CSC418/2504 Computer Graphics

Some Slides/Images adapted from Marschner and Shirley

Some slides courtesy of Wolfgang Hürst, Kyros Kutulakos

CSC418/2504 Computer Graphics

Some Slides/Images adapted from Marschner and Shirley

Some slides courtesy of Wolfgang Hürst, Kyros Kutulakos

Announcements

- Assignment 5 due Friday
- Assignment 6 due two weeks later

- Dave and I are still figuring out how to get me access to his Office, so Office hours still in BA5287 (M4-5 and W5-6)

- Assignment 5 help and Assignment 4 remark request
  - TA Office Hours will be 3pm-4pm on Thursday
- **TA Email Address: *csc418tas@cs.toronto.edu***

How was Midterm 1?

Any Questions ?

A note on transforming normals

- Last time

Answer: If an object is transformed by M
Then the normal vectors are transformed by …

A note on transforming normals

- Last time

Answer: If an object is transformed by M
Then the normal vectors are transformed by $(M^{-1})^T$

A note on transforming normals

- Last time

Answer: If an object is transformed by M
Then the normal vectors are transformed by $(M^{-1})^T$

- Remember to normalize your normal after this transformation! The length of the vector may be changed.

Today: The Modern Computer Graphics Pipeline

The Graphics Pipeline

- Motivation and Questions
- Overview
- Windowing Transformation
- Camera Transformation
- Perspective Transformation

The Graphics Pipeline

- Motivation and Questions
- Overview
- Windowing Transformation
- Camera Transformation
- Perspective Transformation

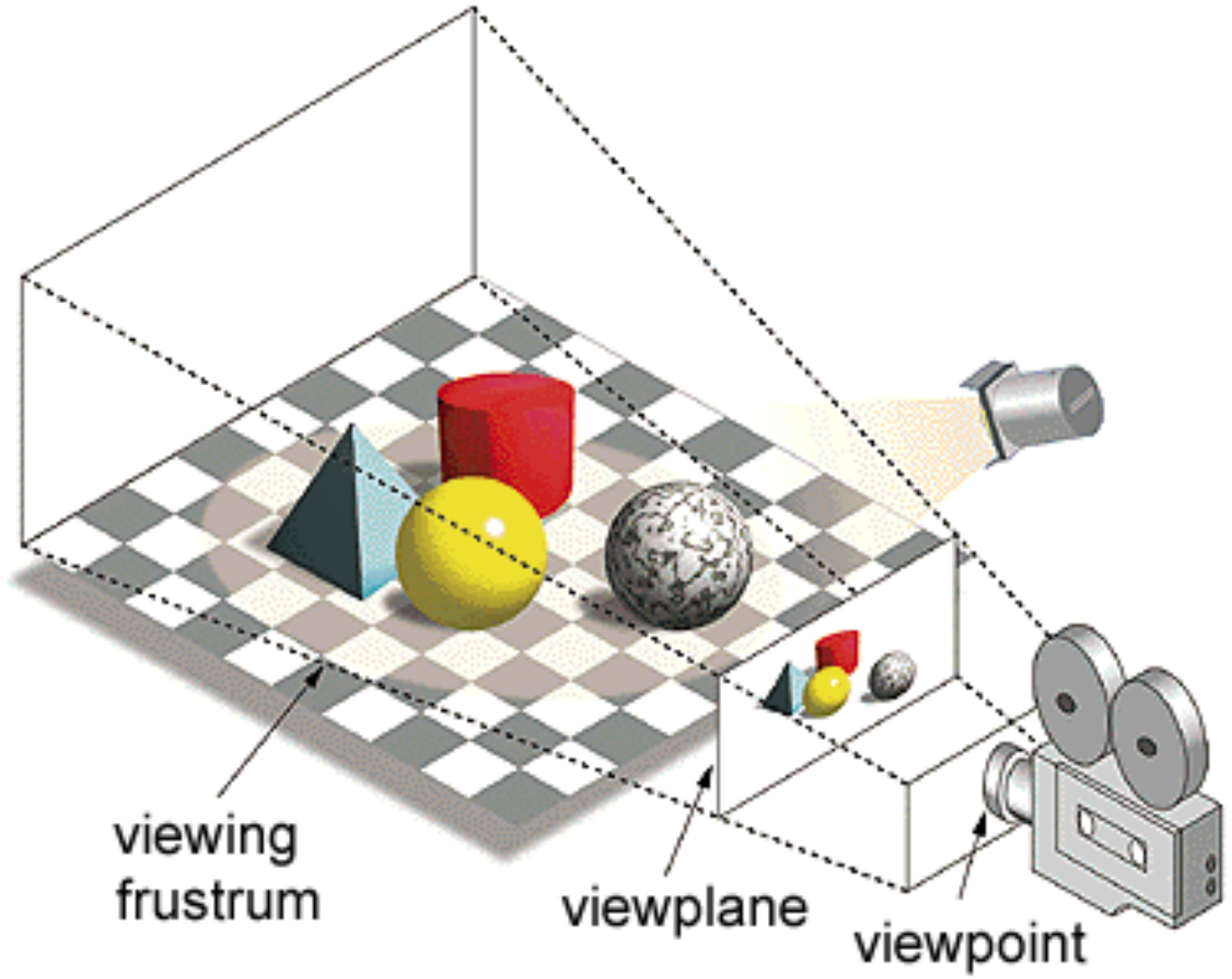
Motivation

For speed-performance, object-order algorithms are used

```
Object-Order
    for-each object
        update the pixels the object
    influences;
```

Motivation

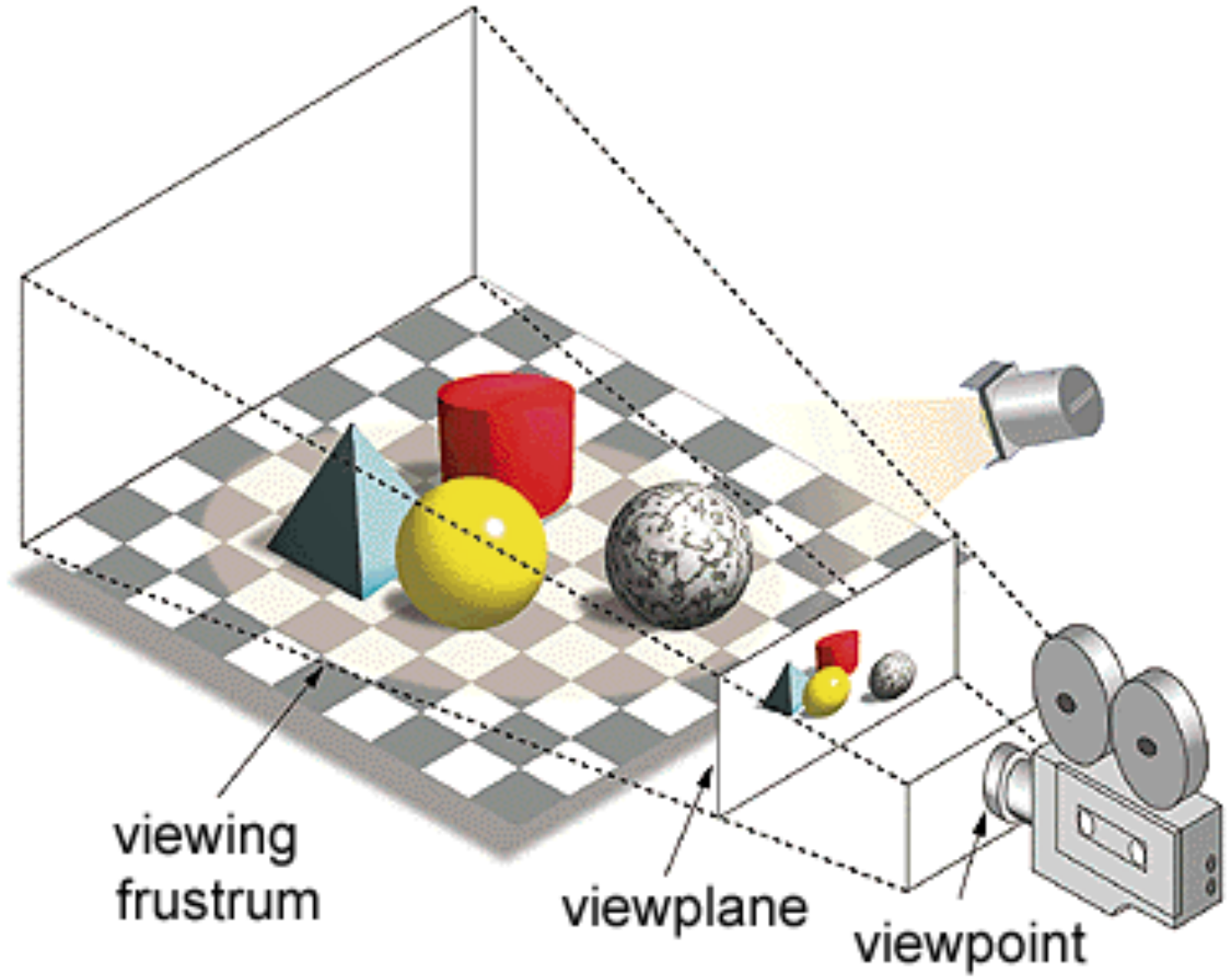
Why are they faster?



Motivation

Why are they faster?

We only need to project to a 2D plane and rasterize each object

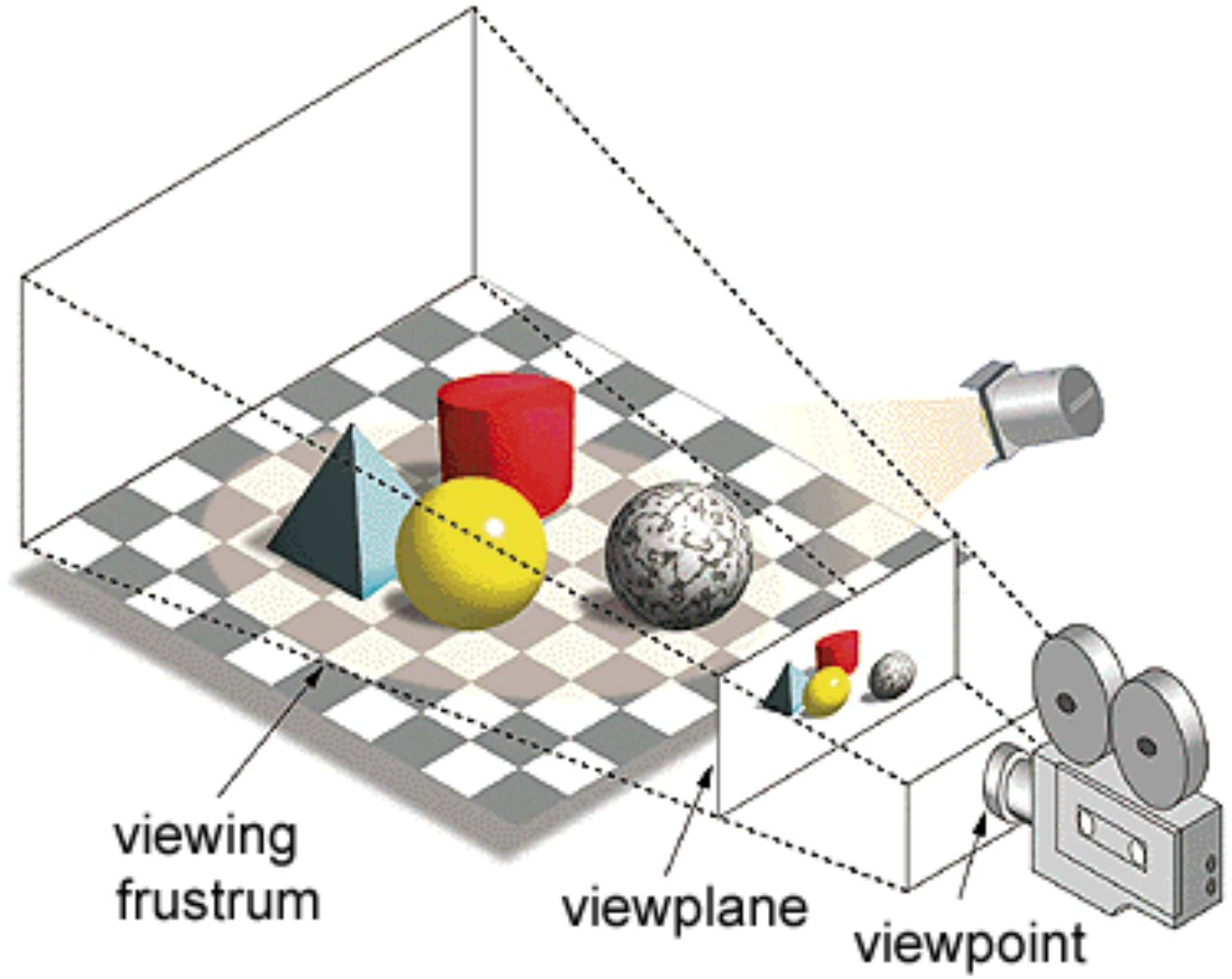


Motivation

Why are they faster?

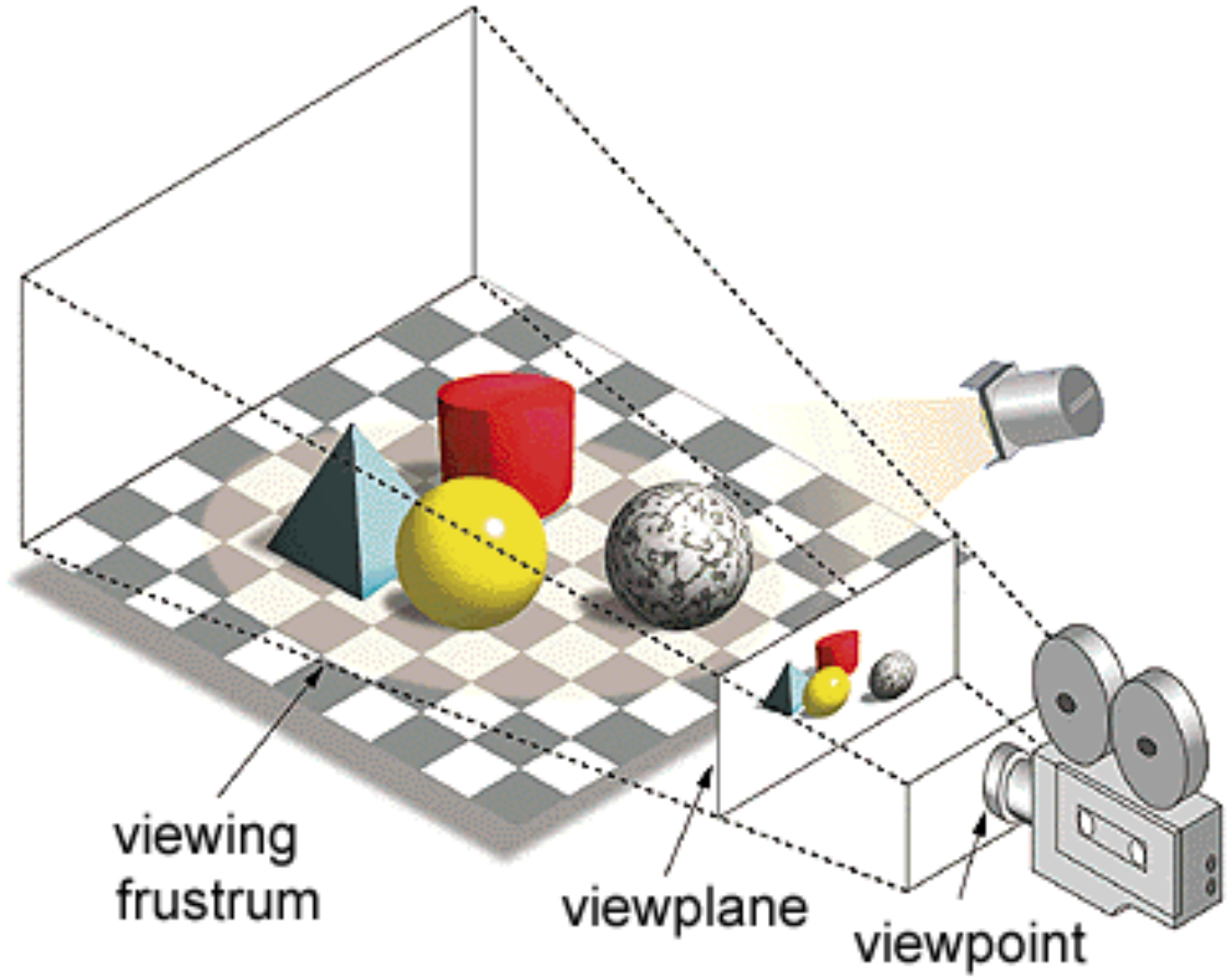
We only need to project to a 2D plane and rasterize each object

Each object tells the pixel what colour to be



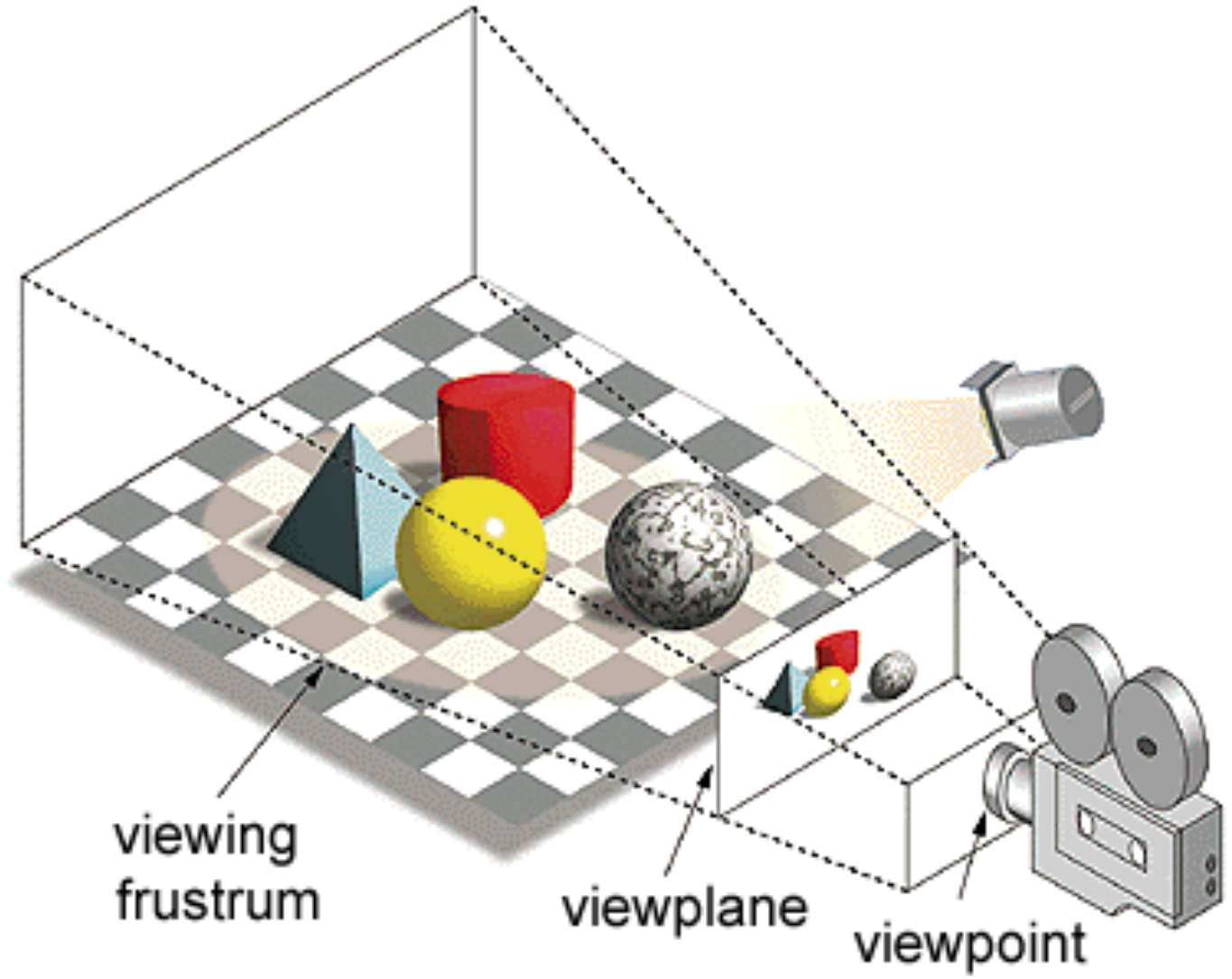
Questions:

- How do we?
  - Get each object into the world efficiently



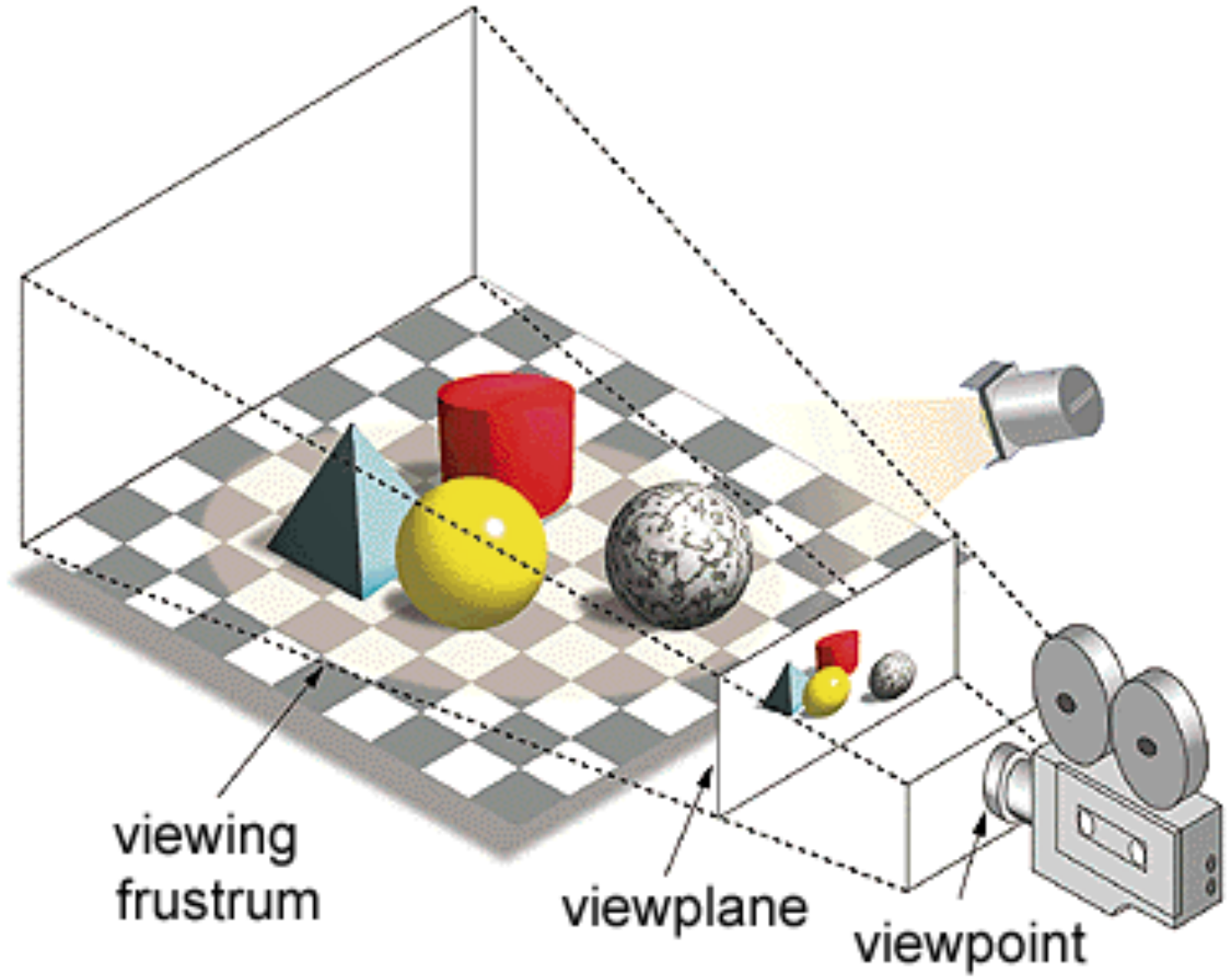
Questions:

- How do we?
  - Get each object into the world efficiently
  - Handle an arbitrary camera (not centered at the origin)



Questions:

- How do we?
  - Get each object into the world efficiently
  - Handle an arbitrary camera (not centered at the origin)
  - Project the scene to the 2D viewplane



