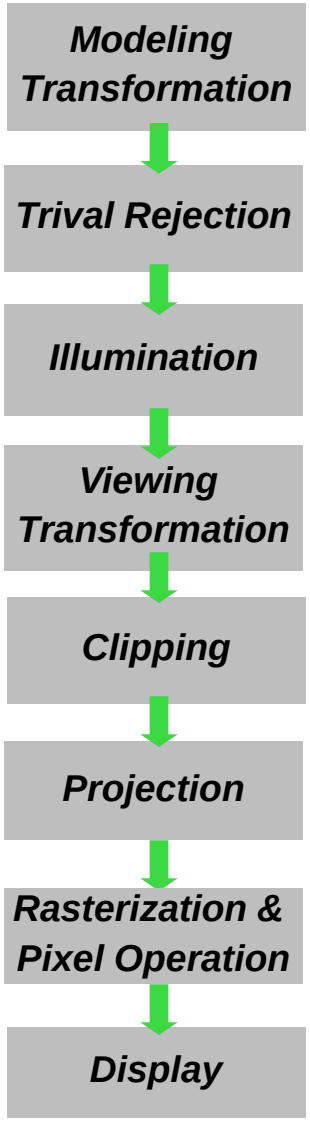
The 3-D Graphics Rendering Pipeline

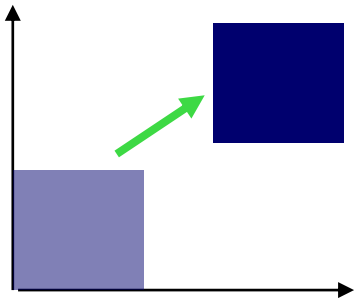


- *Almost every discussion of 3-D graphics begins here*
- *Seldom are any two versions drawn the same way*
- *Seldom are any two versions implemented the same way*
- *Primitives are processed in a series of steps*
- *Each step forwards its result on to the next step*

Transformation

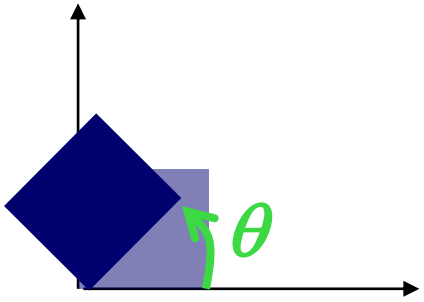
1. 2D Transformation
2. 3D Transformation
3. Viewing Projection

2D Translation



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

2D Rotation

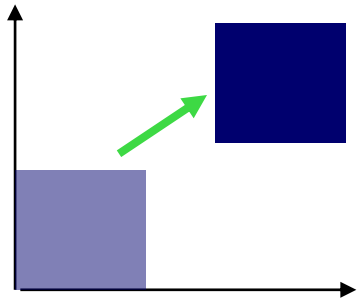


$$\begin{cases} x' = x \cdot \cos\theta - y \cdot \sin\theta \\ y' = x \cdot \cos\theta + y \cdot \sin\theta \end{cases}$$

Matrix and Vector format:

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

Back to Translation



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

Matrix format?

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = M \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} ?? & ?? \\ ?? & ?? \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

$$M = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix}$$

Homogenous coordinates!

$$\begin{bmatrix} wx' \\ wy' \\ w \end{bmatrix} = M \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} m_{00} & m_{01} & m_{02} \\ m_{10} & m_{11} & m_{12} \\ m_{20} & m_{21} & m_{22} \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

2D Translation Properties

1. There exists an inverse mapping for each function
2. There exists an identity mapping

$$M^{-1} = \begin{bmatrix} 1 & 0 & -t_x \\ 0 & 1 & -t_y \\ 0 & 0 & 1 \end{bmatrix}$$

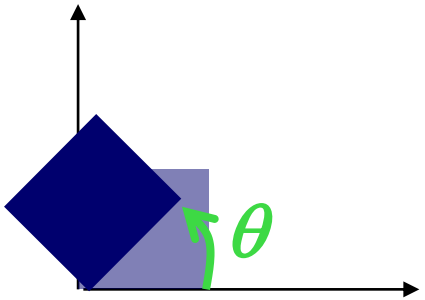
$$M\Big|_{\substack{t_x=0 \\ t_x=0}} = \begin{bmatrix} 1 & 0 & -t_x \\ 0 & 1 & -t_y \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = Identity(I)$$

2D Translation Properties

These properties might seem trivial at first glance, but they are actually very important, because when these conditions are shown for any class of functions it can be proven that such a class is closed under composition (i.e. any series of translations can be composed to a single translation). In mathematical parlance this is the same as saying that translations form an algebraic group.

$$x' = \underbrace{T_1 T_2 \bullet \cdots T_n}_{T'} x$$

Back to Rotation



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

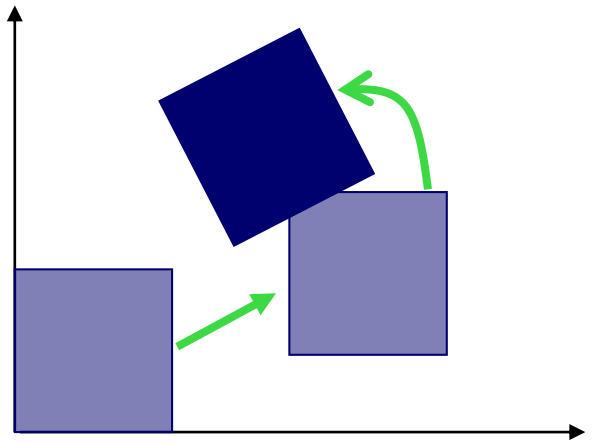
$$M_R = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

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

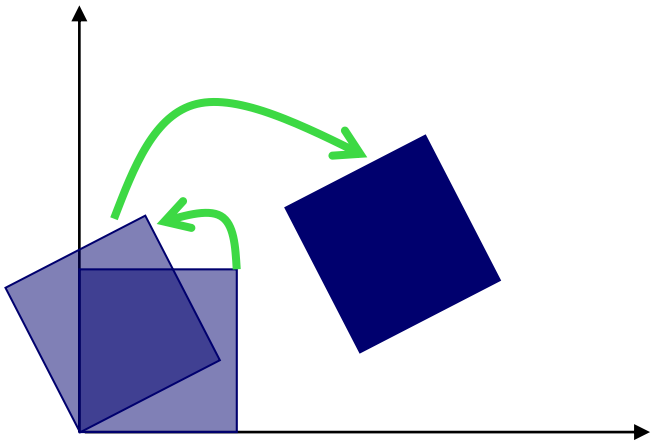
$$M_R\big|_{\theta=0} = Identity$$

Transformation Order

Order matters!

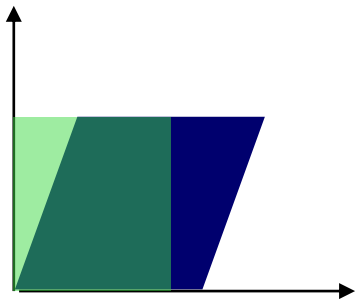


translation ---> rotation

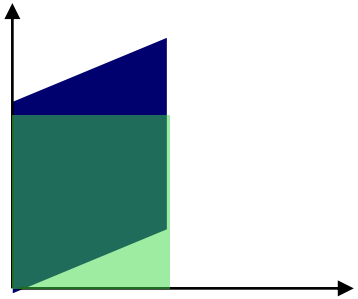


rotation ---> translation

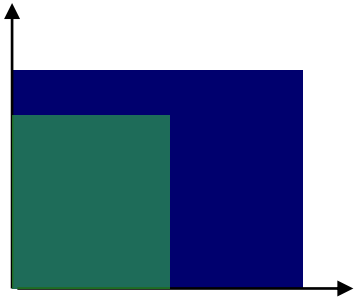
Other 2D Transformations



X-shear



Y-shear



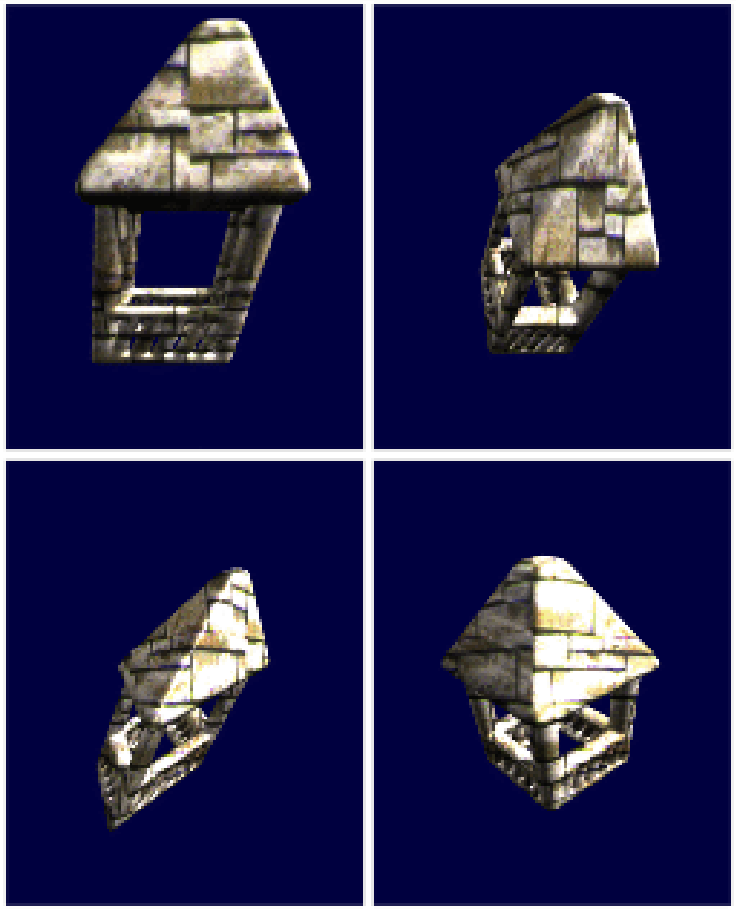
scaling

And more …….

2D Rotation by Shears

http://cfcs.pku.edu.cn/~baoquan/papers/rot2p.pdf

3D Rotation by Shears



http://cfcs.pku.edu.cn/~baoquan/papers/rot.pdf

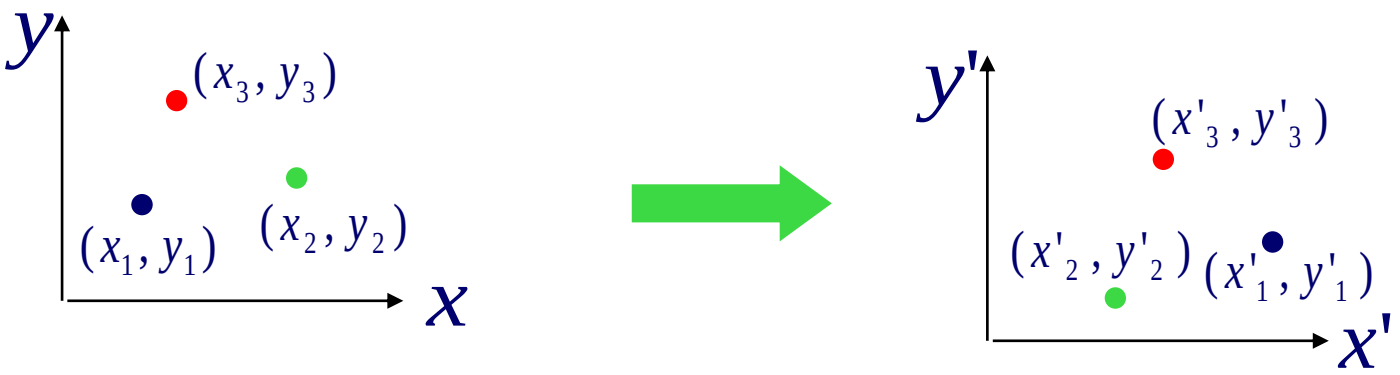
Affine transformation

Property: preserve parallel lines

Remember affine function on vector is equal to linear plus translation

$$M = \begin{bmatrix} m_{00} & m_{01} & m_{02} \\ m_{10} & m_{11} & m_{12} \\ 0 & 0 & 1 \end{bmatrix}$$

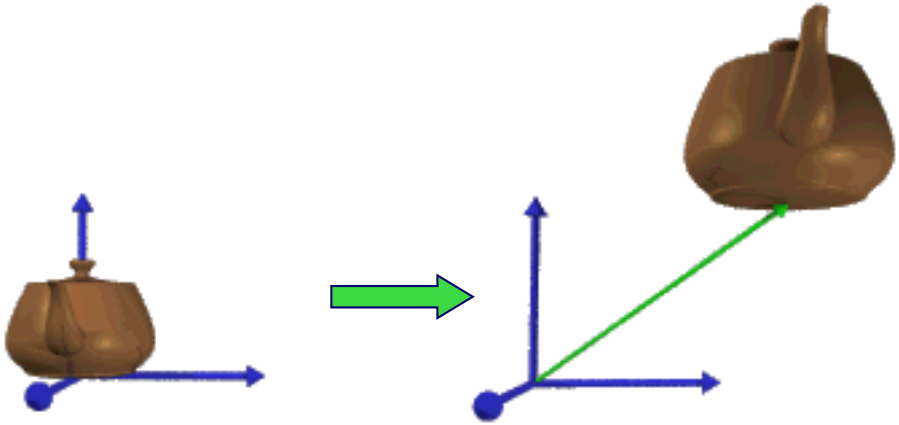
The coordinates of three corresponding points uniquely determine *any* Affine Transform!!



