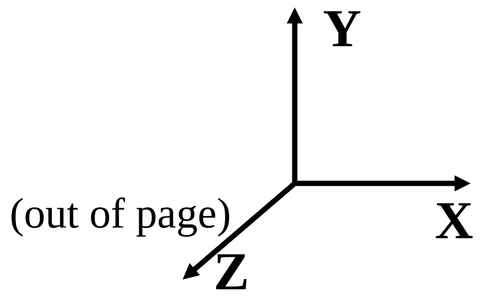
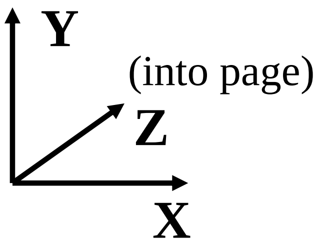
From 2D to 3D: Preliminary

- Right-handed *vs.* left-handed



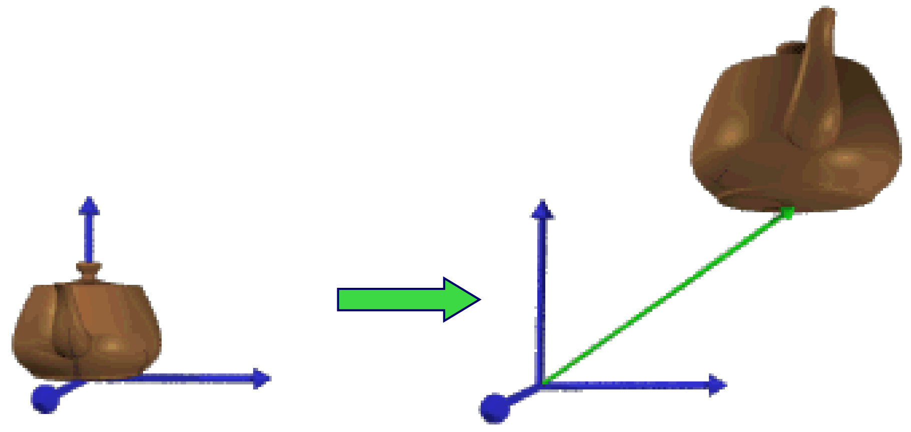


- Z-axis determined from X and Y by cross product: Z=X×Y

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

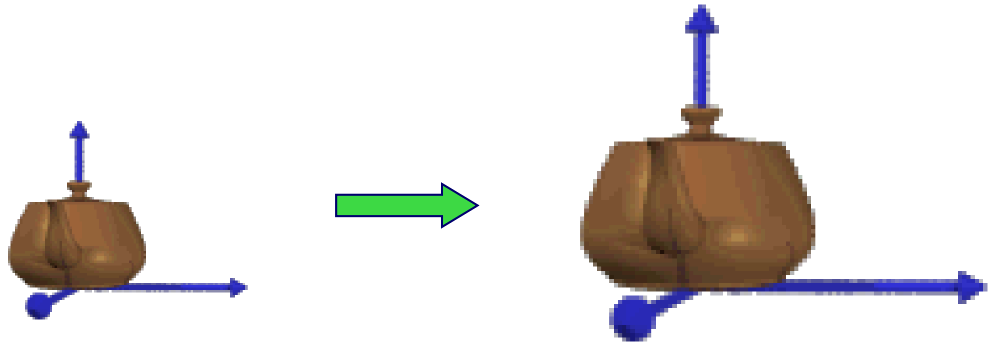
- Cross product follows right-hand rule in a right-handed coordinate system, and left-hand rule in left-handed system.

3D Translation



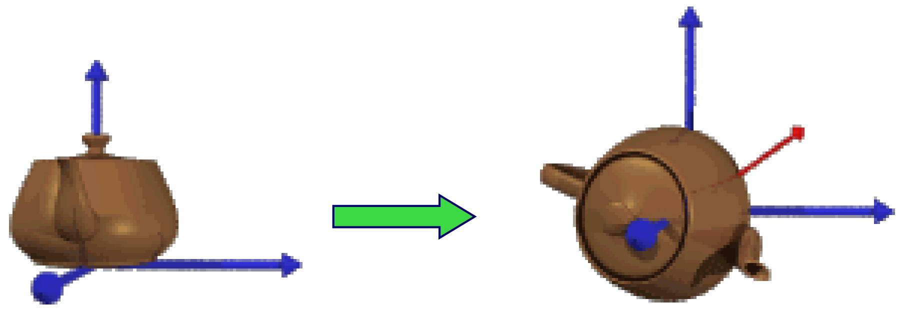
$$\mathbf{T} = \begin{bmatrix} 1 & 0 & 0 & \mathbf{t}_0 \\ 0 & 1 & 0 & \mathbf{t}_1 \\ 0 & 0 & 1 & \mathbf{t}_2 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

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



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

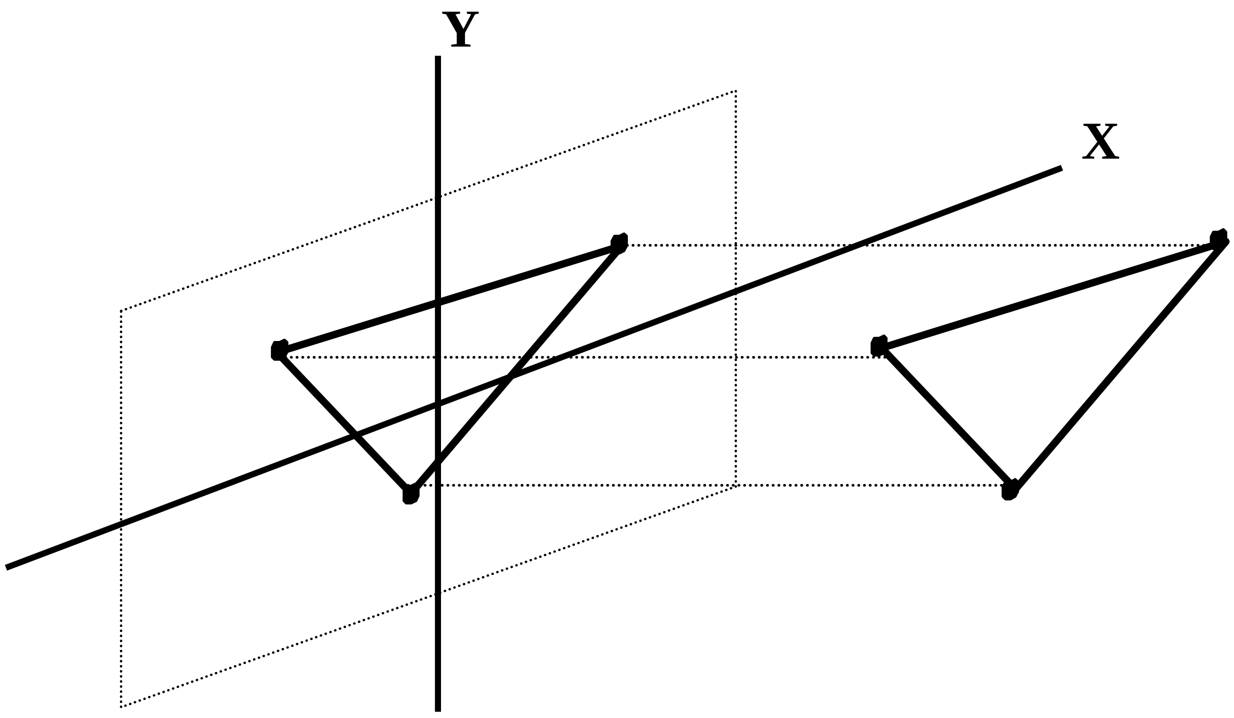
$$R_y = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad R_z = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Transformation

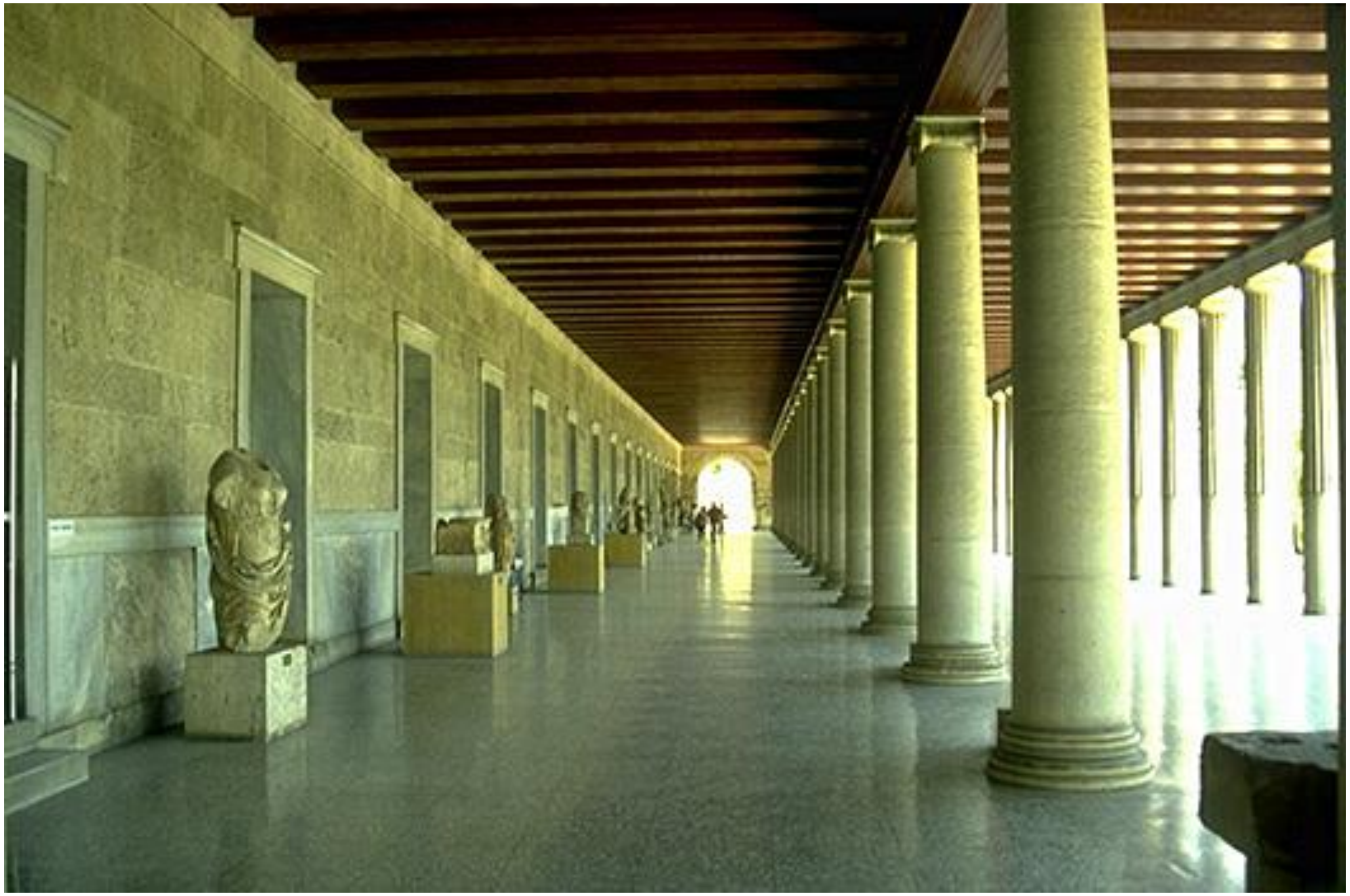
1. 2D Transformation
2. 3D Transformation
3. Viewing Projection

Orthographic Projection

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

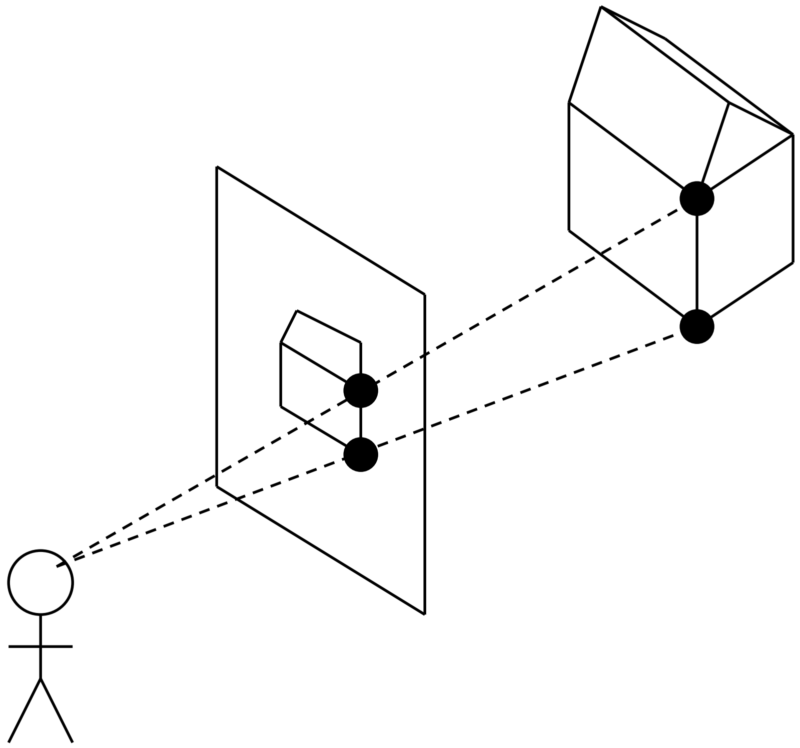


Perspective



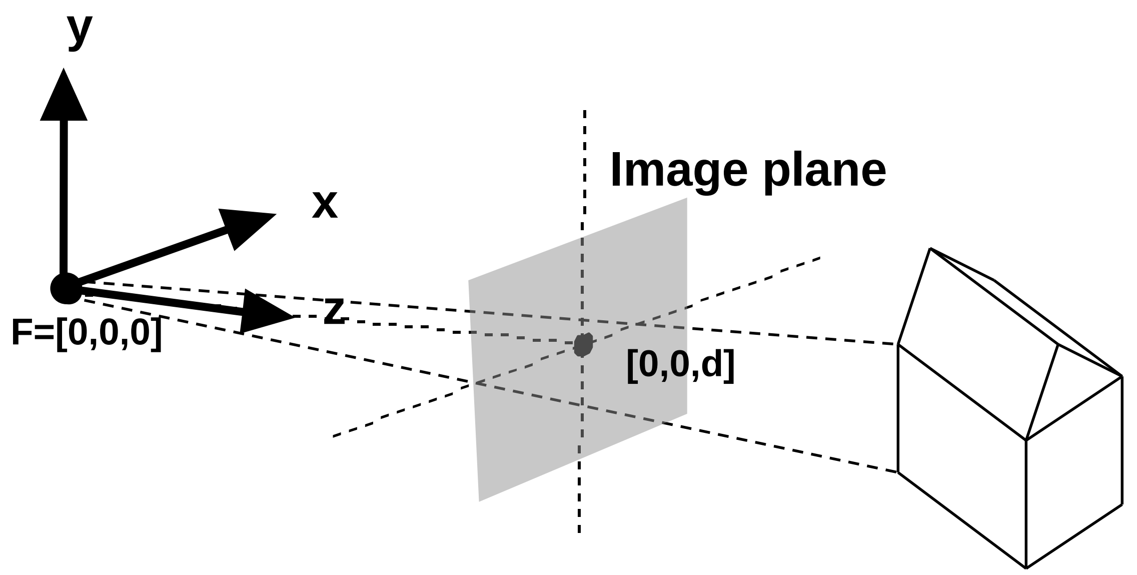
http://www.indiana.edu/~kglowack/athens/

