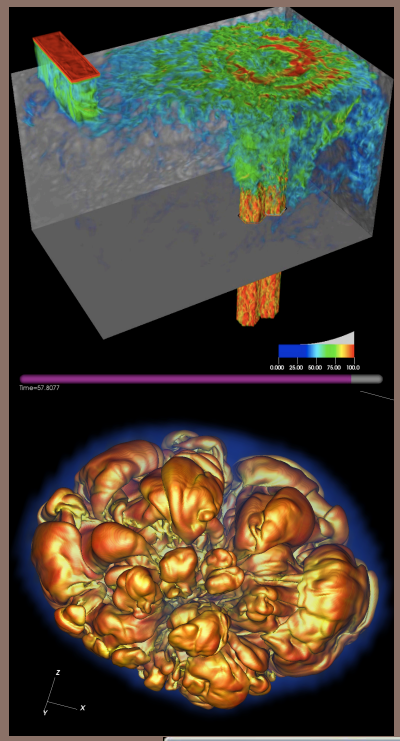
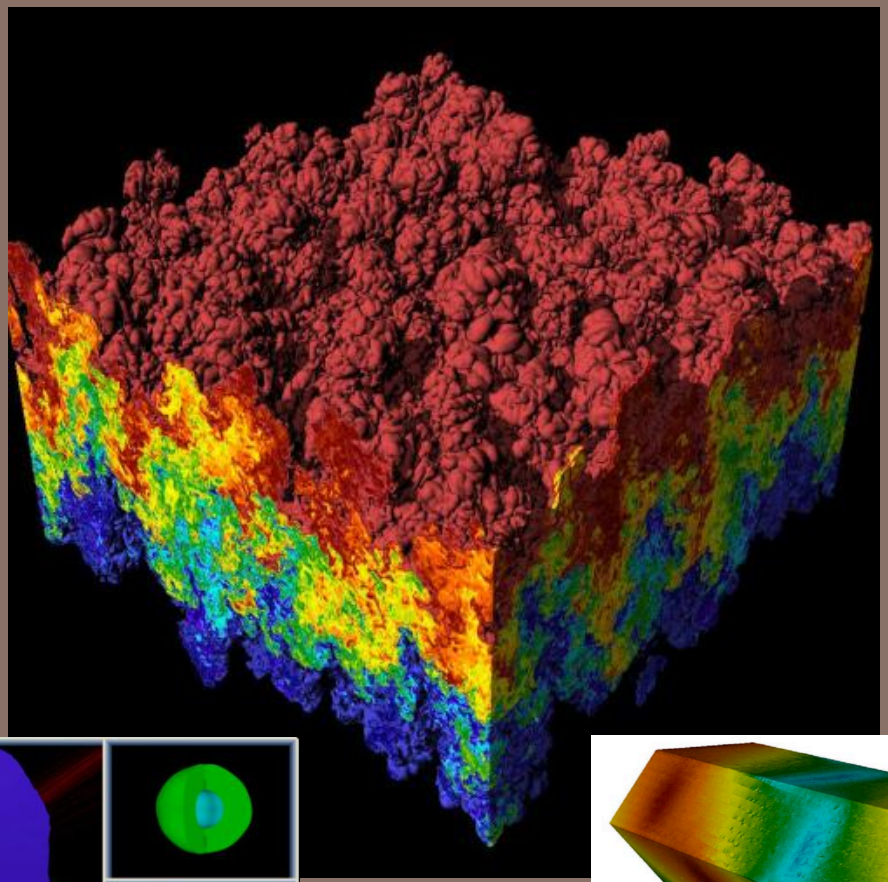
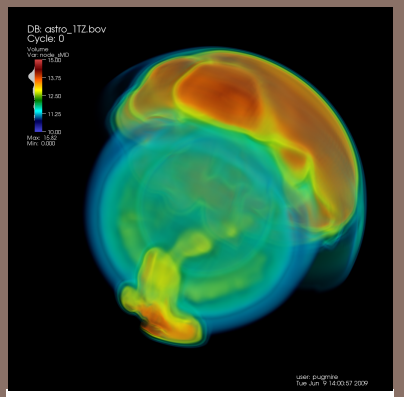
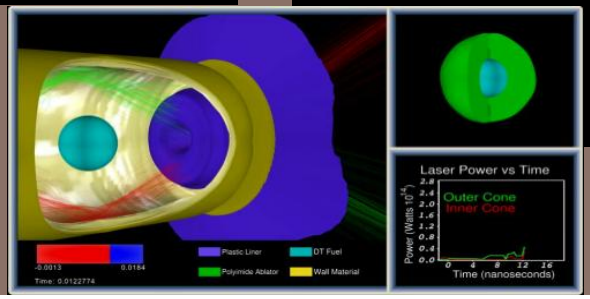
CIS 441/541: Introduction to Computer Graphics

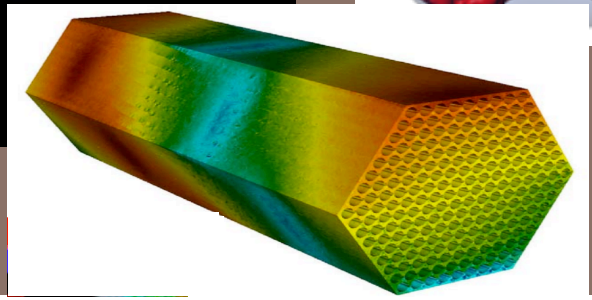
Lecture 12: Camera Transform / View Transform











Oct. 21st, 2016

Hank Childs, University of Oregon

Announcements

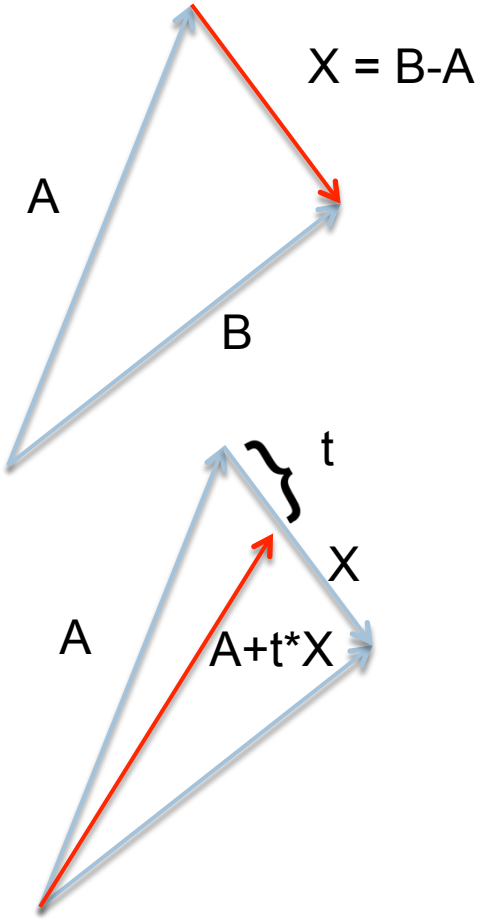
- OH: Dan Li → now Thurs 930am-11am

Class Cancellation

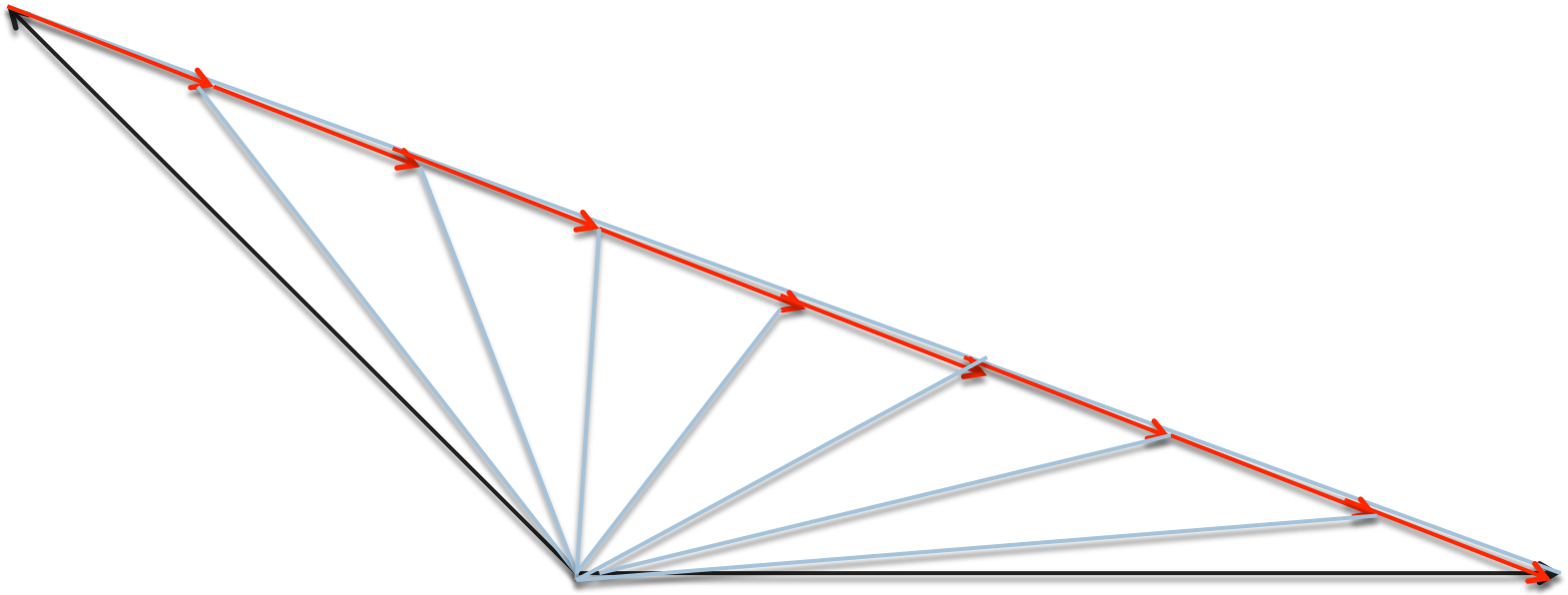
- Class cancelled on Monday, October 24th
- Will do YouTube lecture (if necessary) to stay on track

LERPing vectors

- LERP = Linear Interpolate
- Goal: interpolate vector between A and B.
- Consider vector X, where X = B-A
- Back to normal LERP:
  - A + t*(B-A) = A+t*X
- You will need to LERP vectors for 1E



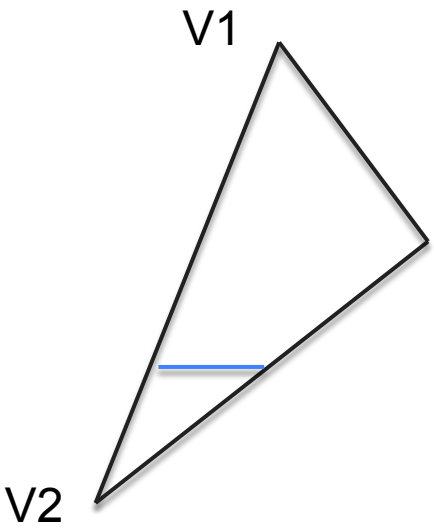
Masado's Comment (Approximately)

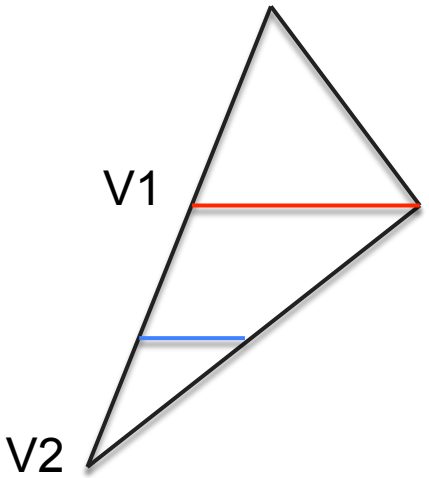


Even steps in t do not lead to even steps in angle
Also, resulting vector is likely not a normal

What you need to understand:
only that we will LERP vectors on a component by component basis
AND you should normalize them after you LERP

498 pixels…





What to do?...

- If you get a different baseline,
  - Please do not assume that you have a special case
  - Instead, use the print statements in the prompt and see where you diverge. Ask yourself "why?"
    - For our 498ers, there is a good reason
- 1E extended until Oct 27th
  - 1F *not* extended
- Note: Hank is out of email contact until Wednesday
  - Chairing symposium, half day meeting, two panels, leading paper writing group…

Project 1E

- Need to be working with unit vectors (length == 1)
- Should re-normalize after each LERP
- I posted all my intermediate steps for triangle 0

Review

Now Let's Start On Arbitrary Camera Positions

Note: Ken Joy's graphics notes are fantastic http://www.idav.ucdavis.edu/education/GraphicsNotes/homepage.html

Basic Transforms

Context

- Models stored in “world space”
  - Pick an origin, store all points relative to that origin
- We have been rasterizing in “device space”
- Our goal: transform from world space to device space
- We will do this using matrix multiplications
  - Multiply point by matrix to get new point

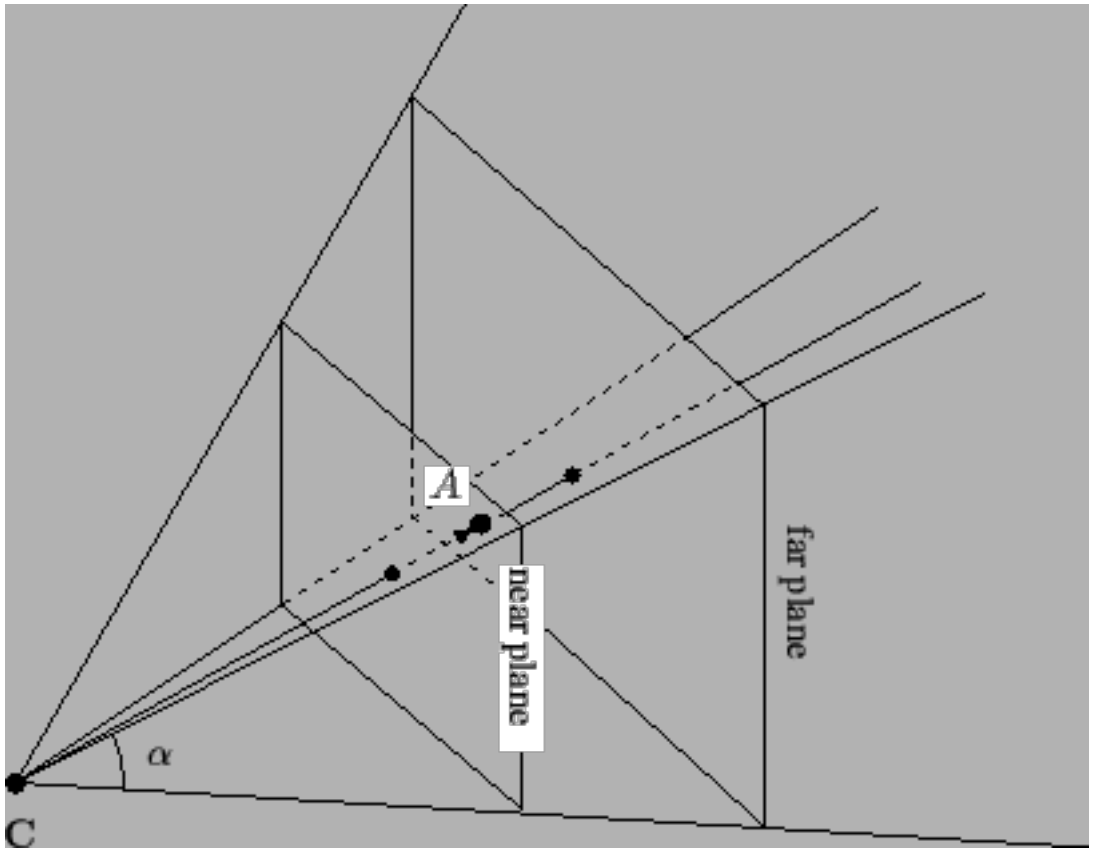
Starting easy … two-dimensional

MP = P’
Matrix M transforms point P to make new point P’.

