CMSC427 Transformations II: Viewing

Credit: some slides from Dr. Zwicker

What next?

- GIVEN THE TOOLS OF …
- The standard rigid and affine transformations
- Their representation with matrices and homogeneous coordinates

- WHAT CAN WE DO WITH THESE TOOLS?
- Modeling – how can we define transformations we want?
- Viewing – how can we use these tools to render objects like polygonal meshes?

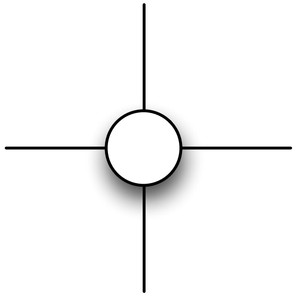
Review: modeling with transformations

- Object space

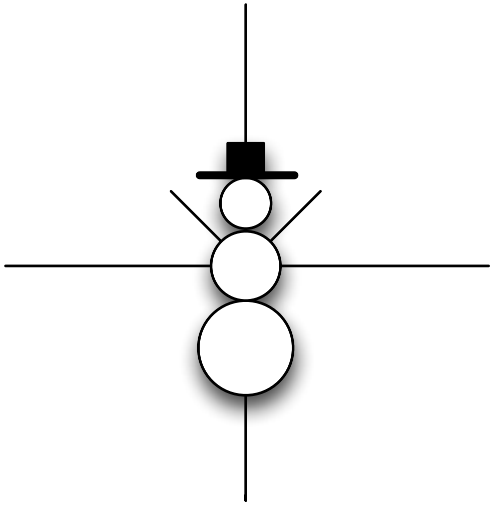
- World space

- Create scene by transforming objects from object to world

- Object coordinate space



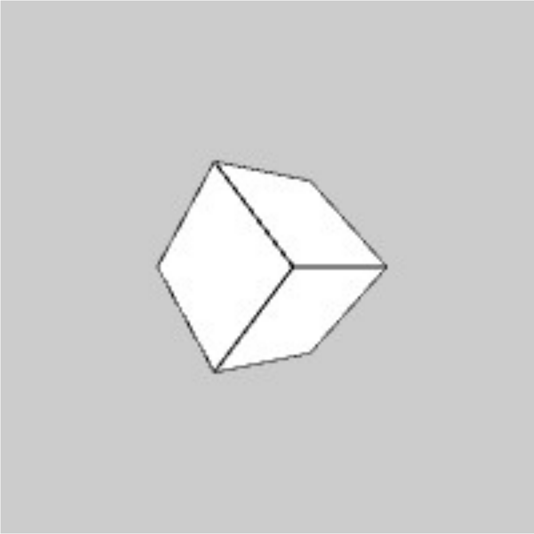
- World coordinate space



Modeling

- Shape object
  - Size, reshape
- Place object
  - Position and orientation

```
// Processing example

size(200,200,P3D);

translate(width/2,height/2,0);

rotateY(PI/4);
rotateX(PI/4);

box(50);
```



Transformations in Processing (OpenGL 1.0 and 2.0 style)

- **Transforms**
- applyMatrix()
- popMatrix()
- printMatrix()
- pushMatrix()
- resetMatrix()
- rotate()
- rotateX()
- rotateY()
- rotateZ()
- scale()
- shearX()
- shearY()
- translate()

- Camera
- beginCamera()
- camera()
- endCamera()
- frustum()
- ortho()
- perspective()

- Tracing
- printMatrix()
- printCamera()
- printProjection()

- Routines *not* in OpenGL 3.0/4.0 but in many utility libraries

Observation 1: Parametric transformations

- Instead of …

```
for (int t=0; t < 2*PI; t += 0.1) {
  float x = cx+ r*cos(t);
  float y = cy+ r*sin(t);
  ellipse(x,y,10,10);
}
```

- Use …

```
for (int t=0; t < 2*PI; t += 0.1) {
  float x = cx+ r*cos(t);
  float y = cy+ r*sin(t);
  translate(x,y); // Or more …
  complexObject();
}
```

- Why? Simplify the code for the object, enable complex transformations.
- Mesh not need understand transformations

Observation 1: Problem!

- Transforms can accumulate

- translate(5,5);
- translate(10,50);
- Result: (15,15)

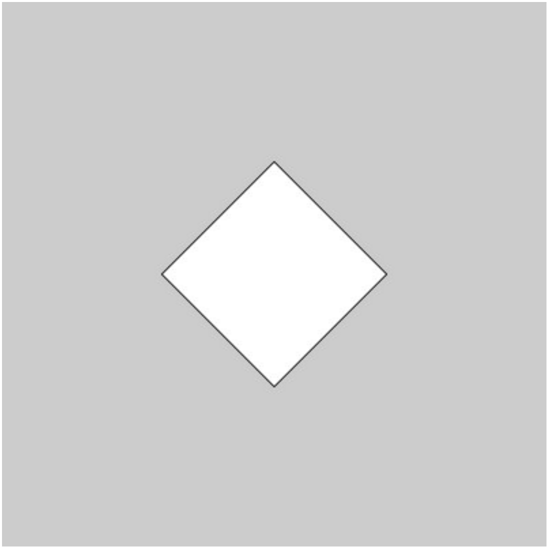
- pushMatrix() preserves existing matrix, popMatrix() restores

```
for (int t=0; t < 2*PI; t += 0.1)
{
   float x = cx+ r*cos(t);
   float y = cy+ r*sin(t);
   pushMatrix();
     translate(x,y); // Or more …
   popMatrix();
   complexObject();
}
```

Observation II: Experimenting with 3D transforms

- Get to know transformations by experimentation
- Processing good for this
- Try
  - translate
  - scale
  - shearX, shearY
  - rotateX,rotateY,rotateZ
- Understand
  - Direction of x, y, z
  - Direction of rotations

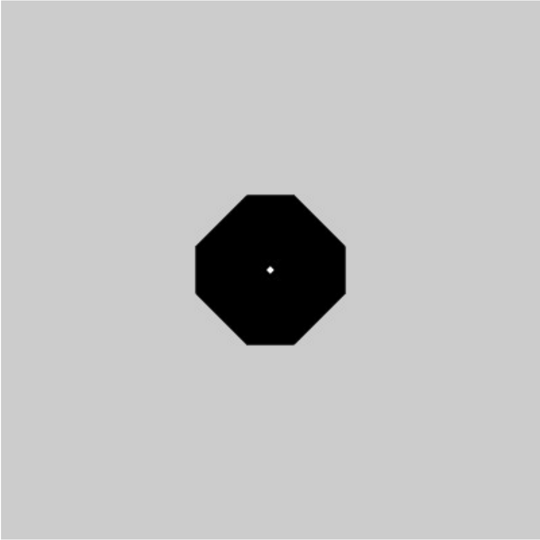
```
// Basic code

size(400,400,P3D);
translate(width/2,height/2,0);
rotateZ(PI/4);
box(100);
```



Observation II: Problem (Processing scales outline stroke)

- More elegant code

- scale(50);
- box(1);



- But we get this picture
- ?????????
- strokeWeight is scaled

- So … box(50) it is.

Observation 3: Tracing matrix

- Can print current transformation matrix to debug

```
size(400,400,P3D);
translate(width/2,height/2,0);
rotateZ(PI/4);
printMatrix();
box(50);
```

```
000.7071 -000.7071  000.0000  000.0000
000.7071  000.7071  000.0000  000.0000
000.0000  000.0000  001.0000 -346.4102
000.0000  000.0000  000.0000  001.0000
```

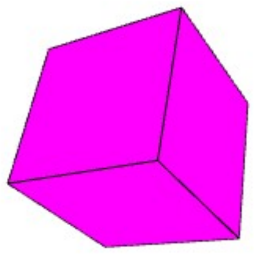
- Why -346?
  Where camera is.
  Viewing from
  +346 in Z

Experiments!

- Translate with positive and negative X,Y,Z
  - Figure out the coordinate system
- Rotate around X, Y, Z in different orders
- Scale non-uniformly in X,Y,Z
- Change order of scale, rotate, translate
- Try a shear

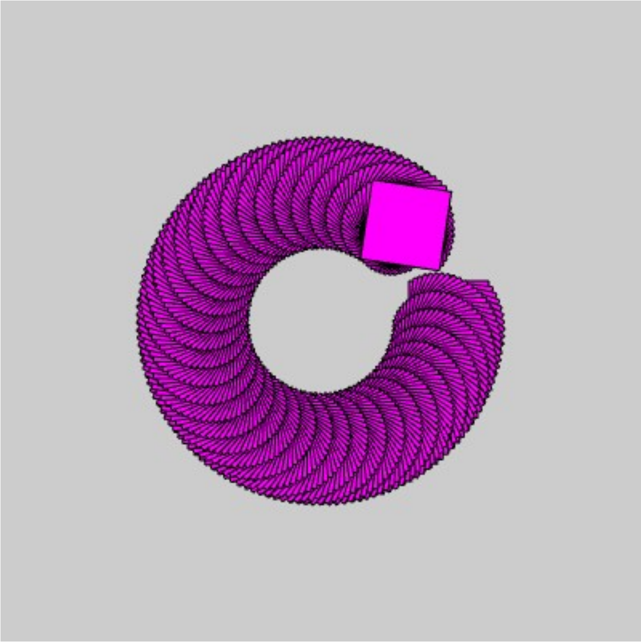
Transformations and animations

```
float theta = 0;

void setup(){
 size(400,400,P3D);
 fill(255,0,255);
}

void draw(){
 background(255);
 translate(width/2,height/2,0);
 rotateZ(theta);
 rotateX(theta);
 rotateZ(theta);
 box(100);
 theta += 0.01;
}
```



Orbiting box?

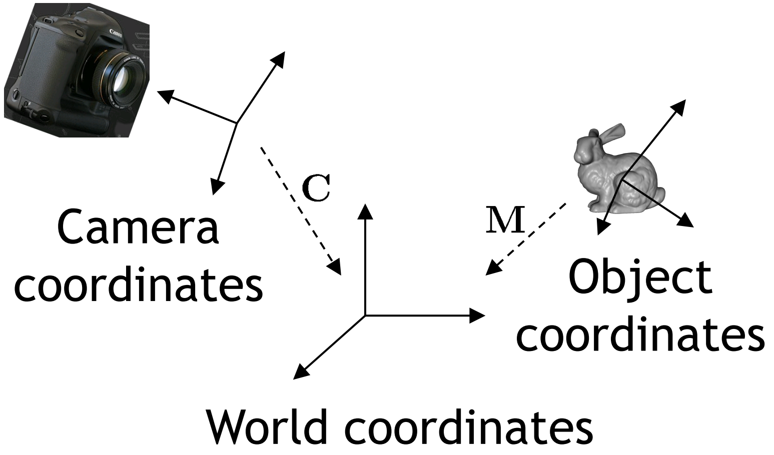
- How animate box rotating around its center as it's orbiting the center of the sketch?



Need to know

- Where is the camera?

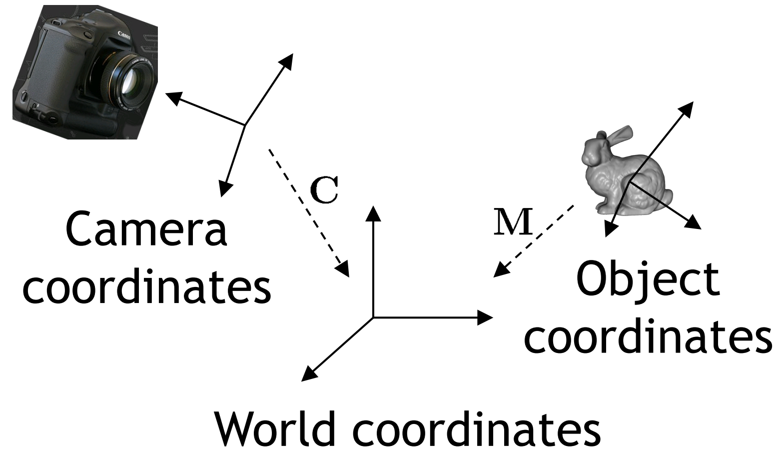
- What lens does it have?



Viewing transformations: the virtual camera

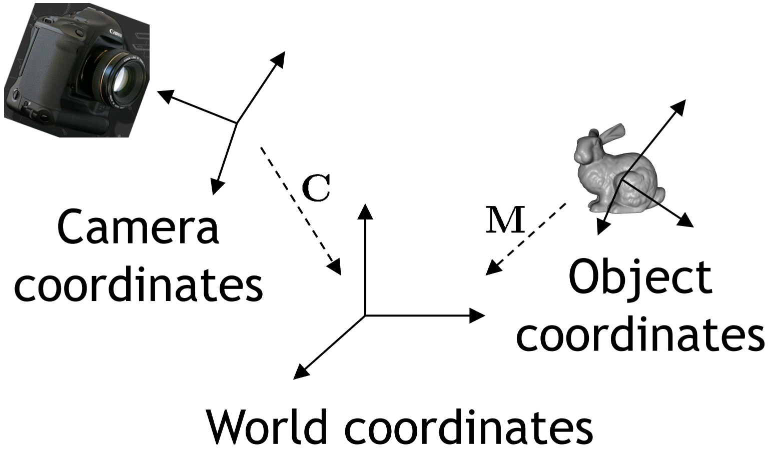
Need to know

- Where is the camera?
  - CAMERA TRANSFORM
- What lens does it have?
  - PROJECTIVE TRANSFORM



Virtual camera routines in Processing

- Camera (where)
- [beginCamera()]
- [camera()]
- [endCamera()]



- Projective (length of lens)
- [frustum()]
- [ortho()]
- [perspective()]

