

# Transformations in OpenGL

Lecturer:

**Carol O'Sullivan**

Professor of Visual Computing

Carol.OSullivan@cs.tcd.ie

Course www:

<http://isg.cs.tcd.ie/cosulliv/>

# Modeling Transformations

- The three OpenGL routines for modelling transformations are: **glTranslate\*()**, **glRotate\*()**, and **glScale\*()**
- These routines transform an object (or coordinate system) by moving, rotating, stretching, shrinking, or reflecting it.
- All three commands are equivalent to producing an appropriate translation, rotation, or scaling matrix, and then calling **glMultMatrix\*()** with that matrix as the argument.

# Modeling Transformations

- `glLoadIdentity()`
  - creates an *identity matrix* (used for clearing all transformations)
- `glLoadMatrixf(matrixptr)`
  - loads a *user specified transformation matrix* where `matrixptr` is defined as `GLfloat matrixptr[16];`

# Translate

- `glTranslate(dx, dy, dz)`
  - translates by *displacement vector* ( $dx$ ,  $dy$ ,  $dz$ )
- Multiplies the current matrix by a matrix that moves (translates) an object by the given x-, y-, and z-values (or moves the local coordinate system by the same amounts).
- Using (0.0, 0.0, 0.0) is the identity operation which means it has no effect on an object or its local coordinate system.

# Rotate

- **glRotatef(angle, vx, vy, vz)**
  - rotates about axis (vx, vy, vz) by angle (specified in degrees)
- Multiplies the current matrix by a matrix that rotates an object (or the local coordinate system) in a counterclockwise direction about the ray from the origin through the point (x,y,z).
- The **angle** parameter specifies the angle of rotation in degrees.
- The effect of **glRotatef(45.0, 0.0, 0.0, 1.0)** is a rotation of 45 degrees about the z-axis.
- If **angle = 0.0** the command has no effect.

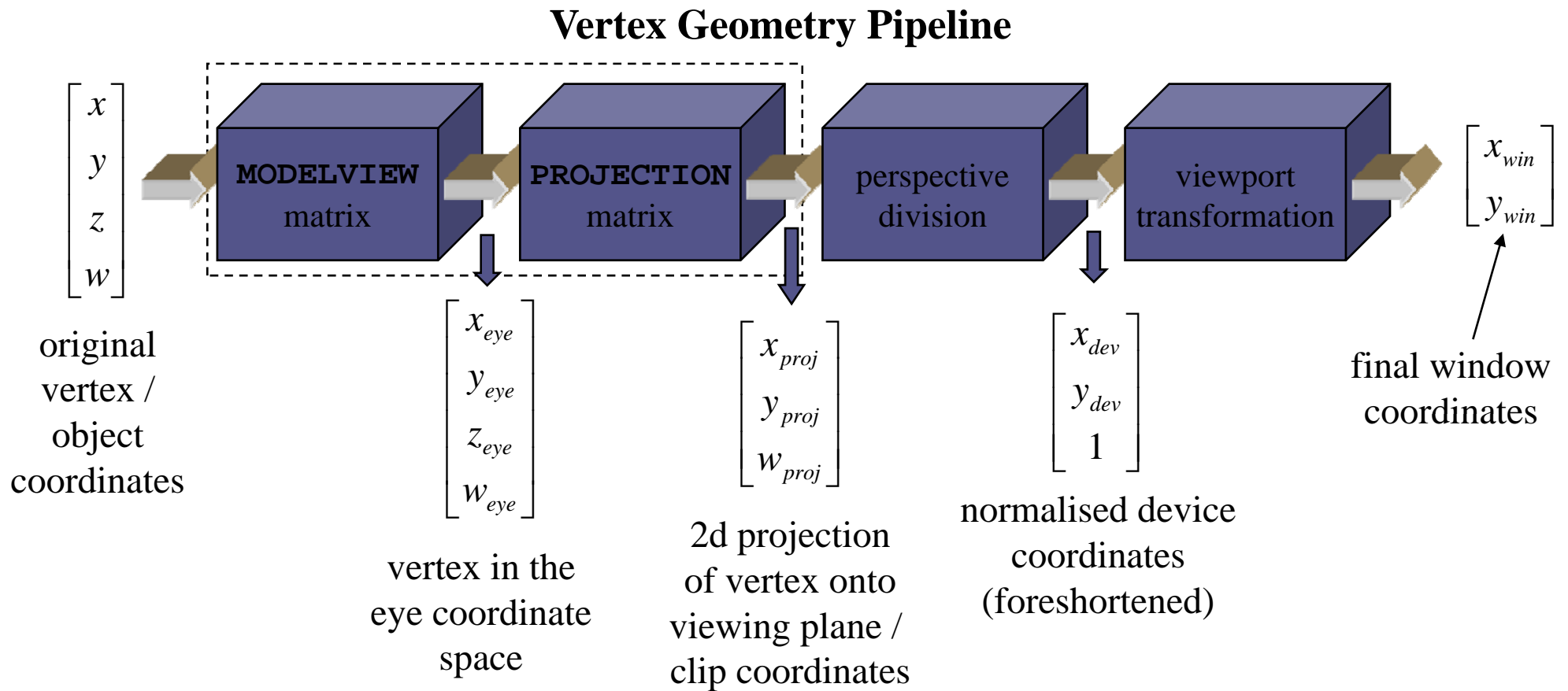
# Scale

- `glScalef(sx, sy, sz)`
  - apply scaling of  $s_x$  in  $x$  direction,  $s_y$  in  $y$  direction and  $s_z$  in  $z$  direction (note that these values specify the *diagonal* of a required matrix)
  - Multiplies the current matrix by a matrix that stretches, shrinks, or reflects an object along the axes. Each  $x$ -,  $y$ -, and  $z$ -coordinate of every point in the object is multiplied by the corresponding argument  $x$ ,  $y$ , or  $z$ .
  - With the local coordinate system approach, the local coordinate axes are stretched, shrunk, or reflected by the  $x$ -,  $y$ -, and  $z$ -factors, and the associated object is transformed with them.

