

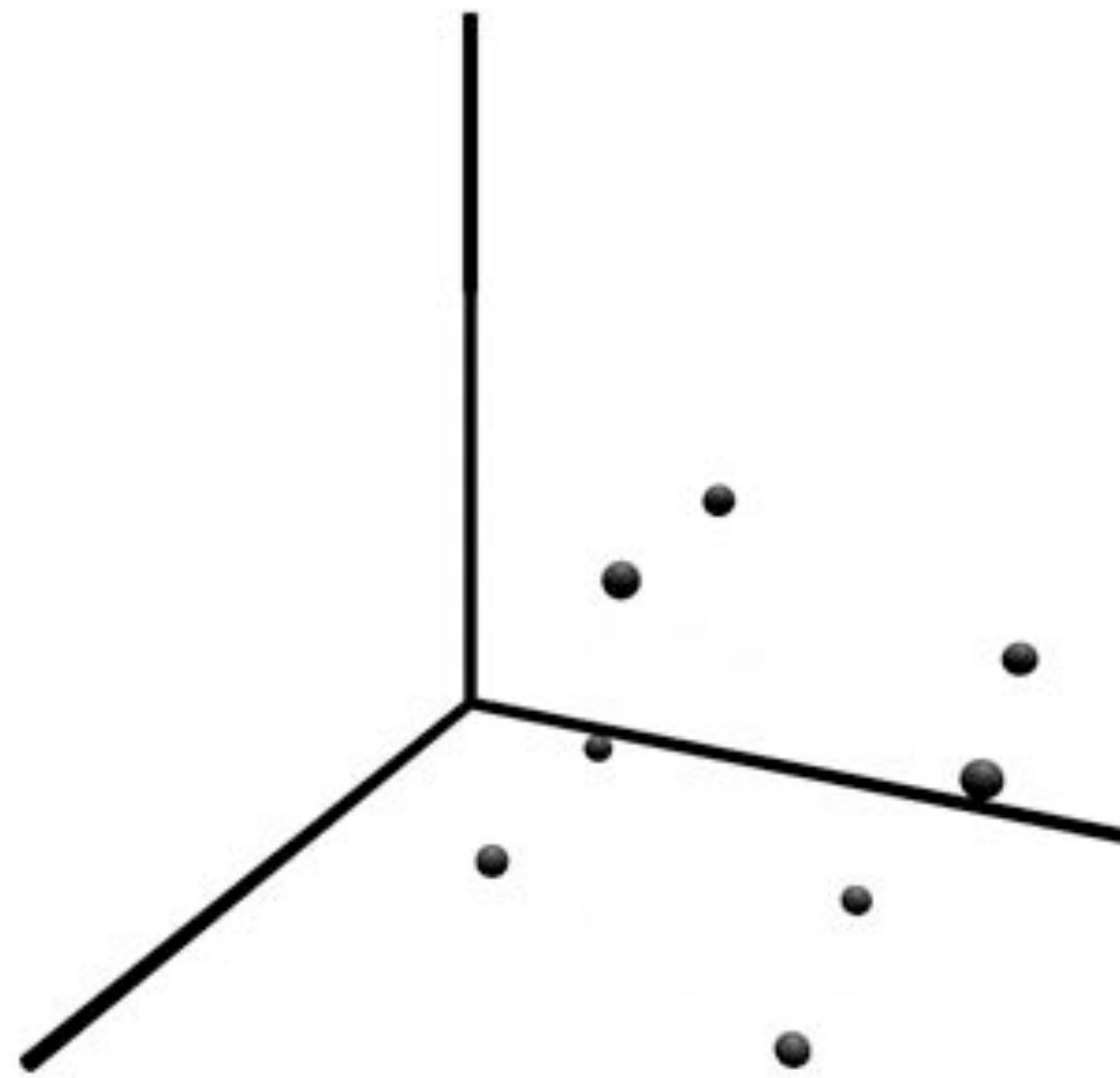
# **Spatial Transformations**

---

**Computer Graphics**  
**CMU 15-462/15-662**

# Spatial Transformation

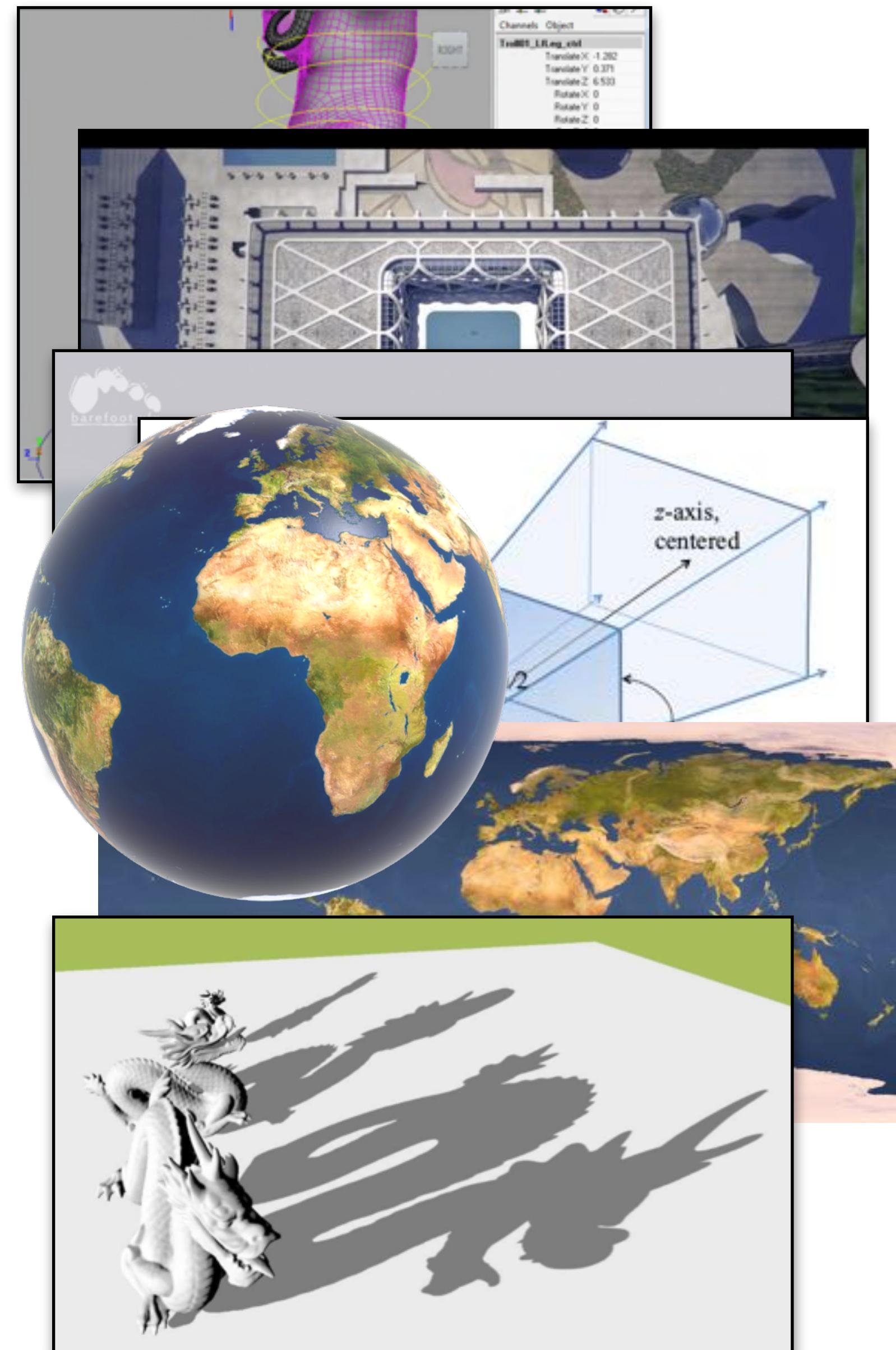
- Basically any function that assigns each point a new location
- Today we'll focus on common transformations of space (rotation, scaling, etc.) encoded by linear maps



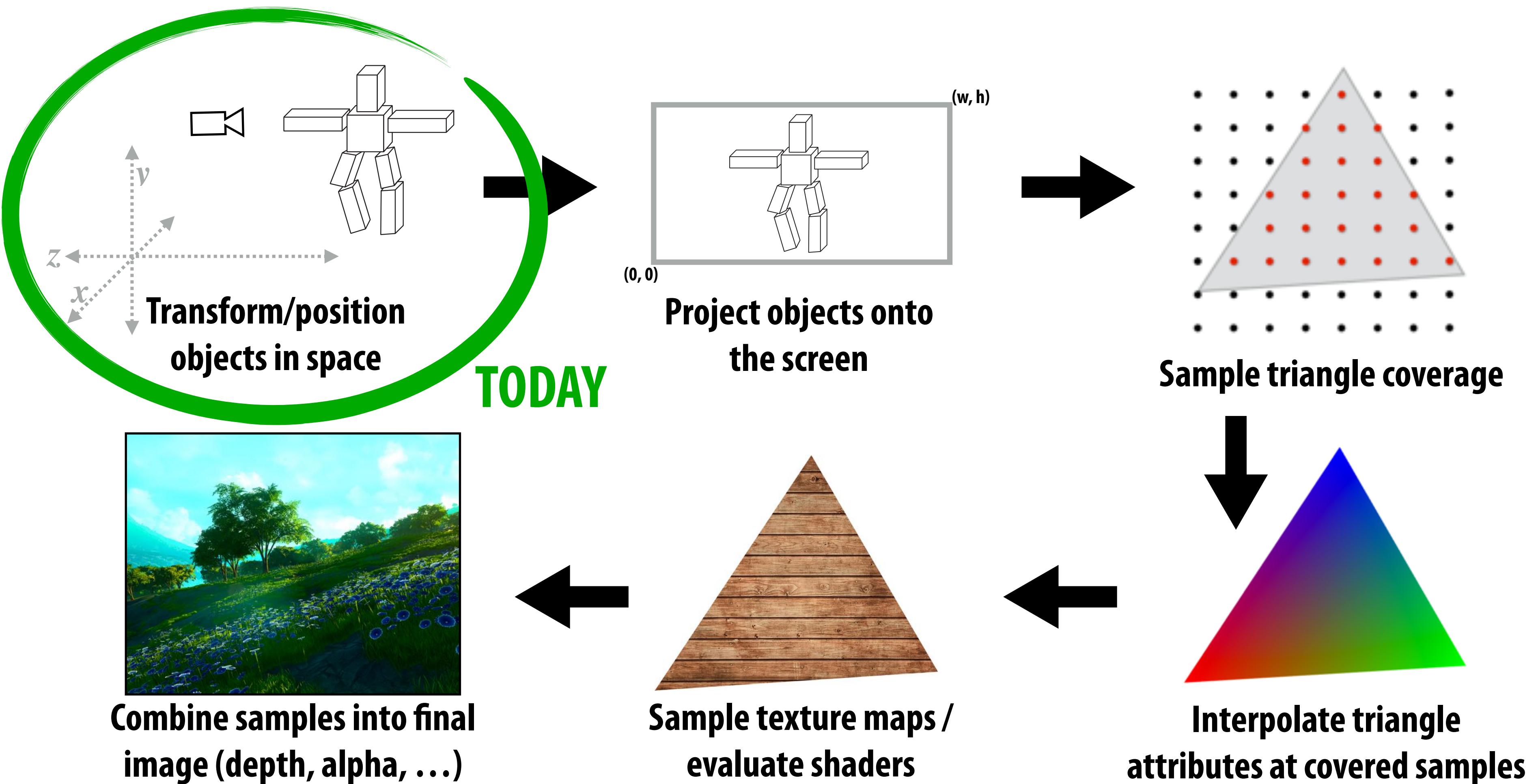
$$f: \mathbb{R}^n \rightarrow \mathbb{R}^n$$

# Transformations in Computer Graphics

- Where are linear transformations used in computer graphics?
- All over the place!
  - Position/deform objects in space
  - Move the camera
  - Animate objects over time
  - Project 3D objects onto 2D images
  - Map 2D textures onto 3D objects
  - Project shadows of 3D objects onto other 3D objects
  - ...



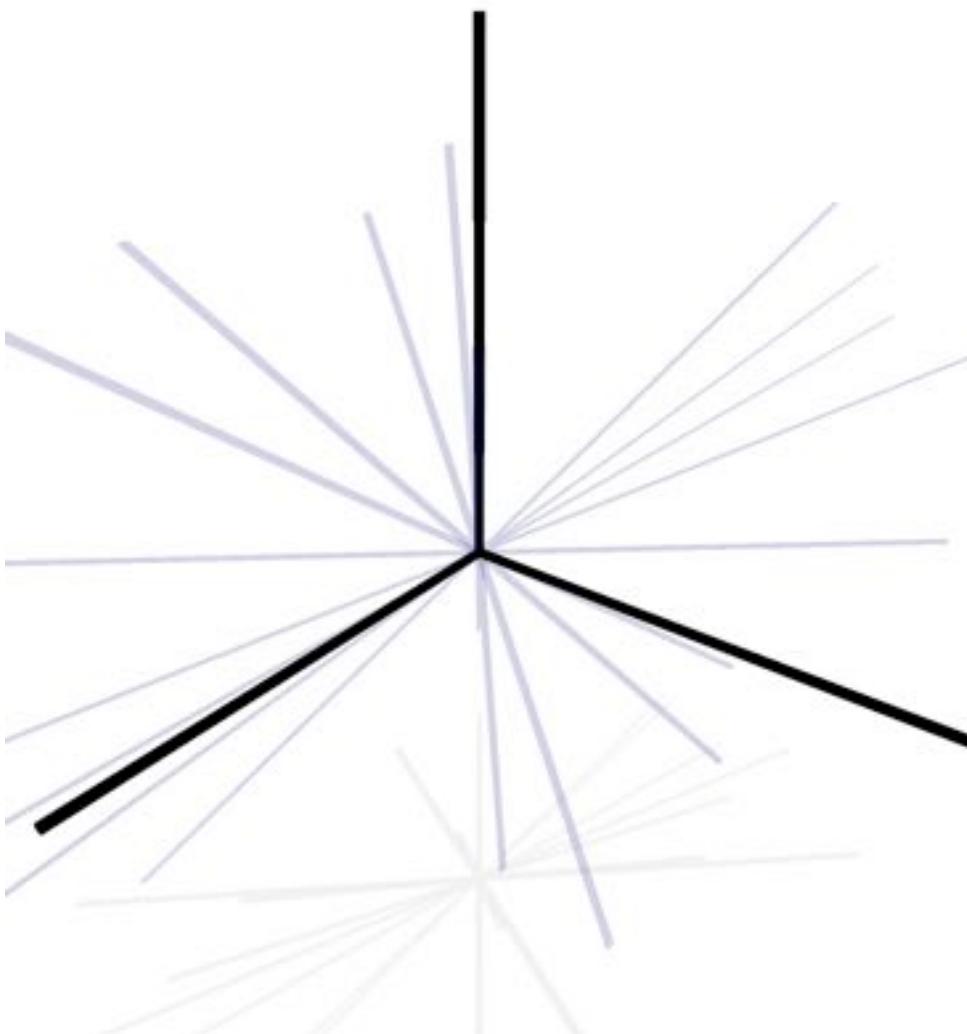
# The Rasterization Pipeline



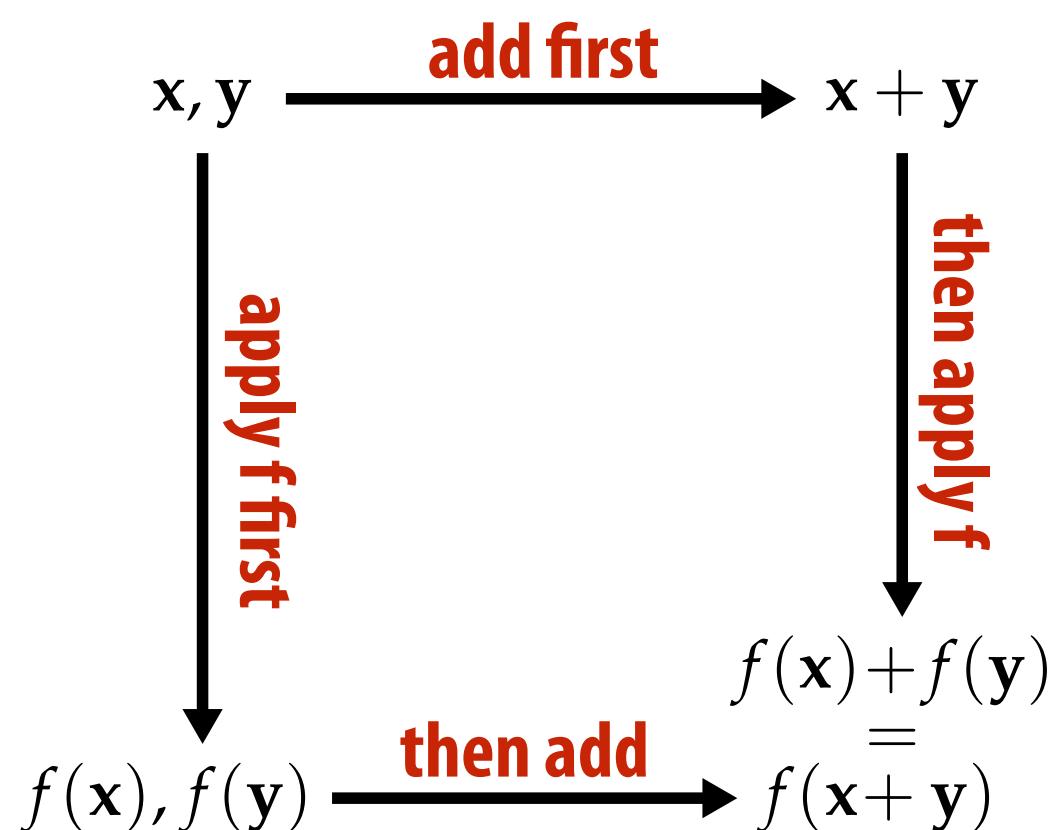
# Review: Linear Maps

Q: What does it mean for a map  $f : \mathbb{R}^n \rightarrow \mathbb{R}^n$  to be linear?

Geometrically: it maps lines to lines, and preserves the origin



Algebraically: preserves vector space operations (addition & scaling)



# Why do we care about linear transformations?

- Cheap to apply
- Usually pretty easy to solve for (linear systems)
- Composition of linear transformations is linear
  - product of many matrices is a single matrix
  - gives uniform representation of transformations
  - simplifies graphics algorithms, systems (e.g., GPUs & APIs)

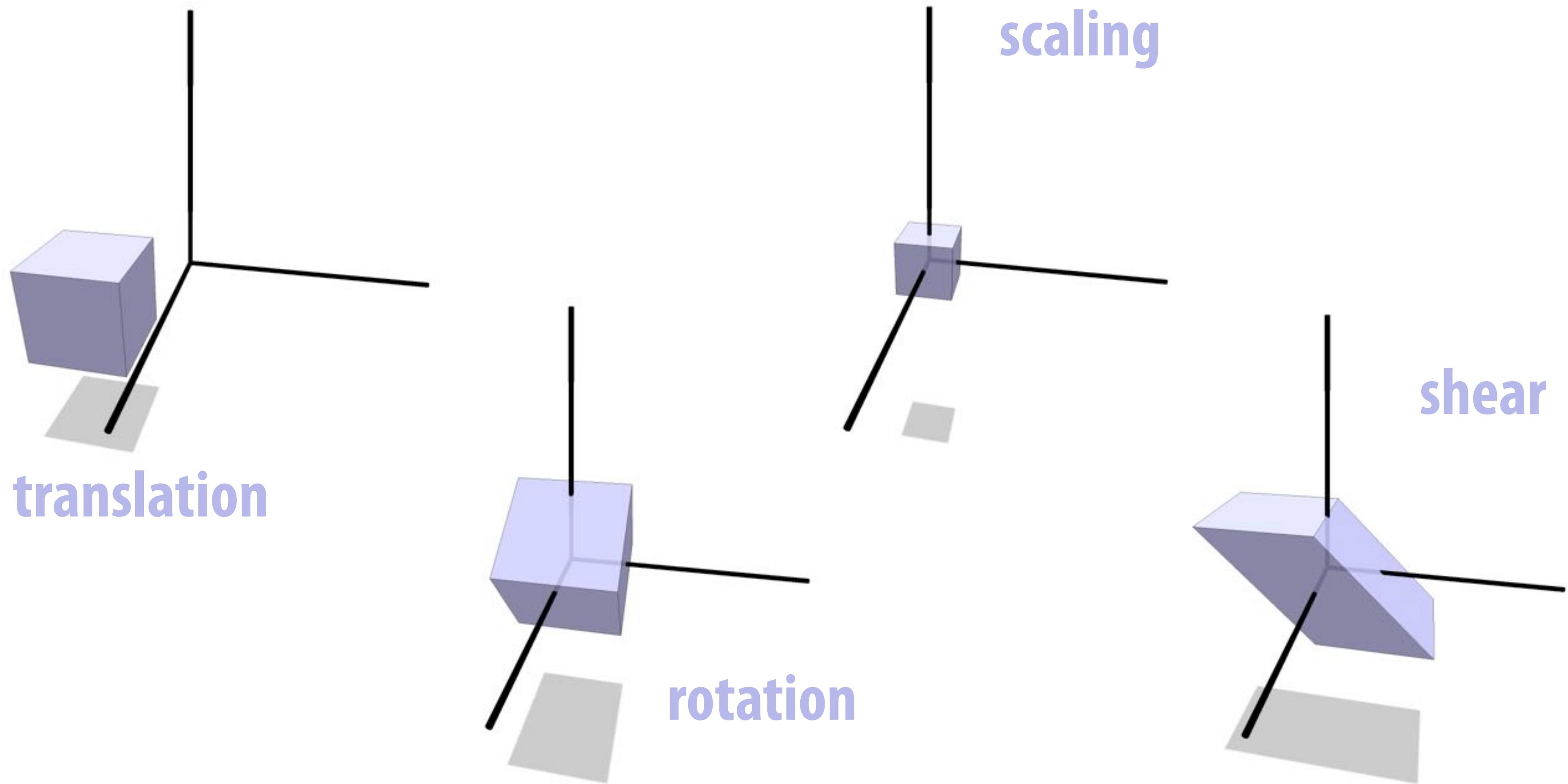
$$\begin{bmatrix} \cos \theta & \sin \theta & 0 \\ -\sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} a & 0 & 0 \\ 0 & b & 0 \\ 0 & 0 & c \end{bmatrix} \begin{bmatrix} \cos \theta & 0 & \sin \theta \\ 0 & 1 & 0 \\ -\sin \theta & 0 & \cos \theta \end{bmatrix} \cdots = \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix}$$

*rotation*                    *scale*                    *rotation*                    *composite transformation*

**What kinds of linear  
transformations can we compose?**

# Types of Transformations

What would you call each of these types of transformations?



**Q: How did you know that? (Hint: you did not inspect a formula!)**

# Invariants of Transformation

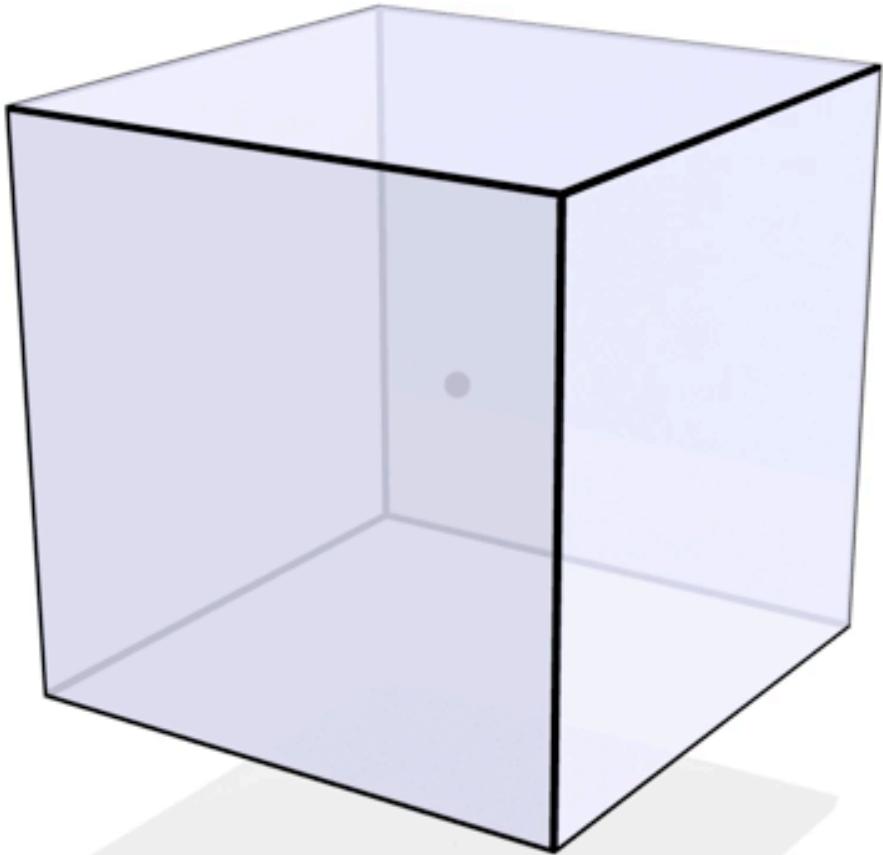
A transformation is determined by the invariants it preserves

transformation	invariants	algebraic description
linear	<i>straight lines / origin</i>	$f(ax+y) = af(x) + f(y),$ $f(0) = 0$
translation	<i>differences between pairs of points</i>	$f(\mathbf{x}-\mathbf{y}) = \mathbf{x}-\mathbf{y}$
scaling	<i>lines through the origin / direction of vectors</i>	$f(\mathbf{x})/ f(\mathbf{x})  = \mathbf{x}/ \mathbf{x} $
rotation	<i>origin / distances between points / orientation</i>	$ f(\mathbf{x})-f(\mathbf{y})  =  \mathbf{x}-\mathbf{y} ,$ $\det(f) > 0$
...	...	...

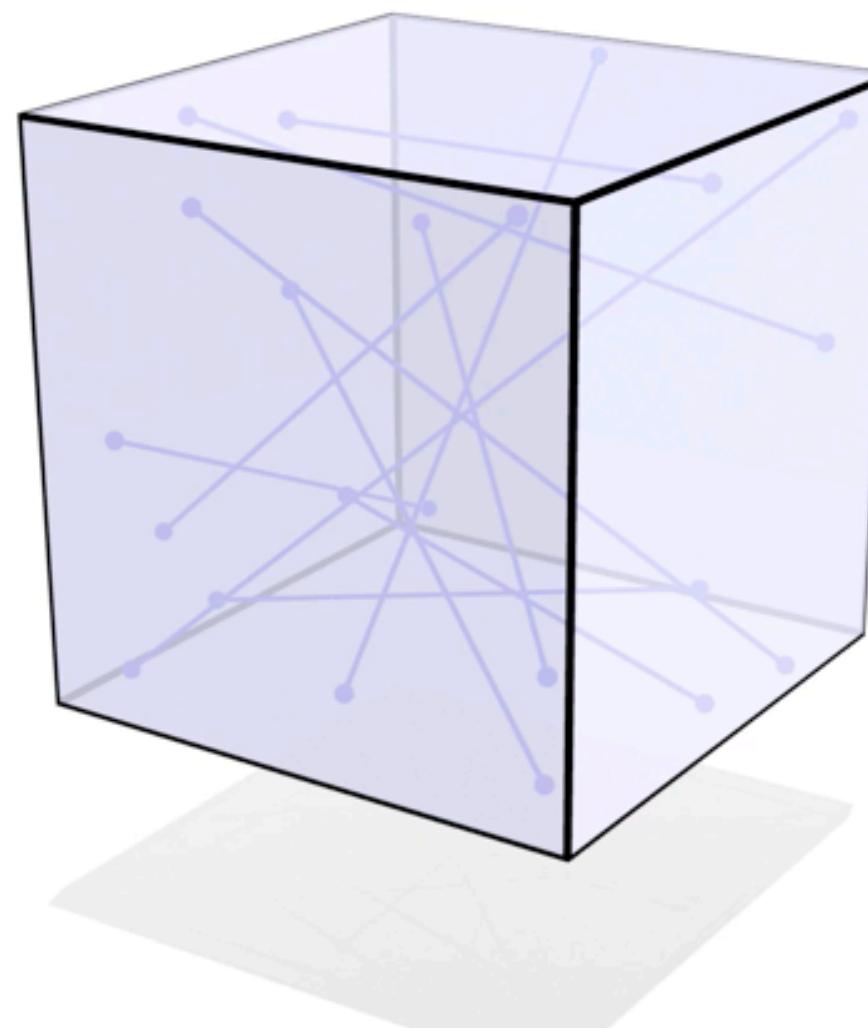
(Essentially how your brain “knows” what kind of transformation you’re looking at...)

# Rotation

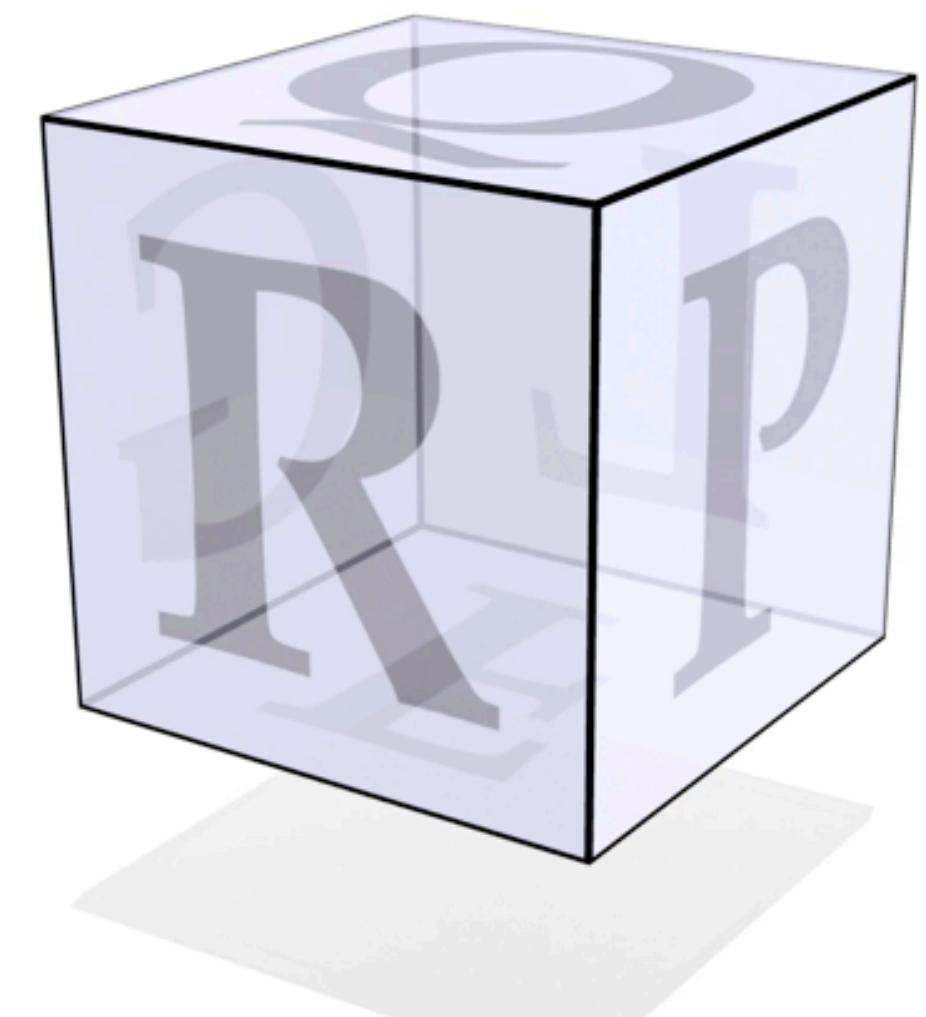
**Rotations defined by three basic properties:**



**keeps origin fixed**



**preserves distances**



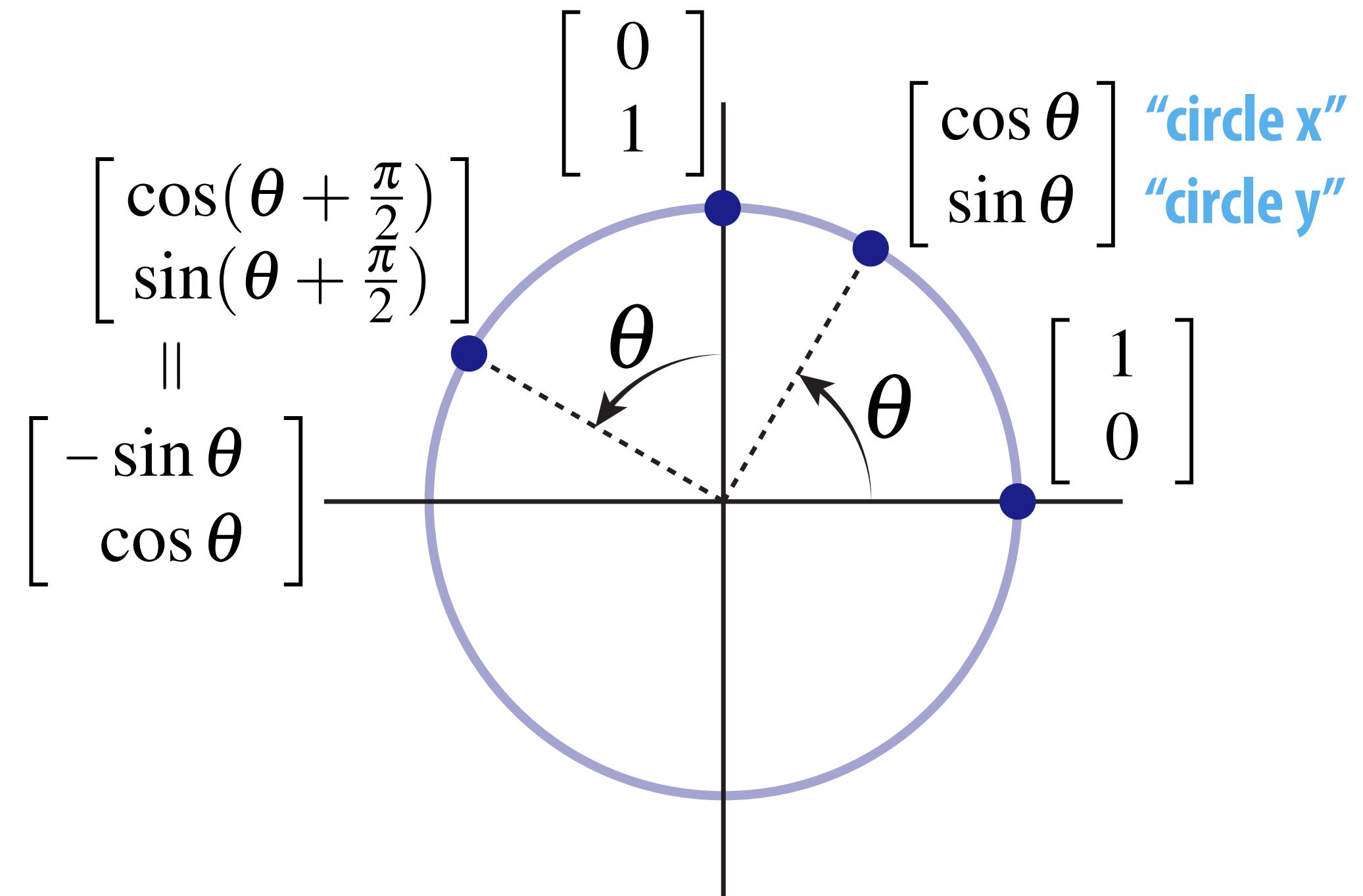
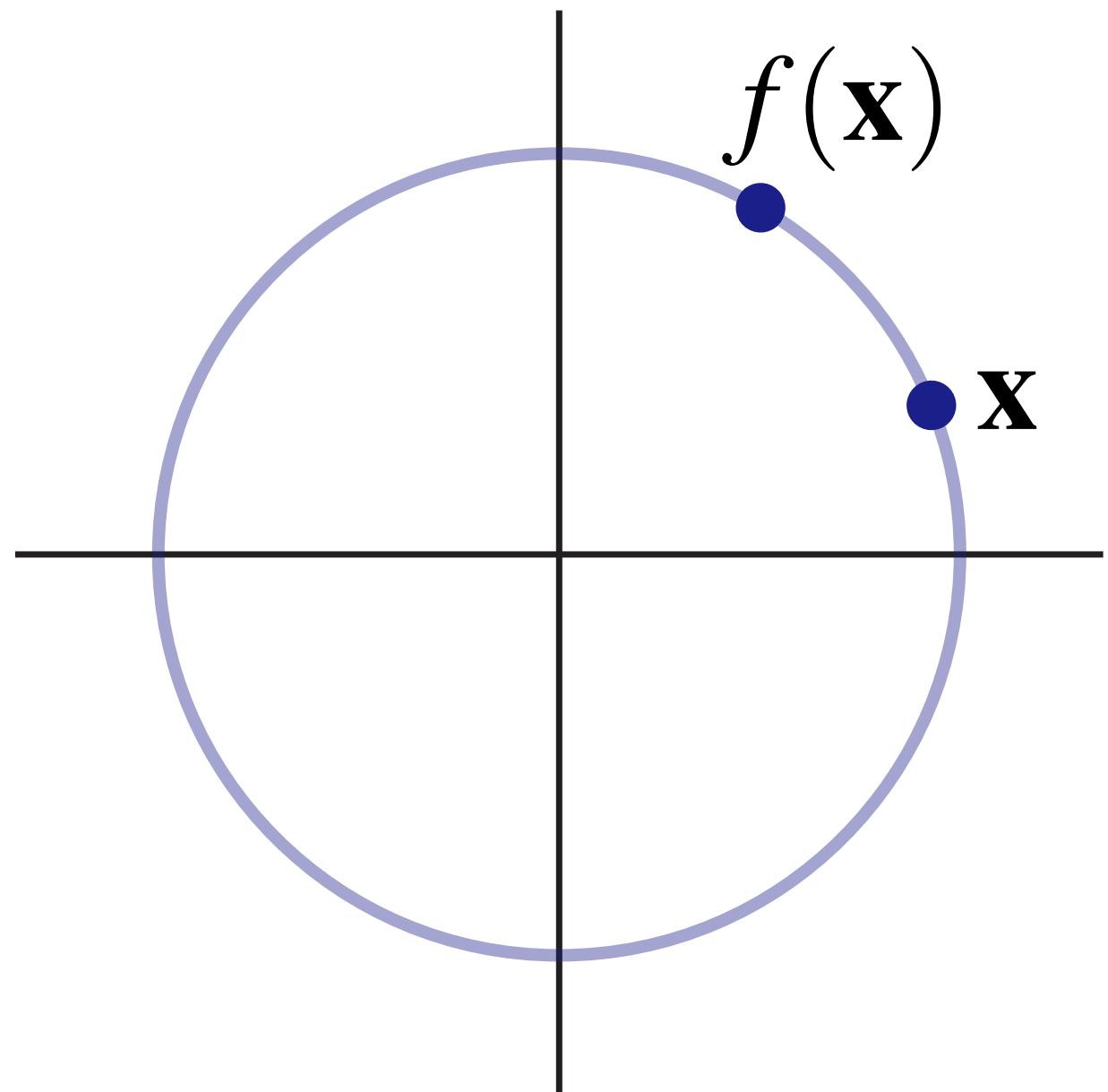
**preserves orientation**

**First two properties together imply that rotations are linear.**

**Will have a lot more to say about rotations next lecture...**

# 2D Rotations—Matrix Representation

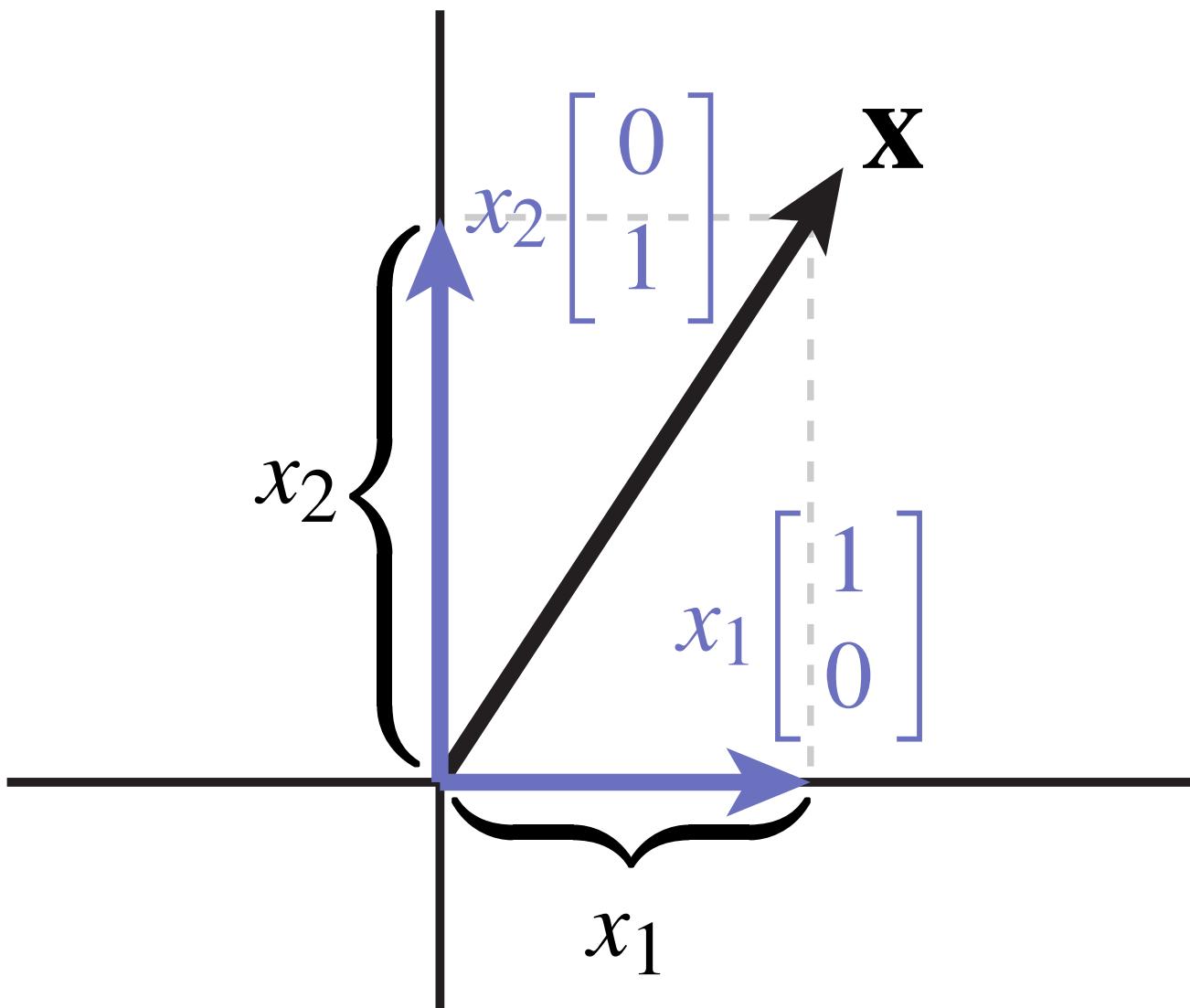
Rotations preserve distances and the origin—hence, a 2D rotation by an angle  $\theta$  maps each point  $\mathbf{x}$  to a point  $f_\theta(\mathbf{x})$  on the circle of radius  $|\mathbf{x}|$ :



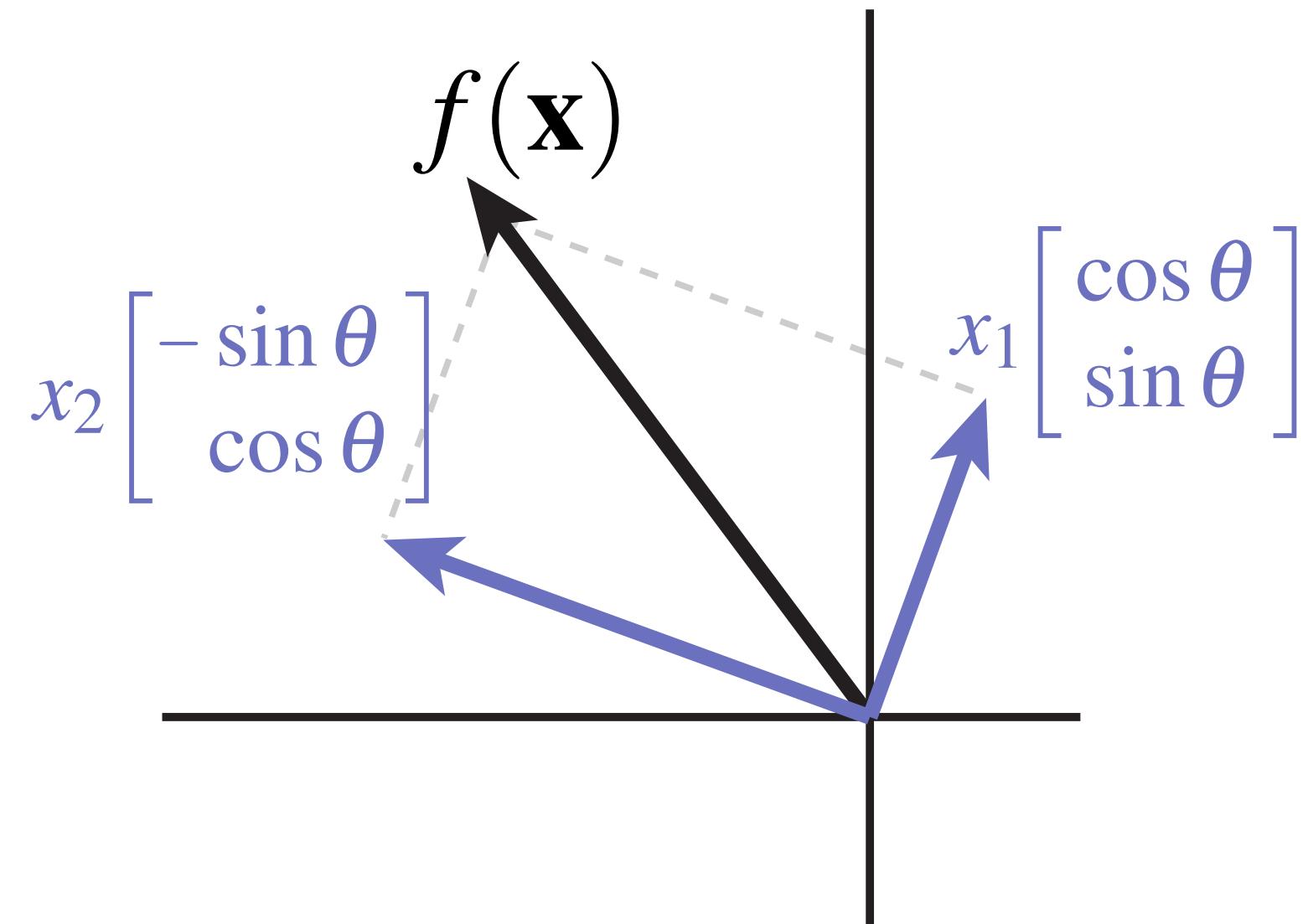
- Where does  $\mathbf{x} = (1,0)$  go if we rotate by  $\theta$  (counter-clockwise)?
- How about  $\mathbf{x} = (0,1)$ ?

What about a general vector  $\mathbf{x} = (x_1, x_2)$ ?

# 2D Rotations—Matrix Representation



$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = x_1 \begin{bmatrix} 1 \\ 0 \end{bmatrix} + x_2 \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$



$$f(\mathbf{x}) = x_1 \begin{bmatrix} \cos \theta \\ \sin \theta \end{bmatrix} + x_2 \begin{bmatrix} -\sin \theta \\ \cos \theta \end{bmatrix}$$

So, How do we represent the 2D rotation function  $f_\theta(\mathbf{x})$  using a matrix?

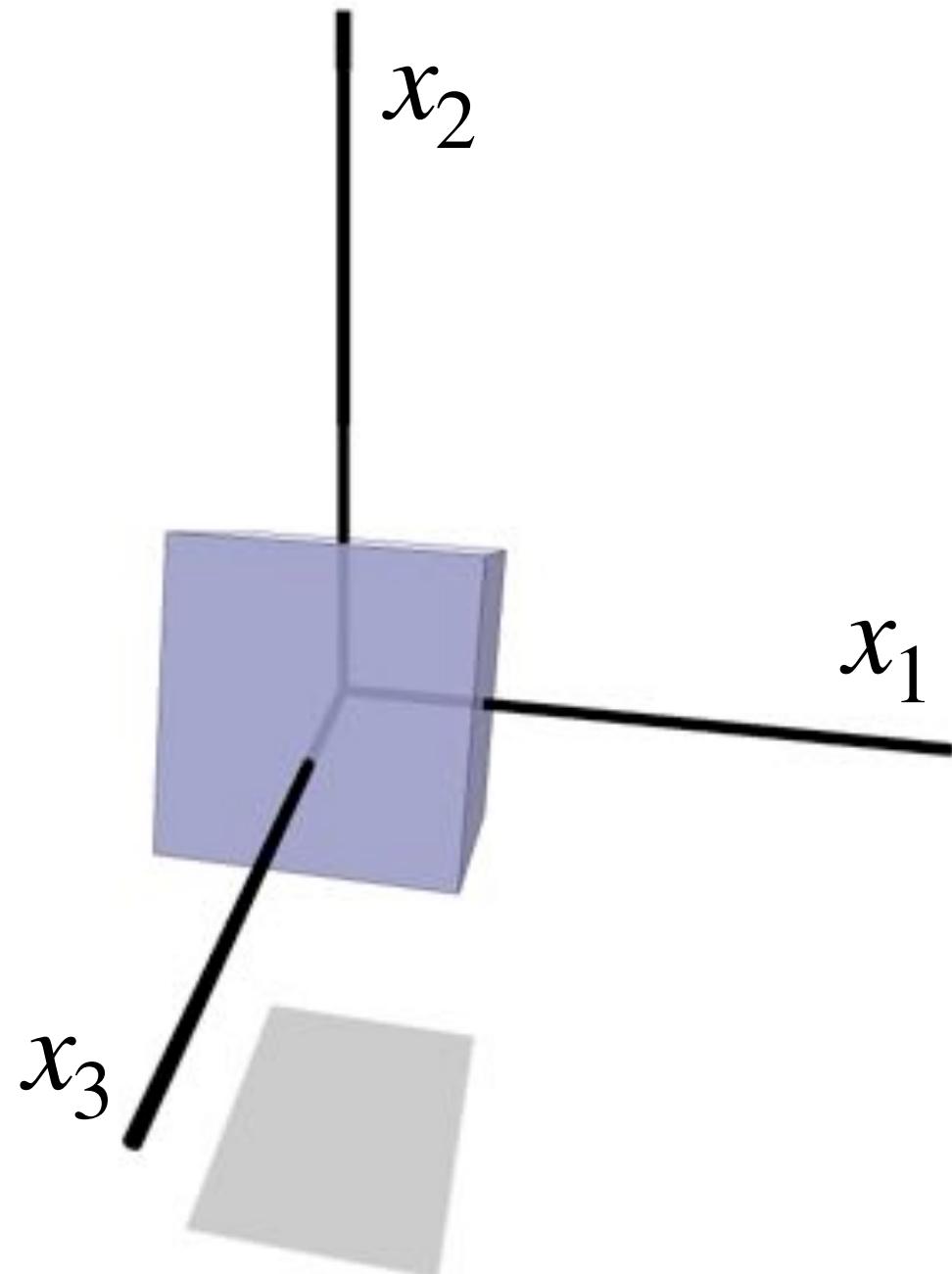
$$f_\theta(\mathbf{x}) = \begin{bmatrix} \cos \theta & -\sin(\theta) \\ \sin \theta & \cos(\theta) \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

# 3D Rotations

- Q: In 3D, how do we rotate around the  $x_3$ -axis?
- A: Just apply the same transformation of  $x_1, x_2$ ; keep  $x_3$  fixed

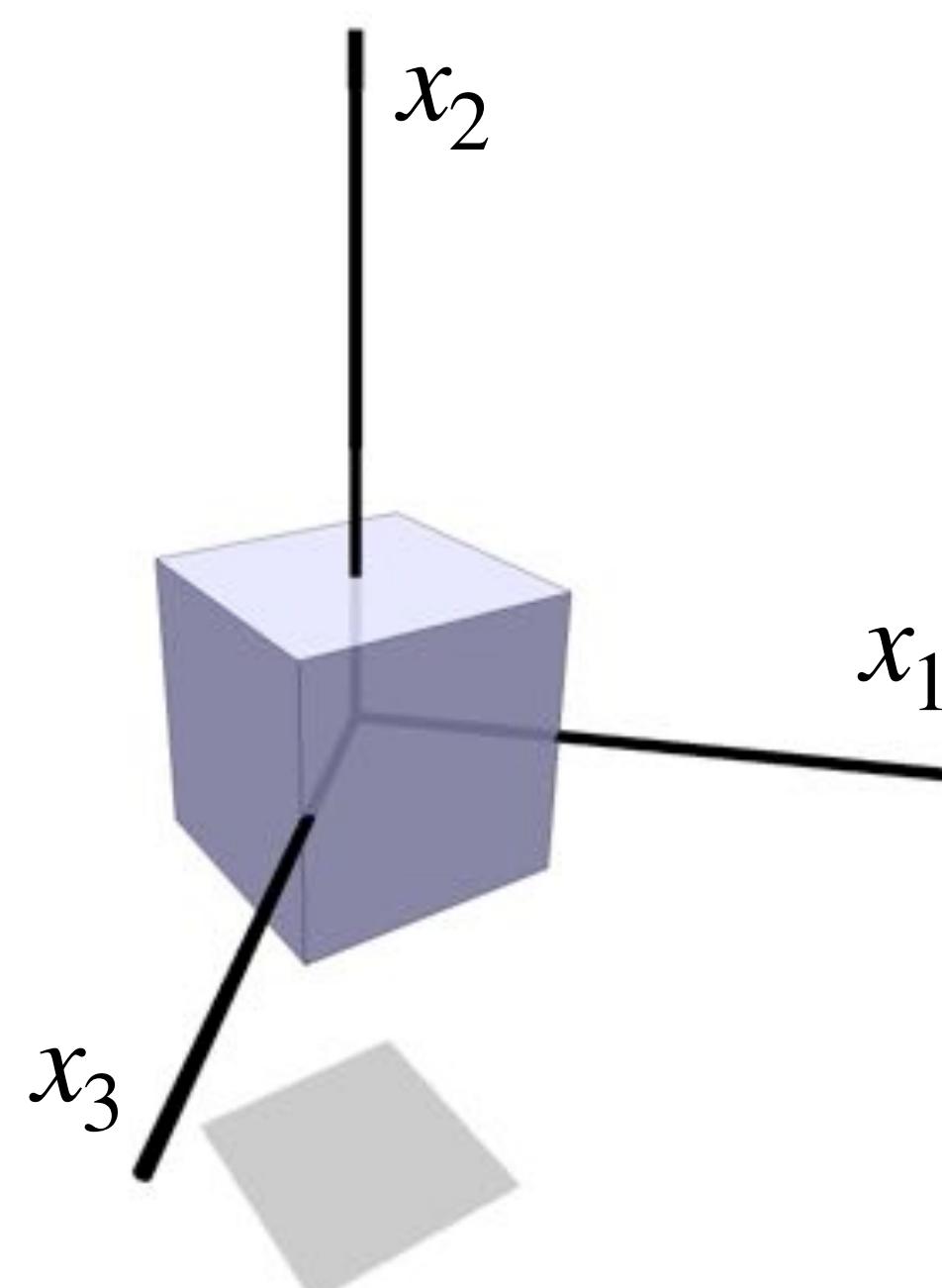
rotate around  $x_1$

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos \theta & -\sin(\theta) \\ 0 & \sin \theta & \cos(\theta) \end{bmatrix}$$



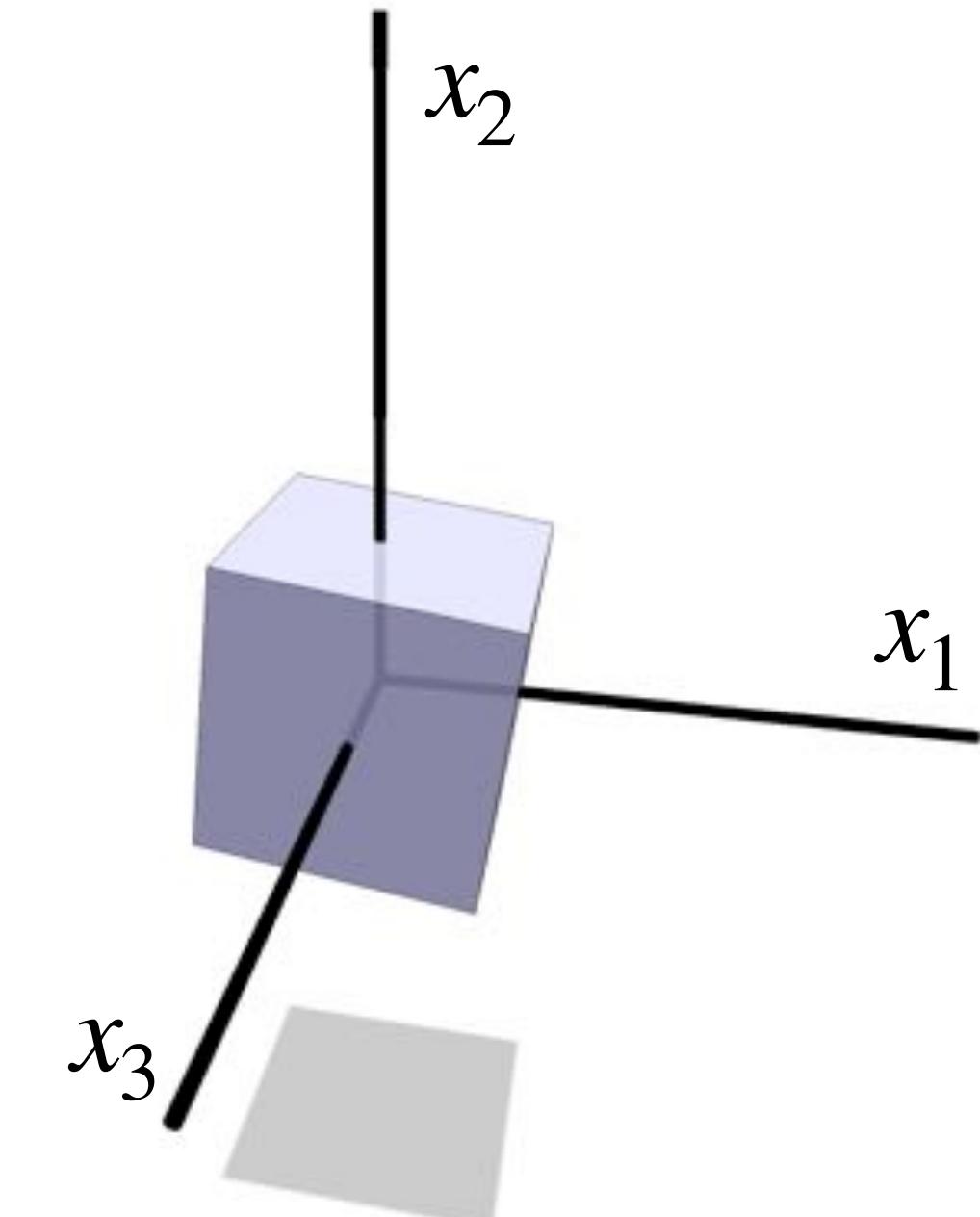
rotate around  $x_2$

$$\begin{bmatrix} \cos \theta & 0 & \sin(\theta) \\ 0 & 1 & 0 \\ -\sin \theta & 0 & \cos(\theta) \end{bmatrix}$$



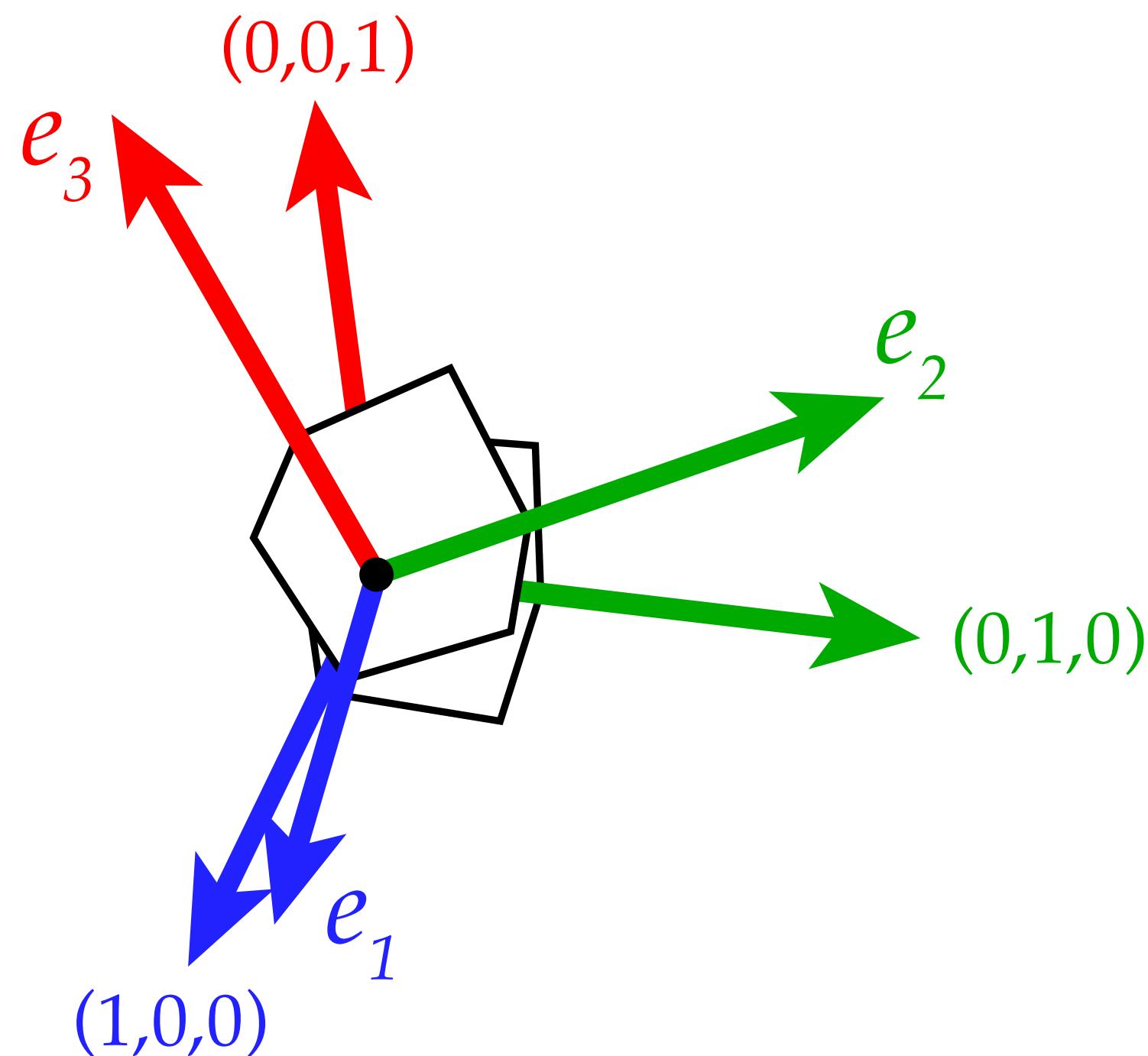
rotate around  $x_3$

$$\begin{bmatrix} \cos \theta & -\sin(\theta) & 0 \\ \sin \theta & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$



# Rotations—Transpose as Inverse

Rotation will map standard basis to orthonormal basis  $e_1, e_2, e_3$ :



$$\begin{aligned} R^T &= \begin{bmatrix} e_1^T \\ e_2^T \\ e_3^T \end{bmatrix} \quad R = \begin{bmatrix} | & | & | \\ e_1 & e_2 & e_3 \\ | & | & | \end{bmatrix} \\ &= \begin{bmatrix} e_1^T e_1 & e_1^T e_2 & e_1^T e_3 \\ e_2^T e_1 & e_2^T e_2 & e_2^T e_3 \\ e_3^T e_1 & e_3^T e_2 & e_3^T e_3 \end{bmatrix} \\ &\quad I \end{aligned}$$

The diagram illustrates the matrix multiplication  $R^T R = I$ . The top part shows the transpose of the rotation matrix  $R^T$  as a column vector of orthonormal basis vectors  $e_1^T, e_2^T, e_3^T$ . The bottom part shows the result of multiplying  $R^T$  by  $R$ , resulting in the identity matrix  $I$ . The columns of the resulting matrix represent the transformed basis vectors  $e_1^T e_i$  for  $i=1, 2, 3$ . The angles between the original basis vectors and their transformed counterparts are indicated by theta symbols.

Hence,  $R^T R = I$ , or equivalently,  $R^T = R^{-1}$ .

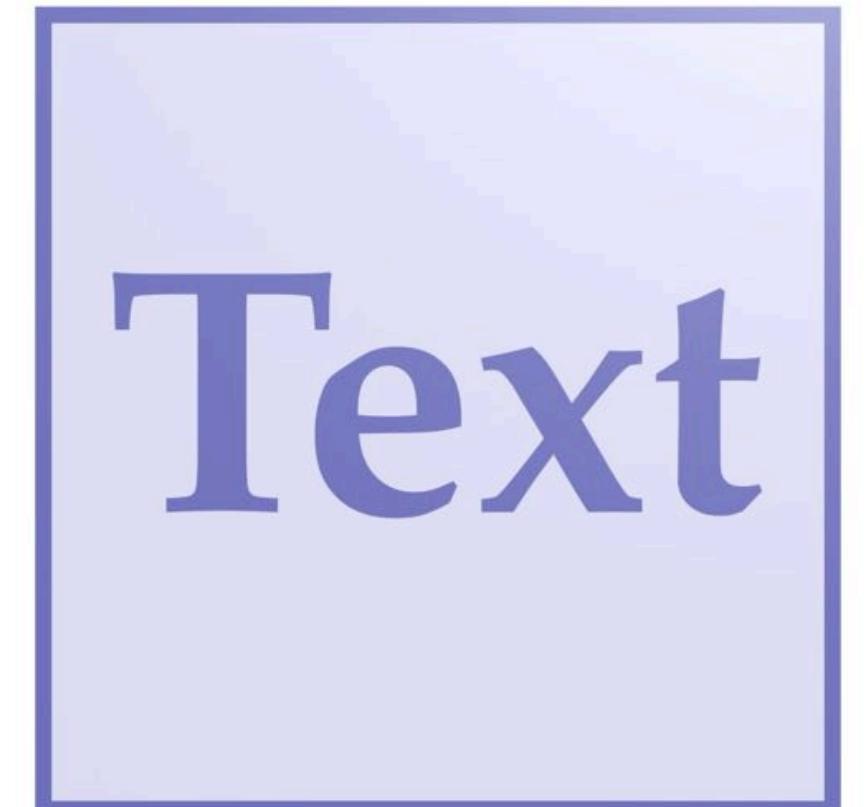
# Reflections

- Q: Does every matrix  $Q^T Q = I$  describe a rotation?
- Remember that rotations must preserve the origin, preserve distances, and preserve orientation
- Consider for instance this matrix:

$$Q = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix} \quad Q^T Q = \begin{bmatrix} (-1)^2 & 0 \\ 0 & 1 \end{bmatrix} = I$$

**Q: Does this matrix represent a rotation?  
(If not, which invariant does it fail to preserve?)**

**A: No! It represents a reflection across the y-axis  
(and hence fails to preserve orientation)**

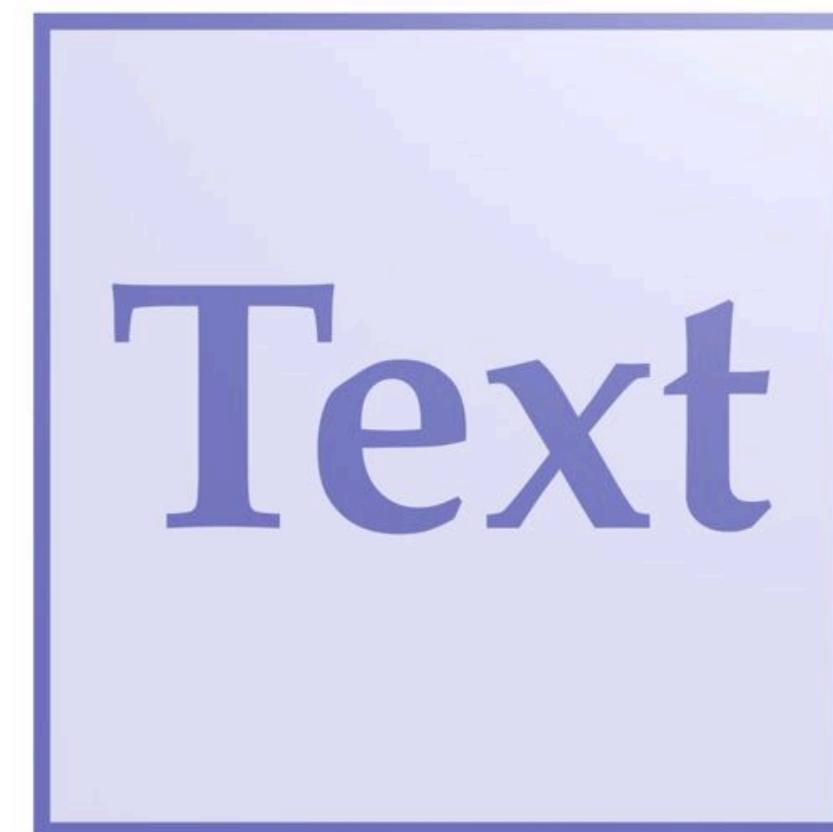


# Orthogonal Transformations

- In general, transformations that preserve distances and the origin are called orthogonal transformations
- Represented by matrices  $Q^T Q = I$ 
  - Rotations additionally preserve orientation:  $\det(Q) > 0$
  - Reflections reverse orientation:  $\det(Q) < 0$



rotation



reflection

# Scaling

\*assuming  $a \neq 0, \mathbf{u} \neq 0$

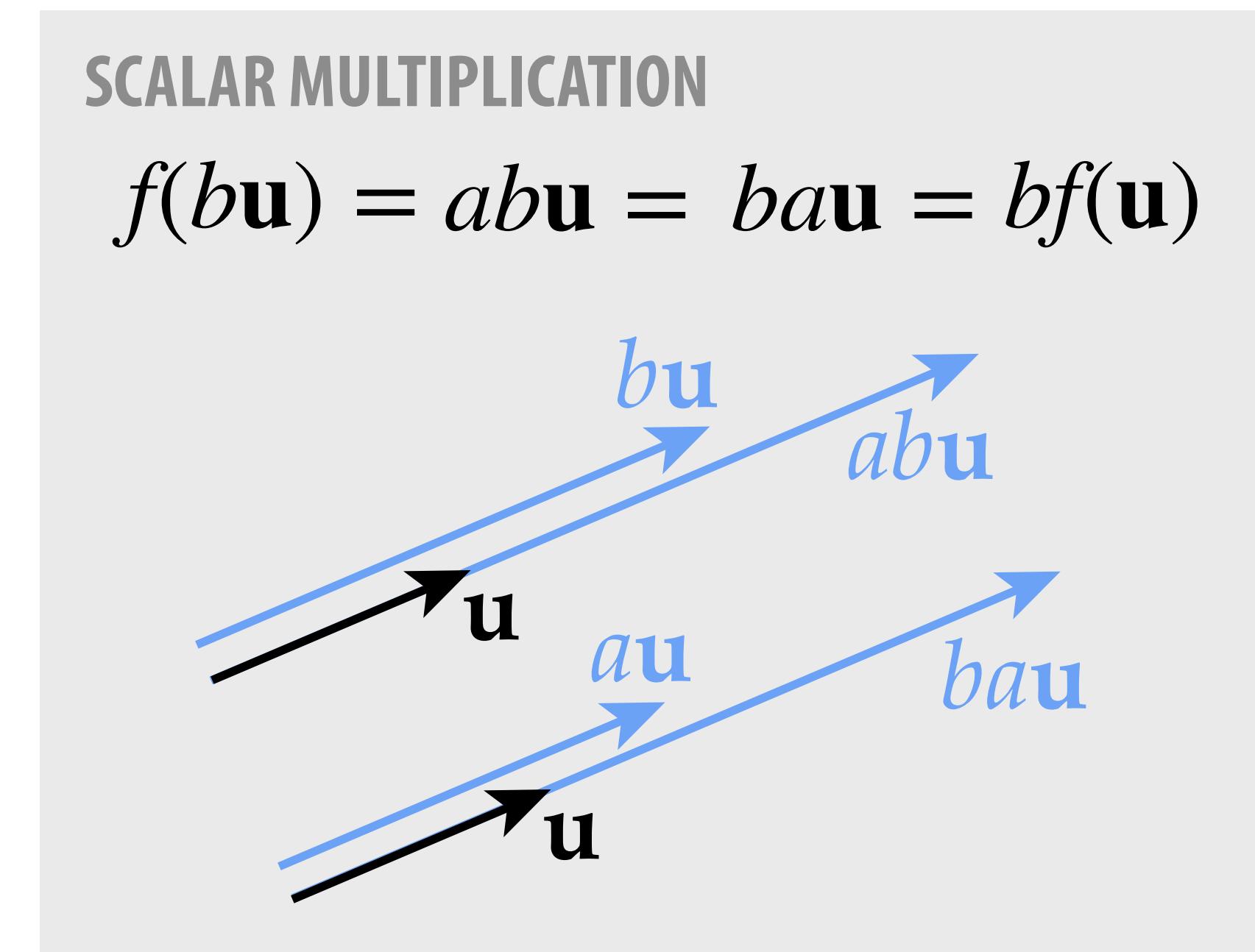
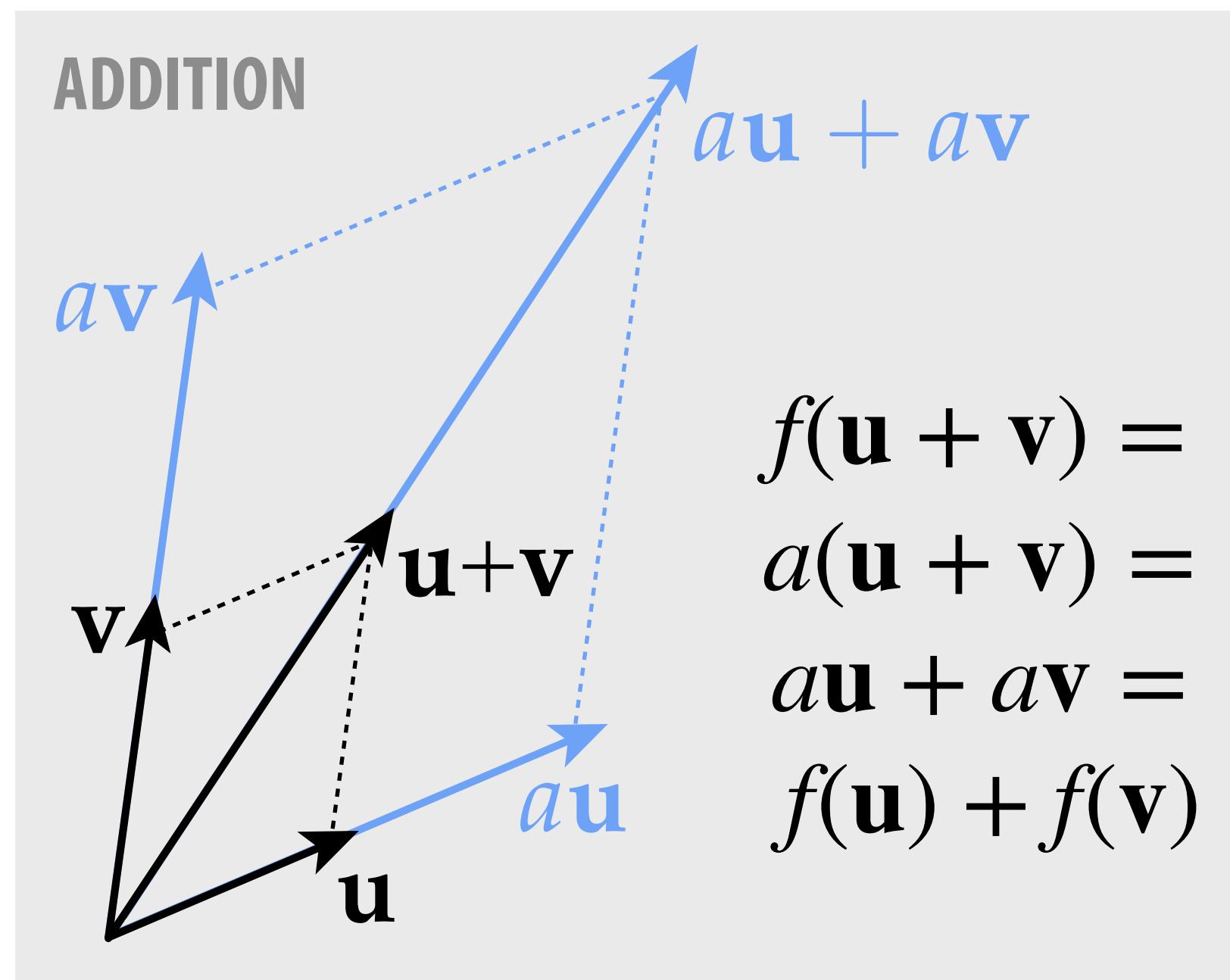
- Each vector  $\mathbf{u}$  gets mapped to a scalar multiple

- $f(\mathbf{u}) = a\mathbf{u}, \quad a \in \mathbb{R}$

- Preserves the direction of all vectors\*

$$\frac{\mathbf{u}}{|\mathbf{u}|} = \frac{a\mathbf{u}}{|a\mathbf{u}|}$$

- Q: Is scaling a linear transformation? A: Yes!



# Scaling — Matrix Representation

**Q:** Suppose we want to scale a vector  $\mathbf{u} = (u_1, u_2, u_3)$  by  $a$ .  
How would we represent this operation via a matrix?

**A:** Just build a diagonal matrix  $D$ , with  $a$  along the diagonal:

$$\underbrace{\begin{bmatrix} a & 0 & 0 \\ 0 & a & 0 \\ 0 & 0 & a \end{bmatrix}}_D \underbrace{\begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix}}_{\mathbf{u}} = \underbrace{\begin{bmatrix} au_1 \\ au_2 \\ au_3 \end{bmatrix}}_{a\mathbf{u}}$$

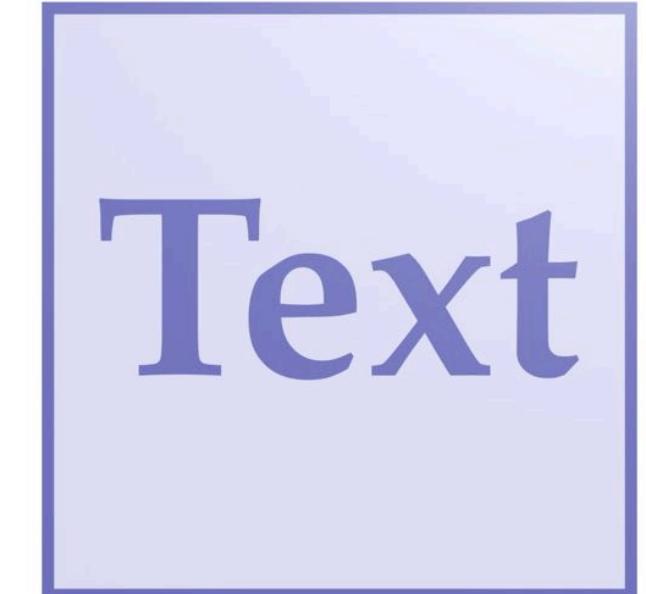
**Q:** What happens if  $a$  is negative?

# Negative Scaling

For  $a = -1$ , can think of scaling by  $a$  as sequence of reflections.

E.g., in 2D:

$$\begin{bmatrix} -1 & 0 \\ 0 & -1 \end{bmatrix} = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$



Since each reflection reverses orientation, orientation is preserved.

What about 3D?

$$\begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{bmatrix} =$$

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



Now we have three reflections, and so orientation is reversed!

# Nonuniform Scaling (Axis-Aligned)

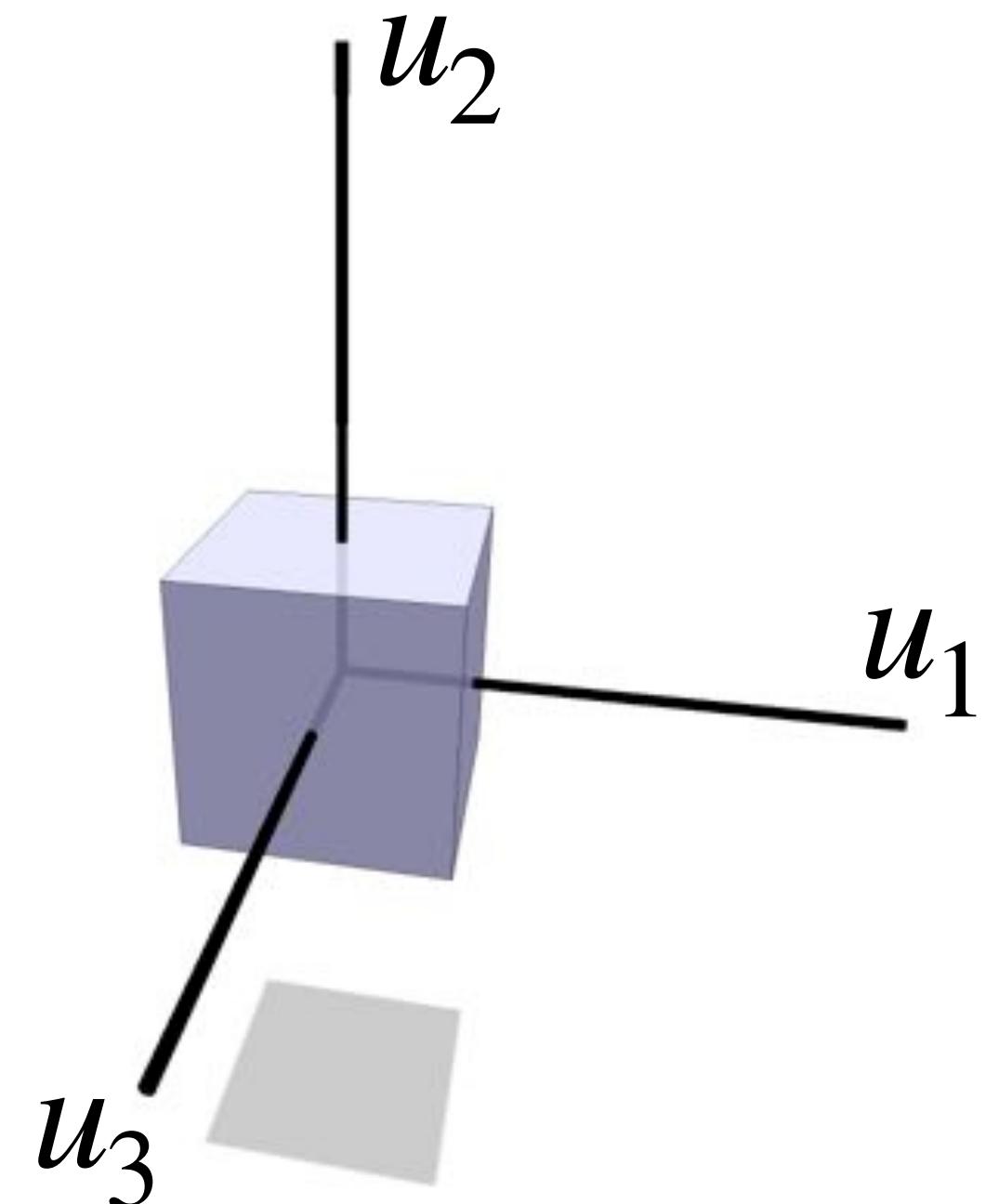
- We can also scale each axis by a different amount

- $f(u_1, u_2, u_3) = (au_1, bu_2, cu_3), \quad a, b, c \in \mathbb{R}$

- Q: What's the matrix representation?

- A: Just put  $a, b, c$  on the diagonal:

$$\begin{bmatrix} a & 0 & 0 \\ 0 & b & 0 \\ 0 & 0 & c \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix} = \begin{bmatrix} au_1 \\ bu_2 \\ cu_3 \end{bmatrix}$$



Ok, but what if we want to scale along some other axes?

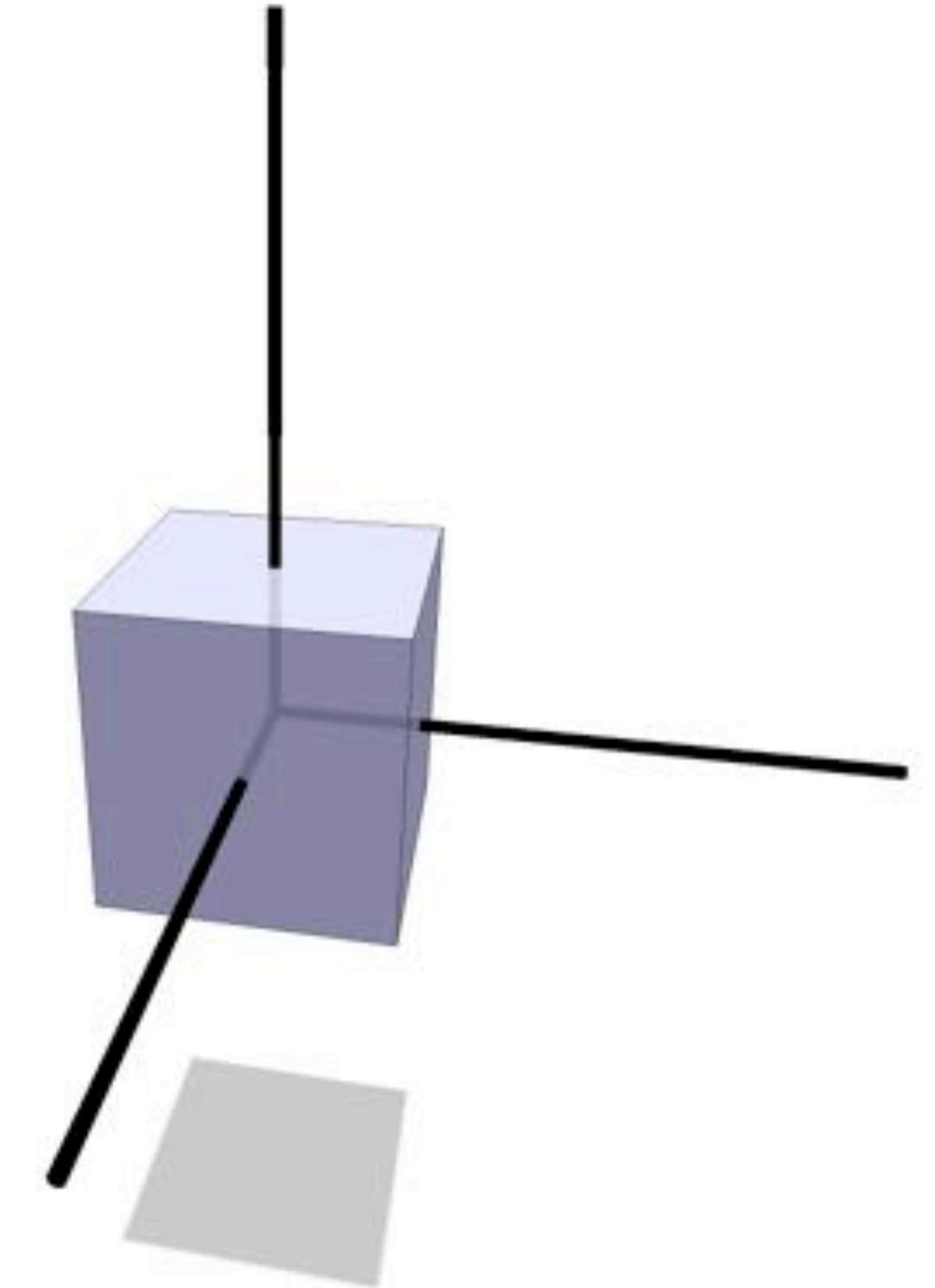
# Nonuniform Scaling

- **Idea.** We could:

- rotate to the new axes ( $R$ )
- apply a diagonal scaling ( $D$ )
- rotate back\* to the original axes ( $R^T$ )

- Notice that the overall transformation is represented by a symmetric matrix

$$A := R^T D R$$



$$f(\mathbf{x}) = R^T D R \mathbf{x}$$

**Q: Do all symmetric matrices represent nonuniform scaling (for some choice of axes)?**

\*Recall that for a rotation, the inverse equals the transpose:  $R^{-1} = R^T$

# Spectral Theorem

- A: Yes! Spectral theorem says a symmetric matrix  $A = A^\top$  has
  - orthonormal eigenvectors  $e_1, \dots, e_n \in \mathbb{R}^n$
  - real eigenvalues  $\lambda_1, \dots, \lambda_n \in \mathbb{R}$
- Can also write this relationship as  $AR = RD$ , where
- Equivalently,  $A = RDR^\top$
- Hence, every symmetric matrix performs a non-uniform scaling along some set of orthogonal axes.
- If  $A$  is positive definite ( $\lambda_i > 0$ ), this scaling is positive.

$$Ae_i = \lambda_i e_i$$

$$R = \begin{bmatrix} e_1 & \cdots & e_n \end{bmatrix} \quad D = \begin{bmatrix} \lambda_1 & & \\ & \ddots & \\ & & \lambda_n \end{bmatrix}$$

Hence, every symmetric matrix performs a non-uniform scaling along some set of orthogonal axes.

If  $A$  is positive definite ( $\lambda_i > 0$ ), this scaling is positive.

# Shear

- A shear displaces each point  $\mathbf{x}$  in a direction  $\mathbf{u}$  according to its distance along a fixed vector  $\mathbf{v}$ :

$$f_{\mathbf{u}, \mathbf{v}}(\mathbf{x}) = \mathbf{x} + \langle \mathbf{v}, \mathbf{x} \rangle \mathbf{u}$$

- Q: Is this transformation linear?
- A: Yes—for instance, can represent it via a matrix

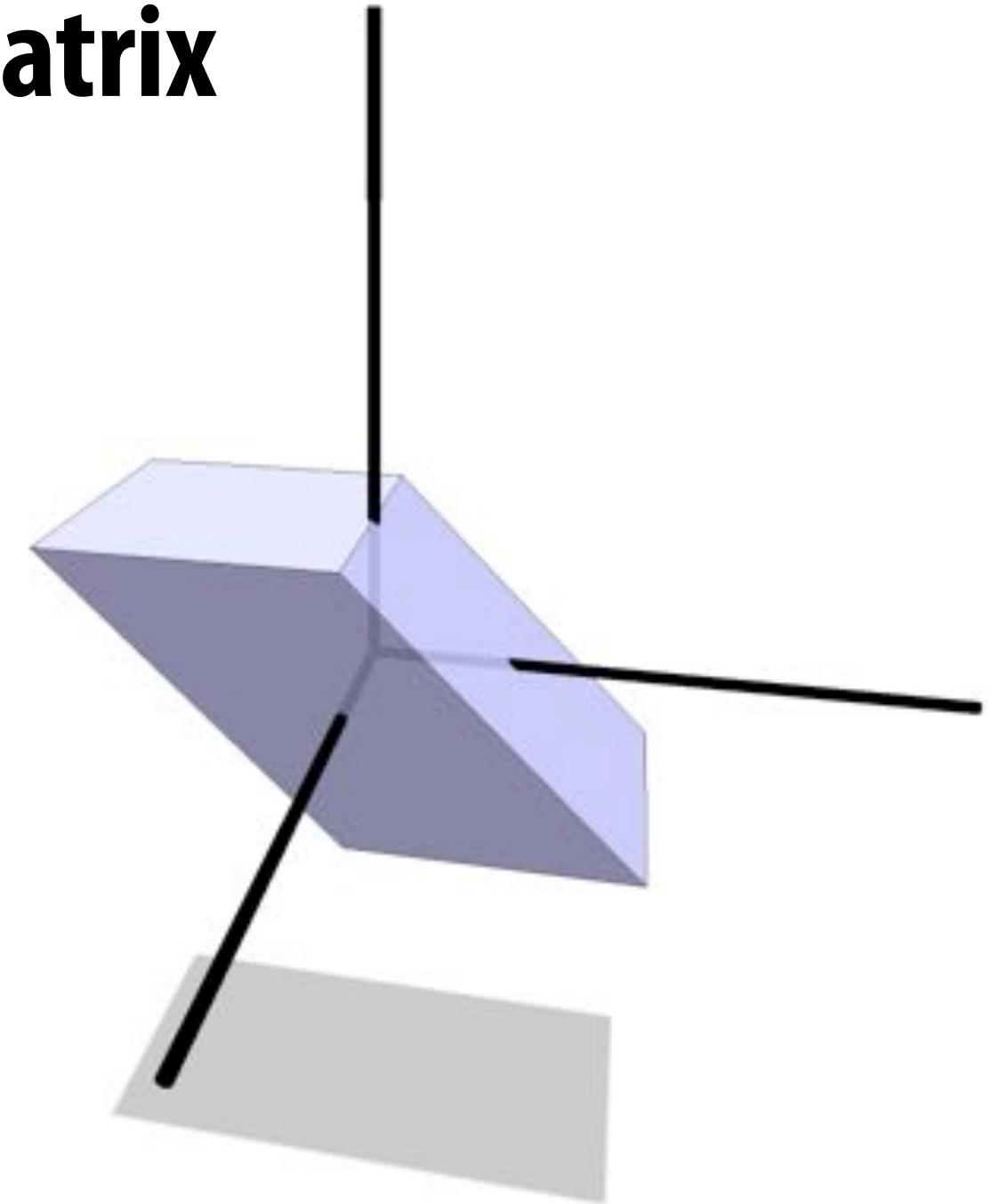
$$A_{\mathbf{u}, \mathbf{v}} = I + \mathbf{u}\mathbf{v}^T$$

Example.

$$\mathbf{u} = (\cos(t), 0, 0)$$

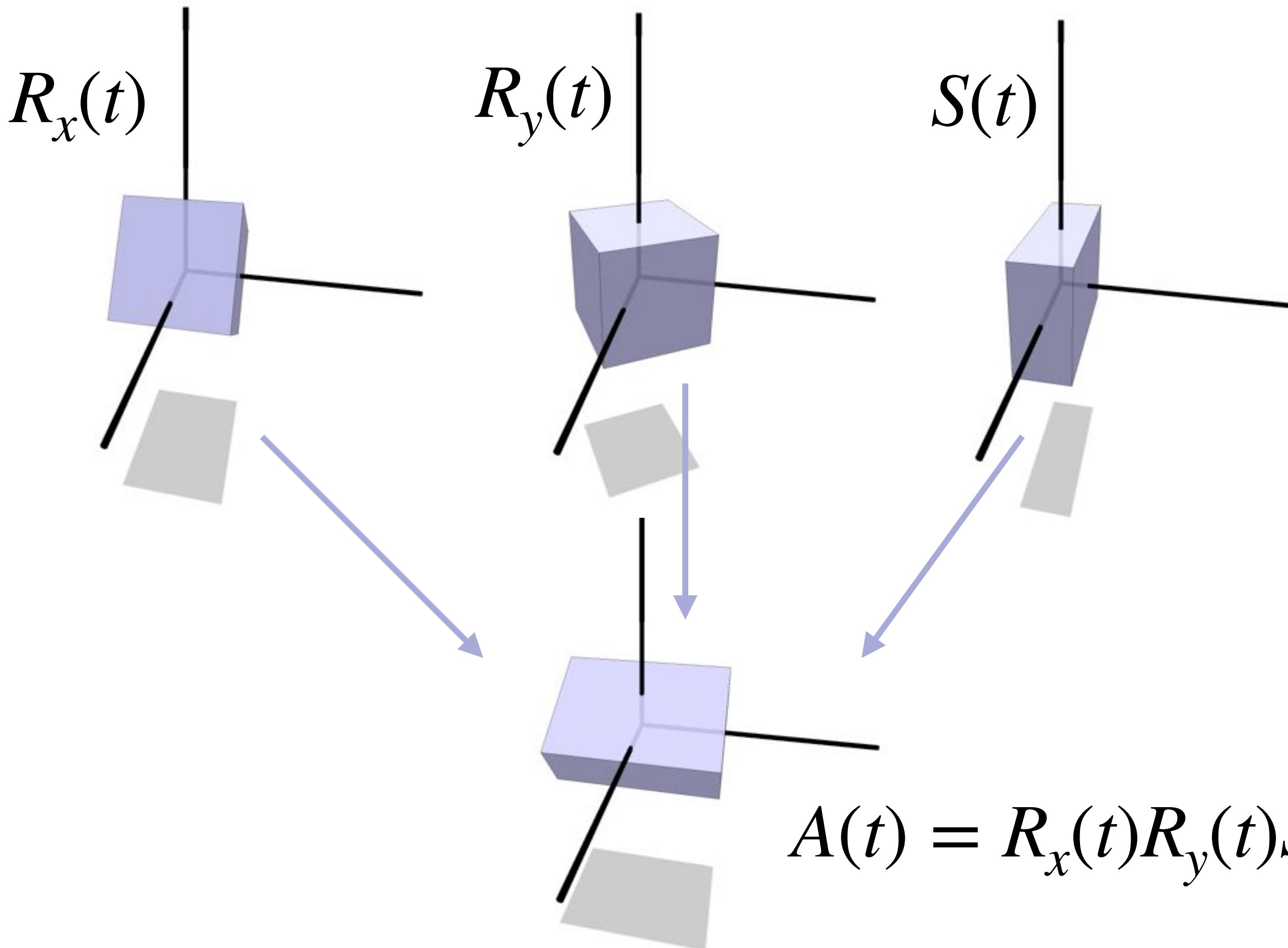
$$\mathbf{v} = (0, 1, 0)$$

$$A_{\mathbf{u}, \mathbf{v}} = \begin{bmatrix} 1 & \cos(t) & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$



# Composite Transformations

From these basic transformations (rotation, reflection, scaling, shear...) we can now build up composite transformations via matrix multiplication:



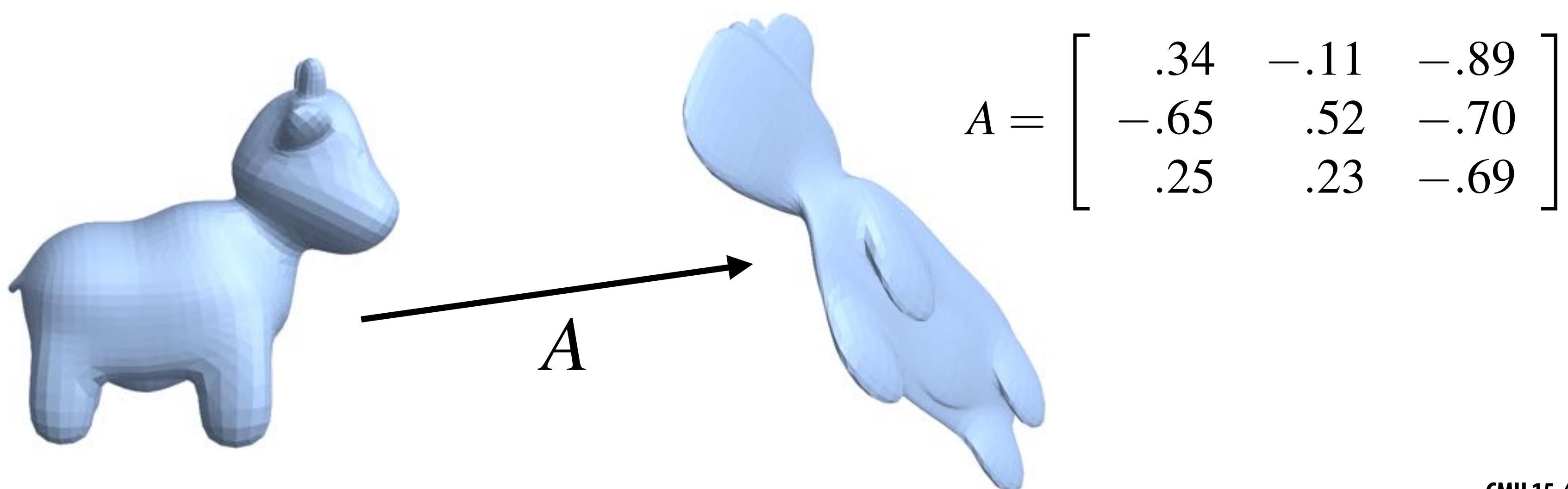
$$A(t) = R_x(t)R_y(t)S(t)$$

**How do we decompose a linear  
transformation into pieces?**

*(rotations, reflections, scaling, ...)*

# Decomposition of Linear Transformations

- In general, no **unique** way to write a given linear transformation as a composition of basic transformations!
- However, there are many useful decompositions:
  - singular value decomposition (good for signal processing)
  - LU factorization (good for solving linear systems)
  - polar decomposition (good for spatial transformations)
  - ...
- Consider for instance this linear transformation:



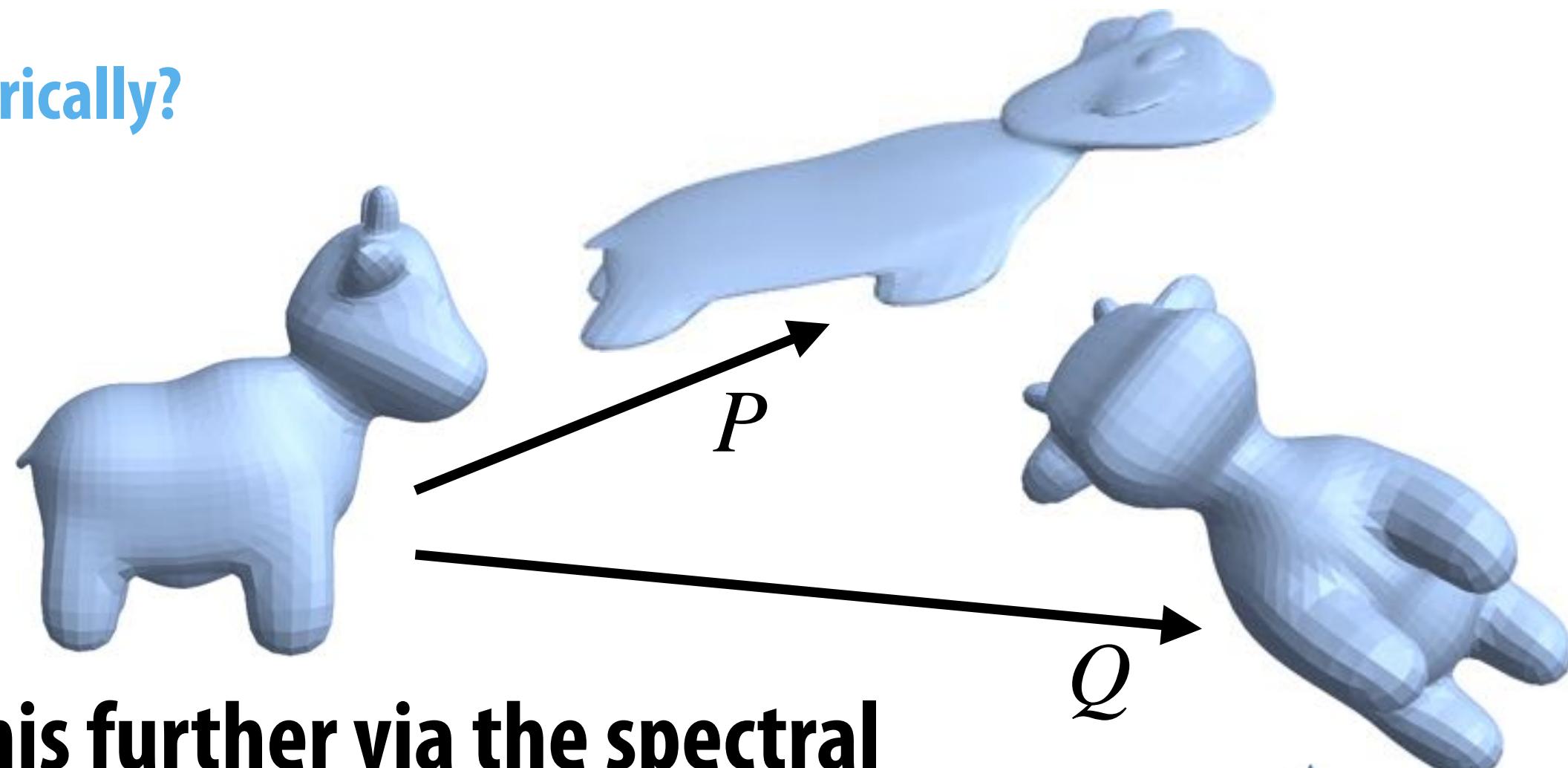
# Polar & Singular Value Decomposition

For example, polar decomposition decomposes any matrix  $A$  into orthogonal matrix  $Q$  and symmetric positive-semidefinite matrix  $P$ :

Q: What do each of the parts mean geometrically?

rotation/reflection      nonnegative,  
nonuniform scaling

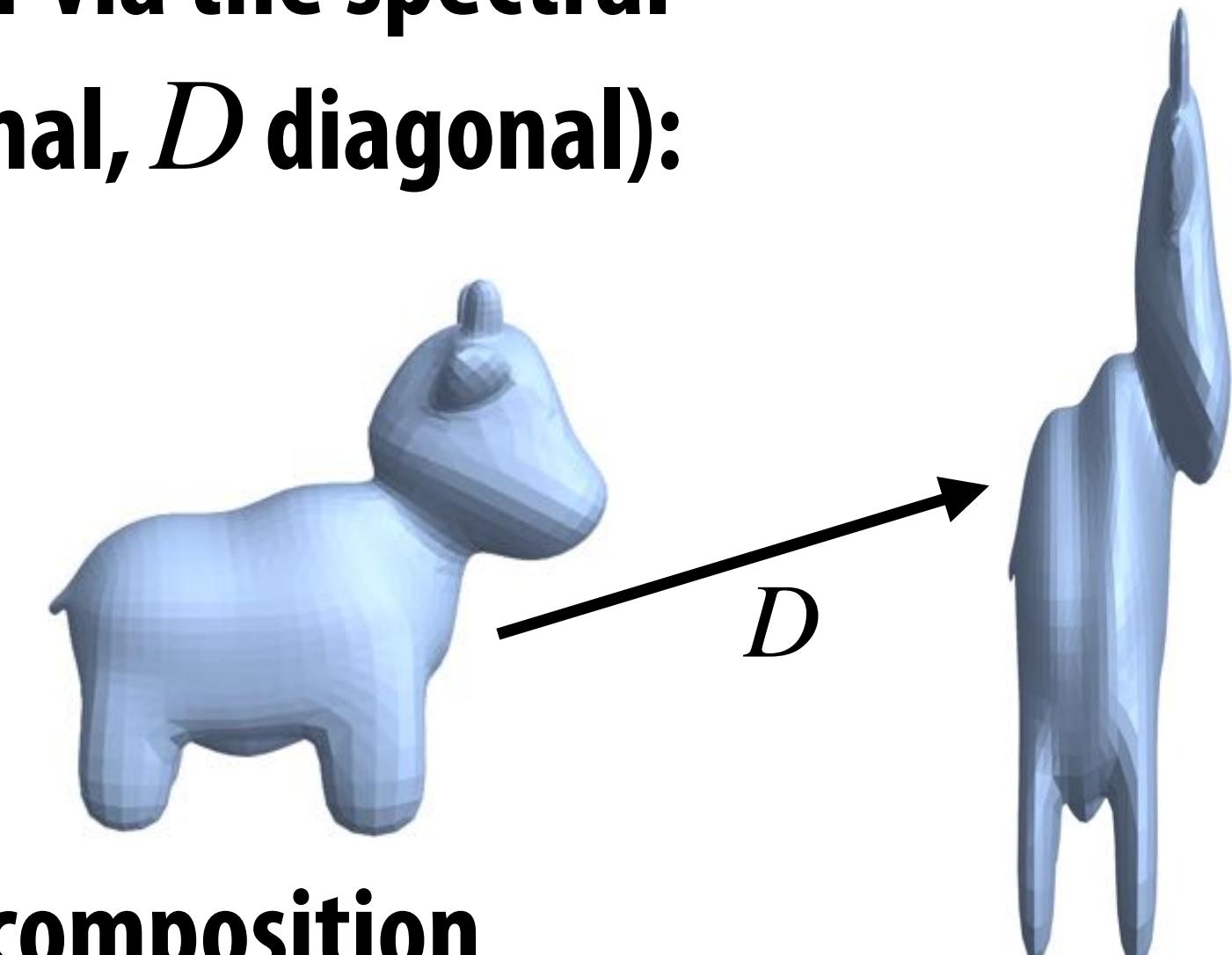
$$A = QP$$



Since  $P$  is symmetric, can take this further via the spectral decomposition  $P = VDV^T$  ( $V$  orthogonal,  $D$  diagonal):

$$A = \underbrace{QP}_{U} V D V^T = UDV^T$$

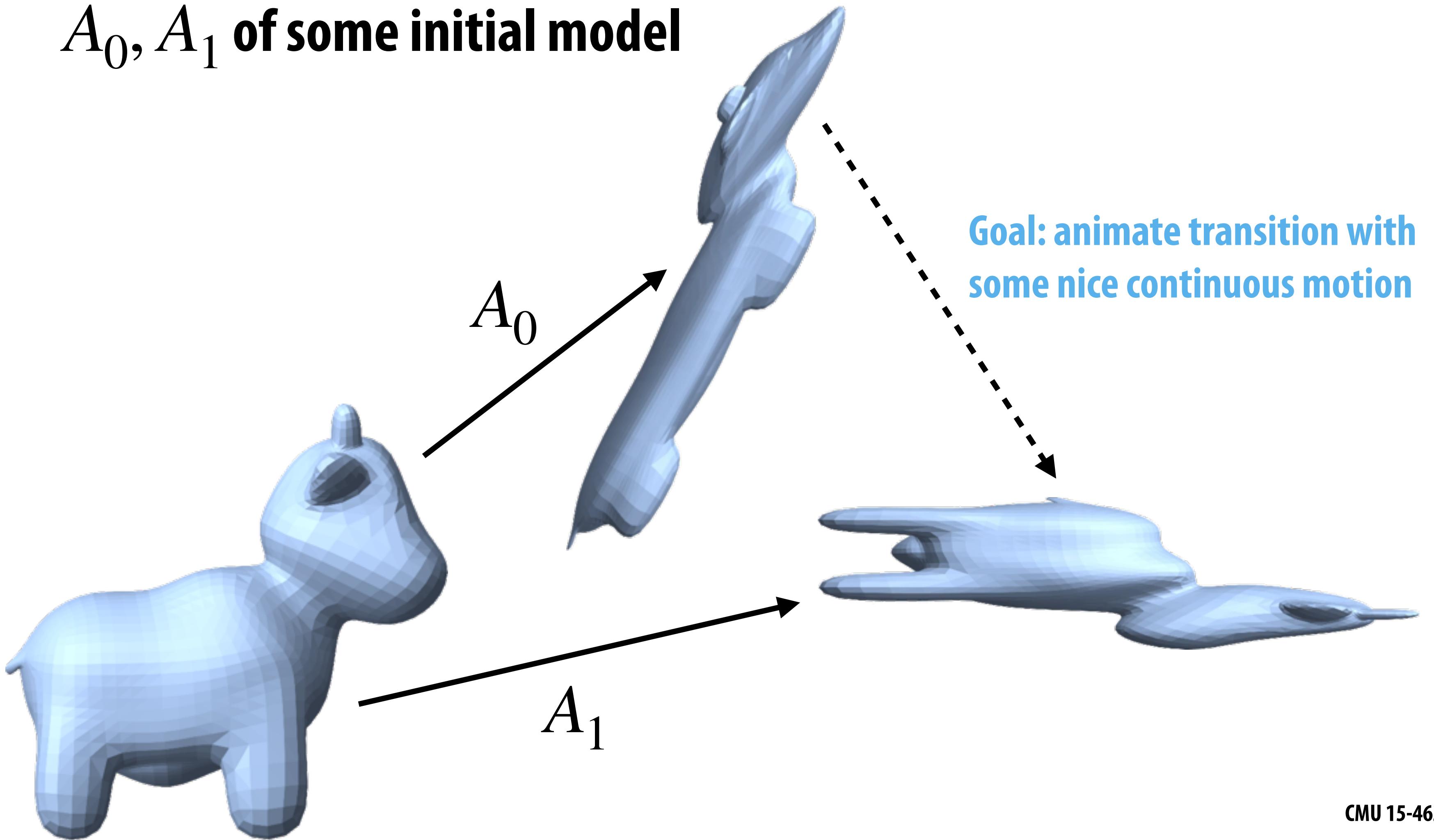
rotation      rotation  
axis-aligned scaling



Result  $UDV^T$  is called the singular value decomposition

# Interpolating Transformations

- How are these decompositions useful for graphics?
- Consider interpolating between two linear transformations  $A_0, A_1$  of some initial model



# Interpolating Transformations—Linear

One idea: just take a linear combination of the two matrices,  
weighted by the current time  $t \in [0, 1]$

$$A(t) = (1 - t)A_0 + tA_1$$



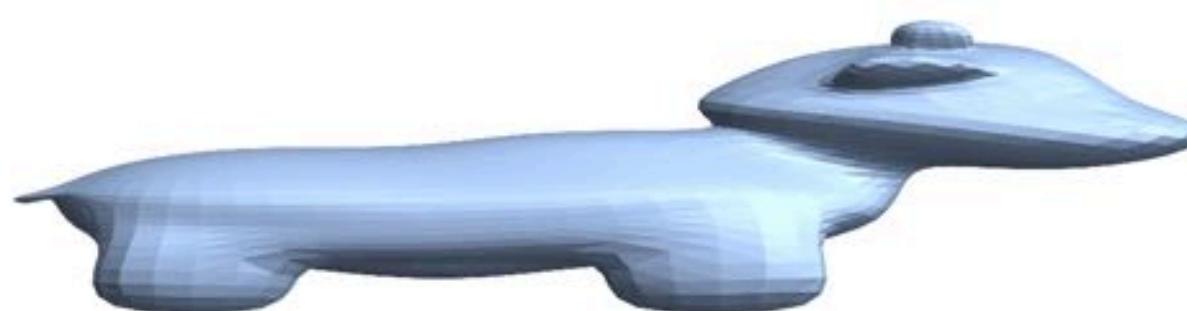
Hits the right start/endpoints... but looks awful in between!

# Interpolating Transformations—Polar

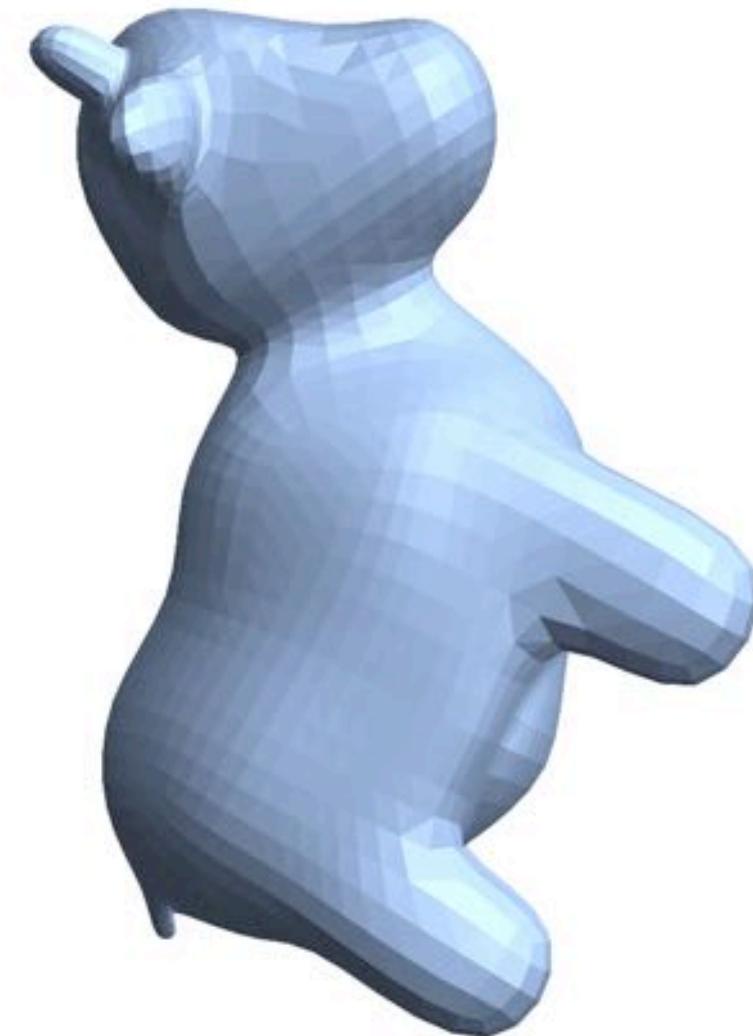
Better idea: separately interpolate components of polar decomposition.

$$A_0 = Q_0 P_0, \quad A_1 = Q_1 P_1$$

scaling



rotation



final interpolation



$$P(t) = (1 - t)P_0 + tP_1$$

$$\widetilde{Q}(t) = (1 - t)Q_0 + tQ_1$$

$$A(t) = Q(t)P(t)$$

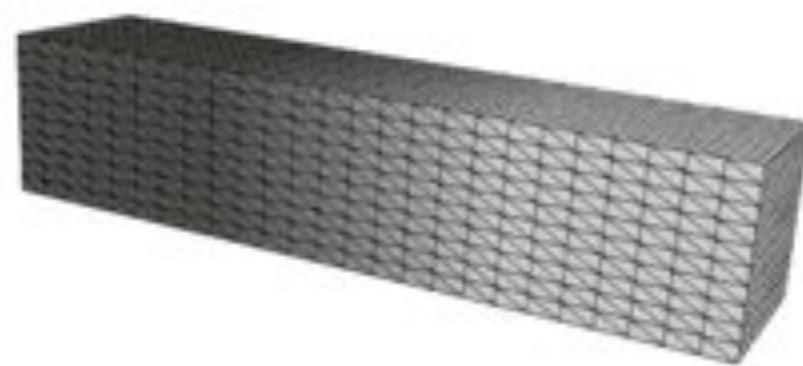
$$\widetilde{Q}(t) = Q(t)X(t)$$

**...looks better!**

# Example: Linear Blend Skinning

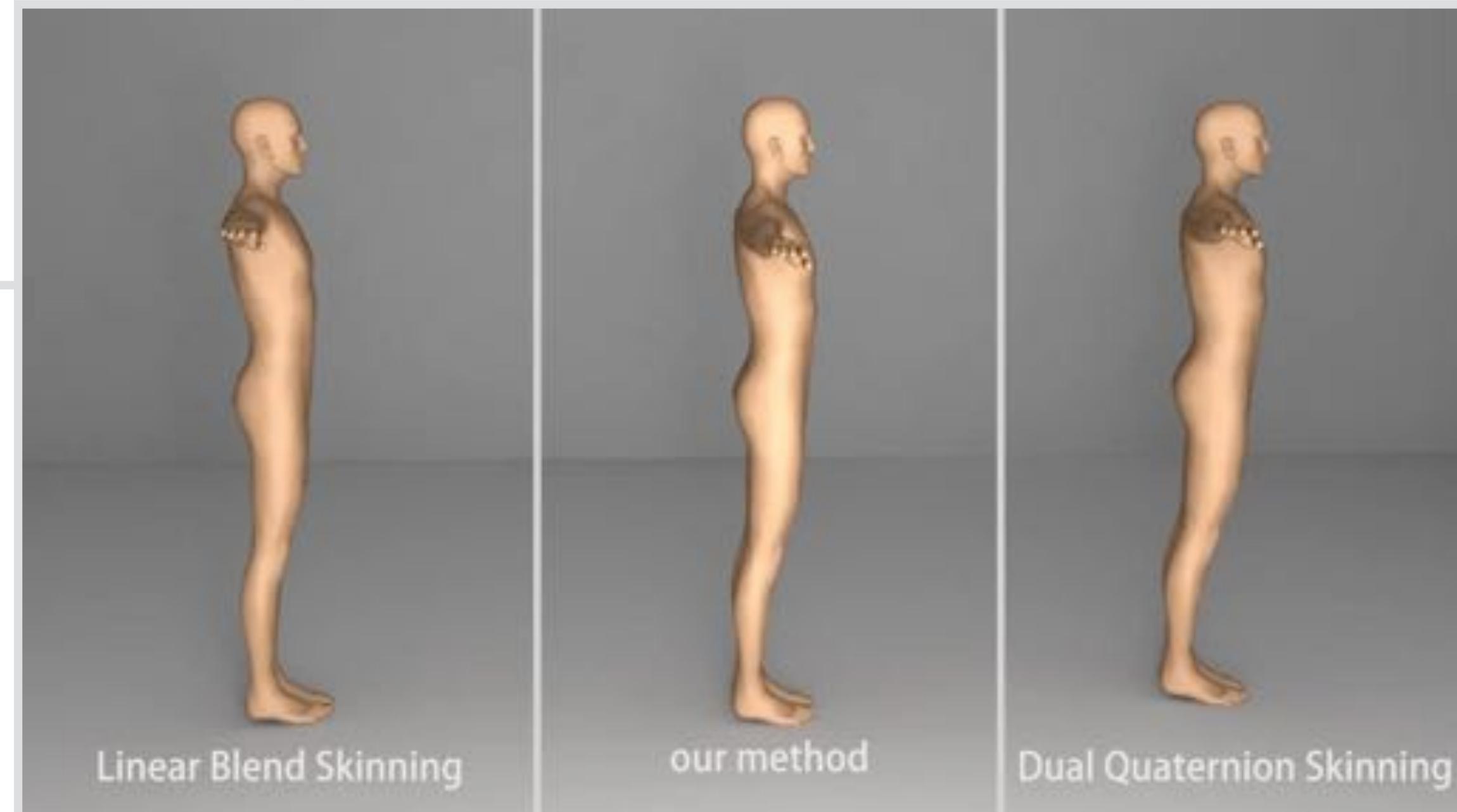
- Naïve linear interpolation also causes artifacts when blending between transformations on a character (“candy wrapper effect”)
- Lots of research on alternative ways to blend transformations...

LBS: candy-wrapper artifact



Rumman & Fratarcangeli (2015)  
“Position-based Skinning for Soft Articulated Characters”

Jacobson, Deng, Kavan, & Lewis (2014)  
“Skinning: Real-time Shape Deformation”



# Translations

- So far we've ignored a basic transformation—translations
- A translation simply adds an offset  $\mathbf{u}$  to the given point  $\mathbf{x}$ :

$$f_{\mathbf{u}}(\mathbf{x}) = \mathbf{x} + \mathbf{u}$$

**Q: Is this transformation linear?**  
**(Certainly seems to move us along a line...)**

Let's carefully check the definition...

additivity

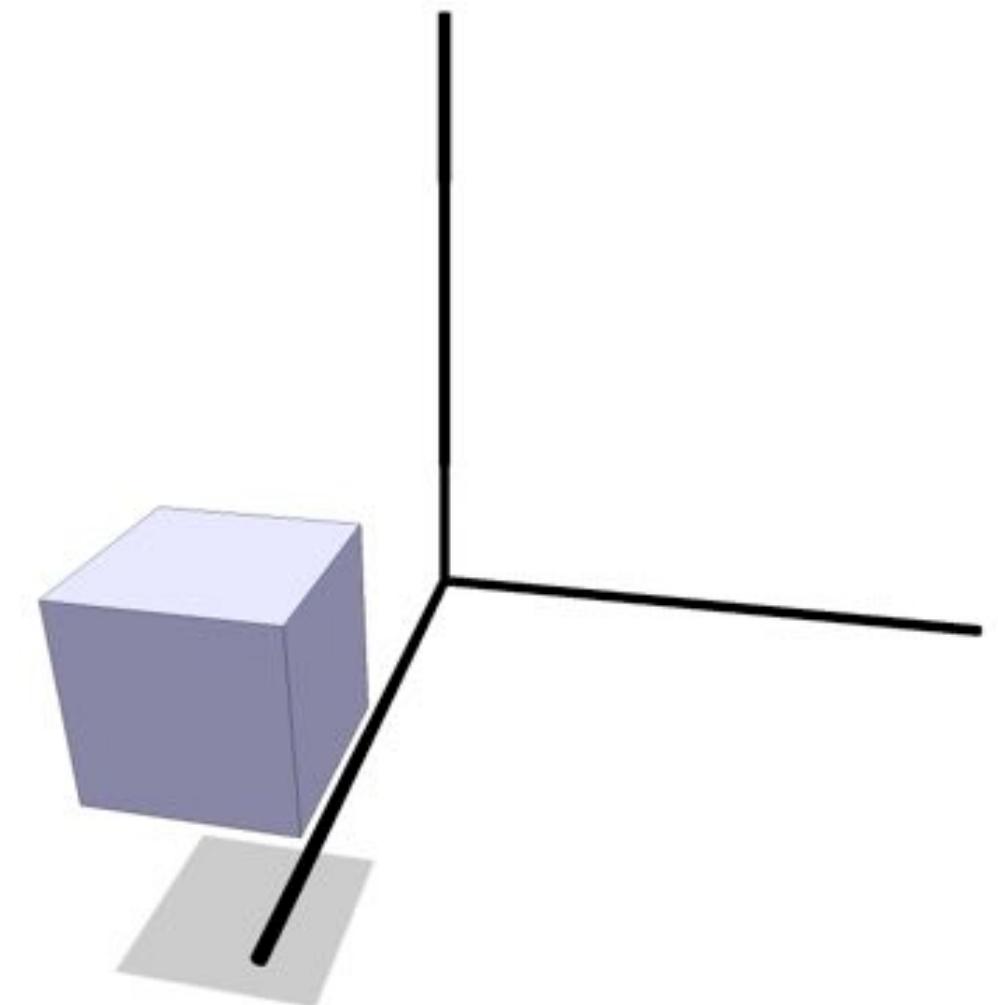
$$f_{\mathbf{u}}(\mathbf{x} + \mathbf{y}) = \mathbf{x} + \mathbf{y} + \mathbf{u}$$

$$f_{\mathbf{u}}(\mathbf{x}) + f_{\mathbf{u}}(\mathbf{y}) = \mathbf{x} + \mathbf{y} + 2\mathbf{u}$$

homogeneity

$$f_{\mathbf{u}}(a\mathbf{x}) = a\mathbf{x} + \mathbf{u}$$

$$af_{\mathbf{u}}(\mathbf{x}) = ax + a\mathbf{u}$$



**A: No! Translation is affine, not linear!**

# Composition of Transformations

- Recall we can compose linear transformations via matrix multiplication:

$$A_3(A_2(A_1 \mathbf{x})) = (A_3 A_2 A_1) \mathbf{x}$$

- It's easy enough to compose translations—just add vectors:

$$f_{\mathbf{u}_3}(f_{\mathbf{u}_2}(f_{\mathbf{u}_1}(\mathbf{x}))) = f_{\mathbf{u}_1 + \mathbf{u}_2 + \mathbf{u}_3}(\mathbf{x})$$

- What if we want to intermingle translations and linear transformations (rotation, scale, shear, etc.)?

$$A_2(A_1 \mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2 = (A_2 A_1) \mathbf{x} + (A_2 \mathbf{b}_1 + \mathbf{b}_2)$$

- Now we have to keep track of a matrix and a vector
- Moreover, we'll see (later) that this encoding won't work for other important cases, such as perspective transformations

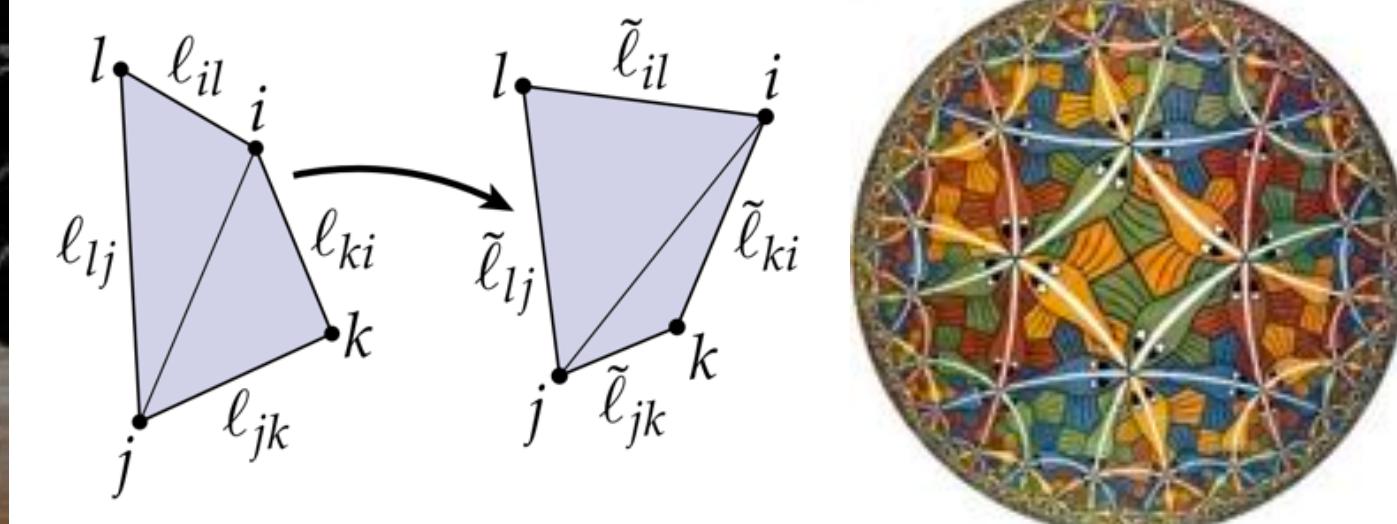
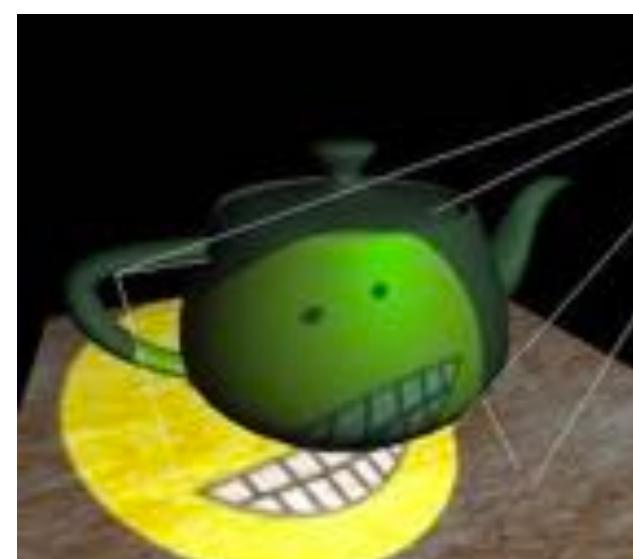
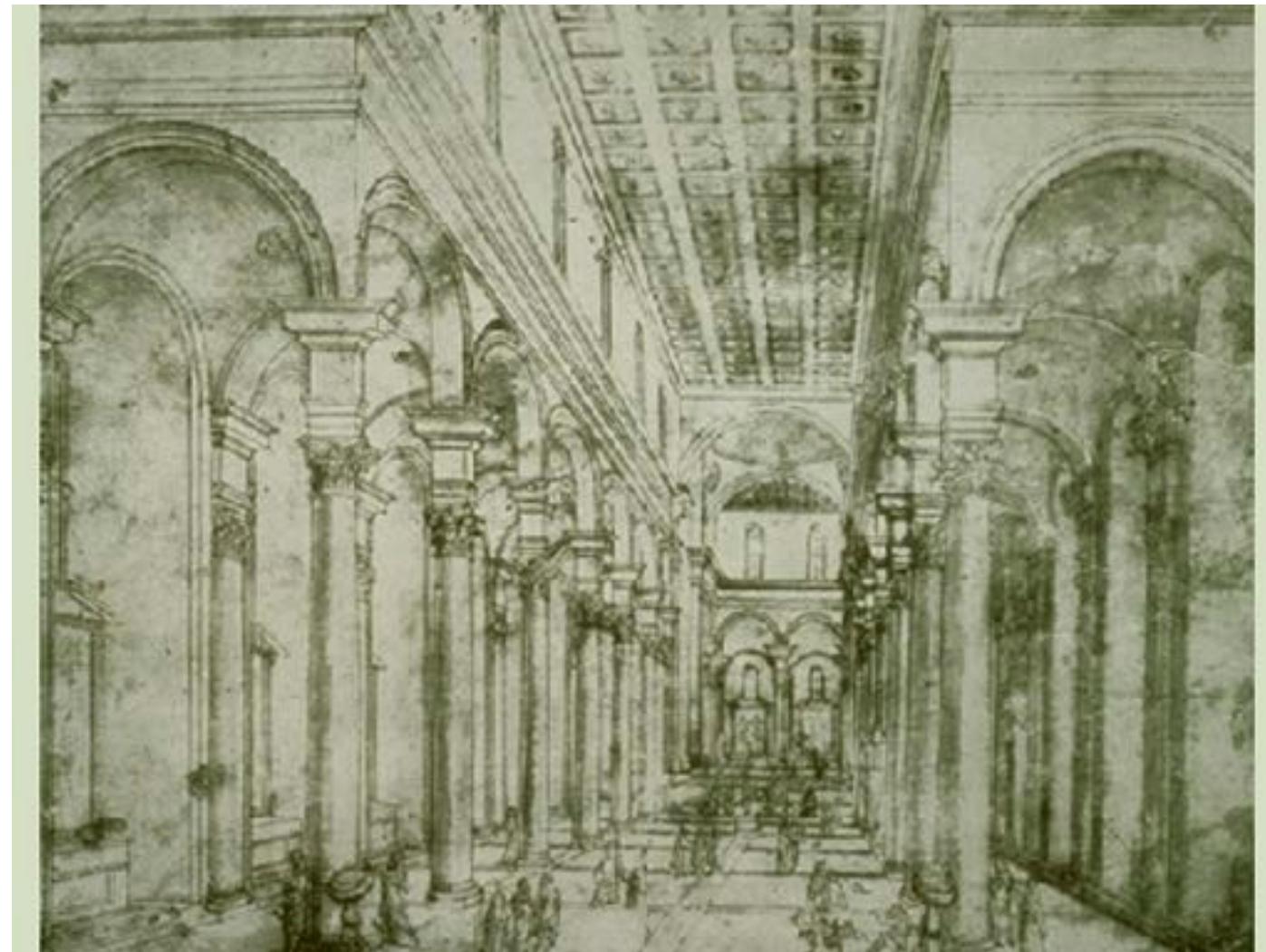
But there is a better way...

**Strange idea:  
Maybe translations turn into linear  
transformations if we go into the  
4th dimension...!**



# Homogeneous Coordinates

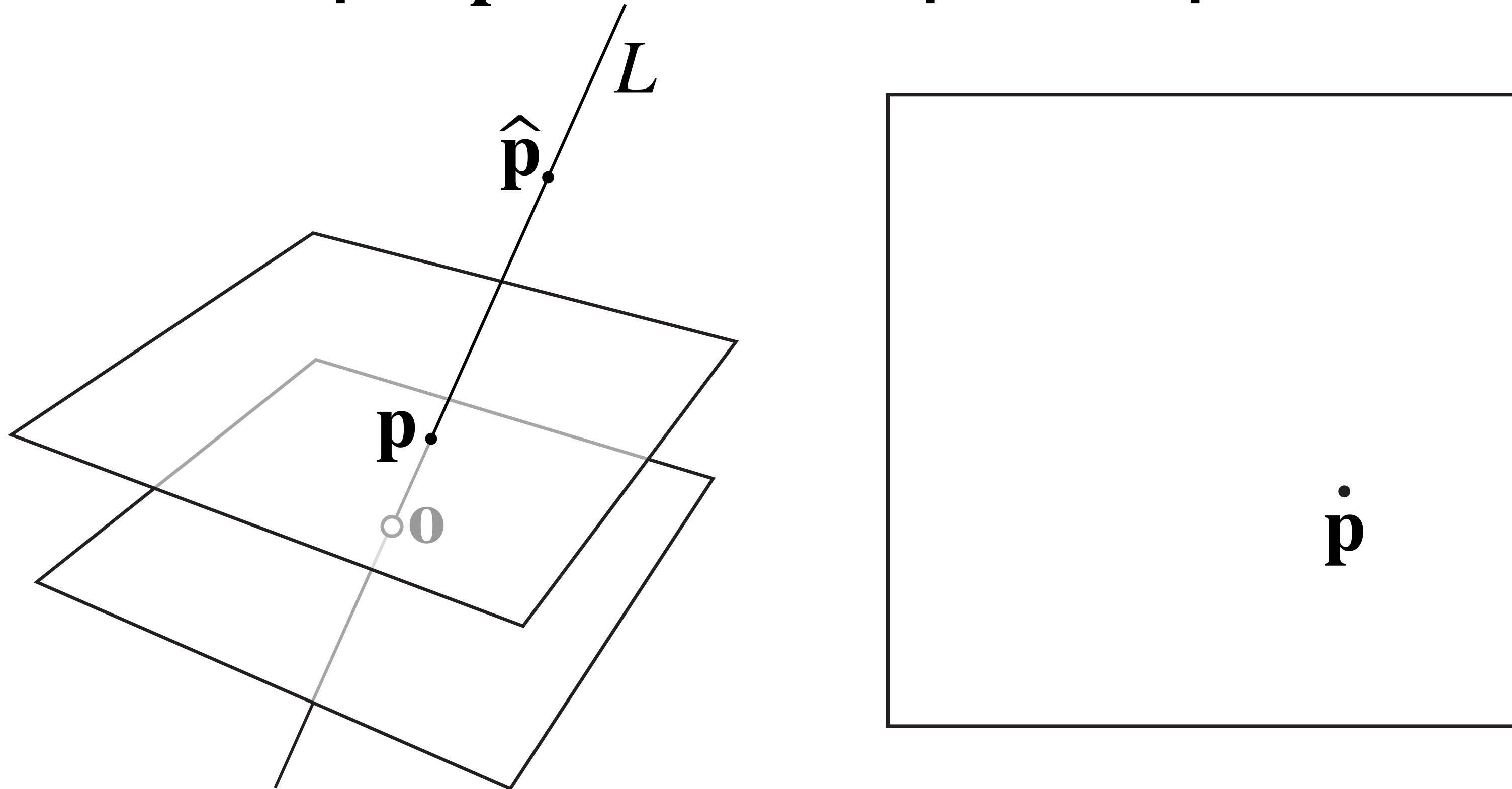
- Came from efforts to study perspective
- Introduced by Möbius as a natural way of assigning coordinates to lines
- Show up naturally in a surprising large number of places in computer graphics:
  - 3D transformations
  - perspective projection
  - quadric error simplification
  - premultiplied alpha
  - shadow mapping
  - projective texture mapping
  - discrete conformal geometry
  - hyperbolic geometry
  - clipping
  - directional lights
  - ...



Probably worth understanding!

# Homogeneous Coordinates—Basic Idea

- Consider any 2D plane that does not pass through the origin  $o$  in 3D
- Every line through the origin in 3D corresponds to a point in the 2D plane
  - Just find the point  $p$  where the line  $L$  pierces the plane

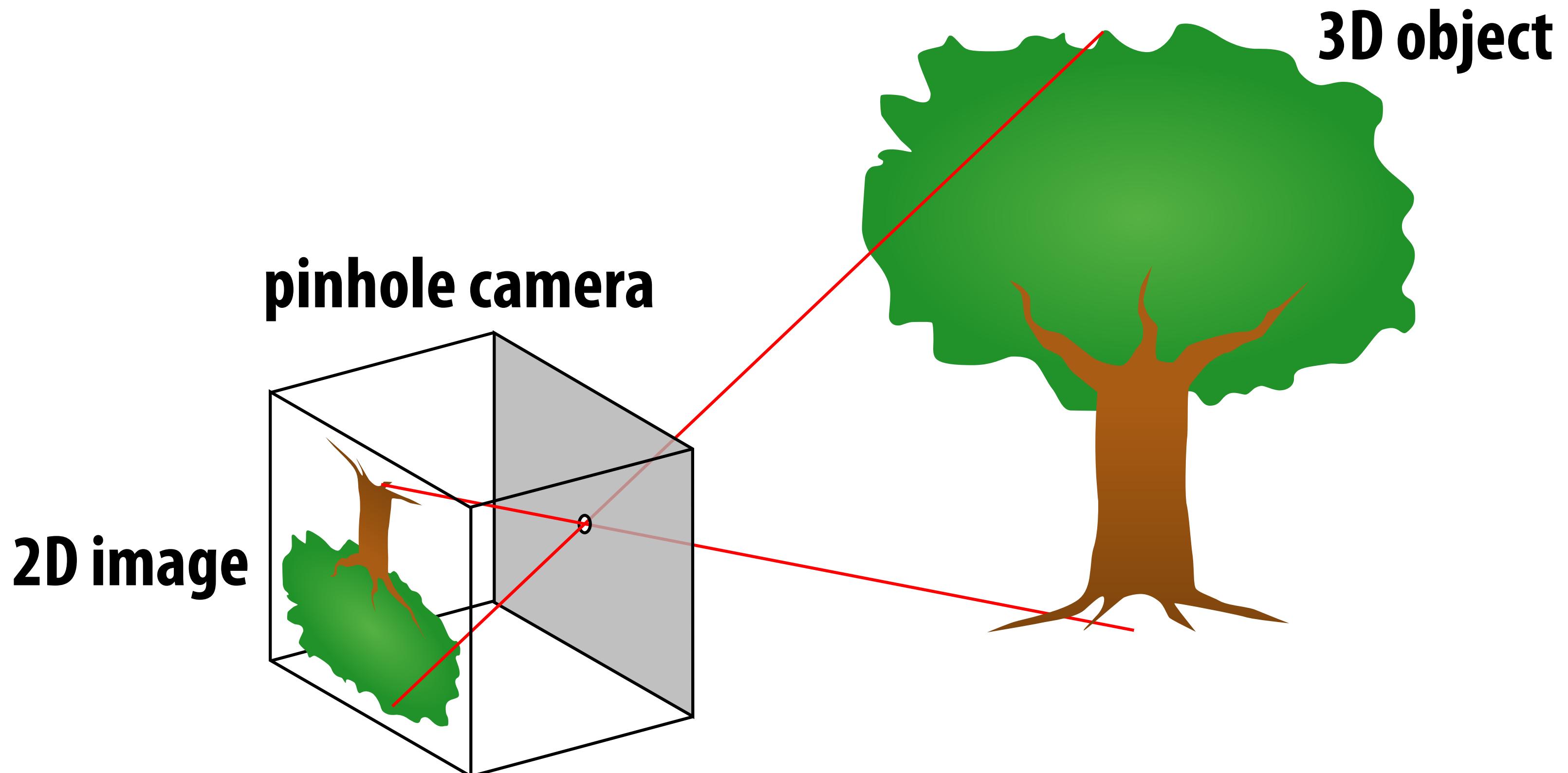


Hence, any point  $\hat{p}$  on the line  $L$  can be used to represent the point  $p$ .

**Q: What does this story remind you of?**

# Review: Perspective projection

- Hopefully it reminds you of our “pinhole camera”
- Objects along the same line project to the same point



If you have an image of a single dot, can't know where it is!  
Only which line it belongs to.

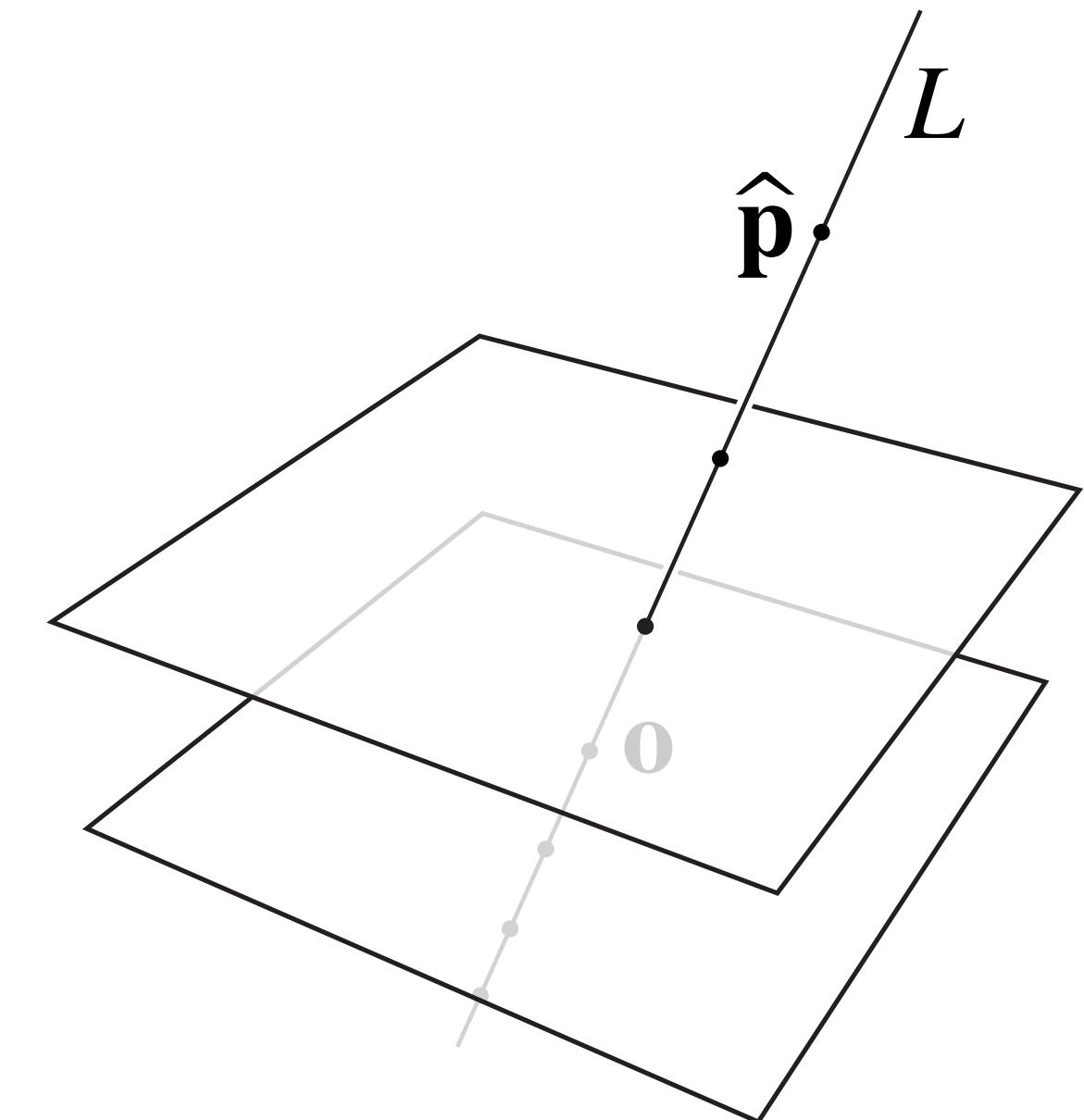
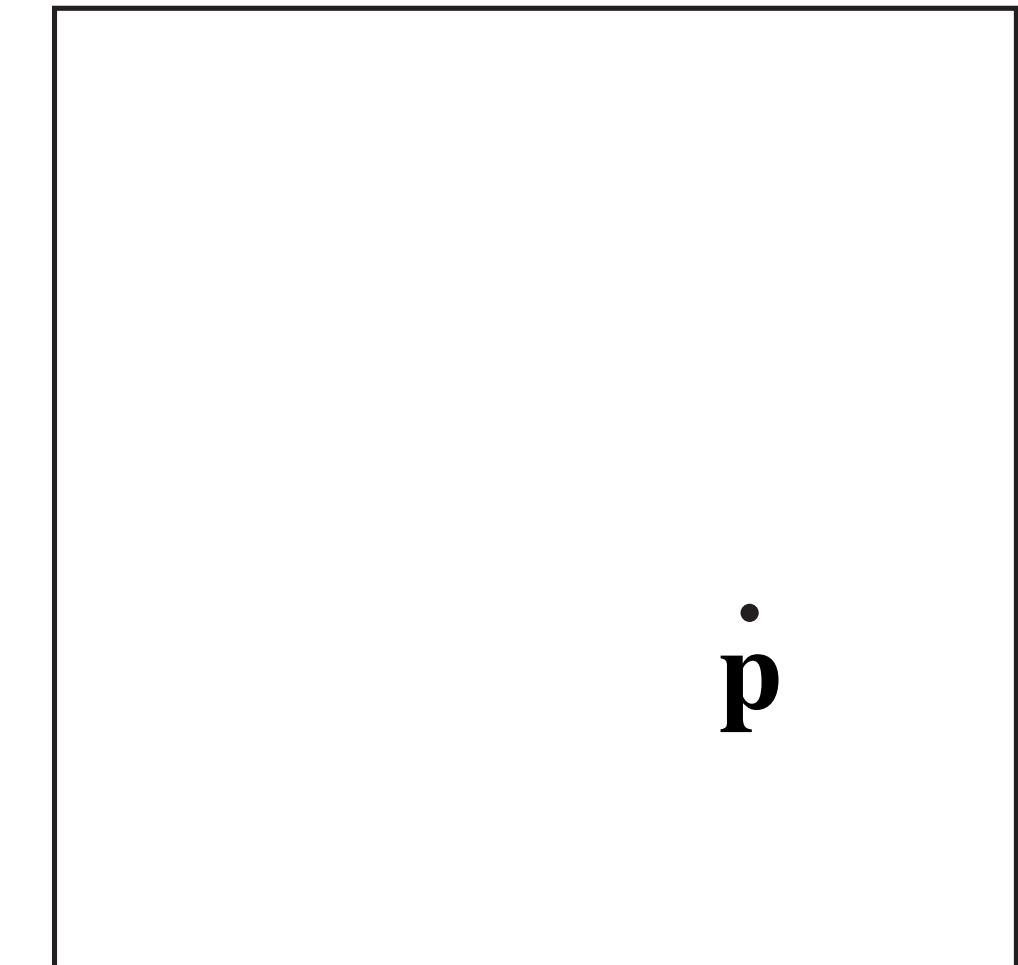
# Homogeneous Coordinates (2D)

- More explicitly, consider a point  $p = (x, y)$ , and the plane  $z = 1$  in 3D

- Any three numbers  $\hat{p} = (a, b, c)$  such that  $(a/c, b/c) = (x, y)$  are homogeneous coordinates for  $p$

- E.g.,  $(x, y, 1)$
- In general:  $(cx, cy, c)$  for  $c \neq 0$

- Hence, two points  $\hat{p}, \hat{q} \in \mathbb{R}^3 \setminus \{O\}$  describe the same point in 2D (and line in 3D) if  $\hat{p} = \lambda \hat{q}$  for some  $\lambda \neq 0$

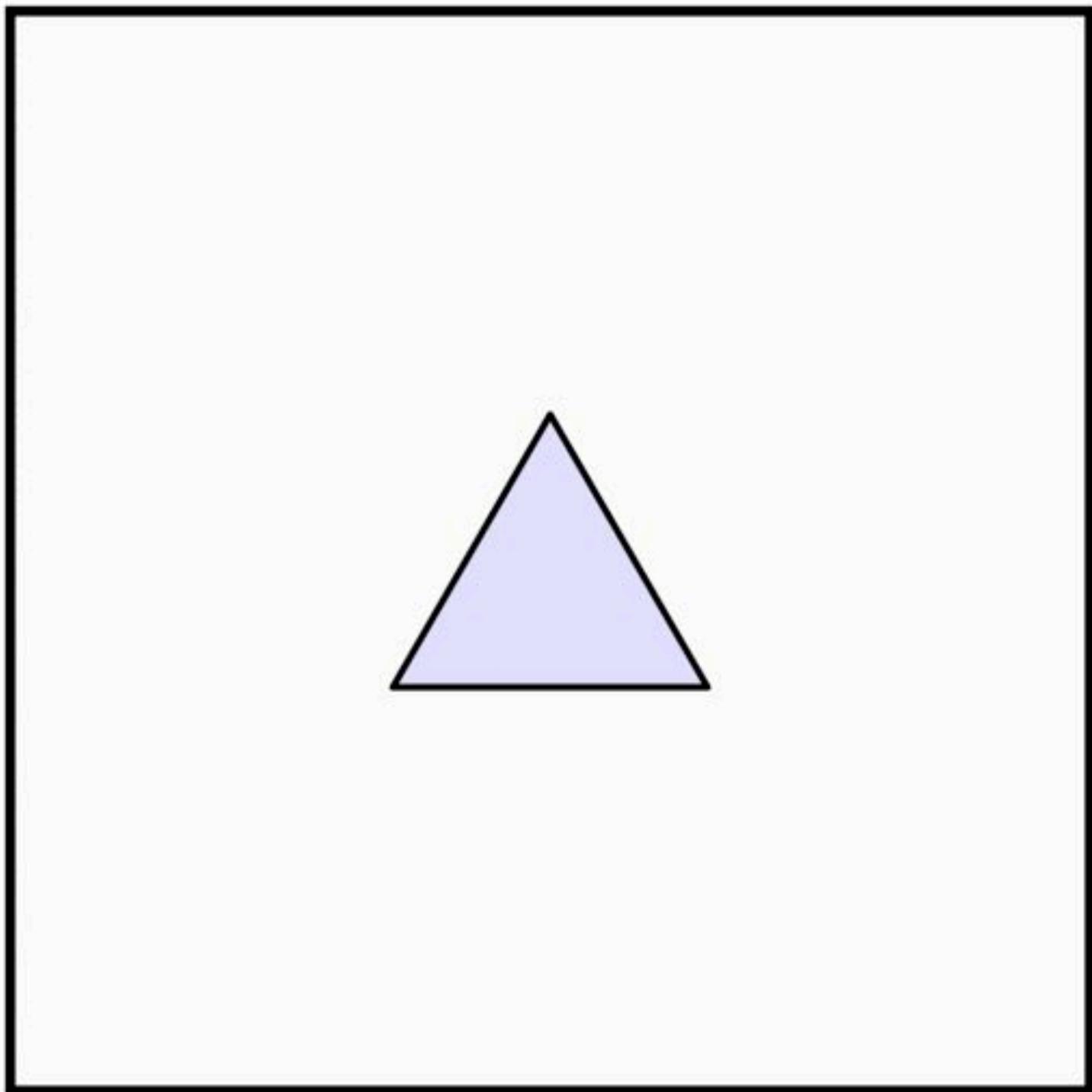


Great... but how does this help us with transformations?

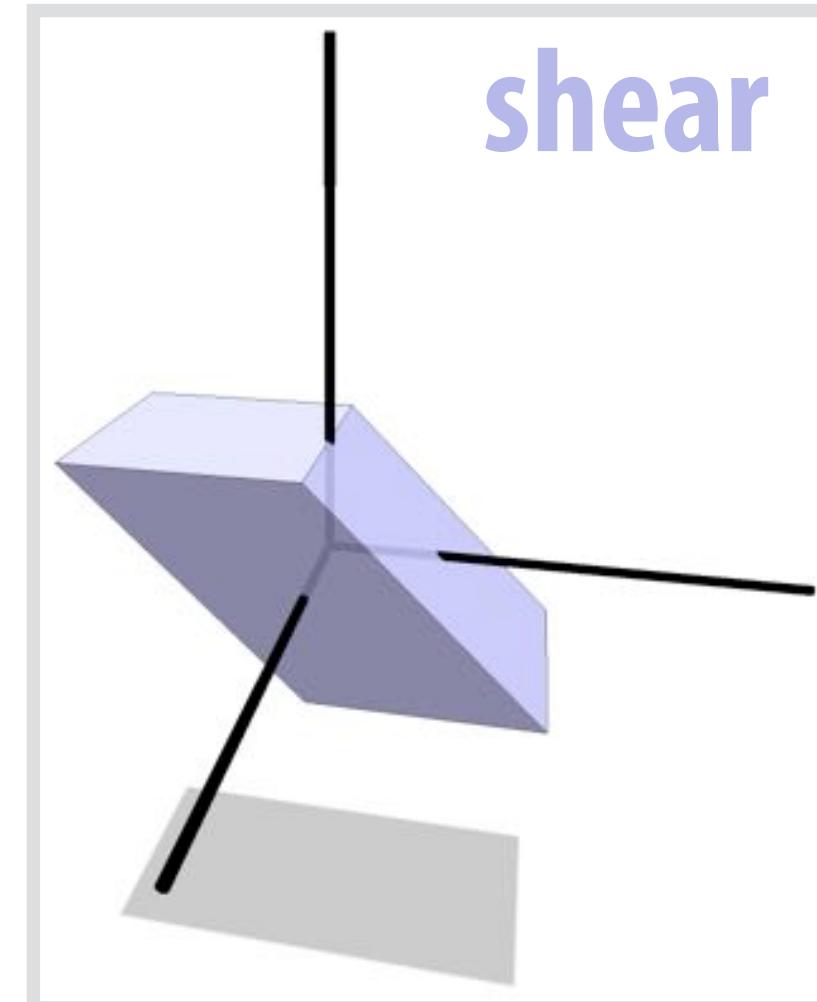
# Translation in Homogeneous Coordinates

Let's think about what happens to our homogeneous coordinates  $\hat{p}$  if we apply a translation to our 2D coordinates  $p$

2D coordinates



Q: What kind of transformation does this look like?



# Translation in Homogeneous Coordinates

- But wait a minute—shear is a linear transformation!
- Can this be right? Let's check in coordinates...
- Suppose we translate a point  $\mathbf{p} = (p_1, p_2)$  by a vector  $\mathbf{u} = (u_1, u_2)$  to get  $\mathbf{p}' = (p_1 + u_1, p_2 + u_2)$
- The homogeneous coordinates  $\hat{\mathbf{p}} = (cp_1, cp_2, c)$  then become  $\hat{\mathbf{p}}' = (cp_1 + cu_1, cp_2 + cu_2, c)$
- Notice that we're shifting  $\hat{\mathbf{p}}$  by an amount  $c\mathbf{u}$  that's proportional to the distance  $c$  along the third axis—a shear

Using homogeneous coordinates, we can represent an affine transformation in 2D as a linear transformation in 3D

# Homogeneous Translation—Matrix Representation

- To write as a matrix, recall that a shear in the direction  $\mathbf{u} = (u_1, u_2)$  according to the distance along a direction  $\mathbf{v}$  is

$$f_{\mathbf{u}, \mathbf{v}}(\mathbf{x}) = \mathbf{x} + \langle \mathbf{v}, \mathbf{x} \rangle \mathbf{u}$$

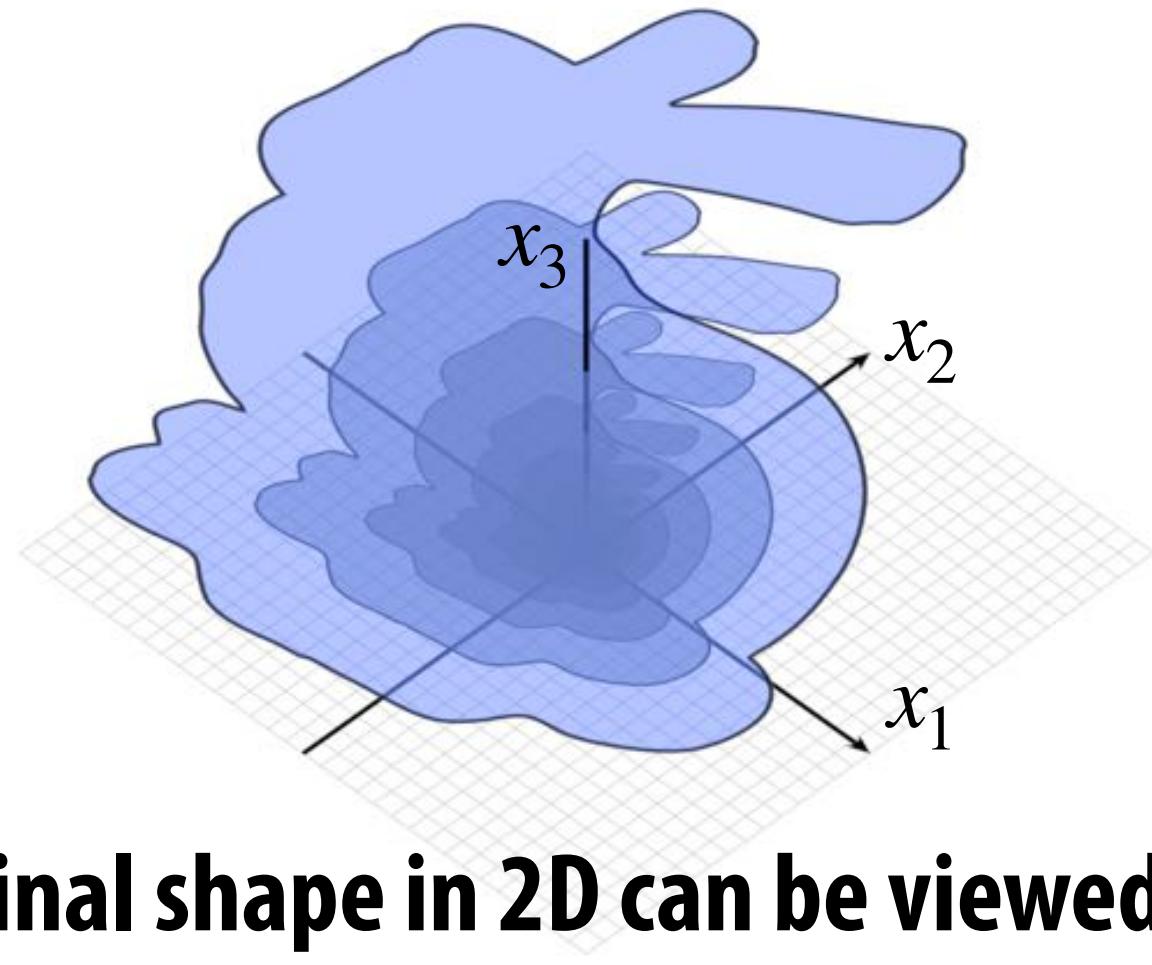
- In matrix form:

$$f_{\mathbf{u}, \mathbf{v}}(\mathbf{x}) = (I + \mathbf{u}\mathbf{v}^T) \mathbf{x}$$

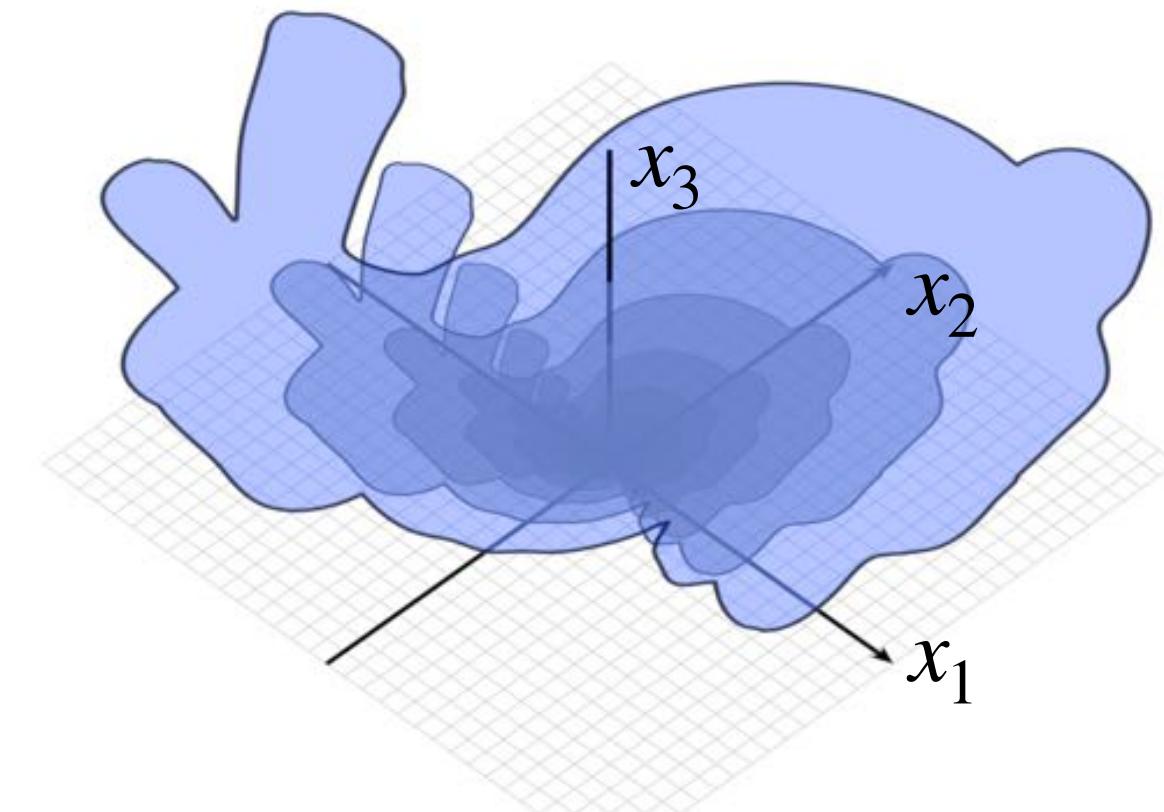
- In our case,  $\mathbf{v} = (0, 0, 1)$  and so we get a matrix

$$\begin{bmatrix} 1 & 0 & u_1 \\ 0 & 1 & u_2 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} cp_1 \\ cp_2 \\ c \end{bmatrix} = \begin{bmatrix} c(p_1 + u_1) \\ c(p_2 + u_2) \\ c \end{bmatrix} \xrightarrow{1/c} \begin{bmatrix} p_1 + u_1 \\ p_2 + u_2 \\ 1 \end{bmatrix}$$

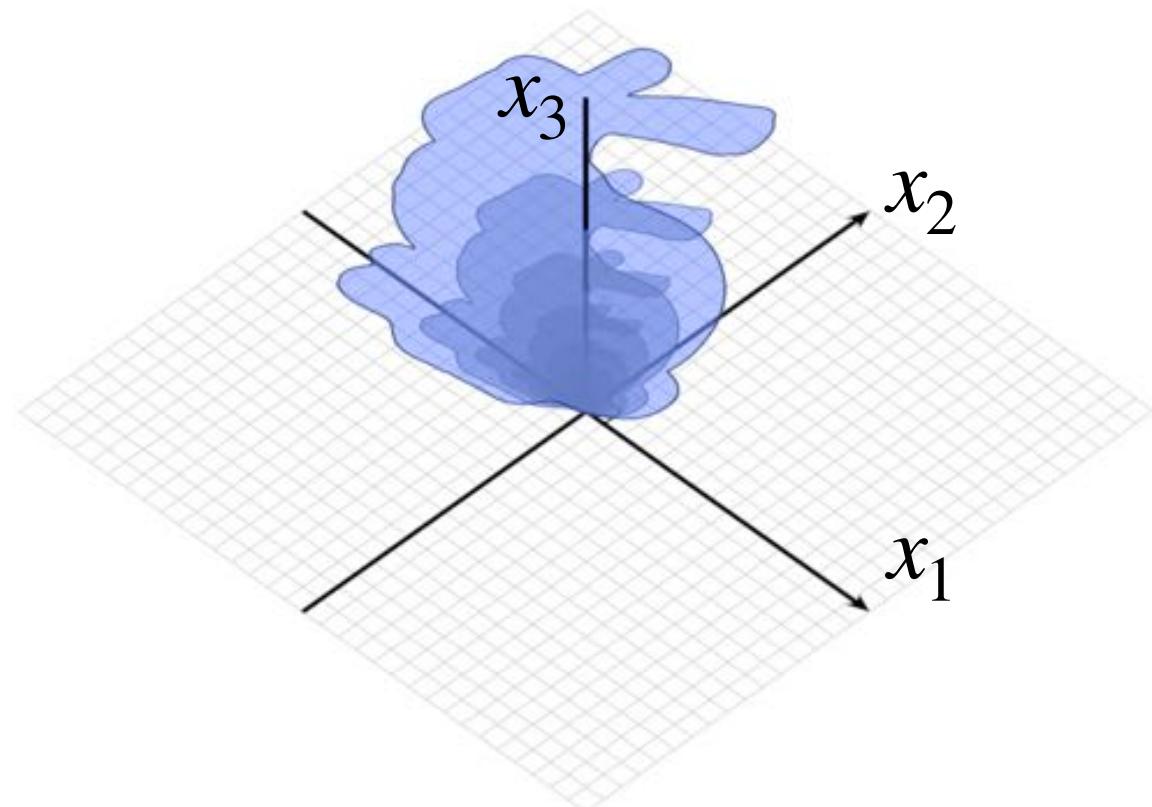
# Other 2D Transformations in Homogeneous Coordinates



**Original shape in 2D can be viewed as many copies, uniformly scaled by  $x_3$**

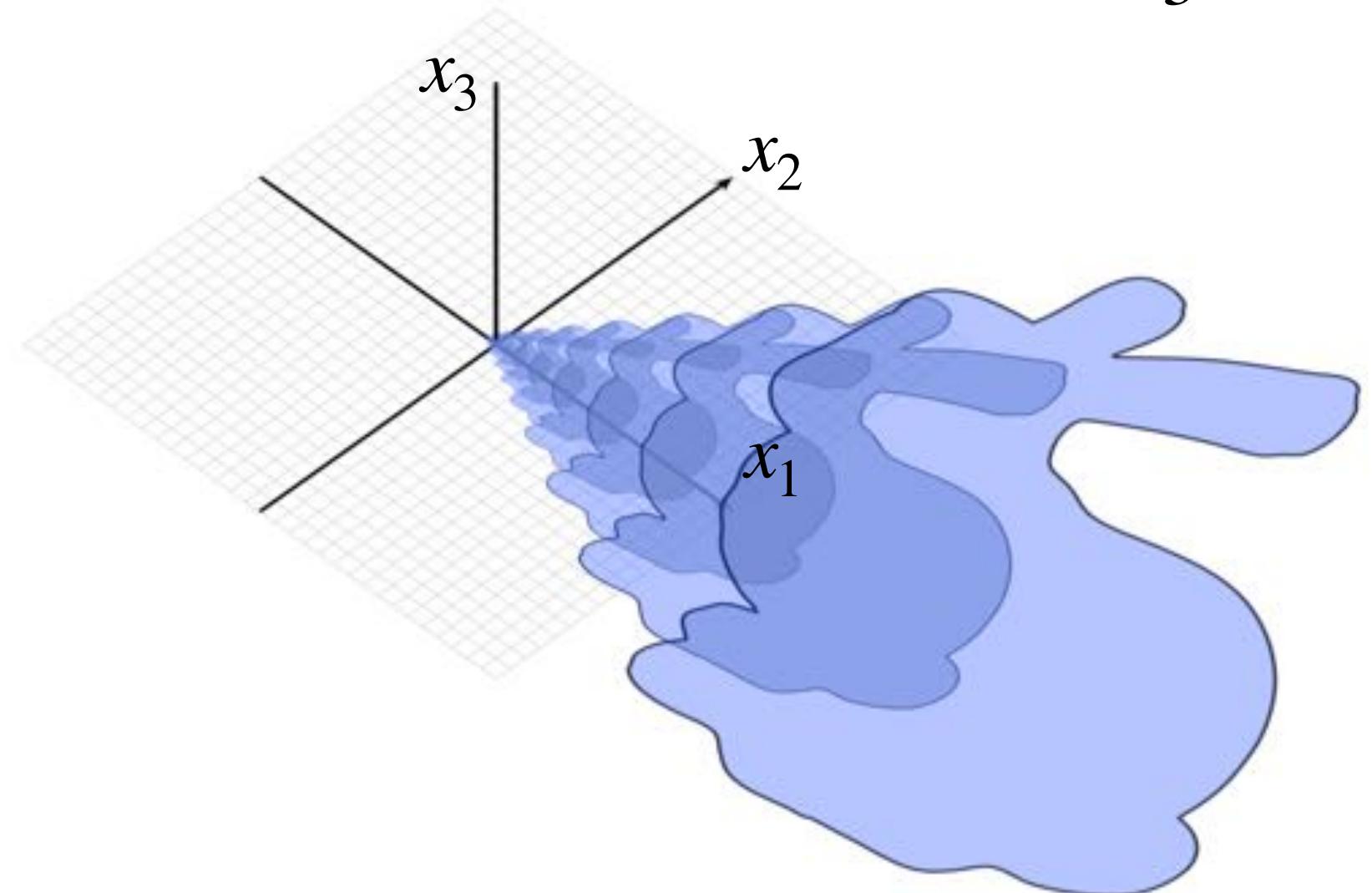


**2D rotation  $\leftrightarrow$  rotate around  $x_3$**



**2D scale  $\leftrightarrow$  scale  $x_1$  and  $x_2$ ; preserve  $x_3$**

**(Q: what happens to 2D shape if you scale  $x_1$ ,  $x_2$ , and  $x_3$  uniformly?)**



**2D translate  $\leftrightarrow$  shear**

**Now easy to compose all these transformations**

# 3D Transformations in Homogeneous Coordinates

- Not much changes in three (or more) dimensions: just append one “homogeneous coordinate” to the first three
- Matrix representations of 3D linear transformations just get an additional identity row/column; translation is again a shear

point in 3D

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

rotate  $(x, y, z)$  around  $y$  by  $\theta$

$$\begin{bmatrix} \cos \theta & 0 & \sin \theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin \theta & 0 & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

shear  $(x, y)$  by  $z$  in  $(s, t)$  direction

$$\begin{bmatrix} 1 & 0 & s & 0 \\ 0 & 1 & t & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

scale  $x, y, z$  by  $a, b, c$

$$\begin{bmatrix} a & 0 & 0 & 0 \\ 0 & b & 0 & 0 \\ 0 & 0 & c & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

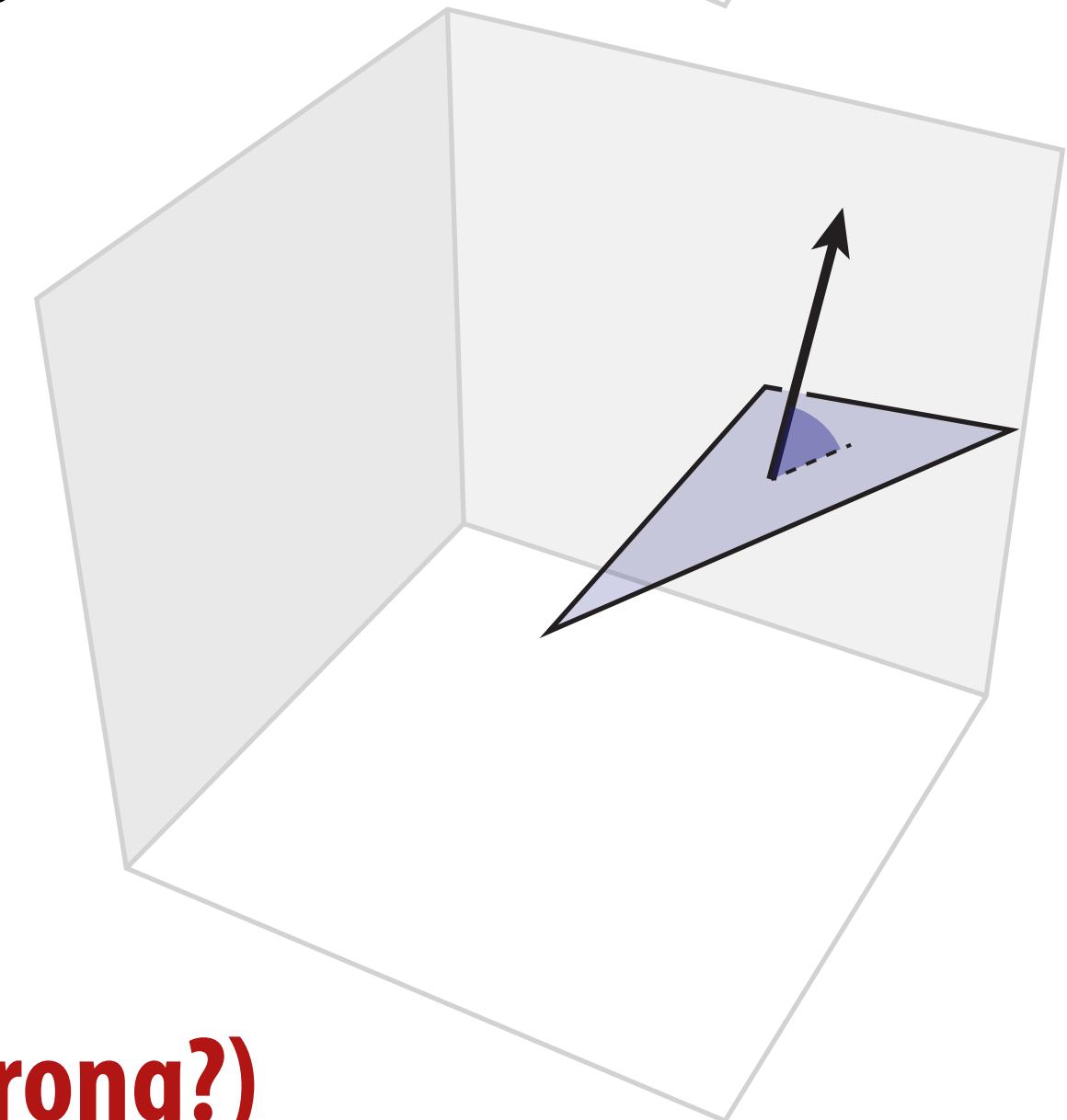
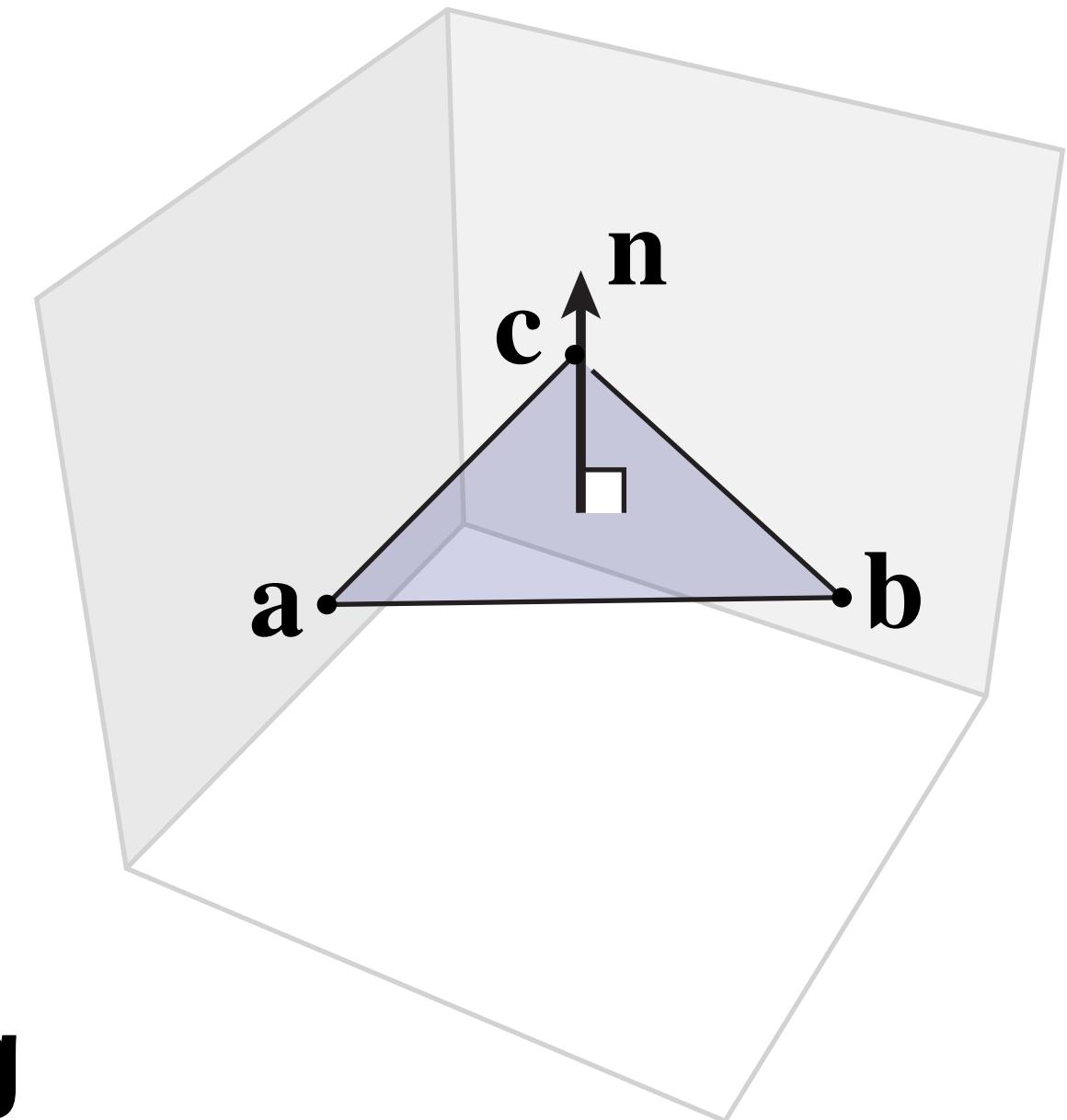
translate  $(x, y, z)$  by  $(u, v, w)$

$$\begin{bmatrix} 1 & 0 & 0 & u \\ 0 & 1 & 0 & v \\ 0 & 0 & 1 & w \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

# Points vs. Vectors

- Homogeneous coordinates have another useful feature: distinguish between points and vectors
- Consider for instance a triangle with:
  - vertices  $a, b, c \in \mathbb{R}^3$
  - normal vector  $n \in \mathbb{R}^3$
- Suppose we transform the triangle by appending “1” to  $a, b, c, n$  and multiplying by this matrix:

$$\begin{bmatrix} \cos \theta & 0 & \sin \theta & u \\ 0 & 1 & 0 & v \\ -\sin \theta & 0 & \cos \theta & w \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



Normal is not orthogonal to triangle! (What went wrong?)

# Points vs. Vectors (continued)

- Let's think about what happens when we multiply the normal vector  $\mathbf{n}$  by our matrix:

rotate normal around  $y$  by  $\theta$

$$\begin{bmatrix} \cos \theta & 0 & \sin \theta \\ 0 & 1 & 0 \\ -\sin \theta & 0 & \cos \theta \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} u \\ v \\ w \\ 1 \end{bmatrix} = \begin{bmatrix} n_1 \\ n_2 \\ n_3 \\ 1 \end{bmatrix}$$

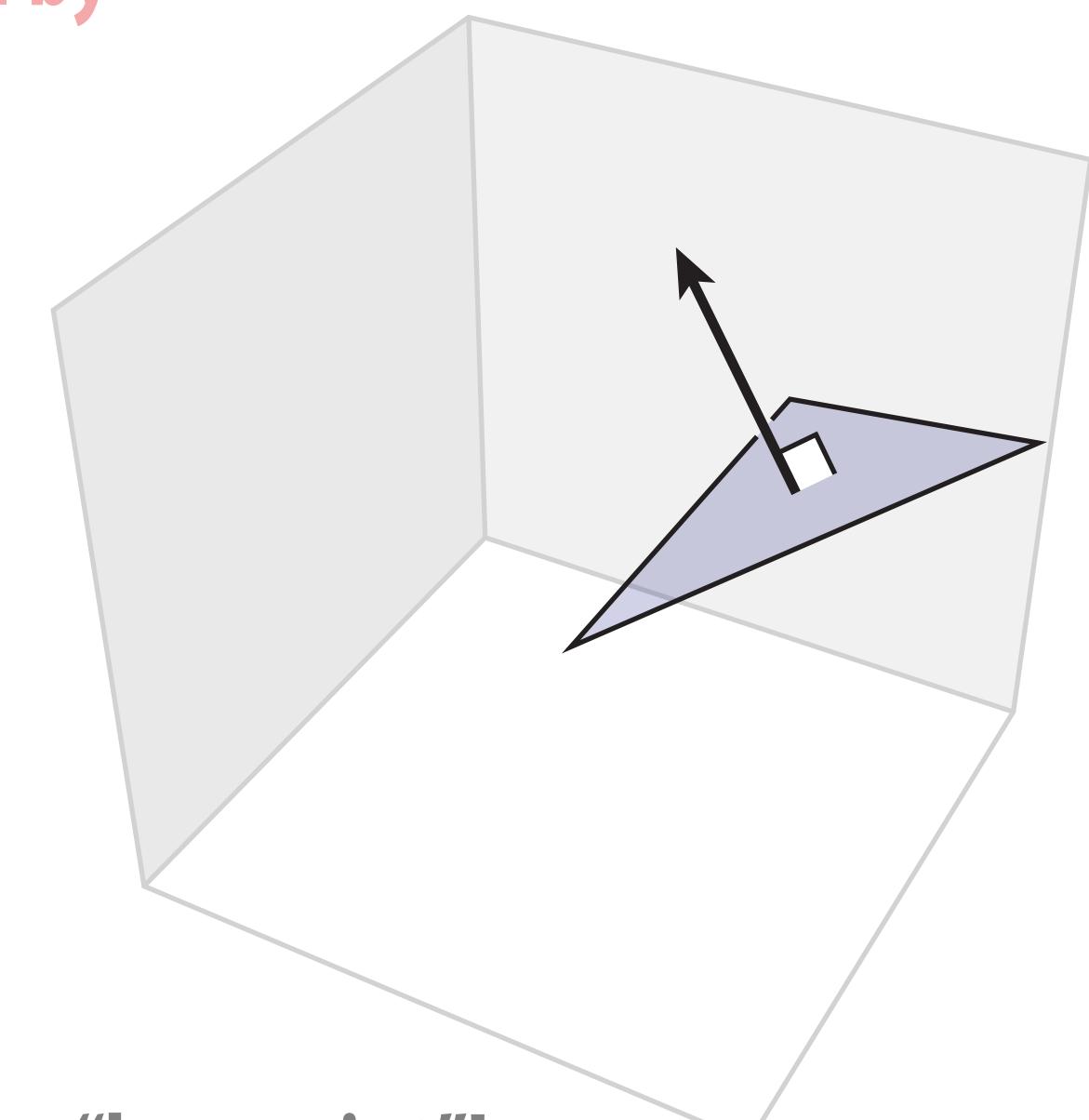
- But when we rotate/translate a triangle, its normal should just rotate!\*

translate normal by  
 $(u, v, w)$

- Solution? Just set homogeneous coordinate to zero!

$$\begin{bmatrix} n_1 \\ n_2 \\ n_3 \\ 0 \end{bmatrix}$$

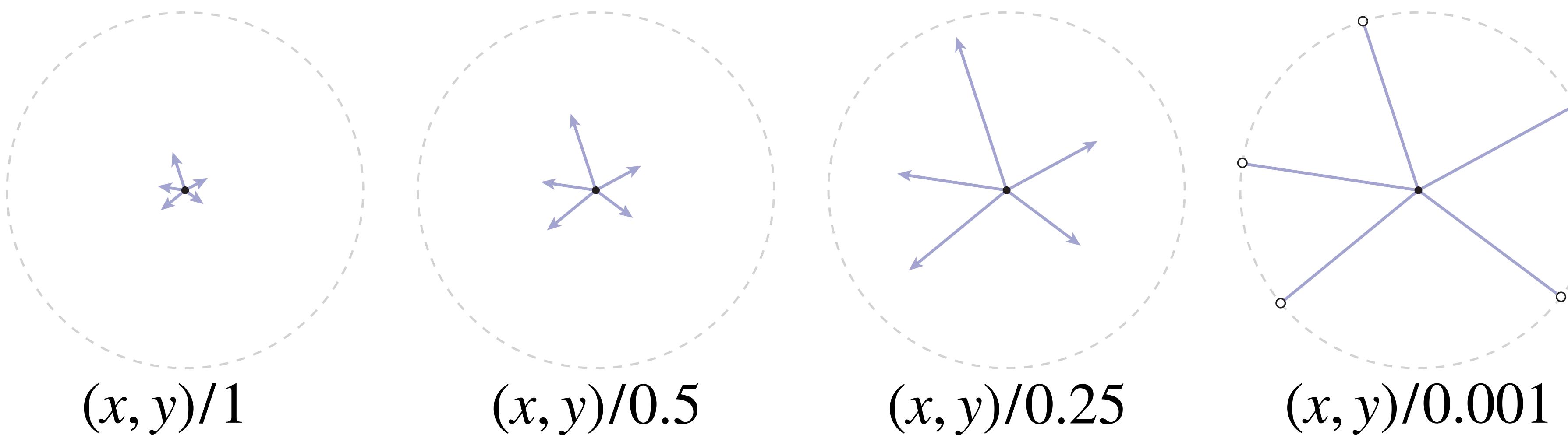
- Translation now gets ignored; normal is orthogonal to triangle



\*Recall that vectors just have direction and magnitude—they don't have a “basepoint”!

# Points vs. Vectors in Homogeneous Coordinates

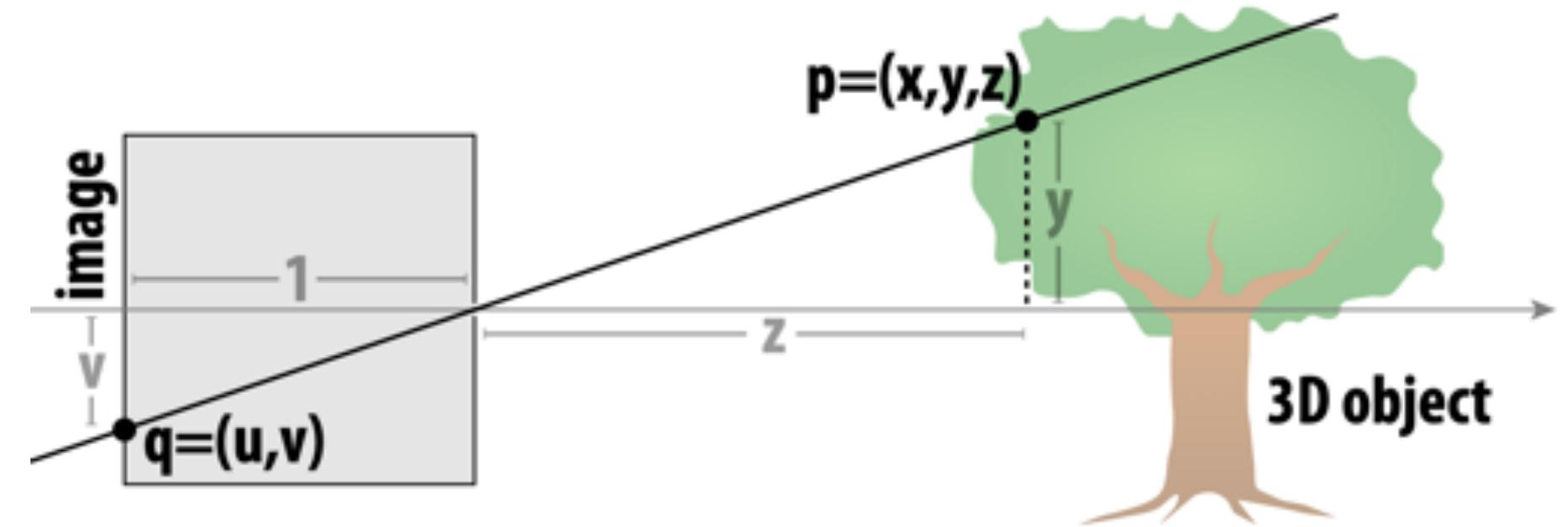
- In general:
  - A point has a nonzero homogeneous coordinate ( $c = 1$ )
  - A vector has a zero homogeneous coordinate ( $c = 0$ )
- But wait... what division by  $c$  mean when it's equal to zero?
- Well consider what happens as  $c \rightarrow 0...$



Can think of vectors as “points at infinity” (sometimes called “ideal points”)  
(In practice: still need to check for divide by zero!)