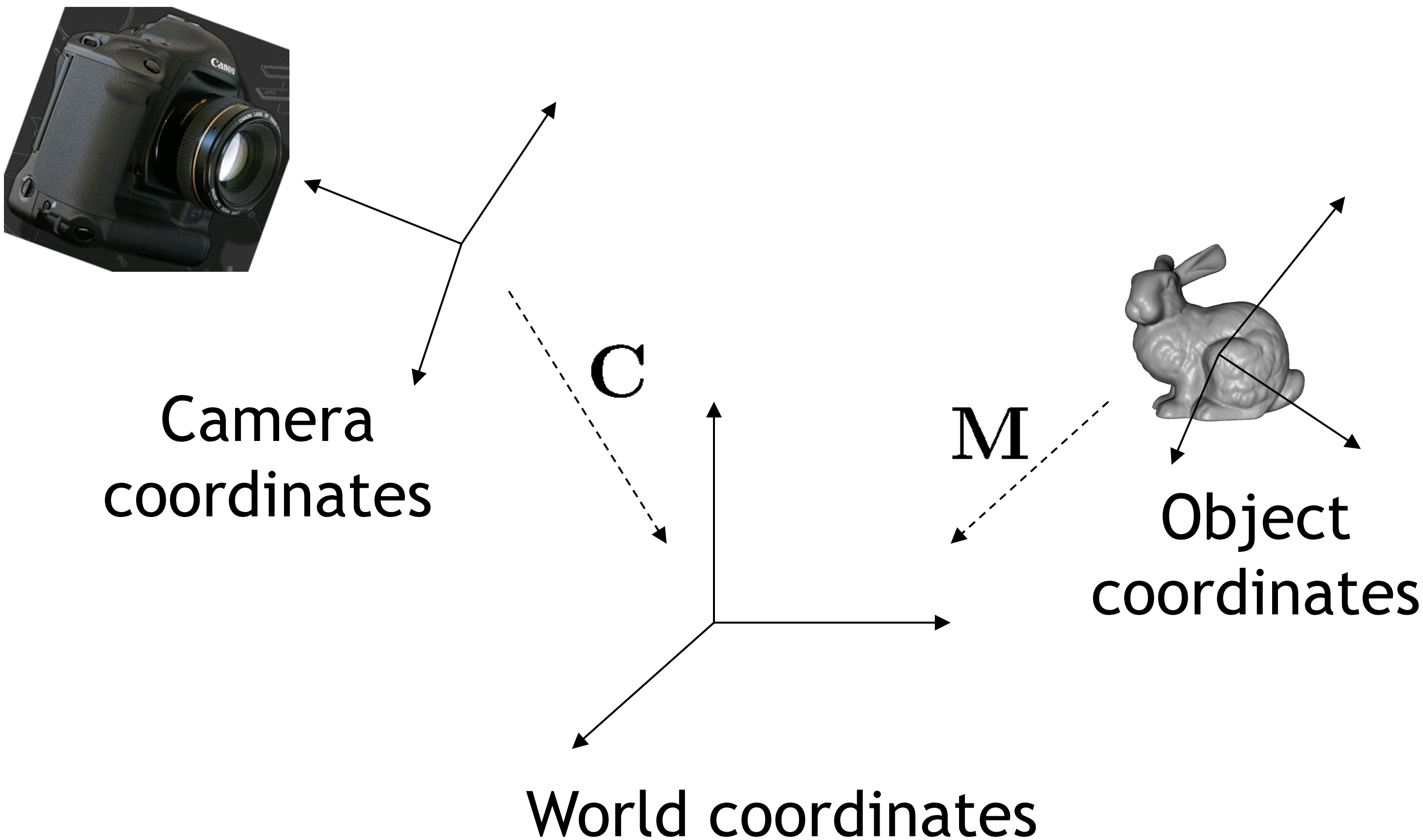
- Tracing
- [printCamera()]
- [printProjection()]

Camera routine in Processing

```
void setup() {
  size(640, 360, P3D);
  }

void draw() {
  background(0);

  camera(width/2, height/2, (height/2) / tan(PI/6),
       width/2, height/2, 0,     0, 1, 0);

  translate(width/2, height/2, -100);
  stroke(255);
  noFill();
  box(200);
}
```

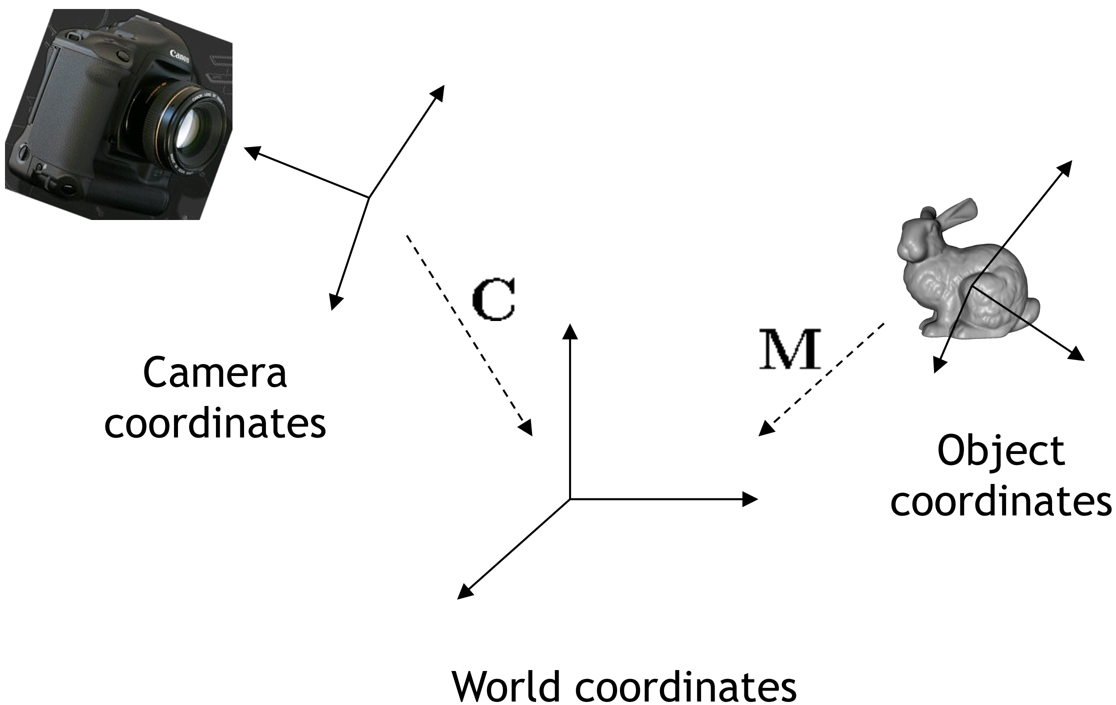
Common coordinate systems

- Camera, world, and object coordinates
- Matrices for change of coordinates $\mathbf{C}$, $\mathbf{M}$



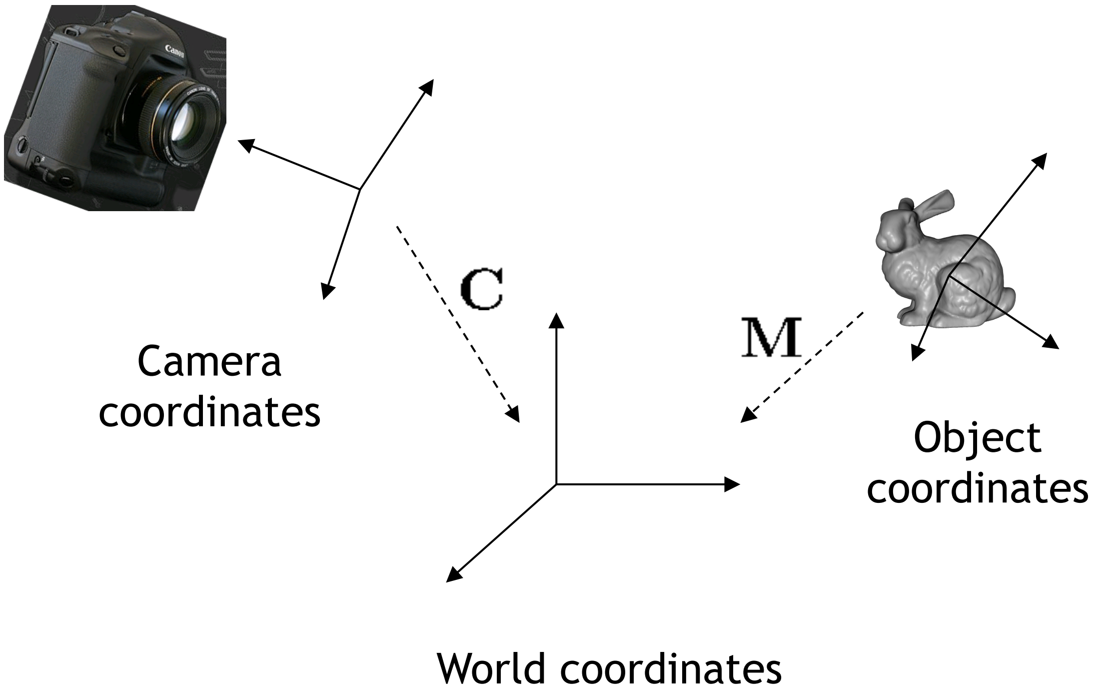
Object coordinates

- Coordinates the object is defined with
- Often origin is in middle, base, or corner of object
- No right answer, whatever was convenient for the creator of the object



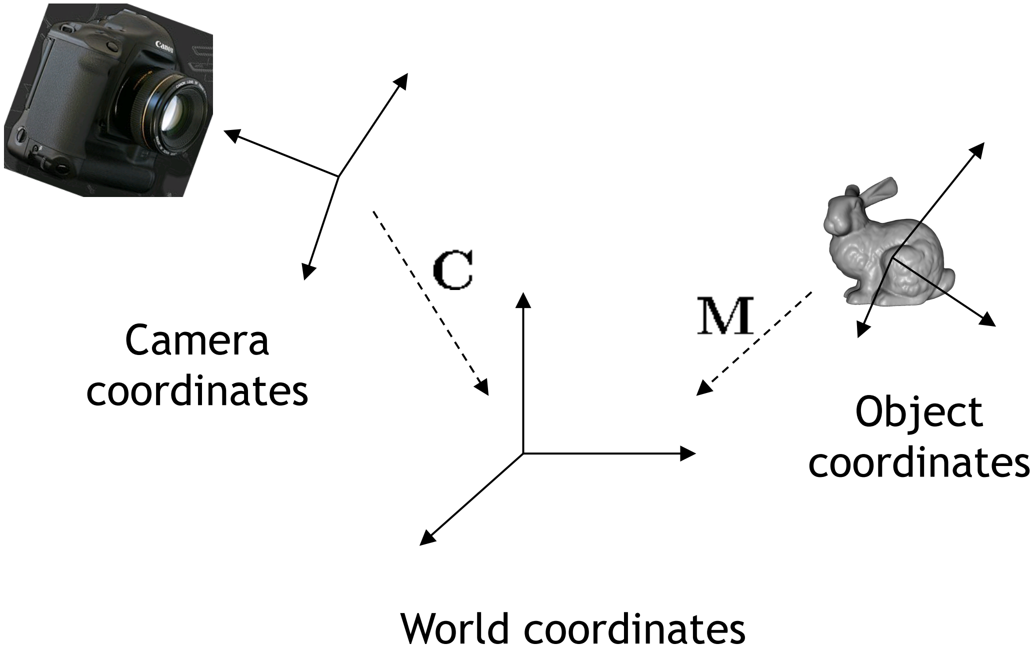
World coordinates

- "World space"
- Common reference frame for all objects in the scene
- Chosen for convenience, no right answer
  - If there is a ground plane, usually x-y is horizontal and z points up



Camera coordinate system

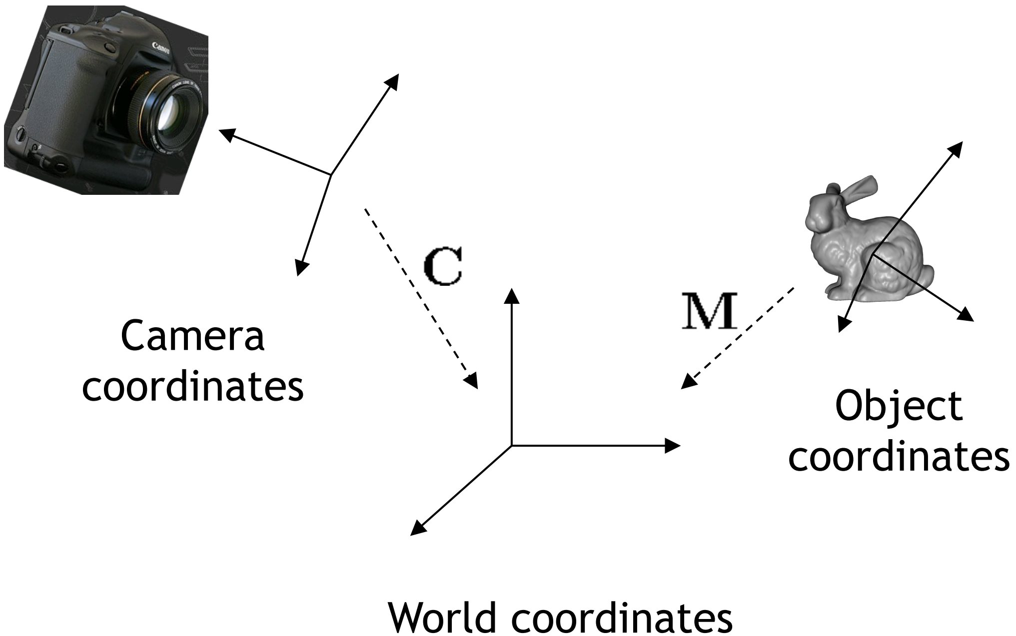
- “Camera space”
- Origin defines center of projection of camera
- Common convention in 3D graphics
  - x-y plane is parallel to image plane
  - z-axis is perpendicular to image plane



Camera coordinate system

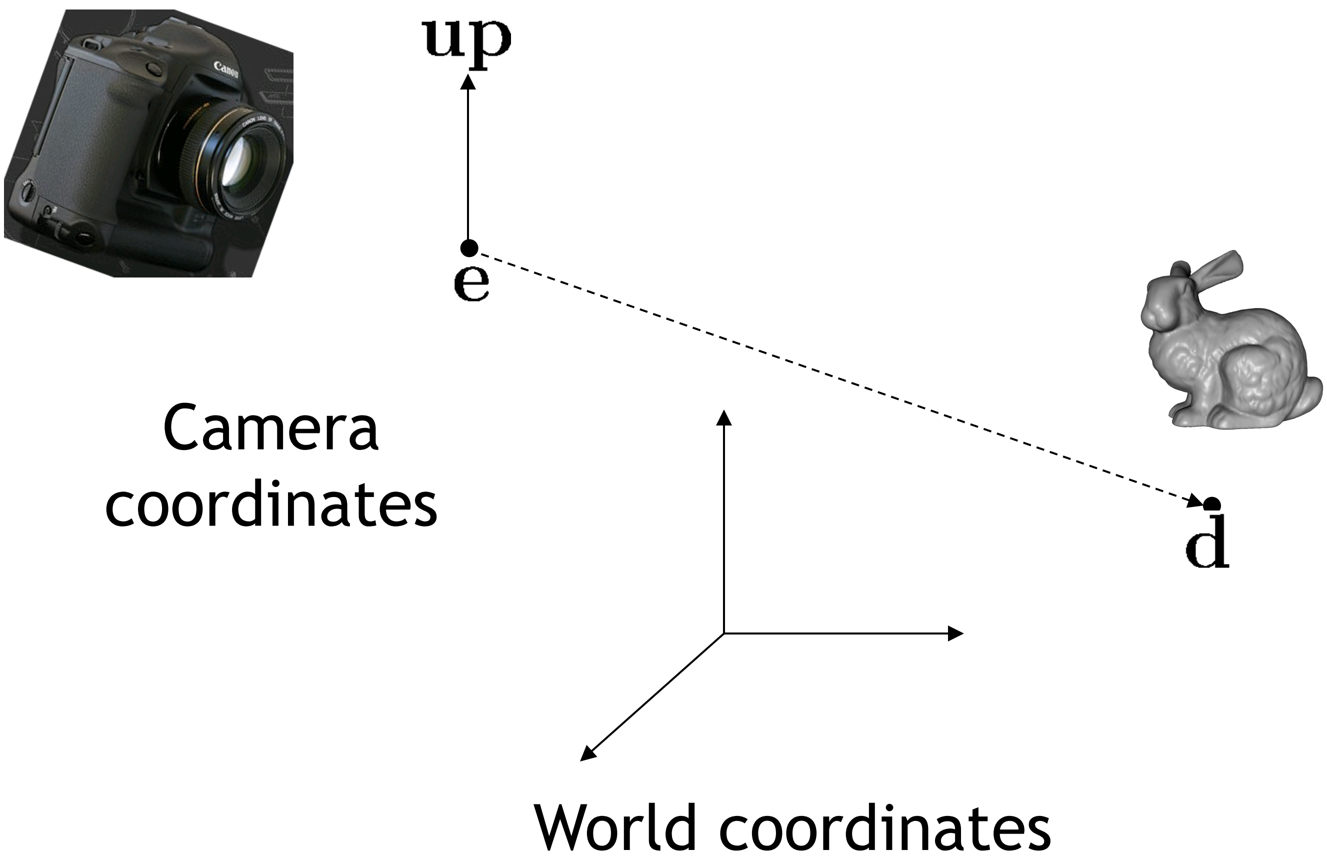
- "Camera matrix" defines transformation from camera to world coordinates
  - Placement of camera in world
- Transformation from object to camera coordinates

