

CS 428: Fall 2009

Introduction to Computer Graphics

Transformations in OpenGL +
hierarchical modeling

Review of affine transformations

- Use projective geometry → staple of CG

Euclidean	Homogeneous
(x, z)	(x, y, w)
(x, y, z)	(x, y, z, w)

- $w = 0$ vector
- $w = 1$ point
- Divide by $w \neq 0$ to get a point in the $w = 1$ plane

General 2D affine transformation

- Matrix

$$\begin{bmatrix} p'_x \\ p'_y \\ 1 \end{bmatrix} = \begin{bmatrix} a & c & e \\ b & d & f \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ 1 \end{bmatrix} \quad \text{- points translate by } \begin{bmatrix} e \\ f \end{bmatrix}$$

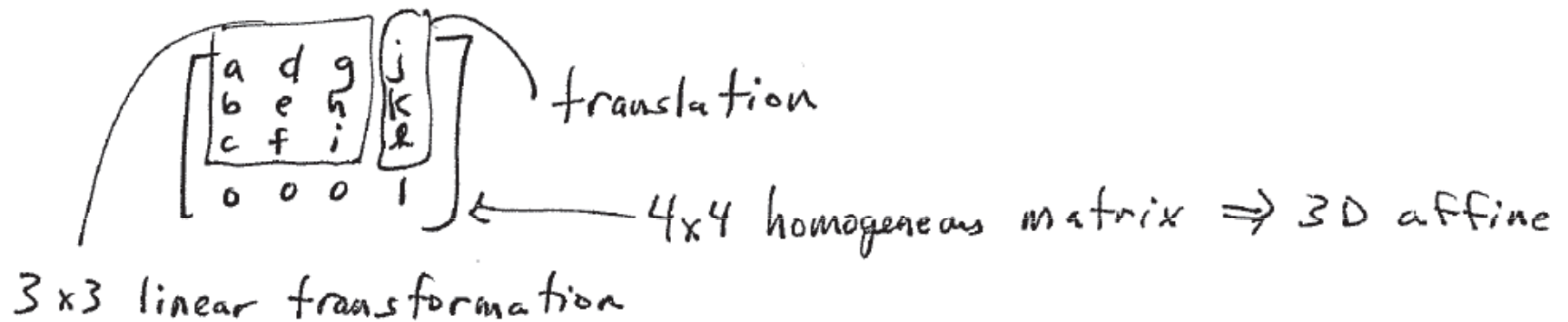
$$\begin{bmatrix} v'_x \\ v'_y \\ 0 \end{bmatrix} = \begin{bmatrix} v_x \\ v_y \\ 0 \end{bmatrix} \quad \text{- vectors don't translate}$$

\swarrow 3×3 homogeneous matrix \Rightarrow 2D affine

linear part affects both
 $\begin{bmatrix} a & c \\ b & d \end{bmatrix}$

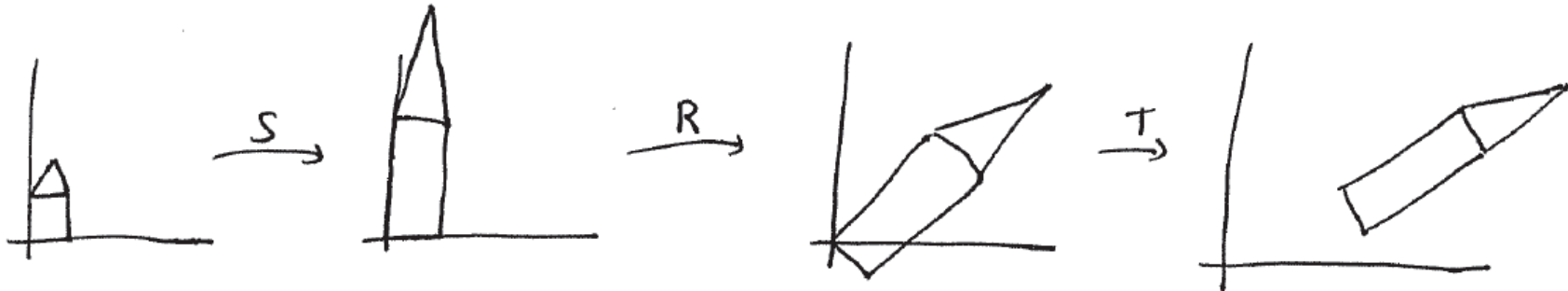
General 3D affine transformation

- Matrix



- Translation in 2D = shear with respect to (w.r.t.) the x/y plane
- Translation in 3D = shear in 4D w.r.t x/y/z hyperplane

OpenGL transformation order



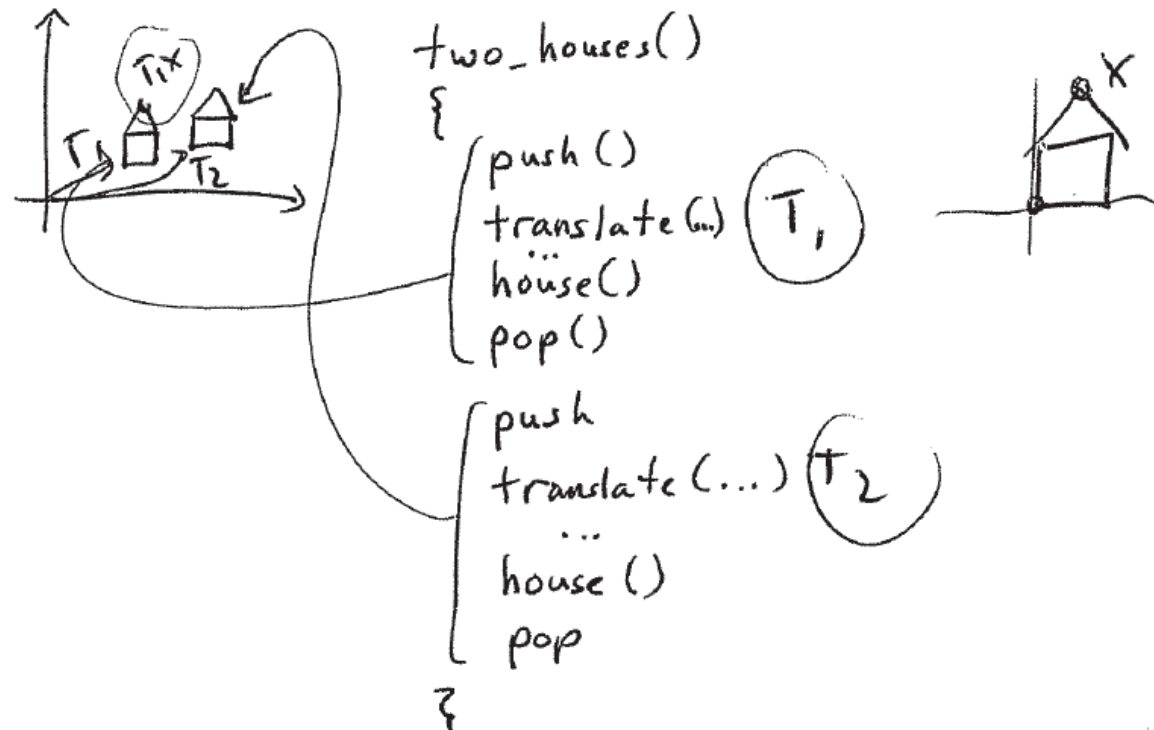
`glLoadIdentity ()` ← **I**
`glTranslatef (...)` ← **IT**
`glRotatef (...)` ← **ITR**
`glScalef (...)` ← **ITRS**
`house ()`

- Seen *backwards* in code?