# Perspective Projection in Homogeneous Coordinates

- Q: How can we perform perspective projection\* using homogeneous coordinates?
- Remember from our pinhole camera model that the basic idea was to “divide by  $z$ ”
- So, we can build a matrix that “copies” the  $z$  coordinate into the homogeneous coordinate
- Division by the homogeneous coordinate now gives us perspective projection onto the plane  $z = 1$



$$(x, y, z) \mapsto (x/z, y/z)$$

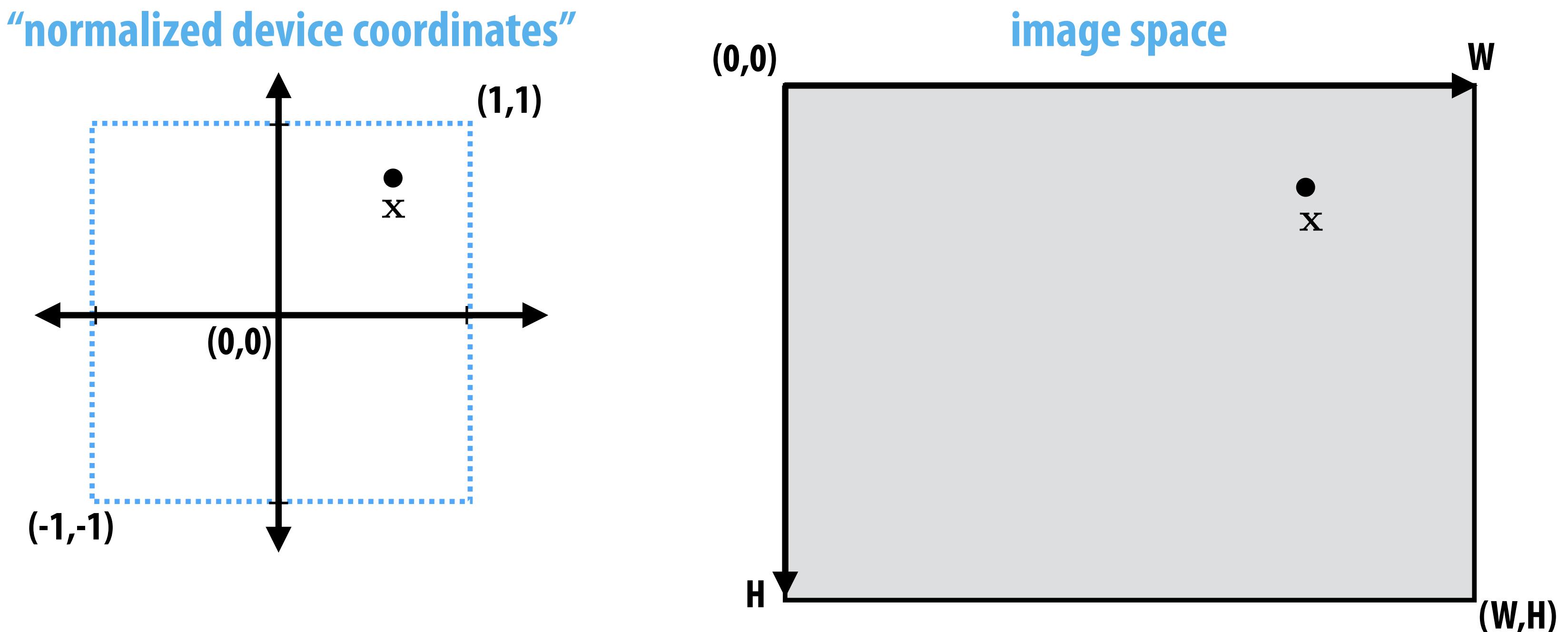
$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \\ z \end{bmatrix}$$

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

\*Assuming a pinhole camera at  $(0,0,0)$  looking down the  $z$ -axis

# Screen Transformation

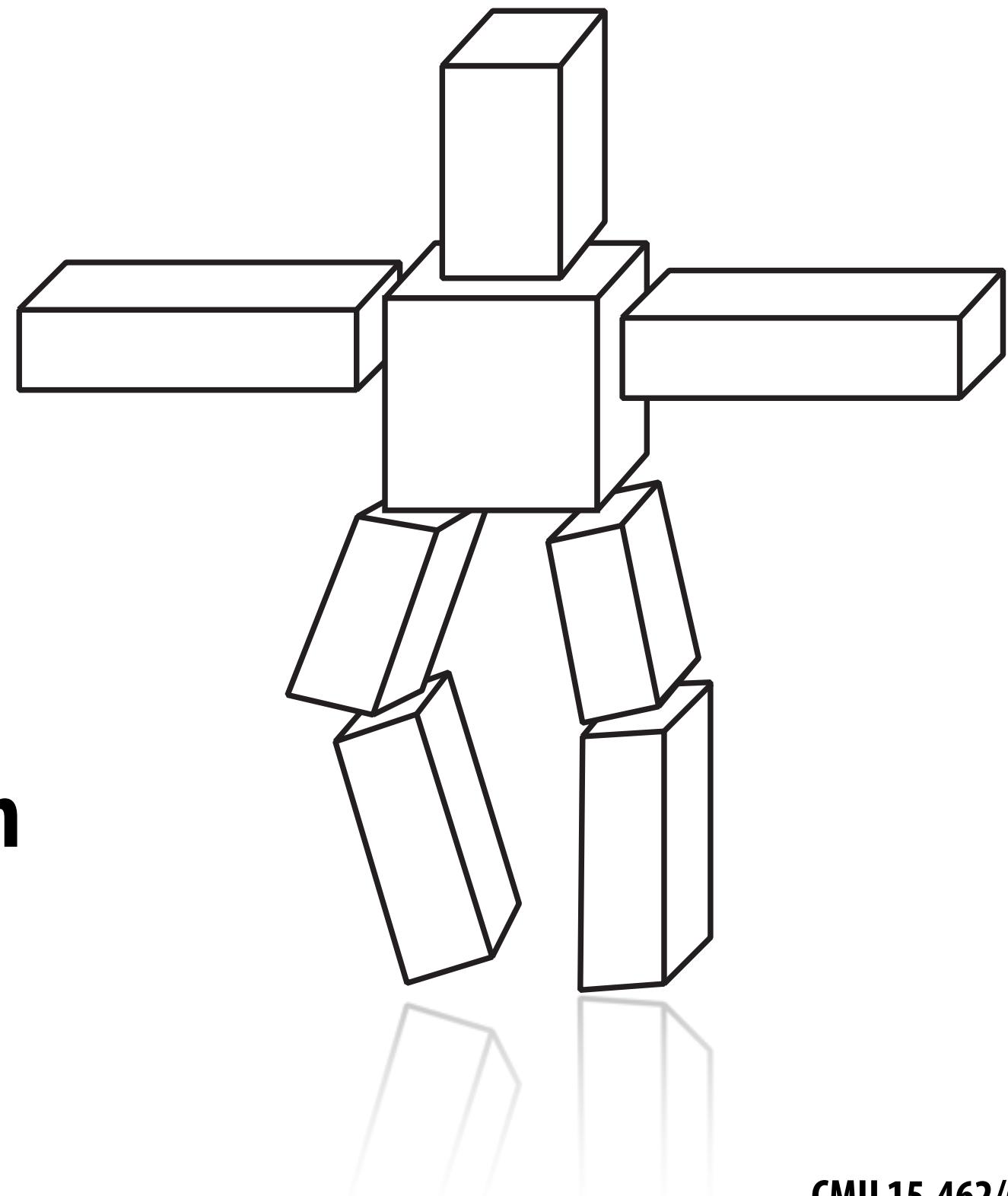
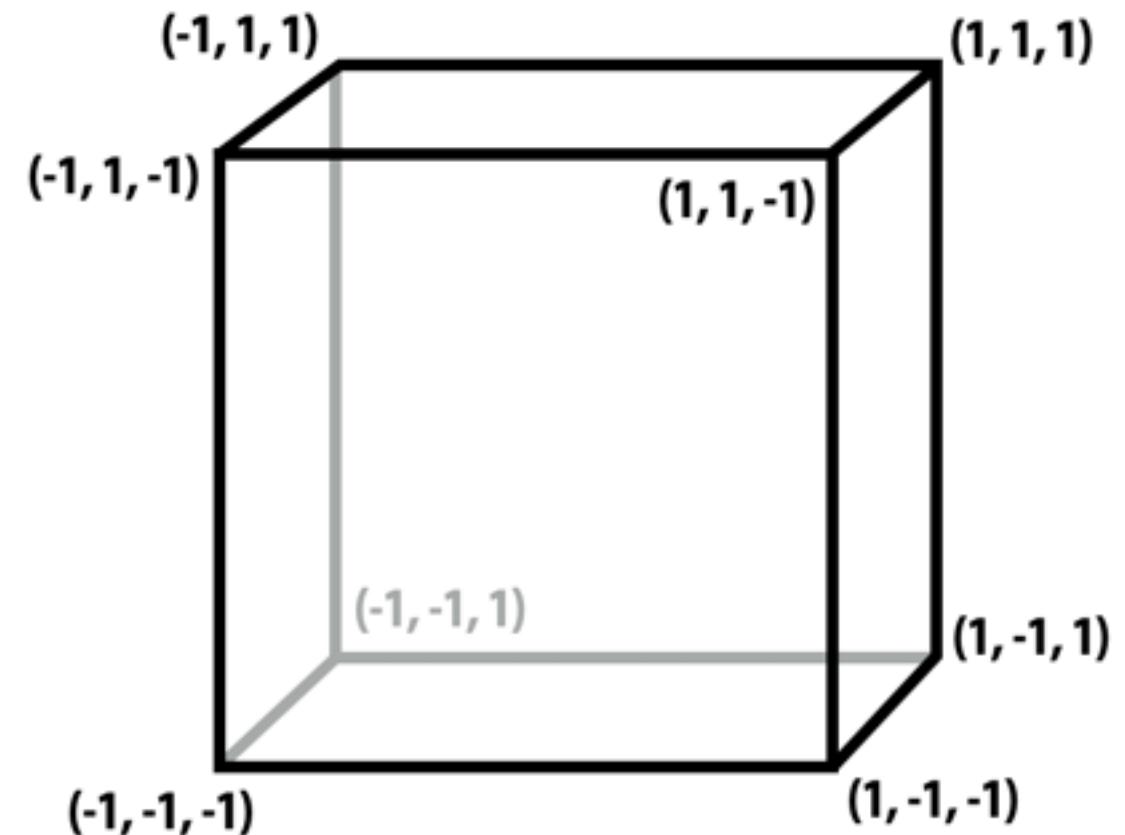
- One last transformation is needed in the rasterization pipeline: transform from viewing plane to pixel coordinates
- E.g., suppose we want to draw all points that fall inside the square  $[-1,1] \times [-1,1]$  on the  $z = 1$  plane, into a  $W \times H$  pixel image



**Q: What transformation(s) would you apply? (Careful:  $y$  is now down!)**

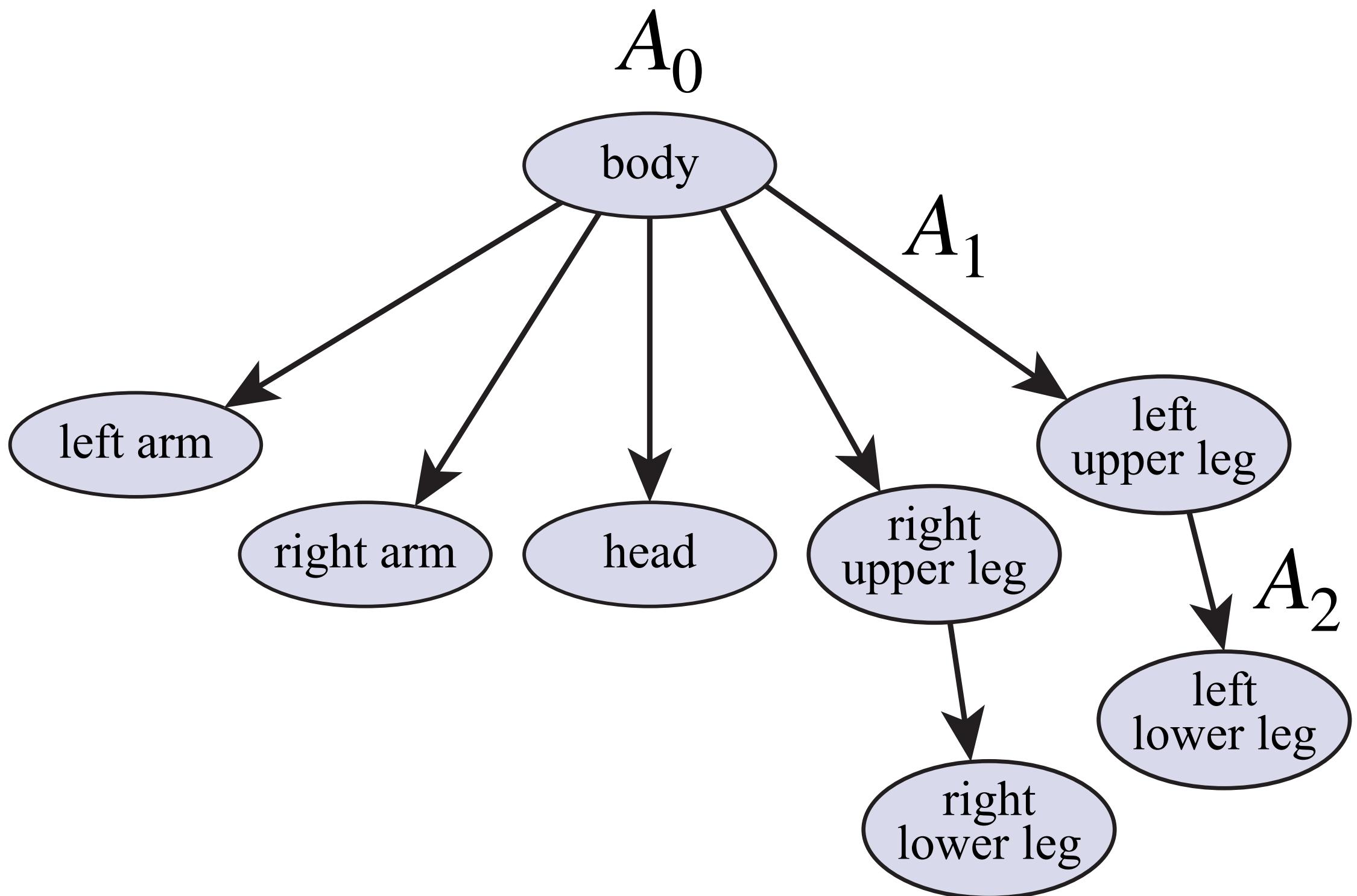
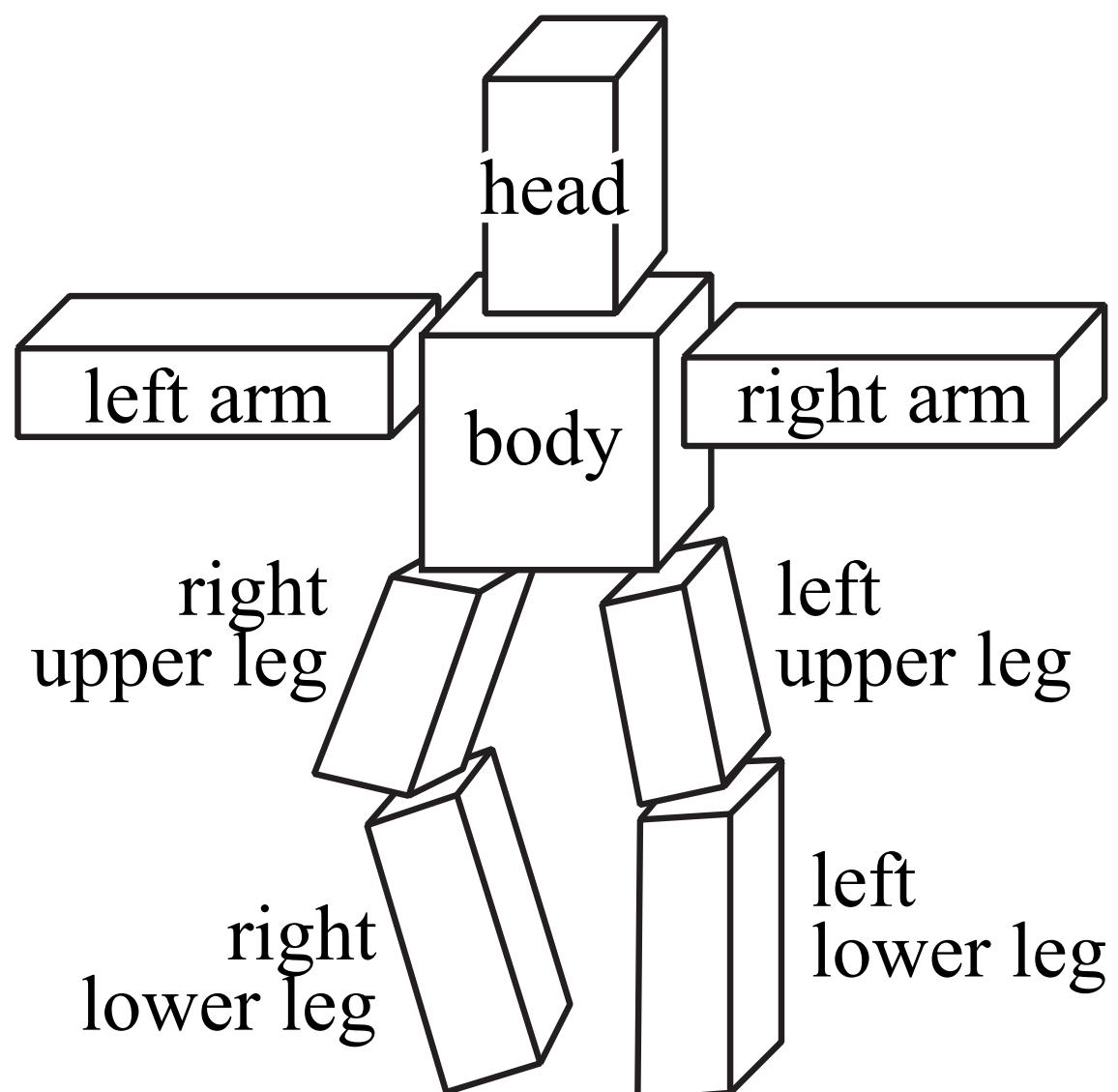
# Scene Graph

- For complex scenes (e.g., more than just a cube!) **scene graph** can help organize transformations
- Motivation: suppose we want to build a “cube creature” by transforming copies of the unit cube
- Difficult to specify each transformation directly
- Instead, build up transformations of “lower” parts from transformations of “upper” parts
  - E.g., first position the body
  - Then transform upper arm relative to the body
  - Then transform lower arm relative to upper arm
  - ...



# Scene Graph (continued)

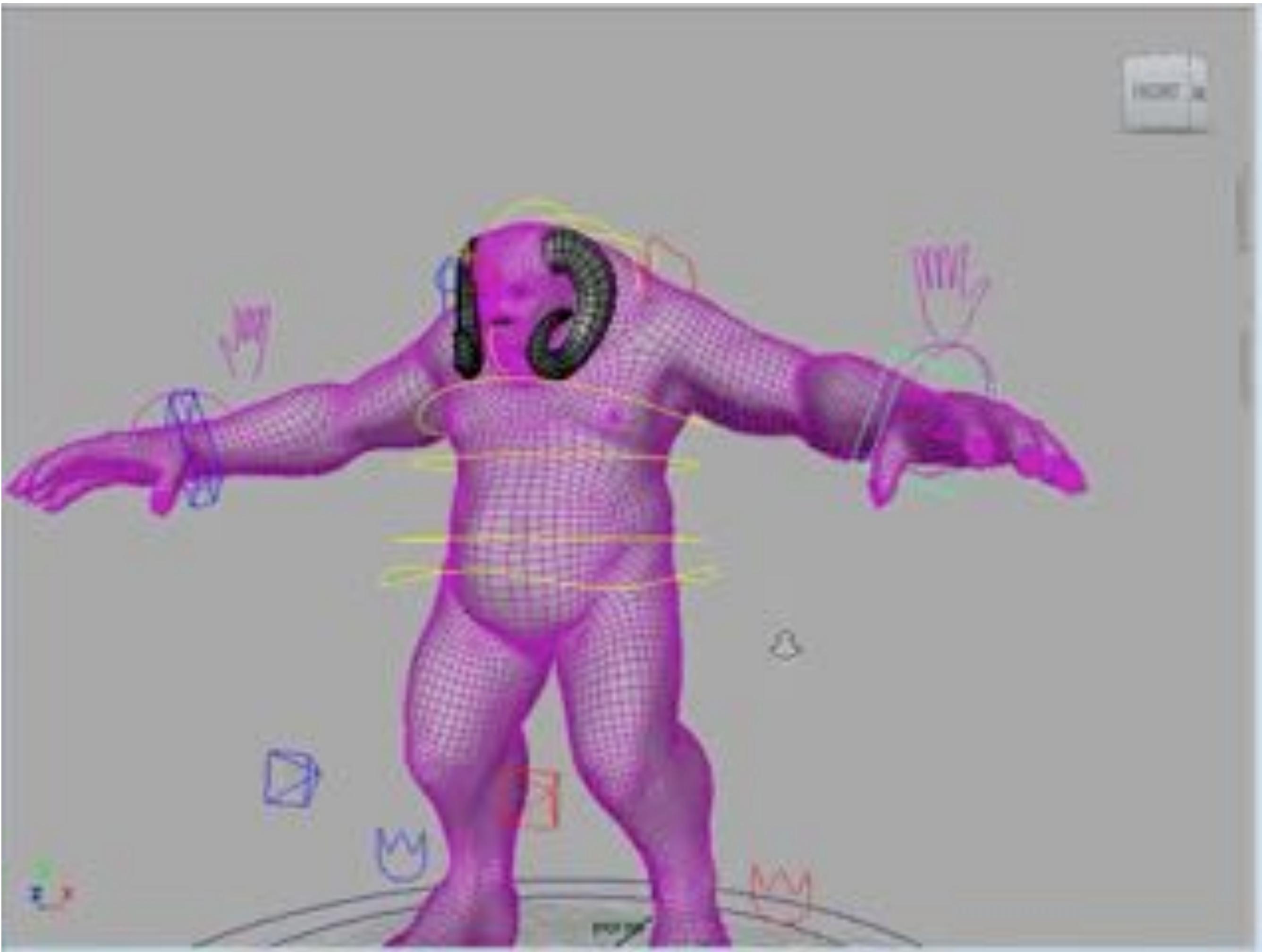
- Scene graph stores relative transformations in directed graph
- Each edge (+root) stores a linear transformation (e.g., a 4x4 matrix)
- Composition of transformations gets applied to nodes



- E.g.,  $A_1A_0$  gets applied to left upper leg;  $A_2A_1A_0$  to left lower leg
- Keep transformations on a stack to reduce redundant multiplication

# Scene Graph—Example

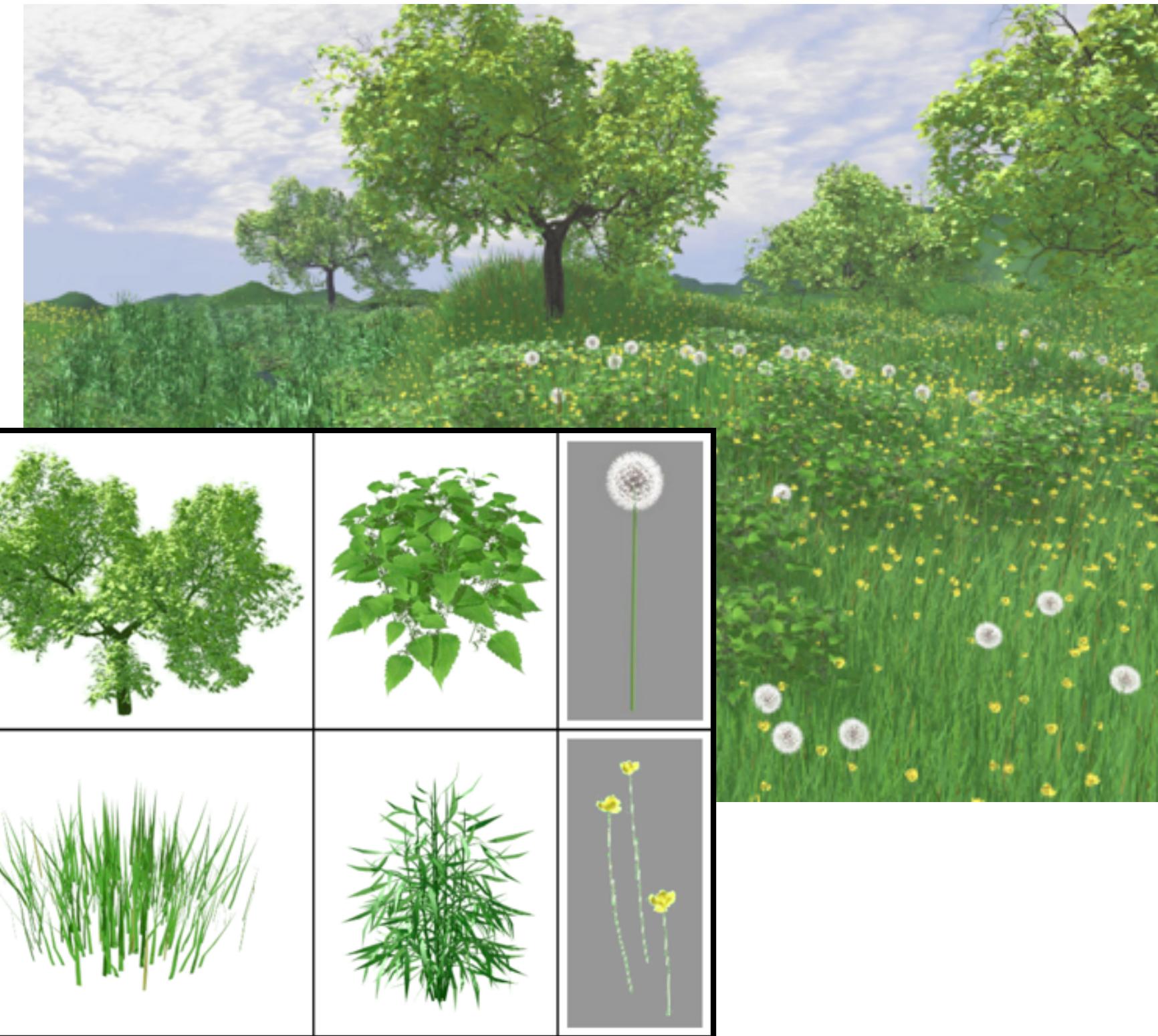
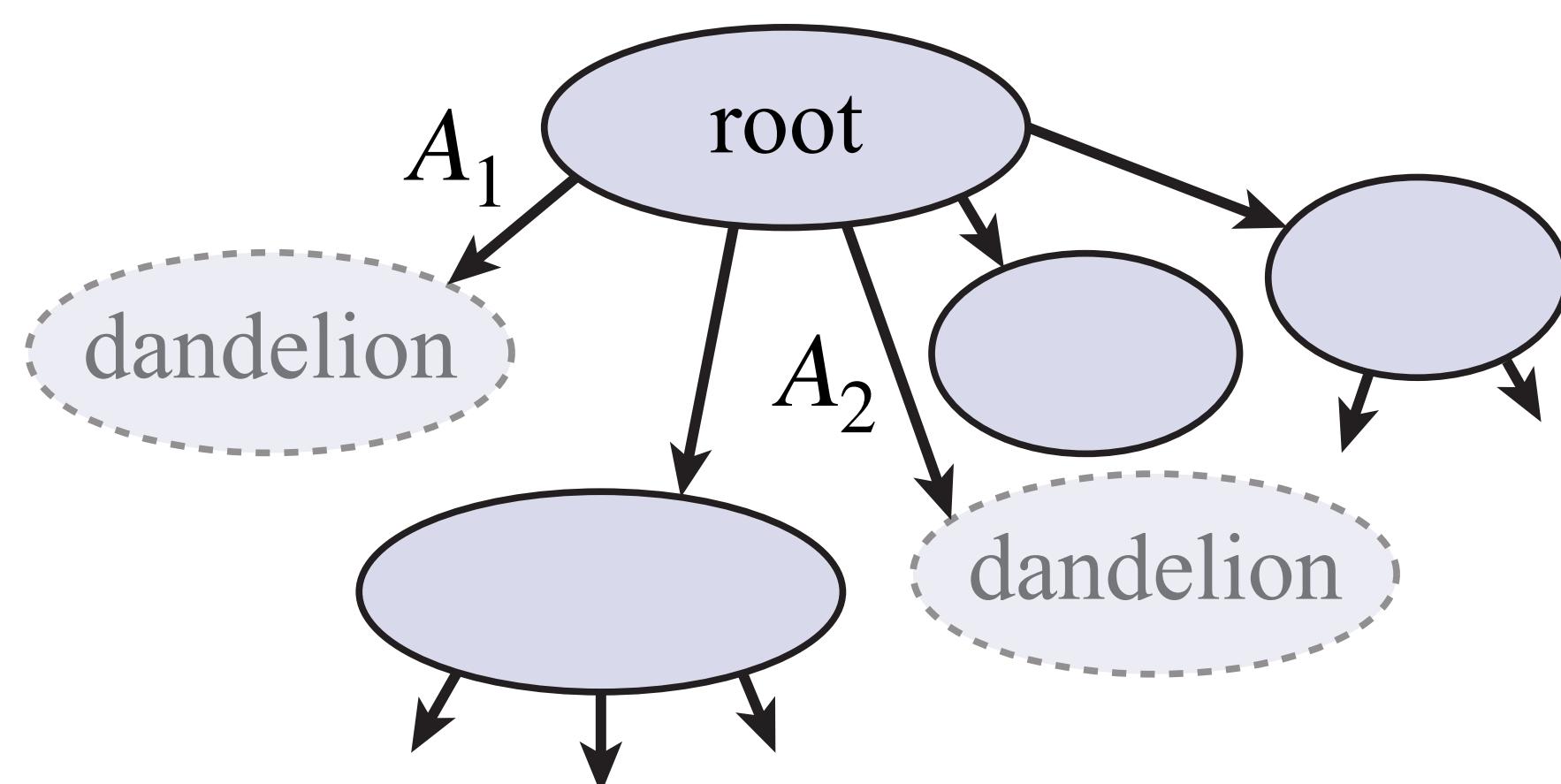
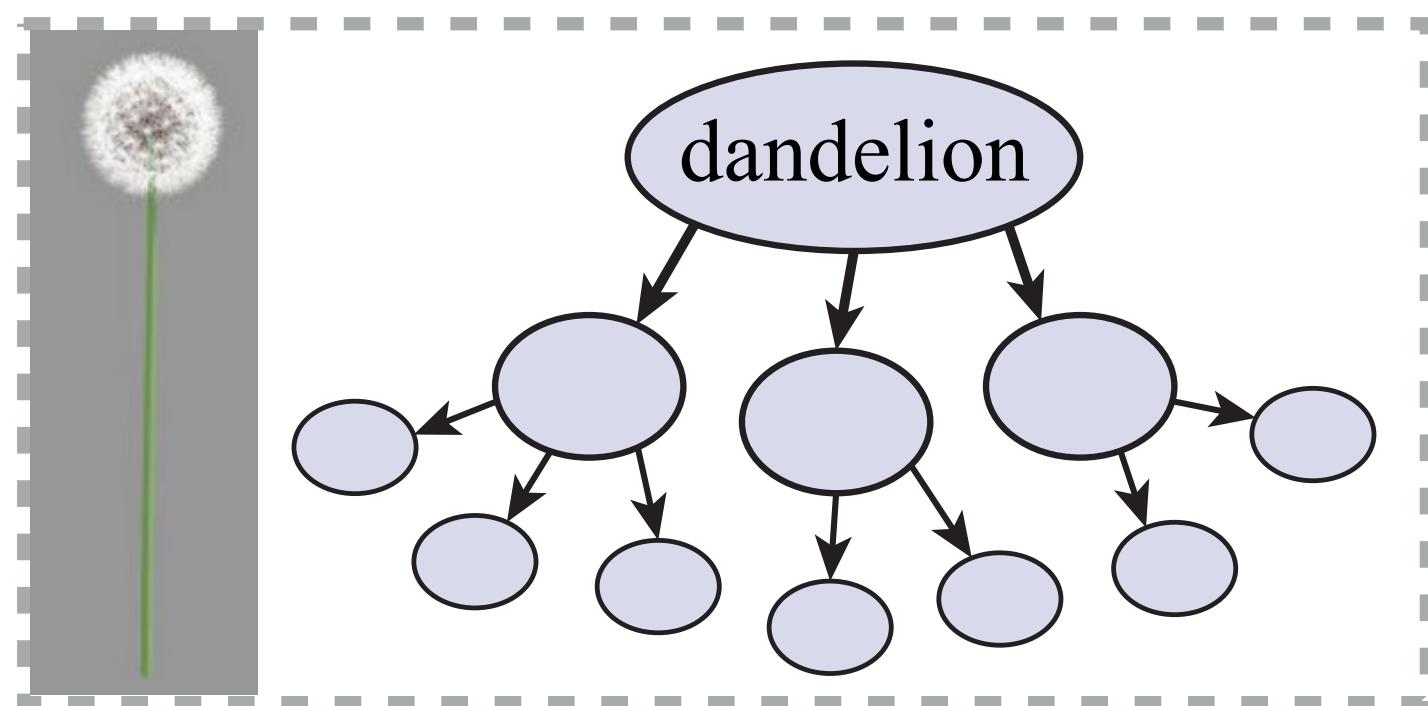
Often used to build up complex “rig”:



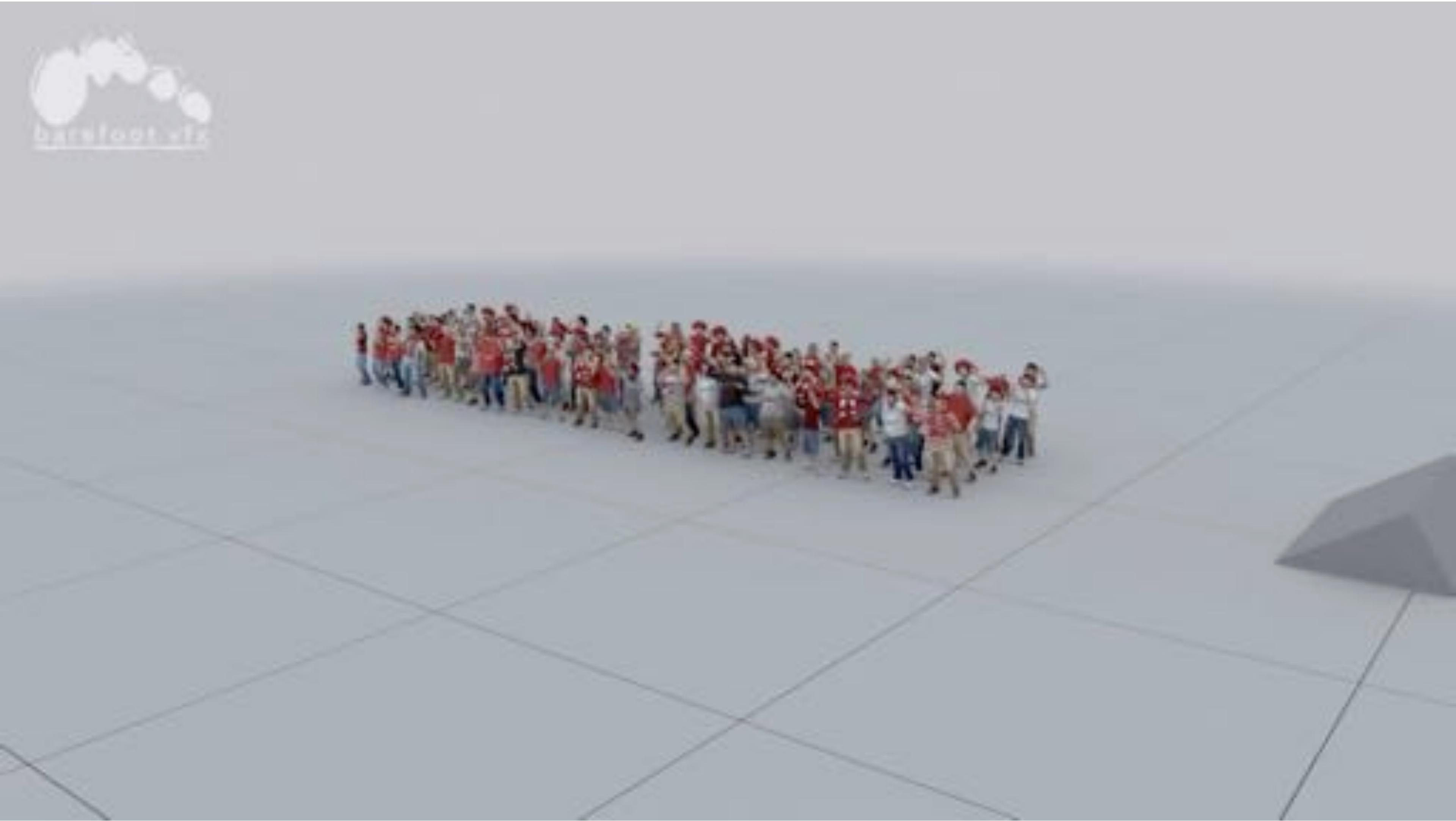
In general, scene graph also includes other models, lights, cameras, ...

# Instancing

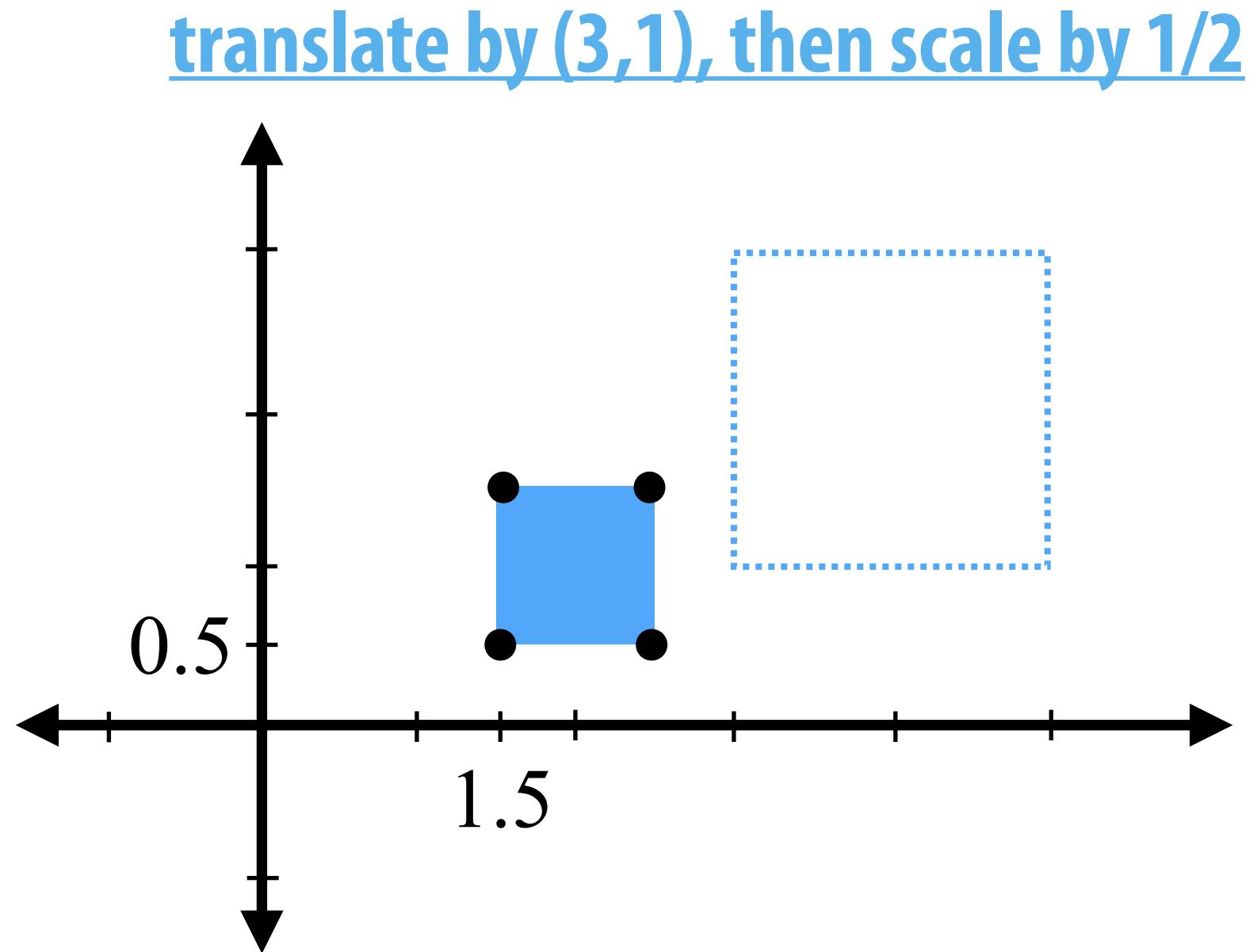
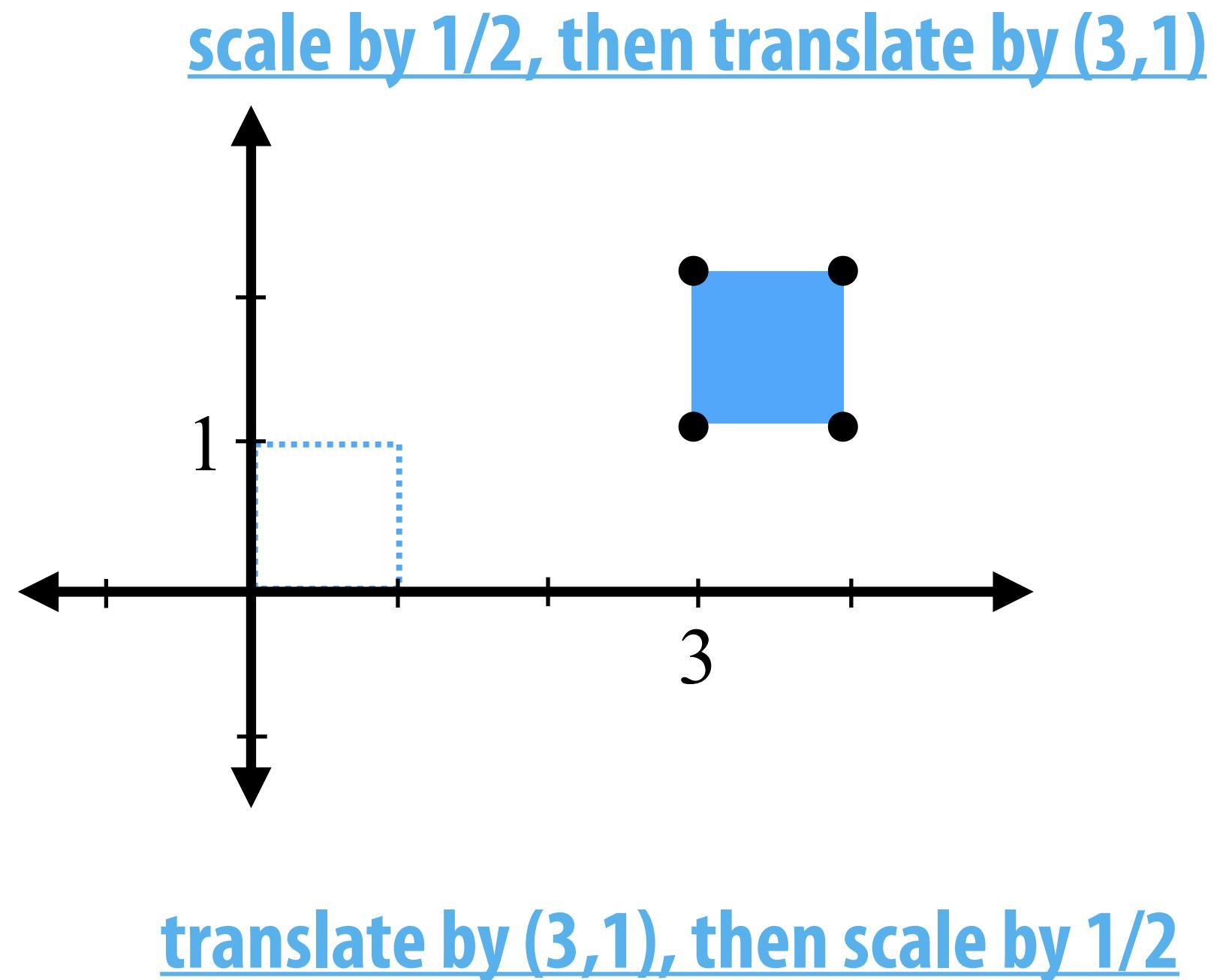
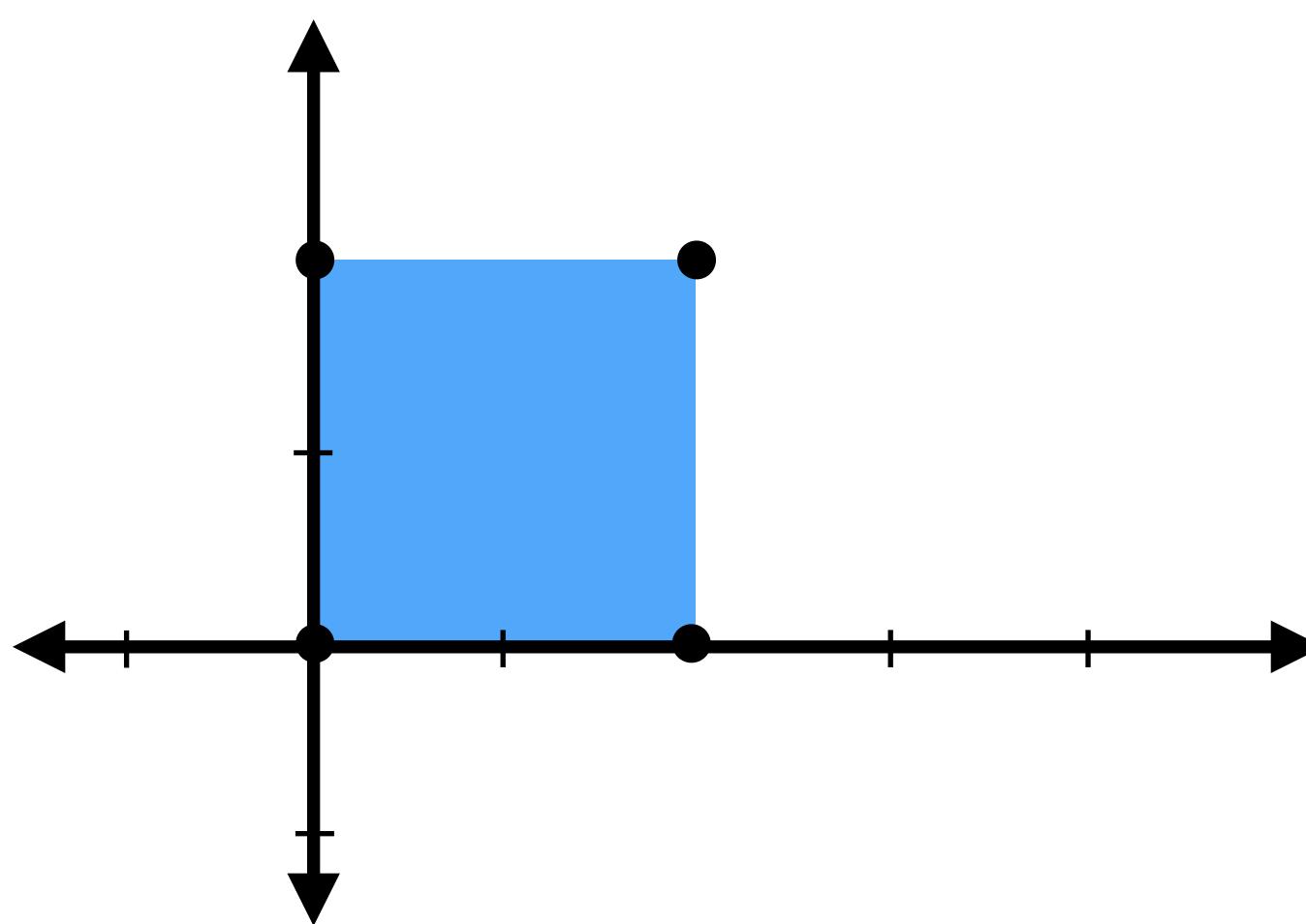
- What if we want many copies of the same object in a scene?
- Rather than have many copies of the geometry, scene graph, etc., can just put a “pointer” node in our scene graph
- Like any other node, can specify a different transformation on each incoming edge



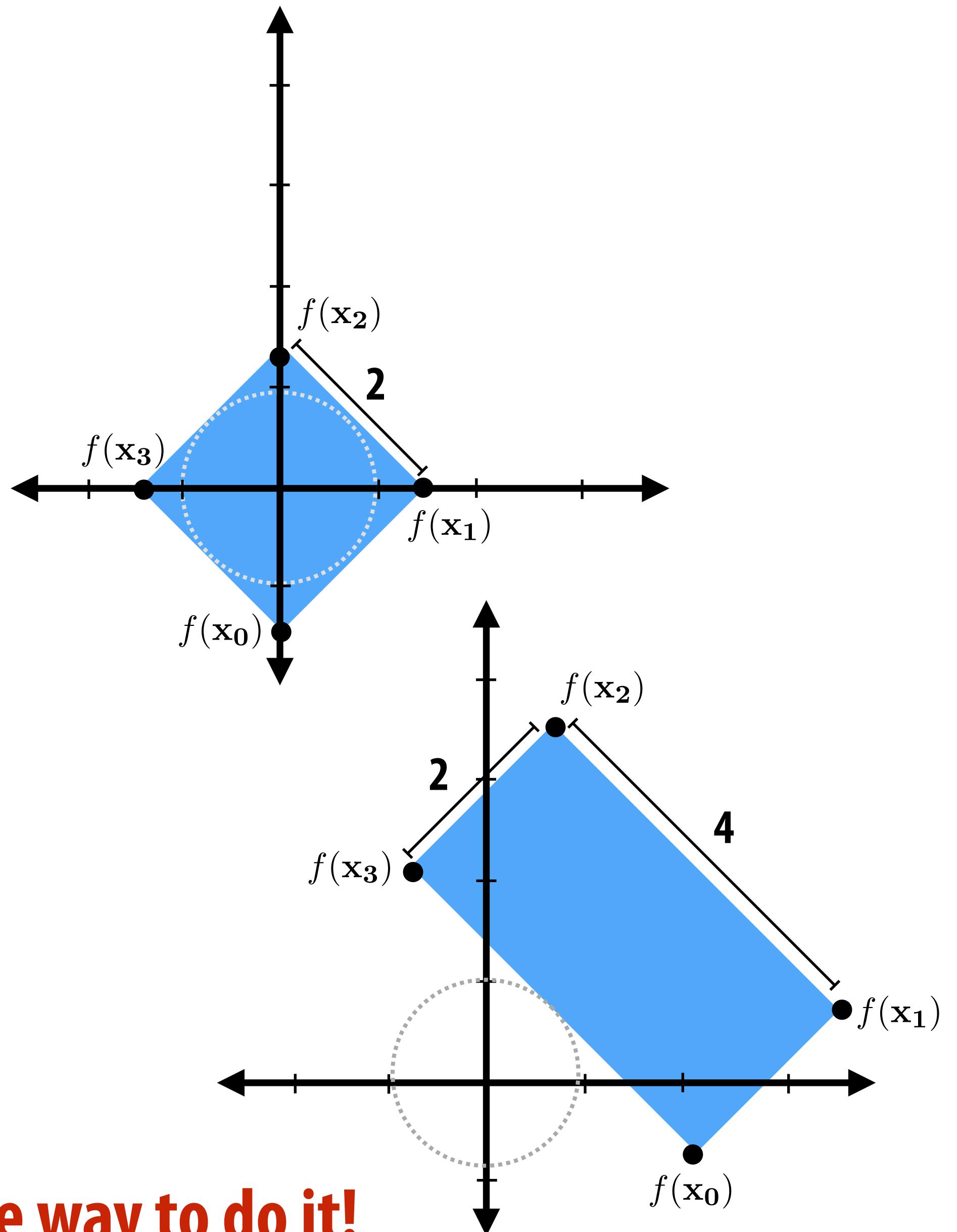
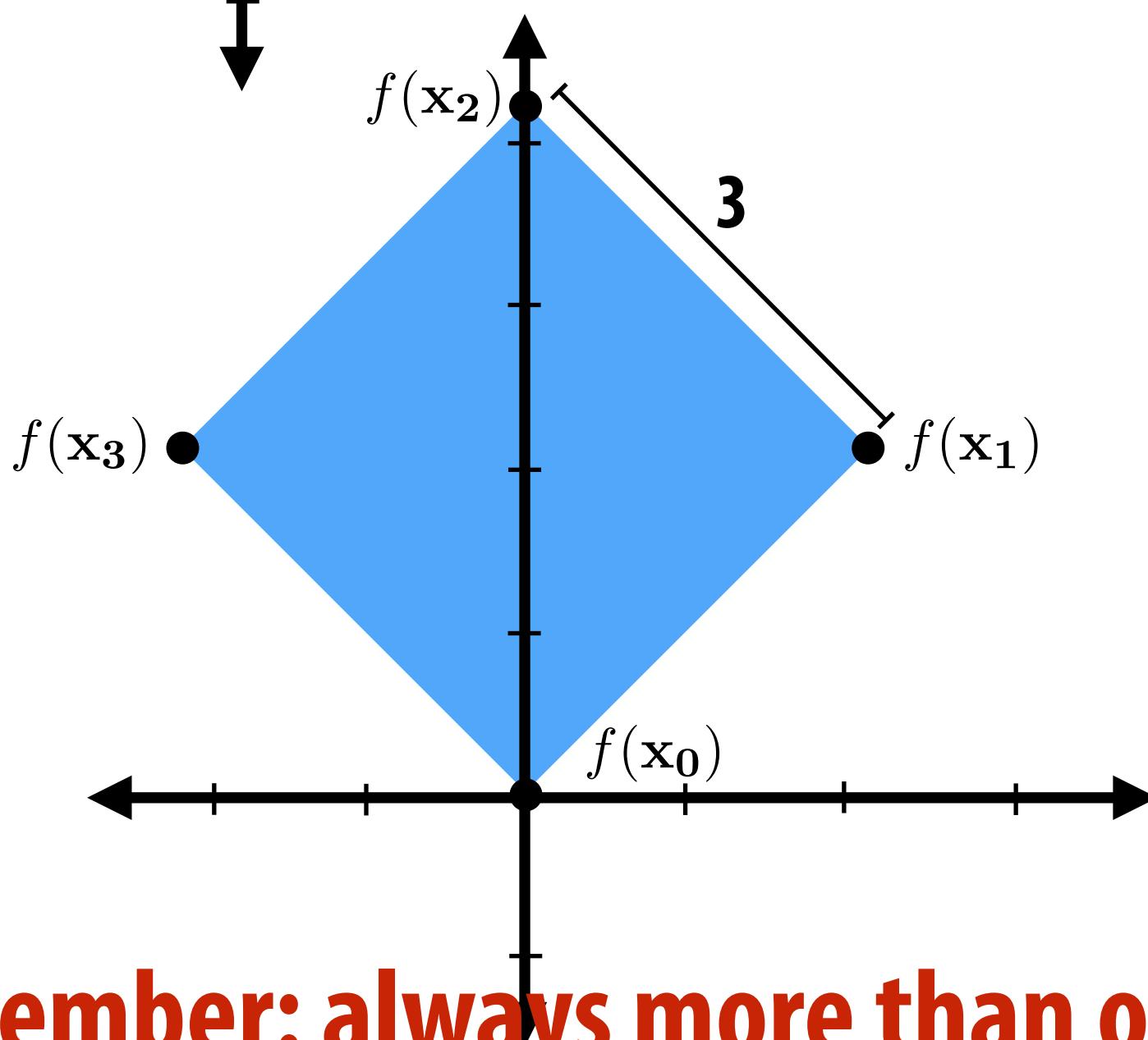
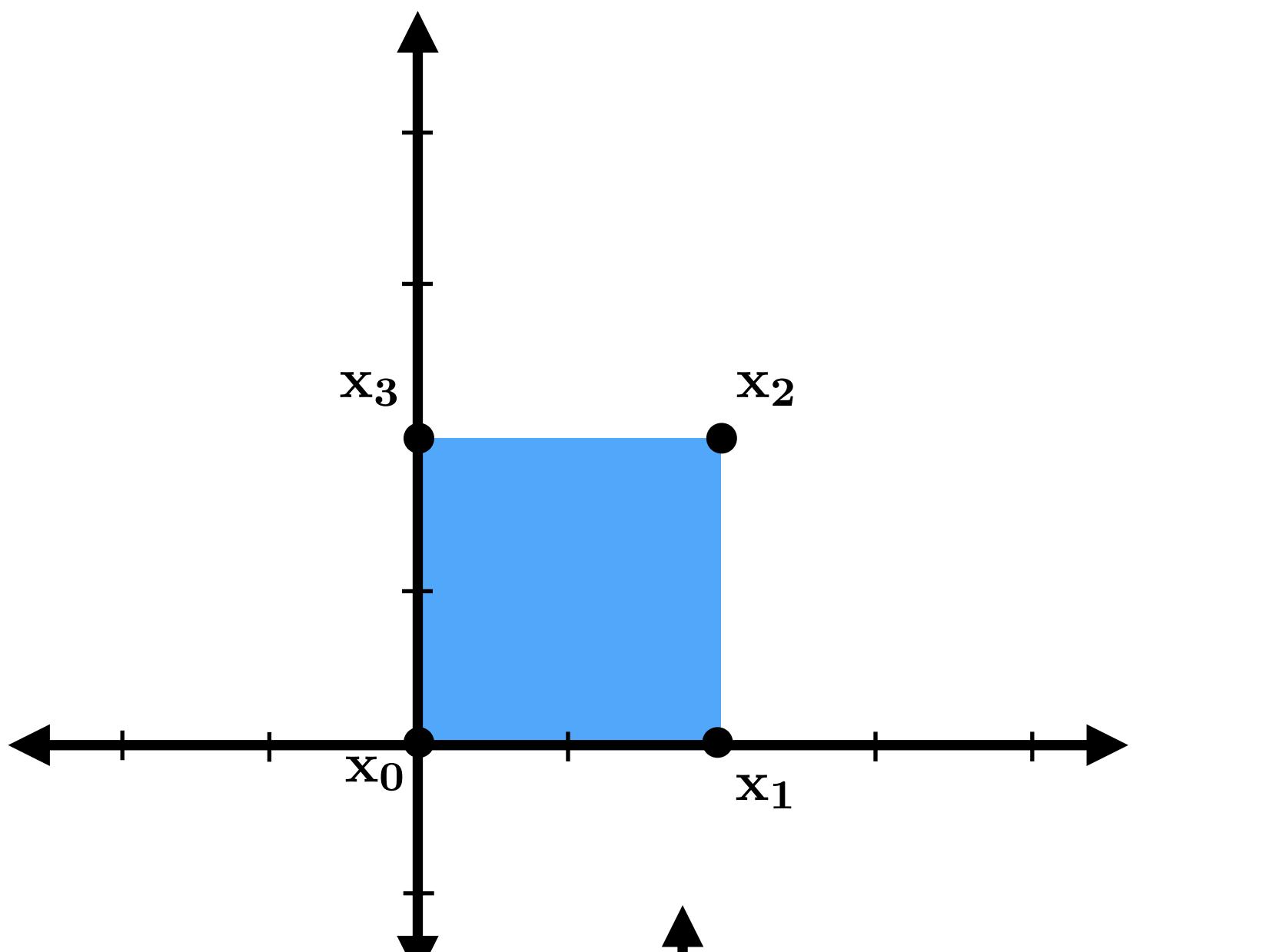
# Instancing—Example



# Order matters when composing transformations!

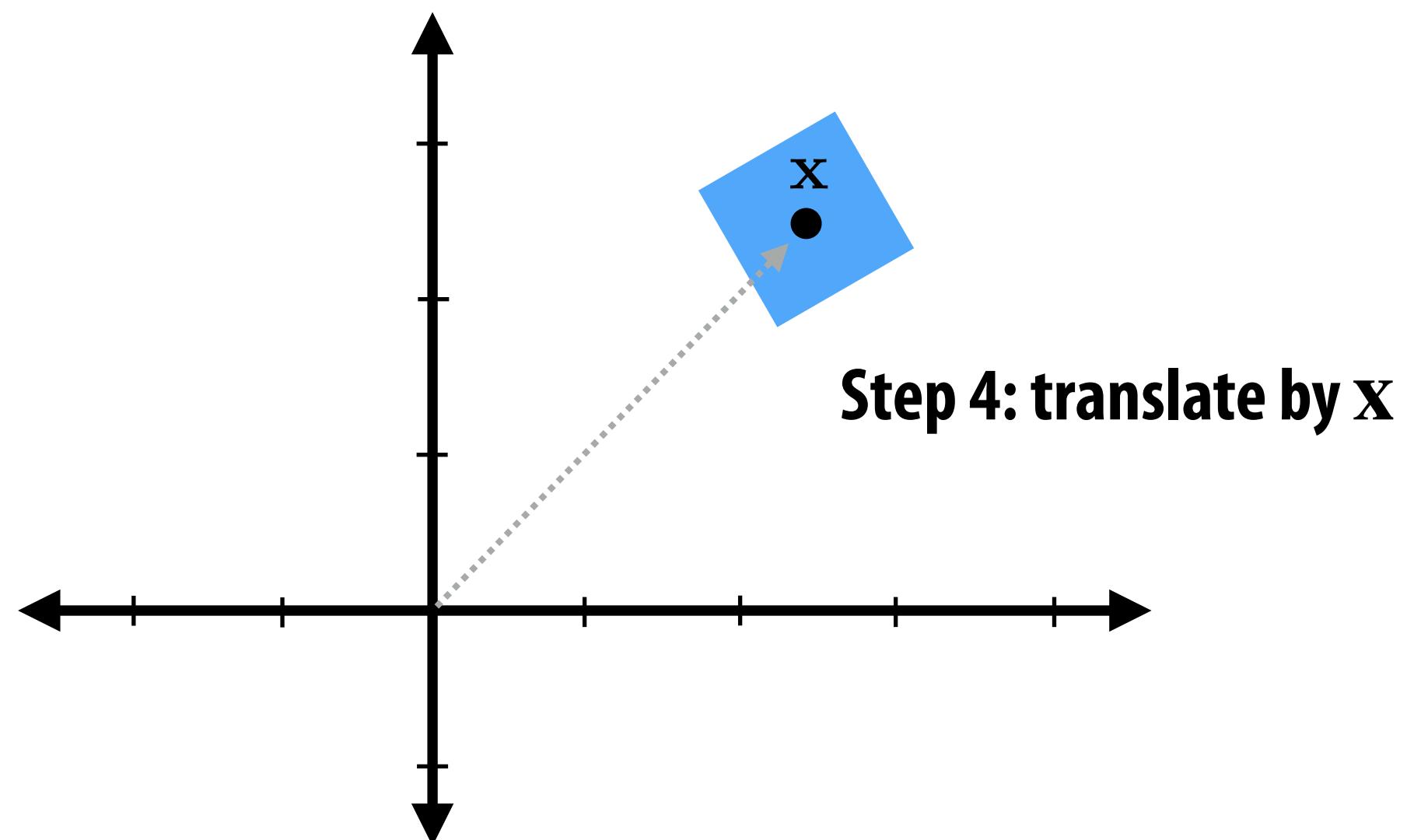
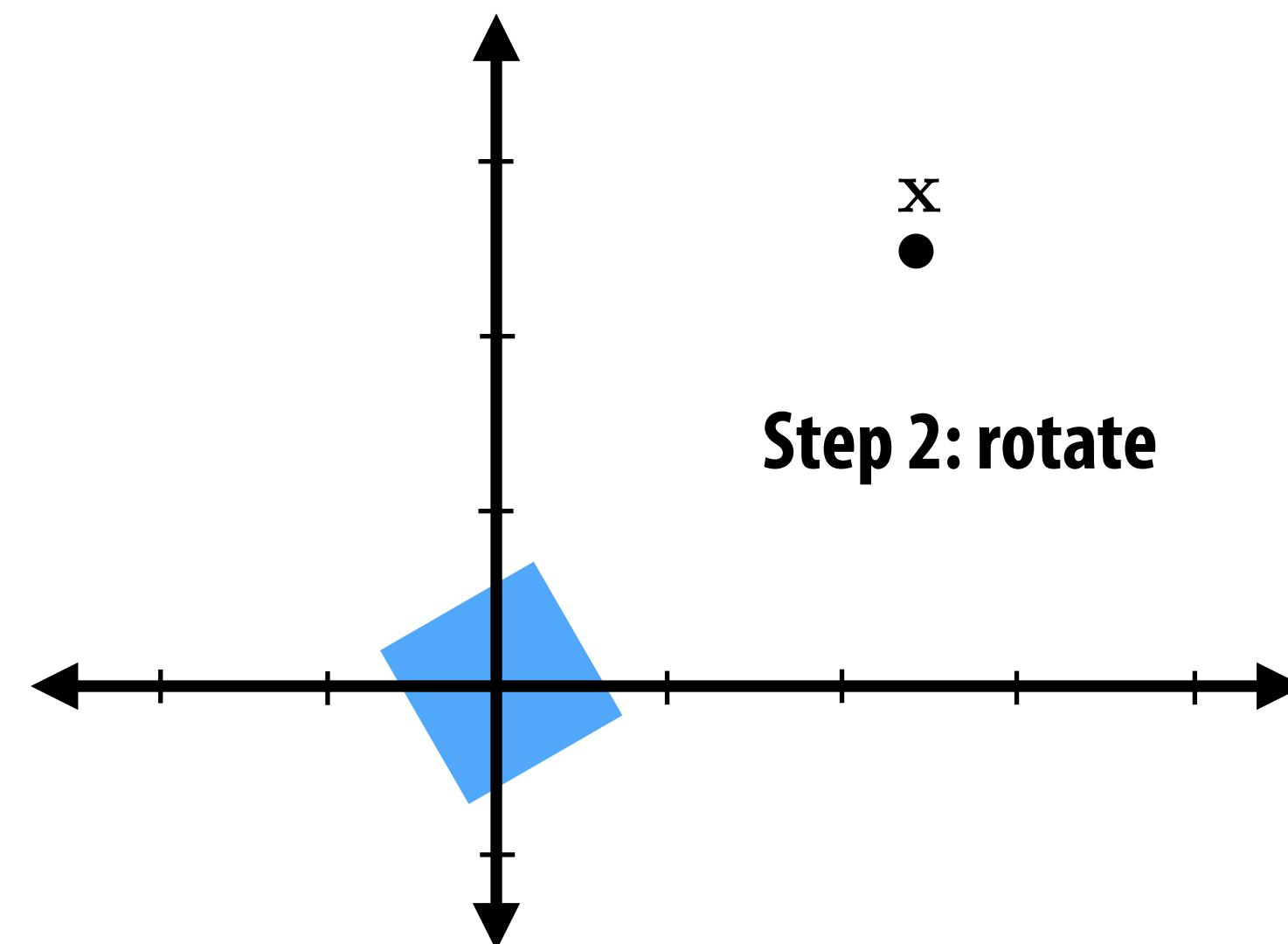
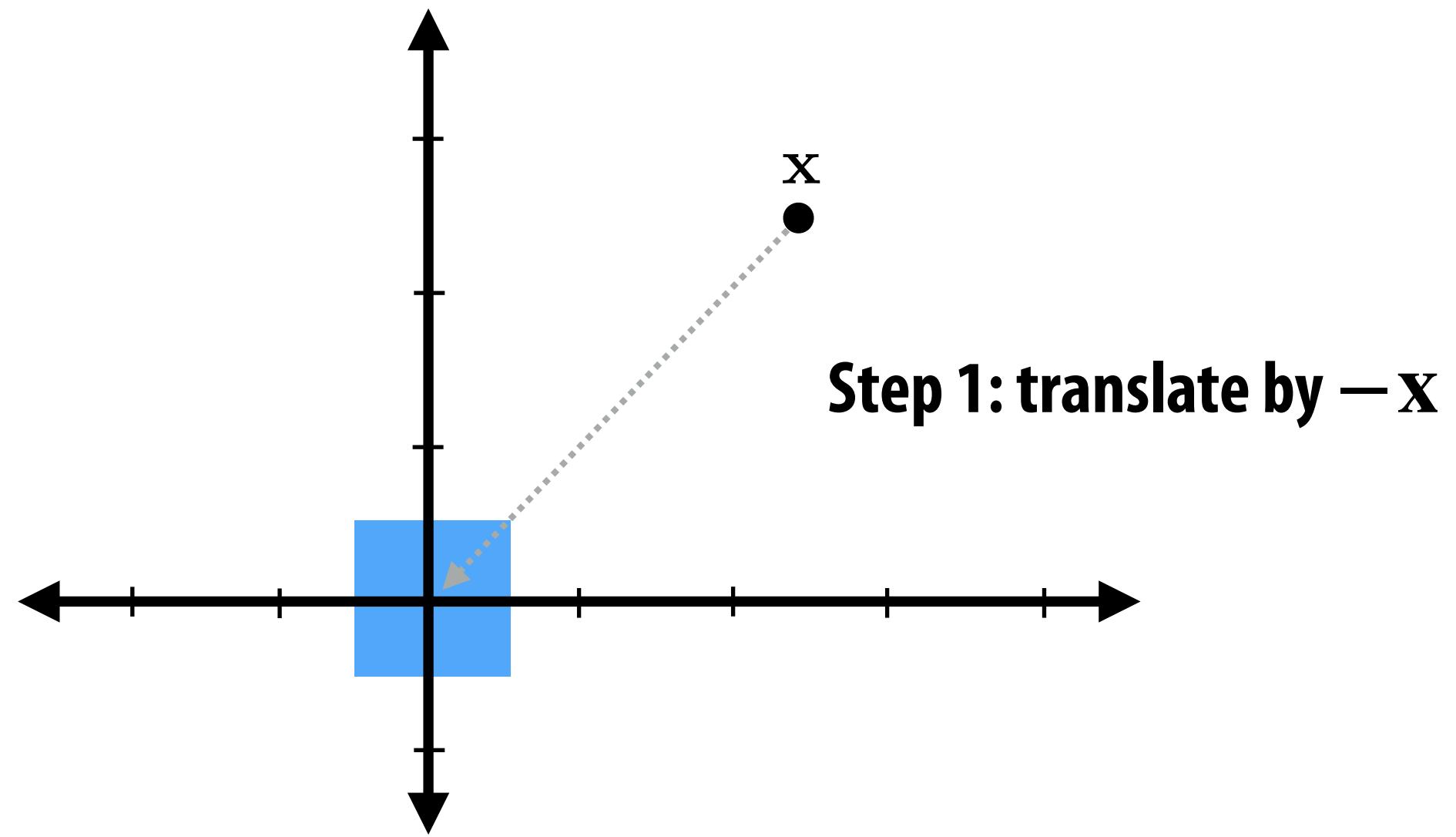
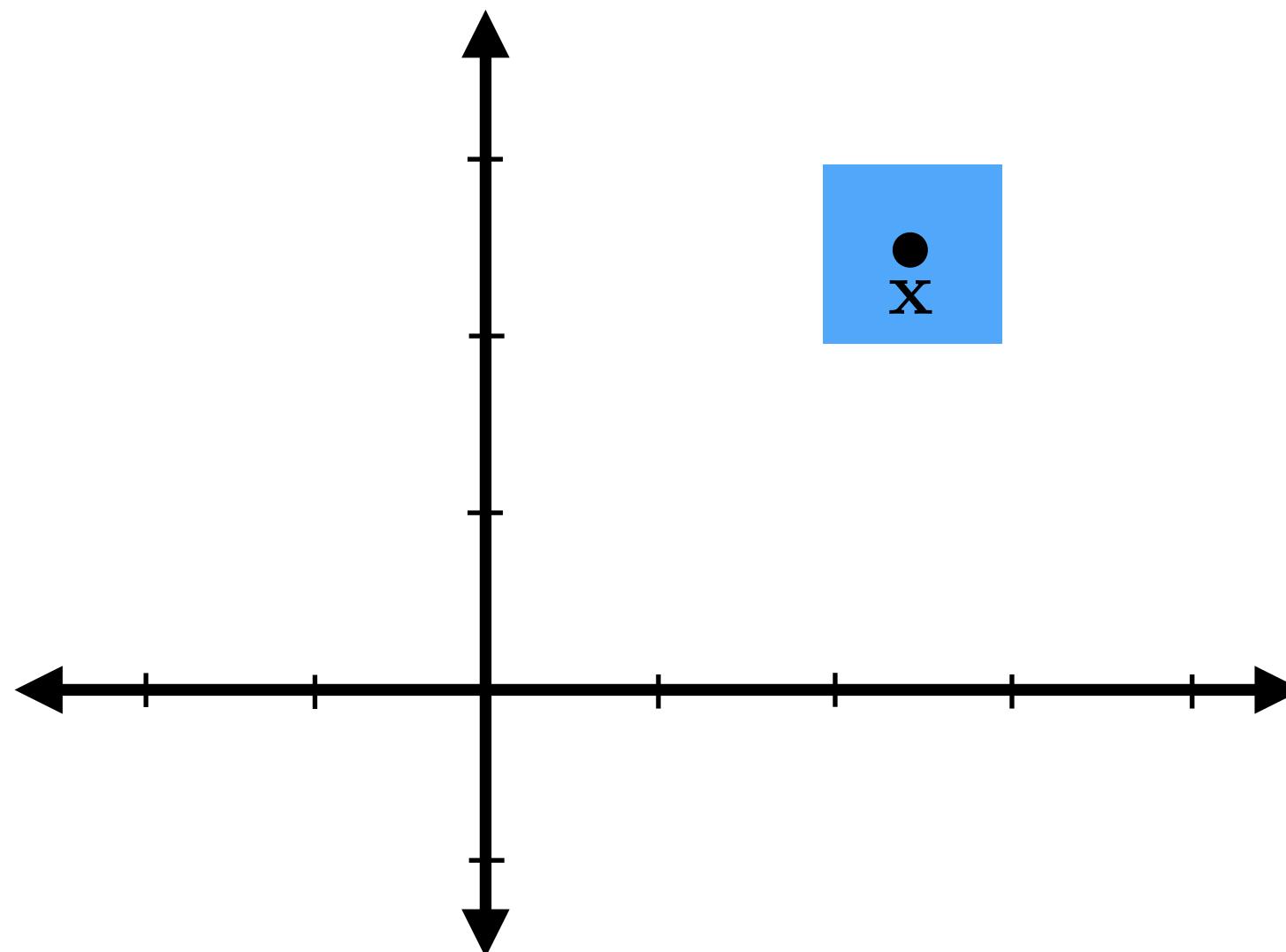


# How would you perform these transformations?



**Remember: always more than one way to do it!**

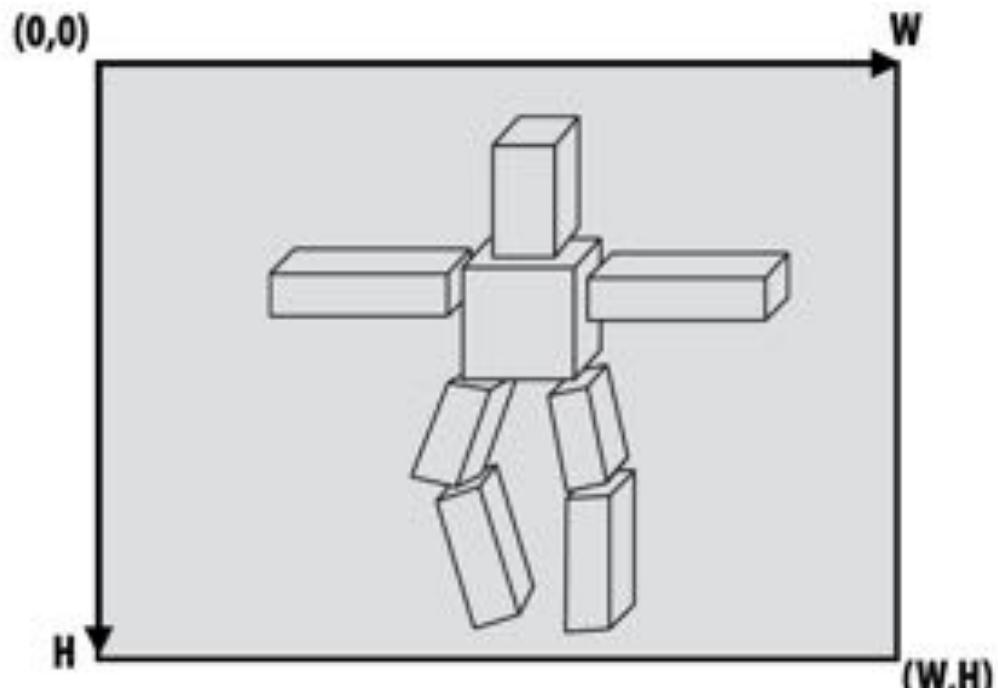
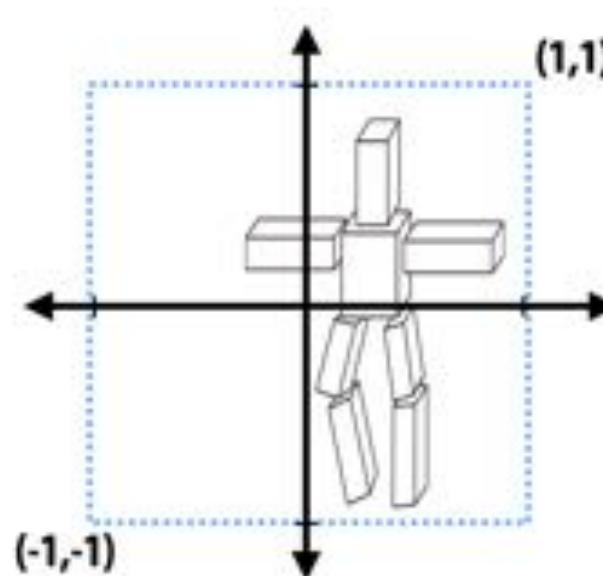
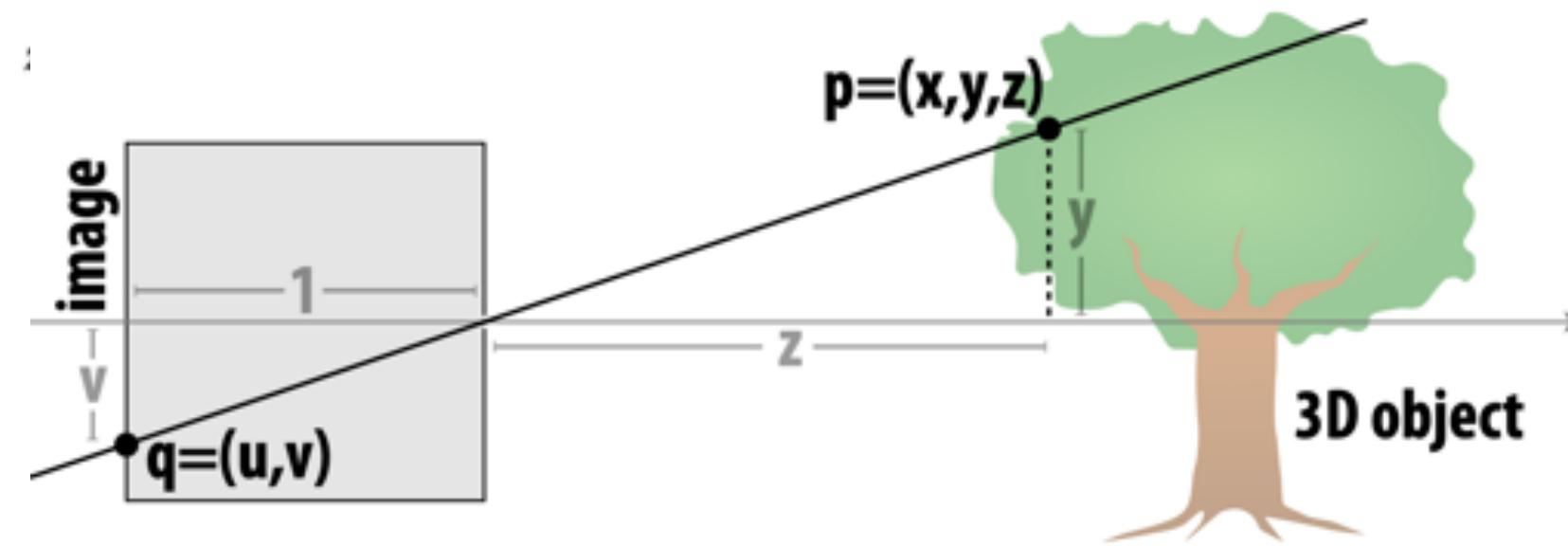
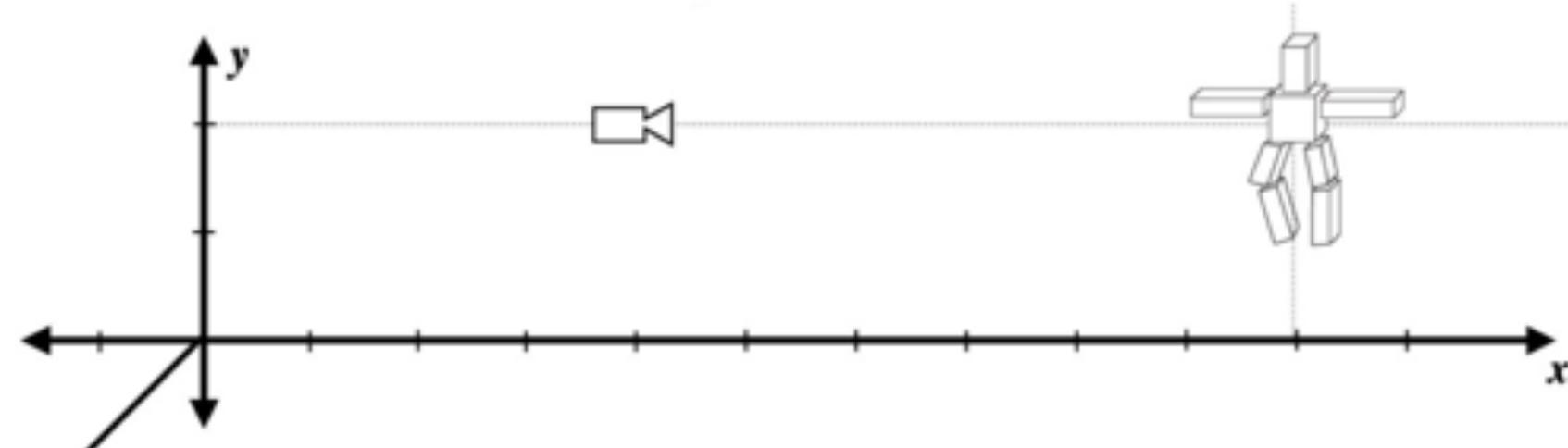
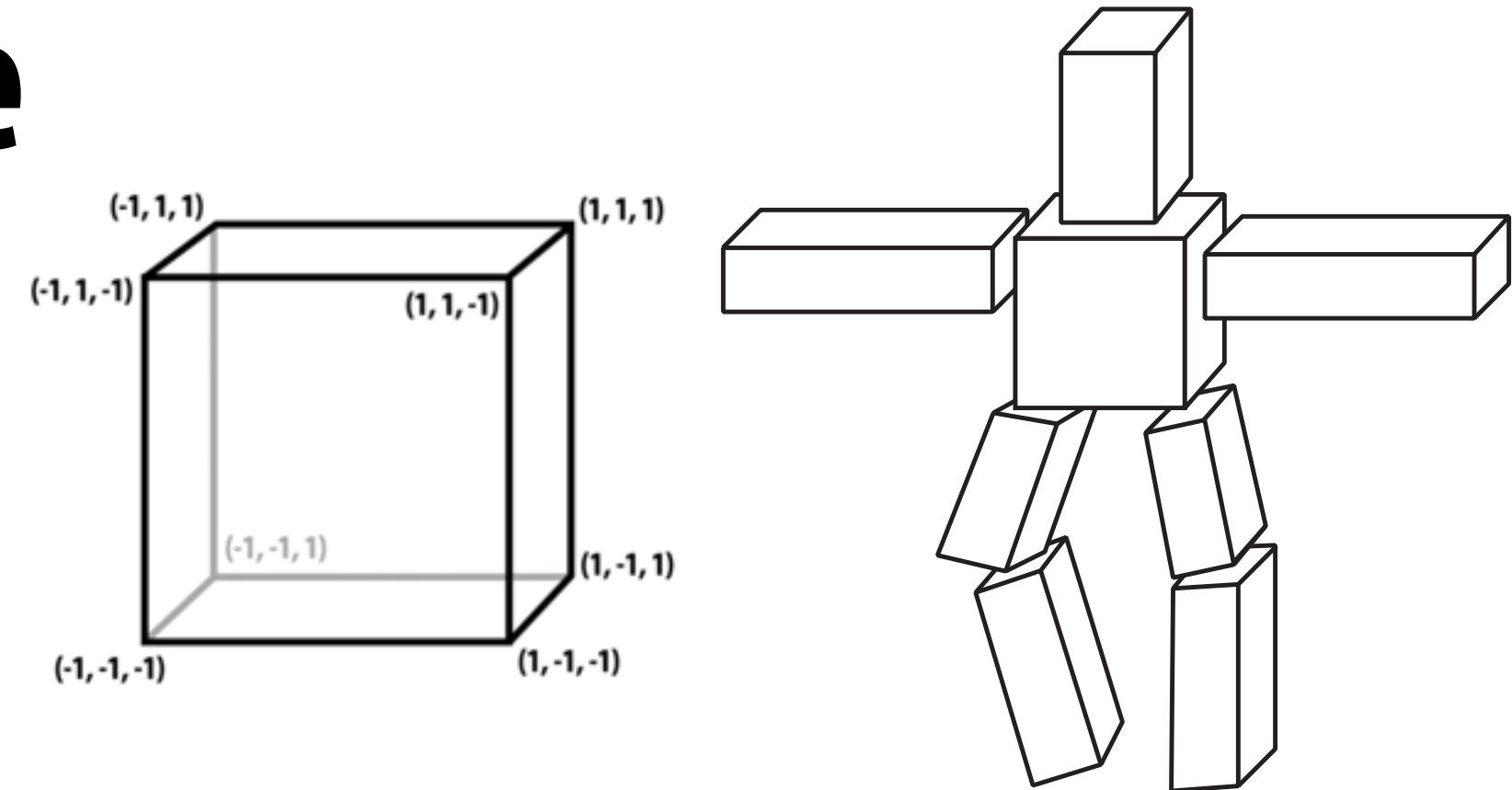
# Common task: rotate about a point $x$



**Q: What happens if we just rotate without translating first?**

# Drawing a Cube Creature

- Let's put this all together: starting with our 3D cube, we want to make a 2D, perspective-correct image of a "cube creature"
- First we use our scene graph to apply 3D transformations to several copies of our cube
- Then we apply a 3D transformation to position our camera
- Then a perspective projection
- Finally we convert to image coordinates (and rasterize)
- ...Easy, right? :-)



# Spatial Transformations—Summary

**transformation defined by its invariants**

## basic linear transformations

scaling  
rotation  
reflection  
shear

## composite transformations

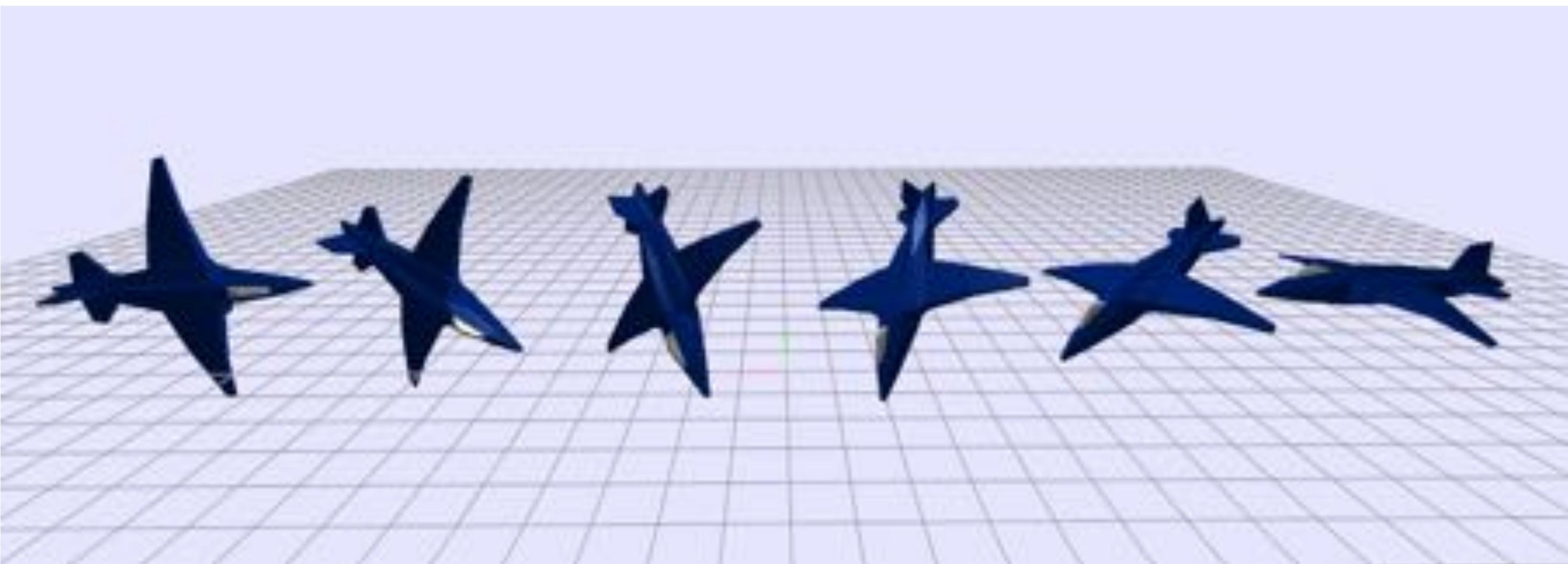
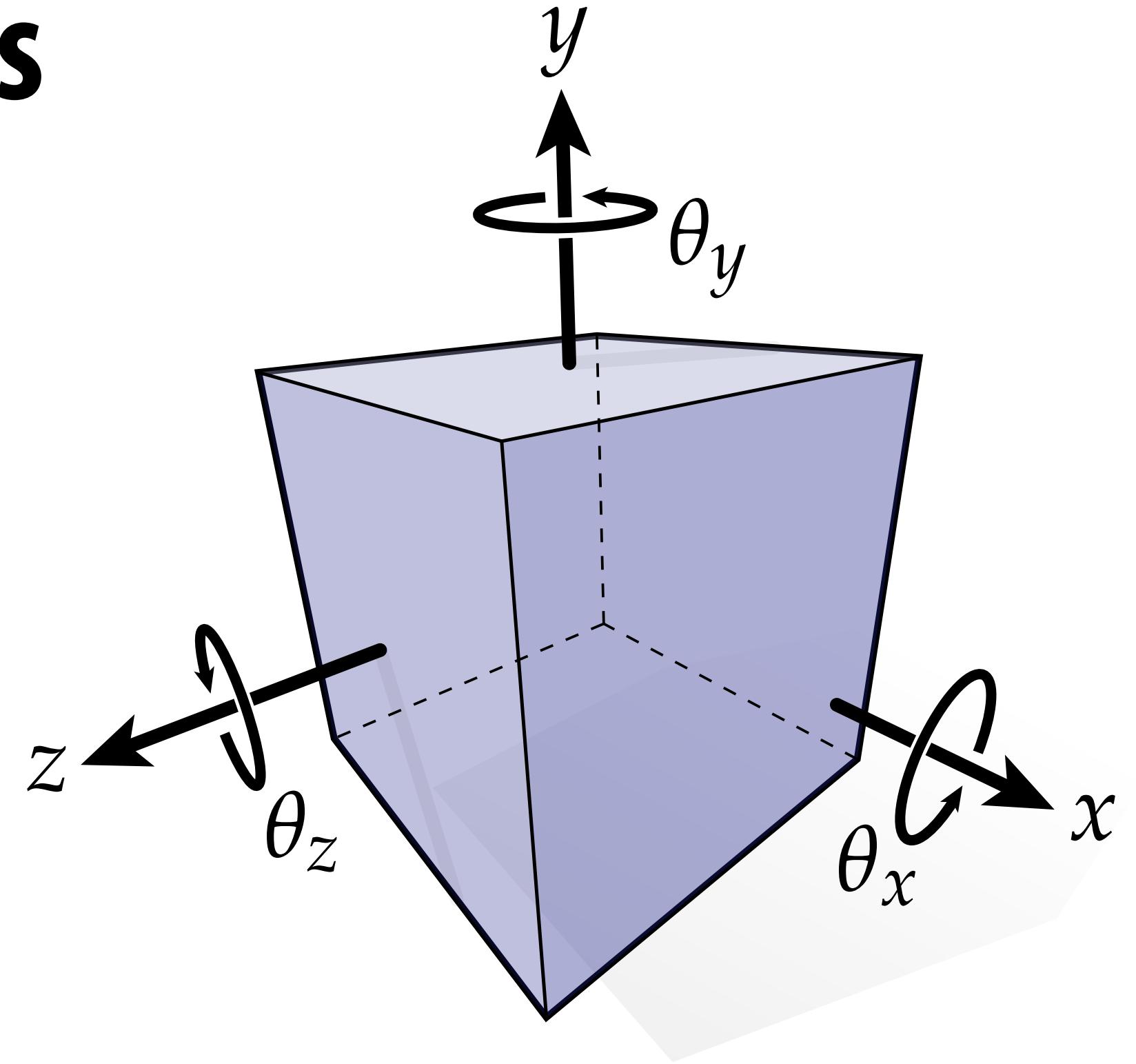
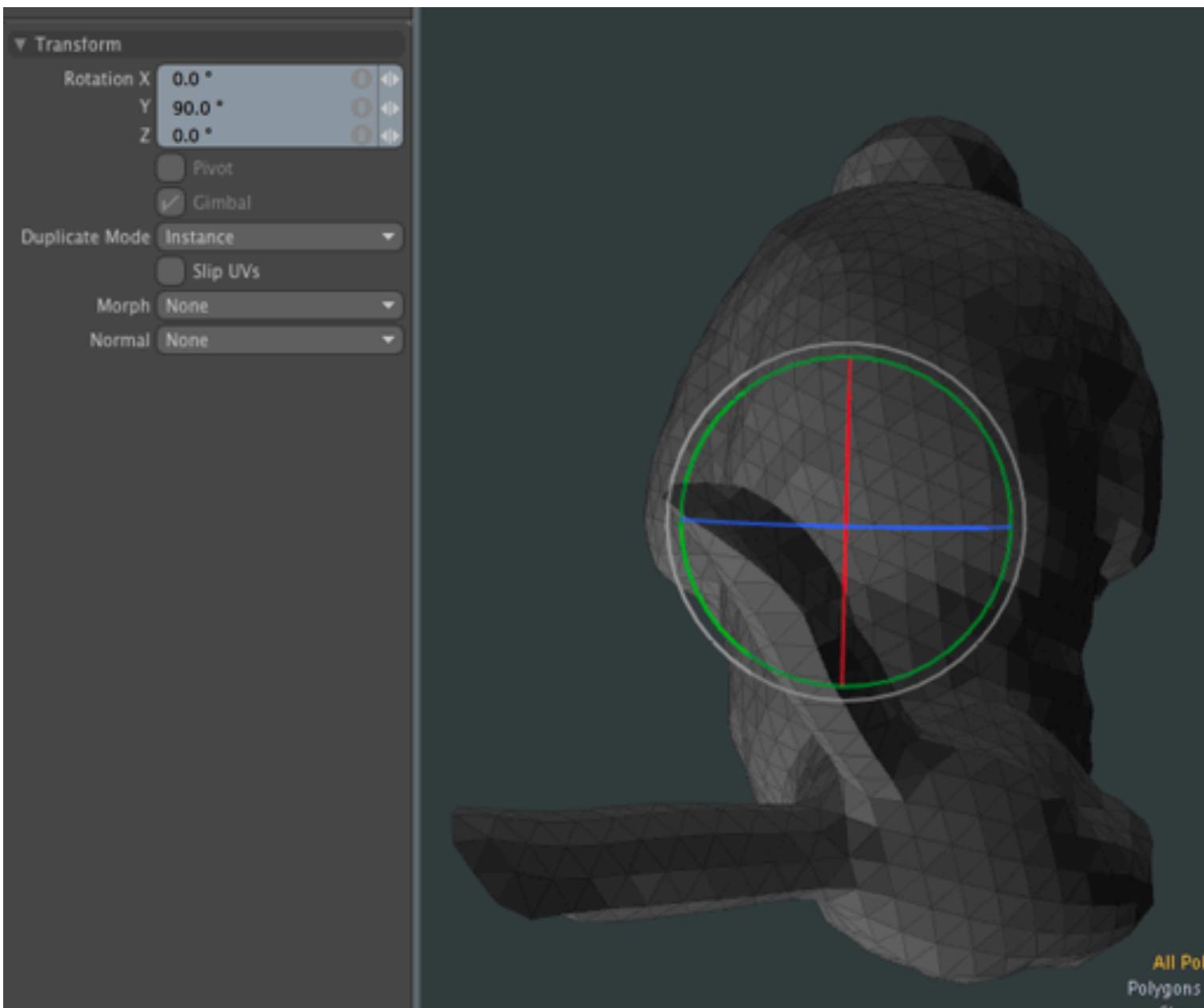
- compose basic transformations to get more interesting ones
- always reduces to a single 4x4 matrix (in homogeneous coordinates)
  - simple, unified representation, efficient implementation
- order of composition matters!
- many ways to decompose a given transformation (polar, SVD, ...)
- use scene graph to organize transformations
- use instancing to eliminate redundancy

## basic nonlinear transformations

translation  
perspective projection

linear when represented via homogeneous coords  
homogeneous coords also distinguish points & vectors

# Next time: 3D Rotations



Computer Graphics

Chapter 7  
2D Geometric Transformations

# Chapter 7

## Two-Dimensional Geometric Transformations

### Part III.

- OpenGL Functions for Two-Dimensional Geometric Transformations
- OpenGL Geometric Transformation Programming Examples

# OpenGL Geometric Transformation Functions

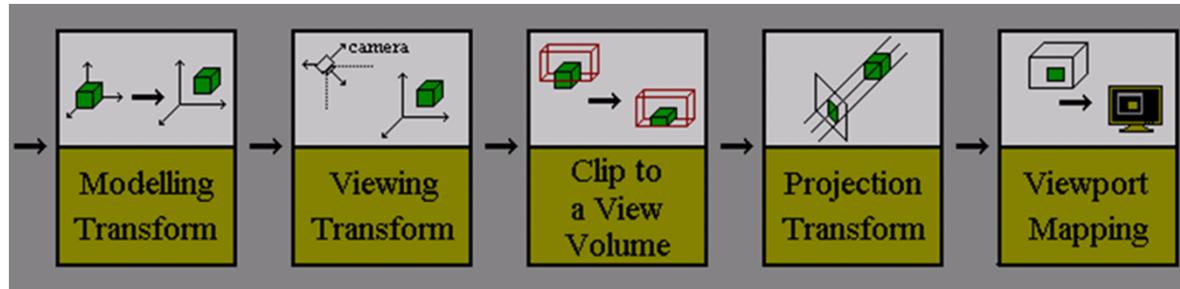
- Be careful of manipulating the matrix in OpenGL

- OpenGL uses **4X4** matrix for transformation.
- The 16 elements are stored as 1D in *column-major order*

$$\begin{pmatrix} a_0 & a_4 & a_8 & a_{12} \\ a_1 & a_5 & a_9 & a_{13} \\ a_2 & a_6 & a_{10} & a_{14} \\ a_3 & a_7 & a_{11} & a_{15} \end{pmatrix} \text{ OpenGL transform matrix}$$

- C and C++ store matrices in *row-major order*
- If you declare a matrix to be used in OpenGL as  
**GLfloat M[4][4];** to access the element in row i and column j, you need to refer to it by **M[j][i];** or, as  
**GLfloat M[16];** and then you need to convert it to conventional *row-major order.*

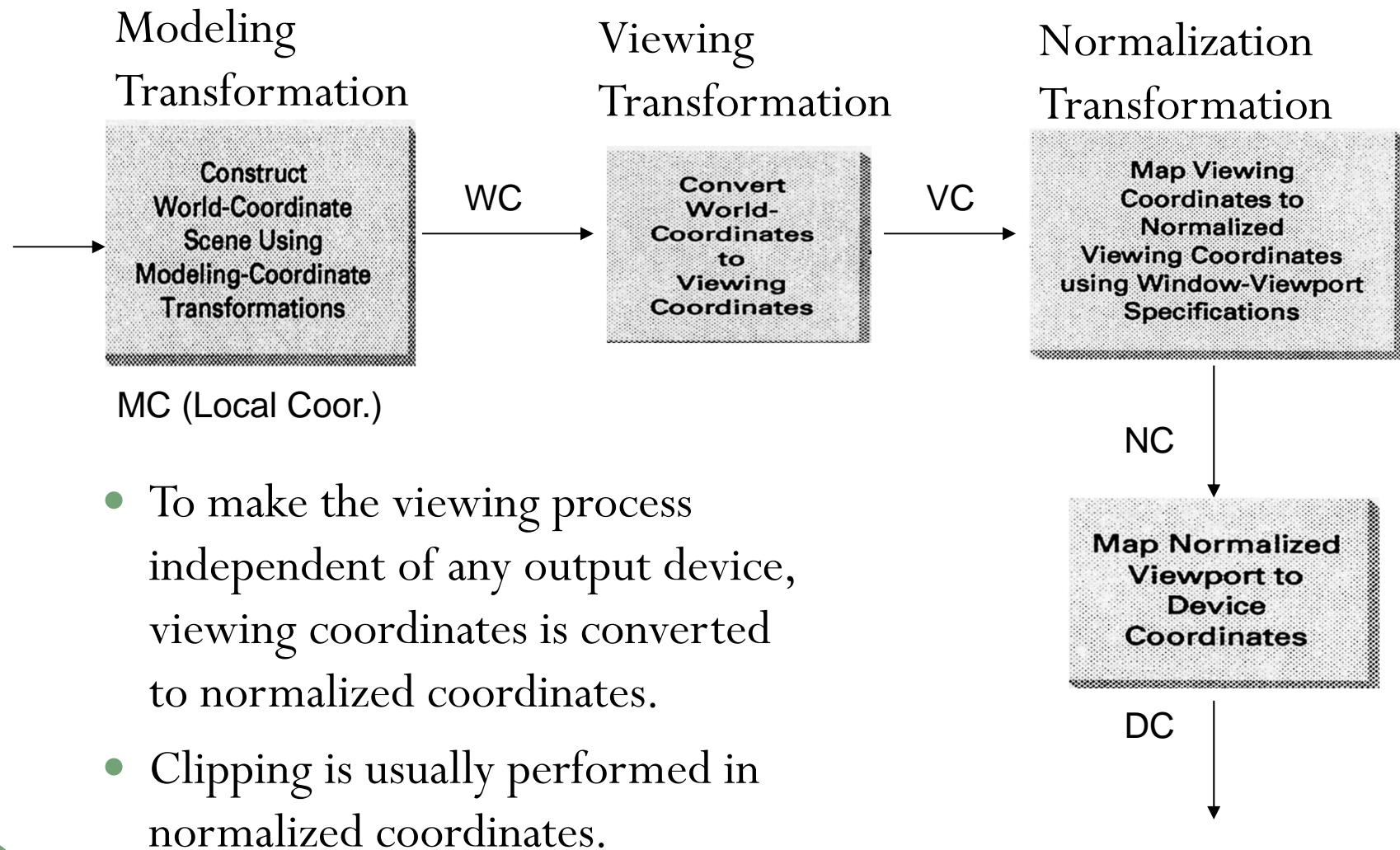
# OpenGL Transformations



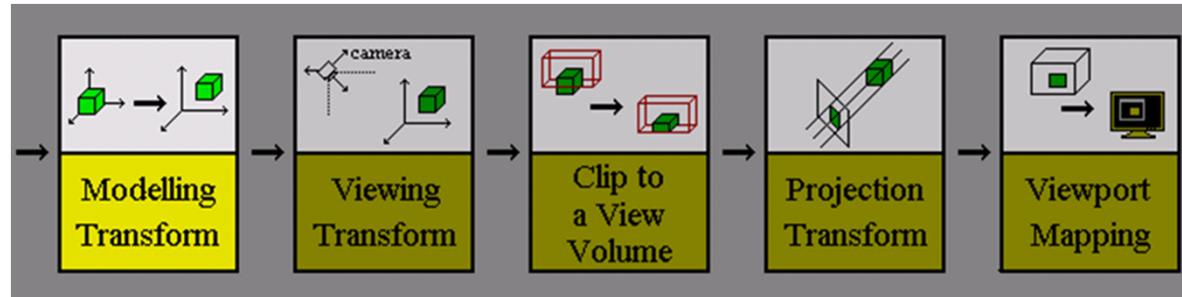
- Three types
  - Modeling, Viewing and Projection

Transformation	Use
Viewing	Specifies the location of the viewer or camera
Modeling	Moves objects around scene
Modelview	Describes the duality of viewing and modeling transformations
Projection	Clips and sizes the viewing volume
Viewport	Scales final output to the window

# Standard 2D Viewing Pipeline

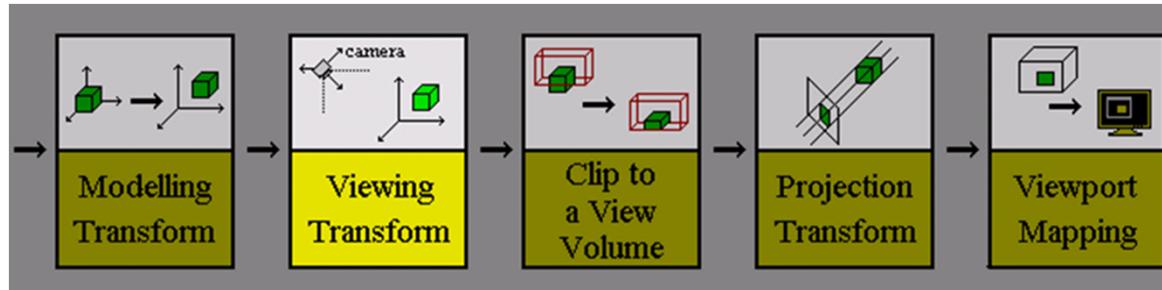


# Modeling Transformations



- *Modeling transformations:* to manipulate/**create** your model and the particular objects within it.
  - Move objects into place, rotates them, and scales them, etc.
  - The final appearance of your scene or object can depend greatly **on the order** in which the modeling transformations are applied.

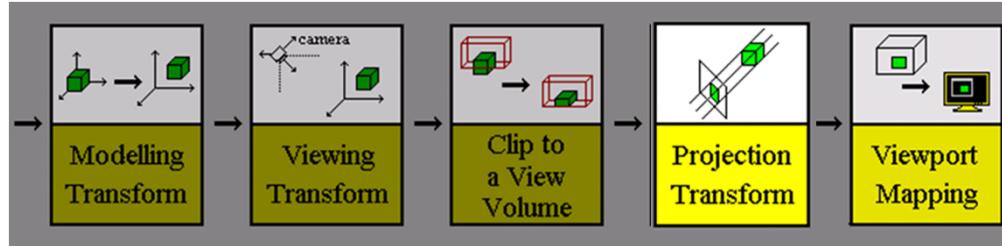
# Viewing Transformation



- *Viewing transformation:* to place & point a camera to view the scene.
  - By default, the point of observation is at the origin  $(0,0,0)$  looking down the negative z-axis (“into” the monitor screen).
    - Objects drawn with positive z values would be behind the observer.
  - You can put the point of observation anywhere you want, and looking in any direction.

Transformation demo - Nate

# Projection and Viewport Transformations



- *Projection transformation*: applied to your final Modelview orientation **in which way to project**.
  - Defines how a constructed scene (after all the modeling is done) is translated to the final 2D image on the viewing plane.
  - Defines the viewing volume and establishes clipping planes.
  - Two types
    - *Orthographic*
    - *Perspective*
- *Viewport transformation*: maps the 2D projection result of your scene to a window somewhere on your screen.

# OpenGL Transformations

- In OpenGL, all the transformations are described as a **multiplication of matrices**.
  - The mathematics behind these transformations are greatly simplified by the mathematical notation of the matrix.
  - Each of the transformations can be achieved by multiplying a **matrix** that contains the vertices, by a **matrix** that describes the transformation.

# OpenGL Geometric Transformation Functions

- OpenGL matrix operation function

```
void glMatrixMode(GLenum mode);
```

- Specify which matrix is the current matrix
- *mode*: **GL\_MODELVIEW**, **GL\_PROJECTION**, **GL\_TEXTURE**;

**GL\_COLOR** (if ARB\_imaging extension is supported).

e.g.: `glMatrixMode (GL_MODELVIEW);`

- OpenGL matrix operations

```
glMatrixMode (GL_MODELVIEW);
```

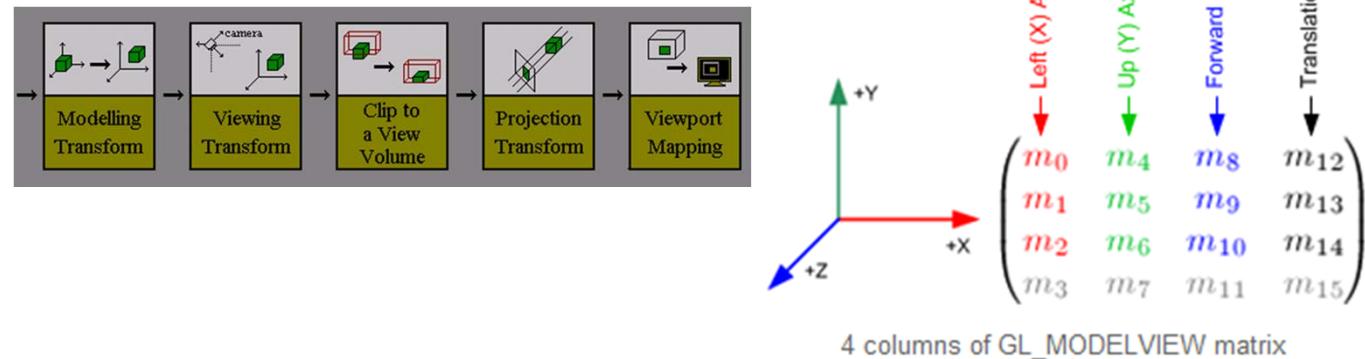
Set up the matrix for  
geometric transformations

```
glLoadIdentity (); // assign identity matrix to the current matrix
```

```
// ... to apply any transformation matrix to transform your scene ...
```

# Model-View Matrix: GL\_MODELVIEW

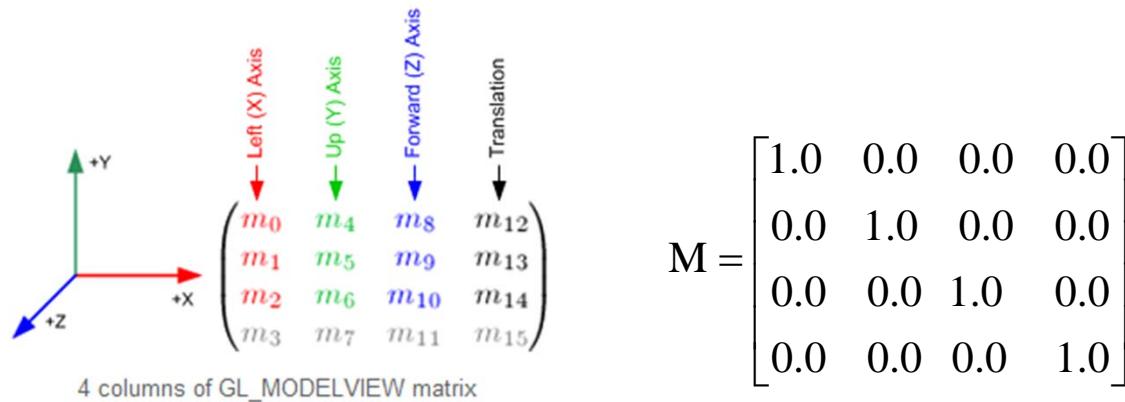
- GL\_MODELVIEW



- Store and combine the geometric transformations to models and viewing-coordinate system
  - Combine viewing matrix and **modeling matrix** into one matrix
- Viewing transformation
  - For example: **gluLookAt()**
- Modeling transformation: OpenGL transformation functions
  - Translation transformation:  $m_{12}, m_{13}, m_{14}$
  - Other Euclidean/affine transformations, such as rotation or scaling:  $(m_0, m_1, m_2), (m_4, m_5, m_6)$  and  $(m_8, m_9, m_{10})$

# Model-View Matrix: GL\_MODELVIEW

- GL\_MODELVIEW



glMatrixMode (GL\_MODELVIEW);

glLoadIdentity (); // assign identity matrix to the current matrix

These 3 sets

- $(m0, m1, m2)$  : +X axis, *left* vector,  $(1, 0, 0)$  by default
- $(m4, m5, m6)$  : +Y axis, *up* vector,  $(0, 1, 0)$  by default
- $(m8, m9, m10)$  : +Z axis, *forward* vector,  $(0, 0, 1)$  by default

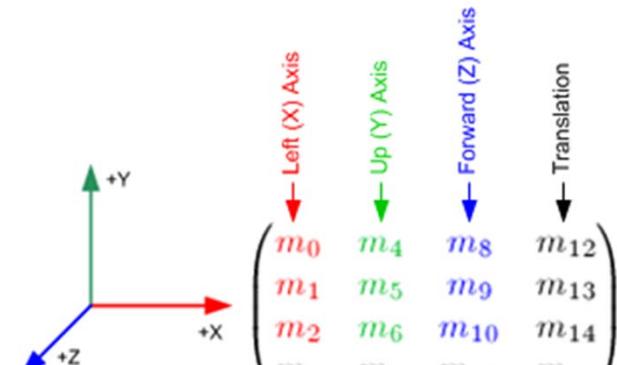
are actually representing 3 orthogonal axes.

glLoadMatrix\* (elements16); // replace the current matrix by your own

# Model-View Matrix: GL\_MODELVIEW

## Example

```
glMatrixMode (GL_MODELVIEW);
glLoadIdentity ();
GLfloat elems [16];
GLint k;
for (k = 0; k < 16; k++)
    elems [k] = float (k);
glLoadMatrixf (elems);
```



$$M = \begin{bmatrix} 0.0 & 4.0 & 8.0 & 12.0 \\ 1.0 & 5.0 & 9.0 & 13.0 \\ 2.0 & 6.0 & 10.0 & 14.0 \\ 3.0 & 7.0 & 11.0 & 15.0 \end{bmatrix}$$

## glMatrixMode()

<http://www.opengl.org/sdk/docs/man/>

# OpenGL Geometric Transformation Functions

- Basic OpenGL geometric transformations on the matrix:

**glTranslate\*** (tx, ty, tz);

[ glTranslatef (25.0, -10.0, 0.0); ] for 2D, set tz = 0.

- Post-multiplies the current matrix by a matrix that moves the object by the given x-, y-, and z-values

**glScale\*** (sx, sy, sz);

[ glScalef (2.0, -3.0, 1.0); ]

- Post-multiplies the current matrix by a matrix that scales an object about the origin.  
**None** of sx, sy or sz is **zero**.

**glRotate\*** (theta, vx, vy, vz);

[ glRotatef (90.0, 0.0, 0.0, 1.0); ]

- Post-multiplies the current matrix by a matrix that rotates the object in a counterclockwise direction. vector v=(vx, vy, vz) defines the orientation for the rotation axis that passes through the coordinate origin. ( the rotation center is (0, 0, 0) )

# OpenGL: Order in Matrix Multiplication

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();      //Set current matrix to the identity.
glMultMatrixf(elemsM2); //Post-multiply identity by matrix M2.
glMultMatrixf(elemsM1); //Post-multiply M2 by matrix M1.
glBegin(GL_POINTS)
    glVertex3f(vertex);
glEnd();
```

Modelview matrix successively contains:

$I$ (identity),  $M_2$ ,  $M_2 \cdot M_1$

The concatenated matrix is:

$$M = M_2 \cdot M_1$$

The transformed vertex is:

$$M_2 \cdot (M_1 \cdot \text{vertex})$$

In OpenGL, a transformation sequence is applied in reverse order of which it is specified.

# OpenGL: Order in Matrix Multiplication

- Example

```
// rotate object 30 degrees around Z-axis  
glRotatef(30.0, 0.0, 0.0, 1.0);  
  
// move object to (2.0, 3.0, 0.0)  
glTranslatef(2.0, 3.0, 0.0);  
  
drawObject();
```

The object will be **translated** first then **rotated**.

# OpenGL Geometric Trans. Programming Examples

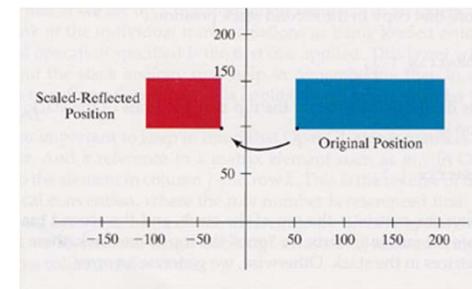
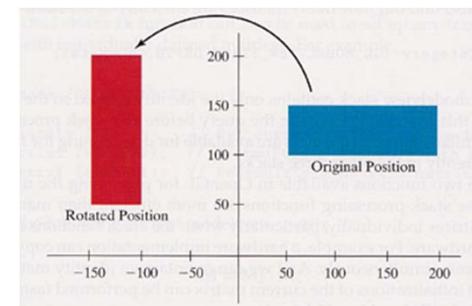
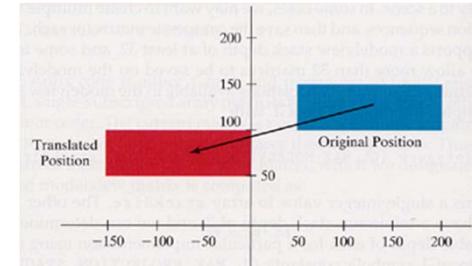
```
glMatrixMode(GL_MODELVIEW); //Identity matrix
```

```
glColor3f(0.0, 0.0, 1.0);      // Set current color to blue  
glRecti(50, 100, 200, 150);   // Display blue rectangle.
```

```
glColor3f(1.0, 0.0, 0.0);      // Red  
glTranslatef(-200.0, -50.0, 0.0); // Set translation parameters.  
glRecti(50, 100, 200, 150);   // Display red, translated rectangle.
```

```
glLoadIdentity();             // Reset current matrix to identity.  
glRotatef(90.0, 0.0, 0.0, 1.0); // Set 90-deg, rotation about z axis.  
glRecti(50, 100, 200, 150);   // Display red, rotated rectangle.
```

```
glLoadIdentity();             // Reset current matrix to identity.  
glScalef(-0.5, 1.0, 1.0);    // Set scale-reflection parameters.  
glRecti(50, 100, 200, 150);   // Display red, transformed rectangle.
```



# Summary

- Basic 2D geometric transformations
  - Translation
  - Rotation
  - Scaling
  - Reflection, shearing...
  - Combination of these transformations
- Homogeneous coordinate representation
- OpenGL geometric transformation functions
  - GL\_MODELVIEW matrix
  - Order in multiple matrix multiplication
  - Example

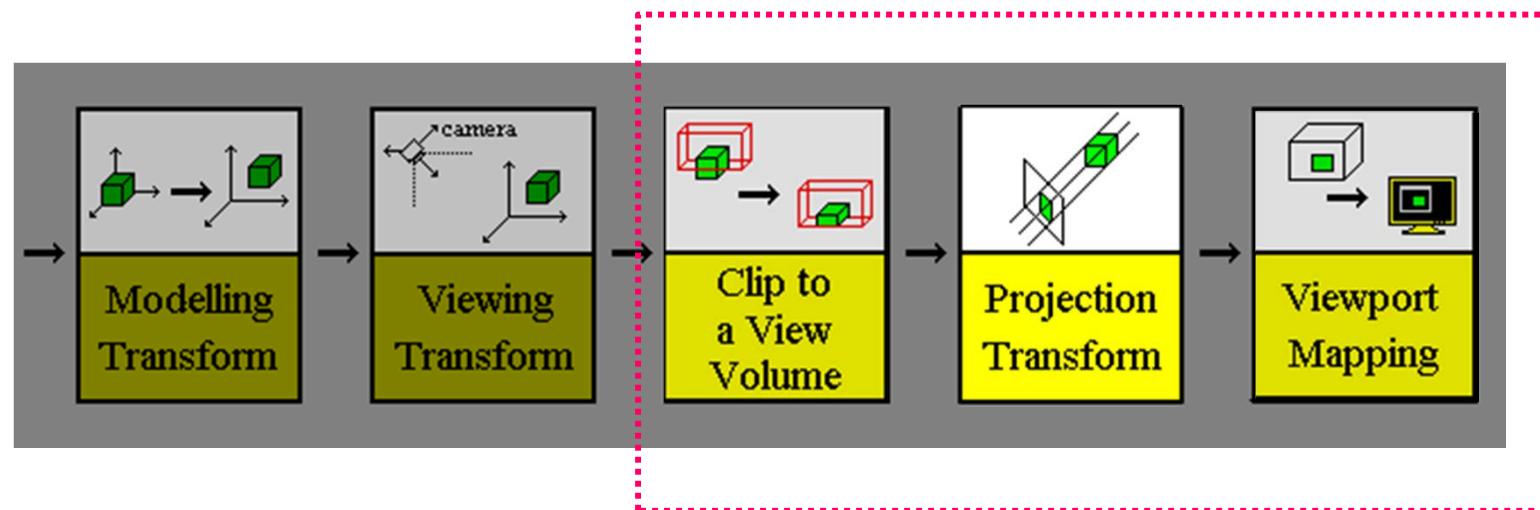
Computer Graphics

Chapter 8 (I)  
Two - Dimensional Viewing

# Outline

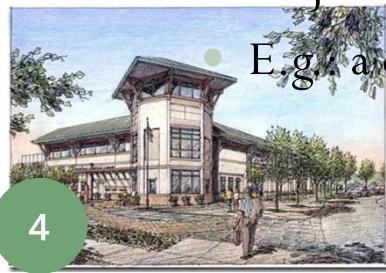
- Viewing Pipeline
- World Coordinates Transfer to Viewing Coordinates
- Normalization and Viewport Transformations
- OpenGL 2D Viewing Functions
- OpenGL 2D Viewing Program Example

# Viewing Pipeline

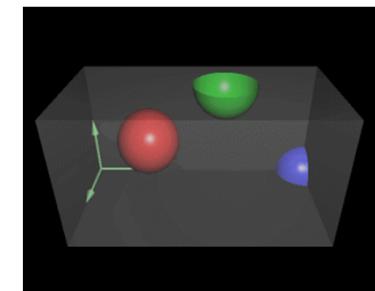


# Viewing Volume

- Viewing volume
  - A closed volume which delimits the **infinite** 3D space to **finite** volume.
  - Points outside it will not appear on the screen.
- Two projections to create viewing volume
  - Orthographic projection
  - Objects rendered are not affected by the distance
  - E.g.: a menu, a text on a screen, 2D objects...
- Perspective projection



4



Orthographic



Perspective



Orthographic

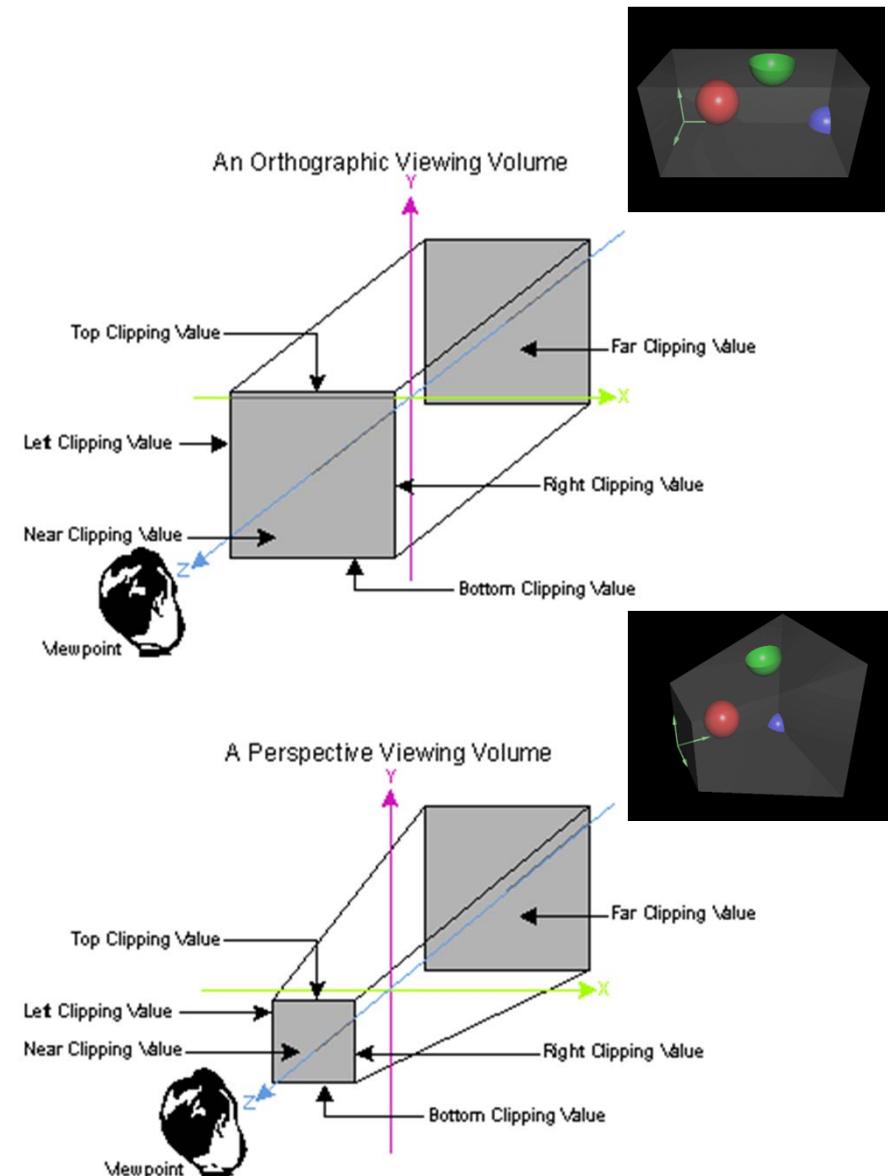


Perspective

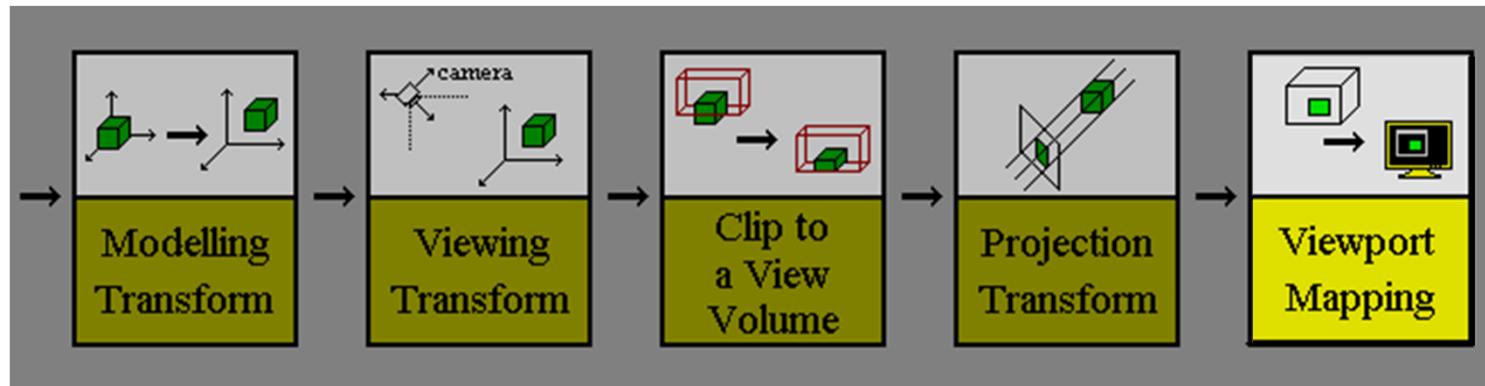
( From OpenGL Super Bible )

# Viewing Volume

- Orthographic projection
  - Viewing volume shape is a parallelepiped(平行六面體).
  - Parallel clipping planes
- Perspective projection
  - Viewing volume shape is a truncated pyramid (its top is cut).
  - Non-parallel side clipping planes



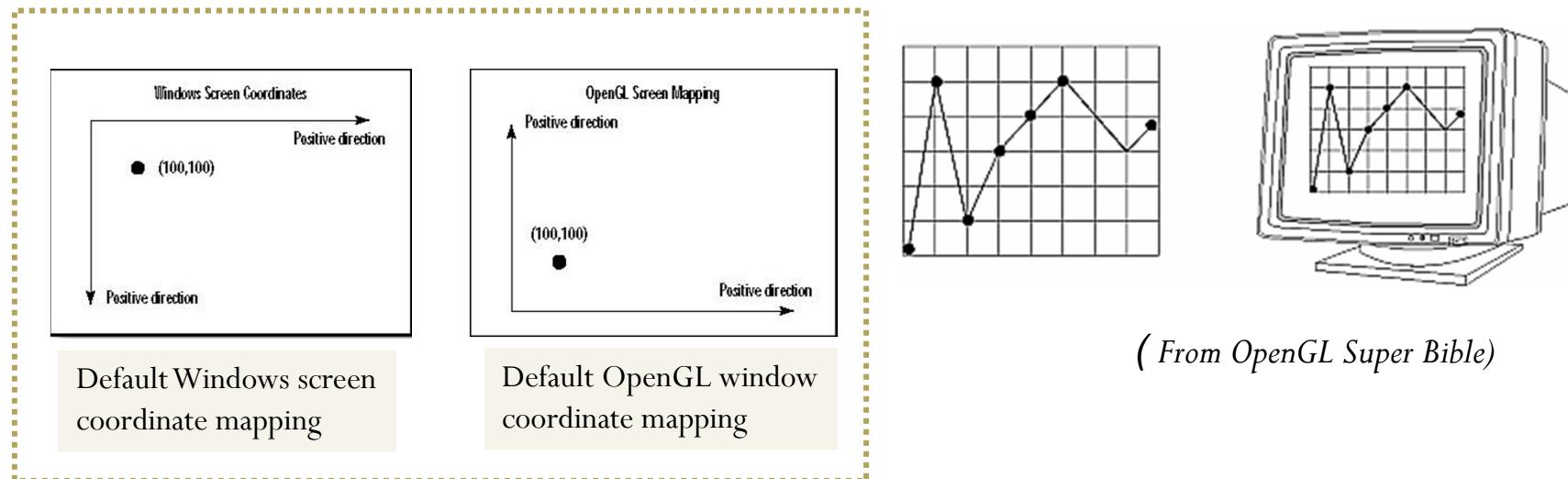
# Viewport



- Viewport
  - 2D drawing region of the screen where the final result is mapped.

# Viewport

- Measured in actual window coordinates



(From OpenGL Super Bible)

```
void glViewport (GLint x, GLint y, GLsizei width, GLsizei height);
```

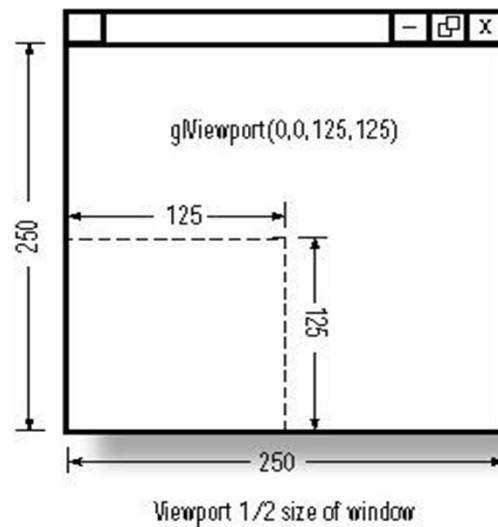
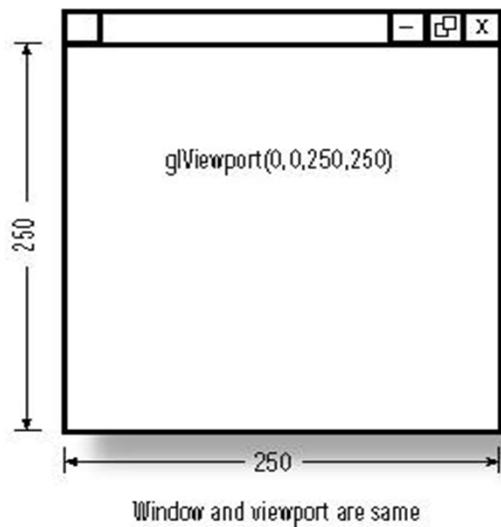
$(x, y)$ : specifies the lower-left corner of the viewport;

$width$  and  $height$ : the size of the viewport rectangle.

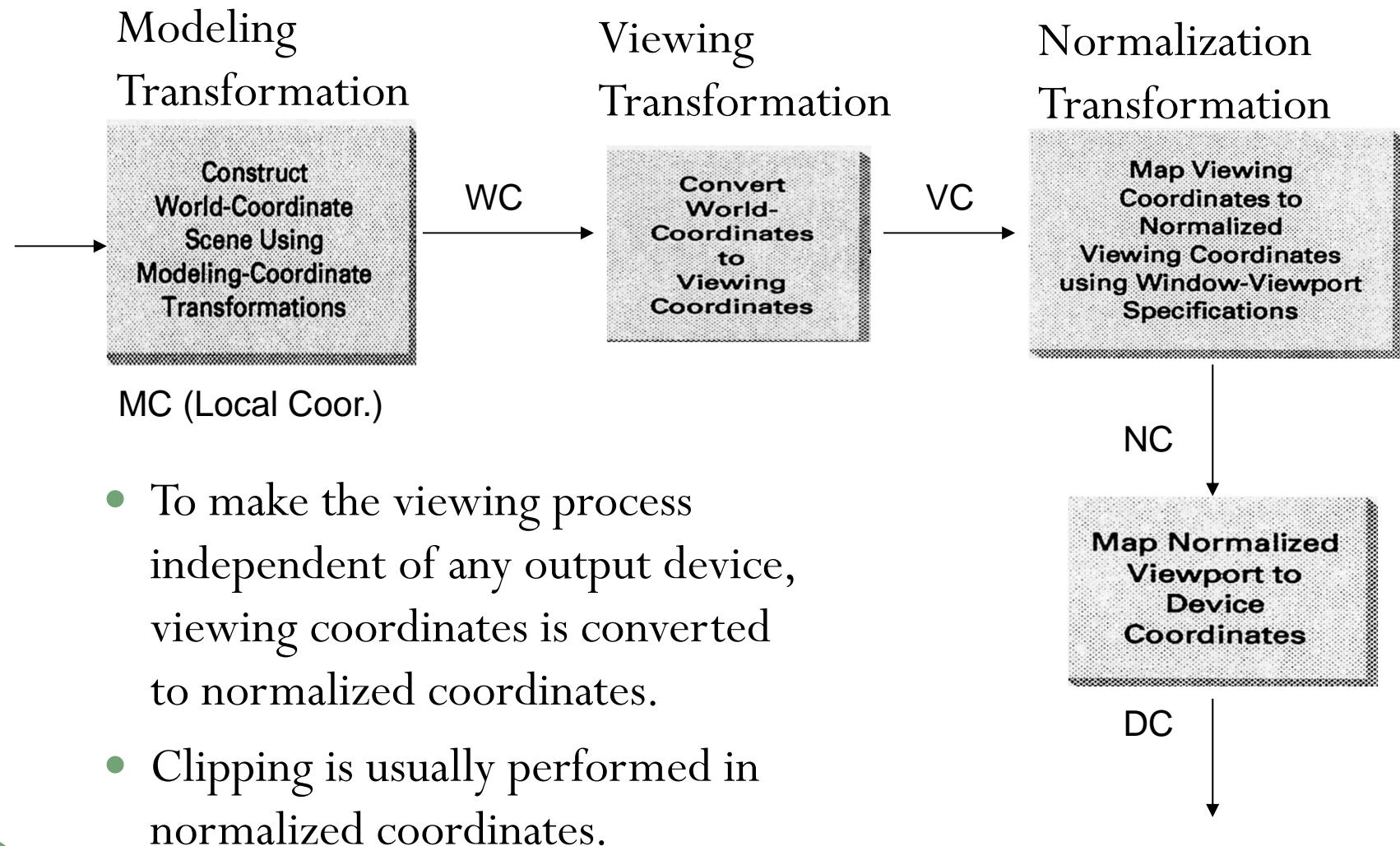
**By default**, the initial viewport are  $(0, 0, \text{winWidth}, \text{winHeight})$ , where  $\text{winWidth}$  and  $\text{winHeight}$  are the size of the window.

# Viewport

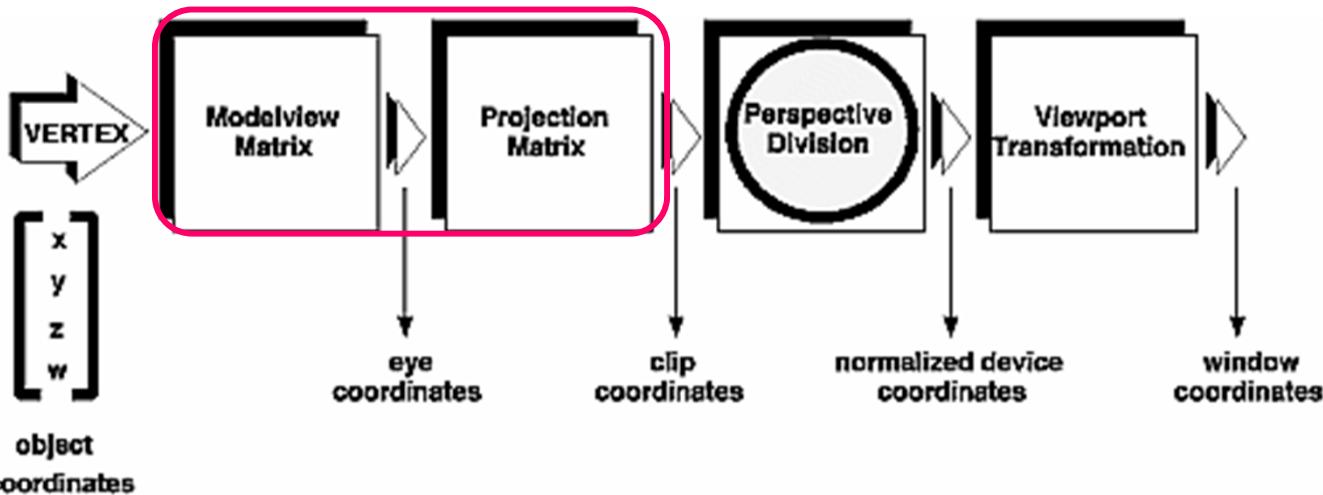
- Example



# 2D Viewing Pipeline

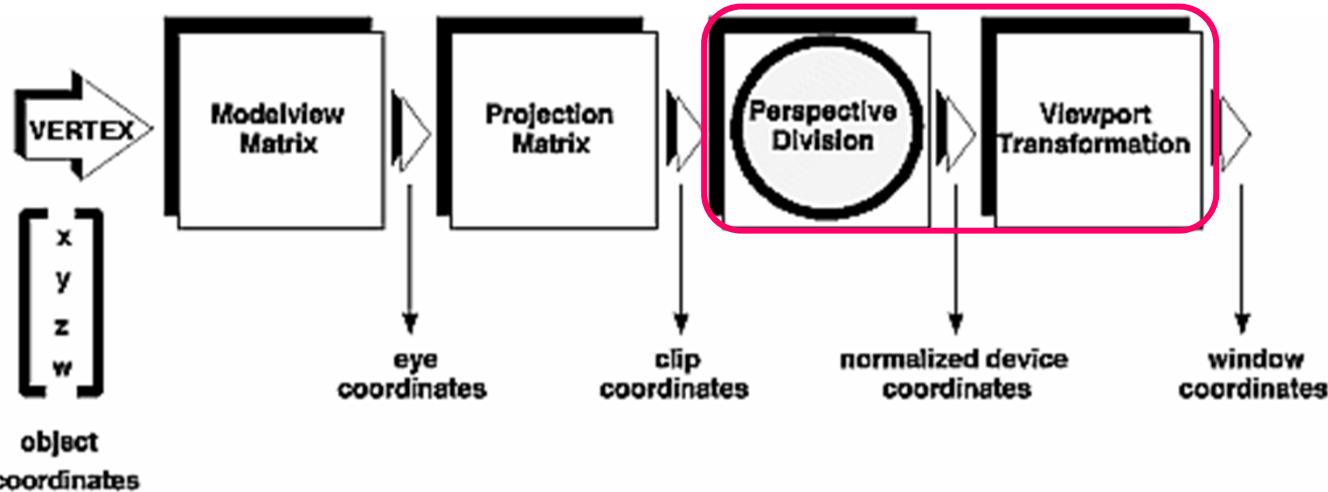


# Stages of Vertex Transformation



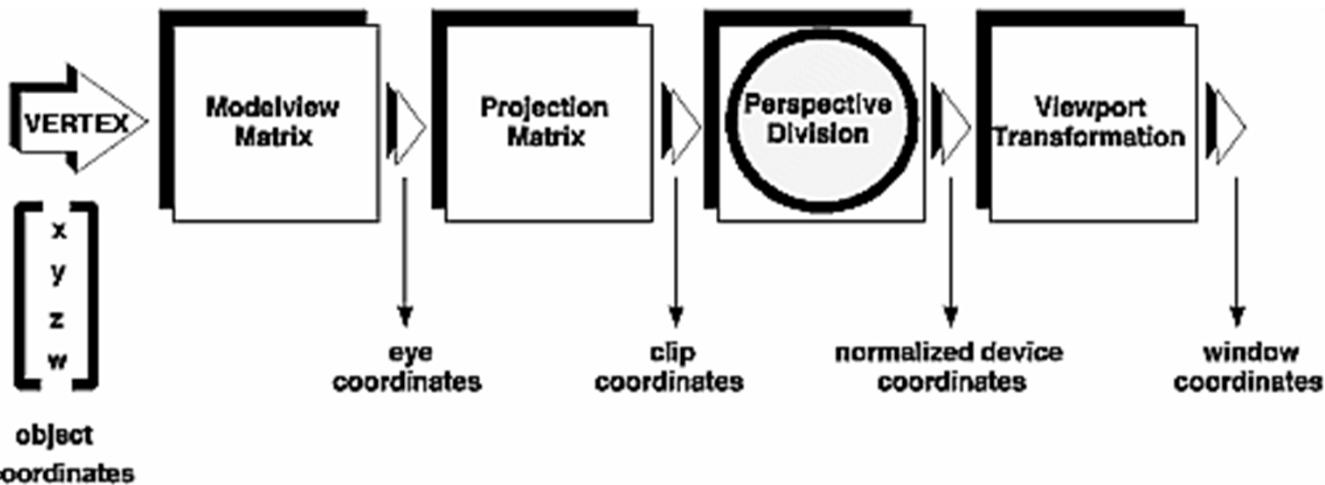
- Matrix-form in OpenGL
  - The **viewing** and **modeling** transformations you specify are combined to form the **modelview** matrix, which is applied to the incoming *object coordinates* to yield **eye (viewing) coordinates**.
  - The **projection** matrix to yield **clip coordinates**.
    - Defines a viewing volume

# Stages of Vertex Transformation



- Matrix form in OpenGL (cont.)
  - The *perspective division* is performed by dividing coordinate values by  $w$ , to produce *normalized device coordinates*.
  - The transformed coordinates are converted to *window coordinates* by applying the *viewport transformation*.
    - You can specify the size of the viewport to cause the final image result.

# Stages of Vertex Transformation

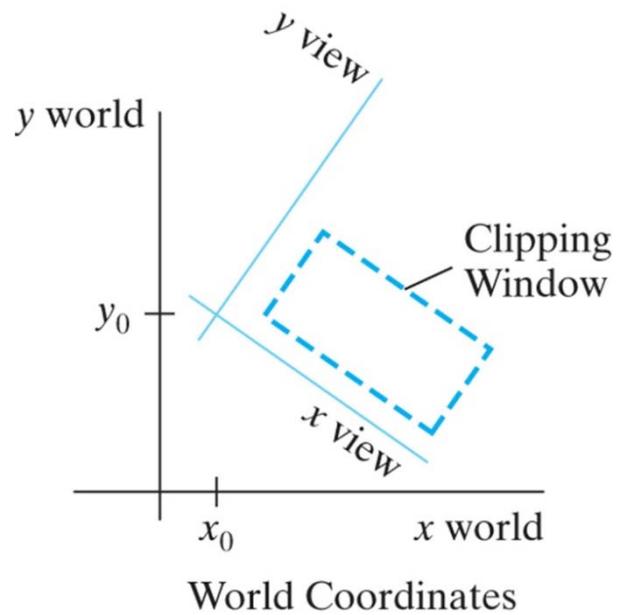


- The viewing, modeling, and projection transformations matrix in OpenGL: **4×4 matrix  $M$**  [in 2D,  $z = 0$ ]
- They are multiplied by the coordinates of each vertex  $v$  in the scene

$$v' = Mv$$

# World Coordinates Convert to Viewing Coordinates

We can set up a 2D viewing coordinate system in the world coordinate frame.



$(x_0, y_0)$ : an origin

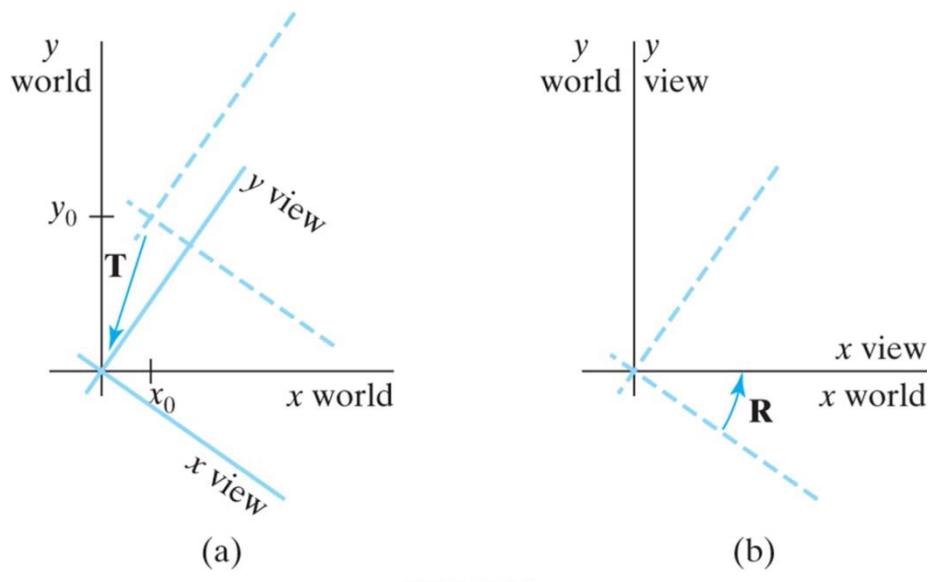
$y_{view}$ : 2D view up vector

This viewing coordinate frame provides a reference for specifying the clipping window.



**Fig. 8-3** A rotated world window in *Viewing Coordinates*.

# World Coordinates Convert to Viewing Coordinates



**FIGURE 8-4** A *Viewing-Coordinate* frame is moved into coincidence with the *World - Coordinate* frame by

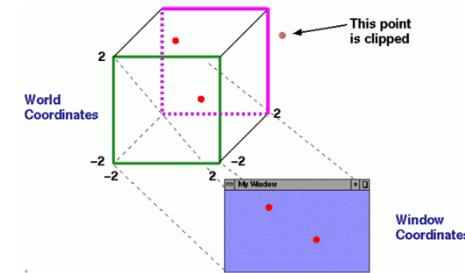
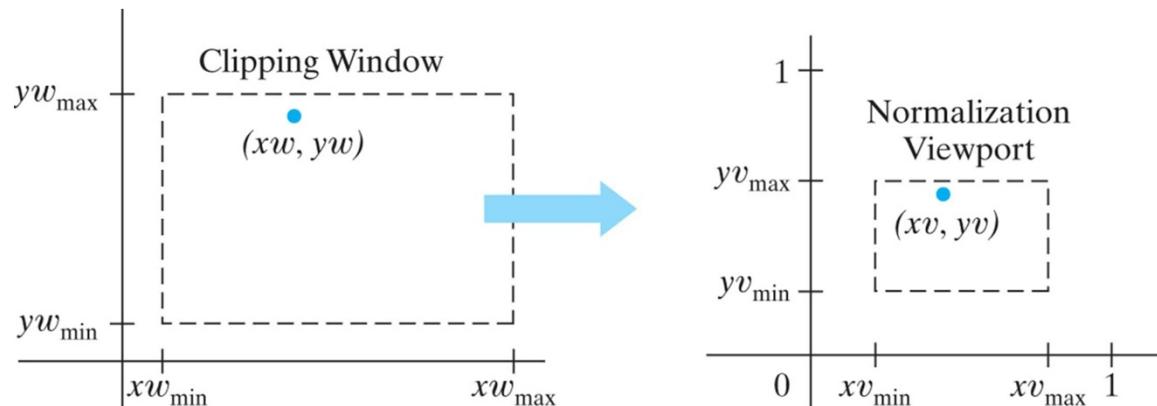
- (a) applying a **translation** matrix  $\mathbf{T}$  to move the viewing origin to the world origin, then
- (b) applying a **rotation** matrix  $\mathbf{R}$  to align the axes of the two systems.

$$\mathbf{M}_{\text{wc-} \rightarrow \text{vc}} = \mathbf{R} \bullet \mathbf{T} \quad (8-1)$$

Where  $\mathbf{T}$  is the translation matrix that takes the viewing origin point  $P_0$  to the world origin, and  $\mathbf{R}$  is the rotation matrix that aligns the axes of the two reference frames

# Normalization and Viewport Transformations

- Mapping a clipping window into a normalized viewport  
(the viewport is given within ( [0,1], [0,1] ) )



$$\frac{xv - xv_{\min}}{xv_{\max} - xv_{\min}} = \frac{xw - xw_{\min}}{xw_{\max} - xw_{\min}}$$

$$\frac{yv - yv_{\min}}{yv_{\max} - yv_{\min}} = \frac{yw - yw_{\min}}{yw_{\max} - yw_{\min}}$$

**FIGURE 8-6** A point  $(xw, yw)$  in a world-coordinate clipping window is mapped to viewport coordinates  $(xv, yv)$ , within a unit square, so that the relative positions of the two points in their respective rectangles are the same.

# OpenGL 2D Viewing Functions

- OpenGL Projection Mode

```
glMatrixMode (GL_PROJECTION); //projection matrix  
glLoadIdentity ();
```

- GLU Clipping-Window Function

```
gluOrtho2D (xwmin, xwmax, ywmin, ywmax);  
2D parallel projection.
```

- OpenGL Viewport Function

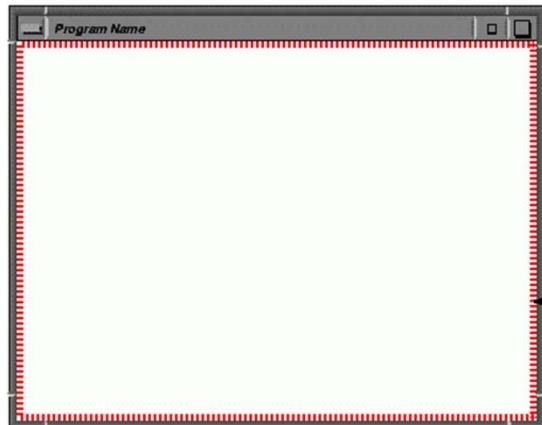
```
glViewport (xvmin, yvmin, vpWidth, vpHeight);
```

```
glGetIntegerv (GL_VIEWPORT, vpArray);
```

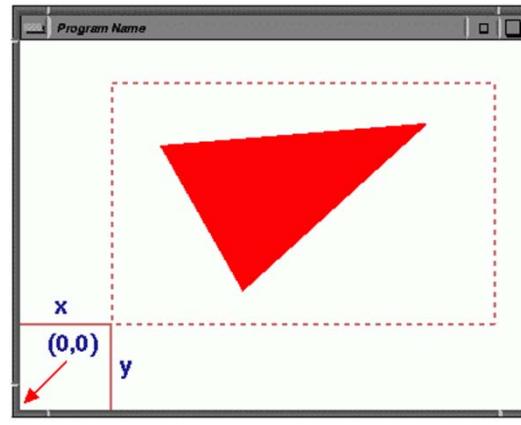
To obtain the parameters for the currently active viewport: xvmin, yvmin, vpWidth, vpHeight.

# OpenGL Viewport Function

- `glViewport (GLint x, GLint y, GLsizei width, GLsizei height);`



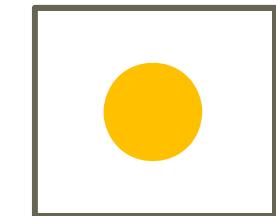
Initial Viewport



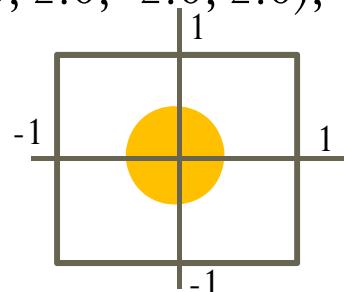
You can change it by  
`glViewport()`.

Always rectangle;  
`x, y` are in window coordinates

- Example: `gluOrtho2D (-2.0, 2.0, -2.0, 2.0);`

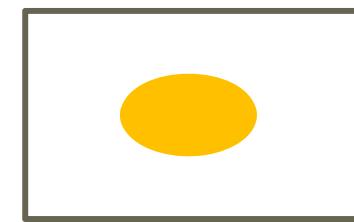


World Coordinates



Normalized Device Coordinates

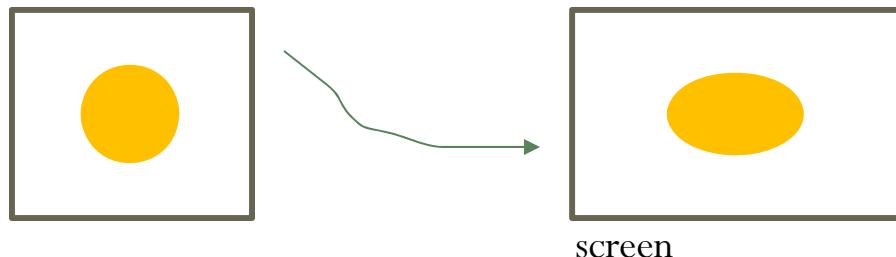
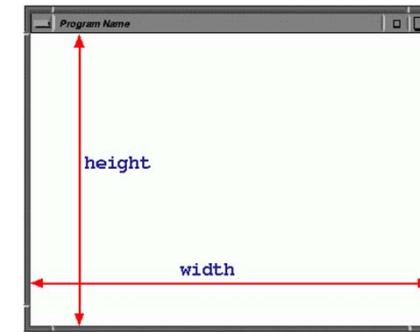
`glViewport (0, 0, 300, 150);`



Window Coordinates

# OpenGL Viewport Function

- The rectangle area has an aspect ratio: width / height
  - Windows
    - `glutInitWindowSize ( width, height );`
  - 2D clipping window
    - `gluOrtho2D ( left, right, bottom, top );`
  - Viewport
    - `glViewport ( x, y, width, height);`
- In general, the clipping window (viewing volume) and viewport need to have the same ratio.

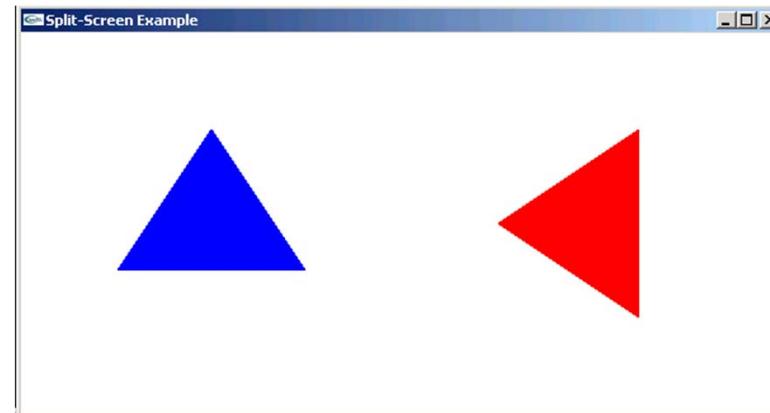


# OpenGL 2D Viewing Program Example

- Two views of a triangle in the xy plane shown in a split screen, with its centroid at the world-coordinate origin

```
#include <GL/glut.h>

class wcPt2D
{
public:
    GLfloat x, y;
};
```



# OpenGL 2D Viewing Program Example

```
void init (void)
{
    /* Set color of display window to white. */
    glClearColor (1.0, 1.0, 1.0, 0.0);

    /* Set parameters for world-coordinate clipping window. */
    glMatrixMode (GL_PROJECTION);
    gluOrtho2D (-100.0, 100.0, -100.0, 100.0);

    /* Set mode for construction geometric transformation matrix. */
    glMatrixMode (GL_MODELVIEW);
}
```

# OpenGL 2D Viewing Program Example

```
void triangle (wcPt2D *verts)
{
    GLint k;

    glBegin (GL_TRIANGLES);
        for (k =0; k < 3; k++)
            glVertex2f (verts [k].x, verts [k].y);
    glEnd ();
}
```

# OpenGL 2D Viewing Program Example

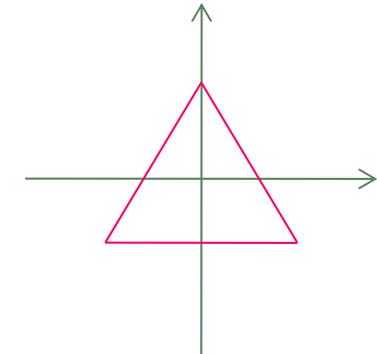
```
void displayFcn (void)
{
    /* Define initial position for triangle. */
    wcPt2D verts [3] = {{-50.0, -25.0}, {50.0, -25.0}, {0.0, 50.0}};

    glClear (GL_COLOR_BUFFER_BIT);      // Clear display window.

    glColor3f (0.0, 0.0, 1.0);        // Set fill color to blue.
    glViewport (0, 0, 300, 300);    // Set left viewport.
    triangle (verts);                // Display red rotated triangle.

    /* Rotate triangle and display in right half of display window. */
    glColor3f (1.0, 0.0, 0.0);        // Set fill color to red.
    glViewport (300, 0, 300, 300);    // Set right viewport.
    glRotatef (90.0, 0.0, 0.0, 1.0); // Rotate about z axis.
    triangle (verts);                // Display red rotated triangle.

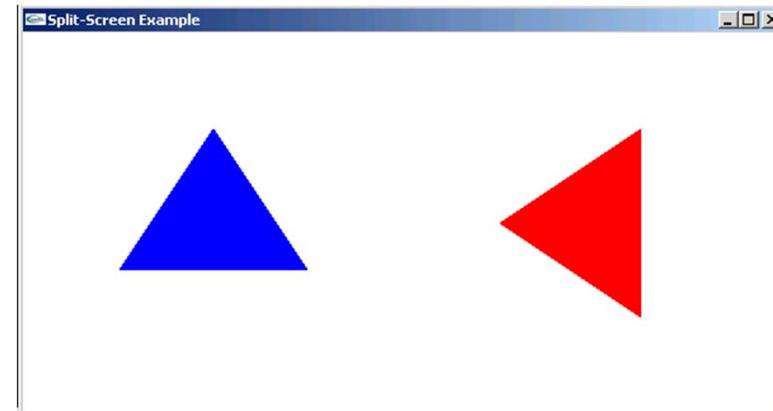
    glFlush (); //Force to execute all OpenGL functions
}
```



# OpenGL 2D Viewing Program Example

```
void main (int argc, char **argv)
{
    glutInit (&argc, argv);
    glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
    glutInitWindowPosition (50, 50);
    glutInitWindowSize (600, 300);
    glutCreateWindow ("Split-Screen Example");

    init();
    glutDisplayFunc (displayFcn);
    glutMainLoop ();
}
```



# OpenGL 2D Viewing Functions

- You need to handle the changes in the window size
  - Registering a window reshape callback
    - void glutReshapeFunc ( void (\*func) (int width, int height) );
  - Defining the reshape callback function: pass the new width and height of the window
    - called before the first call to the display function,
    - and called automatically when the window is reshaped.
    - e.g. MyReshape ();