$$\mathbf{p}_{camera} = \mathbf{C}^{-1}\mathbf{M}\mathbf{p}_{object}$$

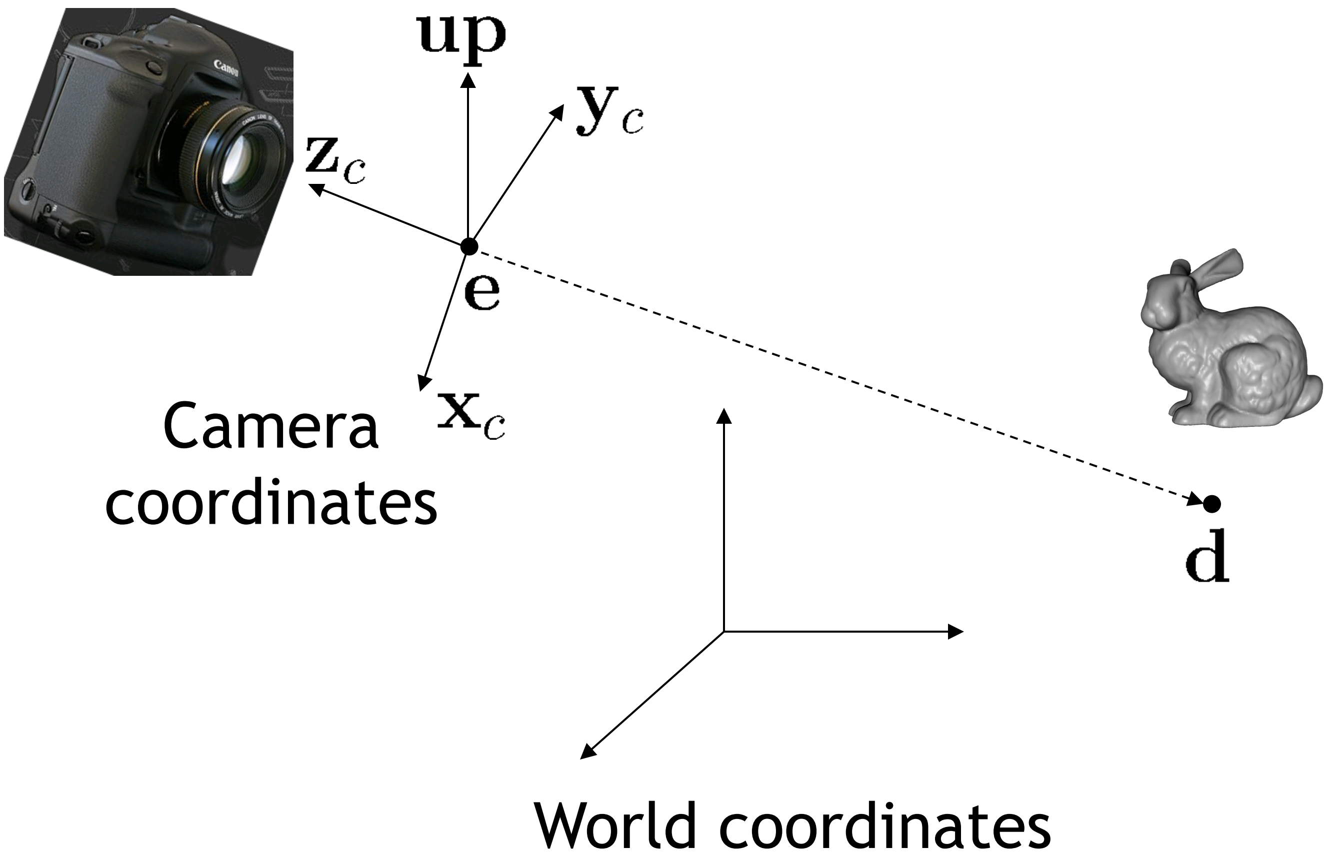


Camera matrix

- Construct from center of projection $\mathbf{e}$, look at $\mathbf{d}$, up-vector $\mathbf{up}$ (given in world coords.)



- Construct from center of projection $\mathbf{e}$, look at $\mathbf{d}$, up-vector $\mathbf{up}$ (given in world coords.)



Camera matrix

- z-axis
- x-axis
- y-axis

$$\mathbf{z}_c = \frac{\mathbf{e} - \mathbf{d}}{\|\mathbf{e} - \mathbf{d}\|}$$

$$\mathbf{x}_c = \frac{\mathbf{up} \times \mathbf{z}_c}{\|\mathbf{up} \times \mathbf{z}_c\|}$$

Camera matrix

- z-axis
- x-axis
- y-axis

$$\mathbf{z}_c = \frac{\mathbf{e} - \mathbf{d}}{\|\mathbf{e} - \mathbf{d}\|}$$

$$\mathbf{x}_c = \frac{\mathbf{up} \times \mathbf{z}_c}{\|\mathbf{up} \times \mathbf{z}_c\|}$$

$$\mathbf{y}_c = \mathbf{z_c} \times \mathbf{x}_c$$

Camera matrix

- Camera to world transformation

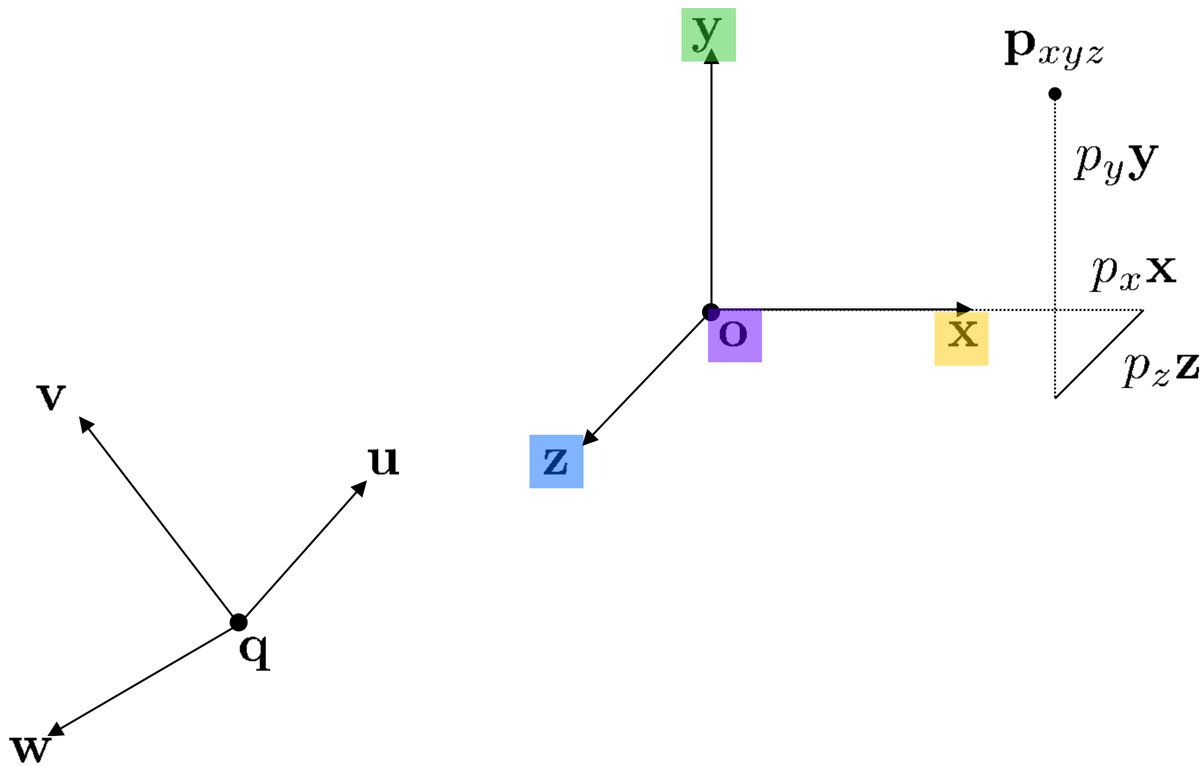
$$\mathbf{C} = \begin{bmatrix} \mathbf{x_c} & \mathbf{y_c} & \mathbf{z_c} & \mathbf{e} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- Think abou pute

$$\mathbf{p}' = \mathbf{C}\mathbf{p}$$

$$\mathbf{q}' = \mathbf{C}^{-1}\mathbf{q}$$

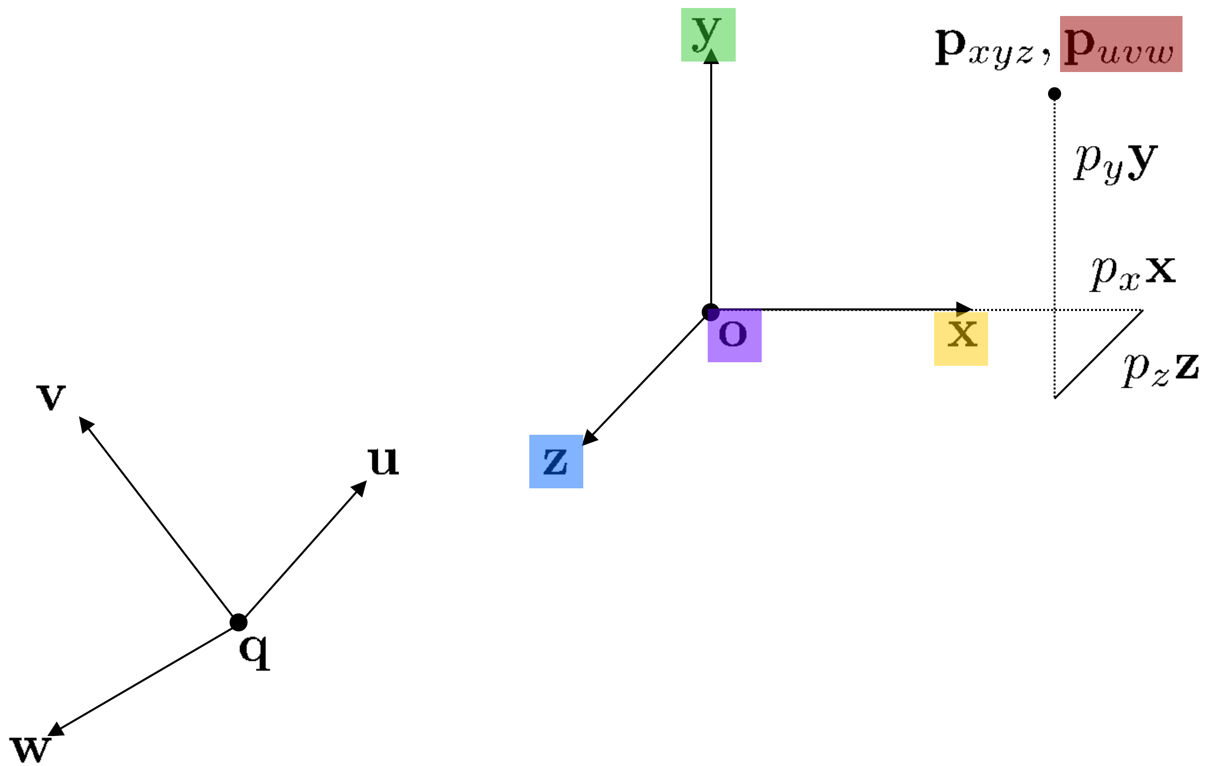
Change of coordinates



Coordinates of $\mathbf{xyzo}$ frame w.r.t. $\mathbf{uvwq}$ frame

$$\mathbf{x} = \begin{bmatrix} x_u \\ x_v \\ x_w \\ 0 \end{bmatrix} \qquad \mathbf{y} = \begin{bmatrix} y_u \\ y_v \\ y_w \\ 0 \end{bmatrix} \qquad \mathbf{z} = \begin{bmatrix} z_u \\ z_v \\ z_w \\ 0 \end{bmatrix} \qquad \mathbf{o} = \begin{bmatrix} o_u \\ o_v \\ o_w \\ 1 \end{bmatrix}$$

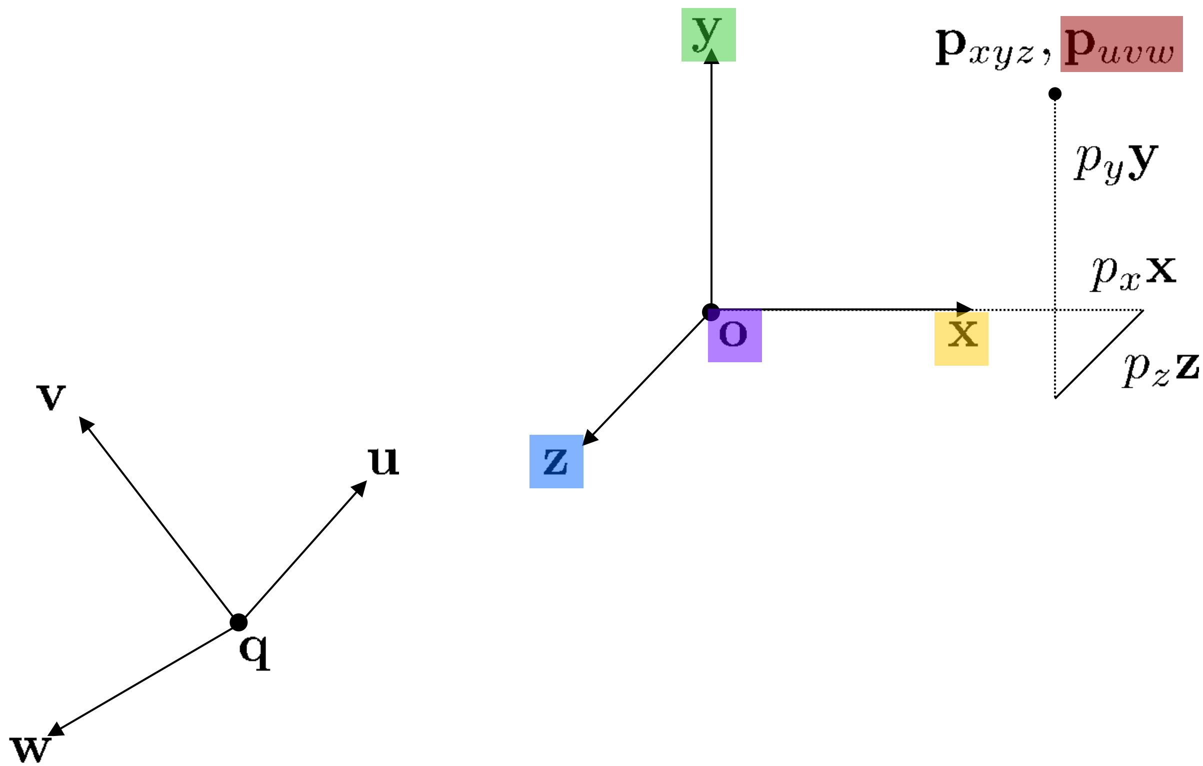
Change of coordinates



Same point $\mathbf{p}$ in 3D, expressed in new $\mathbf{uvwq}$ frame

$$\mathbf{p}_{uvw} = p_x \begin{bmatrix} x_u \\ x_v \\ x_w \\ 0 \end{bmatrix} + p_y \begin{bmatrix} y_u \\ y_v \\ y_w \\ 0 \end{bmatrix} + p_z \begin{bmatrix} z_u \\ z_v \\ z_w \\ 0 \end{bmatrix} + \begin{bmatrix} o_u \\ o_v \\ o_w \\ 1 \end{bmatrix}$$