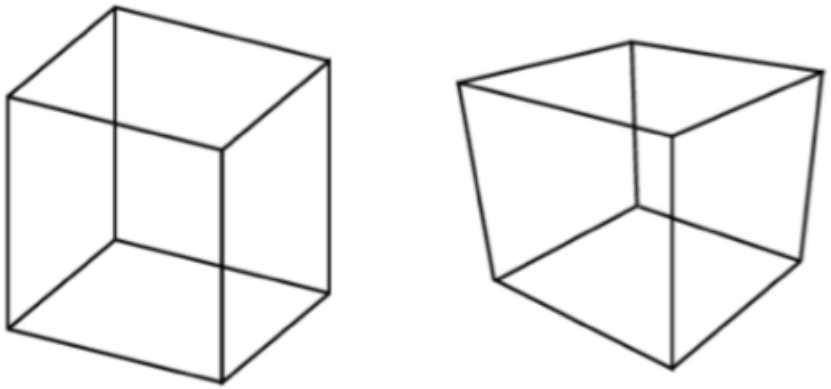
Questions: Perspective

Goal: create 2D images of 3D scenes

Standard approach: linear perspective,
i.e. straight lines in the scene become straight lines in the image
(in contrast to, e.g., fisheye views)

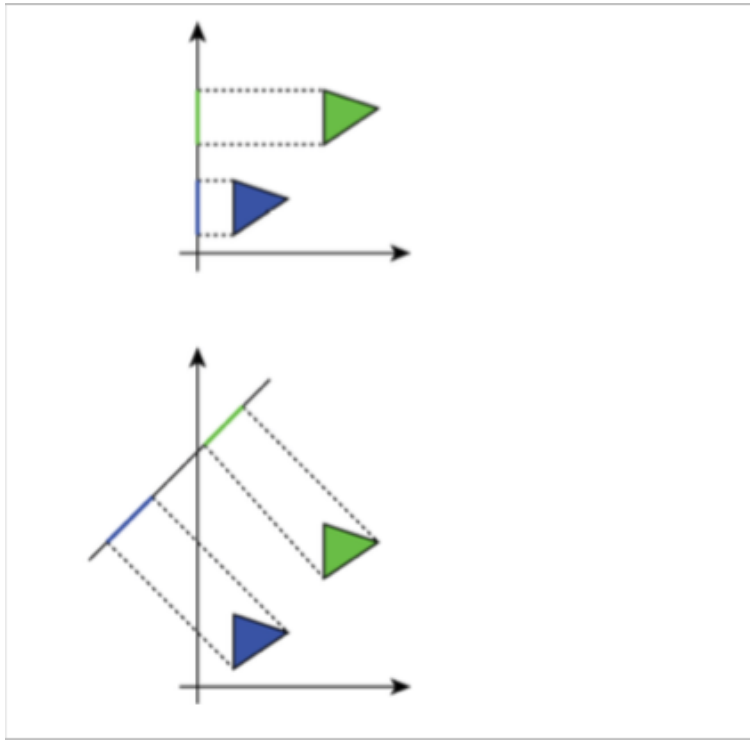
Two important distinctions:

- parallel projection
- perspective projection

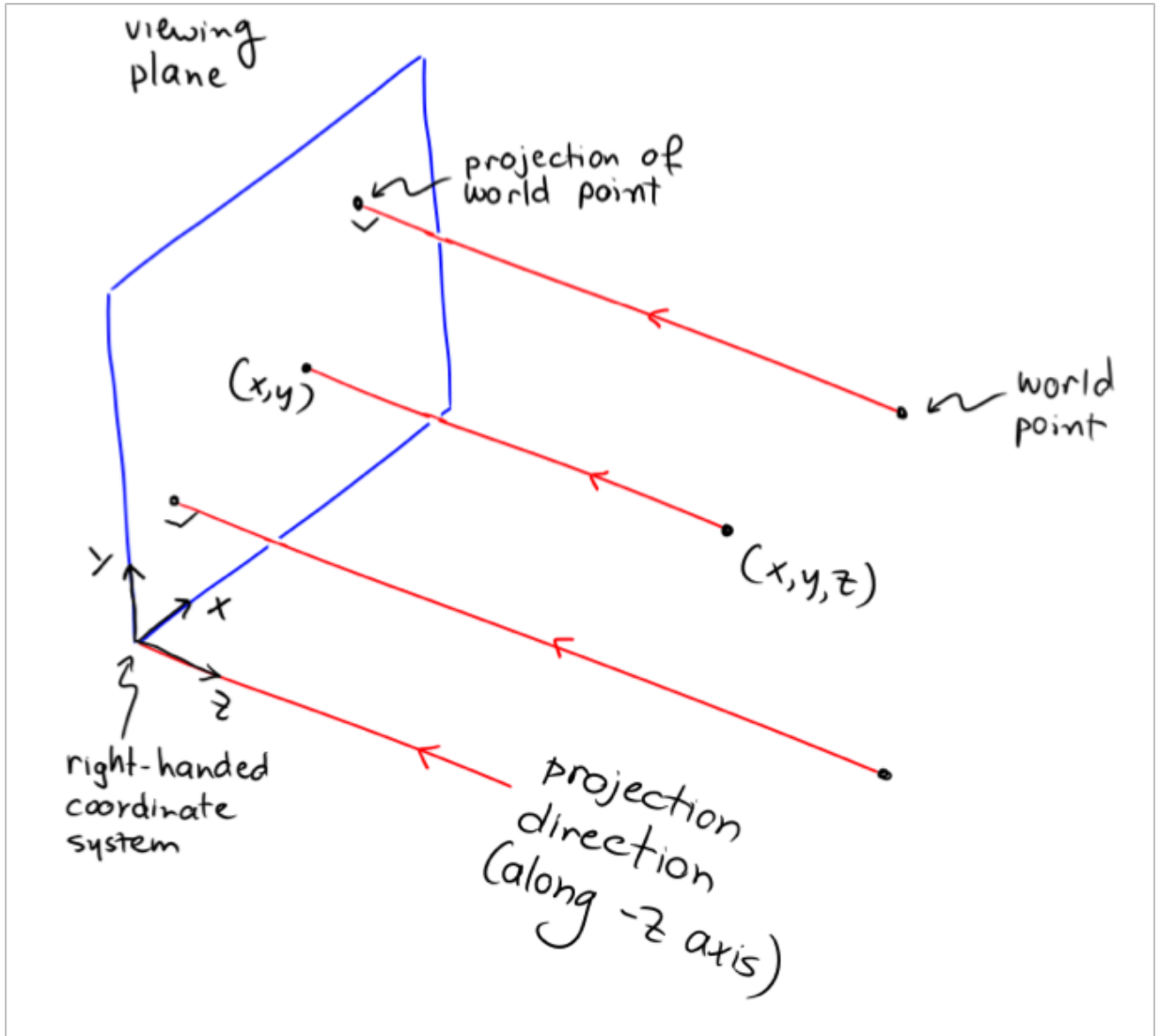


Orthographic Projection

- Orthographic Projection: Maps 3D points to 2D by moving them along a projection direction until they hit an image plane
- Characteristics:
  - Keeps parallel lines parallel
  - Preserves the size and shape of planar objects

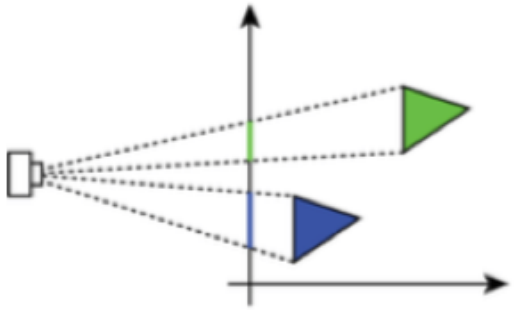


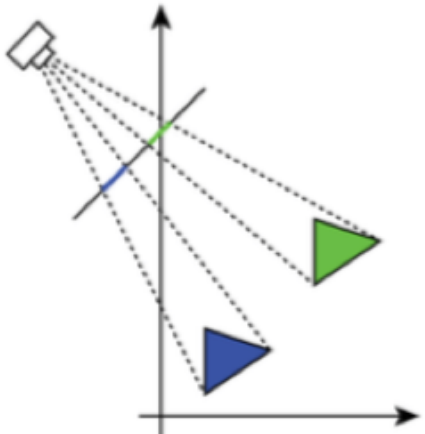
Orthographic Projection



Perspective projection

- Perspective projection: Maps 3D points to 2D by projecting them along lines that pass through a single viewpoint until they hit an image plane.
  - Characteristics:
    - Objects farther from the viewpoint naturally become smaller
- Note: Ray tracing does this “for free”

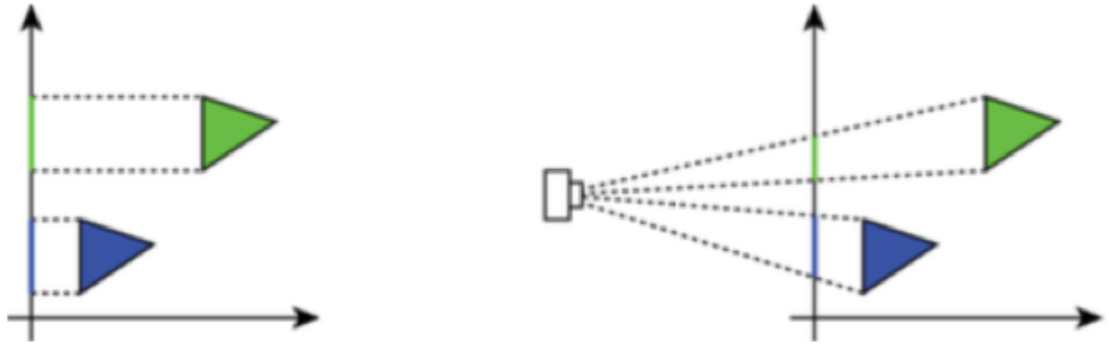





frankfurt airport tunnel (wikipedia.com)
object-to-camera distance affects an object's projection
In real photos ←parallel lines→ do converge in the distance!

Parallel vs Perspective projection

- Parallel: usage in mechanical and architectural drawings
- Perspective projection: more natural and realistic

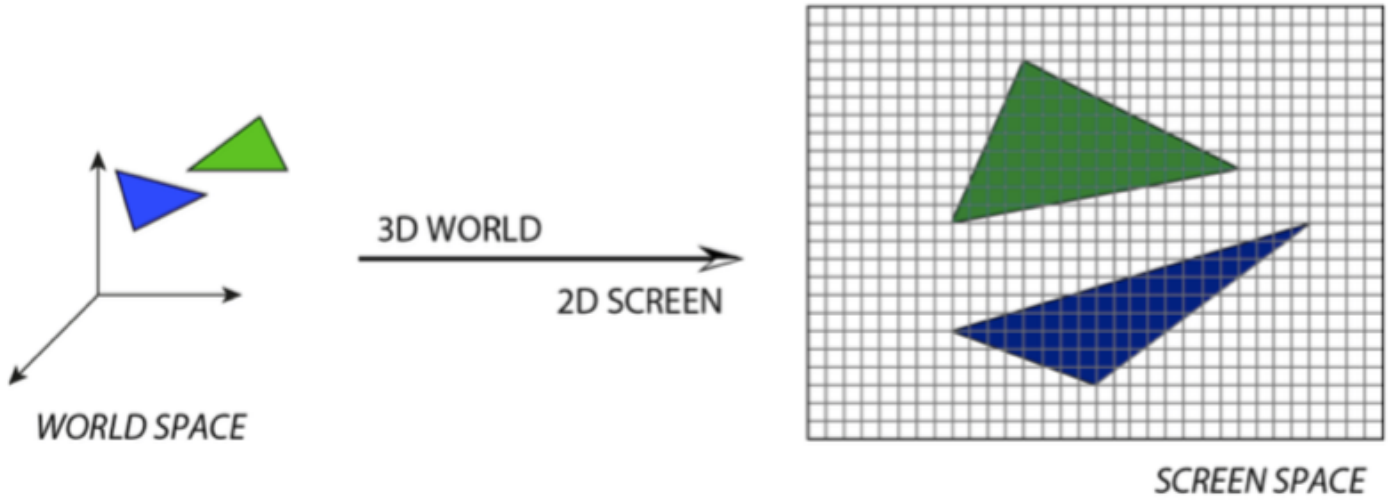


- How to get 3D objects perspectively correct on 2D screen?

Overview of the Graphics Pipeline

Perspective projection

How to get 3D objects perspectively correct on 2D screen?

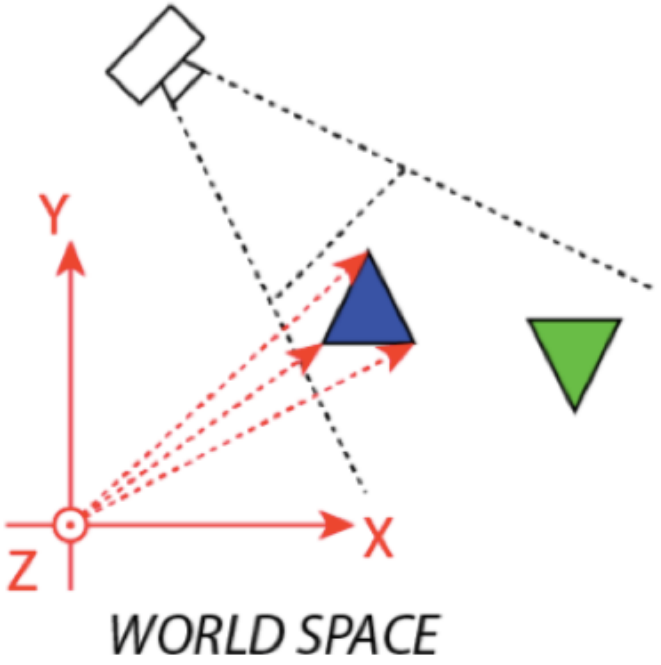


This task is best solved by splitting it in subtasks
that in turn can be solved by matrix multiplication

Let's start with what we got ...

World space

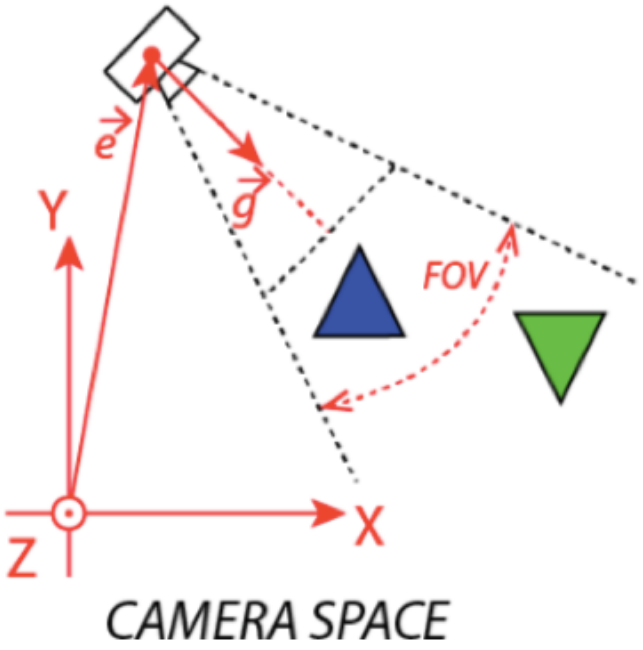
- Our 3D scene is given in world space, i.e. linear combinations of the base vectors $\vec{x}$, $\vec{y}$, and $\vec{z}$
- Given an arbitrary camera position, we want to display our 3D world in a 2D image using perspective projection



Camera position

The camera position is specified by

- the eye vector $\vec{e}$ (it's location)
- the gaze vector $\vec{g}$ (it's direction) (Also called the view vector)
- the image plane (via it's field of view (FOV) and distance from $\vec{e}$)

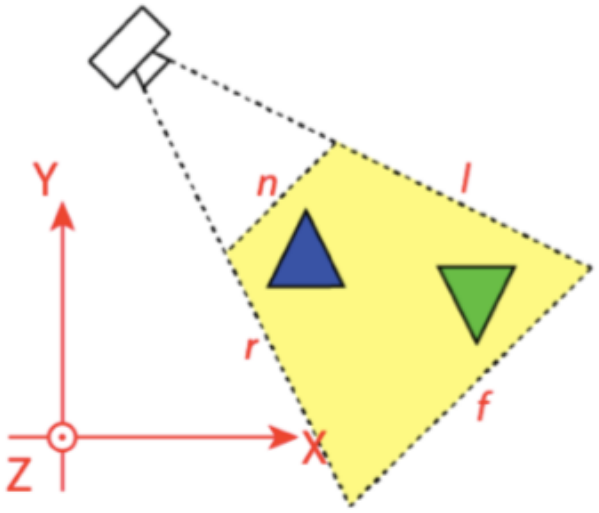


View frustum

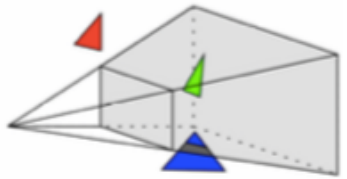
The view frustum (aka view volume) specifies everything that the camera can see. It's defined by

- the left plane l
- the right plane r
- the top plane t
- the bottom plane b
- the near plane n
- the far plane f

For now, we assume *wireframe models* that are *completely within* the view frustum



side view:

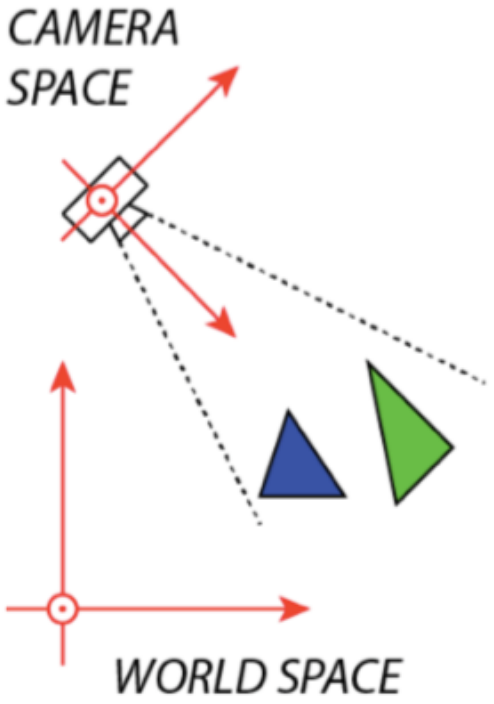


Camera transformation

Hmm, it would be much easier if the camera were at the origin . . .

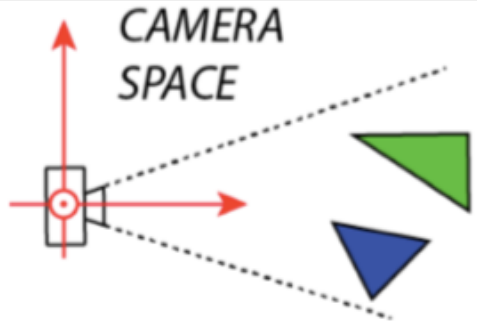
We can do that by moving from world space coordinates to camera space coordinates.

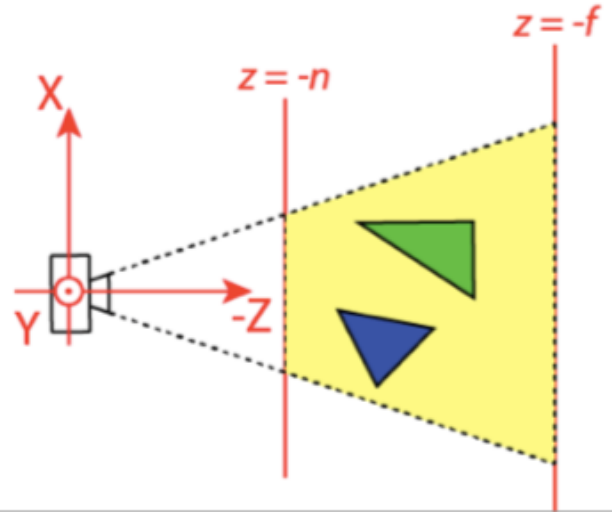
This is just a simple matrix multiplication (cf. later).



Camera transformation

Per convention, we look into the direction of the negative Z-axis



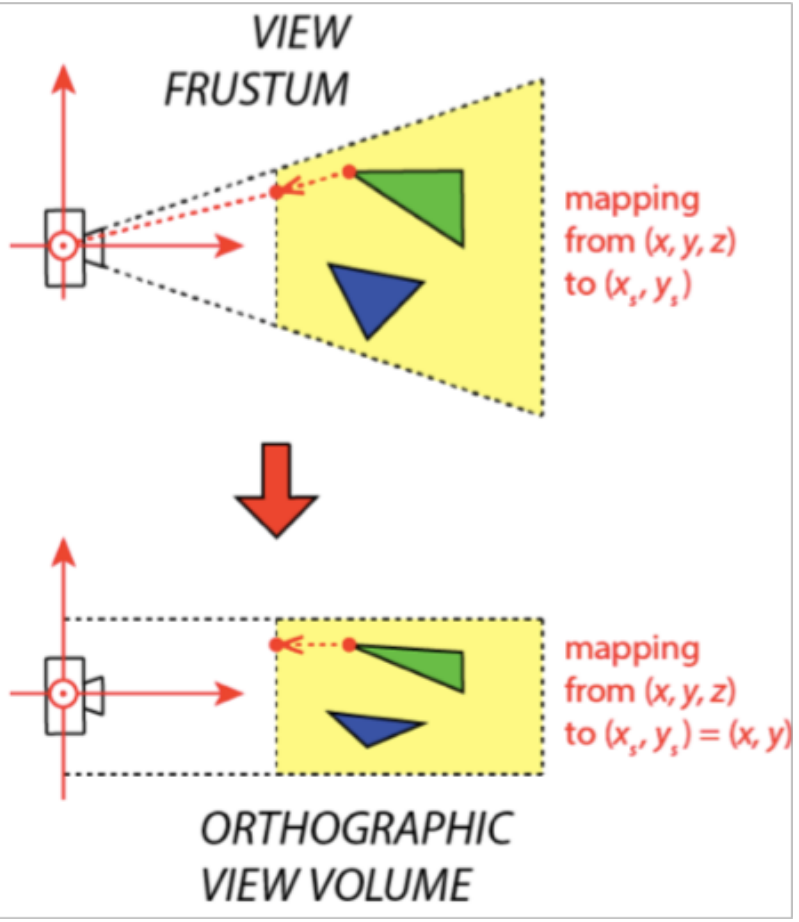


Orthographic projection

Hmm, it would be much easier if we could do orthographic projection …

We can do that by transforming the view frustum to the orthographic view volume.

Again, this is just a matrix multiplication (but this time, it's not that simple, cf. later).

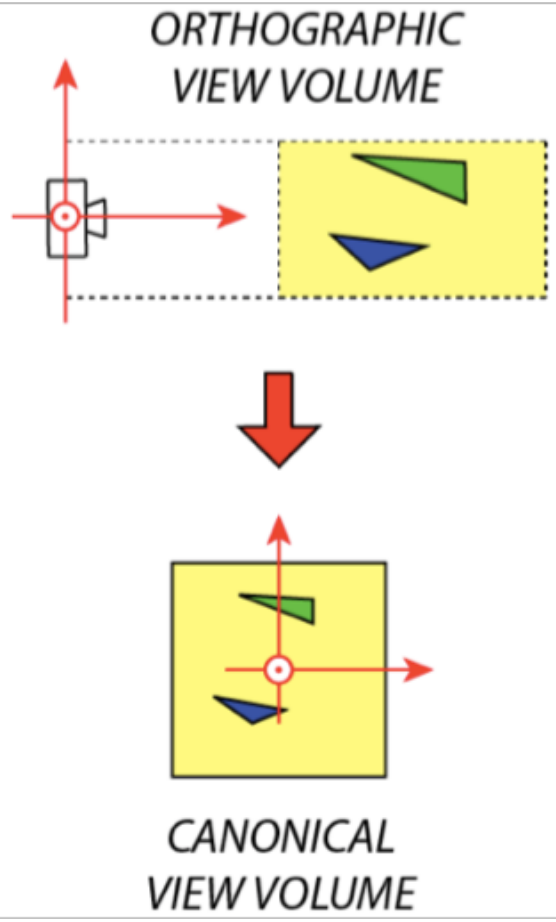


The canonical view volume

Hmm, it would be much easier if our values were between -1 and 1 . . .

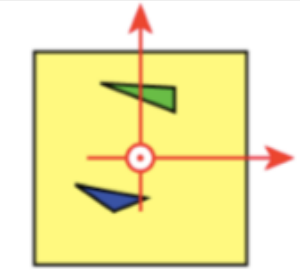
We can do that by transforming the orthographic view volume to the canonical view volume.

Again, this is just a (simple) matrix multiplication (cf. later).

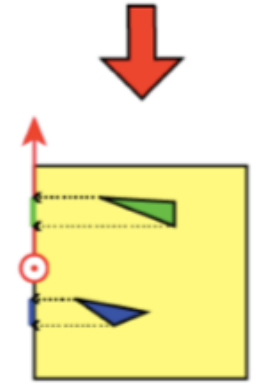


Viewport or windowing transform

Now all that's left is a parallel projection along the Z-axis (very easy) and . . .