# Transformations in OpenGL®

- OpenGL defines 3 matrices for manipulation of 3D scenes:
  - **GL\_MODELVIEW**: manipulates the view and models simultaneously
  - **GL\_PROJECTION**: performs 3D 2D projection for display
  - **GL\_TEXTURE**: for manipulating textures prior to mapping on objects
- Each acts as a state parameter; once set it remains until altered.
- Having defined a **GL\_MODELVIEW** matrix, all subsequent vertices are created with the specified transformation.
- Matrix transformation operations apply to the currently selected system matrix:
  - use **glMatrixMode(GL\_MODELVIEW)** to select modeling matrix

# Stages of Vertex Transformation



# Transformations in OpenGL<sup>®</sup>

- The **MODELVIEW** matrix is a *4x4 affine transformation matrix* and therefore has *12 degrees of freedom*:

$$\begin{bmatrix} a & b & c & t_x \\ d & e & f & t_y \\ g & h & i & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

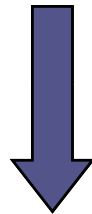
- The MODELVIEW matrix is used for both the model and the camera transformation
  - rotating the model is equivalent to rotating the camera in the opposite direction  $\Rightarrow$  OpenGL uses the same transformation matrix
  - this sometimes causes confusion!

# Transformations in OpenGL®

- Each time an OpenGL transformation **M** is called the current **MODELVIEW** matrix **C** is altered:

$$\mathbf{v}' = \mathbf{C}\mathbf{v} \quad \longrightarrow \quad \mathbf{v}' = \mathbf{C}\mathbf{M}\mathbf{v}$$

```
glTranslatef(1.5, 0.0, 0.0);  
glRotatef(45.0, 0.0, 0.0, 1.0);
```



$$\mathbf{v}' = \mathbf{C}\mathbf{T}\mathbf{R}\mathbf{v}$$

# Example

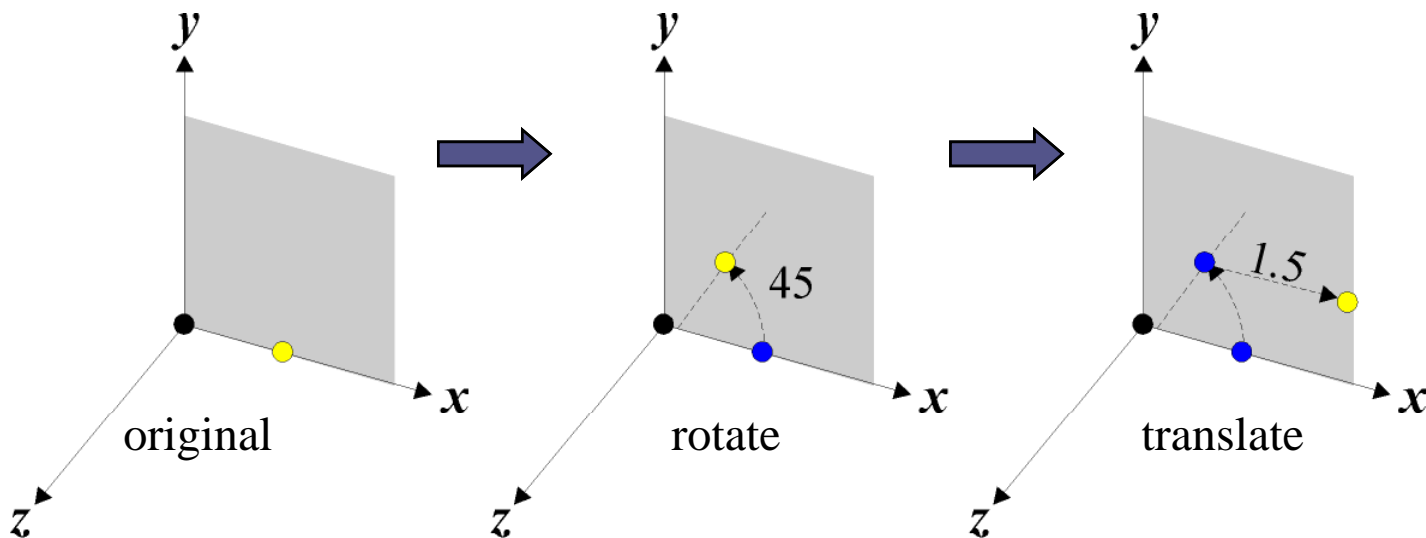
- Draw a single point using three transformations:

```
glMatrixMode(GL_MODELVIEW);  
glLoadIdentity();  
glMultMatrix(N);  
glMultMatrix(M);  
glMultMatrix(L);  
glBegin(GL_POINTS);  
glVertex3f(v);  
glEnd();
```

# Transformations in OpenGL®

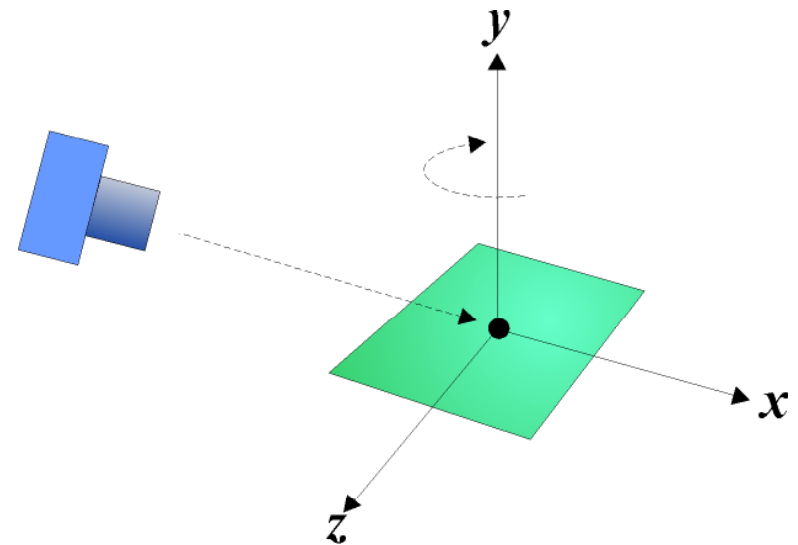
- Transformations are applied *in the order specified* (with respect to the vertex) which appears to be in reverse:

```
glMatrixMode(GL_MODELVIEW);  
glLoadIdentity();  
glTranslatef(1.5, 0.0, 0.0);  
glRotatef(45.0, 0.0, 0.0, 1.0);  
glVertex3f(1.0, 0.0, 0.0);
```



# Camera & Model Transformations

- Given that camera and model transformations are specified using a single matrix we must consider the effect of these transformations on the *coordinate frames* of the camera and models.
- Assume we wish to *orbit* an object at a *fixed orientation*:
  - translate object away from camera
  - rotate around X to look at top of object
  - then pivot around object's Y in order to orbit properly.