OpenGL transformation order

- OpenGL uses **right multiplication**
 - Better for hierarchical object management → common in CG

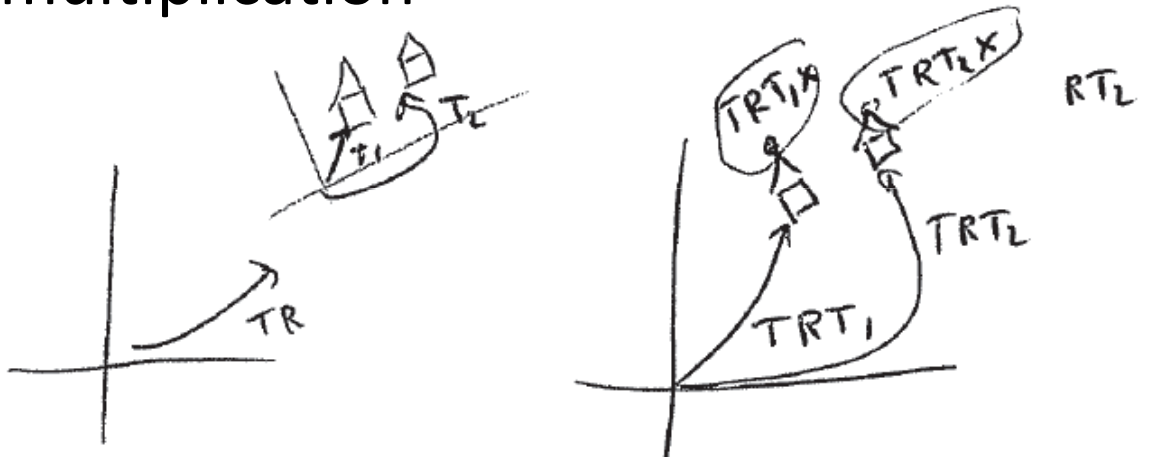
- Example



OpenGL transformation order

- Now suppose we want to move both houses
 - Easy with right multiplication

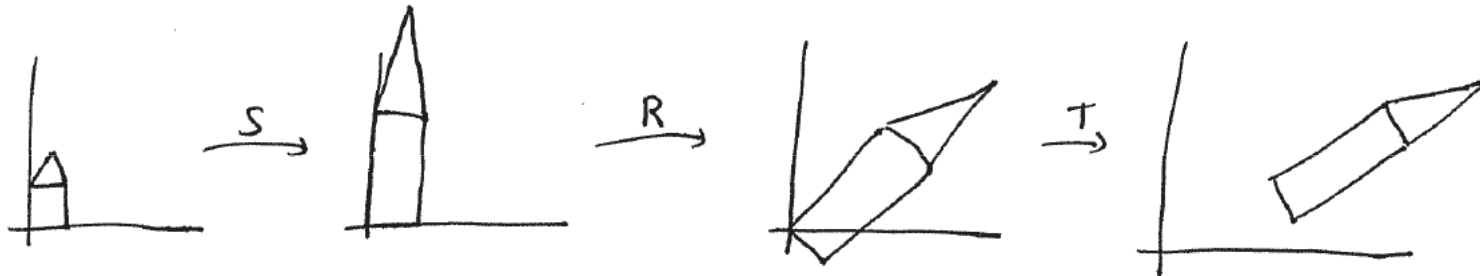
translate(...) T
rotate(...) R
two_houses()



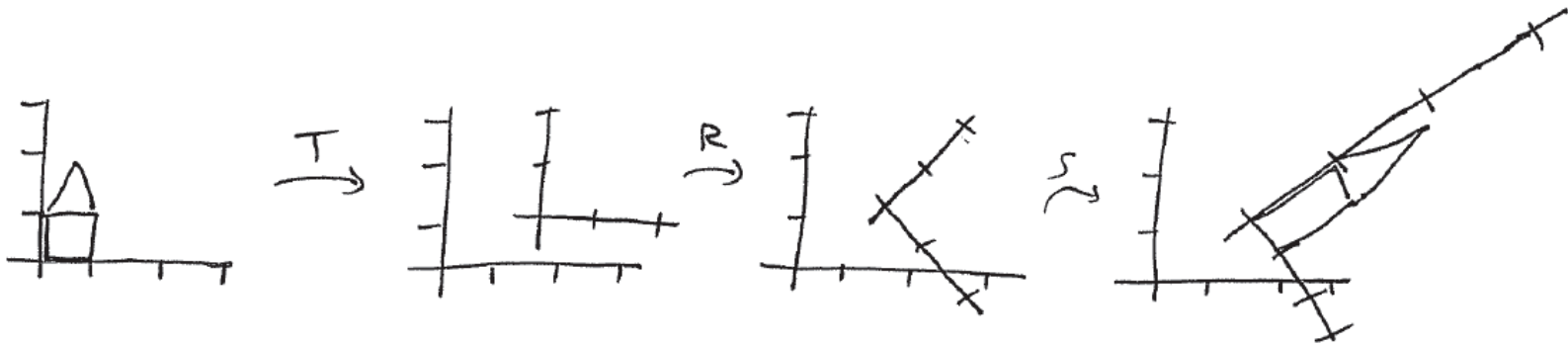
- Can't do this with left multiplication without changing **two_houses()**
 - Would need to store intermediate results until done transforming

