Transformations

Outline

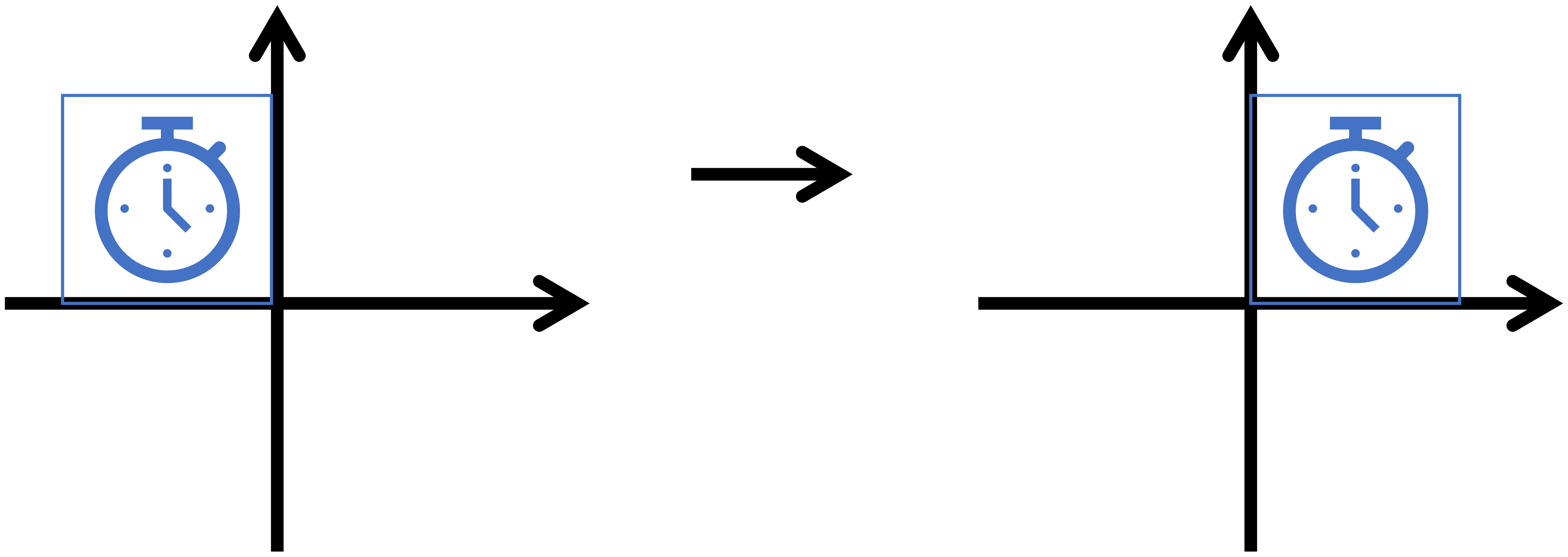
1. 2D transformation
    1. Basic transformation
    2. Homogeneous coordinates

2. 3D transformation
    1. 3D rotation
    2. Orthographic and perspective transformation
    3. Viewport transformation
    4. Transformation in OpenGL

3. Transformation and kinematics
    1. Forward kinematics
    2. Inverse kinematics (Until Later...)

2D transformation

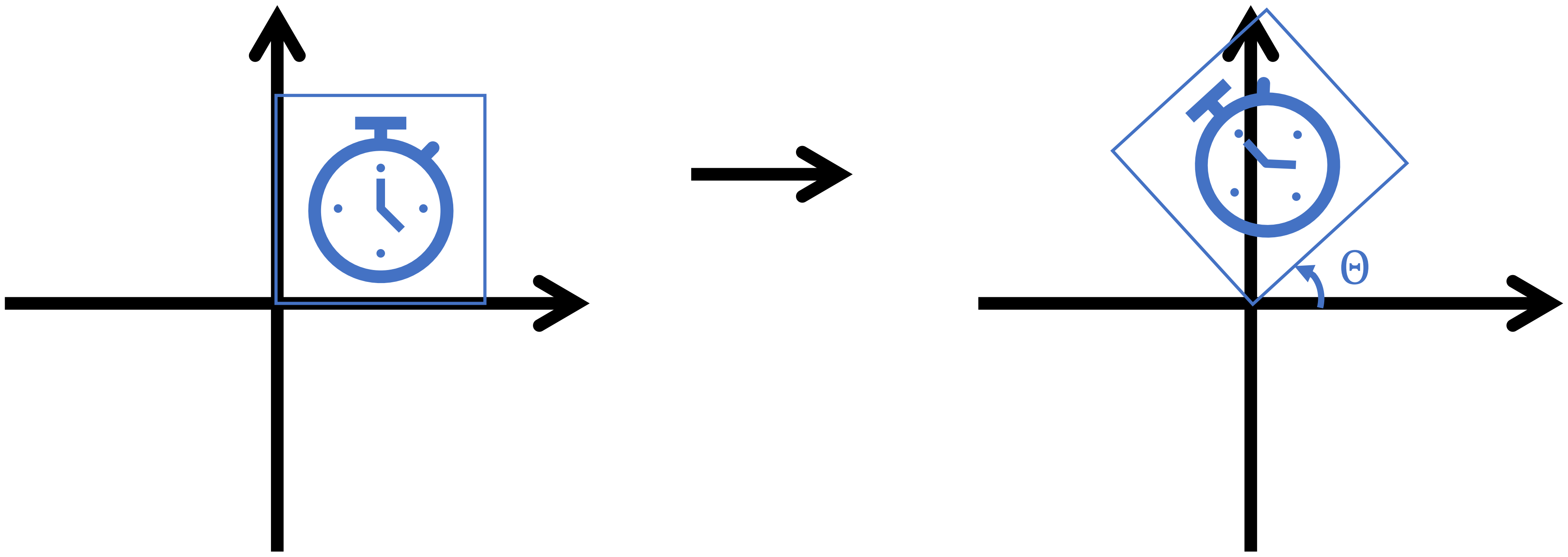
translation



$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} t_x \\ t_y \end{bmatrix} + \begin{bmatrix} x \\ y \end{bmatrix}$$

2D transformation

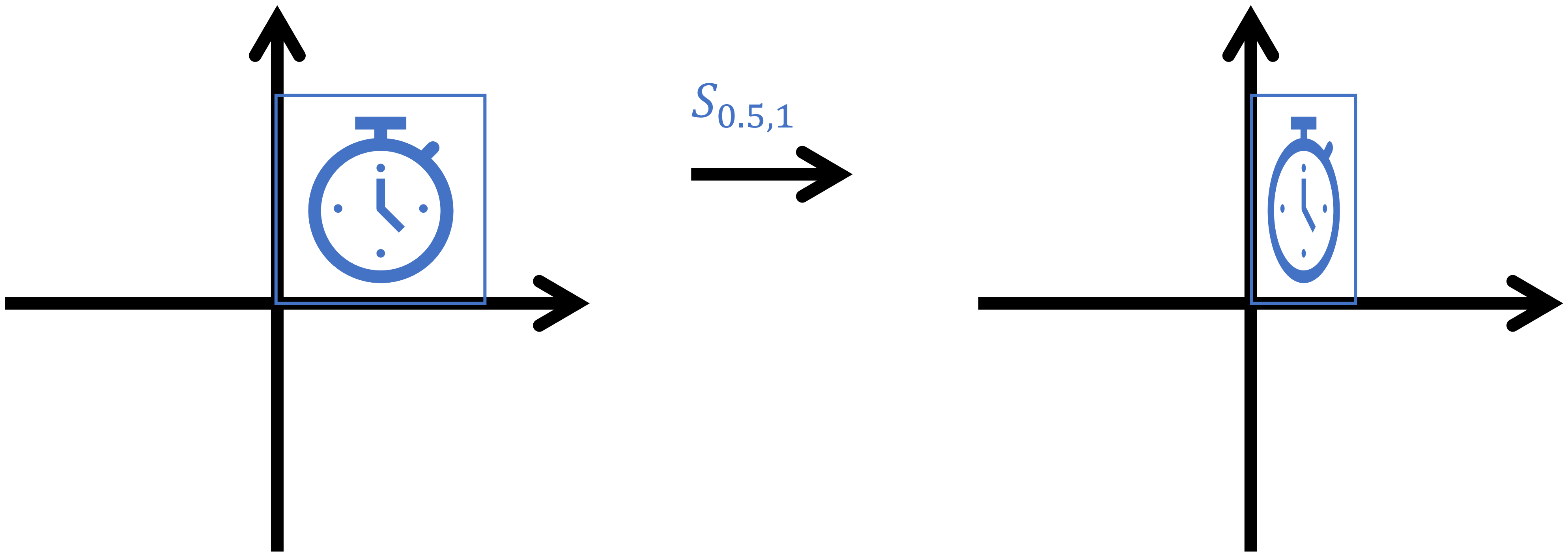
rotation



$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

2D transformation

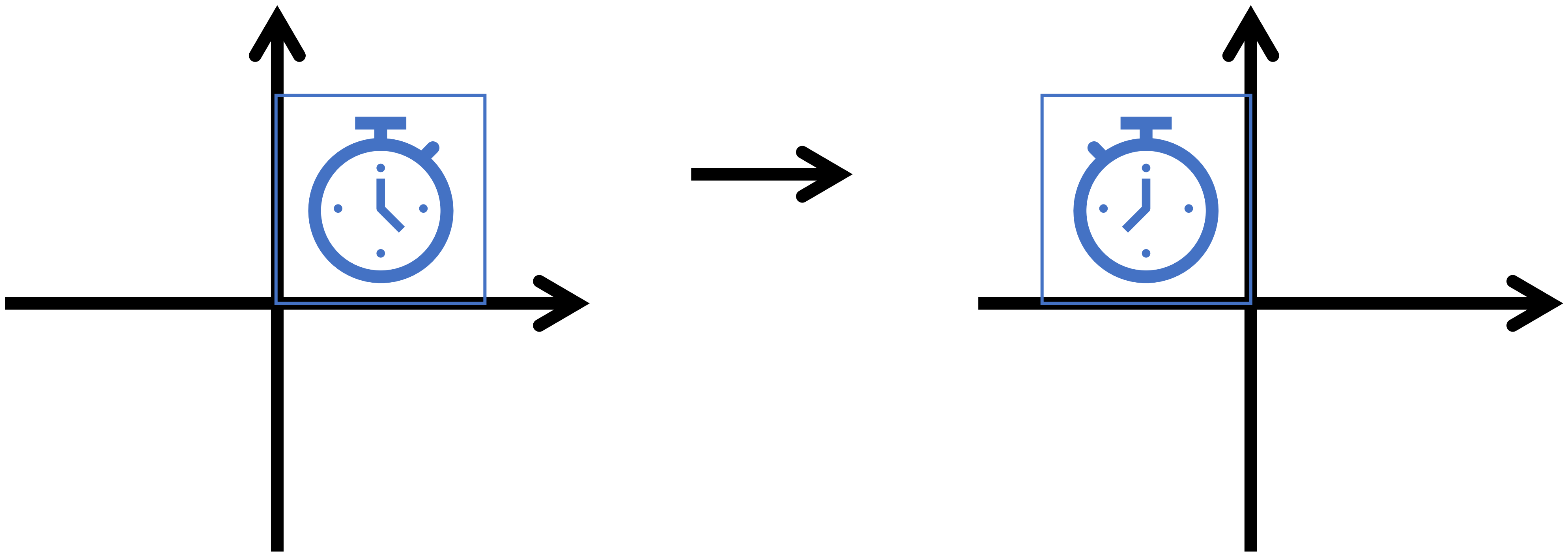
scale



$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} s_x & 0 \\ 0 & \mathrm{s_y} \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

2D transformation

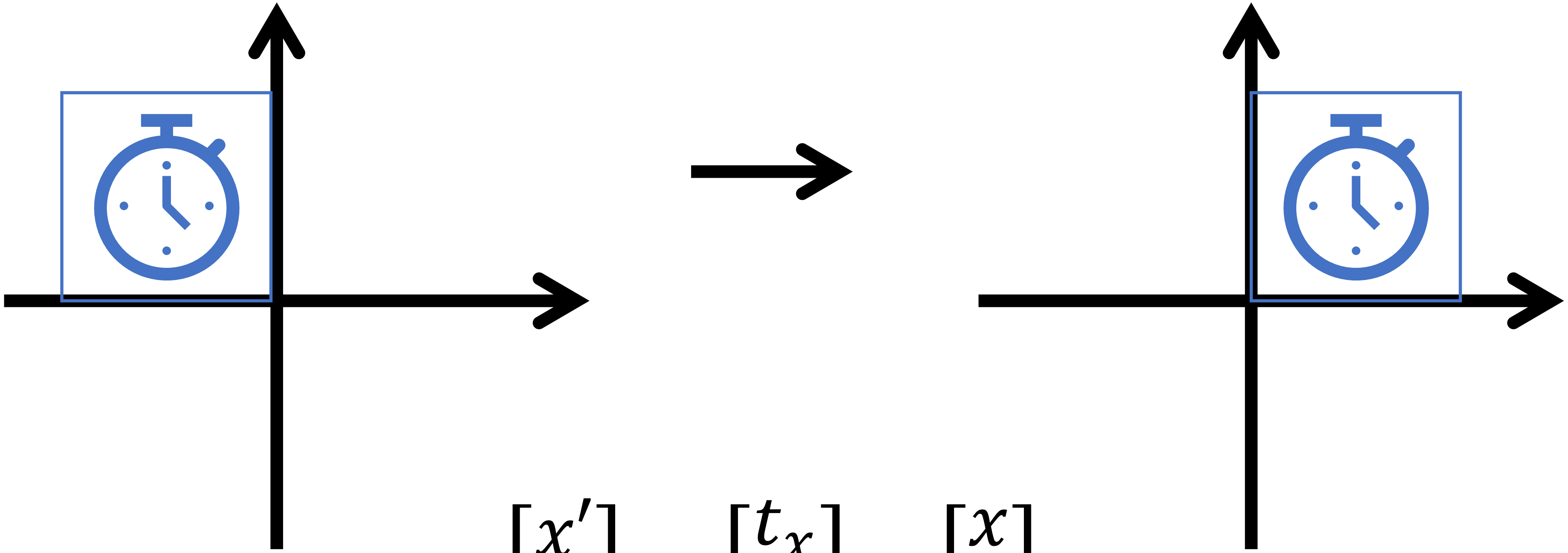
reflection



$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

Question: matrix for translation?

