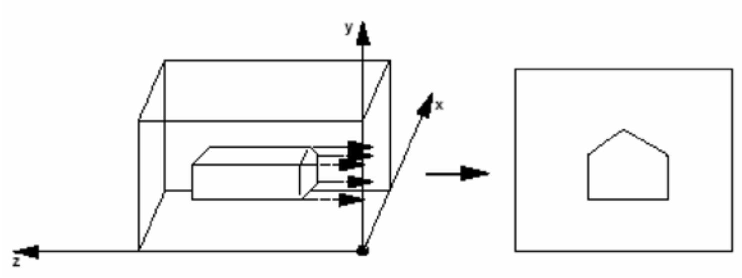
CSE 167:
Introduction to Computer Graphics
Lecture #7: Projection and Frustum Culling

Jürgen P. Schulze, Ph.D.
University of California, San Diego
Fall Quarter 2019

Projection

Projection

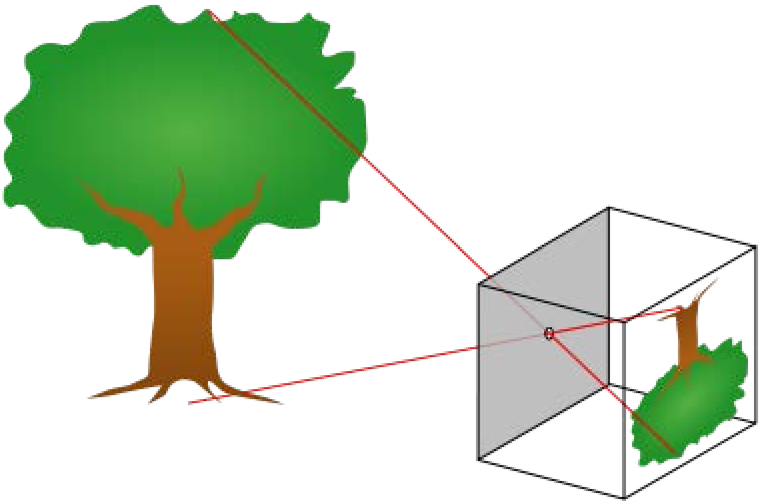
- Goal:
  Given 3D points (vertices) in camera coordinates, determine corresponding image coordinates
- Transforming 3D points into 2D is called Projection
- Typically one of two types of projection is used:
  - Orthographic Projection (=Parallel Projection)



  - Perspective Projection: most commonly used

Perspective Projection

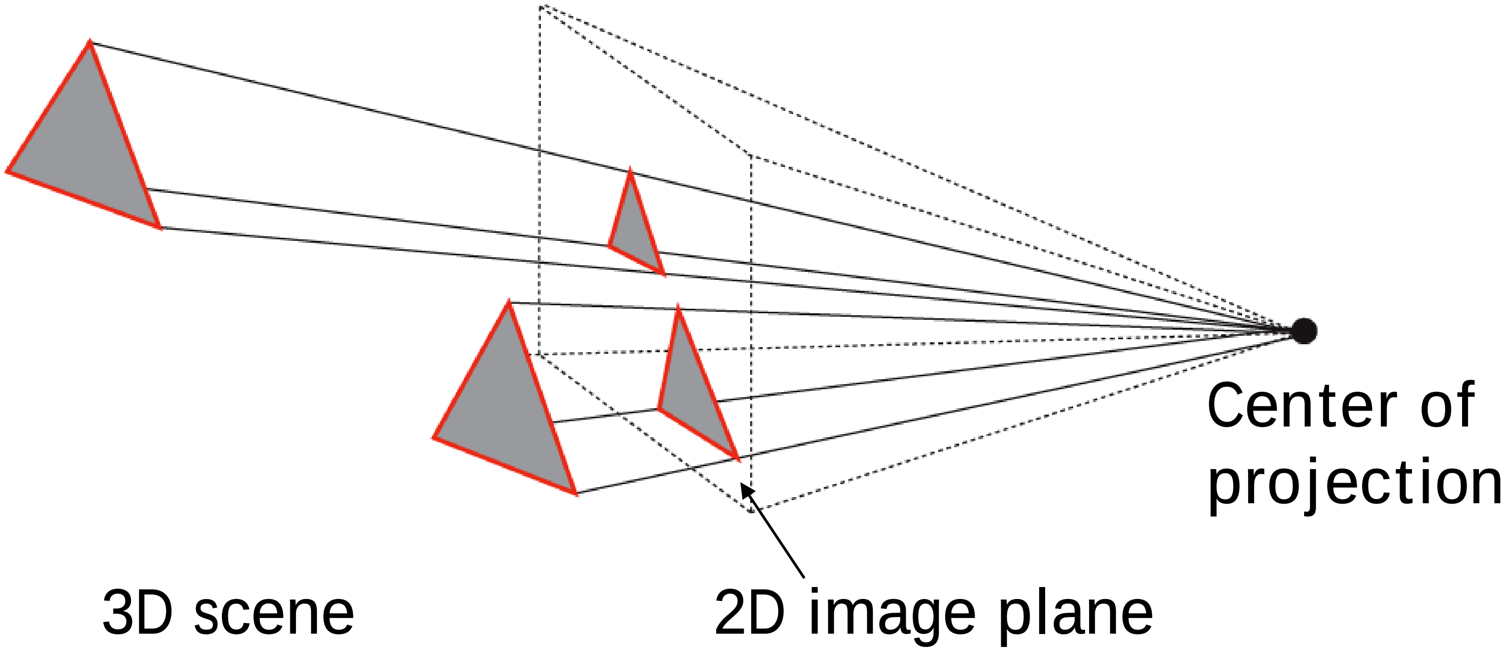
- Most common for computer graphics
- Simplified model of human eye, or camera lens (*pinhole camera*)



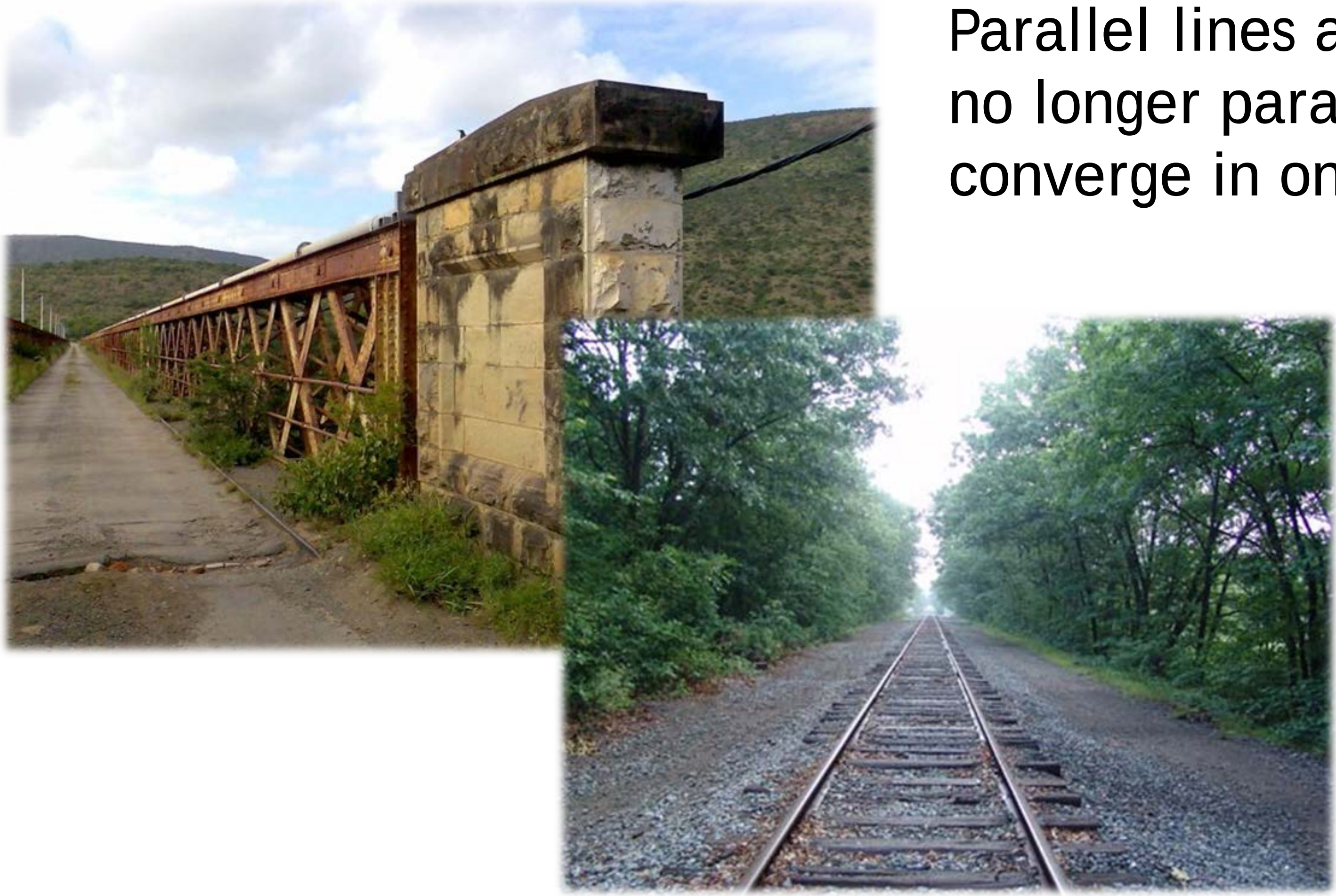
- Things farther away appear to be smaller
- Discovery attributed to Filippo Brunelleschi (Italian architect) in the early 1400's

