
Note that the model-view matrix for the left lower arm is
$\mathbf{M}_{lla\text{-}w} = \mathbf{M}_t \cdot \mathbf{M}_{lua} \cdot \mathbf{M}_{lla}$

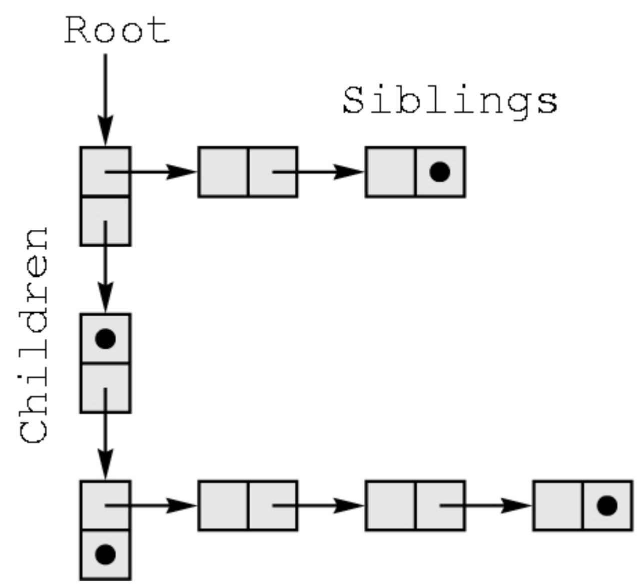
A humanoid model – traversal code

```
void figure() {
   torso();

   glPushMatrix();    // save present MODELVIEW matrix
   glTranslatef();    // update MODELVIEW matrix for the head
   glRotate3();
   head();
   glPopMatrix();     // recover MODELVIEW matrix for the
                      // torso and save the state
   glPushMatrix();
   glTranslatef();    // update MODELVIEW matrix for
   glRotate3();       // the left upper leg
   left_upper_leg();

   glTranslatef();
   glRotate3();       // incremental change for
   left_lower_leg();  // the left_lower_leg
   glPopMatrix();     // recent state recovery
   ...;
}
```

A humanoid model – tree data structure (1)

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

...
treenode torso_node,
         head_node,
         ...;
```





A humanoid model – tree data structure (2)

```
// for the torso
glRotatef(theta[0], 0.0, 1.0, 0.0);
glGetFloatv(GL_MODELVIEW_MATRIX, torso_node.m);
// matrix elements copied to the M of the node

// the torso node has no sibling; and
// the leftmost child is the head node

// rest of the code for the torso node
torso_node.f = torso;
torso_node.sibling = NULL;
torso_node.child = &head_node;
```

A humanoid model – tree data structure (3)

```
// for the upper-arm node
glTranslatef(-(TORSO_RADIUS+UPPER_ARM_RADIUS),
   0.9*TORSO_HEIGHT, 0.0)
glRotatef(theta[3], 1.0, 0.0, 0.0);
glGetFloatv(GL_MODELVIEW_MATRIX, lua_node.m);
// matrix elements copied to the m of the node

lua_node.f = left_upper_arm;
lua_node.sibling = &rua_node;
lua_node.child = &lla_node;
```

A humanoid model – tree data structure (4)

```
// assumption MODELVIEW state
void traverse(treenode* root);
{
    if(root==NULL) return;
    glPushMatrix();
    glMultMatrixf(root->m);
    root->f();
    if(root->child!=NULL) traverse(root->child);
    glPopMatrix();
    if(root->sibling!=NULL) traverse(root->sibling);
} // traversal method is independent of the
  // particular tree!
```

A humanoid model – tree data structure (5)

- We must save model-view matrix (**`glPushMatrix`**) before multiplying it by the node matrix.
- Updated matrix applies to the children of the node.
- But not to its siblings which contain their own matrices; hence we must return to the previous state (**`glPopMatrix`**) before traversing the siblings.
- If we are changing attributes within nodes, we can either push (**`glPushAttrib`**) and pop (**`glPopAttrib`**) attributes within the rendering functions, or push the attributes when we push the model-view matrix.

A humanoid model – tree data structure (6)

```
// generic display callback function
void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT |
            GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();
    traverse(&torso_node);
    glutSwapBuffers();
}
```

Animation can then be implemented by controlling the joint angles (i.e. incremented or decremented) via the mouse or keyboard.

Summary

- Linear modelling does not provide effective ways to retain relationships among the objects of a model.
- Complex models for real world applications can be created and manipulated with hierarchical modelling.
- Hierarchical structure trees are used to implement hierarchical models in computer graphics.
- The code for the humanoid figure is in C (using Struct) and it would be more efficient to write it in C++ (using Class).