CANONICAL VIEW VOLUME

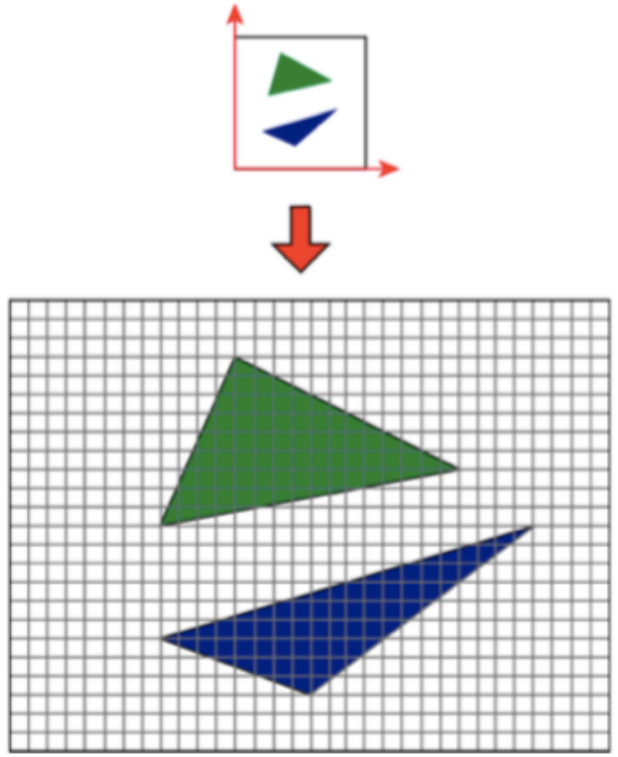


SCREEN SPACE

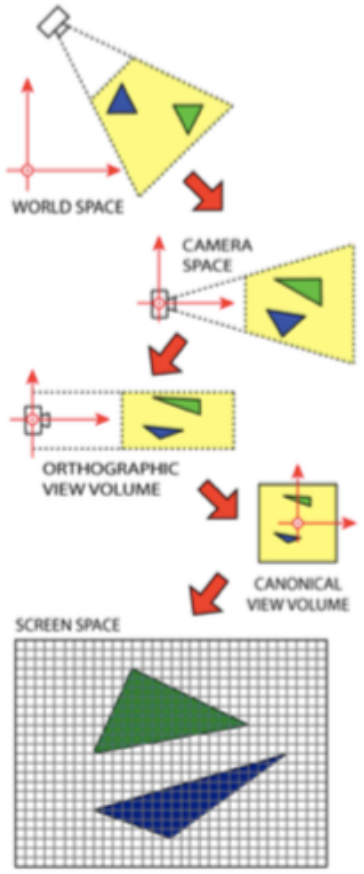
Viewport or windowing transform

. . . a windowing transformation in order to display the square $[-1, 1]^2$ onto an $n_x \times n_y$ image.

Again, these are just some (simple) matrix multiplications (cf. later).



Graphics pipeline (part 1)



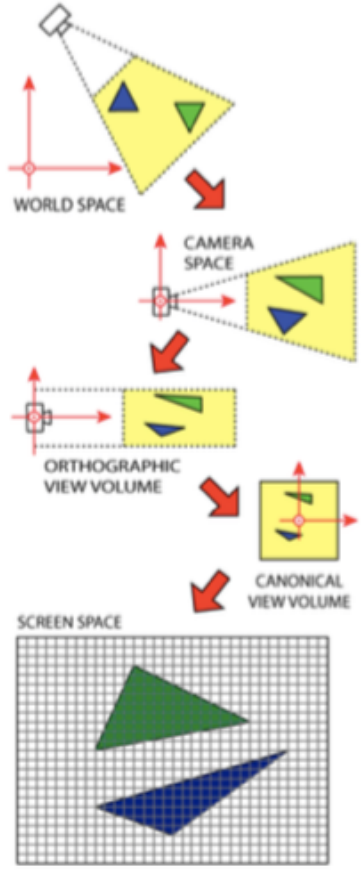
Notice that every step in this sequence can be represented by a matrix operation, so the whole process can be applied by performing a single matrix operation! (well, almost . . .)

We call this sequence a graphics pipeline = a special software or hardware subsystem that efficiently draws 3D primitives in perspective.

The Graphics Pipeline

- ~~Motivation and Questions~~
- ~~Overview~~
- Windowing Transformation
- Camera Transformation
- Perspective Transformation

Overview

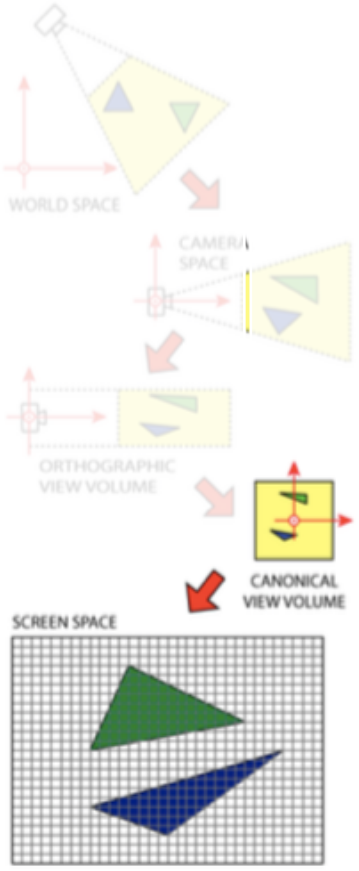


Let's start with the easier stuff, e.g.

Windowing transformation
(aka viewport transformation)

How do we get the data from the canonical view volume to the screen?

Overview



Let's start with the easier stuff, e.g.

Windowing transformation
(aka viewport transformation)

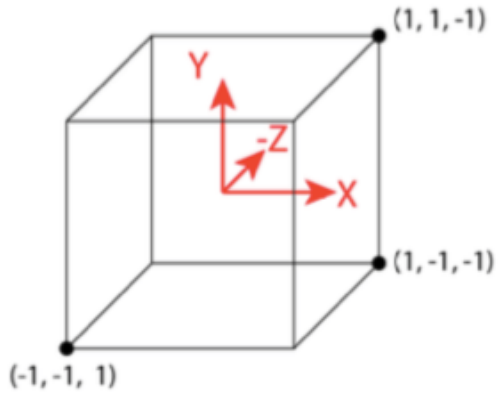
How do we get the data from the canonical view volume to the screen?

The canonical view volume

The canonical view volume is a $2 \times 2 \times 2$ box, centered at the origin.

The view frustum is transformed to this box (and the objects within the view frustum undergo the same transformation).

Vertices in the canonical view volume are orthographically projected onto an $n_x \times n_y$ image.

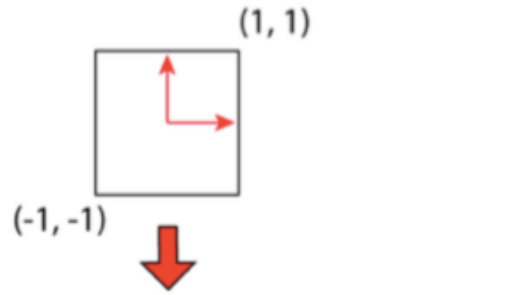


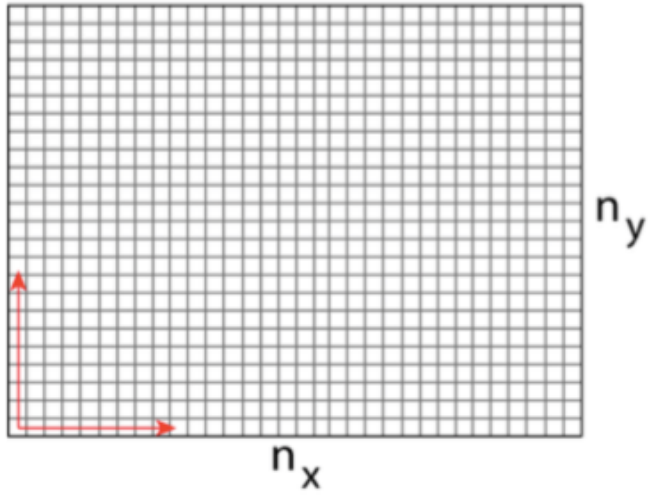
Mapping the canonical view volume

We need to map the square $[-1,1]^2$ onto a rectangle $[0,n_x] \times [0,n_y]$.

The following matrix takes care of that:

$$\begin{pmatrix} \frac{n_x}{2} & 0 & \frac{n_x}{2} \\ 0 & \frac{n_y}{2} & \frac{n_y}{2} \\ 0 & 0 & 1 \end{pmatrix}$$



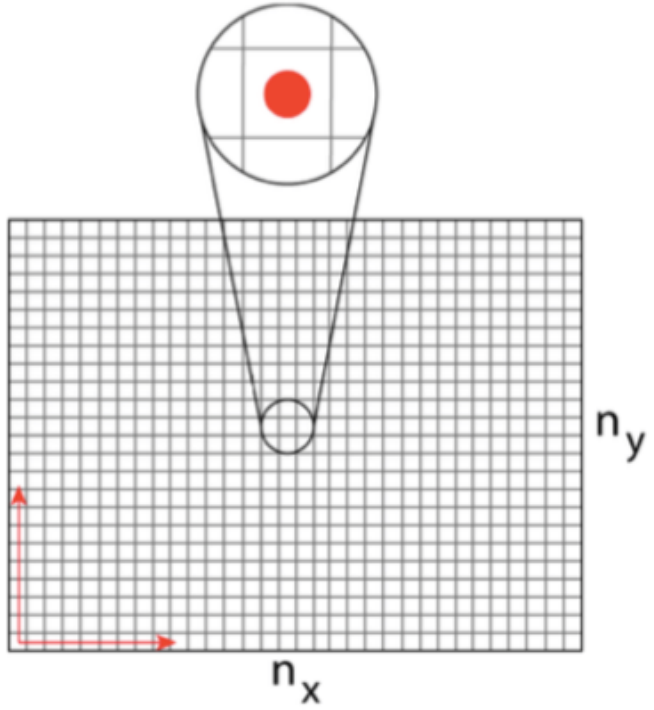


Mapping the canonical view volume

In practice, pixels represent unit squares centered at integer coordinates, so we actually have to map to the rectangle $[-\frac{1}{2}, n_x - \frac{1}{2}] \times [-\frac{1}{2}, n_y - \frac{1}{2}]$.

Hence, our matrix becomes:

$$\begin{pmatrix} \frac{n_x}{2} & 0 & \frac{n_x}{2} - \frac{1}{2} \\ 0 & \frac{n_y}{2} & \frac{n_y}{2} - \frac{1}{2} \\ 0 & 0 & 1 \end{pmatrix}$$



Mapping the canonical view volume

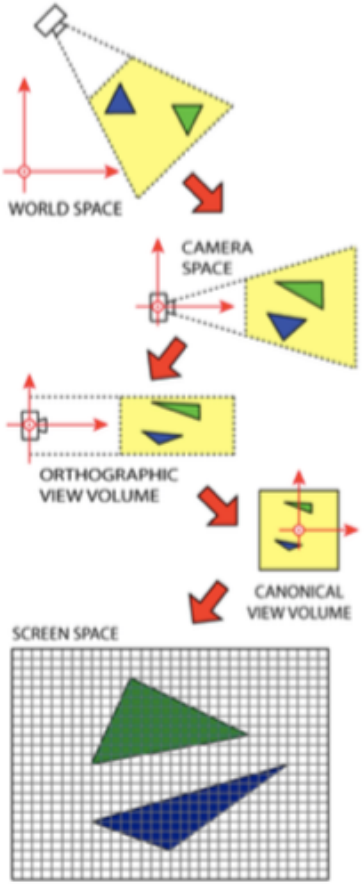
Notice that we did orthographic projection by "throwing away" the z-coordinate.

But since we want to combine all matrices in the end, we need a 4×4 matrix, so we add a row and colum that "doesn't change z".

Our final matrix for the windowing or viewport transformation is

$$M_{vp} = \begin{pmatrix} \frac{n_x}{2} & 0 & 0 & \frac{n_x}{2} - \frac{1}{2} \\ 0 & \frac{n_y}{2} & 0 & \frac{n_y}{2} - \frac{1}{2} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Overview



Hence, our last step will be

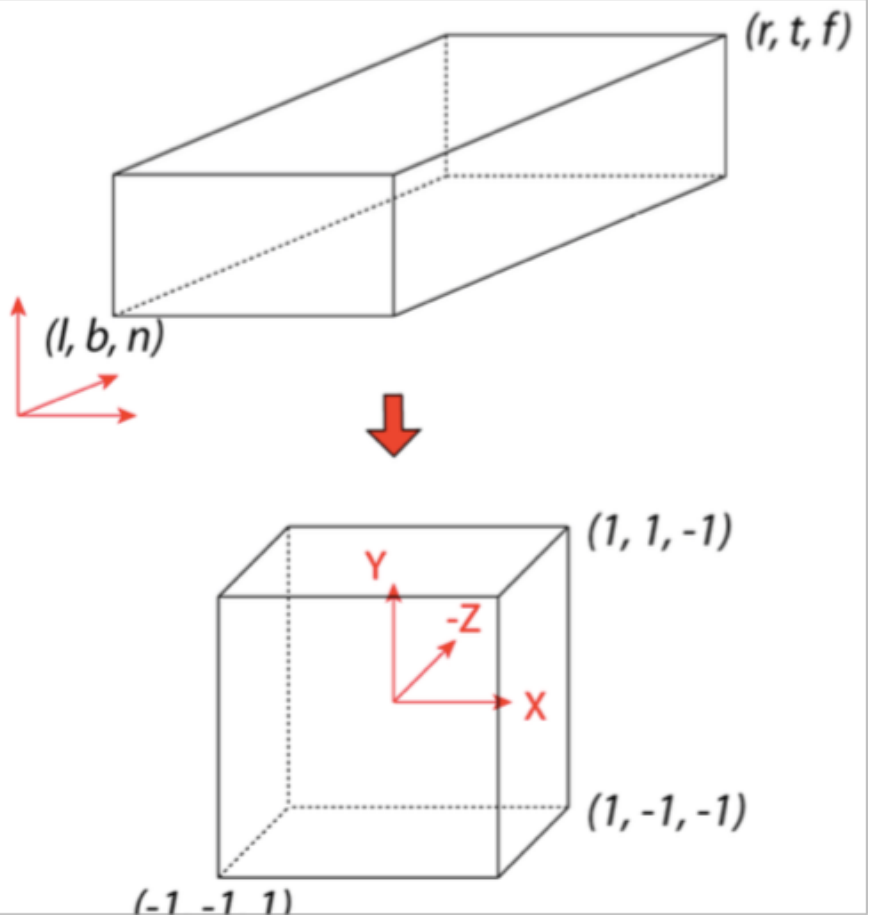
$$\begin{pmatrix} x_{screen} \\ y_{screen} \\ z_{canonical} \\ 1 \end{pmatrix} = M_{vp} \begin{pmatrix} x_{canonical} \\ y_{canonical} \\ z_{canonical} \\ 1 \end{pmatrix}$$

Ok, now let's work our way up:

How do we get the data from the orthographic view volume to the canonical view volume, i.e. ...

The orthographic view volume

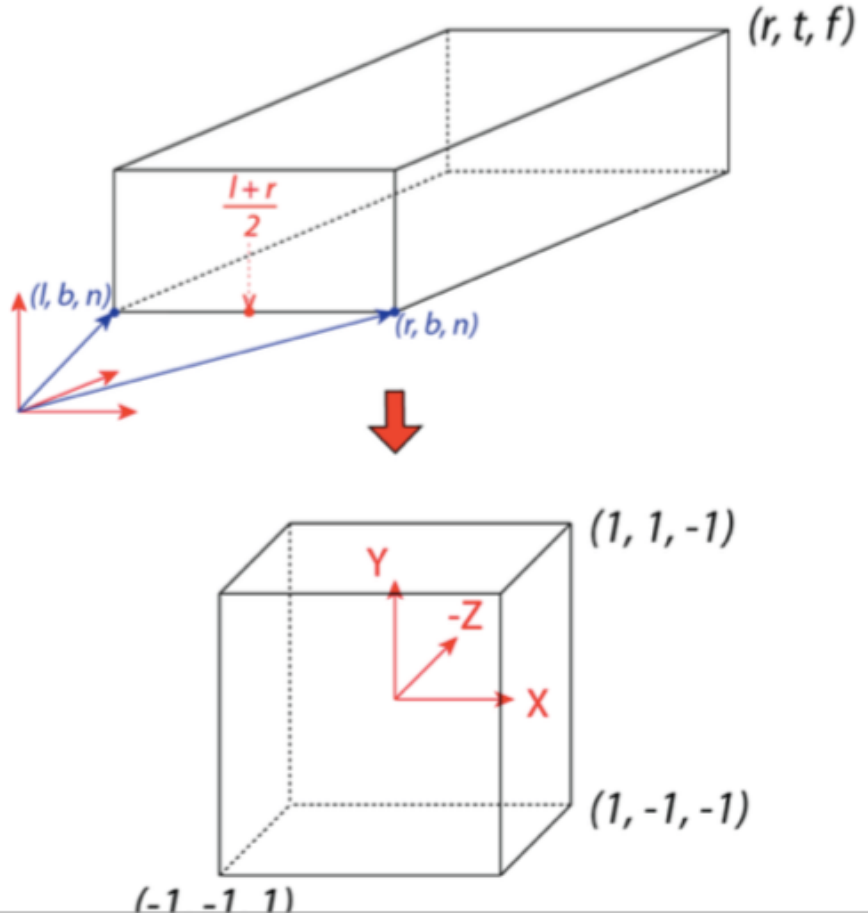
. . . how do we get the data from the axis-aligned box $[l, r] \times [b, t] \times [n, f]$ to a $2 \times 2 \times 2$ box around the origin?



The orthographic view volume

First we need to move the center to the origin:

$$\begin{pmatrix} 1 & 0 & 0 & -\frac{l+r}{2} \\ 0 & 1 & 0 & -\frac{b+t}{2} \\ 0 & 0 & 1 & -\frac{n+f}{2} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

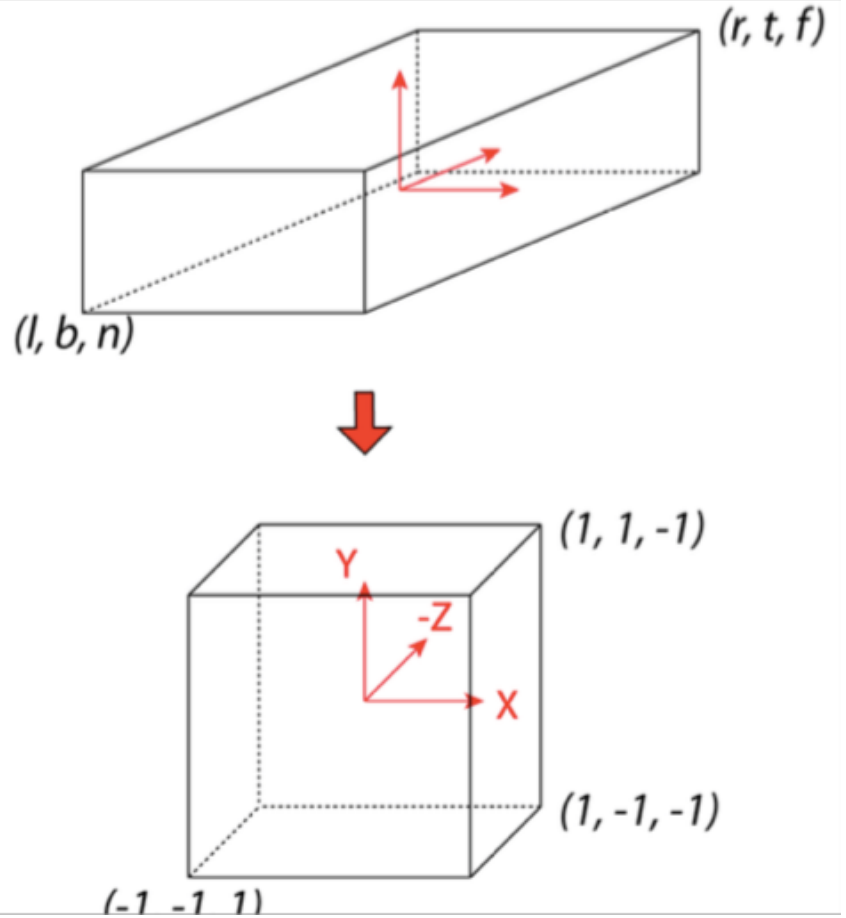


The orthographic view volume

Then we have to scale everything to $[-1, 1]$:

$$\begin{pmatrix} \frac{2}{r-l} & 0 & 0 & 0 \\ 0 & \frac{2}{t-b} & 0 & 0 \\ 0 & 0 & \frac{2}{n-f} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Note: we divide by the length, e.g. $\frac{1}{\frac{r-l}{2}}$ for the $x-$coordinate value



The orthographic view volume

Since these are just matrix multiplications (associative!), we can combine them into one matrix:

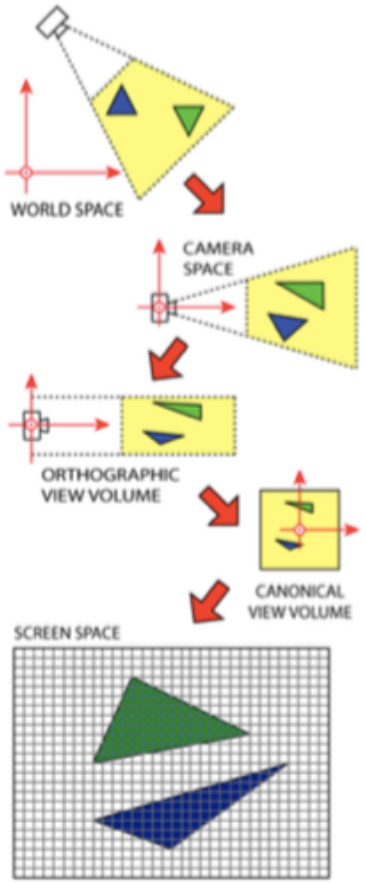
$$M_{orth} = \begin{pmatrix} \frac{2}{r-l} & 0 & 0 & 0 \\ 0 & \frac{2}{t-b} & 0 & 0 \\ 0 & 0 & \frac{2}{n-f} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & -\frac{l+r}{2} \\ 0 & 1 & 0 & -\frac{b+t}{2} \\ 0 & 0 & 1 & -\frac{n+f}{2} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$= \begin{pmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{2}{n-f} & -\frac{n+f}{n-f} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

The Graphics Pipeline

- ~~Motivation and Questions~~
- ~~Overview~~
- ~~Windowing Transformation~~
- Camera Transformation
- Perspective Transformation

Overview



Hence, our last step becomes

$$\begin{pmatrix} x_{pixel} \\ y_{pixel} \\ z_{canonical} \\ 1 \end{pmatrix} = M_{vp} M_{orth} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

Now, how do we get the data in the orthographic view volume?

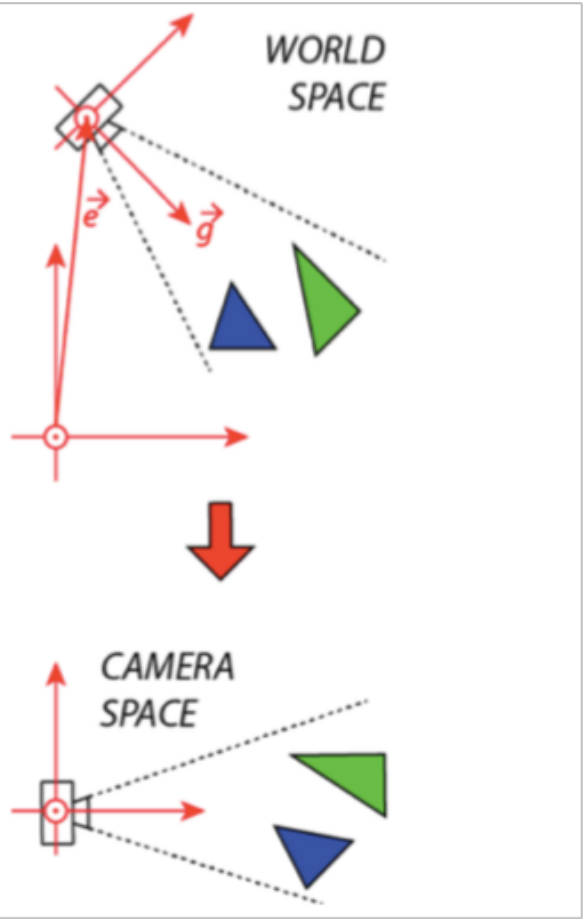
That's more difficult, so let's look at camera transformation first.

Aligning coordinate systems

How do we get the camera to the origin, i.e. how do we move from world space to camera space?

Remember:

- **world space** is expressed by the base vectors $\vec{x}$, $\vec{y}$, and $\vec{z}$
- the **camera** is specified by eye vector $\vec{e}$ and gaze vector $\vec{g}$

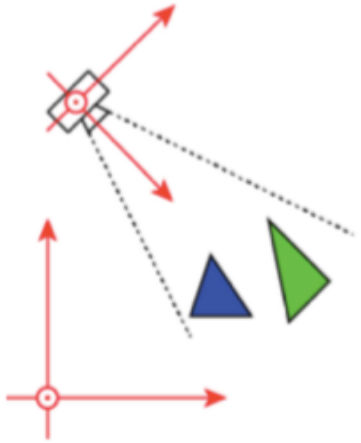


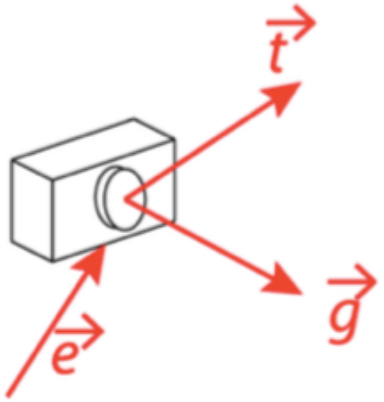
Aligning coordinate systems

To map one space to another, we need a coordinate system for both spaces.

