Introduction to Computer Graphics with WebGL

Ed Angel

Professor Emeritus of Computer Science

Founding Director, Arts, Research, Technology and Science Laboratory

University of New Mexico

Hierarchical Modeling I

Ed Angel

Professor Emeritus of Computer Science,

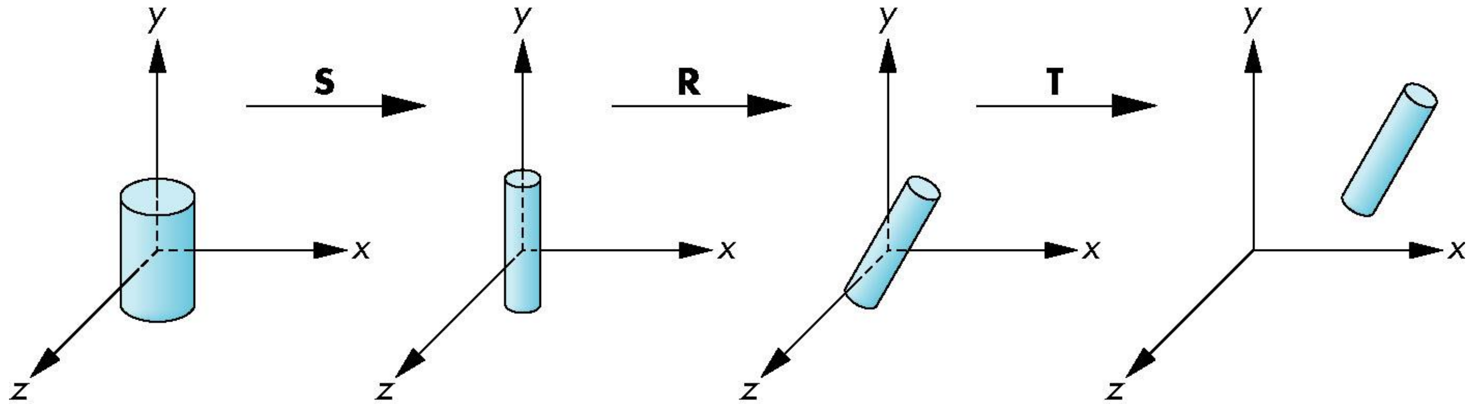
University of New Mexico

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



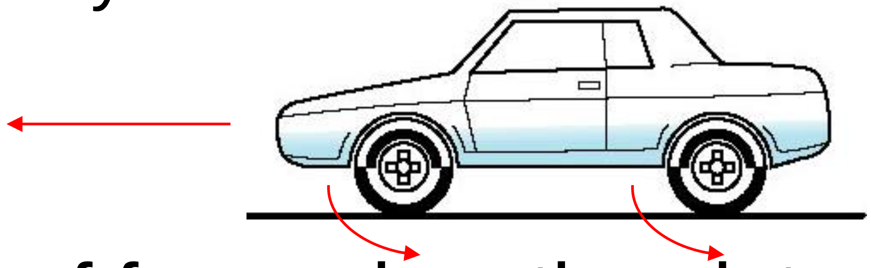
Symbol-Instance Table

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

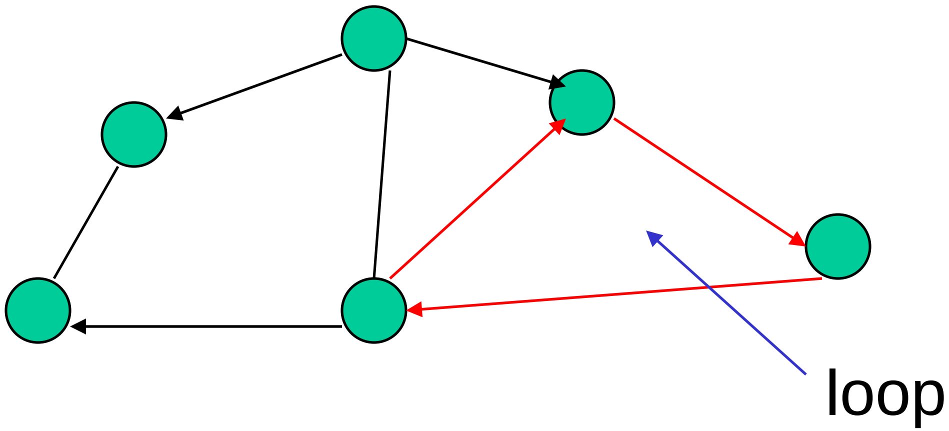