Perspective Projection

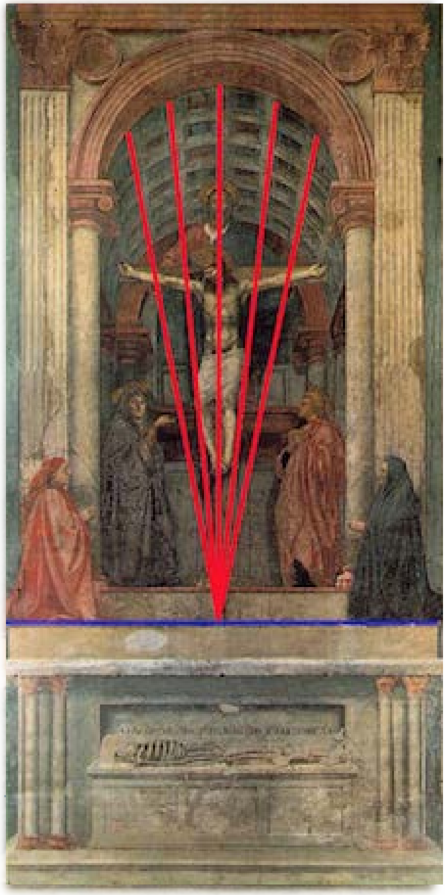
- Project along rays that converge in center of projection



Perspective Projection



Parallel lines are
no longer parallel,
converge in one point



Earliest example:
La Trinitá (1427) by Masaccio

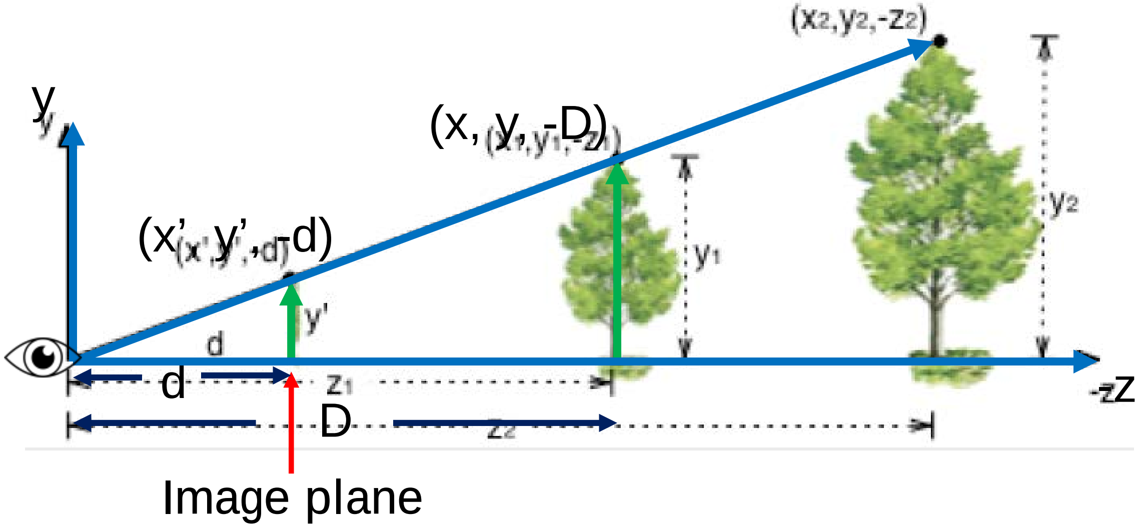
Perspective Projection

From law of ratios in similar triangles follows:

$$\frac{y'}{d} = \frac{y}{D} \rightarrow y' = \frac{yd}{D}$$

Similarly: $x' = \dfrac{xd}{D}$

By definition: $z' = -d$



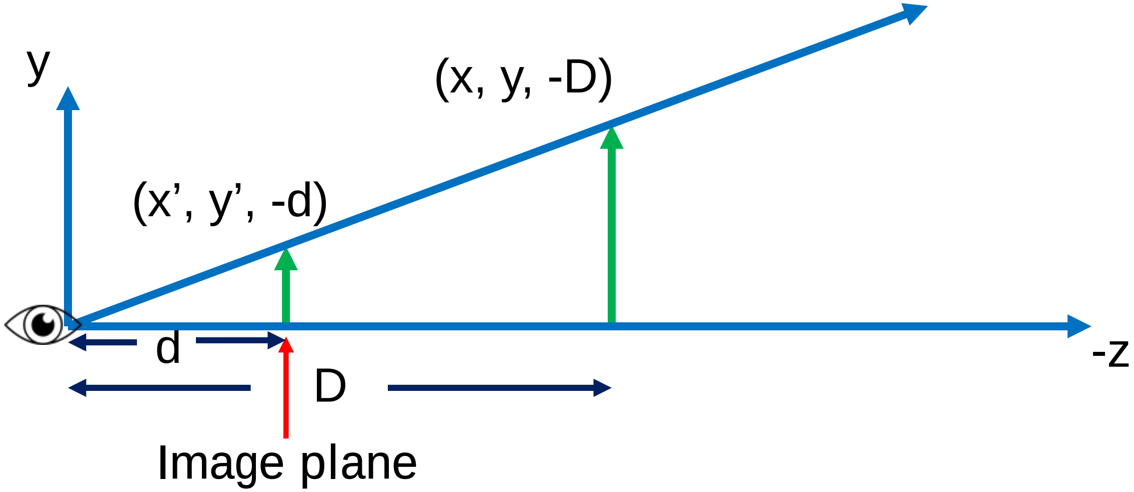
- We can express this using homogeneous coordinates and 4x4 matrices as follows

Perspective Projection

$$x' = \frac{xd}{D}$$

$$y' = \frac{yd}{D}$$

$$z' = -d$$



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

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

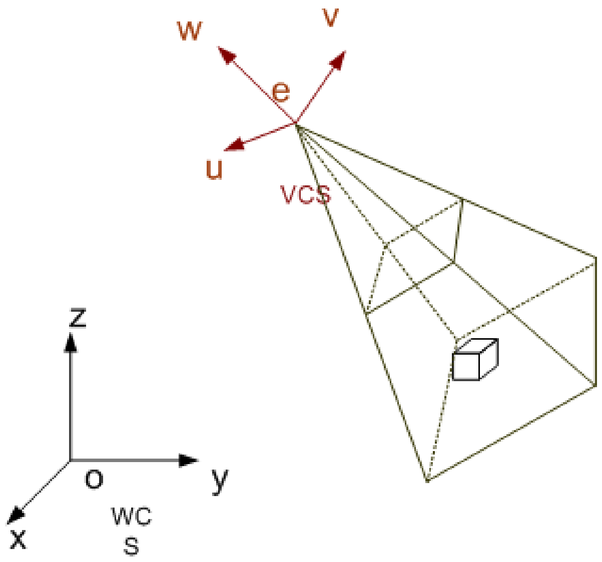
- Using projection matrix, homogeneous division seems more complicated than just multiplying all coordinates by - d/z, so why do it?
- It will allow us to:
  - Handle different types of projections in a unified way
  - Define arbitrary view volumes

