4. Hierarchical Modeling

Reading

Required:

- Angel, sections 8.1 - 8.6

Optional:

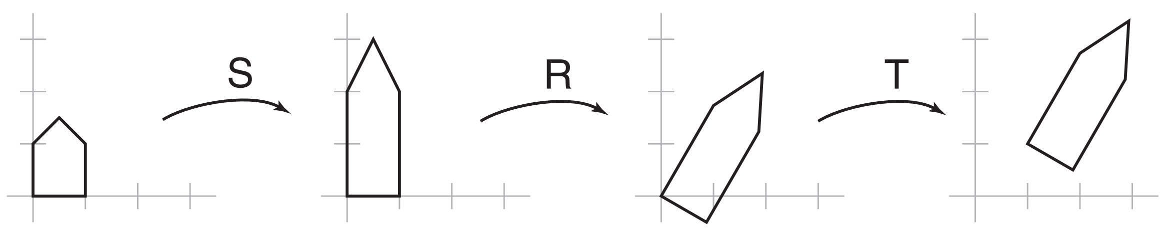
- *OpenGL Programming Guide*, chapter 3

Symbols and instances

Most graphics APIs support a few geometric **primitives**:

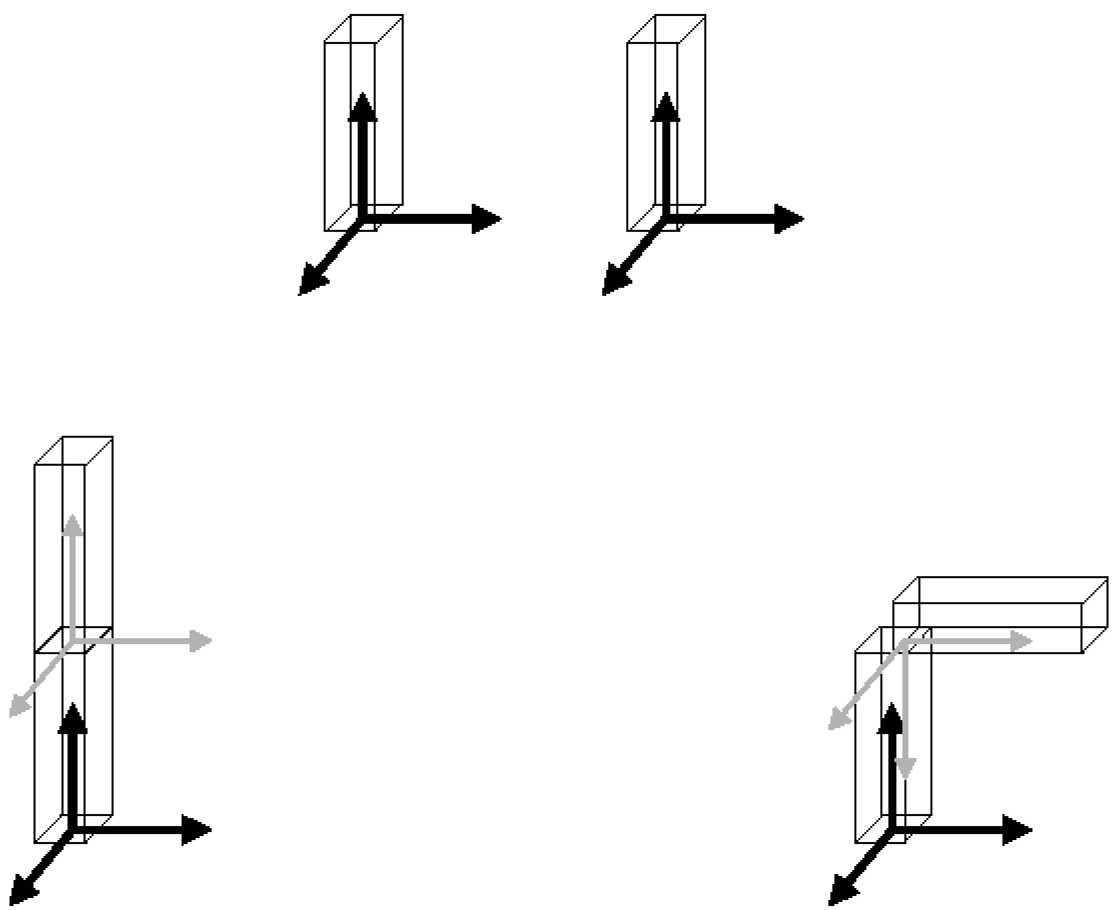
- spheres
- cubes
- cylinders

These symbols are **instanced** using an **instance transformation**.



Q: What is the matrix for the instance transformation above?

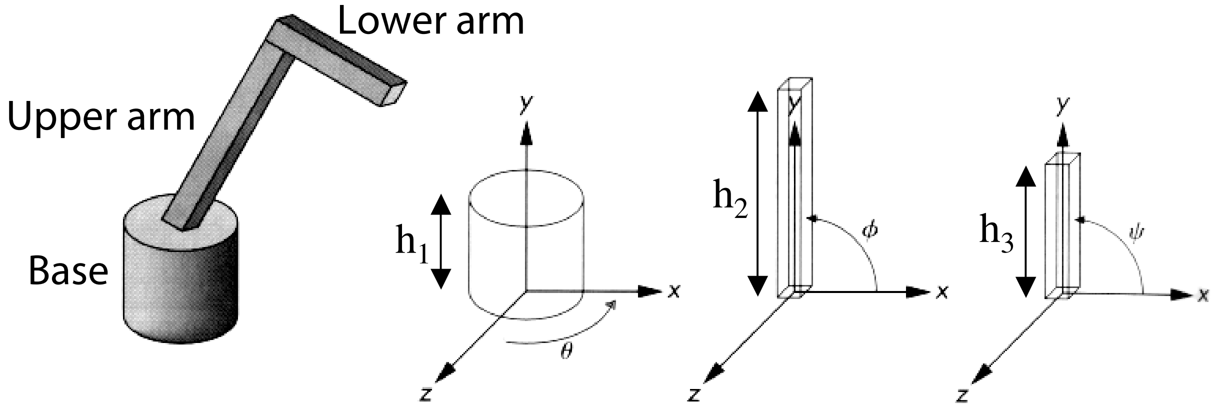
Connecting primitives



3D Example: A robot arm

Consider this robot arm with 3 degrees of freedom:

- Base rotates about its vertical axis by θ
- Upper arm rotates in its *xy*-plane by ϕ