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \times \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} a*x + b*y \\ c*x + d*y \end{pmatrix}$$

M takes point (x,y)
to point (a*x+b*y, c*x+d*y)

How do we specify a camera?



The “viewing pyramid” or “view frustum”.

Frustum: In geometry, a frustum (plural: frusta or frustums) is the portion of a solid (normally a cone or pyramid) that lies between two parallel planes cutting it.

```
class Camera
{
  public:
    double          near, far;
    double          angle;
    double          position[3];
    double          focus[3];
    double          up[3];
};
```

New terms

- Coordinate system:
  - A system that uses coordinates to establish position
- Example: (3, 4, 6) really means…
  - 3x(1,0,0)
  - 4x(0,1,0)
  - 6x(0,0,1)
- Since we assume the Cartesian coordinate system

New terms

- Frame:
  - A way to place a coordinate system into a specific location in a space
- Cartesian example: (3,4,6)
  - It is assumed that we are speaking in reference to the origin (location (0,0,0)).
- A frame F is a collection of n vectors and a location (v1, v2, …, vn, O) over a space if (v1, v2, … vn) form a basis over that space.
  - What is a basis?? ← linear algebra term

What does it mean to form a basis?

- For any vector v, there are unique coordinates (c1, …, cn) such that

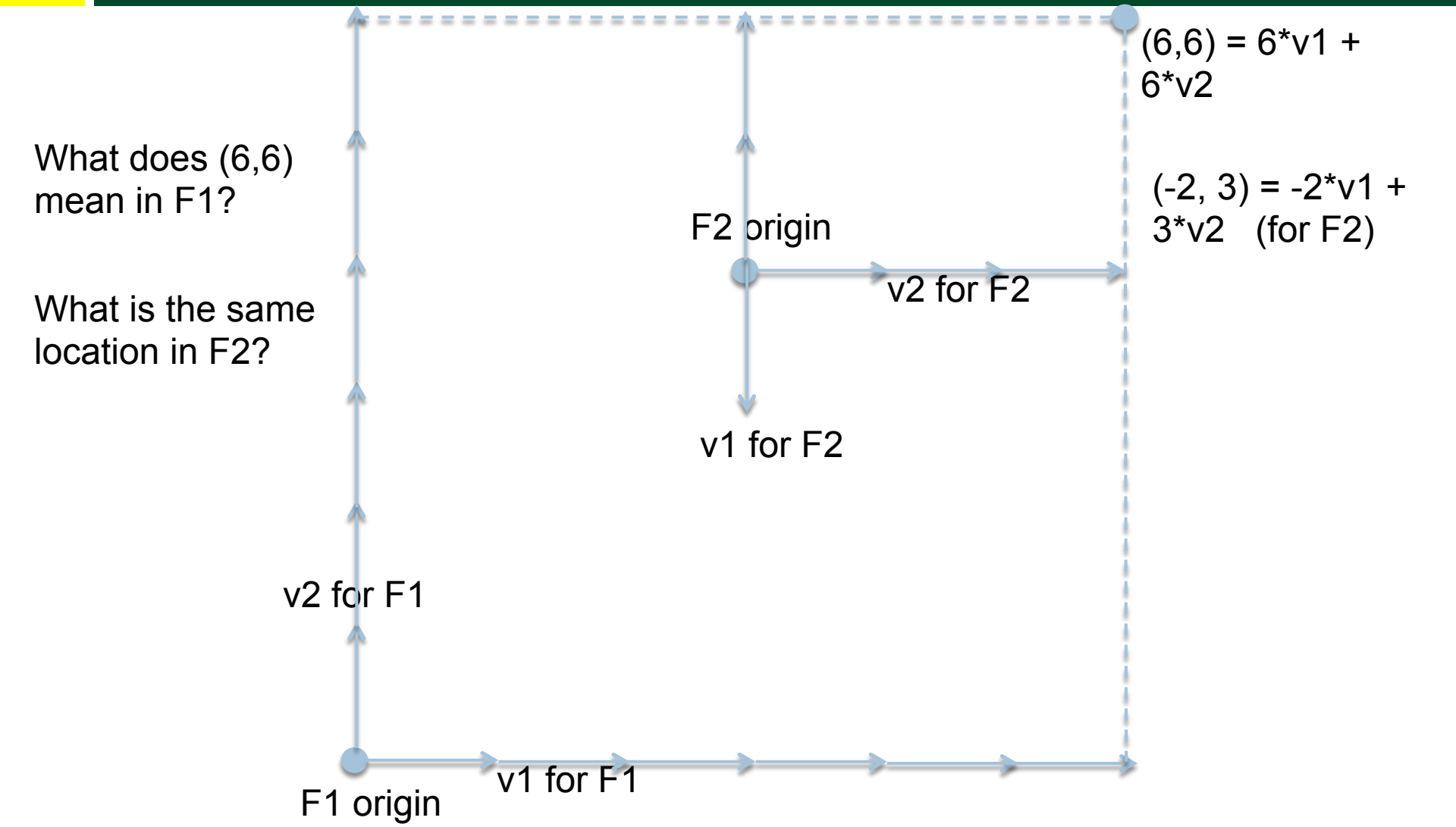
  v = c1*v1 + c2*v2 + … + cn*vn

  - (also more to the definition)
- Consider some point P.
  - The basis has an origin O
  - There is a vector v such that O+v = P
  - We know we can construct v using a combination of vi's
  - Therefore we can represent P in our frame using the coordinates (c1, c2, …, cn)

Example of Frames

- Frame F = (v1, v2, O)
  - v1 = (0, -1)
  - v2 = (1, 0)
  - O = (3, 4)
- What are F's coordinates for the point (6, 6)?

Garett's Question

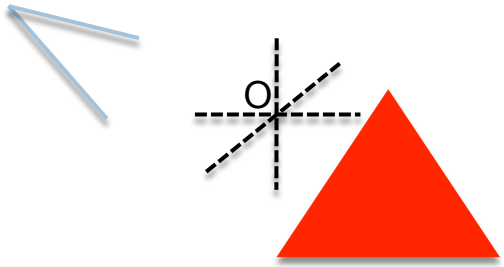


Example of Frames

- Frame F = (v1, v2, O)
  - v1 = (0, -1)
  - v2 = (1, 0)
  - O = (3, 4)
- What are F's coordinates for the point (6, 6)?

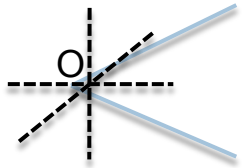
- Answer: (-2, 3)

Our goal



World space:

Triangles in native Cartesian coordinates
Camera located anywhere



Camera space:

Camera located at origin, looking down -Z
Triangle coordinates relative to camera frame

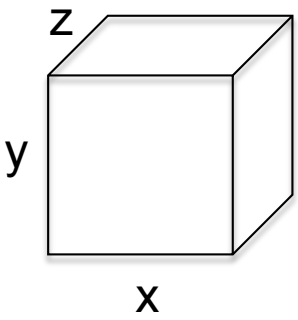
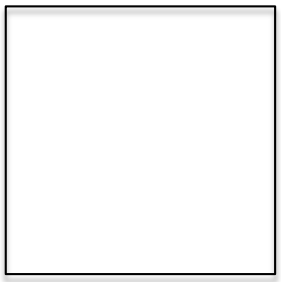


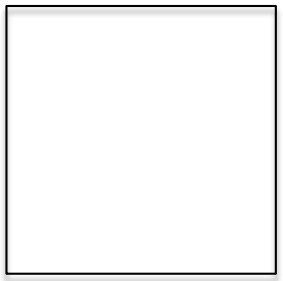
Image space:

All viewable objects within
-1 <= x,y,z <= +1



Screen space:

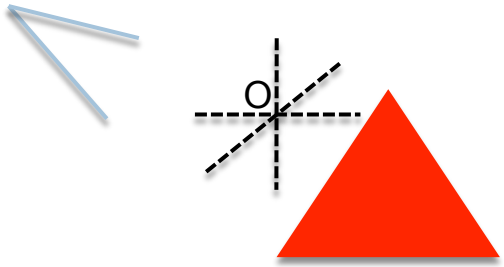
All viewable objects within
-1 <= x, y <= +1



Device space:

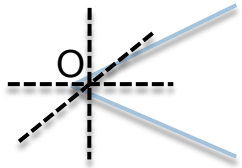
All viewable objects within
0<=x<=width, 0 <=y<=height

Our goal



World space:

Triangles in native Cartesian coordinates
Camera located anywhere



Camera space:

Camera located at origin, looking down -Z
Triangle coordinates relative to camera frame

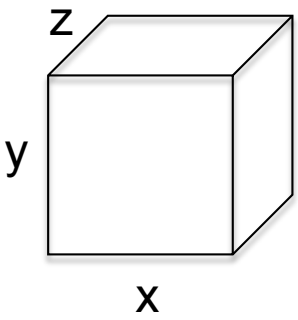
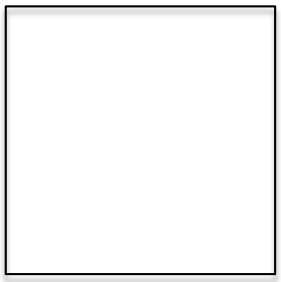


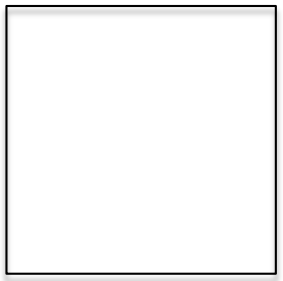
Image space:

All viewable objects within
-1 <= x,y,z <= +1



Screen space:

All viewable objects within
-1 <= x, y <= +1



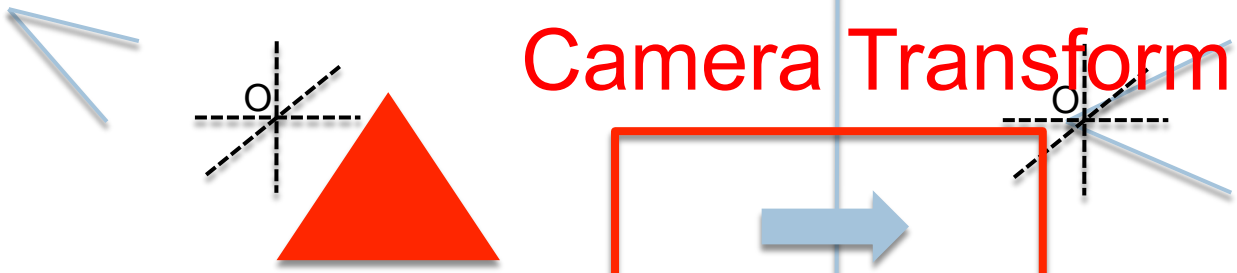
Device space:

All viewable objects within
0<=x<=width, 0 <=y<=height

World Space

- World Space is the space defined by the user's coordinate system.
- This space contains the portion of the scene that is transformed into image space by the camera transform.
- Many of the spaces have "bounds", meaning limits on where the space is valid
- With world space 2 options:
  - No bounds
  - User specifies the bound

Our goal



World space:

Triangles in native Cartesian coordinates
Camera located anywhere

Camera space:

Camera located at origin, looking down -Z
Triangle coordinates relative to camera frame

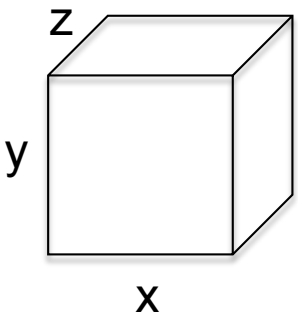
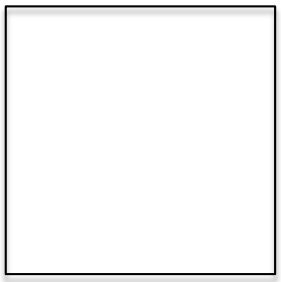


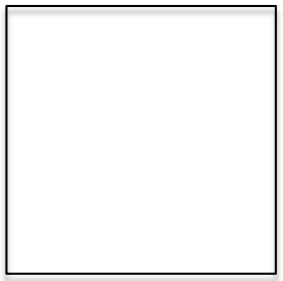
Image space:

All viewable objects within
-1 <= x,y,z <= +1



Screen space:

All viewable objects within
-1 <= x, y <= +1



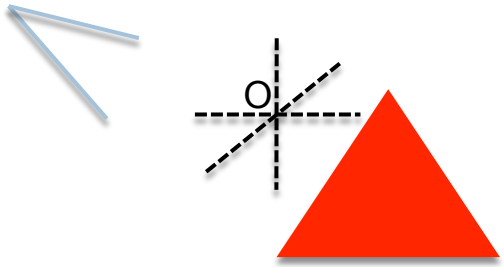
Device space:

All viewable objects within
0<=x<=width, 0 <=y<=height

World Space

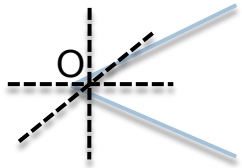
- World Space is the space defined by the user's coordinate system.
- This space contains the portion of the scene that is transformed into image space by the camera transform.
- Many of the spaces have "bounds", meaning limits on where the space is valid
- With world space 2 options:
  - No bounds
  - User specifies the bound

Our goal



World space:

Triangles in native Cartesian coordinates
Camera located anywhere



Camera space:

Camera located at origin, looking down -Z
Triangle coordinates relative to camera frame

