CIS367 - Computer Graphics

Geometry and Transforms

Erik Fredericks - frederer@gvsu.edu

Overview

Coordinate systems

Transforms

Frames

Etc.

Geometry objectives

Main elements

- Scalar
- Vector
- Point

Discuss mathematical operations required in **coordinate-free** manner

Define basic primitives in terms of geometry

- Line segment
- Polygon

Coordinate-free?

Generally started with cartesian

- Point is in space: **p** = (x, y, z)
- Manipulations / results using these coordinates

This was **non-physical**

- Point exists regardless of location in arbitrary coordinate system
- Geometric results independent of coordinate system

- Ex: vectors
    - Two vectors are the same even though they appear different

Linear independence and Dimension

The **dimension** of a space is based on the number of **linearly-independent vectors**

Linearly independent

- Can't represent one in terms of another

$$\alpha_1 v_1 + \alpha_2 v_2 + .. \ \alpha_n v_n = 0 \text{ iff } \alpha_1 = \alpha_2 = \ldots = 0$$

Vector space ➔ maximum number of linearly-independent vectors is fixed

- Forms its *basis*

Basis vector: $v_1, v_2, \ldots, v_n$, can write any vector: $v = \alpha_1 v_1 + \alpha_2 v_2 + \ldots + \alpha_n v_n$

- Assuming $\{\alpha_i\}$ unique

Representation

So far, no real concern for representation!
➜ No frame of reference

Need one to relate points to physical world

- Relating to … eyeball? camera?
    - World?

Coordinate system

Basis: $v_1, v_2, \dots, v_n$

Vector: $a_1v_1, a_2v_2, \dots, a_nv_n$

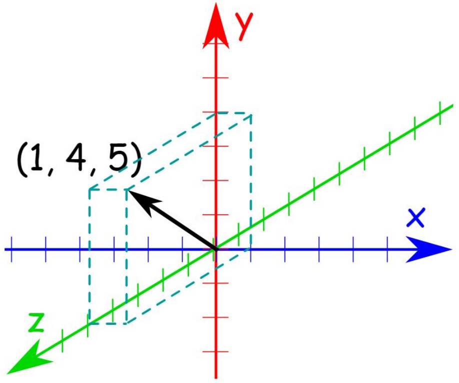
List of scalars $\{a_1, a_2, \dots, a_n\}$ is the **representation** of v with respect to given basis

a $= [a_1, a_2, \dots, a_n]^\mathbf{T} = \begin{pmatrix} a_1, \\ a_2, \\ \dots \\ a_n \end{pmatrix}$

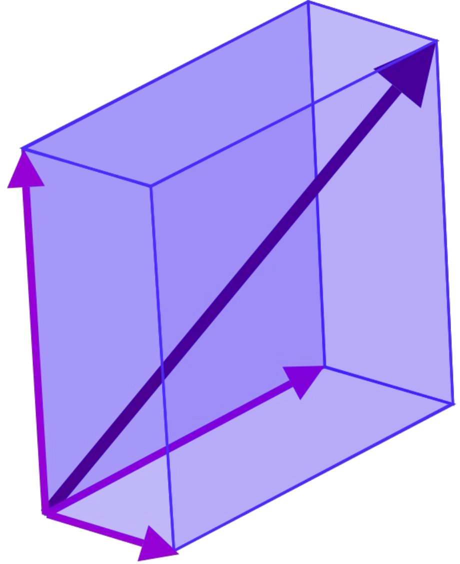
Example

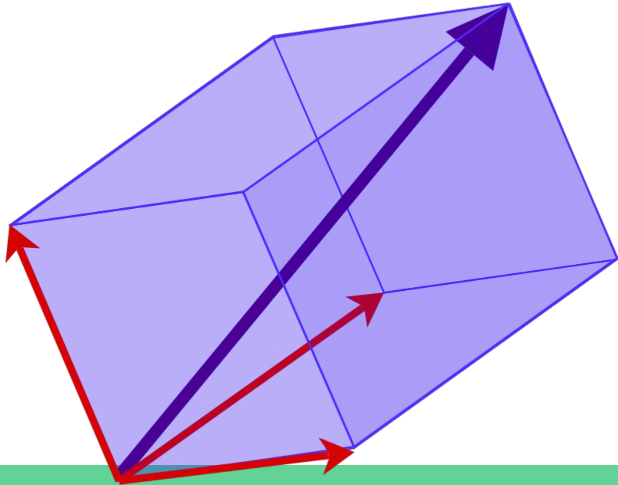
$v = 2v_1 + 3v_2 - 4v_3$

$\mathbf{a} = [2\ 3\ -4]^T$



The vector (1,4,5)

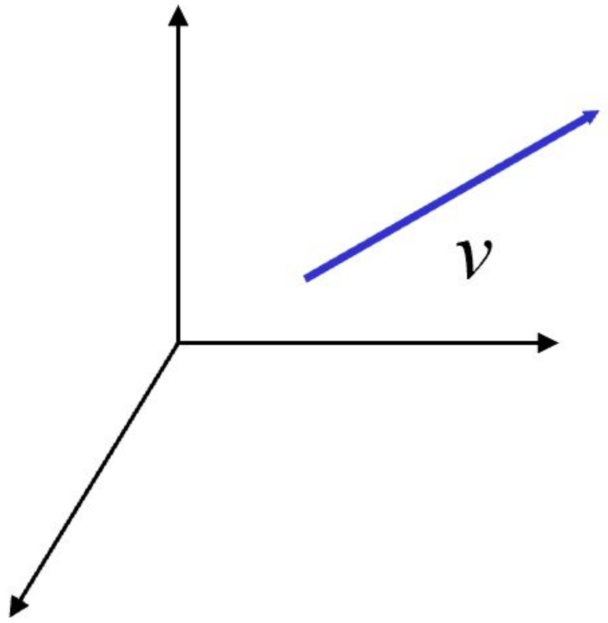


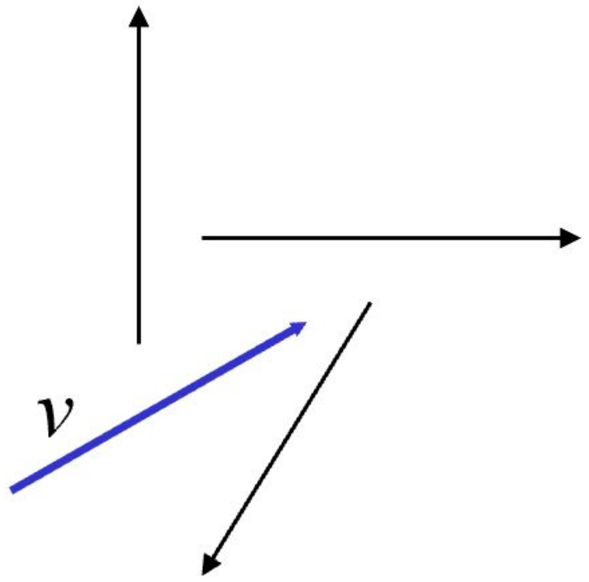


This is with respect to a particular **basis**!

- Later, will be in terms of eye or camera

What's the difference?



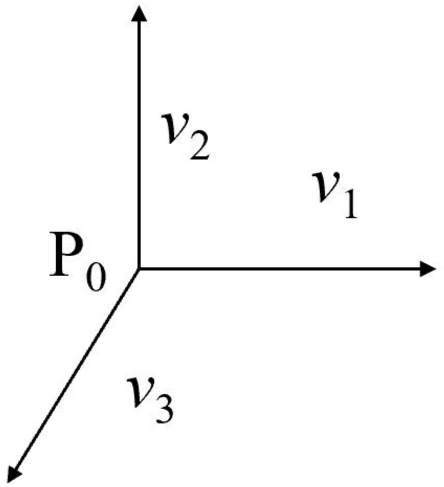


Frame

Coordinate system **insufficient** to represent points

Affine space -- add a single point (**origin**) to basis vectors

- Forms the **frame**



Frame

Frame is: (P_0, v_1, v_2, v_3)

Vectors written as:

$$v = \alpha_1 v_1 + \alpha_2 v_2 + \ldots . + \alpha_n v_n$$

And points written as:

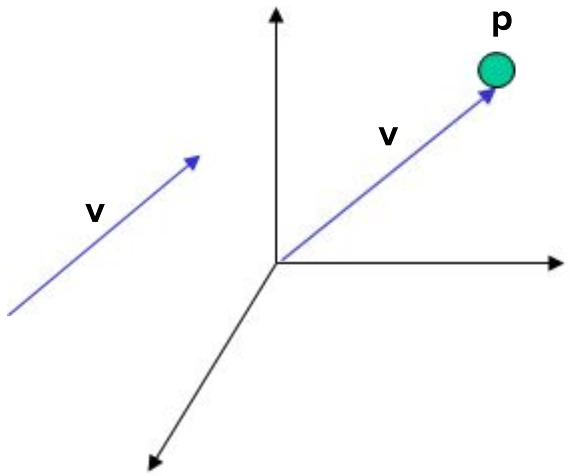
$$P = P_0 + \beta_1 v_1 + \beta_2 v_2 + \ldots . + \beta_n v_n$$

Confusion!

$$P = P_0 + \beta_1 v_1 + \beta_2 v_2 + \ldots . + \beta_n v_n$$

$$v = \alpha_1 v_1 + \alpha_2 v_2 + \ldots . + \alpha_n v_n$$

$\mathbf{p} = [\beta_1\ \beta_2\ \beta_3]$ $\mathbf{v} = [\alpha_1\ \alpha_2\ \alpha_3]$



So what does this get us?

