Lecture 8

SIGGRAPH trailer from 2011

https://www.youtube.com/watch?v=JK9EEE3RsKM

Announcements

- Assignment 3 is still released!
- Due Date: Sunday night (midnight)
  - How are we doing?
- Assignment 4 shaders

Scene Timing & Organizing Movement

Using unit conversion to get precise speeds

- Sometimes you want to use real-world measurements in your virtual world.

  10 feet, 10 feet per second, 10 RPM, etc.

- This allows all your calculations to be in real units, which helps you check that your results make sense.

- In a simulation, you might have other units like mass and force; keeping all measurements in real-world units lets you check your results using physics formulas, and re-use constants from physics.

Using unit conversion to get precise speeds

Problem 1: Rotate or move box back and forth **once per second**.

Given: t

- For our angle we need: f(t) that goes back and forth one per second.
- t is in units of seconds
- Trig functions like sine repeat once every 2 * PI units.

So we have: **f(t) = sin(g(t))**

Using unit conversion to get precise speeds

Problem 1: Rotate or move box back and forth **once per second**.

So we have: **f(t) = sin(g(t))**

- For **g(t)** to equal units of “one sine period”, we need:

  g(t) = a * t where **a** is equal to one unit of: $\frac{2 * PI}{second}$

- Solution: a = 2 * PI
- In its units, “seconds” will cancel out when multiplying by t.
- Final result:

  f(t) = sin(g(t)) = sin(2 * PI * t)

Harder example

Making speeds precise

Using unit conversion to get precise speeds

Suppose you want a cube to rotate around at precisely 10 RPM.

That's 10 units of $\frac{revolutions}{minute}$.

You'd like to change this, through unit conversion, to either:

1. Radians (if you make your rotation a pure function of time calculated fresh each step)

or 2. Radians per frame (if you make it an incremental function of the previous frame's rotation)

Either works.

Using unit conversion to get precise speeds

We know that there are 2*PI radians per revolution, so that gets rid of the revolutions unit and introduces the desired radians unit, but still leaves minutes in the result to deal with.

$$\frac{10 * 2 * PI}{minute}$$

We're about to do some unit cancelling magic to get us to the units we want, and a few more ingredients are needed for the setup.

Using unit conversion to get precise speeds

Let's suppose we don't have `t` yet. From your `display()` function we need the value `program state.animation time`, the number of milliseconds that have passed since the program started.

We know there are 1000 milliseconds in a second and 60 seconds in a minute. Lastly we want animation to progress at a 1:1 rate with real time, so that gets us from program time units into animation time units.

Using unit conversion to get precise speeds

We want $1\frac{runtime_{(program)}}{rate_{(animation)}}$

- As opposed to animating at faster or slower than the proram's clock
- Getting this is not always a given
  - You can mess up
  - Constant incremental motion every frame = jittering as some frames run faster than others
  - Luckily part 1 of this example doesn't use incremental motion

Using unit conversion to get precise speeds

The final calculation is this:

$$radians = 10\frac{revolutions}{minute} * 2PI\frac{radians}{revolution} * \frac{1}{60}\frac{minute}{seconds}\ldots$$

$$*\frac{1}{1000}\frac{second}{milliseconds} * 1\frac{runtime_{(program)}}{rate_{(animation)}} * (\mathbf{time})\frac{milliseconds}{runtime_{(program)}}$$

Notice how all those unwanted units cancel out, at the end leaving only degrees at your animation rate.

$$\Rightarrow$$

$$radians = \frac{10*2*PI}{60*1000} * \mathbf{time}$$

Using unit conversion to get precise speeds

- Pass this many radians into a call to `rotate()`,
- Use the returned rotation matrix directly as your model matrix
- Now you have a rotating function with the relationship with real time that you want: **10 RPM**.

Using unit conversion to get precise speeds

Method #2: Radians per frame

- Doing incremental adjustments to the model matrix
- Instead of overwriting it with a function's result every frame
- Now you need a delta, stored in a different variable: `program_state.animation_delta_time`.

- Measures the time since the last frame (time per frame).
- Subtracting "previous time" from "time" introduces a unit of $\frac{milliseconds}{frame}$
- The frames unit immediately cancels out from the denominator by way of applying that adjustment over every frame.
- Another way to the same answer, giving you precisely 10 RPM again.

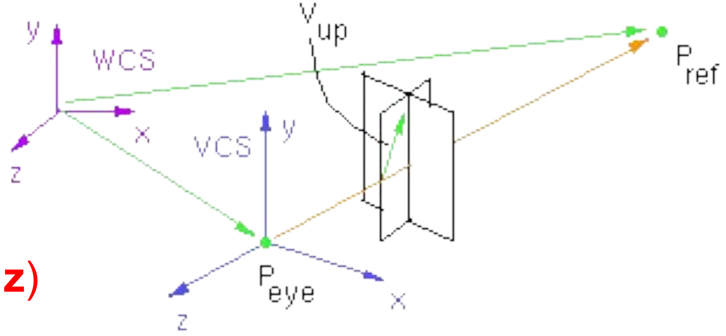
“LookAt” Matrices

Defining $\mathbf{M}_{cam}$

Given:

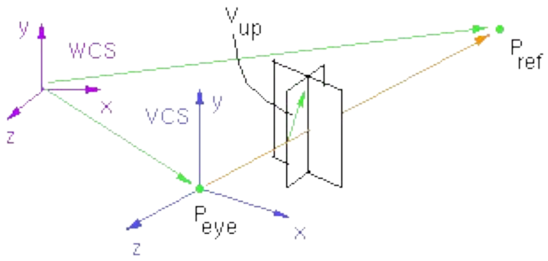
Eye point P_{eye}
Reference point P_{ref}
Up vector $\mathbf{v}_{up}$

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



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

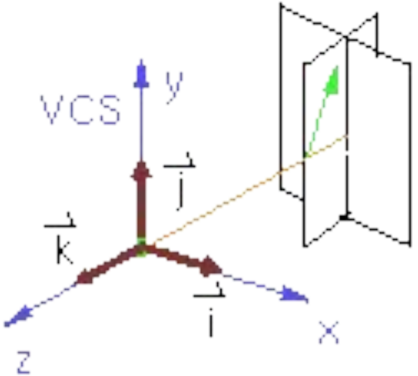
Camera Coordinate System



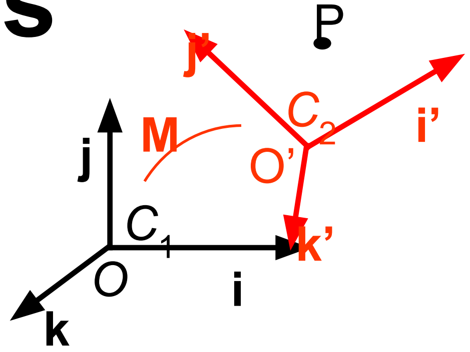
$$\mathbf{k} = \frac{P_{\text{eye}} - P_{\text{ref}}}{|P_{\text{eye}} - P_{\text{ref}}|}$$

