CS 428: Fall 2010

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 ≠ 0 to get a point in the w = 1 plane

General 2D affine transformation

- Matrix

$$\begin{bmatrix} p_x' \\ p_y' \\ 1 \end{bmatrix} = \begin{bmatrix} a & c & e \\ b & d & f \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ 1 \end{bmatrix}$$ · points translate by $\begin{bmatrix} e \\ f \end{bmatrix}$

$$\begin{bmatrix} v_x' \\ v_y' \\ 0 \end{bmatrix} = \text{“} \begin{bmatrix} v_x \\ v_y \\ 0 \end{bmatrix}$$ · vectors don't translate

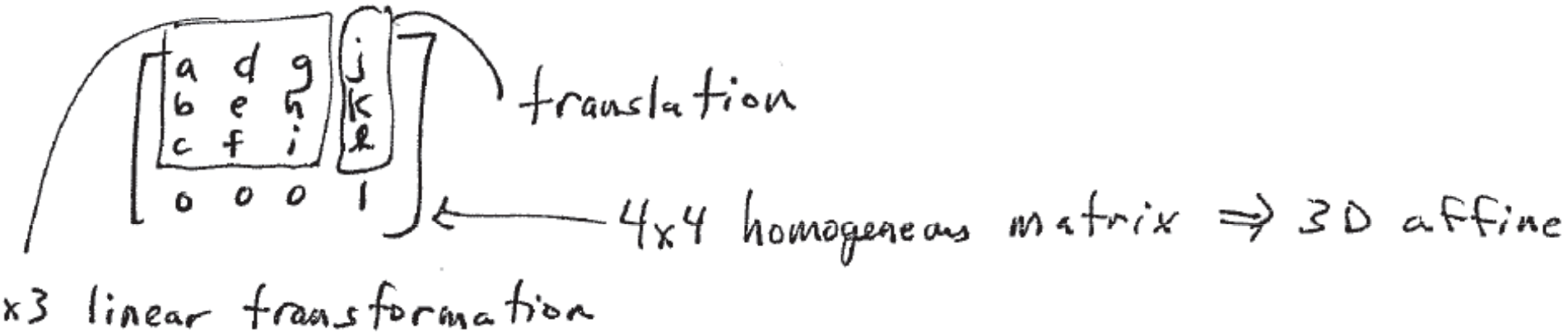
3x3 homogeneous matrix ⇒ 2D affine

linear part affects both

$$\begin{bmatrix} a & c \\ b & d \end{bmatrix}$$

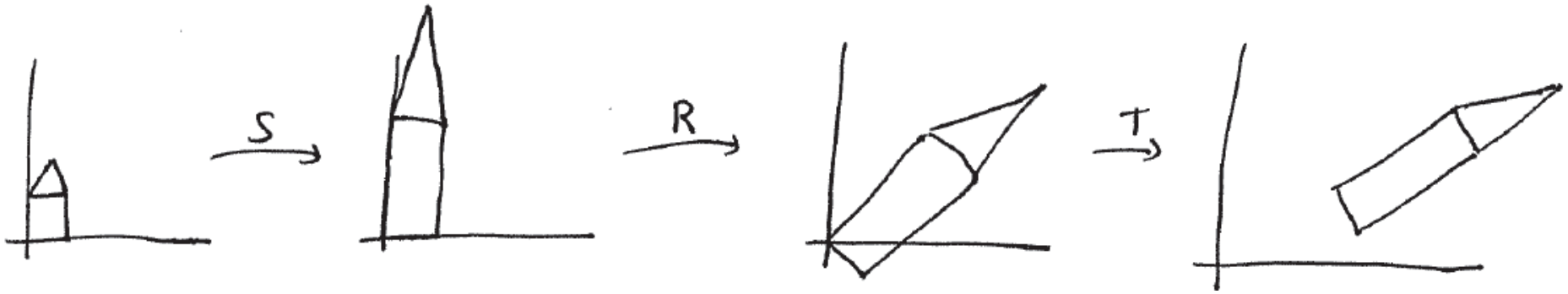
General 3D affine transformation

- 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()
glTranslatef(…)
glRotatef(…)
glScalef(…)
house()
```

I

IT

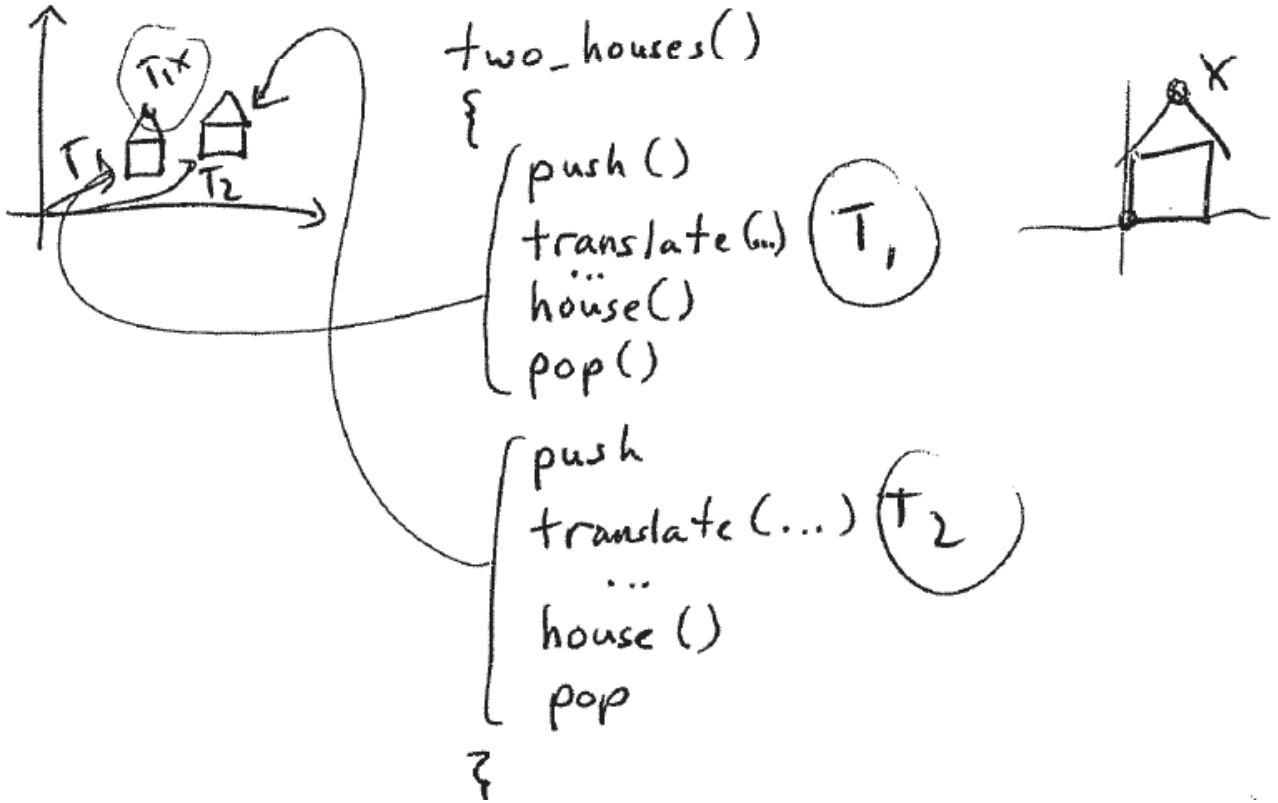
ITR

ITRS

- Seen ***backwards*** in code?

OpenGL transformation order

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

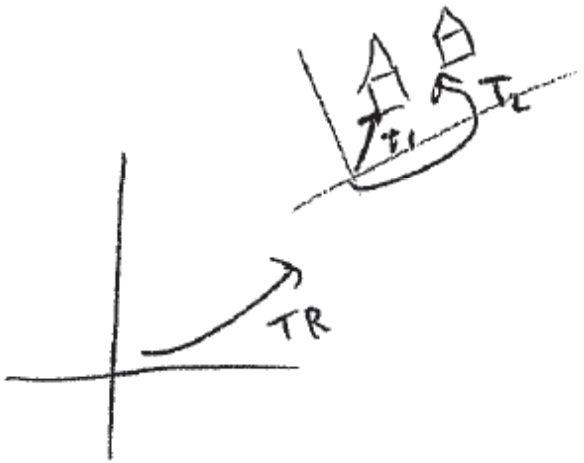


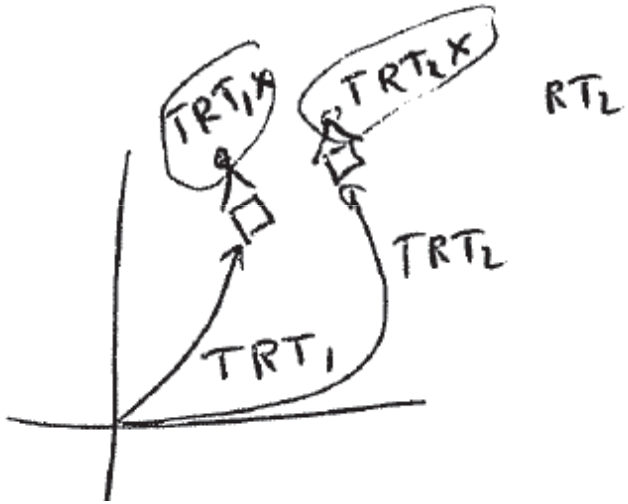
OpenGL transformation order

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

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

two_houses()

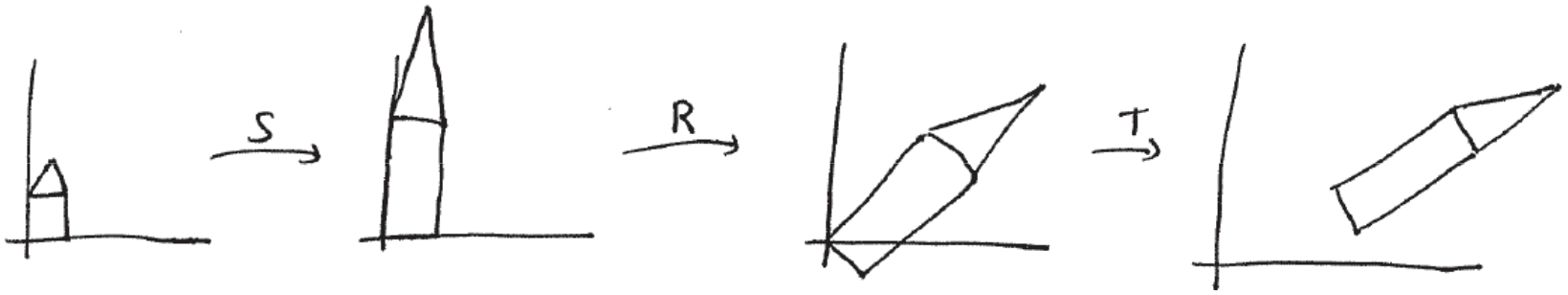