Topics

- View Volumes
- Vertex Transformation
- Rendering Pipeline
- Culling

View Volume

- View volume = 3D volume seen by camera



Camera coordinates

World coordinates

Projection Matrix

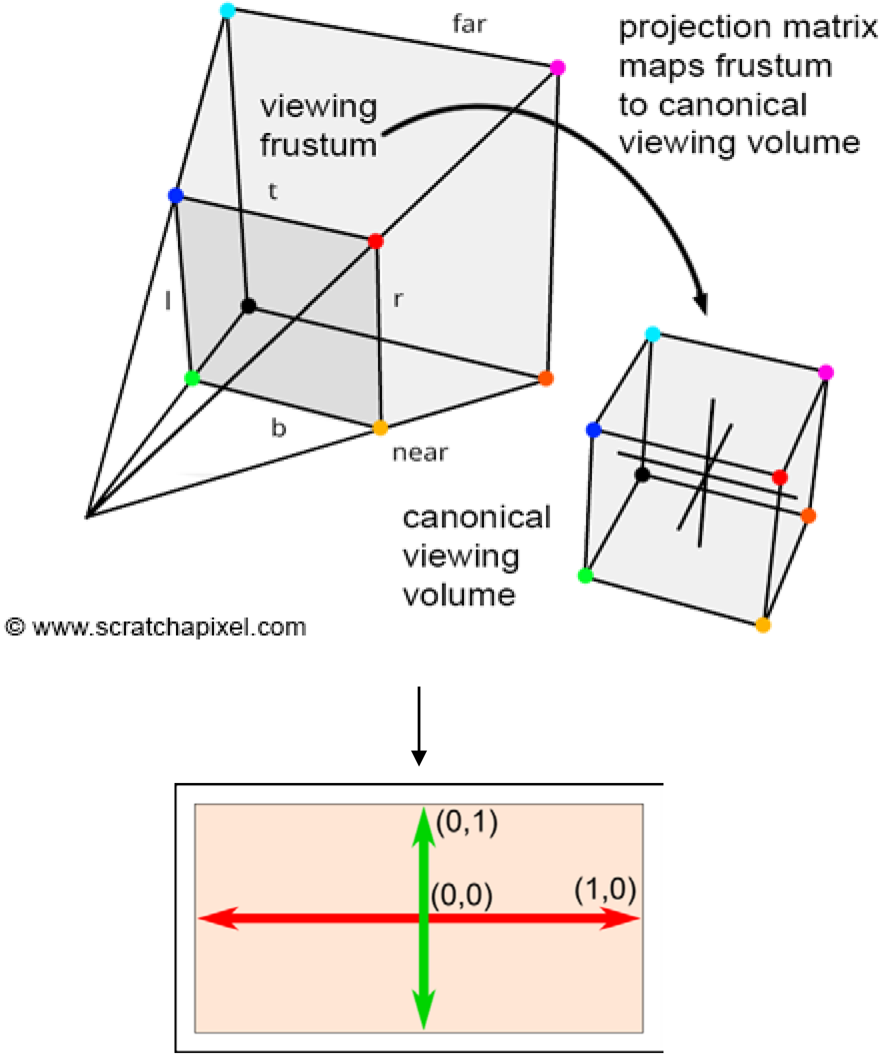
Camera coordinates

Projection matrix

Canonical view volume

Viewport transformation

Image space
(pixel coordinates)

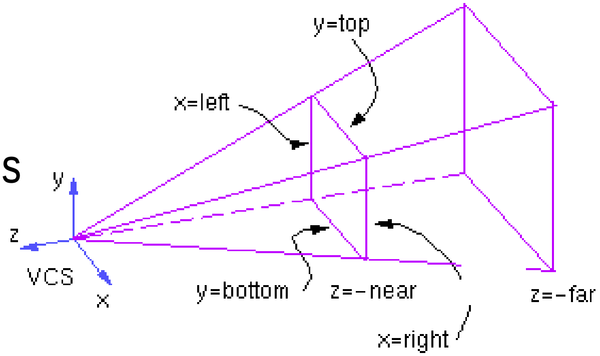


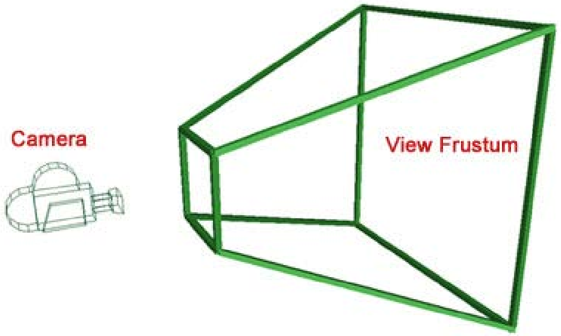
© www.scratchapixel.com

Perspective View Volume

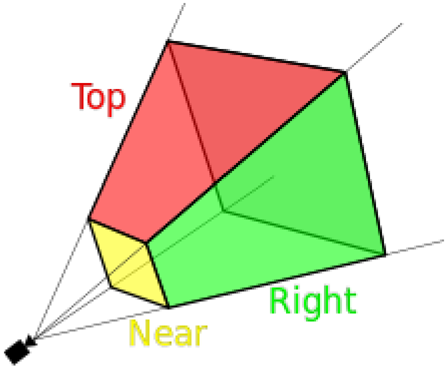
General view volume

Camera coordinates



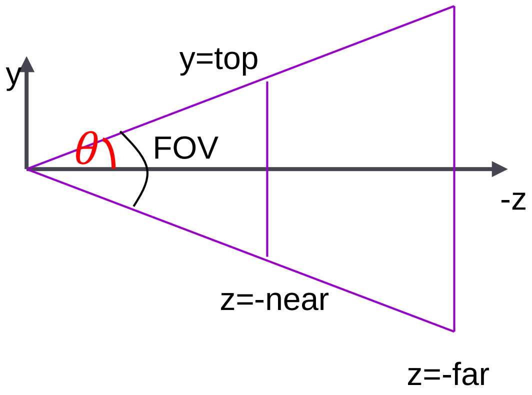


- Defined by 6 parameters, in camera coordinates
  - Left, right, top, bottom boundaries
  - Near, far clipping planes
- Clipping planes to avoid numerical problems
  - Divide by zero (multiplying all coordinates by - d/z)
  - Low precision for distant objects
- Usually symmetric, i.e., left=-right,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

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

$$\tan(\underbrace{\boxed{FOV/2}}_{\theta}) = \frac{top}{near}$$

Perspective View Volume

Rule of thumb to calculate projection matrix:

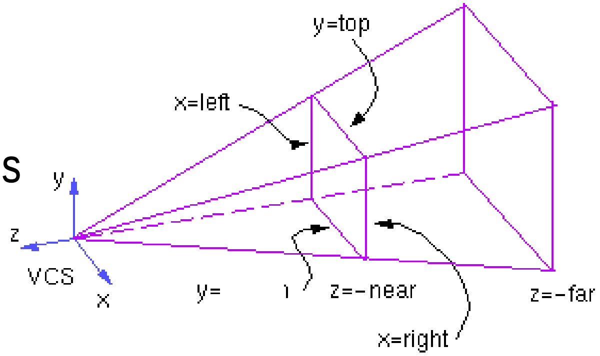
1. Convert the view-frustum to the simple symmetric projection frustum
2. Transform the simple frustum to the canonical view frustum