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

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



Reminder: Change of Basis



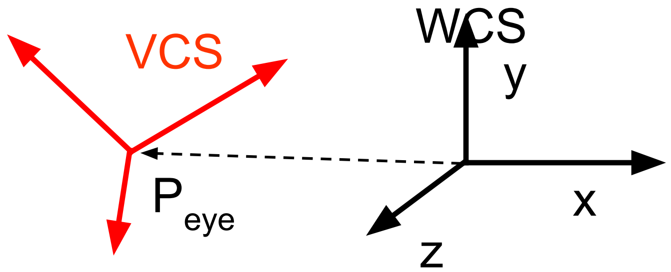
$$P_{C_1} = \mathrm{M} P_{C_2}$$

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

Building $\mathbf{M}_{\text{cam}}$

Change of basis

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



Align WCS with VCS

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

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

Building $\mathrm{M}_{\mathrm{cam}}$ 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

Projections

Transform Process

$$\underbrace{\begin{bmatrix} ? & 0 & 0 & ? \\ 0 & ? & 0 & ? \\ 0 & 0 & ? & ? \\ 0 & 0 & 0 & 1 \end{bmatrix}}_{Viewport} * \underbrace{\begin{bmatrix} ? & 0 & ? & ? \\ 0 & ? & ? & ? \\ 0 & 0 & ? & ? \\ 0 & 0 & ? & ? \end{bmatrix}}_{Projection} * \underbrace{\begin{bmatrix} ? & ? & ? & ? \\ ? & ? & ? & ? \\ ? & ? & ? & ? \\ 0 & 0 & 0 & 1 \end{bmatrix}}_{Camera} * \underbrace{\begin{bmatrix} ? & ? & ? & ? \\ ? & ? & ? & ? \\ ? & ? & ? & ? \\ 0 & 0 & 0 & 1 \end{bmatrix}}_{Model} * \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

- Next, what is the projection matrix?

Transform Process

$$\underset{Viewport}{\begin{bmatrix} ? & 0 & 0 & ? \\ 0 & ? & 0 & ? \\ 0 & 0 & ? & ? \\ 0 & 0 & 0 & 1 \end{bmatrix}} * \underset{Projection}{\begin{bmatrix} ? & 0 & ? & ? \\ 0 & ? & ? & ? \\ 0 & 0 & ? & ? \\ 0 & 0 & ? & ? \end{bmatrix}} * \underset{Camera}{\begin{bmatrix} ? & ? & ? & ? \\ ? & ? & ? & ? \\ ? & ? & ? & ? \\ 0 & 0 & 0 & 1 \end{bmatrix}} * \underset{Model}{\begin{bmatrix} ? & ? & ? & ? \\ ? & ? & ? & ? \\ ? & ? & ? & ? \\ 0 & 0 & 0 & 1 \end{bmatrix}} * \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

- The projection matrix is something <u>you</u> make, using special calls.
- Two built in functions make two kinds of them:
  - `Mat4::perspective()` causes converging lines / vanishing points.
  - `Mat4::orthographic()` causes parallel lines to remain parallel -- like how scenes look when viewed from far enough away.

Transform Process

$$\underset{Viewport}{\begin{bmatrix} ? & 0 & 0 & ? \\ 0 & ? & 0 & ? \\ 0 & 0 & ? & ? \\ 0 & 0 & 0 & 1 \end{bmatrix}} * \underset{Projection}{\begin{bmatrix} ? & 0 & ? & ? \\ 0 & ? & ? & ? \\ 0 & 0 & ? & ? \\ 0 & 0 & ? & ? \end{bmatrix}} * \underset{Camera}{\begin{bmatrix} ? & ? & ? & ? \\ ? & ? & ? & ? \\ ? & ? & ? & ? \\ 0 & 0 & 0 & 1 \end{bmatrix}} * \underset{Model}{\begin{bmatrix} ? & ? & ? & ? \\ ? & ? & ? & ? \\ ? & ? & ? & ? \\ 0 & 0 & 0 & 1 \end{bmatrix}} * \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

- The projection matrix is something <u>you</u> make, using special calls.
  - `perspective()` : The camera is like a point, and will see everything that falls within a **truncated pyramid (<u>frustum</u>)** expanding out from it

Transform Process

$$\underset{Viewport}{\begin{bmatrix} ? & 0 & 0 & ? \\ 0 & ? & 0 & ? \\ 0 & 0 & ? & ? \\ 0 & 0 & 0 & 1 \end{bmatrix}} * \underset{Projection}{\begin{bmatrix} ? & 0 & ? & ? \\ 0 & ? & ? & ? \\ 0 & 0 & ? & ? \\ 0 & 0 & ? & ? \end{bmatrix}} * \underset{Camera}{\begin{bmatrix} ? & ? & ? & ? \\ ? & ? & ? & ? \\ ? & ? & ? & ? \\ 0 & 0 & 0 & 1 \end{bmatrix}} * \underset{Model}{\begin{bmatrix} ? & ? & ? & ? \\ ? & ? & ? & ? \\ ? & ? & ? & ? \\ 0 & 0 & 0 & 1 \end{bmatrix}} * \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

- The projection matrix is something <u>you</u> make, using special calls.
  - `orthographic()`: The camera is like a flat rectangular screen, and will see everything that falls within a rectangular box in front of it.
  - Rectangular boxes are a special case of frustums

Transform Process

$$\underset{Viewport}{\begin{bmatrix} ? & 0 & 0 & ? \\ 0 & ? & 0 & ? \\ 0 & 0 & ? & ? \\ 0 & 0 & 0 & 1 \end{bmatrix}} * \underset{Projection}{\begin{bmatrix} ? & 0 & ? & ? \\ 0 & ? & ? & ? \\ 0 & 0 & ? & ? \\ 0 & 0 & ? & ? \end{bmatrix}} * \underset{Camera}{\begin{bmatrix} ? & ? & ? & ? \\ ? & ? & ? & ? \\ ? & ? & ? & ? \\ 0 & 0 & 0 & 1 \end{bmatrix}} * \underset{Model}{\begin{bmatrix} ? & ? & ? & ? \\ ? & ? & ? & ? \\ ? & ? & ? & ? \\ 0 & 0 & 0 & 1 \end{bmatrix}} * \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

- Both types, perspective() and orthographic(), are projections.
- Projections are different from the camera matrix. It kind of shapes the virtual camera lens, instead of placing, sizing, and pointing the virtual camera.

Transform Process

$$\underset{Viewport}{\begin{bmatrix} ? & 0 & 0 & ? \\ 0 & ? & 0 & ? \\ 0 & 0 & ? & ? \\ 0 & 0 & 0 & 1 \end{bmatrix}} * \underset{Projection}{\begin{bmatrix} ? & 0 & ? & ? \\ 0 & ? & ? & ? \\ 0 & 0 & ? & ? \\ 0 & 0 & ? & ? \end{bmatrix}} * \underset{Camera}{\begin{bmatrix} ? & ? & ? & ? \\ ? & ? & ? & ? \\ ? & ? & ? & ? \\ 0 & 0 & 0 & 1 \end{bmatrix}} * \underset{Model}{\begin{bmatrix} ? & ? & ? & ? \\ ? & ? & ? & ? \\ ? & ? & ? & ? \\ 0 & 0 & 0 & 1 \end{bmatrix}} * \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

- In your code, the projection and camera are both stored in the `Program_State` object
  - One per WebGL canvas, passed into your `display()`
- These aren't multiplied together until the shader program
  - Shader program receives them and does `projection*camera*model * point`

Projection Properties

Projections

- Recall there are two choices for how the view frustum is shaped: Perspective or Orthographic (parallel)
- The frustum has six planes, and the closest to the camera is called the “near plane”
- The projection matrix maps all 3D points that fall inside a frustum onto the near plane of that frustum, thereby reducing all shapes to 2D, for screen display.