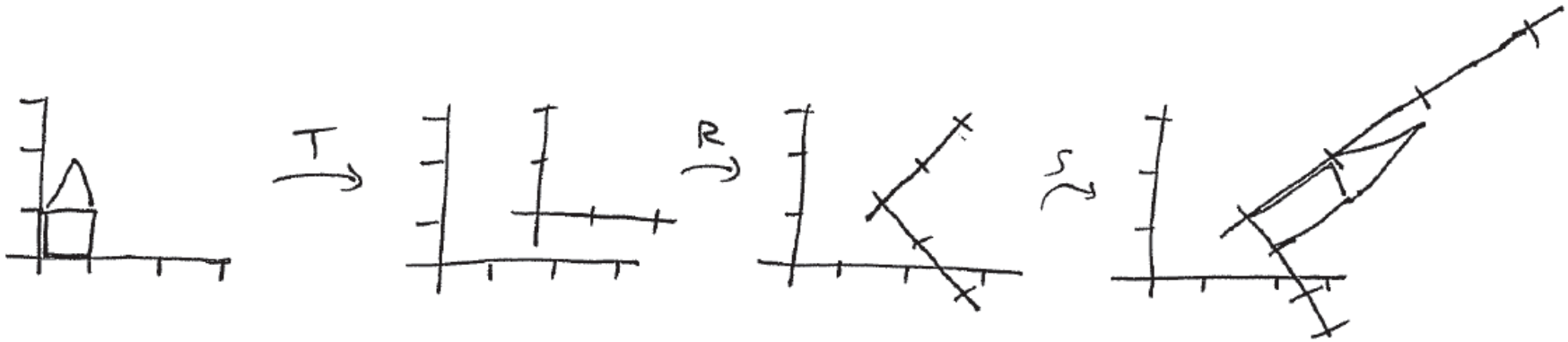
  - Can't do this with left multiplication without changing **`two_houses()`**
    - Would need to store intermediate results until done transforming

OpenGL transformation order

- Earlier picture thinks about a global, fixed coordinate system

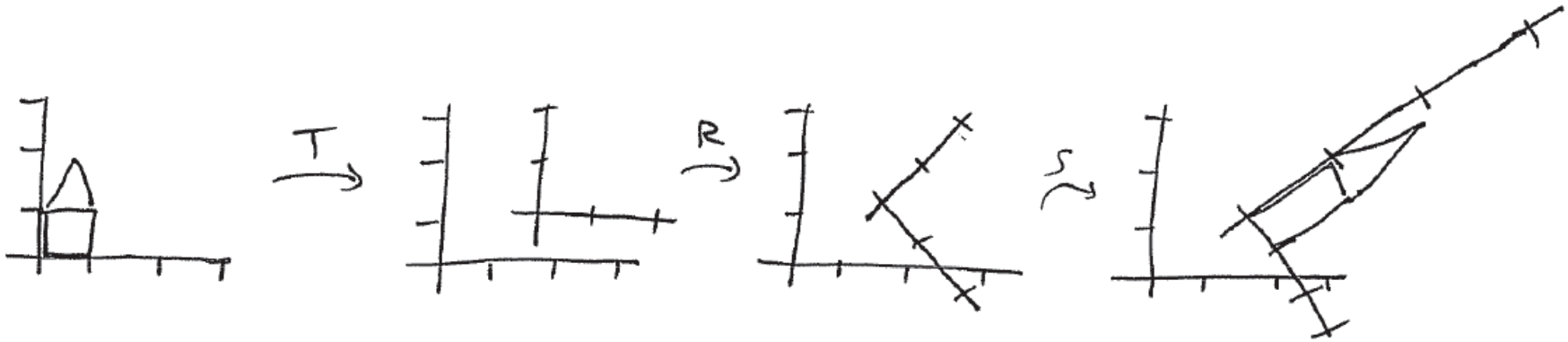


- What about a local, changing coord. system?



OpenGL transformation order

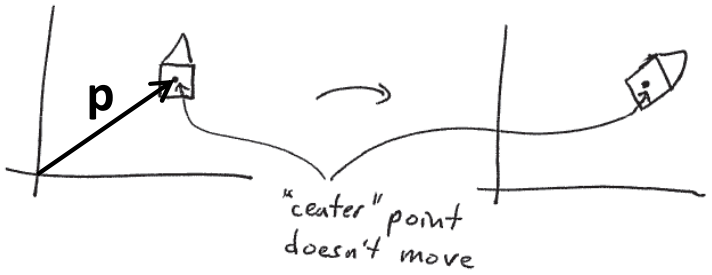
- What about a local, changing coord. system?



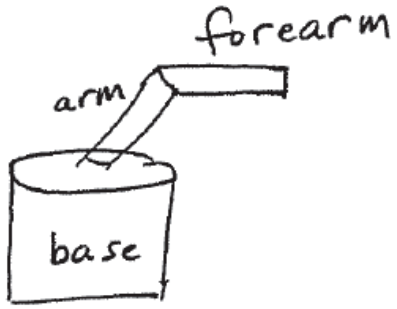
- Coordinate system (local) containing objects is transformed (**all** non-rigid transformations!)
- Same order as OpenGL

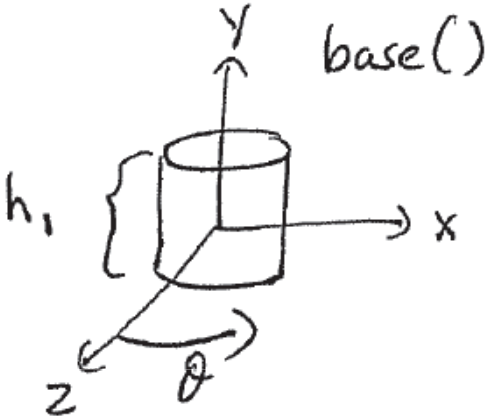
OpenGL transformation order

- Example: in place rotation
  - Translate center point **p** to origin
  - Rotate
  - Translate center point **p** back
  - **`glLoadIdentity()`**
    `glTranslatef(`p_x`, `p_y`, `p_z**`)`**
    `glRotatef(…)`
    `glTranslatef(-`p_x`,-`p_y`,-`p_z**`)`**
    `house()`

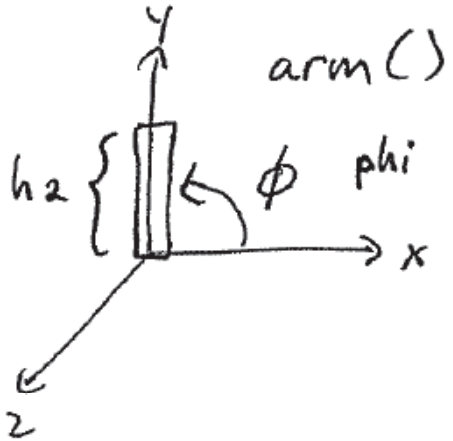


Simple robot example

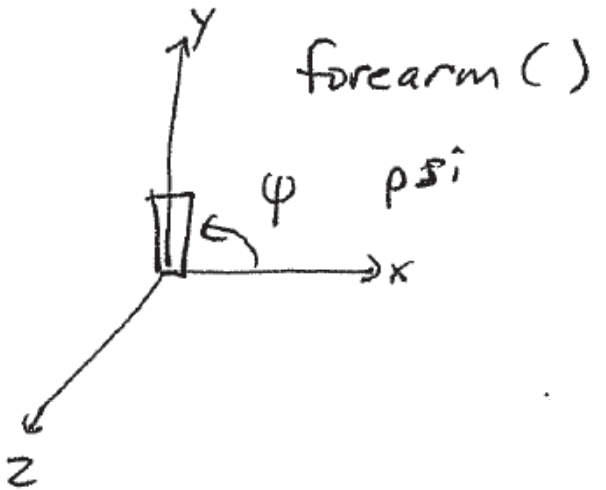




$R_y(\theta)$



$T(0, h_1, 0) R_z(\phi)$



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

- In code (think OpenGL **state**!)

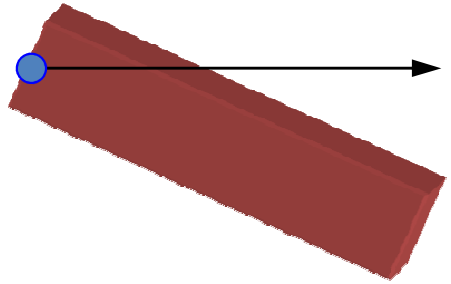
```
robot()
  glPushMatrix()
  glRotatef(θ, 0,1,0)
  base()
```

