CS354R

DR SARAH ABRAHAM

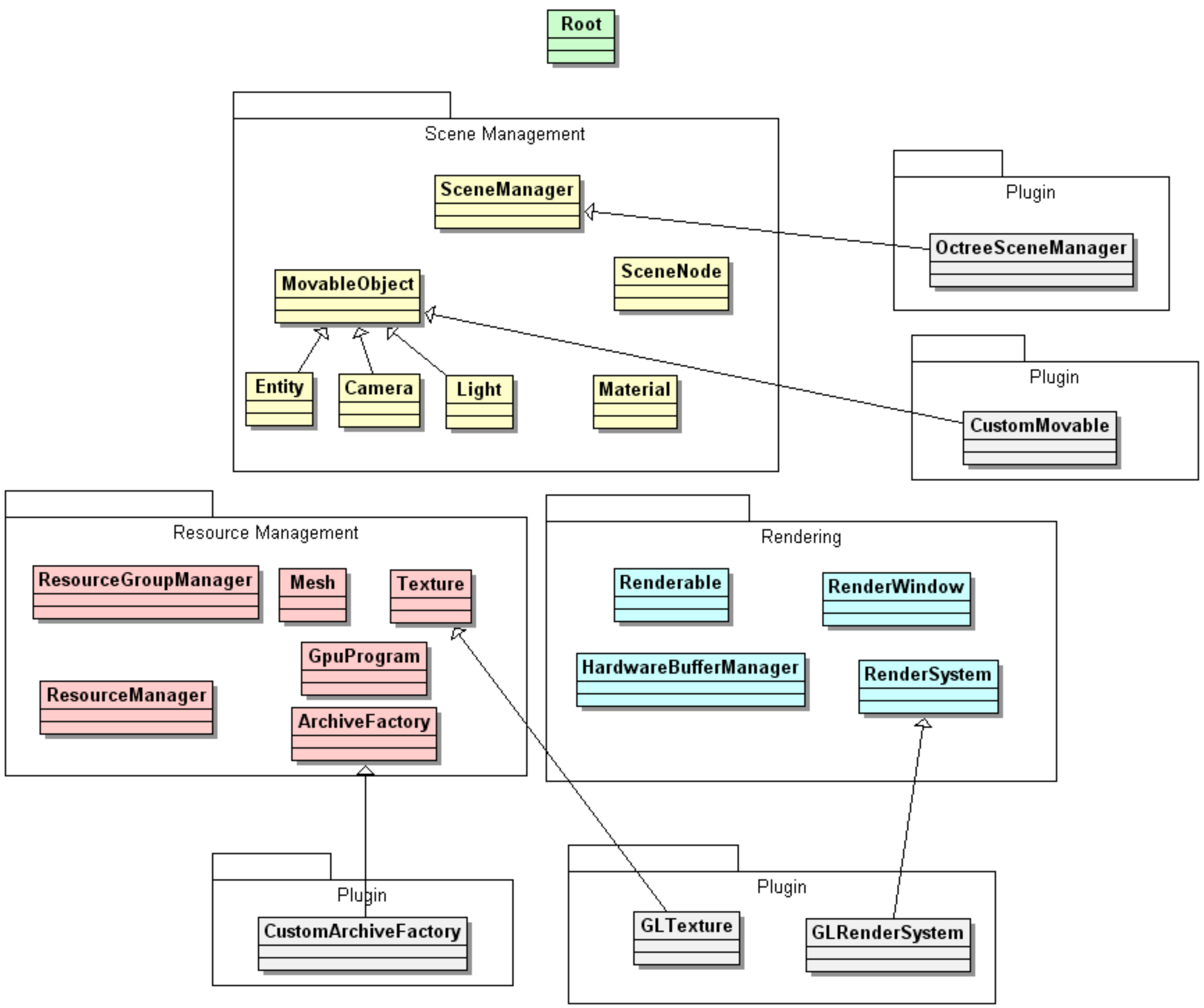
3D ENGINES AND SCENE GRAPHS

3D GRAPHICS ENGINES

- What is a 3D graphics engine and what should it include?

3D ENGINES

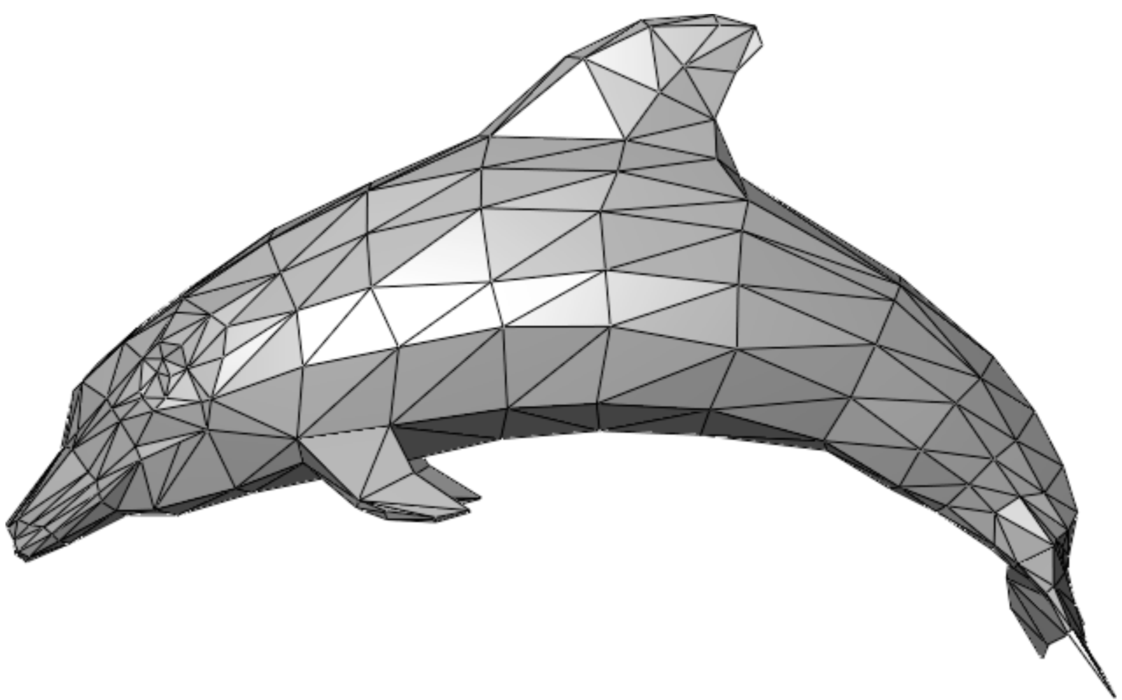
- Handles functionality related to graphics and rendering
- The “graphics” part of a game engine

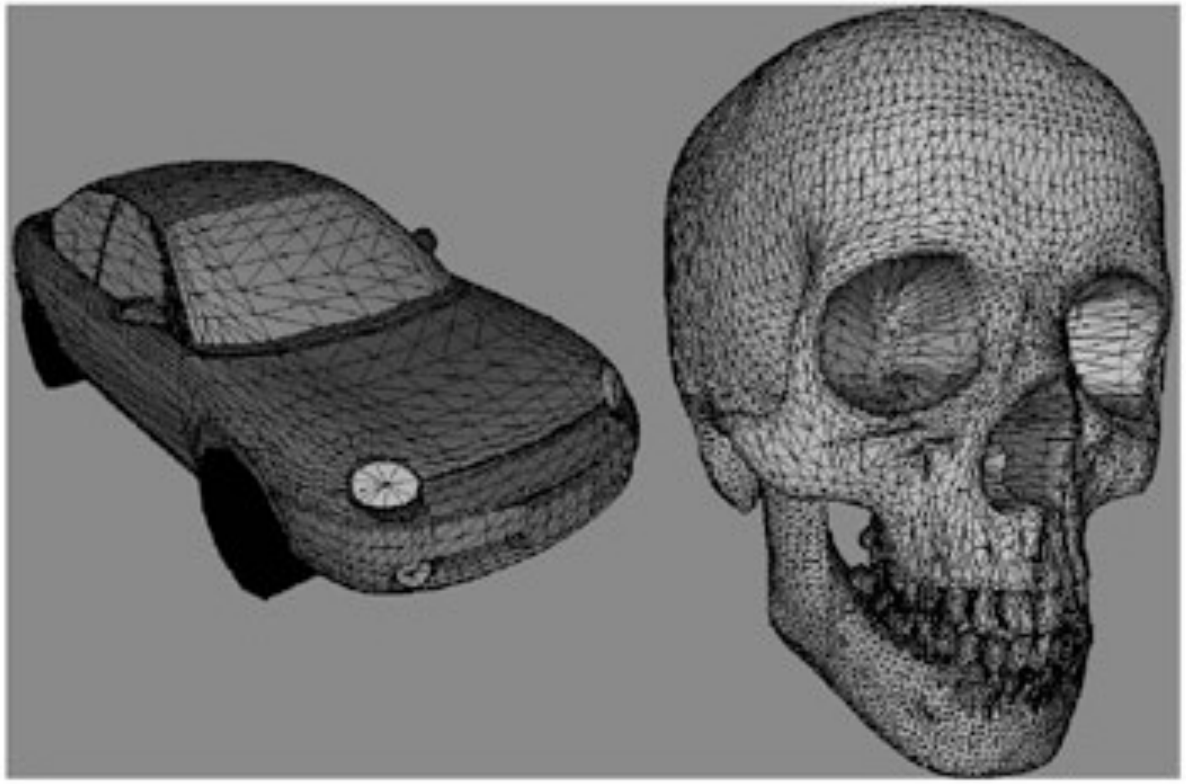


Ogre 1.9 Core class structure

WHAT ARE THE OBJECTS?