We can easily get that using a view up vector $\vec{t}$, i.e. a vector in the plane bisecting the viewer's head into left and right halves and "pointing to the sky"

This gives us an orthonormal base $(\vec{u}, \vec{v}, \vec{w})$ of our camera coordinate system (how?)





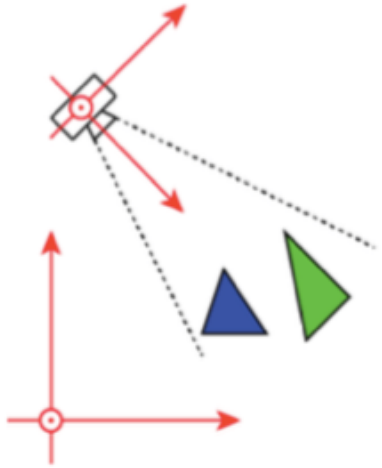
Aligning coordinate systems

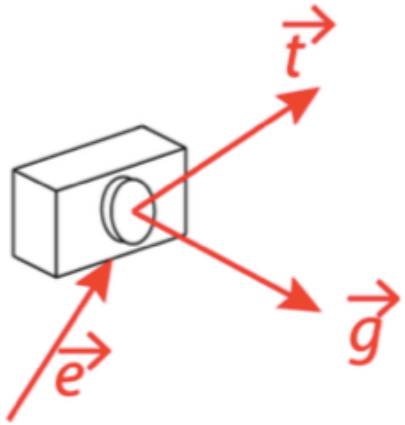
First base vector: $\vec{t} \times \vec{g} = \vec{u}$

2nd base vector: $\vec{g} \times \vec{u} = \vec{v}$

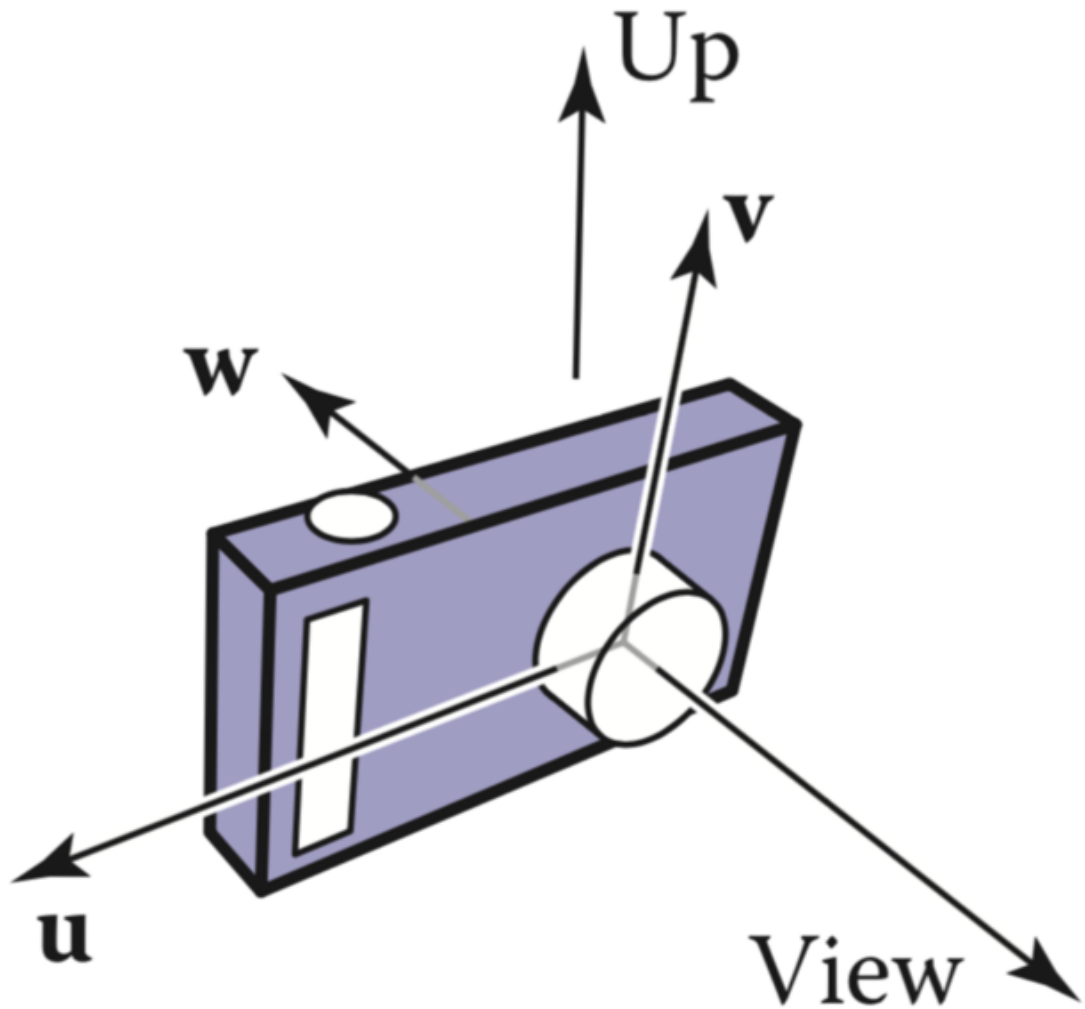
3rd base vector: $-\vec{g} =: \vec{w}$
("-" for looking in negative $z-$direction)

Don't forget to normalize, i.e. multiply with $\frac{1}{\|\cdot\|}$.





The Camera

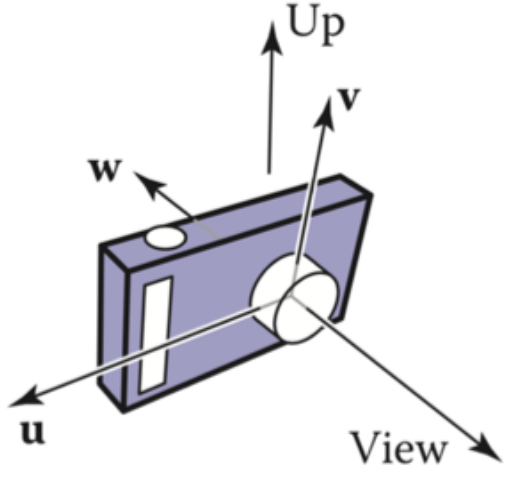


The Camera

$$\mathbf{w} = -\frac{\text{View}}{\|\text{View}\|}$$

$$\mathbf{u} = \text{View} \times \text{Up}$$

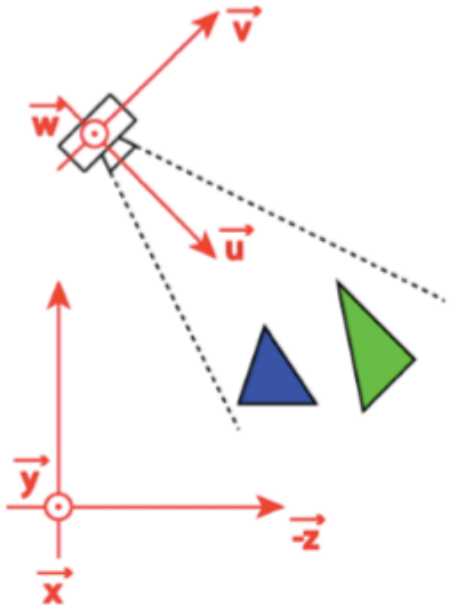
$$\mathbf{v} = \mathbf{w} \times \mathbf{u}$$



Aligning coordinate systems

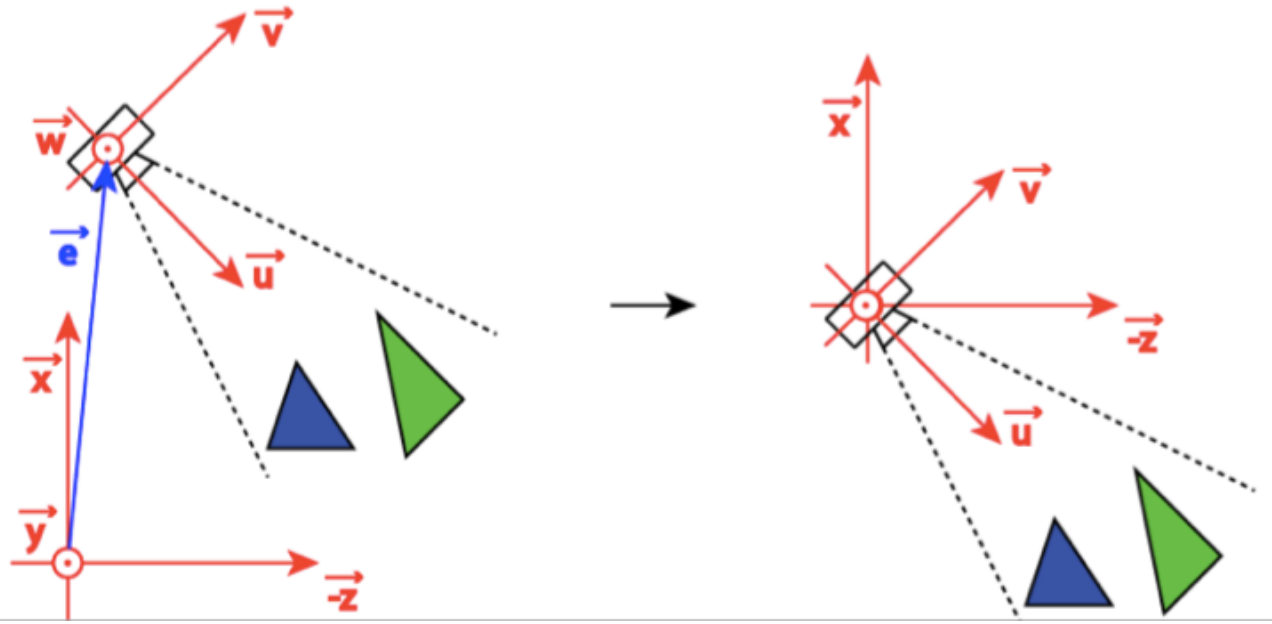
How do we align the two coordinate systems?

1. align the origins
2. align the base vectors



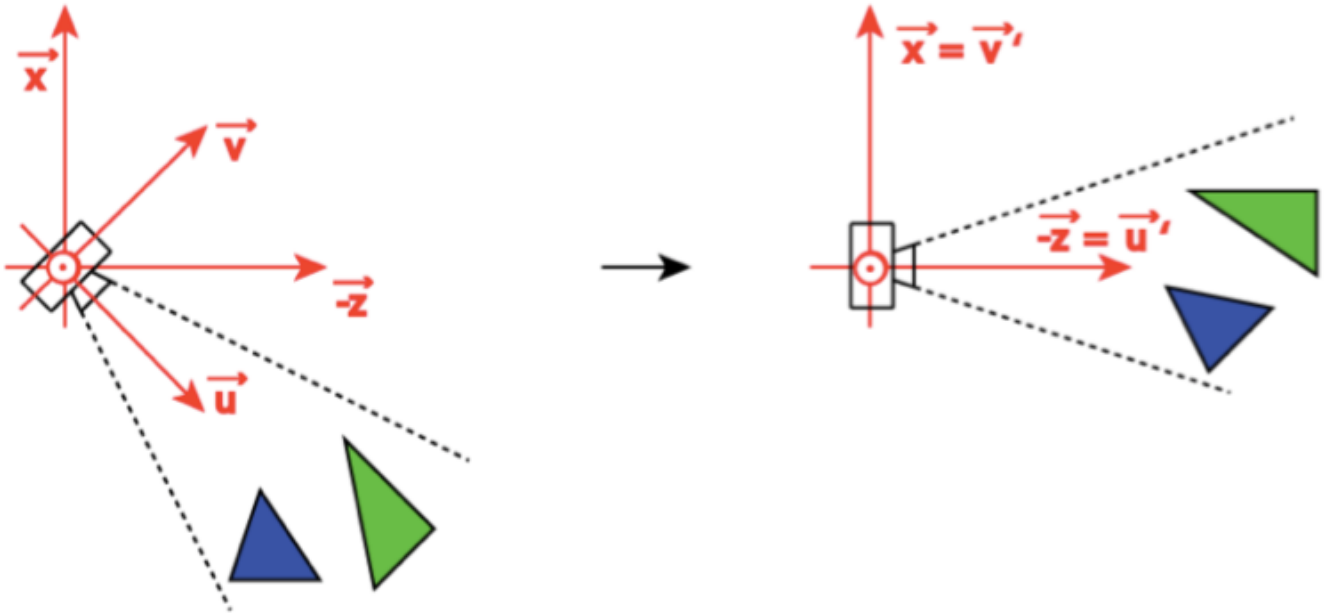
Aligning coordinate systems

Aligning the origins is a simple translation: $\begin{pmatrix} 1 & 0 & 0 & -x_e \\ 0 & 1 & 0 & -y_e \\ 0 & 0 & 1 & -z_e \\ 0 & 0 & 0 & 1 \end{pmatrix}$



Aligning coordinate systems

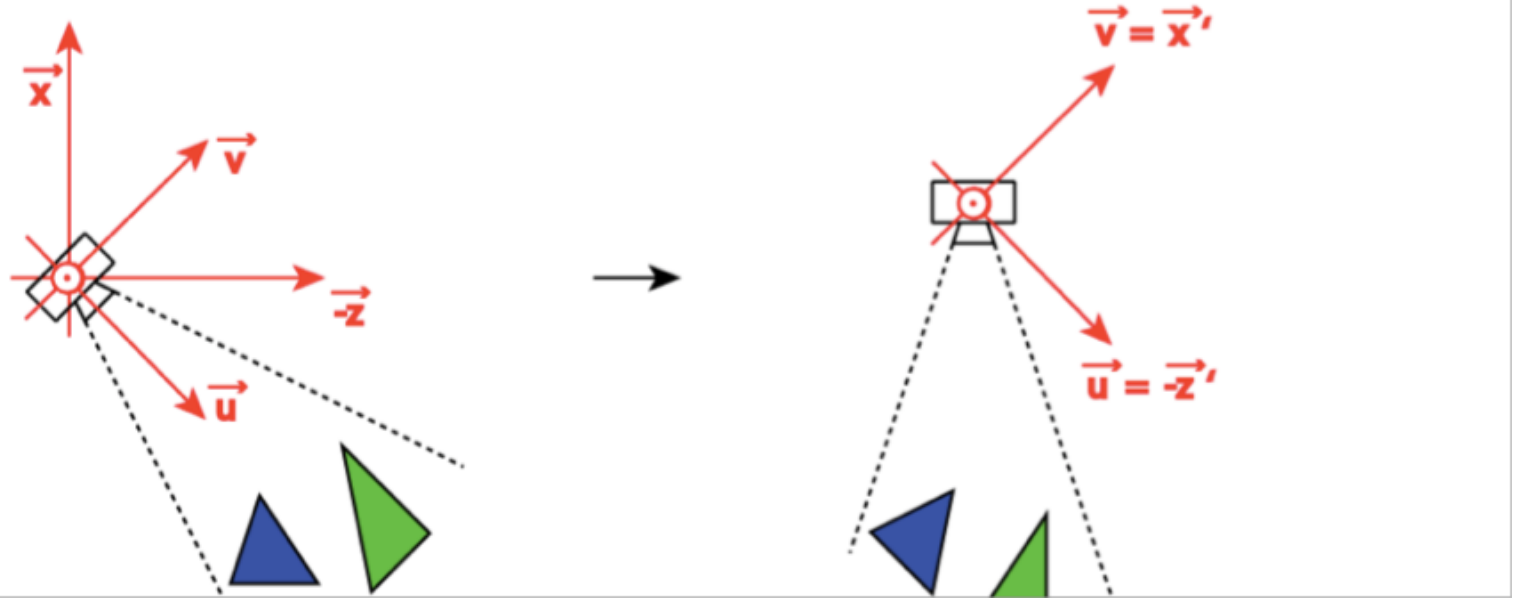
Aligning the axes is a simple rotation, if you remember that the columns of our matrix are just the images of the base vectors under the linear transformation.



Aligning coordinate systems

These are easy to find for the reverse rotation:

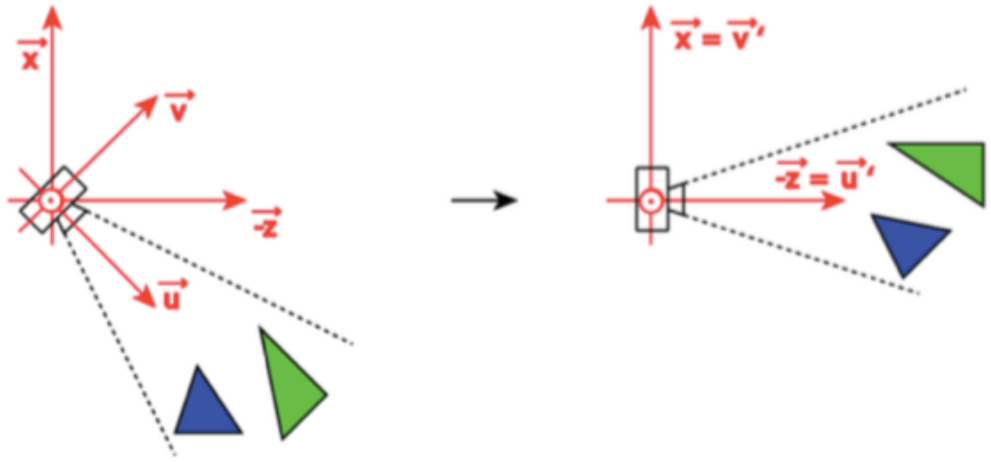
$$\begin{pmatrix} x_u & x_v & x_w & 0 \\ y_u & y_v & y_w & 0 \\ z_u & z_v & z_w & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$



Aligning coordinate systems

Hence, our rotation matrix is:

$$\begin{pmatrix} x_u & y_u & z_u & 0 \\ x_v & y_v & z_v & 0 \\ x_w & y_w & z_w & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

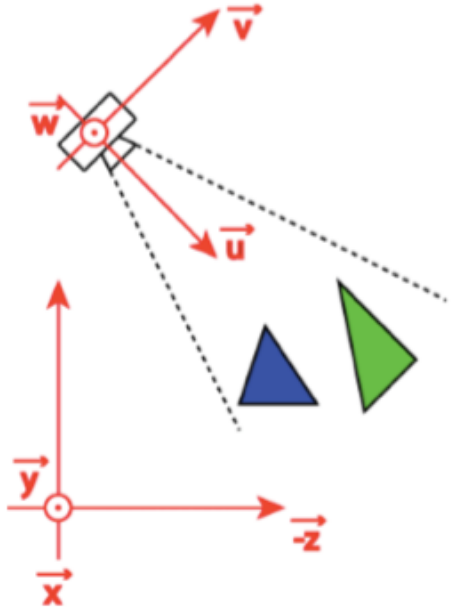


(Remember: the inverse of an orthogonal matrix is always its transposed)

Aligning coordinate systems

For the total transformation we get

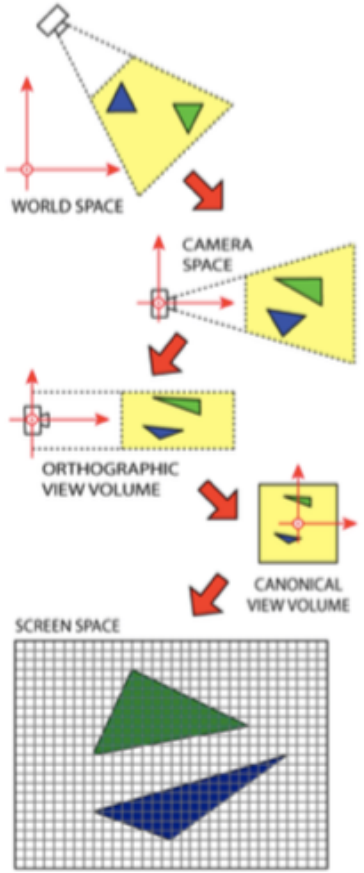
$$M_{cam} = \begin{pmatrix} x_u & y_u & z_u & 0 \\ x_v & y_v & z_v & 0 \\ x_w & y_w & z_w & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & -x_e \\ 0 & 1 & 0 & -y_e \\ 0 & 0 & 1 & -z_e \\ 0 & 0 & 0 & 1 \end{pmatrix}$$



The Graphics Pipeline

- ~~Motivation and Questions~~
- ~~Overview~~
- ~~Windowing Transformation~~
- ~~Camera Transformation~~
- Perspective Transformation

Overview



If it wasn't for perspective projection, we'd be done:

$$\begin{pmatrix} x_{pixel} \\ y_{pixel} \\ z_{canonical} \\ 1 \end{pmatrix} = M_{vp} M_{orth} M_{cam} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

Parallel vs perspective projection

With this, we could already do some nice stuff using orthographic projection, e.g. funny games:

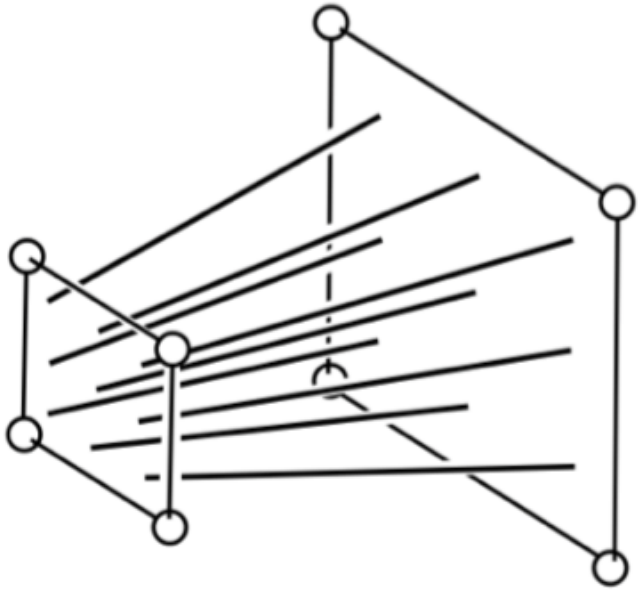
(from "The Simpsons Tapped Out" game)

Yet, for realistic graphics, we need to put things into perspective . . .

Transforming the view frustum

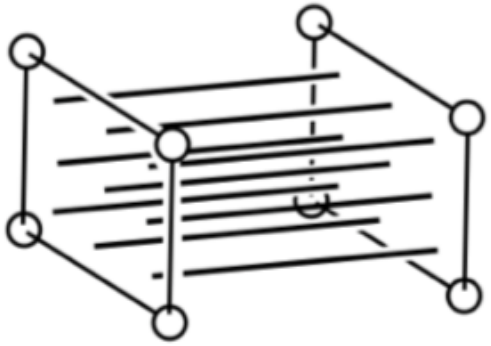
cf. book, fig. 7.13 (3rd ed.) or 7.12 (2nd ed.)