```
  glTranslatef(0, h1, 0)
  glRotatef(φ, 0,0,1)
  arm()
```

```
  glTranslatef(0, h2, 0)
  glRotatef(ψ, 0,0,1)
  forearm()
  glPopMatrix()
```

One more robot example

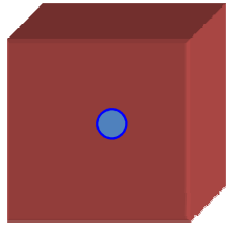
- `glRotatef(angle, 0.0, 0.0, 1.0)`



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

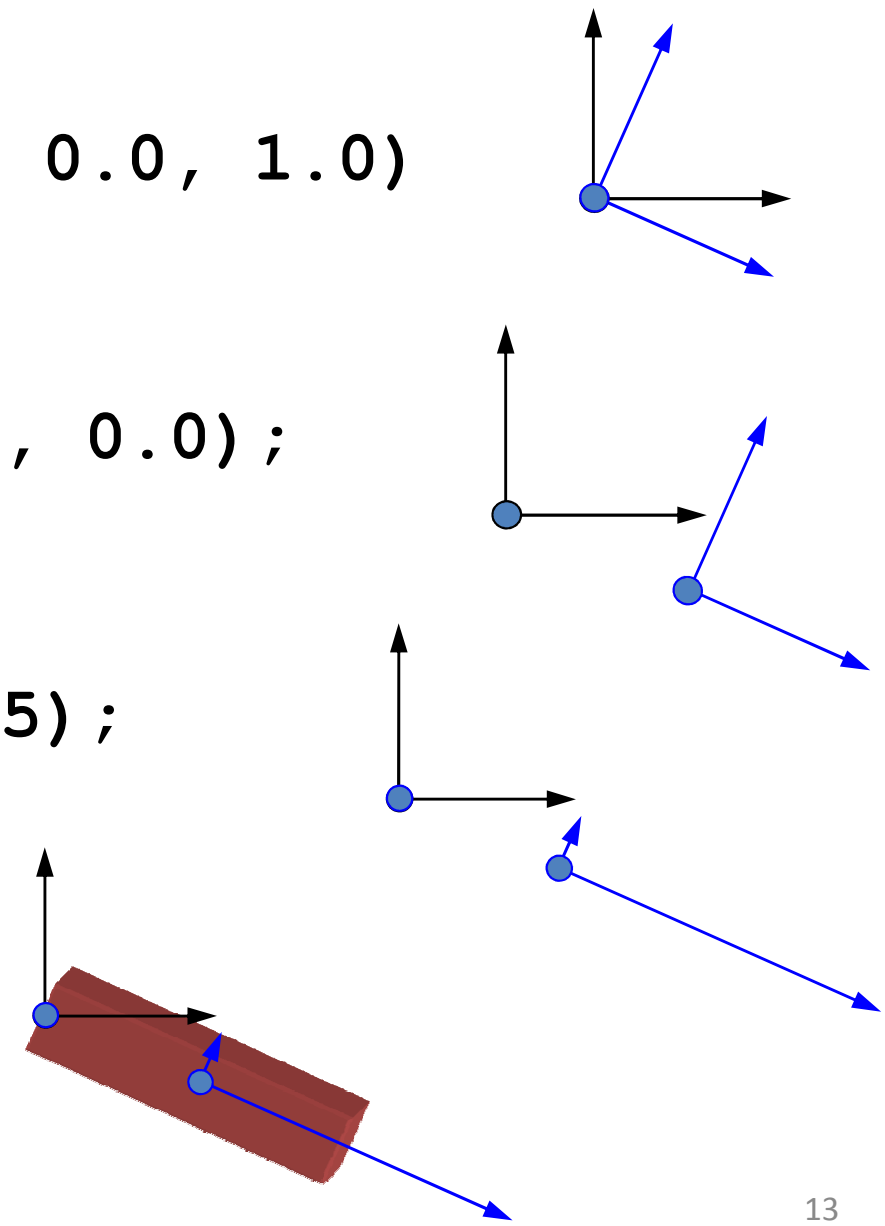
- `glScalef(2.0, 0.5, 0.5);`

- `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);`

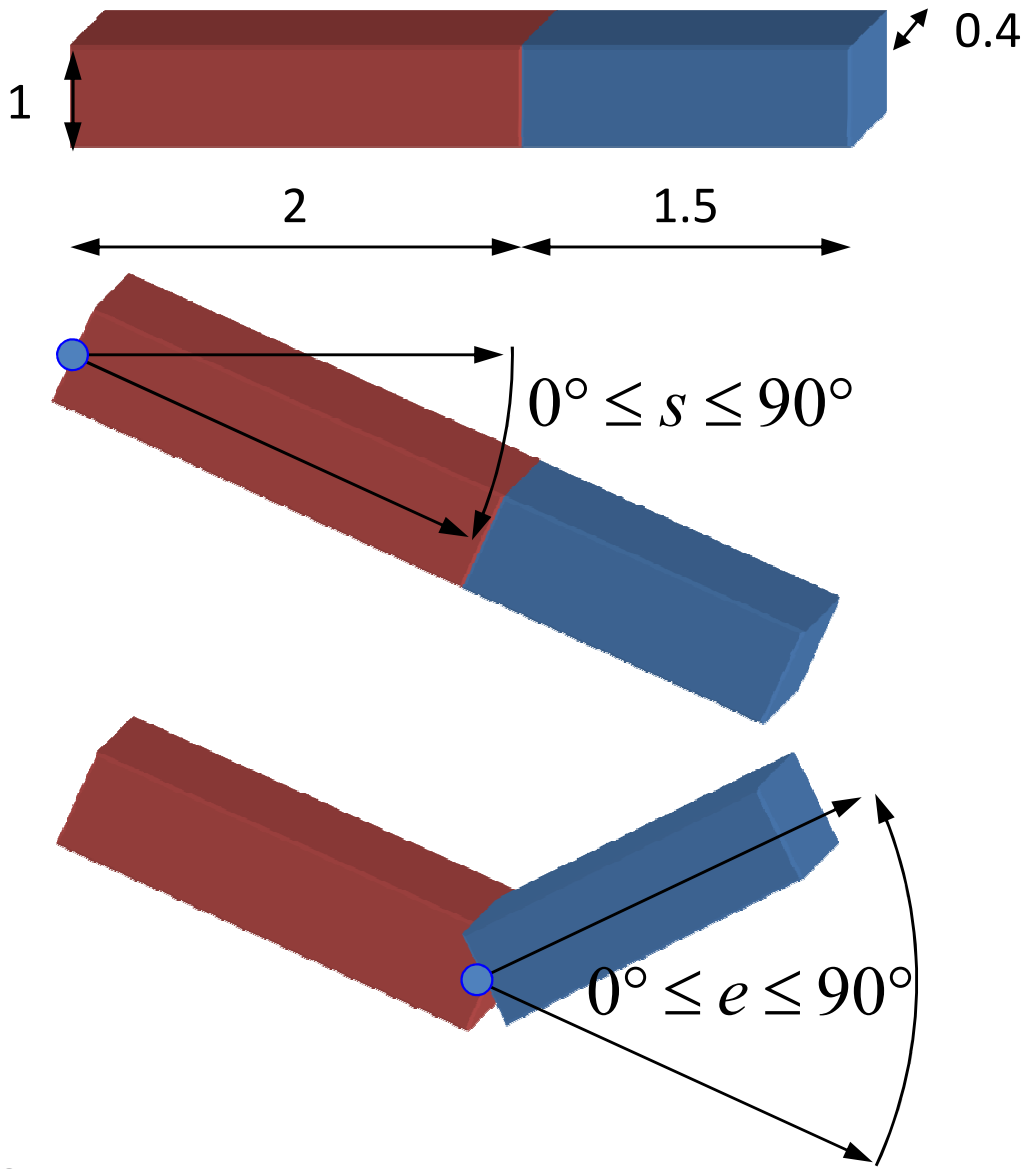


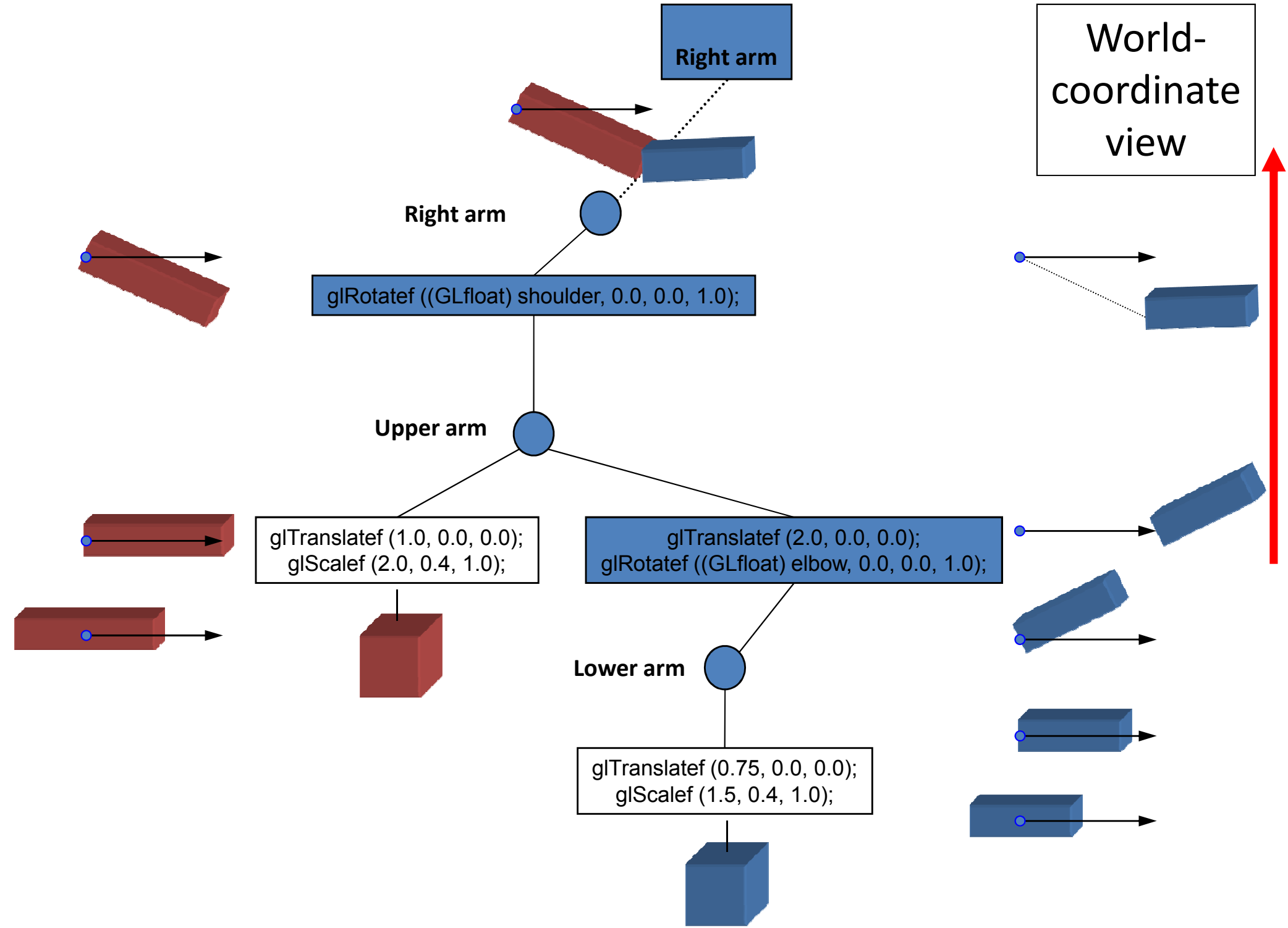
OpenGL matrix stack

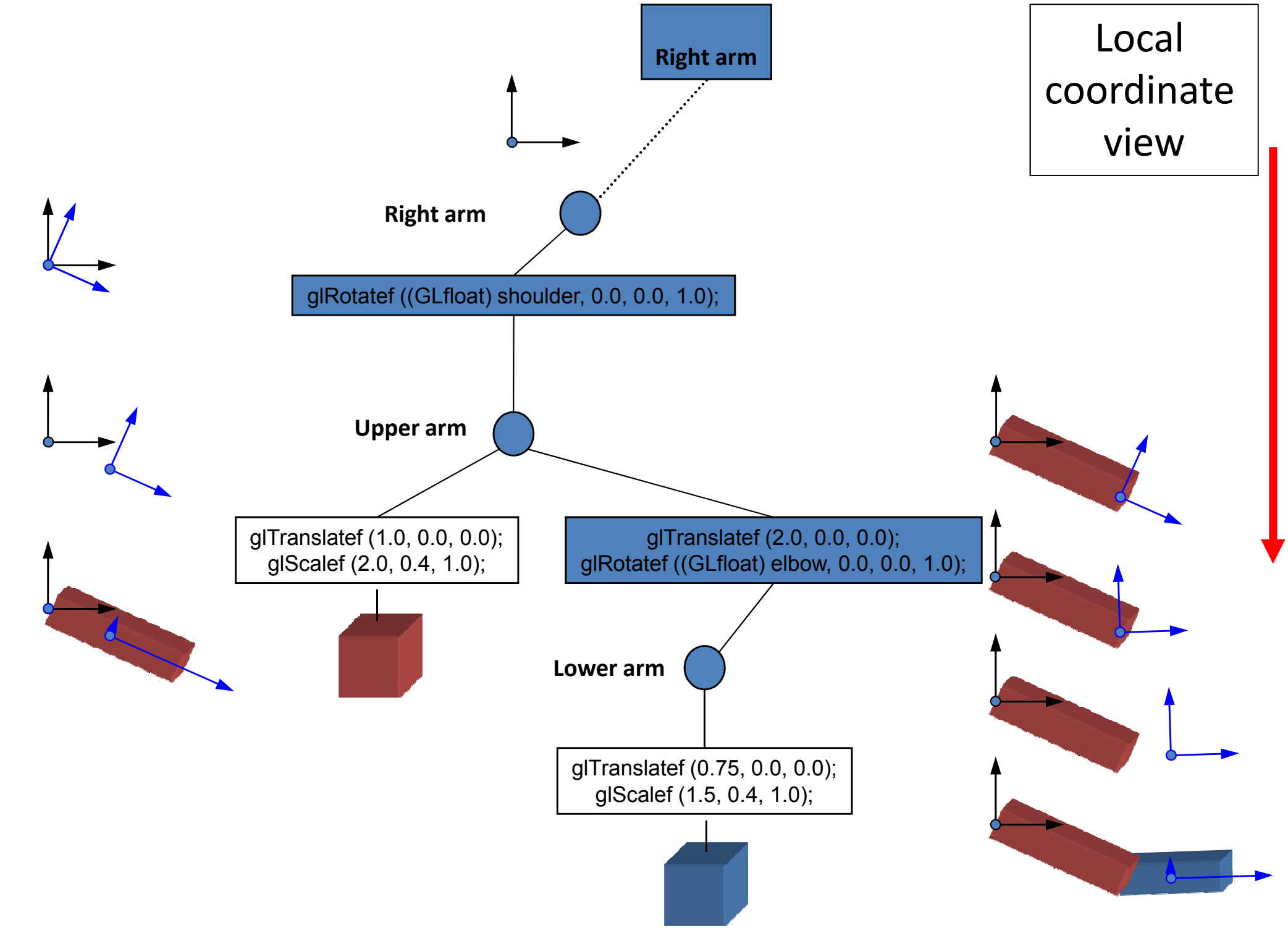
