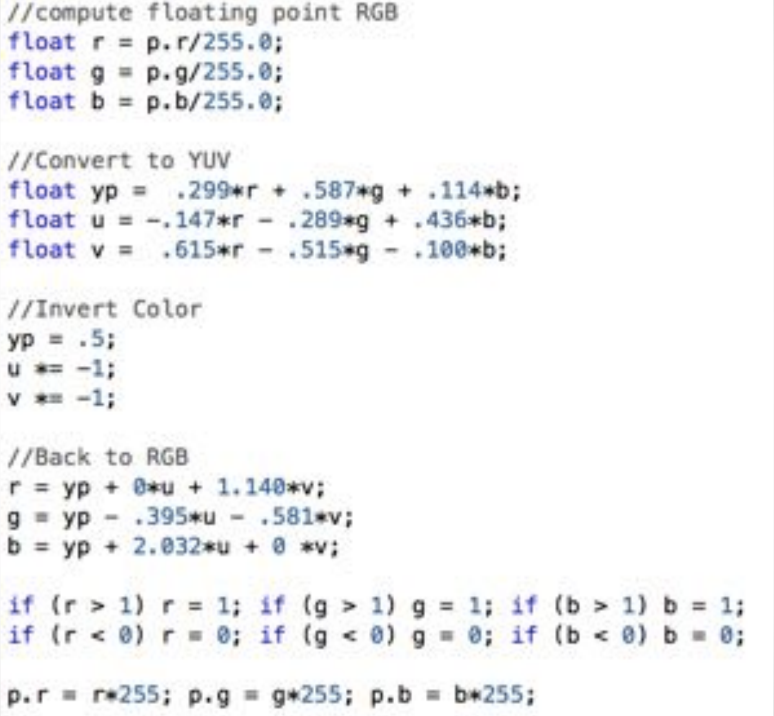
HOPKINS

HOPKINS



```
//compute floating point RGB
float r = p.r/255.0;
float g = p.g/255.0;
float b = p.b/255.0;

//Convert to YUV
float yp =  .299*r + .587*g + .114*b;
float u = -.147*r - .289*g + .436*b;
float v =  .615*r - .515*g - .100*b;

//Invert Color
yp = .5;
u *= -1;
v *= -1;

//Back to RGB
r = yp + 0*u + 1.140*v;
g = yp - .395*u - .581*v;
b = yp + 2.032*u + 0 *v;

if (r > 1) r = 1; if (g > 1) g = 1; if (b > 1) b = 1;
if (r < 0) r = 0; if (g < 0) g = 0; if (b < 0) b = 0;

p.r = r*255; p.g = g*255; p.b = b*255;
```

# Perspective & Rays

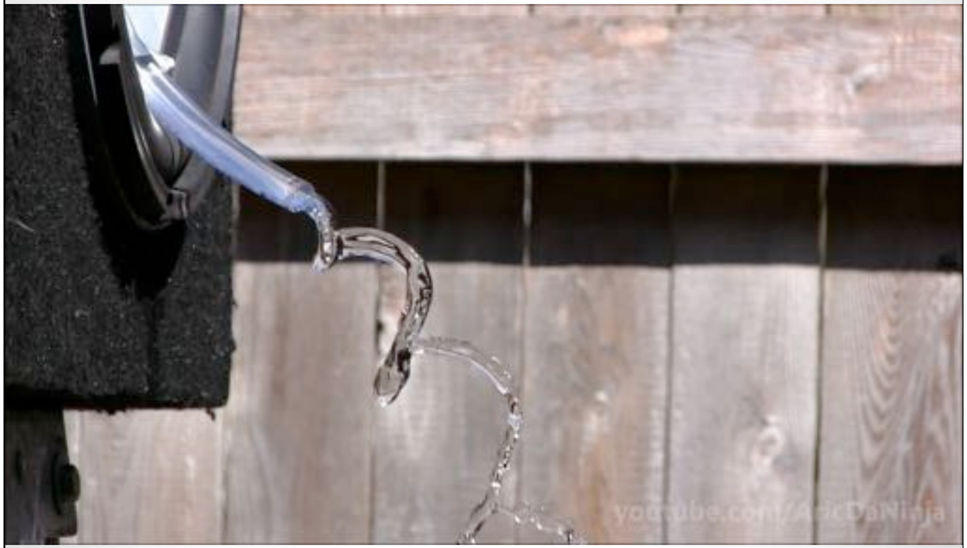
Stephen J. Guy

Oct 2, 2017

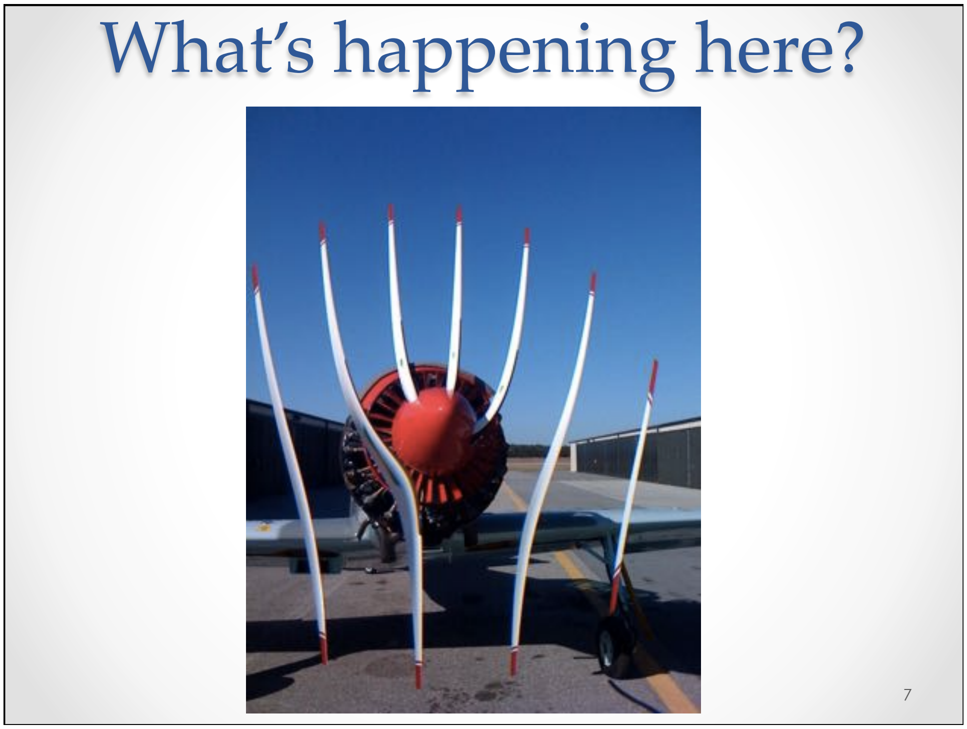
# Processing & Sampling Review

- What's a pixel?
  - A discrete sample of a continuous image
- What's a convolution?
  - A weighted average of a signal
- How to resample (e.g., resize/rotate)?
  - Weighted average of source pixels
- Issues without sampling?
  - Aliasing
- What does aliasing look like?
- When is dithering practical?