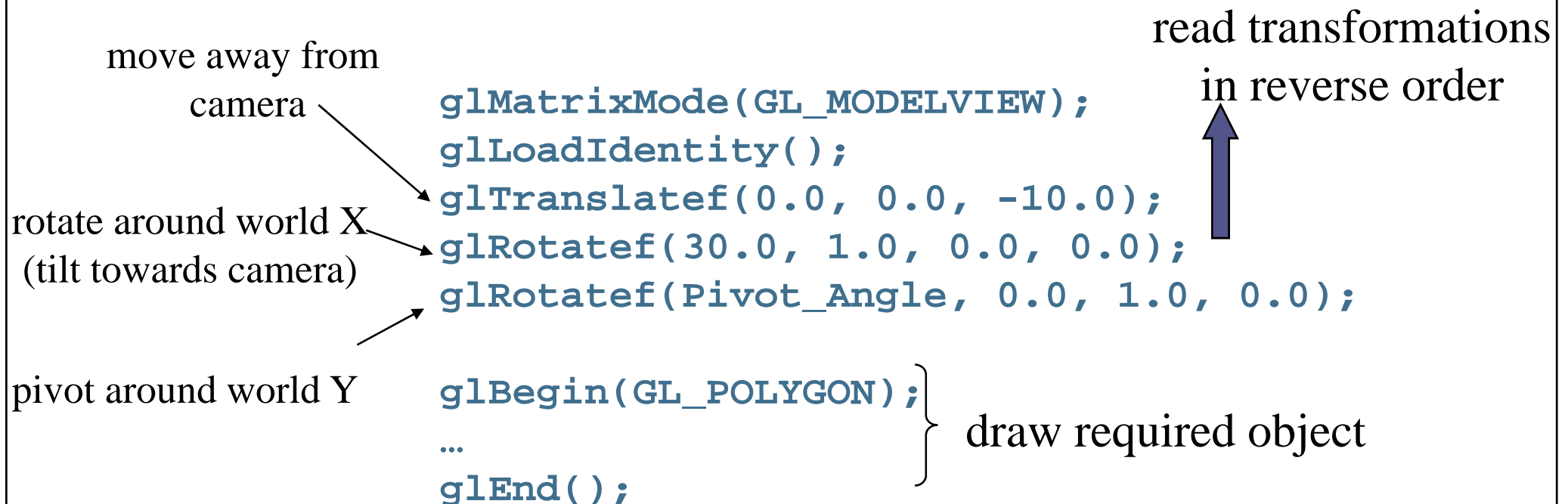


# Camera & Model Transformations

- If the `GL_MODELVIEW` matrix is an identity matrix then the camera frame and the model frame are the same.
  - i.e. they are specified using the same co-ordinate system
- If we issue command `glTranslatef(0.0,0.0,-10.0);` and then create the model:
  - a vertex at  $[0, 0, 0]$  in the model will be at  $[0, 0, -10]$  in the camera frame
  - i.e. we have moved the object away from the camera
- This may be viewed conceptually in 2 ways:
  - we have positioned the object with respect to the world frame
  - we have moved the world-frame with respect to the camera frame

# Camera & Model Transformations

- The first interpretation (the camera view / fixed coordinate system approach) is:



# Camera & Model Transformations

- The second interpretation (using a local frame view) is:

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glTranslatef(0.0, 0.0, -10.0);
glRotatef(30.0, 1.0, 0.0, 0.0);
glRotatef(Pivot_Angle, 0.0, 1.0, 0.0);

