Lecture 07:

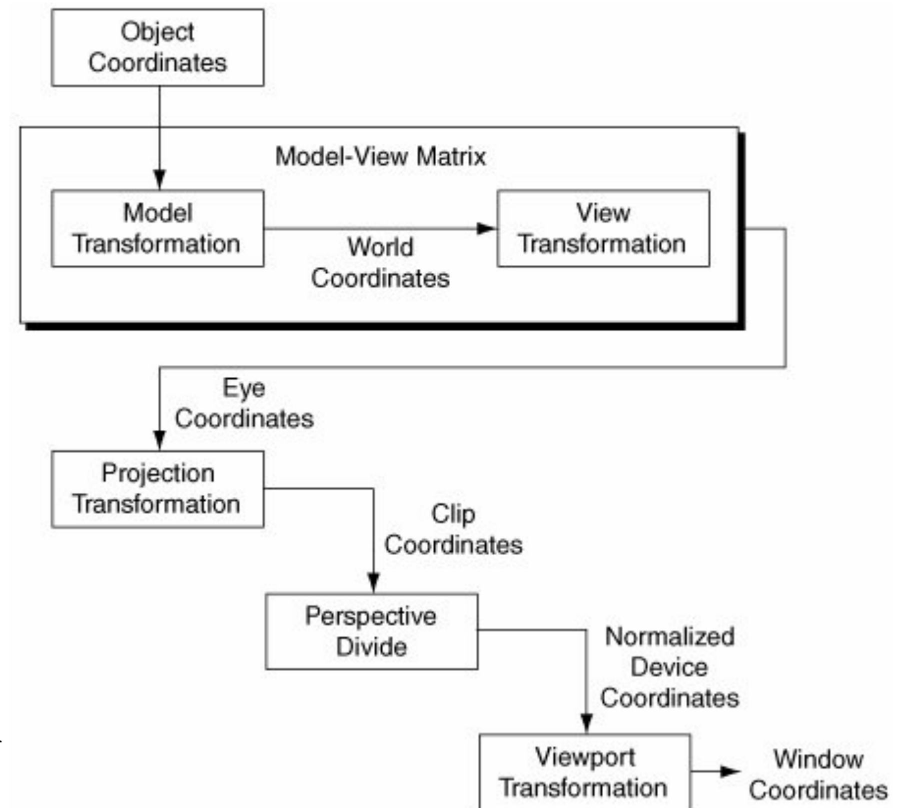
Scene Graph

COMP 175: Computer Graphics

March 10, 2015

Refresher: OpenGL Matrix Transformation Pipeline

- Input: list of 3D coordinates (x, y, z)
- GL_MODELVIEW
  - Model transform
  - View transform
- GL_PROJECTION
  - Projection transform
  - Clipping
  - Perspective division
- Viewport
  - Viewport transform
- Output: 2D coordinates for each input 3D coordinates



Normalization Transformation - Composite

- To recap, to normalize a parallel view volume into canonical form, we perform 3 steps:
  - Translate the view volume to the origin
  - Rotation from (u, v, w) to (x, y, z)
  - Scale into (-1, 1) range in x, y and (0, -1) in z
- Together, we can write this as:

$$M_{orthogonal} = \begin{bmatrix} \frac{2}{width} & 0 & 0 & 0 \\ 0 & \frac{2}{height} & 0 & 0 \\ 0 & 0 & \frac{1}{far} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ w_x & w_y & w_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -P_{n_x} \\ 0 & 1 & 0 & -P_{n_y} \\ 0 & 0 & 1 & -P_{n_z} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

In Summary!

- The full projection matrix: $M_{pp}S_{xyz}R_{uvz2xyz}T_{uvw}$:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{-1}{c+1} & \frac{c}{c+1} \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} \frac{1}{\tan\left(\frac{\theta_w}{2}\right) far} & 0 & 0 & 0 \\ 0 & \frac{1}{\tan\left(\frac{\theta_h}{2}\right) far} & 0 & 0 \\ 0 & 0 & \frac{1}{far} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ w_x & w_y & w_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -e_x \\ 0 & 1 & 0 & -e_y \\ 0 & 0 & 1 & -e_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- Where:
  - Theta_w is the camera angle in the x direction
  - Theta_h is the camera angle in the y direction
    - Only one of the two needs to be specified. The other can be inferred from the screen's aspect ratio. In my implementation, camera angle = Theta_h
  - c = -near/far

Putting Everything All Together

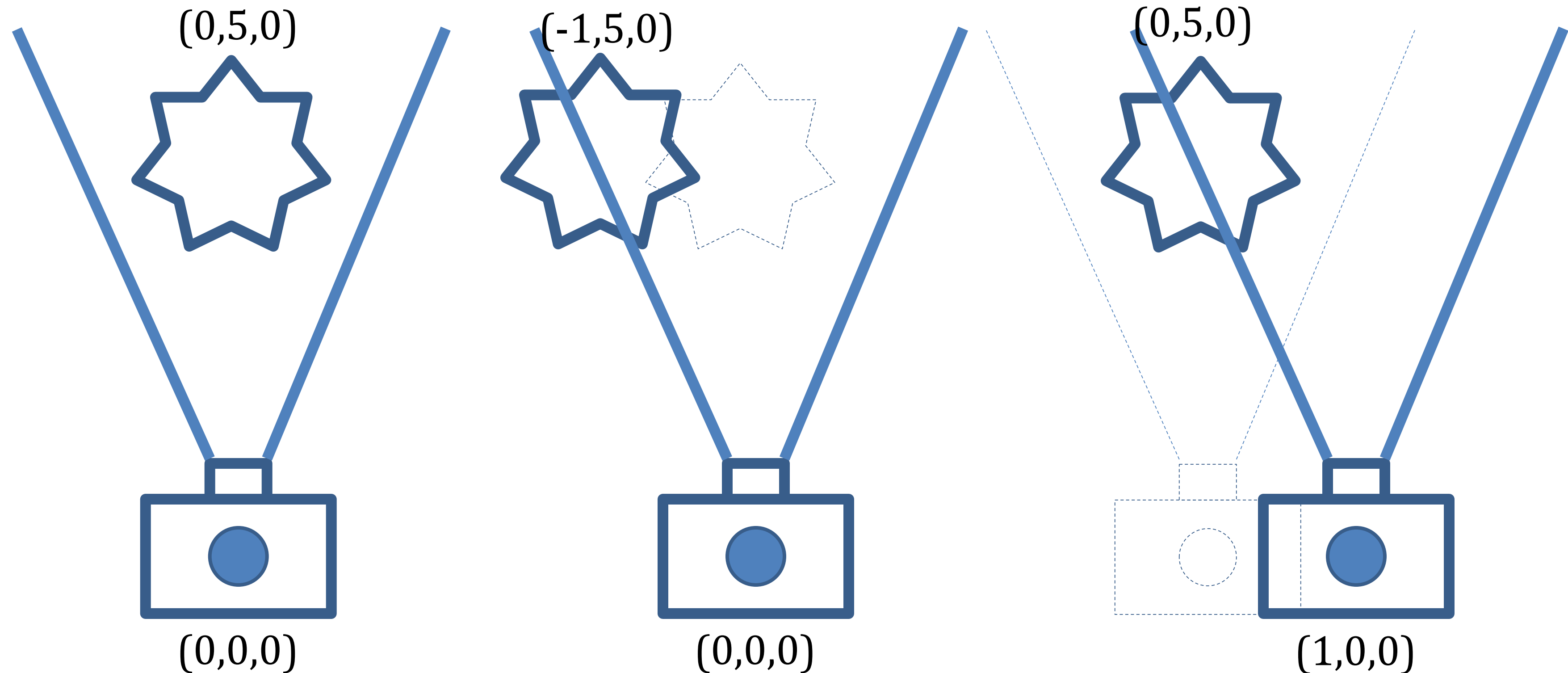
- We know about camera and object modeling transformations now, let's put them together:
- 1) $N_{perspective} = M_{pp} M_{perspective}$
- 2) $CMTM = SRT$
  - The CMTM (Composite Modeling Transformation Matrix) is a composite matrix of all of our object modeling transformations (Scaling, Rotating, Translations, etc)
- 3) $CTM = N_{perspective} * CMTM$
  - The CTM (Composite Transformation Matrix) is the combination of all our camera and modeling transformations
    - In OpenGL it is referred to as the ModelViewProjection Matrix
    - Model: Modeling Transformations
    - View: Camera translate/rotate
    - Projection: Frustum scaling/unhinging

What is the Camera's ModelView Matrix?

```
glMatrixMode(GL_PROJECTION);
glLoadMatrixd(myCamera->getProjectionMatrix());

glMatrixMode(GL_MODELVIEW);
glLoadMatrixd(myCamera->getModelViewMatrix());

Matrix M = T-1 * (R * S) * T;
glMultMatrixd(M.getValues());

DrawObject();
```

- Hrm.. The camera's modelview transform is really indistinguishable from the object's transform matrix as far as OpenGL is concerned.
- Specifically, they are the same glTranslate, glRotate calls!

