UNIVERSITY OF MINNESOTA
Driven to Discover®

Projection and Camera Models

CSCI 4611: Programming Interactive Computer Graphics and Games

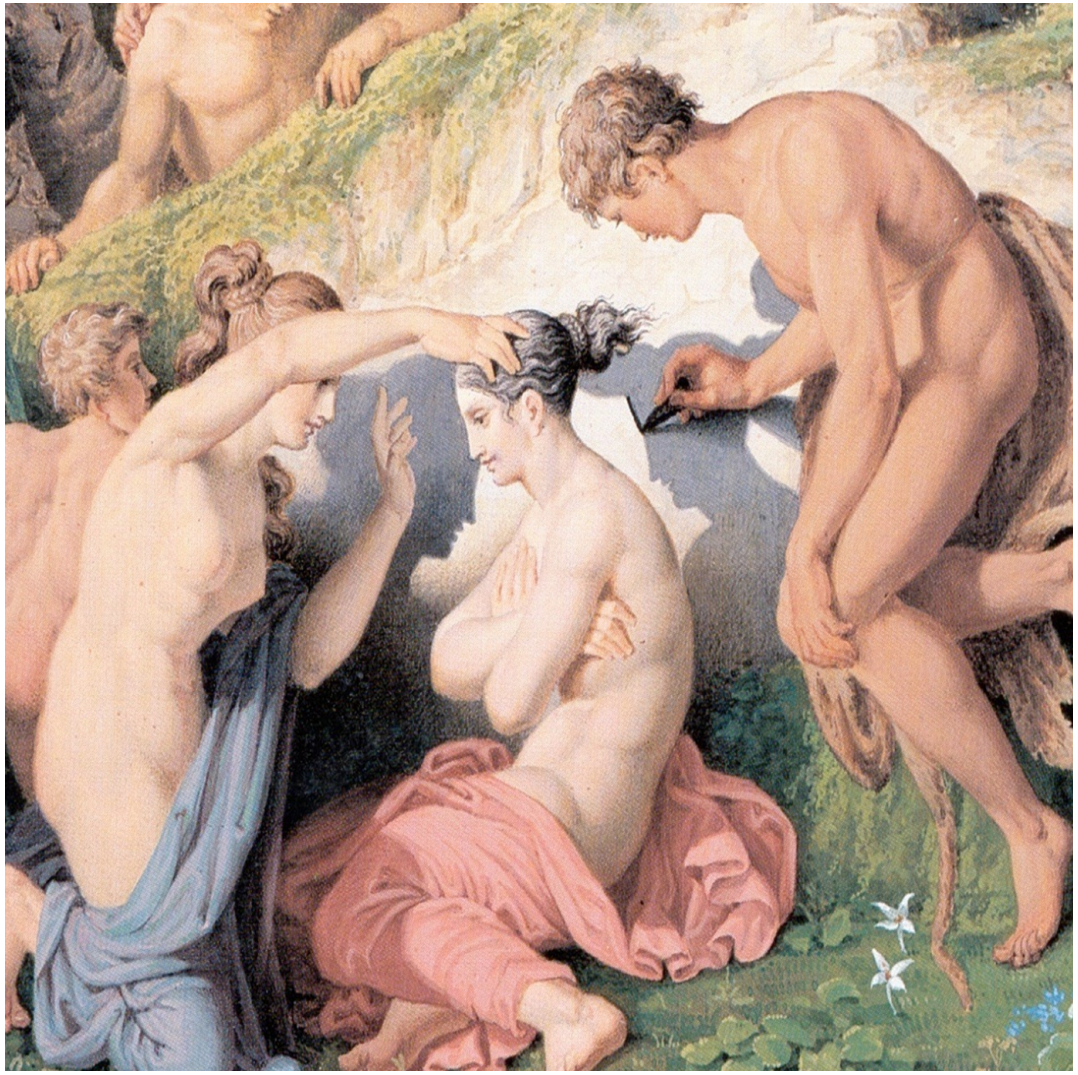
Evan Suma Rosenberg | CSCI 4611 | Fall 2022

Brief History of Projection

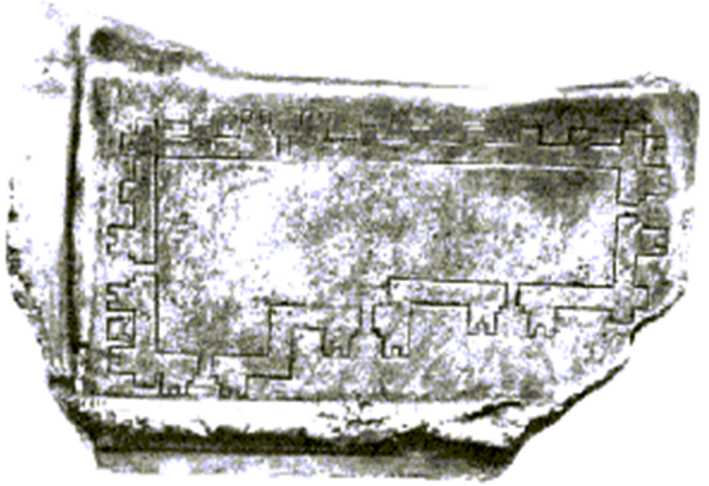
Drawing as Projection

The Invention of Drawing
Karl Friedrich Schinkle, 1830

Painting based on mythical tale told by Pliny the Elder about a Corinthian man tracing the shadow of departing lover.



Early Examples of Projection



Plan view (orthographic projection) from Mesopotamia, 2150 BC, earliest known technical drawing in existence.

Greek vases from late 6th century BC show perspective(!)

Roman architect Vitruvius wrote specifications of a plan with architectural illustrations, *De Architectura* (rediscovered in 1414). The original illustrations for these writings have been lost.

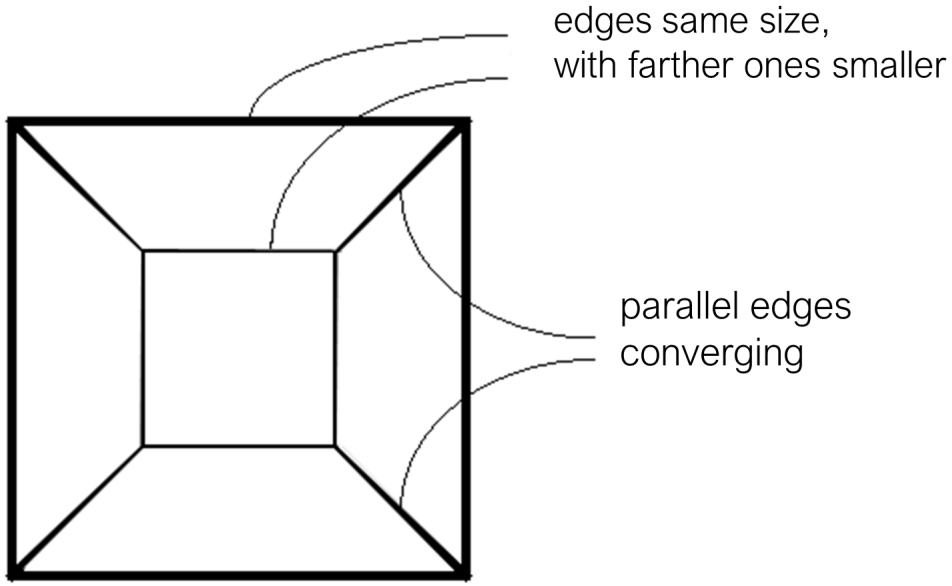
Theseus Killing the Minotaur by the Kleophrades Painter

Most Striking Features of Linear Perspective

Parallel lines converge (in 1, 2, or 3 axes) to **vanishing point(s)**

Objects farther away are more **foreshortened** (i.e., smaller) than closer ones

Example: perspective cube



Early Perspective

Methods of invoking three dimensional space:
shading suggests rounded, volumetric forms
converging lines suggest spatial depth of room

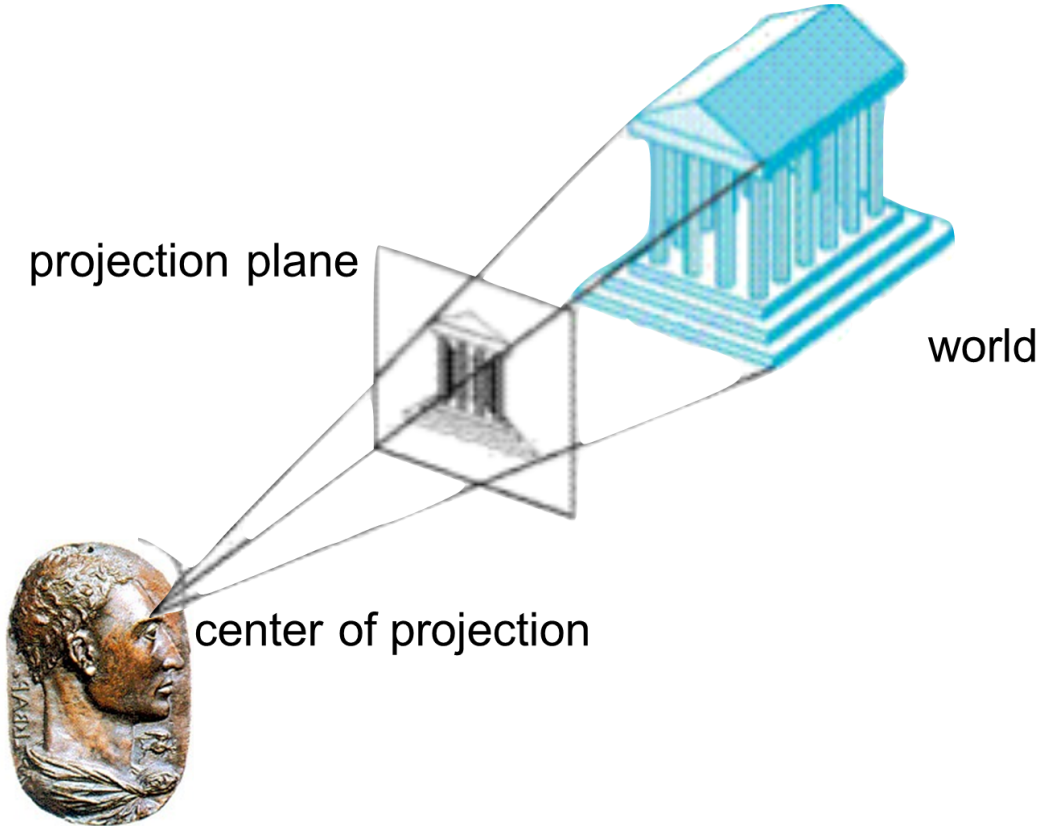
Not systematic. Lines do not converge to single vanishing point.

Giotto, **Franciscan Rule Approved**
Assisi, Upper Basilica, c.1295-1300

Leono Battista Alberti (1404-1472)

Published first treatise on perspective,
Della Pittura, in 1435.

"A painting [**the projection plane**] is the intersection of a visual pyramid [**view volume**] at a given distance, with a fixed center [**center of projection**] and a defined position of light, represented by art with lines and colors on a given surface [**the rendering**]."



Setting for “Invention” of Perspective Projection

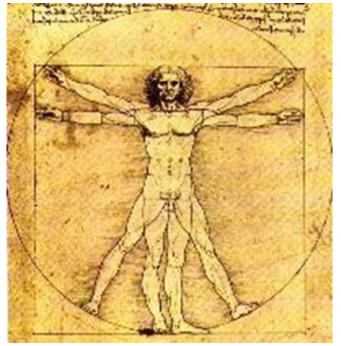
The Renaissance ushered in a new emphasis on importance of individual viewpoint and world interpretation, power of observation—particularly of nature (astronomy, anatomy, botany, etc.)

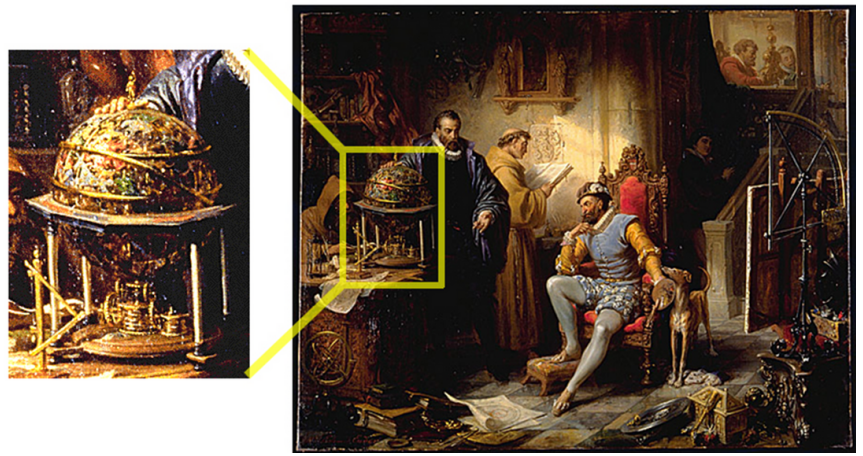
- Massaccio

- Donatello

- Leonardo

- Newton

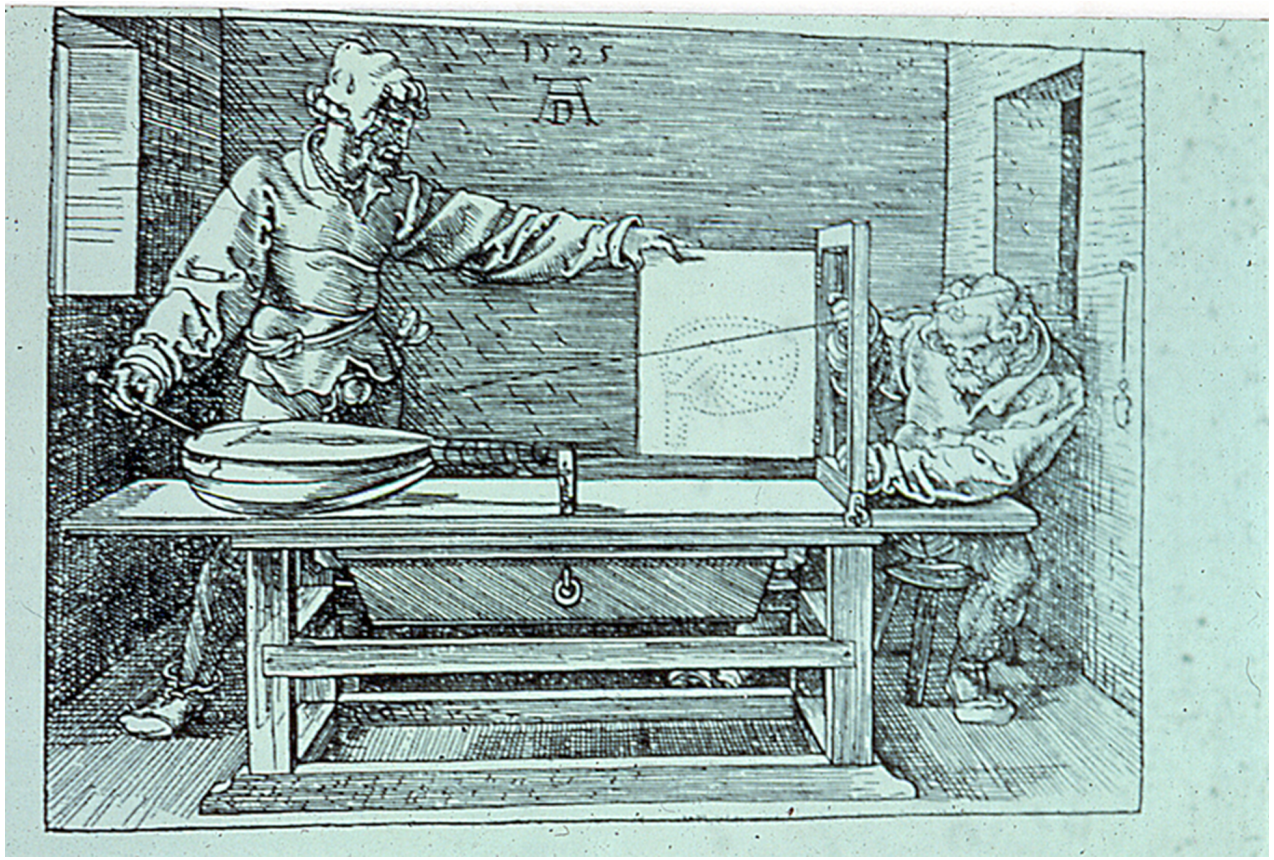




Ender, **Tycho Brahe and Rudolph II in Prague** (detail of clockwork), c. 1855

Albrecht Dürer (1471-1528)