Perspective Projection

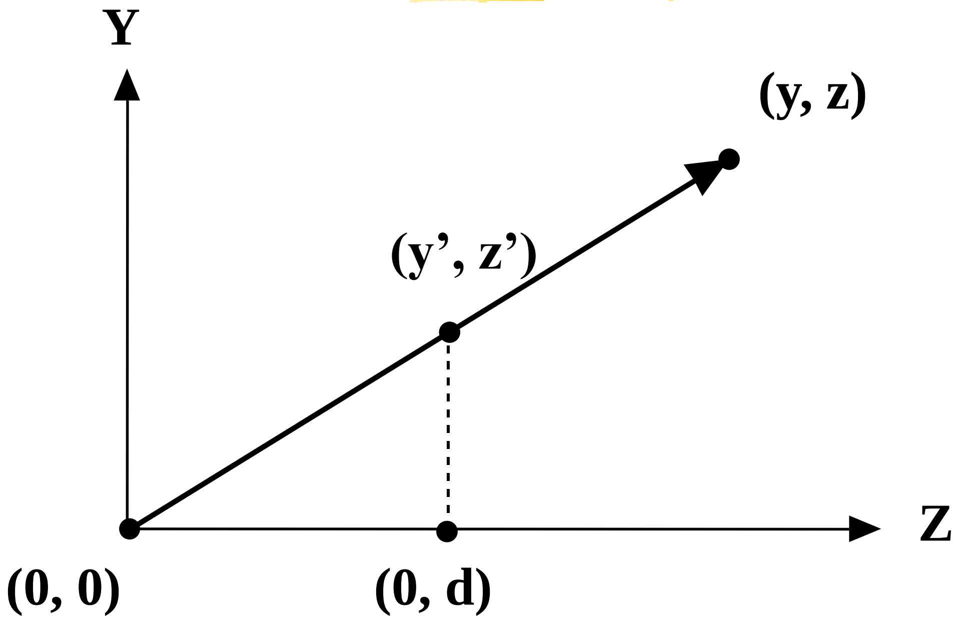


A Simple Perspective Camera

- 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)

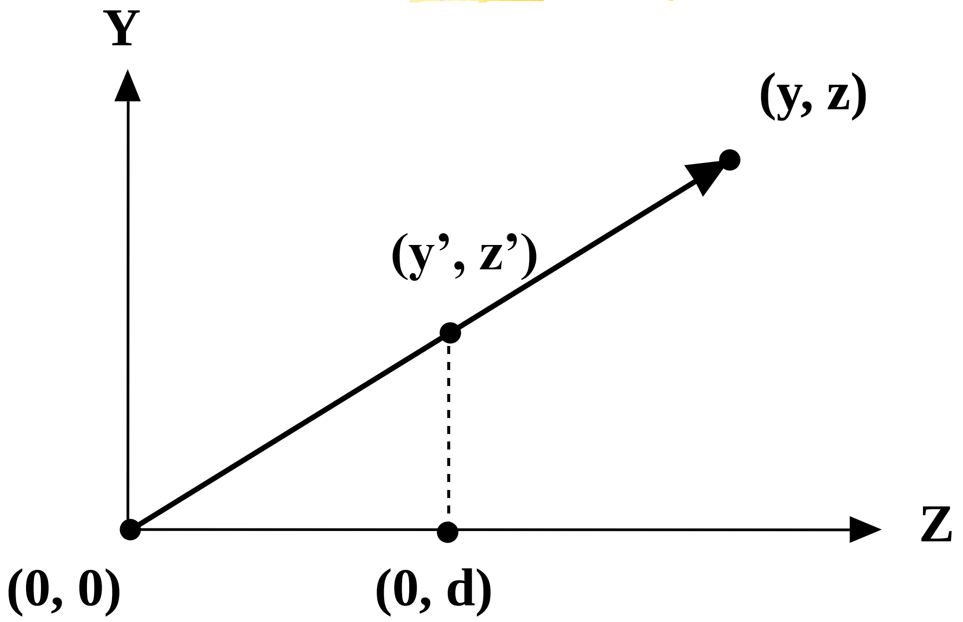


Similar Triangles



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

Similar Triangles



$z' = d$

$y' / z' = y/z$

$y' / d = y/z$

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

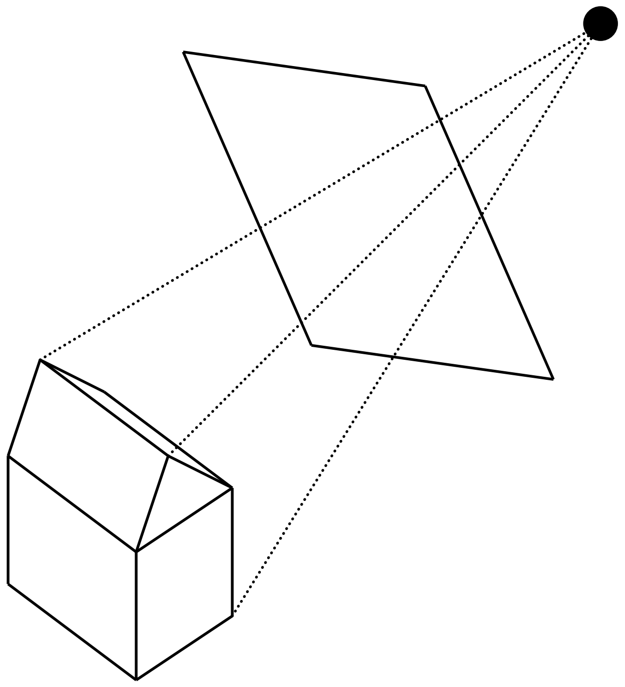
point [x,y,z] projects to [(d/z)x, (d/z)y, d]

A Perspective Projection Matrix

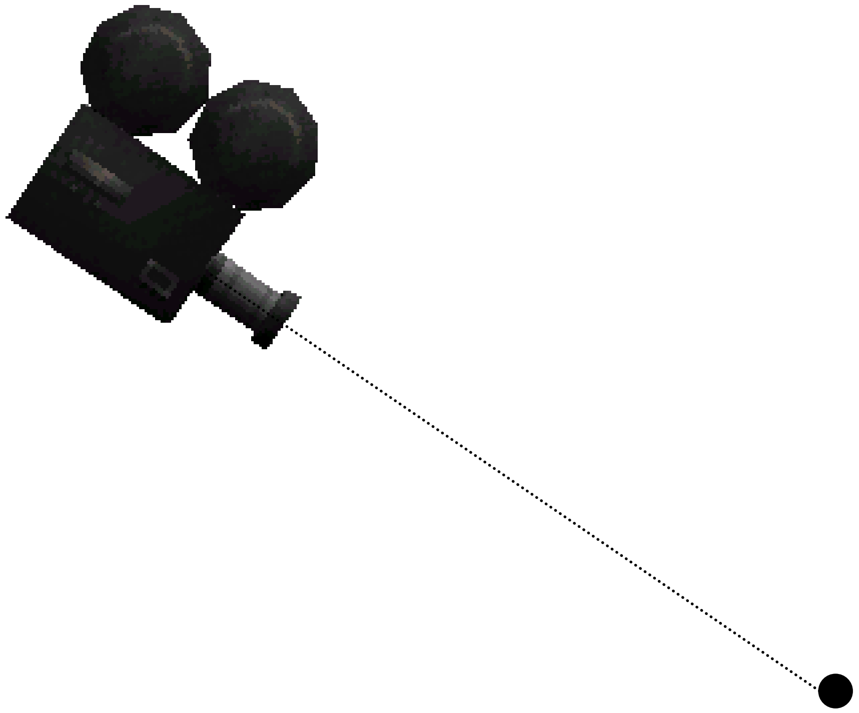
- **Projection using homogeneous coordinates:**
  - transform [x, y, z] to [(d/z)x, (d/z)y, d]

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

Camera Position and Orientation

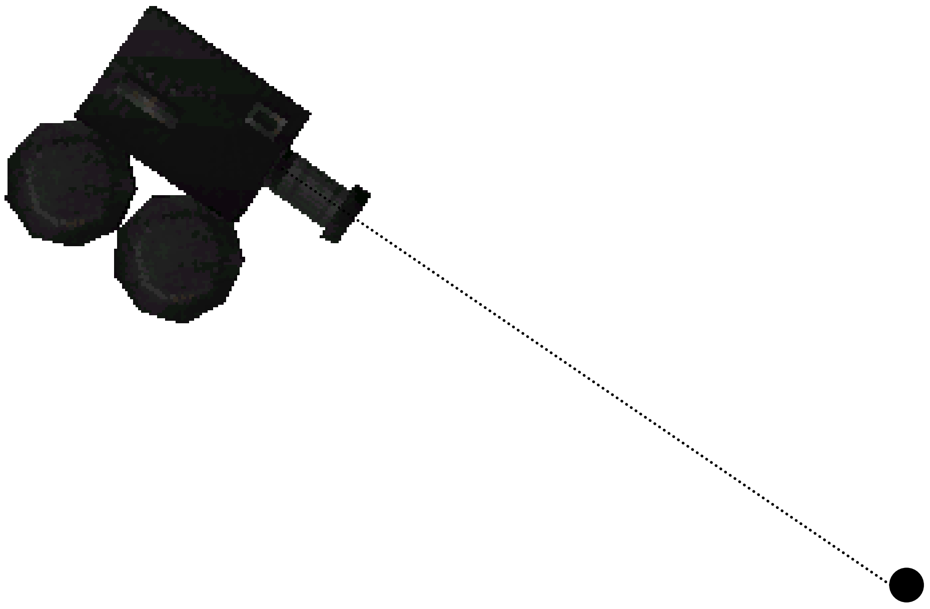


LookFrom And LookAt

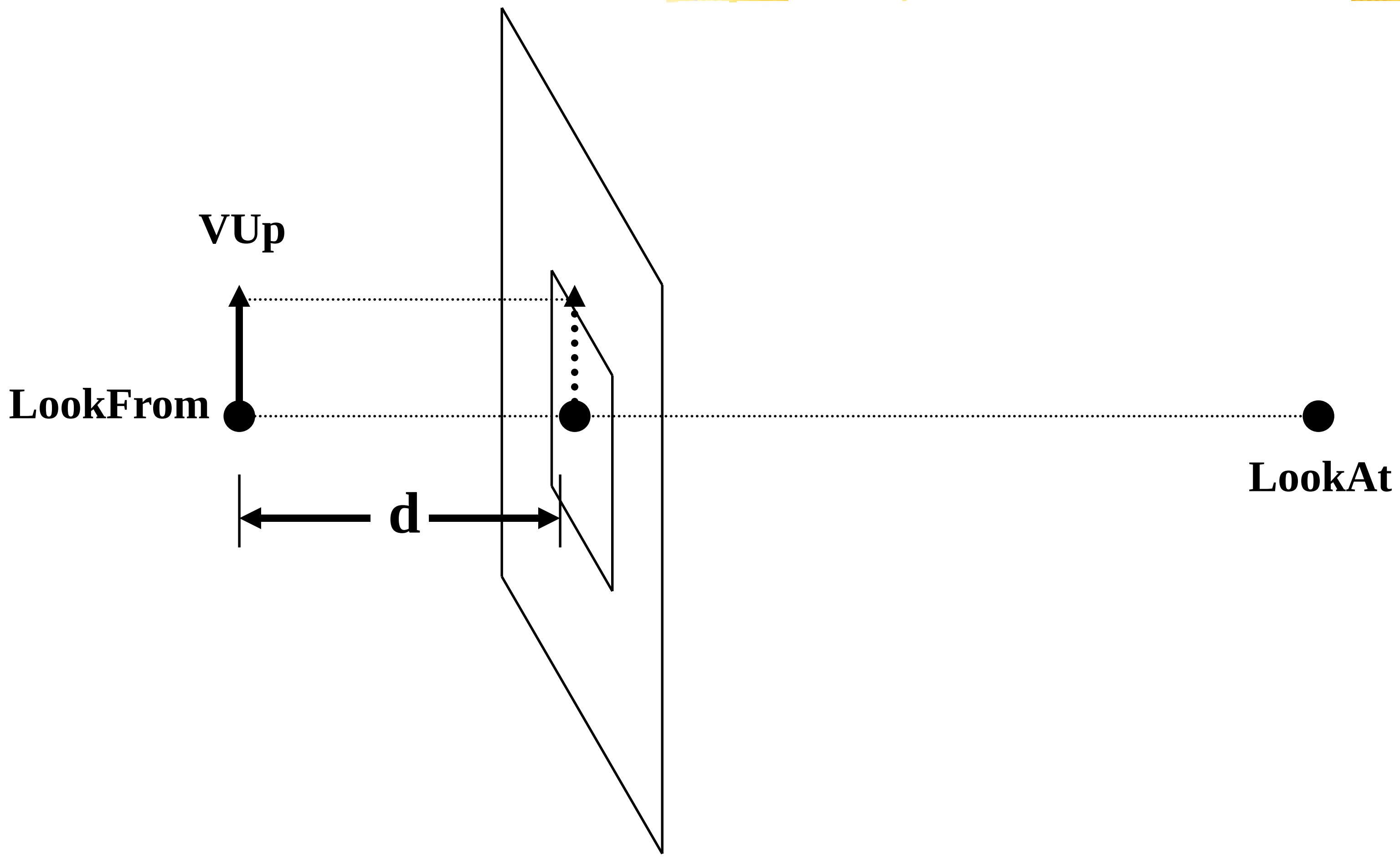


Is This Enough?

