CS174A Lecture 7

Announcements & Reminders

- *10/16/22: A2 due; will be discussed during this week's TA session*
- *10/26/22 and 10/27/22: Office hours, Noon – 1 PM PST, Zoom*
- *10/27/22: Midterm Exam: 6:00 – 7:30 PM PST, in person, in class*
- *11/08/22: Team project proposals due, initial version*
- *11/09/22: A3 due*
- *11/10/22: Midway demo, online zoom*

Last Lecture Recap

- ***Examples of Transformations:***
    - Translation, scaling, rotation, shear

Next Up

- ***Concatenation of Transformations (contd.)***
- ***Spaces:***
  - Model space
  - Object/world space
  - Eye/camera space
  - Screen space
- ***Projections: parallel and perspective***
- ***Midterm***
- ***Lighting, Flat/Smooth Shading***

SIGGRAPH trailers from 2013

Going backwards,

https://www.youtube.com/watch?v=FUGVF_eMeo4

And

https://www.youtube.com/watch?v=JAFhkdGtHck

Composite 3D Rotation About the Origin

$$\mathbf{R}(\theta_1, \theta_2, \theta_3) = \mathbf{R}_z(\theta_3)\mathbf{R}_y(\theta_2)\mathbf{R}_x(\theta_1)$$

- ***This is known as the "Euler angle" representation of 3D rotations***
- ***The order of the rotation matrices is important !!***
- ***Note: The Euler angle representation suffers from singularities***

Gimbal Lock

$$\mathbf{R}(\theta_1,\theta_2,\theta_3) = \mathbf{R}_z(\theta_3)\mathbf{R}_y(\theta_2)\mathbf{R}_x(\theta_1)$$

$$= \begin{bmatrix} \cos\theta_3 & -\sin\theta_3 & 0 & 0 \\ \sin\theta_3 & \cos\theta_3 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\theta_2 & 0 & \sin\theta_2 & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta_2 & 0 & \cos\theta_2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta_1 & -\sin\theta_1 & 0 \\ 0 & \sin\theta_1 & \cos\theta_1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

What happens when the middle angle is 90˚?

$$\mathbf{R}(\theta_1,90^\circ,\theta_3) = \mathbf{R}_z(\theta_3)\mathbf{R}_y(90^\circ)\mathbf{R}_x(\theta_1)$$

$$= \begin{bmatrix} \cos\theta_3 & -\sin\theta_3 & 0 & 0 \\ \sin\theta_3 & \cos\theta_3 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta_1 & -\sin\theta_1 & 0 \\ 0 & \sin\theta_1 & \cos\theta_1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} \cos\theta_3 & -\sin\theta_3 & 0 & 0 \\ \sin\theta_3 & \cos\theta_3 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & \sin\theta_1 & \cos\theta_1 & 0 \\ 0 & \cos\theta_1 & -\sin\theta_1 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} 0 & \cos\theta_3\sin\theta_1 - \sin\theta_3\cos\theta_1 & \cos\theta_3\cos\theta_1 + \sin\theta_3\sin\theta_1 & 0 \\ 0 & \cos\theta_3\cos\theta_1 + \sin\theta_3\sin\theta_1 & -\cos\theta_3\sin\theta_1 + \sin\theta_3\cos\theta_1 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Loss of a Rotational Degree of Freedom

$$\mathbf{R}(\theta_1, 90^\circ, \theta_3) = \begin{bmatrix} 0 & \cos\theta_3 \sin\theta_1 - \sin\theta_3 \cos\theta_1 & \cos\theta_3 \cos\theta_1 + \sin\theta_3 \sin\theta_1 & 0 \\ 0 & \cos\theta_3 \cos\theta_1 + \sin\theta_3 \sin\theta_1 & -\cos\theta_3 \sin\theta_1 + \sin\theta_3 \cos\theta_1 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} 0 & \sin(\theta_1 - \theta_3) & \cos(\theta_1 - \theta_3) & 0 \\ 0 & \cos(\theta_1 - \theta_3) & -\sin(\theta_1 - \theta_3) & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

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

where $\theta = \theta_1 - \theta_3$

Thus, the two remaining rotational degrees of freedom, θ_1 and θ_3, have collapsed into a single rotational degree of freedom θ, which is the difference of the two rotational angles

