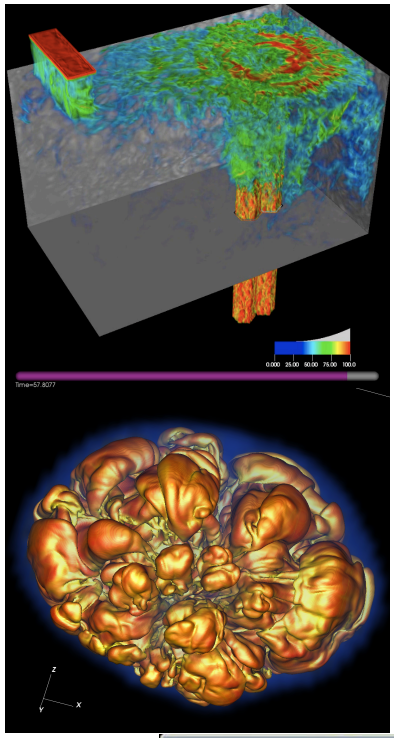
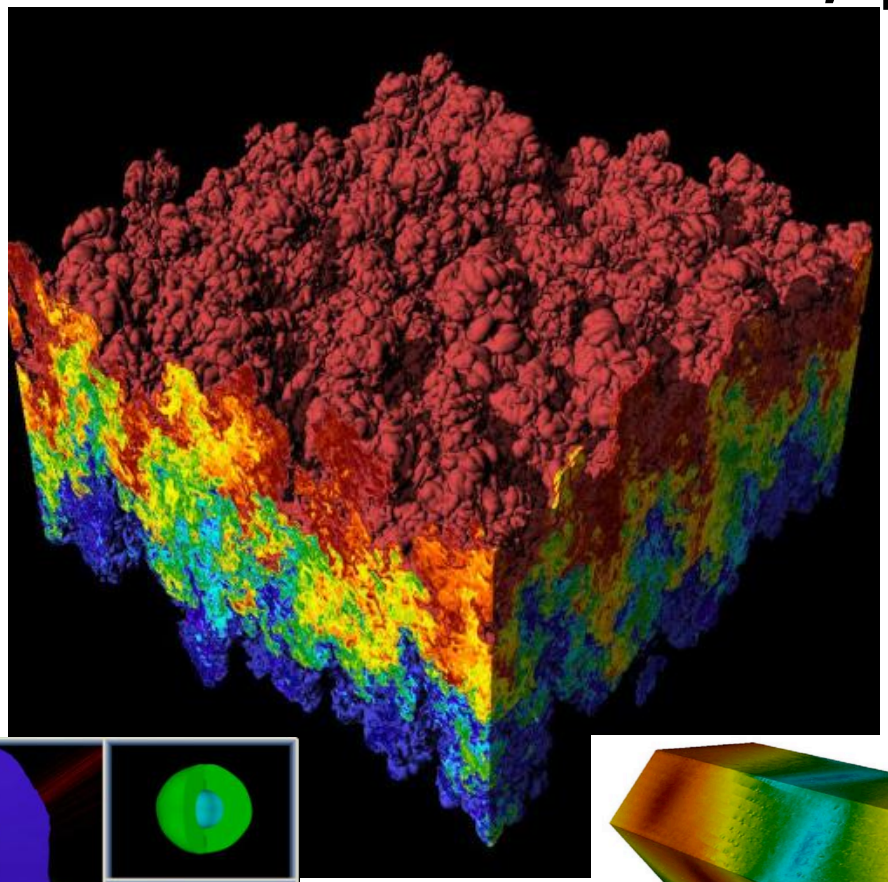
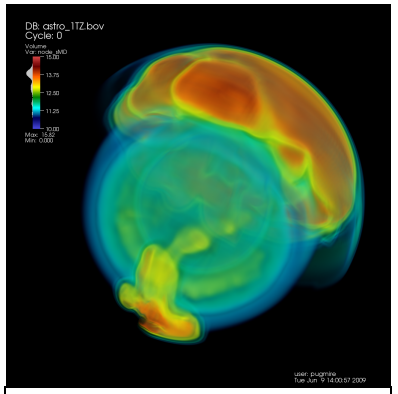
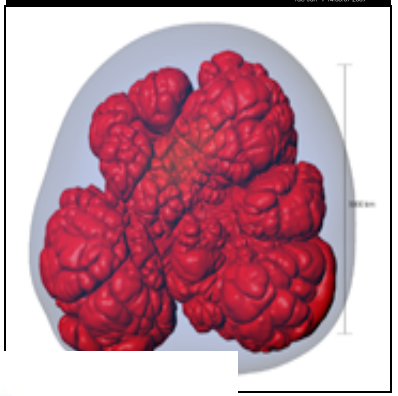
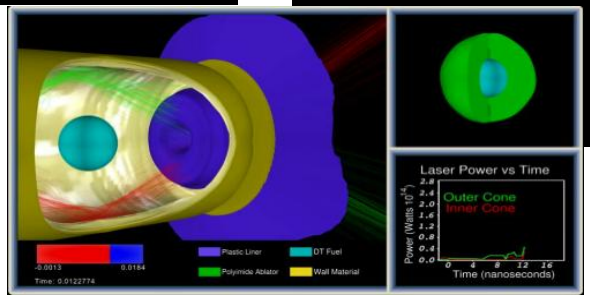
CIS 441/541: Intro to Computer Graphics
Lecture 6: Transforms, pt 2

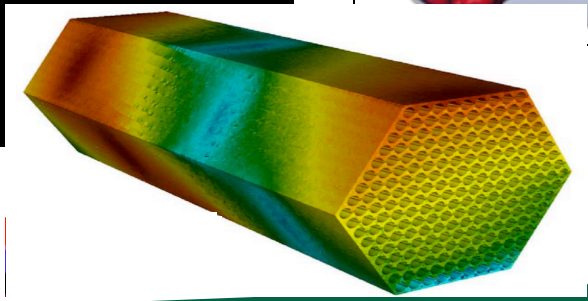












Office Hours: Weeks 4-10

- Monday: 1-2 (Roscoe)
- Tuesday: 1-2 (Roscoe)
- Wednesday: 1-3 (Roscoe)
- Thursday: 1130-1230 (Hank)
- Friday: 1130-1230 (Hank)

- All normal this week!!! ☺

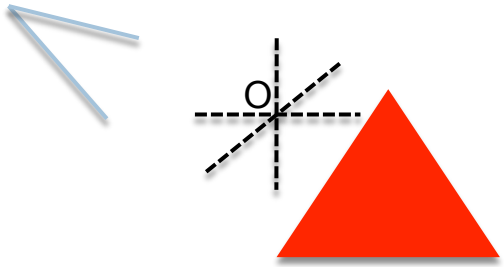
Timeline

- ~~1C: due Weds Jan 23rd~~
- 1D: assigned ~~today~~ (LAST TUESDAY), due Thurs Jan 31st
- 1E: assigned Thurs Jan 31st, due Weds Feb 6th
  - → will be extra support with this. Tough project.
- 1F: assigned Feb 7th, due Feb 19th
  - → not as tough as 1E
- 2A: will be assigned during week of Feb 11th

Sun	Mon	Tues	Weds	Thurs	Fri	Sat
Jan 20	Jan21	Jan 22 Lec4	Jan 23 1C due	Lec 5 1D assigned	Jan 25	Jan 26
Jan 27	Jan 28	Jan 29 (YouTube)	Jan 30	Lec 6 1D due 1E assigned	Feb 1	Feb 2
Feb 3	Feb 4	Feb 5 Lec 7	Feb 6 1E due	Feb 7 1F assigned	Feb 8	Feb 9

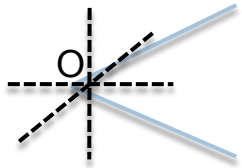
Likely: pre-SuperBowl OH

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

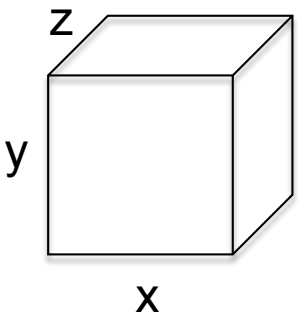
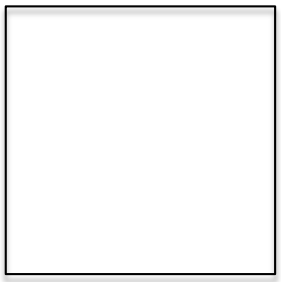


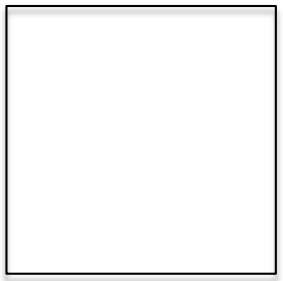
Image space:

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



Screen space:

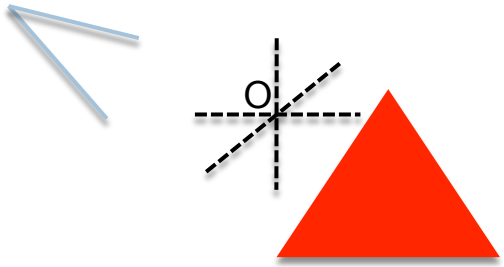
All viewable objects within
-1 <= x, y <= +1



Device space:

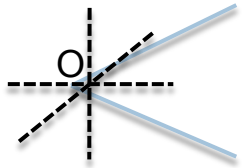
All viewable objects within
0<=x<=width, 0 <=y<=height

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

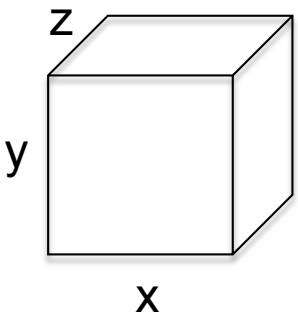
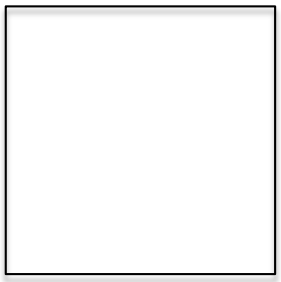


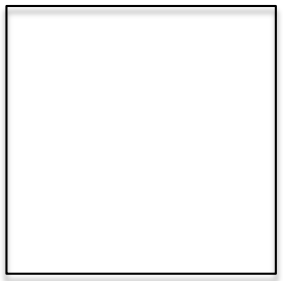
Image space:

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



Screen space:

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



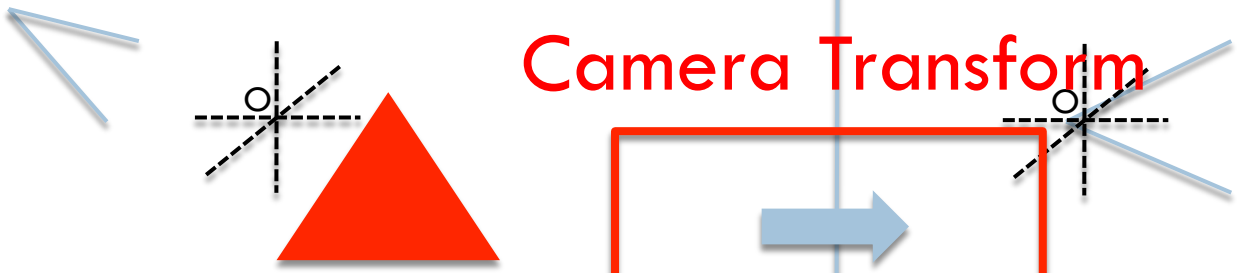
Device space:

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 camera 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



Camera 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

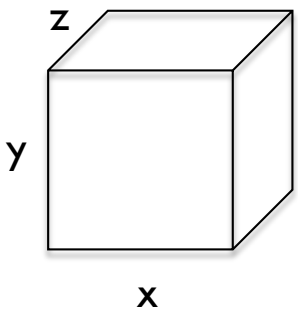
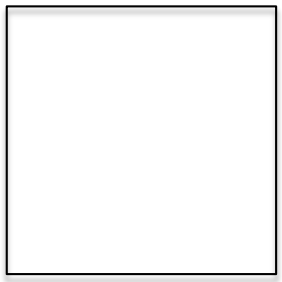


Image space:

All viewable objects within

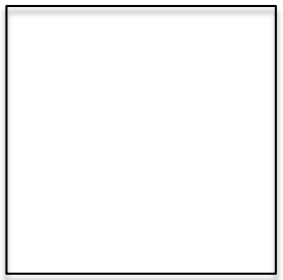
-1 <= x,y,z <= +1



Screen space:

All viewable objects within

-1 <= x, y <= +1

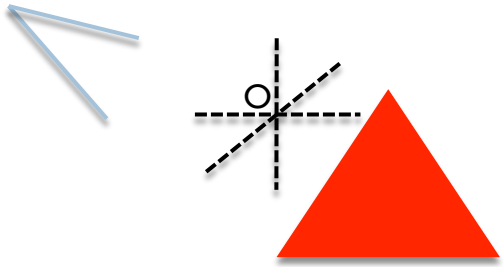


Device space:

All viewable objects within

0<=x<=width, 0 <=y<=height

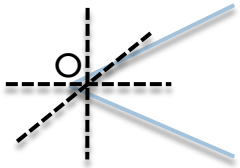
Our goal



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

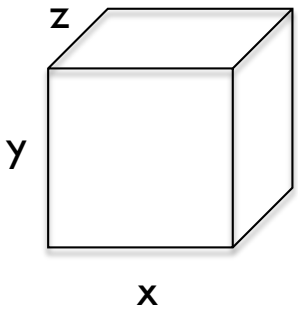
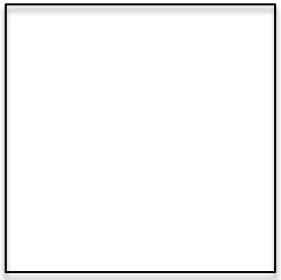


Image space:

All viewable objects within

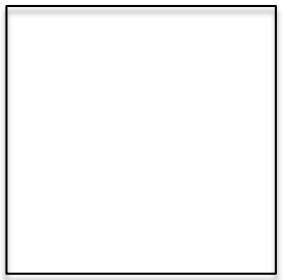
-1 <= x,y,z <= +1



Screen space:

All viewable objects within

-1 <= x, y <= +1

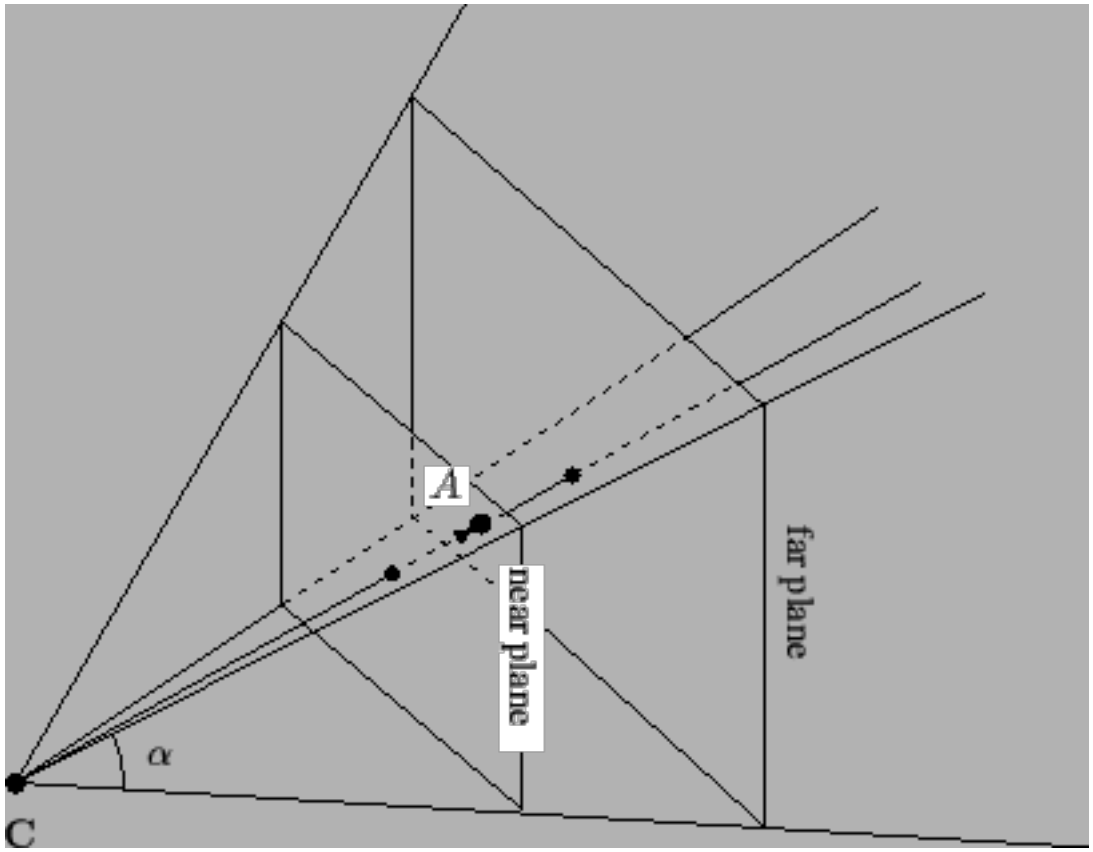


Device space:

All viewable objects within

0<=x<=width, 0 <=y<=height

How do we specify a camera?



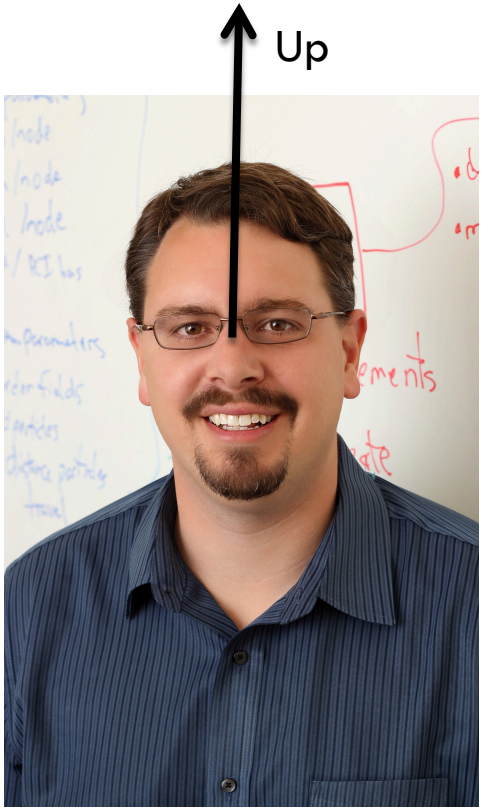
The "viewing pyramid" or "view frustum".