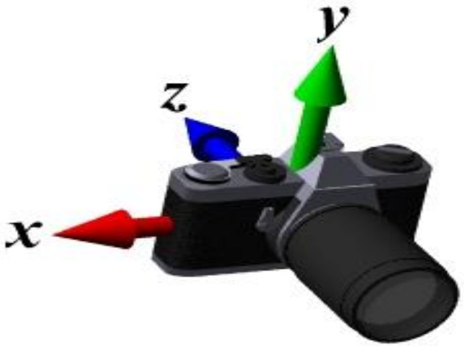
Projections: Online Demos

- http://threejs.org/examples/#webgl_camera
  - Perspective vs orthographic - the difference between the two projection frustums (and what they see) -- press O and P to switch between the two.
  - Clipping planes

- Also: My “Ray Tracer” example will show 3D frustums very clearly
  - Projecting their contents onto the near image plane

OpenGL Convention

In world coordinates, the camera system is defined as follows:



Projections

- We use a right handed system (x cross y = z)
- x and y in traditional plot directions make z go out of board, so
- We look down -z
  - Projection matrix is responsible for this flip!
  - Among other things

Transform Process

$$\underset{Viewport}{\begin{bmatrix} ? & 0 & 0 & ? \\ 0 & ? & 0 & ? \\ 0 & 0 & ? & ? \\ 0 & 0 & 0 & 1 \end{bmatrix}} * \underset{Projection}{\begin{bmatrix} ? & 0 & ? & ? \\ 0 & ? & ? & ? \\ 0 & 0 & ? & ? \\ 0 & 0 & ? & ? \end{bmatrix}} * \underset{Camera}{\begin{bmatrix} ? & ? & ? & ? \\ ? & ? & ? & ? \\ ? & ? & ? & ? \\ 0 & 0 & 0 & 1 \end{bmatrix}} * \underset{Model}{\begin{bmatrix} ? & ? & ? & ? \\ ? & ? & ? & ? \\ ? & ? & ? & ? \\ 0 & 0 & 0 & 1 \end{bmatrix}} * \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

- One more invisible thing happens after our code in addition to the viewport matrix:

  <u>The Perspective Division</u>

  - (Different from Perspective Matrix)

Transform Process

$$\underbrace{\begin{bmatrix} ? & 0 & 0 & ? \\ 0 & ? & 0 & ? \\ 0 & 0 & ? & ? \\ 0 & 0 & 0 & 1 \end{bmatrix}}_{Viewport} * \underbrace{\begin{bmatrix} ? & 0 & ? & ? \\ 0 & ? & ? & ? \\ 0 & 0 & ? & ? \\ 0 & 0 & ? & ? \end{bmatrix}}_{Projection} * \underbrace{\begin{bmatrix} ? & ? & ? & ? \\ ? & ? & ? & ? \\ ? & ? & ? & ? \\ 0 & 0 & 0 & 1 \end{bmatrix}}_{Camera} * \underbrace{\begin{bmatrix} ? & ? & ? & ? \\ ? & ? & ? & ? \\ ? & ? & ? & ? \\ 0 & 0 & 0 & 1 \end{bmatrix}}_{Model} * \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

- To do it, divide final vector [x,y,z,w] by its own w
  - (Not necessarily 1 anymore after projection matrix)
  - Can pull x and y closer to zero as depth increases
- No matrix can do that “row division” effect
  - It’s not a linear operation

Transform Process

- The chain of four special matrices is always the formula.
- Additionally, the Camera and Model parts are usually divided up further: They're stored in their own 4x4 matrix variables that you build up out of even smaller parts by accumulating transforms (rotation, translation, or scale matrices) during the program:

$$\underset{Model_1}{\begin{bmatrix} ? & ? & ? & ? \\ ? & ? & ? & ? \\ ? & ? & ? & ? \\ 0 & 0 & 0 & 1 \end{bmatrix}} * \underset{Model_2}{\begin{bmatrix} ? & ? & ? & ? \\ ? & ? & ? & ? \\ ? & ? & ? & ? \\ 0 & 0 & 0 & 1 \end{bmatrix}} * \underset{Model_3}{\begin{bmatrix} ? & ? & ? & ? \\ ? & ? & ? & ? \\ ? & ? & ? & ? \\ 0 & 0 & 0 & 1 \end{bmatrix}} * \underset{etc.}{\begin{bmatrix} ? & ? & ? & ? \\ ? & ? & ? & ? \\ ? & ? & ? & ? \\ 0 & 0 & 0 & 1 \end{bmatrix}} = [Model]$$

$$\underset{Camera_1}{\begin{bmatrix} ? & ? & ? & ? \\ ? & ? & ? & ? \\ ? & ? & ? & ? \\ 0 & 0 & 0 & 1 \end{bmatrix}} * \underset{Camera_2}{\begin{bmatrix} ? & ? & ? & ? \\ ? & ? & ? & ? \\ ? & ? & ? & ? \\ 0 & 0 & 0 & 1 \end{bmatrix}} * \underset{Camera_3}{\begin{bmatrix} ? & ? & ? & ? \\ ? & ? & ? & ? \\ ? & ? & ? & ? \\ 0 & 0 & 0 & 1 \end{bmatrix}} * \underset{etc.}{\begin{bmatrix} ? & ? & ? & ? \\ ? & ? & ? & ? \\ ? & ? & ? & ? \\ 0 & 0 & 0 & 1 \end{bmatrix}} = [Camera]$$

Matrix Order

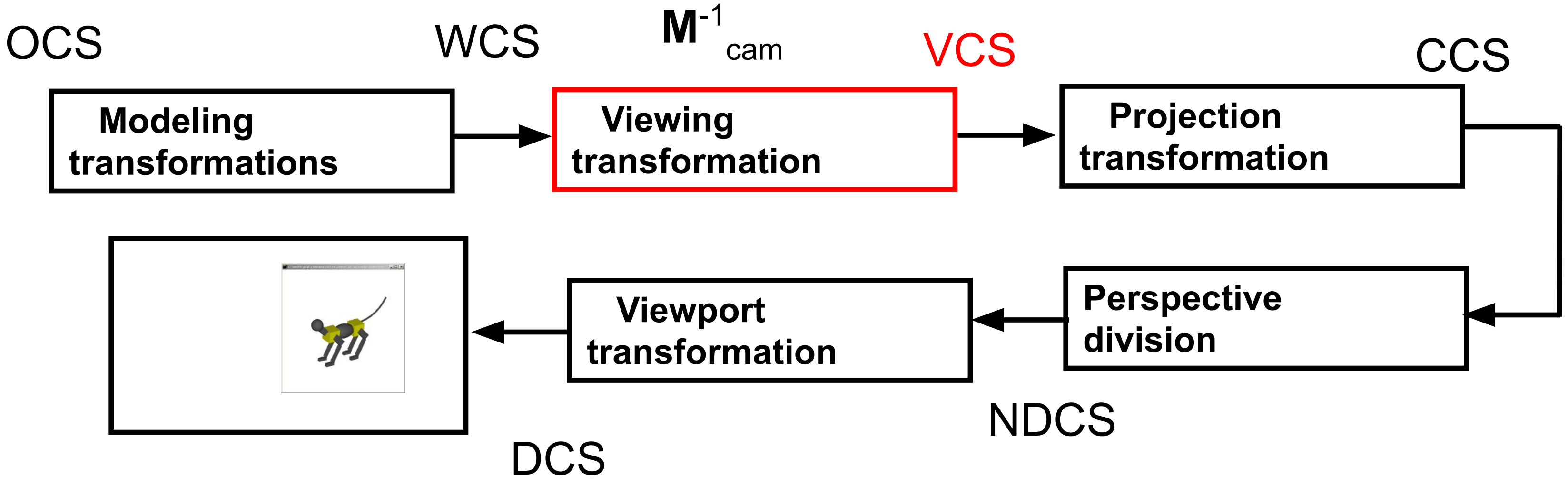
- Let's take a look at the complete formula again:

$$\underbrace{\begin{bmatrix} ? & 0 & 0 & ? \\ 0 & ? & 0 & ? \\ 0 & 0 & ? & ? \\ 0 & 0 & 0 & 1 \end{bmatrix}}_{Viewport} * \underbrace{\begin{bmatrix} ? & 0 & ? & ? \\ 0 & ? & ? & ? \\ 0 & 0 & ? & ? \\ 0 & 0 & ? & ? \end{bmatrix}}_{Projection} * \underbrace{\begin{bmatrix} ? & ? & ? & ? \\ ? & ? & ? & ? \\ ? & ? & ? & ? \\ 0 & 0 & 0 & 1 \end{bmatrix}}_{Camera} * \underbrace{\begin{bmatrix} ? & ? & ? & ? \\ ? & ? & ? & ? \\ ? & ? & ? & ? \\ 0 & 0 & 0 & 1 \end{bmatrix}}_{Model} * \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

- It's just a math formula, so there's no concept of first, last, or "order" while we're composing it.
- Instead of order, there is only left and right (non commutativity), and by convention the vertex is on the far right end in the formula no matter what.

Projection Math

Graphics Pipeline



Projection Transformations

Mapping: $T : R^n$ □ R^m

Projection: $n > m$

We are interested in

R^3□R^2 or

R^4□R^3 in homogenous coordinates

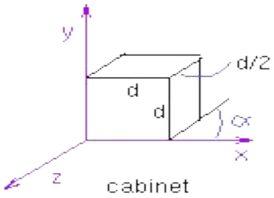
Planar Projections:

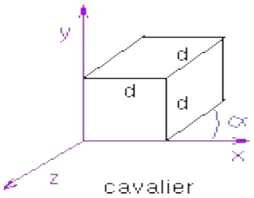
Projections onto a plane

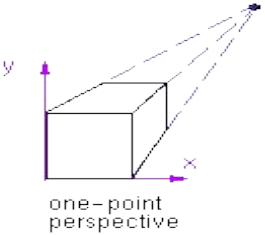
Taxonomy and Examples

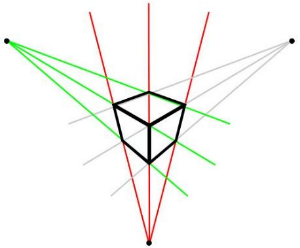
Planar Projections

- Parallel
  - Orthographic
    - Top
    - Front
    - Side
    - Axonometric:
      - Isometric
      - Dimetric
      - Trimetric
  - Oblique
    - Cabinet
    - Cavalier
- Perspective
  - 1 Point
  - 2 Point
  - 3 Point