- The ability to *change our representations as needed*
- We'll do this through **homogeneous coordinates**

Basic representation

If $0 \bullet P = 0$ and $1 \bullet P = P$, then:

$v = \alpha_1 v_1 + \alpha_2 v_2 + \alpha_3 v_3 = [\alpha_1\ \alpha_2\ \alpha_3\ 0]\ [v_1\ v_2\ v_3\ P_0]^T$

$P = P_0 + \beta_1 v_1 + \beta_2 v_2 + \beta_3 V_3 = [\beta_1\ \beta_2\ \beta_3\ 1]\ [v_1\ v_2\ v_3\ P_0]^T$

Then, we end up with a **four-dimensional homogeneous coordinate** representation

$\mathbf{v} = [\alpha_1\ \alpha_2\ \alpha_3\ 0]^T$

$\mathbf{p} = [\beta_1\ \beta_2\ \beta_3\ 1]^T$

What are these *homogeneous coordinates*?

Other than the partial answer to a homework question...

These are the **key** to all graphics systems

- All standard transforms implemented using matrix multiplication in 4x4 matrices
    - Rotation, translation, scaling

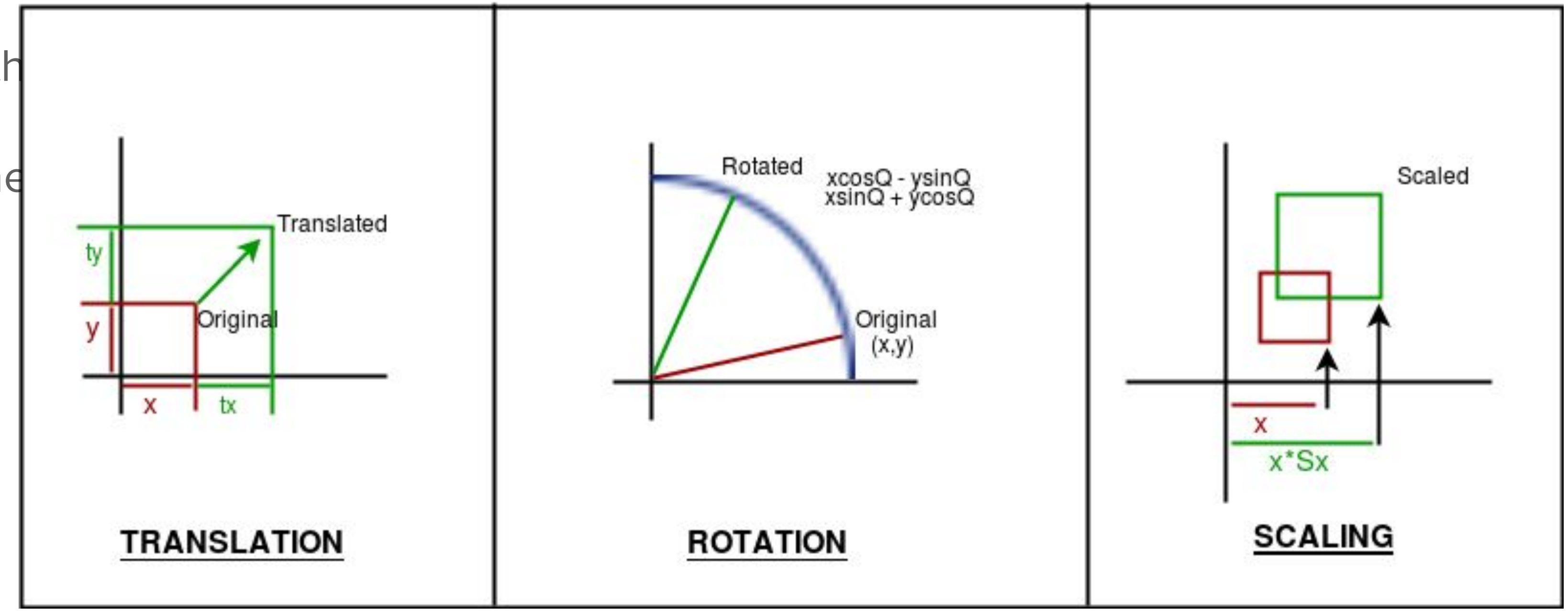
- Hardware pipeline uses 4-D representations
- Orthographic viewing maintains w=0 for vectors, w=1 for points
- Perspective viewing requires *perspective division*

What are these *homogeneous coordinates*?

Oth

The

-
-
-
-



Coordinate change

Let's say we have *two* representations of the same vector with two different bases:

$\mathbf{a} = [\alpha_1\ \alpha_2\ \alpha_3]^T$
$\mathbf{b} = [\beta_1\ \beta_2\ \beta_3]^T$

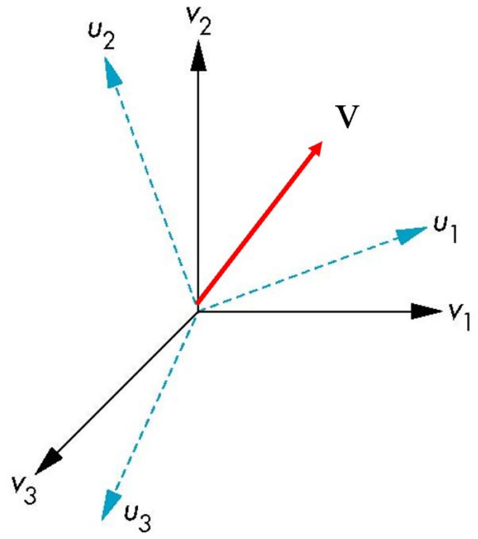
where

$$\begin{aligned} v &= \alpha_1 v_1 + \alpha_2 v_2 + \alpha_3 v_3 &&= [\alpha_1\ \alpha_2\ \alpha_3]\,[v_1\ v_2\ v_3]^T \\ &= \beta_1 u_1 + \beta_2 u_2 + \beta_3 u_3 &&= [\beta_1\ \beta_2\ \beta_3]\,[u_1\ u_2\ u_3]^T \end{aligned}$$

Second basis in terms of first

Each basis vector u_1, u_2, u_3 are vectors represented in terms of the first!

$$u_1 = \gamma_{11}v_1 + \gamma_{12}v_2 + \gamma_{13}v_3$$
$$u_2 = \gamma_{21}v_1 + \gamma_{22}v_2 + \gamma_{23}v_3$$
$$u_3 = \gamma_{31}v_1 + \gamma_{32}v_2 + \gamma_{33}v_3$$



Matrix form

These coefficients (γ) define a 3x3 matrix:

$$M = \begin{bmatrix} \gamma_{11} & \gamma_{12} & \gamma_{13} \\ \gamma_{21} & \gamma_{22} & \gamma_{23} \\ \gamma_{31} & \gamma_{32} & \gamma_{33} \end{bmatrix}$$

Bases related by:

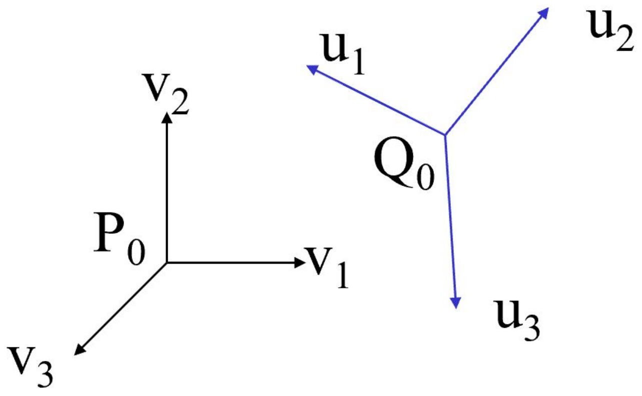
$$\mathbf{a} = \mathbf{M}^T\mathbf{b}$$

Change of frame

Similar procedure in homogeneous coordinates to point/vector representation

Assume we have 2 frames:

(P_0, v_1, v_2, v_3)
(Q_0, u_1, u_2, u_3)



Any point/vector can be represented in either frame!
i.e., represent (Q_0, u_1, u_2, u_3) in terms of (P_0, v_1, v_2, v_3)