# What's happening here?

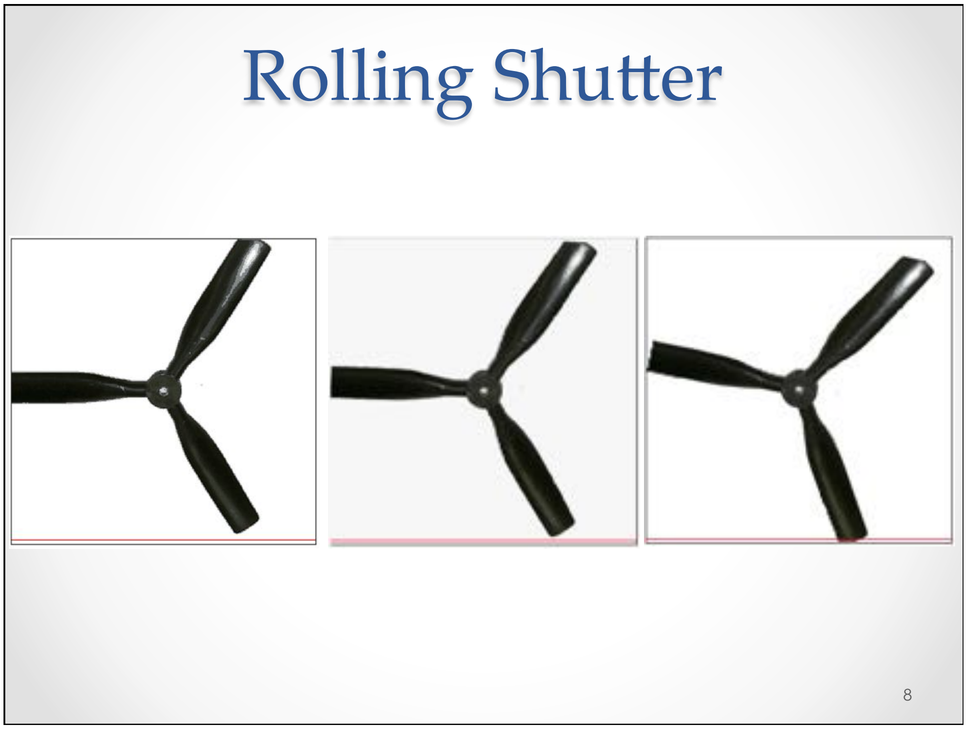


https://www.youtube.com/watch?v=UOfAmsQvgg4



## What's happening here?

7



## Rolling Shutter

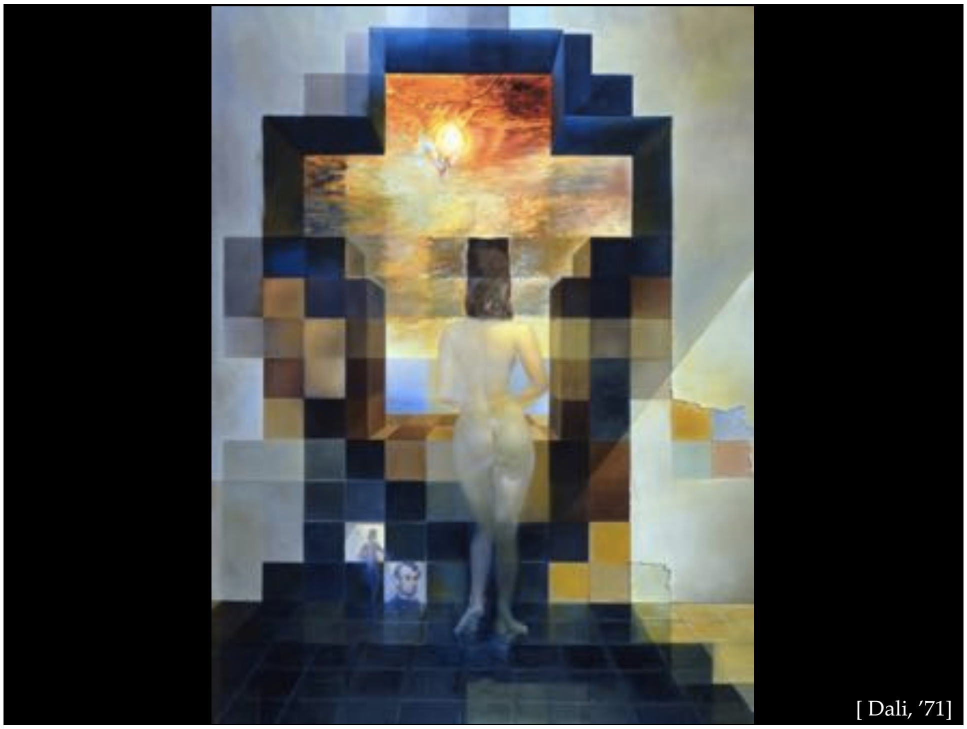
8

# Real-life Dithering

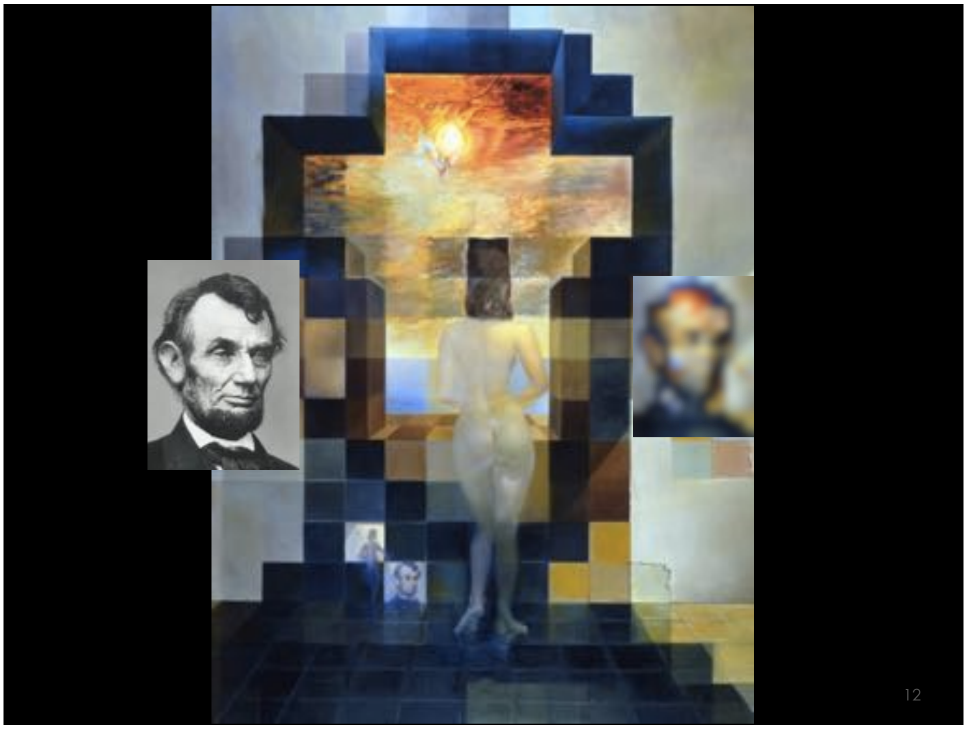
- GIFs have small color pallets

N
Netscape
Navigator 2.0
Copyright ©1995,1996
Netscape Communications Corporation.
All rights reserved.
MAKE GIFS AT GIFSOUP.COM

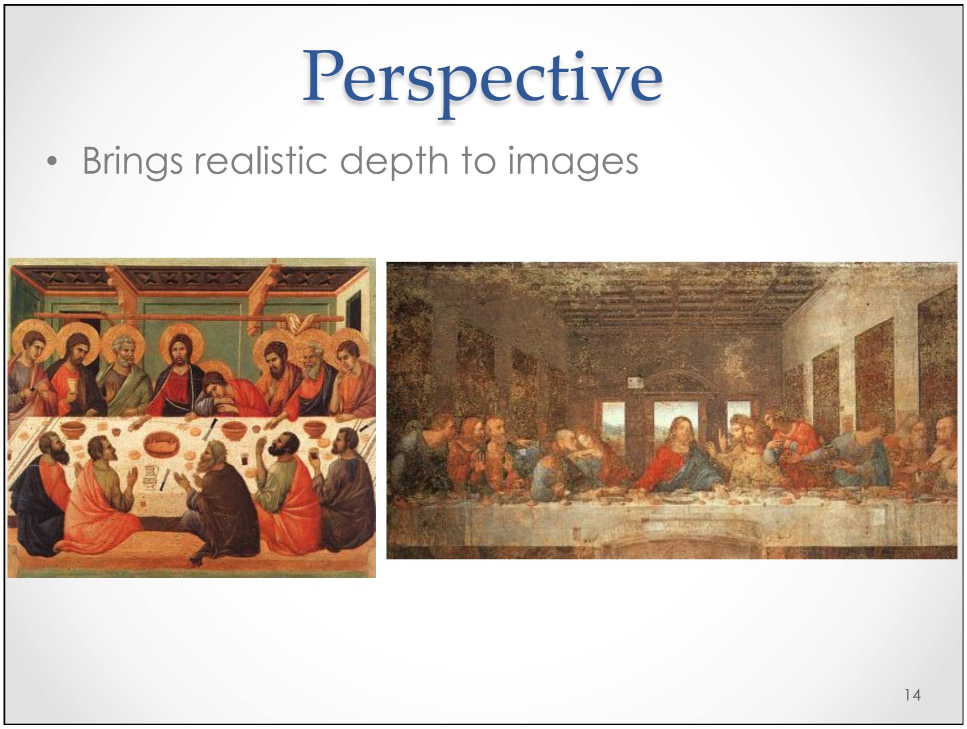
# Mark Rothko



[ Dali, ’71]



# Perspective

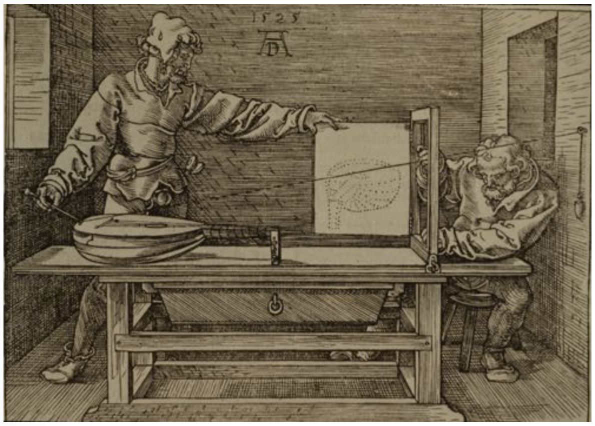


# Perspective

- Brings realistic depth to images

14

# Recall the Durër Woodcuts



- How would you make this an algorithm?
- What is a pixel?
- How to handle multiple objects?

# Two Approaches

- Object Centric vs “Pixel” Centric
  - Which is better?
  - What does physics do?

```
For each Object:
   Compute visible Pixels
   For all visible Pixels:
       If closer than previous object in pixel:
           Color pixel accordingly
```

```
For each Pixel:
   For all Objects:
       If Object intersects pixel ray
          If this is closer than previous intersection:
              Color pixel accordingly
```

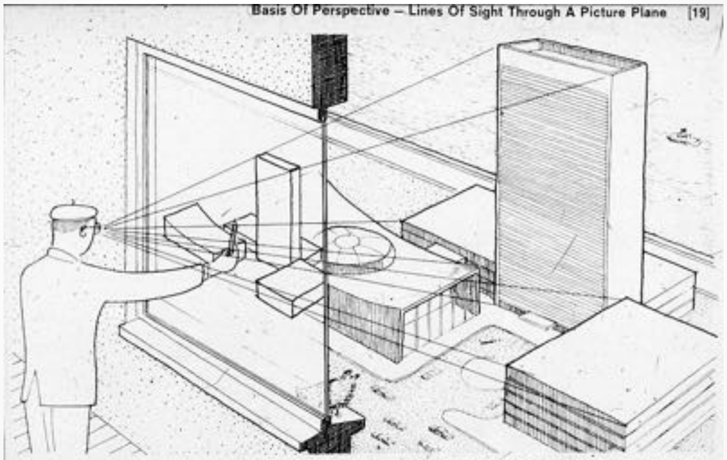
# Overview

- We'll start with pixel/sample-centric (~2 weeks)
  - Then move to object-centric (~3 weeks)
- Today
  - High-level concepts of perspective
  - Basic mathematical framework of ray tracing
- Wednesday
  - Algorithmic details

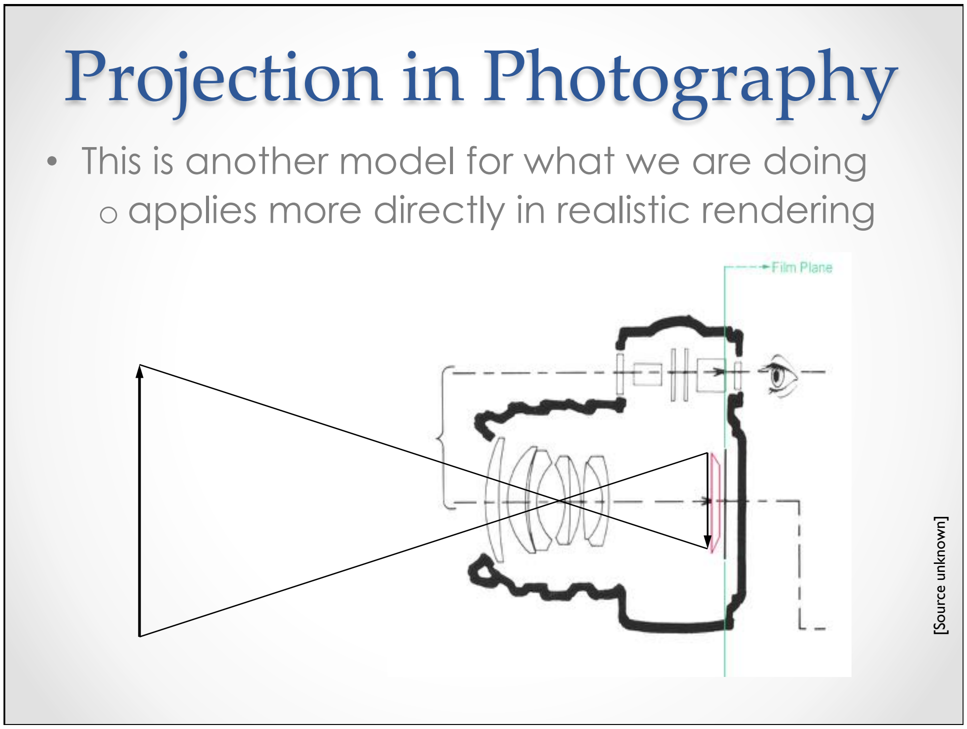
17

# Plane projection in drawing

Basis Of Perspective – Lines Of Sight Through A Picture Plane [19]



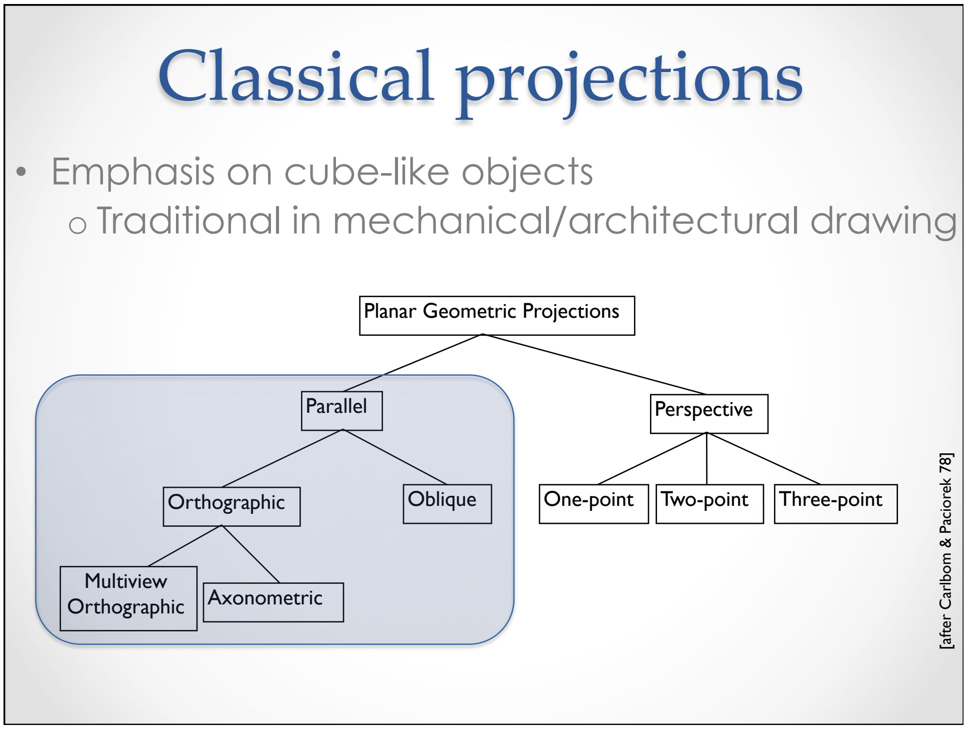
The concept of the picture plane may be better understood by looking through a window or other transparent plane from a fixed viewpoint. Your lines of sight, the multitude of straight lines leading from your eye to the subject, will all intersect this plane. Therefore, if you were to reach out with a grease pencil and draw the image of the subject on this plane you would be "tracing out" the infinite number of points of intersection of sight rays and plane. The result would be that you would have "transferred" a real three-dimensional object to a two-dimensional plane.