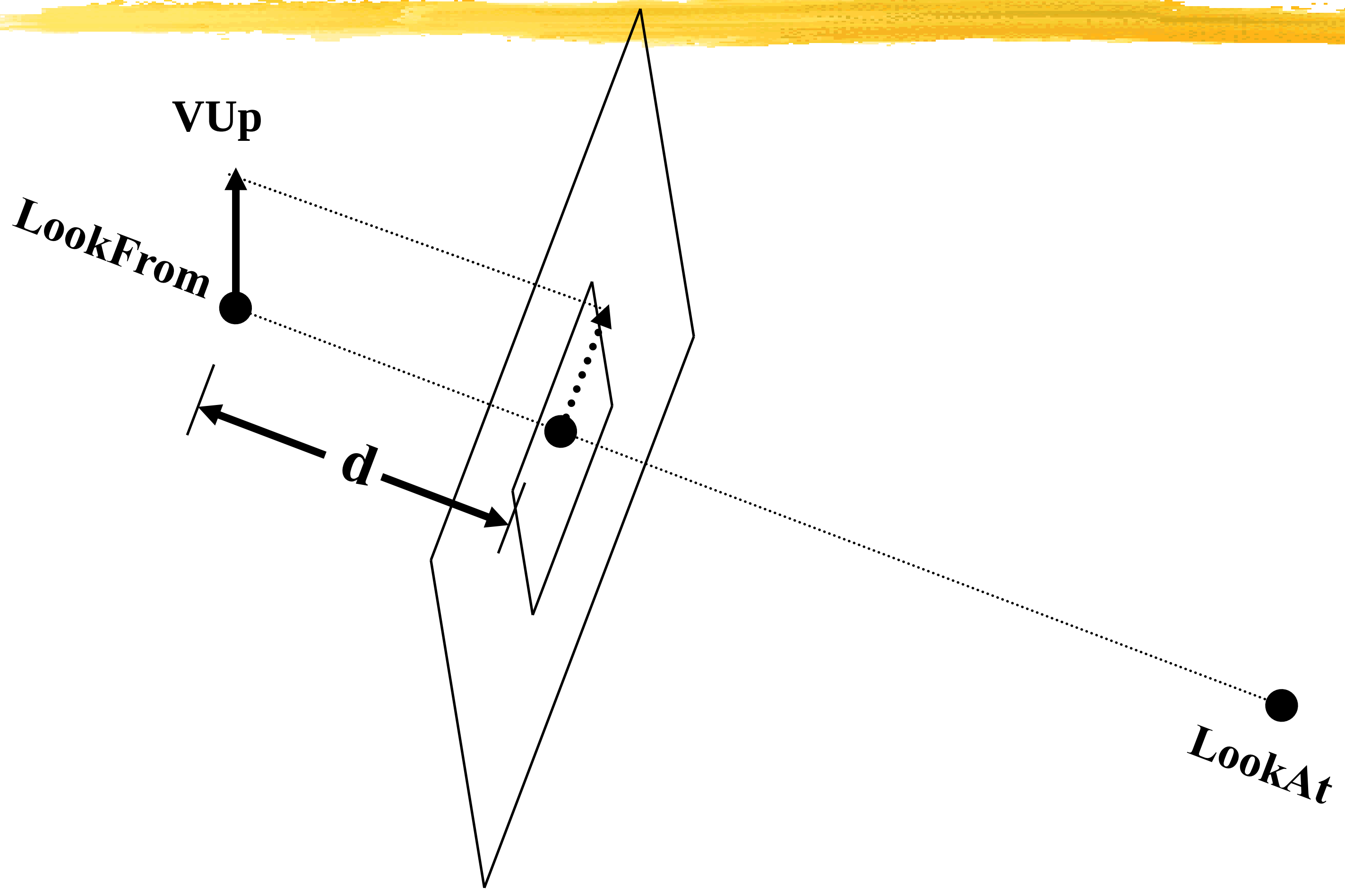
LookFrom And LookAt



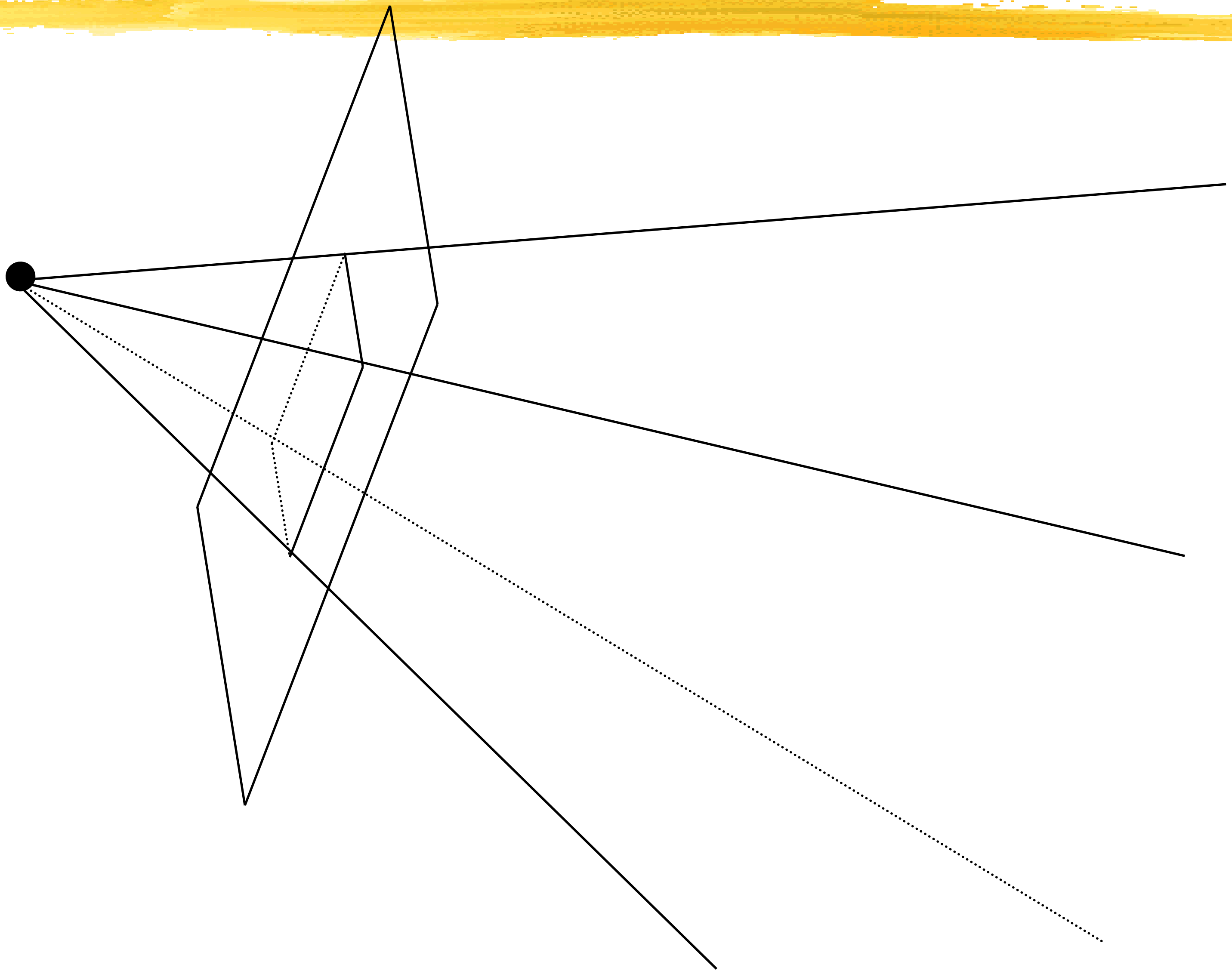
Complete Camera Specification



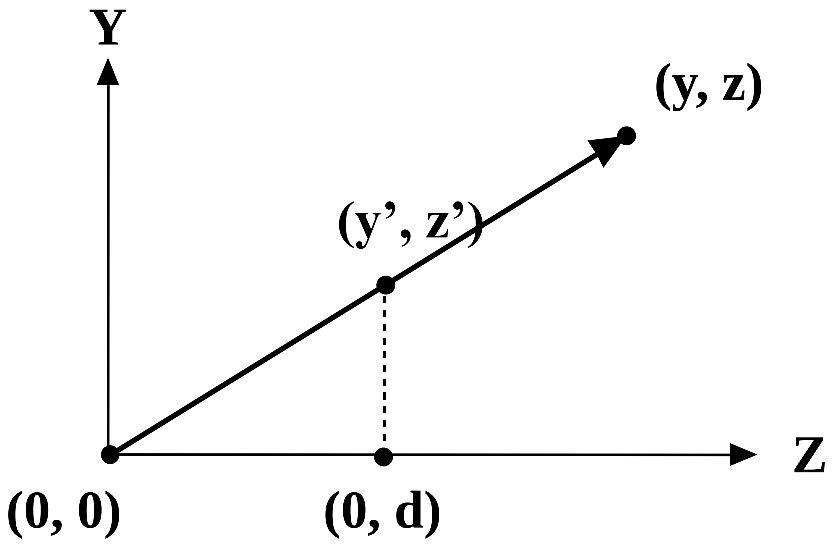
Complete Camera Specification

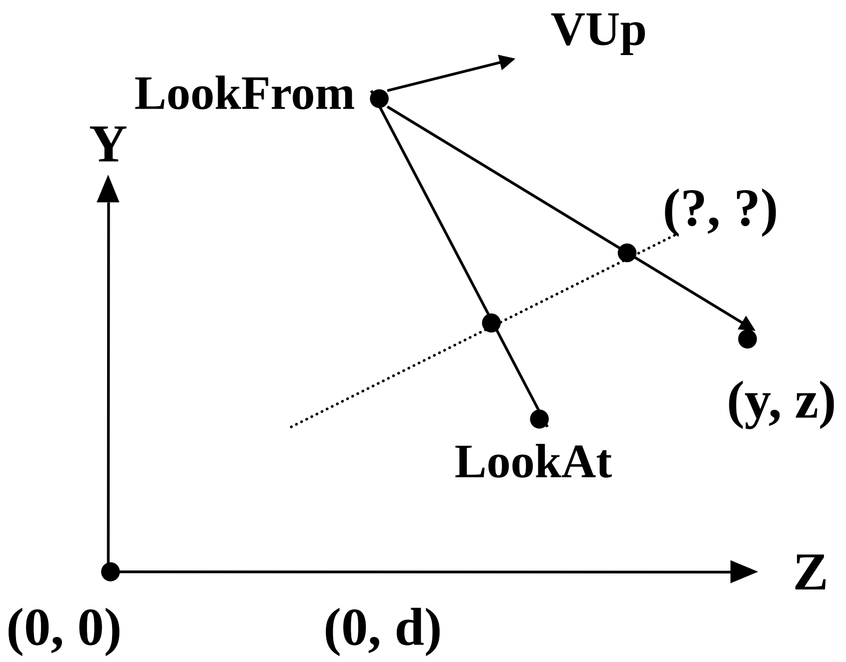


Viewing Volume

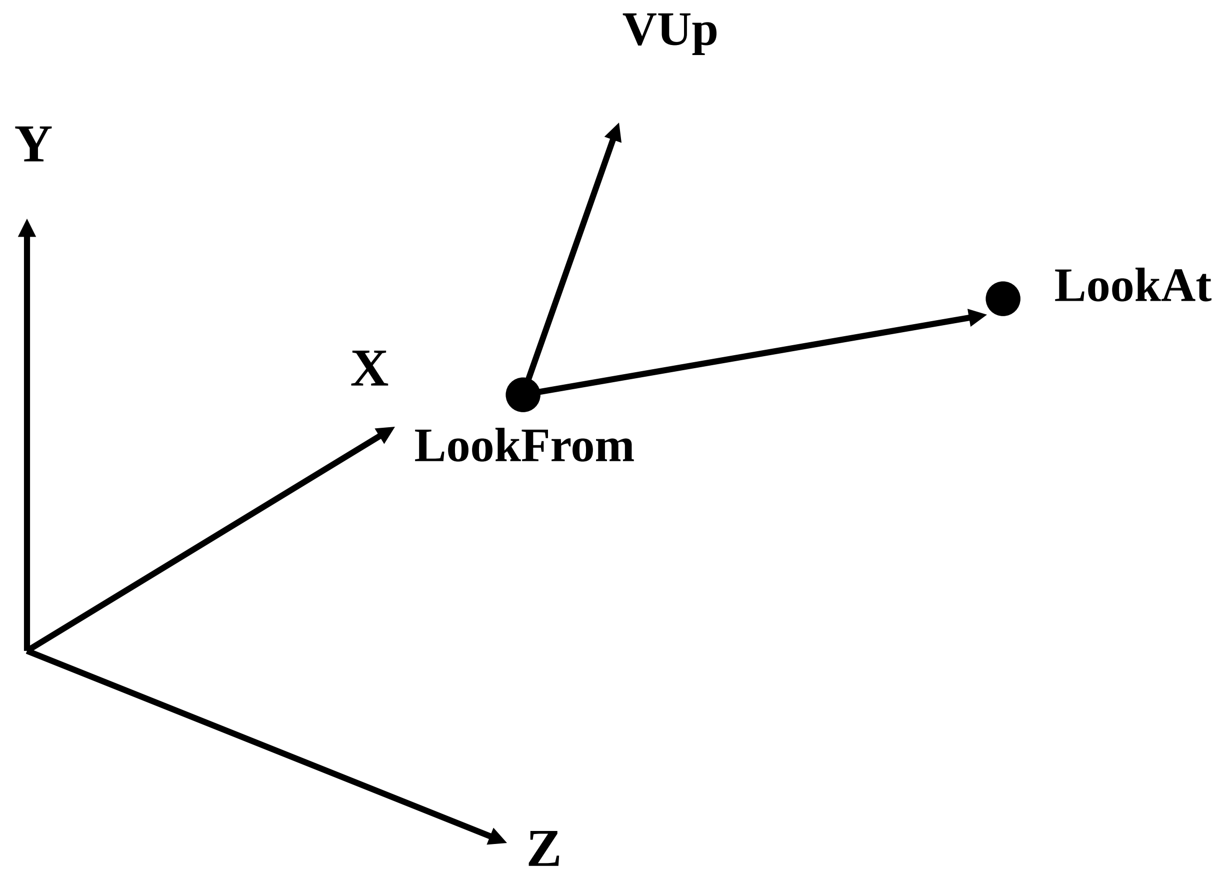


Rendering from any camera position

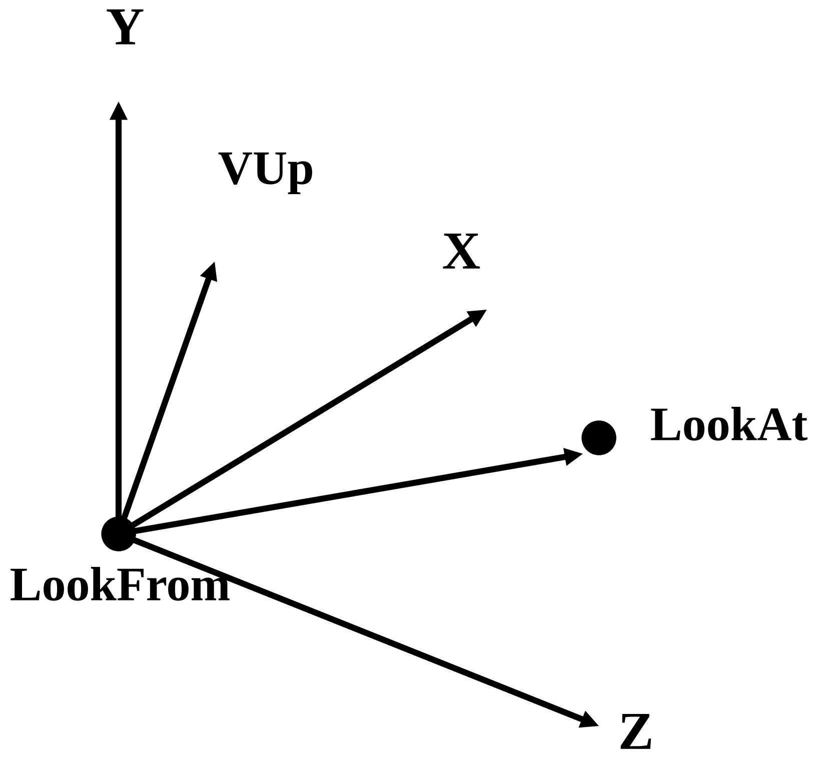




Viewing Transformations

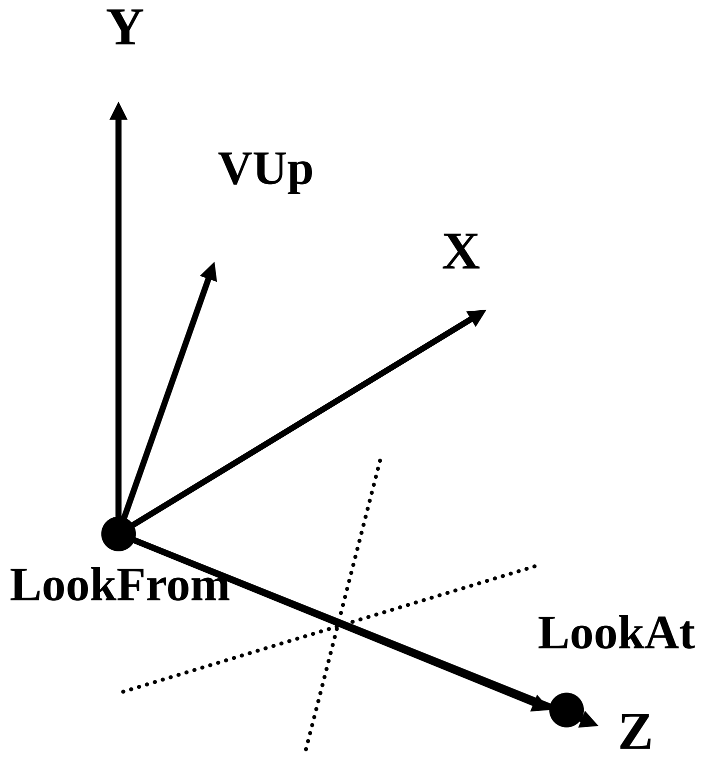


Viewing Transformations



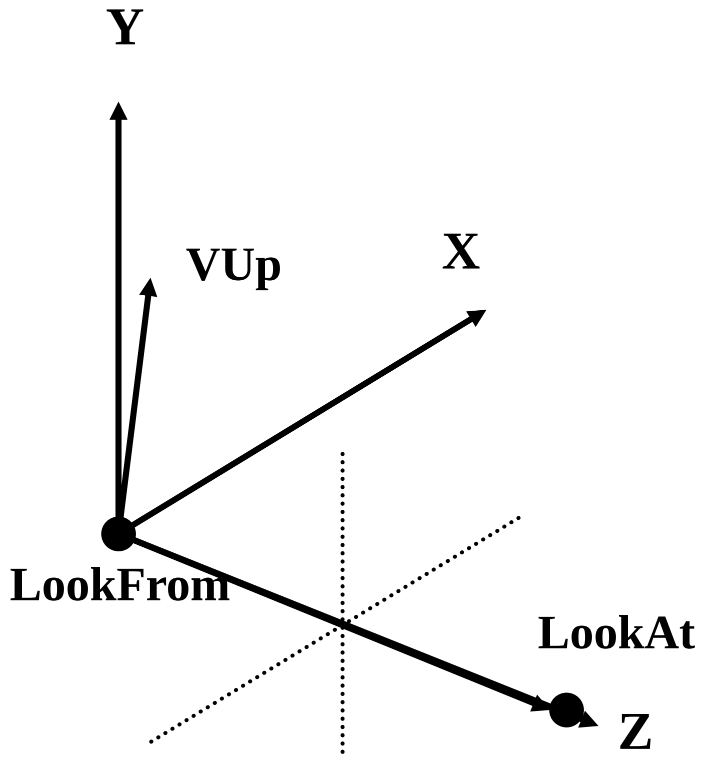
Translate LookFrom to origin

Viewing Transformations



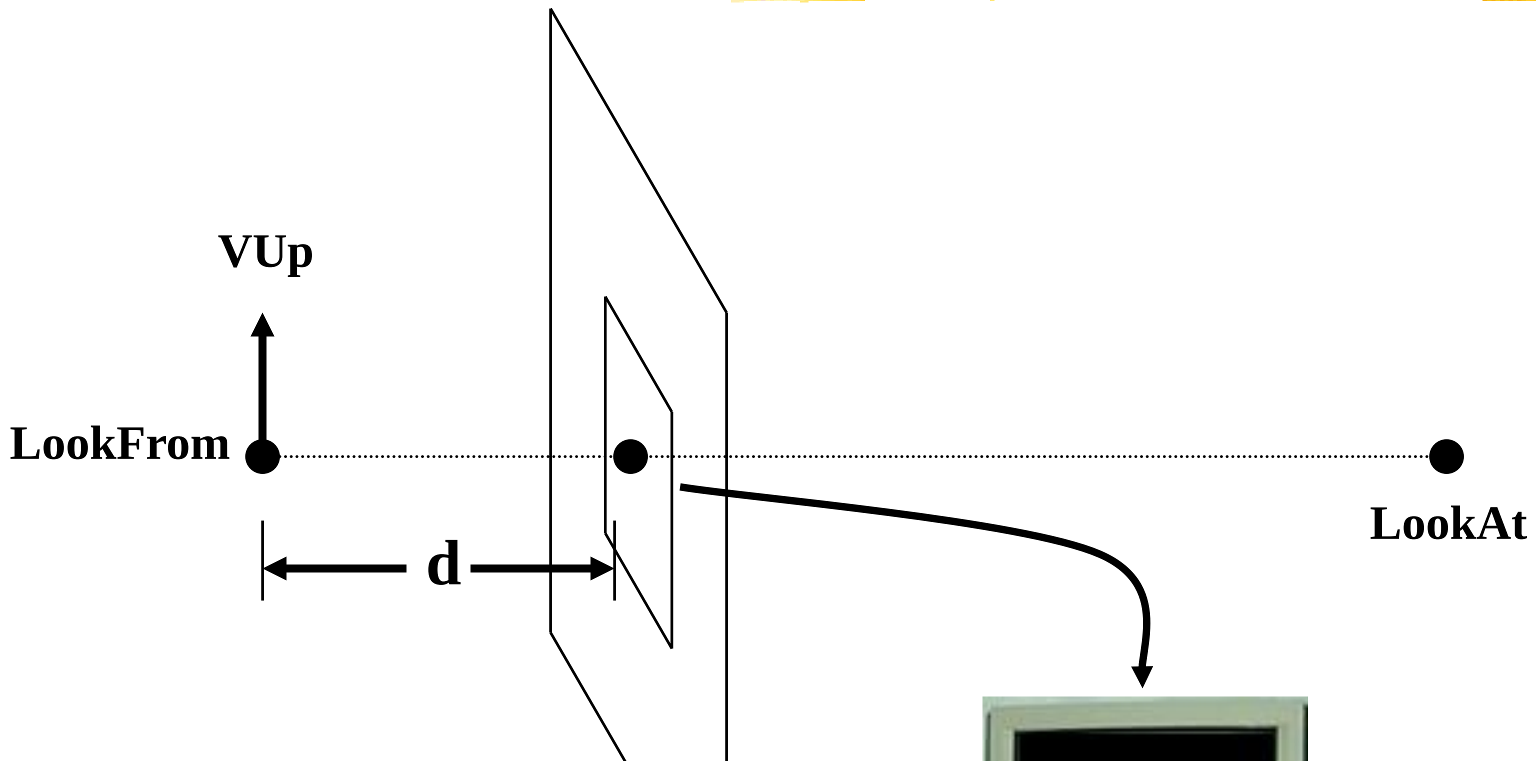
Rotate LookAt to Z axis (axis-angle rotation)

