Coordinate Spaces
& Transformations

- The Rasterization Pipeline
- Transformations
- Homogeneous Coordinates
- 3D Rotations

The Goal Of Graphics

- Render very high complexity 3D scenes
  - Hundreds of thousands to millions to billions of triangles in a scene
  - Complex vertex and fragment shader computations
  - High resolution screen outputs (~10Mpixel + supersampling)
  - 30-120 fps

- Limited hardware resources
  - Can’t always afford an RTX 4090
  - Be efficient enough to run on commercial hardware

Unreal Engine 5 Tech Demo (2020) Epic Games

Processing The Graphics Pipeline

- Modern real time image generation based on rasterization
- **INPUT:**
  - 3D “primitives”—essentially all triangles!
  - Colors
  - Textures
- **OUTPUT:**
  - Bitmap image (possibly w/ depth, alpha, …)

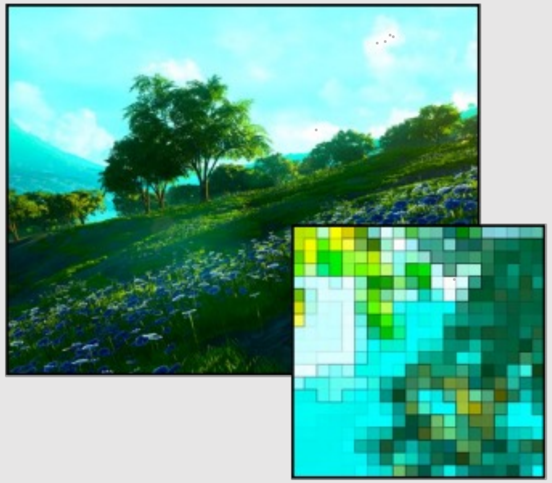
INPUT (TRIANGLES)

```
VERTICES
A: ( 1, 1, 1 )   E: ( 1, 1,-1 )
B: (-1, 1, 1 )   F: (-1, 1,-1 )
C: ( 1,-1, 1 )   G: ( 1,-1,-1 )
D: (-1,-1, 1 )   H: (-1,-1,-1 )

TRIANGLES
EHF, GFH, FGB, CBG,
GHC, DCH, ABD, CDB,
HED, ADE, EFA, BAF
```

RASTERIZATION PIPELINE

OUTPUT (BITMAP IMAGE)



Q: How do we write software to perform rasterization?

Graphics APIs

- Graphics APIs provide a way to interface with GPUs
  - More than just draw calls:
    - State management
    - Memory management
    - Bindings
    - Window/GUI/Events

- Think of a graphics API as a way for the CPU to communicate with the GPU
  - Doesn't necessarily need to be for graphics
    - **Ex:** compute shaders

- Common APIs:
  - OpenGL (Khronos Group)
  - Vulkan (Khronos Group)
  - Metal (Apple)
  - DirectX (Windows)

Hardware Vs Software Rasterization

Hardware

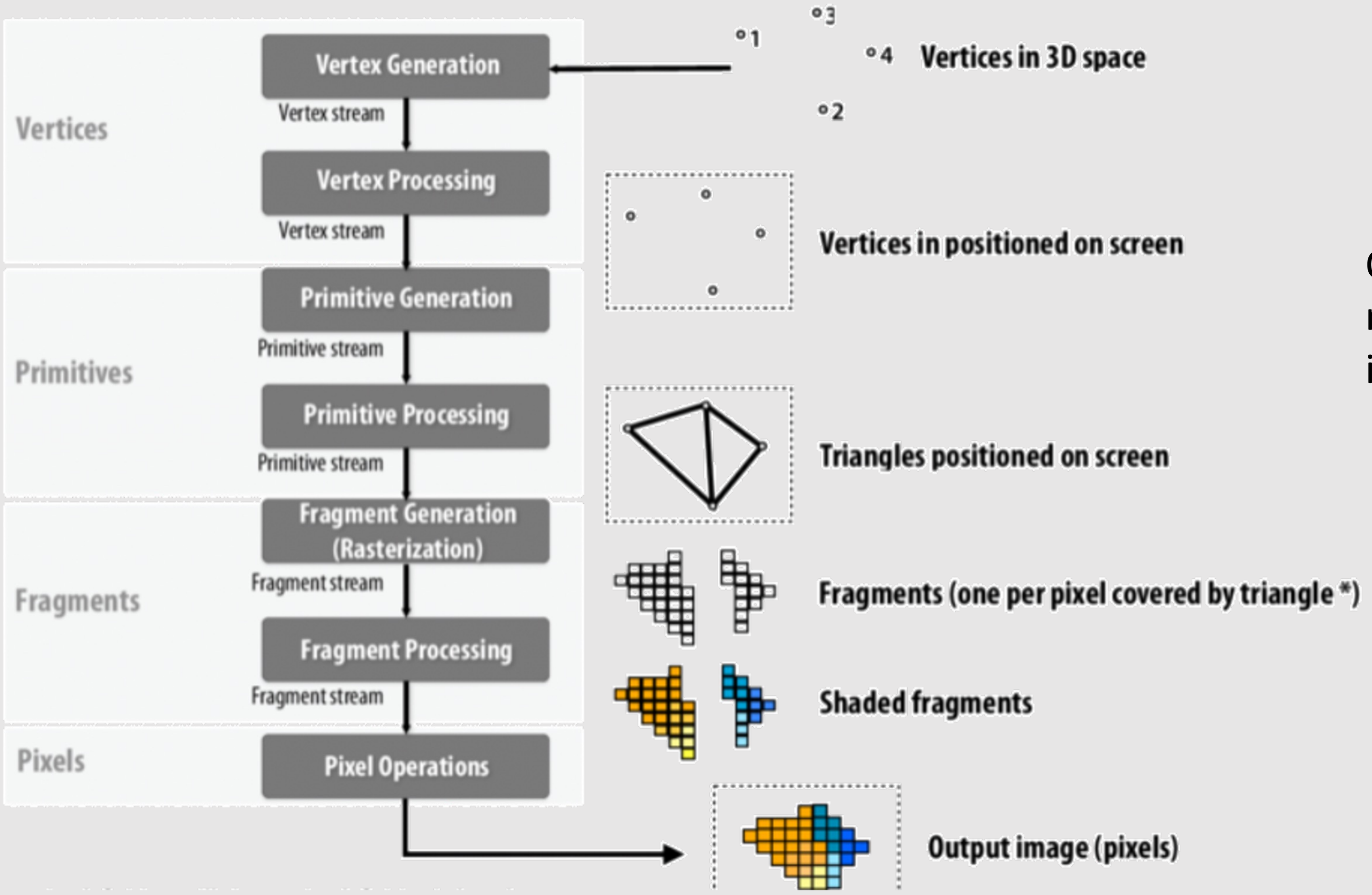
- Written to run on the GPU
- Written using one or more Graphics APIs
- No clear method to debug shaders**
- Much faster execution
- Inherently data-parallel
- Harder to write
- Branching shaders can hurt execution

Software

- Written to run on the CPU
- Modify the framebuffer pixel by pixel
- Very easy to debug
- Very slow execution
- Not parallel
- Easier to write
- Branching doesn’t hurt serial execution

** APIs such as Metal offer debug tools to help profile stages of the rasterization pipeline

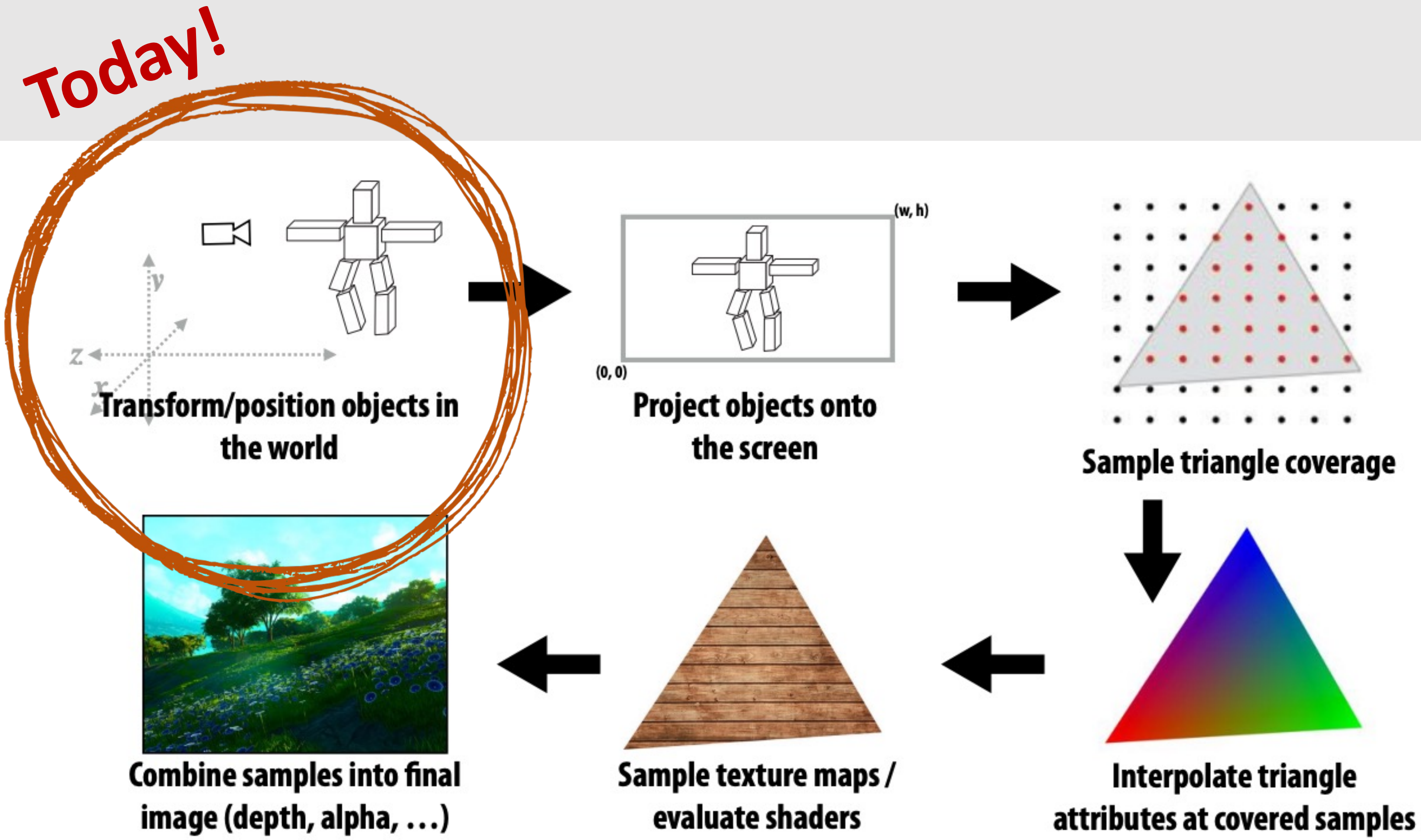
The Graphics Pipeline



Our rasterization pipeline doesn't look much different from "real" pipelines used in modern APIs / graphics hardware

Let’s simplify things a bit

The “Simpler” Graphics Pipeline



- ~~The Rasterization Pipeline~~
- Transformations
- Homogeneous Coordinates
- 3D Rotations

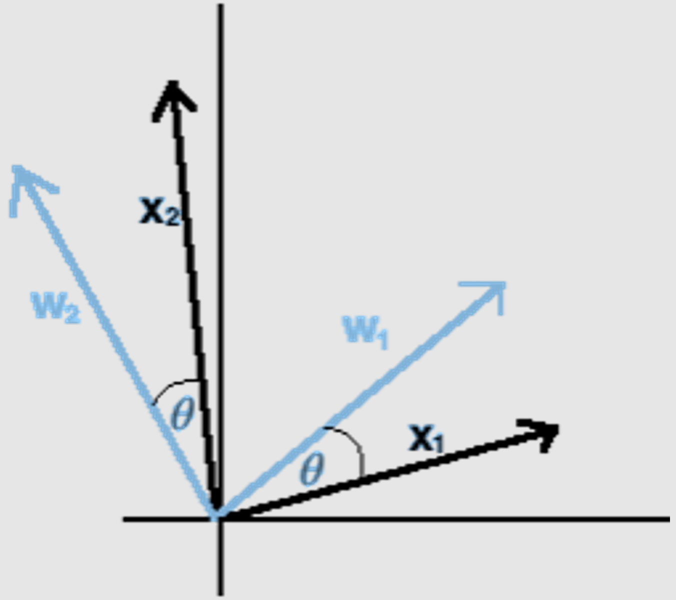
Transformations In Computer Graphics

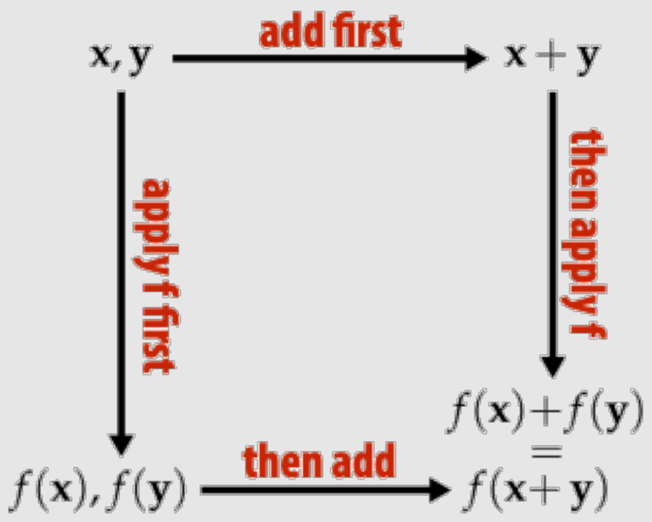
- Common uses of linear transformations:
  - Position/deform objects in space
  - Camera movements
  - Animate objects over time
  - Project 3D objects onto 2D images
  - Map 2D textures onto 3D objects
  - Project shadows of objects onto other objects
- Today we'll focus on common transformations of space (rotation, scaling, etc.) encoded by linear maps

Super Mario 64: Camera Guy (1996) Nintendo

Review: Linear Maps

What does it mean for a map $f: \mathbb{R}^n \to \mathbb{R}^n$ to be linear?