There are Alternatives

It is often convenient to use other representations of 3D rotations that do not suffer from Gimbal Lock

- Advanced concepts
  - *Quaternions*
  - *Exponential Maps*

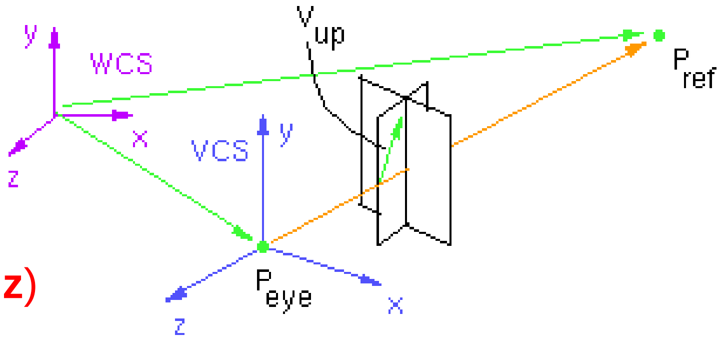
“LookAt” Matrices

Defining M_{cam}

Given:

Eye point P_{eye}
Reference point P_{ref}
Up vector $\mathbf{v}_{up}$

($\mathbf{v}_{up}$ is not necessarily orthogonal to $\mathbf{z}$)



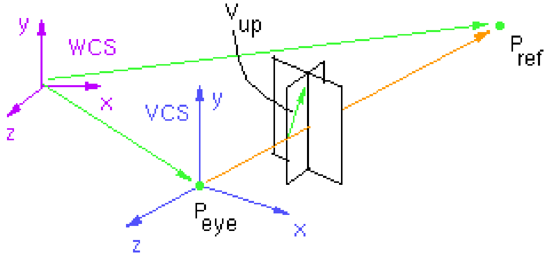
To build $\mathbf{M}_{cam}$ we need to define a camera coordinate system [$\mathbf{i}$ $\mathbf{j}$ $\mathbf{k}$ O]

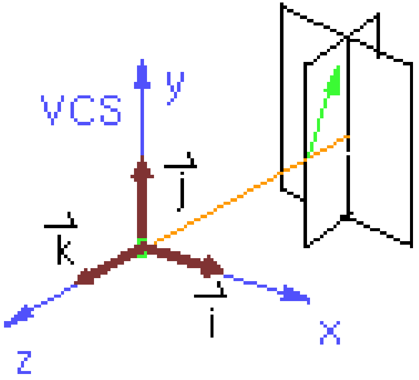
Camera Coordinate System

$$\mathbf{k} = \frac{P_{\text{eye}} - P_{\text{ref}}}{|P_{\text{eye}} - P_{\text{ref}}|}$$

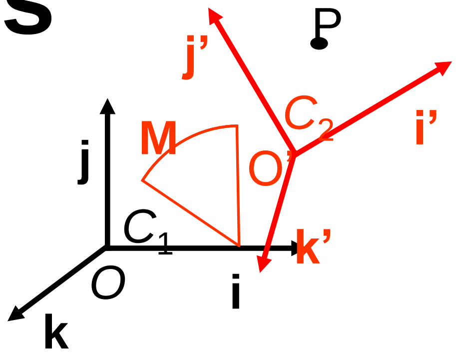
$$\mathbf{i} = \frac{\mathbf{v}_{\text{up}} \times \mathbf{k}}{|\mathbf{v}_{\text{up}} \times \mathbf{k}|}$$

$$\mathbf{j} = \mathbf{k} \times \mathbf{i}$$





Reminder: Change of Basis



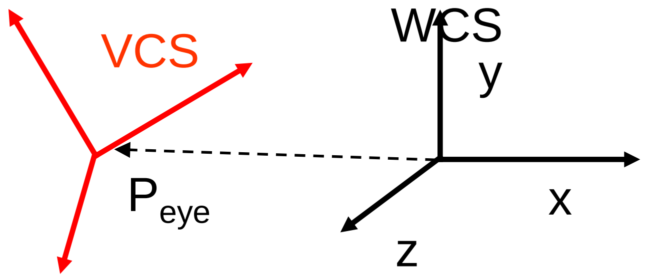
$$P_{C_1} = \mathrm{M} P_{C_2}$$

$$P_{C_1} = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} i'_x & j'_x & k'_x & O'_x \\ i'_y & j'_y & k'_y & O'_y \\ i'_z & j'_z & k'_z & O'_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \mathrm{M} P_{C_2}$$

