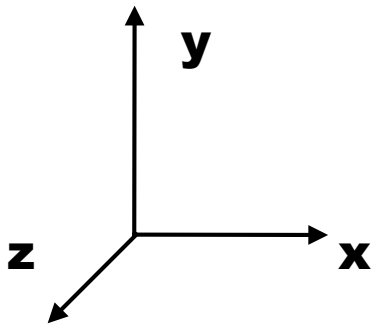
Game Architecture

2/19/16: Rasterization

Viewing

- To render a scene, need to know “Where am I” and “What am I looking at”
- The view transform is the matrix that does this
- Maps a standard “view space” into world space
- Typically defined by a point (location), and two vectors (direction and up)
- Can, of course, use a 4x4 matrix (transform)

View Space



- "Locked" to the camera
- Origin is view position (focal point), almost always in the center of the window
- The position and orientation of this space in world space tells us
  - Where the camera is located
  - What it is looking at

View Space

- x- and y-axes intuitive for screen coordinates
- z-axis is the view direction (depth)
- Can be right- or left-handed:
  - Left-handed can be intuitive; z = depth
  - OpenGL uses right-handed; −z = depth

View Space Transform

- A transform that can map

  World Space → View Space

- Often written as **V**
- Easier to discuss $\mathbf{V}^{-1}$ first
  - $\mathbf{V}^{-1}$ maps view → world
  - “Pick up a camera and aim”

Components of V^{-1}

$\mathbf{V}^{-1}$ is built from simple transforms

- View Translation
  - Where is the camera?
- View Orientation
  - What direction is the camera facing?

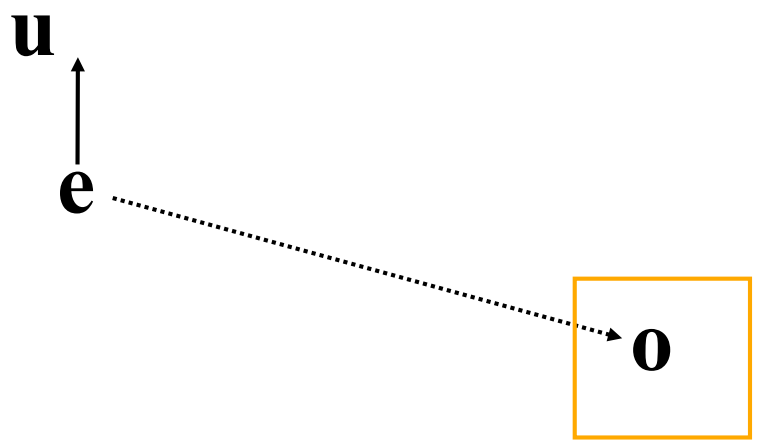
Components of V

V is built from the inverse transforms

- View Translation
  - Move the world to the camera
- View Orientation
  - Turn world to face the camera

Creating View Transform

- Have viewer position **e**, up direction **u**, point we want to look at **o**
- Want to compute view transform



View Translation

- Translate the view center *e* to the origin

$$E = \begin{vmatrix} 1 & 0 & 0 & -e_x \\ 0 & 1 & 0 & -e_y \\ 0 & 0 & 1 & -e_z \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

View Orientation

- View orientation consists of three view vectors (we defined these earlier):

World Space	View Space
View-direction vector	-z axis
View-right vector	x-axis
View-up vector	y-axis

Computing View Orientation

- First compute view direction vector

$$\mathbf{v}_{dir} = \mathbf{o} - \mathbf{e}$$

- Take cross product with up vector to get right vector

$$\mathbf{v}_{right} = \mathbf{v}_{dir} \times \mathbf{u}$$

Computing View Orientation

- Normalize these three vectors
- We have three unit-length, orthogonal vectors – basis vectors!
- Copy to columns of matrix $\mathbf{R_{VW}}$ – transforms from view orientation to world orientation

$$\mathbf{R}_{VW} = \begin{pmatrix} \mathbf{v}_{right} & \mathbf{v}_{up} & -\mathbf{v}_{dir} \end{pmatrix}$$

Verifying $\mathbf{R}_{VW}$

World Space = Transform x View Space

$$v_{right} = \mathbf{R}_{WV} \begin{pmatrix} 1 & 0 & 0 \end{pmatrix}$$

$$v_{up} = \mathbf{R}_{WV} \begin{pmatrix} 0 & 1 & 0 \end{pmatrix}$$

$$v_{dir} = \mathbf{R}_{WV} \begin{pmatrix} 0 & 0 & -1 \end{pmatrix}$$

Finishing View Orientation

$\mathbf{R_{VW}}$ maps view-to-world – we need world-to-view

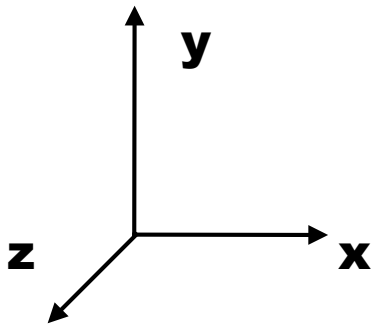
- Just invert $\mathbf{R_{VW}}$. Since $\mathbf{R_{VW}}$ is a rotation (orthogonal), we know:

$$\mathbf{R}_{WV} = \mathbf{R}_{VW}^{-1} = \mathbf{R}_{VW}^{T}$$

Other View Orientation Methods

- View Orientation is just a rotation
- Either use look-at orientation (prev seq)
- or convert orientation from other format (e.g. Quaternions)

Final V



- Viewing transformation **V**:
  - View translation matrix **E**
  - View orientation matrix $\mathbf{R_{WV}}$

$$\mathbf{V} = \mathbf{R_{WV}}\,\mathbf{E}$$

- Maps world space to view space

Prefabricated Look At

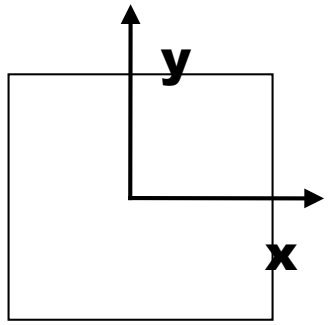
`glm::LookAt()` is equivalent to concatenating $\mathbf{V} = \mathbf{R}_{\mathbf{WV}}\,\mathbf{E}$

- Pretty much just use that
- But it is important to know how it works!
  - Complex camera interactions require an understanding of the view transform

Projection

- How we represent our camera's "lens"
- Maps a subset of the scene onto screen
- Generally, destination is a rectangular window
  - Often the full screen
  - Sometimes a texture (e.g., mirror)
  - Sometimes a GUI window (e.g., minimap)