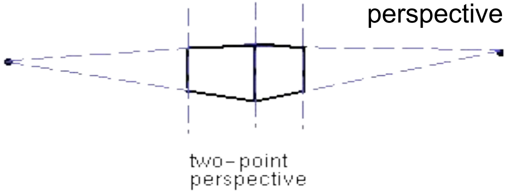


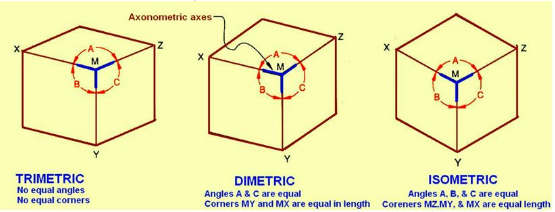






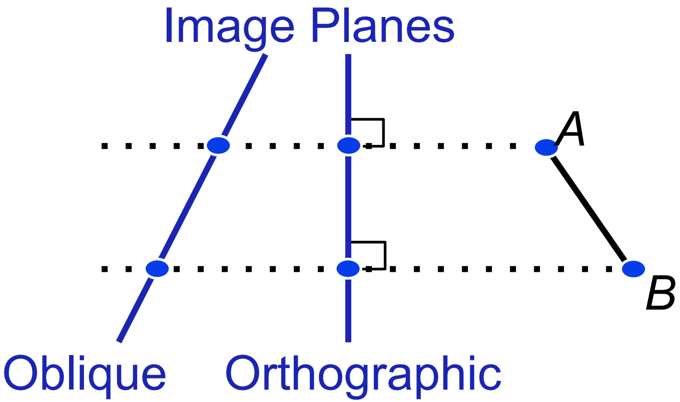
three-point
perspective

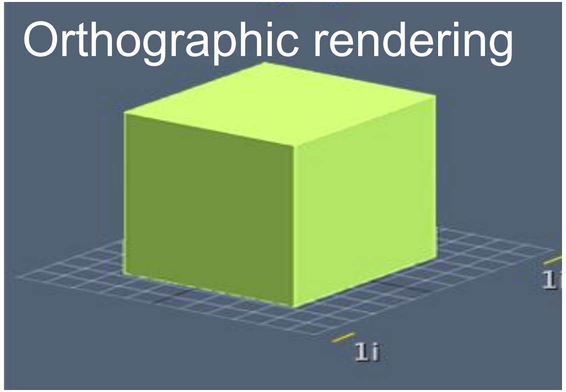




Basic Projections

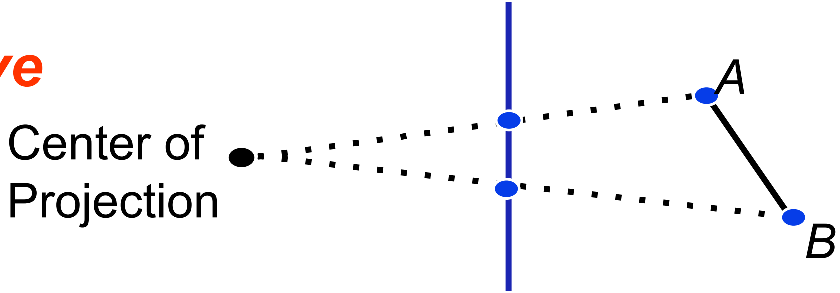
Parallel

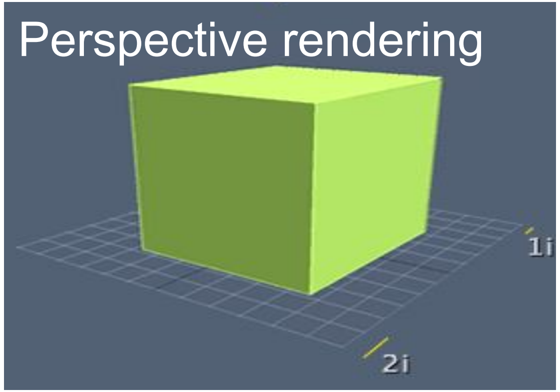




Orthographic rendering

Perspective





Perspective rendering

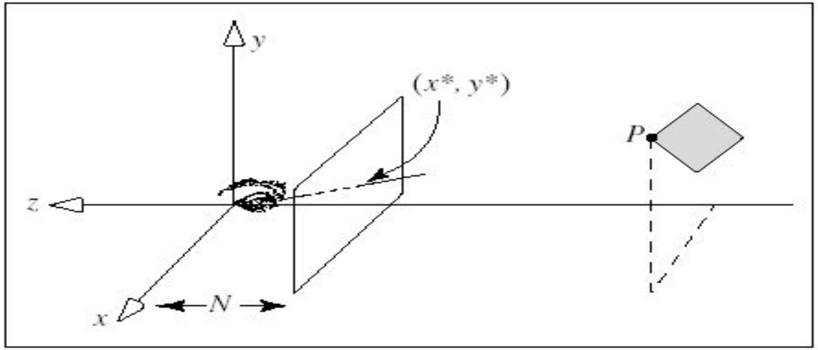
Camera Coordinate System

Camera at (0,0,0)

Looking at $-z$

***Image plane** (aka **near plane**)*

at $z = -N$

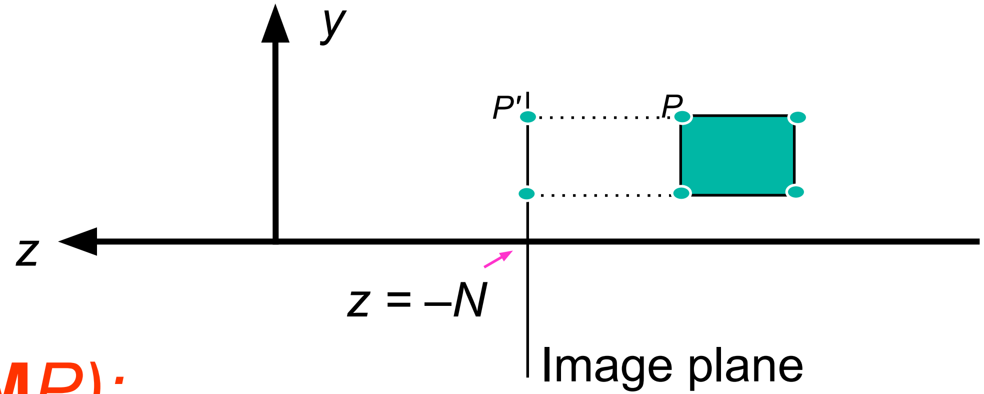


Basic Orthographic Projection

$P'_x = P_x$

$P'_y = P_y$

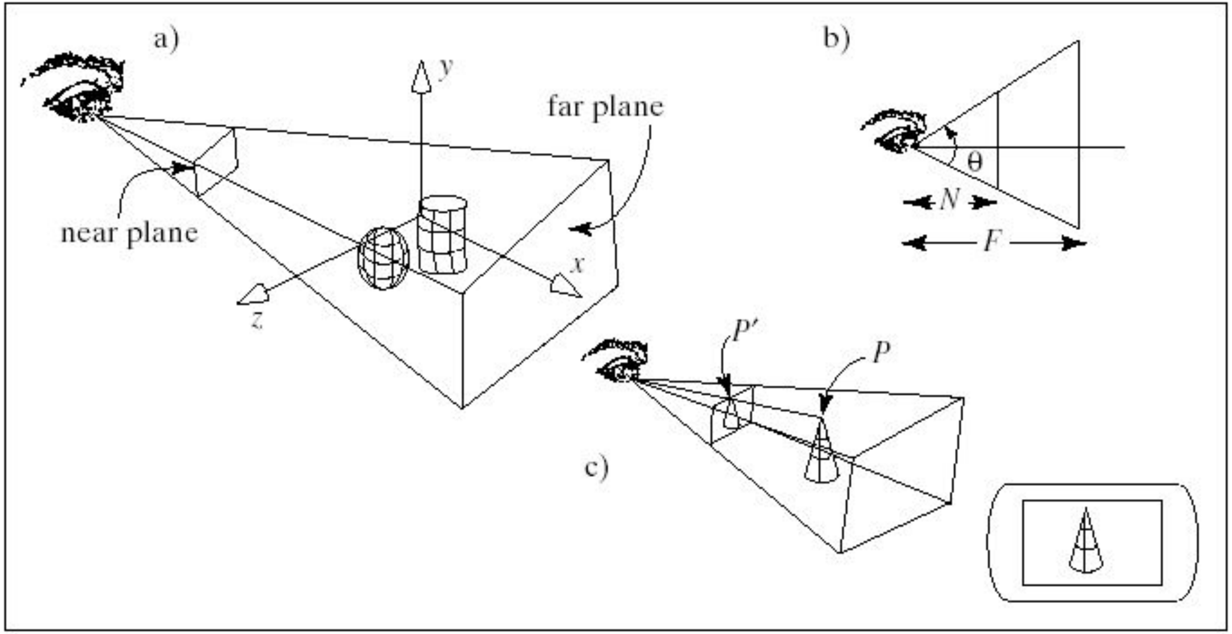
$P'_z = -N$



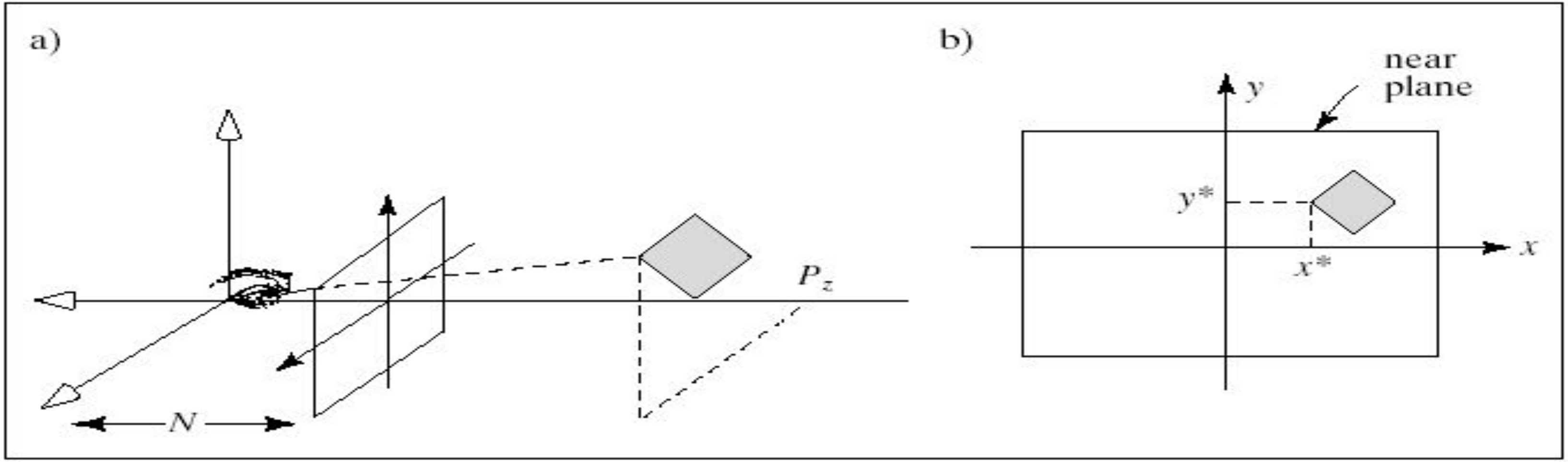
Matrix Form ($P' = \mathbf{M}P$):

$$\begin{bmatrix} P'_x \\ P'_y \\ P'_z \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & -N \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} P_x \\ P_y \\ P_z \\ 1 \end{bmatrix}$$

