CSE 167: Introduction to Computer Graphics Lecture #5: Projection

Jürgen P. Schulze, Ph.D.
University of California, San Diego
Fall Quarter 2018

Announcements

- Tomorrow: late grading for homework 1 2pm-3:15pm in CSE B260
  - Upload code to TritonEd by 2pm
  - Demonstrate in CSE basement labs
- Next Friday: homework 2 due at 2pm
  - Upload to TritonEd
  - Demonstrate in CSE basement labs
- Opportunities for CSE 199/198 or paid programmer positions
- Magic Leap Conference on future of AR:
  - Keynote address at:
    - https://www.youtube.com/watch?v=vV8oGahOSgc

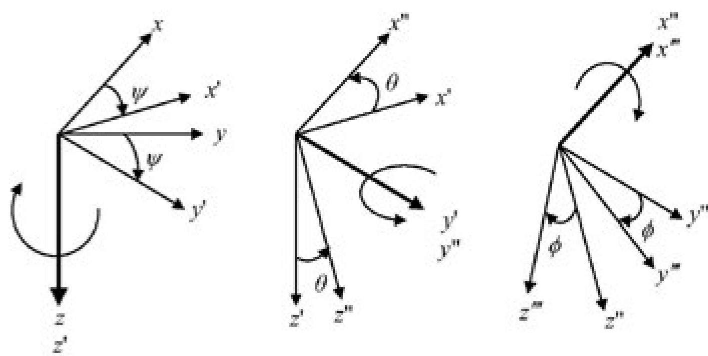
Topics

- Quaternions
- Projection

Quaternions

Rotation Calculations

- Intuitive approach: Euler Angles
  - Simplest way to calculate rotations
  - Defines rotation by 3 sequential rotations about coordinate axes
- Example for rotation order Z-Y-X:



1st, the body frame yaws right an angle ψ about the z-axis

2nd, the body frame pitches up an angle θ about the y'-axis

3rd, the body frame rolls CW an angle ϕ about the x''-axis

http://www.globalspec.com/reference/49379/203279/3-3-euler-angles

Problems With Euler Angles

- Problems with Euler angles:
  - No standard for order of rotations
  - Gimbal Lock, occurs in certain object orientations
    - Video: https://www.youtube.com/watch?v=rrUCBOlJdt4
- Better: rotation about arbitrary axis (no Gimbal lock)
  - Can be done with 4x4 matrix
- But: smoothly interpolating between two orientations is difficult
- → Quaternions

Quaternion Definition

- Given angle and axis of rotation:
  - a: rotation angle
  - {nx,ny,nz}: normalized rotation axis
- Calculation of quaternion coefficients w, x, y, z:
  - w = cos(a /2)
  - x = sin(a /2) * nx
  - y = sin(a /2) * ny
  - z = sin(a /2) * nz

Useful Quaternions

w	x	y	z	Description
1	0	0	0	Identity quaternion, no rotation
0	1	0	0	180° turn around X axis
0	0	1	0	180° turn around Y axis
0	0	0	1	180° turn around Zaxis
sqrt(0.5)	sqrt(0.5)	0	0	90° rotation around X axis
sqrt(0.5)	0	sqrt(0.5)	0	90° rotation around Y axis
sqrt(0.5)	0	0	sqrt(0.5)	90° rotation around Z axis
sqrt(0.5)	-sqrt(0.5)	0	0	-90° rotation around X axis
sqrt(0.5)	0	-sqrt(0.5)	0	-90° rotation around Y axis
sqrt(0.5)	0	0	-sqrt(0.5)	-90° rotation around Z axis

Quaternions in GLM

- Create a quaternion for a 90 degree rotation about the y axis:
  - glm::quat rot = glm::angleAxis(glm::radians(90.f), glm::vec3(0.f, 1.f, 0.f));
- Cast the quaternion into a 4x4 matrix:
  - glm::mat4 rotate = glm::mat4_cast(rot);