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];
};
```

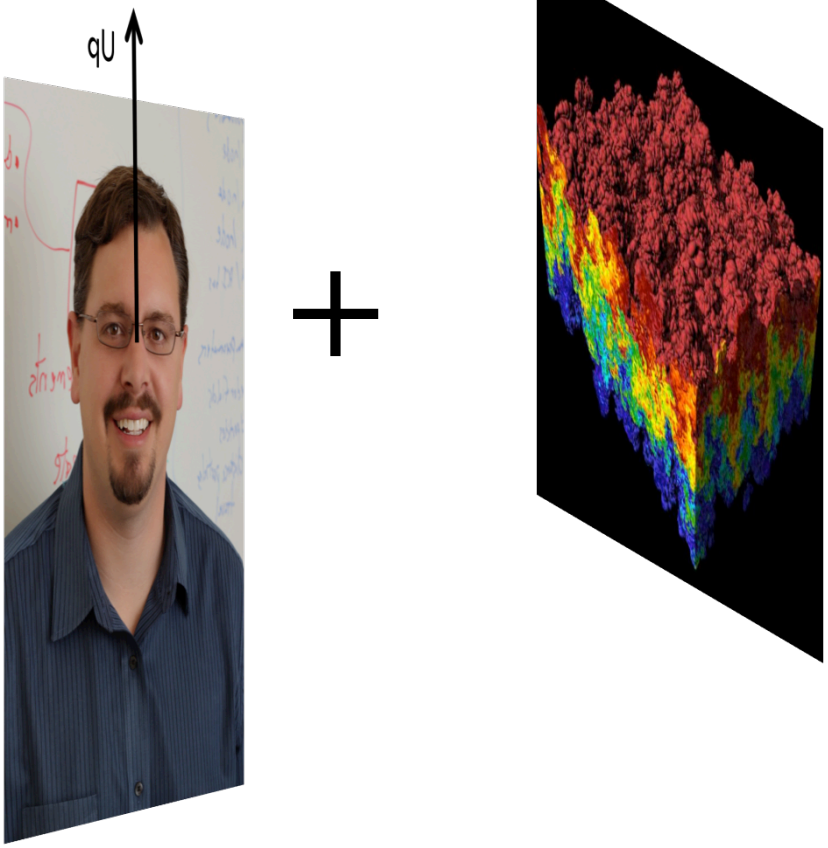
What is the up axis?

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



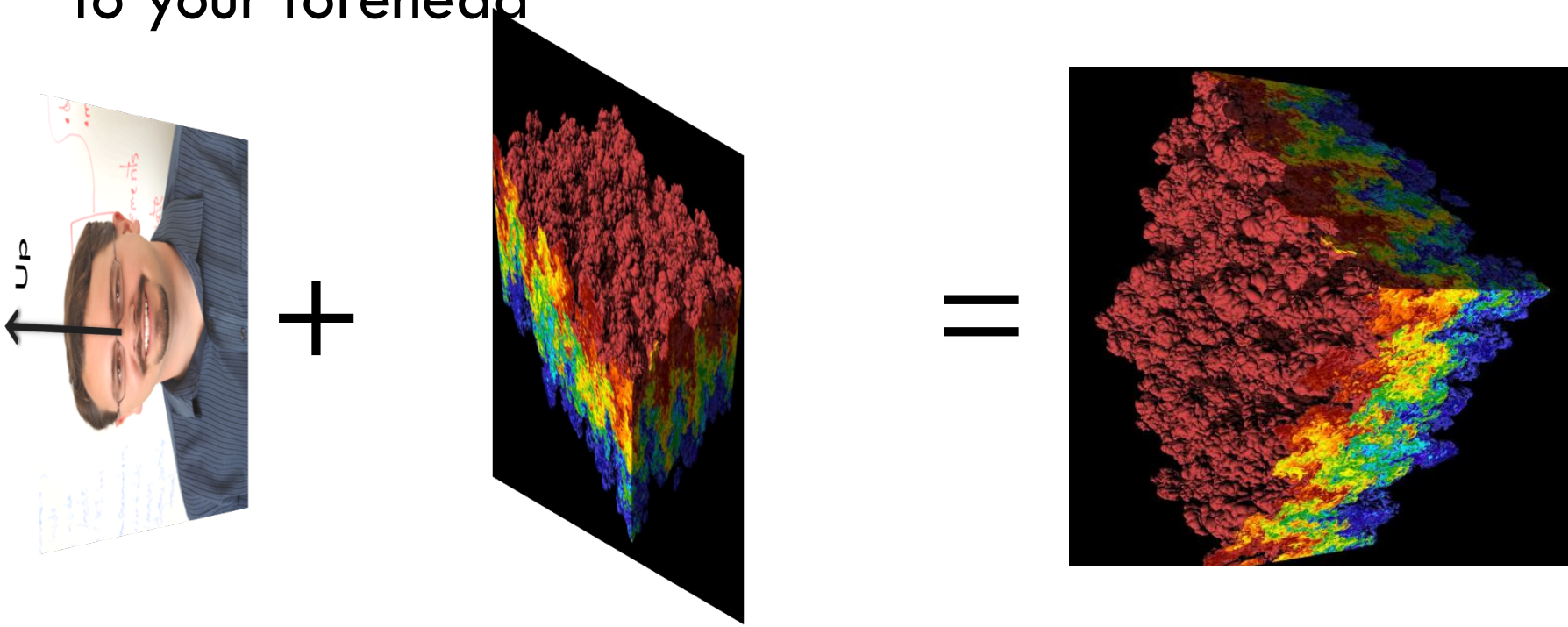
What is the up axis?

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



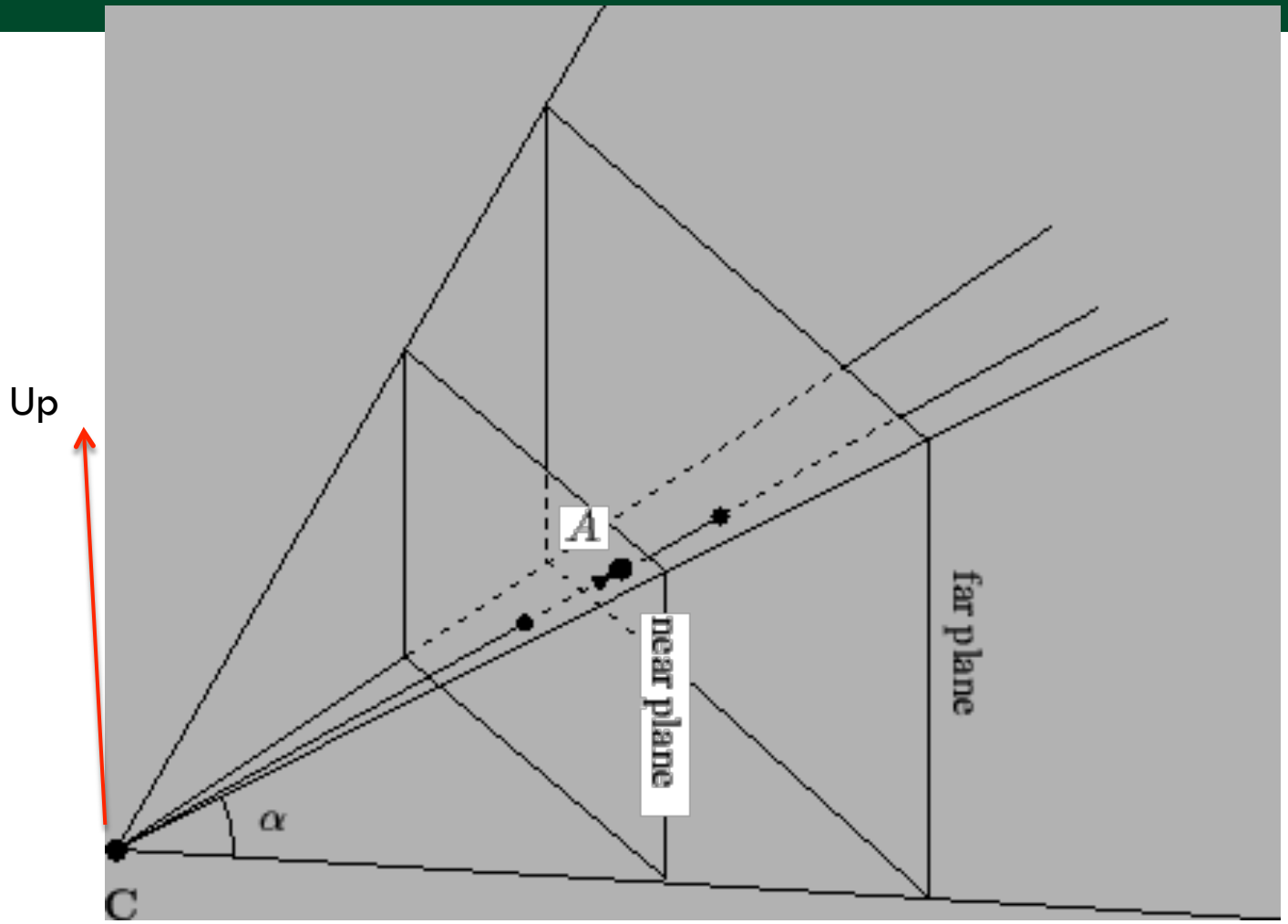
What is the up axis?

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

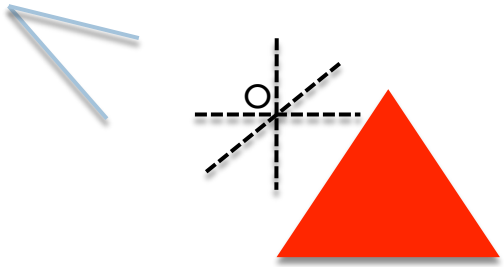


- (if you lie down while watching TV, the screen is sideways)

Image Space Diagram



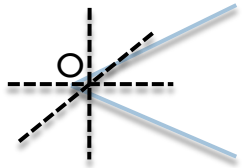
Our goal



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

View Transform

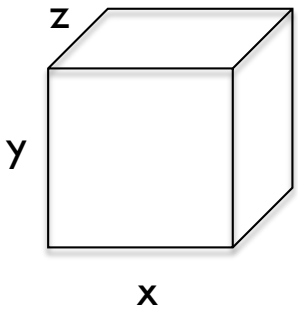
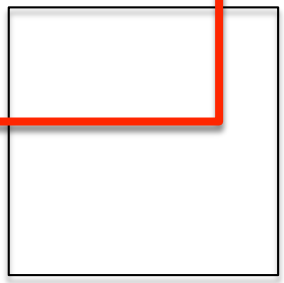


Image space:

All viewable objects within

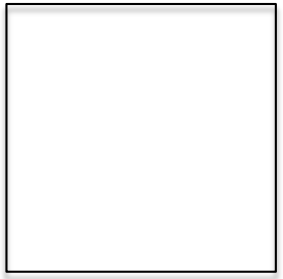
-1 <= x,y,z <= +1



Screen space:

All viewable objects within

-1 <= x, y <= +1



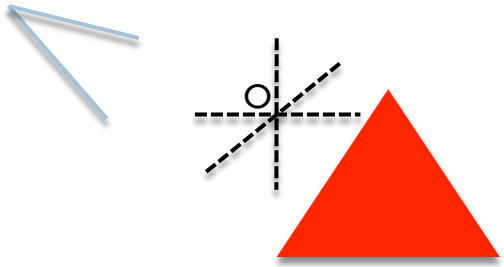
Device space:

All viewable objects within

0<=x<=width, 0

<=y<=height

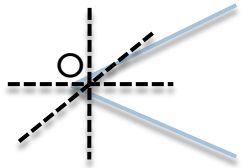
Our goal



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

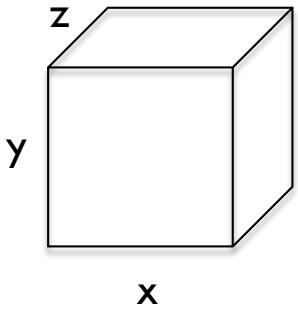
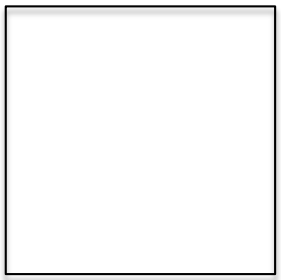


Image space:

All viewable objects within

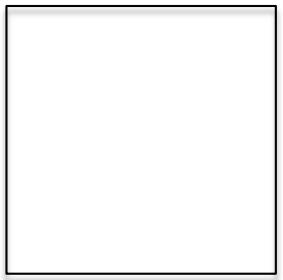
-1 <= x,y,z <= +1



Screen space:

All viewable objects within

-1 <= x, y <= +1



Device space:

All viewable objects within

0<=x<=width, 0

<=y<=height

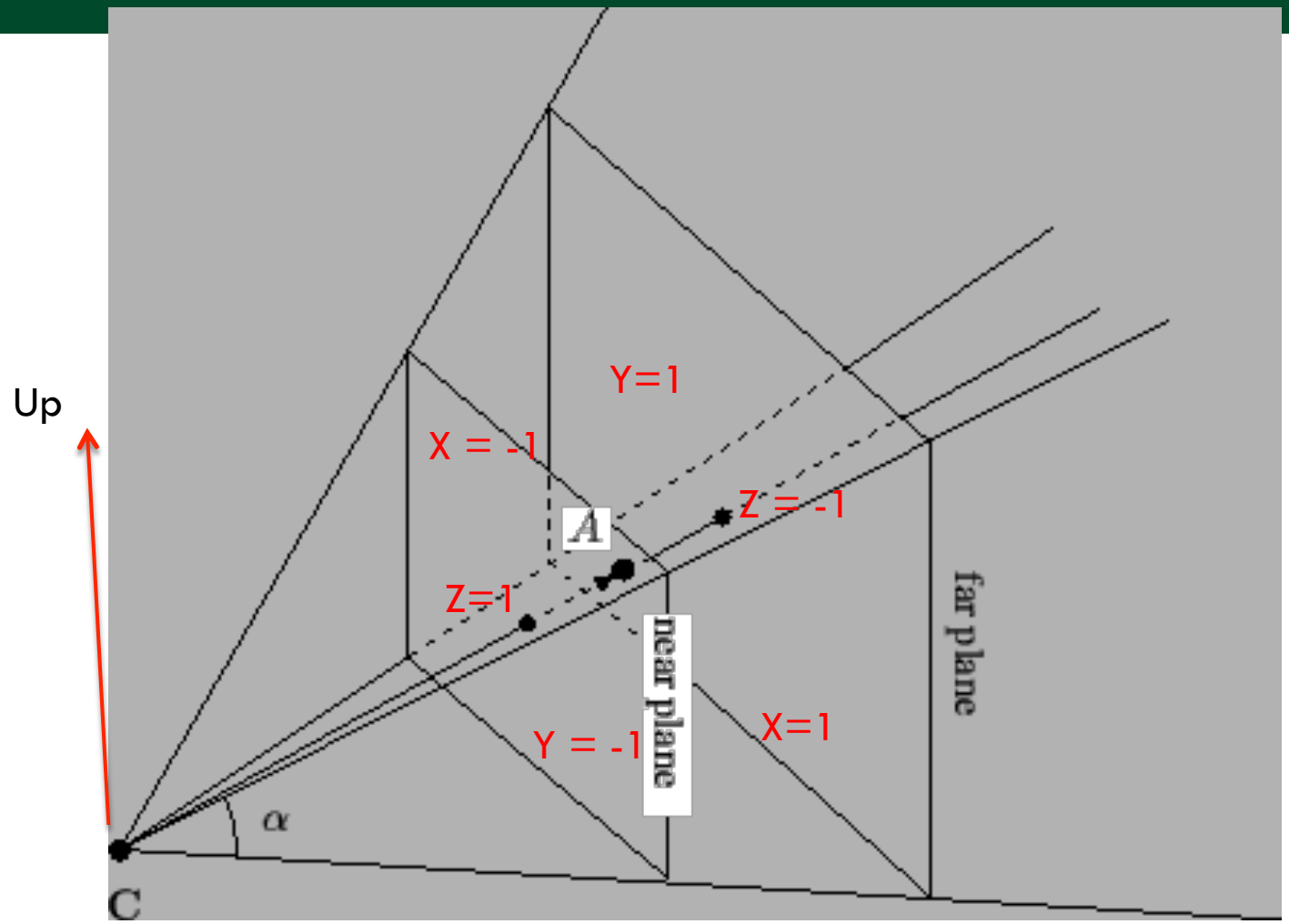
Image Space

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

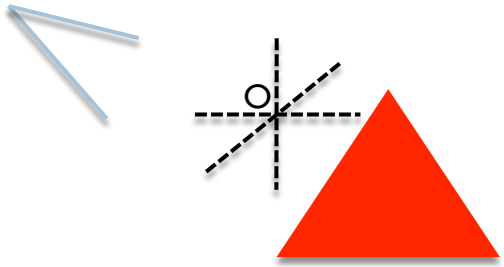
$$\{(x,y,z) : -1 \le x \le 1, -1 \le y \le 1, -1 \le z \le 1\}$$

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

Image Space Diagram



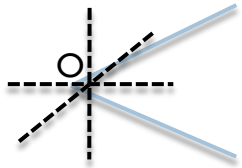
Our goal



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

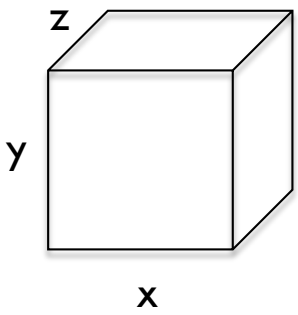
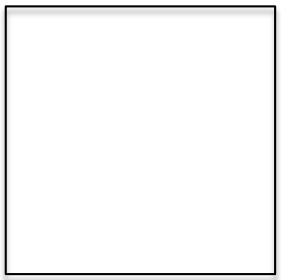


Image space:

All viewable objects within

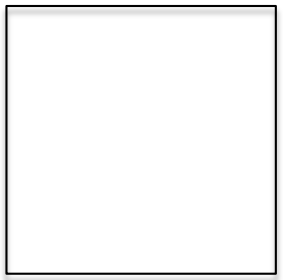
-1 <= x,y,z <= +1



Screen space:

All viewable objects within

-1 <= x, y <= +1



Device space:

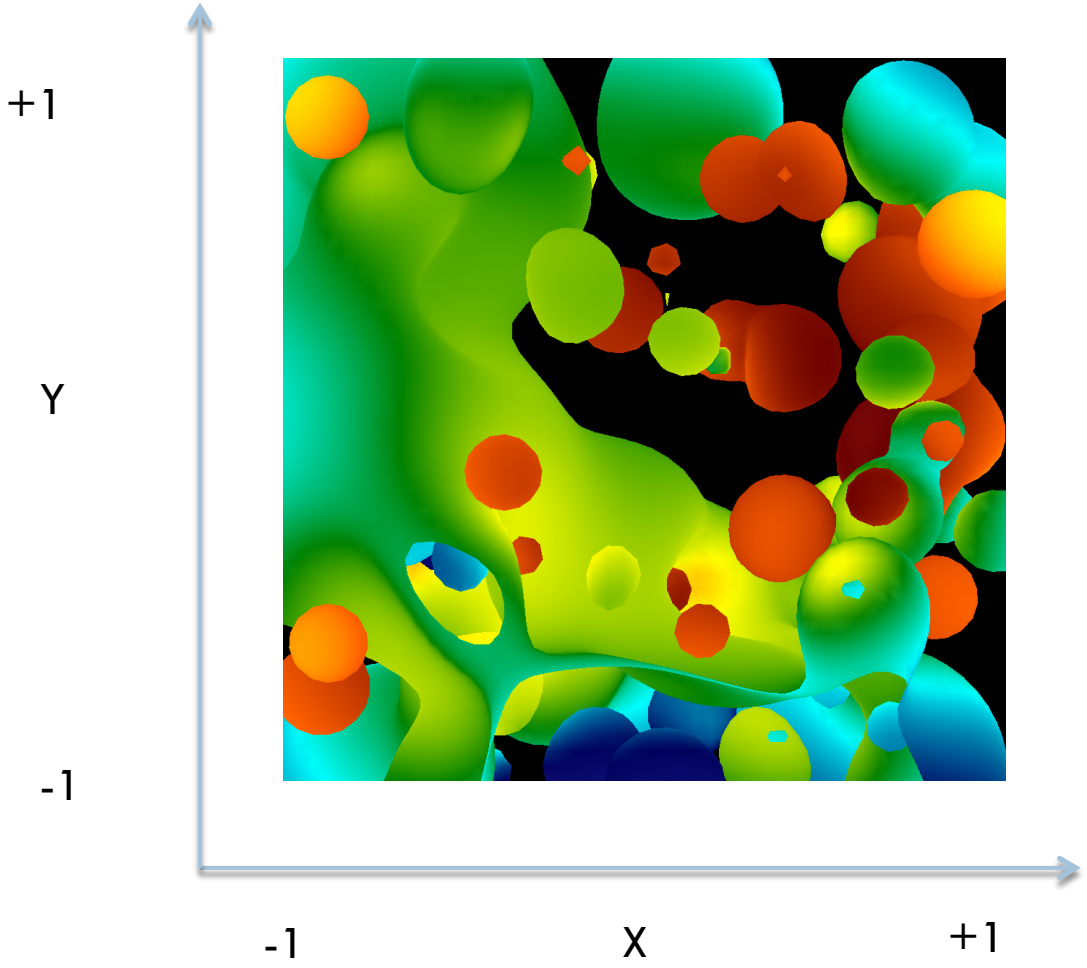
All viewable objects within

0<=x<=width, 0 <=y<=height

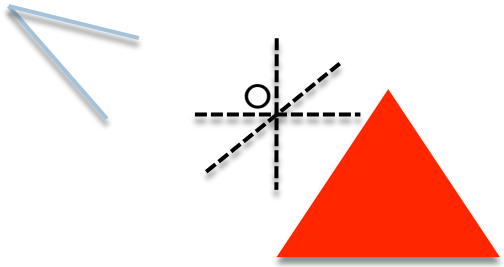
Screen Space

- 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