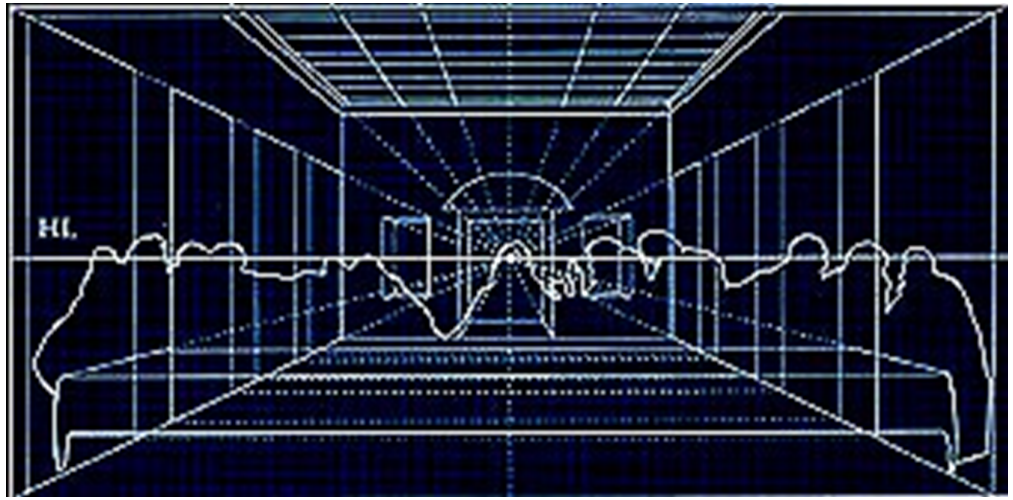
Concept of similar triangles described both geometrically and mechanically in widely read treatise by Albrecht Dürer.

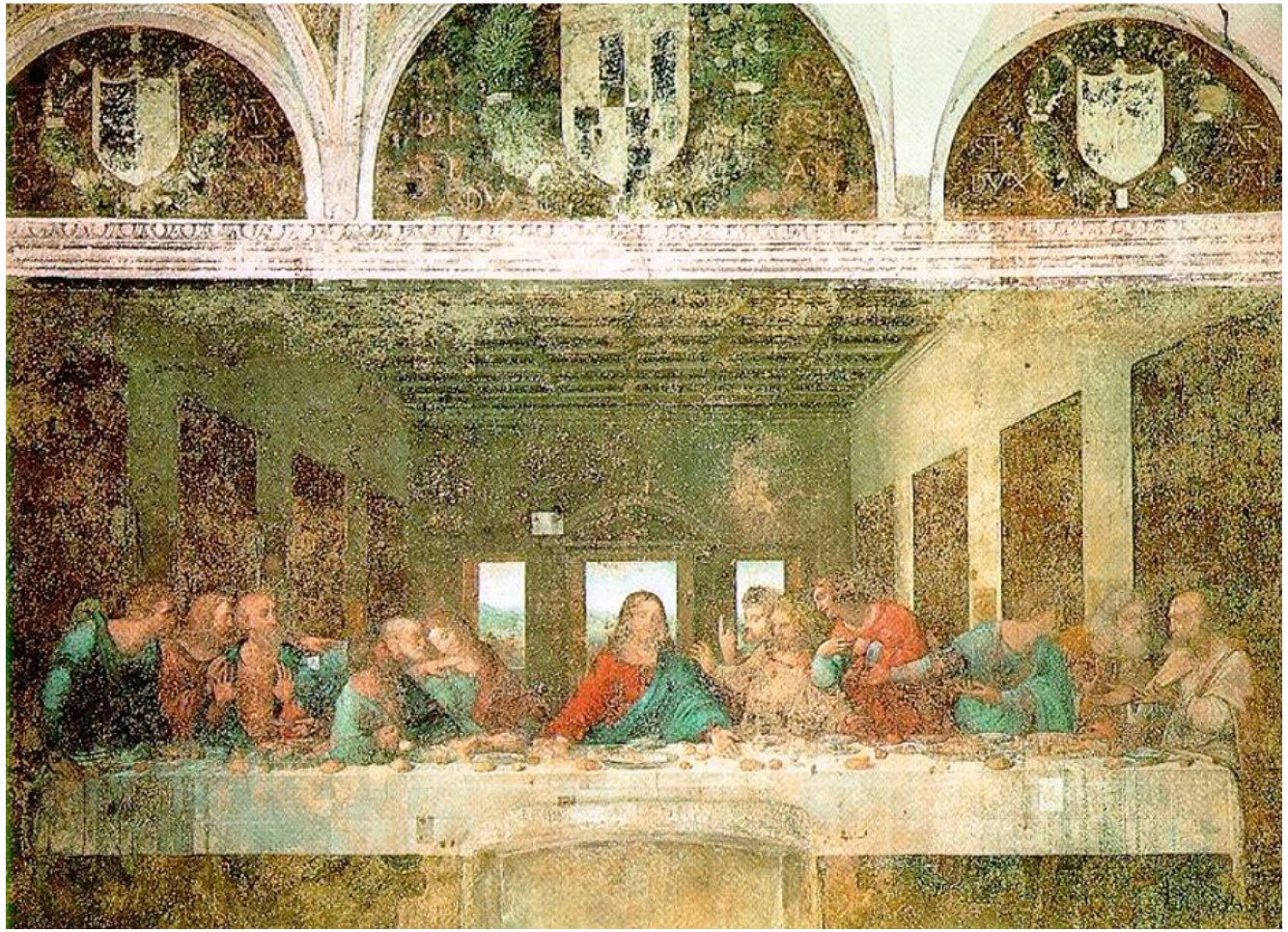


Albrecht Dürer, **Artist Drawing a Lute**

Woodcut from Dürer's work about the Art of Measurement.
Underweysung der messung, Nurenberg, 1525

Leonardo da Vinci, The Last Supper (1495)

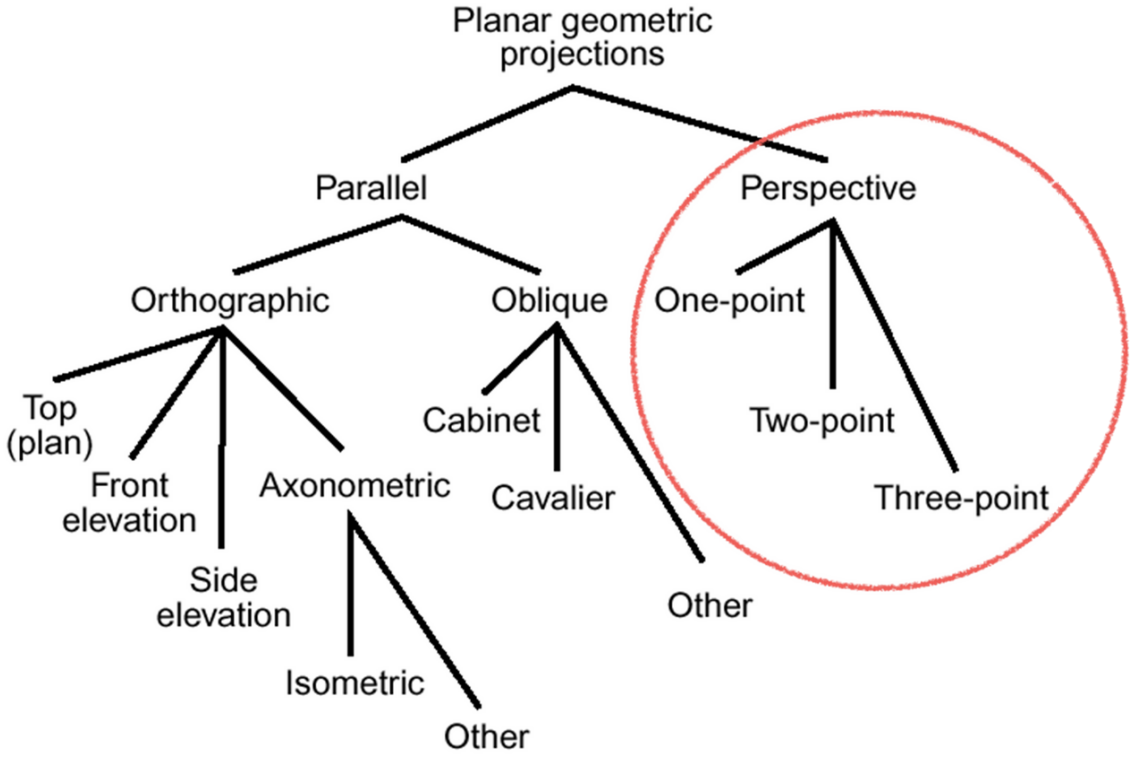




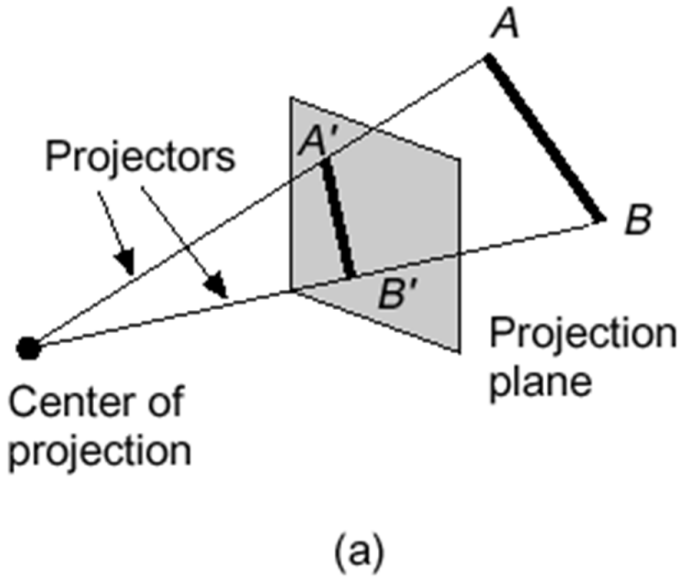
Types of Projections

Perspective projections imitate eyes or cameras and looks more natural. This is what we have seen so far.

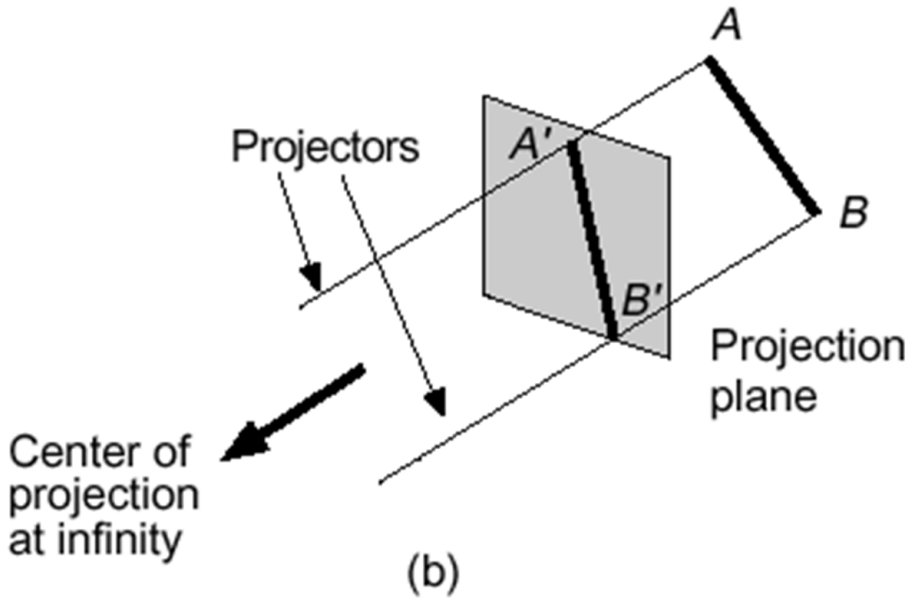
Parallel projections are also possible. They are useful in engineering and architecture because they can be used for measurements.



Main Classes of Planar Geometrical Projections



(a)



(b)

Perspective Projection: determined by **Center of Projection** (COP).

Parallel Projection: determined by **Direction of Projection** (DOP)

The projectors are parallel—they do not converge. Alternatively, the COP is at infinity.