Building M_{cam}

Change of basis

Our reference system is WCS,
we know the camera parameters with respect to the world



Align WCS with VCS

Translation

Rotation

$$\mathrm{M}_{\mathrm{cam}} = \begin{bmatrix} 1 & 0 & 0 & P_{\mathrm{eye}_x} \\ 0 & 1 & 0 & P_{\mathrm{eye}_y} \\ 0 & 0 & 1 & P_{\mathrm{eye}_z} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} i_x & j_x & k_x & 0 \\ i_y & j_y & k_y & 0 \\ i_z & j_z & k_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$P_{\mathrm{WCS}} = \mathrm{M}_{\mathrm{cam}} P_{\mathrm{VCS}}$$

Building M_{cam} Inverse

Invert the smart way

$$\mathbf{M}_{\text{cam}}^{-1} = \left(\begin{bmatrix} 1 & 0 & 0 & P\text{eye}_x \\ 0 & 1 & 0 & P\text{eye}_y \\ 0 & 0 & 1 & P\text{eye}_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} i_x & j_x & k_x & 0 \\ i_y & j_y & k_y & 0 \\ i_z & j_z & k_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \right)^{-1}$$

$$= \begin{bmatrix} i_x & j_x & k_x & 0 \\ i_y & j_y & k_y & 0 \\ i_z & j_z & k_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}^{-1} \begin{bmatrix} 1 & 0 & 0 & P\text{eye}_x \\ 0 & 1 & 0 & P\text{eye}_y \\ 0 & 0 & 1 & P\text{eye}_z \\ 0 & 0 & 0 & 1 \end{bmatrix}^{-1}$$

Building M_{cam} Inverse

Invert the smart way

$$\mathrm{M}_{\text{cam}}^{-1} = \begin{bmatrix} i_x & j_x & k_x & 0 \\ i_y & j_y & k_y & 0 \\ i_z & j_z & k_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}^{-1} \begin{bmatrix} 1 & 0 & 0 & P_{\text{eye}_x} \\ 0 & 1 & 0 & P_{\text{eye}_y} \\ 0 & 0 & 1 & P_{\text{eye}_z} \\ 0 & 0 & 0 & 1 \end{bmatrix}^{-1}$$

Transpose

Negate

$$= \begin{bmatrix} \boxed{\begin{matrix} i_x & i_y & i_z \\ j_x & j_y & j_z \\ k_x & k_y & k_z \end{matrix}} & \begin{matrix} 0 \\ 0 \\ 0 \end{matrix} \\ \begin{matrix} 0 & 0 & 0 \end{matrix} & 1 \end{bmatrix} \begin{bmatrix} \begin{matrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{matrix} & \boxed{\begin{matrix} -P_{\text{eye}_x} \\ -P_{\text{eye}_y} \\ -P_{\text{eye}_z} \end{matrix}} \\ \begin{matrix} 0 & 0 & 0 \end{matrix} & 1 \end{bmatrix}$$

$$P_{\text{VCS}} = \mathrm{M}_{\text{cam}}^{-1} P_{\text{WCS}}$$

How to call `look_at()`

```
// Pass in eye position, at
// position, and up vector.

        Mat4.look_at( Vec.of( 0,0,0 ), Vec.of( 0,0,1 ), Vec.of( 0,1,0 ) ) );

// Or:

    Mat4.look_at( ...Vec.cast( [0,0,0], [0,0,1], [0,1,0] ) );
```

Positioning camera without look_at()

- Not as easy to point directly at things, but valid.
- Generate it using `mult()/rotation()/translation()/scale()` instead of `look_at()`
- Remember `inverse()` concepts apply to cameras
  - Any incremental modifications you make will encounter properties of inverted products (reverse the order ***and*** invert each part)

Summary of the Modelview Transformation

1. ***An affine transformation composed of elementary affine transformations***
2. ***The camera transformation is a change of basis***
3. ***The modelview transformation preserves:***