glBegin(GL_POLYGON);
...
glEnd();
```

clear previous transformations

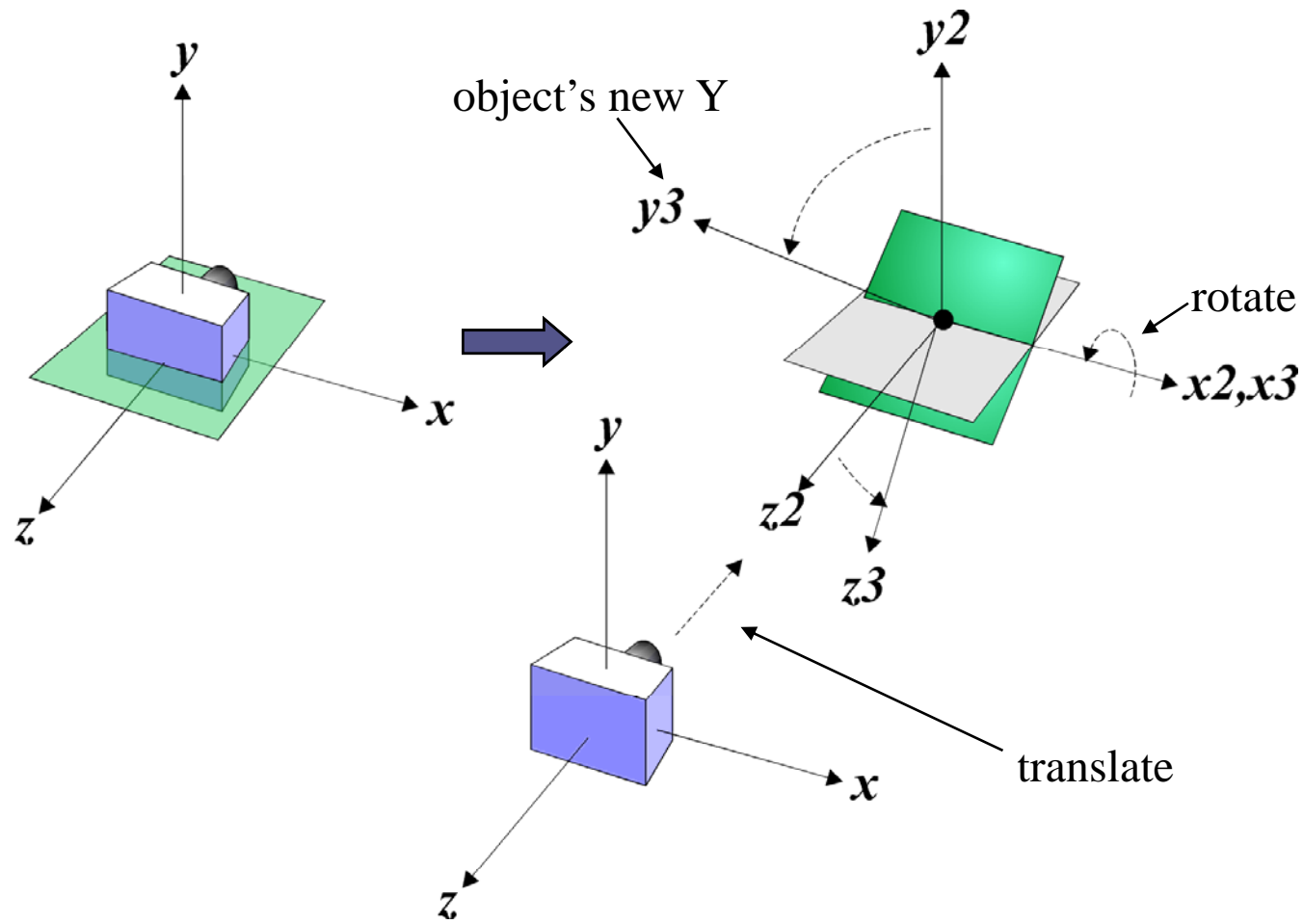
move object frame away from camera

rotate object into view

pivot around object's Y axis

draw required object

# Camera & Model Transformations



# Camera & Model Transformations

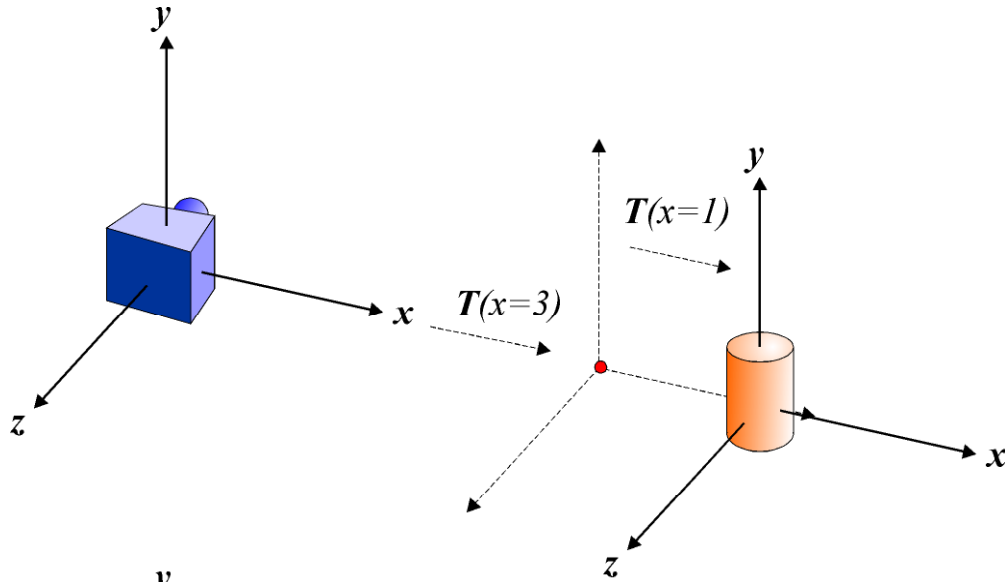
- A “local frame view” is usually adopted as it extends naturally to the specification of *hierarchical model frames*.
- This allows creation of *jointed assemblies*
  - articulated figures (animals, robots etc.)
- In the hierarchical model, each sub-component has its own *local frame*.
- Changes made to the *parent frame* are propagated down to the *child frames* (thus all models in a branch are globally controlled by the parent).
- This simplifies the specification of *animation*.



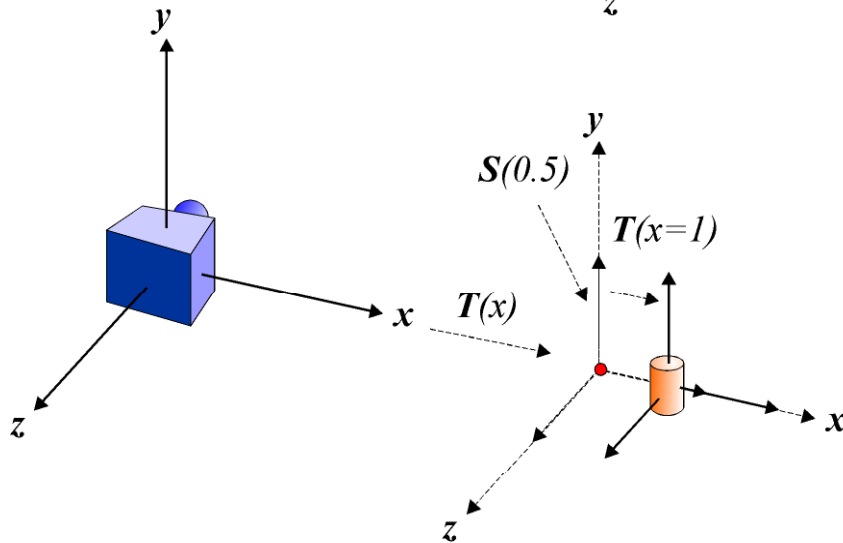
# Model Transformations

- As the **MODELVIEW** matrix is changed objects are created with respect to a changing transformation.
- This is often termed the *current transformation matrix* or *CTM*.
- The *CTM* behaves like a *3D pointer*, selecting new positions and orientations for the creation of geometries.
- The only complication with this view is that each new *CTM* is derived from a previous *CTM*, i.e. all *CTMs* are specified *relative* to previous versions.
- A *scaling transformation* can cause some confusion.

# Scaling Transformation and the CTM



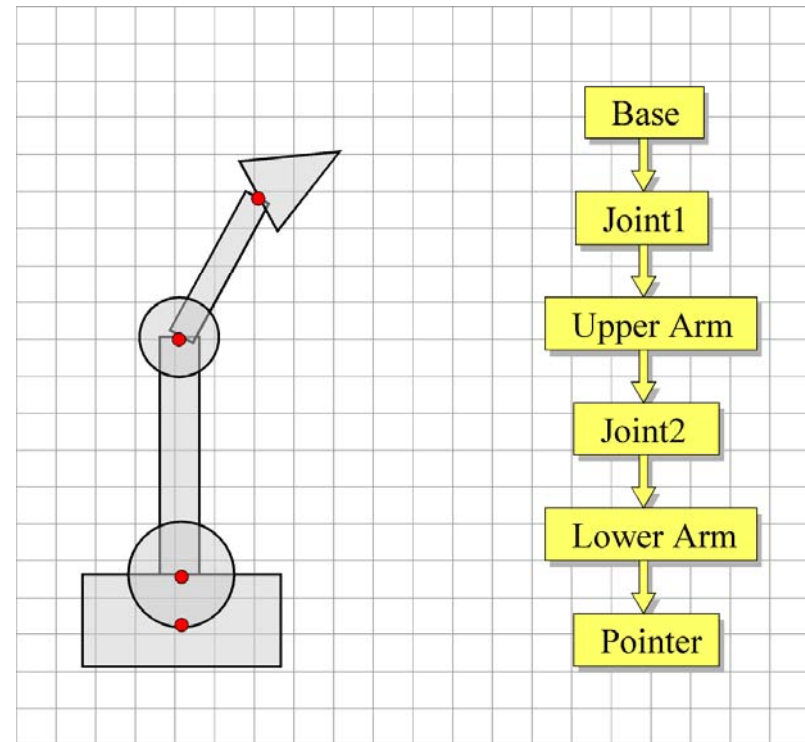
```
glMatrixMode(GL_MODELVIEW);  
glLoadIdentity();  
glTranslatef(3, 0, 0);  
glTranslatef(1, 0, 0);  
  