Geometrically it maps lines to lines, and preserves the origin

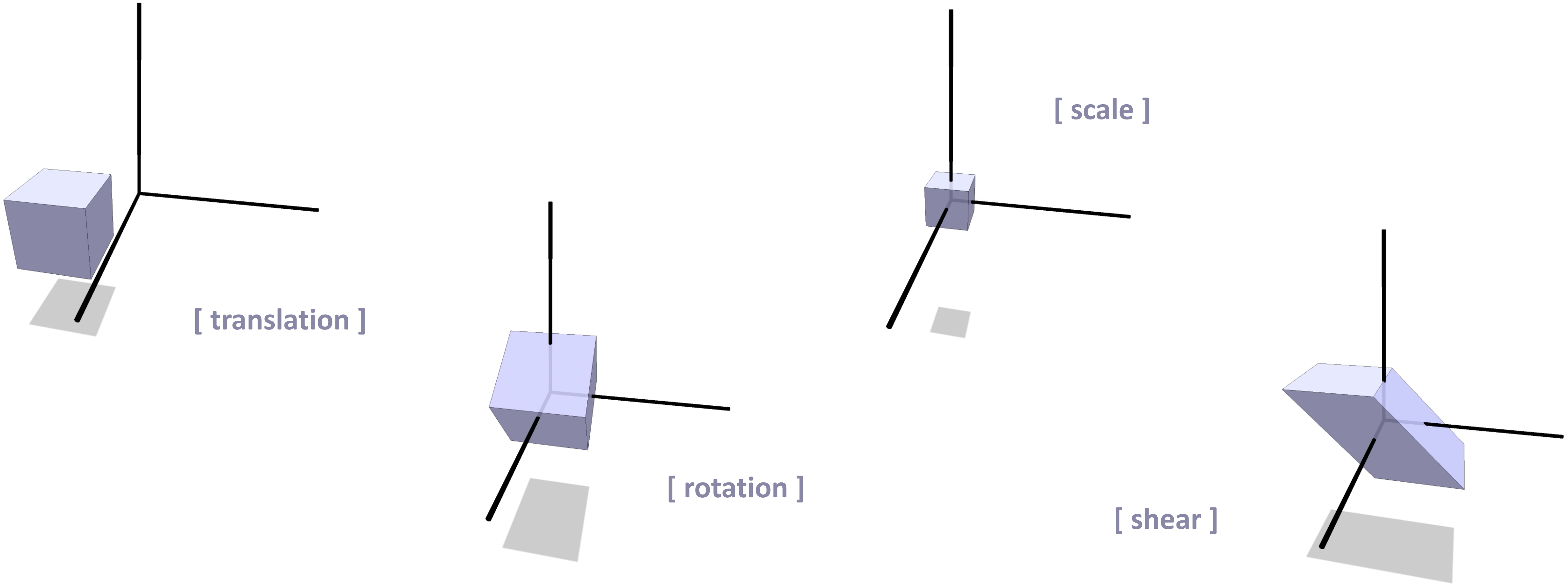
Algebraically it preserves vector space operations (addition & scaling)

Review: Linear Maps

- Why do we care about linear transformations?
  - Cheap to apply
  - Usually pretty easy to solve for (linear systems)
  - **Composition of linear transformations is linear**
    - Product of many matrices is a single matrix
    - Gives uniform representation of transformations
    - Simplifies graphics algorithms, systems (e.g., GPUs & APIs)

$$\underbrace{\begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}}_{\text{[rotation]}} \underbrace{\begin{bmatrix} a & 0 & 0 \\ 0 & b & 0 \\ 0 & 0 & c \end{bmatrix}}_{\text{[scale]}} \underbrace{\begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix}}_{\text{[rotation]}} \cdots = \underbrace{\begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix}}_{\text{[composite]}}$$

Types of Transformations

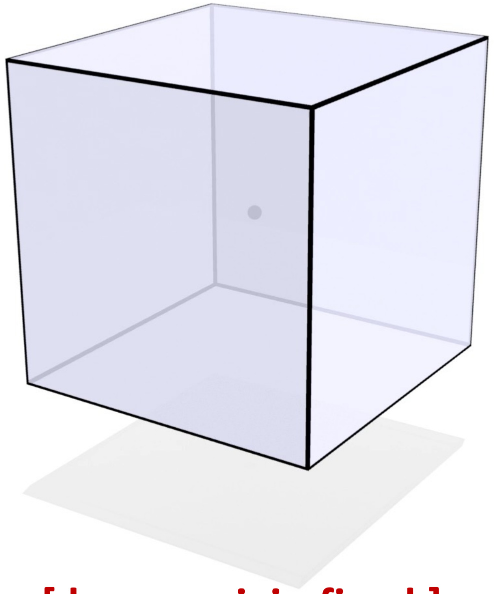


Invariants of Transformation

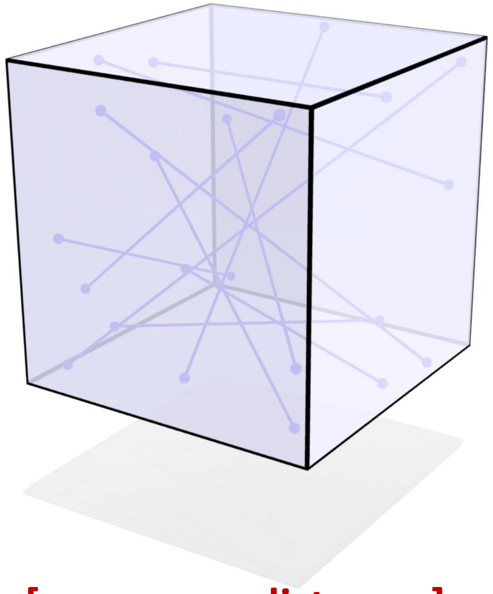
A transformation is determined by the **invariants** it preserves

transformation	invariants	algebraic description
linear	*straight lines / origin*	$f(a\mathbf{x}+\mathbf{y}) = af(\mathbf{x}) + f(\mathbf{y})$, $f(0) = 0$
translation	*differences between pairs of points*	$f(\mathbf{x}-\mathbf{y}) = \mathbf{x}-\mathbf{y}$
scaling	*lines through the origin / direction of vectors*	$f(\mathbf{x})/\lvert f(\mathbf{x})\rvert = \mathbf{x}/\lvert\mathbf{x}\rvert$
rotation	*origin / distances between points / orientation*	$\lvert f(\mathbf{x})-f(\mathbf{y})\rvert = \lvert\mathbf{x}-\mathbf{y}\rvert$, $\det(f) > 0$
…	…	…

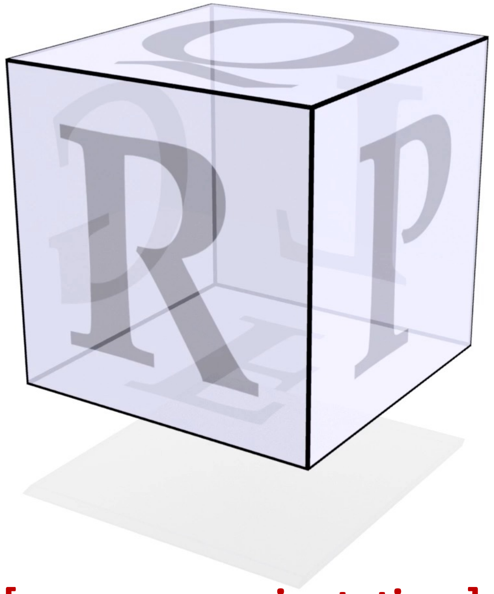
Rotation



[keeps origin fixed]



[preserves distance]



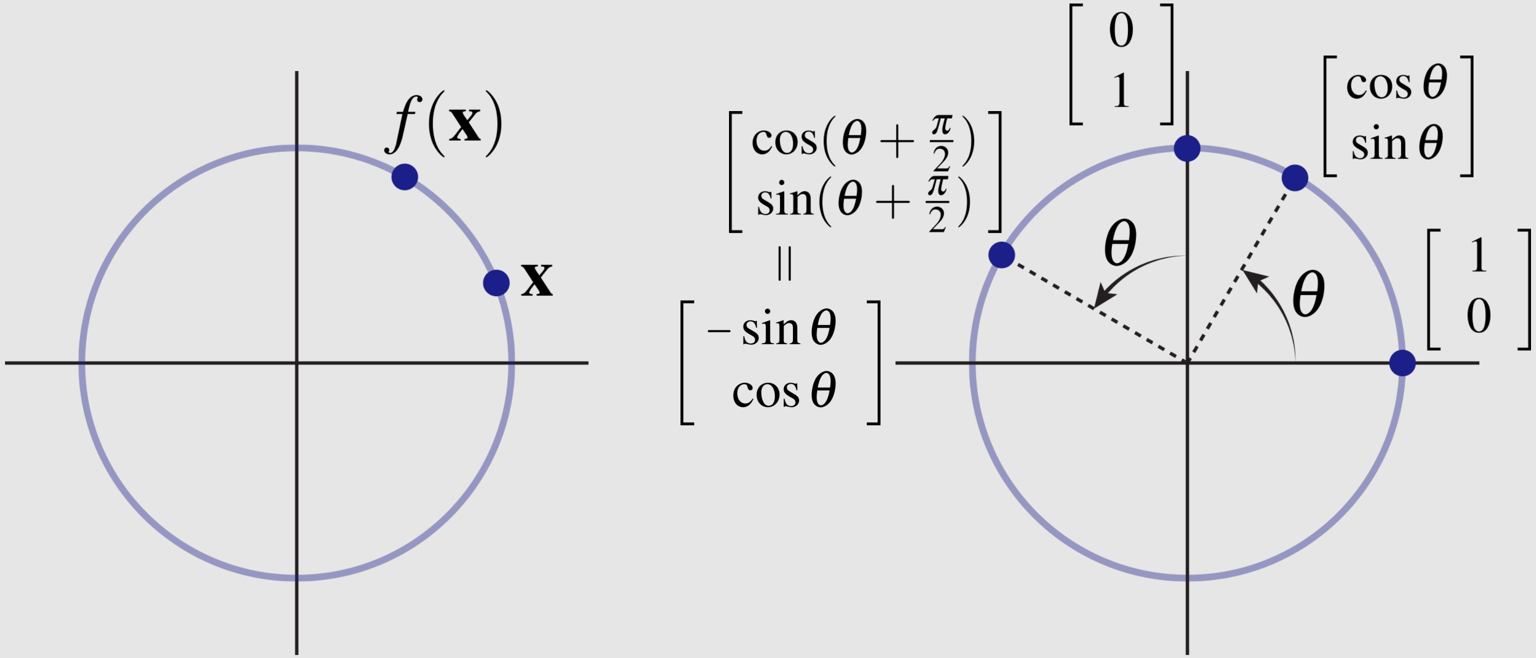
[preserves orientation]

First two properties imply rotations are **linear**

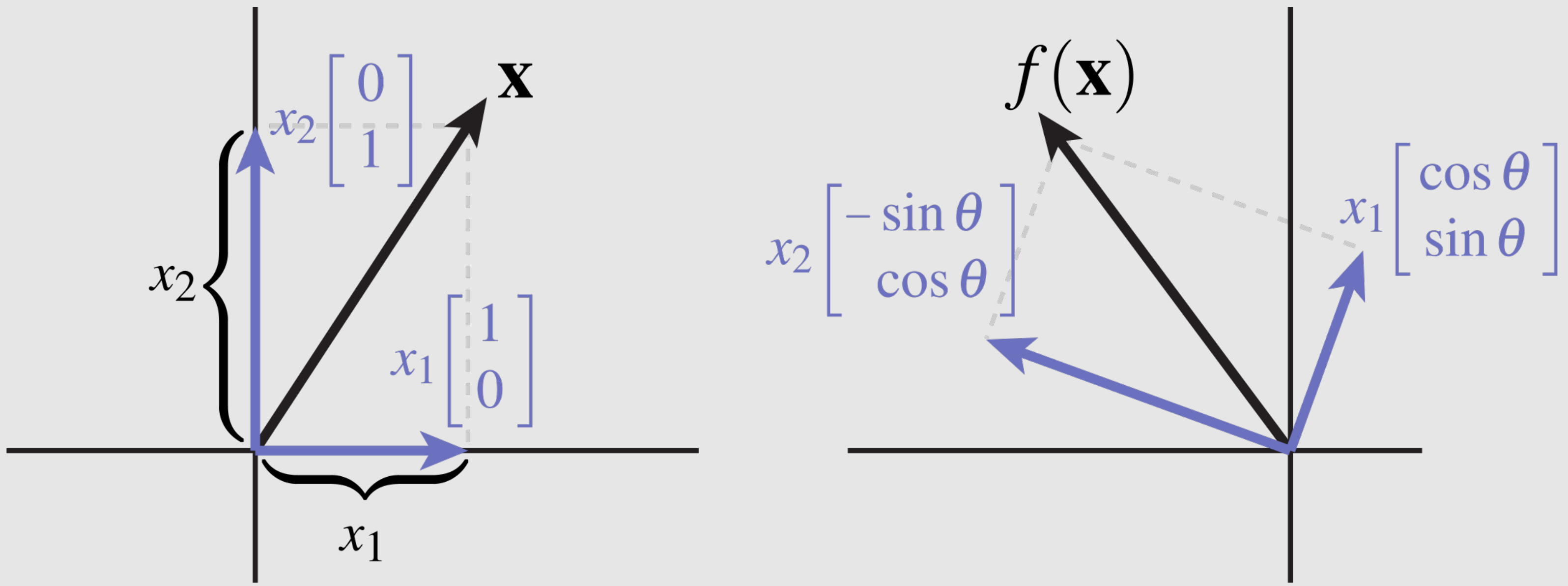
We say that a transform preserves orientation if $\det(T) > 0$

2D Rotations

Rotations preserve distances and the origin—hence, a 2D rotation by an angle θ maps each point x to a point $f(x)$ on the circle of radius $|x|$:



2D Rotations



$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = x_1 \begin{bmatrix} 1 \\ 0 \end{bmatrix} + x_2 \begin{bmatrix} 0 \\ 1 \end{bmatrix} \qquad f(\mathbf{x}) = x_1 \begin{bmatrix} \cos\theta \\ \sin\theta \end{bmatrix} + x_2 \begin{bmatrix} -\sin\theta \\ \cos\theta \end{bmatrix}$$

Rotations (like all transforms) are linear maps.
We can express the transform as a change of bases:

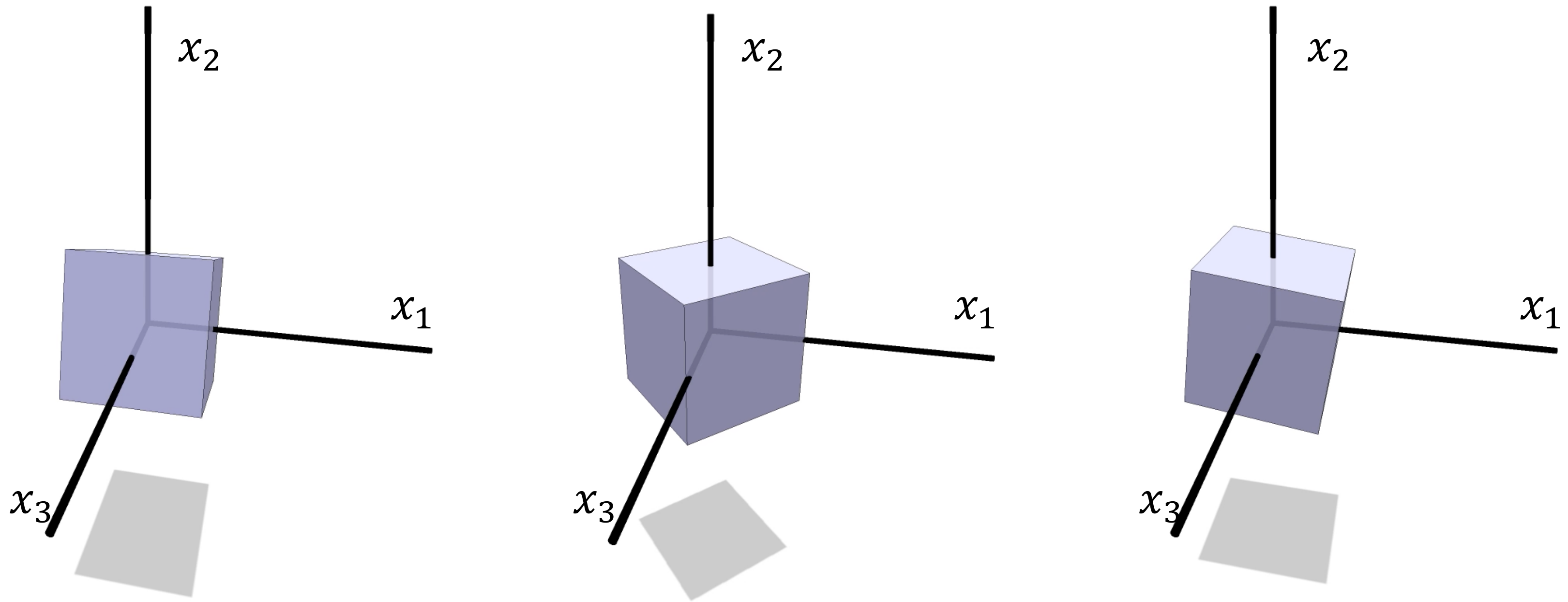
$$f_\theta(\mathbf{x}) = \begin{bmatrix} \cos\theta & -\sin(\theta) \\ \sin\theta & \cos(\theta) \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