```
void MyReshape ( int width, int height )
{
    /* update viewport */
    glViewport (...);
    /* reset viewing volume */
    glMatrixMode ( GL_PROJECTION );
    glLoadIdentity();
    gluOrtho2D (...);
    /* set modelview matrix mode
     ...
}
```

# Summary

- 2D viewing pipeline
- OpenGL 2D viewing functions

Computer Graphics

Chapter 8 (II)  
Two - Dimensional Viewing(Clipping)

# Outline

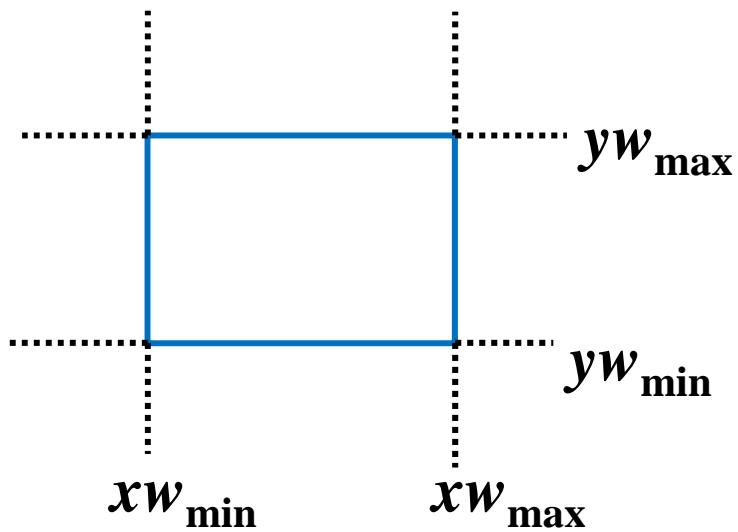
- 2D Point Clipping
- 2D Line Clipping
- Polygon Fill-Area Clipping
- Text Clipping

# 2D Point Clipping

- For a clipping rectangle, which is defined by  $(xw_{\min}, yw_{\min})$  and  $(xw_{\max}, yw_{\max})$

$$xw_{\min} \leq x \leq xw_{\max} \quad (8-12)$$

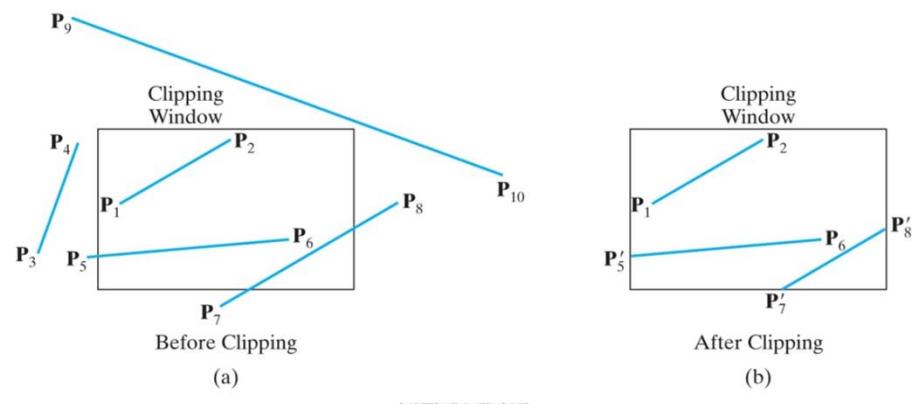
$$yw_{\min} \leq y \leq yw_{\max}$$



# 2D Line Clipping

- Basic process
  - Each line goes through the tests and intersection calculations
- Optimization
  - Minimize the intersection calculations
  - Special cases: which are completely inside or outside a clipping window.
  - Then, general: determine the intersections

Expensive part

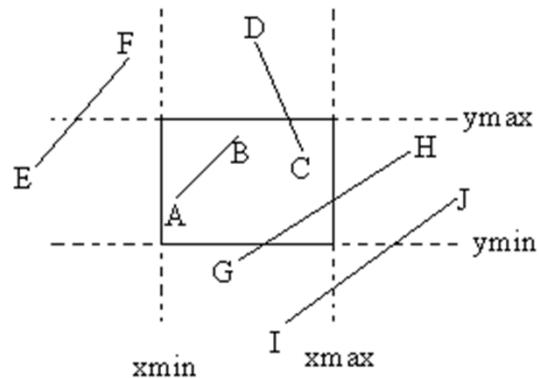


Parametric representation:

$$\begin{cases} x = x_1 + u(x_2 - x_1) \\ y = y_1 + u(y_2 - y_1), \quad 0 \leq u \leq 1 \end{cases}$$

# 2D Line Clipping

- *Cohen - Sutherland Line Clipping Method*



- Relationship between the line segment and the clipping window
  - Completely inside;
  - Completely outside;
  - Other cases

# Cohen - Sutherland Line Clipping

- The line endpoint is encoded by a **region code**.

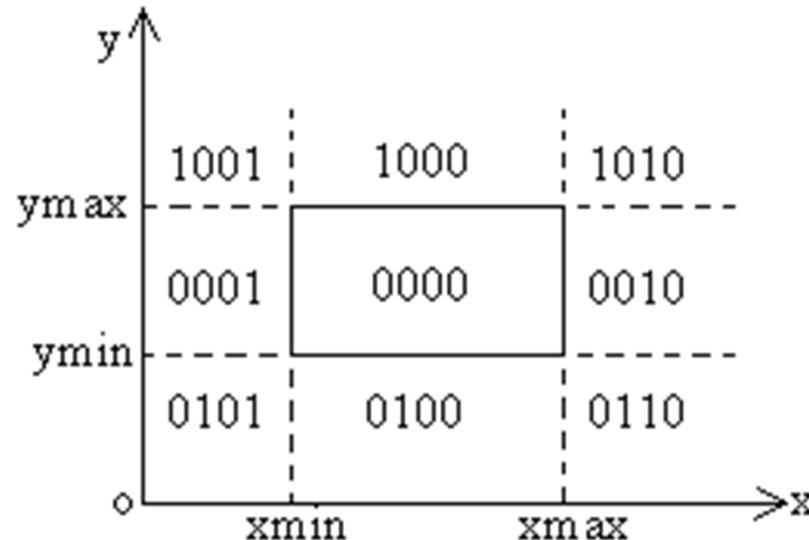
The four clipping-window edges divide 2D space into 9 regions, each of which corresponds to 4-bit code:  $C_t C_b C_r C_l$

$$C_t = \begin{cases} 1 & \text{if } y > y_{\max} \\ 0 & \text{else} \end{cases}$$

$$C_b = \begin{cases} 1 & \text{if } y < y_{\min} \\ 0 & \text{else} \end{cases}$$

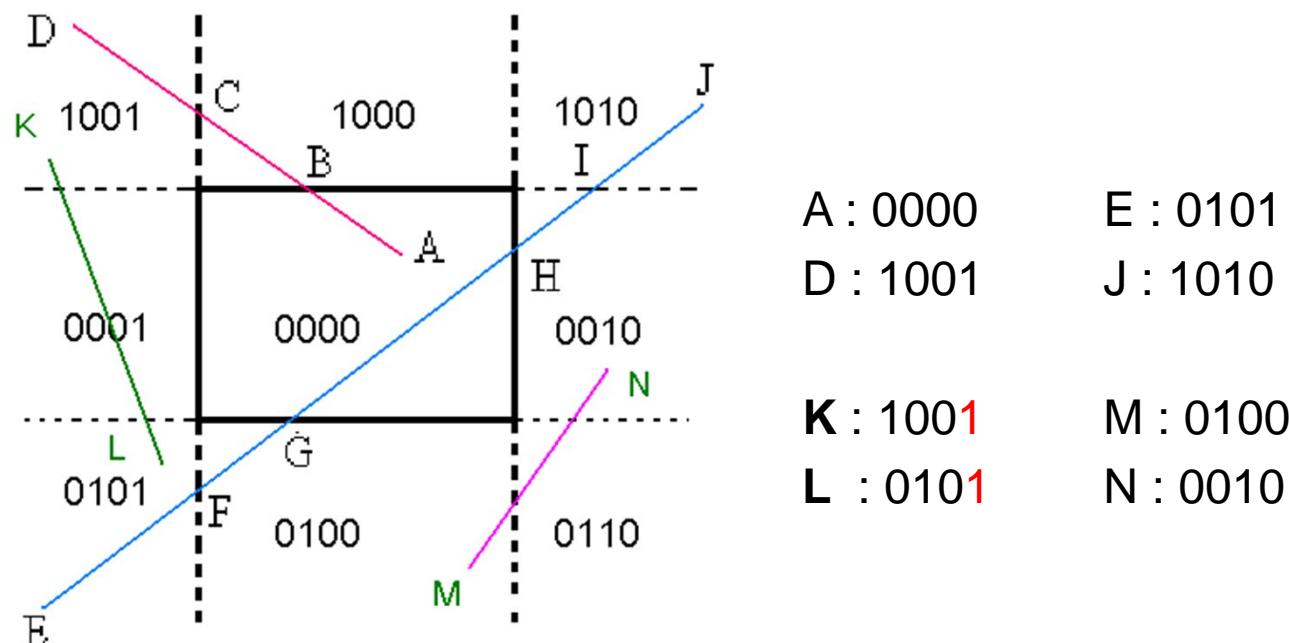
$$C_r = \begin{cases} 1 & \text{if } x > x_{\max} \\ 0 & \text{else} \end{cases}$$

$$C_l = \begin{cases} 1 & \text{if } x < x_{\min} \\ 0 & \text{else} \end{cases}$$



# Cohen - Sutherland Line Clipping

- Endpoint encoding – the region code which it belongs to.
- Conclusion:
  - When the logic ‘**and**’ operation of two endpoints is **nonzero**, the line segment is completely outside (obviously invisible);
  - When the region code 0000 for **both** endpoints, the line segment is completely inside.

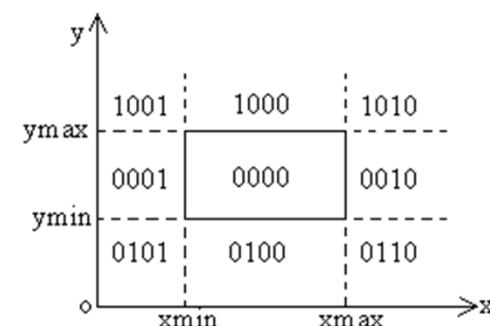
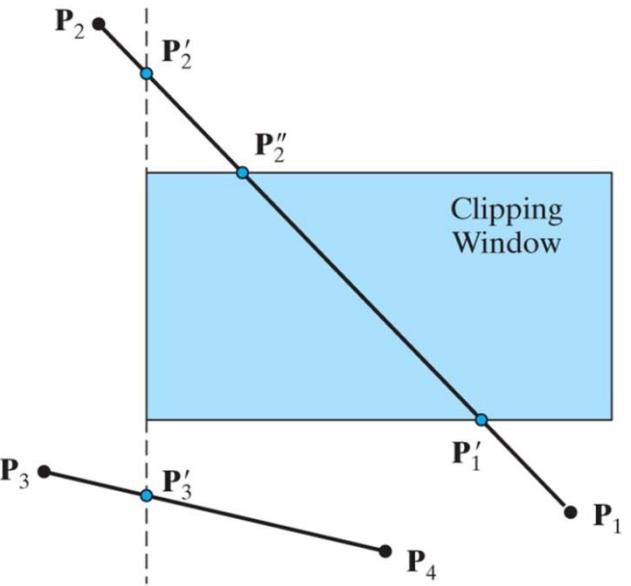


# Cohen - Sutherland Line Clipping

- *Cohen - Sutherland Line Clipping*
  - Produce a **region code** for each endpoint of the line segment, e.g.  $c1$  and  $c2$  for both endpoints
  - Check the region codes of the line segment:
    - ❖ If both  $c1$  and  $c2$  are '0000's, the line segment is within the clip window; (**the inside case**)
    - ❖ If  $(c1 \& c2) \neq '0000'$  the line segment is fully outside the clip window; (**the outside case**)
    - ❖ Otherwise, a next check is made for the intersection of the line with the window boundaries and the intersection are calculated. (HowTO: **the next slide**)

# Cohen - Sutherland Line Clipping

- Lines that cannot be identified as being completely inside or outside the clipping window, to process : L -> R -> B -> T
- P<sub>1</sub>P<sub>2</sub>: P<sub>1</sub>(0100), P<sub>2</sub>(1001) – **left boundary**
  - Calculate the intersection P<sub>2</sub>';
  - Clip off P<sub>2</sub>P<sub>2</sub>';  
The remain is inside the right border line (the 2nd-bit are same, no need to check “R”);
- Next, check the **bottom** border line.
  - P<sub>1</sub> is below, P<sub>2</sub>' is above it; [P<sub>1</sub>(0100), P<sub>2</sub>(1001)]
  - Calculate the intersection P<sub>1</sub>';
  - Clip off P<sub>1</sub>P<sub>1</sub>';
- Next, check the **top** border line; [P<sub>1</sub>(0100), P<sub>2</sub>(1001)]
  - Get the intersection P<sub>2</sub>”;
  - Clip off P<sub>2</sub>'P<sub>2</sub>”.



# Cohen - Sutherland Line Clipping

- P3P4: P3(0101), P4(0100)

- Calculate the intersection P3';
  - Clip off P3P3';

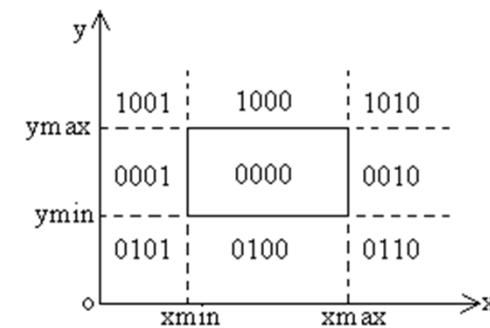
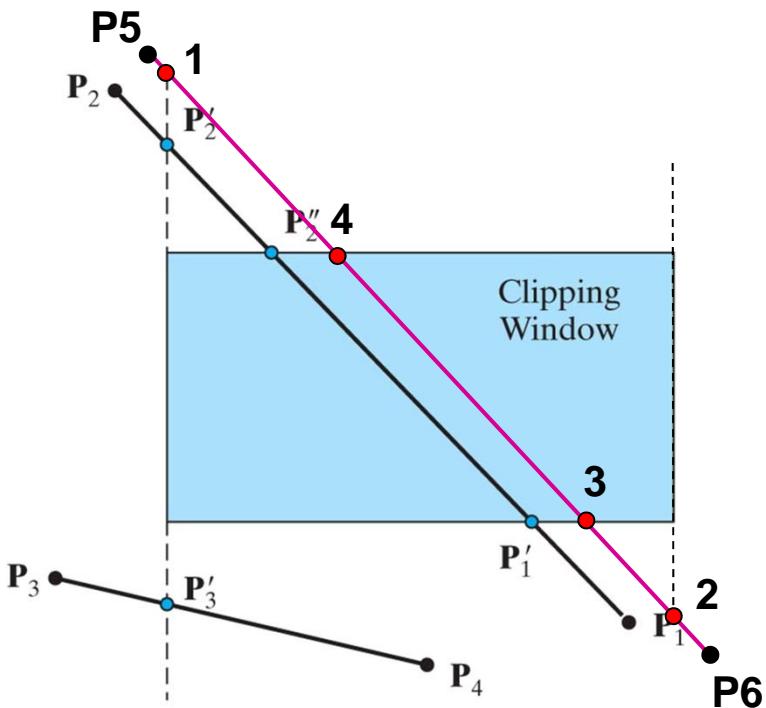
The remain is inside the right border line;

- Next, check the **bottom** border line;
  - The region codes show that all is below the clipping window.
  - Clip off P3'P4.

- P5P6: P5(1001), P6(0110)

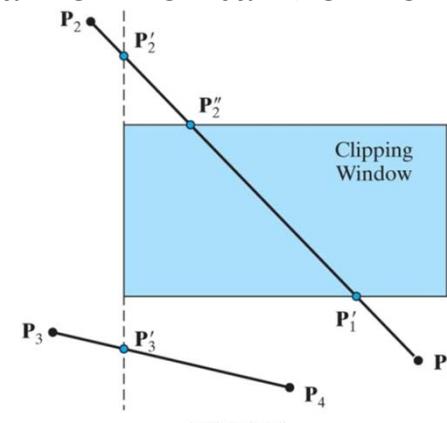
- Four intersection positions should be calculated in the order of L, R, B, T.

Variations of this basic approach have been developed



# Cohen - Sutherland Line Clipping

- To effectively calculate the intersections
  - Using the **slope equation** of the line
  - For line  $(x_0, y_0)$  and  $(x_{end}, y_{end})$
  - y coordinate of the intersection point with a vertical clipping border line:  
$$y = y_0 + m(x - x_0),$$
 where  $m = (y_{end} - y_0) / (x_{end} - x_0)$
  - x coordinate of the intersection point with a horizontal border line:  
$$x = x_0 + (y - y_0) / m.$$



# Cohen - Sutherland Line Clipping

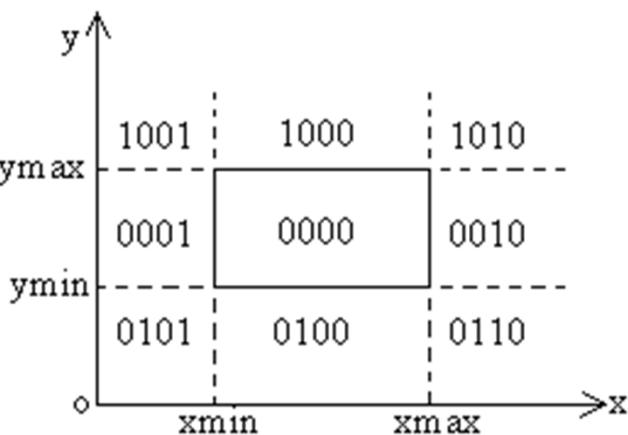
- To efficiently determine the region code

Comparing the coordinate values of an endpoint to the clipping boundaries and using **the sign bit of the result**, instead of using inequality testing.

- bit 1: left – the sign bit of  $(x - x_{w_{\min}})$ ;
- bit 2: right – the sign bit of  $(x_{w_{\max}} - x)$ ;
- bit 3: below – the sign bit of  $(y - y_{w_{\min}})$ ;
- bit 4: above – the sign bit of  $(y_{w_{\max}} - y)$ .

*/\* 2D Cohen-Sutherland Program Code \*/*

(See it in the textbook P279)



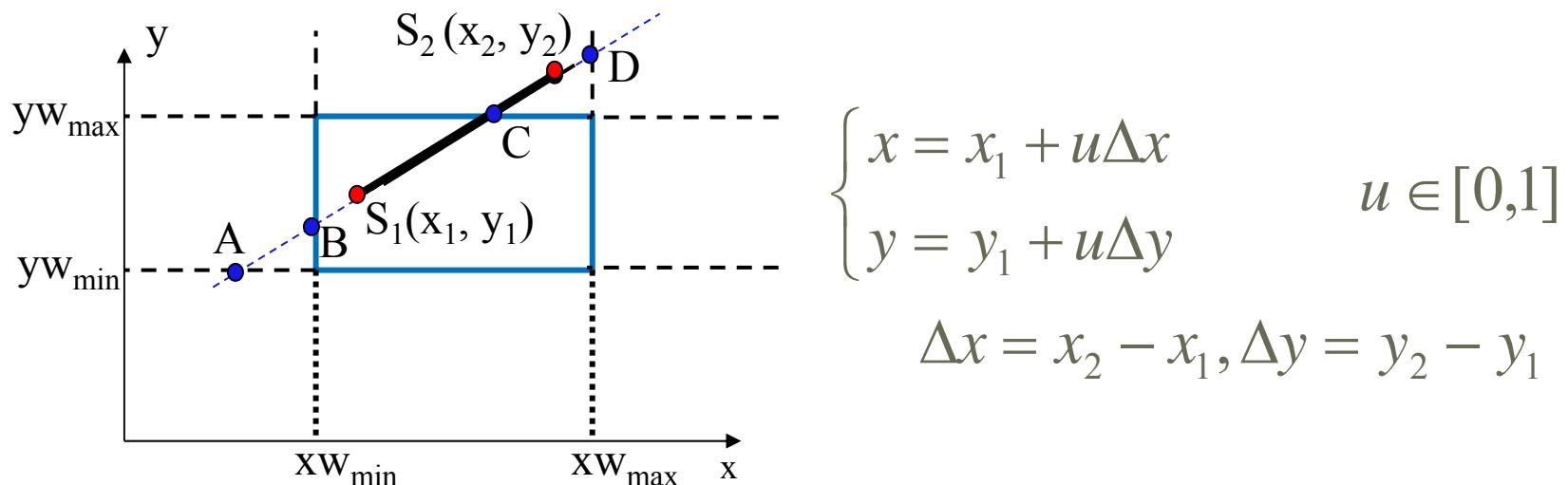
# Liang-Barsky Line Clipping

A faster line clipping algorithm: based on the parametric line equations.

Condition: Given a line segment  $S_1S_2$ .

A, B, C, D are the intersection points with the clipping window edges.

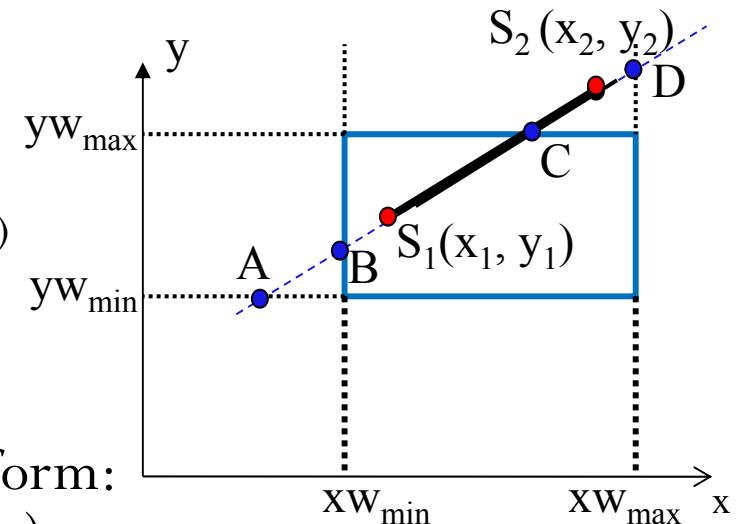
Key idea: to find the nearest point to  $S_2$  from **A**, **B** and  $S_1$  ( $S_1$  in the Fig); to find the nearest point to  $S_1$  from **C**, **D** and  $S_2$  ( $C$  in the Fig).  $S_1C$  is the visible part of  $S_1S_2$ .



# Liang-Barsky Line Clipping

- Line segment in parametric form:

$$\begin{cases} x = x_1 + u\Delta x & \Delta x = x_2 - x_1 \\ y = y_1 + u\Delta y & \Delta y = y_2 - y_1 \end{cases} \quad 0 \leq u \leq 1 \quad (8-16)$$



Point-clipping condition in parametric form:

$$\begin{cases} xw_{\min} \leq x_1 + u\Delta x \leq xw_{\max} \\ yw_{\min} \leq y_1 + u\Delta y \leq yw_{\max} \end{cases} \implies \begin{cases} u(-\Delta x) \leq x_1 - xw_{\min} \\ u\Delta x \leq xw_{\max} - x_1 \\ u(-\Delta y) \leq y_1 - yw_{\min} \\ u(\Delta y) \leq yw_{\max} - y_1 \end{cases}$$

$$up_k \leq q_k, \quad k = 1, 2, 3, 4$$

$$p_1 = -\Delta x$$

$$q_1 = x_1 - xw_{\min}$$

$$p_2 = \Delta x$$

$$q_2 = xw_{\max} - x_1$$

$$p_3 = -\Delta y$$

$$q_3 = y_1 - yw_{\min}$$

$$p_4 = \Delta y$$

$$q_4 = yw_{\max} - y_1$$

in another form

# Liang-Barsky Line Clipping

$$\begin{cases} x = x_1 + u\Delta x \\ y = y_1 + u\Delta y \end{cases} \quad u \in [0,1]$$

$$\Delta x = x_2 - x_1, \Delta y = y_2 - y_1$$

Decide the direction:

$$u(-\Delta x) \leq x_1 - xw_{\min}$$

$$up_k \leq q_k, \quad k = 1,2,3,4$$

$$u\Delta x \leq xw_{\max} - x_1$$

$$p_1 = -\Delta x$$

$$q_1 = x_1 - xw_{\min}$$

$$u(-\Delta y) \leq y_1 - yw_{\min}$$

$$\longrightarrow p_2 = \Delta x$$

$$q_2 = xw_{\max} - x_1$$

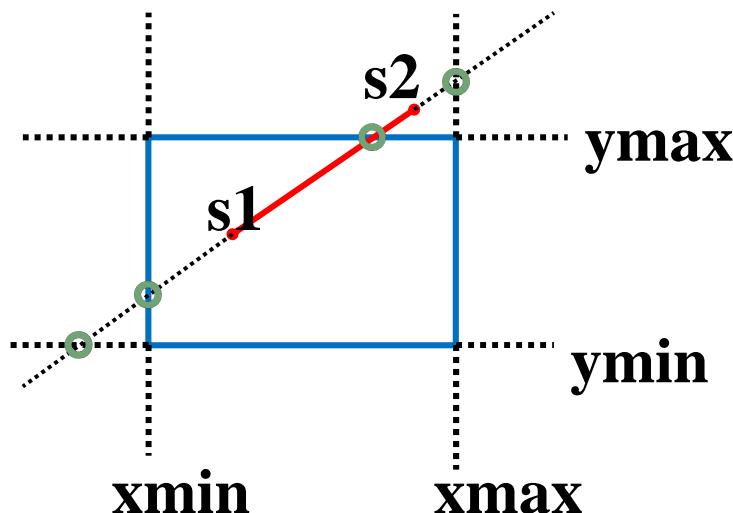
$$u(\Delta y) \leq yw_{\max} - y_1$$

$$p_3 = -\Delta y$$

$$q_3 = y_1 - yw_{\min}$$

$$p_4 = \Delta y$$

$$q_4 = yw_{\max} - y_1$$



Definition	Starting edge	Ending edge
$\Delta x \geq 0$	xmin	xmax
$\Delta x < 0$	xmax	xmin
$\Delta y \geq 0$	ymin	ymax
$\Delta y < 0$	ymax	ymin

- Here,  $\Delta x > 0$  and  $\Delta y > 0$   
so, xmin and ymin are the starting edge;

# Liang-Barsky Line Clipping

$$\begin{cases} x = x_1 + u\Delta x \\ y = y_1 + u\Delta y \end{cases} \quad u \in [0,1]$$

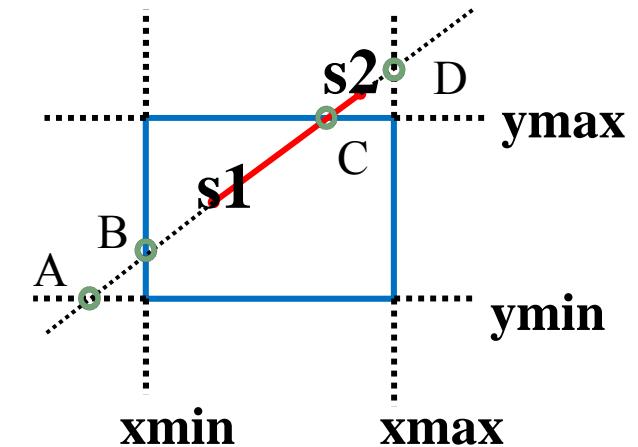
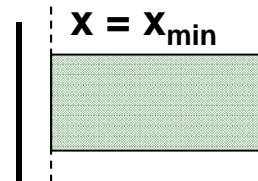
$$\Delta x = x_2 - x_1, \Delta y = y_2 - y_1$$

Start the  
test :

## Special cases

- If  $p_k=0$  ( $k=1,2$ ), the line segment is **parallel** to the clipping edge, then
  - If  $q_k < 0$  ( $k=1$  or  $2$ ), the line is completely outside of the clipping region, so discard it;
  - If  $q_k > 0$  ( $k=1$  or  $2$ ), line segment is possible visible, get the intersections with the window edges.

(Similarly,  $p_k=0$  ( $k=3,4$ ))



$$up_k \leq q_k, \quad k = 1, 2, 3, 4$$

$$p_1 = -\Delta x \quad q_1 = x_1 - xw_{\min}$$

$$p_2 = \Delta x \quad q_2 = xw_{\max} - x_1$$

$$p_3 = -\Delta y \quad q_3 = y_1 - yw_{\min}$$

$$p_4 = \Delta y \quad q_4 = yw_{\max} - y_1$$

# Liang-Barsky Line Clipping

Start the  
test (cont.):

general  
cases

- The parameters of the intersections of  $S_1S_2$  with the **starting** edge are referred as  $u1'$ ,  $u1''$

$u1 = \max\{u1', u1'', 0\}$  – the nearest clipping point to  $S_2$

- The parameters of the intersections of  $S_1S_2$  with two **ending** edges are referred as  $u2'$ ,  $u2''$

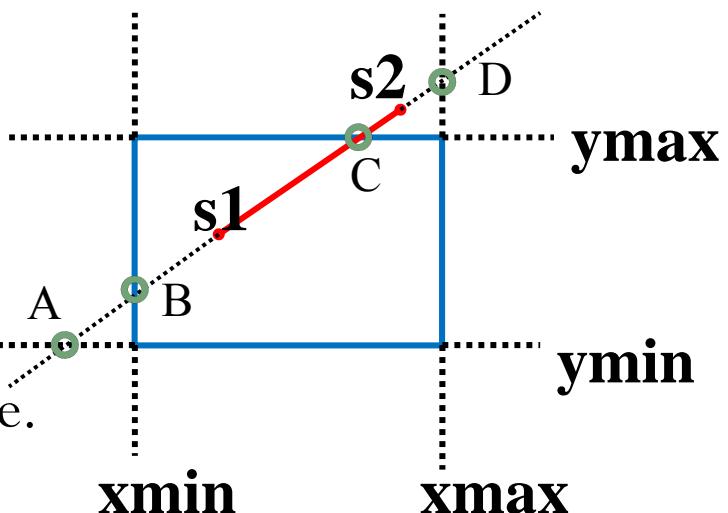
$u2 = \min\{u2', u2'', 1\}$  – the nearest clipping point to  $S_1$

## Conclusion:

1. If  $u1 < u2$  ,  
 $x = x1 + \Delta x * u$   
 $y = y1 + \Delta y * u$   
 $(u1 < u < u2)$

they define the visible part.

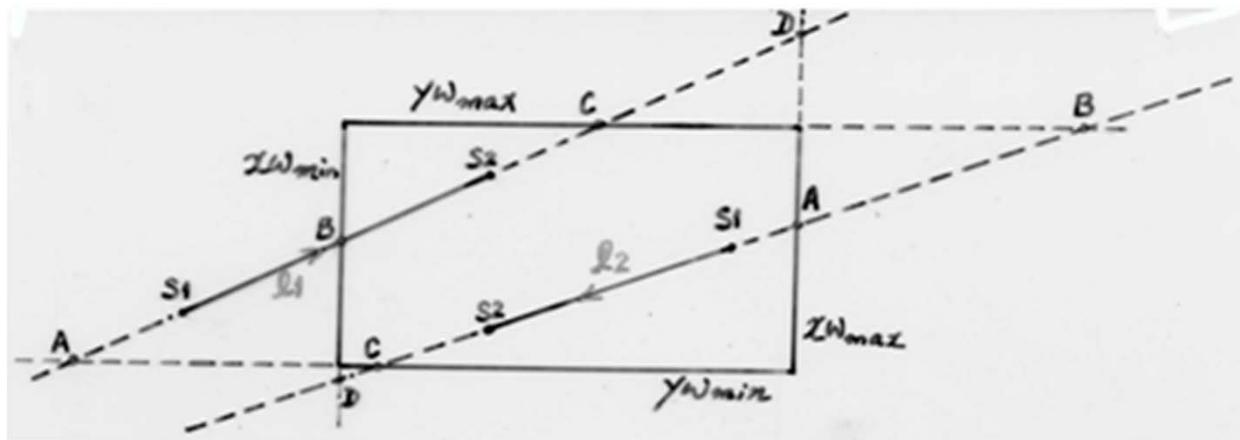
2. If  $u1 > u2$  , the line is invisible.



# 2D Line Clipping

- Liang-Barsky Line Clipping example 1

	Starting edge	Ending edge
$\Delta x \geq 0$	xmin	xmax
$\Delta x < 0$	xmax	xmin
$\Delta y \geq 0$	ymin	ymax
$\Delta y < 0$	ymax	ymin



11:  $\Delta x > 0$  and  $\Delta y > 0$ , starting edge:  $x_{w\min}$  and  $y_{w\min}$  (A, B);

12:  $\Delta x < 0$  and  $\Delta y < 0$ , starting edge:  $x_{w\max}$  and  $y_{w\max}$  (A, B);

1.  $u_1'$ ,  $u_1''$  refer to intersections of  $S_1S_2$  with two starting edges

$u_1 = \max \{u_1', u_1'', 0\}$  – the nearest clipping point to  $S_2$

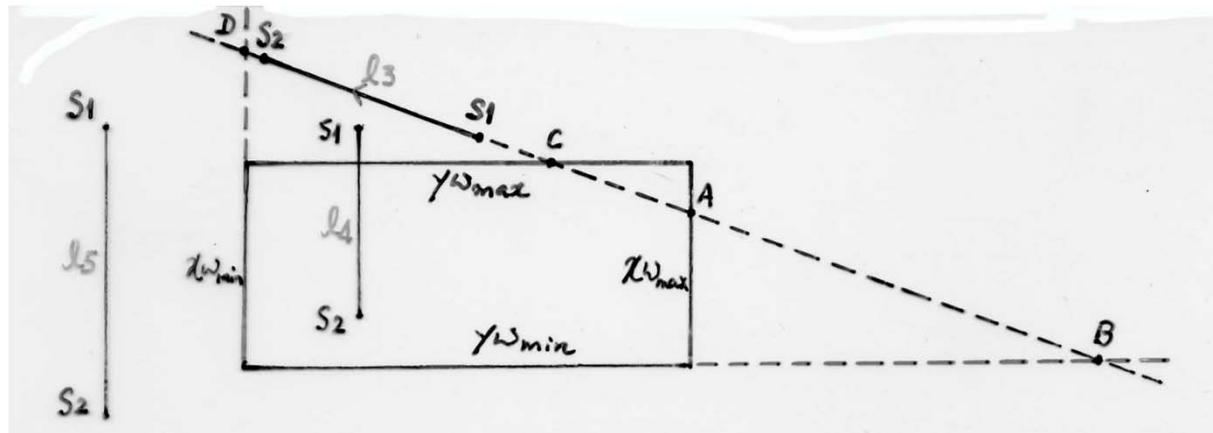
2.  $u_2'$ ,  $u_2''$  refer to intersections of  $S_1S_2$  with two ending edges

$u_2 = \min \{u_2', u_2'', 1\}$  – the nearest clipping point to  $S_1$

# 2D Line Clipping

- Liang-Barsky Line Clipping example 2

	Starting edge	Ending edge
$\Delta x \geq 0$	xmin	xmax
$\Delta x < 0$	xmax	xmin
$\Delta y \geq 0$	ymin	ymax
$\Delta y < 0$	ymax	ymin



$$u_p \leq q_k, k = 1, 2, 3, 4$$

$$p_1 = -\Delta x, \quad q_1 = x_1 - xw_{\min}$$

$$p_2 = \Delta x, \quad q_2 = xw_{\max} - x_1$$

$$p_3 = -\Delta y, \quad q_3 = y_1 - yw_{\min}$$

$$p_4 = \Delta y, \quad q_4 = yw_{\max} - y_1$$

$I_3$ :  $\Delta x < 0, \Delta y > 0$ , starting edges are  $xw_{\max}$  and  $yw_{\min}$  (A, B)

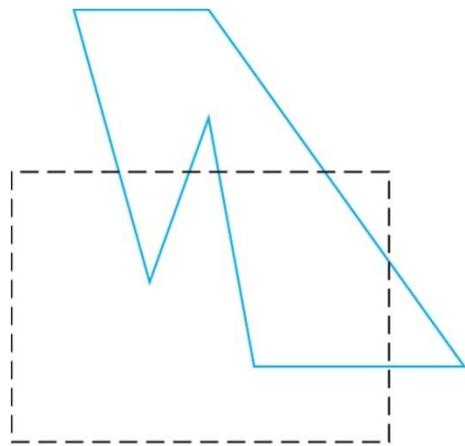
$$u_1 = u_{s1} = 0, u_2 = u_C < 0$$

$\because u_1 > u_2, \therefore S_1 S_2$  invisible.

$I_4$ :  $p_1 = p_2 = 0$  but  $q_1 > 0, q_2 > 0 \therefore$  partly visible possibly;

$I_5$  :  $p_1 = p_2 = 0, q_1 < 0$ , invisible.

# Polygon Fill-Area Clipping



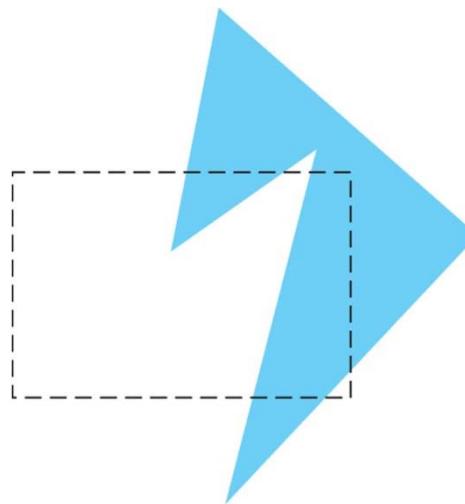
Before Clipping

(a)



After Clipping

(b)



Before Clipping

(a)



After Clipping

(b)

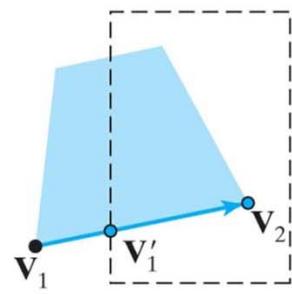
**Figure 8-19** A line-clipping algorithm applied to the line segments of the polygon boundary in (a) generates the unconnected set of lines in (b).

**Figure 8-20** Display of a correctly clipped polygon.

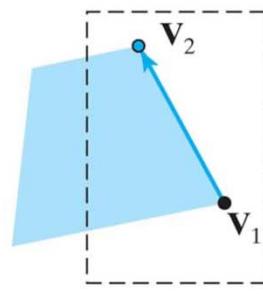
# Sutherland-Hodgman Polygon Clipping

Intersections of the polygon edge  $V_1V_2$  with the window edge L:

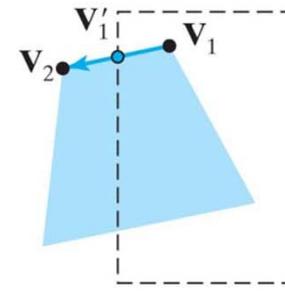
1. If  $V_1, V_2$  lies to the same side of L, their visibility is same and without intersection.
2. If  $V_1, V_2$  lies to the different side of L, their visibility is different and with intersection.



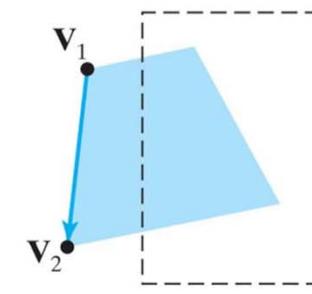
(1)  
out → in  
Output:  $\mathbf{V}'_1, \mathbf{V}_2$



(2)  
in → in  
Output:  $\mathbf{V}_2$



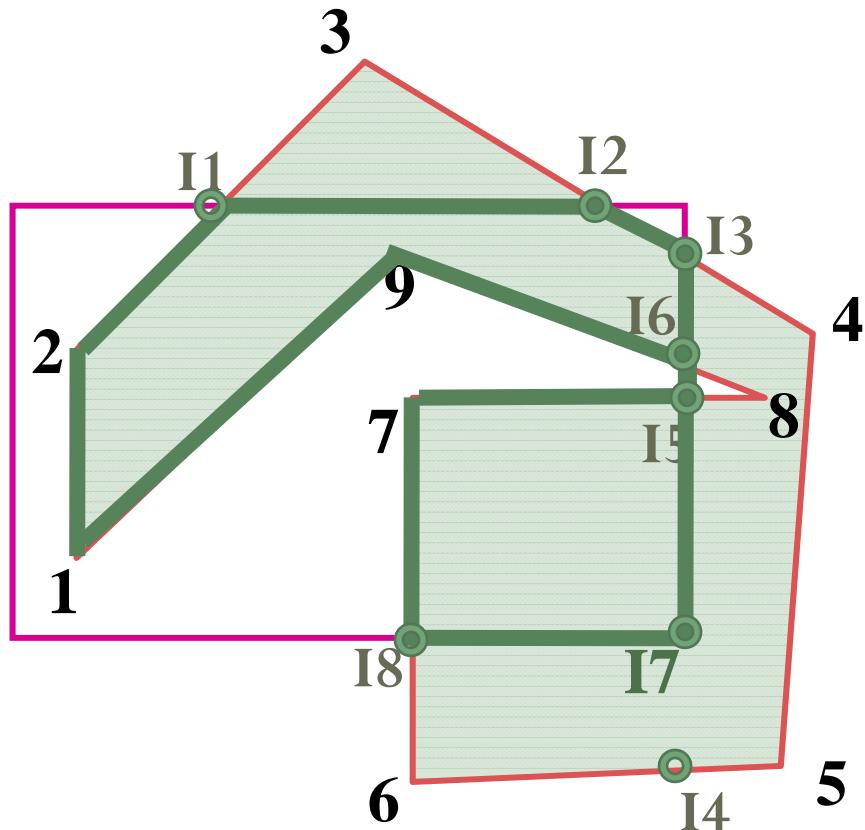
(3)  
in → out  
Output:  $\mathbf{V}'_1$



(4)  
out → out  
Output: none

# Sutherland-Hodgman Polygon Clipping

Example1: A polygon 1,2,3,4,5,6,7,8,9



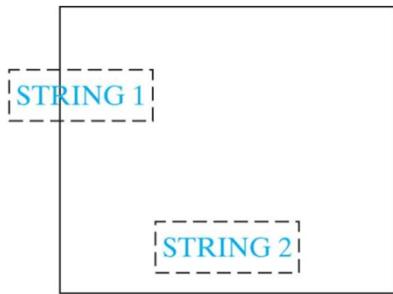
After clipping by xmin  
1,2,3,4,5,6,7,8,9

After clipping by ymax  
1,2,I1,I2,4,5,6,7,8,9

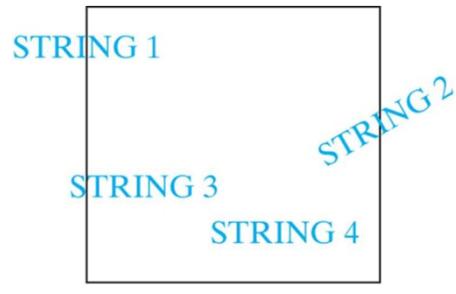
After clipping by xmax  
1,2,I1,I2,I3,I4,6,7,I5,I6,9

After clipping by ymin:  
1,2, I1,I2,I3,I7,I8,7,I5,I6,9

# Text Clipping



Before Clipping



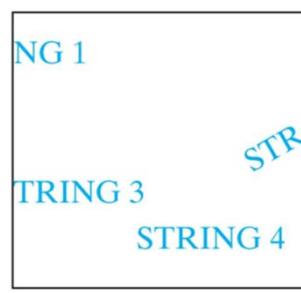
Before Clipping



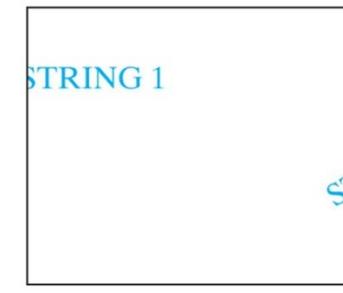
Before Clipping



After Clipping



After Clipping



After Clipping

Copyright ©2011 Pearson Education, publishing as Prentice Hall

Copyright ©2011 Pearson Education, publishing as Prentice Hall

Copyright ©2011 Pearson Education, publishing as Prentice Hall

Simplest method: all-or-none string clipping

Alternative one: all-or-none character clipping

Most accurate one: clipping on the components of individual characters

# Summary

- 2D clipping algorithm
  - 2D point clipping
  - 2D line clipping
    - Cohen-Sutherland
    - Liang-Barsky
  - Polygon clipping
  - Text clipping

Computer Graphics

Chapter 9  
Three-Dimensional Geometric  
Transformations

# Outline

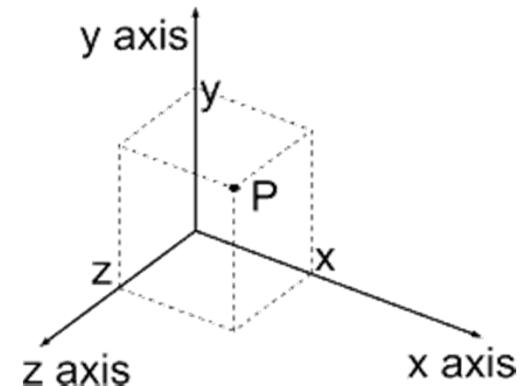
- 3D Translation
- 3D Rotation
- 3D Scaling
- Transformations between 3D Coordinate Systems
- OpenGL Geometric Transformation Functions
- OpenGL 3D Geometric Transformation Programming Examples

# 3D Transformation

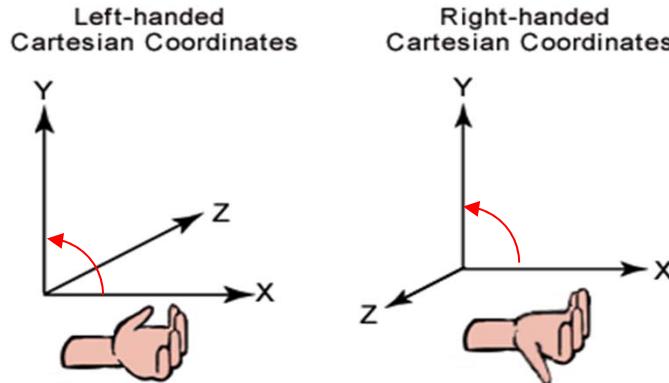
- Same as 2D.
- Add z-axis and z-coordinate.
- Use 4X4 homogenous matrix.

(x, y, z, w)

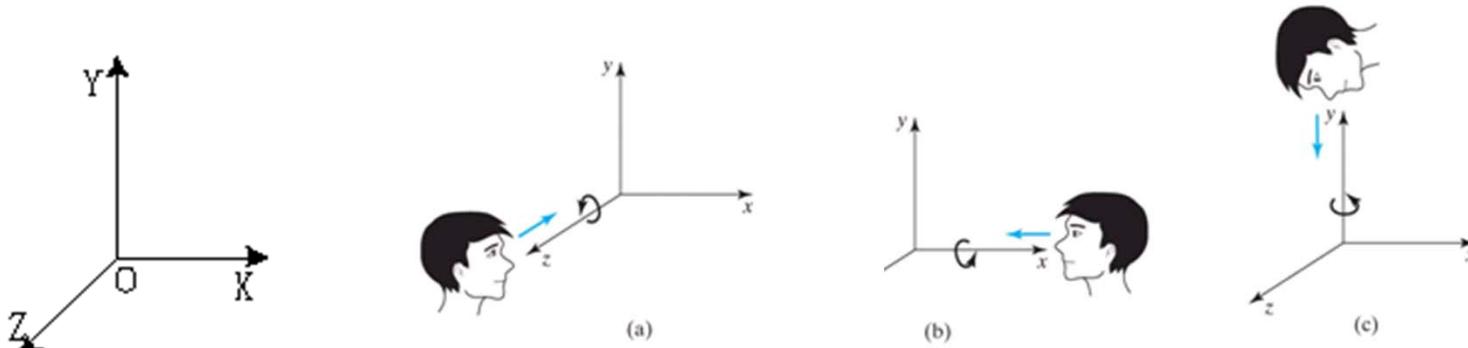
$$\begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix} = \begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$



# Right-hand coordinate system



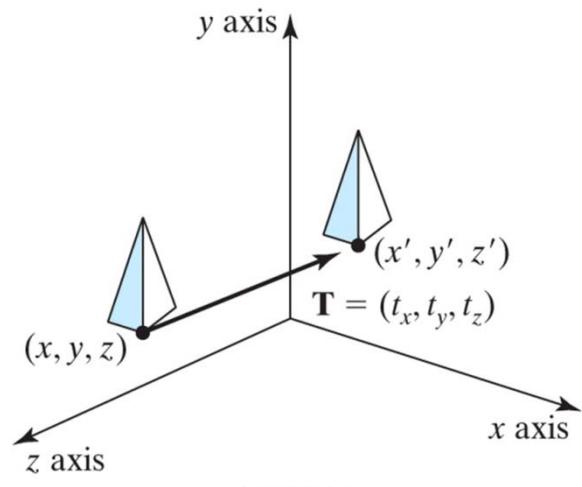
- OpenGL: **right-hand**
  - The positive x and y axes point right and up, and the z axis points to the viewer.
  - Positive rotation is **counterclockwise** about the axis of rotation when looking along the positive half of the axis toward the origin.



# 3D Translation

- A position  $P=(x, y, z)$  in 3D space is translated to a location  $P'=(x', y', z')$  by adding translation distances  $t_x$ ,  $t_y$ , and  $t_z$ :

$$x' = x + t_x \quad y' = y + t_y \quad z' = z + t_z \quad (9-1)$$



**FIGURE 9-2** Shifting the position of a three-dimensional object using translation vector  $T$ .

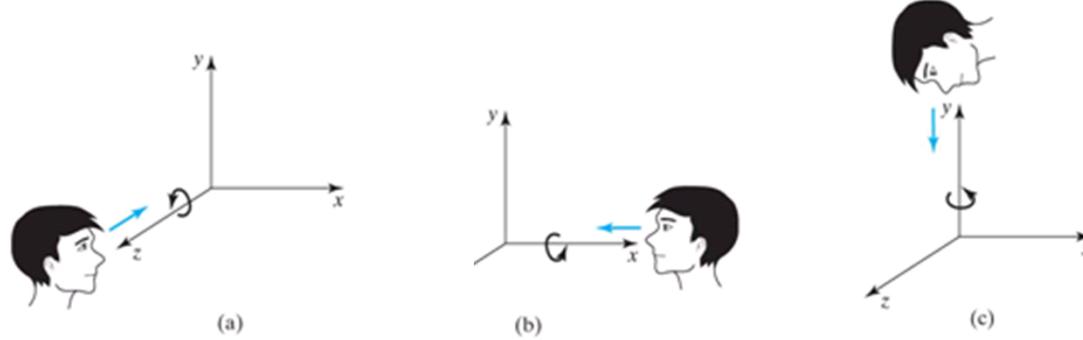
By matrix form:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \quad (9-2)$$

or

$$\mathbf{P}' = \mathbf{T} \cdot \mathbf{P} \quad (9-3)$$

# 3D Rotation



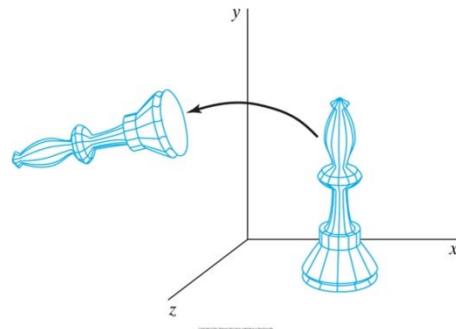
Copyright ©2011 Pearson Education, publishing as Prentice Hall

**Positive rotations:** **counterclockwise** when looking along the positive half of the axis toward the origin

- Coordinate-Axes Rotations
  - X-axis, Y-axis or Z-axis rotation
  - Rotation about an axis that is in parallel to one of the coordinate axes
- General 3D Rotations
  - Rotation about an arbitrary axis

# 3D Coordinate Axis Rotation

- Rotation of an object about the z axis



$$x' = x \cos\theta - y \sin\theta$$

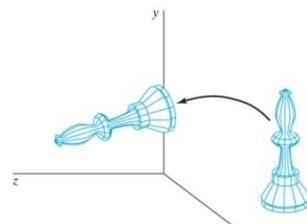
$$y' = x \sin\theta + y \cos\theta$$

$$z' = z$$

$$T = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$Rx = \begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

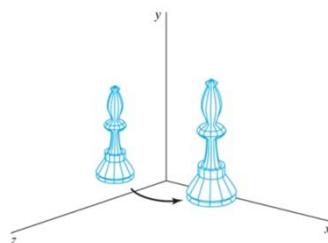
- Rotation of an object about the x axis



$z \rightarrow x; x \rightarrow y; y \rightarrow z.$

$$Rz = \begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

- Rotation of an object about the y axis

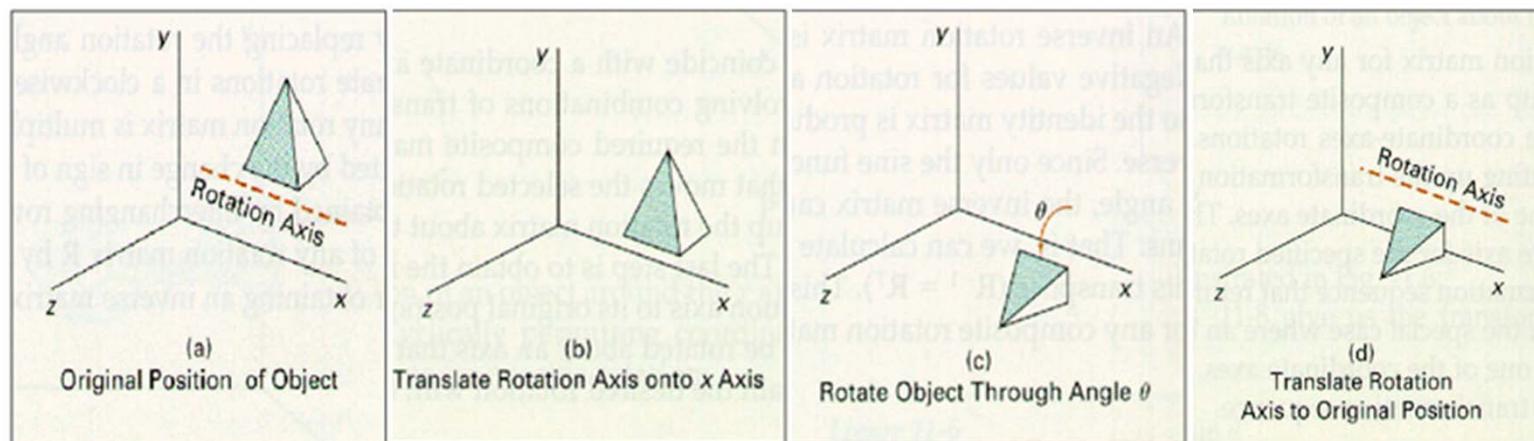


$x \rightarrow y; y \rightarrow z; z \rightarrow x.$

$$Ry = \begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

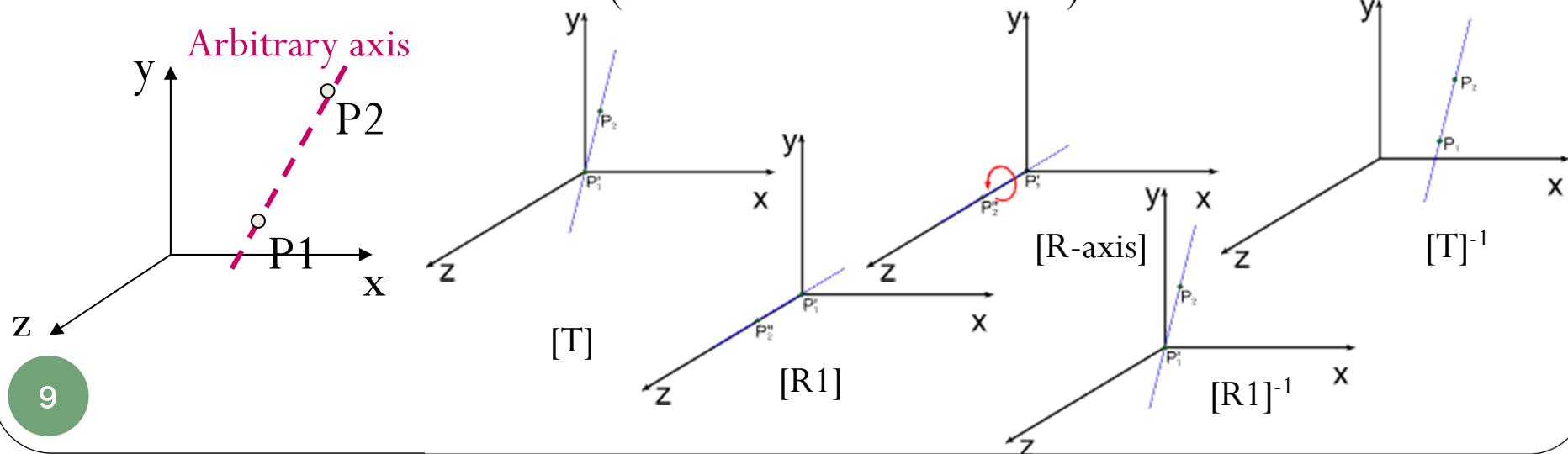
# 3D Rotations Parallel to Axes

- Rotation an axis that is parallel to one of the coordinate axes
  - Translate the object so that the rotation axis coincides with the parallel coordinate axis
  - Perform the specified rotation about that axis
  - Translate the object so that the rotation axis is moved back to its original position



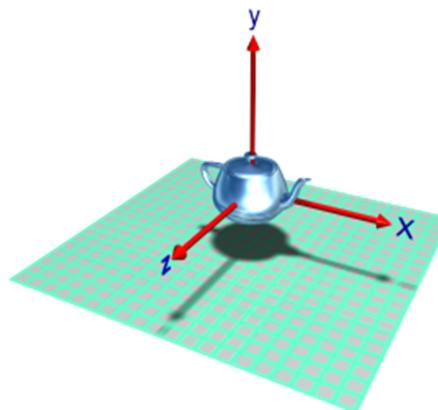
# 3D Rotations about Arbitrary Axis

- Rotate about the arbitrary axis through P1 and P2:
  1. Translate P1 to origin.
  2. Rotate so that the rotation axis is aligned with one of the principle coordinate axes.
  3. Perform the desired rotation about coordinate axis.
  4. Rotate axis back (inverse rotation of 2).
  5. Translate axis back (inverse translation of 1).

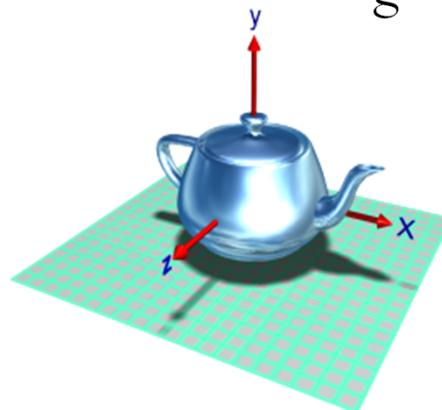


# 3D Scaling

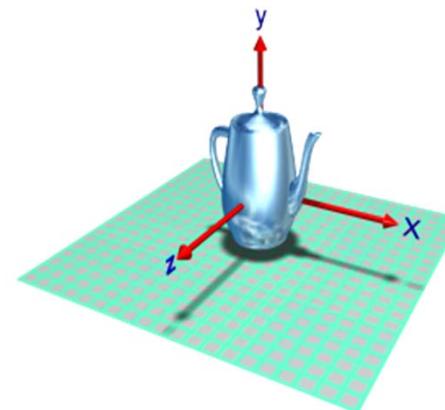
- Scale objects relative to the coordinate origin  $(0, 0, 0)$ 
  - All vectors are scaled from the origin



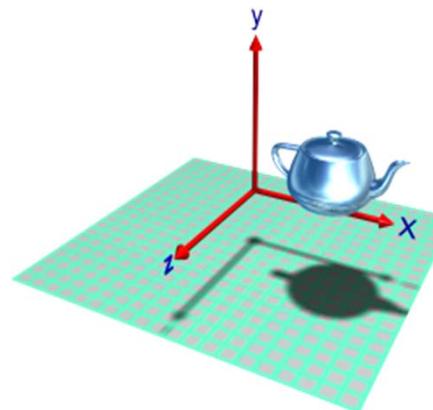
Original



scale all axes

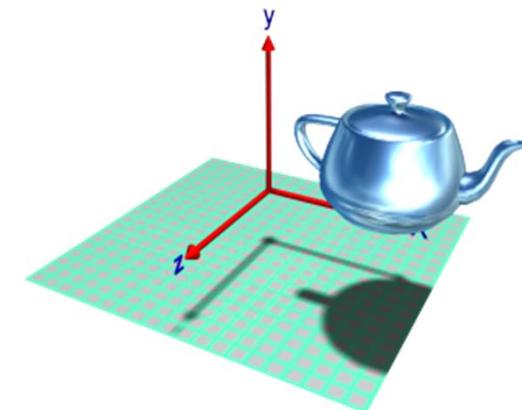


scale Y axis

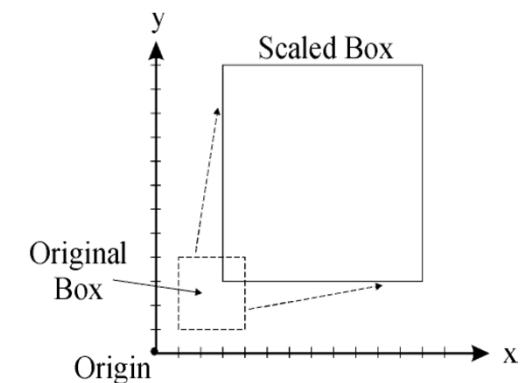


10

offset from origin



distance from origin also scales

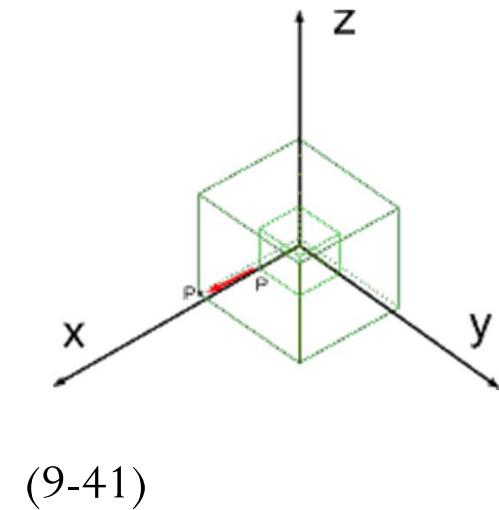


# 3D Scaling

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} s_x x \\ s_y y \\ s_z z \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & s_z \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} \Rightarrow P' = S \cdot P$$

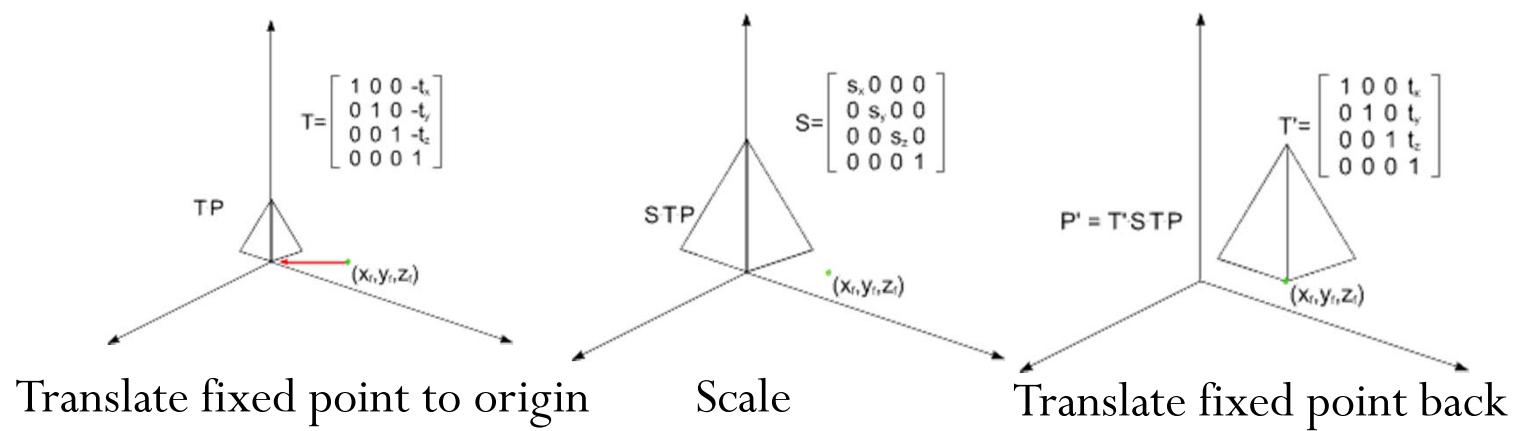
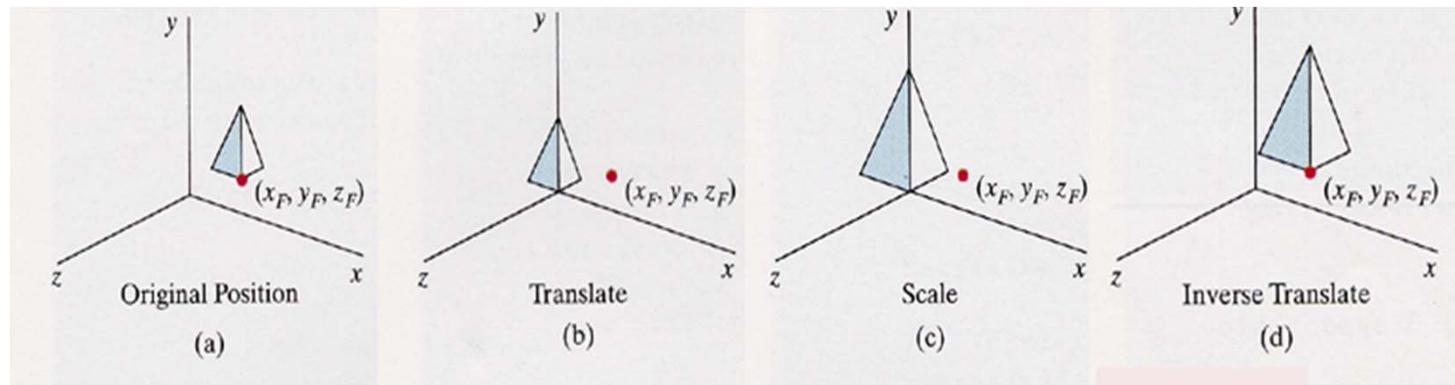
or in 3D homogeneous coordinates

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} s_x x \\ s_y y \\ s_z z \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$



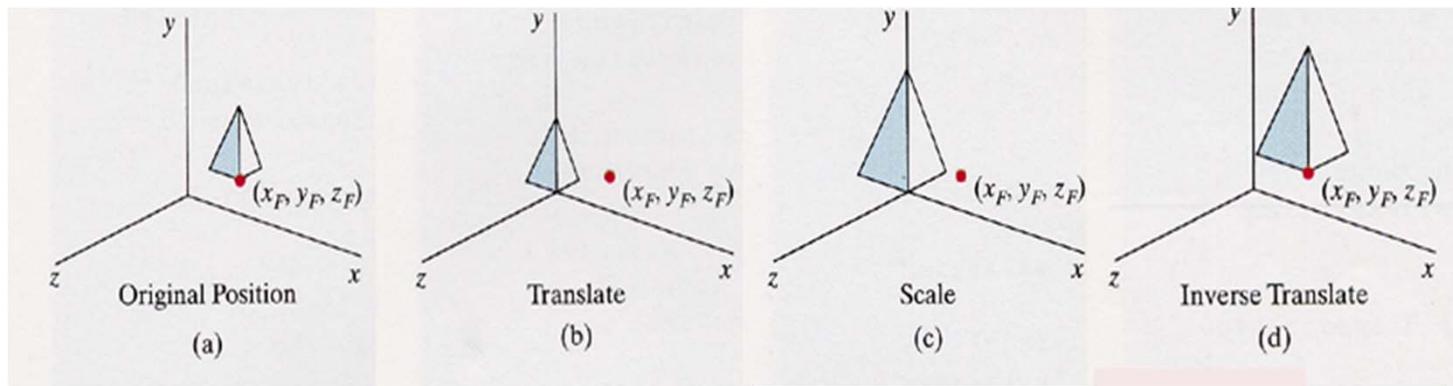
# 3D Scaling

- Scaling objects relative to a selected fixed point  $(x_f, y_f, z_f)$



# 3D Scaling

- Scaling objects relative to a selected fixed point  $(x_f, y_f, z_f)$   
(cont.)



$$T(x_f, y_f, z_f) \cdot S(s_x, s_y, s_z) \cdot T(-x_f, -y_f, -z_f) = \begin{bmatrix} 1 & 0 & 0 & x_f \\ 0 & 1 & 0 & y_f \\ 0 & 0 & 1 & z_f \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -x_f \\ 0 & 1 & 0 & -y_f \\ 0 & 0 & 1 & -z_f \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$T(x_f, y_f, z_f) \cdot S(s_x, s_y, s_z) \cdot T(-x_f, -y_f, -z_f) = \begin{bmatrix} s_x & 0 & 0 & (1-s_x)x_f \\ 0 & s_y & 0 & (1-s_y)y_f \\ 0 & 0 & s_z & (1-s_z)z_f \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

# Matrix Composition

- Transformations can be combined by matrix multiplication

$$\begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & tx \\ 0 & 1 & 0 & ty \\ 0 & 0 & 1 & tz \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos \theta & -\sin \theta & 0 & 0 \\ \sin \theta & \cos \theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} sx & 0 & 0 & 0 \\ 0 & sy & 0 & 0 \\ 0 & 0 & sz & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$
$$\mathbf{p}' = T(t_x, t_y) R(Q) S(s_x, s_y) \mathbf{p}$$

$$\boxed{\begin{aligned} \mathbf{p}' &= (T * (R * (S * \mathbf{p}))) \\ \mathbf{p}' &= (T * R * S) * \mathbf{p} \end{aligned}}$$

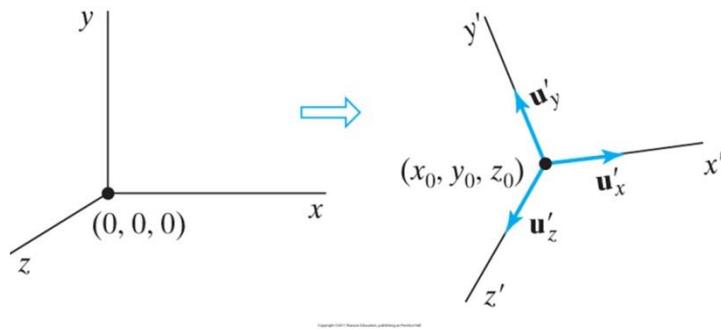
- Order of transformations**

- Matrix multiplication is not commutative

$$\mathbf{p}' = T * R * S * \mathbf{p}$$

$\xleftarrow{\hspace{2cm}} \xrightarrow{\hspace{2cm}}$   
“Global”      “Local”

# Transformations Between 3D Coordinate Systems



**FIGURE 9-21** A new  $x'y'z'$  coordinate system defined within an  $xyz$  system. A scene description is transferred to the new coordinate reference using a transformation sequence that superimposes the  $x'y'z'$  frame on the  $xyz$  axes.

- To transfer the  $xyz$  coordinate descriptions  $\rightarrow x'y'z'$  coordinate system
  - **Translation:** bring the  $x'y'z'$  coordinate origin to the position of the  $xyz$  origin.
  - **Transform  $x'y'z'$  onto the corresponding axes  $xyz$ :** the coordinate-axis rotation matrix formed by the unit axis vectors.

$$R = \begin{bmatrix} u'_{x1} & u'_{x2} & u'_{x3} & 0 \\ u'_{y1} & u'_{y2} & u'_{y3} & 0 \\ u'_{z1} & u'_{z2} & u'_{z3} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \text{(P313)}$$

which transforms unit vector  $\mathbf{u}'_x$ ,  
 $\mathbf{u}'_y$ ,  $\mathbf{u}'_z$  onto the  $x$ ,  $y$  and  $z$  axes.

$$\mathbf{M}_{xyz, x'y'z'} = \mathbf{R} \bullet \mathbf{T}(-x_0, -y_0, -z_0)$$

- If different scales are used in the two coordinate systems, the scaling transformation may also be needed.

# OpenGL Geometric Transformation Functions

- Be careful of manipulating the matrix in OpenGL

- OpenGL uses **4X4** matrix for transformation.
- The 16 elements are stored as 1D in *column-major order*

$$\begin{pmatrix} a_0 & a_4 & a_8 & a_{12} \\ a_1 & a_5 & a_9 & a_{13} \\ a_2 & a_6 & a_{10} & a_{14} \\ a_3 & a_7 & a_{11} & a_{15} \end{pmatrix} \text{ OpenGL transform matrix}$$

- C and C++ store matrices in *row-major order*
- If you declare a matrix to be used in OpenGL as  
**GLfloat M[4][4];** to access the element in row i and column j, you need to refer to it by **M[j][i];** or, as  
**GLfloat M[16];** and then you need to convert it to conventional *row-major order.*

# OpenGL Transformations

- All the transformations done by OpenGL can be described as a **multiplication of two or more matrices**.
  - The mathematics behind these transformations are greatly simplified by the mathematical notation of the matrix.
  - Each of the transformations can be achieved by multiplying a **matrix** that contains the vertices, by a **matrix** that describes the transformation.

# OpenGL Geometric Transformation Functions

- Basic OpenGL geometric transformations on the matrix:

**glTranslate\*** (tx, ty, tz);

[ glTranslatef (25.0, -10.0, 10.0);

- Post-multiplies the current matrix by a matrix that moves the object by the given x-, y-, and z-values

**glScale\*** (sx, sy, sz);

[ glScalef (2.0, -3.0, 1.0); ]

- Post-multiplies the current matrix by a matrix that scales an object about the origin.  
None of sx, sy or sz is zero.

**glRotate\*** (theta, vx, vy, vz);

[ glRotatef (90.0, 0.0, 0.0, 1.0); ]

- Post-multiplies the current matrix by a matrix that rotates the object in a  
counterclockwise direction. vector v=(vx, vy, vz) defines the orientation for the  
rotation axis that passes through the coordinate origin. ( the rotation center is (0, 0, 0) )

# OpenGL: Order in Matrix Multiplication

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();      //Set current matrix to the identity.
glMultMatrixf(elemsM2); //Post-multiply identity by matrix M2.
glMultMatrixf(elemsM1); //Post-multiply M2 by matrix M1.
glBegin(GL_POINTS)
    glVertex3f(vertex);
glEnd();
```

Modelview matrix successively contains:

$I(\text{identity}), M_2, M_2 \cdot M_1$

The concatenated matrix is:

$$M = M_2 \cdot M_1$$

The transformed vertex is:

$$M_2 \cdot (M_1 \cdot \text{vertex})$$

In OpenGL, a transformation sequence is applied in reverse order of which it is specified.

# OpenGL: Order in Matrix Multiplication

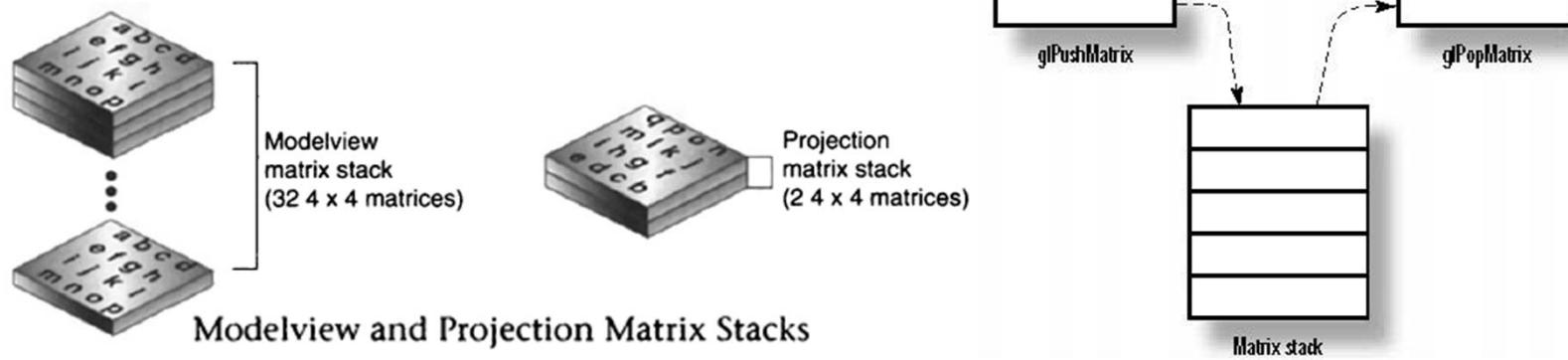
- Example

```
// rotate object 30 degrees around X-axis  
glRotatef(30.0, 1.0, 0.0, 0.0);  
  
// move object to (x, y, z)  
glTranslatef(x, y, z);  
  
drawObject();
```

The object will be **translated** first then **rotated**.

# Independent Models: Matrix Stacks

- How OpenGL implement the independent models?
- OpenGL maintains a **stack** of matrices.
  - Each type of the matrix modes has a matrix stack (modelview, projection, texture, and color)
  - Initial value is identity matrix
  - The top matrix on the stack at any time: the current matrix
  - New matrix transformation function is applied to the current matrix
  - To use push or pop functions to modify it.



# Functions About Matrix Stack Operations

- Find the maximum allowable number of matrices in stack

```
glGetIntegerv (GL_MAX_MODELVIEW_STACK_DEPTH,  
stackSize);
```

```
glGetIntegerv (GL_MAX_PROJECTION_STACK_DEPTH,  
stackSize);
```

- Find out how many matrices are currently in the stack

```
glGetIntegerv (GL_MODELVIEW_STACK_DEPTH, numMats);
```

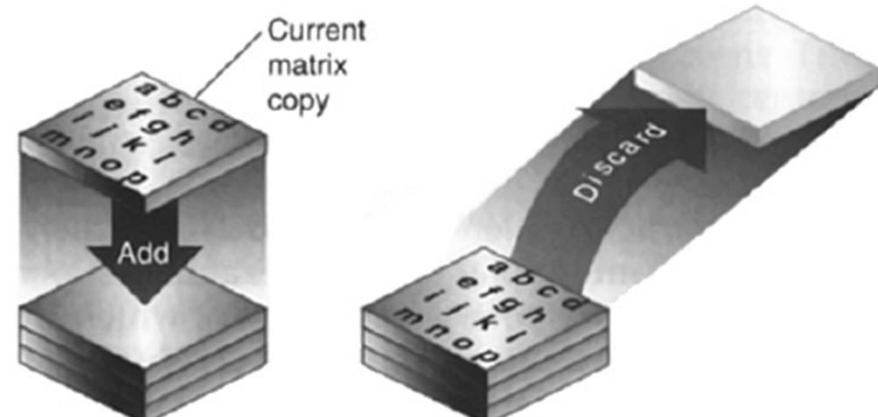
# Functions About Matrix Stack Operations

## **glPushMatrix ()**

- Push the current matrix down one level and copy the current matrix

## **glPopMatrix ()**

- Pop the top matrix off the stack



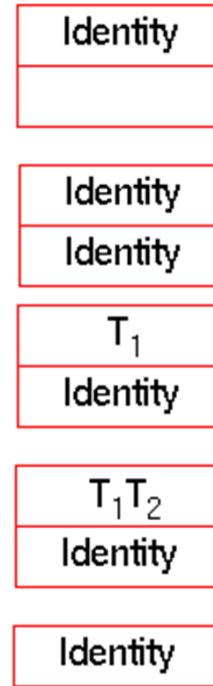
Pushing and Popping the Matrix Stack

- Matrix stack is very useful for creating hierarchical model (body, car ...).
  - Save the current position (modelview)
  - Load a previous position or new ones

# Matrix Stack Operations

- Example

```
glMatrixMode (GL_MODELVIEW);  
glPushMatrix ();  
    glTranslatef ( 0.0, 0.0, -8.0 );  
    glTranslatef ( 1.0, 0.0, 0.0 );  
    DrawObj ();  
glPopMatrix ();
```



modelview matrix stack

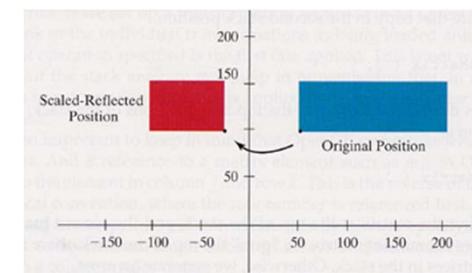
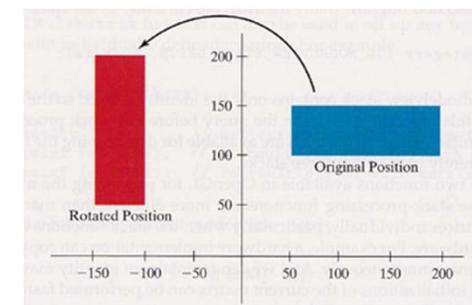
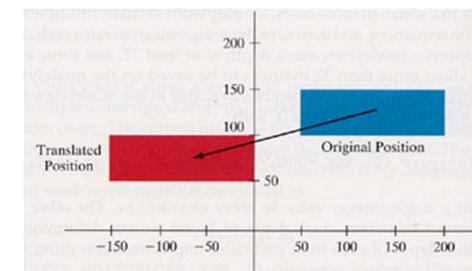
# OpenGL Geometric Trans. Programming Examples

```
glMatrixMode(GL_MODELVIEW); //Identity matrix  
glColor3f(0.0, 0.0, 1.0);      // Set current color to blue  
glRecti(50, 100, 200, 150);    // Display blue rectangle.
```

```
glColor3f(1.0, 0.0, 0.0);      // Red  
glTranslatef(-200.0, -50.0, 0.0); // Set translation parameters.  
glRecti(50, 100, 200, 150);    // Display red, translated rectangle.
```

```
glLoadIdentity();           // Reset current matrix to identity.  
glRotatef(90.0, 0.0, 0.0, 1.0); // Set 90-deg, rotation about z axis.  
glRecti(50, 100, 200, 150);   // Display red, rotated rectangle.
```

```
glLoadIdentity();           // Reset current matrix to identity.  
glScalef(-0.5, 1.0, 1.0);   // Set scale-reflection parameters.  
glRecti(50, 100, 200, 150); // Display red, transformed rectangle.
```



# OpenGL Geometric Trans. Programming Examples

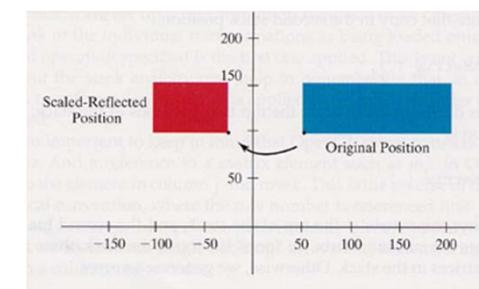
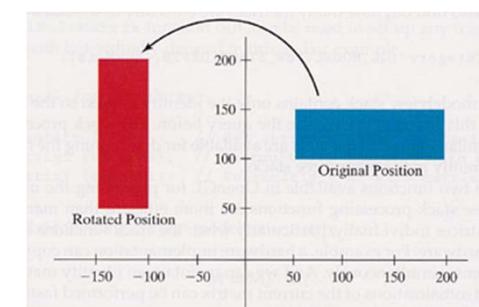
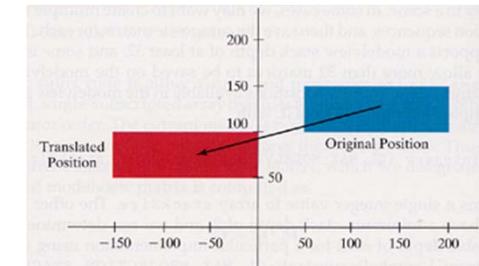
- More efficient way: `glPushMatrix`/`glPopMatrix`

```
glMatrixMode(GL_MODELVIEW);
glColor3f(0.0, 0.0, 1.0);      // Set current color to blue.
glRecti(50, 100, 200, 150);    // Display blue rectangle.
```

```
glPushMatrix();               // Make copy of identity (top) matrix.
glColor3f(1.0, 0.0, 0.0);     // Set current color to red.
glTranslatef(-200.0, -50.0, 0.0); // Set translation parameters.
glRecti(50, 100, 200, 150);    // Display red, translated rectangle.
glPopMatrix();                // Throw away the translation matrix.
```

```
glPushMatrix();               // Make copy of identity (top) matrix.
glRotatef(90.0, 0.0, 0.0, 1.0); // Set 90-deg, rotation about z axis.
glRecti(50, 100, 200, 150);    // Display red, rotated rectangle.
glPopMatrix();                // Throw away the rotation matrix.
```

```
glScalef(-0.5, 1.0, 1.0);    // Set scale-reflection parameters.
glRecti(50, 100, 200, 150);    // Display red, transformed rectangle.
```

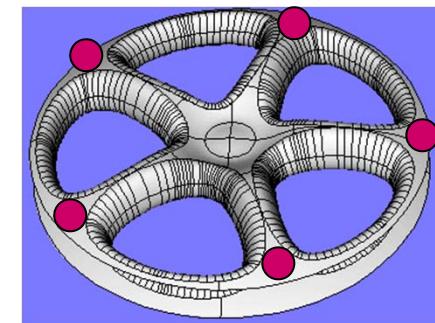


# Example

The wheels and bolt axes are coincident with z-axis; the bolts are evenly spaced every 72 degrees, 3 units from the center of the wheel.

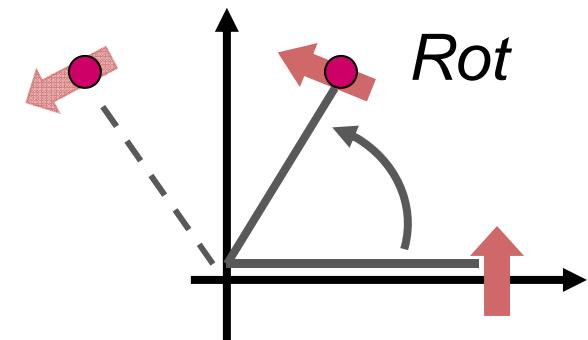
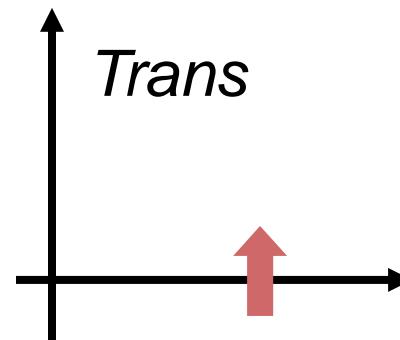
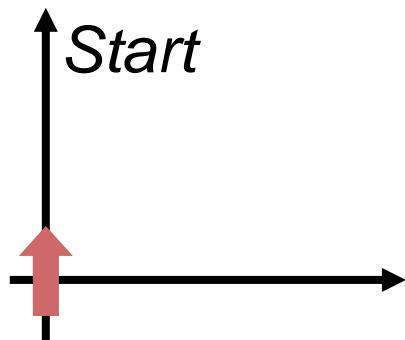
- Drawing a car's wheels with bolts

```
draw_wheel();
for (j=0; j<5; j++) {
    glPushMatrix();
        glRotatef(72.0*j, 0.0, 0.0, 1.0);
        glTranslatef (3.0, 0.0, 0.0);
        draw_bolt ();
    glPopMatrix();
}
```



R  
RT  
RTv

Global – Bottom Up



# Summary

- Basic 3D geometric transformations
  - Translation
  - Rotation
  - Scaling
  - Combination of these transformations
- OpenGL 3D geometric transformation functions
  - GL\_MODELVIEW matrix
  - Order in multiple matrix multiplication
- Matrix stack
  - glPushMatrix ()
  - glPopMatrix ()

Computer Graphics

Chapter 10  
Three-Dimensional Viewing

# Chapter 10

## Three-Dimensional Viewing

---

### Part I.

Overview of 3D Viewing Concept

3D Viewing Pipeline vs. OpenGL Pipeline

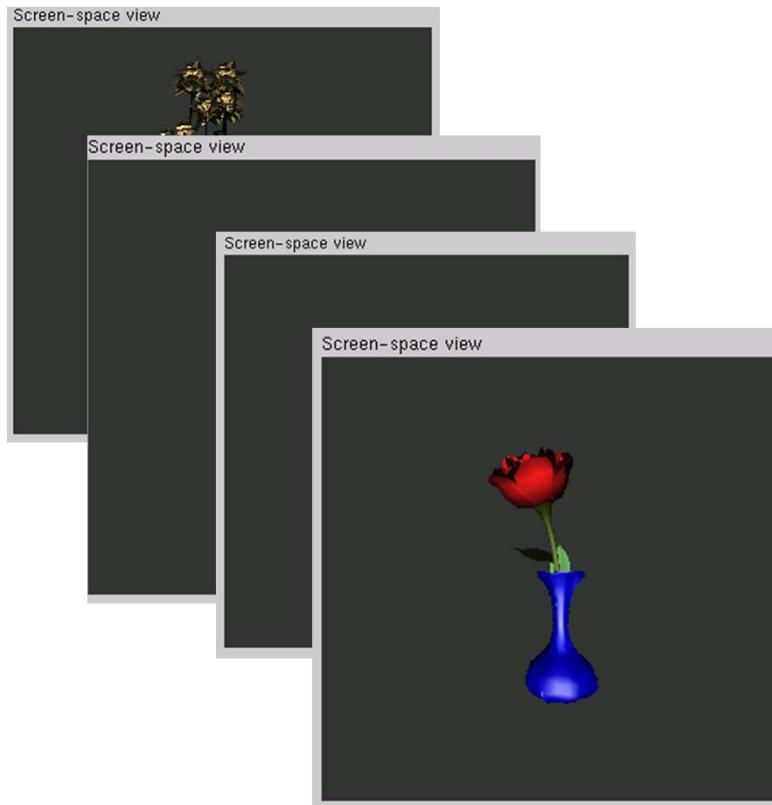
3D Viewing-Coordinate Parameters

Projection Transformations

Viewport Transformation and 3D Screen Coordinates

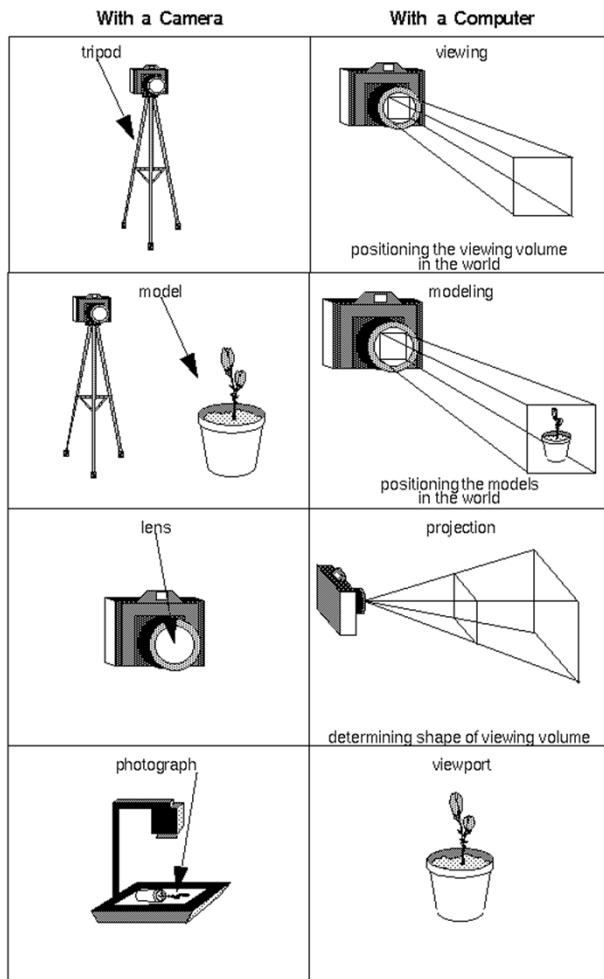
# Overview of 3D Viewing Concept

- How to construct 3D scenes in computers?
- How to take a picture by camera?



# Overview of 3D Viewing Concept

- Camera analog

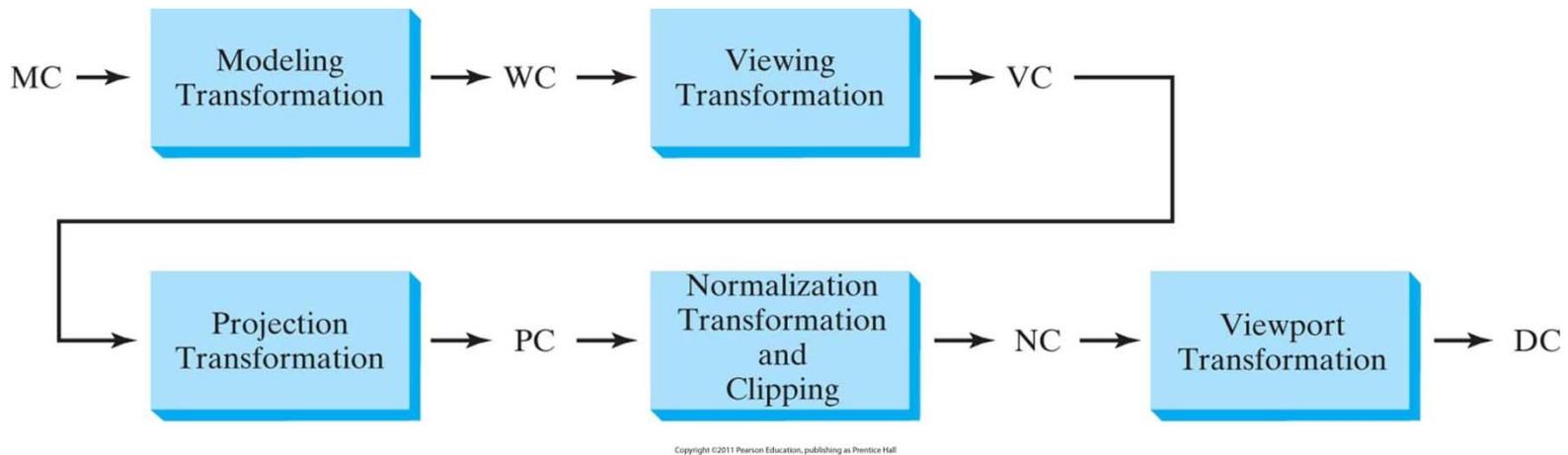


- Choose the position of the camera and pointing it at the scene (**viewing transformation**).
- Arranging the scene to be photographed into the desired composition (**modeling transformation**).
- Choosing a camera lens or adjusting the zoom (**projection transformation**).
- Determining how large you want the final photograph to be (**viewport transformation**).

(From: the red book)

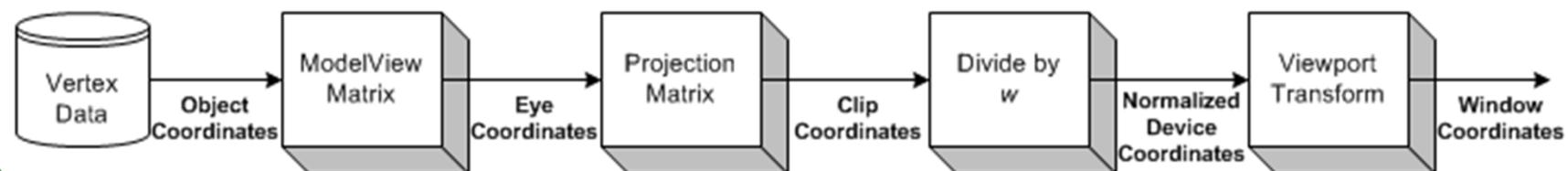


# 3D Viewing Pipeline vs. OpenGL Pipeline

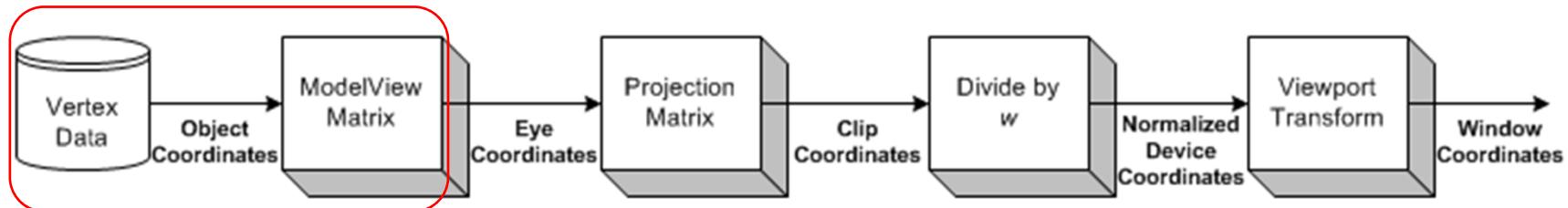


In OpenGL pipeline, geometric data such as vertex positions and normal vectors are transformed via **Vertex Operation** and **Primitive Assembly Operation** before rasterization process.

OpenGL vertex transformation:



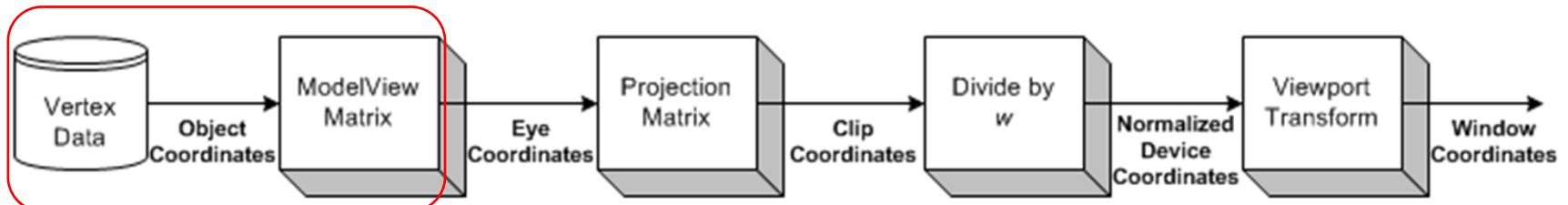
# OpenGL Vertex Transformation



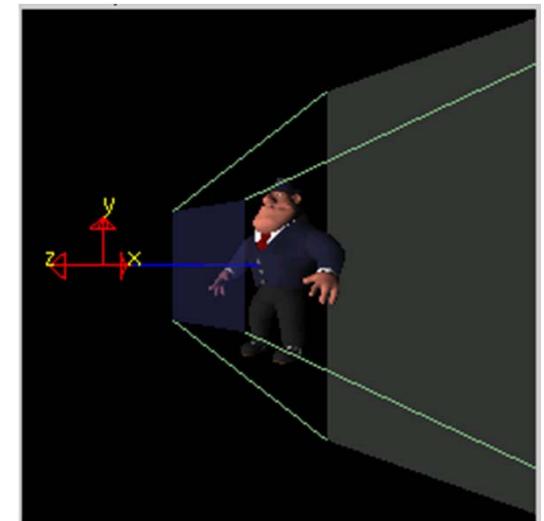
- Object coordinates
  - The local coordinate system of objects and represent the initial position and orientation of objects before any transform is applied.
  - Specify them with `glVertex*`() or `glVertexPointer()`.
  - To transform objects, use `glRotatef()`, `glTranslatef()`, `glScalef()`.



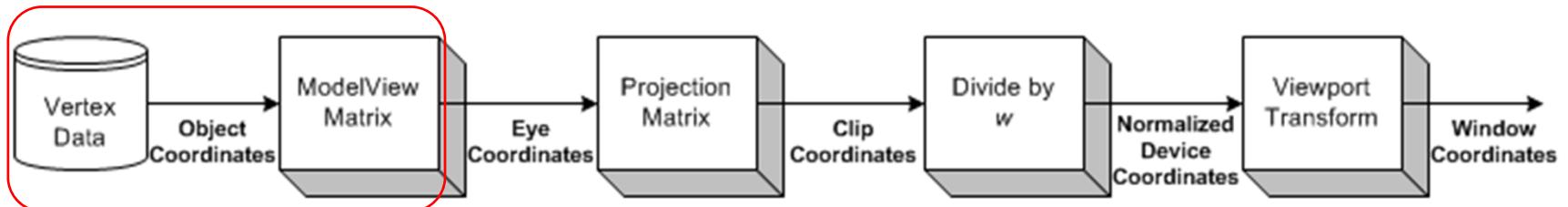
# OpenGL Vertex Transformation



- Eye coordinates
  - Using **GL\_MODELVIEW** matrix to transform objects from “object space” to “eye space”.  
(multiply **GL\_MODELVIEW** matrix and object coordinates)
  - **GL\_MODELVIEW** matrix is a combination of Model and View matrices ( $M_{view} \cdot M_{model}$ ).
    - $M_{model}$  is to construct objects from “object space/local space” to “world space”.
    - $M_{view}$  is to convert objects from “world space” to “eye space” (camera).



# OpenGL Vertex Transformation

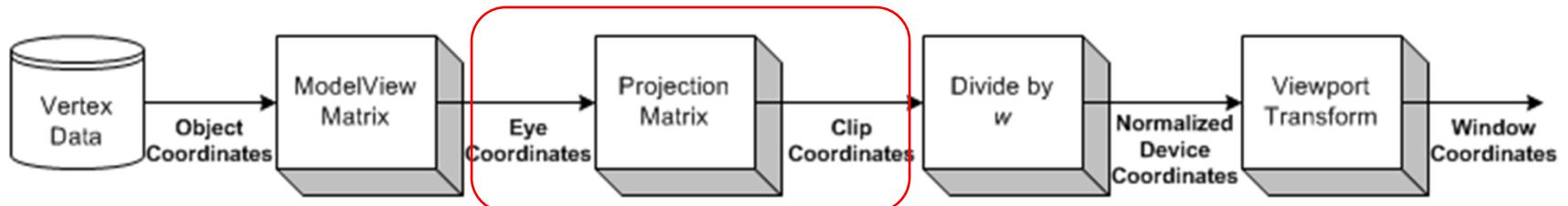


- Eye coordinates (cont.)

$$\begin{pmatrix} x_{eye} \\ y_{eye} \\ z_{eye} \\ w_{eye} \end{pmatrix} = M_{modelView} \cdot \begin{pmatrix} x_{obj} \\ y_{obj} \\ z_{obj} \\ w_{obj} \end{pmatrix} = M_{view} \cdot M_{model} \cdot \begin{pmatrix} x_{obj} \\ y_{obj} \\ z_{obj} \\ w_{obj} \end{pmatrix}$$

- Note: by default, OpenGL defines that the camera is always located at (0, 0, 0) and facing to -Z axis in the **eye space coordinates**.

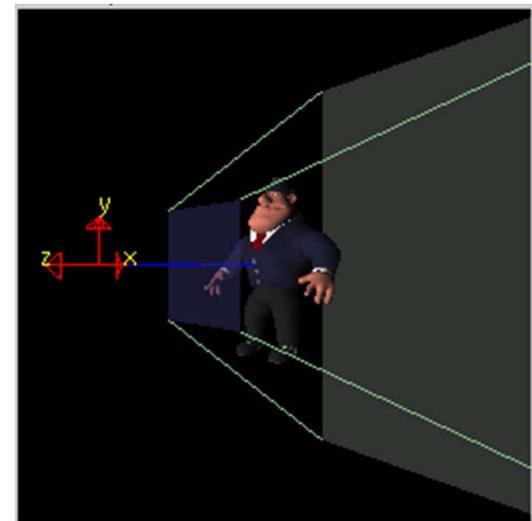
# OpenGL Vertex Transformation



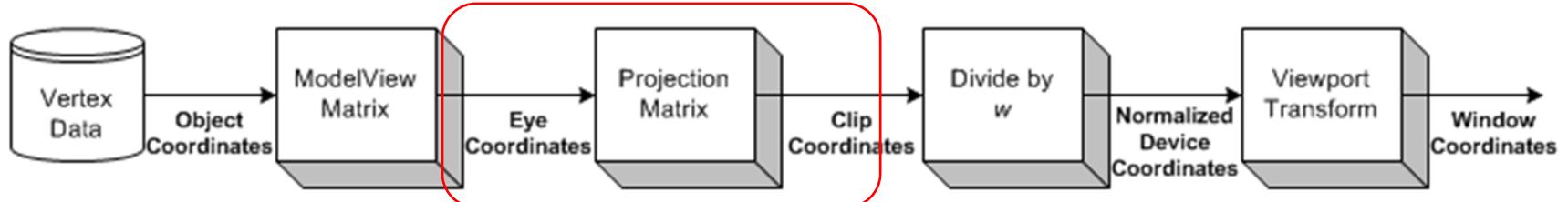
- Clip coordinates
  - Apply the **Projection matrix** to transform objects from Eye Coordinates to Clip Coordinates.

$$\begin{pmatrix} x_{clip} \\ y_{clip} \\ z_{clip} \\ w_{clip} \end{pmatrix} = M_{projection} \cdot \begin{pmatrix} x_{eye} \\ y_{eye} \\ z_{eye} \\ w_{eye} \end{pmatrix}$$

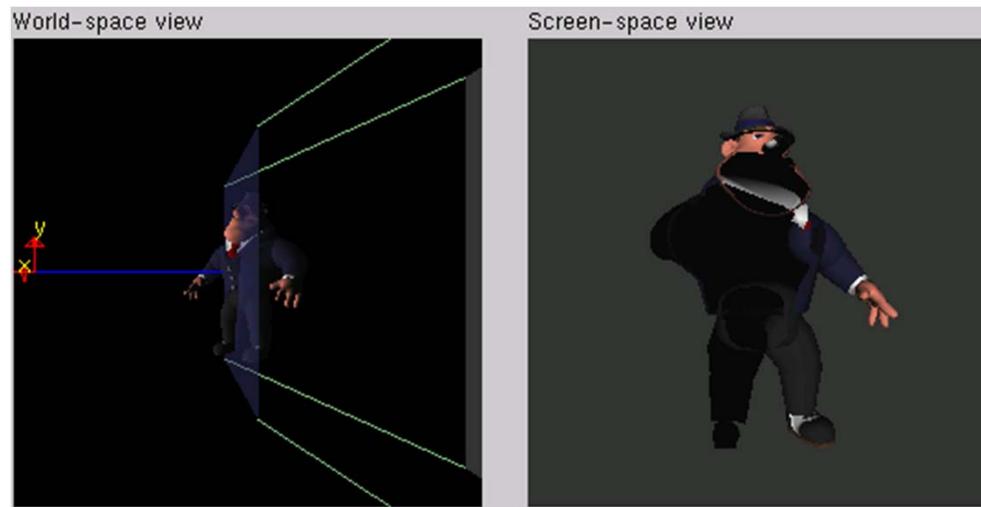
- Viewing volume
  - How objects are projected onto screen (perspective or parallel(orthogonal));
  - Which objects or portions of objects are clipped out of the final image.



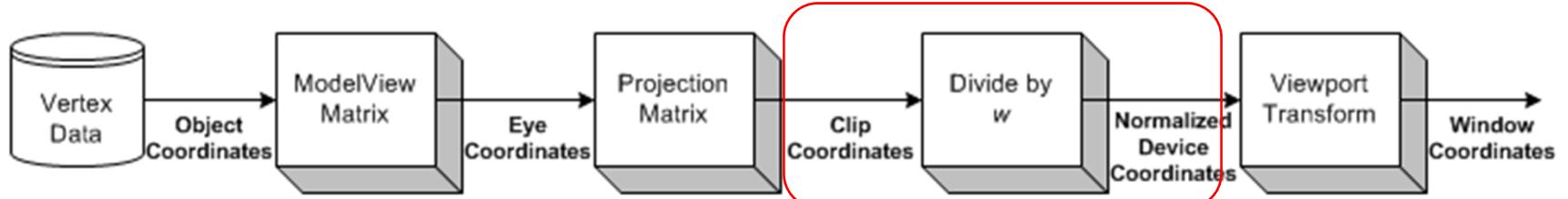
# OpenGL Vertex Transformation



- Clip coordinates (cont.)
  - Objects are clipped out from the **viewing volume**



# OpenGL Vertex Transformation

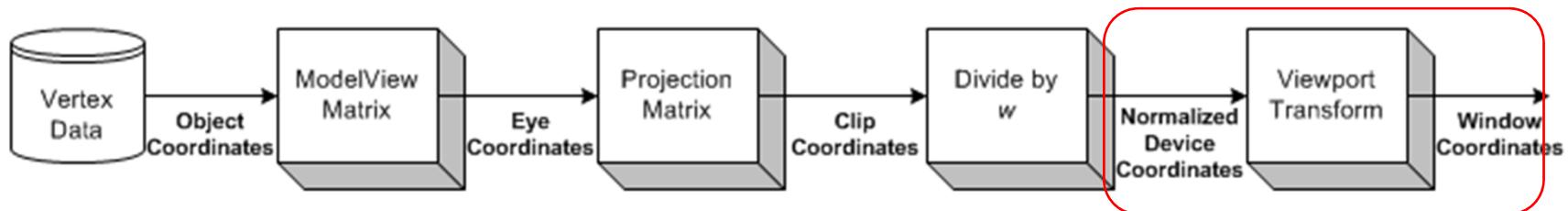


- Normalized Device Coordinates (NDC)
  - Transform the values into the range of [-1, 1] in all three axes.
  - In OpenGL, it is implemented by the **Perspective Division** on the Clip Coordinates.

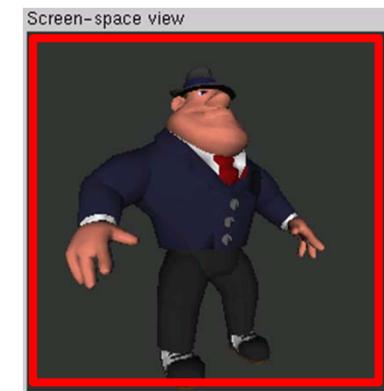
$$\begin{pmatrix} x_{ndc} \\ y_{ndc} \\ z_{ndc} \end{pmatrix} = \begin{pmatrix} x_{clip}/w_{clip} \\ y_{clip}/w_{clip} \\ z_{clip}/w_{clip} \end{pmatrix}$$

*(That divides the Clip Coordinates by  $w_{clip}$ .)*

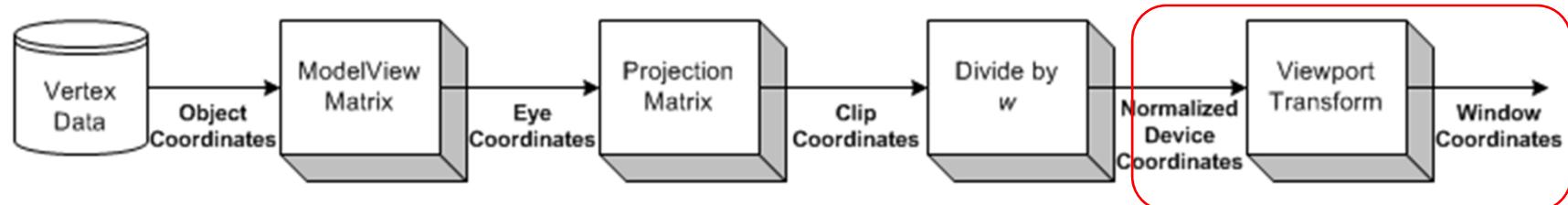
# OpenGL Vertex Transformation



- Window Coordinates
  - Result from scaling and translating Normalized Device Coordinates by the **viewport** transformation.
  - They are controlled by the parameters of the viewport you defined
    - **glViewport()**: to define the rectangle of the rendering area where the final image is mapped.
    - **glDepthRange()**: to determine the z value of the window coordinates.



# OpenGL Vertex Transformation



- Window Coordinates (cont.)
  - `glViewport(x, y, w, h);`  
`glDepthRange(n, f);` n: near, f: far
  - NDC->WC (Viewport)  
 $[-1,1; -1,1; -1,1] \Rightarrow [x, x+w; y, y+h; n,f]$

$$\begin{pmatrix} x_w \\ y_w \\ z_w \end{pmatrix} = \begin{pmatrix} \frac{w}{2}x_{ndc} + (x + \frac{w}{2}) \\ \frac{h}{2}y_{ndc} + (y + \frac{h}{2}) \\ \frac{f-n}{2}z_{ndc} + \frac{f+n}{2} \end{pmatrix}$$



# Chapter 10

## Three-Dimensional Viewing (OpenGL functions)

### Part I.

Overview of 3D Viewing Concept

3D Viewing Pipeline vs. OpenGL Pipeline

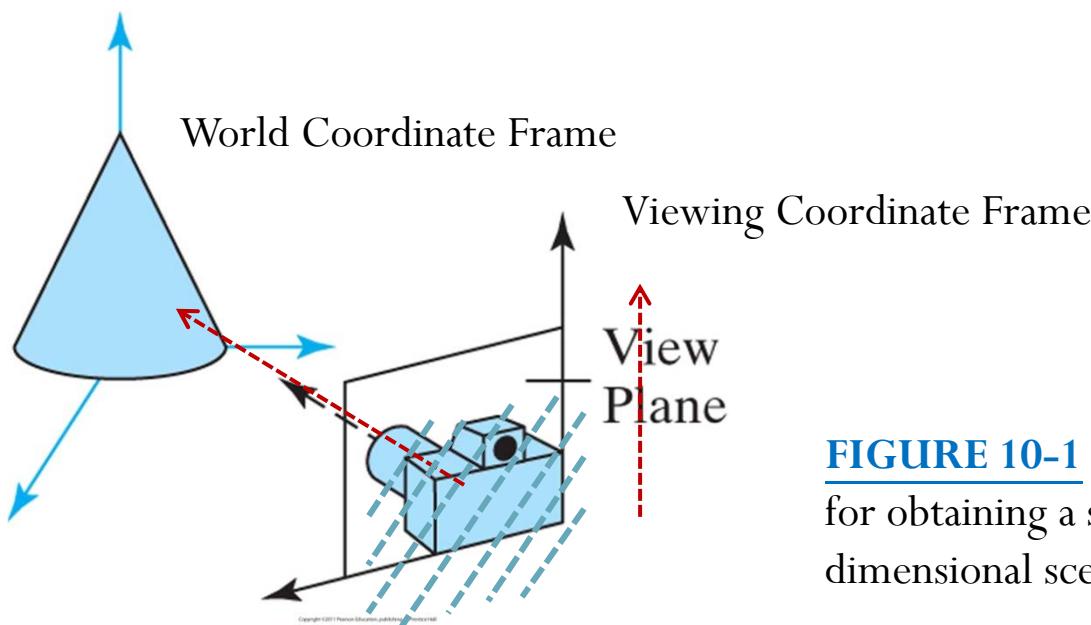
### **3D Viewing-Coordinate Parameters**

Projection Transformations

Viewport Transformation and 3D Screen Coordinates

# Coordinate reference for “camera”

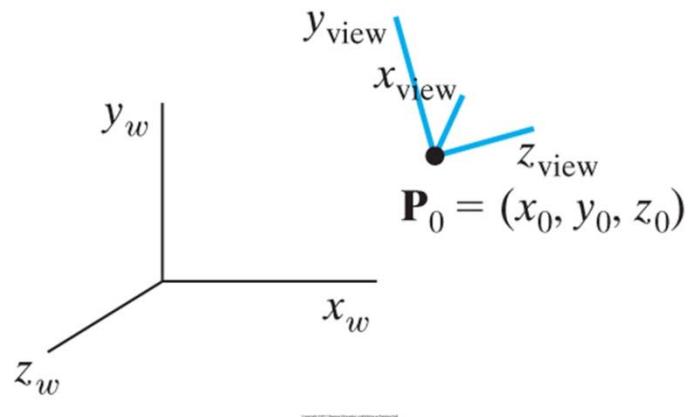
- To set up the **viewing coordinate reference** (or camera)
  - Position and orientation of a **view plane** (or projection plane)
  - Objects are transferred to the **viewing reference coordinates** and projected onto the **view plane**



**FIGURE 10-1** Coordinate reference for obtaining a selected view of a three-dimensional scene.

# 3D Viewing-Coordinate Parameters

- Establish a 3D viewing reference frame
  - Right-handed

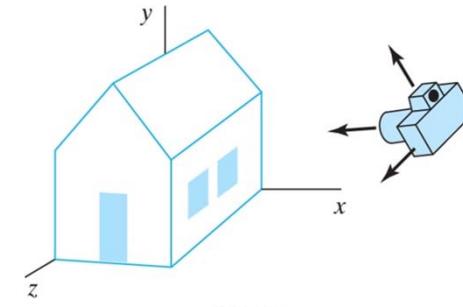


## a. The viewing origin

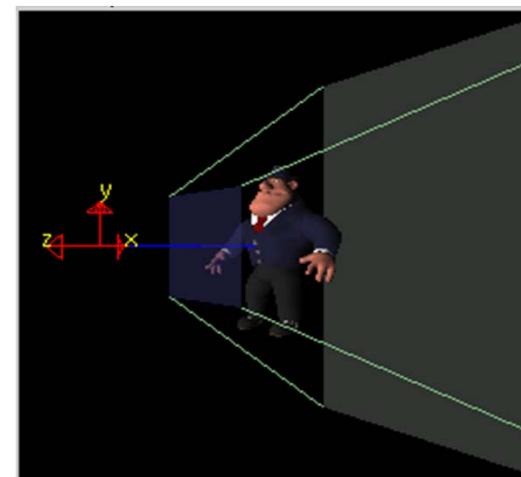
- Define the view point or viewing position (sometimes referred to as the eye position or the camera position)

## b. $y_{\text{view}}$ -- view-up vector $\mathbf{V}$

- Defines  $y_{\text{view}}$  direction



**FIGURE 10-7** A right-handed viewing-coordinate system, with axes  $x_{\text{view}}$ ,  $y_{\text{view}}$ ,  $z_{\text{view}}$ , relative to a right-handed world-coordinate frame



# 3D Viewing-Coordinate Parameters

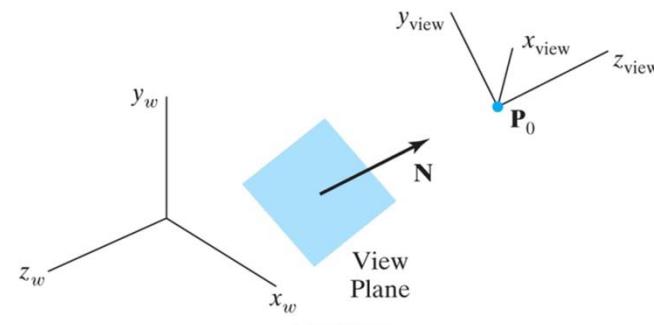
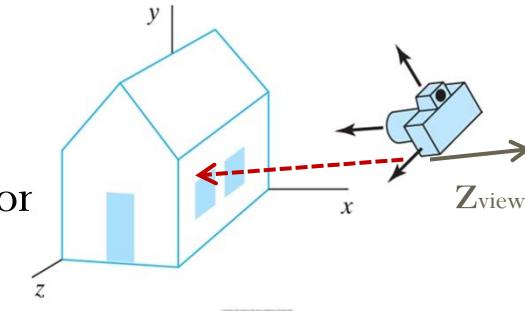
- Viewing direction and view plane

- c.  $\mathbf{z}_{\text{view}}$ : viewing direction

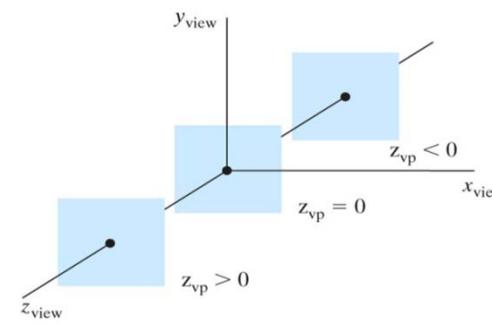
- Along the  $\mathbf{z}_{\text{view}}$  axis, often in the negative  $\mathbf{z}_{\text{view}}$  direction

- d. The view plane (also called projection plane)

- Perpendicular to  $\mathbf{z}_{\text{view}}$  axis
  - The orientation of the view plane can be defined by a view-plane **normal vector  $\mathbf{N}$**
  - The different position of the view-plane along the  $\mathbf{z}_{\text{view}}$  axis



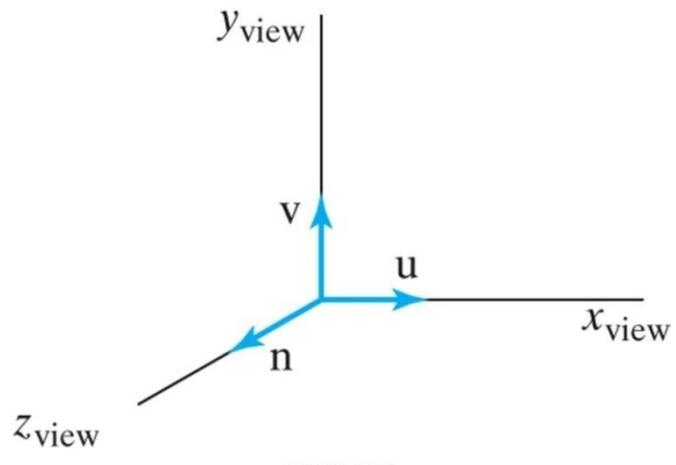
**FIGURE 10-8** Orientation of the view plane and view-plane normal vector  $\mathbf{N}$



**FIGURE 10-9** Three possible positions for the view plane along the  $\mathbf{z}_{\text{view}}$  axis

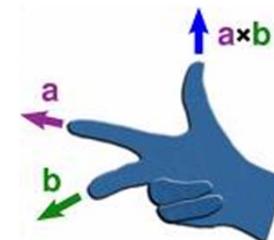
# 3D Viewing-Coordinate Parameters

- The **uvn** Viewing-Coordinate Reference Frame (*Viewing Coordinate System*)
  - Direction of  $\mathbf{z}_{\text{view}}$  axis: the **view-plane normal** vector  $\mathbf{N}$ ;
  - Direction of  $\mathbf{y}_{\text{view}}$  axis: the **view-up vector**  $\mathbf{V}$ ;
  - Direction of  $\mathbf{x}_{\text{view}}$  axis: taking the vector cross product of  $\mathbf{V}$  and  $\mathbf{N}$  to get  $\mathbf{U}$ .



$$\begin{aligned}\mathbf{n} &= \frac{\mathbf{N}}{|\mathbf{N}|} = (n_x, n_y, n_z) \\ \mathbf{u} &= \frac{\mathbf{V} \times \mathbf{n}}{|\mathbf{V} \times \mathbf{n}|} = (u_x, u_y, u_z) \\ \mathbf{v} &= \mathbf{n} \times \mathbf{u} = (v_x, v_y, v_z)\end{aligned}\quad (10-1)$$

**FIGURE 10-12** A right-handed viewing system defined with unit vectors  $\mathbf{u}$ ,  $\mathbf{v}$ , and  $\mathbf{n}$ .



# Chapter 10

## Three-Dimensional Viewing

---

### Part I.

Overview of 3D Viewing Concept

3D Viewing Pipeline vs. OpenGL Pipeline

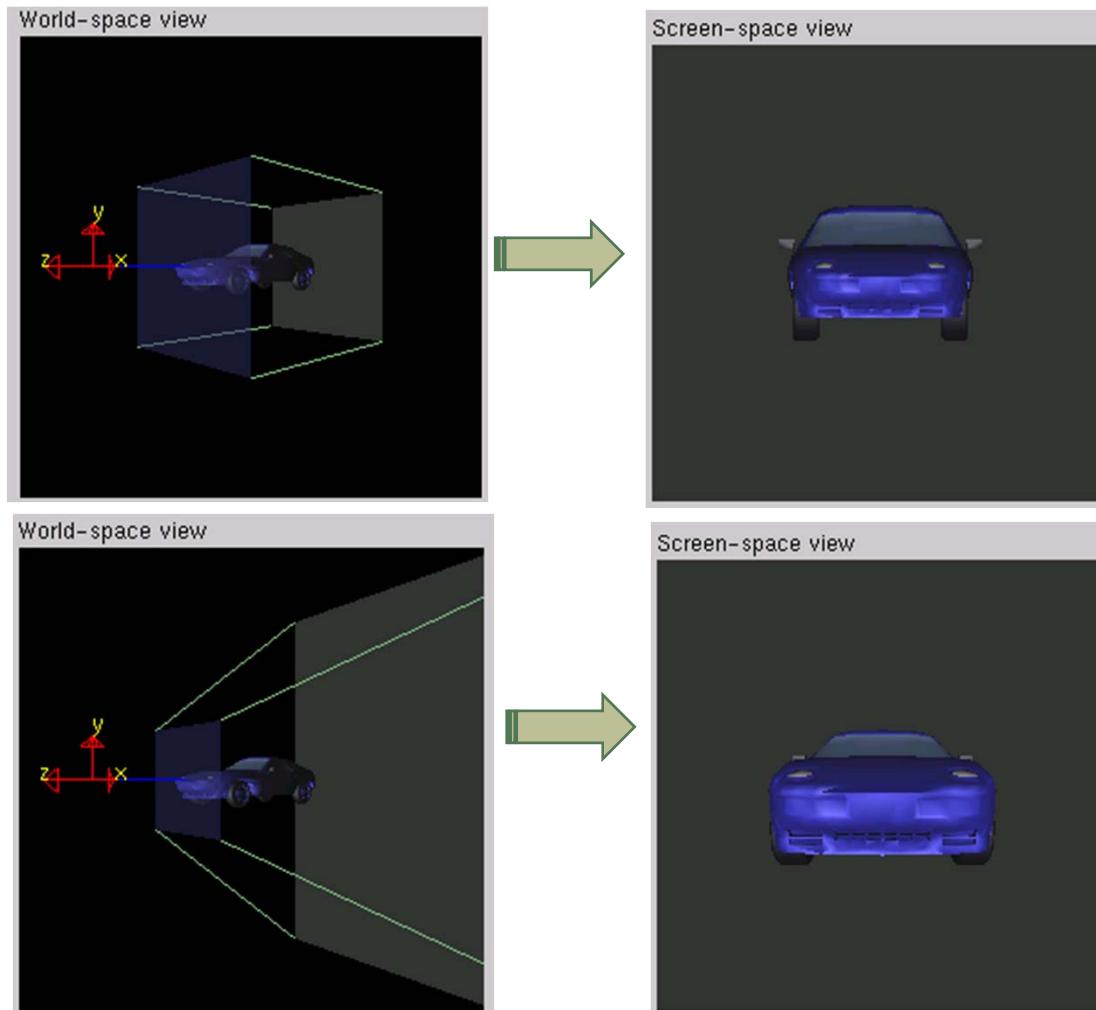
3D Viewing-Coordinate Parameters

### **Projection Transformations**

Viewport Transformation and 3D Screen Coordinates

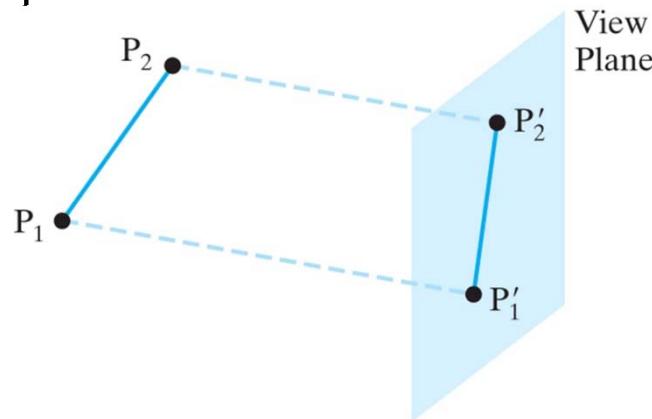
# Projection Transformations

- Objects are projected to the view plane.

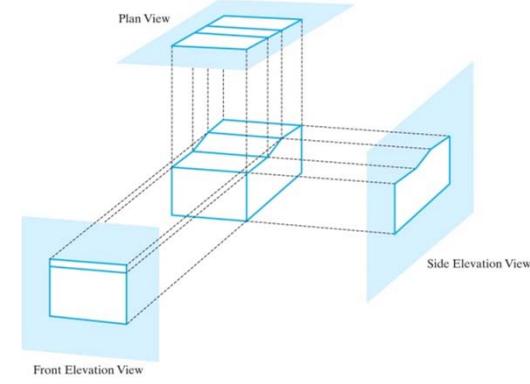
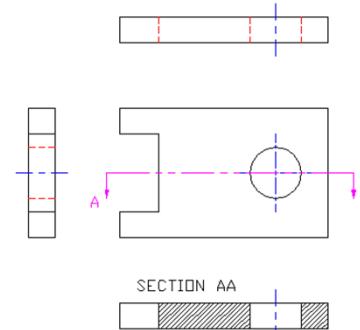
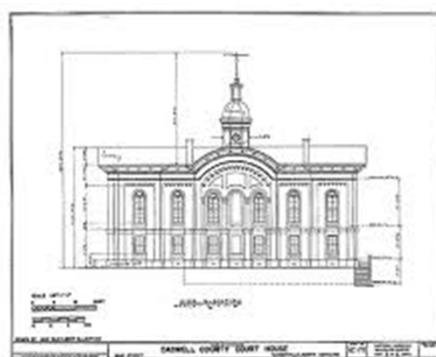


# Orthogonal Projection (a special case of Parallel Proj.)

- Coordinates positions are transferred to the view plane along parallel lines



**FIGURE 10-15** Parallel projection of a line segment onto a view plane.

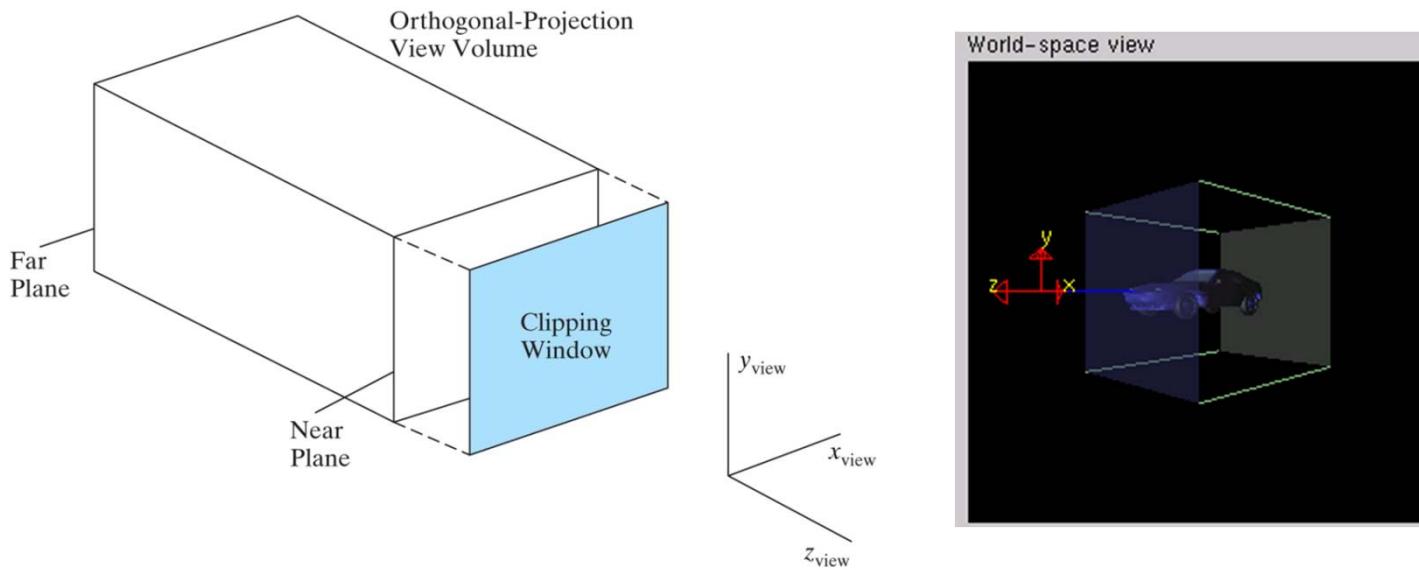


Engineering and architecture drawings commonly employ it. Length and angles are accurately depicted.

# Orthogonal Projection

(Projector is orthogonal to the view plane)

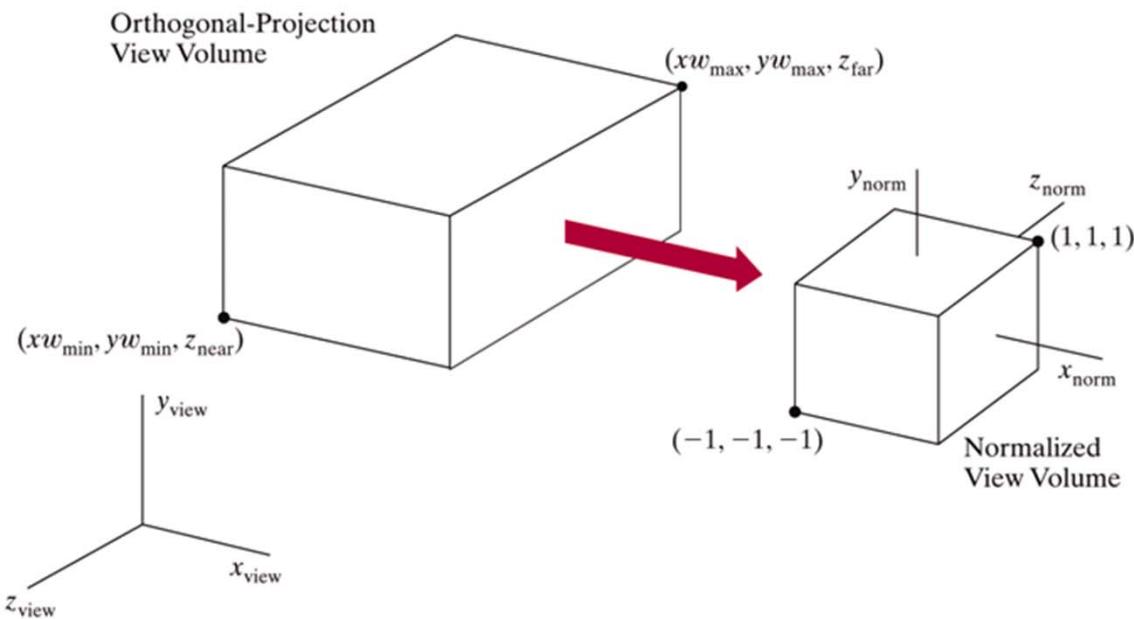
- Clipping window and orthogonal-projection view volume
  - The **clipping window**: the **x** and **y** limits of the scene you want to display
  - These form the **orthogonal-projection view volume**
  - The depth is limited by near and far clipping planes in  **$z_{\text{view}}$**



**FIGURE 10-22** A finite orthogonal view volume with the **view plane** “in front” of the near plane

# Normalization Transformation for Orthogonal Projections

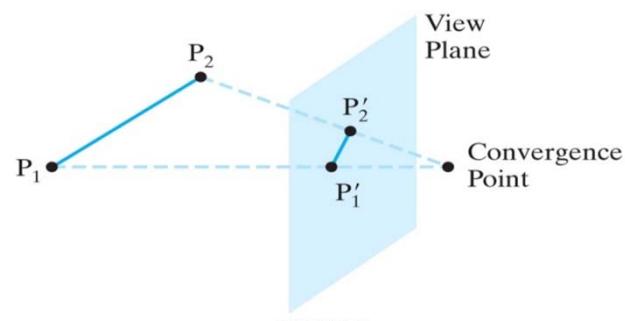
- Map the view volume into a **normalized view volume**



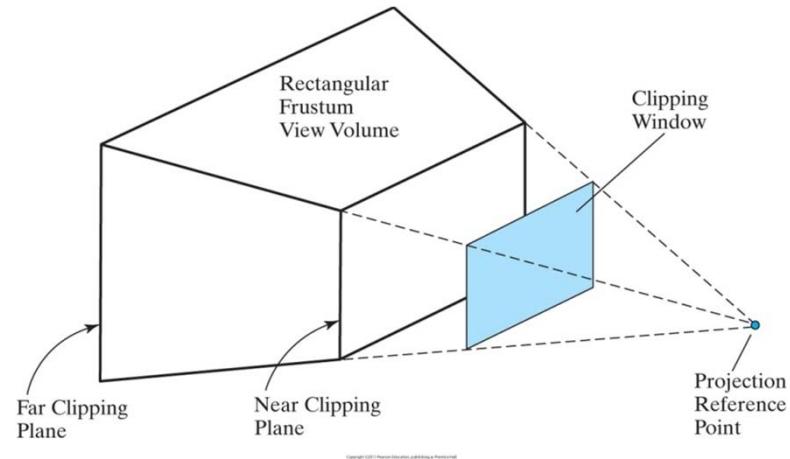
**FIGURE 10-24** Normalization transformation from an orthogonal-projection view volume to the symmetric normalization cube within a left-handed reference frame.

# Perspective Projections

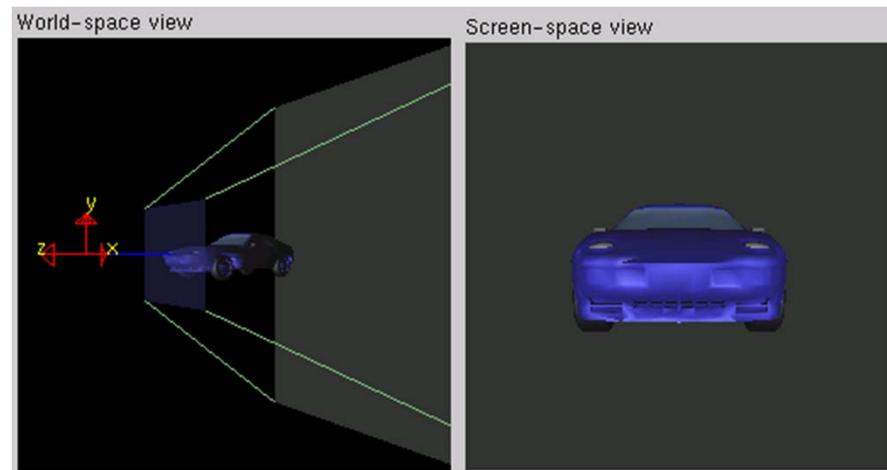
- Positions are transferred along lines that converge to a point behind the view plane.



**FIGURE 10-16** Perspective projection of a line segment onto a view plane.

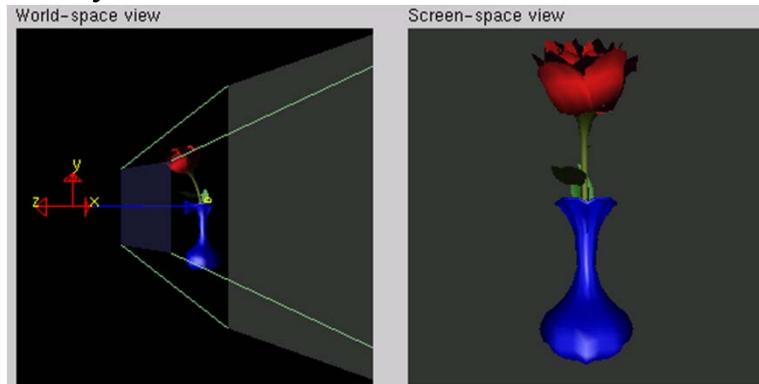


**FIGURE 10-39**  
A perspective-projection **frustum view volume** with the **view plane** “in front” of the **near clipping plane**

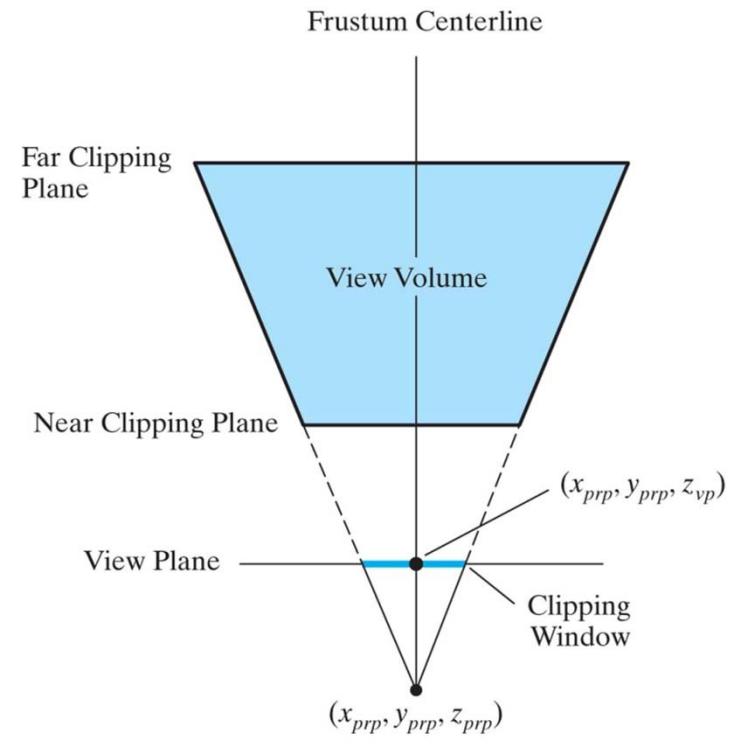
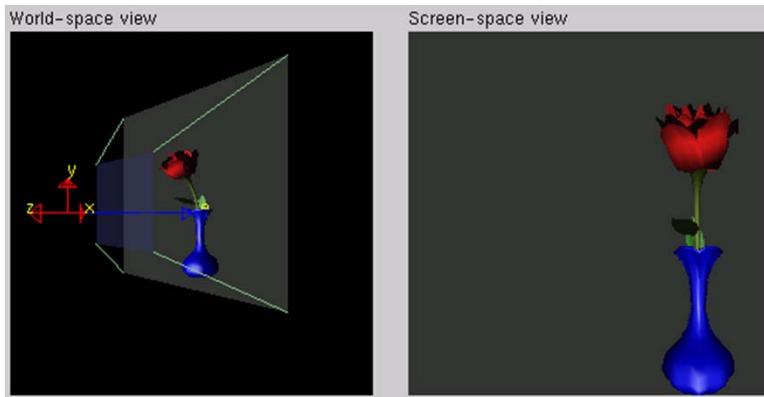


# Perspective Projections

- Perspective projection view volume
  - Symmetric



- Asymmetric

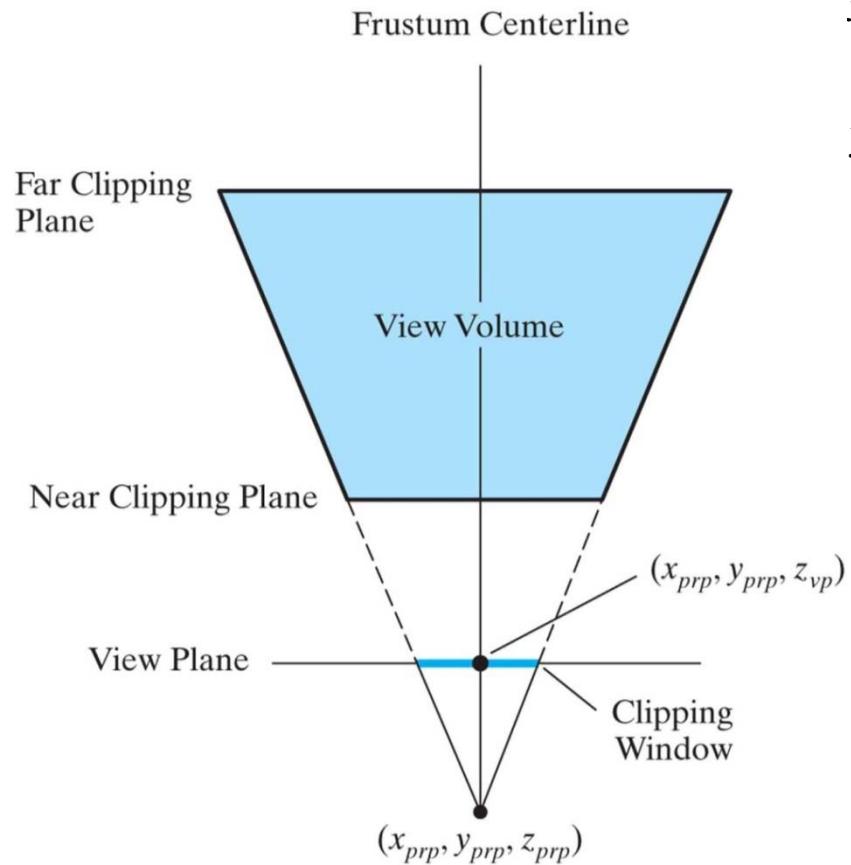


**FIGURE 10-40** A symmetric perspective-projection frustum view volume.

# Symmetric Perspective Projections

## Frustum

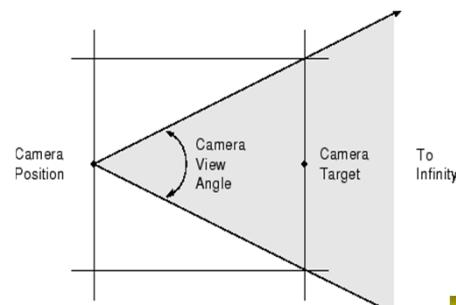
- The corner positions for the clipping window in terms of the window's width and height:



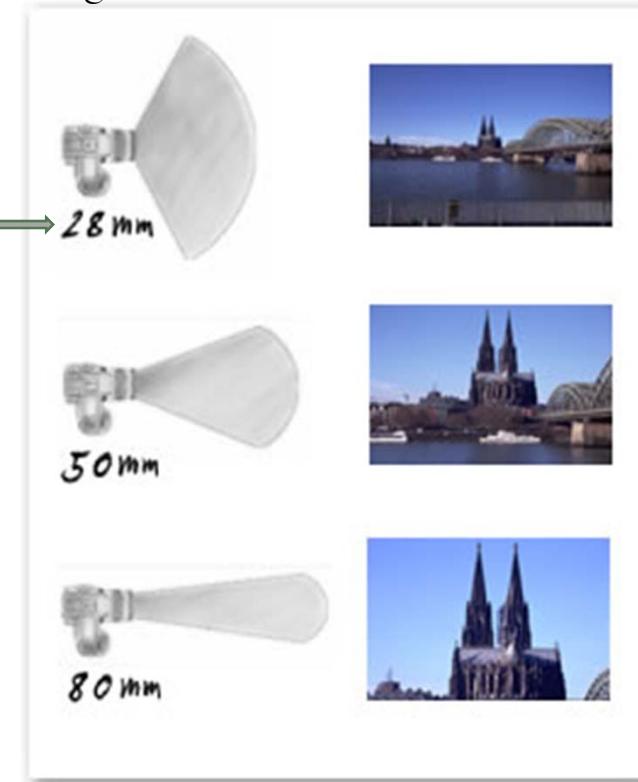
$$xw_{\min} = x_{prp} - \frac{width}{2}, \quad xw_{\max} = x_{prp} + \frac{width}{2}$$
$$yw_{\min} = y_{prp} - \frac{height}{2}, \quad yw_{\max} = y_{prp} + \frac{height}{2}$$

# Symmetric Perspective Projections Frustum: field-of-view angle/ angle of view

- Another way to specify the symmetric-perspective projection volume
  - Approximate the properties of a camera lens: the **field-of-view angle / angle of view**
    - E.g.: a wide-angle lens corresponds to a larger angle of view.

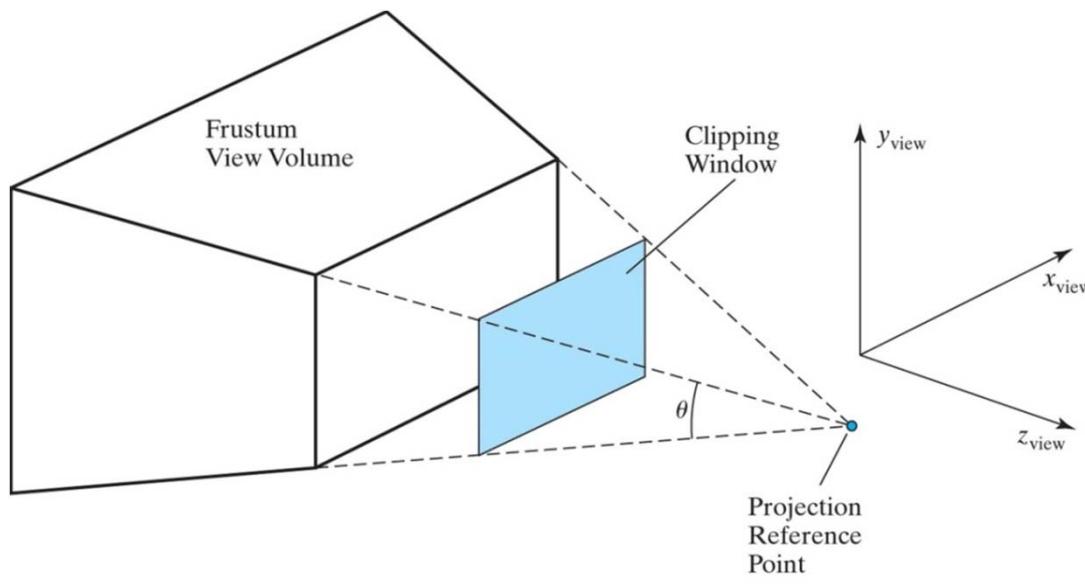


*Focal Length*



# Symmetric Perspective Projections Frustum: field-of-view angle/ angle of view

- Another way to specify the symmetric-perspective projection volume
  - In CG, the angle is between the top clipping plane and the bottom clipping plane

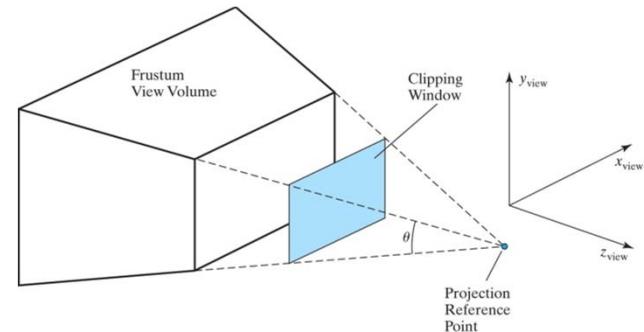


**FIGURE 10-41** Field-of-view angle  $\theta$  for a symmetric perspective-projection view volume.

# Symmetric Perspective Projections

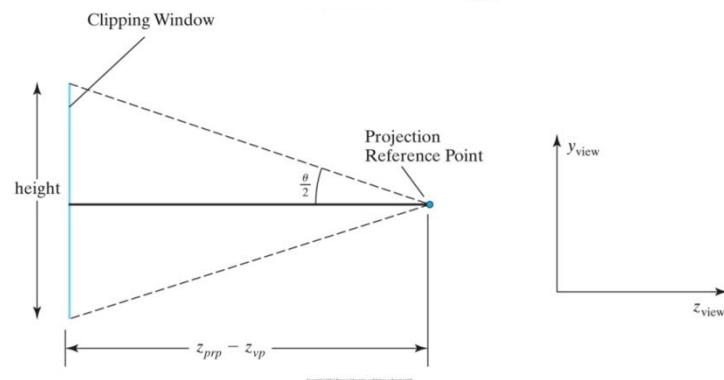
## Frustum: field-of-view angle

- For a given projection reference point and view plane position
  - The height of the clipping window is:



$$\tan\left(\frac{\theta}{2}\right) = \frac{height/2}{z_{prp} - z_{vp}}$$

$$height = 2(z_{prp} - z_{vp})\tan\left(\frac{\theta}{2}\right)$$



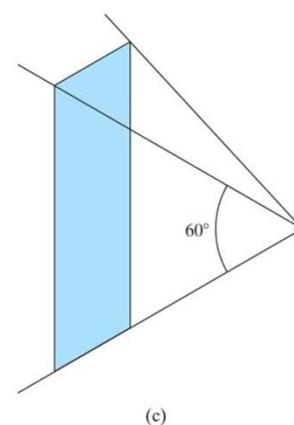
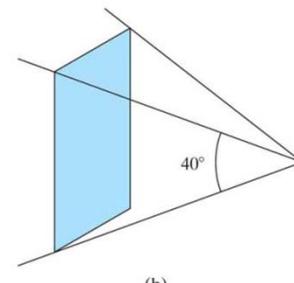
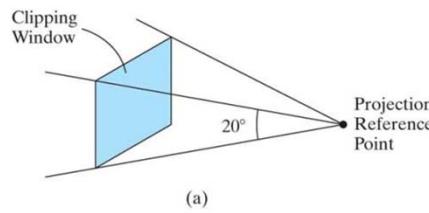
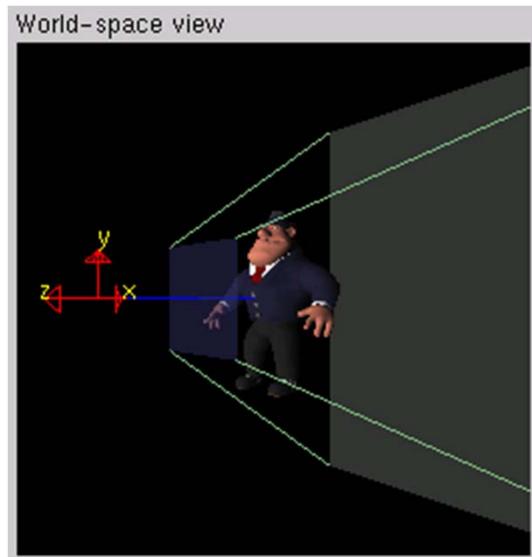
How about the width?

Another parameter: the aspect ratio.

# Symmetric Perspective Projections

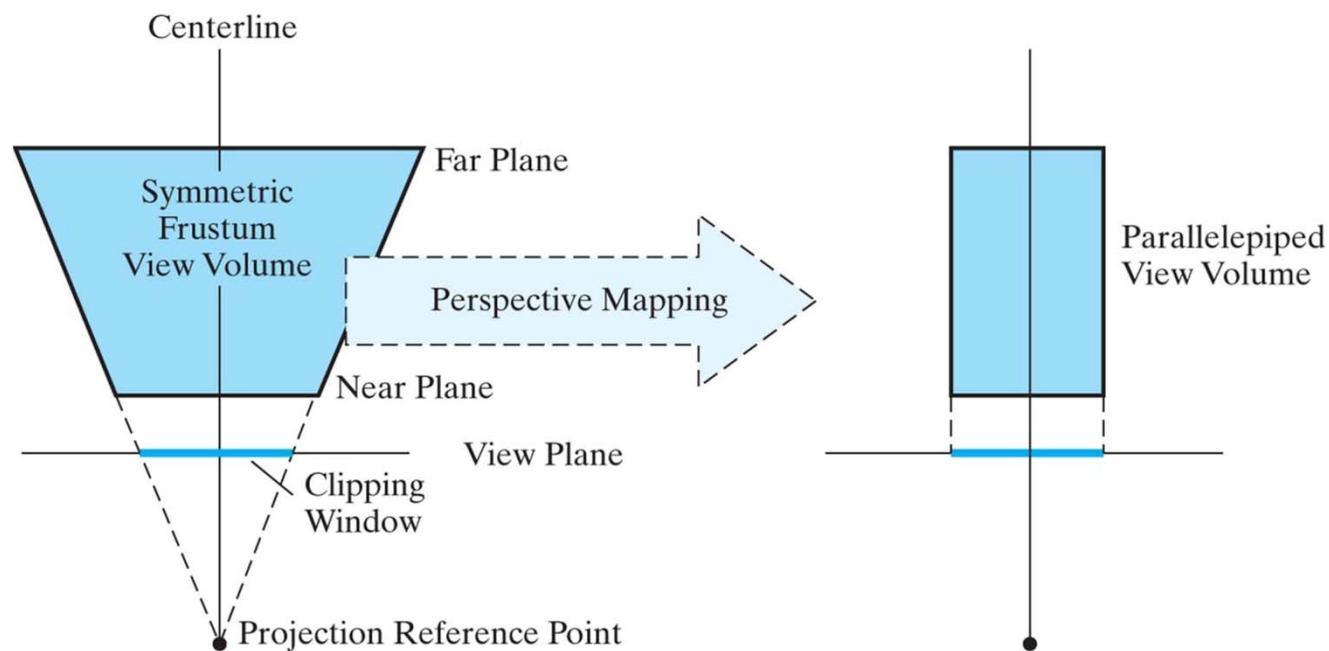
## Frustum: field-of-view angle

- Changing field-of-view angle



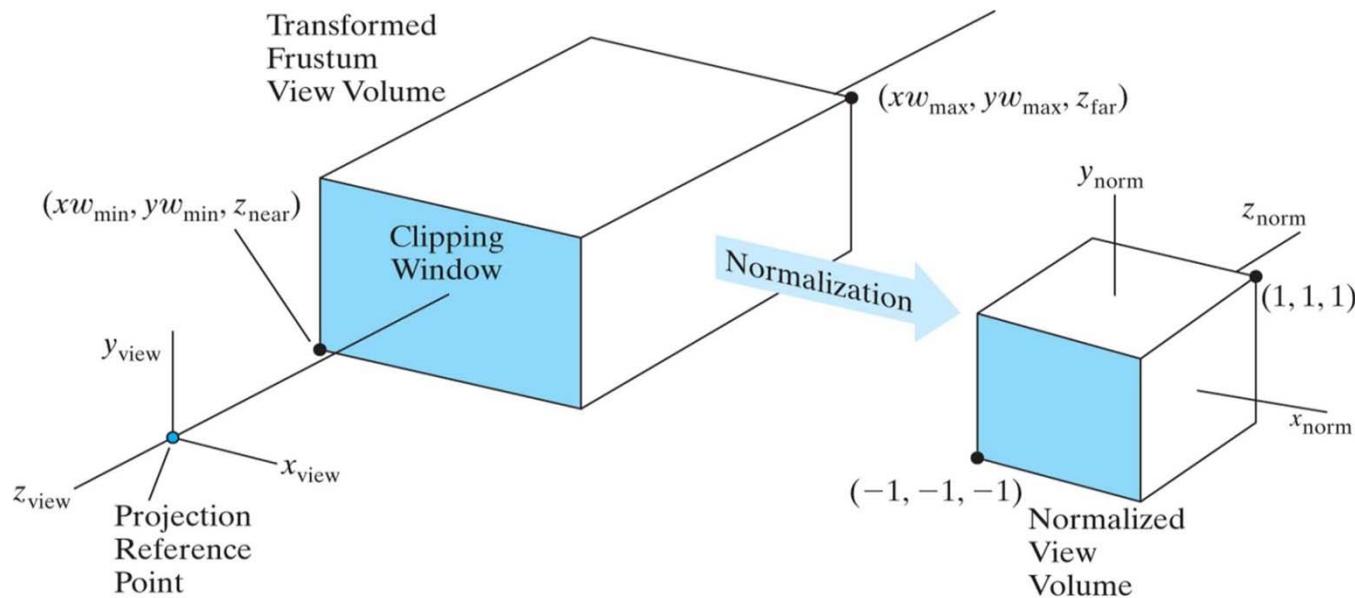
# Normalization Transformation of Perspective Projections

- Mapped to a rectangular parallelepiped (平行六面体)
  - The centerline of the parallelepiped is the frustum centerline.
  - All points along a projection line within the frustum map to the same point on the view plane -> each projection line is converted by the perspective transformation to a line that is perpendicular to the view plane, and parallel to the frustum centerline.



# Normalization Transformation of Perspective Projections

- The rectangular parallelepiped is mapped to a symmetric normalized cube within a left-handed frame.



**FIGURE 10-46** Normalization transformation from a transformed **perspective projection view volume** (rectangular parallelepiped) to the symmetric **normalization cube** within a left-handed reference frame, with the near clipping plane as the view plane and the **projection reference point** at the viewing-coordinate origin.

# Chapter 10

## Three-Dimensional Viewing

---

### Part I.

Overview of 3D Viewing Concept

3D Viewing Pipeline vs. OpenGL Pipeline

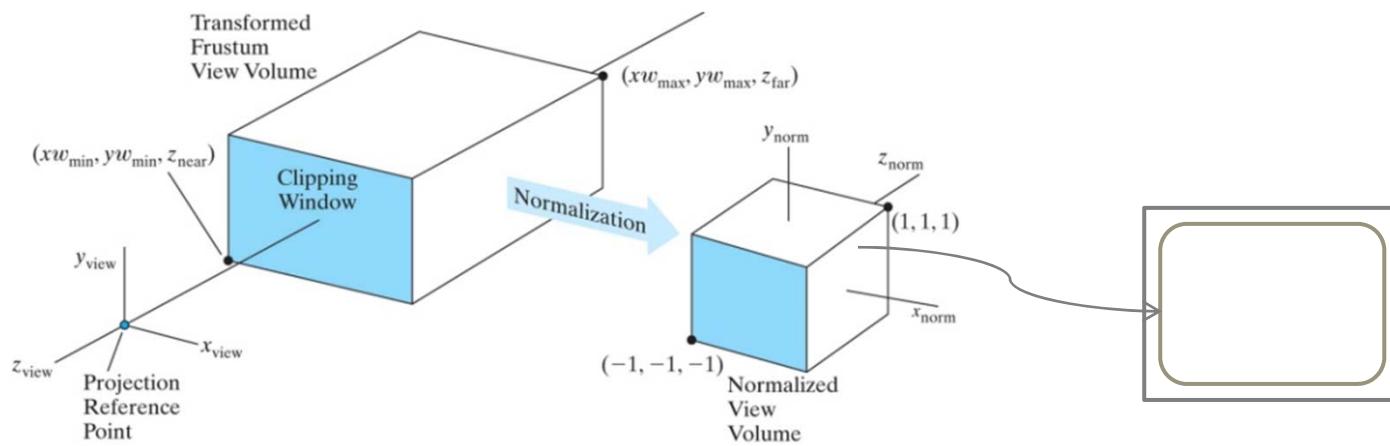
3D Viewing-Coordinate Parameters

Projection Transformations

**Viewport Transformation and 3D Screen  
Coordinates**

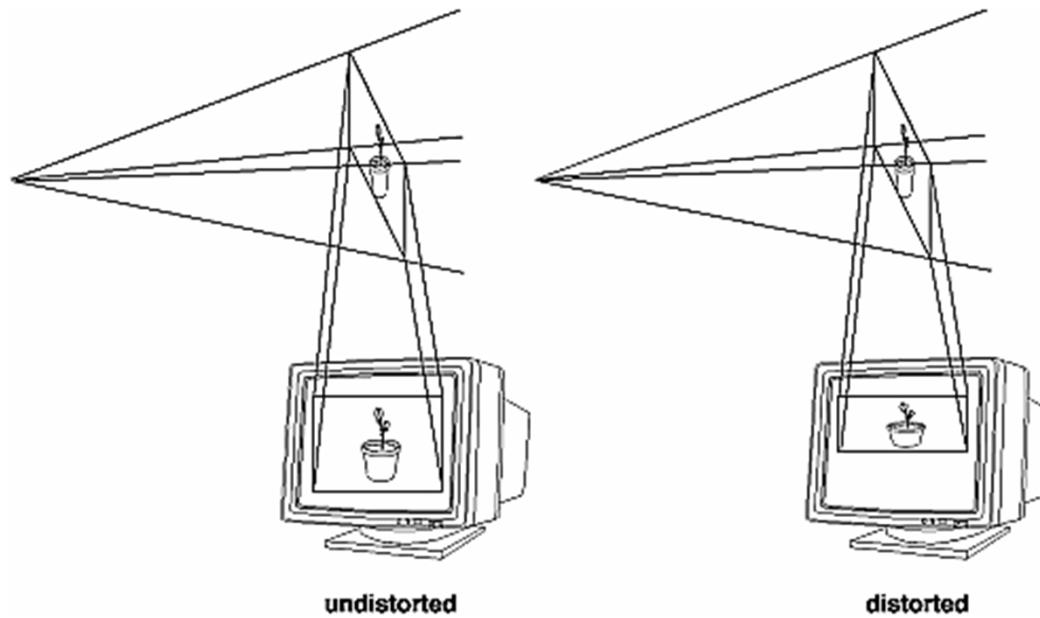
# Viewport Transformation and 3D Screen Coordinates

- Transform the normalized projection coordinates to screen coordinates (3D screen coordinates)
  - For **x** and **y** in the normalized clipping window, the transformation is the same as 2D viewport transformation
  - For **z** (depth)
    - for the visibility testing and surface rendering algorithms
- In normalized coordinates, the  $Z_{\text{norm}} = -1$  face of symmetric cube corresponds to the clipping-window area
  - This face is mapped to the 2D screen viewport, that is  $Z_{\text{screen}} = 0$ .
- The z (depth) value for each screen point
  - Depth buffer or z-buffer



# Viewport Mapping

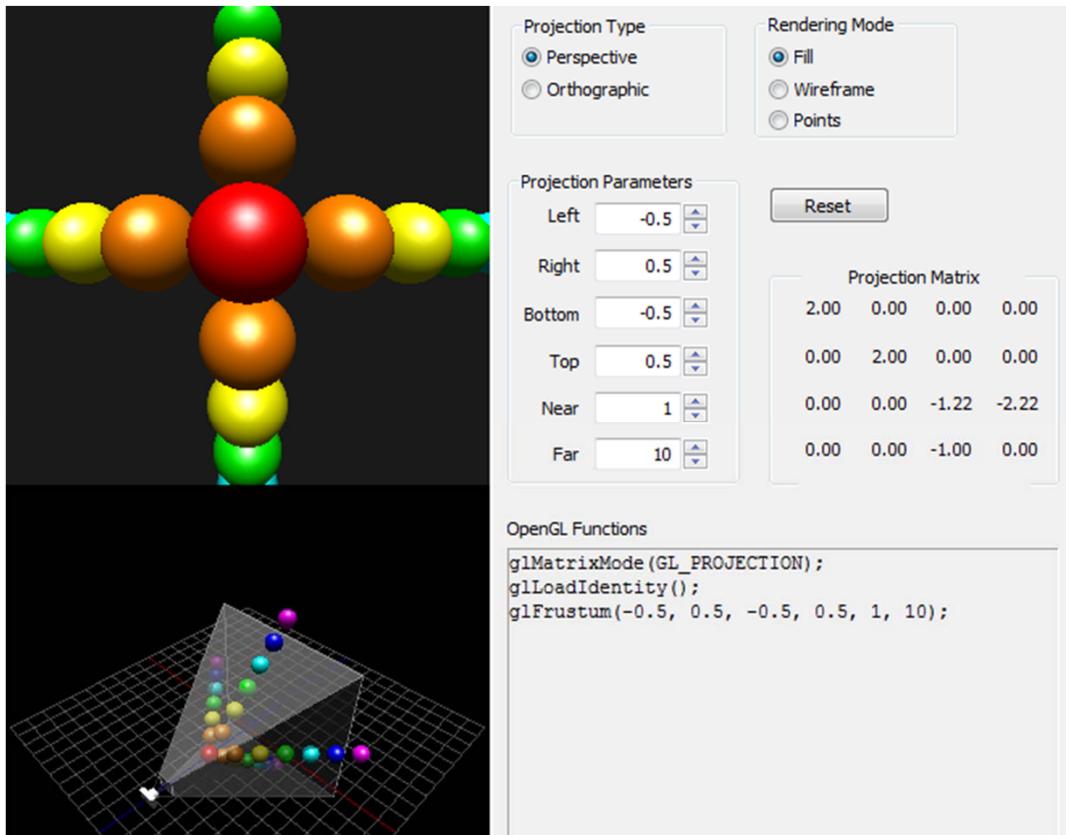
- Mapping the viewing volume to the viewport



(From OpenGL Super Bible)

The aspect ratio of a **viewport** should generally equal the aspect ratio of the **viewing volume**. If the two ratios are different, the projected image will be distorted when mapped to the viewport.

# Projection Demo



# Chapter 10

## Three-Dimensional Viewing

---

Part II.

OpenGL 3D Viewing Functions

OpenGL 3D Projection Functions

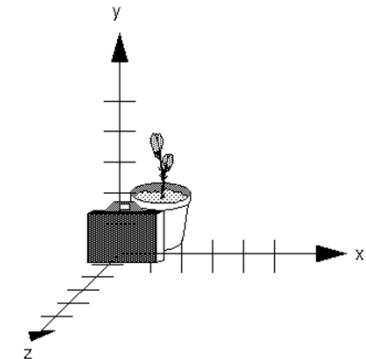
    Orthogonal-Projection Function

    Perspective-Projection Functions

OpenGL 3D Viewing Program Example

# OpenGL 3D Viewing Functions

- A **viewing transformation** changes the position and orientation of the viewpoint.
  - Recall the camera analogy, it positions the camera tripod, pointing the camera toward the model.
  - Composed of **translations** and **rotations**.
  - The same effects can be implemented either
    - move the camera or
    - move the objects in the opposite direction.

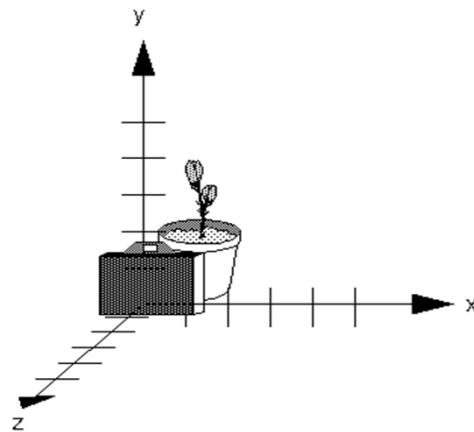


Note: The **viewing transformation** commands should be called **before** any **modeling transformations** are performed, so that the modeling transformations take effect on the objects first.

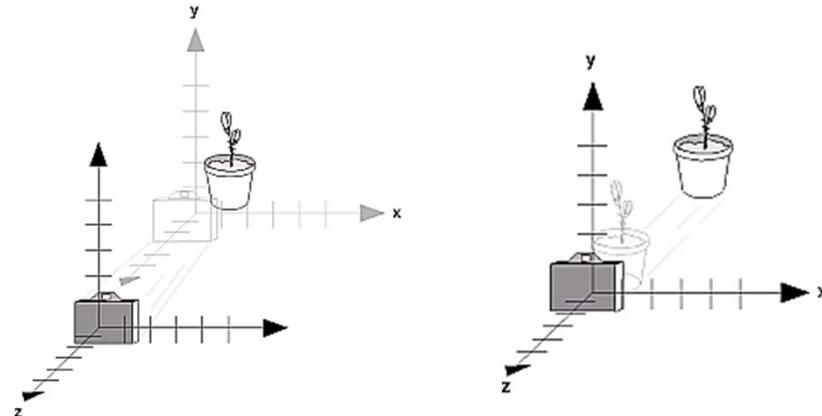
# OpenGL 3D Viewing Functions

## Method a. using `glTranslate*`() and `glRotate*`()

- To emulate the **viewpoint** movement in a desired way by translating the **objects**:



Object and Viewpoint: at the Origin.



Move the object away from the viewpoint by translating the object along “-z” direction:  
`glTranslatef(0.0, 0.0, -5.0);`

*(From: the red book)*

# OpenGL 3D Viewing Functions

Method b. using the **gluLookAt(eyex, eyey, eyez, atx, aty, atz, upx, upy, upz)** utility routine

- 3 sets of arguments
  - The **location of the viewpoint**,
  - A **reference point** where you look at
    - Some position in the center of a scene
  - **View-Up** direction

$$n = \frac{N}{|N|} = (n_x, n_y, n_z) \quad (z+)$$

$$u = \frac{V \times n}{|V|} = (u_x, u_y, u_z) \quad (x+) \quad (10-1)$$

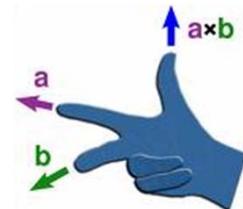
$$v = n \times u = (v_x, v_y, v_z) \quad (y+)$$

- The **default OpenGL viewing parameters** are:

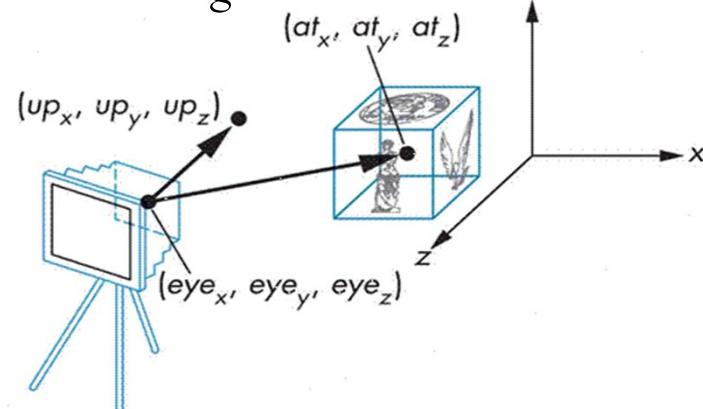
$$P_0 = (0, 0, 0), \quad P_{\text{ref}} = (0, 0, -1), \\ V = (0, 1, 0)$$

- The viewing reference frame defined by the viewing parameters

- Zview +:  $\mathbf{N} = \mathbf{at}-\mathbf{eye}$ ;
- Yview +:  $\mathbf{V} = \mathbf{up}$ ;
- Xview +:  $\mathbf{U} = \mathbf{V} \times \mathbf{N}$ .

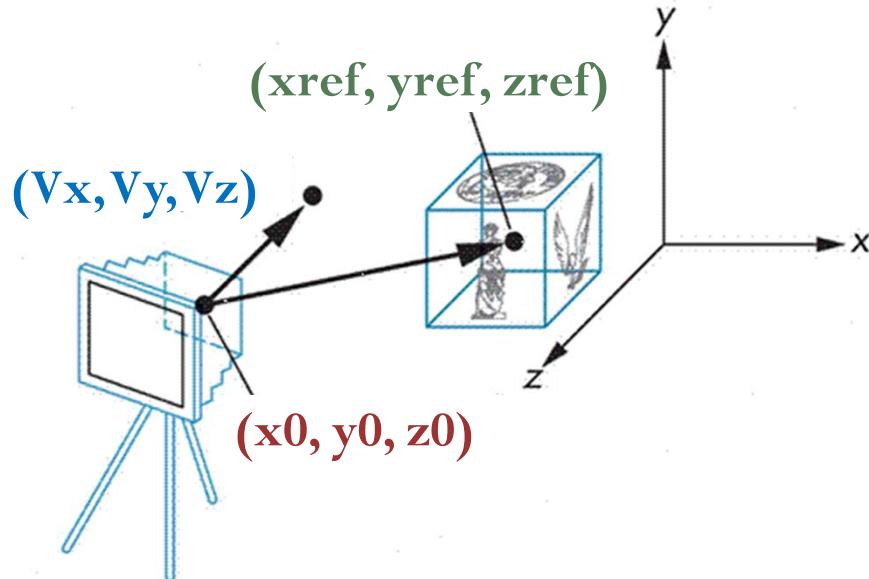


The unit axis vectors ( $uvn$ ) for the viewing reference frame:



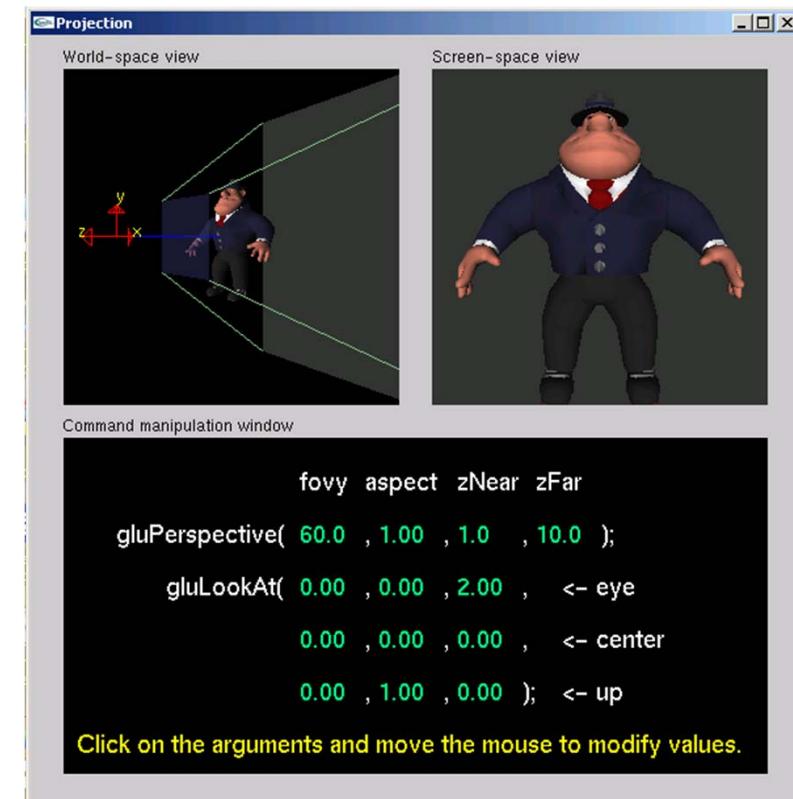
# OpenGL 3D Viewing Functions

gluLookAt ( x0, y0, z0, xref, yref, zref, Vx,Vy,Vz );



```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
gluLookAt(x0, y0, z0, xref, yref, zref,
          Vx, Vy, Vz);

//gluLookAt( 0.0, 0.0, 5.0, 0.0, 0.0,
//           0.0, 0.0, 1.0, 0.0);
```



The default OpenGL viewing parameters:  
 $P_0 = (0, 0, 0)$ ,  $P_{\text{ref}} = (0, 0, -1)$ ,  $V = (0, 1, 0)$

# Viewing Transformation: HowTo

**Method a.** Use one or more modeling transformation commands (that is, **glTranslate\***() and **glRotate\***()).

**Method b.** Use the Utility Library routine **gluLookAt()** to define a line of sight. This routine encapsulates a series of rotation and translation commands.

**Method c.** Create your own utility routine that encapsulates rotations and translations.

# OpenGL 3D Projection Functions

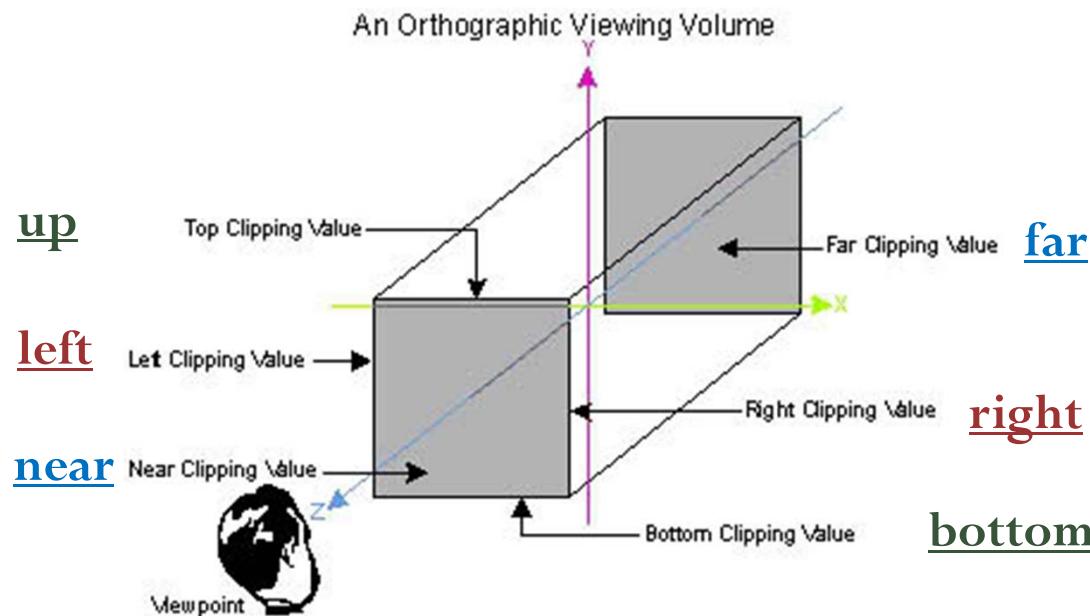
- The purpose of the **projection transformation** is to define a *viewing volume*, which is used in **two ways**
  - How an object is projected onto the screen;
  - Which objects or portions of objects are clipped out of the final image.
- Before issuing any of projection commands, you should call  
**`glMatrixMode(GL_PROJECTION);`**  
**`glLoadIdentity();`**  
so that the commands affect the **projection matrix** rather than the **modelview matrix**.

# OpenGL 3D Projection Functions

- OpenGL provides two functions
  - **glOrtho()**: to produce a orthographic (parallel) projection
  - **glFrustum()**: to produce a perspective projection (general)  
Both functions require **6** parameters to specify six clipping planes:  
*left*, *right*, *bottom*, *top*, *near* and *far* planes.
- GLU library for symmetric perspective-projection
  - **gluPerspective()**: with only **4** parameters

# OpenGL Orthogonal-Projection Function

**glOrtho ( left, right, bottom, up, near, far );**



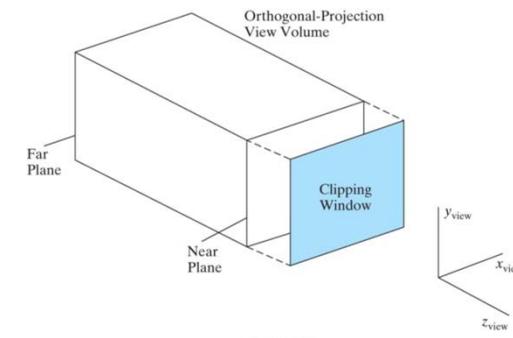
$$\begin{pmatrix} \frac{2}{right - left} & 0 & 0 & t_x \\ 0 & \frac{2}{top - bottom} & 0 & t_y \\ 0 & 0 & \frac{-2}{farVal - nearVal} & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

where

$$t_x = -\frac{right + left}{right - left}, \quad t_y = -\frac{top + bottom}{top - bottom}$$
$$t_z = -\frac{farVal + nearVal}{farVal - nearVal}$$

In OpenGL there is no option for the placement of the view plane:

**The near clipping plane = the view plane;**

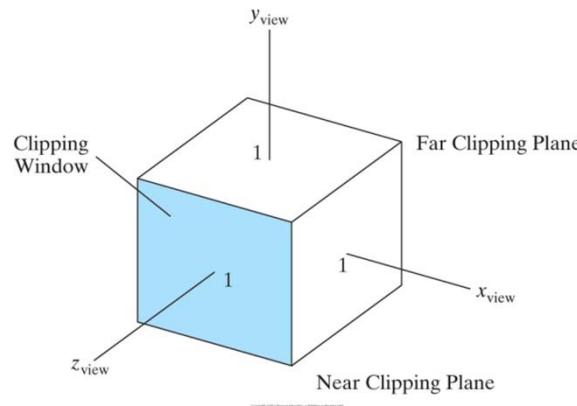


# OpenGL Orthogonal-Projection Function

**glOrtho ( left, right, bottom, up, near, far );**

- the **default** one:

```
glOrtho(-1.0, 1.0, -1.0, 1.0, -1.0, 1.0);
```



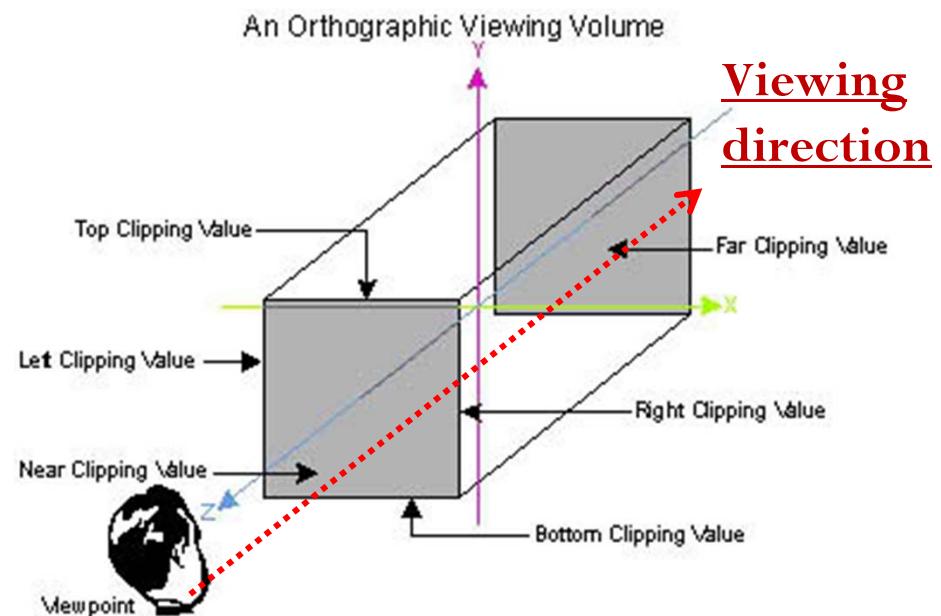
**FIGURE 10-47** Default orthogonal-projection view volume.

2D: gluOrtho2D(left, right, bottom, up);  
 $\Leftrightarrow$  a call to glOrtho() with near = -1.0 and far = 1.0.

glOrtho ( left, right, bottom, up, 0, 5.0 );

the far clipping plane:  $z_{far} = -5.0$ .

If **near** or **far** are **negative**, the plane is at the positive  $z_{view}$  axis (**behind** the viewing origin)



# OpenGL Perspective-Projection Functions

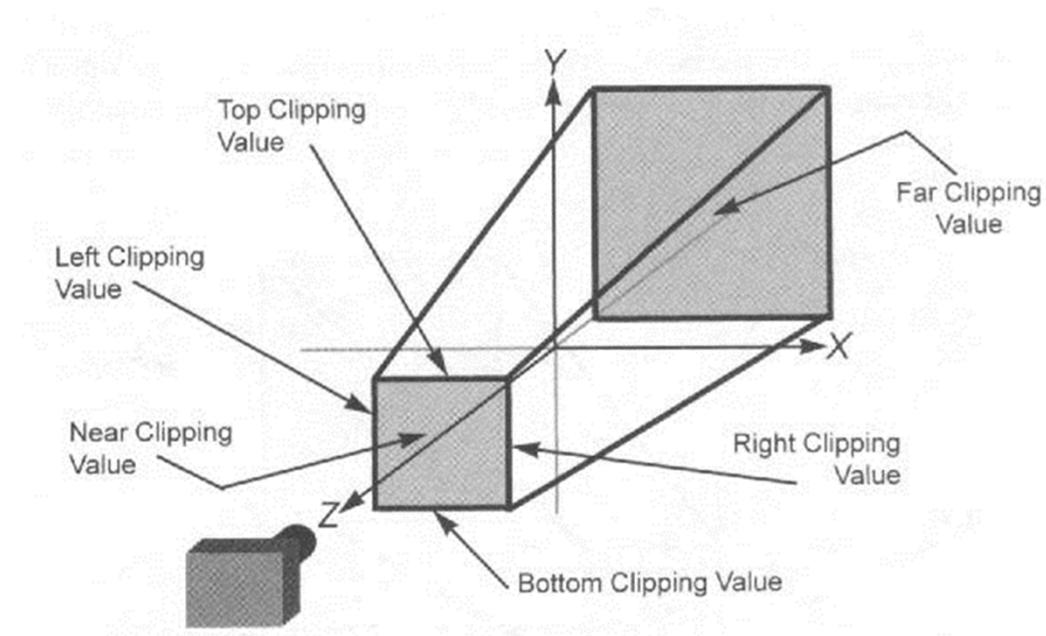
- General perspective-projection function

**glFrustum (left, right, bottom, up, near, far);**

left, right, bottom, up: set the size of the clipping window on the near plane.

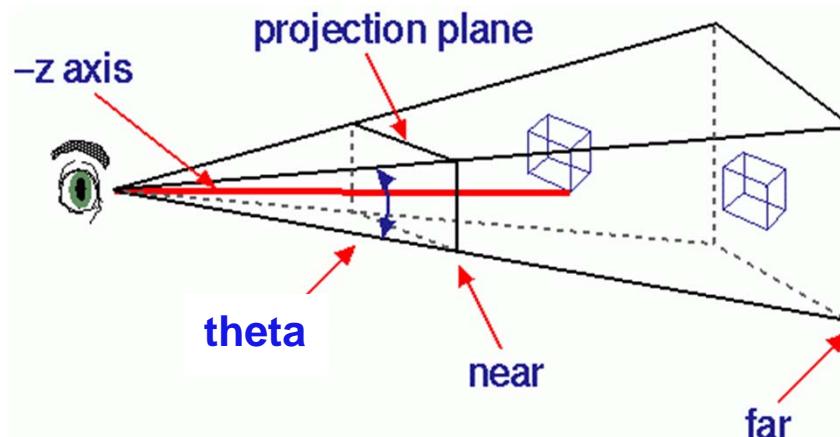
near, far: the **distances** from the origin to the near and far clipping planes along the  $-z_{\text{view}}$  axis. They **must be positive**. ( $z_{\text{near}} = -\text{near}$  and  $z_{\text{far}} = -\text{far}$ )

$$\begin{bmatrix} \frac{2 \cdot \text{nearVal}}{\text{right} - \text{left}} & 0 & A & 0 \\ 0 & \frac{2 \cdot \text{nearVal}}{\text{top} - \text{bottom}} & B & 0 \\ 0 & 0 & C & D \\ 0 & 0 & -1 & 0 \end{bmatrix}$$
$$A = \frac{\text{right} + \text{left}}{\text{right} - \text{left}} \quad B = \frac{\text{top} + \text{bottom}}{\text{top} - \text{bottom}}$$
$$C = -\frac{\text{farVal} + \text{nearVal}}{\text{farVal} - \text{nearVal}}$$
$$D = -\frac{2 \cdot \text{farVal} \cdot \text{nearVal}}{\text{farVal} - \text{nearVal}}$$



# OpenGL Perspective-Projection Functions

- Symmetric perspective-projection function  
**gluPerspective( theta, aspect, near, far );**  
theta: the **field-of-view angle** between the top and bottom clipping planes in the range [  $0^\circ$  ,  $180^\circ$  ].  
aspect: the **aspect ratio (width/height)** of the clipping window.  
near, far: specify the distances from the view point (coordinate origin) to the near and far clipping planes.  
Both **near** and **far** must be **positive** values.  
 $\mathbf{z_{near}}$  = -near and  $\mathbf{z_{far}}$  = -far refer to the positions of the near and far planes.



# OpenGL 3D Viewing Program Example

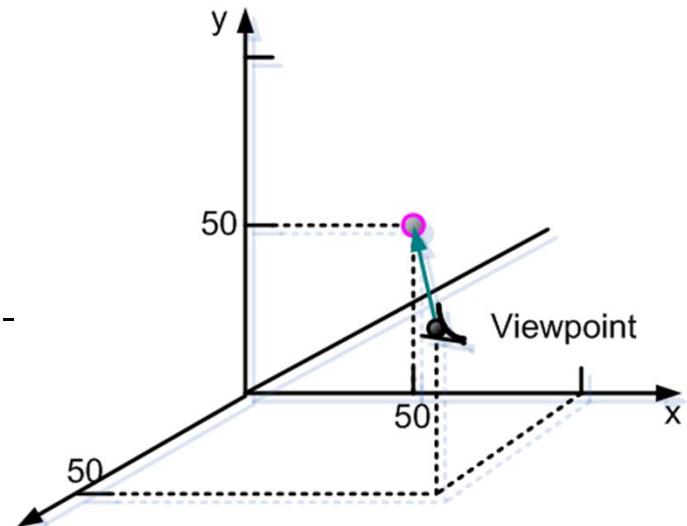
```
#include <GL/glut.h>
GLint winWidth=600, winHeight=600;      // Initial display-window size.
GLfloat x0=100.0, y0=50.0, z0=50.0;    // Viewing-coordinate origin P0.
GLfloat xref=50.0, yref=50.0, zref=0.0; // Look-at point Pref;
GLfloat Vx=0.0, Vy=1.0, Vz=0.0;        // View-up vector
/*positive zview axis N = P0 - Pref = (50.0, 0.0, 50.0)

/* Set coordinate limits for the clipping window: */
GLfloat xwMin = -40.0, xwMax= 40.0, ywMin = -60.0, ywMax= 60.0;

/* Set positions for near and far clipping planes: */
GLfloat dnear=25.0, dfar=125.0;
```



**FIGURE 10-48** Output display generated by the three-dimensional viewing example program.



# OpenGL 3D Viewing Program Example

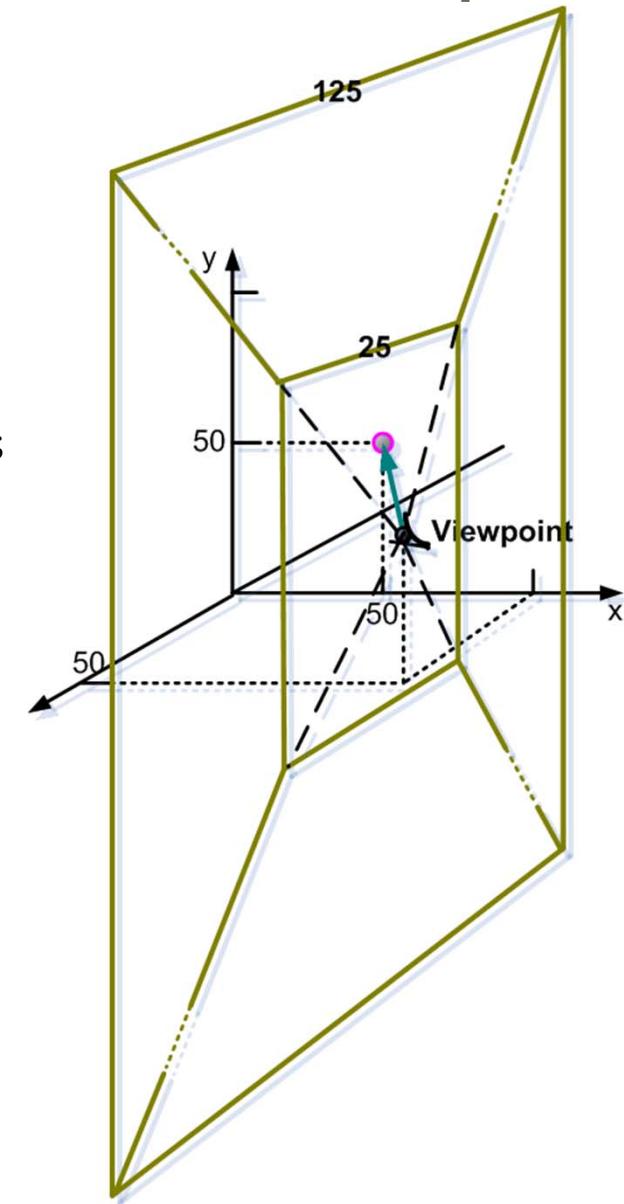
```
void init()
{
    glClearColor(1.0, 1.0, 1.0, 0.0);

    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glFrustum(xwMin, xwMax, ywMin, ywMax, dnear, dfar);

    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    gluLookAt(x0, y0, z0, xref, yref, zref, Vx, Vy, Vz);
}

void reshapeFcn(GLsizei newWidth, GLsizei newHeight)
{
    glViewport(0, 0, newWidth, newHeight);
    winWidth=newWidth;
    winHeight=newHeight;
}
```

50



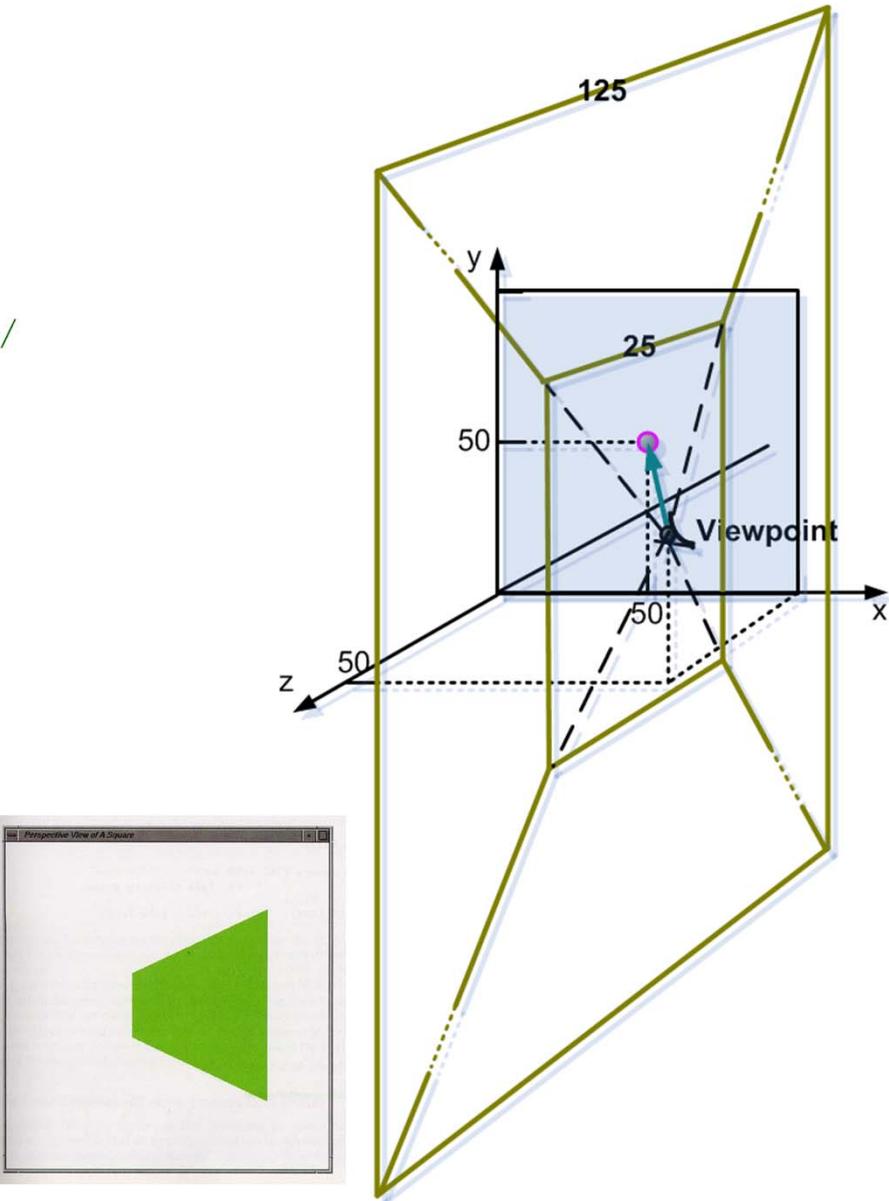
# OpenGL 3D Viewing Program Example

```
void displayFcn(void)
{
    glClear(GL_COLOR_BUFFER_BIT |
            GL_DEPTH_BUFFER_BIT);

    /* Set parameters for a square fill area. */
    // Set fill color to green.
    glColor3f(0.0, 1.0, 0.0);
    glPolygonMode(GL_FRONT, GL_FILL);
    // Wire-frame back face.
    glPolygonMode(GL_BACK, GL_LINE);
    glBegin(GL_QUADS);
    glVertex3f(0.0, 0.0, 0.0);

    glVertex3f(100.0, 0.0, 0.0);
    glVertex3f(100.0, 100.0, 0.0);
    glVertex3f(0.0, 100.0, 0.0);
    glEnd();
    glFlush();
}
```

Foreshortening effect  
Square -> trapezoid



# Summary

- 3D viewing pipeline and camera analogy
- 3D viewing transformation and projected transformation
  - viewing coordinates (eye)
  - Orthogonal projection
  - Perspective projection
- OpenGL and utility functions
  - `glMatrixMode();`
    - `GL_MODELVIEW/GL_PROJECTION`
  - `gluLookAt();`
  - `glOrtho();`
  - `glFrustum();`
  - `gluPerspective();`



# RCS-603: COMPUTER GRAPHICS

## UNIT-III

Presented By :

**Dr. Vinod Jain (Associate Professor, GLBITM)**



**GL BAJAJ**  
Institute of Technology & Management

[Approved by AICTE, Govt. of India & Affiliated to Dr. APJ  
Abdul Kalam Technical University, Lucknow, U.P. India]

**Department Of Computer Science & Engineering**



# Topics Left

- 3d Object Representation (chapter 10, 10.1)
- 3-D Geometric Primitives, (9.1,9.2)
- 3-D viewing, projections, 3-D Clipping. (ch 12)

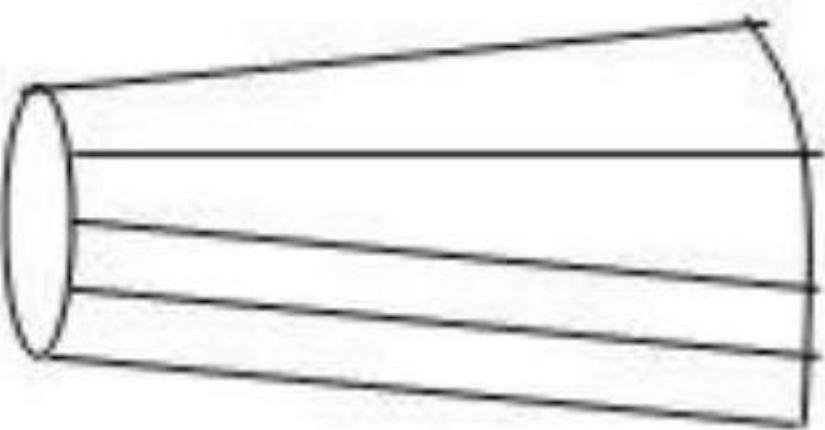


# 3d Object Representation

3D object representation is divided into two categories.

- **Boundary Representations (B-reps) –**
- It describes a 3D object as a **set of surfaces** that separates the object interior from the environment.
- **Space–partitioning representations –**
- It is used to describe interior properties, by partitioning the spatial region containing an object into a **set of small, non-overlapping, contiguous solids (usually cubes)**.

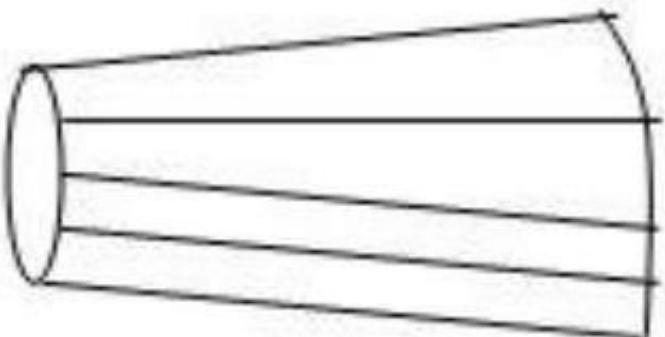
# 3d Object Representation : Polygon Surfaces



A 3D object represented by polygons

# 3d Object Representation : Polygon Surfaces

- The most commonly used representation for a 3D graphics object.
- It is a set of surface polygons that enclose the object interior.
- Set of polygons are stored for object description.
- This simplifies and speeds up the surface rendering and display of object since all surfaces can be described with linear equations.



A 3D object represented by polygons

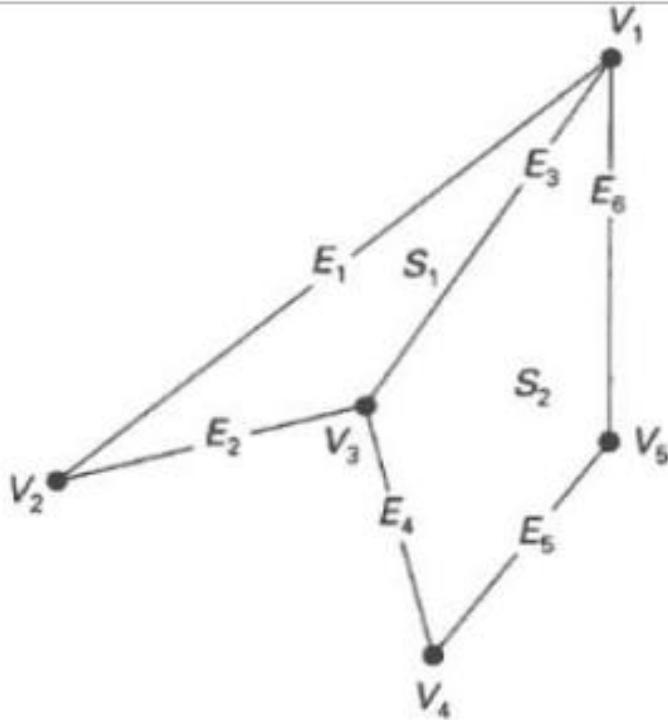


# 3d Object Representation : Polygon Surfaces

Three ways to represent polygon surfaces

1. Polygon Tables
2. Plane Equations
3. Polygon Meshes

# Polygon Tables



VERTEX TABLE	
$V_1:$	$x_1, y_1, z_1$
$V_2:$	$x_2, y_2, z_2$
$V_3:$	$x_3, y_3, z_3$
$V_4:$	$x_4, y_4, z_4$
$V_5:$	$x_5, y_5, z_5$

EDGE TABLE	
$E_1:$	$V_1, V_2$
$E_2:$	$V_2, V_3$
$E_3:$	$V_3, V_1$
$E_4:$	$V_3, V_4$
$E_5:$	$V_4, V_5$
$E_6:$	$V_5, V_1$

POLYGON-SURFACE TABLE	
$S_1:$	$E_1, E_2, E_3$
$S_2:$	$E_3, E_4, E_5, E_6$



# Polygon Tables

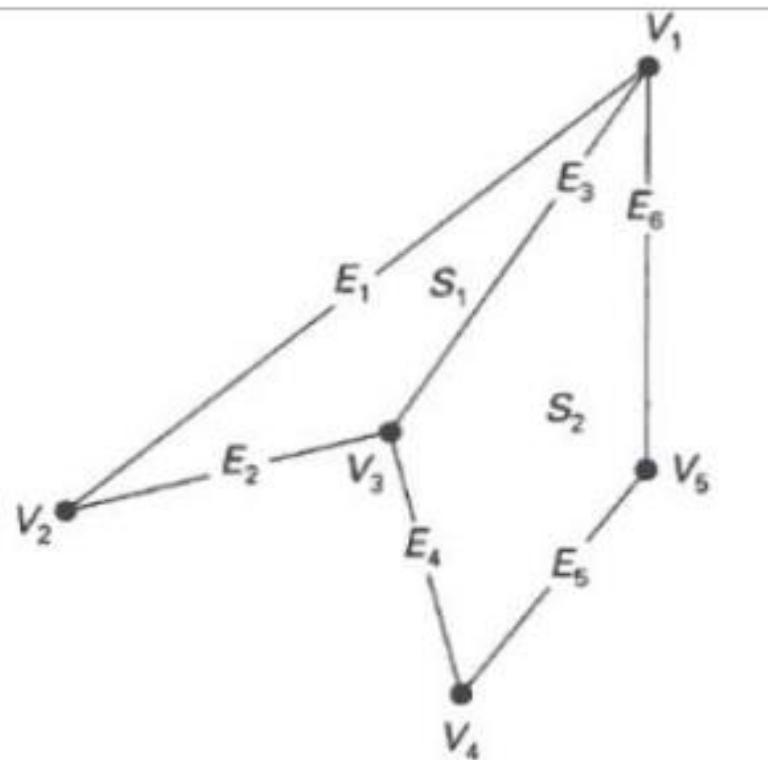
The object is stored by using three tables

1. Vertex Table
2. Edge table
3. Polygon-Surface table

# Polygon Tables- Vertex Table

## Vertex Table

It stores x, y, and z coordinate information of all the vertices as  $v_1: x_1, y_1, z_1$ .

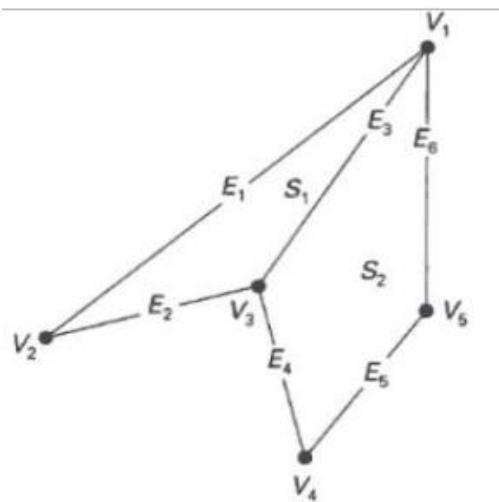


VERTEX TABLE		
$V_1:$	$x_1, y_1, z_1$	
$V_2:$	$x_2, y_2, z_2$	
$V_3:$	$x_3, y_3, z_3$	
$V_4:$	$x_4, y_4, z_4$	
$V_5:$	$x_5, y_5, z_5$	

# Polygon Tables - Edge table

## Edge table

- The Edge table is used to store the edge information of polygon.
- In the following figure, edge  $E_1$  lies between vertex  $v_1$  and  $v_2$  which is represented in the table as  $E_1: v_1, v_2$ .



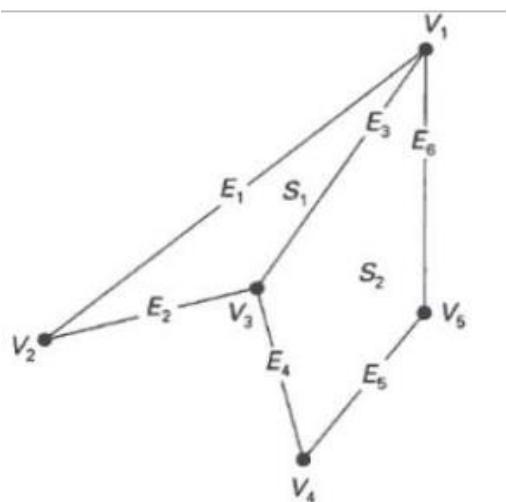
EDGE TABLE	
$E_1:$	$v_1, v_2$
$E_2:$	$v_2, v_3$
$E_3:$	$v_3, v_1$
$E_4:$	$v_3, v_4$
$E_5:$	$v_4, v_5$
$E_6:$	$v_5, v_1$

# Polygon Tables - Polygon-Surface table

## Polygon-Surface table

Polygon surface table stores the number of surfaces present in the polygon.

From the following figure, surface  $S_1$  is covered by edges  $E_1$ ,  $E_2$  and  $E_3$  which can be represented in the polygon surface table as  $S_1: E_1, E_2$ , and  $E_3$



POLYGON-SURFACE TABLE	
$S_1:$	$E_1, E_2, E_3$
$S_2:$	$E_3, E_4, E_5, E_6$



# Plane Equations

- The equation for plane surface can be expressed as –

$$Ax + By + Cz + D = 0$$

- Where  $(x, y, z)$  is any point on the plane, and the coefficients A, B, C, and D are constants describing the spatial properties of the plane.
- We can obtain the values of A, B, C, and D by solving a set of three plane equations using the coordinate values for three non collinear points in the plane. Let us assume that three vertices of the plane are  $(x_1, y_1, z_1)$ ,  $(x_2, y_2, z_2)$  and  $(x_3, y_3, z_3)$ .



# Plane Equations

- The equation for plane surface can be expressed as –

$$Ax + By + Cz + D = 0$$

- We can obtain the values of A, B, C, and D by solving a set of three plane equations.
- Let us assume that three vertices of the plane are  $(x_1, y_1, z_1)$ ,  $(x_2, y_2, z_2)$  and  $(x_3, y_3, z_3)$ .



# Plane Equations

- Let us solve the following simultaneous equations for ratios A/D, B/D, and C/D. You get the values of A, B, C, and D.

$$(A/D) x_1 + (B/D) y_1 + (C/D) z_1 = -1$$

$$(A/D) x_2 + (B/D) y_2 + (C/D) z_2 = -1$$

$$(A/D) x_3 + (B/D) y_3 + (C/D) z_3 = -1$$

# Plane Equations

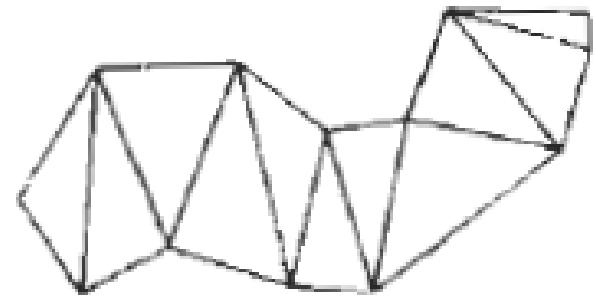
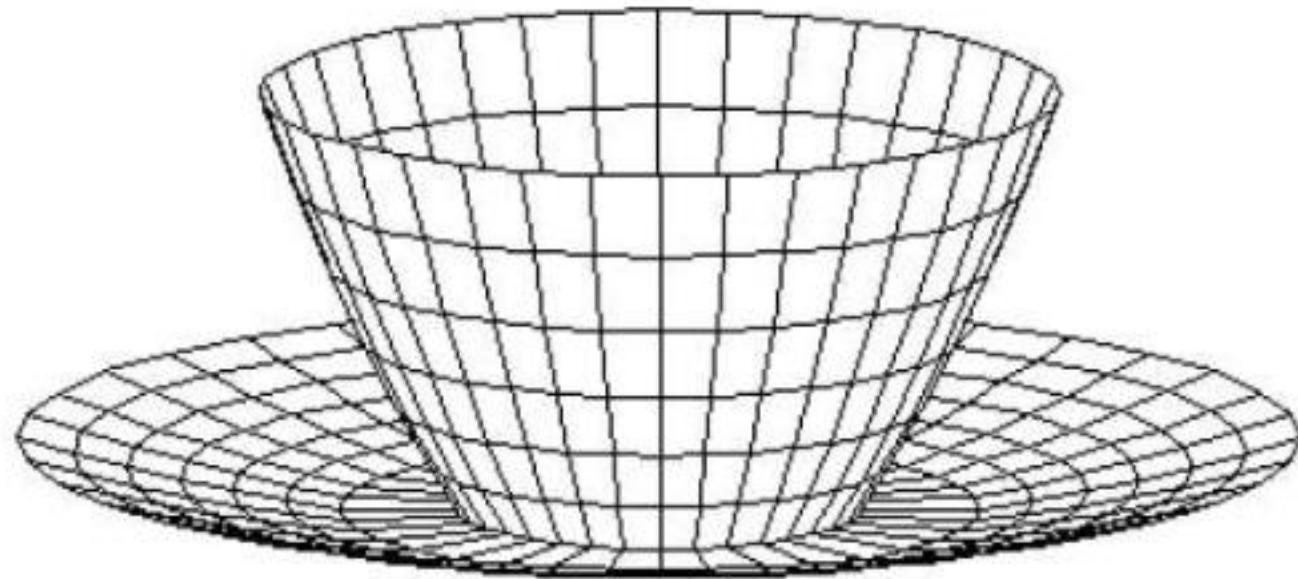
To obtain the above equations in determinant form, apply Cramer's rule to the above equations.

$$A = \begin{bmatrix} 1 & y_1 & z_1 \\ 1 & y_2 & z_2 \\ 1 & y_3 & z_3 \end{bmatrix} B = \begin{bmatrix} x_1 & 1 & z_1 \\ x_2 & 1 & z_2 \\ x_3 & 1 & z_3 \end{bmatrix} C = \begin{bmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{bmatrix} D =$$
$$- \begin{bmatrix} x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ x_3 & y_3 & z_3 \end{bmatrix}$$

For any point  $(x, y, z)$  with parameters  $A, B, C$ , and  $D$ , we can say that –

- $Ax + By + Cz + D \neq 0$  means the point is not on the plane.
- $Ax + By + Cz + D < 0$  means the point is inside the surface.
- $Ax + By + Cz + D > 0$  means the point is outside the surface.

# Plane Equations

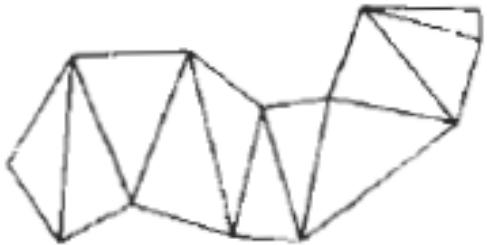


# Polygon Meshes

3D surfaces and solids can be approximated by a set of polygonal and line elements. Such surfaces are called **polygonal meshes**.

In polygon mesh, each edge is shared by at most two polygons.

The set of polygons or faces, together form the “skin” of the object.



---

*Figure 10-6*  
A triangle strip formed with  
11 triangles connecting 13  
vertices.



# Polygon Meshes

- **Advantages**

- It can be used to model almost any object.
- They are easy to represent as a collection of vertices.
- They are easy to transform.
- They are easy to draw on computer screen.

- **Disadvantages**

- Curved surfaces can only be approximately described.
- It is difficult to simulate some type of objects like hair or liquid.



# Three Dimensional Concepts

## Three Dimensional Display Methods:

- Parallel Projection
- Perspective Projection
- Depth Queing
- Visible Line and Surface Identification
- Surface Rendering
- Exploded and Cutaway Views
- Three-Dimensional and Stereoscopic Views

# Three Dimensional Concepts

## Three Dimensional Display Methods:

- To obtain a display of a three dimensional scene that has been modeled in world coordinates, we must setup a co-ordinate reference for the ‘camera’.

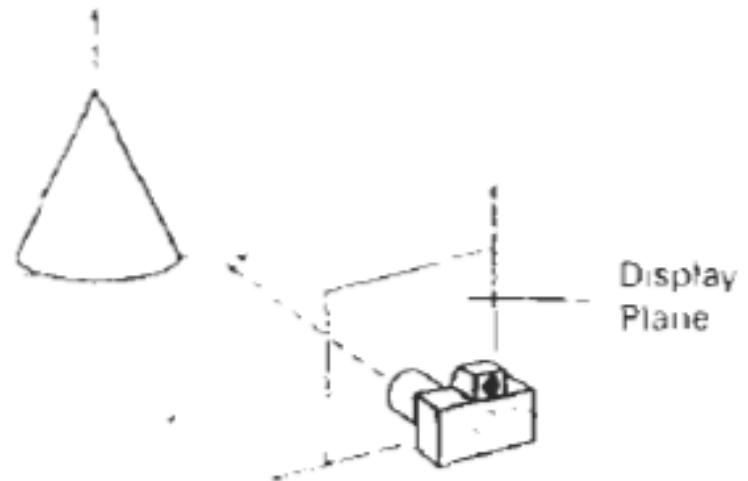
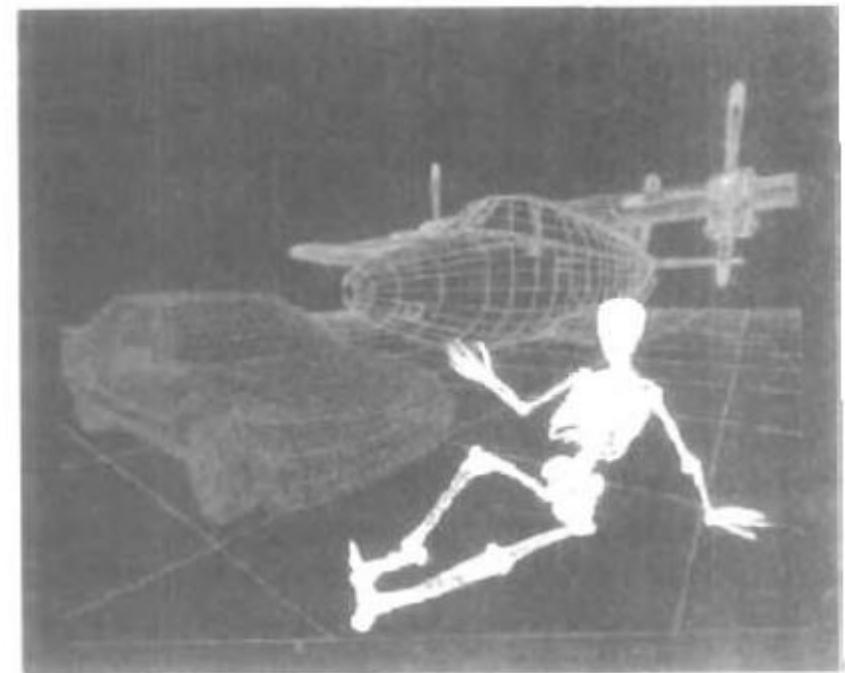


Figure 9.1  
Coordinate reference for obtaining  
a particular view of a  
three-dimensional scene.

# Three Dimensional Concepts

## Three Dimensional Display Methods:

- The objects can be displayed in wire frame form, or we can apply lighting and surface rendering techniques to shade the visible surfaces.



# Parallel Projection

- Parallel projection is a method for generating a view of a solid object. This is to **project points on the object surface along parallel lines** onto the display plane.
- This technique is used in engineering and architectural drawings to **represent an object with a set of views** that maintain relative proportions of the object.

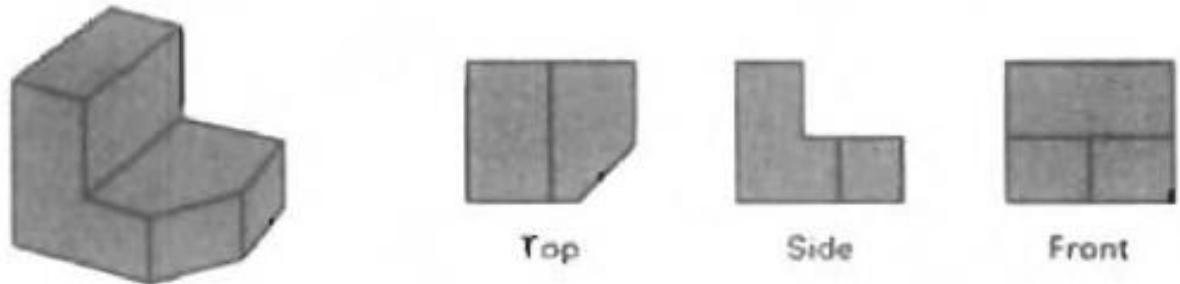
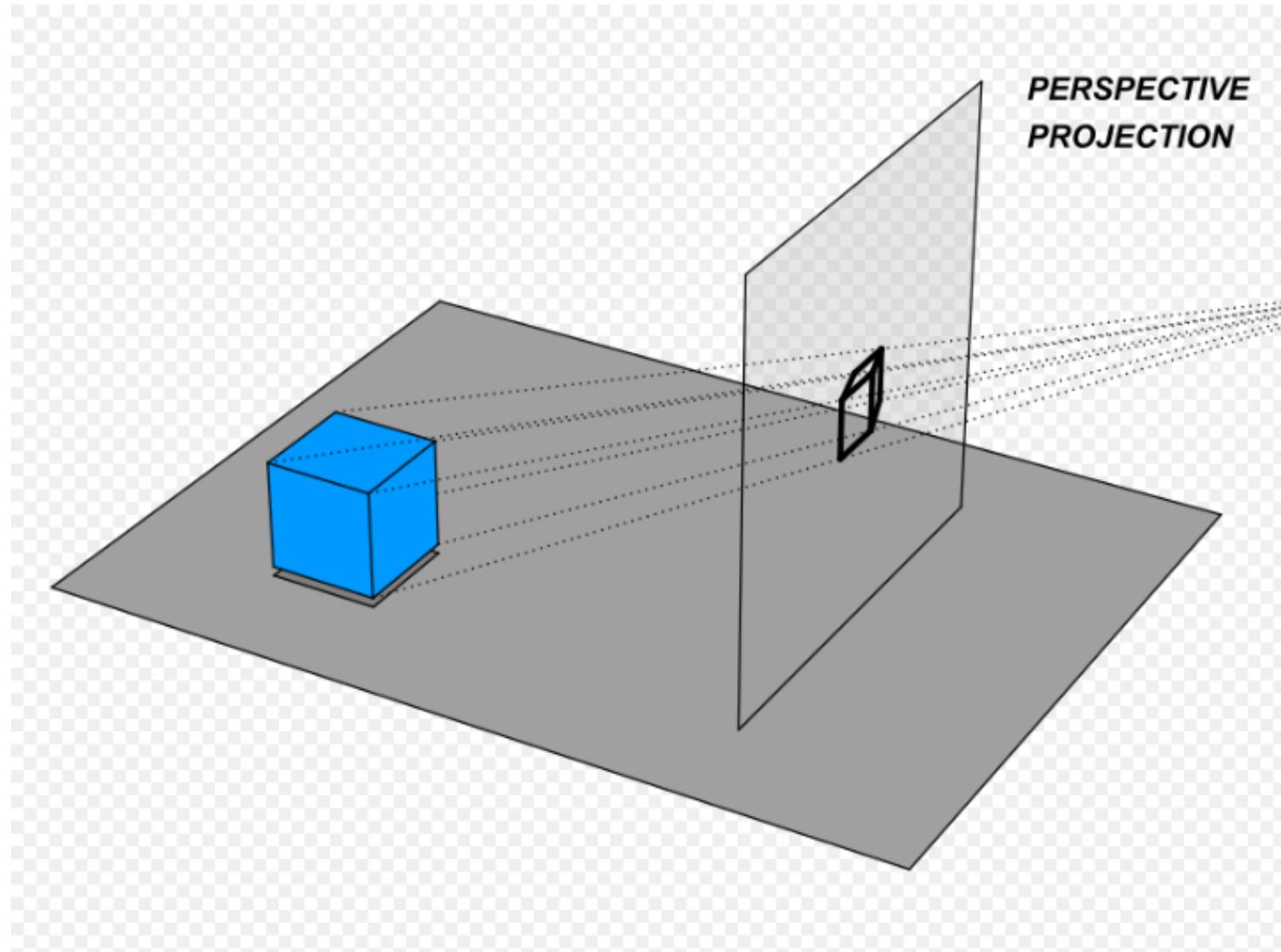


Figure 9-3

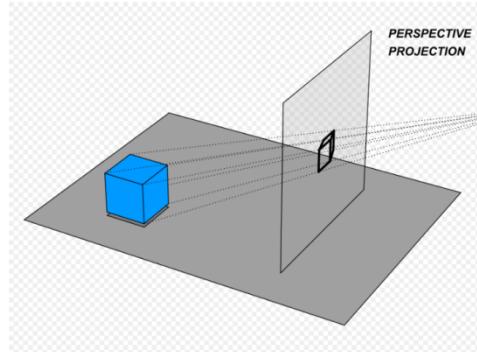
Three parallel-projection views of an object, showing relative proportions from different viewing positions.

# Perspective Projection



# Perspective Projection

- It is a method for generating a view of a three dimensional scene is to **project points to the display plane along converging paths**.
- In a perspective projection, parallel lines in a scene that are not **parallel to the display plane are projected into converging lines**.
- Scenes displayed using perspective **projections appear more realistic**, since this is the way that our eyes and a camera lens form images.

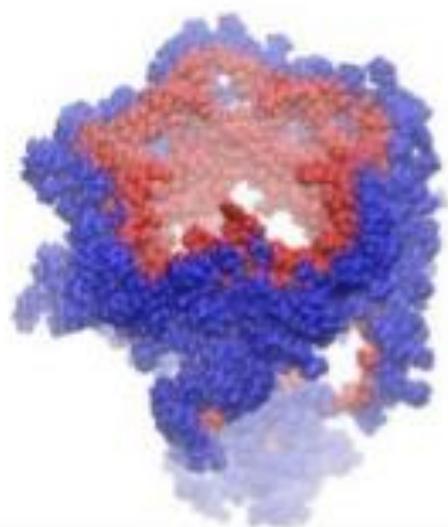


# Depth Cueing:

## Depth Cueing

---

- Hidden surfaces are not removed but displayed with different effects such as intensity, color, or shadow for giving hint for third dimension of the object.
- Simplest solution: use different colors-intensities based on the dimensions of the shapes.





# Depth Cueing:

- Depth cueing is a method for indicating depth with wire frame displays by varying the intensity of objects according to their distance from the viewing position.
- Depth cueing is applied by choosing maximum and minimum intensity (or color) values and a range of distance over which the intensities are to vary.



# Visible Line and Surface Identification

- A simplest way to identify the visible line is to **highlight the visible lines or to display them in a different color.**
- Another method is to display the non visible lines as dashed lines.

# Visible Line and Surface Identification

- The wireframe representation of the pyramid in
- (a) contains no depth information to indicate whether the viewing direction is
- (b) downward from a position above the apex or
- (c) upward from a position below the base.



(a)

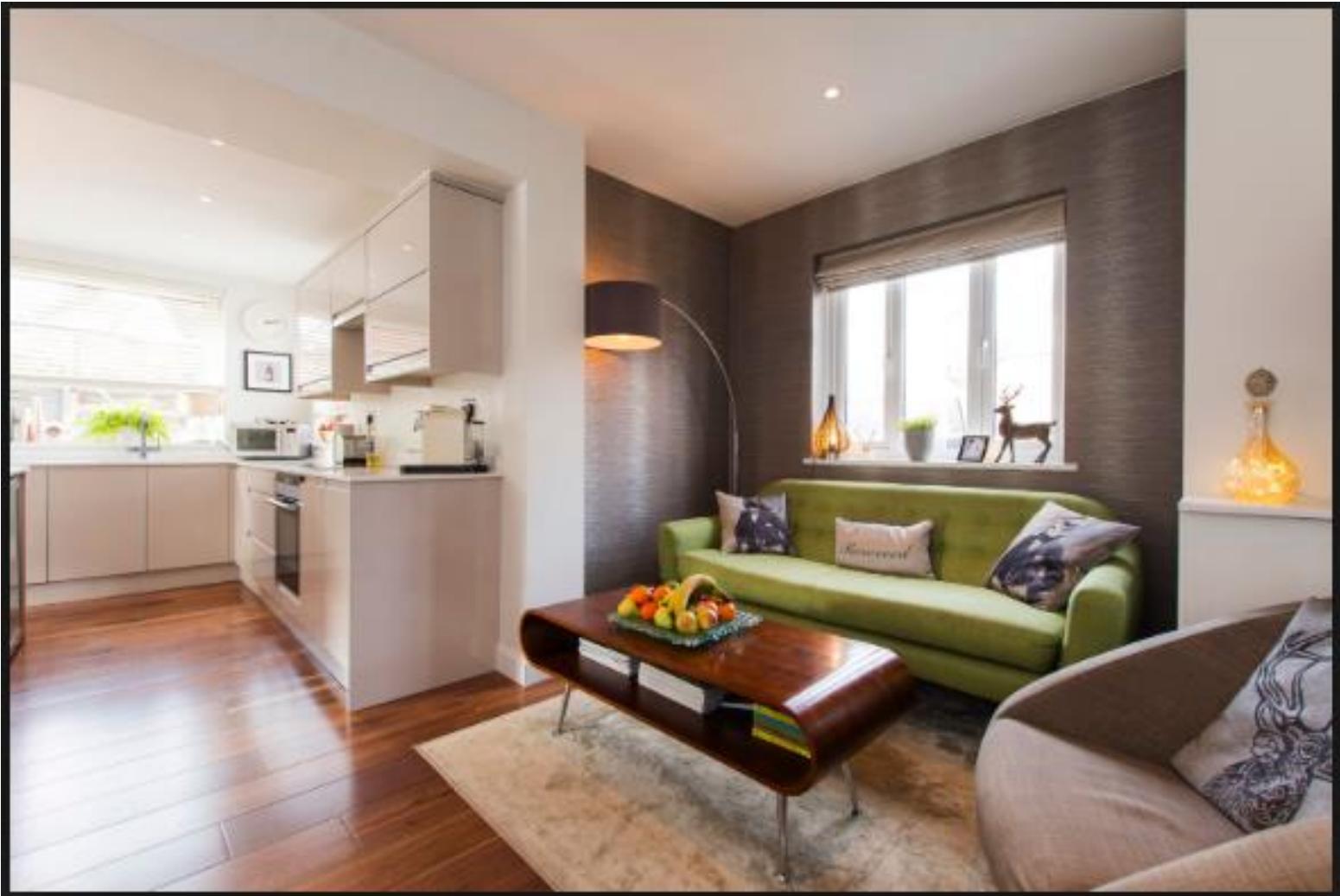


(b)



(c)

# Surface Rendering



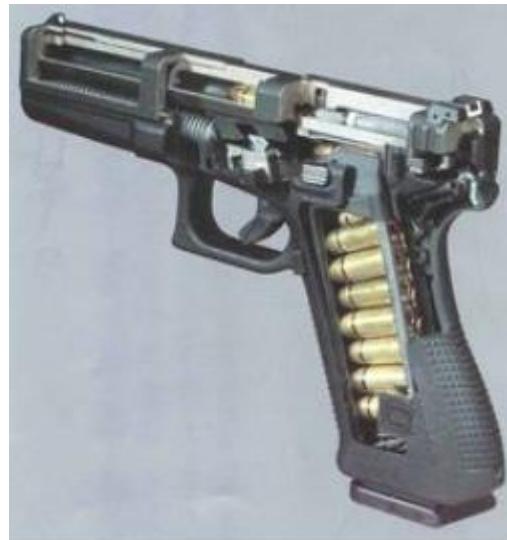


# Surface Rendering

- Surface rendering method is used to generate a degree of realism in a displayed scene.
- Realism is attained in displays by setting the surface intensity of objects according to the lighting conditions in the scene and surface characteristics.
- Lighting conditions include the intensity and positions of light sources and the background illumination.
- Surface characteristics include degree of transparency and how rough or smooth the surfaces are to be.