NDC Space



Normalized Device Coordinates

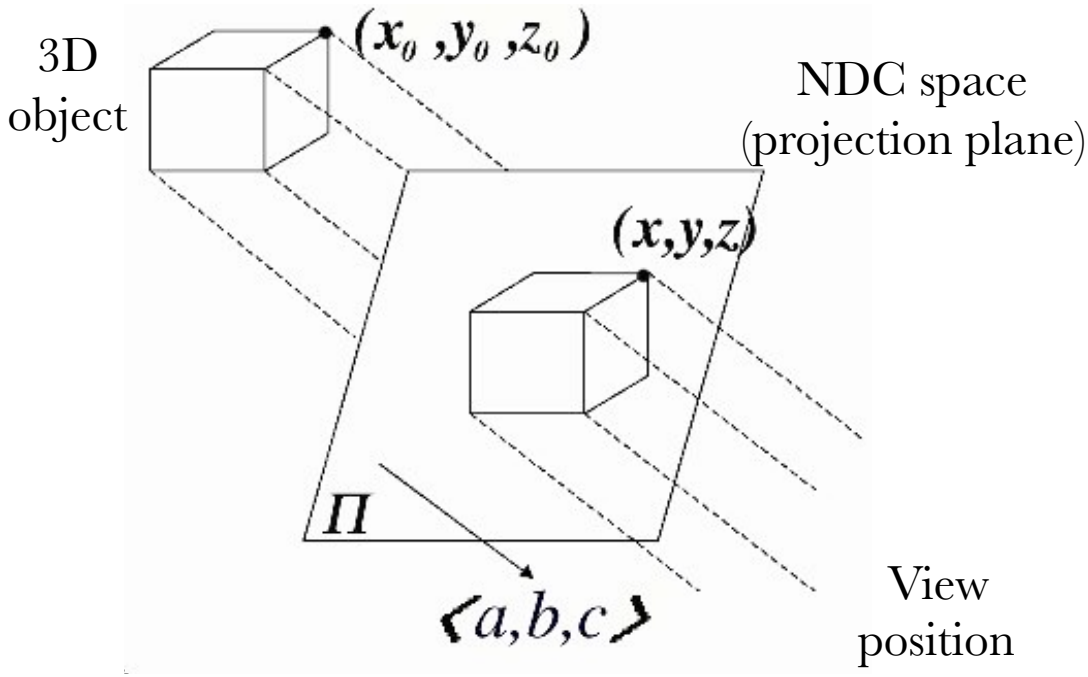
- The space of our "window"
- Lies in a plane
- Resolution independent
- Square (-1,-1) -> (1, 1)
- Origin at (0,0)
- "Visible" objects transformed into NDC space

Projection

- Must *project* (flatten) 3D view of camera to a plane — called projective transformation
- NDC space defines visible area of projection plane
- Two kinds
  - Parallel - linear
  - Perspective - non-linear

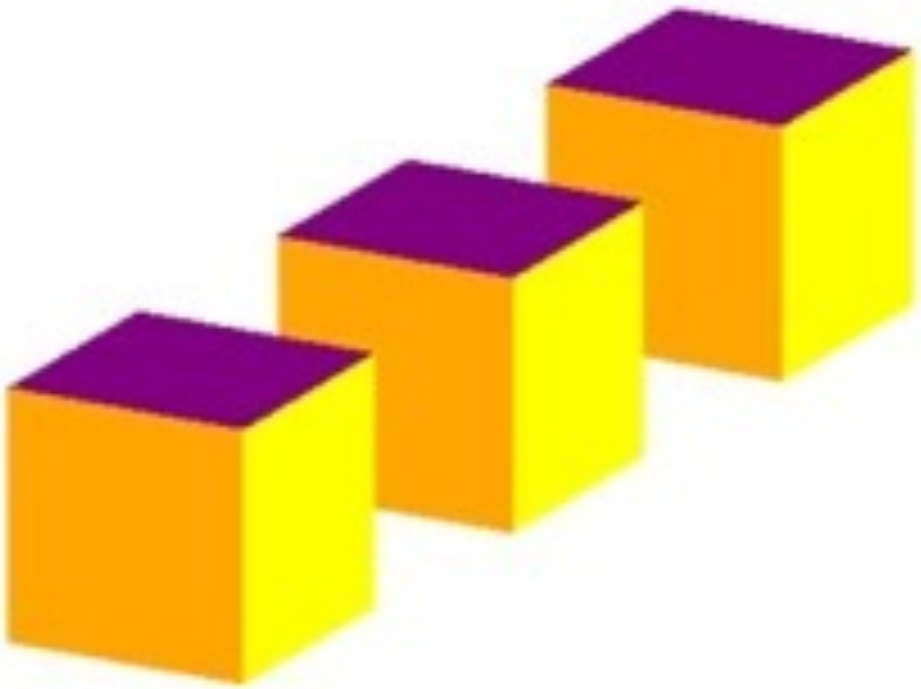
Parallel Projection

- "Flatten" in a constant direction
- Infinitely distant center of projection

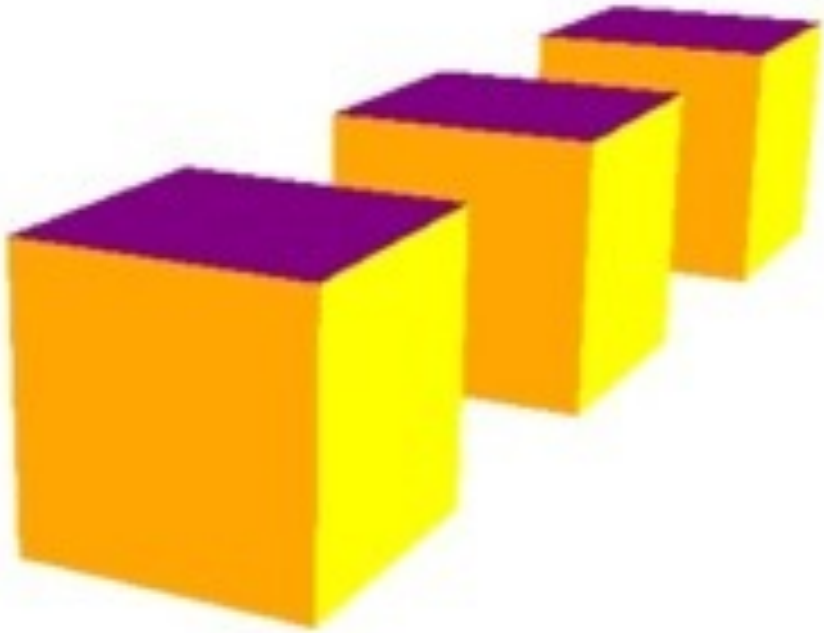


Parallel Projection

- Parallel lines remain parallel
- Orthographic most commonly used – projection perpendicular to plane
- Not how we see the world
- Mainly useful for art tools and special effects