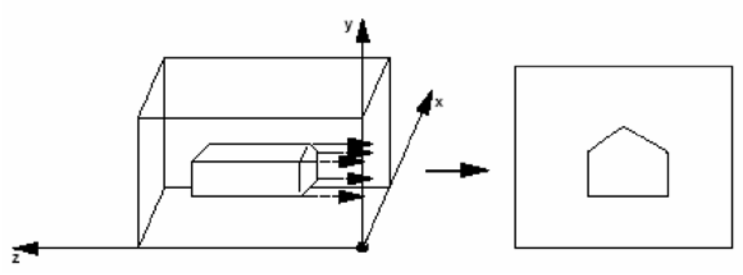
Quaternions: Further Reading

- Rotating Objects Using Quaternions:
  - http://www.gamasutra.com/view/feature/131686/rotating_objects_using_quaternions.php
- Quaternions in GLM:
  - http://www.opengl-tutorial.org/intermediate-tutorials/tutorial-17-quaternions/
- Quaternions in Unity 3D:
  - https://docs.unity3d.com/ScriptReference/Quaternion.html
- Quaternions in OpenSceneGraph :
  - http://www.openscenegraph.org/index.php/documentation/knowledge-base/40-quaternion-maths

Projection

Projection

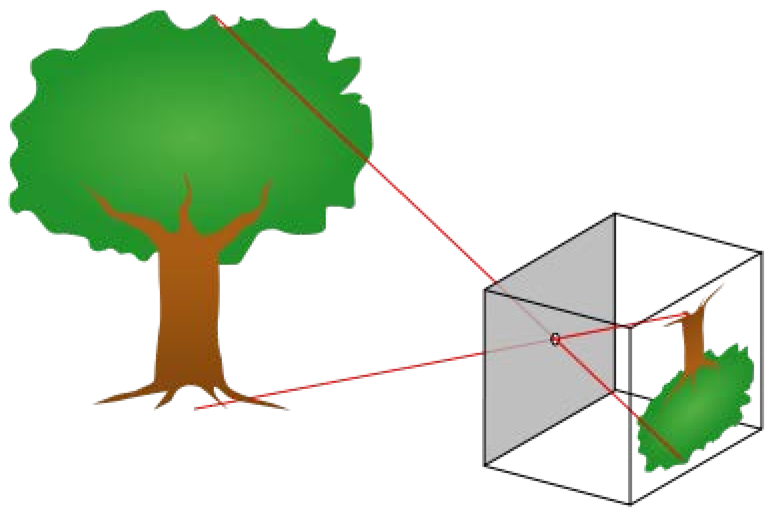
- Goal:
  Given 3D points (vertices) in camera coordinates, determine corresponding image coordinates
- Transforming 3D points into 2D is called Projection
- Typically one of two types of projection is used:
  - Orthographic Projection (=Parallel Projection)