Structure Through Function Calls

```
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

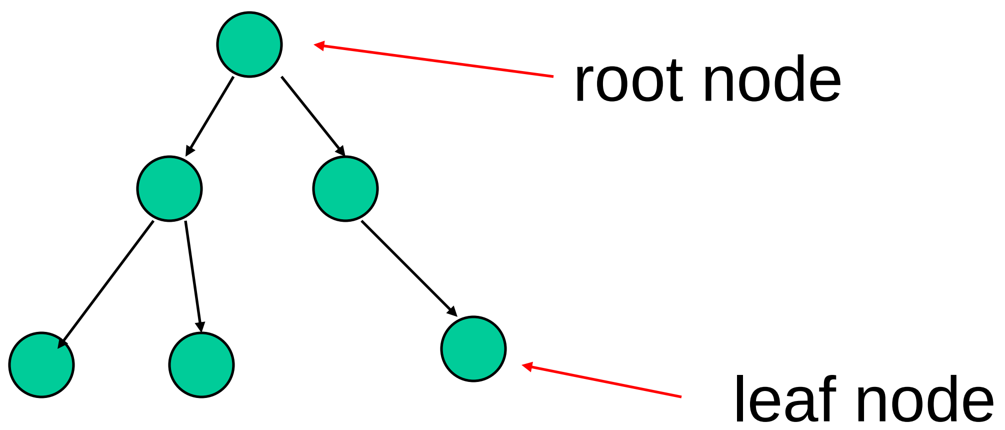
Graphs

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

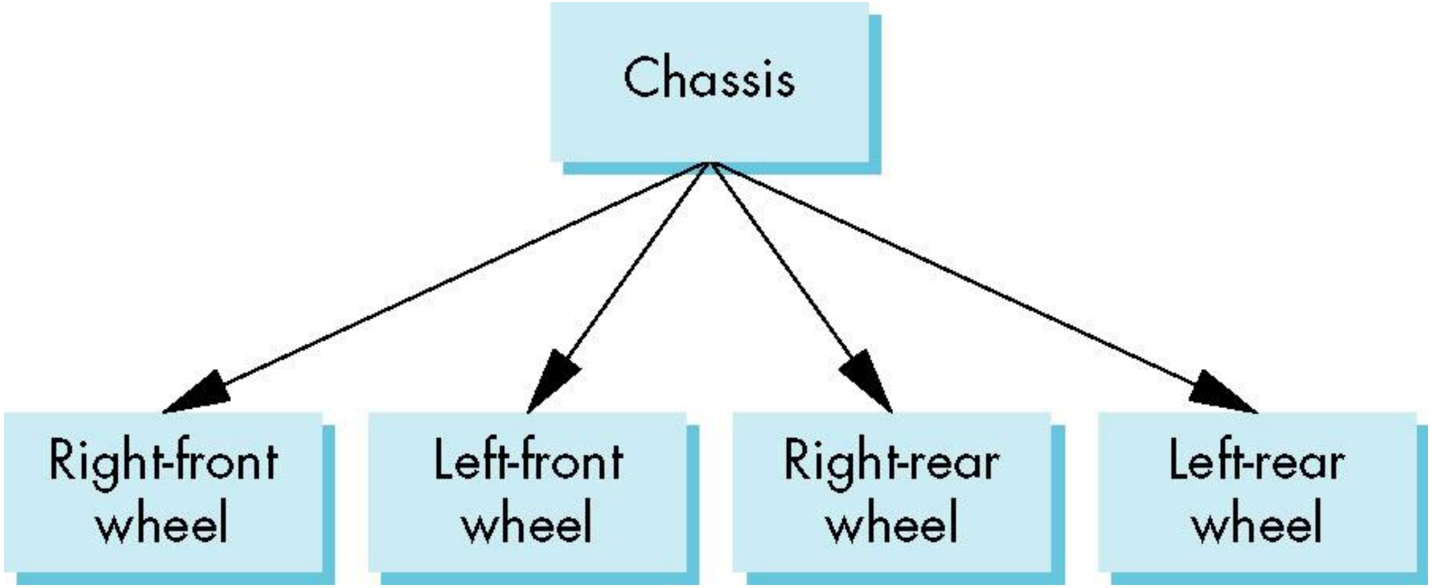


Tree

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

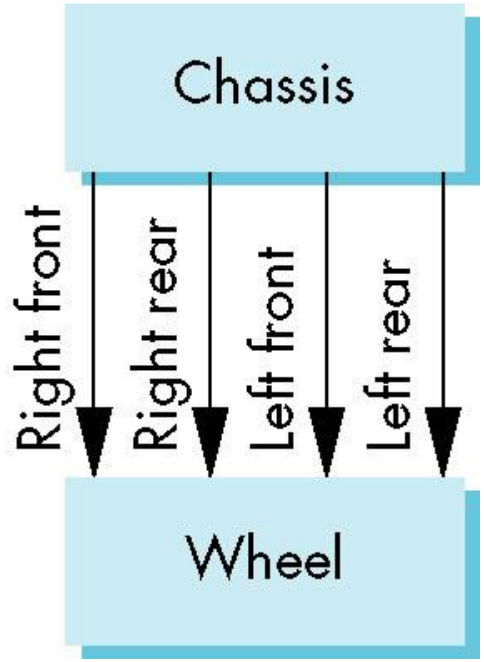


Tree Model of Car



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

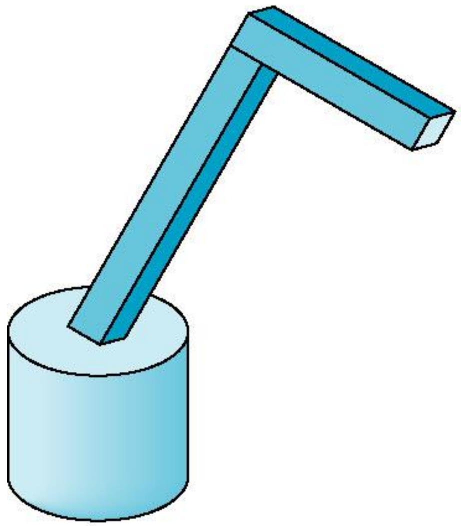


Modeling with Trees

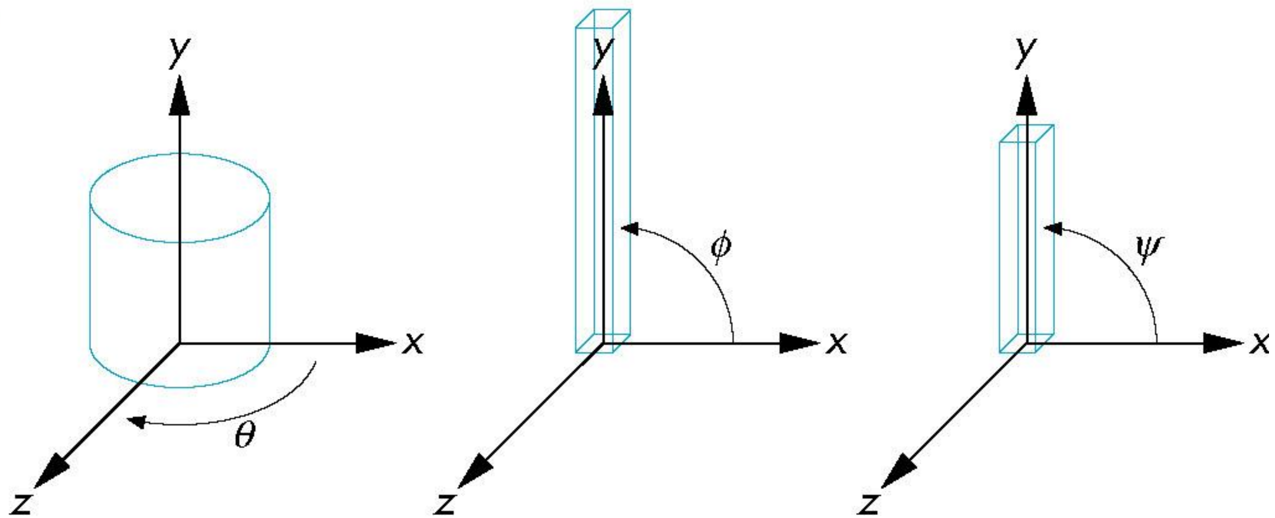