Change of frame

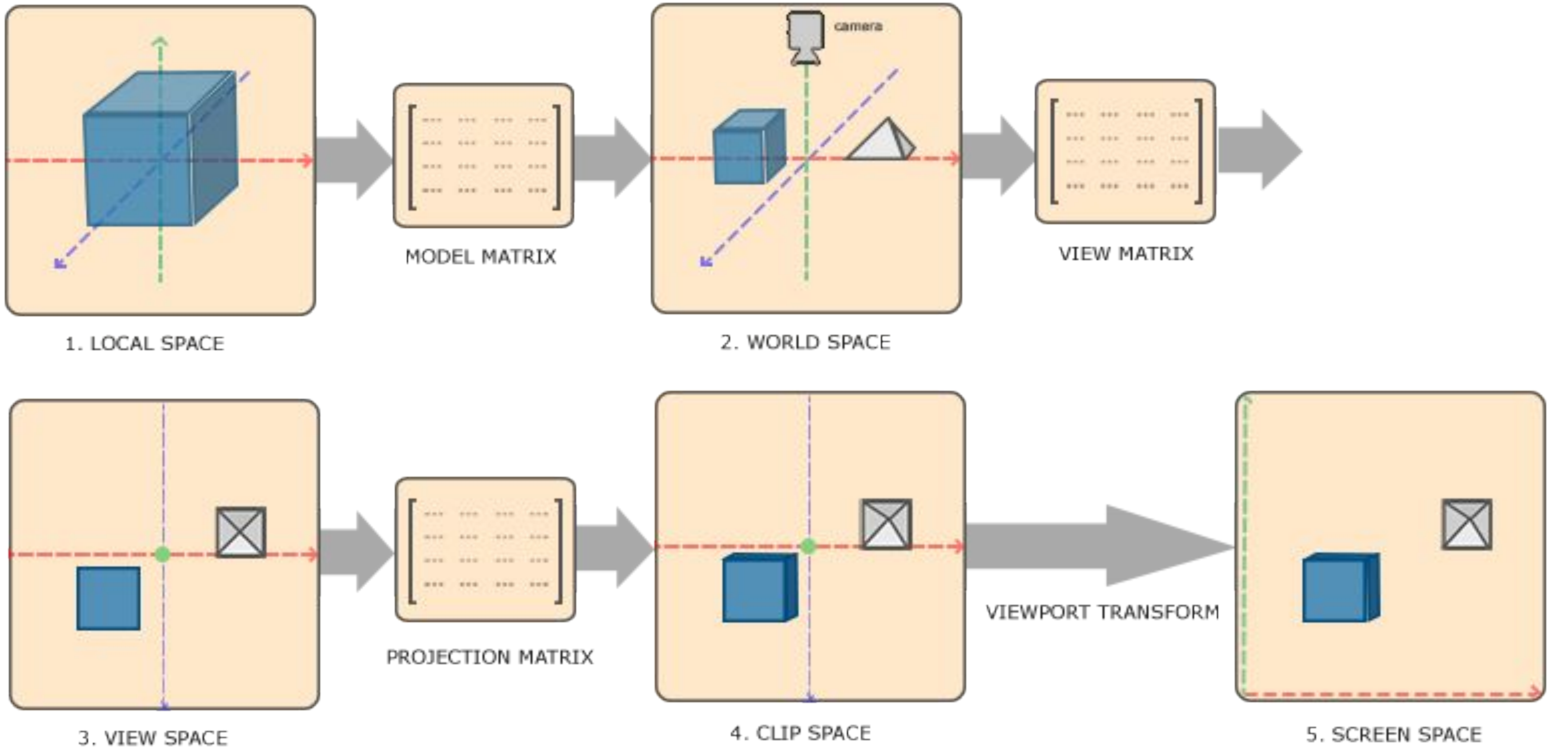
Similar to change of base:

Represent basis/reference point of second frame in terms of first

$M = \begin{bmatrix} \gamma_{11} & \gamma_{12} & \gamma_{13} & 0 \\ \gamma_{21} & \gamma_{22} & \gamma_{23} & 0 \\ \gamma_{31} & \gamma_{32} & \gamma_{33} & 0 \\ \gamma_{41} & \gamma_{42} & \gamma_{43} & 1 \end{bmatrix}$

$u_1 = \gamma_{11}v_1 + \gamma_{12}v_2 + \gamma_{13}v_3$
$u_2 = \gamma_{21}v_1 + \gamma_{22}v_2 + \gamma_{23}v_3$
$u_3 = \gamma_{31}v_1 + \gamma_{32}v_2 + \gamma_{33}v_3$
$Q_0 = \gamma_{41}v_1 + \gamma_{42}v_2 + \gamma_{43}v_3 + \gamma_{44}P_0$

where $\mathbf{a} = \mathbf{M}^T\mathbf{b}$


camera
MODEL MATRIX
VIEW MATRIX
1. LOCAL SPACE
2. WORLD SPACE
PROJECTION MATRIX
VIEWPORT TRANSFORM
3. VIEW SPACE
4. CLIP SPACE
5. SCREEN SPACE

Representations

In these frames, any point/vector has a representation of the *same form*

a = $[\alpha_1\ \alpha_2\ \alpha_3\ \alpha_4]$ ➜ first frame
b = $[\beta_1\ \beta_2\ \beta_3\ \beta_4]$ ➜ second frame

$\alpha_4 = \beta_4 = 1$ for points, $\alpha_4 = \beta_4 = 0$ for vectors

a = $\mathbf{M}^T\mathbf{b}$ ➜ Matrix **M** is 4x4 and specifies an **affine** transformation in homogeneous coordinates

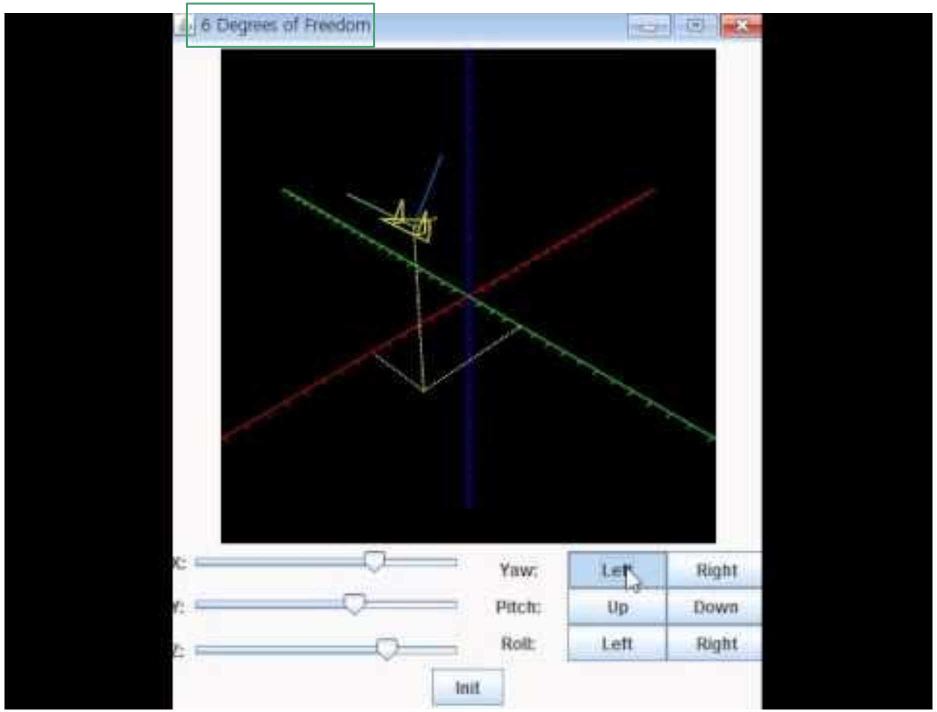
Affine transformations

Every linear transformation is equivalent to a **change in frame**

- Every affine transformation preserves lines

BUT

- Affine transforms have only 12 degrees of freedom
  - 4 elements of matrix are fixed!
  - Subset of all possible 4x4 linear



World/camera frames

Representations mean working with n-tuples / arrays of scalars

- Changes in frame defined by 4x4 matrices

OpenGL (WebGL) ➜ base frame is the **world frame**

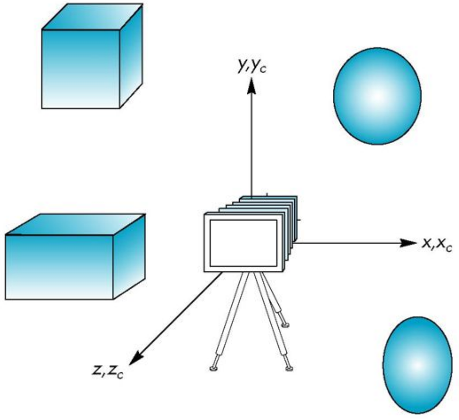
- Eventually, we'll represent entities in camera frame by changing world representation with a model-view matrix

Initially, world/camera are the same (**M** = **I**)