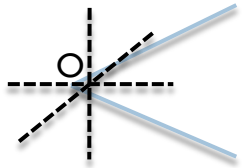
Our goal



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

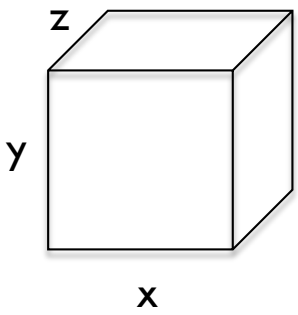
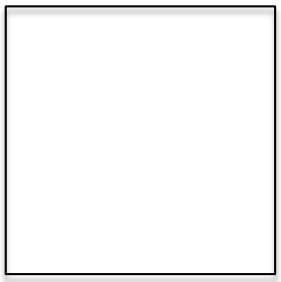


Image space:

All viewable objects within

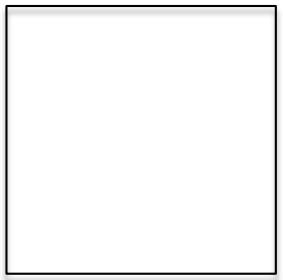
-1 <= x,y,z <= +1



Screen space:

All viewable objects within

-1 <= x, y <= +1



Device space:

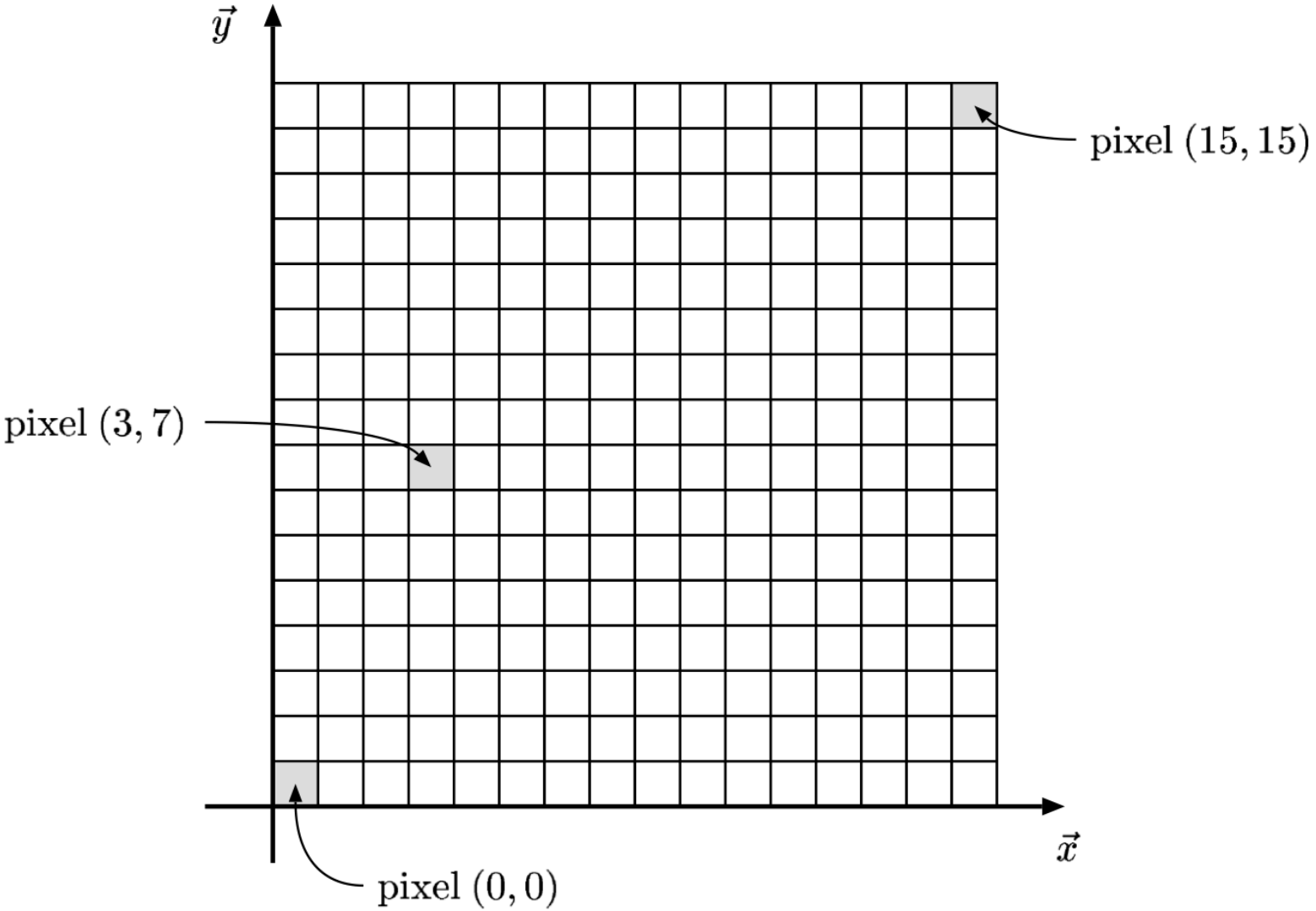
All viewable objects within

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 $-1 \leq z \leq 0$)

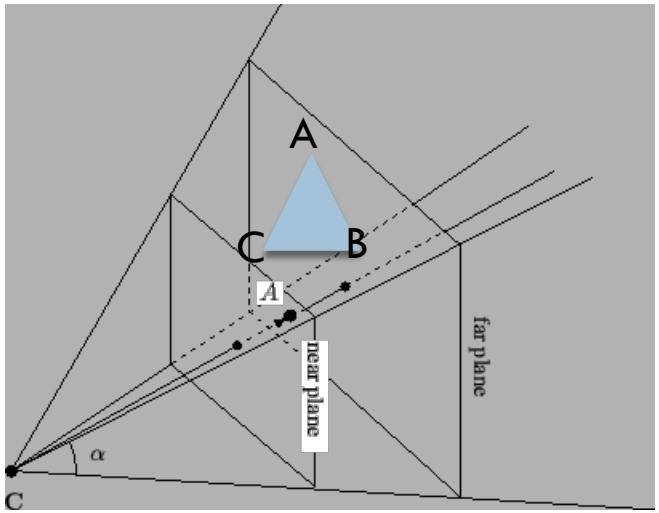
Start Part 2 of YouTube Video

- In Part 2:
  - Device Space Transform
  - More Math Primer

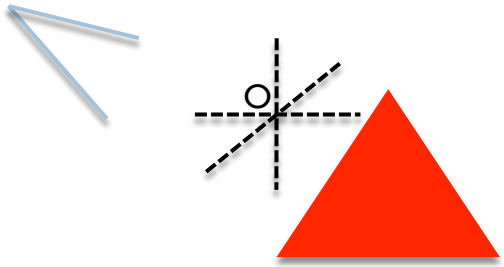
How do we transform?

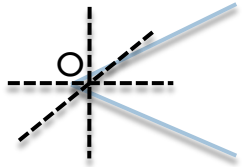
- For a camera C,
  - Calculate Camera Frame
  - From Camera Frame, calculate Camera Transform
  - Calculate View Transform
  - Calculate Device Transform
  - Compose 3 Matrices into 1 Matrix (M)
- For each triangle T, apply M to each vertex of T, then apply rasterization/ zbuffer

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



Easiest Transform





World space:

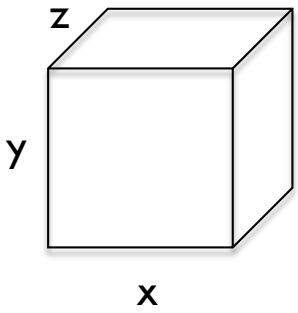
Triangles in native Cartesian coordinates

Camera located anywhere

Camera space:

Camera located at origin, looking down -Z

Triangle coordinates relative to camera frame



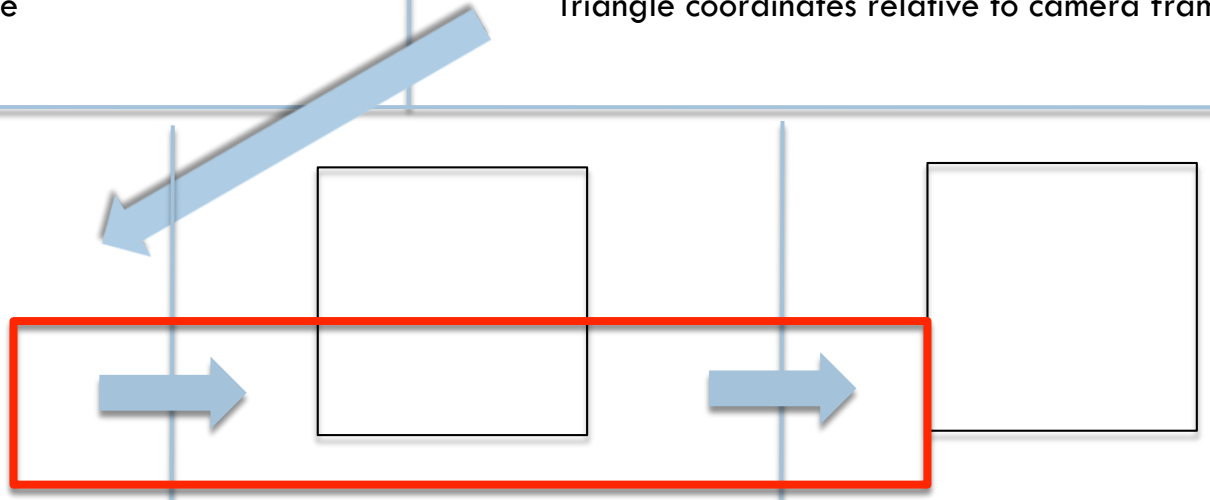


Image space:

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

$$(x \quad y \quad z \quad 1) \quad x \quad \begin{pmatrix} n/2 & 0 & 0 & 0 \\ 0 & m/2 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ n/2 & m/2 & 0 & 1 \end{pmatrix}$$

More Math Prep

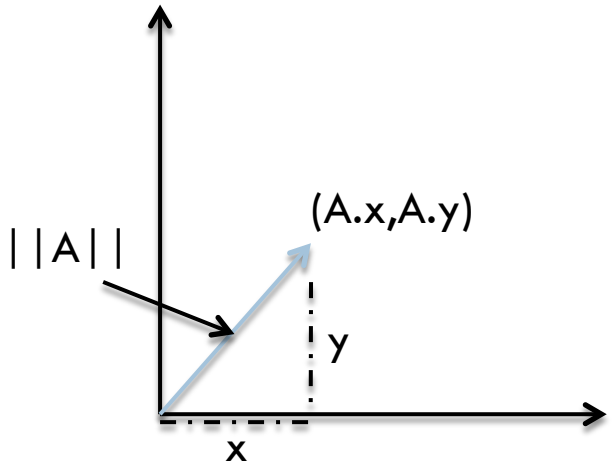
Note: Ken Joy's graphics notes are fantastic http://www.idav.ucdavis.edu/education/GraphicsNotes/homepage.html

What is the norm of a vector?

- The norm of a vector is its length
  - Denoted with $||\cdot||$
- For a vector A = (A.x, A.y),

$$||A|| = \text{sqrt}(A.x*A.x+A.y*A.y)$$

- Physical interpretation:



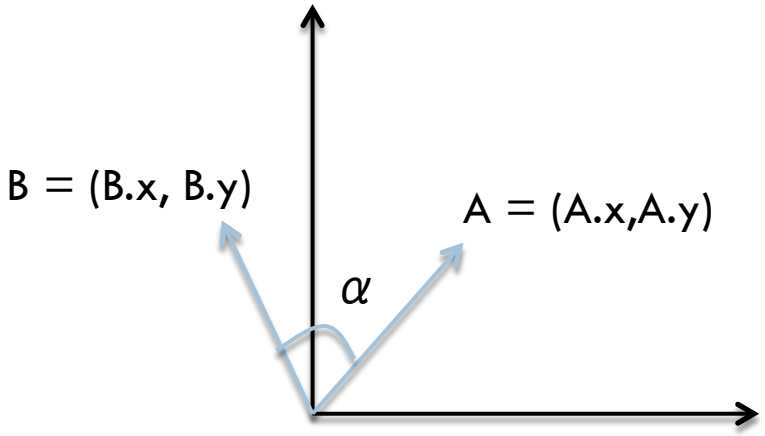
- For 3D, $||A|| = \text{sqrt}(A.x*A.x+A.y*A.y+A.z*A.z)$

What does it means for a vector to be normalized?

- The vector A is normalized if ||A|| = 1.
  - This is also called a unit vector.
- To obtain a normalized vector, take A/||A||
- Many of the operations we will discuss today will only work correctly with normalized vectors.