Transformation

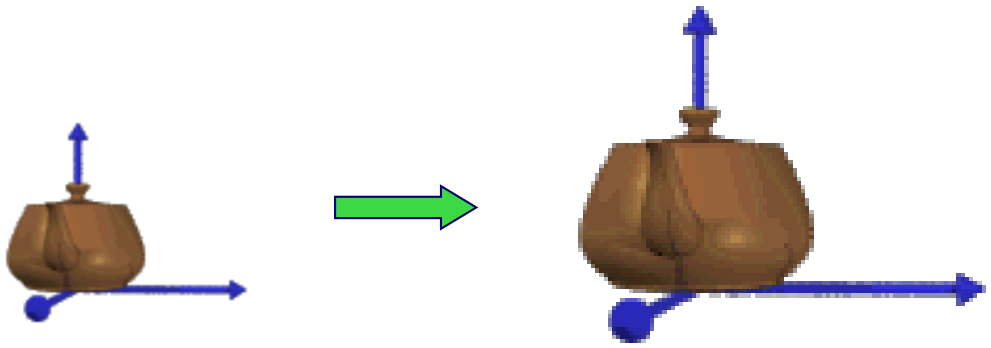
1. 2D Transformation
2. 3D Transformation
3. Viewing Projection

3D Translation



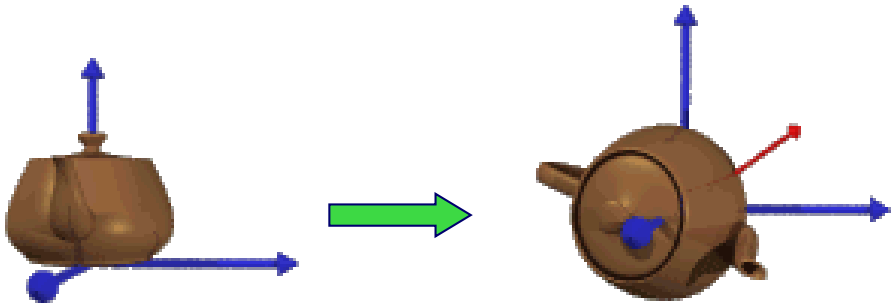
$$\mathbf{T} = \begin{bmatrix} 1 & 0 & 0 & \mathbf{t}_0 \\ 0 & 1 & 0 & \mathbf{t}_1 \\ 0 & 0 & 1 & \mathbf{t}_2 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

3D Scaling



$$\mathbf{S} = \begin{bmatrix} \mathbf{s}_0 & 0 & 0 & 0 \\ 0 & \mathbf{s}_1 & 0 & 0 \\ 0 & 0 & \mathbf{s}_2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

3D Rotation

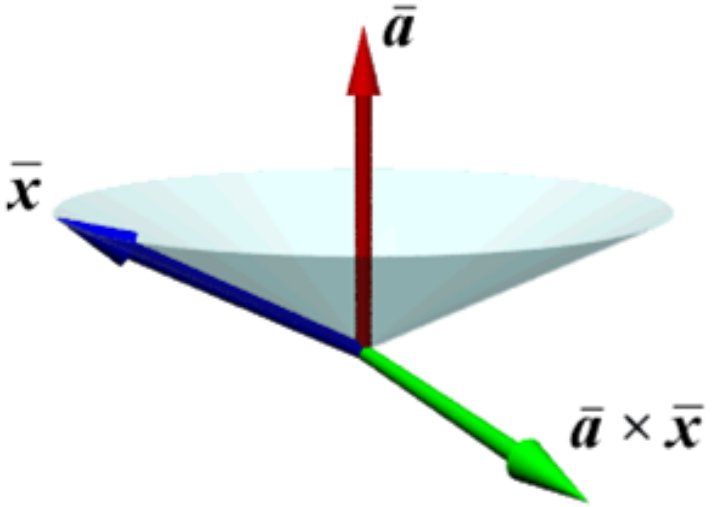


$$R_x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

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

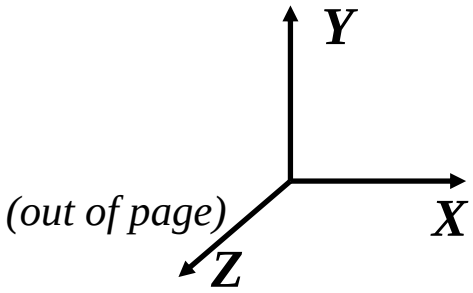
$$R_z = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

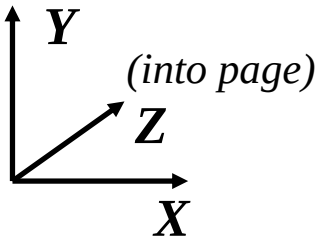
Axis-angle rotation



From 2D to 3D: Preliminary

- Right-handed *vs.* left-handed





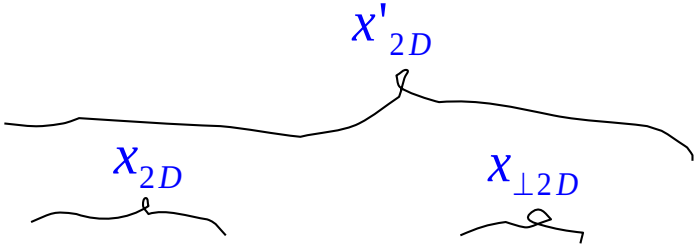
- Z-axis determined from X and Y by cross product: Z=X×Y

$$\mathbf{Z} = \mathbf{X} \times \mathbf{Y} = \begin{bmatrix} X_2Y_3 - X_3Y_2 \\ X_3Y_1 - X_1Y_3 \\ X_1Y_2 - X_2Y_1 \end{bmatrix}$$

- Cross product follows right-hand rule in a right-handed coordinate system, and left-hand rule in left-handed system.

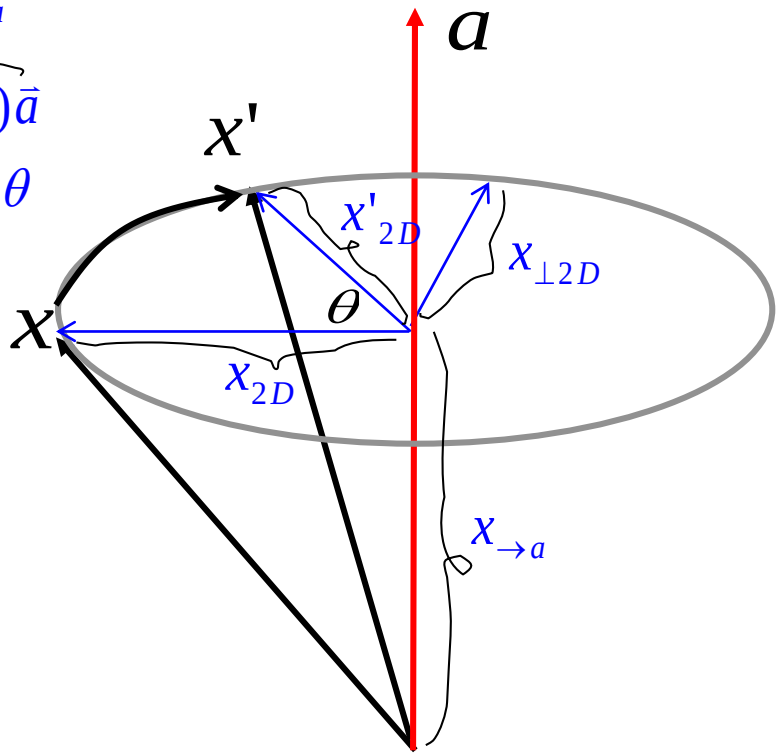
Rotation as Vector Operation

$$x' = x'_{2D} + x_{\to a}$$



$$\begin{aligned}\vec{x}' &= [(\vec{x} - (\vec{x}\cdot\vec{a})\vec{a})\cos\theta] + (\vec{x}\times\vec{a})\sin\theta + (\vec{x}\cdot\vec{a})\vec{a} \\ &= (\cos\theta)\vec{x} + (\vec{x}\cdot\vec{a})\vec{a})(1-\cos\theta) + (\vec{x}\times\vec{a})\sin\theta\end{aligned}$$

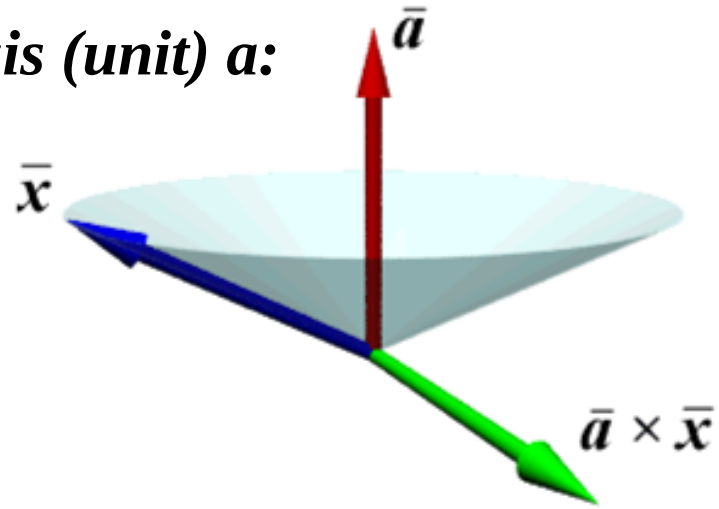
$$\vec{x}' = [(\cos\theta)I + (\vec{a}\vec{a}^{\,T})(1-\cos\theta) + (\vec{a}^{\,*})\sin\theta]\vec{x}$$



Axis-angle rotation

The matrix R for rotation by θ about axis (unit) a:

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



$\boldsymbol{a}\boldsymbol{a}^T$	*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*

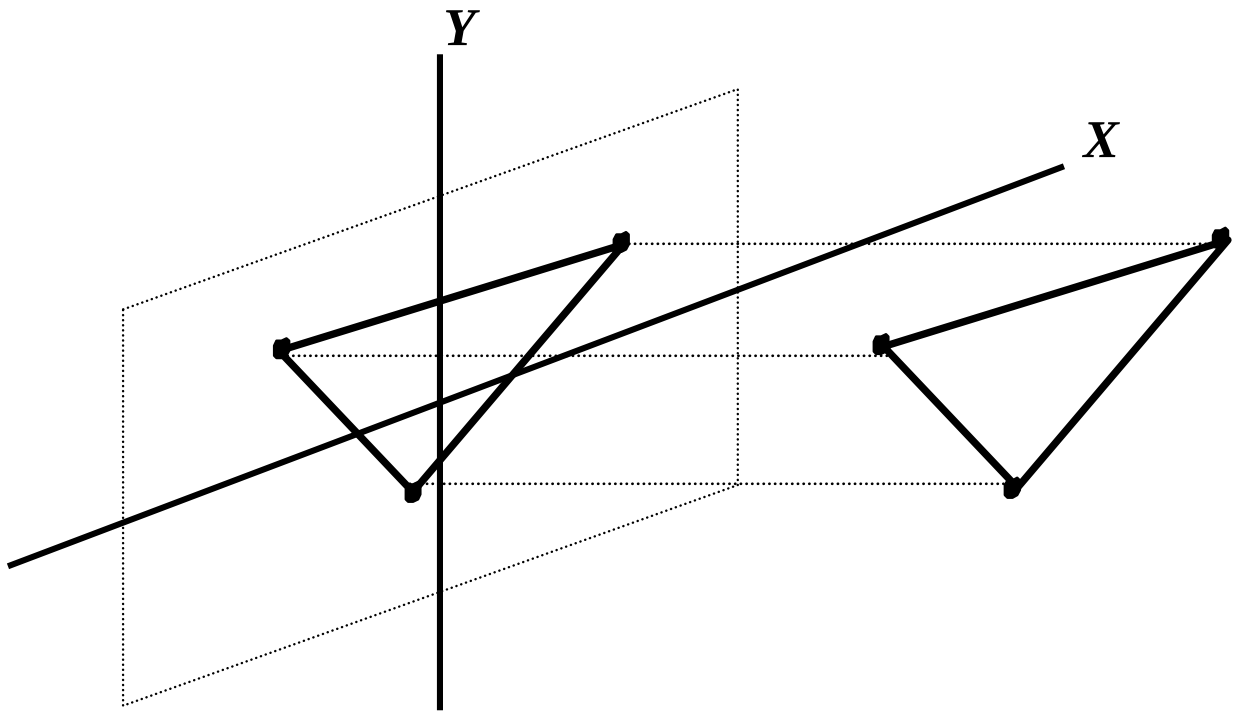
(assumes $\boldsymbol{a}$ *is unit.)*

Transformation

1. 2D Transformation
2. 3D Transformation
3. Viewing Projection

Orthographic Projection

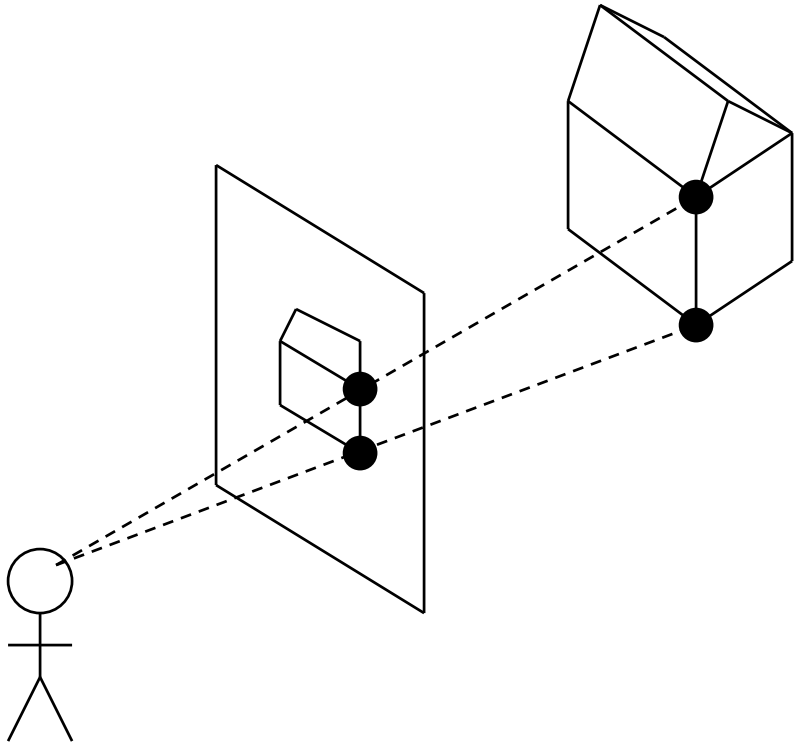
- Throw away Z coordinates
- Get points on the XY plane



Perspective

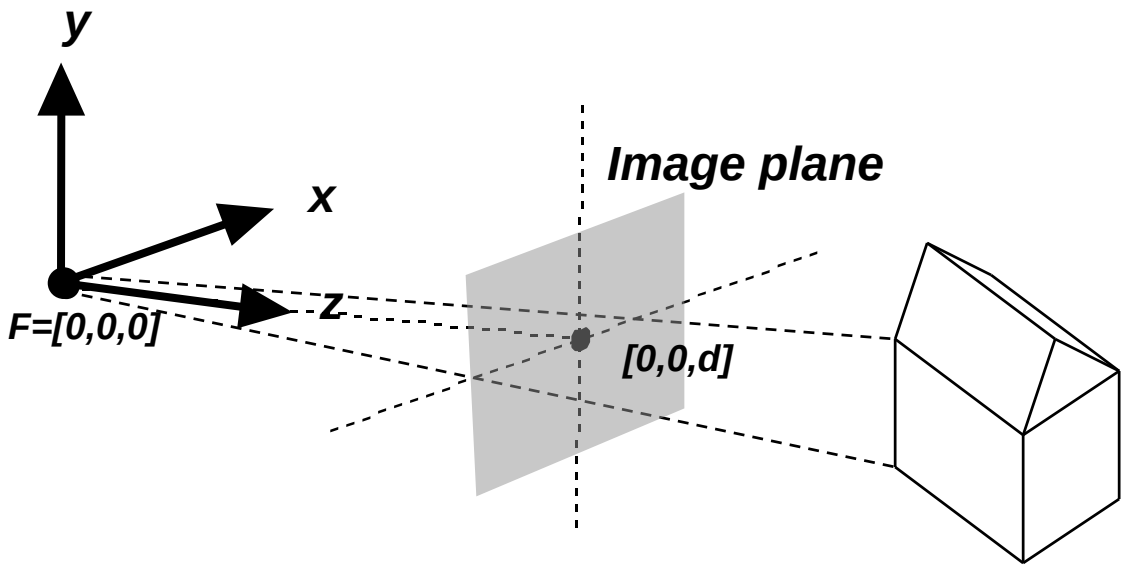
http://www.indiana.edu/~kglowack/athens/

