Hierarchical Modeling

CS 432 Interactive Computer Graphics
Prof. David E. Breen
Department of Computer Science

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

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Scale</th>
<th>Rotate</th>
<th>Translate</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>s_0, s_1, s_2</td>
<td>x_0, x_1, x_2</td>
<td>y_0, y_1, y_2, z_0, z_1, z_2</td>
</tr>
</tbody>
</table>

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

```c
int 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

• 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)

Transformations to Change Coordinate Systems

• Issue: the world has many different relative frames of reference
• How do we transform among them?
• Example: CAD Assemblies & Animation Models
Transformations to Change Coordinate Systems

- 4 coordinate systems
- 1 point \( P \)
  
\[
M_{1\rightarrow 2} = T(4,2) \\
M_{2\rightarrow 3} = T(2,3) \cdot S(0.5,0.5) \\
M_{3\rightarrow 4} = T(6.7,1.8) \cdot R(45^\circ)
\]

\[
M_{i\rightarrow j} = M_{i\rightarrow j} \cdot M_{j\rightarrow k}
\]

Coordinate System Example (1)

- Translate the House to the origin
  
\[
M_{1\rightarrow 2} = T(x_1, y_1) \\
M_{2\rightarrow 1} = (M_{1\rightarrow 2})^{-1} = T(-x_1, -y_1)
\]

The matrix \( M_j \) that maps points from coordinate system \( j \) to \( i \) is the inverse of the matrix \( M_i \) that maps points from coordinate system \( j \) to coordinate system \( i \).

Coordinate System Example (2)

- Transformation Composition:
  
\[
M_{1\rightarrow 2} = M_{5\rightarrow 4} \cdot M_{4\rightarrow 3} \cdot M_{3\rightarrow 2} \cdot M_{2\rightarrow 1}
\]

World Coordinates and Local Coordinates

- To move the tricycle, we need to know how all of its parts relate to the WCS
- Example: front wheel rotates on the ground wrt the front wheel’s z axis:

\[
P^{(wo)} = T(cw, 0, 0) \cdot R_z(\alpha) \cdot P^{(wh)}
\]

Coordinates of \( P \) in wheel coordinate system:

\[
P^{(wh)} = R_z(\alpha) \cdot P^{(wo)}
\]

Robot Arm

- Robot arm is an example of an articulated model
  - Parts connected at joints
  - Can specify state of model by giving all joint angles

Articulated Models
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: \( R_b \)
  - Apply \( M = R_b \) to base
- Translate lower arm relative to base: \( T_{ls} \)
- Rotate lower arm around joint: \( R_{la} \)
  - Apply \( M = R_b T_{ls} R_{la} \) to lower arm
- Translate upper arm relative to lower arm: \( T_{ua} \)
- Rotate upper arm around joint: \( R_{ua} \)
  - Apply \( M = R_b T_{la} R_{ua} \) to upper arm

WebGL Code for Robot

```javascript
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();
  requestAnimationFrame(render);
};
```

OpenGL Code for Robot

- At each level of hierarchy, calculate ModelView matrix in application.
- Send matrix to shaders
- Draw geometry for one level of hierarchy
- Apply ModelView matrix in shader

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

```
Draw
  ↘ M
  ↘ Child
      ↘ Child
```

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?

Objectives

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

Humanoid Figure

Building the Model

- Can build a simple implementation using quadrics: ellipsoids and cylinders
- Access parts through functions
  - torso()
  - left_upper_arm()
- Matrices describe position of node with respect to its parent
  - \( M_{\text{left lower arm}} \) positions left lower arm with respect to left upper arm

Tree with Matrices

Display and Traversal

- The position of the figure is determined by 10 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
  - Mh positions head with respect to torso
  - Mlua, Mrua, Mlul, Mrul position arms and legs with respect to torso
  - Mlha, Mrha, Mllh, Mrlh 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 MMh and draw head
- For left-upper arm need MMlua and so on
- Rather than recomputing MMlua 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

```c
figure() {
 PushMatrix();
  torso();
  Rotate(...);
  head();
 PopMatrix();

 PushMatrix();
  Translate(...);
  Rotate(...);
  left_upper_arm();
 PopMatrix();
}
```

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 has two pointers
  - Left: linked list of children
  - Right: next node (i.e. siblings)

Left-Child Right-Sibling Tree

- Root
- Group
  - Children
- Group
  - Children
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

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

Initializing Nodes

```javascript
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 10 joint angles stored in `theta[10]`
- 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

Traversal Code & Matrices

```javascript
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);
}
```

```javascript
var render = function() {
    gl.clear(gl.COLOR_BUFFER_BIT);
    traverse(torsoId);
    requestAnimFrame(render);
}
```
Traversals Code & Matrices

- PushMatrix()
- Translate();
- Rotate();
- left_lower_arm();
- PopMatrix();
- PushMatrix();
- Translate();
- Rotate();
- right_upper_arm();

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

Solids and Solid Modeling

- Solid modeling introduces a mathematical theory of solid shape
  - Domain of objects
  - Set of operations on the domain of objects
  - Representation that is
    - Unambiguous
    - Accurate
    - Unique
    - Compact
    - Efficient

Solid Objects and Operations

- Solids are point sets
  - Boundary and interior
- Point sets can be operated on with boolean algebra (union, intersect, etc)

Constructive Solid Geometry (CSG)

- A tree structure combining primitives via regularized boolean operations
- Primitives can be solids or half spaces
A Sequence of Boolean Operations

- Boolean operations
- Rigid transformations

The Induced CSG Tree

- Can also be represented as a directed acyclic graph (DAG)

Issues with Constructive Solid Geometry

- Non-uniqueness
- Choice of primitives
- How to handle more complex modeling?
  - Sculpted surfaces? Deformable objects?