Ref: http://info.ee.surrey.ac.uk/Teaching/Courses/eem.cgi/lectures_pdf/lecture3.pdf

Perspective Projection Matrix

- General view frustum with 6 parameters

Camera coordinates

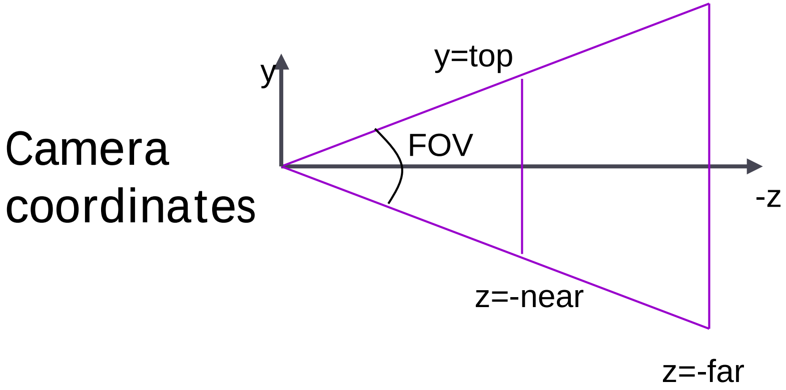


$$\mathbf{P}_{persp}(left, right, top, bottom, near, far) =$$

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

Perspective Projection Matrix

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



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

Ref: https://www.youtube.com/watch?v=ohksz3A00fk&t=79s

Projection Matrix

- How to determine if a matrix is projection matrix?

Canonical View Volume

- Goal: create projection matrix so that
  - User defined view volume is transformed into canonical view volume: cube [-1,1]x[-1,1]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

Canonical View Volume

- Summary so far in a demo

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:

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

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:

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

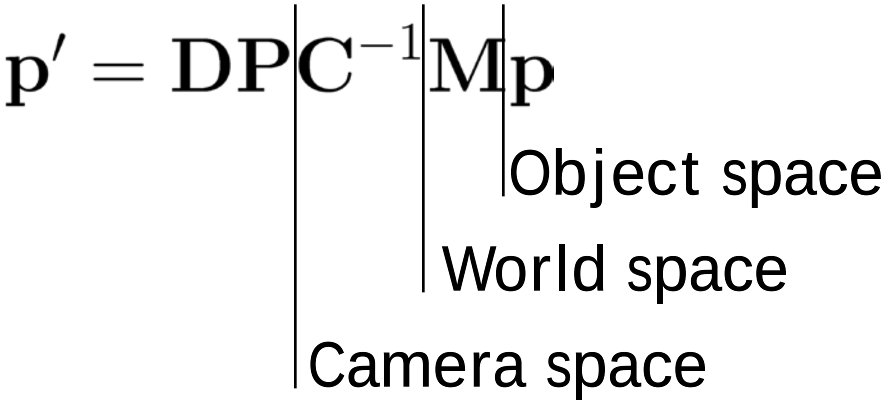
Object space

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:

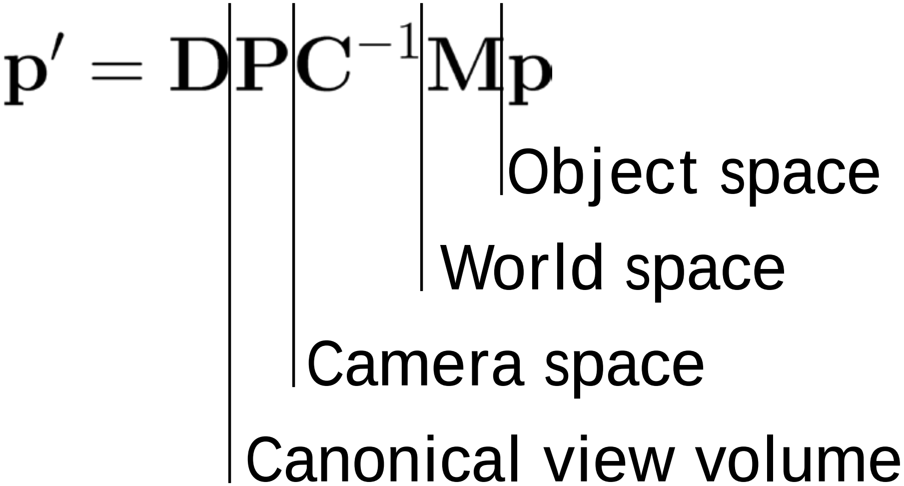


$$\mathbf{p}' = \mathbf{D}\mathbf{P}\mathbf{C}^{-1}\mathbf{M}\mathbf{p}$$

  - **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

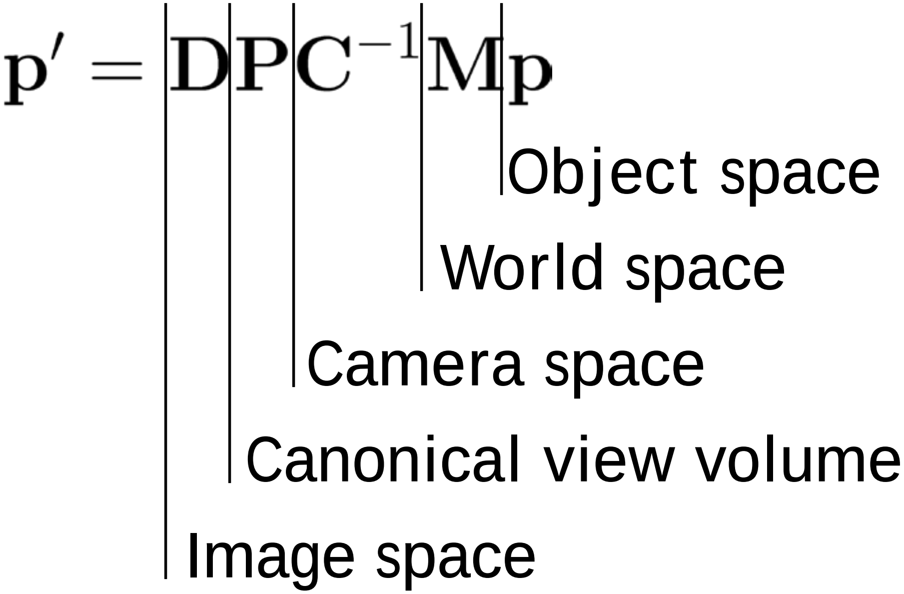
Camera space

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: x'/w' y'/w'

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

Complete Vertex Transformation in OpenGL

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

$$\mathbf{p}' = \mathbf{D}\underbrace{\mathbf{P}}_{\text{Projection matrix}}\overbrace{\mathbf{C}^{-1}\mathbf{M}}^{\text{ModelView matrix}}\mathbf{p}$$

  - **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: **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)