Viewing Transformations



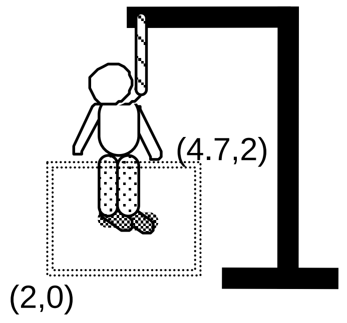
Rotate about Z to get the projection of Vup parallel to the Y axis

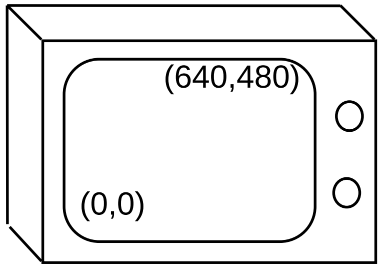
Screen Coordinates

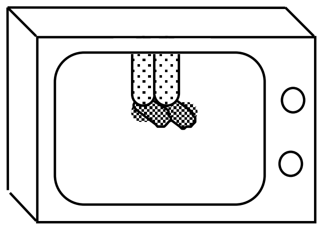


Viewport Transformations

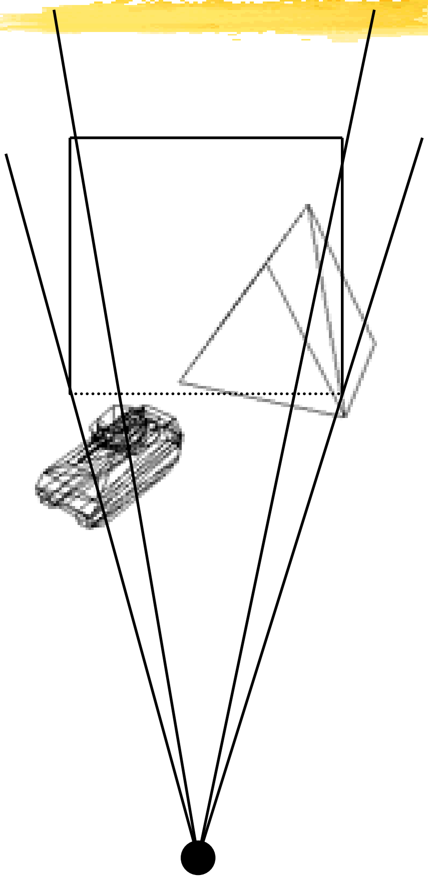
- A transformation maps the visible (model) world onto screen or window coordinates
- In OpenGL a viewport transformation, e.g. glOrtho(), defines what part of the world is mapped in standard "Normalized Device Coordinates" ((-1,-1) to (1,1))
- The viewpoint transformation maps NDC into actual window,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