- Lower arm rotates in its *xy*-plane by ψ



Q: What matrix do we use to transform the base?

Q: What matrix for the upper arm?

Q: What matrix for the lower arm?

Robot arm implementation

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

```
Matrix M_model;

main()
{
     . . .
     robot_arm();
     . . .
}

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*:

```
Matrix M_model;

main()
{
    . . .
    M_model = Identity();
    robot_arm();
    . . .
}

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();
}
```

Robot arm implementation, OpenGL

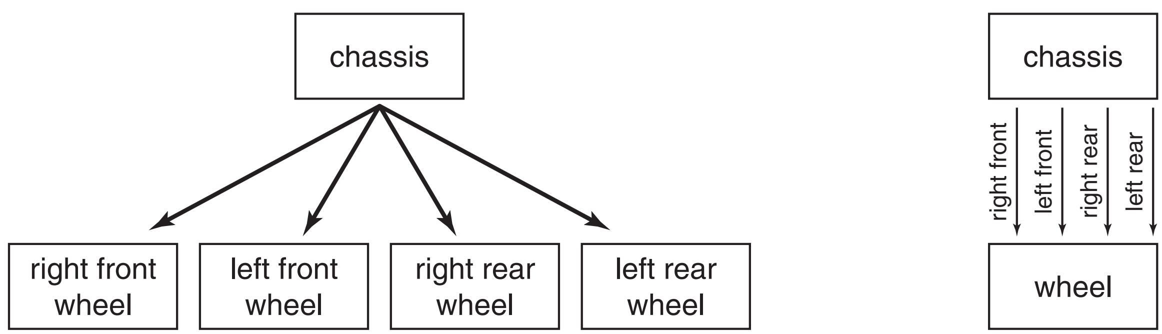
OpenGL maintains a global state matrix called the **model-view matrix**.