Multiview Orthographic Projection

Used for:

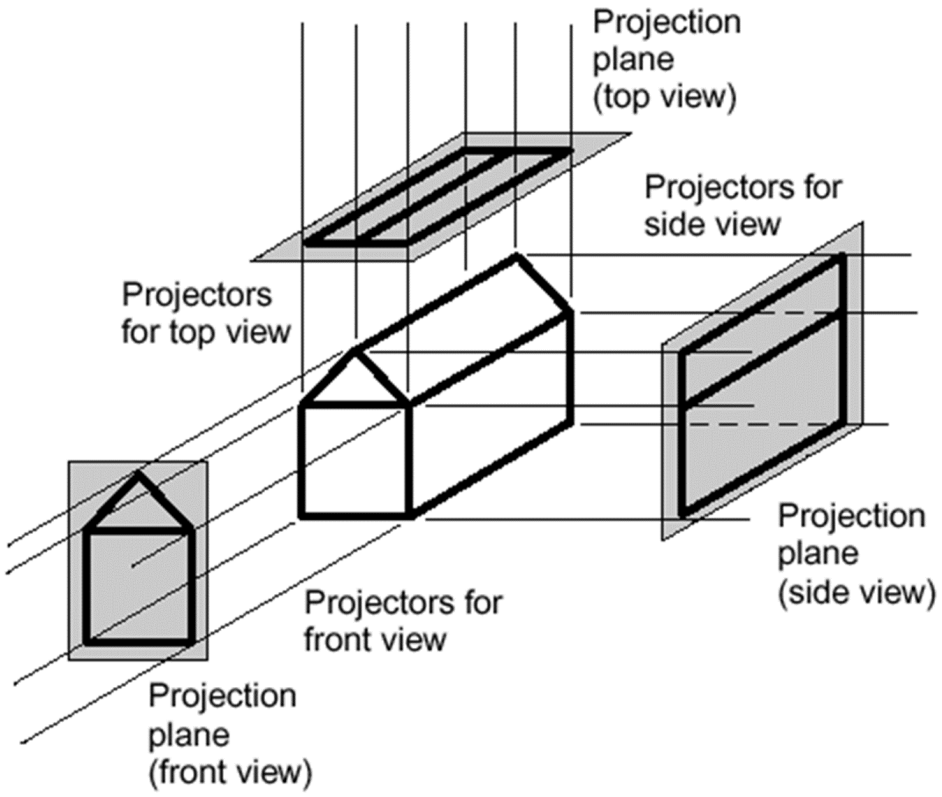
- engineering drawings of machines, machine parts
- working architectural drawings

Pros:

- accurate measurement possible
- all views are at same scale

Cons:

does not provide “realistic” view or sense of 3D form



Isometric Projection

Used for:

- catalogue illustrations
- patent office records
- furniture design
- structural design
- 3d Modeling in real time (Maya, AutoCad, etc.)

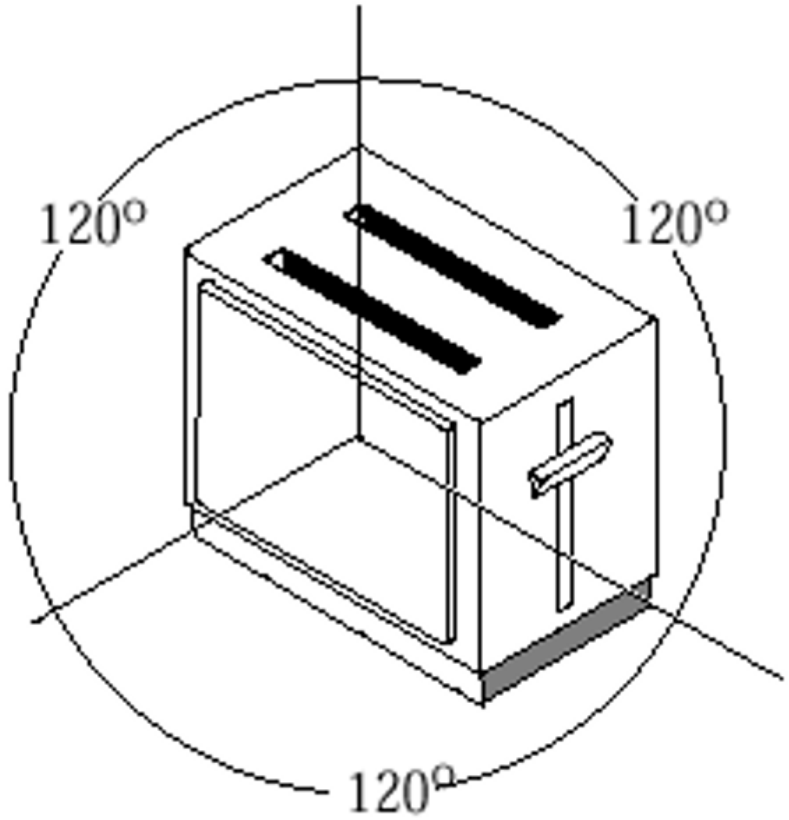
Pros:

- don't need multiple views
- illustrates 3D nature of object
- measurements can be made to scale along principal axes

Cons:

lack of foreshortening creates distorted appearance

more useful for rectangular than curved shapes

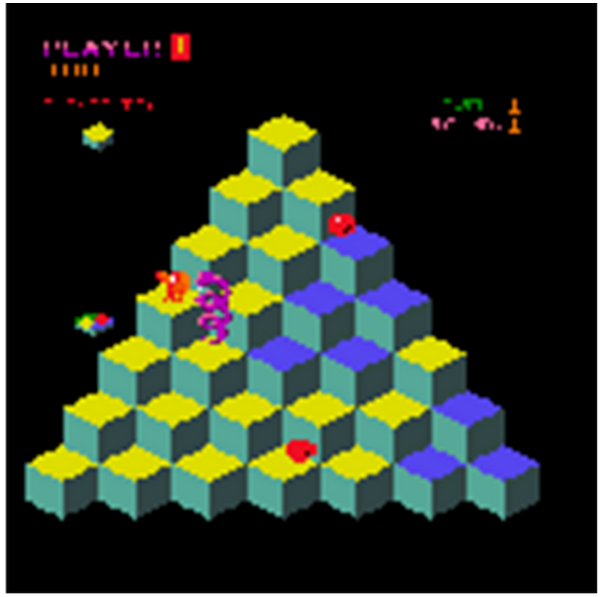


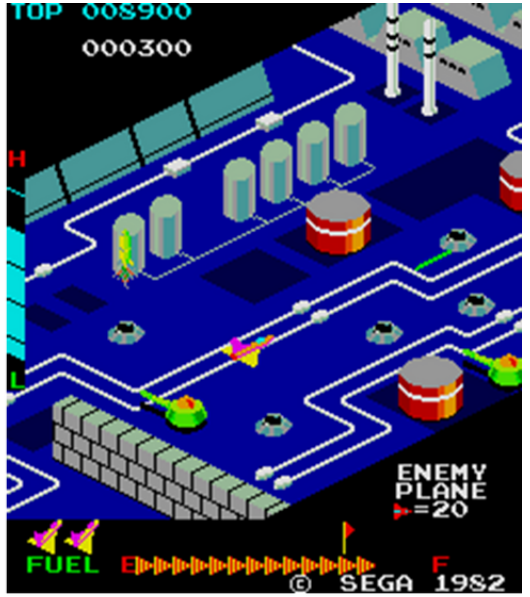
Example

Carlbom Fig.2.2

Isometric Projection

Games have been using isometric projection for ages. It all started in 1982 with **Q*Bert** and **Zaxxon**, which were made possible by advances in raster graphics hardware.





Still in use today when you want to see things in distance as well as things close up (e.g. strategy, simulation games).

Technically some games today aren't isometric but instead are a general axonometric (trimetric) with arbitrary angles, but people still call them isometric to avoid learning a new word. Other inappropriate terms used for axonometric views are "2.5D" and "three-quarter."

SimCity IV (Trimetric)

StarCraft II

The Synthetic Camera Model

The Synthetic Camera

The **synthetic camera** is the programmer's model to specify 3D view projection parameters.

Each graphics package has its own but they are all (nearly) equivalent.

General synthetic camera:

- position of camera
- orientation
- field of view (wide angle, normal…)
- depth of field (near distance, far distance)
  perspective or parallel projection

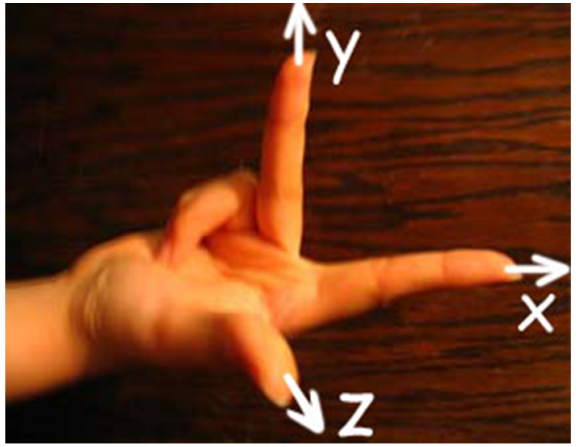
Position

Determining the position is analogous to a photographer deciding the vantage point from which to shoot a photo.

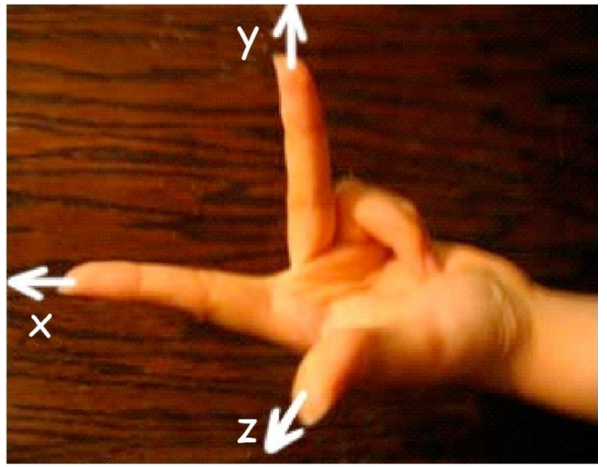
Three degrees of freedom: x, y, and z coordinates in 3-space

This **x, y, z** coordinate system is **right-handed**:

If you open your right hand, align your palm and fingers with the +x axis, and curl your middle finger towards the +y axis, your thumb will point along the +z axis.

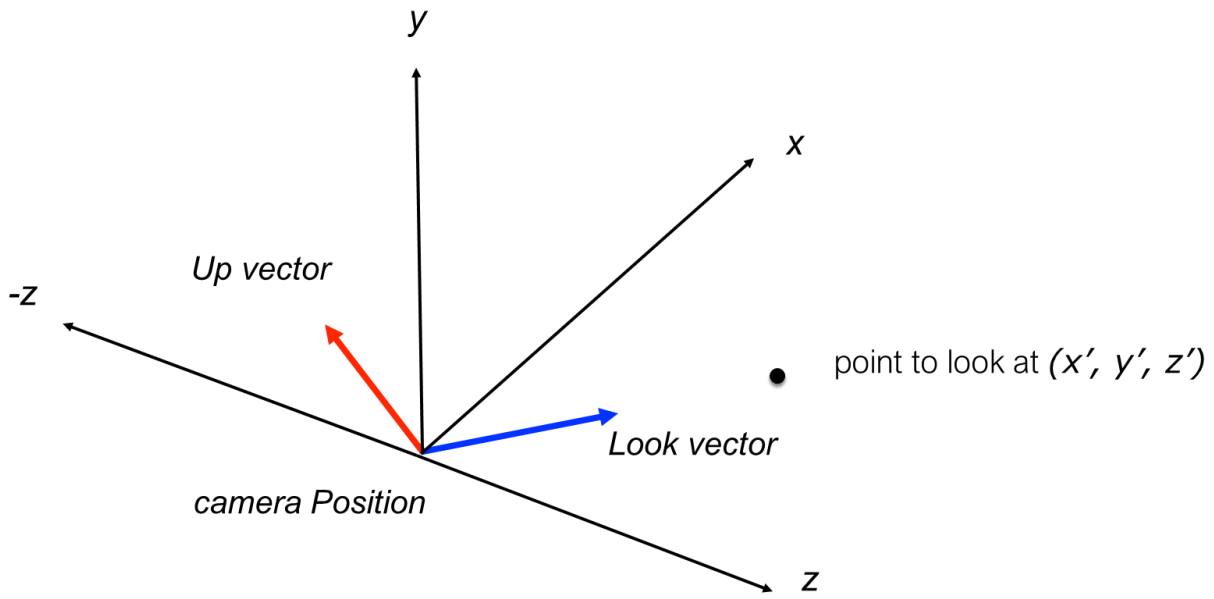


This is a right-handed coordinate system.



This is a left-handed coordinate system.
Not used in our course.

Orientation

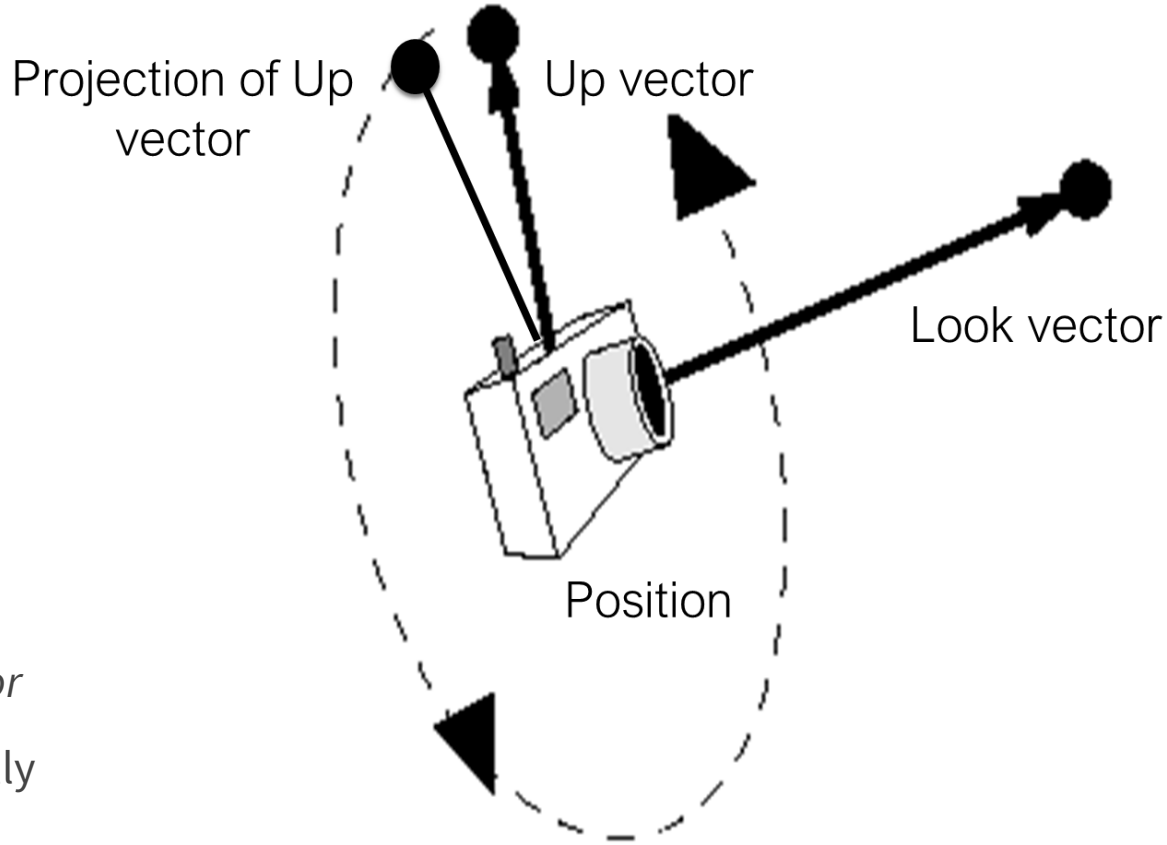


Orientation is specified by a point in 3D space to look at (or a direction to look in) and an angle of rotation about this direction.