  - Perspective Projection: most commonly used

Perspective Projection

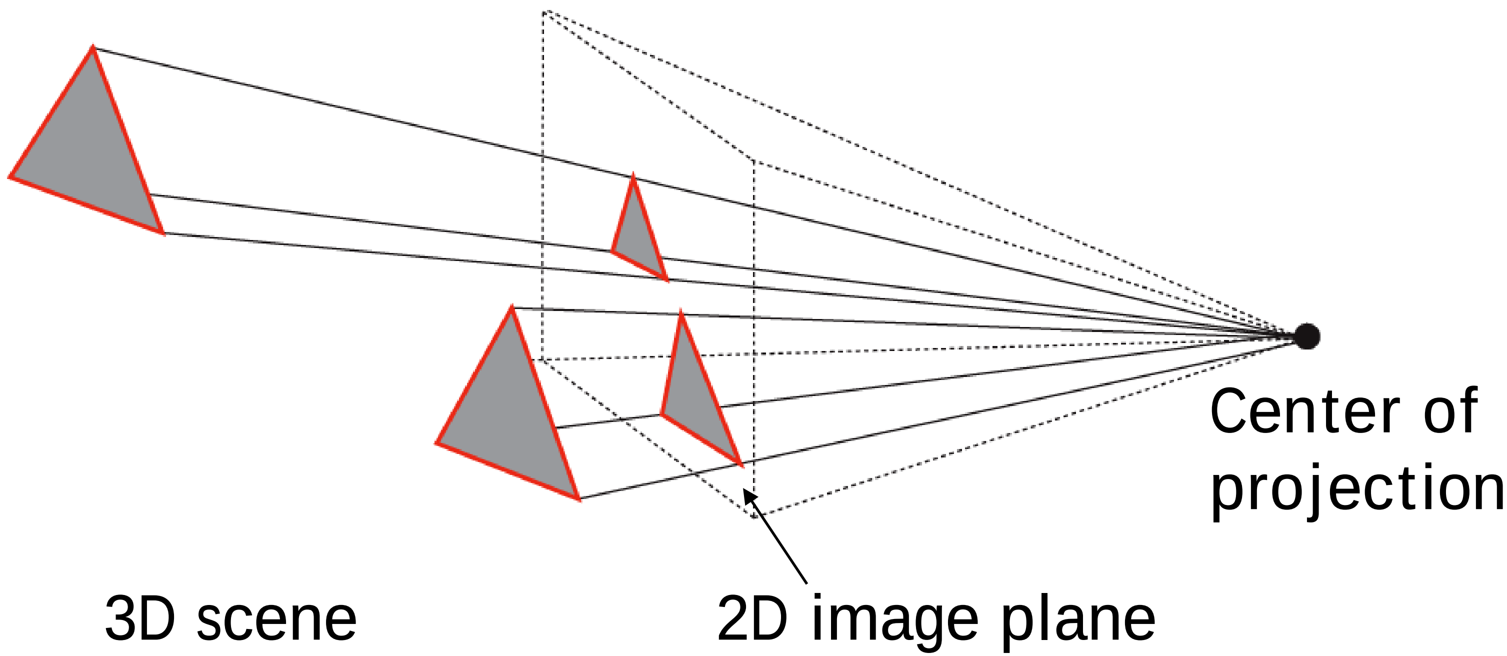
- Most common for computer graphics
- Simplified model of human eye, or camera lens (*pinhole camera*)



- Things farther away appear to be smaller
- Discovery attributed to Filippo Brunelleschi (Italian architect) in the early 1400's

Perspective Projection

- Project along rays that converge in center of projection



Perspective Projection

Parallel lines are
no longer parallel,
converge in one point

Earliest example:
La Trinitá (1427) by Masaccio

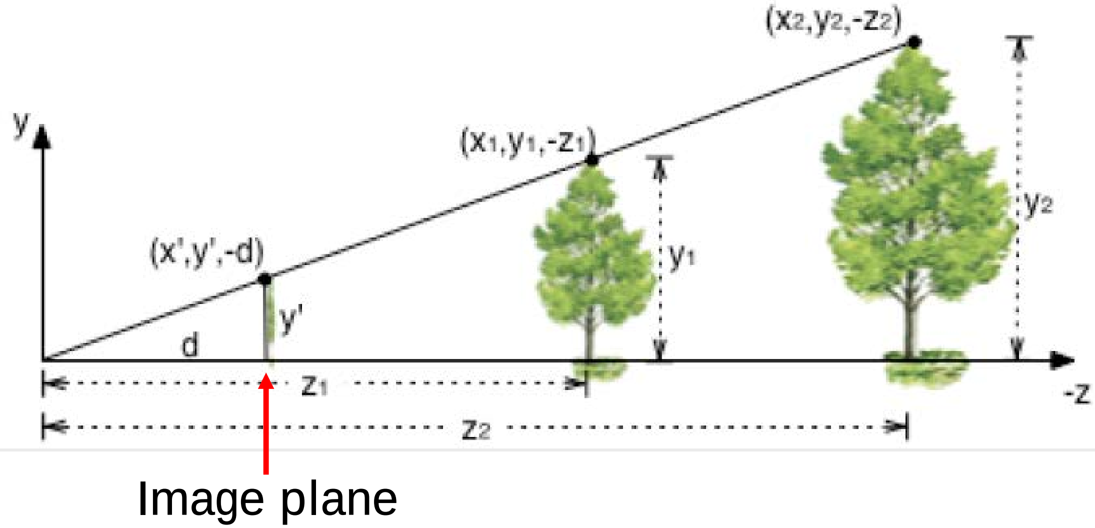
Perspective Projection

From law of ratios in similar triangles follows:

$$\frac{y'}{d} = \frac{y_1}{z_1} \rightarrow y' = \frac{y_1 d}{z_1}$$

Similarly: $$x' = \frac{x_1 d}{z_1}$$

By definition: $$z' = d$$



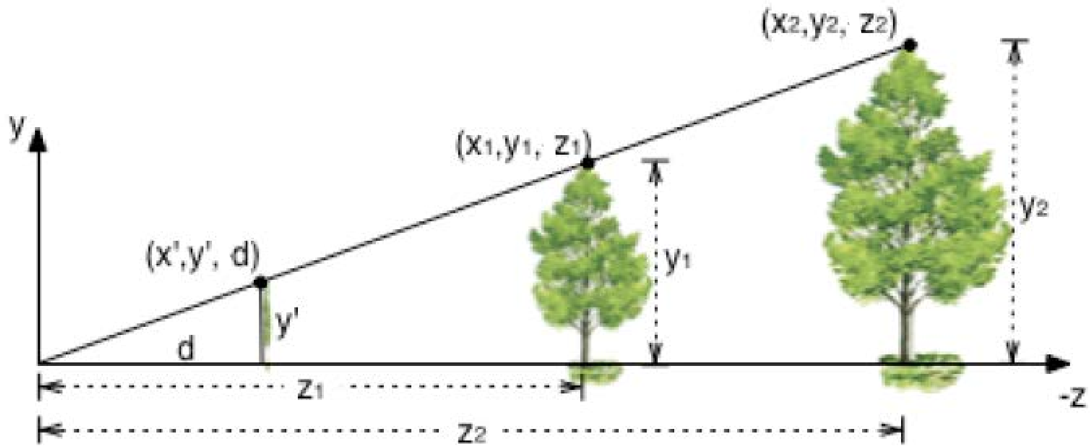
- We can express this using homogeneous coordinates and 4x4 matrices as follows

Perspective Projection

$$x' = \frac{x_1 d}{z_1}$$

$$y' = \frac{y_1 d}{z_1}$$

$$z' = d$$



$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \\ z/d \end{bmatrix} \Rightarrow \begin{bmatrix} xd/z \\ yd/z \\ d \\ 1 \end{bmatrix}$$

Projection matrix Homogeneous division

Perspective Projection

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \\ z/d \end{bmatrix} = \begin{bmatrix} xd/z \\ yd/z \\ d \\ 1 \end{bmatrix}$$

Projection matrix P

- Using projection matrix, homogeneous division seems more complicated than just multiplying all coordinates by d/z, so why do it?
- It will allow us to:
  - Handle different types of projections in a unified way
  - Define arbitrary view volumes