- The current matrix is on the top of the stack
- Intermediate storage needed for matrices
- Matrix stack
  - **`glPushMatrix();`**
    - Copies the top matrix (= stores current matrix)
  - **`glPopMatrix();`**
    - Removes the top matrix (= use old matrix)
- Push and pop bracket transformations
- OpenGL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

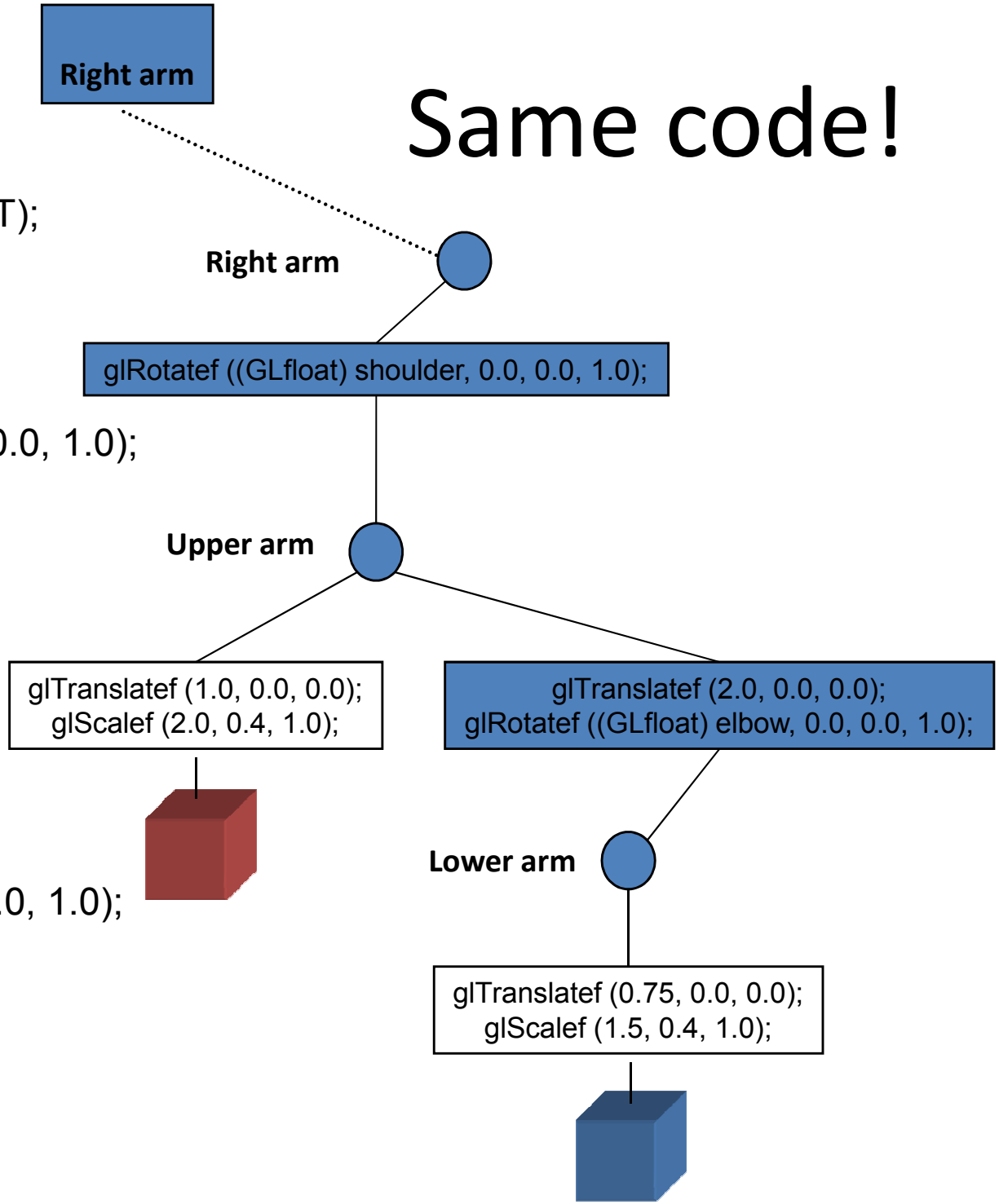



Right arm
World-coordinate view
Right arm
glRotatef ((GLfloat) shoulder, 0.0, 0.0, 1.0);
Upper arm
glTranslatef (1.0, 0.0, 0.0);
glScalef (2.0, 0.4, 1.0);
glTranslatef (2.0, 0.0, 0.0);