# Exploded and Cutaway Views

- Exploded and cutaway views of objects can be used to show the internal structure and relationship of the object's parts.
- An alternative to exploding an object into its component parts is the cut away view which removes part of the visible surfaces to show internal structure.



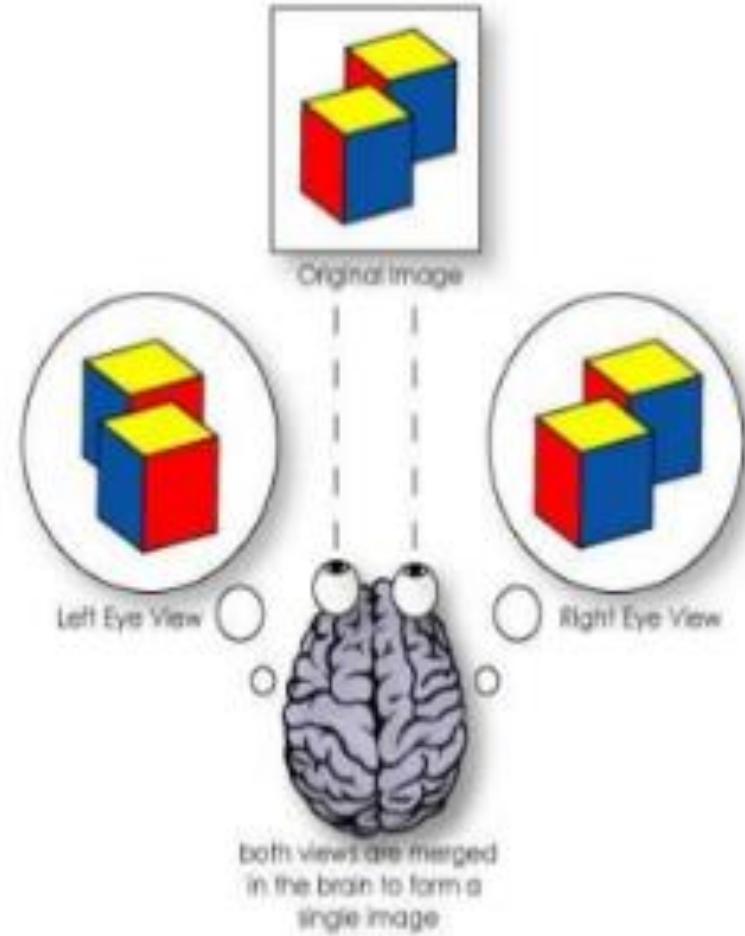


# Three-Dimensional and Stereoscopic Views

- In Stereoscopic views, three dimensional views can be obtained by reflecting a raster image from a vibrating flexible mirror.
- The vibrations of the mirror are synchronized with the display of the scene on the CRT.
- Stereoscopic devices present two views of a scene; one for the left eye and the other for the right eye.

# Three-Dimensional and Stereoscopic Views

- Stereoscopic devices present two views of a scene;
- one for the left eye
- and the other for the right eye.





# 3-D Transformation

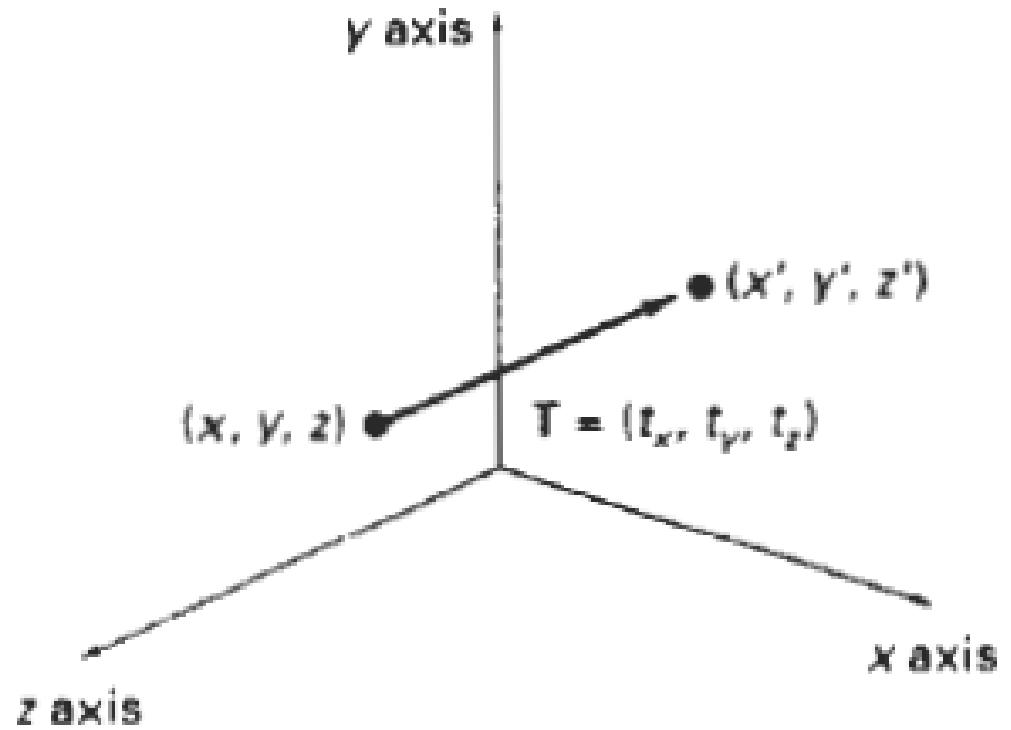
- Methods for geometric transformations are extended from two-dimensional methods by including considerations for the  $z$  coordinate.
- We will discuss following transformations in 3D
  1. Translation
  2. Rotation
  3. Scaling
  4. Reflection
  5. Shear



# Unit- III

1. 3-D Geometric Primitives
2. 3-D Object representation
- 3. 3-D Transformation**
4. 3-D viewing
5. projections
6. 3-D Clipping.

# 3-D Translation



---

*Figure 11-1*  
Translating a point with translation  
vector  $\mathbf{T} = (t_x, t_y, t_z)$ .

# 3-D Translation by translation vector

$$T = (t_x, t_y, t_z)$$

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

$$\mathbf{P}' = \mathbf{T} \cdot \mathbf{P}$$

$$x' = x + t_x, \quad y' = y + t_y, \quad z' = z + t_z$$



# 3-D Rotation

To generate a rotation transformation for an object, we must designate

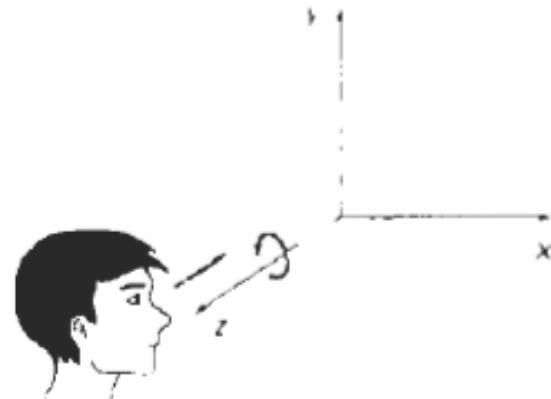
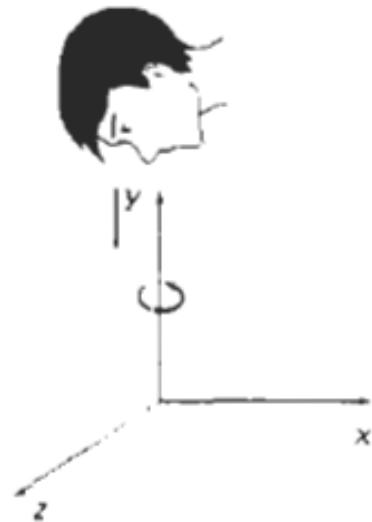
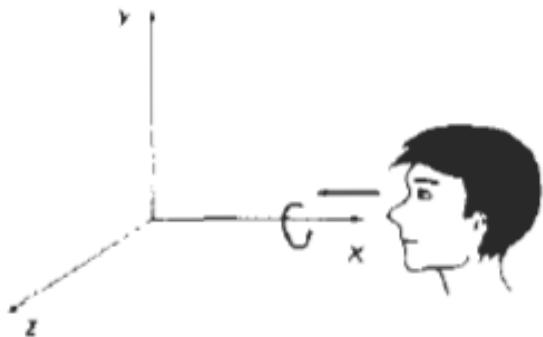
1. An axis of rotation (about which the object is to be rotated)
2. The amount of angular rotation.



# 3-D Rotation

- In two-dimensional applications, where all transformations are carried out in the xy plane
- A three-dimensional rotation can be specified around any line in space.
- The easiest rotation axes to handle are those that are parallel to the coordinate axes.

# 3-D Rotation – three rotation axis are



3-D Rotation : In homogeneous coordinates  
z-axis rotation equations are expressed as

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos \theta & -\sin \theta & 0 & 0 \\ \sin \theta & \cos \theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

$$x' = x \cos \theta - y \sin \theta$$

$$y' = x \sin \theta + y \cos \theta$$

$$z' = z$$

# 3-D Rotation : In homogeneous coordinates z-axis rotation equations are expressed as

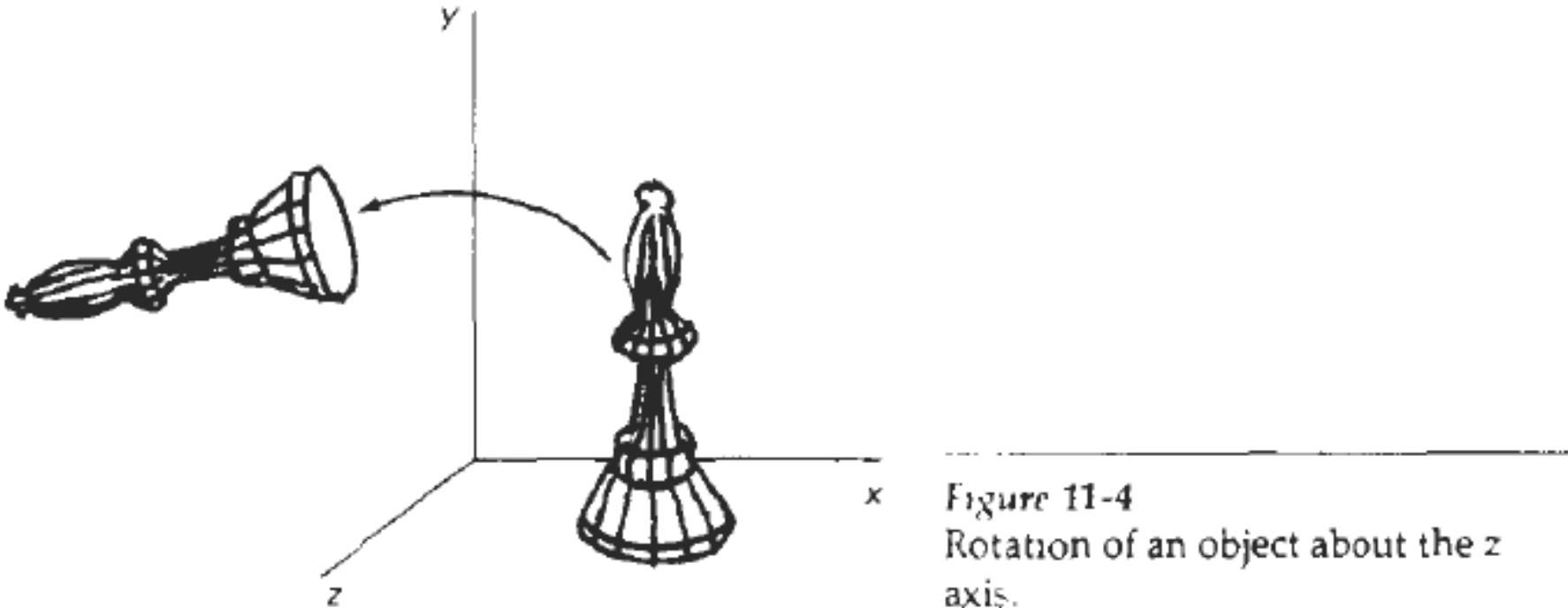


Figure 11-4  
Rotation of an object about the z axis.

which we can write more compactly as

$$\mathbf{P}' = \mathbf{R}_z(\theta) \cdot \mathbf{P}$$

# 3-D Rotation : In homogeneous coordinates x-axis rotation equations are expressed as

Substituting permutations 11-7 in Eqs. 11-4, we get the equations for an x-axis rotation:

$$\begin{aligned}y' &= y \cos \theta - z \sin \theta \\z' &= y \sin \theta + z \cos \theta \\x' &= x\end{aligned}\tag{11-8}$$

which can be written in the homogeneous coordinate form

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \theta & -\sin \theta & 0 \\ 0 & \sin \theta & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}\tag{11-9}$$

$$\mathbf{P}' = \mathbf{R}_x(\theta) \cdot \mathbf{P}$$

3-D Rotation : In homogeneous coordinates  
**y-axis rotation equations are expressed as**

$$z' = z \cos \theta - x \sin \theta$$

$$x' = z \sin \theta + x \cos \theta$$

$$y' = y$$

The matrix representation for y-axis rotation is

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos \theta & 0 & \sin \theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin \theta & 0 & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

or

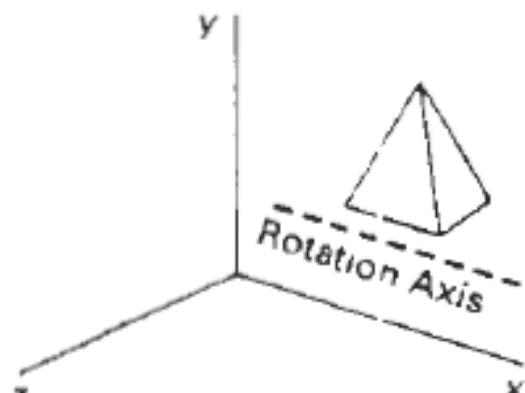
$$\mathbf{P}' = \mathbf{R}_y(\theta) \cdot \mathbf{P}$$



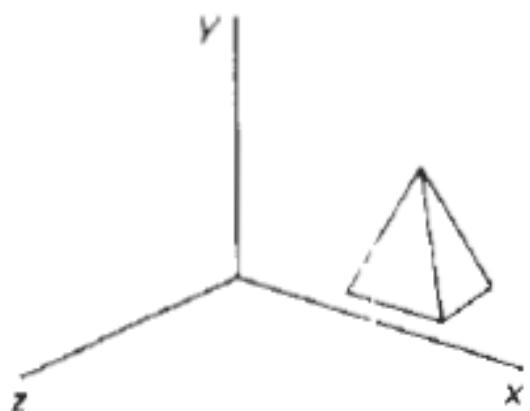
# General Three-Dimensional Rotations (About any given axis)

## Three Steps

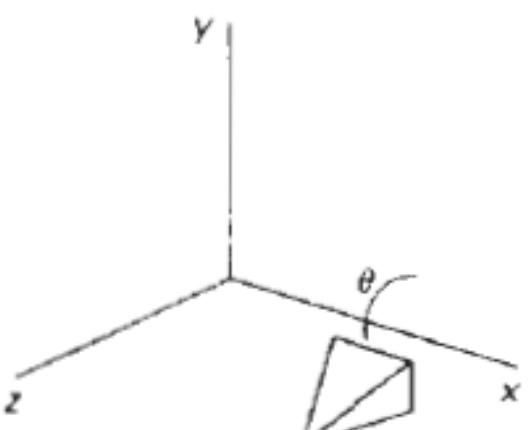
1. Translate the object so that the rotation axis coincides with the parallel coordinate axis.
2. Perform the specified rotation about that axis.
3. Translate the object so that the rotation axis is moved back to its original position.



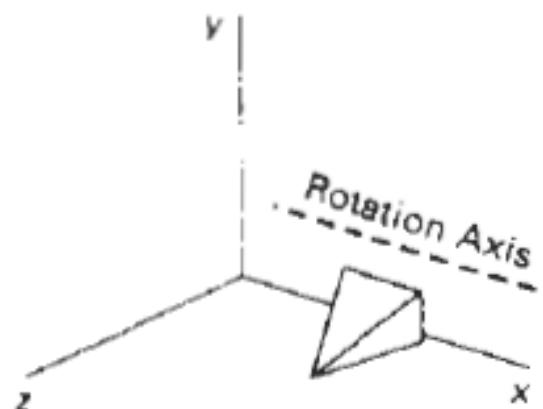
(a)  
Original Position of Object



(b)  
Translate Rotation Axis onto  $x$  Axis



(c)  
Rotate Object Through Angle  $\theta$



(d)  
Translate Rotation  
Axis to Original Position



# General Three-Dimensional Rotations (About any given axis)

The steps in this sequence are illustrated in Fig. 11-8. Any coordinate position  $\mathbf{P}$  on the object in this figure is transformed with the sequence shown as

$$\mathbf{P}' = \mathbf{T}^{-1} \cdot \mathbf{R}_z(\theta) \cdot \mathbf{T} \cdot \mathbf{P}$$

where the composite matrix for the transformation is

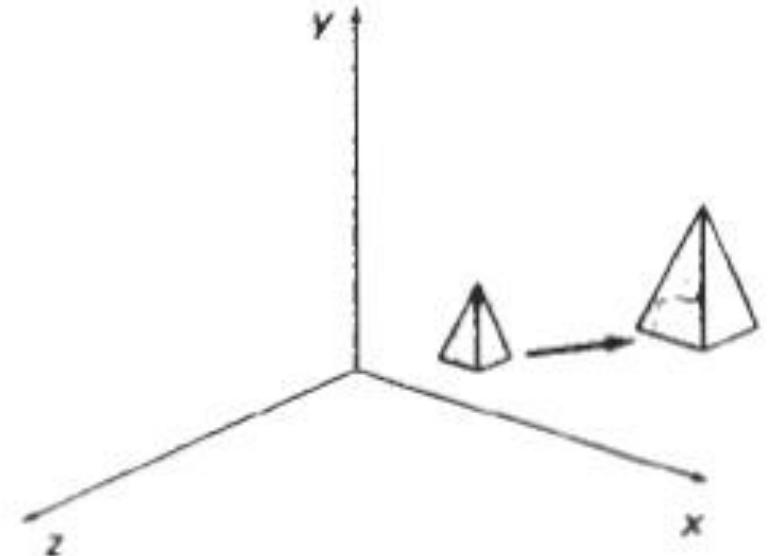
$$\mathbf{R}(\theta) = \mathbf{T}^{-1} \cdot \mathbf{R}_z(\theta) \cdot \mathbf{T}$$

# 3 D SCALING

- The matrix expression for the scaling transformation of a position  $P = (x, y, z)$  relative to the coordinate origin can be written as

$$P' = S \cdot P$$

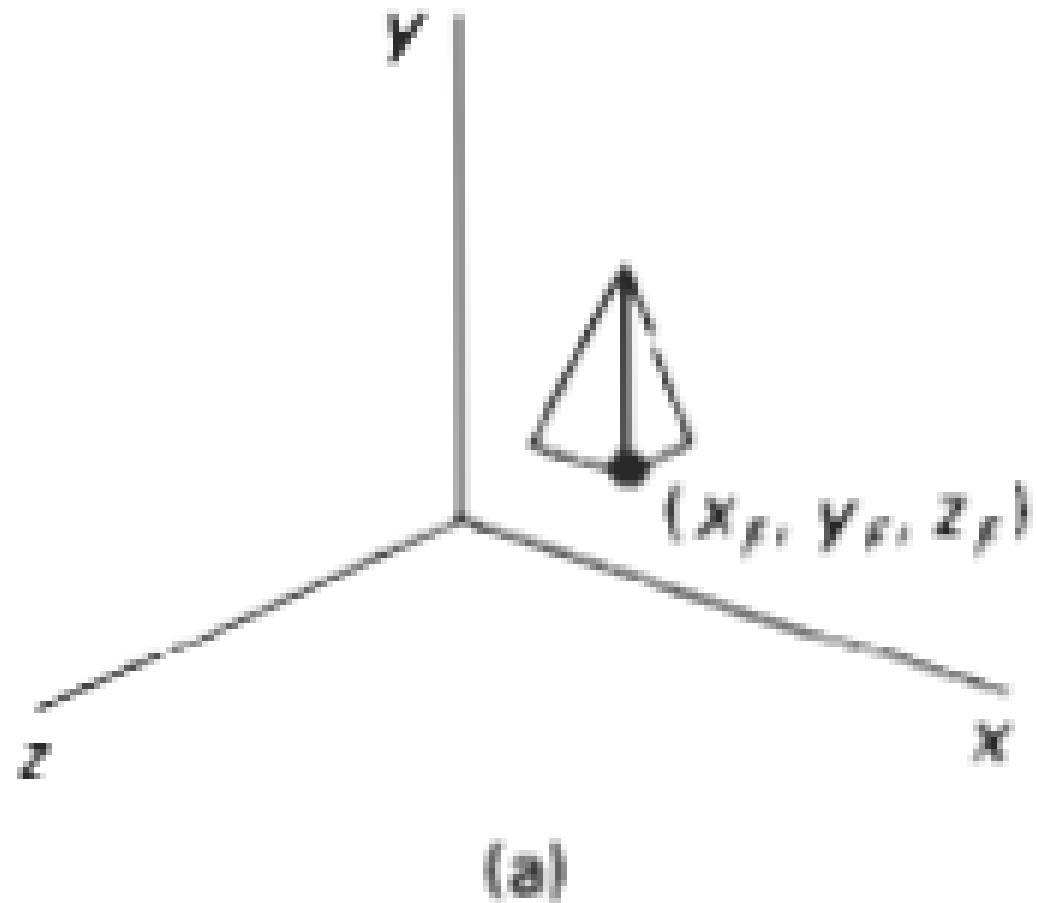
Where  $S_x$ ,  $S_y$  and  $S_z$  are scaling factors along three axis



$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

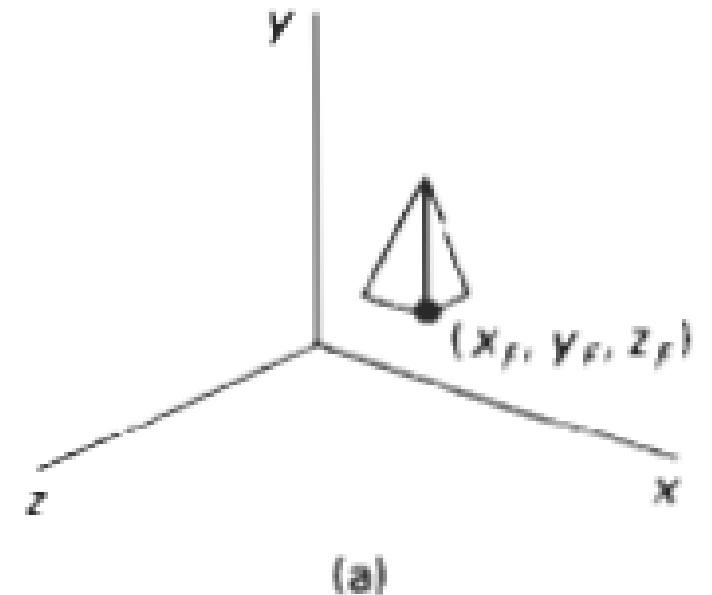
$$P' = S \cdot P$$

Scaling with respect to a selected fixed position  $(x_f, y_f, z_f)$

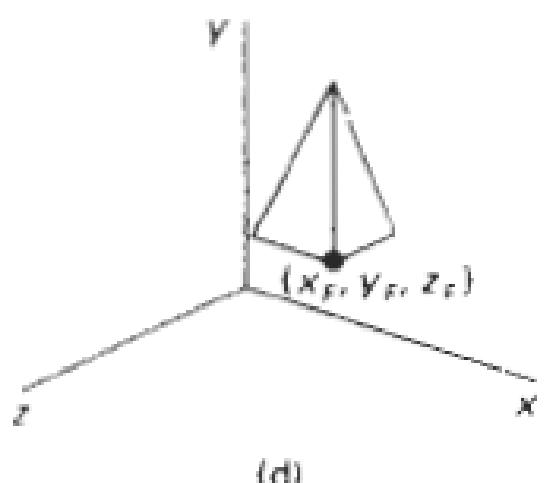
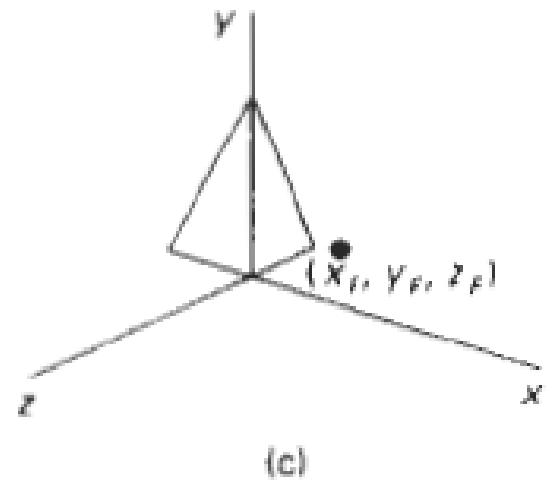
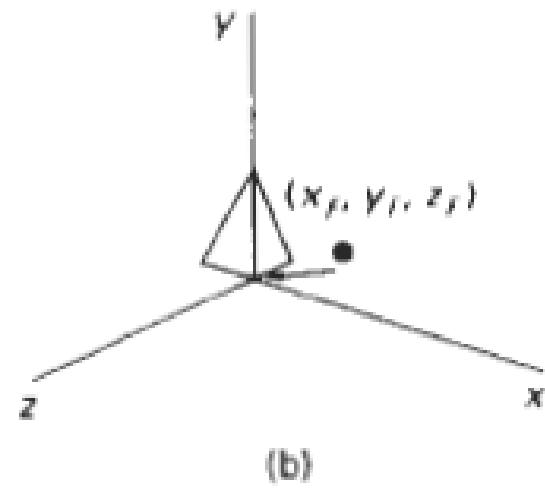
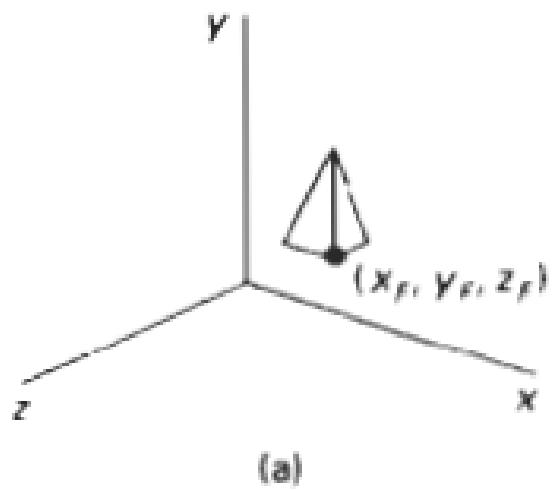


# Scaling with respect to a selected fixed position ( $x_f, y_f, z_f$ )

- Scaling with respect to a selected fixed position can be represented with the following transformation sequence:
  1. Translate the fixed point to the origin.
  2. Scale the object relative to the coordinate origin.
  3. Translate the fixed point back to its original position.



Scaling  
with  
respect to  
a selected  
fixed  
position  
 $(x_f, y_f, z_f)$



# Scaling with respect to a selected fixed position ( $x_f, y_f, z_f$ )

- The matrix representation for an arbitrary fixed-point scaling can then be expressed as the concatenation of these translate-scale-translate transformations as

$$\mathbf{T}(x_f, y_f, z_f) \cdot \mathbf{S}(s_x, s_y, s_z) \cdot \mathbf{T}(-x_f, -y_f, -z_f) = \begin{bmatrix} s_x & 0 & 0 & (1 - s_x)x_f \\ 0 & s_y & 0 & (1 - s_y)y_f \\ 0 & 0 & s_z & (1 - s_z)z_f \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



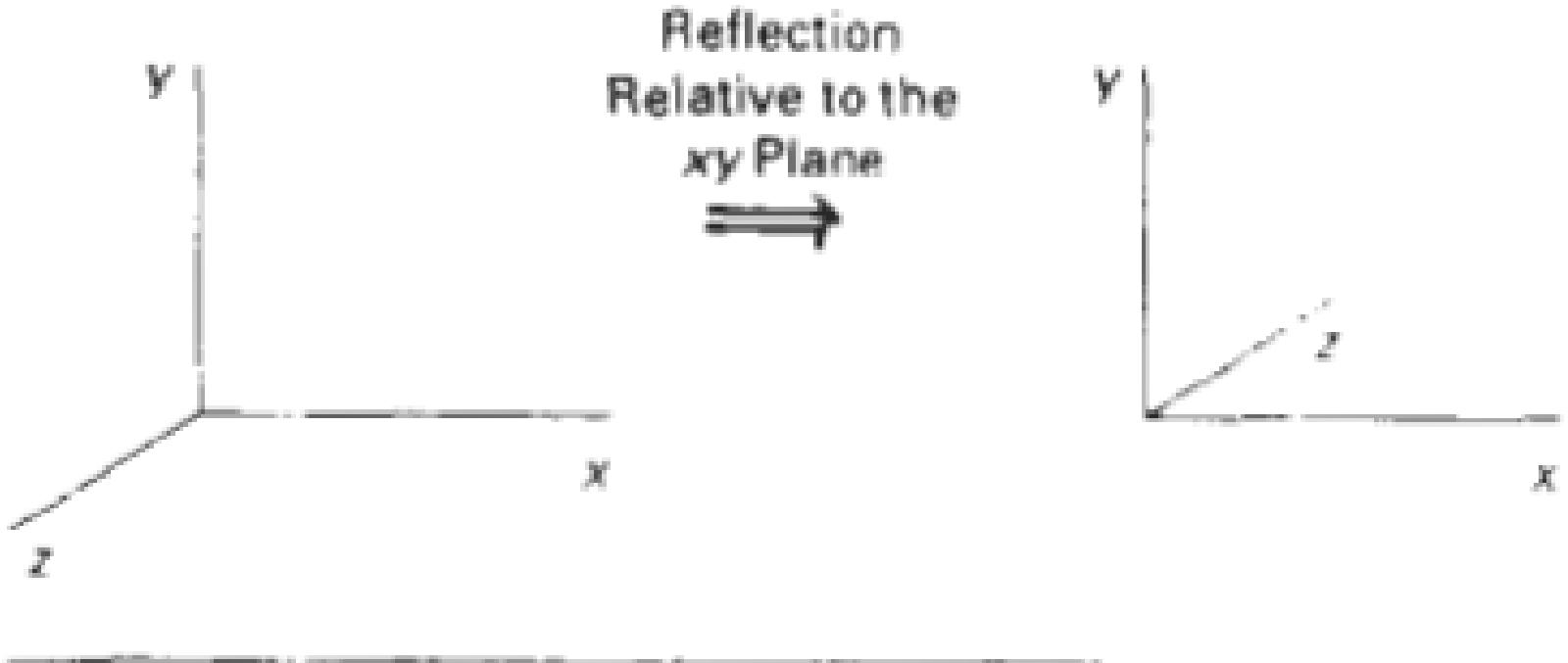
# Other Transformations in 3D

- Reflection
- Shear

# Reflection in 3D

- A three-dimensional reflection can be performed relative to a selected *reflection axis* or with respect to a selected reflection *plane*.
- When the reflection plane is a coordinate plane (either  $xy$ ,  $xz$ , or  $yz$ ), we can think of the transformation as a conversion between Left-handed and right-handed systems.

# Reflection related to $xy$ plane



**Figure II-19**  
Conversion of coordinate specifications from a right-handed to a left-handed system can be carried out with the reflection transformation 11-46.

# Reflection related to xy plane

- The matrix representation for this reflection of points relative to the xy plane is

$$RF_x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



# Shear in 3D

- In two dimensions, we discussed transformations relative to the x or y axes to produce distortions in the shapes of objects.
- In three dimensions, we can also generate shears relative to the z axis.

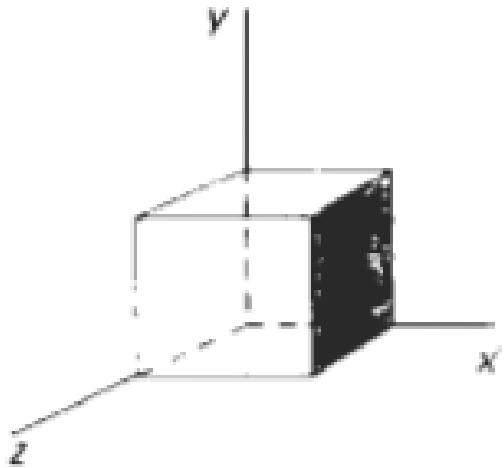
# Shear in 3D

- As an example of three-dimensional shearing, the following transformation produces a z-axis shear:

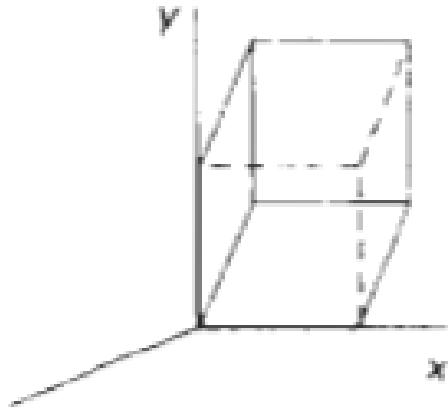
$$SH_z = \begin{bmatrix} 1 & 0 & a & 0 \\ 0 & 1 & b & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- Parameters  $a$  and  $b$  can be assigned any real values.
- The effect of this transformation matrix is to alter **x**- and **y**-coordinate values by an amount that is proportional to the **z** value, while leaving the **z** coordinate unchanged.

# Shear in 3D



(a)



(b)

$$SH_z = \begin{bmatrix} 1 & 0 & a & 0 \\ 0 & 1 & b & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- A unit cube (a) is sheared (b) by transformation matrix with  $a=b=1$

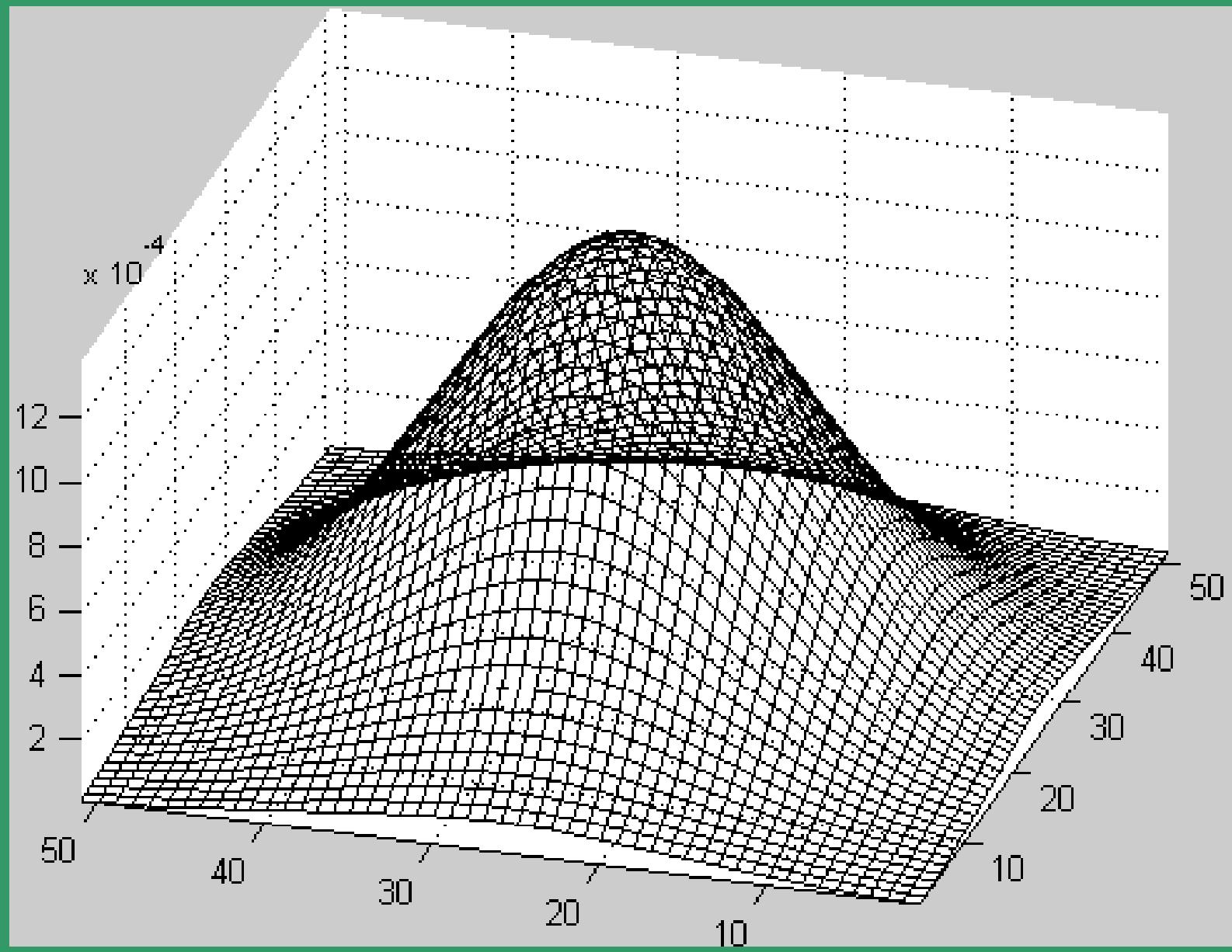


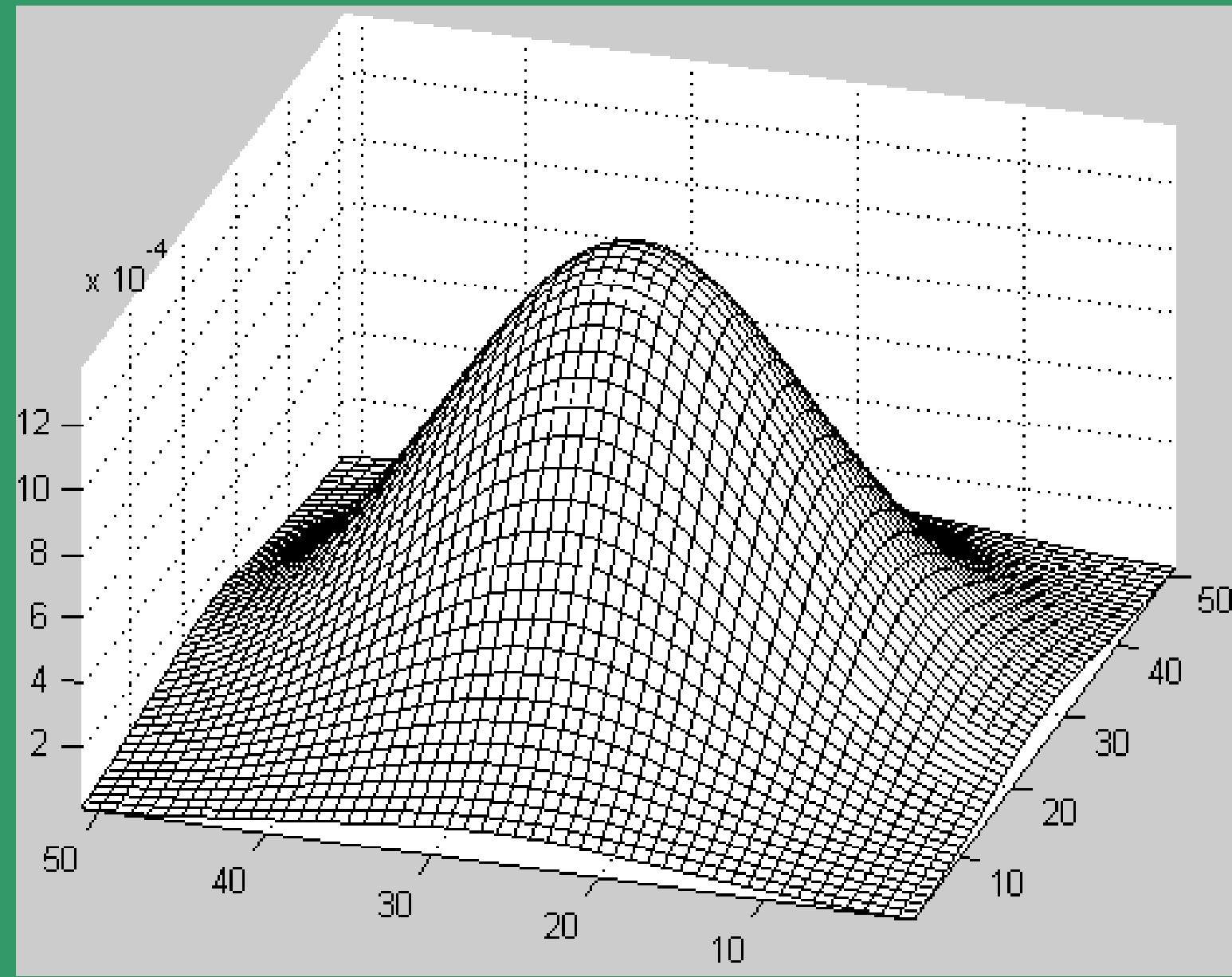
# References

3d Object Representation :

[https://www.tutorialspoint.com/computer\\_graphics/computer\\_graphic  
s\\_surfaces.htm](https://www.tutorialspoint.com/computer_graphics/computer_graphic_s_surfaces.htm)

# VISIBLE SURFACE DETECTION





# The problem of Visibility – Occlusion.

## Problem Definition:

Given a set of 3-D surfaces to be projected onto a 2-D screen, obtain the nearest surface corresponding to any point on the screen.

## Two types of methods used:

- Object-space methods (Continuous):

Compares parts of objects to each other to determine which surfaces should be labeled as visible (use of bounding boxes, and check limits along each direction).

**Order the surfaces being drawn, such that it provides the correct impression of depth variations and positions.**

### **Image Space methods (discrete):**

**Visibility is decided point by point at each pixel position on the projection plane. Screen resolution can be a limitation.**

**Hidden Surface – (a) Surface for rendering  
or  
(b) Line drawing**

# Coherence properties:

- **Object Coherence** – If one object is entirely separate from another, do not compare.
- **Face Coherence** – smooth variations across a face; incrementally modify.
- **Edge Coherence** – Visibility changes if a edge crosses behind a visible face.
- **Implied edge coherence** – Line of intersection of a planar face penetrating another, can be obtained from two points on the intersection.

- **Scanline coherence** – Successive lines have similar spans.
- **Area Coherence** – Span of adjacent group of pixels is often covered by the same visible face.
- **Depth Coherence** – Use difference equation to estimate depths of nearby points on the same surface.
- **Frame Coherence** – Pictures of two successive frames of an animation sequence are quite similar (small changes in object and viewpoint).

## Different Visible Surface Detection Methods:

- Back-face Detection
- Depth (Z) buffer method
- Scan-line method
- Depth-sorting method
- Area-subdivision method
- Octree methods
- A-buffer method
- BSP Trees
- Ray casting method

Visible surface techniques are 3D versions of sorting algorithms  
– *basically compare depth.*

## Back face Culling or removal

A Polygon (in 3D) is a back face if:  
 $V \cdot N > 0.$

Let  $V = (0, 0, V_z)$  and  $N = A\mathbf{i} + B\mathbf{j} + C\mathbf{k}$ .

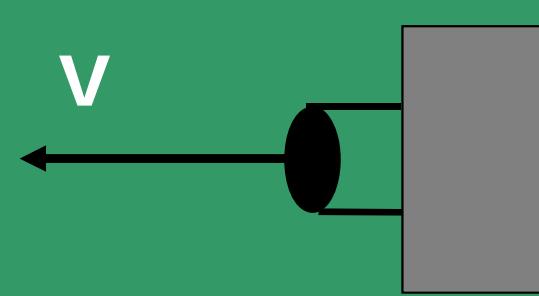
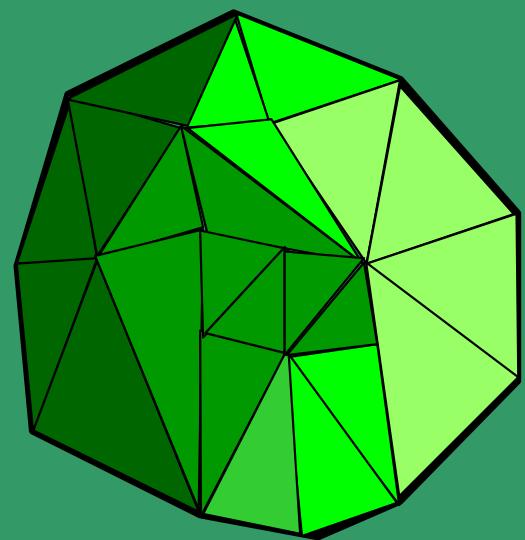
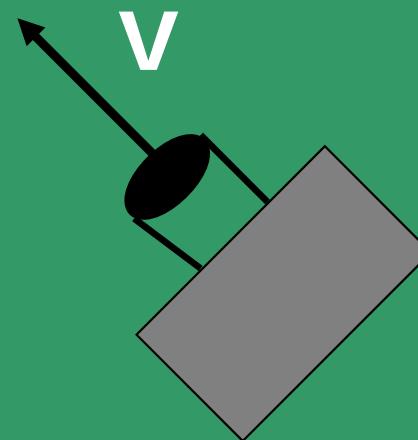
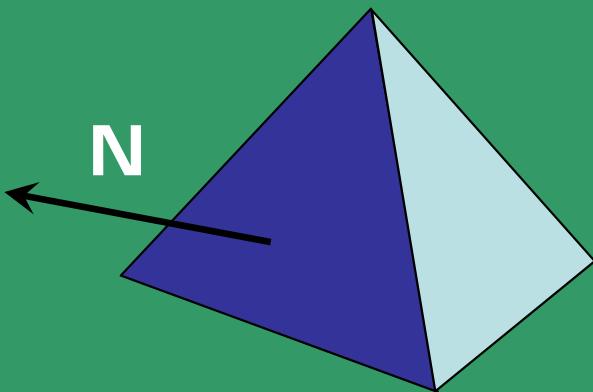
Then  $V \cdot N = V_z \cdot C$ ;

Let  $V_z$  be +ve (view along +ve Z-direction),

Check the sign of  $C$ .

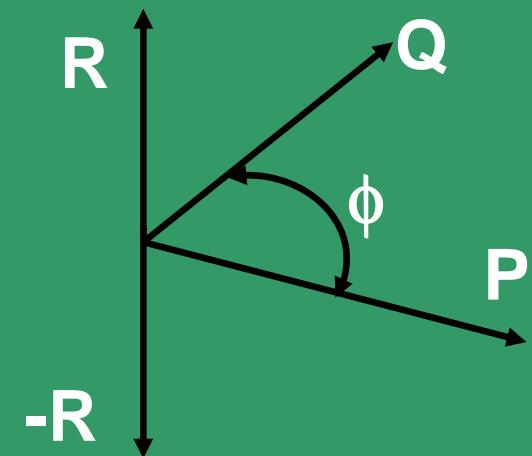
Condition of back face is thus:  
 $\text{sgn}(C) \geq 0$ .

What happens if  
 $V \cdot N = 0 ??$

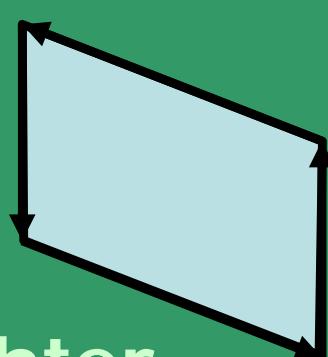


# How to get normal vector (N) for a 3D surface, polygon ?

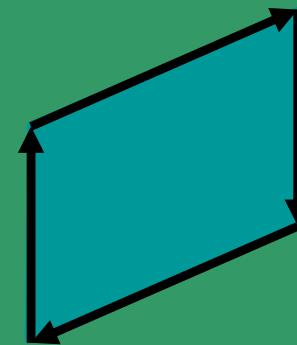
$$R = P \times Q$$



Order of vertices for calculation of N – w.r.t the front side of the surface.



Brighter Side

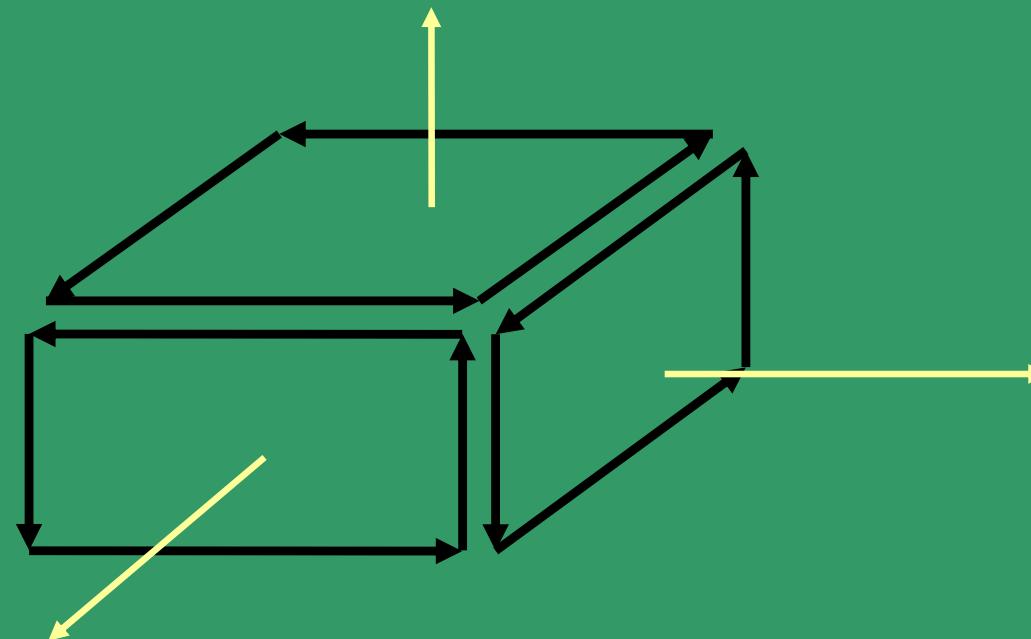


Darker side

**Take the order of vertices to be counter-clockwise for the brighter side.**

**Take the case of a cube:**

**Exterior faces of a cube:**



You can choose any two vectors (edges) to obtain the normal to the surface.

Any risks in such a case, if we randomly choose any two vectors?

Better Solution to obtain the direction cosines of N:

Assume n vertices of the polygon:  
 $(X_i, Y_i, Z_i);$   
 $i=1,2,\dots,n.$

Compute:

$$\left| \begin{array}{l} a = \sum_{i=1}^n (Y_i - Y_j)(Z_i + Z_j) \\ b = \sum_{i=1}^n (Z_i - Z_j)(X_i + X_j) \\ c = \sum_{i=1}^n (X_i - X_j)(Y_i + Y_j) \end{array} \right| \text{where } j = i + 1$$

If  $i = n, j = 1$

$$a = \sum_{i=1}^n (Y_i - Y_j)(Z_i + Z_j)$$

$$b = \sum_{i=1}^n (Z_i - Z_j)(X_i + X_j) \quad | \text{ where } j = i + 1$$

$$c = \sum_{i=1}^n (X_i - X_j)(Y_i + Y_j)$$

If  $i = n, j = 1$

Take the expression of "c":

Can you relate that to one property of the 2-D polygon in X-Y plane?

*a, b and c also describe the projection surface of the polygon on the Y-Z, Z-X and X-Y planes (or along the X, Y and Z axis) respectively.*

$$\begin{aligned}
 c &= (X_1 - X_2)(Y_1 + Y_2) + (X_2 - X_3)(Y_2 + Y_3) + (X_3 - X_1)(Y_3 + Y_1) \\
 &= X_1(Y_2 - Y_3) + X_2(Y_3 - Y_1) + X_3(Y_1 - Y_2)
 \end{aligned}$$

If A is the total area of the polygon, the projected areas are:

$$A_{YZ} = a/2; \quad A_{XZ} = b/2; \quad A_{XY} = c/2;$$

The surface normal appears to be pointing outwards. Facing towards the viewer, the edges of the polygon appears to be drawn counter-clockwise.

The polygon is a back face, if  $c \geq 0$

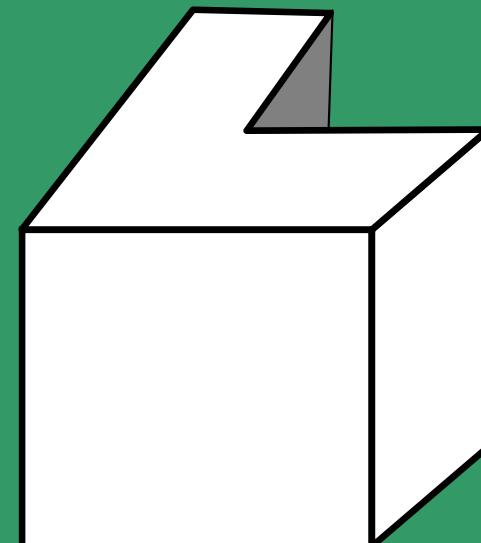
If the view direction is along the -ve Z-direction, the above condition becomes:

$$c \leq 0$$

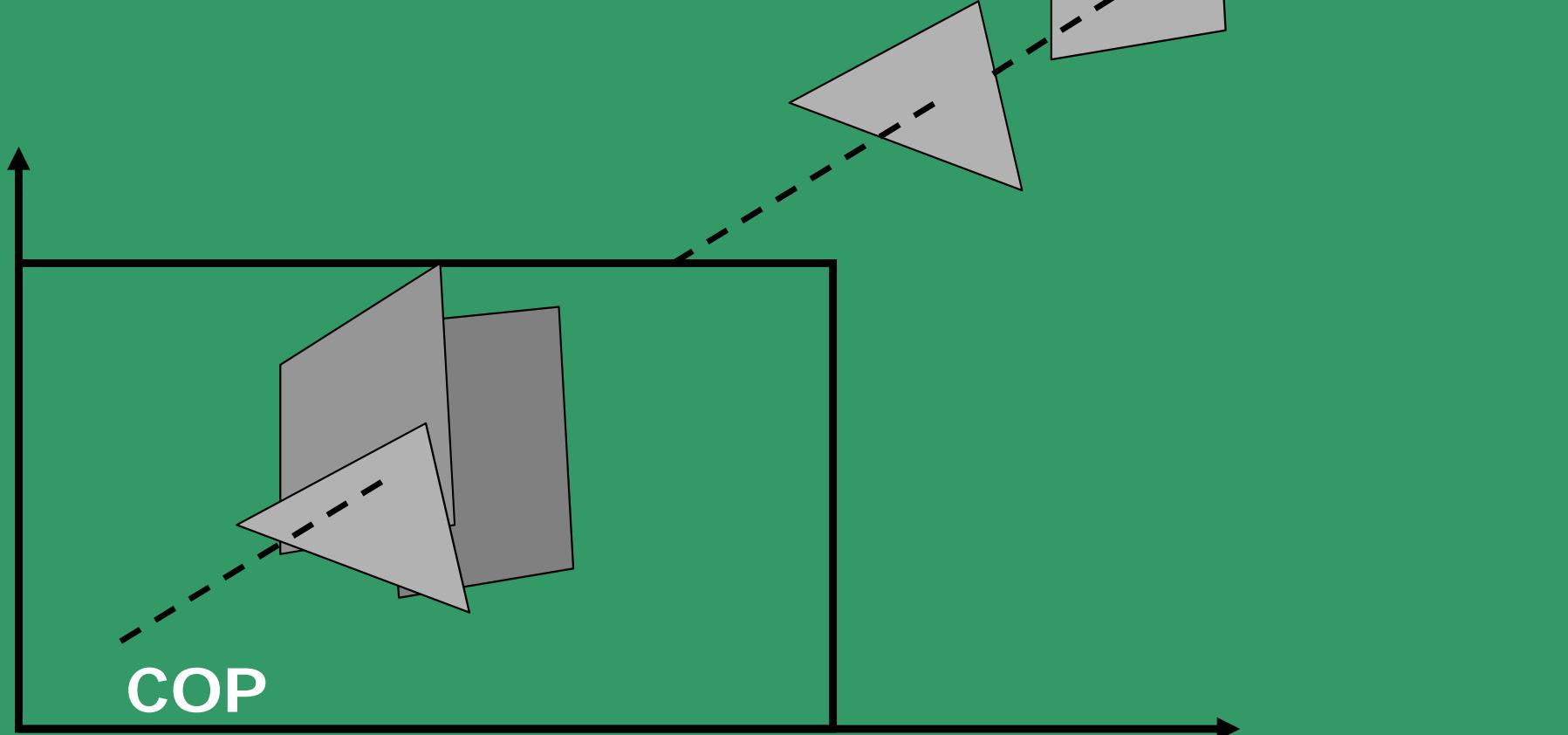
## Drawbacks of back face culling:

- Partially hidden faces cannot be determined by this method
- Not useful for ray tracing, or photometry/radiosity.

However, this is still useful as a pre-processing step, as almost 50% of the surfaces are eliminated.



Depth-buffer  
or  
Z-buffer method



**Each  $(X,Y,Z)$  point on a polygon surface, corresponds to the orthographic projection point  $(X, Y)$  on the view plane.**

**At each point  $(X, Y)$  on the PP, object depths are compared by using the depth( $Z$ ) values.**

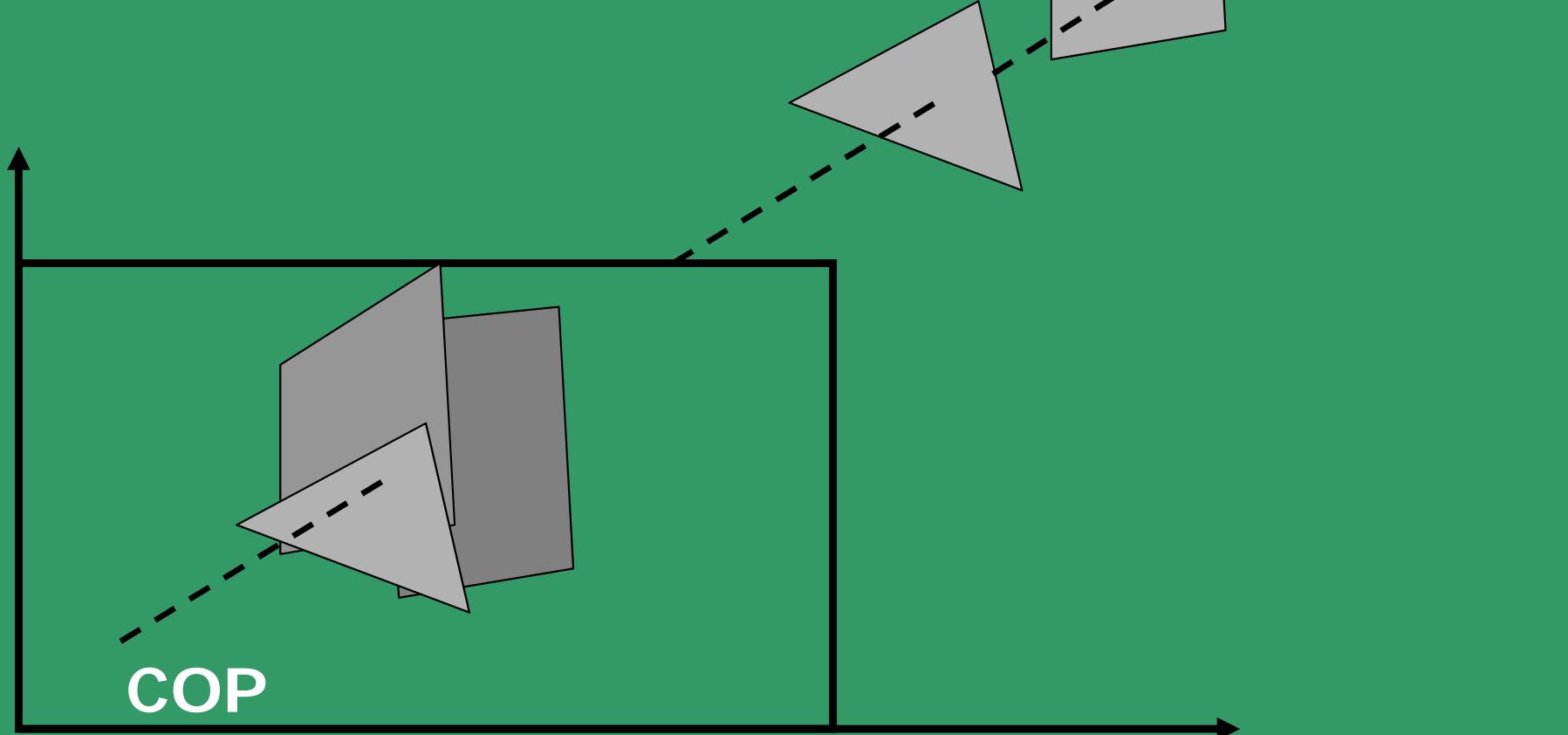
**Assume normalized coordinates:**

**(FCP)  $Z_{\max} > Z > 0$  (BCP), where  $Z_{\max} = 1$ .**

**Two buffer areas are used:**

- (i) Depth (Z) buffer:** To store the depth values for each (X, Y) position, as surfaces are processed.
- (ii) Refresh Buffer:** To store the intensity value at each position (X, Y).

Depth-buffer  
or  
Z-buffer method



## Steps for Processing:

(a) Initialize

$I_B$  = background  
intensity

$$\forall(X,Y) \begin{cases} depth(X,Y) = Z_{max} \\ refresh(X,Y) = I_B \end{cases}$$

(b) For each position on each polygon surface:

(i) Calculate depth  $Z$  for each position  $(X, Y)$  on the polygon.

(ii) If  $Z < depth(X, Y)$  then

$depth(X, Y) = Z;$

$refresh(X, Y) = I_s(X, Y)$

$I_s$  is the projected intensity value of the surface at position (X, Y), which has the minimum value of Z, at the current stage of iteration.

### Calculation of Z:

**Equation of the surface:**

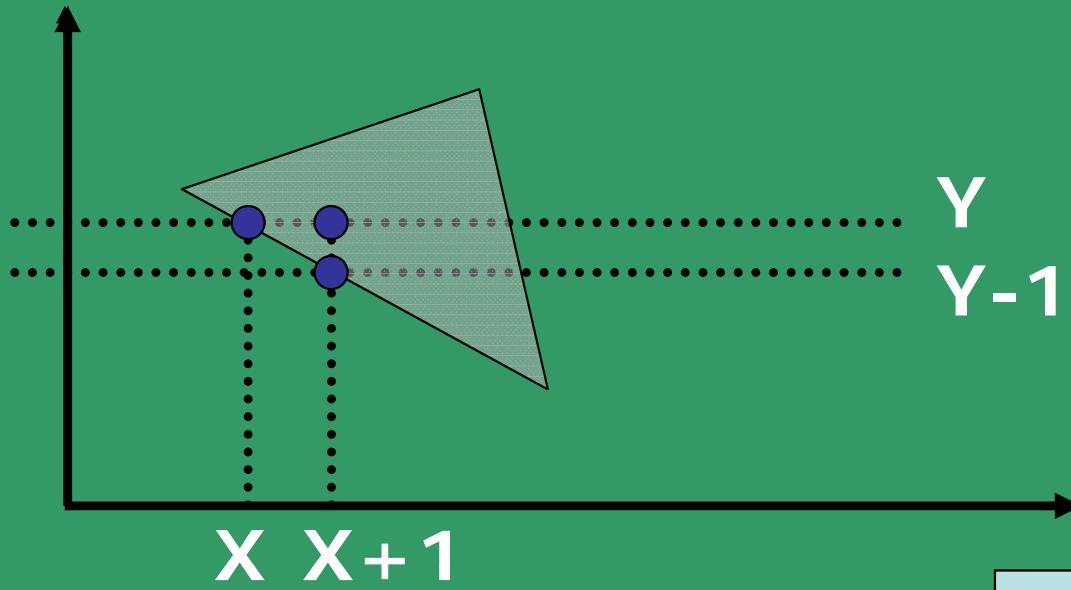
$$AX + BY + CZ + D = 0;$$

$$Z = \frac{-AX - BY - D}{C}$$

**Equation for Z at the next scan line:**

**Remember scanline/Polyfill algorithm?  
Using edge coherence, we get for the next  
scanline ( $Y+1$ ):**

$$X' = X - 1/m;$$



$$Z_{X+1} =$$

$$Z_{Y+1} =$$

For a vertical edge:  $Z' =$

So to implement the algorithm, three constants are required for each surface:

$$-\frac{A}{C}, \frac{\left(\frac{A}{m} + B\right)}{C}, \frac{B}{C}$$

What is the special condition,  $C = 0$ ?

Using the above three constants, we can keep calculating the successive depth values along and for successive scanlines. Similar approaches can be used for curved surfaces:  
 $Z = f(X, Y)$ .

## Z-Buffer Algo:

```
for all (x,y)
    depth(x,y) = -∞ /* Watch this change */
    refresh(x,y) = IB

for each polygon P
    for each position (x,y) on polygon P
        calculate depth z

        if z > depth(x,y) then
            1. depth(x,y) = z
            2. refresh(x,y) = IP(x,y)
```

Lets take an example →

Z-values of the coordinates:

4	4
4	4

9	8
8	7

7	6	5
7	6	5





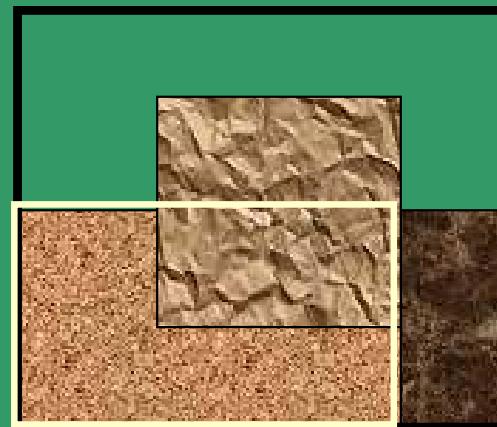
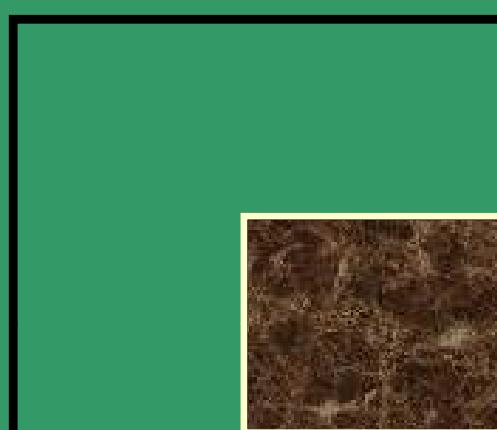
**Z-Buffer  
values**

**Z-values of the coordinates:**

4	4
4	4

9	8
8	7

7	6	5
7	6	5



	9	8	
7	8	7	4

7	6	5	4

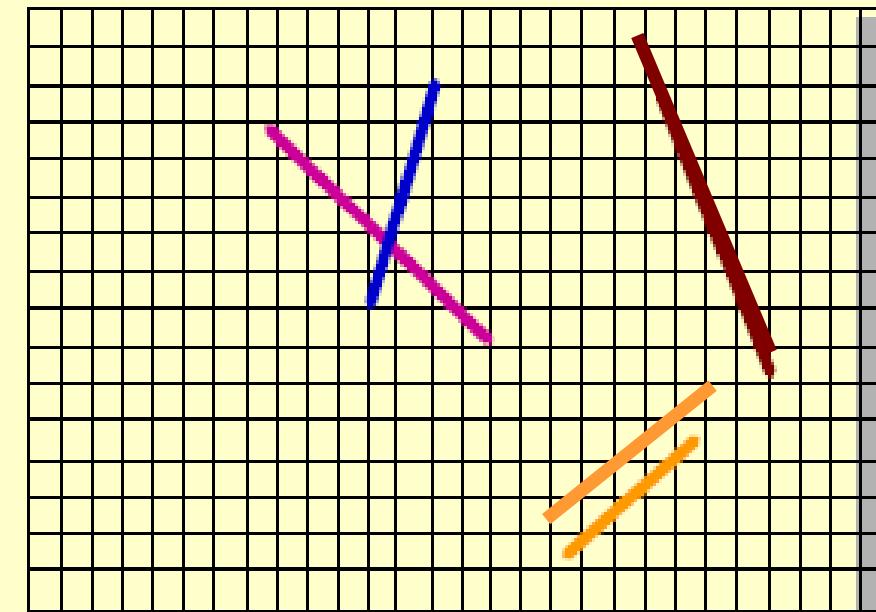
**Z-Buffer  
values**

**IMAGES (after pseudo-texture rendering)**

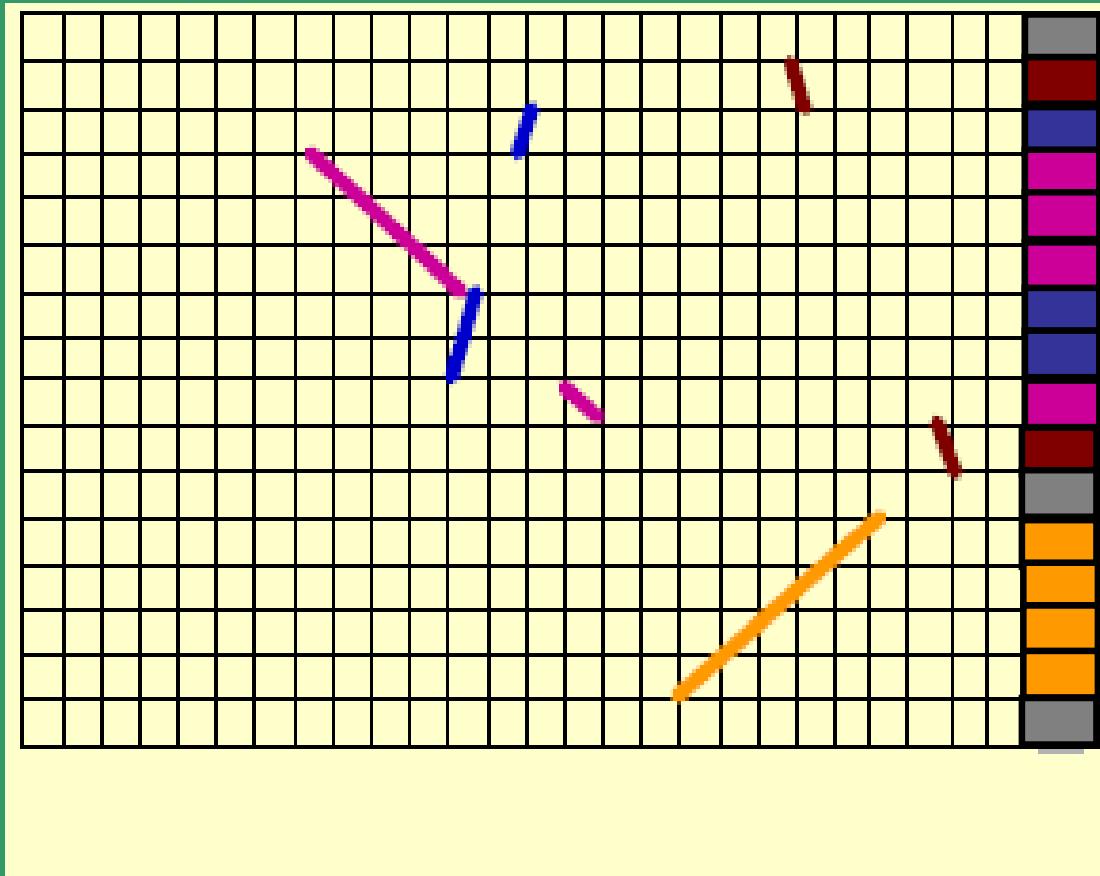
V



*z*-Buffer



V



## SCAN-LINE Algorithms (for VSD)

Extension of 2-D Scanline (polyfill) algorithm.

Here we deal with a set of polygons.

Data structure used:

ET (Edge Table),

AET (Active Edge Table) and

PT (Polygon Table).

Edge Table entries contain information about edges of the polygon, bucket sorted based on each edge's smaller Y coordinate.

Entries within a bucket are ordered by increasing X-coordinate of their endpoint.

## Structure of each entry in ET:

- X-Coordn. of the end point with the smaller Y-Coordn.
- Y-Coordn. of the edge's other end point
- $\Delta X = 1/m$
- Polygon ID

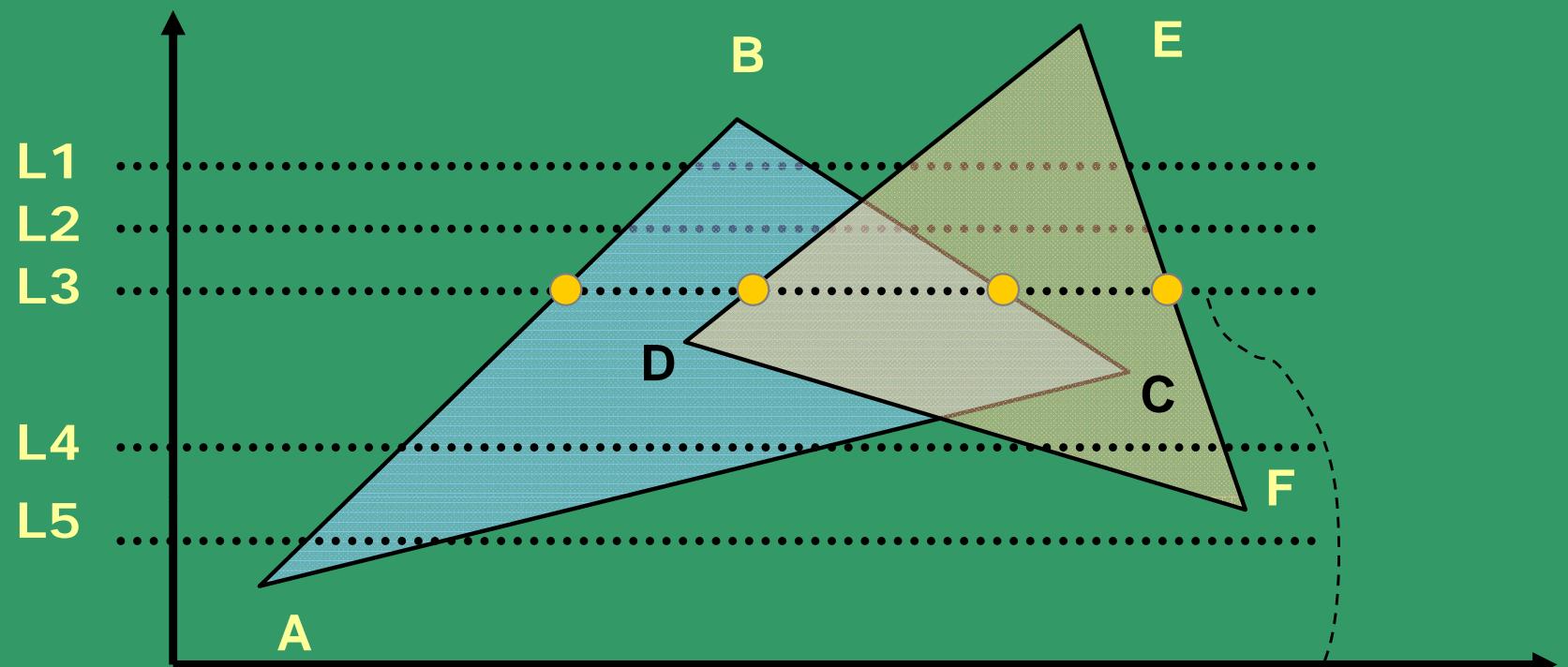


## Structure of each entry in PT:

- Coefficients of the plane equations
- Shading or color information of the polygon
- Flag (IN/OUT), initialized to 'false'



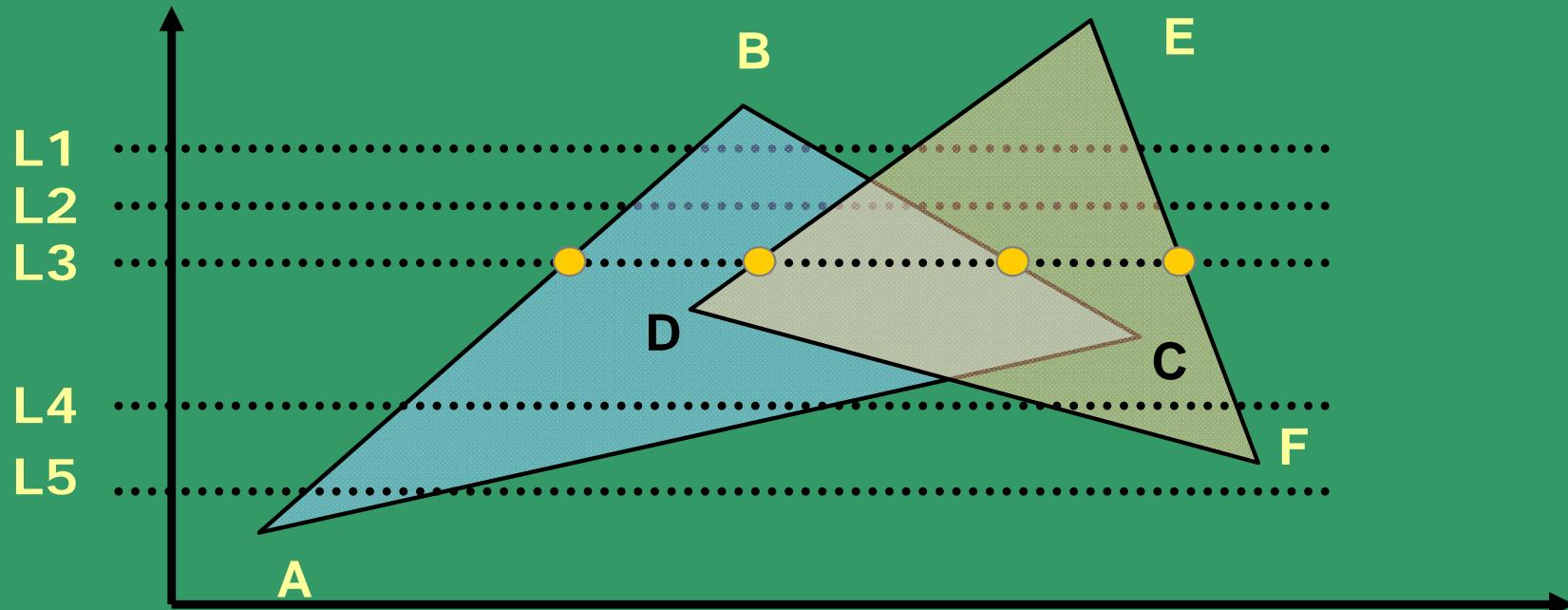
Take Adjacent (not successive) pairs of intersections to fill;  
FILL (within pair) if POLY\_FLAG is set to IN.



L3 —————— •————— •————— •————— •—————

ABC IN;    ABC IN;    ABC OUT;  
DEF OUT    DEF IN.    DEF IN.

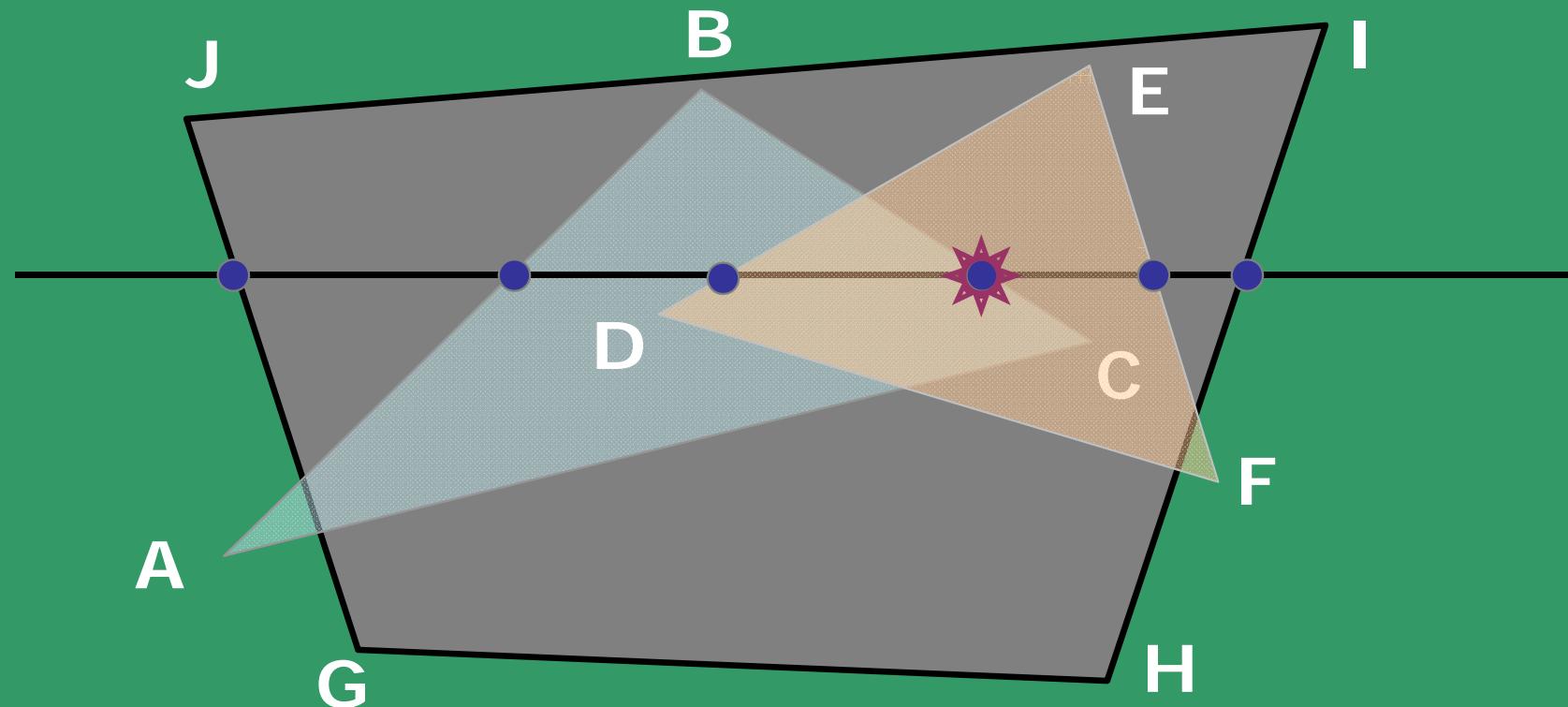
Compare Z values in this region  
to find the visible Z value.



## AET Contents

Scan Line	Entries				
L5	AB	CA			
L4	AB	CA	FD	EF	
L3, L2	AB	DE	BC	EF	
L1	AB	BC	DE	EF	

## Three non-intersecting polygons

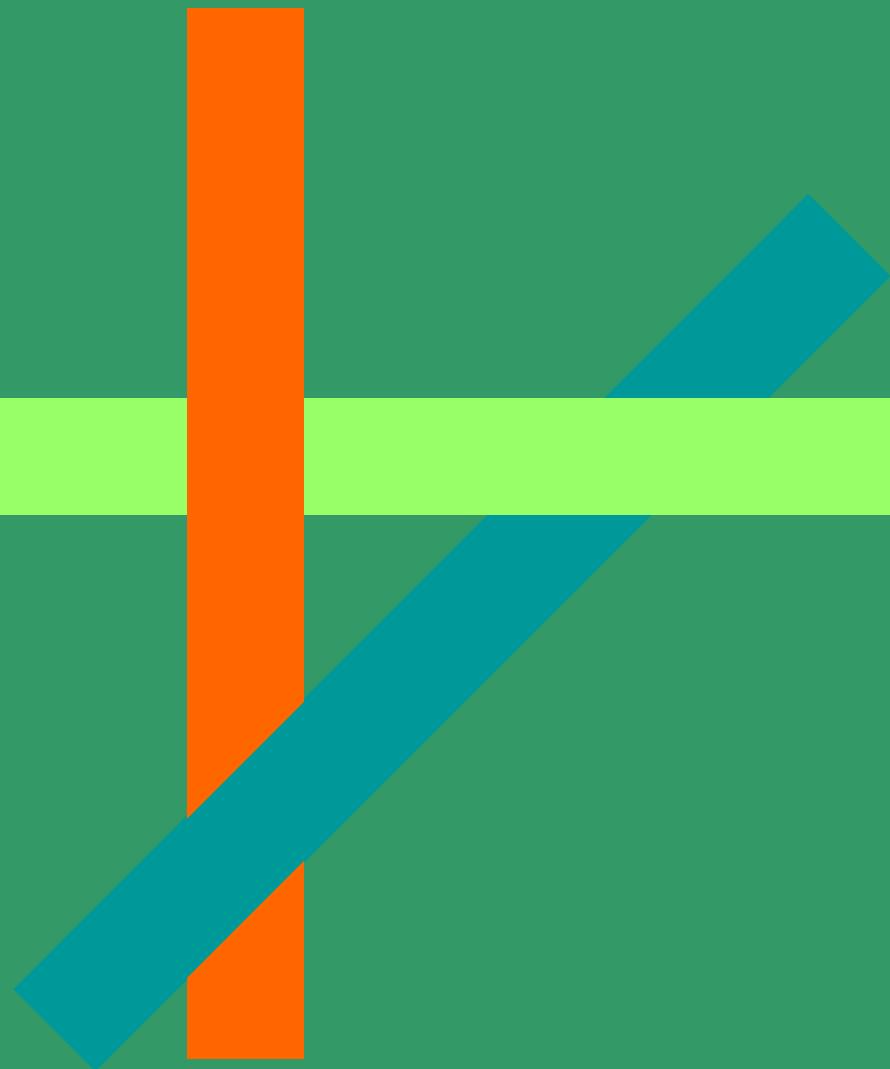
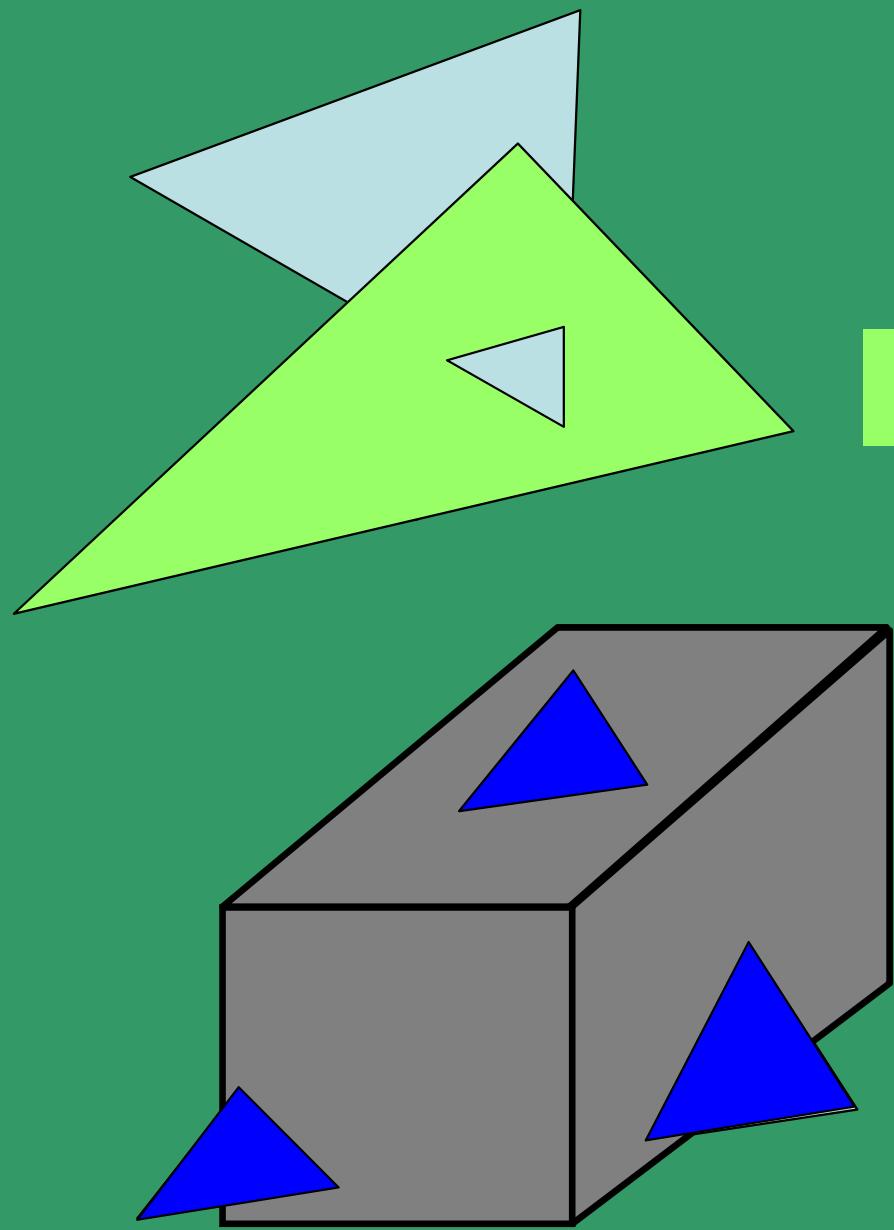


GHIJ is behind ABC and DEF.

So, when scanline leaves edge BC, it is still inside polygons DEF and GHIJ. If polygons do not intersect, depth calculations and comparisons between GHIJ and DEF can be avoided.

Thus depth computations are unnecessary when the scanline leaves an obscured polygon. It is required only when it leaves an obscuring polygon.

Additional treatment is necessary for intersecting polygons.



## Depth-sorting or Painter's Algorithm

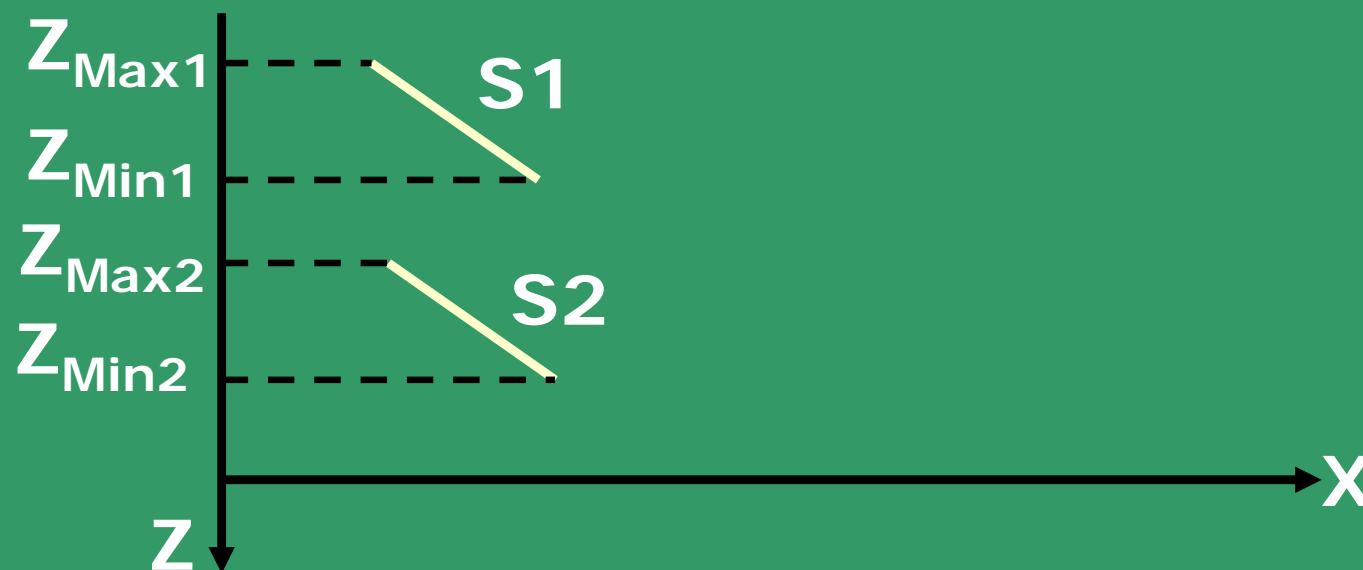
**Paint the polygons in the frame buffer in order of decreasing distance from the viewpoint.**

### **Broad Steps:**

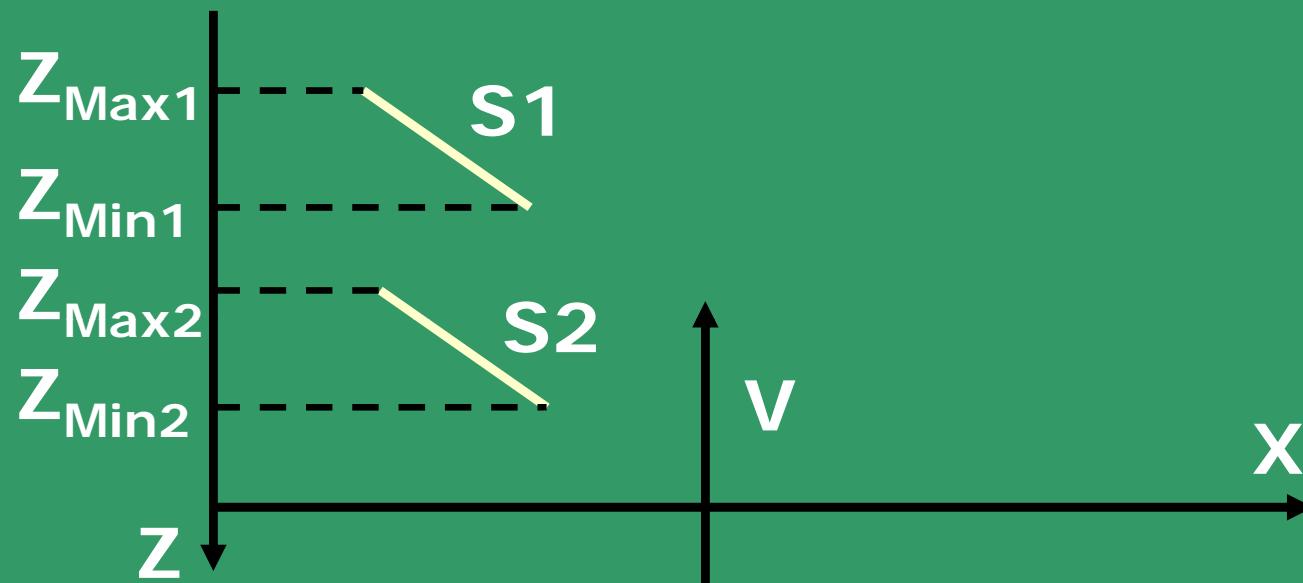
- **Surfaces are sorted in increasing order of DEPTH.**
- **Resolve ambiguities when polygons overlap (in DEPTH), splitting polygons if necessary.**
- **Surfaces are scan converted in order, starting with the surface of greatest DEPTH.**

## Principle:

Each layer of paint (polygon surface of an object) covers up the previous layers while drawing.



Since,  $Z_{\text{Min1}} > Z_{\text{Max2}}$  no overlap occurs.  
S1 is first scan converted, and then S2.  
This goes on, as long as no overlap occurs.



If depth overlap occurs, additional comparisons are necessary to reorder the surfaces.

The following set of tests are used to ensure that no re-ordering of surfaces is necessary:

## Four(4) Tests in Painter's algorithm

1. The boundary rectangles in the X-Y plane for the two surfaces do not overlap.
2. Surface S is completely behind the overlapping surface relative to the viewing position.
3. The overlapping surface is completely in front of S relative to the viewing position.
4. The projections of the two surfaces onto the view plane do not overlap.

Tests must be performed in order, as specified.

### Condition to RE-ORDER the surfaces:

If any one of the 4 tests is TRUE, we proceed to the next overlapping surface. i.e. If all overlapping surfaces pass at least one of the tests, none of them is behind S.

No re-ordering is required, and then S is scan converted.

RE-ORDERing is required if all the four tests fail.

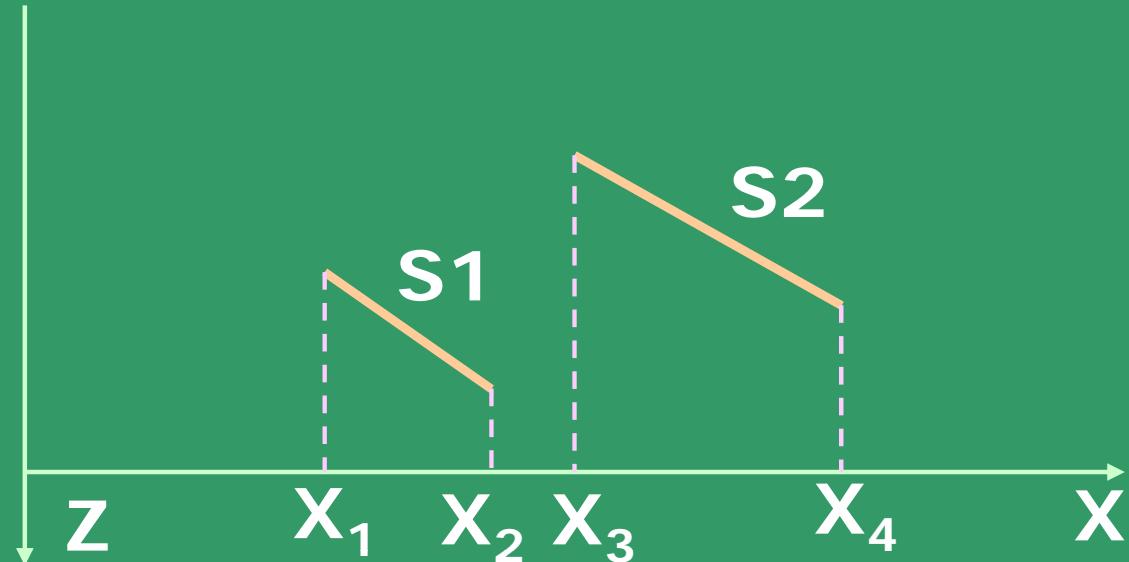
## TEST #1:

The boundary rectangles in the X-Y plane for the two surfaces do not overlap.

We have depth overlap, but no overlap in X-direction.

Hence, Test #1 is passed, scan convert S2 and then S1.

If we have X overlap, check for the rest.



## TEST #2:

**Surface S is completely behind the overlapping surface relative to the viewing position.**

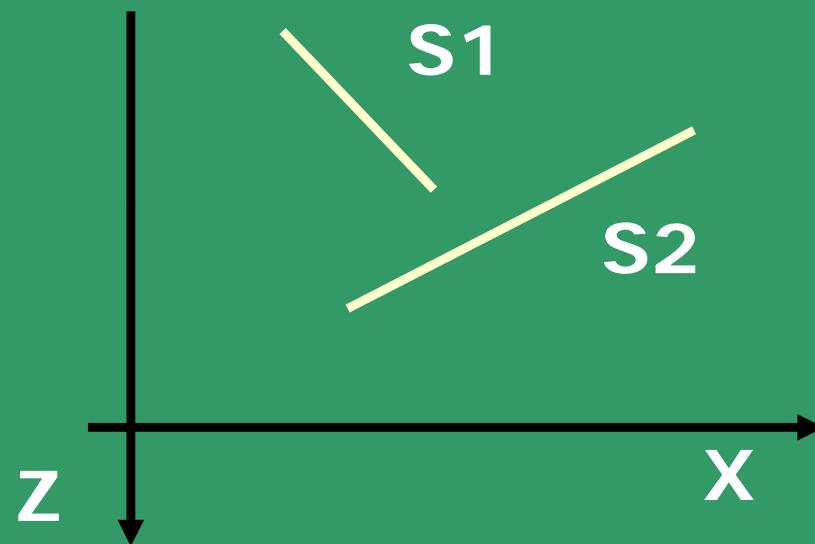


Fig. 1. S1 is completely behind/inside the overlapping surface S2

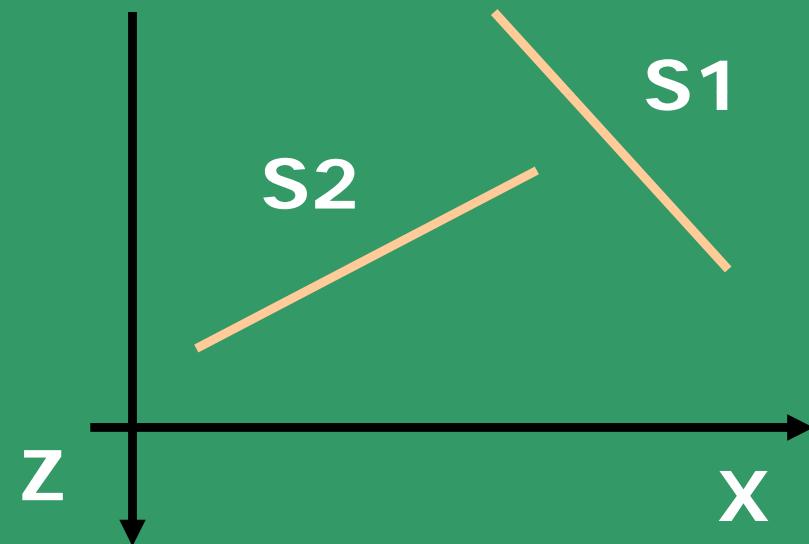
### TEST #3:

The overlapping surface is completely in front of S relative to the viewing position.

S1 is not completely behind S2.

So, Test #2 fails.

Fig. 2. Overlapping surface S2 is completely front/outside S1.



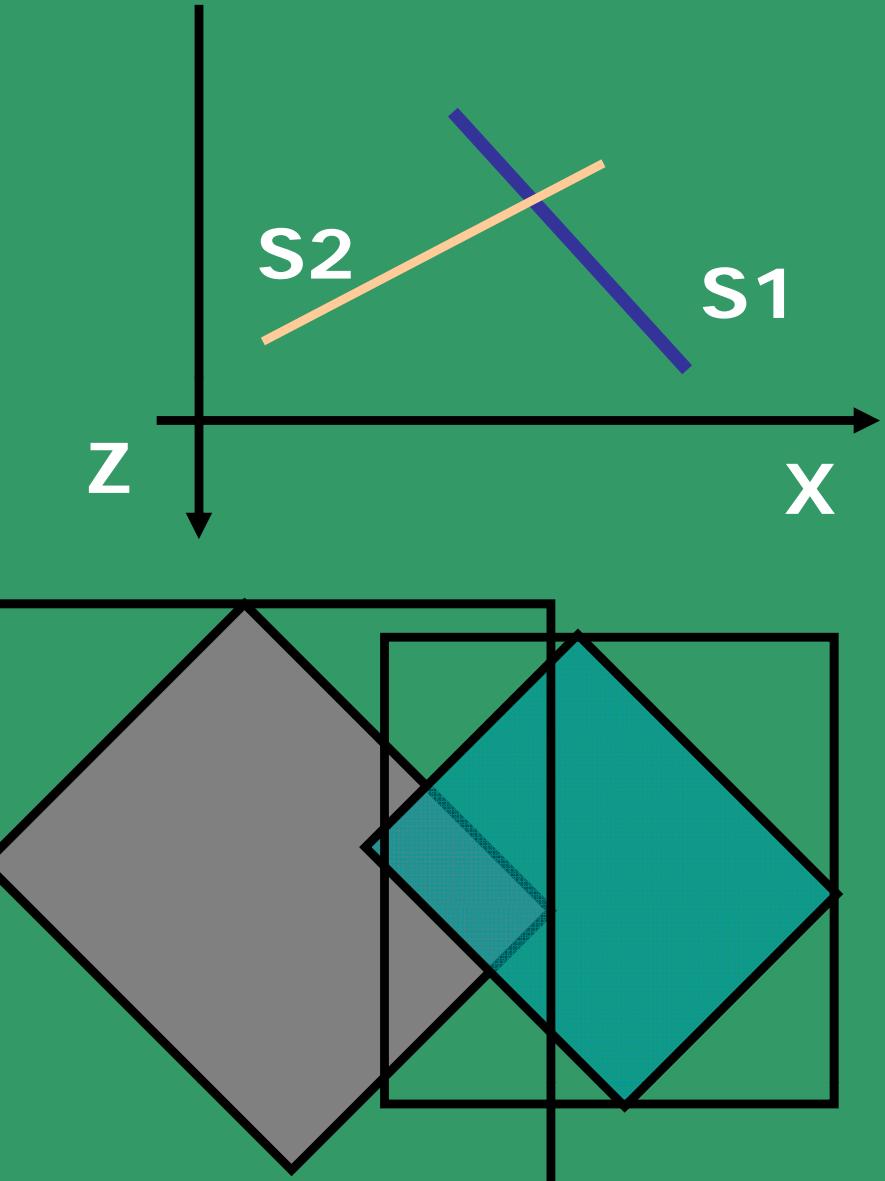
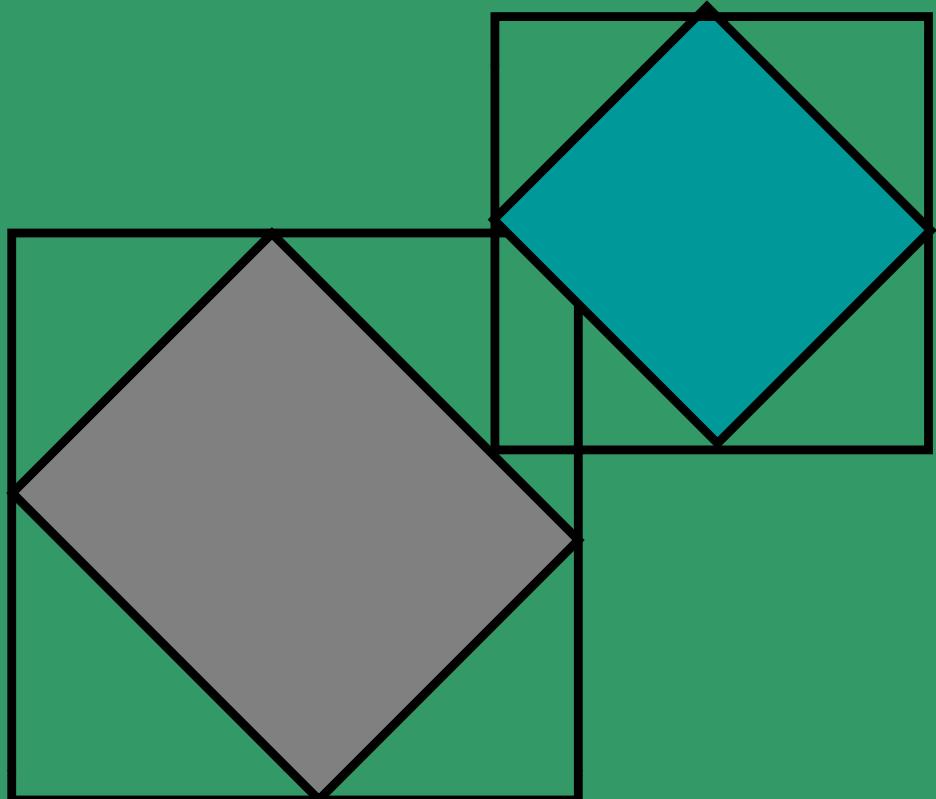
In Fig. 2, S2 is in front of S1, but S1 is not completely inside S2 – Test #2 is not TRUE or FAILS, although Test #3 is TRUE.

## How to check these conditions?

- i) Set the plane equation of S2, such that the surface S2 is towards the viewing position.
- ii) Substitute the coordinates of all vertices of S1 into the plane equation of S2 and check for the sign.
- iii) If all vertices of S1 are inside S2, then S1 is behind S2. (Fig. 1).
- iv) If all vertices of S1 are outside S2, S1 is in front of S2.

## TEST #4:

The projections of the two surfaces onto the view plane do not overlap.



## Four(4) Tests in Painter's algorithm

### - revisited

1. The boundary rectangles in the X-Y plane for the two surfaces do not overlap.
2. Surface S is completely behind the overlapping surface relative to the viewing position.
3. The overlapping surface is completely in front of S relative to the viewing position.
4. The projections of the two surfaces onto the view plane do not overlap.

Tests must be performed in order, as specified.

### Condition to RE-ORDER the surfaces:

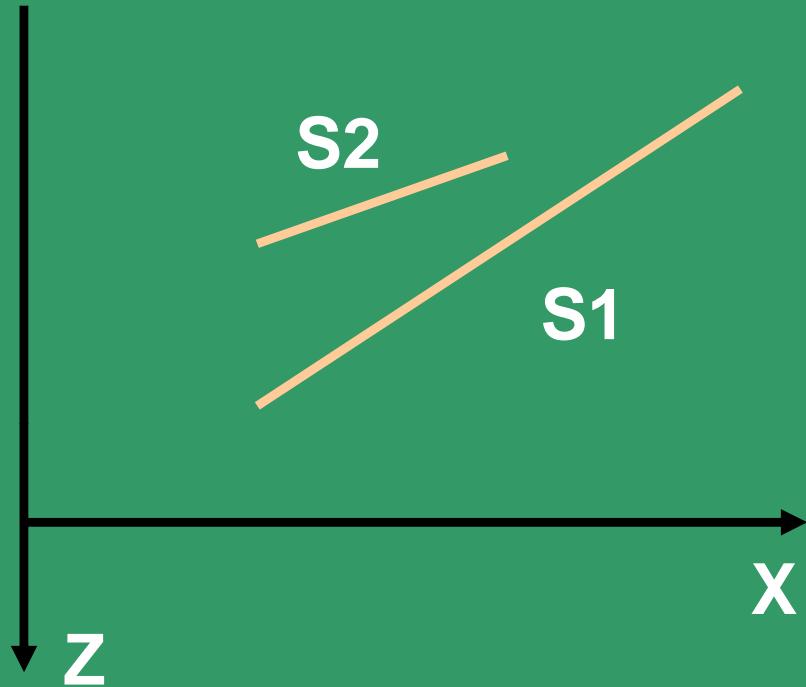
If any one of the 4 tests is TRUE, we proceed to the next overlapping surface. i.e. If all overlapping surfaces pass at least one of the tests, none of them is behind S.

No re-ordering is required, and then S is scan converted.

RE-ORDERing is required if all the four tests fail.

# **Case Studies – examples of Painter's Algorithm**

## Case Study - I



Initial order:  
**S1 -> S2**

Change the order:  
**S2 -> S1**

## Case Study - II

Initial Order:

$S_1 \rightarrow S_2 \rightarrow S_3$ .

$S_1 \rightarrow S_2$ ,

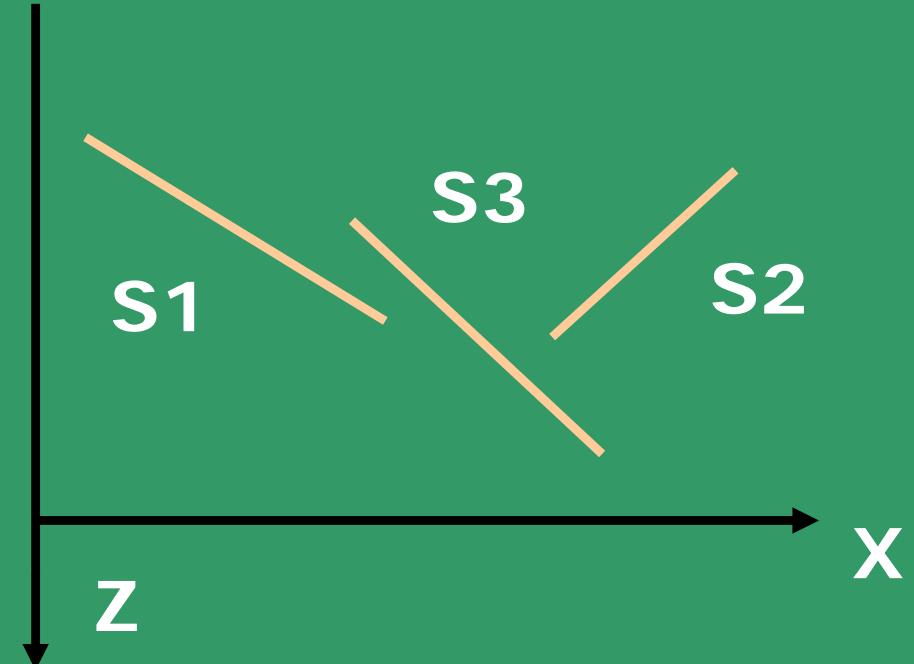
Test 1 passed.

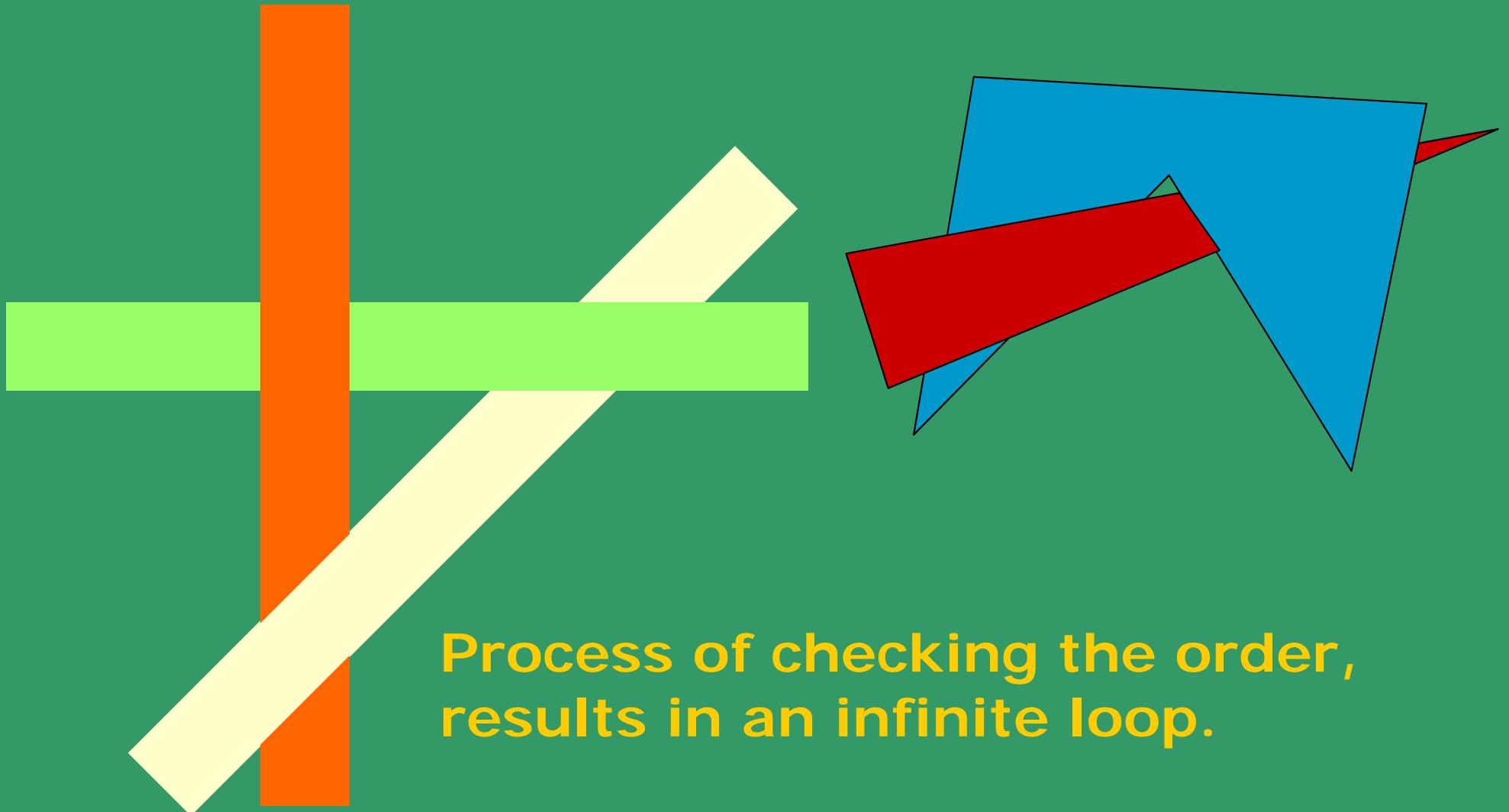
Check  $S_1$ ,  $S_3$ . All tests fail.

Interchange.  $S_3 \rightarrow S_2 \rightarrow S_1$ .

Check  $S_2$  and  $S_3$ . All tests fail again.

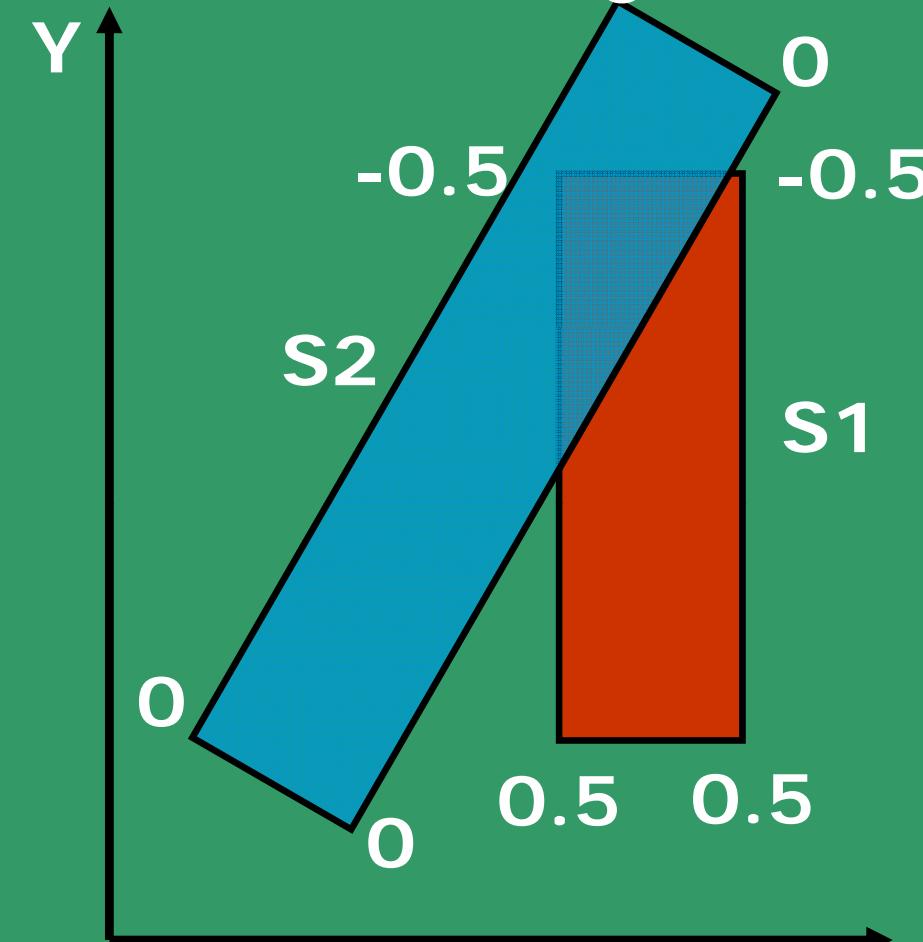
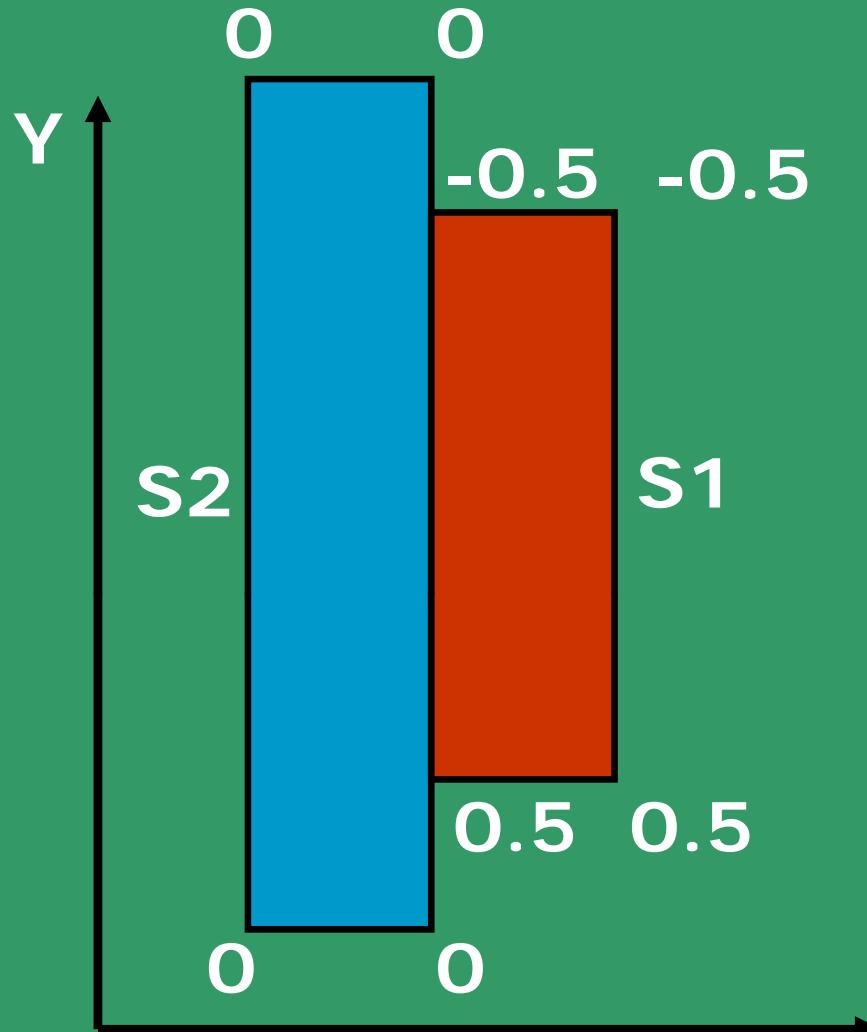
Correct Order:  $S_2 \rightarrow S_3 \rightarrow S_1$ .





**Process of checking the order,  
results in an infinite loop.**

**Avoided by setting a FLAG for a  
surface that has been re-ordered  
or shuffled. If it has to be altered  
again, split it into two parts.**



What happens in these cases ?

## Area sub-division method

**Examine and divide if necessary**

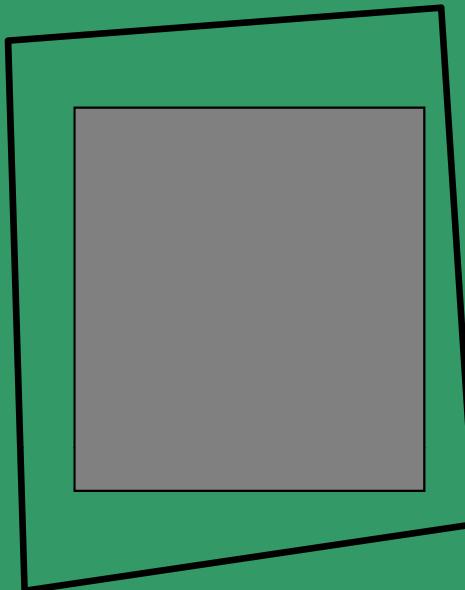
- Works in image-space
- Area Coherence is exploited
- Divide-and-conquer strategy.

**WARNOCK's**  
**Algorithm**

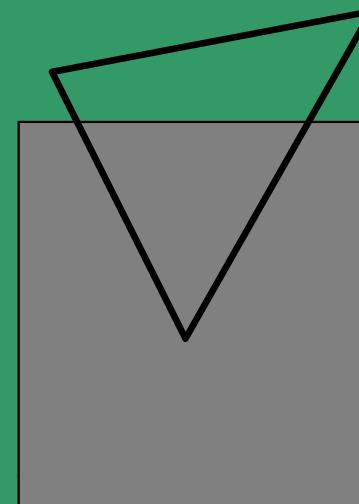
Possible relationships of the area of interest (rectangular, AOI) and the projection of the polygon:

1. Surrounding polygons are completely overlapping the area of interest.
2. Intersecting polygons intersect the area.
3. Contained polygons are completely inside the area.
4. Disjoint polygons are completely outside the area.

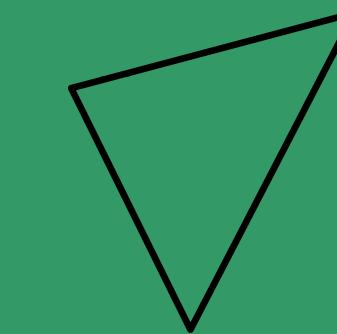
## Four cases of Polygon position w.r.t rectangular AOI.



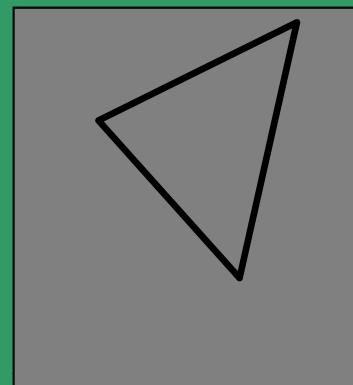
Surrounding



Intersecting



Contained



Disjoint

Treat interior part of the intersection and contained as equivalent. Disjoint is irrelevant. The decisions about division of an area is based on:

1. All polygons are disjoint from the area.  
Display background color.
2. Only one intersecting part or interior (contained): Fill the area with background color and then scan-convert the polygon.
3. Single surrounding polygon, and no intersecting or contained polygon. Use color of surrounding polygon to shade the area.

**4. More than one polygon intersect, contained in and surrounding the area. But the surrounding polygon is in front of all other polygons. Then fill the area with the color of the surrounding polygon.**

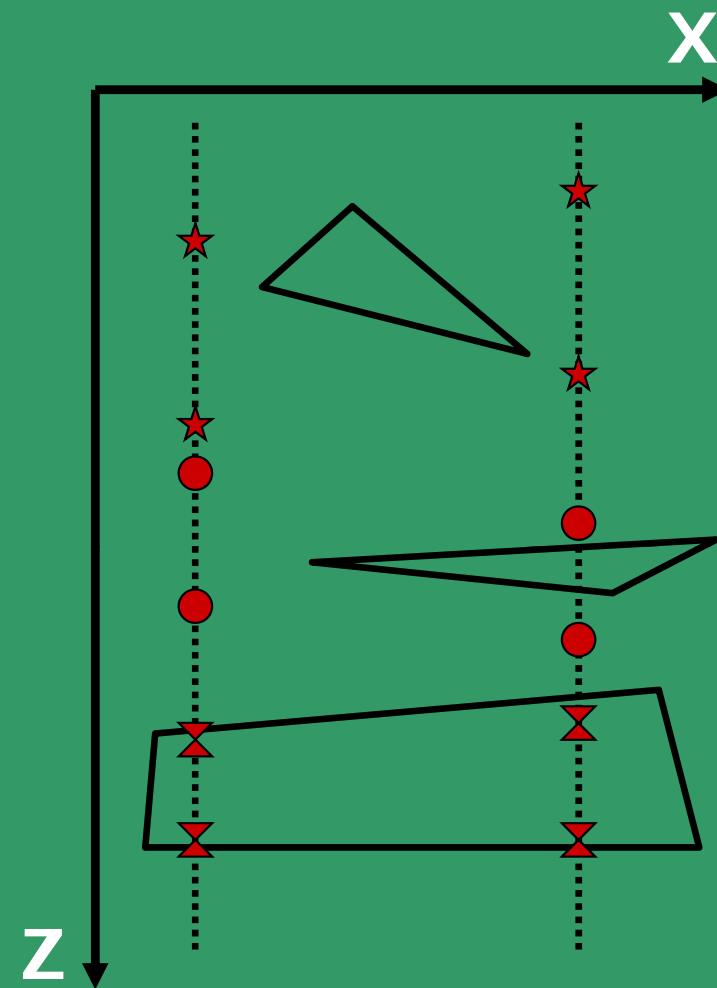
**Step 4 is implemented by comparing the Z-coordinates of the planes of all polygons (involved) at the four corners of the area**

**If all four tests fail, divide the rectangular area into four equal parts.**

**Recursively apply this logic, till you reach the lowest minimum area or maximum resolution – pixel size.**

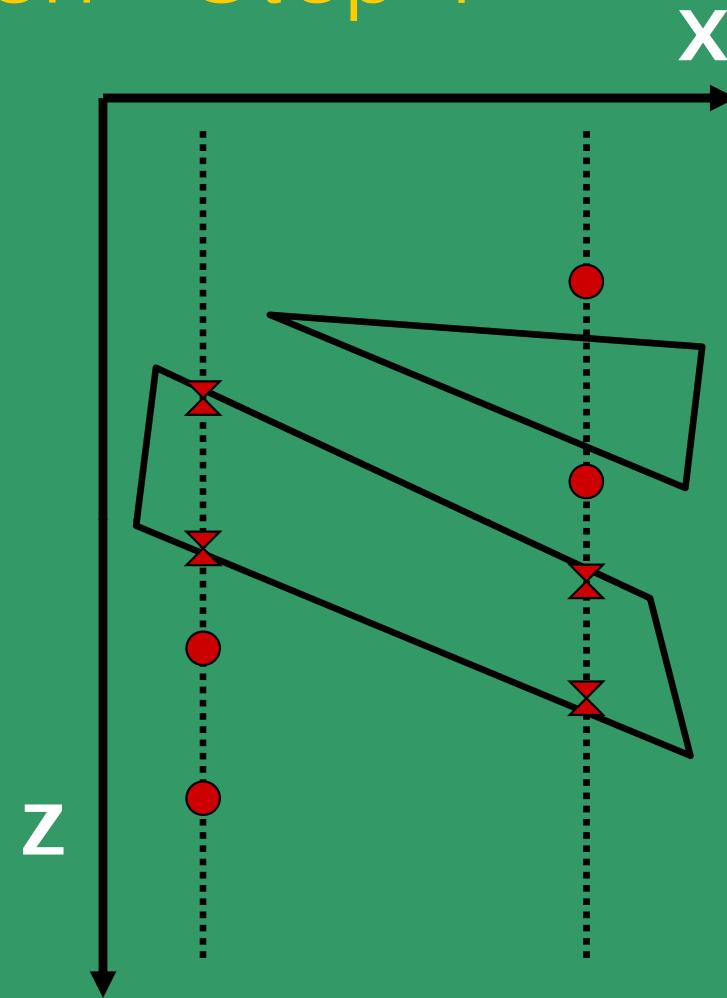
**Use nearest surface in that case to paint/shade.**

## Decision - Step 4



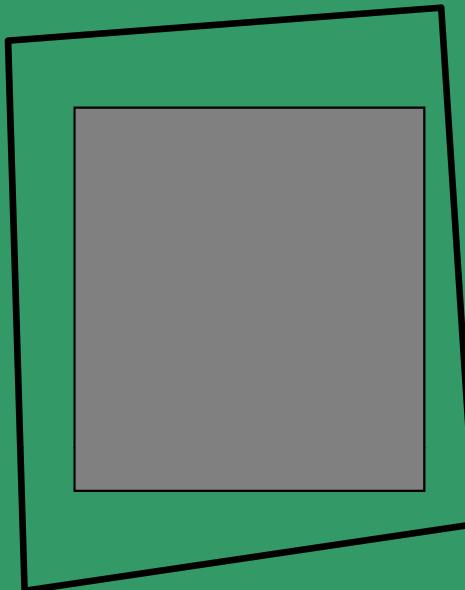
Intersection with:

- ★ Contained Polygon
- Intersecting Polygon

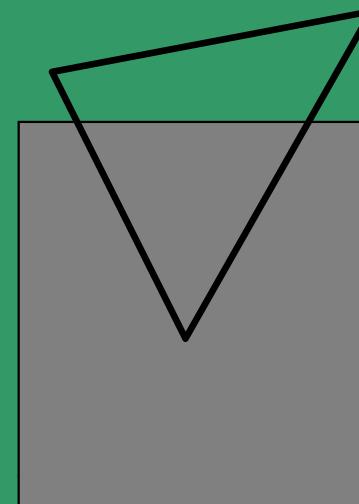


★ Surrounding  
Polygon

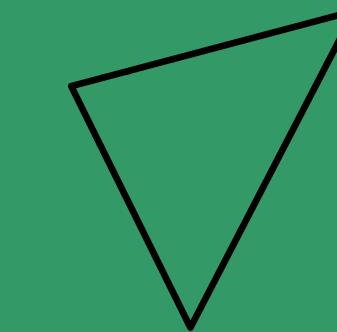
## Four cases of Polygon position w.r.t rectangular AOI.



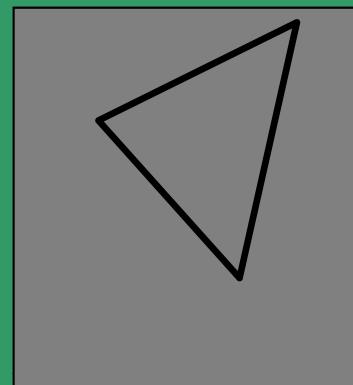
Surrounding



Intersecting



Contained

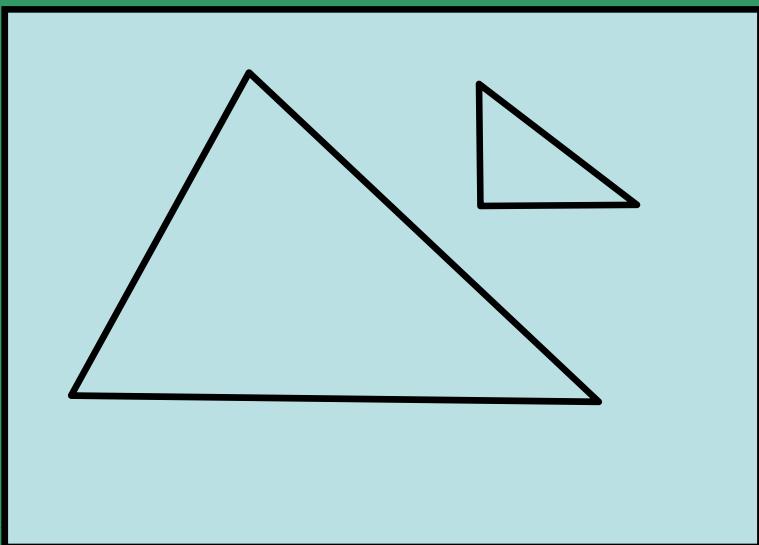


Disjoint

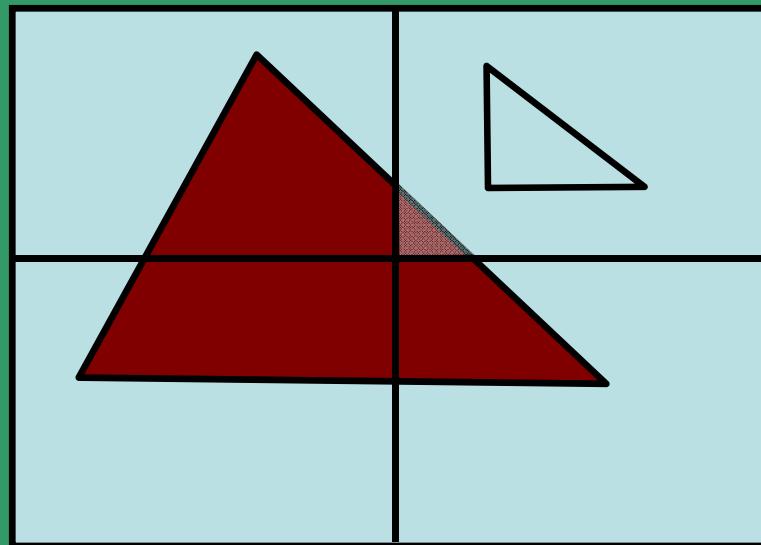
# **Case Studies – examples of**

## **Area Sub-division method**

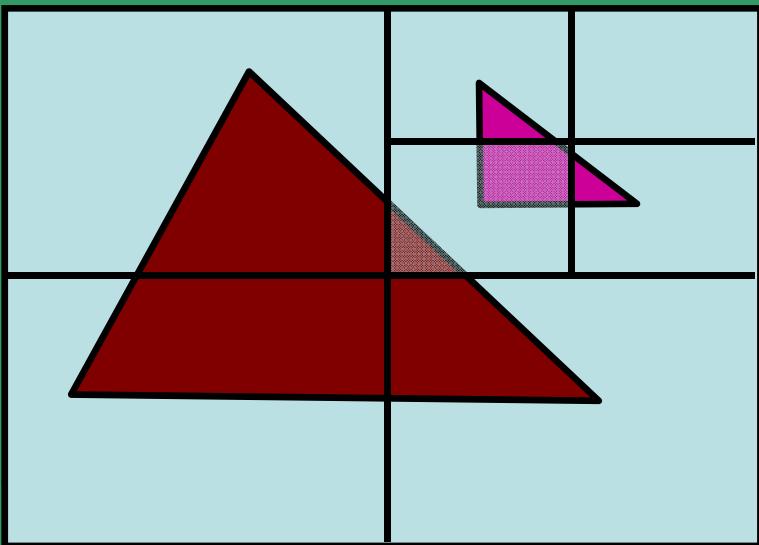
(i)



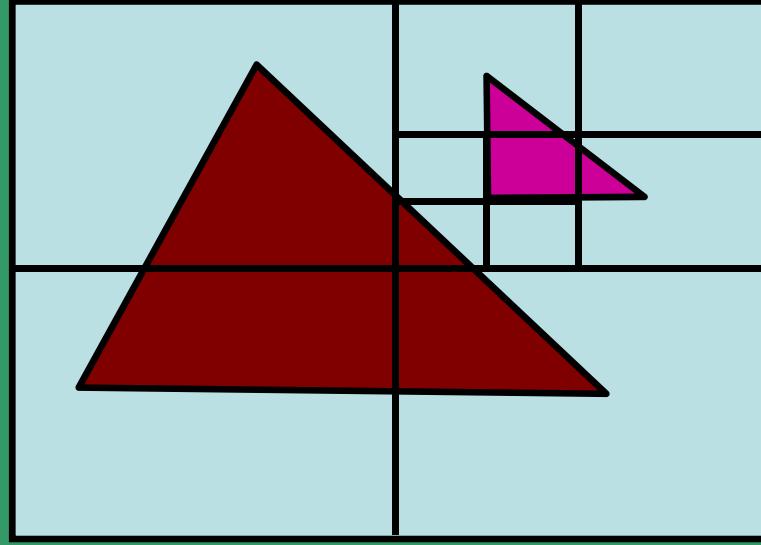
(ii)

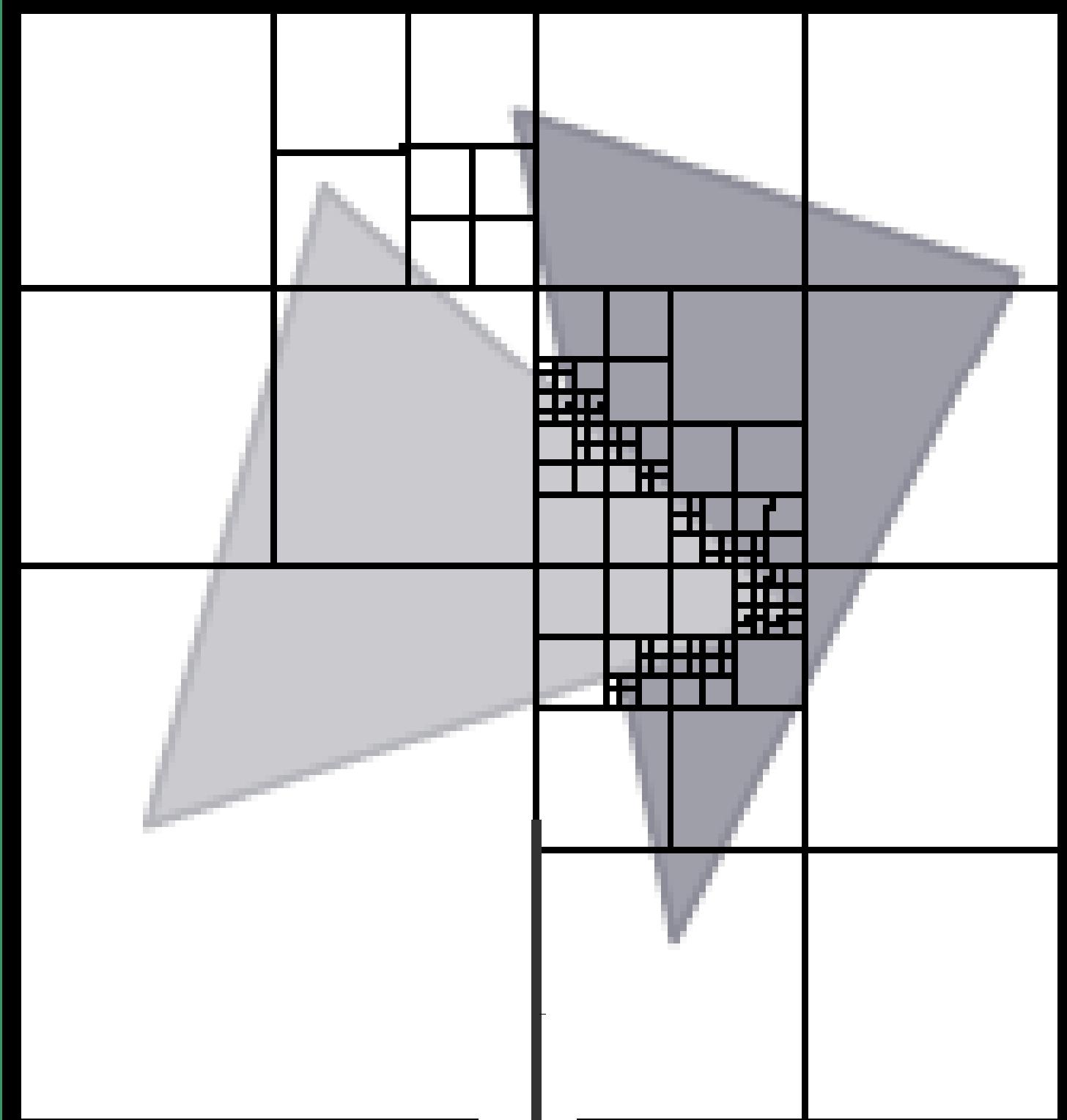


(iii)

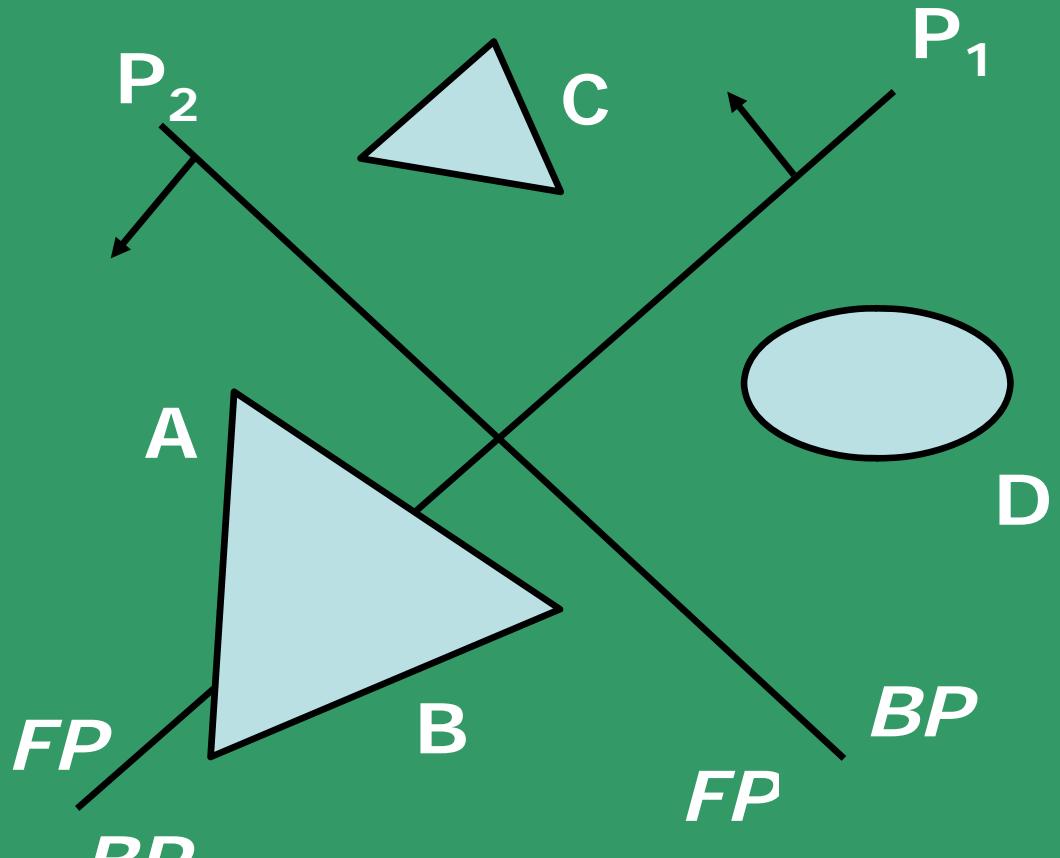


(iv)

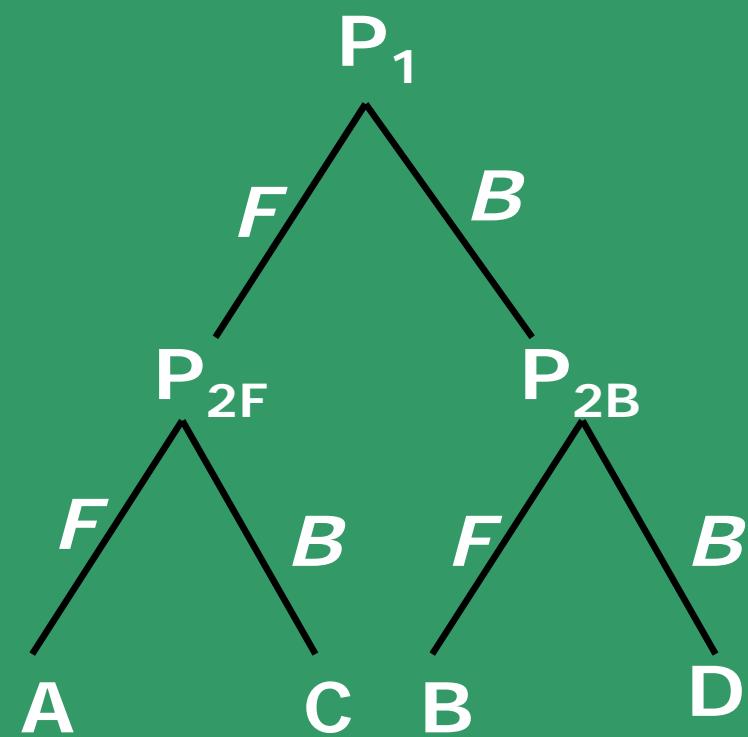




## BSP (Binary Space Partition) trees



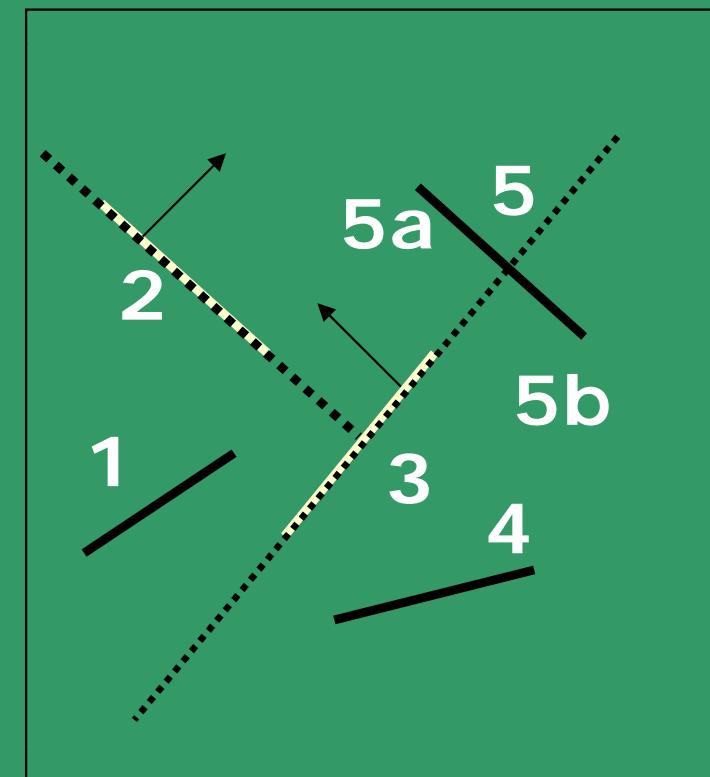
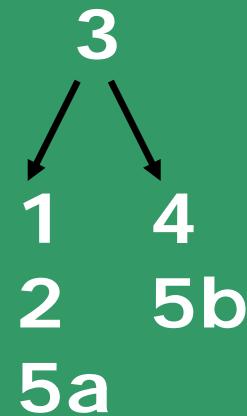
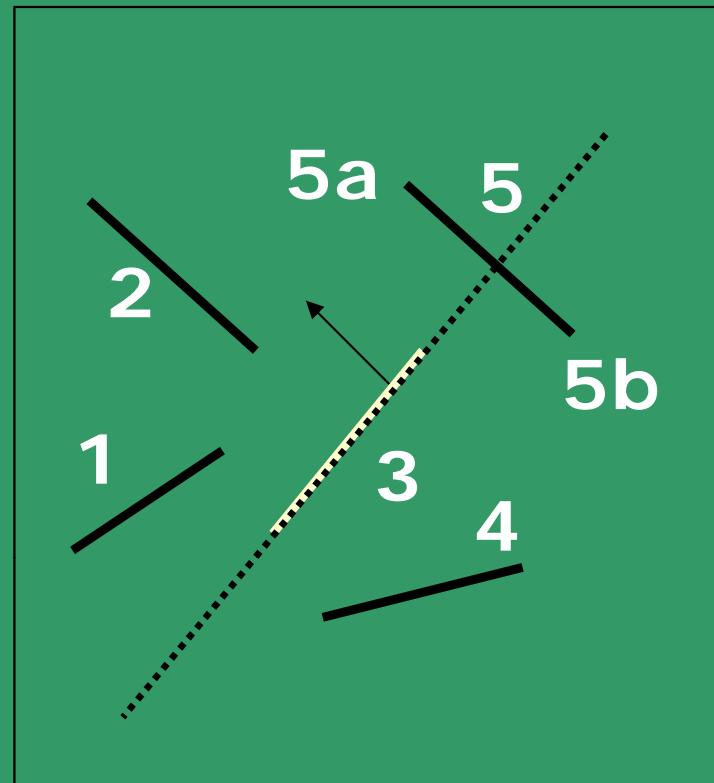
BSP (Binary Space  
Partition) trees

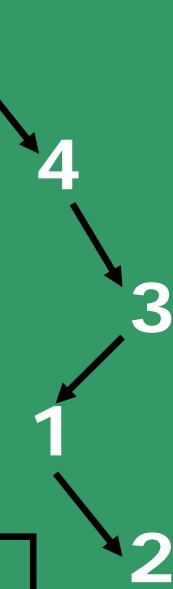
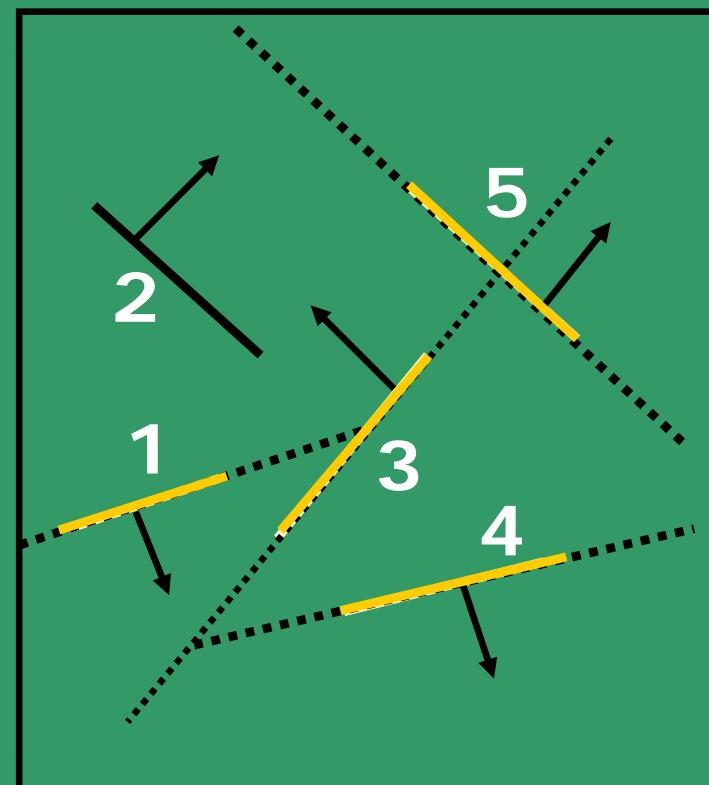
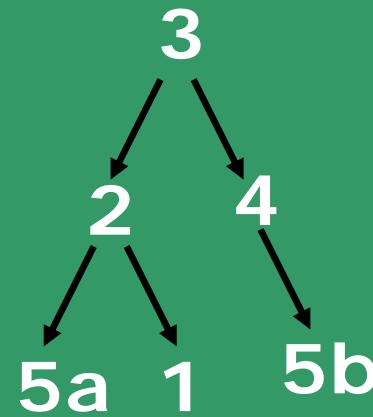
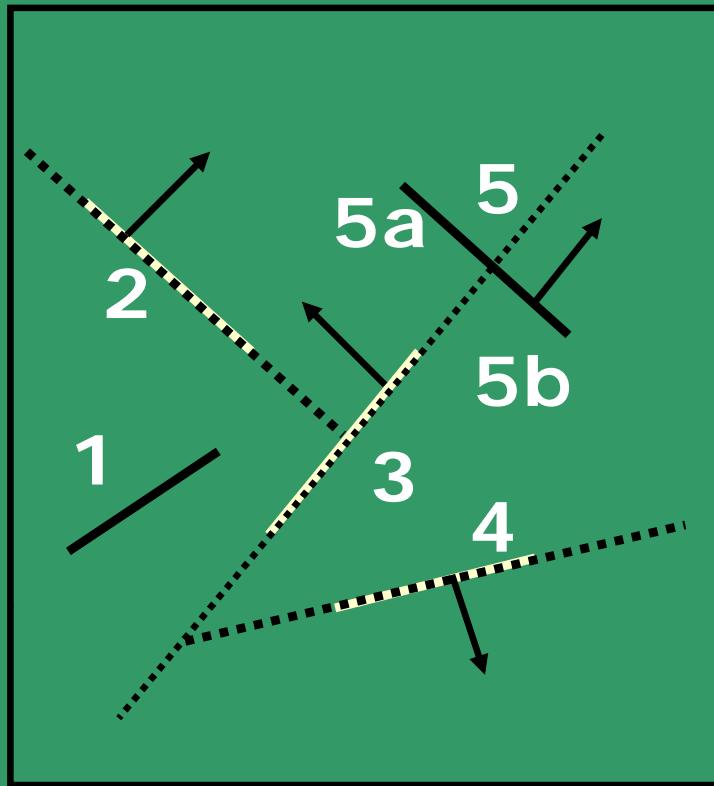


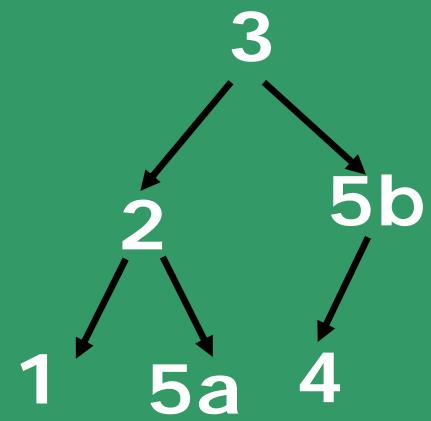
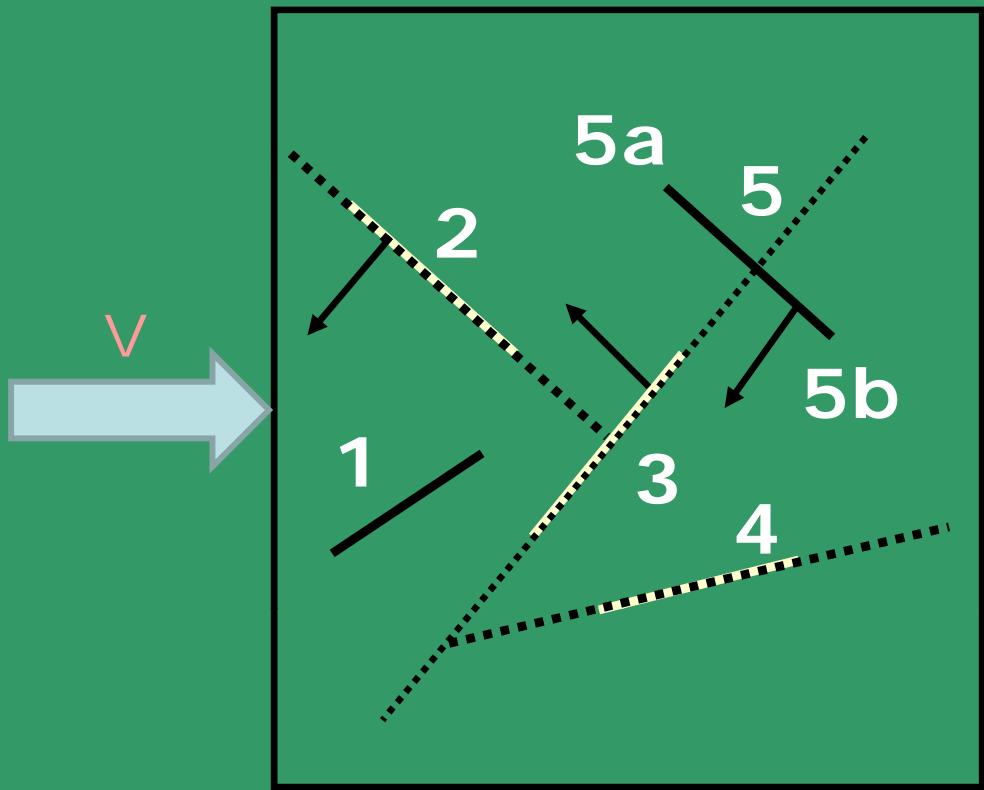
## Salient features:

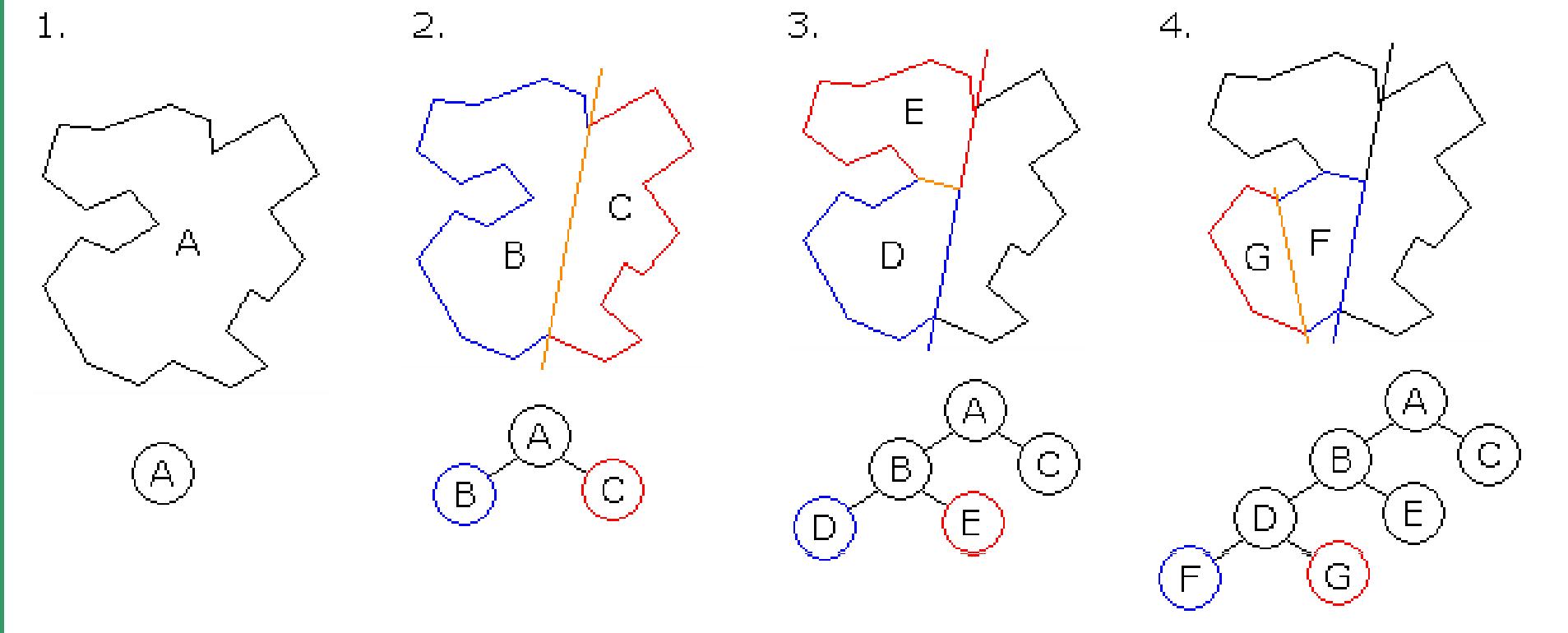
- Identify surfaces that are inside/front and outside/back w.r.t. the partitioning plane at each step of the space division, relative to the viewing direction.
- Start with any plane and find one set of objects behind and the rest in the front.
- In the tree, the objects are represented as terminal nodes with front objects as left branches and back objects as right branches.

- **BSP tree's root is a polygon selected from those to be displayed.**
- **One polygon each from the root polygon's front and back half plane, becomes its front and back children**
- **The algorithm terminates when each node contains only a single polygon.**
- **Intersection and sorting at object space/precision**









**Also see:**

**potential visibility sets;**

**KD trees, R+-trees;**

**Bounding Volume Hierarchy (BVH)**

and

**Shadow Volume BSP Tree (SVBSP)**

**Other applications:**

- Robot navigation
- Collision Detection
- GIS
- Image Registration

## Display list of a BSP tree

- BSP tree may be traversed in a “modified in-order tree” walk to yield a correctly priority-ordered polygon list for an arbitrary viewpoint.
- If the viewer is in the root polygon’s front half space, the algorithm must first display:
  1. All polygons in the root’s rear half-space.
  2. Then the root.
  3. And finally all polygons in the front half-space.
- Use back face culling
- Each of the root’s children is recursively processed.

## Disadvantages

- more polygon splitting may occur than in Painter's algorithm
- appropriate partitioning hyperplane selection is quite complicated and difficult

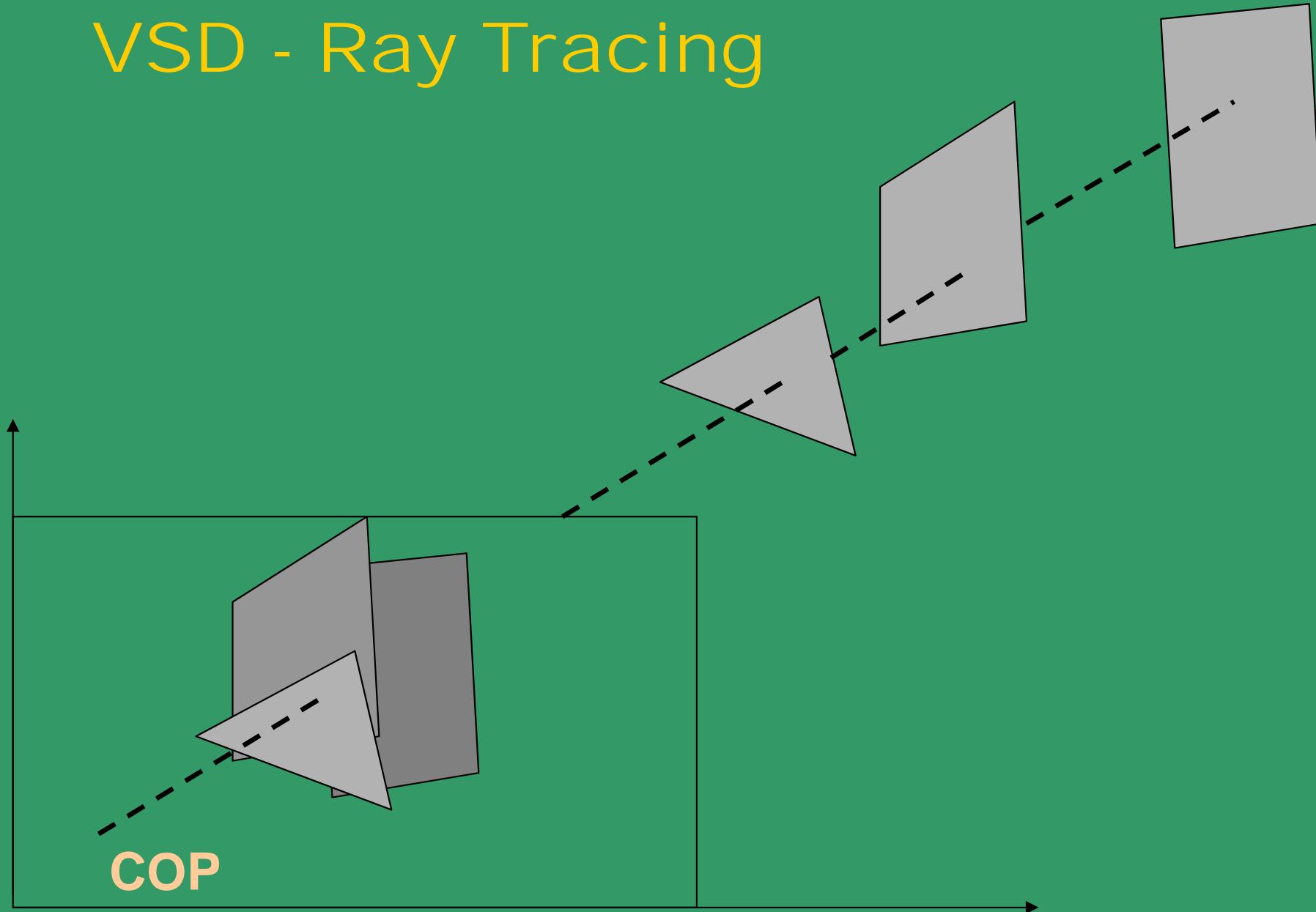
### Finding Optimal BSP:

Finding an optimal root node is by testing a small number of candidates.

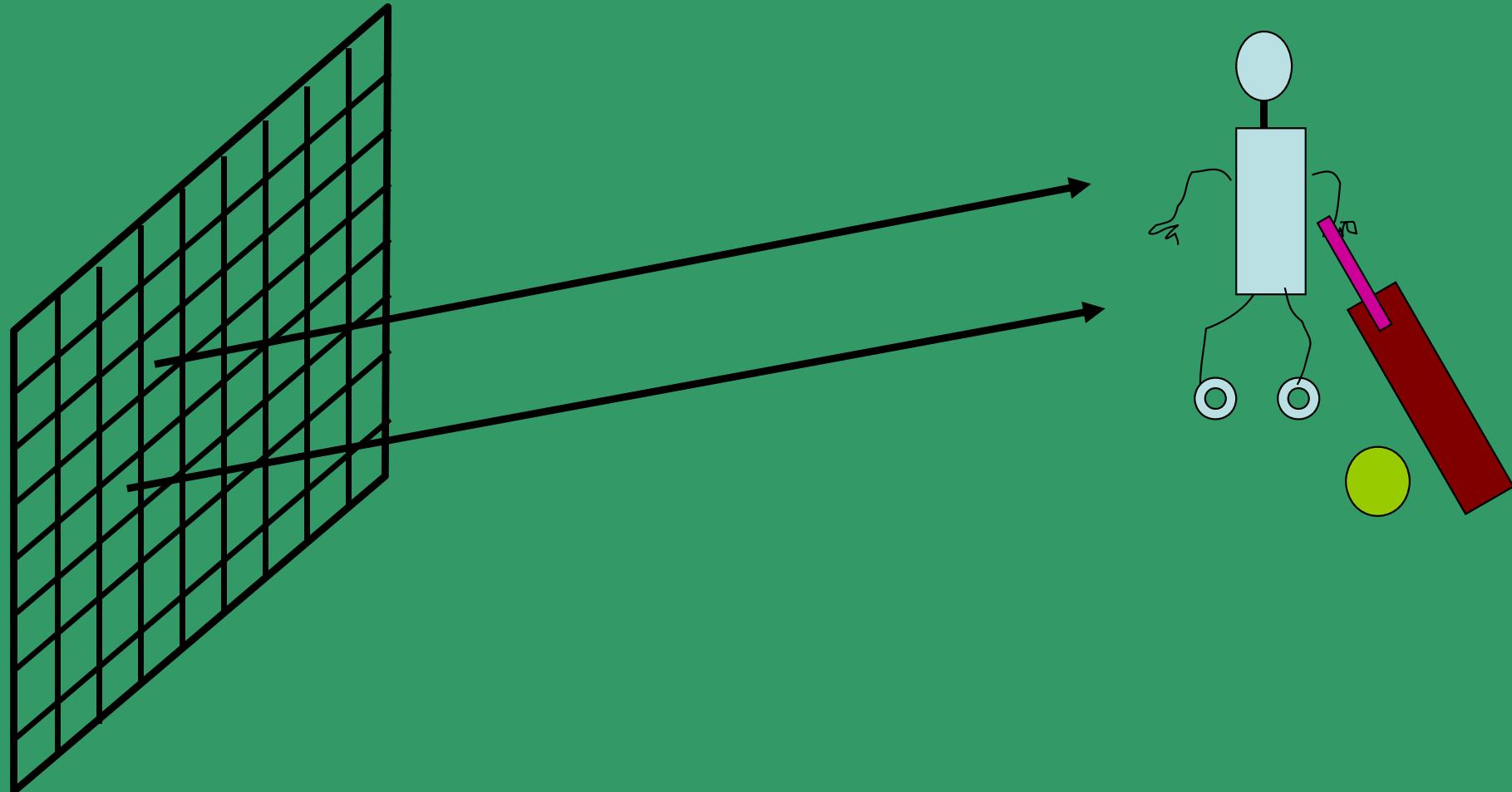
The node that results in the smallest number of nodes in the BSP tree should be chosen



# VSD - Ray Tracing



# Figure for illustrating VSD - Ray Tracing



**In reality, possibilities with a light ray:**  
**absorption, reflection, refraction, scattering, fluorescence or chromatic aberration (error in lens).**

**Remember? For Z-buffer:**

Each  $(X, Y, Z)$  point on a polygon surface, corresponds to the orthographic projection point  $(X, Y)$  on the view plane.

At each point  $(X, Y)$  on the PP, object depths are compared by using the depth( $Z$ ) values.

**For Ray-tracing:**

- Shoot ray from eye point through pixel  $(x, y)$  into scene
- Intersect with all surfaces, find first one the ray hits

## For Ray-tracing (cont'd):

- Line of Sight of each pixel is intersected with all surfaces
- Shade that point to compute the color of pixel  $(x,y)$
- Consider only the closest surface for shading
- Based on optics of image formation, paths of light rays are traced
- Light rays are traced backward

## For Ray-tracing (cont'd):

- Suitable for complex curved surfaces
- Computationally expensive
- Need of an efficient ray-surface intersection technique
- Almost all visual effects can be generated
- Can be parallelized
- Has aliasing problems

**Mathematically, the intersection problem is of finding the roots of an equation:**

**Surface – RAY = 0;**

**Ray Equation:  $r(t) = t(P - C)$**

**Eqns. for**

**LINE:**  $x = x_0 + t\Delta x; y = y_0 + t\Delta y; z = z_0 + t\Delta z;$

**SPHERE:**  $(x-a)^2 + (y-b)^2 + (z-c)^2 = r^2$

**Substitution  
gives us:**

$$\begin{aligned}(x_0 + t\Delta x)^2 - 2a(x_0 + t\Delta x) + a^2 \\ + (y_0 + t\Delta y)^2 - 2b(y_0 + t\Delta y) + b^2 \\ + (z_0 + t\Delta z)^2 - 2c(z_0 + t\Delta z) + c^2 = r^2\end{aligned}$$

**Collecting terms  
gives us:**

$$\begin{aligned}(\Delta x^2 + \Delta y^2 + \Delta z^2)t^2 + \\ 2t[\Delta x(x_0 - a) + \Delta y(y_0 - b) + \Delta z(z_0 - c)] \\ + (x_0 - a)^2 + (y_0 - b)^2 + (z_0 - c)^2 - r^2 = 0\end{aligned}$$

**The Equation is Quadratic, with coefficients from the constants of sphere and ray equations.**

Cases:

- No real roots - Surface and ray do not intersect
- One real root - Ray tangentially grazes the surface
- Two real roots - From both intersections, get the one with the smallest value of t.

**What is the shade, at that point of intersection?**

**Sphere has center  $(a, b, c)$ .**

**The surface normal at any point of intersection is:**

$$\left[ \frac{x_p - a}{r}, \frac{y_p - b}{r}, \frac{z_p - c}{r} \right]$$

**What about intersection with other (regular) non-linear surfaces ??**

- Gaussian
- Ellipsoid
- Directional sinusoid
- Torus

Eqns. for

LINE :  $x = x_0 + t\Delta x; y = y_0 + t\Delta y; z = z_0 + t\Delta z;$

PLANE :  $Ax + By + Cz + D = 0$

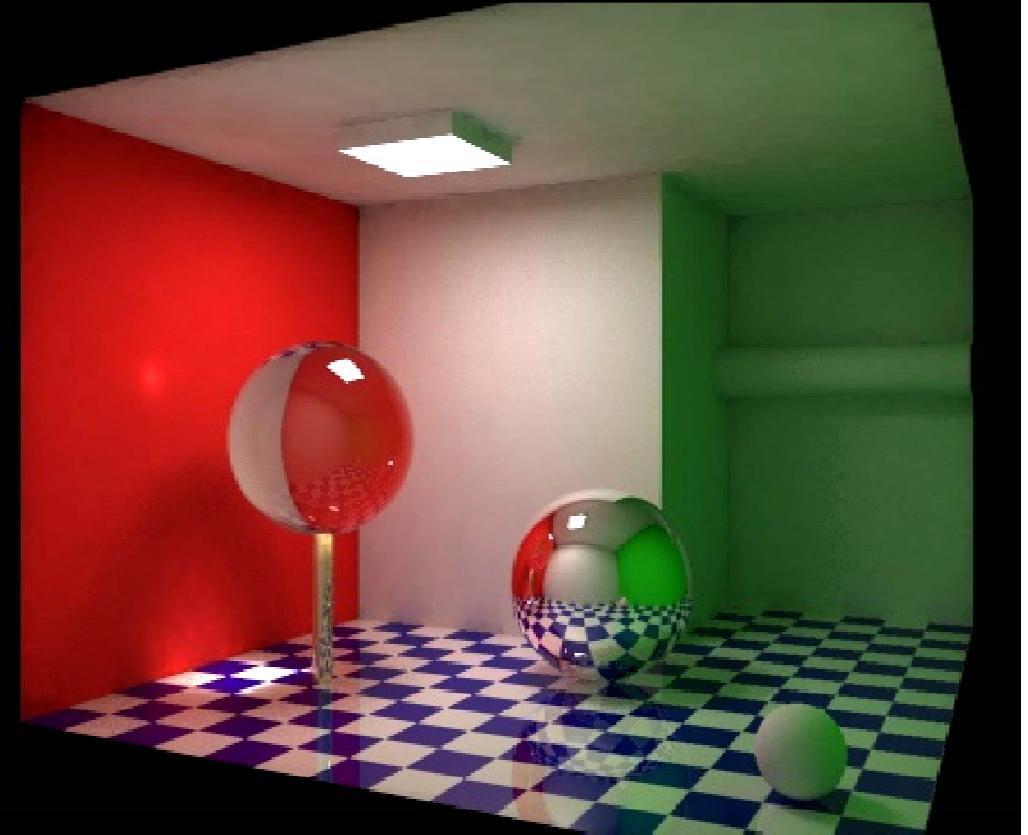
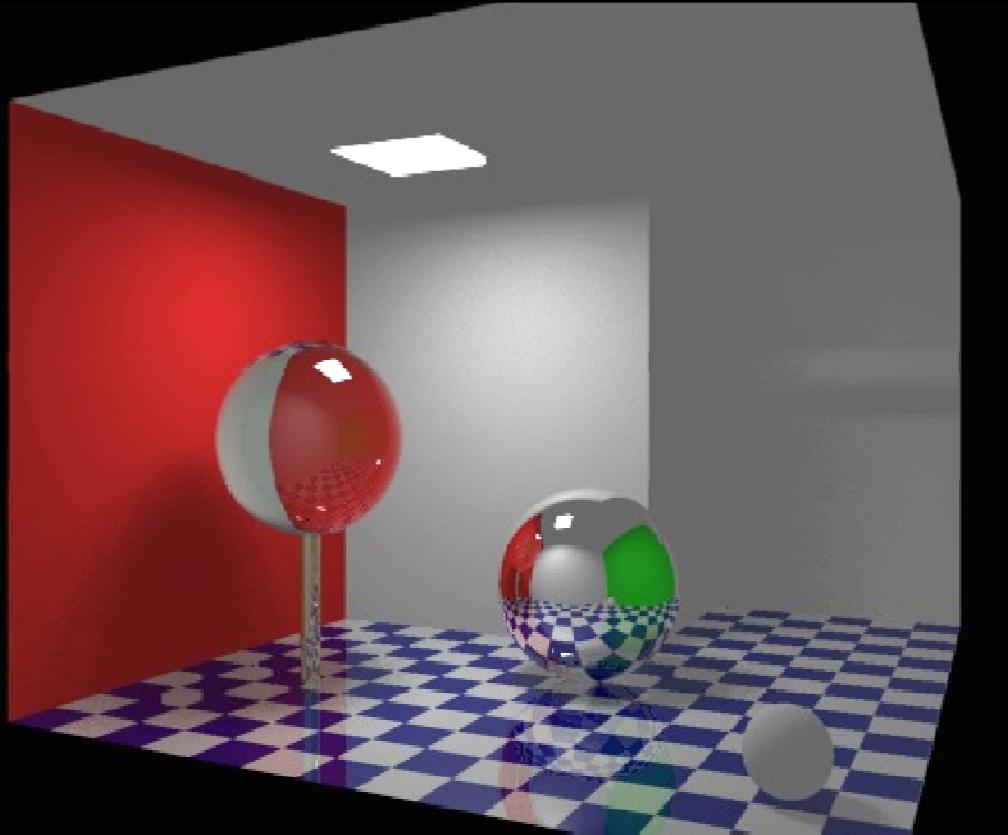
Substitution gives us:

$$t = -\frac{(Ax_0 + By_0 + Cz_0 + D)}{(A\Delta x + B\Delta y + C\Delta z)}$$

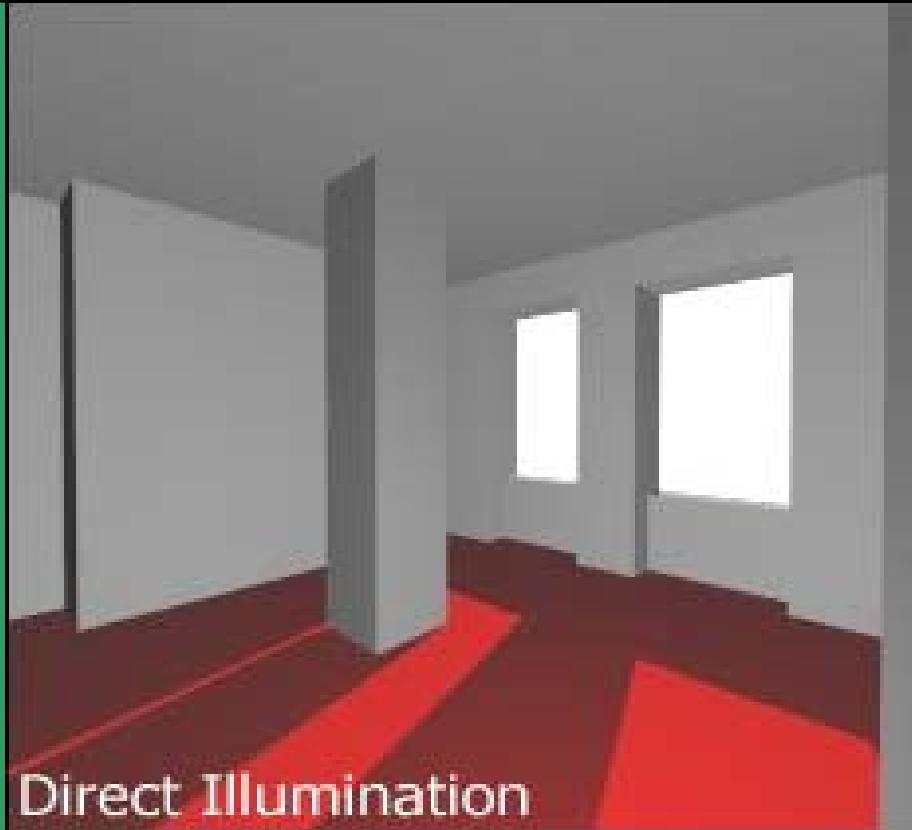
- Denominator should not be zero;
- Find if intersection is within the polygon, by projecting onto a suitable coordinate plane;
- Overall processing is ray-wise, not polygon-wise.

Read about:

- Stochastic/Distributed Ray Tracing (Monte Carlo on Graphics)
- Global Illumination
- Radiosity
- The Metropolis light transport (MLT) - SIGGRAPH 1997
- Bi-directional Ray Tracing



**Rendering with global illumination:** Light is reflected by surfaces, and colored light transfers from one surface to another. Color from the red wall and green wall (latter not visible) reflects onto other surfaces in the scene. Also notable is the caustic projected onto the red wall from light passing through the glass sphere.



Direct Illumination



Radiosity



## Comparison of VSD (HSR) techniques

Algorithms/ Methods	Memory	Speed
Z-Buffer	Two arrays	Depth complexity
Painter's	One array	Apriori sorting helps speed-up
Ray casting	Object data base	$O(\#pixels, \#surfaces \text{ or } objects)$
Scanline, Area sub-division	-	Slowest

Algorithms /Methods	Issues in Implementation	Remarks
Z-Buffer	Scan conversion, Hardware	Commonly used
Painter's	Scan conversion	Splitting and sorting the major bottleneck
Ray casting	Spatial data structures help speedup	Excellent for CSG, shadows, transparency
Scanline, Area sub-division	Hard	Cannot be generalized for non-polygonal models.

**VSD - “Visible Surface Detection”**  
is also called:

**Hidden Surface Elimination (HSE); Also: HSR, OC.**

**So, what is HLE:**

**Hidden Line Elimination.**

**Problem:**

**Take any VSD algorithm and  
convert to an HLE algorithm**

End of lectures on

VISIBLE SURFACE  
DETECTION

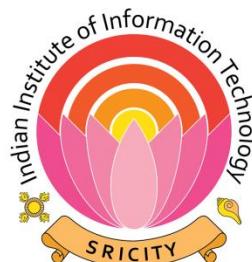


# **Computer Graphics and Multimedia**

## **Light: Radiometry and Reflectance**

**Dr. Mrinmoy Ghorai**

**Indian Institute of Information Technology  
Sri City, Chittoor**



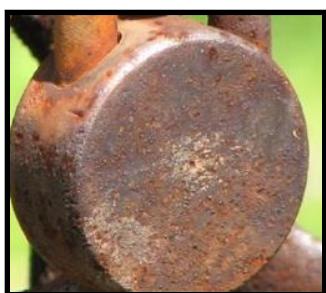
# Today's Agenda

- Light
  - Radiometry
  - Reflectance

# Why should we care?

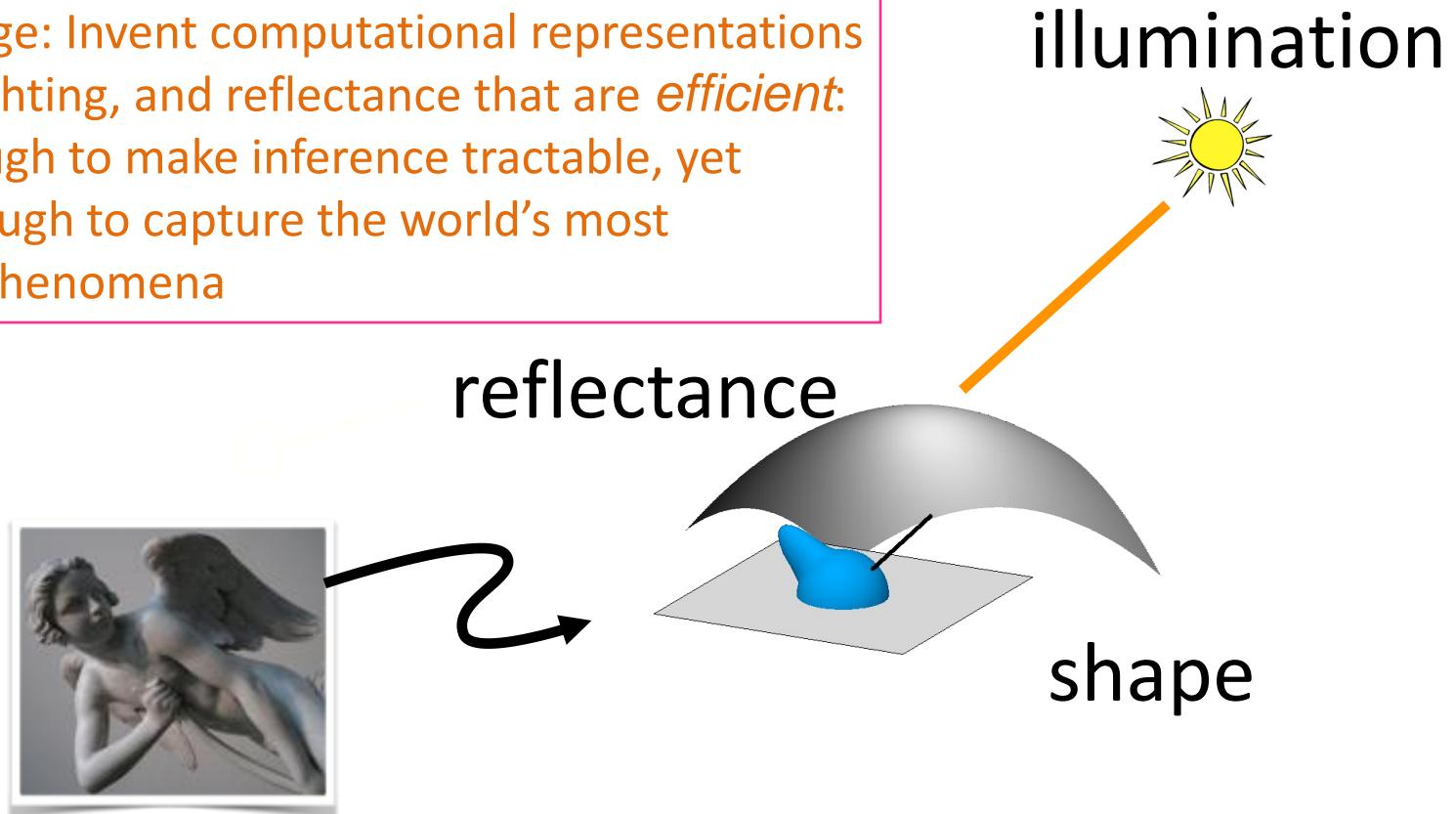
- The **appearance** of objects is given by the way in which they **reflect** and transmit **light**.
- The **color** of objects is determined by the parts of the **spectrum of** (incident white) **light** that are reflected or transmitted without being absorbed.

# Appearance



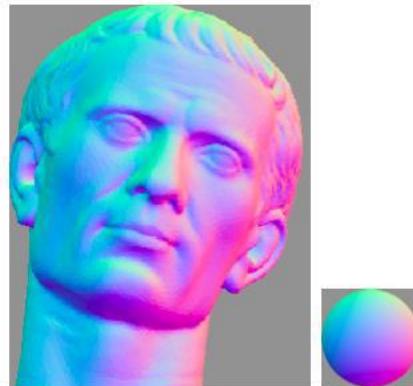
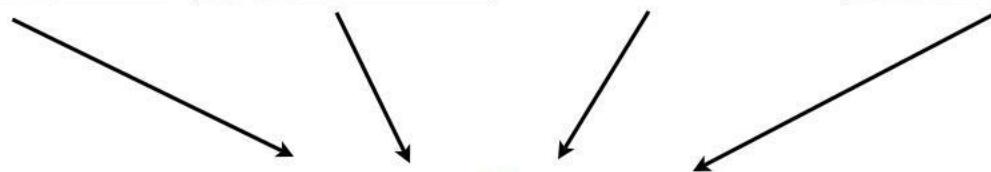
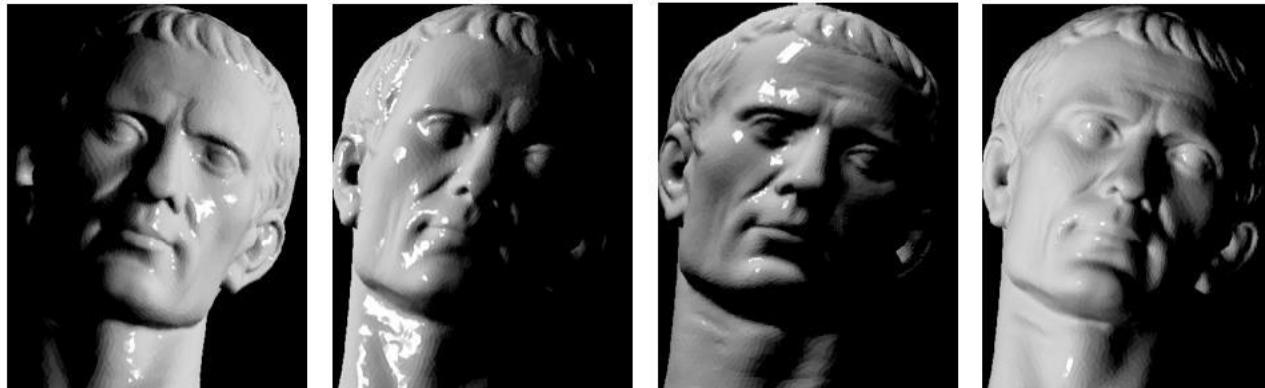
# “Physics-based” computer vision (a.k.a “inverse optics”)

Our challenge: Invent computational representations of shape, lighting, and reflectance that are *efficient*: simple enough to make inference tractable, yet general enough to capture the world’s most important phenomena



$I \rightarrow \text{shape, illumination, reflectance}$

# Application: Photometric Stereo



Analysis under different lighting conditions to estimate a normal direction at each pixel.

- Why study the physics (optics) of the world?
- Lets see some pictures!