- Must decide what information to place in nodes and what to put in edges
- Nodes
  - What to draw
  - Pointers to children
- Edges
  - May have information on incremental changes to transformation matrices (can also store in nodes)

The University of New Mexico

Robot Arm



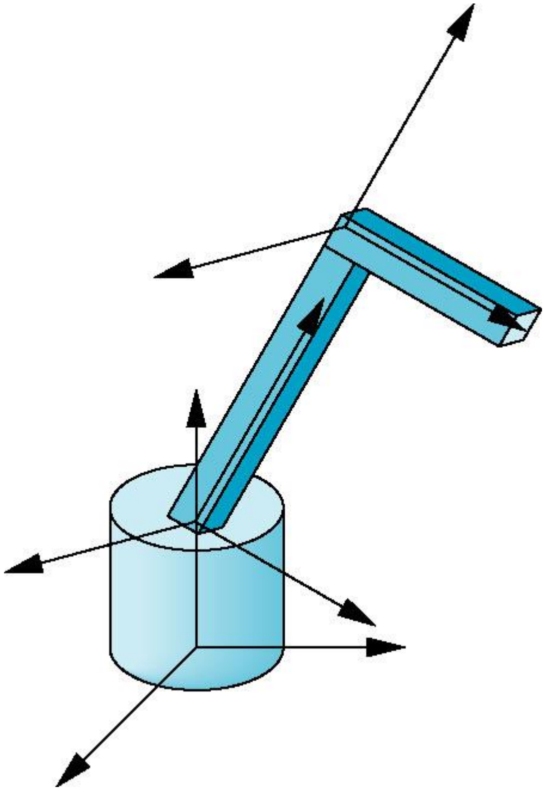
robot arm



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

The University of New Mexico

Required Matrices

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

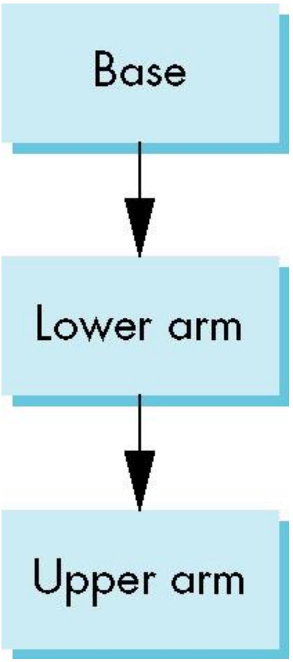
WebGL Code for Robot