OpenGL transformation order

- Earlier picture thinks about a global, fixed coordinate system

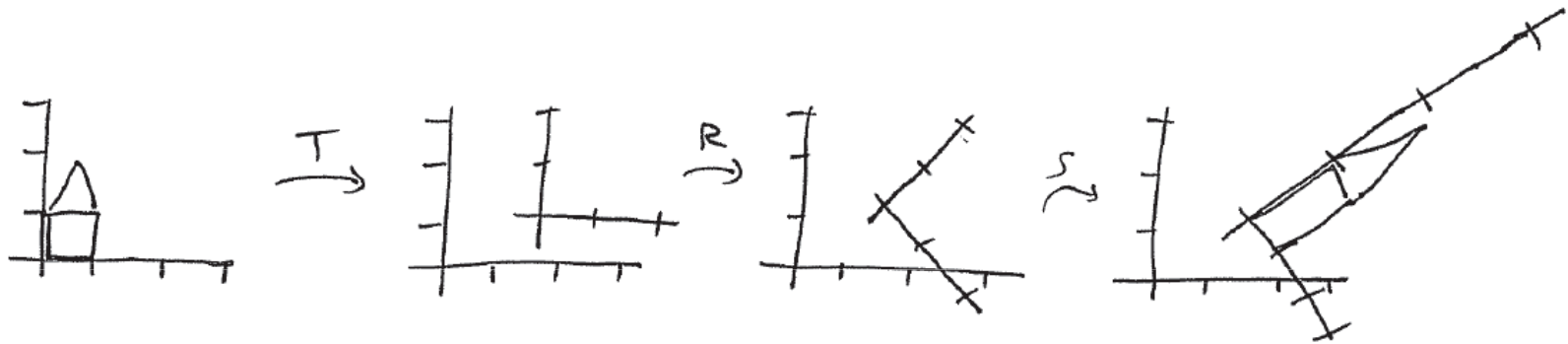


- What about a local, changing coord. system?



OpenGL transformation order

- What about a local, changing coord. system?



- Coordinate system (local) containing objects is transformed (**all** non-rigid transformations!)
- Same order as OpenGL

OpenGL transformation order

- Example: in place rotation
 - Translate center point \mathbf{p} to origin

- Rotate

- Translate center point \mathbf{p} back

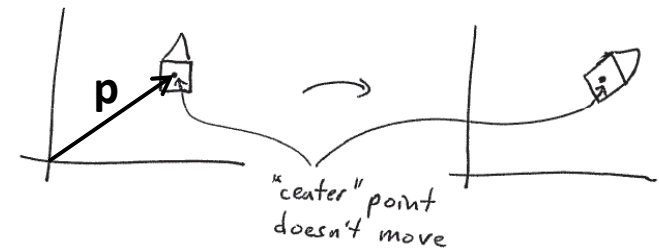
- `glLoadIdentity()`

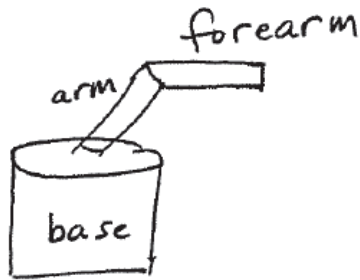
```
glTranslatef( $p_x$ ,  $p_y$ ,  $p_z$ )
```

```
glRotatef(...)
```

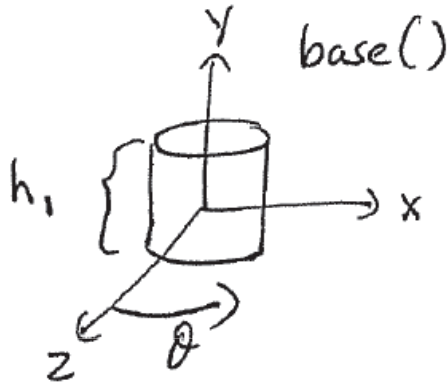
```
glTranslatef( $-p_x$ ,  $-p_y$ ,  $-p_z$ )
```

```
house()
```

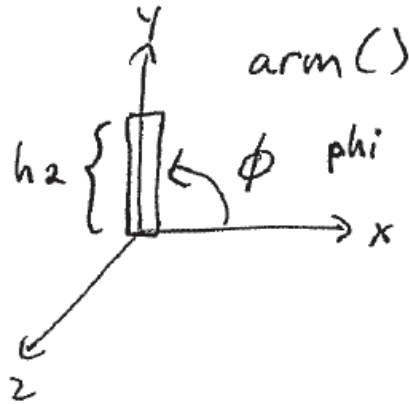




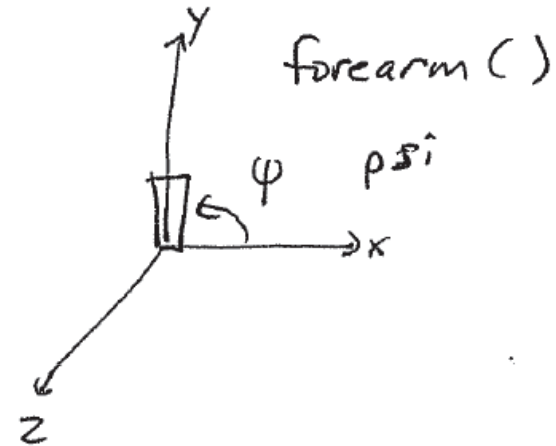
Simple robot example



$$R_y(\theta)$$



$$T(0, h_2, 0) R_z(\phi)$$



$$T(0, h_2, 0) R(\psi)$$

- In code (think OpenGL state!)

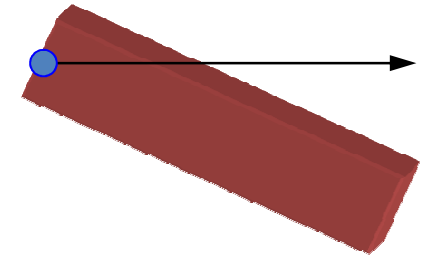
```
robot.push(c)
[gl Rotate f(θ, 0, 1, 0)
 base()]
```

```
[gl Translate f(0, h1, 0)
 gl Rotate f(φ, 0, 0, 1)
 arm()]
```

```
[gl Translate f(0, h2, 0)
 gl Rotate f(ψ, 0, 0, 1)
 forearm()
 pop()]
```