translation



$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} t_x \\ t_y \end{bmatrix} + \begin{bmatrix} x \\ y \end{bmatrix}$$

N-D Translation is not a linear transformation in N-D.

linear transformation: T(u+v) = T(u) + T(v); T(c*u)=cT(u)

We can turn it into a linear transformation using a higher dimension

2D transformation

homogeneous coordinates: adding one dimension [x y w]

translation

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

scale

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

rotation

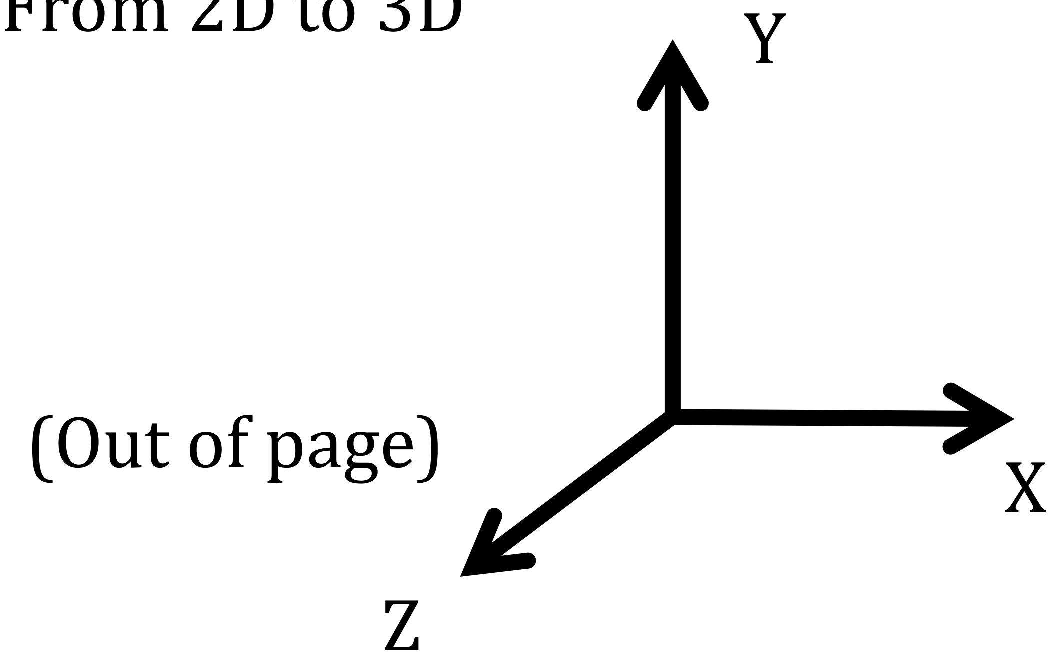
$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

reflection

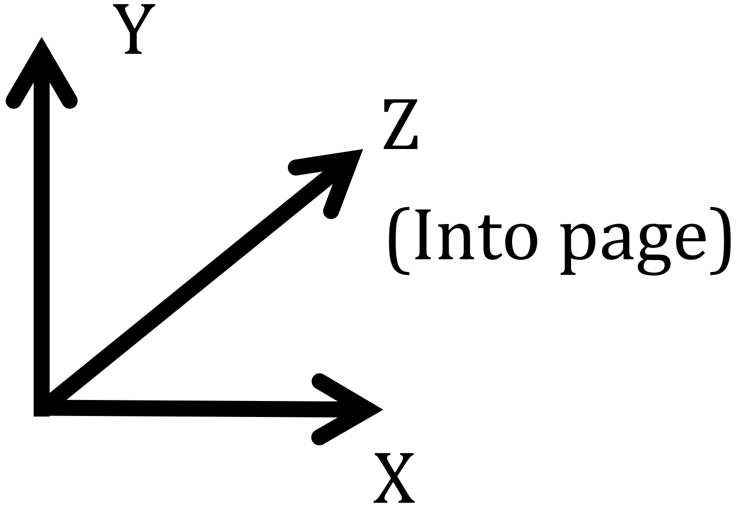
$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

3D transformation

From 2D to 3D



Right Hand



Left Hand

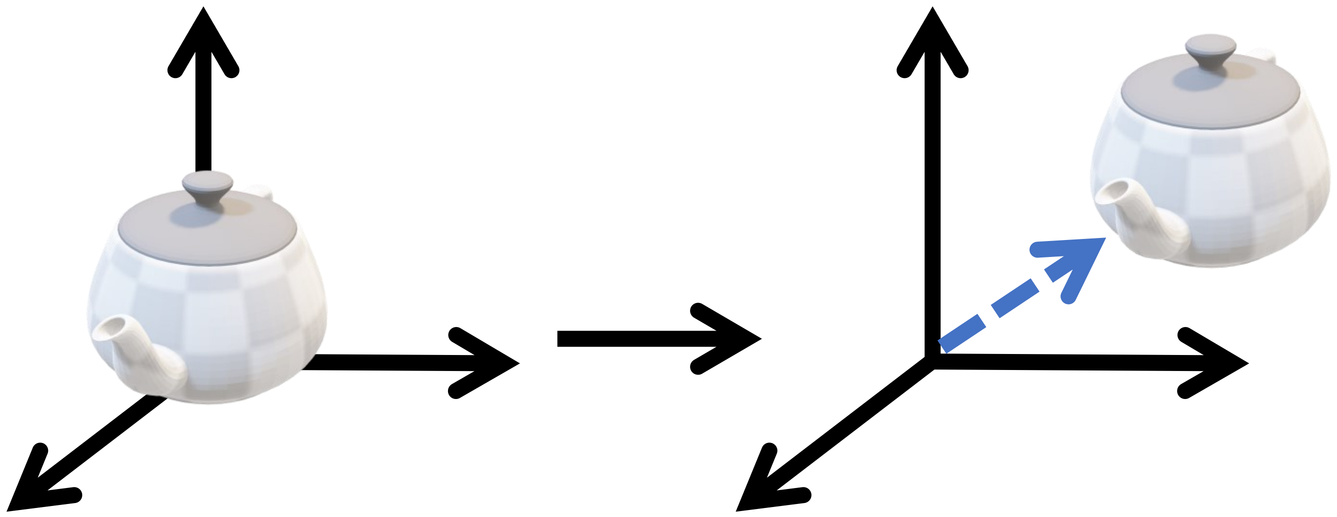
Z-axis determined from X and Y by cross product: $Z = X \times Y$

3D transformation

From 2D to 3D

3D translation

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ \mathrm{z} \\ 1 \end{bmatrix}$$

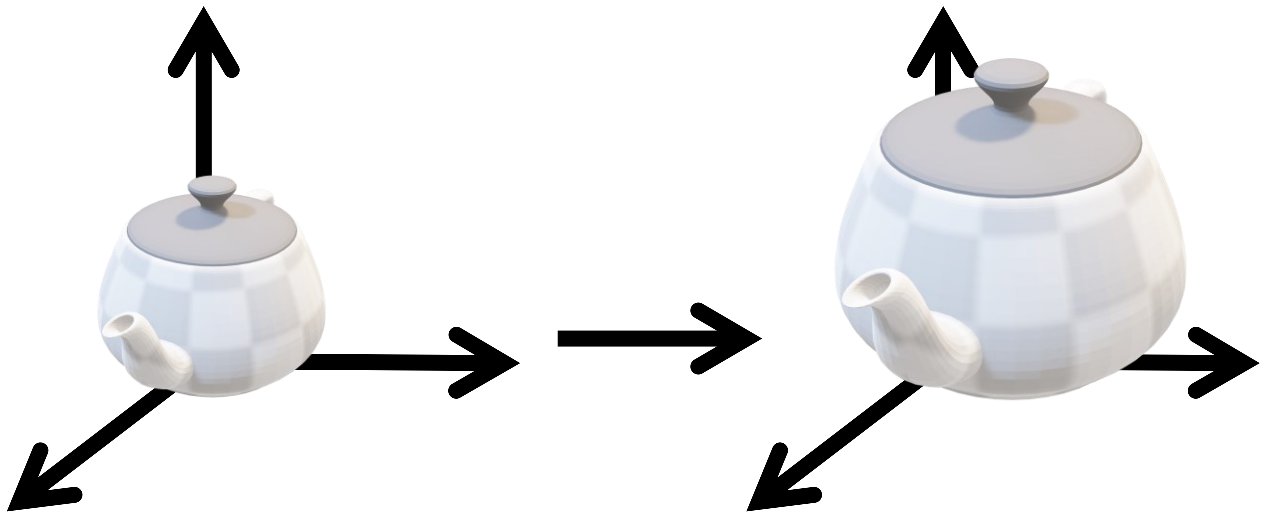


3D transformation

From 2D to 3D

3D scale

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ \mathrm{z} \\ 1 \end{bmatrix}$$

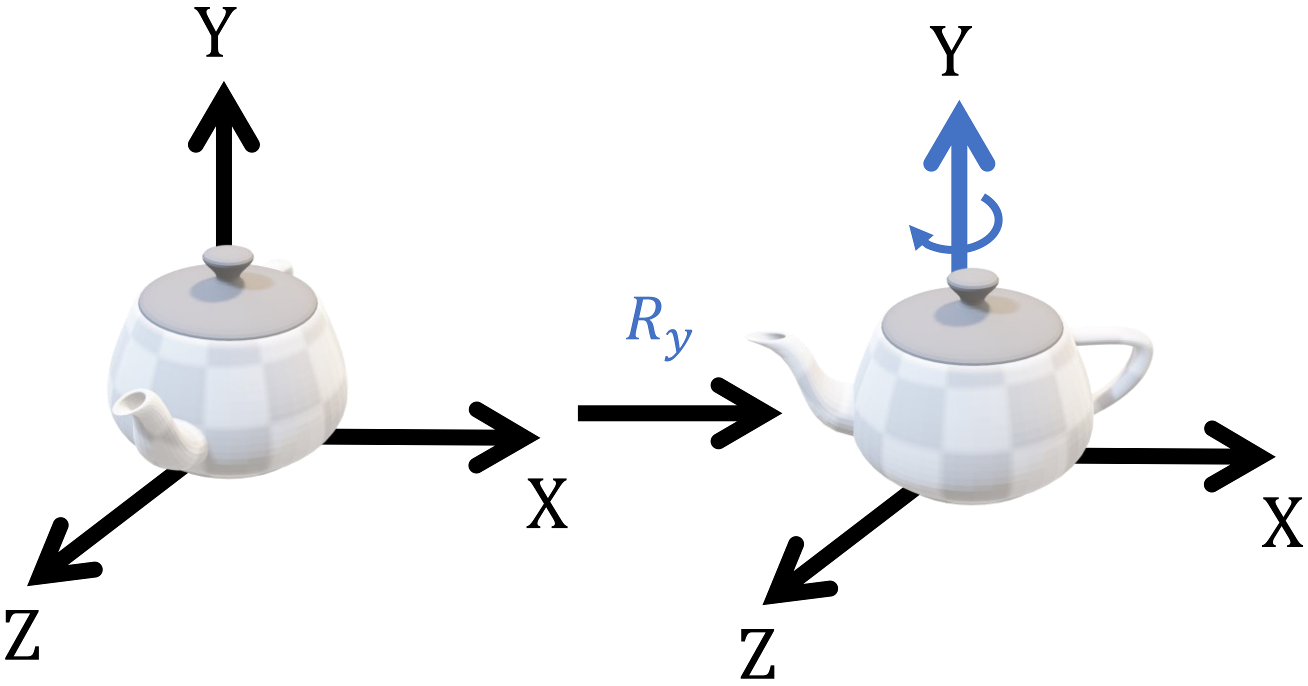


3D transformation

From 2D to 3D

3D rotation

$$R_x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) & 0 \\ 0 & \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



$$R_y = \begin{bmatrix} \cos(\theta) & 0 & \sin(\theta) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R_z = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 & 0 \\ \sin(\theta) & \cos(\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

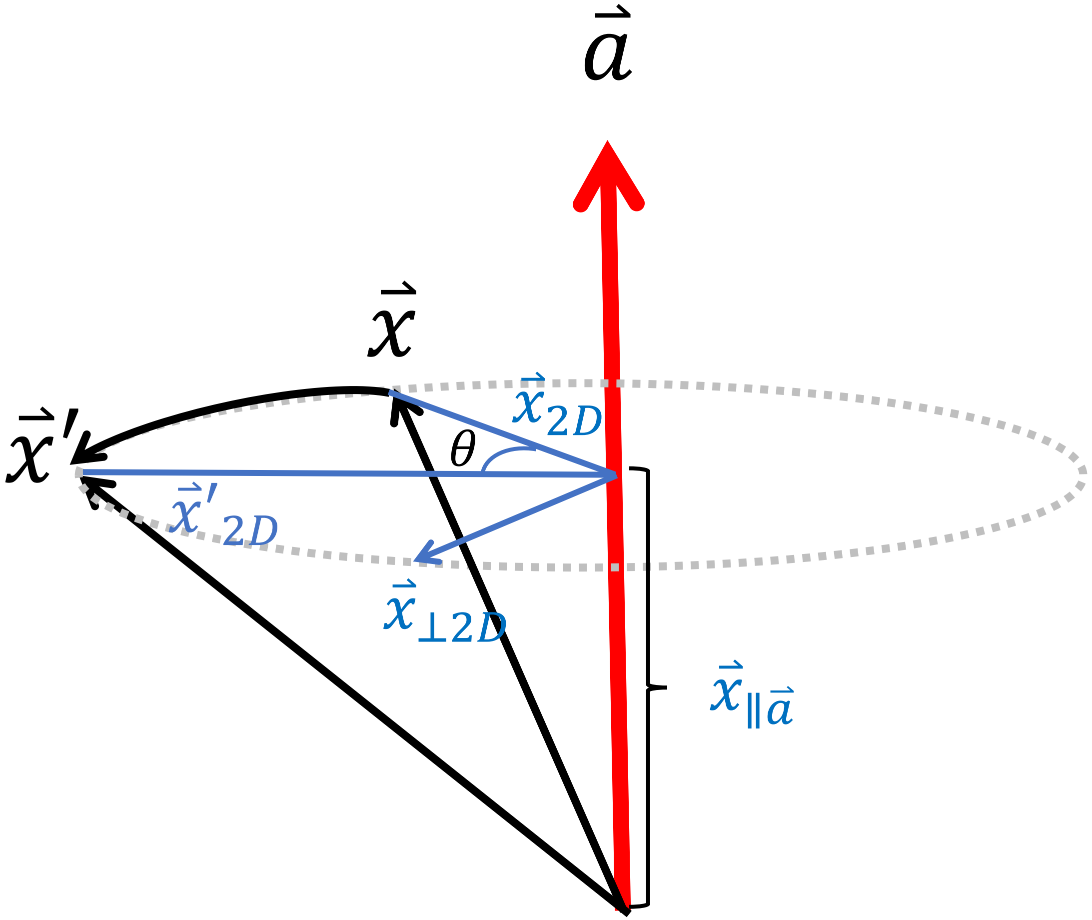
3D transformation

Axis-angle rotation

$$\vec{x}' = \vec{x}'_{2D} + \vec{x}_{\parallel\vec{a}}$$

$$\vec{x}' = [\overbrace{(\vec{x} - (\vec{x} \cdot \vec{a})\vec{a})}^{\vec{x}_{2D}} \cos\theta] + \overbrace{(\vec{a} \times \vec{x})}^{\vec{x}_{\perp 2D}} \sin\theta + \overbrace{(\vec{x} \cdot \vec{a})\vec{a}}^{\vec{x}_{\parallel\vec{a}}}$$

$$= \cos\theta\, \vec{x} + (1 - \cos\theta)(\vec{x} \cdot \vec{a})\vec{a} + \sin\theta\, (\vec{a} \times \vec{x})$$