gluCylinder(...);
```



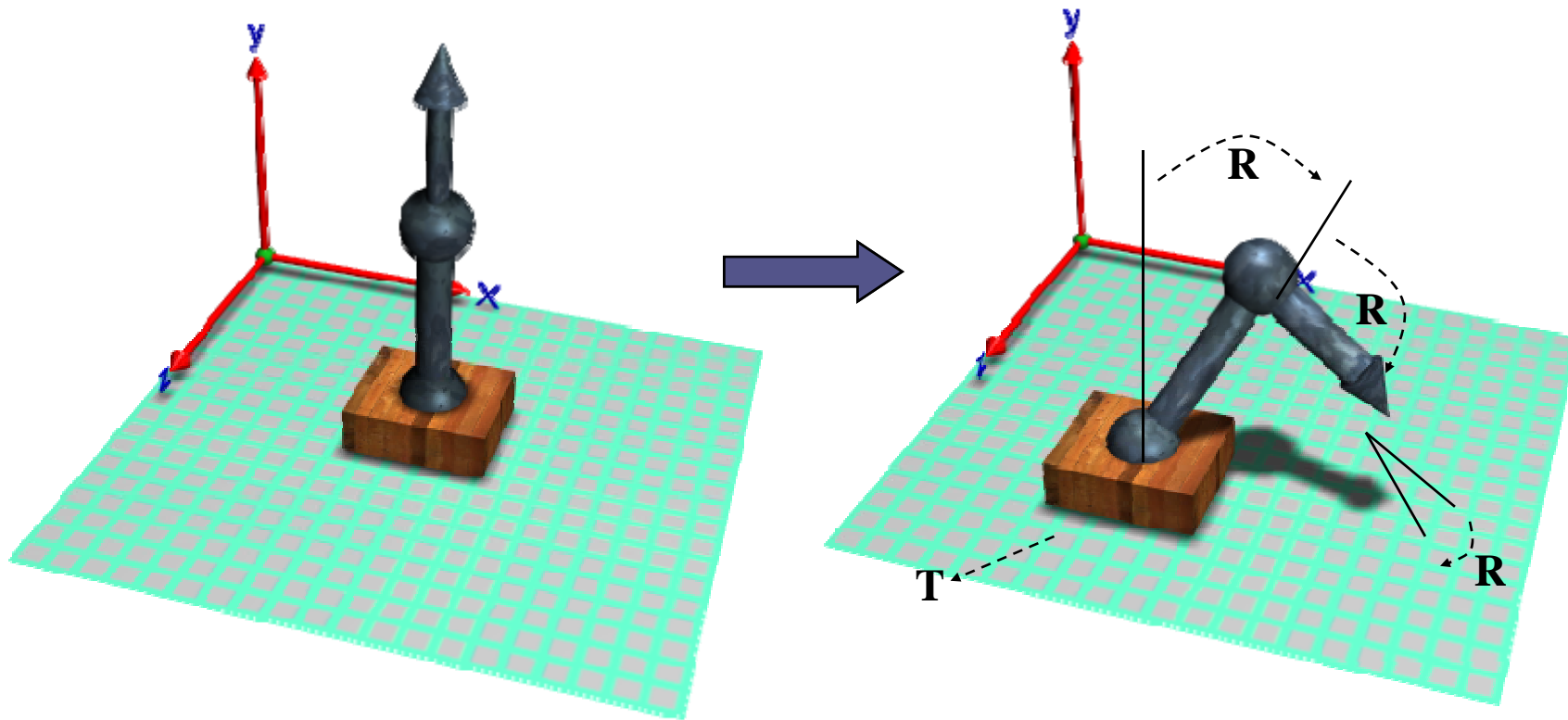
```
glMatrixMode(GL_MODELVIEW);  
glLoadIdentity();  
glTranslatef(3, 0, 0);  
glScalef(0.5, 0.5, 0.5);  
glTranslatef(1, 0, 0);  
  