One more robot example

- `glRotatef(angle, 0.0, 0.0, 1.0)`



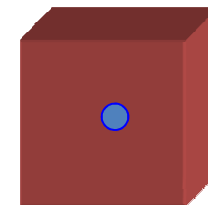
- `glTranslatef(1.0, 0.0, 0.0);`



- `glScalef(2.0, 0.5, 0.5);`

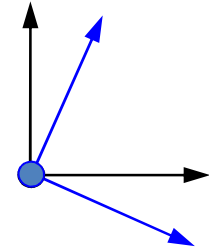


- `glutSolidCube(1.0);`

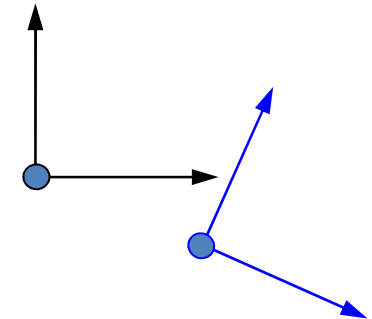


One more robot example

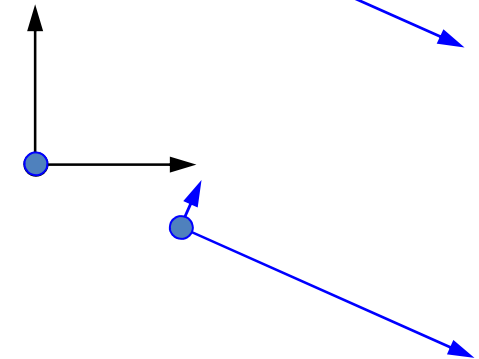
- `glRotatef(angle, 0.0, 0.0, 1.0)`



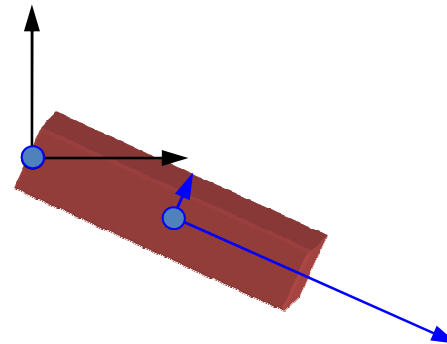
- `glTranslatef(1.0, 0.0, 0.0);`



- `glScalef(2.0, 0.5, 0.5);`



- `glutSolidCube(1.0);`

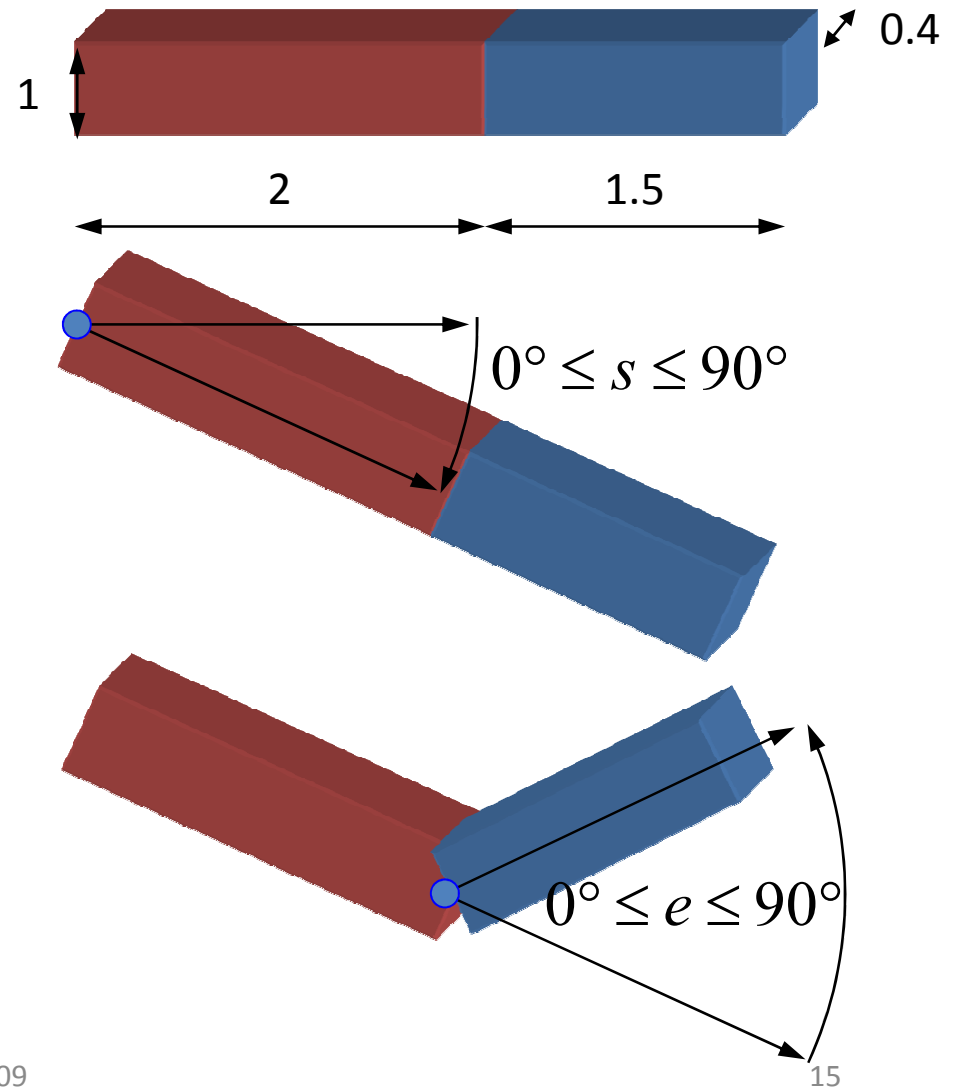


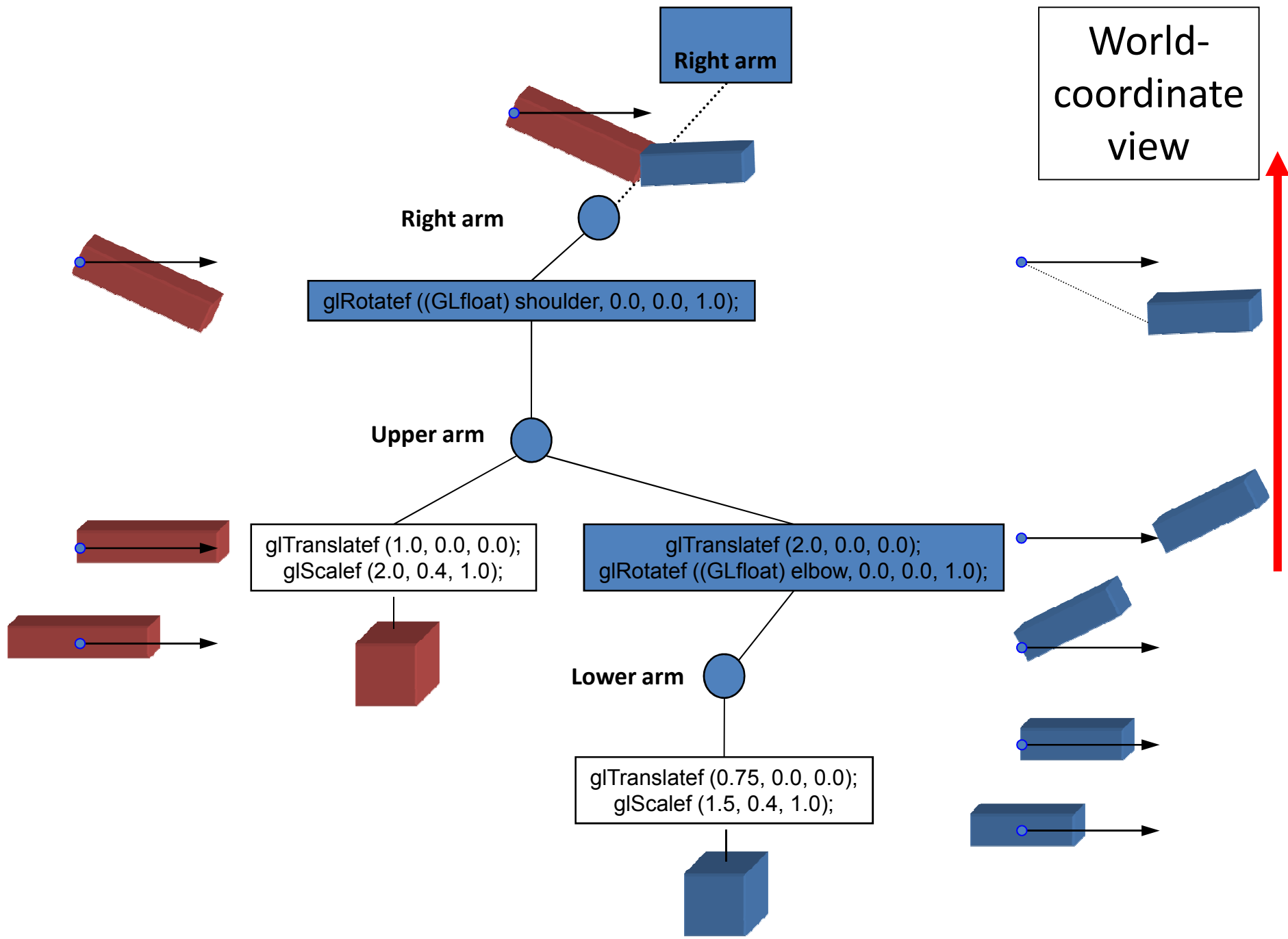
OpenGL matrix stack