Rodrigues' rotation formula

3D transformation

Axis-angle rotation in matrix representation

$$\vec{x}' = \cos\theta\,\vec{x} + (1-\cos\theta)(\vec{x}\cdot\vec{a})\vec{a} + \sin\theta\,(\vec{a}\times\vec{x})$$

- Cross-product as a matrix multiply: $\vec{a}^*\,\vec{x} = \vec{a} \times \vec{x}$, If $\vec{a}$ =[a_x, a_y,a_z], then

$$\vec{a}^* = \begin{bmatrix} 0 & -a_z & a_y \\ a_z & 0 & -a_x \\ -a_y & a_x & 0 \end{bmatrix}$$ is the dual matrix of $\vec{a}$

- $(\vec{x}\cdot\vec{a})\vec{a}$ as a matrix multiply:

$$\vec{a}\vec{a}^T = \begin{bmatrix} a_x \\ a_y \\ a_z \end{bmatrix} [a_x\ a_y\ a_z] = \begin{bmatrix} a_x^2 & a_x a_y & a_x a_z \\ a_x a_y & a_y^2 & a_y a_z \\ a_x a_z & a_y a_z & a_z^2 \end{bmatrix} \qquad (\vec{a}\vec{a}^T)\begin{pmatrix} x \\ y \\ z \end{pmatrix} = Symmetric\left(\begin{bmatrix} a_x \\ a_y \\ a_z \end{bmatrix}\right)\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \vec{a}(\vec{a}\cdot\vec{x})$$

3D transformation

Axis-angle rotation in matrix representation

$$\vec{x}' = \cos\theta\,\vec{x} + (1-\cos\theta)(\vec{x}\cdot\vec{a})\vec{a} + \sin\theta\,(\vec{a}\times\vec{x})$$

$$\vec{x}' = [(\cos\theta)I + (1-\cos\theta)(\vec{a}\vec{a}^T) + \sin\theta\,(\vec{a}^*)]\vec{x}$$

The matrix $\mathbf{R}$ for rotation by θ about axis (unit) $\boldsymbol{a}$:

$$\mathbf{R} = \boldsymbol{a}\boldsymbol{a}^T + \cos\theta(\boldsymbol{I} - \boldsymbol{a}\boldsymbol{a}^T) + \sin\theta\,\boldsymbol{a}^*$$

$\boldsymbol{a}\boldsymbol{a}^T$	Project onto $\boldsymbol{a}$
$\boldsymbol{I} - \boldsymbol{a}\boldsymbol{a}^T$	Project onto $\boldsymbol{a}$'s normal plane
$\boldsymbol{a}^*$	Dual matrix. Project onto normal plane, flip by 90°
$\cos\theta, \sin\theta$	Rotate by θ in normal plane (assumes $\boldsymbol{a}$ is unit.)

3D transformation

Axis-angle rotation in matrix representation

When $\theta = 0$:

$$R = \vec{a}\vec{a}^T(1-1) + 0\vec{a}^* + 1I = I$$

When rotate around x-axis ($\vec{a} = [1\ 0\ 0]^T$):

$$R_x = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}(1-\cos\theta) + \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & 1 & 0 \end{bmatrix}\sin\theta + \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}\cos\theta$$

$$= \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix}$$

3D transformation

Axis-angle rotation in matrix representation

When rotate around y-axis ($\vec{a} = [0\ 1\ 0]^T$):

$$R_y = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}(1 - \cos\theta) + \begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ -1 & 0 & 0 \end{bmatrix}\sin\theta + \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}\cos\theta$$

$$= \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix}$$

3D transformation

Axis-angle rotation in matrix representation

When rotate around z-axis ($\vec{a} = [0\ 1\ 0]^T$):

$$R_z = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} (1 - \cos\theta) + \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \sin\theta + \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \cos\theta$$

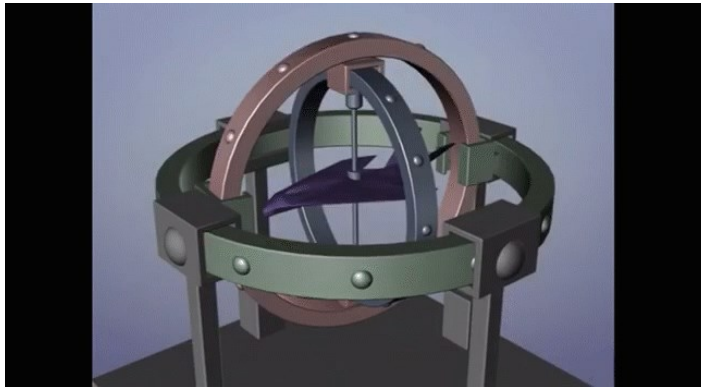
$$= \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

3D transformation

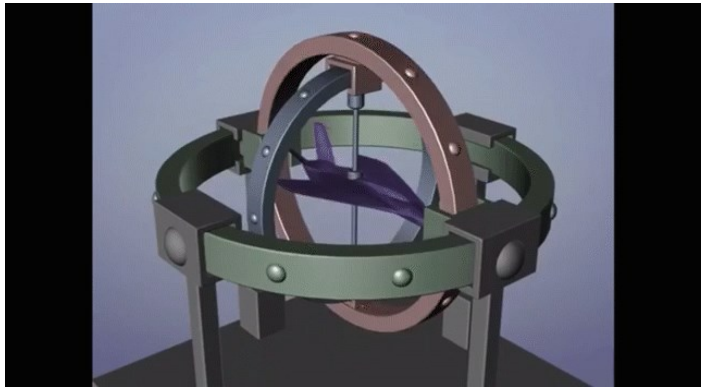
3D rotation

Euler angles – 3 rotations about each coordinate axis

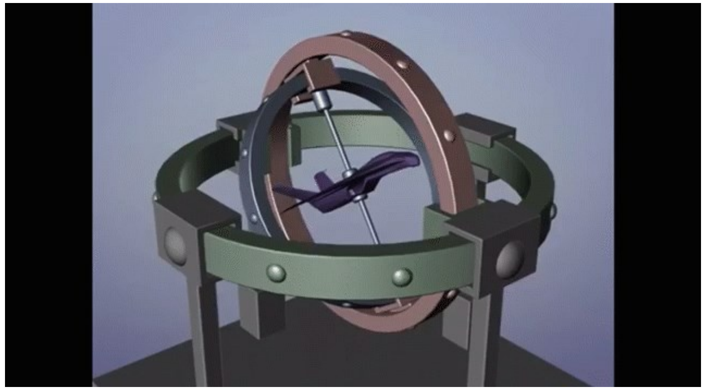
Widely used, because they're 'simple'



yaw



pitch



roll

- Heading / Yaw = rotation z
- Pitch = rotation x
- Roll = rotation y

3D transformation

3D rotation

Euler angles – 3 rotations about each coordinate axis

Widely used, because they're 'simple'

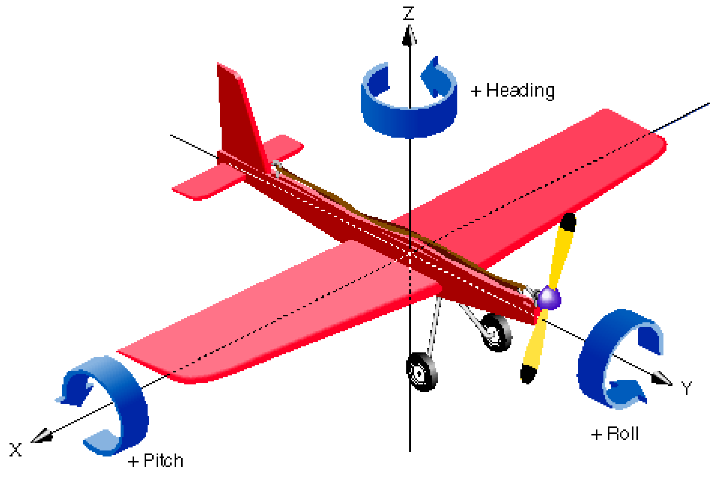
However,

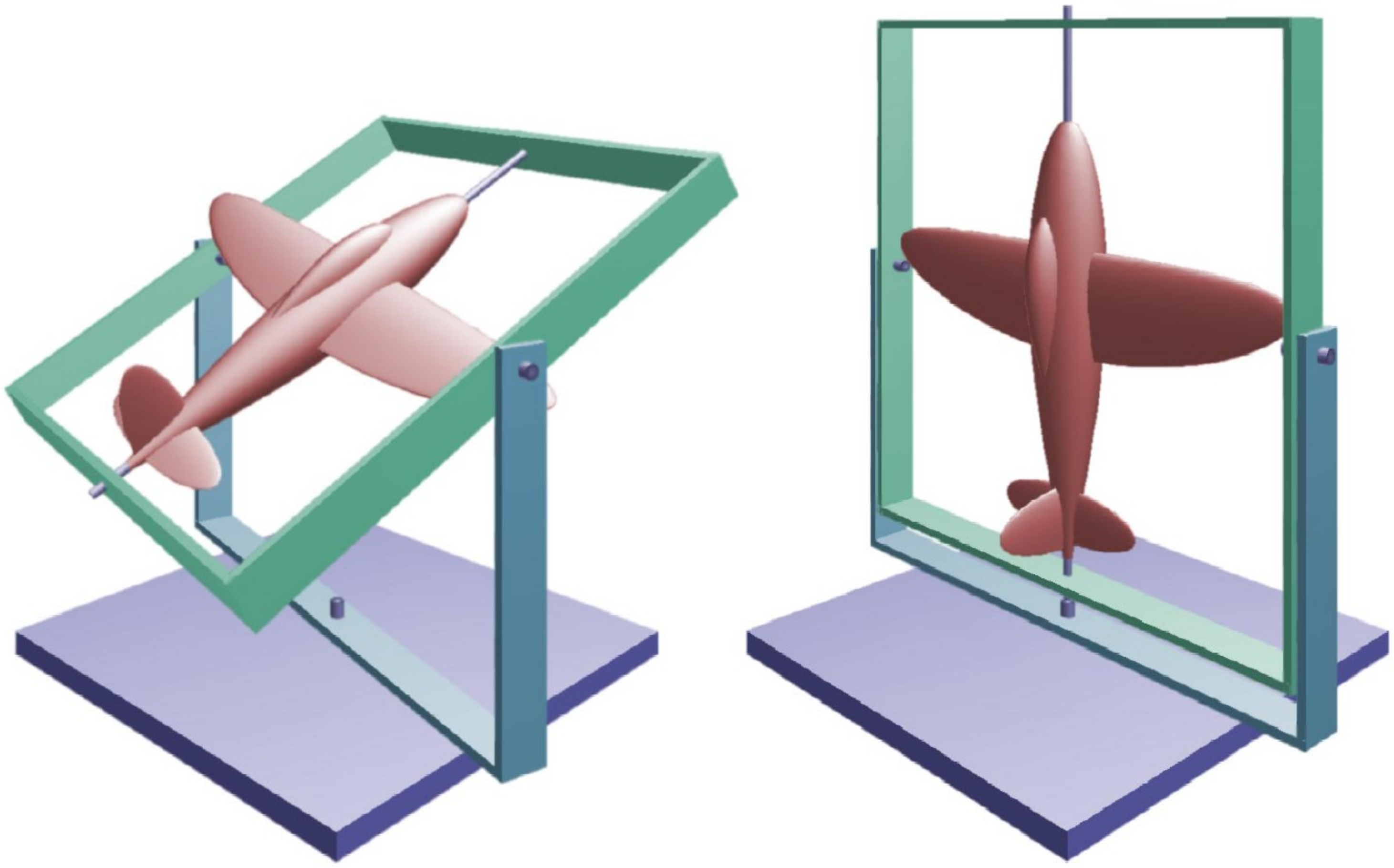
- angle interpolation generates bizarre motions (not linear)
  - $R_z(90°)R_y(90°) = R_{[1\ 1\ 1]^T}(120°)$

    But, $R_z(30°)R_y(30°) \approx R_{[1\ 0.3\ 1]^T}(42°) \neq\ R_{[1\ 1\ 1]^T}(40°)$
- rotations are order-dependent, but no conventions about the order
  - $R_z(\gamma)\ R_y(\beta)\ R_x(\alpha)\ \neq R_y(\beta)\ R_x(\alpha) R_z(\gamma)$

3D transformation

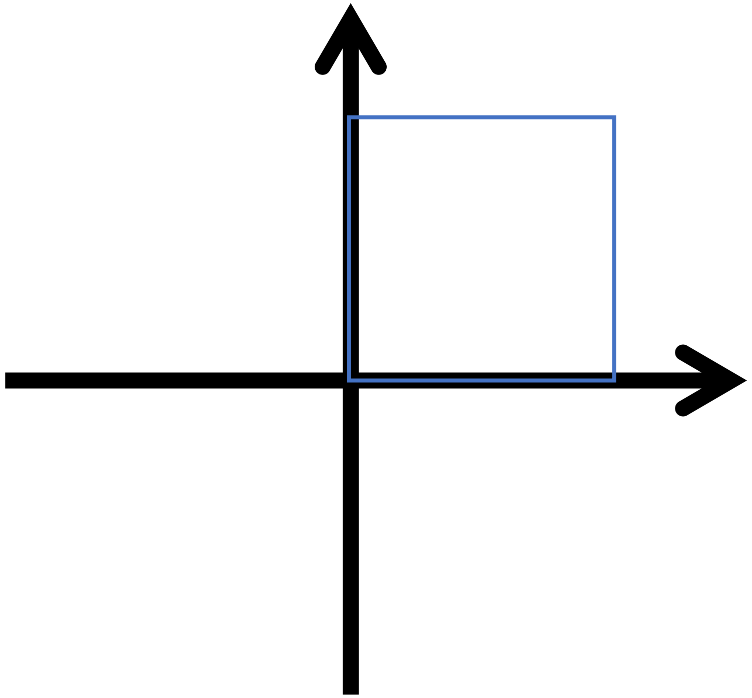
Euler angles – The alignment of two or more axes results in a loss of rotational DoFs.