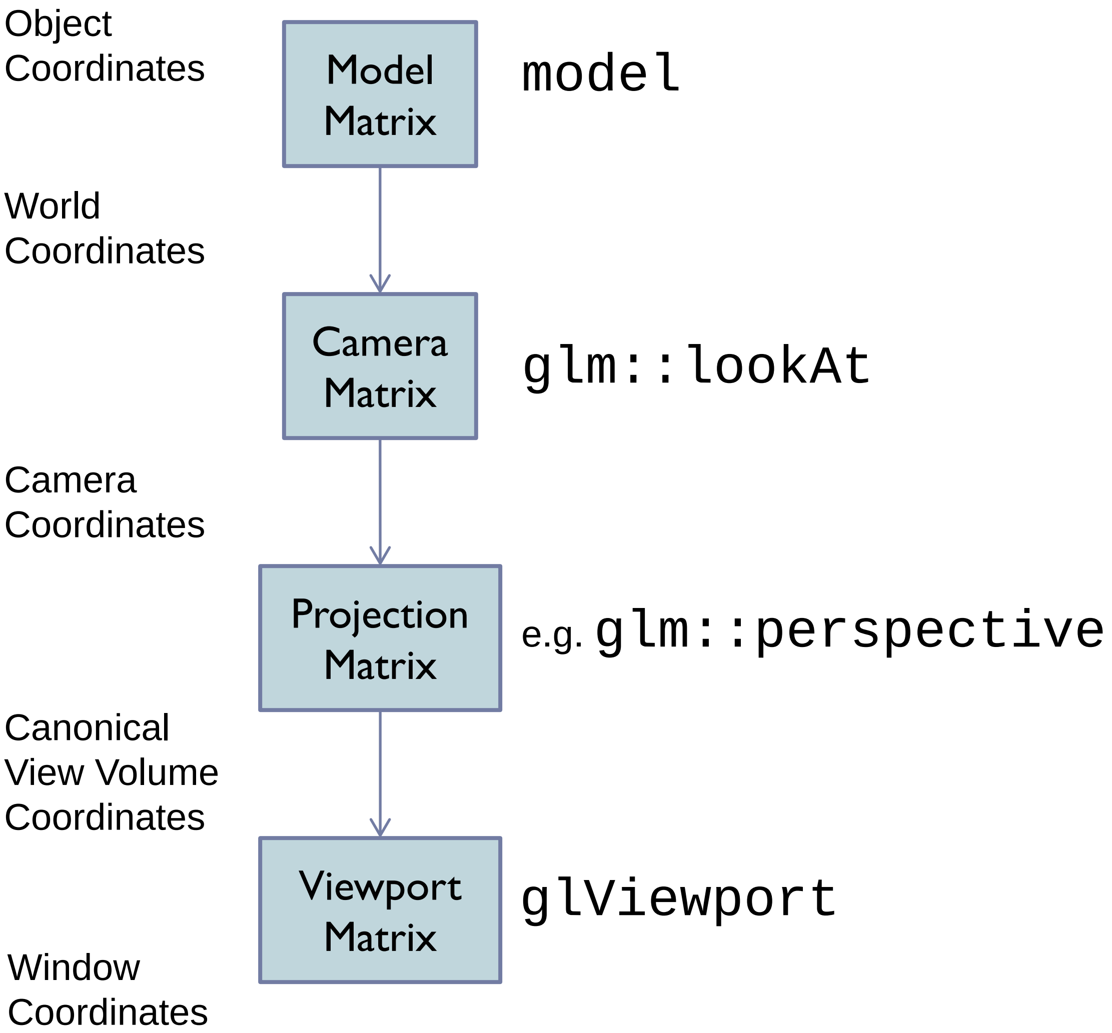
Vertex Shader Code

```
layout (location = 0) in vec3 position;
// ...

uniform mat4 projection;
uniform mat4 view;
uniform mat4 model;

void main() {
  gl_Position = projection * view *
model * vec4(position, 1.0);
    // ...
}
```

The Complete Vertex Transformation



Object
Coordinates

Model
Matrix

`model`

World
Coordinates

Camera
Matrix

`glm::lookAt`

Camera
Coordinates

Projection
Matrix

e.g. `glm::perspective`

Canonical
View Volume
Coordinates

Viewport
Matrix

`glViewport`

Window
Coordinates

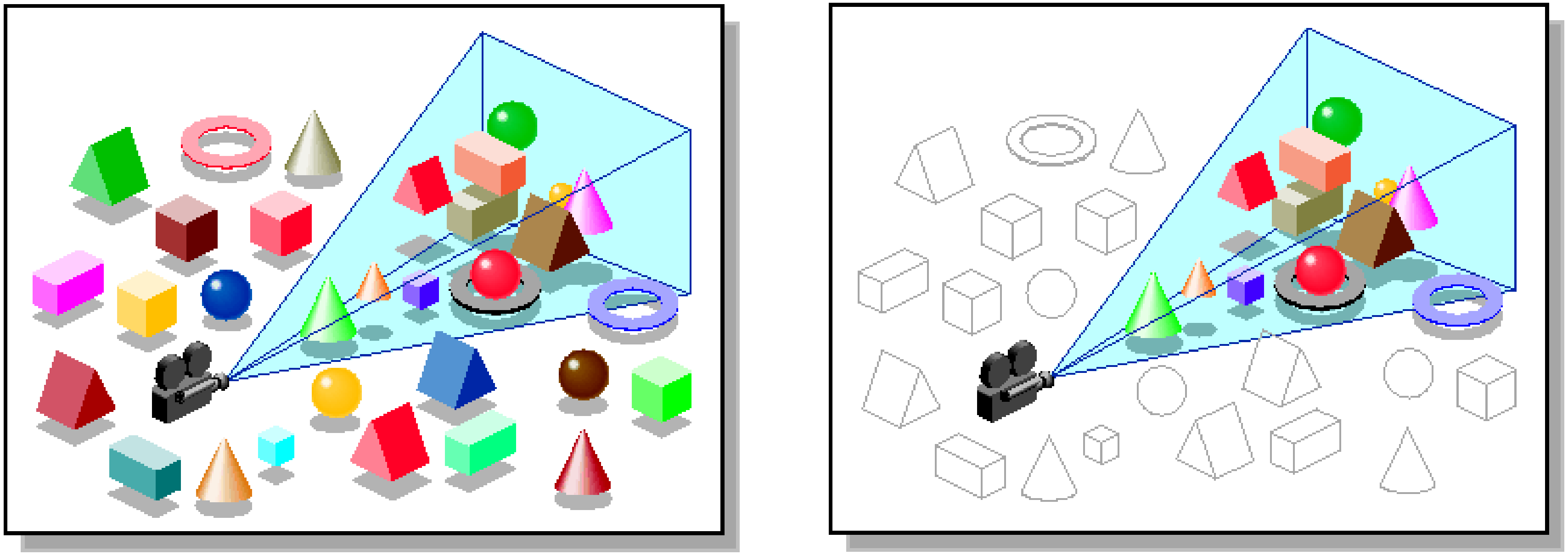
Visibility Culling

Visibility Culling

- Goal:
  Discard geometry that does not need to be drawn to speed up rendering
- Types of culling:
  - View frustum culling
  - Small object culling
  - Degenerate culling
  - Backface culling
  - Occlusion culling

View Frustum Culling

- Triangles outside of view frustum are off-screen



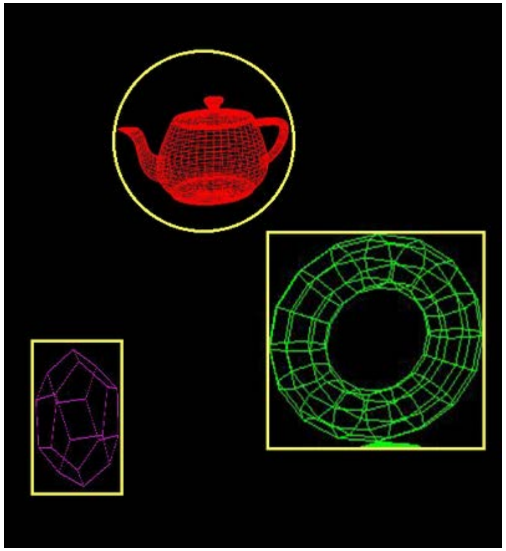
Images: SGI OpenGL Optimizer Programmer's Guide

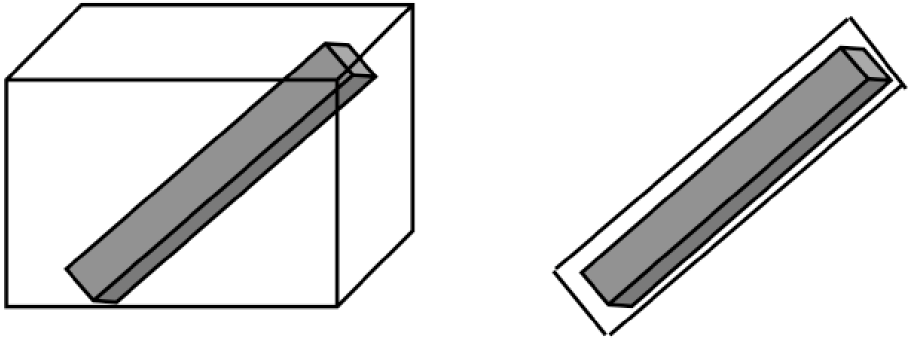
Videos

- Rendering Optimizations - Frustum Culling
  - http://www.youtube.com/watch?v=kvVHp9wMAO8
- View Frustum Culling Demo
  - http://www.youtube.com/watch?v=bJrYTBGpwic
- View Frustum Culling in Action
  - http://giant.gfycat.com/InexperiencedMadKiskadee.webm