3D Rotations

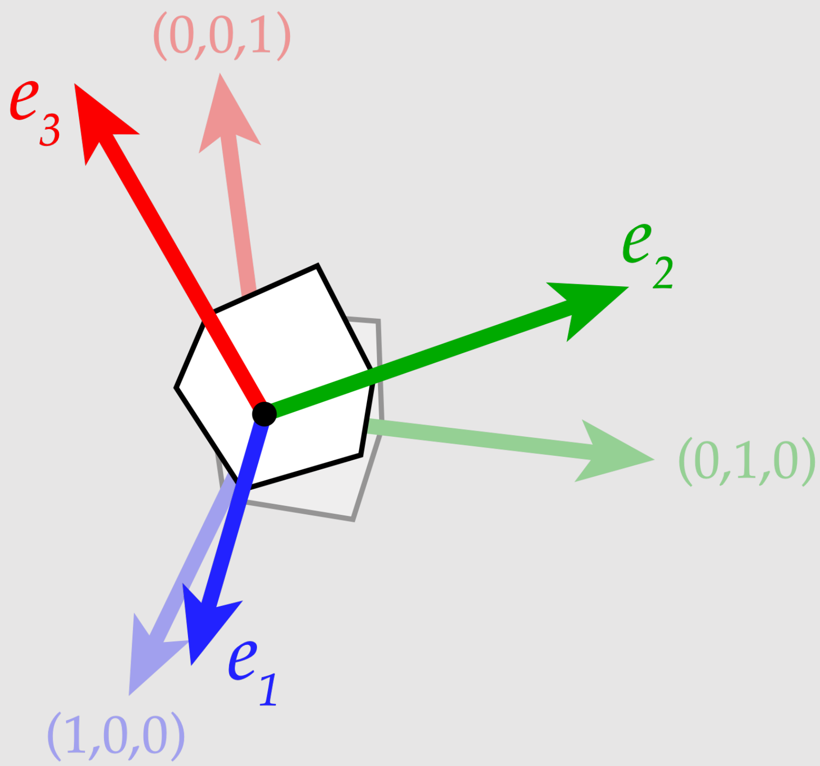
In 3D, keep one axis fixed and rotate the other two:

[rotate around x_1] **[rotate around x_2]** **[rotate around x_3]**

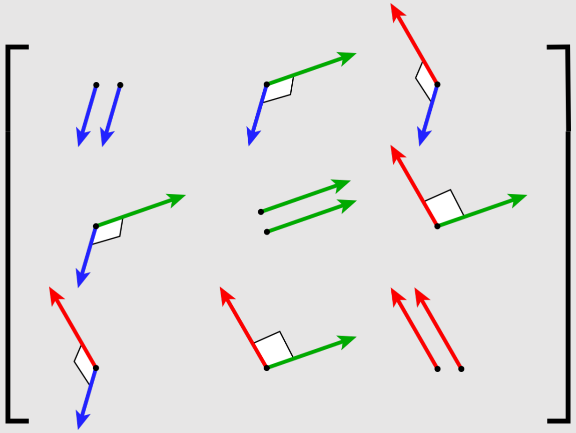
$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin(\theta) \\ 0 & \sin\theta & \cos(\theta) \end{bmatrix} \begin{bmatrix} \cos\theta & 0 & \sin(\theta) \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos(\theta) \end{bmatrix} \begin{bmatrix} \cos\theta & -\sin(\theta) & 0 \\ \sin\theta & \cos(\theta) & 0 \\ 0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



3D Inverse Rotations



$$\begin{array}{cc} R^{\mathsf{T}} & R \\ \begin{bmatrix} \text{—} & e_1^T & \text{—} \\ \text{—} & e_2^T & \text{—} \\ \text{—} & e_3^T & \text{—} \end{bmatrix} & \begin{bmatrix} | & | & | \\ e_1 & e_2 & e_3 \\ | & | & | \end{bmatrix} \end{array}$$

$=$ 

$$= \begin{bmatrix} e_1^T e_1 & e_1^T e_2 & e_1^T e_3 \\ e_2^T e_1 & e_2^T e_2 & e_2^T e_3 \\ e_3^T e_1 & e_3^T e_2 & e_3^T e_3 \end{bmatrix}$$

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

$$\boldsymbol{R^T R = I \;\; \Rightarrow \;\; R^T = R^{-1}}$$

Reflections

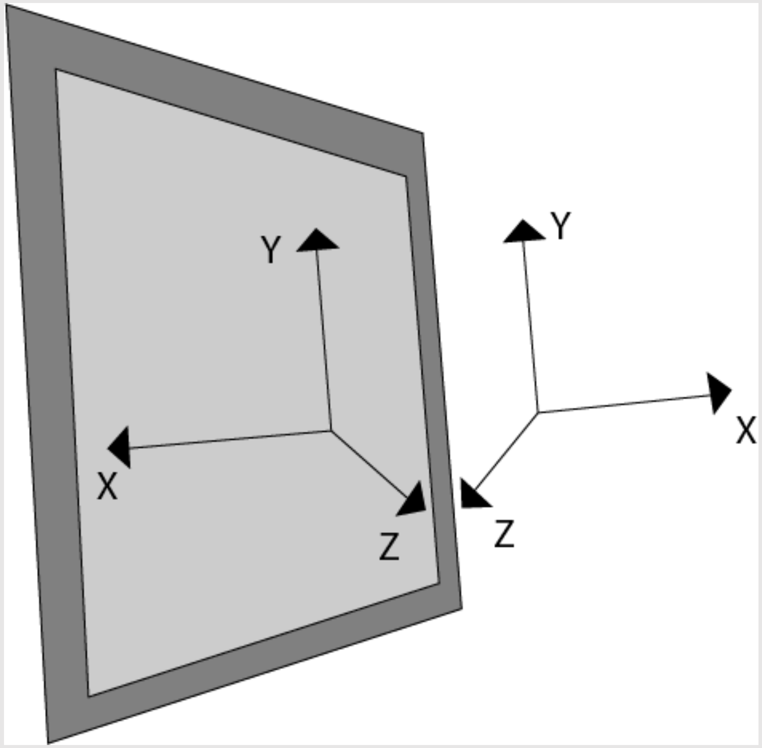
- Does every matrix $Q^{\mathsf{T}}Q = I$ represent a rotation?
  - Must preserve:
    - Origin
    - Distance
    - Orientation
- Consider:

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

- Just like rotations, Q has nice inverse properties:

$$Q^{\mathsf{T}}Q = \begin{bmatrix} (-1)^2 & 0 \\ 0 & 1 \end{bmatrix} = I$$

- But the determinant is **negative!**
  - Not orientation preserving



Scaling

- Each vector u gets scaled by some scalar a

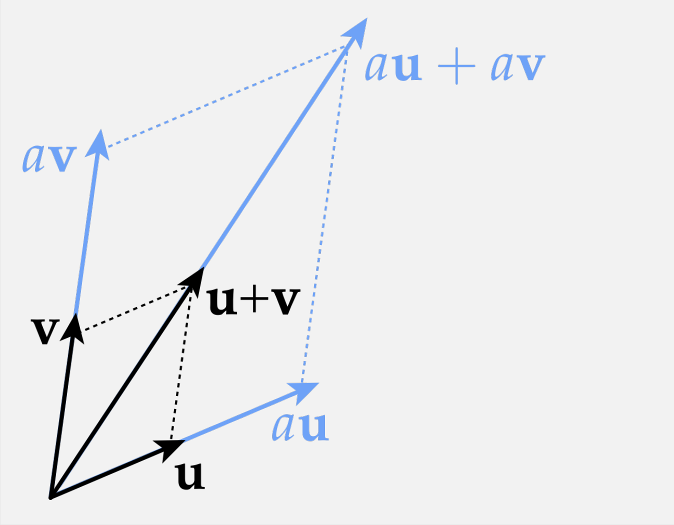
$$f(\mathbf{u}) = a\mathbf{u}, a \in \mathbb{R}$$

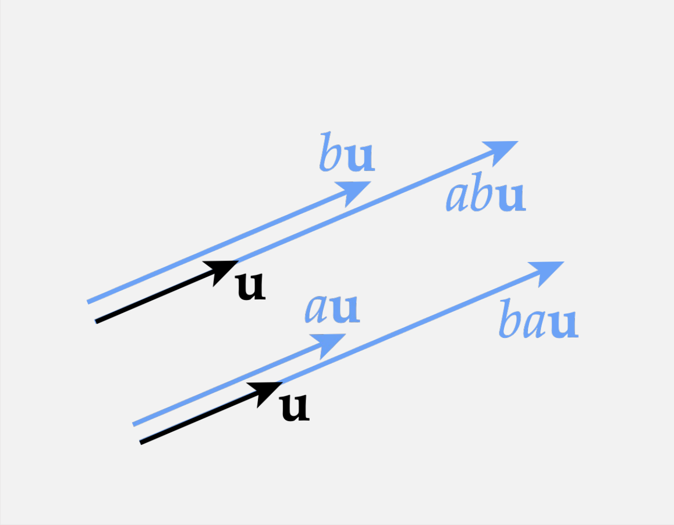
- Scaling is a linear transformation
  - Addition:

$$f(b\mathbf{u}) = ab\mathbf{u} = ba\mathbf{u} = bf(\mathbf{u})$$

  - Multiplication:

$$\begin{gathered} f(\mathbf{u} + \mathbf{v}) = \\ a(\mathbf{u} + \mathbf{v}) = \\ a\mathbf{u} + a\mathbf{v} = \\ f(\mathbf{u}) + f(\mathbf{v}) \end{gathered}$$





Negative Scaling

Can think of negative scaling as a series of reflections

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

Also works in 3D:

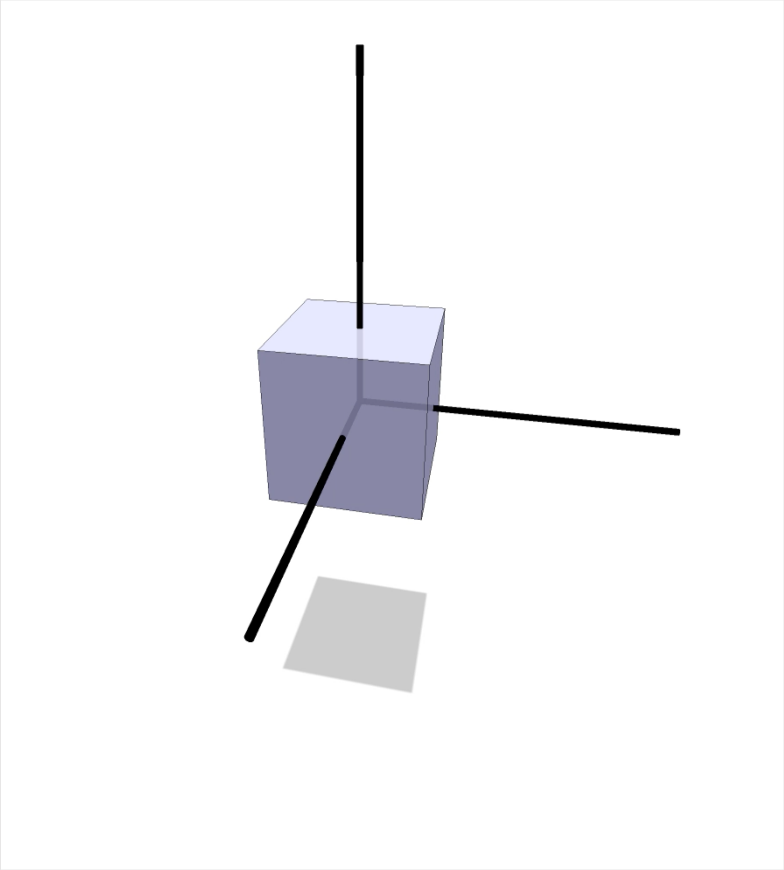
$$\begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{bmatrix} = \underbrace{\begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}}_{\text{[flip x]}} \underbrace{\begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix}}_{\text{[flip y]}} \underbrace{\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{bmatrix}}_{\text{[flip z]}}$$

In 2D, reflection reverses orientation twice ($\det(T) > 0$)
In 3D, reflection reverses orientation thrice ($\det(T) < 0$)

Non-Uniform Scaling

- To scale a vector u by a non-uniform amount (a, b, c):

$$\begin{bmatrix} a & 0 & 0 \\ 0 & b & 0 \\ 0 & 0 & c \end{bmatrix}\begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix} = \begin{bmatrix} au_1 \\ bu_2 \\ cu_3 \end{bmatrix}$$



- The above works only if scaling is axis-aligned. What if it isn't?
- Idea:
  - Rotate to a new axis R
  - Perform axis-aligned scaling D
  - Rotate back to original axis R^T

$$A \coloneqq R^T D R$$

  - Resulting transform A is a symmetric matrix

- **Q:** Do all symmetric matrices represent non-uniform scaling?