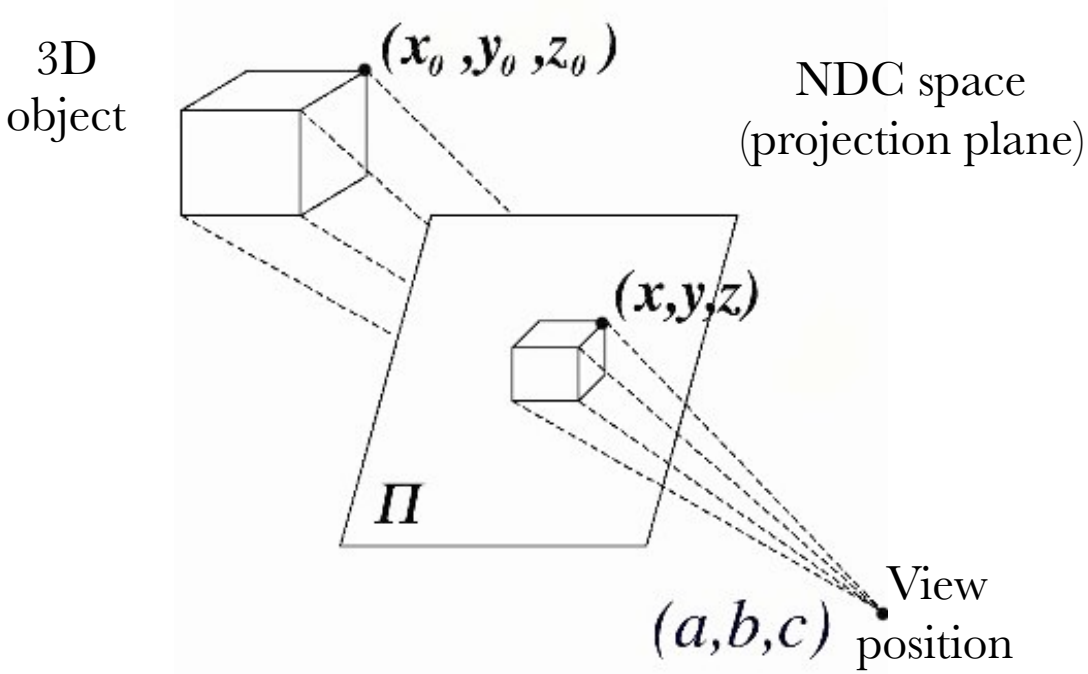
Orthographic Projection



Perspective Projection

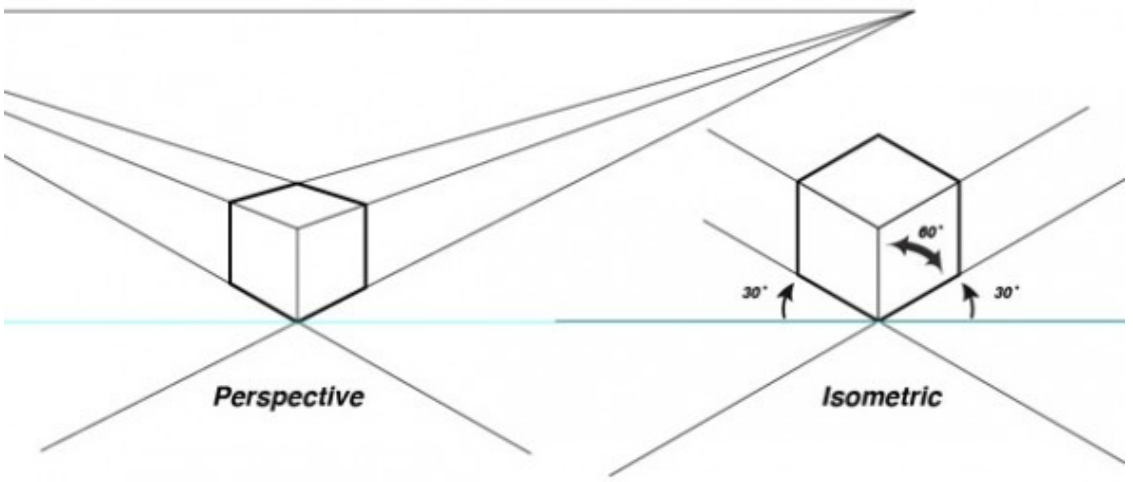
Perspective Projection

- Lines converge to single center of projection



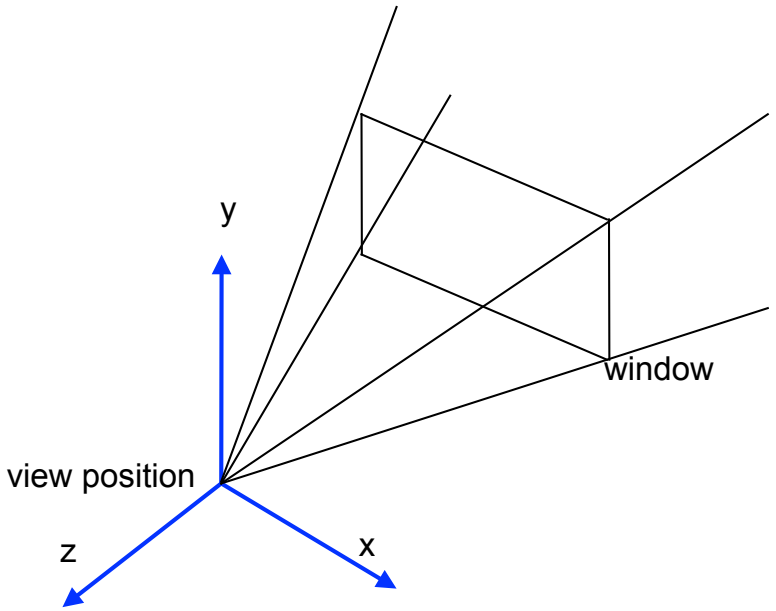
Perspective Projection

- Gives view we're used to
- Parallel lines in view direction merge
- Distances appear to shrink
- Non-linear