# Projection in Photography

- This is another model for what we are doing
  - applies more directly in realistic rendering

[Source unknown]



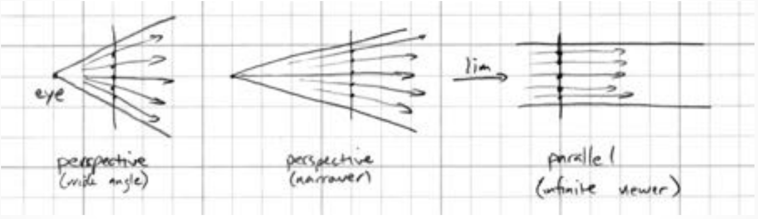
# Classical projections

- Emphasis on cube-like objects
  - Traditional in mechanical/architectural drawing

[after Carlbom & Paciorek 78]

# Parallel Projection

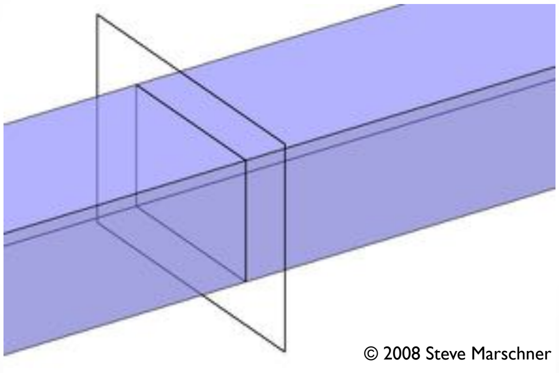
- Viewing rays are parallel rather than diverging
  - like a perspective camera that’s far away



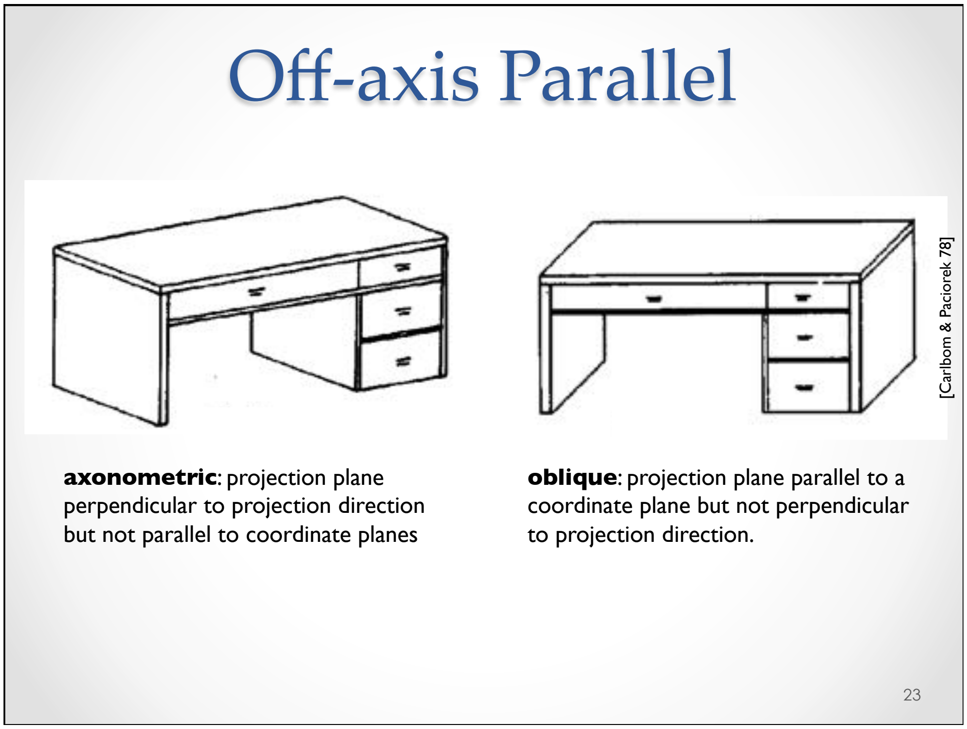
© 2008 Steve Marschner •

# View volume: Orthographic

- Bundle of rays are parallel



© 2008 Steve Marschner



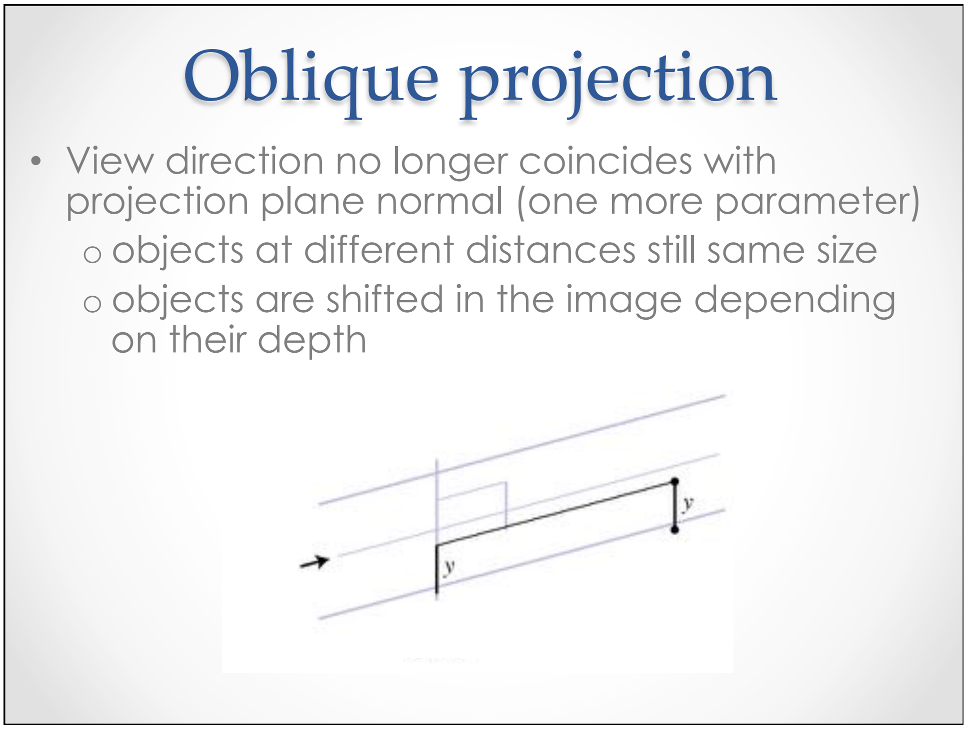
# Off-axis Parallel

[Carlbom & Paciorek 78]

**axonometric**: projection plane perpendicular to projection direction but not parallel to coordinate planes

**oblique**: projection plane parallel to a coordinate plane but not perpendicular to projection direction.

23

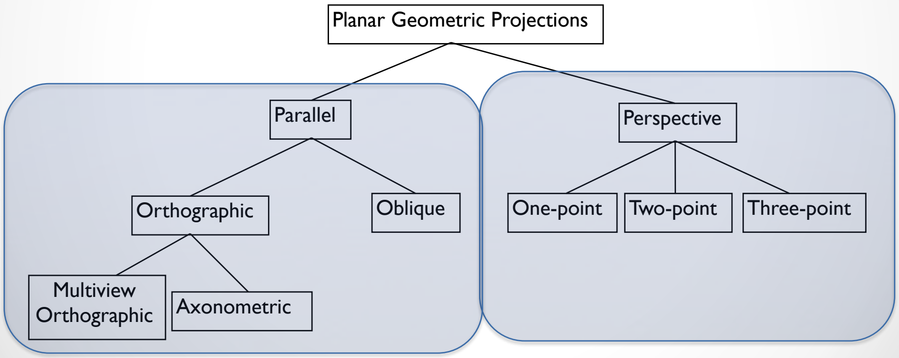


# Oblique projection

- View direction no longer coincides with projection plane normal (one more parameter)
  - o objects at different distances still same size
  - o objects are shifted in the image depending on their depth

# Classical projections

- Emphasis on cube-like objects
  - Traditional in mechanical/architectural drawing



Planar Geometric Projections

Parallel

Orthographic

Oblique

Multiview Orthographic

Axonometric

Perspective

One-point

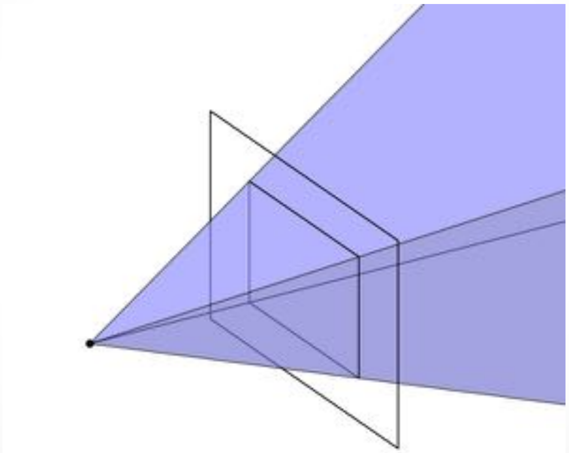
Two-point

Three-point

[after Carlbom & Paciorek 78]

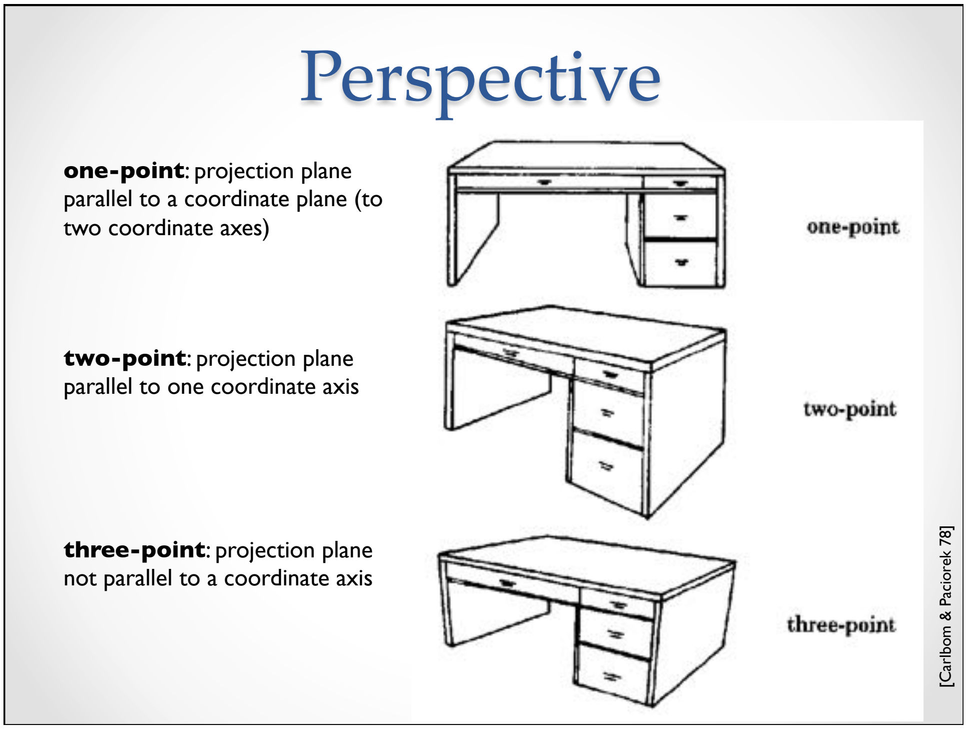
# View volume: Perspective

- Bundle of rays converge at the viewer



© 2008 Steve Marschner

26



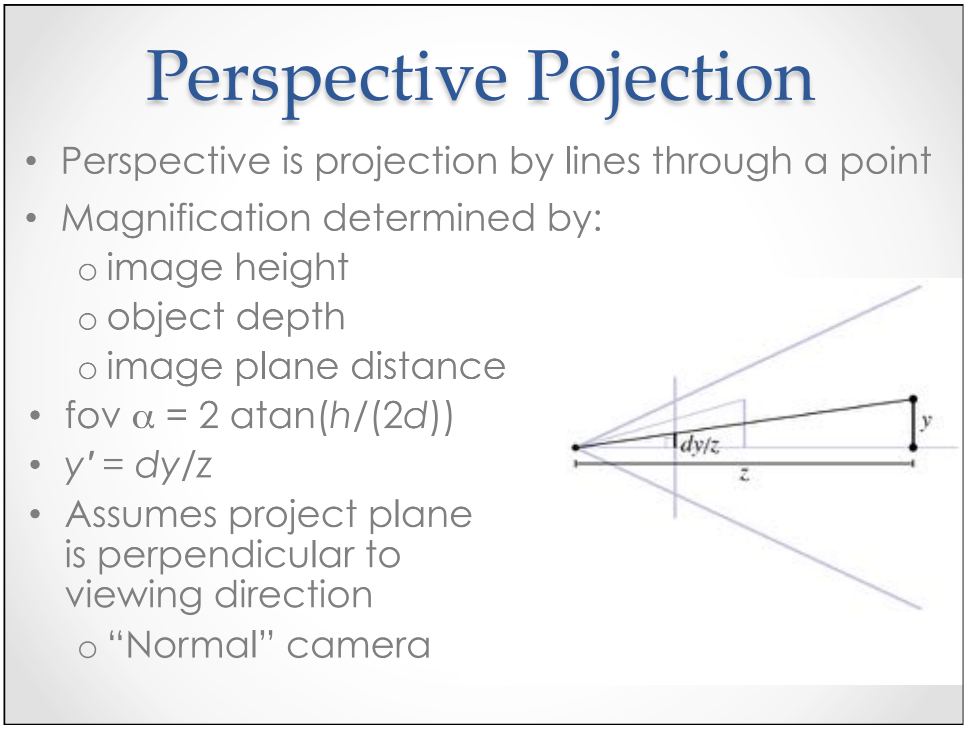
# Perspective

**one-point**: projection plane parallel to a coordinate plane (to two coordinate axes)

**two-point**: projection plane parallel to one coordinate axis

**three-point**: projection plane not parallel to a coordinate axis

[Carlbom & Paciorek 78]