Spectral Theorem

- **Spectral theorem** says a symmetric matrix $A = A^T$ has:
  - Orthonormal eigenvectors $e_1, \dots, e_n \in \mathbb{R}^n$
  - Real eigenvalues $\lambda_1, \dots, \lambda_n \in \mathbb{R}$

- Eigenvalues represent the diagonals of the scalar transform
- Eigenvectors are axis which we are scaling about
  - Can be represented as a rotation transform

$$R = \begin{bmatrix} e_1 & \cdots & e_n \end{bmatrix} \quad D = \begin{bmatrix} \lambda_1 & & \\ & \ddots & \\ & & \lambda_n \end{bmatrix}$$

- Can write the relationship as $AR = RD$
  - Equivalently, $A = RDR^{\mathsf{T}}$

- Hence, every symmetric matrix performs a non-uniform scaling along some set of orthogonal axes

Shear

- A shear displaces each point x in a direction u according to its distance along a fixed vector v:

$$f_{\mathbf{u},\mathbf{v}}(\mathbf{x}) = \mathbf{x} + \langle \mathbf{v}, \mathbf{x} \rangle \mathbf{u}$$

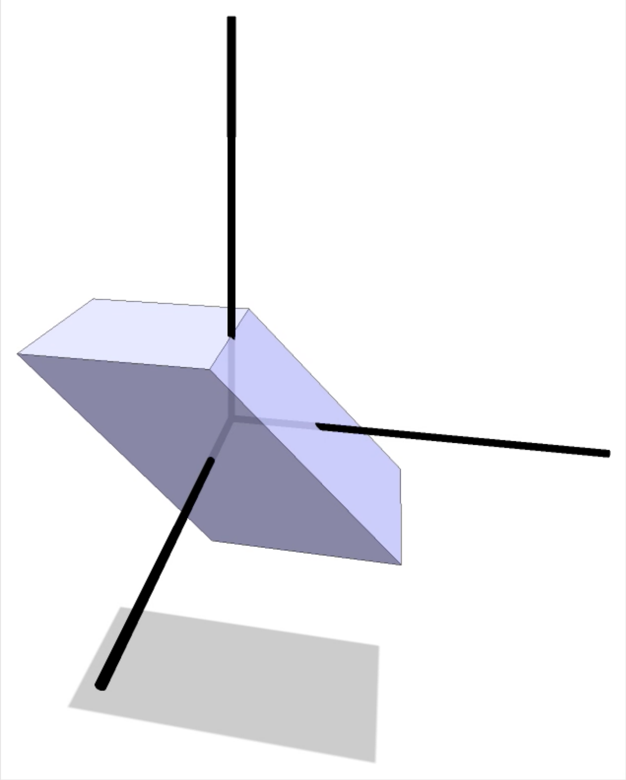
- Still a linear transformation—can be rewritten as:

$$A_{\mathbf{u},\mathbf{v}} = I + \mathbf{u}\mathbf{v}^\mathsf{T}$$

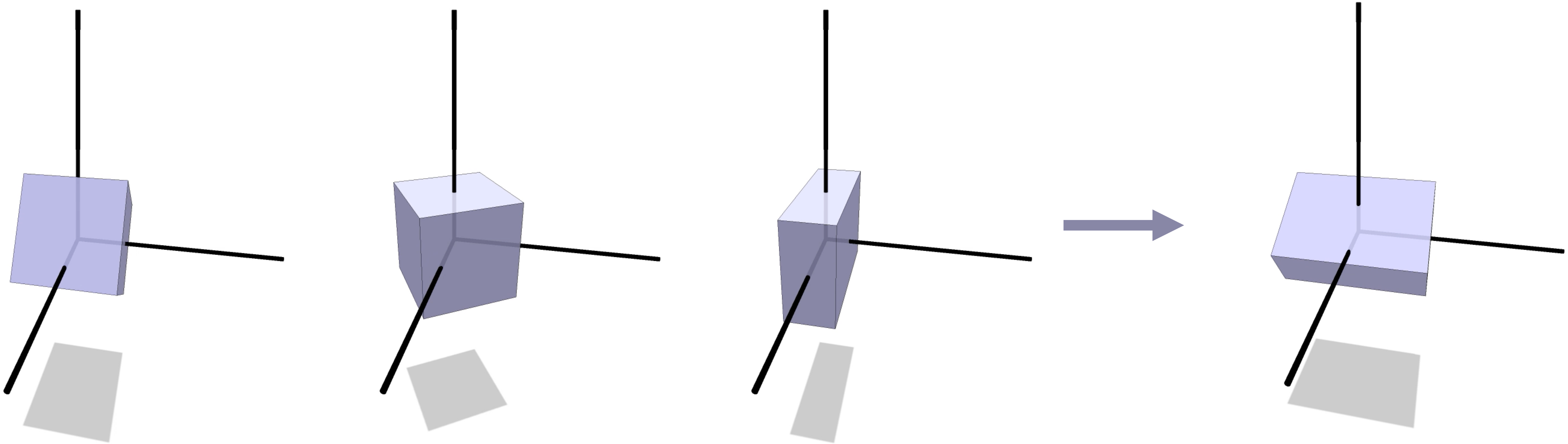
- Example:

$$\mathbf{u} = (\cos(t), 0,0)$$
$$\mathbf{v} = (0,1,0)$$

$$A_{\mathbf{u},\mathbf{v}} = \begin{bmatrix} 1 & \cos(t) & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$



Composing Transforms

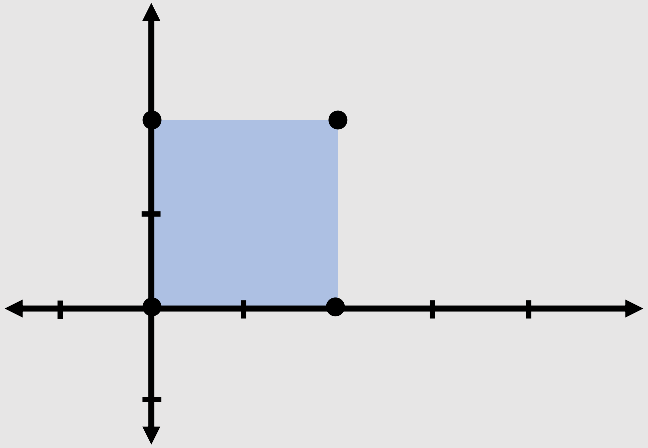


$R_x(t)$ $\quad$ $R_y(t)$ $\quad$ $S(t)$ $\quad$ $A(t) = R_x(t)R_y(t)S(t)$

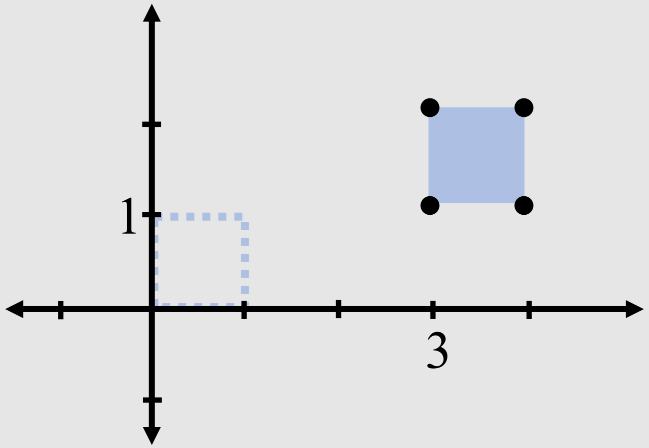
We can now build up composite transformations via matrix multiplication

Composing Transforms

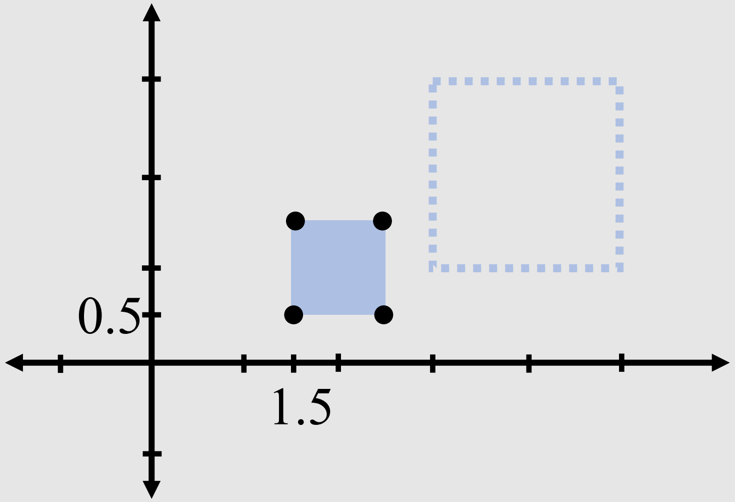
- Order matters when compositing transforms!



[original]



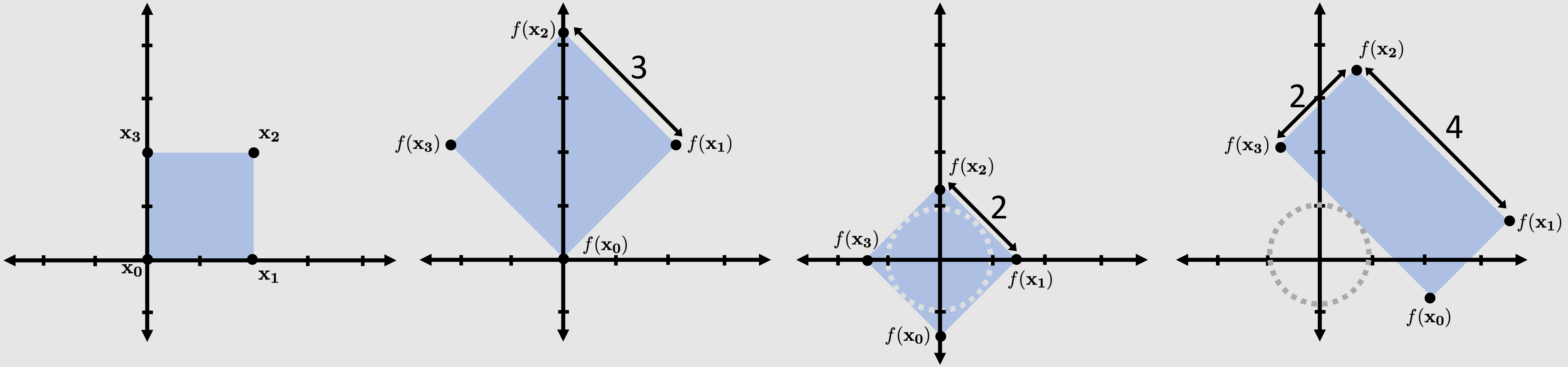
[scale by 1/2, then translate by (3,1)]



[translate by (3,1), then scale by 1/2]

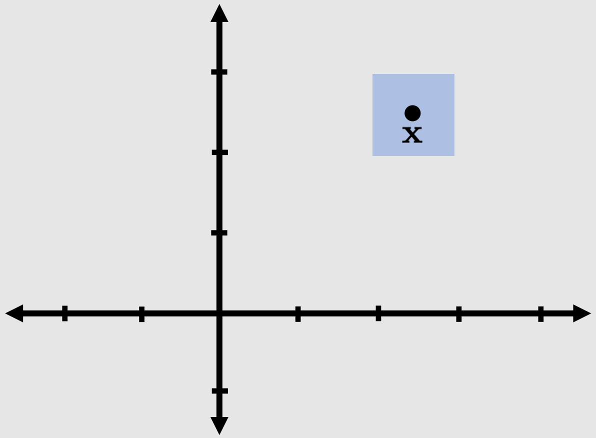
Composing Transforms

How would you perform these transformations?**

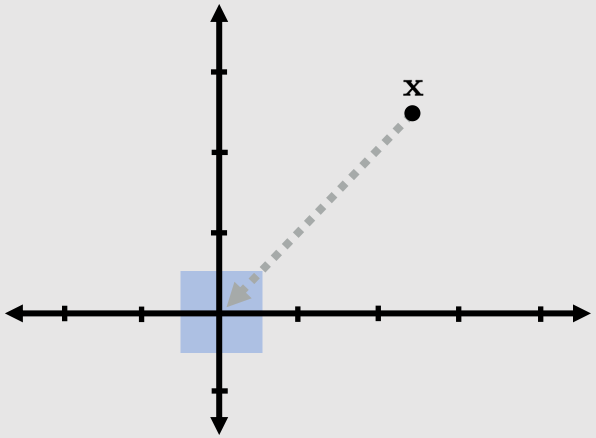


**remember there's always more than one way to do so

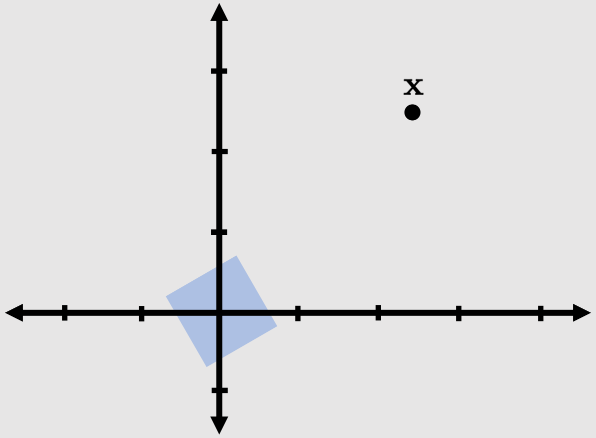
Rotating About A Point



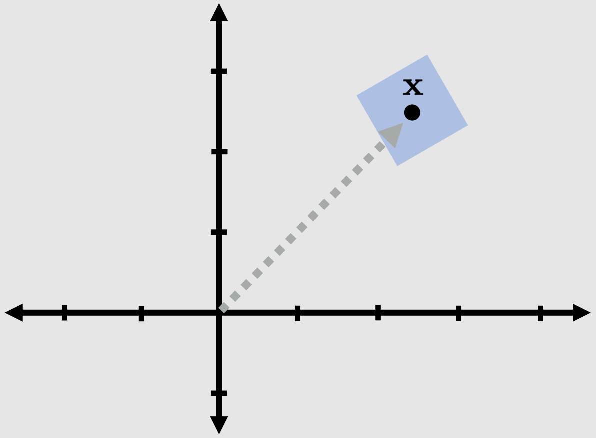
[Step 0] compute x (dist. from origin)



[Step 1] translate by -x



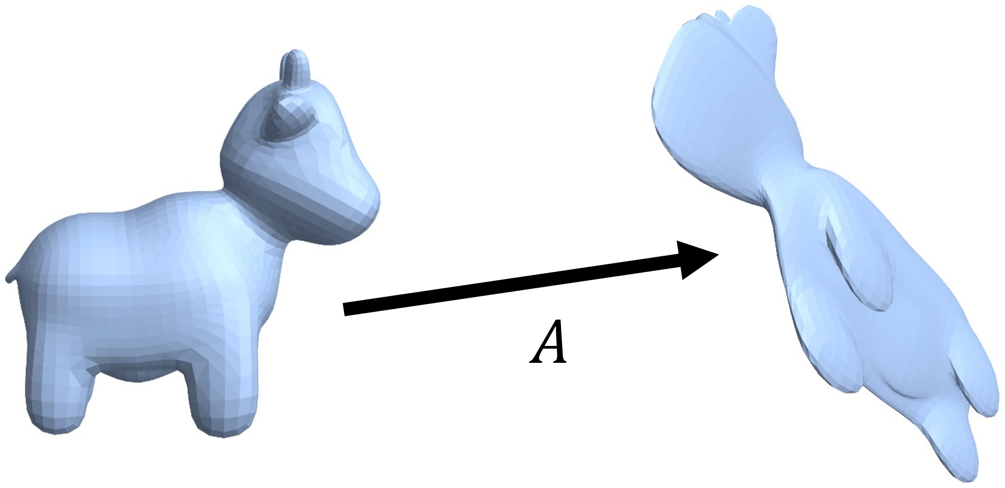
[Step 2] rotate



[Step 3] translate by x

Decomposing Transforms

- In general, no **unique** way to write a given linear transformation as a composition of basic transformations!
  - However, there are many useful decompositions:
    - **Singular value decomposition**
      - Good for signal processing
    - **LU factorization**
      - Good for solving linear systems
    - **Polar decomposition**
      - Good for spatial transformations



$$A = \begin{bmatrix} .34 & -.11 & -.89 \\ -.65 & .52 & -.70 \\ .25 & .23 & -.69 \end{bmatrix}$$

Polar & Single Value Decomposition

Polar decomposition decomposes any matrix A into orthogonal matrix Q and symmetric positive-semidefinite matrix P

rotation/reflection

nonnegative nonuniform scaling

$$A = QP$$

Since P is symmetric, can take this further via the spectral decomposition $P = VDV^T$ (V orthogonal, D diagonal):

$$A = \underbrace{QV}_{U} DV^{\mathsf{T}} = UDV^{\mathsf{T}}$$

rotation

axis-aligned scaling

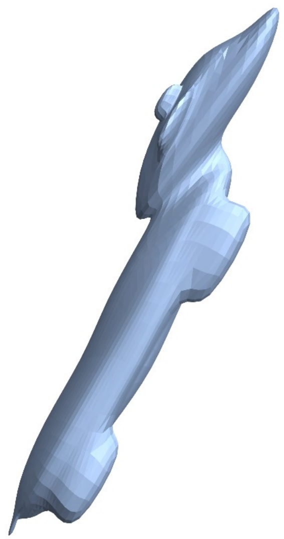
rotation

Result UDV^T is called the **singular value decomposition**

Interpolating Transformations [Linear]

Consider interpolating between two linear transformations
A_0, A_1 of some initial model

Idea: take a linear combination of the two matrices



$$A(t) = (1-t)A_0 + tA_1$$

$$t \in [0,1]$$

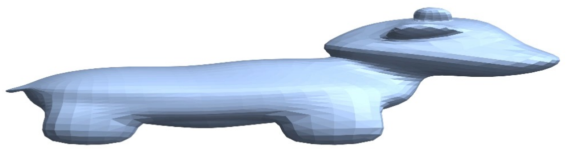
Hits the right start/endpoints… but looks awful in between!