- The current matrix is on the top of the stack
- Intermediate storage needed for matrices
- Matrix stack
 - **glPushMatrix() ;**
 - Copies the top matrix (= stores current matrix)
 - **glPopMatrix() ;**
 - Removes the top matrix (= use old matrix)
- Push and pop bracket transformations
- OpenGL commonly holds up to 32 matrices

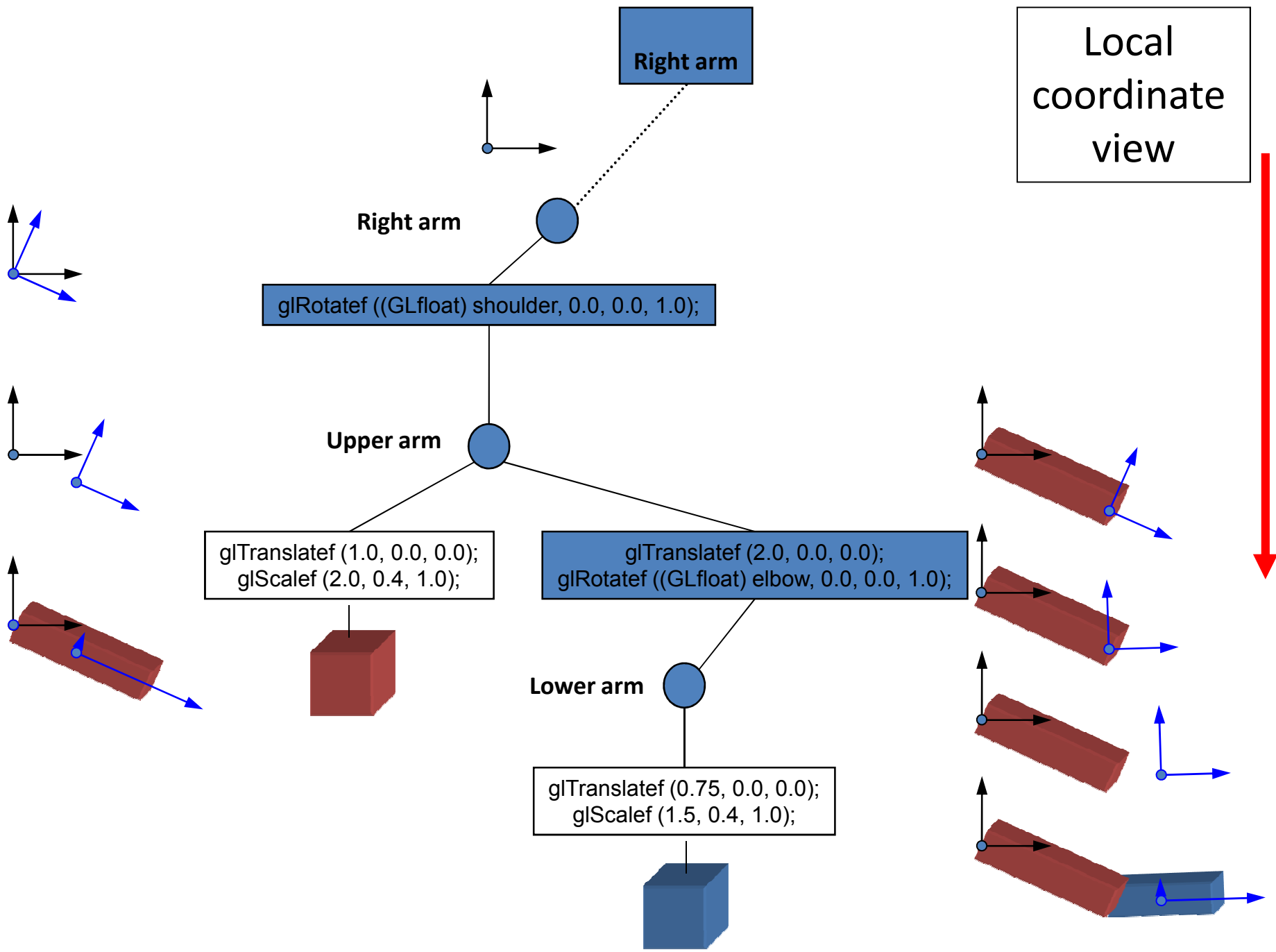
One more robot example

- Upper and lower arm as cuboids
- Move parts by transforming shoulder (s) and elbow (e) joints in the allowed value ranges
- Use geometric primitives
gluWireCube (1.0)





Local coordinate view



Same code!