# Perspective Pojection

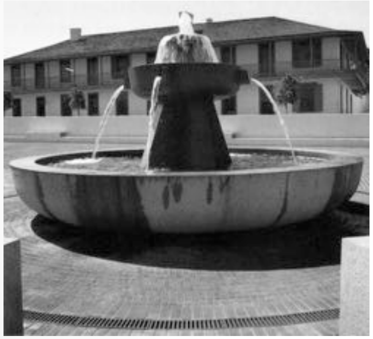
- Perspective is projection by lines through a point
- Magnification determined by:
  - image height
  - object depth
  - image plane distance
- fov $\alpha = 2\ \mathrm{atan}(h/(2d))$
- $y' = dy/z$
- Assumes project plane is perpendicular to viewing direction
  - “Normal” camera

# Field of view (or f.o.v.)

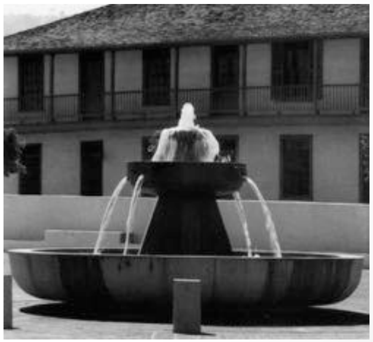
- Angle between rays corresponding to opposite edges of a perspective image
  - o easy to compute only for “normal” perspective
  - o different measures: vert., horiz., or diag.
- In cameras, determined by focal length
  - o confusing because of many image sizes
  - o for 35mm format (36mm by 24mm image)
    - 18mm = 67° v.f.o.v. – super-wide angle
    - 28mm = 46° v.f.o.v. – wide angle
    - 50mm = 27° v.f.o.v. – “normal”
    - 100mm = 14° v.f.o.v. – narrow angle (“telephoto”)

# Field of view

- Effects “strength” of perspective effects
  - o Real cameras wide lenses are difficult, specialty tools
  - o Graphics you can type any number … w/ odd effects!

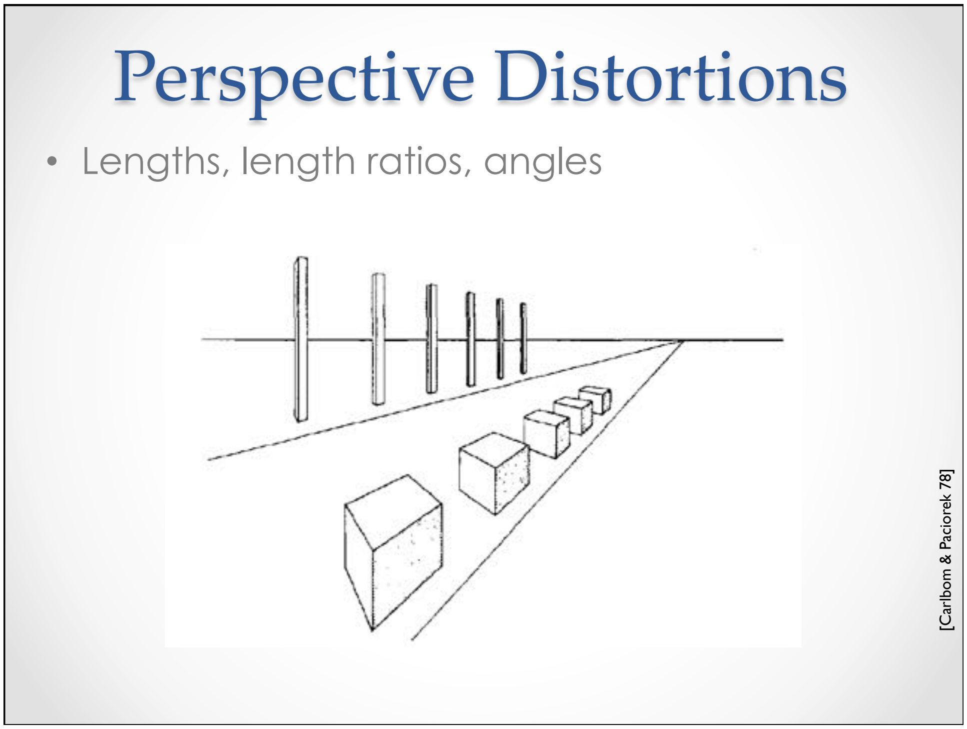


close viewpoint
wide angle
prominent foreshortening



far viewpoint
narrow angle
little foreshortening

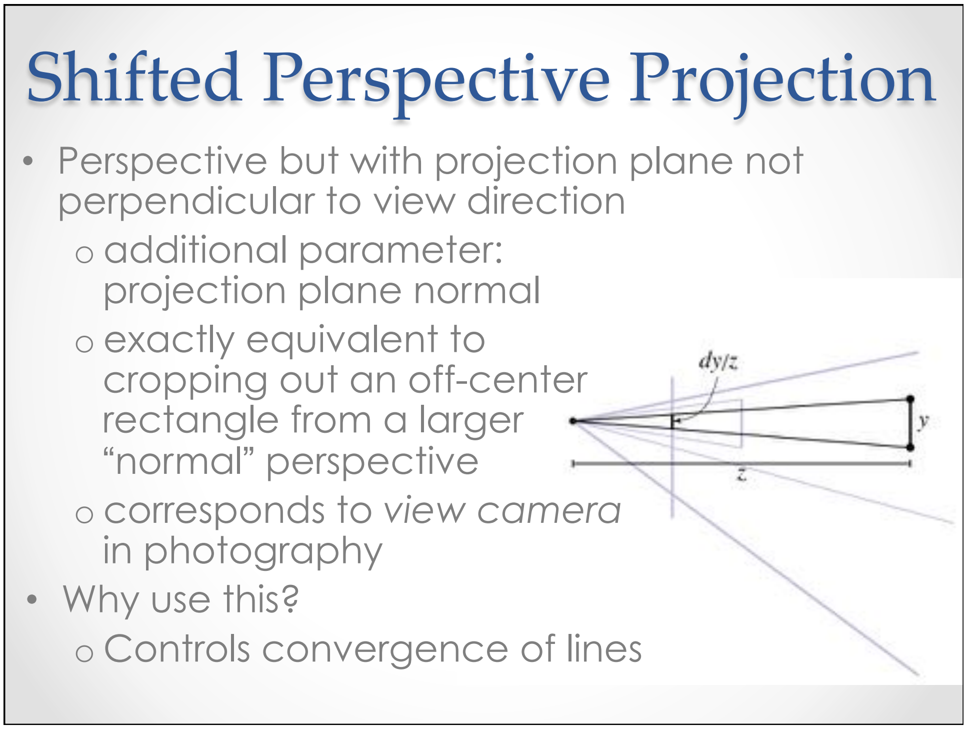
[Ansel Adams]



# Perspective Distortions

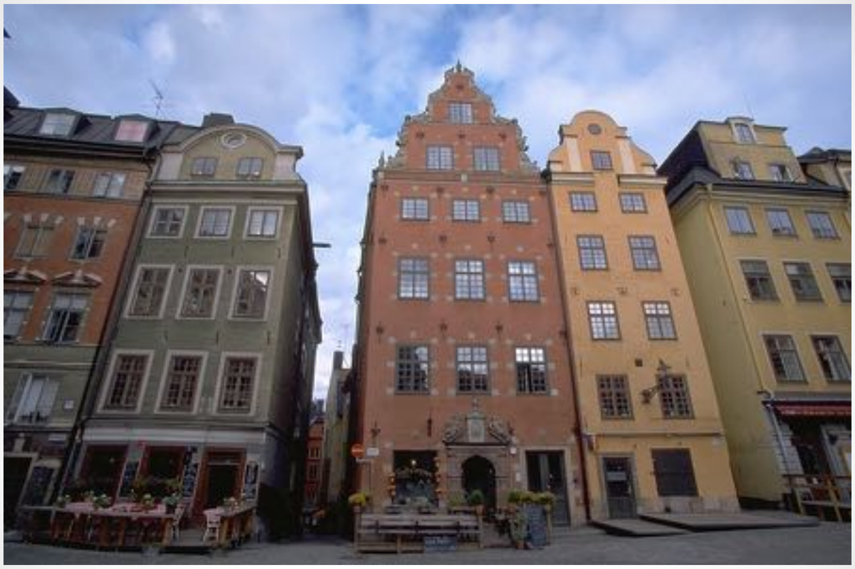
- Lengths, length ratios, angles

[Carlbom & Paciorek 78]



# Shifted Perspective Projection

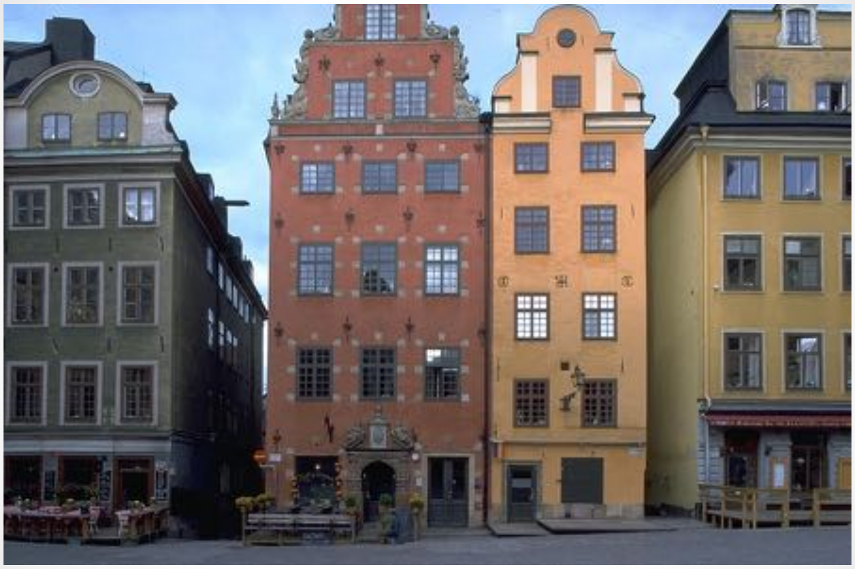
- Perspective but with projection plane not perpendicular to view direction
  - o additional parameter: projection plane normal
  - o exactly equivalent to cropping out an off-center rectangle from a larger “normal” perspective
  - o corresponds to *view camera* in photography
- Why use this?
  - o Controls convergence of lines



[Philip Greenspun]