Bounding Volumes

- Simple shape that completely encloses an object
- Generally a box or sphere
  - Easier to calculate culling for spheres
  - Easier to calculate tight fits for boxes
- Intersect bounding volume with view frustum instead of each primitive





Bounding Box

- How to cull objects consisting of may polygons?
- Cull bounding box
  - Rectangular box, parallel to object space coordinate planes
  - Box is smallest box containing the entire object

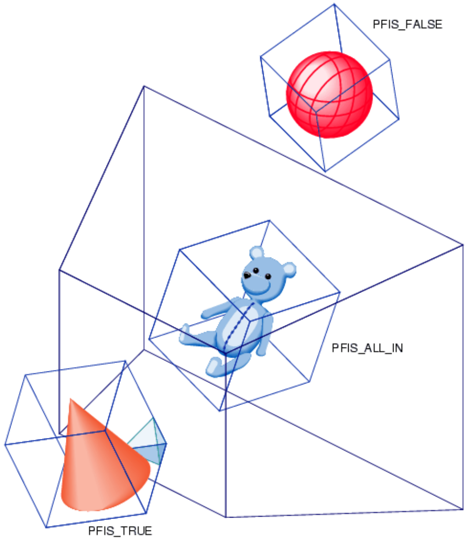
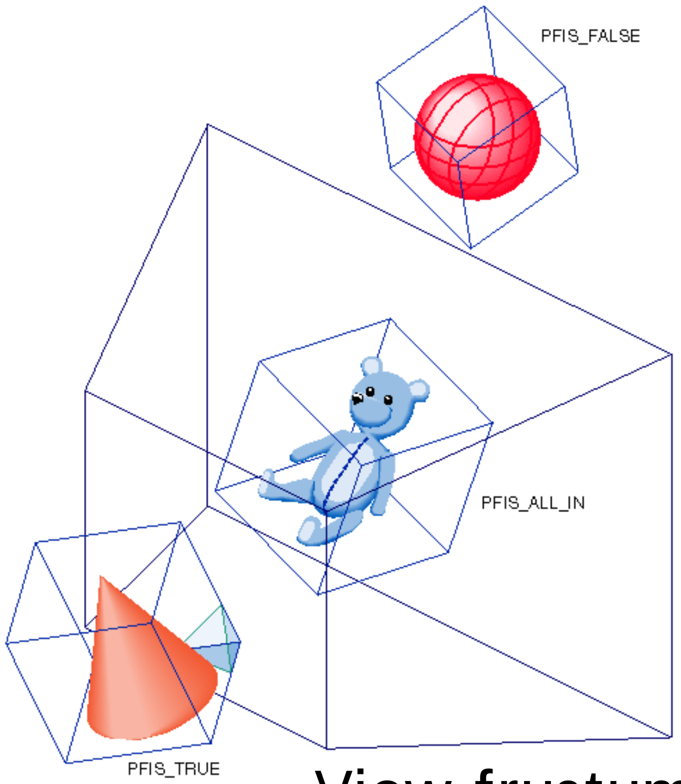


Image: SGI OpenGL Optimizer Programmer's Guide

View Frustum Culling

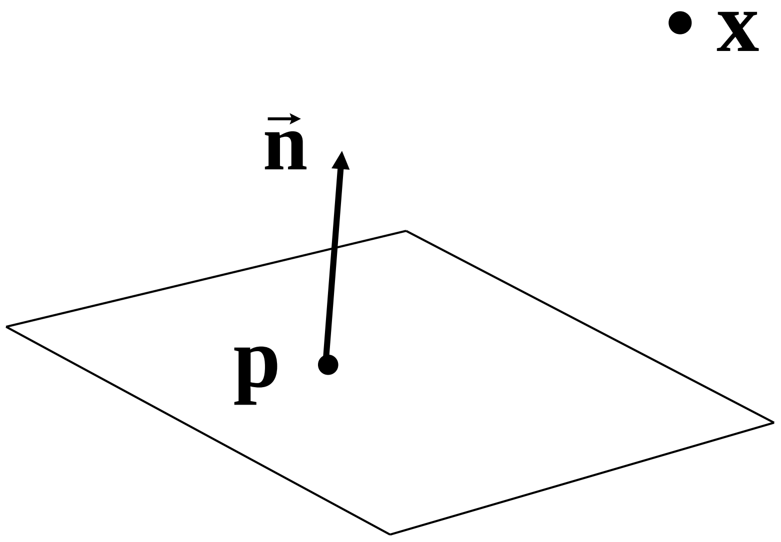
- Frustum defined by 6 planes
- Each plane divides space into "outside", "inside"
- Check each object against each plane
  - Outside, inside, intersecting
- If "outside" of at least one plane
  - Outside the frustum
- If "inside" all planes
  - Inside the frustum
- Else partly inside and partly out



View frustum

Distance to Plane

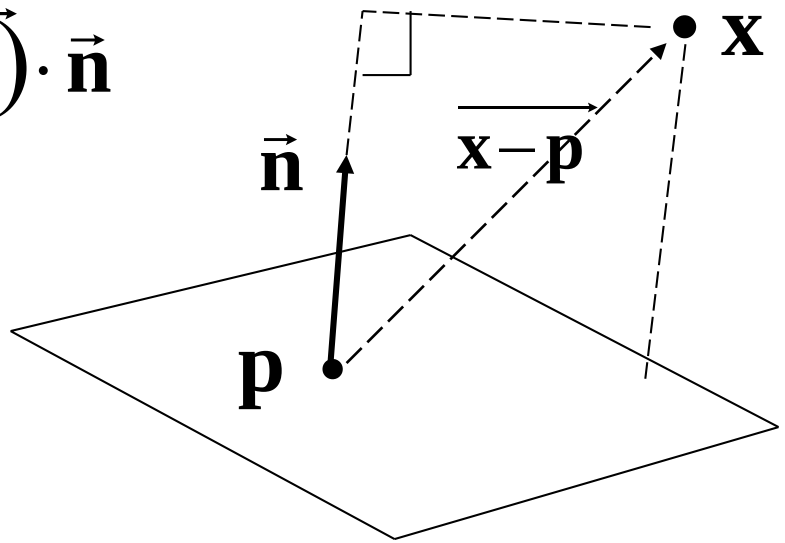
- A plane is described by a point **p** on the plane and a unit normal **n**
- Find the (perpendicular) distance from point **x** to the plane



Distance to Plane

- The distance is the length of the projection of **x-p** onto **n**

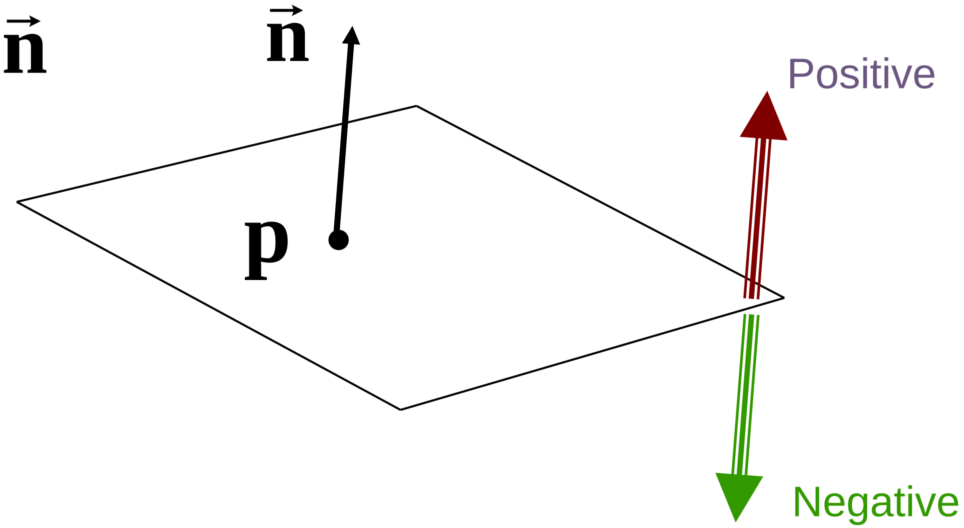
$$dist = \overrightarrow{(\mathbf{x} - \mathbf{p})} \cdot \vec{\mathbf{n}}$$