Camera as a model



- For rendering, the middle and right models create the same image.
  - Meaning that we can either think of model transform and view transforms as **Inverses** of each other

Coordinate Spaces

- This means that we have to be careful with our coordinate spaces.

- For this class, there are 3 coordinate spaces that we care about:
  - World Coordinate Space
  - Camera Coordinate Space
  - Object Coordinate Space

For Example:

```
Matrix M = T-1 * (R * S) * T;
glMultMatrixd(M.getValues());

DrawObject();
```

- When DrawObject is called, it assumes an Object Coordinate Space
  - That is, when you draw your sphere, cube, cone, cylinder, these objects are drawn at the origin and with specific size and orientation
- Matrix M transforms a point from the Object Coordinate Space into the World Coordinate Space

In the Case of Camera…

- The ModelView matrix:

$$\begin{bmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ w_x & w_y & w_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -P_{n_x} \\ 0 & 1 & 0 & -P_{n_y} \\ 0 & 0 & 1 & -P_{n_z} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- These matrices bring a data point from the World Coordinate Space into the Camera Coordinate Space

Canonical Camera Coordinate

- To further bring a point from World Coordinate Space into a Canonical Camera Coordinate Space
  - That is, to "scale" the camera space into a pre-defined size (of -1 to 1)

$$\begin{bmatrix} \frac{2}{width} & 0 & 0 & 0 \\ 0 & \frac{2}{height} & 0 & 0 \\ 0 & 0 & \frac{1}{far} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Which Space Does a Matrix transform To and From?

- This is often the most confusing part of 3D graphics.
- A good way to think about it is to think about the following operation (M is a matrix):

Point p = M * origin

- For example, consider the (Ortho) ModelView Matrix:

$$M = \begin{bmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ w_x & w_y & w_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -P_{n_x} \\ 0 & 1 & 0 & -P_{n_y} \\ 0 & 0 & 1 & -P_{n_z} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

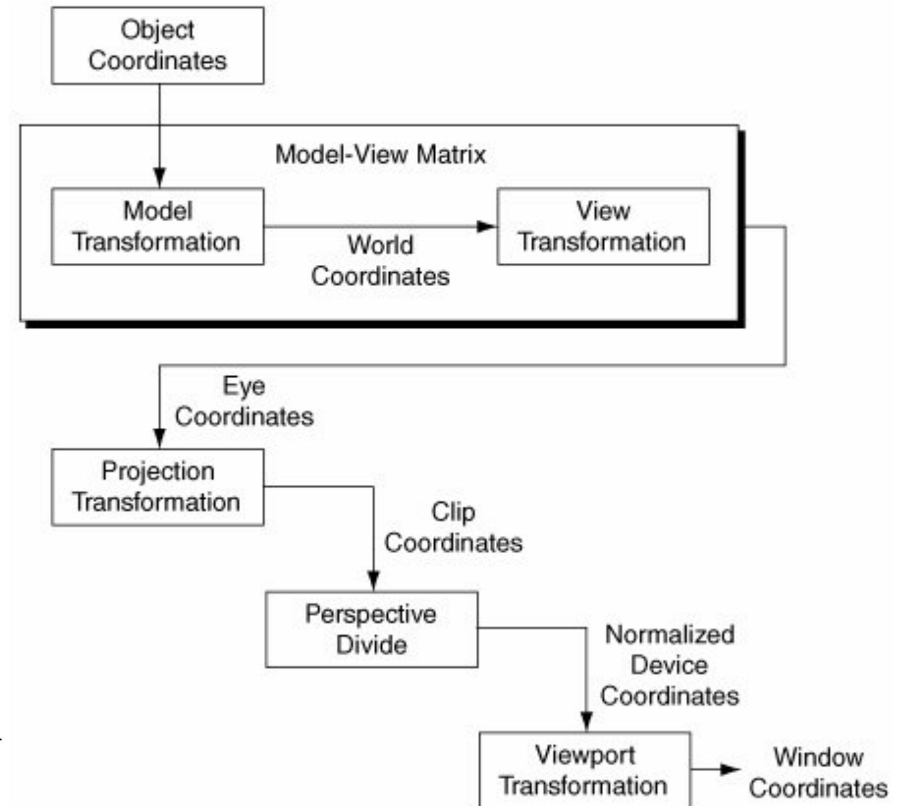
- In this case, when applied to the origin, you get back negative (inverse) of the camera's eye point.
  - Think about it for a second… So let's say that your camera is at (2, 0, 0)
  - After applying this matrix to the origin, you get back (-2, 0, 0)
  - So it's as if the coordinate system shifted such that the camera goes to (0, 0, 0)
  - As such, the matrix must be a transform from the world space (xyz) to the camera (uvw) space

Best Thing about Matrices

- If M transforms from A to B coordinate systems
- Then M^{-1} is the inverse that transforms from B to A coordinate systems
- This is important!! Keep this in mind because for the rest of the class, we will be navigating through coordinate systems!
- Why?
  - Because certain mathematical operations are MUCH easier in certain spaces.
  - For example, Ray-Object intersection

Refresher: OpenGL Matrix Transformation Pipeline

- Input: list of 3D coordinates (x, y, z)
- GL_MODELVIEW
  - Model transform
  - View transform
- GL_PROJECTION
  - Projection transform
  - Clipping
  - Perspective division
- Viewport
  - Viewport transform
- Output: 2D coordinates for each input 3D coordinates



Questions?

Applying Composite Matrices

- Here's the thought: Assume that you are given a Transformation Matrix in the form of $T_{xyz}R_{\theta_x\theta_y\theta_z}S_{xyz}$, and you want to apply it to one of the objects that you created in shapes.

- How would you do this without the use of OpenGL?

Applying Composite Matrices

```
Matrix M = T * (R * S);
for (i=0; i<numTriangles; i++) {
        Triangle t = MyObject->triangles[i];
        glBegin(GL_TRIANGLES);
        for (j=0; j<3; j++) {
                Position p = t->vertex[j]->GetPosition();
                Position newP = M * p;
                glVertex3d (newP.x, newP.y, newP.z);
        }
        glEnd();
}
```

- Yikes! Matrix multiplication in software. SLOW!

Applying Composite Matrices with OpenGL

```
Matrix M = T * (R * S);
glLoadMatrixd(M.getValues());
for (i=0; i<numTriangles; i++) {
        Triangle t = MyObject->triangles[i];
        glBegin(GL_TRIANGLES);
        for (j=0; j<3; j++) {
                Position p = t->vertex[j]->GetPosition();
                glVertex3d (newP.x, newP.y, newP.z);
        }
        glEnd();
}
```

Be careful! Read the OpenGL spec to see how you are arranging your matrix orientation and make sure that it's consistent with OpenGL.

You might need to do a transpose. See Algebra.H for how this is done

- Much better... Now we're using the graphics card to do the computation for us.