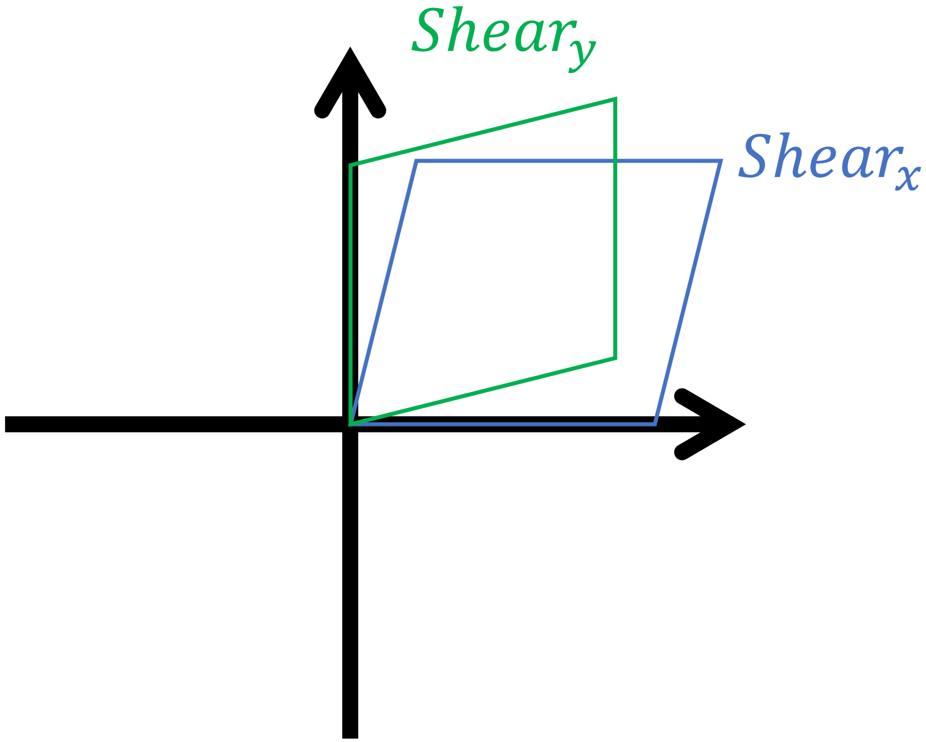




Other transformation

shear





$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & sh_x & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ sh_y & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

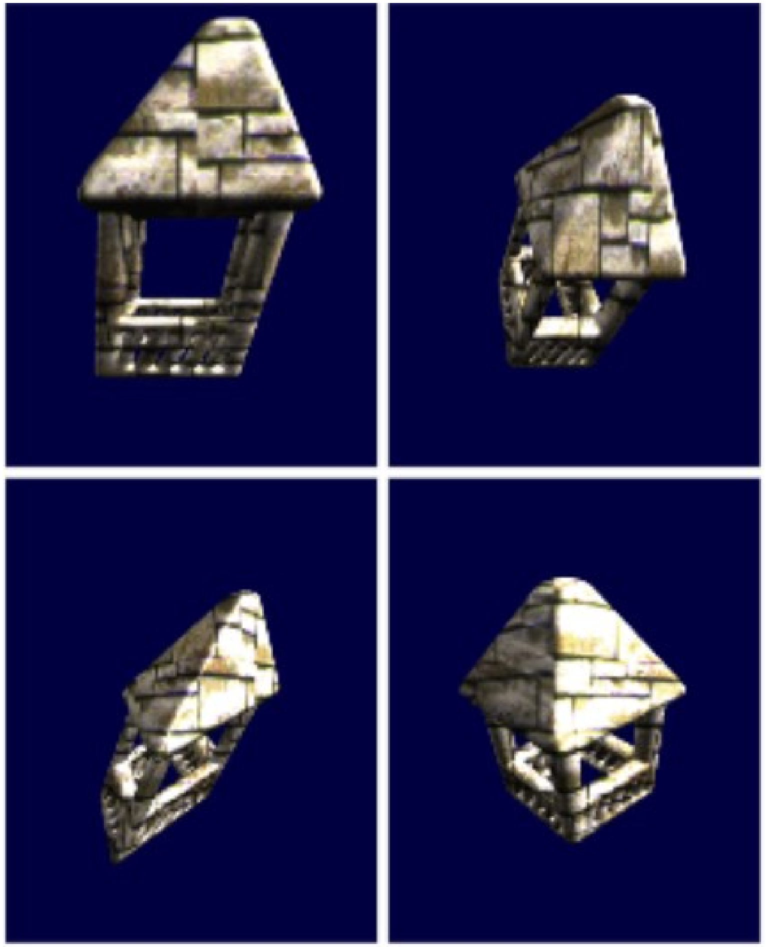
Other transformation

2D Rotation by Shears

Baoquan Chen and Arie Kaufman, Two-Pass Image and Volume Rotation, IEEE Workshop on Volume Graphics, 2001.
https://cfcs.pku.edu.cn/baoquan/docs/20180622110639032149.pdf

Other transformation

3D Rotation by Shears



Baoquan Chen and Arie Kaufman, 3D Volume Rotation Using Shear Transformations, Graphical Models, vol. 62, 2000, pp 308 -- 322. https://cfcs.pku.edu.cn/baoquan/docs/20180622110606347062.pdf

Up to this point: Affine Transformations

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} a_{xx} & a_{xy} & a_{xz} & b_x \\ a_{yx} & a_{yy} & a_{yz} & b_y \\ a_{zx} & a_{zy} & a_{zz} & b_z \\ a_x & a_y & a_z & \mathrm{b} \end{bmatrix} \begin{bmatrix} x \\ y \\ \mathrm{z} \\ 1 \end{bmatrix}$$

$$\begin{matrix} 0 & 0 & 0 & 1 \end{matrix}$$

Translation

Rotation

Scale

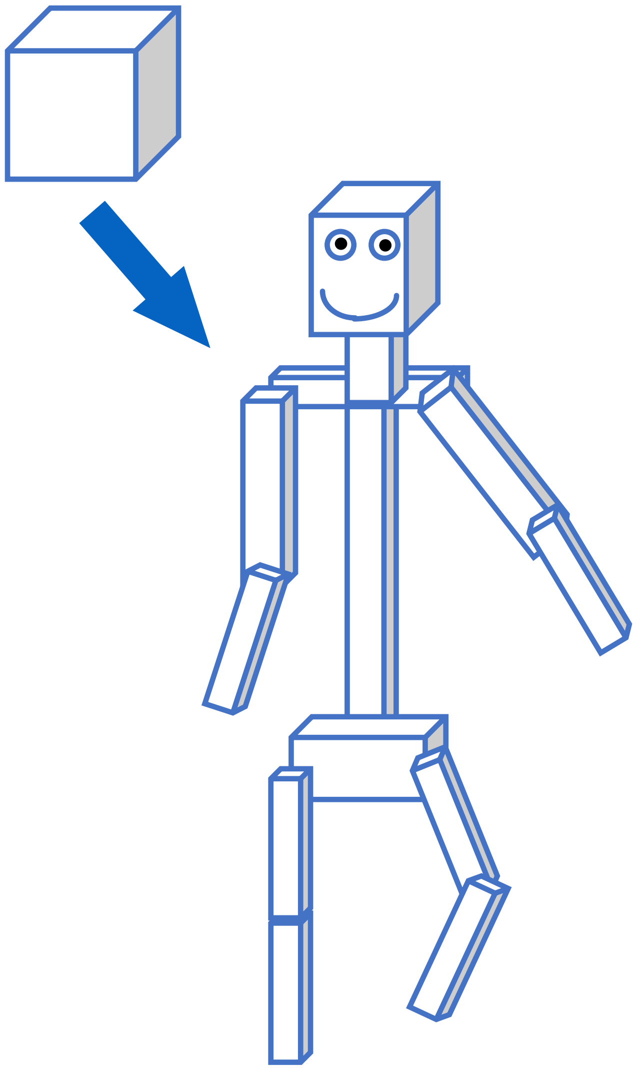
Shear

Coordinate transformation

……

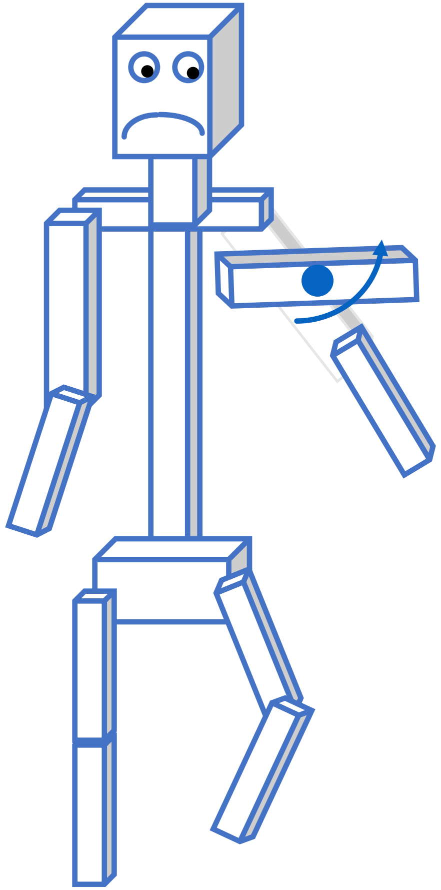
Model and Transformation Hierarchy

How to Model a Stick Person



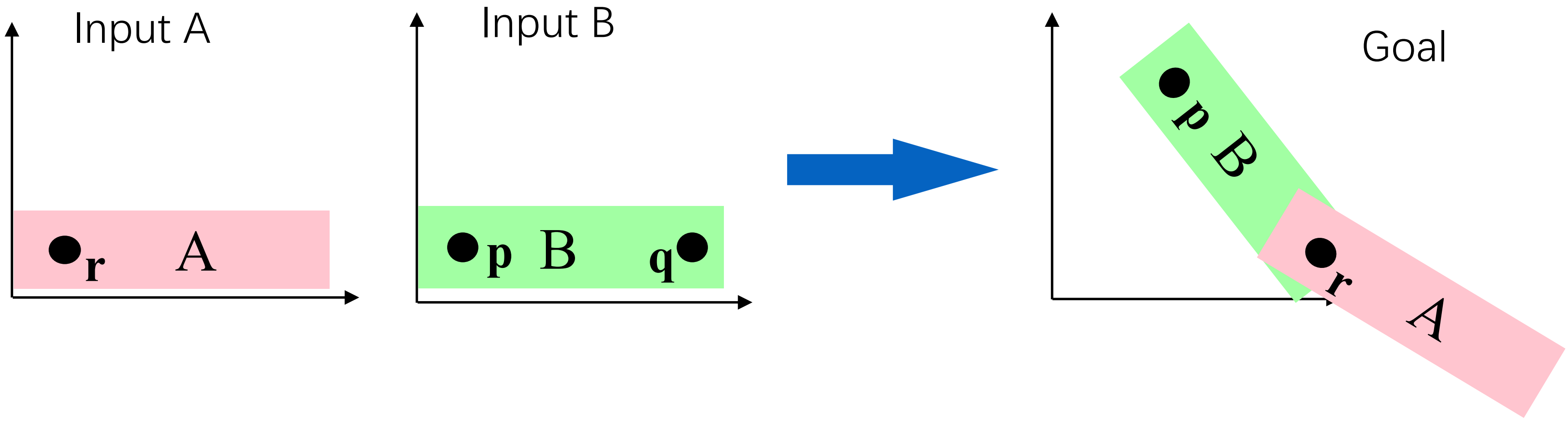
- Make a stick person out of cubes
- Just translate, rotate, and scale each one to get the right size, shape, position, and orientation.