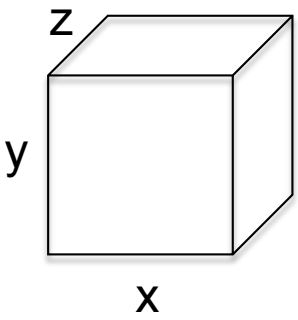
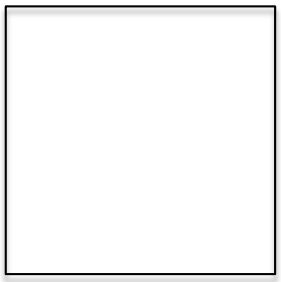


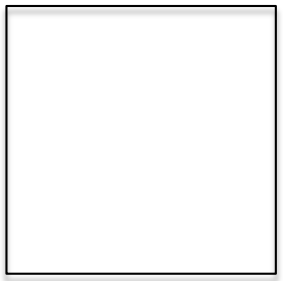
Image space:

All viewable objects within
-1 <= x,y,z <= +1



Screen space:

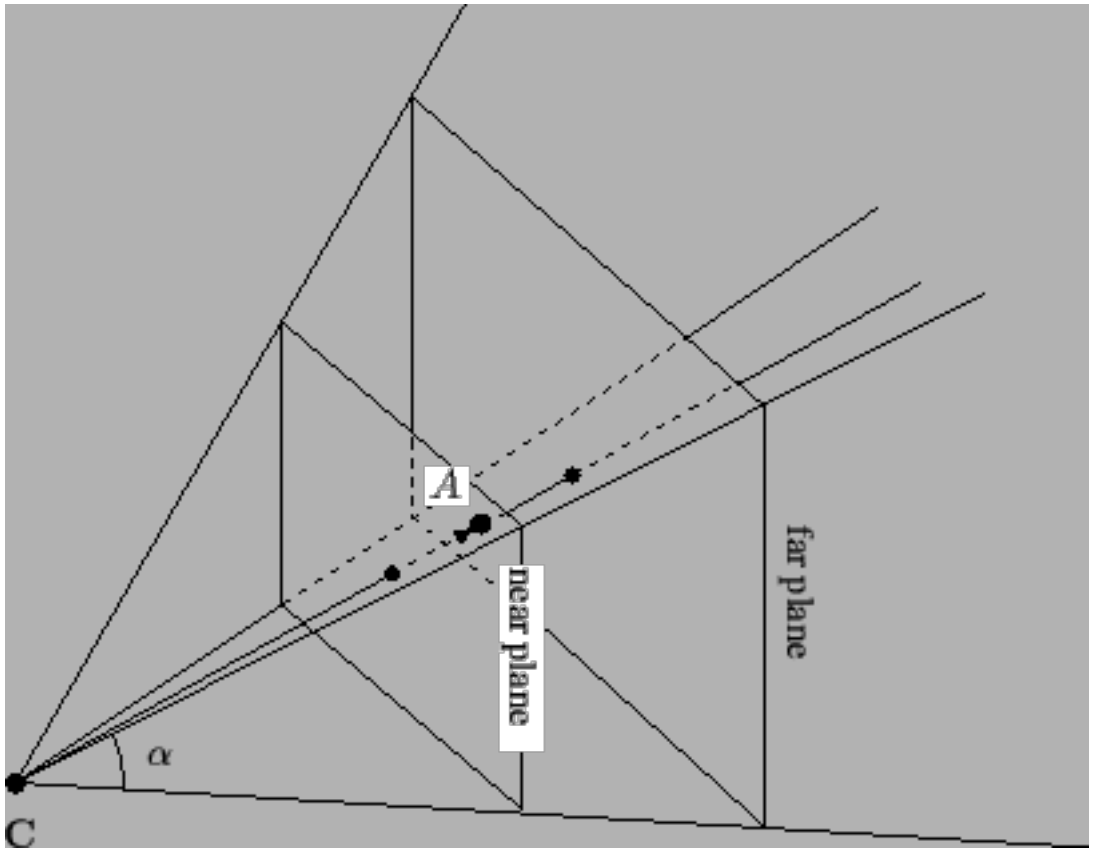
All viewable objects within
-1 <= x, y <= +1



Device space:

All viewable objects within
0<=x<=width, 0 <=y<=height

How do we specify a camera?

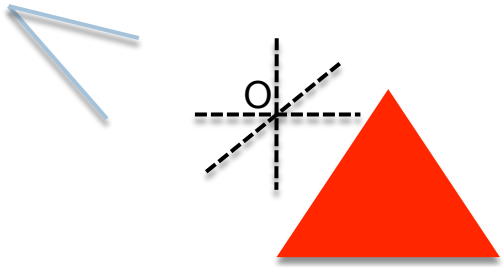


The “viewing pyramid” or “view frustum”.

Frustum: In geometry, a frustum (plural: frusta or frustums) is the portion of a solid (normally a cone or pyramid) that lies between two parallel planes cutting it.

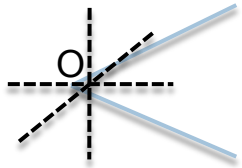
```
class Camera
{
  public:
    double          near, far;
    double          angle;
    double          position[3];
    double          focus[3];
    double          up[3];
};
```

Our goal



World space:

Triangles in native Cartesian coordinates
Camera located anywhere



Camera space:

Camera located at origin, looking down -Z
Triangle coordinates relative to camera frame

View Transform

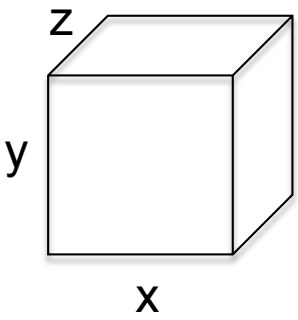
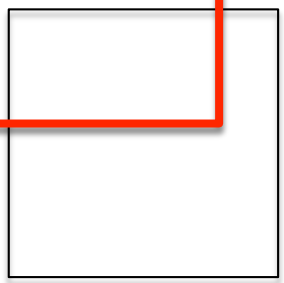


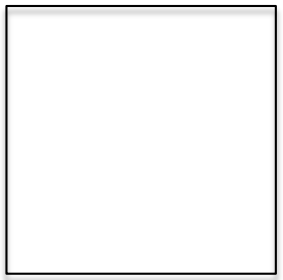
Image space:

All viewable objects within
-1 <= x,y,z <= +1



Screen space:

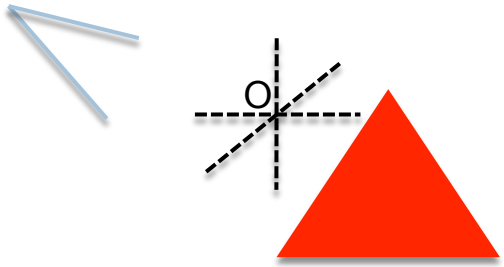
All viewable objects within
-1 <= x, y <= +1



Device space:

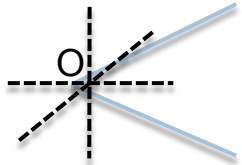
All viewable objects within
0<=x<=width, 0 <=y<=height

Our goal



World space:

Triangles in native Cartesian coordinates
Camera located anywhere



Camera space:

Camera located at origin, looking down -Z
Triangle coordinates relative to camera frame

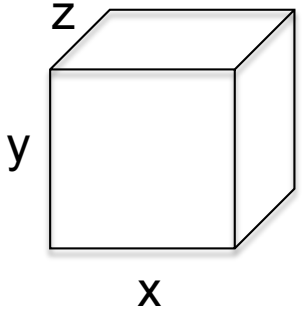
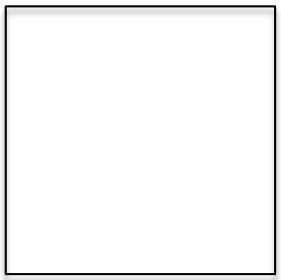


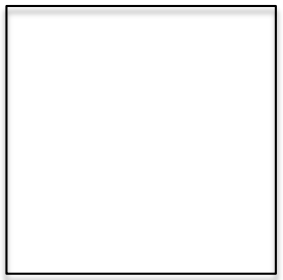
Image space:

All viewable objects within
-1 <= x,y,z <= +1



Screen space:

All viewable objects within
-1 <= x, y <= +1



Device space:

All viewable objects within
0<=x<=width, 0 <=y<=height

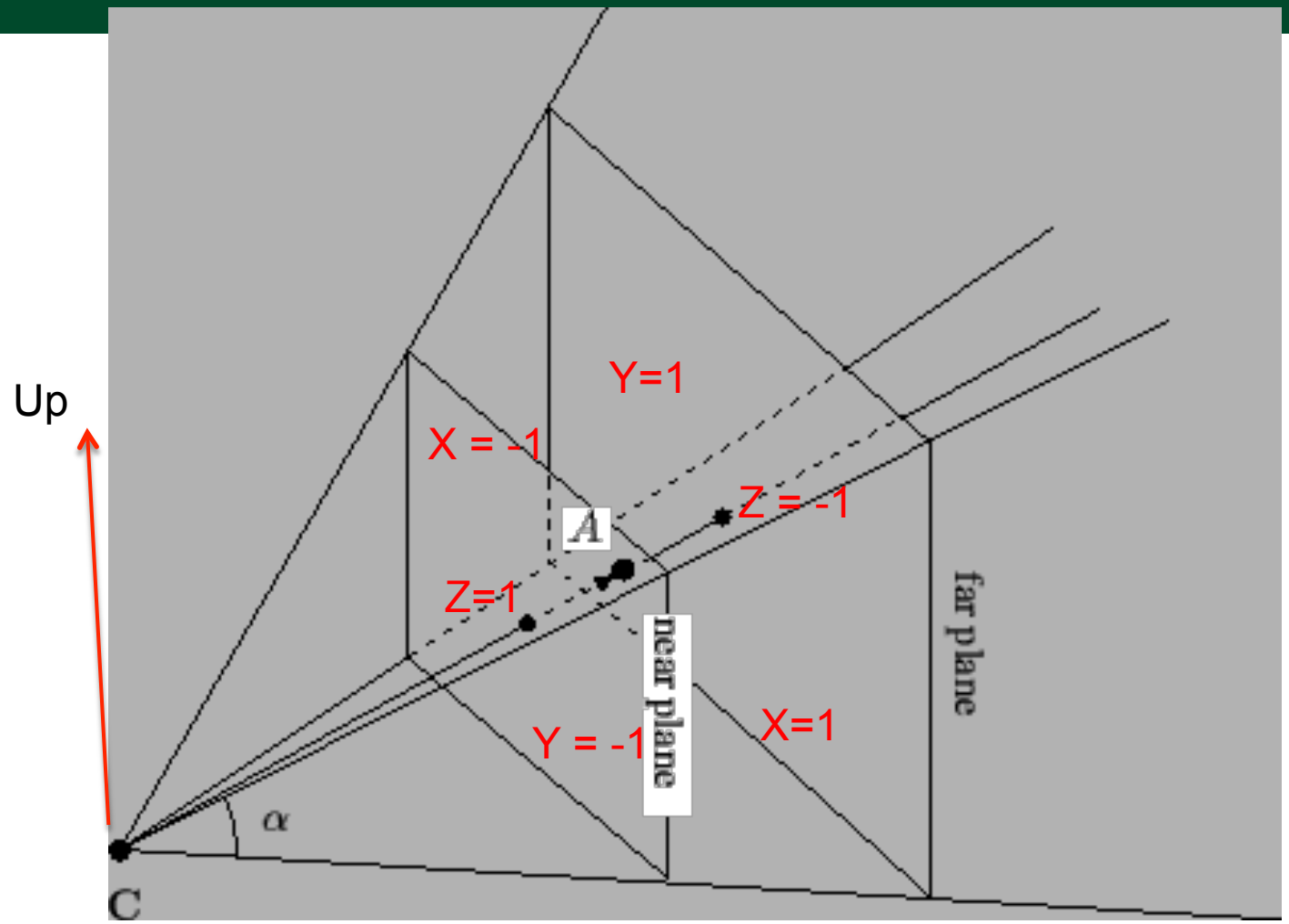
Image Space

- Image Space is the three-dimensional coordinate system that contains screen space.
- It is the space where the camera transformation directs its output.
- The bounds of *Image Space* are 3-dimensional cube.

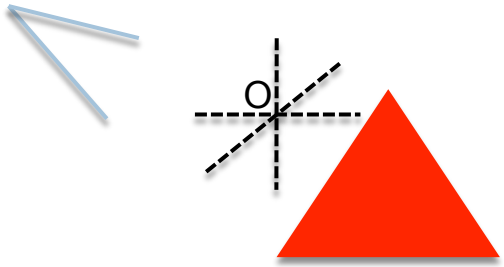
$$\{(x,y,z) : -1 \le x \le 1, -1 \le y \le 1, -1 \le z \le 1\}$$

(or $-1 \le z \le 0$)

Image Space Diagram

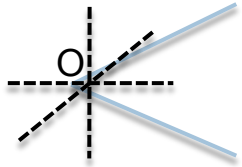


Our goal



World space:

Triangles in native Cartesian coordinates
Camera located anywhere



Camera space:

Camera located at origin, looking down -Z
Triangle coordinates relative to camera frame

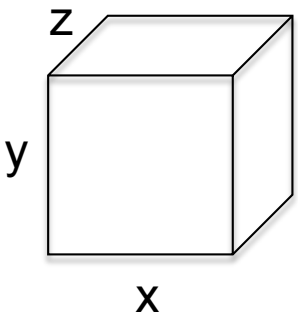
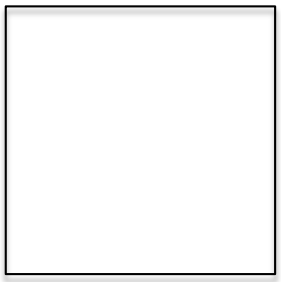


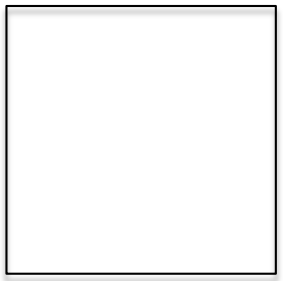
Image space:

All viewable objects within
-1 <= x,y,z <= +1



Screen space:

All viewable objects within
-1 <= x, y <= +1



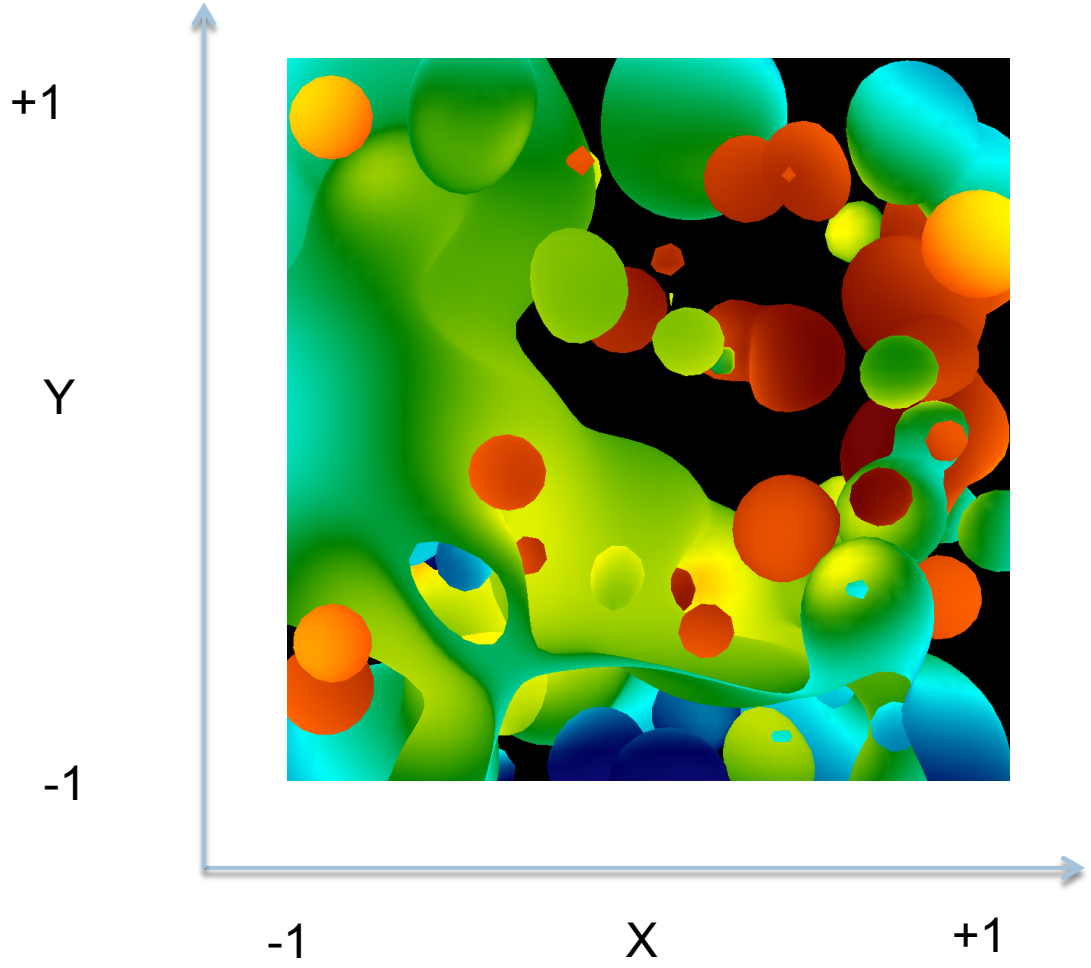
Device space:

All viewable objects within
0<=x<=width, 0 <=y<=height

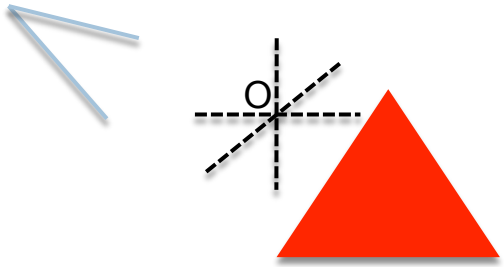
Screen Space

- Screen Space is the intersection of the xy-plane with Image Space.
- Points in image space are mapped into screen space by projecting via a parallel projection, onto the plane $z = 0$.
- Example:
  - a point (0, 0, z) in image space will project to the center of the display screen

Screen Space Diagram

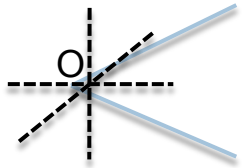


Our goal



World space:

Triangles in native Cartesian coordinates
Camera located anywhere



Camera space:

Camera located at origin, looking down -Z
Triangle coordinates relative to camera frame

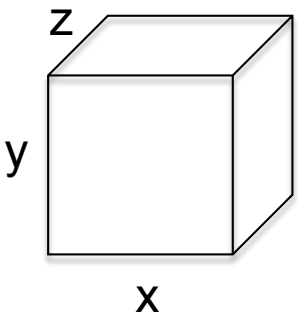
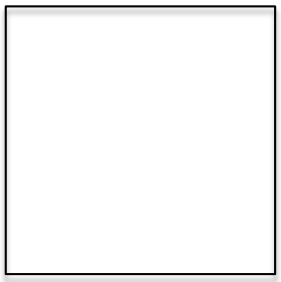


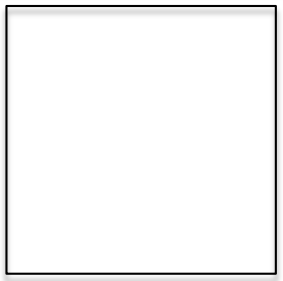
Image space:

All viewable objects within
-1 <= x,y,z <= +1



Screen space:

All viewable objects within
-1 <= x, y <= +1



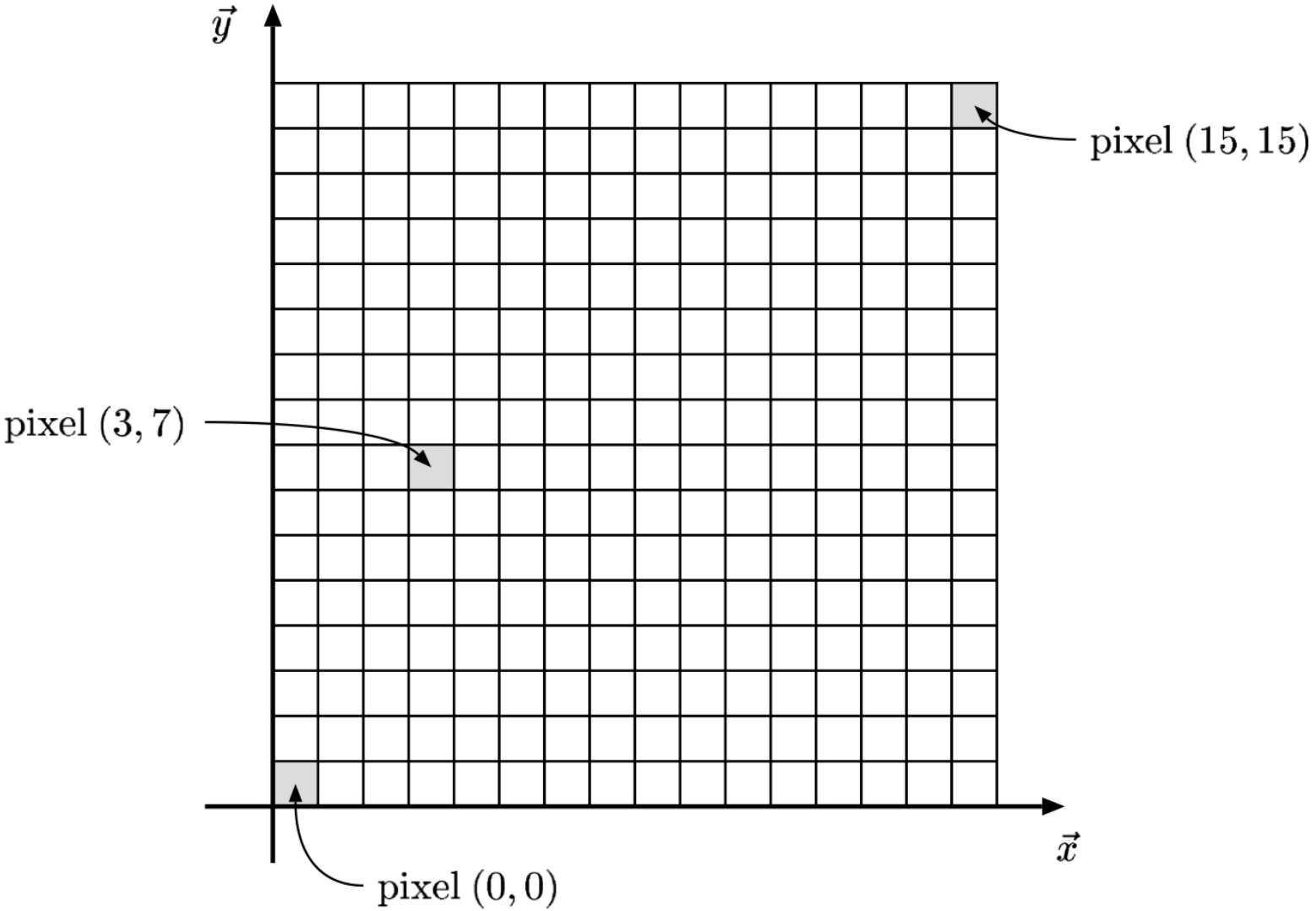
Device space:

All viewable objects within
0<=x<=width, 0 <=y<=height

Device Space

- Device Space is the lowest level coordinate system and is the closest to the hardware coordinate systems of the device itself.
- Device space is usually defined to be the n × m array of pixels that represent the area of the screen.
- A coordinate system is imposed on this space by labeling the lower-left-hand corner of the array as (0,0), with each pixel having unit length and width.

Device Space Example



Device Space With Depth Information

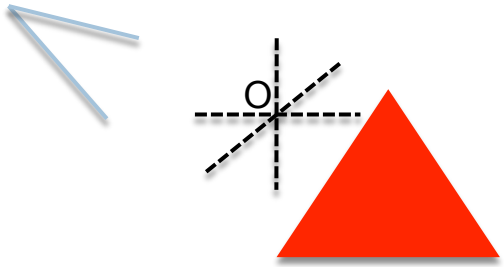
- Extends Device Space to three dimensions by adding z-coordinate of image space.
- Coordinates are (x, y, z) with

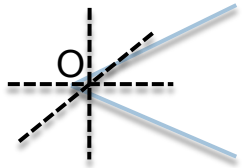
$$0 \leq x \leq n$$

$$0 \leq y \leq m$$

z arbitrary (but typically $-1 \leq z \leq +1$ or $-1 \leq z \leq 0$)

Easiest Transform



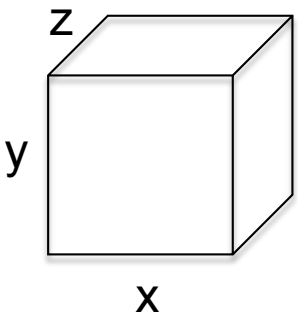


World space:

Triangles in native Cartesian coordinates
Camera located anywhere

Camera space:

Camera located at origin, looking down -Z
Triangle coordinates relative to camera frame



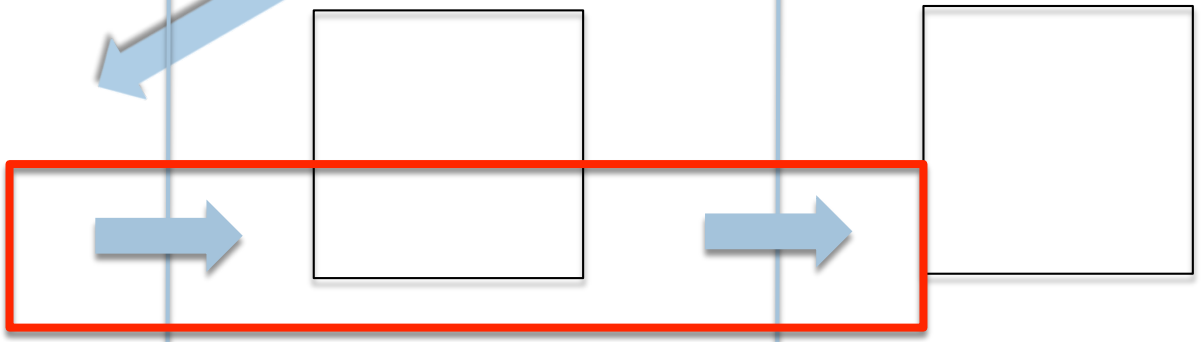


Image space:

All viewable objects within
-1 <= x,y,z <= +1

Screen space:

All viewable objects within
-1 <= x, y <= +1

Device space:

All viewable objects within
0<=x<=width, 0 <=y<=height

Image Space to Device Space

- (x, y, z) → (x', y', z'), where
  - x' = n*(x+1)/2
  - y' = m*(y+1)/2
  - z' = z
  - (for an n x m image)
- Matrix:

  (x' 0 0 0)

  (0 y' 0 0)

  (0 0 z' 0)

  (0 0 0 1)

Translations

- Translation is harder:

$$\begin{pmatrix} a \\ b \end{pmatrix} + \begin{pmatrix} c \\ d \end{pmatrix} = \begin{pmatrix} a+c \\ b+d \end{pmatrix}$$

But this doesn't fit our nice matrix multiply model…
What to do??

Homogeneous Coordinates

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

Add an extra dimension.
A math trick … don't overthink it.

Homogeneous Coordinates

$$\begin{pmatrix} 1 & 0 & dx \\ 0 & 1 & dy \\ 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} x+dx \\ y+dy \\ 1 \end{pmatrix}$$

Translation

We can now fit translation into
our matrix multiplication system.

(now new content)

Homogeneous Coordinates

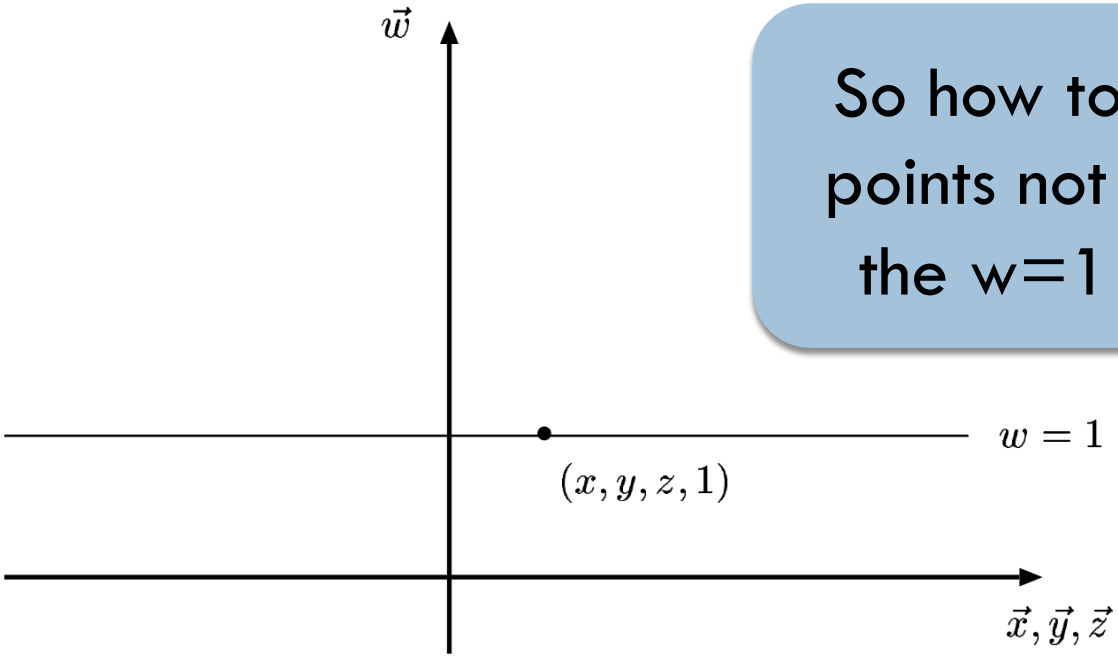
- Defined: a system of coordinates used in projective geometry, as Cartesian coordinates are used in Euclidean geometry
- Primary uses:
  - 4×4 matrices to represent general 3-dimensional transformations
  - it allows a simplified representation of mathematical functions – the rational form – in which rational polynomial functions can be simply represented
- We only care about the first
  - I don't really even know what the second use means

Interpretation of Homogeneous Coordinates

- 4D points: (x, y, z, w)
- Our typical frame: (x, y, z, 1)

Interpretation of Homogeneous Coordinates

- 4D points: (x, y, z, w)
- Our typical frame: (x, y, z, 1)



So how to treat points not along the w=1 line?