Distance to Plane

- The distance has a sign
  - positive on the side of the plane the normal points to
  - negative on the opposite side
  - zero exactly on the plane
- Divides 3D space into two infinite half-spaces

$$dist(\mathbf{x}) = \overrightarrow{(\mathbf{x} - \mathbf{p})} \cdot \vec{\mathbf{n}}$$



Distance to Plane

- Simplification

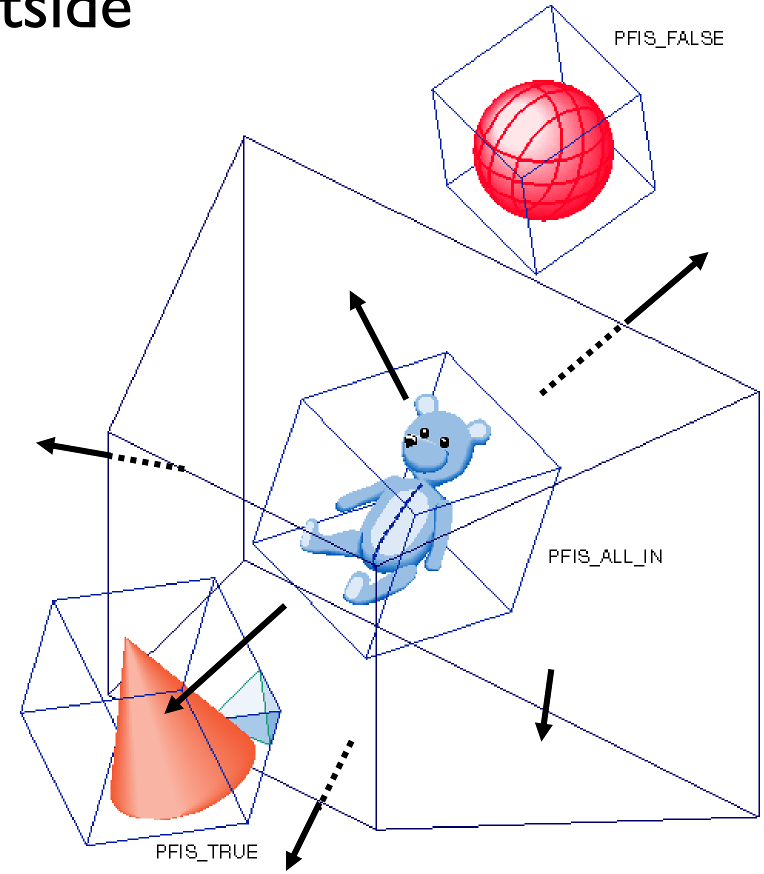
$$\begin{aligned} dist(\mathbf{x}) &= (\mathbf{x} - \mathbf{p}) \cdot \mathbf{n} \\ &= \mathbf{x} \cdot \mathbf{n} - \mathbf{p} \cdot \mathbf{n} \end{aligned}$$

$$dist(\mathbf{x}) = \mathbf{x} \cdot \mathbf{n} - d, \quad d = \mathbf{pn}$$

- *d* is independent of **x**
- *d* is distance from the origin to the plane
- We can represent a plane with just *d* and **n**

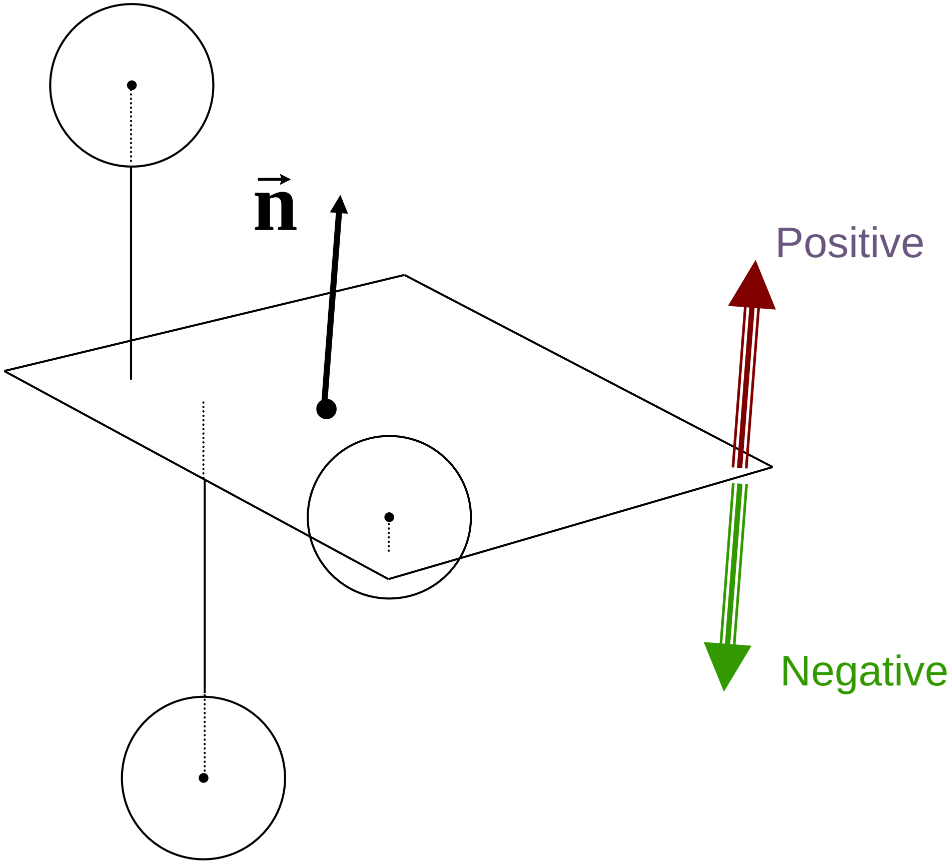
Frustum With Signed Planes

- Normal of each plane points outside
  - “outside” means positive distance
  - “inside” means negative distance



Test Sphere and Plane

- For sphere with radius r and origin $\mathbf{x}$, test the distance to the origin, and see if it is beyond the radius
- Three cases:
  - $dist(\mathbf{x}) > r$
    - completely above
  - $dist(\mathbf{x}) < -r$
    - completely below
  - $-r < dist(\mathbf{x}) < r$
    - intersects

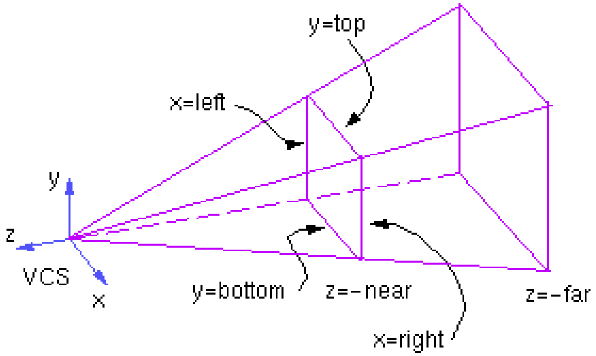


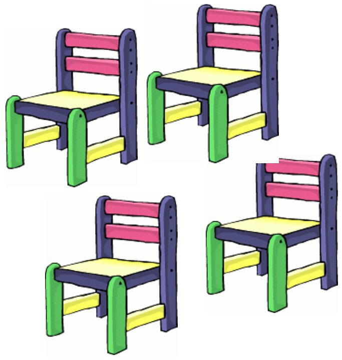
Culling Summary

- Transform view frustum plane equations in camera space.
- Pre-compute the normal $\mathbf{n}$ and value d for each of the six planes.
- Given a sphere with center $\mathbf{x}$ and radius r in camera space.
- For each plane:
  - if $dist(\mathbf{x}) > r$: sphere is outside! (no need to continue loop)
  - add 1 to count if $dist(\mathbf{x}) < -r$
- If we made it through the loop, check the count:
  - if the count is 6, the sphere is completely inside
  - otherwise the sphere intersects the frustum
  - *(can use a flag instead of a count)*