The Right Control Knobs



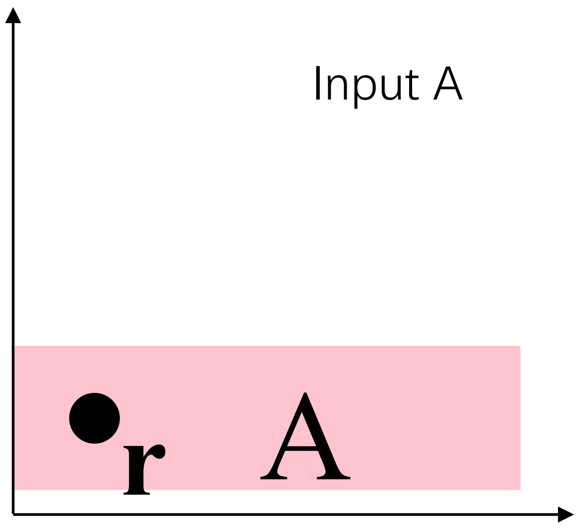
- As soon as you want to change something, the model *likely* falls apart
- Reason: the thing you're modeling is *constrained* but your model doesn't know it
- Solution:
    - some sort of representation of *structure*
    - *Control knob*
- This kind of control knob is convenient for static models, and *vital* for animation!
- Key: using a hierarchy to structure the transformations in the right way

Making an Articulated Model



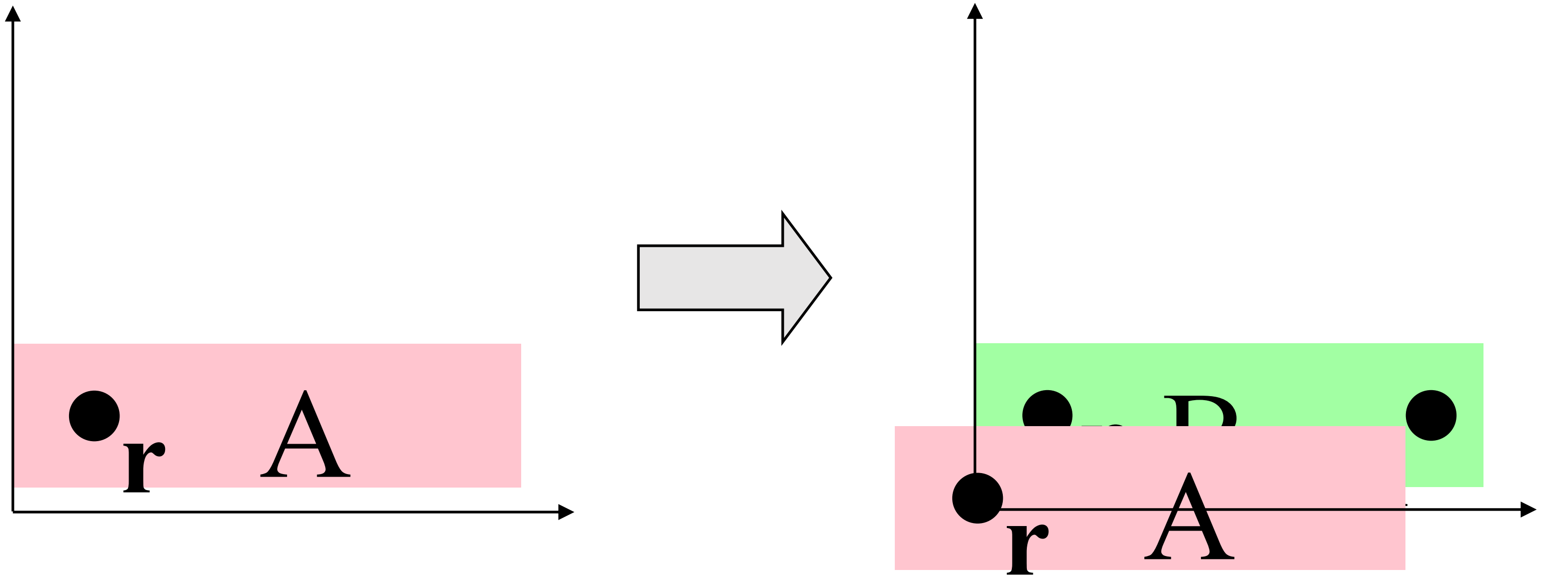
- A minimal 2-D jointed object:
  - Two pieces, *A* ("forearm") and *B* ("upper arm")
  - Attach point q on *B* to point r on *A* ("elbow")
  - Desired control knobs:
    - u: shoulder angle (*A* and *B* rotate together about p)
    - v: elbow angle (*A* rotates about r, which stays attached to p)

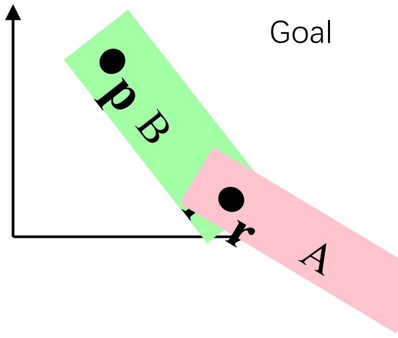
Making an Arm, step 1



- Start with A and B in their untransformed configurations (B is hiding behind A)
- First apply a series of transformations to A, leaving B where it is…

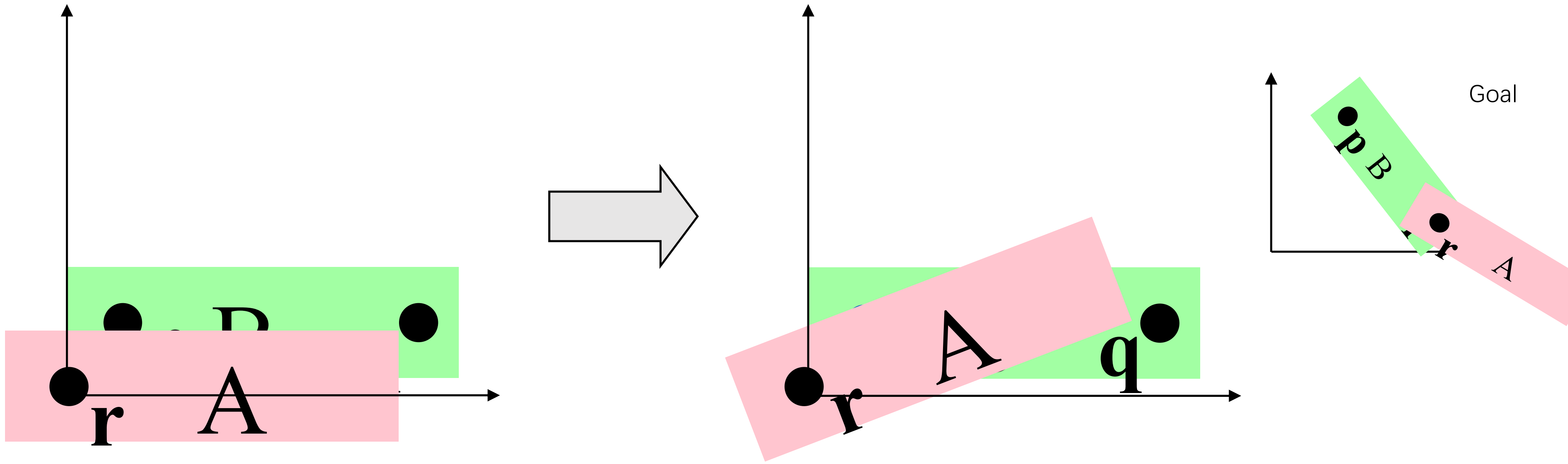
Making an Arm, step 2





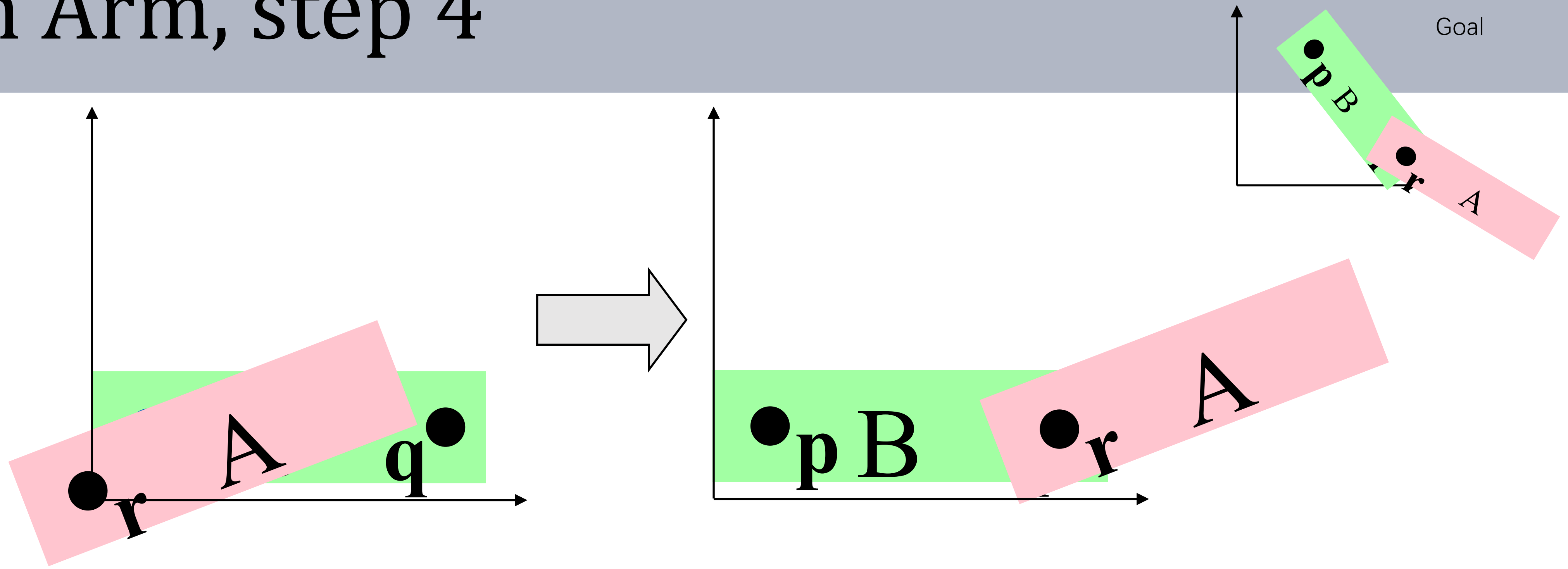
- Translate by -r, bringing r to the origin
- You can now see *B* peeking out from behind *A*

Making an Arm, step 3



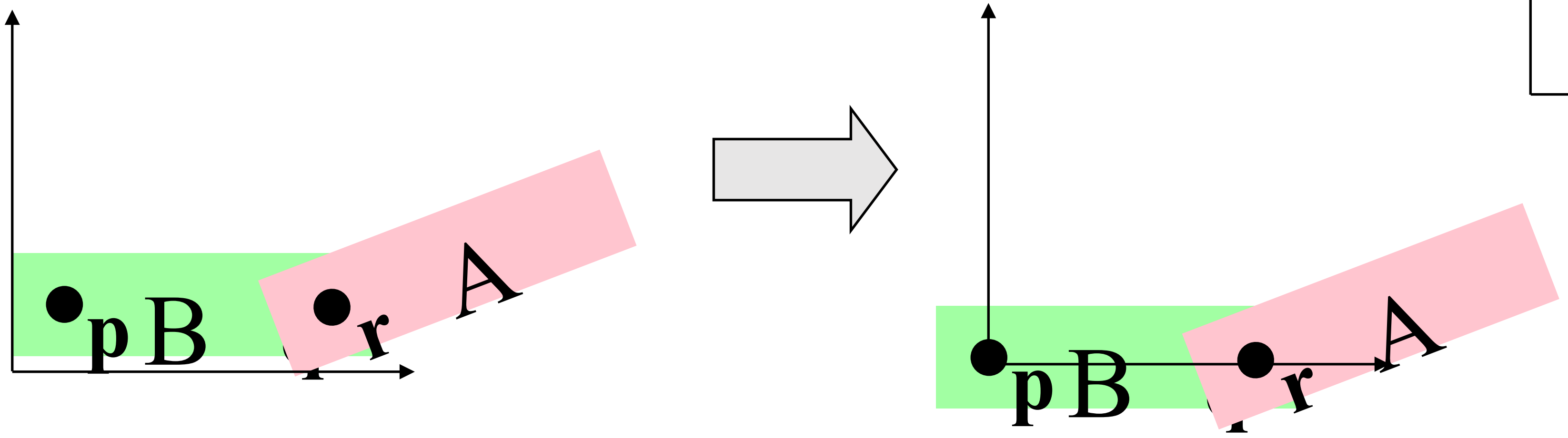
- Next, we rotate *A* by v (the “elbow” angle)

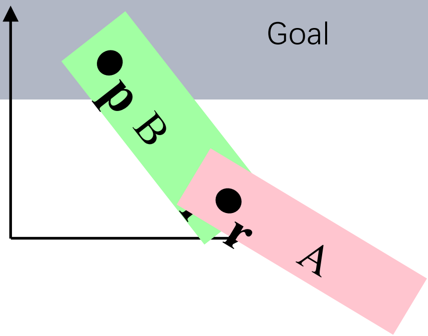
Making an Arm, step 4



- Translate *A* by q, bringing r and q together to form the elbow joint
- We can regard q as the origin of the *elbow coordinate system*, and regard A as being in this coordinate system.