# Light and Shadows



# Light and Shadows



# Reflections



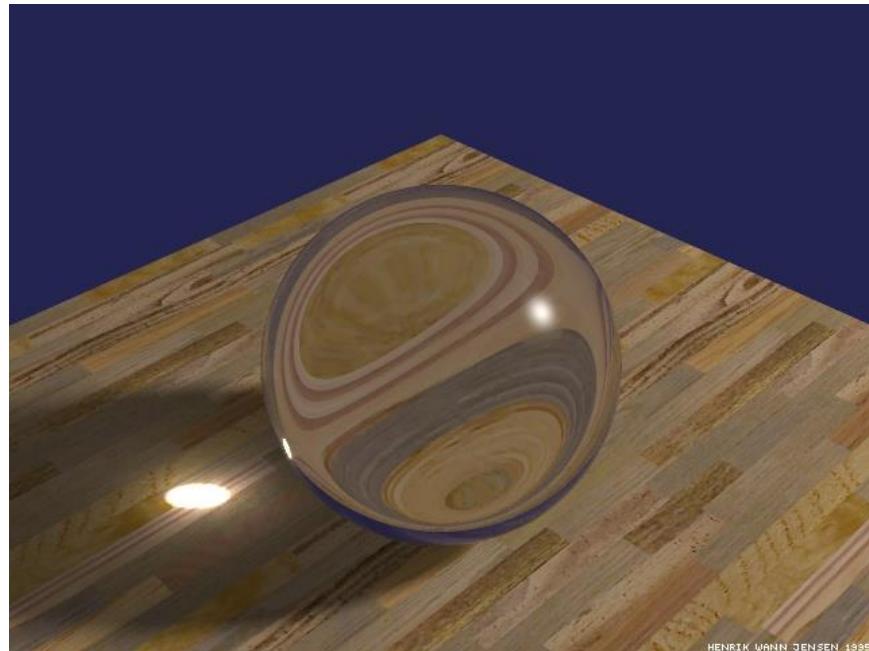
# Reflections



# Refractions



HENRIK WANN JENSEN 1995



HENRIK WANN JENSEN 1995

# Refractions

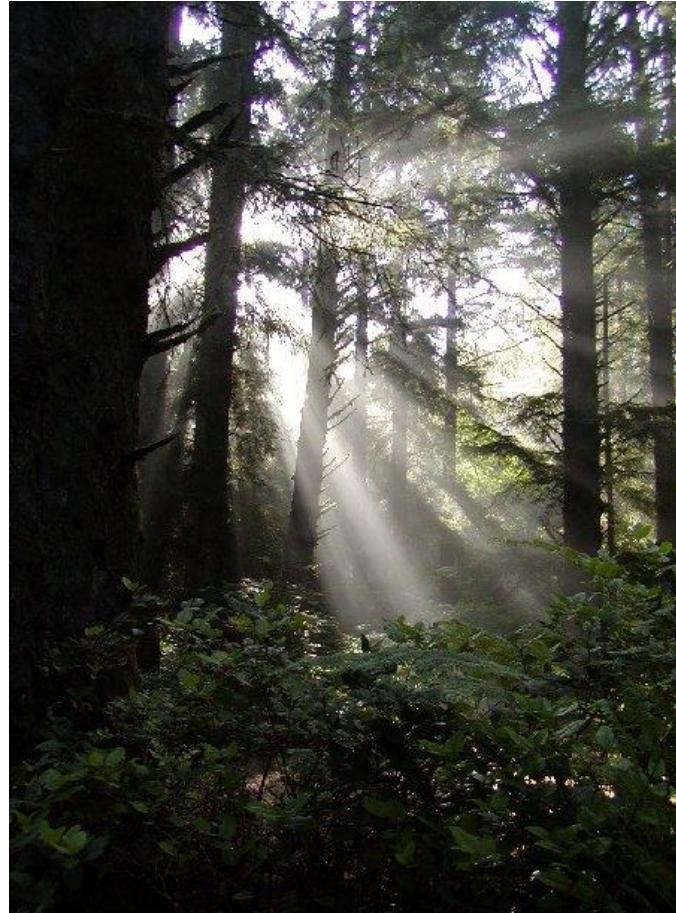


# Inter-reflections

Mies Courtyard House with Curved Elements



# Scattering



# Scattering



# More Complex Appearances

# More Complex Appearances

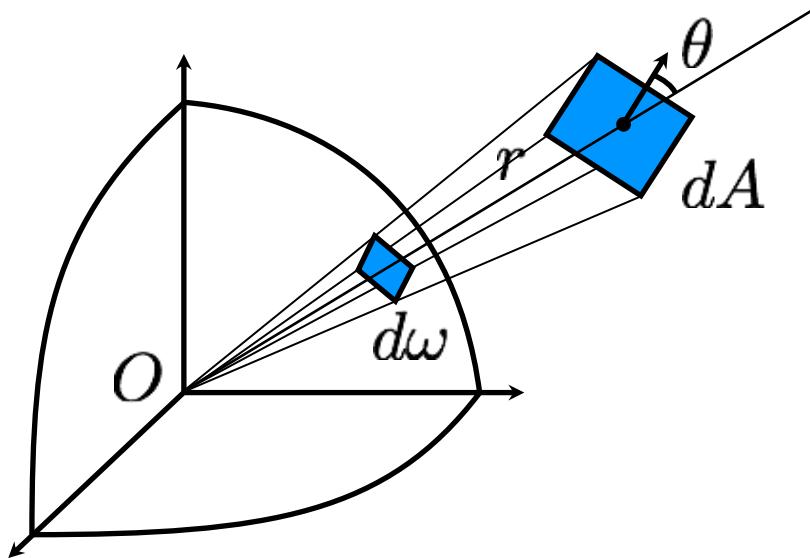


# More Complex Appearances



# Measuring Light and Radiometry

- **Solid angle:** The *solid angle* subtended by a small surface patch with respect to point O is the area of its central projection onto the unit sphere about O



Depends on:

- orientation of patch
- distance of patch

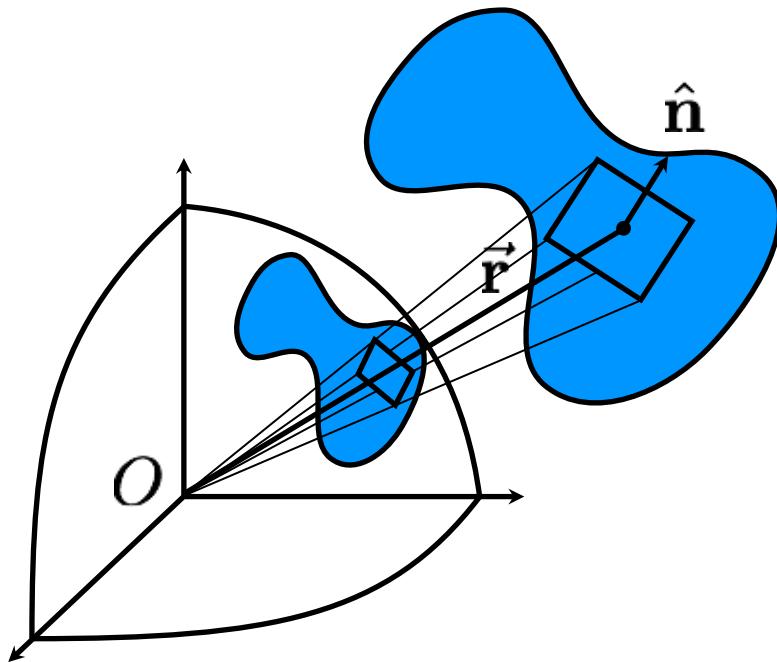
One can show:

$$d\omega = \frac{dA \cos \theta}{r^2}$$

Units: steradians [sr]

# Measuring Light and Radiometry

- To calculate solid angle subtended by a surface  $S$  relative to  $O$  you must add up (integrate) contributions from all tiny patches (nasty integral)



$$\Omega = \iint_S \frac{\vec{r} \cdot \hat{n} \, dS}{|\vec{r}|^3}$$

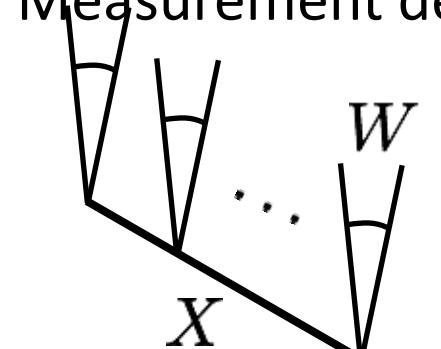
One can show:

$$d\omega = \frac{dA \cos \theta}{r^2}$$

Units: steradians [sr]

# Quantifying light: flux, irradiance, and radiance

- Imagine a sensor that counts photons passing through planar patch  $X$  in directions within angular wedge  $W$
- It measures *radiant flux* [watts = joules/sec]: rate of photons hitting sensor area
- Measurement depends on sensor area  $|X|$

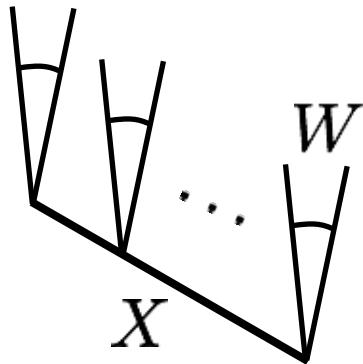


\* shown in 2D for clarity; imagine three dimensions

radiant flux  $\Phi(W, X)$

# Quantifying light: flux, irradiance, and radiance

- *Irradiance:*
  - A measure of incoming light that is independent of sensor area  $|X|$
- Units: watts per square meter [ $\text{W/m}^2$ ]



$$\frac{\Phi(W, X)}{|X|}$$

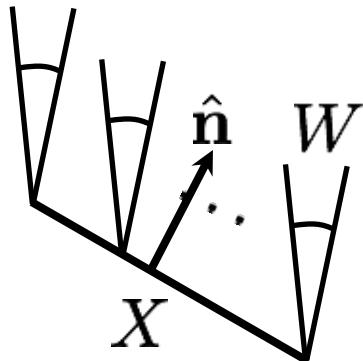
# Quantifying light: flux, irradiance, and radiance

- *Irradiance:*
  - A measure of incoming light that is independent of sensor area  $|X|$
- Units: watts per square meter [ $\text{W/m}^2$ ]

The diagram shows a triangular region labeled  $X$  representing a sensor area. Several smaller triangles, each containing a symbol resembling a light bulb, are shown within  $X$ . An ellipsis indicates that there are more triangles. To the right, the expression  $\lim_{X \rightarrow x}$  is written above a fraction: 
$$\frac{\Phi(W, X)}{|X|}$$

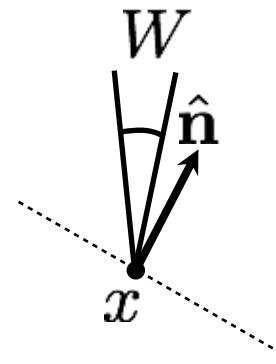
# Quantifying light: flux, irradiance, and radiance

- *Irradiance:*  
A measure of incoming light that is independent of sensor area  $|X|$
- Units: watts per square meter [ $\text{W/m}^2$ ]
- Depends on sensor direction normal.



$$\frac{\Phi(W, X)}{|X|}$$

$$\lim_{X \rightarrow x}$$



$$E_{\hat{n}}(W, x)$$

- We keep track of the normal because a planar sensor with distinct orientation would converge to a different limit
- In the literature, notations  $n$  and  $W$  are often omitted, and values are implied by context

# Quantifying light: flux, irradiance, and radiance

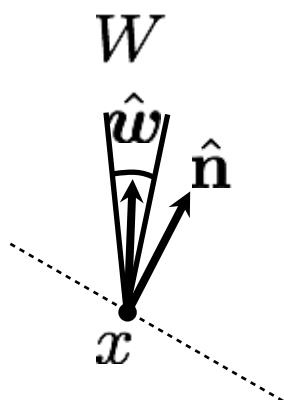
- *Radiance:*
  - A measure of incoming light that is independent of sensor area  $|X|$ , orientation  $\hat{n}$ , and wedge size (solid angle)  $|W|$
- Units: watts per steradian per square meter [ $W/(m^2 \cdot sr)$ ]

$$\frac{E_{\hat{n}}(W, x)}{|W|} \quad L_{\hat{n}}(\hat{\omega}, x)$$

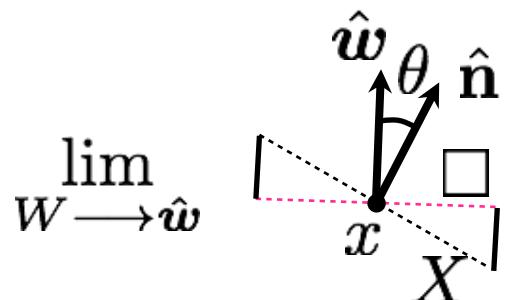
- Has correct units, but still depends on sensor orientation
- To correct this, convert to measurement that would have been made if sensor was perpendicular to direction  $\omega$

# Quantifying light: flux, irradiance, and radiance

- *Radiance:*  
A measure of incoming light that is independent of sensor area  $|X|$ , orientation  $\hat{n}$ , and wedge size (solid angle)  $|W|$
- Units: watts per steradian per square meter [ $W/(m^2 \cdot sr)$ ]



$$\frac{E_{\hat{n}}(W, x)}{|W|}$$



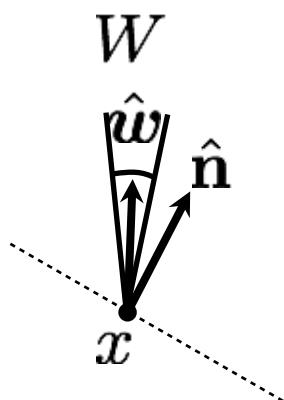
$$L_{\hat{n}}(\hat{w}, x)$$

$$\begin{aligned}\cos \theta &= \frac{\square/2}{|X|/2} \\ \rightarrow \square &= |X| \cos \theta \\ &\text{"foreshortened area"}\end{aligned}$$

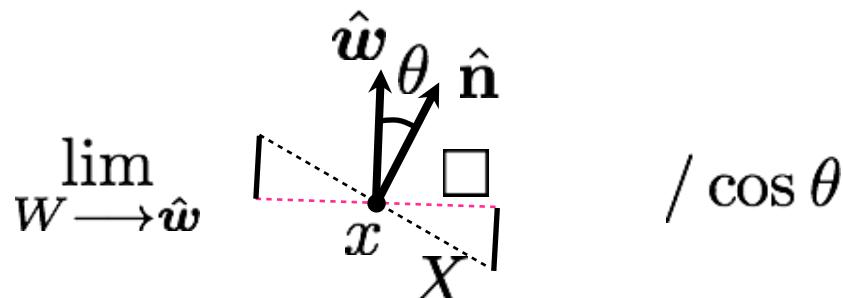
- Has correct units, but still depends on sensor orientation
- To correct this, convert to measurement that would have been made if sensor was perpendicular to direction  $\omega$

# Quantifying light: flux, irradiance, and radiance

- *Radiance:*
  - A measure of incoming light that is independent of sensor area  $|X|$ , orientation  $\hat{n}$ , and wedge size (solid angle)  $|W|$
- Units: watts per steradian per square meter [ $W/(m^2 \cdot sr)$ ]



$$\frac{E_{\hat{n}}(W, x)}{|W|}$$



$$L_{\hat{n}}(\hat{\omega}, x)$$

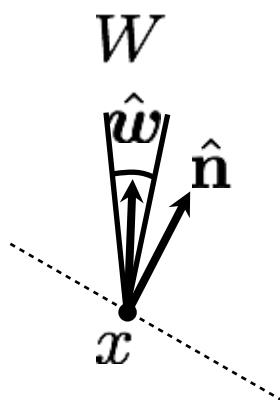


$$L(\hat{\omega}, x)$$

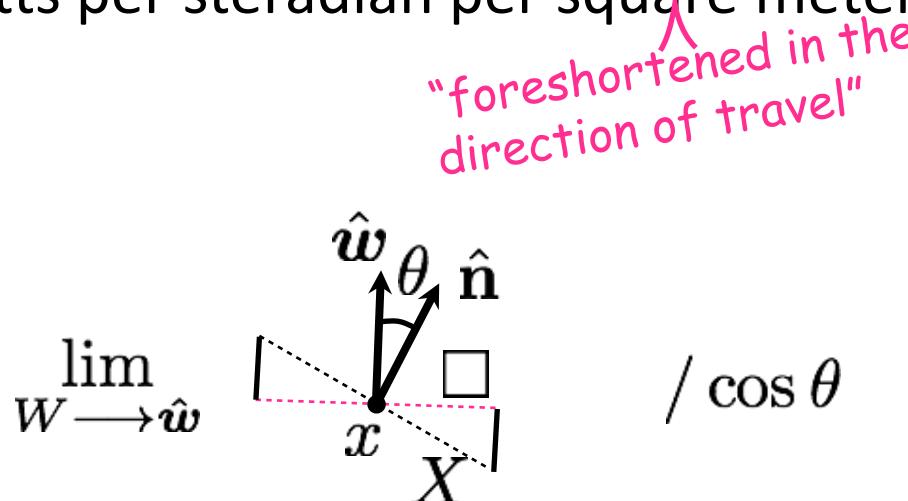
- Has correct units, but still depends on sensor orientation
- To correct this, convert to measurement that would have been made if sensor was

# Quantifying light: flux, irradiance, and radiance

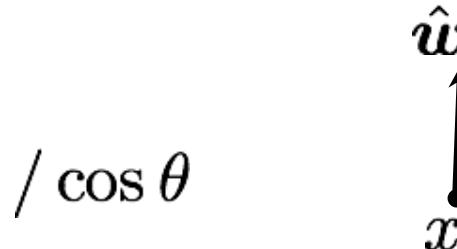
- *Radiance:*  
A measure of incoming light that is independent of sensor area  $|X|$ , orientation  $\hat{n}$ , and wedge size (solid angle)  $|W|$
- Units: watts per steradian per square meter  $[W/(m^2 \cdot sr)]$



$$\frac{E_{\hat{n}}(W, x)}{|W|}$$



$$L_{\hat{n}}(\hat{\omega}, x)$$

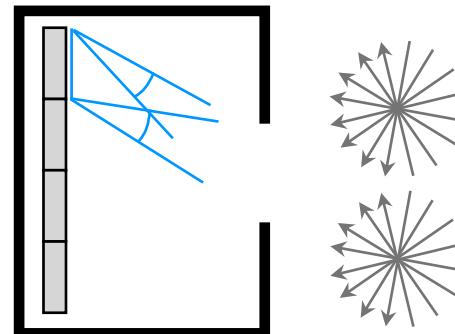


$$L(\hat{\omega}, x)$$

- Has correct units, but still depends on sensor orientation
- To correct this, convert to measurement that would have been made if sensor was perpendicular to direction  $\omega$

# Quantifying light: flux, irradiance, and radiance

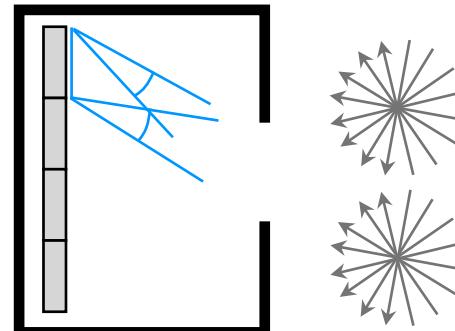
- Attractive properties of radiance:
  - Allows computing the radiant flux measured by *any* finite sensor



# Quantifying light: flux, irradiance, and radiance

- Attractive properties of radiance:
  - Allows computing the radiant flux measured by *any* finite sensor

$$\Phi(W, X) = \int_X \int_W L(\hat{\omega}, x) \cos \theta d\omega dA$$



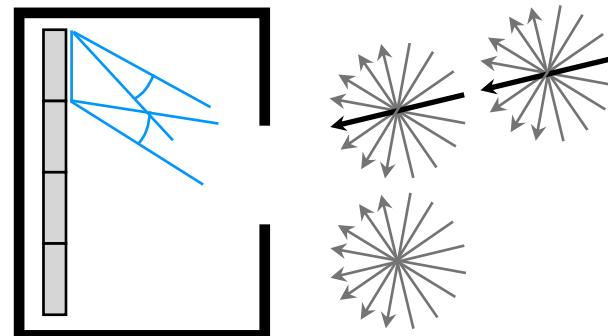
# Quantifying light: flux, irradiance, and radiance

- Attractive properties of radiance:
  - Allows computing the radiant flux measured by *any* finite sensor

$$\Phi(W, X) = \int_X \int_W L(\hat{\omega}, x) \cos \theta d\omega dA$$

- Constant along a ray in free space

$$L(\hat{\omega}, x) = L(\hat{\omega}, x + \hat{\omega})$$



# Quantifying light: flux, irradiance, and radiance

- Attractive properties of radiance:
  - Allows computing the radiant flux measured by *any* finite sensor

$$\Phi(W, X) = \int_X \int_W L(\hat{\omega}, x) \cos \theta d\omega dA$$

- Constant along a ray in free space

$$L(\hat{\omega}, x) = L(\hat{\omega}, x + \hat{\omega})$$

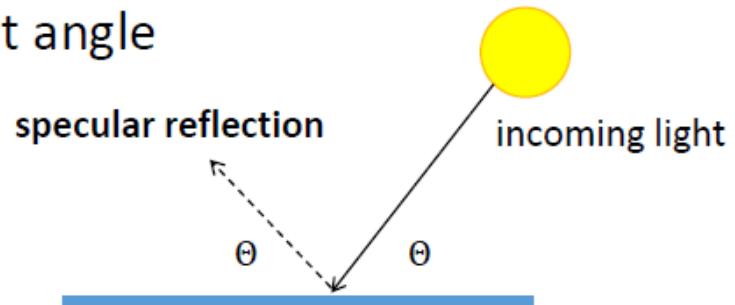
- A camera measures radiance (after a one-time radiometric calibration). So RAW pixel values are proportional to radiance.
  - “Processed” images (like PNG and JPEG) are not linear radiance measurements!!

# Reflectance and BRDF

# Basic models of reflection

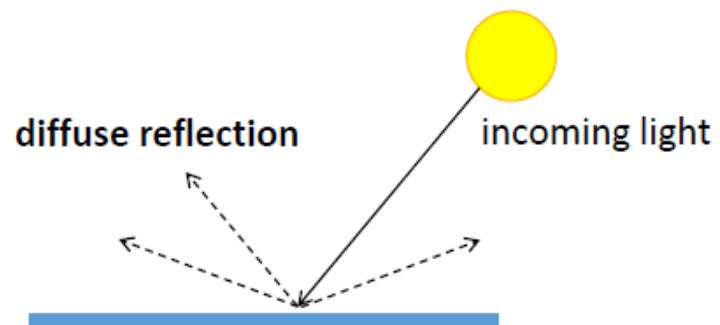
- Specular: light bounces off at the incident angle

- E.g., mirror



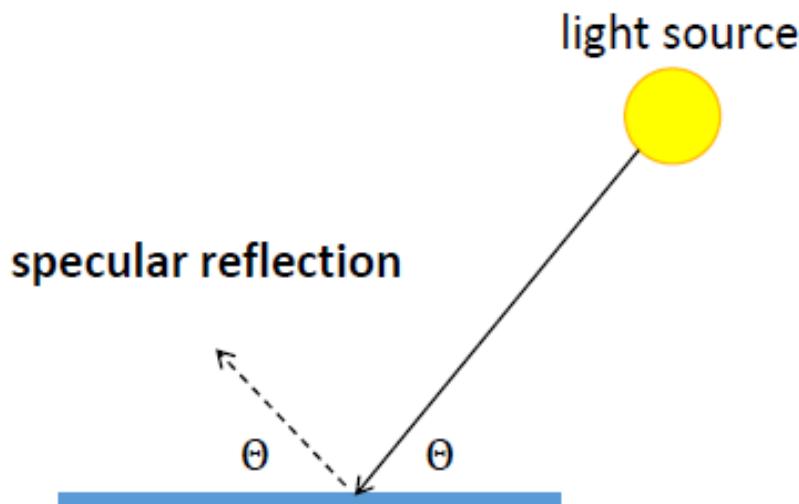
- Diffuse: light scatters in all directions

- E.g., brick, cloth, rough wood



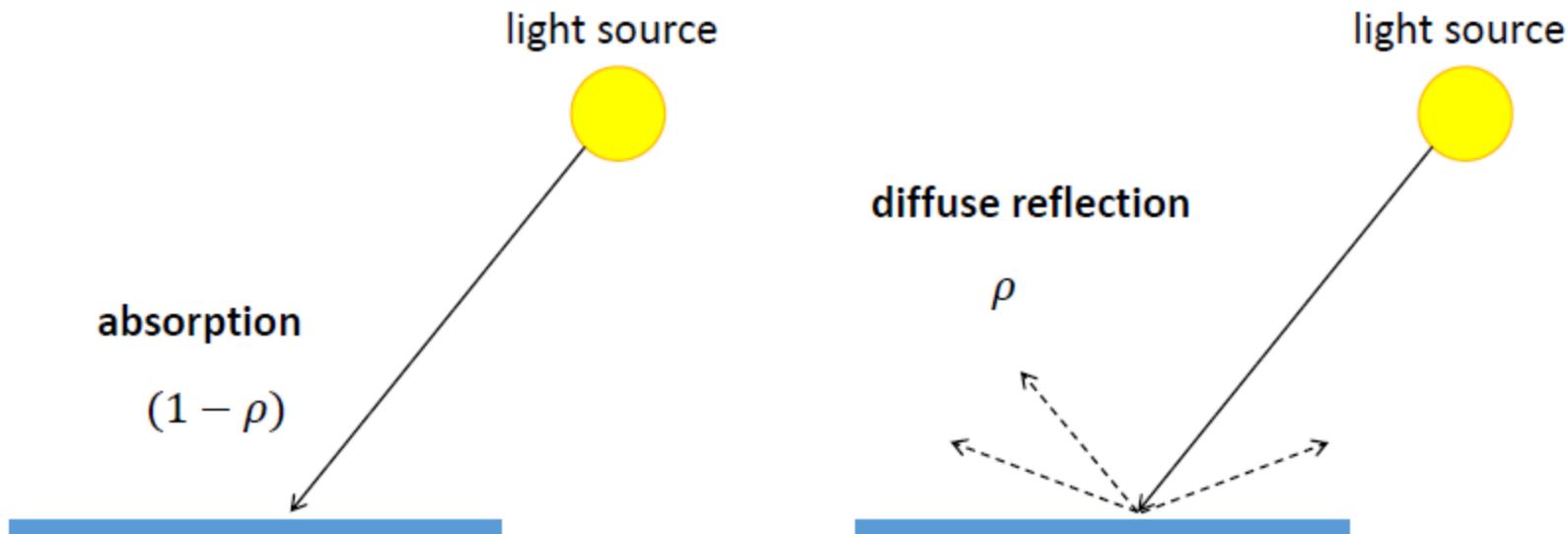
# Specular Reflection

- Reflected direction depends on light orientation and surface normal
  - E.g., mirrors are fully specular



# Lambertian reflectance model

- Some light is absorbed (function of albedo  $\rho$ )
- Remaining light is scattered (diffuse reflection)
- Examples: soft cloth, concrete, matte paints



# Most surfaces have both specular and diffuse components

- Specularity = spot where specular reflection dominates (typically reflects light source)



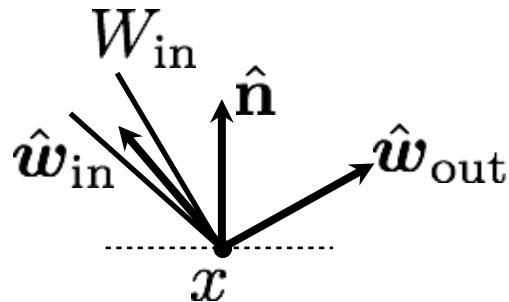
Typically, specular component is small

Slide credit: Derek Hoiem

Photo: northcountryhardwoodfloors.com

# Reflectance

- Ratio of outgoing energy to incoming energy at a single point
- Want to define a ratio such that it:
  - converges as we use smaller and smaller incoming and outgoing wedges
  - does not depend on the size of the wedges (i.e. is intrinsic to the material)

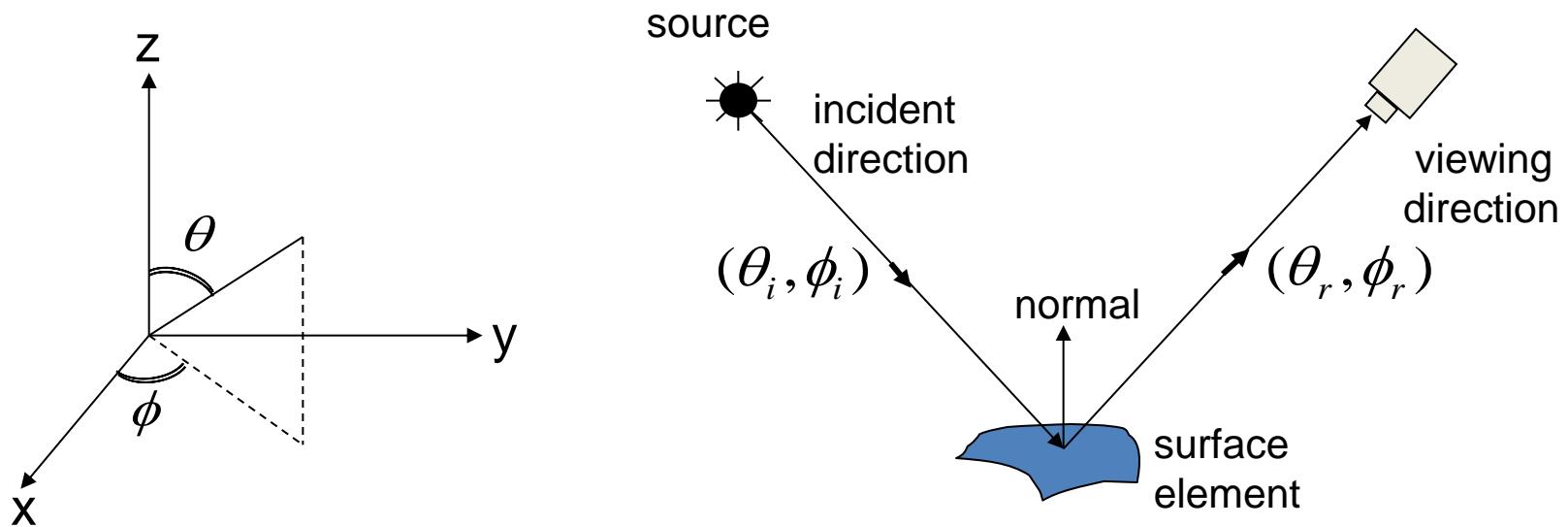


$$\lim_{W_{\text{in}} \rightarrow \hat{\omega}_{\text{in}}} f_{x,\hat{\mathbf{n}}}(\hat{\omega}_{\text{in}}, \hat{\omega}_{\text{out}})$$

$$f_{x,\hat{\mathbf{n}}}(W_{\text{in}}, \hat{\omega}_{\text{out}}) = \frac{L^{\text{out}}(x, \hat{\omega}_{\text{out}})}{E_{\hat{\mathbf{n}}}^{\text{in}}(W_{\text{in}}, x)}$$

- Notations x and n often implied by context and omitted; directions \omega are expressed in local coordinate system defined by normal n (and some chosen tangent vector)
- Units: sr<sup>-1</sup>
- Called Bidirectional Reflectance Distribution Function (BRDF)

# BRDF: Bidirectional Reflectance Distribution Function



$E^{surface}(\theta_i, \phi_i)$  Irradiance at Surface in direction  $(\theta_i, \phi_i)$

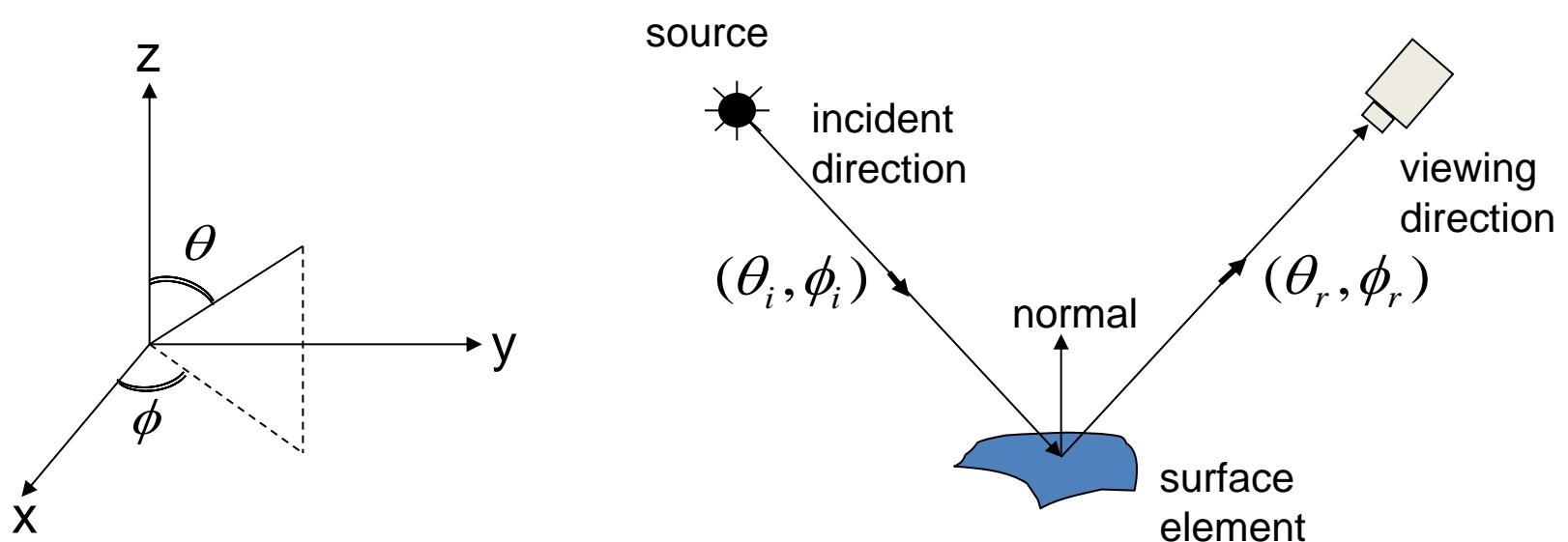
$L^{surface}(\theta_r, \phi_r)$  Radiance of Surface in direction  $(\theta_r, \phi_r)$

$$\text{BRDF : } f(\theta_i, \phi_i; \theta_r, \phi_r) = \frac{L^{surface}(\theta_r, \phi_r)}{E^{surface}(\theta_i, \phi_i)}$$

# Reflectance: BRDF

- Units:  $\text{sr}^{-1}$
- Real-valued function defined on the double-hemisphere
- Has many useful properties

# Important Properties of BRDFs

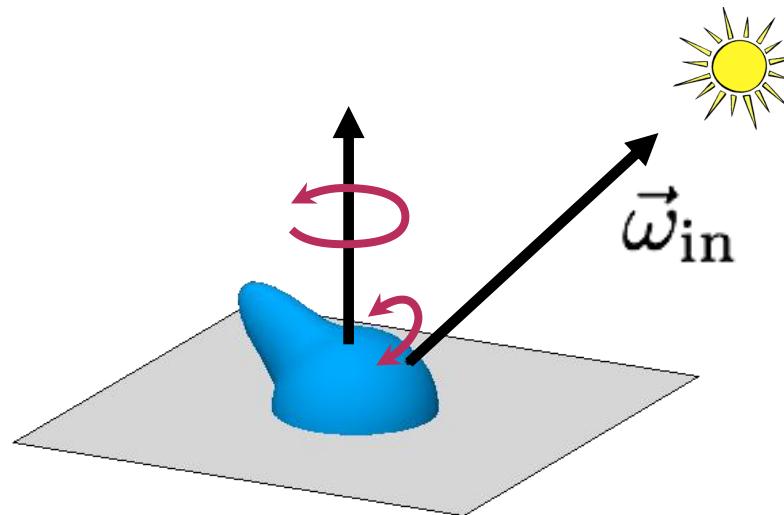
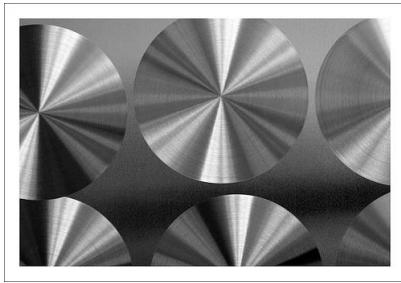


- Conservation of Energy:

$$\forall \hat{\omega}_{\text{in}}, \int_{\Omega_{\text{out}}} f(\hat{\omega}_{\text{in}}, \hat{\omega}_{\text{out}}) \cos \theta_{\text{out}} d\hat{\omega}_{\text{out}} \leq 1$$

Why smaller than or equal?

# Common assumption: Isotropy



$$f_r(\vec{\omega}_{in}, \cdot)$$

BRDF does not change  
when surface is rotated  
about the normal.

**4D → 3D**

$$f_r(\vec{\omega}_{in}, \vec{\omega}_{out})$$



[Matusik et al., 2003]

Bi-directional Reflectance Distribution Function (BRDF)

Can be written as a function of 3 variables :  $f(\theta_i, \theta_r, \phi_i - \phi_r)$

# Reflectance: BRDF

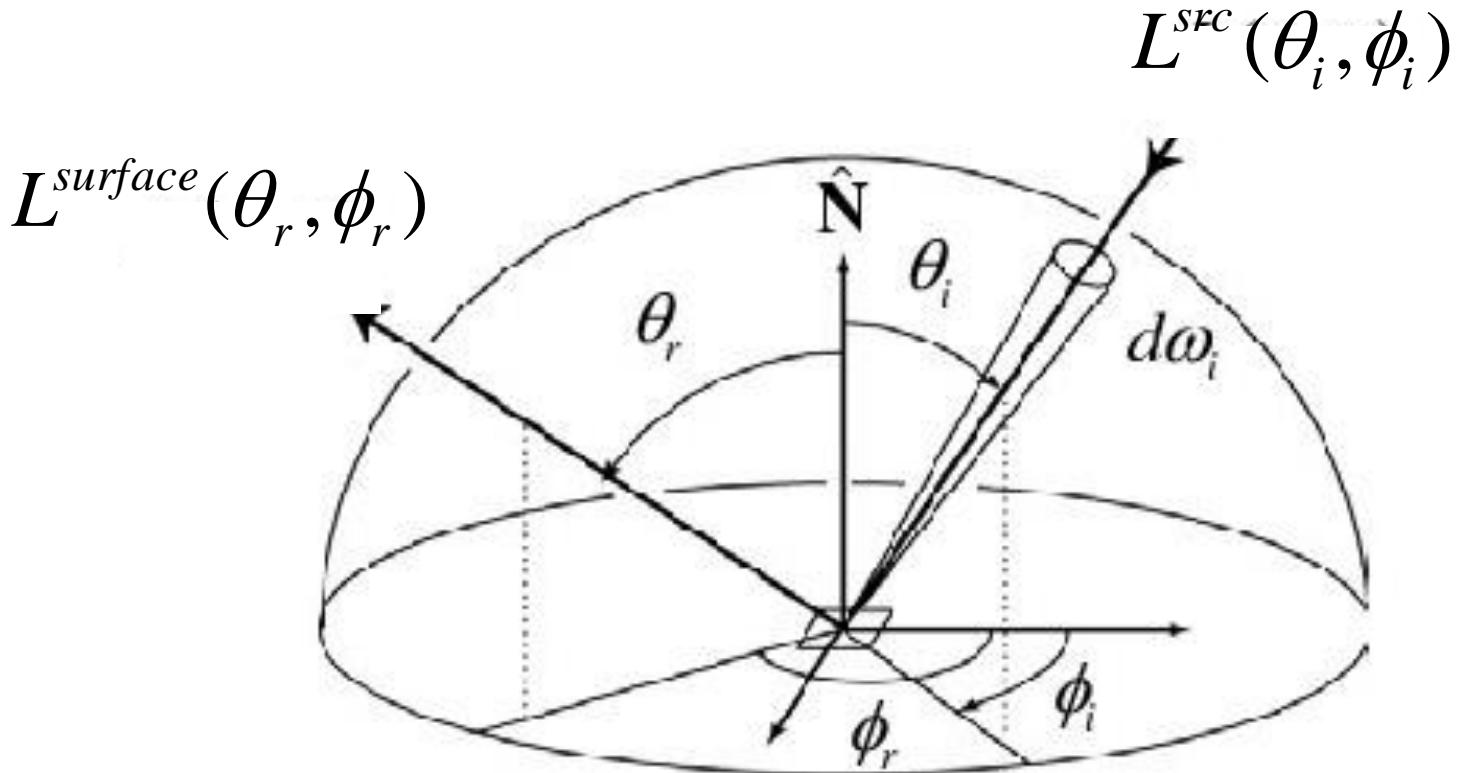
- Units:  $\text{sr}^{-1}$
- Real-valued function defined on the double-hemisphere
- Has many useful properties
- Allows computing output radiance (and thus pixel value) for *any* configuration of lights and viewpoint

$$L^{\text{out}}(\hat{\omega}) = \int_{\Omega_{\text{in}}} f(\hat{\omega}_{\text{in}}, \hat{\omega}_{\text{out}}) L^{\text{in}}(\hat{\omega}_{\text{in}}) \cos \theta_{\text{in}} d\hat{\omega}_{\text{in}}$$

reflectance equation

Why is there a cosine in the reflectance equation?

# Derivation of the Reflectance Equation



From the definition of BRDF:

$$L^{surface}(\theta_r, \phi_r) = E^{surface}(\theta_i, \phi_i) f(\theta_i, \phi_i; \theta_r, \phi_r)$$

# Derivation of the Scene Radiance Equation

From the definition of BRDF:

$$L^{surface}(\theta_r, \phi_r) = \underline{E^{surface}(\theta_i, \phi_i)f(\theta_i, \phi_i; \theta_r, \phi_r)}$$

Write Surface Irradiance in terms of Source Radiance:

$$L^{surface}(\theta_r, \phi_r) = \underline{L^{src}(\theta_i, \phi_i)f(\theta_i, \phi_i; \theta_r, \phi_r)} \underline{\cos\theta_i d\omega_i}$$

Integrate over entire hemisphere of possible source directions:

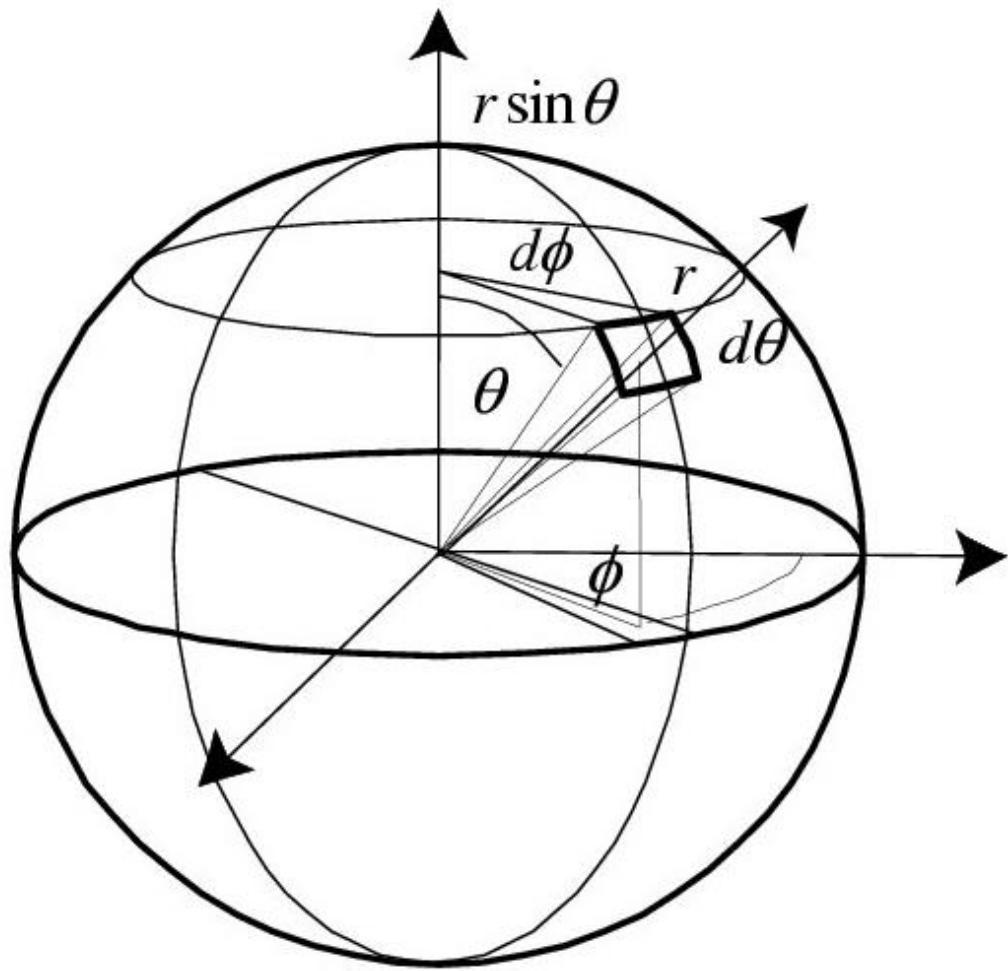
$$L^{surface}(\theta_r, \phi_r) = \int_{2\pi} L^{src}(\theta_i, \phi_i)f(\theta_i, \phi_i; \theta_r, \phi_r)\cos\theta_i \underline{d\omega_i}$$

Convert from solid angle to theta-phi representation:

$$L^{surface}(\theta_r, \phi_r) = \int_{-\pi}^{\pi} \int_0^{\pi/2} L^{src}(\theta_i, \phi_i)f(\theta_i, \phi_i; \theta_r, \phi_r)\cos\theta_i \sin\theta_i \underline{d\theta_i d\phi_i}$$

# Differential Solid Angles

---

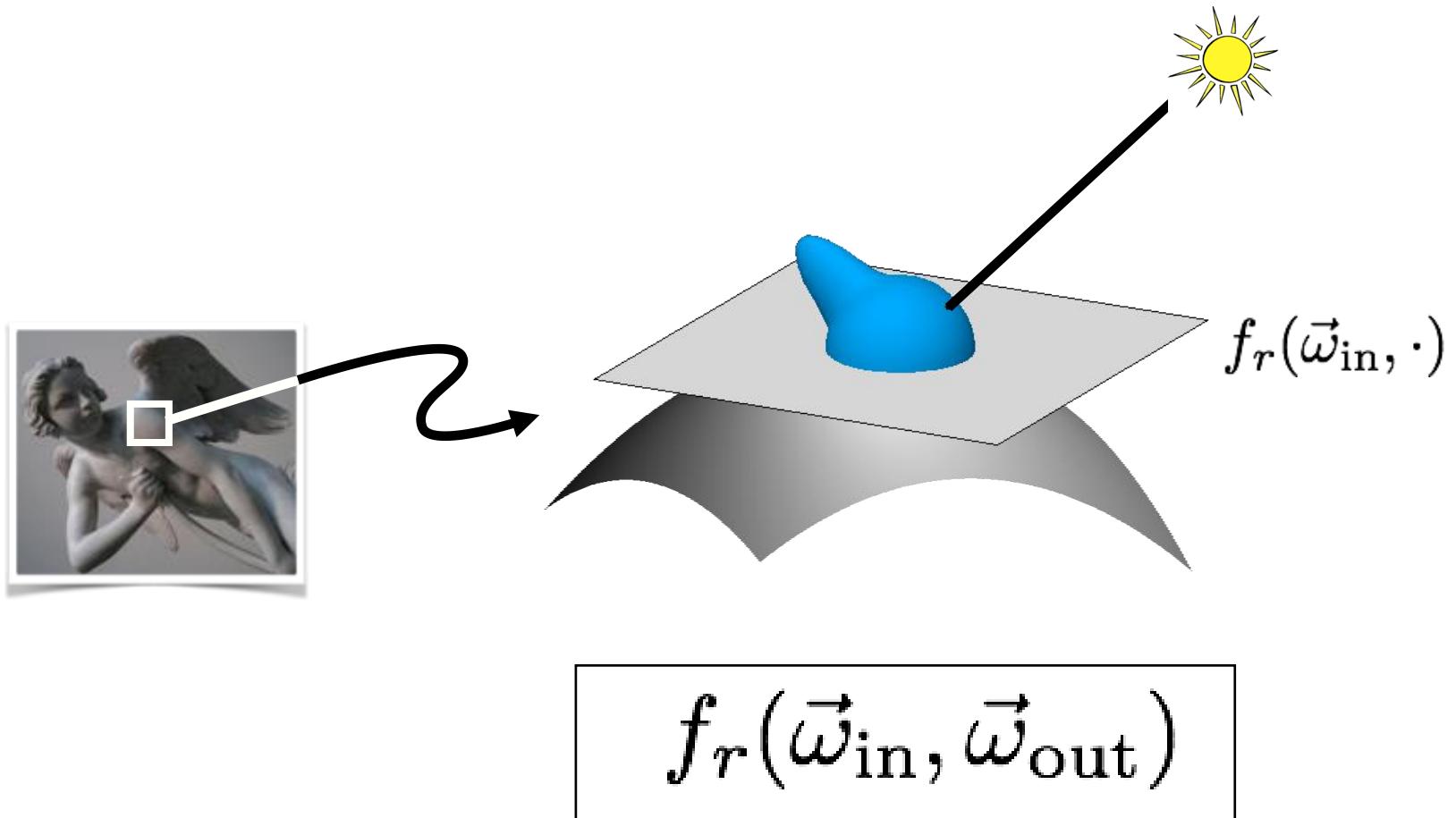


$$\begin{aligned} dA &= (r d\theta)(r \sin \theta d\phi) \\ &= r^2 \sin \theta d\theta d\phi \end{aligned}$$

$$d\omega = \frac{dA}{r^2} = \sin \theta d\theta d\phi$$

$$S = \int_0^{\pi} \int_0^{2\pi} \sin \theta d\theta d\phi = 4\pi$$

# BRDF

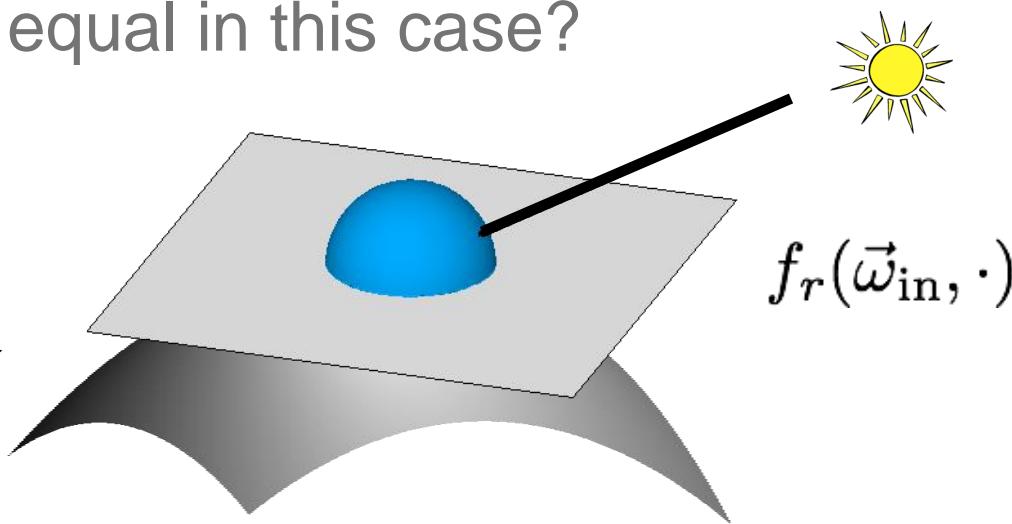
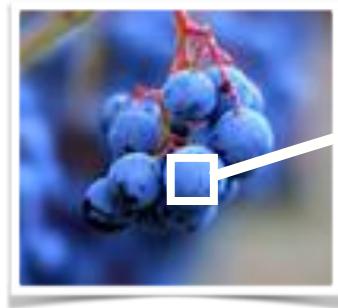


Bi-directional Reflectance Distribution Function (BRDF)

# BRDF

Lambertian (diffuse) BRDF: energy equally distributed in all directions

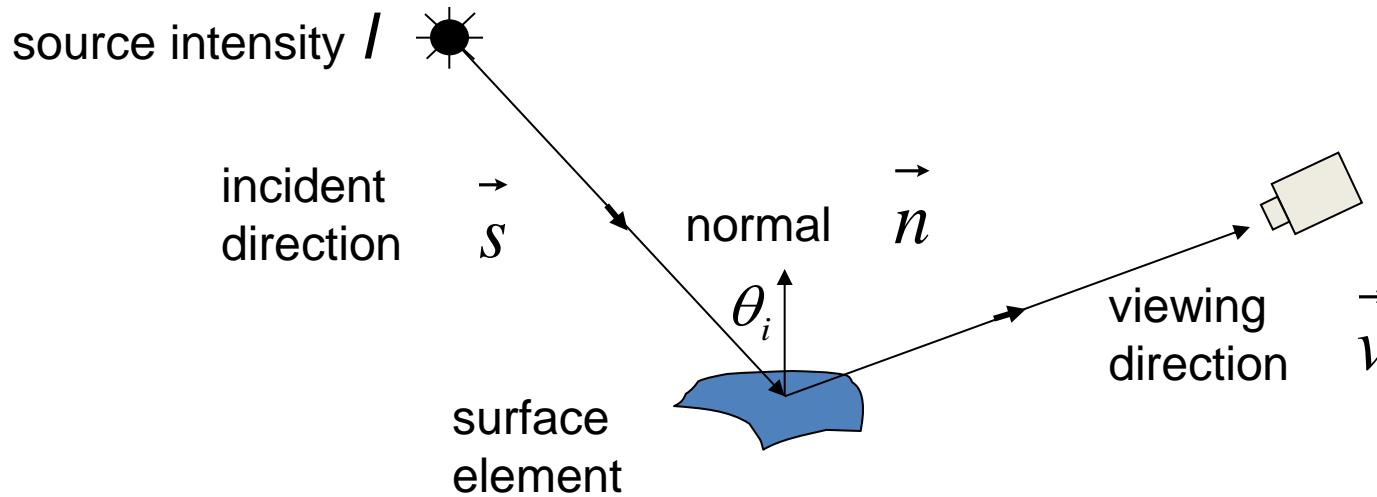
What does the BRDF equal in this case?



$$f_r(\vec{\omega}_{\text{in}}, \vec{\omega}_{\text{out}})$$

Bi-directional Reflectance Distribution Function (BRDF)

# Diffuse Reflection and Lambertian BRDF

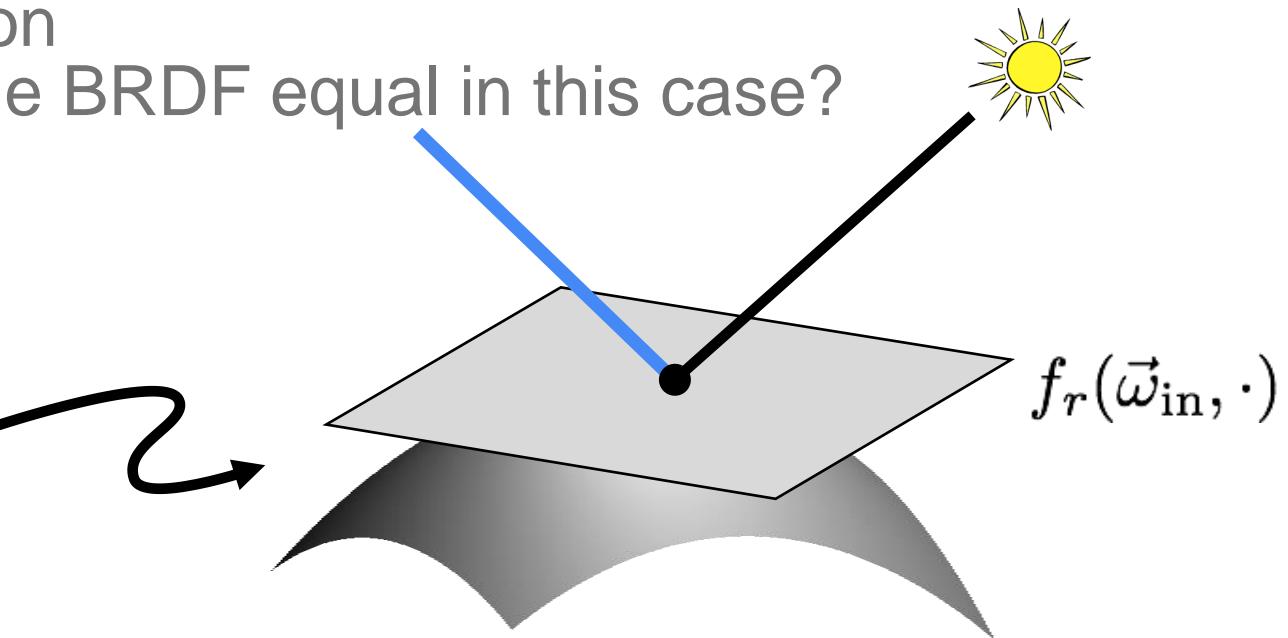


- Surface appears equally bright from ALL directions! (independent of  $\vec{v}$ )
- Lambertian BRDF is simply a constant :  $f(\theta_i, \phi_i; \theta_r, \phi_r) = \frac{\rho_d}{\pi}$  albedo
- Most commonly used BRDF in Vision and Graphics!

# BRDF

Specular BRDF: all energy concentrated in mirror direction

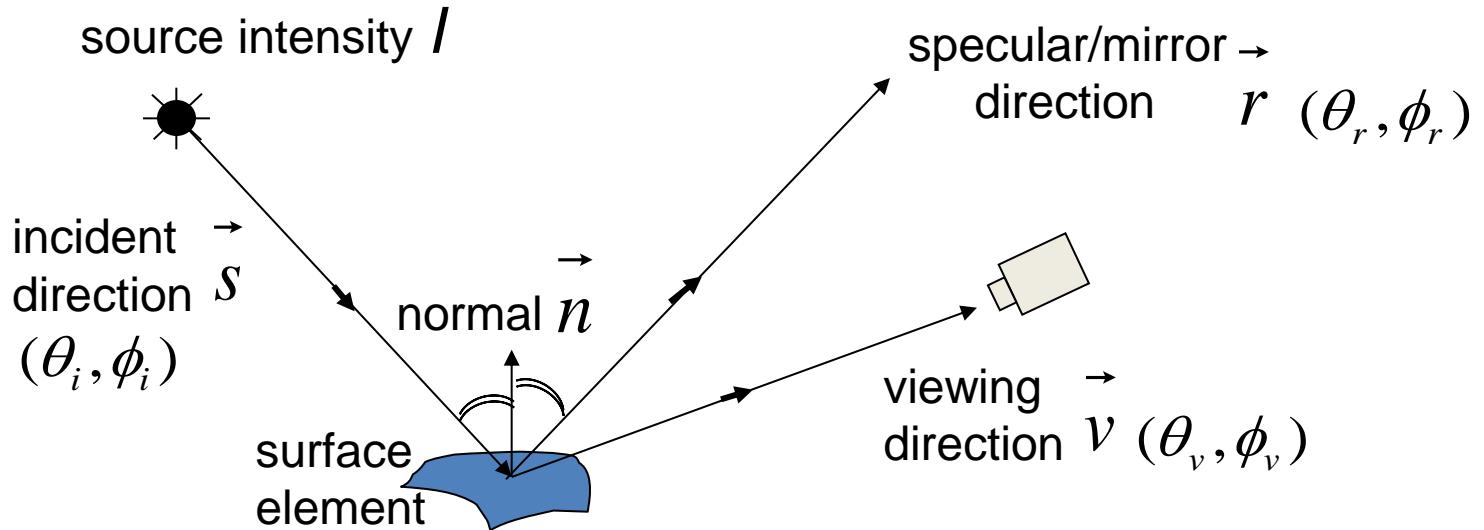
What does the BRDF equal in this case?



$$f_r(\vec{\omega}_{in}, \vec{\omega}_{out})$$

Bi-directional Reflectance Distribution Function (BRDF)

# Specular Reflection and Mirror BRDF



- Valid for very smooth surfaces.
- All incident light energy reflected in a SINGLE direction (only when  $\vec{v} = \vec{r}$ ).
- Mirror BRDF is simply a double-delta function :

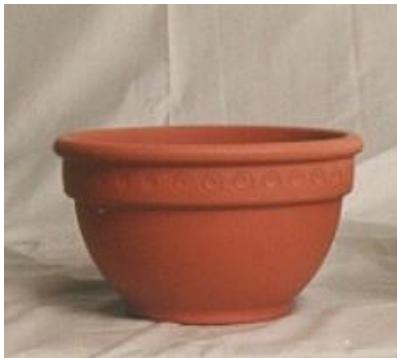
$$f(\theta_i, \phi_i; \theta_v, \phi_v) = \rho_s \delta(\theta_i - \theta_v) \delta(\phi_i + \pi - \phi_v)$$

specular albedo

# Example Surfaces

Body Reflection:

- Diffuse Reflection
- Matte Appearance
- Non-Homogeneous Medium
- Clay, paper, etc



Surface Reflection:

- Specular Reflection
- Glossy Appearance
- Highlights
- Dominant for Metals

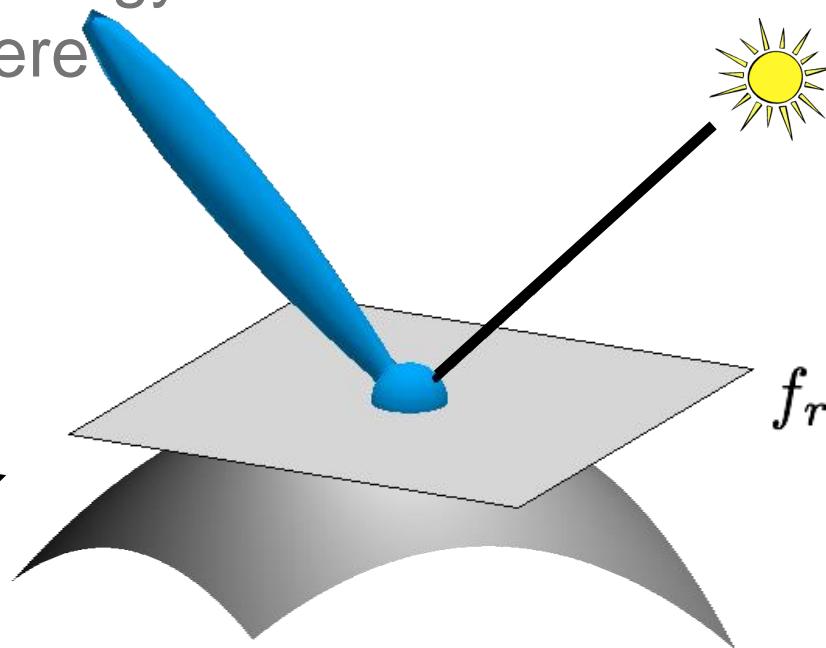
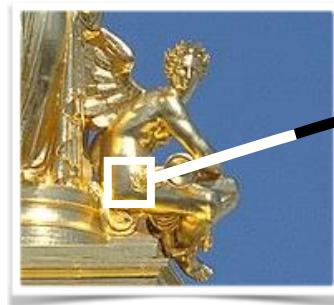


Many materials exhibit both Reflections:



# BRDF

Glossy BRDF: more energy concentrated in mirror direction than elsewhere



$$f_r(\vec{\omega}_{in}, \vec{\omega}_{out})$$

Bi-directional Reflectance Distribution Function (BRDF)

**Thank you: Question?**

# **Meshes and Geometry**

# **Processing**

---

**Computer Graphics**  
**CMU 15-462/15-662**

# Last time: overview of geometry

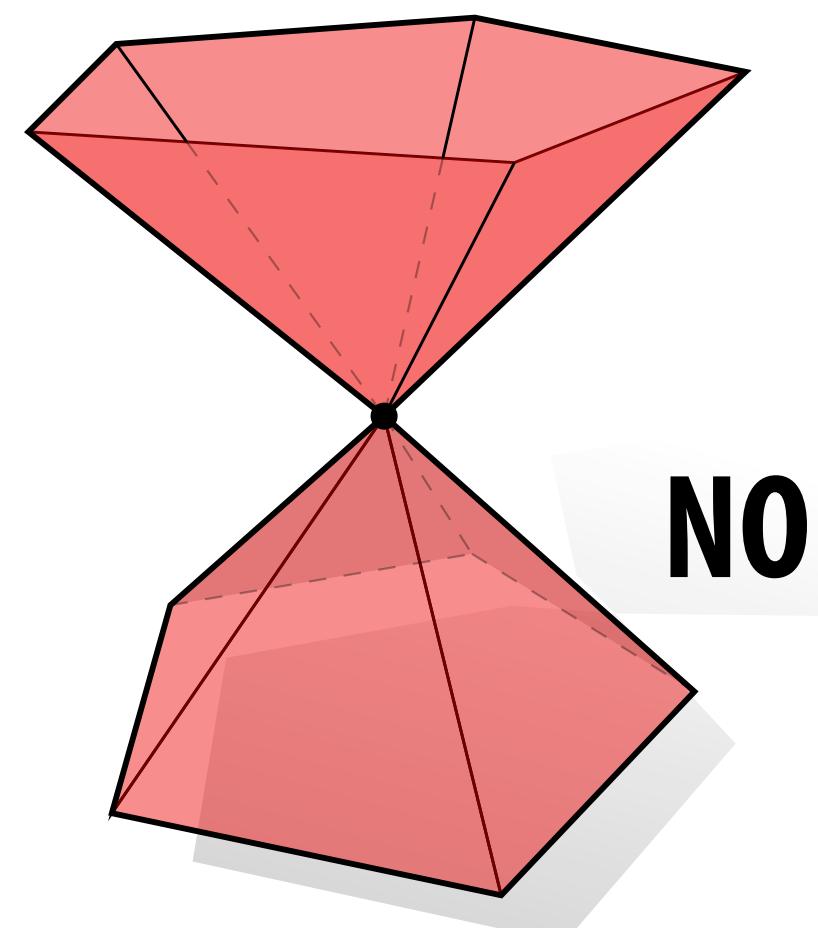
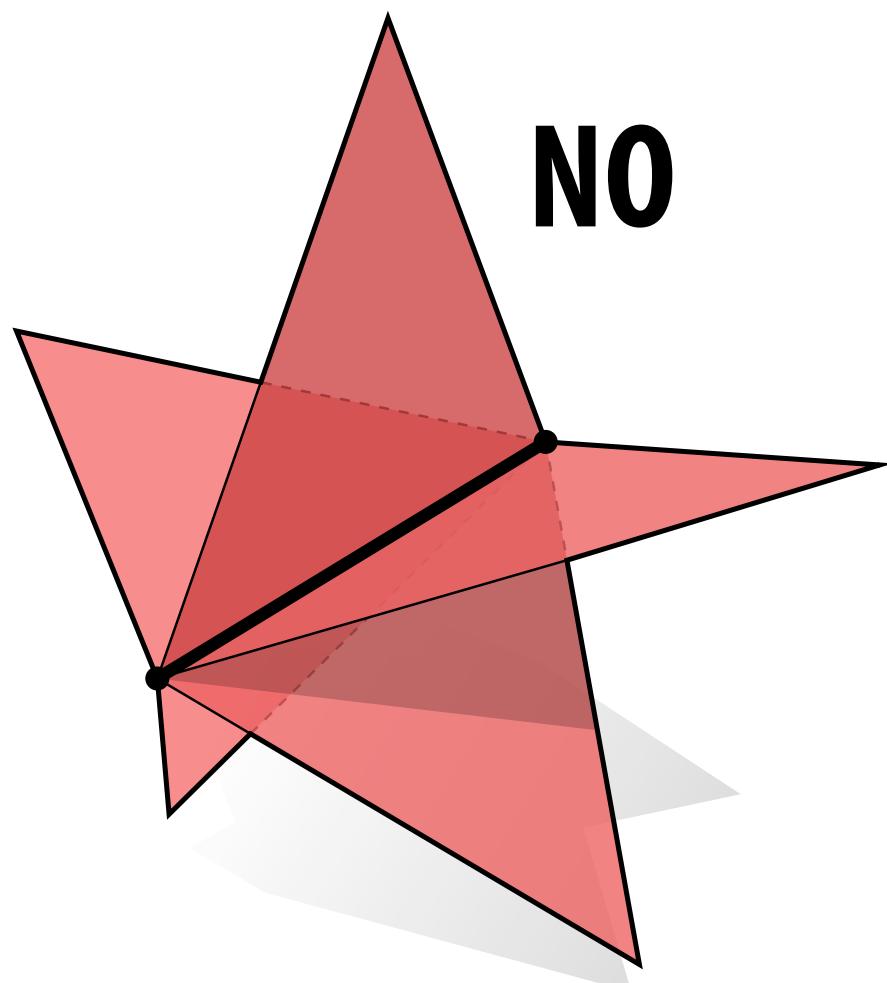
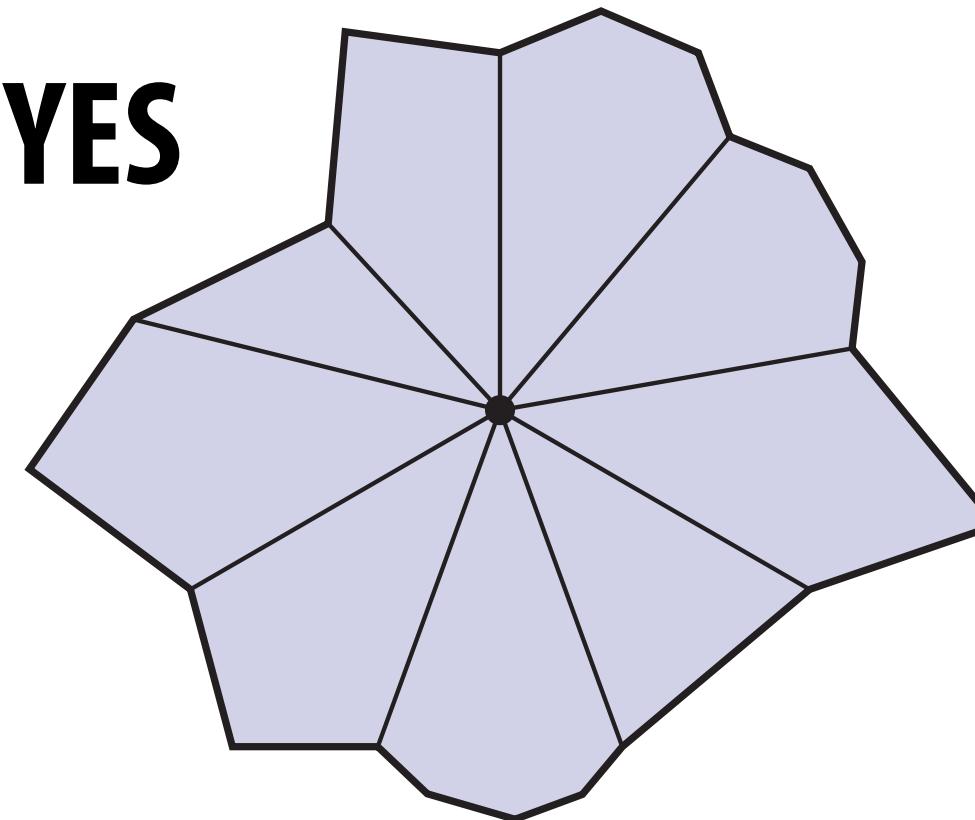
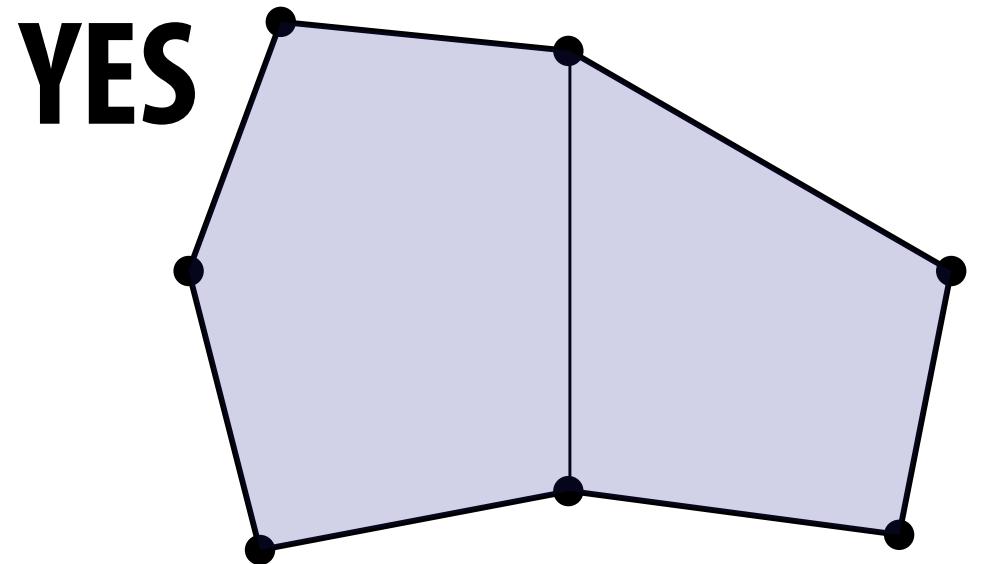
- Many types of geometry in nature
- Demand sophisticated representations
- Two major categories:
  - IMPLICIT - “tests” if a point is in shape
  - EXPLICIT - directly “lists” points
- Lots of representations for both
- Introduction to manifold geometry
- Today:
  - nuts & bolts of polygon meshes
  - geometry processing / resampling

Geometry



# From Monday: A manifold polygon mesh has fans, not fins

- For polygonal surfaces just two easy conditions to check:
  1. Every edge is contained in only two polygons (no “fins”)
  2. The polygons containing each vertex make a single “fan”

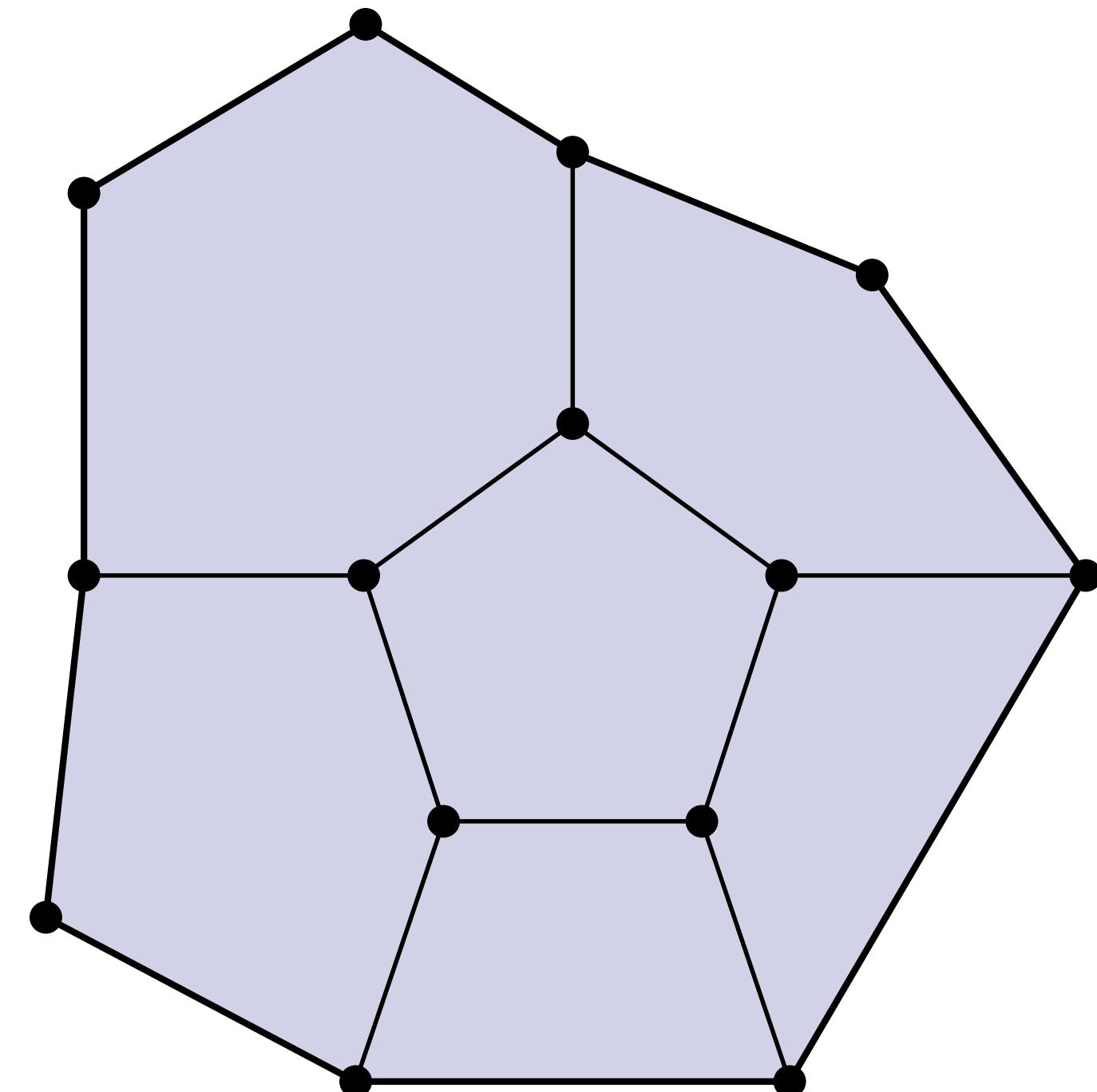
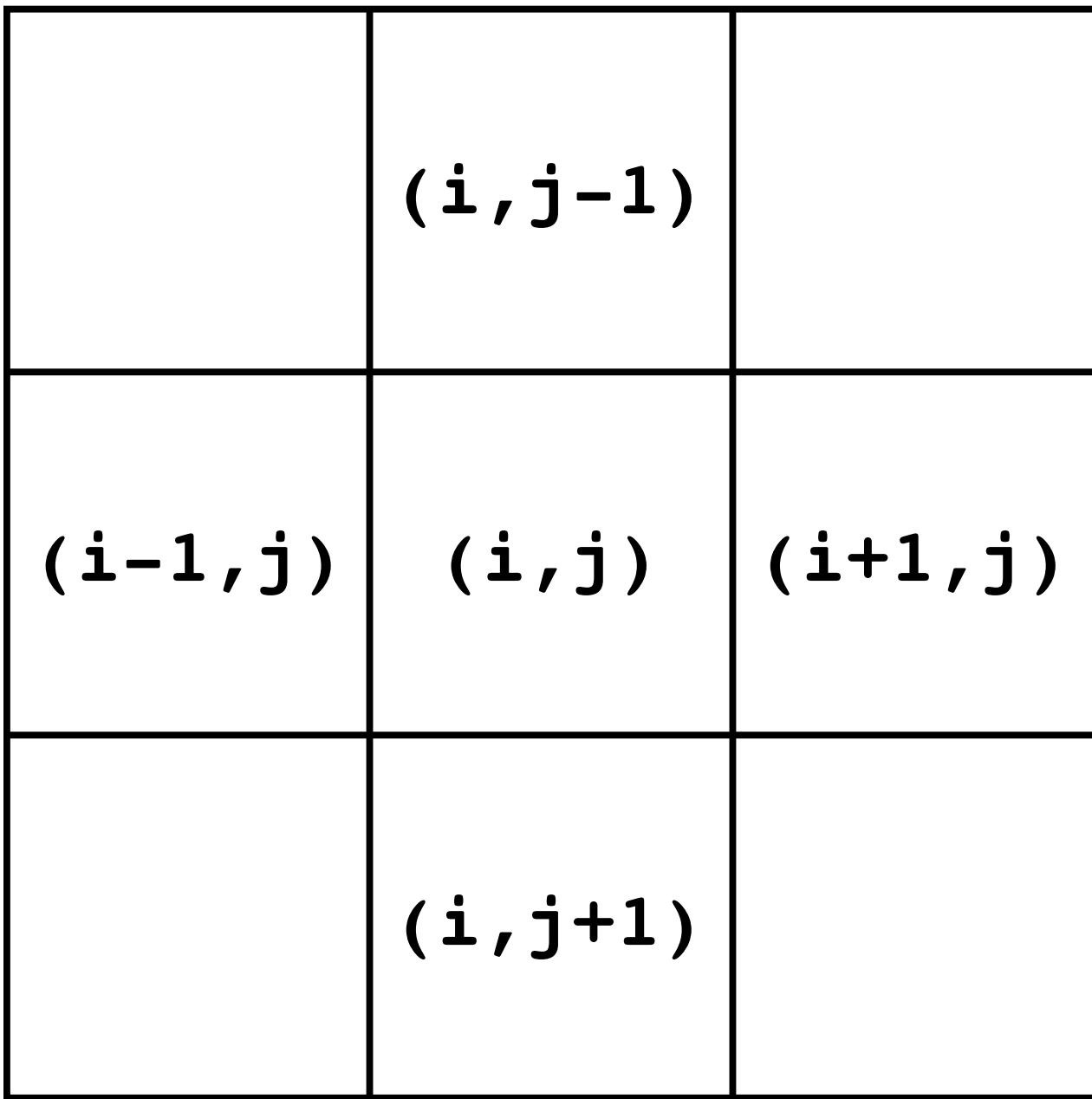


**Ok, but why is the manifold  
assumption useful?**

# Keep it Simple!

## ■ Same motivation as for images:

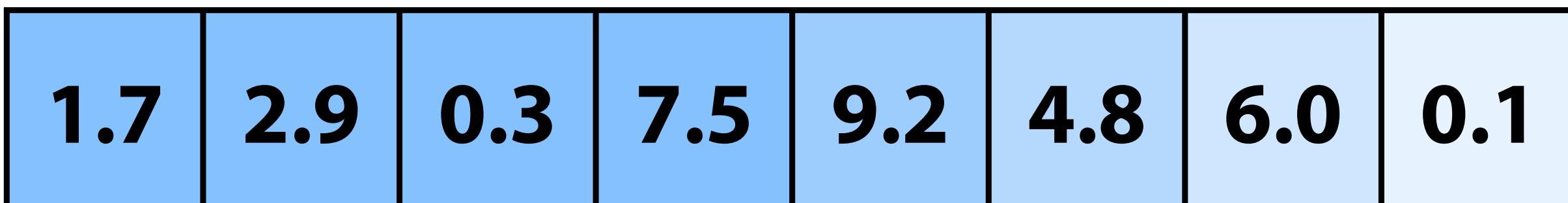
- make some assumptions about our geometry to keep data structures/algorithms simple and efficient
- in many common cases, doesn't fundamentally limit what we can do with geometry



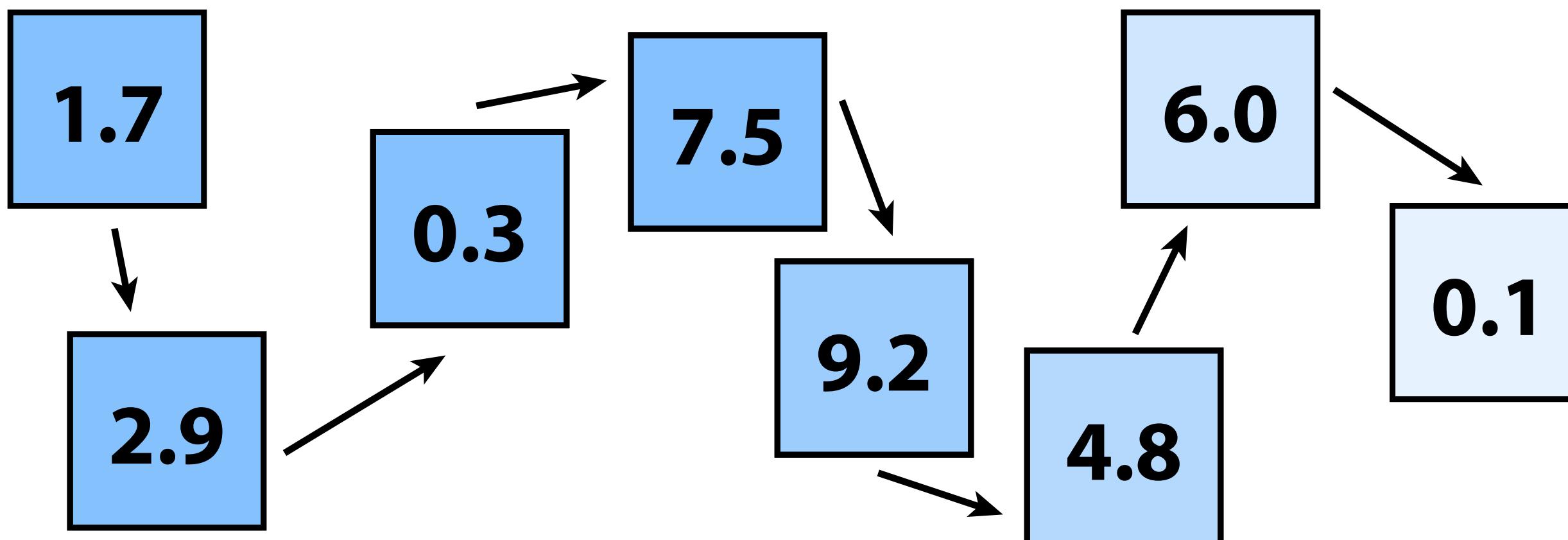
**How do we actually encode all this data?**

# Warm up: storing numbers

- Q: What data structures can we use to store a list of numbers?
- One idea: use an array (constant time lookup, coherent access)



- Alternative: use a linked list (linear lookup, incoherent access)



- Q: Why bother with the linked list?
- A: For one, we can easily insert numbers wherever we like...

# Polygon Soup

## ■ Most basic idea:

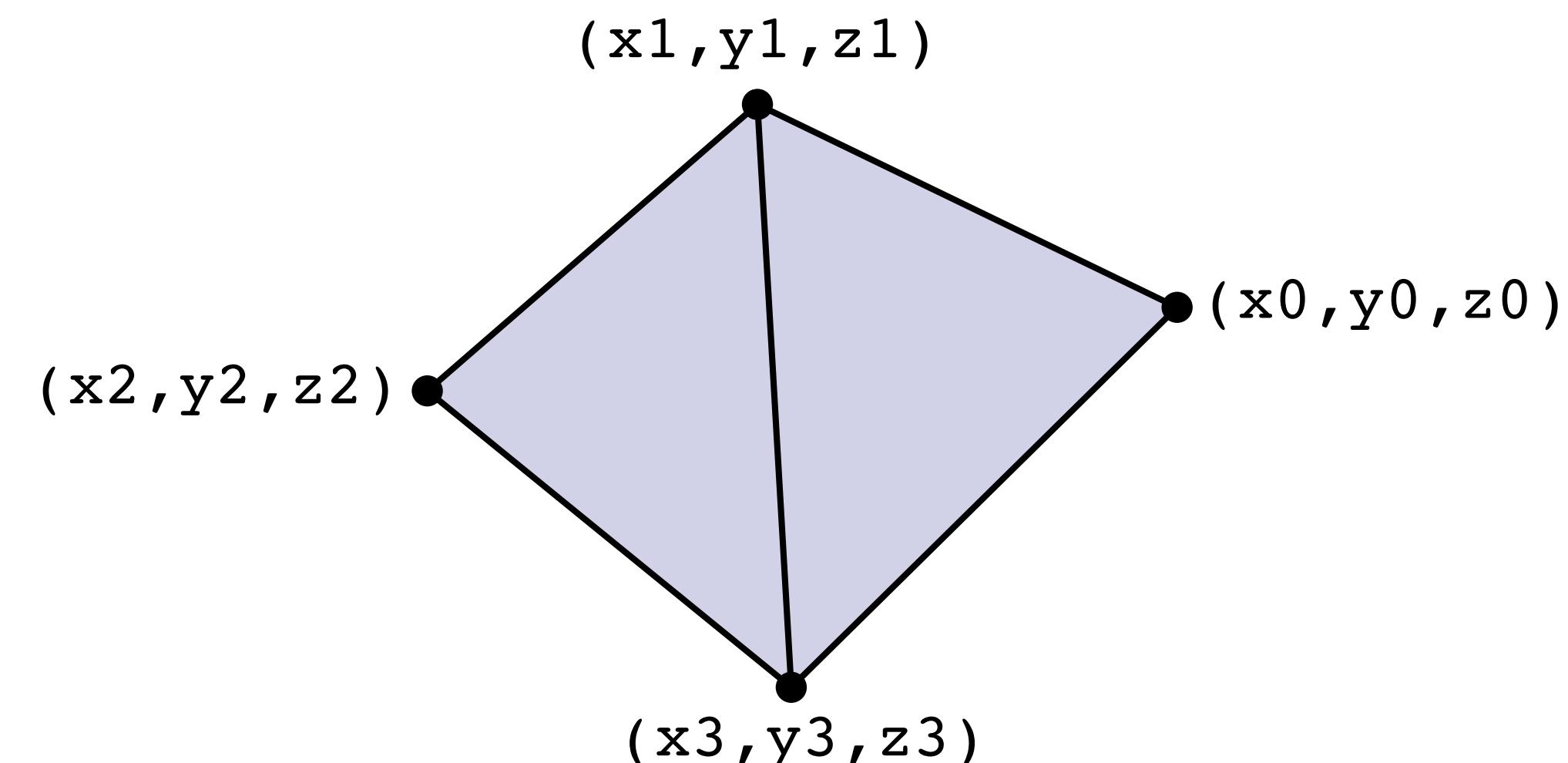
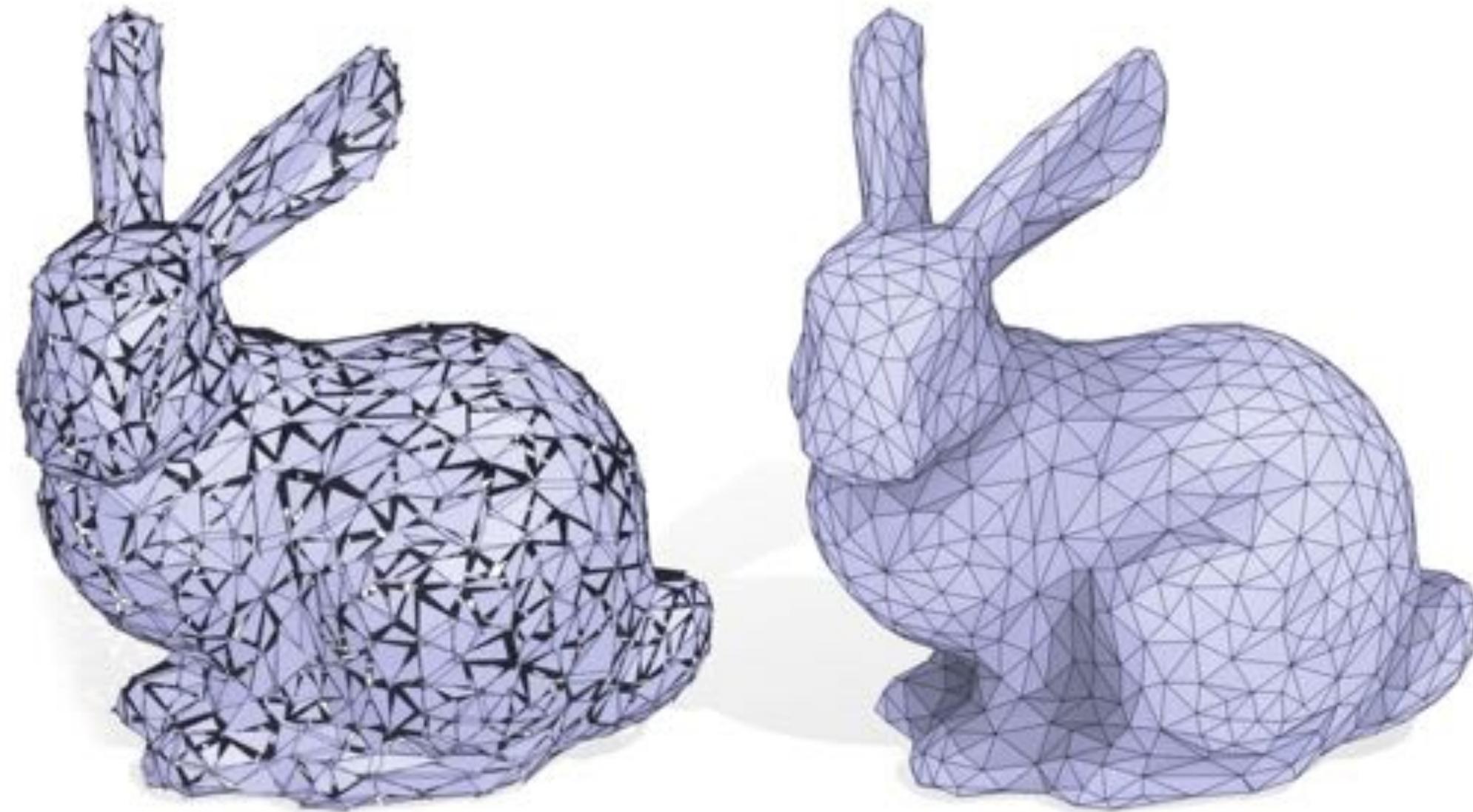
- For each triangle, just store three coordinates
- No other information about connectivity
- Not much different from point cloud! ("Triangle cloud?")

## ■ Pros:

- Really stupidly simple

## ■ Cons:

- Redundant storage
- Hard to do much beyond simply drawing the mesh on screen
- Need spatial data structures (later) to find neighbors

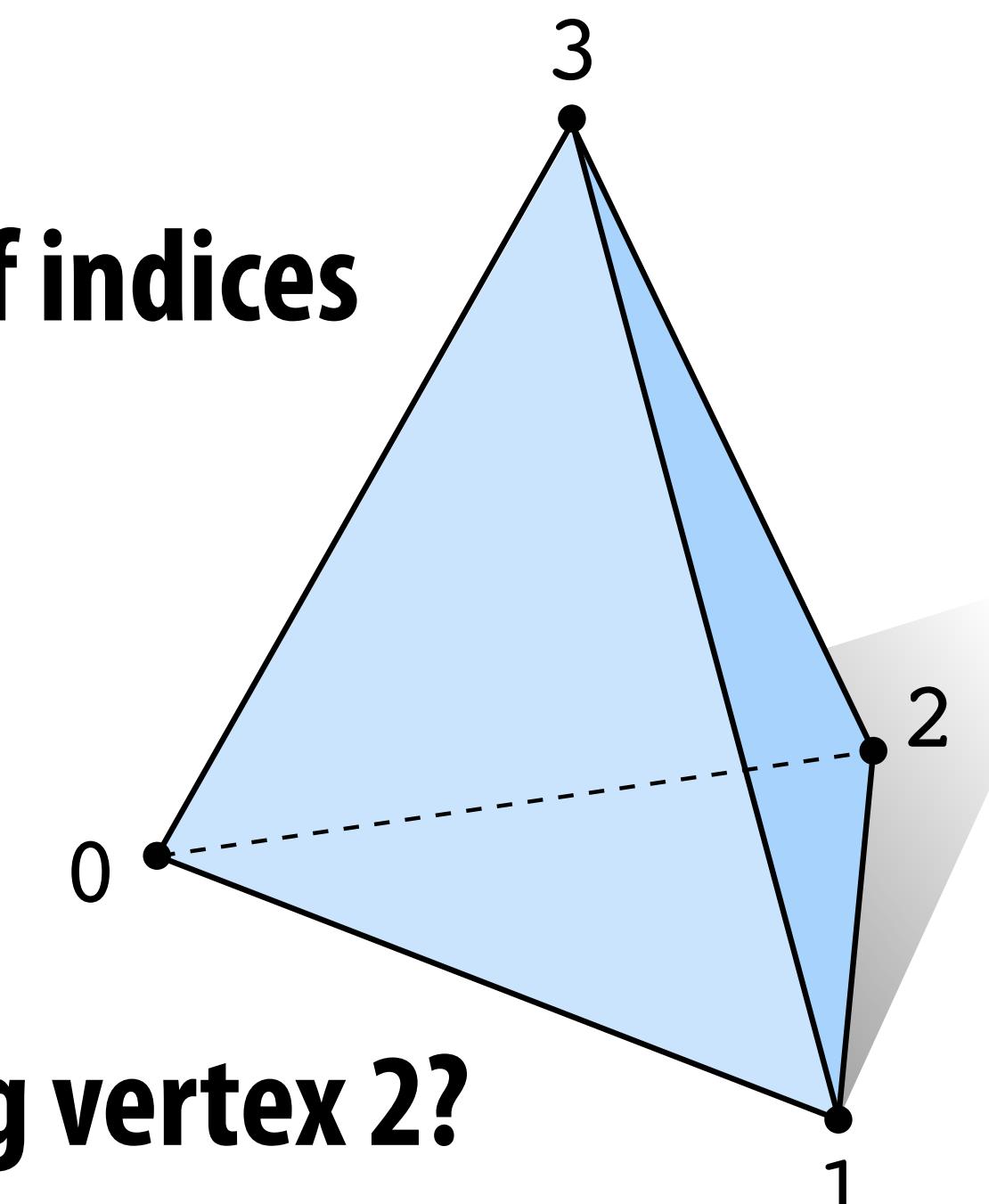


$x_0, y_0, z_0$	$x_1, y_1, z_1$	$x_3, y_3, z_3$
$x_1, y_1, z_1$	$x_2, y_2, z_2$	$x_3, y_3, z_3$

# Adjacency List (Array-like)

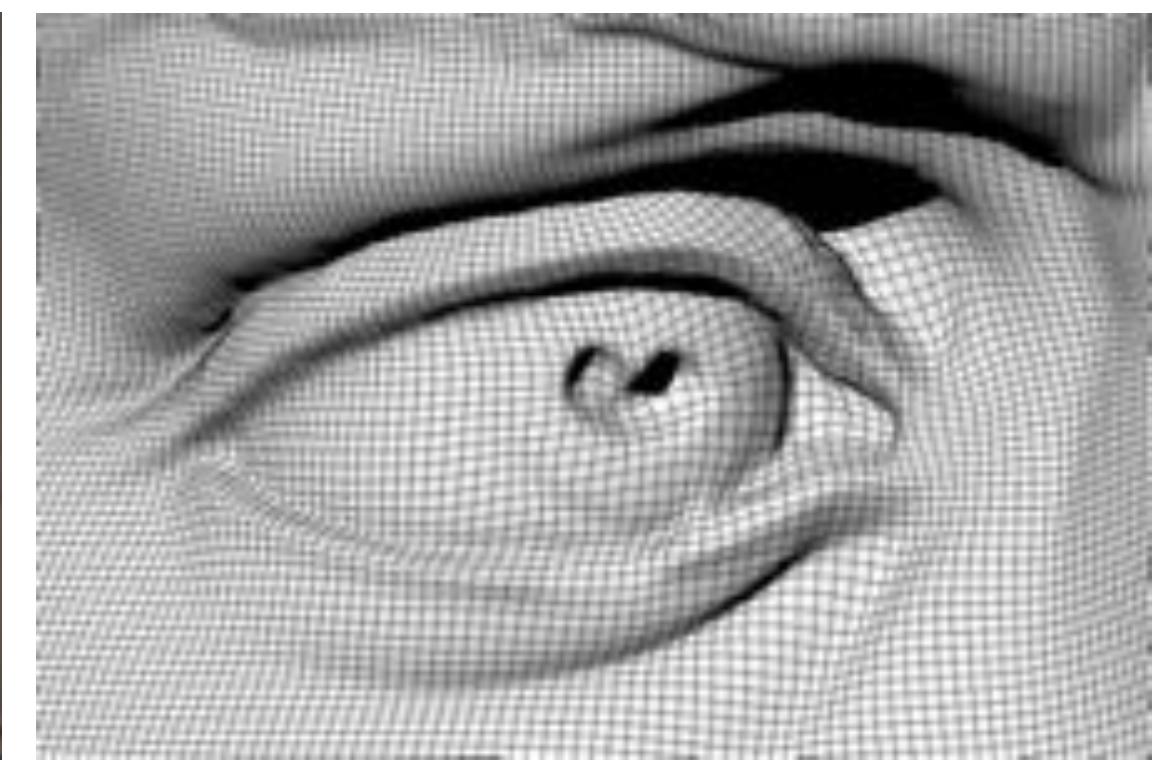
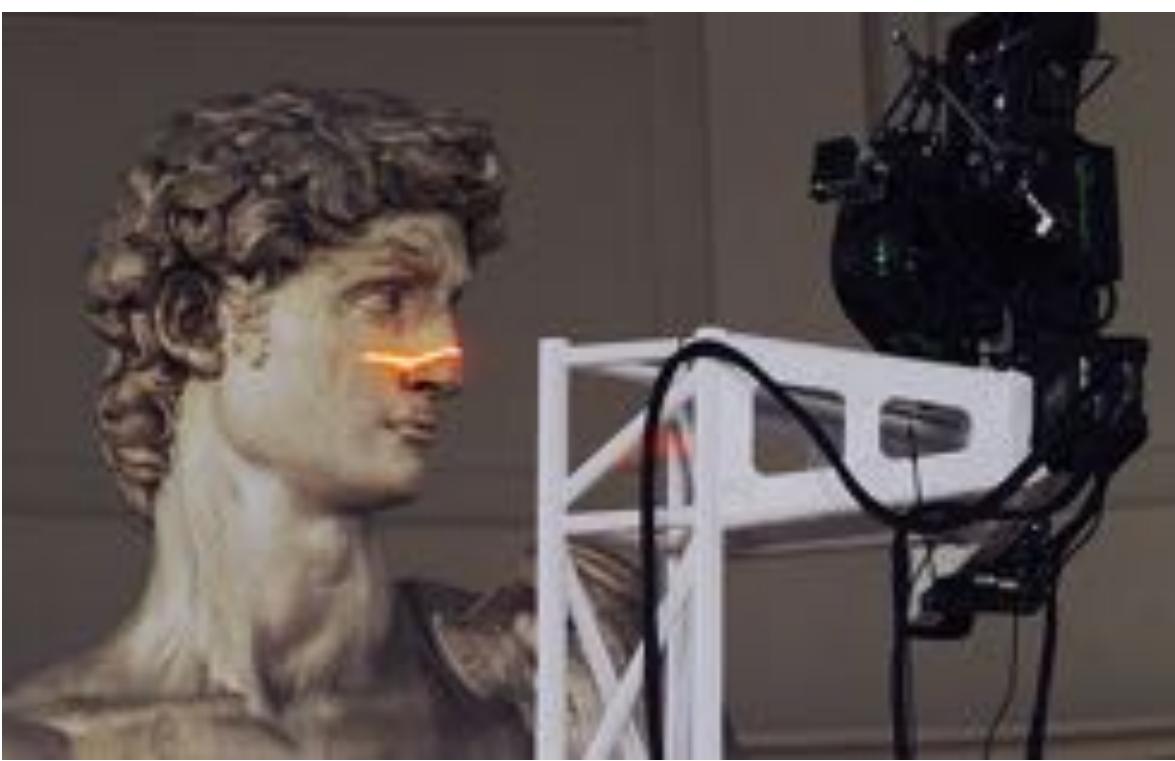
- Store triples of coordinates ( $x,y,z$ ), tuples of indices
- E.g., tetrahedron:

	VERTICES			POLYGONS		
	x	y	z	i	j	k
0:	-1	-1	-1	0	2	1
1:	1	-1	1	0	3	2
2:	1	1	-1	3	0	1
3:	-1	1	1	3	1	2



- Q: How do we find all the polygons touching vertex 2?
- Ok, now consider a more complicated mesh:

~1 billion polygons



Very expensive to find the neighboring polygons! (What's the cost?)

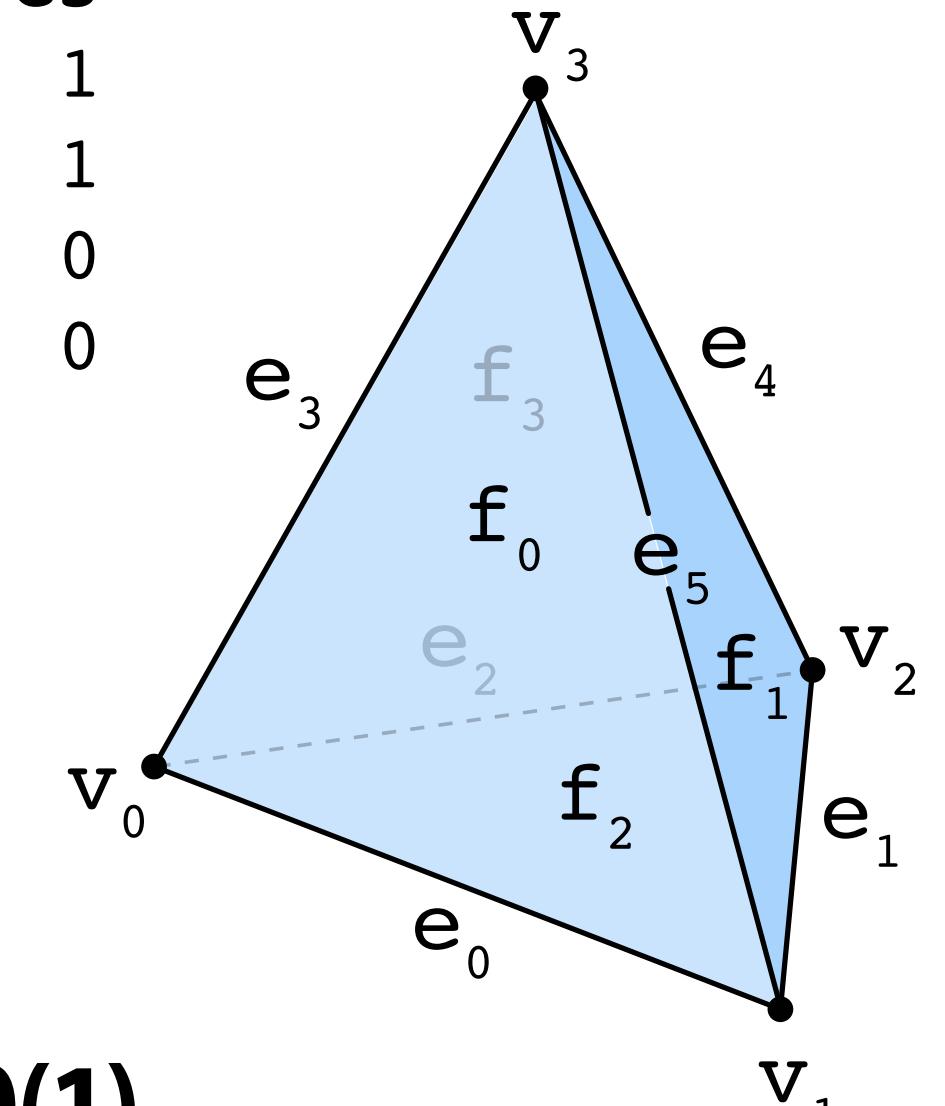
# Incidence Matrices

- If we want to know who our neighbors are, why not just store a list of neighbors?
- Can encode all neighbor information via incidence matrices
- E.g., tetrahedron:

VERTEX $\leftrightarrow$ EDGE

	v0	v1	v2	v3		e0	e1	e2	e3	e4	e5
e0	1	1	0	0	f0	1	0	0	1	0	1
e1	0	1	1	0	f1	0	1	0	0	1	1
e2	1	0	1	0	f2	1	1	1	0	0	0
e3	1	0	0	1	f3	0	0	1	1	1	0
e4	0	0	1	1							
e5	0	1	0	1							

EDGE $\leftrightarrow$ FACE

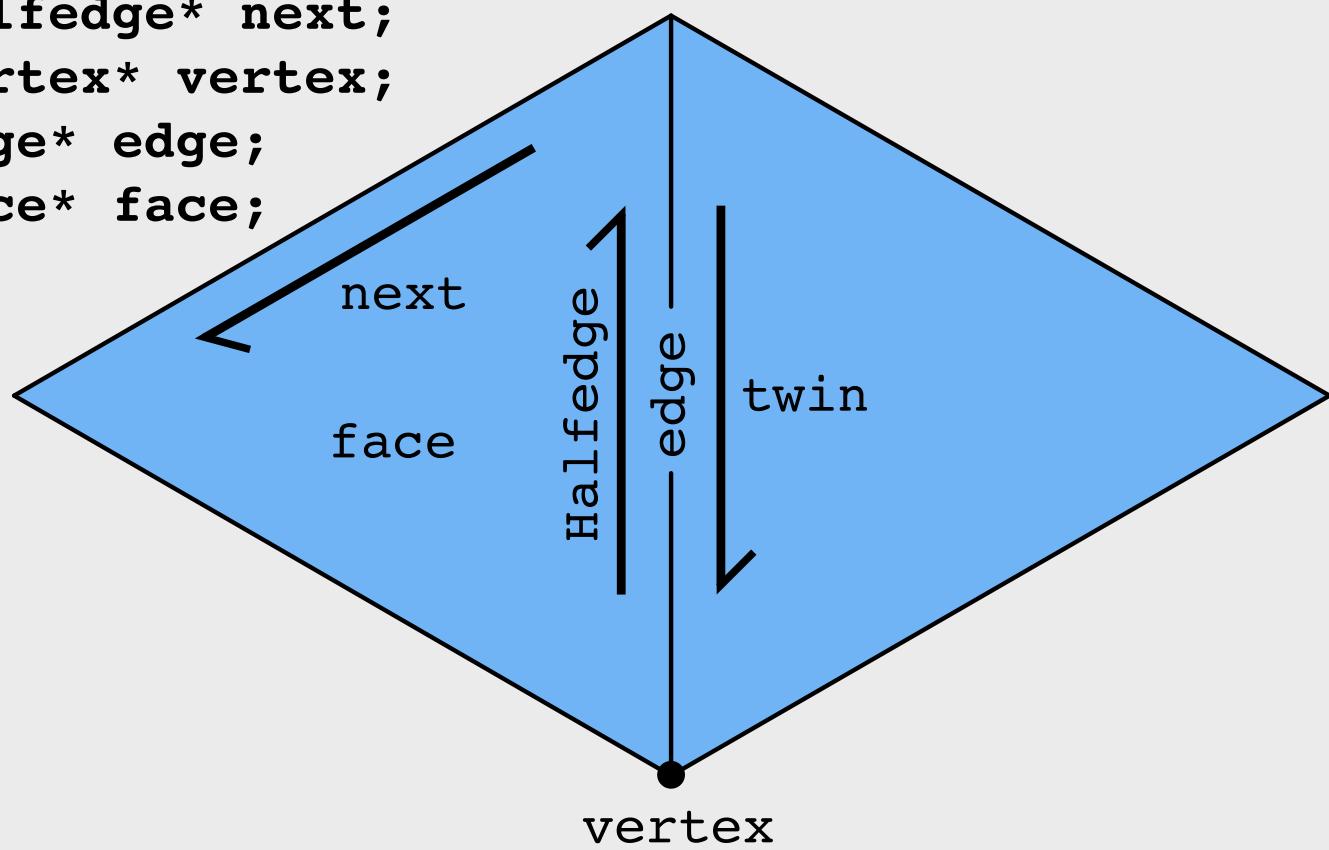


- 1 means “touches”; 0 means “does not touch”
- Instead of storing lots of 0's, use sparse matrices
- Still large storage cost, but finding neighbors is now  $O(1)$
- Hard to change connectivity, since we used fixed indices
- Bonus feature: mesh does not have to be manifold

# Halfedge Data Structure (Linked-list-like)

- Store some information about neighbors
- Don't need an exhaustive list; just a few key pointers
- Key idea: two halfedges act as “glue” between mesh elements:

```
struct Halfedge
{
    Halfedge* twin;
    Halfedge* next;
    Vertex* vertex;
    Edge* edge;
    Face* face;
};
```



```
struct Edge
{
    Halfedge* halfedge;
};
```

```
struct Face
{
    Halfedge* halfedge;
};
```

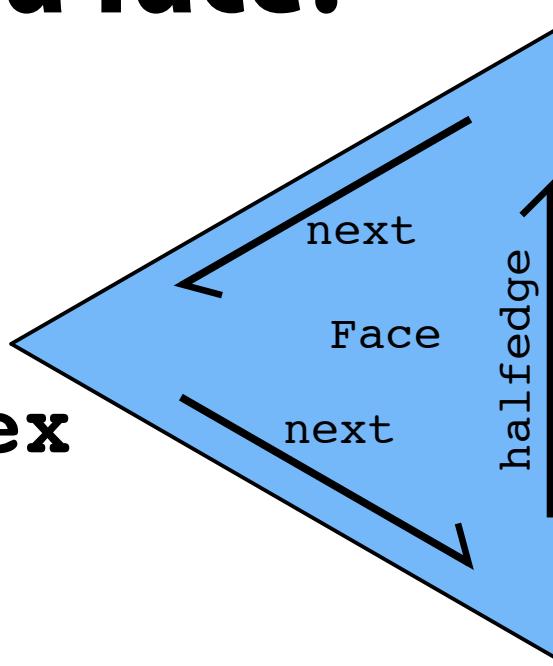
```
struct Vertex
{
    Halfedge* halfedge;
};
```

- Each vertex, edge face points to just one of its halfedges.

# Halfedge makes mesh traversal easy

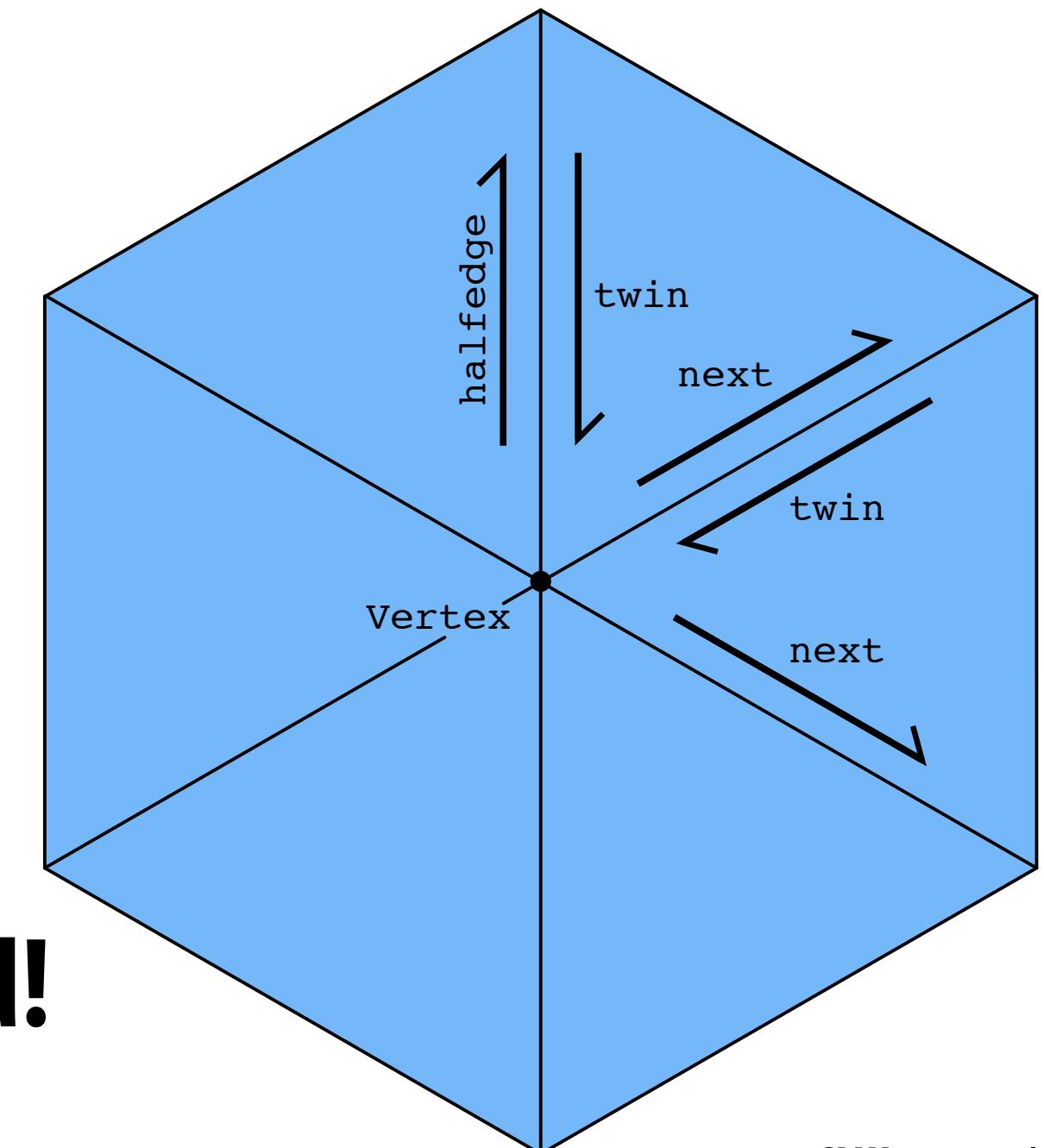
- Use “twin” and “next” pointers to move around mesh
- Use “vertex”, “edge”, and “face” pointers to grab element
- Example: visit all vertices of a face:

```
Halfedge* h = f->halfedge;  
do {  
    h = h->next;  
    // do something w/ h->vertex  
}  
while( h != f->halfedge );
```



- Example: visit all neighbors of a vertex:

```
Halfedge* h = v->halfedge;  
do {  
    h = h->twin->next;  
}  
while( h != v->halfedge );
```



- Note: only makes sense if mesh is manifold!

# Halfedge connectivity is always manifold

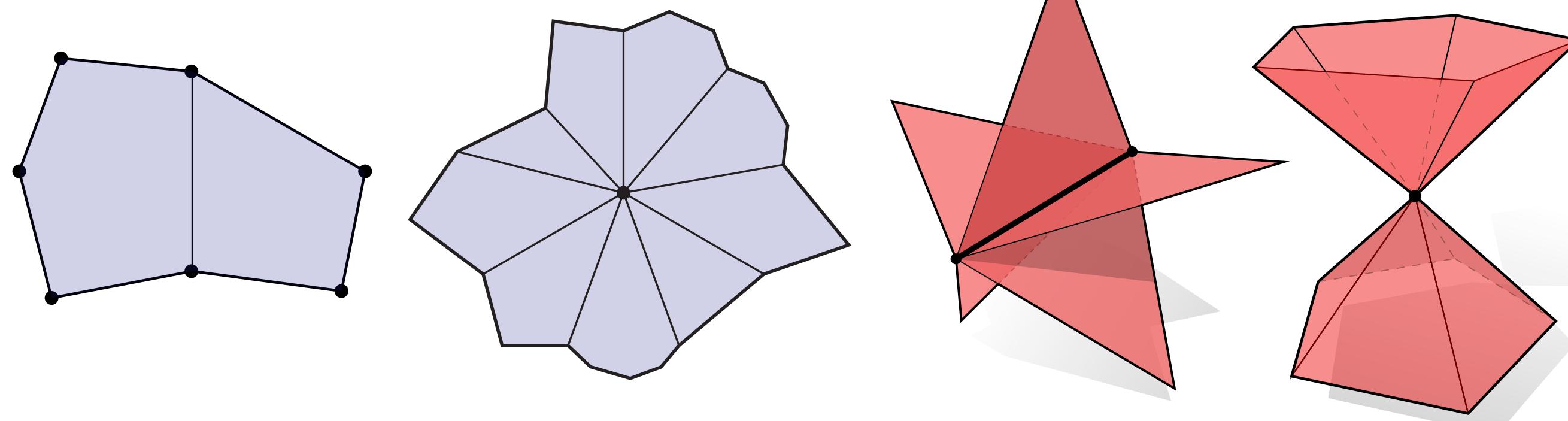
- Consider simplified halfedge data structure
- Require only “common-sense” conditions

```
struct Halfedge {  
    Halfedge *next, *twin;  
};
```

(pointer to yourself!)

twin->twin == this  
twin != this  
every he is someone's "next"

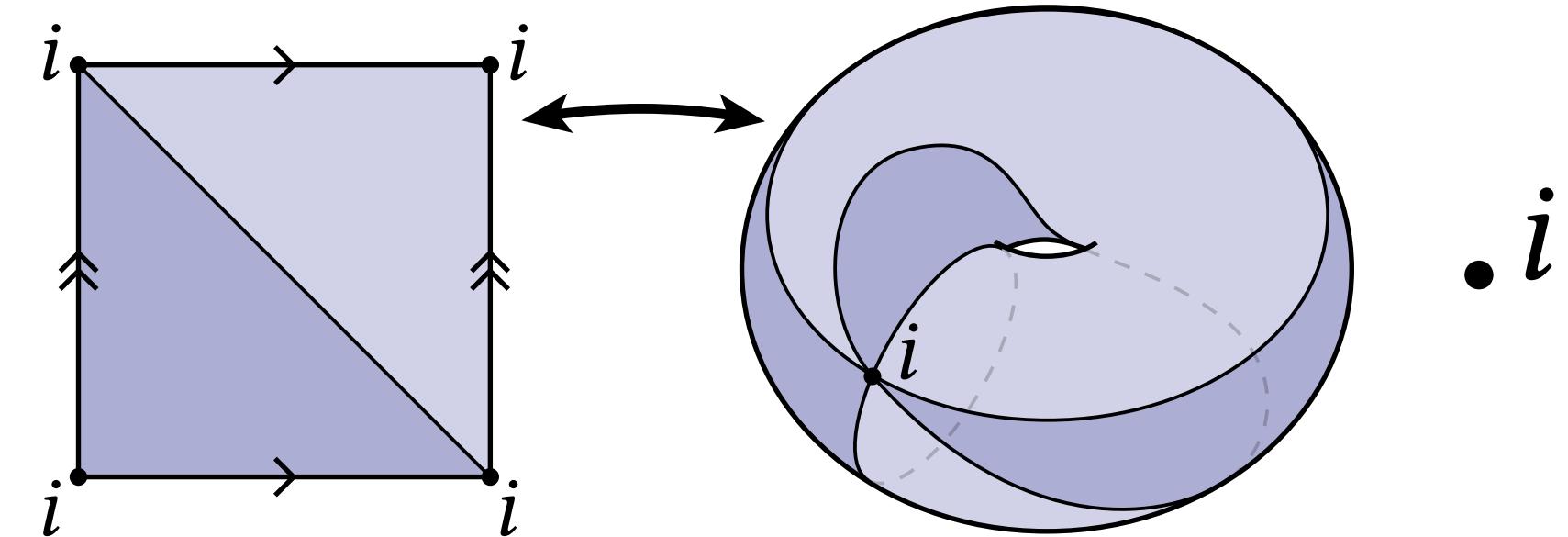
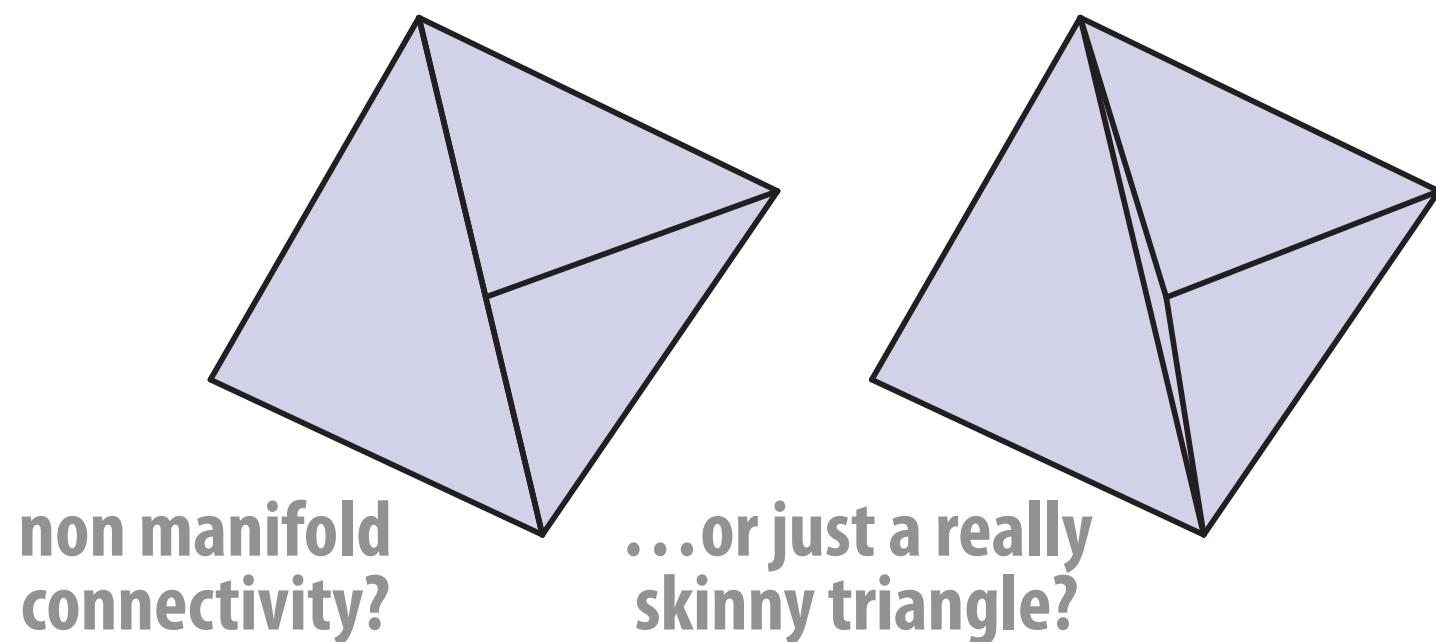
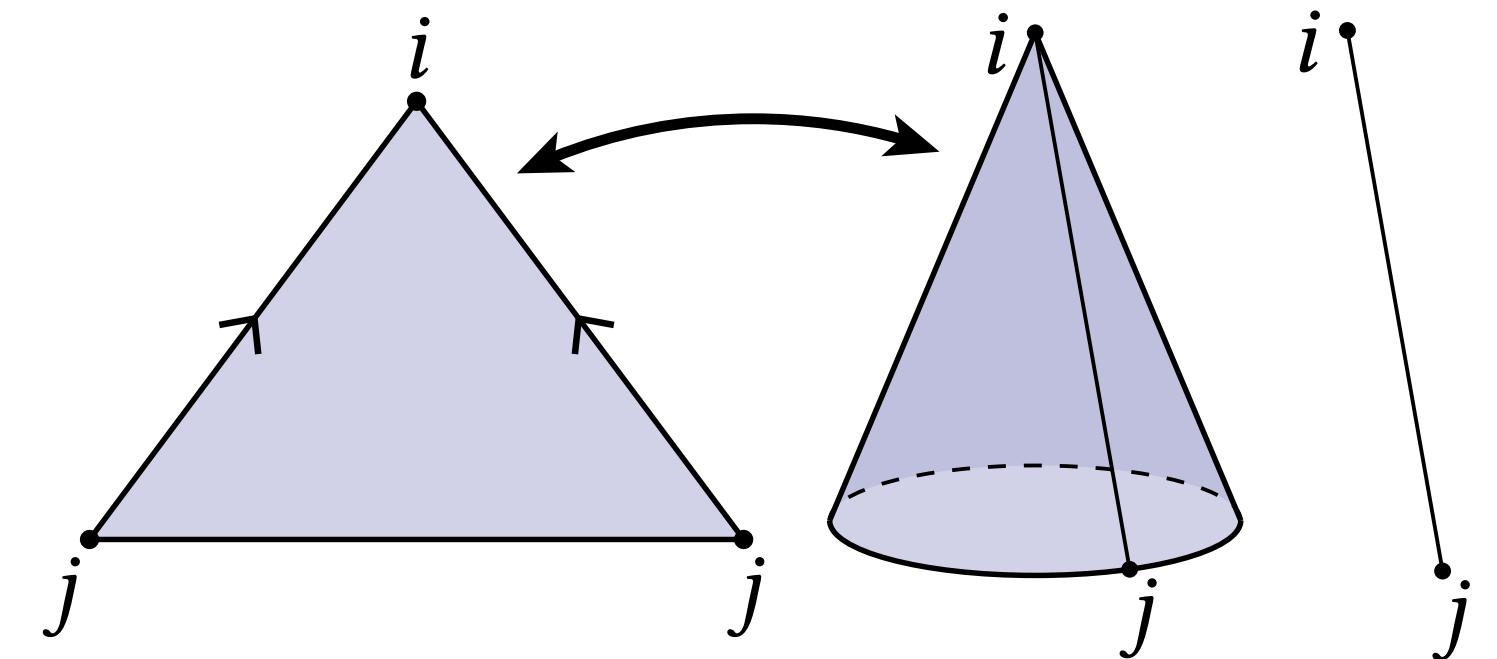
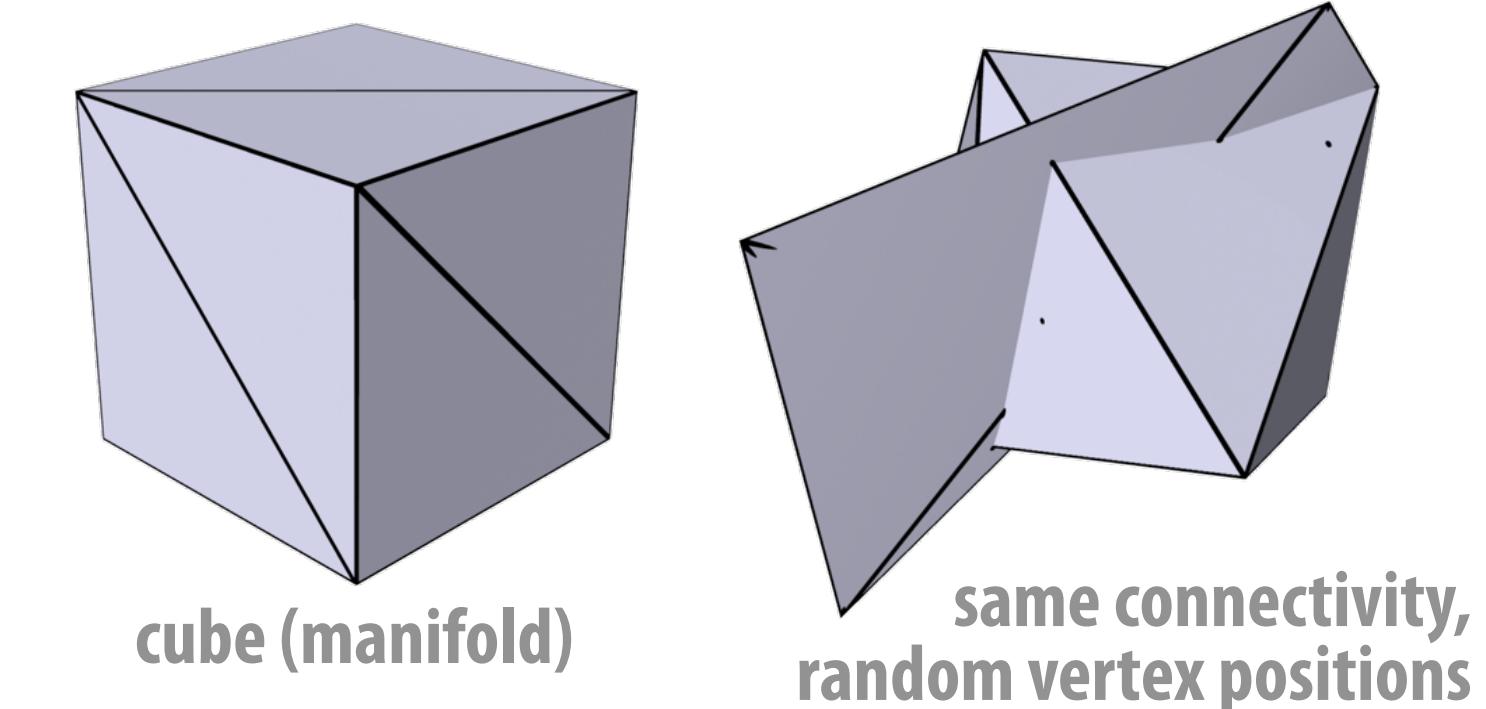
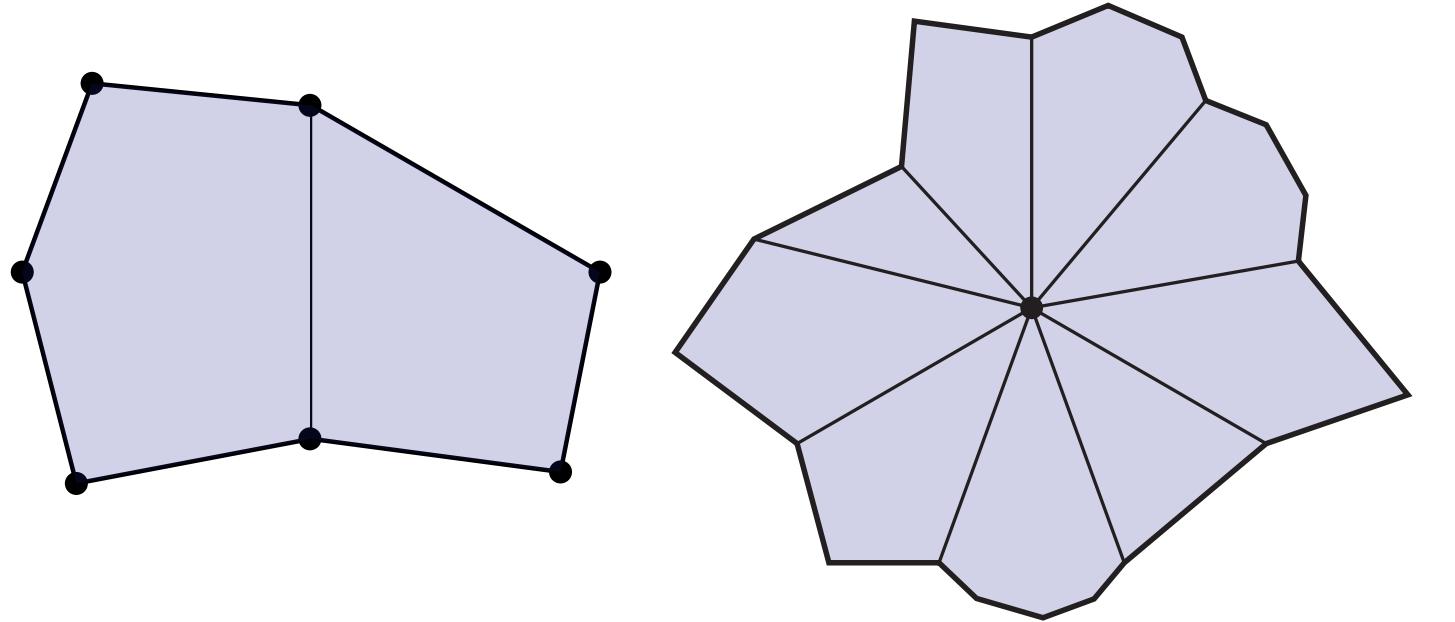
- Keep following `next`, and you'll get faces.
- Keep following `twin` and you'll get edges.
- Keep following `next->twin` and you'll get vertices.



**Q: Why, therefore, is it impossible to encode the red figures?**

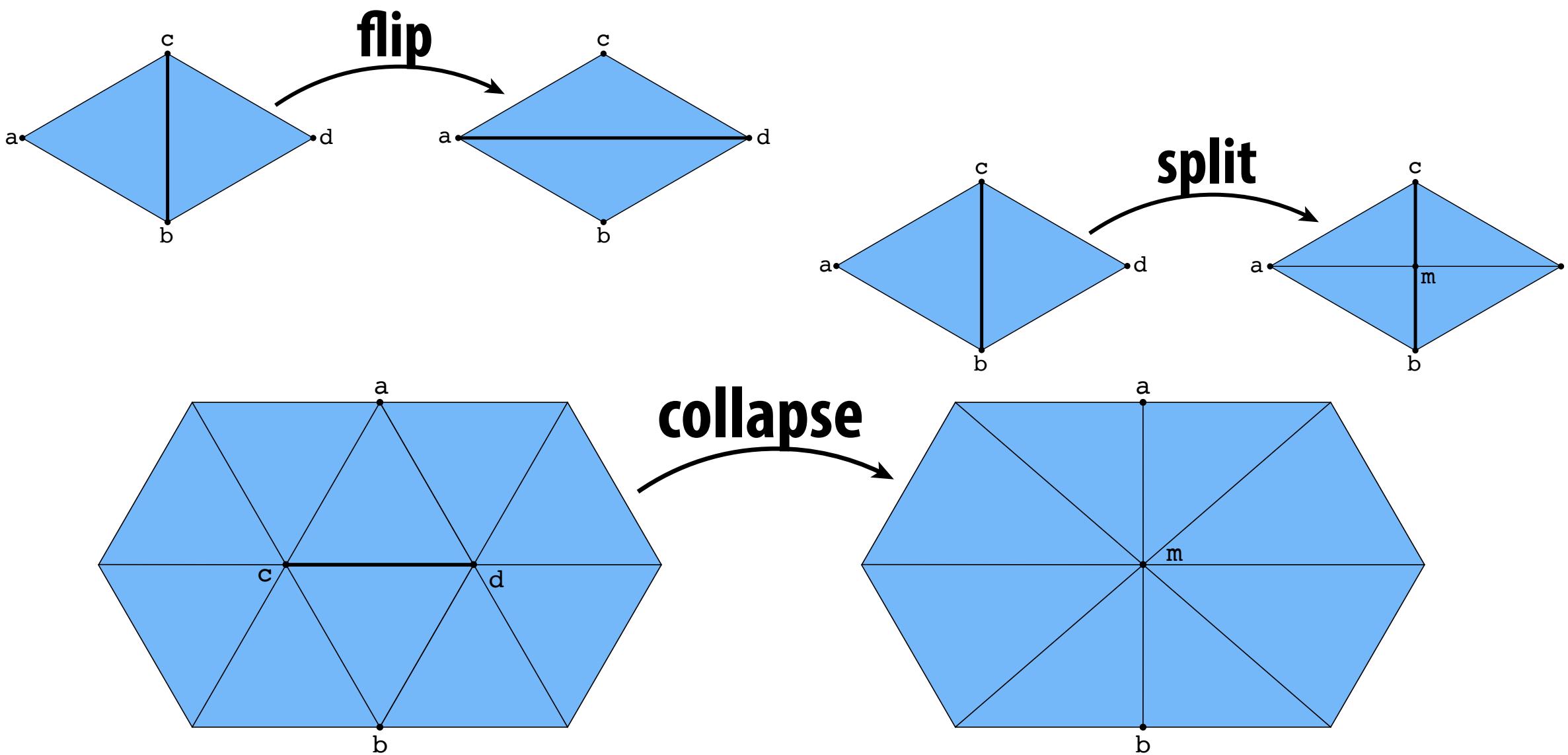
# Connectivity vs. Geometry

- Recall manifold conditions (fans not fins):
  - every edge contained in two faces
  - every vertex contained in one fan
- These conditions say nothing about vertex positions! Just connectivity
- Hence, can have perfectly good (manifold) connectivity, even if geometry is awful
- In fact, sometimes you can have perfectly good manifold connectivity for which any vertex positions give “bad” geometry!
- Can lead to confusion when debugging: mesh looks “bad”, even though connectivity is fine



# Halfedge meshes are easy to edit

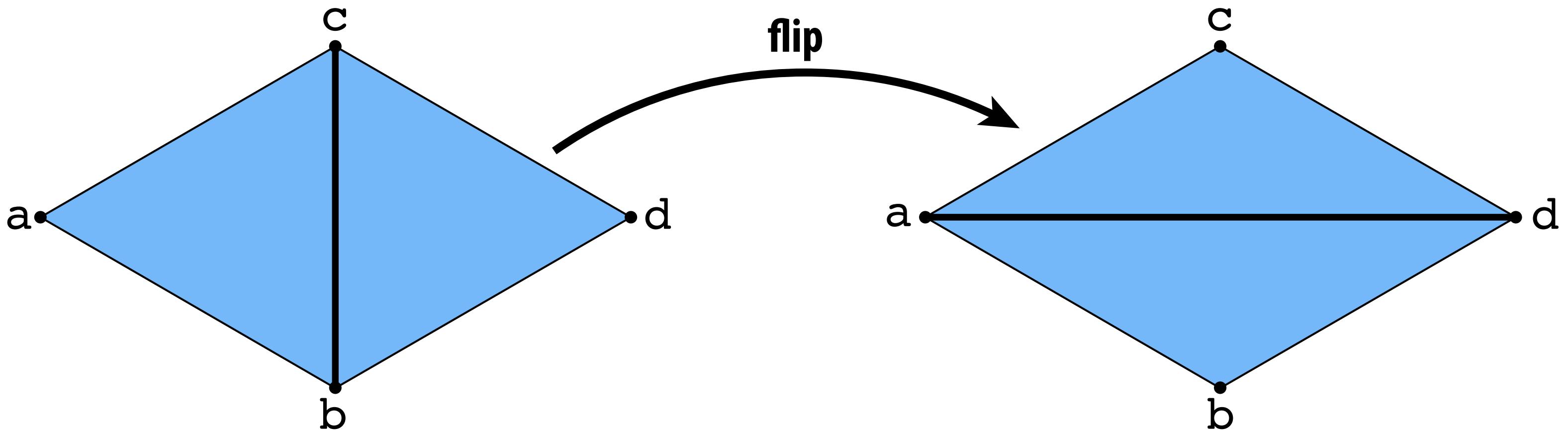
- Remember key feature of linked list: insert/delete elements
- Same story with halfedge mesh (“linked list on steroids”)
- E.g., for triangle meshes, several atomic operations:



- How? Allocate/delete elements; reassigning pointers.
- Must be careful to preserve manifoldness!

# Edge Flip (Triangles)

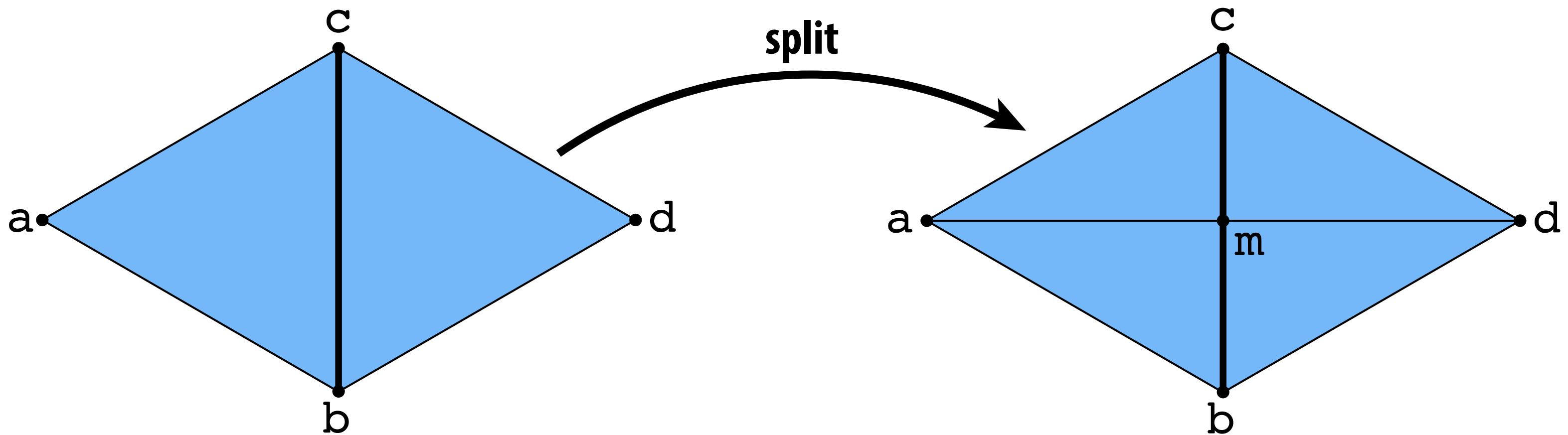
- Triangles  $(a,b,c), (b,d,c)$  become  $(a,d,c), (a,b,d)$ :



- Long list of pointer reassessments (`edge->halfedge = ...`)
- However, no elements created/destroyed.
- Q: What happens if we flip twice?
- Challenge: can you implement edge flip such that pointers are unchanged after two flips?

# Edge Split (Triangles)

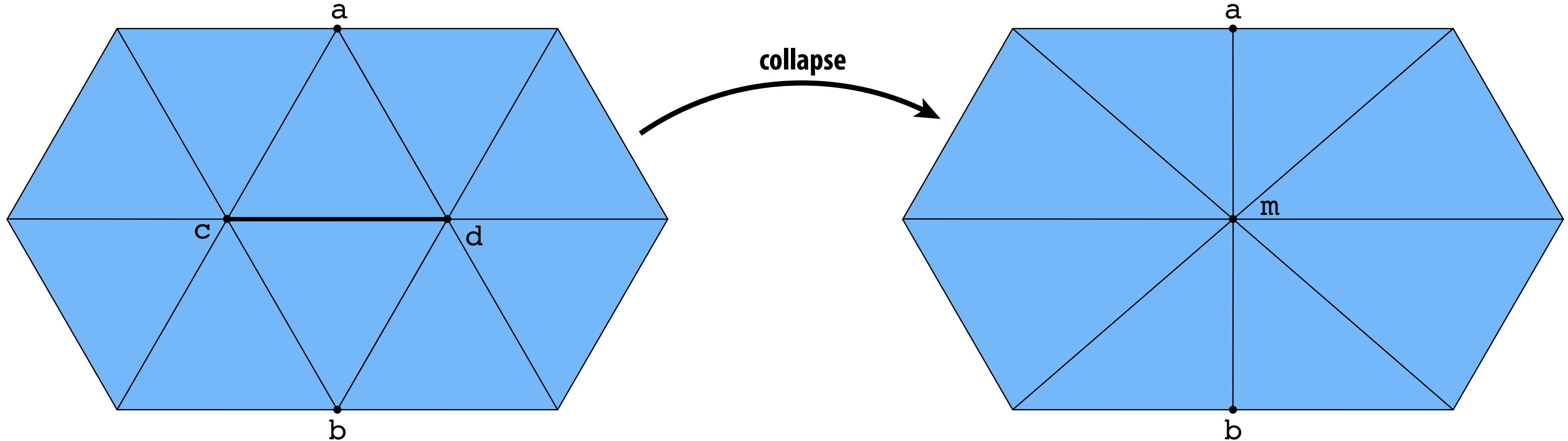
- Insert midpoint  $m$  of edge  $(c,b)$ , connect to get four triangles:



- This time, have to add new elements.
- Lots of pointer reassessments.
- Q: Can we “reverse” this operation?

# Edge Collapse (Triangles)

- Replace edge (b,c) with a single vertex m:



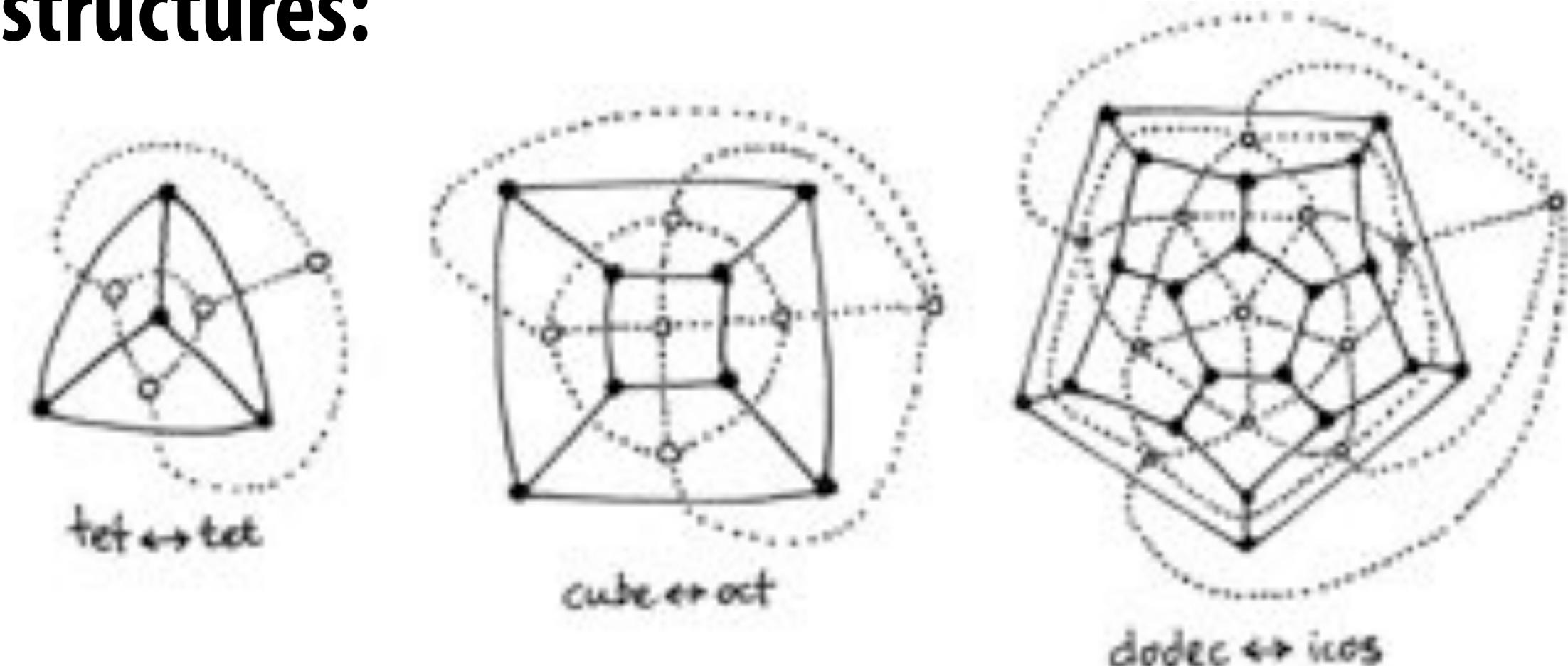
- Now have to delete elements.
- Still lots of pointer assignments!
- Q: How would we implement this with an adjacency list?
- Any other good way to do it? (E.g., different data structure?)

# Alternatives to Halfedge

Paul Heckbert (former CMU prof.)  
quadedge code - <http://bit.ly/1QZLHos>

## ■ Many very similar data structures:

- winged edge
- corner table
- quADEDGE
- ...



## ■ Each stores local neighborhood information

## ■ Similar tradeoffs relative to simple polygon list:

- **CONS:** additional storage, incoherent memory access
- **PROS:** better access time for individual elements, intuitive traversal of local neighborhoods

## ■ With some thought\*, can design halfedge-type data structures with coherent data storage, support for non manifold connectivity, etc.

\*see for instance <http://geometry-central.net/>

# Comparison of Polygon Mesh Data Structures

	Adjacency List	Incidence Matrices	Halfedge Mesh
constant-time neighborhood access?	NO	YES	YES
easy to add/remove mesh elements?	NO	NO	YES
nonmanifold geometry?	YES	YES	NO

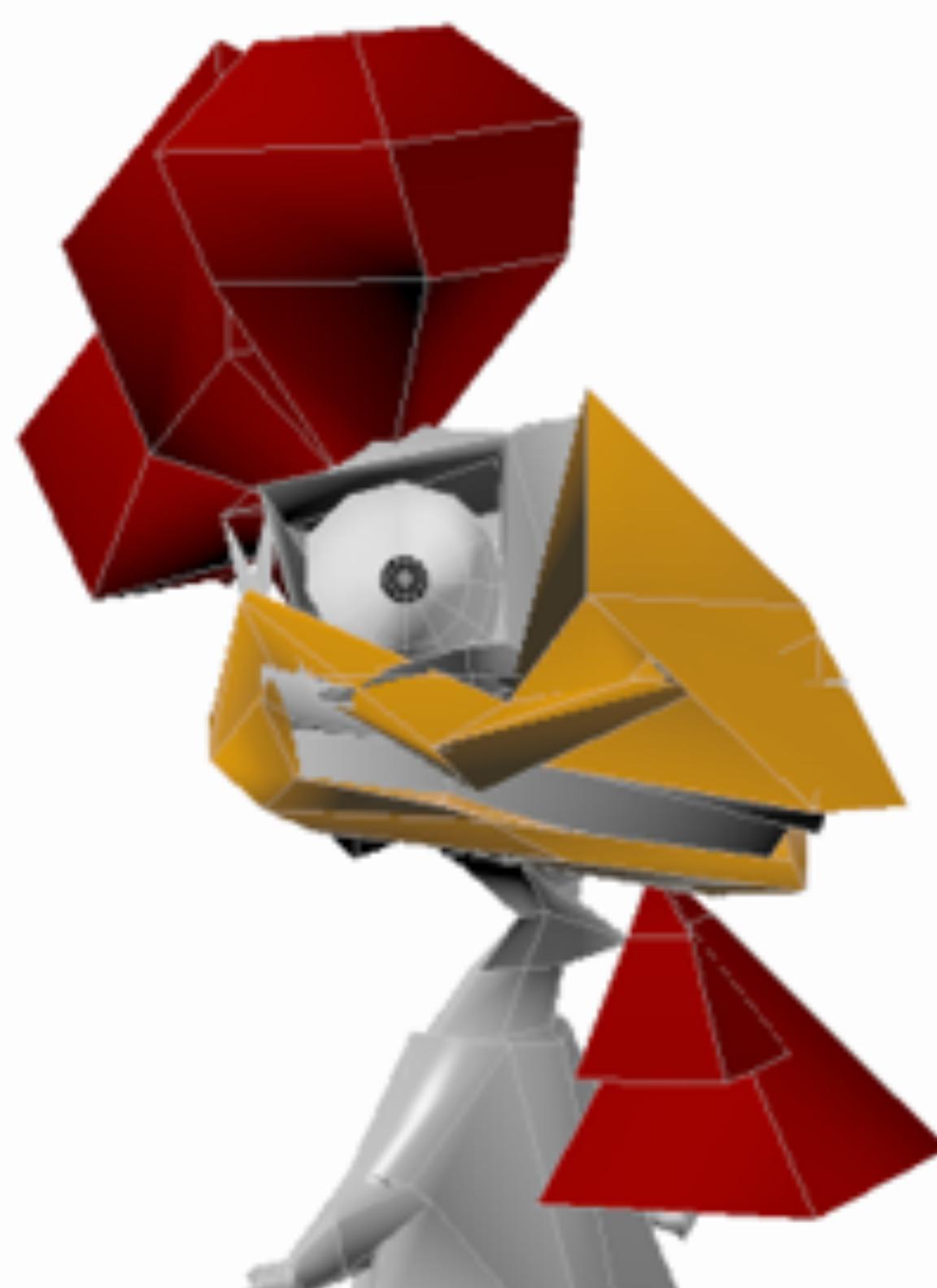
Conclusion: pick the right data structure for the job!

**Ok, but what can we actually do with our  
fancy new data structures?**

# Subdivision Modeling

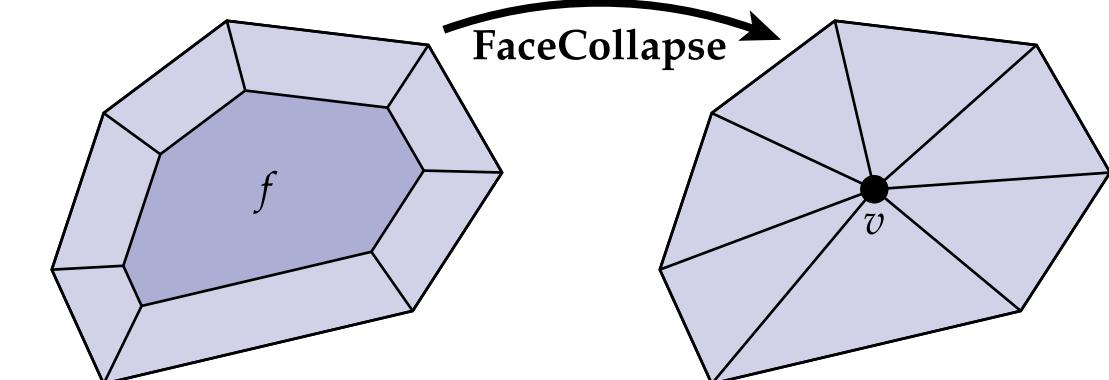
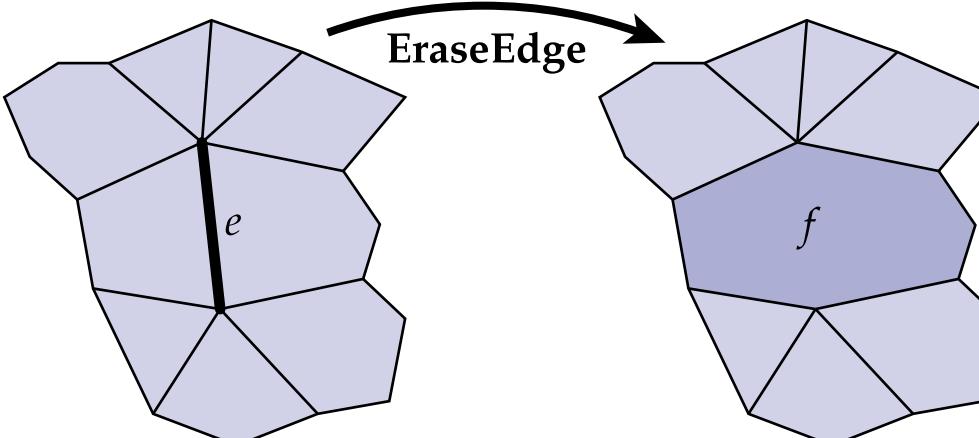
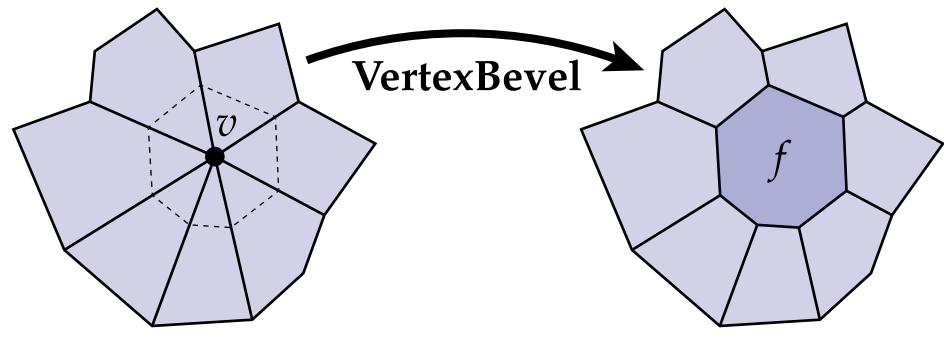
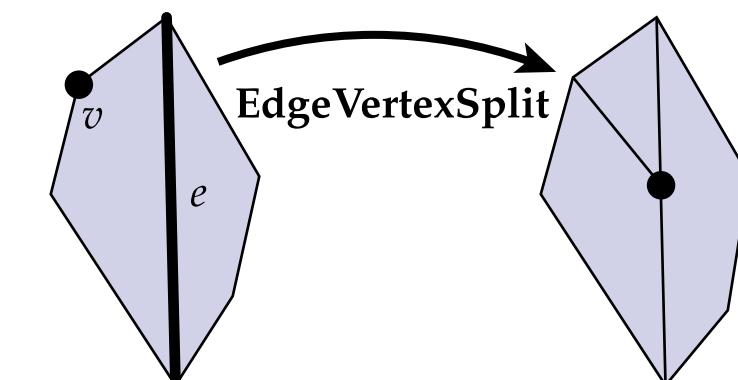
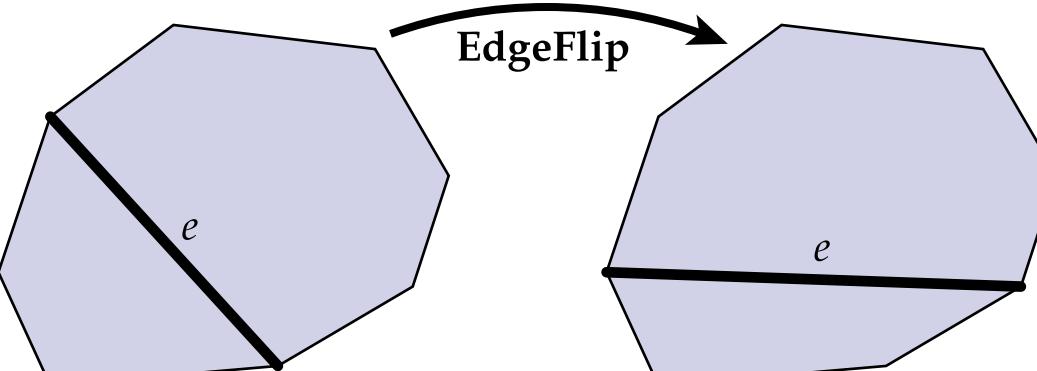
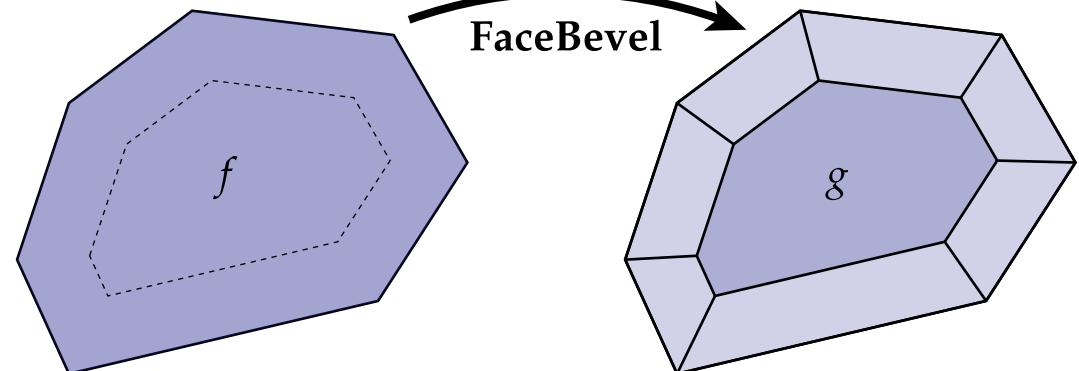
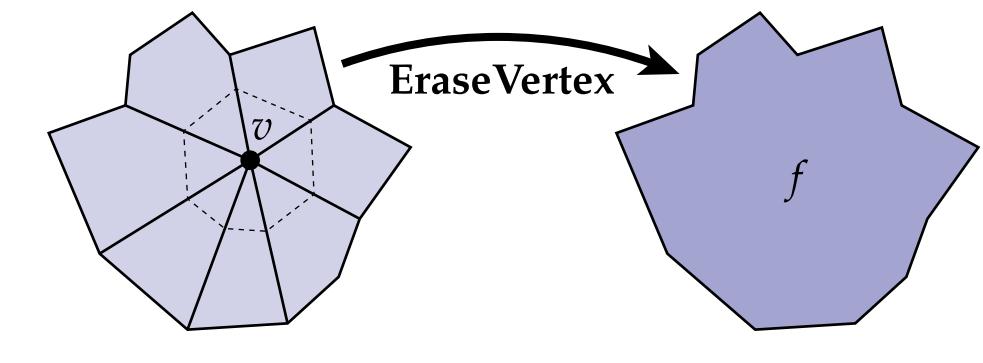
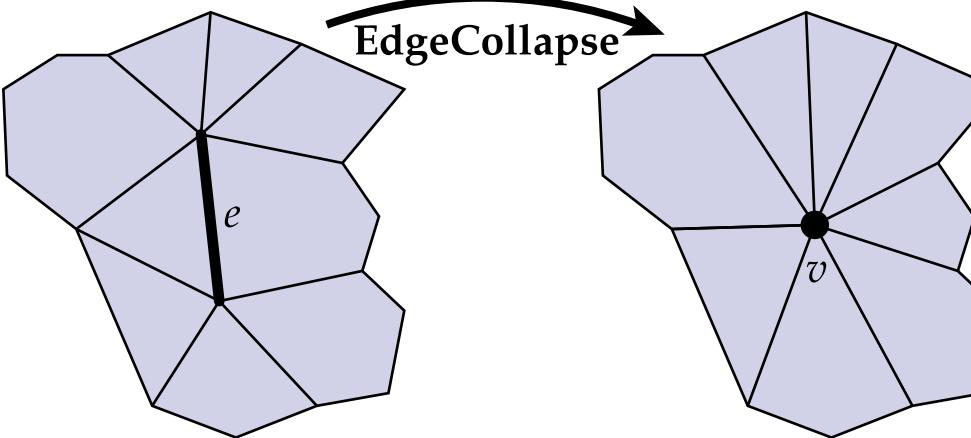
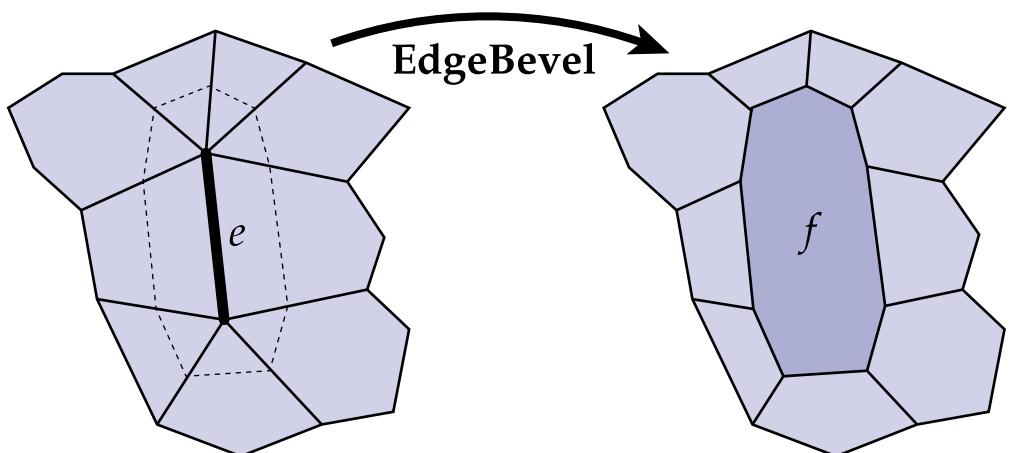
- Common modeling paradigm in modern 3D tools:

- Coarse “control cage”
- Perform local operations to control/edit shape
- Global subdivision process determines final surface



# Subdivision Modeling—Local Operations

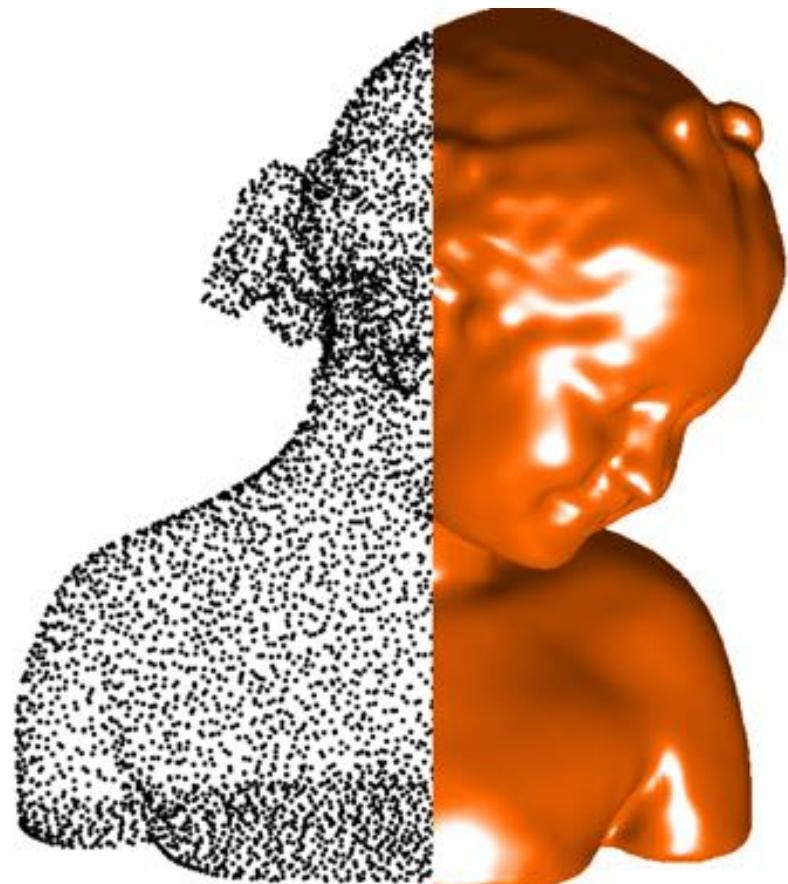
- For general polygon meshes, we can dream up lots of local mesh operations that might be useful for modeling:



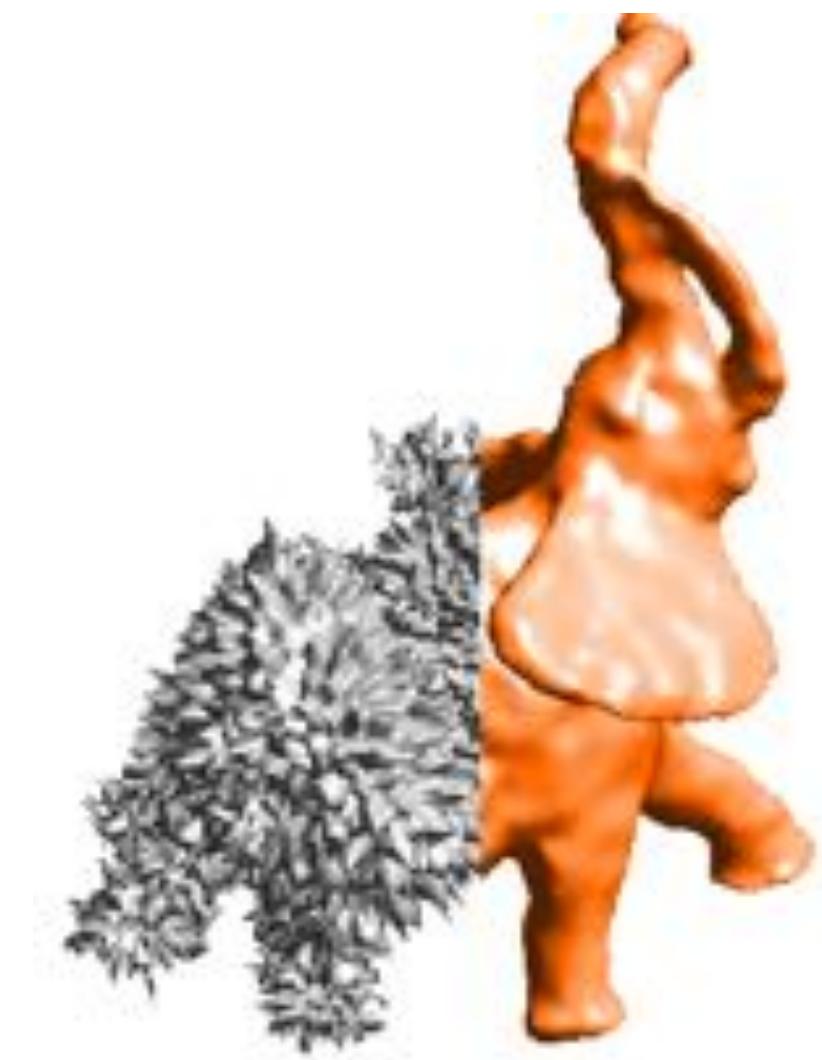
...and many, many more!

**What else can we do with geometric data?**

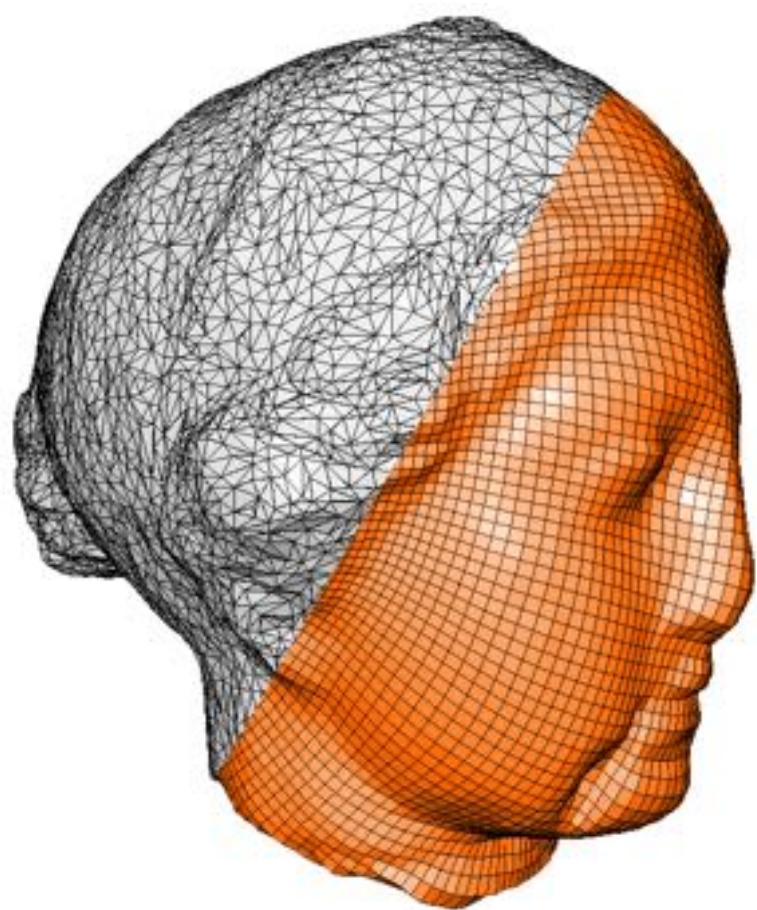
# Geometry Processing



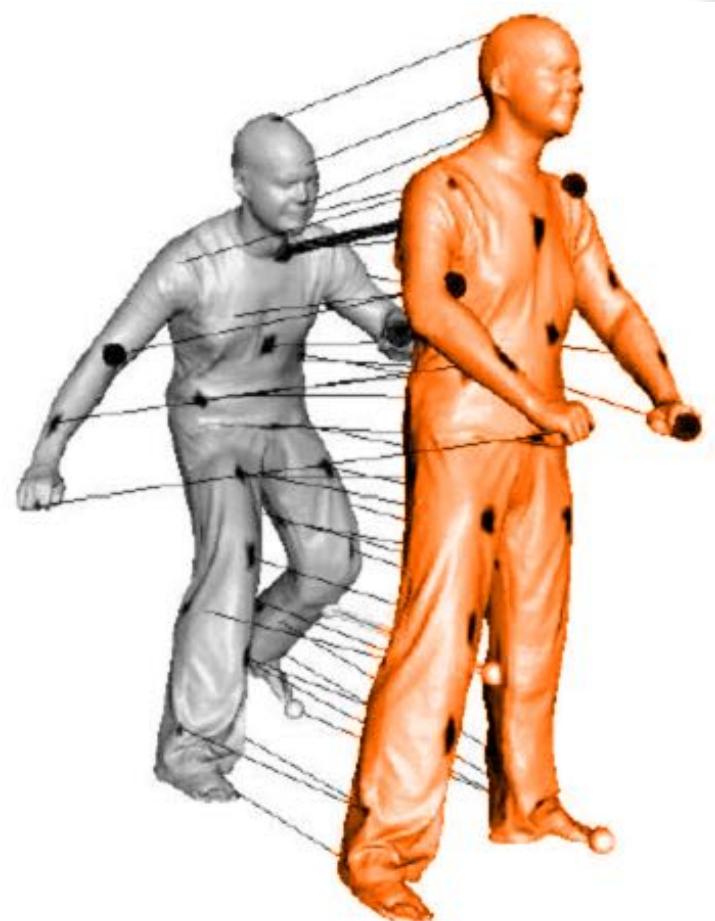
reconstruction



filtering



remeshing



shape analysis



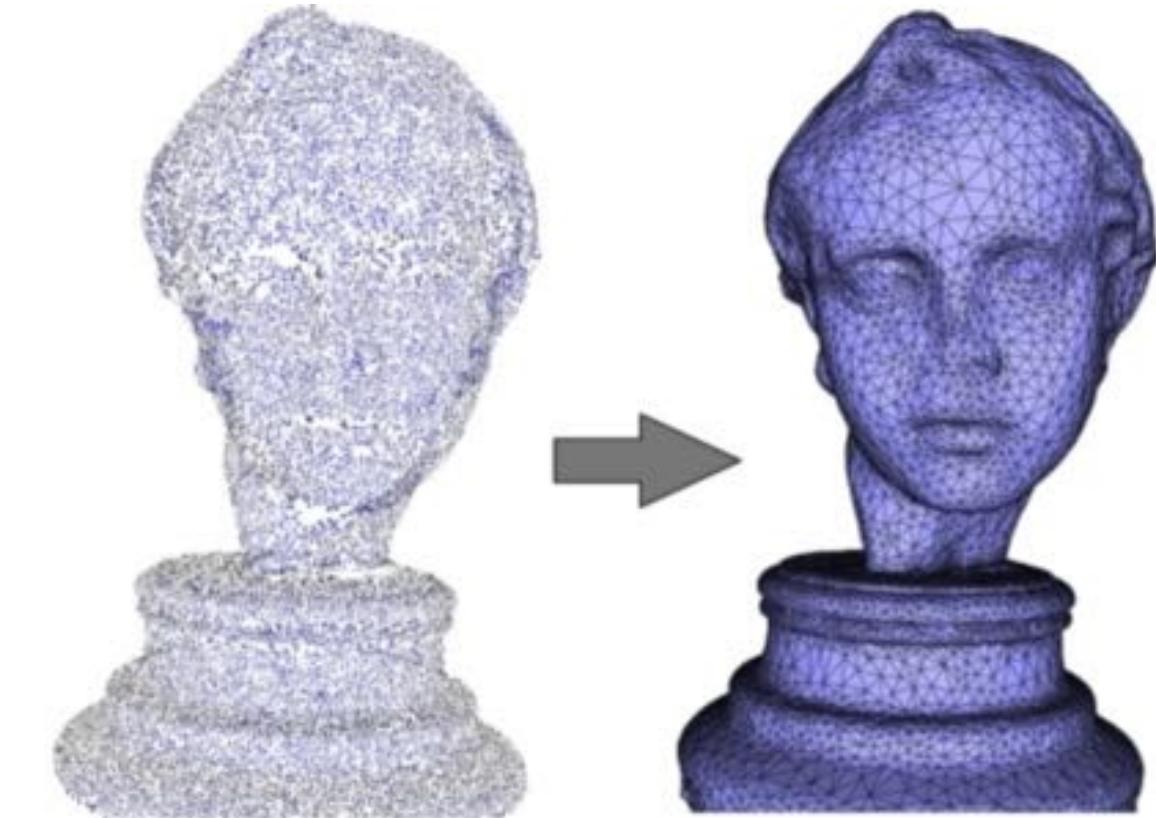
parameterization



compression

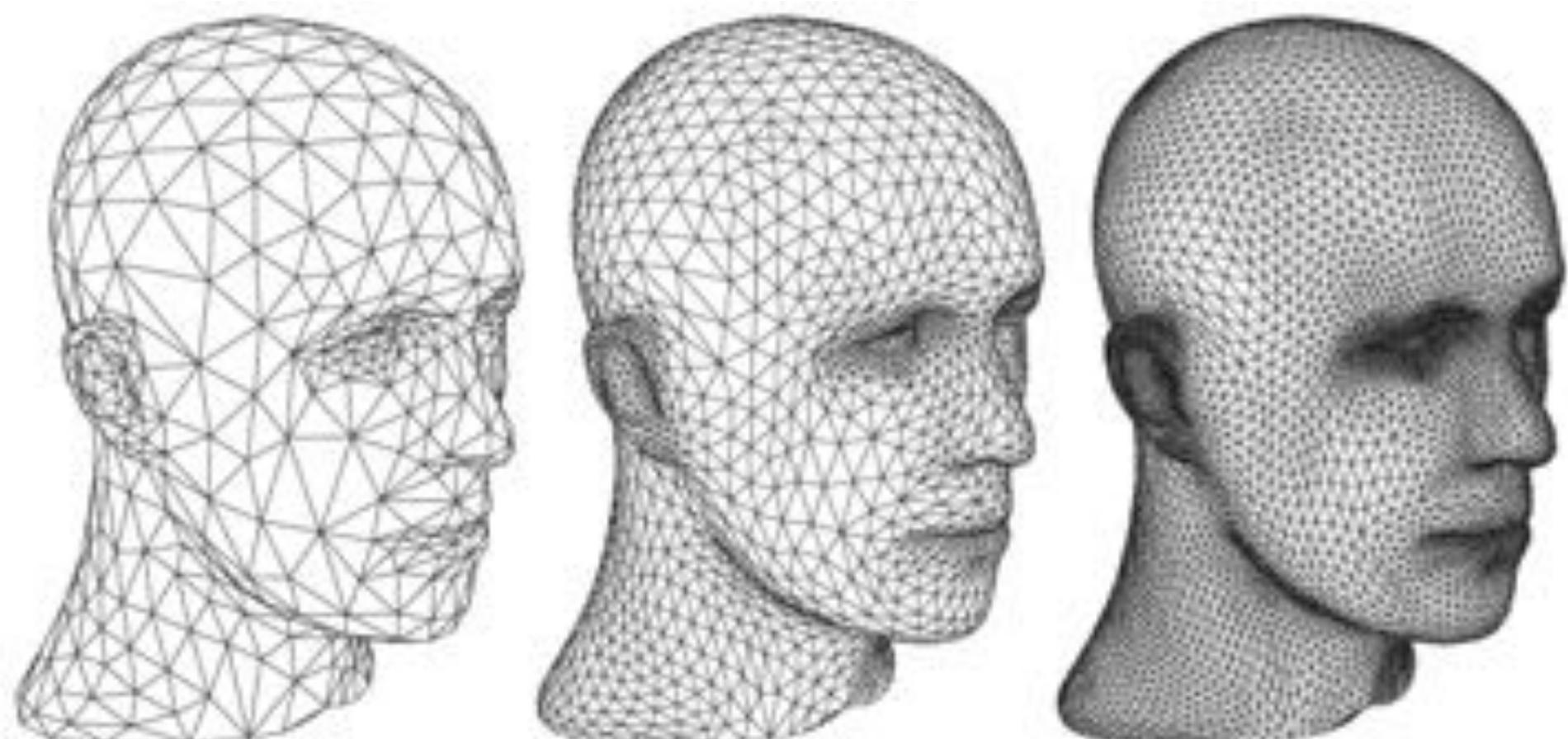
# Geometry Processing: Reconstruction

- Given samples of geometry, reconstruct surface
- What are “samples”? Many possibilities:
  - points, points & normals, ...
  - image pairs / sets (multi-view stereo)
  - line density integrals (MRI/CT scans)
- How do you get a surface? Many techniques:
  - silhouette-based (visual hull)
  - Voronoi-based (e.g., power crust)
  - PDE-based (e.g., Poisson reconstruction)
  - Radon transform / isosurfacing (marching cubes)



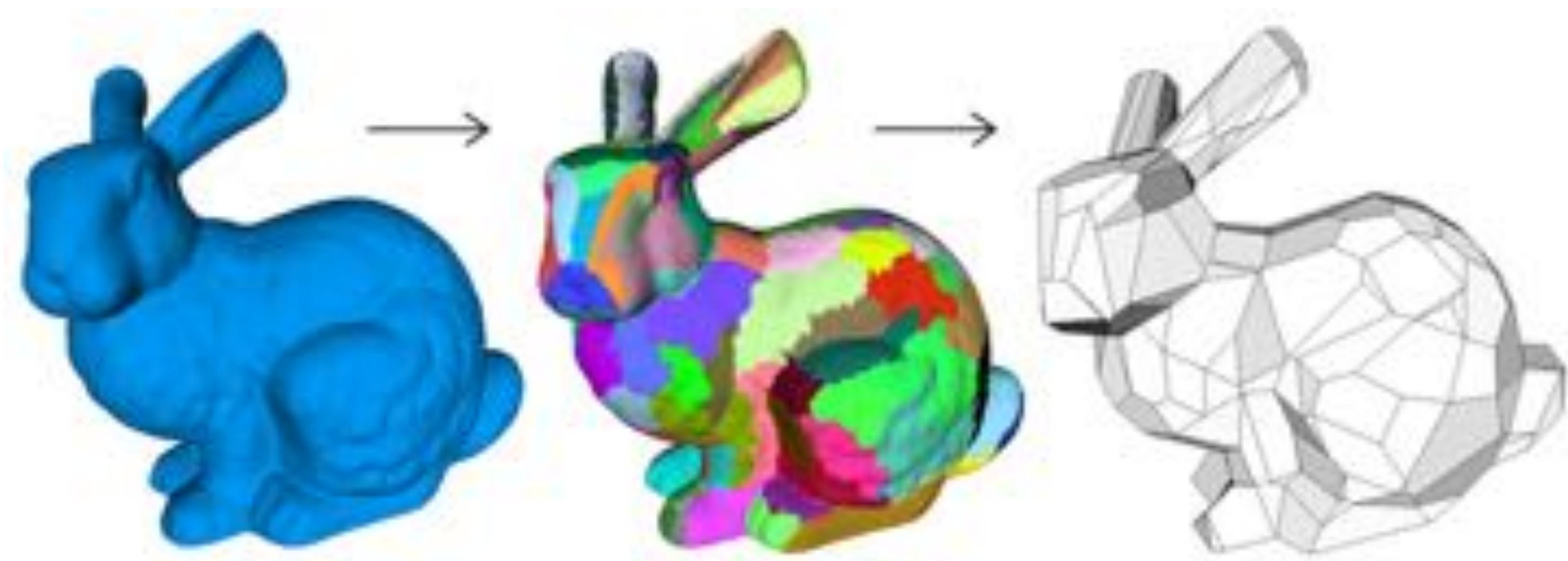
# Geometry Processing: Upsampling

- Increase resolution via interpolation
- Images: e.g., bilinear, bicubic interpolation
- Polygon meshes:
  - subdivision
  - bilateral upsampling
  - ...



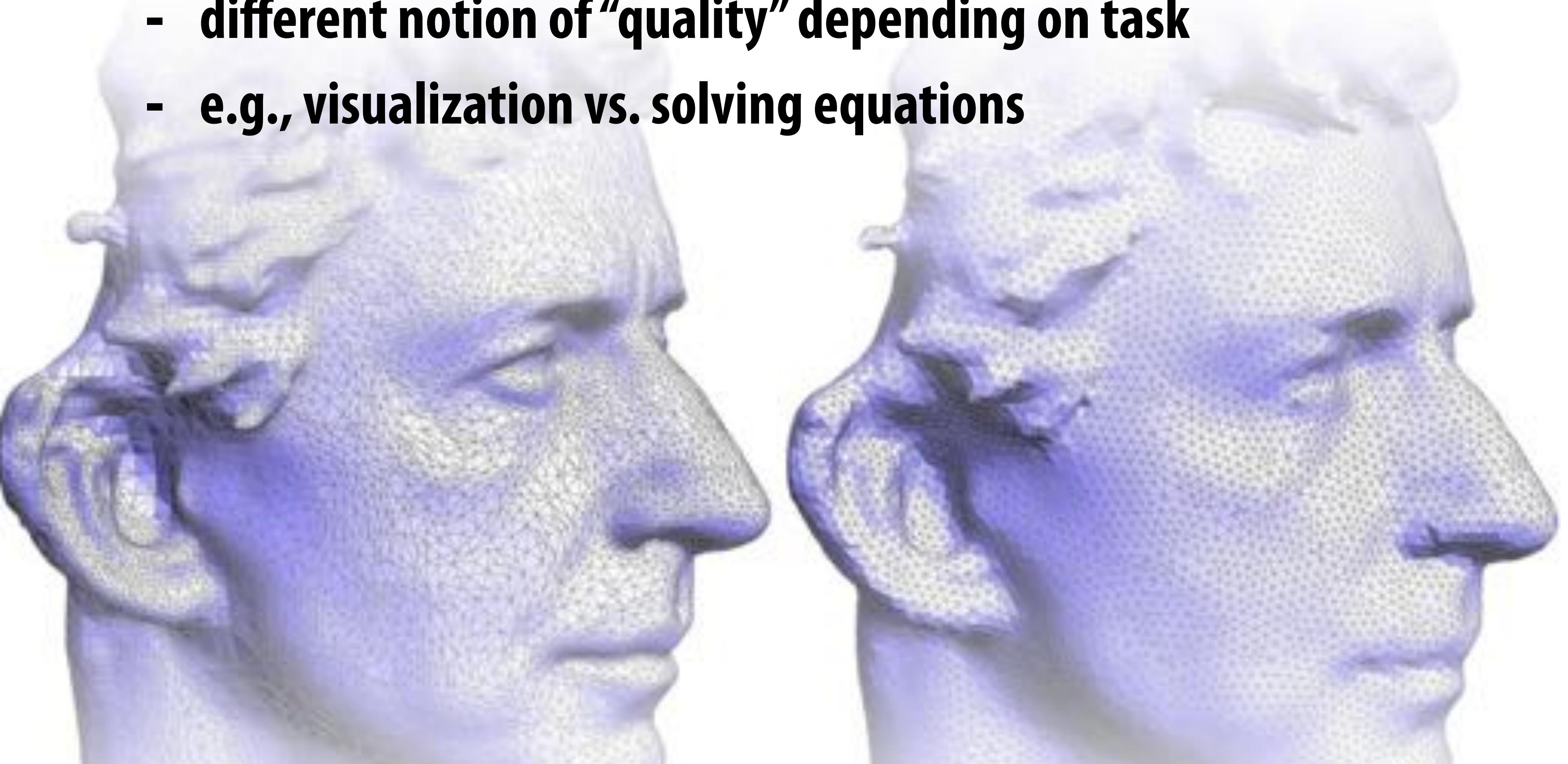
# Geometry Processing: Downsampling

- Decrease resolution; try to preserve shape/appearance
- Images: nearest-neighbor, bilinear, bicubic interpolation
- Point clouds: subsampling (just take fewer points!)
- Polygon meshes:
  - iterative decimation, variational shape approximation, ...



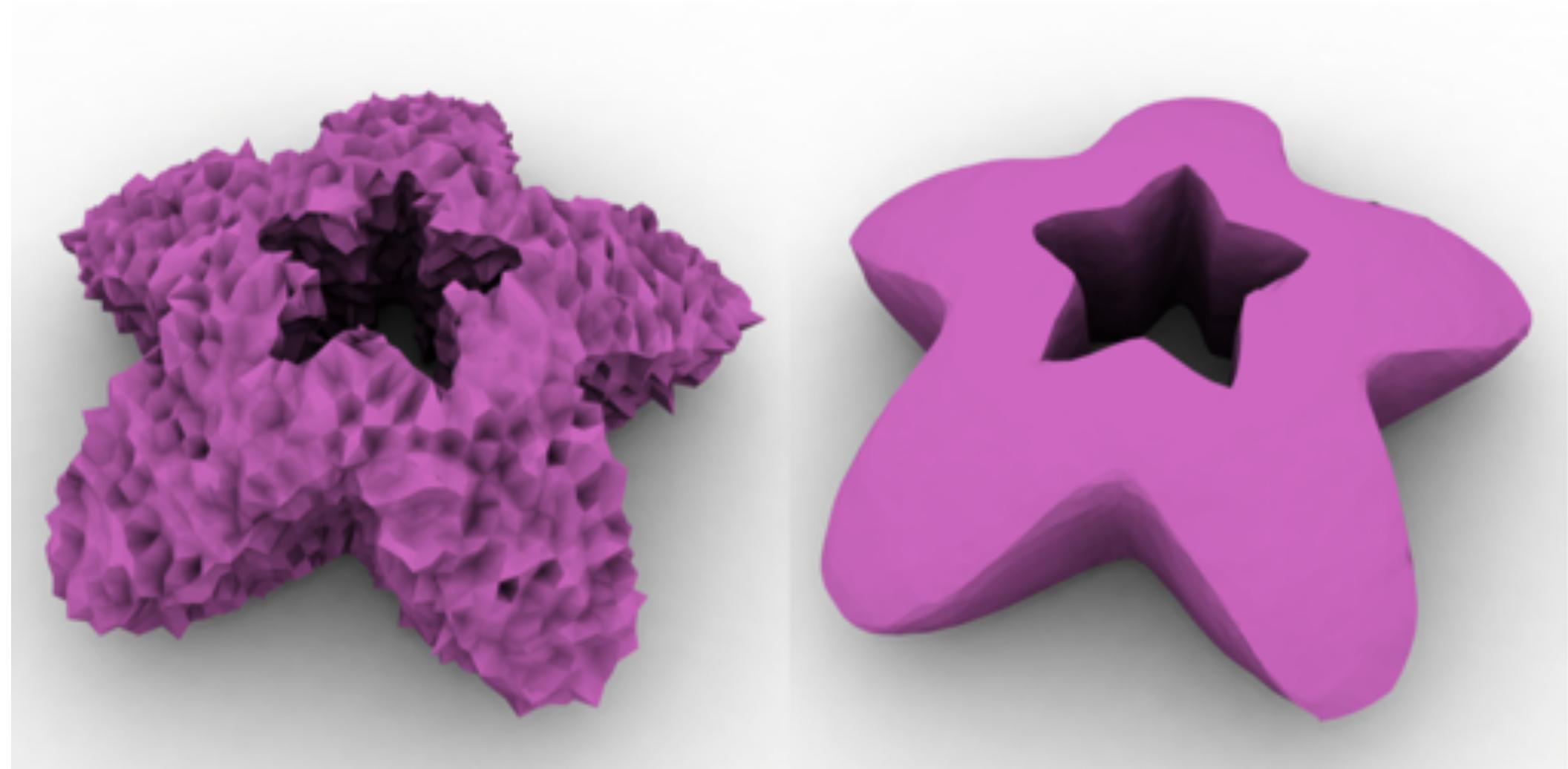
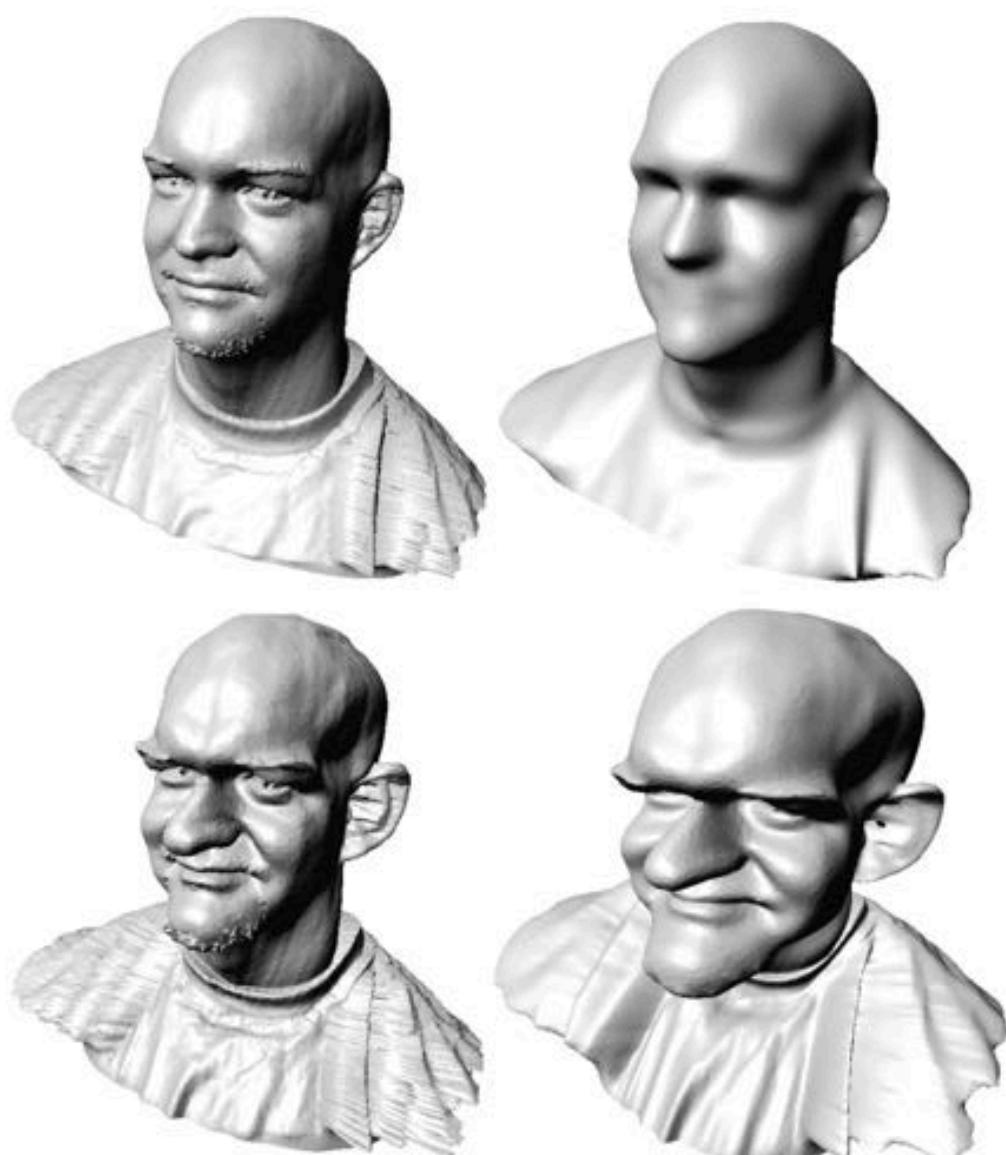
# Geometry Processing: Resampling

- Modify sample distribution to improve quality
- Images: not an issue! (Pixels always stored on a regular grid)
- Meshes: shape of polygons is extremely important!
  - different notion of “quality” depending on task
  - e.g., visualization vs. solving equations



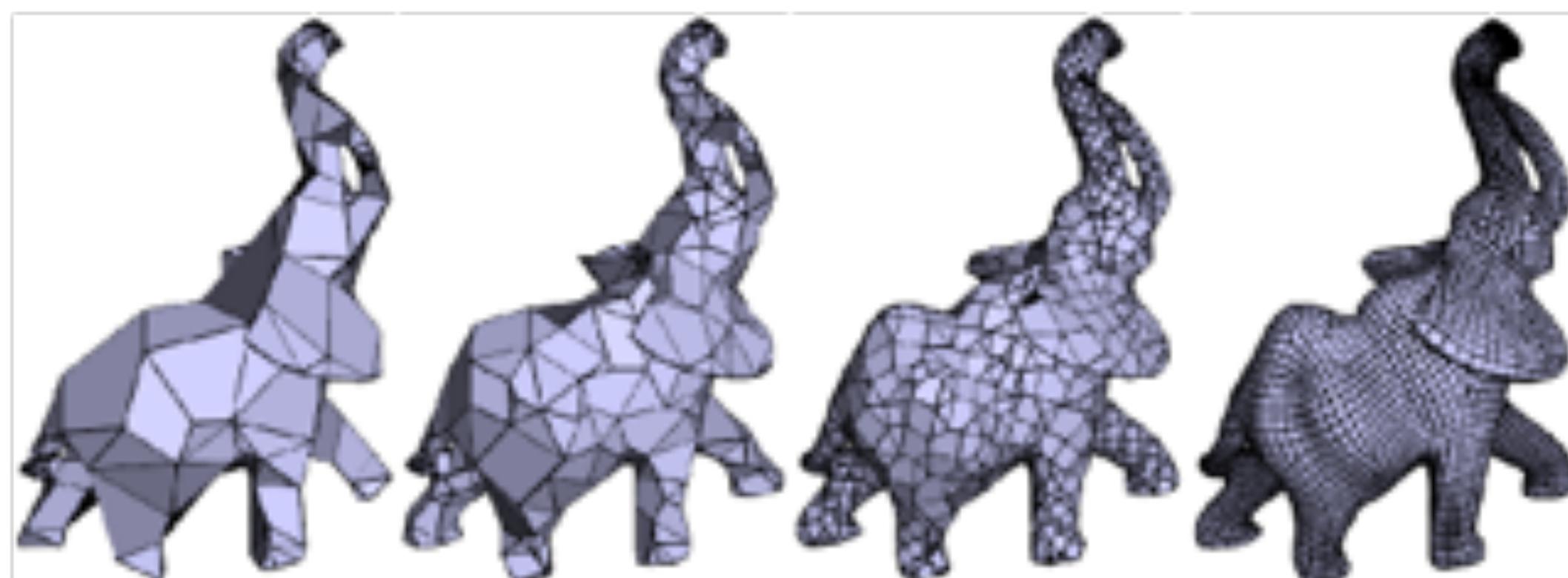
# Geometry Processing: Filtering

- Remove noise, or emphasize important features (e.g., edges)
- Images: blurring, bilateral filter, edge detection, ...
- Polygon meshes:
  - curvature flow
  - bilateral filter
  - spectral filter



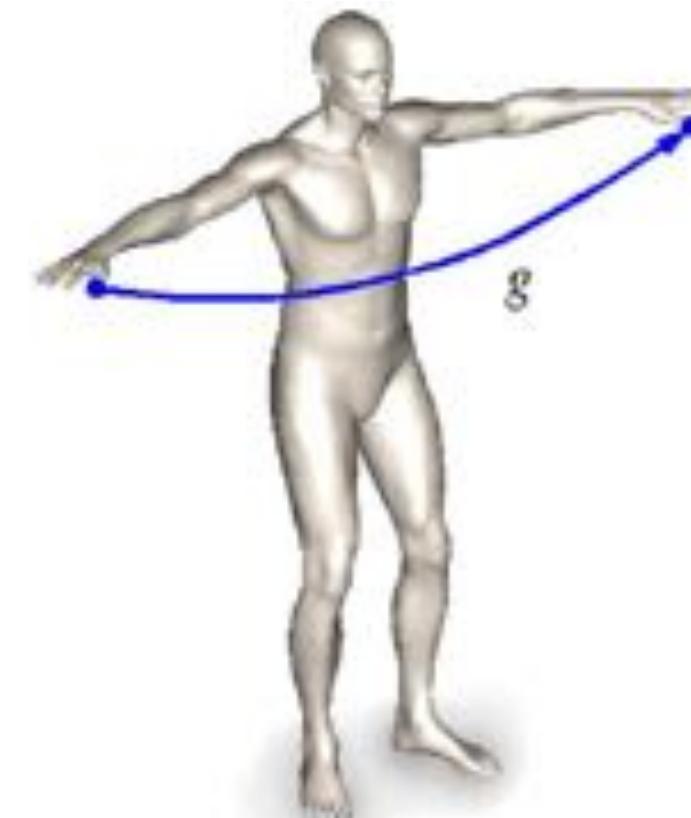
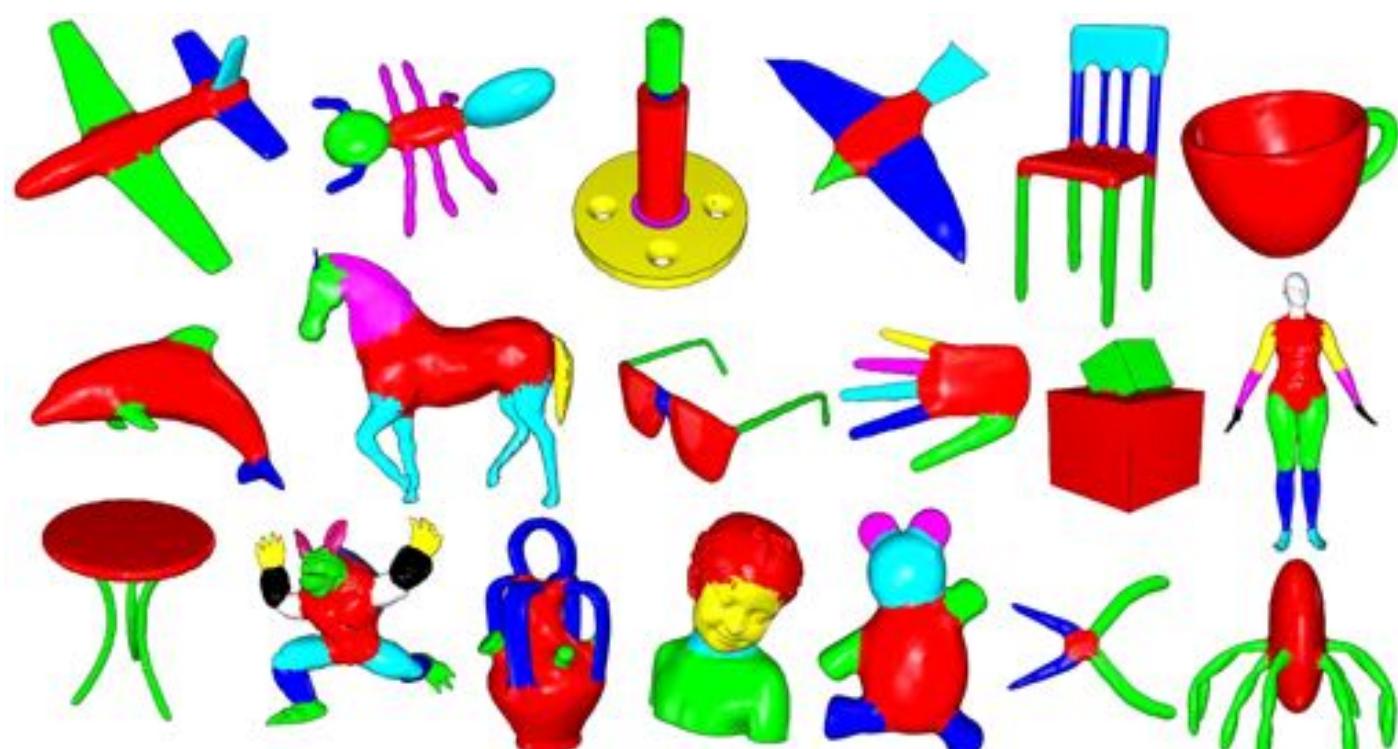
# Geometry Processing: Compression

- Reduce storage size by eliminating redundant data/approximating unimportant data
- Images:
  - run-length, Huffman coding - lossless
  - cosine/wavelet (JPEG/MPEG) - lossy
- Polygon meshes:
  - compress geometry and connectivity
  - many techniques (lossy & lossless)



# Geometry Processing: Shape Analysis

- Identify/understand important semantic features
- Images: computer vision, segmentation, face detection, ...
- Polygon meshes:
  - segmentation, correspondence, symmetry detection, ...