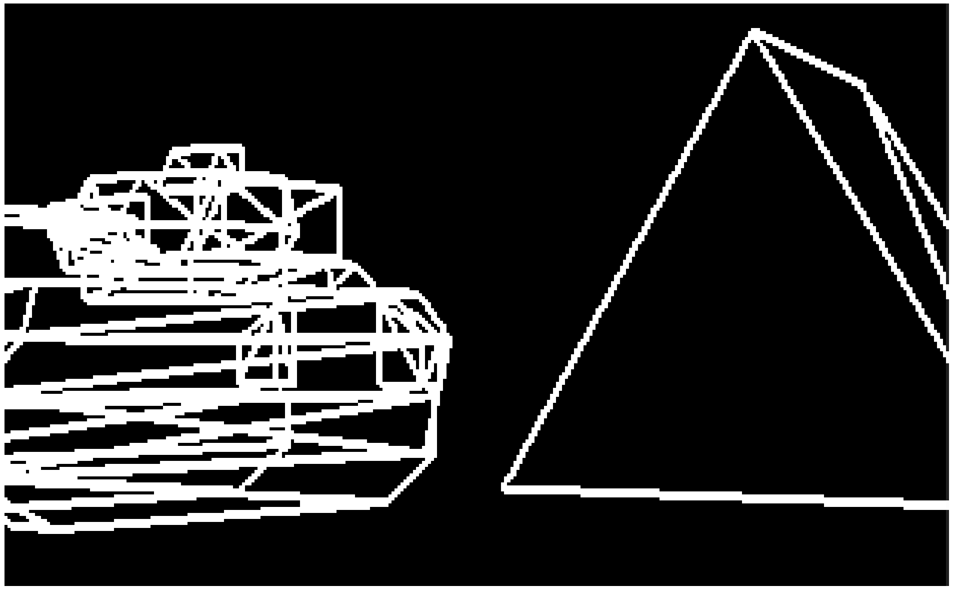




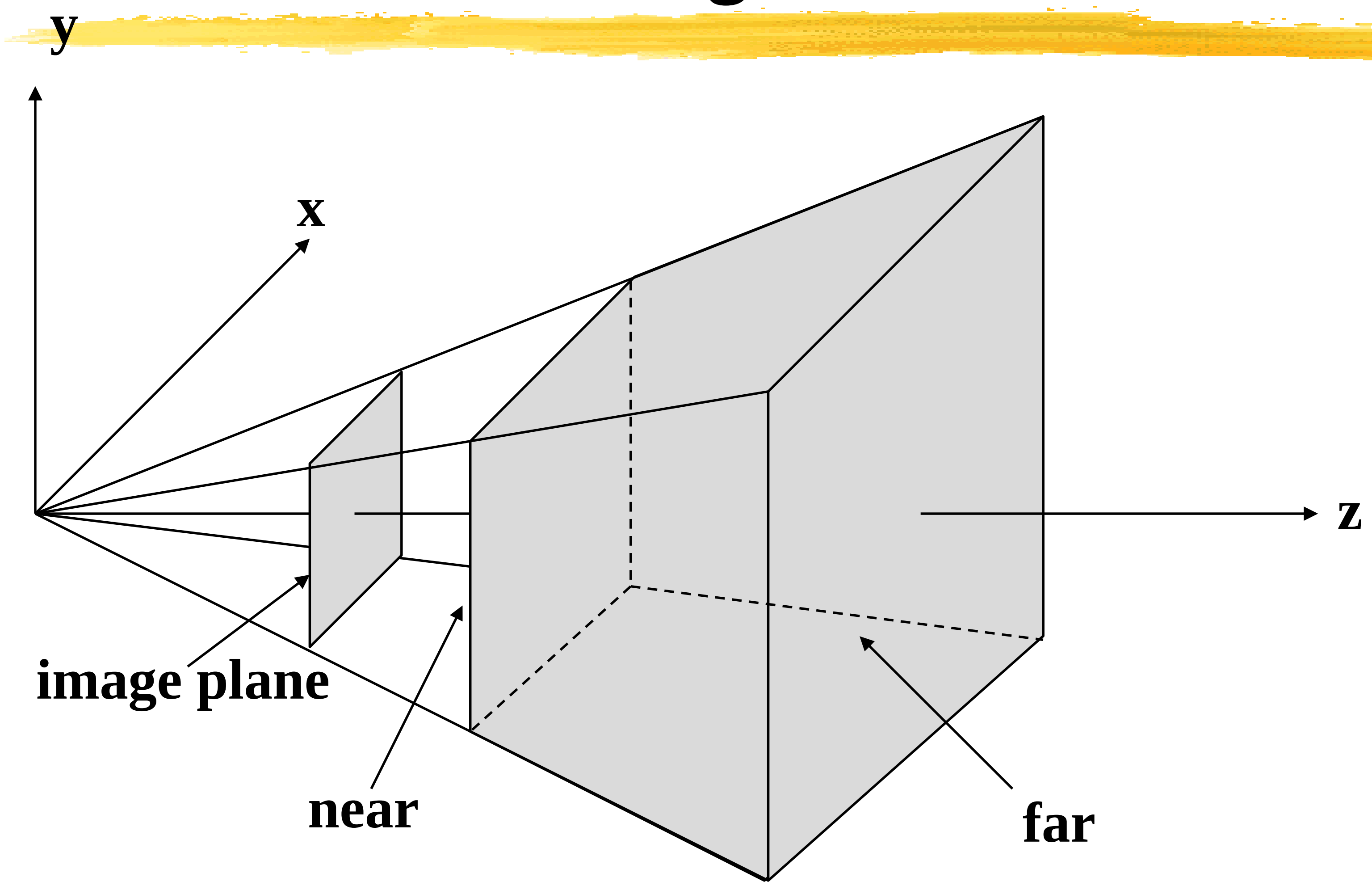


Clipping



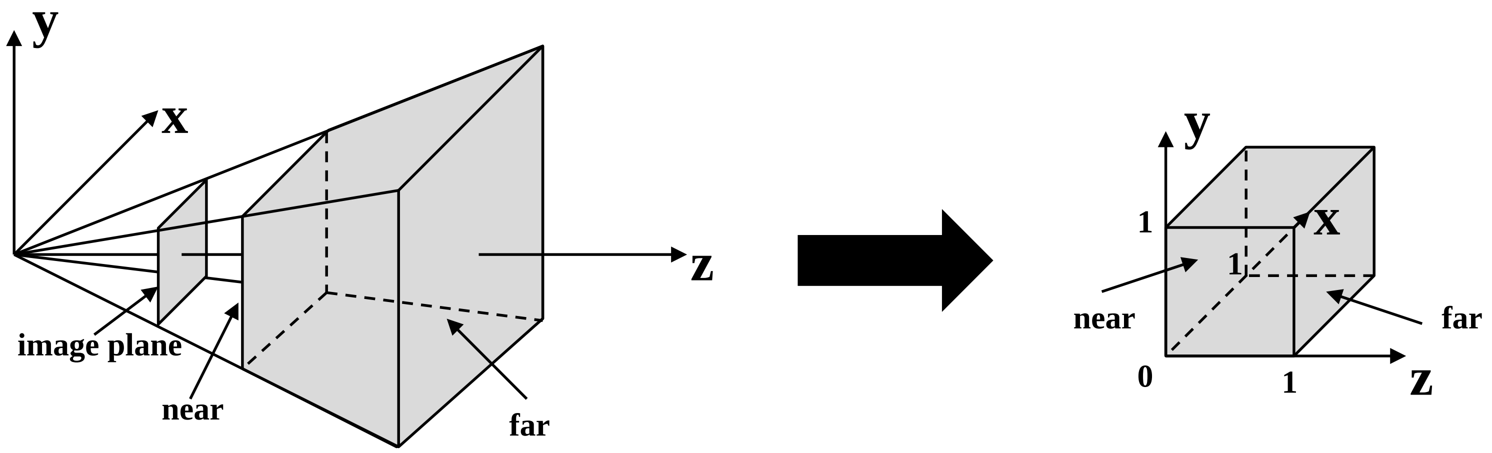


The Viewing Frustum



Normalizing the Viewing Frustum

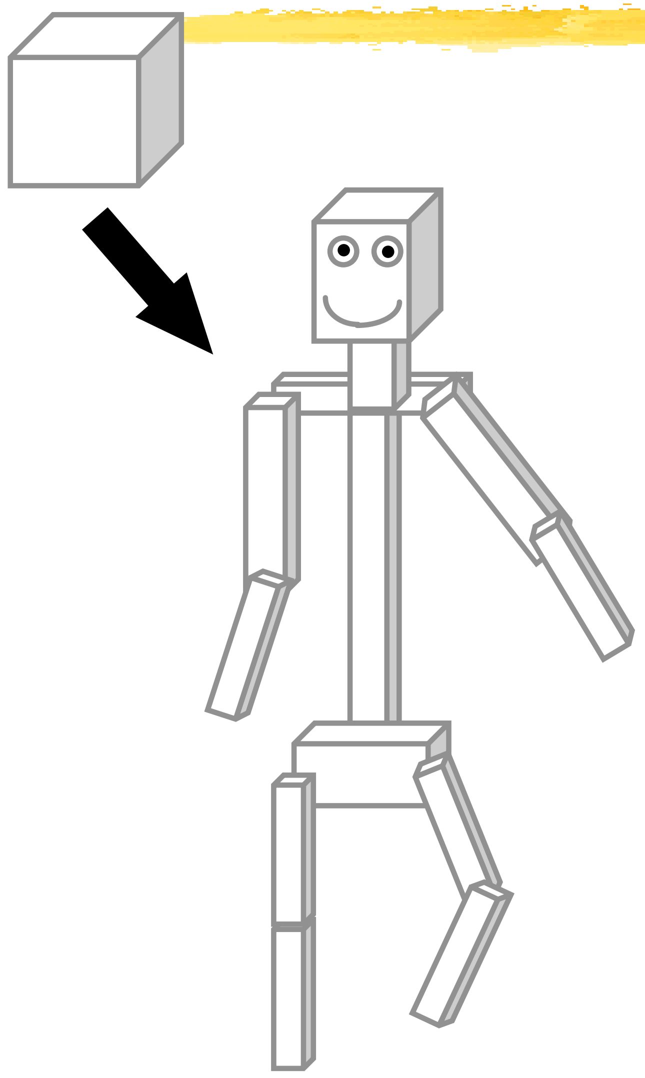
- Transform frustum to a cube before clipping



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

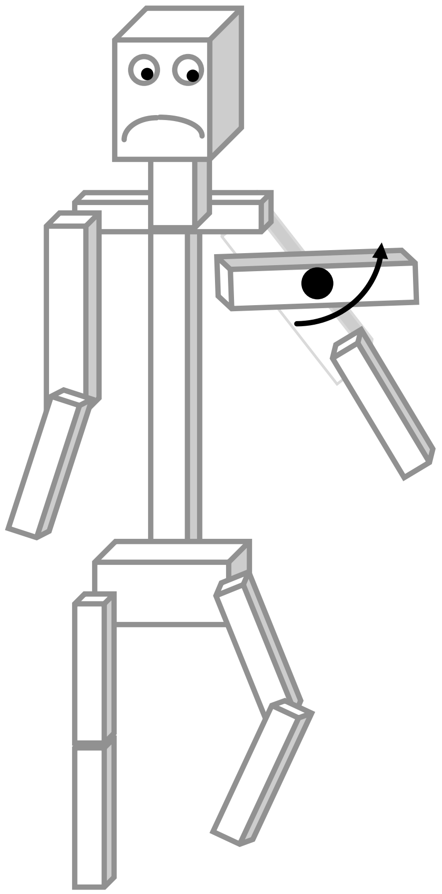
Model and Transformation Hierarchy

How to Model a Stick Person



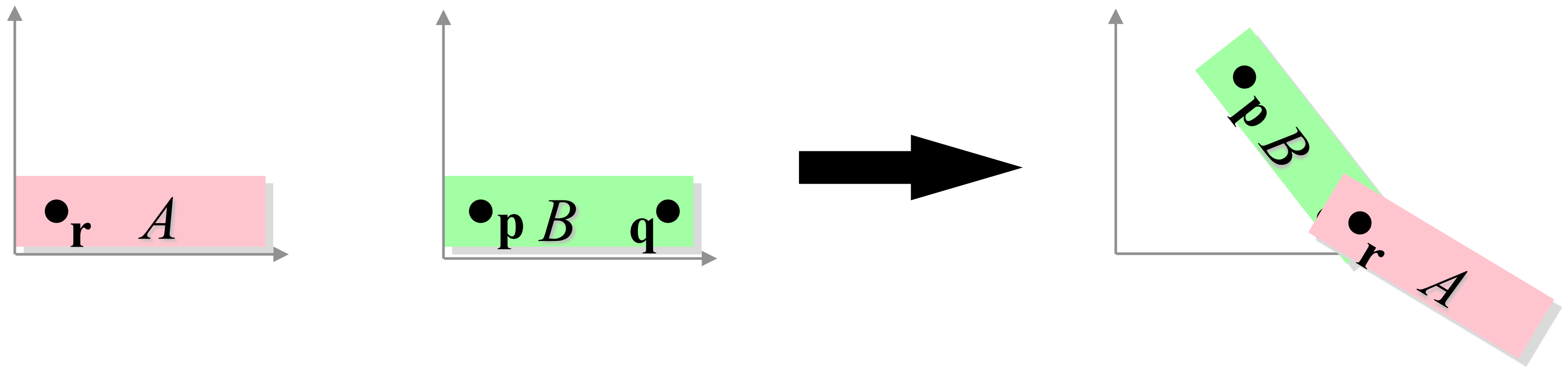
- Make a stick person out of cubes
- Just translate, rotate, and scale each one to get the right size, shape, position, and orientation.
- Looks great, until you try to make it move.

The Right Control Knobs



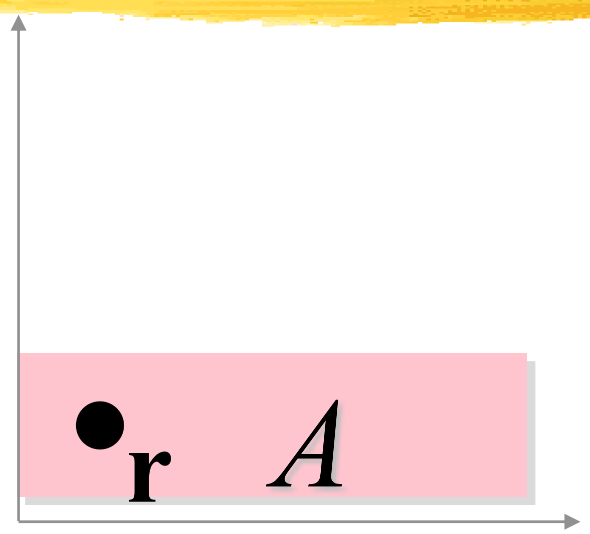
- As soon as you want to change something, the model *likely* falls apart
- Reason: the thing you’re modeling is *constrained* but your model doesn’t know it
- Wanted:
  - some sort of representation of *structure*
  - *Control knob*
- This kind of control knob is convenient for static models, and *vital*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

Making an Articulated Model



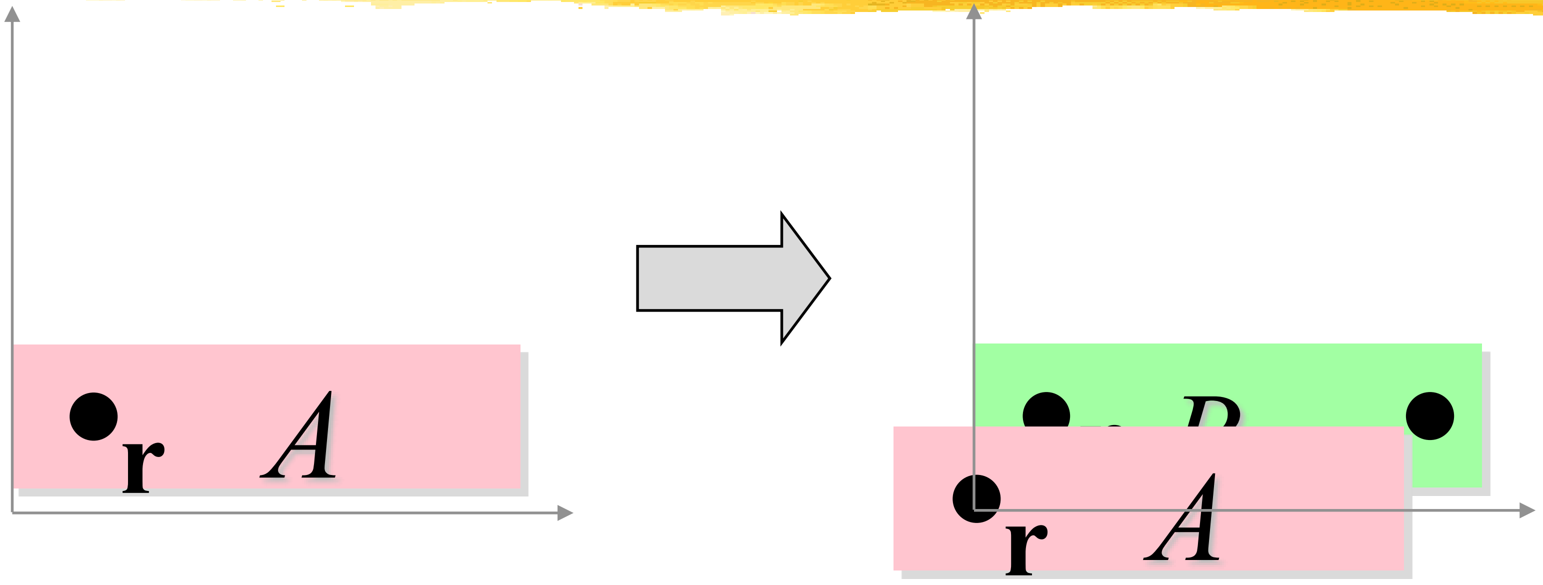
- A minimal 2-D jointed object:
  - Two pieces, *A* (“forearm”) and *B* (“upper arm”)
  - Attach point q on *B* to point r on *A* (“elbow”)
  - Desired control knobs:
    - » u: shoulder angle (*A* and *B* rotate together about p)
    - » v: elbow angle (*A* rotates about r, which stays attached to p)