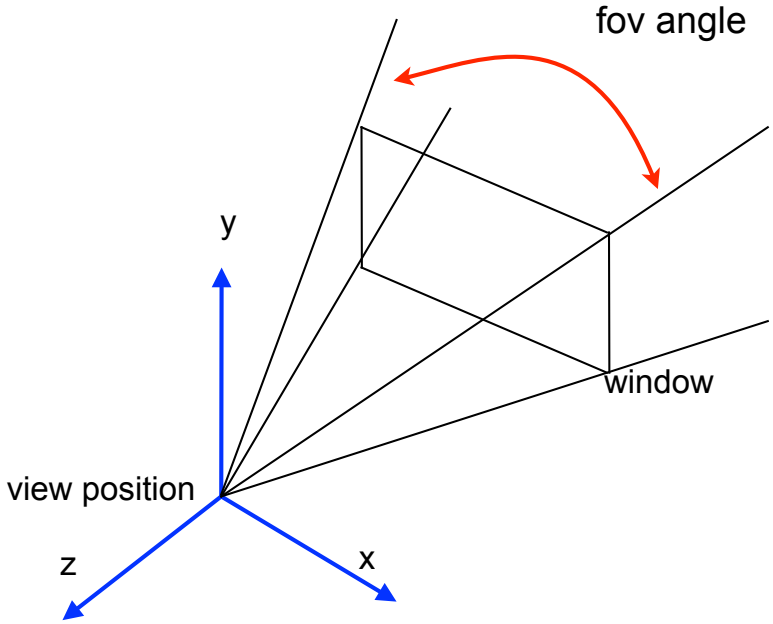
The View Frustum

- We need to know what part of view space to render. A "window" into the world



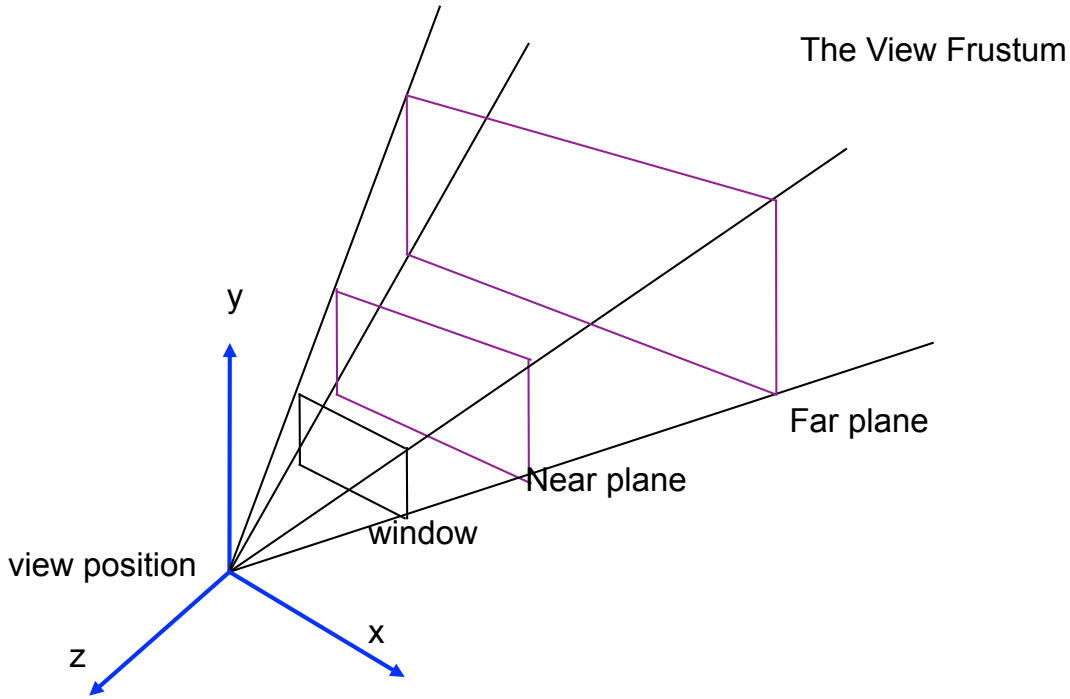
The View Frustum

- Size of window and closeness to eye determine field of view



The View Frustum

- Also defines how far objects are visible and how near objects are visible

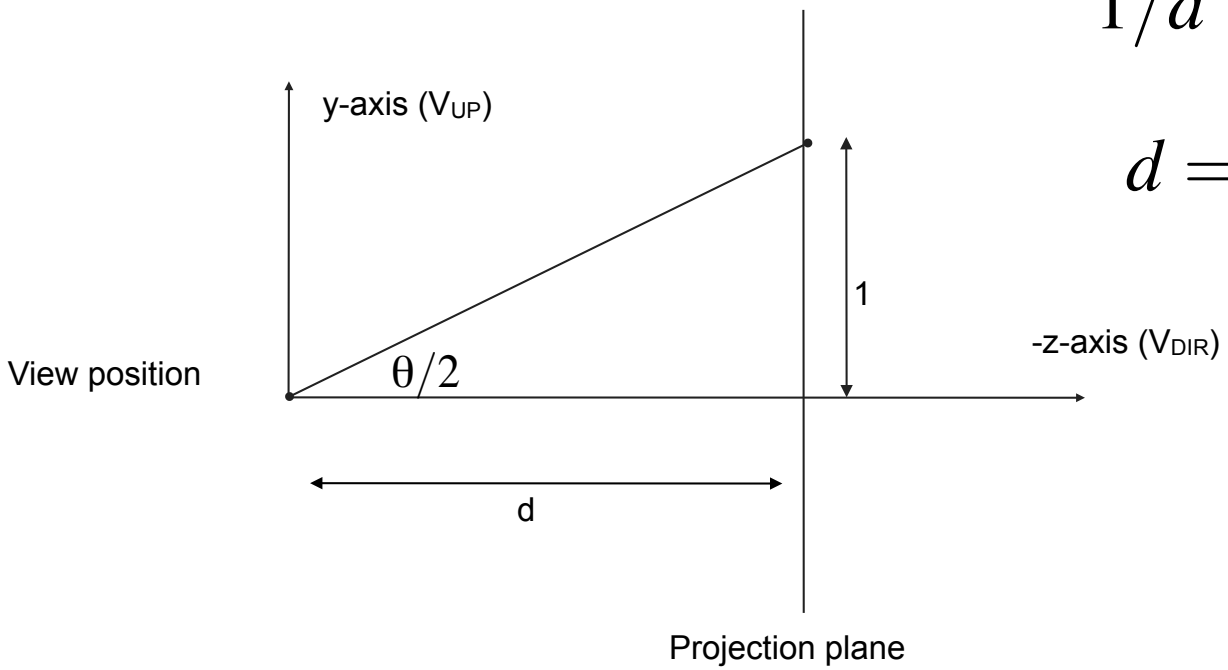


Field of View

- Related to location of projection plane

$$1/d = \tan(\theta_{fov}/2)$$

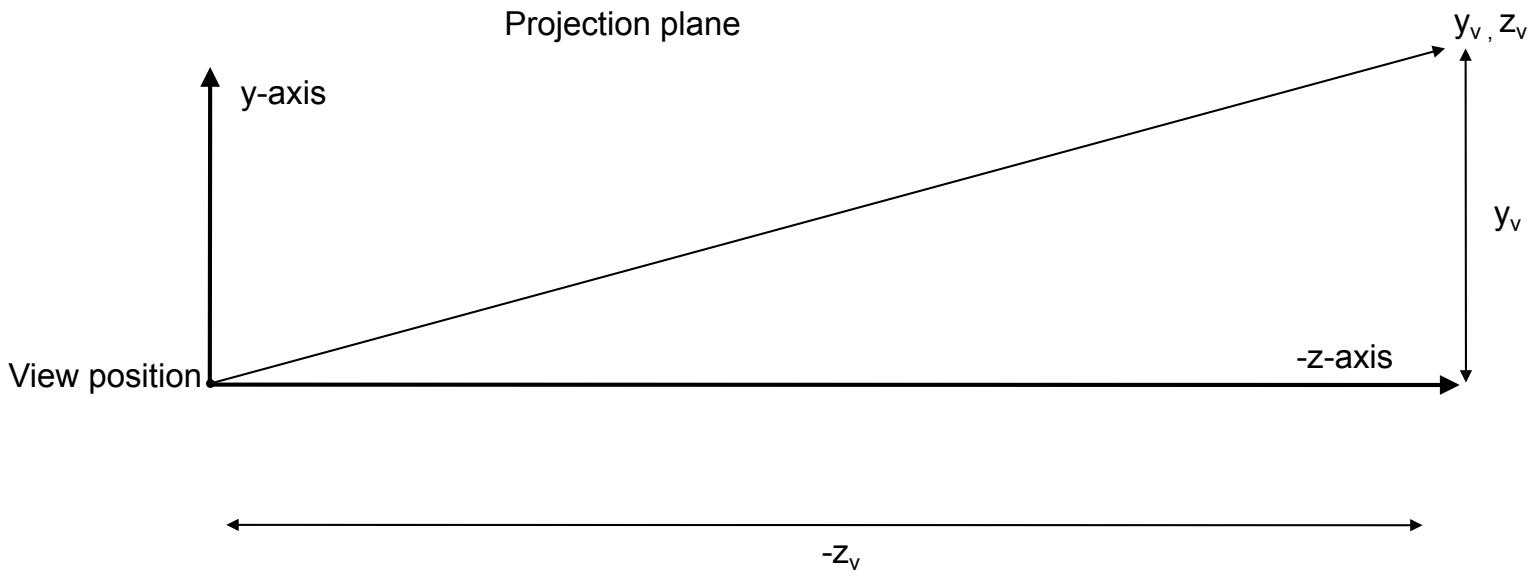
$$d = \frac{1}{\tan(\theta_{fov}/2)}$$



The Perspective Projection

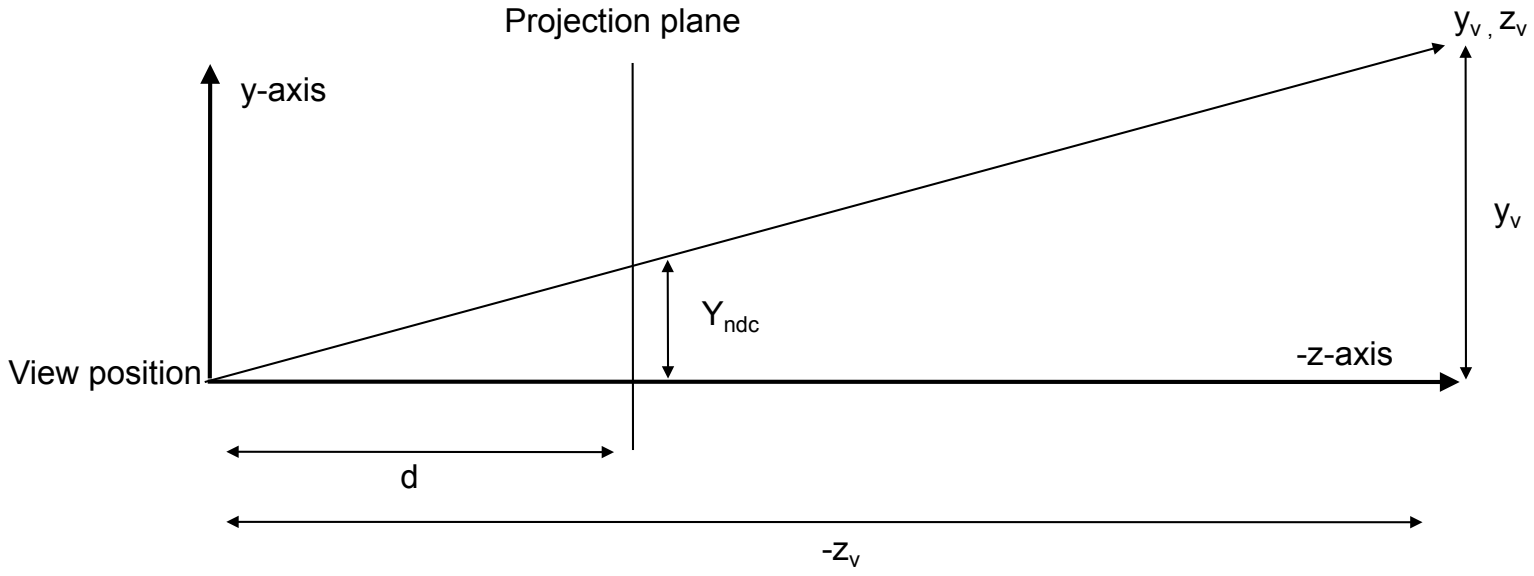
- Want to project a view-space point

The Perspective Projection



The Perspective Projection

- Similar triangles gives us $\frac{y_{ndc}}{d} = \frac{y_v}{-z_v}$



The Perspective Projection

$$\frac{y_{ndc}}{d} = \frac{y_v}{-z_v}$$

Solving for y_{ndc}

$$y_{ndc} = \frac{dy_v}{-z_v}$$

Aspect Ratio

- Screen probably isn't square
- Need to adjust area covered by projection plane by *aspect ratio*

$$a = \frac{w_s}{h_s}$$

- Assume y height remains 1, adjust x