Applying Composite Matrices with Camera

```
glMatrixMode(GL_PROJECTION);
glLoadMatrixd(myCamera->getProjectionMatrix().getValues());
glMatrixMode(GL_MODELVIEW);
glLoadMatrixd(myCamera->getModelViewMatrix().getValues());
Matrix M = T * (R * S);
glMultMatrixd(M.getValues());
for (i=0; i<numTriangles; i++) {
        Triangle t = MyObject->triangles[i];
        glBegin(GL_TRIANGLES);
        for (j=0; j<3; j++) {
                Position p = t->vertex[j]->GetPosition();
                glVertex3d (newP.x, newP.y, newP.z);
        }
        glEnd();
}
```

Drawing Multiple Objects

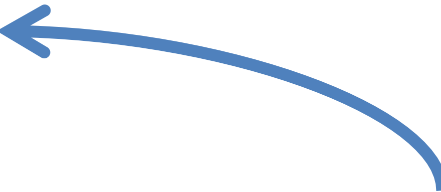
- Let's say that you now have 2 objects, myCube and mySphere.
- Each object needs to be transformed differently:
  - For myCube, the transforms are $M_c = T_c R_c S_c$
  - For mySphere, the transforms are: $M_s = T_s R_s S_s$
- Let's see how we will write this code:

Drawing Multiple Objects

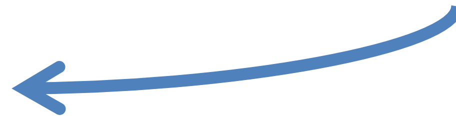
```
glMatrixMode(GL_PROJECTION);
glLoadMatrixd(myCamera->getProjectionMatrix().getValues());
glMatrixMode(GL_MODELVIEW);
glLoadMatrixd(myCamera->getModelViewMatrix().getValues());

Matrix Mc = Tc * (Rc * Sc);
glMultMatrixd(Mc.getValues());
myCube.draw();

Matrix Ms = Ts * (Rs * Ss);
glMultMatrixd(Ms.getValues());
mySpher.draw();
```

Drawing Multiple Objects

```
glMatrixMode(GL_PROJECTION);
glLoadMatrixd(myCamera->getProjectionMatrix().getValues());
glMatrixMode(GL_MODELVIEW);
glLoadMatrixd(myCamera->getModelViewMatrix().getValues());

Matrix Mc = Tc * (Rc * Sc);
glMultMatrixd(Mc.getValues());
myCube.draw();

Matrix Ms = Ts * (Rs * Ss);
glMultMatrixd(Ms.getValues());
mySpher.draw();
```



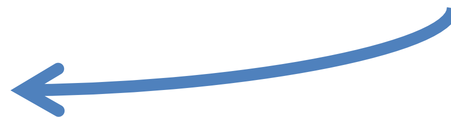
What is wrong with this?



Drawing Multiple Objects

```
glMatrixMode(GL_PROJECTION);
glLoadMatrixd(myCamera->getProjectionMatrix().getValues());
glMatrixMode(GL_MODELVIEW);
glLoadMatrixd(myCamera->getModelViewMatrix().getValues());

Matrix Mc = Tc * (Rc * Sc);
glMultMatrixd(Mc.getValues());
myCube.draw();

Matrix Mc_Inv = Mc.Inverse();
glMultMatrixd(Mc_Inv.getValues());

Matrix Ms = Ts * (Rs * Ss);
glMultMatrixd(Ms.getValues());
mySpher.draw();
```

What about…



Drawing Multiple Objects

```
glMatrixMode(GL_PROJECTION);
glLoadMatrixd(myCamera->getProjectionMatrix().getValues());
glMatrixMode(GL_MODELVIEW);
glLoadMatrixd(myCamera->getModelViewMatrix().getValues());

glPushMatrix();
        Matrix Mc = Tc * (Rc * Sc);
        glMultMatrixd(Mc.getValues());
        myCube.draw();
glPopMatrix();

glPushMatrix();
        Matrix Ms = Ts * (Rs * Ss);
        glMultMatrixd(Ms.getValues());
        mySpher.draw();
glPopMatrix();
```

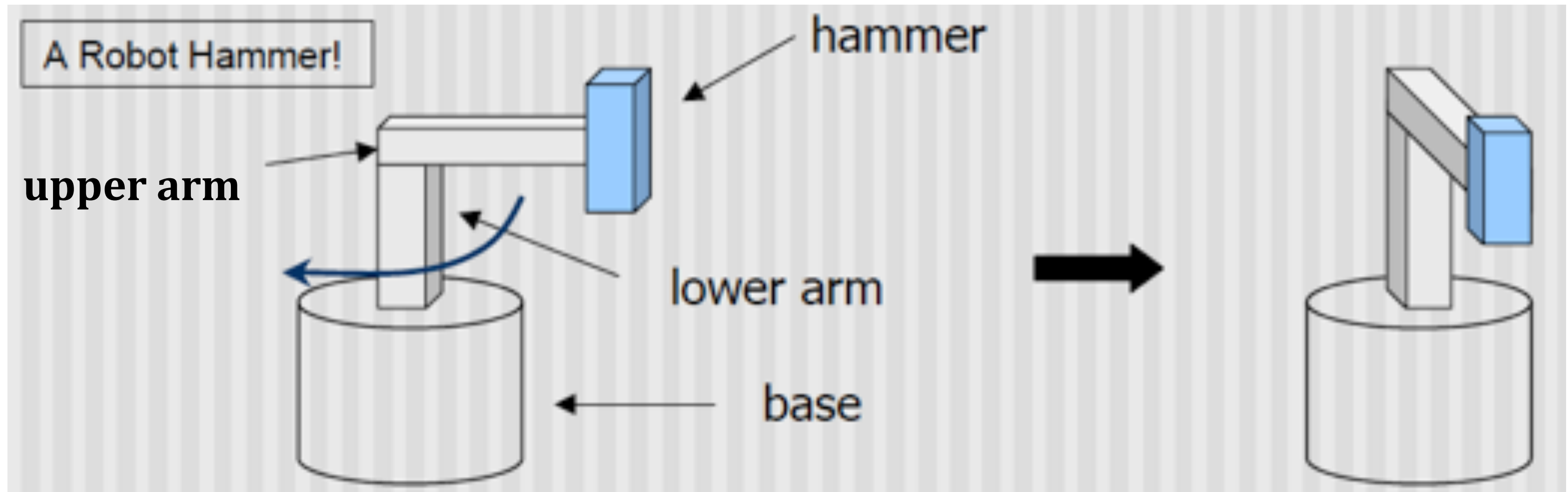
Drawing Multiple Objects – Being Paranoid…

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glLoadMatrixd(myCamera->getProjectionMatrix().getValues());
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glLoadMatrixd(myCamera->getModelViewMatrix().getValues());

glPushMatrix();
        Matrix Mc = Tc * (Rc * Sc);
        glMultMatrixd(Mc.getValues());
        myCube.draw();
glPopMatrix();