gluCylinder(...);
```

# Hierarchical Transformations

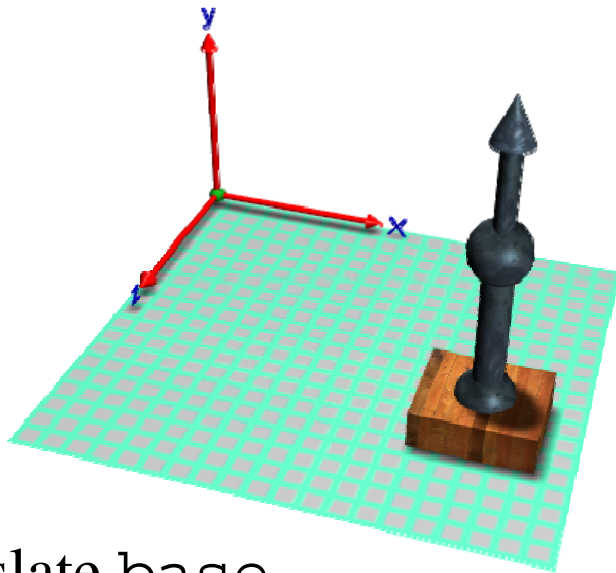
- For geometries with an implicit *hierarchy* we wish to associate local frames with sub-objects in the assembly.
- *Parent-child frames* are related via a transformation.
- Transformation linkage is described by a *tree*:
- Each node has its own *local co-ordinate system*.



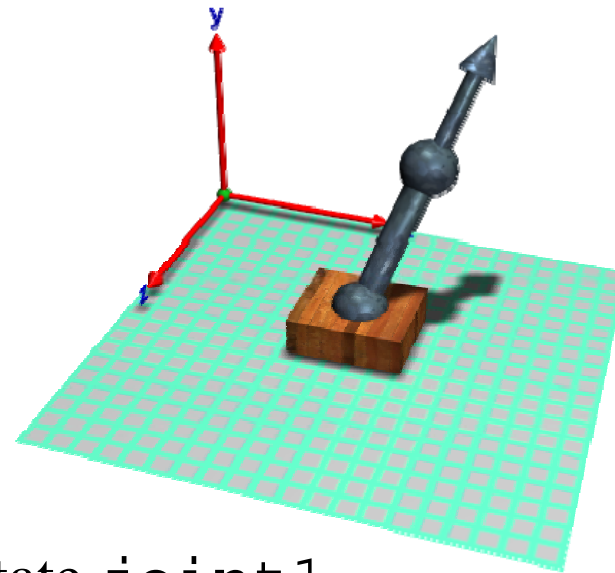
# Hierarchical Transformations



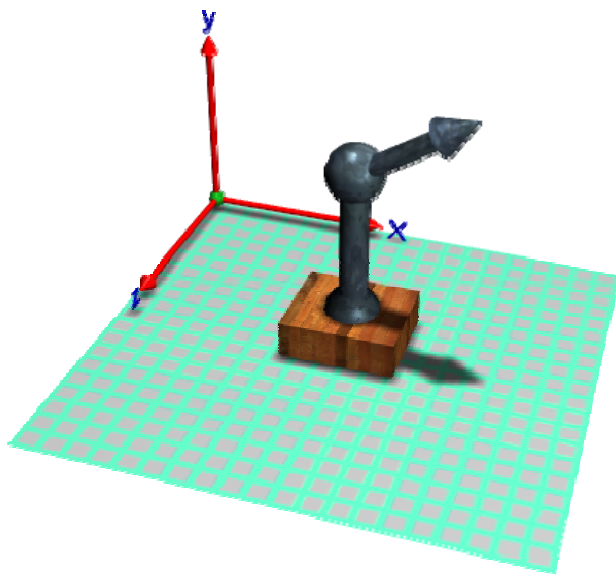
Hierarchical transformation allow independent control over sub-parts of an assembly



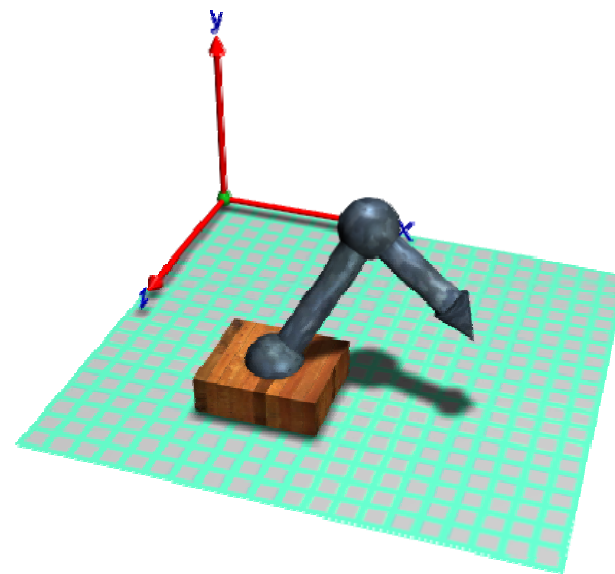
translate base



rotate joint1



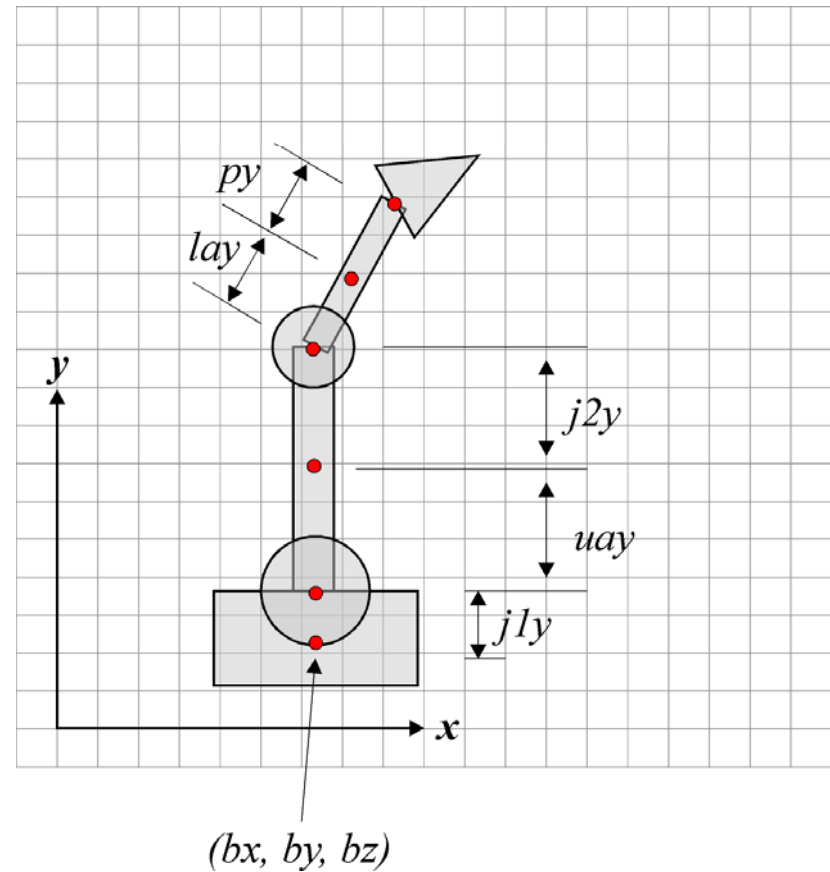
rotate joint2



complex hierarchical transformation

# OpenGL<sup>®</sup> Implementation

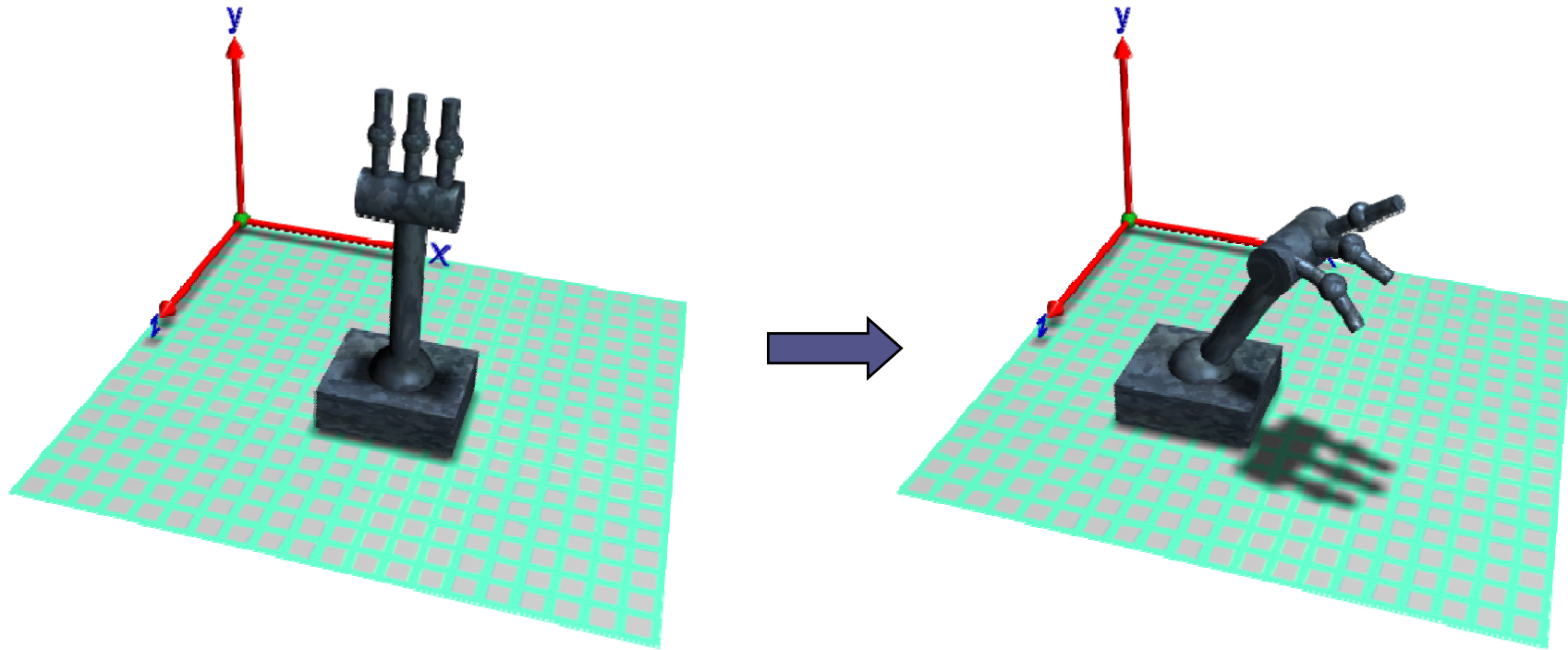
```
glMatrixMode(GL_MODELVIEW);  
glLoadIdentity();  
glTranslatef(bx, by, bz);  
    create_base();  