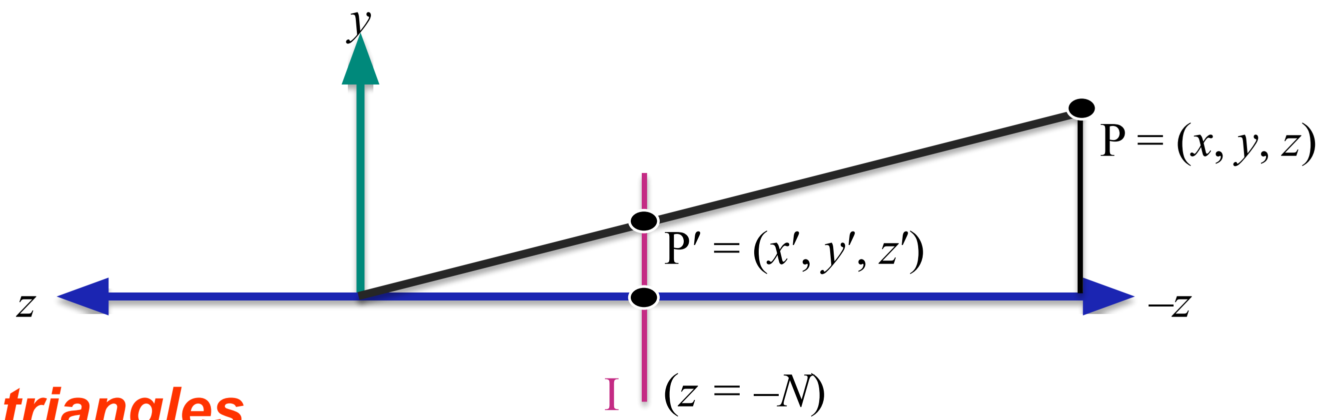
Perspective Projection



Perspective Projection of a Point



Basic Perspective Projection



Similar triangles

$$y'/N = y/-z \quad \Rightarrow \quad P_y' = P_y N/-P_z$$

Similarly

$$P_x' = P_x N/-P_z$$

$$P_z' = -N$$

This is a non-linear transformation!

Observations

- Projection undefined for $P_z = 0$
- If P is behind the eye,
  P_z changes sign
- Near plane just scales the picture
- Straight line → straight line
- Perspective foreshortening

$$P'_x = -N\frac{P_x}{P_z}$$

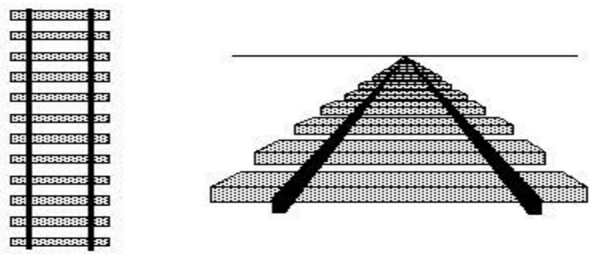
$$P'_y = -N\frac{P_y}{P_z}$$

$$P'_z = -N$$

Be Able to Answer:

- ***Given a point in x, y, z space, how do we calculate where it appears on the screen?***
- ***How is the perspective projection different from affine transformations?***
- ***What do perspective projections preserve?***
    - Parallel lines?
    - Ratios of points along a line?

Perspective transforms



What happens to parallel lines during the transform?

What happens to ratios along straight lines?

In Homogeneous Matrix Form

Reminder:

$$\begin{bmatrix} P_x \\ P_y \\ P_z \end{bmatrix} \rightarrow \begin{bmatrix} P_x \\ P_y \\ P_z \\ 1 \end{bmatrix} \xrightarrow[\times w]{} \begin{bmatrix} wP_x \\ wP_y \\ wP_z \\ w \end{bmatrix} \xrightarrow[\div w]{\text{homogenize}} \begin{bmatrix} P_x \\ P_y \\ P_z \\ 1 \end{bmatrix} \rightarrow \begin{bmatrix} P_x \\ P_y \\ P_z \end{bmatrix}$$

(a line in 4D space)

Perspective projection:

$$\begin{bmatrix} P'_x \\ P'_y \\ P'_z \\ 1 \end{bmatrix} = \begin{bmatrix} P_x N/(-P_z) \\ P_y N/(-P_z) \\ -N \\ 1 \end{bmatrix} \xrightarrow[\times -P_z/N]{} \begin{bmatrix} P_x \\ P_y \\ P_z \\ -P_z/N \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -1/N & 0 \end{bmatrix} \begin{bmatrix} P_x \\ P_y \\ P_z \\ 1 \end{bmatrix}$$

Therefore: $$\underbrace{\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -1/N & 0 \end{bmatrix}}_{\text{Matrix } \mathbf{M}} \begin{bmatrix} P_x \\ P_y \\ P_z \\ 1 \end{bmatrix} \xrightarrow[\div -P_z/N]{\text{and then: homogenize}} \begin{bmatrix} P'_x \\ P'_y \\ P'_z \\ 1 \end{bmatrix}$$

$\div -P_z/N$

Homogenization step:
"Perspective Division"
(divide by $w = -P_z/N$)

Perspective Projection of a Line

$$L(t) = \mathbf{P} + \mathbf{v}t = \begin{bmatrix} P_x \\ P_y \\ P_z \\ 1 \end{bmatrix} + \begin{bmatrix} v_x \\ v_y \\ v_z \\ 0 \end{bmatrix} t$$

Perspective Division & drop fourth coordinate

$\longrightarrow$

Is it still a line?

$$\text{Original:}\quad L(t) = \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} P_x + v_x t \\ P_y + v_y t \\ P_z + v_z t \end{bmatrix}$$

$$\text{Projected:}\quad L'(t) = \begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} -Nx / z \\ -Ny / z \\ -N \end{bmatrix} = \begin{bmatrix} -N(P_x + v_x t) / (P_z + v_z t) \\ -N(P_y + v_y t) / (P_z + v_z t) \\ -N \end{bmatrix}$$

$$x' = -N(P_x + v_x t) / (P_z + v_z t) \Rightarrow x'(P_z + v_z t) = -N(P_x + v_x t) \Rightarrow$$

$$x' P_z + x' v_z t = -NP_x - Nv_x t \Rightarrow \begin{cases} x' P_z + NP_x = -(x' v_z + Nv_x)t \\ \quad \text{and similarly for y:} \\ y' P_z + NP_y = -(y' v_z + Nv_y)t \end{cases}$$