glPushMatrix();
        Matrix Ms = Ts * (Rs * Ss);
        glMultMatrixd(Ms.getValues());
        mySpher.draw();
glPopMatrix();
```

Questions?

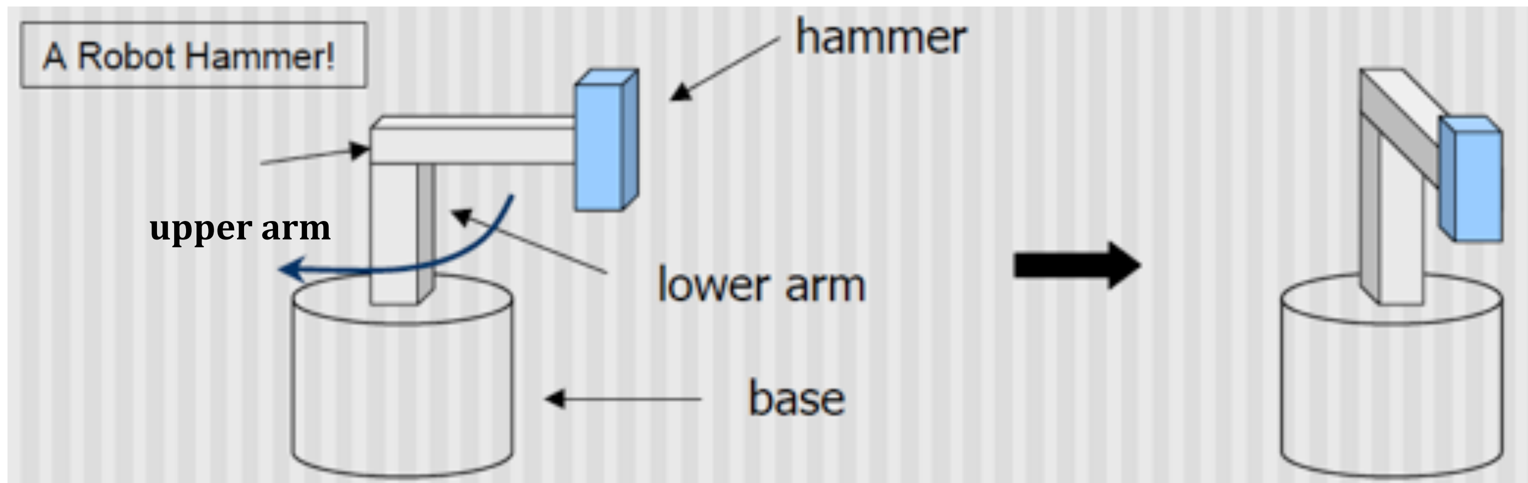
Hierarchical Transform

- 3D objects are often constructed in a way that have dependency
  - This means that an object's position, orientation, etc. will depend on its parents position, orientation, etc.



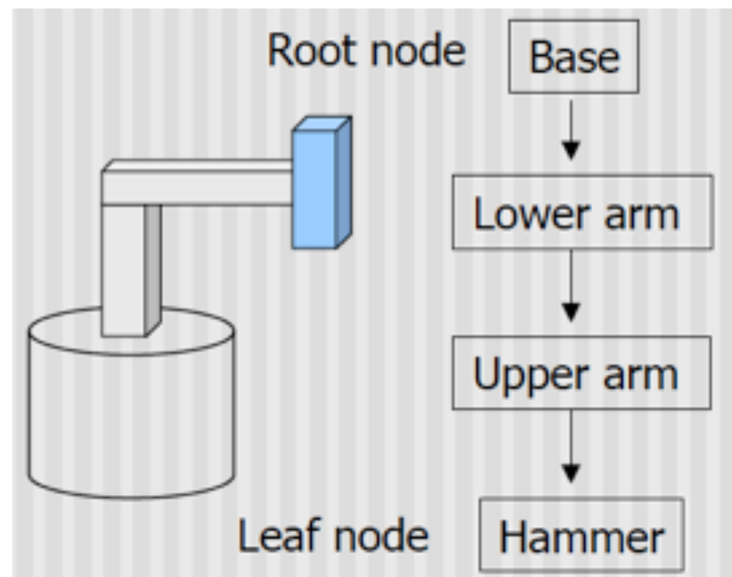
Hierarchical Transform

- While you can compute how the hammer will move based on lower arm's rotation, this is really a pain.
  - You'll need to calculate the rotation of the lower arm.
  - Followed by the upper arm.
  - And finally the hammer.



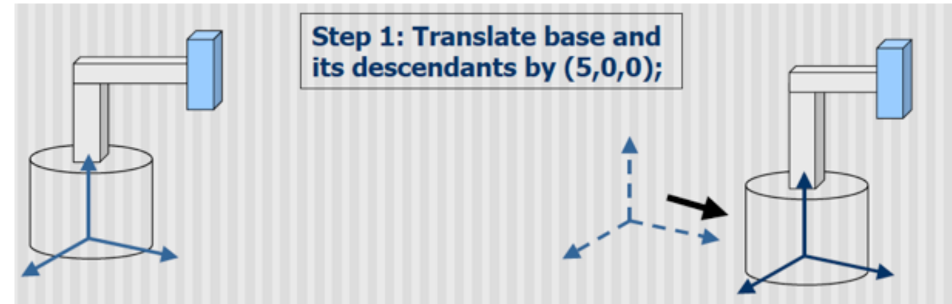
Hierarchical Transform

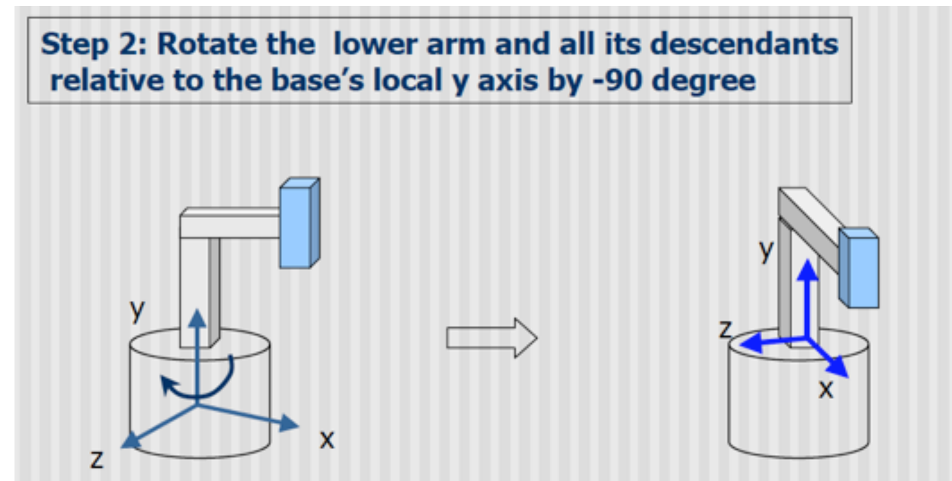
- If we think of the Robot Arm as a hierarchy of objects…
  - Then an object's position and orientation will be based on its parents, grand parents, grand grand parents, etc.
- This hierarchical representation is called a "**Scene Graph**"



Basic Approach

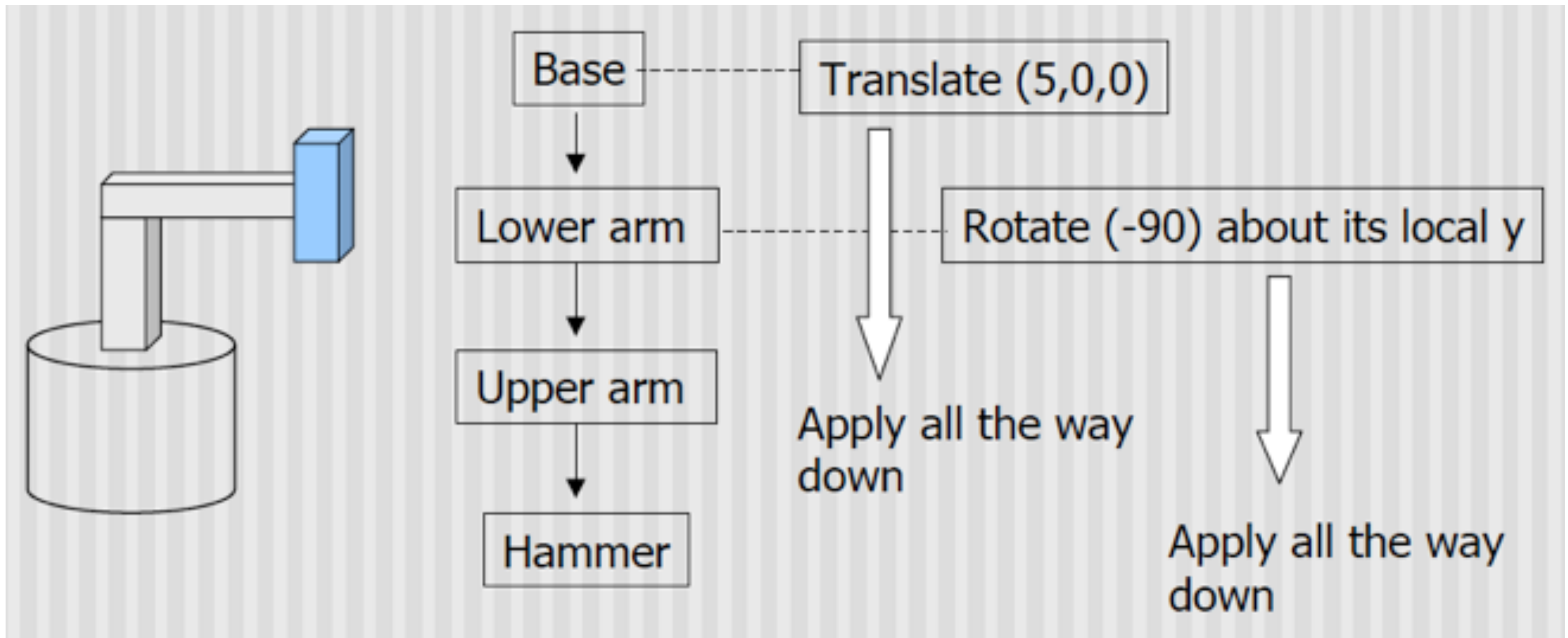
- Let’s say that you want to move the Robotic Arm to the right (in +x axis) by 5 units, and the upper arm rotated by some amount.
  - Using the Scene Graph, this could mean that we iteratively apply:
    - The same Translation to all the children of the root node (base)
    - The same Rotation to all the children of the upper arm





Relative Transformation

- In this sense, all the transformations are relative to an object's parents / ancestors.



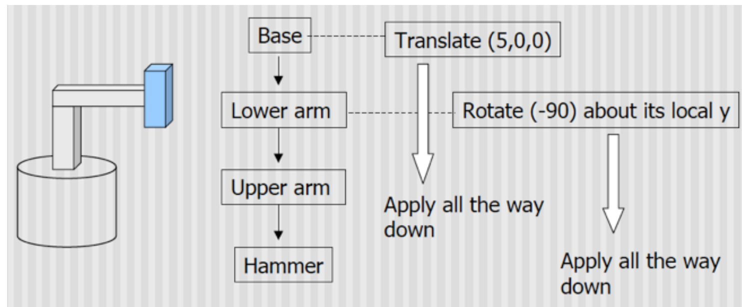
OpenGL Matrix Stack

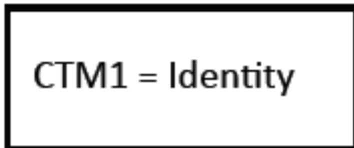
- We had seen earlier how the OpenGL Matrix Stack could work.
- Recall earlier, when we tried to draw 2 objects:

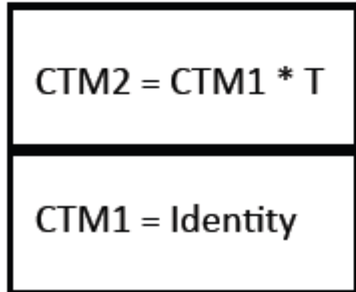
```
glMatrixMode(GL_PROJECTION);
glLoadMatrixd(myCamera->getProjectionMatrix().getValues());
glMatrixMode(GL_MODELVIEW);
glLoadMatrixd(myCamera->getModelViewMatrix().getValues());