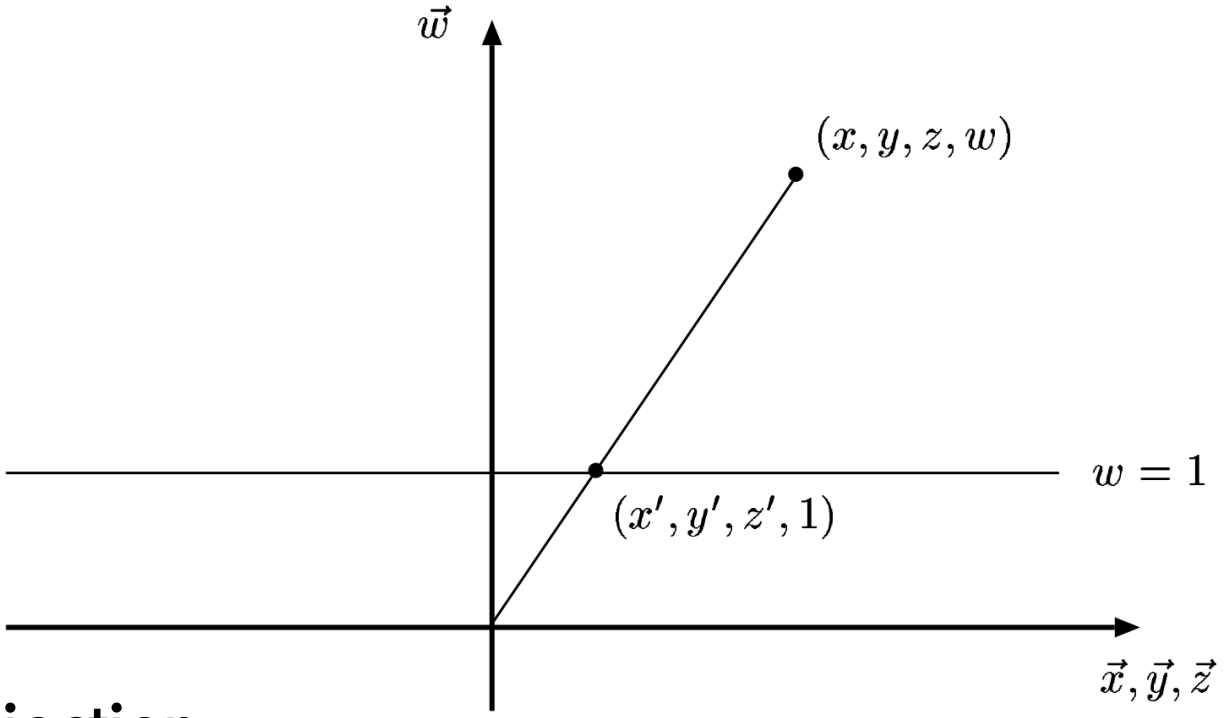
Our typical frame in the context of 4D points

Projecting back to w=1 line

- Let P = (x, y, z, w) be a 4D point with w != 1
- Goal: find P' = (x', y', z', 1) such P projects to P'
  - (We have to define what it means to project)

- Idea for projection:
  - Draw line from P to origin.
  - If Q is along that line (and Q.w == 1), then Q is a projection of P

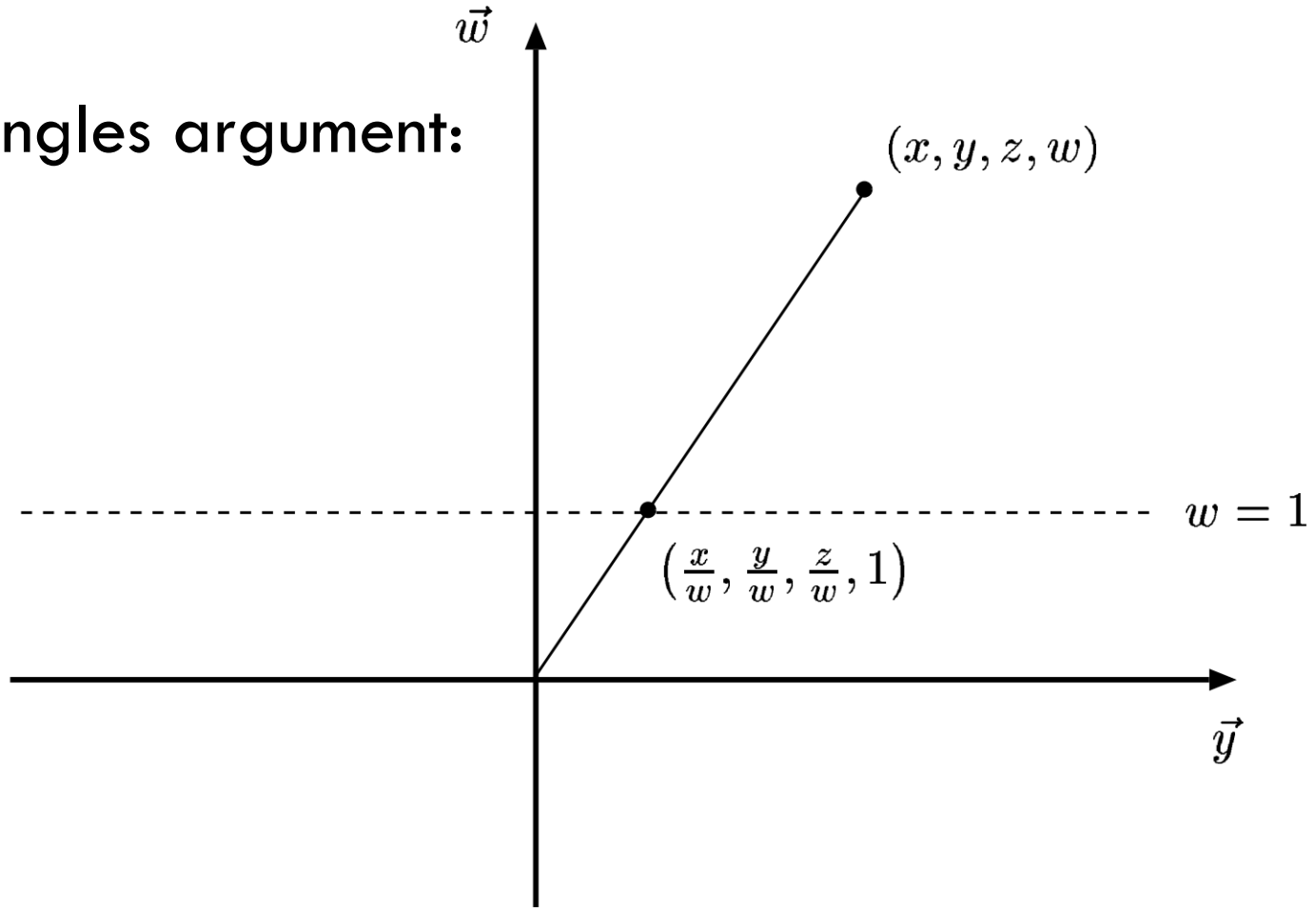
Projecting back to w==1 line



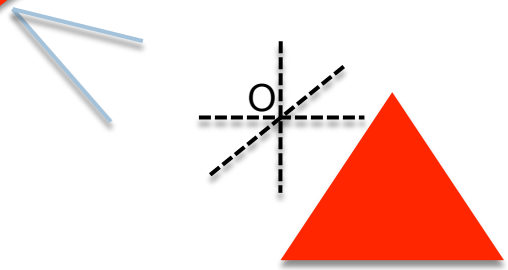
- Idea for projection:
  - Draw line from P to origin.
  - If Q is along that line (and Q.w == 1), then Q is a projection of P

So what is Q?

- Similar triangles argument:
  - x' = x/w
  - y' = y/w
  - z' = z/w

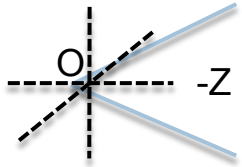


Our goal



World space:

Triangles in native Cartesian coordinates
Camera located anywhere



Camera space:

Camera located at origin, looking down -Z
Triangle coordinates relative to camera frame

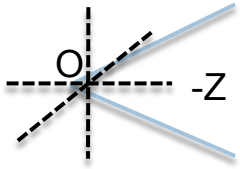
- Need to construct a Camera Frame
- Need to construct a matrix to transform points from Cartesian Frame to Camera Frame
  - Transform triangle by transforming its three vertices

Basis pt 2
(more linear algebra-y this time)

- Camera frame must be a basis:
  - Spans space … can get any point through a linear combination of basis functions
  - Every member must be linearly independent

Camera frame construction

- Must choose (v1,v2,v3,O)



Camera space:

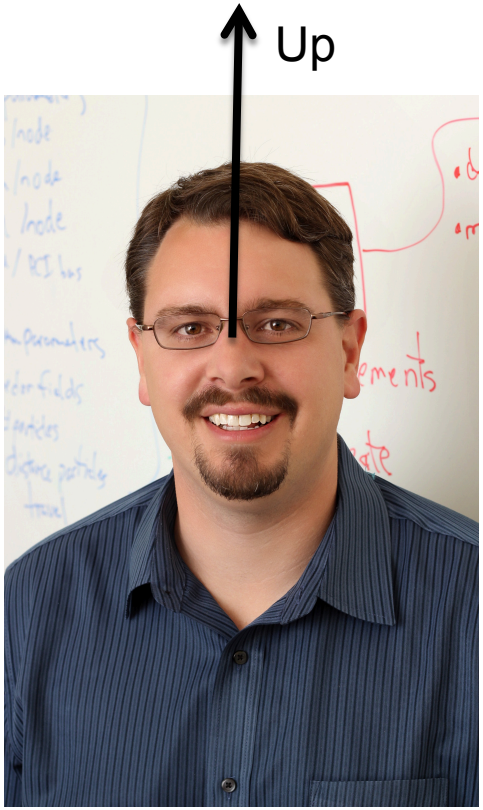
Camera located at origin, looking down -Z
Triangle coordinates relative to camera frame

```
class Camera
{
  public:
    double          near, far;
    double          angle;
    double          position[3];
    double          focus[3];
    double          up[3];
};
```

- O = camera position
- v3 = O-focus
  - Not “focus-O”, since we want to look down -Z

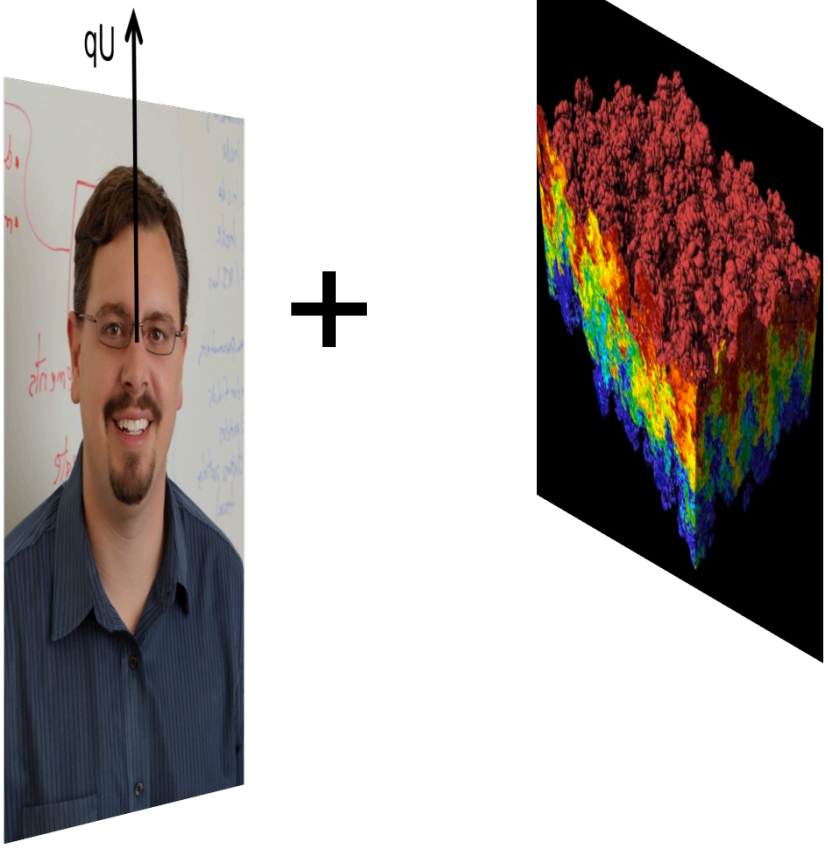
What is the up axis?