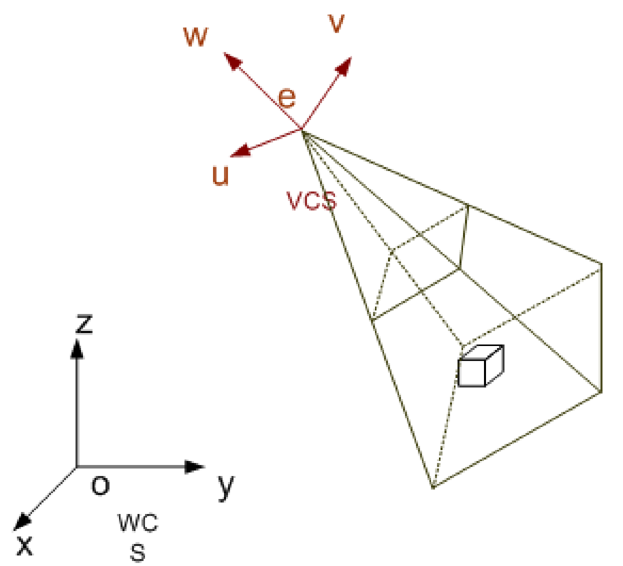
Topics

- View Volumes
- Vertex Transformation
- Rendering Pipeline
- Culling

View Volume

- View volume = 3D volume seen by camera

Camera coordinates



World coordinates

Projection Matrix

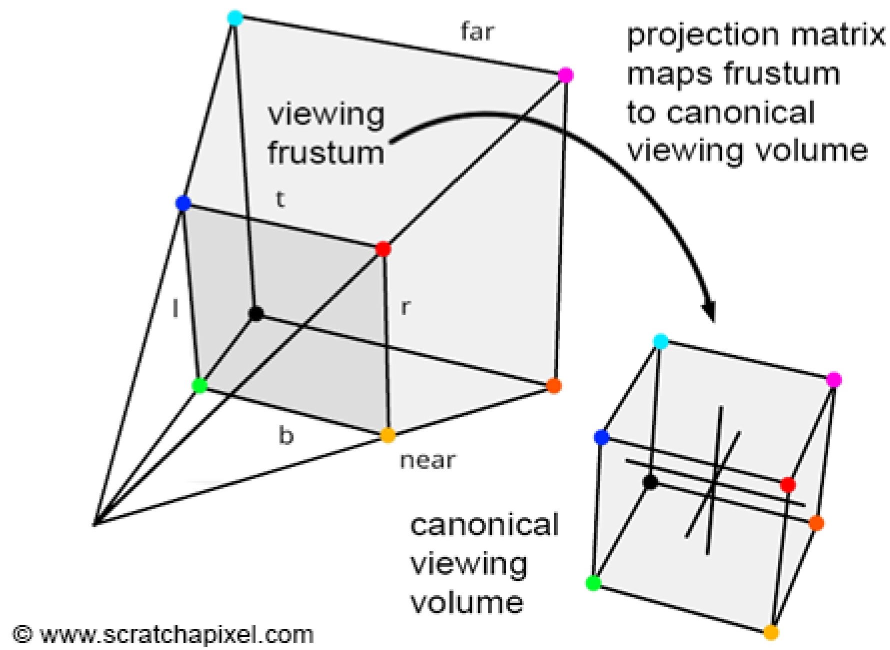
Camera coordinates

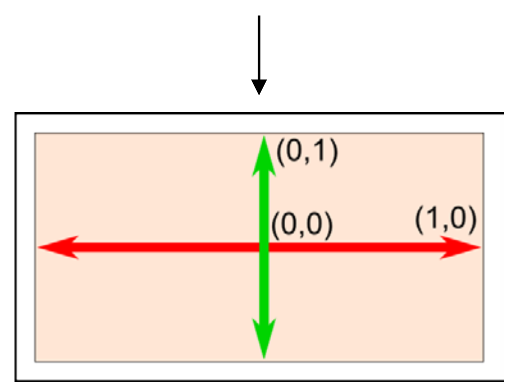
Projection matrix

Canonical view volume

Viewport transformation

Image space
(pixel coordinates)