- Example: A=(3,4,0). Then:
  - ||A|| = 5
  - A/||A|| = (0.6, 0.8, 0)

What is a dot product?

- $A \cdot B = A.x*B.x + A.y*B.y$
  - (or $A.x*B.x + A.y*B.y + A.z*B.z$)
- Physical interpretation:
  - $A \cdot B = \cos(\alpha)*(||A||*||B||)$



What is the cross product?

- AxB = (A.y*B.z - A.z*B.y,
  B.x*A.z - A.x*B.z,
  A.x*B.y - A.y*B.x)

- What is the physical interpretation of a cross product?
  - Finds a vector perpendicular to both A and B.

Homogeneous Coordinates

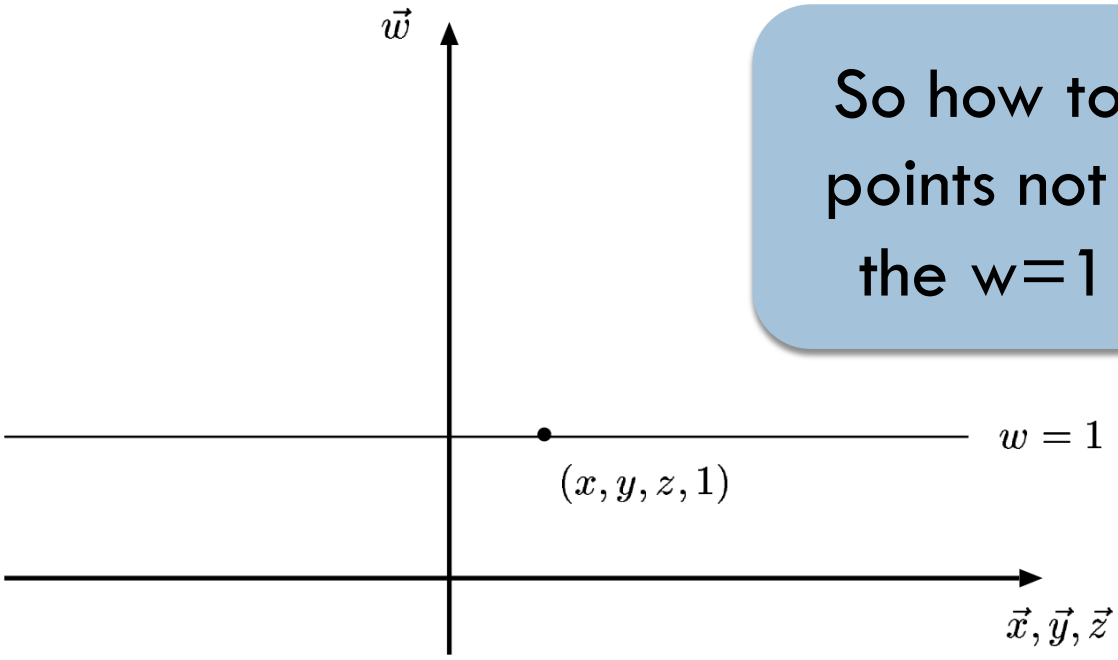
- Defined: a system of coordinates used in projective geometry, as Cartesian coordinates are used in Euclidean geometry
- Primary uses:
  - 4×4 matrices to represent general 3-dimensional transformations
  - it allows a simplified representation of mathematical functions – the rational form – in which 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?

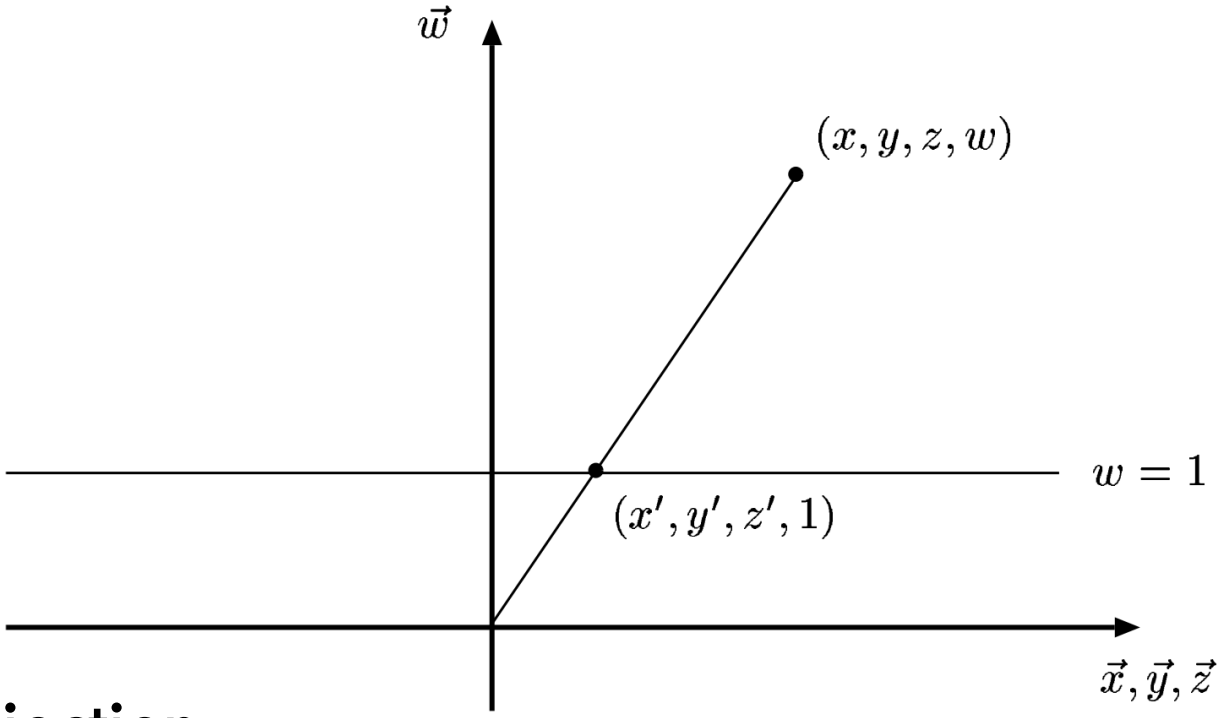
Our typical frame in the context of 4D points

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

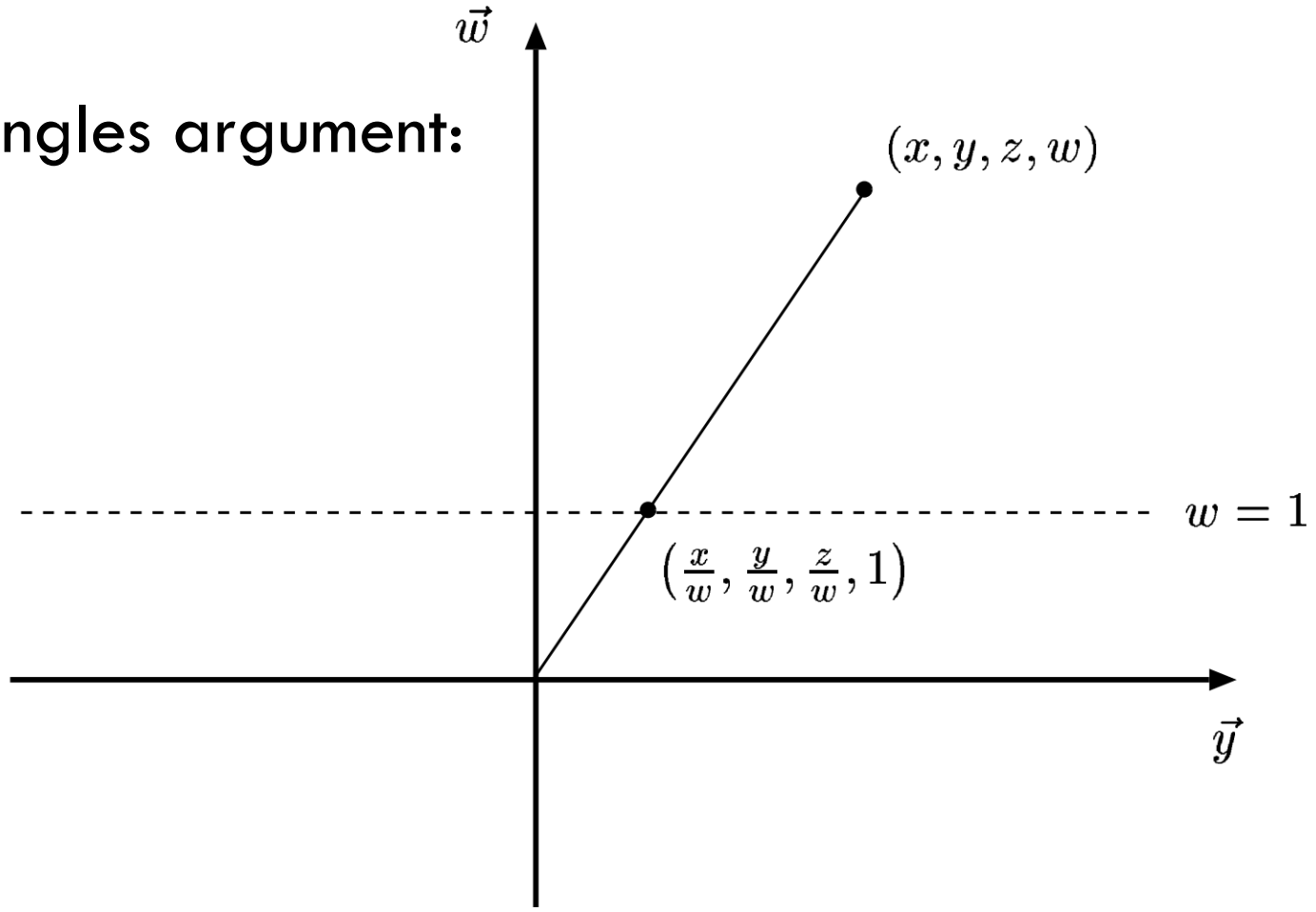
Projecting back to w==1 line



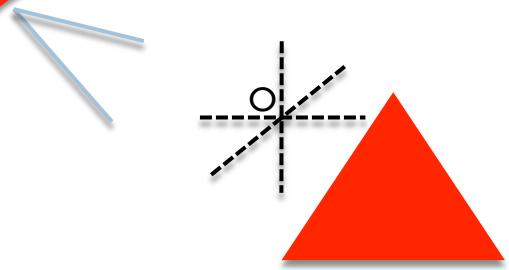
- Idea for projection:
  - Draw line from P to origin.
  - If Q is along that line (and Q.w == 1), then Q is a projection of P

So what is Q?

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



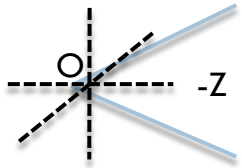
Our goal



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

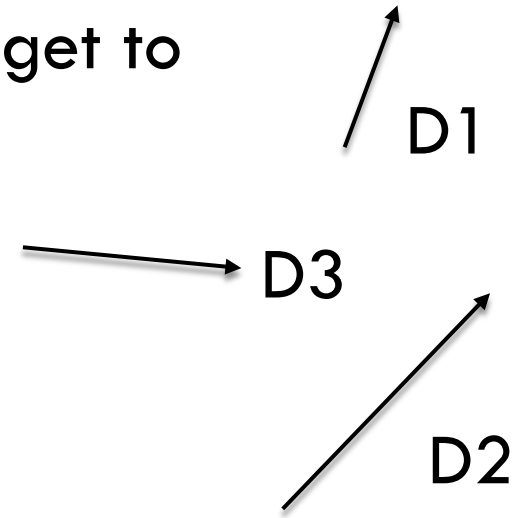
Basis pt 2
(more linear algebra-y this time)

- Camera frame must be a basis:
  - Spans space … can get any point through a linear combination of basis vectors
  - Every member must be linearly independent
    - → we didn't talk about this much on Thursday.
    - linearly independent means that no basis vector can be represented via others
    - Repeat slide (coming up) shows linearly *dependent* vectors

(REPEAT) Why unique?

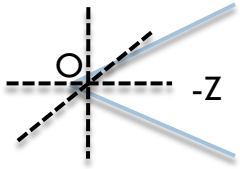
- Let (a, b, c) mean:
  - The number of steps ‘a’ in direction D1
  - The number of steps ‘b’ in direction D2
  - The number of steps ‘c’ in direction D3

- Then there is more than one way to get to some point X in S, i.e.,
  - (a1, b1, c1) = X and
  - (a2, b2, c2) = X



Camera frame construction

- Must choose (u,v,w,O)



Camera space:

Camera located at origin, looking down -Z