- Geometry - polygon (triangle, quad) meshes
  - Vertices form edges
  - Edges form faces





OBJECTS OF INCREASING COMPLEXITY…

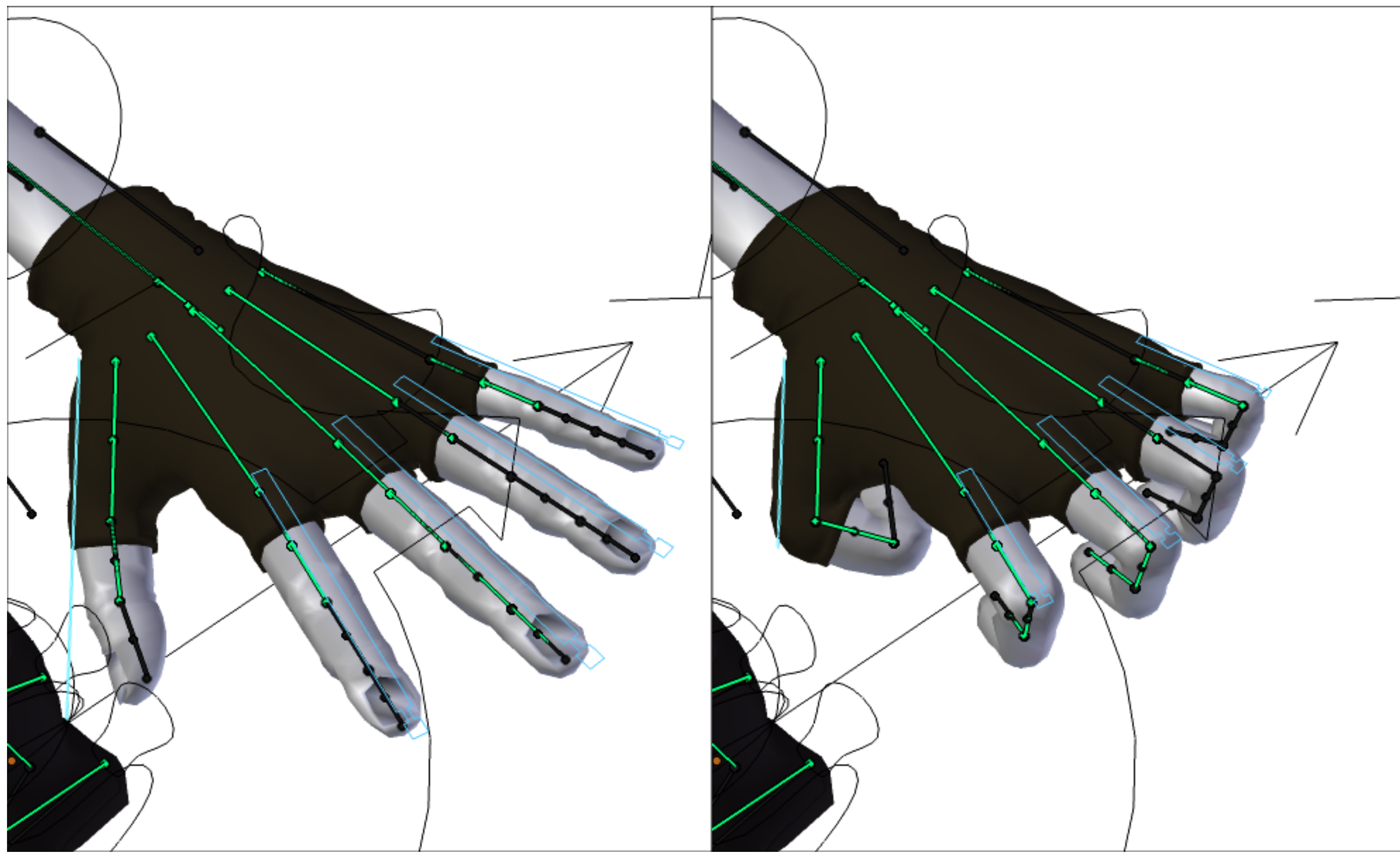
Monster Hunter World

HIERARCHICAL MODELING

- Ways character can move:
  - Move the whole character wrt the world
  - Move legs, arms, head wrt body
  - Move hands wrt arms
  - Move upper vs. lower arm
  - Same for legs

THE HIGHER LEVEL (3D MODELED OBJECTS)

- Modeling
- **Rigging**
- Skinning
- Animating



Wikipedia (Skeletal Animation)

THE LOWER LEVEL (SYMBOLS AND INSTANCES)

- Most graphics APIs support a few geometric **primitives**:
  - Spheres
  - Cubes
  - Triangles
- These symbols are **instanced** using an **instance transformation**.

TRANSFORMATION REPRESENTATION

- We can represent a 2D point, p = (x, y), in the plane as a column vector:

$$\begin{bmatrix} x \\ y \end{bmatrix}$$

- We can represent a 2-D transformation M by a matrix:

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

$$\mathbf{p}' = \mathbf{M}\mathbf{p}$$

- If p is a column vector, M goes on the left:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

2D TRANSFORMATIONS

- Here's all you get with a 2x2 transformation matrix **M**:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

- So:

$$x' = ax + by$$
$$y' = cx + dy$$

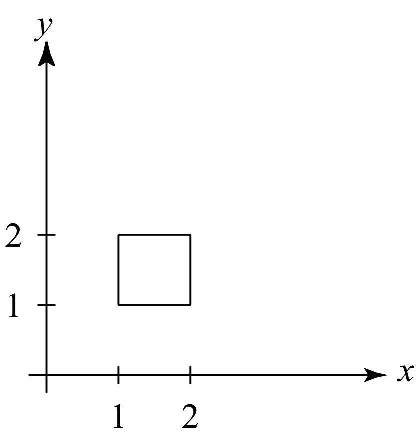
IDENTITY

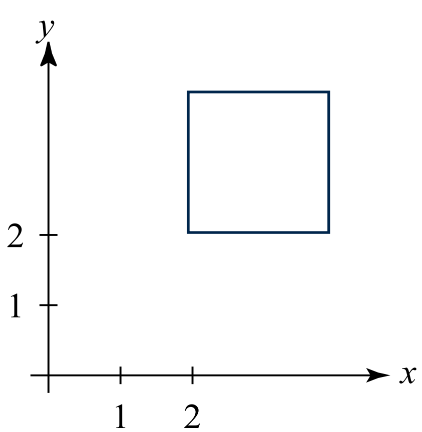
- Suppose we choose a = d = 1, b = c = 0:
- Gives the identity matrix: $\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$
- Doesn't move the point at all

SCALING

- Suppose b = c = 0, but let a and d take on any positive value
- Gives a scaling matrix:

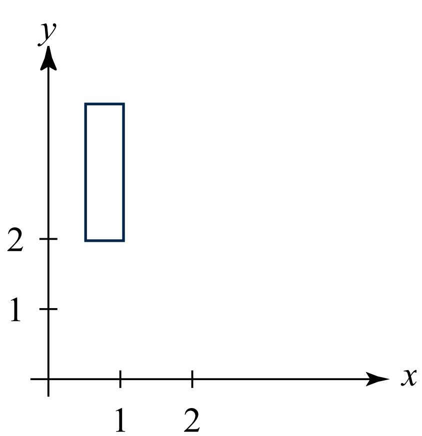
$$\begin{bmatrix} a & 0 \\ 0 & d \end{bmatrix} \qquad \begin{aligned} x' &= ax \\ y' &= dy \end{aligned}$$





$$\begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix}$$

Can have differential (non-uniform) scaling in x and y

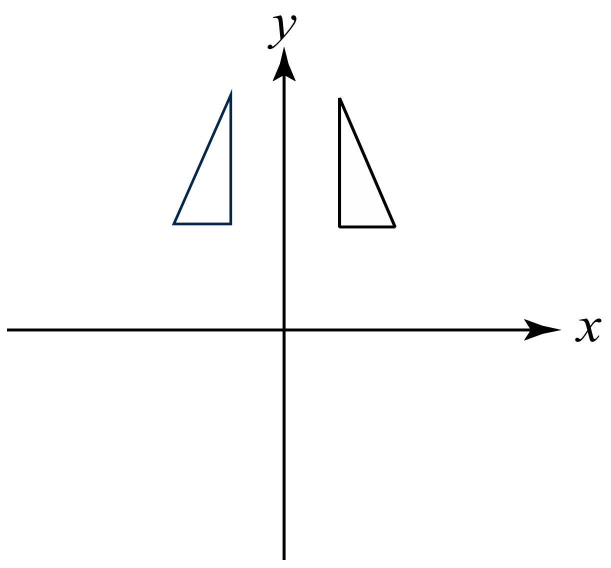


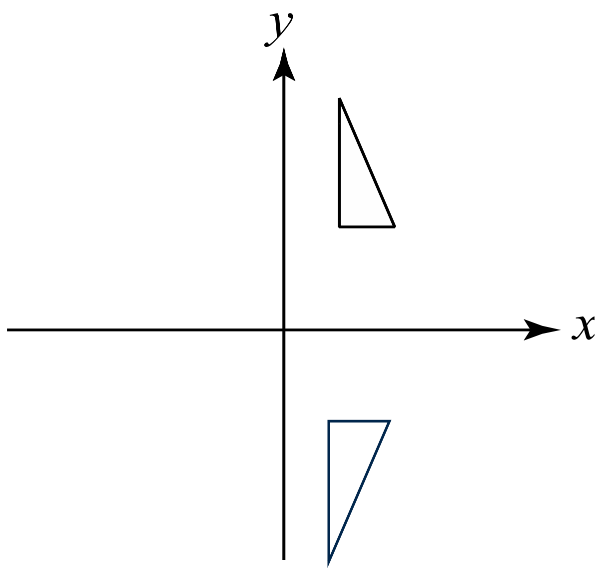
$$\begin{bmatrix} 1/2 & 0 \\ 0 & 2 \end{bmatrix}$$

REFLECTION

- Suppose b = c = 0, but either a or d goes negative
- Consider:

$$\begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix}$$





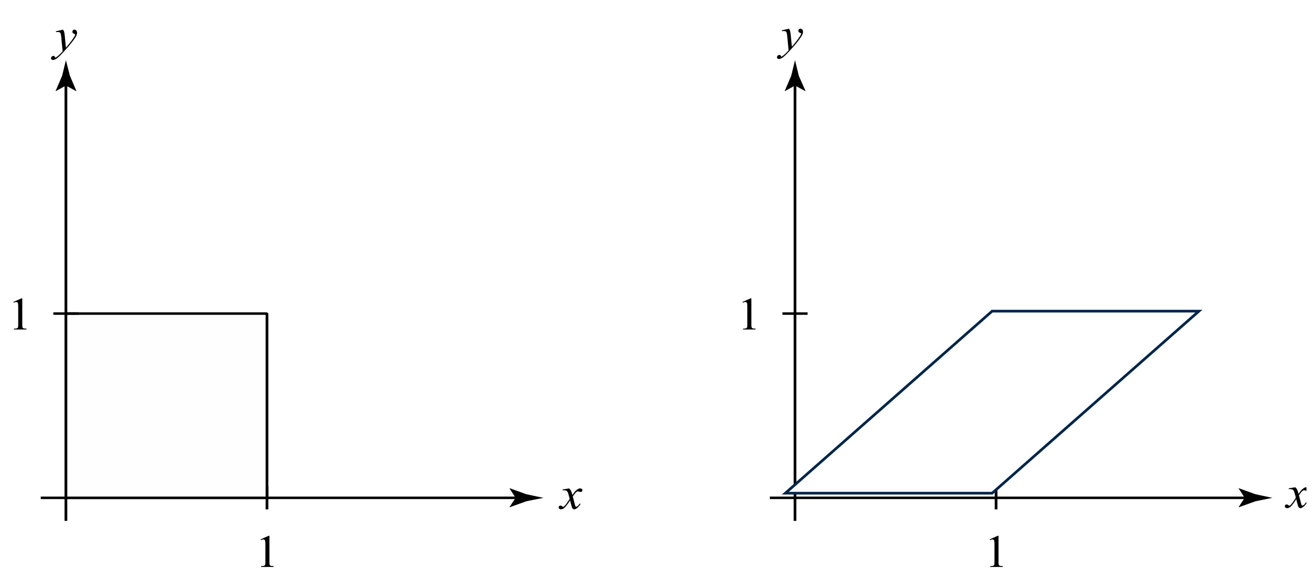
$$\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

SHEAR

- Now leave a = d = 1 and experiment with b

$$\begin{bmatrix} 1 & b \\ 0 & 1 \end{bmatrix} \qquad \begin{aligned} x' &= x + by \\ y' &= y \end{aligned}$$

- Consider:



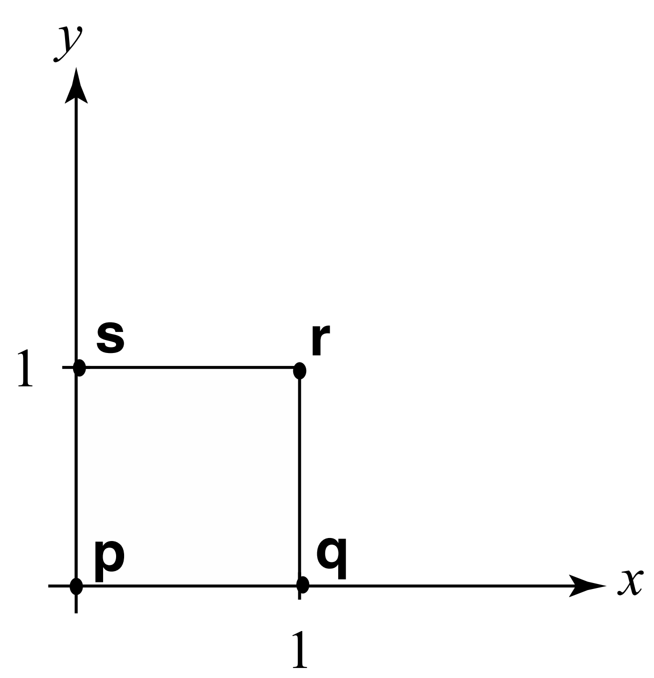
$$\begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$$

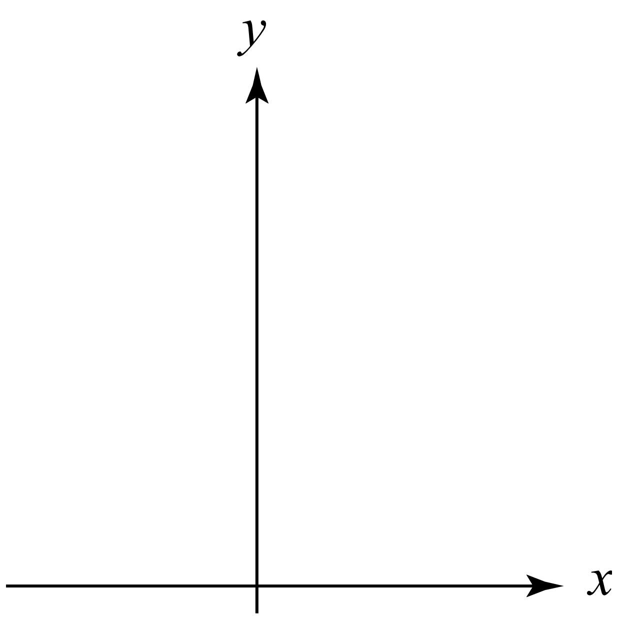
EFFECT ON UNIT SQUARE

- A general 2 x 2 transformation M on the unit square:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} \mathbf{p} & \mathbf{q} & \mathbf{r} & \mathbf{s} \end{bmatrix} = \begin{bmatrix} \mathbf{p}' & \mathbf{q}' & \mathbf{r}' & \mathbf{s}' \end{bmatrix}$$

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 0 & a & a+b & b \\ 0 & c & c+d & d \end{bmatrix}$$



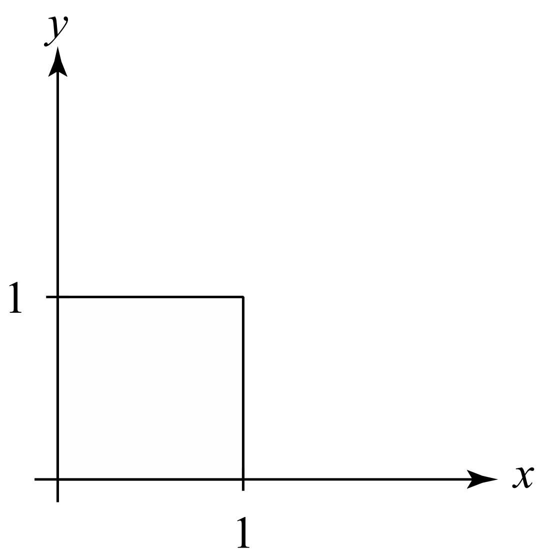


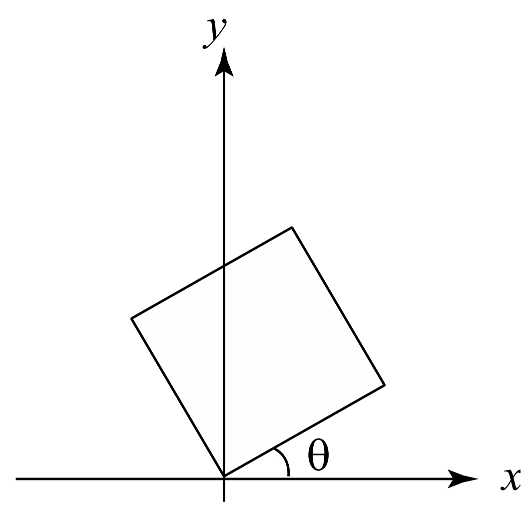
OBSERVATIONS

- Origin invariant under **M**
- **M** can be determined just by knowing how the corners (1,0) and (0,1) are mapped
- *a* and *d* give x- and y-scaling
- *b* and *c* give x- and y-shearing

ROTATION

- From our observations of the effect on the unit square, the matrix for "rotation about the origin":





$$\begin{bmatrix} 1 \\ 0 \end{bmatrix} \rightarrow \begin{bmatrix} \cos(\theta) \\ \sin(\theta) \end{bmatrix}$$

$$\begin{bmatrix} 0 \\ 1 \end{bmatrix} \rightarrow \begin{bmatrix} -\sin(\theta) \\ \cos(\theta) \end{bmatrix}$$

- Thus:

$$M_R = R(\theta) = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix}$$

LINEAR TRANSFORMATIONS

- The unit square observations suggest the 2x2 matrix transformation is representing a point in a new coordinate system:

$$\mathbf{p}' = \mathbf{M}\mathbf{p}$$
$$= \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$
$$= \begin{bmatrix} \mathbf{u} & \mathbf{v} \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$
$$= x \cdot \mathbf{u} + y \cdot \mathbf{v}$$

- where **u** = $[a\ c]^T$ and **v** = $[b\ d]^T$ are vectors that define a new basis for a **linear space**.
- The transformation to this new basis (a.k.a., change of basis) is a **linear transformation**.

LIMITATIONS OF THE 2X2 MATRIX

- A 2x2 linear transformation matrix allows:
  - Scaling
  - Rotation
  - Reflection
  - Shearing

- What important operation does that leave out?

AFFINE TRANSFORMATIONS

- In order to incorporate the idea that both the basis and the origin can change, we augment the linear space **u**, **v** with an origin **t**.
- Note that while **u** and **v** are basis vectors, the origin **t** is a point.
- We call **u**, **v**, and **t** (basis and origin) a **frame** for an **affine space**.
- Then, we can represent a change of frame as:

$$\mathbf{p}' = x \cdot \mathbf{u} + y \cdot \mathbf{v} + \mathbf{t}$$

- This change of frame is also known as an **affine transformation**.

HOMOGENEOUS COORDINATES

- To represent transformations among affine frames, we can loft the problem up into 3-space, adding a third component to every point:
- Note that:
  - $[a\ c\ 0]^T$ and $[b\ d\ 0]^T$ represent vectors
  - $[t_x\ t_y\ 1]^T$, $[x\ y\ 1]^T$ and $[x'\ y'\ 1]^T$ represent points.

$$\mathbf{p}' = \mathbf{M}\mathbf{p}$$

$$= \begin{bmatrix} a & b & t_x \\ c & d & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

$$= \begin{bmatrix} \mathbf{u} & \mathbf{v} & \mathbf{t} \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

$$= x \cdot \mathbf{u} + y \cdot \mathbf{v} + 1 \cdot \mathbf{t}$$

HOMOGENEOUS COORDINATES

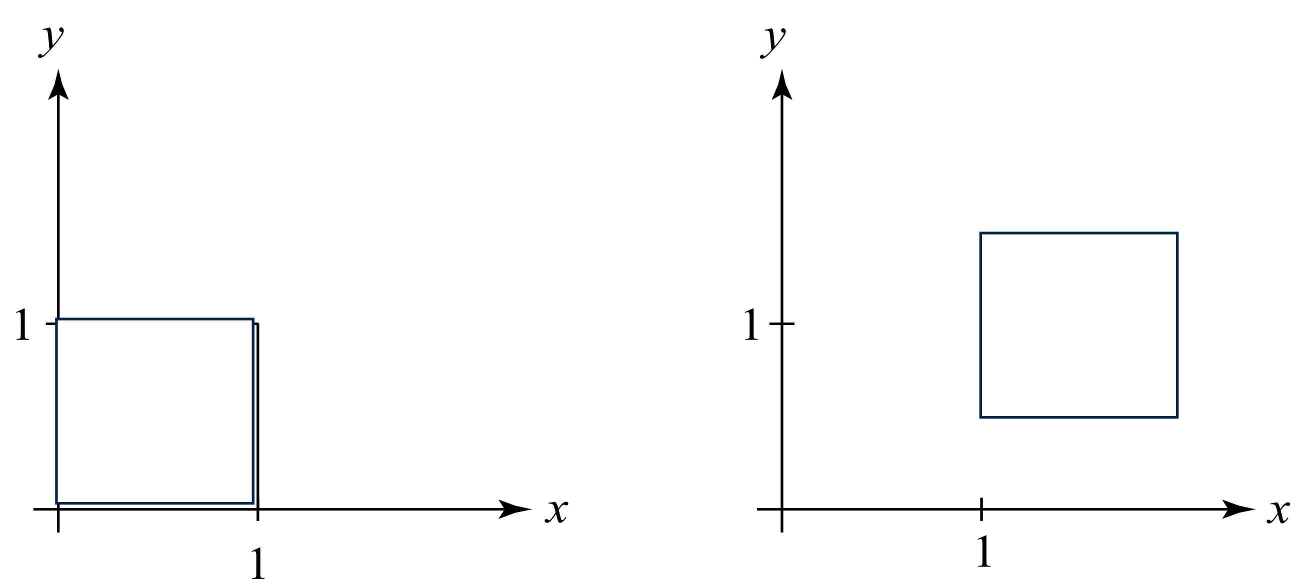
- This allows us to perform translation as well as the linear transformations as a matrix operation:

$$\mathbf{p}' = \mathbf{M}_T\mathbf{p}$$

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

$$x' = x + t_x$$

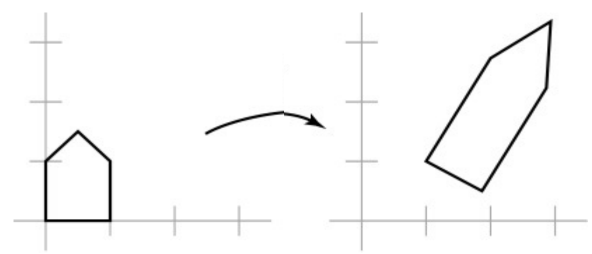
$$y' = y + t_y$$



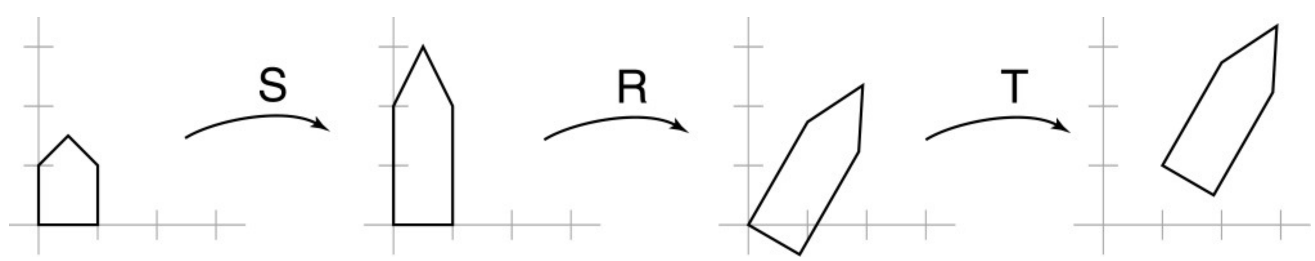
$$\begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 1/2 \\ 0 & 0 & 1 \end{bmatrix}$$