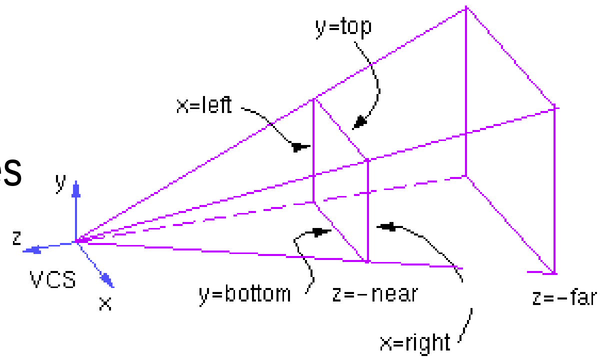


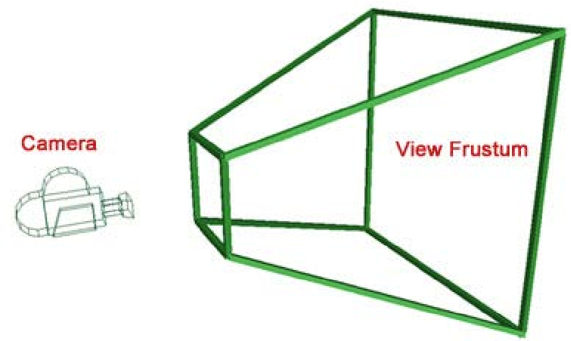


Perspective View Volume

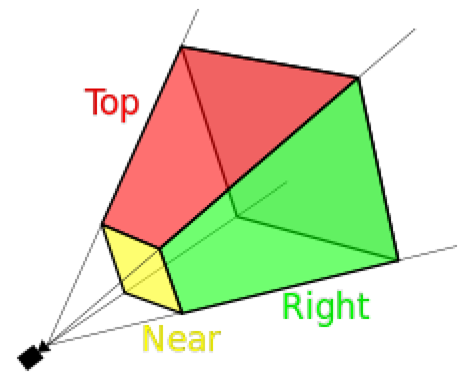
General view volume

Camera coordinates



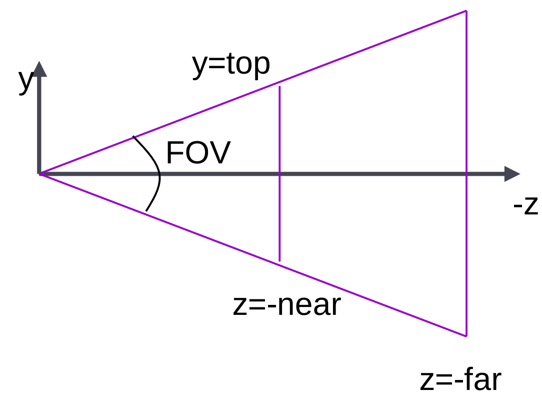


- Defined by 6 parameters, in camera coordinates
  - Left, right, top, bottom boundaries
  - Near, far clipping planes
- Clipping planes to avoid numerical problems
  - Divide by zero
  - Low precision for distant objects
- Usually symmetric, i.e., left=-right, top=-bottom



Perspective View Volume

Symmetrical view volume



- Only 4 parameters
  - Vertical field of view (FOV)
  - Image aspect ratio (width/height)
  - Near, far clipping planes

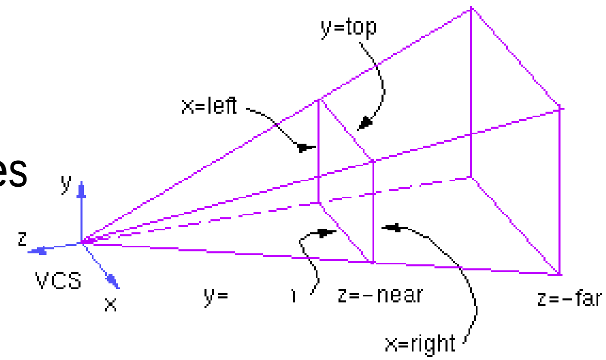
$$\text{aspect ratio} = \frac{right - left}{top - bottom} = \frac{right}{top}$$