Change of coordinates



$$\mathbf{p}_{uvw} = \begin{bmatrix} x_u & y_u & z_u & o_u \\ x_v & y_v & z_v & o_v \\ x_w & y_w & z_w & o_w \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ p_z \\ 1 \end{bmatrix} = \begin{bmatrix} \mathbf{x} & \mathbf{y} & \mathbf{z} & \mathbf{o} \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ p_z \\ 1 \end{bmatrix}$$

Change of coordinates

- Given coordinates

$$\mathbf{x} = \begin{bmatrix} x_u \\ x_v \\ x_w \\ 0 \end{bmatrix} \qquad \mathbf{y} = \begin{bmatrix} y_u \\ y_v \\ y_w \\ 0 \end{bmatrix} \qquad \mathbf{z} = \begin{bmatrix} z_u \\ z_v \\ z_w \\ 0 \end{bmatrix} \qquad \mathbf{o} = \begin{bmatrix} o_u \\ o_v \\ o_w \\ 1 \end{bmatrix}$$

c… **yzo** wi… …ct to n… …e **uvw…**

- Coordinates of any point $\mathbf{p}_{xyz}$ with respect to new frame **uvwq** are

$$\mathbf{p}_{uvw} = \begin{bmatrix} x_u & y_u & z_u & o_u \\ x_v & y_v & z_v & o_v \\ x_w & y_w & z_w & o_w \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ p_z \\ 1 \end{bmatrix} = \begin{bmatrix} \mathbf{x} & \mathbf{y} & \mathbf{z} & \mathbf{o} \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ p_z \\ 1 \end{bmatrix}$$

- Matri…
coorc…ates (u,v,w,q)

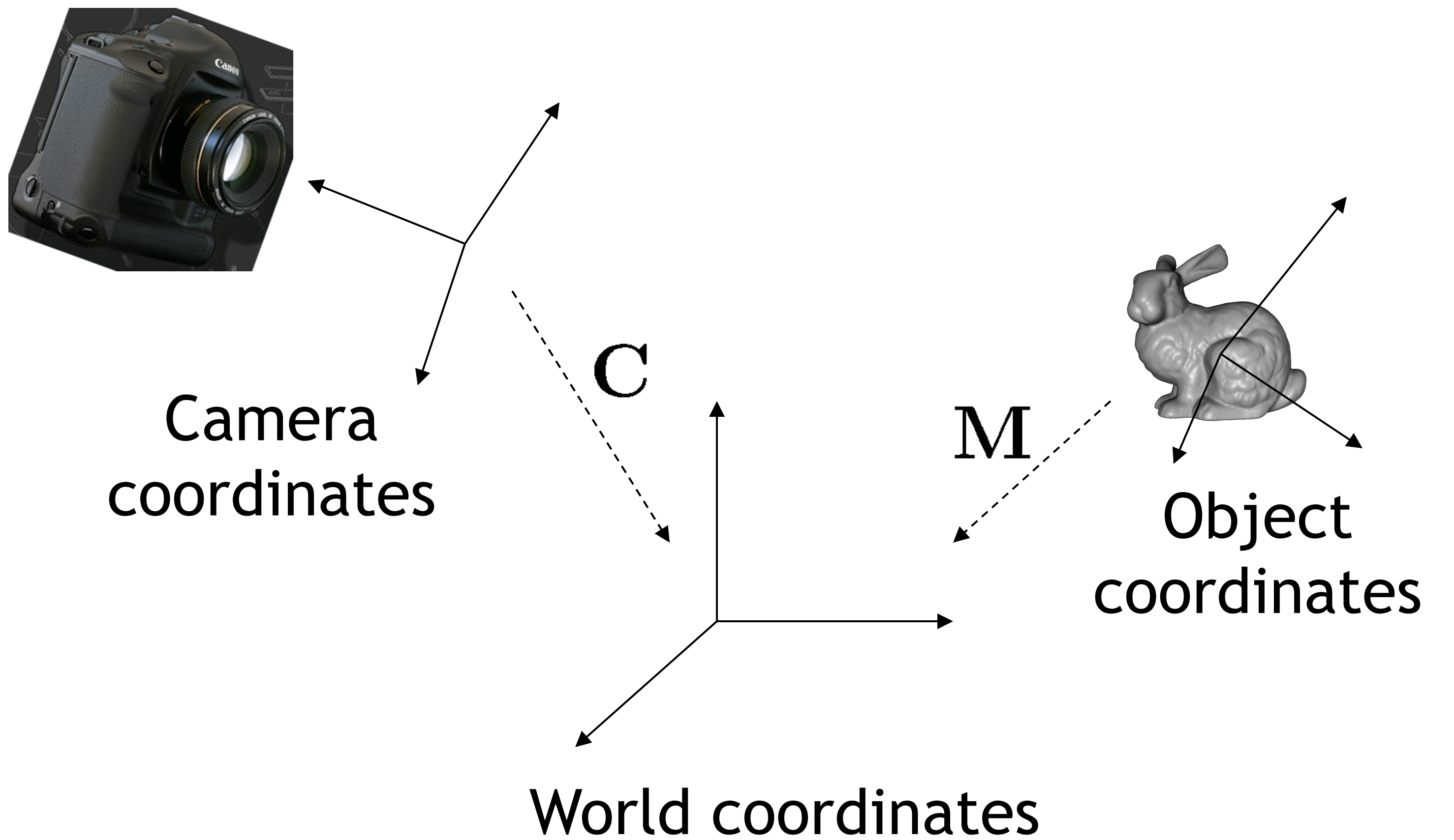
Change of coordinates

Inverse transformation

- Given point $\mathbf{p}_{uvw}$ w.r.t. frame $\mathbf{u}, \mathbf{v}, \mathbf{w}, \mathbf{q}$
- Want coordinates $\mathbf{p}_{xyz}$ w.r.t. frame $\mathbf{x}, \mathbf{y}, \mathbf{z}, \mathbf{o}$

$$\mathbf{p}_{xyz} = \begin{bmatrix} x_u & y_u & z_u & o_u \\ x_v & y_v & z_v & o_v \\ x_w & y_w & z_w & o_w \\ 0 & 0 & 0 & 1 \end{bmatrix}^{-1} \begin{bmatrix} p_u \\ p_v \\ p_w \\ 1 \end{bmatrix}$$

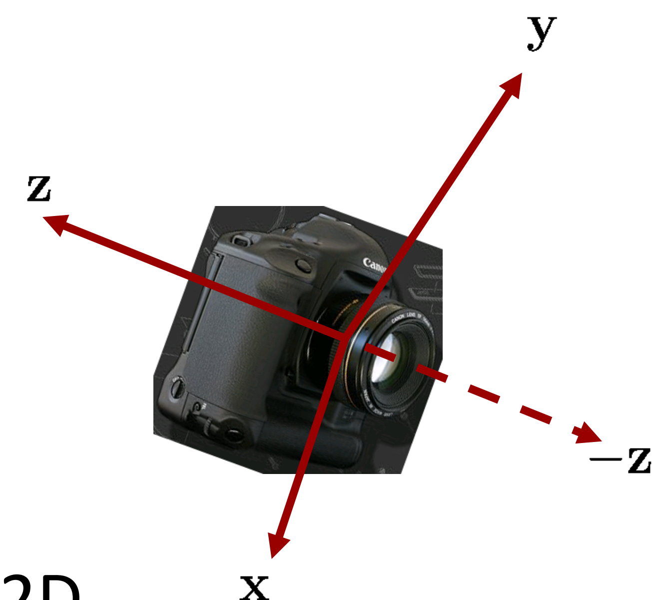
Object, world, camera coords.



$$\mathrm{p}' = \mathrm{C}^{-1}\mathrm{M}\mathrm{p}$$

Objects in camera coordinates

- We have things lined up the way we like them on screen
  - x to the right
  - y up
  - $-z$ going into the screen
  - Objects to look at are in front of us, i.e. have negative z values
- But objects are still in 3D
- Today: how to project them into 2D



Camera coordinates

Projections

- Given 3D points (vertices) in camera coordinates, determine corresponding 2D image coordinates

Orthographic projection

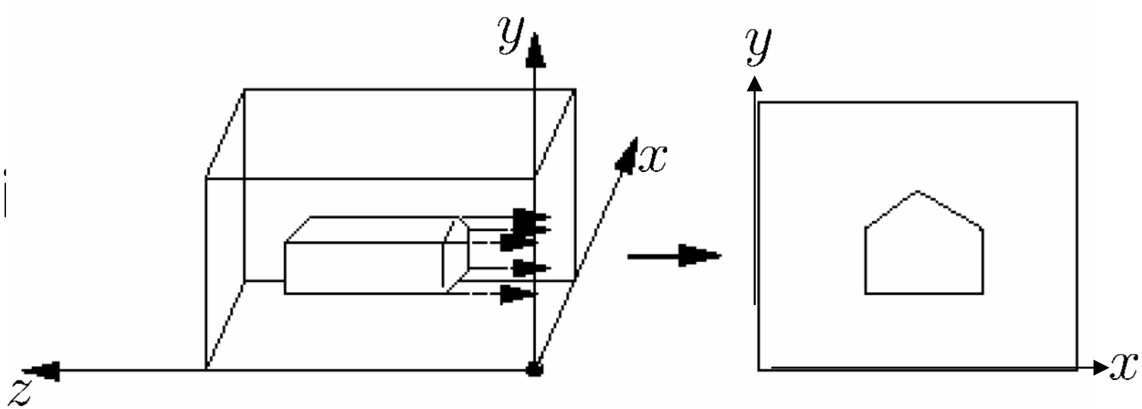
- Simply ignore z-coordinate
- Use camera space xy coordinates as image coordinates

- What we want, or not?

Orthographic projection

- Project points to x-y plane along parallel lines

- Graphi



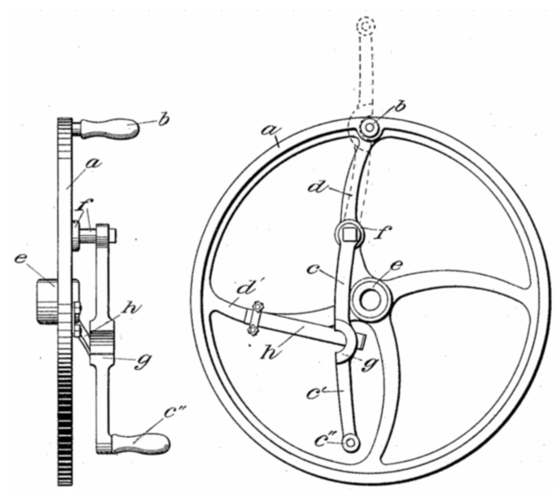
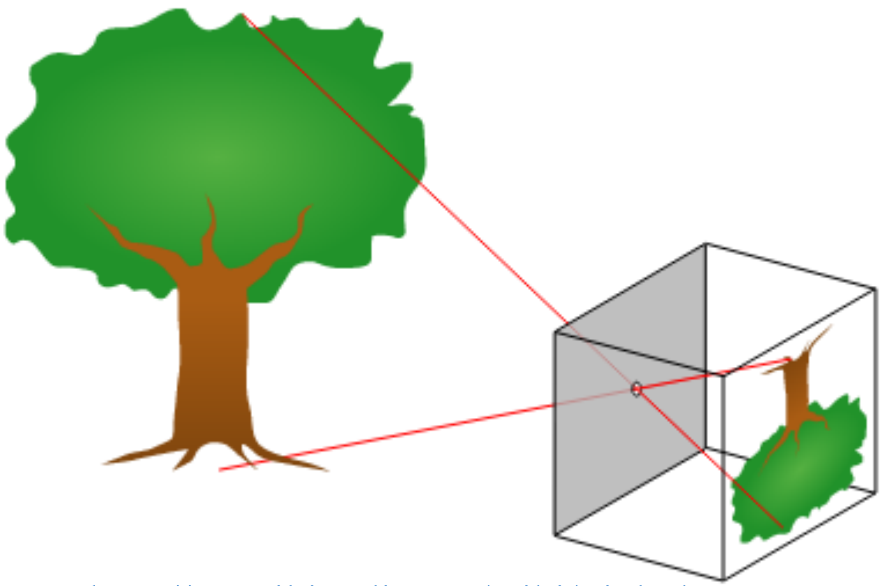


Fig. 2. Fig. 1.

Perspective projection

- Most common for computer graphics
- Simplified model of human eye, or camera lens (pinhole camera)
- Things farther away seem smaller
- Discovery/description attributed to Filippo Brunelleschi, early 1400's