USE A SERIES OF TRANSFORMATIONS

- A particular geometric instance is transformed by one combined transformation matrix:



- But it's convenient to build this single matrix from a series of simpler transformations:



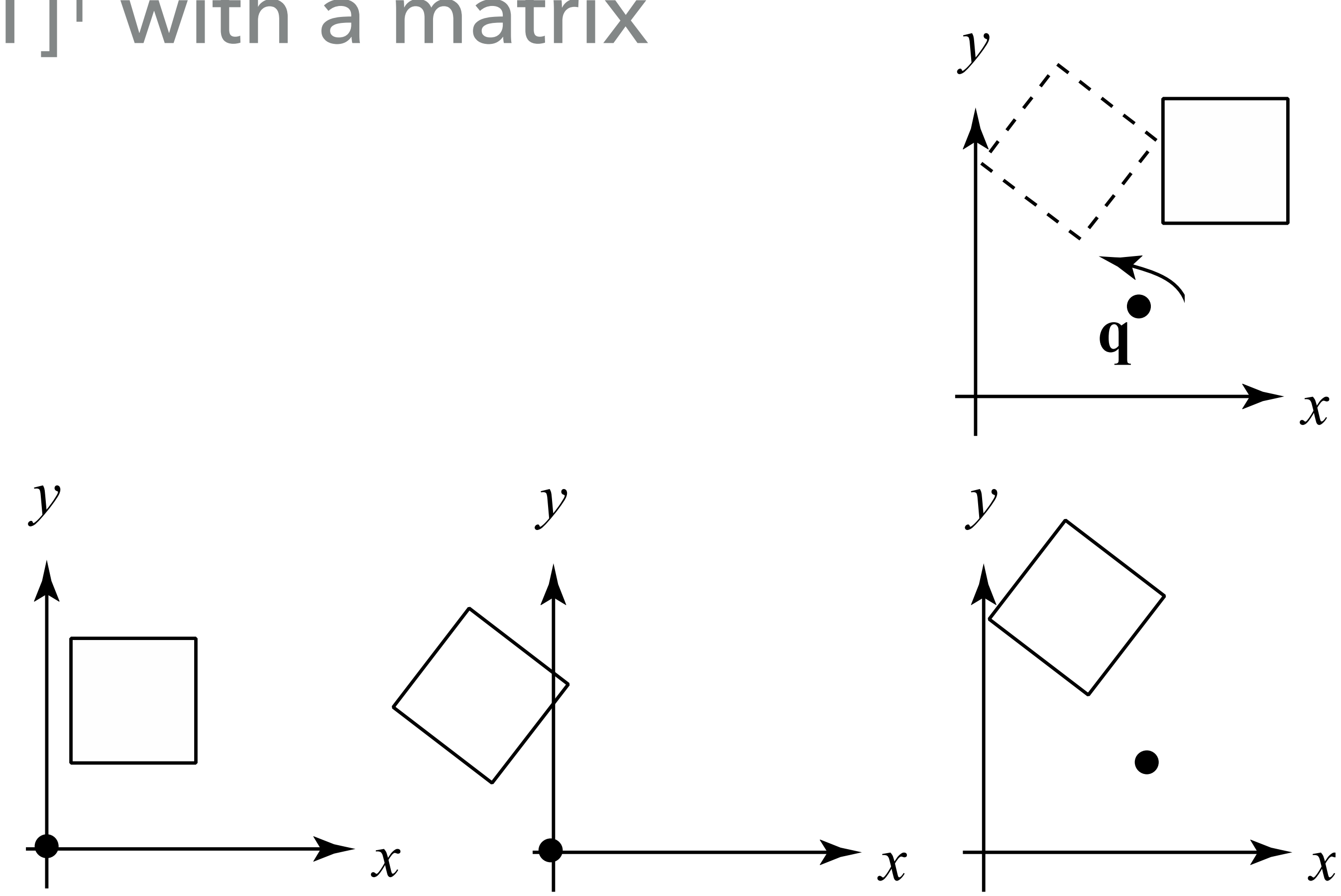
- We have to be careful about how we think about composing these transformations.

  (Mathematical reason: Transformation matrices don't commute under matrix multiplication!)

ROTATION ABOUT ARBITRARY POINTS

Until now, we've only considered rotation about the origin

With homogeneous coordinates, you can specify a rotation **Rq** about any point **q** = $[q_x\ q_y\ 1]^T$ with a matrix

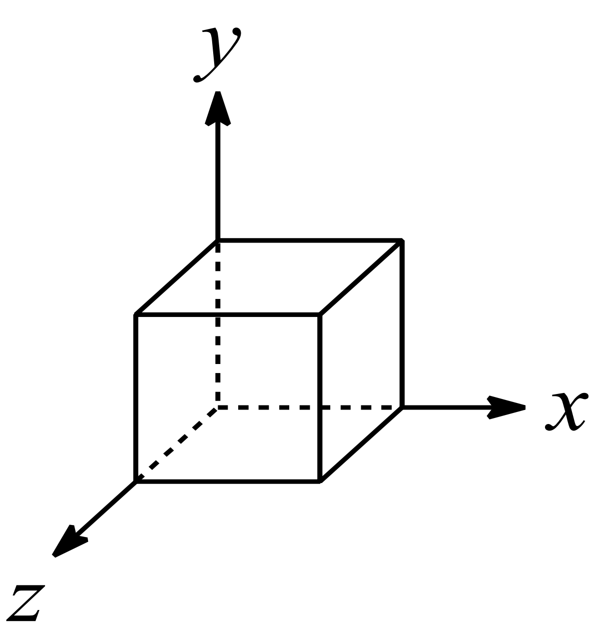


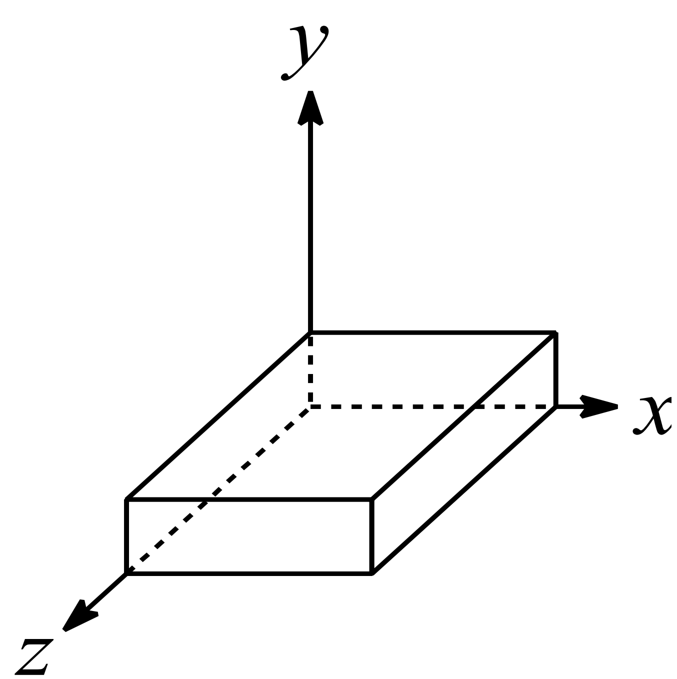
1. Translate **q** to origin
2. Rotate
3. Translate back

Note: Line up the matrices for these steps in right to left order and multiply (this is why transformation order matters!)

SCALING IN 3D

- Some of the 3-D transformations are just like the 2-D ones.
- For example, scaling:

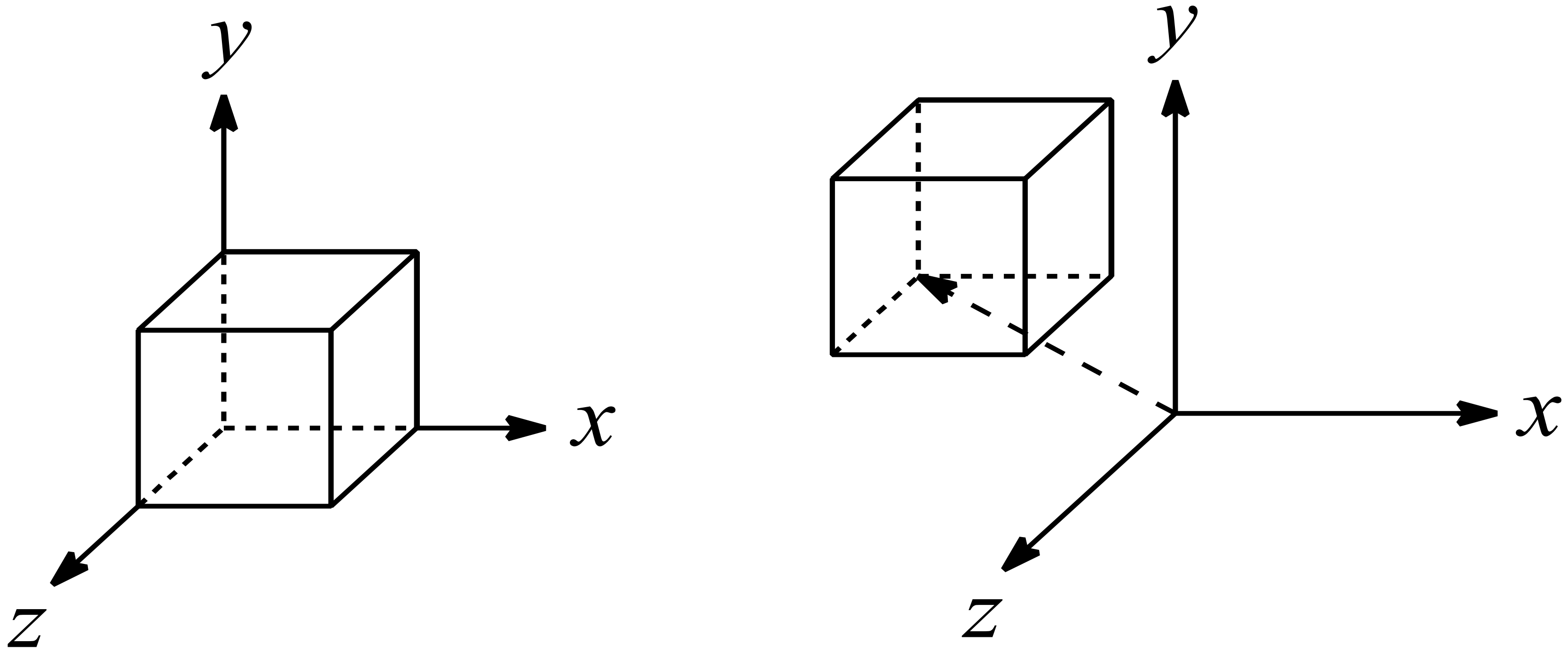




$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

TRANSLATION IN 3D

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



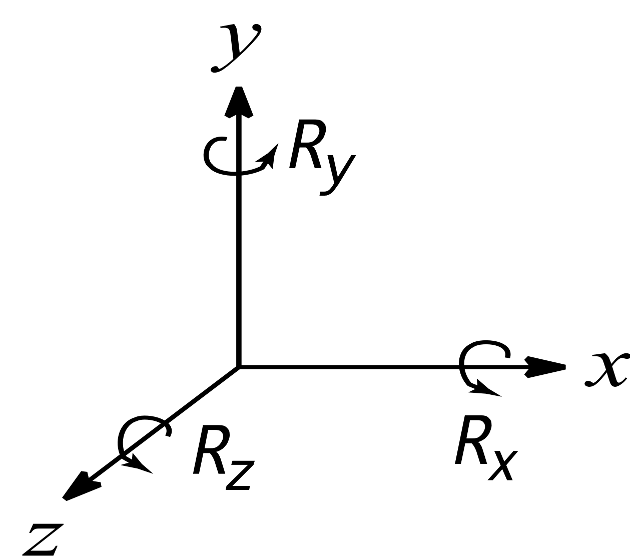
ROTATION IN 3D

- Rotation now has more possibilities in 3D:

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

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

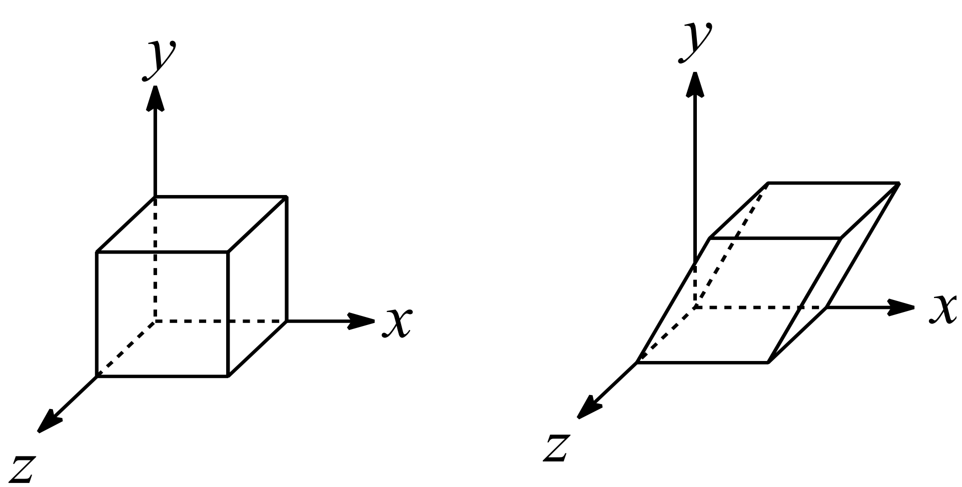
$$R_z(\theta) = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 & 0 \\ \sin(\theta) & \cos(\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



Use right hand rule

SHEARING IN 3D

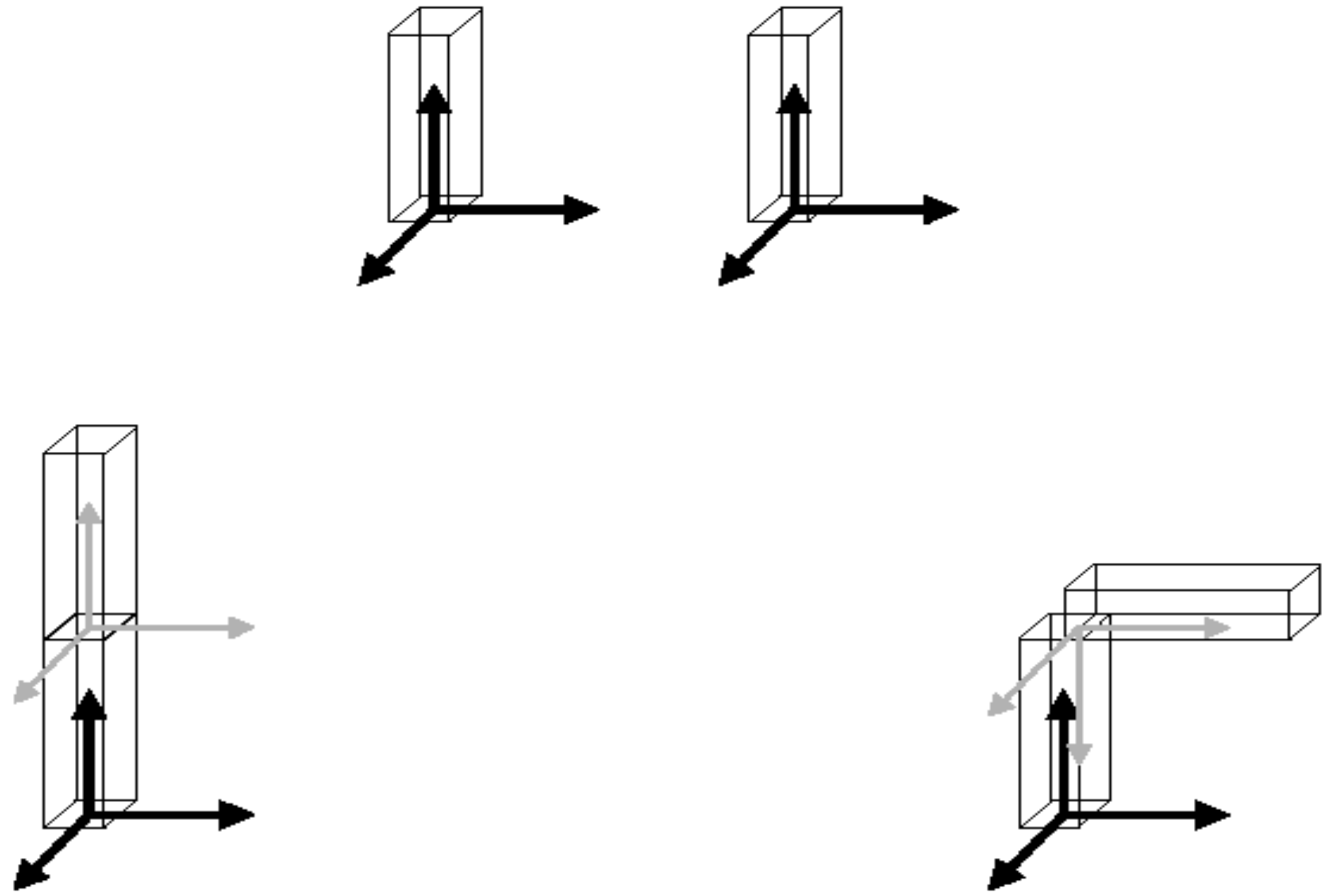
- Shearing is also more complicated. Here is one example:



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

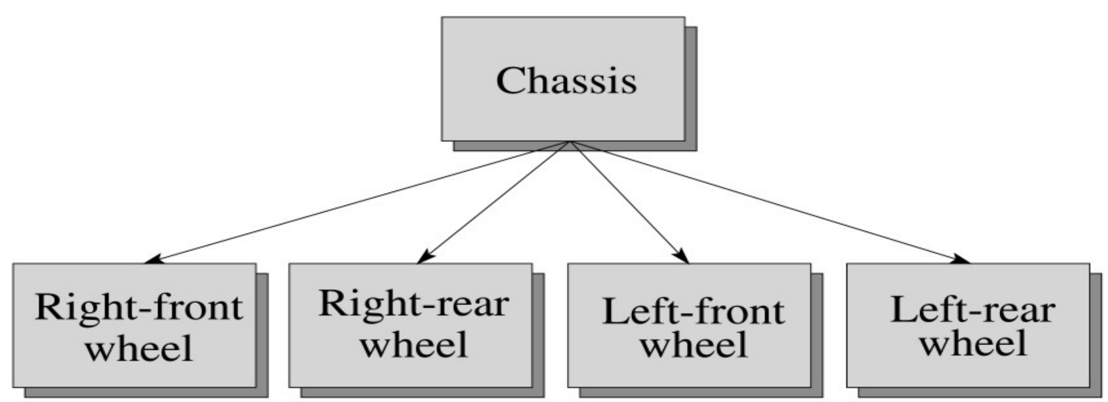
- We call this a shear with respect to the x-z plane.

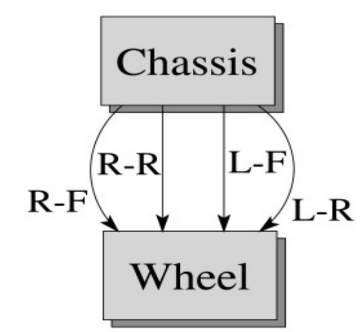
COMBINING TRANSFORMATIONS AND PRIMITIVES



HIERARCHICAL MODELING

- Hierarchical models can be composed of instances using trees or DAGs:

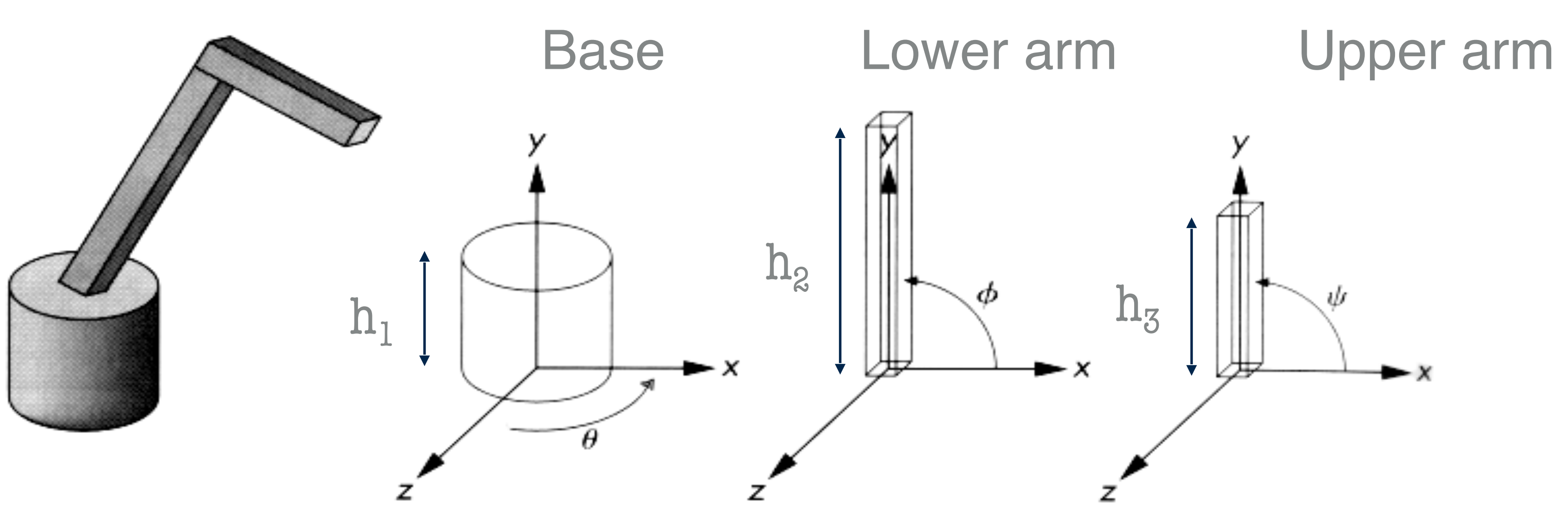




  - Edges contain geometric transformations
  - Nodes contain geometry (and possibly drawing attributes)