glRotatef ((GLfloat) elbow, 0.0, 0.0, 1.0);
Lower arm
glTranslatef (0.75, 0.0, 0.0);
glScalef (1.5, 0.4, 1.0);

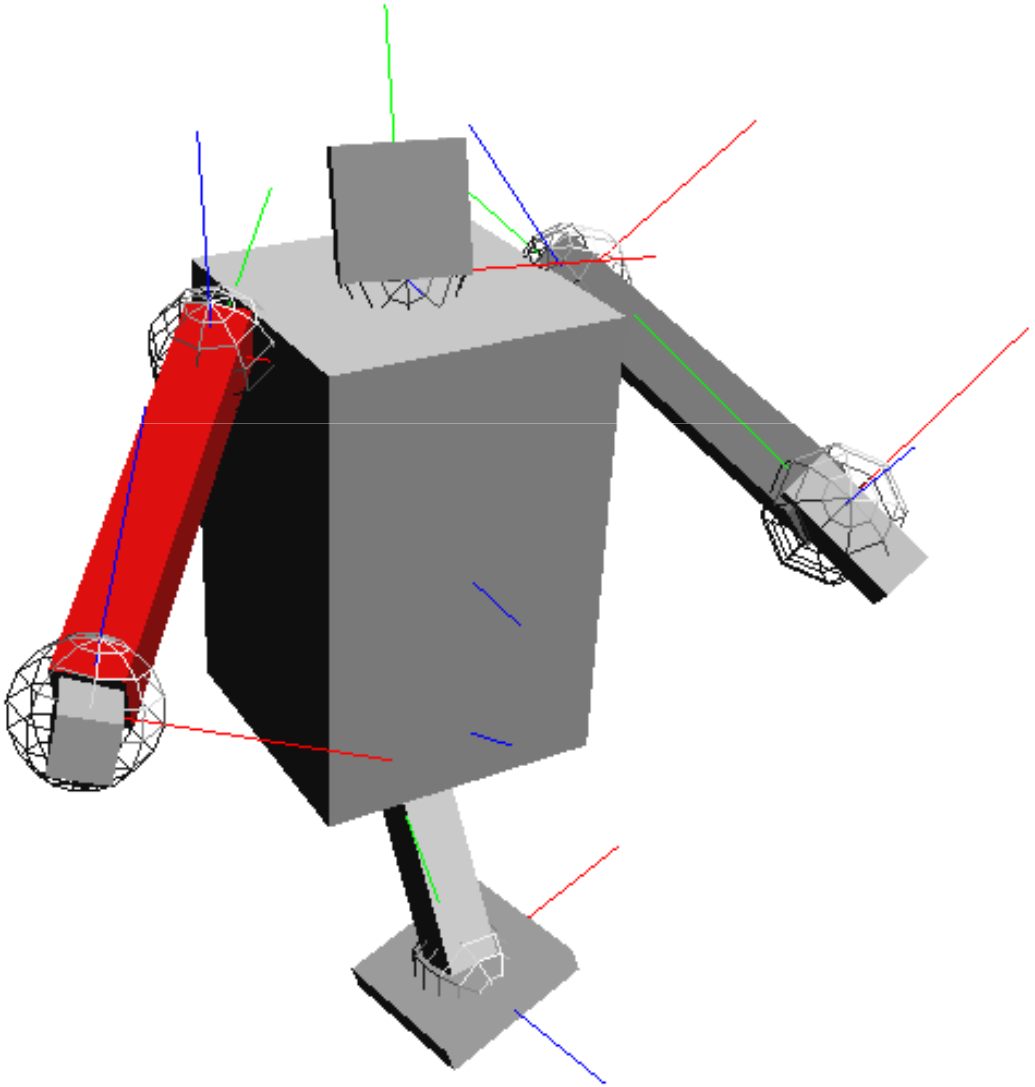

Right arm
Local coordinate view
Right arm
glRotatef ((GLfloat) shoulder, 0.0, 0.0, 1.0);
Upper arm
glTranslatef (1.0, 0.0, 0.0);
glScalef (2.0, 0.4, 1.0);
glTranslatef (2.0, 0.0, 0.0);
glRotatef ((GLfloat) elbow, 0.0, 0.0, 1.0);
Lower arm
glTranslatef (0.75, 0.0, 0.0);
glScalef (1.5, 0.4, 1.0);

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();

glPushMatrix();
   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);
glPopMatrix();

glFlush();
```