Perspective Projection

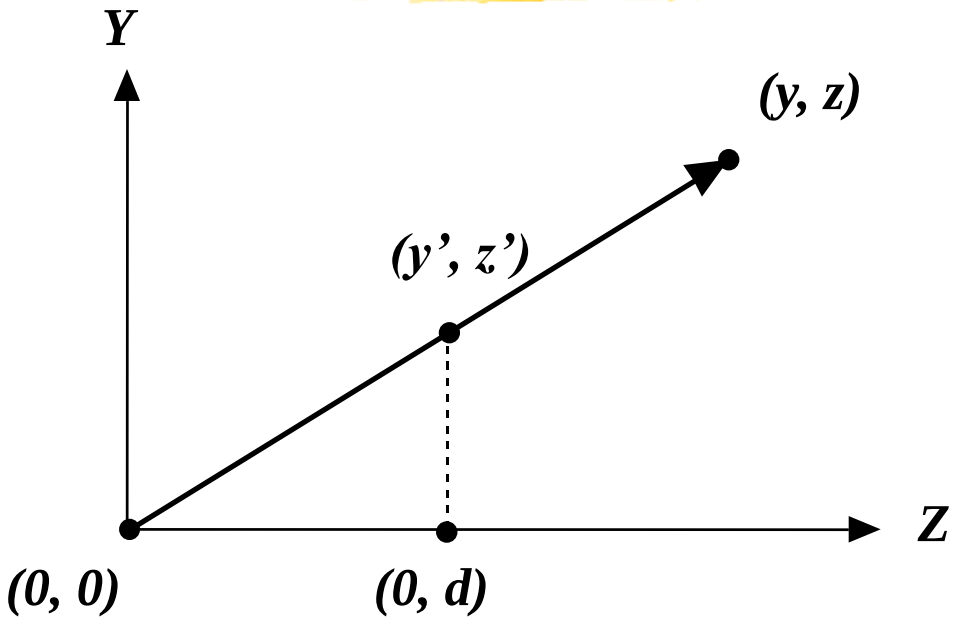


A Simple Perspective Camera

- Canonical case:
  - camera looks along the *z*-axis
  - focal point is the origin
  - image plane is parallel to the *xy*-plane at distance *d*
  - (We call d the focal length, mainly for historical reasons)

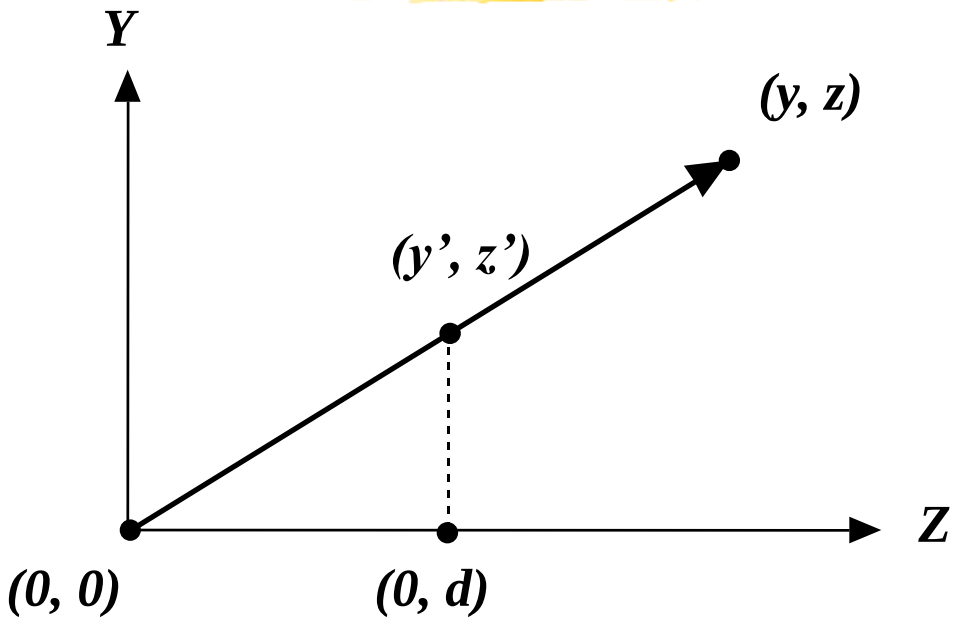


Similar Triangles



- Diagram shows *y*-coordinate, *x*-coordinate is similar

Similar Triangles



$z' = d$

$y'/z' = y/z$

$y'/d = y/z$

$y' = (d/z)*y$

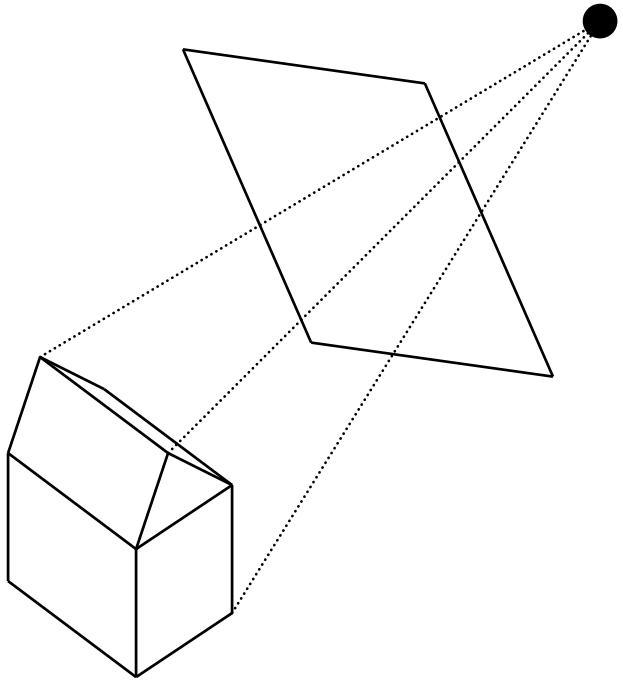
point [x,y,z] projects to [(d/z)x, (d/z)y, d]

A Perspective Projection Matrix

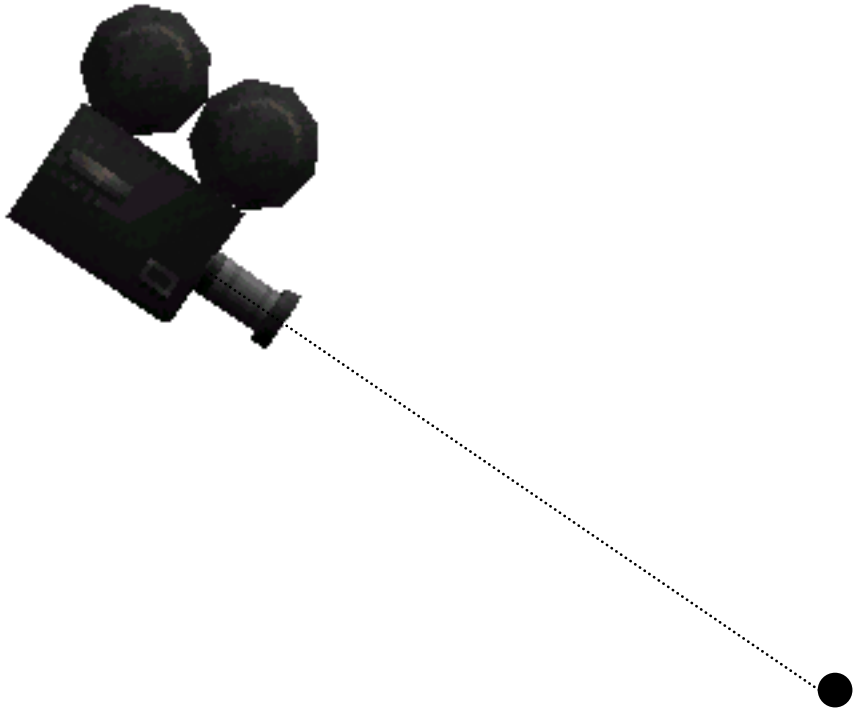
- ***Projection using homogeneous coordinates:***
  - *transform [x, y, z] to [(d/z)x, (d/z)y, d]*

$$\begin{bmatrix} wx' \\ wy' \\ wz' \\ w \end{bmatrix} = \begin{bmatrix} d & 0 & 0 & 0 \\ 0 & d & 0 & 0 \\ 0 & 0 & d & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} dx \\ dy \\ dz \\ z \end{bmatrix} \xrightarrow{\frac{1}{w}} \begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} \frac{d}{z}x \\ \frac{d}{z}y \\ d \end{bmatrix}$$

Camera Position and Orientation

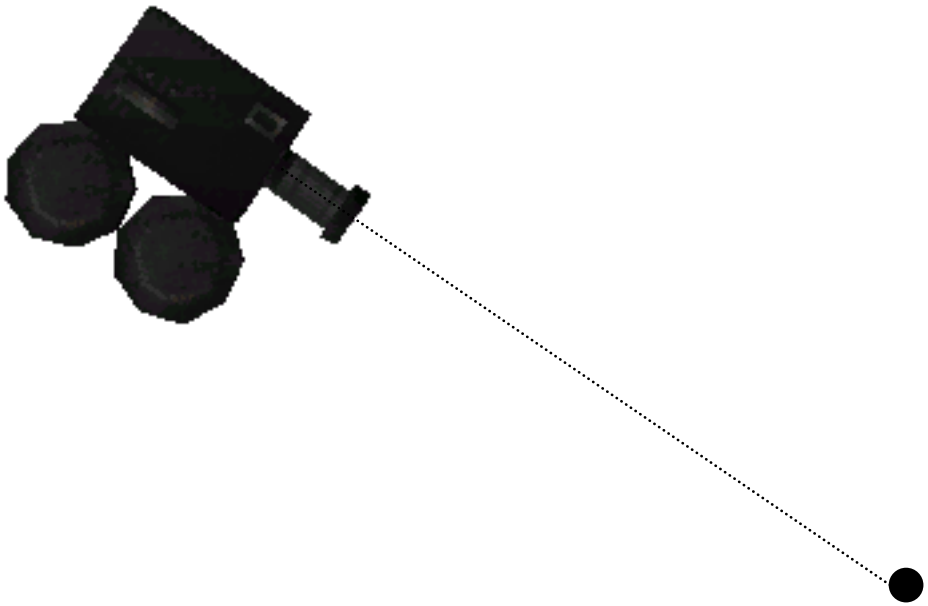


LookFrom And LookAt

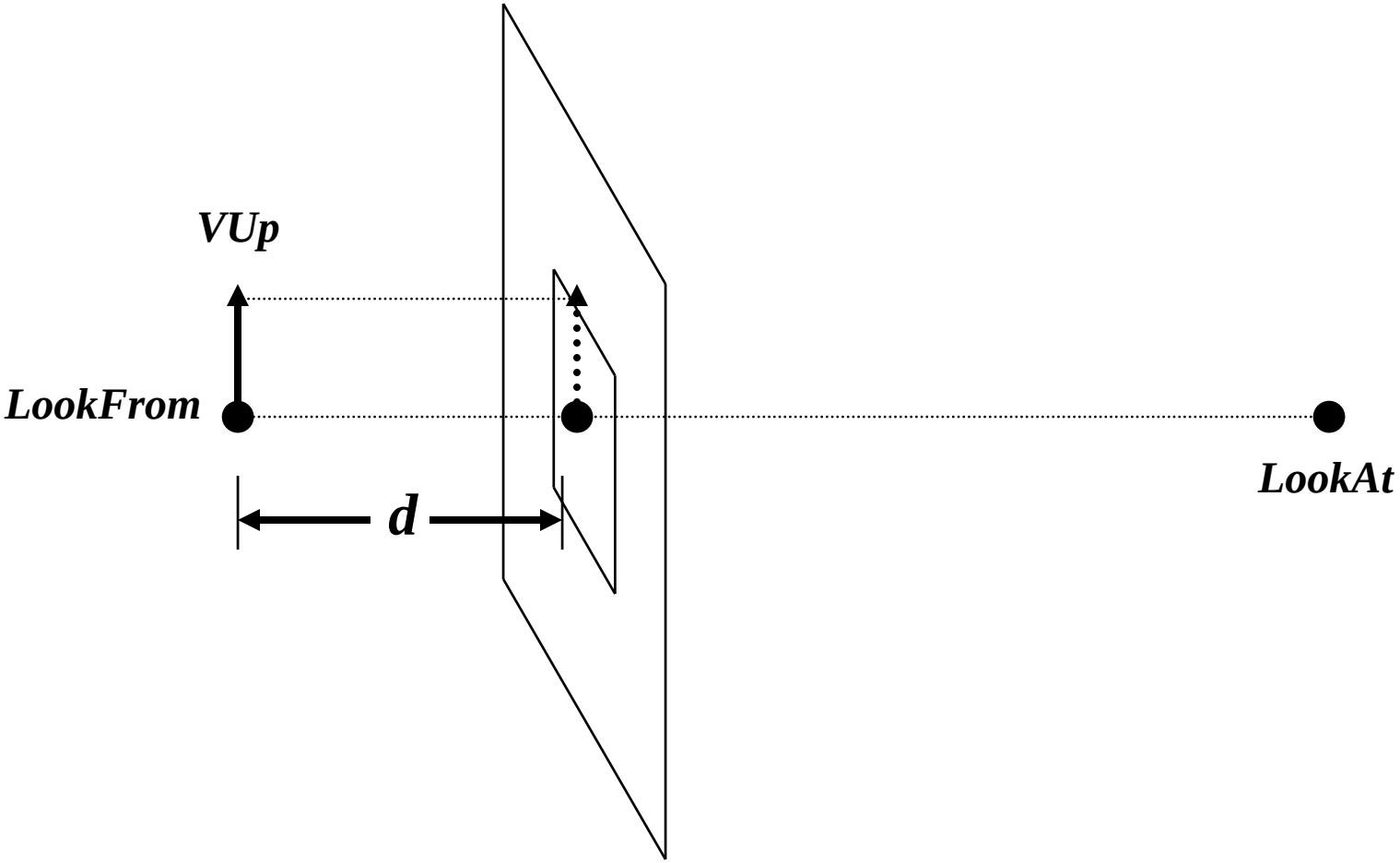


Is This Enough?