View frustum



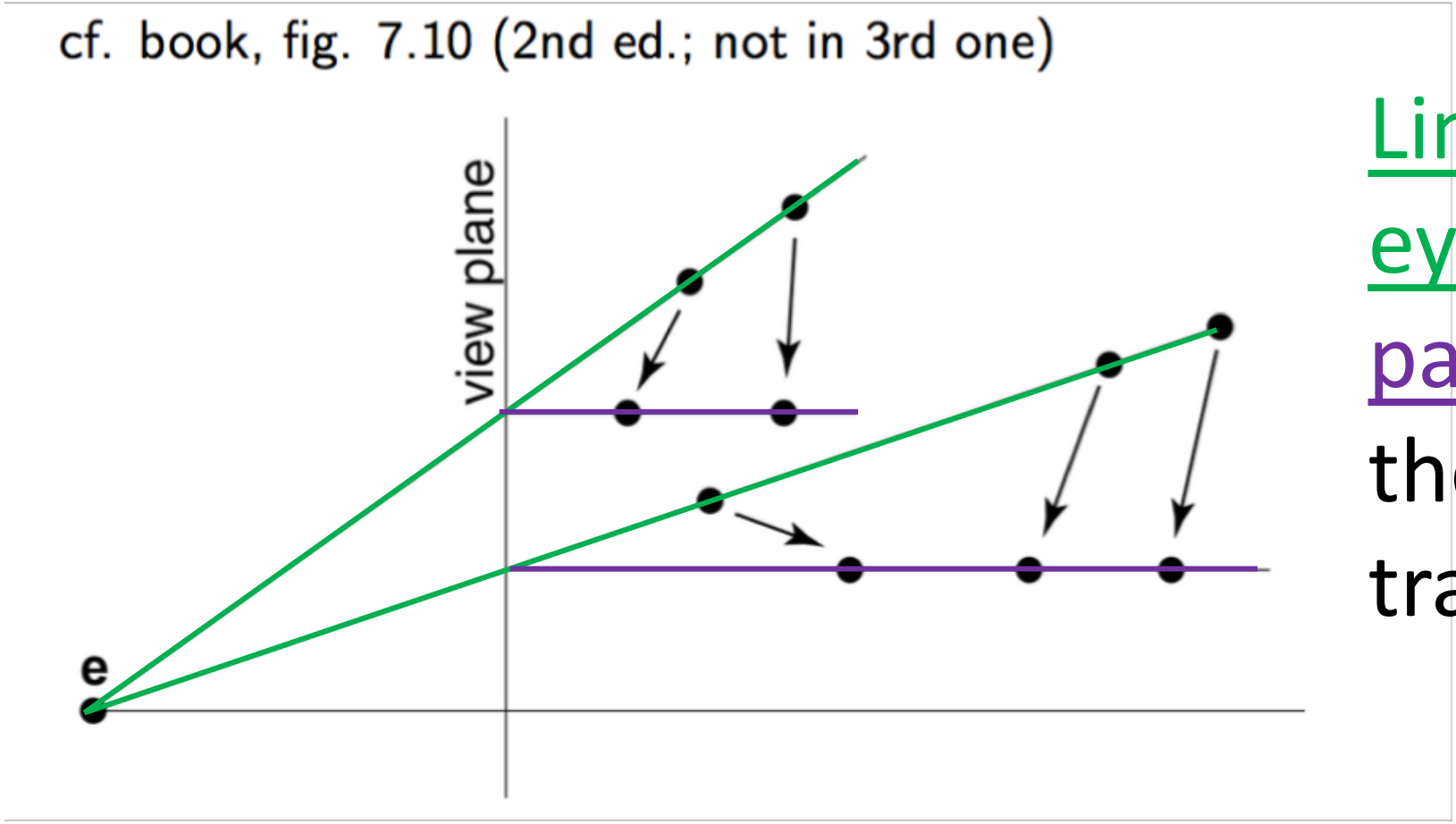
Perspective projection

Orthographic view volume



Parallel/orthographic projection

Transforming the view frustum

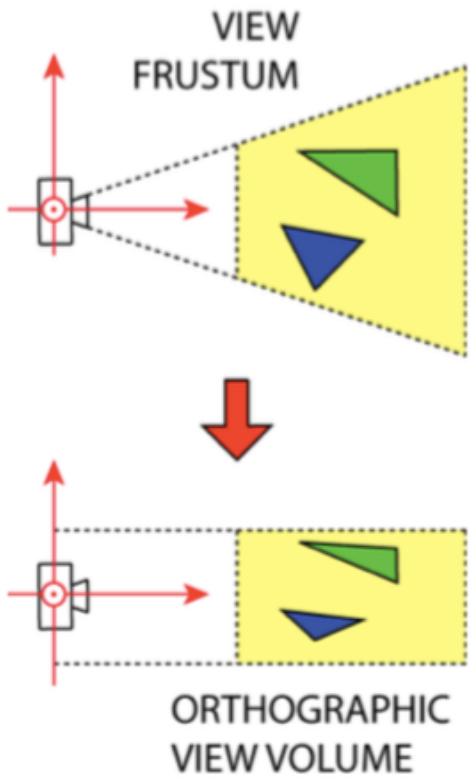


Lines through the eye become parallel lines after the transformation

Transforming the view frustum

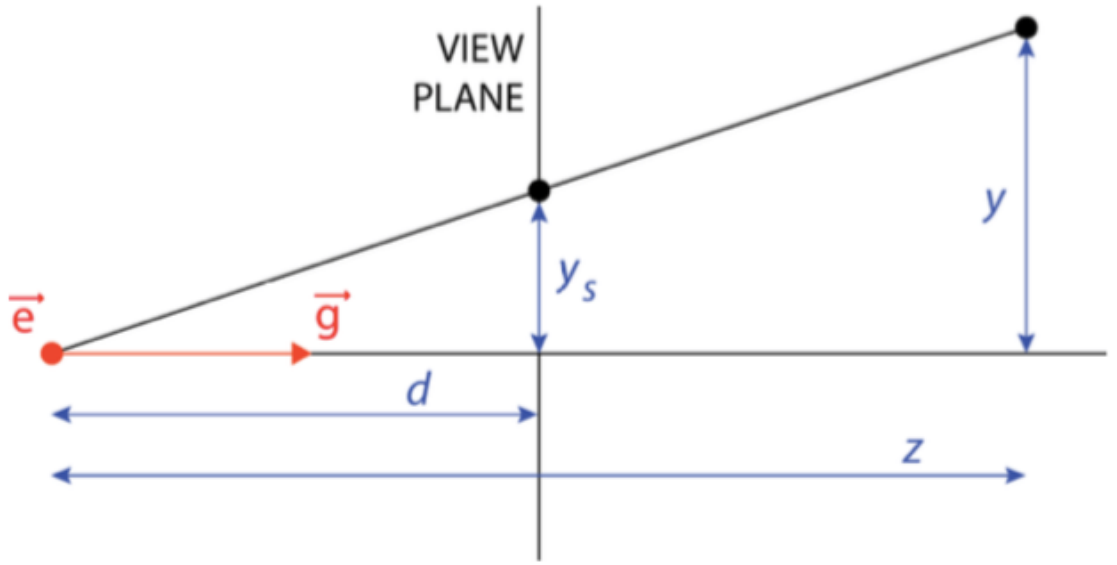
We have to transform the view frustum into the orthographic view volume. The transformation needs to

- Map lines through the origin to lines parallel to the z axis
- Map points on the viewing plane to themselves.
- Map points on the far plane to (other) points on the far plane.
- Preserve the near-to-far order of points on a line.



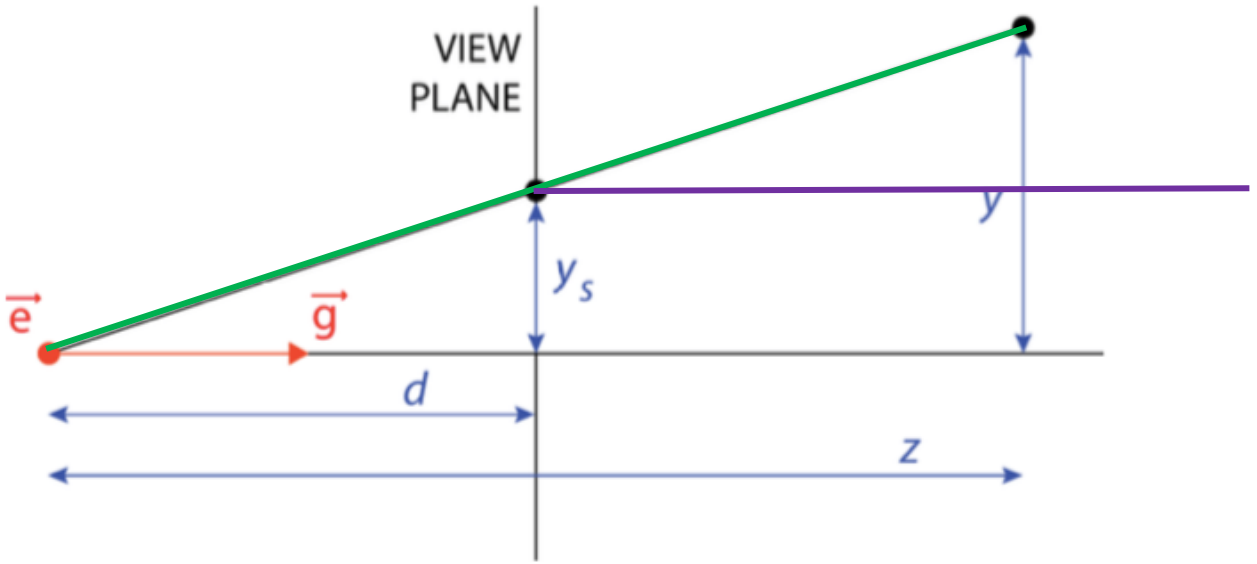
Transforming the view frustum

How do we calculate this? (cf. book, fig. 7.8/7.9 (3rd/2nd ed.))



Transforming the view frustum

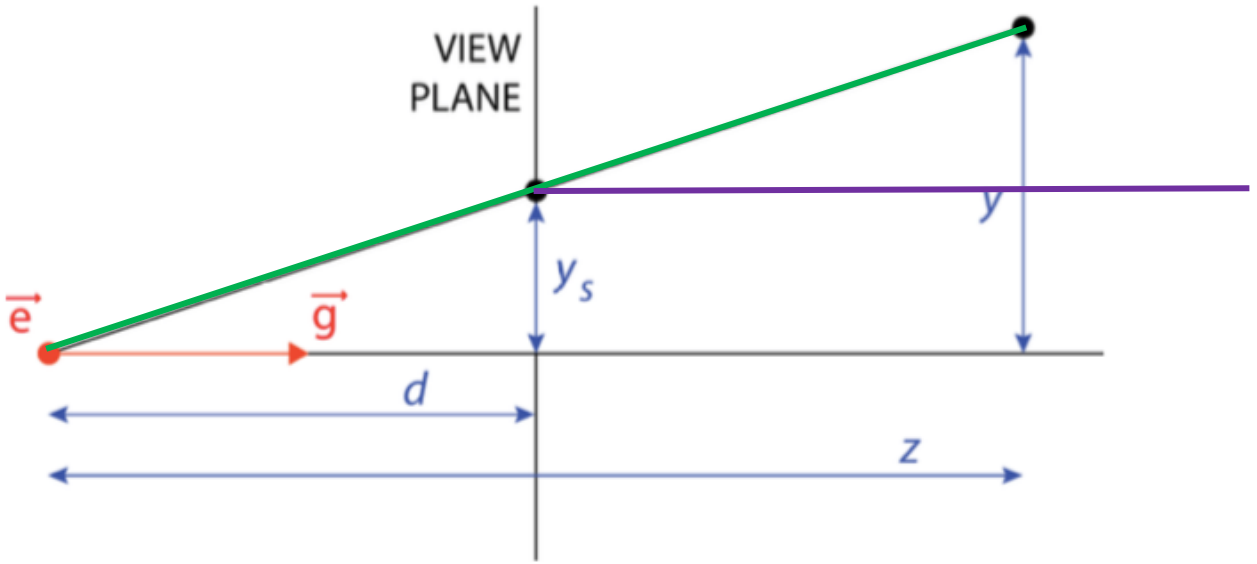
How do we calculate this? (cf. book, fig. 7.8/7.9 (3rd/2nd ed.))



We want to transform the y value to be on the purple line

Transforming the view frustum

How do we calculate this? (cf. book, fig. 7.8/7.9 (3rd/2nd ed.))



From basic geometry we know:

$$\frac{y_s}{y} = \frac{d}{z} \qquad \text{and thus} \qquad y_s = \frac{d}{z}y$$

We want to transform the y value to be on the purple line

Transforming the view frustum

In the following, we assume that

- we are looking in negative $z-$direction and
- we project onto the near plane.

Hence, the distance $d = -n$, and we need a matrix that gives us

- $x_s = \frac{dx}{-z} = \frac{nx}{z}$
- $y_s = \frac{dy}{-z} = \frac{ny}{z}$

and a z-value that

- stays the same for all points on the near and fare planes
- does not change the order along the $Z-$axis for all other points

Problem: we can't do division with matrix multiplication

”Dividing” in Homogeneous coordinates

- Recall that the homogeneous coordinate of a point is 1 and of a vector is 0

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

In homogeneous coordinates

point

$$\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

or

vector

$$\begin{bmatrix} x \\ y \\ z \\ 0 \end{bmatrix}$$

”Dividing” in Homogeneous coordinates

- What if we had the last coordinate as a different value w?

$$\begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

”Dividing” in Homogeneous coordinates

- What if we had the last coordinate as a different value w?
- For w ≠ 0, we call it a point

$$\begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

”Dividing” in Homogeneous coordinates

- What if we had the last coordinate as a different value w?
- For w ≠ 0, we call it a point
- To get it in our usual form for points, we divide by w

$$\begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} \qquad \begin{bmatrix} \frac{x}{w} \\ \frac{y}{w} \\ \frac{z}{w} \\ 1 \end{bmatrix}$$

”Dividing” in Homogeneous coordinates

- What if we had the last coordinate as a different value w?
- For w ≠ 0, we call it a point
- To get it in our usual form for points, we divide by w

$$\begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

This process is called “homogenization”

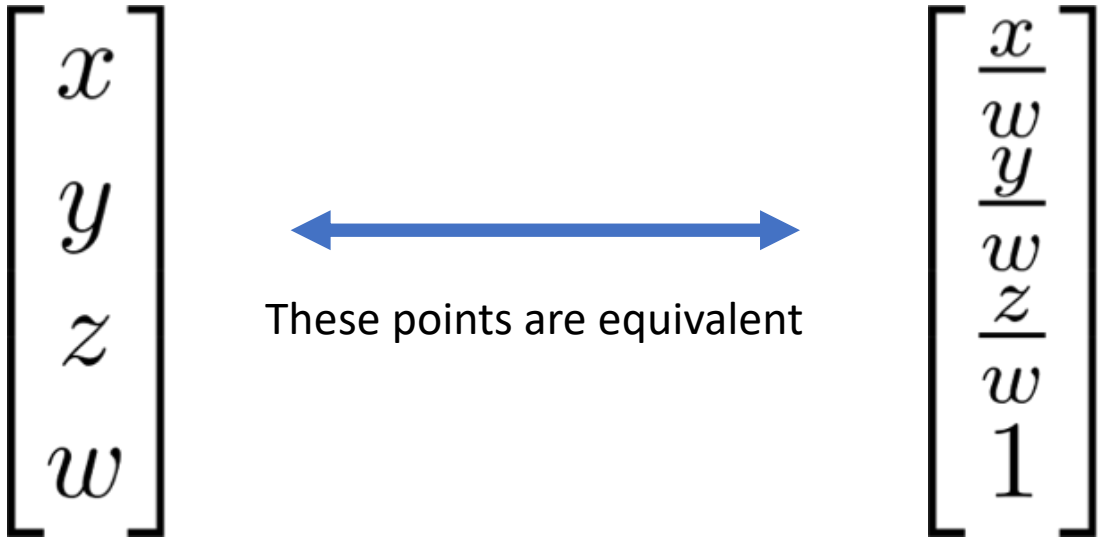
$$\begin{bmatrix} \frac{x}{w} \\ \frac{y}{w} \\ \frac{z}{w} \\ 1 \end{bmatrix}$$

”Dividing” in Homogeneous coordinates

- We say that two points in homogeneous coordinates are “the same” or “equivalent” if they have the same homogenized form

”Dividing” in Homogeneous coordinates

- This way, we can “divide” by w



Perspective transformation matrix

$$\begin{pmatrix} \frac{nx}{z} \\ \frac{ny}{z} \\ z^* \\ 1 \end{pmatrix}$$

(z^* denotes a z-value fulfilling the conditions that we specified)

- $x_s = \frac{dx}{-z} = \frac{nx}{z}$
- $y_s = \frac{dy}{-z} = \frac{ny}{z}$

and a z-value that

- stays the same for all points on the near and fare planes
- does not change the order along the $Z-$axis for all other points

Perspective transformation matrix

The following matrix will do the trick:

$$\begin{pmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & n+f & -fn \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

Remember that

- we are looking in negative Z-direction
- n, f denote the near and far plane of the view frustum
- n serves as projection plane

Let's verify that ...

Perspective transformation matrix

$$\begin{pmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & n+f & -fn \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x \\ y \\ z\frac{n+f}{n} - f \\ \frac{z}{n} \end{pmatrix} \xrightarrow{\text{homogenize}} \begin{pmatrix} \frac{nx}{z} \\ \frac{ny}{z} \\ n+f-\frac{fn}{z} \\ 1 \end{pmatrix}$$

Indeed, that gives the correct values for x_s and y_s.

But what about z? Remember our requirements for z:

- stays the same for all points on the near and fare planes
- does not change the order along the $Z-$axis for all other points

Homogeneous coordinates and perspective transformation

We have $z_s = n + f - \frac{fn}{z}$ and need to prove that ...

- points on the near plane are mapped to themselves,
  i.e. if $z = n$, then $z_s = n$:

$$z_s = n + f - \frac{fn}{n} = n + f - f = n$$

  and obviously $x_s = \frac{nx}{n} = x$ and $y_s = \frac{ny}{n} = y$.

- points on the far plane stay on the far plane,
  i.e. if $z = f$, then $z_s = f$:

$$z_s = n + f - \frac{fn}{f} = n + f - n = f$$

and ...

Homogeneous coordinates and perspective transformation

We have $z_s = n + f - \frac{fn}{z}$ and need to prove that ...

- z–values for points within the view frustum stay within the view frustum,
  i.e. if $z > n$ then $z_s > n$:

$$z_s = n + f - \frac{fn}{z} > n + f - \frac{fn}{n} = n$$

  and if $z < f$ then $z_s < f$:

$$z_s = n + f - \frac{fn}{z} < n + f - \frac{fn}{f} = f$$

and ...

Homogeneous coordinates and perspective transformation

We have $z_s = n + f - \frac{fn}{z}$ and need to prove that ...

- the order along the Z-axis is preserved, i.e. if $0 > n \geq z_1 > z_2 \geq f$ then $z_{1s} > z_{2s}$:

  With $z_{1s} = n + f - \frac{fn}{z_1}$ and $z_{2s} = n + f - \frac{fn}{z_2}$ we get:

  $$z_{1s} - z_{2s} = \frac{fn}{z_2} - \frac{fn}{z_1} = \frac{(z_1 - z_2)fn}{z_1 z_2}.$$

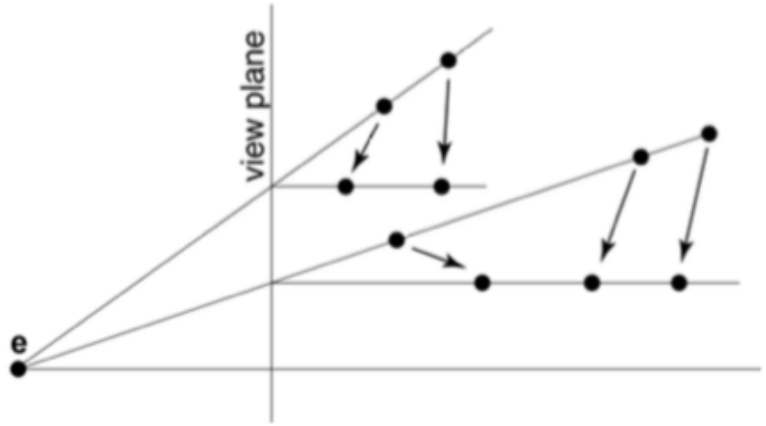
  Because of $f, z_1, z_2, n < 0$ we have $\frac{fn}{z_1 z_2} > 0$, and because of $z_1 > z_2$, we have $z_1 - z_2 > 0$, so

  $$z_{1s} - z_{2s} > 0 \text{ or}$$

  $$z_{1s} > z_{2s}$$

Homogeneous coordinates and perspective transformation

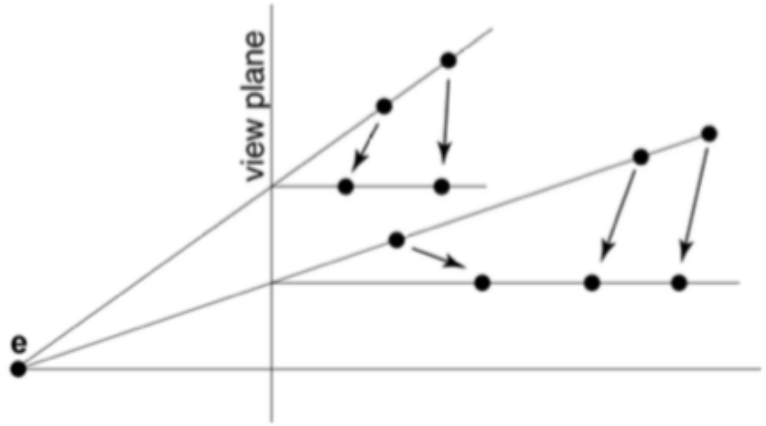
Hence, the order is preserved. But how?



$z_s = n + f - \frac{fn}{z}$,
so z_s is proportional to $-\frac{1}{z}$

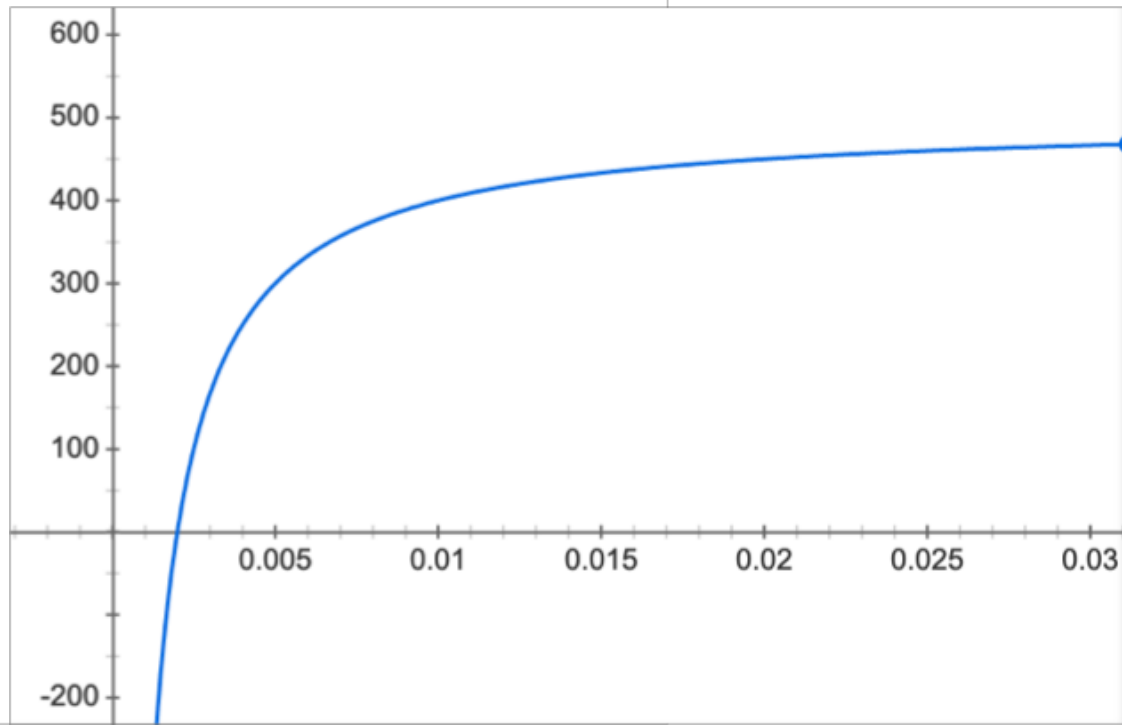
Homogeneous coordinates and perspective transformation

Hence, the order is preserved. But how?



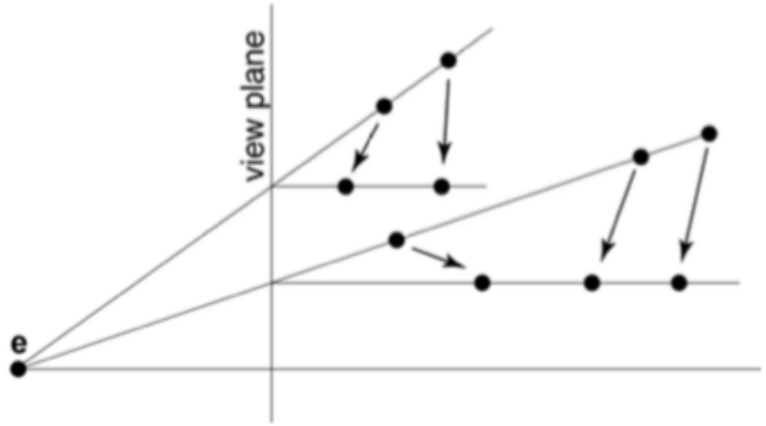
$z_s = n + f - \frac{fn}{z}$,
so z_s is proportional to $-\frac{1}{z}$

Non-linear! Actual distance is not preserved



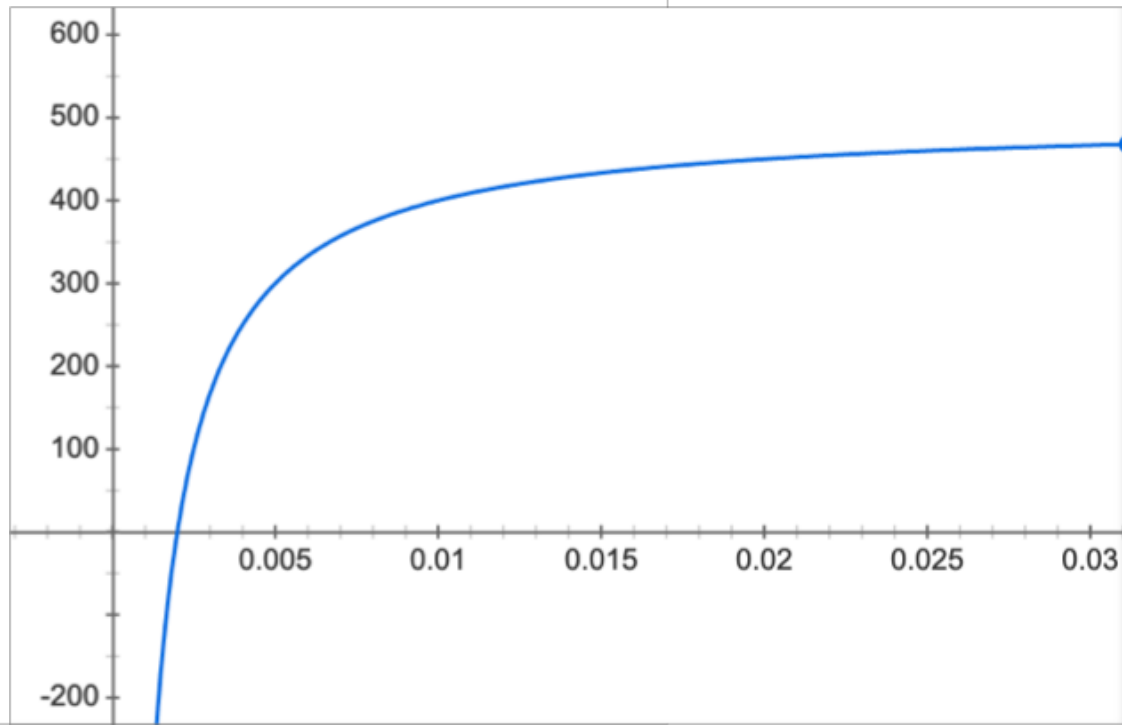
Homogeneous coordinates and perspective transformation

Hence, the order is preserved. But how?