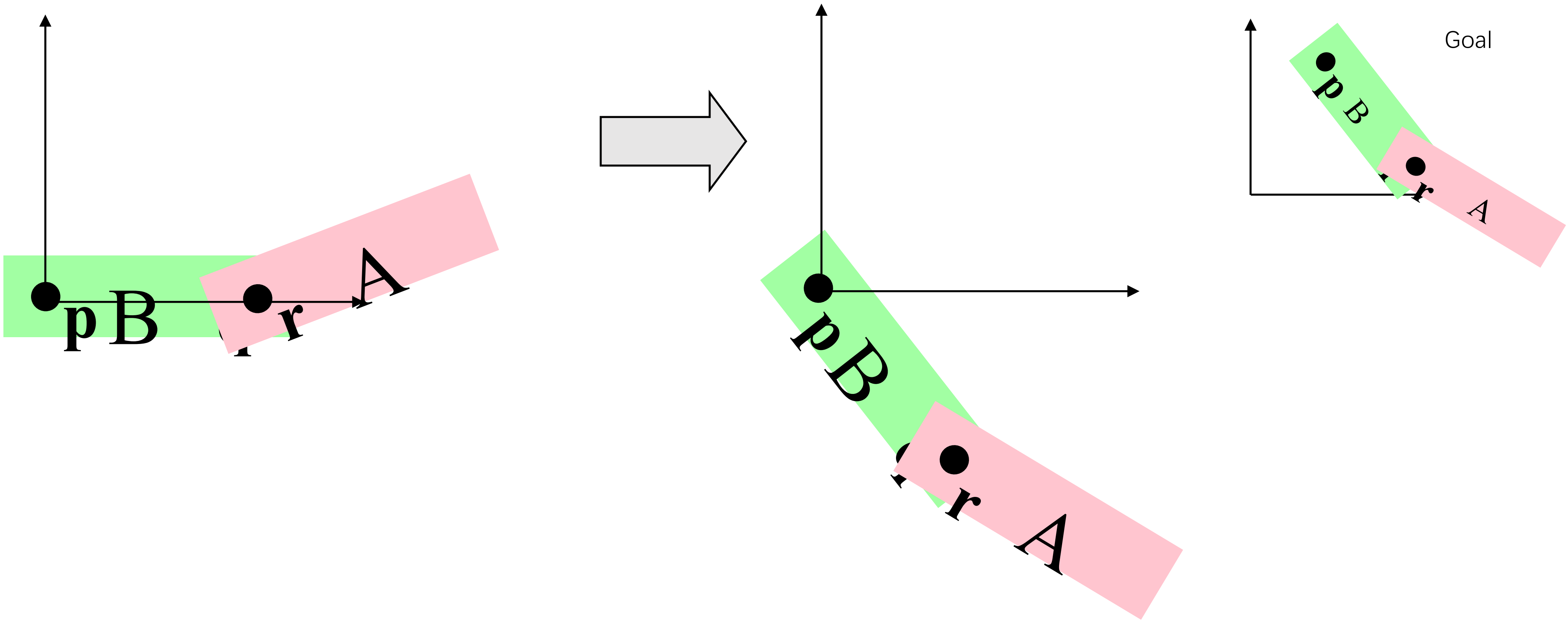
Making an Arm, step 5





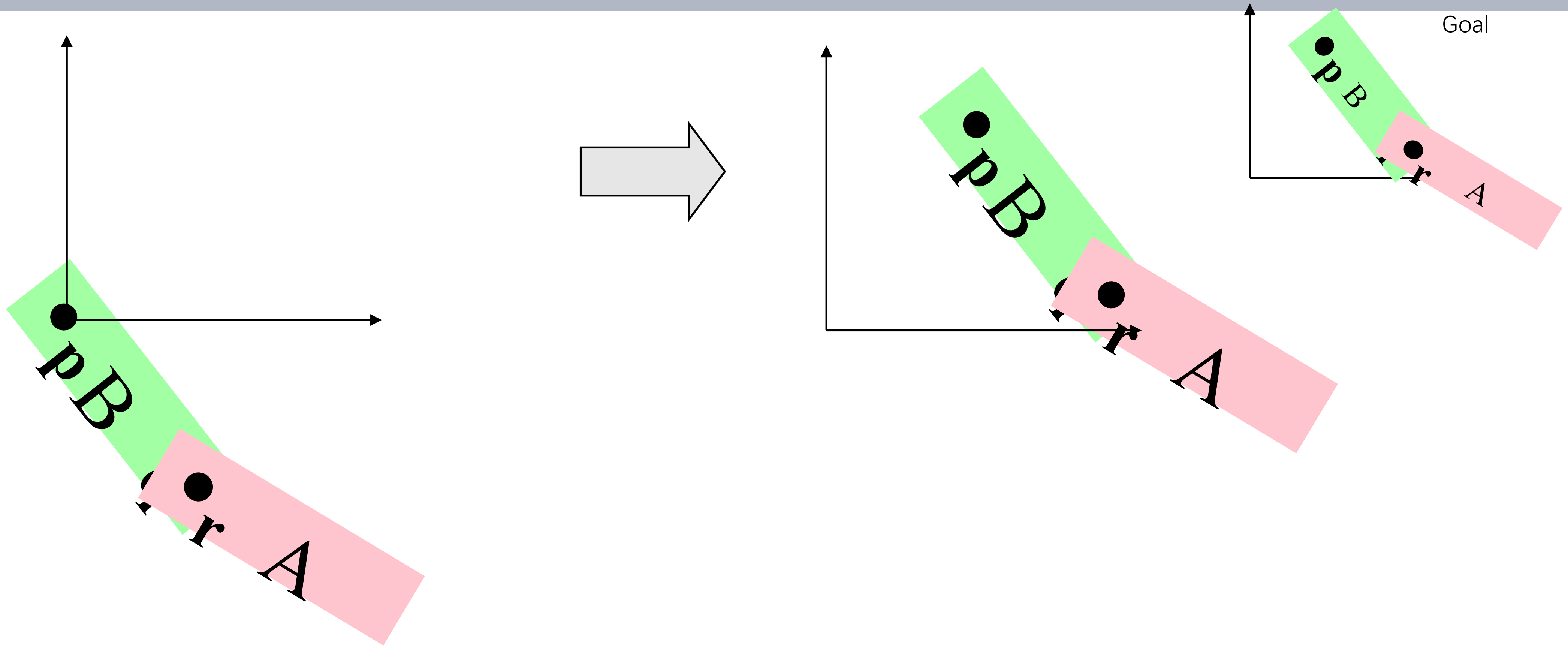
- From now on, each transformation applies to *both* A and B (This is important!)
- First, translate by -p, bringing p to the origin
- A and B both move together, so the elbow doesn't separate!

Making an Arm, step 6



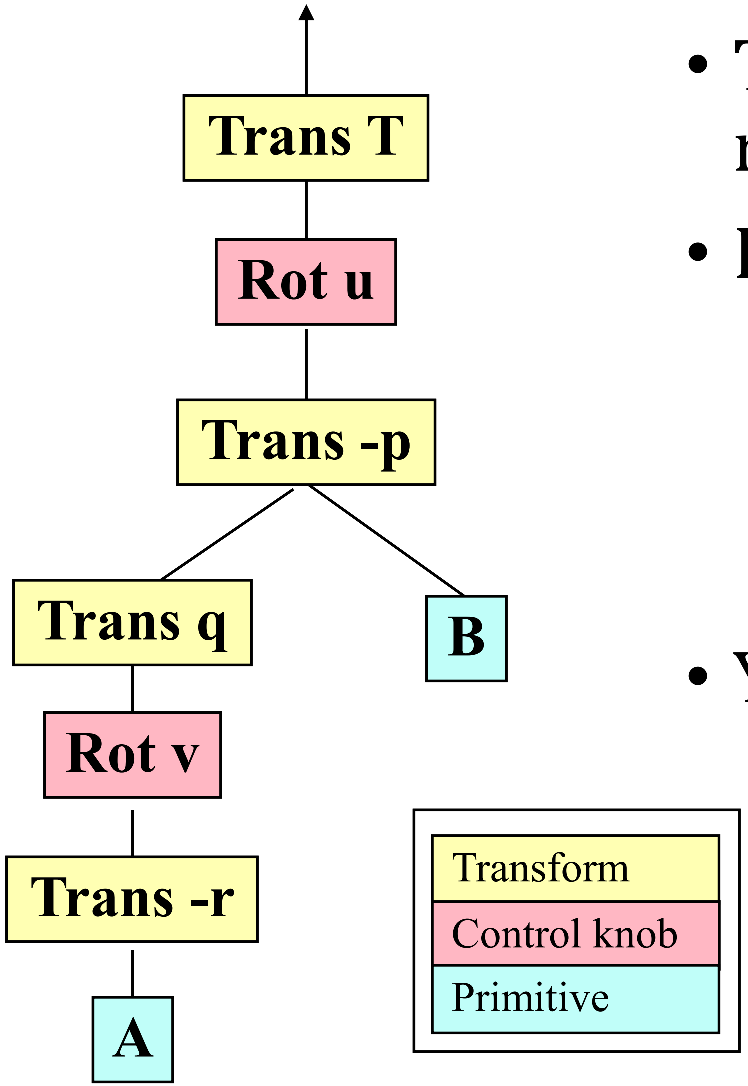
- Then, we rotate by u, the “shoulder” angle
- Again, A and B rotate together

Making an Arm, step 7



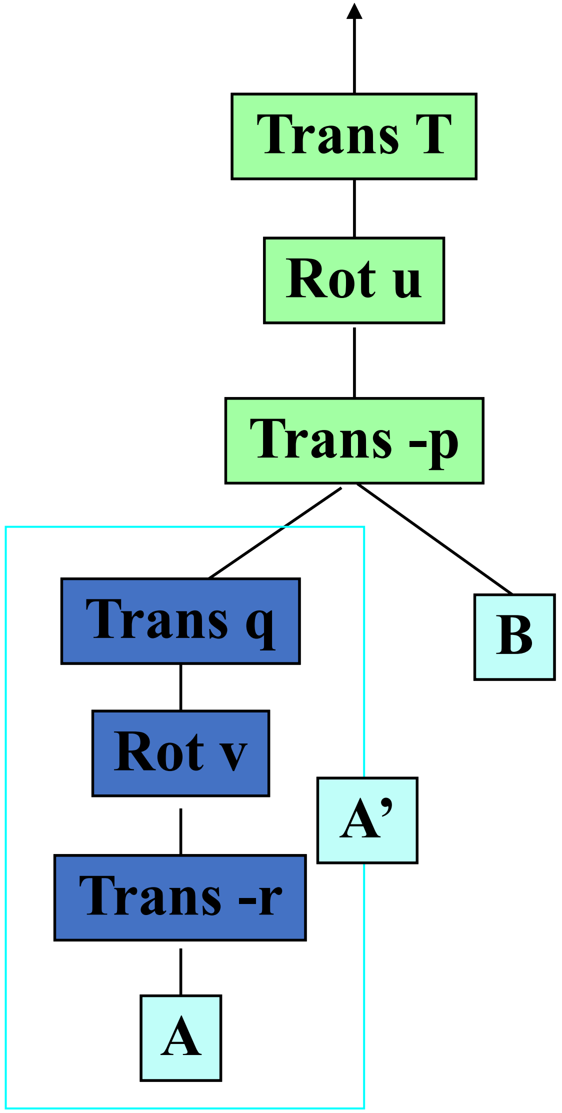
- Finally, translate by T, bringing the arm where we want it
- p is at origin of *shoulder coordinate system*

Transformation Hierarchies



- This is the build-an-arm sequence, represented as a tree
- Interpretation:
  - Leaves are geometric primitives
  - Internal nodes are transformations
  - Transformations apply to everything under them—start at the bottom and work your way up
- You can build a wide range of models this way
  - Similar data structures: scene graphs, omni-verse

Transformation Hierarchies

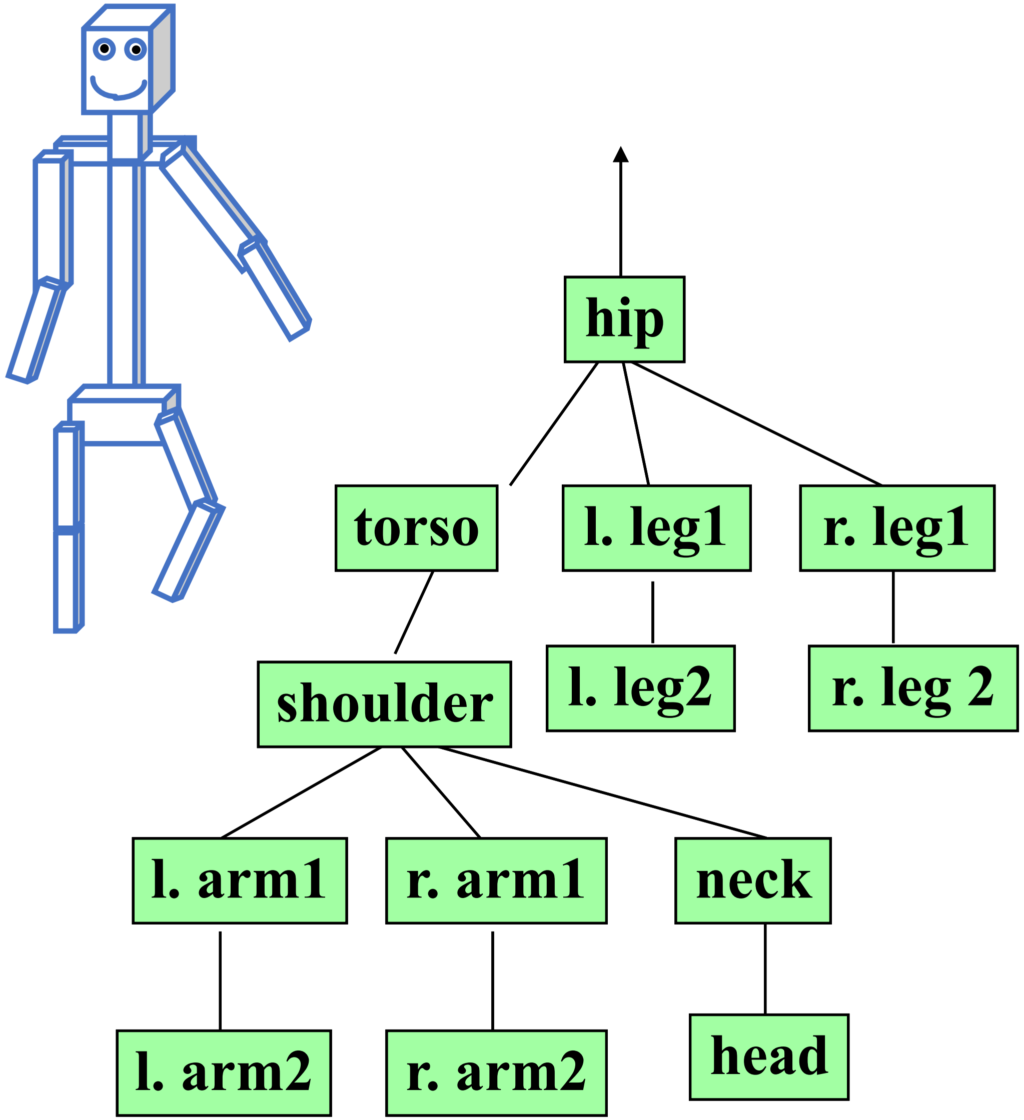


Another **hierarchical** point of view:

- The shoulder coordinate transformation moves everything below it with respect to the shoulder:
  - B
  - A and its transformation
- The elbow coordinate transformation moves A with respect to the elbow – A’

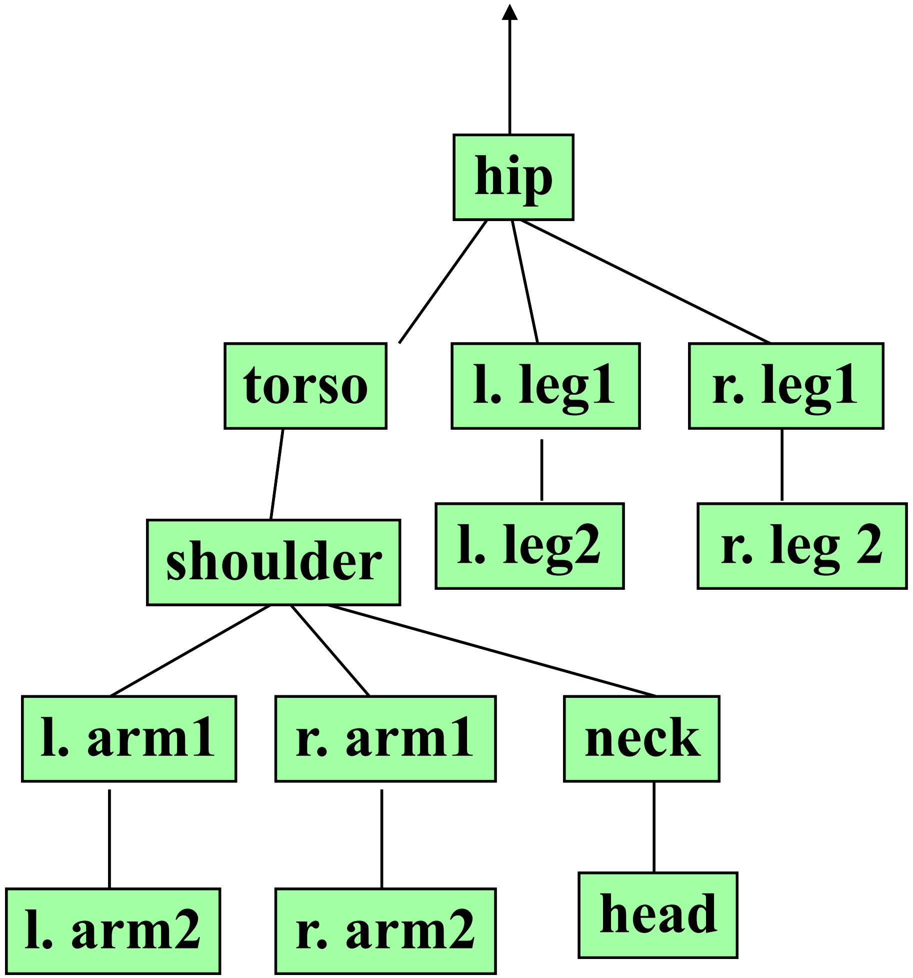
Shoulder coordinate transform
Elbow coordinate transform
Primitive