- lines and planes
- parallelism of lines and planes
- affine combinations of points and relative ratios

Normals in Graphics

Normals

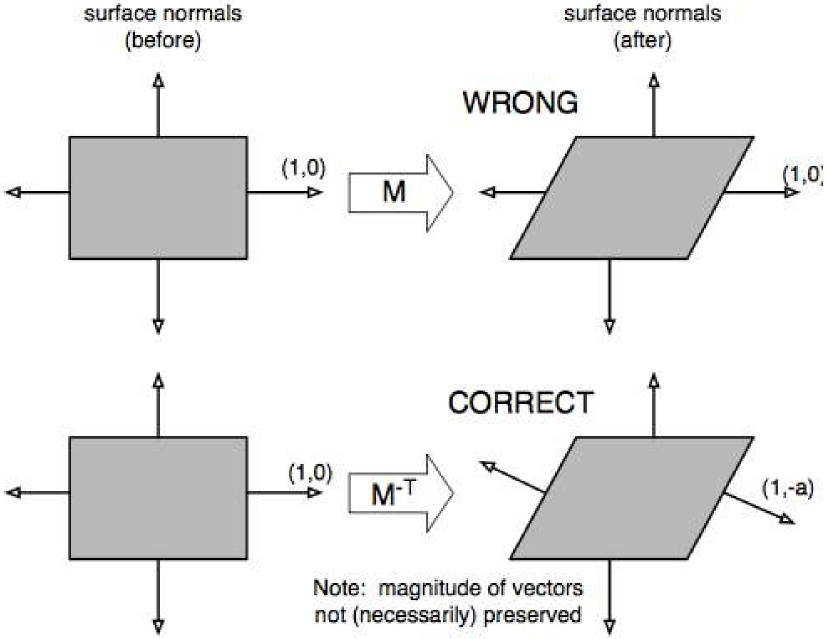
- Mathematics:
  - A vector that is perpendicular to the surface at a given point
  - Point “outward” or “away” from the object
- True realism:
  - Would require calculation of normal/derivative at every point along a continuous shape
  - Not feasible
- Graphics:
  - We’re only interested in vertex normals
  - Discrete “approximations” of the normal sampled at points on the imaginary surface

Transforming Normals

Normal vectors are transformed along with vertices and polygons.

- How do you transform a normal ?
- What about unit magnitude ?

Consider the shear matrix: $$M = \begin{bmatrix} 1 & a \\ 0 & 1 \end{bmatrix} \quad M^{-T} = \begin{bmatrix} 1 & 0 \\ -a & 1 \end{bmatrix}$$



The good thing is, this problem does not happen with tangent vectors!!!

Mathematical Reason for Inverse Transpose:

All we know about the transformed normal is that the dot product with tangent (V) must equal zero:

$$N^T V = 0$$

$$N^T M^{-1} M V = 0$$

$$\underbrace{\left(M^{-T} N\right)^T}_{N'^T} \underbrace{(MV)}_{V'} = 0$$

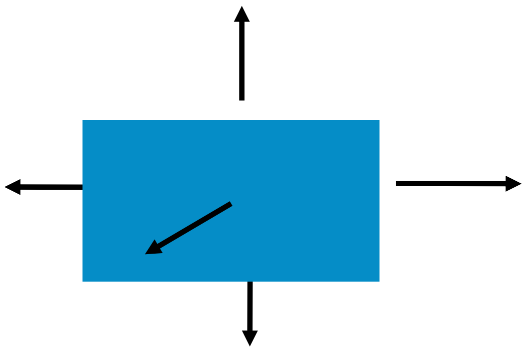
Polygon Attributes

Per vertex

- Position
- Texture coordinates

Per vertex or per face (if flat shading)

- Color
- Normal



Reminder: What are normals for?

- Lighting!
- The direction of the normal determines how the light will bounce off each surface when modeling light rays.