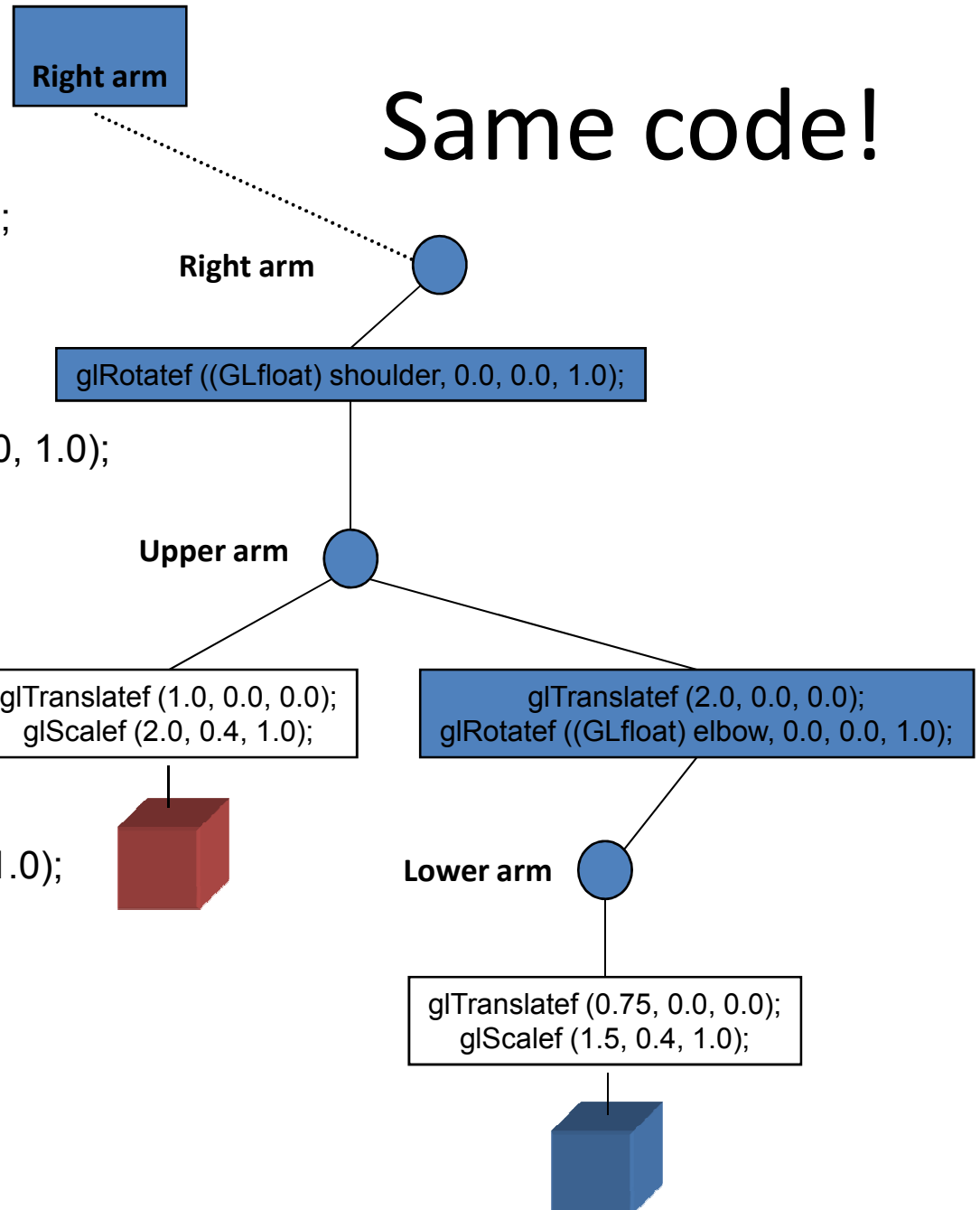
```
glClear (GL_COLOR_BUFFER_BIT);  
glColor3f(0,0,0);  
glLoadIdentity();  
gluLookAt( 0,0,10, 0,0,0, 0,1,0);
```

```
glRotatef ((GLfloat) shoulder, 0.0, 0.0, 1.0);
```

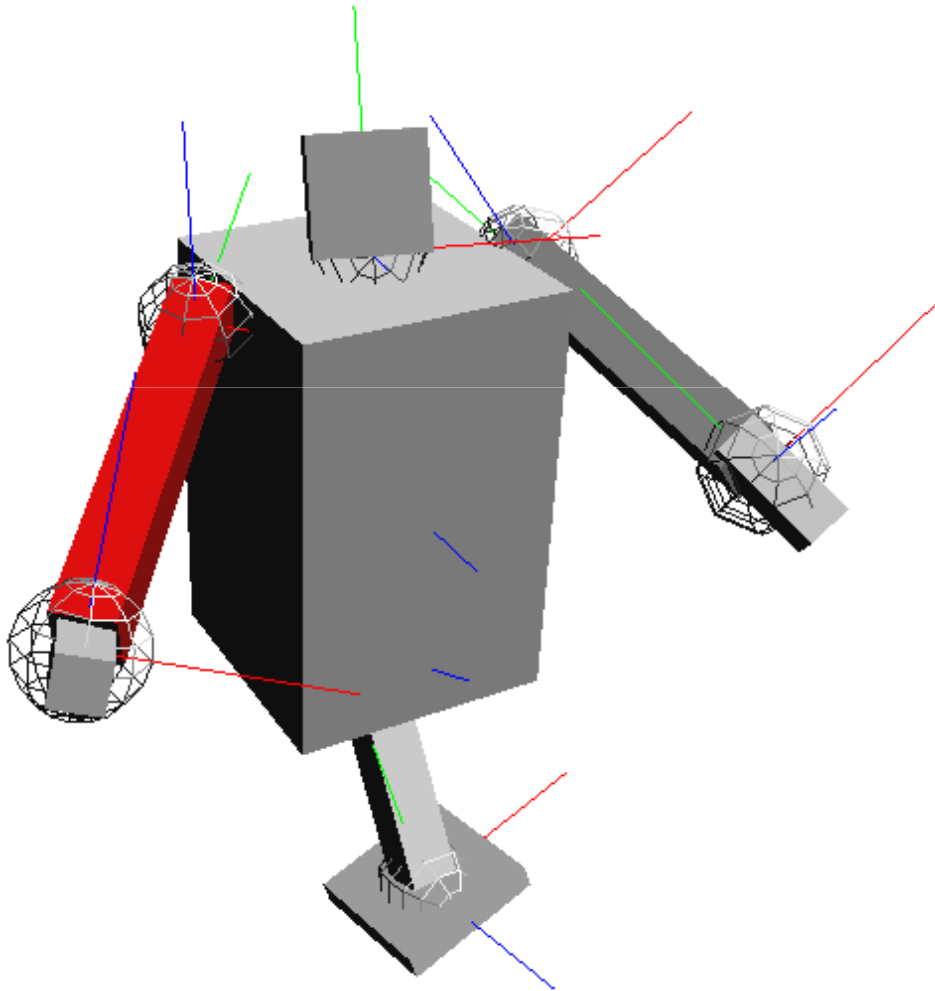
```
glPushMatrix();  
  glTranslatef (1.0, 0.0, 0.0);  
  glScalef (2.0, 0.4, 1.0);  
  glutWireCube (1.0);  
glPopMatrix();
```

```
glTranslatef (2.0, 0.0, 0.0);  
glRotatef ((GLfloat) elbow, 0.0, 0.0, 1.0);
```

```
glTranslatef (0.75, 0.0, 0.0);  
glScalef (1.5, 0.4, 1.0);  
glutWireCube (1.0);  
glFlush();
```