- Up axis is the direction from the base of your nose to your forehead



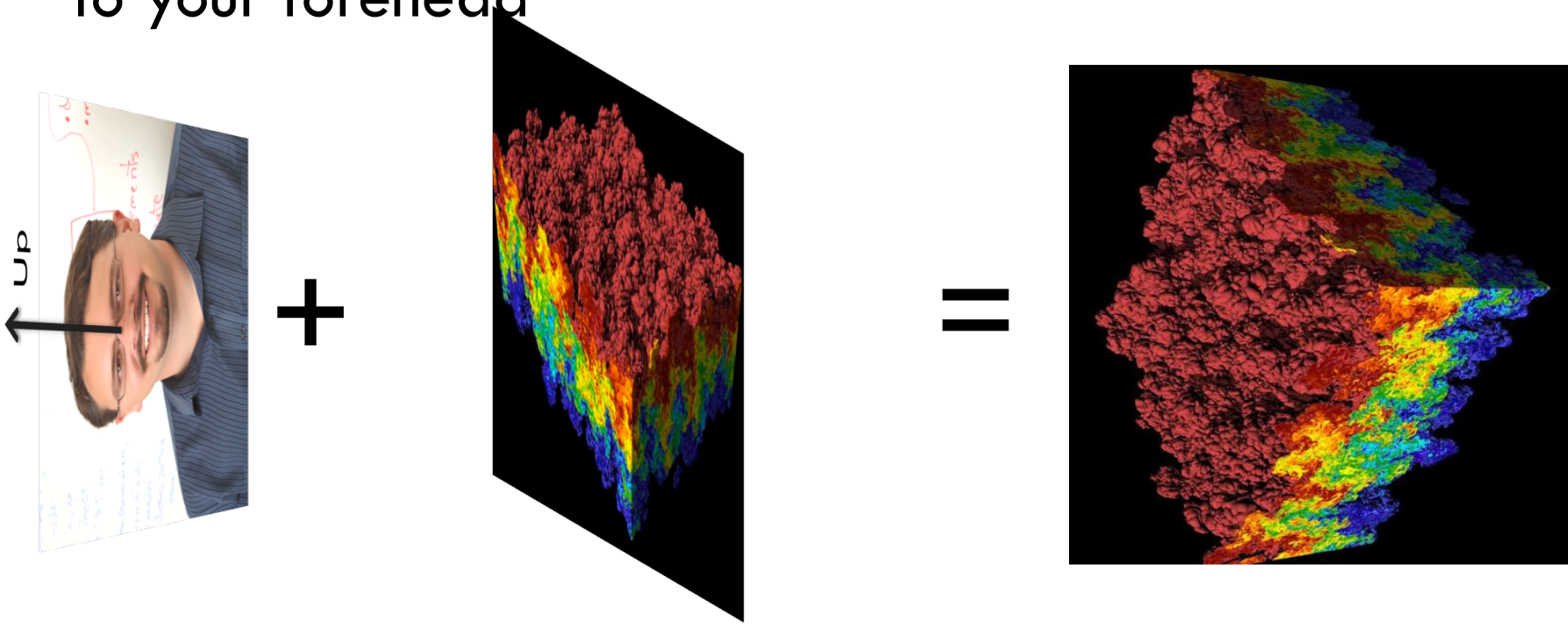
What is the up axis?

- Up axis is the direction from the base of your nose to your forehead



What is the up axis?

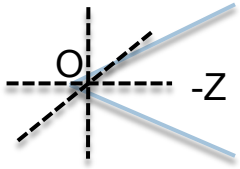
- Up axis is the direction from the base of your nose to your forehead



- (if you lie down while watching TV, the screen is sideways)

Camera frame construction

- Must choose (v1,v2,v3,O)



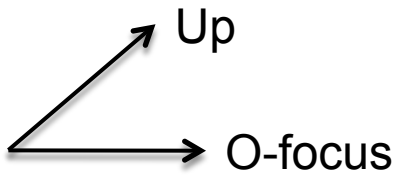
Camera space:

Camera located at origin, looking down -Z
Triangle coordinates relative to camera frame

```
class Camera
{
  public:
    double          near, far;
    double          angle;
    double          position[3];
    double          focus[3];
    double          up[3];
};
```

- O = camera position
- v3 = O-focus
- v2 = up
- v1 = up x (O-focus)

But wait … what if dot(v2,v3) != 0?



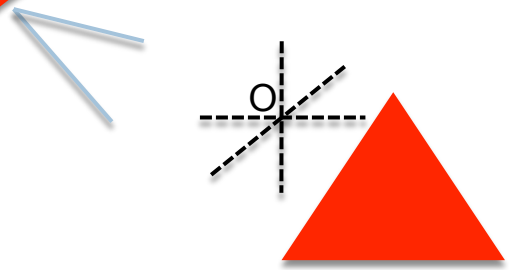
- We can get around this with two cross products:
  - v1 = Up x (O-focus)
  - v2 = (O-focus) x v1

Camera frame summarized

- O = camera position
- v1 = Up x (O-focus)
- v2 = (O-focus) x v1
- v3 = O-focus

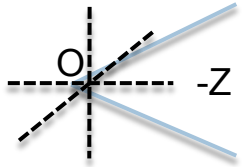
```
class Camera
{
  public:
    double          near, far;
    double          angle;
    double          position[3];
    double          focus[3];
    double          up[3];
};
```

Our goal



World space:

Triangles in native Cartesian coordinates
Camera located anywhere



Camera space:

Camera located at origin, looking down -Z
Triangle coordinates relative to camera frame

- Need to construct a Camera Frame ← ✔
- Need to construct a matrix to transform points from Cartesian Frame to Camera Frame
  - Transform triangle by transforming its three vertices

The Two Frames of the Camera Transform

- Our two frames:

- Cartesian:
  - <1,0,0>
  - <0,1,0>
  - <0,0,1>
  - (0,0,0)

- Camera:
  - v1 = up x (O-focus)
  - v2 = (O-focus) x u
  - v3 = (O-focus)
  - O

The Two Frames of the Camera Transform

- Our two frames:

- Cartesian:
  - <1,0,0>
  - <0,1,0>
  - <0,0,1>
  - (0,0,0)

- Camera:
  - v1 = up x (O-focus)
  - v2 = (O-focus) x u
  - v3 = (O-focus)
  - O

Camera is a Frame, so we can express any vector in Cartesian as a combination of v1, v2, and v3.

Converting From Cartesian Frame To Camera Frame

- The Cartesian vector <1,0,0> can be represented as some combination of the Camera basis functions v1, v2, v3:
  - <1,0,0> = e1,1 * v1 + e1,2 * v2 + e1,3 * v3
- So can the Cartesian vector <0,1,0>
  - <0,1,0> = e2,1 * v1 + e2,2 * v2 + e2,3 * v3
- So can the Cartesian vector <0,0,1>
  - <0,0,1> = e3,1 * v1 + e3,2 * v2 + e3,3 * v3
- So can the vector: Cartesian origin – Camera origin
  - (0,0,0) - O = e4,1 * v1 + e4,2 * v2 + e4,3 * v3 →
  - (0,0,0) = e4,1 * v1 + e4,2 * v2 + e4,3 * v3 + O

Putting Our Equations Into Matrix Form

- <1,0,0> = e1,1 * v1 + e1,2 * v2 + e1,3 * v3
- <0,1,0> = e2,1 * v1 + e2,2 * v2 + e2,3 * v3
- <0,0,1> = e3,1 * v1 + e3,2 * v2 + e3,3 * v3
- (0,0,0) = e4,1 * v1 + e4,2 * v2 + e4,3 * v3 + O
- →
- [<1,0,0>] [e1,1 e1,2 e1,3 0] [v1]
- [<0,1,0>] [e2,1 e2,2 e2,3 0] [v2]
- [<0,0,1>] = [e3,1 e3,2 e3,3 0] [v3]
- (0,0,0) [e4,1 e4,2 e4,3 1] [O]

Here Comes The Trick…

- Consider the meaning of Cartesian coordinates (x,y,z):

[x y z 1][<1,0,0>]

[<0,1,0>] = (x,y,z)

[<0,0,1>]

[(0,0,0)]

But:

[<1,0,0>] [e1,1 e1,2 e1,3 0] [v1]

[<0,1,0>] [e2,1 e2,2 e2,3 0] [v2]

[<0,0,1>] = [e3,1 e3,2 e3,3 0] [v3]

(0,0,0) [e4,1 e4,2 e4,3 1] [O]

Here Comes The Trick…

But:

$$[x\ y\ z\ 1]\begin{bmatrix}<1,0,0> \\ <0,1,0> \\ <0,0,1> \\ (0,0,0)\end{bmatrix} = [x\ y\ z\ 1]\begin{bmatrix}e1,1 & e1,2 & e1,3 & 0 \\ e2,1 & e2,2 & e2,3 & 0 \\ e3,1 & e3,2 & e3,3 & 0 \\ e4,1 & e4,2 & e4,3 & 1\end{bmatrix}\begin{bmatrix}v1 \\ v2 \\ v3 \\ O\end{bmatrix}$$

Coordinates of (x,y,z) with respect to Cartesian frame.

Coordinates of (x,y,z) with respect to Camera frame.
So this matrix is the camera transform!!

And Cramer's Rule Can Solve This, For Example…

$$e_{1,1} = \frac{(< 1,0,0 > \times \vec{v}) \cdot \vec{w}}{(\vec{u} \times \vec{v}) \cdot \vec{w}} \quad ,$$

$$e_{1,2} = \frac{(\vec{u} \times < 1,0,0 >) \cdot \vec{w}}{(\vec{u} \times \vec{v}) \cdot \vec{w}} \quad , \text{and}$$

$$e_{1,3} = \frac{(\vec{u} \times \vec{v}) \cdot < 1,0,0 >}{(\vec{u} \times \vec{v}) \cdot \vec{w}}$$

(u == v1, v == v2, w == v3 from previous slide)

Solving the Camera Transform

$$\begin{bmatrix} e1,1 & e1,2 & e1,3 & 0 \\ e2,1 & e2,2 & e2,3 & 0 \\ e3,1 & e3,2 & e3,3 & 0 \\ e4,1 & e4,2 & e4,3 & 1 \end{bmatrix} = \begin{bmatrix} v1.x & v2.x & v3.x & 0 \\ v1.y & v2.y & v3.y & 0 \\ v1.z & v2.z & v3.z & 0 \\ v1\cdot t & v2\cdot t & v3\cdot t & 1 \end{bmatrix}$$

Where t = (0,0,0)-O

How do we know?: Cramer's Rule + simplifications

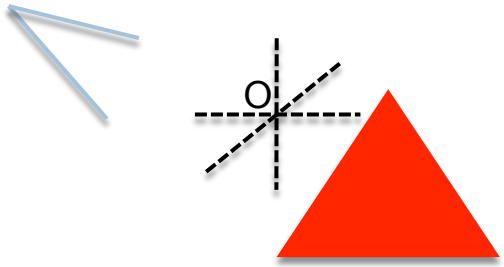
Want to derive?:

http://www.idav.ucdavis.edu/education/GraphicsNotes/Camera-Transform/Camera-Transform.html

NOTE

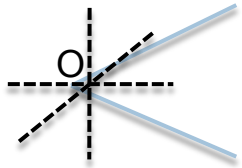
- Motivating examples in previous lectures had points right-multiplying into matrices
- The Camera Transform slides have left-multiplying into matrices

Our goal



World space:

Triangles in native Cartesian coordinates
Camera located anywhere



Camera space:

Camera located at origin, looking down -Z
Triangle coordinates relative to camera frame

View Transform

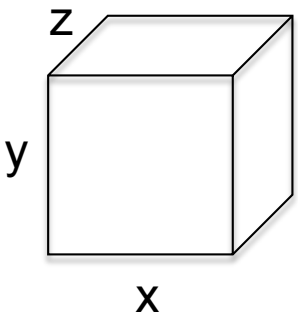
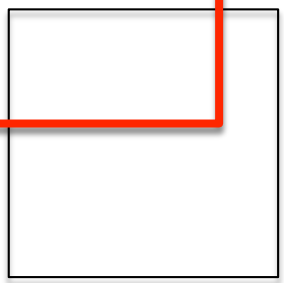


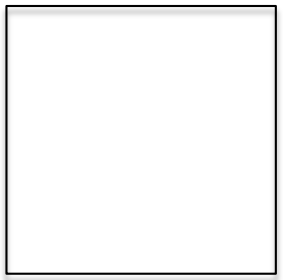
Image space:

All viewable objects within
-1 <= x,y,z <= +1



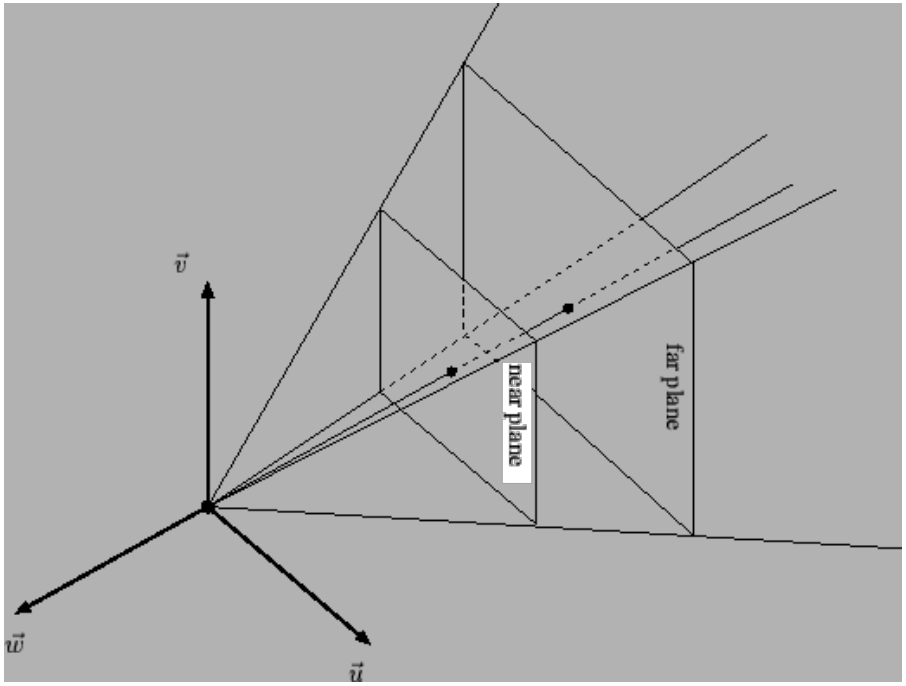
Screen space:

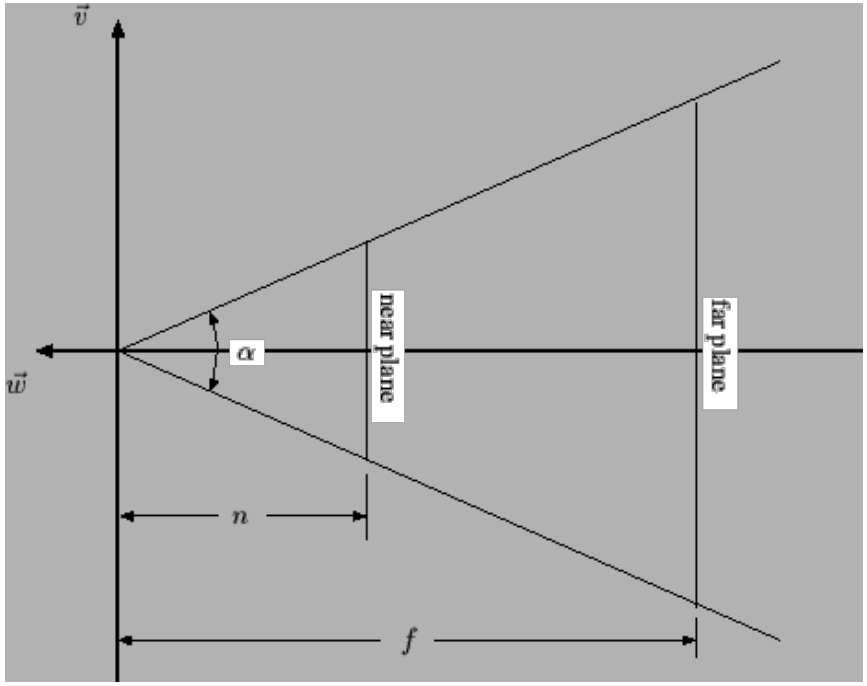
All viewable objects within
-1 <= x, y <= +1