Matrix Mc = Tc * (Rc * Sc);
glMultMatrixd(Mc.getValues());
myCube.draw();

Matrix Ms = Ts * (Rs * Ss);
glMultMatrixd(Ms.getValues());
mySpher.draw();
```

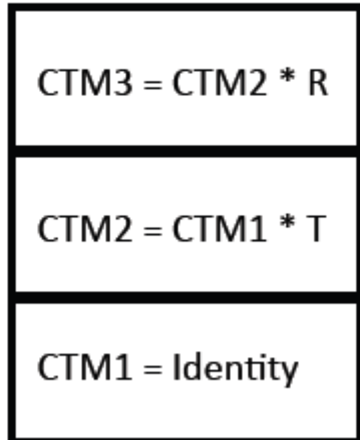
OpenGL Matrix Stack

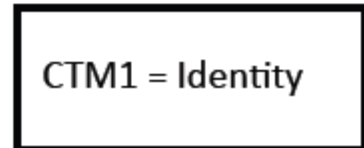
- Combine that with the use of **glPushMatrix** and **glPopMatrix**
  - note: CTM = current transformation matrix











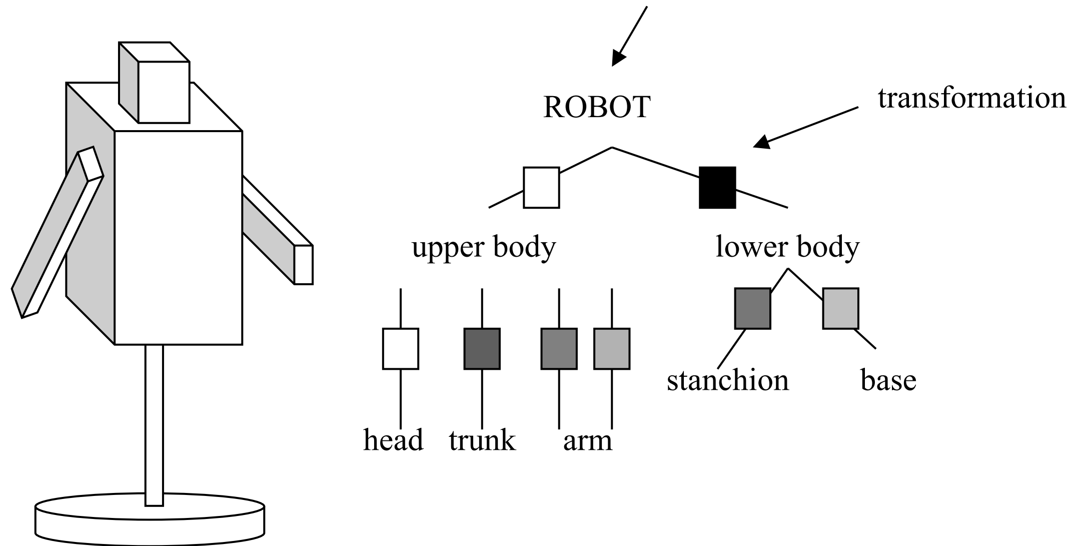
OpenGL Matrix Stack

```
drawRobotArm() {
	// push to copy the CTM so we can restore it at the end of the routine
	glPushMatrix(); // CTM2 = CTM1
		Matrix T = trans_mat(5,0,0);
		glMultMatrix(T); // CTM2 = CTM2 * T
		drawBase();