Issues with CSG

- Minor changes in primitive objects greatly affect outcomes
- Shift up top solid face
Uses of Constructive Solid Geometry

- Found (basically) in every CAD system
- Elegant, conceptually and algorithmically appealing
- Good for
  - Rendering, ray tracing, simulation
  - BRL CAD

Graphical Objects and Scene Graphs

Objectives

- Introduce graphical objects
- Generalize the notion of objects to include lights, cameras, attributes
- Introduce scene graphs
- three.js (threejs.org)

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 disappeared from the graphical system
- To redraw the object, either changed or the same, we had to reexecute the code
- Display lists provided only a partial solution to this problem

Retained Mode Graphics

- Display lists were server side
- GPUs allowed data to be stored on GPU
- Essentially all immediate mode functions have been deprecated
- Nevertheless, OpenGL is a low level API
OpenGL and Objects

- OpenGL lacks an object orientation
- Consider, for example, a green sphere
  - We can model the sphere with polygons
  - Its color is determined by the OpenGL state and is not a property of the object
  - Loose linkage with vertex attributes
- Defies our notion of a physical object
- We can try to build better objects in code using object-oriented languages/techniques

Object-Oriented Programming Model

- In this model, the representation is stored with the object
  - Application sends a message to the object
  - The object contains functions (methods) which allow it to transform itself

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
  ```javascript
  var mycube = new Cube();
  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
  ```javascript
  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
  ```javascript
  mycube.render();
  ```

Imperative Programming Model

- Example: rotate a cube
  - The rotation function must know how the cube is represented
    - Vertex list
    - Edge list

C/C++/Java/JS

- Can try to use C structs to build objects
- C++/Java/JS provide better support
  - Use class construct
  - With C++ we can hide implementation using public, private, and protected members
  - JS provides multiple methods for object
Building the Cube Object

```javascript
var cube {
    var color[3];
    var matrix[4][4];
}
```

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

Other Objects

- Other objects have geometric aspects
  - Cameras
  - Light sources
- But we should be able to have nongeometric objects too
  - Materials
  - Colors
  - Transformations (matrices)

JS Objects

```javascript
cube mycube;
material plastic;
mycube.setMaterial(plastic);
camera frontView;
frontView.position(x, y, z);
```

Light Object

```javascript
var myLight = new Light();

// match Phong model
myLight.type = 0; //directional
myLight.position = ...;
myLight.orientation = ...;
myLight.specular = ...;
myLight.diffuse = ...;
myLight.ambient = ...;
```
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 JS objects, we should be able to show them in a tree
  - Render scene by traversing this tree

Traversal

```javascript
myScene = new Scene();
myLight = new Light();
myLight.Color = ....;
...
myscene.Add(myLight);
object1 = new Object();
object1.color = ...
myscene.add(object1);
...

myscene.render();
```

Scene Graph

```
Scene
  | Light          |
  v
Object 1
  | Color |
  v
Object 2
  | Material |
  v
Camera
  v
  | Position |
  v
Position
  v
Instance
  v
Instance
  v
Rotate
  v
Clip
```

Scene Graph History

• OpenGL development based largely on people who wanted to exploit hardware
  - real time graphics
  - animation and simulation
  - stand-alone applications
• CAD community needed to be able to share databases
  - real time not and photorealism not issues
  - need cross-platform capability
  - first attempt: PHIGS

Scene Graph Organization

```
Scene Graph
  v
Scene Graph API
  v
Database
  v
WebGL
  v
OpenGL
  v
Direct X
  v
WWW
```

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, WebGL or DirectX (for PCs)
VRML

- Want to have a scene graph that can be used over the World Wide Web
- Need links to other sites to support distributed data bases
- **Virtual Reality Markup Language**
  - Based on Inventor data base
  - Implemented with OpenGL

Open Scene Graph

- Supports very complex geometries by adding occlusion culling in first pass
- 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

three.js

- Popular scene graph built on top of WebGL
  - also supports other renderers
- See threejs.org
  - easy to download
  - many examples
- Also Eric Haines' Udacity course
- Major differences in approaches to computer graphics

three.js scene

```javascript
var scene = new THREE.Scene();
var camera = new THREE.PerspectiveCamera(75, window.innerWidth/ window.innerHeight, 0.1, 1000);

var renderer = new THREE.WebGLRenderer();
renderer.setSize(window.innerWidth, window.innerHeight);
document.body.appendChild(renderer.domElement);

var geometry = new THREE.CubeGeometry(1,1,1);
var material = new THREE.MeshBasicMaterial({color: 0x00ff00});
var cube = new THREE.Mesh(geometry, material);
scene.add(cube);

camera.position.z = 5;
```

three.js render loop

```javascript
var render = function () {
  requestAnimationFrame(render);
  cube.rotation.x += 0.1;
  cube.rotation.y += 0.1;
  renderer.render(scene, camera);
  renderer.render(scene, camera);
};
render();
```