```
var render = function() {
   gl.clear( gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT );
   modelViewMatrix = rotate(theta[Base], 0, 1, 0 );
   base();
   modelViewMatrix = mult(modelViewMatrix,
           translate(0.0, BASE_HEIGHT, 0.0));
   modelViewMatrix = mult(modelViewMatrix,
            rotate(theta[LowerArm], 0, 0, 1 ));
   lowerArm();
   modelViewMatrix  = mult(modelViewMatrix,
           translate(0.0, LOWER_ARM_HEIGHT, 0.0));
   modelViewMatrix  = mult(modelViewMatrix,
           rotate(theta[UpperArm], 0, 0, 1) );
   upperArm();
   requestAnimFrame(render);
}
```

Tree Model of Robot

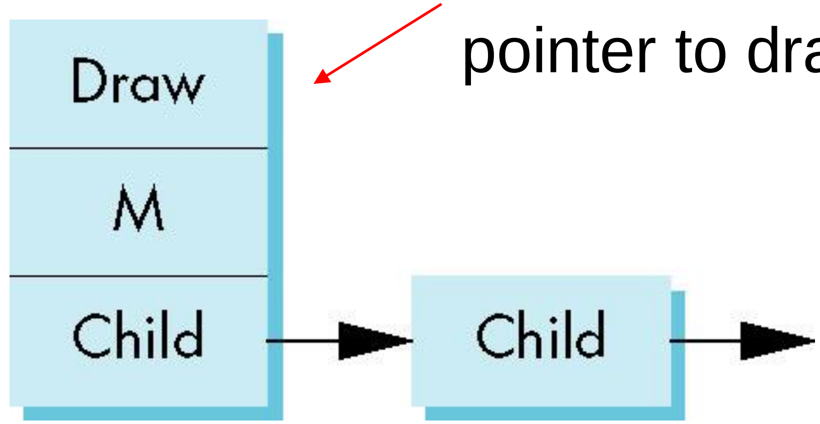
- Note code shows relationships between parts of model
  - Can change "look" of parts easily without altering relationships
- Simple example of tree model
- Want a general node structure