		Matrix R = rot_mat(-90, 0,1,0);
		glPushMatrix(); // CTM3 = CTM2
			glMultMatrix(R); // CTM3 = CTM3*R
			drawLowerArm();
			drawUpperArm();
			drawHammer();
		glPopMatrix(); // pop off CTM3
	glPopMatrix(); // pop off CTM2
	// now back in the base coordinate system of CTM1
}
```

Scene Graph

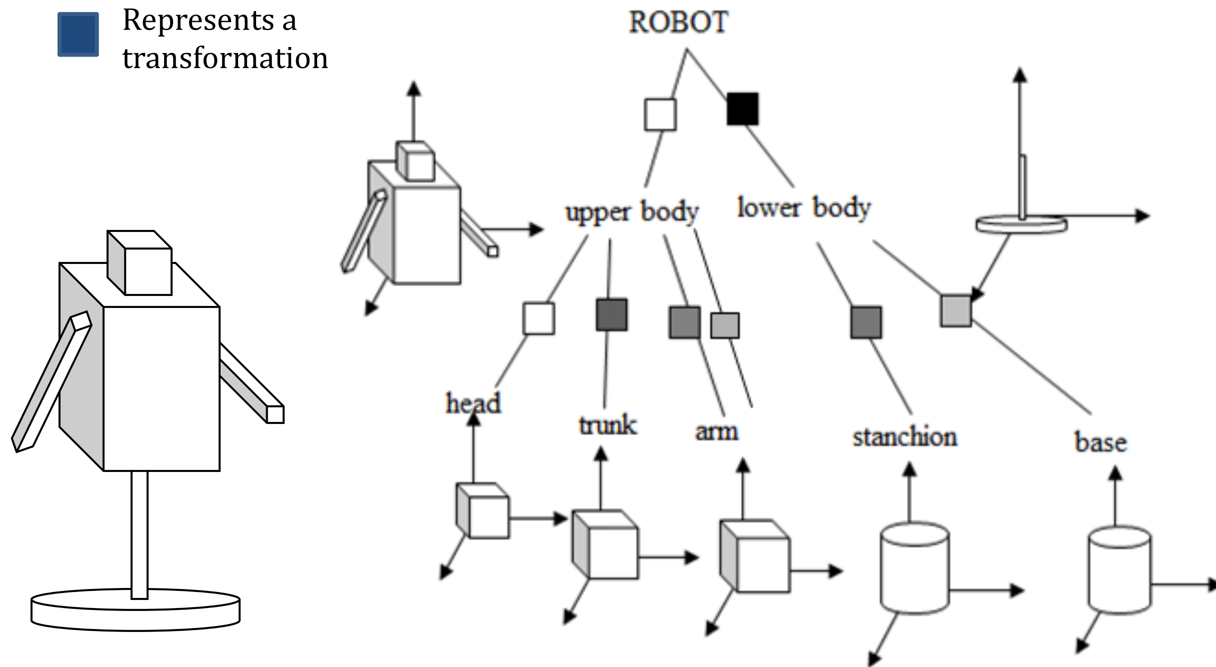
- As you might have noticed, each object in the Robot Arm can have:
  - An object description (cone, sphere, cube, etc.)
  - A transformation description (scale, rotate, translate)
  - Additional attribute information (such as appearance information: color, texture, etc.)
  - One parent (and ONLY one parent)
  - Some number of children (or none at all)

A Slightly More Complex Case



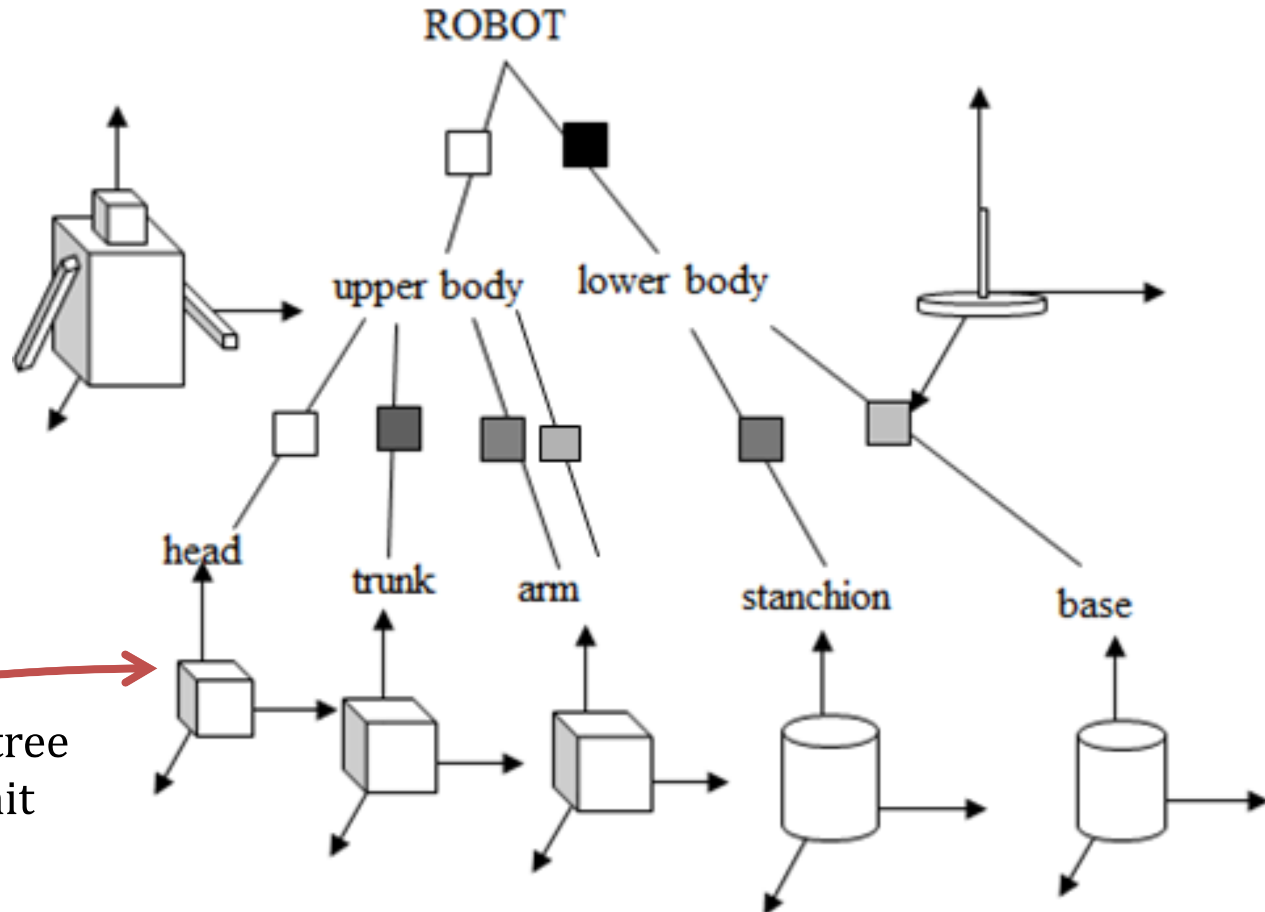
- In this case, notice that there are groupings of objects…
  - So we'll also need to consider an "abstract" node such that we can apply a transform to it.

Scene Graph



- We will iteratively apply the transforms “upwards”, starting at the bottom…

Applying Transforms

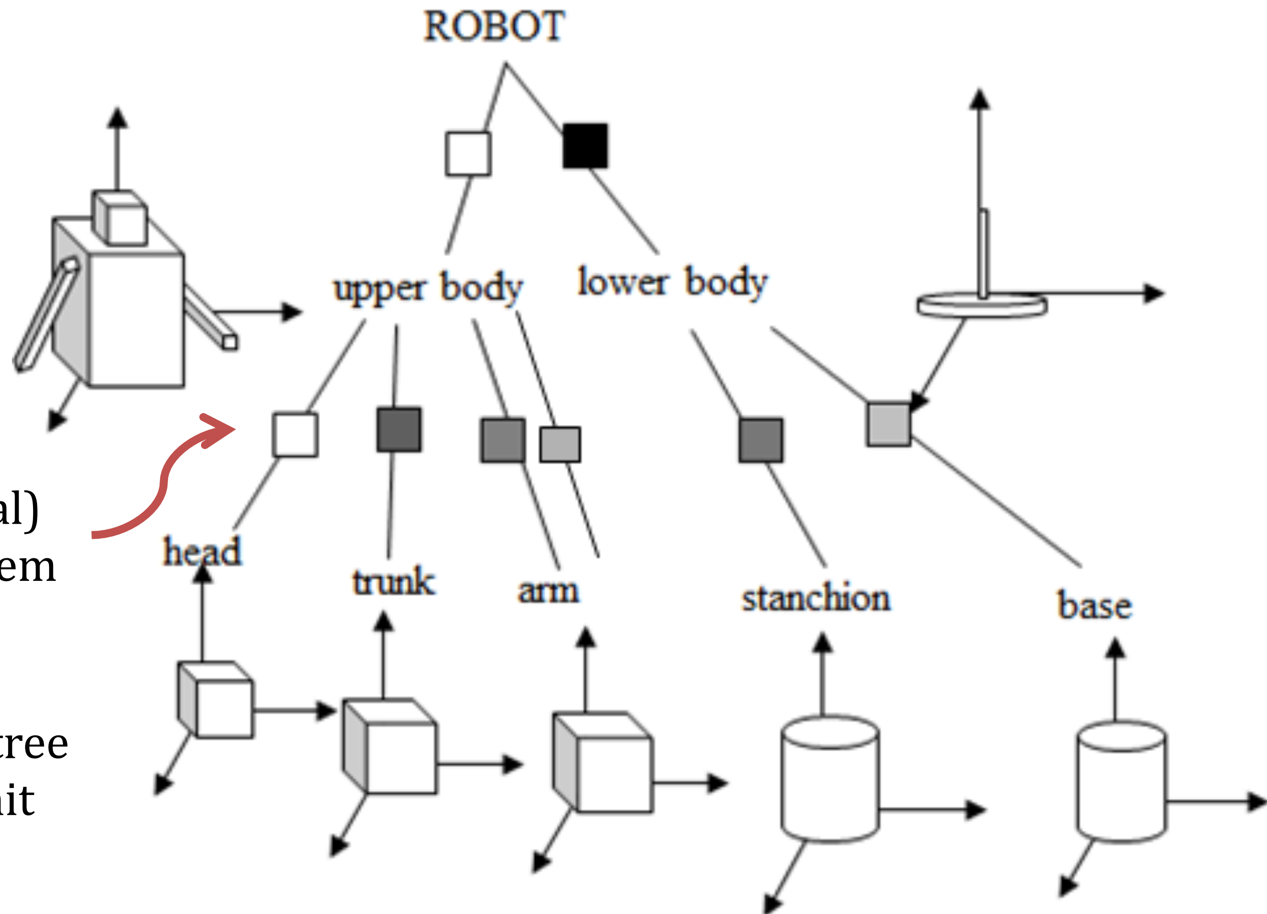


1. Leaves of the tree are standard (unit size) primitives

Applying Transforms

2. We apply (local) transforms to them