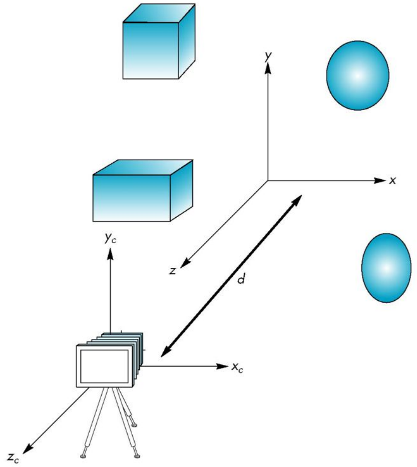
Moving the camera

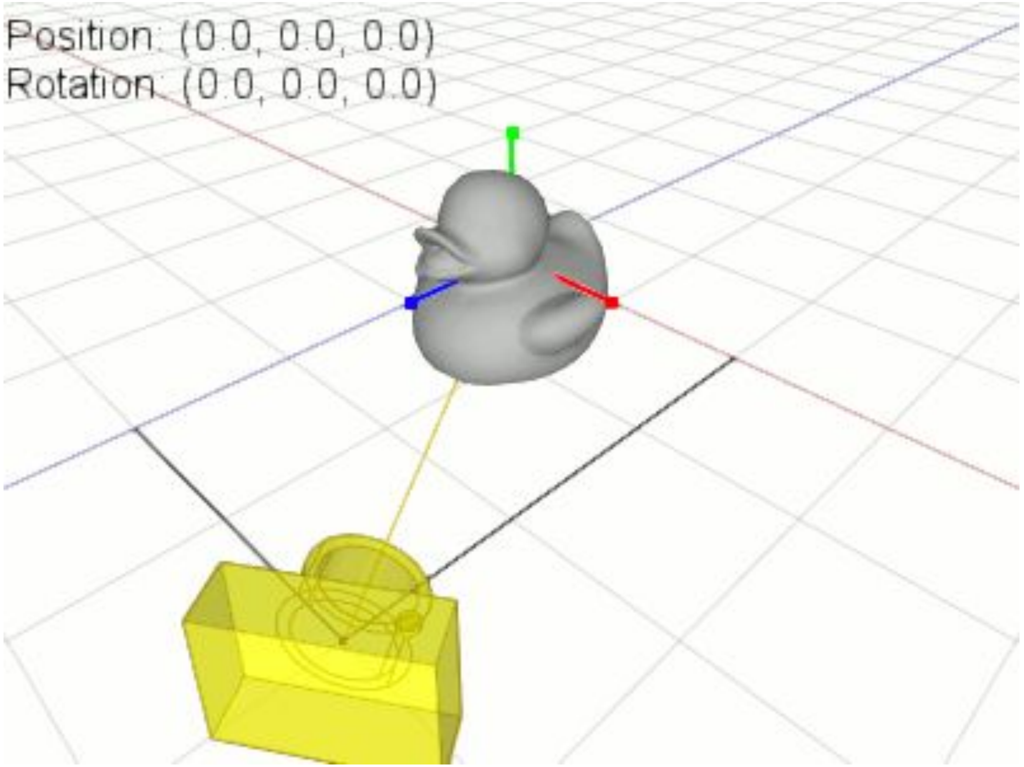
If objects are on both sides of z=0, camera frame must move

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



(a)

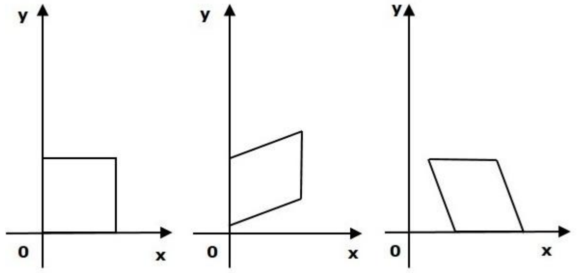



Position: (0.0, 0.0, 0.0)
Rotation: (0.0, 0.0, 0.0)

Now let's talk about transforms

Standard transforms:

- Rotation
- Translation
- Scaling
- Shear



Original object After y shear After x shear

How do we accomplish these?

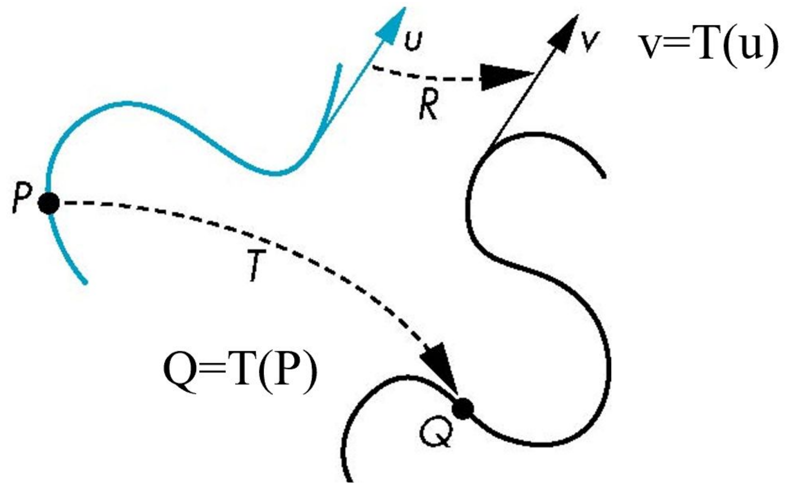
- Homogeneous coordinate transforms!

Transforms

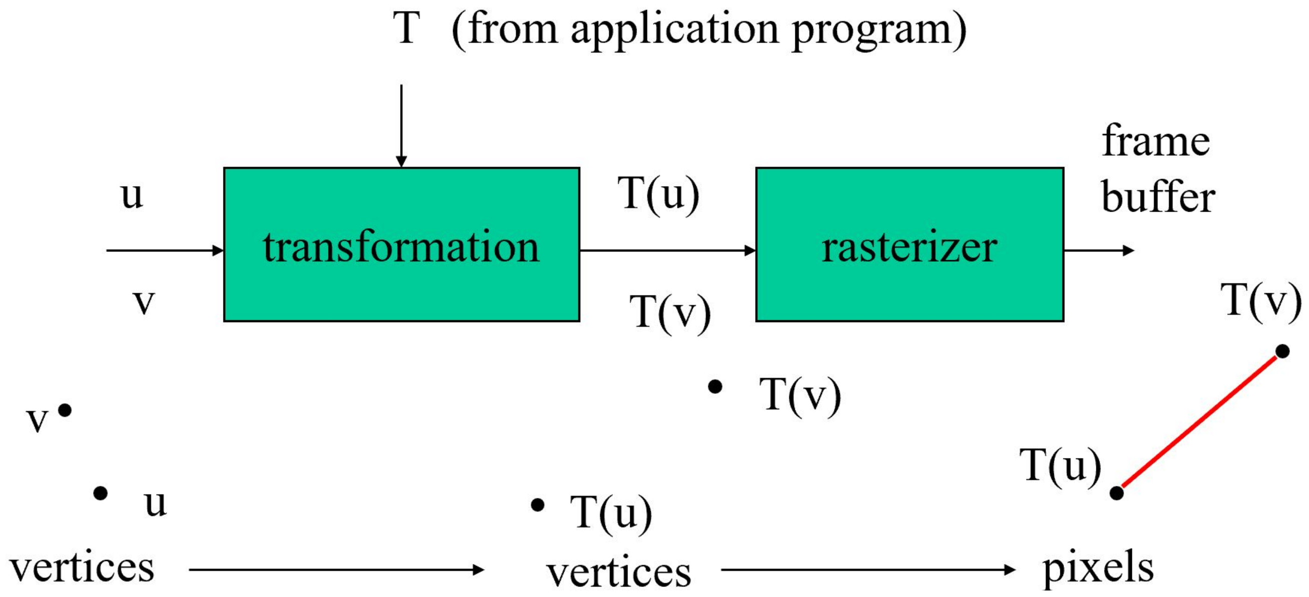
Transform maps points/vectors to other points/vectors, respectively

Affine transform:

- Only need to transform endpoints
- Let implementation draw line!



Pipeline implementation



You may be concerned about notation

(if you're not asleep already)$^{\text{(are professors allowed to say that?)}}$

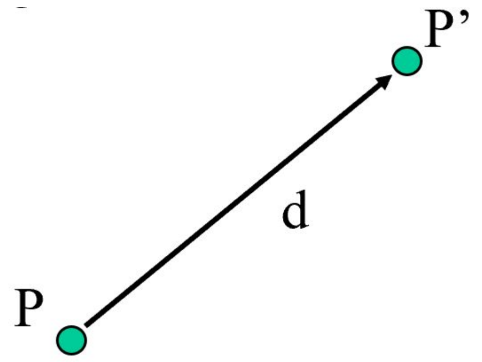
Coordinate-free representations within frame

P, Q, R	➜ points in affine space
u, v, w	➜ vectors in affine space
α, □, ɣ	➜ scalars
p, **q**, **r**	➜ representations of points (4 scalar array in homogeneous coords)
u, **v**, **w**	➜ representations of vectors (4 scalar array in homogeneous coords)

Translation

Transform that moves a point to new location

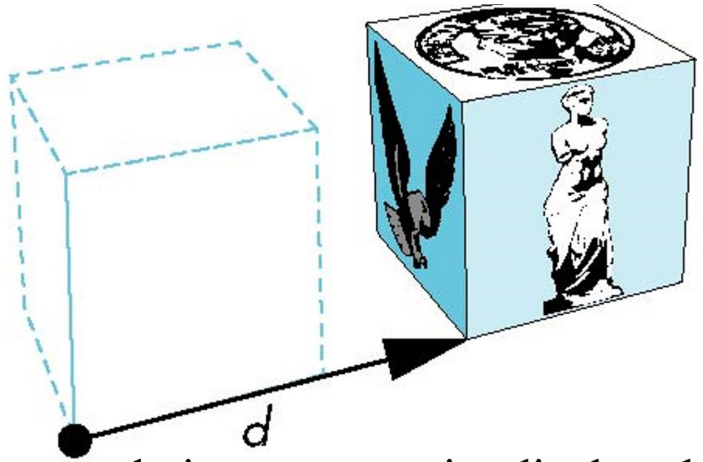
- Translate / displace



Determined by vector d

- 3 degrees of freedom
- P' = P + d

object



translation: every point displaced by same vector

Translation via representation

Using homogeneous coordinate representation in *some frame*

p $= [\ x\ y\ z\ 1\ ]^T$
p' $= [\ x'\ y'\ z'\ 1\ ]^T$
d $= [dx\ dy\ dz\ 0]^T$

This is in 4 dimensions and expresses that point = vector + point!

Making: **p'** = **p** + **d** OR

$x' = x + d_x$
$y' = y + d_y$
$z' = z + d_z$

Translation matrix

Translation also can be expressed with a 4x4 matrix (**T**) in homogeneous coordinates (**p'** = **Tp**)

where,

$$\mathbf{T} = \mathbf{T}(d_x, d_y, d_z) = \begin{bmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Better for implementation!