Making an Arm, step 1



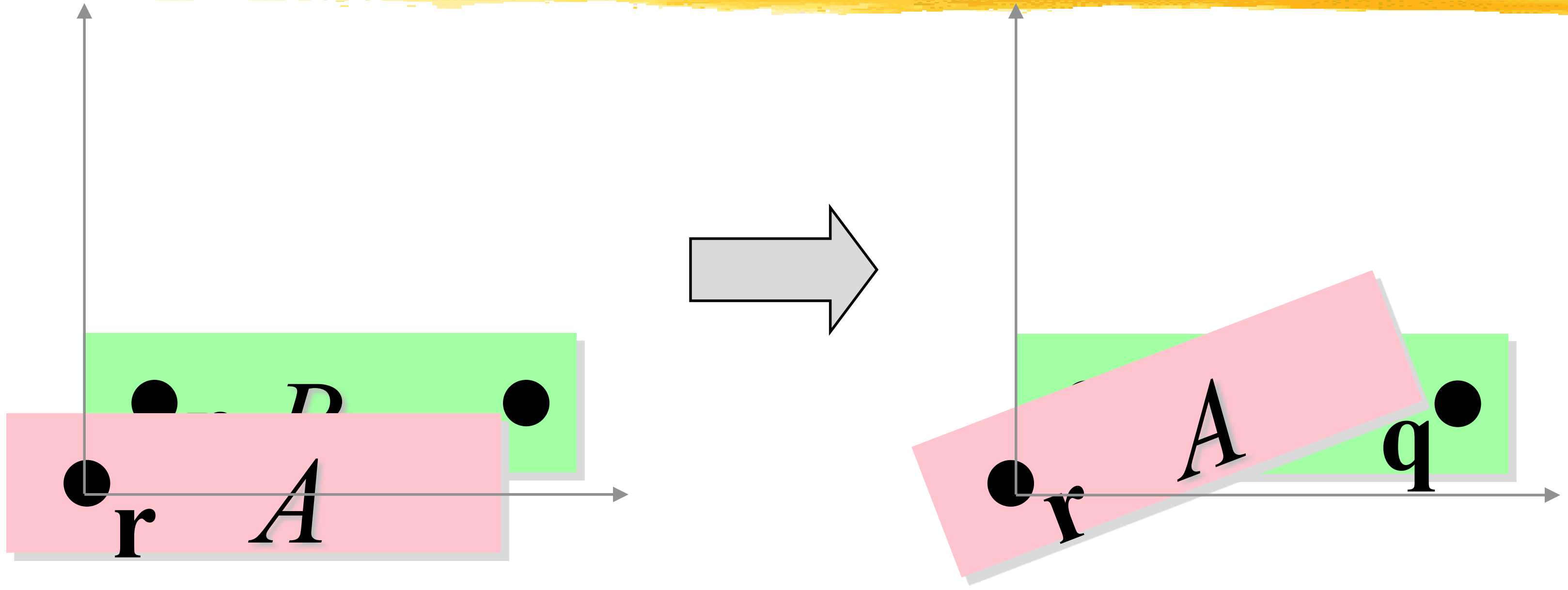
- Start with *A* and *B* in their untransformed configurations (*B* is hiding behind *A*)
- First apply a series of transformations to *A*, leaving *B* where it is…

Making an Arm, step 2



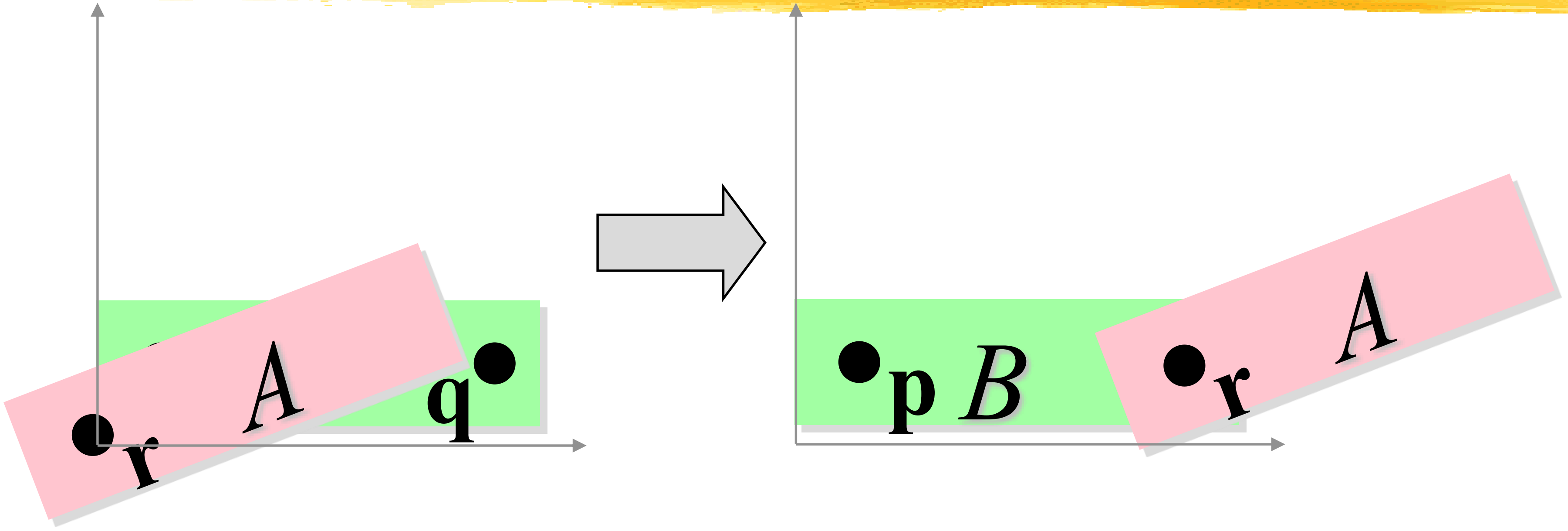
- Translate by −r, bringing r to the origin
- You can now see *B* peeking out from behind *A*

Making an Arm, step 3



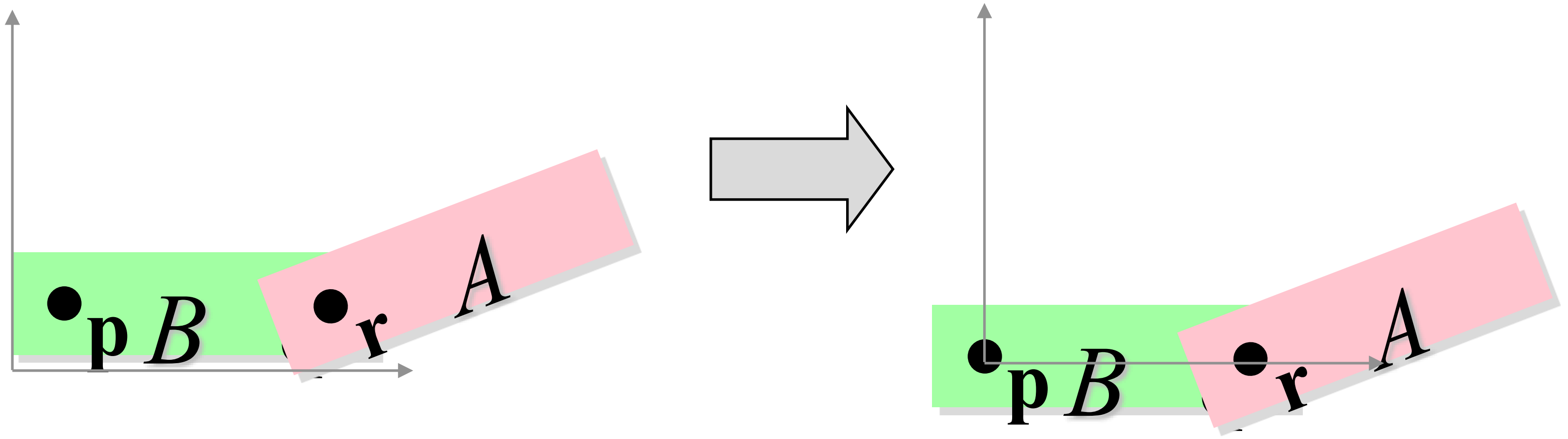
- Next, we rotate *A* by v (the “elbow” angle)

Making an Arm, step 4



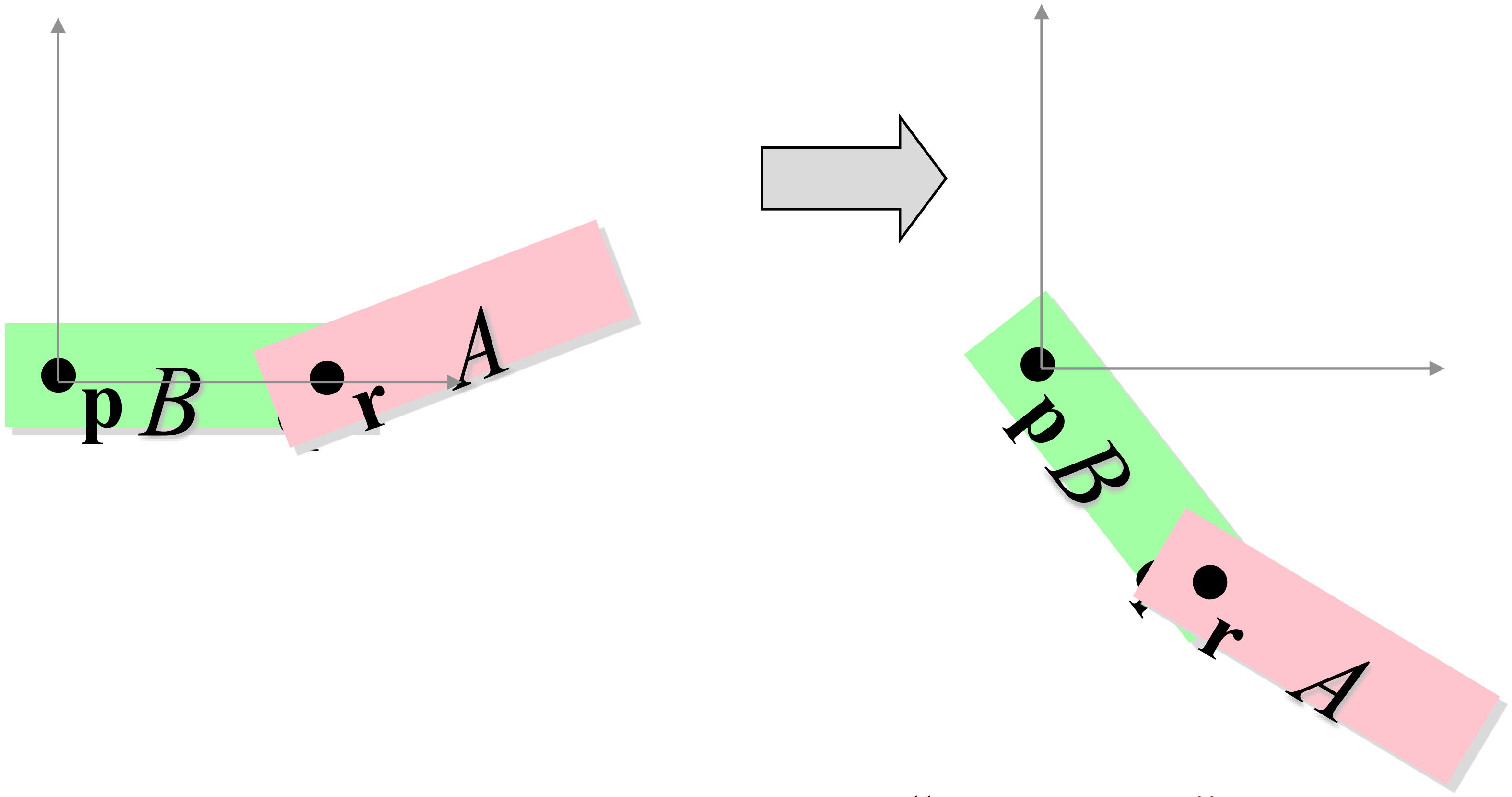
- Translate *A* by q, bringing r and q together to form the elbow joint
- We can regard q as the origin of the *elbow coordinate system*, and regard A as being in this coordinate system.

Making an Arm, step 5



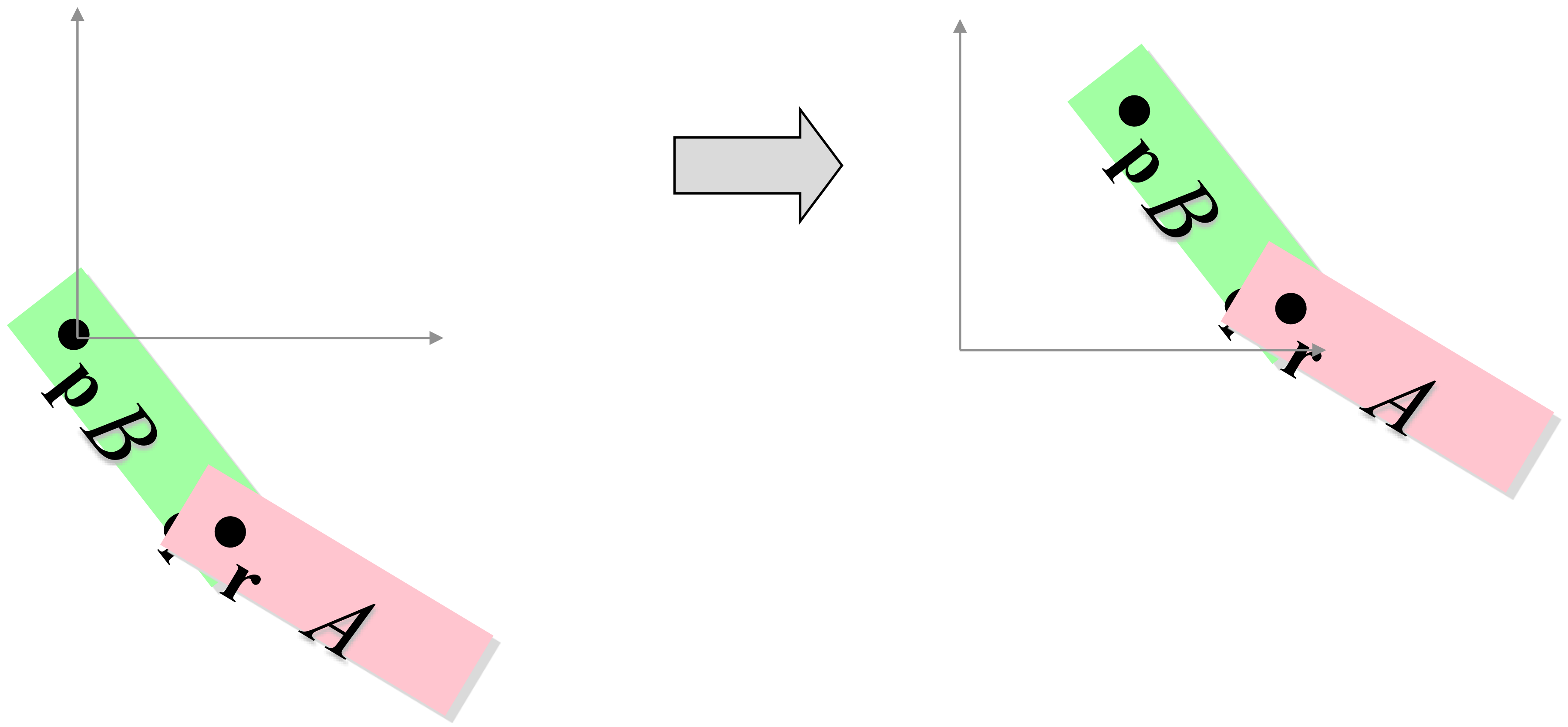
- From now on, each transformation applies to *both* *A* and *B* (This is important!)
- First, translate by -p, bringing p to the origin
- *A* and *B* both move together, so the elbow doesn’t separate!