Extrinsic symmetry



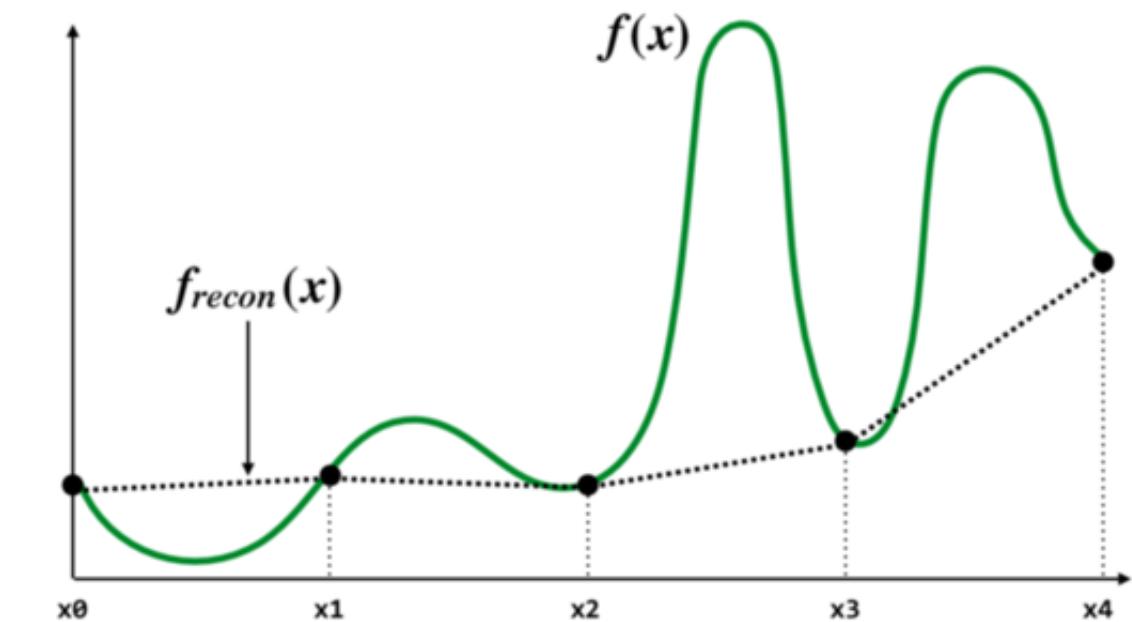
Intrinsic symmetry



**Enough overview—  
Let's process some geometry!**

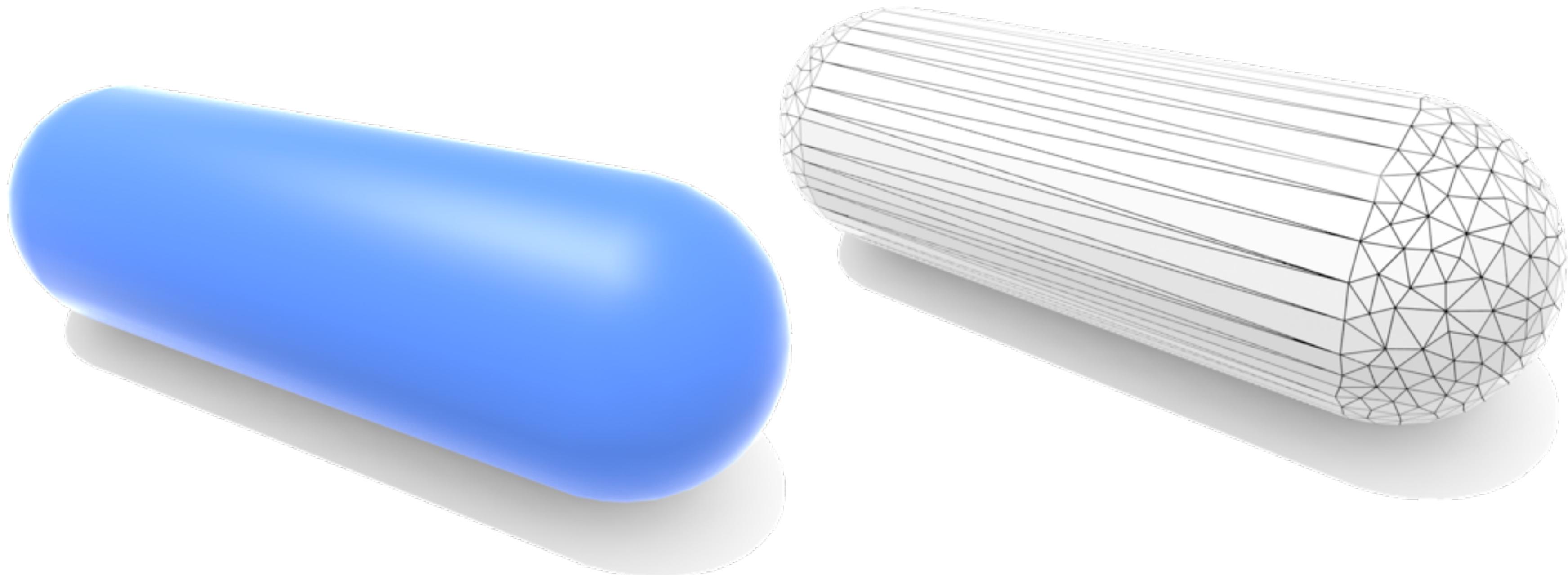
# Remeshing as resampling

- Remember our discussion of aliasing
- Bad sampling makes signal appear different than it really is
- E.g., undersampled curve looks flat
- Geometry is no different!
  - undersampling destroys features
  - oversampling bad for performance



# What makes a “good” mesh?

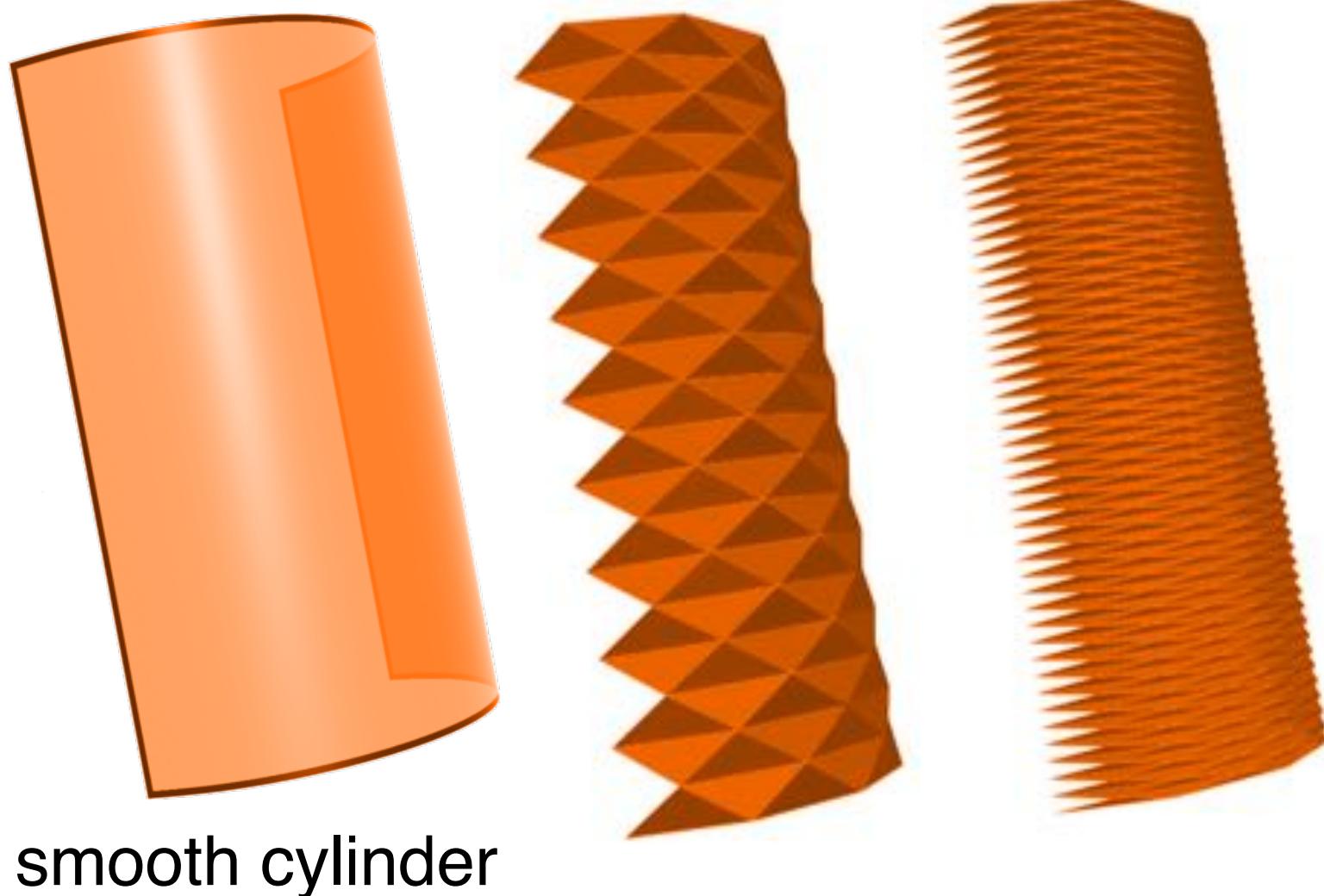
- One idea: good approximation of original shape!
- Keep only elements that contribute information about shape
- Add additional information where, e.g., curvature is large



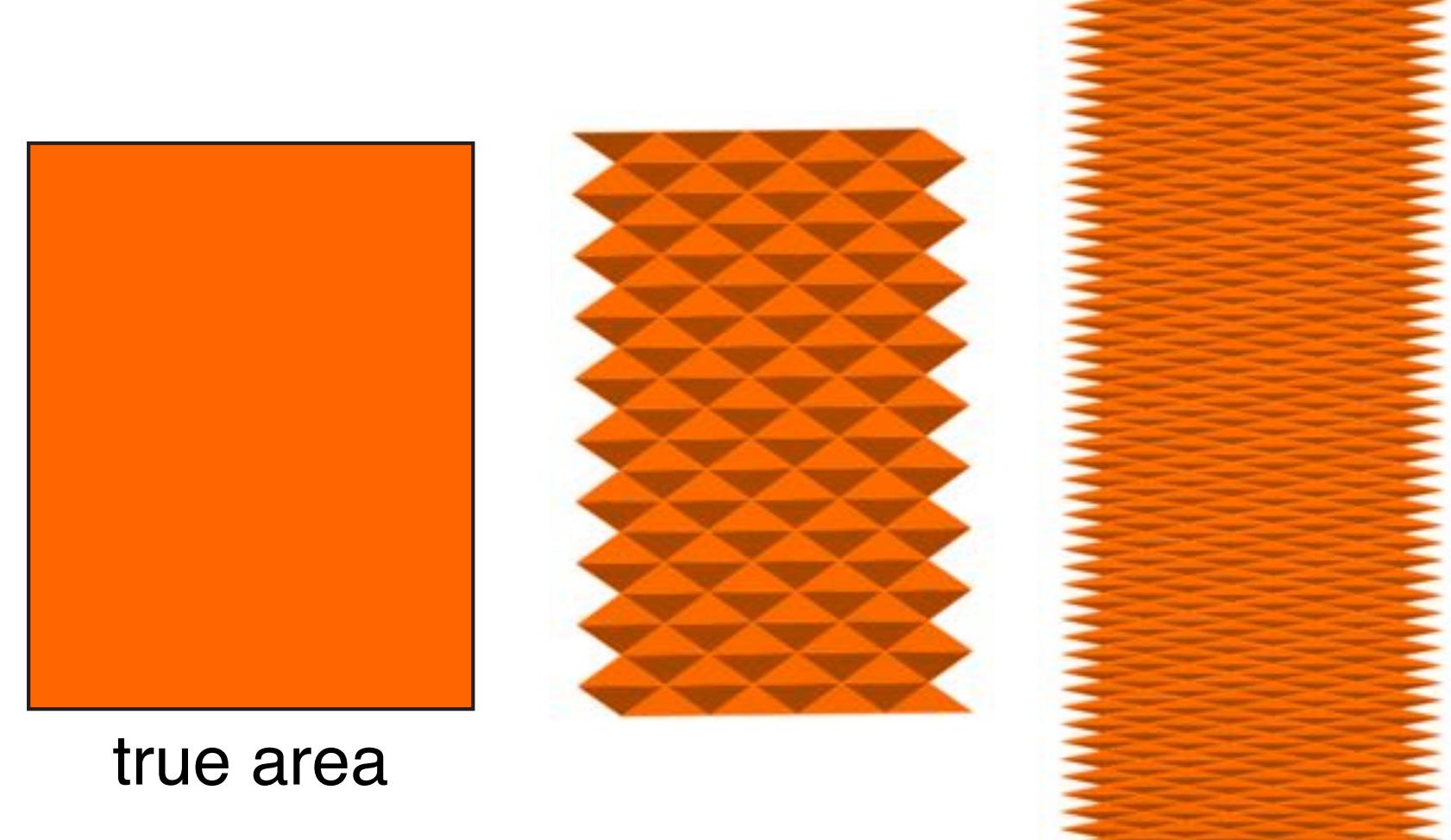
# Approximation of position is not enough!

- Just because the vertices of a mesh are close to the surface it approximates does not mean it's a good approximation!
- Can still have wrong appearance, wrong area, wrong...
- Need to consider other factors\*, e.g., close approximation of surface normals

**vertices exactly on smooth cylinder**

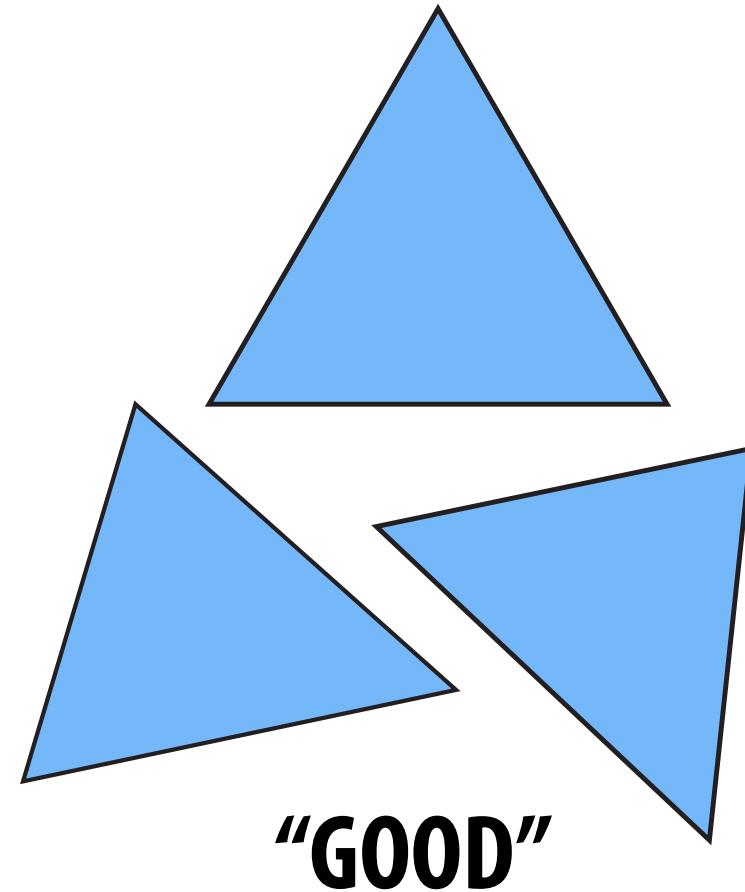


**flattening of smooth cylinder & meshes**

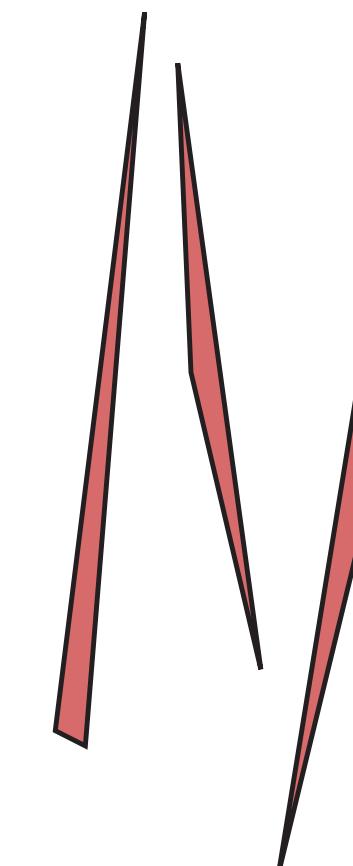


# What else makes a “good” triangle mesh?

## ■ Another rule of thumb: triangle



“GOOD”

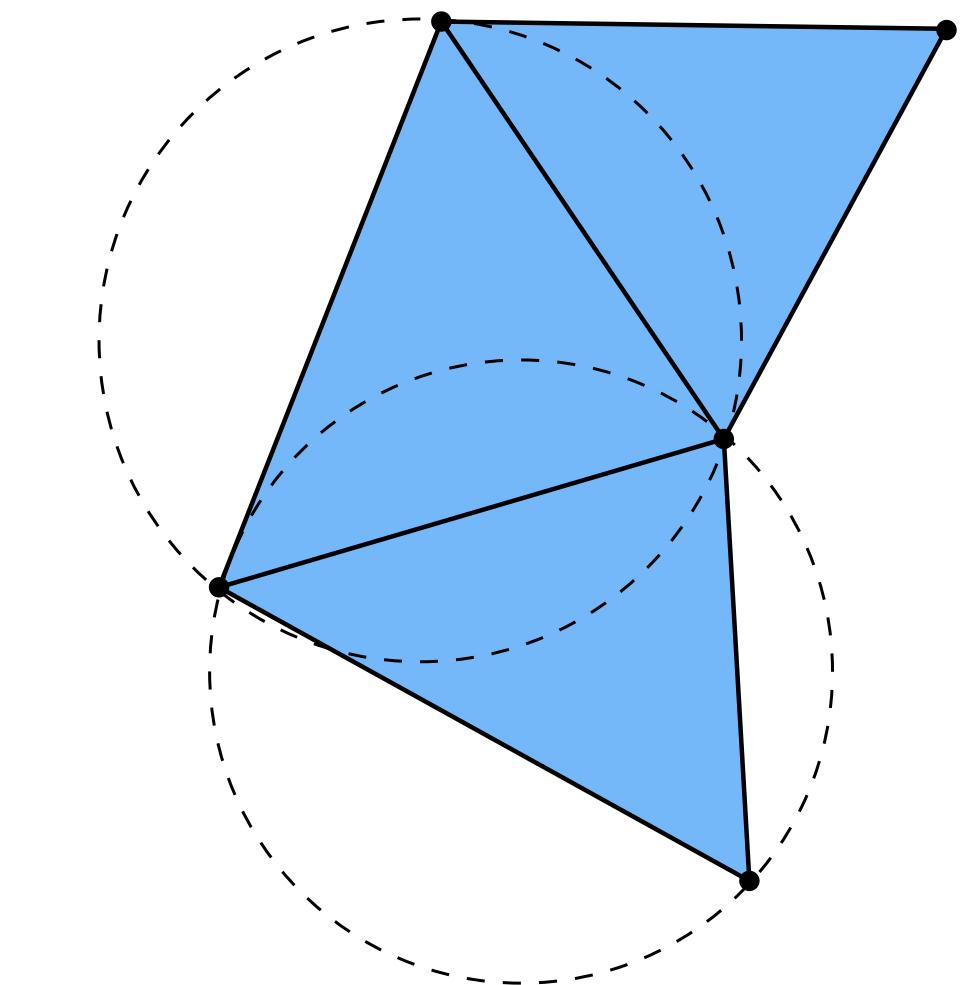


“BAD”

pronunciation:



DELAUNAY

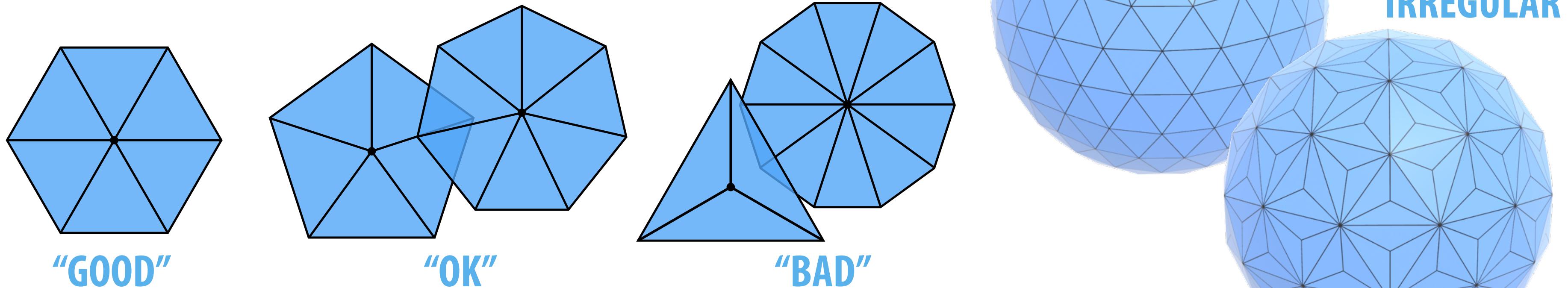


- E.g., all angles close to 60 degrees
- More sophisticated condition: Delaunay (empty circumcircles)
  - often helps with numerical accuracy/stability
  - coincides with shockingly many other desirable properties  
**(maximizes minimum angle, provides smoothest interpolation, guarantees maximum principle...)**
- Tradeoffs w/ good geometric approximation\*
  - e.g., long & skinny might be “more efficient”

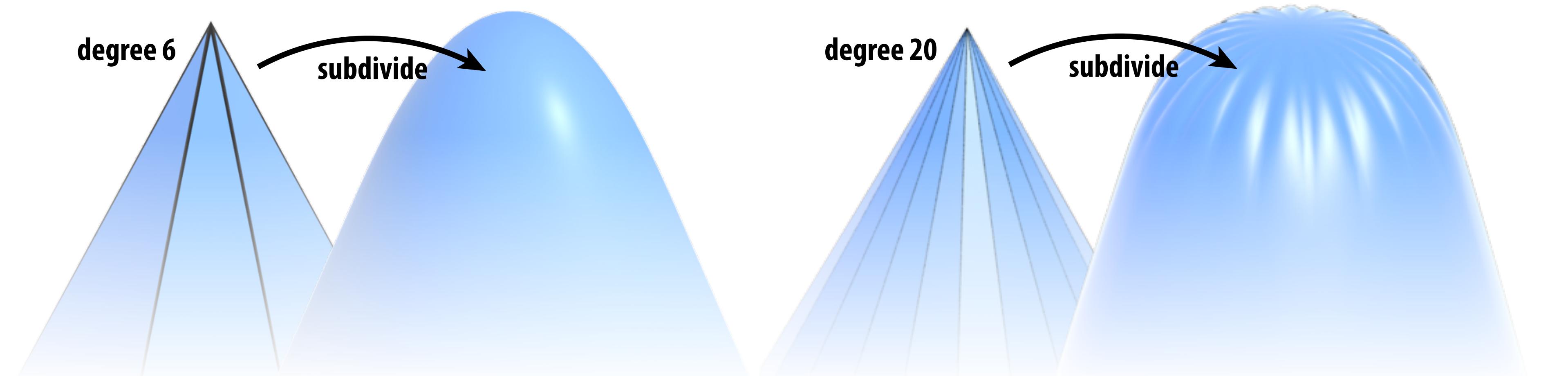
\*see Shewchuk, “What is a Good Linear Element”

# What else constitutes a “good” mesh?

- Another rule of thumb: regular vertex degree
- Degree 6 for triangle mesh, 4 for quad mesh



Why? Better polygon shape; more regular computation; smoother subdivision:



**Fact: in general, can't have regular vertex degree everywhere!**

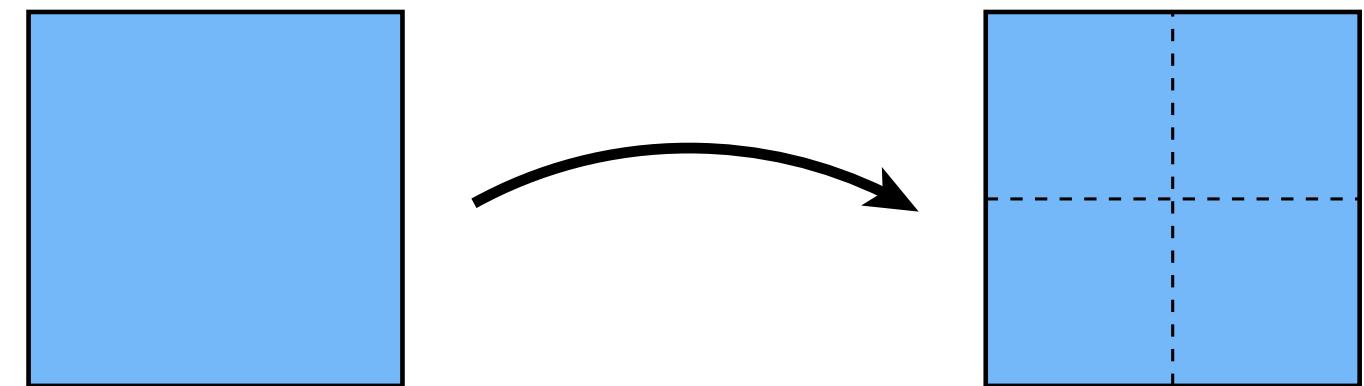
# **How do we upsample a mesh?**

# Upsampling via Subdivision

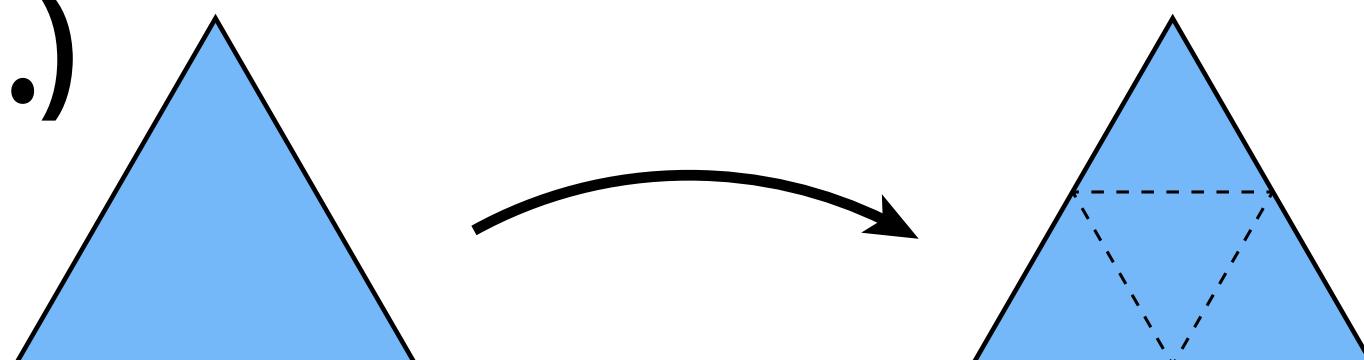
- Repeatedly split each element into smaller pieces
- Replace vertex positions with weighted average of neighbors

- Main considerations:

- interpolating vs. approximating

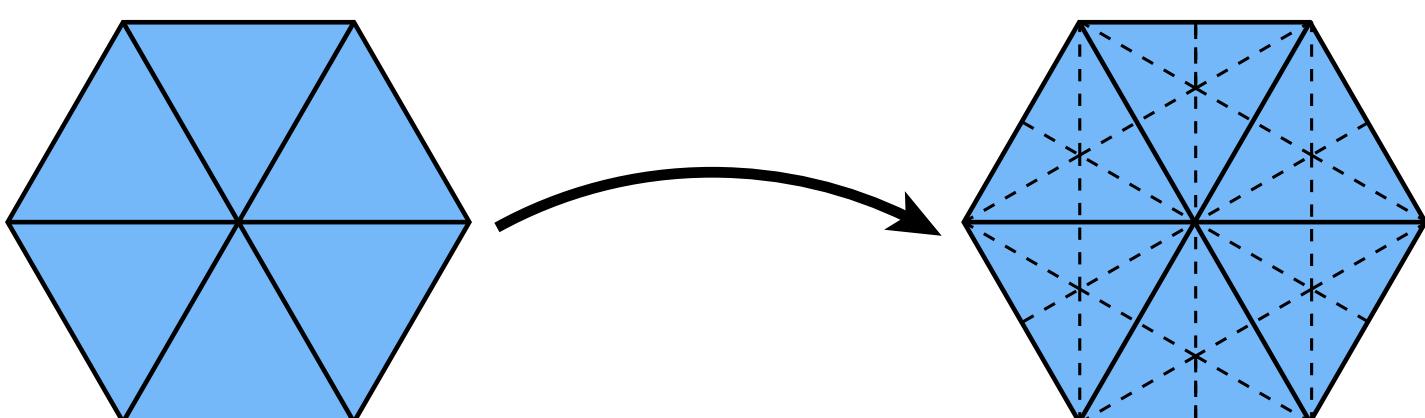


- limit surface continuity ( $C^1, C^2, \dots$ )



- Many options:

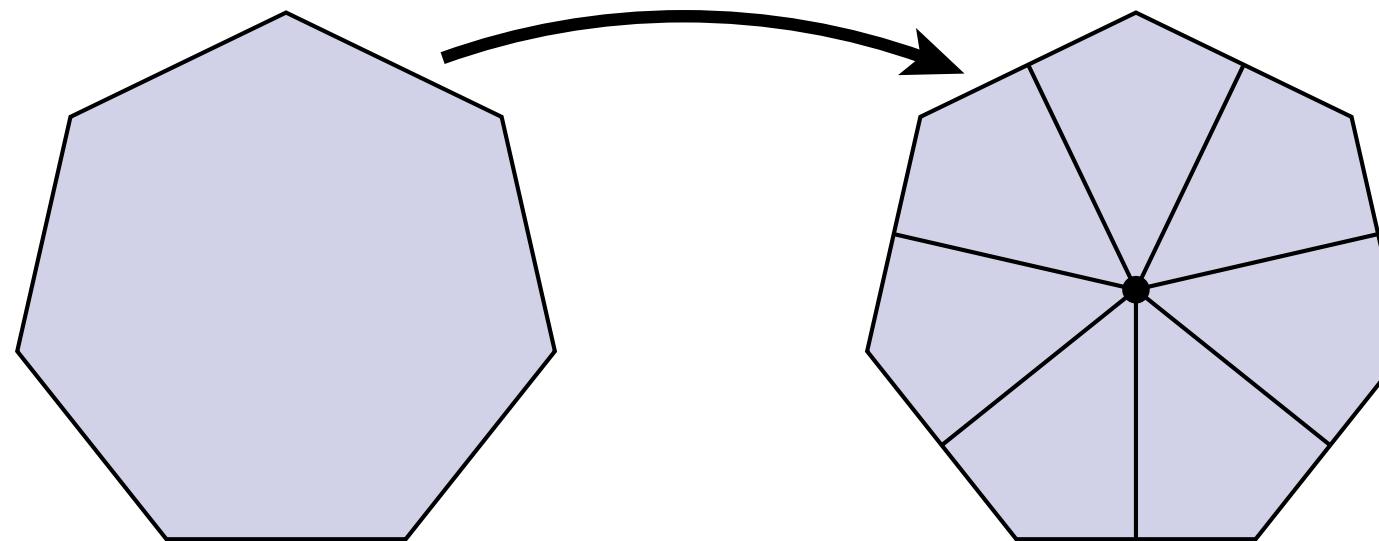
- Quad: Catmull-Clark



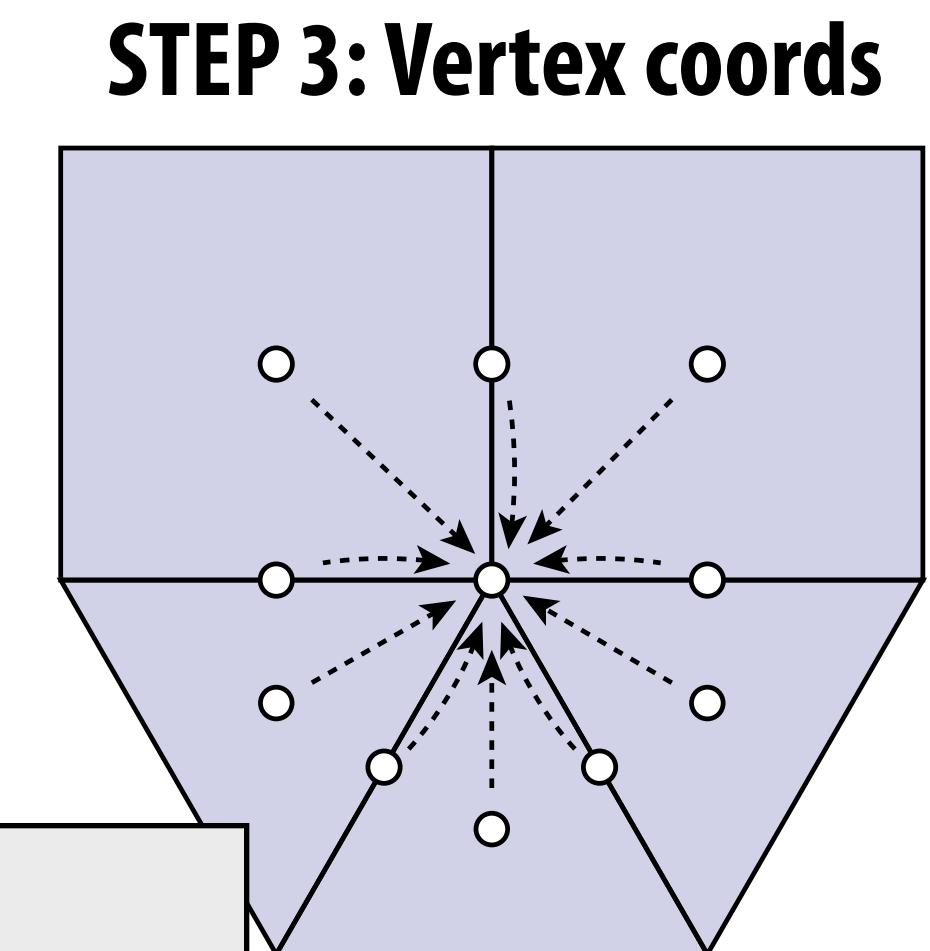
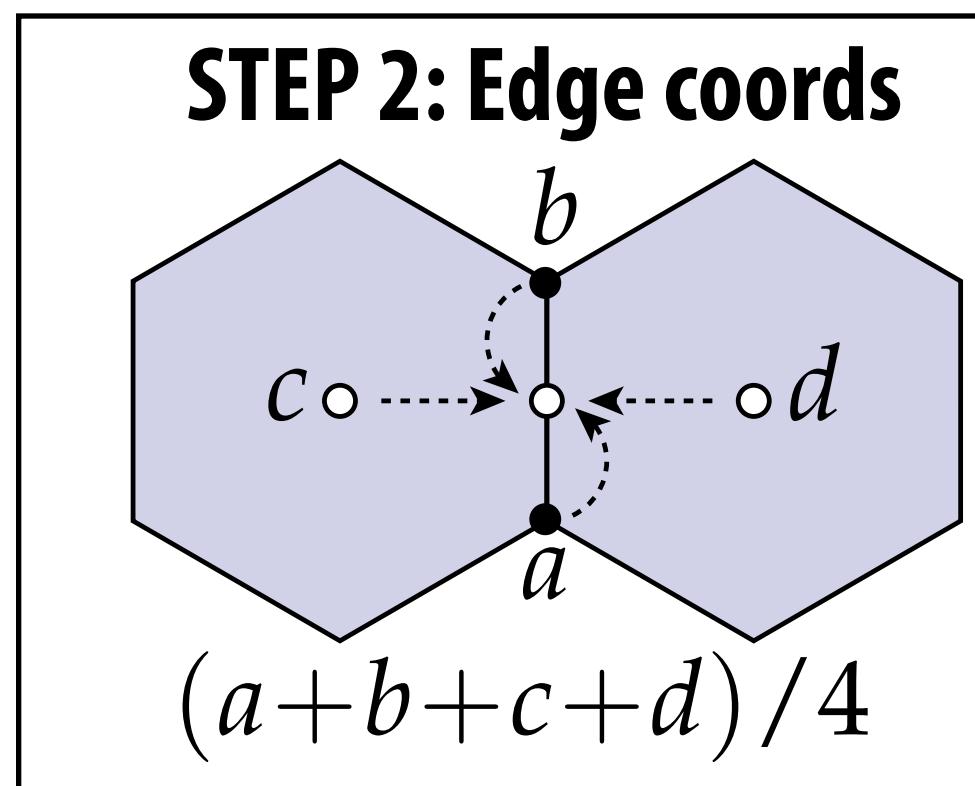
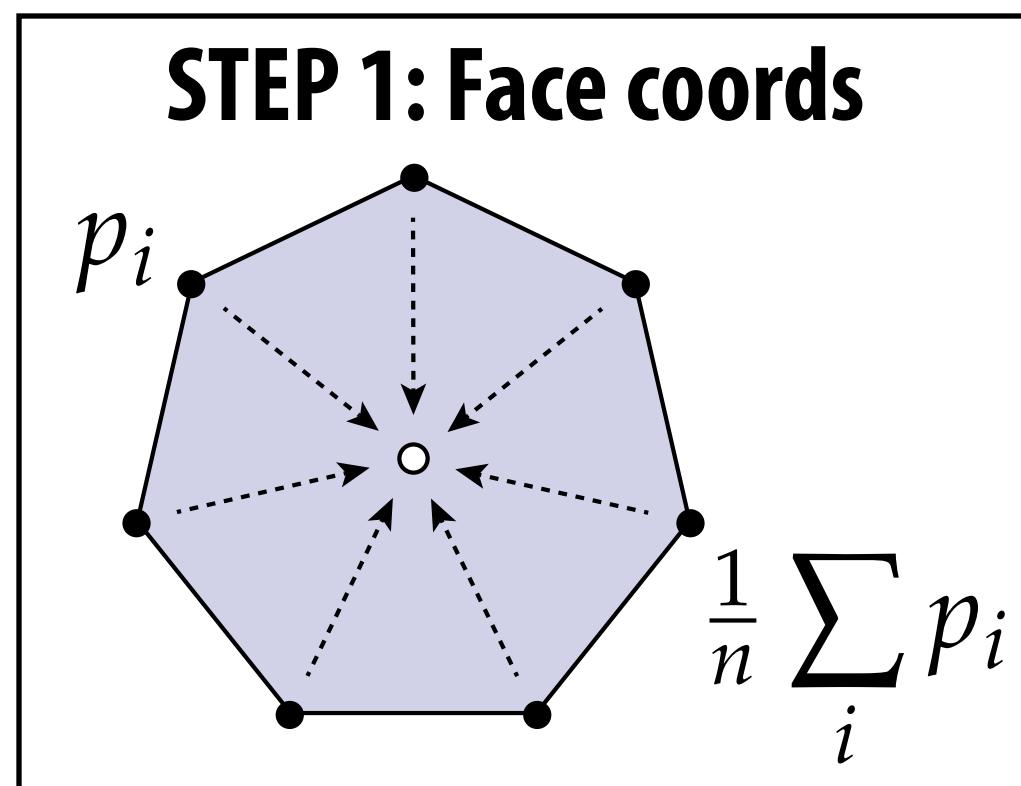
- Triangle: Loop, Butterfly, Sqrt(3)

# Catmull-Clark Subdivision

- Step 0: split every polygon (any # of sides) into quadrilaterals:



- New vertex positions are weighted combination of old ones:



New vertex coords:

$$\frac{Q + 2R + (n - 3)S}{n}$$

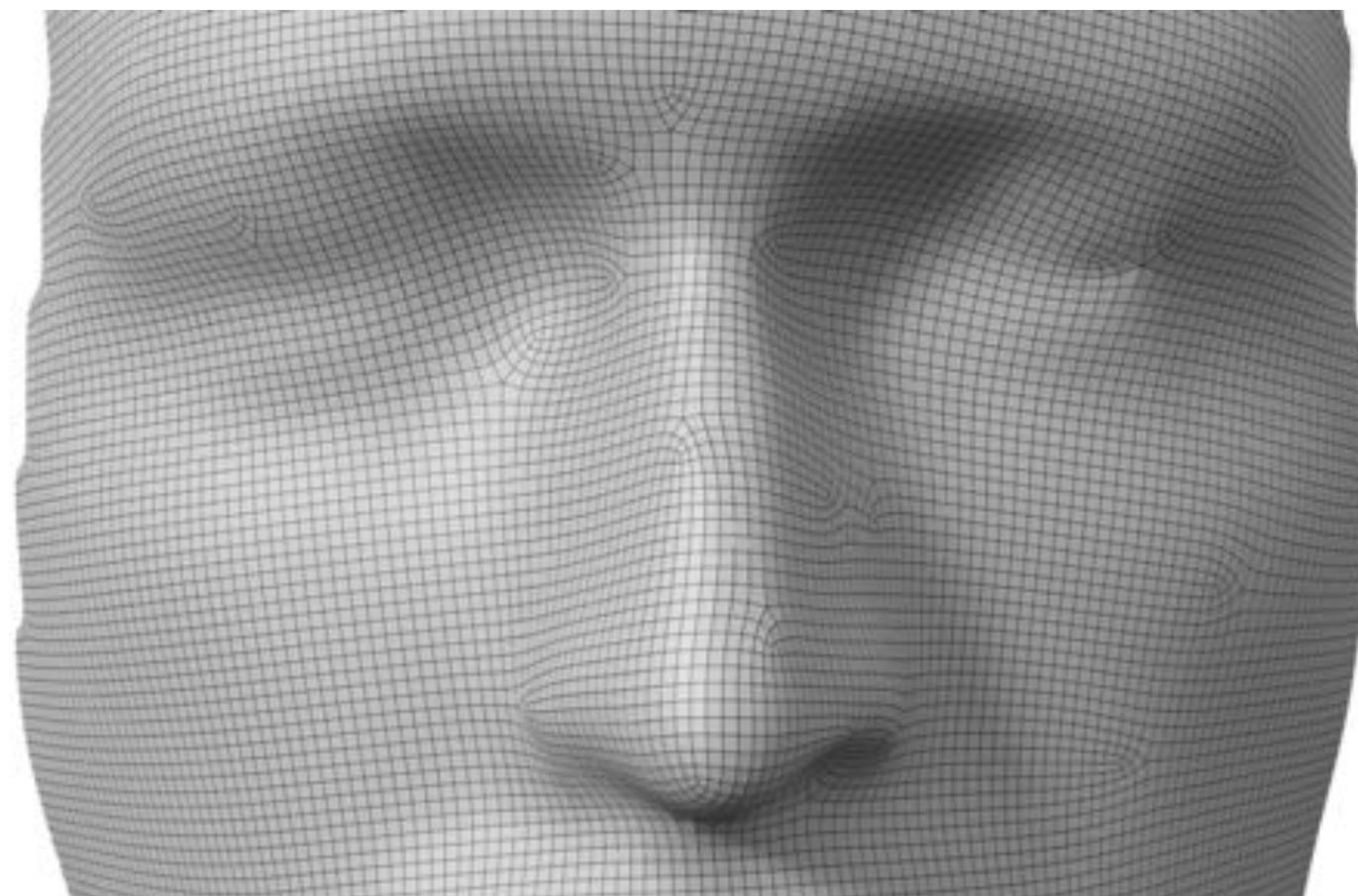
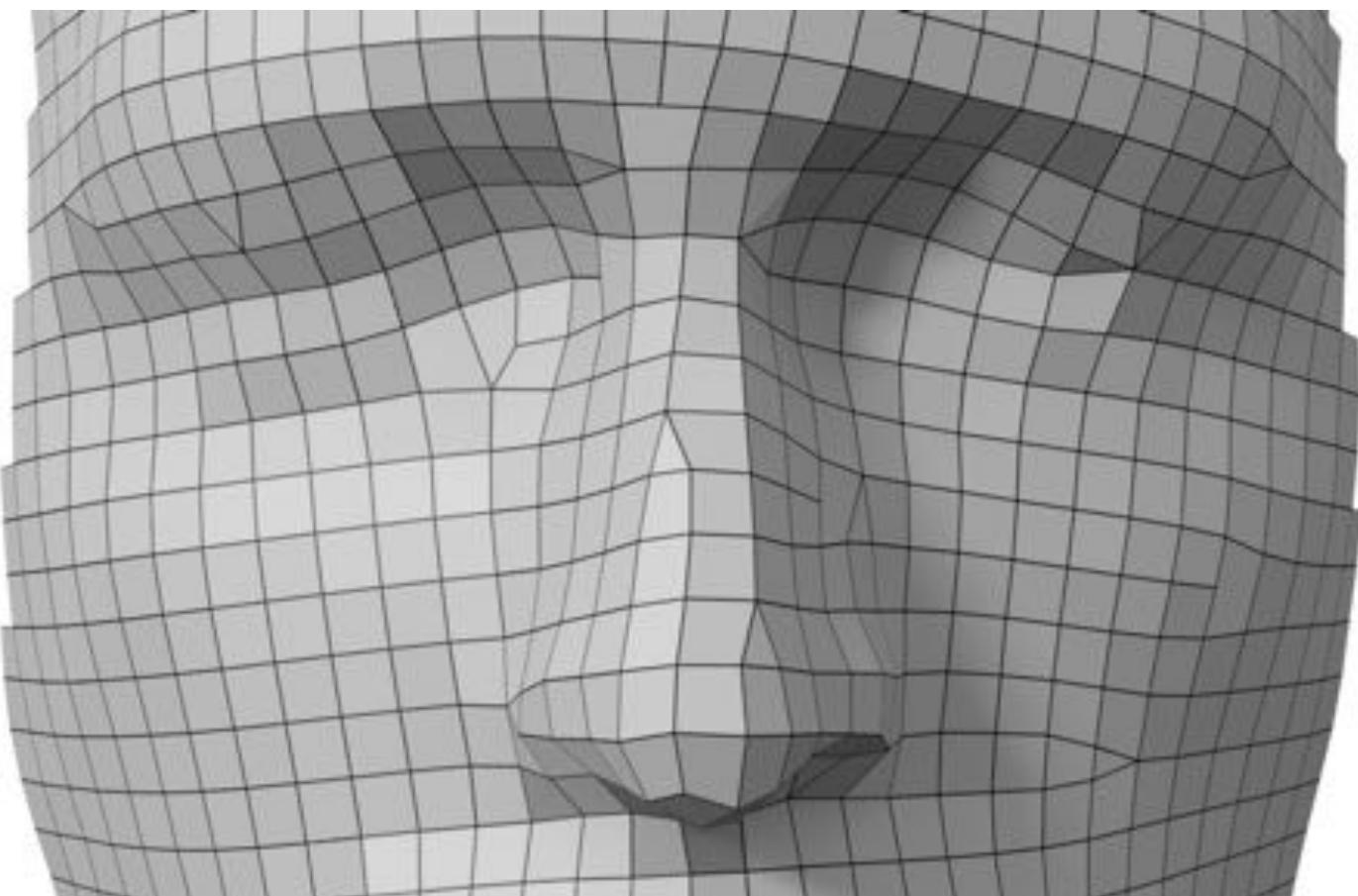
$n$  – vertex degree

$Q$  – average of face coords around vertex

$R$  – average of edge coords around vertex

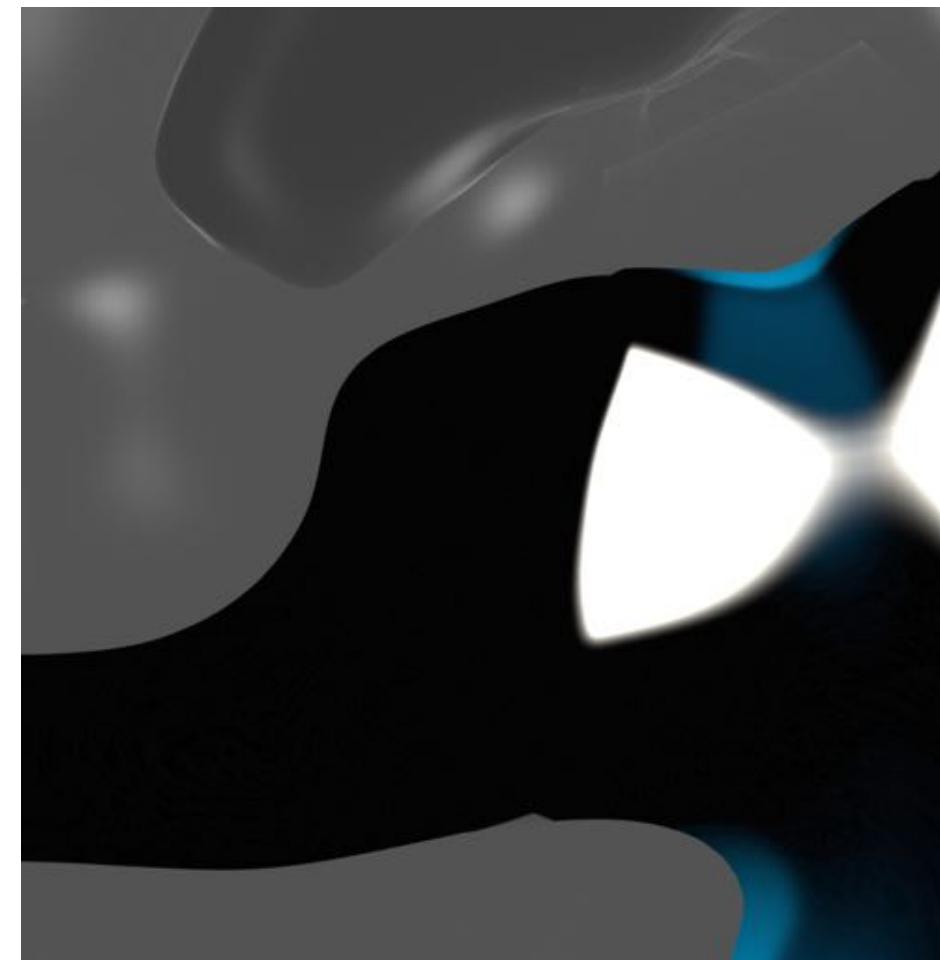
$S$  – original vertex position

# Catmull-Clark on quad mesh

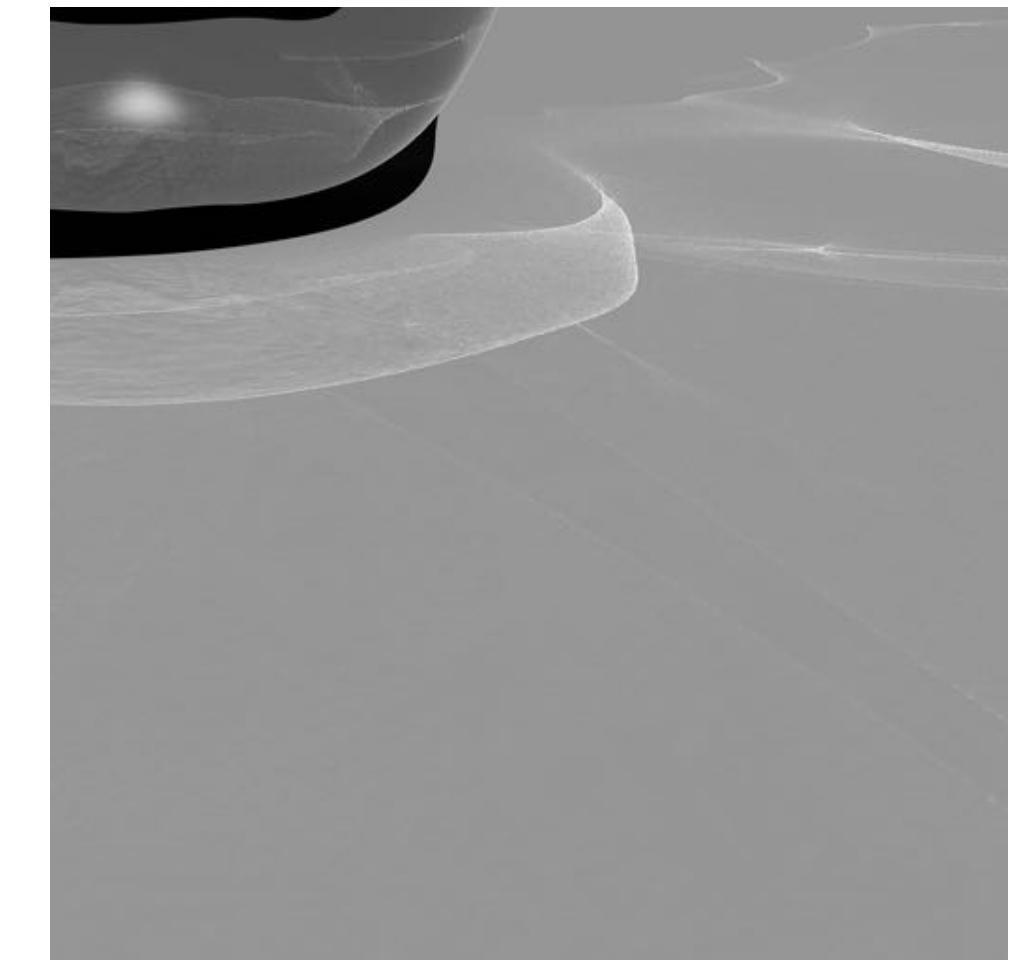


**few irregular vertices**

$\Rightarrow$  **smoothly-varying surface normals**

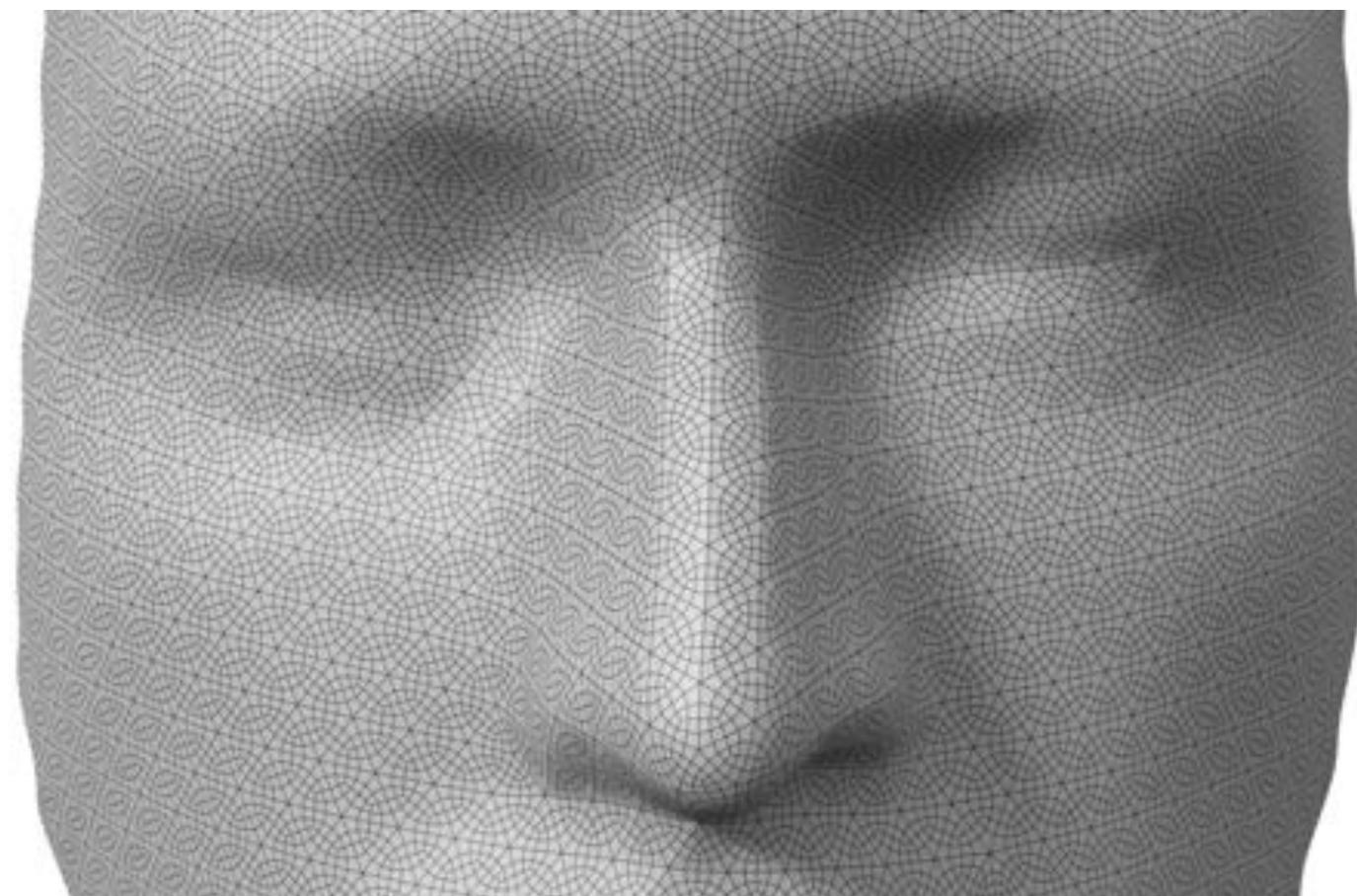
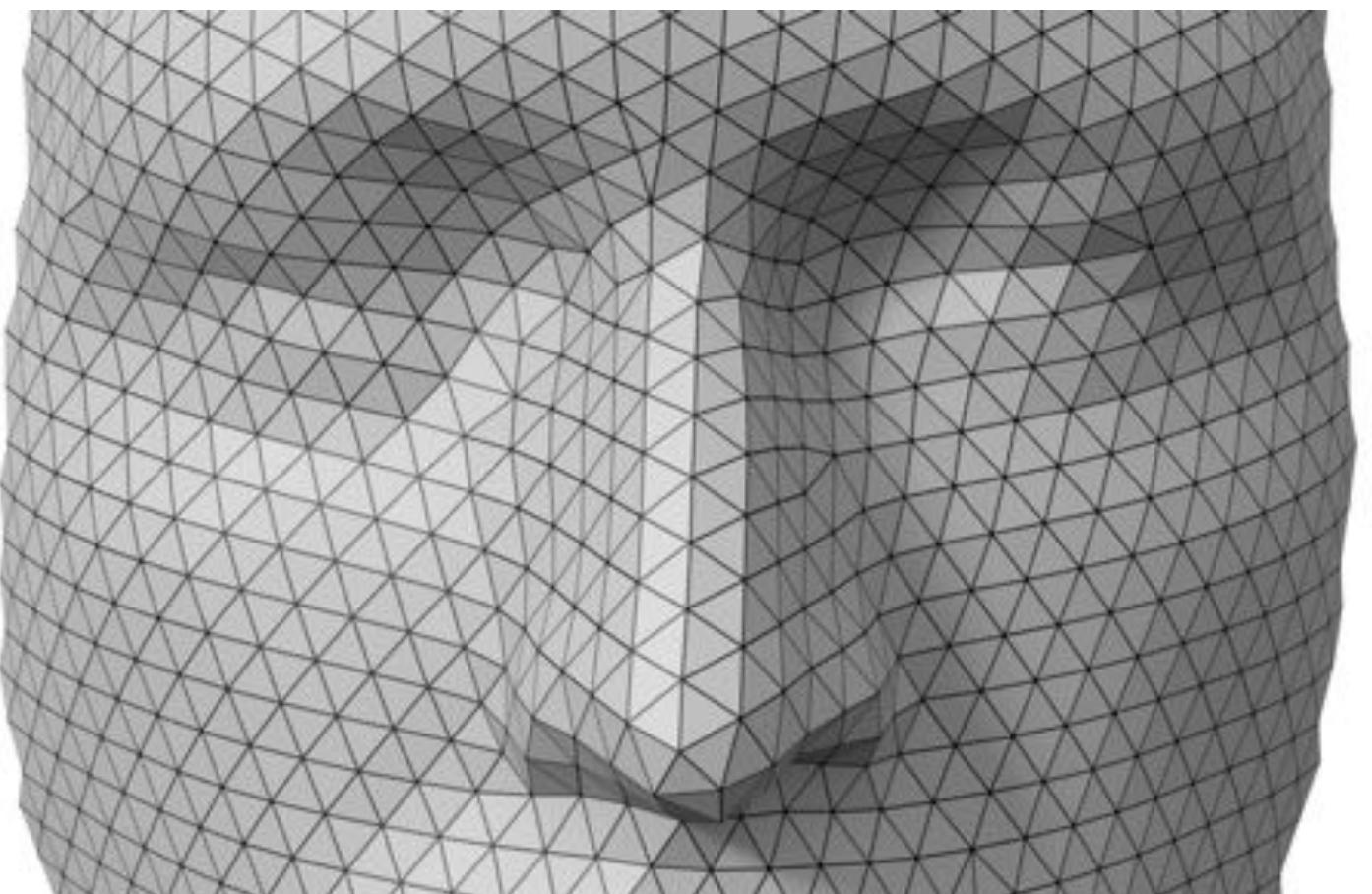


**smooth  
reflection lines**

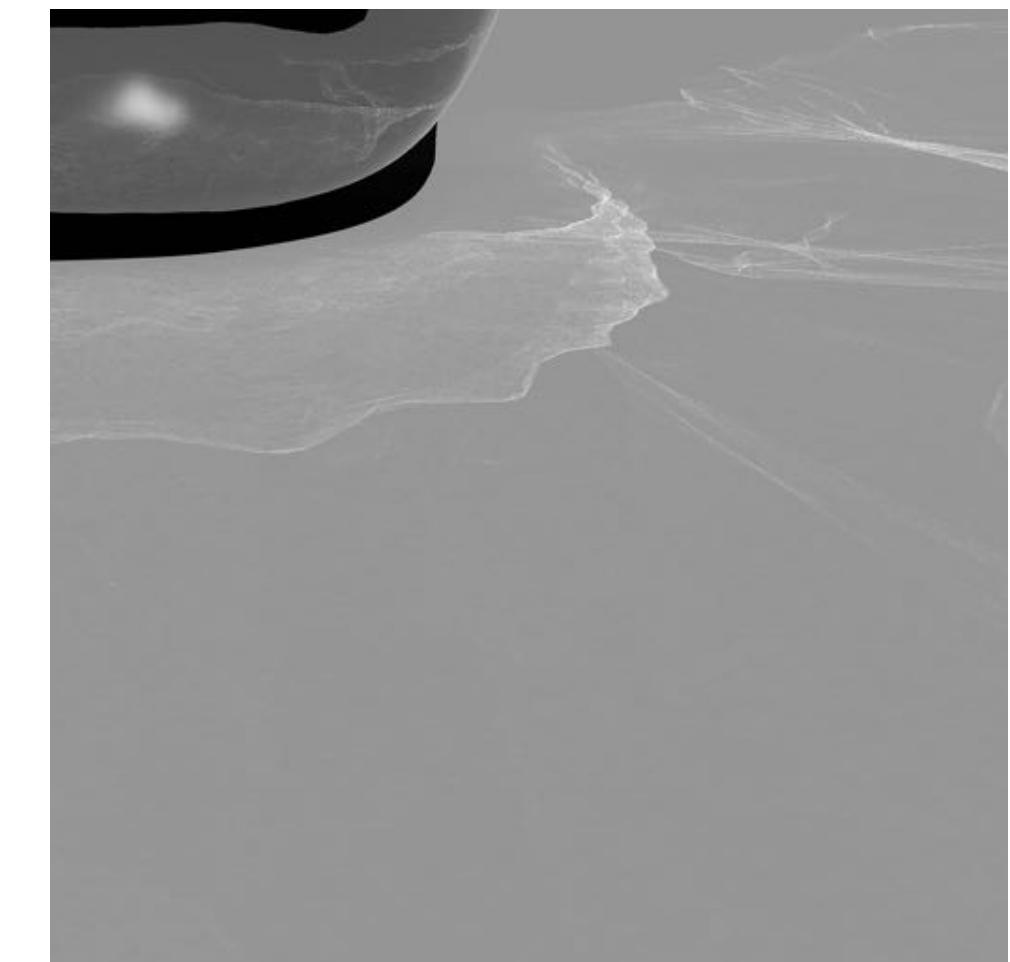
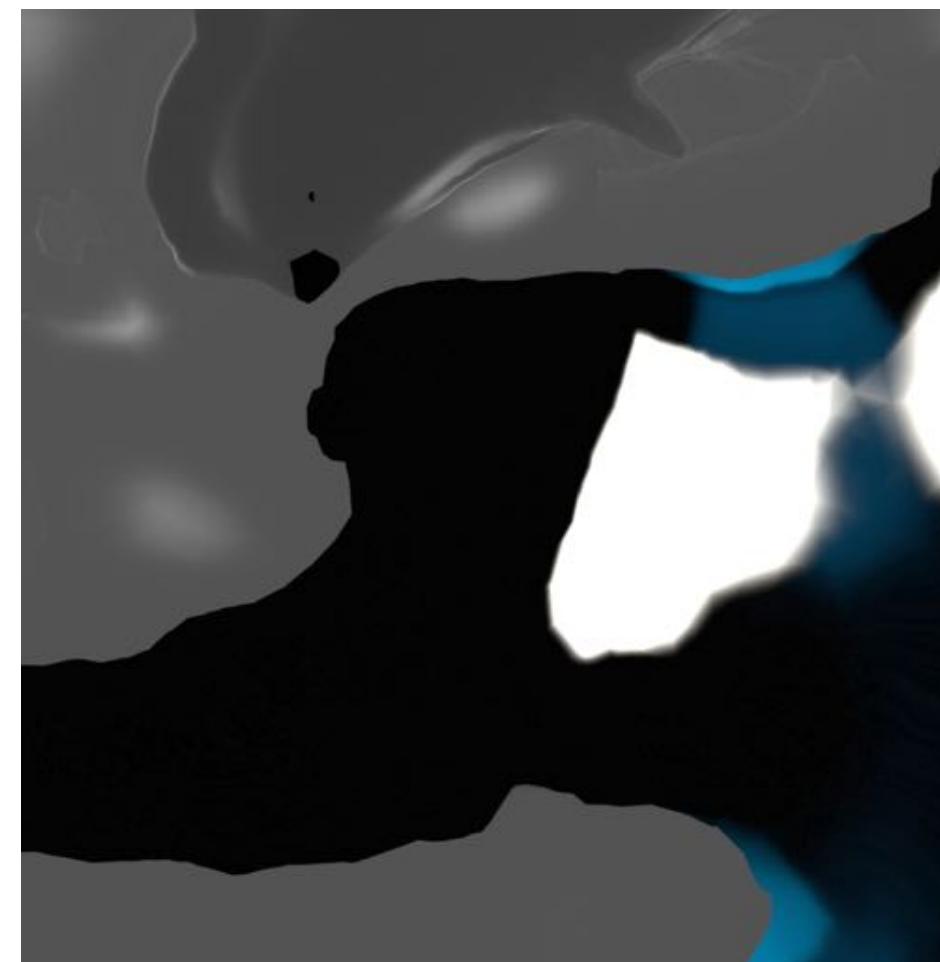


**smooth  
caustics**

# Catmull-Clark on triangle mesh



**many irregular vertices**  
⇒ erratic surface normals

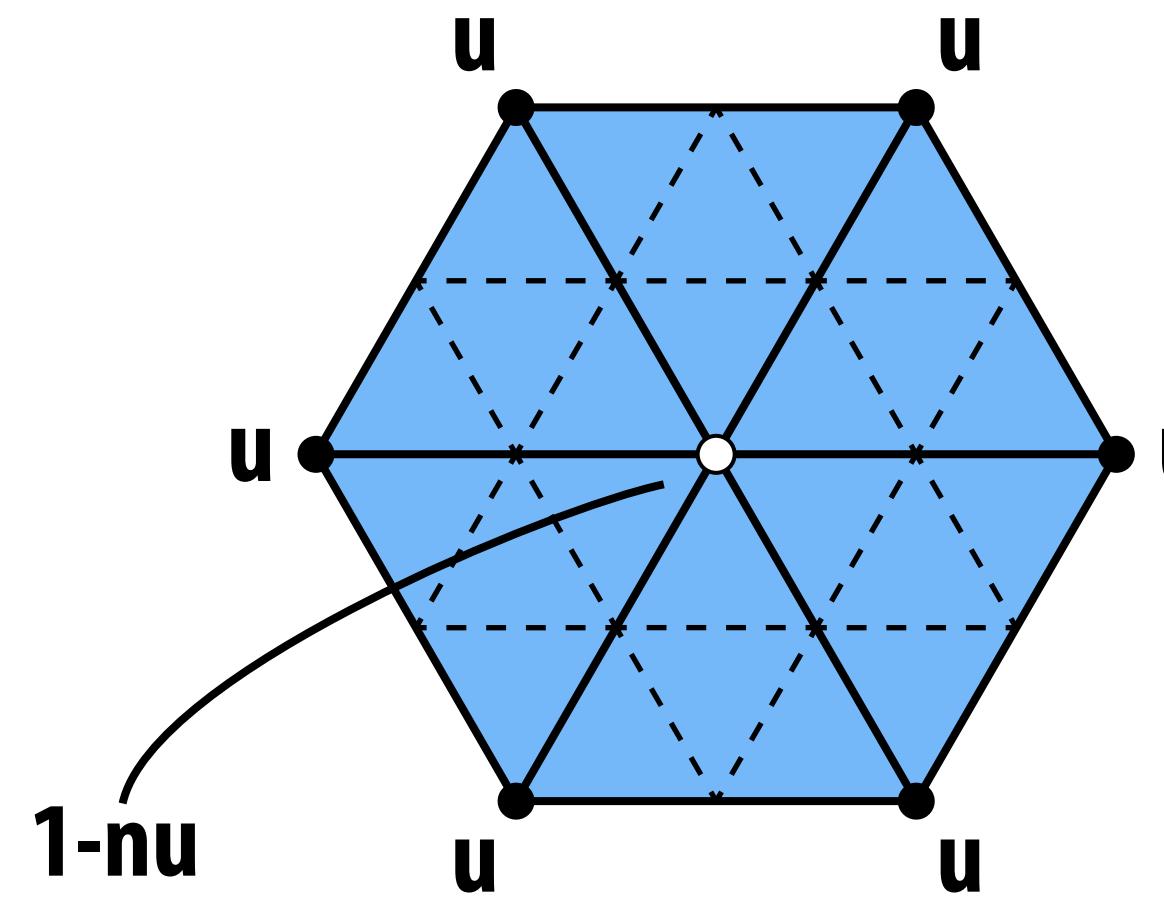
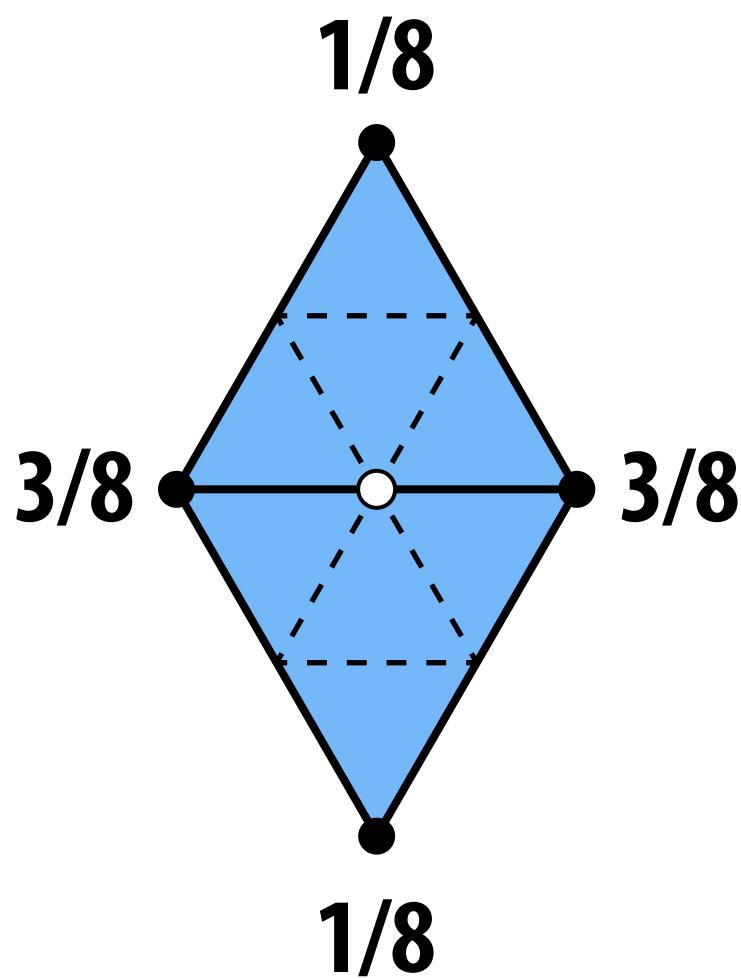
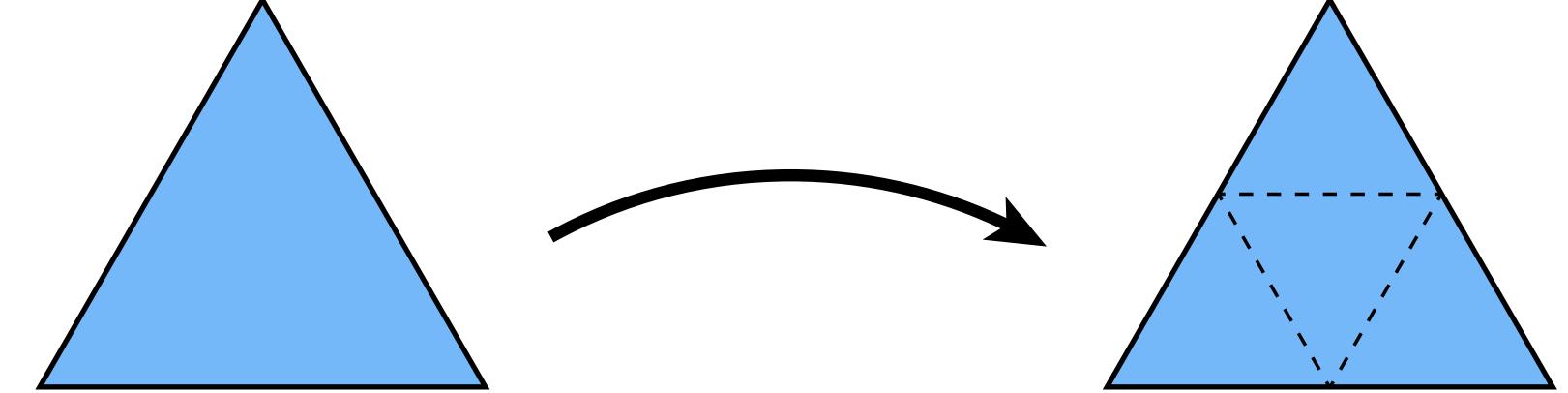


jagged  
reflection lines

jagged  
caustics

# Loop Subdivision

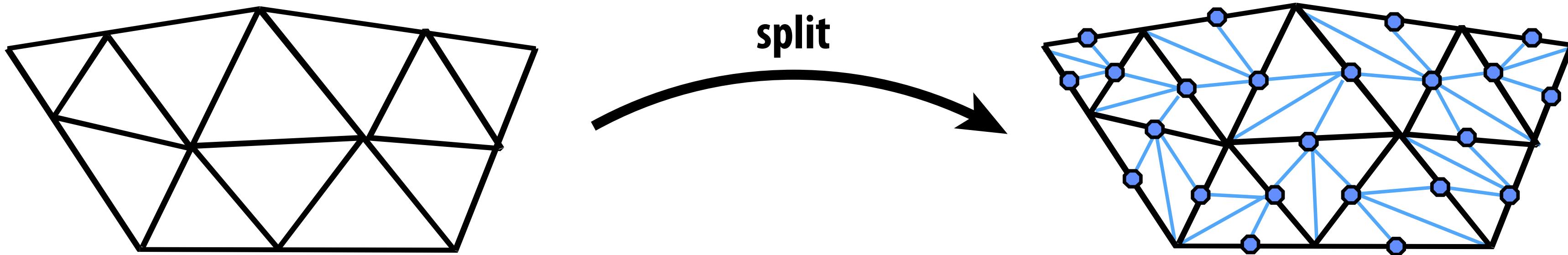
- Alternative subdivision scheme for triangle meshes
- Curvature is continuous away from irregular vertices ("C<sup>2</sup>")
- Algorithm:
  - Split each triangle into four
  - Assign new vertex positions according to weights:



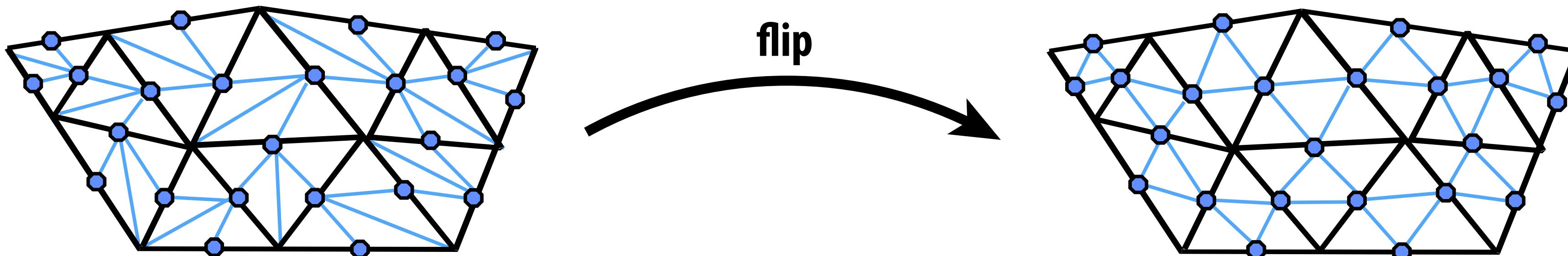
n: vertex degree  
u:  $3/16$  if  $n=3$ ,  $3/(8n)$  otherwise

# Loop Subdivision via Edge Operations

- First, split edges of original mesh in any order:



- Next, flip new edges that touch a new & old vertex:



**(Don't forget to update vertex positions!)**

**What if we want fewer triangles?**

# Simplification via Edge Collapse

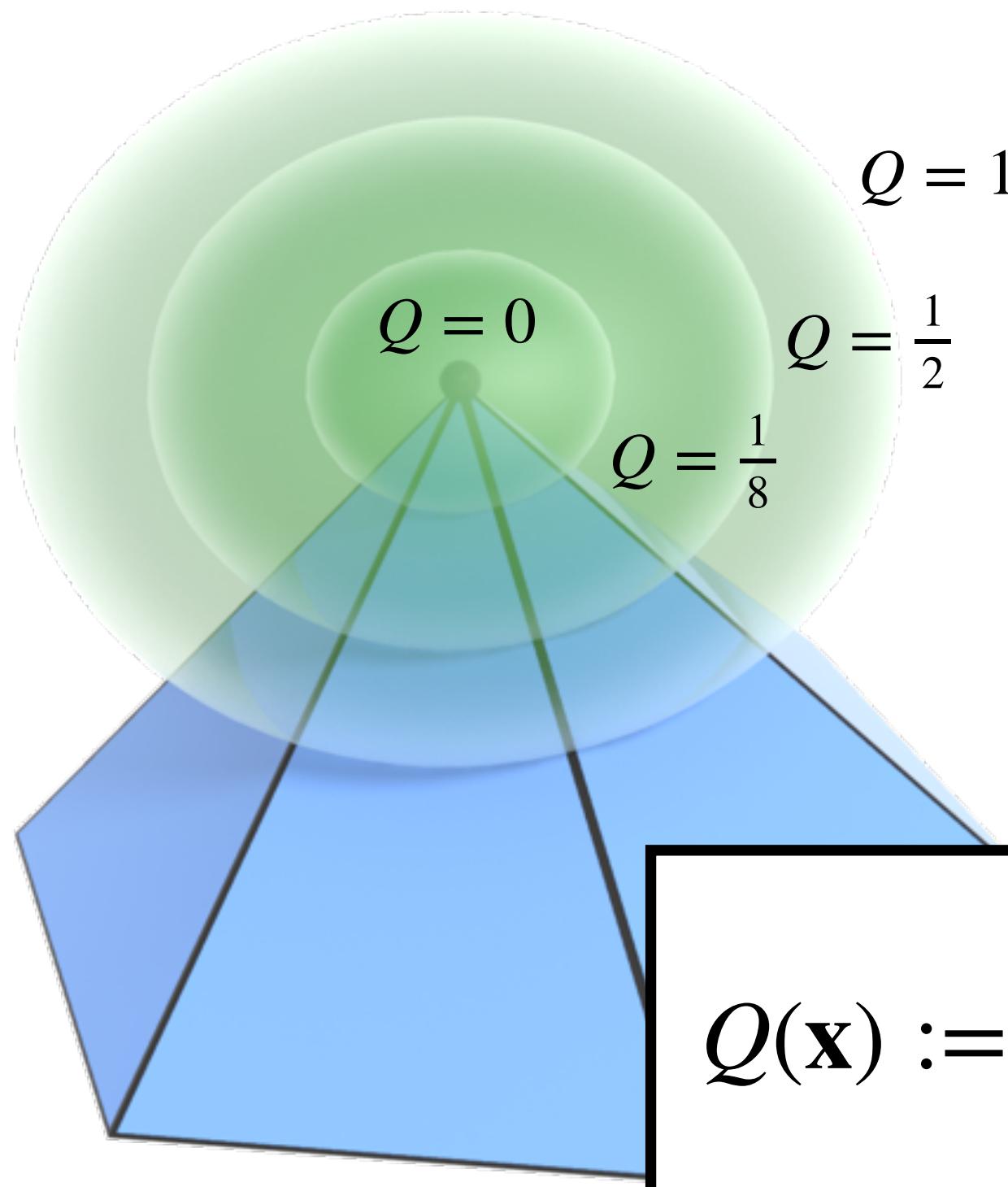
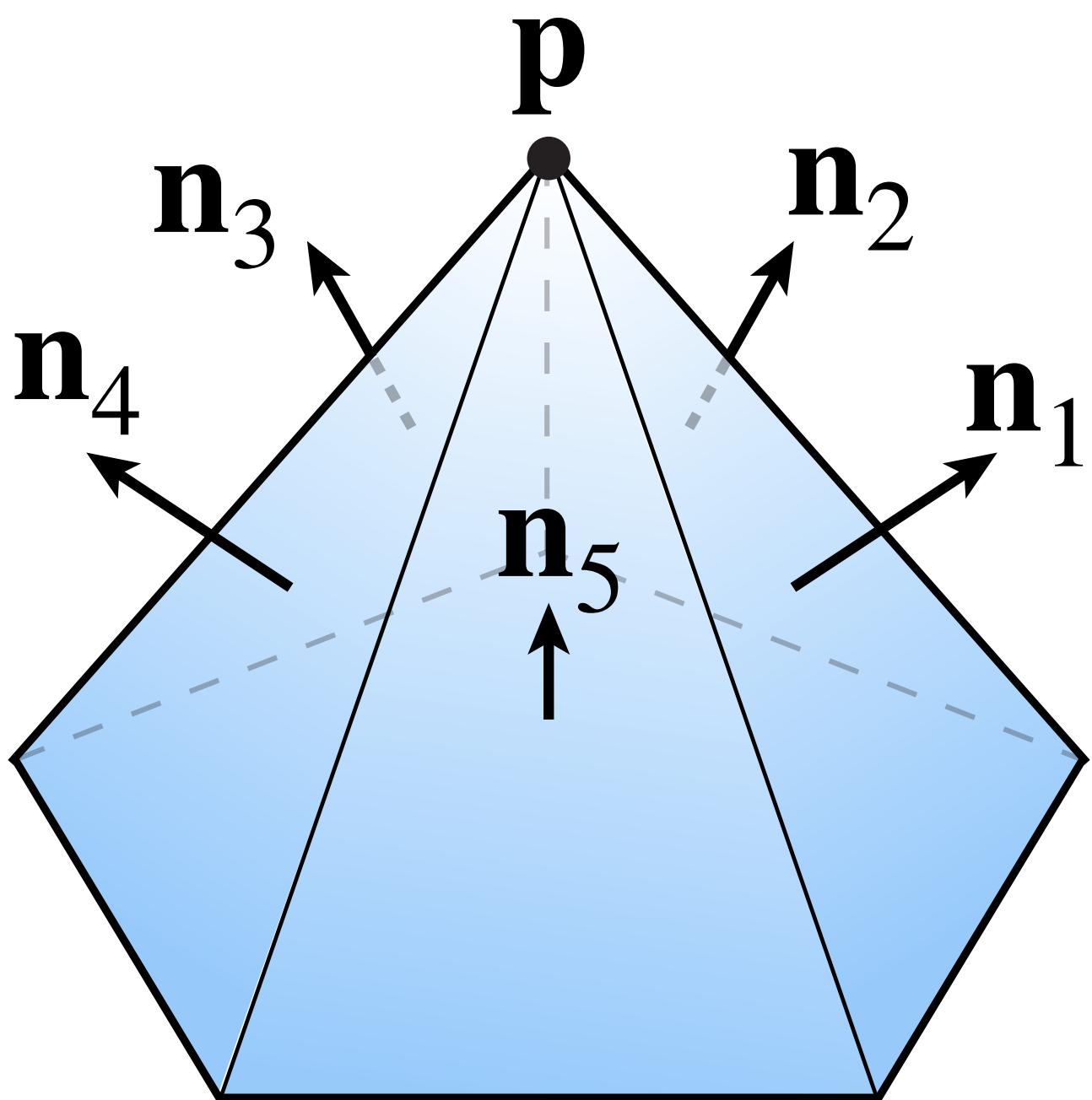
- One popular scheme: iteratively collapse edges
- Greedy algorithm:
  - assign each edge a cost
  - collapse edge with least cost
  - repeat until target number of elements is reached
- Particularly effective cost function: quadric error metric\*



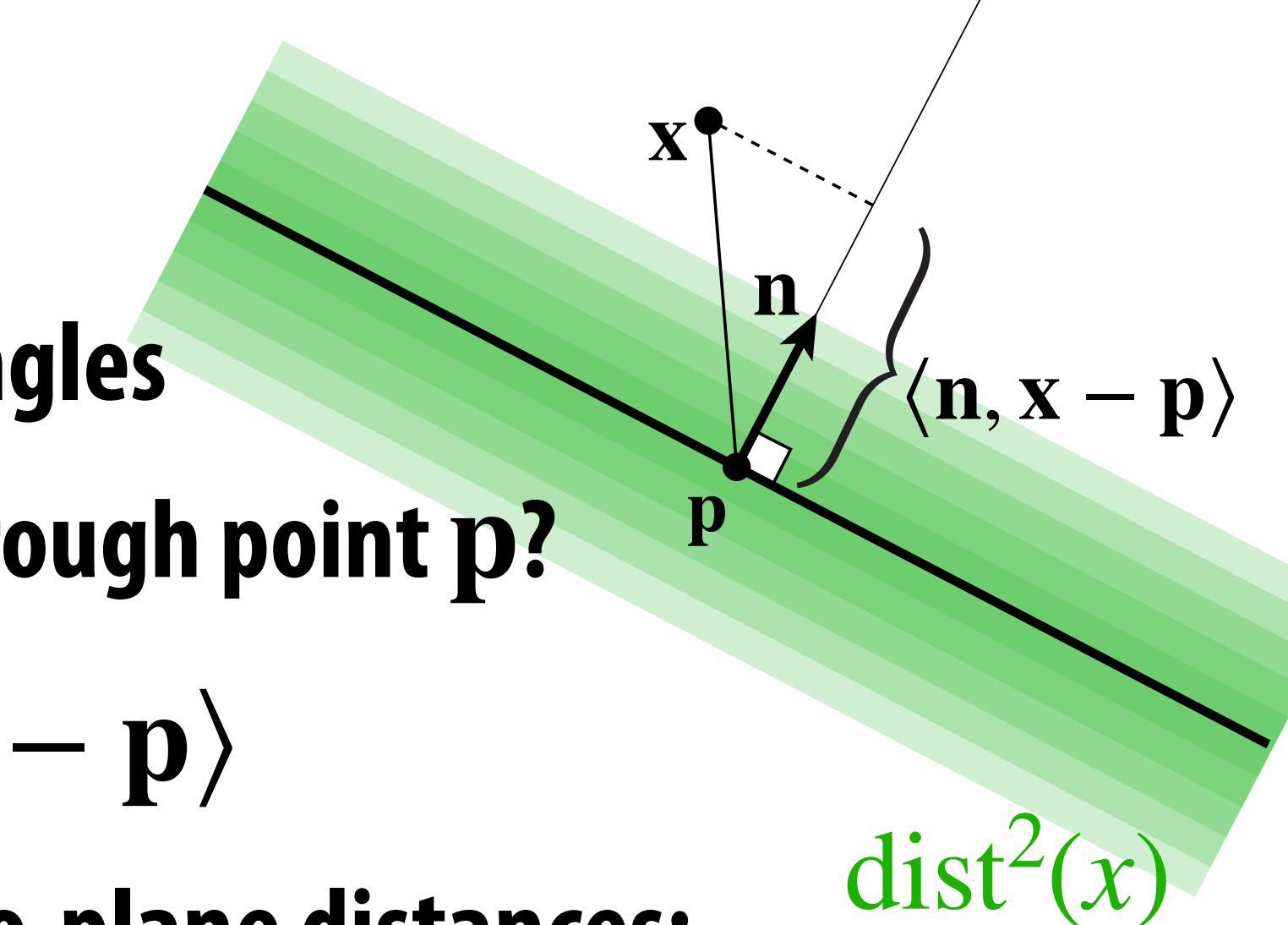
\*invented at CMU (Garland & Heckbert 1997)

# Quadric Error Metric

- Approximate distance to a collection of triangles
- Q: Distance to plane w/ normal  $\mathbf{n}$  passing through point  $p$ ?
- A:  $\text{dist}(\mathbf{x}) = \langle \mathbf{n}, \mathbf{x} \rangle - \langle \mathbf{n}, \mathbf{p} \rangle = \langle \mathbf{n}, \mathbf{x} - \mathbf{p} \rangle$
- Quadric error is then sum of squared point-to-plane distances:



$$Q(\mathbf{x}) := \sum_{i=1}^k \langle \mathbf{n}_i, \mathbf{x} - \mathbf{p} \rangle^2$$



# Quadric Error - Homogeneous Coordinates

- Suppose in coordinates we have

- a query point  $\mathbf{x} = (x, y, z)$
- a normal  $\mathbf{n} = (a, b, c)$
- an offset  $d := -\langle \mathbf{n}, \mathbf{p} \rangle$

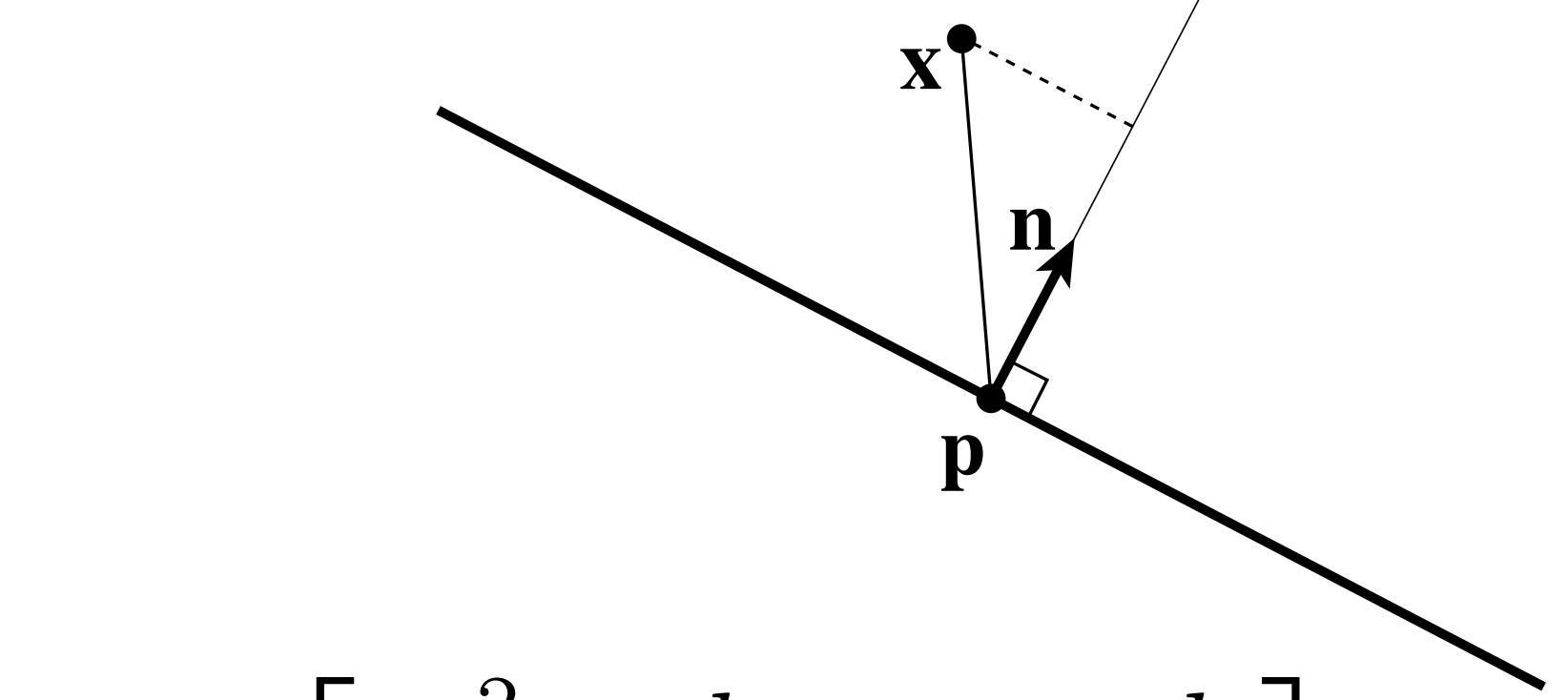
- In homogeneous coordinates, let

- $\mathbf{u} := (x, y, z, 1)$
- $\mathbf{v} := (a, b, c, d)$

- Signed distance to plane is then just  $\langle \mathbf{u}, \mathbf{v} \rangle = ax + by + cz + d$
- Squared distance is  $\langle \mathbf{u}, \mathbf{v} \rangle^2 = \mathbf{u}^\top (\mathbf{v} \mathbf{v}^\top) \mathbf{u} =: \mathbf{u}^\top K \mathbf{u}$
- Matrix  $K = \mathbf{v} \mathbf{v}^\top$  encodes squared distance to plane

Key idea: sum of matrices  $K$   $\Leftrightarrow$  distance to union of planes

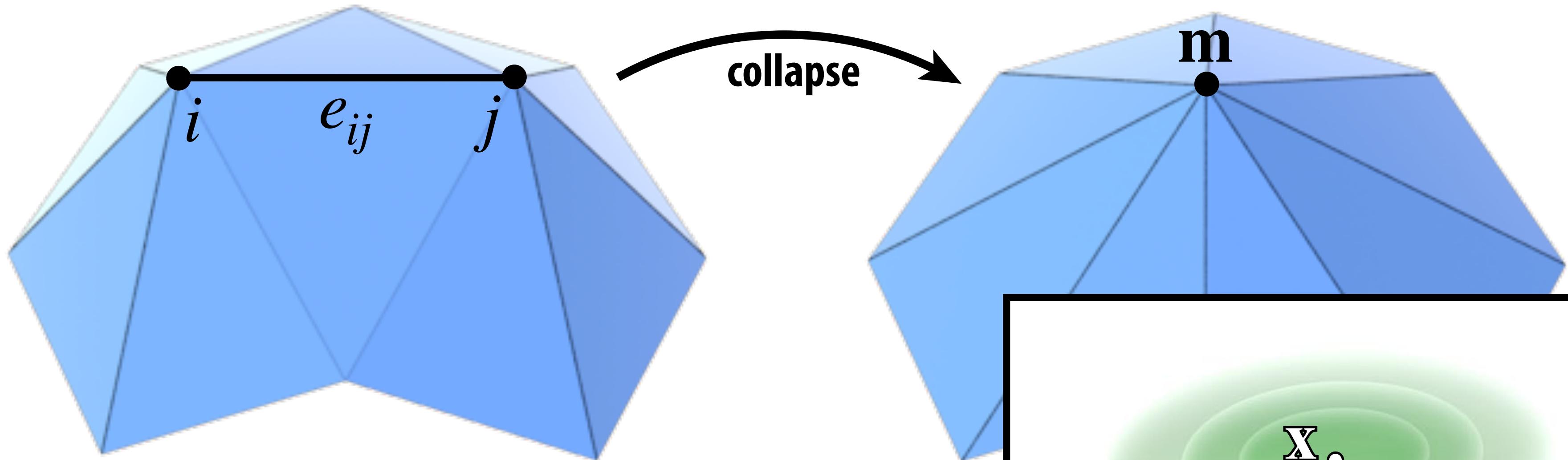
$$\mathbf{u}^\top K_1 \mathbf{u} + \mathbf{u}^\top K_2 \mathbf{u} = \mathbf{u}^\top (K_1 + K_2) \mathbf{u}$$



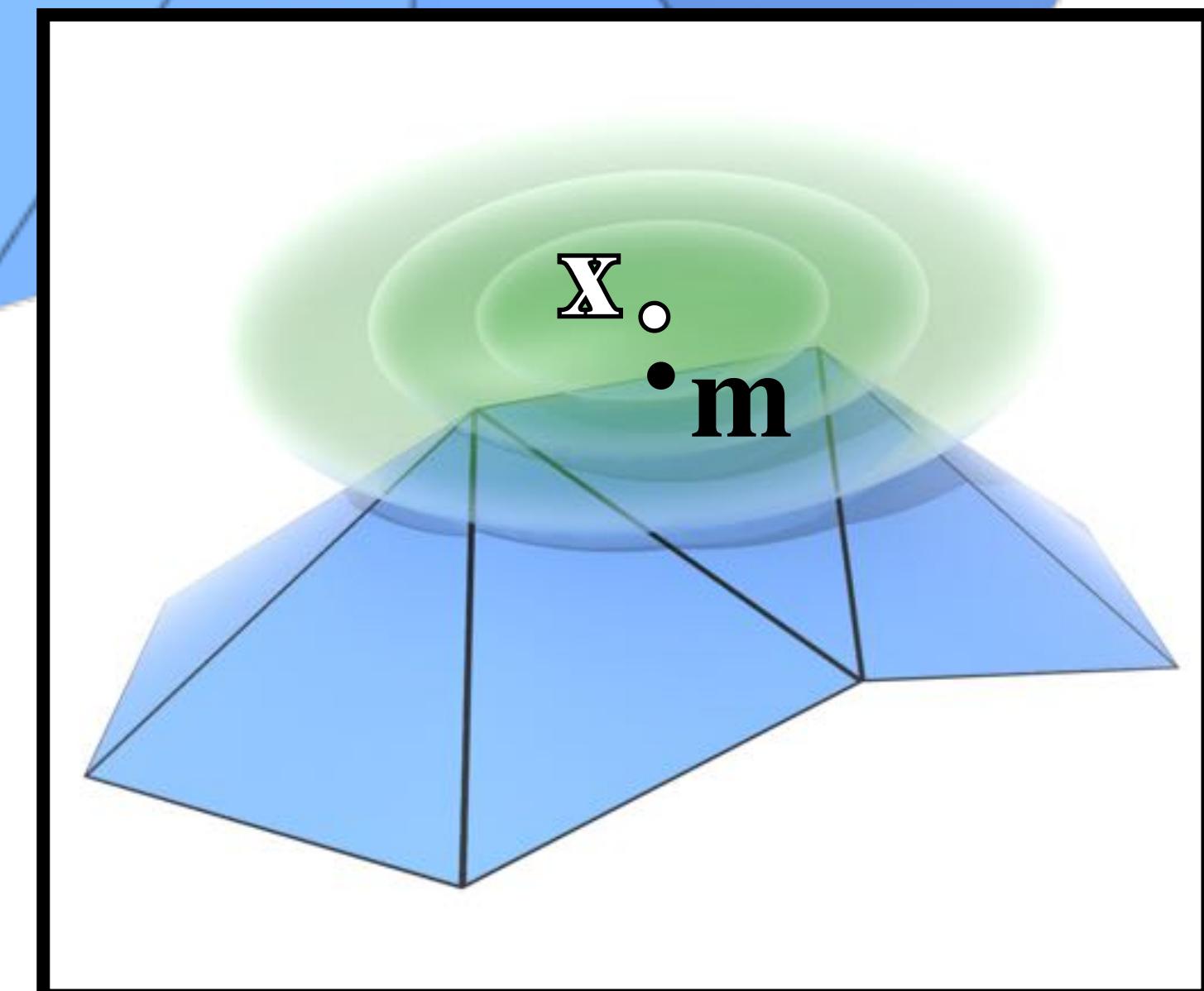
$$K = \begin{bmatrix} a^2 & ab & ac & ad \\ ab & b^2 & bc & bd \\ ac & bc & c^2 & cd \\ ad & bd & cd & d^2 \end{bmatrix}$$

# Quadric Error of Edge Collapse

- How much does it cost to collapse an edge  $e_{ij}$ ?
- Idea: compute midpoint  $m$ , measure error  $Q(m) = m^T(K_i + K_j)m$
- Error becomes “score” for  $e_{ij}$ , determining priority



- Better idea: find point  $x$  that minimizes error!
- Ok, but how do we minimize quadric error?



# Review: Minimizing a Quadratic Function

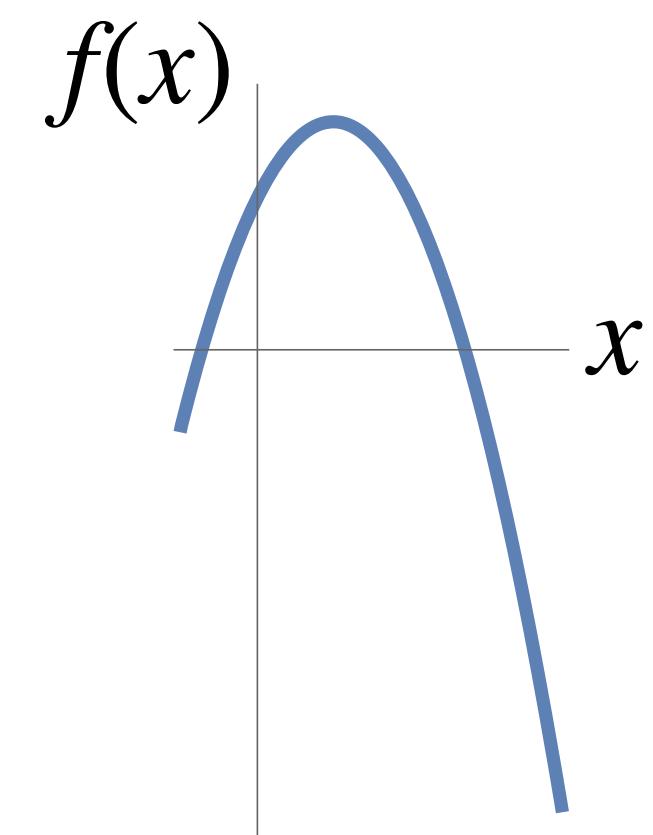
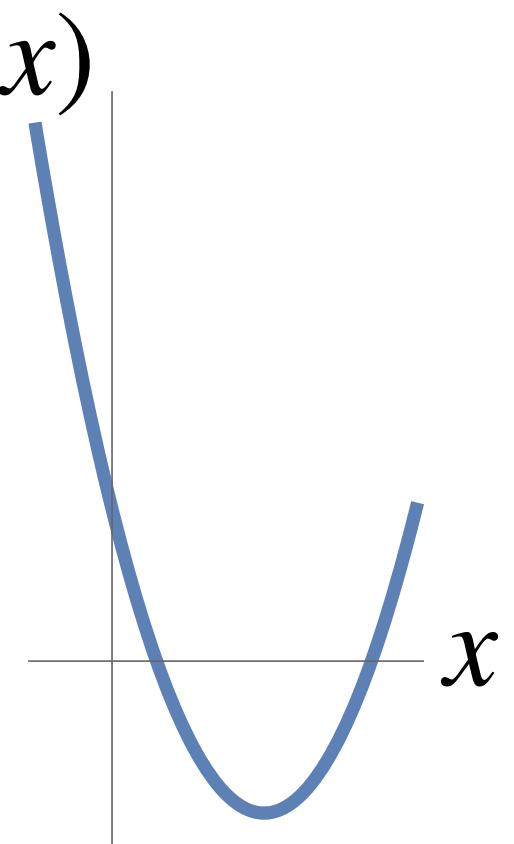
- Suppose you have a function  $f(x) = ax^2 + bx + c$
- Q: What does the graph of this function look like?
- Could also look like this!
- Q: How do we find the minimum?
- A: Find where the function looks “flat” if we zoom in really close
- I.e., find point  $x$  where 1st derivative vanishes:

$$f'(x) = 0$$

$$2ax + b = 0$$

$$x = -b/2a$$

(What does  $x$  describe for the second function?)



# Minimizing Quadratic Polynomial

- Not much harder to minimize a quadratic polynomial in  $n$  variables
- Can always write in terms of a symmetric matrix  $A$
- E.g., in 2D:  $f(x, y) = ax^2 + bxy + cy^2 + dx + ey + g$

$$\mathbf{x} = \begin{bmatrix} x \\ y \end{bmatrix} \quad A = \begin{bmatrix} a & b/2 \\ b/2 & c \end{bmatrix} \quad \mathbf{u} = \begin{bmatrix} d \\ e \end{bmatrix}$$

$$f(x, y) = \mathbf{x}^\top A \mathbf{x} + \mathbf{u}^\top \mathbf{x} + g$$

(will have this same form for any  $n$ )

- **Q: How do we find a critical point (min/max/saddle)?**
- **A: Set derivative to zero!**

$$2A\mathbf{x} + \mathbf{u} = 0$$

$$\mathbf{x} = -\frac{1}{2}A^{-1}\mathbf{u}$$

(compare with  
our 1D solution)

$$x = -b/2a$$

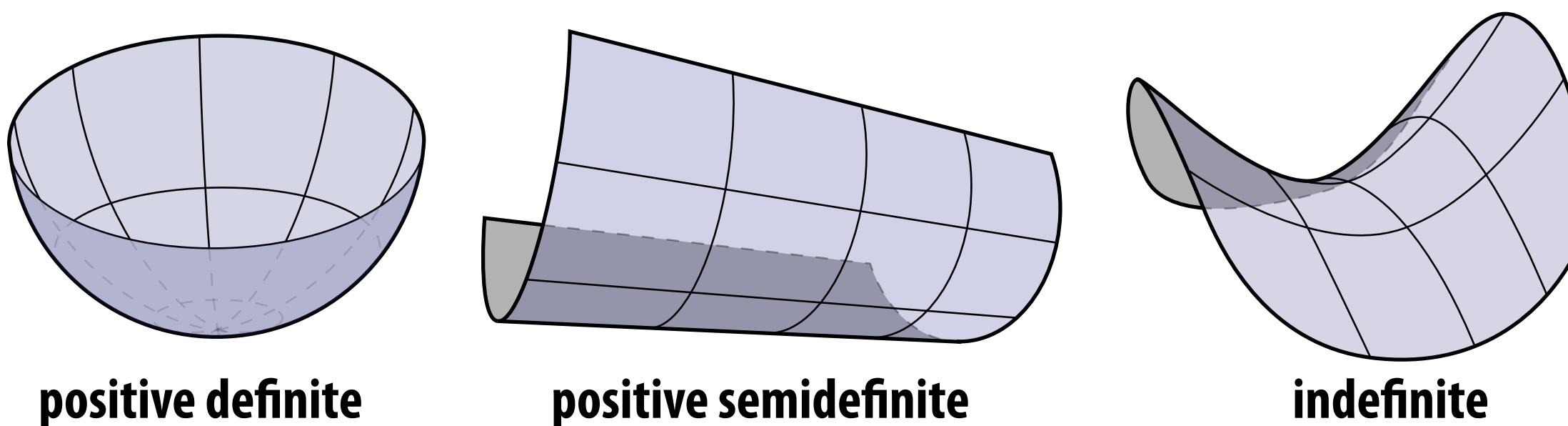
(Can you show this is true, at least in 2D?)

# Positive Definite Quadratic Form

- Just like our 1D parabola, critical point is not always a min!
- Q: In 2D, 3D, nD, when do we get a minimum?
- A: When matrix A is positive-definite:

$$\mathbf{x}^\top A \mathbf{x} > 0 \quad \forall \mathbf{x}$$

- 1D: Must have  $xax = ax^2 > 0$ . In other words:  $a$  is positive!
- 2D: Graph of function looks like a “bowl”:



Positive-definiteness **extremely important** in computer graphics:  
means we can find minimizers by solving linear equations. Starting  
point for many algorithms (geometry processing, simulation, ...)

# Minimizing Quadric Error

- Find “best” point for edge collapse by minimizing quadratic form

$$\min_{\mathbf{u} \in \mathbb{R}^4} \mathbf{u}^T K \mathbf{u}$$

- Already know fourth (homogeneous) coordinate for a point is 1
- So, break up our quadratic function into two pieces:

$$\begin{bmatrix} \mathbf{x}^\top & 1 \end{bmatrix} \begin{bmatrix} B & \mathbf{w} \\ \mathbf{w}^\top & d^2 \end{bmatrix} \begin{bmatrix} \mathbf{x} \\ 1 \end{bmatrix}$$
$$= \mathbf{x}^\top B \mathbf{x} + 2\mathbf{w}^\top \mathbf{x} + d^2$$

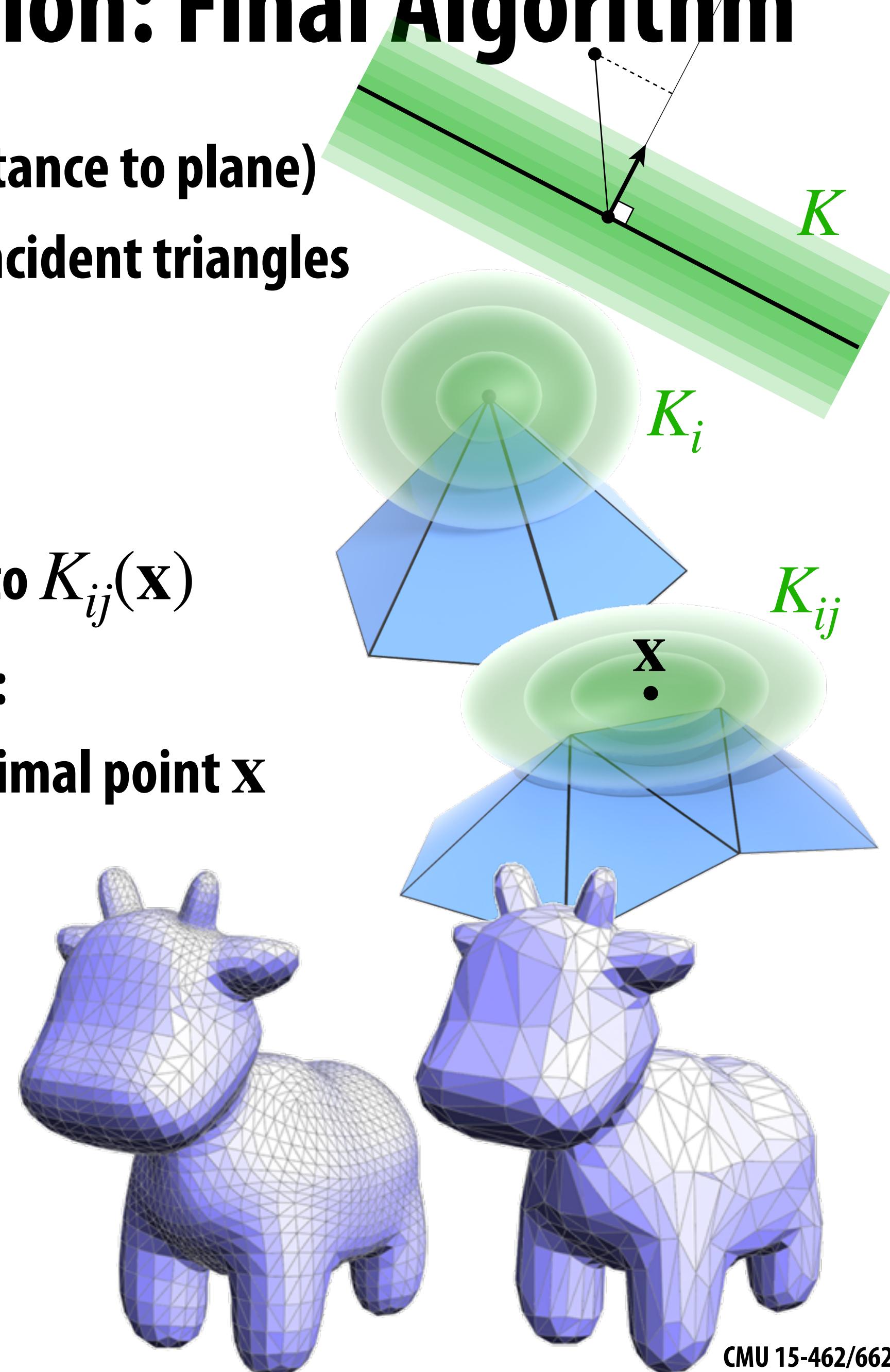
- Now we have a quadratic polynomial in the unknown position  $\mathbf{x} \in \mathbb{R}^3$
- Can minimize as before:

$$2B\mathbf{x} + 2\mathbf{w} = 0 \qquad \iff \qquad \mathbf{x} = -B^{-1}\mathbf{w}$$

**Q: Why should  $B$  be positive-definite?**

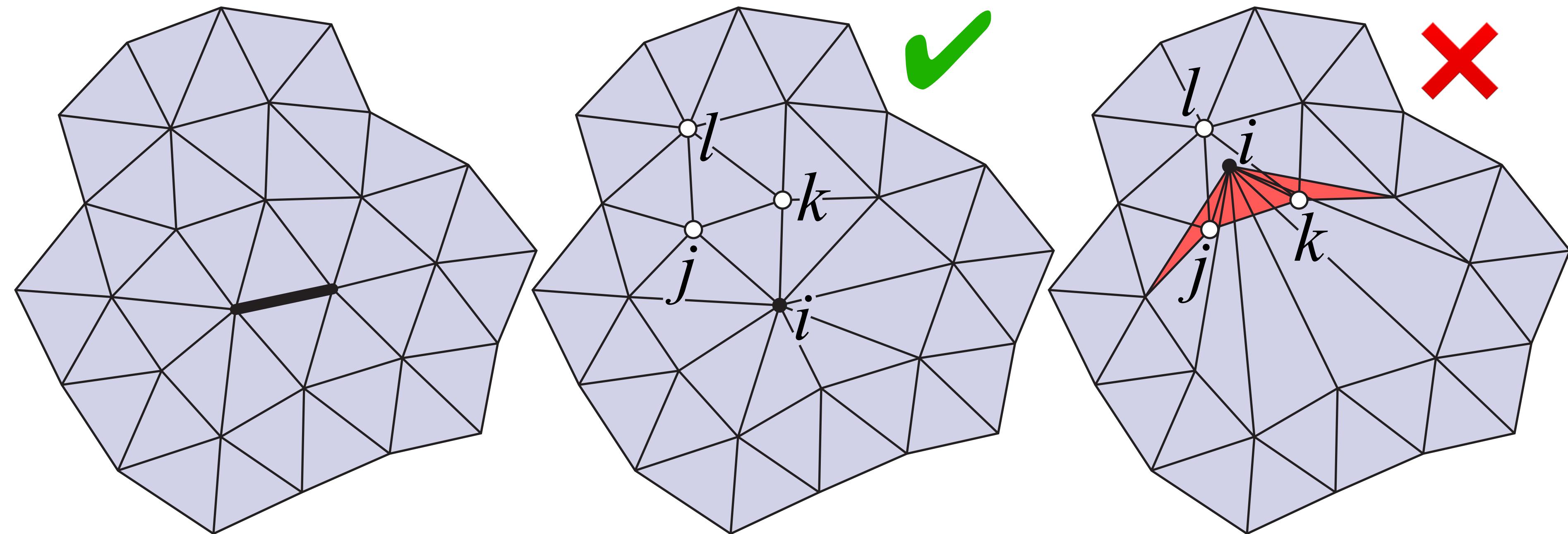
# Quadric Error Simplification: Final Algorithm

- Compute  $K$  for each triangle (squared distance to plane)
- Set  $K_i$  at each vertex to sum of  $K$ s from incident triangles
- For each edge  $e_{ij}$ :
  - set  $K_{ij} = K_i + K_j$
  - find point  $\mathbf{x}$  minimizing error, set cost to  $K_{ij}(\mathbf{x})$
- Until we reach target number of triangles:
  - collapse edge  $e_{ij}$  with smallest cost to optimal point  $\mathbf{x}$
  - set quadric at new vertex to  $K_{ij}$
  - update cost of edges touching new vertex
- More details in assignment writeup!



# Quadric Simplification—Flipped Triangles

- Depending on where we put the new vertex, one of the new triangles might be “flipped” (normal points in instead of out):

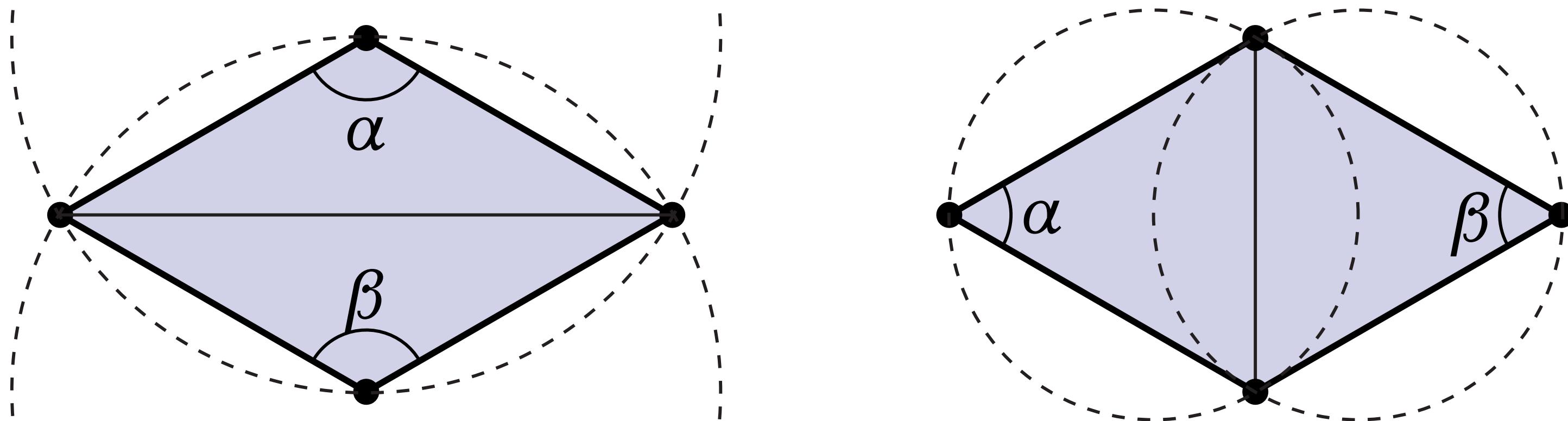


- Easy solution: for each triangle  $ijk$  touching collapsed vertex  $i$ , consider normals  $N_{ijk}$  and  $N_{kjl}$  (where  $kjl$  is other triangle containing edge  $jk$ )
- If  $\langle N_{ijk}, N_{kjl} \rangle$  is negative, don't collapse this edge!

**What if we're happy with the number of triangles, but want to improve quality?**

# How do we make a mesh “more Delaunay”?

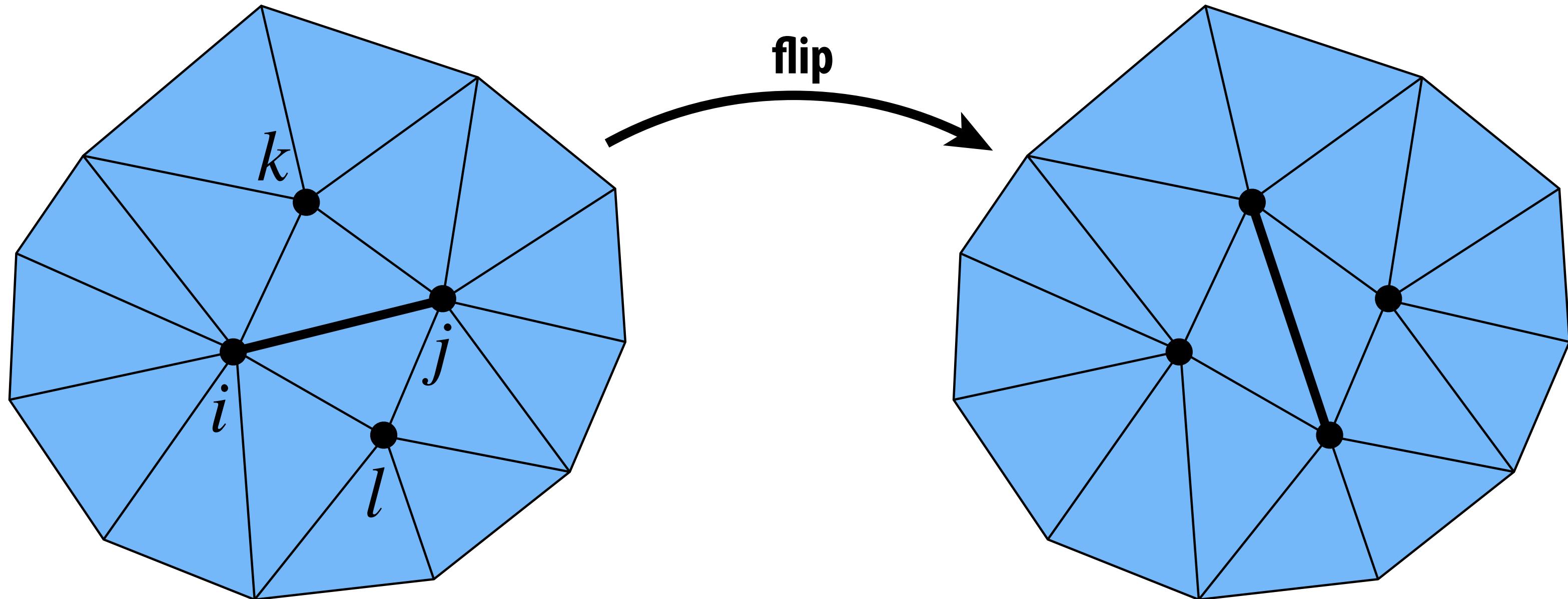
- Already have a good tool: edge flips!
- If  $\alpha + \beta > \pi$ , flip it!



- FACT: in 2D, flipping edges eventually yields Delaunay mesh
- Theory: worst case  $O(n^2)$ ; doesn't always work for surfaces in 3D
- Practice: simple, effective way to improve mesh quality

# Alternatively: how do we improve degree?

- Same tool: edge flips!
- If total deviation from degree-6 gets smaller, flip it!

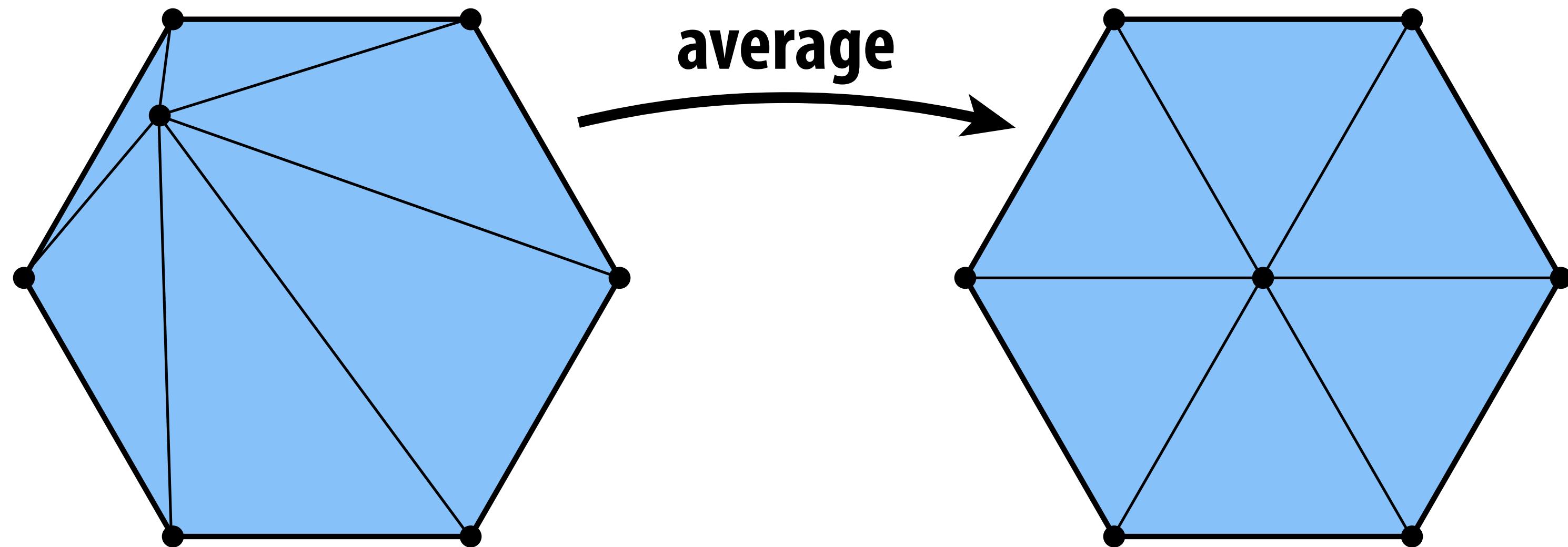


$$\text{total deviation: } |d_i - 6| + |d_j - 6| + |d_k - 6| + |d_l - 6|$$

- FACT: average degree approaches 6 as number of elements increases
- Iterative edge flipping acts like “discrete diffusion” of degree
- No (known) guarantees; works well in practice

# How do we make a triangles “more round”?

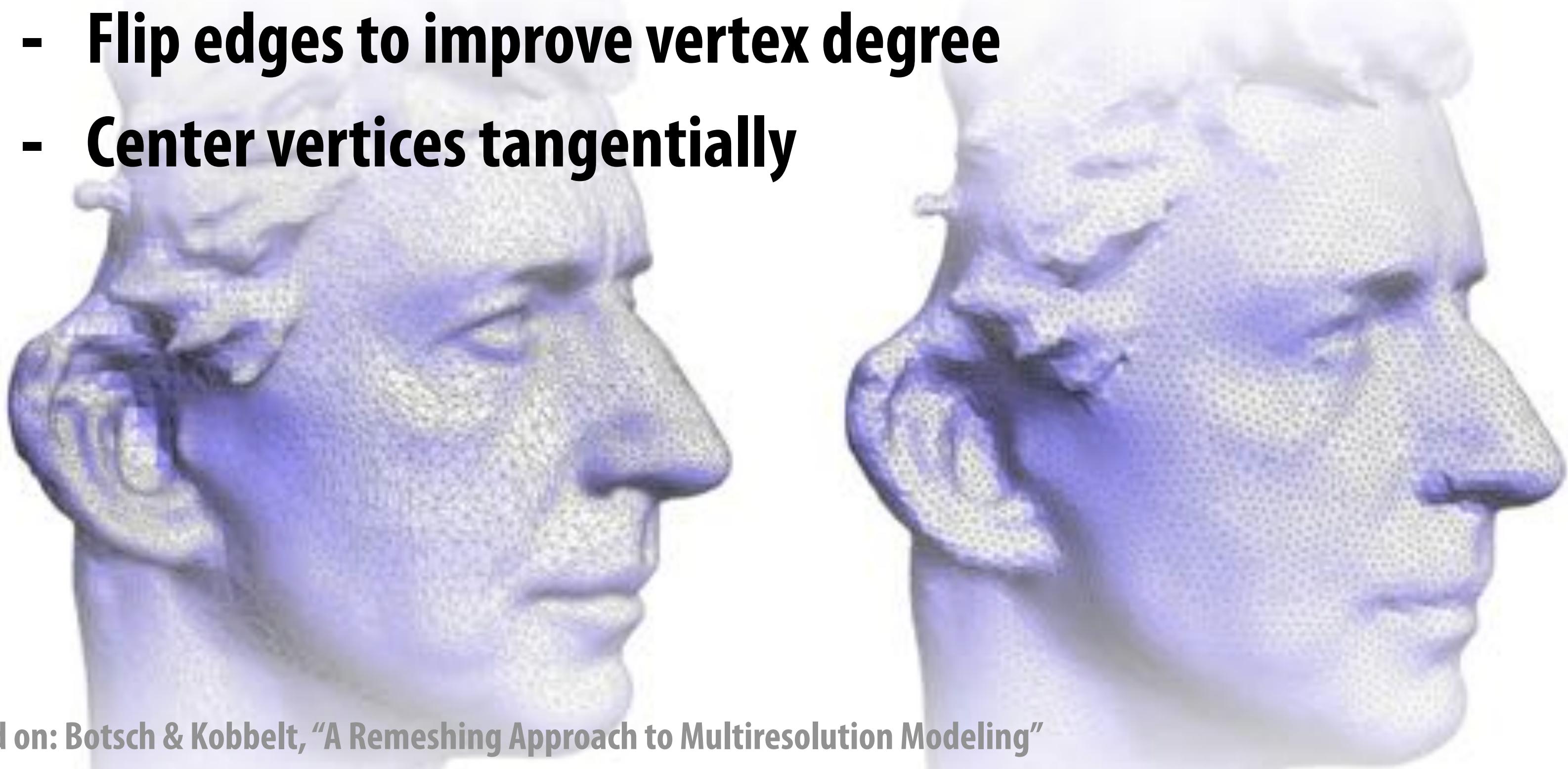
- Delaunay doesn’t guarantee triangles are “round” (angles near 60°)
- Can often improve shape by centering vertices:



- Simple version of technique called “Laplacian smoothing”
- On surface: move only in tangent direction
- How? Remove normal component from update vector

# Isotropic Remeshing Algorithm

- Try to make triangles uniform shape & size
- Repeat four steps:
  - Split any edge over 4/3rds mean edge length
  - Collapse any edge less than 4/5ths mean edge length
  - Flip edges to improve vertex degree
  - Center vertices tangentially

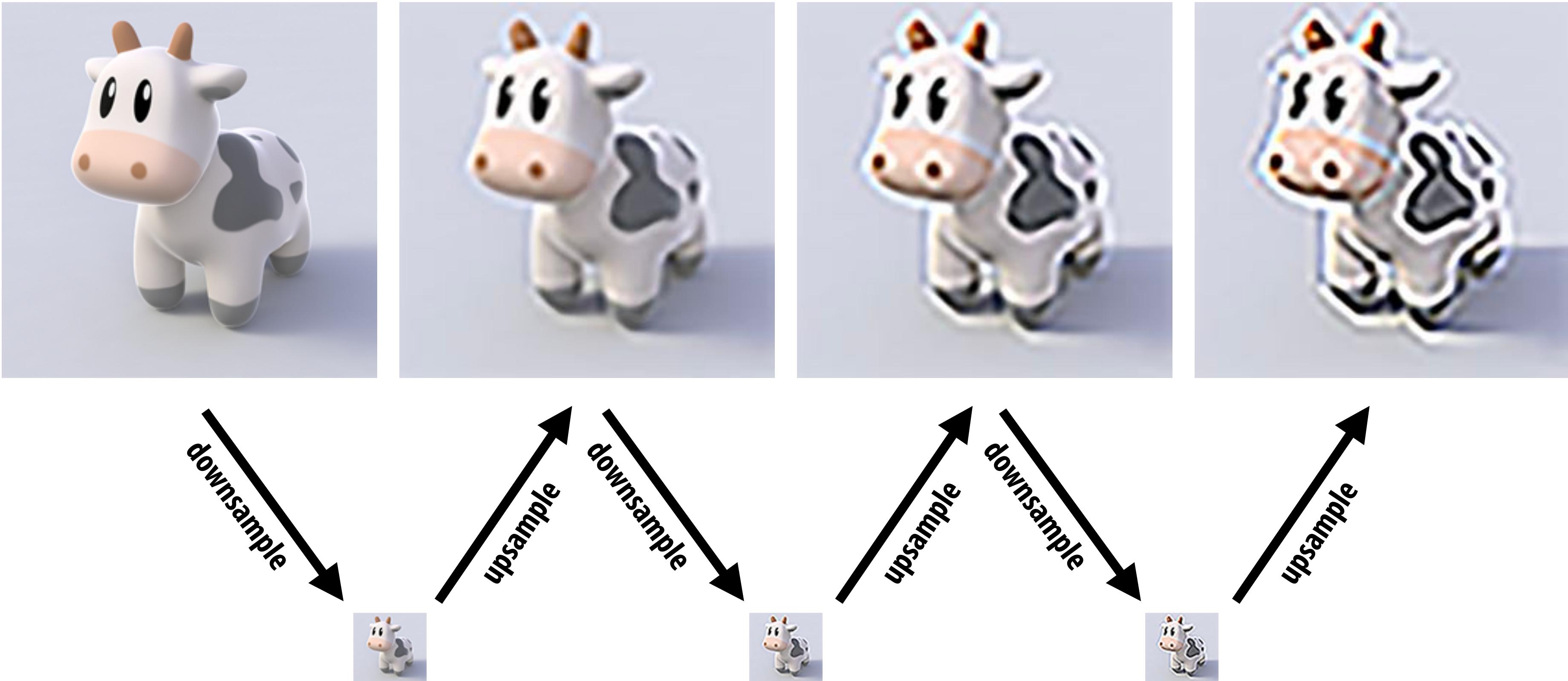


Based on: Botsch & Kobbelt, "A Remeshing Approach to Multiresolution Modeling"

**What can go wrong when  
you resample a signal?**

# Danger of Resampling

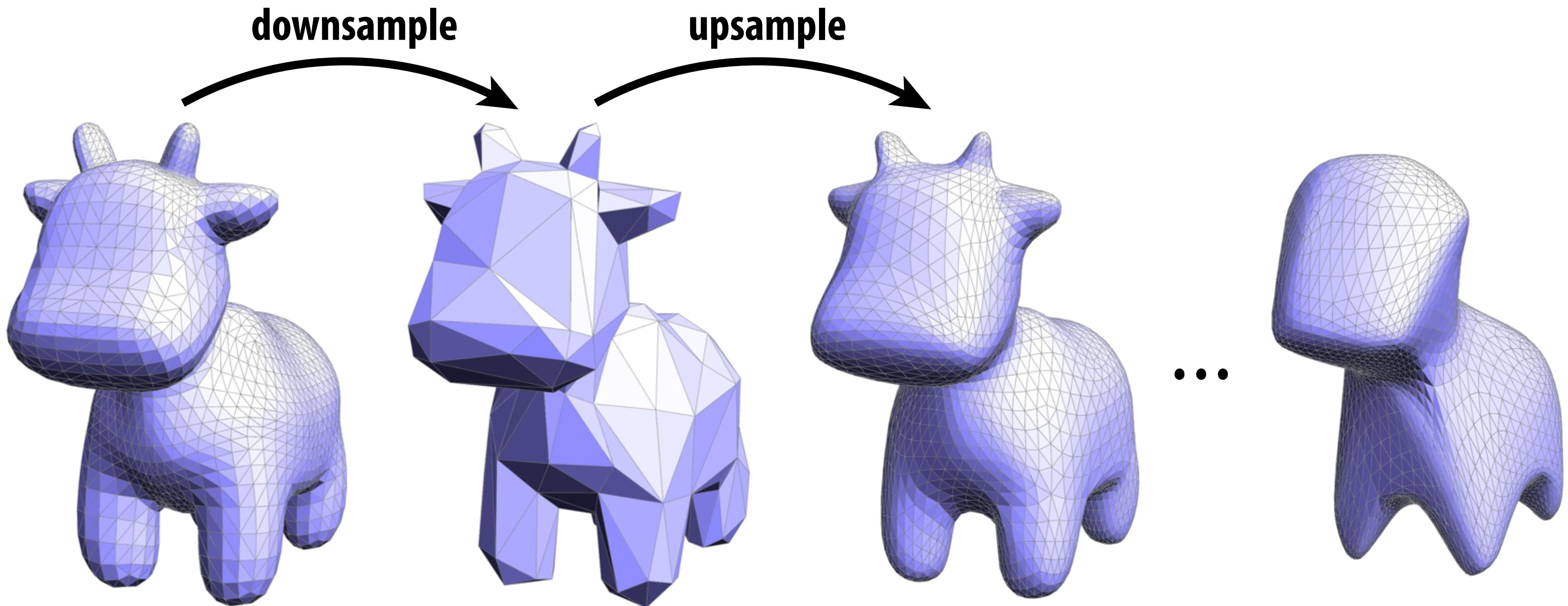
Q: What happens if we repeatedly resample an image?



A: Signal quality degrades!

# Danger of Resampling

**Q: What happens if we repeatedly resample a mesh?**



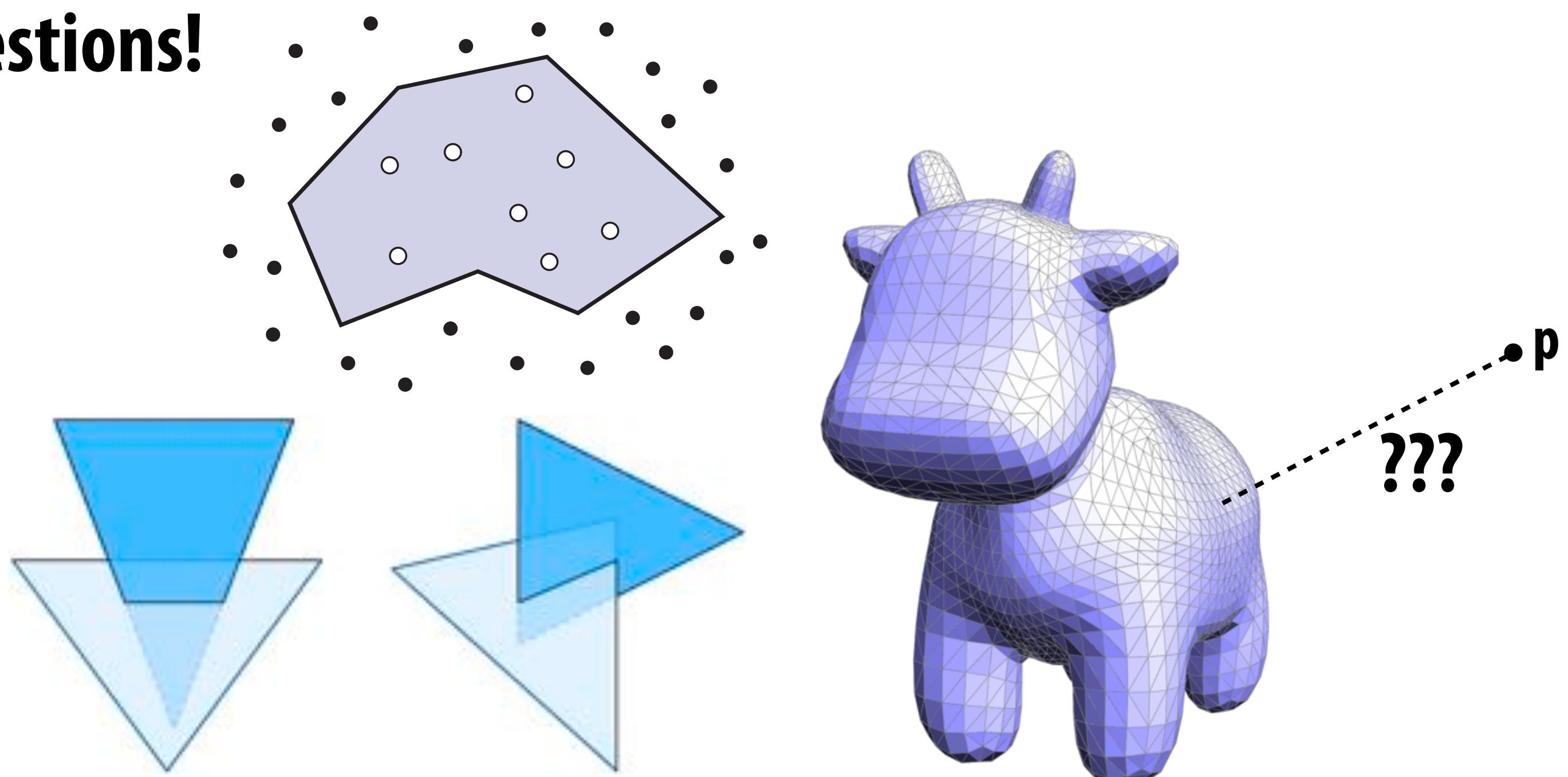
**A: Signal also degrades!**

**But wait: we have the original signal (mesh).**

**Why not just project each new sample point  
onto the closest point of the original mesh?**

# Next Time: Geometric Queries

- Q: Given a point, in space, how do we find the closest point on a surface? Are we inside or outside the surface? How do we find intersection of two triangles? Etc.
- Do implicit/explicit representations make such tasks easier?
- What's the cost of the naïve algorithm, and how do we accelerate such queries for large meshes?
- So many questions!





# COMPUTER GRAPHICS AND MULTIMEDIA

---

## Unit-6: Color Model

**Dr. Mrinmoy Ghorai**

**Indian Institute of Information Technology  
Sri City, Andhra Pradesh**

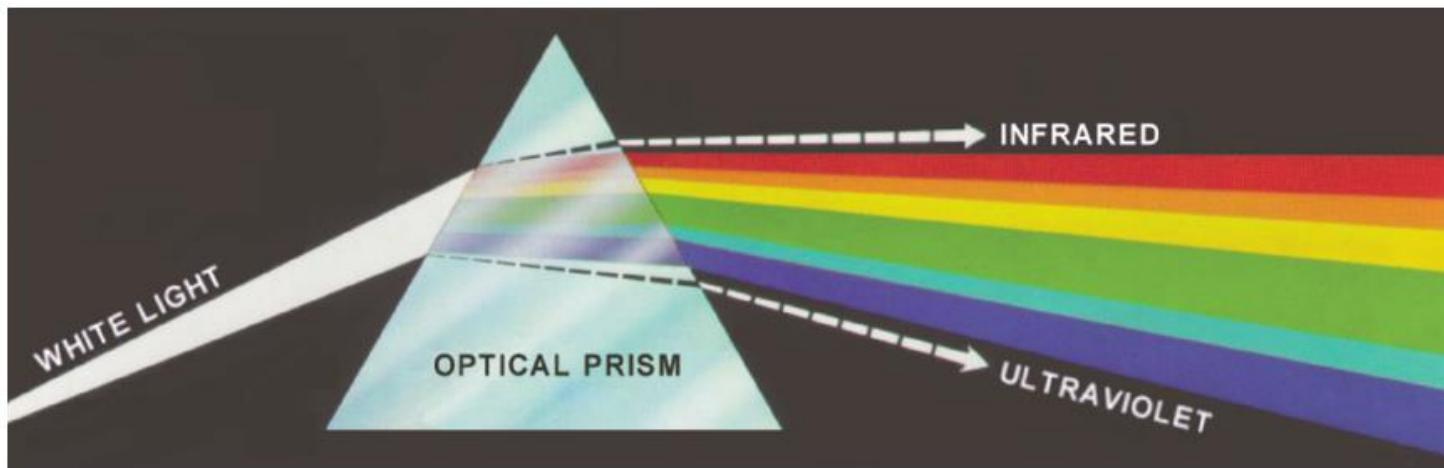
# Today's Lecture



- Color Fundamentals
- Color Models

# Color Model

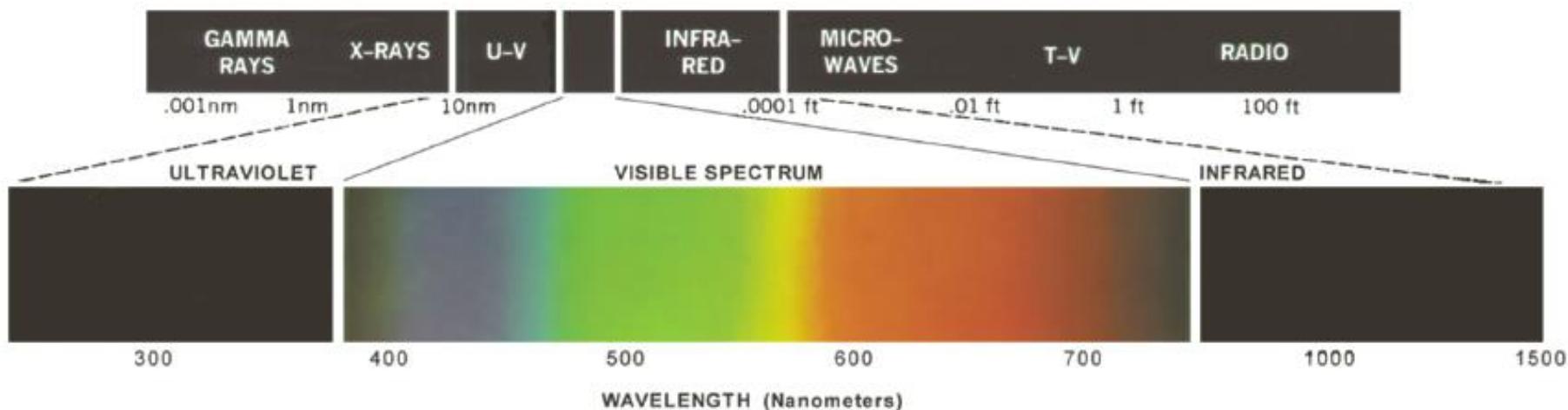
## Color Fundamentals



**FIGURE 6.1** Color spectrum seen by passing white light through a prism. (Courtesy of the General Electric Co., Lamp Business Division.)

# Color Model

## Color Fundamentals



**FIGURE 6.2** Wavelengths comprising the visible range of the electromagnetic spectrum. (Courtesy of the General Electric Co., Lamp Business Division.)



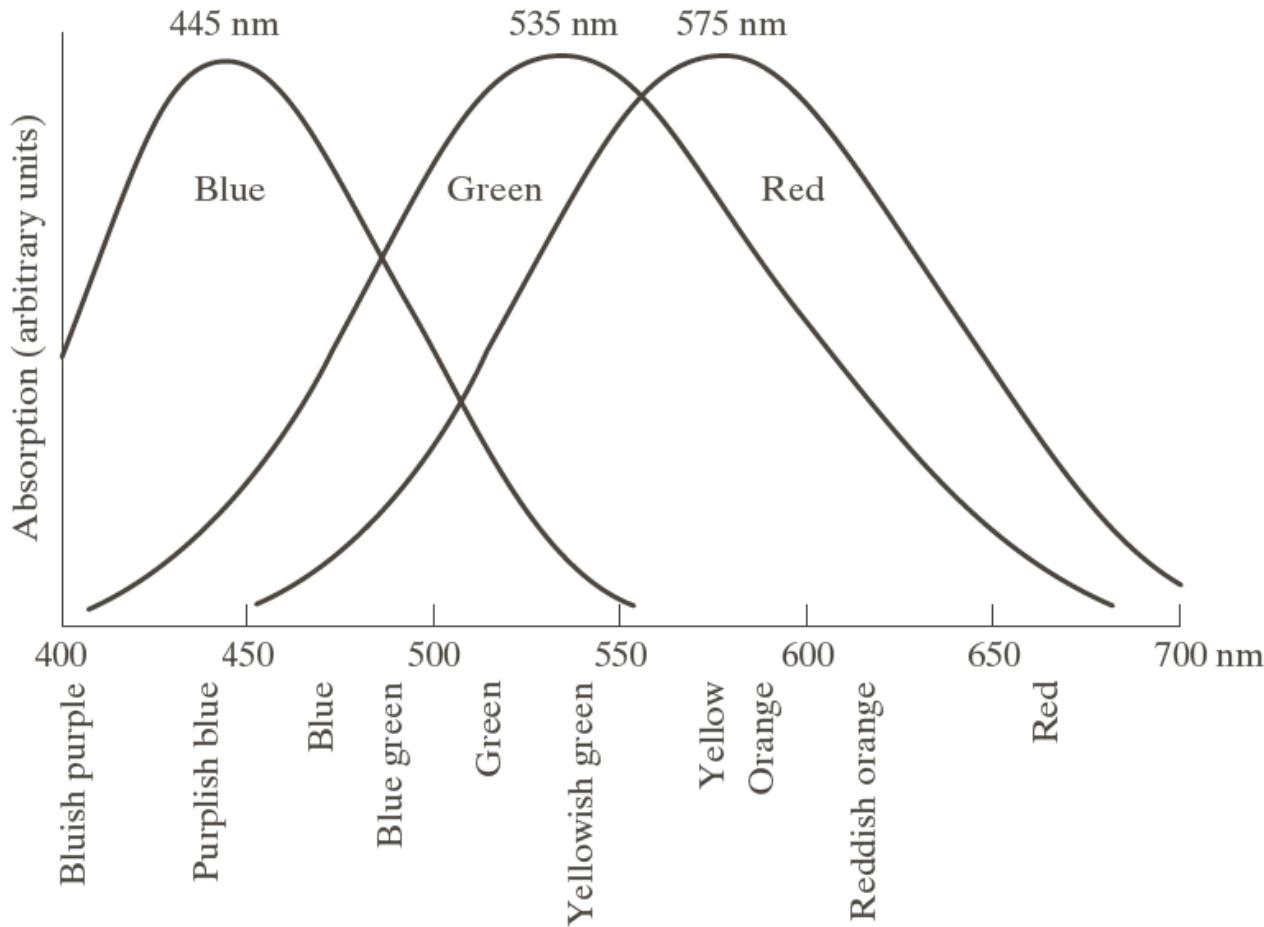
# Color Model

## Color Fundamentals

- 6 to 7 million cones in the human eye can be divided into three principal sensing categories, corresponding roughly to red, green, and blue.
- 65%: red    33%: green    2%: blue (blue cones are the most sensitive)

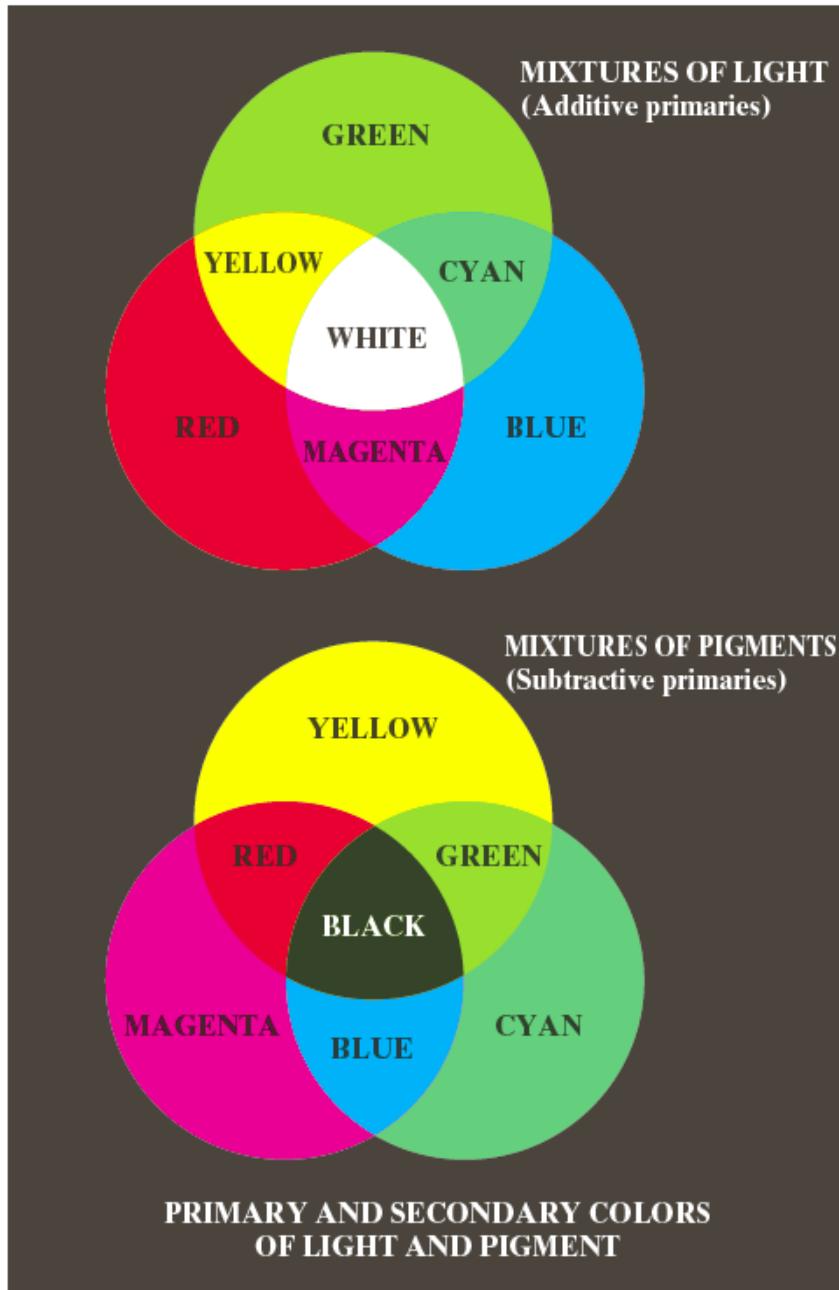
# Color Model

## Color Fundamentals



**FIGURE 6.3**

Absorption of light by the red, green, and blue cones in the human eye as a function of wavelength.



a  
b

**FIGURE 6.4**  
Primary and secondary colors of light and pigments.  
(Courtesy of the General Electric Co., Lamp Business Division.)



# Color Model

## Color Fundamentals

- The characteristics generally used to distinguish one color from another are brightness, hue, and saturation

**Brightness:** the achromatic notion of intensity.

**Hue:** dominant wavelength in a mixture of light waves, represents dominant color as perceived by an observer.

**Saturation:** relative purity or the amount of white light mixed with its hue.



# Color Model

## Color Fundamentals

- Tristimulus

Red, green, and blue are denoted X, Y, and Z, respectively. A color is defined by its trichromatic coefficients, defined as

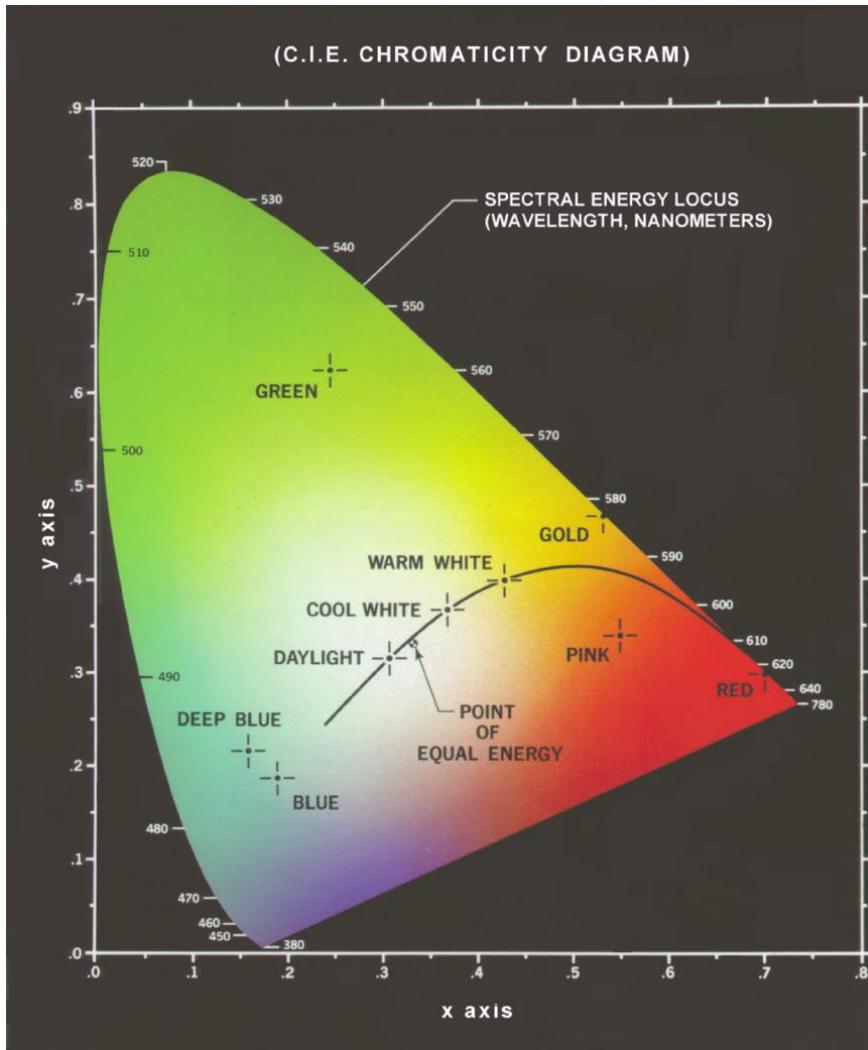
$$x = \frac{X}{X + Y + Z}$$

$$y = \frac{Y}{X + Y + Z}$$

$$z = \frac{Z}{X + Y + Z}$$

# Color Model

## CIE Chromaticity Diagram

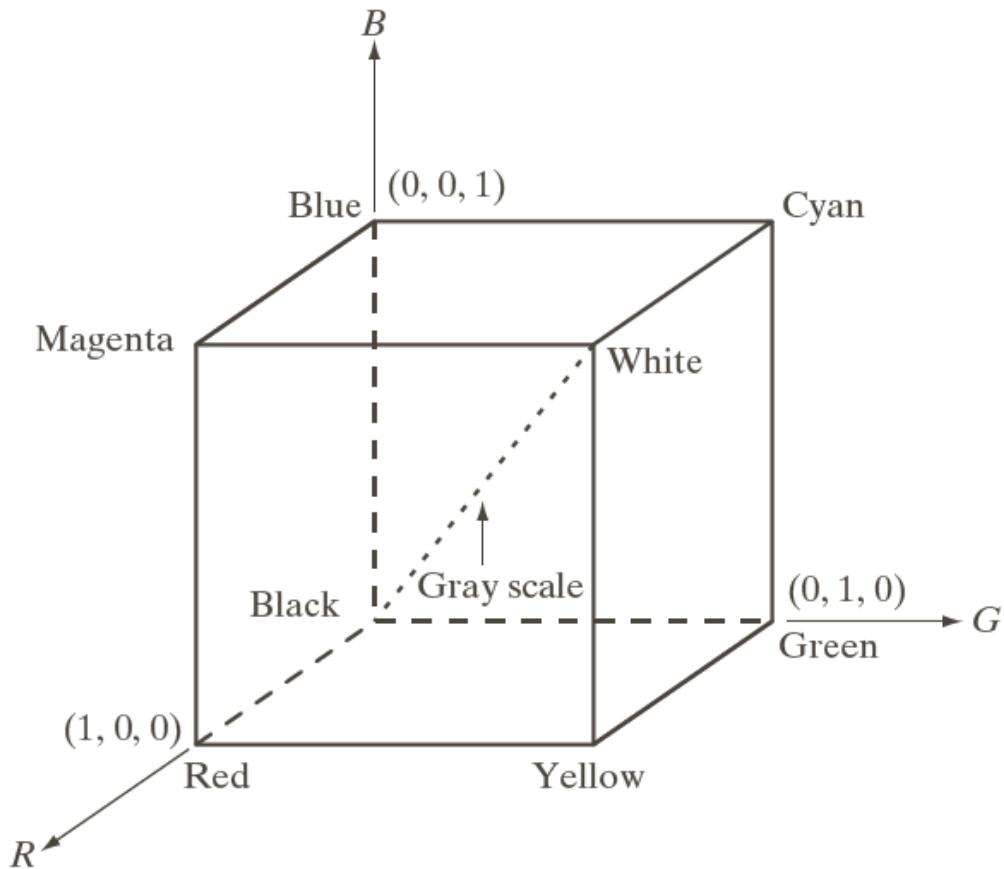


**FIGURE 6.5**  
Chromaticity  
diagram.  
(Courtesy of the  
General Electric  
Co., Lamp  
Business  
Division.)

It shows color composition as a function of x (red) and y (green)

# Color Model

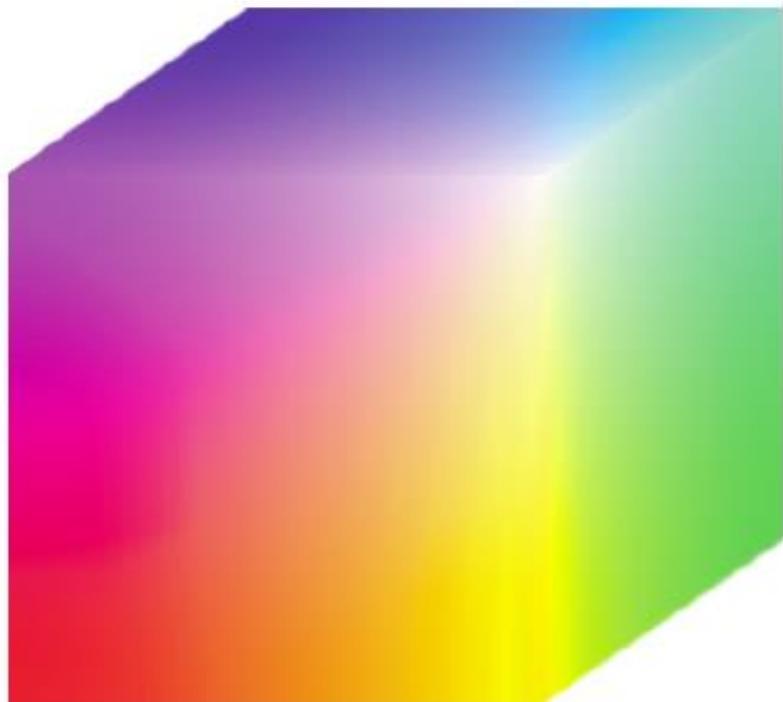
## RGB Color Model



**FIGURE 6.7**  
 Schematic of the RGB color cube. Points along the main diagonal have gray values, from black at the origin to white at point  $(1, 1, 1)$ .

# Color Model

## RGB Color Model



**FIGURE 6.8** RGB  
24-bit color cube.

The total number  
of colors in a 24-bit  
RGB image is  $(2^8)^3$   
 $= 16,777,216$



# Color Model

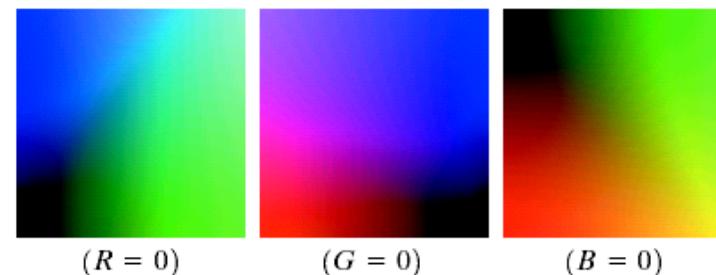
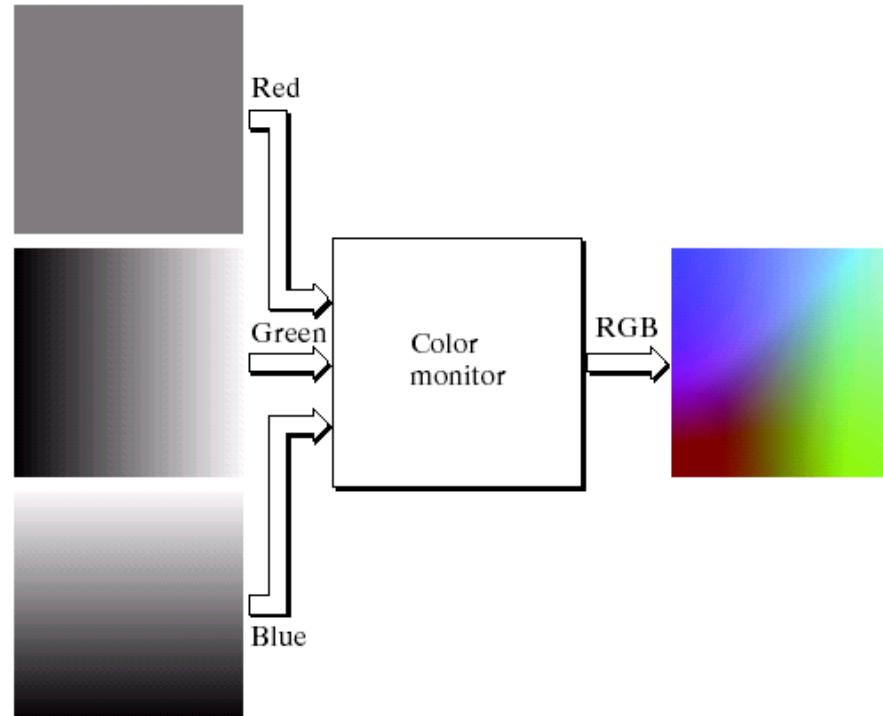
## RGB Color Model

a  
b

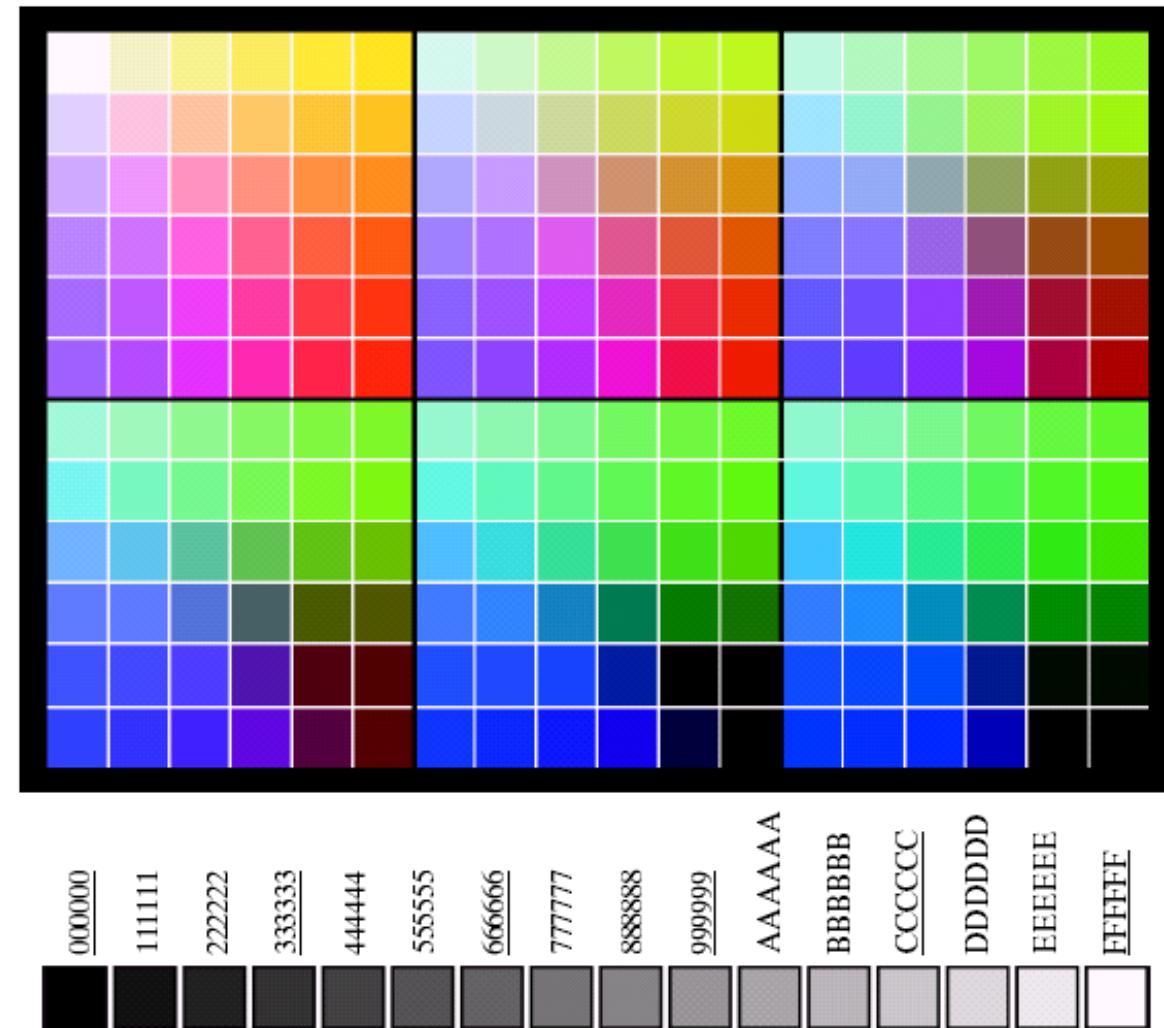
**FIGURE 6.9**

(a) Generating the RGB image of the cross-sectional color plane  $(127, G, B)$ .

(b) The three hidden surface planes in the color cube of Fig. 6.8.



**TABLE 6.1**  
Valid values of each RGB component in a safe color.

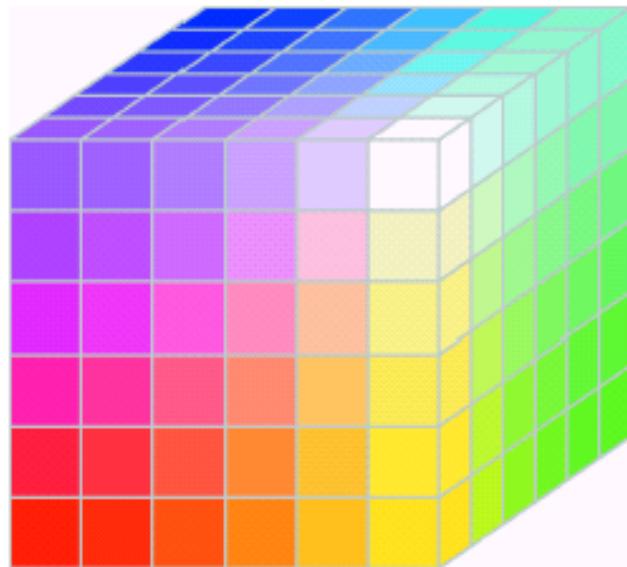


a  
b

**FIGURE 6.10**  
(a) The 216 safe RGB colors.  
(b) All the grays in the 256-color RGB system (grays that are part of the safe color group are shown underlined).

# Color Model

## RGB Color Model



**FIGURE 6.11** The RGB safe-color cube.



# Color Model

## The CMY and CMYK Color Models

$$\begin{bmatrix} C \\ M \\ Y \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} - \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

- Equal amounts of the pigment primaries, cyan, magenta, and yellow should produce black. In practice, combining these colors for printing produces a muddy-looking black.
- To produce true black, the predominant color in printing, the fourth color, black, is added, giving rise to the CMYK color model.



CMY vs. CMYK





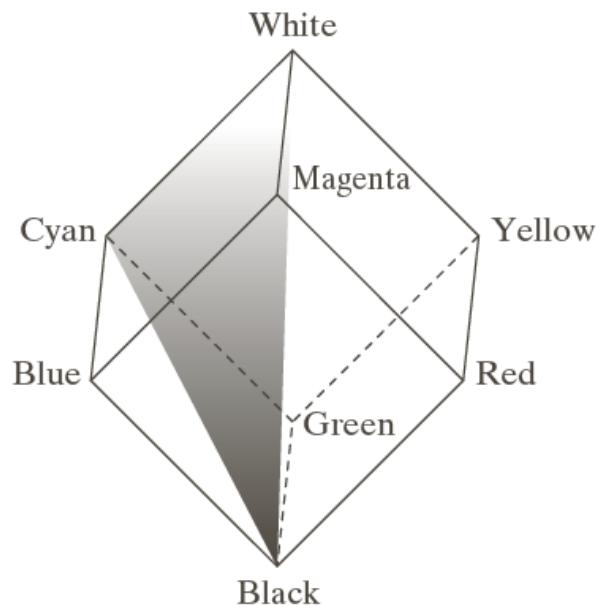
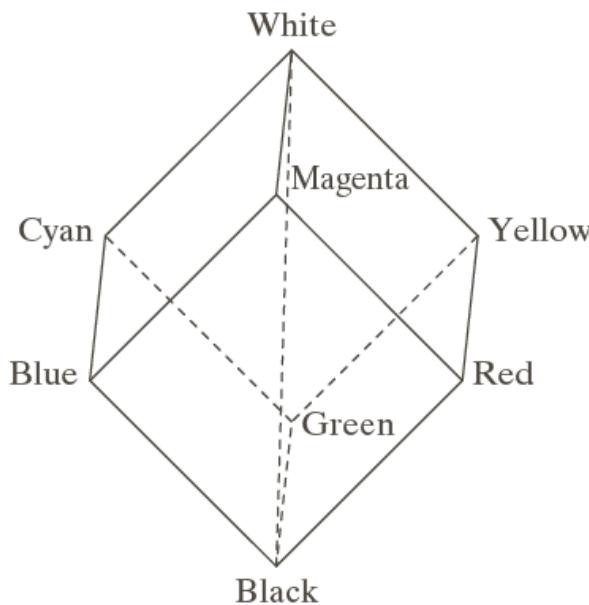
# Color Model

## HSI Color Model

- **Hue:** dominant wavelength in a mixture of light waves, represents dominant color as perceived by an observer.
- **Saturation:** relative purity or the amount of white light mixed with its hue.
- **Intensity:** the achromatic notion of **intensity**.

# Color Model

## HSI Color Model

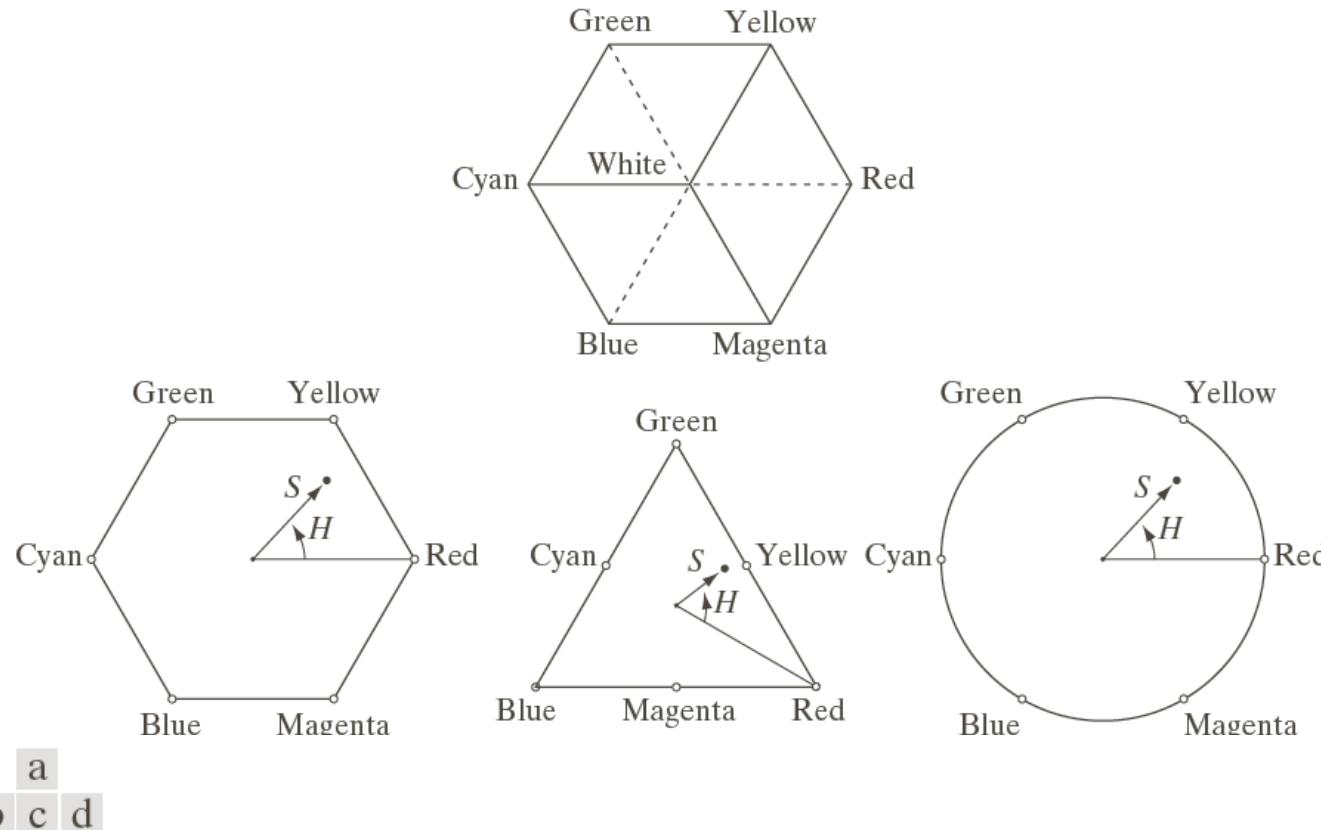


a b

**FIGURE 6.12**  
Conceptual  
relationships  
between the RGB  
and HSI color  
models.

# Color Model

## HSI Color Model



**FIGURE 6.13** Hue and saturation in the HSI color model. The dot is an arbitrary color point. The angle from the red axis gives the hue, and the length of the vector is the saturation. The intensity of all colors in any of these planes is given by the position of the plane on the vertical intensity axis.

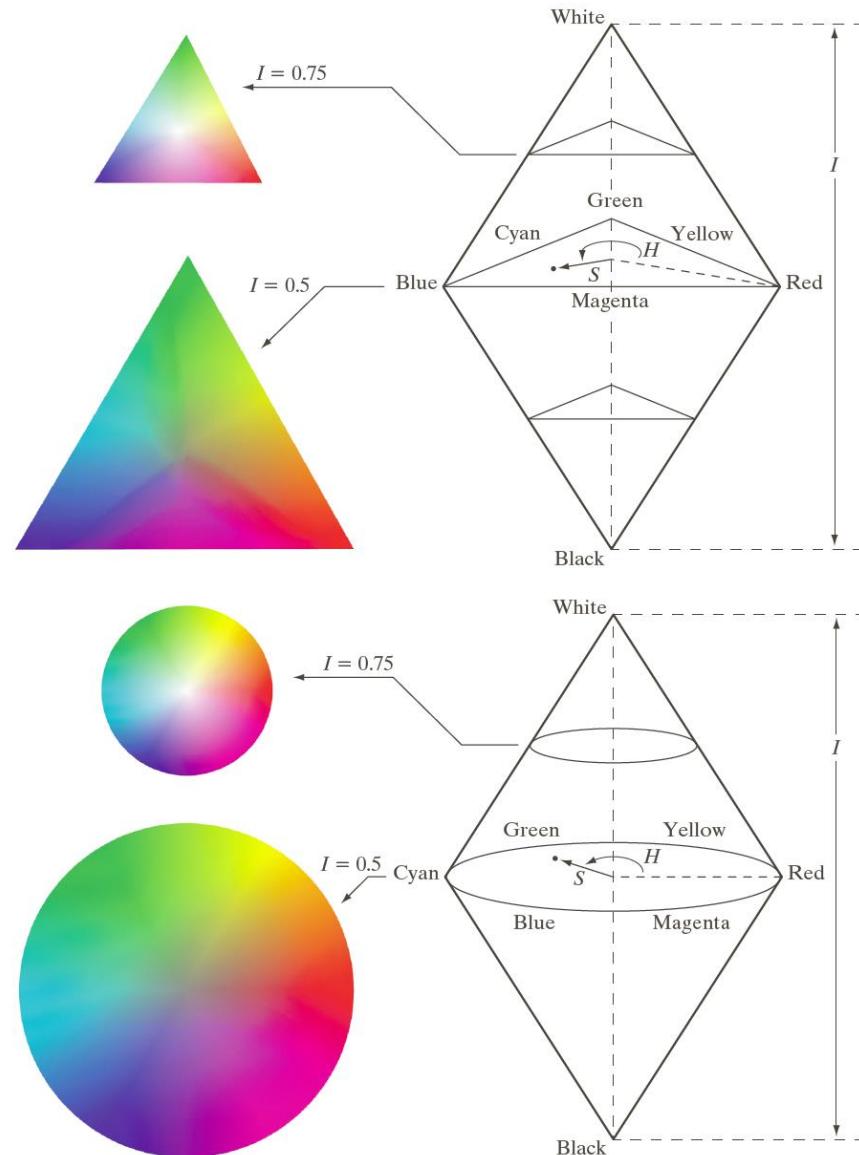
# Color Model

## HSI Color Model



a  
b

**FIGURE 6.14** The HSI color model based on (a) triangular and (b) circular color planes. The triangles and circles are perpendicular to the vertical intensity axis.





# Color Model

## Converting Colors from RGB to HSI

- Given an image in RGB color format, the H component of each RGB pixel is obtained using the equation

$$H = \begin{cases} \theta & \text{if } B \leq G \\ 360 - \theta & \text{if } B > G \end{cases}$$

$$\theta = \cos^{-1} \left\{ \frac{\frac{1}{2}[(R-G) + (R-B)]}{\left[ (R-G)^2 + (R-B)(G-B) \right]^{1/2}} \right\}$$



# Color Model

## Converting Colors from RGB to HSI

- Given an image in RGB color format, the saturation component is given by

$$S = 1 - \frac{3}{(R + G + B)} [\min(R, G, B)]$$



# Color Model

## Converting Colors from RGB to HSI

- Given an image in RGB color format, the intensity component is given by

$$I = \frac{1}{3}(R + G + B)$$



# Color Model

## Converting Colors from HSI to RGB

RG sector ( $0^\circ \leq H < 120^\circ$ )

$$B = I(1 - S)$$

$$R = I \left[ 1 + \frac{S \cos H}{\cos(60^\circ - H)} \right]$$

and

$$G = 3I - (R + B)$$



# Color Model

## Converting Colors from HSI to RGB

RG sector ( $120^\circ \leq H < 240^\circ$ )

$$H = H - 120^\circ$$

$$R = I(1 - S)$$

$$G = I \left[ 1 + \frac{S \cos H}{\cos(60^\circ - H)} \right]$$

and

$$B = 3I - (R + G)$$



# Color Model

## Converting Colors from HSI to RGB

RG sector ( $240^\circ \leq H \leq 360^\circ$ )

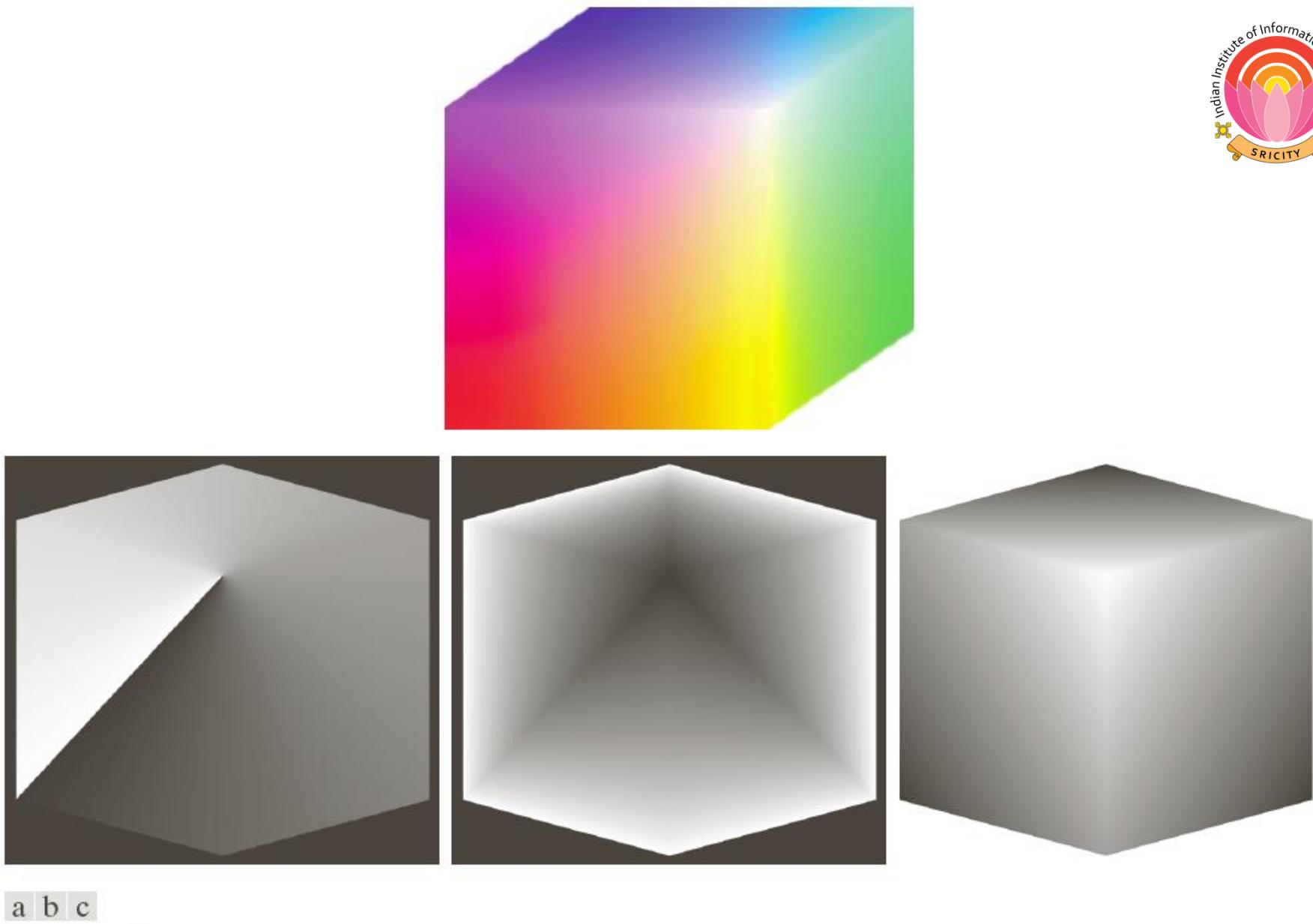
$$H = H - 240^\circ$$

$$G = I(1 - S)$$

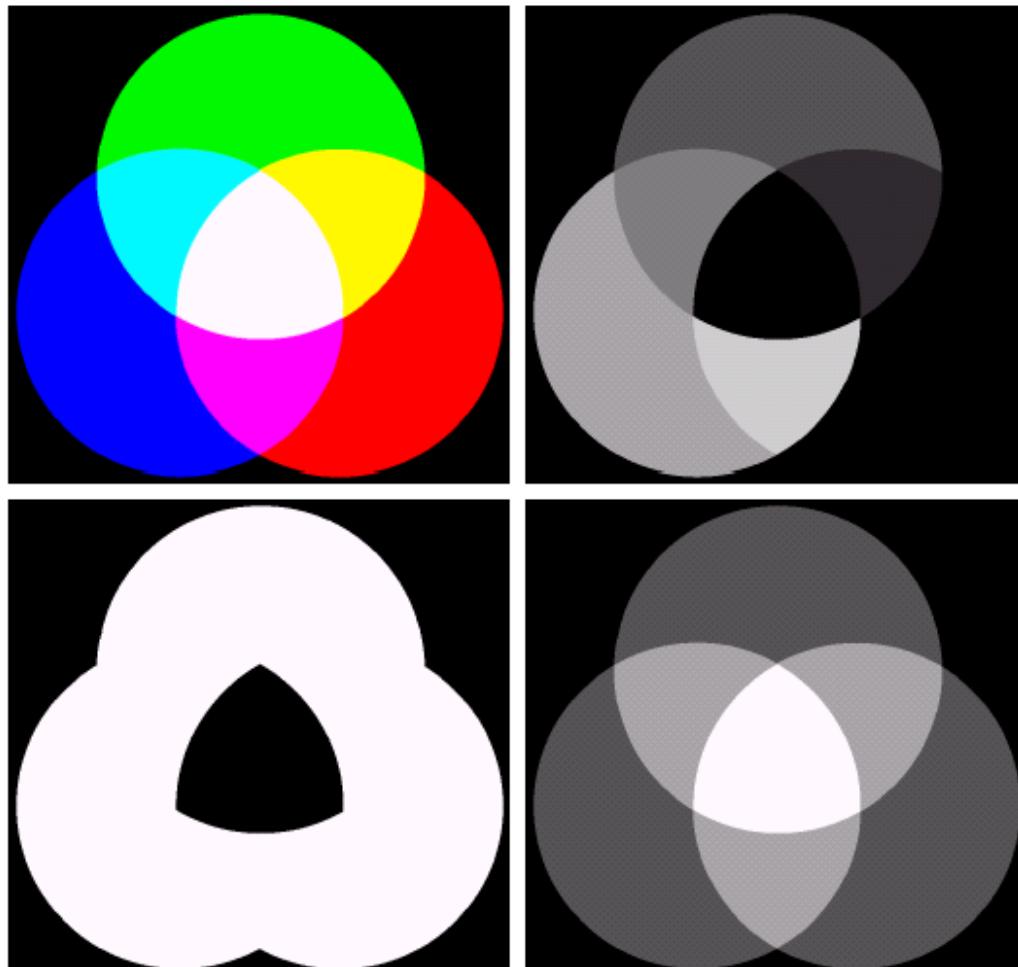
$$B = I \left[ 1 + \frac{S \cos H}{\cos(60^\circ - H)} \right]$$

and

$$R = 3I - (G + B)$$

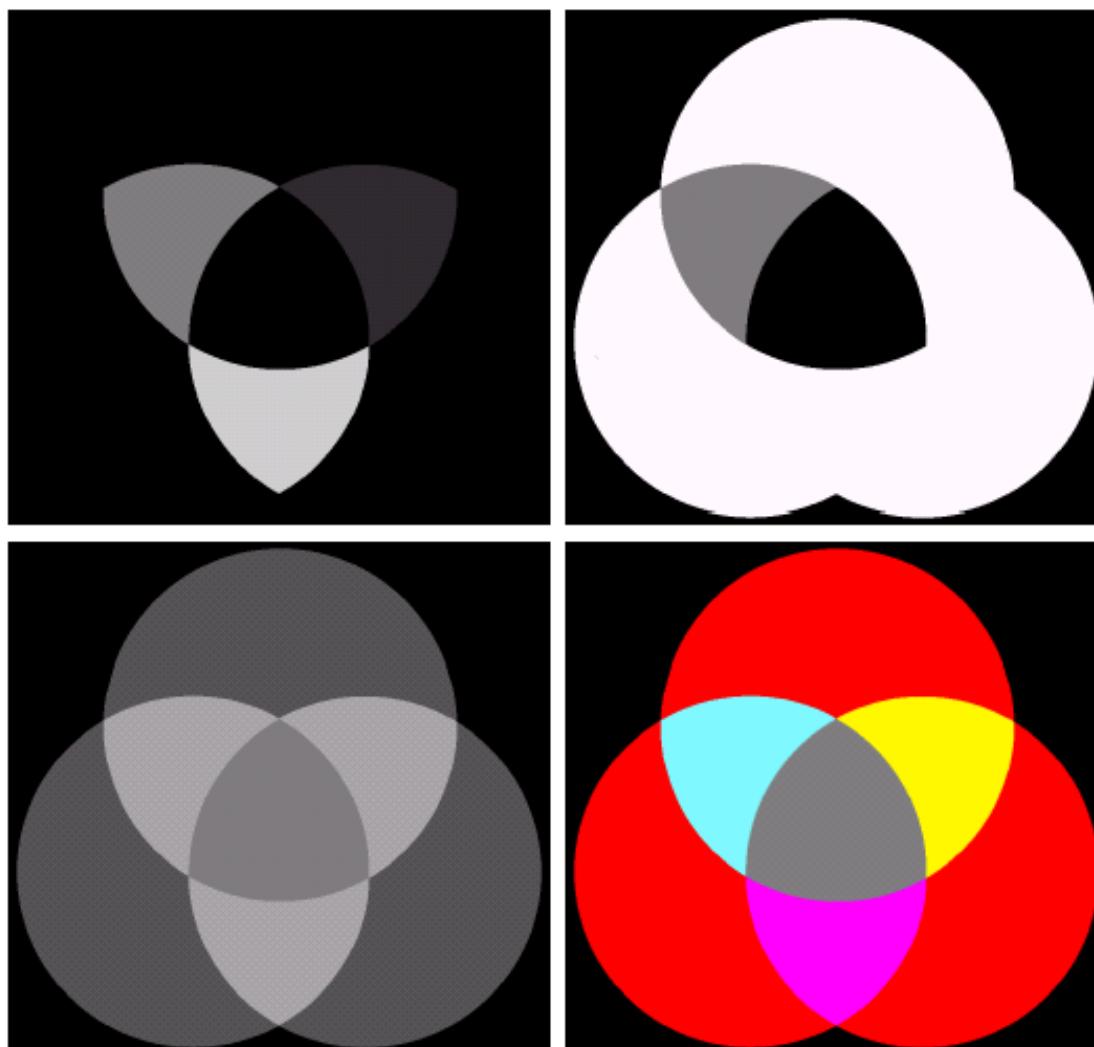


**FIGURE 6.15** HSI components of the image in Fig. 6.8. (a) Hue, (b) saturation, and (c) intensity images.



a b  
c d

**FIGURE 6.16** (a) RGB image and the components of its corresponding HSI image:  
(b) hue, (c) saturation, and (d) intensity.



a	b
c	d

**FIGURE 6.17** (a)–(c) Modified HSI component images. (d) Resulting RGB image.  
(See Fig. 6.16 for the original HSI images.)