$z_s = n + f - \frac{fn}{z}$,
so z_s is proportional to $-\frac{1}{z}$

Non-linear! Actual distance is not preserved



This new z value is called pseudo-depth

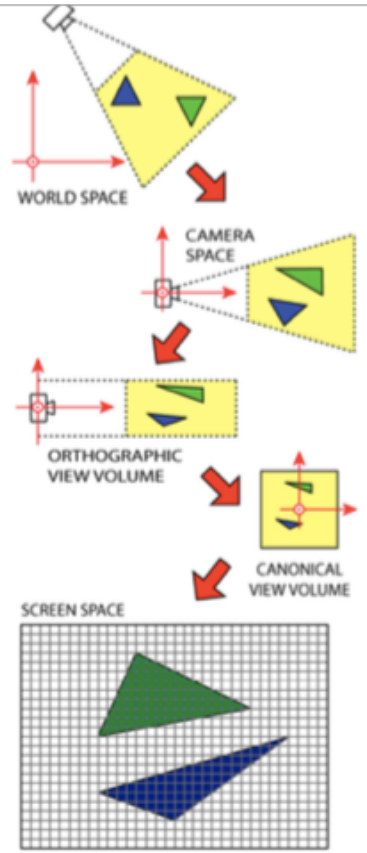
Perspective transformation matrix

With this, we got our final matrix P. To map the perspective view frustum to the orthographic view volume, we need to combine it with the orthographic projection matrix M_{orth}, i.e. $M_{per} =$

$$M_{orth}P = M_{orth}\begin{pmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & n+f & -fn \\ 0 & 0 & 1 & 0 \end{pmatrix} = \begin{pmatrix} \frac{2n}{r-l} & 0 & \frac{l+r}{l-r} & 0 \\ 0 & \frac{2n}{t-b} & \frac{b+t}{b-t} & 0 \\ 0 & 0 & \frac{f+n}{n-f} & \frac{2fn}{f-n} \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

Overview

The following achieved orthographic projection:



$$\begin{pmatrix} x_{pixel} \\ y_{pixel} \\ z_{canonical} \\ 1 \end{pmatrix} = M_{vp} M_{orth} M_{cam} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

And if we replace M_{orth} with M_{per} we get perspective projection:

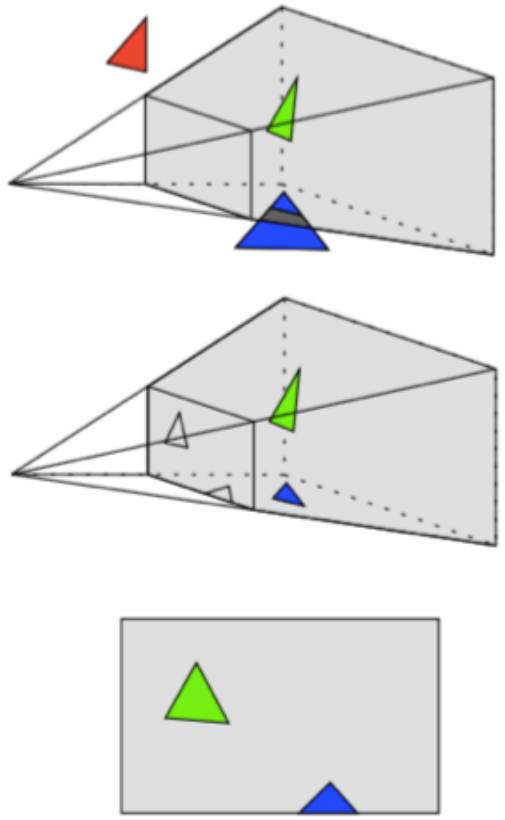
$$\begin{pmatrix} x_{pixel} \\ y_{pixel} \\ z_{canonical} \\ 1 \end{pmatrix} = M_{vp} M_{per} M_{cam} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

What's left?

Now we can draw points and lines.
But there's more ...

- Triangles that lie (partly) outside the view frustum need not be projected, and are clipped.
- The remaining triangles are projected if they are front facing.
- Projected triangles have to be shaded and/or textured.

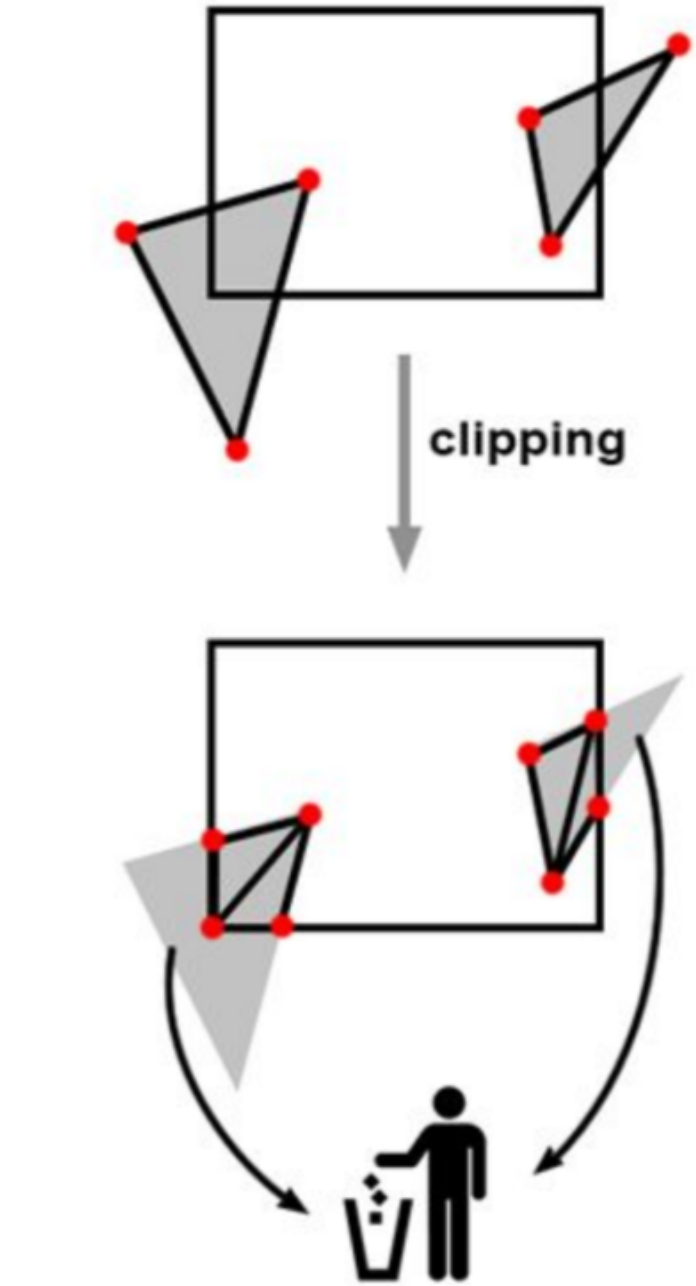
We will talk about this in the upcoming lectures.



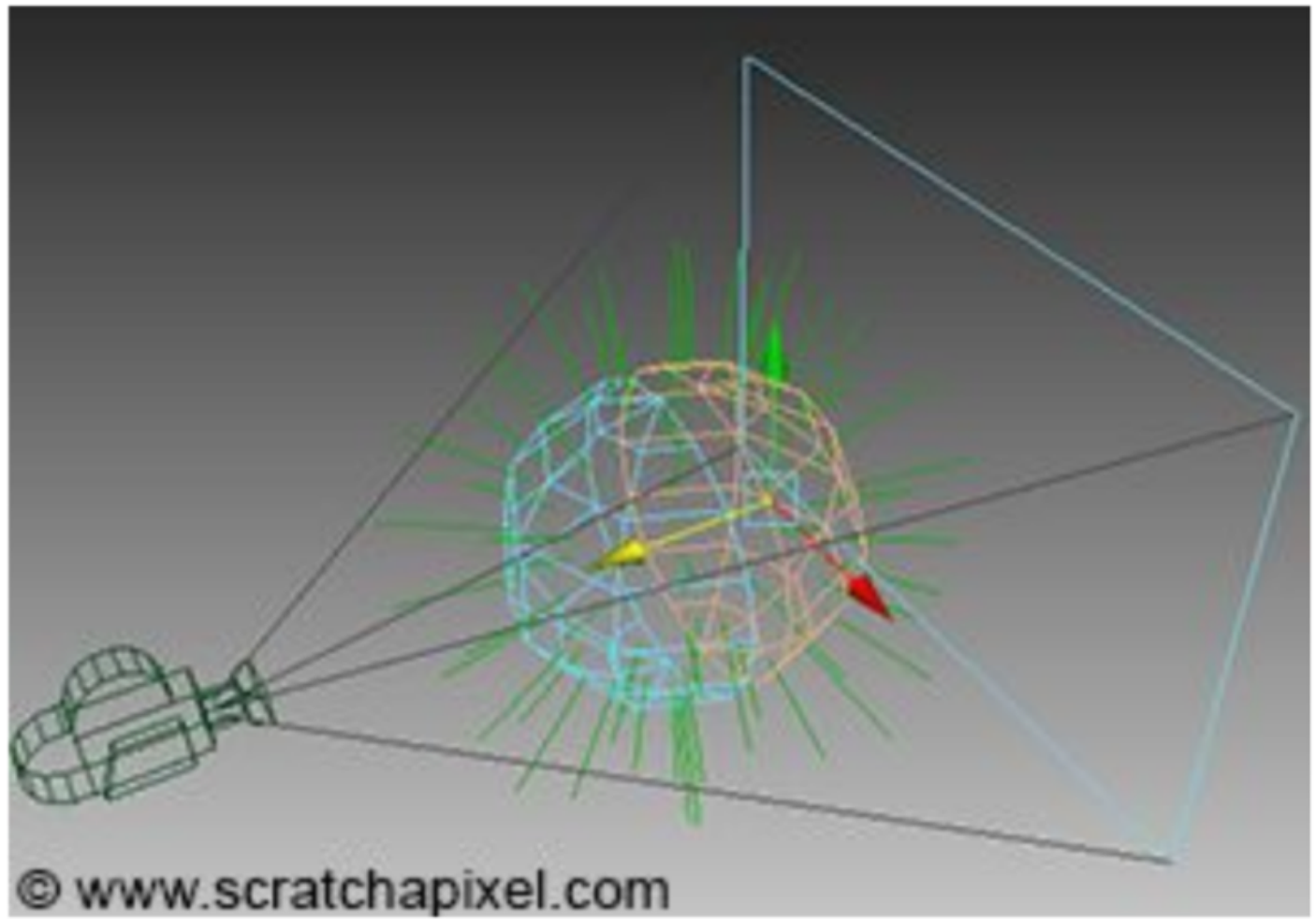
What’s left?

- Clipping
- Culling
- Visibility
- The shader pipeline

Clipping



Culling



Culling

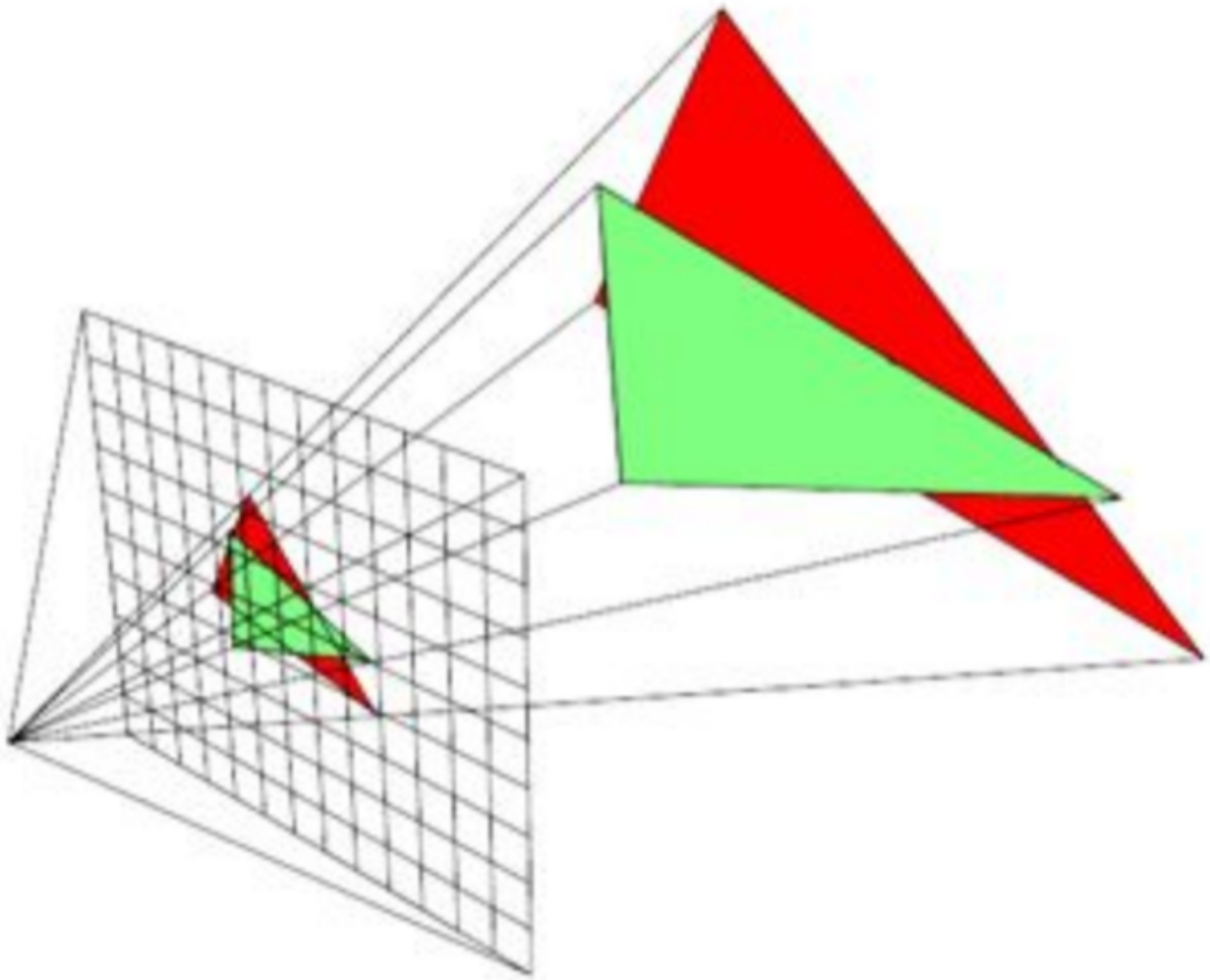
Only geometry whose normals are facing the camera will be visible in the final image

Discard all triangles where the dot product of their surface normal and the camera-to-triangle vector is greater than or equal to zero

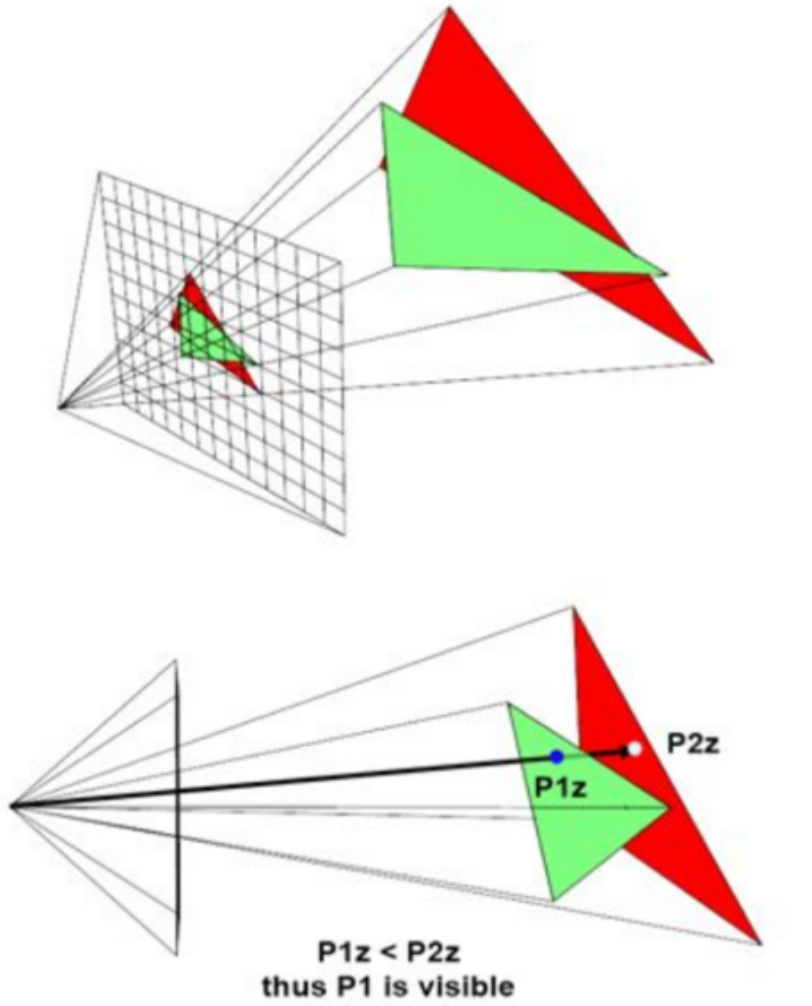
$$(V - e) \cdot n < 0$$

V is a vertex on the triangle, e is the camera origin, and n is the triangle face normal

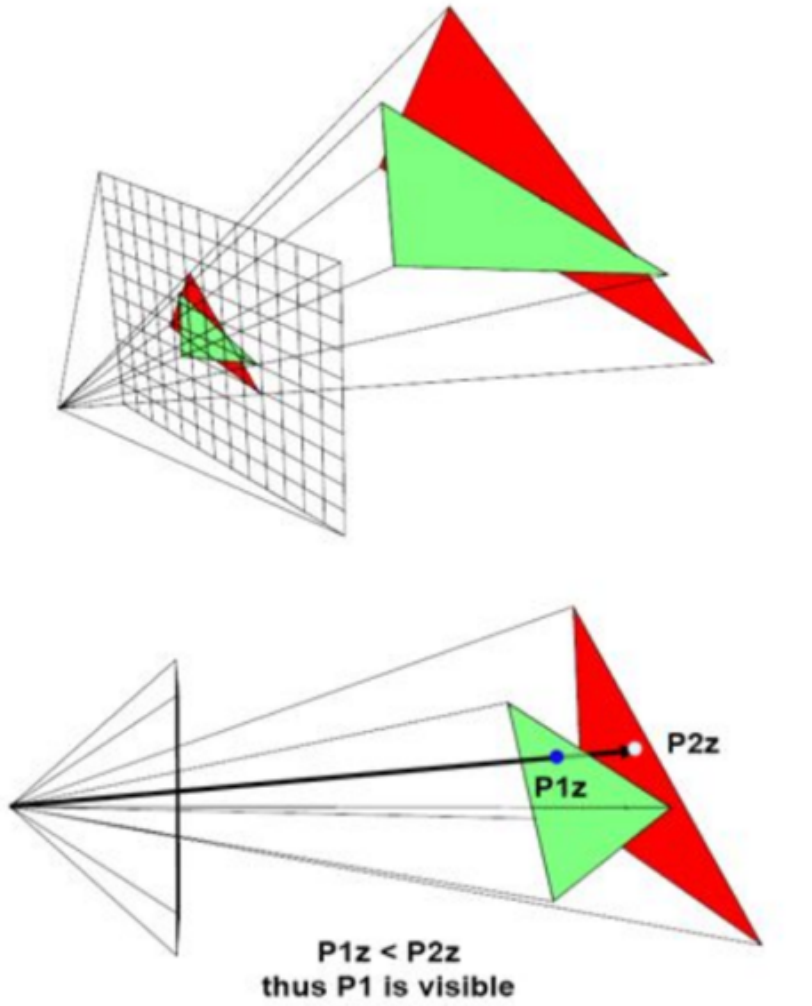
But how do we determine which objects are visible?



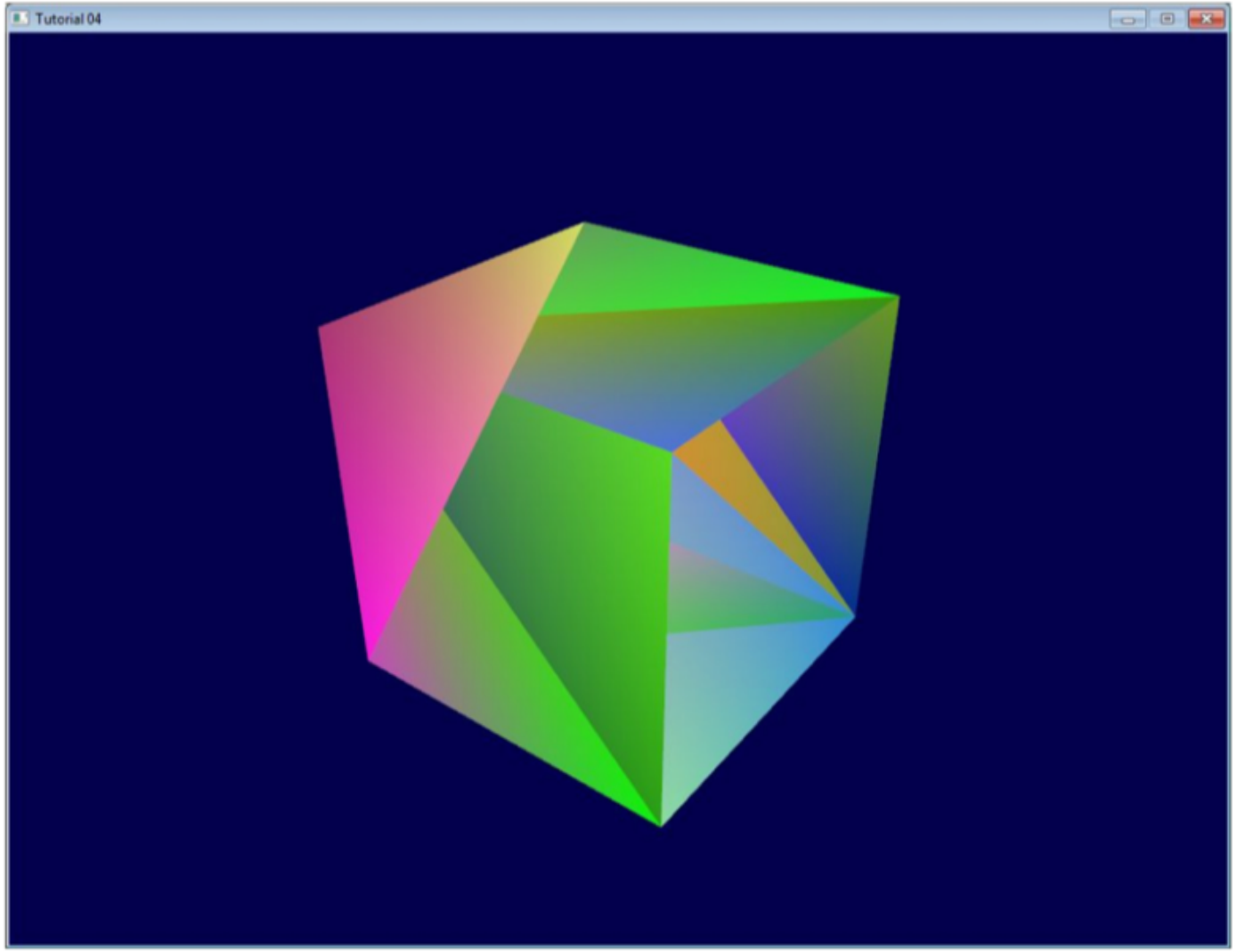
But how do we determine which objects are visible?



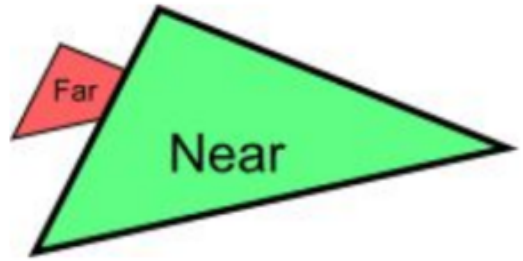
But how do we determine which objects are visible?