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?

The University of New Mexico

Introduction to Computer Graphics with WebGL

Ed Angel

Professor Emeritus of Computer Science

Founding Director, Arts, Research, Technology and Science Laboratory

University of New Mexico

The University of New Mexico

Hierarchical Modeling II

Ed Angel

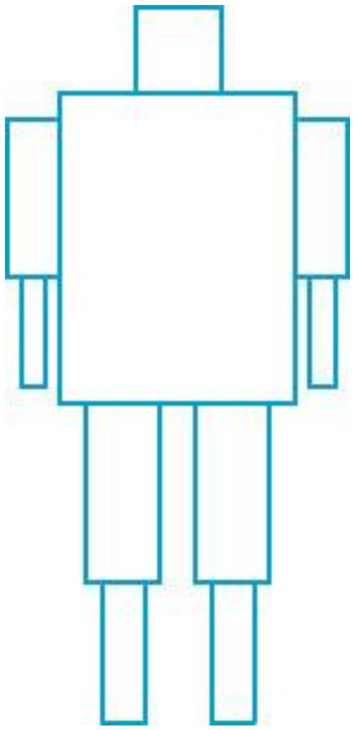
Professor Emeritus of Computer Science

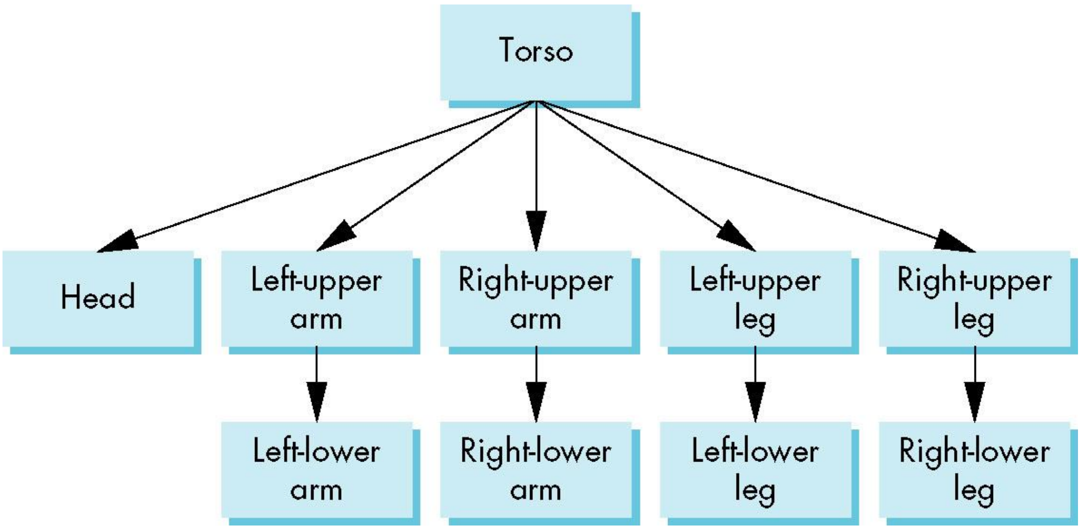
University of New Mexico

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

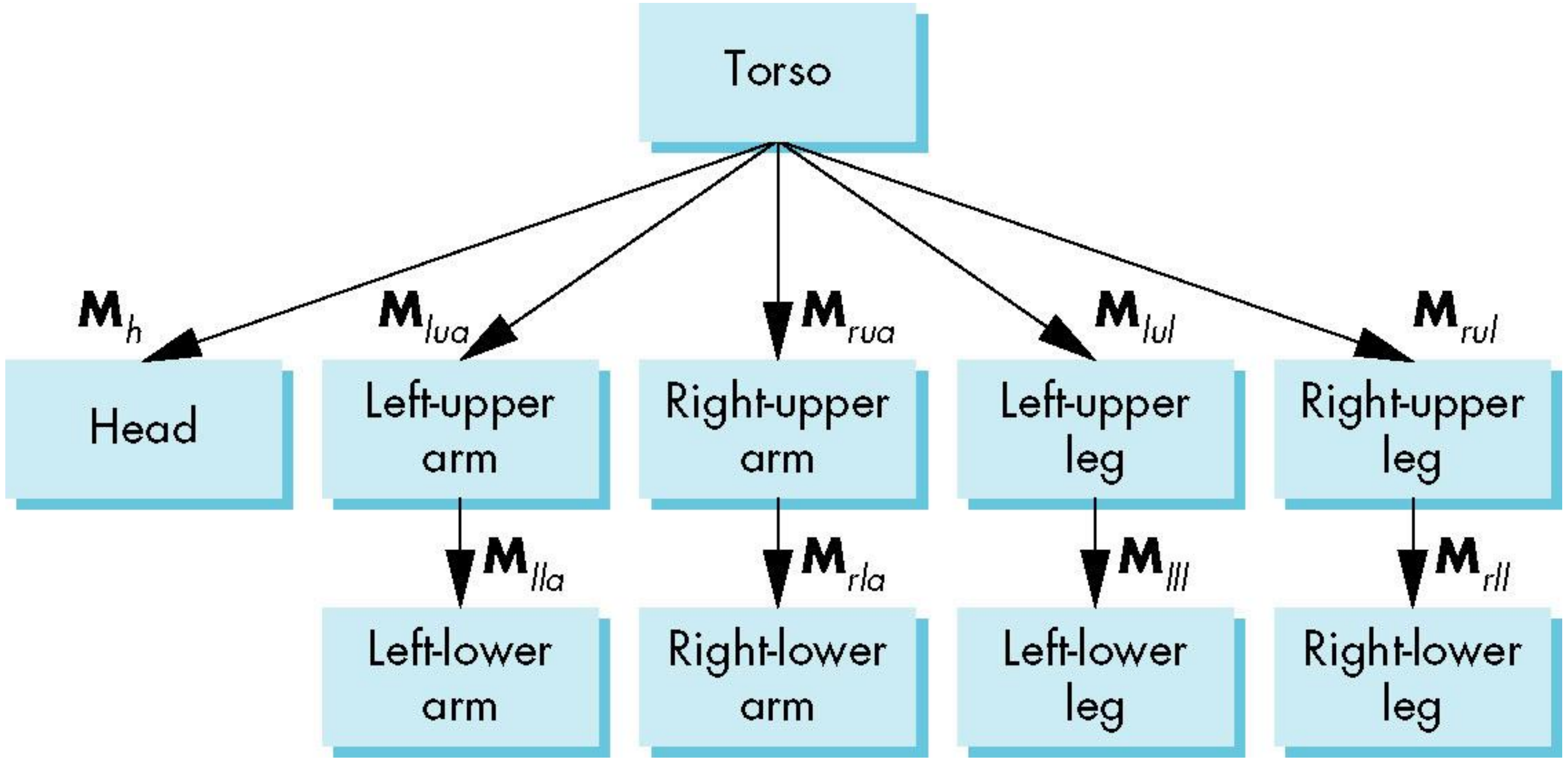




Building the Model

- Can build a simple implementation using quadrics: ellipsoids and cylinders
- Access parts through functions
  - **`torso()`**
  - **`leftUpperArm()`**
- Matrices describe position of node with respect to its parent
  - $\mathbf{M}_{lla}$ positions left lower leg with respect to left upper arm