Default (canonical) orientation is looking down the negative z-axis and up direction pointing straight up the y-axis.

In general, the camera is located at the origin and is looking at an arbitrary point with an arbitrary up direction.

Look and Up Vectors



Look Vector

- The direction the camera is pointing
- Three degrees of freedom; can be any vector in 3-space

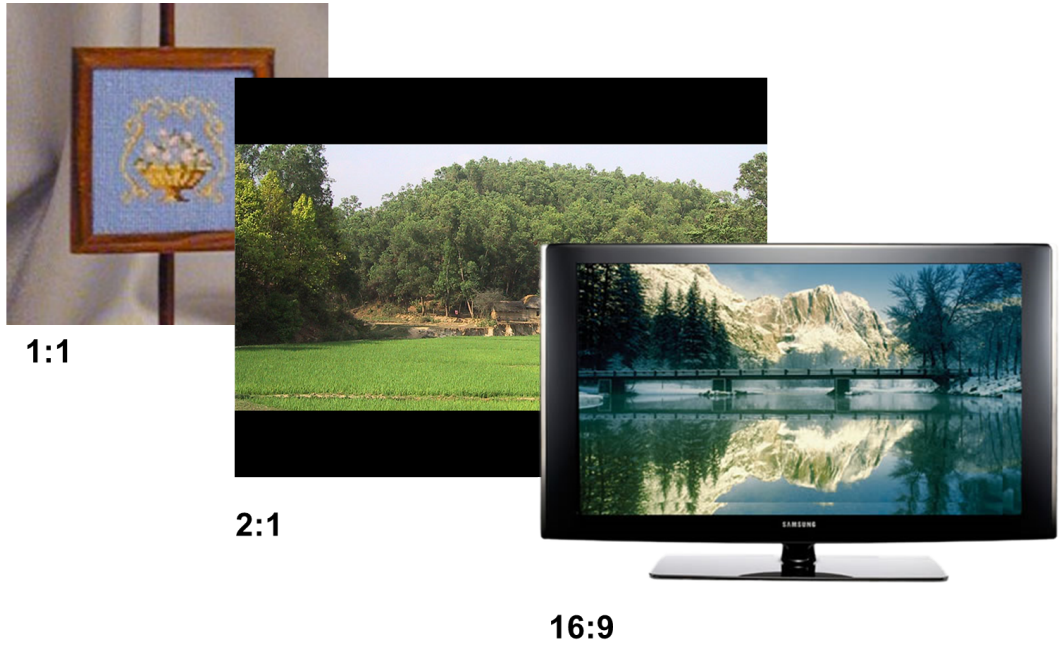
Up Vector

- Determines how the camera is rotated around the *Look vector*
- For example, whether you are holding the camera horizontally or vertically (or in between)
- *Projection* of *Up vector* must be in the plane perpendicular to the Look vector (this allows *Up vector* to be specified at an arbitrary angle to its *Look vector*)

Note: For ease of specification, ***Up vector*** need not be perpendicular to ***Look vector***, but cannot be collinear

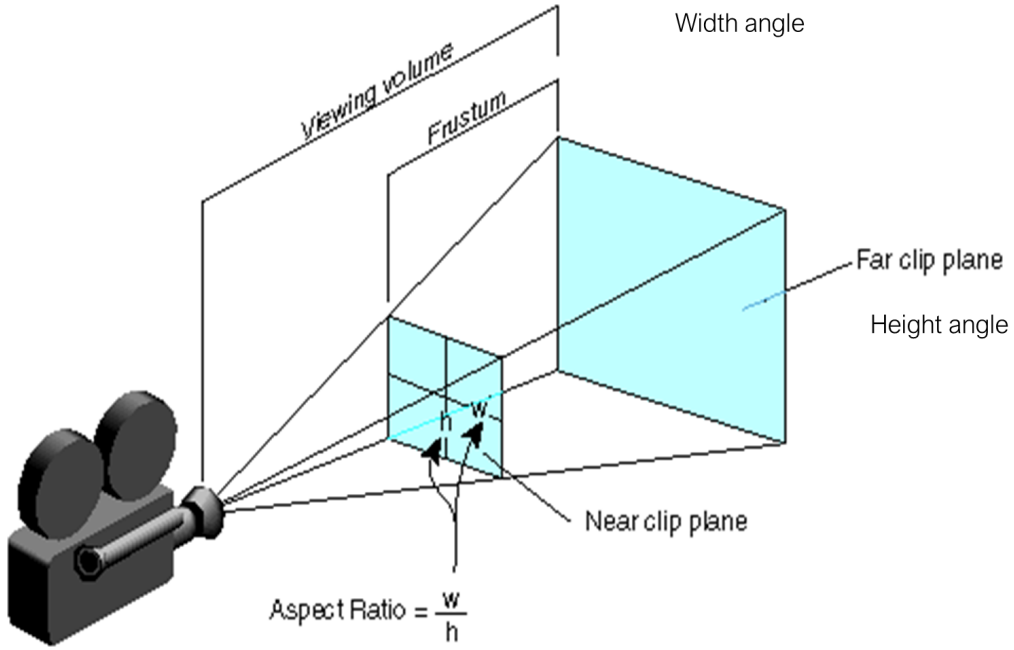
Aspect Ratio

- Analogous to the size of film used in a camera
- Determines proportion of width to height of image displayed on screen
- Square viewing window has aspect ratio of 1:1
- Movie theater "letterbox" format has aspect ratio of 2:1
- NTSC television has an aspect ratio of 4:3, and HDTV is 16:9

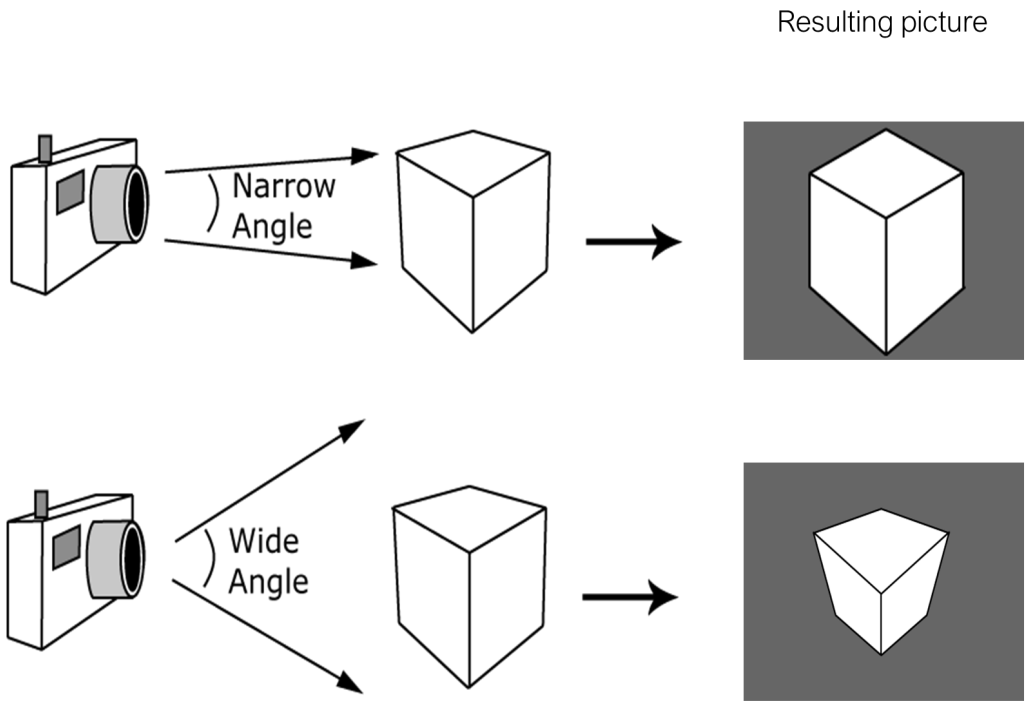


View Angle

- Determines amount of perspective distortion in picture, from none (parallel projection) to a lot (wide-angle lens).
- In a **frustum**, two viewing angles for width and height.
- Choosing a *View angle is* analogous to a photographer choosing a specific type of lens (e.g., a wide-angle or telephoto lens).



View Angle

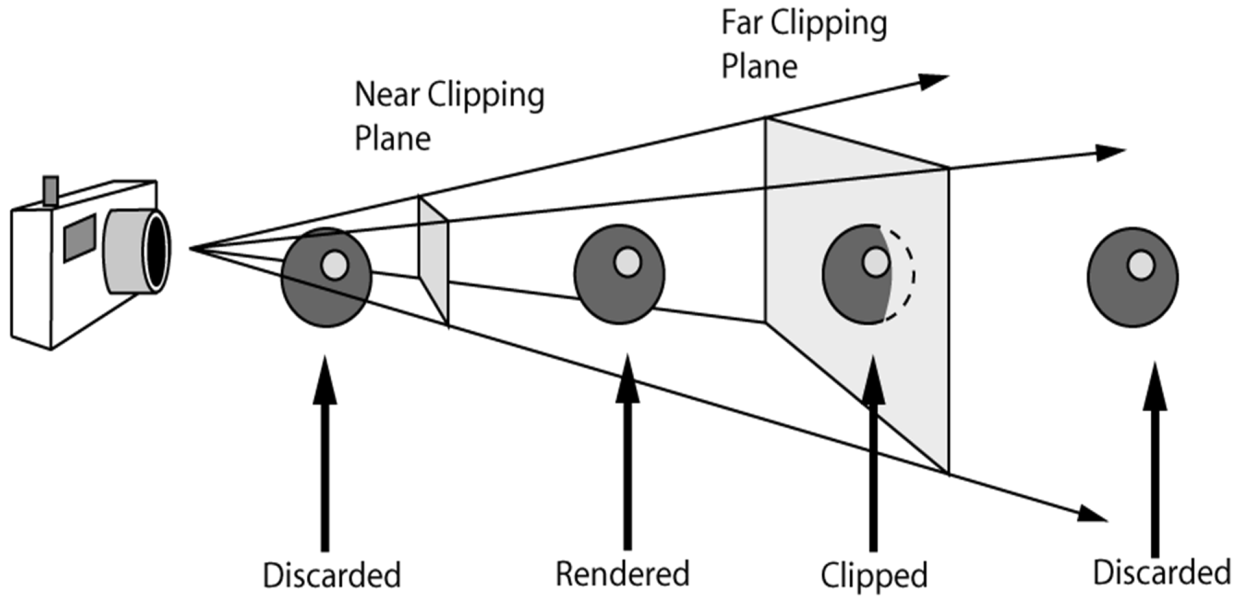


- Lenses made for distance shots often have a nearly parallel viewing angle and cause little perspective distortion, though they foreshorten depth.

  Wide-angle lenses cause a lot of perspective distortion.

Clipping Planes

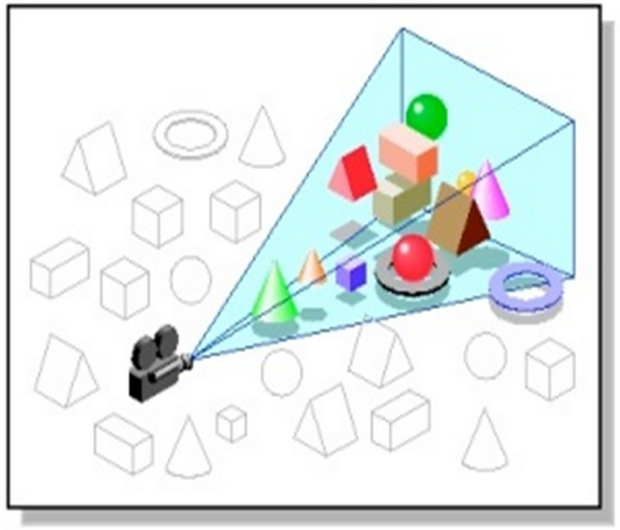
- Don't want to draw things behind the camera.
- Drawing lots of faraway things makes rendering slow.
- There are a finite number of bits in the depth buffer!

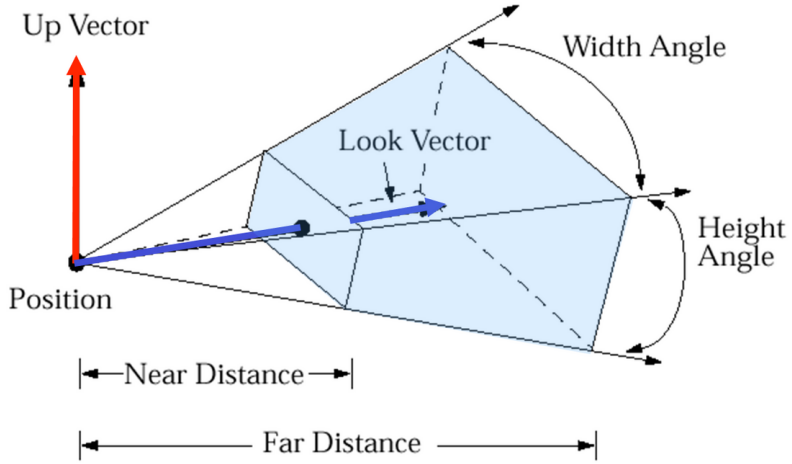


Focal Length

- Some camera models take a **focal length.**
- Focal length is a measure of ideal focusing range; approximates behavior of a real camera lens.
- Objects at distance equal to focal length from camera are rendered in focus.
- Objects closer or farther away than focal length get blurred.
- Focal length used in conjunction with clipping planes. Only objects within view volume are rendered, whether blurred or not. Objects outside of view volume still get discarded.

View Volume Specification





frustum = truncated pyramid

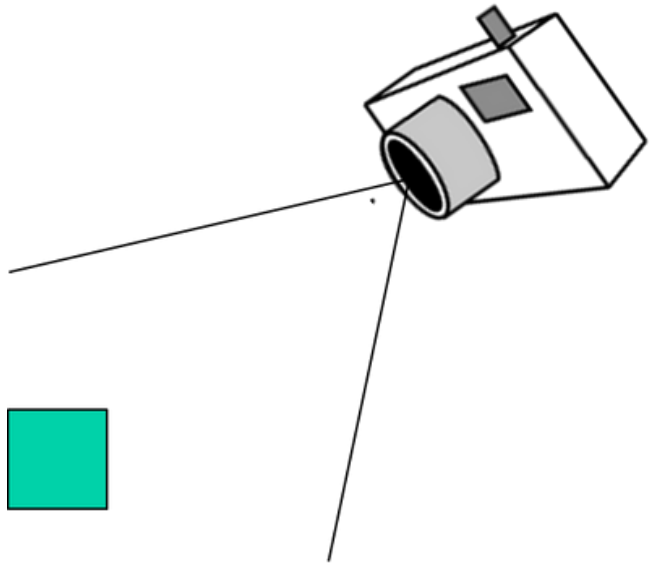
Position, Look vector, Up vector, Aspect ratio, Height angle, Clipping planes, and (optionally) *Focal length* together specify a truncated view volume.