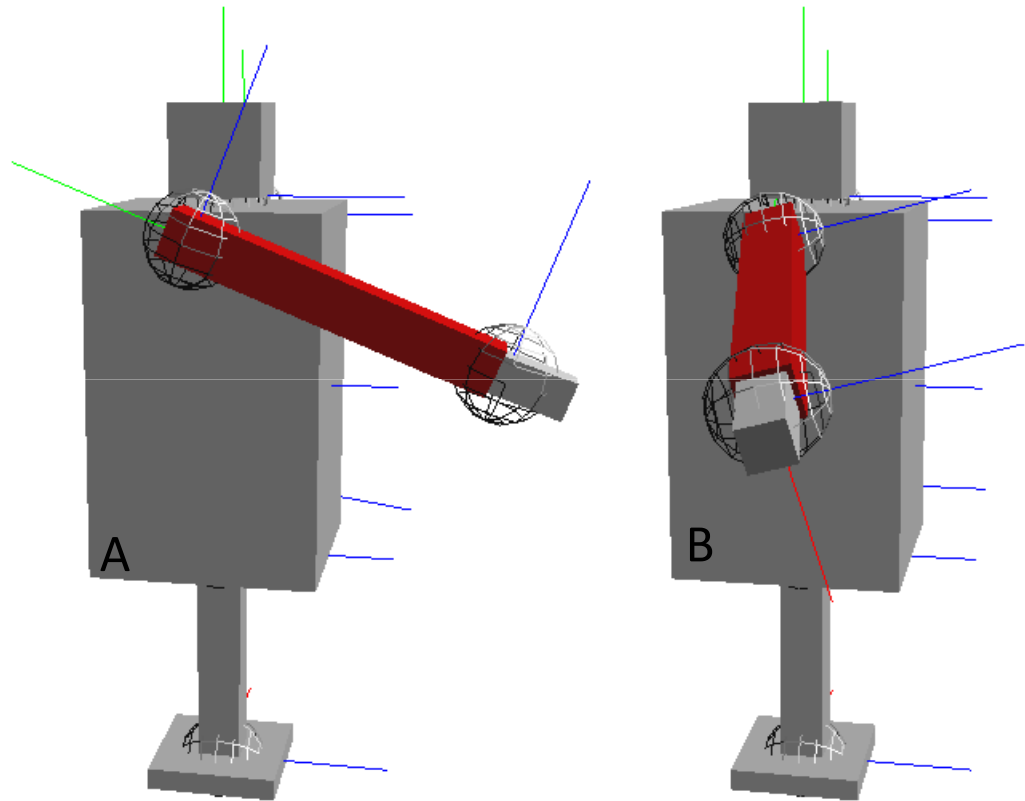
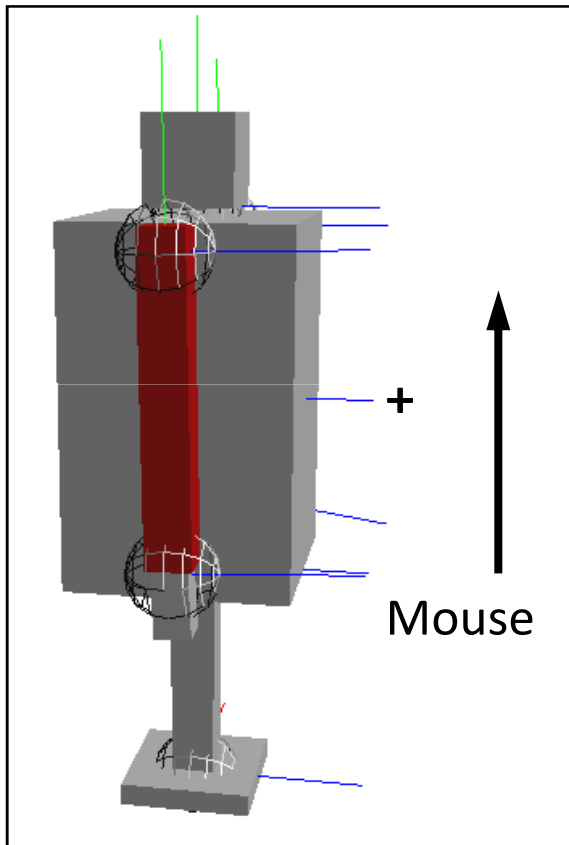
And yet another robot



- Euler angles
 - $\alpha_x, \alpha_y, \alpha_z$ per node
 - Gimbal lock singularity
- User interface problem
 - Rotations are performed in local coordinates (demo)
 - These are independent of the viewing position
 - Can be quite unintuitive

Coordinate system

Where to rotate ?



Result A or B ?