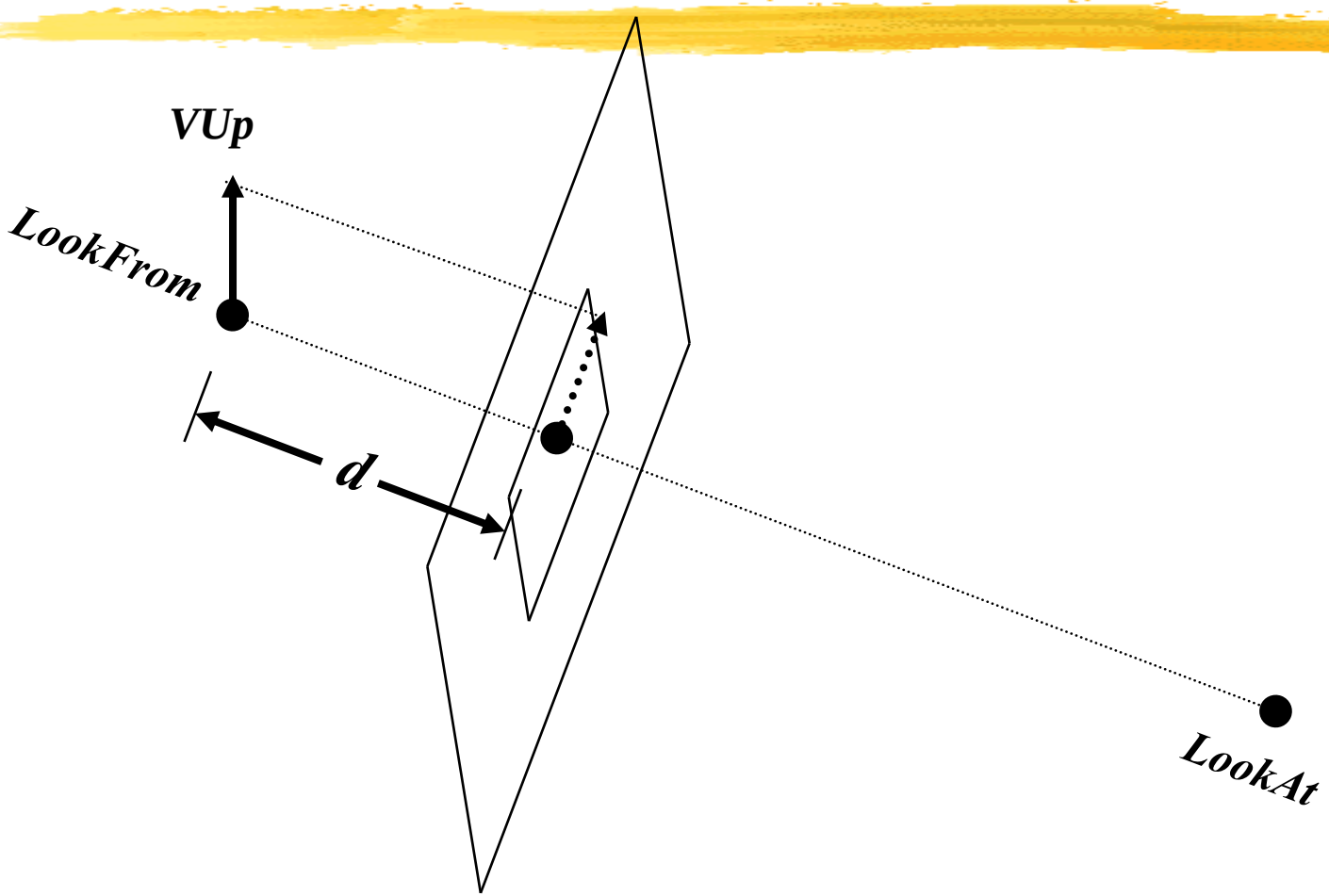
LookFrom And LookAt



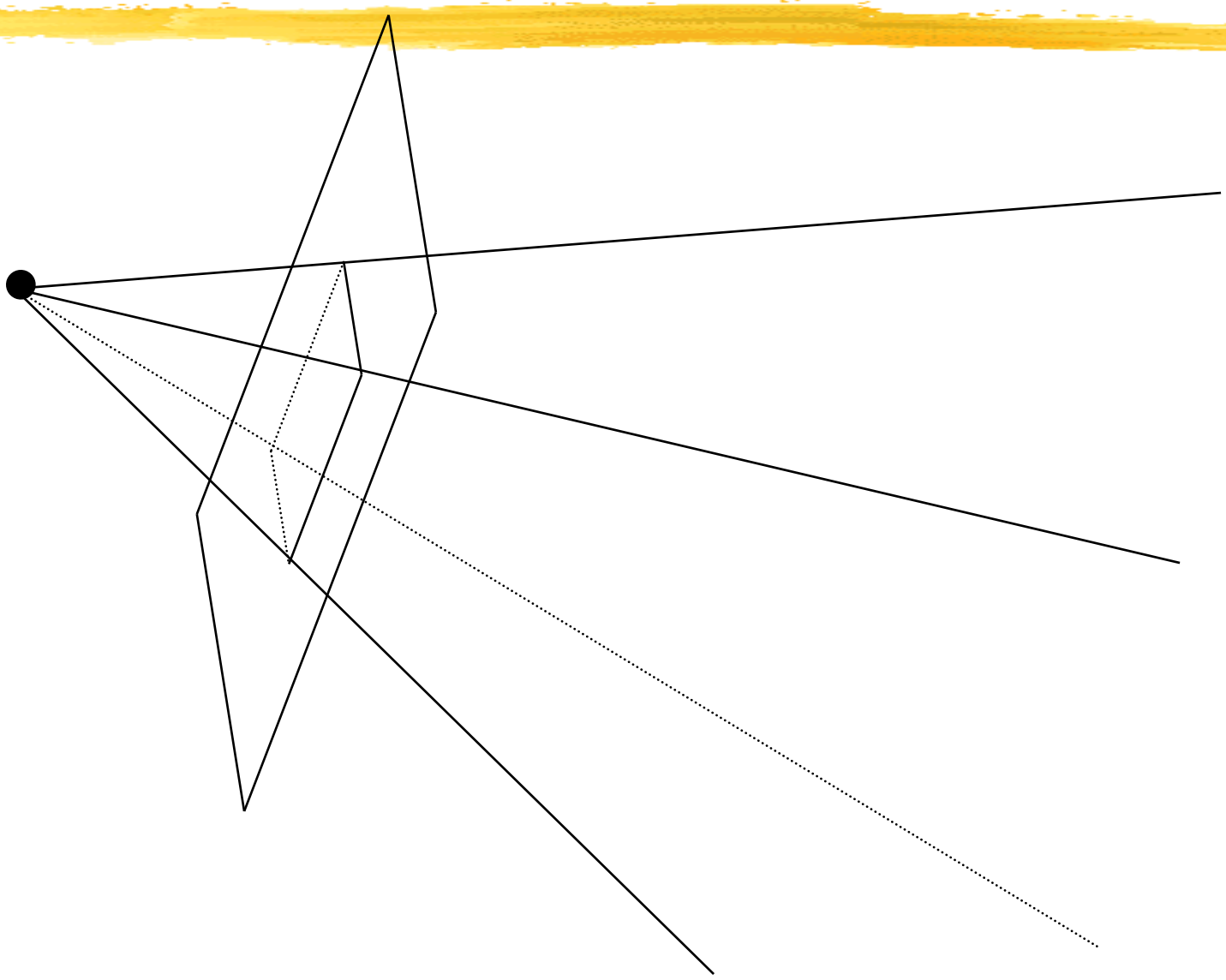
Complete Camera Specification



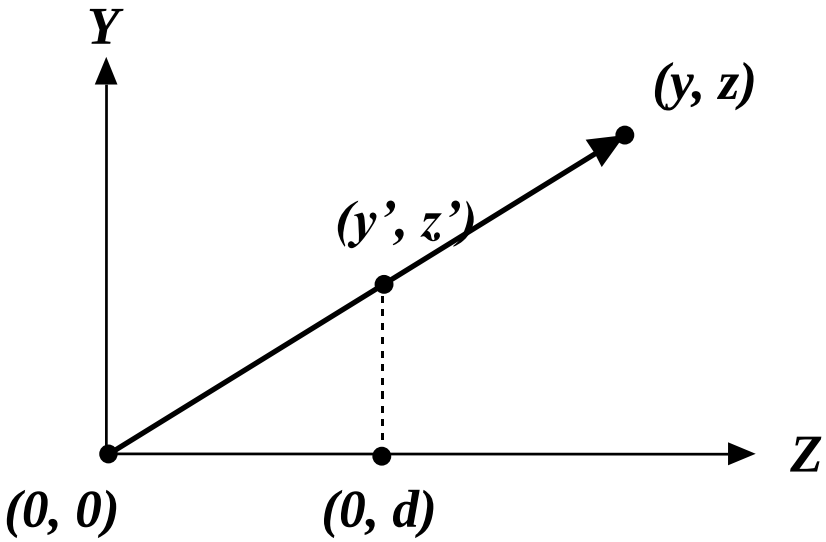
Complete Camera Specification

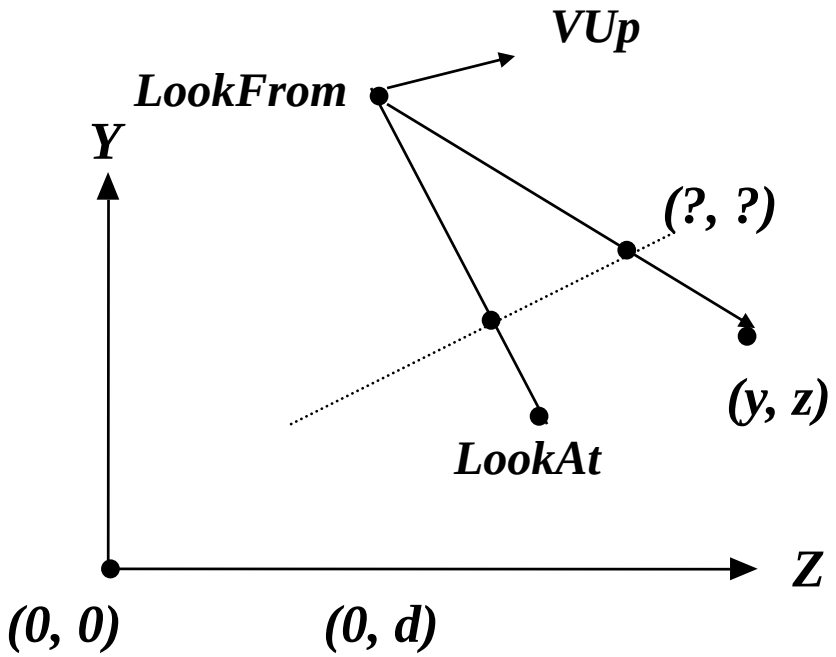


Viewing Volume



Rendering from any camera position

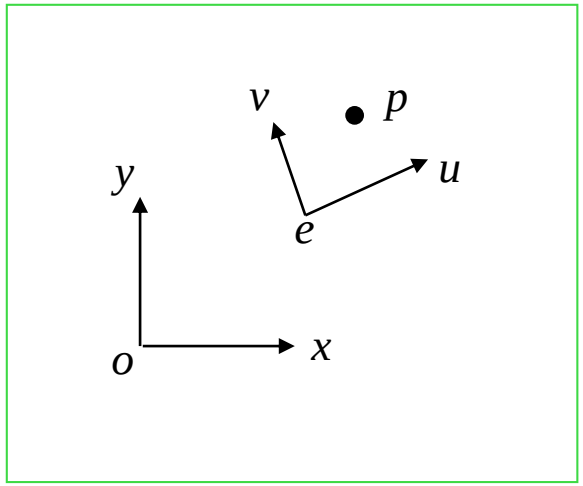




Coordinate Transformation

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

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

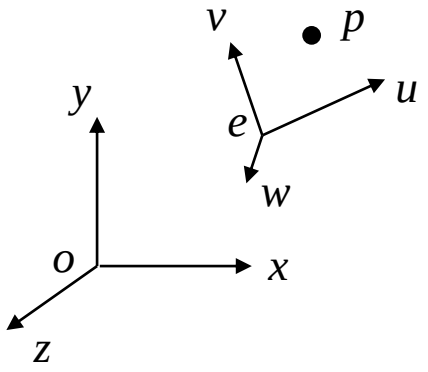


$$\begin{bmatrix} p_x \\ p_y \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & e_x \\ 0 & 1 & e_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u_x & v_x & 0 \\ u_y & v_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p_u \\ p_v \\ 1 \end{bmatrix}$$