Interpolating Transformations [Polar]

Better idea: separately interpolate components of polar decomposition

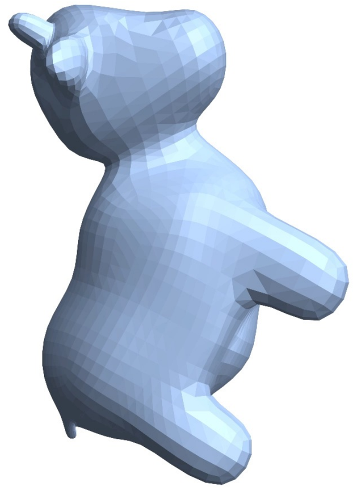
$$A_0 = Q_0 P_0$$
$$A_1 = Q_1 P_1$$

[scaling]



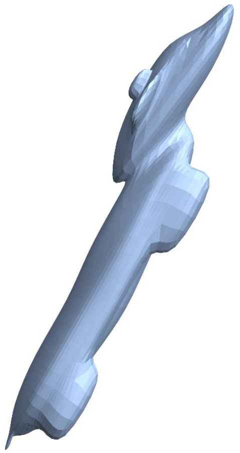
$$P(t) = (1-t)P_0 + tP_1$$

[rotation]



$$Q(t) = (1-t)Q_0 + tQ_1$$

[composite]



$$A(t) = Q(t)P(t)$$

Translation

- So far we've ignored a basic transformation—translations
  - A translation simply adds an offset $\mathbf{u}$ to the given point $\mathbf{x}$

$$f_{\mathbf{u}}(\mathbf{x}) = \mathbf{x} + \mathbf{u}$$

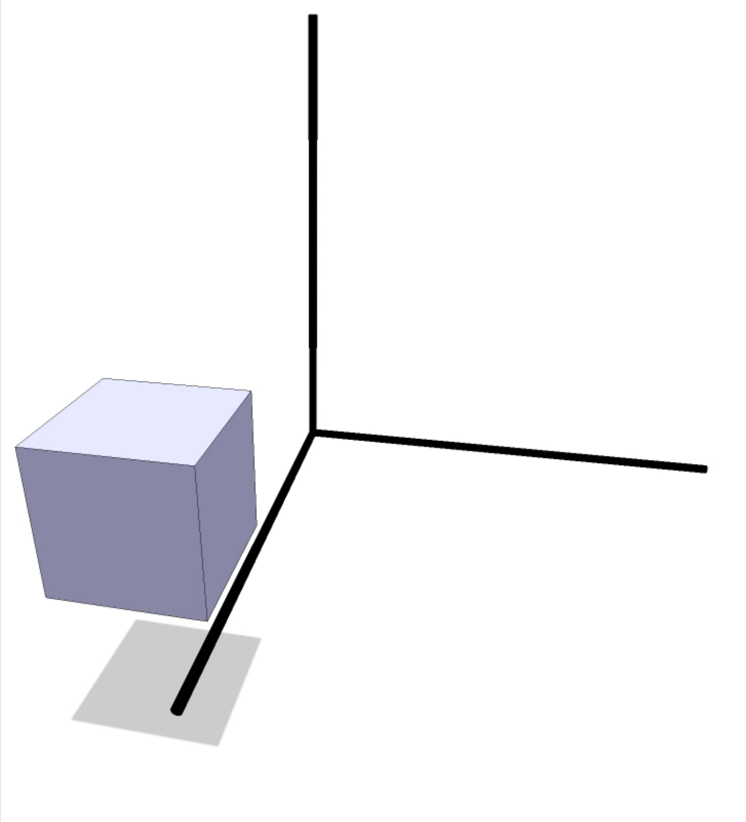
- Is this translation linear?
  - (certainly seems to move across a line…)

[additivity]

$$f_{\mathbf{u}}(\mathbf{x}+\mathbf{y}) = \mathbf{x}+\mathbf{y}+\mathbf{u}$$
$$f_{\mathbf{u}}(\mathbf{x}) + f_{\mathbf{u}}(\mathbf{y}) = \mathbf{x}+\mathbf{y}+2\mathbf{u}$$

[homogeneity]

$$f_{\mathbf{u}}(a\mathbf{x}) = a\mathbf{x}+\mathbf{u}$$
$$af_{\mathbf{u}}(\mathbf{x}) = a\mathbf{x}+a\mathbf{u}$$



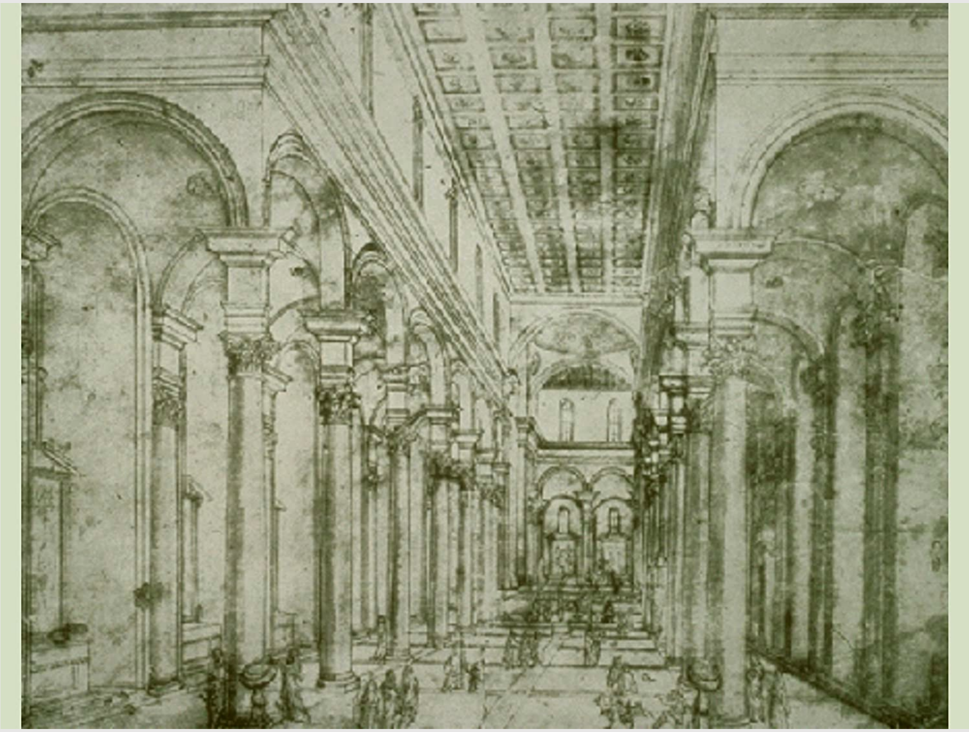
Translations are not linear!

Maybe translations turn linear when we go into the 4^{th} dimension…

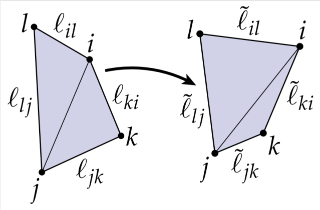
- ~~The Rasterization Pipeline~~
- ~~Transformations~~
- Homogeneous Coordinates
- 3D Rotations

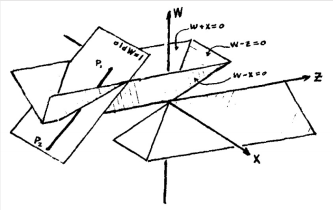
Homogeneous Coordinates

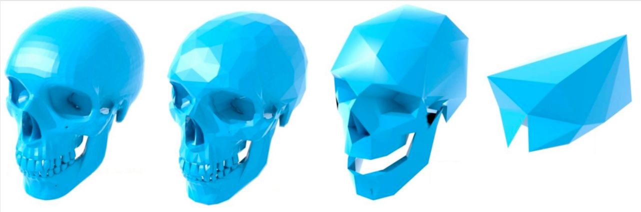
- Came from efforts to study perspective
- Introduced by Möbius as a natural way of assigning coordinates to lines
- Show up naturally in a surprising large number of places in computer graphics:
  - 3D transformations
  - Perspective projection
  - Quadric error simplification
  - Premultiplied alpha
  - Shadow mapping
  - Projective texture mapping
  - Discrete conformal geometry
  - Hyperbolic geometry
  - Clipping
  - Directional lights
  - …

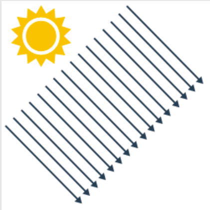


Church of Santo Spirito (1428) Filippo Brunelleschi

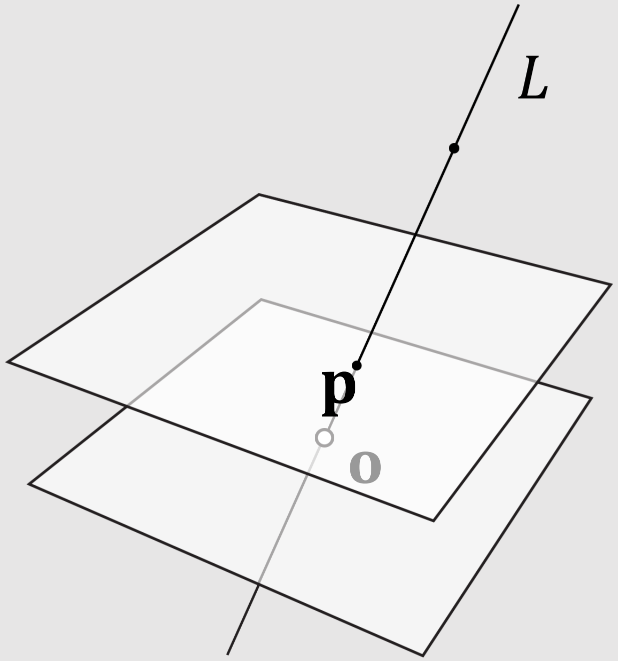








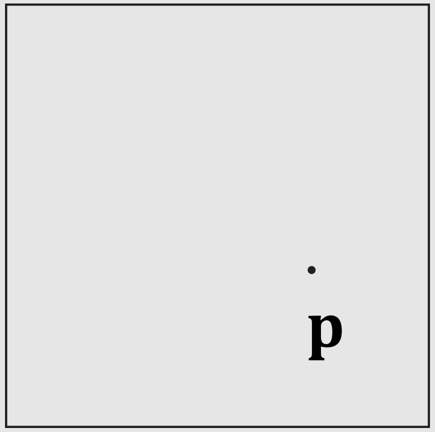
Homogeneous Coordinates in 2D



- Consider any 2D plane that does not pass through the origin o in 3D
  - Every line through the origin in 3D corresponds to a point in the 2D plane
  - Just find the point p where the line L pierces the plane

- Consider a point $p' = (x, y)$, and the plane $z = 1$ in 3D
  - Any three numbers $p = (a, b, c)$ such that $\left(\frac{a}{c}, \frac{b}{c}\right) = (x, y)$ are homogeneous coordinates for p
    - Example: $(x, y, 1)$
    - In general: (cx, cy, c) for $c \neq 0$
      - The c is commonly referred to as the homogeneous coordinate

- Great, but how does this help us with transforms?



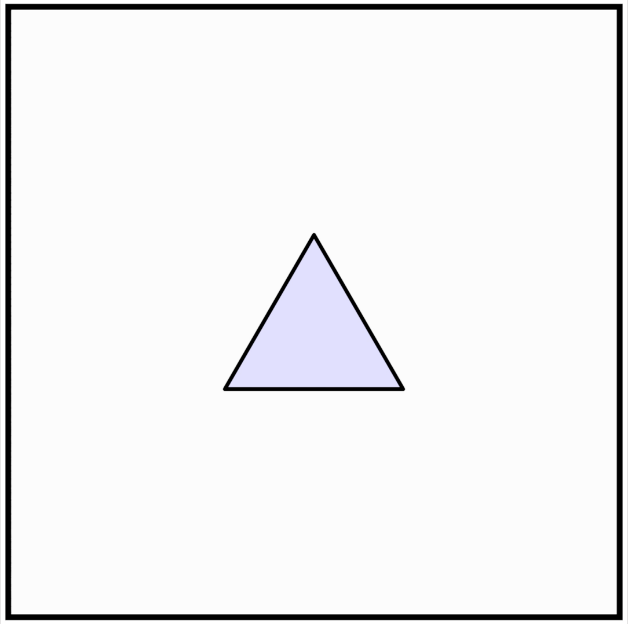
Translation in Homogeneous Coordinates

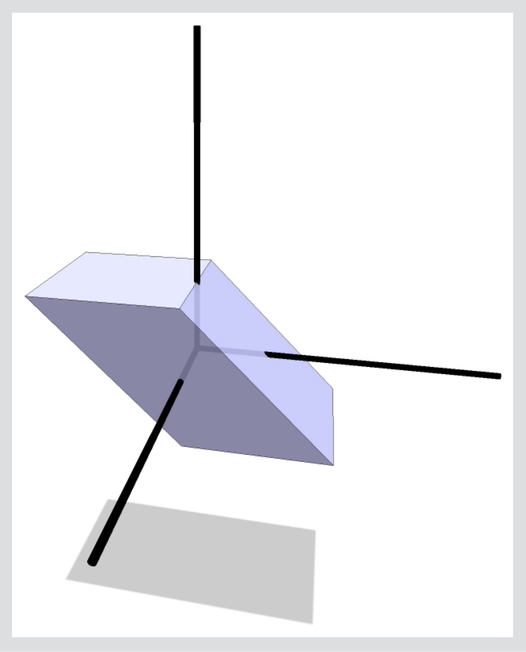
- A 2D translation is similar to a 3D shear
  - Moving a slice up/down the shear moves the shape
- Recall shear is written as:

$$f_{\mathbf{u},\mathbf{v}}(\mathbf{x}) = \mathbf{x} + \langle \mathbf{v}, \mathbf{x} \rangle \mathbf{u}$$

$$f_{\mathbf{u},\mathbf{v}}(\mathbf{x}) = \left(I + \mathbf{u}\mathbf{v}^{\mathsf{T}}\right)\mathbf{x}$$

- In our case, $v = (0, 0, 1)$, so**

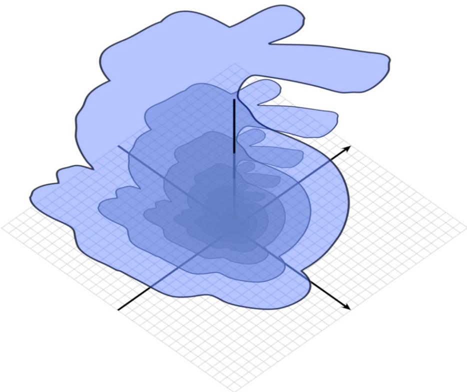




$$\begin{bmatrix} 1 & 0 & u_1 \\ 0 & 1 & u_2 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} cp_1 \\ cp_2 \\ c \end{bmatrix} = \begin{bmatrix} c(p_1 + u_1) \\ c(p_2 + u_2) \\ c \end{bmatrix} \xrightarrow{1/c} \begin{bmatrix} p_1 + u_1 \\ p_2 + u_2 \end{bmatrix}$$

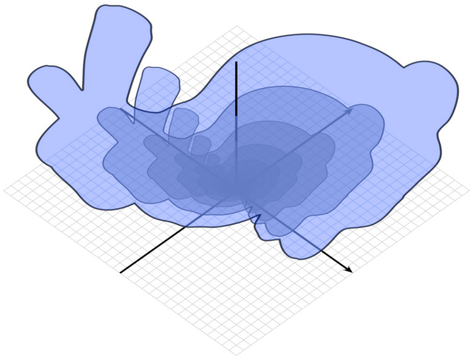
**most often in this class we will also use c = 1

2D Transforms in Homogeneous Coordinate



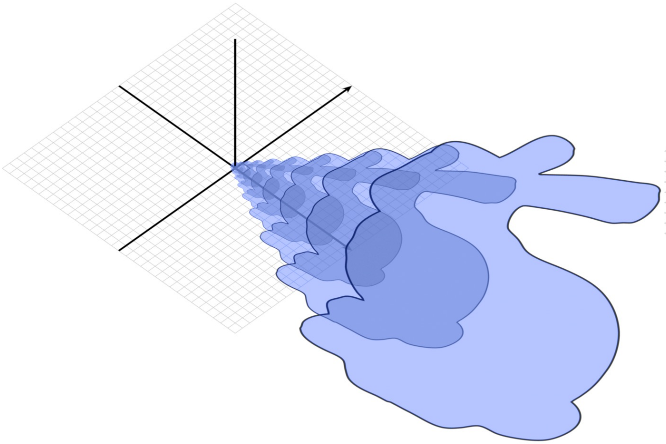
[original]

Original shape in 2D can be viewed as many copies along the z-axis



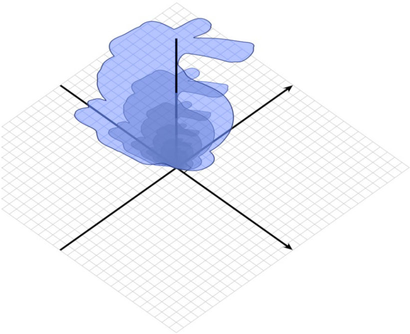
[2D rotation]

Rotate around the z-axis



[2D translate]

Shear in direction of translation



[2D scale]

Scale x-axis and y-axis, preserve z-axis

Q: What about 3D homogeneous coordinates?

