CS 4204 Computer Graphics

Structured Graphics and Hierarchical Modeling

Adapted from slides by Yong Cao
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>$s_x$, $s_y$, $s_z$</td>
<td>$\theta_x$, $\theta_y$, $\theta_z$</td>
<td>$d_x$, $d_y$, $d_z$</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.</td>
<td></td>
<td></td>
<td></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

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

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 Diagram]

- **root node**
- **leaf node**
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
OpenGL Code for Robot

robot_arm()
{
    glRotate(theta, 0.0, 1.0, 0.0);
    base();
    glTranslate(0.0, h1, 0.0);
    glRotate(phi, 0.0, 1.0, 0.0);
    lower_arm();
    glTranslate(0.0, h2, 0.0);
    glRotate(psi, 0.0, 1.0, 0.0);
    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**

<table>
<thead>
<tr>
<th>Base</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lower arm</td>
</tr>
<tr>
<td>Upper arm</td>
</tr>
</tbody>
</table>
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

![Diagram of a human figure showing parts connected to the torso.](image)
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

- $M$ positions and orients entire figure through the torso which is the root node
- $M_h$ positions head with respect to torso
- $M_{lue}, M_{rue}, M_{lul}, M_{rul}$ position arms and legs with respect to torso
- $M_{llu}, M_{rlu}, M_{lrl}, M_{rul}$ 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
Traversal 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
    glPushMatrix();
    ...
    glPushMatrix();
    ...
    glPushMatrix();
    ...
    glPushMatrix();
    ...
}
```
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 `glPushAttrib` and `glPopAttrib` 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 OpenGL this matrix is a 1D array storing matrix by columns
typedef struct treenode
{
    GLfloat m[16];
    void (*f)();
    struct treenode *sibling;
    struct treenode *child;
} treenode;
Defining the torso node

treenode torso_node, head_node, lua_node, ... ;
/* use OpenGL functions to form matrix */
glLoadIdentity();
glRotatef(theta[0], 0.0, 1.0, 0.0);
/* move model-view matrix to m */
glGetFloatv(GL_MODELVIEW_MATRIX, torso_node.m)

torso_node.f = torso; /* torso() draws torso */
Torso_node.sibling = NULL;
Torso_node.child = &head_node;
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 `glRotate` and `glTranslate`.

- More efficient than manual calculation.
- Because the matrix is formed in model-view matrix, we may want to first push original model-view matrix on matrix stack.
Preorder Traversal

```c
void traverse(treenode *root)
{
    if(root == NULL) return;
    glPushMatrix();
    glMultMatrix(root->m);
    root->f();
    if(root->child != NULL)
        traverse(root->child);
    glPopMatrix();
    if(root->sibling != NULL)
        traverse(root->sibling);
}
```
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

*If we use pointers, the structure can be dynamic*

typedef treenode *tree_ptr;

tree_ptr torso_ptr;

torso_ptr = malloc(sizeof(treenode));

*Definition of nodes and traversal are essentially the same as before but we can add and delete nodes during execution*