Truncated view volume is a specification of bounded space that camera can "see."

A 2D view of the 3D scene can be computed from truncated view volume and projected onto the film plane.

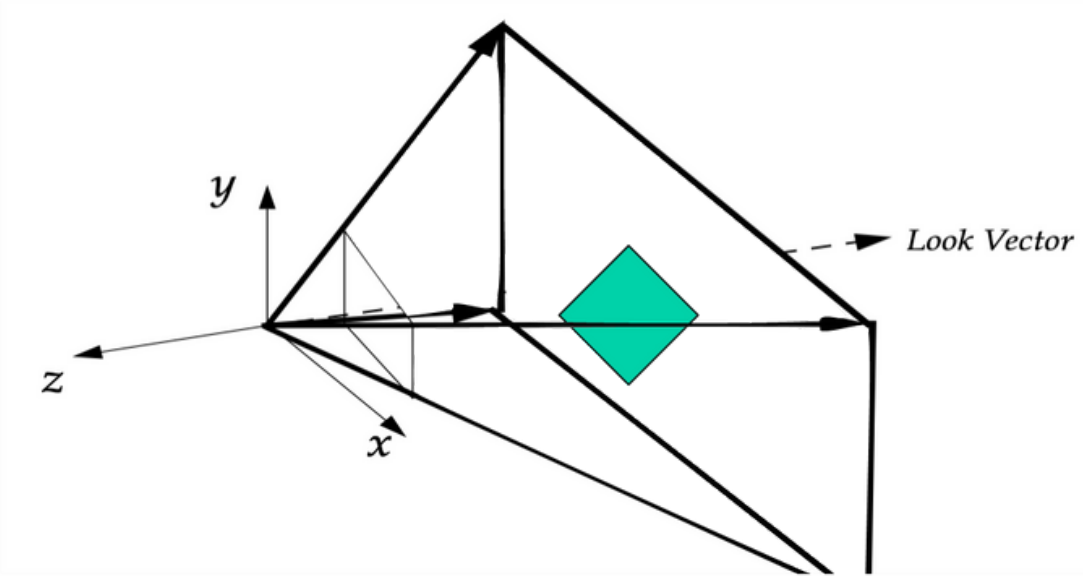
The View Matrix

world space



World space

eye space



Camera space

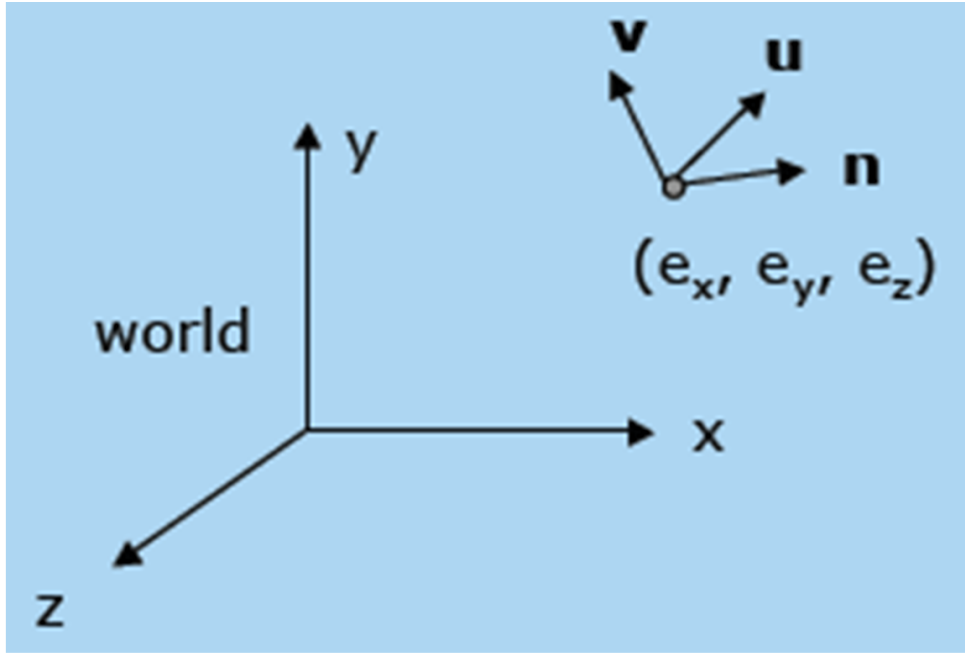
view matrix

Official Definition of the View Matrix

The view matrix transforms world points into camera coordinates, a.k.a. the world-to-eye transform.

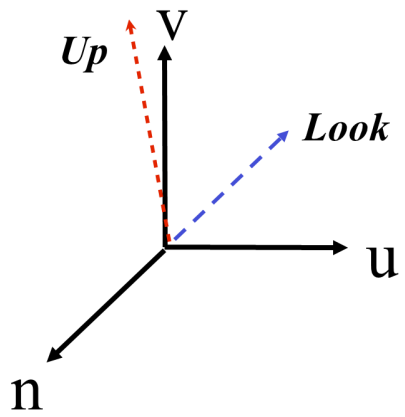
We can think of it as the matrix that:

- moves the camera position (eye) to the origin
- rotates to align u,v,n with x,y,z



u,v,n are typically used to denote the camera's local coordinate system, where -n = the look direction

Step 1: Find u,v,n from Position, Look, and Up



We know that we want the *(u, v, n)* axes to have the following properties:

- our arbitrary *Look Vector* will lie along the negative n-axis
- a projection of the *Up Vector* into the plane defined by the n-axis as its normal will lie along the v-axis
- the *u*-axis will be mutually perpendicular to the *v* and *n*-axes, and will form a right-handed coordinate system

Plan of attack: first find *n* from *Look*, then find *u* as a normal for the plane defined by *Up* and *n*, then find *v* as a normal to the plane defined by *n* and *u*

Step 2: Create the View Matrix

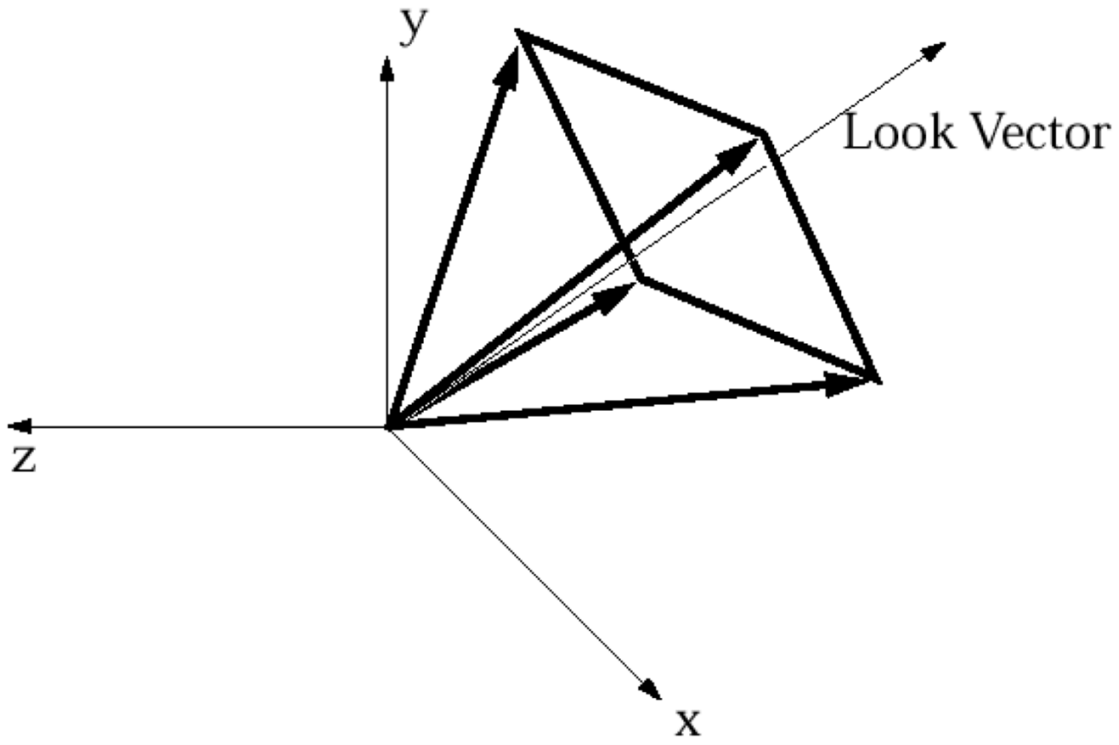
The translation (move eye to origin) part is pretty easy.

We want to transform (e_x, e_y, e_z) to (0,0,0).

Solution: translate by <-e_x, -e_y, -e_z>.

$$T = \begin{pmatrix} 1 & 0 & 0 & -e_x \\ 0 & 1 & 0 & -e_y \\ 0 & 0 & 1 & -e_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Step 3: Rotate to align u,v,n with x,y,z

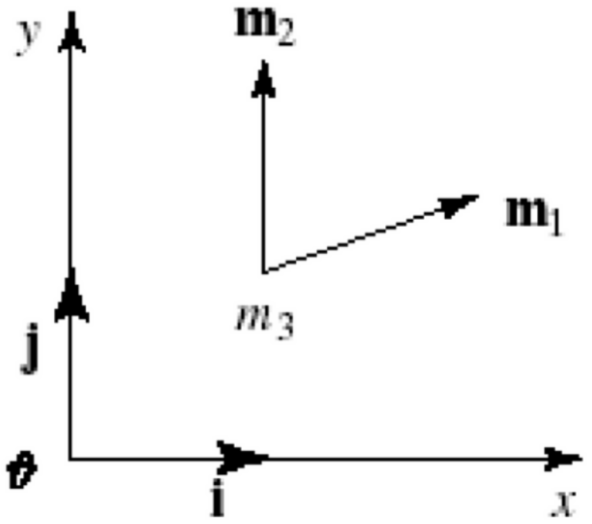


We also need to rotate so the camera ends up aligned properly with the x,y,z axes.

How do we construct a rotation matrix that takes one set of axes and aligns it to another?

Transformation Matrices: What do the Columns Tell Us?

$$M = \begin{pmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ 0 & 0 & 1 \end{pmatrix} = \left(m_1 \mid m_2 \mid m_3\right)$$



The first two columns are:

- vectors (3rd component is 0)
- the X-axis and Y-axis of the coordinate frame specified by the transformation -- if a rotation matrix, these columns will show the vectors into which the X and Y-axes rotate.

Third column is:

a point (3rd component is 1)

the origin of the coordinate frame

Thus, the following will rotate x,y,z into u,v,n:

$$R_{xyz2uvn} = \begin{pmatrix} u_x & v_x & n_x & 0 \\ u_y & v_y & n_y & 0 \\ u_z & v_z & n_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Thus, the following will rotate x,y,z into u,v,n:

$$R_{xyz2uvn} = \begin{pmatrix} u_x & v_x & n_x & 0 \\ u_y & v_y & n_y & 0 \\ u_z & v_z & n_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Now, this isn't exactly right. We want to go the other way, from *uvn* to *xyz*.

Thus, the following will rotate x,y,z into u,v,n:

$$R_{xyz2uvn} = \begin{pmatrix} u_x & v_x & n_x & 0 \\ u_y & v_y & n_y & 0 \\ u_z & v_z & n_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Now, this isn't exactly right. We want to go the other way, from *uvn* to *xyz*.

Recall, the inverse of a pure rotation matrix is its transpose:

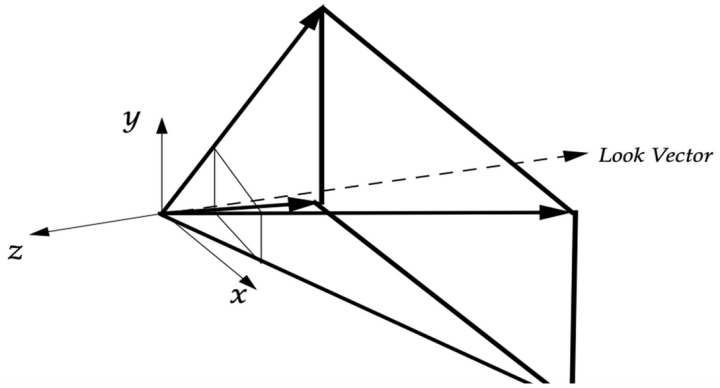
$$R_{uvn2xyz} = \begin{pmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ n_x & n_y & n_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Final View Matrix

Transforms points in world coordinates to eye coordinates.

You can also think of this as going from the situation where the camera is at some arbitrary location and orientation in the scene to a situation where it is positioned and oriented in the standard arrangement as shown below.

You can apply this exact matrix manually onto a Camera in a graphics API, and it will work. BUT… we usually use a utility function such as **LookAt()**.