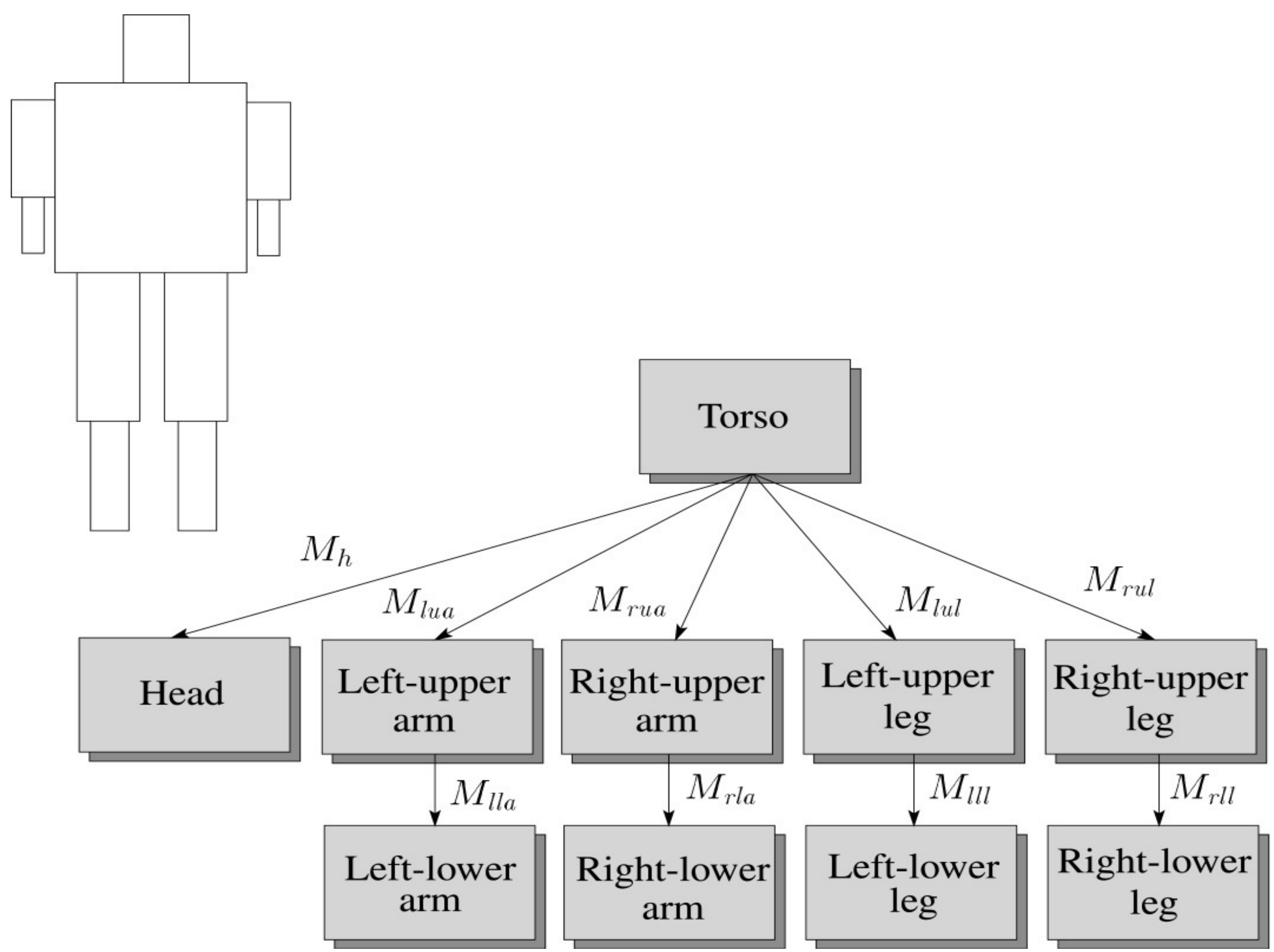
3D EXAMPLE: A ROBOT ARM

- Consider this robot arm with 3 degrees of freedom:
  - Base rotates about its vertical axis by θ
  - Lower arm rotates in its xy-plane by ϕ
  - Upper arm rotates in its xy-plane by ψ
- How might we draw the tree for the robot arm?



A COMPLEX EXAMPLE: HUMAN FIGURE

- What's the most sensible way to traverse this tree?



HUMAN FIGURE IMPLEMENTATION

```
torso();
glPushMatrix();
    glTranslate( ... );
    glRotate( ... );
    head();
glPopMatrix();
glPushMatrix();
    glTranslate( ... );
    glRotate( ... );
    left_upper_arm();
    glPushMatrix();
        glTranslate( ... );
        glRotate( ... );
        left_lower_arm();
    glPopMatrix();
glPopMatrix();
```

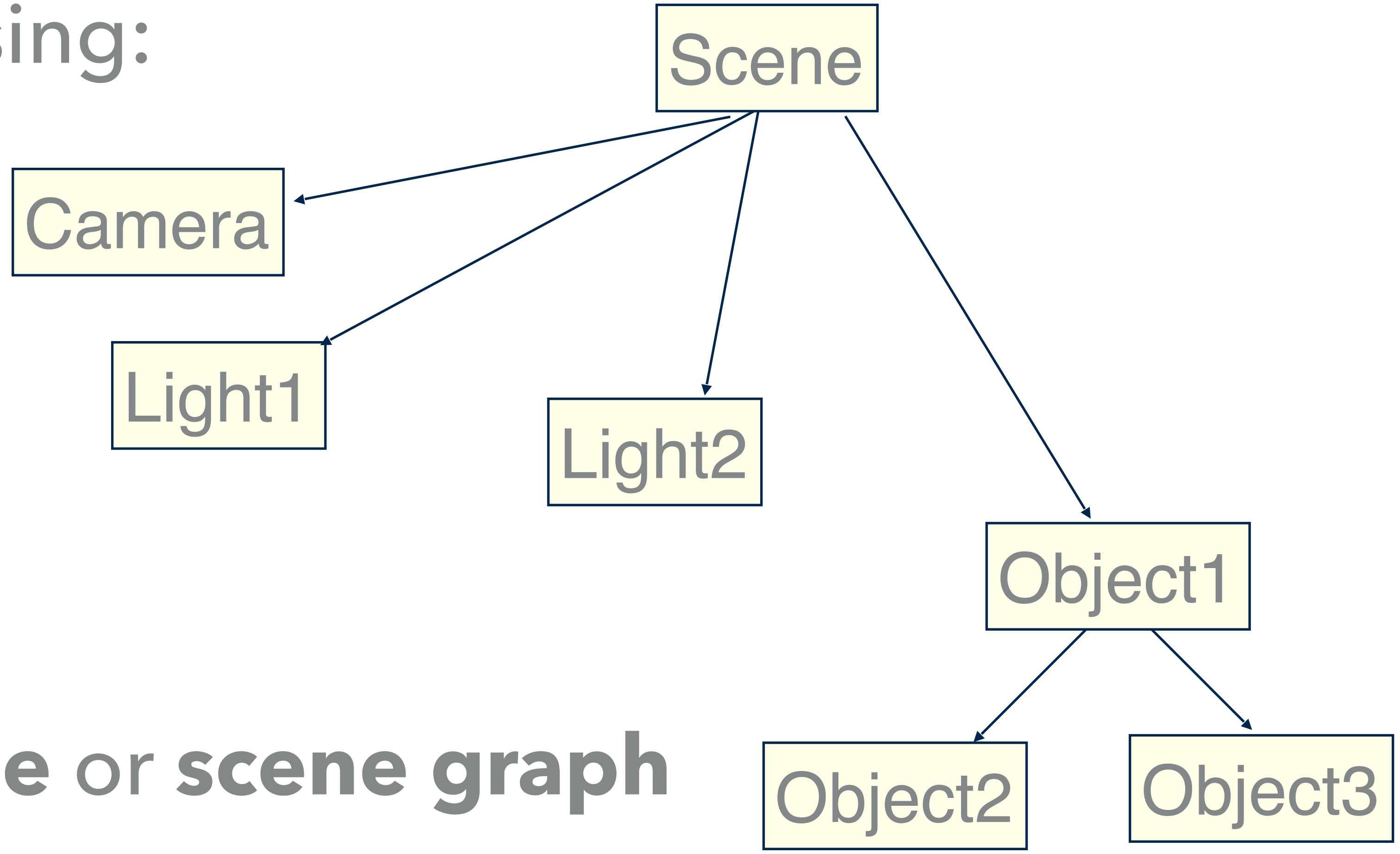
Note: Fixed pipeline OpenGL is outdated but works well for illustrative purposes!

ON OUR WAY TO ANIMATING!