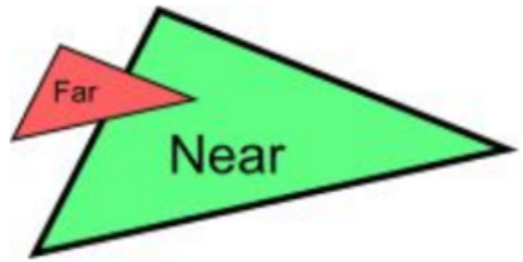


What can happen?

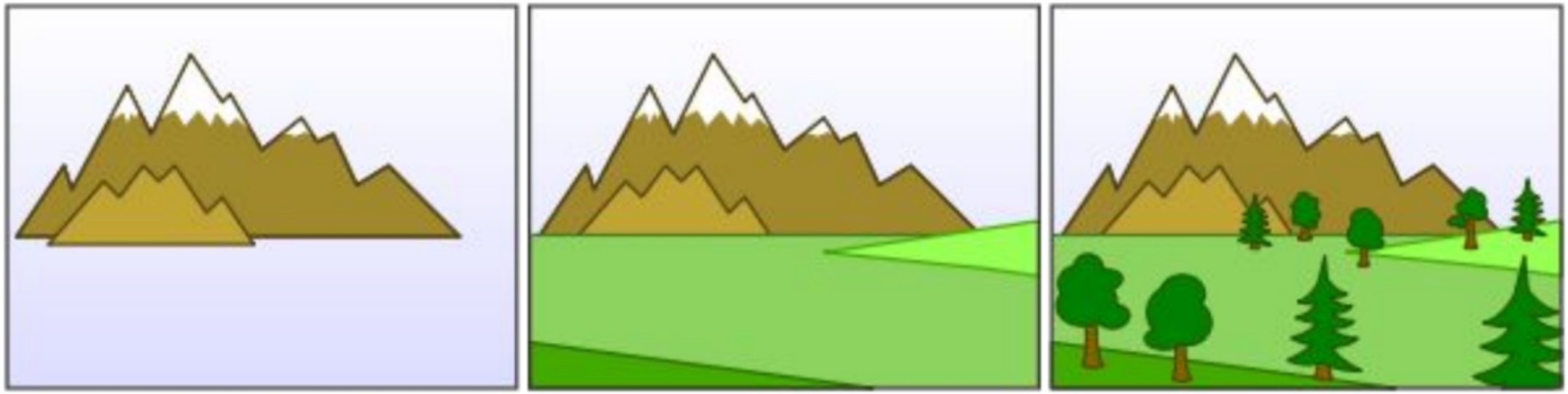


What can happen?





Painter's algorithm



Painter's algorithm

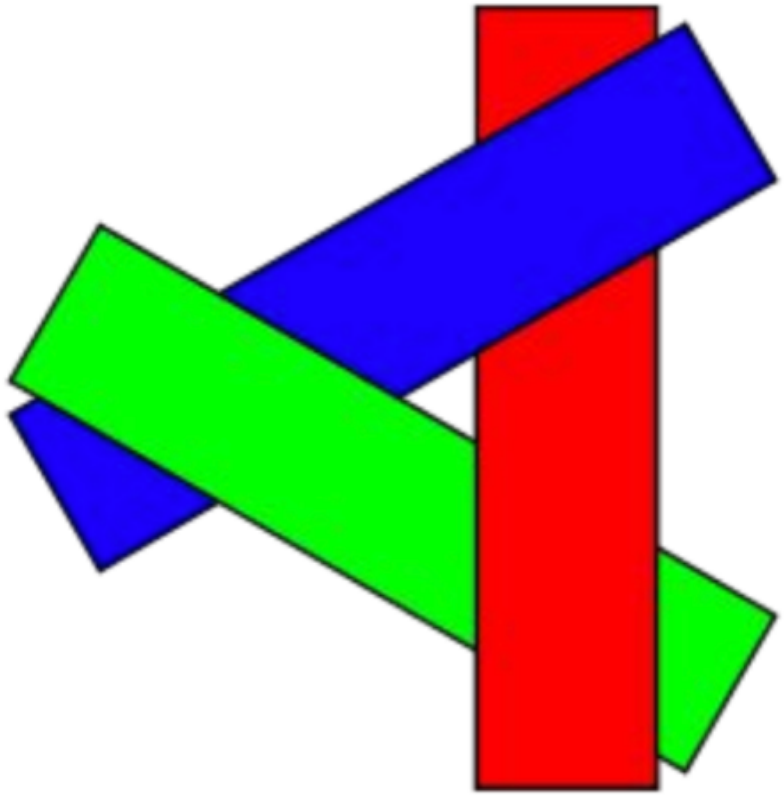
also known as priority fill

sorts all the polygons in a scene by their *furthest* depth and then paints them in this order, farthest to closest

paint over the parts that are normally not visible

What could possibly go wrong?

Problems with Painter's



Why does this happen?

topological sorting

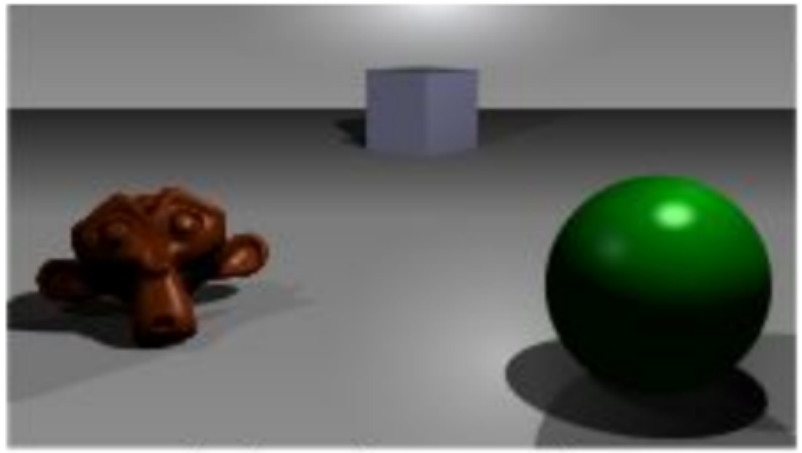
recall: like DFS

Problems with Painter's

Renders everything, even if not needed

Computationally expensive

Z-buffer



A simple three-dimensional scene

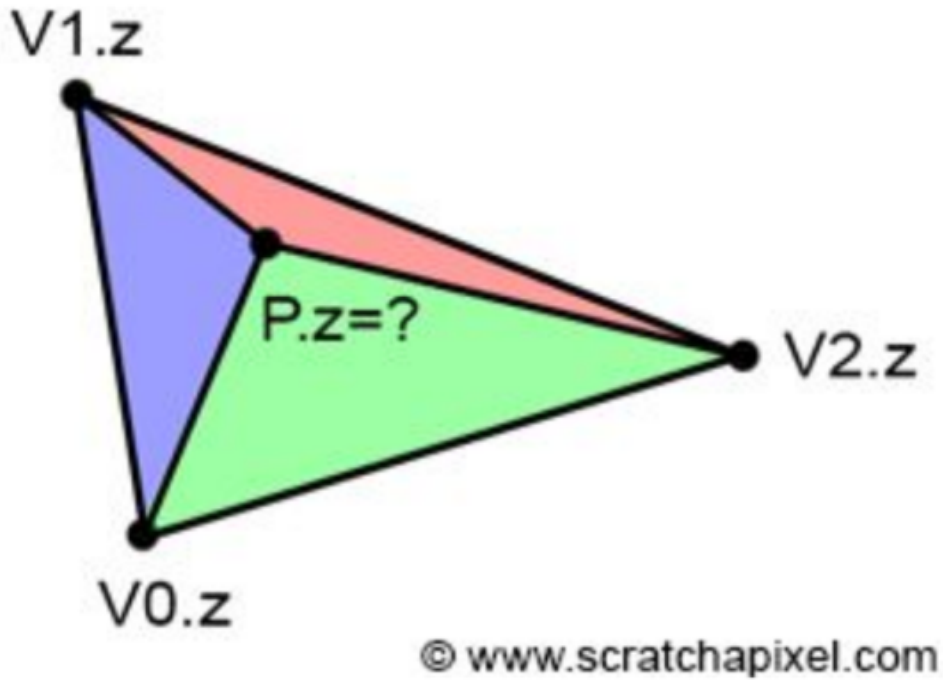
Z-buffer representation

Z-buffer

```
initialize z-buffer to size of # of pixels with max values
for each object/triangle in scene do
  project vertices of triangles to screen
  for each pixel do
    if (pixel is contained in projected triangle) then
      Compute z
      if (z is closer than object stored in z-buffer)
        update z-buffer with new z value
```

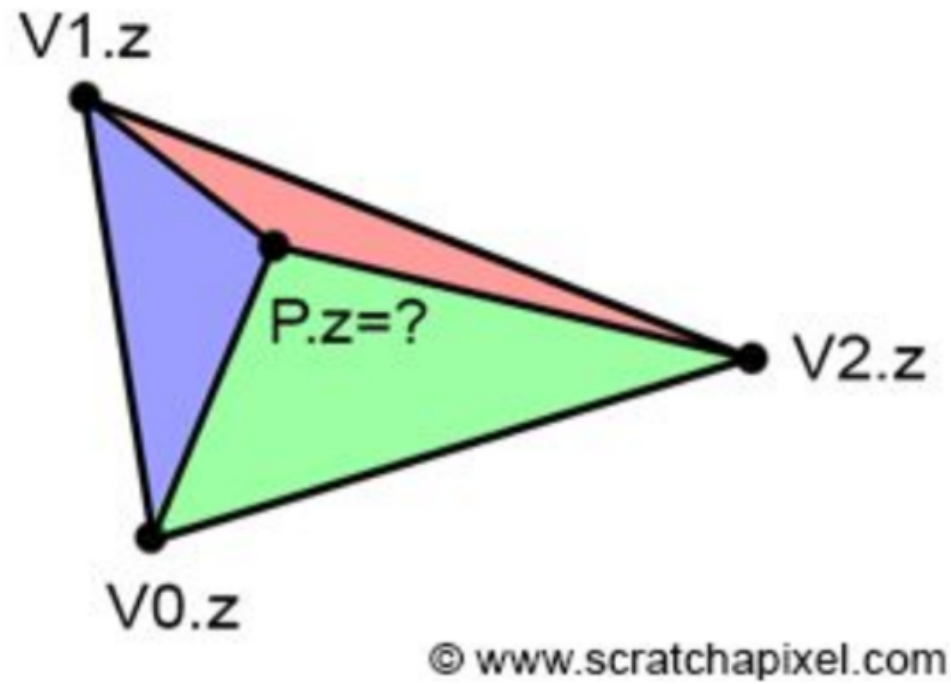
Z-buffer

How to compute z values?



Z-buffer

Can we use barycentric coordinates?

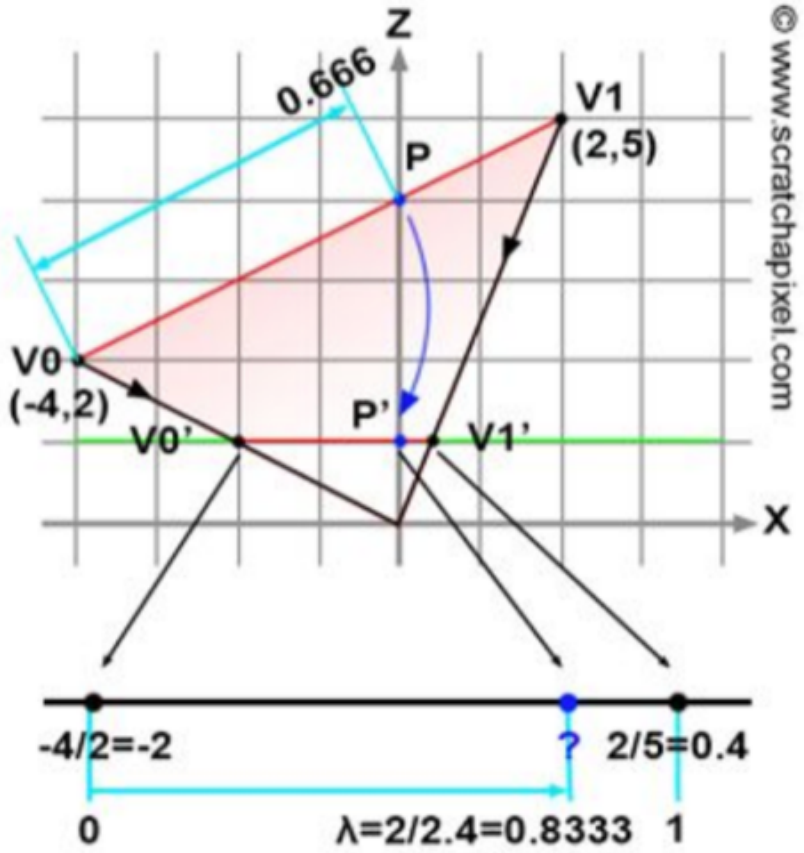


Z-buffer

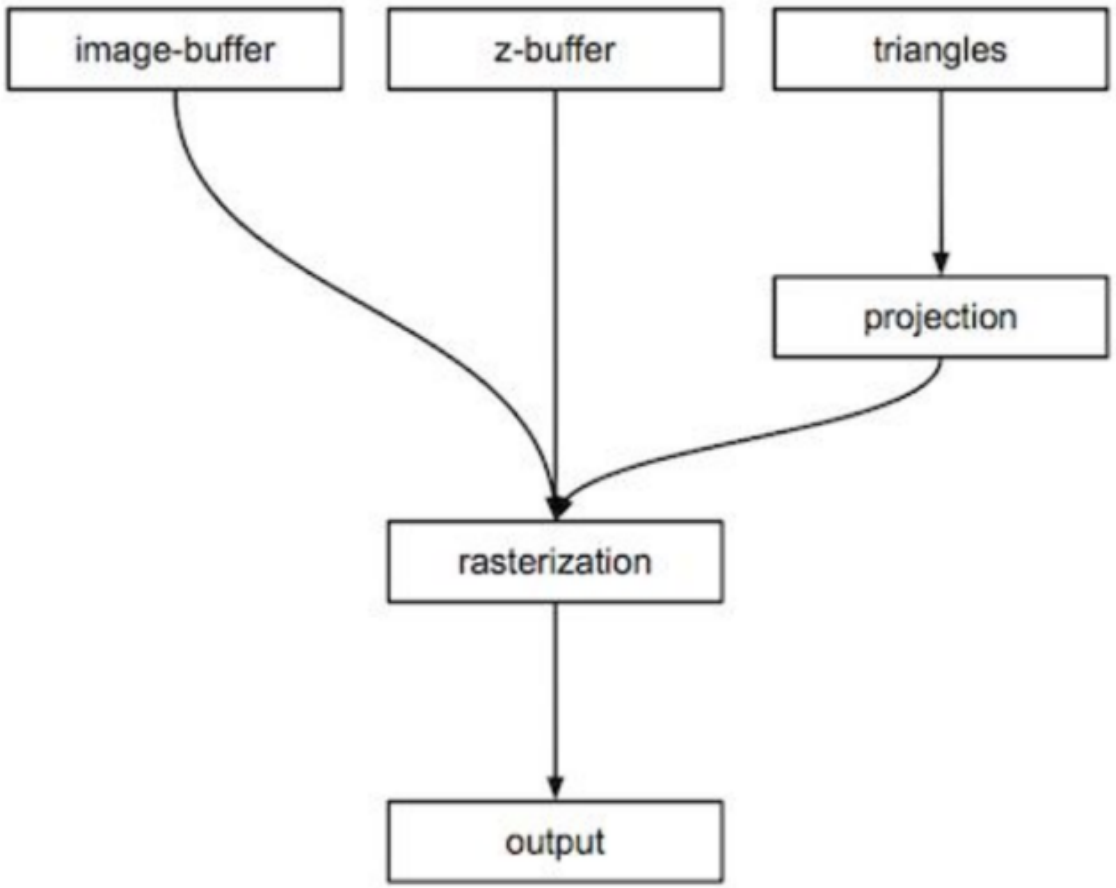
No! 🤯

after we projected the vertices and performed the perspective projection divide, the z values don't vary linearly anymore!

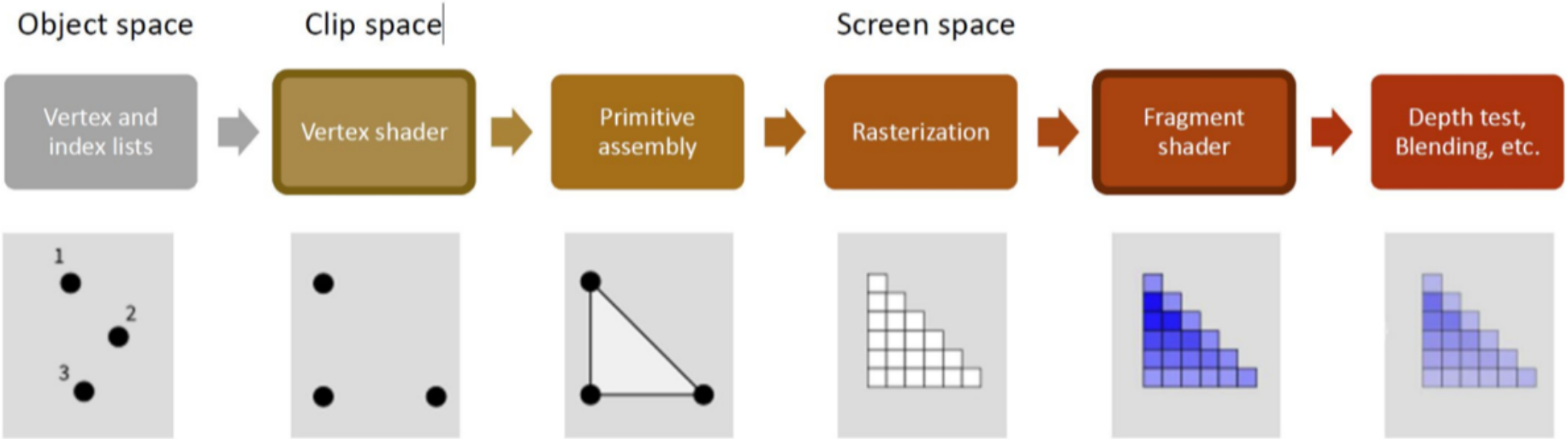
perspective projection does not preserve distances



Pipeline (space)



Pipeline (time)



Pipeline

We do things to each vertex, and then to each pixel

This is what we call "embarrassingly parallelizable"

Let's do it to each unit (vertex and fragment) at the same time

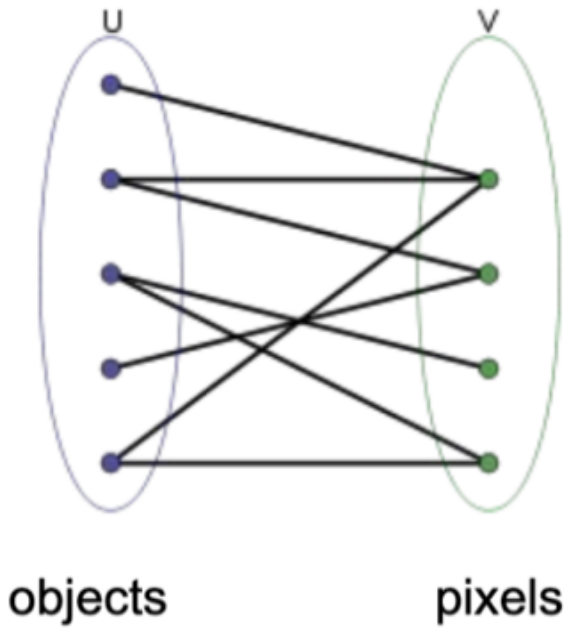
Data dependency

Why is it so parallelizable?

In raytracing, we use a ray for each pixel.

Parallelizable? yes, but for one pixel, we need multiple objects' information.

Rays bounce and data is accumulated.

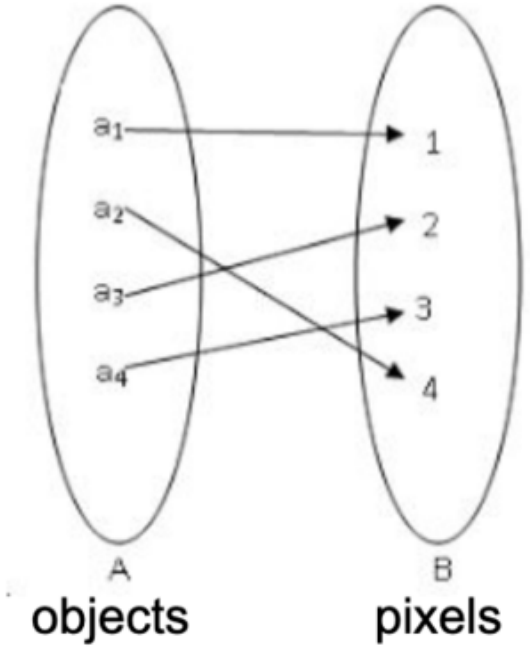


Data dependency

Why is it so parallelizable?

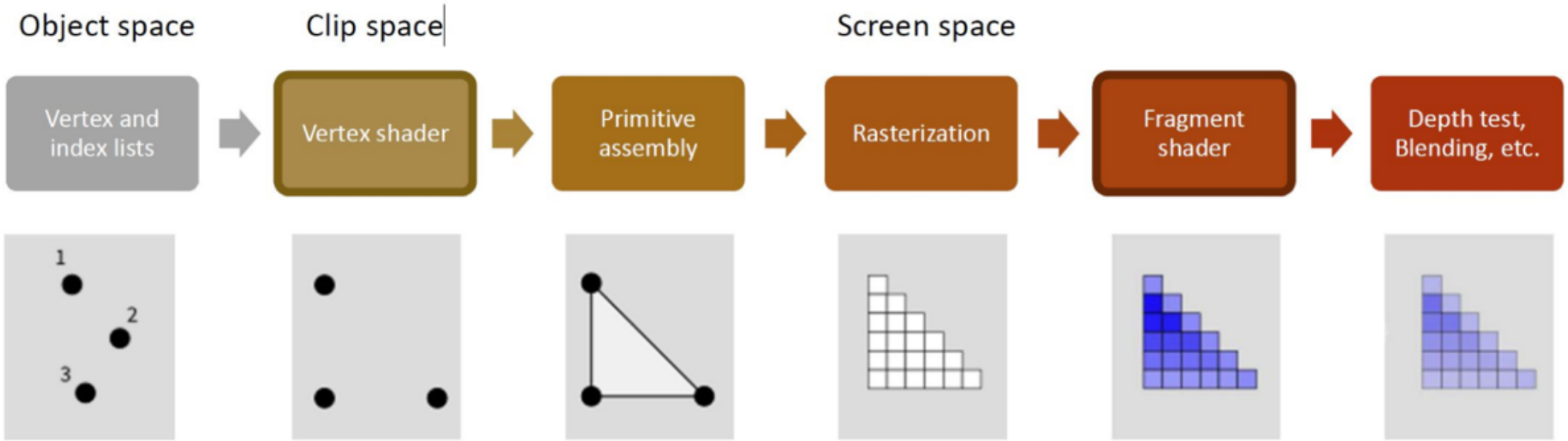
In rasterization, for each pixel, we only need data for one object at a time.

We can do everything to each vertex *and* each fragment at the same time.



It's so parallelizable, our hardware does it for us

a thread for every pixel!



What do you really need to know for A6?