A Schematic Humanoid



- Each node represents
  - rotation(s)
  - geometric primitive(s)
  - struct. transformations
- The root can be anywhere. We chose the hip *(can re-root)*
- Control for each joint angle, plus global position and orientation
- A realistic human would be *much* more complex

Directed Acyclic Graph



- This is a graph, so you can re-root it.
- It's *directed*, rendering traversal only follows links one way.
- It's *acyclic*, to avoid infinite loops in rendering.
- Not necessarily a tree.
  - e.g. l.arm2 and r.arm2 primitives might be two instantiations (one mirrored) of the same geometry

What Hierarchies Can and Can't Do

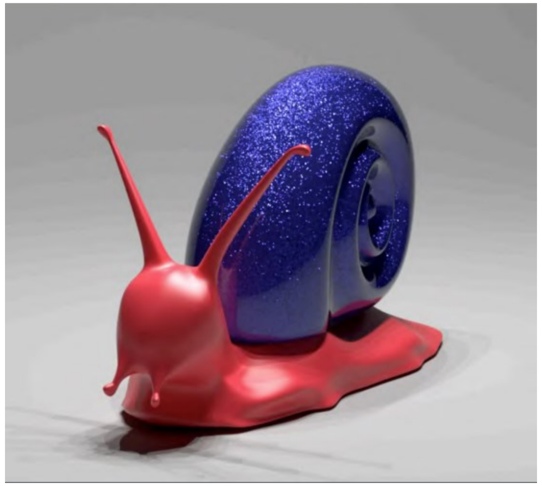
- Advantages:
  - Reasonable control knobs
  - Maintains structural constraints
- Disadvantages:
  - Can't do closed kinematic chains (keep hand on hip)
- A more general approach:
  - inverse kinematics - more complex, but better knobs (later!)
- Hierarchies are a vital tool for modeling and animation

Implementing Hierarchies

- Building block: a *matrix stack* that you can push/pop
- Recursive algorithm that descends your model tree, doing transformations, pushing, popping, and drawing
- Tailored to OpenGL's state machine architecture (or vice versa)

OpenGL

- What is OpenGL ?
  - ➢Software interface to graphics hardware
  - ➢About 120 C-callable routines for 3D graphics
  - ➢Platform independent graphics library



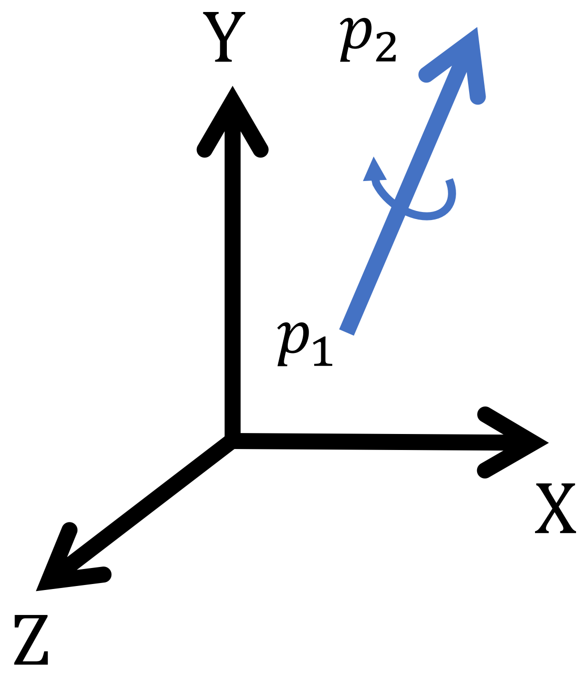
- What can it do ?
  - Display primitives
  - Coordinate transformations
  - Lighting calculations
  - Antialiasing
  - etc

OpenGL

- [Tutorial]
- Example of transformation function in OpenGL
  - Rotate with arbitrary vector (p1, p2)

```
void rotate3D(wcPt3D p1, wcPt3D p2, GLfloat thetaDegrees)
{
    float vx = (p2.x - p1.x);
    float vy = (p2.y - p1.y);
    float vz = (p2.z - p1.z);

    glTranslaterf (p1.x, p1.y, p1.z);
    glRotatef (thetaDegrees, vx, vy, vz);
    glTranslaterf (-p1.x, -p1.y, -p1.z);
}
```



glRotate and others will act on the current matrix, which is by default GL_MODELVIEW.
GL_MODELVIEW affects any 3D object drawn.
The model-view matrix is then applied to any geometry rendered with glVertex, glDrawArrays, etc.

OpenGL

- [Tutorial](https://learnopengl.com/) (https://learnopengl.com/)
- Example of transformation function in OpenGL
  - Translation : glTranslate (tx, ty, tz)
  - Rotation : glRotate(theta, vx, vy, vz)
  - Scale : glScale(sx, sy, sz)
  - Matrix : glMatrixMode(Mode)
    - GL_MODELVIEW : Applies subsequent matrix operations to the modelview matrix stack.
    - GL_PROJECTION : Applies subsequent matrix operations to the projection matrix stack.
    - GL_TEXTURE : Applies subsequent matrix operations to the texture matrix stack.
    - GL_COLOR : Applies subsequent matrix operations to the color matrix stack.
  - Matrix stack : manage transformation in a stack

The Matrix Stack

- Idea of Matrix Stack:
  - LIFO (Last In First Out) stack of matrices with push and pop operations
  - current transformation matrix (product of all transformations on stack)
  - transformations modify matrix at the top of the stack
- Recursive algorithm:
  - load the identity matrix
  - for each internal **node**:
    - » push a new matrix onto the stack
    - » concatenate transformations onto current transformation matrix
    - » recursively descend tree
    - » pop matrix off of stack
  - for each leaf node:
    - » draw the geometric primitive using the current transformation matrix

Relevant OpenGL routines

glPushMatrix(), glPopMatrix()

push and pop the stack. push leaves a copy of the current matrix on top of the stack

glLoadIdentity(), glLoadMatrixd(M)

load the Identity matrix, or an arbitrary matrix, onto top of the stack

glMultMatrixd(M)

multiply the matrix C on top of stack by M. C = CM

glRotatef(theta,x,y,z), glRotated(…)

axis/angle rotate. "f" and "d" take floats and doubles, respectively

glTranslatef(x,y,z), glScalef(x,y,z)

translate, rotate. (also exist in "d" versions.)

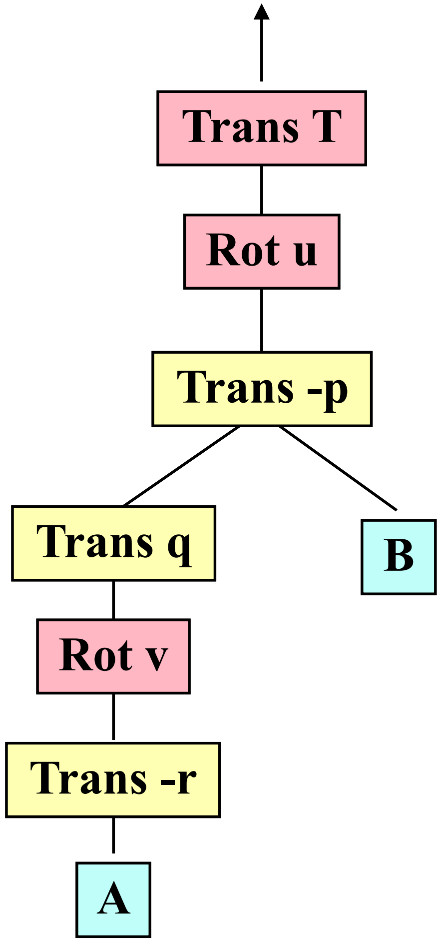
glOrtho (x0,y0,x1,y1,z0,z1)

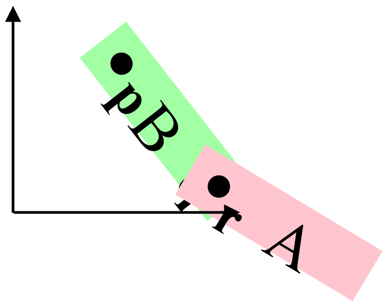
set up parallel projection matrix

Two-link arm, revisited, in OpenGL

Trace of Opengl calls

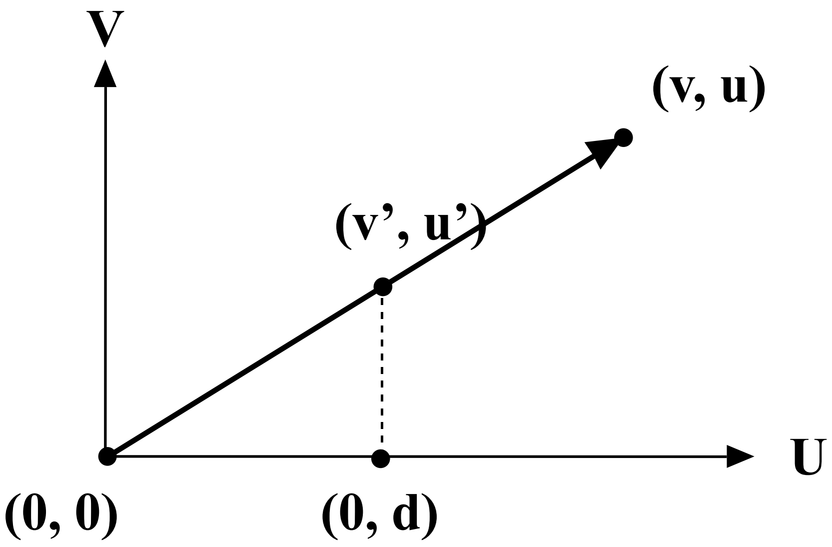
```
glLoadIdentity();
glOrtho(...);
glPushMatrix();
    glTranslatef(Tx,Ty,0);
    glRotatef(u,0,0,1);
    glTranslatef(-px,-py,0);
    glPushMatrix();
        glTranslatef(qx,qy,0);
        glRotatef(v,0,0,1);
        glTranslatef(-rx,-ry,0);
        Draw(A);
    glPopMatrix();
    Draw(B);
glPopMatrix();
```



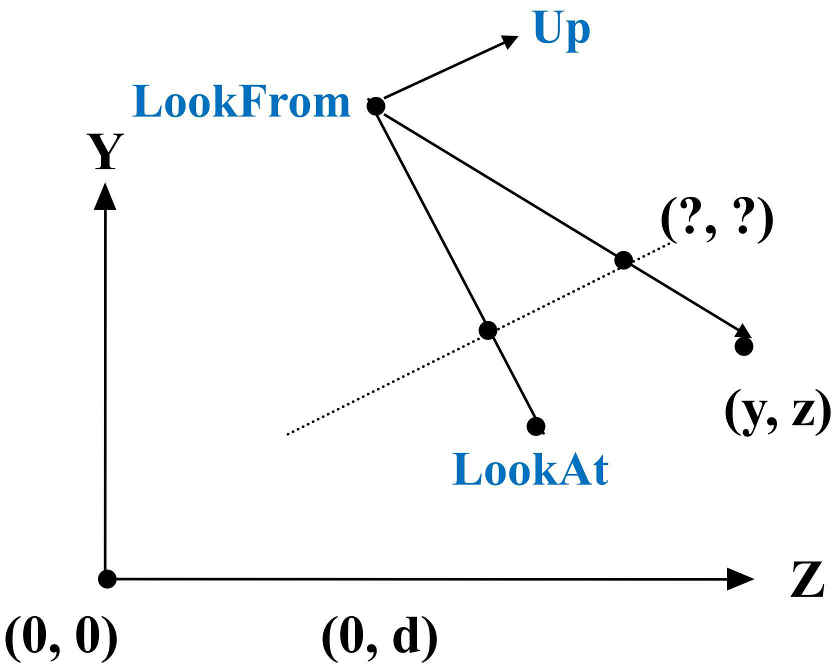


Viewing Transformation

Rendering from any camera position



Viewing Coord.

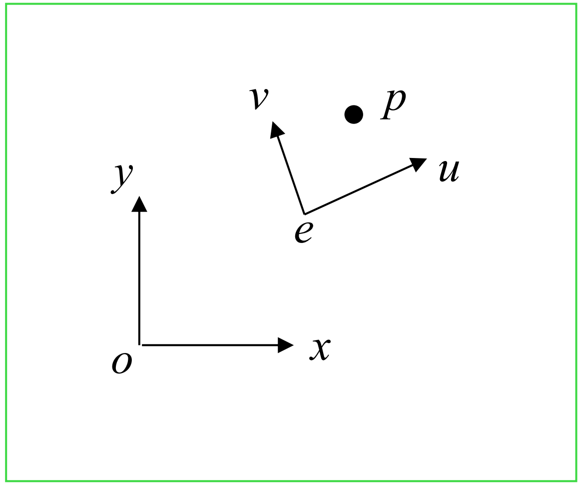


World Coord.