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

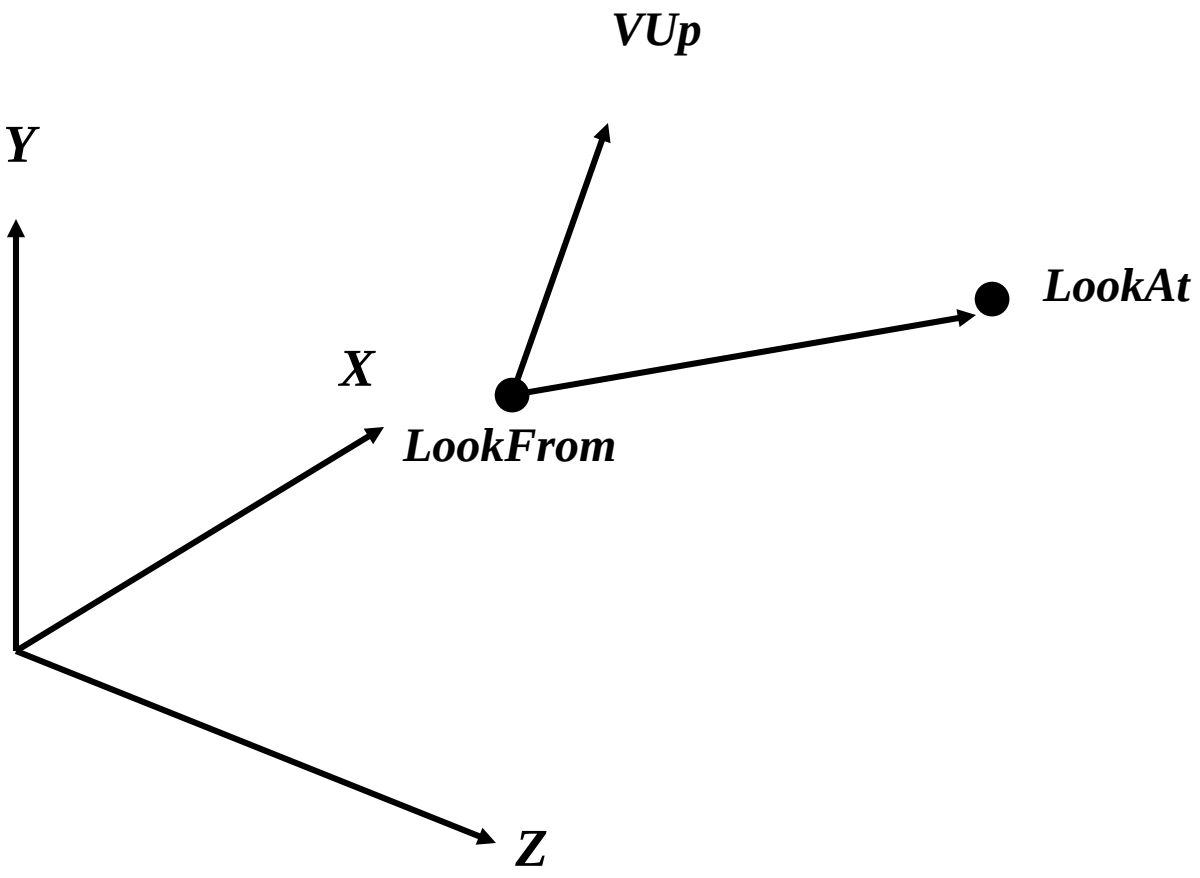
Coordinate Transformation

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

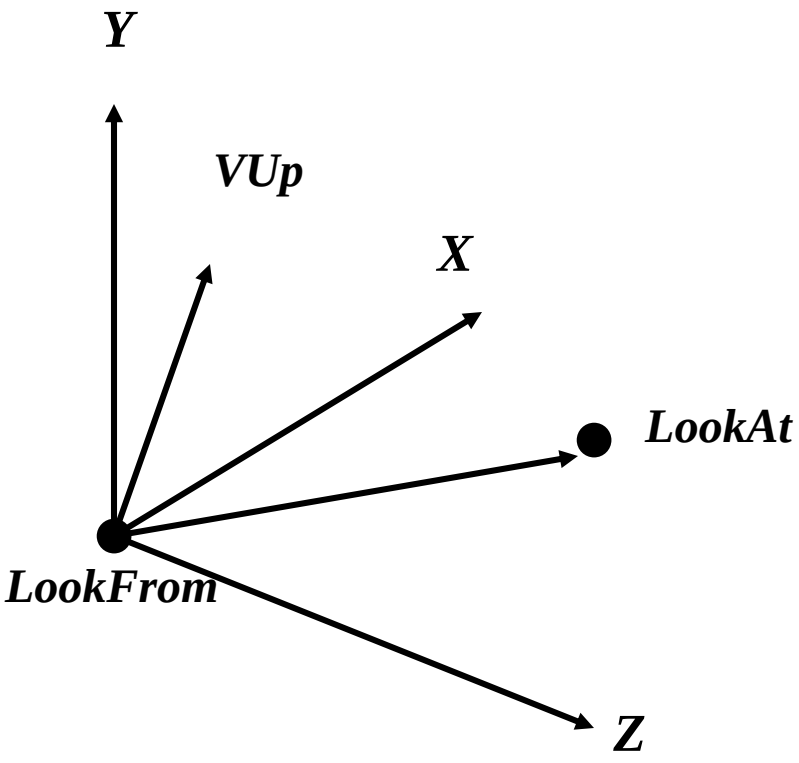


$$\begin{bmatrix} p_u \\ p_v \\ p_w \\ 1 \end{bmatrix} = \begin{bmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ w_x & w_y & w_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -e_x \\ 0 & 1 & 0 & -e_y \\ 0 & 0 & 1 & -e_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p_x \\ p_y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

Viewing Transformations

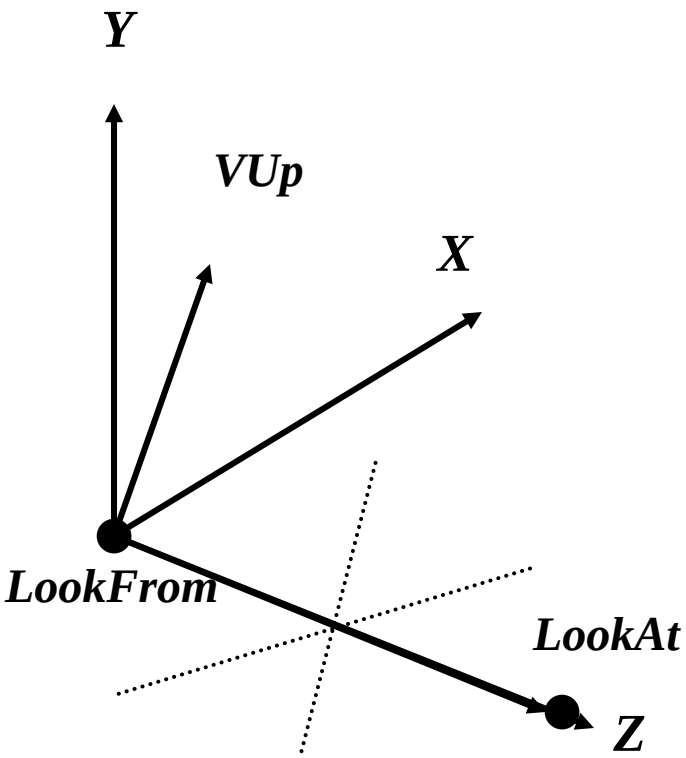


Viewing Transformations



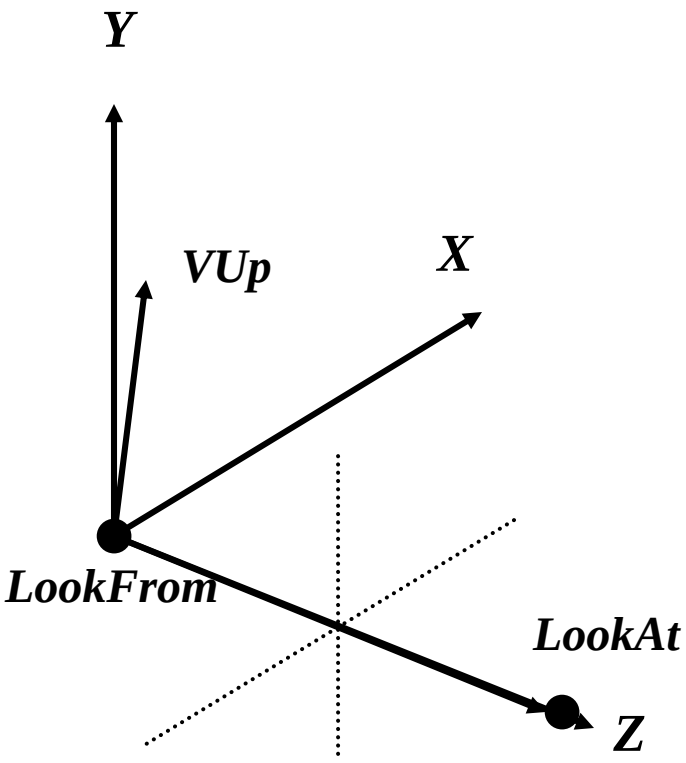
Translate LookFrom to origin

Viewing Transformations



Rotate LookAt to Z axis (axis-angle rotation)

Viewing Transformations



Rotate about Z to get the projection of Vup parallel to the Y axis

Implementation

Implementing the *lookat/lookfrom/vup* viewing scheme

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

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

» v = (*lookat-lookfrom*) (normalized) and z = [0,0,1]

» rotation axis: a = (v×z)/|v×z|

» rotation angle: $\cos\theta = \mathbf{v}\cdot\mathbf{z}$ and $\sin\theta = |\mathbf{v}\times\mathbf{z}|$

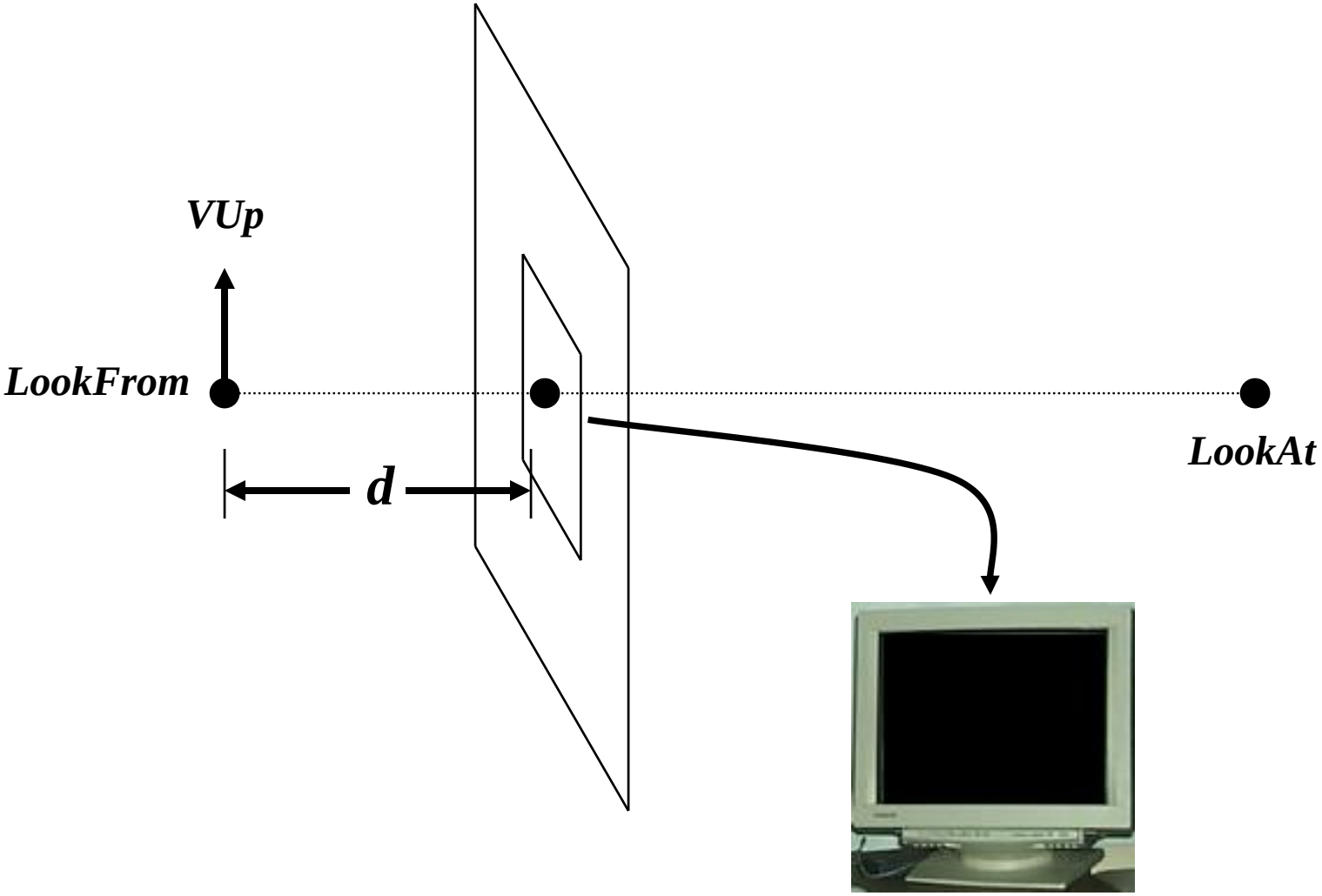
$$\mathbf{R} = \mathbf{a}\mathbf{a}^T + (\mathbf{v}\cdot\mathbf{z})(\mathbf{I} - \mathbf{a}\mathbf{a}^T) + |\mathbf{v}\times\mathbf{z}|\mathbf{a}^* \quad \textit{where} \quad \mathbf{a}^* = \begin{bmatrix} 0 & -a_z & a_y \\ a_z & 0 & -a_x \\ -a_y & a_x & 0 \end{bmatrix}$$

or: glRotate(θ, a_x, a_y, a_z)

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

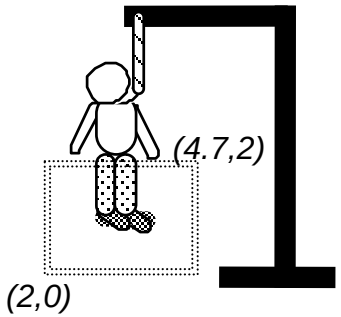
» *watch out if vup is along the z-axis*

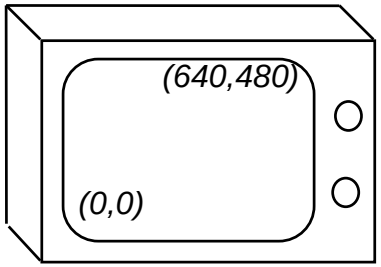
Screen Coordinates

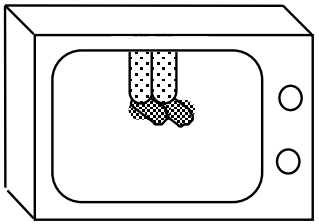


Viewport Transformations

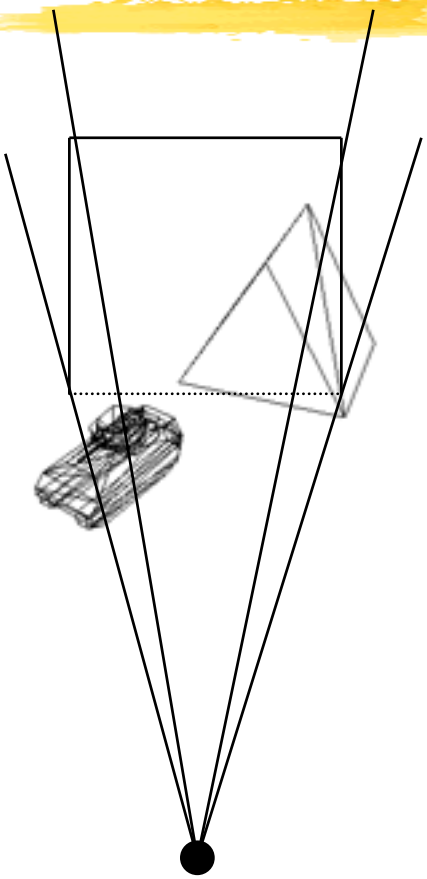
- A transformation maps the visible (model) world onto screen or window coordinates
- In OpenGL a viewport transformation, e.g. glOrtho(), defines what part of the world is mapped in standard "Normalized Device Coordinates" ((-1,-1) to (1,1))
- The viewpoint transformation maps NDC into actual window, pixel coordinates
  - by default this fills the window
  - otherwise use glViewport