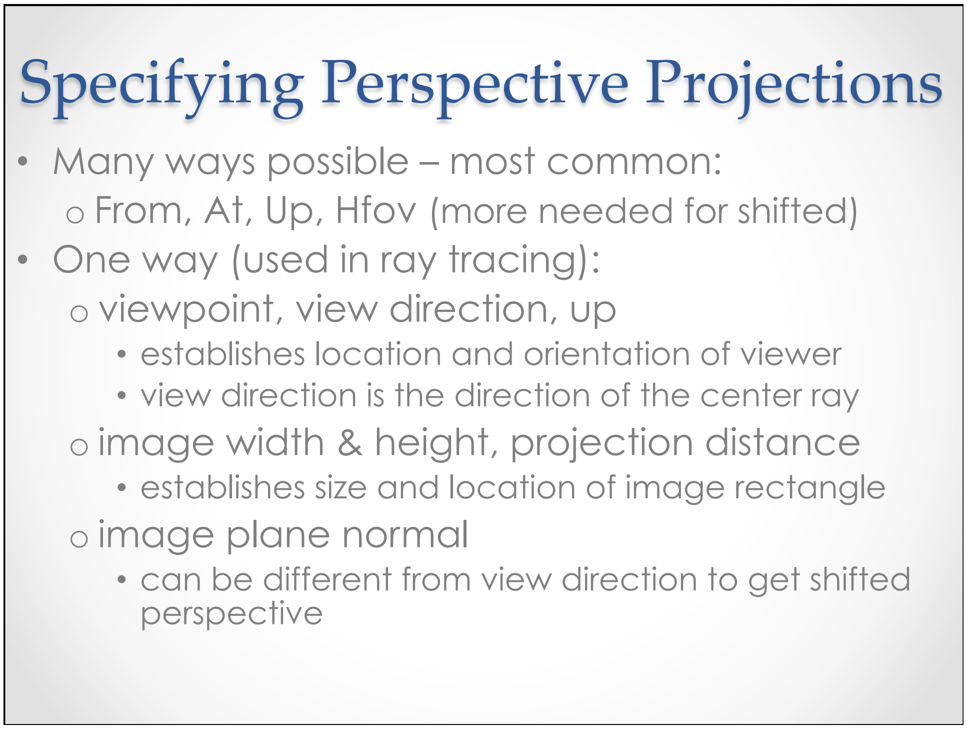
camera tilted up: converging vertical lines

33



[Philip Greenspun]

lens shifted up: parallel vertical lines



# Specifying Perspective Projections

- Many ways possible – most common:
  - o From, At, Up, Hfov (more needed for shifted)
- One way (used in ray tracing):
  - o viewpoint, view direction, up
    - establishes location and orientation of viewer
    - view direction is the direction of the center ray
  - o image width & height, projection distance
    - establishes size and location of image rectangle
  - o image plane normal
    - can be different from view direction to get shifted perspective

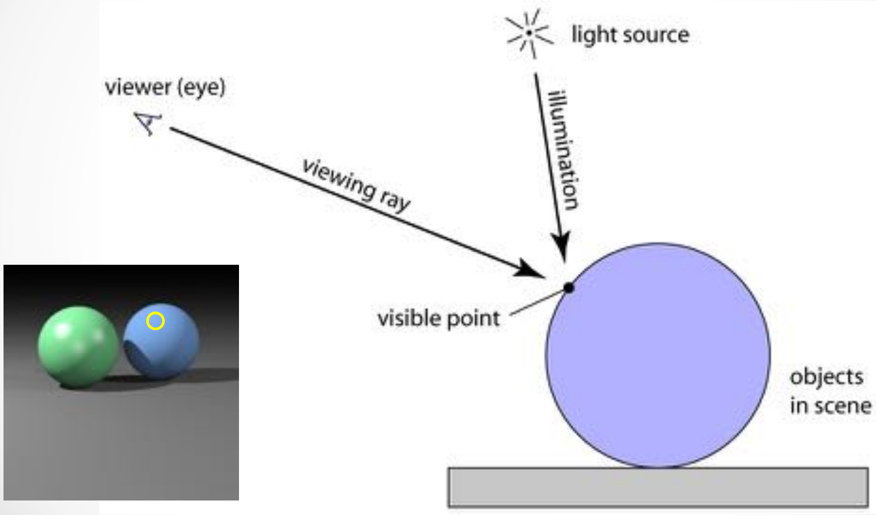
# Ray Tracing

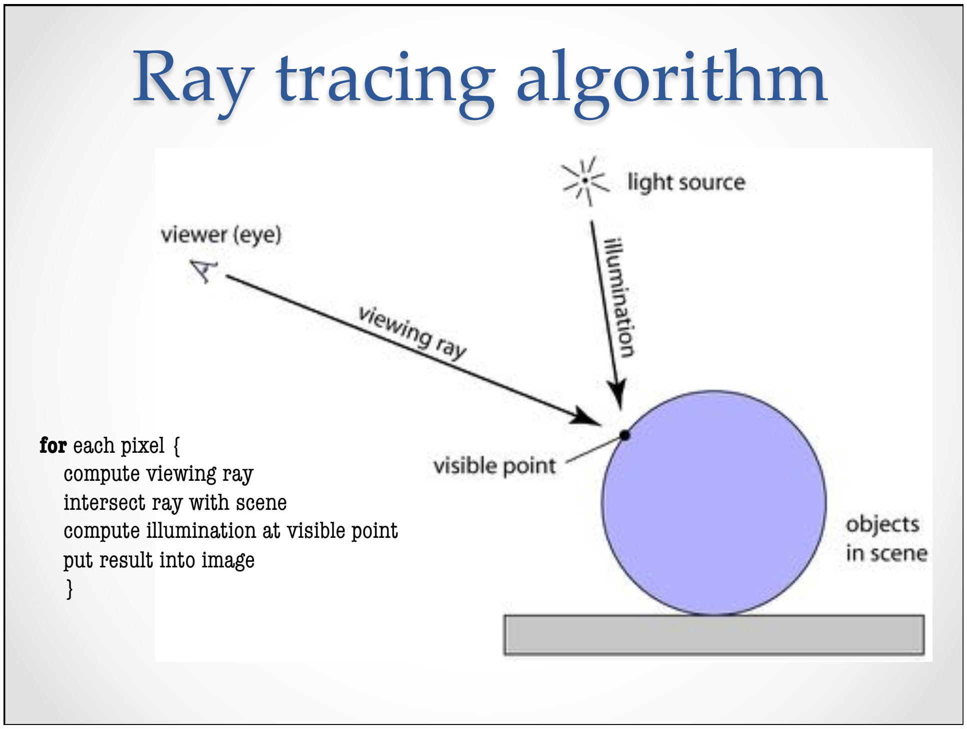
36

# Ray Tracing Lectures

- Today
  - Introduction & Basic Idea
  - Just enough to get you started thinking about how to approach HW2
- Wednesday
  - Equations & Algorithms
  - Everything you need to finish HW2
- Next Week
  - Advanced lighting models
  - Role of recursion
  - Acceleration data structures
  - Everything needed to finish HW3

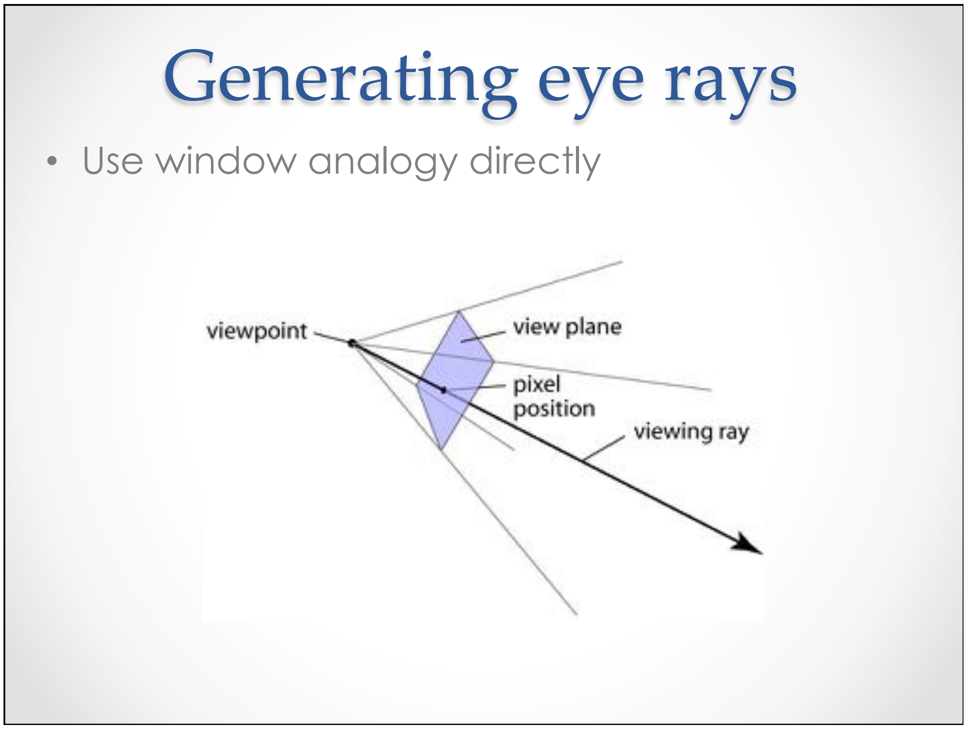
# Ray tracing idea





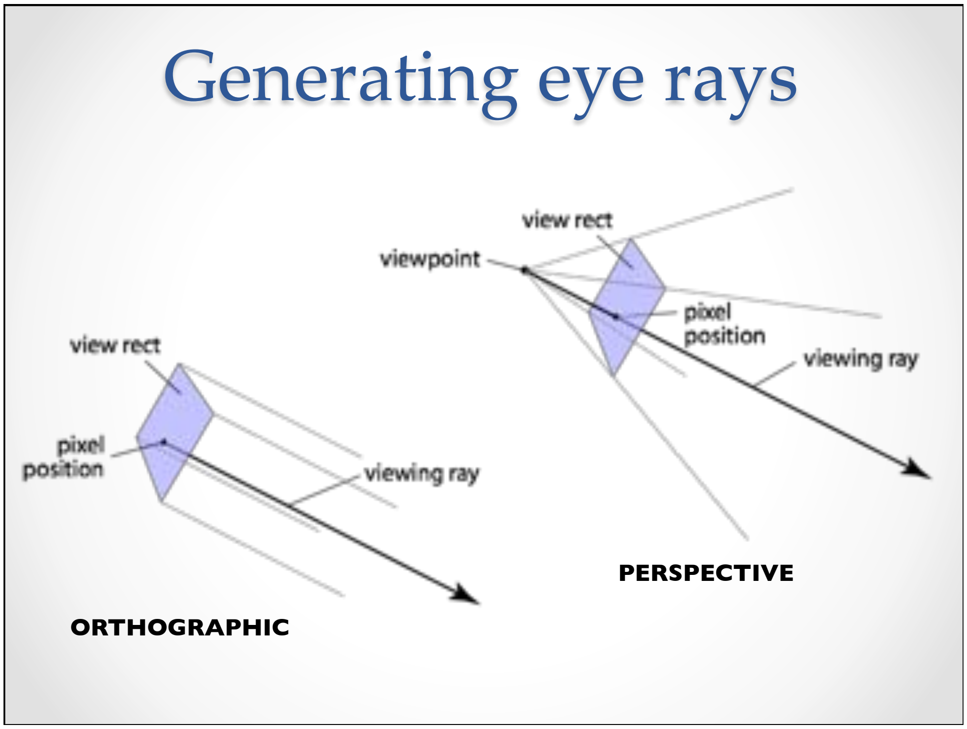
# Ray tracing algorithm

```
for each pixel {
   compute viewing ray
   intersect ray with scene
   compute illumination at visible point
   put result into image
   }
```