This involves Coordinate Transformation

$$\vec{p} = (p_x, p_y) \equiv \vec{o} + p_x\vec{x} + p_y\vec{y}$$

$$\vec{p} = (p_u, p_v) \equiv \vec{e} + p_u\vec{u} + p_v\vec{v}$$

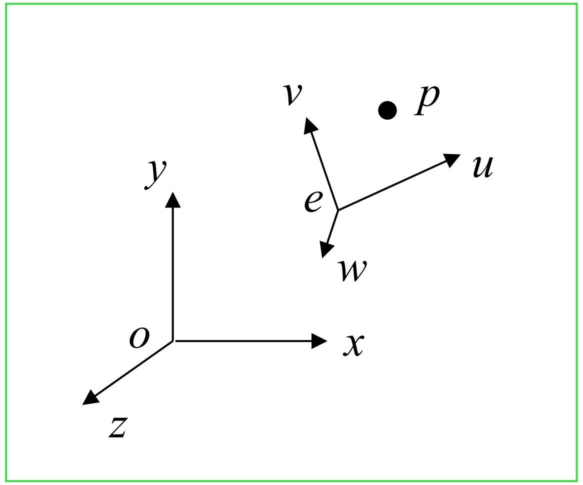


$$\begin{bmatrix} p_x \\ p_y \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & e_x \\ 0 & 1 & e_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u_x & v_x & 0 \\ u_y & v_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p_u \\ p_v \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} p_u \\ p_v \\ 1 \end{bmatrix} = \begin{bmatrix} u_x & u_y & 0 \\ v_x & v_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -e_x \\ 0 & 1 & -e_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ 1 \end{bmatrix} = \begin{bmatrix} u_x & u_y & -e_x \\ v_x & v_y & -e_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ 1 \end{bmatrix}$$

3D Coordinate Transformation

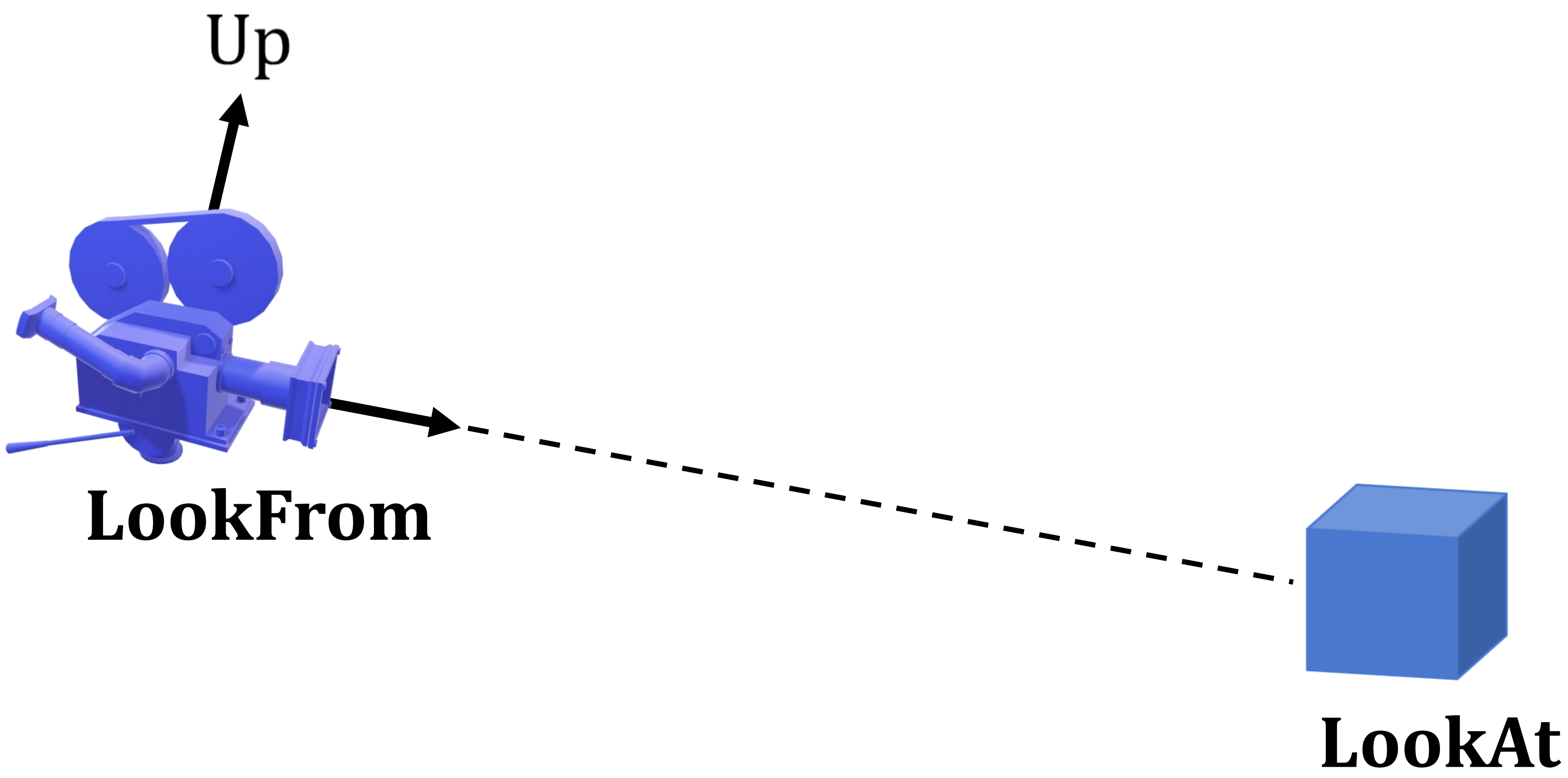
$$\begin{bmatrix} p_x \\ p_y \\ p_z \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & e_x \\ 0 & 1 & 0 & e_y \\ 0 & 0 & 1 & e_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u_x & v_x & w_x & 0 \\ u_y & v_y & w_y & 0 \\ u_z & v_z & w_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p_u \\ p_v \\ p_w \\ 1 \end{bmatrix}$$



$$\begin{bmatrix} p_u \\ p_v \\ p_w \\ 1 \end{bmatrix} = \begin{bmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ w_x & w_y & w_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -e_x \\ 0 & 1 & 0 & -e_y \\ 0 & 0 & 1 & -e_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ p_z \\ 1 \end{bmatrix}$$

Viewing transformations

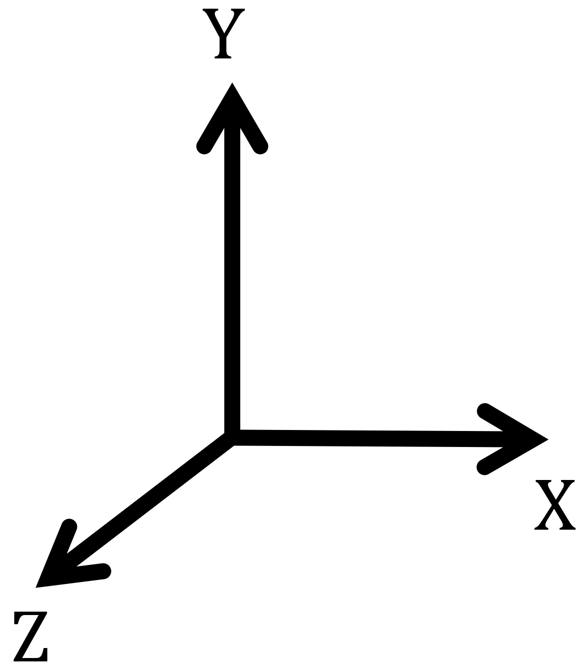
Camera model

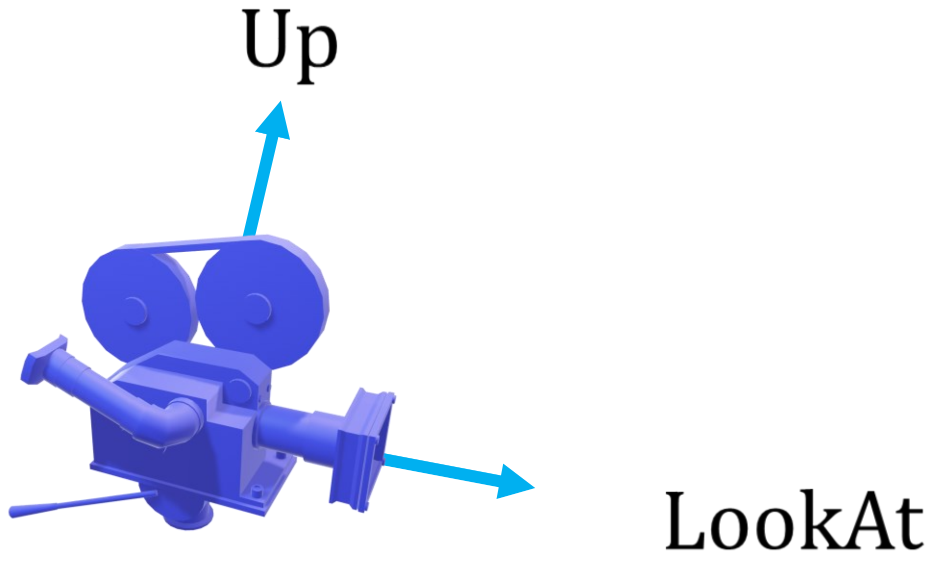


viewing vector: $\boldsymbol{v}$ = (*lookat-lookfrom*)

Viewing transformations

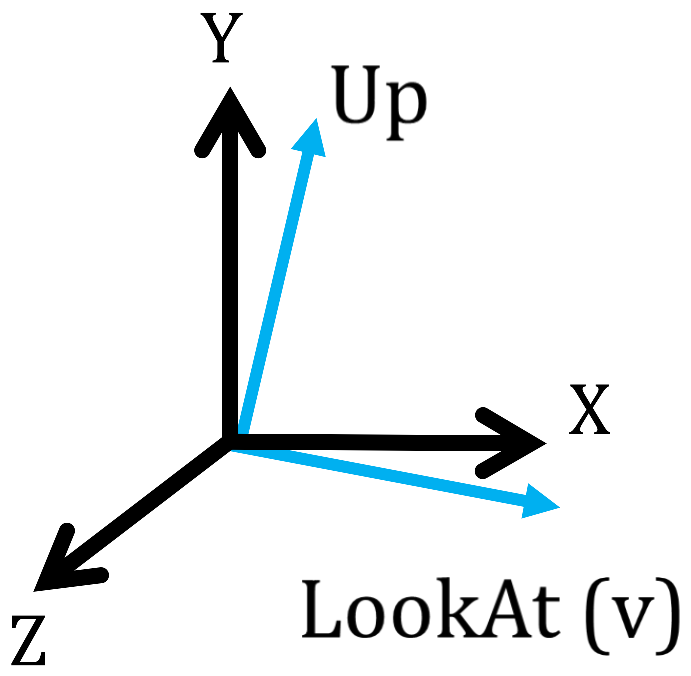
Coordinate transform



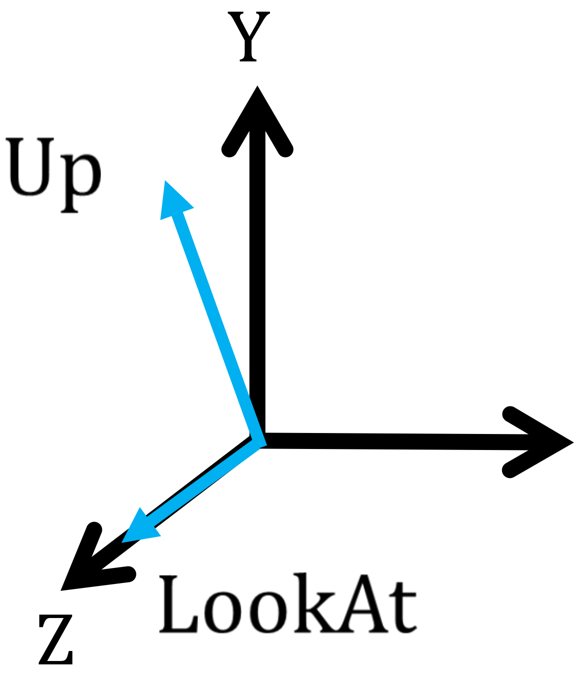


Viewing transformations

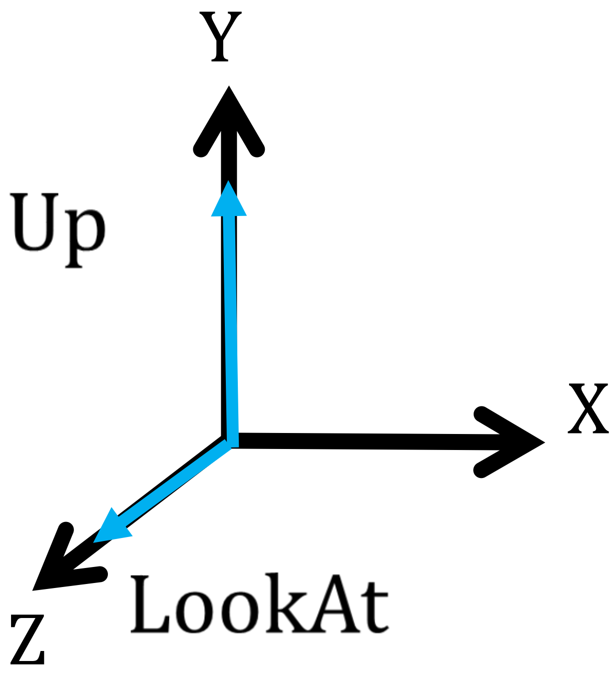
Coordinate transform



Translate camera to origin



Rotate v to Z axis
Rotate around v×z



Rotate Up to Y axis
Rotate around Z

Implementation

Implementing the *lookat/lookfrom/up* viewing scheme

(1) Translate by -*lookfrom*, bring focal point to origin

(2) Rotate *lookat-lookfrom* to the z-axis with matrix R:

- $\boldsymbol{v}$ = (*lookat-lookfrom*) (normalized) and $\boldsymbol{z} = [0\ 0\ 1]$
- rotation axis: $\boldsymbol{a} = (\boldsymbol{v} \times \boldsymbol{z})/|\boldsymbol{v} \times \boldsymbol{z}|$
- rotation angle: $\cos\theta = \boldsymbol{v} \cdot \boldsymbol{z}$ and $\sin\theta = |\boldsymbol{v} \times \boldsymbol{z}|$

$$\mathbf{R} = \boldsymbol{a}\boldsymbol{a}^T + (\boldsymbol{v} \cdot \boldsymbol{z})(\boldsymbol{I} - \boldsymbol{a}\boldsymbol{a}^T) + |\boldsymbol{v} \times \boldsymbol{z}|\boldsymbol{a}^* \quad \text{where} \quad \boldsymbol{a}^* = \begin{bmatrix} 0 & -a_z & a_y \\ a_z & 0 & -a_x \\ -a_y & a_x & 0 \end{bmatrix}$$

or: glRotate(θ, a_x, a_y, a_z)

(3) Rotate about z-axis to get projection of Up parallel to the y-axis

Viewing Projection