Making an Arm, step 6



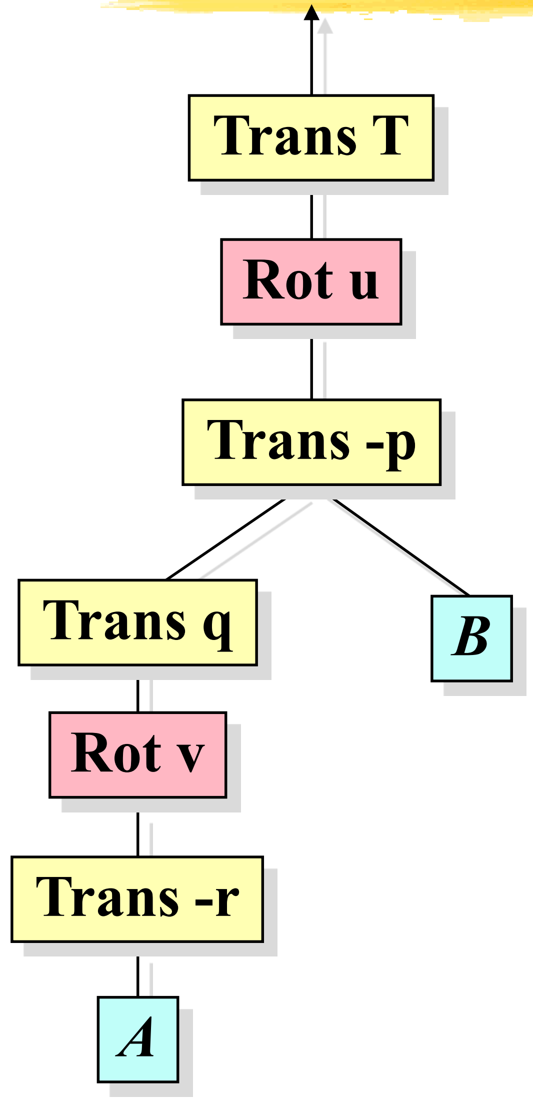
- Then, we rotate by u, the "shoulder" angle
- Again, *A* and *B* rotate together

Making an Arm, step 7

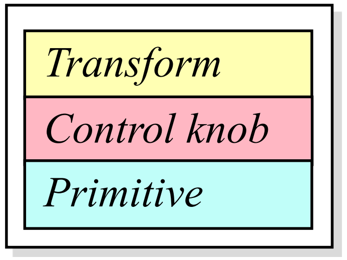


- Finally, translate by T, bringing the arm where we want it
- p is at origin of *shoulder coordinate system*

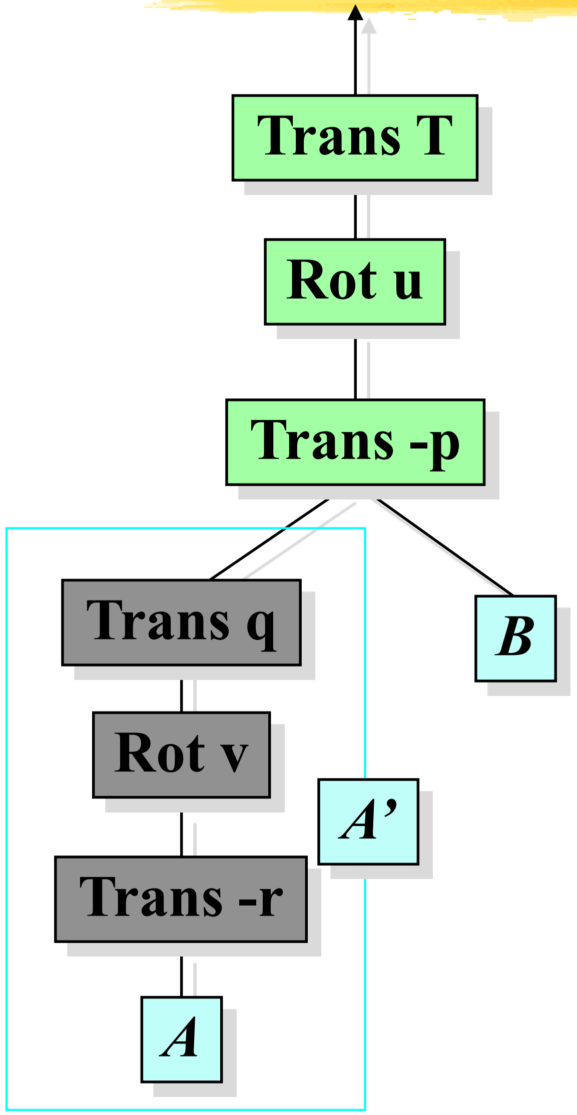
Transformation Hierarchies



- 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



Transformation Hierarchies

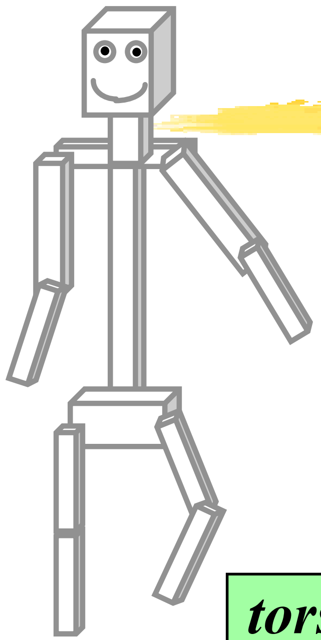


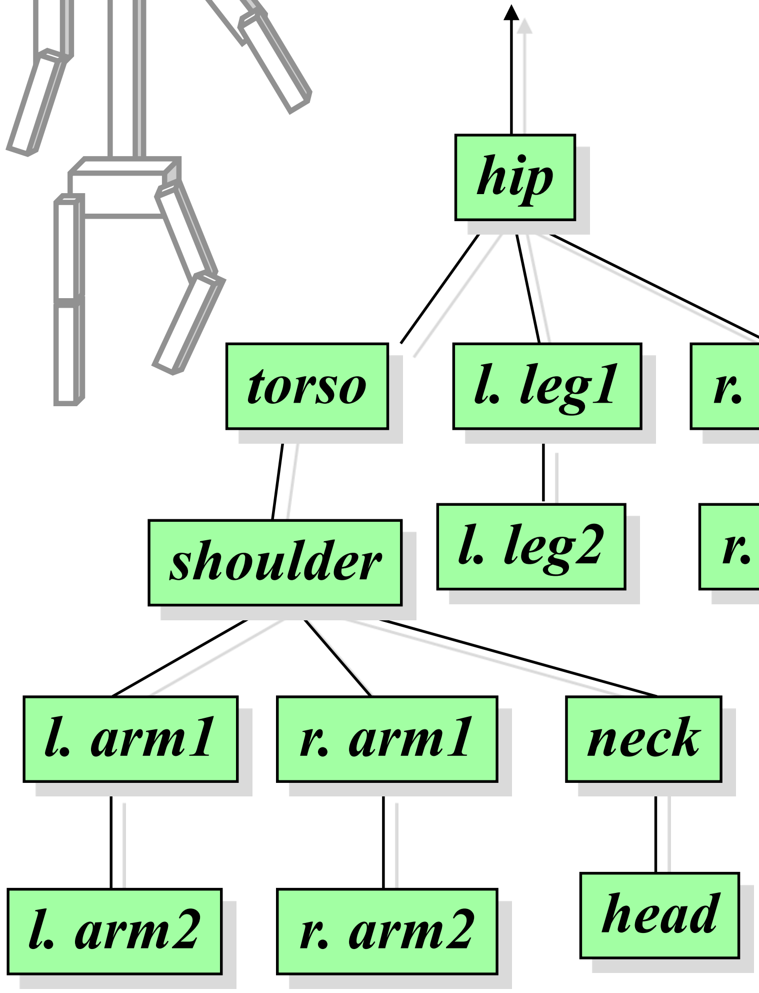
Another point of view:

- 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'

🟩	*Shoulder coordinate transform*
⬛	*Elbow coordinate transform*
🟦	*Primitive*

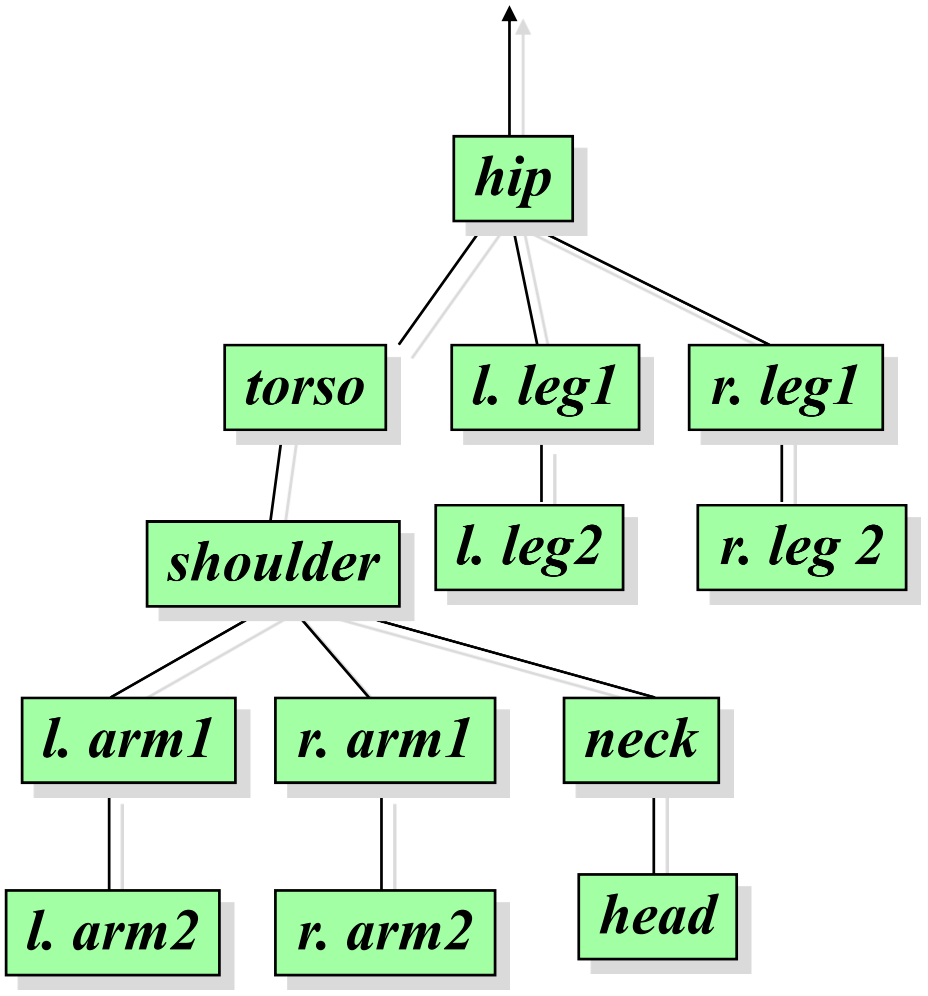
A Schematic Humanoid





- 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:
  - Doesn’t always give the “right” control knobs
    - » e.g. hand or foot position – re-rooting may help
  - Can’t do closed kinematic chains (keep hand on hip)
  - Other constraints: do not walk through walls
- 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?
- Extensions:
  - expressions, languages.

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(…)

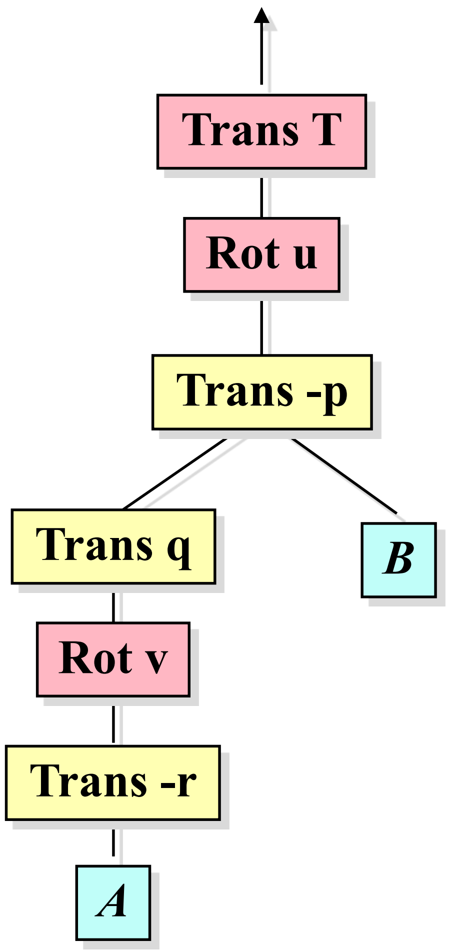
axis/angle rotate. "f" and "d" take floats and doubles, respectively

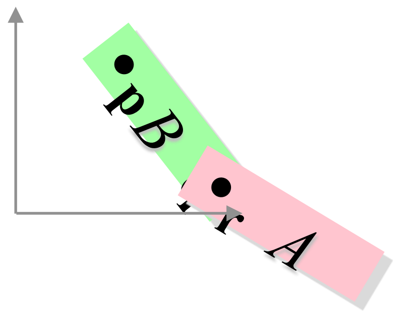
glTranslatef(x,y,z), glScalef(x,y,z)