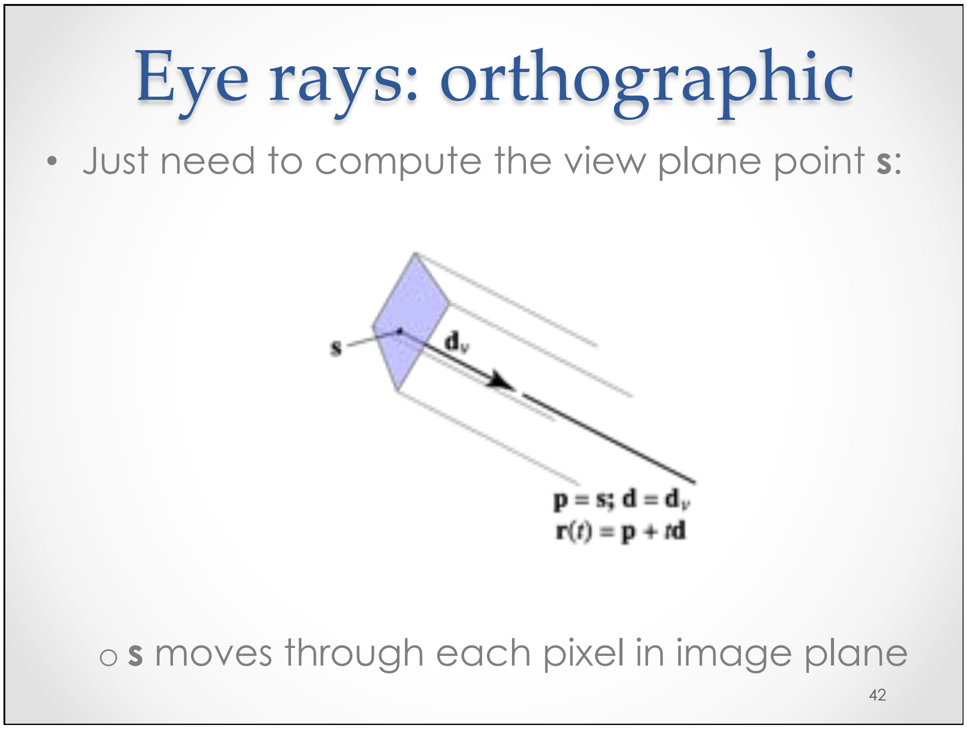


# Generating eye rays

- Use window analogy directly



# Generating eye rays

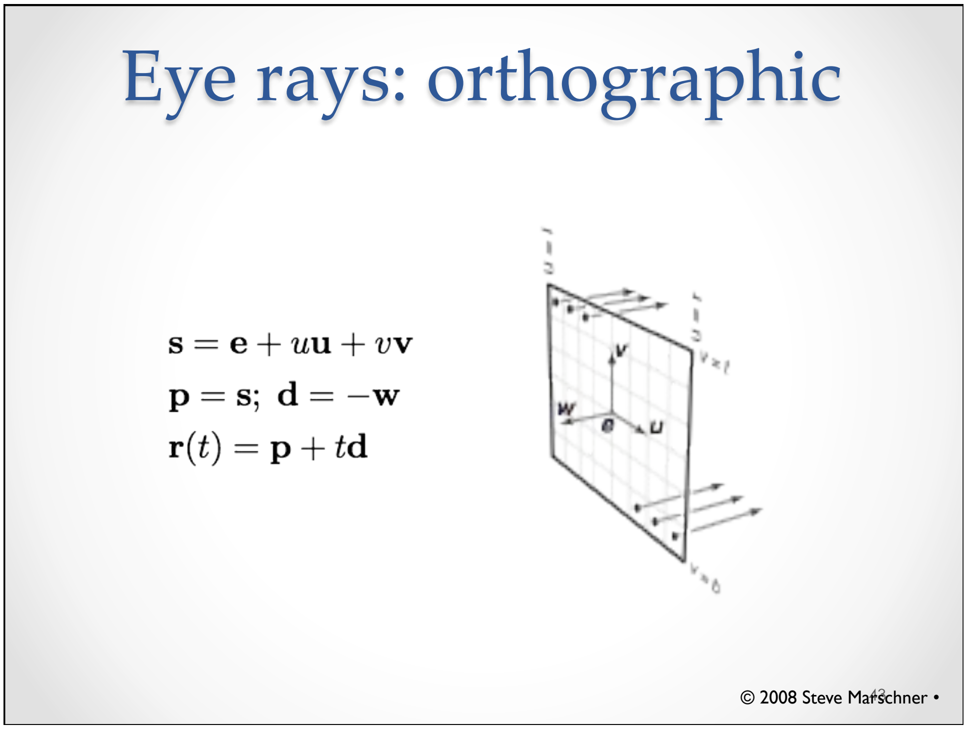


# Eye rays: orthographic

- Just need to compute the view plane point **s**:

s
$\mathbf{d}_v$
$\mathbf{p} = \mathbf{s};\ \mathbf{d} = \mathbf{d}_v$
$\mathbf{r}(t) = \mathbf{p} + t\mathbf{d}$

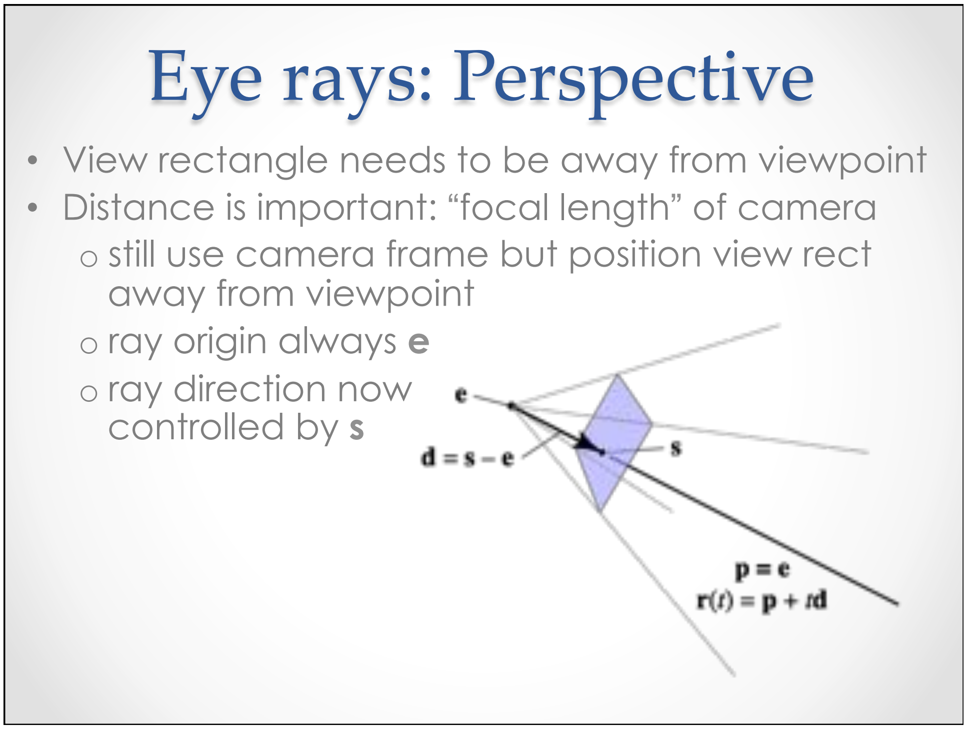
- **s** moves through each pixel in image plane



# Eye rays: orthographic

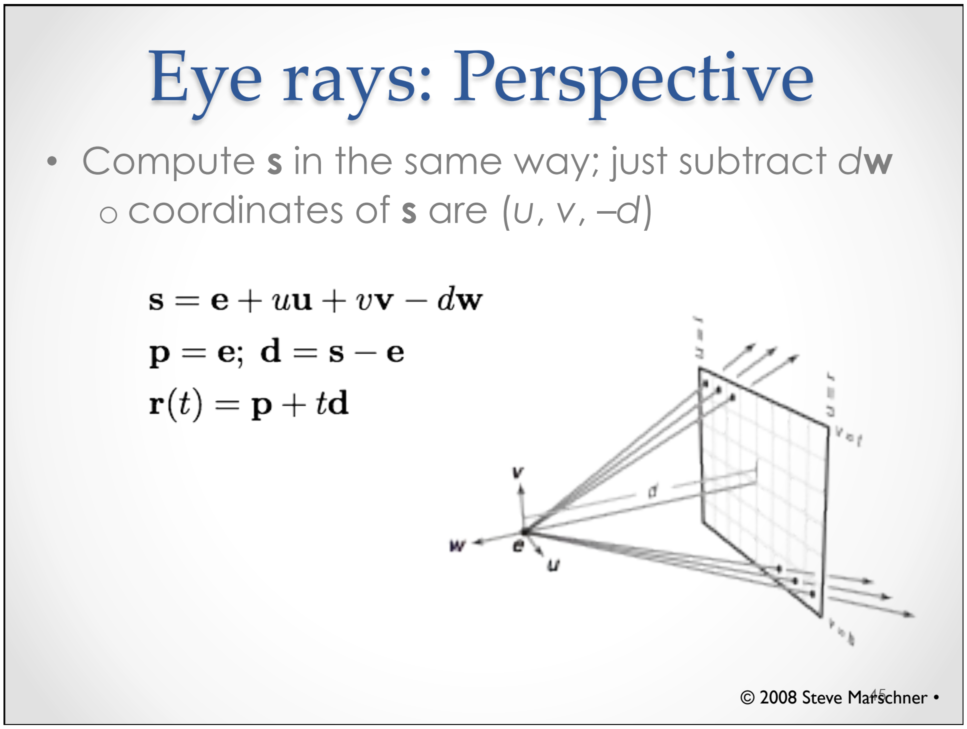
$$\mathbf{s} = \mathbf{e} + u\mathbf{u} + v\mathbf{v}$$
$$\mathbf{p} = \mathbf{s};\ \mathbf{d} = -\mathbf{w}$$
$$\mathbf{r}(t) = \mathbf{p} + t\mathbf{d}$$

© 2008 Steve Marschner • 43



# Eye rays: Perspective

- View rectangle needs to be away from viewpoint
- Distance is important: "focal length" of camera
  - o still use camera frame but position view rect away from viewpoint
  - o ray origin always **e**
  - o ray direction now controlled by **s**

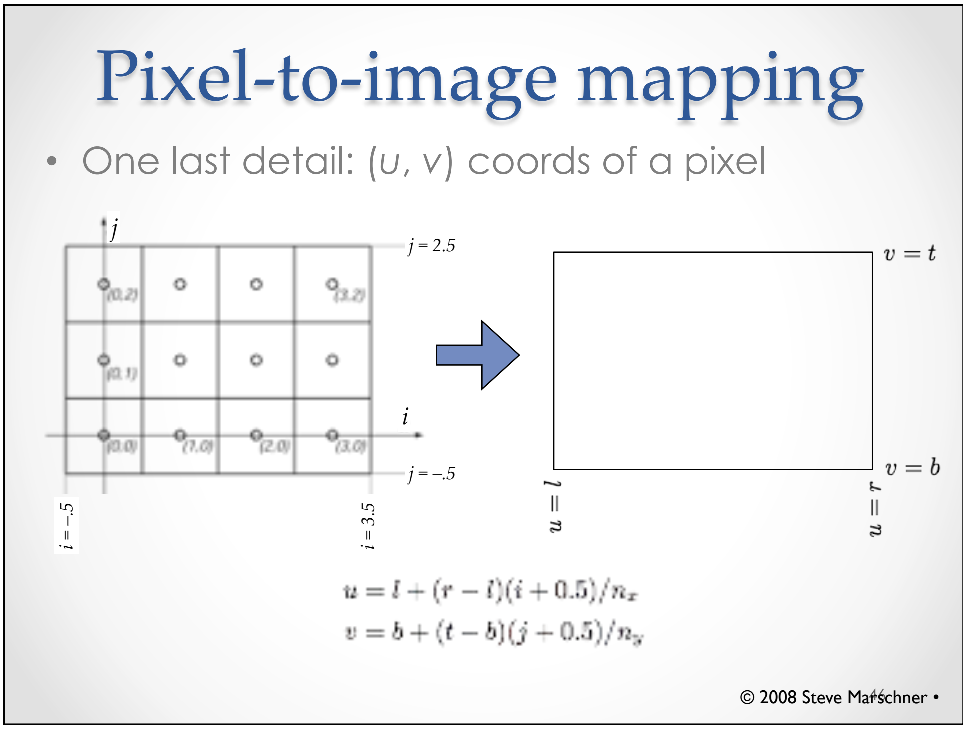


# Eye rays: Perspective

- Compute **s** in the same way; just subtract *d***w**
  - coordinates of **s** are (*u*, *v*, –*d*)

$$\mathbf{s} = \mathbf{e} + u\mathbf{u} + v\mathbf{v} - d\mathbf{w}$$
$$\mathbf{p} = \mathbf{e};\ \mathbf{d} = \mathbf{s} - \mathbf{e}$$
$$\mathbf{r}(t) = \mathbf{p} + t\mathbf{d}$$