# **Color**

---

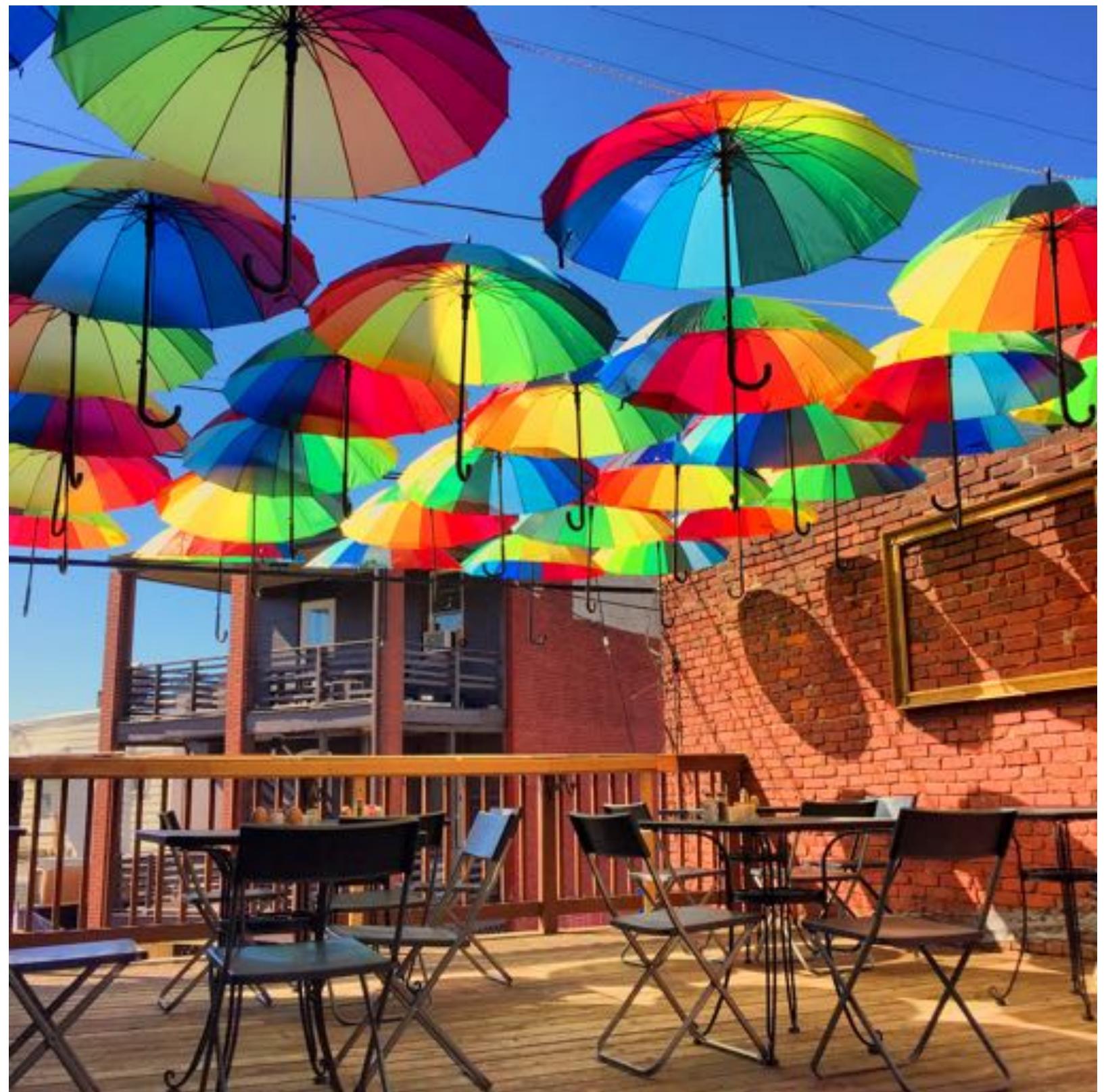
**Computer Graphics**  
**CMU 15-462/15-662**

# Why do we need to be able to talk precisely about color?

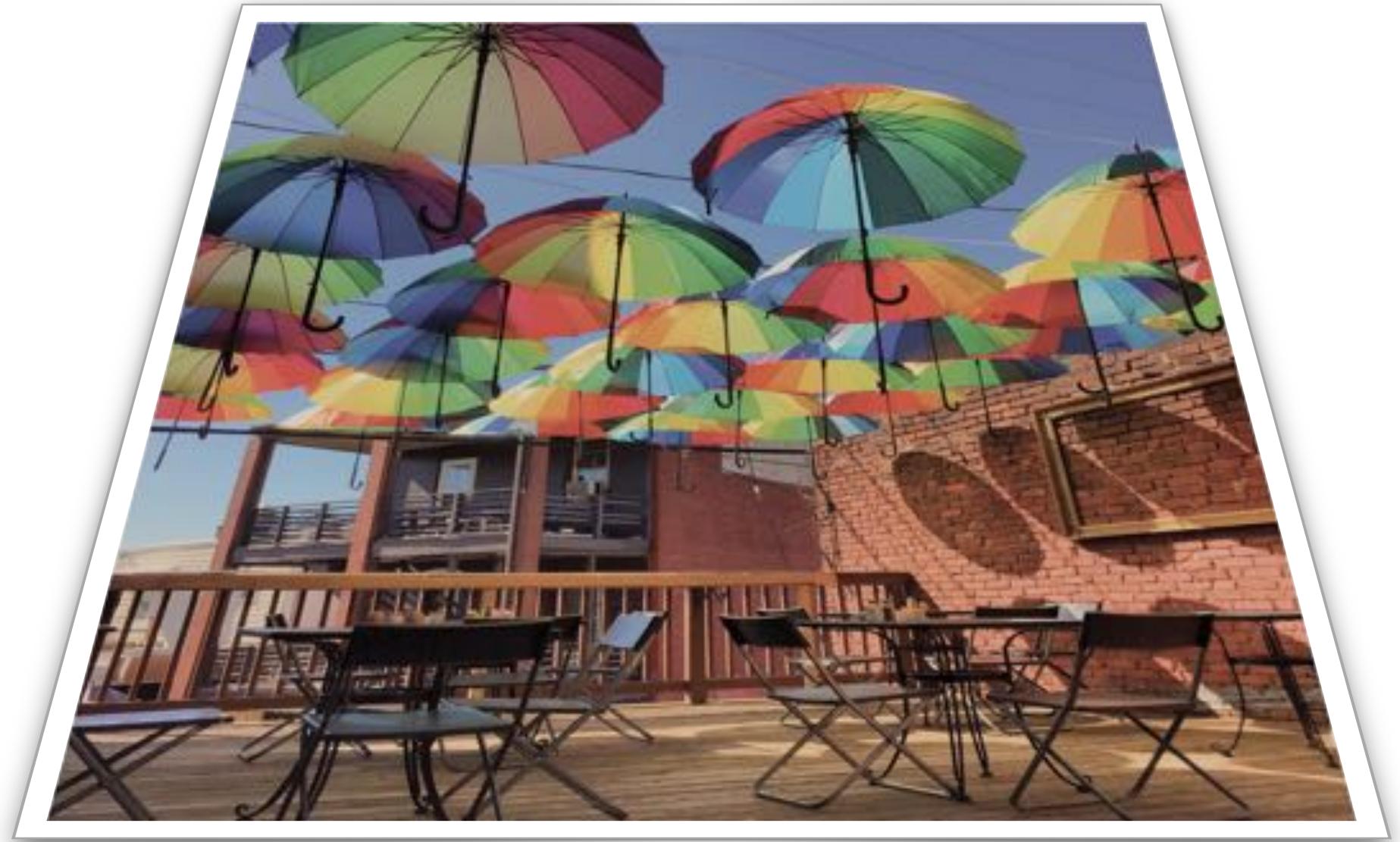








**on screen**

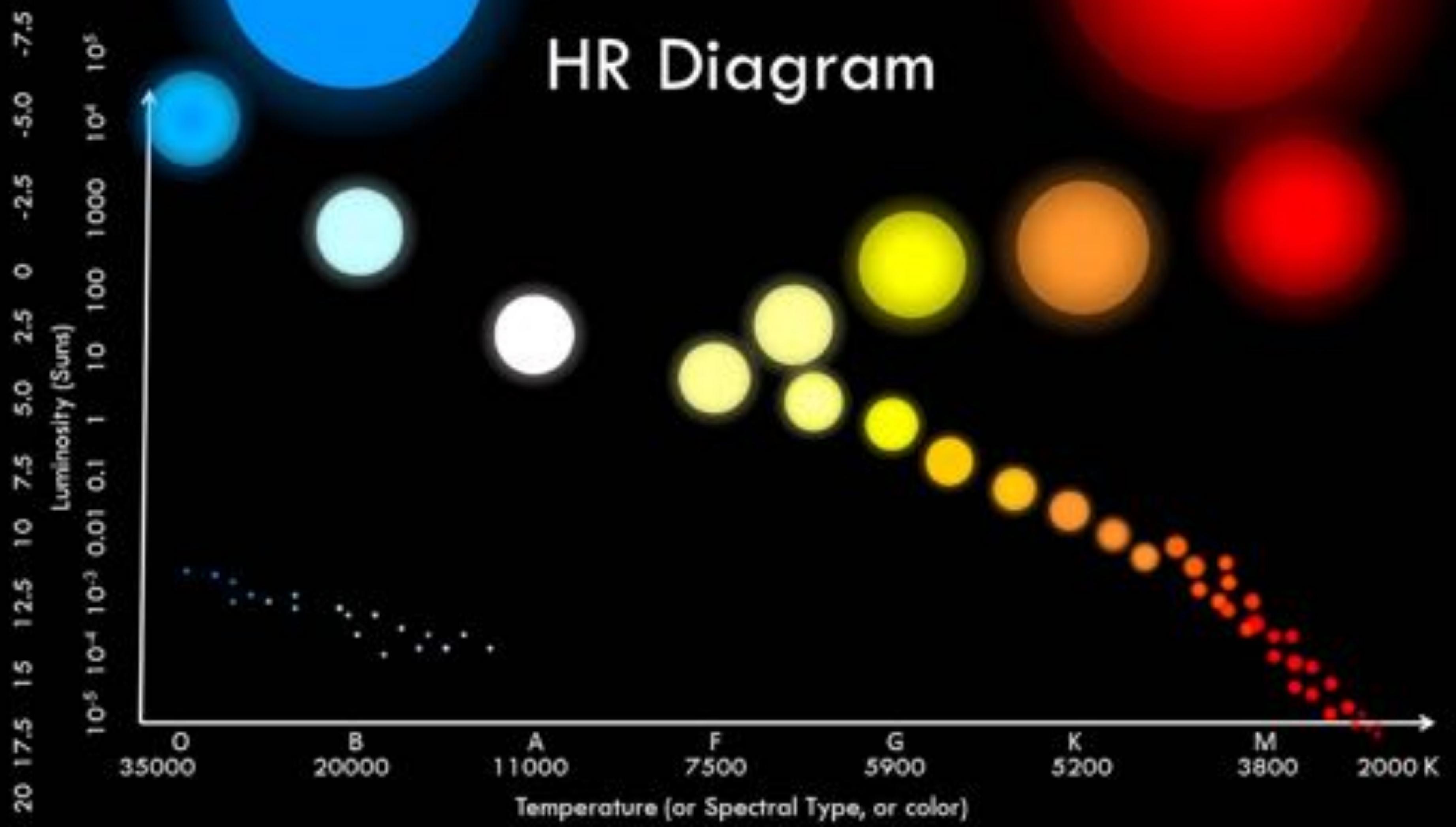


**printed**



Zhangye Danxia Geological Park, China

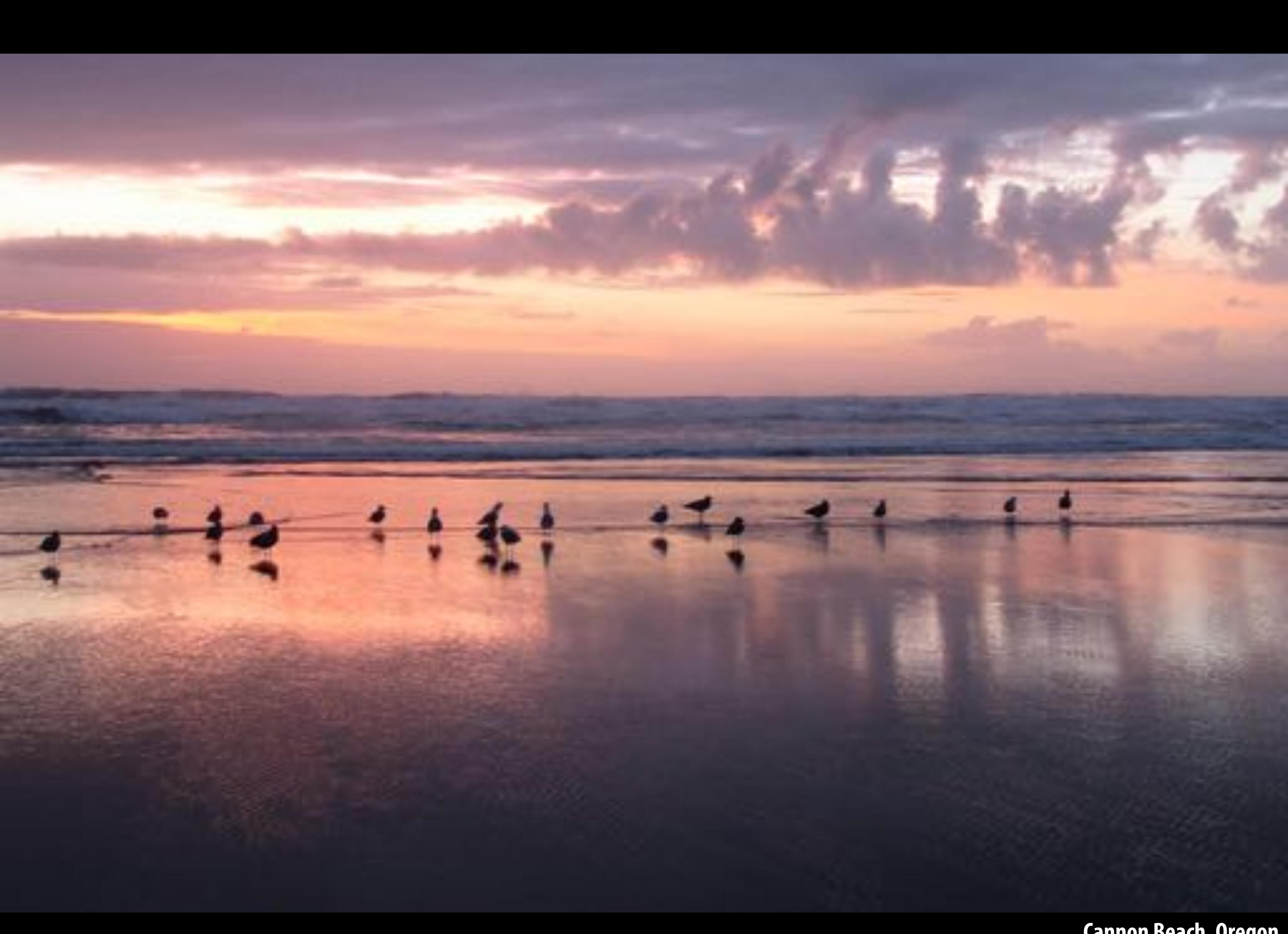
# HR Diagram



Hertzsprung-Russell diagram



**Starry Night, Van Gogh**

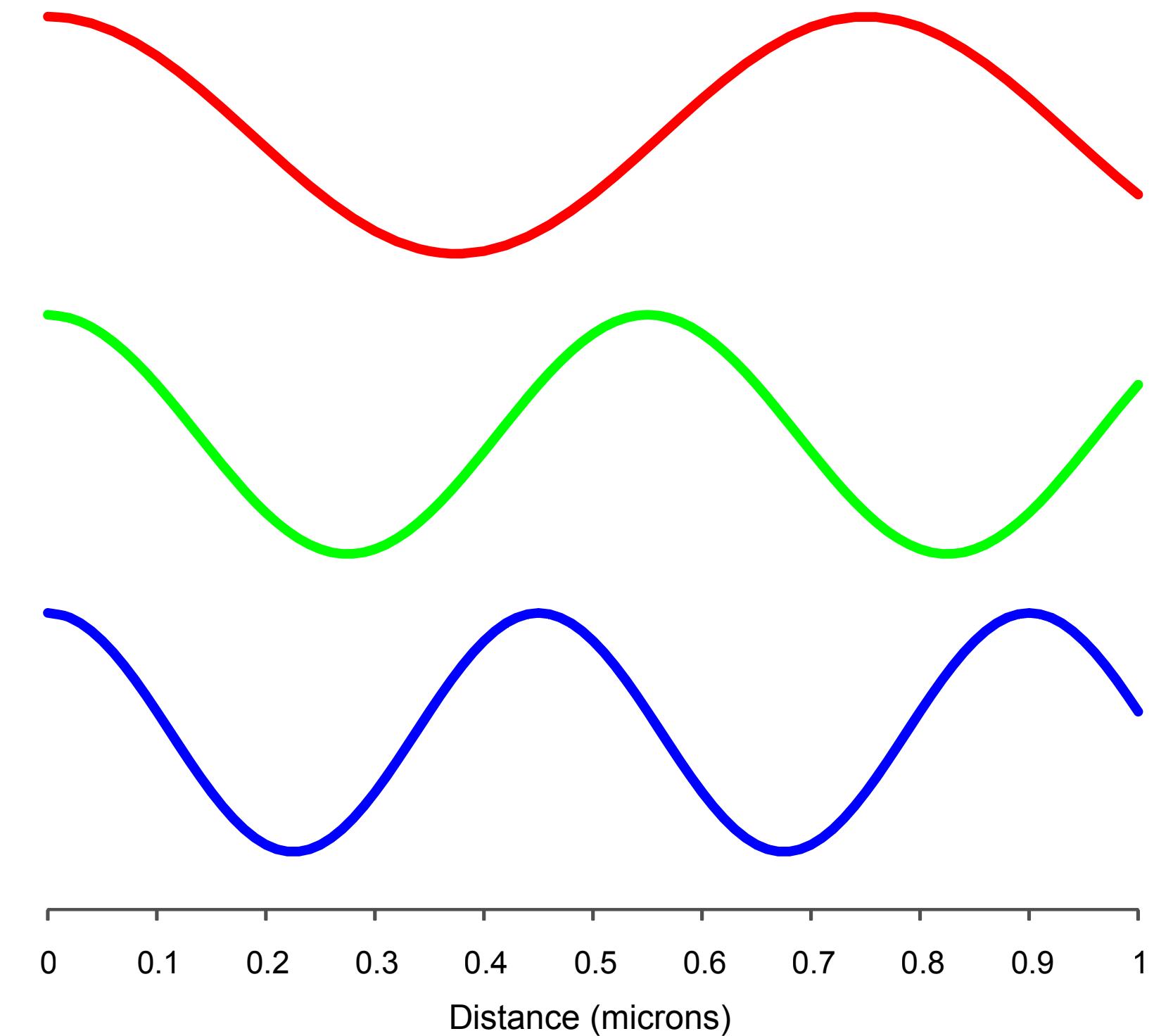
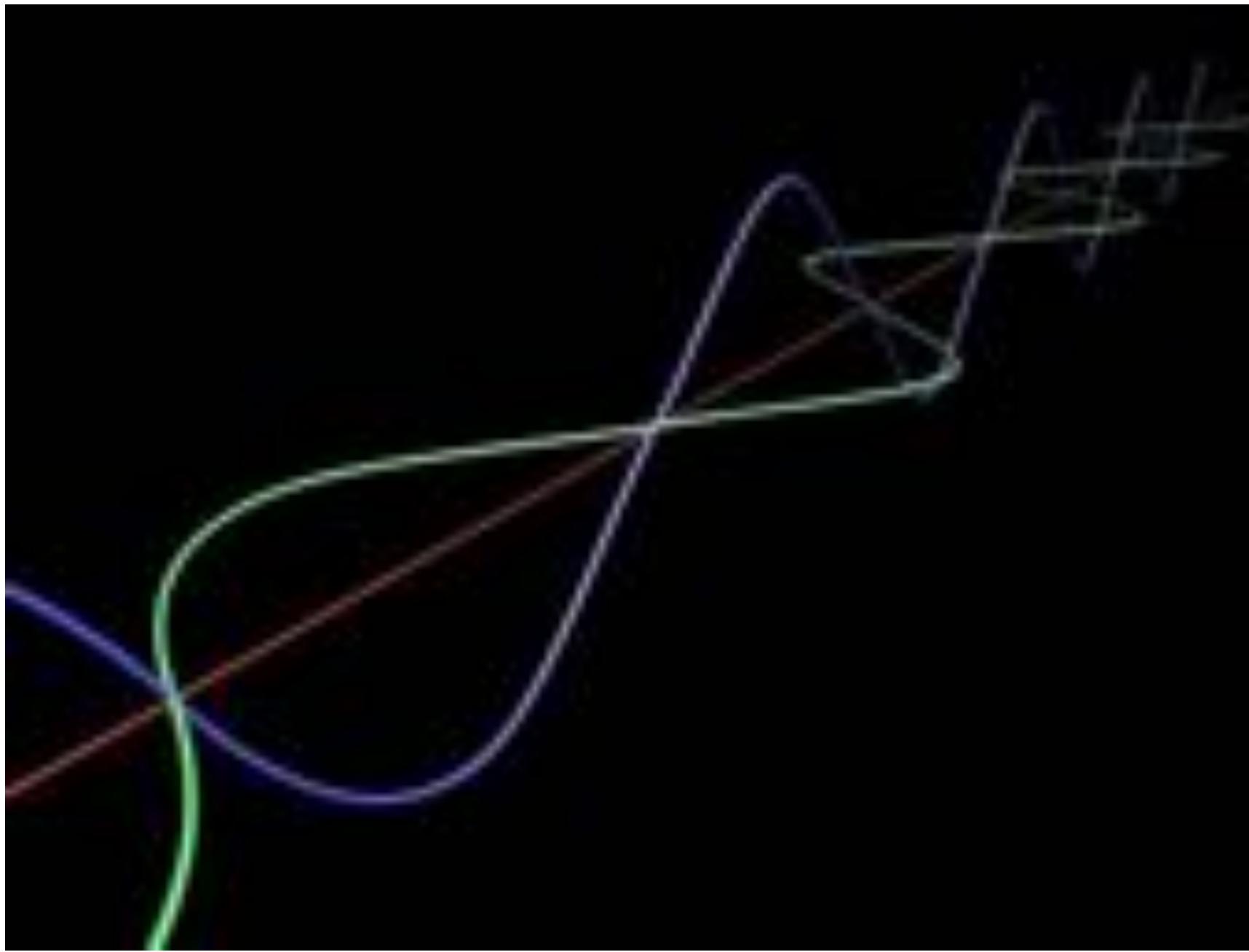


Cannon Beach, Oregon

# **What is color?**

# Light is EM Radiation; Color is Frequency

- Light is oscillating electric & magnetic field
- KEY IDEA: frequency determines color of light
- Q: What is the difference between frequency and wavelength?

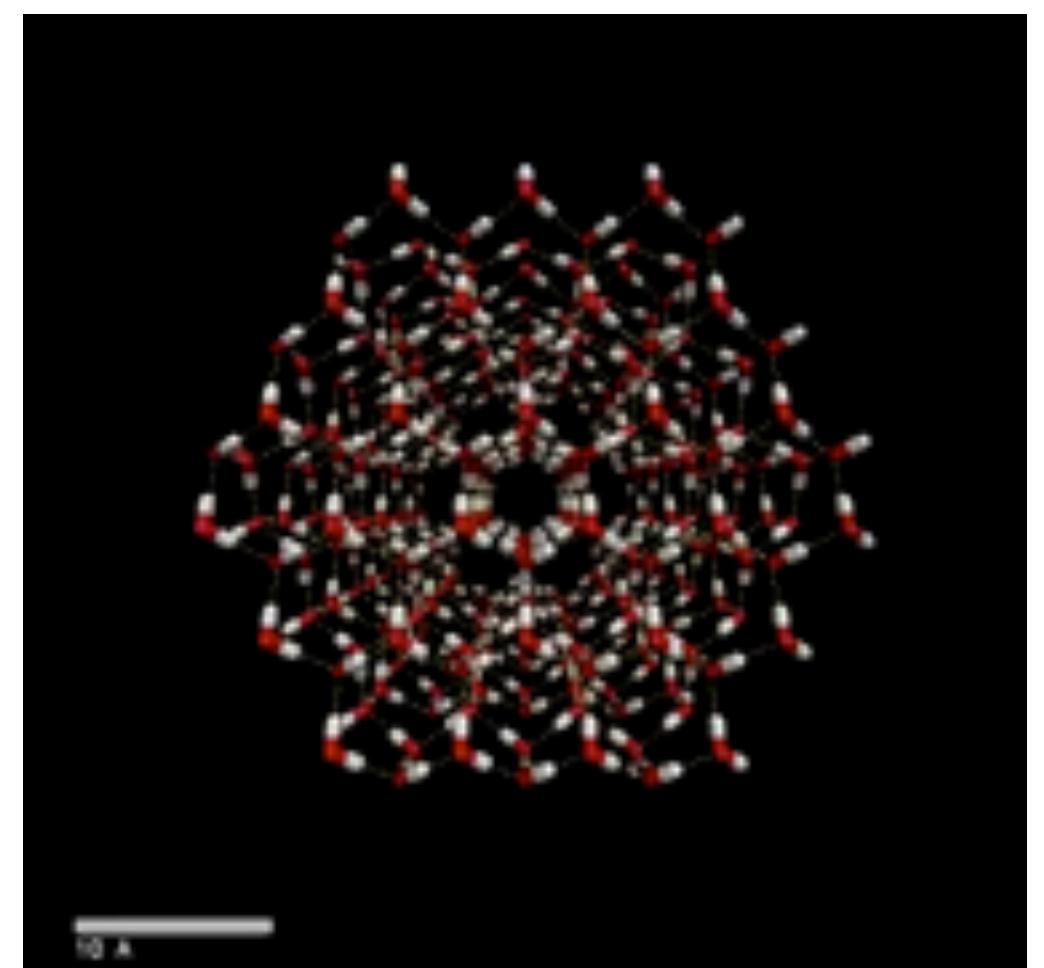
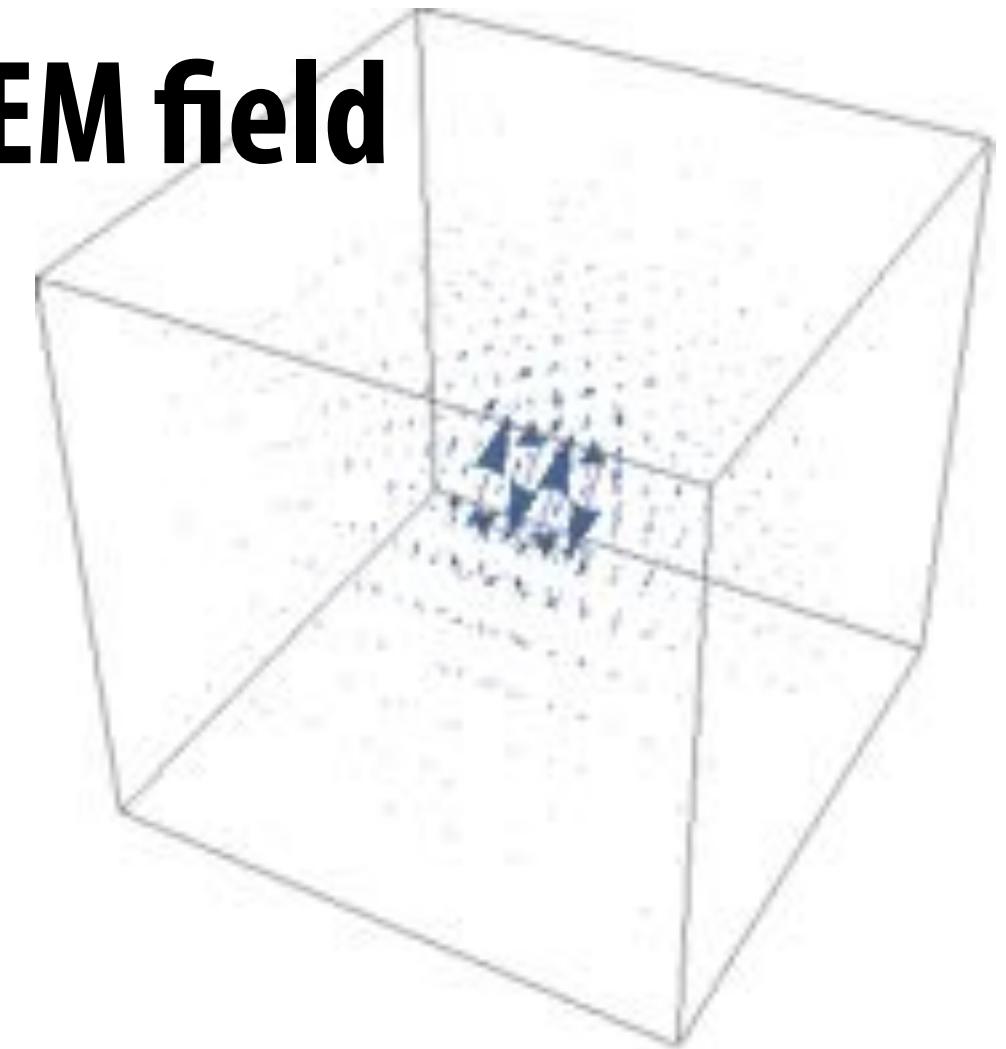




**Q: Why does your stove turn **red** when it heats up?**

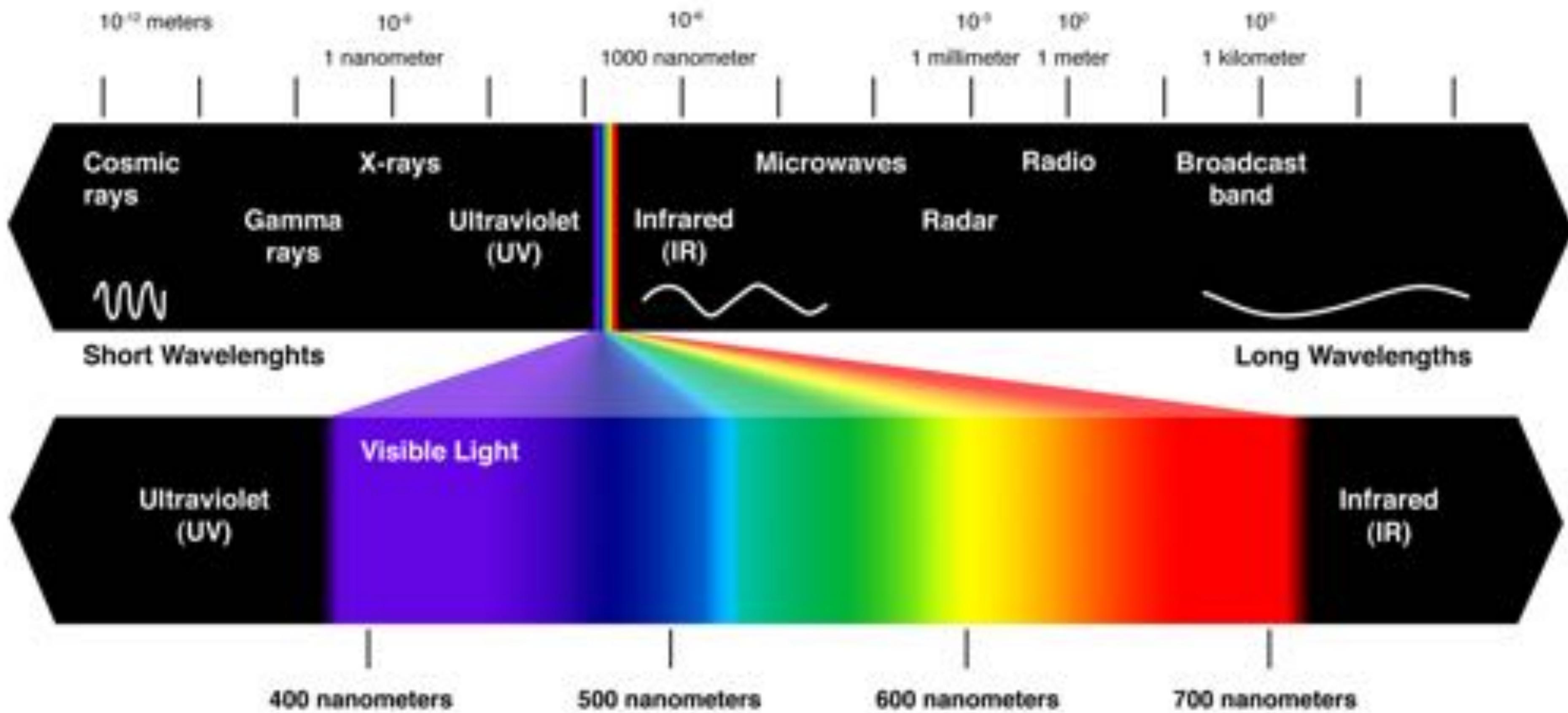
# Heat generates light

- One of many ways light is produced:
- Maxwell: motion of charged particles creates EM field
- Thermodynamics: ...particles jiggle around!
- Hence, anything moving generates light
- In other words:
  - every object around you is producing color!
  - frequency determined by temperature



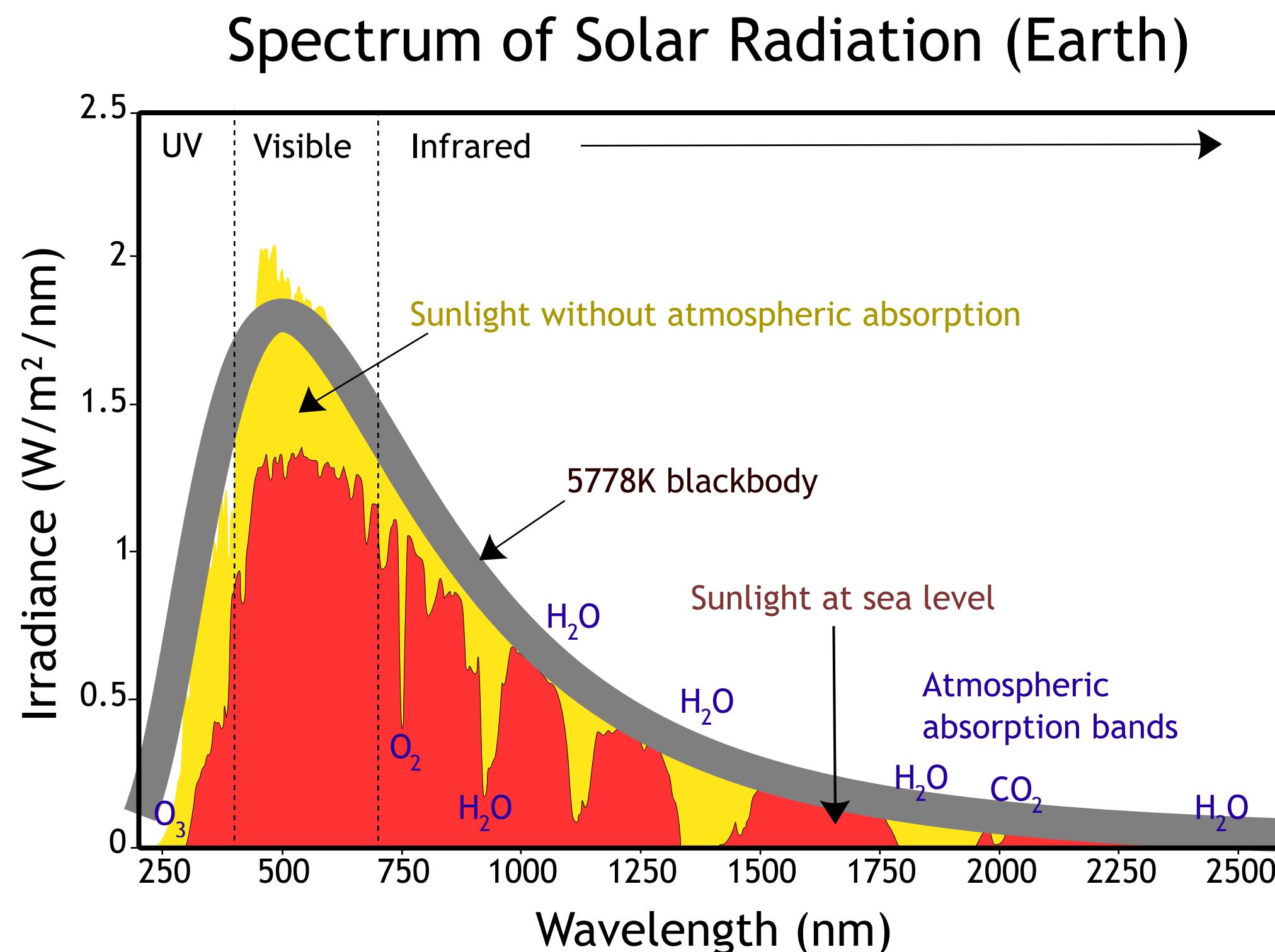
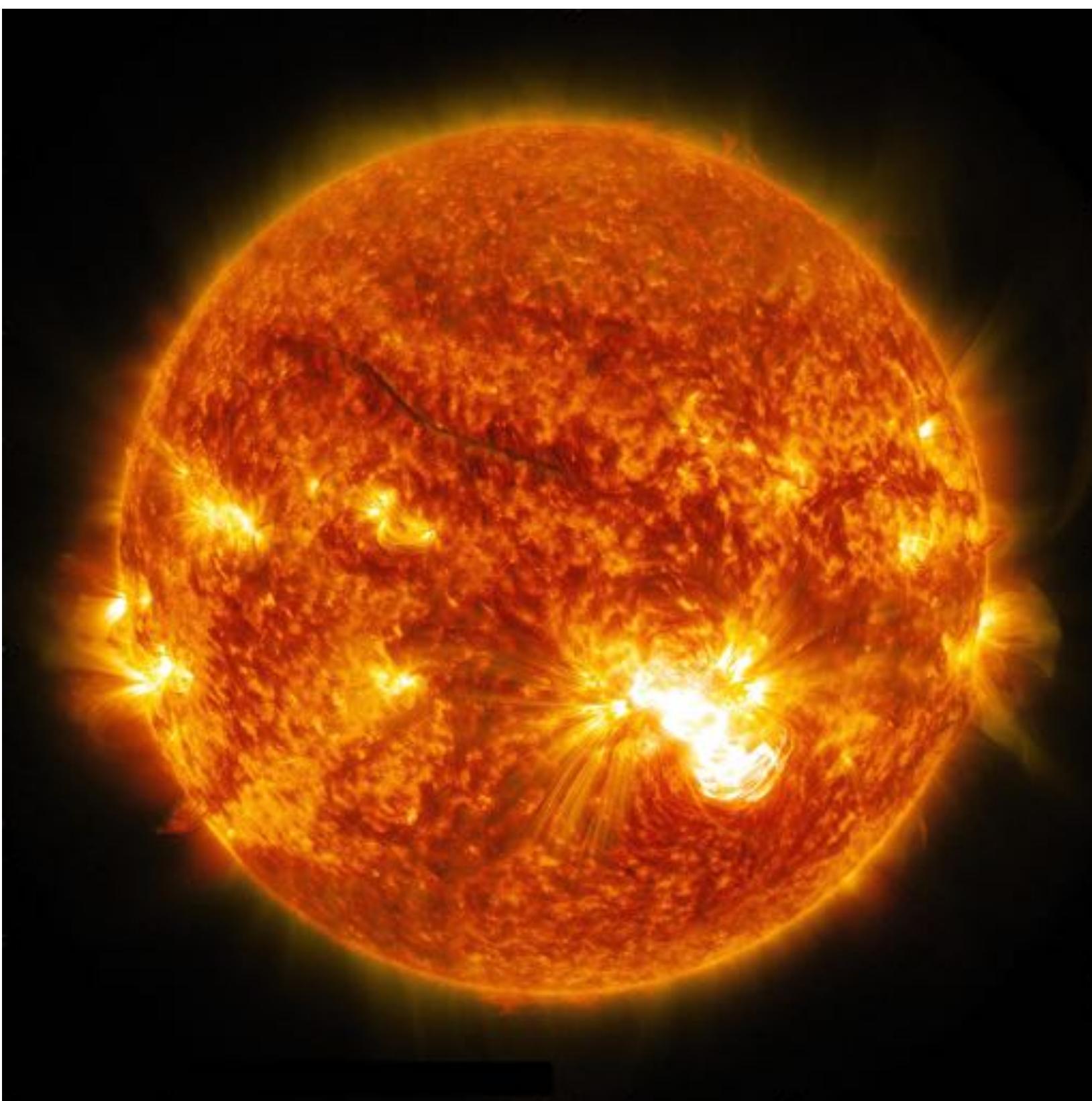
# Most light is not visible!

- Frequencies visible by human eyes are called “visible spectrum”
- These frequencies what we normally think of as “color”



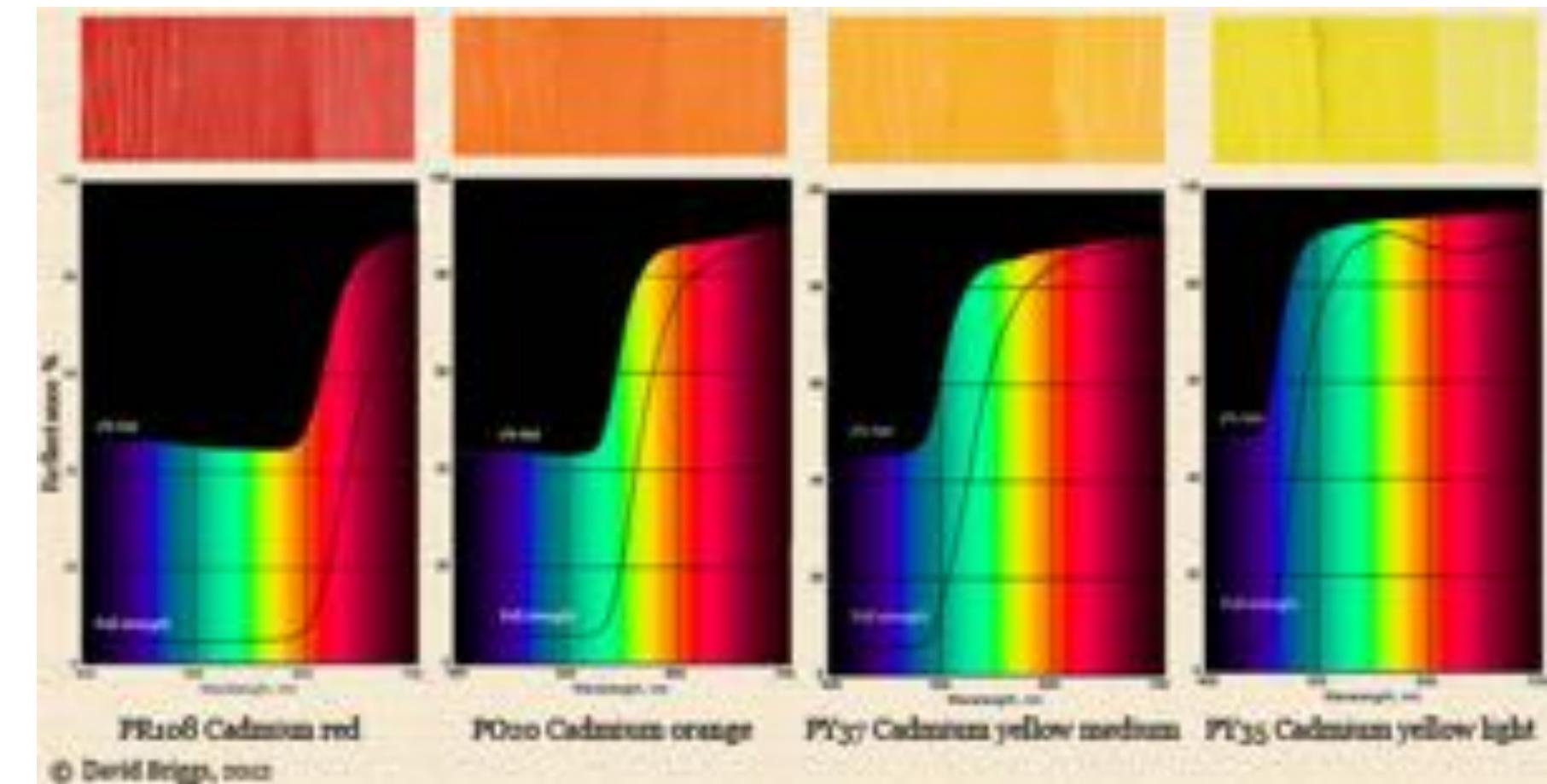
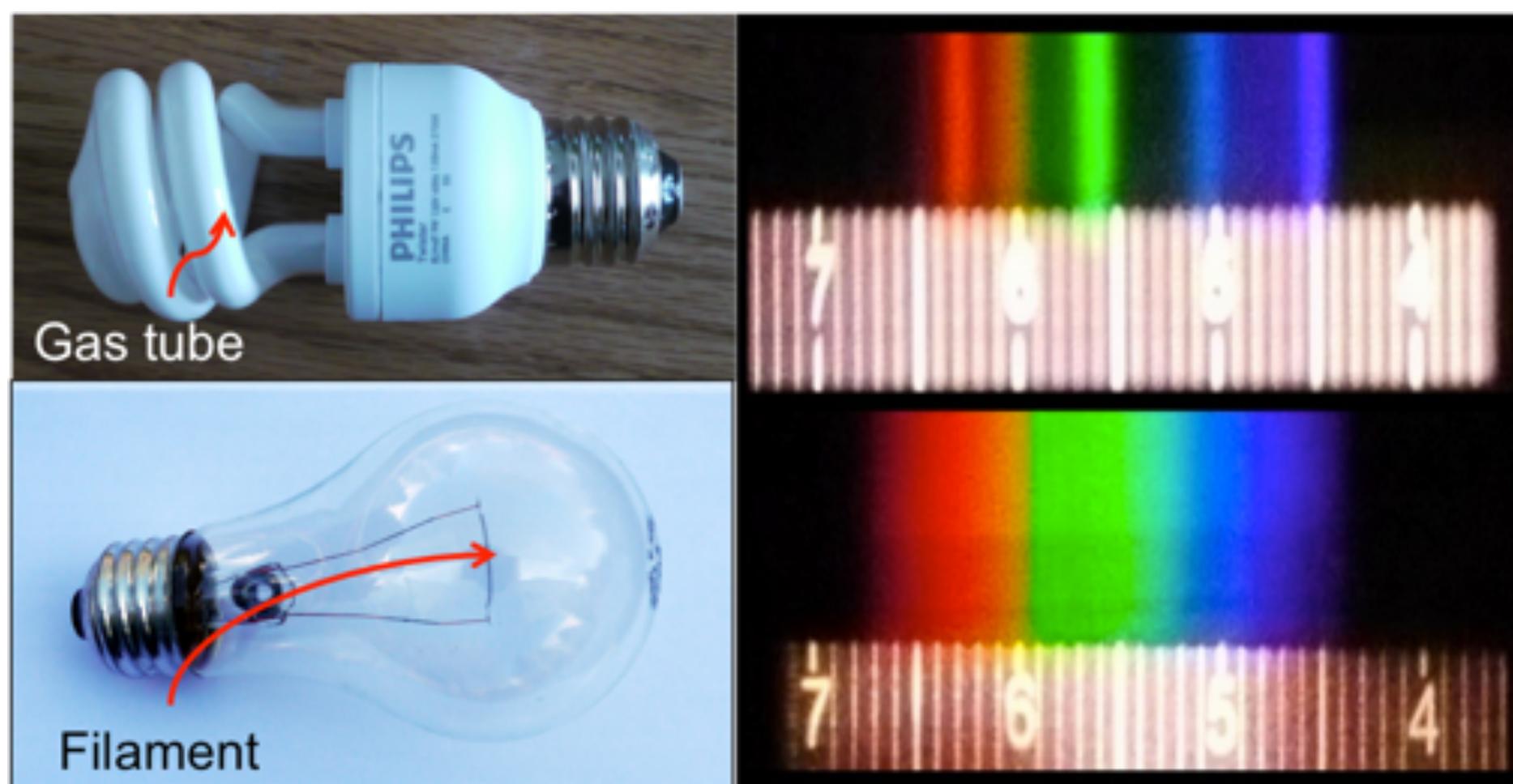
# Natural light is a mixture of frequencies

- “White” light is really a mixture of all (visible) frequencies
- E.g., the light from our sun



# Additive vs. Subtractive Models of Light

- Spectrum we just saw for the sun “emission spectrum”
  - How much light is produced (by heat, fusion, etc.)
  - Useful for, e.g., characterizing color of a lightbulb
- Another useful description: “absorption spectrum”
  - How much light is absorbed (e.g., turned into heat)
  - Useful for, e.g., characterizing color of paint, ink, etc.



# Emission Spectrum

Describes light intensity as a function of frequency

Below: spectrum of various common light sources:

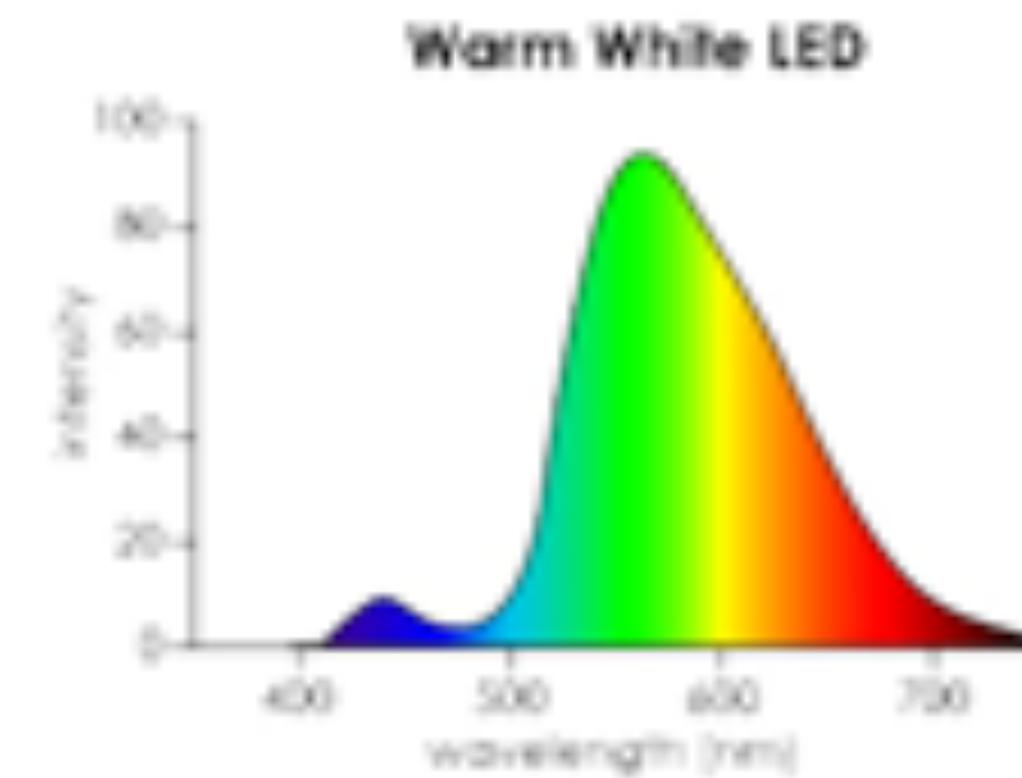
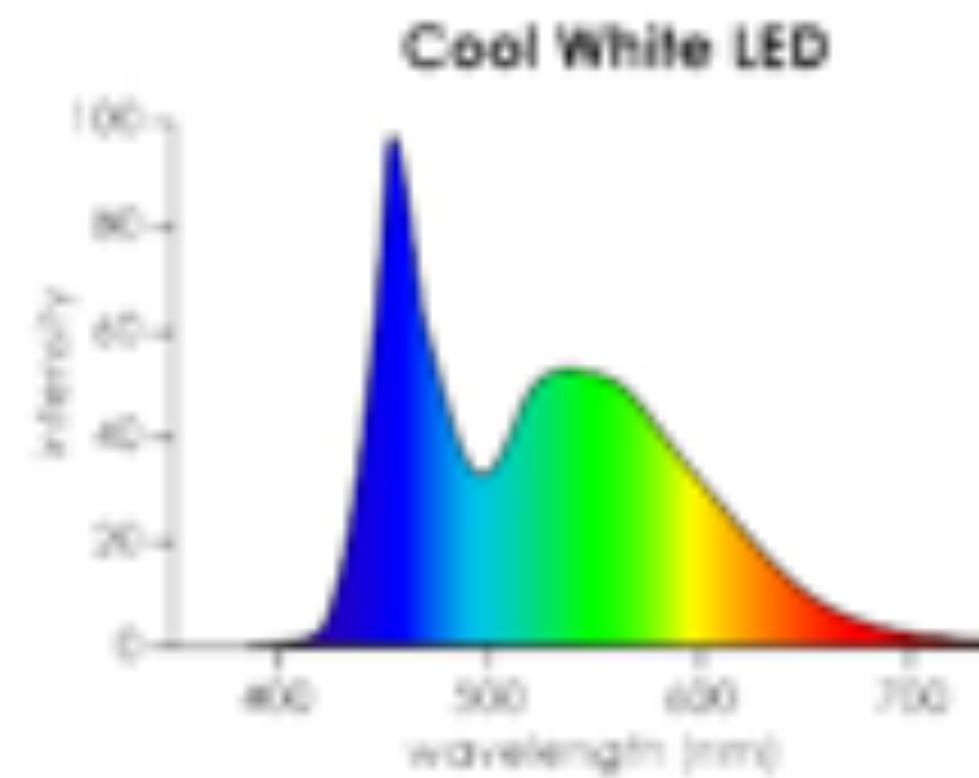
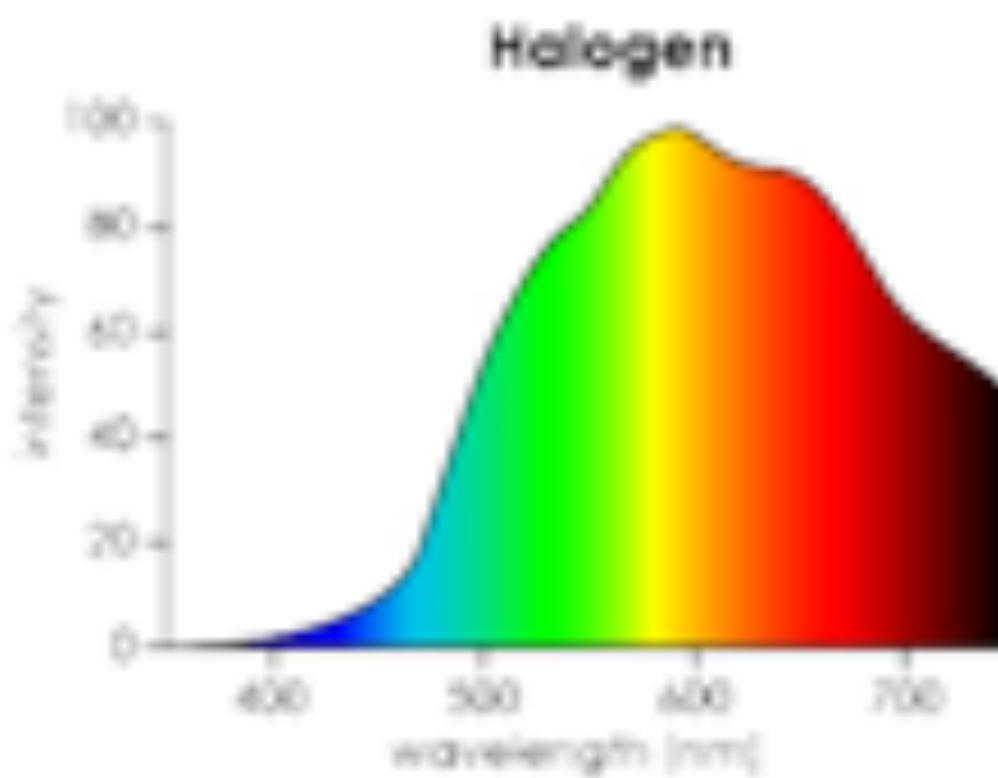
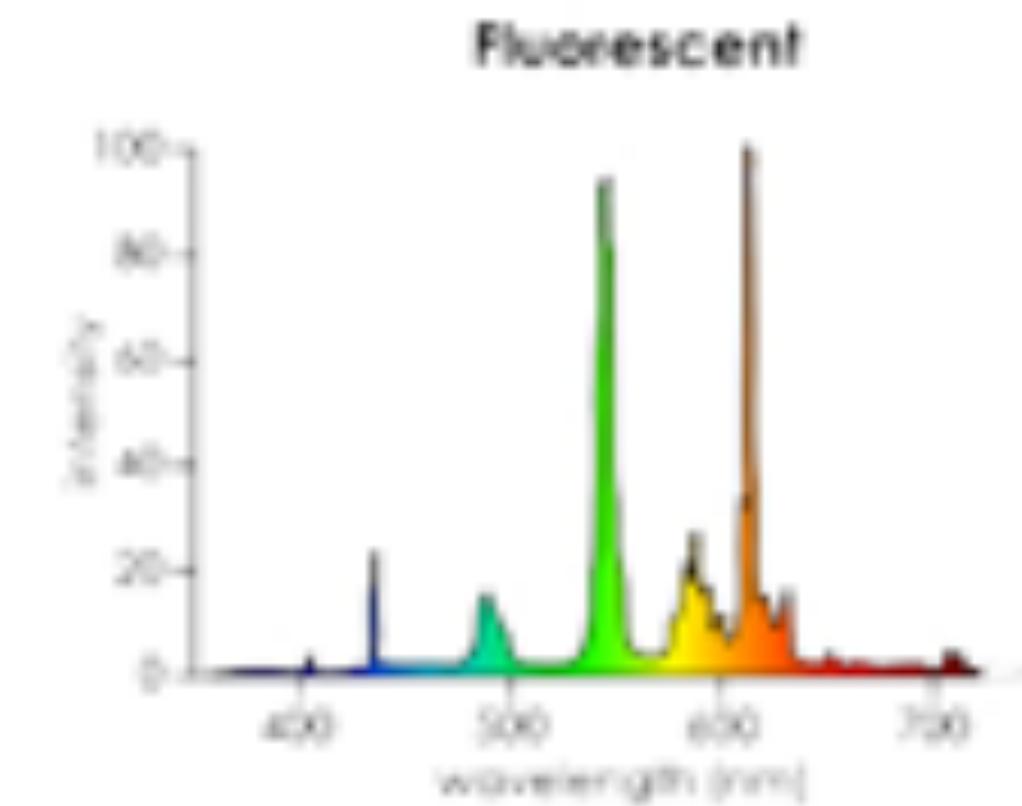
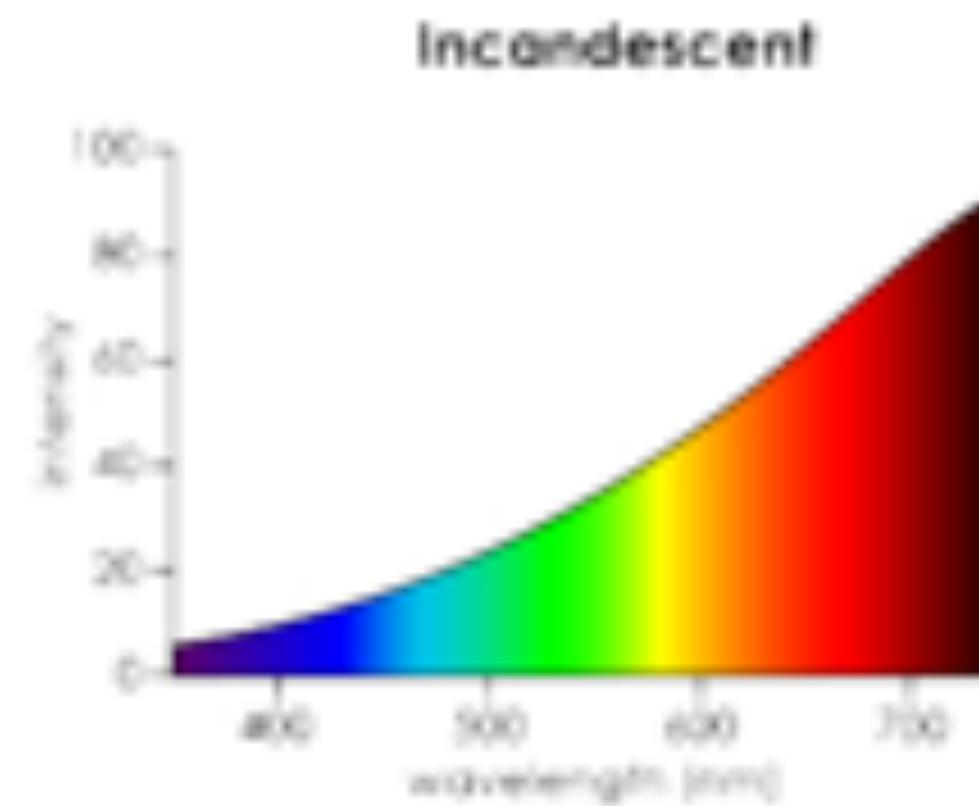
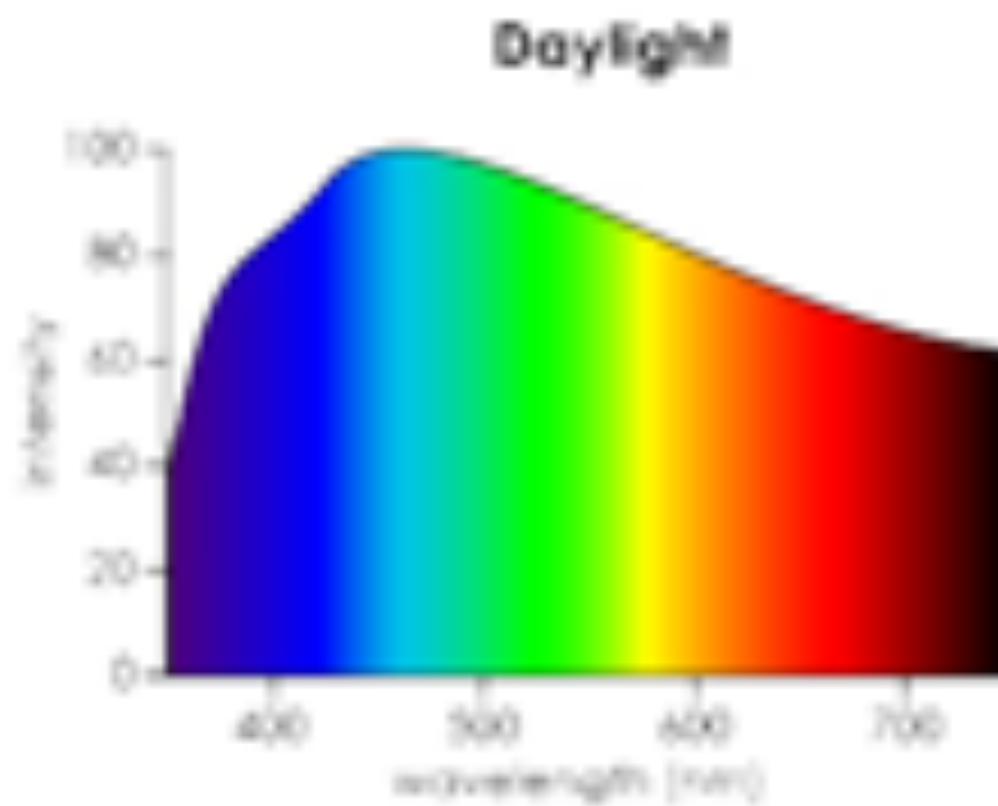


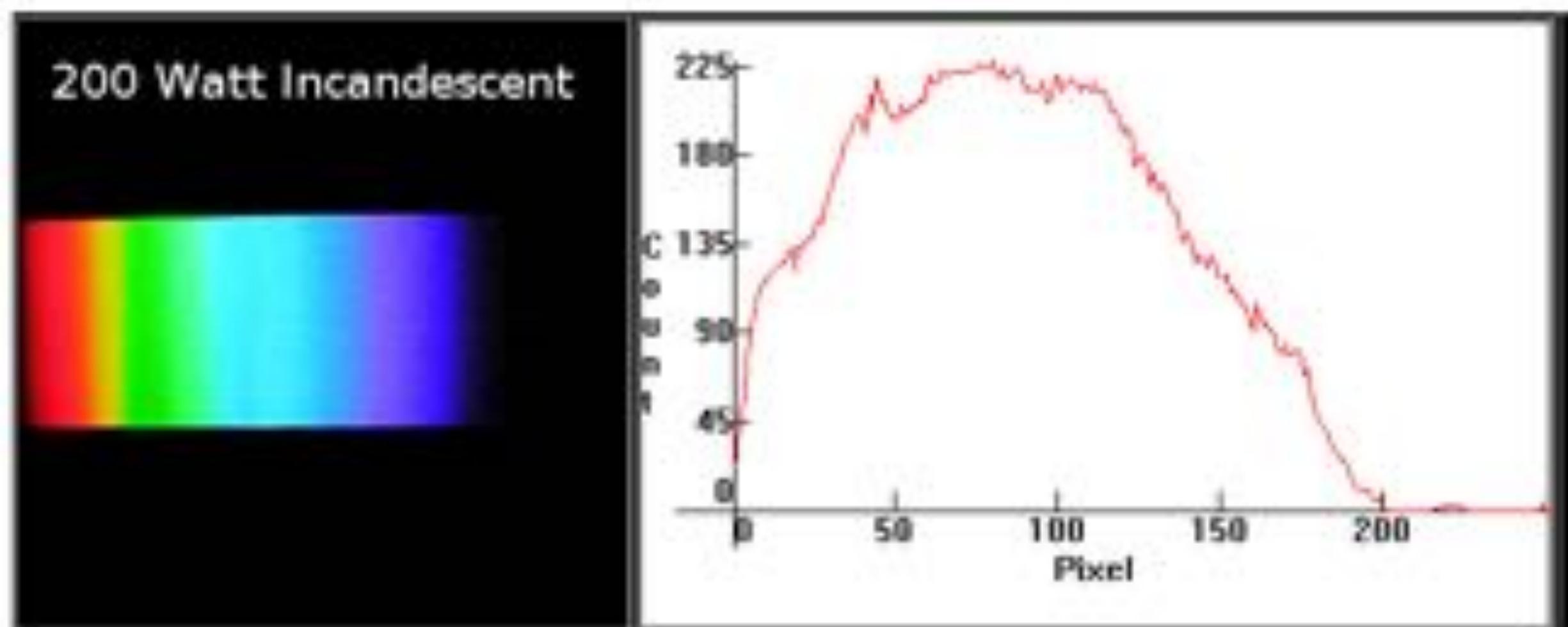
Figure credit:

# Emission Spectrum—Example

- Why so many different kinds of lightbulbs on the market?
- “Quality” of light:

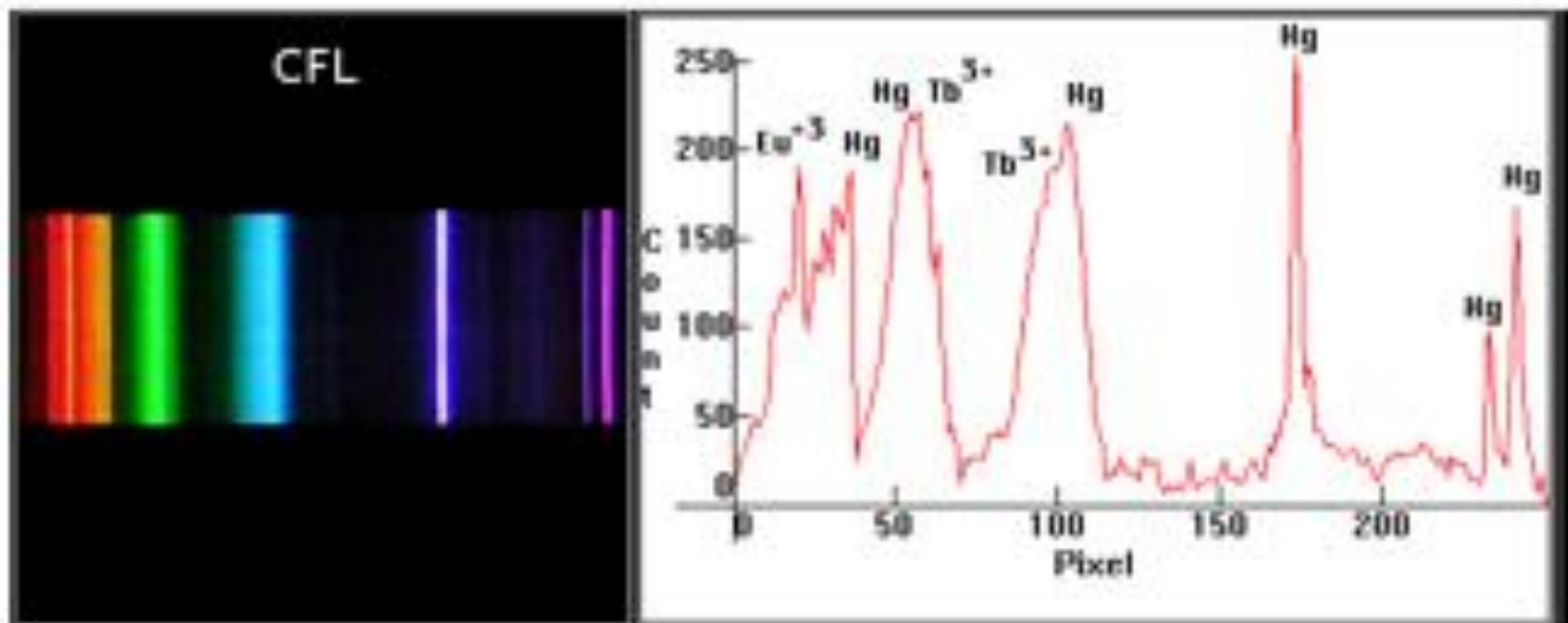
Incandescent:

- +more sun-like
- power-hungry



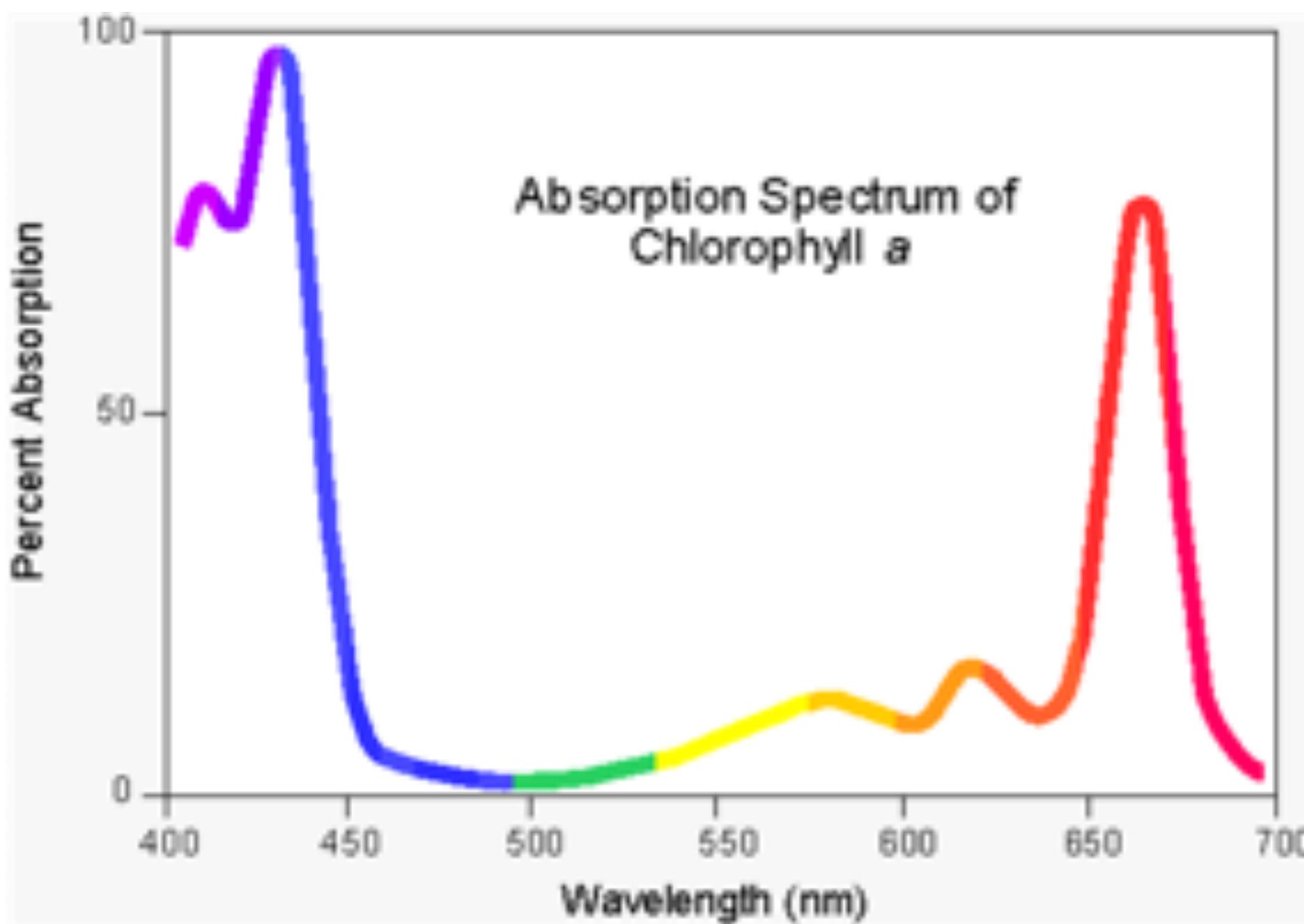
CFL:

- “choppy” spectrum
- +power efficient



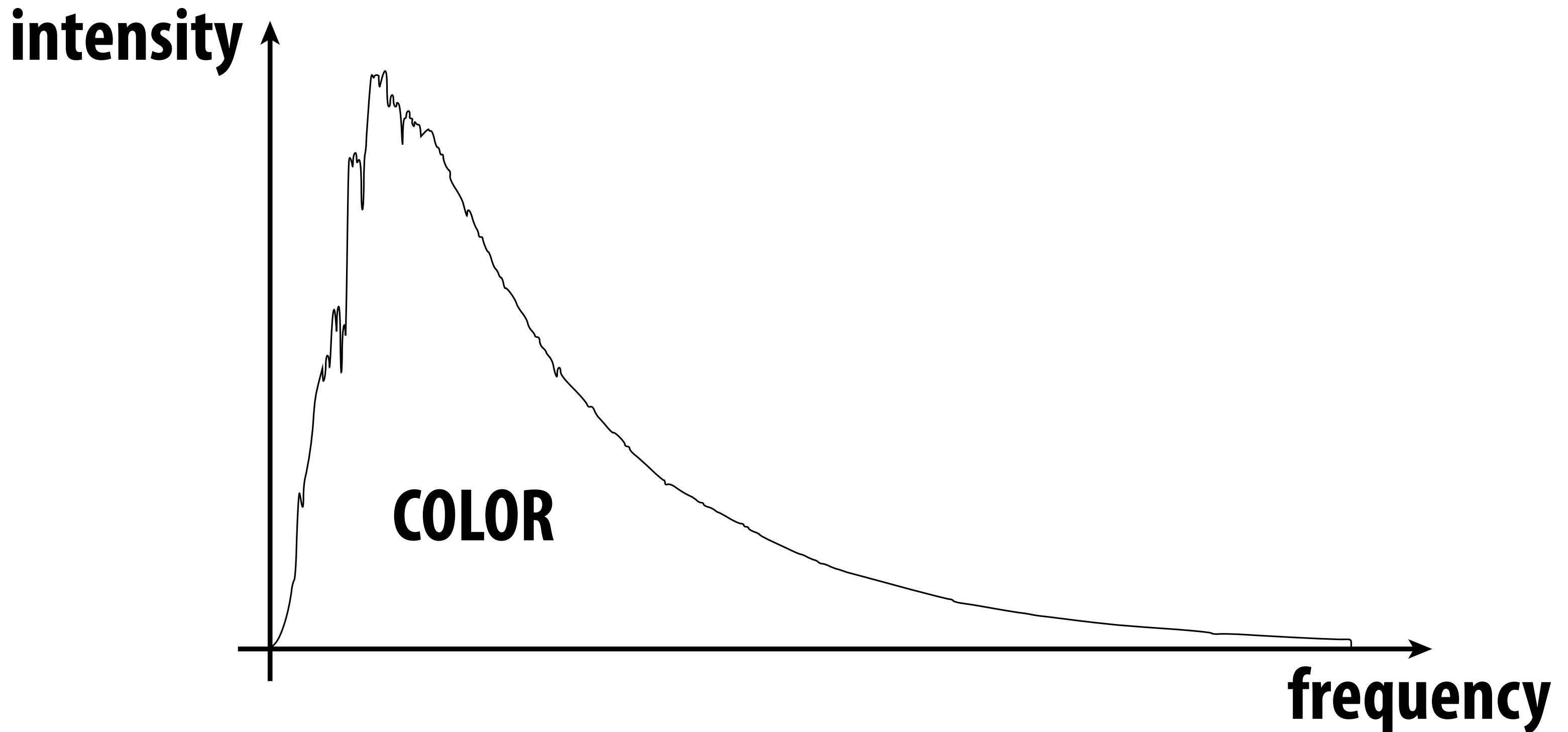
# Absorption Spectrum

- Emission spectrum is intensity as a function of frequency
- Absorption spectrum is fraction absorbed as function of frequency



Q: What color is an object with this absorption spectrum?

**This is the fundamental description of color:  
intensity or absorption as a function of frequency**



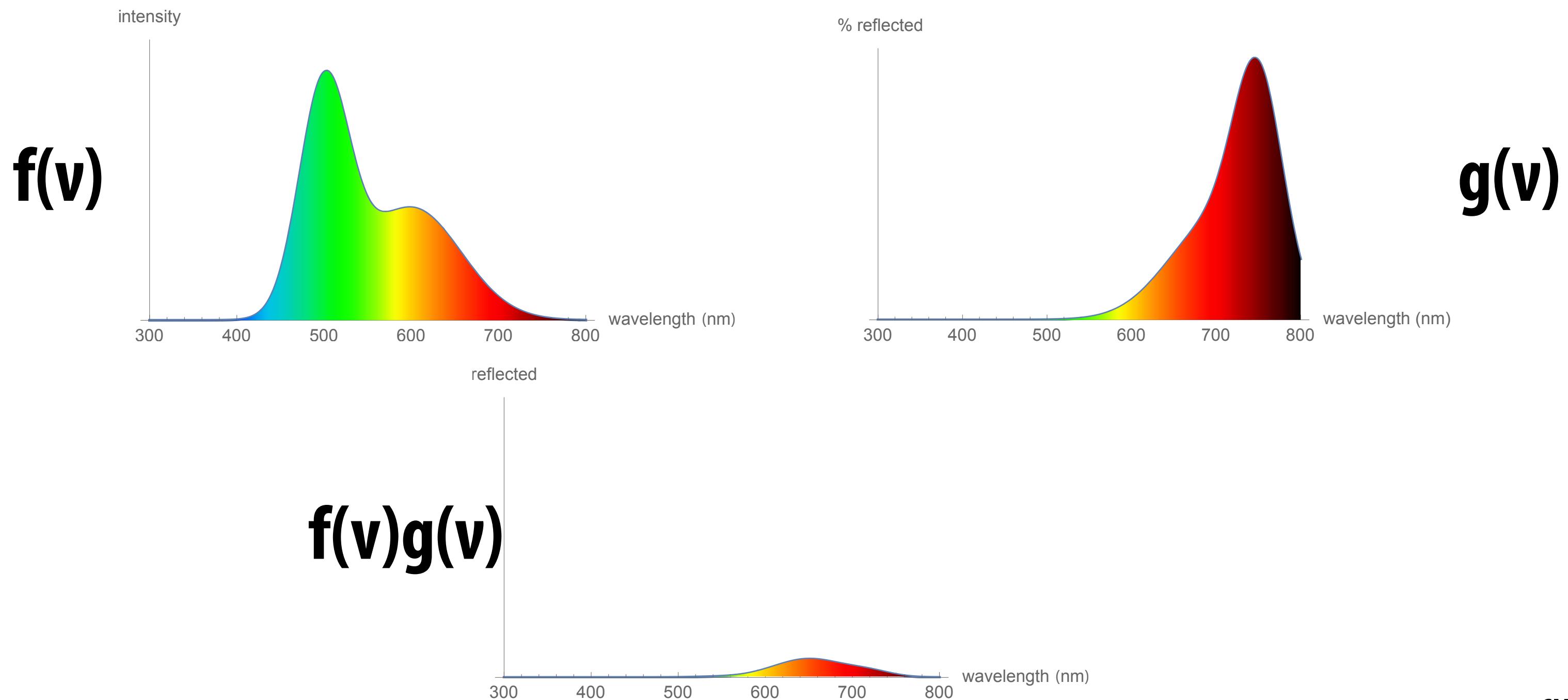
**Everything else is merely a convenient approximation!**

**If you remember to use spectral  
description as a starting point, the  
issues surrounding color theory/  
practice will make a lot more sense!**

If on the other hand you always think of color in terms of approximate digital encodings (RGB, CMYK) etc., there are certain phenomena you simply cannot explain/understand!

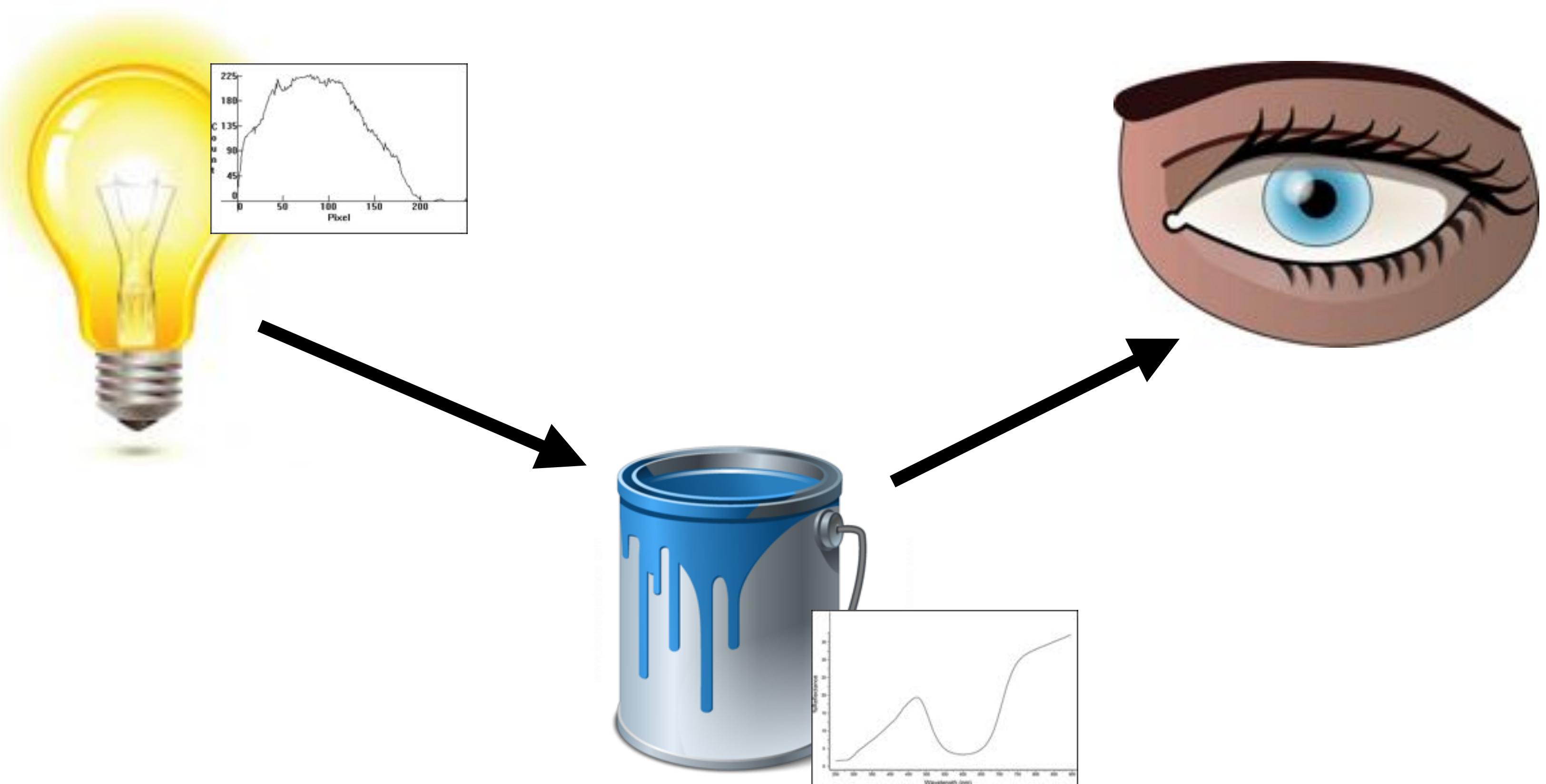
# Interaction of emission and reflection

- Toy model for what happens when light gets reflected
  - $\nu$ —frequency (Greek “nu”)
  - Light source has emission spectrum  $f(\nu)$
  - Surface has reflection spectrum  $g(\nu)$
  - Resulting intensity is the product  $f(\nu)g(\nu)$



# Color reproduction is hard!

- Color clearly starts to get complicated as we start combining emission and absorption/reflection (real-world challenge!)



(What color ink should we use to get the desired appearance?)

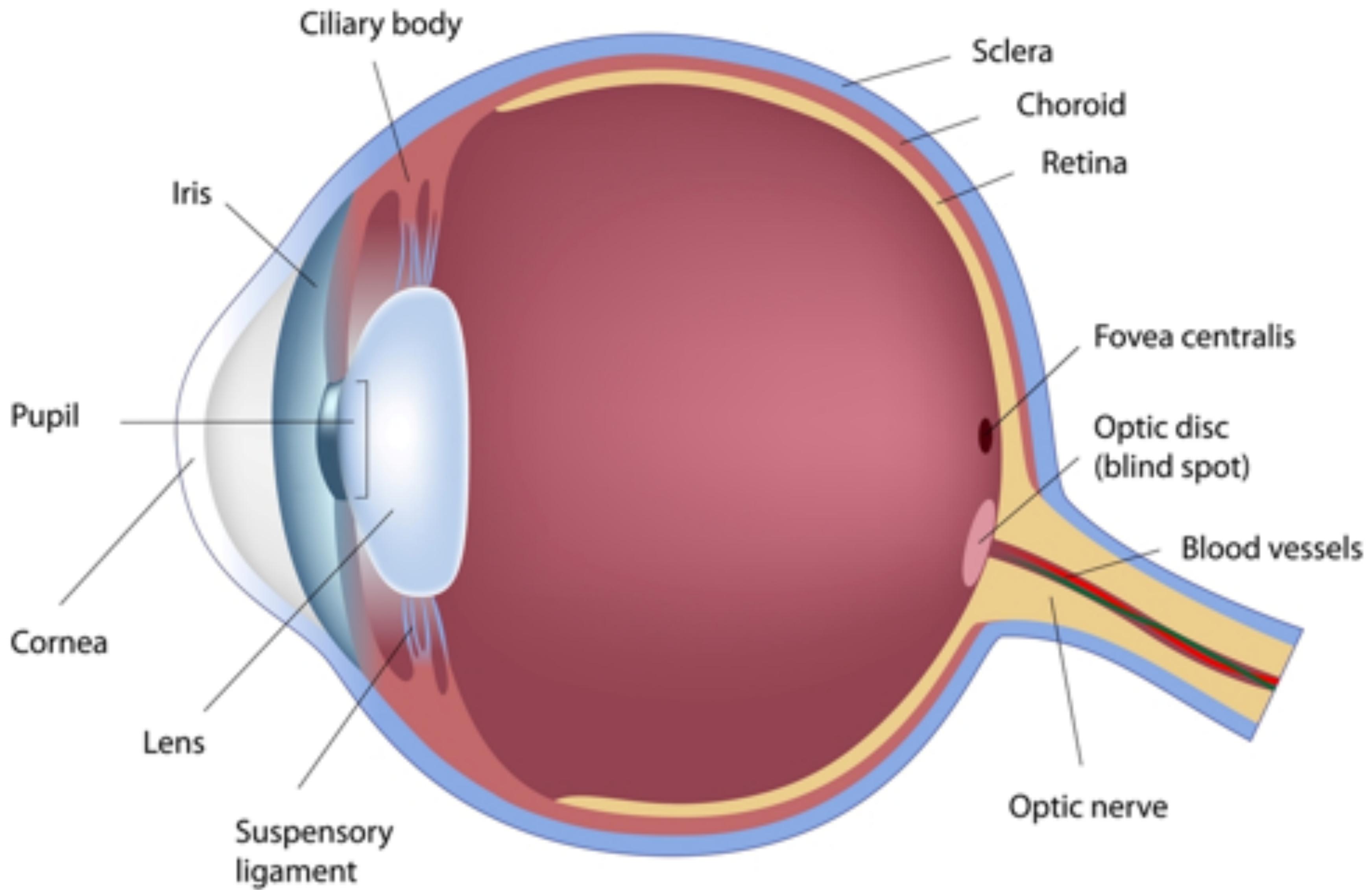
# ...And what about perception?

**Q: What color is this dress?**

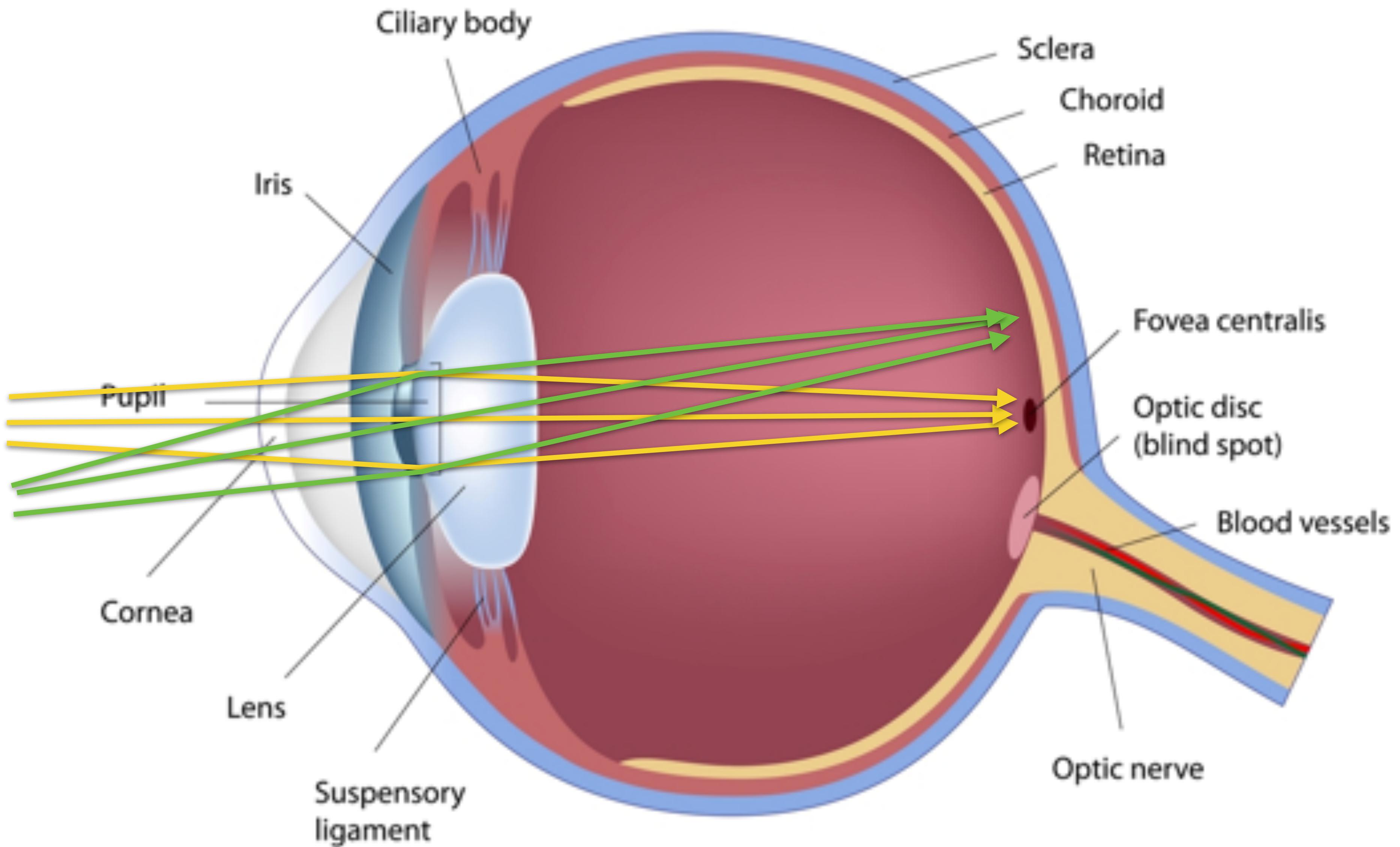


**How does electromagnetic radiation (with a given power distribution) end up being perceived by a human as a certain color?**

# The eye



# The eye (optics)



# Photosensor response (eye, camera, ...)

- Photosensor input: light

- Electromagnetic power distribution over wavelengths:  $\Phi(\lambda)$

- Photosensor output: a “response”... a number

- e.g., encoded in electrical signal

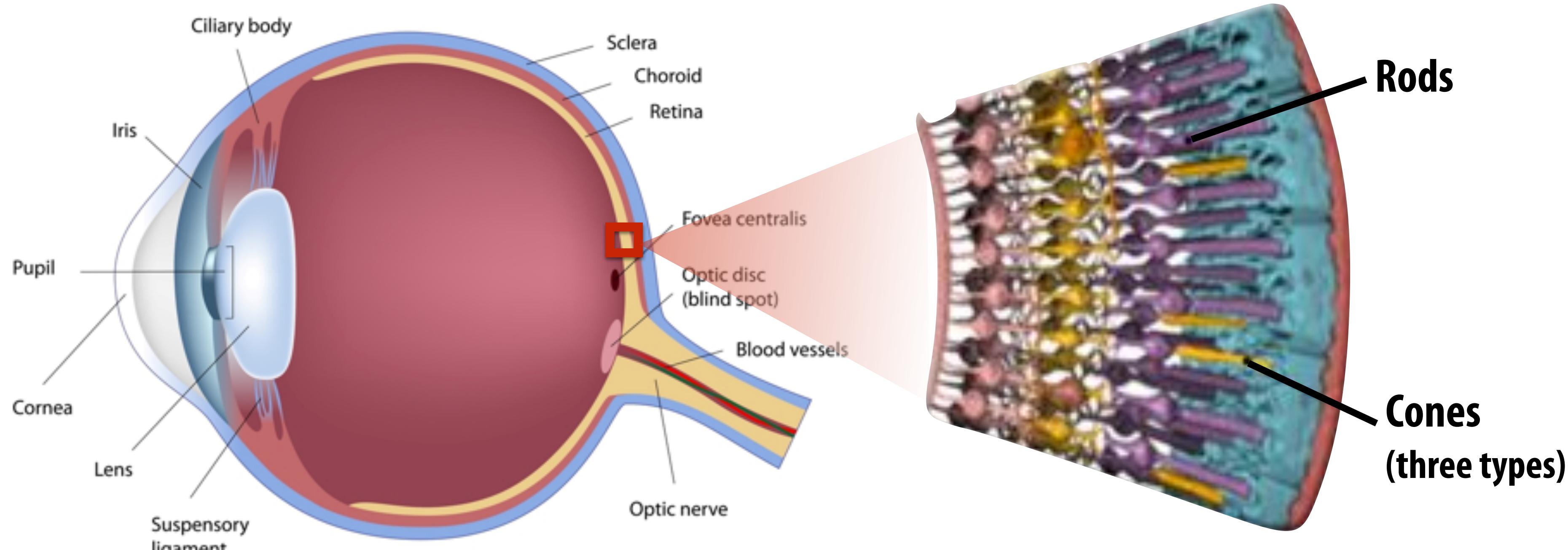
- Spectral response function:  $f(\lambda)$

- Sensitivity of sensor to light of a given wavelength
  - Greater  $f(\lambda)$  corresponds to more efficient sensor (when  $f(\lambda)$  is large, a small amount of light at wavelength  $\lambda$  will trigger a large sensor response)

- Total response of photosensor:

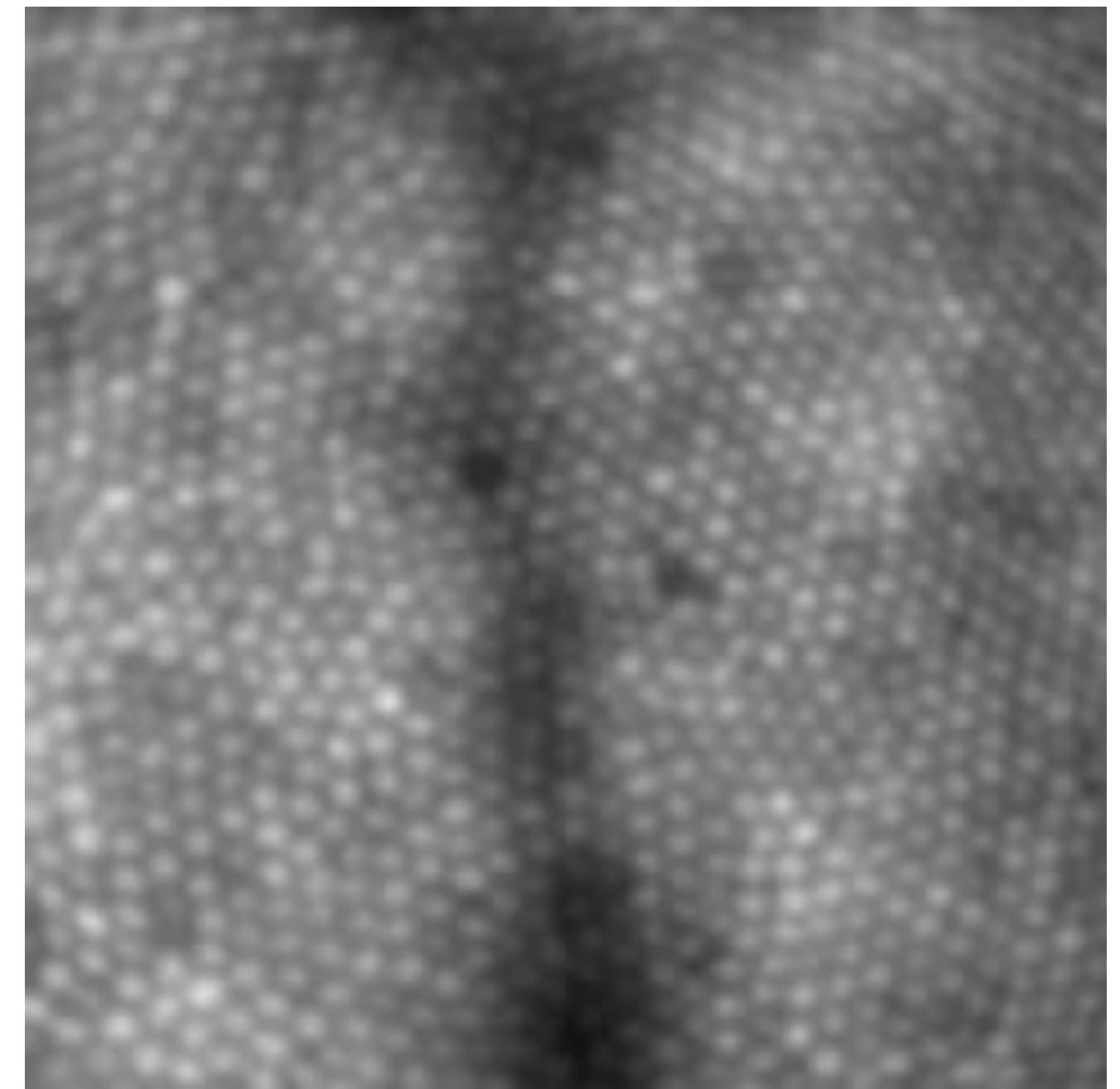
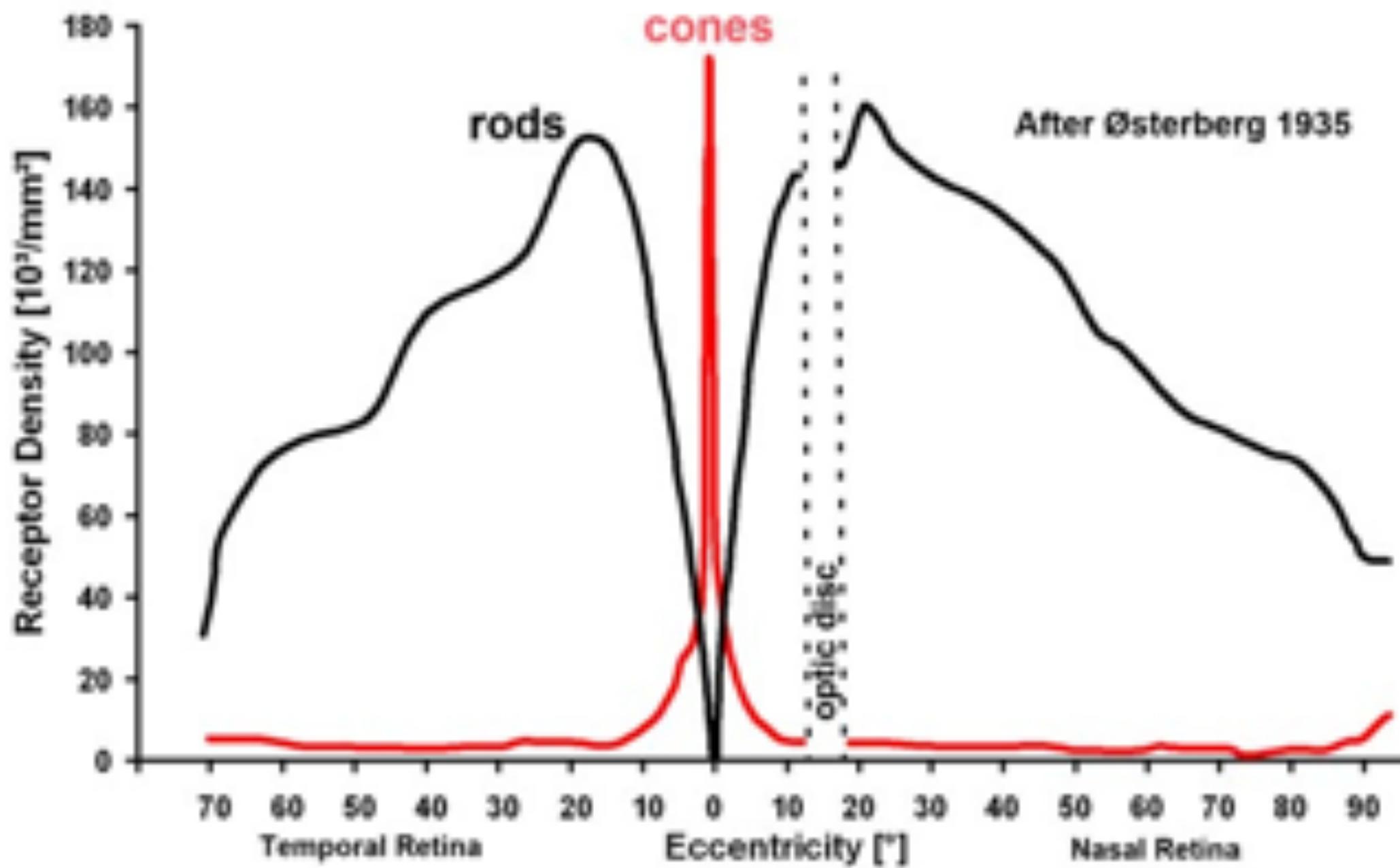
$$R = \int_{\lambda} \Phi(\lambda) f(\lambda) d\lambda$$

# The eye's photoreceptor cells: rods & cones



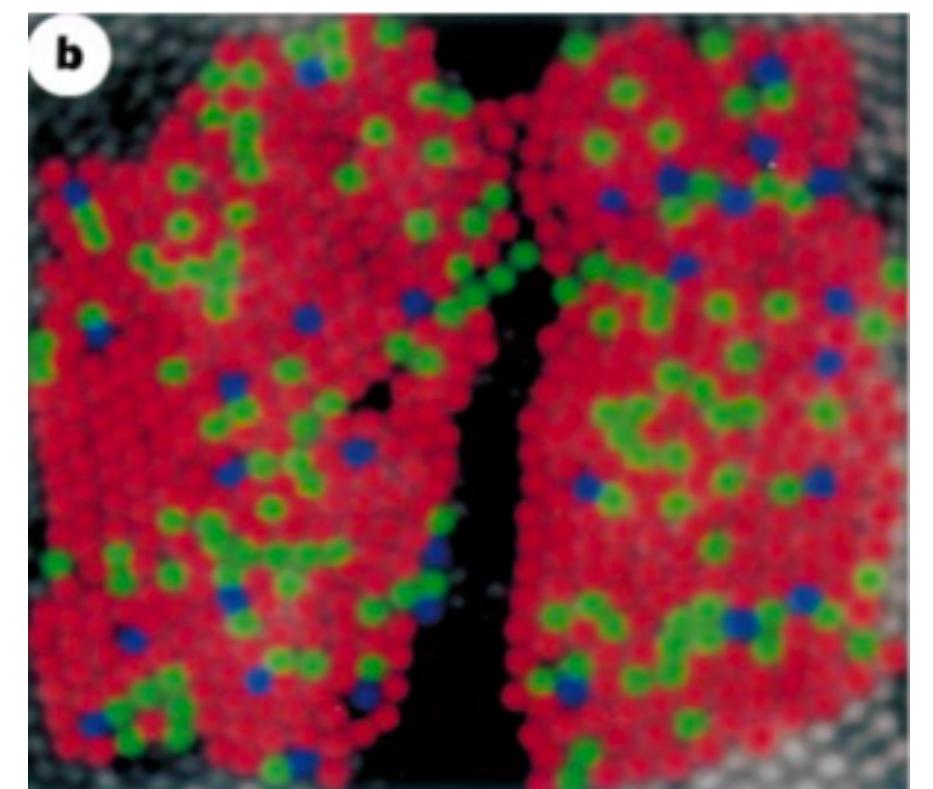
- Rods are primary receptors under dark viewing conditions (scotopic conditions)
  - Approx. 120 million rods in human eye
- Cones are primary receptors under high-light viewing conditions (photopic conditions, e.g., daylight)
  - Approx. 6-7 million cones in the human eye
  - Each of the three types of cone feature a different spectral response. This will be critical to color vision (much more on this in the coming slides)

# Density of rods and cones in the retina



[Roorda 1999]

- Highest density of cones is in fovea  
(best color vision at center of where human is looking)
- Note “blind spot” due to optic nerve



# ACTIVITY: Rods vs. Cones

- Grab someone and try it at home!
  - Have them hold up colored markers in peripheral vision
  - All you have to do is say what color it is (easy!)



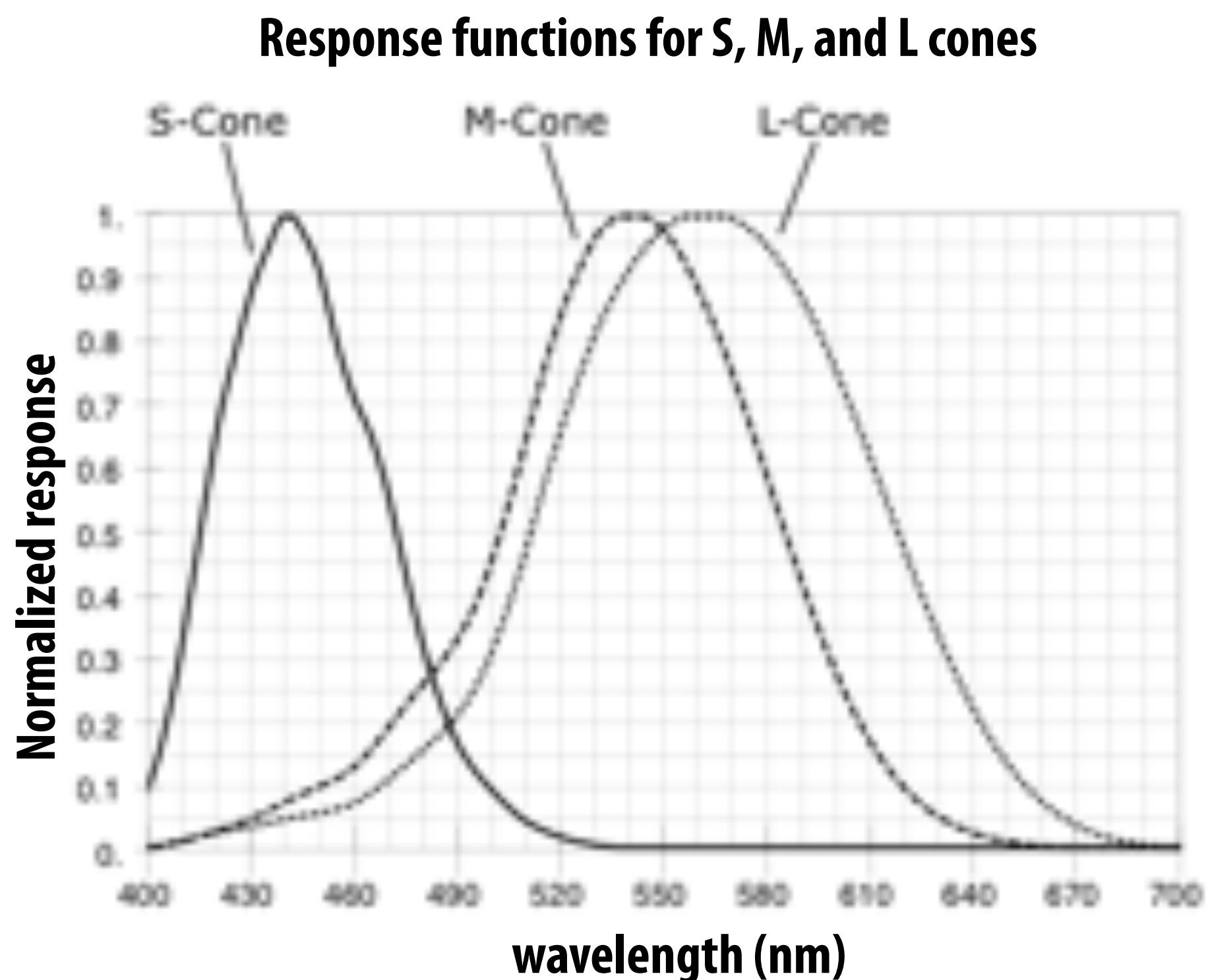
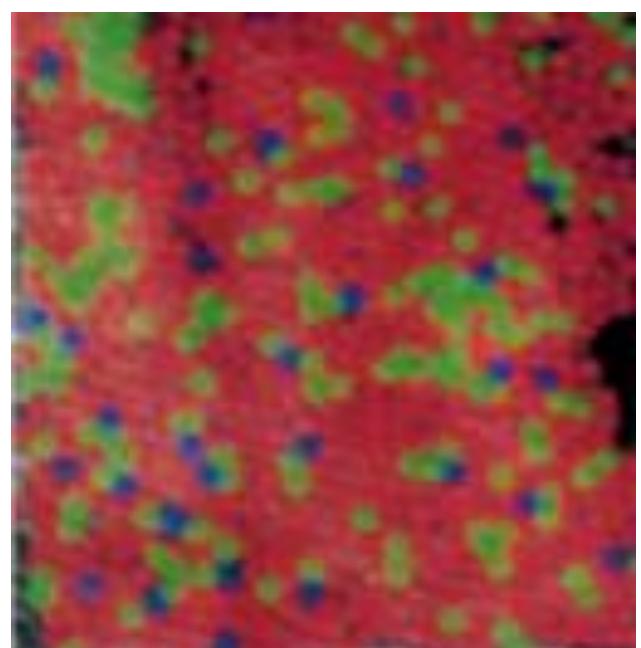
# Spectral response of cones

Three types of cones: S, M, and L cones (corresponding to peak response at short, medium, and long wavelengths)

$$S = \int_{\lambda} \Phi(\lambda) S(\lambda) d\lambda$$

$$M = \int_{\lambda} \Phi(\lambda) M(\lambda) d\lambda$$

$$L = \int_{\lambda} \Phi(\lambda) L(\lambda) d\lambda$$

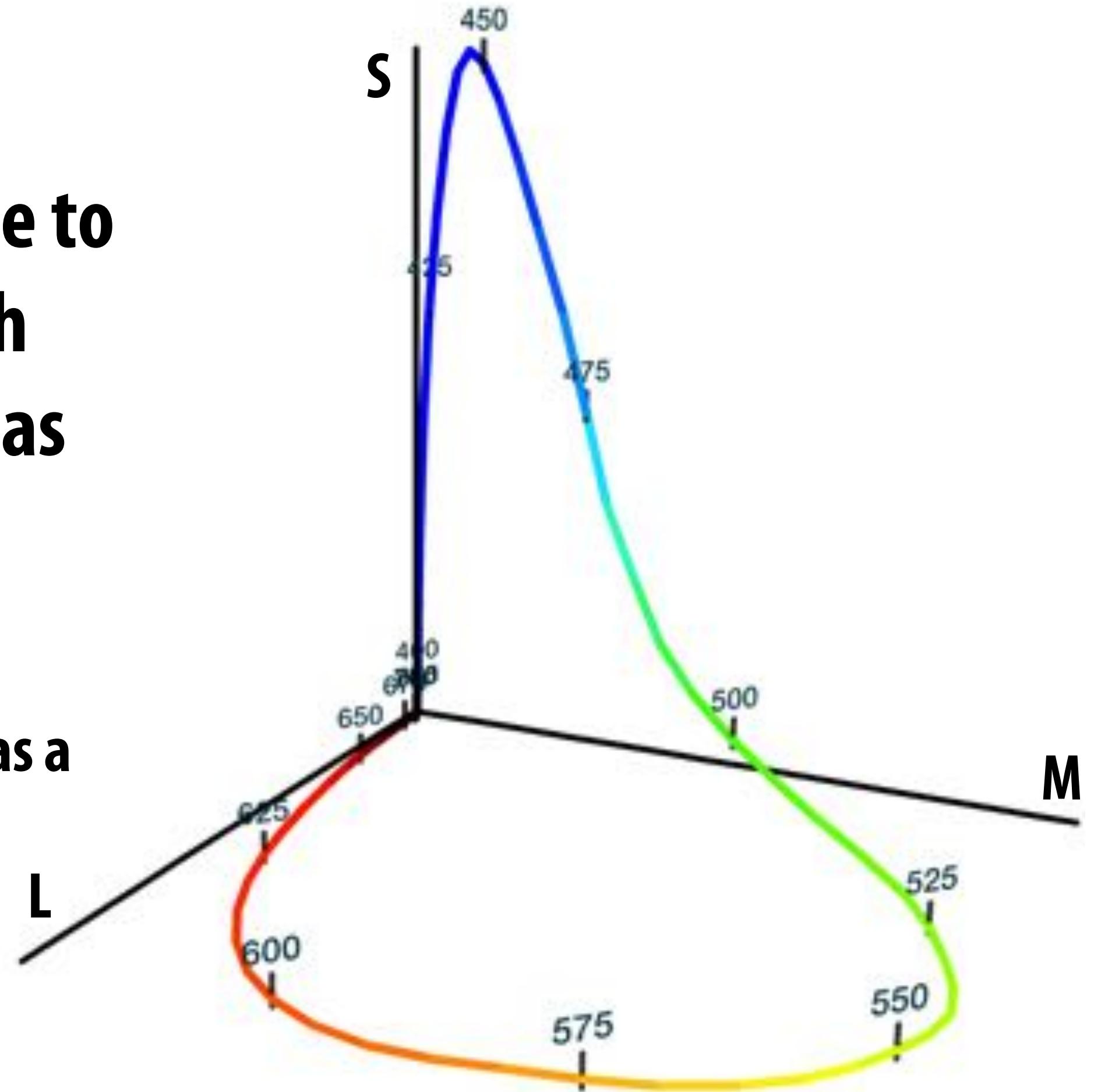


Uneven distribution of cone types in eye  
~64% of cones are L cones, ~32% M cones

# Response of S,M,L cones to monochromatic light

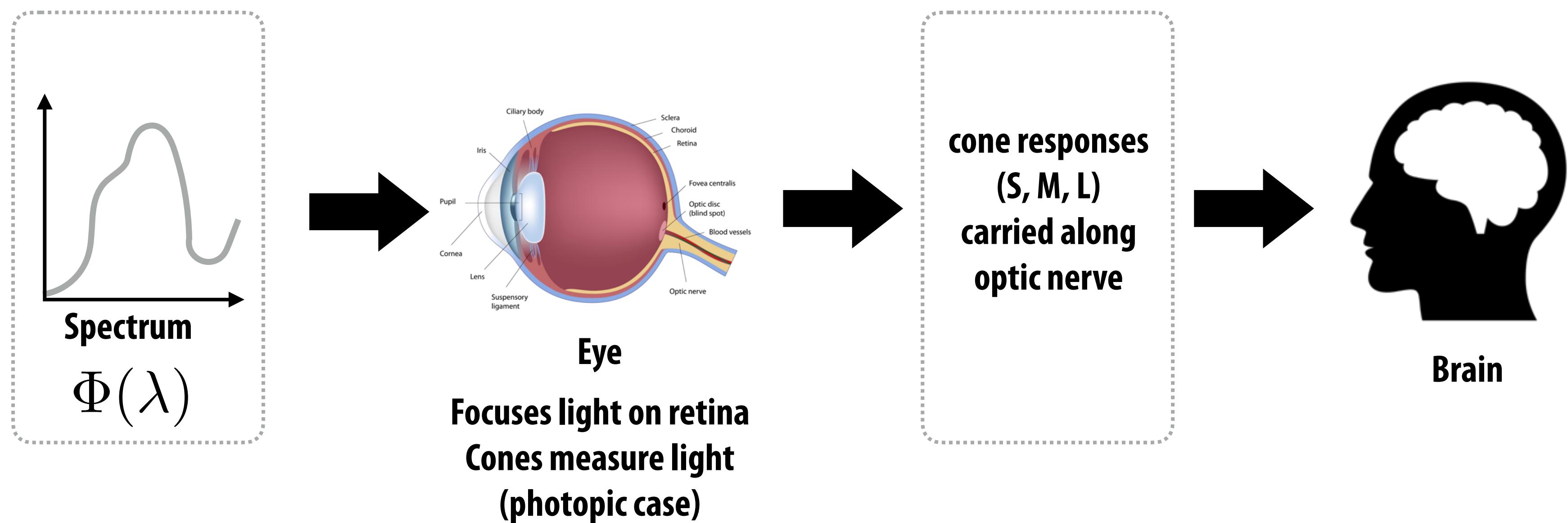
Figure visualizes cone's response to monochromatic light (light with energy in a single wavelength) as points in 3D space

(plots value of S, M, L response functions as a point in 3D space)



# The human visual system

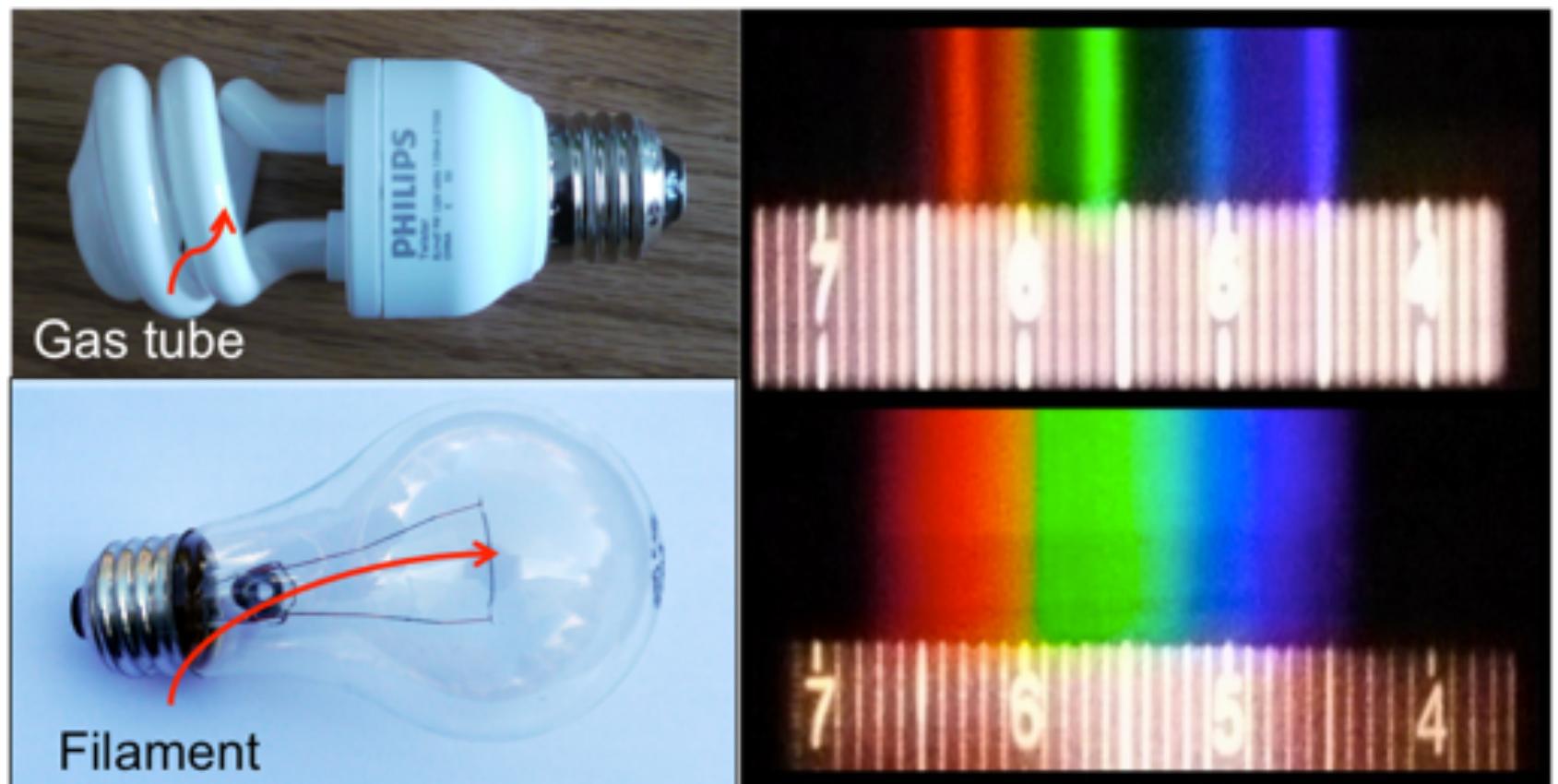
- Human eye does not directly measure the spectrum of incoming light
  - i.e., the brain does not receive “a spectrum” from the eye
- The eye measures three response values = (S, M, L). The result of integrating the incoming spectrum against response functions of S, M, L-cones



**Q: Is it possible for two functions  
to integrate to the same value?**

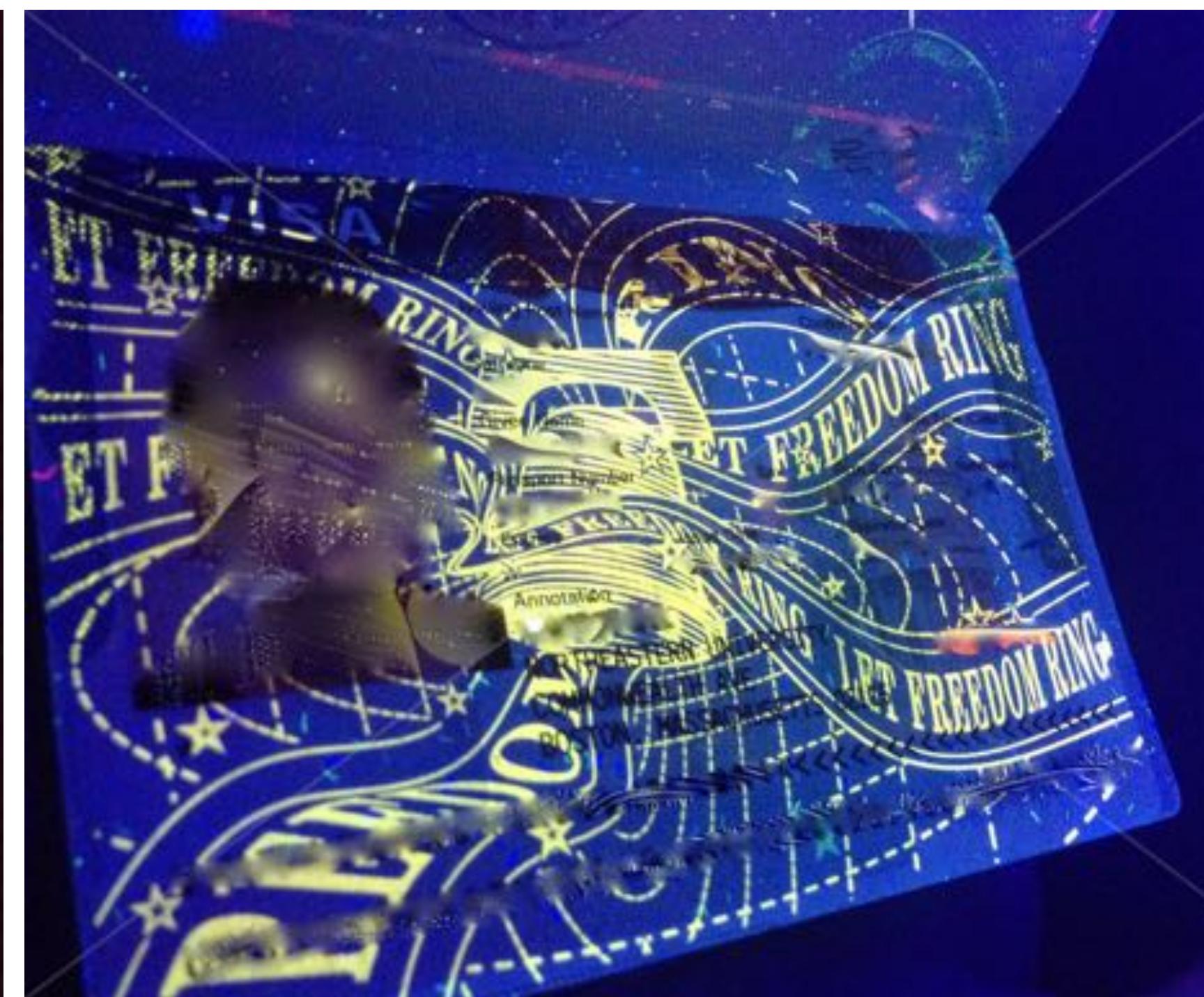
# Metamers

- Metamers = two different spectra that integrate to the same (S,M,L) response!
- The fact that metameters exist is critical to color reproduction: we don't have to reproduce the exact same spectrum that was present in a real world scene in order to reproduce the perceived color on a monitor (or piece of paper, or paint on a wall)
- ...On the other hand, combination of light & paint could still cause trouble—different objects appearing “wrong” under different lighting conditions.



# Example: Counterfeit Detection

- Many countries print currency, passports, etc., with special inks that yield different appearance under UV light:

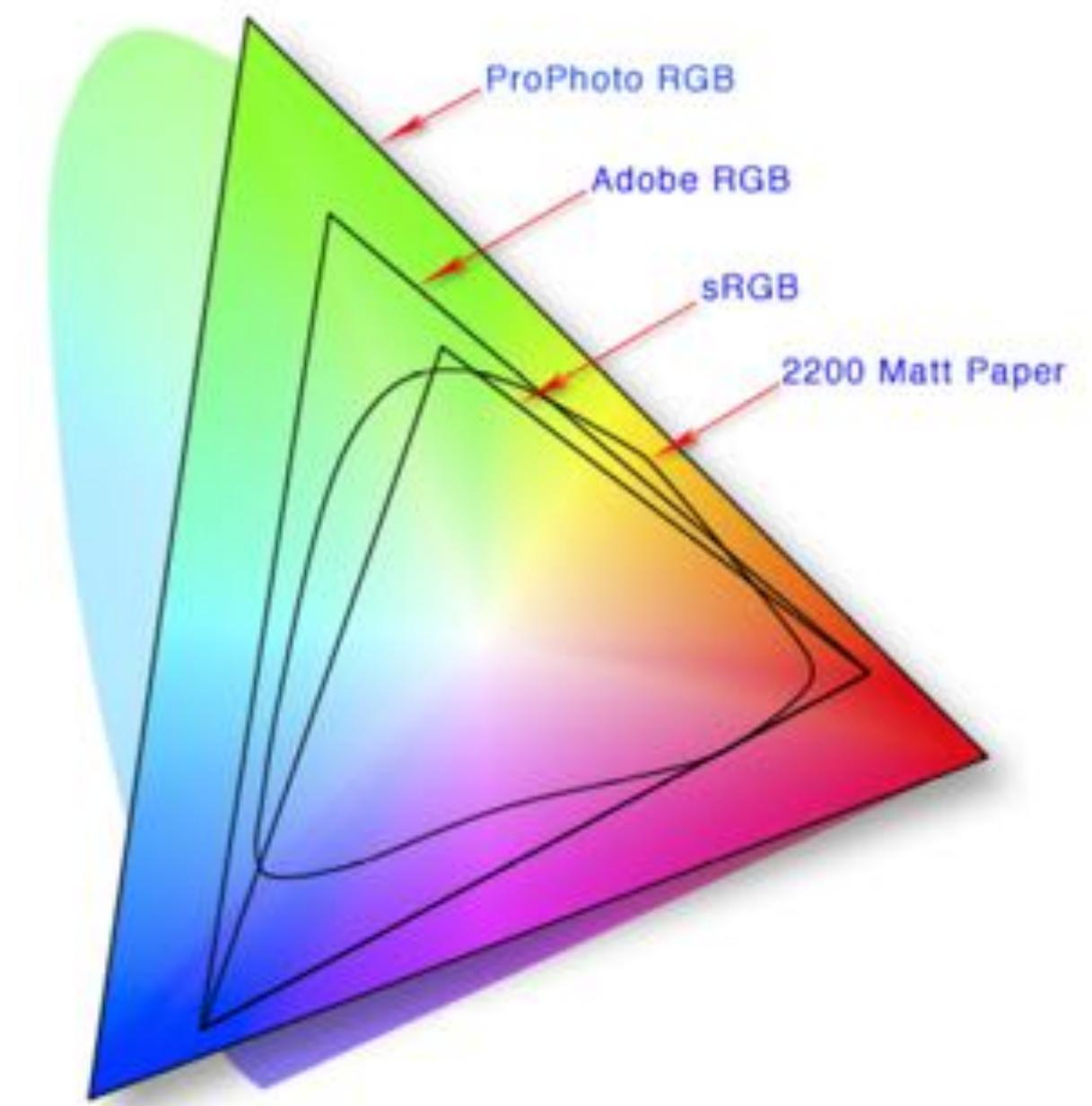


**Ok, so color can get pretty complicated!**

**How do we encode it in a simple(r) way?**

# Color Spaces and Color Models

- Many ways to specify a color
  - storage
  - convenience
- In general, specify a color from some color space using a color model
- Color space is like artist's palette: full range of colors we can choose from
- Color model is the way a particular color in a color space is specified:
  - artist's palette: "yellow ochre"
  - RGB color model: 204, 119, 34



# Additive vs. Subtractive Color Models

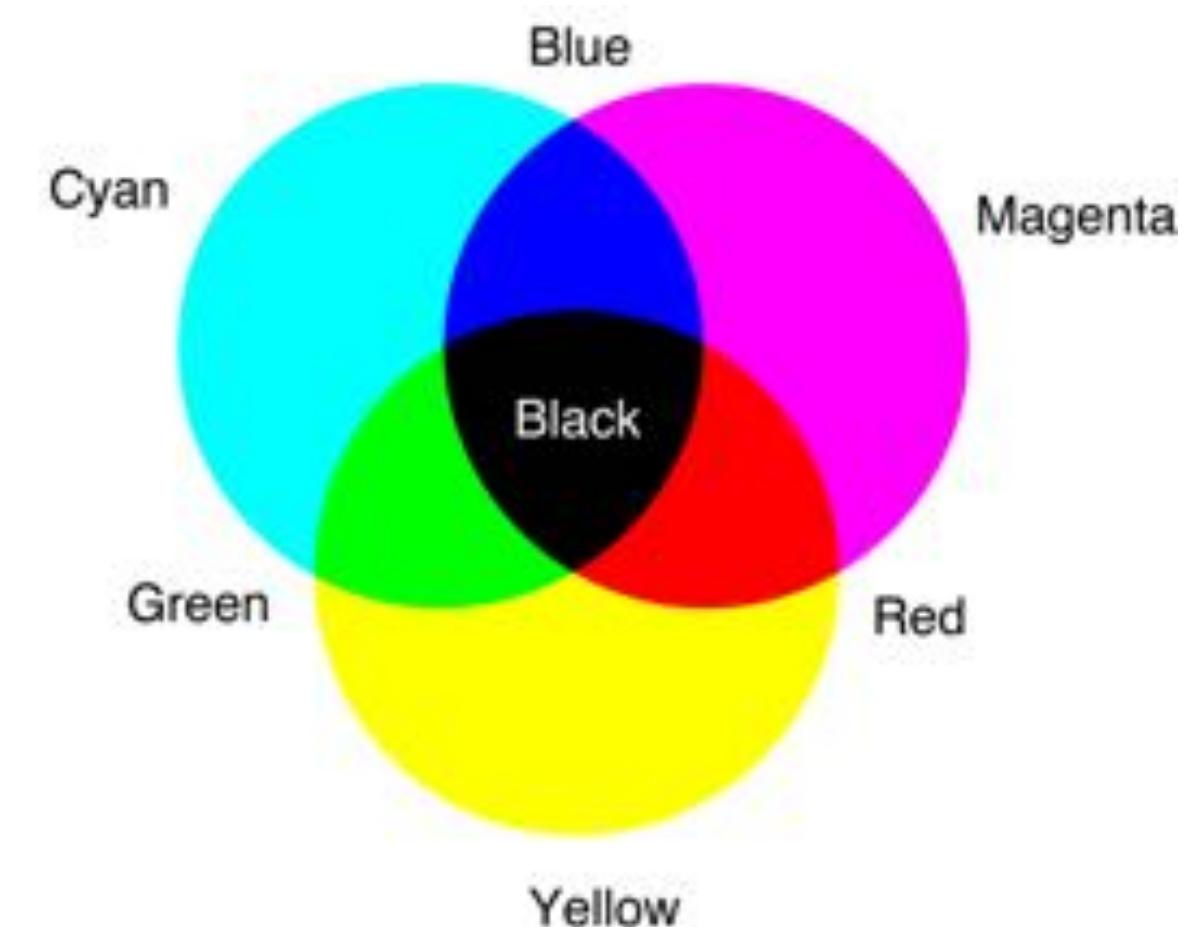
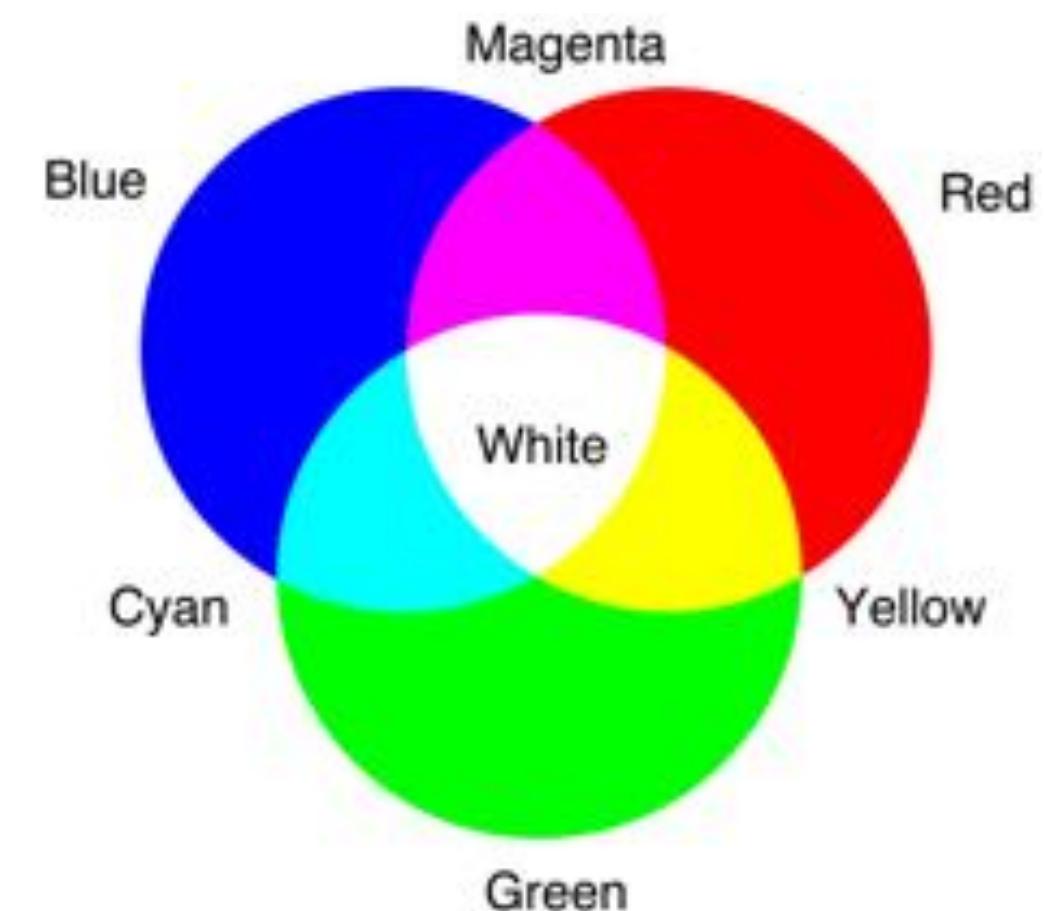
- Just like we had emission & absorption spectra, we have additive and subtractive\* color models

- Additive

- Used for, e.g., combining colored lights
  - Prototypical example: RGB

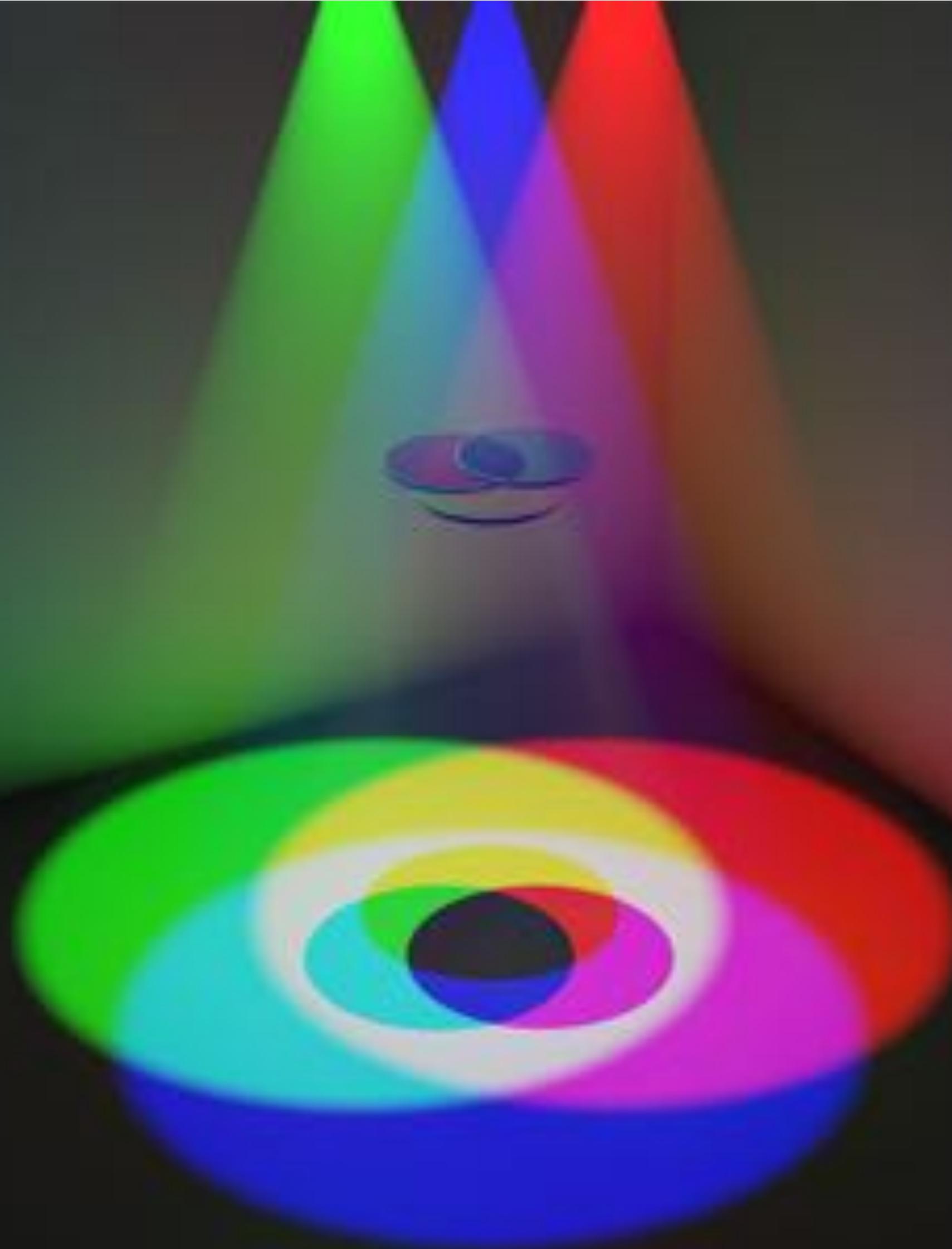
- Subtractive

- Used for, e.g., combining paint colors
  - Prototypical example: CMYK



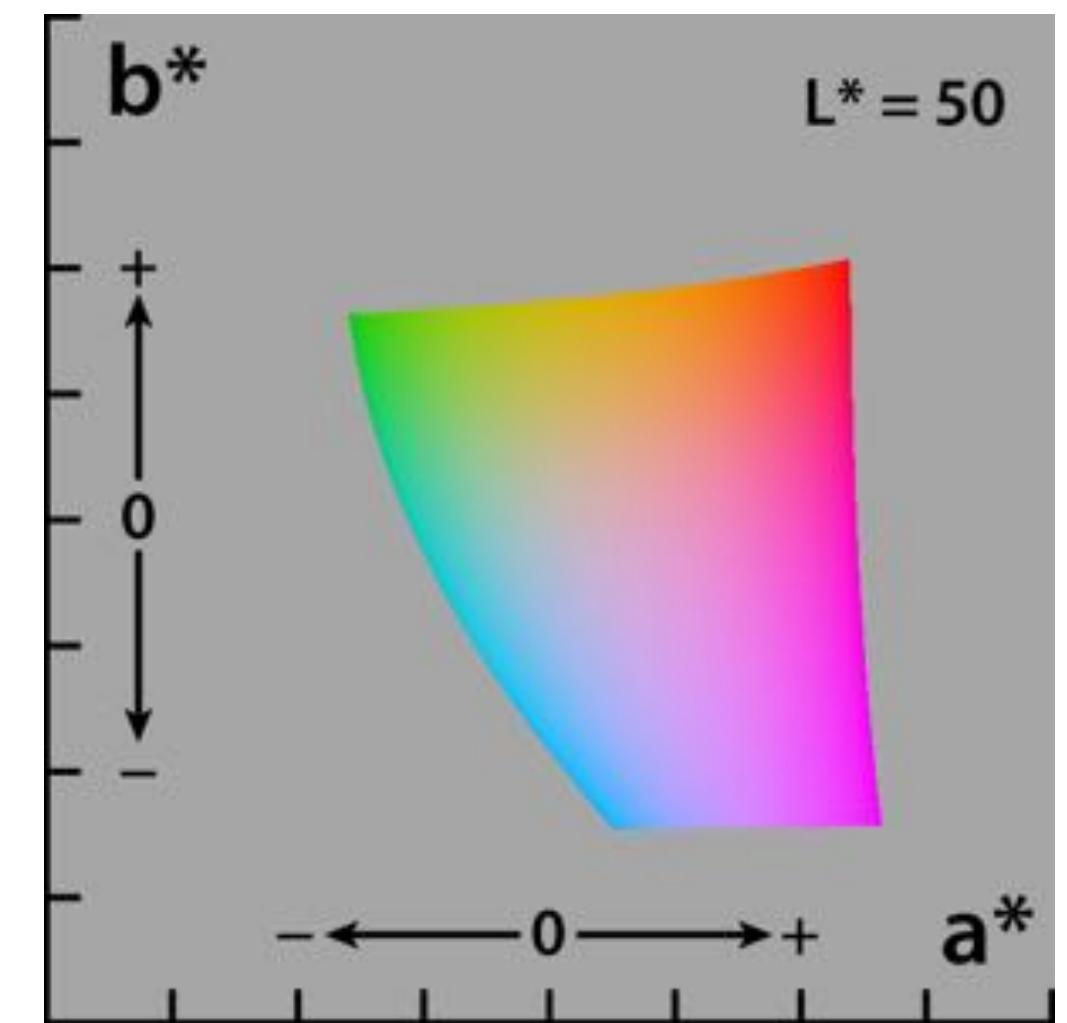
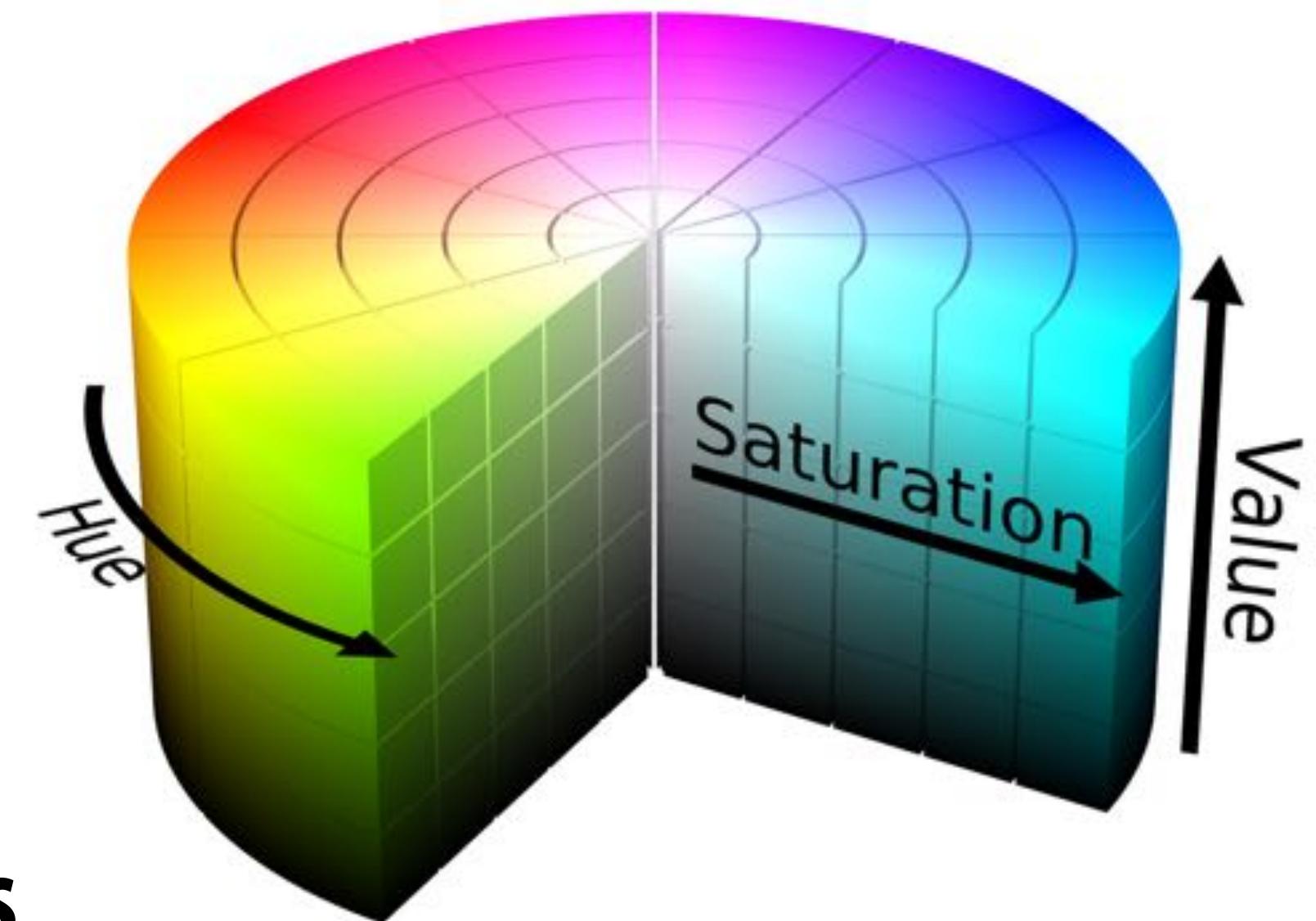
\*A better name than subtractive might be multiplicative, since we multiply to get the final color!

# Let's shed some light on this picture...



# Other Common Color Models

- **HSV**
  - hue, saturation, value
  - more intuitive than RGB/CMYK
- **SML—physiological model**
  - corresponds to stimulus of cones
  - not practical for most color work
- **XYZ—perceptually-driven model**
  - Y captures luminance (intensity)
  - X,Z capture chromaticity (color)
  - related to, but different from, SML
- **Lab—“perceptually uniform” modification of XYZ**



# Practical Encoding of Color Values

- How do colors actually get encoded digitally?
- One common encoding (e.g., HTML): 8bpc hexadecimal values\*:

**# 1B1F8A**

- What does this string mean? Common encoding of RGB.
- Want to store 8-bits per channel (red, green, blue), corresponding to 256 possible values
- Rather than use digits 0-9, use 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F
- Single character now encodes 16 values, two characters encode  $16 \times 16 = 256$  values
- Q: Roughly what color is # ff6600?



\*Upper vs. lowercase letters? Makes absolutely no difference!

# Other Ways of Specifying Color?

- Other color specifications not based on continuous color space
- E.g., Pantone Matching System
  - industry standard (proprietary)
  - 1,114 colors
  - Combination of 13 base pigments
- And not to forget...



# Why use different color models?

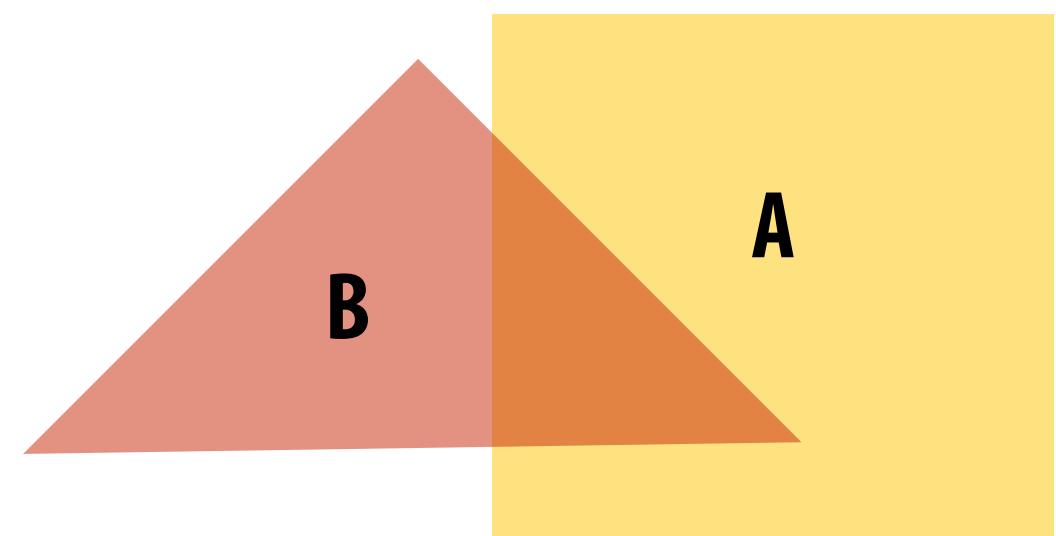
## ■ Convenience

- Is it easy for a user to choose the color they want?



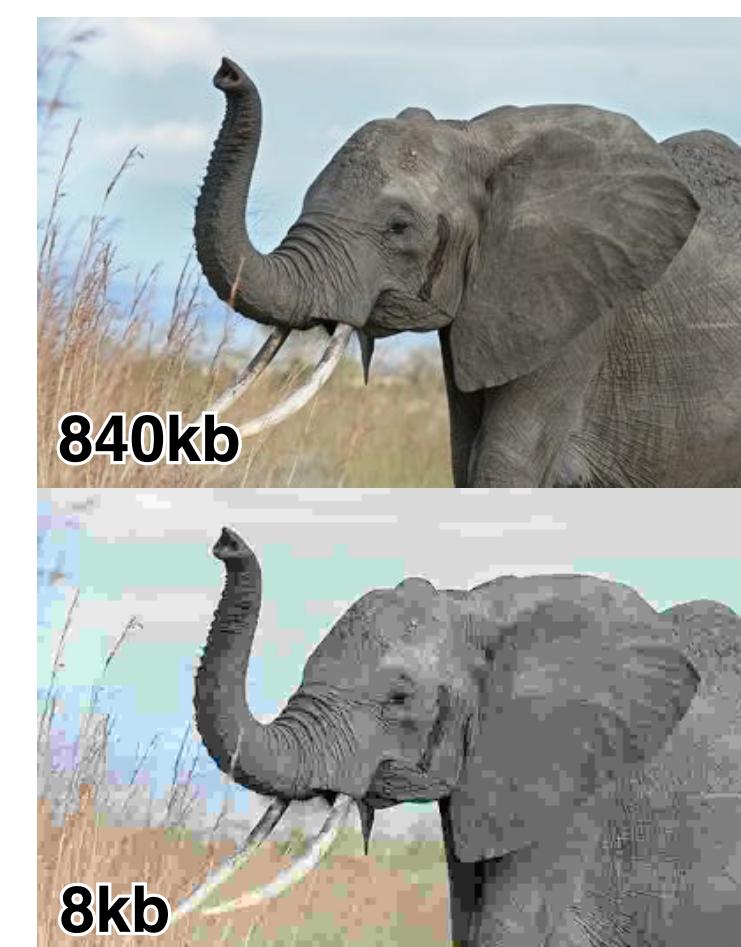
## ■ Color compositing/processing

- Does it matter what color space we interpolate / blend in?



## ■ Efficiency of encoding

- E.g., use more of numerical range for perceptually significant colors
- Compression!



# Example: Y'CbCr color model

- Common for modern digital video
- $Y'$  = luma: perceived luminance (same as  $L^*$  in CIELAB)
- $Cb$  = blue-yellow deviation from gray
- $Cr$  = red-cyan deviation from gray

(input)

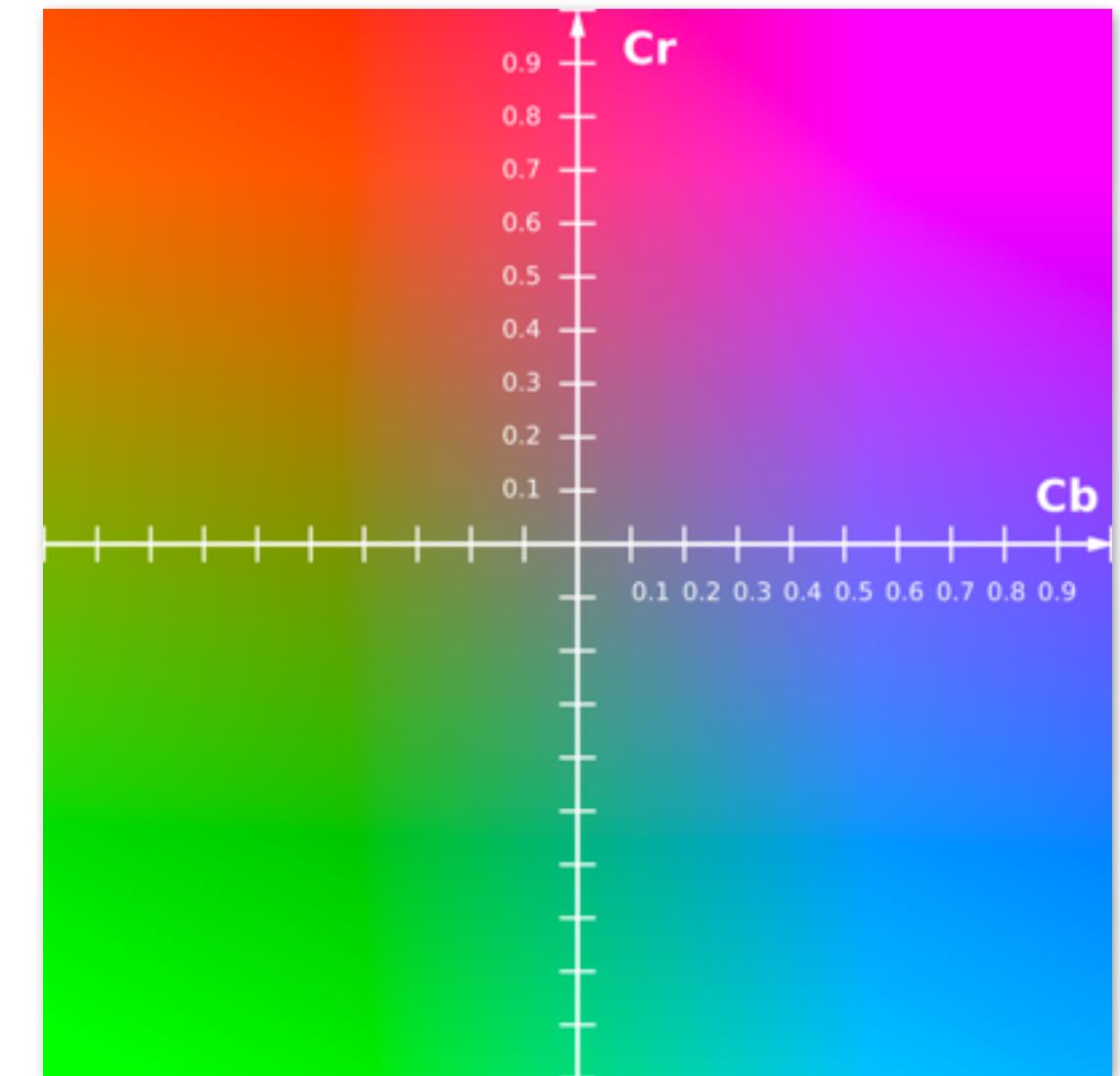


$Y'$

$Cb$



$Cr$





**Original picture**



**Contents of CbCr color channels downsampled by a factor of 20 in each dimension  
(400x reduction in number of samples)**



**Full resolution sampling of luma ( $Y'$ )**



**Reconstructed result  
(looks pretty good)**



**Original picture**

**By the way, how might we reduce this artifact?**



**Reconstructed result**

# Why use different color models? (cont.)

## ■ Convenience

- Is it easy for a user to choose the color they want?

## ■ Efficiency of encoding

- E.g., use more of numerical range for perceptually significant colors
- Do color images compress well?

## ■ Gamut

- Which colors can be expressed using a given model?
- Very different for print vs. display



RGB



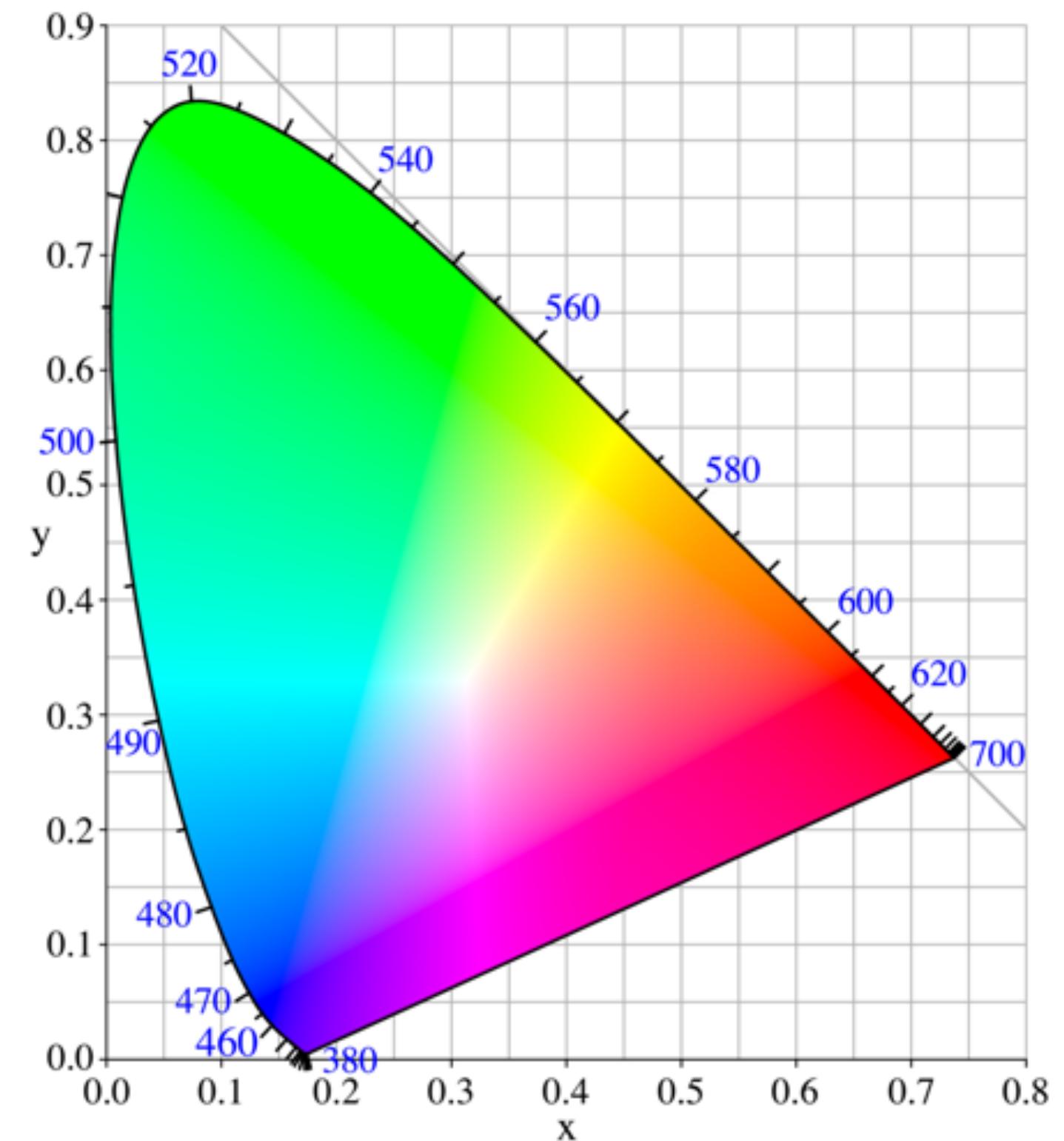
CMYK

**Which raises a very important question:**

**Which actual colors (i.e., spectra) do  
these values get mapped to?**

# CIE 1931\* Color Space

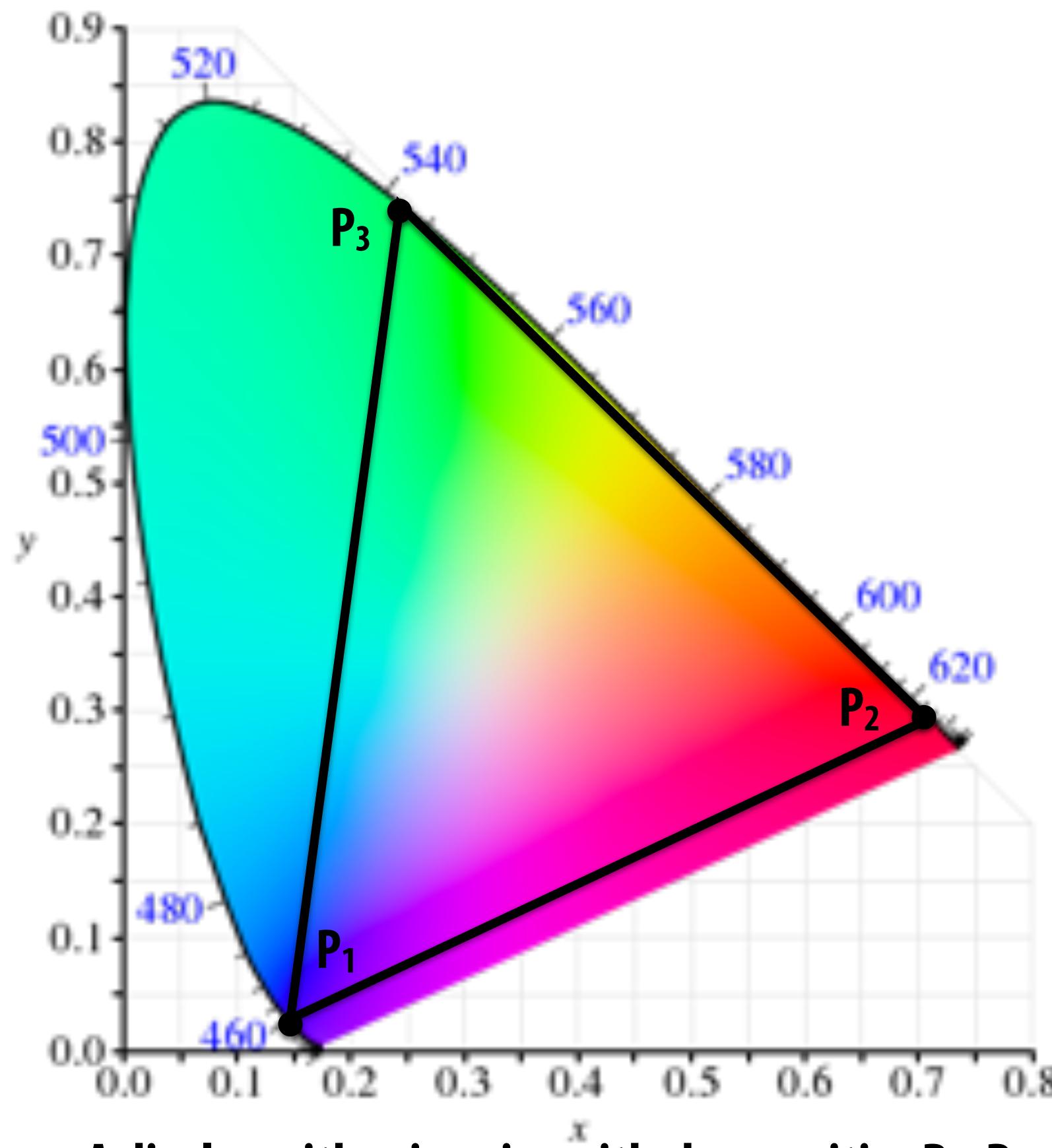
- Standard “reference” color space
- Encompasses all colors visible by “most” human observers
  - associated color model (XYZ) captures perceptual effects
  - e.g., perception of color (“chromaticity”) changes w/ brightness (“luminosity”)
  - different from specifying direct simulation of cones (SML)



\*CIE 1931 does not mean anything important: “created in 1931 by the Commission Internationale de l’Éclairage”

# Chromaticity Diagrams

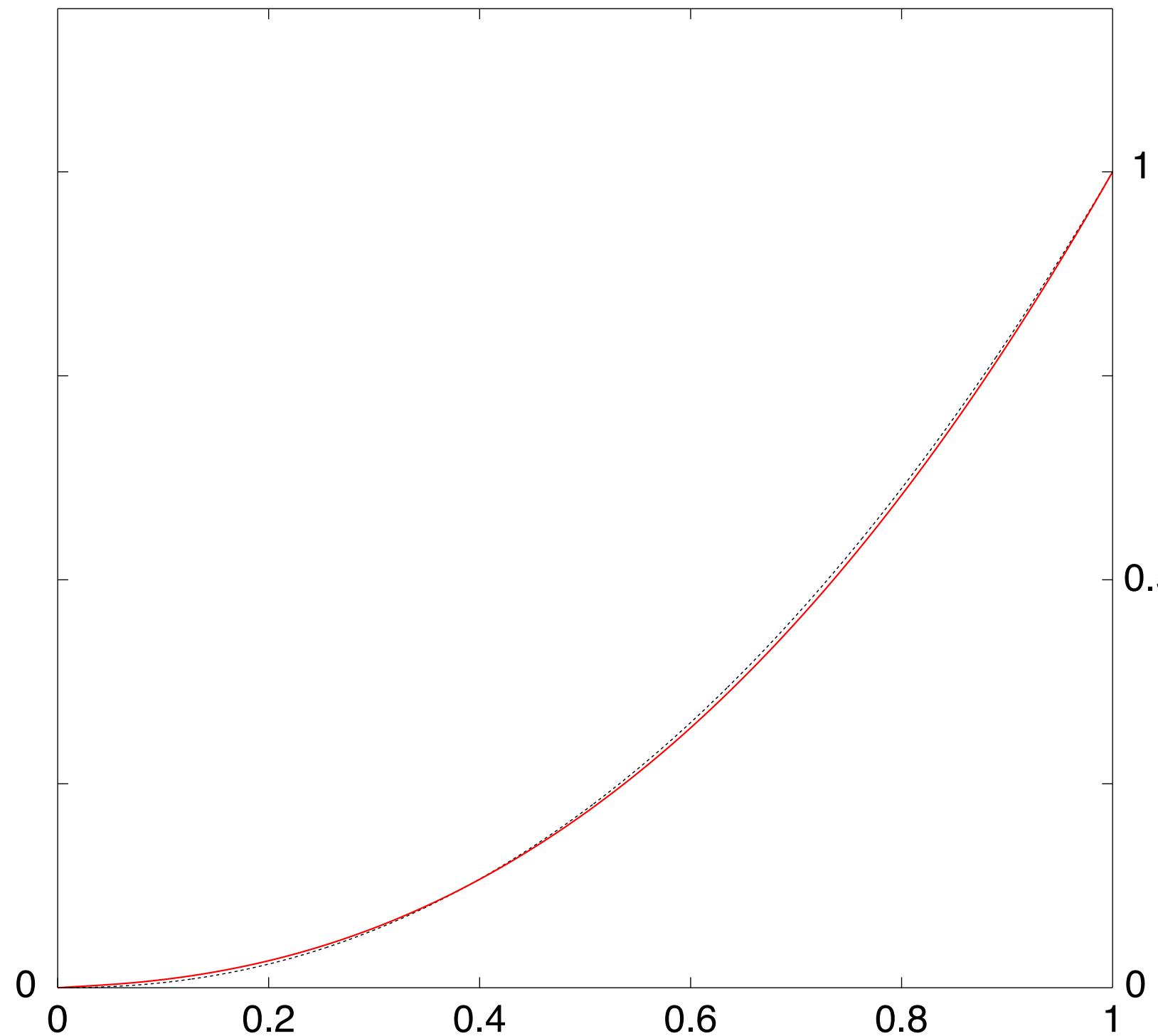
- Chromaticity is the intensity-independent component of a color
- Chromaticity diagram used to visualize extent of a color space



A display with primaries with chromacities  $P_1$ ,  $P_2$ ,  $P_3$   
can create colors that are combinations of these  
primaries (colors that fall within the triangle)

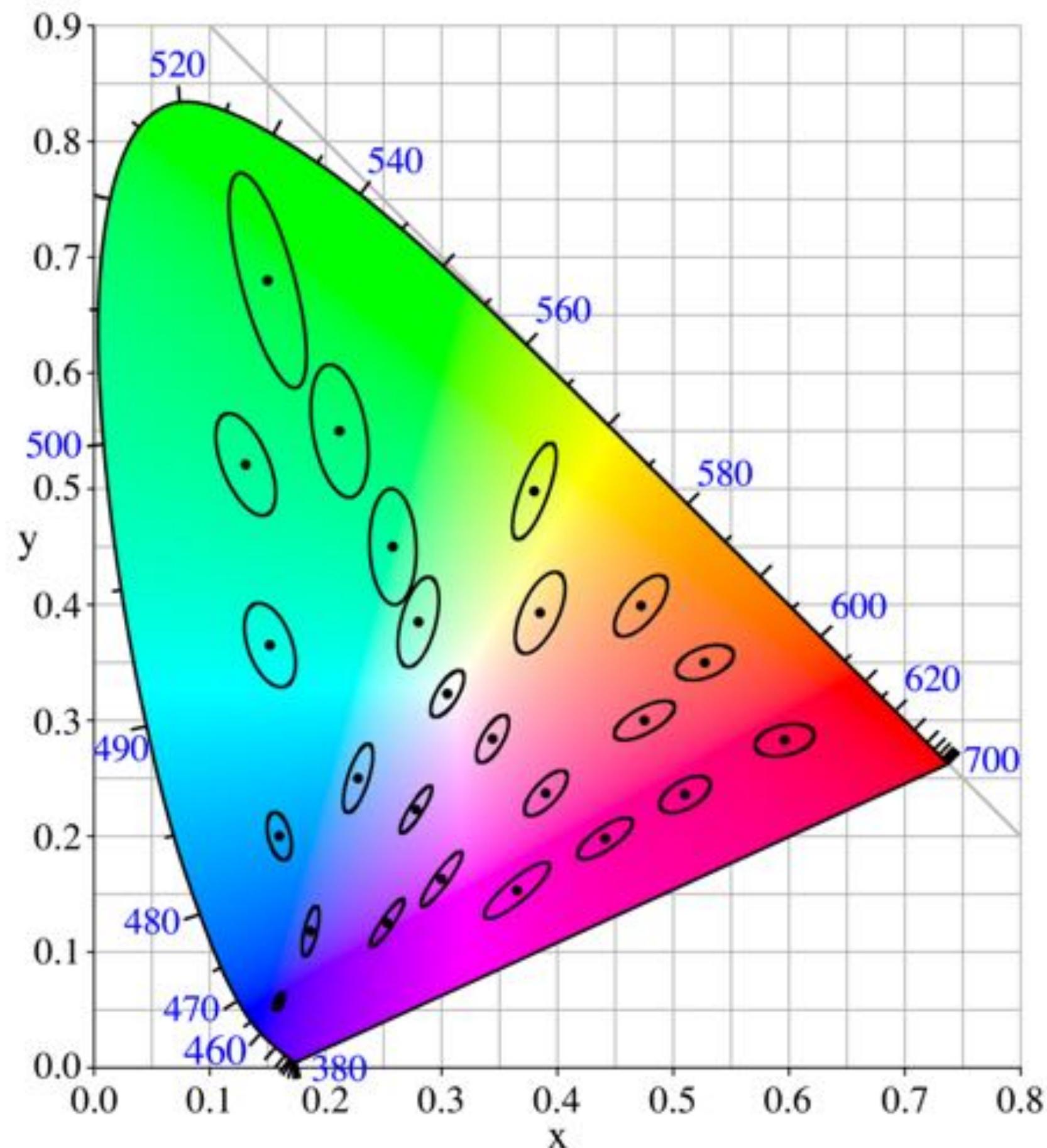
# sRGB Color Space

- CIE 1934 captured all possible human-visible colors
- sRGB (roughly) subset of colors available on displays, printers, ...
- Nonlinear relationship between stored RGB values & intensity
  - Makes better use of limited set of numerical values



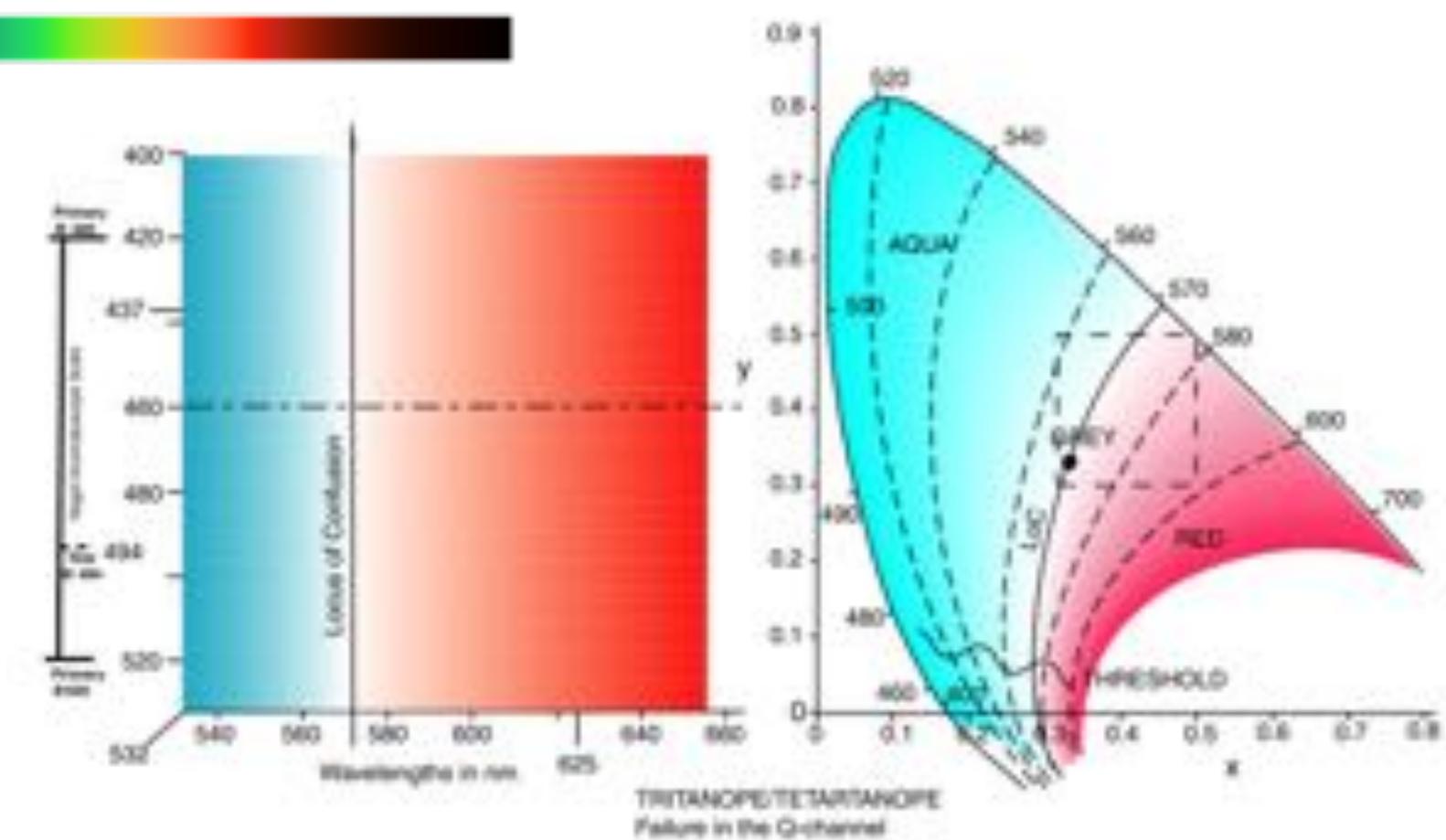
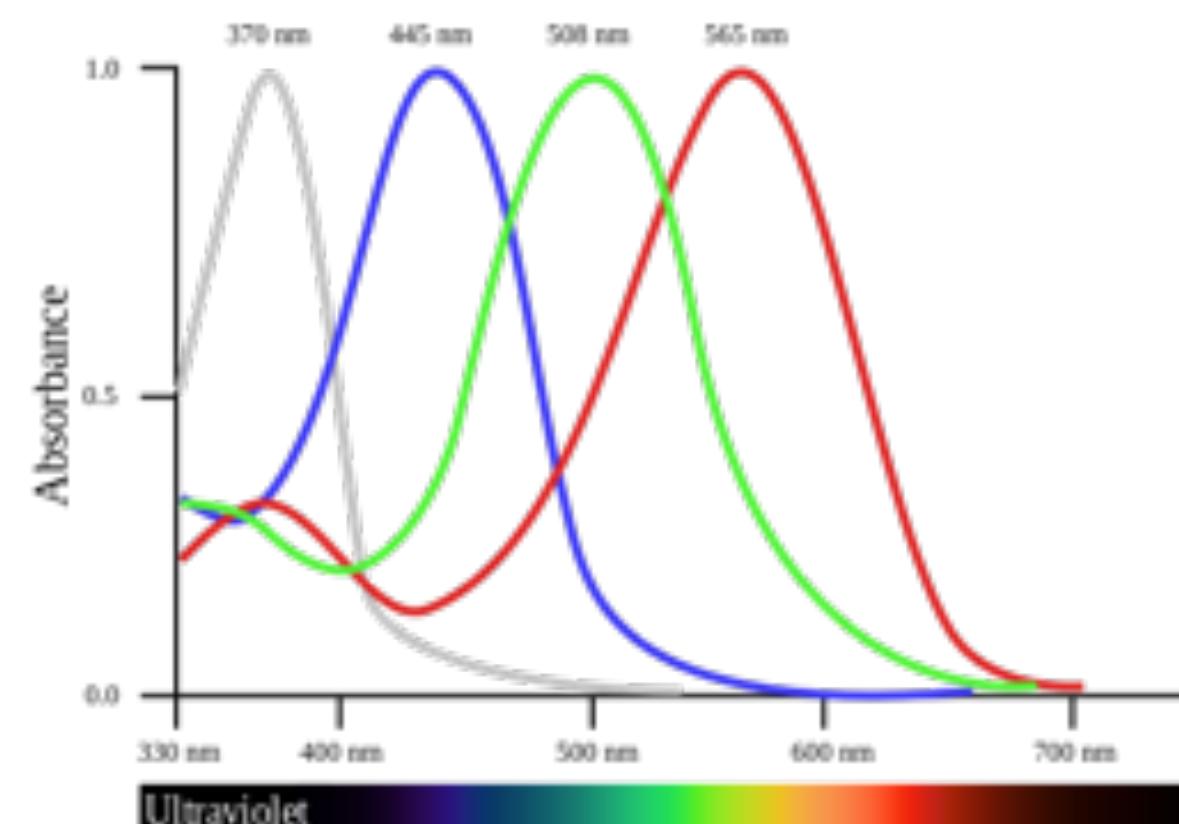
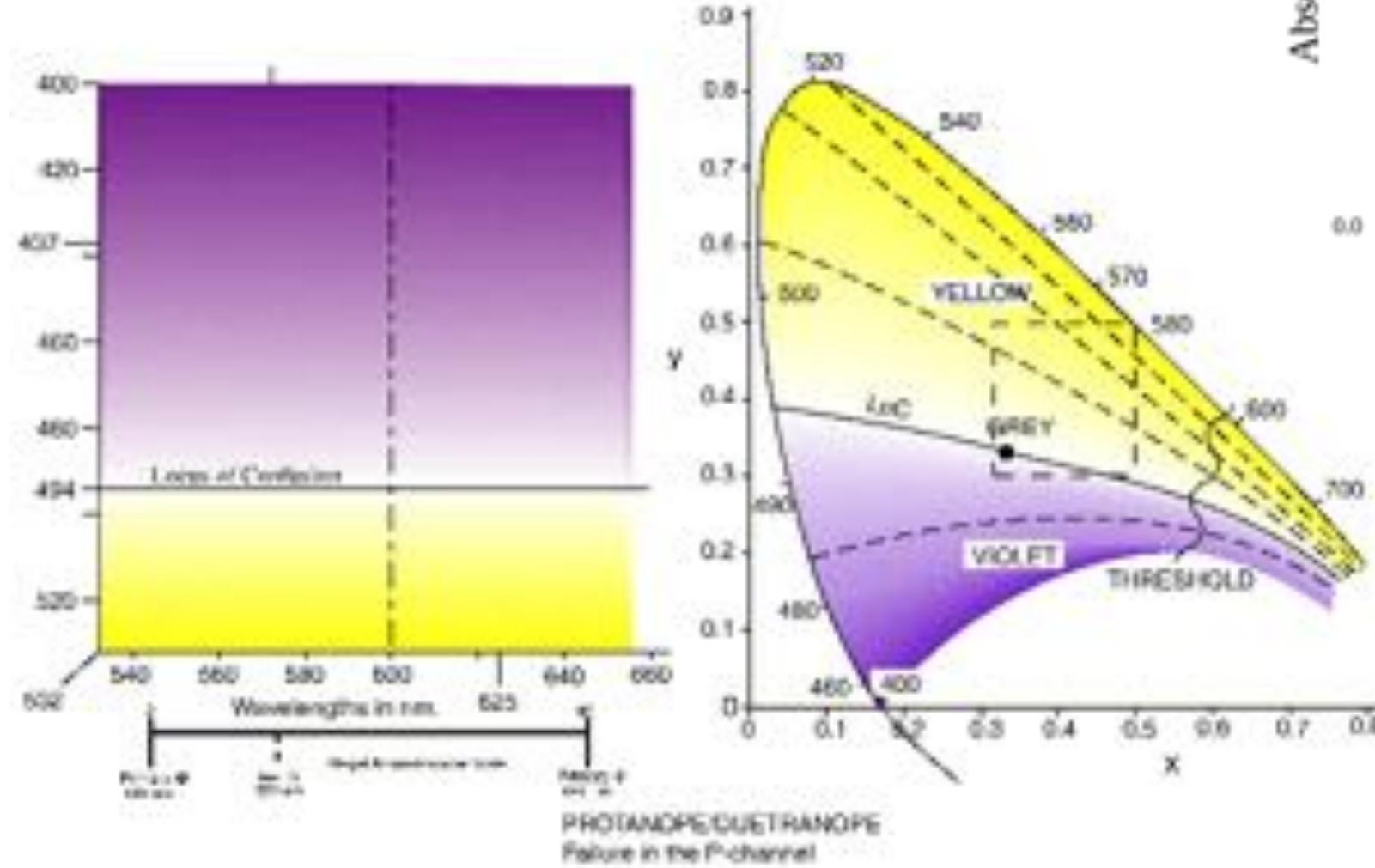
# Color Acuity (MacAdam Ellipse)

- In addition to range of colors visible, one might be interested in how sensitive people are to changes in color
- Each ellipse corresponds to a region of “just noticeable differences” of color (chromaticity)
- So, if you want to make two colors distinct, at bare minimum should avoid overlapping ellipses...



# Nonstandard Color Vision

- Morphological differences in eye can cause people (& animals) to see different ranges of color (e.g., more/fewer cone types)
  - Alternative chromaticity diagrams help visualize color gamut, useful for designing, e.g., widely-accessible interfaces



# Color Conversion

- Given a color specified in one model/space (e.g., sRGB), try to find corresponding color in another model (e.g., CMYK)
- In a perfect world: want to match output spectrum
- Even matching perception of color would be terrific (metamers)
- In reality: may not always be possible!
  - Depends on the gamut of the output device
  - E.g., VR headset vs. inkjet printer
- Complicated task!
- Lots of standards & software
  - ICC Profiles



# Gamma correction

(non-linear correction for CRT display)

Old CRT display:

1. Image contains value X
2. CRT display converts digital signal to an electron beam voltage  $V(x)$  (linear relationship)
3. Electron beam voltage converted to light:  
(non-linear relationship)

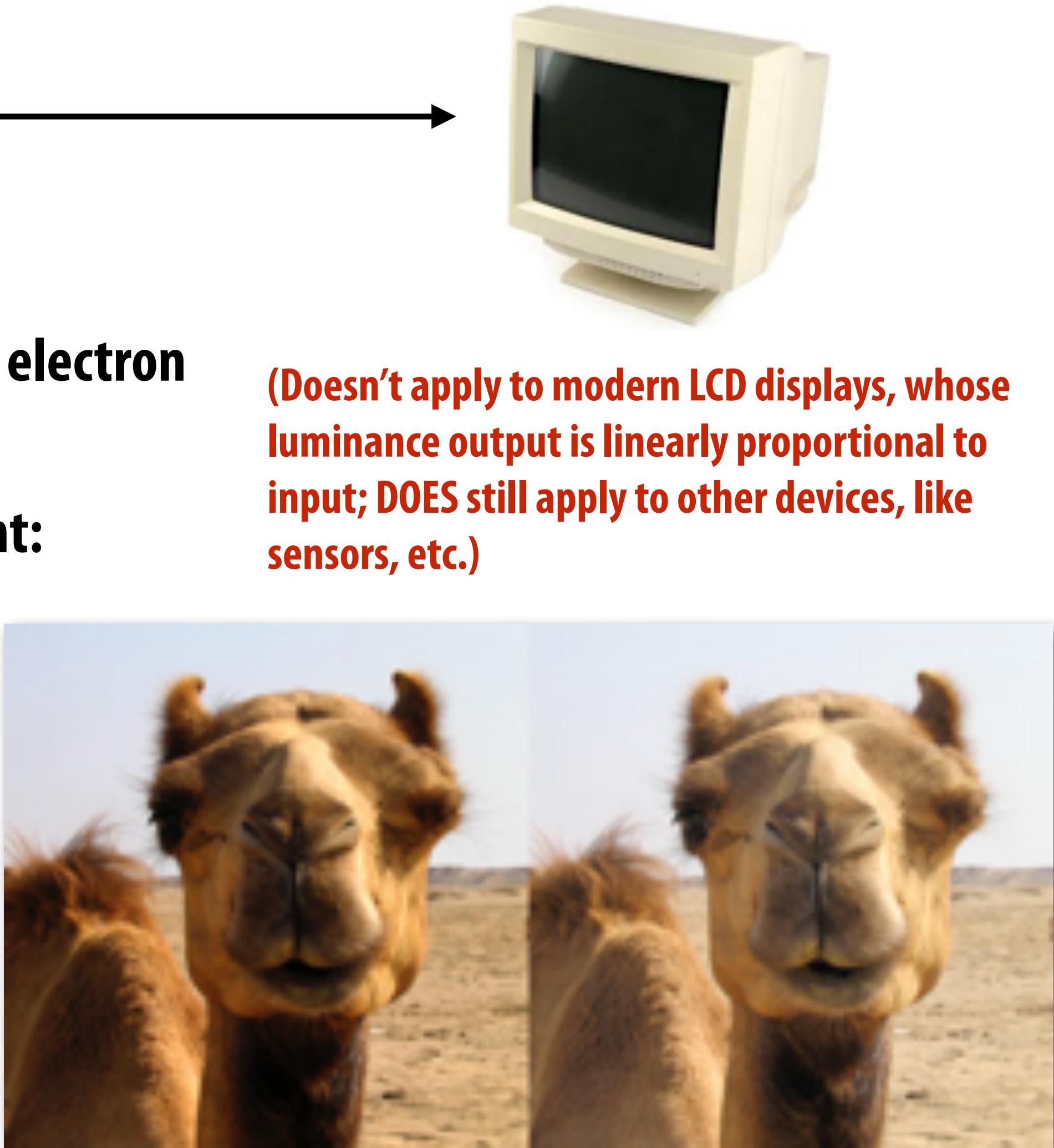
$$Y \propto V^\gamma$$

Where:  $\gamma \approx 2.5$

So if pixels store Y, what will the display's output look like?