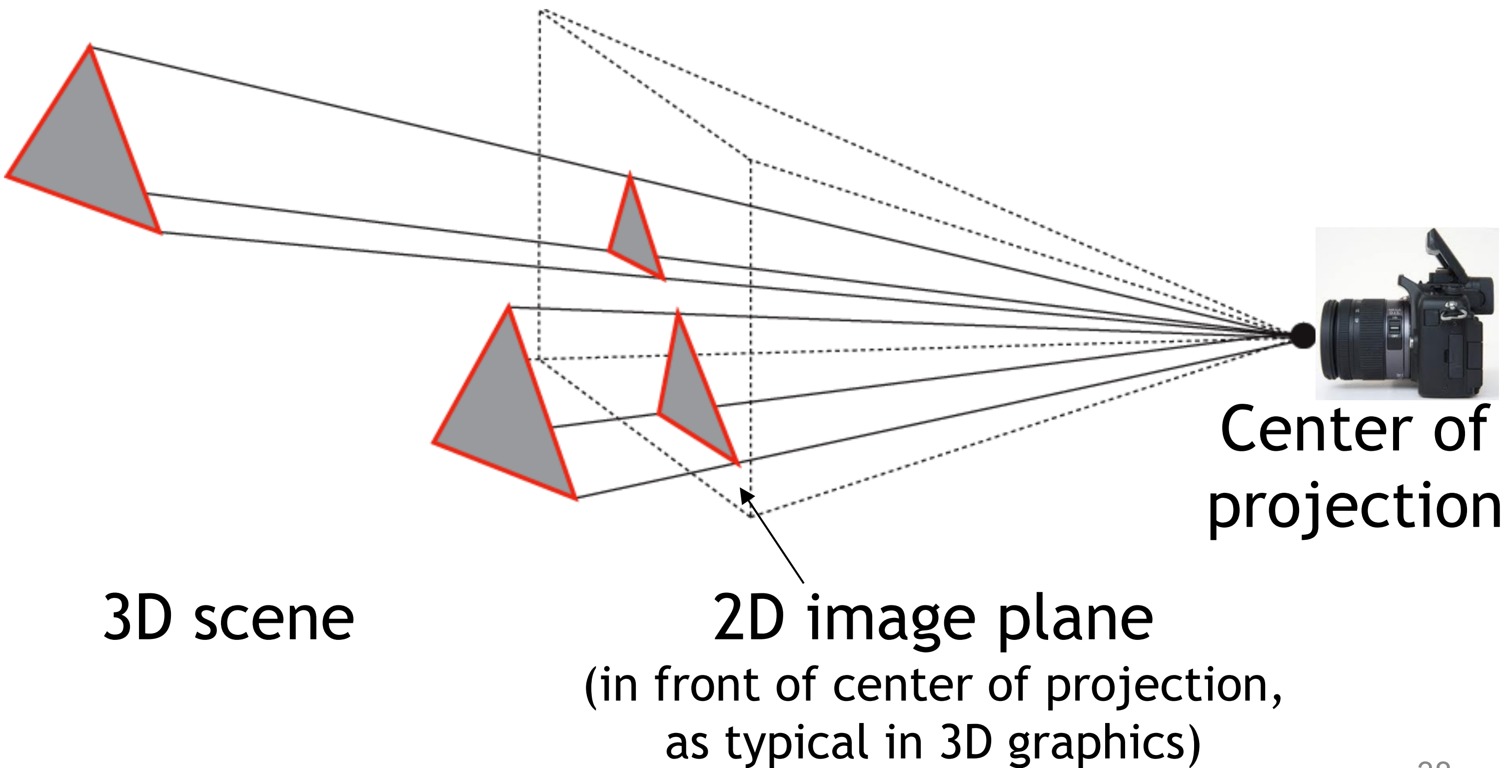


http://en.wikipedia.org/wiki/Pinhole_camera

Projection plane behind center of projection, flipped image

Perspective projection

- Project along rays that converge in center of projection



Perspective projection

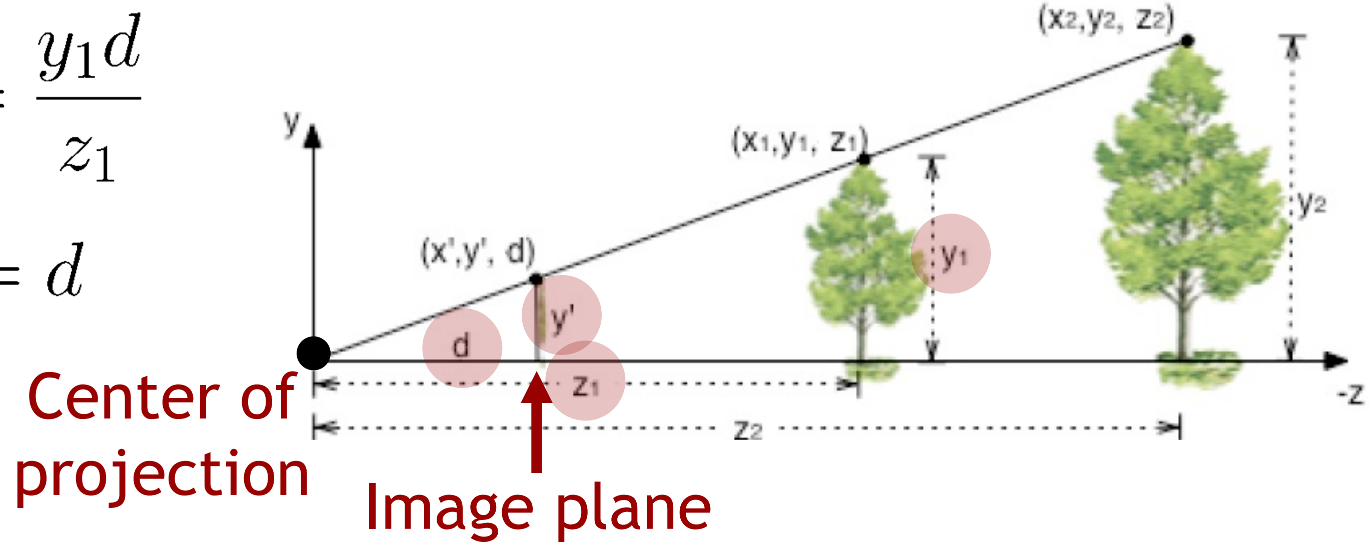
Parallel lines
no longer parallel,
converge at one point

Earliest example
La Trinitá (1427) by Masaccio
http://en.wikipedia.org/wiki/Holy_Trinity_(Masaccio)

The math: simplified case

$$y' = \frac{y_1 d}{z_1}$$

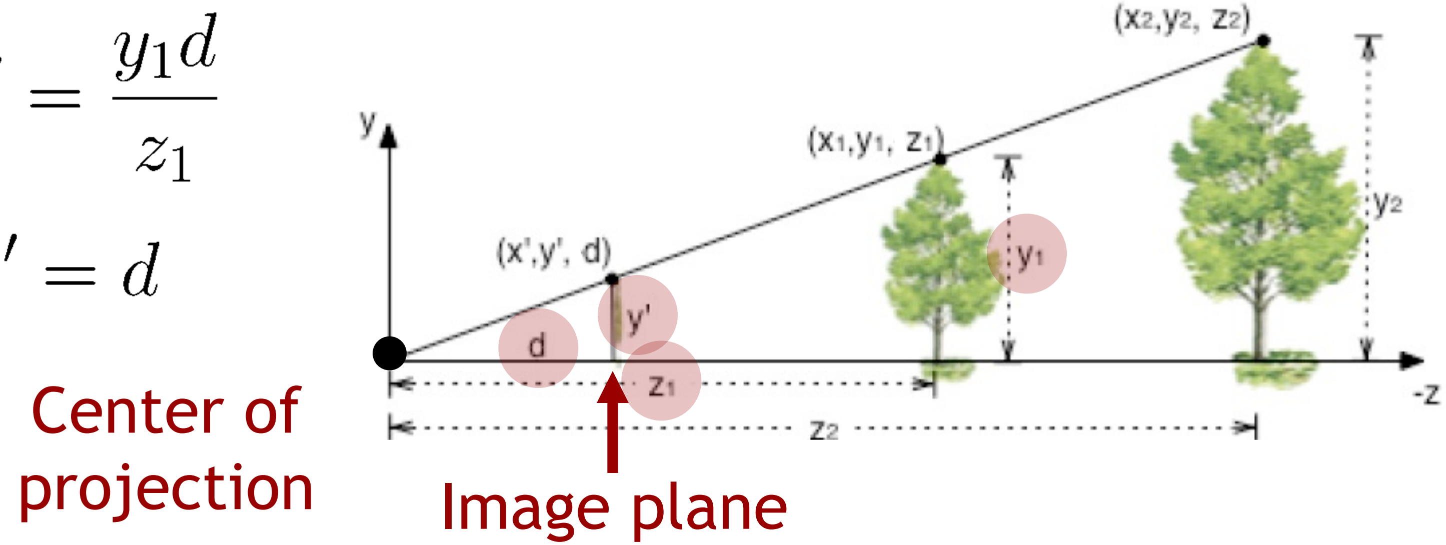
$$z' = d$$



The math: simplified case

$$y' = \frac{y_1 d}{z_1}$$

$$z' = d$$

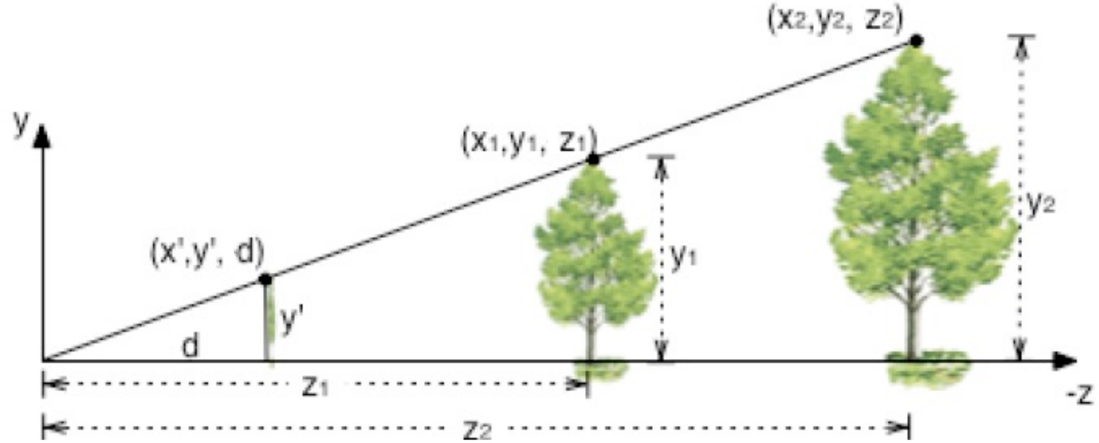


- Can express this using homogeneous coordinates, 4x4 matrices

The math: simplified case

$$y' = \frac{y_1 d}{z_1}$$

$$z' = d$$



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

Projection matrix

Homogeneous coord. != 1!
Homogeneous division

Perspective projection

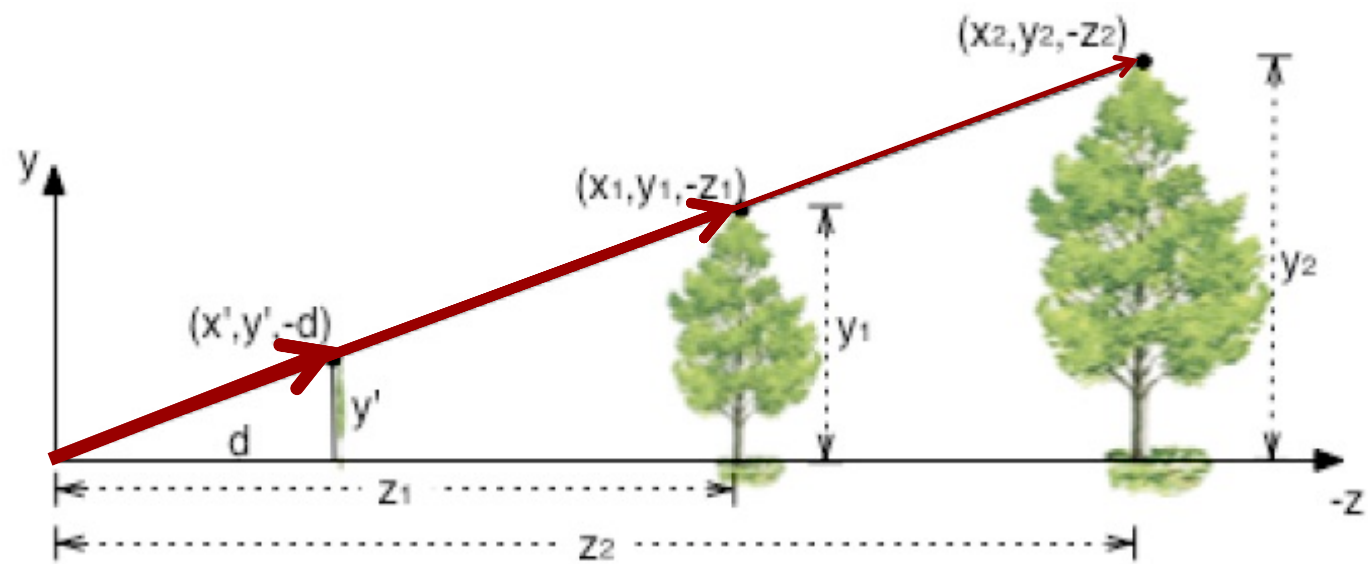
$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \\ z/d \end{bmatrix} \Rightarrow \begin{bmatrix} xd/z \\ yd/z \\ d \\ 1 \end{bmatrix}$$

Projection matrix **Homogeneous division**

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

Intuitive example

- All points that lie on one projection line (i.e., a "line-of-sight", intersecting with center of projection of camera) are projected onto same image point
- All 3D points on one projection line are equivalent
- Projection lines form 2D projective space, or 2D projective plane



3D Projective space

- Projective space $\mathbf{P}^3$ represented using $\mathbf{R}^4$ and homogeneous coordinates
  - Each point along 4D ray is equivalent to same 3D point at w=1

$$\begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} \sim \begin{bmatrix} \lambda x \\ \lambda y \\ \lambda z \\ \lambda w \end{bmatrix} \sim \begin{bmatrix} x/w \\ y/w \\ z/w \\ 1 \end{bmatrix}$$

„equivalent“

1D vector subspace, arbitrary scalar value λ

Equivalent element, for any λ

3D Projective space

- Projective mapping (transformation):
  any non-singular linear mapping on homogeneous coordinates, for example,

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

- Generalization of affine mappings
  - 4th row of matrix is arbitrary (not restricted to $[0\ 0\ 0\ 1]$)
- Projective mappings are collineations
  http://en.wikipedia.org/wiki/Projective_linear_transformation
  http://en.wikipedia.org/wiki/Collineation
  - Preserve straight lines, but not parallel lines
- Much more theory
  http://www.math.toronto.edu/mathnet/questionCorner/projective.html
  http://en.wikipedia.org/wiki/Projective_space

Projective space

Projective space

http://en.wikipedia.org/wiki/Projective_space

- [xyzw] homogeneous coordinates
- includes points at infinity ($w=0$)
- projective mappings (perspective projection)

Vector space

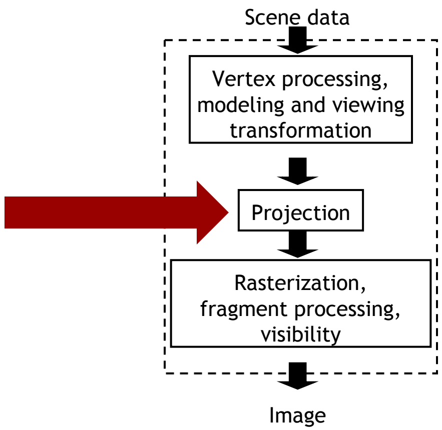
- [xyz] coordinates
- represents vectors
- linear mappings (rotation around origin, scaling, shear)

Affine space

- [xyz1], [xyz0] homogeneous coords.
- distinguishes points and vectors
- affine mappings (translation)

In practice

- Use 4x4 homogeneous matrices like other 4x4 matrices
- Modeling & viewing transformations are affine mappings
  - points keep $w=1$
  - no need to divide by w when doing modeling operations or transforming into camera space
- 3D-to-2D projection is a projective transform
  - Resulting w coordinate not always 1
- Divide by w (perspective division, homogeneous division) after multiplying with projection matrix
  - OpenGL rendering pipeline (graphics hardware) does this automatically



Today

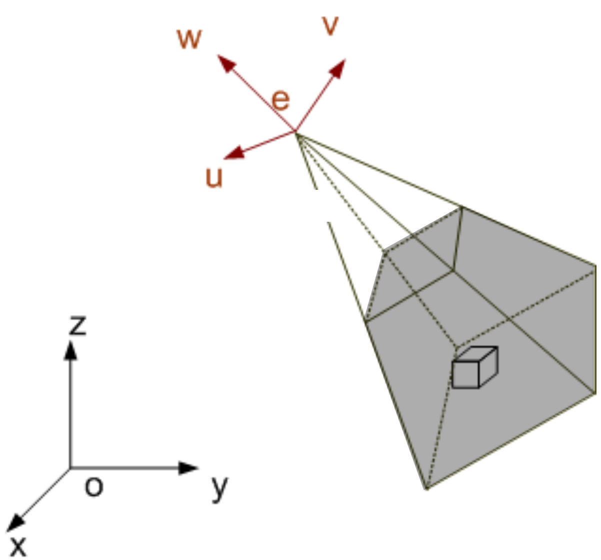
- Rendering pipeline
- Projections
- View volumes, clipping
- Viewport transformation

