Hierarchical Modeling

Symbols and instances
Most graphics APIs support a few geometric primitives:
- spheres
- cubes
- cylinders

These symbols are instanced using an instance transformation.

Instancing in OpenGL
In OpenGL, instancing is created by modifying the model-view matrix:
```c
GLuint mode( GL_MODELVIEW );
GLuint identity();
GLuint translate(...);
GLuint rotate(...);
GLuint scale(...);
house();
```

Do the transforms seem to be backwards? Why was OpenGL designed this way?

Instancing in real OpenGL
The advantage of right-multiplication is that it places the earlier transforms closer to the primitive.
```c
glPushMatrix();
glTranslatef(...);
glRotatef(...);
house();
```
```c
glPushMatrix();
glTranslatef(...);
glRotatef(...);
house();
glPopMatrix();
```
3D Example: A robot arm

Consider this robot arm with 3 degrees of freedom:
- Base rotates about its vertical axis by $\theta$
- Lower arm rotates in its $xy$-plane by $\phi$
- Upper arm rotates in its $xy$-plane by $\psi$

Q: What matrix do we use to transform the base?
Q: What matrix for the lower arm?
Q: What matrix for the upper arm?

Robot arm implementation

The robot arm can be displayed by keeping a global matrix and computing it at each step:

```c
robot_arm() {
    M_model = R_y(theta);
    base();
    M_model = R_y(theta)* T(0,h1,0)* R_z(phi);
    upper_arm();
    M_model = R_y(theta)* T(0,h1,0)* R_z(phi)* T(0,h2,0)* R_z(psi);
    lower_arm();
}
```

Do the matrix computations seem wasteful?

Robot arm implementation, better

Instead of recalculating the global matrix each time, we can just update it in place:

```c
robot_arm() {
    M_model = identity();
    robot_arm();
    M_model *= R_y(theta);
    base();
    M_model *= T(0,h1,0)* R_z(phi);
    upper_arm();
    M_model *= T(0,h2,0)* R_z(psi);
    lower_arm();
}
```
OpenGL maintains a global state matrix called the model-view matrix.

```c
main() {
    ... glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    robot_arm(a, b, c);
    ...
}
robot_arm(theta, phi, psi) {
    glTranslatef(theta, 0.0, 0.0);
    base();
    glRotatef(phi, 0.0, 0.0, 1.0);
    lower_arm();
    glTranslatef(0.0, h2, 0.0);
    glRotatef(psi, 0.0, 0.0, 1.0);
    upper_arm();
}
```

Hierarchical modeling

Hierarchical models can be composed of instances using trees or DAGs:

- edges contain geometric transformations
- nodes contain geometry (and possibly drawing attributes)

Hierarchical models can be composed of instances using trees or DAGs:

A complex example: human figure

Human figure implementation

The traversal can be implemented by saving the model-view matrix on a stack:

```c
figure() {
    glPushMatrix();
    glTranslatef(...);
    torso();
    glPushMatrix();
    ...;
    head();
    ...;
    ...;
    head();
    ...;
    ...;
    ...
}
```
Animation
The above examples are called **articulated models**:
- rigid parts
- connected by joints
They can be animated by specifying the joint angles (or other display parameters) as functions of time.

Scene graphs
The idea of hierarchical modeling can be extended to an entire scene, encompassing:
- many different objects
- lights
- camera position
This is called a **scene tree** or **scene graph**.

Summary
Here’s what you should take home from this lecture:
- How primitives can be instanced and composed to create hierarchical models using geometric transforms.
- How transforms can be thought of as affecting either the geometry, or the coordinate system which it is drawn in.
- How the notion of a model tree or DAG can be extended to entire scenes.