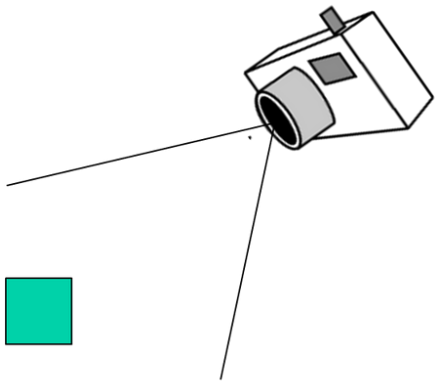
$$V = \begin{pmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ n_x & n_y & n_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & -e_x \\ 0 & 1 & 0 & -e_y \\ 0 & 0 & 1 & -e_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$V = \begin{pmatrix} u_x & u_y & u_z & d_x \\ v_x & v_y & v_z & d_y \\ n_x & n_y & n_z & d_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

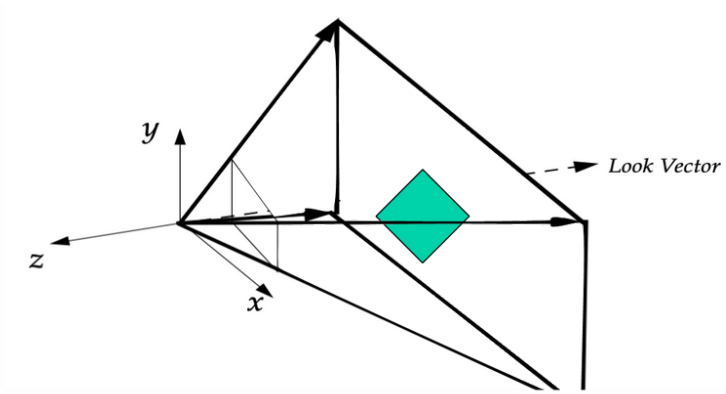
where **d** = (**-e·u**, **-e·v**, **-e·n**)

The Projection Matrix

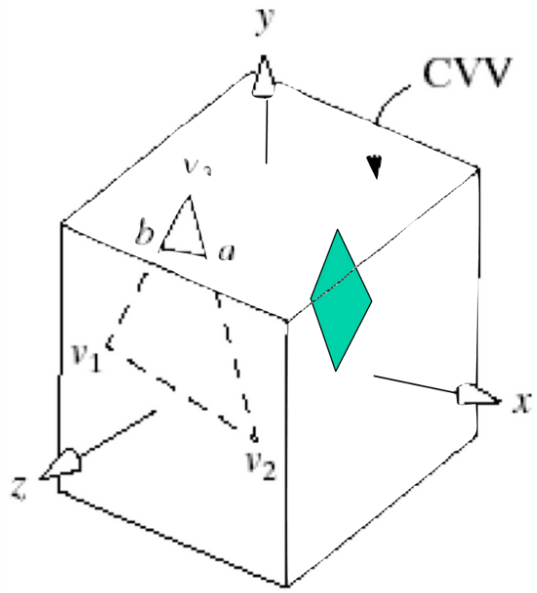
world space



eye space



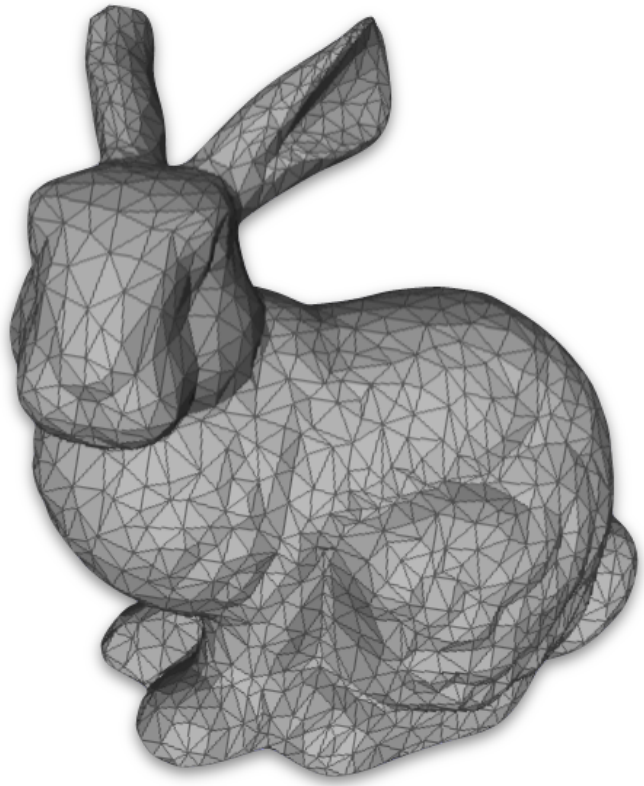
canonical view volume



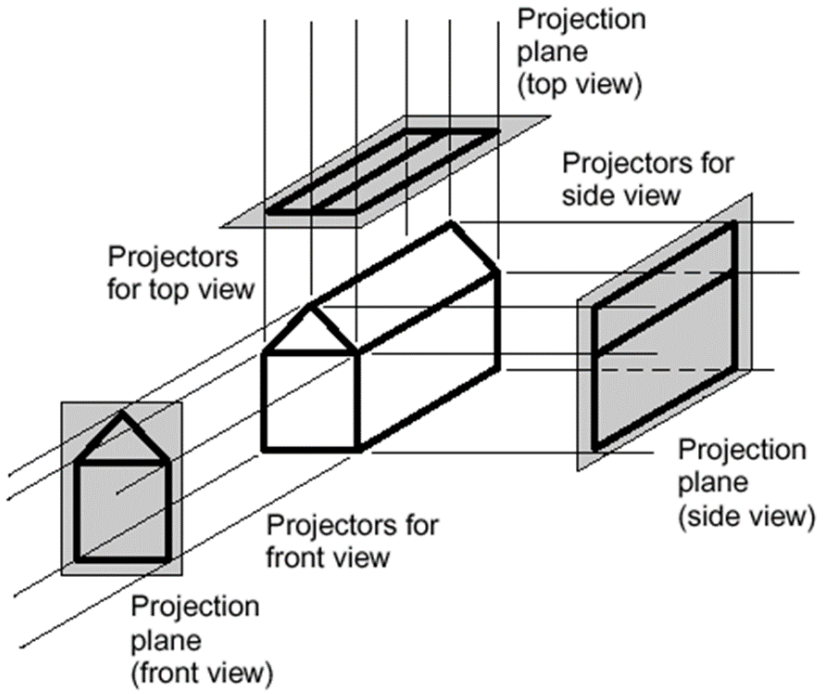
view matrix

proj matrix

Two Most Common Forms of Projection



Perspective Projection

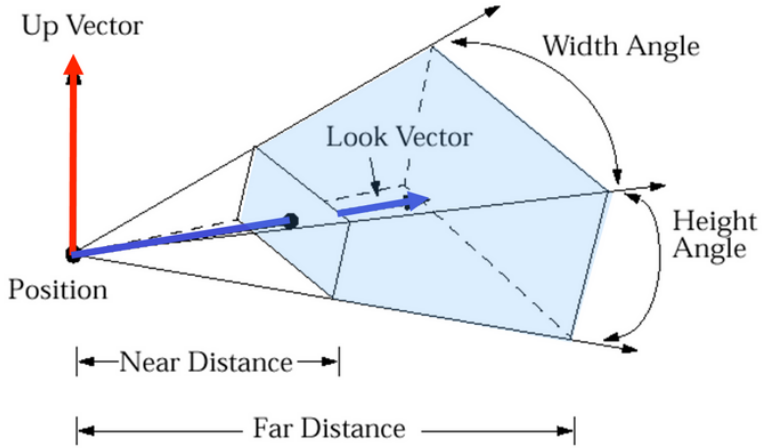


Parallel Projection

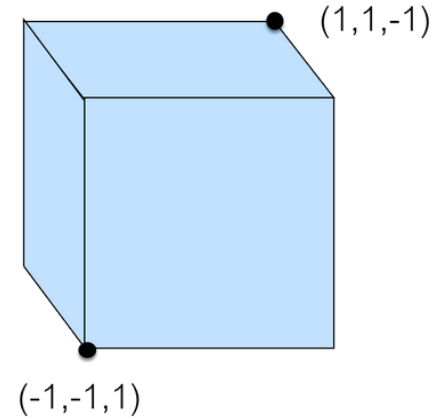
“View Volume”

“Canonical View Volume”

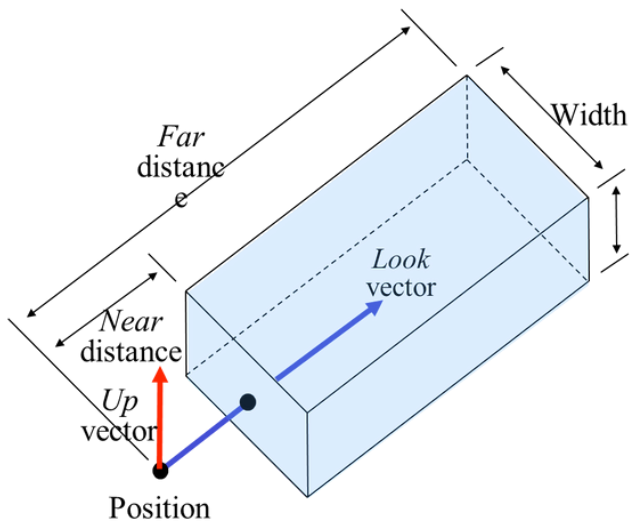
Perspective



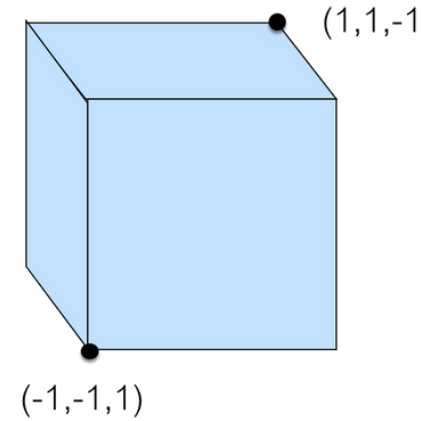
proj matrix



Parallel



proj matrix



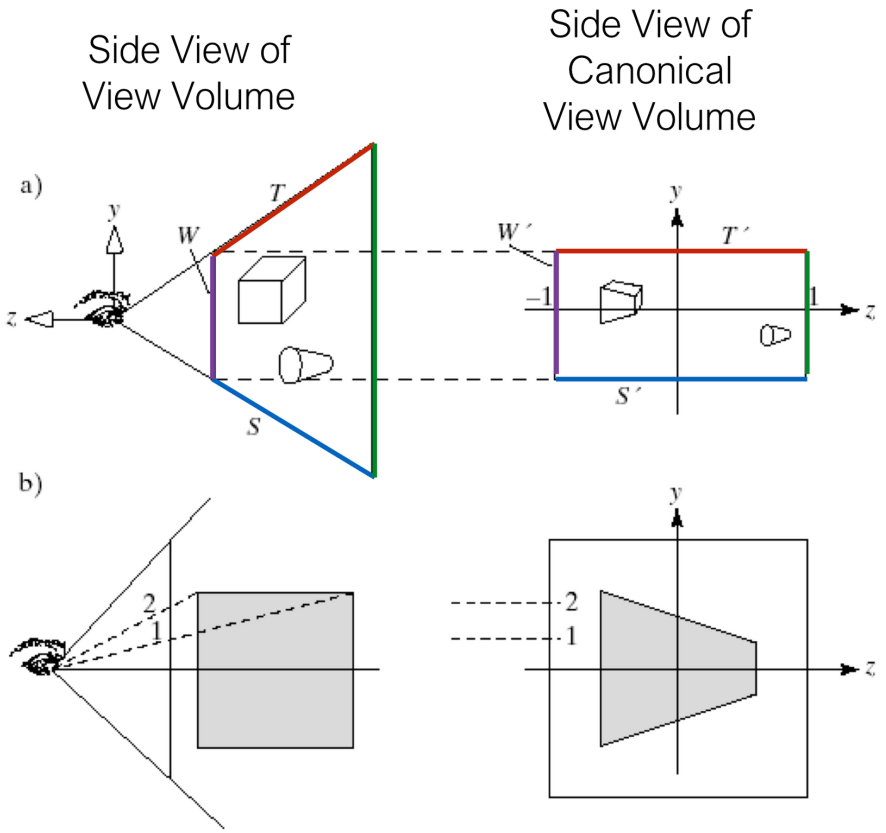
Canonical View Volume: Benefit #1

This is a super cool way to handle perspective projection!

The view volume is transformed from a truncated pyramid into a box.

For all the objects inside the view volume, this means objects that are farther away will get "smushed" in the X and Y directions (i.e., objects that are farther away appear smaller).

This is actually just what we want!

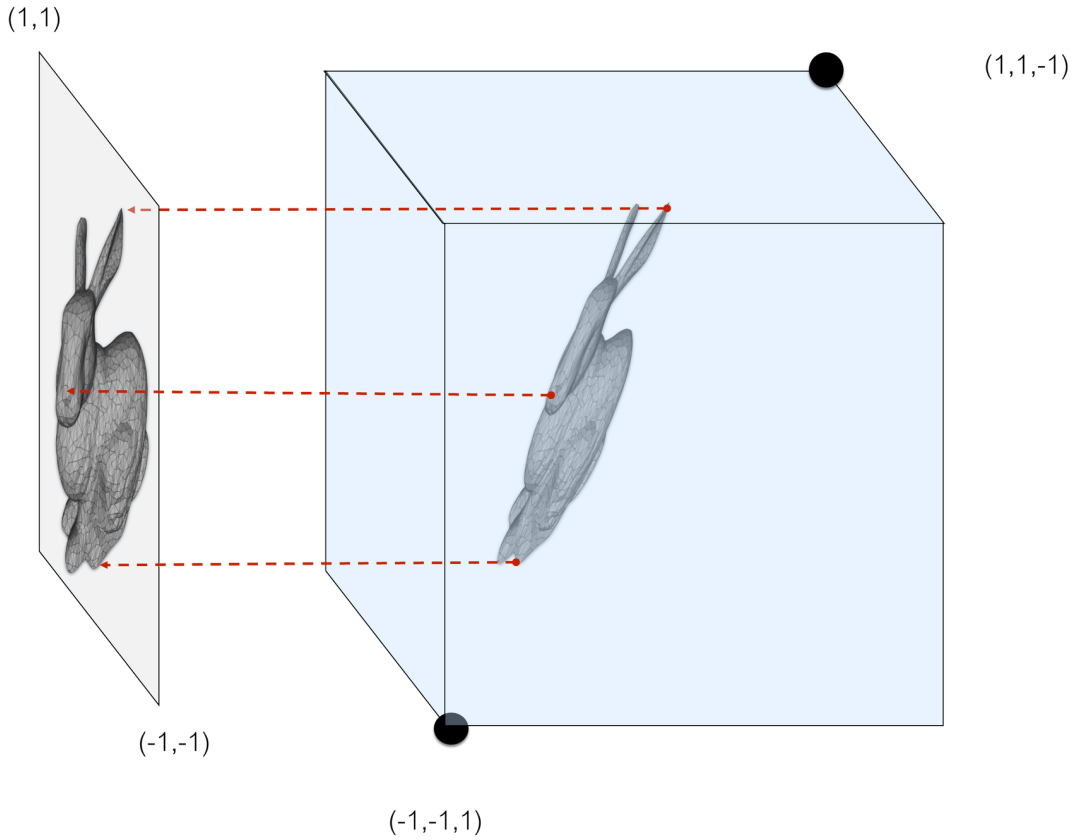


Canonical View Volume: Benefit #2

In this space, projection is super simple.

Just "drop" the Z coordinate for each vertex.

Or, think of it as moving each vertex to the z=1 side of the cube.



Canonical View Volume: Benefit #3

It preserves relative depth ordering.