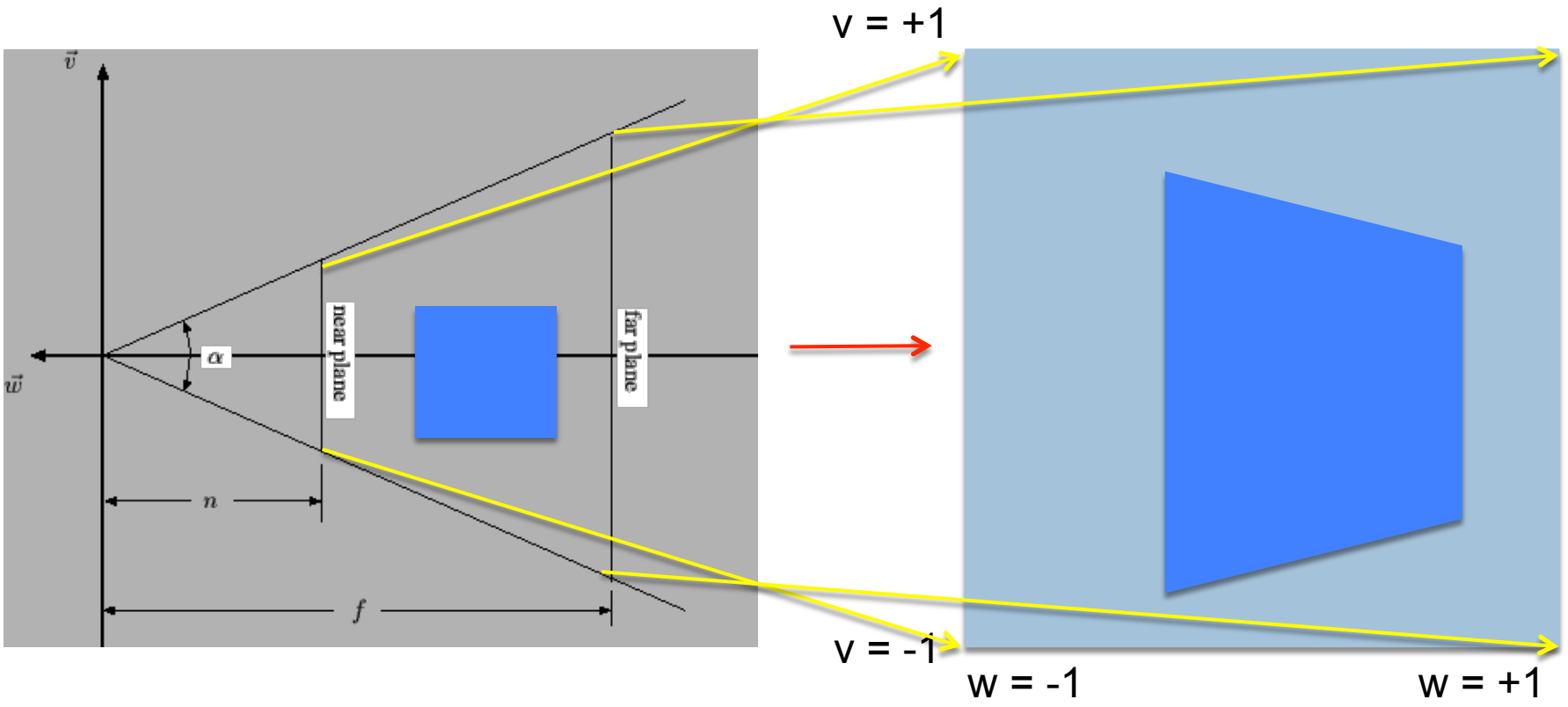
Device space:

All viewable objects within
0<=x<=width, 0 <=y<=height


$\vec{v}$
near plane
far plane
$\vec{w}$
$\vec{u}$


$\vec{v}$
α
near plane
far plane
$\vec{w}$
n
f

View Transformation



The viewing transformation is not a combination of simple translations, rotations, scales or shears: it is more complex.

Derivation of Viewing Transformation

- I personally don't think it is a good use of class time to derive this matrix.
- Well derived at:
  - http://www.idav.ucdavis.edu/education/GraphicsNotes/Viewing-Transformation/Viewing-Transformation.html

The View Transformation

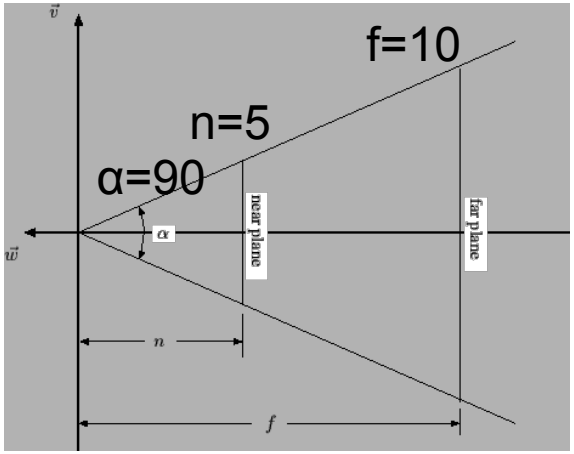
- Input parameters: (α , n, f)
- Transforms view frustum to image space cube
  - View frustum: bounded by viewing pyramid and planes z=-n and z=-f
  - Image space cube: -1 <= u,v,w <= 1

$$\begin{bmatrix} \cot(\alpha/2) & 0 & 0 & 0 \\ 0 & \cot(\alpha/2) & 0 & 0 \\ 0 & 0 & (f+n)/(f-n) & -1 \\ 0 & 0 & 2fn/(f-n) & 0 \end{bmatrix}$$

  - Cotangent = 1/tangent

Let's do an example

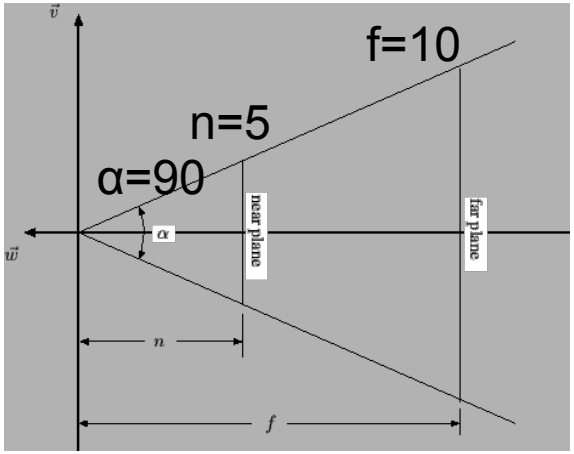
- Input parameters: (α , n, f) = (90, 5, 10)



$$\begin{bmatrix} \cot(\alpha/2) & 0 & 0 & 0 \\ 0 & \cot(\alpha/2) & 0 & 0 \\ 0 & 0 & (f+n)/(f-n) & -1 \\ 0 & 0 & 2fn/(f-n) & 0 \end{bmatrix}$$

Let's do an example

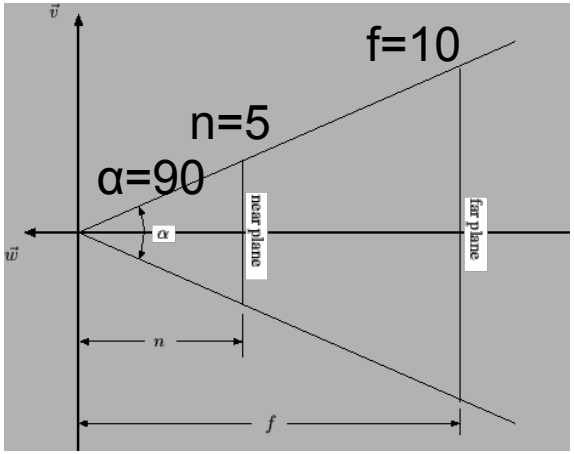
- Input parameters: (α , n, f) = (90, 5, 10)



$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 3 & -1 \\ 0 & 0 & 20 & 0 \end{bmatrix}$$

Let's do an example

- Input parameters: (α , n, f) = (90, 5, 10)

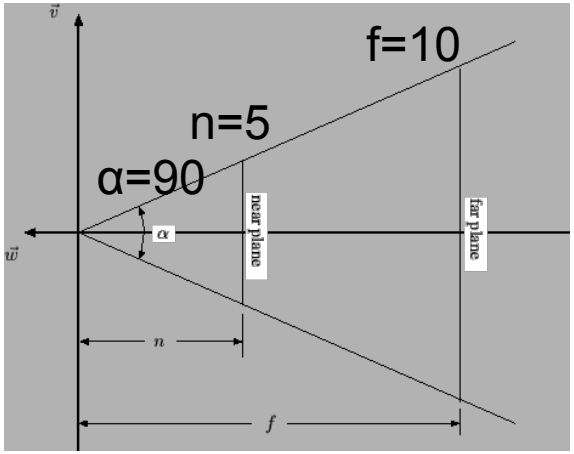


$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 3 & -1 \\ 0 & 0 & 20 & 0 \end{bmatrix}$$

Let's multiply some points:
(0,7,-6,1)
(0,7,-8,1)

Let's do an example

- Input parameters: (α , n, f) = (90, 5, 10)

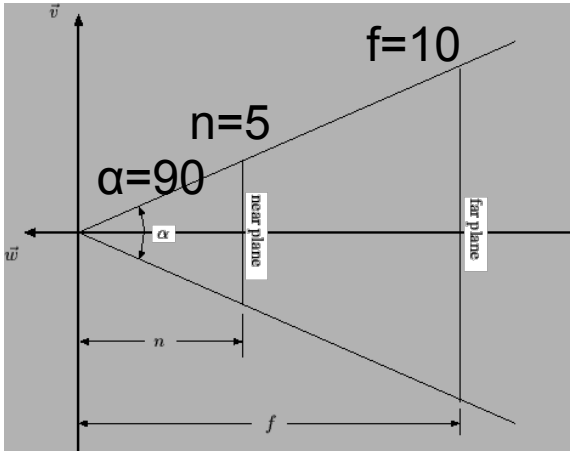


$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 3 & -1 \\ 0 & 0 & 20 & 0 \end{bmatrix}$$

Let's multiply some points:
(0,7,-6,1) = (0,7,-2,6) = (0, 1.16, -0.33)
(0,7,-8,1) = (0,7,4,8) = (0, 0.88, 0.5)

Let's do an example

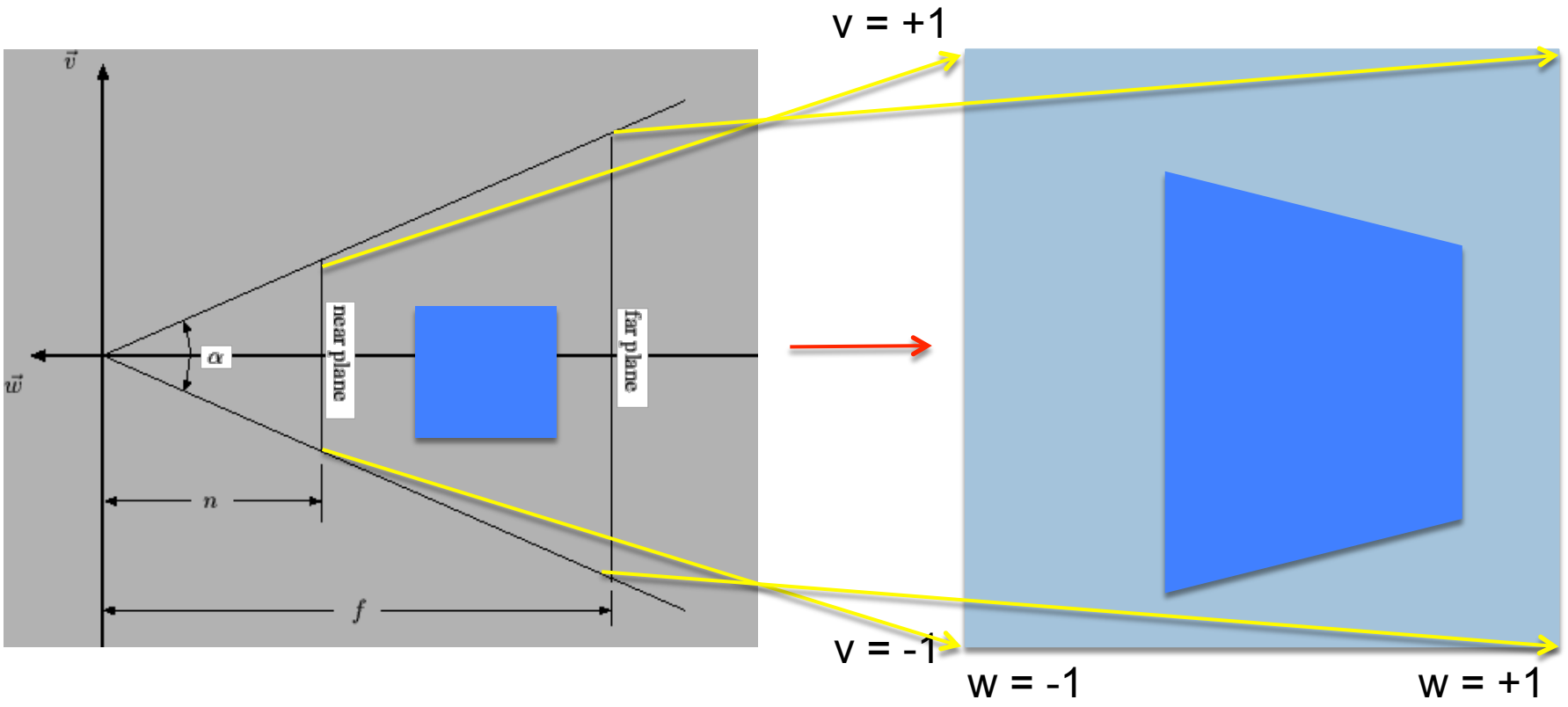
- Input parameters: (α , n, f) = (90, 5, 10)



$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 3 & -1 \\ 0 & 0 & 20 & 0 \end{bmatrix}$$

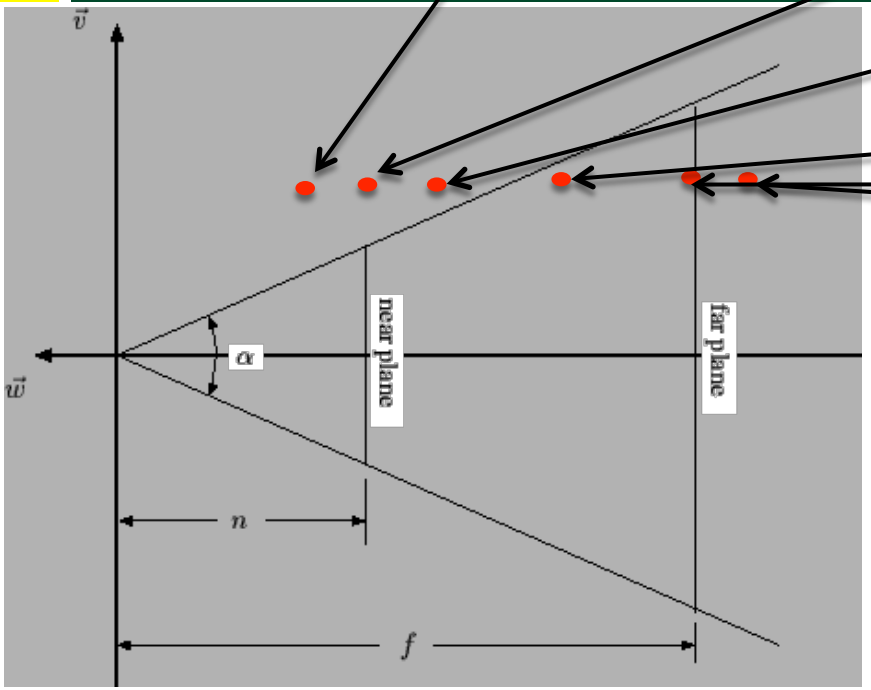
More points:
(0,7,-4,1) = (0,7,-12,4) = (0, 1.75, -3)
(0,7,-5,1) = (0,7,-15,3) = (0, 2.33, -1)
(0,7,-6,1) = (0,7,-2,6) = (0, 1.16, -0.33)
(0,7,-8,1) = (0,7,4,8) = (0, 0.88, 0.5)
(0,7,-10,1) = (0,7,10,10) = (0, 0.7, 1)
(0,7,-11,1) = (0,7,13,11) = (0, .63, 1.18)

View Transformation



The viewing transformation is not a combination of simple translations, rotations, scales or shears: it is more complex.