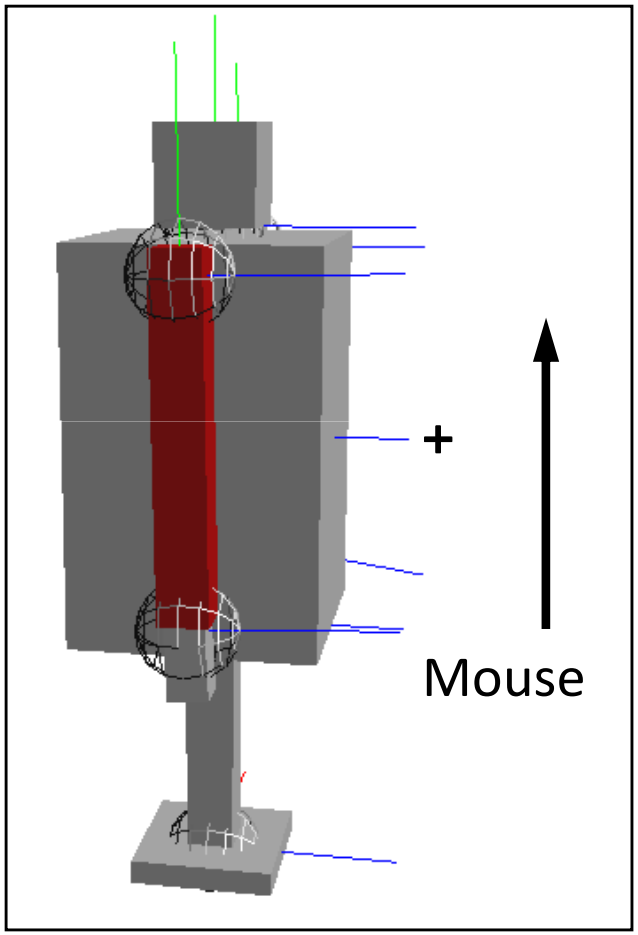
And yet another robot

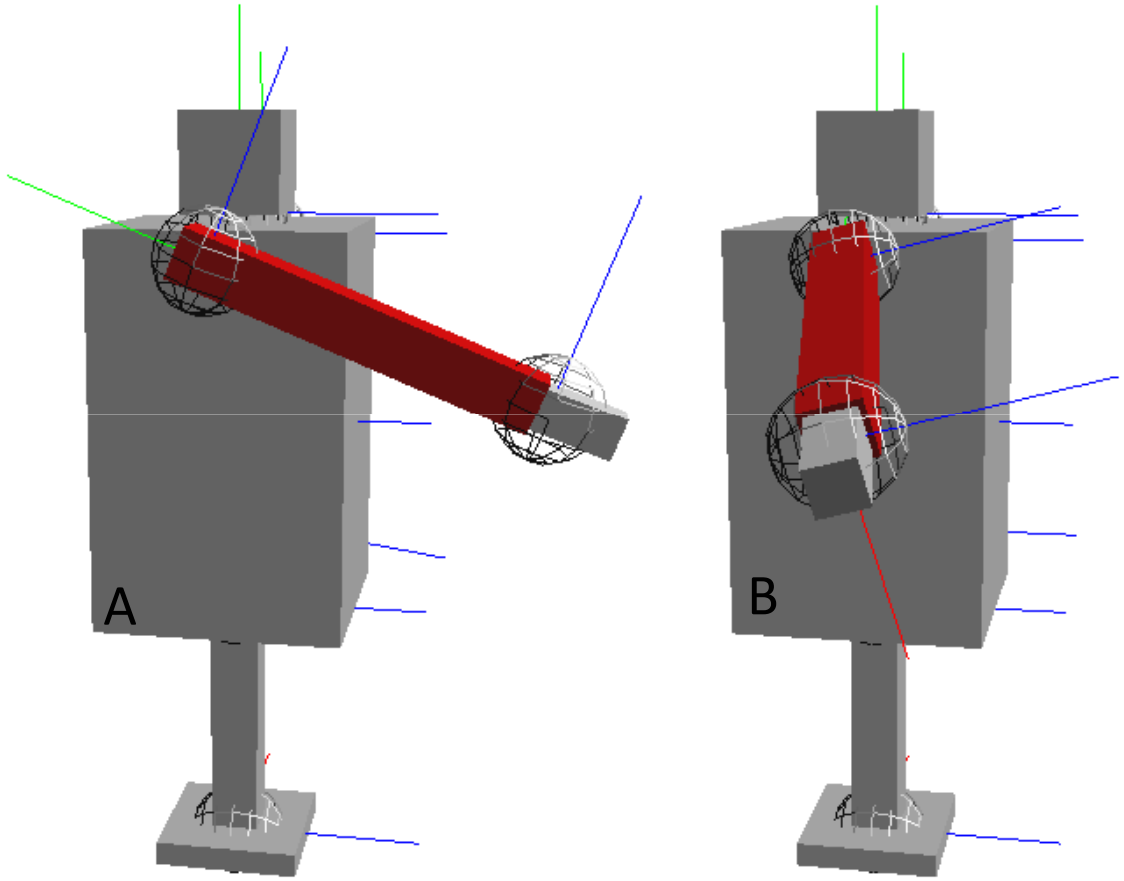


- Euler angles
  - α_x, α_y, α_z per node
  - Gimbal lock singularity
- User interface problem
  - Rotations are performed in local coordinates (demo)
  - These are independent of the viewing position
  - Can be quite unintuitive

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());
```

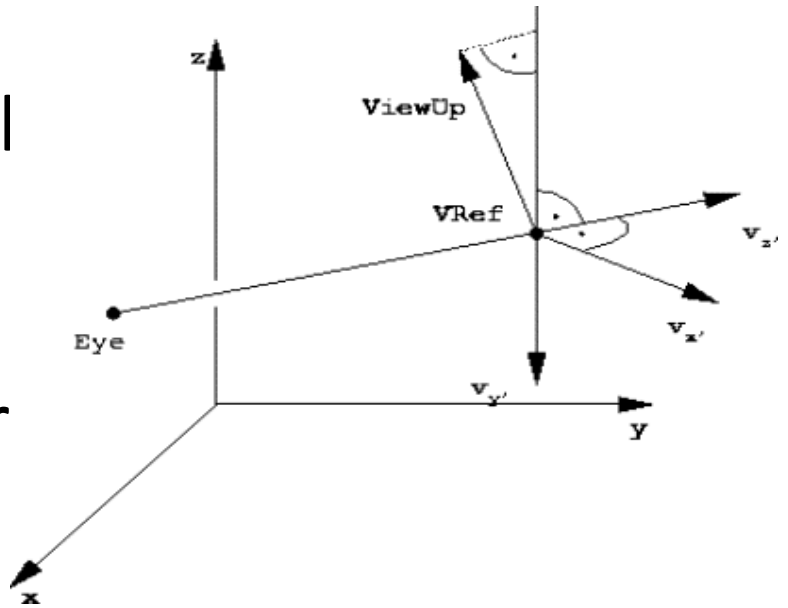
Instead of