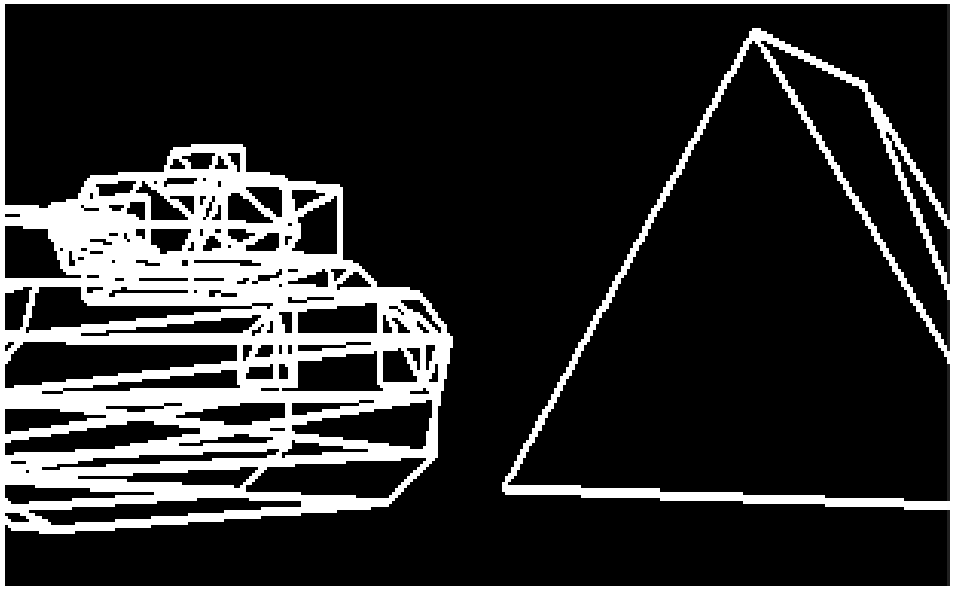




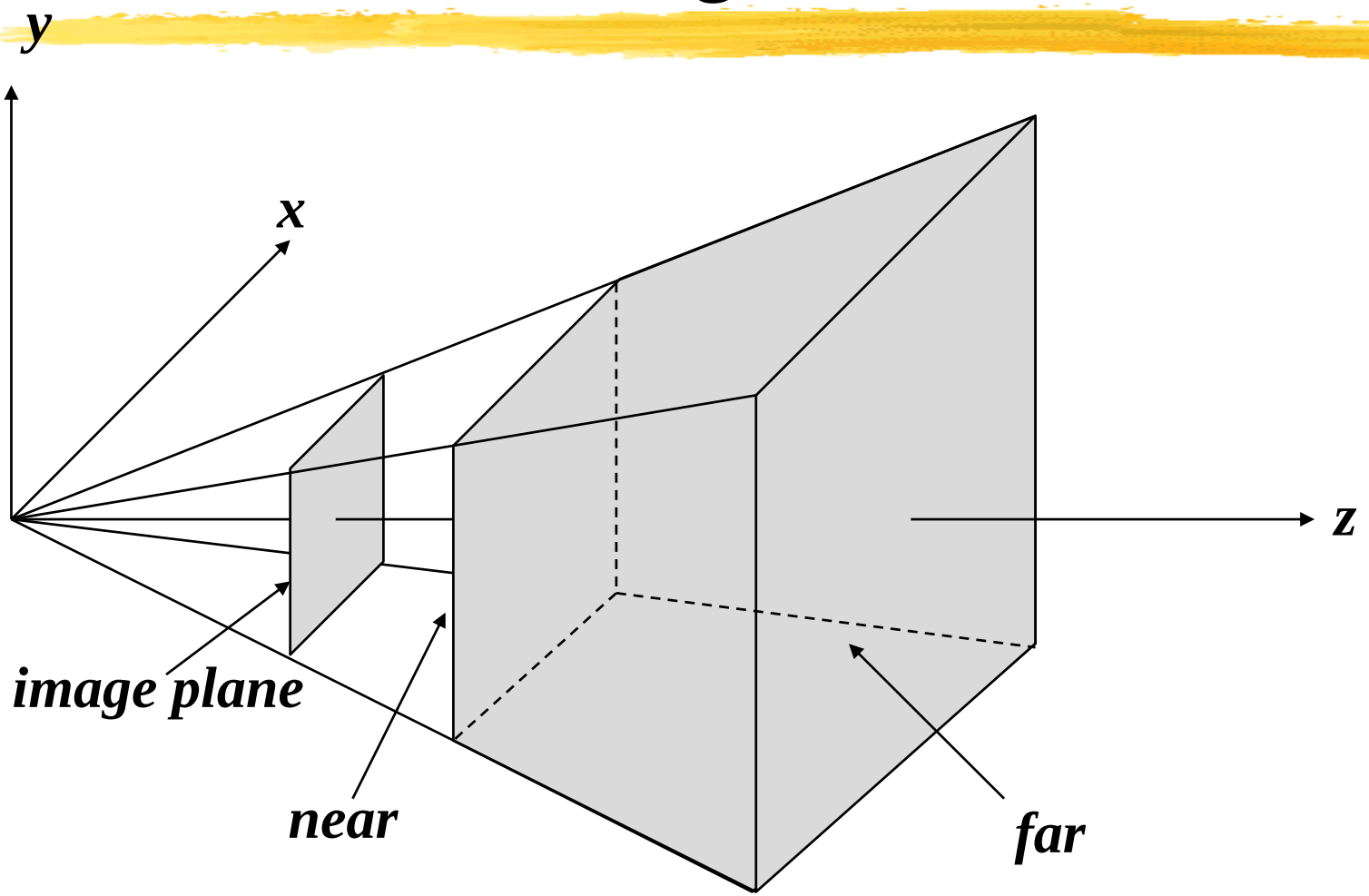


Clipping



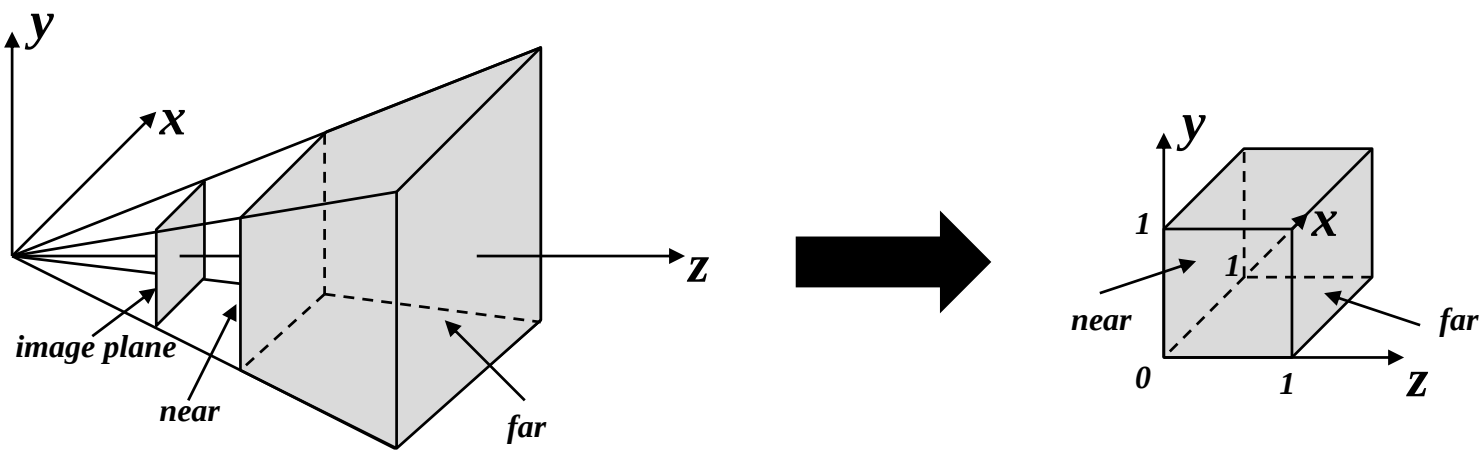


The Viewing Frustum



Normalizing the Viewing Frustum

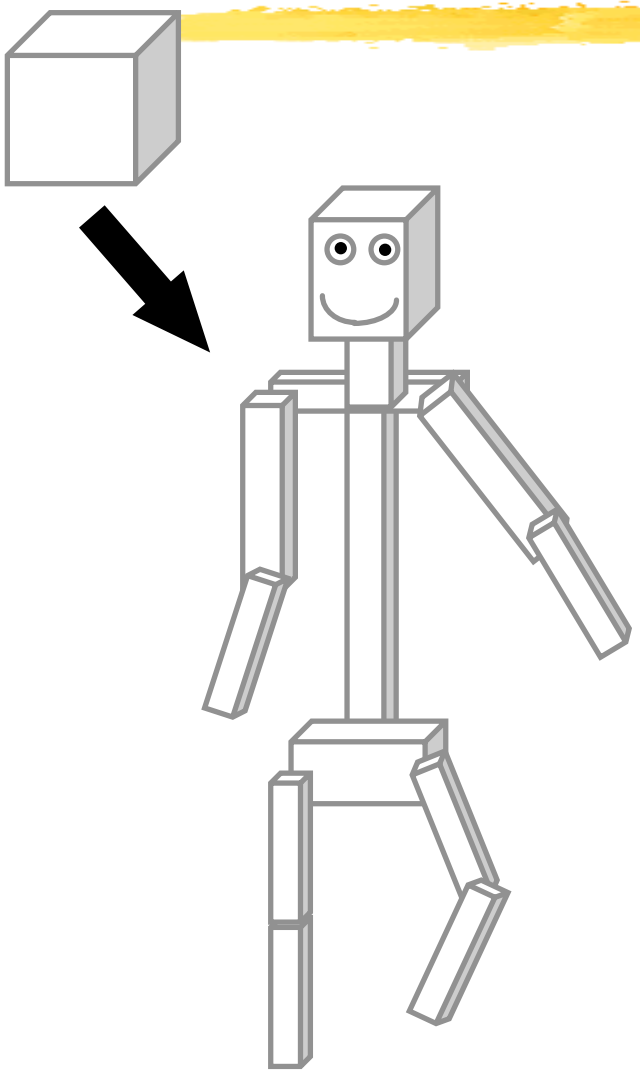
- Transform frustum to a cube before clipping



- *Converts perspective frustum to orthographic frustum*
- *Very similar to our perspective transformation – just another matrix*

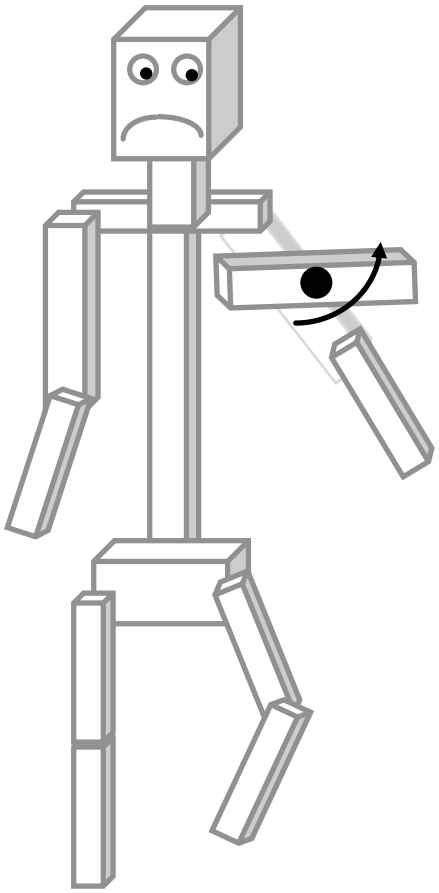
Model and Transformation Hierarchy

How to Model a Stick Person



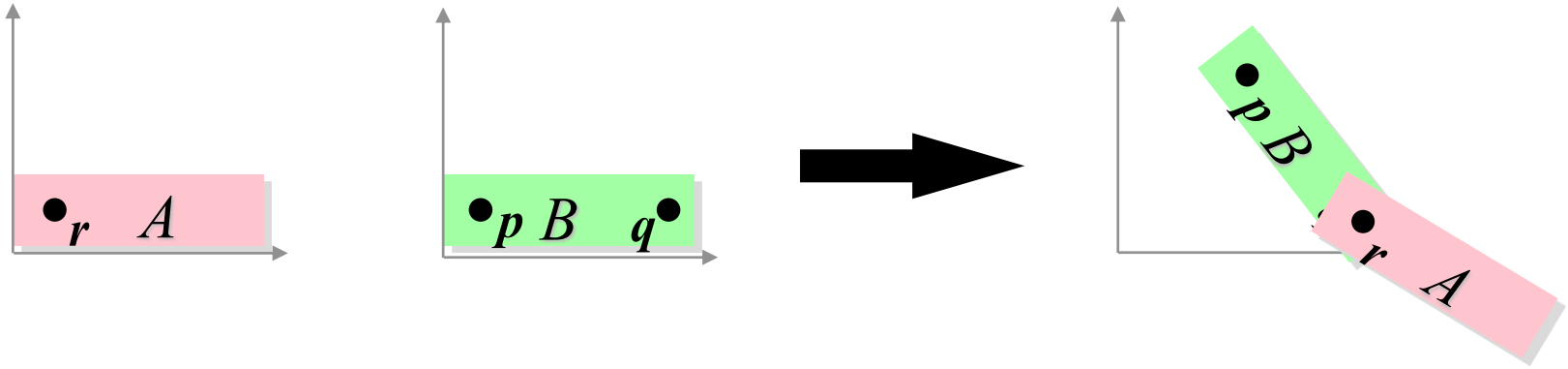
- Make a stick person out of cubes
- Just translate, rotate, and scale each one to get the right size, shape, position, and orientation.
- Looks great, until you try to make it move.