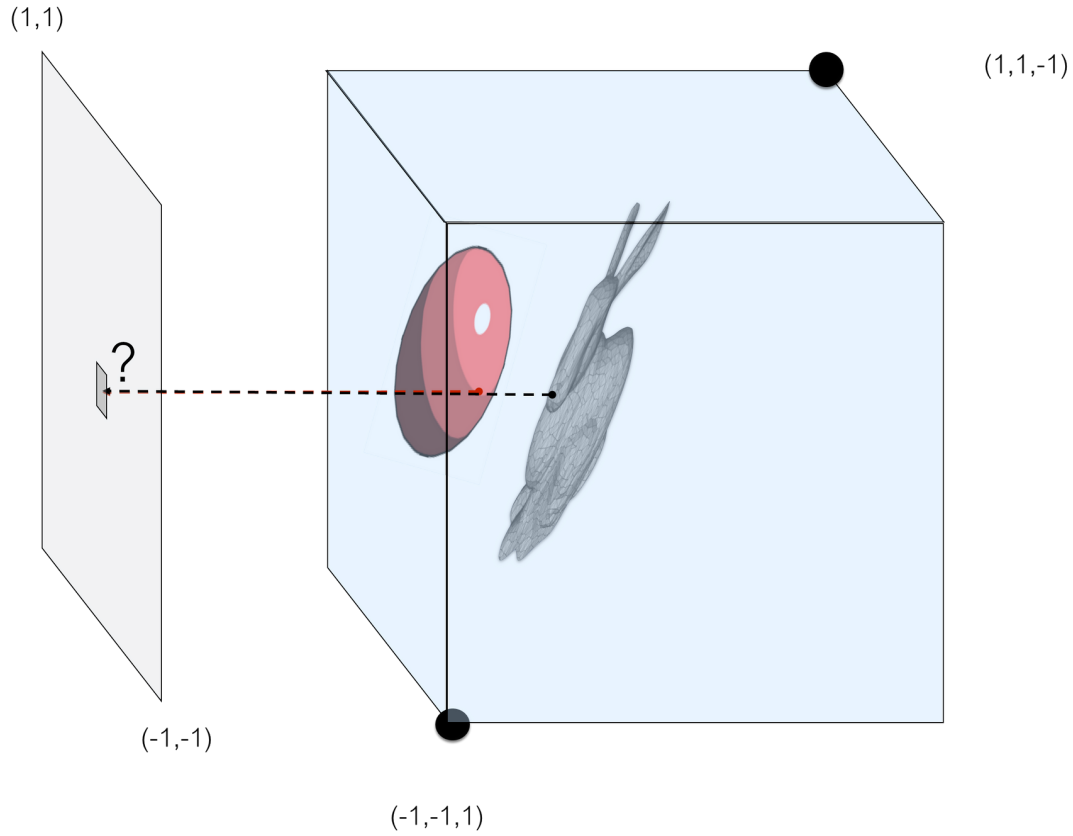
Can we really just “drop” the Z coordinate? For projection, yes!

But, there is also the issue of depth ordering.

We need to know the relative depth of each fragment so we can tell which color should end up on top.

So, we actually keep the -1 to +1 Z values.

Note, we don’t need the “actual” depth from the camera (this is costly to compute). We just need the relative depth. So, the -1 to +1 range is fine, and we call this **pseudodepth**.

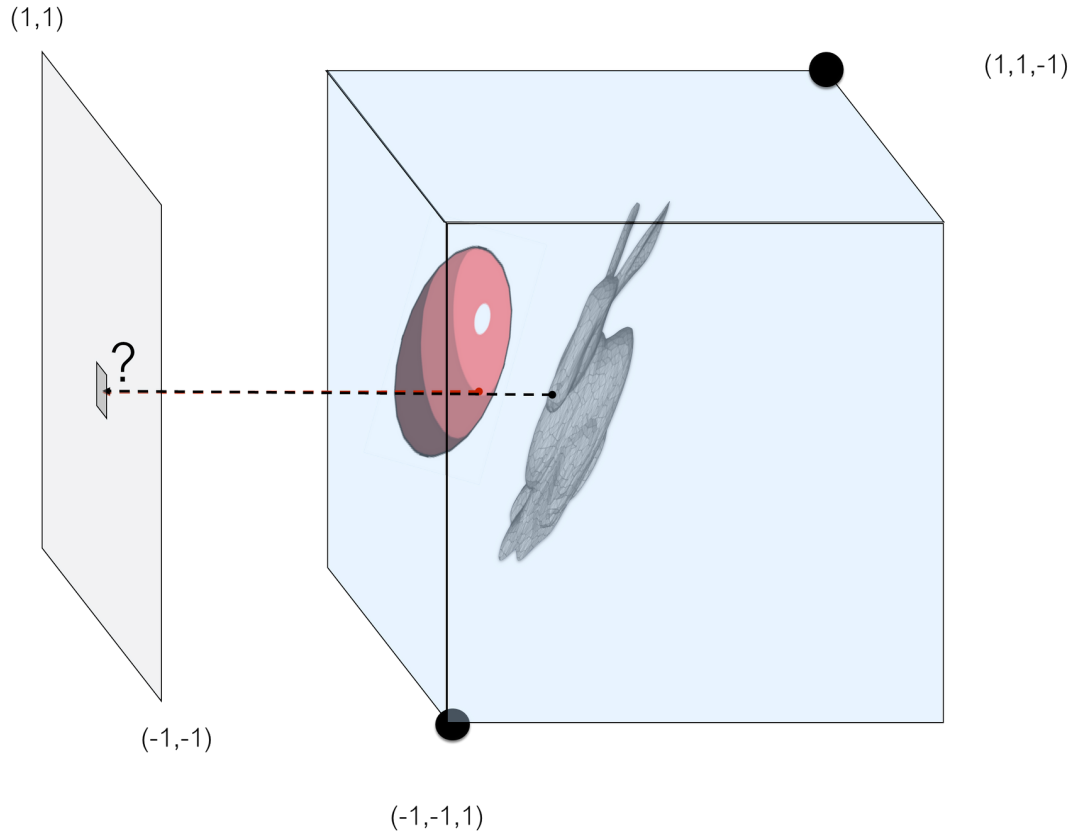


Using Pseudodepth

To make sure we get the right color on top, we could sort everything front to back based on pseudodepth and then draw the objects in back to front order… but this is costly!

It is not just the objects or even the vertices that we would need to sort, we need to sort each fragment because sometimes objects are very close or intersect each other.

So, we adopt a different solution that let's us draw the objects in any order, but this means that in addition to the *r,g,b* color for each pixel, we also need to store the depth of each pixel.



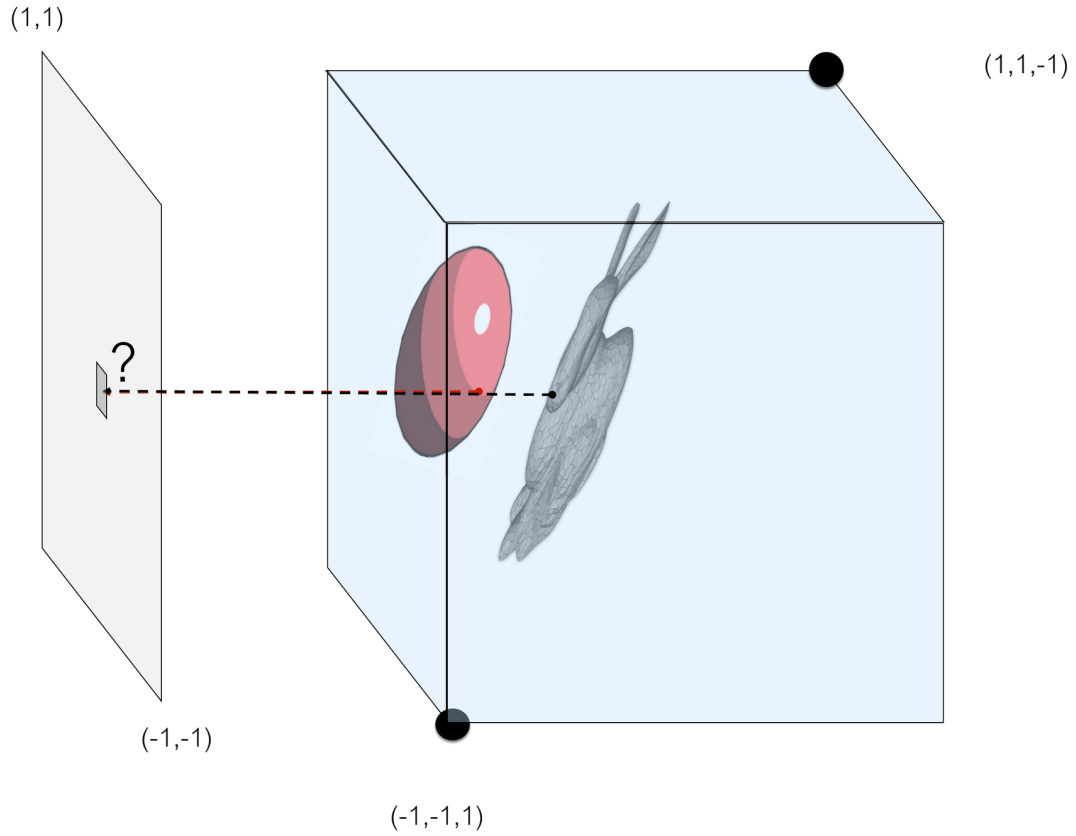
The “Z Buffer” and “Z Test”

Every time we write (*r,g,b*) to the color buffer, we also write pseudodepth to a **Z buffer**.

Before rendering each fragment, the rasterizer has already figured out which pixel on the screen the fragment will be projected to. So, we look up the current pseudodepth for that pixel in the Z-Buffer and compare it to the pseudodepth for the new fragment.

If the new fragment is closer, it passes the **Z-Test**, and we run the fragment shader and write to the color buffer and z buffer.

If the Z-Test fails, we know there is some other object closer to the camera that blocks the view of this fragment, so we discard the fragment, and most of the time the pipeline is smart enough to not even run the fragment shader in this case.



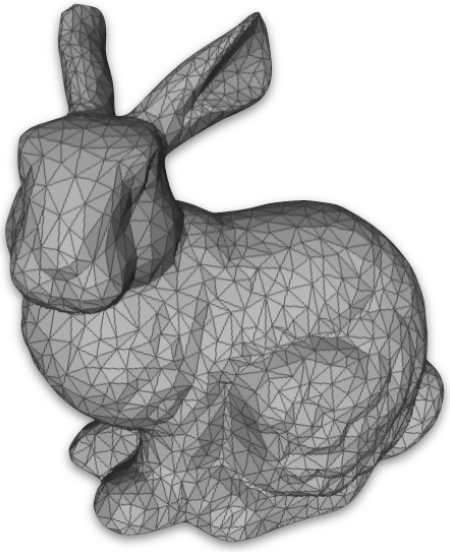
Canonical View Volume Summary

It's pretty awesome!

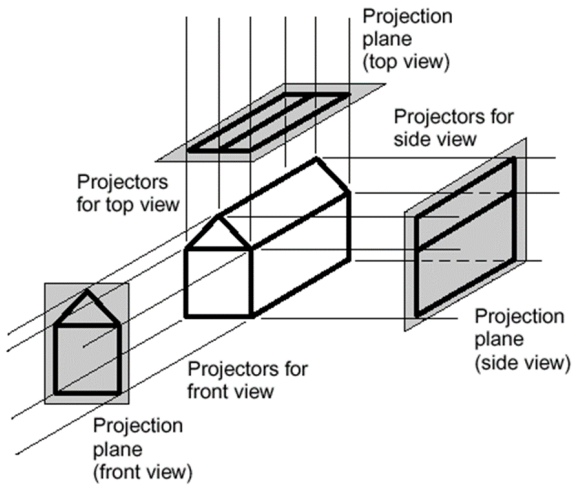
1. **Perspective distortion just works!**
2. **Projection onto a plane is as easy as dropping Z.**
3. **Relative depth ordering is preserved with pseudodepth.**

The Big Remaining Question

How do we use matrix multiplication to transform from the original view volume to the canonical view volume in our two cases?



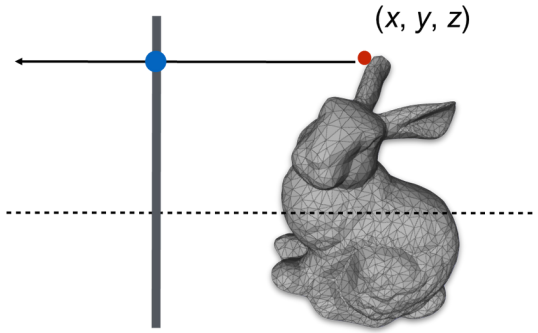
Perspective Projection

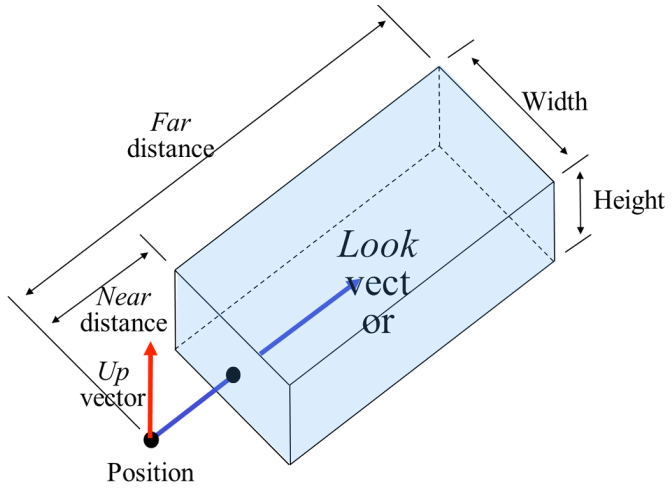


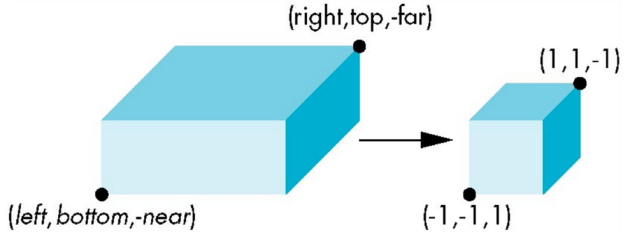
Parallel Projection

Parallel Projection

This case is really easy.