```
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);
```

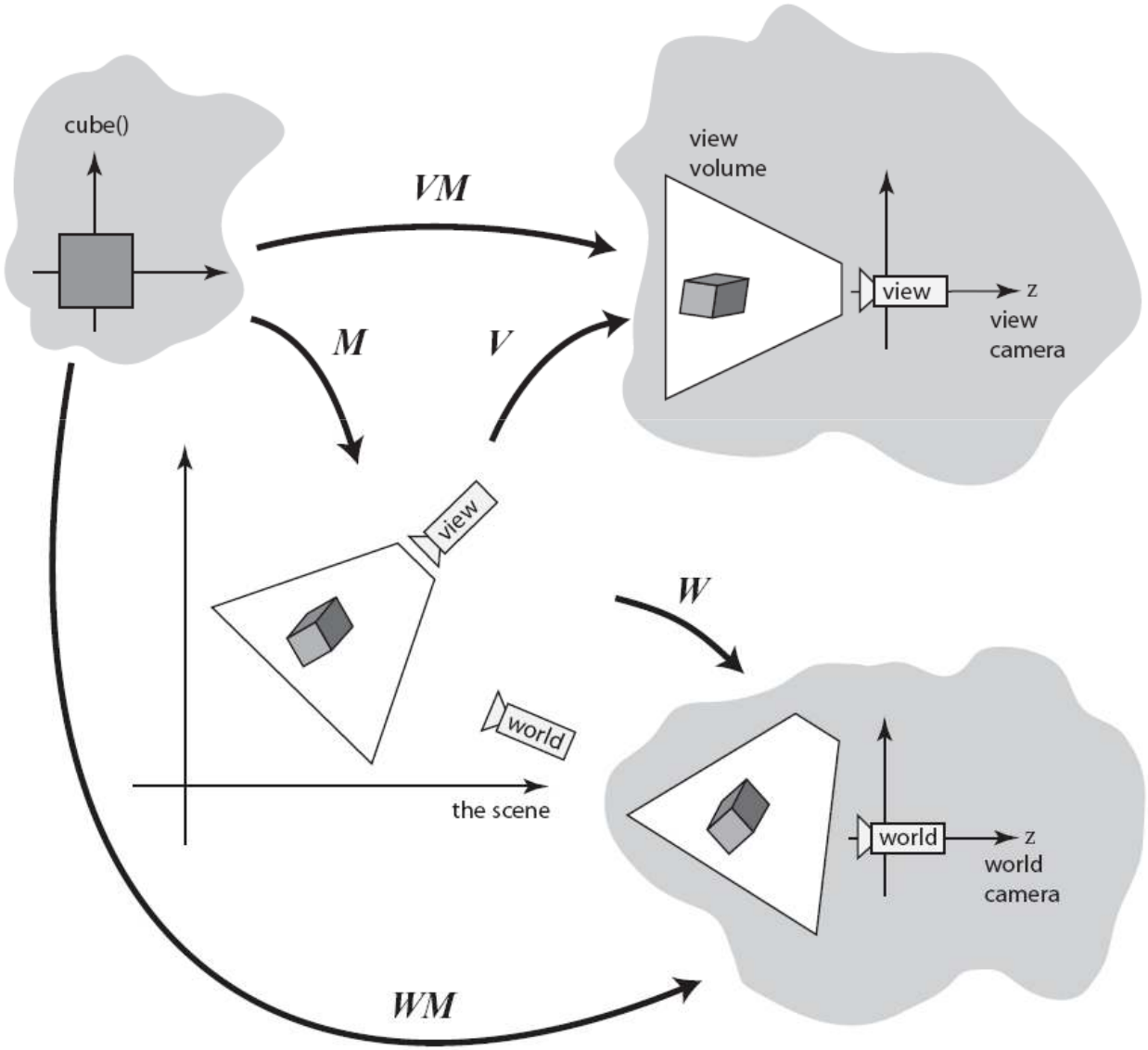
    - In this implementation $\mathbf{M}$ only stores the rotation, since the center of rotation is usually shifted

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)



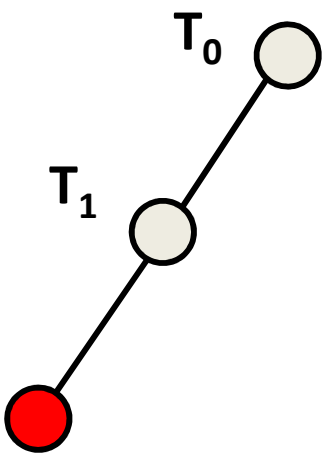
Modelview transformation



Affine Transformations

Relative to the viewer

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



$T_2=R_xR_y$

User-defined transformation

$$v' = T_0T_1\,v$$

$$v = (T_0T_1)^{-1}\,v'$$

v‘ in global coordinates

v in local coordinates