View volumes

- View volume is 3D volume seen by camera

Perspective view volume

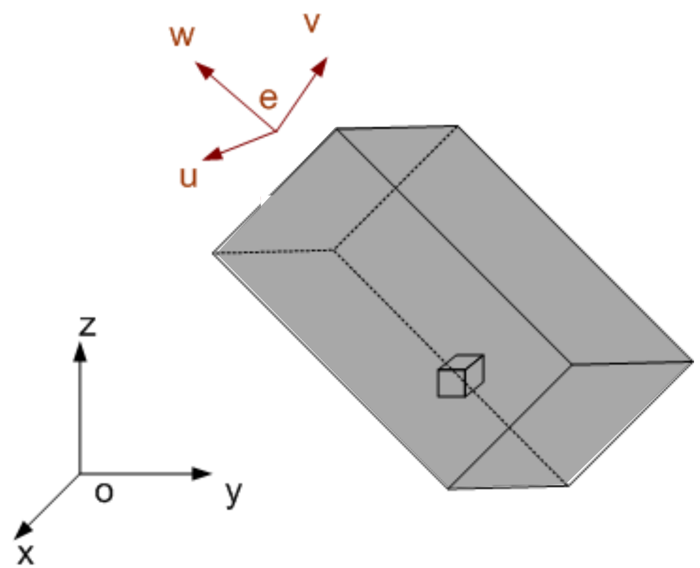
Camera coordinates



World coordinates

Orthographic view volume

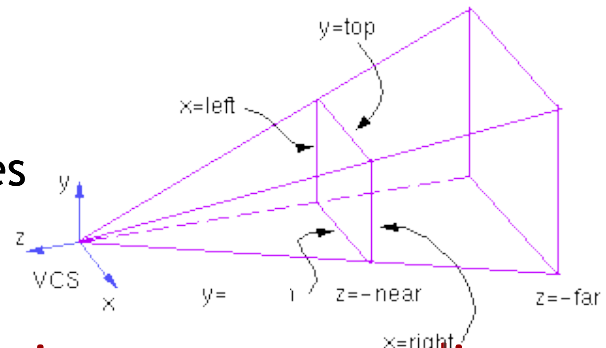
Camera coordinates



World coordinates

General view volume

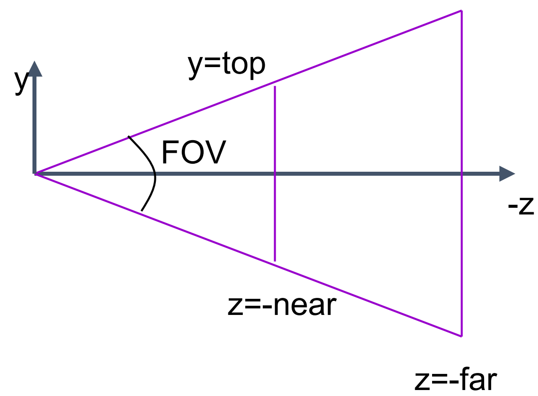
Camera coordinates



- Defined by 6 parameters, in camera coordinates
  - Left, right, top, bottom boundaries
  - Near, far clipping planes
- Clipping planes to avoid numerical problems
  - Divide by zero
  - Low precision for distant objects
- Often symmetric, i.e., left=-right, top=-bottom

Perspective view volume

Symmetric view volume

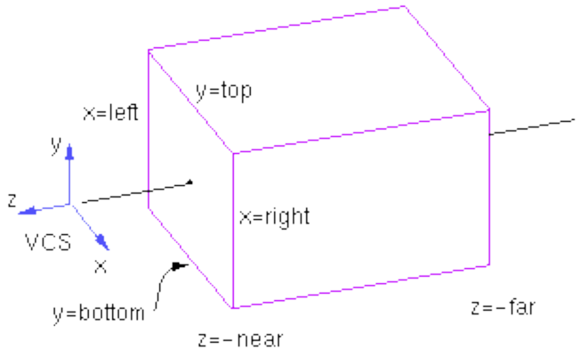


- Only 4 parameters
  - Vertical field of view (FOV)
  - Image aspect ratio (width/height)
  - Near, far clipping planes

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

$$\tan(FOV/2) = \frac{top}{near}$$

Orthographic view volume

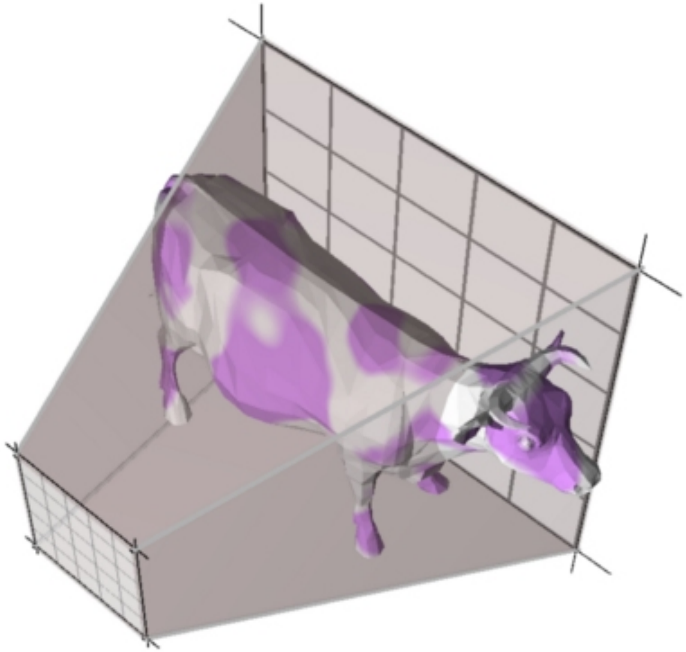


Camera coordinates

- Parametrized by 6 parameters
  - Right, left, top, bottom, near, far
- If symmetric
  - Width, height, near, far

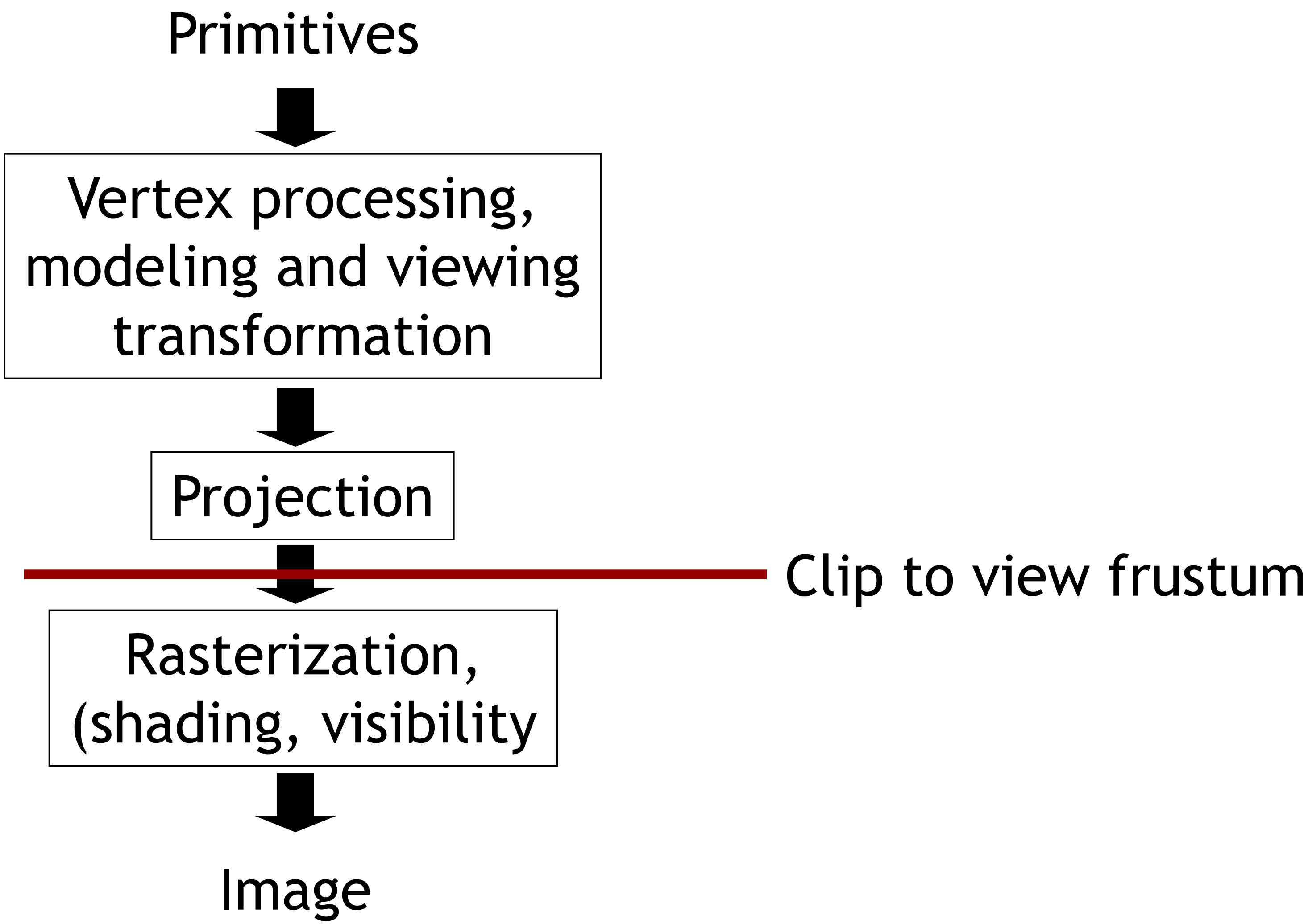
Clipping

- Need to identify objects outside view volume
  - Avoid division by zero
  - Efficiency, don’t draw objects outside view volume
- Performed by OpenGL rendering pipeline
- Clipping always to canonic view volume
  - Cube [-1..1]x[-1..1]x[-1..1] cent



- Need to transform desired view frustum to canonic view frustum

Clipping


Primitives
Vertex processing, modeling and viewing transformation
Projection
Clip to view frustum
Rasterization, (shading, visibility
Image

Canonic view volume

- Projection matrix is set such that
  - User defined view volume is transformed into canonic view volume, i.e., unit cube [-1,1]x[-1,1]x[-1,1]

    “Multiplying vertices of view volume by projection matrix and performing homogeneous divide yields canonic view volume, i.e., cube [-1,1]x[-1,1]x[-1,1]“

- Perspective and orthographic projection are treated exactly the same way

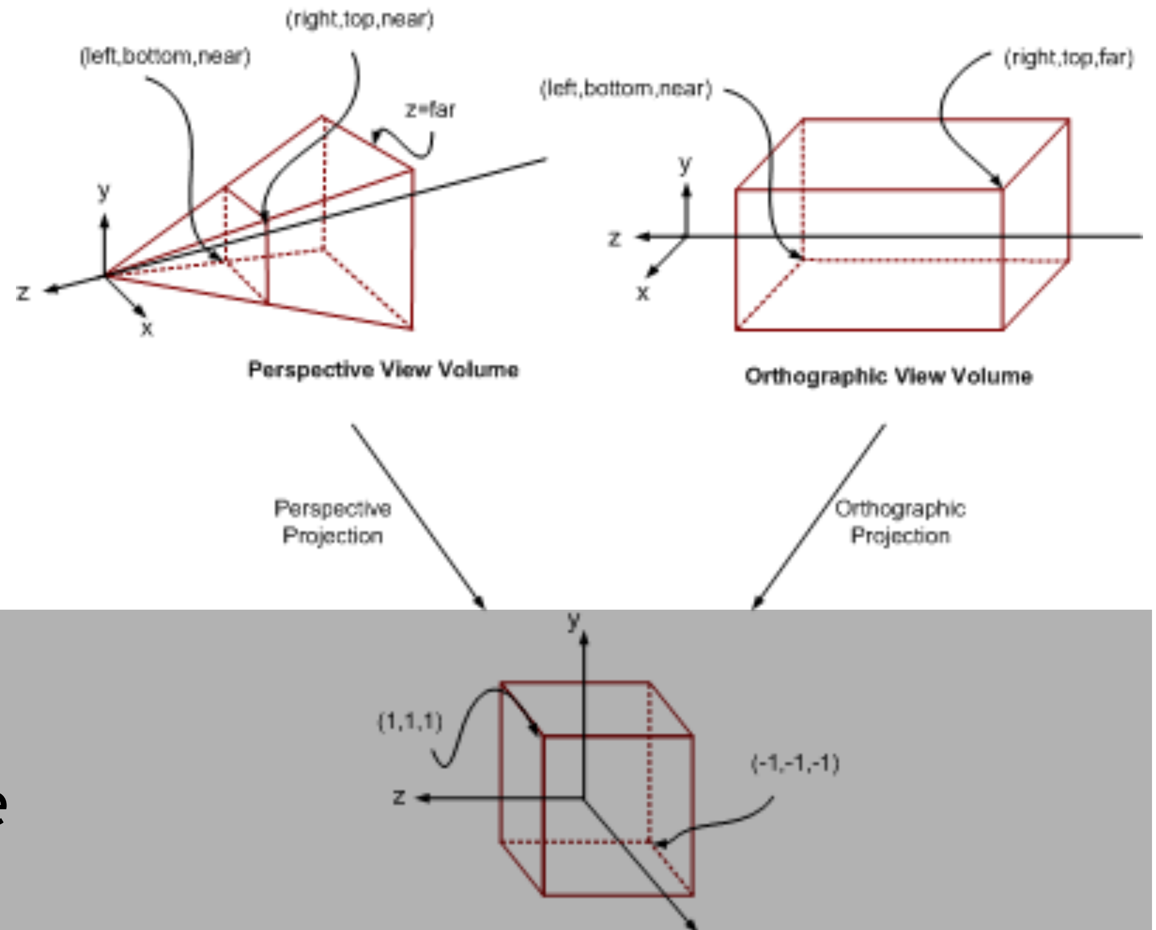
Projection matrix

Camera coordinates

Projection matrix

Canonic view volume

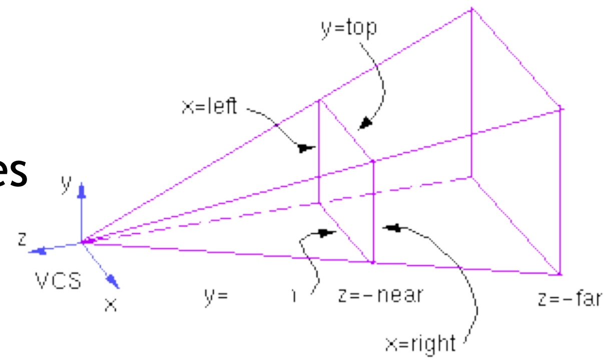
Viewport transformation (later)