https://youtu.be/vOGhAV-84iI?t=1m45s

SCENE GRAPHS

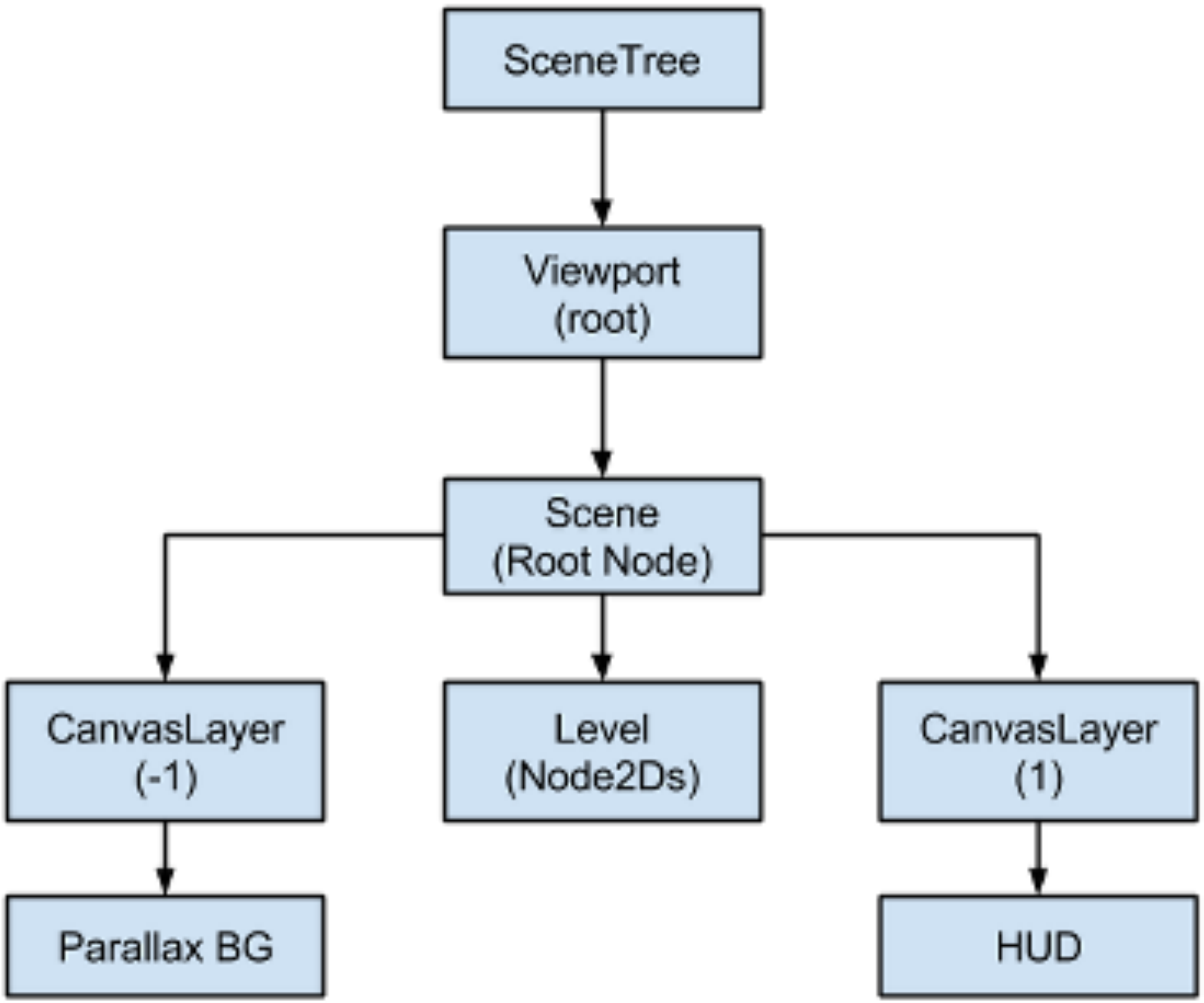
- The idea of hierarchical modeling can be extended to an entire scene, encompassing:
  - Multiple objects
  - Lights
  - Camera position
- This is called a **scene tree** or **scene graph**



SCENE GRAPHS IN GODOT

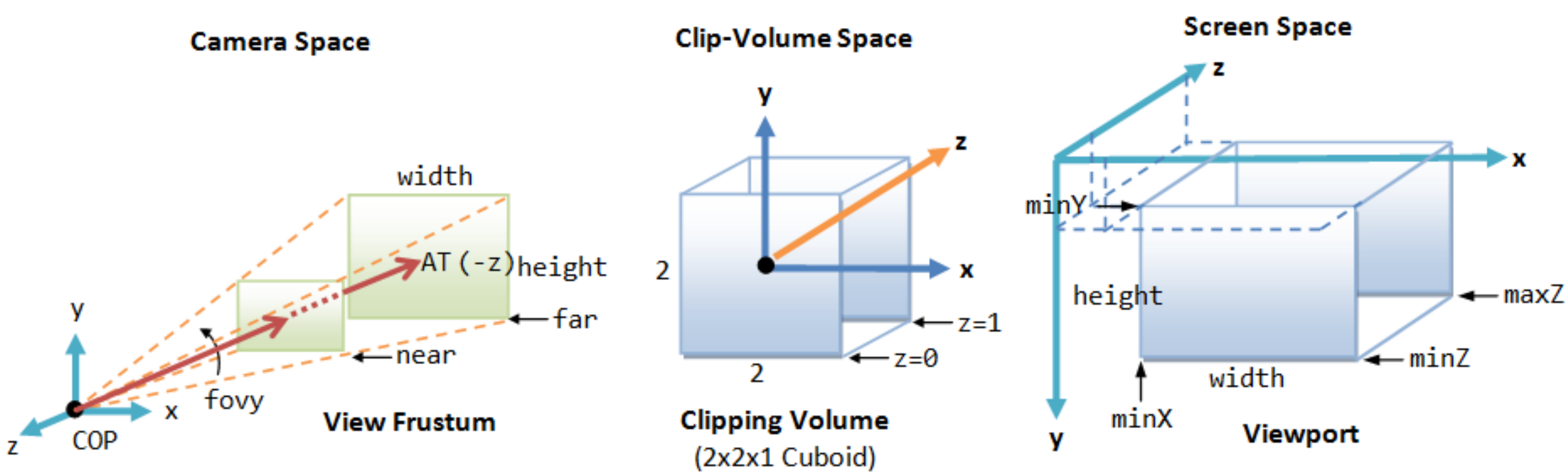
- Godot originally a 2D game engine
  - Added support for 3D in 3.0
- 2D scene graphs built of CanvasItems
  - Control inherits for GUI items
  - Node2Ds used for 2D scene graphs
- 3D scene graphics built on top of Node3Ds
  - `Transform` property is 3x4 matrix
  - 3 `Vector3` properties for translate, rotate, and scale

2D SCENE GRAPH IN GODOT



VIEWPORTS

- Viewports are how scenes are rendered out to a screen
- Allows for easier rendering to multiple screen resolutions



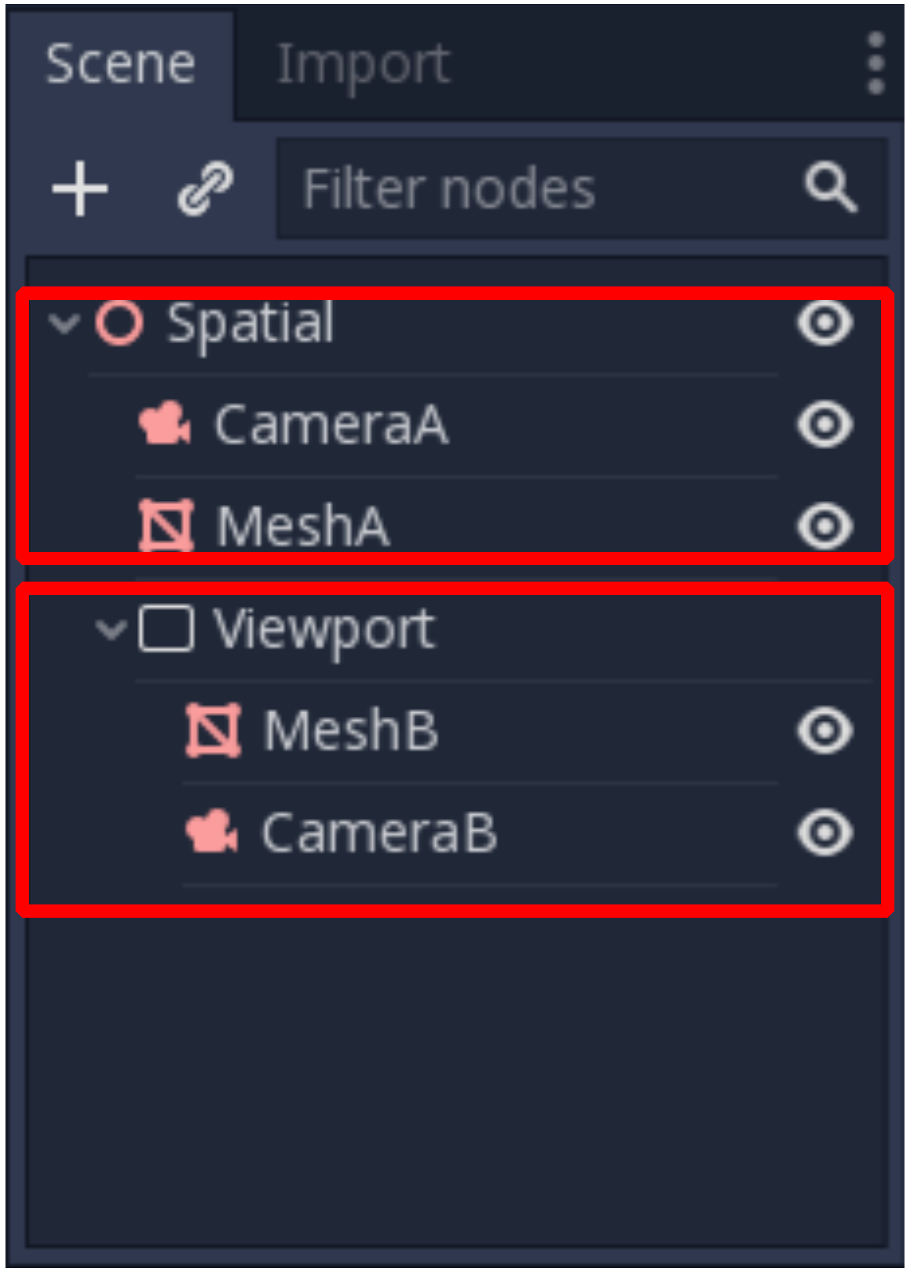
(https://www.ntu.edu.sg/home/ehchua/programming/opengl/CG_Examples.html)

VIEWPORTS IN GAMES

- Game utilize multiple viewports for:
  - Displaying multiple cameras
  - Rendering 2D elements in 3D scenes
  - Rendering to textures
  - etc
- Can add multiple viewports to the scene graphs in Godot
- Viewport Containers help set the outputted viewport size, and connect objects to display with its viewport

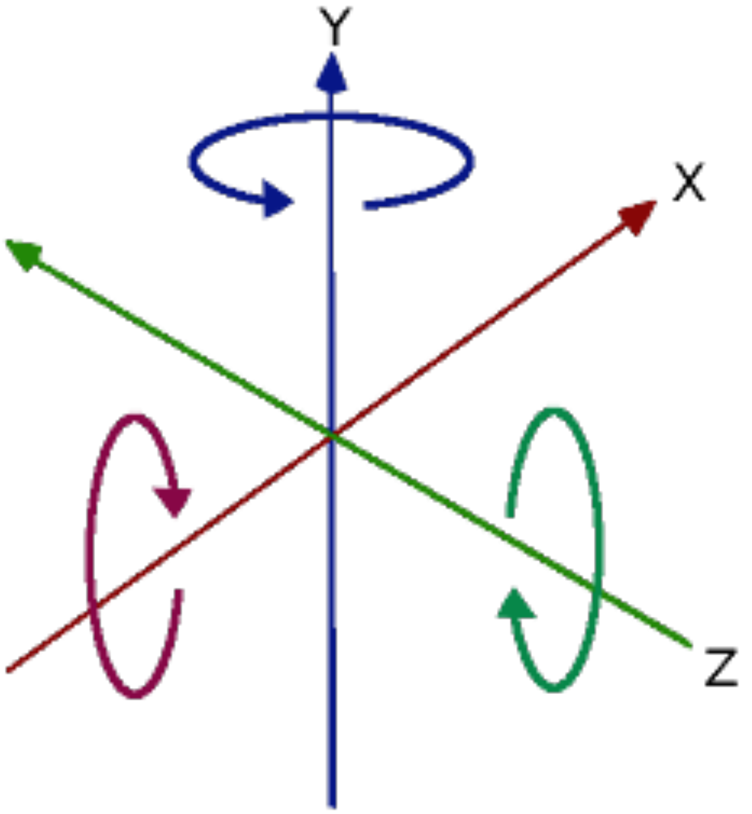
WHAT ABOUT CAMERAS?