The Perspective Projection

- Our projection equations

$$x_{ndc} = \frac{dx_v}{-az_v}$$

$$y_{ndc} = \frac{dy_v}{-z_v}$$

- It's a *non-linear* transformation

Homogeneous Perspective

- Transformation has linear and non-linear parts
- Linear part are the scales
- Non-linear part is the division
- Put linear part in a special homogeneous transformation matrix
- Dividing out the *w* can get us the perspective division

Perspective matrix

- The homogeneous perspective matrix

$$\begin{vmatrix} \frac{d}{a} & 0 & 0 & 0 \\ 0 & d & 0 & 0 \\ 0 & 0 & -d & 0 \\ 0 & 0 & -1 & 0 \end{vmatrix}$$

- Note that *w* is no longer 1
- Perspective matrix non-invertible
  - (right column all zeroes)

Perspective In Action

- Multiply linear part

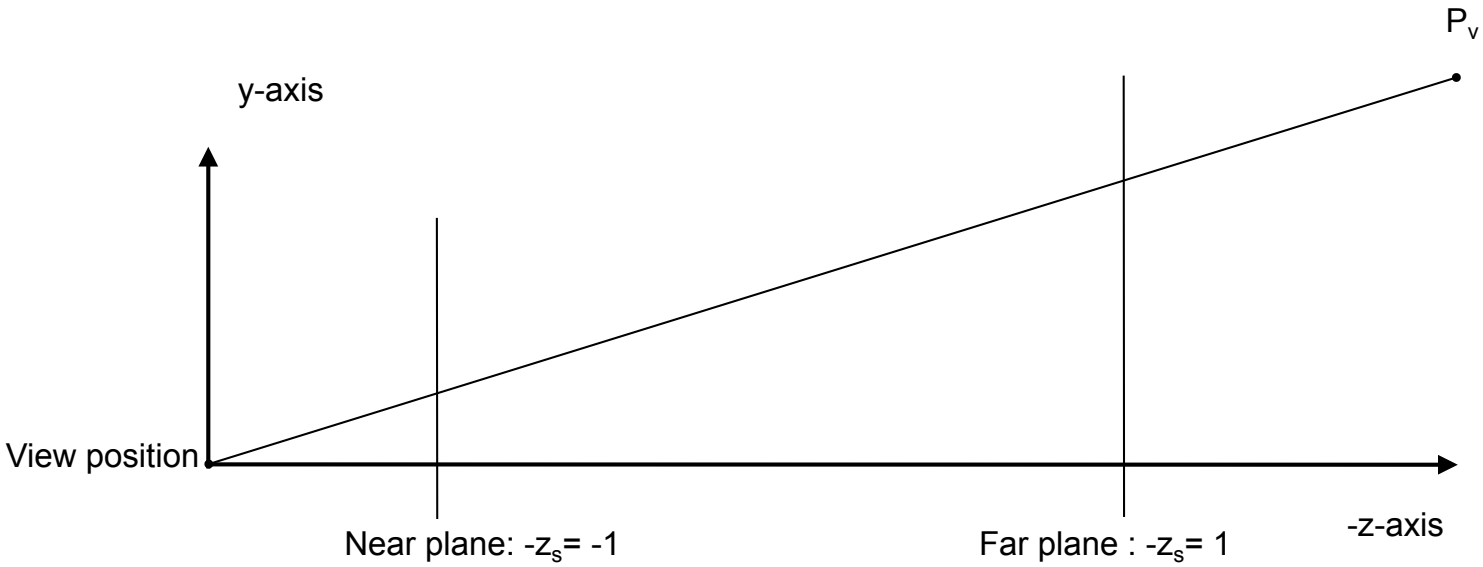
$$\begin{vmatrix} \frac{d}{a} & 0 & 0 & 0 \\ 0 & d & 0 & 0 \\ 0 & 0 & -d & 0 \\ 0 & 0 & -1 & 0 \end{vmatrix} \begin{vmatrix} x_v \\ y_v \\ z_v \\ w_v \end{vmatrix} = \begin{vmatrix} dx_v/a \\ dy_v \\ -dz_v \\ -z_v \end{vmatrix}$$

- Divide out the w

$$(x_{ndc}\, y_{ndc}\, z_{ndc}) = \frac{dx_v}{-az_v}\, \frac{dy_v}{-z_v}\, d$$

What happens to Z?