Perspective projection matrix

- General view frustum

Camera coordinates



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

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

Perspective projection matrix

- Compare to simple projection matrix from before

Simple projection

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{bmatrix}$$

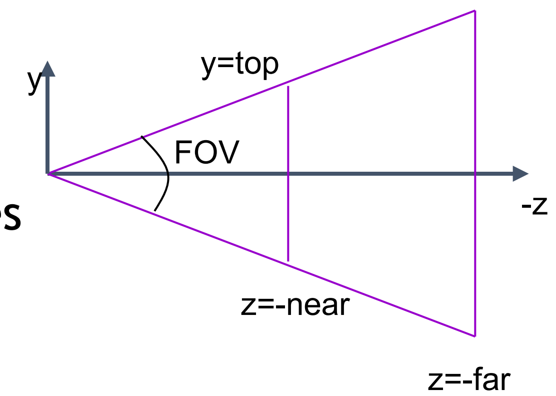
General view frustum

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

Perspective projection matrix

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

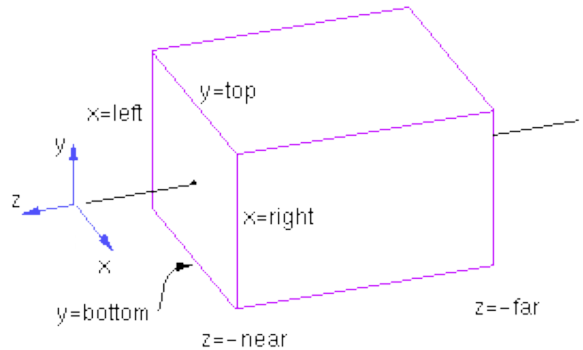
Camera coordinates



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

Orthographic projection matrix

Camera coordinates



$$\mathbf{P}_{ortho}(right, left, top, bottom, near, far) = \begin{bmatrix} \frac{2}{right-left} & 0 & 0 & -\frac{right+left}{right-left} \\ 0 & \frac{2}{top-bottom} & 0 & -\frac{top+bottom}{top-bottom} \\ 0 & 0 & \frac{2}{far-near} & \frac{far+near}{far-near} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{P}_{ortho}(width, height, near, far) = \begin{bmatrix} \frac{2}{width} & 0 & 0 & 0 \\ 0 & \frac{2}{height} & 0 & 0 \\ 0 & 0 & \frac{2}{far-near} & \frac{far+near}{far-near} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

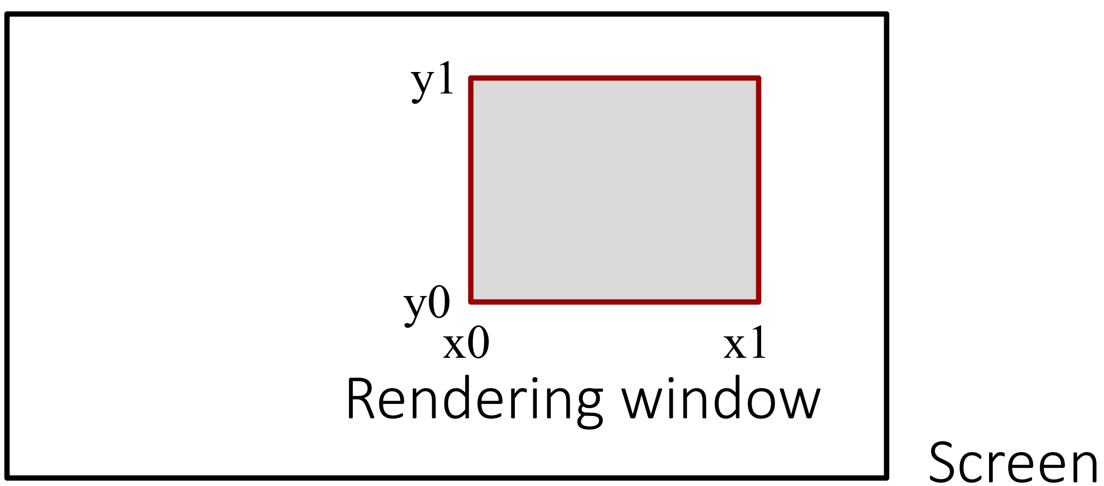
$w = 1$ after mult. with orthographic projection matrix

Today

- Rendering pipeline
- Projections
- View volumes
- Viewport transformation

Viewport transformation

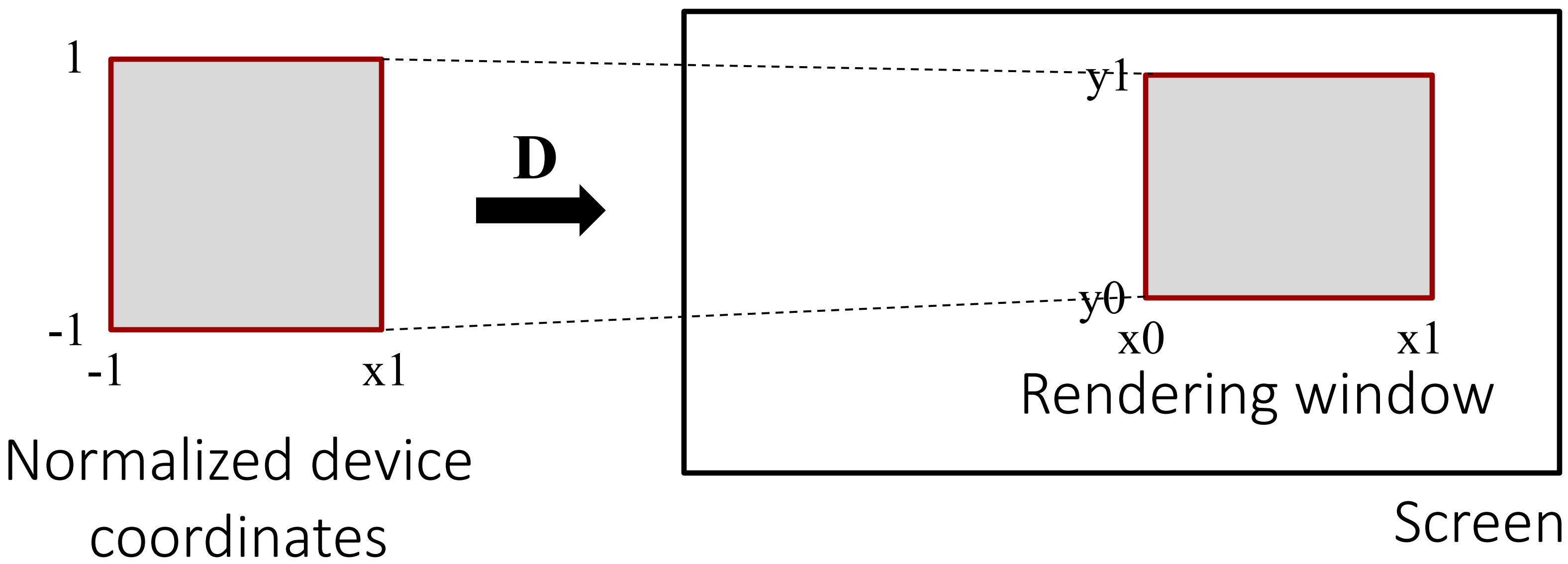
- After applying projection matrix, image points are in normalized view coordinates
  - Per definition range [-1..1] x [-1..1]
- Map points to image (i.e., pixel) coordinates
  - User defined range [x0…x1] x [y0…y1]
  - E.g., position of rendering window on screen



Viewport transformation

- Scale and translation

$$\mathbf{D}(x_0, x_1, y_0, y_1) = \begin{bmatrix} (x_1 - x_0)/2 & 0 & 0 & (x_0 + x_1)/2 \\ 0 & (y_1 - y_0)/2 & 0 & (y_0 + y_1)/2 \\ 0 & 0 & 1/2 & 1/2 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



The complete transform

- Mapping a 3D point in object coordinates to pixel coordinates
- Object-to-world matrix $\mathbf{M}$, camera matrix $\mathbf{C}$, projection matrix $\mathbf{C}$, viewport matrix $\mathbf{D}$

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

Object space

The complete transform

- Mapping a 3D point in object coordinates to pixel coordinates
- Object-to-world matrix $\mathbf{M}$, camera matrix $\mathbf{C}$, projection matrix $\mathbf{C}$, viewport matrix $\mathbf{D}$

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

Object space

World space

The complete transform

- Mapping a 3D point in object coordinates to pixel coordinates
- Object-to-world matrix $\mathbf{M}$, camera matrix $\mathbf{C}$, projection matrix $\mathbf{C}$, viewport matrix $\mathbf{D}$

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

Object space

World space

Camera space

The complete transform

- Mapping a 3D point in object coordinates to pixel coordinates
- Object-to-world matrix $\mathbf{M}$, camera matrix $\mathbf{C}$, projection matrix $\mathbf{C}$, viewport matrix $\mathbf{D}$

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

Object space

World space

Camera space

Canonic view volume

The complete transform

- Mapping a 3D point in object coordinates to pixel coordinates
- Object-to-world matrix $\mathbf{M}$, camera matrix $\mathbf{C}$, projection matrix $\mathbf{C}$, viewport matrix $\mathbf{D}$

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

Object space

World space

Camera space

Canonic view volume

Image space

The complete transform

- Mapping a 3D point in object coordinates to pixel coordinates
- Object-to-world matrix $\mathbf{M}$, camera matrix $\mathbf{C}$, projection matrix $\mathbf{C}$, viewport matrix $\mathbf{D}$

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

$$\mathbf{p}' = \begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix}$$

Pixel coordinates $\begin{matrix} x'/w' \\ y'/w' \end{matrix}$

OpenGL details

- Object-to-world matrix $\mathbf{M}$, camera matrix $\mathbf{C}$, projection matrix $\mathbf{P}$, viewport matrix $\mathbf{D}$

Model-view matrix

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

Projection matrix

- OpenGL rendering pipeline performs these matrix multiplications in vertex shader program
  - More on shader programs later in class
- User just specifies the model-view and projection matrices
- See Java code jrtr.GLRenderContext.draw and default vertex shader in file default.vert

OpenGL details

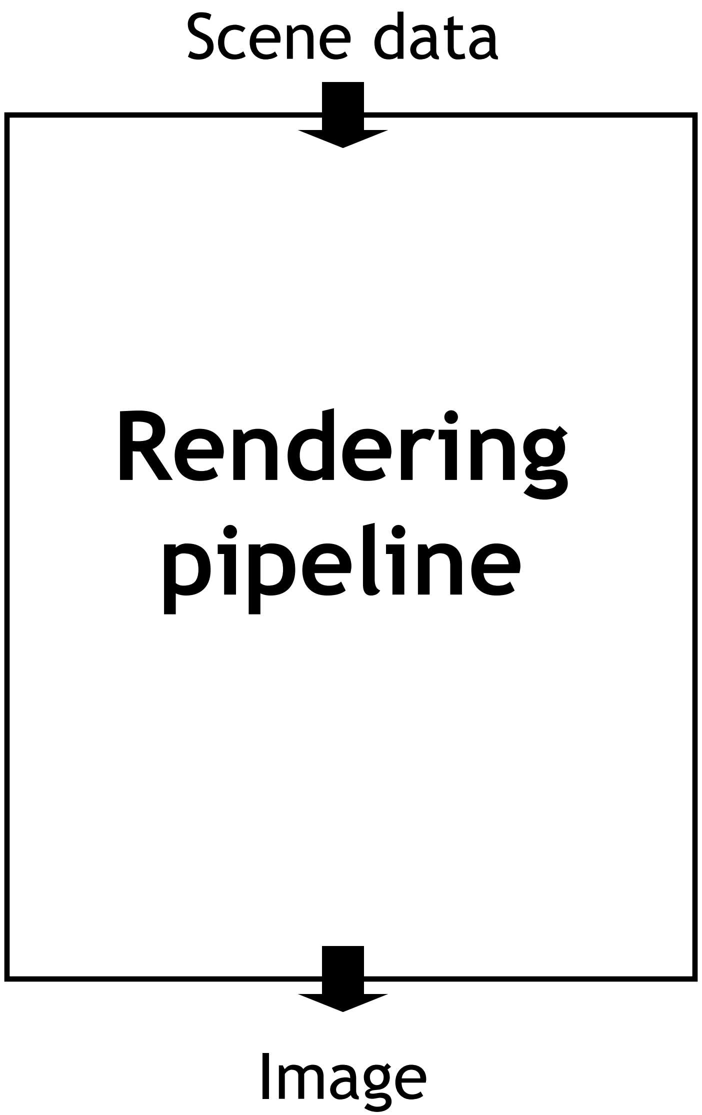
- Object-to-world matrix $\mathbf{M}$, camera matrix $\mathbf{C}$, projection matrix $\mathbf{P}$, viewport matrix $\mathbf{D}$

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

- Exception: viewport matrix, **D**
  - Specified implicitly via glViewport()
  - No direct access, not used in shader program

Rendering pipeline

http://en.wikipedia.org/wiki/Graphics_pipeline

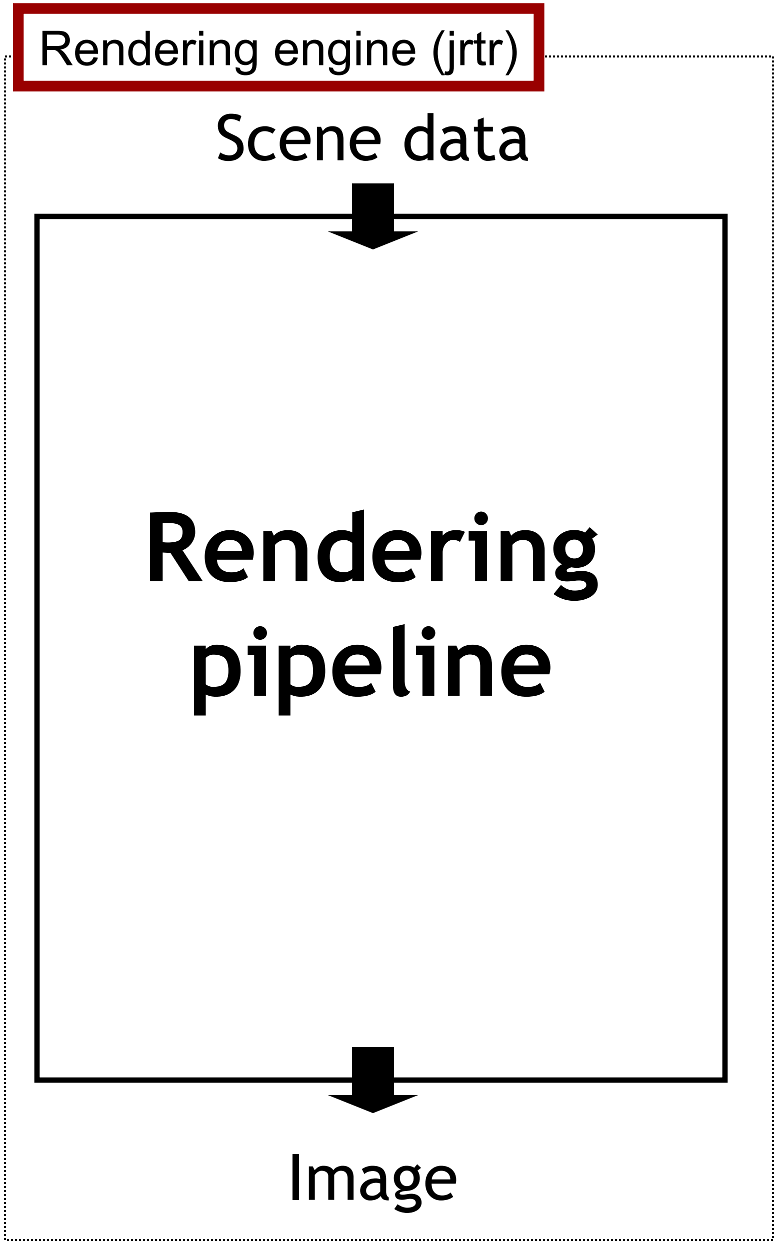


- Hardware & software that draws 3D scenes on the screen
- Most operations performed by specialized hardware (graphics processing unit, GPU, http://en.wikipedia.org/wiki/Graphics_processing_unit)
- Access to hardware through low-level 3D API (DirectX, OpenGL)
  - jogl is a Java binding to OpenGL, used in our projects http://jogamp.org/jogl/www/
- All scene data flows through the pipeline at least once for each frame (i.e., image)