- Cameras automatically display on closest parent viewport
  - Only one active camera per viewport
  - Viewport nodes only display objects that are their children
- Must instance the world scene to **both** viewports for displaying splitscreens/ overhead maps/etc



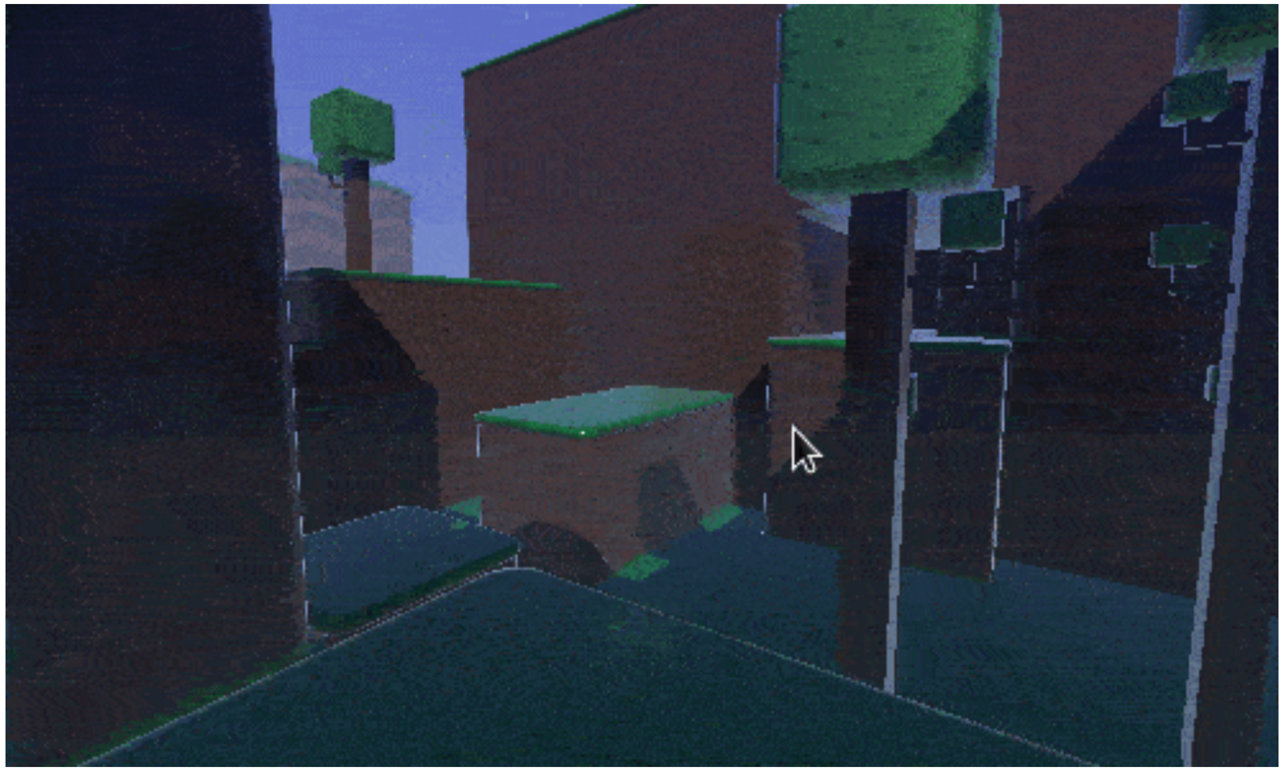
UNDERSTANDING ROTATION

- Euler angles are a common way of representing orientation and rotation
  - Rotations about the x, y, and z axis can be composed to form any arbitrary rotation
  - Yaw (up-axis), pitch (side-axis), and roll (front-axis)
- If any orientation/rotation can be represented, why are Euler angles insufficient?

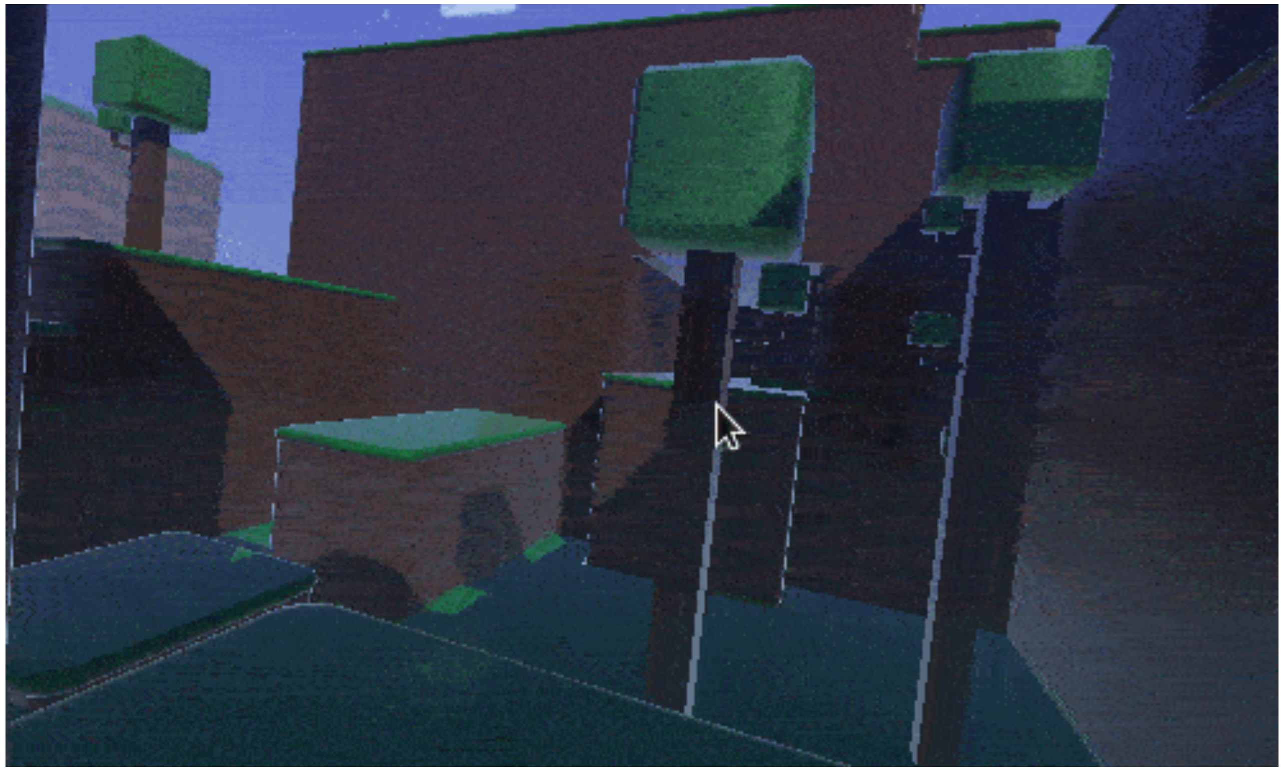


GIMBAL LOCK

- Gimbal Lock Explained:
  - https://www.youtube.com/watch?v=zc8b2Jo7mno



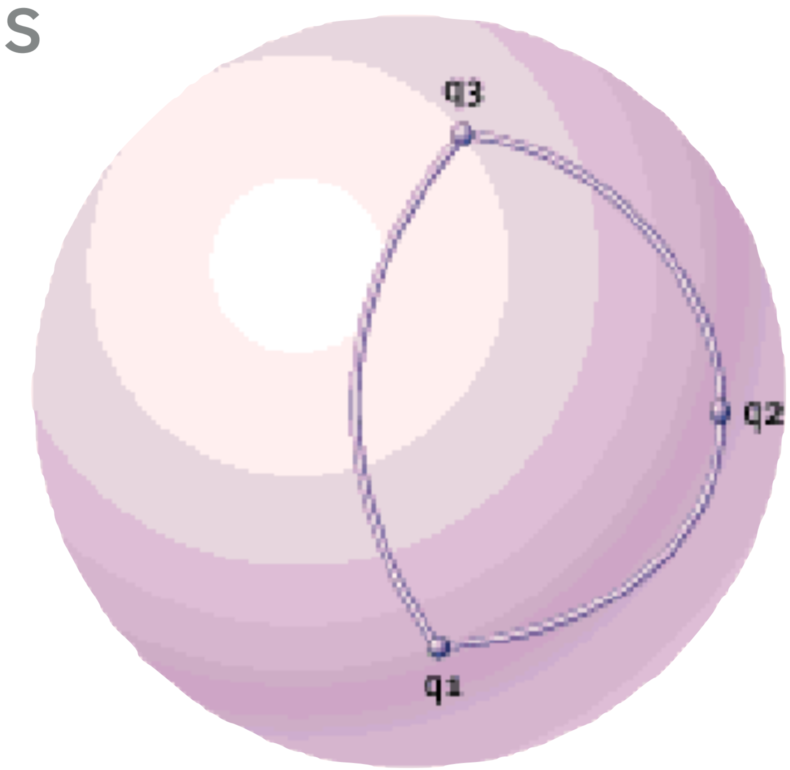
YX rotation



XY rotation

QUATERNIONS

- Mathematical notation for representing object orientation and rotation
- Complex planes rather than Cartesian planes
- Alternative to Euler angles and matrices
- No gimbal lock
- Simpler representation
- Finds closest path



Quaternion Rotation (Gamasutra)

NOTATION

- Complex Number Notation:

$$q = w + xi + yj + zk$$

- 4D Vector Notation:

$$q = [w, v] \text{ where } v = (x, y, z)$$

- Rotate by angle *ϴ* about axis v̂:

$$q = [cos\tfrac{1}{2}\theta, sin\tfrac{1}{2}\theta\hat{v}]$$

- Can apply Euler rotations using axis-angle notation above
  - Must apply rotations in correct order as quaternion multiplication is not commutative!

QUATERNION INTERPOLATION

- SLERP (Spherical Linear Interpolation)
  - Equation for LERP: $p_t = p_1 + (p_2 - p_1)t$
  - Equation for SLERP: $q_t = \frac{sin((1-t)\theta)}{sin(\theta)} q_1 + \frac{sin(t\theta)}{sin(\theta)} q_2$
- SQAD (Spherical and Quadrangle)
  - Smoothly interpolate over a path of rotations (cubic)
  - Defines “helper” quaternion that acts as a control point
- Caveat: when the angular distance between p_1 and p_2 is small, sin(θ) approaches zero. Must switch back to LERP.

WORKING WITH ROTATIONS IN GAMES

- Often easier to think of rotations as Euler angles…
- But should convert to quaternions whenever applying rotations/ interpolations!
- One way to do this:
  1. Get current and target orientation values as Euler angles
  2. Convert Euler angles to quaternions
  3. Slerp between current and target quaternion
  4. Convert back to Euler angles
- Some overhead but your designers will thank you!

FURTHER READING ON QUATERNIONS

- Understanding Quaternions (Jeremiah van Oosten)
  - http://3dgep.com/understanding-quaternions/
- Rotating Objects Using Quaternions (Nick Bobic)
  - http://www.gamasutra.com/view/feature/131686/rotating_objects_using_quaternions.php