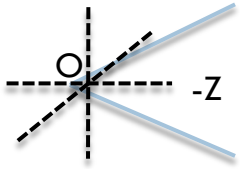
Triangle coordinates relative to camera frame

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

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

Camera frame construction

- Must choose (u,v,w,O)



Camera space:

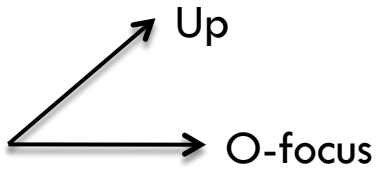
Camera located at origin, looking down -Z

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
- w = O-focus
- v = up
- u = up x (O-focus)

But wait … what if dot(v2,v3) != 0?



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

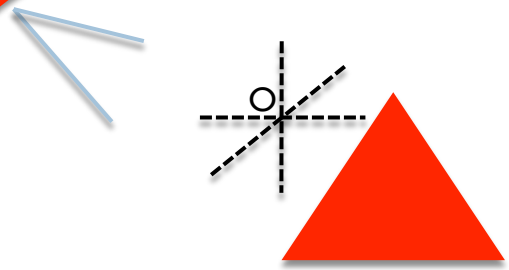
Camera frame summarized

- O = camera position
- u = Up x (O-focus)
- v = (O-focus) x u
- w = O-focus

- Important note:
  u, v, and w need to be normalized!

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

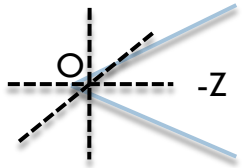
Our goal



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

This Will Come Up Later

- 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)]

The Two Frames of the Camera Transform

- Our two frames:

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

- Camera:
  - u = up x (O-focus)
  - v = (O-focus) x u
  - w = (O-focus)
  - O

The Two Frames of the Camera Transform

- Our two frames:

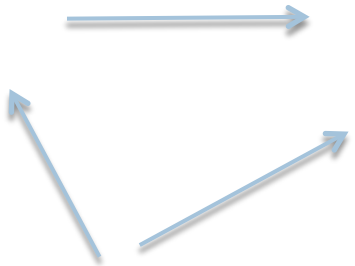
- Cartesian:
  - <1,0,0>
  - <0,1,0>
  - <0,0,1>
  - (0,0,0)

- Camera:
  - u = up x (O-focus)
  - v = (O-focus) x u
  - w = (O-focus)
  - O

The “Camera Frame” is a Frame, so we can express any Cartesian vector as a combination of u, v, w.

Converting From Cartesian Frame To Camera Frame

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



Putting Our Equations Into Matrix Form

- <1,0,0> = e1,1 * u + e1,2 * v + e1,3 * w
- <0,1,0> = e2,1 * u + e2,2 * v + e2,3 * w
- <0,0,1> = e3,1 * u + e3,2 * v + e3,3 * w
- (0,0,0) = e4,1 * u + e4,2 * v + e4,3 * w + O
- →
- [<1,0,0>] [e1,1 e1,2 e1,3 0] [u]
- [<0,1,0>] [e2,1 e2,2 e2,3 0] [v]
- [<0,0,1>] = [e3,1 e3,2 e3,3 0] [w]
- (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] [u]

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

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

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

Here Comes The Trick…

But:

$$\begin{matrix}[\langle 1,0,0\rangle] \\ [x\ y\ z\ 1][\langle 0,1,0\rangle] \\ [\langle 0,0,1\rangle] \\ [(0,0,0)]\end{matrix} = \begin{matrix}[e1,1 & e1,2 & e1,3 & 0] \\ [x\ y\ z\ 1]\ [e2,1 & e2,2 & e2,3 & 0] \\ [e3,1 & e3,2 & e3,3 & 0] \\ [e4,1 & e4,2 & e4,3 & 1]\end{matrix} \begin{matrix}[u] \\ [v] \\ [w] \\ [O]\end{matrix}$$

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

$$\begin{bmatrix} e_{1,1} & e_{1,2} & e_{1,3} & 0 \\ e_{2,1} & e_{2,2} & e_{2,3} & 0 \\ e_{3,1} & e_{3,2} & e_{3,3} & 0 \\ e_{4,1} & e_{4,2} & e_{4,3} & 1 \end{bmatrix} = \begin{bmatrix} u.x & v.x & w.x & 0 \\ u.y & v.y & w.y & 0 \\ u.z & v.z & w.z & 0 \\ u \cdot t & v \cdot t & w \cdot t & 1 \end{bmatrix}$$

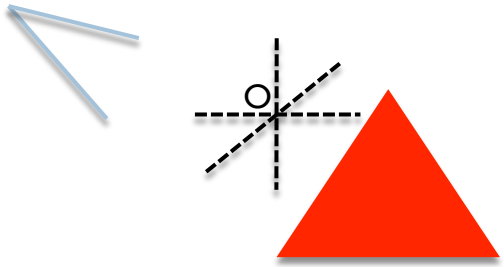
Where t = (0,0,0)-O

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

Want to derive?:

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

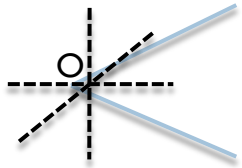
Our goal



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

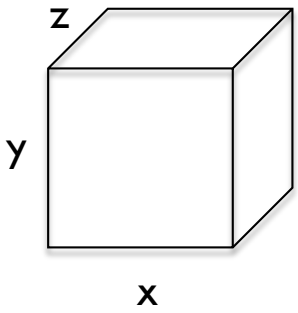
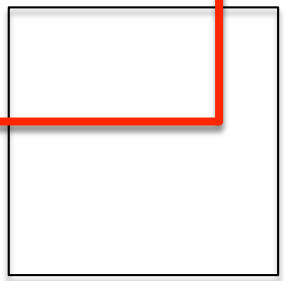


Image space:

All viewable objects within

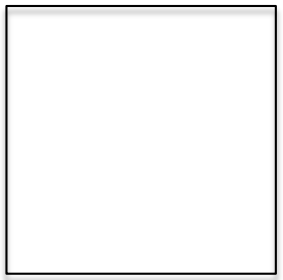
-1 <= x,y,z <= +1



Screen space:

All viewable objects within

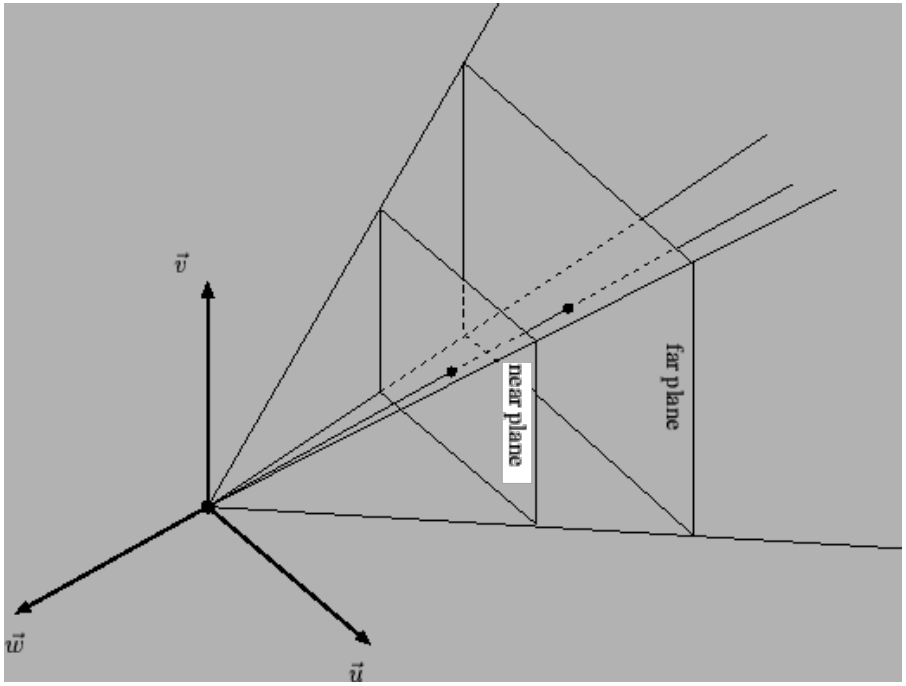
-1 <= x, y <= +1

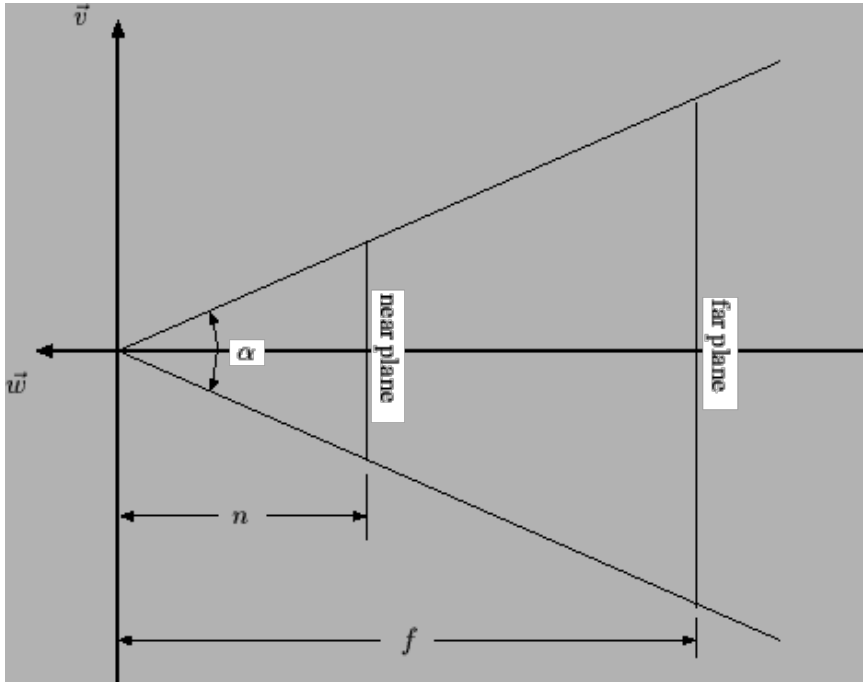


Device space:

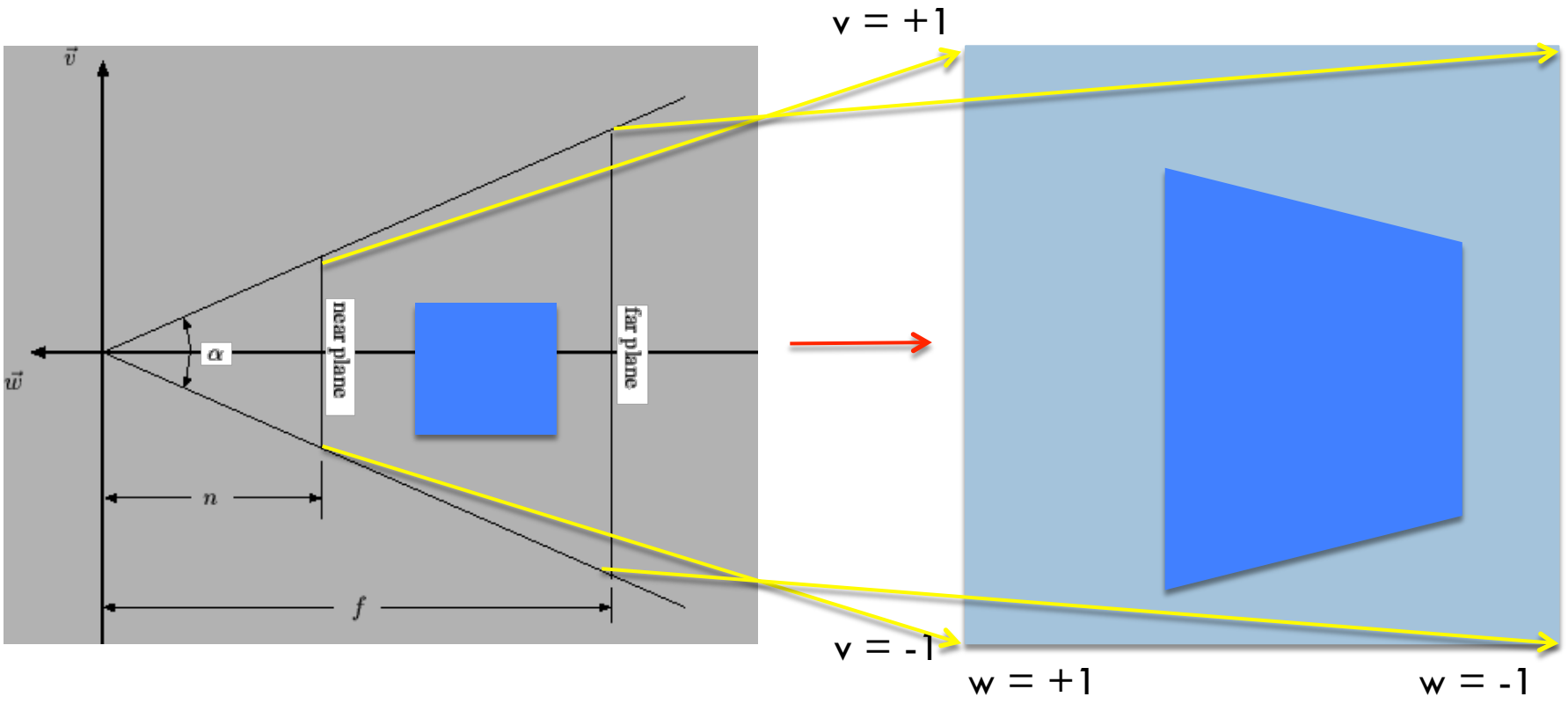
All viewable objects within

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

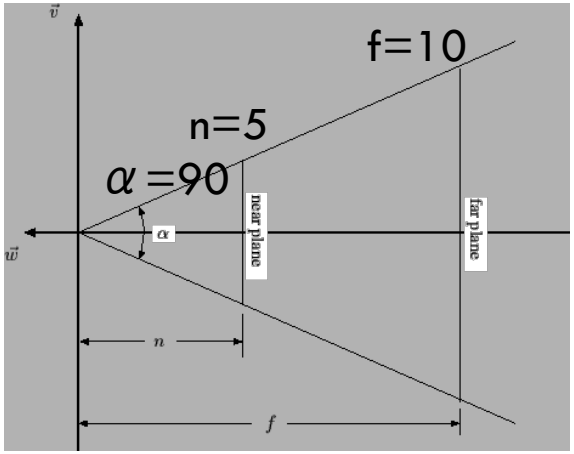
- Input parameters: (α , n, f)
- Transforms view frustum to image space cube
  - View frustum: bounded by viewing pyramid and planes z=-n and z=-f
  - Image space cube: -1 <= u,v,w <= 1

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

  - Cotangent = 1/tangent

Let's do an example

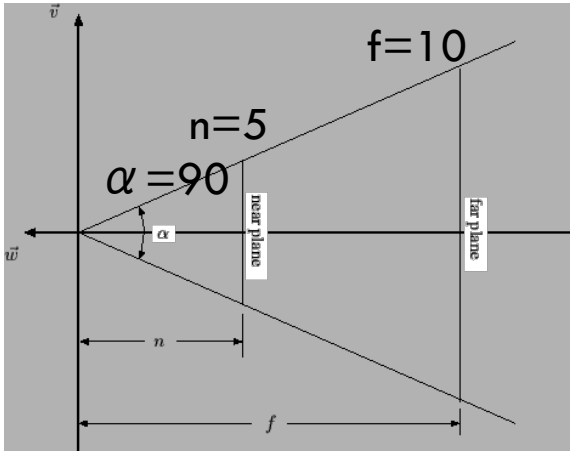
- Input parameters: (α , n, f) = (90, 5, 10)



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

Let's do an example

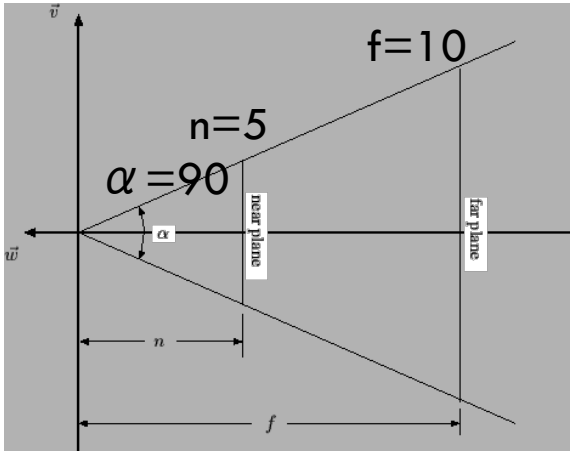
- Input parameters: (α , n, f) = (90, 5, 10)



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

Let's do an example

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

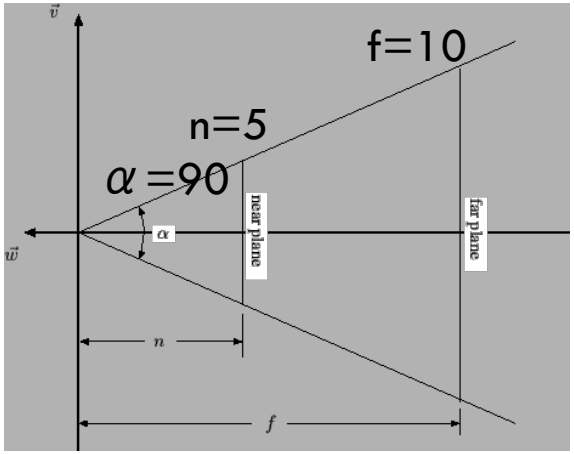


$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 3 & -1 \\ 0 & 0 & 20 & 0 \end{bmatrix}$$

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)

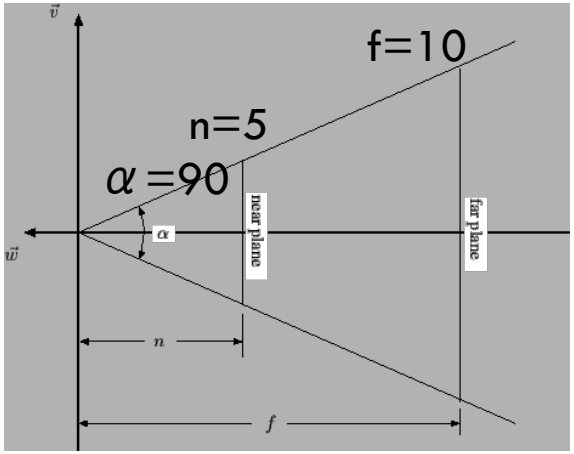


$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 3 & -1 \\ 0 & 0 & 20 & 0 \end{bmatrix}$$