1. Leaves of the tree are standard (unit size) primitives

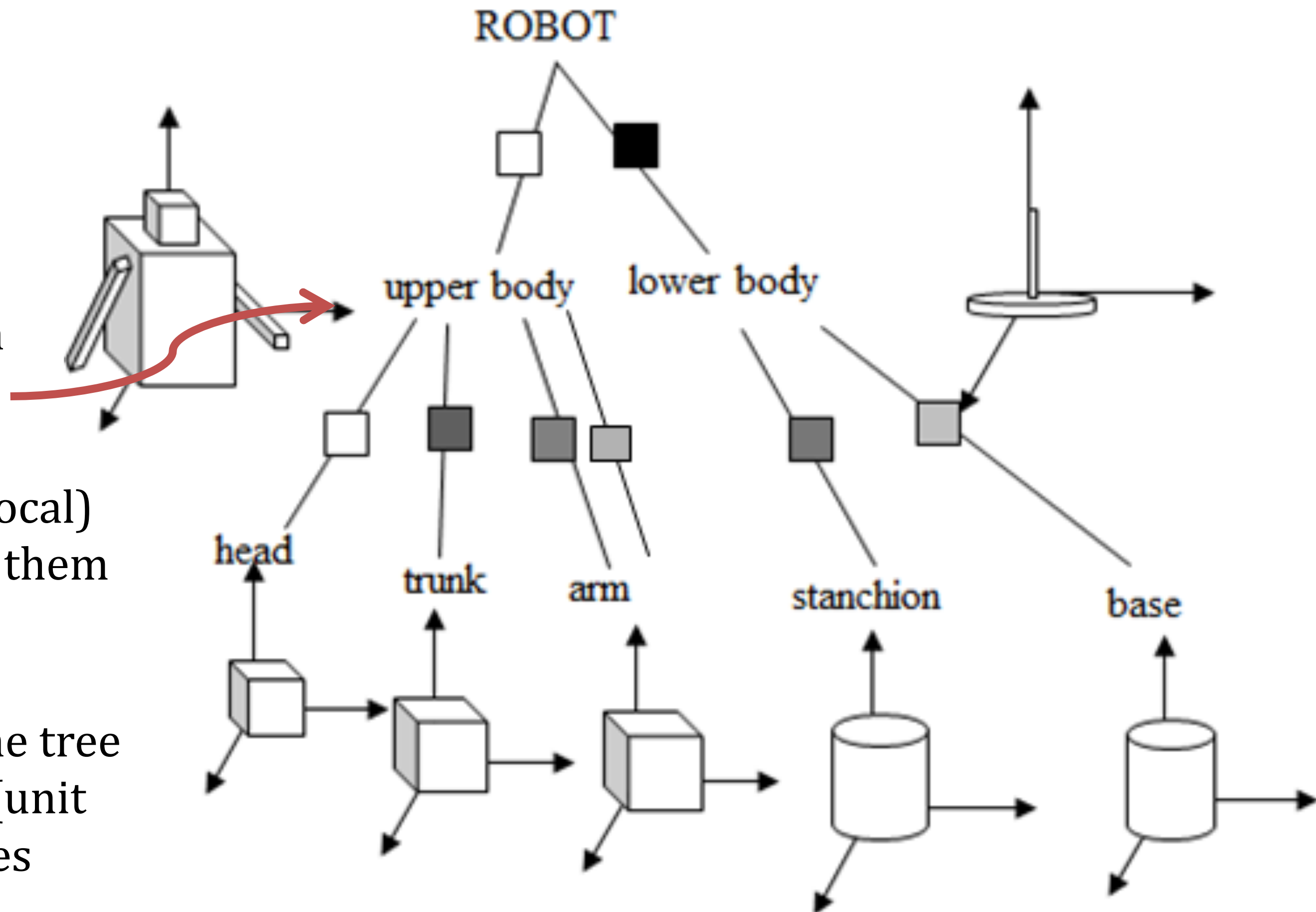


Applying Transforms

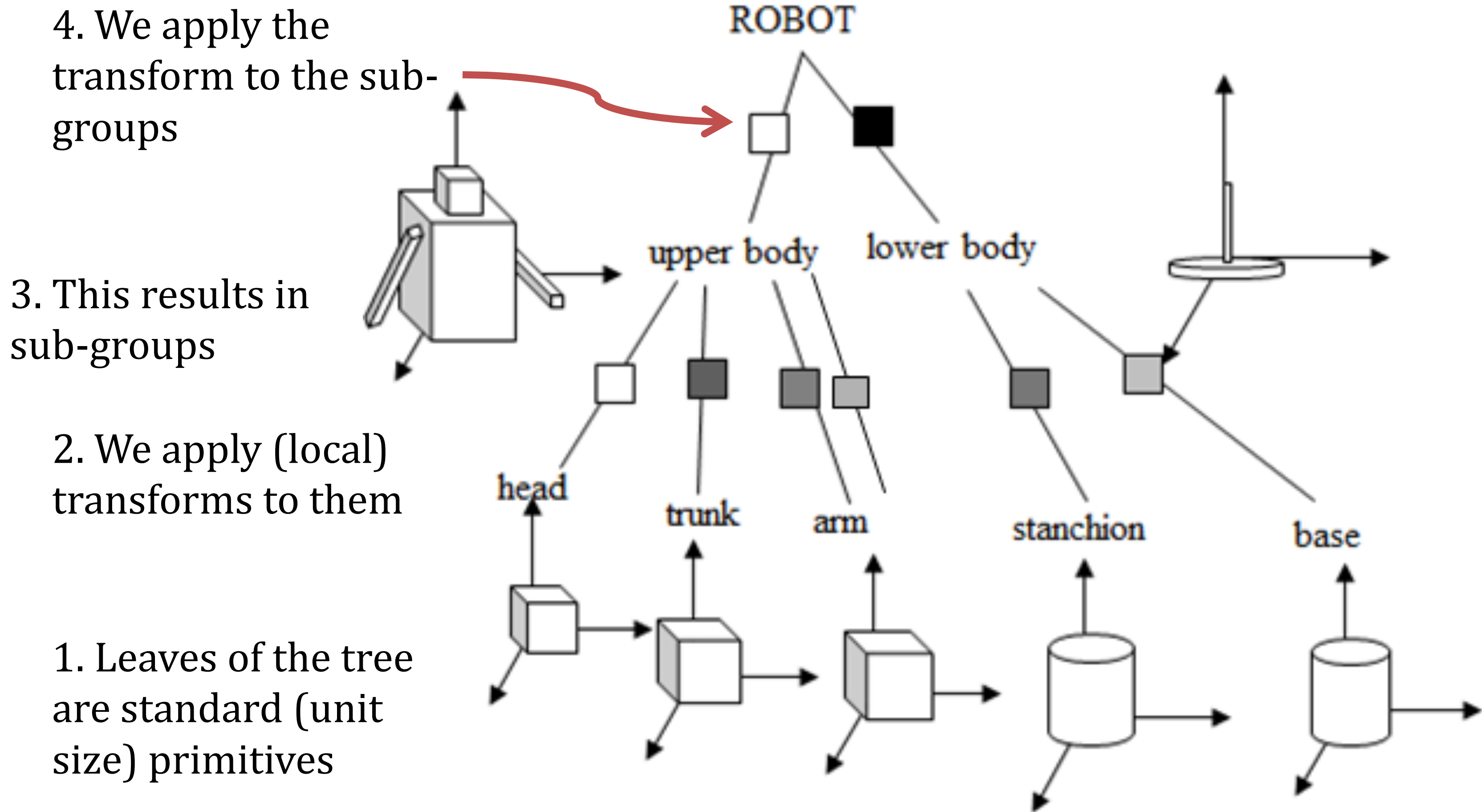
3. This results in sub-groups

2. We apply (local) transforms to them

1. Leaves of the tree are standard (unit size) primitives

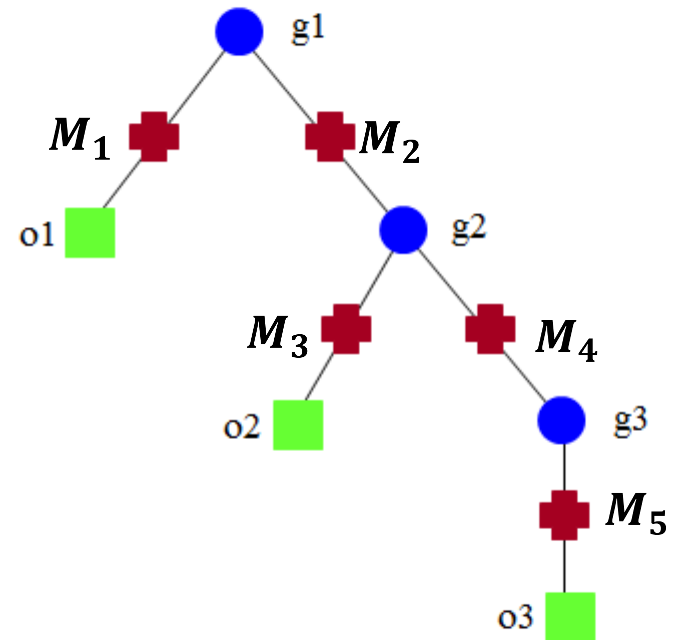


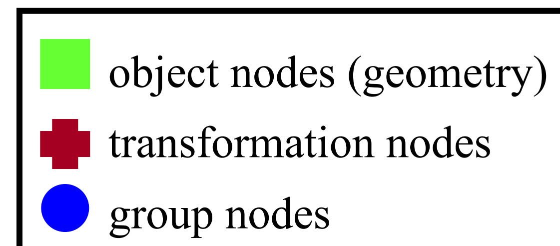
Applying Transforms



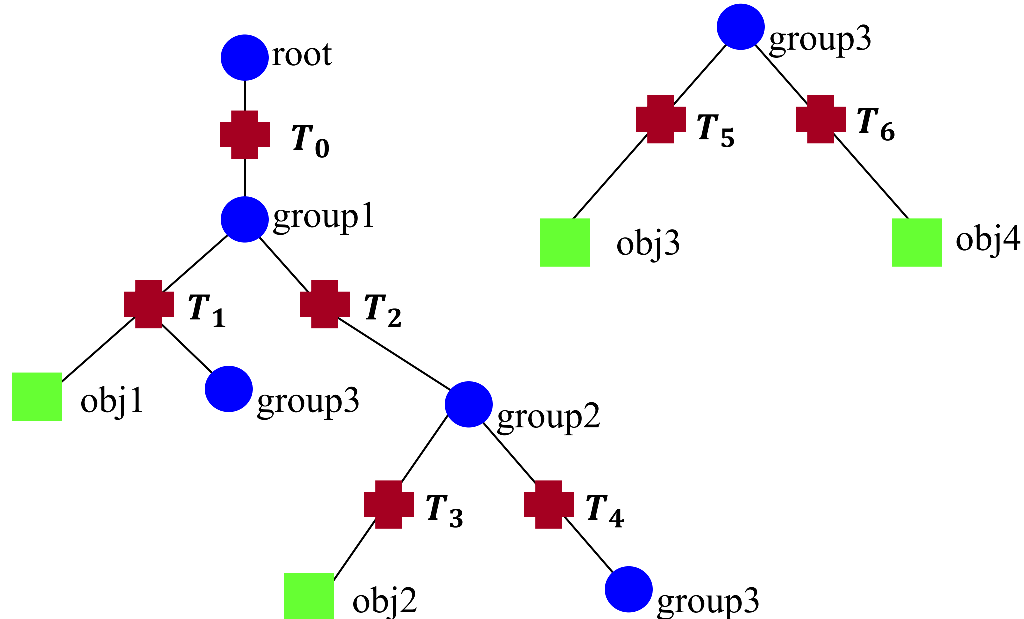
Transformation and Scene Graphs

- As discussed earlier, this results in a cumulative effect as one moves up the tree.
- In OpenGL:
  - For o1: CTM = M_1
  - For o2: CTM = $M_2 M_3$
  - For o3: CTM = $M_2 M_4 M_5$
  - For a vertex v in o3, v' in world coordinate would be:
    - $v' = M_2 M_4 M_5 v$





Reusing Objects / Trees



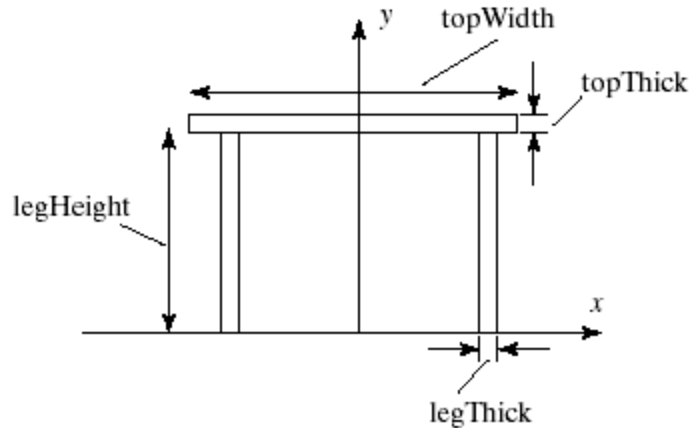
- You can create reusable sub-trees
  - group3 is used twice in the example above
    - The transforms within group3 don't change (T5 and T6)
    - but in each instance on the lft it is applied with different CTMs
      - $T_0 T_1$ vs. $T_0 T_2 T_4$

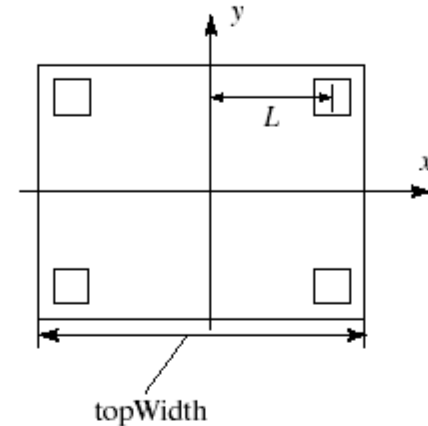
Representing the SceneGraph as a SceneFile

- There is a bit of intricacy in the SceneGraph, can we represent it as a text-based SceneFile?