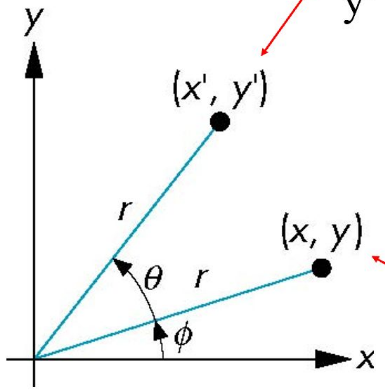
- All affine xforms can be expressed this way
- **Multiple xforms** can be *concatenated together!*

Rotation (2D)

Consider rotation about the origin by θ degrees

- Radius stays same, angle increases by θ

$$x' = r\cos(\phi + \theta)$$
$$y' = r\sin(\phi + \theta)$$



$$\boxed{\begin{aligned} x' &= x\cos\theta - y\sin\theta \\ y' &= x\sin\theta + y\cos\theta \end{aligned}}$$

$$x = r\cos\phi$$
$$y = r\sin\phi$$

Rotation about z axis

What does it mean to rotate about z?

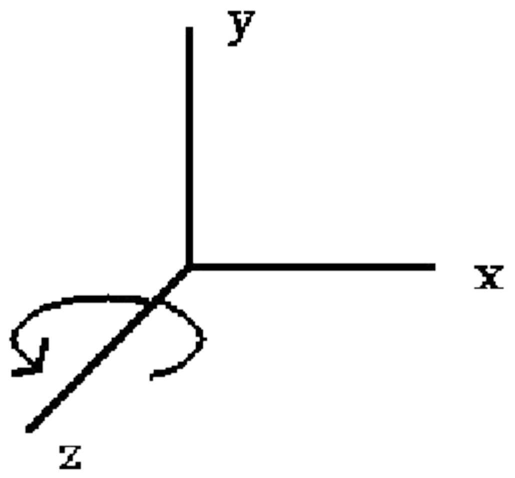
- All points are the **same** in z!
- Equivalent to rotation in 2-D in planes of constant z

$x' = x \cos\theta - y \sin\theta$
$y' = x \sin\theta - y \cos\theta$
$z' = z$

In homogeneous coordinates:
$\mathbf{p}' = \mathbf{R}_z(\theta)\mathbf{p}$

Rotation matrix

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



Rotation about other axes?

Same argument for z rotation!

- Rotation about x axis ➜ x unchanged
- Rotation about y axis ➜ y unchanged

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

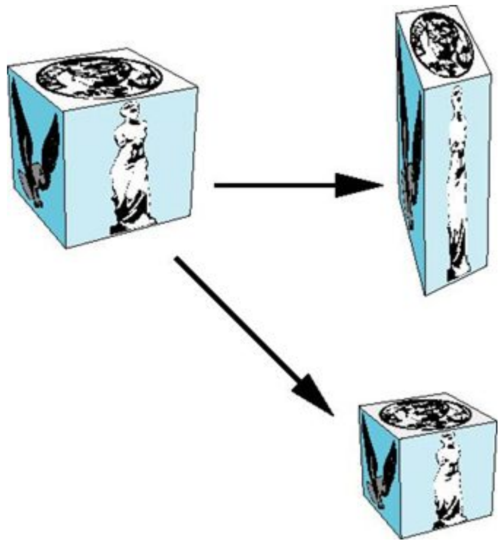
$$\mathbf{R} = \mathbf{R}_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Scaling

Expand/contract along each axis (fixed point of origin)

$x' = s_x x$
$y' = s_y y$
$z' = s_z z$

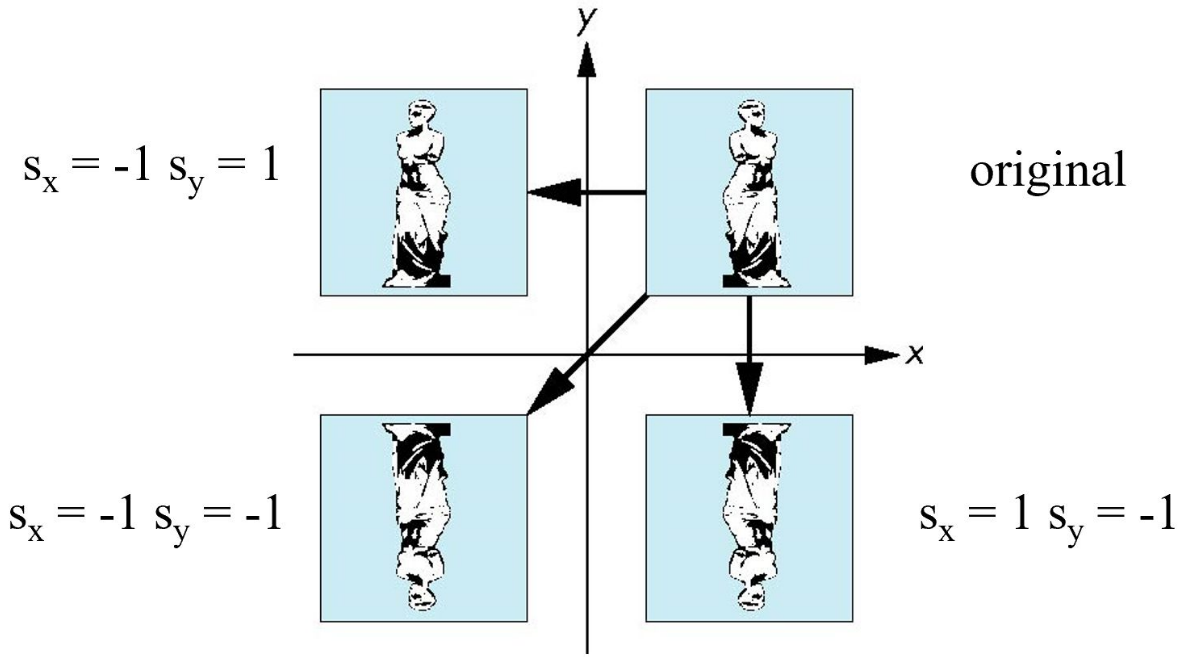
$$\mathbf{S} = \mathbf{S}(s_x, s_y, s_z) = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



Reflection

How can we accomplish this using scale?

- Negative scale factor!



Ok, so do we do this one at a time?

No ➞ concatenate!

- Combine *arbitrary* affine transform matrices by multiplying together the rotation/translation/scaling matrices

- **Same** transform applied to **many** vertices
  - Cost of forming matrix **M = ABCD** is not significant compared to calculating it one by one

- Hard part is **how** to form the transform based on your application!

Order of transformations

Matrix on the **right** applied first

The following are equivalent (mathematically):

- $\mathbf{p' = ABCp = A(B(Cp))}$

Many references use column matrices to represent points. For column matrices:

- $\mathbf{p}'^{T} = \mathbf{p}^{T}\mathbf{C}^{T}\mathbf{B}^{T}\mathbf{A}^{T}$

Ex

(From Mozilla -- https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API/Matrix_math_for_the_web)

Scale by 80%, move down by 200 pixels, rotate 90deg about origin

```
    xform = rotate * translate * scale
```

What happens when we reverse the order?

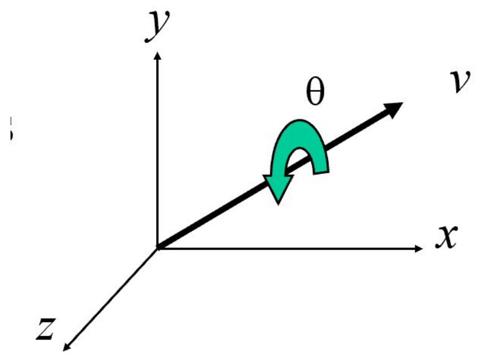
General rotation about origin

Rotation by θ about some *arbitrary axis* can be decomposed into concatenation of rotation about x, y, z axes

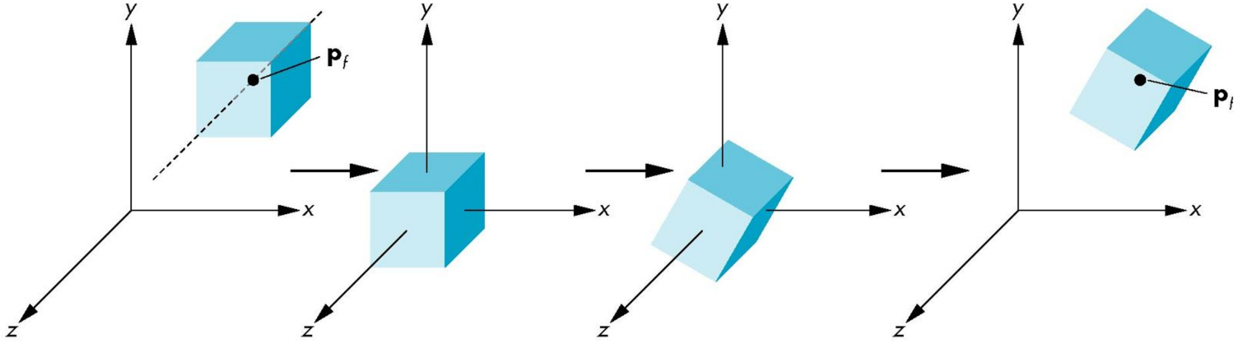
$\mathbf{R}(\theta) = \mathbf{R}_z(\theta_z)\mathbf{R}_y(\theta_y)\mathbf{R}_x(\theta_x)$

$\theta_x\ \theta_y\ \theta_z$ are called Euler angles

Rotations don't commute -- use rotations in another order but with different angles



How do we rotate about some point *other* than origin?