The Right Control Knobs



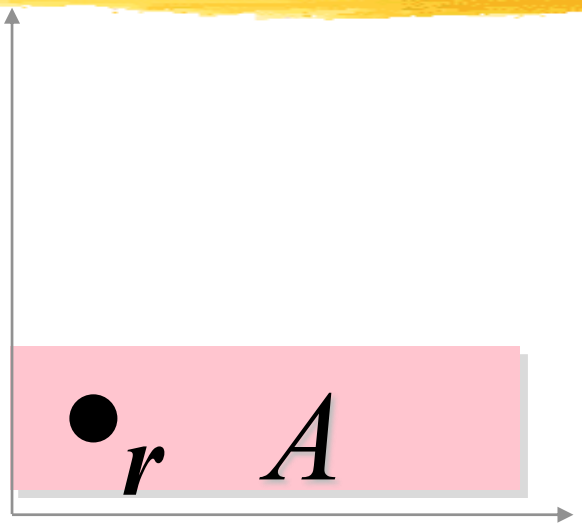
- As soon as you want to change something, the model *likely* falls apart
- Reason: the thing you're modeling is *constrained* but your model doesn't know it
- Wanted:
  - some sort of representation of *structure*
  - *Control knob*
- This kind of control knob is convenient for static models, and *vital*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

Making an Articulated Model



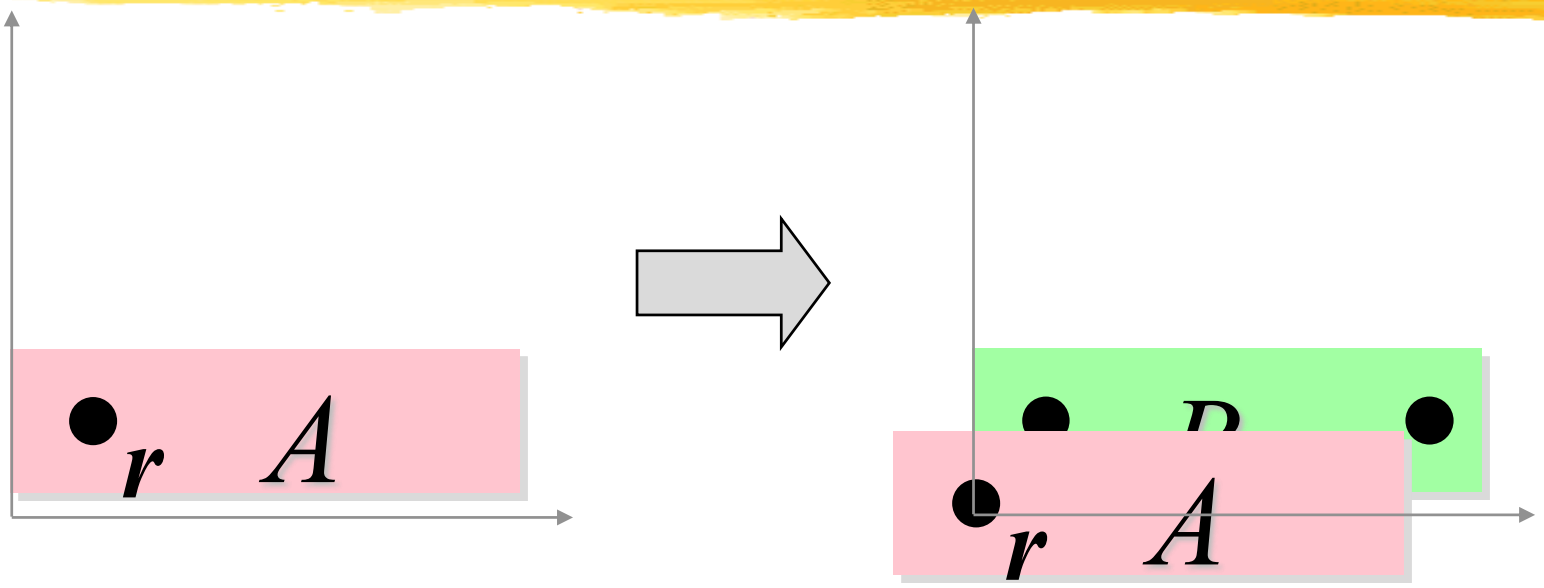
- A minimal 2-D jointed object:
  - Two pieces, *A* ("forearm") and *B* ("upper arm")
  - Attach point q on *B* to point r on *A* ("elbow")
  - Desired control knobs:
    - » u: shoulder angle (*A* and *B* rotate together about p)
    - » v: elbow angle (*A* rotates about r, which stays attached to p)

Making an Arm, step 1



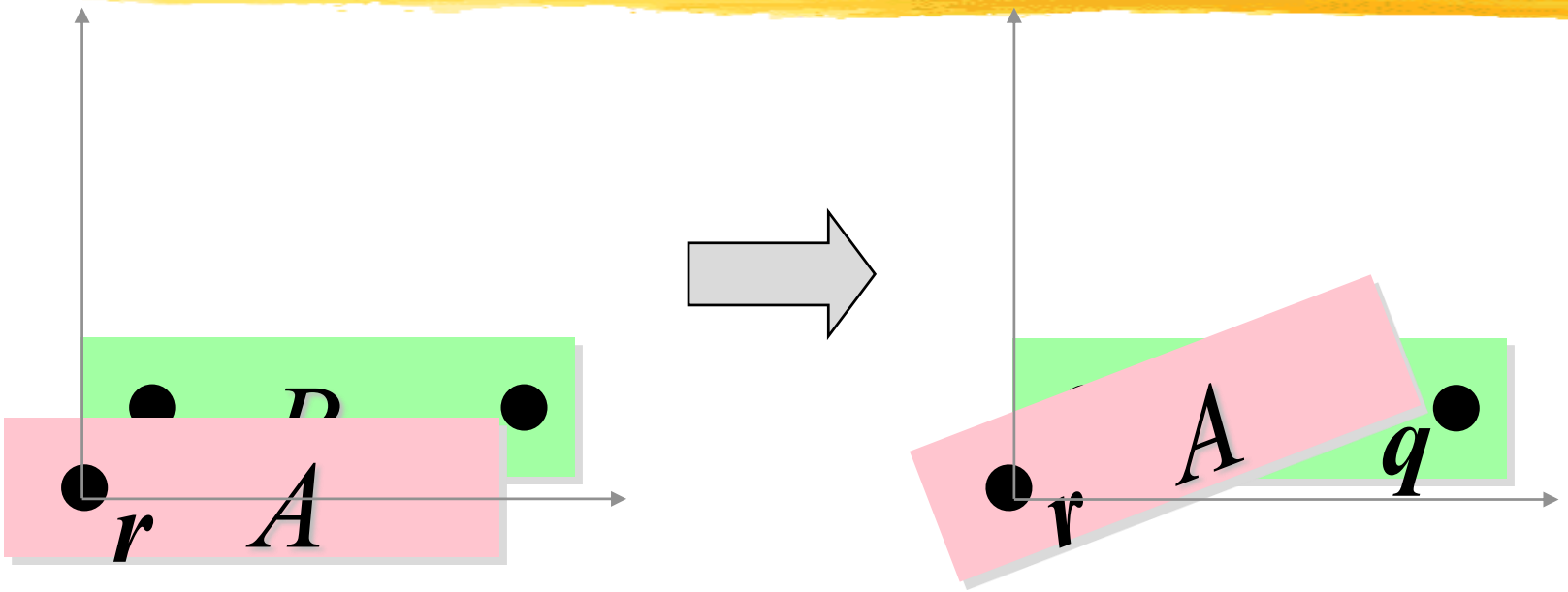
- Start with A and B in their untransformed configurations (B is hiding behind A)
- First apply a series of transformations to A, leaving B where it is…

Making an Arm, step 2



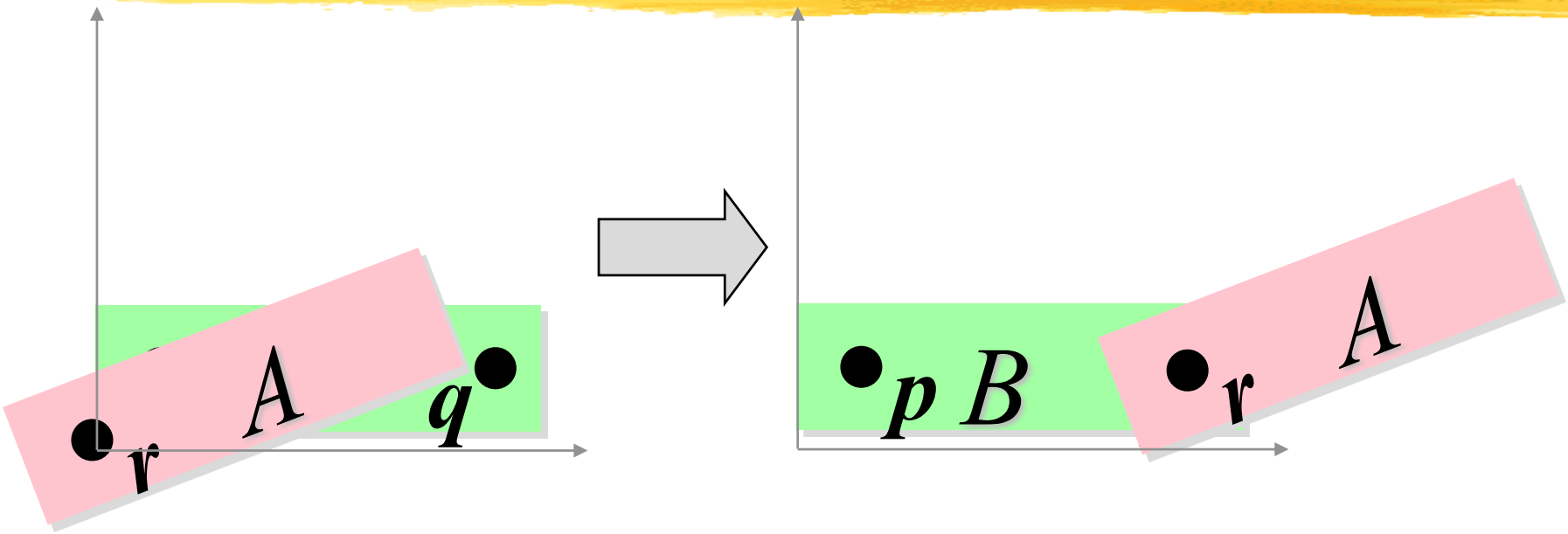
- Translate by -r, bringing r to the origin
- You can now see *B* peeking out from behind *A*

Making an Arm, step 3



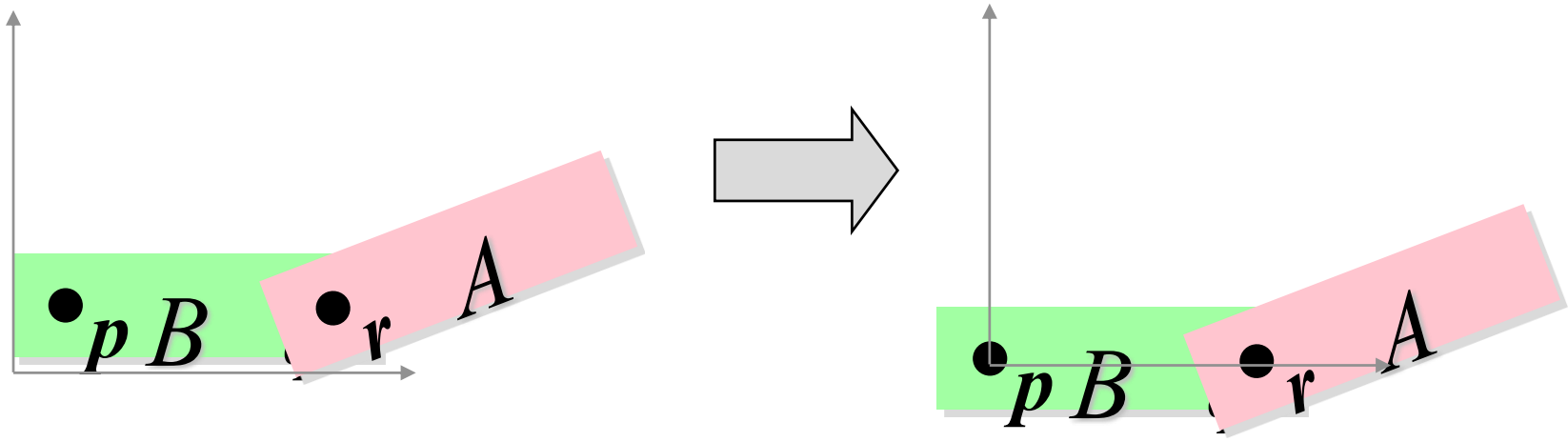
- Next, we rotate A by v (the "elbow" angle)

Making an Arm, step 4



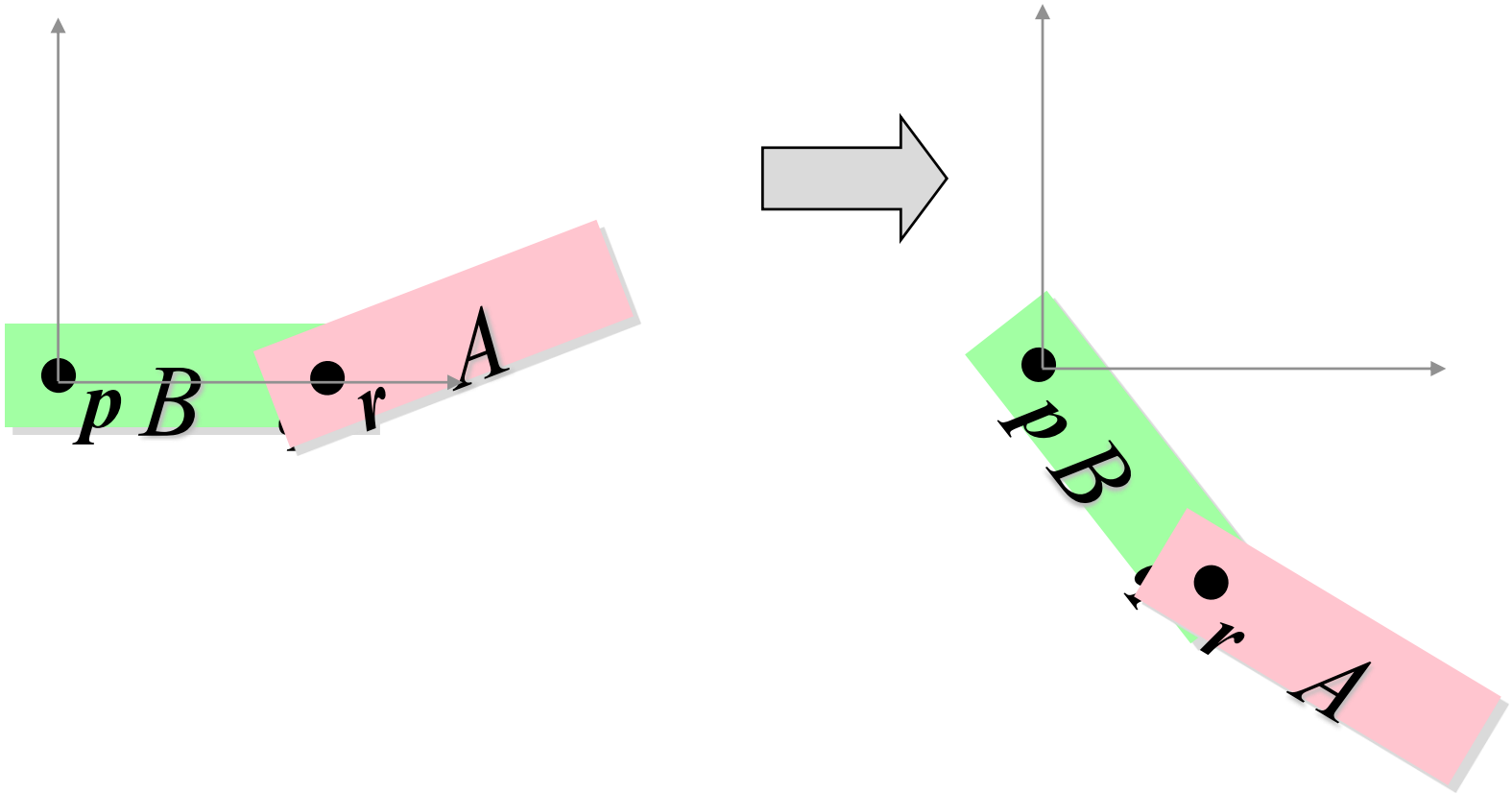
- Translate *A* by q, bringing r and q together to form the elbow joint
- We can regard q as the origin of the *elbow coordinate system*, and regard A as being in this coordinate system.