Rendering pipeline

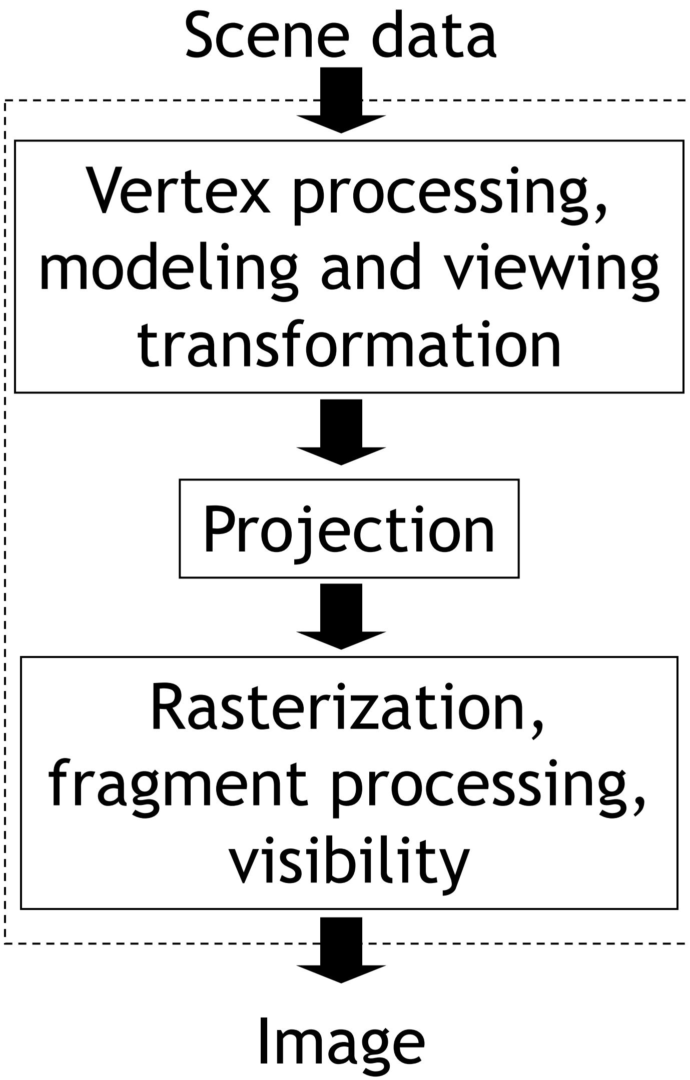
- Rendering pipeline implements object order algorithm
  - Loop over all objects
  - Draw triangles one by one (rasterization)
- Alternatives?
- Advantages, disadvantages?

Rendering engine

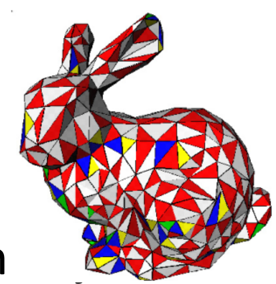


- Additional software layer ("middle-ware") encapsulating low-level API (OpenGL, DirectX, ...)
- Additional functionality (file I/O, scene management, ...)
- Layered software architecture common in industry
  - Game engines http://en.wikipedia.org/wiki/Game_engine

Rendering pipeline stages (simplified)

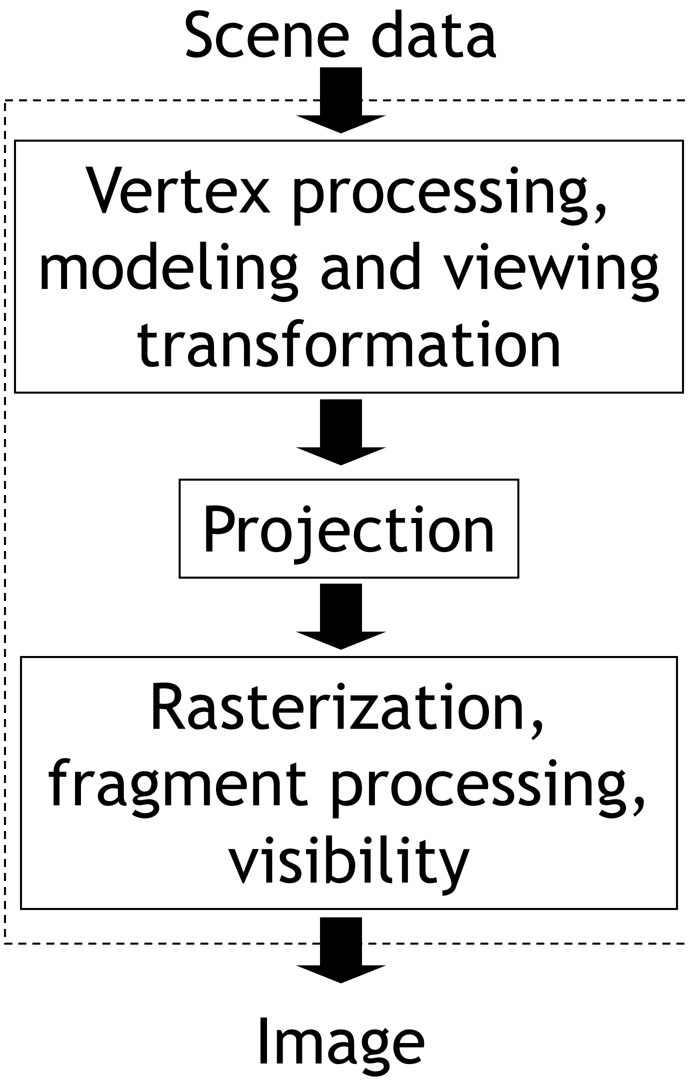


- Geometry
  - Vertices and how they are connected
  - Triangles, lines, point sprites, triangle strips
  - Attributes such as color



- Specified in dinates
- Processed by the rendering pipeline one-by-one

Rendering pipeline stages (simplified)

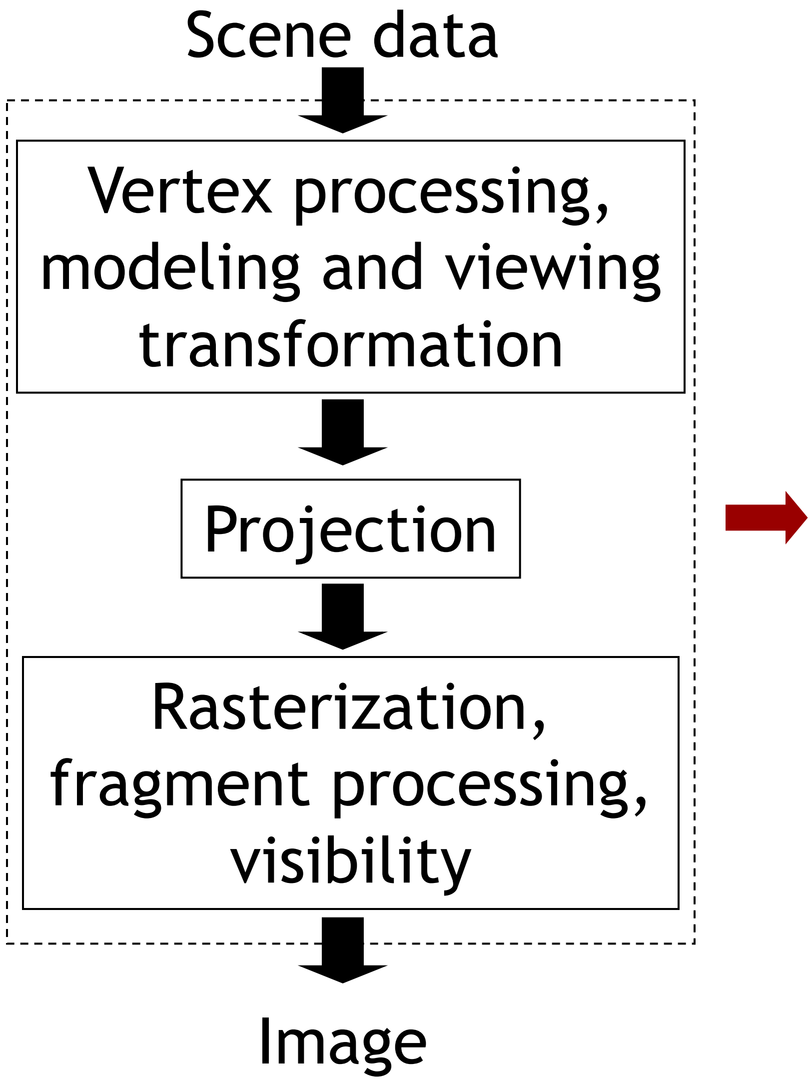


- Transform object to camera coordinates

$$\mathbf{p}_{camera} = \underbrace{\mathbf{C}^{-1}\mathbf{M}}_{\text{MODELVIEW matrix}}\mathbf{p}_{object}$$

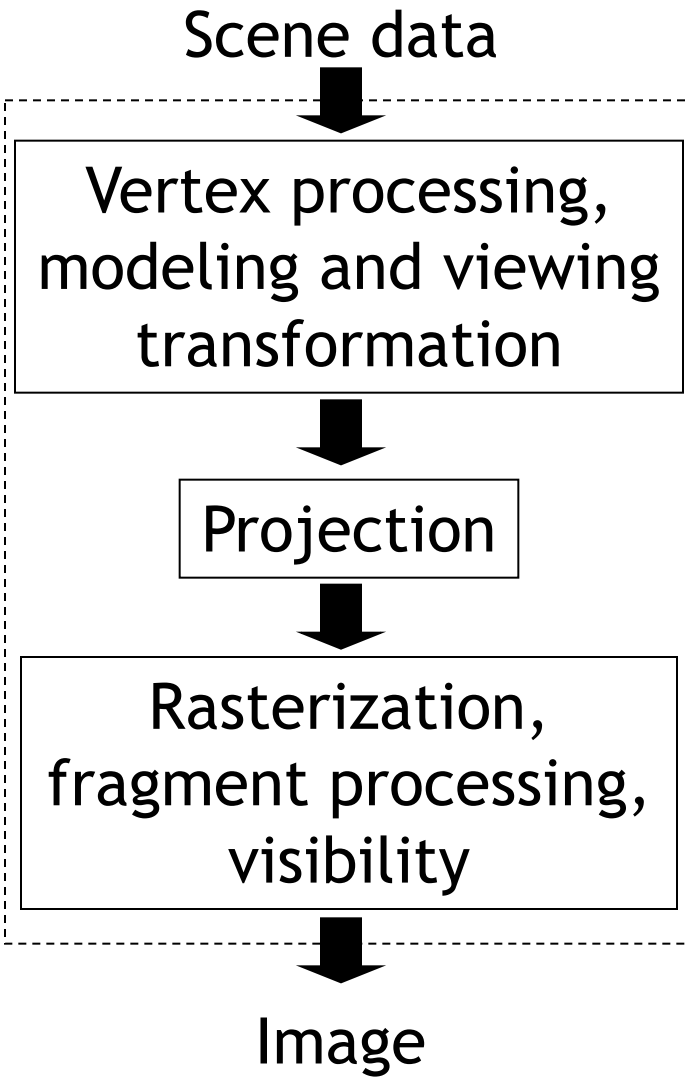
- Additional processing on per-vertex basis
  - Shading, i.e., computing per-vertex colors
  - Deformation, animation
  - Etc.

Rendering pipeline stages (simplified)

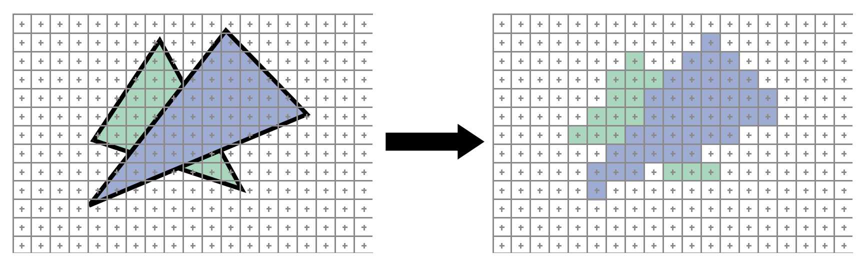


- Project 3D vertices to 2D image positions
- This lecture

Rendering pipeline stages (simplified)

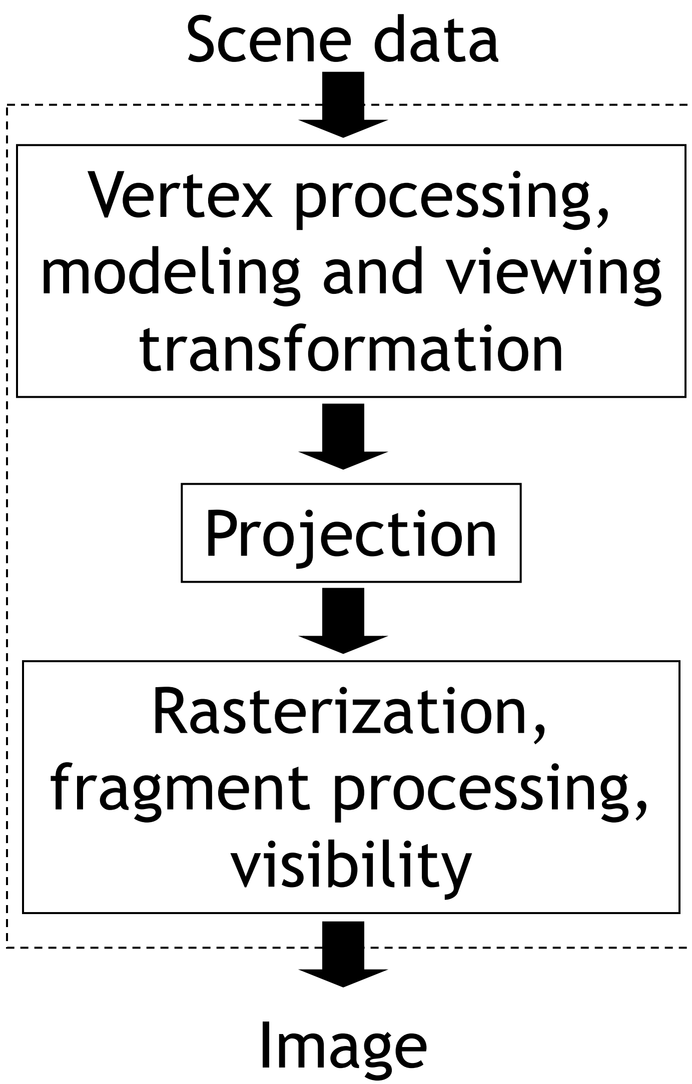


- Draw primitives pixel by pixel on 2D image (triangles, lines, point sprites, etc.)
- Compute per fragment (i.e., pixel) color
- Determine what is visible
- Next lecture



Rasterization

Rendering pipeline stages (simplified)



- Grid (2D array) of RGB pixel colors

References

- For today's Processing experiments see

- https://processing.org/tutorials/p3d/
- https://processing.org/tutorials/transform2d/