- Note a few properties:
  - The Scene Graph is a DAG (directed acyclic graph)
  - The structure is inherently hierarchical
  - There are components that should be reusable
  - There are three main types:
    - Transform nodes
    - Object nodes
    - Group nodes

One Example: Scene Description Language (SDL)

```
def leg{
    push
        translate 0 .15 0
        scale .01 .15 .01
        cube
    pop
}

def table{
    push
        translate 0 .3 0
        scale .3 .01 .3
        cube
    pop
    push
        translate .275 0 .275
        use leg
        translate 0 0 -.55
        use leg
        translate -.55 0 .55
        use leg
        translate 0 0 -.55
        use leg
    Pop
}
```





- Calling this object:

```
push
    translate 0.4 0 0.4
    use table
pop
```

Using XML

- Assignment 4 uses an XML structure. Here's an example of what your scenefile will look like:

```
<transblock>
	<rotate x="0" y="1" z="0" angle="60"/>
	<scale x=".5" y=".5" z=".5"/>
	<object type="tree">
		<transblock>
			<translate x="0" y="2" z="0"/>
			<scale x="1" y=".5" z="1"/>
			<object type="primitive" name="sphere">
				<diffuse r="1" g="1" b="0"/>
			</object>
		</transblock>
	</object>
</transblock>
```

Questions?