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();
```





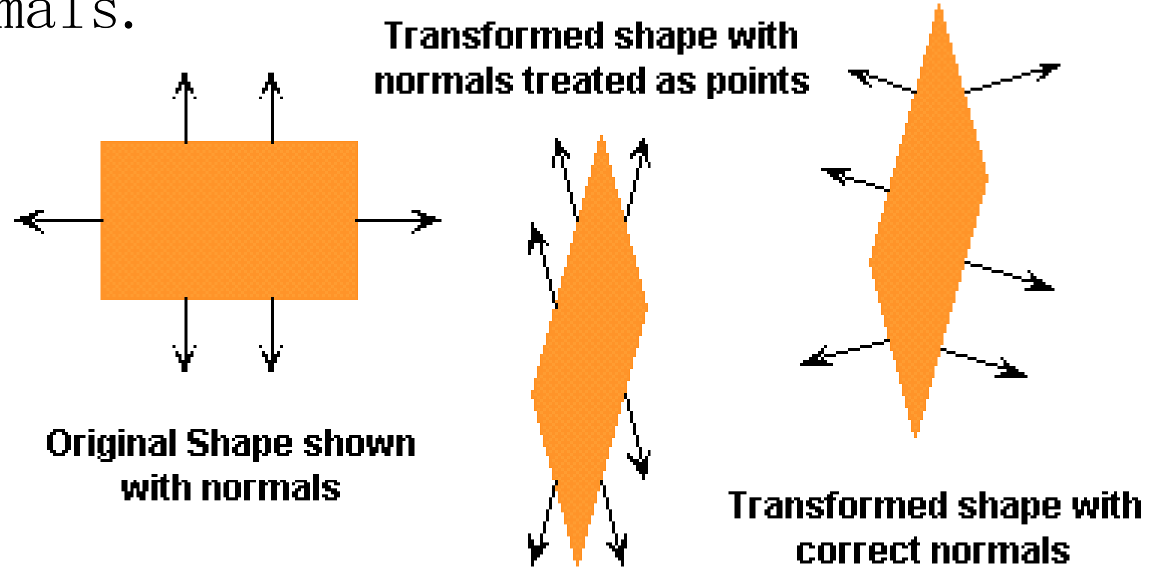
The following not covered in this course

Vector Transformation

- For affine transformation, simply transform (x,y,z,0).
- For perspective transformation, more complicated
- For normal transformation, special case

Transforming Normals

- It' s tempting to think of normal vectors as being like porcupine quills, so they would transform like points
- But it' s not so --- consider the 2D example affine transformation below.
- We need a different rule to transform normals.


Transformed shape with normals treated as points
Original Shape shown with normals
Transformed shape with correct normals

Normals Do Not Transform Like Points

- If M is a 4x4 transformation matrix, then
  - To transform points, use p’=Mp, where p=[x y z 1]^T
  - *So to transform normals,* ***n’=Mn****, where* ***n****=[a b c 1]^T right?*
  - Wrong! This formula doesn’t work for general M.

Normals Transform Like Planes

A plane $ax + by + cz + d = 0$ can be written

$\mathbf{n} \cdot \mathbf{p} = \mathbf{n}^T\mathbf{p} = 0$, where $\mathbf{n} = \begin{bmatrix} a & b & c & d \end{bmatrix}^T$, $\mathbf{p} = \begin{bmatrix} x & y & z & 1 \end{bmatrix}^T$

(a,b,c) is the plane normal, d is the offset.

If $\mathbf{p}$ is transformed, how should $\mathbf{n}$ transform?

To find the answer, do some magic :

$$\begin{aligned} 0 &= \mathbf{n}^T\mathbf{I}\mathbf{p} && \text{equation for point on plane in original space} \\ &= \mathbf{n}^T(\mathbf{M}^{-1}\mathbf{M})\mathbf{p} \\ &= (\mathbf{n}^T\mathbf{M}^{-1})(\mathbf{M}\mathbf{p}) \\ &= \mathbf{n}'^T\mathbf{p}' && \text{equation for point on plane in transformed space} \end{aligned}$$

$\mathbf{p}' = \mathbf{M}\mathbf{p}$ to transform point

$\mathbf{n}' = (\mathbf{n}^T\mathbf{M}^{-1})^T = \mathbf{M}^{-1^T}\mathbf{n}$ to transform plane

Transforming Normals - Cases

- For general transformations M that include perspective, use full formula (M inverse transpose), use the right *d*
  - *d* matters, because parallel planes do not transform to parallel planes in this case
- For affine transformations, *d* is irrelevant, can use *d*=0.
- For rotations, M inverse transpose = M, can transform normals and points with same formula.

Quaternions

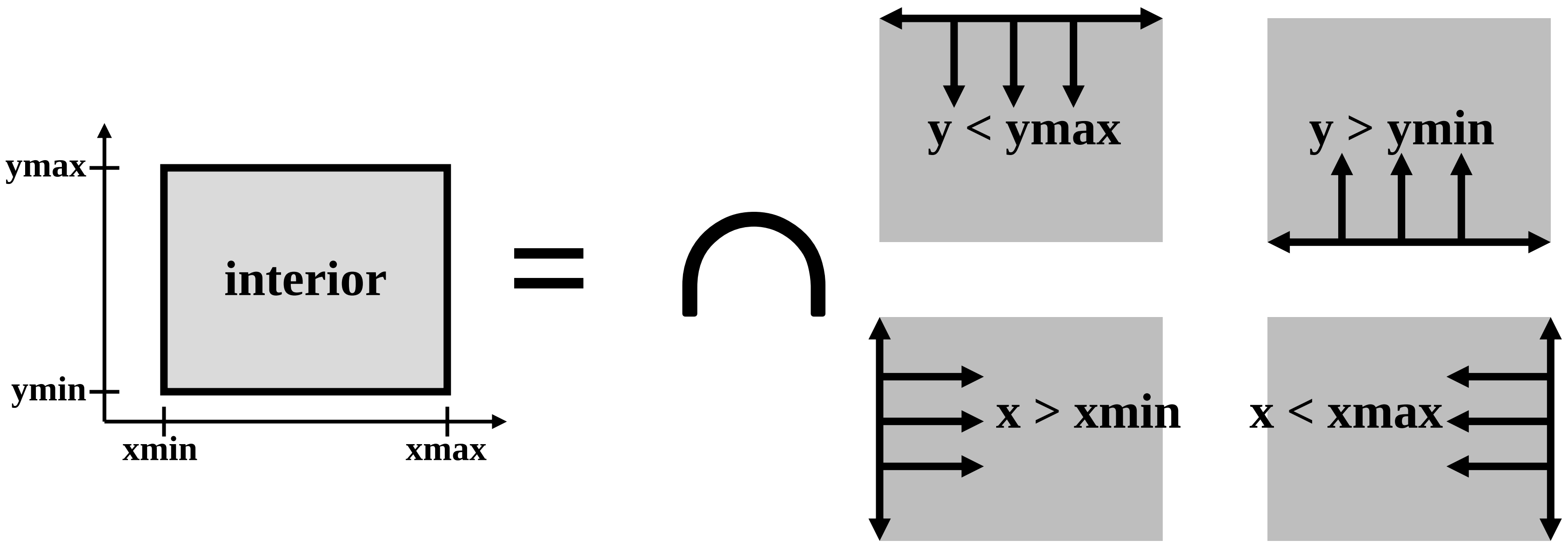
- The rotations are the *unit quaternions*.
- Quaternions, a generalization of complex numbers, can represent 3-D rotations
  - $a+b\mathbf{i}+c\mathbf{j}+d\mathbf{k}$ where $a,b,c,d\in\mathbf{R}$ and $a^2+b^2+c^2+d^2=1$
- Example: rotation by α about the unit vector $[b\ \ c\ \ d]^T$:
  - $\cos\frac{\alpha}{2}+b\sin\frac{\alpha}{2}\mathbf{i}+c\sin\frac{\alpha}{2}\mathbf{j}+d\sin\frac{\alpha}{2}\mathbf{k}$
- Successive rotations corresponds to multiplying quaternions based on distributive law and rules:
  - $\mathbf{i}^2+\mathbf{j}^2+\mathbf{k}^2=-1, \mathbf{ij}=\mathbf{k}=-\mathbf{ji}, \mathbf{jk}=\mathbf{i}=-\mathbf{kj}, \mathbf{ki}=\mathbf{j}=-\mathbf{ik}.$
- A unit quaternion represents a point on the unit sphere in 4D.
  - Interpolation: shortest path between two points on the sphere (*a great arc*)

Quaternions

- Advantages:
  - no trigonometry required
  - multiplying quaternions gives another rotation (quaternion)
  - rotation matrices can be calculated from them
  - direct rotation (with no matrix)
  - no favored direction or axis
- Disadvantages:
  - $R_{\vec{v}}(\alpha) = R_{-\vec{v}}(-\alpha)$
    but, $Quaternion(R_{\vec{v}}(\alpha)) \neq Quaternion(R_{-\vec{v}}(-\alpha))$
  - $R_{\vec{v}}(0^\circ) \neq R_{\vec{v}}(360^\circ)$
    but, $Quaternion(R_{\vec{v}}(0^\circ)) = Quaternion(R_{\vec{v}}(360^\circ)) = (1 + 0\mathbf{i} + 0\mathbf{j} + 0\mathbf{k})$

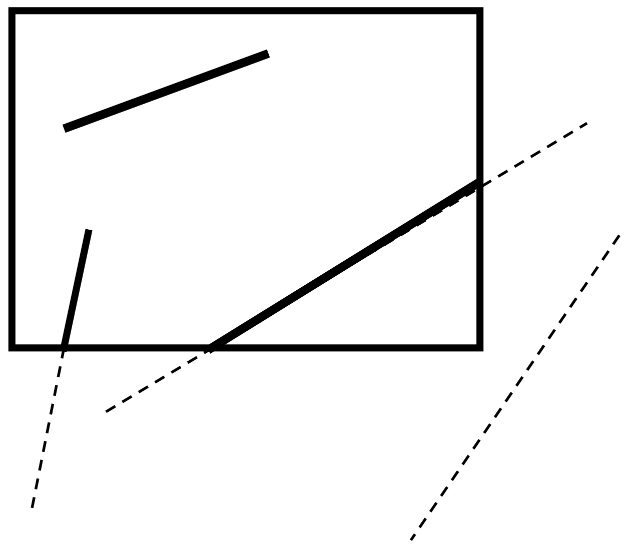
Line Clipping

- Modify endpoints of lines to lie in rectangle
- How to define "interior" of rectangle?
- Convenient def.: intersection of 4 half-planes
  - Nice way to decompose the problem
  - Generalizes easily to 3D (intersection of 6 half-planes)



Line Clipping

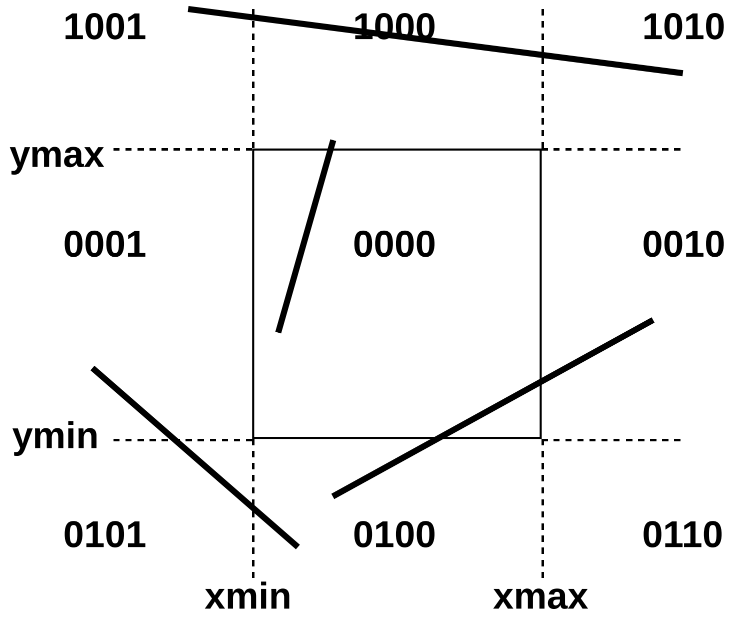
- Modify end points of lines to lie in rectangle
- Method:
  - Is end-point inside the clip region? (half-plane tests)
  - If outside, calculate intersection between the line and the clipping rectangle and make this the new end point



- Both endpoints inside: trivial accept
- One inside: find intersection and clip
- Both outside: either clip or reject (tricky case)

Cohen-Sutherland Algorithm

- Uses *outcodes* to encode the half-plane tests results



bit 1: y>ymax
bit 2: y<ymin
bit 3: x>xmax
bit 4: x<xmin

- Rules:
  - Trivial accept: outcode(end1) and outcode(end2) both *zero*
  - Trivial reject: outcode(end1) & (bitwise and) outcode(end2) *nonzero*
  - Else subdivide

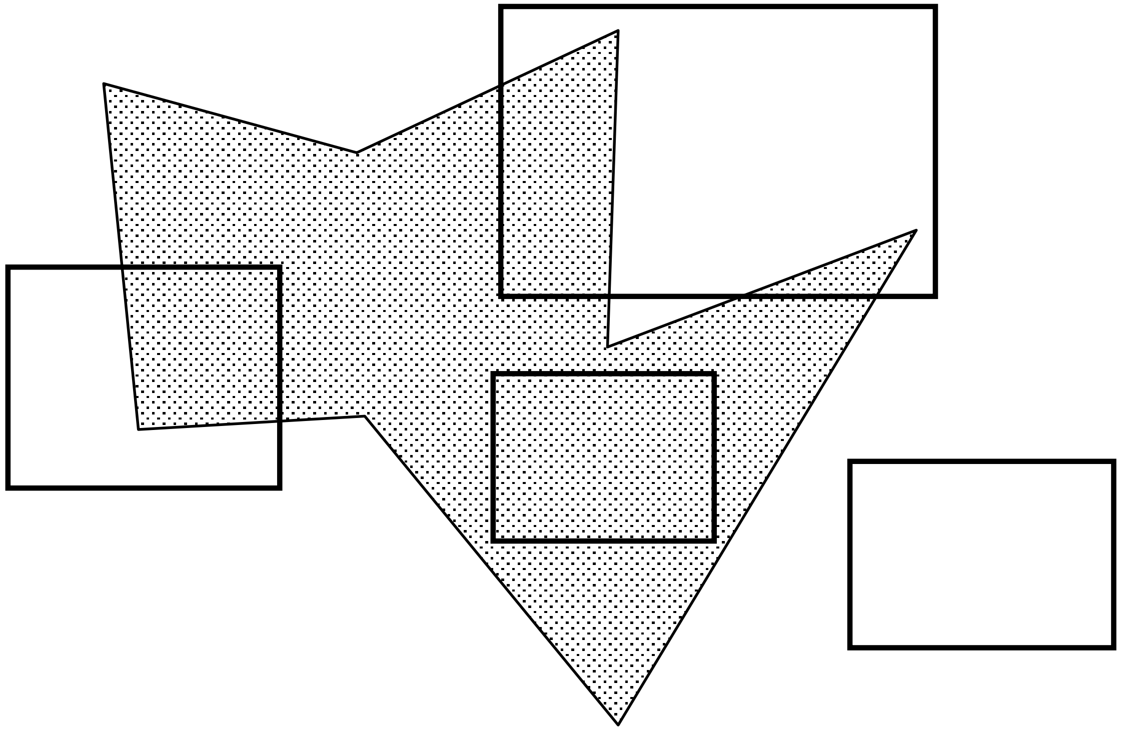
Cohen-Sutherland Algorithm: Subdivision

- If neither trivial accept nor reject:
  - Pick an outside endpoint (with nonzero outcode)
  - Pick an edge that is crossed (nonzero bit of outcode)
  - Find line's intersection with that edge
  - Replace outside endpoint with intersection point
  - Repeat until trivial accept or reject

- Other clipping algorithms
  - Cyrus-Beck/Liang-Barksy or Nicholl-Lee-Nicholl

Polygon Clipping

Convert a polygon into one *or more* polygons that form the intersection of the original with the clip window



Sutherland-Hodgman Polygon Clipping Algorithm

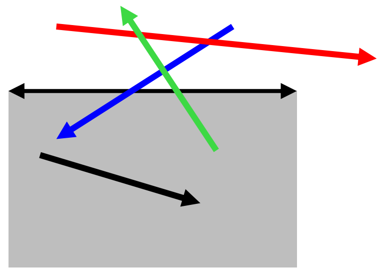
- Subproblem:
  - clip a polygon (vertex list) against a single clip plane
  - output the vertex list(s) for the resulting clipped polygon(s)

- Clip against all four planes
  - generalizes to 3D (6 planes)
  - generalizes to any convex clip polygon/polyhedron

Sutherland-Hodgman Polygon Clipping Algorithm (Cont.)

- To clip vertex list against one half-plane:
  - if first vertex is inside - output it
  - loop through list testing inside/outside transition - output depends on transition:

> **in-to-in:**	**output vertex**
> **out-to-out:**	**no output**
> **in-to-out:**	**output intersection**
> **out-to-in:**	**output intersection and vertex**



Summary

- Started with orthographic projection: just throw out the Z coordinate
- Perspective projection from origin along Z axis: use projection matrix
- Moving the camera: transform the entire world so that we can do projection from the origin along the Z axis
- Screen coordinates: translate and scale entire world so that projection yields pixel coordinates
- Clipping: transform world so that viewing frustum becomes a unit cube. Clip lines against half-planes.