Hierarchical Modeling and Scene Graphs

Adapted from material prepared by Ed Angel
Objectives

- Examine the limitations of linear modeling
  - Symbols and instances
- Introduce hierarchical models
  - Articulated models
  - Robots
- Introduce Tree and DAG models
- Examine various traversal strategies
- Generalize the notion of objects to include lights, cameras, attributes
- Introduce scene graphs
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 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

- Chassis
  - Right-front wheel
  - Left-front wheel
  - Right-rear wheel
  - Left-rear wheel
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 coordinate 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: $R_b$
  - Apply $M = R_b$ to base
- Translate lower arm relative to base: $T_{lu}$
- Rotate lower arm around joint: $R_{lu}$
  - Apply $M = R_b T_{lu} R_{lu}$ to lower arm
- Translate upper arm relative to upper arm: $T_{uu}$
- Rotate upper arm around joint: $R_{uu}$
  - Apply $M = R_b T_{lu} R_{lu} T_{uu} R_{uu}$ to upper arm
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();
}
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
Humanoid Figure

Diagram showing the hierarchical structure of a humanoid figure with nodes for Torso, Head, Left-upper arm, Right-upper arm, Left-upper leg, Right-upper leg, Left-lower arm, Right-lower arm, Left-lower leg, and Right-lower leg.
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
- \( M_{lla} \) positions left lower leg with respect to left upper arm
Tree with Matrices

- Torso
  - Left-upper arm
    - Left-lower arm
  - Right-upper arm
  - Left-upper leg
  - Right-upper leg
  - Left-lower leg
  - Right-lower leg
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
  - $M_h$ positions head with respect to torso
  - $M_{lua}$, $M_{rua}$, $M_{lul}$, $M_{rul}$ position arms and legs with respect to torso
  - $M_{lla}$, $M_{rla}$, $M_{lll}$, $M_{rll}$ 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 $MM_h$ and draw head
- For left-upper arm need $MM_{lua}$ and so on
- Rather than recomputing $MM_{lua}$ from scratch or using an inverse matrix, we can use the matrix stack to store $M$ and other matrices as we traverse the tree
Traversing Code

```c
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
}```
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.
The position of figure is determined by 11 joint angles stored in \texttt{theta[11]}

Animate by changing the angles and redisplaying

We form the required matrices using \texttt{glRotate} and \texttt{glTranslate}

Because the matrix is formed in model-view matrix, we may want to first push original model-view matrix on matrix stack
void traverse(treenode *root) {
    if (root == NULL) return;
    glPushMatrix();
    glLoadIdentity();
    glMultMatrix(root->m);
    root->f();
    if (root->child != NULL)
        traverse(root->child);
    glPopMatrix();
    if (root->sibling != NULL)
        traverse(root->sibling);
}
- 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
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
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
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]=........
We would also like to have functions that act on the cube such as

```java
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

```java
mycube.render();
```
```cpp
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)
cube mycube;
material plastic;
mycube.setMaterial(plastic);

camera frontView;
frontView.position(x, y, z);
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

Separator

Color

Translate

Rotate

Object 1

Translate

Object 2

Separator

Translate

Object 3

Rotate
Preorder Traversal

```c
glPushAttrib
glPushMatrix
glColor
glTranslate
glRotate
Object1
glTranslate
Object2
glPopMatrix
```
Separator Nodes

- Necessary to isolate state changes
  - 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 separator node, state changes can propagate
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
- **Virtual Reality Markup Language**
  - Based on Inventor data base
  - Implemented with OpenGL