Culling Groups of Objects

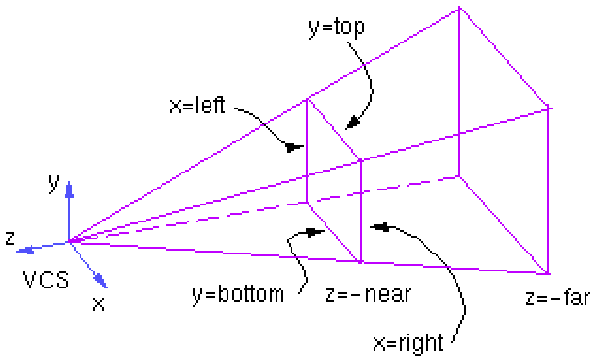
- Want to be able to cull the whole group quickly
- But if the group is partly in and partly out, want to be able to cull individual objects

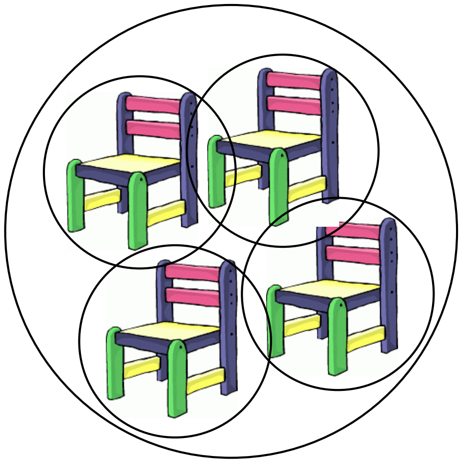




Hierarchical Bounding Volumes

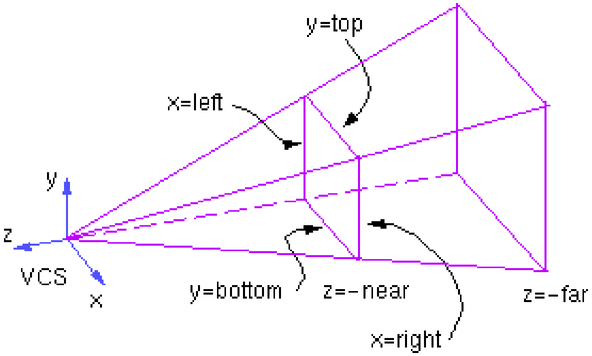
- Given hierarchy of objects
- Bounding volume of each node encloses the bounding volumes of all its children
- Start by testing the outermost bounding volume
  - If it is entirely outside, don’t draw the group at all
  - If it is entirely inside, draw the whole group

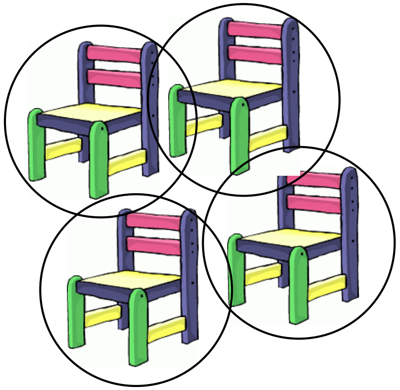




Hierarchical Culling

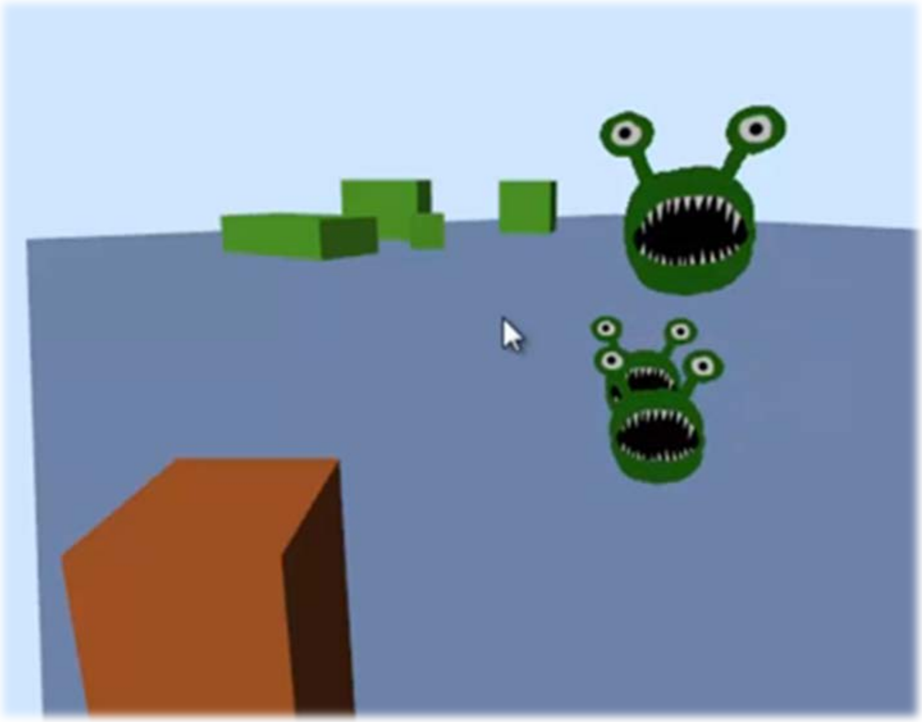
- If the bounding volume is partly inside and partly outside
  - Test each child's bounding volume individually
  - If the child is in, draw it; if it's out cull it; if it's partly in and partly out, recurse.
  - If recursion reaches a leaf node, draw it normally





Video

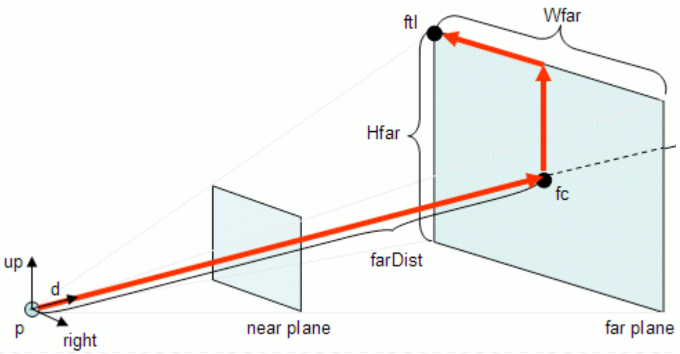
- Math for Game Developers - Frustum Culling
  - http://www.youtube.com/watch?v=4p-E_31XOPM



Find the frustum planes

- p – the camera position
- d – a vector with the direction of the camera’s view ray. In here it is assumed that this vector has been normalized
- Wnear – the “width” of the near plane
- nearDist – the distance from the camera to the near plane
- farDist – the distance from the camera to the far plane
- up – the up vector obtained by normalizing (ux, uy, uz) from the last parameters of gluLookAt
- right – the right vector obtained by cross product between up and d.

```
nc = p + d * nearDist
fc = p + d * farDist
```



Find the frustum planes

- near plane: d as normal, nc as a point on the plane.
- far plane: –d as normal, fc as a point on the plane.
- right plane: p as a point on the plane. normal can be found in this tutorial, the pseudocode is copied here.

```
nc = p + d * nearDist
a = (nc + right * Wnear / 2) – p
a.normalize()
normalRight = up × a
```