Demo in the application

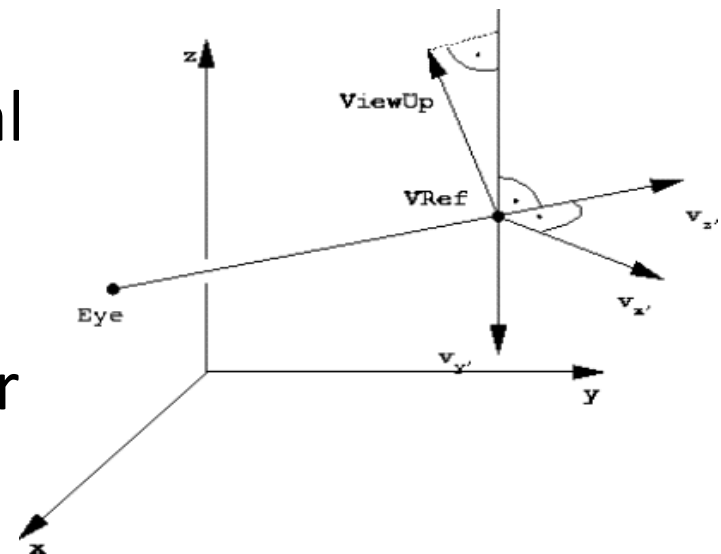
Affine transformations

Per node in the hierarchy

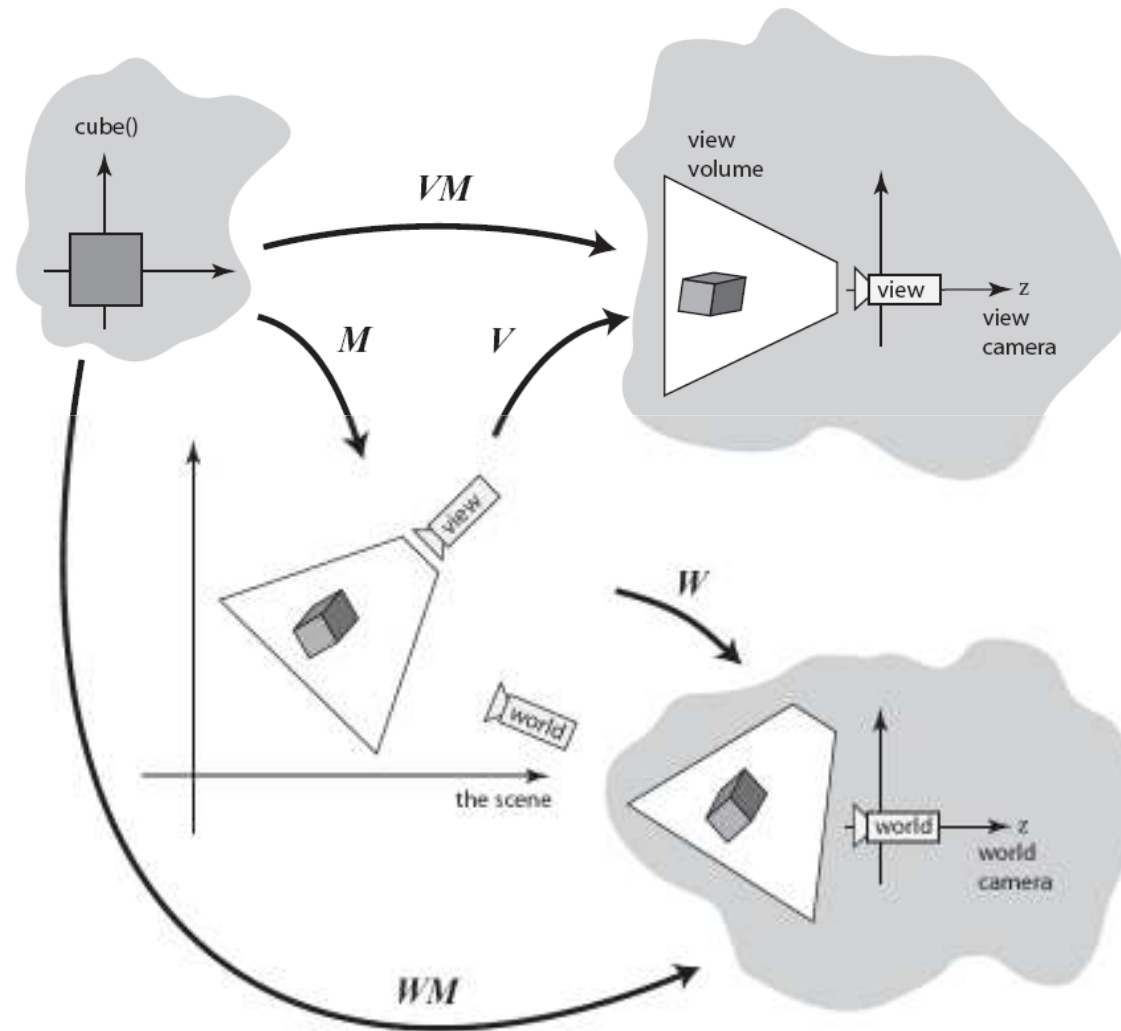
- Solution to the gimbal lock problem
 - Store a transformation matrix \mathbf{M} per node
 - Accumulate small rotations by multiplication
(from left): $\mathbf{M}_{\text{new}} = \mathbf{M}_{\text{rot}} \cdot \mathbf{M}_{\text{old}} = \mathbf{M}_{\text{rx}} \cdot \mathbf{M}_{\text{ry}} \cdot \mathbf{M}_{\text{old}}$
 - Now we can rotate around any axis at any point
`glMultMatrixf (node->getTransformArray ());`
Instead of
`glRotatef (node->rx, 1.0, 0.0, 0.0);`
`glRotatef (node->ry, 0.0, 1.0, 0.0);`
`glRotatef (node->rz, 0.0, 0.0, 1.0);`
 - In this implementation \mathbf{M} only stores the rotation, since the center of rotation is usually shifted

Viewing transformation

- This is also part of the modelview matrix
 - Usually applied to the OpenGL matrix before any other transformation using `gluLookAt(0, 0, 3, 0, 0, 0, 0, 1, 0);`
 - This constructs a matrix which transforms the virtual camera to the origin, looking down negative z, with the specified up-vector
 - (more on this next time)



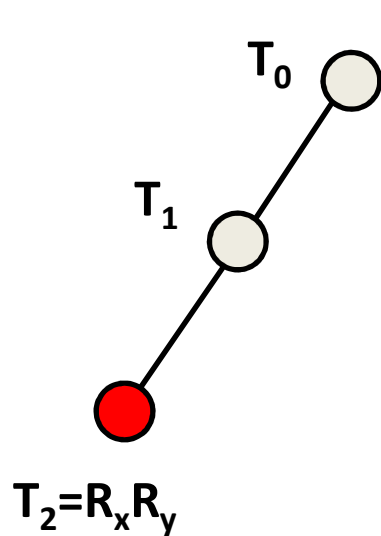
Modelview transformation



Affine Transformations

Relative to the viewer

- Transformations must be applied in the **global** coordinate system



User-defined transformation

$$\begin{aligned} \mathbf{v}' &= T_0 T_1 \mathbf{v} \\ \mathbf{v} &= (T_0 T_1)^{-1} \mathbf{v}' \end{aligned}$$

\mathbf{v}' in global coordinates

\mathbf{v} in local coordinates

Previously:

$$\mathbf{v}' = T_0 T_1 T_2 \mathbf{v} = T_0 T_1 \mathbf{R}_x \mathbf{R}_y \mathbf{v}$$

To rotate in global coordinates:

$$\mathbf{T}_{2,\text{new}} = (T_0 T_1)^{-1} \mathbf{R}_x \mathbf{R}_y T_0 T_1 \mathbf{T}_{2,\text{old}}$$

Affine Transformations

Relative to the viewer

- See „rotation around an arbitrary axis“ in the lecture slides

$$T_{2,\text{new}} = \boxed{(T_0 T_1)^{-1}} \boxed{R_x R_y} \boxed{T_0 T_1} T_{2,\text{old}}$$

Transformation to the local coordinate system

Rotation in the global coordinate system

Transformation to the global coordinate system

Affine Transformations

Relative to the viewer

- **Alternative:** transform global x and y axes to the local coordinate system, and rotate around these ($e_x = (1,0,0)^T$, etc.)

Unit canonical vectors
in local coordinates

$$l_x = (T_0 T_1)^{-1} e_x$$

$$l_y = (T_0 T_1)^{-1} e_y$$

$$T_{2,new} = R_{l_x} R_{l_y} T_{2,old}$$

Affine Transformations

Relative to the viewer