Making an Arm, step 5



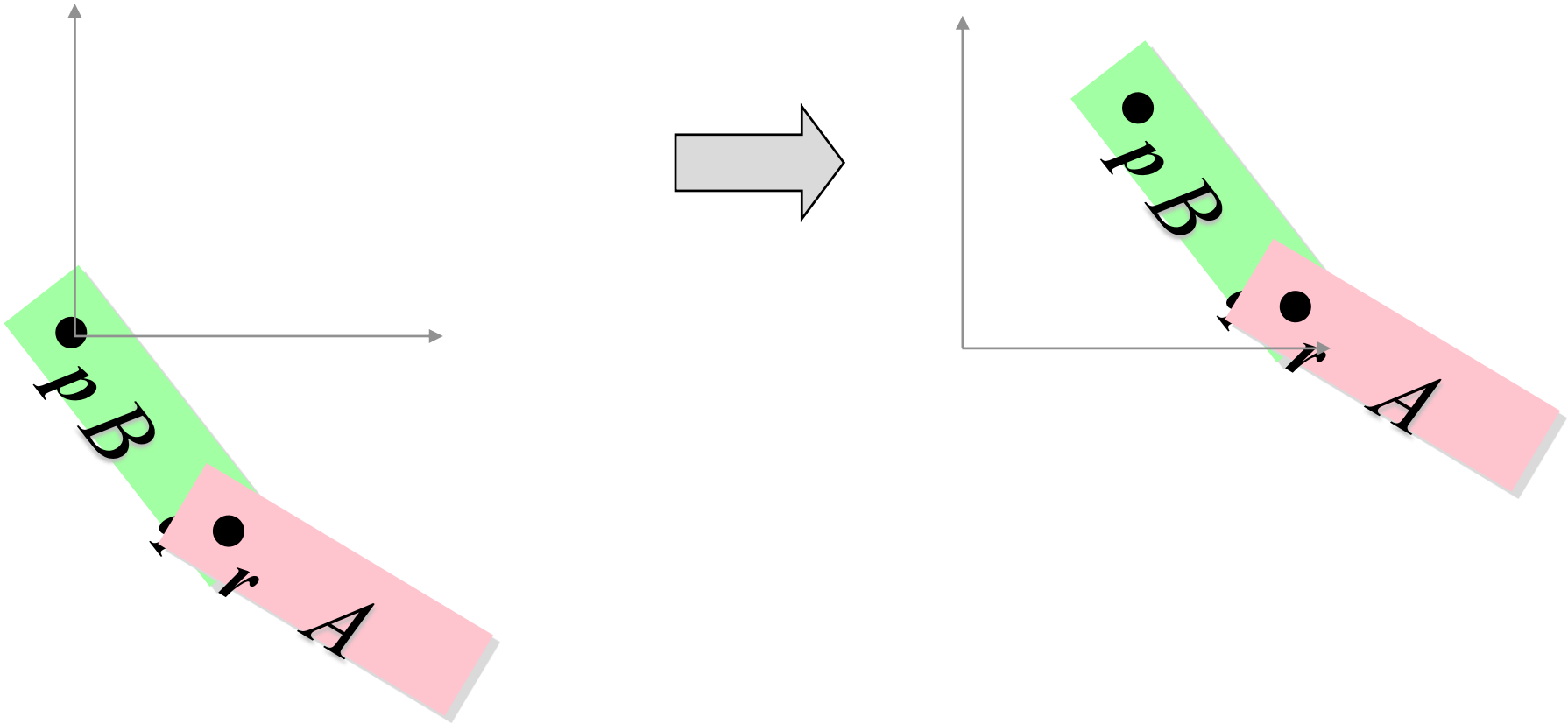
- From now on, each transformation applies to *both* A and B (This is important!)
- First, translate by -p, bringing p to the origin
- A and B both move together, so the elbow doesn't separate!

Making an Arm, step 6



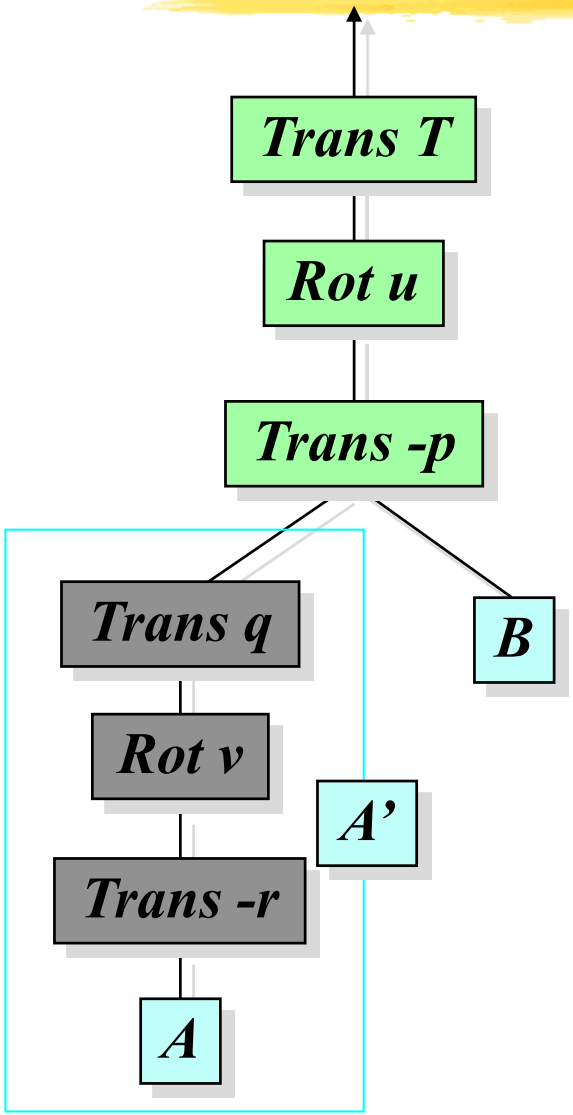
- Then, we rotate by u, the "shoulder" angle
- Again, A and B rotate together

Making an Arm, step 7



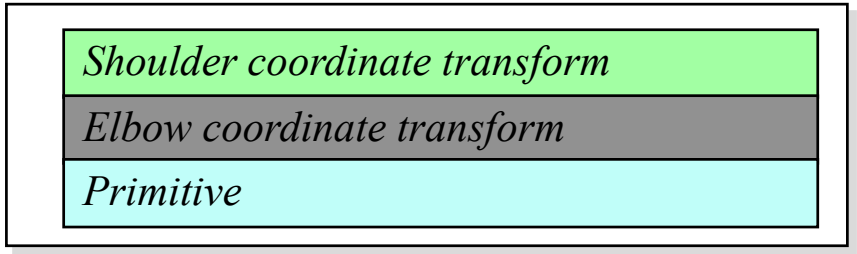
- Finally, translate by T, bringing the arm where we want it
- p is at origin of *shoulder coordinate system*

Transformation Hierarchies

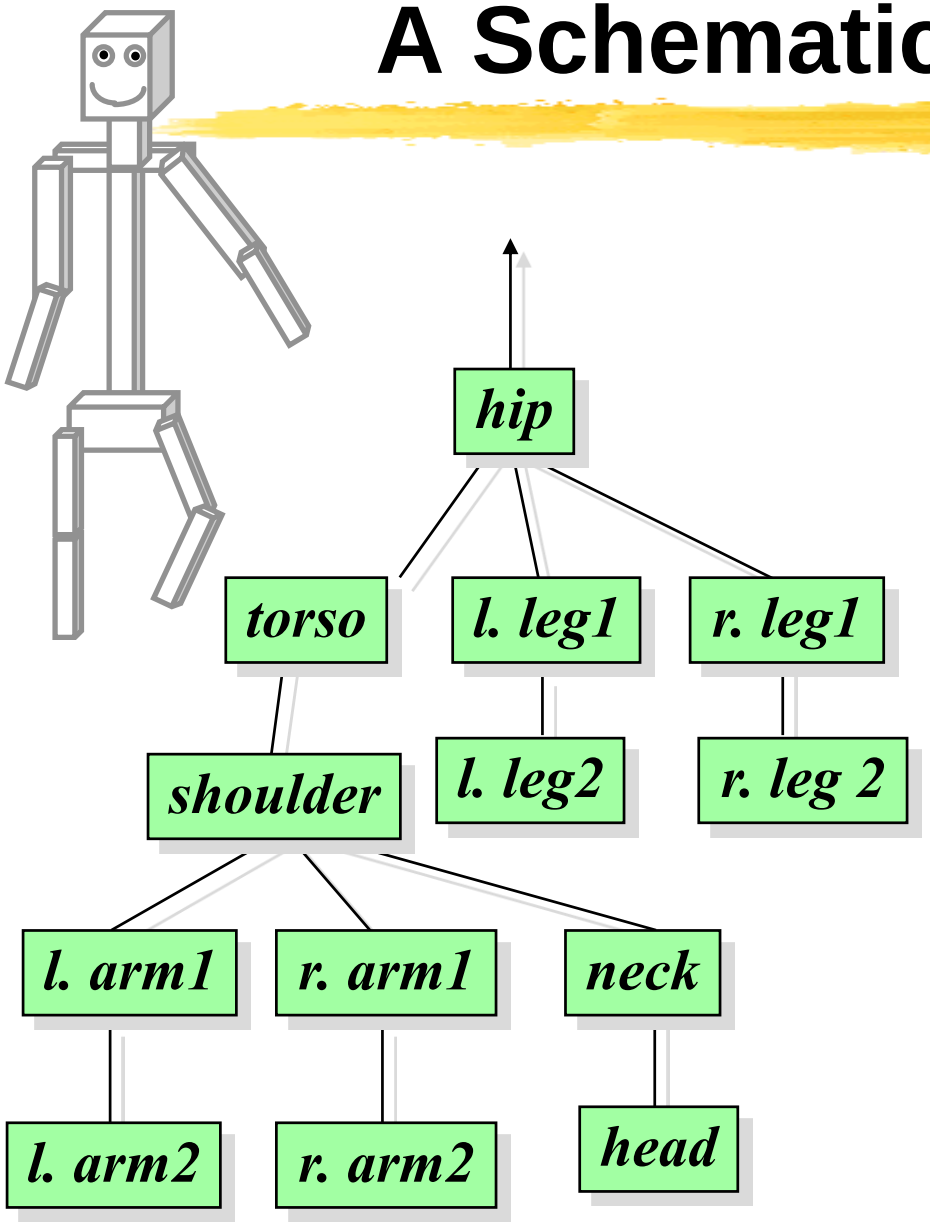


Another point of view:

- 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'

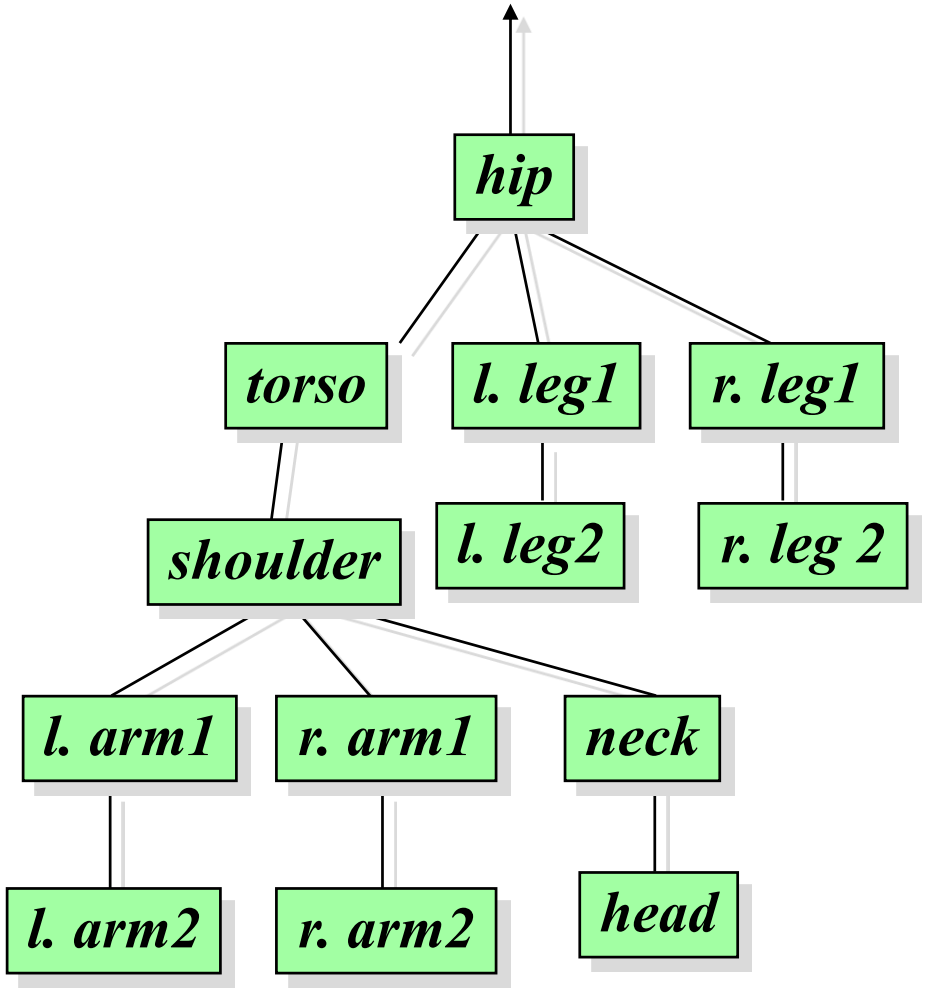


A Schematic Humanoid



- 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
- 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)
- Nuts-and-bolts issues:
  - What kind of nodes should I put in my hierarchy?
  - What kind of interface should I use to construct and edit hierarchical models?

The Matrix Stack

- *Idea of Matrix Stack:*
  - *LIFO 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

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

set up parallel projection matrix

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.)

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();
```