Move fixed point to origin
Rotate
Move back

$\mathbf{M} = \mathbf{T}(p_f)\mathbf{R}(\theta)\mathbf{T}(-p_f)$

But how do we that? We're using matrices!

Use geometry to handle inverse!

Translation: $\mathbf{T}^{-1}(d_x, d_y, d_z) = \mathbf{T}(-d_x, -d_y, -d_z)$

Rotation: $\mathbf{R}^{-1}(\theta) = \mathbf{R}(-\theta)$

- holds for any rotation matrix!
- Note that $\cos(-\theta) = \cos(\theta)$, $\sin(-\theta) = -\sin(\theta)$
  - $\mathbf{R}^{-1}(\theta) = \mathbf{R}^{T}(-\theta)$

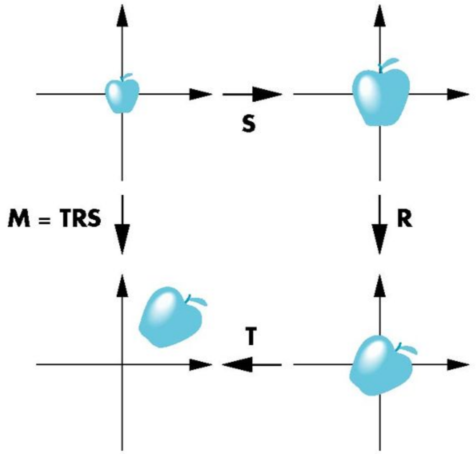
Scaling: $\mathbf{S}^{-1}(s_x, s_y, s_z) = \mathbf{S}(1/s_x, 1/s_y, 1/s_z)$

Instancing

Modeling objects often starts with a simple object **centered at origin**, **oriented with axis**, and at a **standard size**

Apply instance transformation to its vertices:

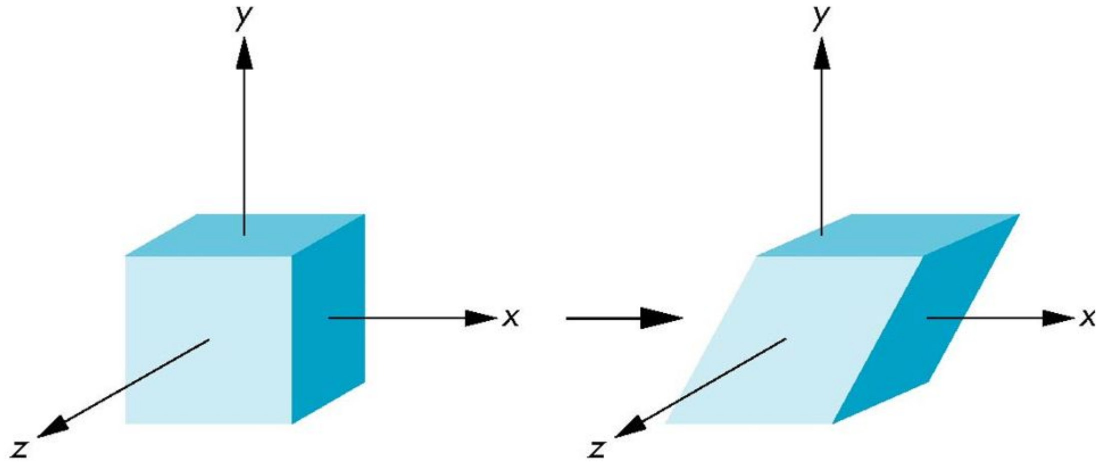
- Scale
- Orient
- Locate



Shear (transform)

Another basic transform

- Pulls faces in opposite directions

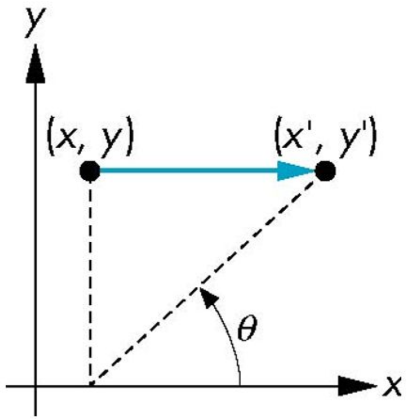


Shear matrix

Shear along x axis

x' = x + y cotθ
y' = y
z' = z

$$\mathbf{H}(\theta) = \begin{bmatrix} 1 & \cot\theta & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



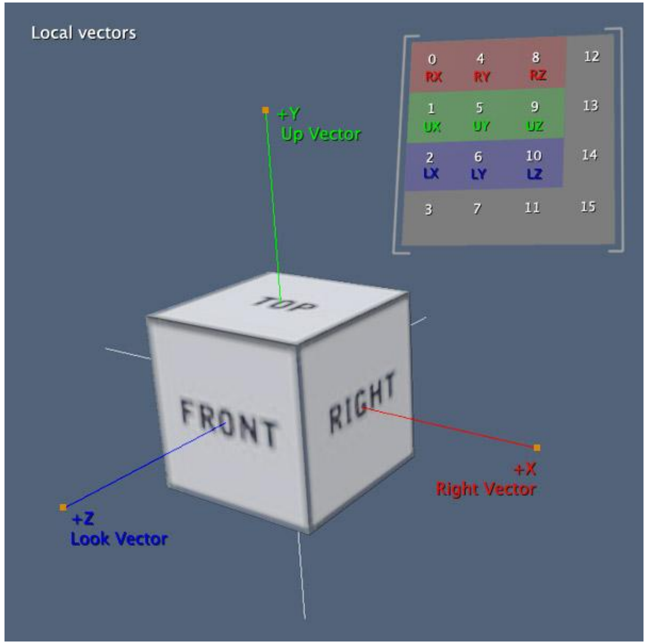
Now let's do it in WebGL!

A brief touch on history

OpenGL world, considered various matrix types

- `GL_MODELVIEW`
- `GL_PROJECTION`
- `GL_TEXTURE`
- `GL_COLOR`

Manipulate via `glMatrixMode(GL_MODELVIEW);`



Deprecation

This has been all deprecated

Why deprecate?

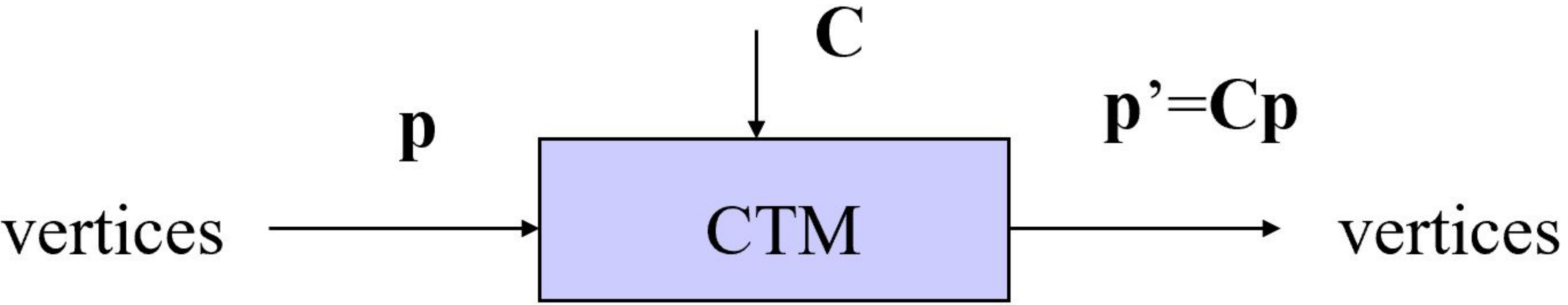
- Functions were part of fixed-CPU pipeline
    - Model-view/projection matrices applied using CPU
- Doesn't really help matters if we're going GPU!

Current transformation matrix ➜ current state of our matrix that *can* be applied to shaders

Current transformation matrix (CTM)

CTM ➜ 4x4 homogeneous coordinate matrix

- Part of state
- Applied to all vertices as they go down pipeline
- Defined in user program (application) and loaded



Operations

CTM altered via loading or post-multiplication

(Load) Identity matrix:	$\mathbf{C} \leftarrow \mathbf{I}$
(Load) Arbitrary matrix:	$\mathbf{C} \leftarrow \mathbf{M}$
(Load) Translation matrix:	$\mathbf{C} \leftarrow \mathbf{T}$
(Load) Arbitrary matrix:	$\mathbf{C} \leftarrow \mathbf{R}$
(Load) Scaling matrix:	$\mathbf{C} \leftarrow \mathbf{S}$
Postmultiply by arbitrary matrix:	$\mathbf{C} \leftarrow \mathbf{CM}$
Postmultiply by translation matrix:	$\mathbf{C} \leftarrow \mathbf{CT}$
Postmultiply by rotation matrix:	$\mathbf{C} \leftarrow \mathbf{CR}$
Postmultiply by scaling matrix:	$\mathbf{C} \leftarrow \mathbf{CS}$

Rotation about fixed point

1) Start with identity matrix: **$C \leftarrow I$**
2) Move fixed point to origin: **$C \leftarrow CT$**
3) Rotate: **$C \leftarrow CR$**
4) Move fixed point back: **$C \leftarrow CT^{-1}$**

Result: **$C = TRT^{-1}$**, **backwards**!