© 2008 Steve Marschner • 45

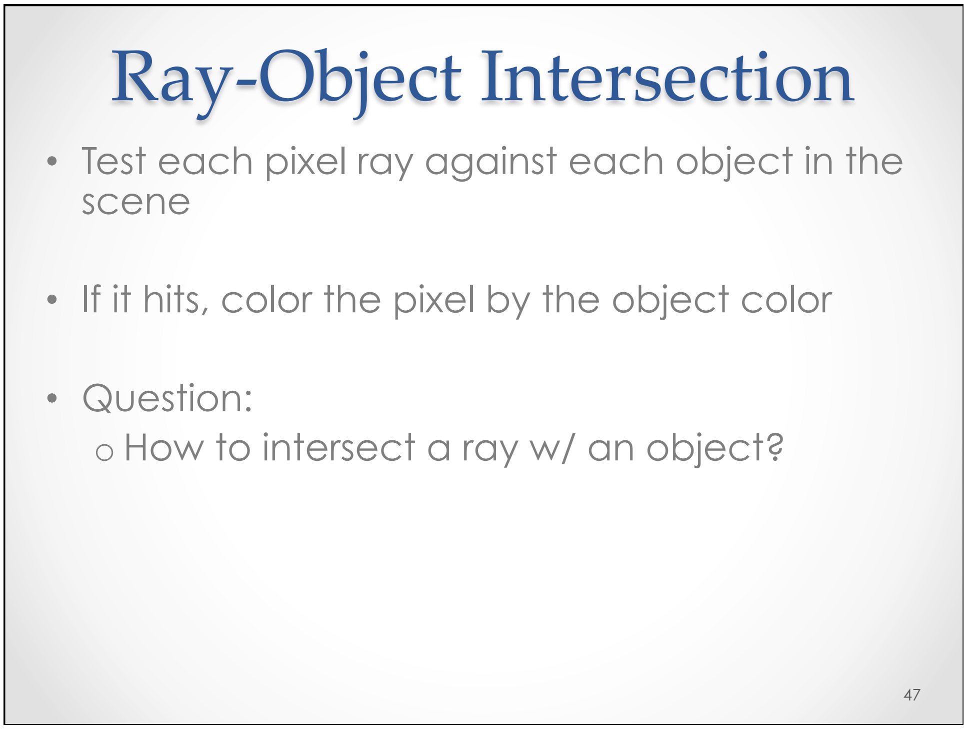


# Pixel-to-image mapping

- One last detail: (*u*, *v*) coords of a pixel

$$u = l + (r - l)(i + 0.5)/n_x$$
$$v = b + (t - b)(j + 0.5)/n_y$$

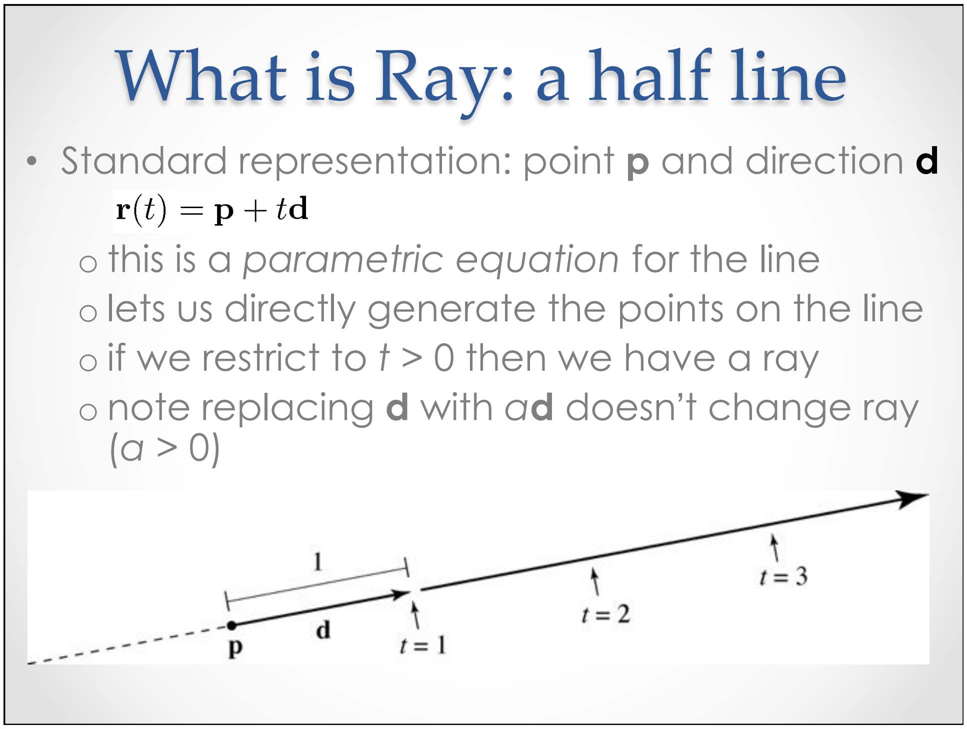
© 2008 Steve Marschner • 46



# Ray-Object Intersection

- Test each pixel ray against each object in the scene
- If it hits, color the pixel by the object color
- Question:
  - How to intersect a ray w/ an object?

47

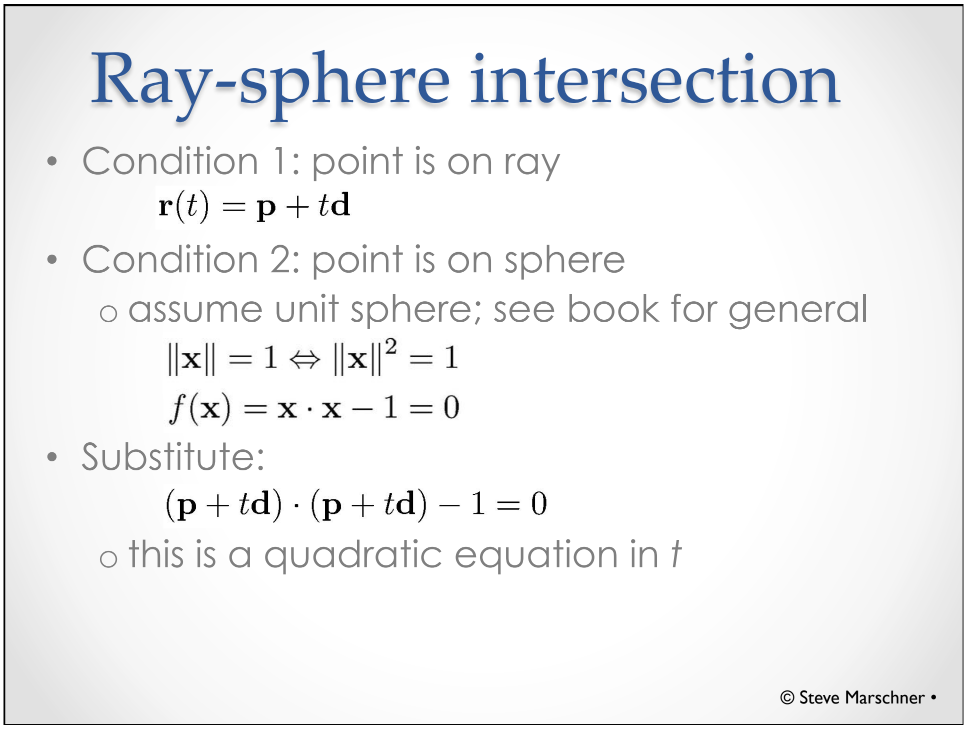


# What is Ray: a half line

- Standard representation: point **p** and direction **d**

  $\mathbf{r}(t) = \mathbf{p} + t\mathbf{d}$

  - this is a *parametric equation* for the line
  - lets us directly generate the points on the line
  - if we restrict to *t* > 0 then we have a ray
  - note replacing **d** with *a***d** doesn't change ray (*a* > 0)



# Ray-sphere intersection

- Condition 1: point is on ray

  $$\mathbf{r}(t) = \mathbf{p} + t\mathbf{d}$$

- Condition 2: point is on sphere
  - assume unit sphere; see book for general

    $$\|\mathbf{x}\| = 1 \Leftrightarrow \|\mathbf{x}\|^2 = 1$$

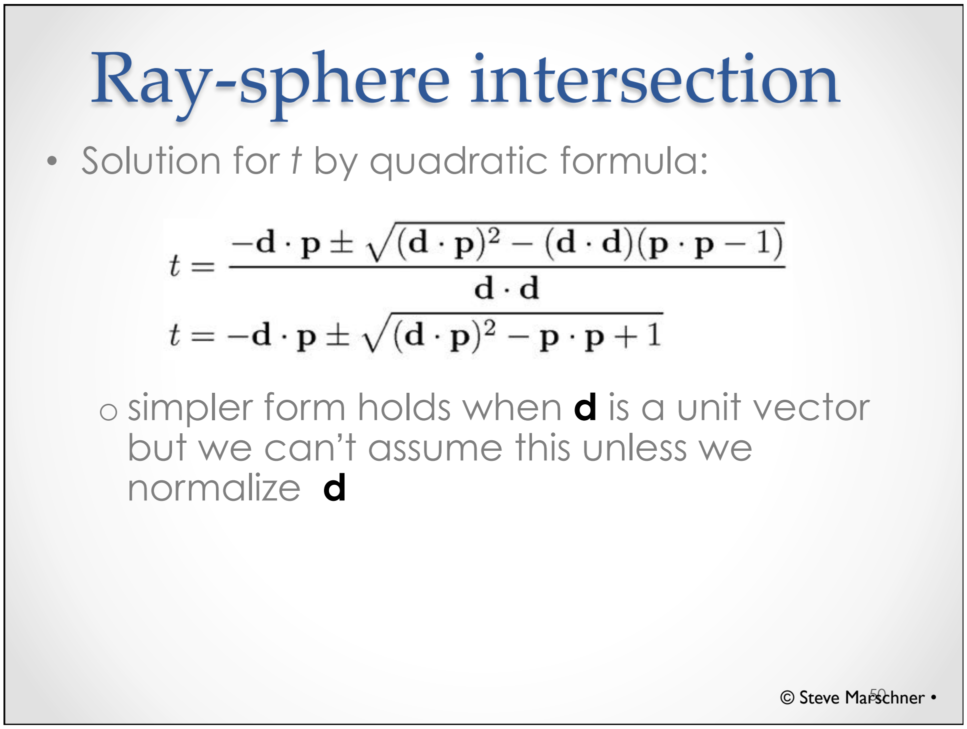
    $$f(\mathbf{x}) = \mathbf{x} \cdot \mathbf{x} - 1 = 0$$

- Substitute:

  $$(\mathbf{p} + t\mathbf{d}) \cdot (\mathbf{p} + t\mathbf{d}) - 1 = 0$$

  - this is a quadratic equation in *t*

© Steve Marschner •



# Ray-sphere intersection

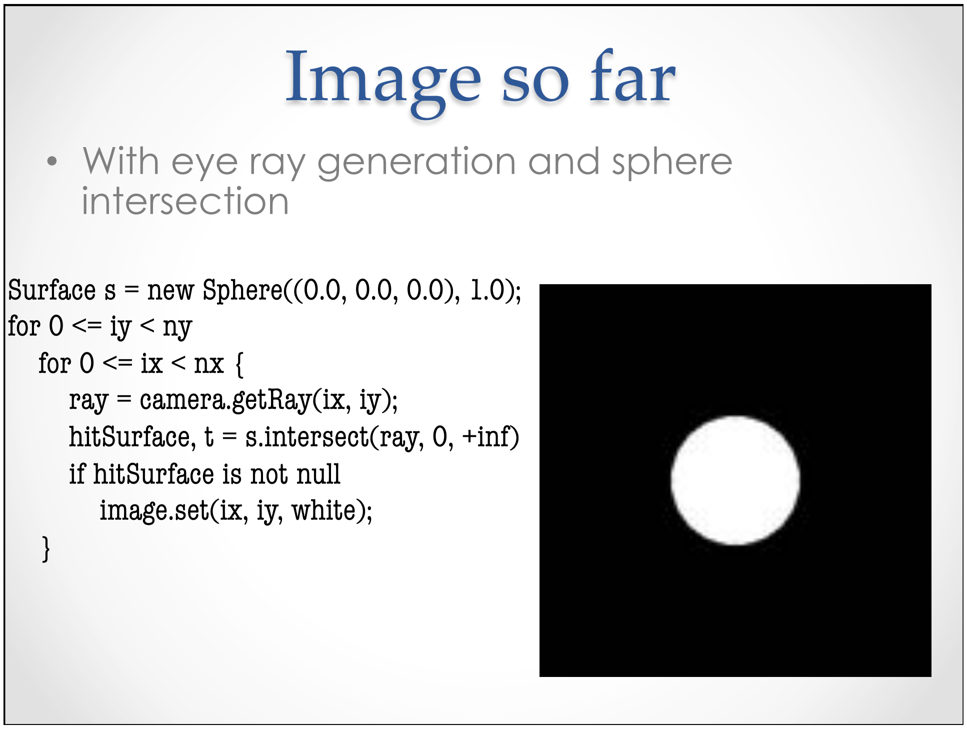
- Solution for *t* by quadratic formula:

  $$t = \frac{-\mathbf{d} \cdot \mathbf{p} \pm \sqrt{(\mathbf{d} \cdot \mathbf{p})^2 - (\mathbf{d} \cdot \mathbf{d})(\mathbf{p} \cdot \mathbf{p} - 1)}}{\mathbf{d} \cdot \mathbf{d}}$$

  $$t = -\mathbf{d} \cdot \mathbf{p} \pm \sqrt{(\mathbf{d} \cdot \mathbf{p})^2 - \mathbf{p} \cdot \mathbf{p} + 1}$$