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)

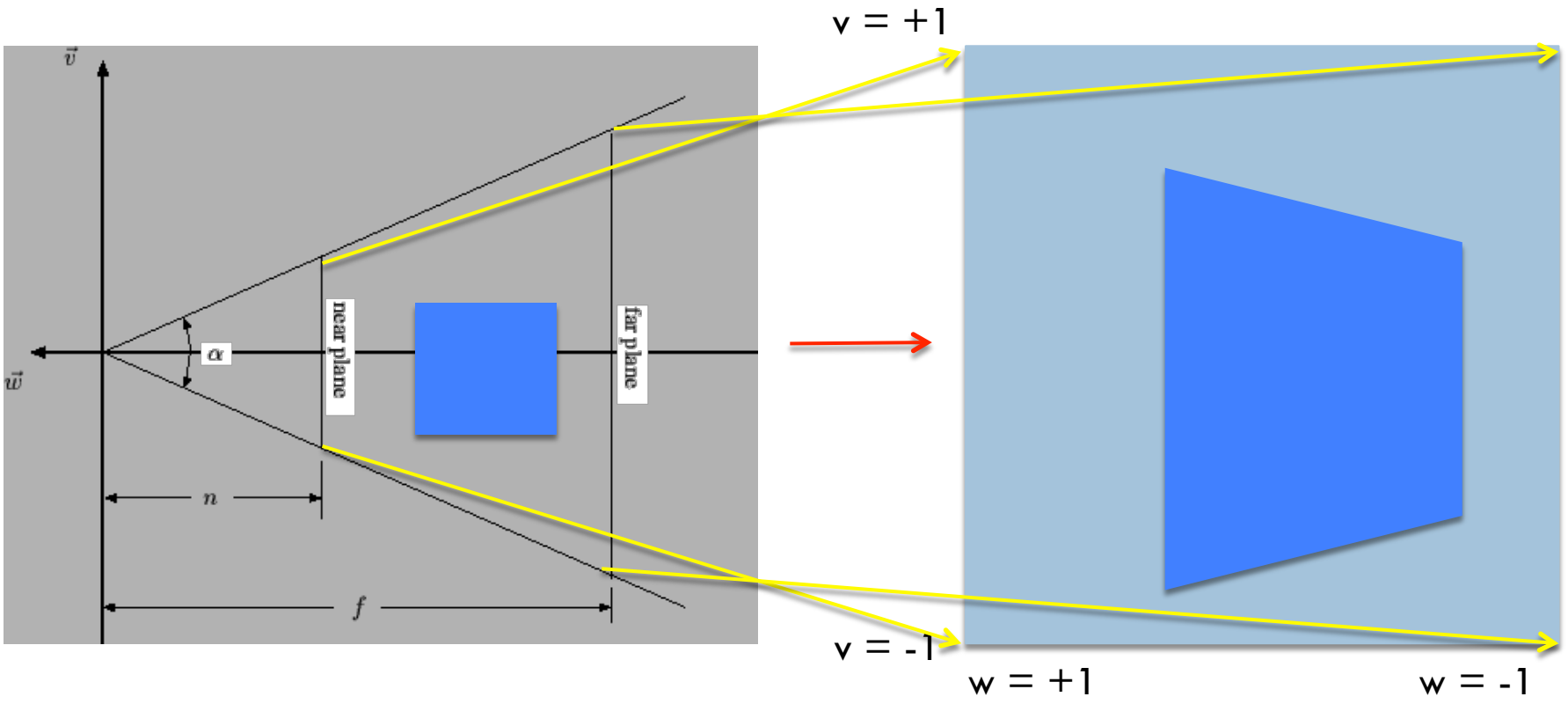


$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 3 & -1 \\ 0 & 0 & 20 & 0 \end{bmatrix}$$

More points:

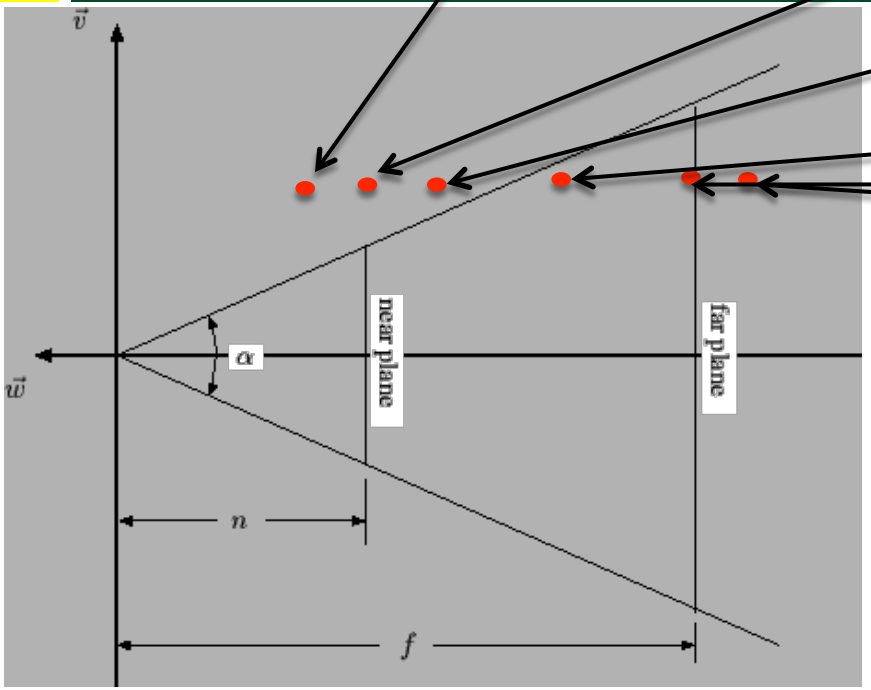
(0,7,-4,1) = (0,7,8,4) = (0, 1.75, 2)
(0,7,-5,1) = (0,7,5,5) = (0, 1.4, 1)
(0,7,-6,1) = (0,7,2,6) = (0, 1.16, 0.33)
(0,7,-8,1) = (0,7,-4,8) = (0, 0.88, -0.5)
(0,7,-10,1) = (0,7,-10,10) = (0, 0.7, -1)
(0,7,-11,1) = (0,7,-13,11) = (0, .63, -1.18)

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, 2)

(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) = (0,7,8,4) = (0, 1.75, 2)

(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)

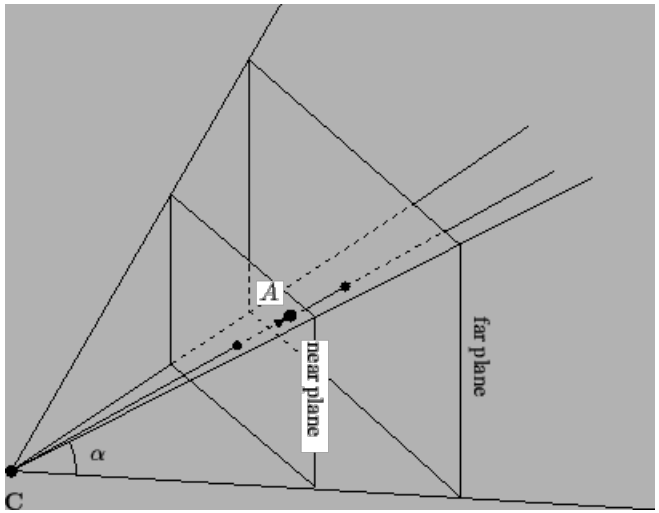
Note there is a non-linear relationship in W ("Z").

Putting It All Together

How do we transform?

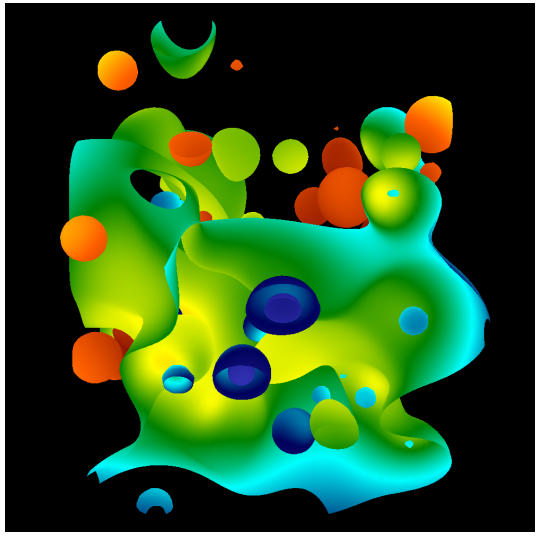
- For a camera C,
  - Calculate Camera Frame
  - From Camera Frame, calculate Camera Transform
  - Calculate View Transform
  - Calculate Device Transform
  - Compose 3 Matrices into 1 Matrix (M)
- For each triangle T, apply M to each vertex of T, then apply rasterization/ zbuffer

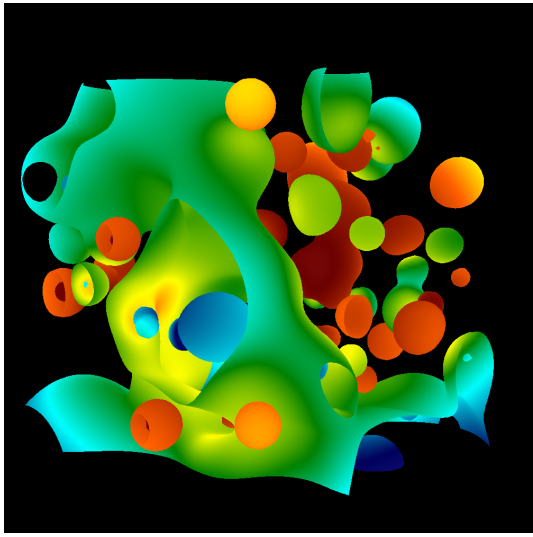
```
class Camera
{
  public:
    double          near, far;
    double          angle;
    double          position[3];
    double          focus[3];
    double          up[3];
};
```

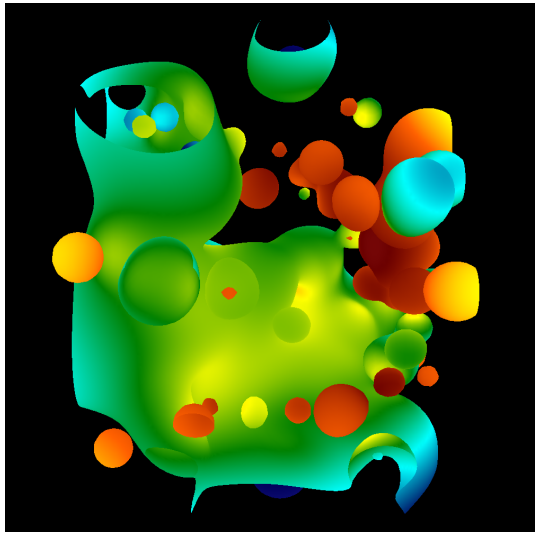


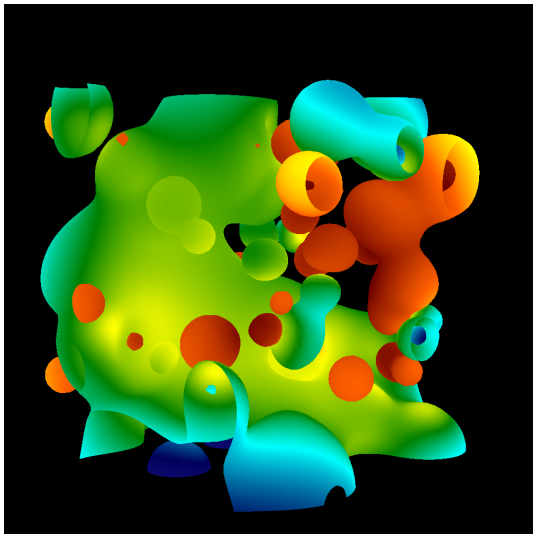
Project 1E

Project #1E (6%), Due Weds Feb 6th









- Goal: add arbitrary camera positions
- Extend your project1D code
- New: proj1e_geometry.vtk available on web (9MB), “reader1e.cxx”.
- New: Matrix.cxx, Camera.cxx
- No Cmake, project1E.cxx

Project #1E, 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 #1E, 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 #1E, 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
    - Then you should wait for 1F. Then we will have shading too.

Project #1E, game plan

```
vector<Triangle> t = GetTriangles();
AllocateScreen();
for (int i = 0 ; i < 4 ; i++)
{   int f = 250*i;
    InitializeScreen();
    Camera c = GetCamera(f, 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 #1E 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.

Project #1E, 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 Feb 19th

- Goal: add shading, movie