3D Transforms in Homogeneous Coordinate

$$\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

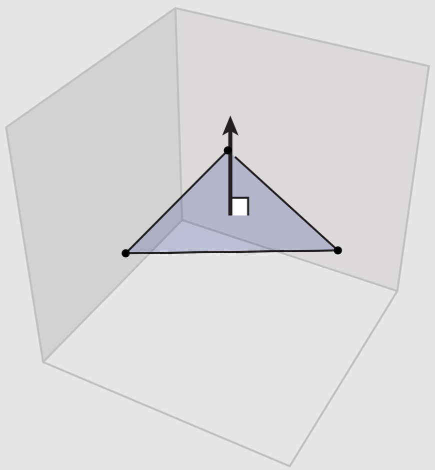
[point in 3D]

Matrix representations of 3D linear transformations just get an additional identity row/column:

$$\begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \begin{bmatrix} 1 & 0 & s & 0 \\ 0 & 1 & t & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \begin{bmatrix} a & 0 & 0 & 0 \\ 0 & b & 0 & 0 \\ 0 & 0 & c & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \begin{bmatrix} 1 & 0 & 0 & u \\ 0 & 1 & 0 & v \\ 0 & 0 & 1 & w \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

[rotate around y by θ] [shear by z in (s,t) direction] [scale by a,b,c] [translate by (u,v,w)]

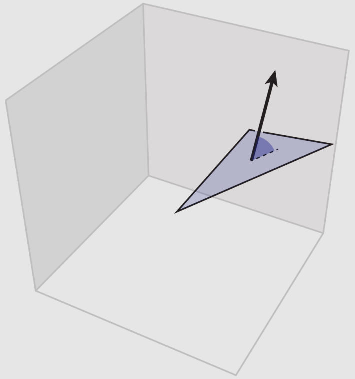
Points vs. Vectors



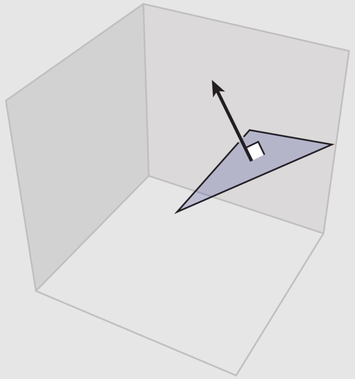
- Homogeneous coordinates should be used differently for points and vectors:
  - Triangle vertices are "points" and should be translated and rotated
    - But if we do the same for the normal, it no longer becomes a normal

  - **Idea:** normal is a "vector" and should just rotate!**
    - Set homogeneous coordinate to 0

$$\begin{bmatrix} \cos\theta & 0 & \sin\theta & u \\ 0 & 1 & 0 & v \\ -\sin\theta & 0 & \cos\theta & w \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} n_1 \\ n_2 \\ n_3 \\ 1 \end{bmatrix}$$



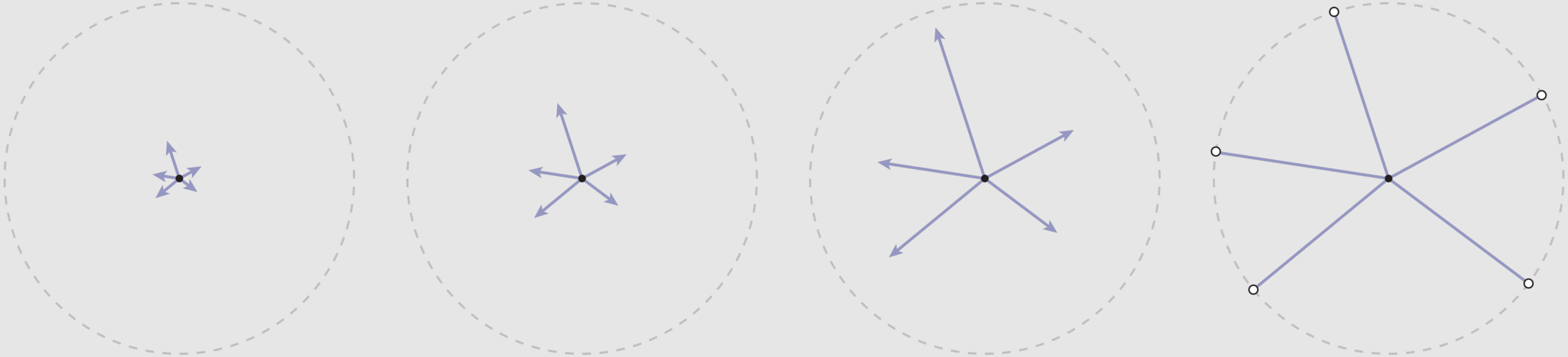
$$\begin{bmatrix} \cos\theta & 0 & \sin\theta & u \\ 0 & 1 & 0 & v \\ -\sin\theta & 0 & \cos\theta & w \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} n_1 \\ n_2 \\ n_3 \\ 0 \end{bmatrix}$$



**translating or scaling a triangle should never change the normal

Points vs. Vectors in Homogeneous Coordinates

- In general:
  - A point has a nonzero homogeneous coordinate (c = 1)
  - A vector has a zero homogeneous coordinate (c = 0)
- But wait… what division by c mean when it's equal to zero?
- Well consider what happens as c approaches 0…



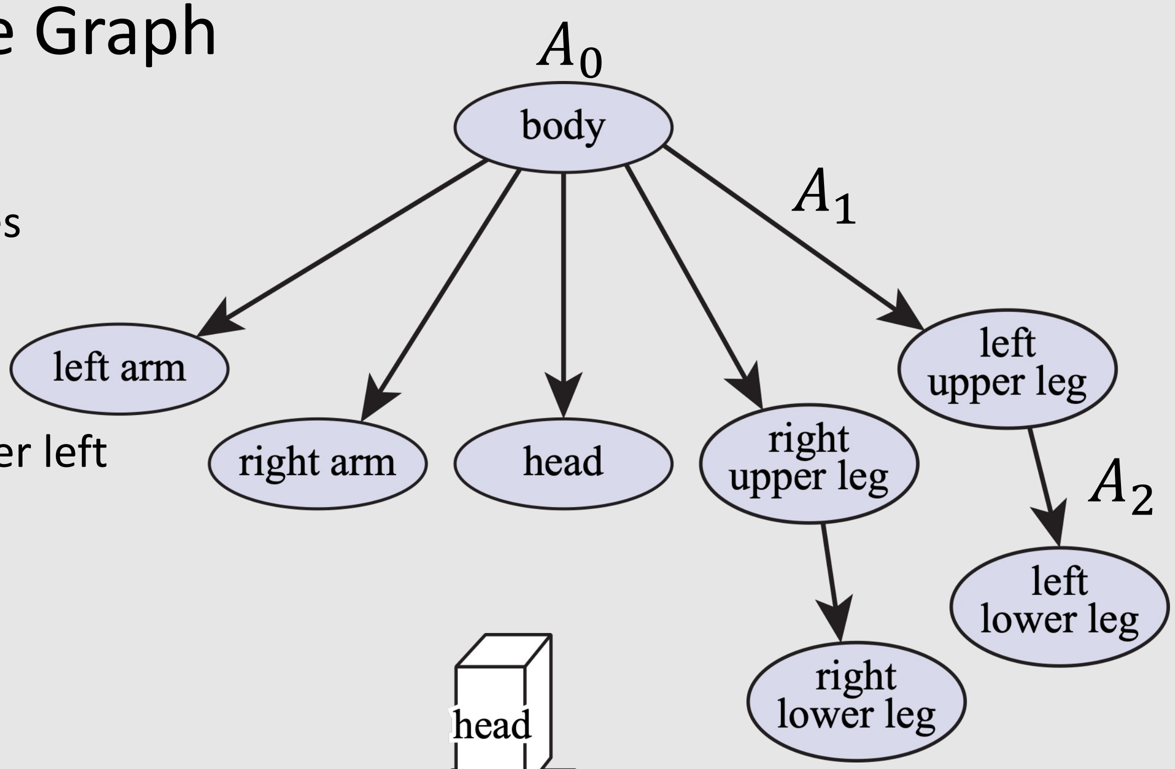
$(x, y)/1$ $(x, y)/0.5$ $(x, y)/0.25$ $(x, y)/0.001$

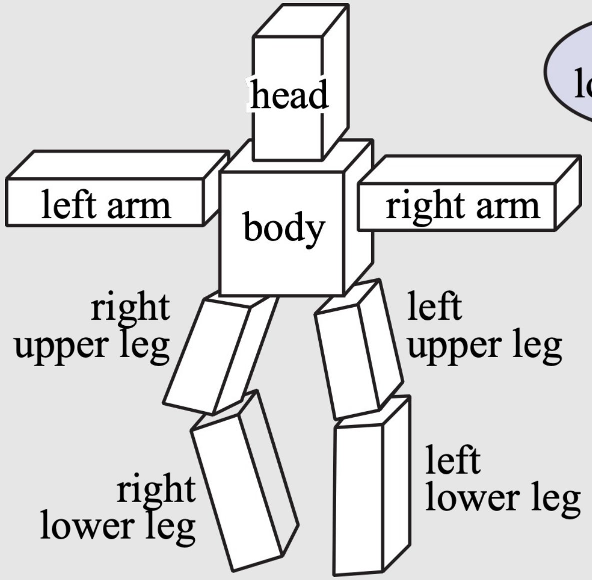
- Can think of vectors as "points at infinity" (sometimes called "ideal points")
  - **But don't actually go dividing by zero…**

Where can we use transforms?

Scene Graph

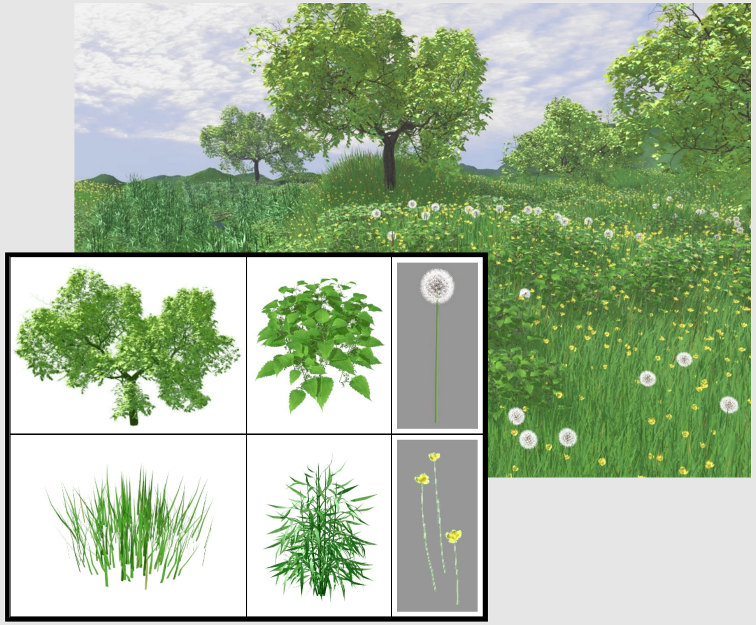
- Suppose we want to build a skeleton out of cubes
  - **Idea:** transform cubes in world space
    - Store transform of each cube
- **Problem:** If we rotate the left upper leg, the lower left leg won't track with it
  - **Better Idea:** store a hierarchy of transforms
    - Known as a **scene graph**
    - Each edge (+root) stores a linear transformation
    - Composition of transformations gets applied to nodes
      - Keep transformations on a stack to reduce redundant multiplication
- **Lower left leg transform:** $A_2 A_1 A_0$



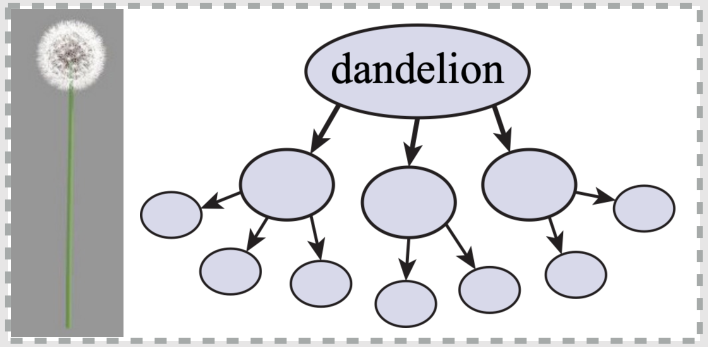


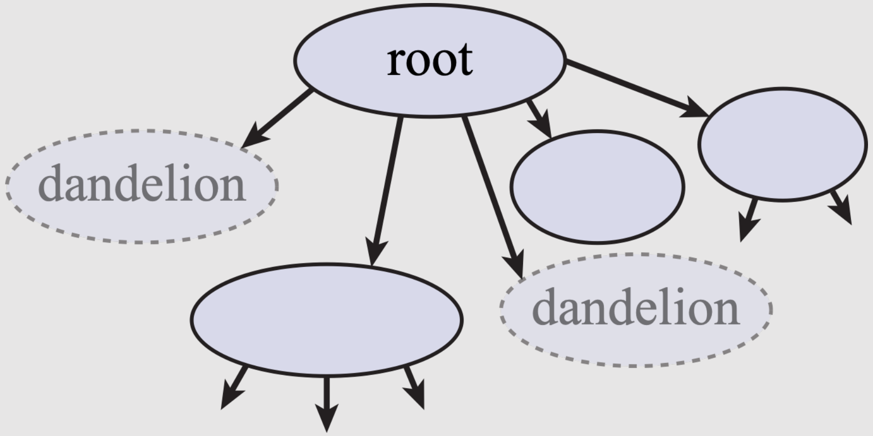
Instancing

- What if we want many copies of the same object in a scene?
  - Rather than have many copies of the geometry, scene graph, we can just put a "pointer" node in our scene graph
    - Saves a reference to a shared geometry
    - Specify a transform for each reference
      - **Careful!** Modifying the geometry will modify all references to it