$$\tan(FOV/2) = \frac{top}{near}$$

Perspective Projection Matrix

- General view frustum with 6 parameters

Camera coordinates

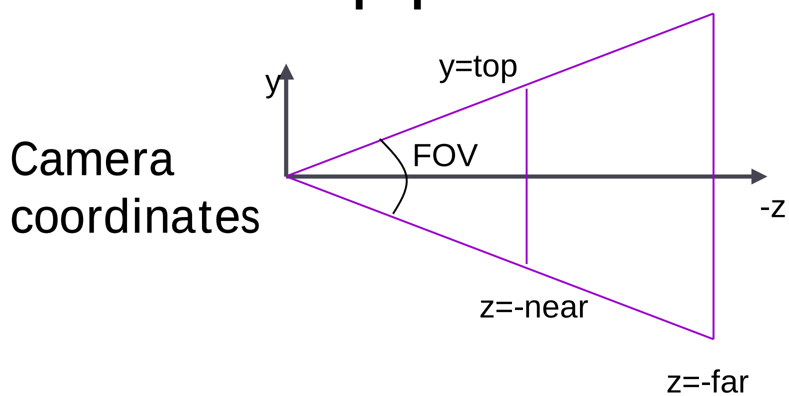


$$\mathbf{P}_{persp}(left, right, top, bottom, near, far) =$$

$$\begin{bmatrix} \frac{2near}{right-left} & 0 & \frac{right+left}{right-left} & 0 \\ 0 & \frac{2near}{top-bottom} & \frac{top+bottom}{top-bottom} & 0 \\ 0 & 0 & \frac{-(far+near)}{far-near} & \frac{-2far \cdot near}{far-near} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Perspective Projection Matrix

- Symmetrical view frustum with field of view, aspect ratio, near and far clip planes



$$\mathbf{P}_{persp}(FOV, aspect, near, far) = \begin{bmatrix} \frac{1}{aspect \cdot \tan(FOV/2)} & 0 & 0 & 0 \\ 0 & \frac{1}{\tan(FOV/2)} & 0 & 0 \\ 0 & 0 & \frac{near+far}{near-far} & \frac{2 \cdot near \cdot far}{near-far} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Canonical View Volume

- Goal: create projection matrix so that
  - User defined view volume is transformed into canonical view volume: cube $[-1,1]\text{x}[-1,1]\text{x}[-1,1]$
  - Multiplying corner vertices of view volume by projection matrix and performing homogeneous divide yields corners of canonical view volume
- Perspective and orthographic projection are treated the same way
- Canonical view volume is last stage in which coordinates are in 3D
  - Next step is projection to 2D frame buffer

Viewport Transformation

- After applying projection matrix, scene points are in *normalized viewing coordinates*
  - Per definition within range [-1..1] x [-1..1] x [-1..1]
- Next is projection from 3D to 2D (not reversible)
- Normalized viewing coordinates can be mapped to image (=pixel=frame buffer) coordinates
  - Range depends on window (view port) size: [x0…x1] x [y0…y1]
- Scale and translation required:

$$\mathbf{D}(x_0, x_1, y_0, y_1) = \begin{bmatrix} (x_1 - x_0)/2 & 0 & 0 & (x_0 + x_1)/2 \\ 0 & (y_1 - y_0)/2 & 0 & (y_0 + y_1)/2 \\ 0 & 0 & 1/2 & 1/2 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Lecture Overview

- View Volumes
- Vertex Transformation
- Rendering Pipeline
- Culling

Complete Vertex Transformation

- Mapping a 3D point in object coordinates to pixel coordinates:

$$\mathbf{p}' = \mathbf{D}\mathbf{P}\mathbf{C}^{-1}\mathbf{M}\mathbf{p}$$

Object space

  - **M**: Object-to-world matrix
  - **C**: camera matrix
  - **P**: projection matrix
  - **D**: viewport matrix

Complete Vertex Transformation

- Mapping a 3D point in object coordinates to pixel coordinates:

$$\mathbf{p}' = \mathbf{D}\mathbf{P}\mathbf{C}^{-1}\mathbf{M}\mathbf{p}$$

Object space

World space

  - **M**: Object-to-world matrix
  - **C**: camera matrix
  - **P**: projection matrix
  - **D**: viewport matrix

Complete Vertex Transformation

- Mapping a 3D point in object coordinates to pixel coordinates:

$$\mathbf{p}' = \mathbf{D}\mathbf{P}\mathbf{C}^{-1}\mathbf{M}\mathbf{p}$$

Object space

World space

Camera space

  - **M**: Object-to-world matrix
  - **C**: camera matrix
  - **P**: projection matrix
  - **D**: viewport matrix

Complete Vertex Transformation

- Mapping a 3D point in object coordinates to pixel coordinates:

$$\mathbf{p}' = \mathbf{D}\mathbf{P}\mathbf{C}^{-1}\mathbf{M}\mathbf{p}$$

Object space

World space

Camera space

Canonical view volume

  - **M**: Object-to-world matrix
  - **C**: camera matrix
  - **P**: projection matrix
  - **D**: viewport matrix