```
main()
{
    . . .
    glMatrixMode( GL_MODELVIEW );
    glLoadIdentity();
    robot_arm();
    . . .
}

robot_arm()
{
    glRotatef( theta, 0.0, 1.0, 0.0 );
    base();
    glTranslatef( 0.0, h1, 0.0 );
    glRotatef( phi, 0.0, 0.0, 1.0 );
    upper_arm();
    glTranslatef( 0.0, h2, 0.0 );
    glRotatef( psi, 0.0, 0.0, 1.0 );
    lower_arm();
}
```

Hierarchical modeling

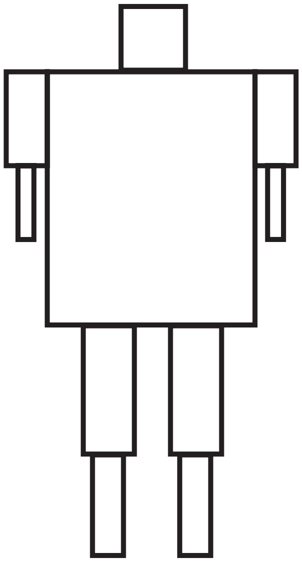
Hierarchical models can be composed of instances using trees or DAGs:

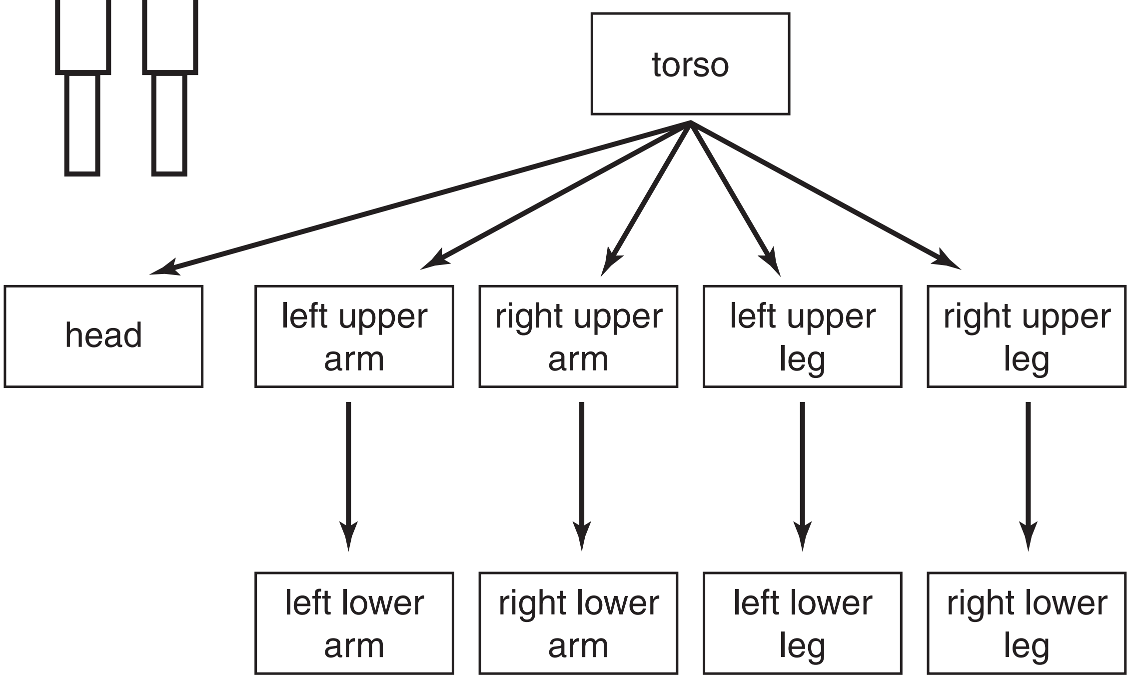


- edges contain geometric transformations
- nodes contain geometry (and possibly drawing attributes)

How might we draw the tree for the robot arm?

A complex example: human figure





Q: What's the most sensible way to traverse this tree?

Human figure implementation

We can also design code for drawing the human figure, with a slight modification due to the branches in the tree:

```
figure()
{
    torso();
    M_save = M_model;
    M_model *= T(. . .)*R(. . .);
    head();
    M_model = M_save;
    M_model *= T(. . .)*R(. . .);
    left_upper_arm();
    M_model *= T(. . .)*R(. . .);
    left_lower_arm();
    M_model = M_save;
    .
    .
    .
}
```

Human figure with hand

What if we add a hand?

```
figure()
{
    torso();
    M_save = M_model;
    M_model *= T(. . .)*R(. . .);
    head();
    M_model = M_save;
    M_model *= T(. . .)*R(. . .);
    left_upper_arm();
    M_model *= T(. . .)*R(. . .);
    left_lower_arm();
    M_model *= T(. . .)*R(. . .);
    left_hand();
    M_save2 = M_model;
    M_model *= T(. . .)*R(. . .);
    left_thumb();
    M_model = M_save2;
    M_model *= T(. . .)*R(. . .);
    left_forefinger();
    M_model = M_save2;
    . . .
}
```

Is there a better way to keep track of piles of matrices that need to be saved, modified, and restored?

Human figure implementation, better

```
figure()
{
    torso();
    push(M_model);
        M_model *= T(. . .)*R(. . .);
        head();
    M_model = pop(M_model);
    push(M_model);
        M_model *= T(. . .)*R(. . .);
        left_upper_arm();
        M_model *= T(. . .)*R(. . .);
        left_lower_arm();
        M_model *= T(. . .)*R(. . .);
        left_hand();
        push(M_model);
          M_model *= T(. . .)*R(. . .);
          left_thumb();
        M_model = pop(M_model);
        push(M_model);
           M_model *= T(. . .)*R(. . .);
           left_forefinger();
        M_model = pop(M_model);
        push(M_model);
    . . .
}
```

Human figure implementation, OpenGL