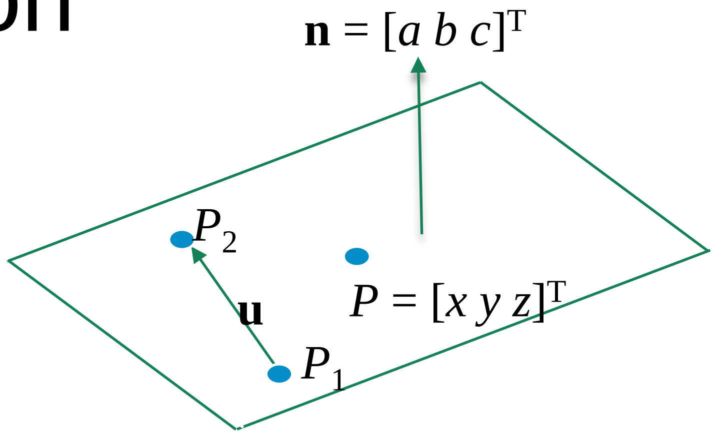
Our shapes so far have easy normal vectors.

- “Z axis” vector is perpendicular to Triangle and Square
- For a cube, normals would also just be axis-aligned
- For a sphere, we know analytically that the vector pointing away from the center (perpendicular to the formula’s surface) will be the normal.
  - Just assign normal = position coord.

What do you do when the normals aren’t known?

- Hint: Think per-triangle.

Plane Equation



Normal / point form

$$F(x, y, z) = ax + by + cz + d = \mathbf{n} \bullet P + d$$

For points on plane, $F(x, y, z) = 0$

Observation: Let's take an arbitrary vector $\mathbf{u}$ that lies on the plane which can be defined by two points; e.g., P_1, P_2 on the plane.

$$\mathbf{u} = P_2 - P_1$$

$$\left.\begin{aligned} \mathbf{n} \bullet P_1 + d &= 0 \\ \mathbf{n} \bullet P_2 + d &= 0 \end{aligned}\right\} \Rightarrow \mathbf{n} \bullet (P_2 - P_1) = 0 \Rightarrow \mathbf{n} \bullet \mathbf{u} = 0 \Rightarrow \boxed{\mathbf{n} \perp \mathbf{u}}$$

Computing Normal / Point Form From 3 Points

$F(x, y, z) = ax + by + cz + d = \mathbf{n} \bullet P + d$

Points on Plane $F(x, y, z) = 0$

First way (4 equations in unknowns a, b, c, d):

$\mathbf{n} \bullet P_0 + d = 0$

$\mathbf{n} \bullet P_1 + d = 0$

$\mathbf{n} \bullet P_2 + d = 0$

$|\mathbf{n}| = 1$ (arbitrary choice)

Second way:

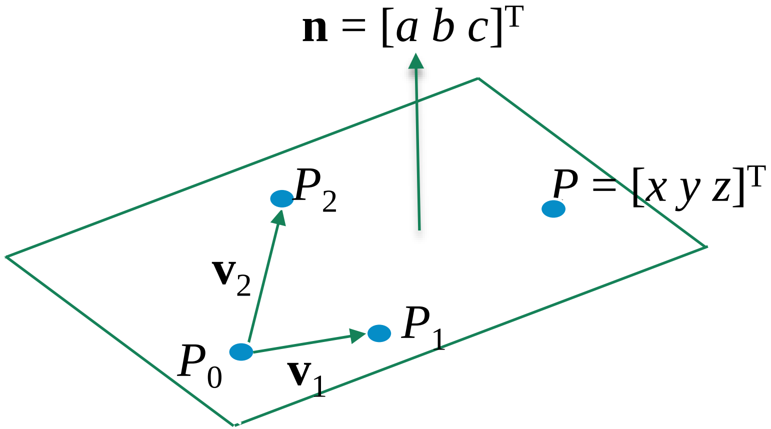
$\mathbf{n}$ is normal to the plane

Let's find a normal vector:

$\mathbf{n} = (P_1 - P_0) \times (P_2 - P_0) = \mathbf{v}_1 \times \mathbf{v}_2$

Compute d :

$d = -\mathbf{n} \bullet P_0$



When the normals aren’t known:

- Using the indices, collect the positions of the three points of the triangle.
- Create two vectors out of the triangle.
- Use a cross product.

Cross Product Normals

- The result might point inside the shape instead of out!
  - (AxB = -BxA)
  - Hard to know which edges to make “A” and “B”
- How to detect an inward vector? Assume shape is convex and centered at the origin.