Previously:

$$v' = T_0T_1\,T_2\,v = T_0T_1\,R_xR_y\,v$$

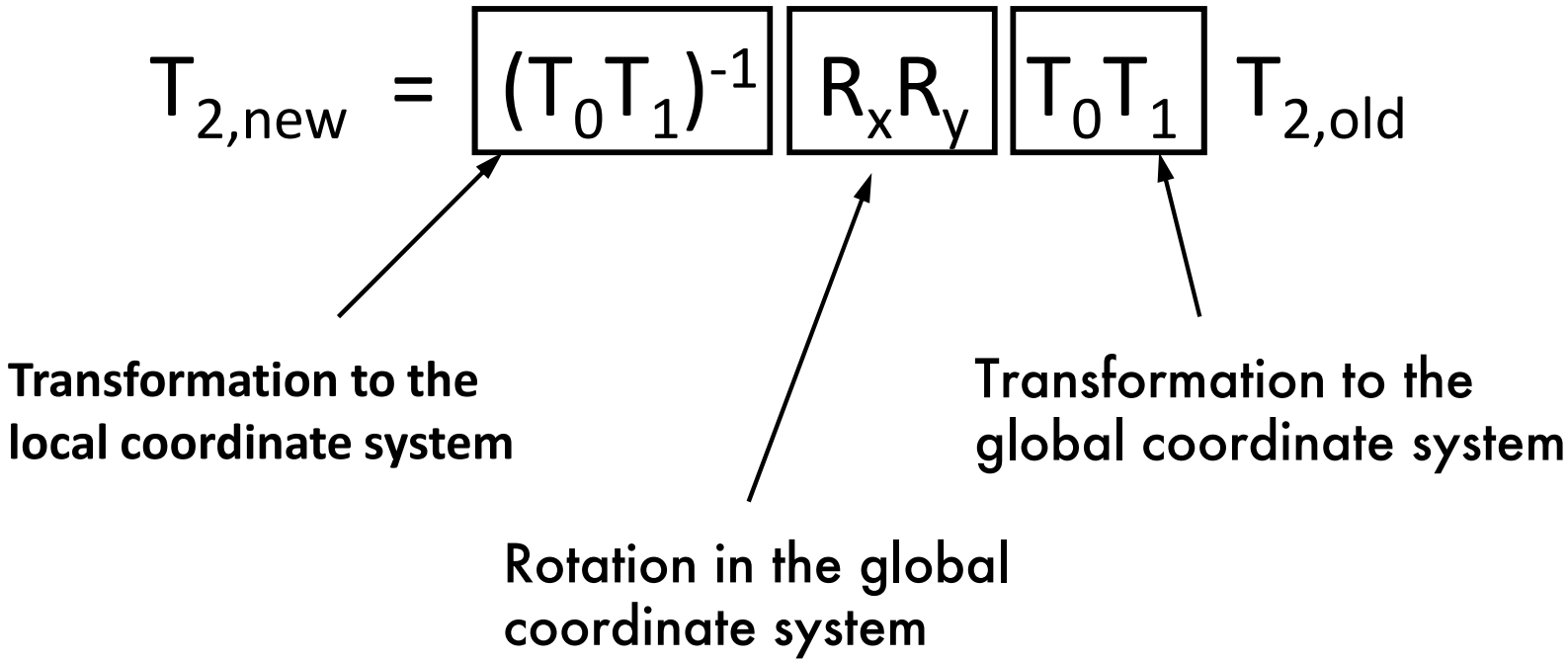
To rotate in global coordinates:

$$T_{2,new} = (T_0T_1)^{-1}\,R_xR_y\,T_0T_1\,T_{2,old}$$

Affine Transformations

Relative to the viewer

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



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.)

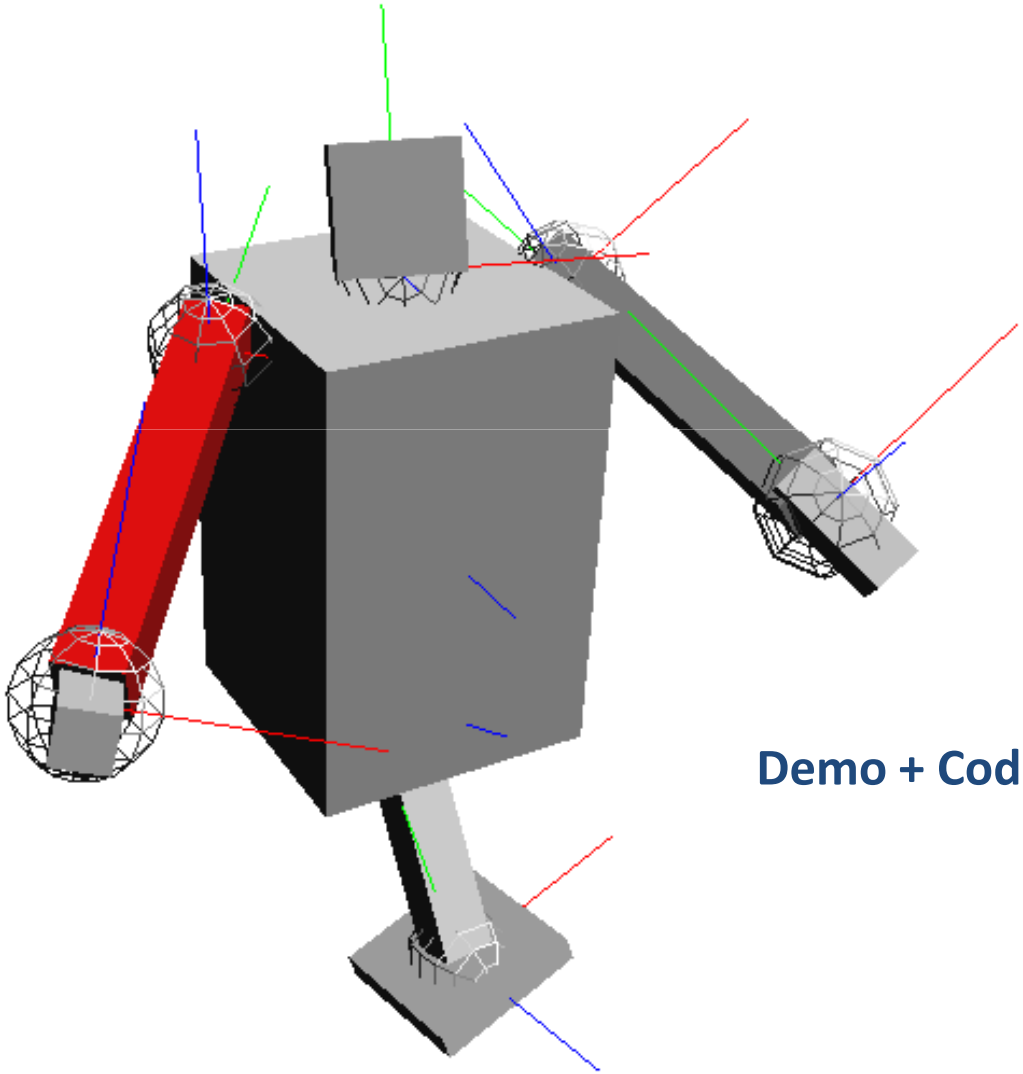
Unit canonical vectors in local coordinates

$$lx = (T_0 T_1)^{-1} e_x$$
$$ly = (T_0 T_1)^{-1} e_y$$

$$T_{2,new} = R_{lx} R_{ly} T_{2,old}$$

Affine Transformations

Relative to the viewer



Demo + Code