```
figure()
{
    torso();
    glPushMatrix();
        glTranslate( ... );
        glRotate( ... );
        head();
    glPopMatrix();
    glPushMatrix();
        glTranslate( ... );
        glRotate( ... );
        left_upper_arm();
        glTranslate( ... );
        glRotate( ... );
        left_lower_arm();
        glTranslate( ... );
        glRotate( ... );
        left_hand();
        glPushMatrix();
            glTranslate( ... );
            glRotate( ... );
            left_thumb();
        glPopMatrix();
        glPushMatrix();
            glTranslate( ... );
            glRotate( ... );
            left_forefinger();
        glPopMatrix();
    . . .
}
```

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.

Kinematics and dynamics

Definitions:

- **Kinematics:** how the positions of the parts vary as a function of the joint angles.
- **Dynamics:** how the positions of the parts vary as a function of applied forces.

Questions:

Q: What do the terms **inverse kinematics** and **inverse dynamics** mean?

Q: Why are these problems more difficult?

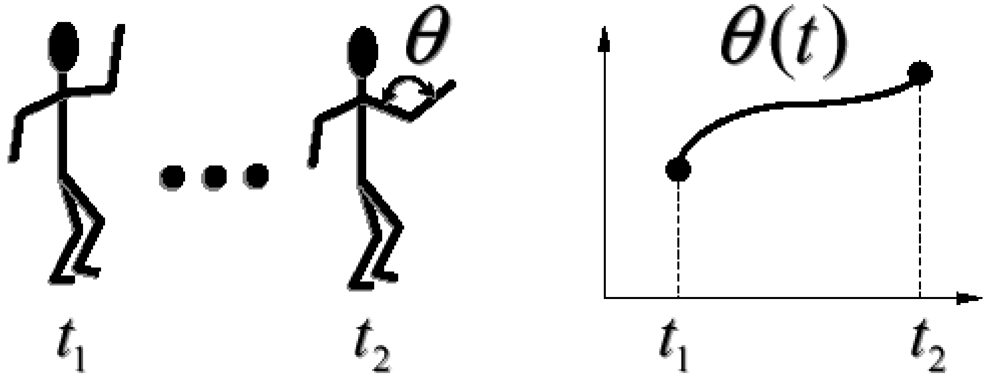
Key-frame animation

The most common method for character animation in production is **key-frame animation**.

- Each joint specified at various **key frames** (not necessarily the same as other joints)
- System does interpolation or **in-betweening**

Doing this well requires:

- A way of smoothly interpolating key frames: **splines**
- A good interactive system
- A lot of skill on the part of the animator

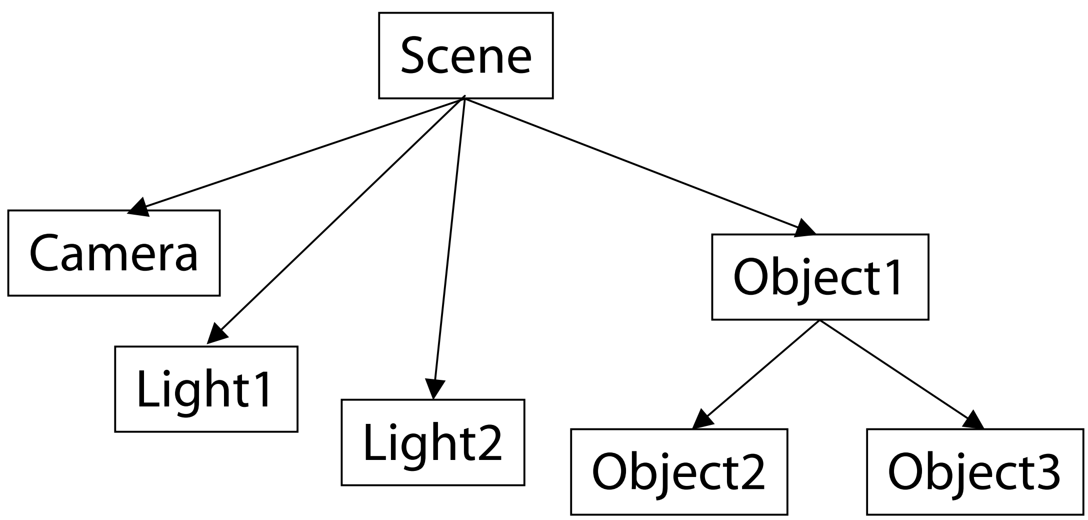


Scene graphs

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**.



The peculiarity of OpenGL ordering

Let's revisit the very first simple example in this lecture.

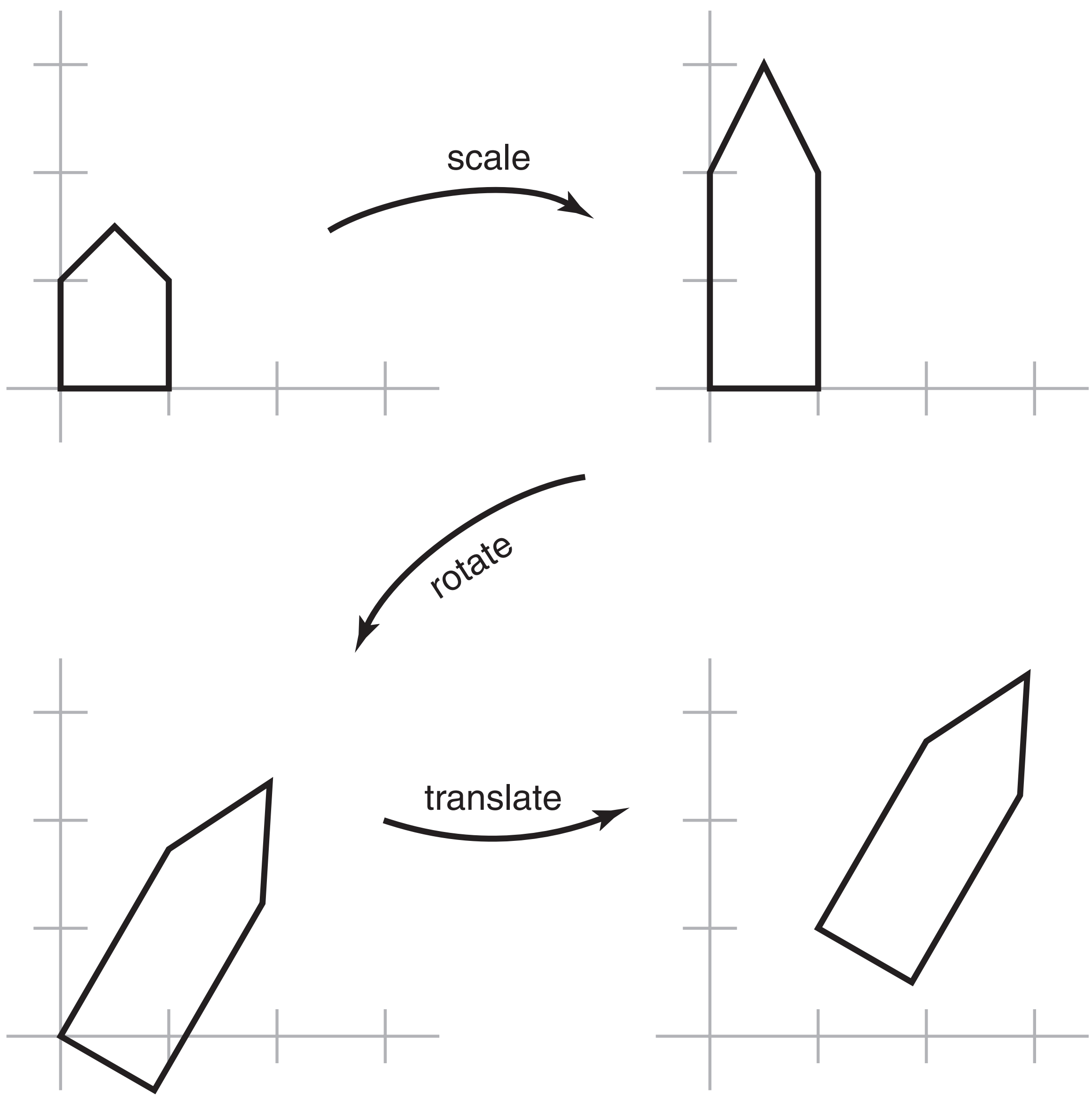
To draw the transformed house, we would write OpenGL code like:

```
glMatrixMode( GL_MODELVIEW );
glLoadIdentity();
glTranslatef( ... );
glRotatef( ... );
glScalef( ... );
house();
```

Is there something a little funny about the order of operations?

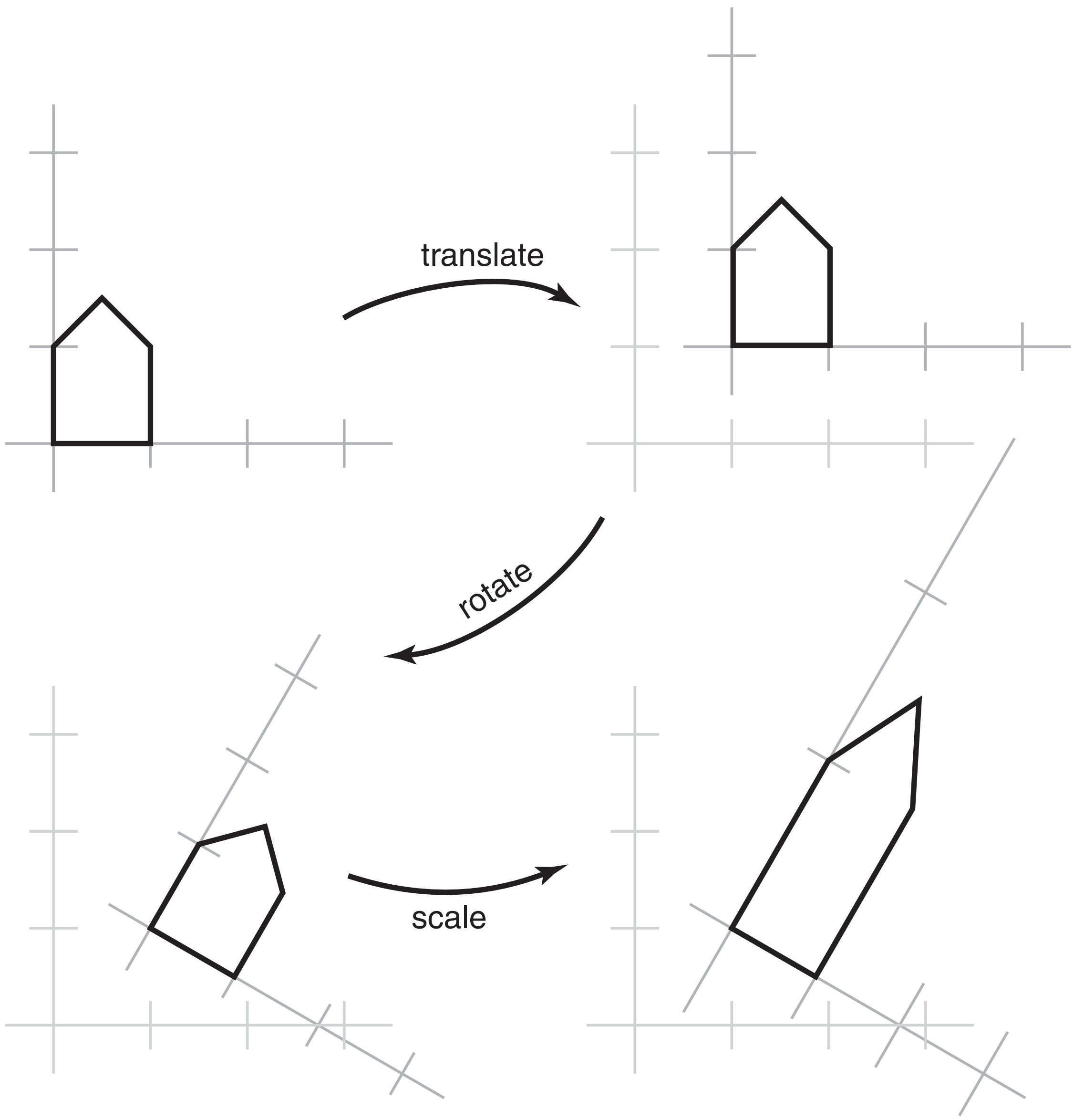
Global, fixed coordinate system

OpenGL's transforms, logical as they may be, still *seem backwards*. They are, if you think of them as transforming the object in a **fixed** coordinate system.



Local, changing coordinate system

Another way to view transformations is as affecting a *local coordinate system* that the primitive is drawn in. Now the transforms appear in the “right” order.



Summary

Here's what you should take home from this lecture:

- All the **boldfaced terms**.
- How primitives can be instanced and composed to create hierarchical models using geometric transforms.
- How the notion of a model tree or DAG can be extended to entire scenes.
- How keyframe animation works.
- How transforms can be thought of as affecting either the geometry, or the coordinate system which it is drawn in.