Realistic modeling and rendering of plant ecosystems (1998) Deussen et al



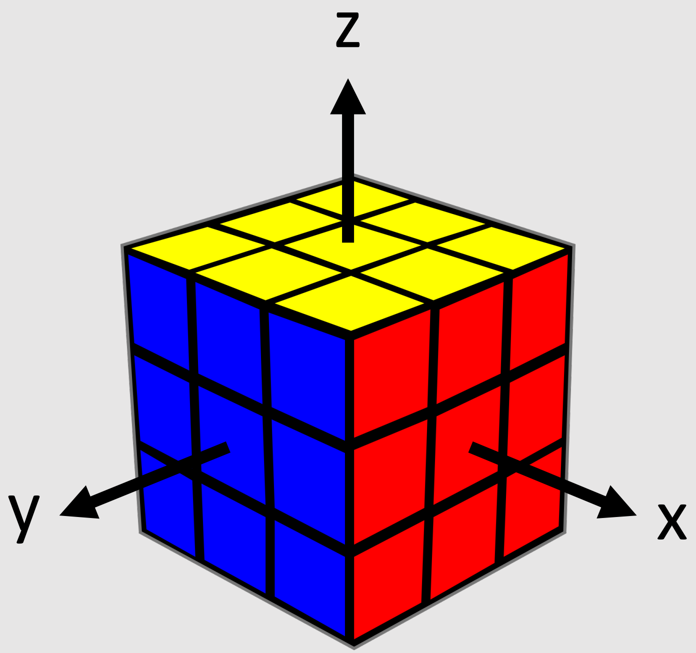


- ~~The Rasterization Pipeline~~
- ~~Transformations~~
- ~~Homogeneous Coordinates~~
- 3D Rotations

3D Rotations

- Rotating in 2D is the same as rotating around the z-axis
- **Idea:** independently rotate around each (x,y,z)-axis for 3D rotations

- **Problem:** order of rotation matters!
  - Rotate a Rubik's cube 90deg around the y-axis and 90deg around the z-axis
  - Rotate a Rubik's cube 90deg around the z-axis and 90deg around the y-axis
    - They will not be the same!
  - Order of rotation must be specified



3D Rotations in Matrix Form – Euler Angles

Idea: independently rotate around each (x,y,z)-axis for 3D rotations:

$$R_x = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta_x & -\sin\theta_x \\ 0 & \sin\theta_x & \cos\theta_x \end{bmatrix} \quad R_y = \begin{bmatrix} \cos\theta_y & 0 & \sin\theta_y \\ 0 & 1 & 0 \\ -\sin\theta_y & 0 & \cos\theta_y \end{bmatrix} \quad R_z = \begin{bmatrix} \cos\theta_z & -\sin\theta_z & 0 \\ \sin\theta_z & \cos\theta_z & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Combining the matrices:

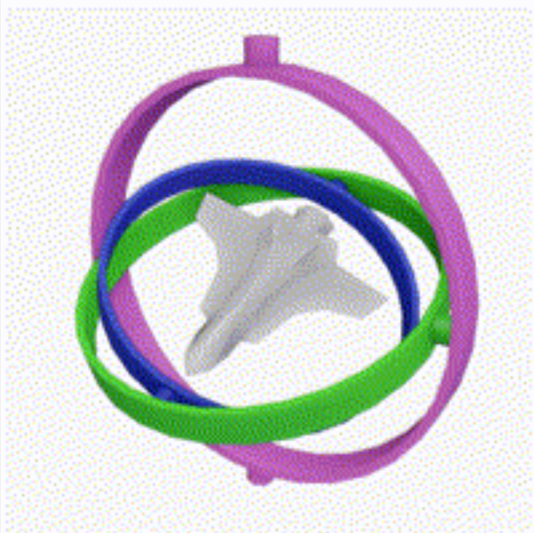
$$R_xR_yR_z = \begin{bmatrix} \cos\theta_y\cos\theta_z & -\cos\theta_y\sin\theta_z & \sin\theta_y \\ \cos\theta_z\sin\theta_x\sin\theta_y + \cos\theta_x\sin\theta_z & \cos\theta_x\cos\theta_z - \sin\theta_x\sin\theta_y\sin\theta_z & -\cos\theta_y\sin\theta_x \\ -\cos\theta_x\cos\theta_z\sin\theta_y + \sin\theta_x\sin\theta_z & \cos\theta_z\sin\theta_x + \cos\theta_x\sin\theta_y\sin\theta_z & \cos\theta_x\cos\theta_y \end{bmatrix}$$

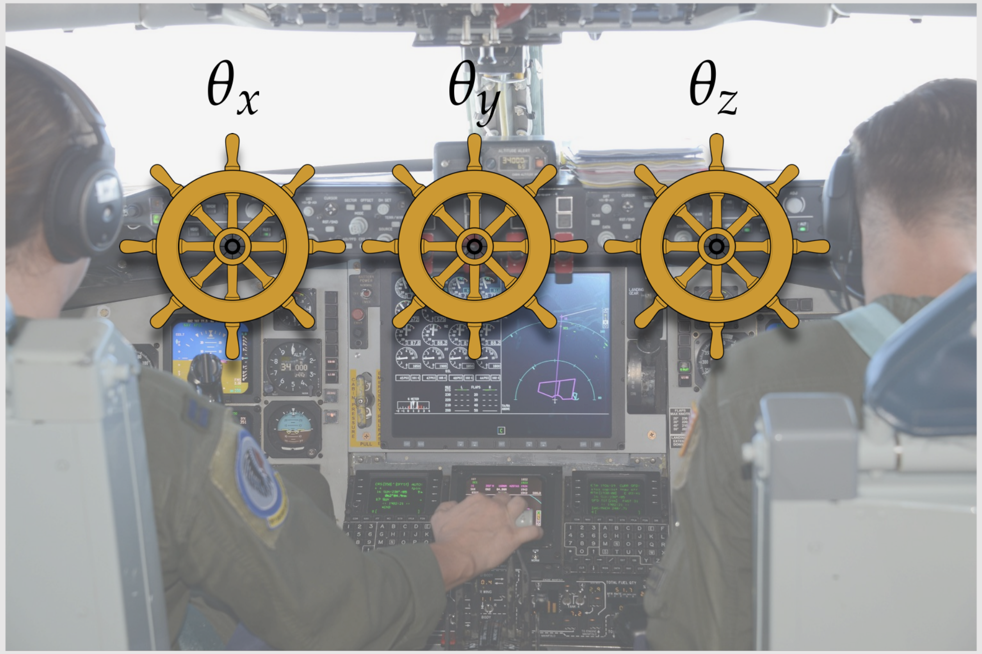
Consider the special case $\theta_y = \pi/2$ (so, $\cos\theta_y = 0$, $\sin\theta_y = 1$):

$$\Longrightarrow \begin{bmatrix} 0 & 0 & 1 \\ \cos\theta_z\sin\theta_x + \cos\theta_x\sin\theta_z & \cos\theta_x\cos\theta_z - \sin\theta_x\sin\theta_z & 0 \\ -\cos\theta_x\cos\theta_z + \sin\theta_x\sin\theta_z & \cos\theta_z\sin\theta_x + \cos\theta_x\sin\theta_z & 0 \end{bmatrix}$$

Gimbal Lock

- **No matter how we adjust θx, θz, can only rotate in one plane!**
- We are now “locked” into a single axis of rotation
  - Not a great design for airplane controls!





$$\Longrightarrow \begin{bmatrix} 0 & 0 & 1 \\ \cos\theta_z \sin\theta_x + \cos\theta_x \sin\theta_z & \cos\theta_x \cos\theta_z - \sin\theta_x \sin\theta_z & 0 \\ -\cos\theta_x \cos\theta_z + \sin\theta_x \sin\theta_z & \cos\theta_z \sin\theta_x + \cos\theta_x \sin\theta_z & 0 \end{bmatrix}$$

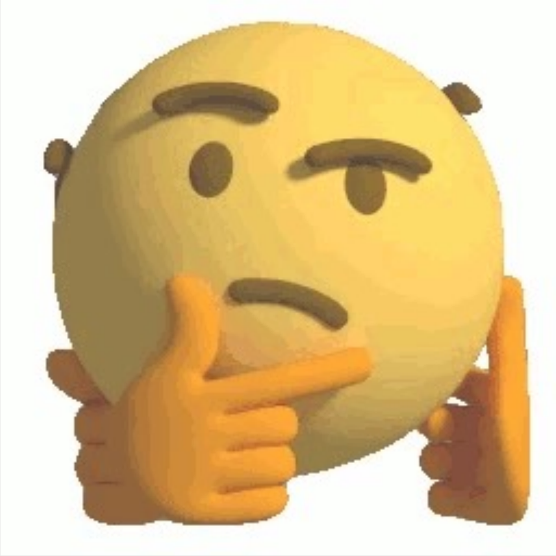
Rotation From Axis/Angle

Alternatively, there is a general expression for a matrix that performs a rotation around a given axis u by a given angle θ:

$$\begin{bmatrix} \cos\theta + u_x^2(1-\cos\theta) & u_x u_y(1-\cos\theta) - u_z\sin\theta & u_x u_z(1-\cos\theta) + u_y\sin\theta \\ u_y u_x(1-\cos\theta) + u_z\sin\theta & \cos\theta + u_y^2(1-\cos\theta) & u_y u_z(1-\cos\theta) - u_x\sin\theta \\ u_z u_x(1-\cos\theta) - u_y\sin\theta & u_z u_y(1-\cos\theta) + u_x\sin\theta & \cos\theta + u_z^2(1-\cos\theta) \end{bmatrix}$$

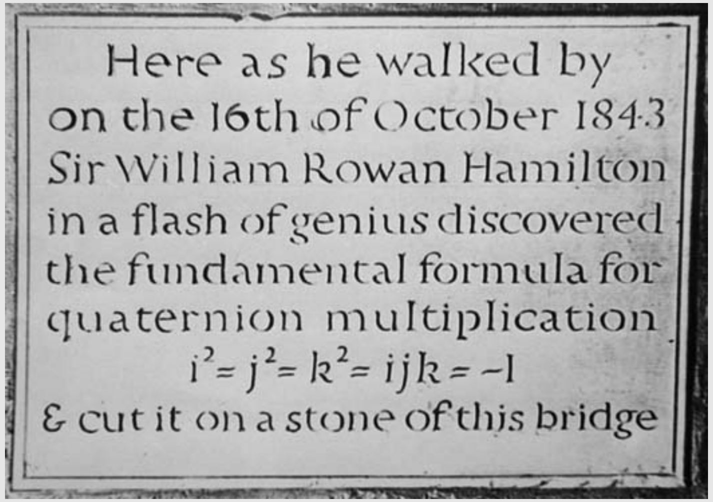
Just memorize this matrix! :)

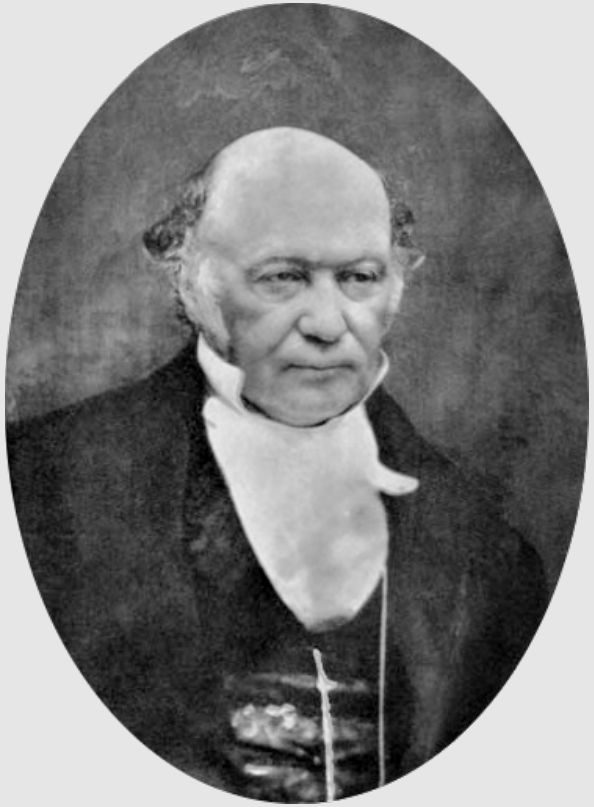
Is there a better way to perform 3D rotations?



Bridging The Rotation Gap

- Hamilton wanted to make a 3D equivalent for complex numbers
  - One day, when crossing a bridge, he realized he needed 4 (not 3) coordinates to describe 3D complex number space
    - 1 real and 3 complex components
  - He carved his findings onto a bridge (still there in Dublin)
  - Later known as quaternions





William Rowan Hamilton
[1805 – 1865]

Quaternions For Math People

- 4 coordinates (1 real, 3 complex) comprise coordinates.
  - H is known as the 'Hamilton Space'

$$\mathbb{H} := \text{span}(\{1, \imath, \jmath, k\})$$

$$q = a + b\imath + c\jmath + dk \in \mathbb{H}$$

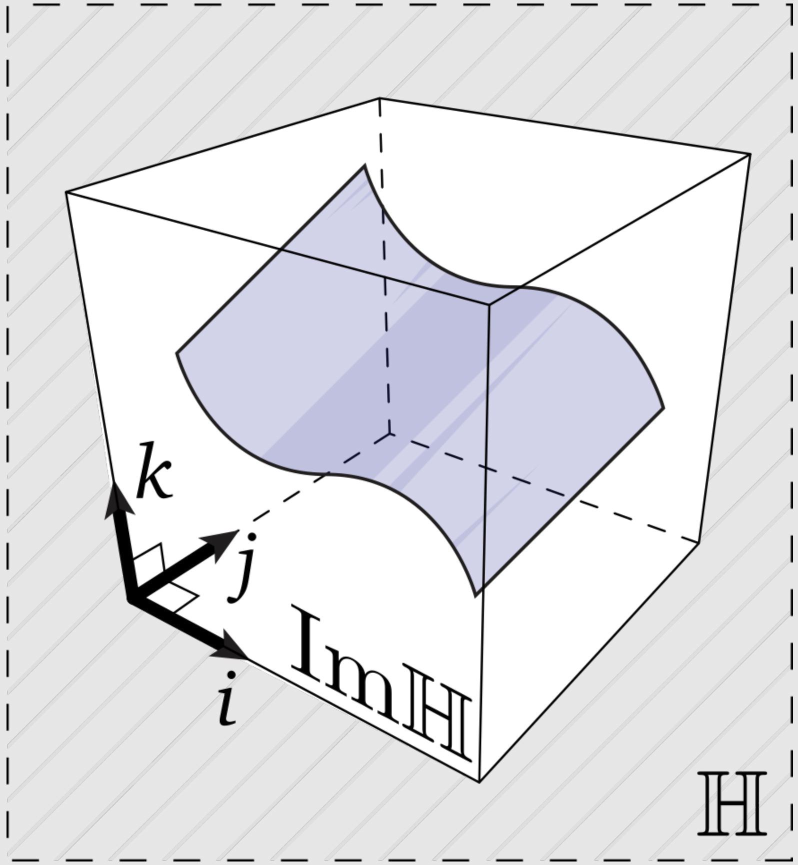
- Quaternion product determined by:

$$\imath^2 = \jmath^2 = k^2 = \imath\jmath k = -1$$

- **Warning:** product no longer commutes!

$$\text{For } q, p \in \mathbb{H}, \quad qp \neq pq$$

  - With 3D rotations, order matters.