(Consequence of postmultiplications)

Reversing order

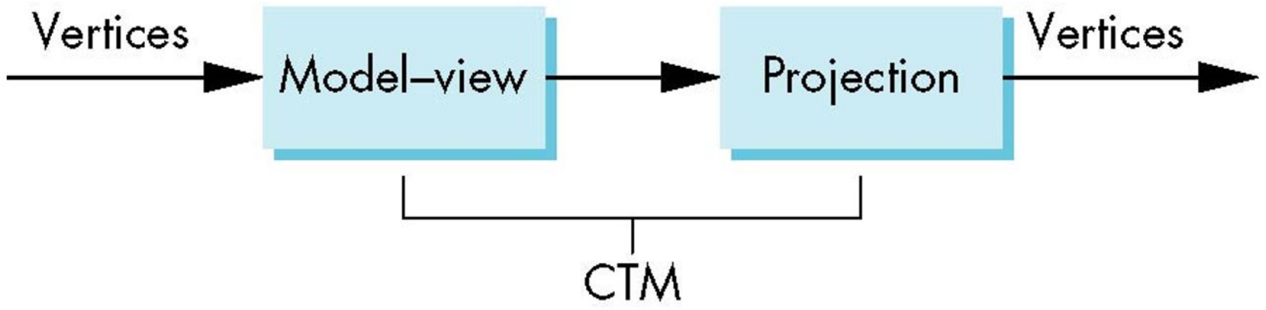
Want: $C = \mathbf{T}^{-1}\mathbf{R}\mathbf{T}$, so change order

1) $\mathbf{C} \leftarrow \mathbf{I}$
2) $\mathbf{C} \leftarrow \mathbf{C}\mathbf{T}^{-1}$
3) $\mathbf{C} \leftarrow \mathbf{C}\mathbf{R}$
4) $\mathbf{C} \leftarrow \mathbf{C}\mathbf{T}$

Each operation is a function call in program
Last operation specified is first executed!

CTM in WebGL

Emulate model-view/projection matrix in OpenGL pipeline



Model-view matrix

WebGL, model-view matrix used for:

- Camera positioning
    - Rotations/translations
    - Buuuuut, let's use the lookAt function in MV.js :)

- Object models

Project matrix:

- Define view volume and select camera lens

Rotation, translation, scaling (WebGL / Appl Code)

Identity matrix: `var m = mat4();`

Multiply (on right) by rotation matrix of theta in degrees
(`vx, vy, vz`) defines axis of rotation:

```
    var r = rotate(theta, vx, vy, vz);
    m = mult(m, r)
```

Similar with translation/scaling:

```
    var s = scale(sx, sy, sz);
    var t = translate(dx, dy, dz);
    m = mult(s, t);
```

EX

Rotate about z-axis by 30 degrees with fixed point of (1.0, 2.0, 3.0)

```
var m = mult(translate(1.0, 2.0, 3.0),
             rotate(30.0, 0.0, 0.0, 1.0));

m = mult(m, translate(-1.0, -2.0, -3.0));
```

Last matrix specified in program is first applied!!!

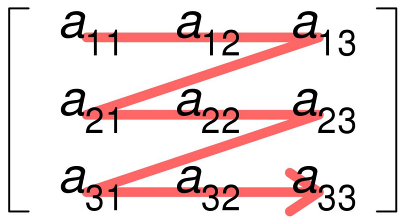
Arbitrary matrices

Load/multiply matrices defined in application

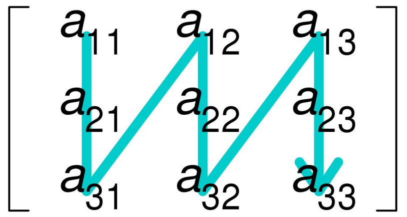
- Stored as 1-D array of 16
- Treated as 4x4 in row-major order
    - (OpenGL wants column-major)

- `gl.uniformMatrix4f` has parameter for automated transposition, set to **false**
- `flatten` converts to column-major order

Row-major order



Column-major order



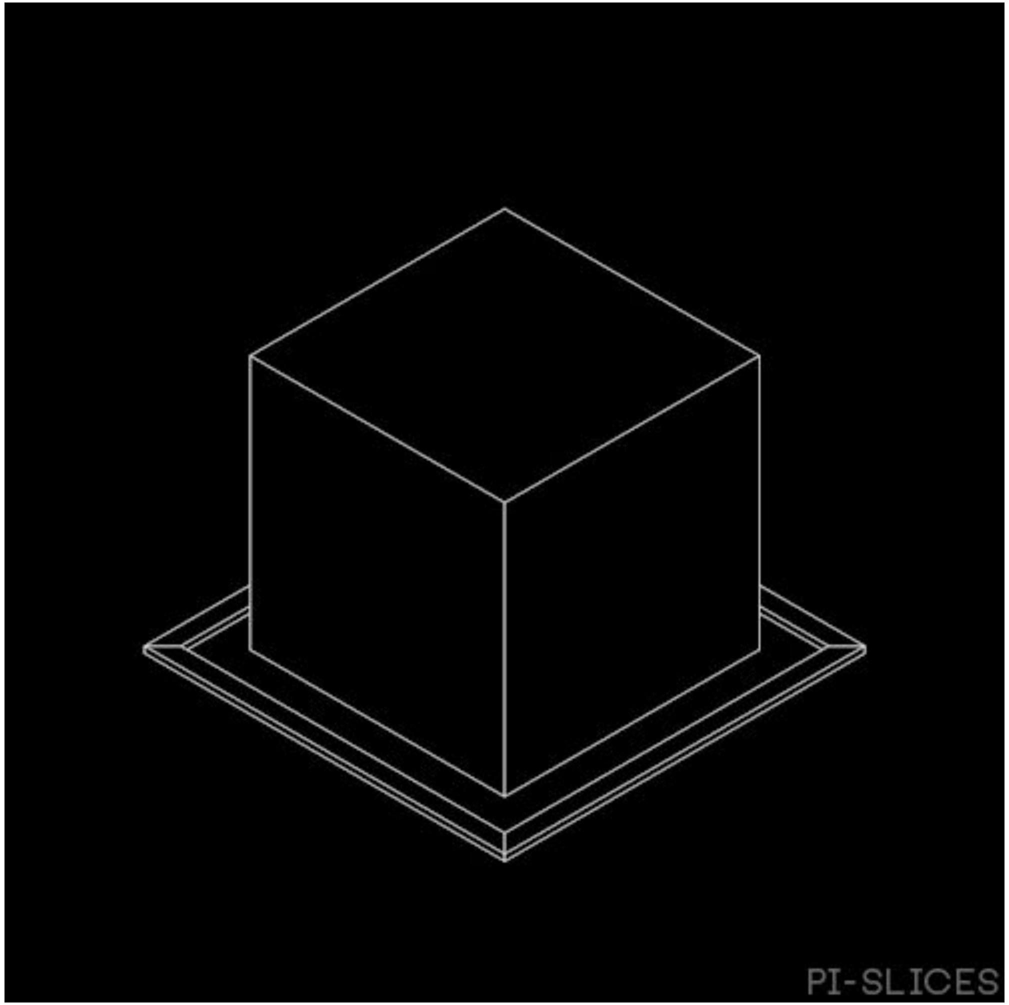
Matrix stacks

May want to save transform matrices

OpenGL used to use *stacks*

JavaScript will use *arrays/lists*

```
var stack = [];
stack.push(modelViewMatrix);
modelViewMatrix = stack.pop();
```


PI-SLICES

Now let's use these xforms to rotate a cube!

cube.html

Cube rotating on start, mouse/button listener changes direction

Where do we apply the transforms?

Similar to rotating square

Do we apply:

- In application to vertices?
- In vertex shader, send the MV matrix?
- In vertex shader, send angles?

Choice unclear

Trigonometry once in CPU or for each vertex in shader

- GPU has trigonometric functions hardcoded in silicon

Event listeners

```
document.getElementById( "xButton" ).onclick = function () {
  axis = xAxis;
};

document.getElementById( "yButton" ).onclick = function () {
  axis = yAxis;
};

document.getElementById( "zButton" ).onclick = function () {
  axis = zAxis;
};
```

render function

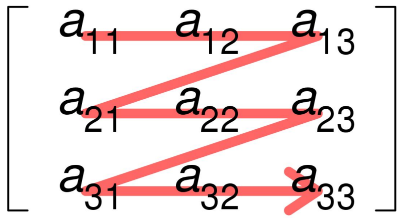
```
function render(){
    gl.clear( gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);
    theta[axis] += 2.0;
    gl.uniform3fv(thetaLoc, theta);
    gl.drawArrays( gl.TRIANGLES, 0, numVertices );
    requestAnimFrame( render );
}
```

Rotation shader

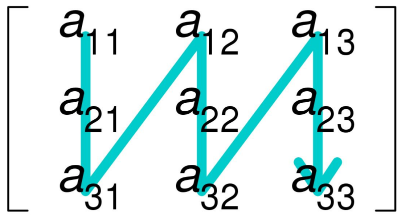
```
attribute vec4 vPosition;
attribute vec4 vColor;
varying vec4 fColor;
uniform vec3 theta;

void main() {
    vec3 angles = radians( theta );
    vec3 c = cos( angles );
    vec3 s = sin( angles );
    // Remember: these matrices are column-major
    mat4 rx = mat4( 1.0,  0.0,  0.0, 0.0,
                    0.0,  c.x,  s.x, 0.0,
                    0.0, -s.x,  c.x, 0.0,
                    0.0,  0.0,  0.0, 1.0 );
```