- Conceptually, it's lost in the projection
- In practice, we keep it for sorting chores (depth buffer!)
  - Map near plane to -1 and far plane to 1; the tighter they are, the better z-precision



Projecting The Z

- Projection equation for z

$$z_s = \frac{z_v(n+f)+2nf}{n-f}\left(\frac{1}{-z_v}\right)$$

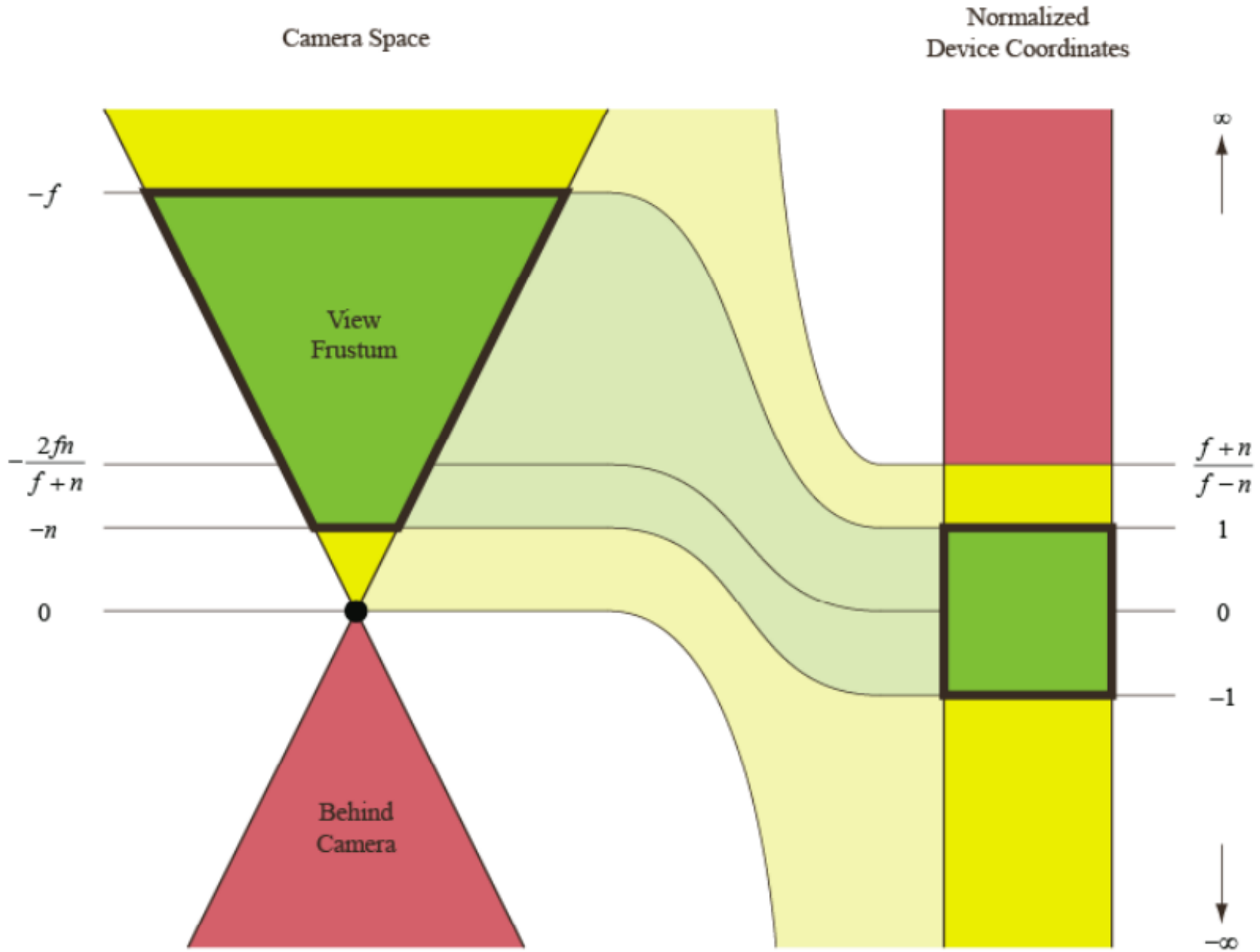
where n = near plane, f = far plane (distance from eye)

- Maps near to -1, far to 1 (OpenGL)
- Need to map to [0, 1] for DirectX

Final Projection Matrix

- This projection matrix has all this built in

$$\begin{vmatrix} \frac{d}{a} & 0 & 0 & 0 \\ 0 & d & 0 & 0 \\ 0 & 0 & \frac{n+f}{n-f} & \frac{2nf}{n-f} \\ 0 & 0 & -1 & 0 \end{vmatrix}$$


Camera Space
Normalized
Device Coordinates
$-f$
$-\frac{2fn}{f+n}$
$-n$
0
View
Frustum
Behind
Camera
∞
$\frac{f+n}{f-n}$
1
0
−1
$-\infty$

Notes on Projection

- Have described a basic viewing and projection system
- Others are more powerful, allowing oblique projections in which the projection plane is not parallel to the view vector
- All the major concepts in this one apply in the other ones

Generalized Perspective Projection

Introduction

Perspective projection is a well-understood aspect of 3D graphics. It is not something that 3D programmers spend much time thinking about. Most OpenGL applications simply select a field of view, specify near and far clipping plane distances, and call `gluPerspective` or `glFrustum`. These functions suffice in the vast majority of cases.