Quaternions For Non-Math People

- Recall axis-angle rotations
  - Represent an axis with 3 coordinates (i, j, k)
  - Represent an angle by some scalar a

$$q = a + b\imath + c\jmath + dk \in \mathbb{H}$$

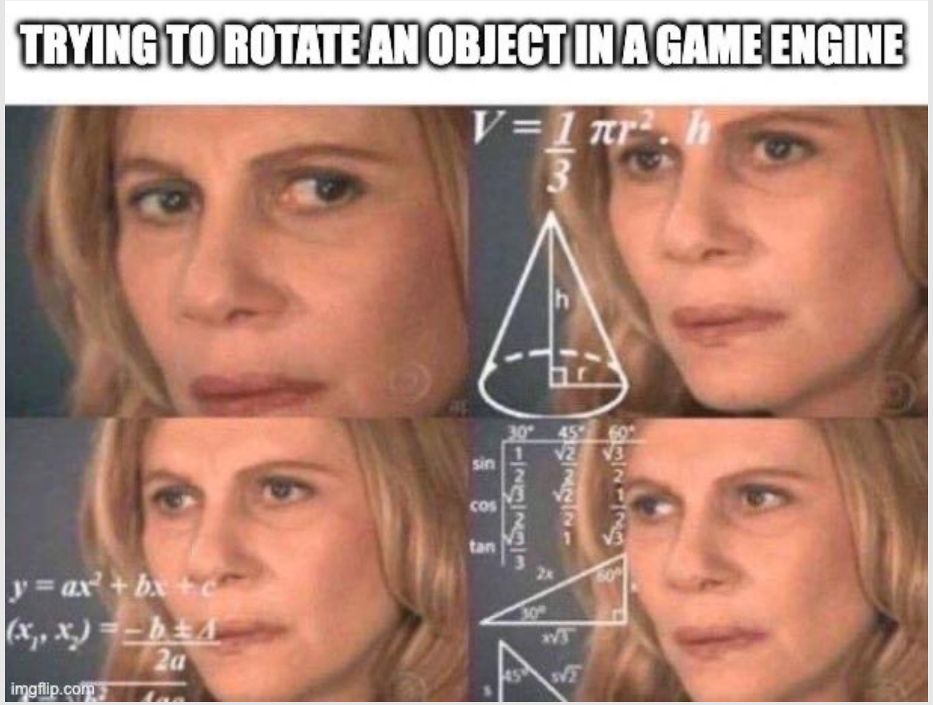
- Just like how we multiply rotation matrices together, we can also multiply complex components. If we represent:
  - i as a 90deg rotation about x-axis
  - j as a 90deg rotation about y-axis
  - k as a 90deg rotation about z-axis

$$\imath^2 = \jmath^2 = k^2 = \imath\jmath k = -1$$

  - Then two 90deg rotations about the same axis will produce the inverted image, the same as scaling by -1
  - This can also be rewritten as:

$$ij = k$$

    - A 90deg x-axis rotation and a 90deg y-axis rotation is the same as a 90deg z-axis rotation
    - Can be rewritten in any other way



Multiplying Quaternions

Given two quaternions:

$$q = a_1 + b_1\imath + c_1\jmath + d_1k$$
$$p = a_2 + b_2\imath + c_2\jmath + d_2k$$

Can express their product as:

$$\begin{aligned} qp = &\boxed{a_1a_2 - b_1b_2 - c_1c_2 - d_1d_2} \\ &+(a_1b_2 + b_1a_2 + c_1d_2 - d_1c_2)\imath \\ &+(a_1c_2 - b_1d_2 + c_1a_2 + d_1b_2)\jmath \\ &+(a_1d_2 + b_1c_2 - c_1b_2 + d_1a_2)k \end{aligned}$$

recall

$$\imath^2 = \jmath^2 = k^2 = \imath\jmath k = -1$$

The result still looks like a quaternion
But there's a better way to multiply…

Multiplying Quaternions

Recall quaternions can be thought of as an axis and angle:

$$(x, y, z) \mapsto 0 + x\imath + y\jmath + zk$$

$$\left(\underset{\mathbb{R}}{\underset{\cap}{\text{scalar},}}\ \underset{\mathbb{R}^3}{\underset{\cap}{\text{vector}}}\right) \in \mathbb{H}$$

Can express their product as:

$$(a, \mathbf{u})(b, \mathbf{v}) = (ab - \mathbf{u} \cdot \mathbf{v}, a\mathbf{v} + b\mathbf{u} + \mathbf{u} \times \mathbf{v})$$

If the scalar components are 0, we get:

$$\mathbf{u}\mathbf{v} = \mathbf{u} \times \mathbf{v} - \mathbf{u} \cdot \mathbf{v}$$

Rotating With Quaternions

- **Goal:** rotate x by angle θ around axis $u = (x, y, z)$:
  - Make x imaginary, and build q based on u and θ
    - **Note:** components of q must be normalized!

$$x \in \mathrm{Im}(\mathbb{H})$$

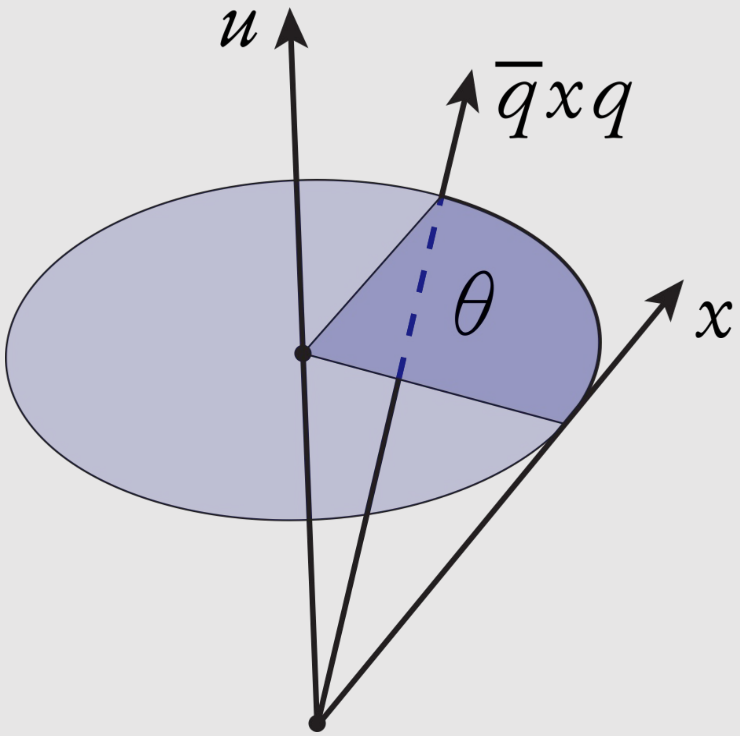
$$q \in \mathbb{H}, \quad |q|^2 = 1$$

$$q = \cos(\theta/2) + \sin(\theta/2)u$$

- q now looks like:

$$q = a + b\imath + c\jmath + dk \in \mathbb{H}$$

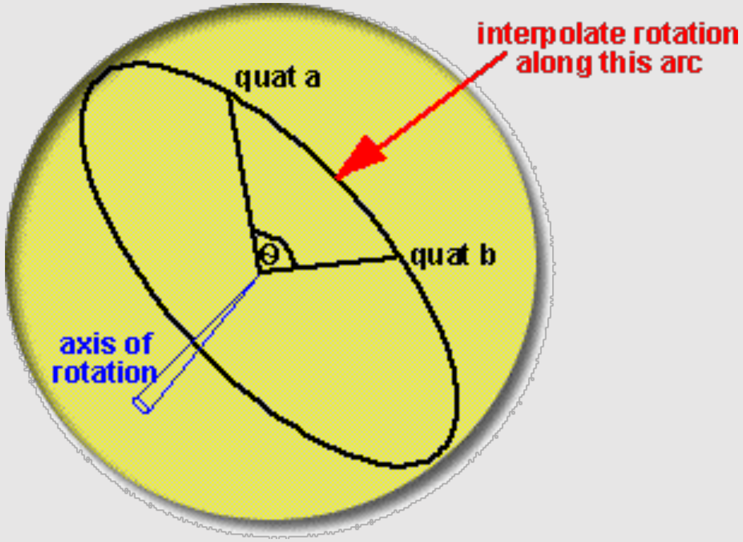
- $\bar{q}$ is q with every complex component negative
- Now just compute $\bar{q}xq$ to get final rotation



Interpolating With Quaternions

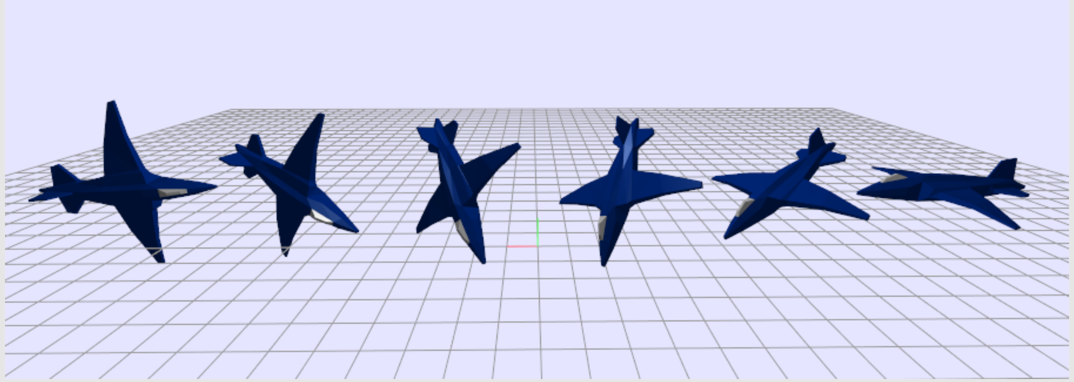
- Interpolating Euler angles can yield strange-looking paths, non-uniform rotation speed, etc.
  - Simple solution w/ quaternions: "SLERP" (spherical linear interpolation):

$$\mathrm{Slerp}(q_0, q_1, t) = q_0(q_0^{-1}q_1)^t, \quad t \in [0,1]$$



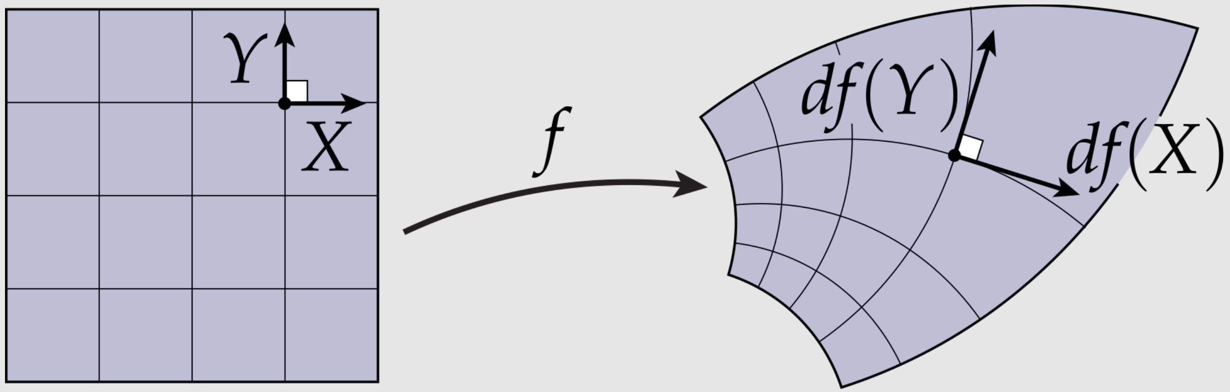
Animating Rotation with Quaternion Curves (1985) Shoemake

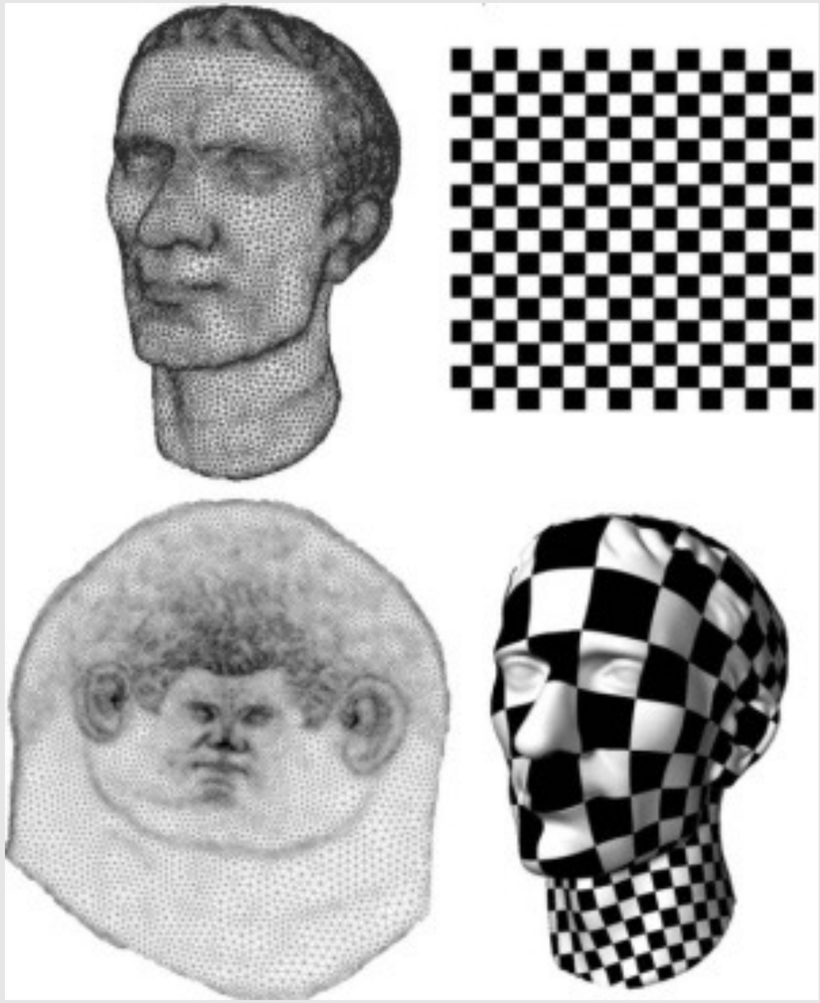
Fifa '15 (2014) Electronic Arts



Texture Mapping With Quaternions

- Quaternions can be used to generate texture maps coordinates
  - Complex numbers are natural language for angle-preserving (“conformal”) maps





Spatial Transformation Summary

[linear transformations]

- scaling
- rotation
- reflection
- shear

[nonlinear transformations]

- translation
- perspective projection

next lecture

- Compose basic transformations to get more interesting ones
  - Always reduces to a single 4x4 matrix (in homogeneous coordinates)
  - Order of composition matters!
- Homogeneous coordinates can turn nonlinear transformations linear
- Many ways to decompose a given transformation (polar, SVD, …)
- Use scene graph to organize transformations
- Use instancing to eliminate redundancy
- Quaternions help avoid troubles with Euler rotations in 3D (Gimbal Lock, Interpolation inconsistencies)


Maxwell the cat (2022) Gary's Mod