  - simpler form holds when **d** is a unit vector but we can't assume this unless we normalize **d**

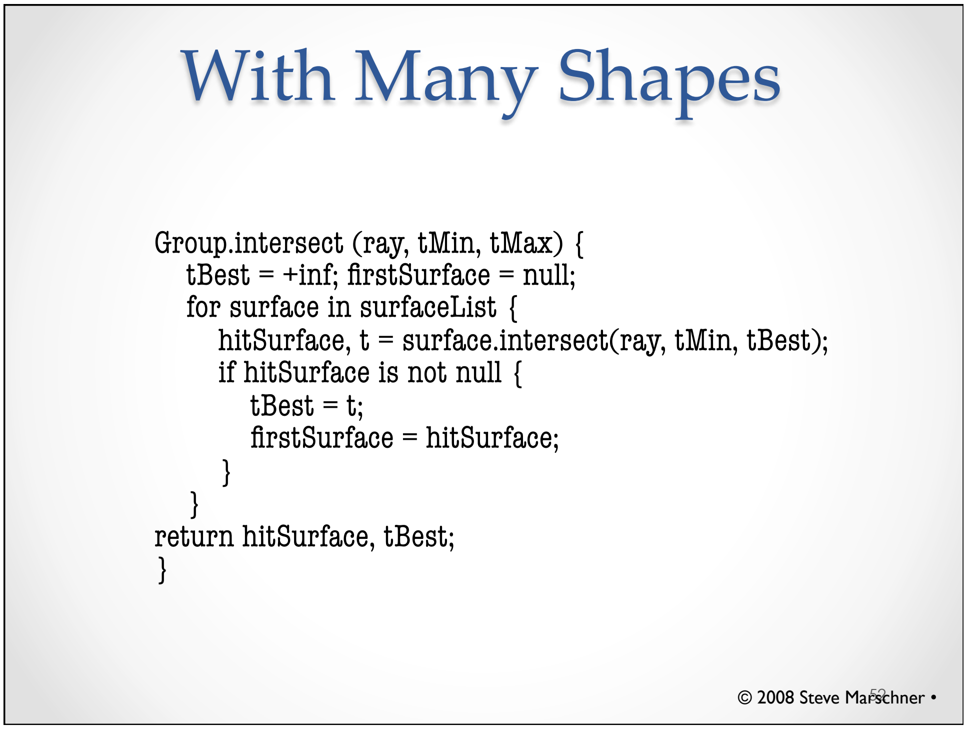
© Steve Marschner • 50



# Image so far

- With eye ray generation and sphere intersection

```
Surface s = new Sphere((0.0, 0.0, 0.0), 1.0);
for 0 <= iy < ny
   for 0 <= ix < nx {
      ray = camera.getRay(ix, iy);
      hitSurface, t = s.intersect(ray, 0, +inf)
      if hitSurface is not null
         image.set(ix, iy, white);
   }
```



# With Many Shapes

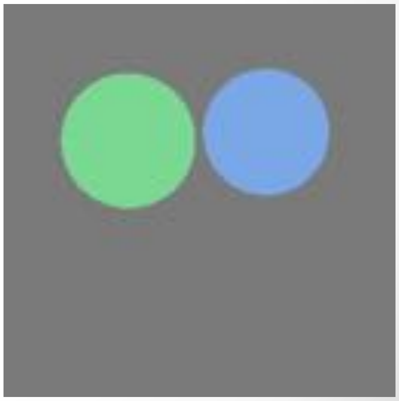
```
Group.intersect (ray, tMin, tMax) {
   tBest = +inf; firstSurface = null;
   for surface in surfaceList {
      hitSurface, t = surface.intersect(ray, tMin, tBest);
      if hitSurface is not null {
         tBest = t;
         firstSurface = hitSurface;
      }
   }
return hitSurface, tBest;
}
```

© 2008 Steve Marschner • 59

# Image so far

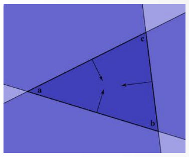
- With eye ray generation and scene intersection

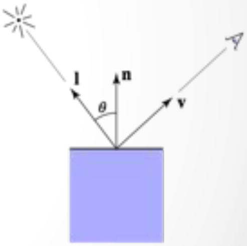
```
for 0 <= iy < ny
   for 0 <= ix < nx {
      ray = camera.getRay(ix, iy);
      c = scene.trace(ray, 0, +inf);
      image.set(ix, iy, c);
   }

...

Scene.trace(ray, tMin, tMax) {
   surface, t = surfs.intersect(ray, tMin, tMax);
   if (surface != null) return surface.color();
   else return black;
}
```



# Advancements

- Support more shapes
  - Ray-plane intersection
  - Ray-triangle intersection
- Lighting models (shading)
  - How much light gets reflected to the camera?
  - Mirrored/glossy materials
  - Shadows

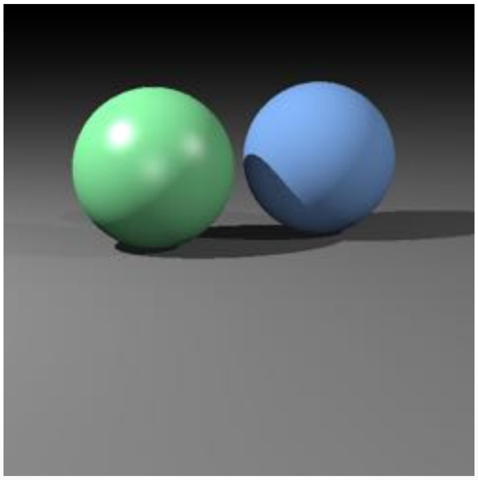




54

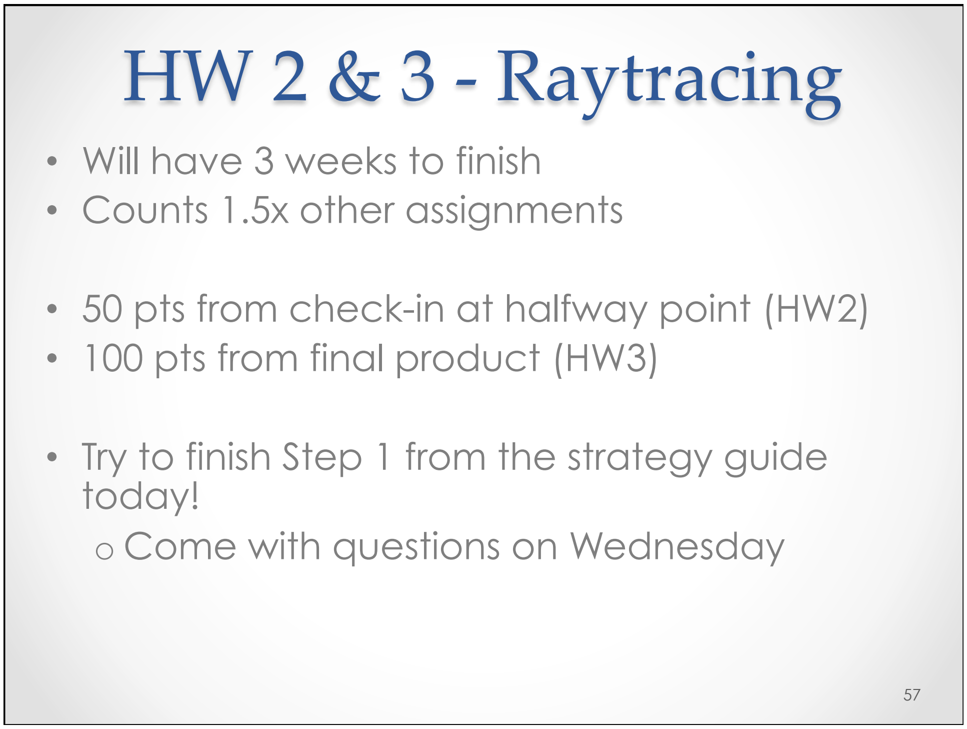
# Next Time

- Phong Shading w/ Shadows



# Announcements

- Assignment 1 grades are posted
- Assignment 2 should be online – Raytracing!
- One of the most rewarding & frustrating projects in all of CS
  - o Split into two parts over ~3 weeks
  - o First part due on Wednesday 10/11
  - o 2nd part due on Monday 10/23
  - o Midterm Week of 10/29

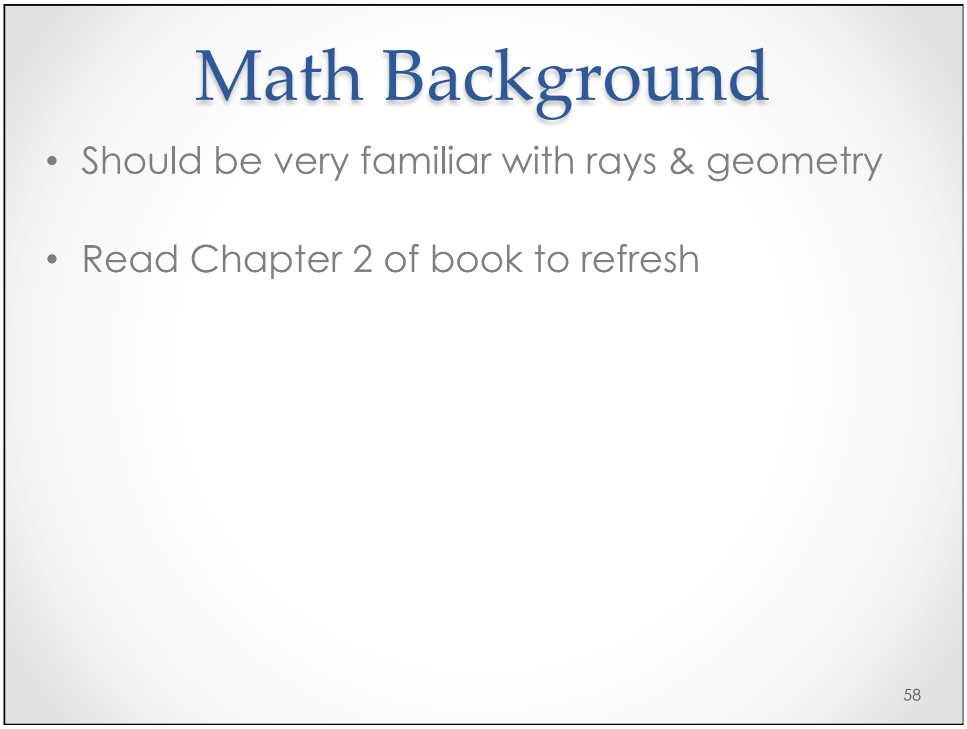


# HW 2 & 3 - Raytracing

- Will have 3 weeks to finish
- Counts 1.5x other assignments

- 50 pts from check-in at halfway point (HW2)
- 100 pts from final product (HW3)

- Try to finish Step 1 from the strategy guide today!
  - Come with questions on Wednesday



# Math Background

- Should be very familiar with rays & geometry

- Read Chapter 2 of book to refresh

# HW 0 is Graded

- Let me know if there is any issue
  - I'll share some of the art entries soon

- H1 submissions are looking cool:

Jack Stanek