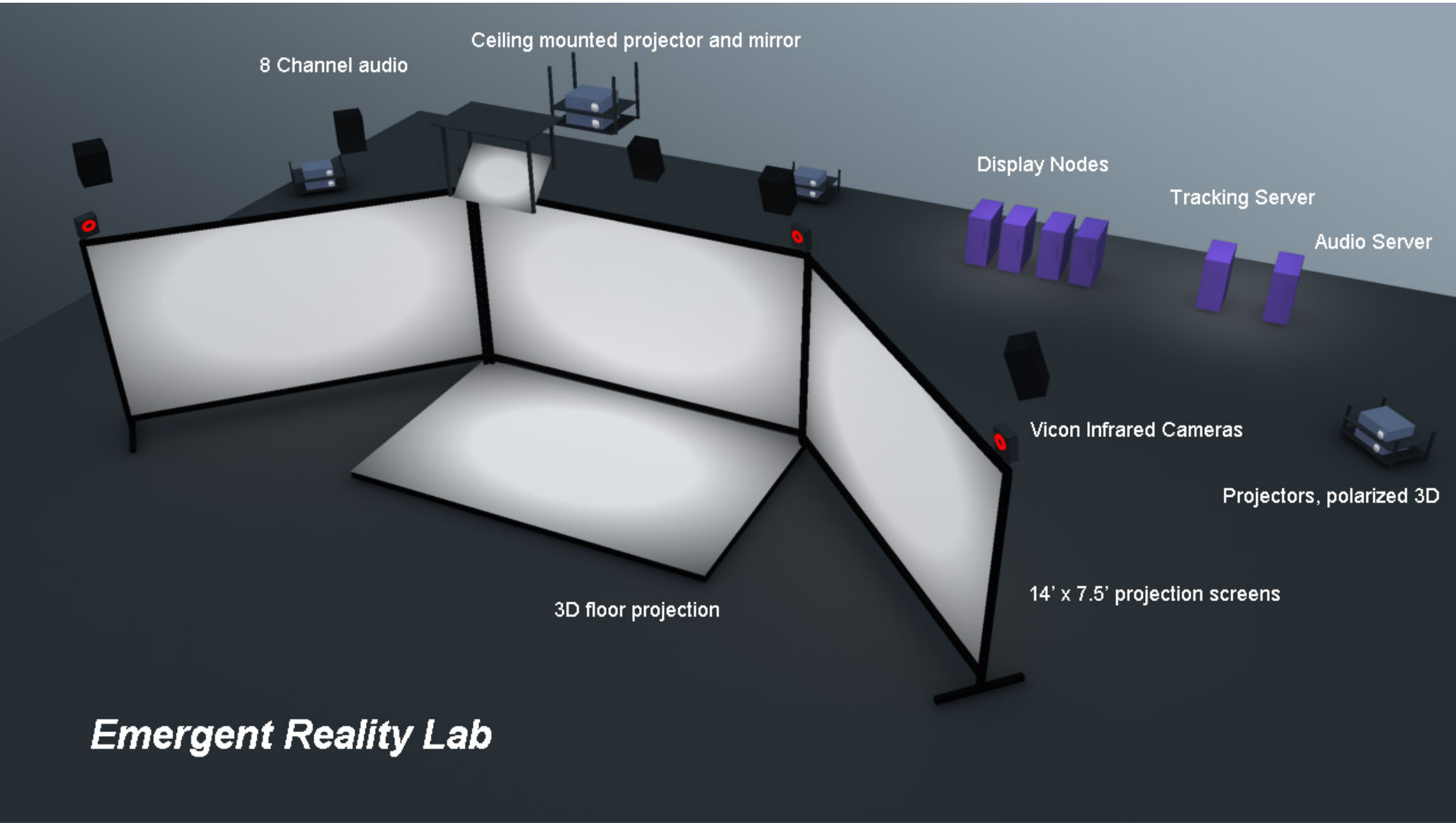
But there are a few assumptions implicit in these. `gluPerspective` assumes that the user is positioned directly in front of the screen, facing perpendicular to it, and looking at the center of it. `glFrustum` generalizes the position of the view point, but still assumes a perspective rooted at the origin and a screen lying in the *XY* plane.

The configuration of the user and his screen seldom satisfy these criteria, but perspective projection remains believable in spite of this. Leonardo's *The Last Supper* uses perspective, but still appears to be a painting of a room full of people regardless of the position from which you view it. Likewise, one can still enjoy a movie even when sitting off to the side of the theater.

Motivation

The field of Virtual Reality introduces circumstances under which these assumptions fail and the resulting incorrectness is not tolerable. VR involves a number complicating aspects: first-person motion-tracked perspective,

http://csc.lsu.edu/~kooima/articles/genperspective/index.html


Ceiling mounted projector and mirror
8 Channel audio
Display Nodes
Tracking Server
Audio Server
Vicon Infrared Cameras
Projectors, polarized 3D
14' x 7.5' projection screens
3D floor projection
Emergent Reality Lab

John Carmack
@ID_AA_Carmack

Following

Touchscreen games that use tilt controls could probably be improved by dynamically modifying the projection matrix to compensate.

Reply Retweet Favorite More

97 RETWEETS | **64** FAVORITES

9:51 PM - 9 Feb 13

Cameras are only the Start

- Projection matrices have many more uses today
- Shadows, dynamic lights, and some mirror techniques all use projection matrices
- These effects often require the more complex projections (oblique, etc)

Rendering: The Last Step

- The final goal is drawing to the screen
- Drawing involves creating a digital image of the projected scene
- Involves assigning colors to every pixel on the screen
- This is now done by dedicated hardware (consoles, modern PCs)
  - Used to be hand-optimized software (older cell phones, *old* PCs)

Destination: The Framebuffer

- We store the rendered image in the framebuffer
  - 2D digital image (grid of pixels)
  - Generally RGB(A)
- The graphics hardware reads these color values out to the screen and displays them to the user

Top-level Steps

We have a few steps remaining

Tessellation: How do we represent the surfaces of objects?

Shading: How do we assign colors to points on these surfaces?

Rasterization: How do we color pixels based on the shaded geometry?

Representing Surfaces: Points

- The geometry pipeline leaves us with:
  - Points in 2D screen (pixel) space (x_s, y_s)
  - Per-point depth value (z_{ndc})
- We could render these points directly by coloring the pixel containing each point
- If we draw enough of these points, *maybe* we could represent objects with them
  - But that could take millions of points (or more)

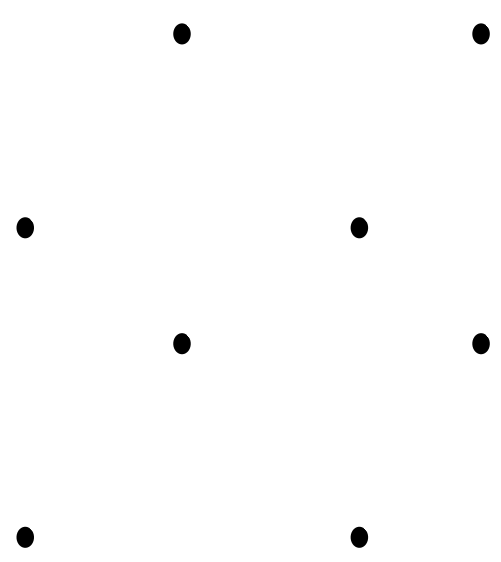
Representing Surfaces: Line Segments

- Could connect pairs of points with line segments
  - Draw lines in the framebuffer, just like a 2D drafting program
- This is called *wireframe* rendering
  - Useful in engineering and app debugging
  - Not very realistic
- But the *idea* is good – use sets of vertices to define higher-level primitives