Is it still a line? (cont'd)

$$\left.\begin{array}{l} x'P_z + NP_x = -(x'v_z + Nv_x)t \\ y'P_z + NP_y = -(y'v_z + Nv_y)t \end{array}\right| \Rightarrow \left.\begin{array}{l} x'P_z + NP_x = -(x'v_z + Nv_x)t \\ (y'v_z + Nv_y)t = -(y'P_z + NP_y) \end{array}\right| \Rightarrow$$

Multiply the two equations and divide through by *t*

$$(x'P_z + NP_x)(y'v_z + Nv_y) = (x'v_z + Nv_x)(y'P_z + NP_y) \Rightarrow$$

$$x'P_z y'v_z + x'P_z N v_y + NP_x y'v_z + N^2 P_x v_y = x'v_z y'P_z + x'v_z N P_y + Nv_x y'P_z + N^2 P_y v_x \Rightarrow$$

$$(P_z N c_y - v_z N P_y)x' + (NP_x v_z + Nv_x P_z)y' + N^2(P_x v_y + P_y v_x) = 0 \Rightarrow$$

$\Rightarrow$ $\boxed{ax' + by' + c = 0}$ which is the equation of a line in the x'-y' plane

But is There a Difference?

$$\text{Original:} \quad L(t) = \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} P_x + v_x t \\ P_y + v_y t \\ P_z + v_z t \end{bmatrix}$$

$$\text{Projected:} \quad L'(t) = \begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} -Nx / z \\ -Ny / z \\ -N \end{bmatrix} = \begin{bmatrix} -N(P_x + v_x t) / (P_z + v_z t) \\ -N(P_y + v_y t) / (P_z + v_z t) \\ -N \end{bmatrix}$$

But is There a Difference?

The "speed along the lines" if $v_z \neq 0$

$$\text{Original:} \quad L(t) = \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} P_x + v_x t \\ P_y + v_y t \\ P_z + v_z t \end{bmatrix} \Rightarrow \boxed{\frac{\partial L(t)}{\partial t} = \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix} = \mathbf{v}}$$

$$\text{Projected:} \quad L'(t) = \begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} -Nx/z \\ -Ny/z \\ -N \end{bmatrix} = \begin{bmatrix} -N(P_x + v_x t)/(P_z + v_z t) \\ -N(P_y + v_y t)/(P_z + v_z t) \\ -N \end{bmatrix} \Rightarrow$$

$$\frac{\partial x'}{\partial t} = -N \frac{\partial}{\partial t}((P_x + v_x t)/(P_z + v_z t)) = -N \frac{v_x (P_z + v_z t) - (P_x + v_x t) v_z}{(P_z + v_z t)^2} = -N \frac{v_x P_z - P_x v_z}{(P_z + v_z t)^2} \Rightarrow$$

$$\boxed{\frac{\partial L'(t)}{\partial t} = \frac{-N}{(P_z + v_z t)^2} \begin{bmatrix} v_x P_z - P_x v_z \\ v_y P_z - P_y v_z \\ 0 \end{bmatrix}}$$

As time t tends to infinity, the speed along the projected line L' tends to zero

Effect of Perspective Projection on Lines

Line equations

$$\text{Original:}\quad L(t) = \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} P_x + v_x t \\ P_y + v_y t \\ P_z + v_z t \end{bmatrix}$$

$$\text{Projected:}\quad L'(t) = \begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} -Nx/z \\ -Ny/z \\ -N \end{bmatrix} = \begin{bmatrix} -N(P_x + v_x t)/(P_z + v_z t) \\ -N(P_y + v_y t)/(P_z + v_z t) \\ -N \end{bmatrix}$$

If lines in space are parallel to the image plane then:

$$v_z = 0 \rightarrow L'(t) = -\frac{N}{P_z}\left[\begin{array}{c} P_x + v_x t \\ P_y + v_y t \\ P_z \end{array}\right]\Bigg]$$

slope of line: $\dfrac{v_y}{v_x}$

So, parallel lines parallel to the image plane remain parallel

Effect of Perspective Projection on Lines

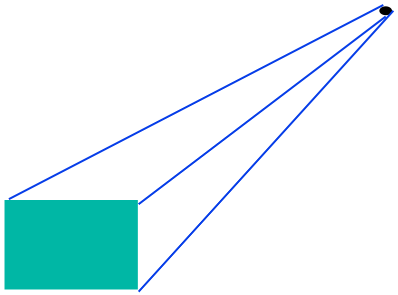
Line equations (again)

$$\text{Original:}\quad L(t) = \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} P_x + v_x t \\ P_y + v_y t \\ P_z + v_z t \end{bmatrix}$$

$$\text{Projected:}\quad L'(t) = \begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} -Nx/z \\ -Ny/z \\ -N \end{bmatrix} = \begin{bmatrix} -N(P_x + v_x t)/(P_z + v_z t) \\ -N(P_y + v_y t)/(P_z + v_z t) \\ -N \end{bmatrix}$$

If lines are not parallel to the image plane then:

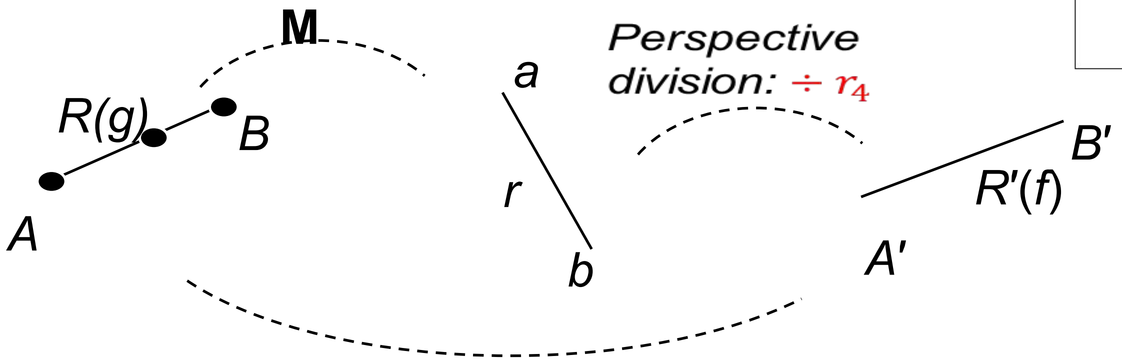
$$v_z \neq 0 \rightarrow \lim_{t \to \infty} L'(t) = \begin{bmatrix} -N v_x / v_z \\ -N v_y / v_z \\ -N \end{bmatrix}$$

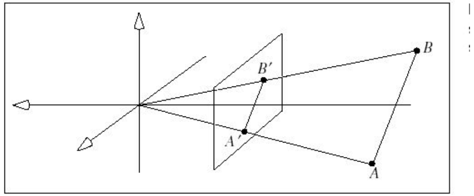


Lines converge to a **vanishing point**

Foreshortening: In-Between Points on Perspective-Projected Lines

How do points on lines transform?