Complete Vertex Transformation

- Mapping a 3D point in object coordinates to pixel coordinates: $\mathbf{p}' = \mathbf{D}\mathbf{P}\mathbf{C}^{-1}\mathbf{M}\mathbf{p}$

Object space

World space

Camera space

Canonical view volume

Image space

  - **M**: Object-to-world matrix
  - **C**: camera matrix
  - **P**: projection matrix
  - **D**: viewport matrix

Complete Vertex Transformation

- Mapping a 3D point in object coordinates to pixel coordinates:

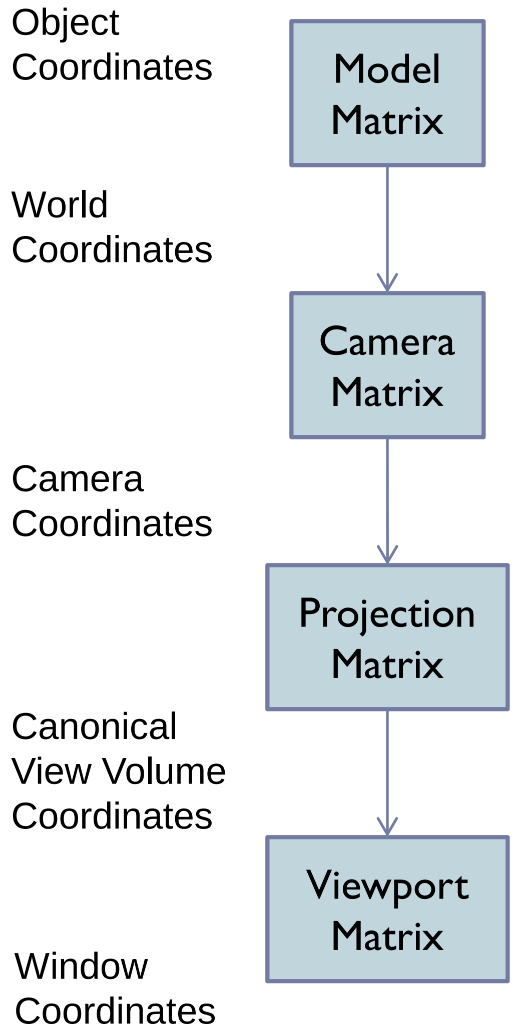
$$\mathbf{p}' = \mathbf{DPC}^{-1}\mathbf{Mp}$$

$$\mathbf{p}' = \begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix}$$

Pixel coordinates: $\begin{matrix} x'/w' \\ y'/w' \end{matrix}$

  - **M**: Object-to-world matrix
  - **C**: camera matrix
  - **P**: projection matrix
  - **D**: viewport matrix

The Complete Vertex Transformation



Complete Vertex Transformation in OpenGL

- Mapping a 3D point in object coordinates to pixel coordinates:

Projection matrix

$$\mathbf{p}' = \mathbf{D}\boxed{\mathbf{P}}\boxed{\mathbf{C}^{-1}\mathbf{M}}\mathbf{p}$$

  - **M**: Object-to-world matrix
  - **C**: camera matrix
  - **P**: projection matrix
  - **D**: viewport matrix

Complete Vertex Transformation in OpenGL

- ModelView matrix: $\mathbf{C}^{-1}\mathbf{M}$
  - Defined by the programmer.
  - Think of the ModelView matrix as where you stand with the camera and the direction you point it.
- Projection matrix: $\mathbf{P}$
  - Think of the projection matrix as describing the attributes of your camera, such as field of view, focal length, etc.
- Viewport, **D**
  - Specify via glViewport(x, y, width, height)

Vertex Shader Code

```
in vec4 vertexPosition;
// ...

uniform mat4 ModelView, Projection;

void main() {
    gl_Position = Projection * ModelView
* vertexPosition;
    // ...
}
```