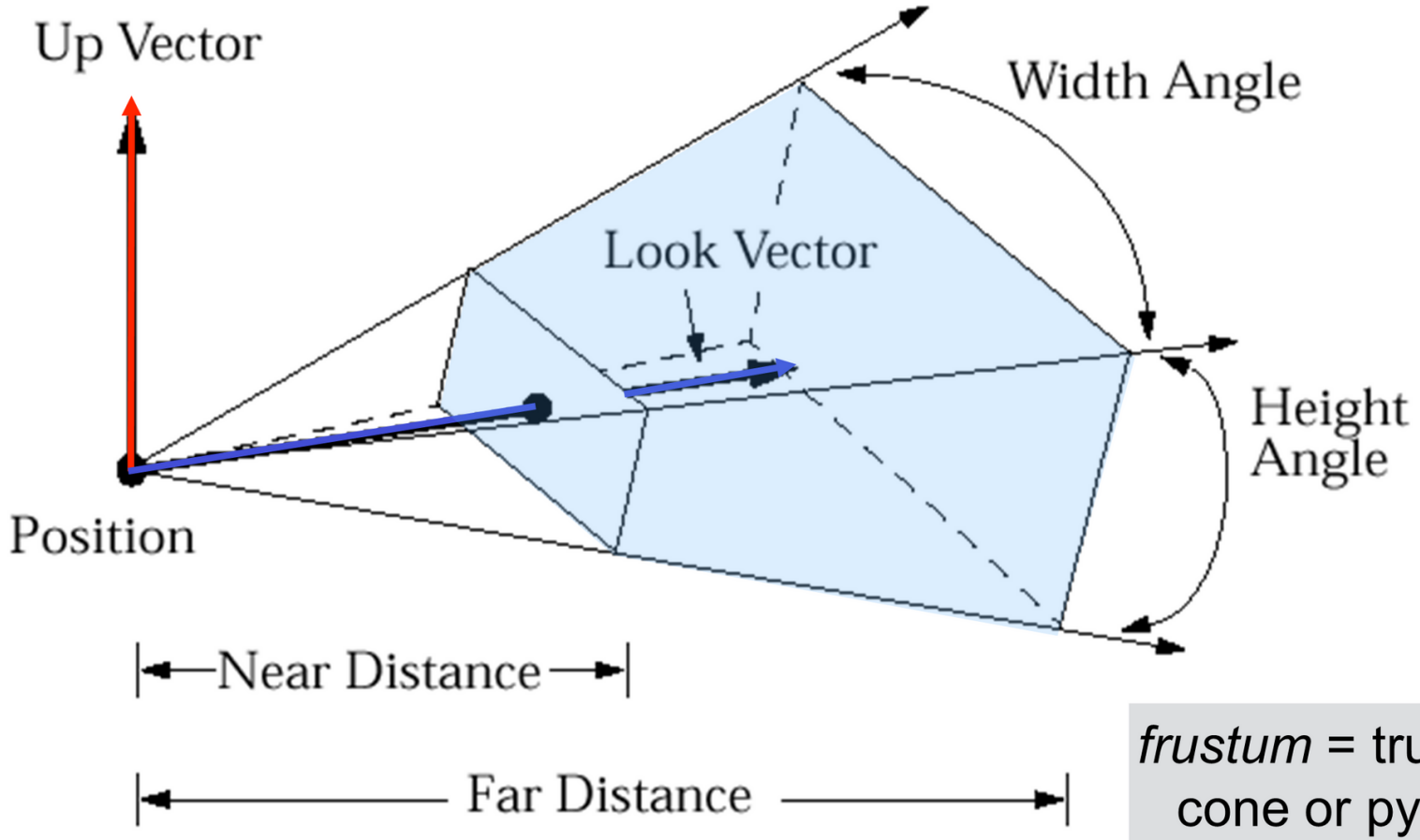




All we have to do is a non-uniform scale and a translation. We just use the same transformation matrices we would use to transform a box!

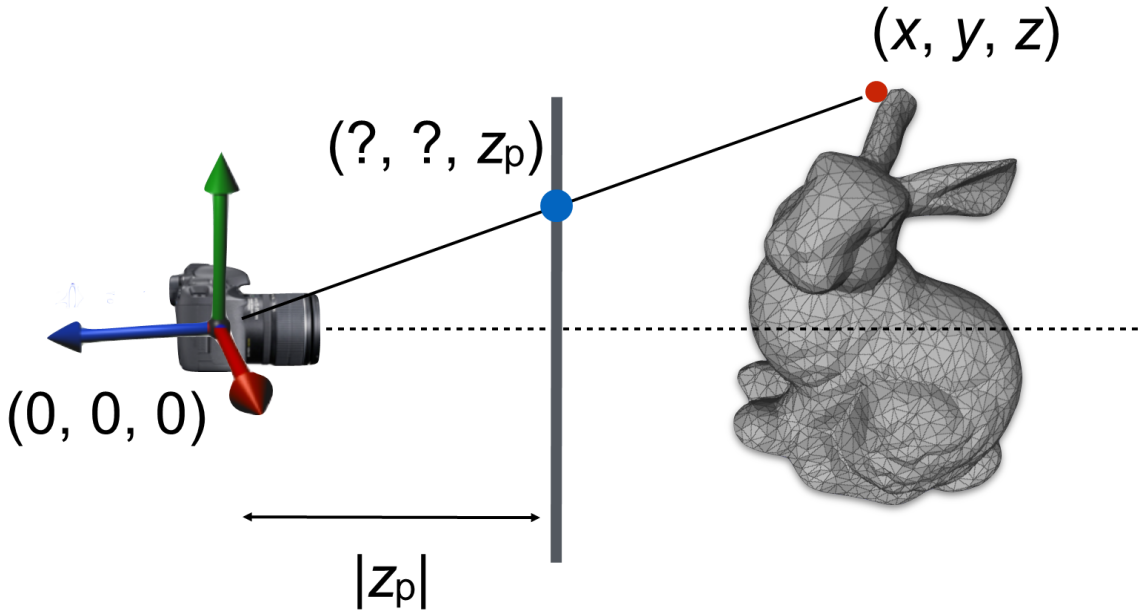
Orthographic parallel projection: projectors are perpendicular to projection plane.

Perspective View Volume



frustum = truncated cone or pyramid

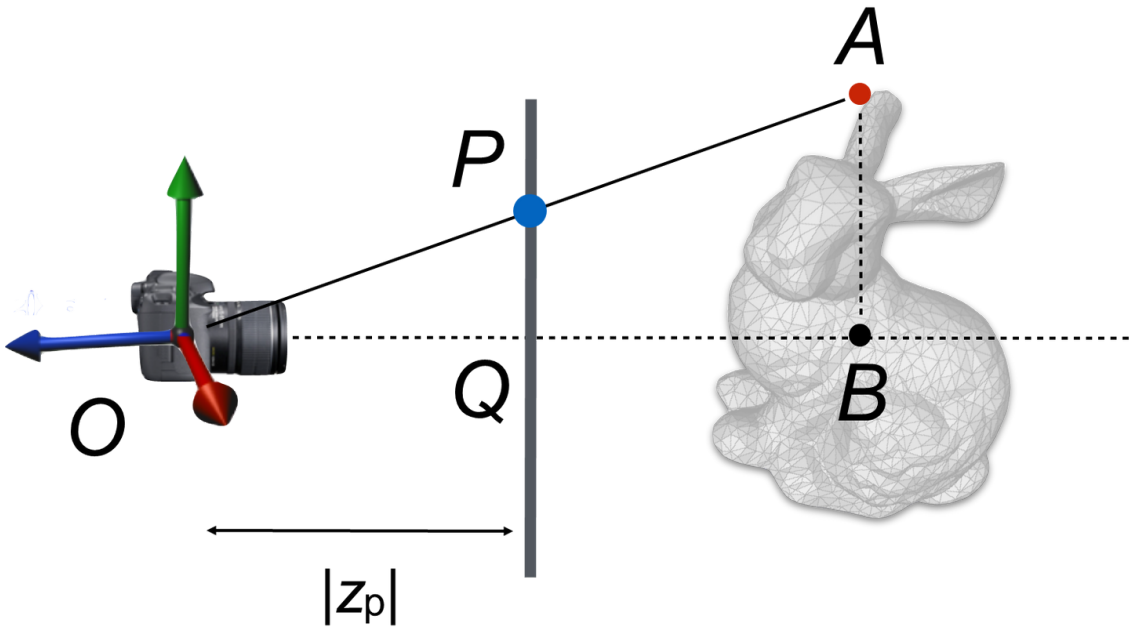
Perspective Projection



In view space (a.k.a. eye space, camera space), the projection plane is parallel to *xy* and perpendicular to *z*.

Where does a point (x, y, z) project to?

Perspective Projection

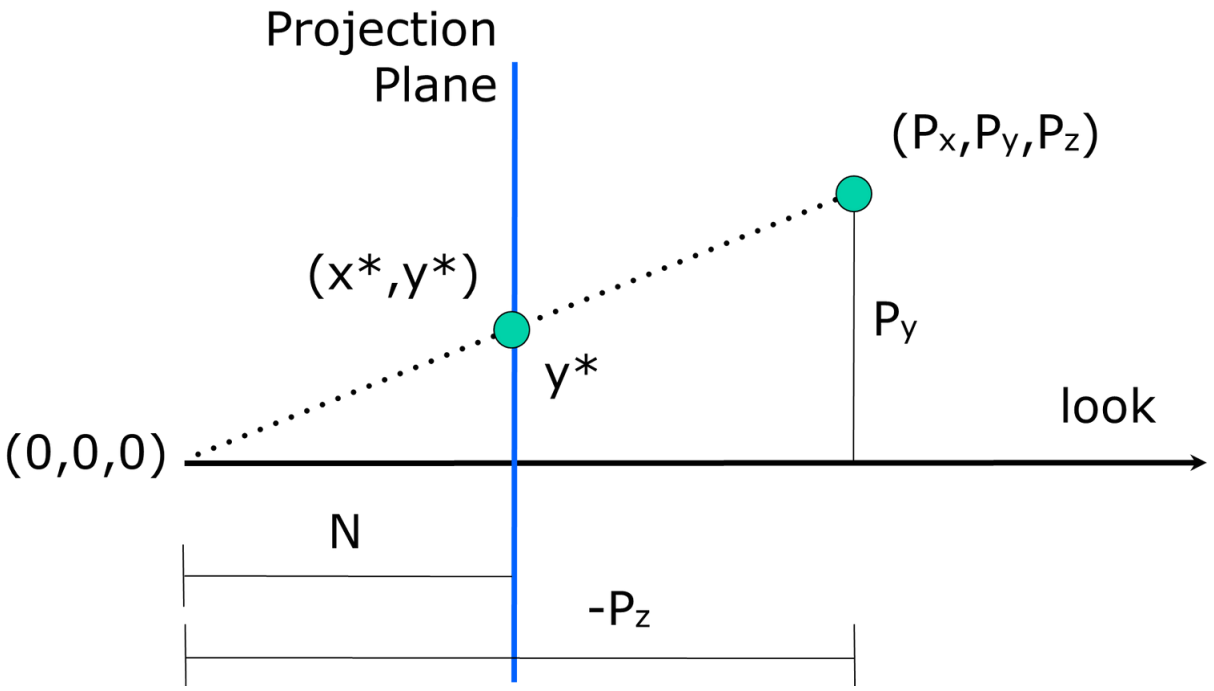


Similar triangles:

$PQ/OQ = AB/OB$

$y_p/z_p = y/z$

Perspective Projection of a Point



$$\frac{y^*}{P_y} = \frac{N}{-P_z} \quad \text{or} \quad y^* = \frac{NP_y}{-P_z}$$

$$(x^*, y^*) = \left(\frac{NP_x}{-P_z}, \frac{NP_y}{-P_z}\right)$$

Calculating Pseudodepth

We need the depth info to figure out which objects are closest to the camera. We could calculate distance the usual way...

$$d = \sqrt{P_x^2 + P_y^2 + P_z^2}$$

Calculating Pseudodepth

We need the depth info to figure out which objects are closest to the camera. We could calculate distance the usual way...

$$d = \sqrt{P_x^2 + P_y^2 + P_z^2}$$

But, this is a pretty expensive calculation to do for every point, and really we just need some measure of relative distance.

$$(x^*, y^*, z^*) = (\frac{NP_x}{-P_z}, \frac{NP_y}{-P_z}, \frac{aP_z + b}{-Pz})$$

If we choose a and b correctly, we can make z^* vary from -1 to 1 (canonical cube)

Pseudodepth: Determining *a* and *b*

Perspective Projection of a Point with "Pseudodepth":

$$(x^*, y^*, z^*) = (\frac{NP_x}{-P_z}, \frac{NP_y}{-P_z}, \frac{aP_z + b}{-Pz})$$

$$z^* = \frac{aP_z + b}{-P_z}$$

Solving for a and b, given 2 conditions:

1. z* = -1 when Pz = -N

2. z* = 1 when Pz = -F

We get: $a = \frac{-(F+N)}{F-N}$ $b = \frac{-2FN}{F-N}$

Perspective Matrix Function

All 3D graphics APIs provide a **perspective matrix function** (sometimes inside a Camera class) that creates the special 4x4 matrix that transforms (*x,y,z*) points into (*x*,y*,z**) according to the equations we just derived:

$$(x^*, y^*, z^*) = (\frac{NP_x}{-P_z}, \frac{NP_y}{-P_z}, \frac{aP_z + b}{-Pz})$$

$$a = \frac{-(F+N)}{F-N}$$

$$b = \frac{-2FN}{F-N}$$

Perspective Matrix Function

All 3D graphics APIs provide a **perspective matrix function** (sometimes inside a Camera class) that creates the special 4x4 matrix that transforms (*x,y,z*) points into (*x*,y*,z**) according to the equations we just derived:

$$(x^*, y^*, z^*) = \left(\frac{NP_x}{-P_z}, \frac{NP_y}{-P_z}, \frac{aP_z + b}{-Pz}\right) \qquad a = \frac{-(F+N)}{F-N} \qquad b = \frac{-2FN}{F-N}$$

The actual matrix looks like this:

$$R = \begin{pmatrix} \frac{2N}{right - lef\,t} & 0 & \frac{right + lef\,t}{right - lef\,t} & 0 \\ 0 & \frac{2N}{top - bottom} & \frac{top + bottom}{top - bottom} & 0 \\ 0 & 0 & \frac{-(F+N)}{F-N} & \frac{-2FN}{F-N} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

- N = near clipping dist
- F = far clipping dist
- top = N tan(viewAngle/2)
- bottom = -top
- right = top*aspectRatio
- left = -right

Take Away Messages

- Perspective transformation can be thought of as warping the space of the truncated pyramid view volume into the canonical view volume.
- This has the result of making objects that are far away from the camera appear smaller -- perfect!
- It makes sense that the matrix to implement perspective transformation is based on the near clip distance, far clip distance, and aspect ratio because these are the parameters that define the geometric shape of that truncated pyramid.
- In contrast, remember the view matrix was based on the camera position, look vector, and up vector. So, between the two matrices, we cover all 6 parameters of the computer graphics camera model!

Take Away Messages

- **Even though we think of the perspective transformation as "warping space," really it follows the same approach as all of our other transformation matrices.**

- **It is just a 4x4 transformation matrix and it doesn't really transform space – it transforms the vertices of our 3D models using a matrix multiplication the same as for every other transformation we've seen.**

- **Think of it as the very last transformation we do to those vertices is to project them onto the screen.**

- **I have left out the derivation of the 4x4 perspective matrix because this requires some serious linear algebra that goes a bit beyond this course.**
  However, if you are interested in diving into a bit more of this beautiful world of homogeneous coordinates, the math is widely available on the web.

Sources

- CSCI 4611 course materials from Daniel Keefe, 2021
- Carlbom, Ingrid and Paciorek, Joseph, Planar Geometric Projections and Viewing Transformations, Computing Surveys, Vol. 10, No. 4 December 1978.
- Kemp, Martin, The Science of Art, Yale University Press, 1992
- Mitchell, William J., The Reconfigured Eye, MIT Press, 1992
- Foley, van Dam, et. al., Computer Graphics: Principles and Practice, Addison-Wesley, 1995

  Wernecke, Josie, The Inventor Mentor, Addison-Wesley, 1994