Row-major order


a_{11} a_{12} a_{13}
a_{21} a_{22} a_{23}
a_{31} a_{32} a_{33}

Column-major order


a_{11} a_{12} a_{13}
a_{21} a_{22} a_{23}
a_{31} a_{32} a_{33}

Rotation shader

```
 mat4 ry = mat4( c.y, 0.0, -s.y, 0.0,
                 0.0, 1.0,  0.0, 0.0,
                 s.y, 0.0,  c.y, 0.0,
                 0.0, 0.0,  0.0, 1.0 );

 mat4 rz = mat4( c.z, -s.z, 0.0, 0.0,
                 s.z,  c.z, 0.0, 0.0,
                 0.0,  0.0, 1.0, 0.0,
                 0.0,  0.0, 0.0, 1.0 );

  fColor = vColor;
  gl_Position = rz * ry * rx * vPosition;
}
```

Incremental rotation

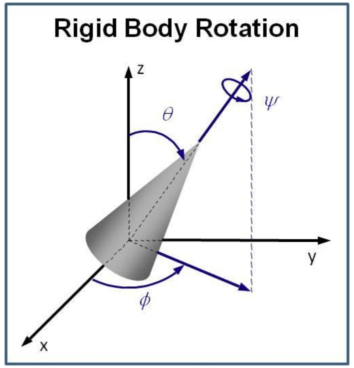
Consider two approaches:

- For a sequence of rotation matrices $\mathbf{R}_0, \mathbf{R}_1, \ldots, \mathbf{R}_n$, find Euler angles for each
  - Use $\mathbf{R}_i = \mathbf{R}_{iz}\mathbf{R}_{iy}\mathbf{R}_{iz}$
  - Not very efficient!

- Or, use final positions to determine axis/angle of rotation
  - Increment only angle

- However,
  - Quaternions can be more efficient!

WikiPedia:
The Euler angles are three angles introduced by Leonhard Euler to describe the orientation of a rigid body with respect to a fixed coordinate system.



Quaternion

Extension of imaginary numbers from 2 to 3 dimensions

Requires one real and three imaginary components (**i, j, k**)

$q = q_0 + q_1\mathbf{i} + q_2\mathbf{j} + q_3\mathbf{k}$

Express rotations smoothly and efficiently

- Model-view matrix ➡ quaternion
  - Carry out operations with quaternions
- Quaternion ➡ model-view matrix

Kind of fun to read engineers argue online:

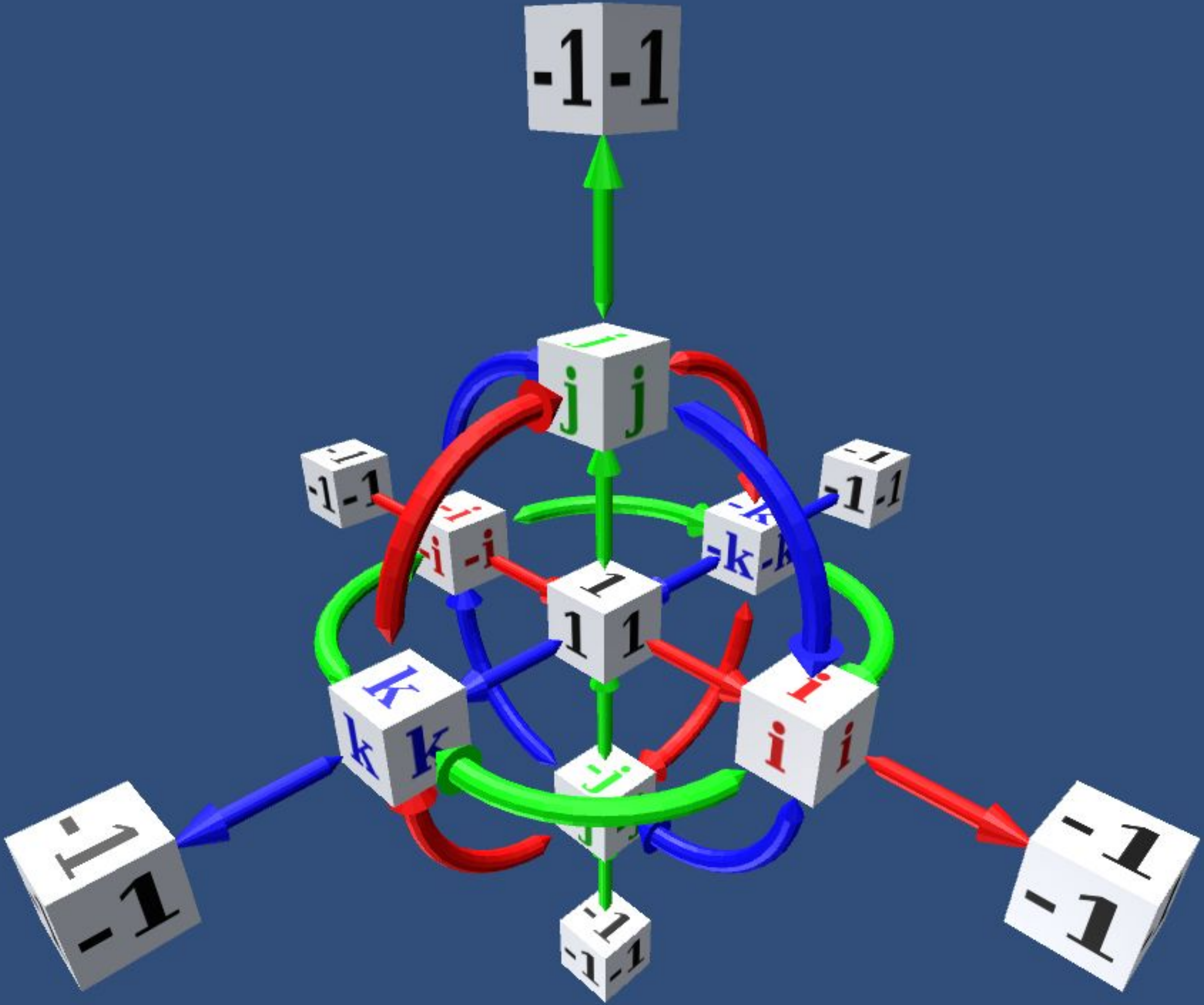
https://stackoverflow.com/questions/832805/euler-angles-vs-quaternions-problems-caused-by-the-tension-between-internal-s

Ex:

Multiply by k, rotate around 1, k, -1, -k axis/circle

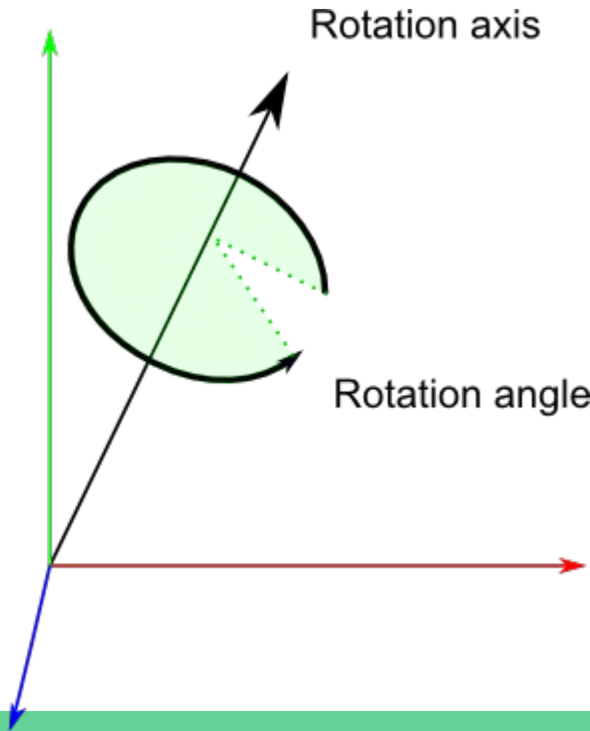
(consider a right-hand rule)

(thumb follows k-line, fingers rotate around – follow the blue!)



http://www.opengl-tutorial.org/intermediate-tutorials/tutorial-17-quaternions/#quaternions

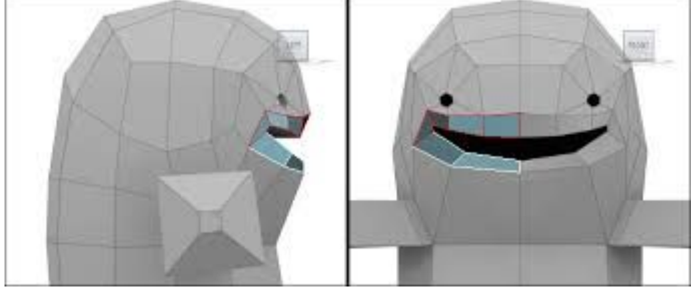
```
// RotationAngle is in radians
x = RotationAxis.x * sin(RotationAngle / 2)
y = RotationAxis.y * sin(RotationAngle / 2)
z = RotationAxis.z * sin(RotationAngle / 2)
w = cos(RotationAngle / 2)
```



https://www.youtube.com/watch?v=3BR8tK-LuB0

https://www.reddit.com/r/math/comments/42yc0i/visualizing_quaternions/

IC



Describe, in pseudo code, how you would *intelligently* track vertices/edges rather than managing each vertex directly.