View Transformation



(0,1.75, -3)

(0,1.4, -1)

(0,1.16, -0.33)

(0,0.88,-0.5)

(0,0.7,1)

(0,0.63,1.18)

v = +1

v = -1

w = -1

w = +1

More points:

(0,7,-4,1) = (0,7,-12,4) = (0, 1.75, -3)

(0,7,-5,1) = (0,7,-5,5) = (0, 1.4, -1)

(0,7,-6,1) = (0,7,-2,6) = (0, 1.16, -0.33)

(0,7,-8,1) = (0,7,4,8) = (0, 0.88, -0.5)

(0,7,-10,1) = (0,7,10,10) = (0, 0.7, 1)

(0,7,-11,1) = (0,7,13,11) = (0, .63, 1.18)

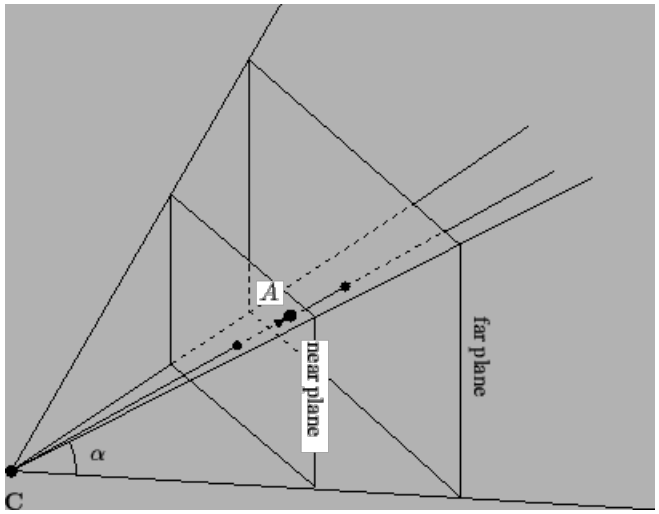
Note there is a non-linear relationship in Z.

Putting It All Together

How do we transform?

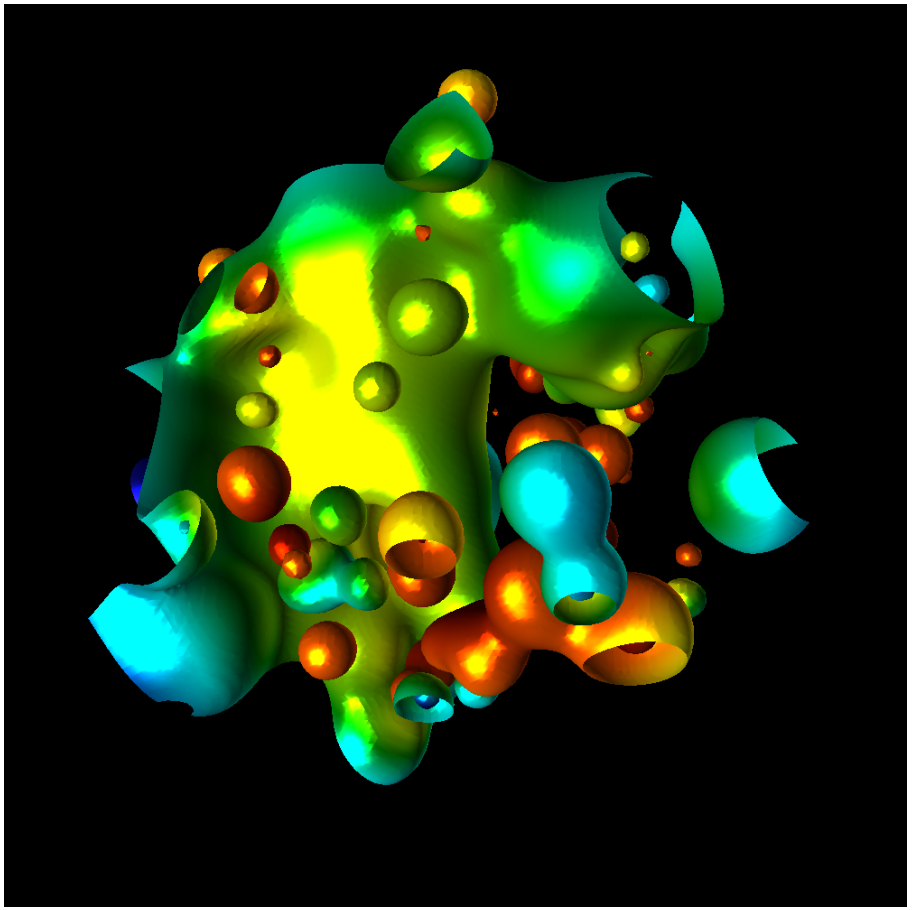
- For a camera C,
  - Calculate Camera Frame
  - From Camera Frame, calculate Camera Transform
  - Calculate View Transform
  - Calculate Device Transform
  - Compose 3 Matrices into 1 Matrix (M)
- For each triangle T, apply M to each vertex of T, then apply rasterization/ zbuffer/Phong shading

```
class Camera
{
  public:
    double          near, far;
    double          angle;
    double          position[3];
    double          focus[3];
    double          up[3];
};
```



Project 1F

Project #1F (8%), Due Oct 30th



- Goal: add arbitrary camera positions
- Extend your project1E code
- Re-use:

  proj1e_geometry.vtk available on web (9MB), "reader1e.cxx", "shading.cxx".
- No Cmake, project1F.cxx
- New: Matrix.cxx, Camera.cxx

Project #1F, expanded

- Matrix.cxx: complete
- Methods:

```
class Matrix
{
  public:
    double         A[4][4];

    void           TransformPoint(const double *ptIn, double *ptOut);
    static Matrix  ComposeMatrices(const Matrix &, const Matrix &);
    void           Print(ostream &o);
};
```

Project #1F, expanded

- Camera.cxx: you work on this

```
class Camera
{
  public:
    double          near, far;
    double          angle;
    double          position[3];
    double          focus[3];
    double          up[3];

    Matrix          ViewTransform(void) {;};
    Matrix          CameraTransform(void) {;};
    Matrix          DeviceTransform(void) {;};
    // Will probably need something for calculating Camera Frame as well
};
```

Also: GetCamera(int frame, int nFrames)

Project #1F, deliverables

- Same as usual, but times 4
  - 4 images, corresponding to
    - GetCamera(0, 1000)
    - GetCamera(250,1000)
    - GetCamera(500,1000)
    - GetCamera(750,1000)
- If you want:
  - Generate all thousand images, make a movie
    - Can discuss how to make a movie if there is time

Project #1F, game plan

```
vector<Triangle> t = GetTriangles();
AllocateScreen();
for (int i = 0 ; i < 1000 ; i++)
{
    InitializeScreen();
    Camera c = GetCamera(i, 1000);
    TransformTrianglesToDeviceSpace(); // involves setting up and applying matrices
                                       //… if you modify vector<Triangle> t,
                                       // remember to undo it later
    RenderTriangles()
    SaveImage();
}
```

Correct answers given for GetCamera(0, 1000)

Camera Frame: U = 0, 0.707107, -0.707107
Camera Frame: V = -0.816497, 0.408248, 0.408248
Camera Frame: W = 0.57735, 0.57735, 0.57735
Camera Frame: O = 40, 40, 40
Camera Transform
(0.0000000 -0.8164966 0.5773503 0.0000000)
(0.7071068 0.4082483 0.5773503 0.0000000)
(-0.7071068 0.4082483 0.5773503 0.0000000)
(0.0000000 0.0000000 -69.2820323 1.0000000)
View Transform
(3.7320508 0.0000000 0.0000000 0.0000000)
(0.0000000 3.7320508 0.0000000 0.0000000)
(0.0000000 0.0000000 1.0512821 -1.0000000)
(0.0000000 0.0000000 10.2564103 0.0000000)
Transformed 37.1132, 37.1132,37.1132, 1 to 0, 0,1
Transformed -75.4701, -75.4701,-75.4701, 1 to 0, 0,-1

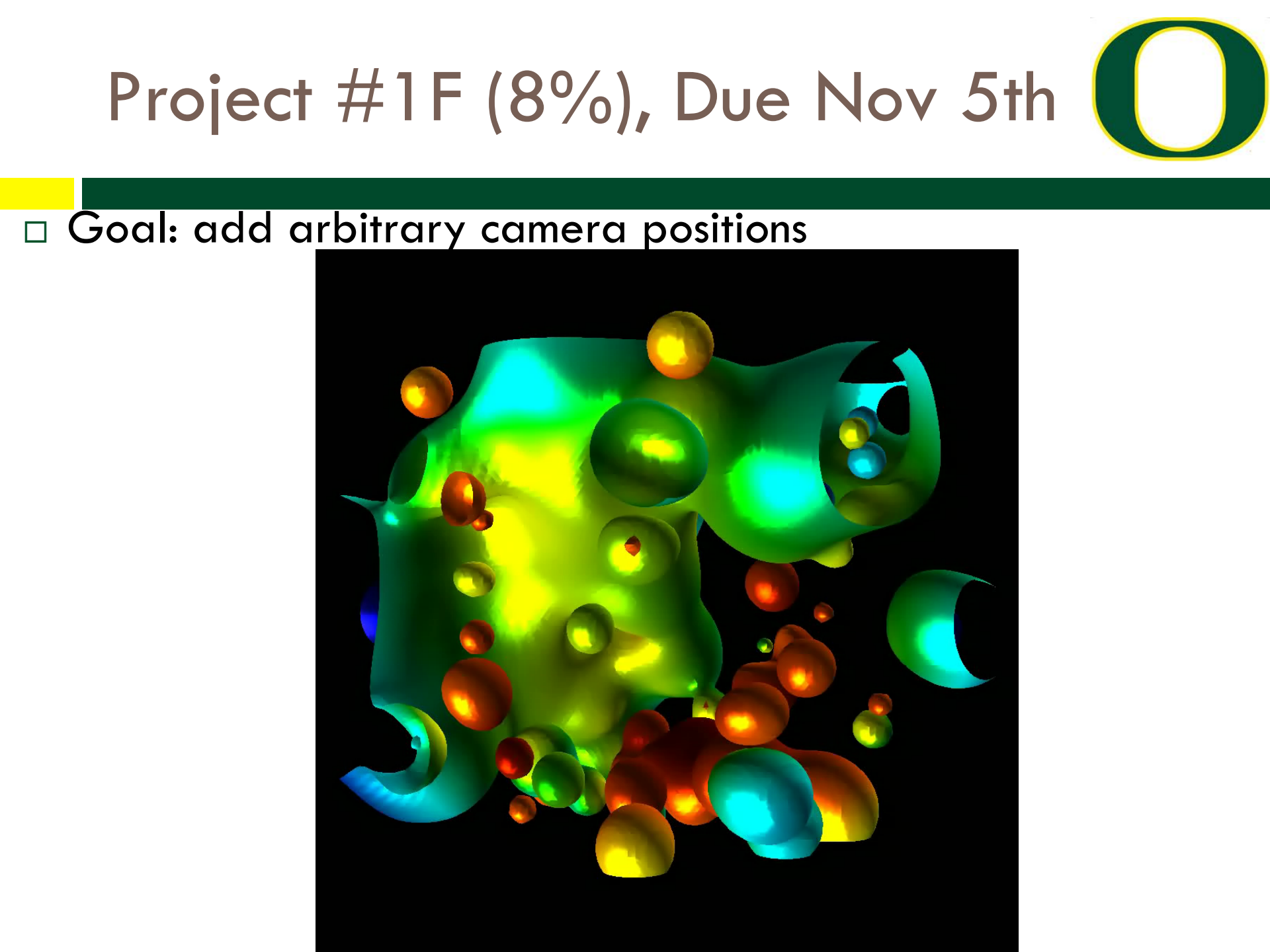
Project #1F, pitfalls

- All vertex multiplications use 4D points. Make sure you send in 4D points for input and output, or you will get weird memory errors.
  - Make sure you divide by w.
- Your Phong lighting assumed a view of (0,0,-1). The view will now be changing with each render and you will need to incorporate that view direction in your rendering.

Project #1F, pitfalls

- People often get a matrix confused with its transpose. Use the method Matrix::Print() to make sure the matrix you are setting up is what you think it should be. Also, remember the points are left multiplied, not right multiplied.
- Regarding multiple renderings:
  - Don't forget to initialize the screen between each render
  - If you modify the triangle in place to render, don't forget to switch it back at the end of the render

Project #1F (8%), Due Nov 5th



- Goal: add arbitrary camera positions

Project 1F: Making a Movie

- You can also generate 1000 images and use a movie encoder to make a movie (i.e., ffmpeg to make a mpeg)
  - Could someone post a how to?

Project 1F: Shading

- We used viewDirection = (0, 0, -1) for 1E
- For 1F, we will do it right:
  - Prior to transforming, you can calculate viewDirection
    - triangle vertex minus camera position
    - then call CalculateShading with correct viewDir
    - then associate shading value as a scale on the triangle
    - and then LERP that shading value across scanline