glTranslatef(0, j1y, 0);  
glRotatef(joint1_orientation);  
    create_joint1();  
glTranslatef(0, uay, 0);  
    create_upperarm();  
glTranslatef(0, j2y);  
glRotatef(joint2_orientation);  
    create_joint2();  
glTranslatef(0, lay, 0);  
    create_lowerarm();  
glTranslatef(0, py, 0);  
glRotatef(pointer_orientation);  
    create_pointer();
```



# Hierarchical Transformations

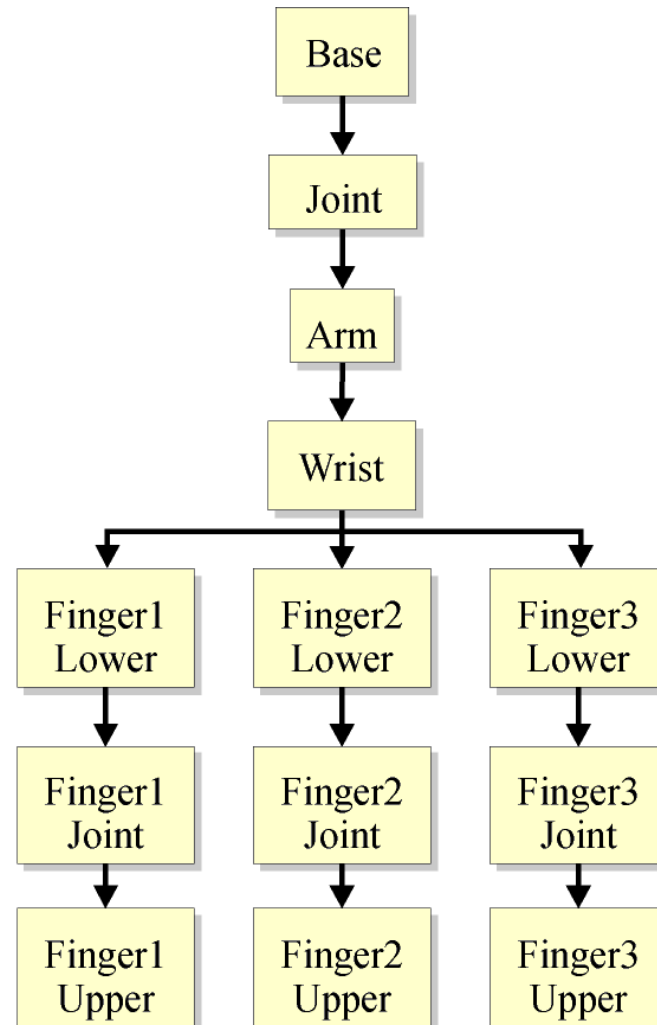
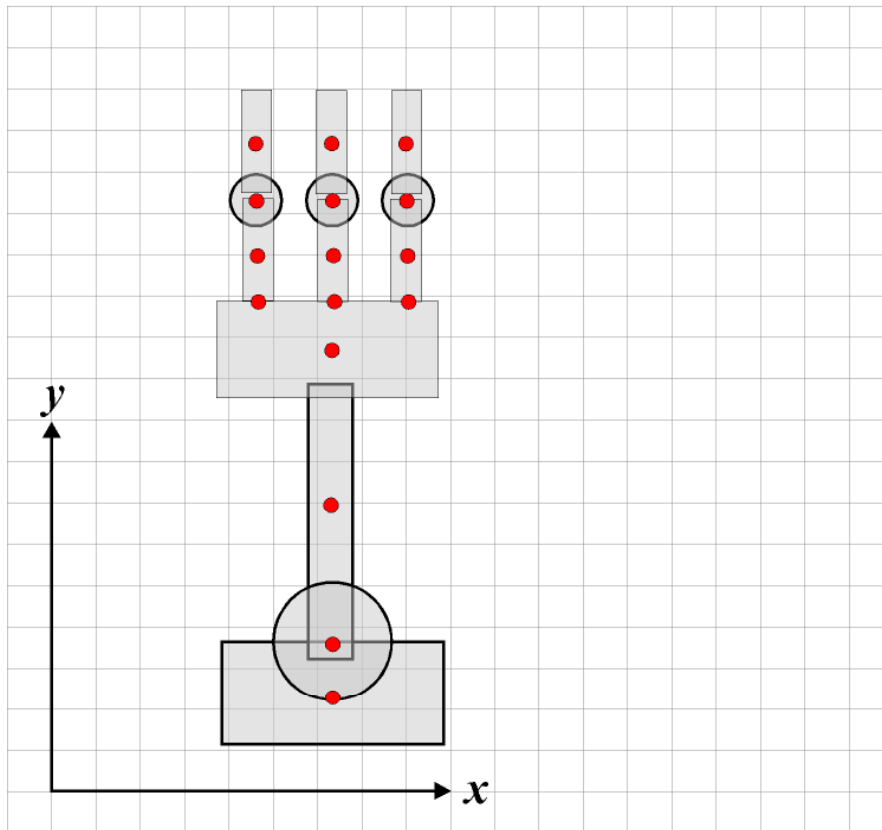
- The previous example had simple *one-to-one* parent-child linkages.
- In general there may be many *child frames* derived from a single parent frame.
- We need some mechanism to remember the parent frame and return to it when creating new children.
- OpenGL provide a matrix stack for just this purpose:
  - **glPushMatrix()** saves the *CTM*
  - **glPopMatrix()** returns to the last saved *CTM*

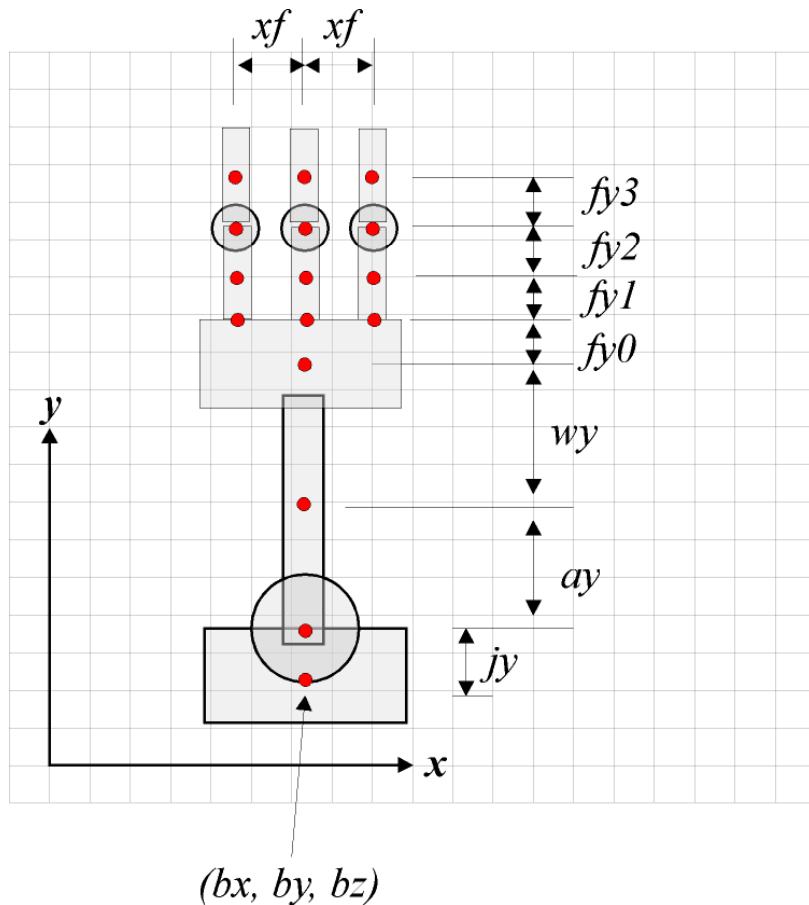
# Hierarchical Transformations



Each finger is a child of the parent (wrist)  
⇒ independent control over the orientation  
of the fingers relative to the wrist

# Hierarchical Transformations





```

glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glTranslatef(bx, by, bz);
    create_base();
glTranslatef(0, jy, 0);
glRotatef(joint1_orientation);
    create_joint1();
glTranslatef(0, ay, 0);
    create_upperarm();
glTranslatef(0, wy);
glRotatef(wrist_orientation);
    create_wrist();
glPushMatrix(); // save frame
glTranslatef(-xf, fy0, 0);
glRotatef(lowerfinger1_orientation);
glTranslatef(0, fy1, 0);
    create_lowerfinger1();
glTranslatef(0, fy2, 0);
glRotatef(upperfinger1_orientation);
    create_fingerjoint1();
glTranslatef(0, fy3, 0);
    create_upperfinger1();
glPopMatrix(); // restore frame
glPushMatrix();
    // do finger 2...
glPopMatrix();
glPushMatrix();
    // do finger 3...
glPopMatrix();

```

Finger1