In View Coordinate System (VCS): $R(g) = (1 - g)A + gB$

Projected homogeneous 4D: $r = \mathbf{M}R$

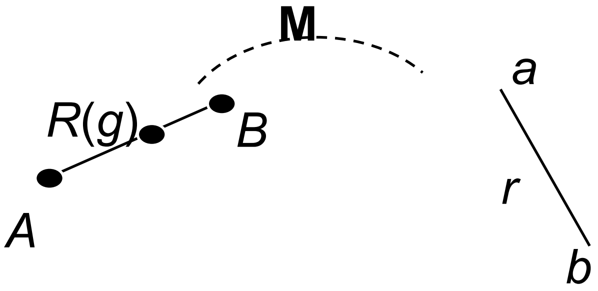
Projected homogeneous 3D: $R'(f) = (1 - f)A' + fB'$

***g* and *f* are <u>not</u> the same**

What is the relationship between *g* and *f* ?

First Step

Viewing space to homogeneous space (4D)



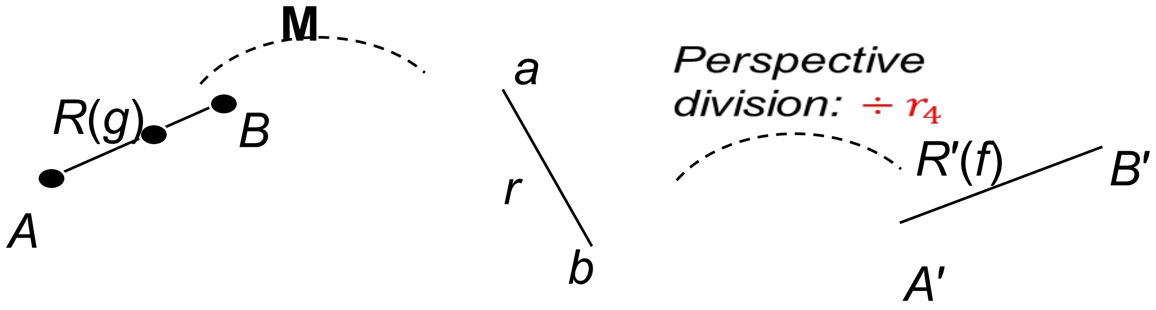
$$R = (1-g)A + gB$$

$$a = \mathbf{M}A = [a_1, a_2, a_3, a_4]^T$$
$$b = \mathbf{M}B = [b_1, b_2, b_3, b_4]^T$$

$$r = \mathbf{M}R = \mathbf{M}[(1-g)A + gB] = (1-g)\mathbf{M}A + g\mathbf{M}B \Rightarrow$$
$$r = (1-g)a + gb$$

Second Step

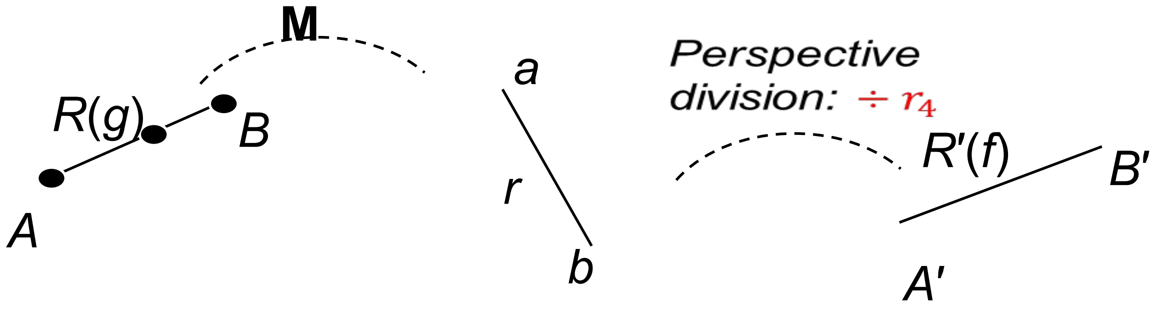
Perspective division



$$\left\{ \begin{array}{l} r = (1-g)a + gb \\ a = [a_1, a_2, a_3, a_4]^T \\ b = [b_1, b_2, b_3, b_4]^T \end{array} \right\} \Rightarrow \boxed{R'_1 = \frac{r_1}{r_4} = \frac{(1-g)a_1 + gb_1}{(1-g)a_4 + gb_4}}$$

And similarly for R'_2 *and* R'_3 $(R'_4 = 1)$

Putting it Together



$$R'_1 = \frac{(1-g)a_1 + gb_1}{(1-g)a_4 + gb_4} = \frac{\mathrm{lerp}(a_1, b_1, g)}{\mathrm{lerp}(a_4, b_4, g)}$$

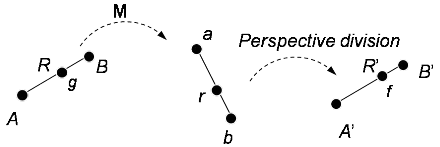
lerp: linear Interpolation
(done by hardware acceleration)

Furthermore:

$$R' = (1-f)A' + fB' \Rightarrow R'_1 = (1-f)A'_1 + fB'_1$$

$$R'_1 = (1-f)\frac{a_1}{a_4} + f\frac{b_1}{b_4} = \mathrm{lerp}\left(\frac{a_1}{a_4}, \frac{b_1}{b_4}, f\right)$$

Relation Between the Fractions

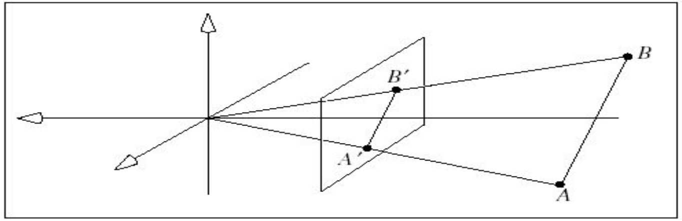


$$\left.\begin{array}{l} R'_1(f) = \dfrac{\operatorname{lerp}(a_1, b_1, g)}{\operatorname{lerp}(a_4, b_4, g)} \\[2ex] R'_1(f) = \operatorname{lerp}\left(\dfrac{a_1}{a_4}, \dfrac{b_1}{b_4}, f\right) \end{array}\right\} \Rightarrow g = \frac{f}{\operatorname{lerp}(\frac{b_4}{a_4}, 1, f)}$$

substituting this in $R(g) = (1-g)A + gB$ yields

$$R_1 = \frac{\operatorname{lerp}(\frac{A_1}{a_4}, \frac{B_1}{b_4}, f)}{\operatorname{lerp}(\frac{1}{a_4}, \frac{1}{b_4}, f)}$$

similarly for R_2 & R_3



WHAT THIS MEANS: For a given *f* in **image space** and *A, B* in **viewing space,** we can find the corresponding *R (or g)* in **viewing space** using the above formula

This works if "*A*", "*B*" are **positions**, **texture coordinates**, **color**, **normals**, etc.

So, it is generally VERY useful during rasterization (to be covered later)

Summary

Perspective projection is <u>non-linear</u>

Lines project to lines

Parallel lines either project to parallel lines or they intersect at the vanishing point

Foreshortening of projected lines and the "Inbetweeneness" relationship