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>2</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>5</td>
<td>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
void 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

DAG Model

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 \( P \)

\[
M_{1\rightarrow 2} = T(4,2) \\
M_{2\rightarrow 3} = T(2,3) \cdot S(0.5,0.5) \\
M_{3\rightarrow 4} = T(6.7,1.8) \cdot R(45\degree)
\]

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

Coordinate System Example (1)

- Translate the House to the origin

\[
M_{x_1, y_1} = T(x_1, y_1) \\
M_{x_2, y_2} = (M_{x_1, y_1})^{-1} = T(-x_1, -y_1)
\]

The matrix \( M_{ij} \) that maps points from coordinate system \( j \) to \( i \) is the inverse of the matrix \( M_{ji} \) that maps points from coordinate system \( j \) to coordinate system \( i \).

Coordinate System Example (2)

- Transformation Composition:

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

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:

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

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

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

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: \( T_{lu} \)
• 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

OpenGL Code for Robot

```c
mat4 ctm; // current transformation matrix
robot_arm() {
  ctm = RotateY(theta);  // base rotation
  ctm *= Translate(0.0, h1, 0.0);  // translate base
  ctm *= RotateZ(phi);  // rotate on base
  ctm = Translate(0.0, h2, 0.0);  // translate lower arm
  ctm *= RotateZ(psi);  // rotate lower arm
  upper_arm();
}
```

OpenGL Code for Robot

• At each level of hierarchy, calculate \( ctm \) matrix in application.
• Send matrix to shaders
• Draw geometry for one level of hierarchy
• Apply \( ctm \) 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

- 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:
  - $M_{lla}$ positions left lower arm with respect to left upper arm

**Tree with Matrices**

**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
  - $M_{\text{positions}}$ positions and orients entire figure through the torso which is the root node
  - $M_{\text{head}}$ positions head with respect to torso
  - $M_{\text{arms}}, M_{\text{legs}}, M_{\text{limbs}}$ position arms and legs with respect to torso
  - $M_{\text{lower limbs}}$: 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 $MM_{\text{head}}$ and draw head
- For left-upper arm need $MM_{\text{arms}}$ and so on
- Rather than recomputing $MM_{\text{arms}}$ 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();
    PushMatrix();
    update ctm for head
    recover original ctm
    save it again
    update ctm for left upper arm
    recover and save original ctm 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 use a `PushAttrib` and `PopAttrib` 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: linked list of children
  - Right: next node (i.e. siblings)

Left-Child Right-Sibling Tree

![Diagram of a left-child right-sibling tree structure]
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 OpenGL, this matrix is a 1D array storing matrix by columns

C Definition of treenode

typedef struct treenode
{
    mat4 m;
    void (*f)();
    struct treenode *sibling;
    struct treenode *child;
} treenode;

Notes

- The position of the 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{Rotate} and \texttt{Translate}
- Because the matrix is formed using the model-view matrix, we may want to first push original model-view matrix on matrix stack

torso and head nodes

treenode torso_node, head_node, lua_node, ...;

torso_node.m = \texttt{RotateY(theta[0])};
torso_node.f = torso;
torso_node.sibling = NULL;
torso_node.child = &head_node;

head_node.m = \texttt{translate(0.0, TORSO_HEIGHT +0.5*HEAD_HEIGHT, 0.0)*RotateX(theta[1])}
* \texttt{RotateY(theta[2])};
head_node.f = head;
head_node.sibling = &lua_node;
head_node.child = NULL;

Preorder Traversal

void traverse(treenode* root)
{
    if(root==NULL) return;
    mstack.push(ctm);
    ctm = ctm*root->m;
    root->f();
    if(root->child!=NULL) traverse(root->child);
    ctm = mstack.pop();
    if(root->sibling!=NULL) traverse(root->sibling);
}

Traversal Code & Matrices

- \texttt{figure()} called with CTM set
- \texttt{M}_{fig} defines figure's place in world

\begin{align*}
\texttt{figure}() & \quad \text{Stack} \quad \text{CTM} \\
\texttt{PushMatrix()} & \quad \text{Stack} \quad \text{M}_{fig} \\
\texttt{torso()} & \quad \text{Stack} \quad \text{M}_{fig}M_{head} \\
\texttt{Rotate(...)} & \quad \text{Stack} \quad \text{CTM} \\
\texttt{head()} & \quad \text{Stack} \quad \text{M}_{fig}M_{head} \\
\texttt{PopMatrix()} & \quad \text{Stack} \quad \text{CTM} \\
\texttt{PushMatrix()} & \quad \text{Stack} \quad \text{M}_{fig}M_{head}M_{left_upper_arm} \\
\texttt{Translate(...)} & \quad \text{Stack} \quad \text{M}_{fig}M_{head}M_{left_upper_arm} \\
\texttt{Rotate(...)} & \quad \text{Stack} \quad \text{CTM} \\
\texttt{left_upper_arm()} & \quad \text{Stack} \quad \text{M}_{fig}M_{head}M_{left_upper_arm} \\
\end{align*}
Traversal Code & Matrices

```c
void PushMatrix() {
    Stack = CTM;
    M_id = M_id * M_id;
}
void Translate(…);
void Rotate(…);
void left_lower_arm();
void PopMatrix();
void right_upper_arm();
Stack = CTM;
M_id = M_id * M_id;
void PushMatrix();
void Translate(…);
void Rotate(…);
Stack = CTM;
M_id = M_id * M_id;
```

Notes

- We must save current transformation 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

- If we use pointers, the structure can be dynamic
  ```c
typedef treenode *tree_ptr;
tree_ptr torso_ptr;
torus_ptr = malloc(sizeof(treenode));
```
- Definition of nodes and traversal are essentially the same as before but we can add and delete nodes during execution

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