Tree with Matrices



The University of New Mexico

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

Stack-based Traversal

- Set model-view matrix to $\mathbf{M}$ 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 $\mathbf{M}$ and other matrices as we traverse the tree

Traversal Code

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

save present model-view matrix

update model-view matrix for head

recover original model-view matrix

save it again

update model-view matrix for left upper arm

recover and save original model-view matrix again

rest of code

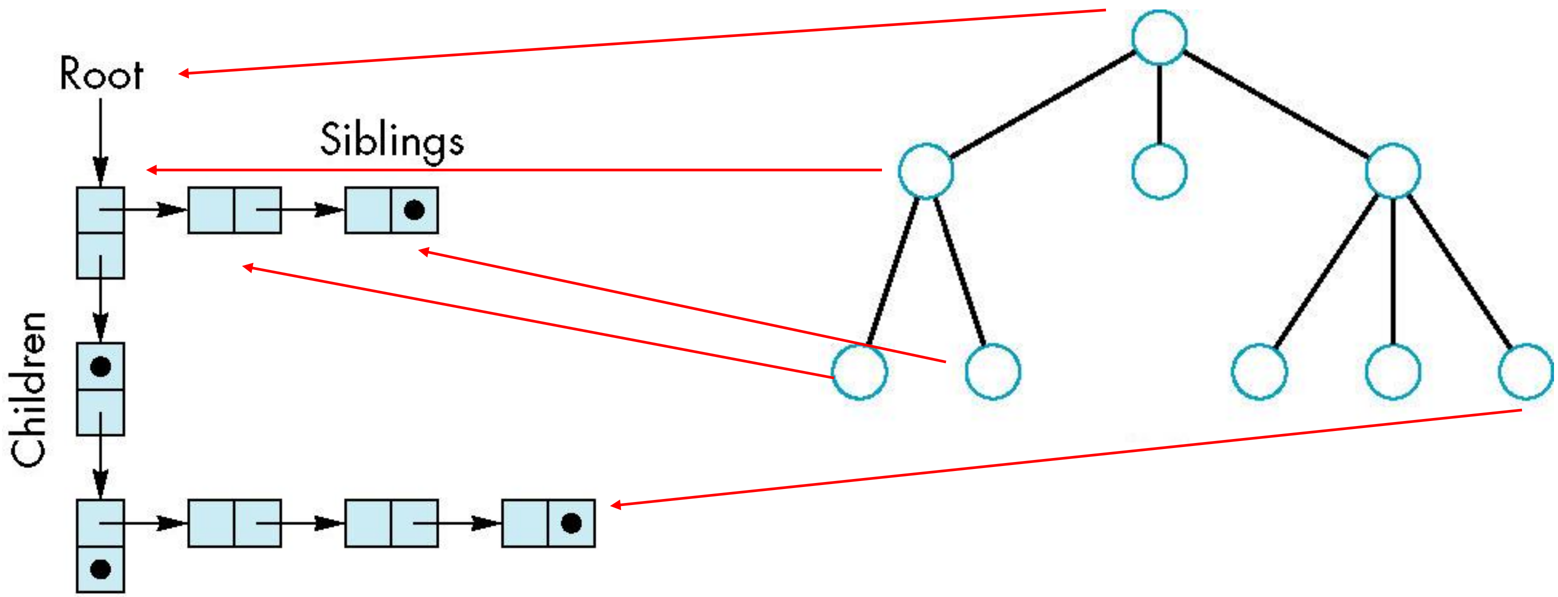
Analysis

- 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 push and pop other attributes 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 WebGL this matrix is a 1D array storing matrix by columns

Creating a treenode