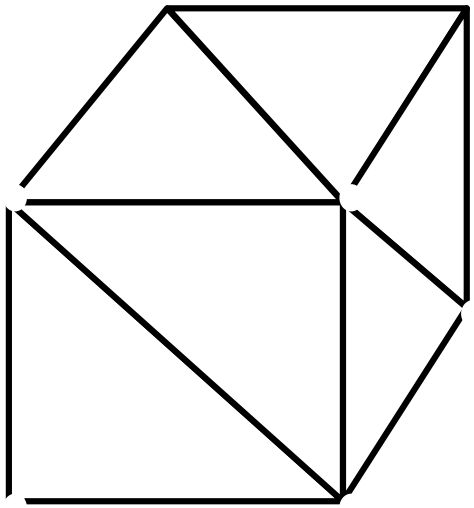
Representing Surfaces: Triangles

- We saw earlier that three points define a triangle
- Triangles are planar surfaces, bounded by three edges
- In real-time 3D, we use triangles to join together discrete points into surfaces

A Simple Cube



Cube represented by 8 points



Cube represented by 12 triangles

(defined by 8 points)

The Power of Triangles

- Flexible
  - We can approximate a wide range of surfaces using sets of triangles
- Tunable
  - A smooth surface can be approximated by more or fewer triangles
  - Allows us to trade off speed and accuracy
- Simple but Effective
  - We already know how to transform and project the three vertices that define a triangle

Vertices: Points++

- As we will see, we often need to store additional data at each distinct point that define our triangles
- We call these data structures *vertices*
- Vertices provide a way of storing the added values we need to compute the [color, normal vector, texture coords, etc] of the surface while rendering

Common Per-vertex Values

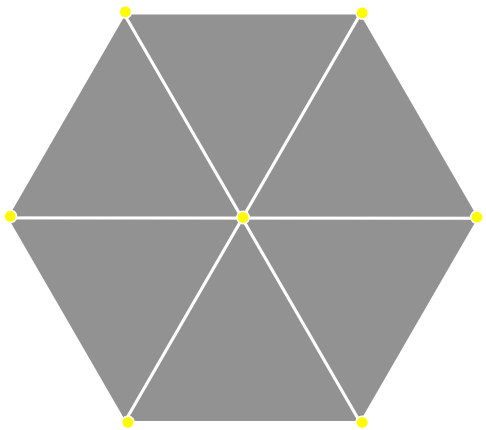
- Position
  - The “points” we’ve been transforming
- Normal vector
  - A vector perpendicular to the surface at the point
- Texture (or UV) coordinates
  - A value used to apply a digital image (akin to a decal on a plastic model) to the surface
- Color
  - The color of the surface at/near the point

Efficient Rendering of Triangle Sets

Indexed geometry allows vertices to be shared by several triangles

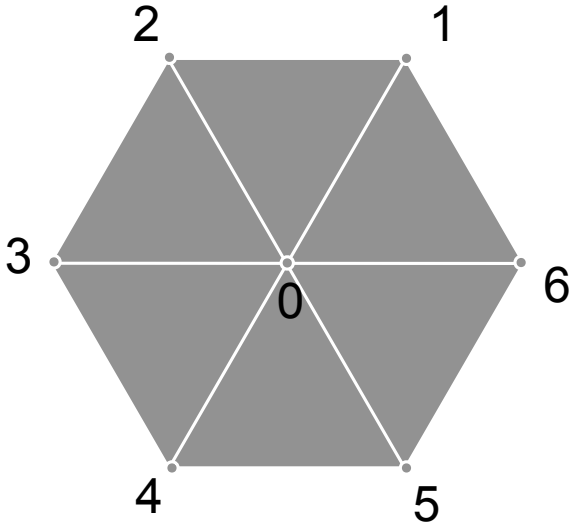
- Uses an array of indices into the array of vertices
- Each set of three indices determine a triangle
- Uses much fewer verts than (3 x Tris)

Indexed Geometry Example



Geometry

18 Individual vertices
(exploded view)



7 shared vertices

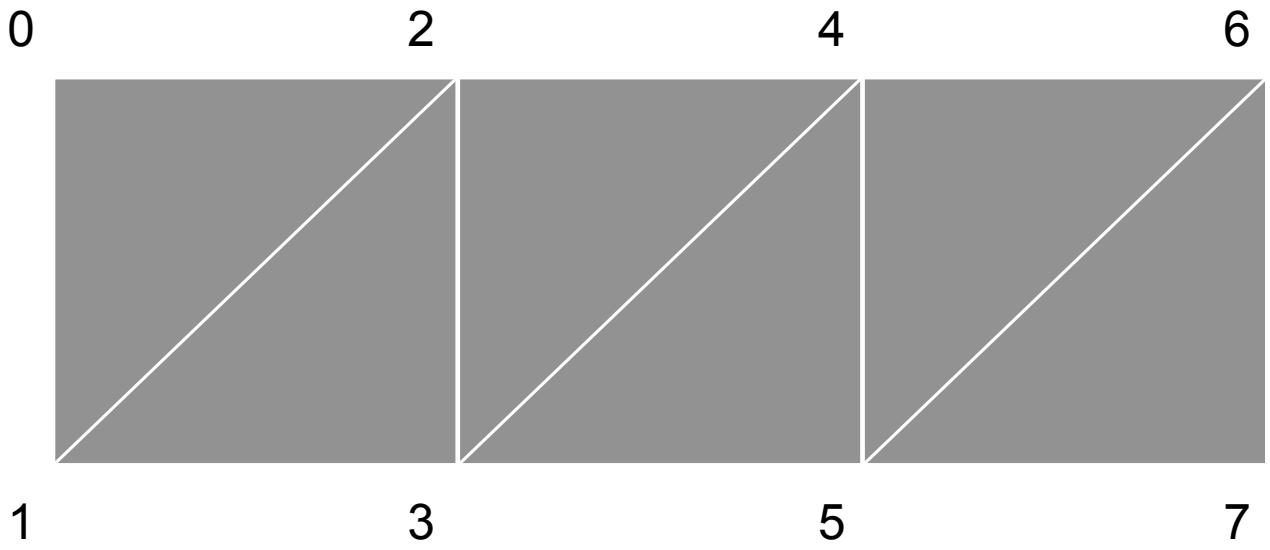
Index list for shared vertices (0,1,2),(0,2,3),(0,3,4),(0,4,5),(0,5,6),(0,6,1)

Note ccw winding, determines which side is the “front”

Even More Efficient

- Triangle Strips (tristrips)
  - Every vertex after the first two defines a triangle: (i-2, i-1, i)
- Tristrips use much shorter index lists
  - (2 + Tris) versus (3 x Tris)

Tristrip Example



Index list for shared vertices

(0, 1, 2, 3, 4, 5, 6, 7)

Triangle Set Implementation

- Most 3D hardware is optimized for tristrips
- However, transformed vertices are often "cached" in a limited-size collection
  - Indexed geometry is not "perfect" – using a vertex in two triangles may not gain much performance if the vertex is kicked out of the cache between them
- Lots of papers available from the hardware vendors on how to optimize for their caches

Geometry Representation Summary

- Generally, we draw triangles
- Triangles are defined by three screen-space vertices
- Vertices include additional information required to store or compute colors
- Indexed primitives allow us to represent triangles more efficiently

Homework 4

- Scan the source code of http://www.cogsci.rpi.edu/public_html/destem/gamearch/hw4.html and try not to panic
- Get the cube to rotate using your previously-written functions
- C++ users: I recommend using GLM, GLEW, and GLFW
- WebGL/OpenGL requires *a lot* to get going
- It gets better. Kinda.