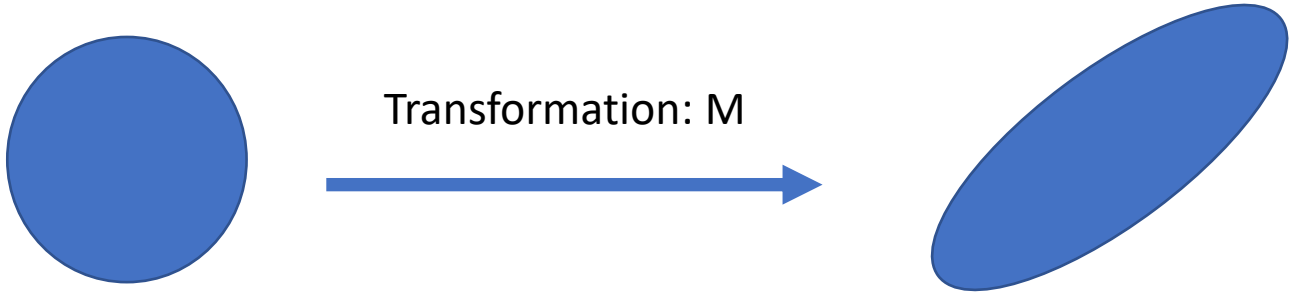
- Vertex shaders
- Tessellation shaders
- Fragment shaders

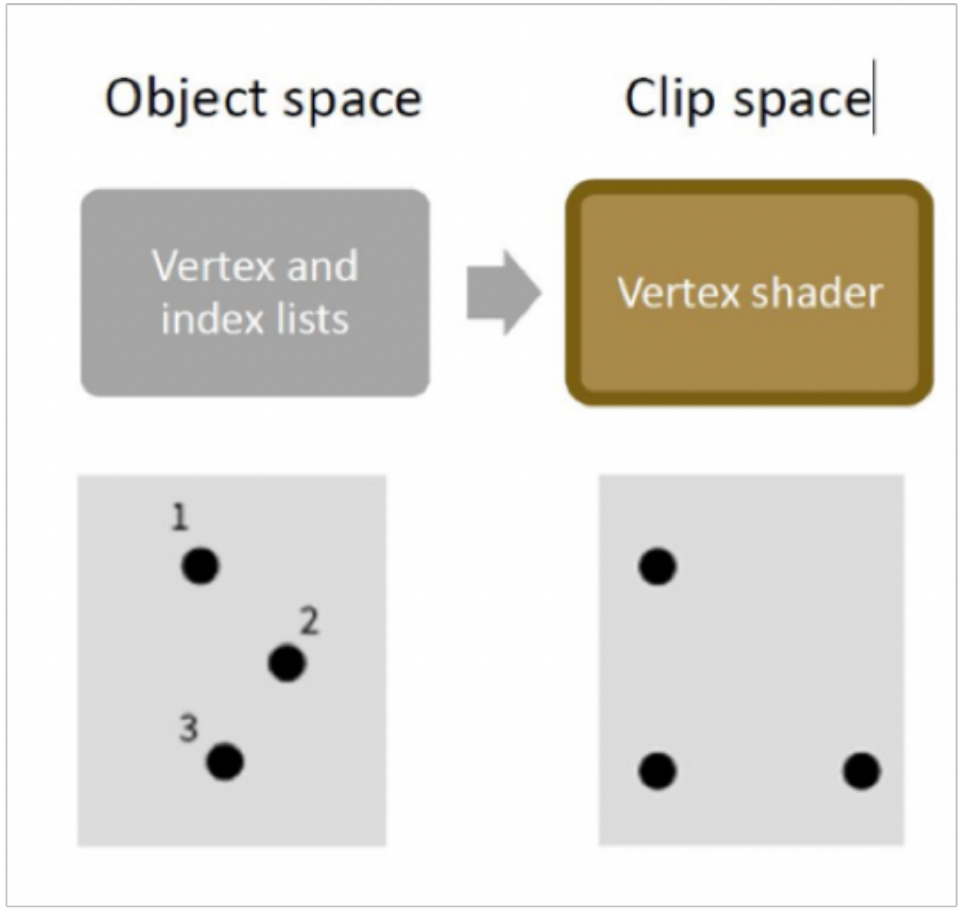
What don’t you need to know?

- All the parts of the old graphics pipeline, that’s all put into the matrix view, which transforms from world coordinates, to screen space. The hardware does this for you

First: Vertex shader

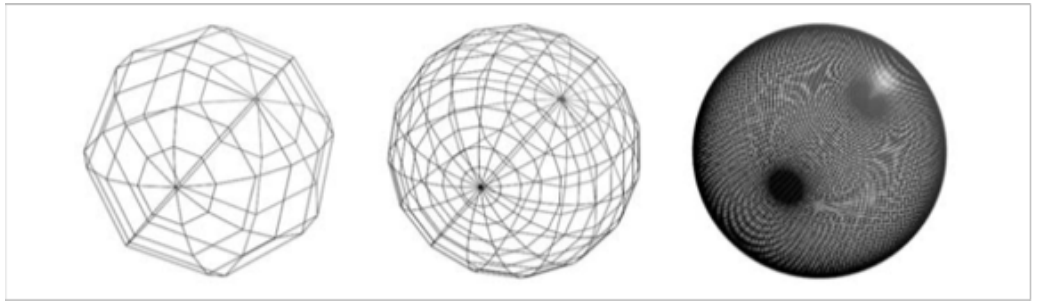
- Transforms an object from “object space” to our clipped world space
- For your assignment, you need to know how to take a model object (ie the unit sphere at the origin) to a transformed object (ie an arbitrary ellipse)

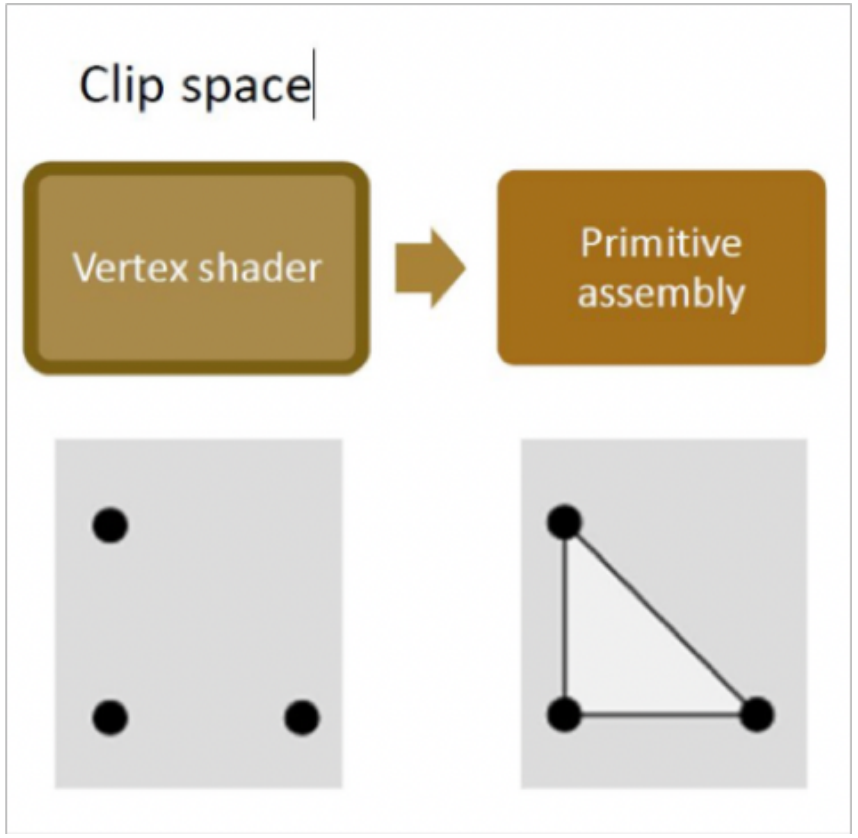




Next: Tessellation shader

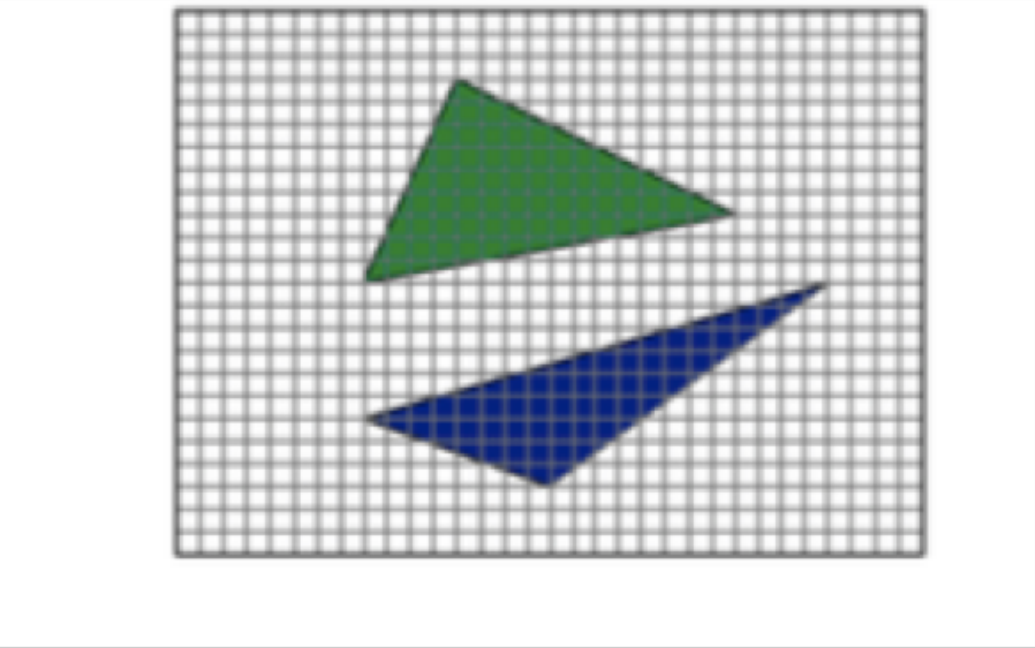
- The Tessellation shader makes our triangles for us
- We can specify a finer resolution mesh here, by telling the Tessellation shader to split up our triangle
- The assignment outlines this

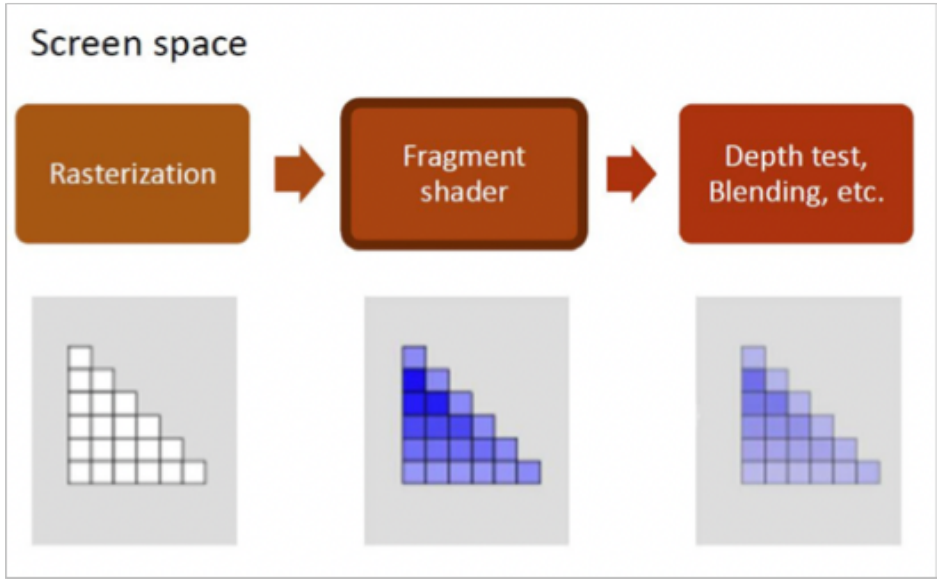




Last: Fragment shader

- After rasterization the fragment shader gives a color to the fragments.
- Fragments are the sample of the pixels covered by a primitive (an example of a primitive is a triangle)





Phong vs Gouraud Shading

```
for each object/triangle in scene do
  project vertices of triangles to screen
  for each pixel do
    if (pixel is contained in projected triangle) then
      Evaluate shading model and set pixel to that color
    else
      set pixel color to background color
```

How to shade a fragment?

Phong vs Gouraud Shading

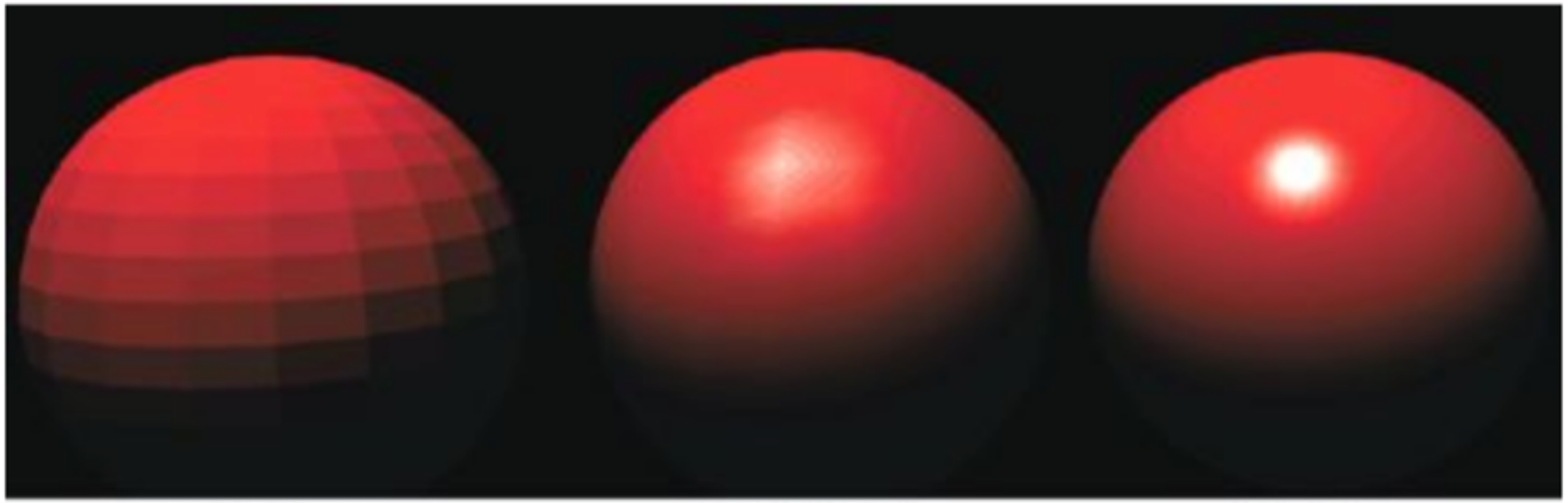
```
for each object/triangle in scene do
  project vertices of triangles to screen
  for each pixel do
    if (pixel is contained in projected triangle) then
      Evaluate shading model and set pixel to that color
    else
      set pixel color to background color
```

Blinn-Phong lighting model ≠ Phong shading

Phong vs Gouraud shading

Ordering is different

Interpolation at different stages

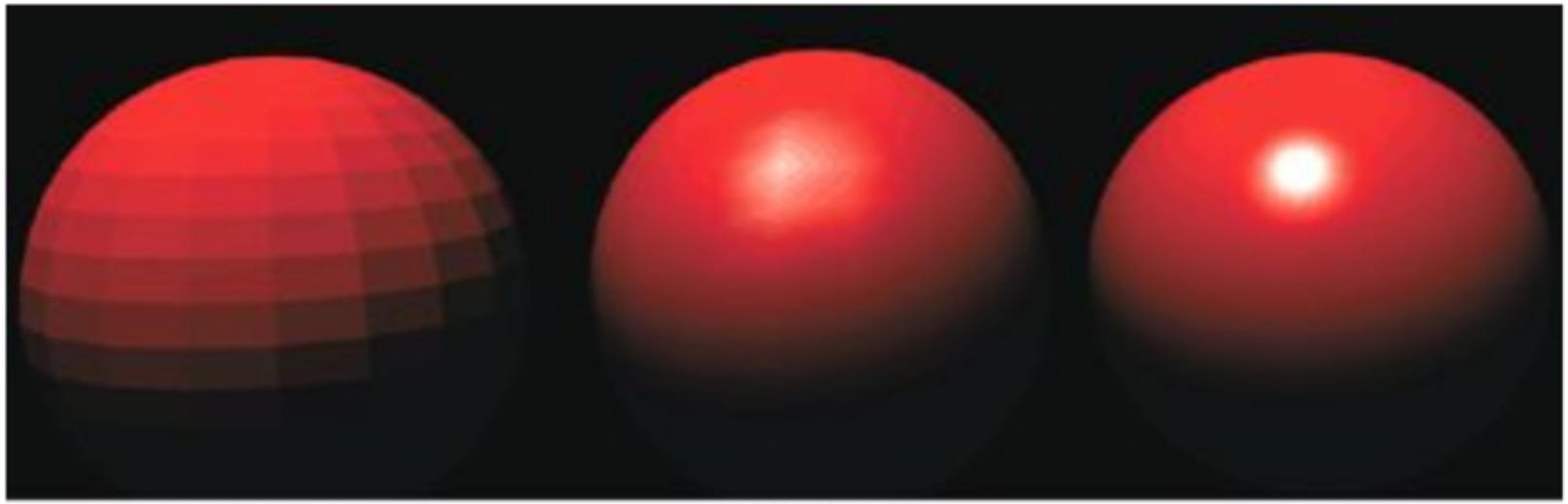


Flat shading

Find triangle normals

Apply Phong model to each triangle

Each triangle (or other polygon) has a constant color across it

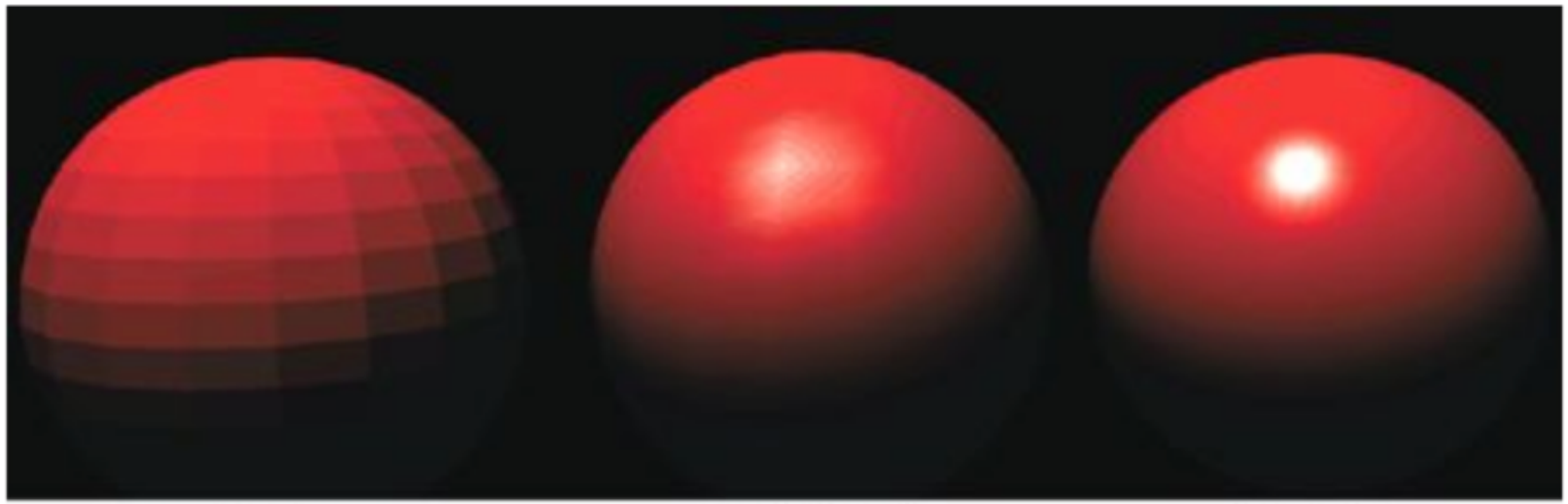


Gouraud shading

Find averaged vertex normals (like in the mesh assignment)

Apply Phong model to each vertex

Interpolate vertex colors across each triangle



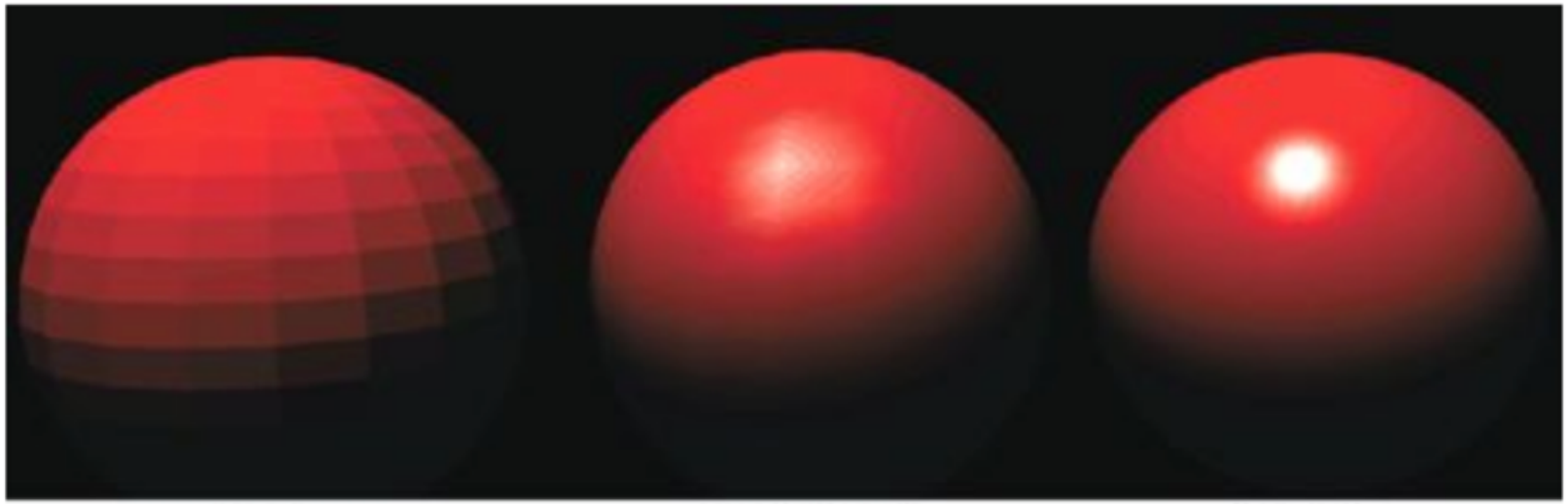
Phong shading

Find averaged vertex normals

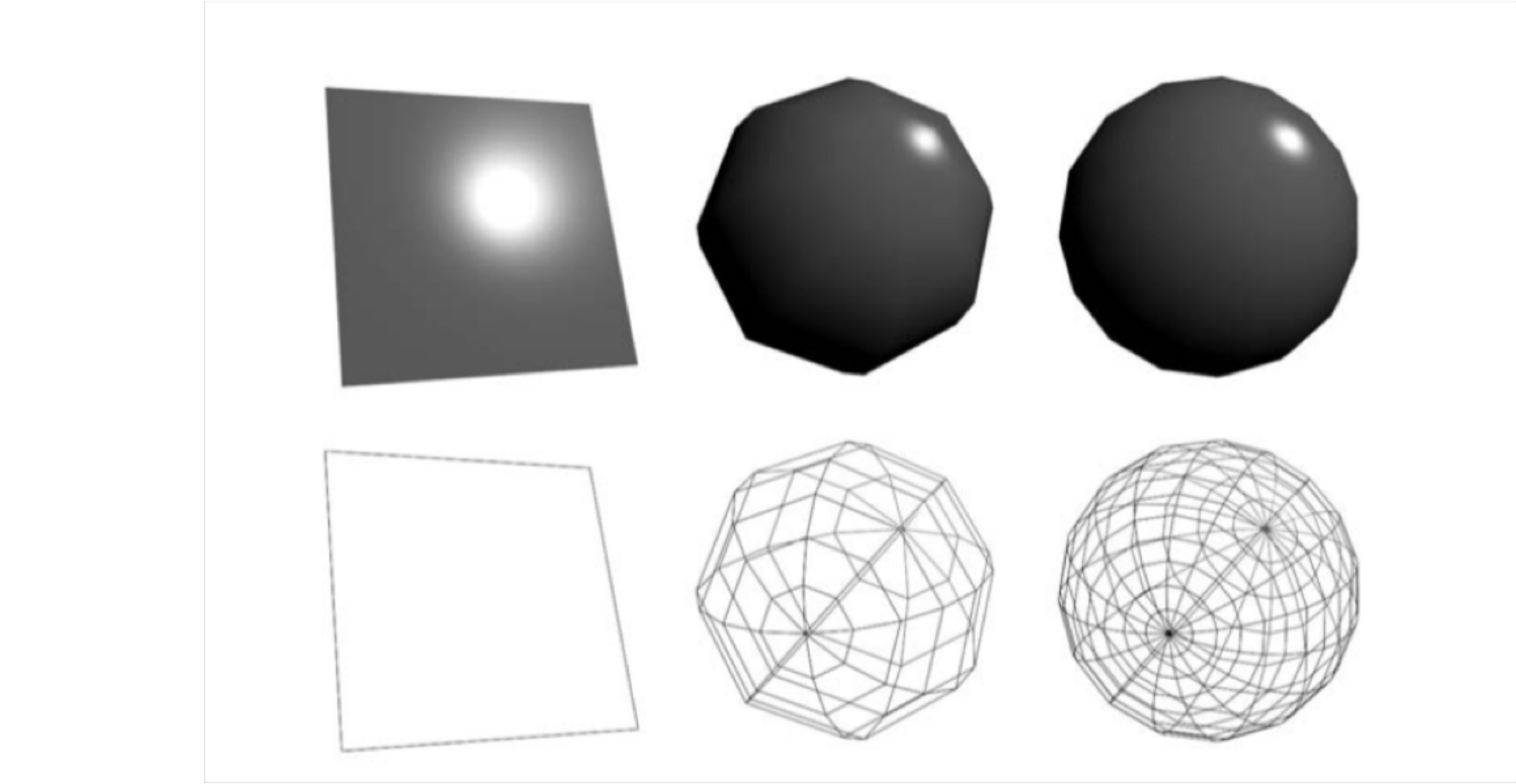
Interpolate normals across triangle edges

Interpolate edge normals across triangle (or other polygon)

Apply Phong model to each fragment (which corresponds to each pixel)



Phong shading = interpolate normals



What do you really need to know for A6?

- Special note: Make sure you know what the row-major/column-major order for initializing a matrix in glsl!

```
return mat4(
1,0,0,0,
0,1,0,0,
0,0,1,0,
0,0,0,1);
```