```
function createNode(transform,
            render, sibling, child) {
    var node = {
    transform: transform,
    render: render,
    sibling: sibling,
    child: child,
    }
    return node;
};
```

Initializing Nodes

```
function initNodes(Id) {
    var m = mat4();
        switch(Id) {
        case torsoId:
            m = rotate(theta[torsoId], 0, 1, 0 );
          figure[torsoId] = createNode( m, torso, null, headId );
          break;
      case head1Id:
      case head2Id:
          m = translate(0.0, torsoHeight+0.5*headHeight, 0.0);
          m = mult(m, rotate(theta[head1Id], 1, 0, 0));
          m = mult(m, rotate(theta[head2Id], 0, 1, 0));
          m = mult(m, translate(0.0, -0.5*headHeight, 0.0));
          figure[headId] = createNode( m, head, leftUpperArmId, null);
          break;
```

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 **`rotate`** and **`translate`**
- Because the matrix is formed using the model-view matrix, we may want to first push original model-view matrix on matrix stack

Preorder Traversal

```
function traverse(Id) {
   if(Id == null) return;
   stack.push(modelViewMatrix);
   modelViewMatrix = mult(modelViewMatrix, figure[Id].transform);
   figure[Id].render();
   if(figure[Id].child != null) traverse(figure[Id].child);
   modelViewMatrix = stack.pop();
   if(figure[Id].sibling != null) traverse(figure[Id].sibling);
 }
var render = function() {
        gl.clear( gl.COLOR_BUFFER_BIT );
        traverse(torsoId);
        requestAnimFrame(render);
}
```

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

- Because we are using JS, the nodes and the node structure can be changed during execution
- Definition of nodes and traversal are essentially the same as before but we can add and delete nodes during execution
- In desktop OpenGL, if we use pointers, the structure can be dynamic