Fix: pixels sent to display must store:

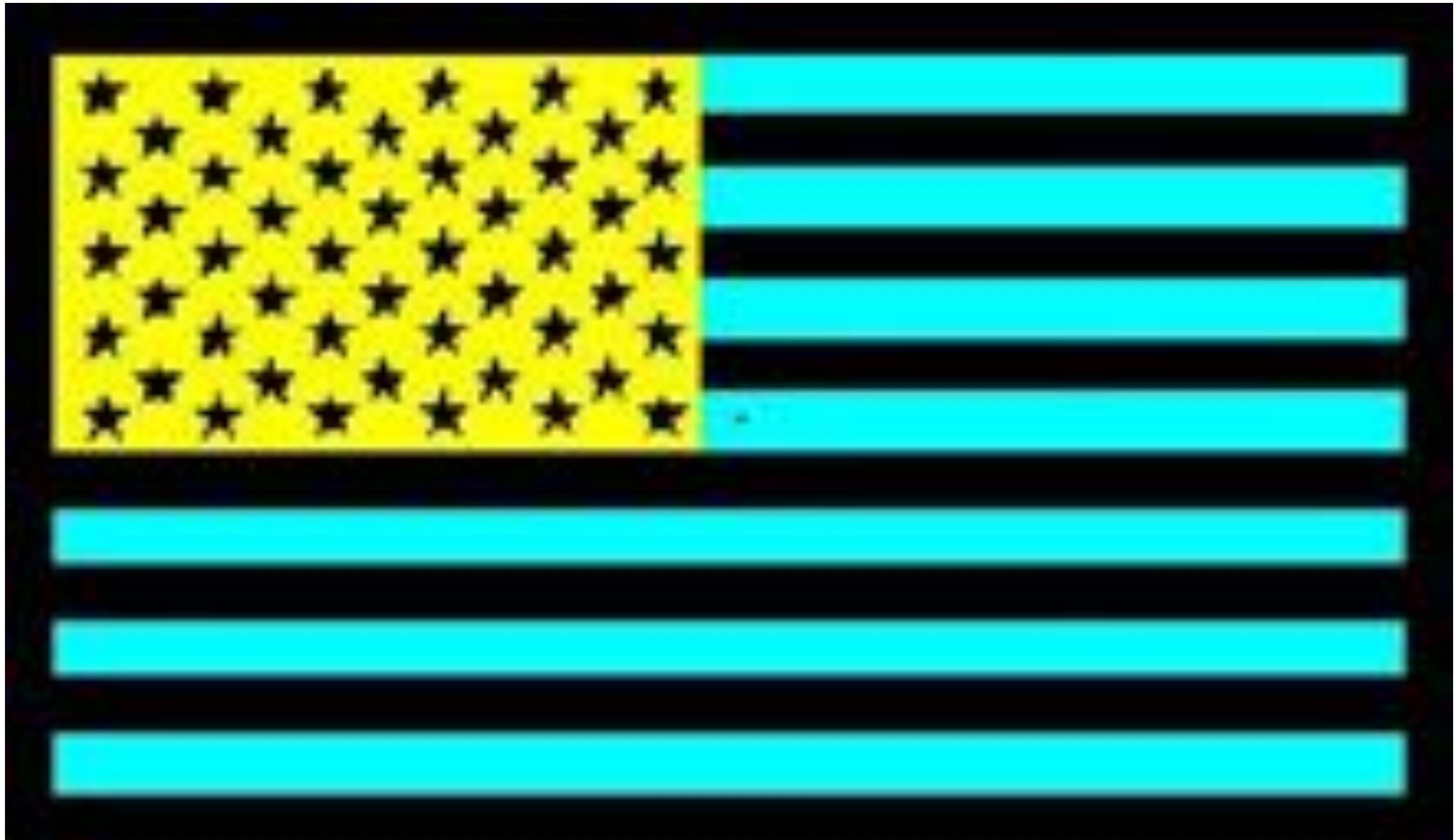
$$Y^{1/2.5} = Y^{0.4}$$



Observed  
display output

Desired  
display output

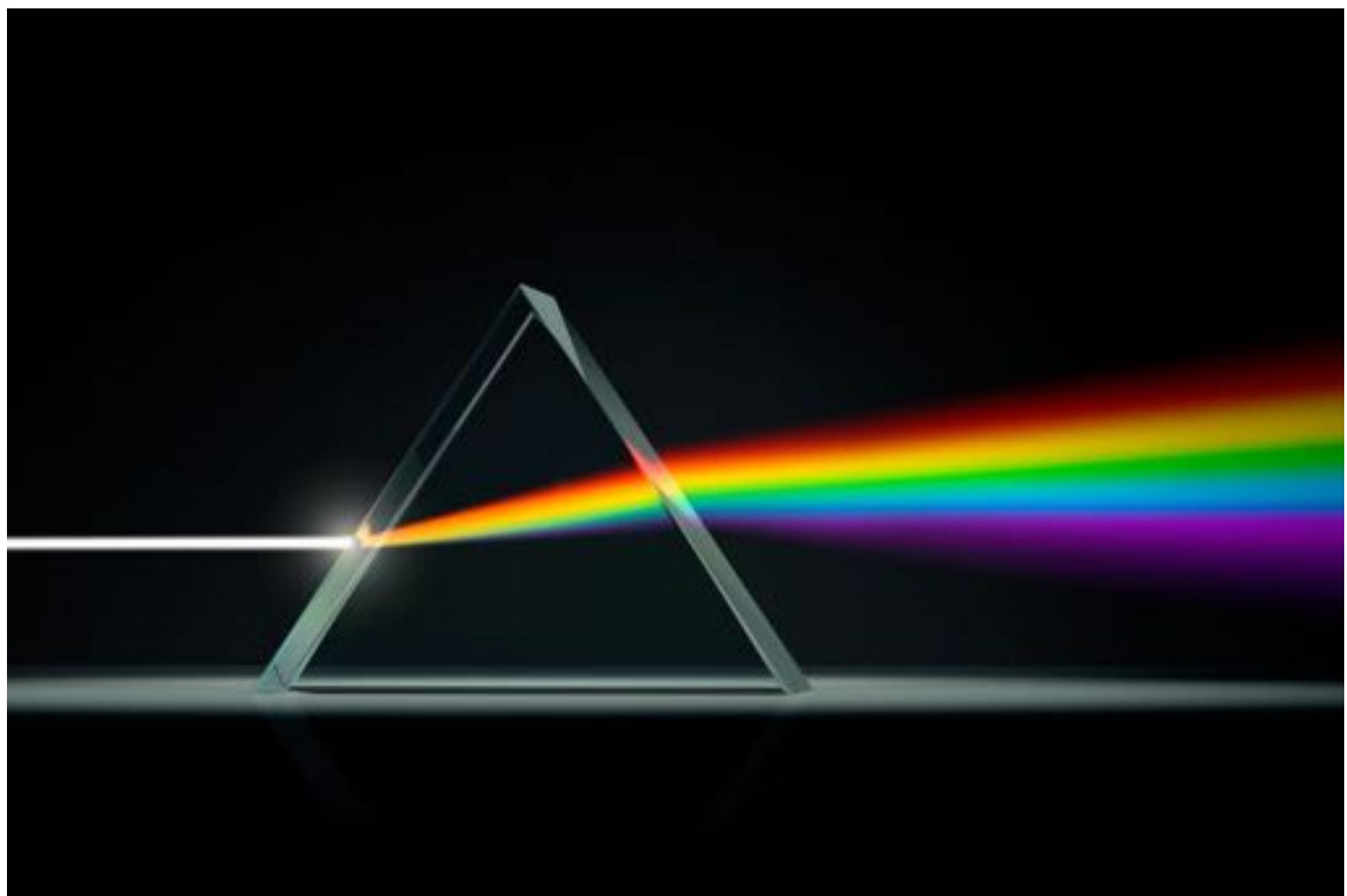
# Human Perception—Accommodation Effect



# **Human Perception—A accommodation Effect**

# Next time...

- A whole spectrum of things to know about light & color
- In the next few lectures we'll talk more about
  - radiometry
  - cameras
  - scattering
  - ...



# The Rendering Equation

---

**Computer Graphics  
CMU 15-462/15-662**

# Recap: Incident vs. Exitant Radiance

**INCIDENT**



**EXITANT**



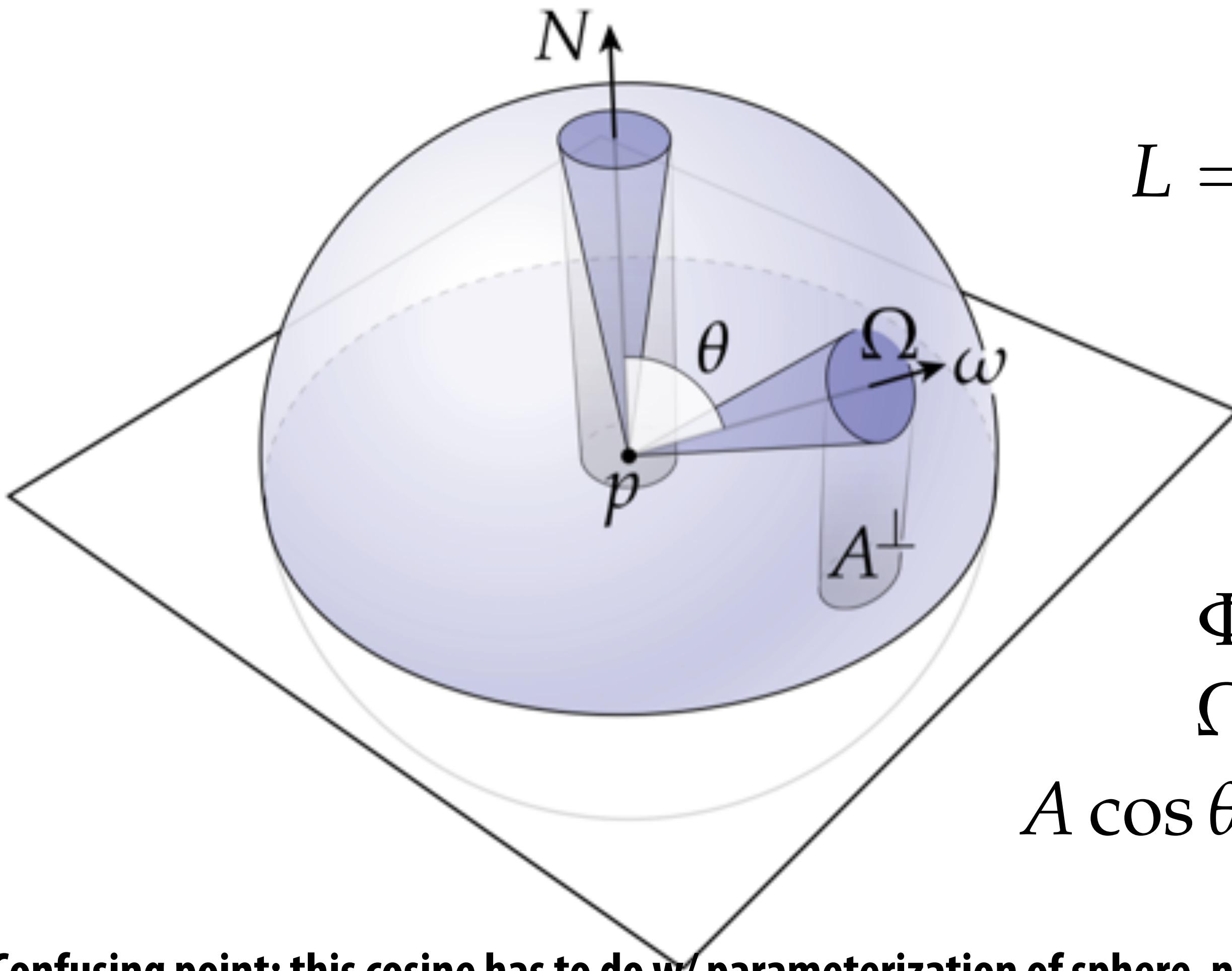
**In both cases: intensity of illumination is highly dependent on **direction** (not just location in space or moment in time).**

# Recap: Radiance and Irradiance



# Recap: What is radiance?

- Radiance at point  $p$  in direction  $N$  is radiant energy ("#hits") per unit time, per solid angle, per unit area perpendicular to  $N$ .



$$L = \frac{\partial^2 \Phi}{\partial \Omega \partial A \cos \theta}$$

$\Phi$  — radiant flux

$\Omega$  — solid angle

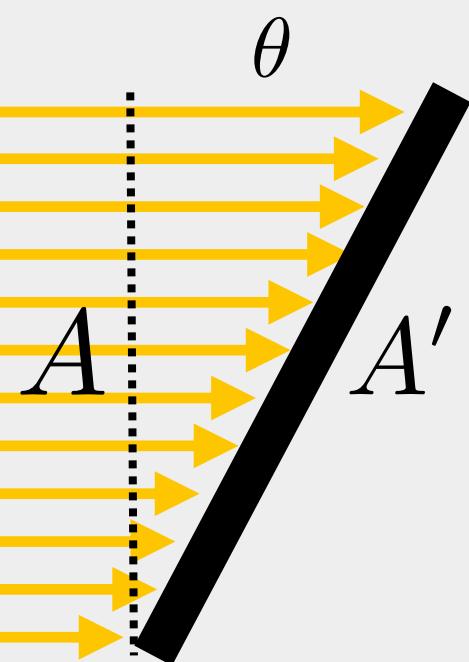
$A \cos \theta$  — projected area\*

\*Confusing point: this cosine has to do w/ parameterization of sphere, not Lambertian reflectance

# Aside: A Tale of Two Cosines

- Confusing point first time you study photorealistic rendering:  
“cos  $\theta$ ” shows up for two completely unrelated reasons

## LAMBERT'S LAW



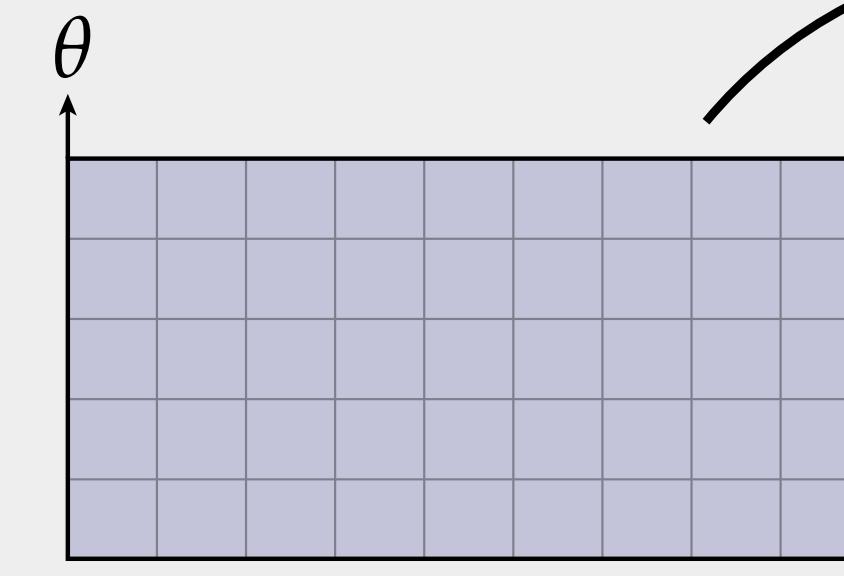
$$A = A' \cos \theta$$

## SPHERICAL INTEGRALS

$$\int_{S^2} f dA$$

==

$$\int_0^{2\pi} \int_{-\pi/2}^{\pi/2} f(\theta, \phi) \cos \theta d\theta d\phi$$



# **Question du jour:**

**How do we use all this stuff  
to generate images?**

# The Rendering Equation

- Core functionality of photorealistic renderer is to estimate radiance at a given point  $p$ , in a given direction  $\omega_o$
- Summed up by the **rendering equation (Kajiya)**:

$$L_o(p, \omega_o) = L_e(p, \omega_o) + \int_{\mathcal{H}^2} f_r(p, \omega_i) \rightarrow \omega_o L_i(p, \omega_i) \cos \theta d\omega_i$$

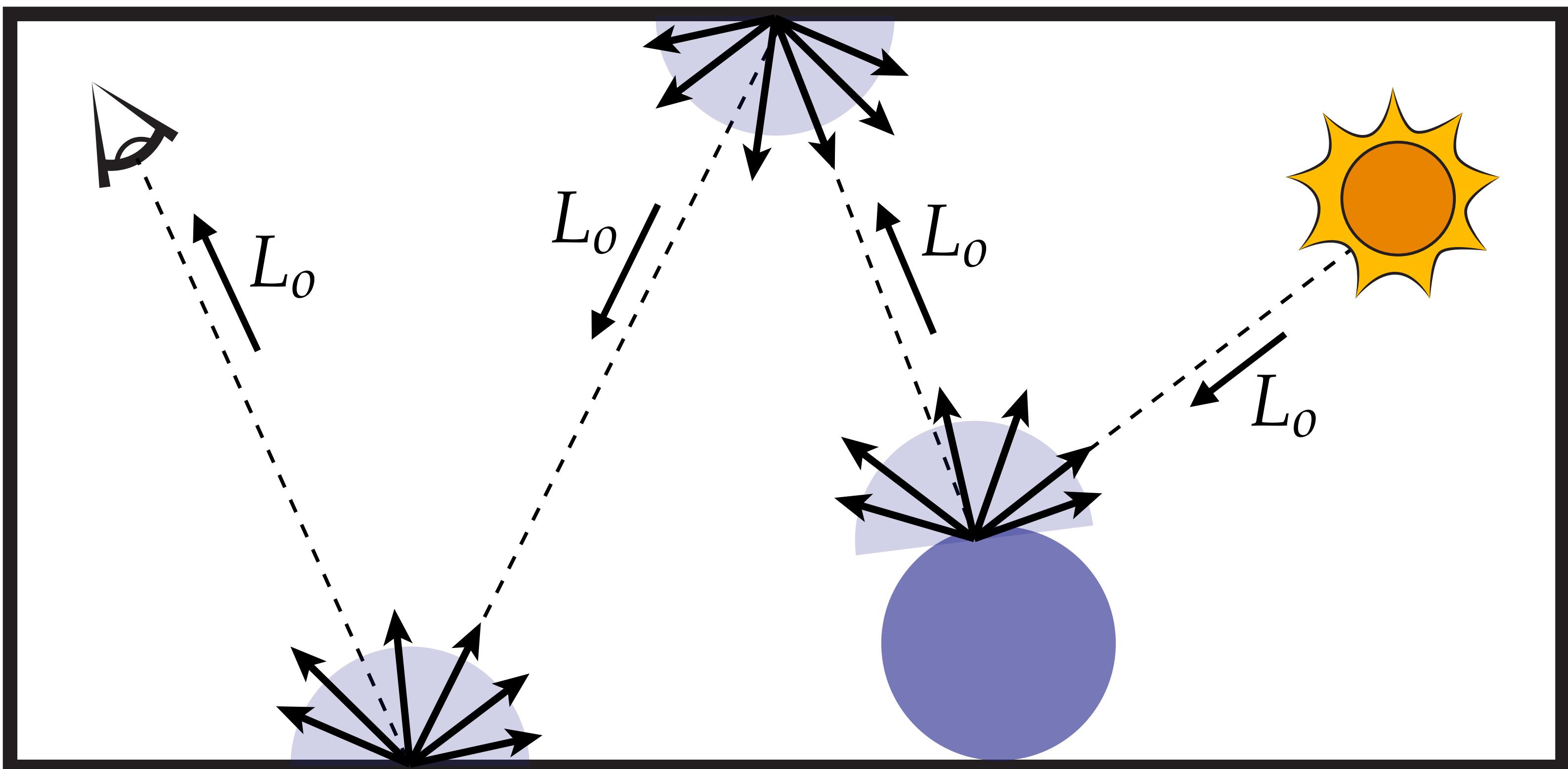
Diagram illustrating the components of the rendering equation:

- outgoing/observed radiance**:  $L_o(p, \omega_o)$
- emitted radiance (e.g., light source)**:  $L_e(p, \omega_o)$
- point of interest**:  $p$
- direction of interest**:  $\omega_o$
- all directions in hemisphere**:  $\mathcal{H}^2$
- scattering function**:  $f_r(p, \omega_i)$
- incoming radiance**:  $L_i(p, \omega_i)$
- angle between incoming direction and normal**:  $\cos \theta$
- incoming direction**:  $\omega_i$

**Key challenge: to evaluate incoming radiance, we have to compute yet another integral. I.e., rendering equation is recursive.**

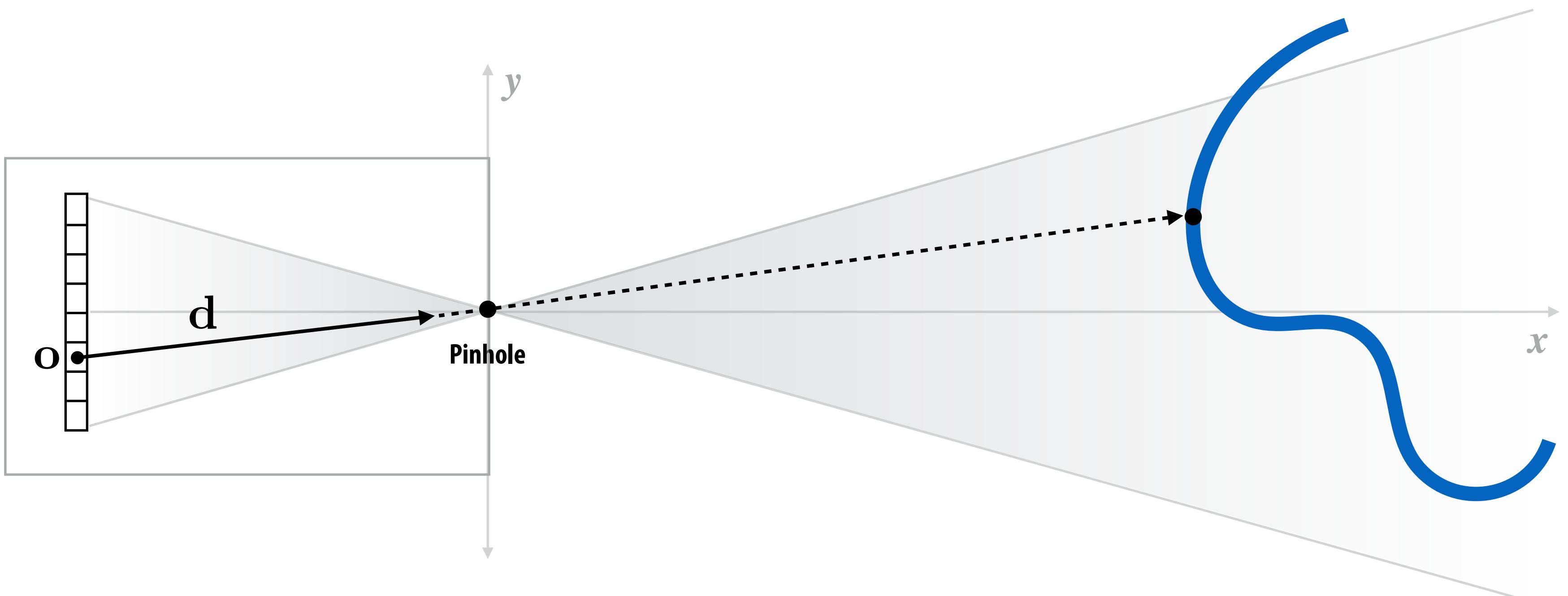
# Recursive Raytracing

- Basic strategy: recursively evaluate rendering equation!



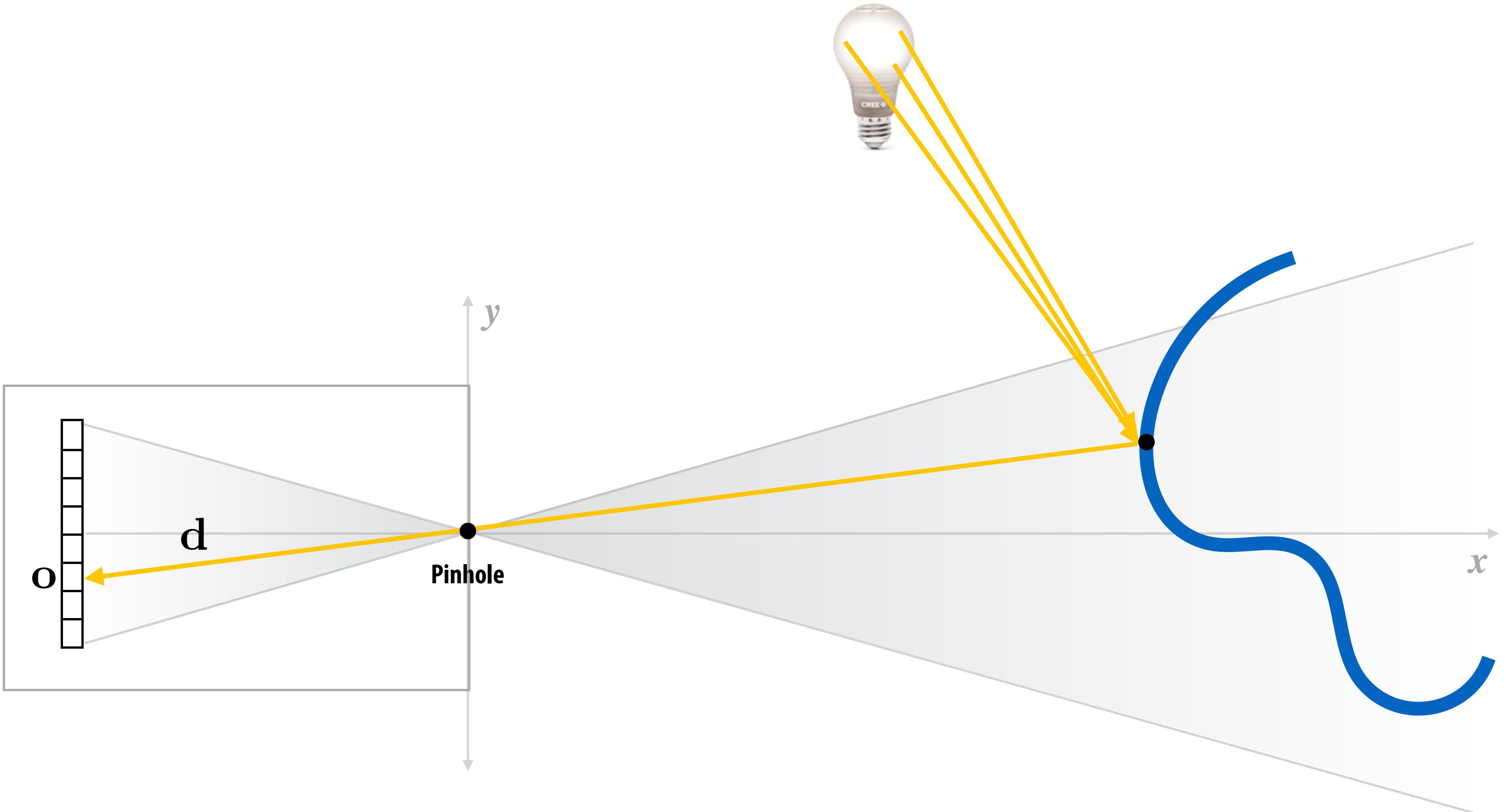
(This is why you're writing a ray tracer—rasterizer isn't enough!)

# Renderer measures radiance along a ray

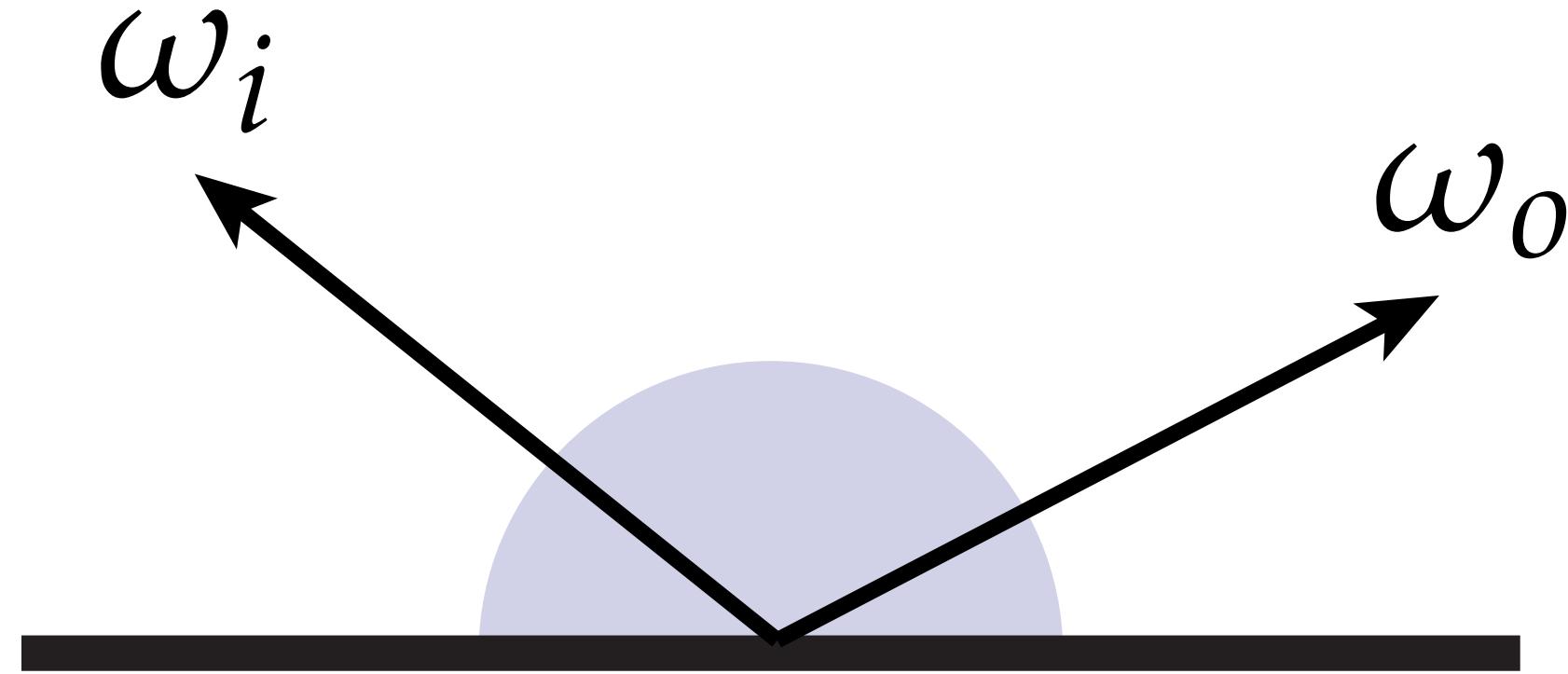


At each “bounce,” want to measure radiance traveling in the direction opposite the ray direction.

# Renderer measures radiance along a ray



Radiance entering camera in direction  $d$  = light from scene light sources that is reflected off surface in direction  $d$ .



**How does reflection of light affect  
the outgoing radiance?**

$$L_o(\mathbf{p}, \omega_o) = \int_{\mathcal{H}^2} f_r(\mathbf{p}, \omega_i, \omega_o) L_i(\mathbf{p}, \omega_i) \cos \theta d\omega_i$$

# Reflection models

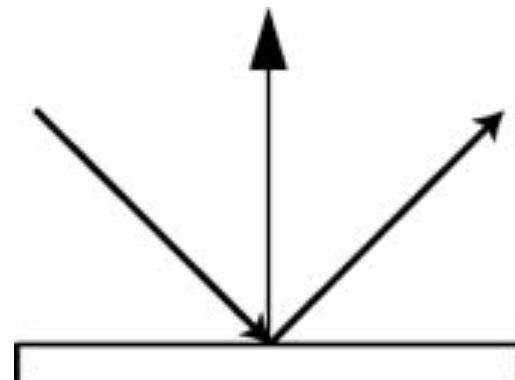
- Reflection is the process by which light incident on a surface interacts with the surface such that it leaves on the incident (same) side without change in frequency
- Choice of reflection function determines surface appearance



# Some basic reflection functions

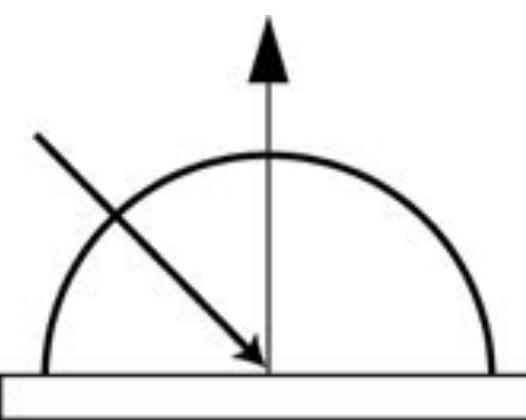
## ■ Ideal specular

Perfect mirror



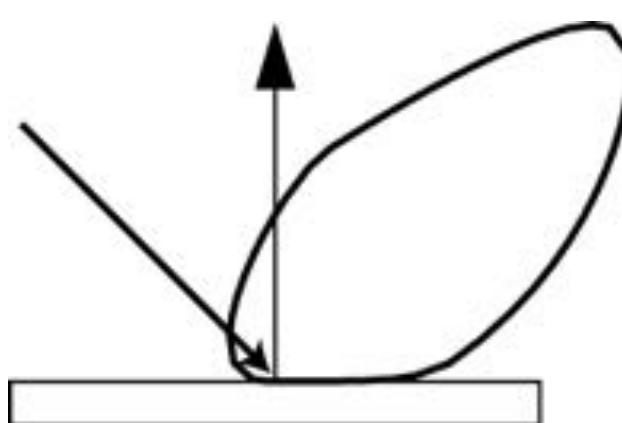
## ■ Ideal diffuse

Uniform reflection in all directions



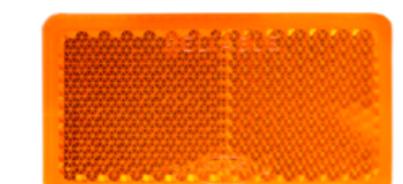
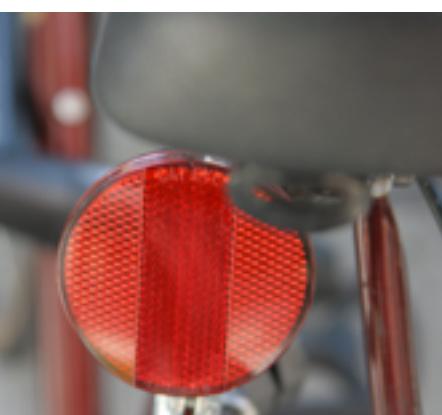
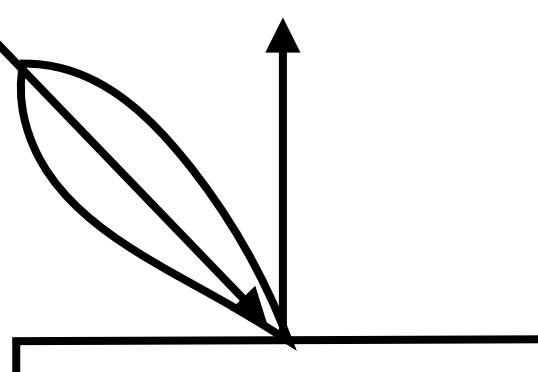
## ■ Glossy specular

Majority of light distributed in reflection direction



## ■ Retro-reflective

Reflects light back toward source



Diagrams illustrate how incoming light energy from given direction is reflected in various directions.

# Materials: diffuse



# Materials: plastic



# Materials: red semi-gloss paint



# Materials: Ford mystic lacquer paint



# Materials: mirror



# Materials: gold



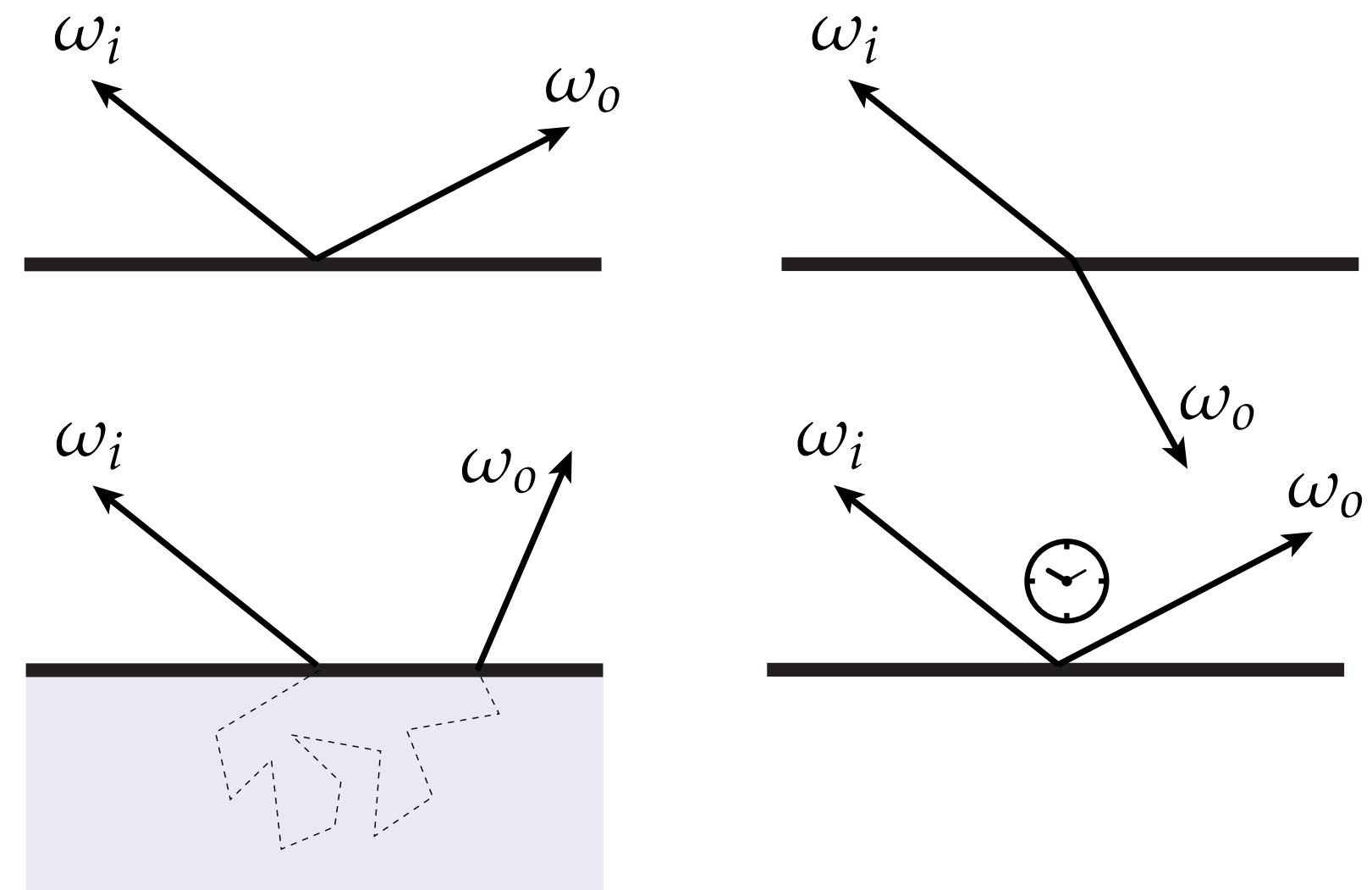
# Materials



# Models of Scattering

- How can we model “scattering” of light?
- Many different things that could happen to a photon:

- bounces off surface
- transmitted through surface
- bounces around inside surface
- absorbed & re-emitted
- ...

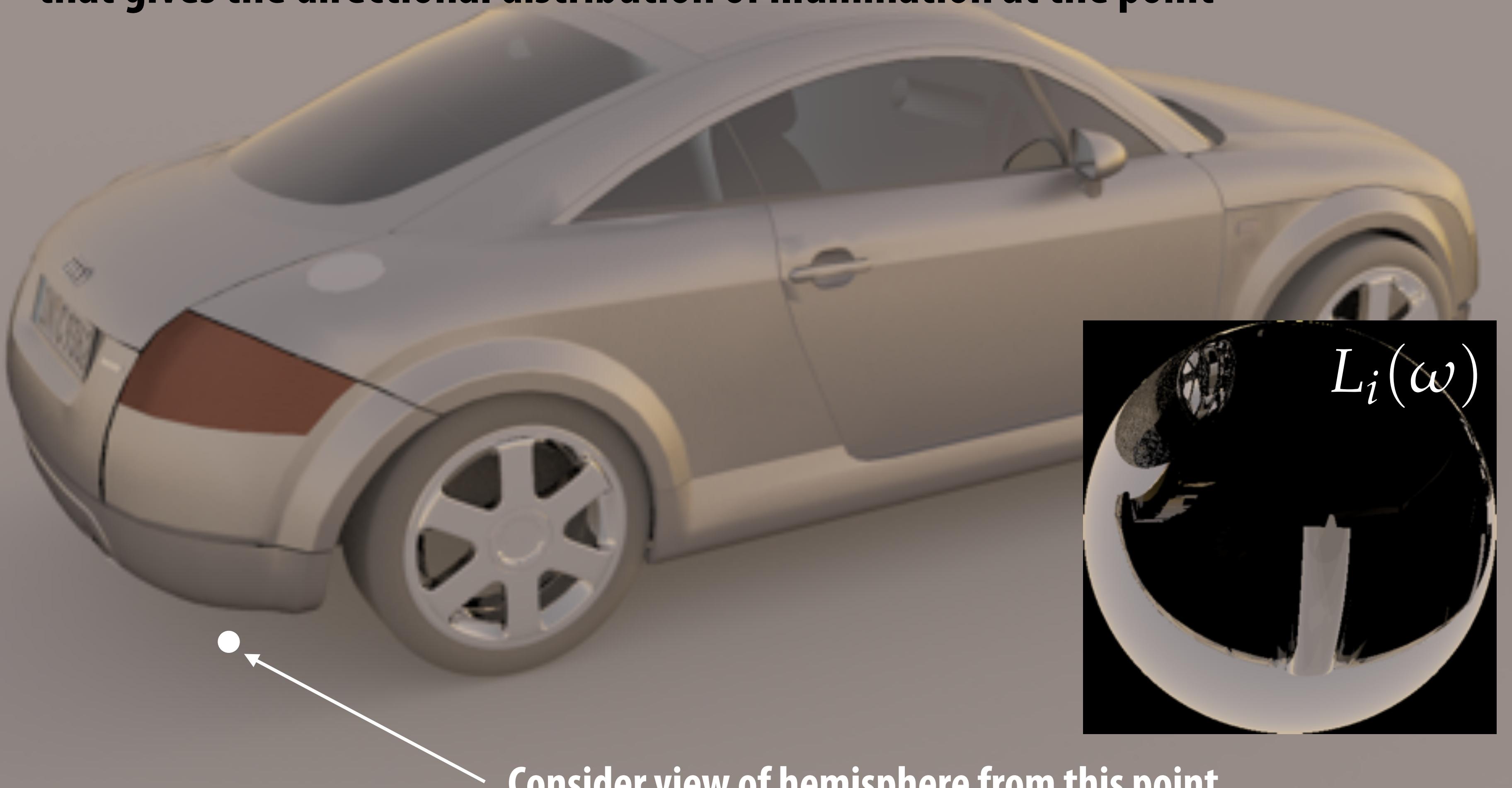


- What goes in must come out! (Total energy must be conserved)
- In general, can talk about “probability\*” a particle arriving from a given direction is scattered in another direction

\*Somewhat more complicated than this, because some light is absorbed!

# Hemispherical incident radiance

At any point on any surface in the scene, there's an incident radiance field that gives the directional distribution of illumination at the point



Consider view of hemisphere from this point

# Diffuse reflection

Exitant radiance is the same in all directions



Incident radiance



Exitant radiance

# Ideal specular reflection

Incident radiance is “flipped around normal” to get exitant radiance



Incident radiance



Exitant radiance

# Plastic

Incident radiance gets “flipped and blurred”



Incident radiance



Exitant radiance

# Copper

More blurring, plus coloration (nonuniform absorption across frequencies)



Incident radiance



Exitant radiance

# Scattering off a surface: the BRDF

- “Bidirectional reflectance distribution function”
- Encodes behavior of light that “bounces off” surface
- Given incoming direction  $\omega_i$ , how much light gets scattered in any given outgoing direction  $\omega_o$ ?
- Describe as distribution  $f_r(\omega_i \rightarrow \omega_o)$

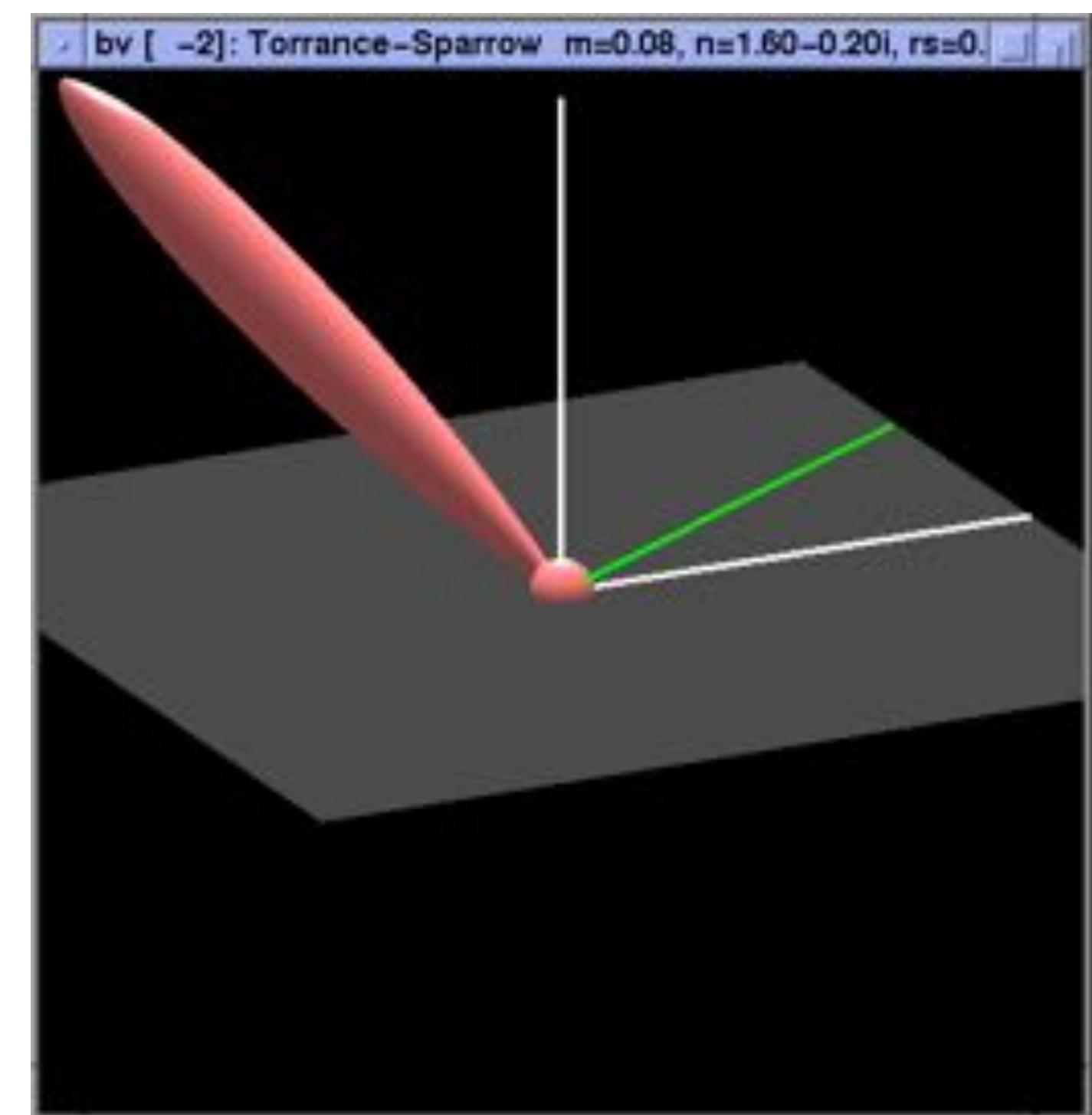
$$f_r(\omega_i \rightarrow \omega_o) \geq 0$$

$$\int_{\mathcal{H}^2} f_r(\omega_i \rightarrow \omega_o) \cos \theta d\omega_i \leq 1$$

why less than or equal?  
where did the rest of the energy go?!

$$f_r(\omega_i \rightarrow \omega_o) = f_r(\omega_o \rightarrow \omega_i)$$

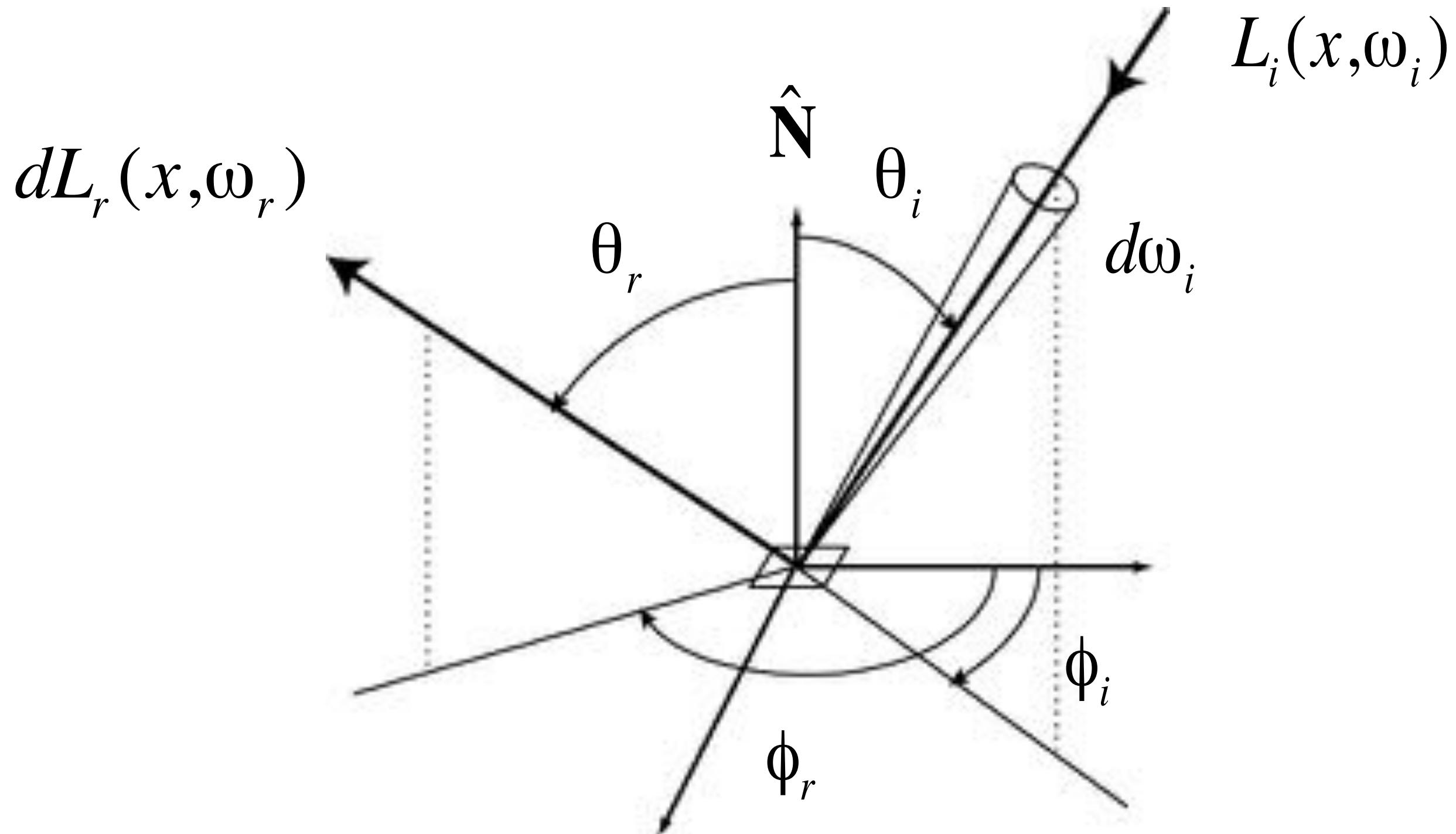
“Helmholtz reciprocity”



**bv** (Szymon Rusinkiewicz)

Q: Why should Helmholtz reciprocity hold? Think about little mirrors...

# Radiometric description of BRDF

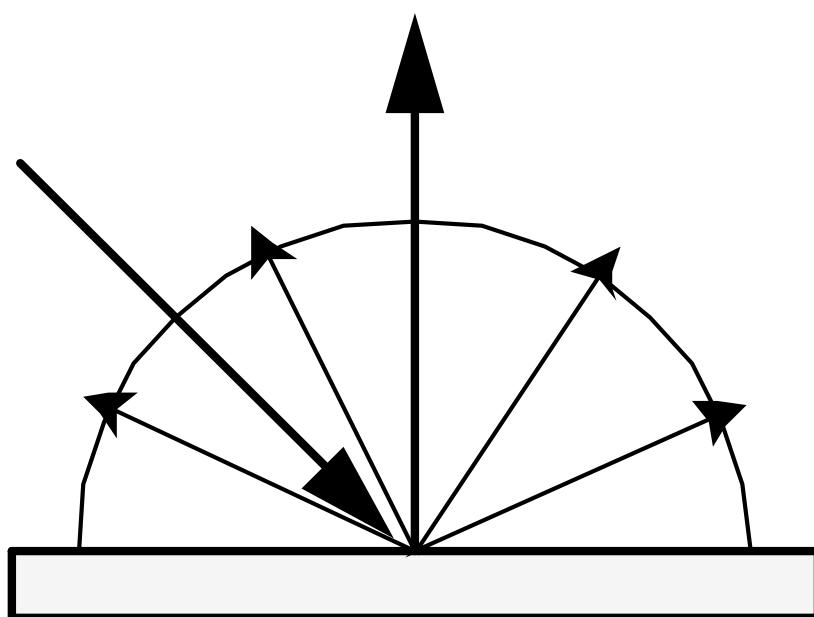


$$f_r(\omega_i \rightarrow \omega_o) = \frac{dL_o(\omega_o)}{dE_i(\omega_i)} = \frac{dL_o(\omega_o)}{dL_i(\omega_i) \cos \theta_i} \left[ \frac{1}{sr} \right]$$

**“For a given change in the incident irradiance, how much does the exitant radiance change?”**

# Example: Lambertian reflection

Assume light is equally likely to be reflected in each output direction



$$f_r = c$$

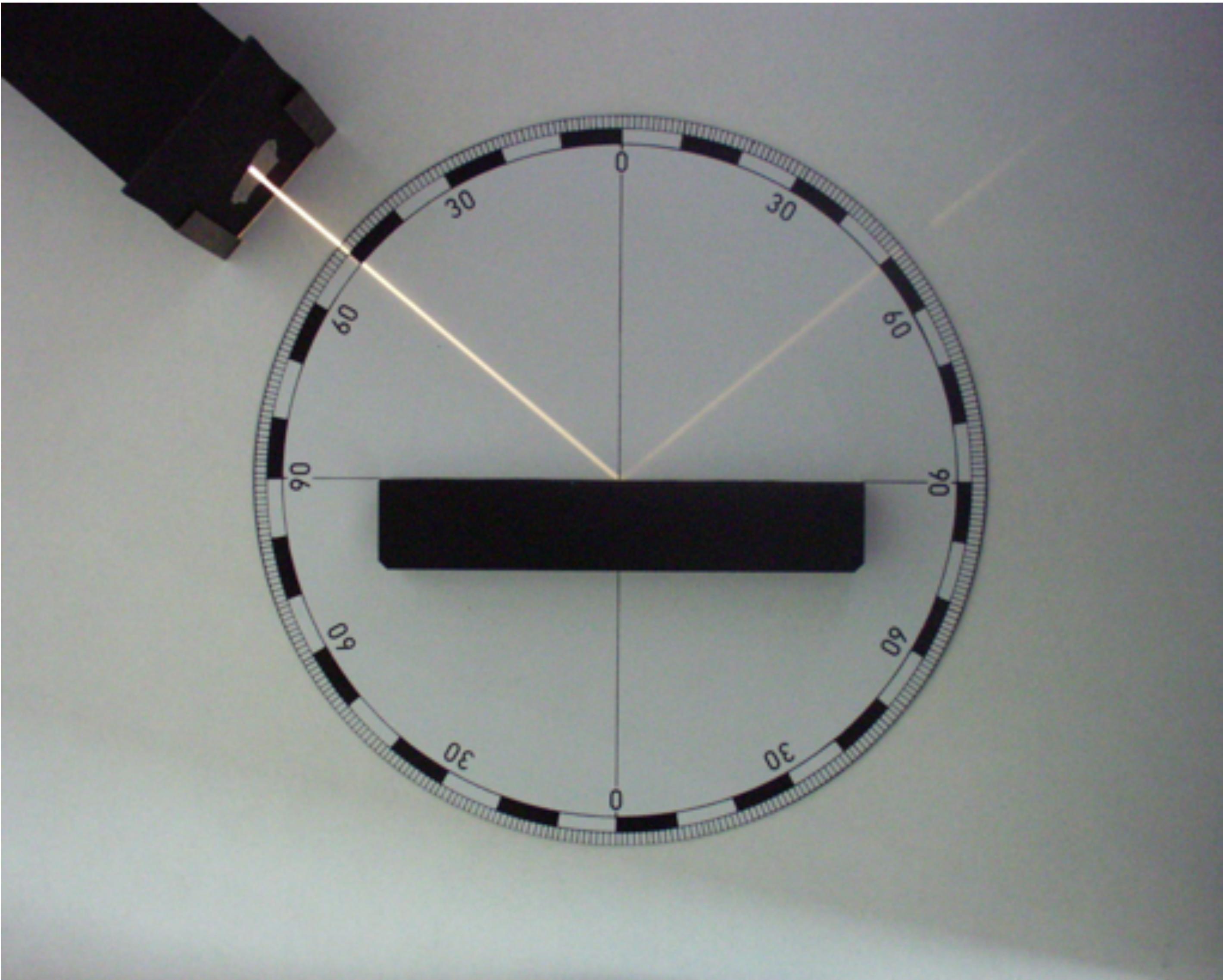
“albedo” (between 0 and 1)

$$f_r = \frac{\rho}{\pi}$$

$$\begin{aligned} L_o(\omega_o) &= \int_{H^2} f_r L_i(\omega_i) \cos \theta_i d\omega_i \\ &= f_r \int_{H^2} L_i(\omega_i) \cos \theta_i d\omega_i \\ &= f_r E \end{aligned}$$

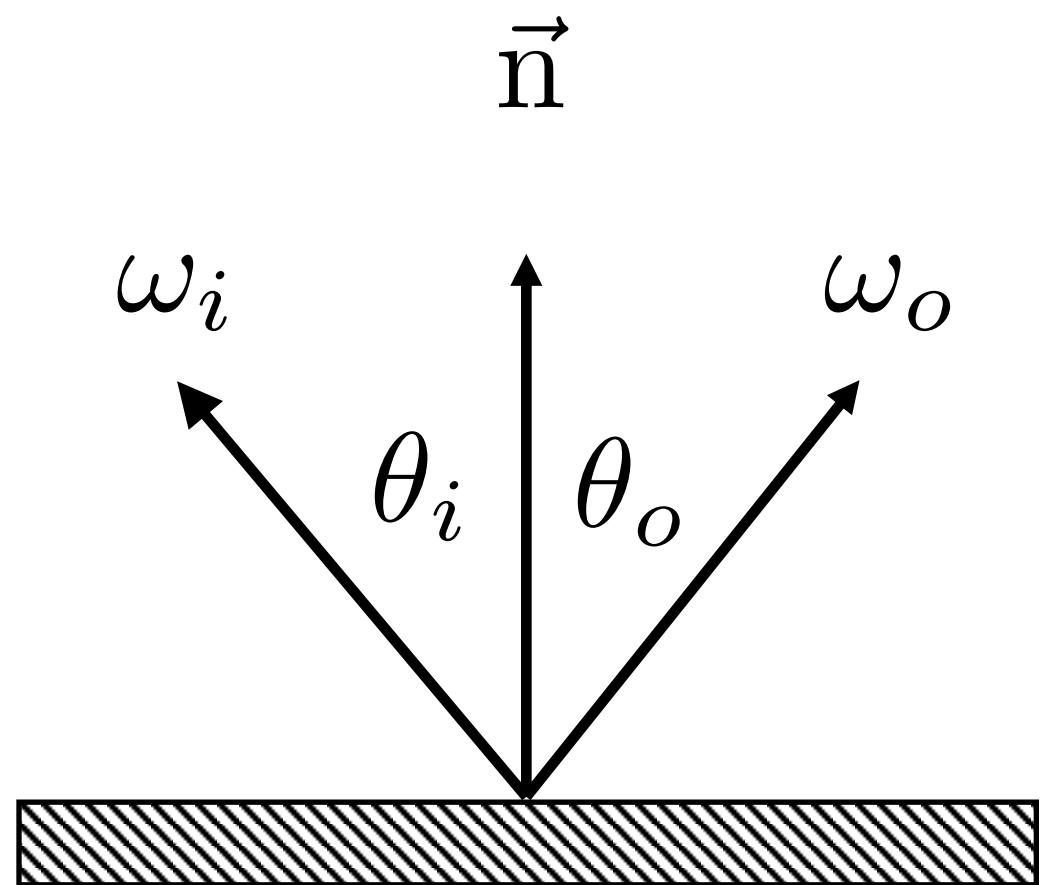


# Example: perfect specular reflection



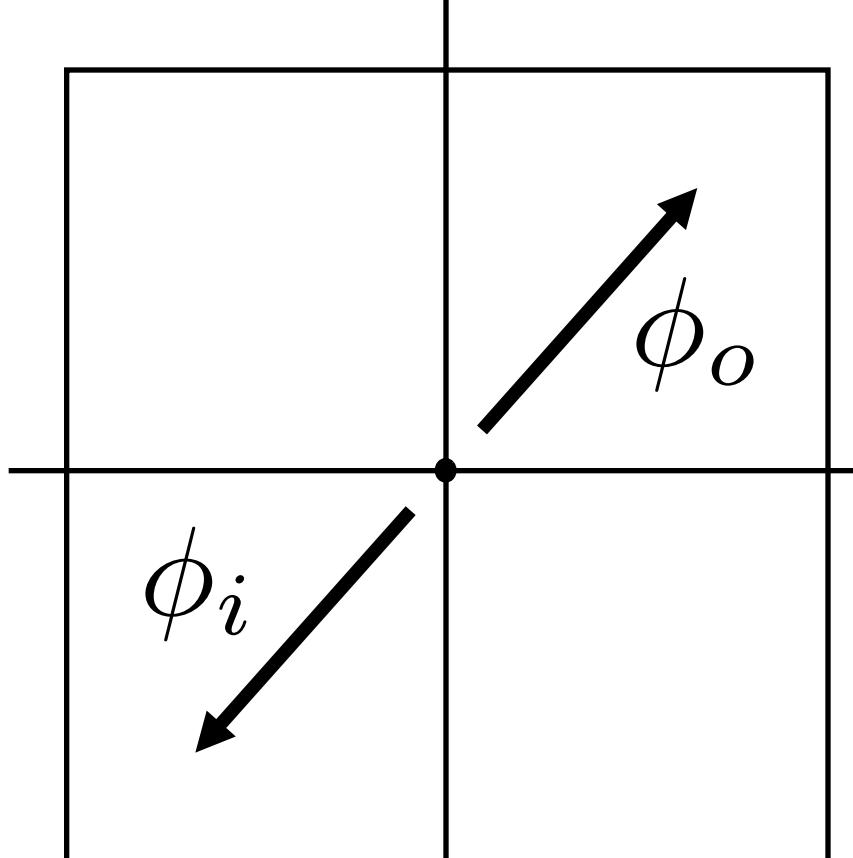
[Zátoky Sándor]

# Geometry of specular reflection



$$\theta = \theta_o = \theta_i$$

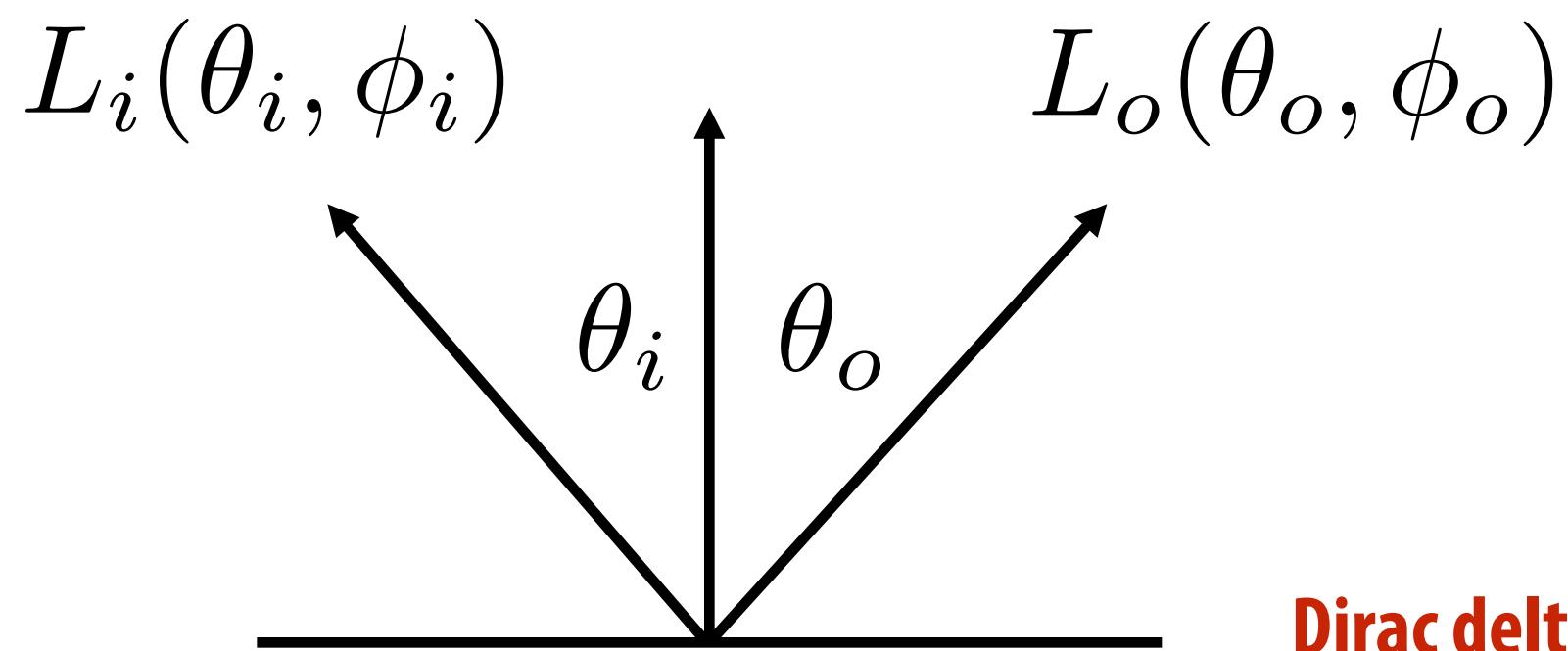
**Top-down view  
(looking down on surface)**



$$\phi_o = -\phi_i$$

$$\omega_o = -\omega_i + 2(\omega_i \cdot \vec{n})\vec{n}$$

# Specular reflection BRDF



$$L_o(\theta_o, \phi_o) = L_i(\theta_i, \phi_i)$$

$$f_r(\theta_i, \phi_i; \theta_o, \phi_o) = \frac{\delta(\cos \theta_i - \cos \theta_o)}{\cos \theta_i} \delta(\phi_i - \phi_o \pm \pi)$$

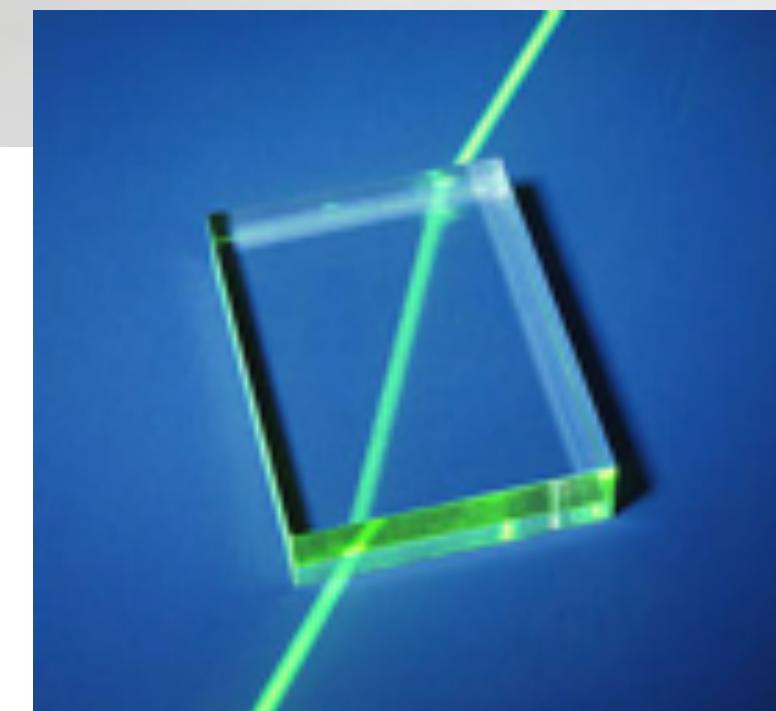
- Strictly speaking,  $f_r$  is a distribution, not a function
- In practice, no hope of finding reflected direction via random sampling; simply pick the reflected direction!



# Transmission

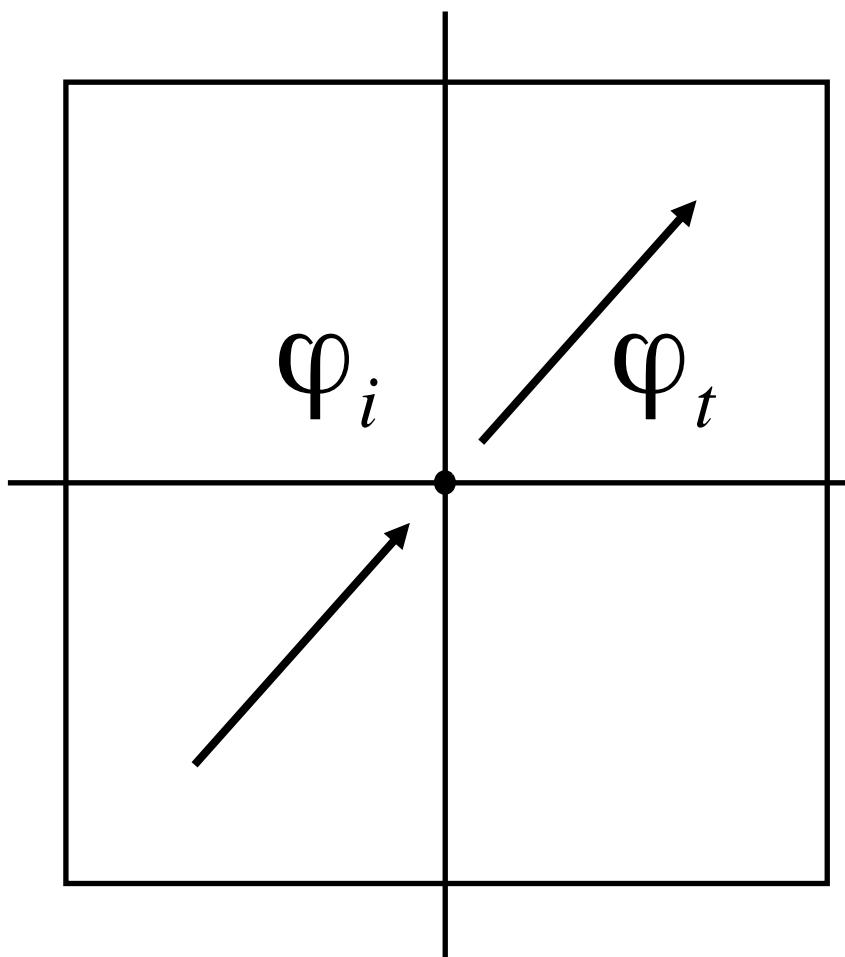
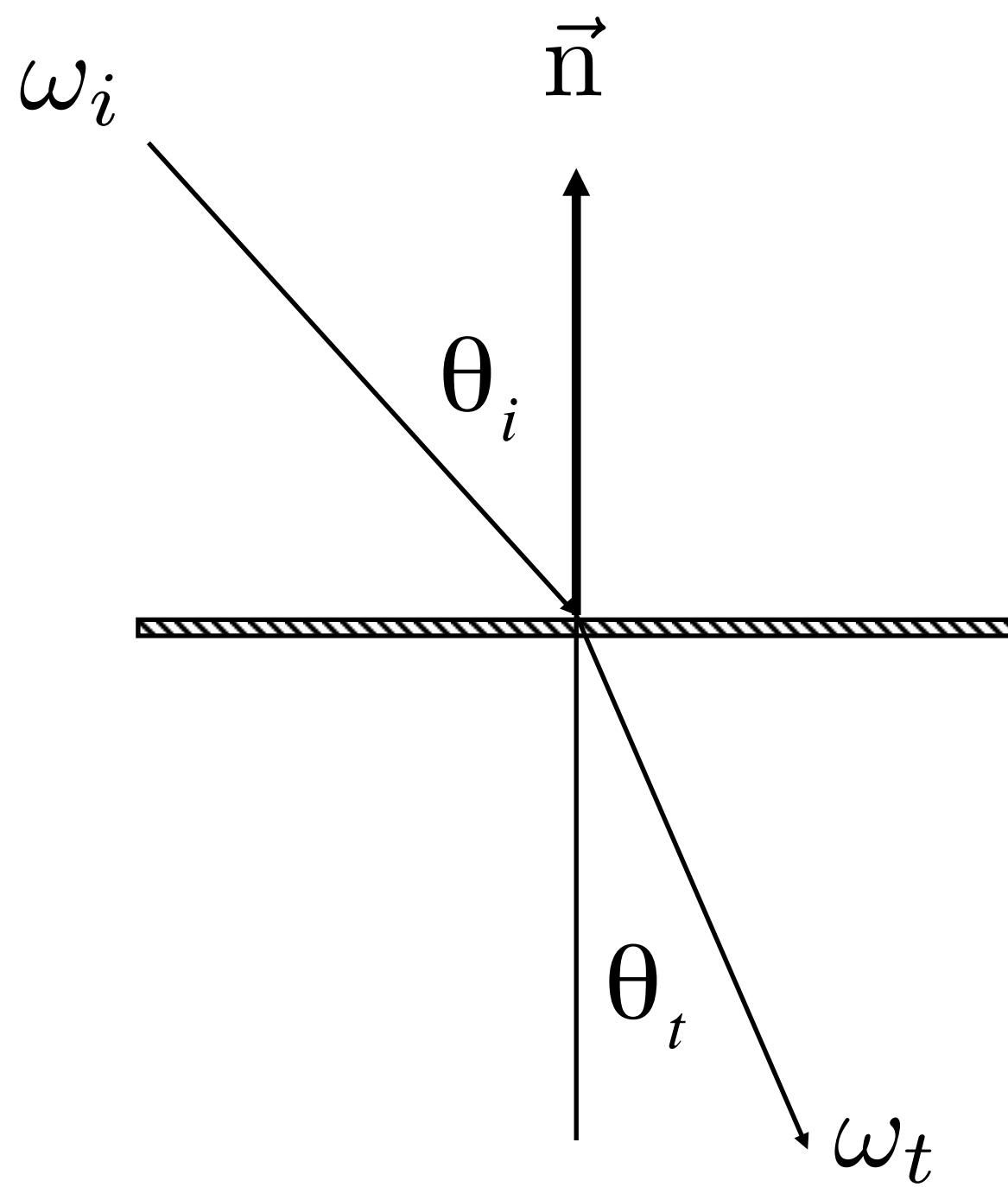
In addition to reflecting off surface, light may be transmitted through surface.

Light refracts when it enters a new medium.



# Snell's Law

Transmitted angle depends on relative index of refraction of material ray is leaving/entering.



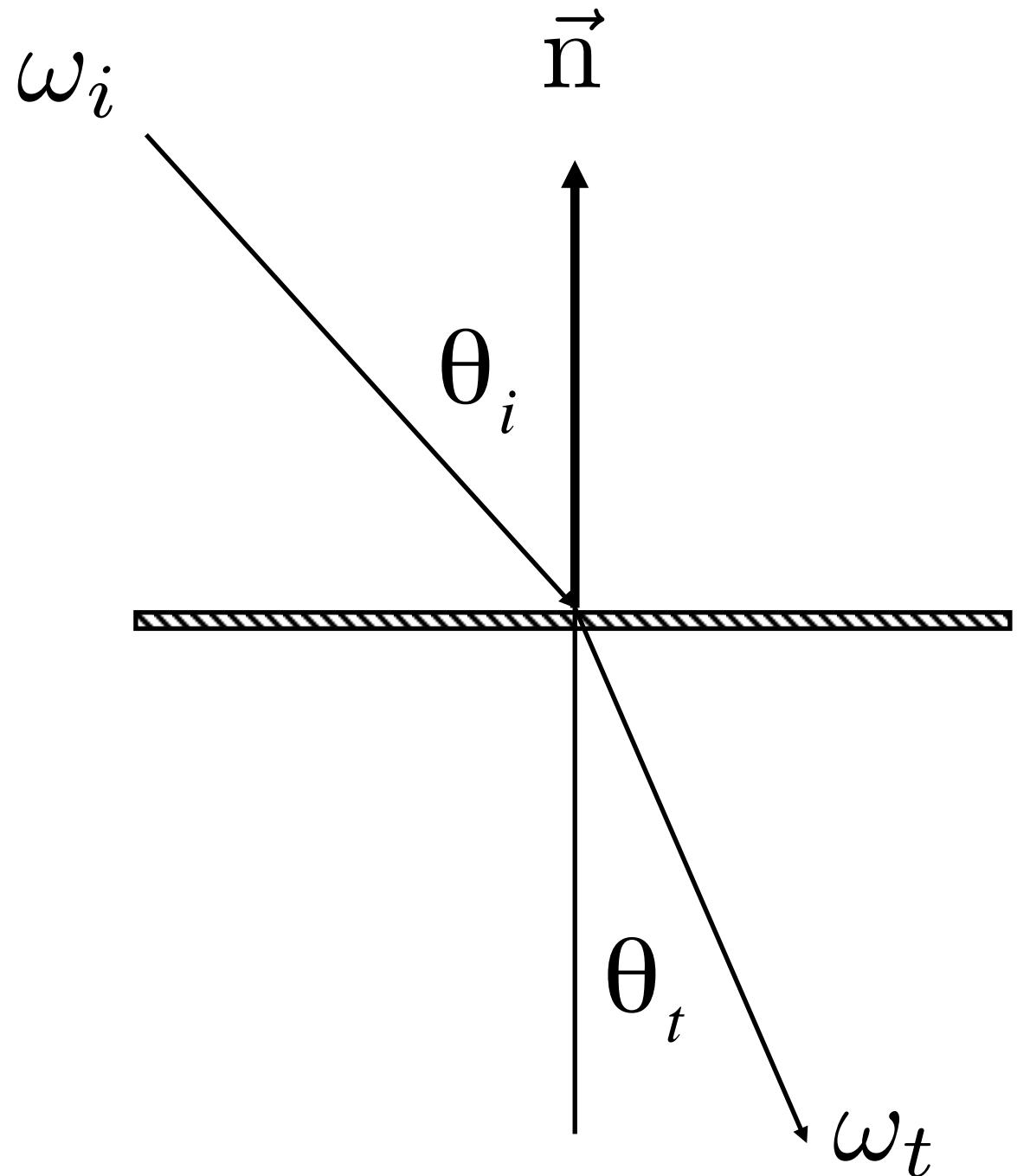
Medium	$\eta^*$
Vacuum	1.0
Air (sea level)	1.00029
Water (20°C)	1.333
Glass	1.5-1.6
Diamond	2.42

\* index of refraction is wavelength dependent (these are averages)

$$\eta_i \sin \theta_i = \eta_t \sin \theta_t$$

# Law of refraction

$$\eta_i \sin \theta_i = \eta_t \sin \theta_t$$



$$\cos \theta_t = \sqrt{1 - \sin^2 \theta_t}$$

$$= \sqrt{1 - \left(\frac{\eta_i}{\eta_t}\right)^2 \sin^2 \theta_i}$$

$$= \sqrt{1 - \left(\frac{\eta_i}{\eta_t}\right)^2 (1 - \cos^2 \theta_i)}$$

$$1 - \left(\frac{\eta_i}{\eta_t}\right)^2 (1 - \cos^2 \theta_i) < 0$$

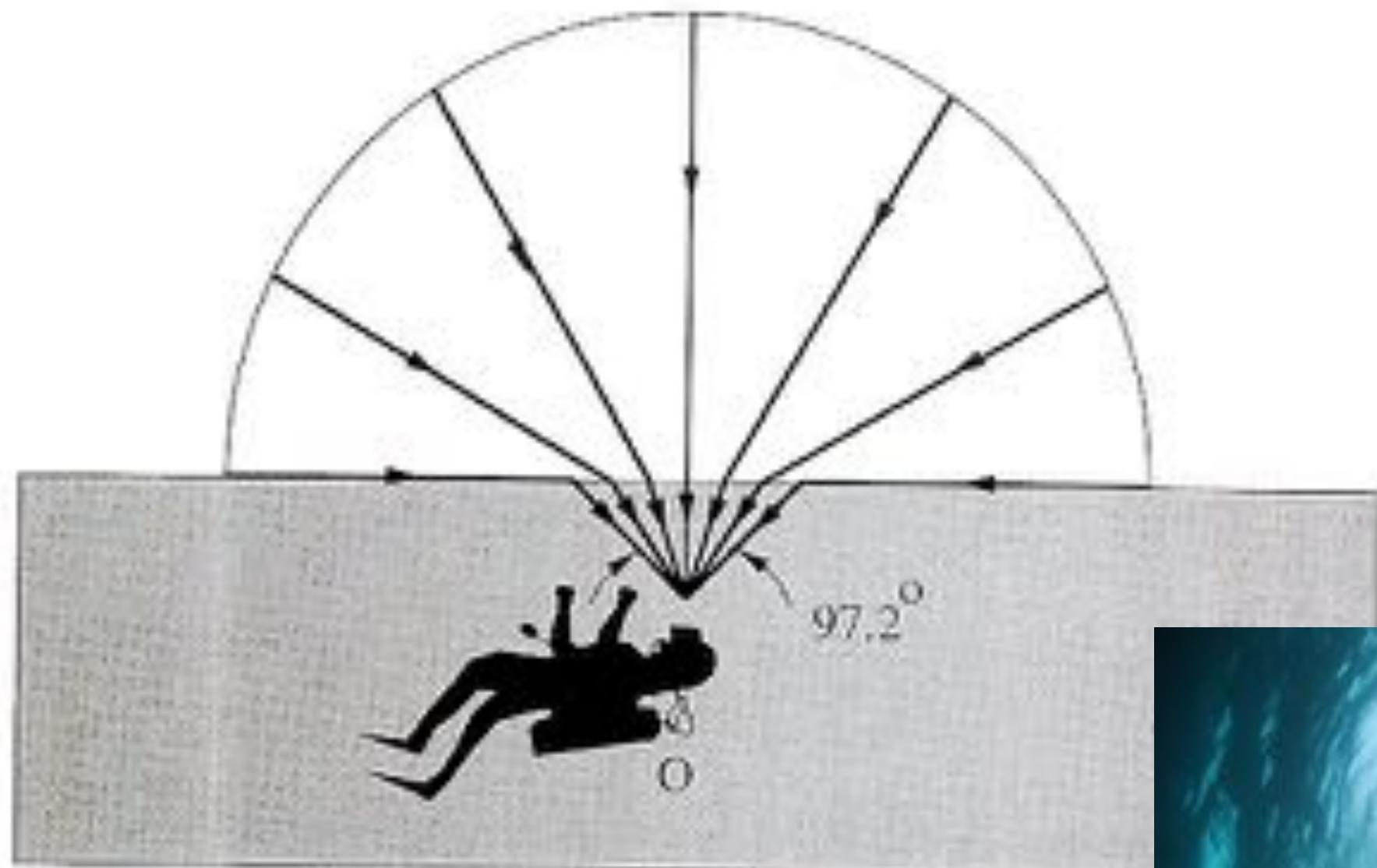
**Total internal reflection:**

When light is moving from a more optically dense medium to a less optically dense medium:  $\frac{\eta_i}{\eta_t} > 1$

Light incident on boundary from large enough angle will not exit medium.

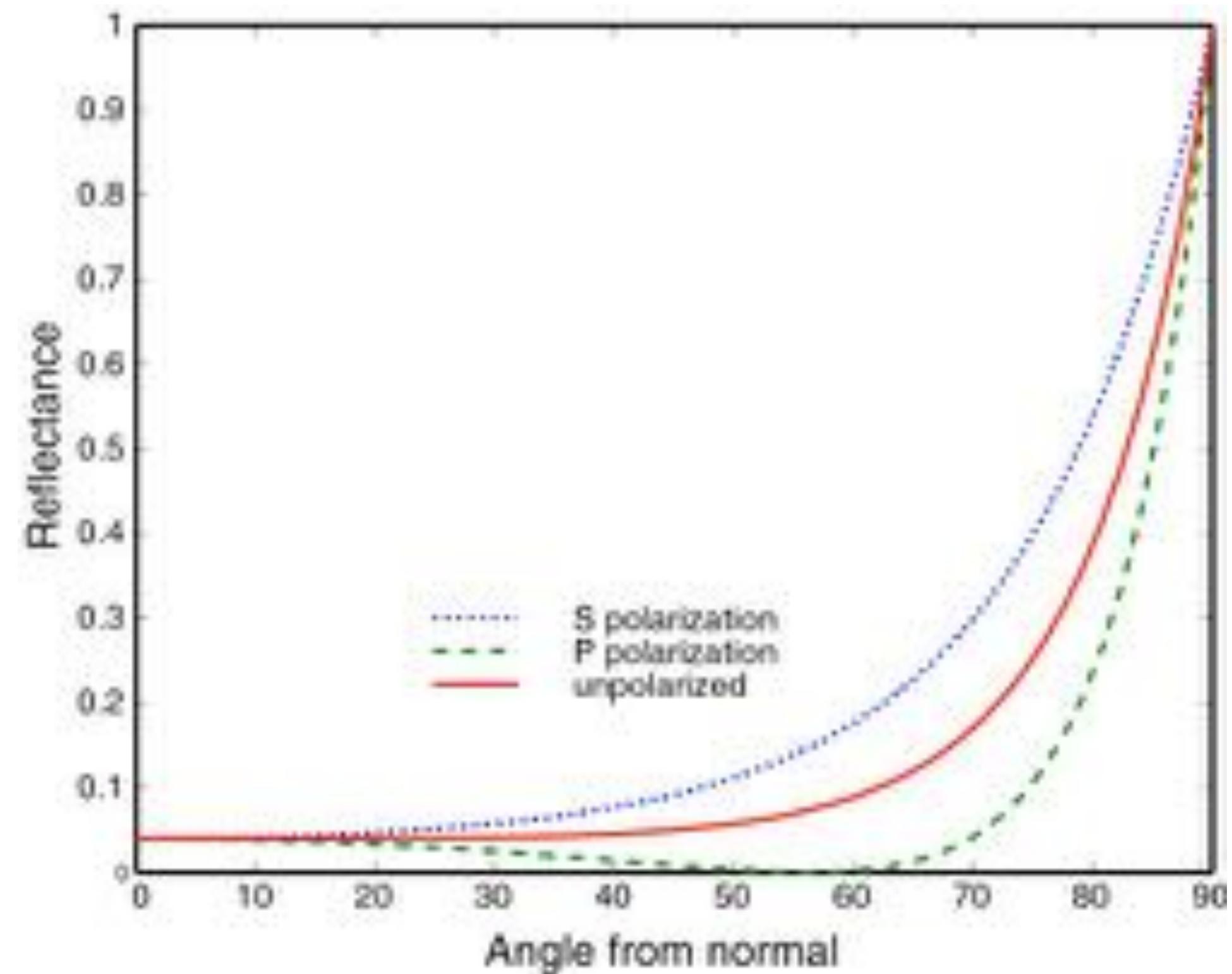
# Optical manhole

Only small “cone” visible, due to total internal reflection (TIR)



# Fresnel reflection

Many real materials:  
reflectance increases w/  
viewing angle



[Lafontaine et al. 1997]

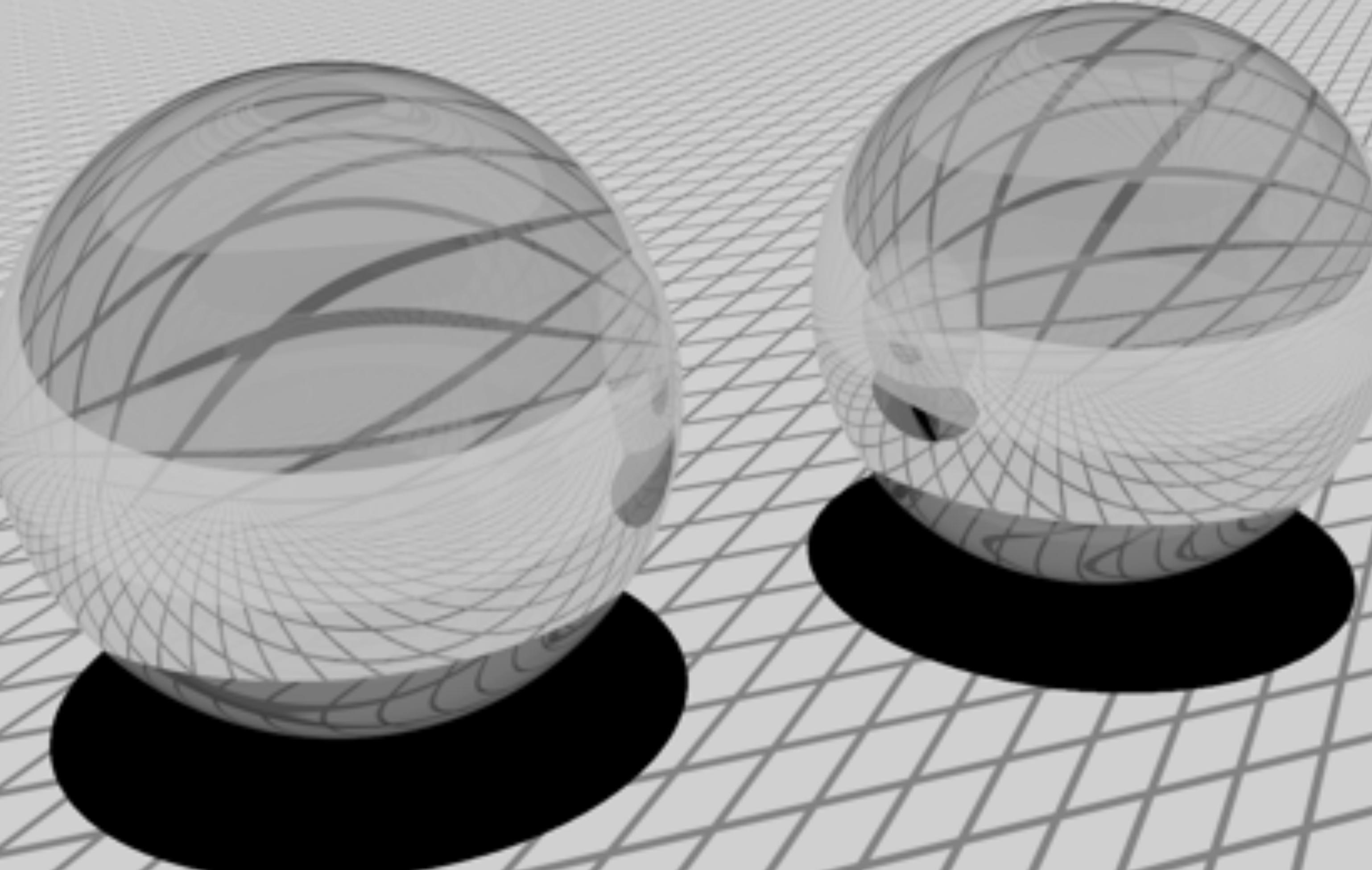
# Snell + Fresnel: Example



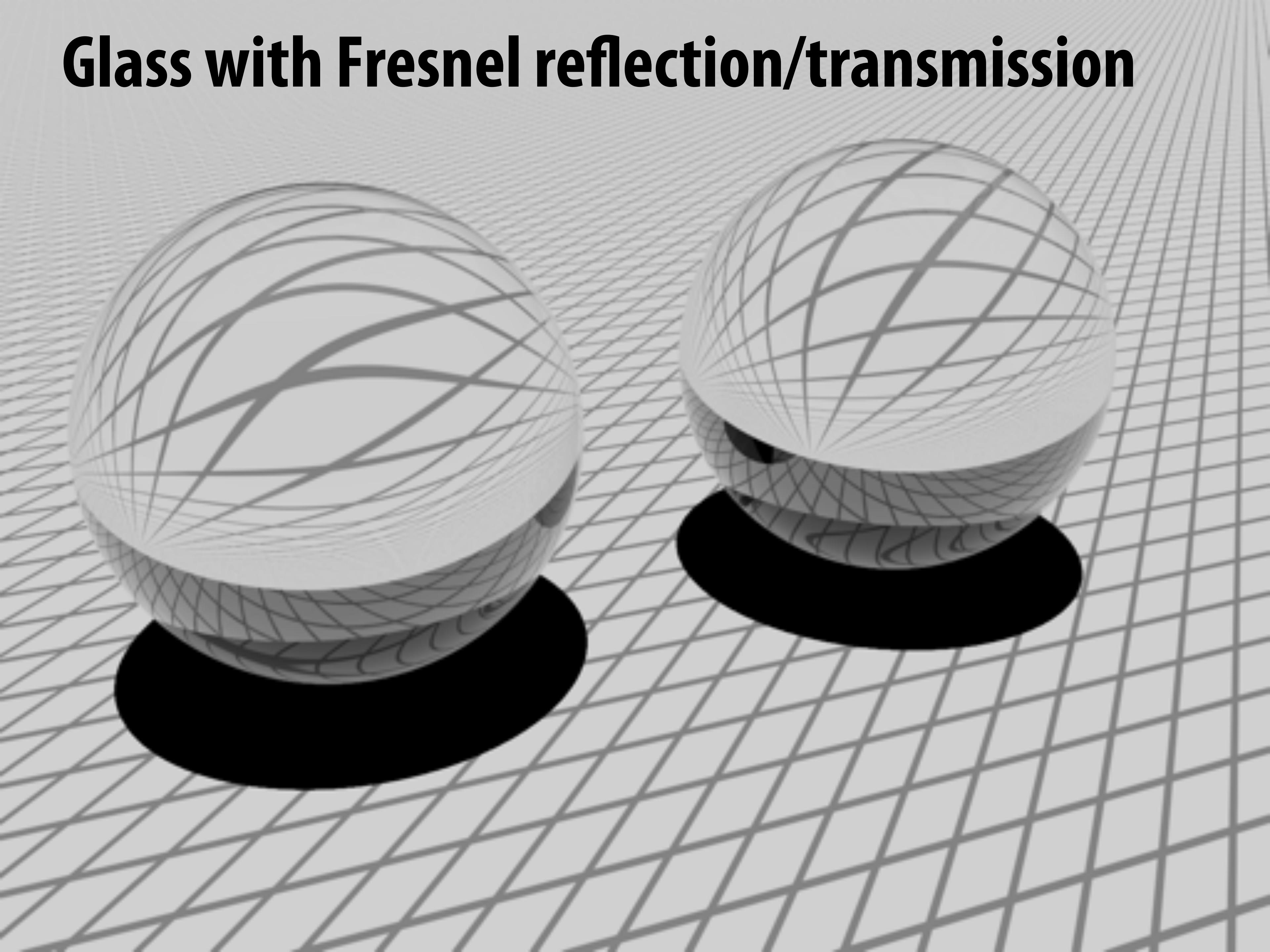
Refraction (Snell)

Reflection (Fresnel)

# Without Fresnel (fixed reflectance/transmission)



# Glass with Fresnel reflection/transmission

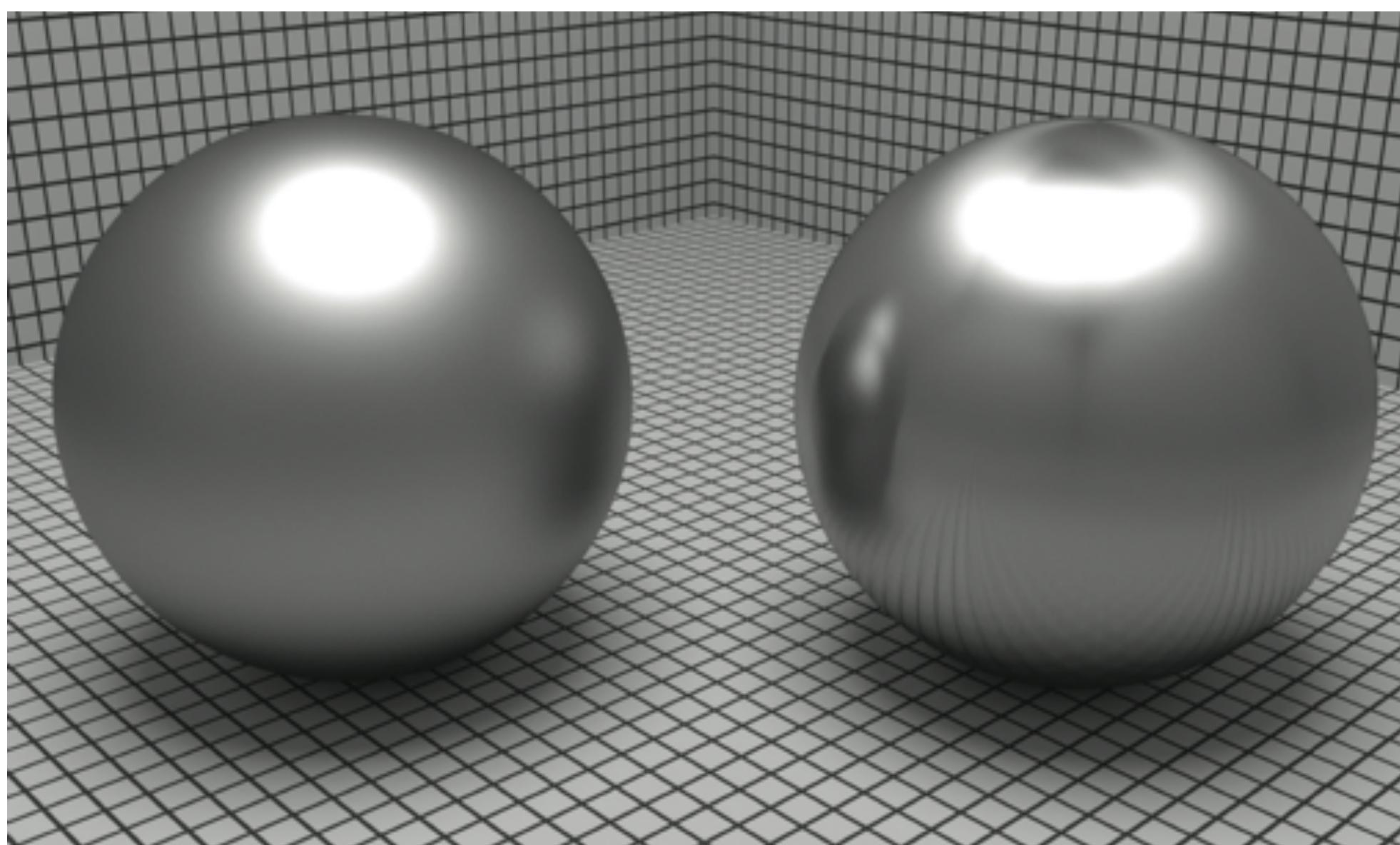


# Anisotropic reflection

Reflection depends on azimuthal angle  $\phi$



Results from oriented microstructure of surface  
e.g., brushed metal



# Translucent materials: Jade



# Translucent materials: skin

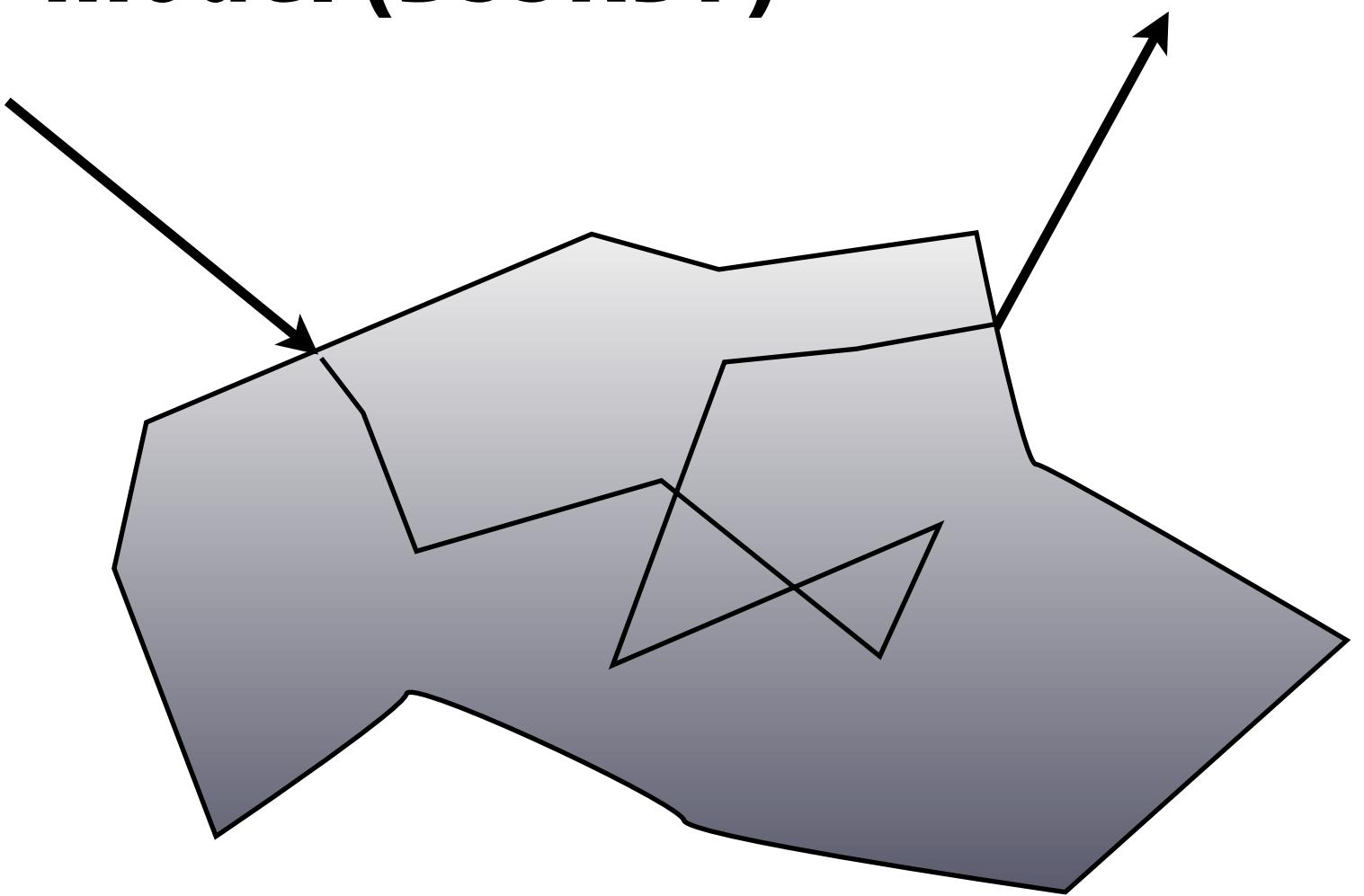


# Translucent materials: leaves



# Subsurface scattering

- Visual characteristics of many surfaces caused by light entering at different points than it exits
  - Violates a fundamental assumption of the BRDF
  - Need to generalize scattering model (BSSRDF)



[Jensen et al 2001]



[Donner et al 2008]

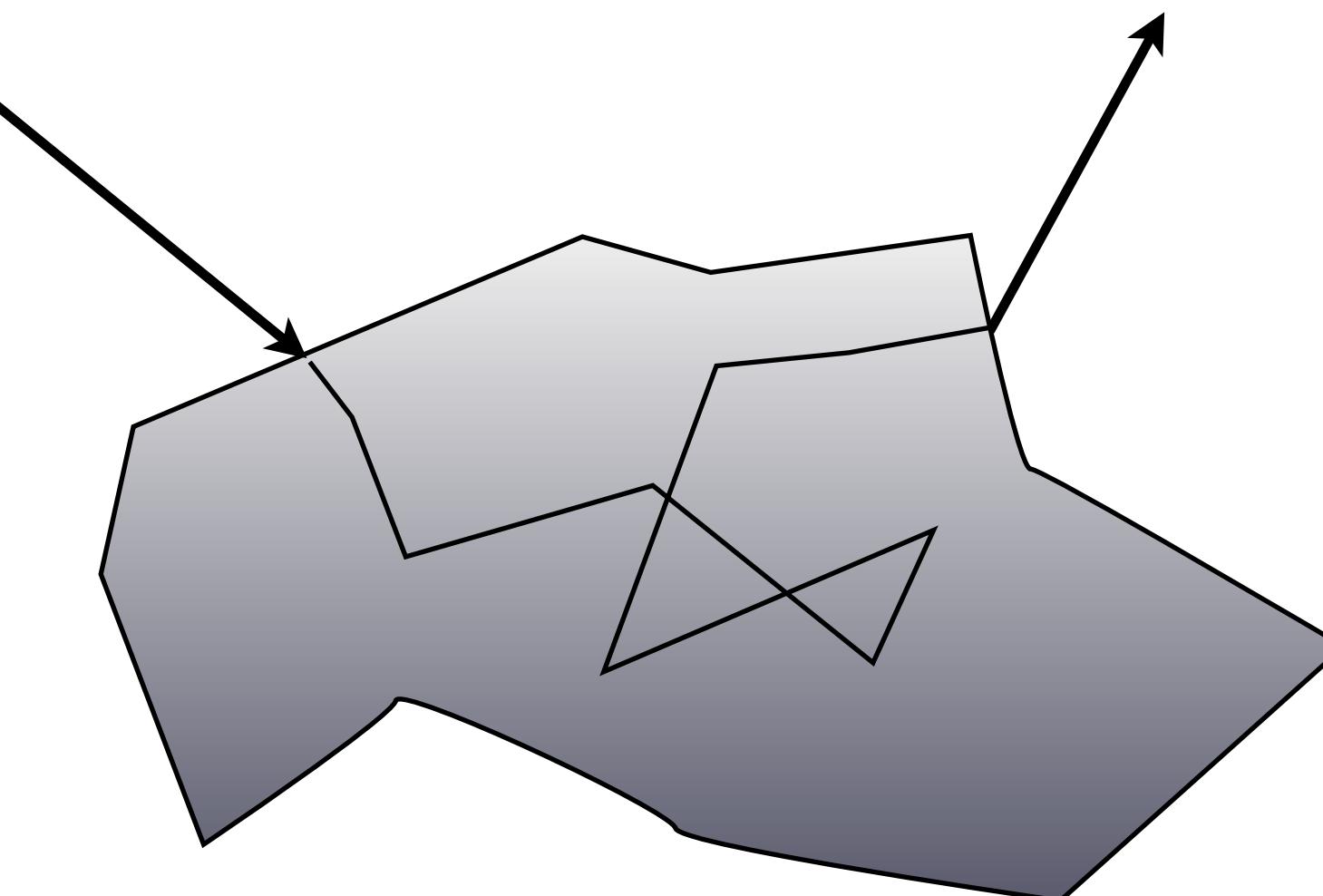
# Scattering functions

- Generalization of BRDF; describes exitant radiance at one point due to incident differential irradiance at another point:

$$S(x_i, \omega_i, x_o, \omega_o)$$

- Generalization of reflection equation integrates over all points on the surface and all directions(!)

$$L(x_o, \omega_o) = \int_A \int_{H^2} S(x_i, \omega_i, x_o, \omega_o) L_i(x_i, \omega_i) \cos \theta_i d\omega_i dA$$



# BRDF



# BSSRDF

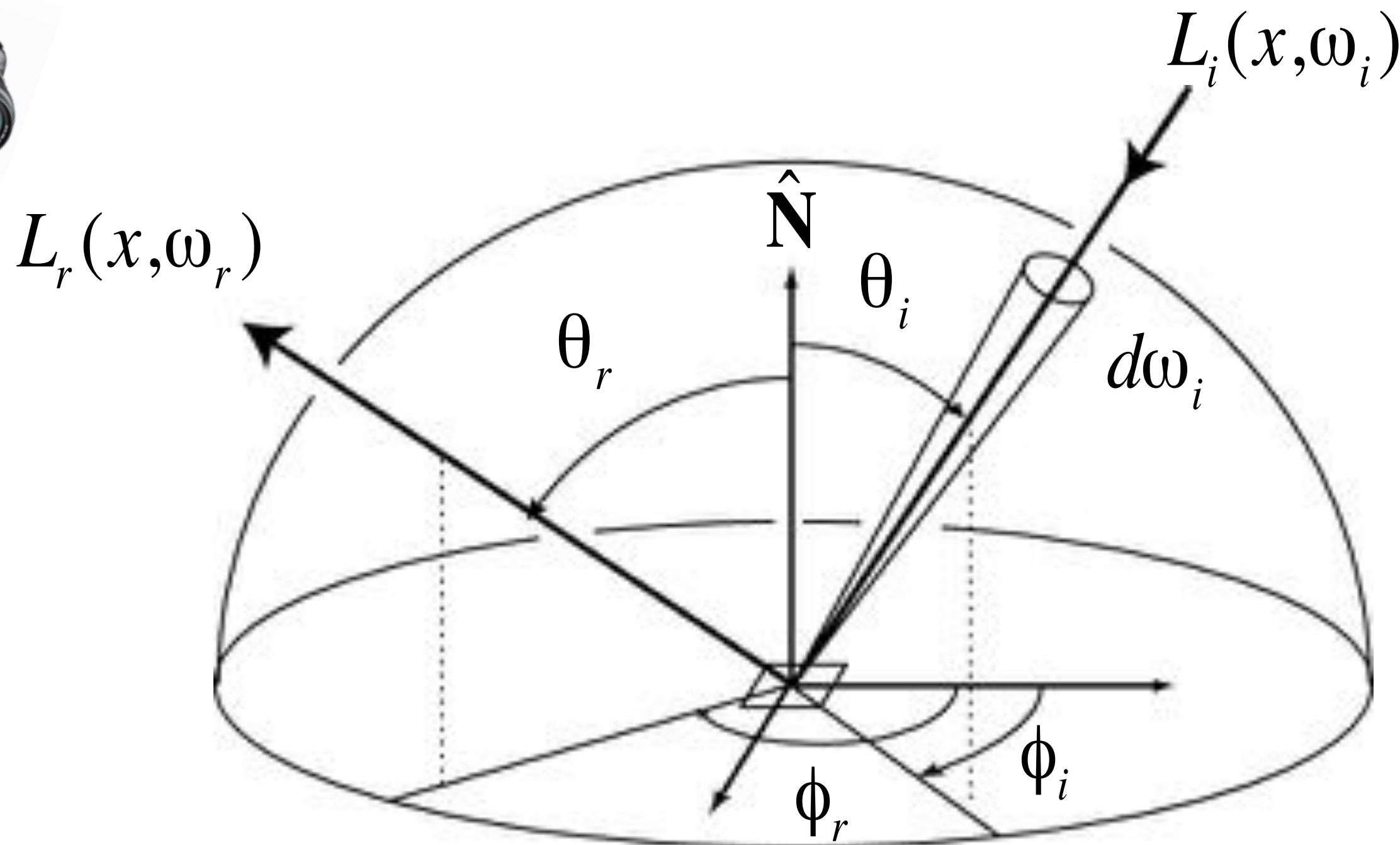


**Ok, so scattering is complicated!**

**What's a (relatively simple) algorithm  
that can capture all this behavior?**

**We start by returning to reflection  
without scattering (using the BRDF)**

# The reflection equation



$$dL_r(\omega_r) = f_r(\omega_i \rightarrow \omega_r) dL_i(\omega_i) \cos \theta_i$$

$$L_r(p, \omega_r) = \int_{H^2} f_r(p, \omega_i \rightarrow \omega_r) L_i(p, \omega_i) \cos \theta_i d\omega_i$$

# The reflection equation

- Key piece of overall rendering equation:

$$L_r(p, \omega_r) = \int_{H^2} f_r(p, \omega_i \rightarrow \omega_r) L_i(p, \omega_i) \cos \theta_i d\omega_i$$

- Approximate integral via Monte Carlo integration
- Generate directions  $\omega_j$  sampled from some distribution  $p(\omega)$
- Compute the estimator

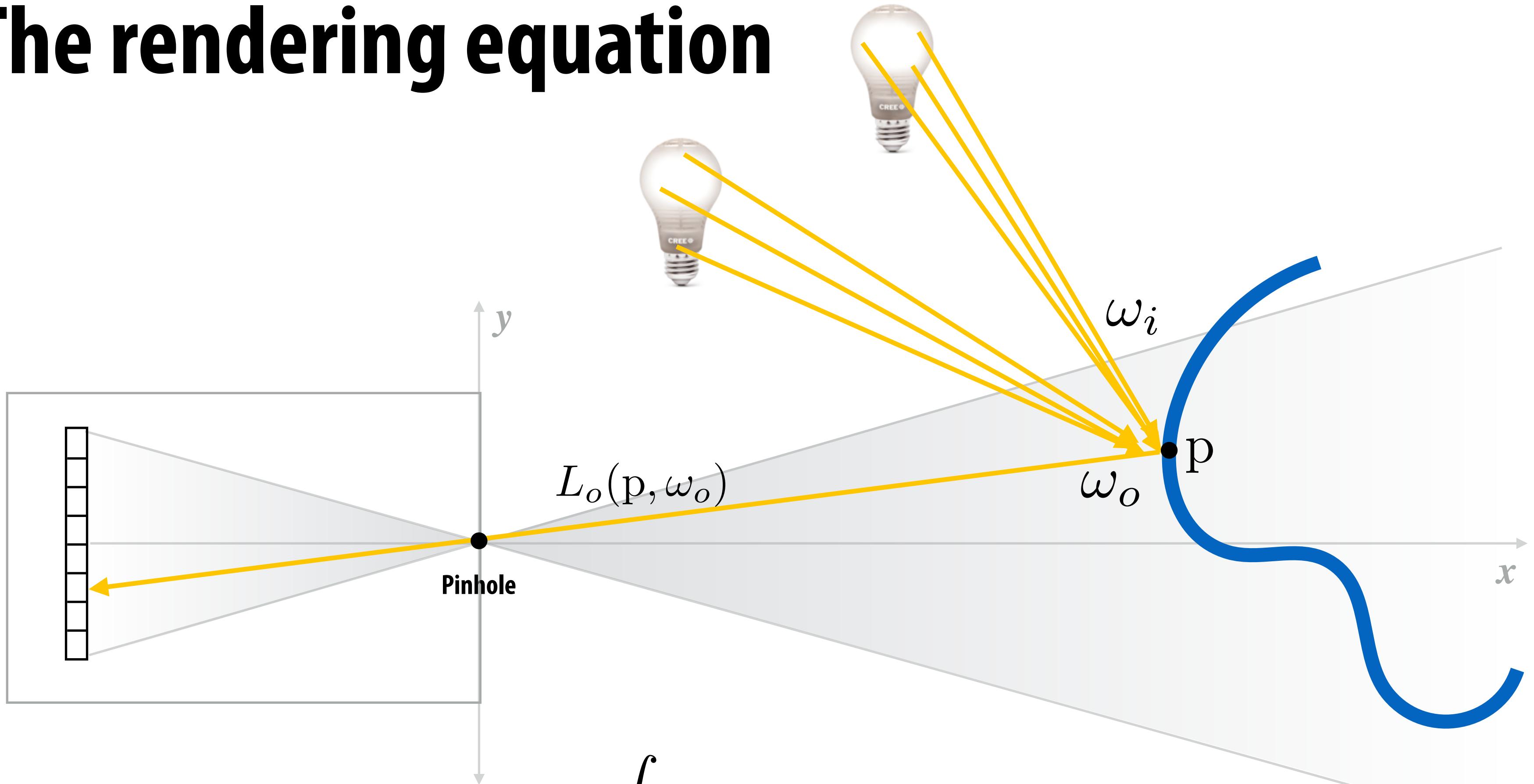
$$\frac{1}{N} \sum_{j=1}^N \frac{f_r(p, \omega_j \rightarrow \omega_r) L_i(p, \omega_j) \cos \theta_j}{p(\omega_j)}$$

- To reduce variance  $p(\omega)$  should match BRDF or incident radiance function

# Estimating reflected light

```
// Assume:  
// Ray ray hits surface at point hit_p  
// Normal of surface at hit point is hit_n  
  
Vector3D wr = -ray.d;    // outgoing direction  
Spectrum Lr = 0.;  
for (int i = 0; i < N; ++i) {  
    Vector3D wi;          // sample incident light from this direction  
    float pdf;             // p(wi)  
  
    generate_sample(brdf, &wi, &pdf);    // generate sample according to brdf  
  
    Spectrum f = brdf->f(wr, wi);  
    Spectrum Li = trace_ray(Ray(hit_p, wi)); // compute incoming Li  
    Lr += f * Li * fabs(dot(wi, hit_n)) / pdf;  
}  
return Lr / N;
```

# The rendering equation



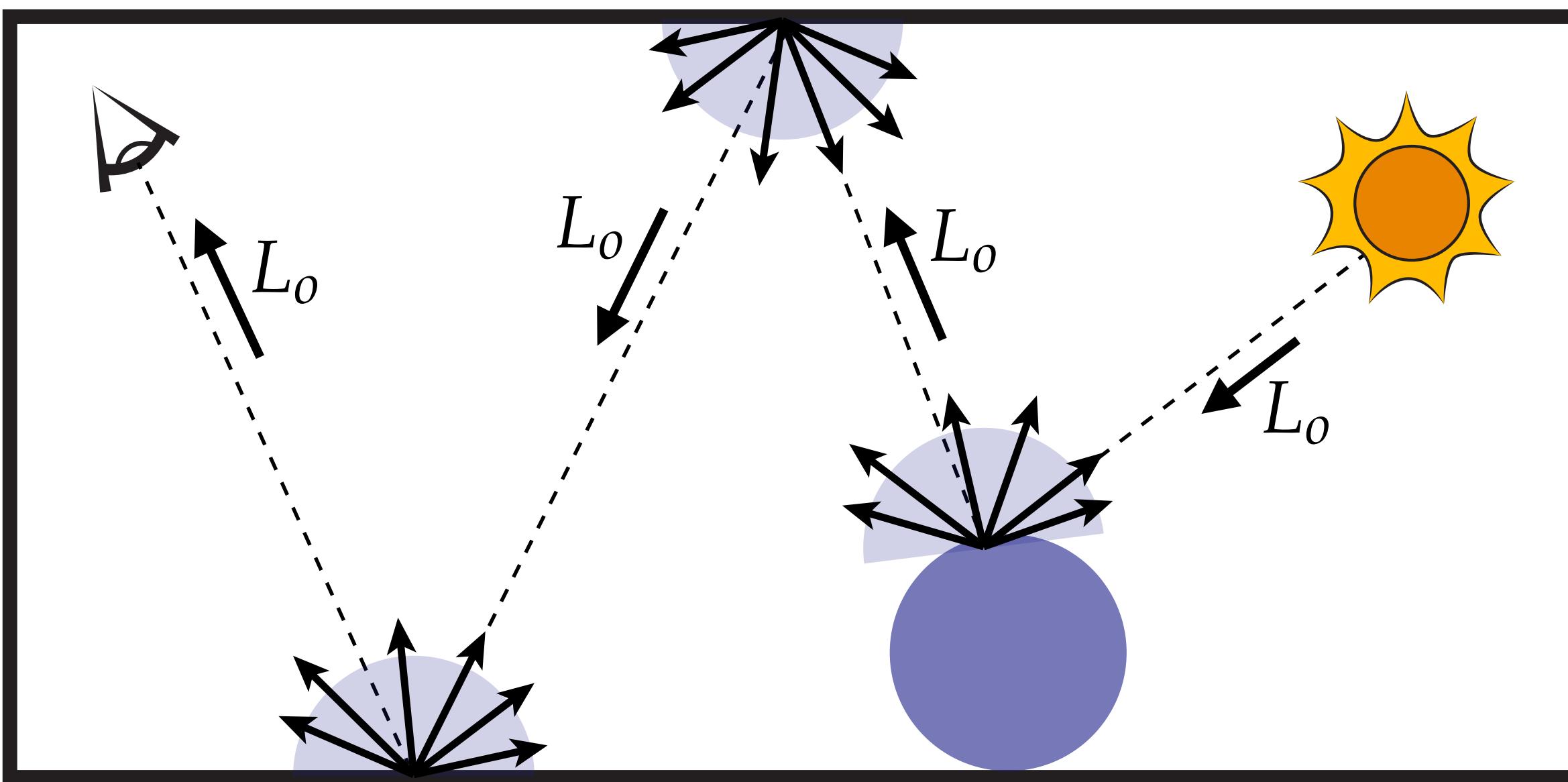
$$L_o(p, \omega_o) = L_e(p, \omega_o) + \int_{H^2} f_r(p, \omega_i \rightarrow \omega_o) L_i(p, \omega_i) \cos \theta_i d\omega_i$$

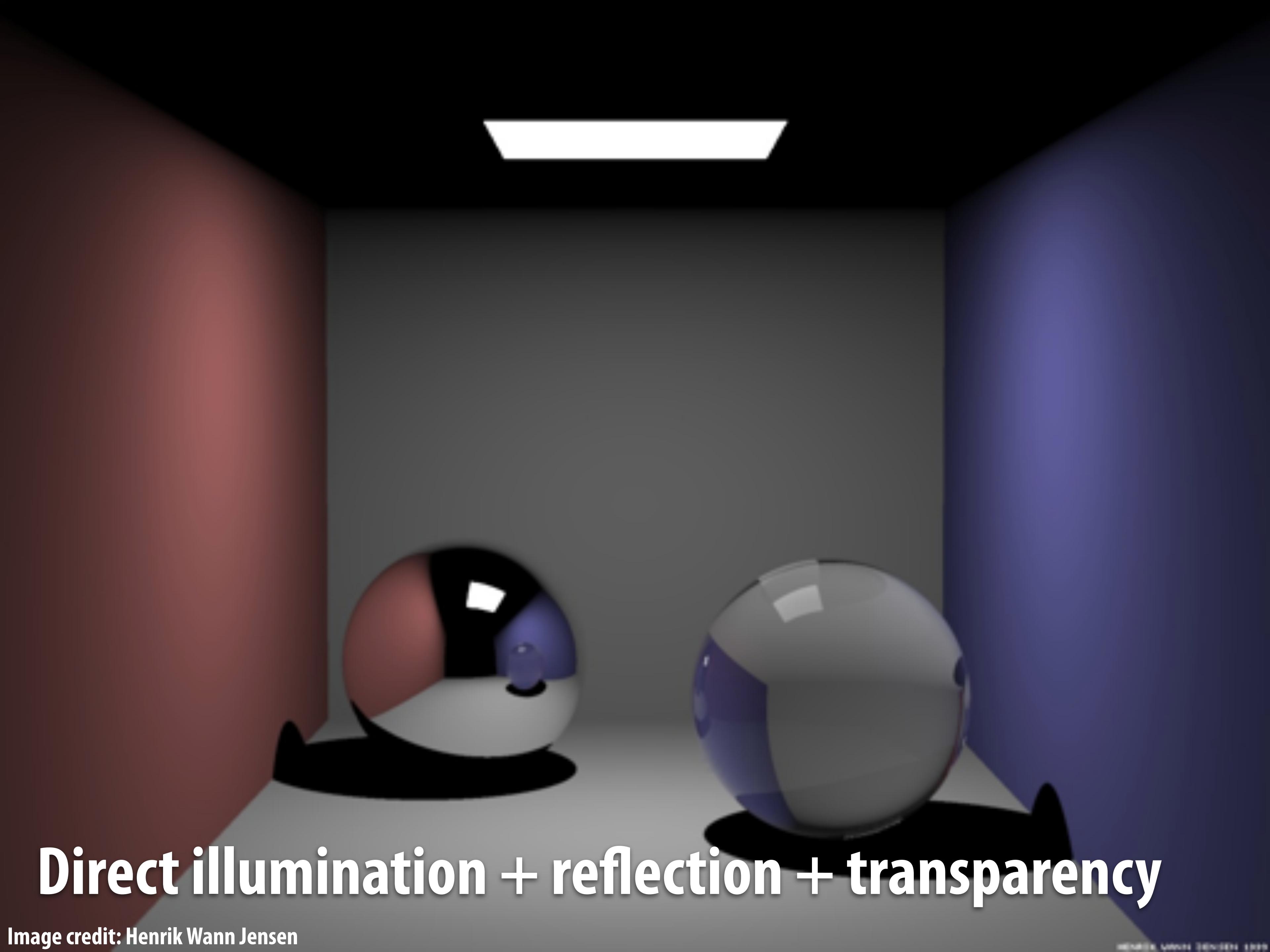
Now that we know how to handle reflection, how do we solve the full rendering equation? Have to determine incident radiance...

**Key idea in (efficient) rendering: take advantage of special knowledge to break up integration into “easier” components.**

# Path tracing: overview

- Partition the rendering equation into direct and indirect illumination
- Use Monte Carlo to estimate each partition separately
  - One sample for each
  - Assumption: 100s of samples per pixel
- Terminate paths with Russian roulette





**Direct illumination + reflection + transparency**

# Global illumination solution

Image credit: Henrik Wann Jensen

HENRIK WANN JENSEN 2000

# Next Time: Monte Carlo integration



$$\int_{\Omega} f(p) \, dp \approx \text{vol}(\Omega) \frac{1}{N} \sum_{i=1}^N f(X_i)$$

# **Monte Carlo Rendering**

---

**Computer Graphics  
CMU 15-462/15-662**

# TODAY: Monte Carlo Rendering

- How do we render a photorealistic image?
- Put together many of the ideas we've studied:
  - color
  - materials
  - radiometry
  - numerical integration
  - geometric queries
  - spatial data structures
  - rendering equation
- Combine into final Monte Carlo ray tracing algorithm
- Alternative to rasterization, lets us generate much more realistic images (usually at much greater cost...)



# Photorealistic Rendering—Basic Goal

What are the **INPUTS** and **OUTPUTS**?

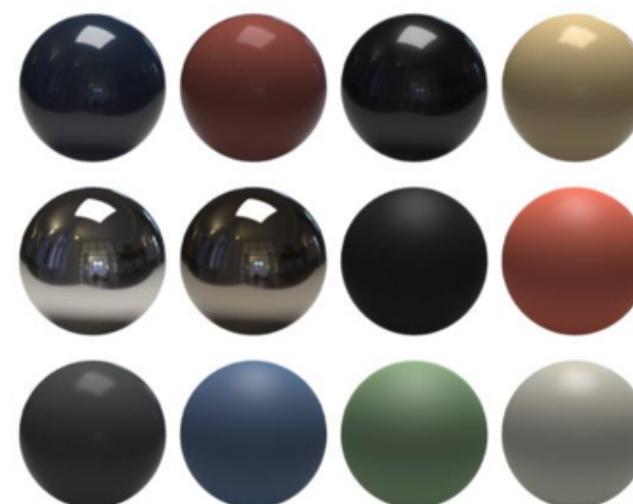
camera



geometry



materials



lights



(“**scene**”)

Ray Tracer

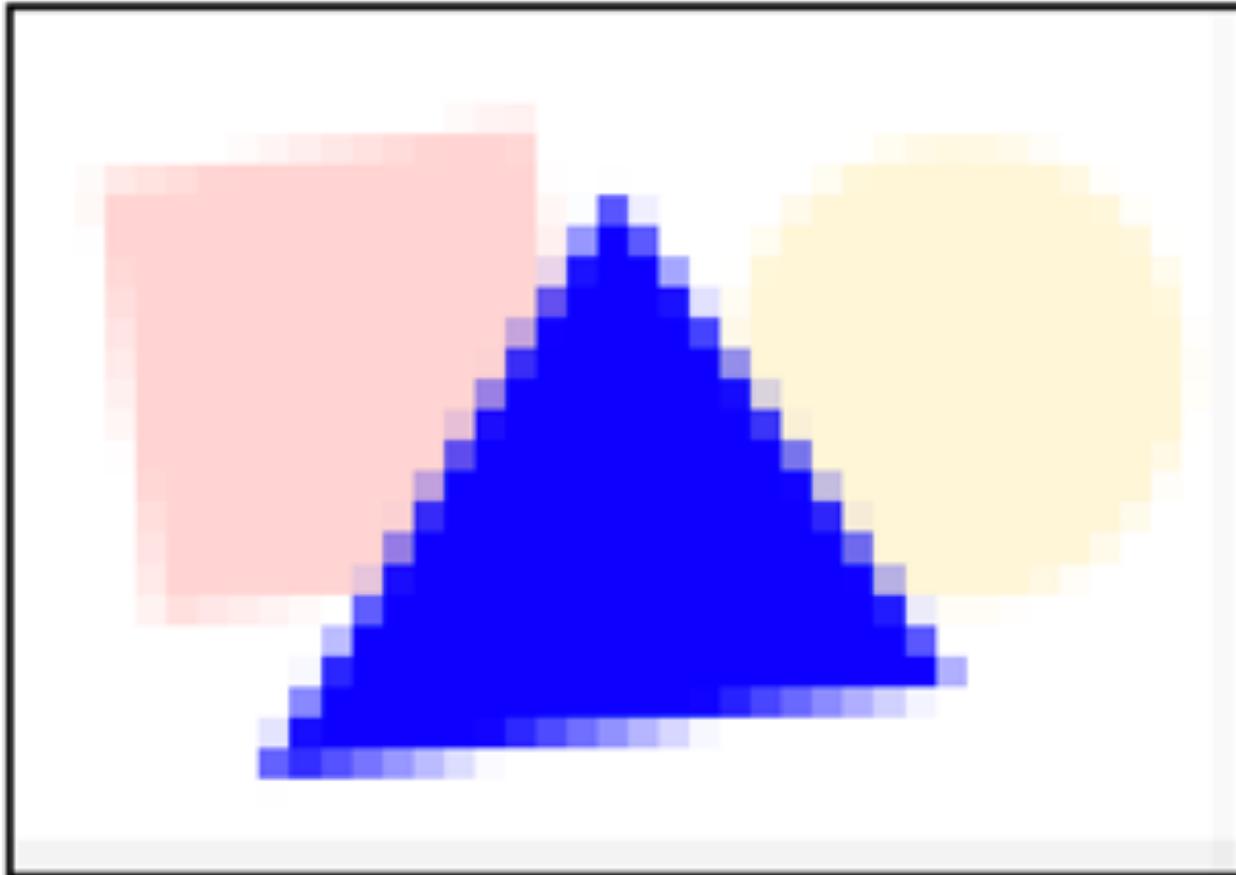


image

# Ray Tracing vs. Rasterization—Order

- Both rasterization & ray tracing will generate an image
- What's the difference?
- One basic difference: order in which we process samples

RASTERIZATION



for each primitive:

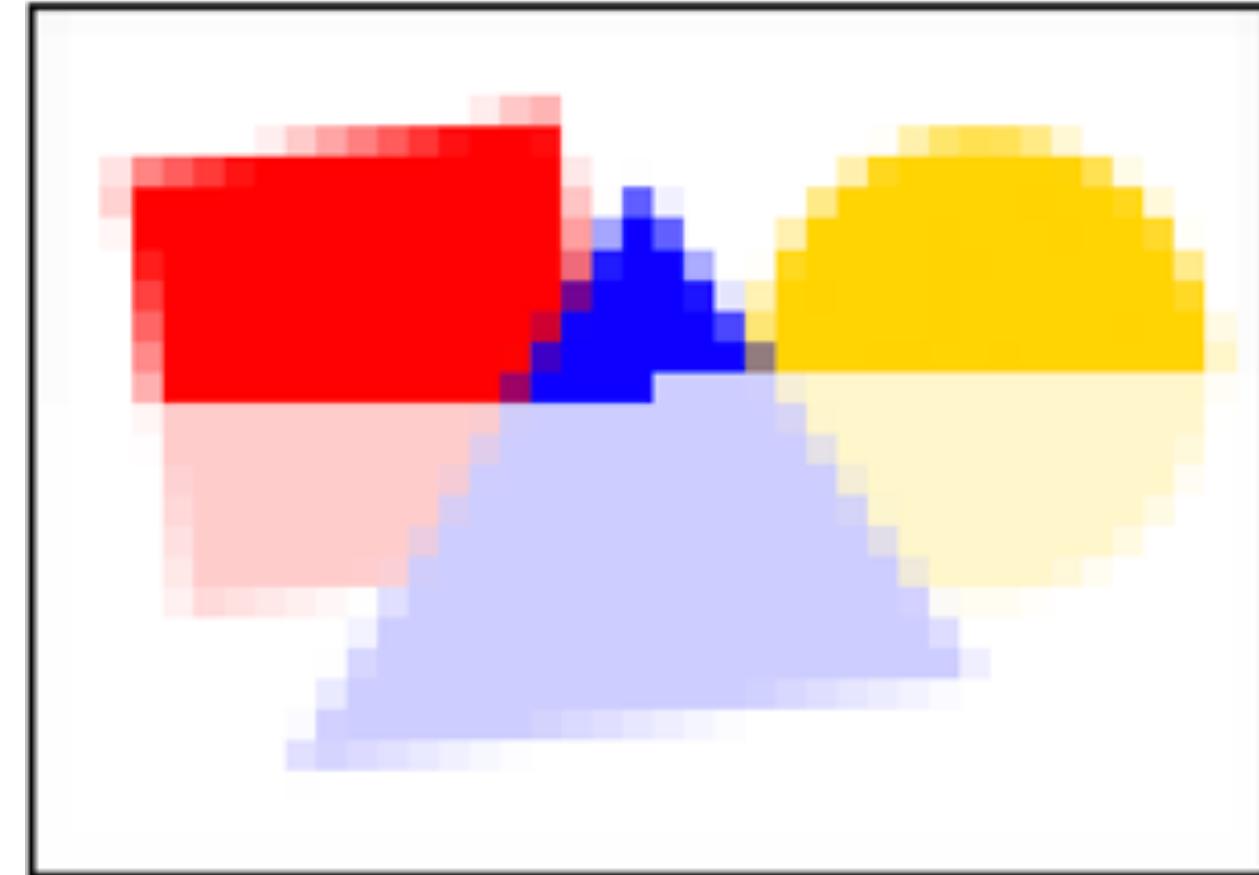
    for each sample:

        determine coverage

        evaluate color

(Use Z-buffer to determine  
which primitive is visible)

RAY TRACING



for each sample:

    for each primitive:

        determine coverage

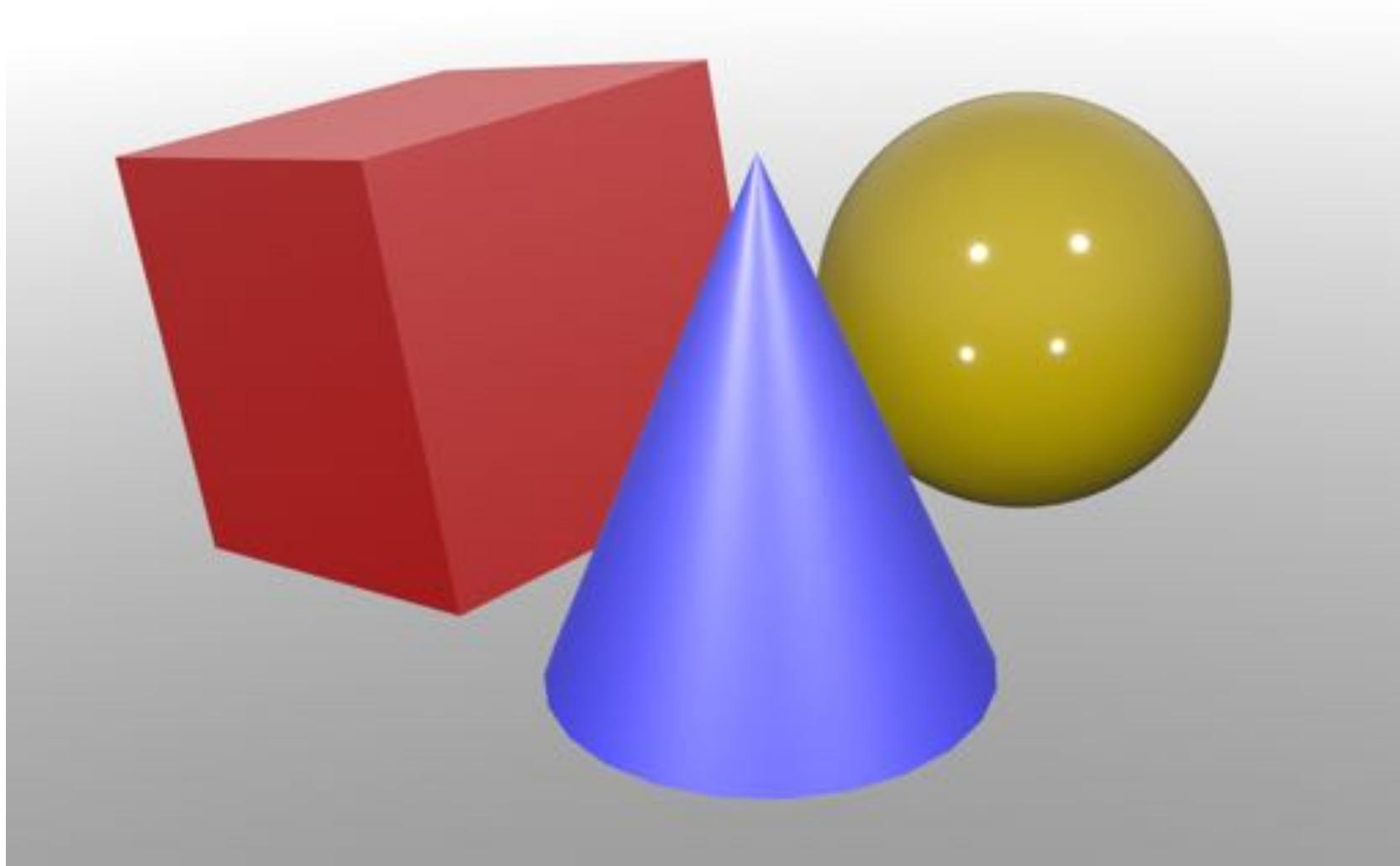
        evaluate color

(Use spatial data structure like BVH to  
determine which primitive is visible)

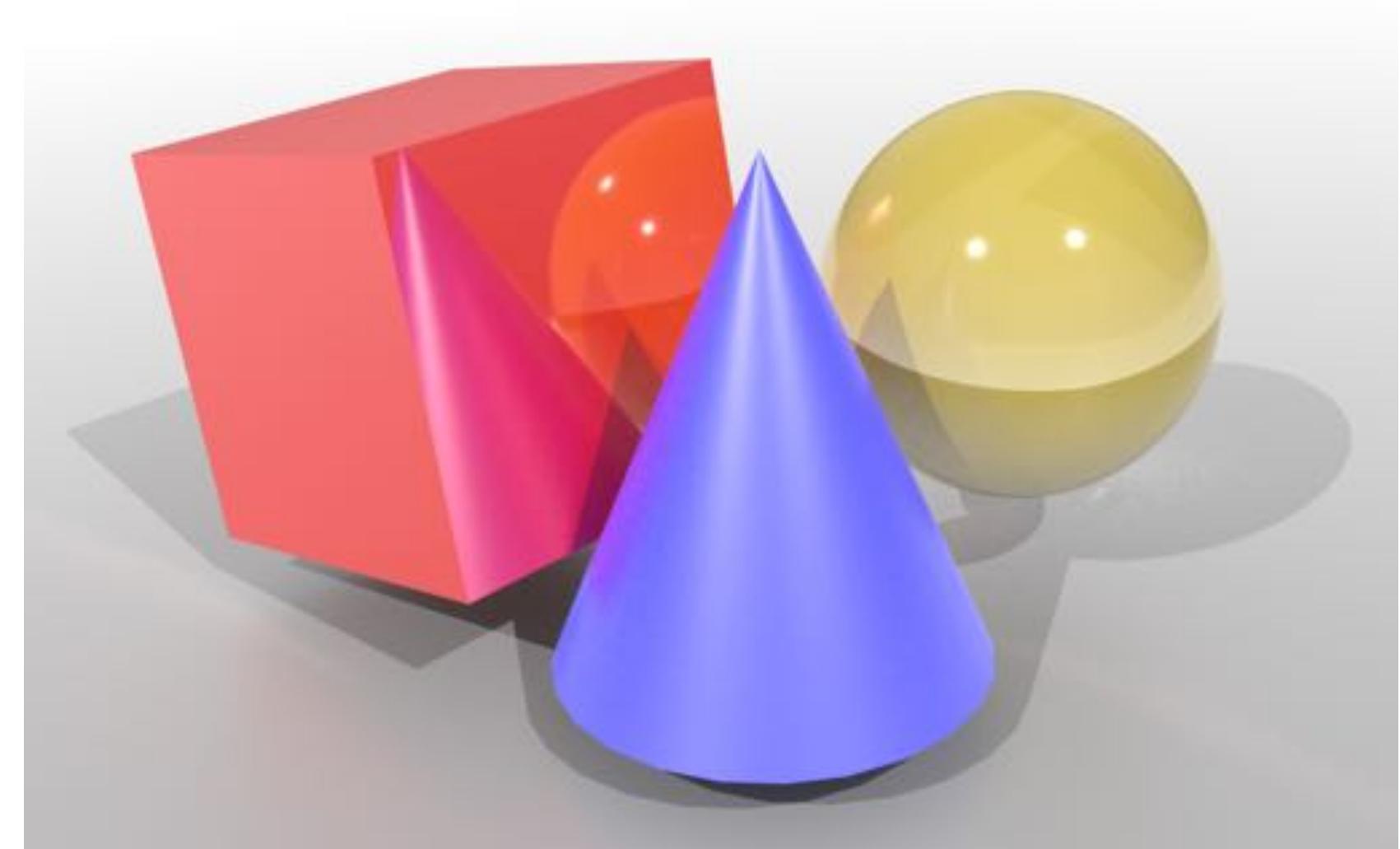
# Ray Tracing vs. Rasterization—Illumination

- More major difference: sophistication of illumination model
  - [LOCAL] rasterizer processes one primitive at a time; hard\* to determine things like “A is in the shadow of B”
  - [GLOBAL] ray tracer processes on ray at a time; ray knows about everything it intersects, easy to talk about shadows & other “global” illumination effects

RASTERIZATION



RAY TRACING



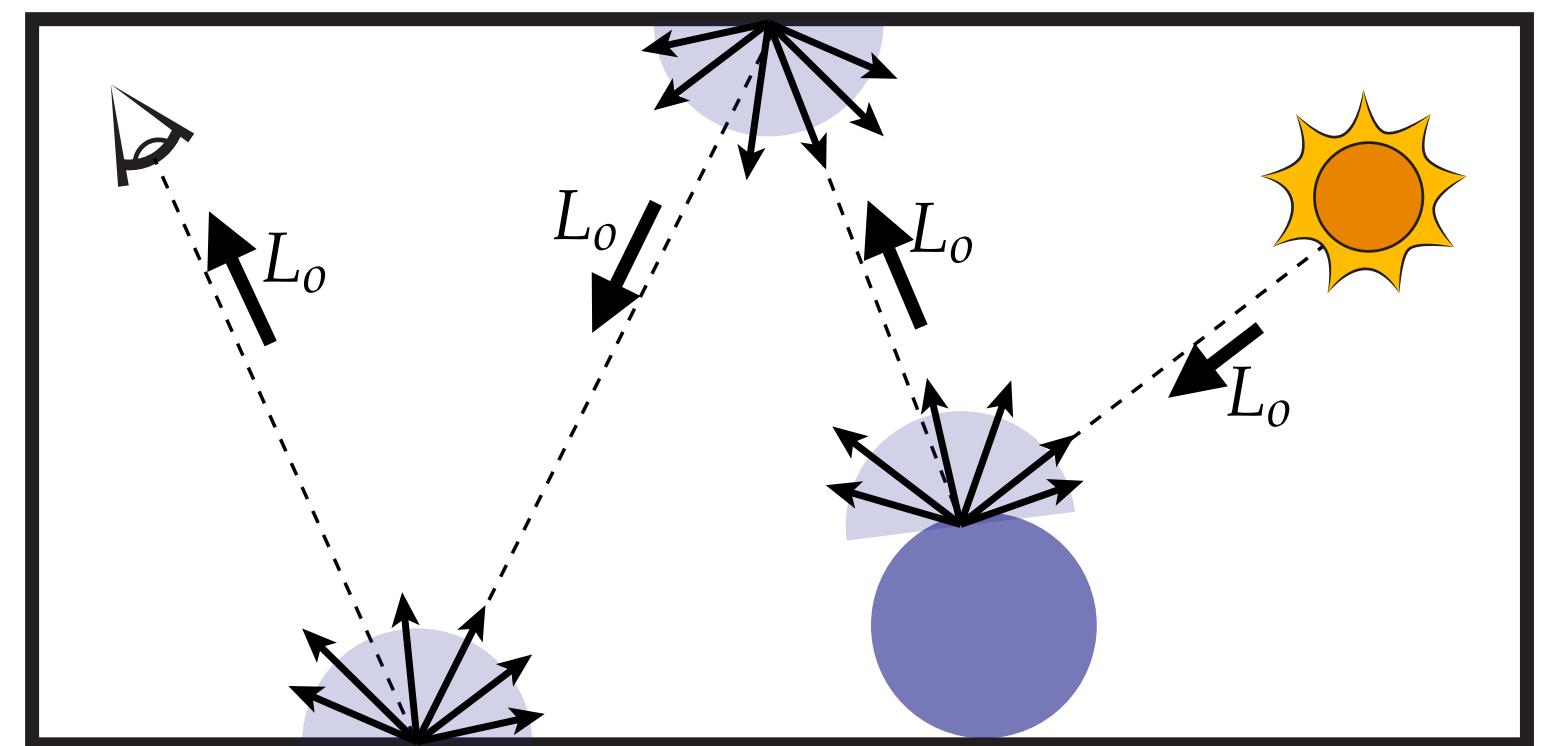
**Q: What illumination effects are missing from the image on the left?**

\*But not impossible to do some things with rasterization (e.g., shadow maps)... just results in more complexity

# Monte Carlo Ray Tracing

- To develop a full-blown photorealistic ray tracer, will need to apply Monte Carlo integration to the rendering equation
- To determine color of each pixel, integrate incoming light
- What function are we integrating?
  - illumination along different paths of light
- What does a “sample” mean in this context?
  - each path we trace is a sample

$$L_o(\mathbf{p}, \omega_o) = L_e(\mathbf{p}, \omega_o) + \int_{\mathcal{H}^2} f_r(\mathbf{p}, \omega_i \rightarrow \omega_o) L_i(\mathbf{p}, \omega_i) \cos \theta d\omega_i$$



# Monte Carlo Integration

- Started looking at Monte Carlo integration in our lecture on numerical integration
- Basic idea: take average of random samples
- Will need to flesh this idea out with some key concepts:
  - EXPECTED VALUE — what value do we get on average?
  - VARIANCE — what's the expected deviation from the average?
  - IMPORTANCE SAMPLING — how do we (correctly) take more samples in more important regions?

$$\lim_{N \rightarrow \infty} \frac{|\Omega|}{N} \sum_{i=1}^N f(X_i) = \int_{\Omega} f(x) dx$$

# Expected Value

**Intuition: what value does a random variable take, on average?**

- E.g., consider a fair coin where heads = 1, tails = 0
- Equal probability of heads & is tails (1/2 for both)
- Expected value is then  $(1/2) \cdot 1 + (1/2) \cdot 0 = 1/2$

$$E(Y) := \sum_{i=1}^k p_i y_i$$

expected value of random variable Y

number of possible outcomes

probability of ith outcome

value of ith outcome

**Properties of expectation:**

$$E \left[ \sum_i Y_i \right] = \sum_i E[Y_i]$$
$$E[aY] = aE[Y]$$

**(Can you show these are true?)**

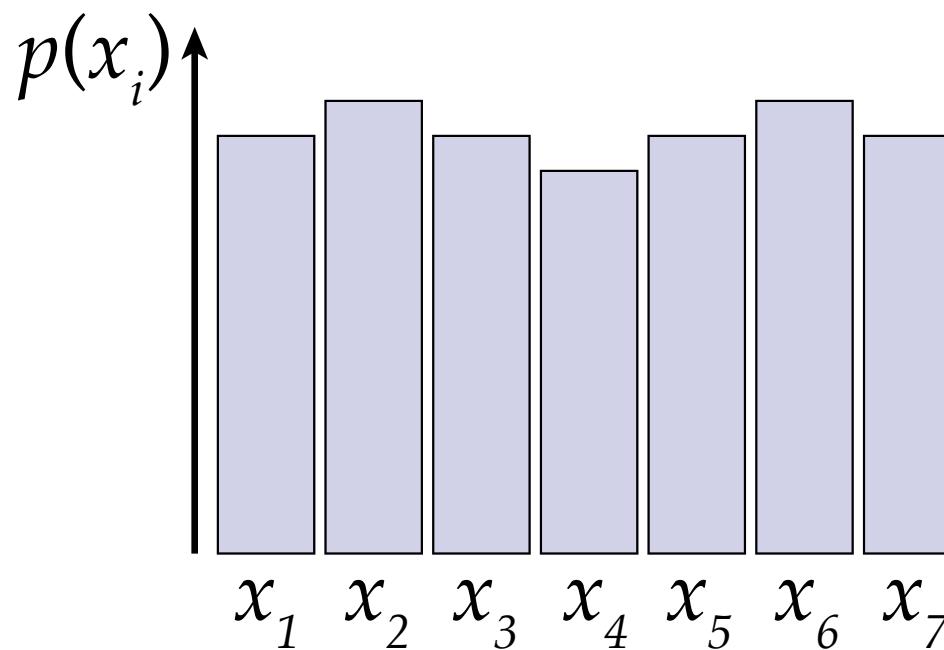
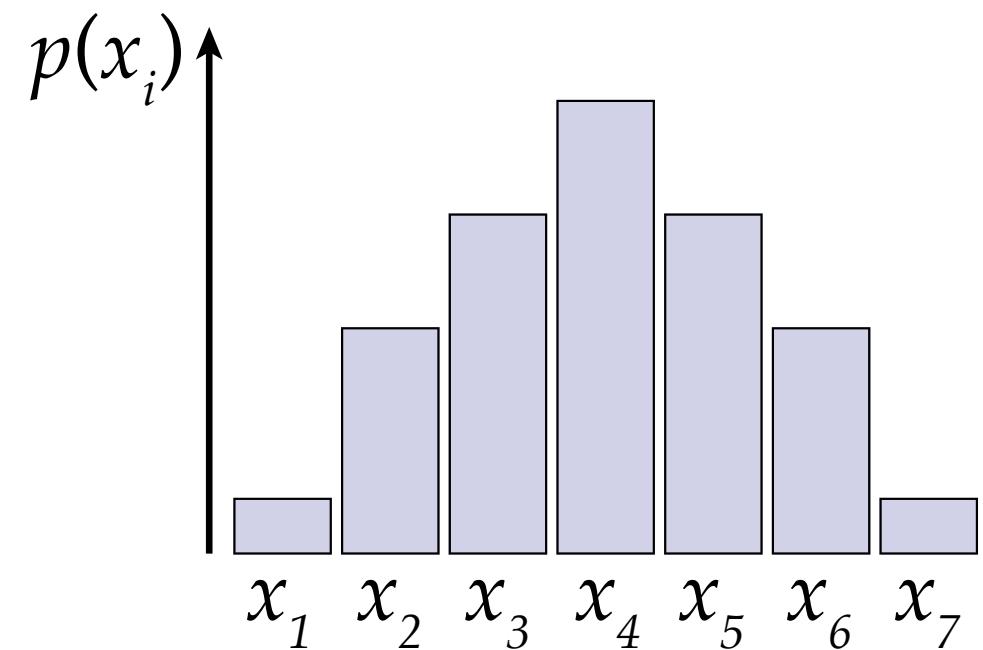
# Variance

**Intuition: how far are our samples from the average, on average?**

## Definition

$$V[Y] = E[(Y - E[Y])^2]$$

**Q: Which of these has higher variance?**



**Properties of variance:**

$$V[Y] = E[Y^2] - E[Y]^2$$

$$V \left[ \sum_{i=1}^N Y_i \right] = \sum_{i=1}^N V[Y_i]$$

$$V[aY] = a^2 V[Y]$$

**(Can you show these are true?)**

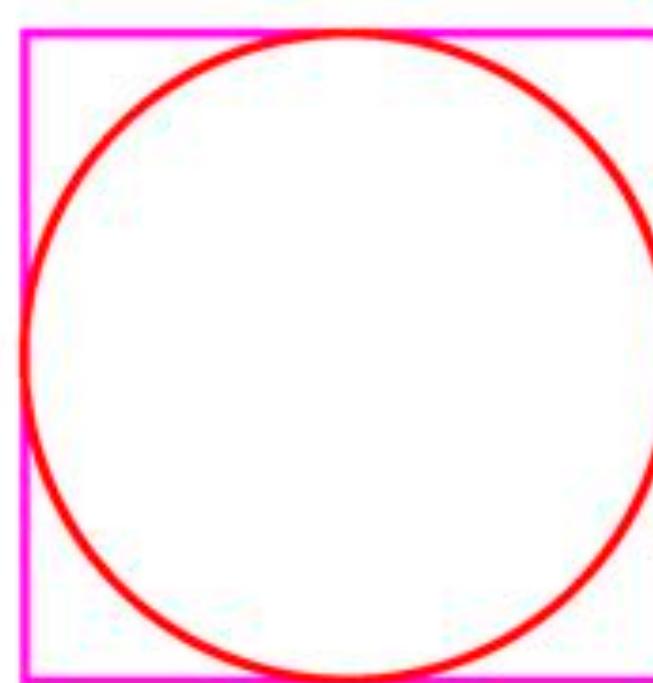
# Law of Large Numbers

- Important fact: for any random variable, the average value of  $N$  trials approaches the expected value as we increase  $N$
- Decrease in variance is always linear in  $N$ :

$$V \left[ \frac{1}{N} \sum_{i=1}^N Y_i \right] = \frac{1}{N^2} \sum_{i=1}^N V[Y_i] = \frac{1}{N^2} N V[Y] = \frac{1}{N} V[Y]$$

Consider a coconut...

nCoconuts	estimate of $\pi$
1	4.000000
10	3.200000
100	3.240000
1000	3.112000
10000	3.163600
100000	3.139520
1000000	3.141764



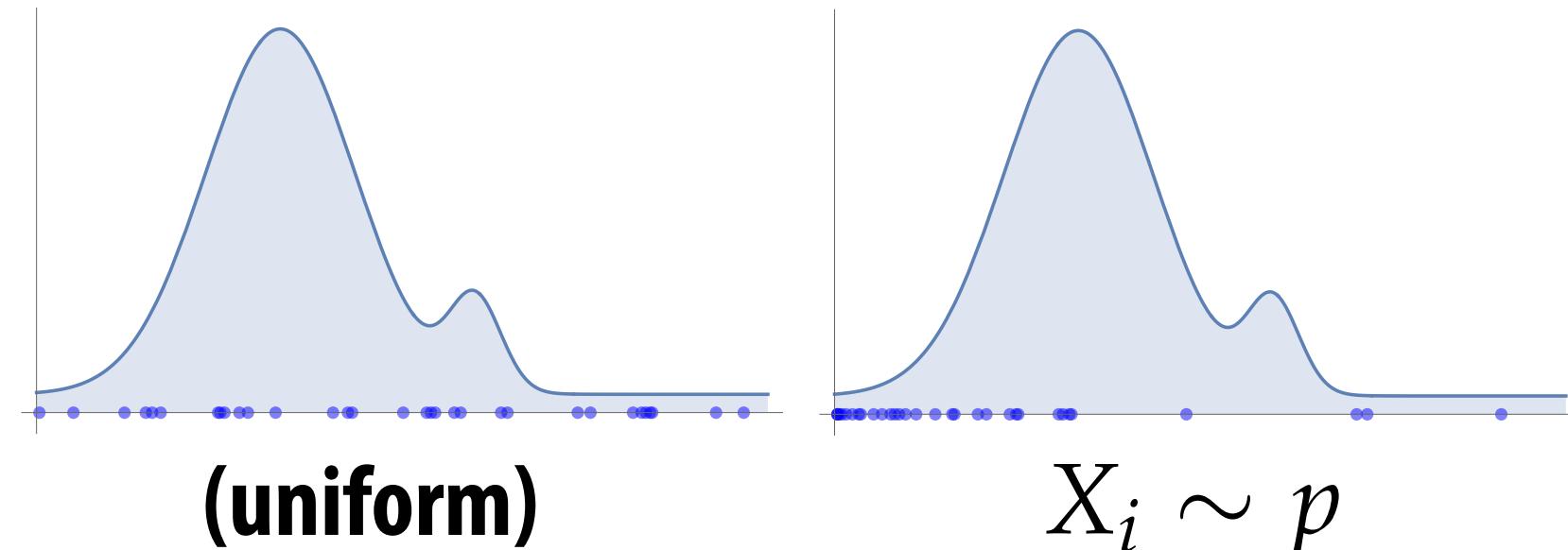
**Q: Why is the law of large numbers important for Monte Carlo ray tracing?**

**A: No matter how hard the integrals are (crazy lighting, geometry, materials, etc.), can always\* get the right image by taking more samples.**

**\*As long as we make sure to sample all possible kinds of light paths...**

# Biassing

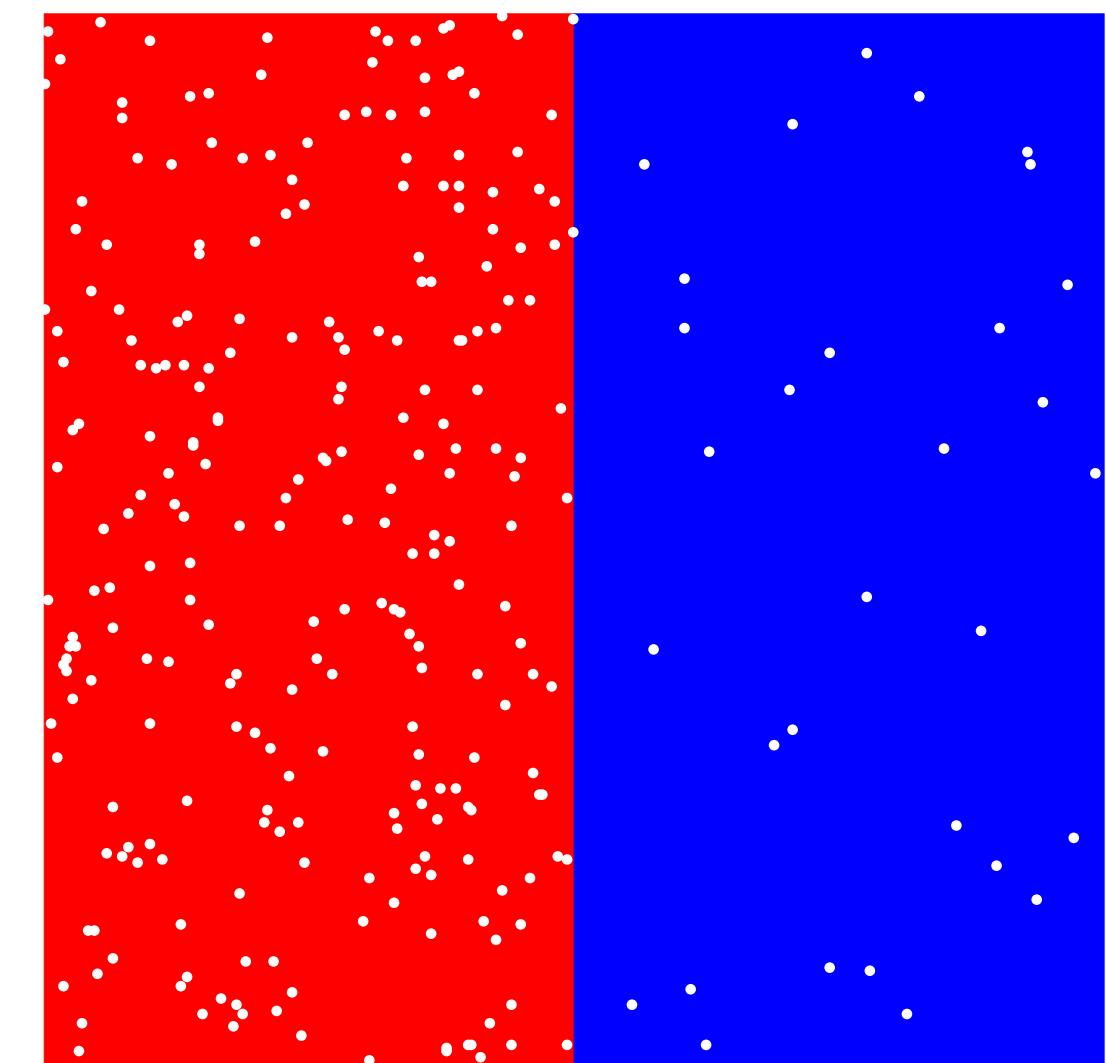
- So far, we've picked samples uniformly from the domain (every point is equally likely)
- Suppose we pick samples from some other distribution (more samples in one place than another)



- Q: Can we still use samples  $f(X_i)$  to get a (correct) estimate of our integral?
- A: Sure! Just weight contribution of each sample by how likely we were to pick it
- Q: Are we correct to divide by  $p$ ? Or... should we multiply instead?

- A: Think about a simple example where we sample RED region 8x as often as BLUE region
  - average color over square should be purple
  - if we multiply, average will be TOO RED
  - if we divide, average will be JUST RIGHT

$$\int_{\Omega} f(x) dx \approx \frac{1}{N} \sum_{i=1}^N \frac{f(X_i)}{p(X_i)}$$

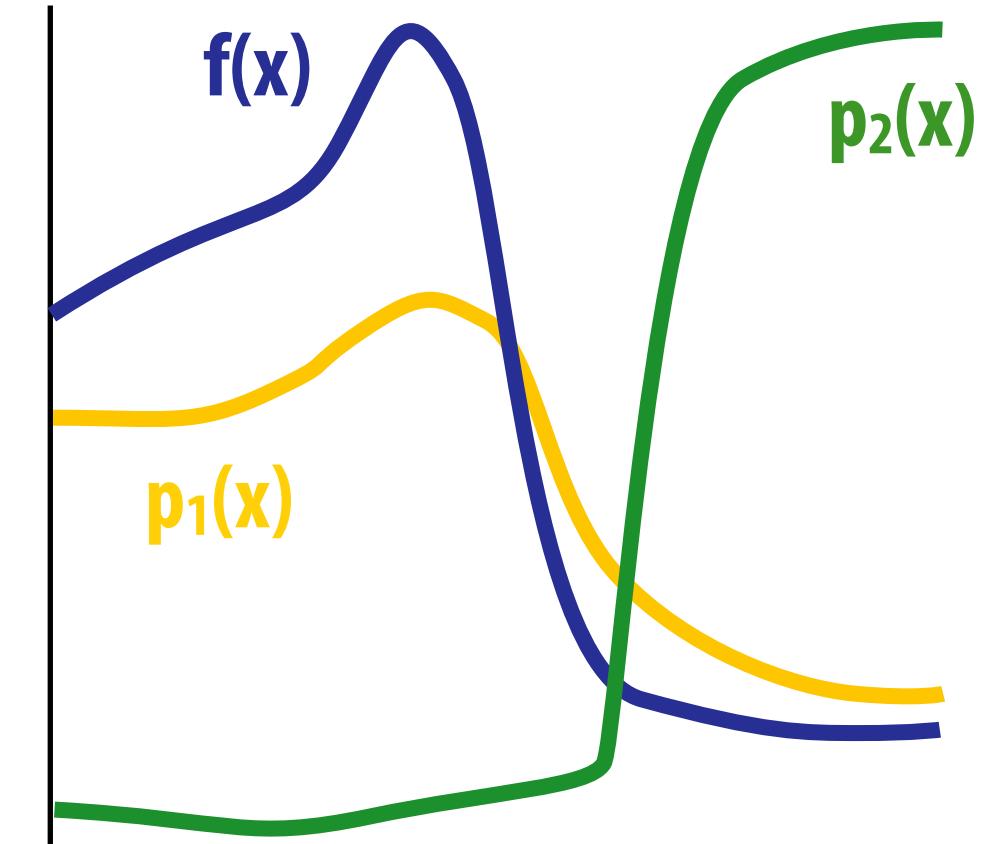


# Importance sampling

Q: Ok, so then WHERE is the best place to take samples?

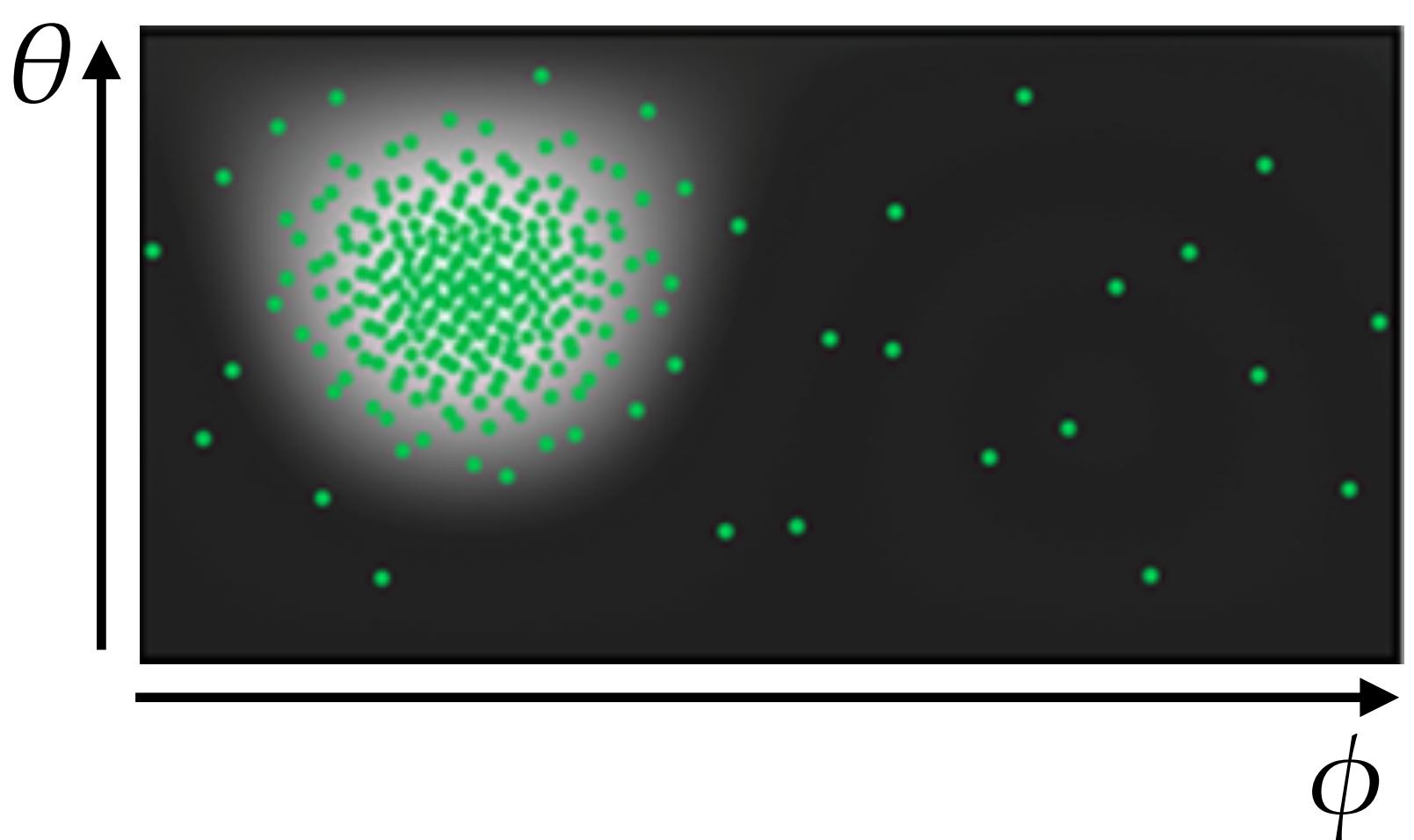
Think:

- What is the behavior of  $f(x)/p_1(x)$ ?  $f(x)/p_2(x)$ ?
- How does this impact the variance of the estimator?



Idea: put more where integrand is large (“most useful samples”). E.g.:

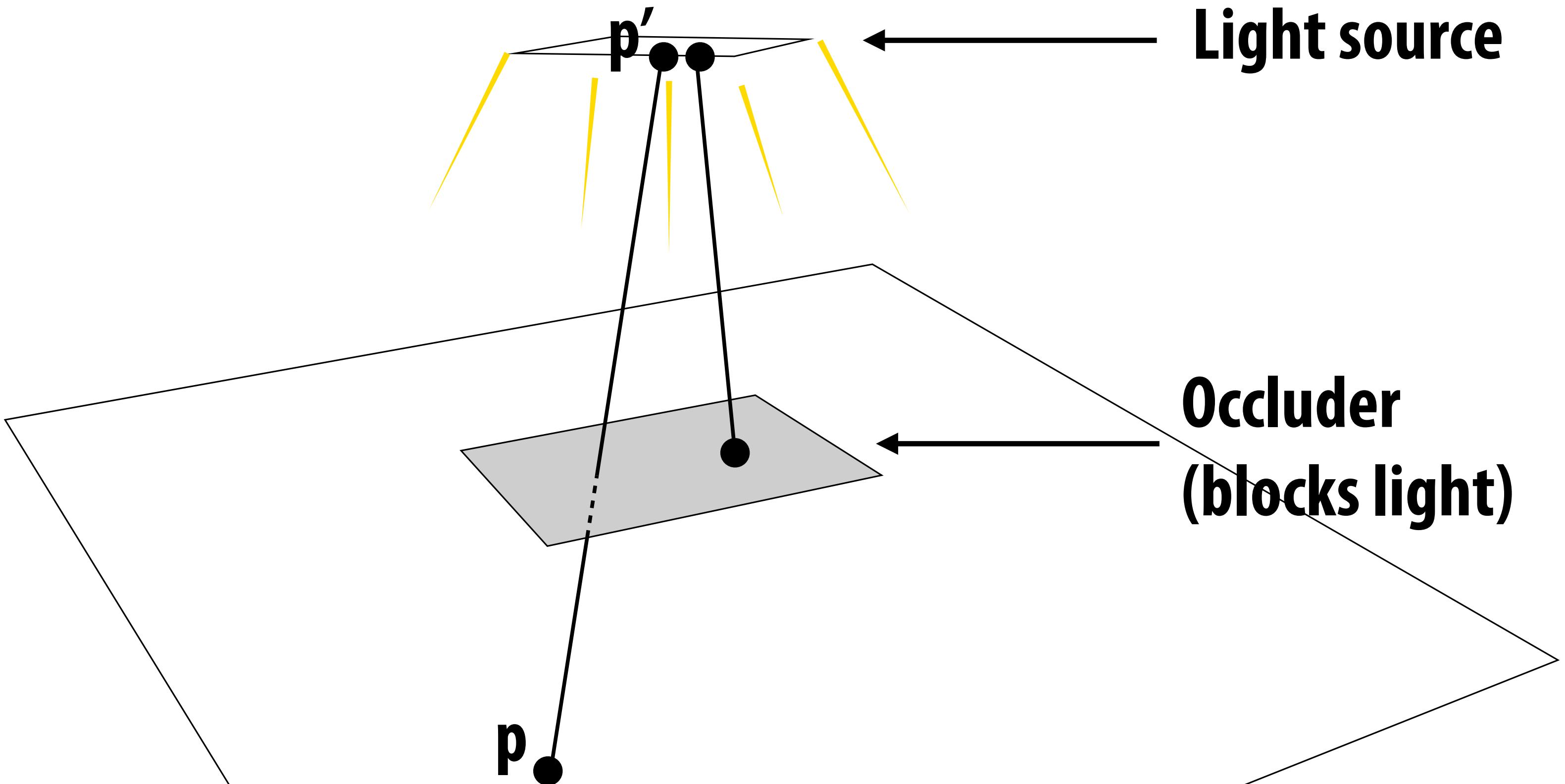
(BRDF)



(image-based lighting)



# Example: Direct Lighting



Visibility function:

$$V(p, p') = \begin{cases} 1, & p \text{ ``sees'' } p' \\ 0, & \text{otherwise} \end{cases}$$

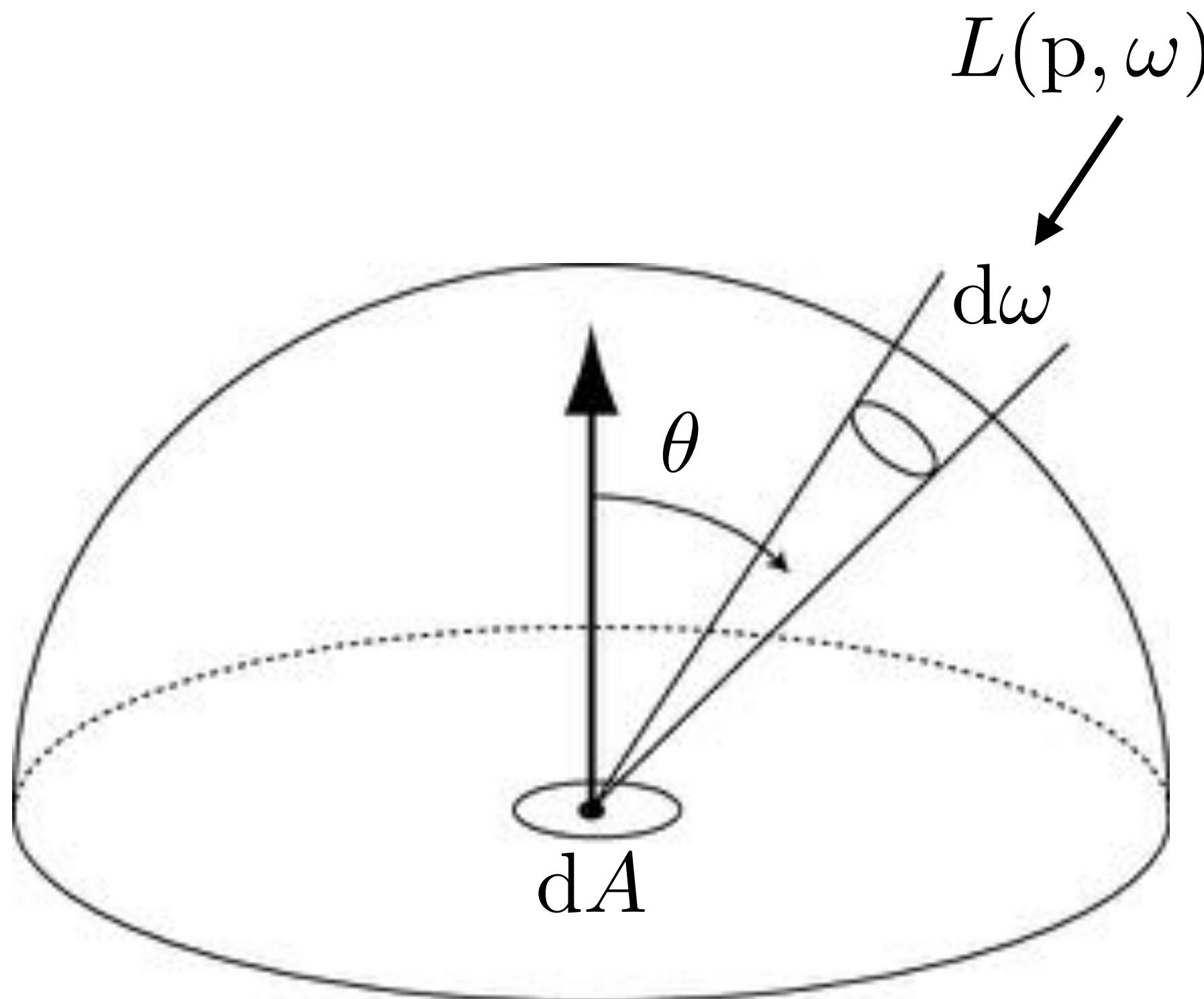
**How bright is each point on the ground?**

# Direct lighting—uniform sampling

Uniformly-sample hemisphere of directions with respect to solid angle

$$p(\omega) = \frac{1}{2\pi}$$

$$E(p) = \int L(p, \omega) \cos \theta d\omega$$



**Estimator:**

$$X_i \sim p(\omega)$$

$$Y_i = f(X_i)$$

$$Y_i = L(p, \omega_i) \cos \theta_i$$

$$F_N = \frac{2\pi}{N} \sum_{i=1}^N Y_i$$

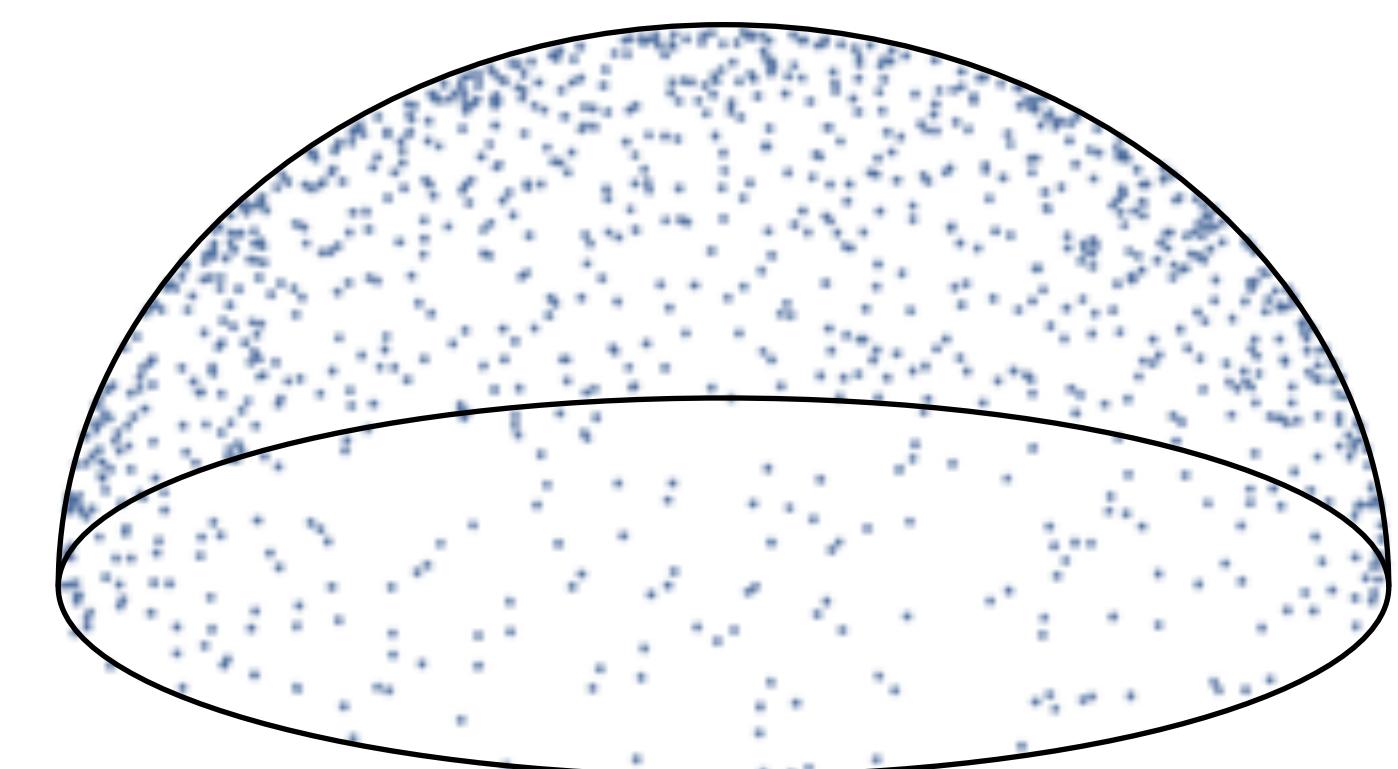
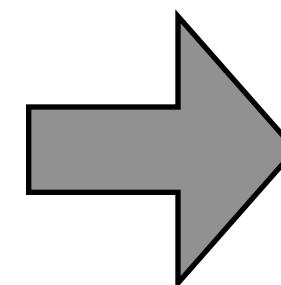
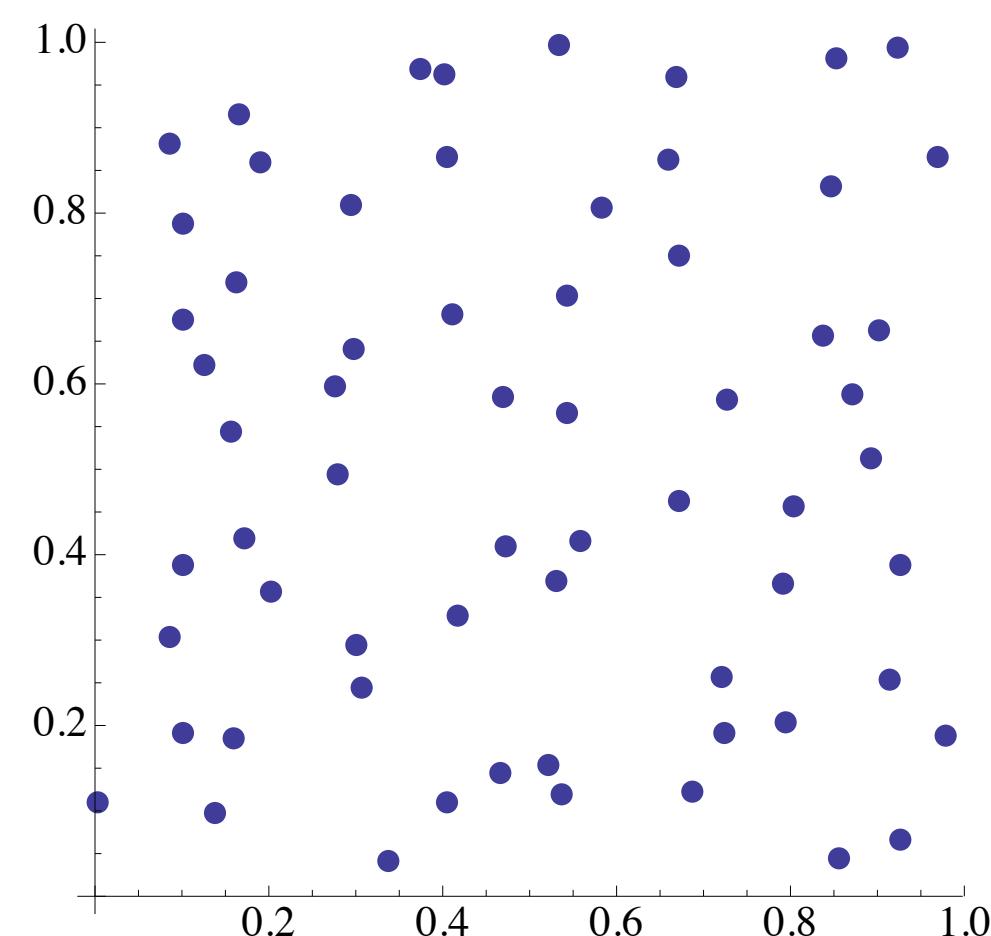
# Aside: Picking points on unit hemisphere

How do we uniformly sample directions from the hemisphere?

One way: use rejection sampling. (How?)

Another way: “warp” two values in  $[0,1]$  via the inversion method:

$$(\xi_1, \xi_2) = (\sqrt{1 - \xi_1^2} \cos(2\pi\xi_2), \sqrt{1 - \xi_1^2} \sin(2\pi\xi_2), \xi_1)$$



**Exercise: derive from the inversion method**

# Direct lighting—uniform sampling (algorithm)

Uniformly-sample hemisphere of directions with respect to solid angle

$$p(\omega) = \frac{1}{2\pi}$$

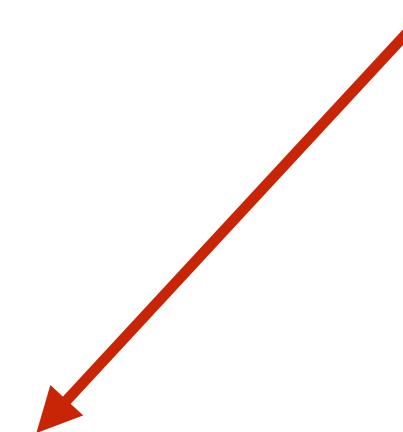
$$E(p) = \int L(p, \omega) \cos \theta d\omega$$

Given surface point p

A ray tracer evaluates radiance along a ray  
(see Raytracer::trace\_ray() in raytracer.cpp)

For each of N samples:

Generate random direction:  $\omega_i$

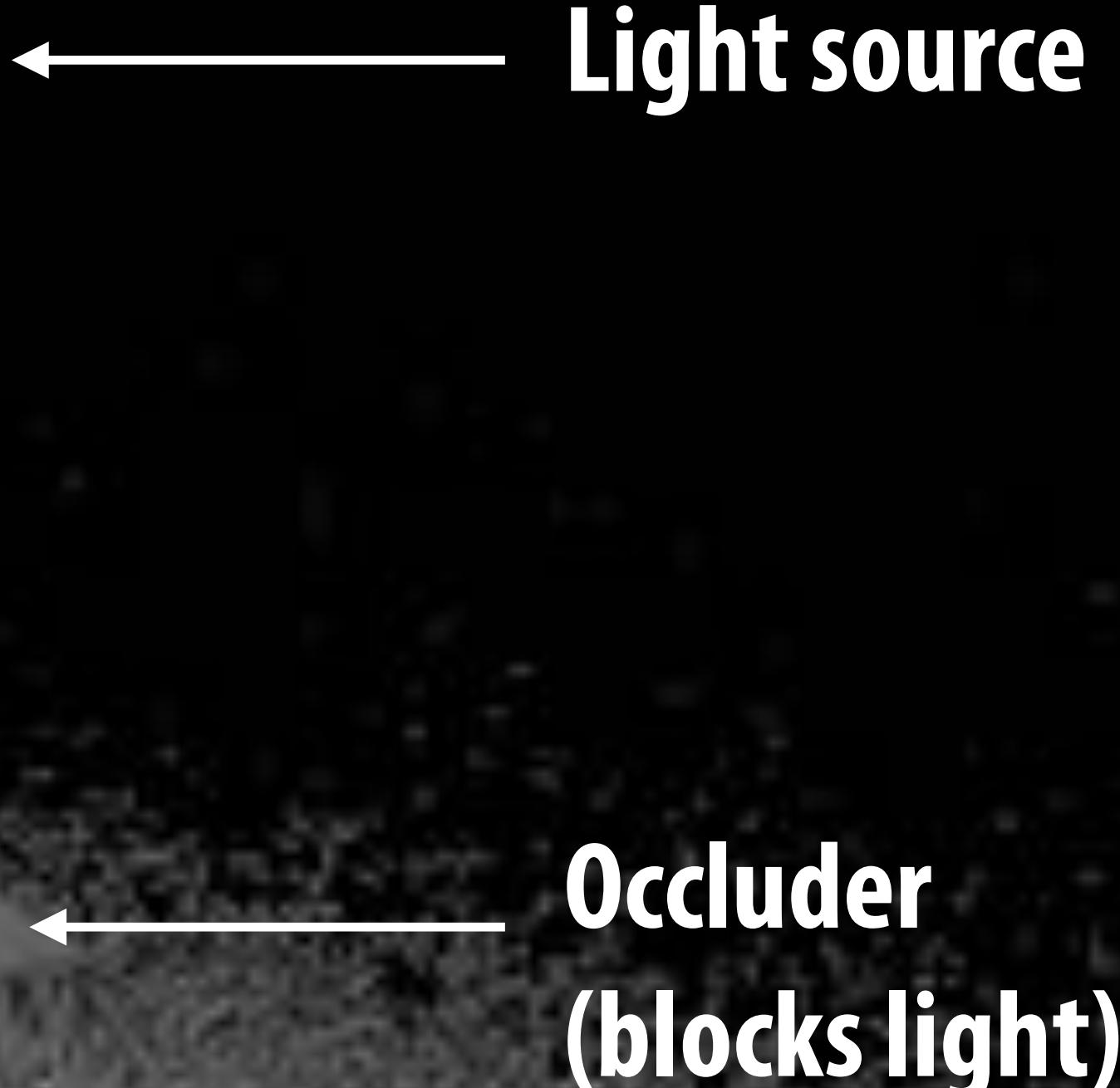


Compute incoming radiance arriving  $L_i$  at p from direction:  $\omega_i$

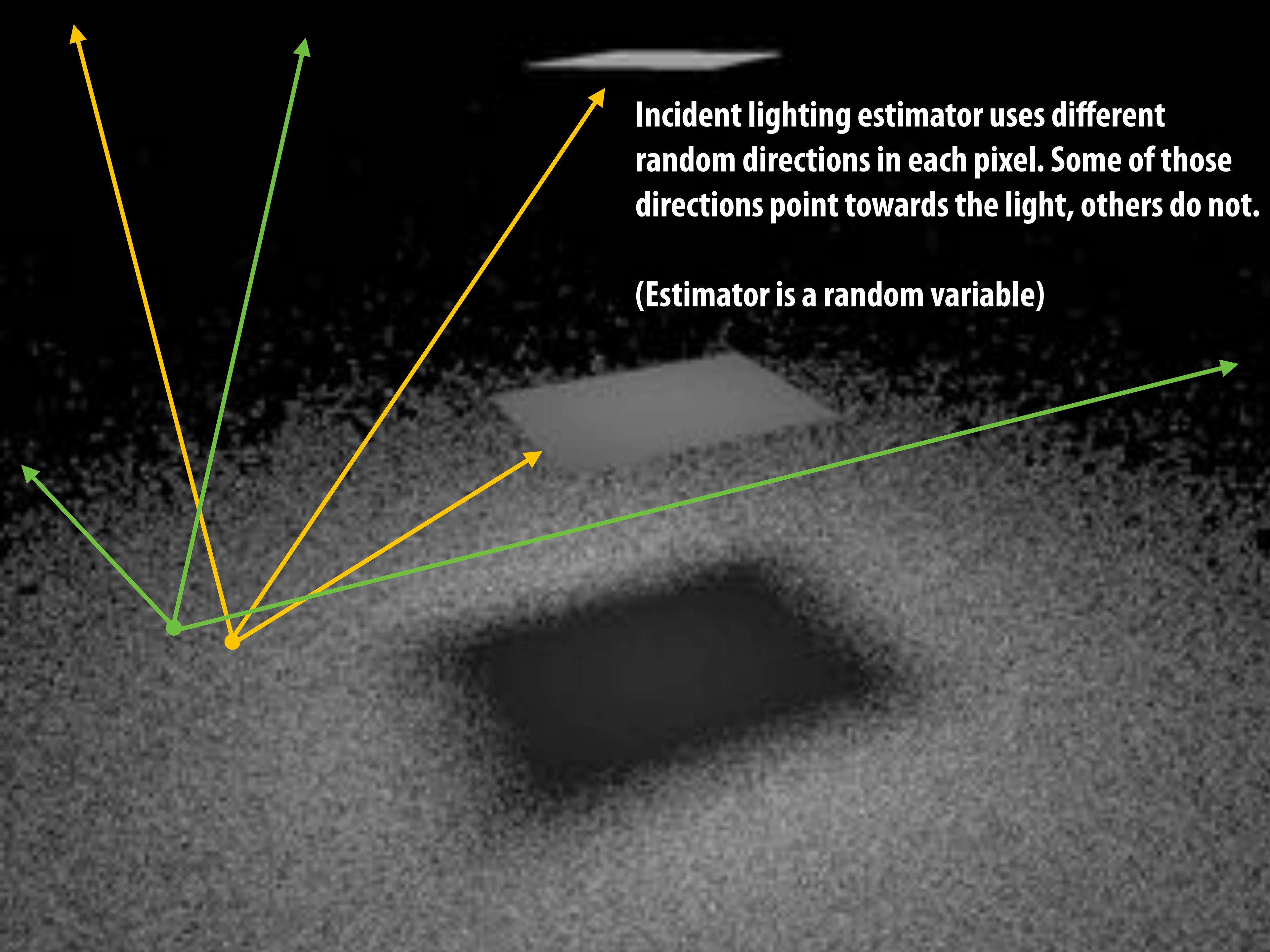
Compute incident irradiance due to ray:  $dE_i = L_i \cos \theta_i$

Accumulate  $\frac{2\pi}{N} dE_i$  into estimator

**Hemispherical solid angle  
sampling, 100 sample rays  
(random directions drawn  
uniformly from hemisphere)**



**Why is the image in the previous slide “noisy”?**



**Incident lighting estimator uses different random directions in each pixel. Some of those directions point towards the light, others do not.**

**(Estimator is a random variable)**

# **How can we reduce noise?**

**One idea: just take more samples!**

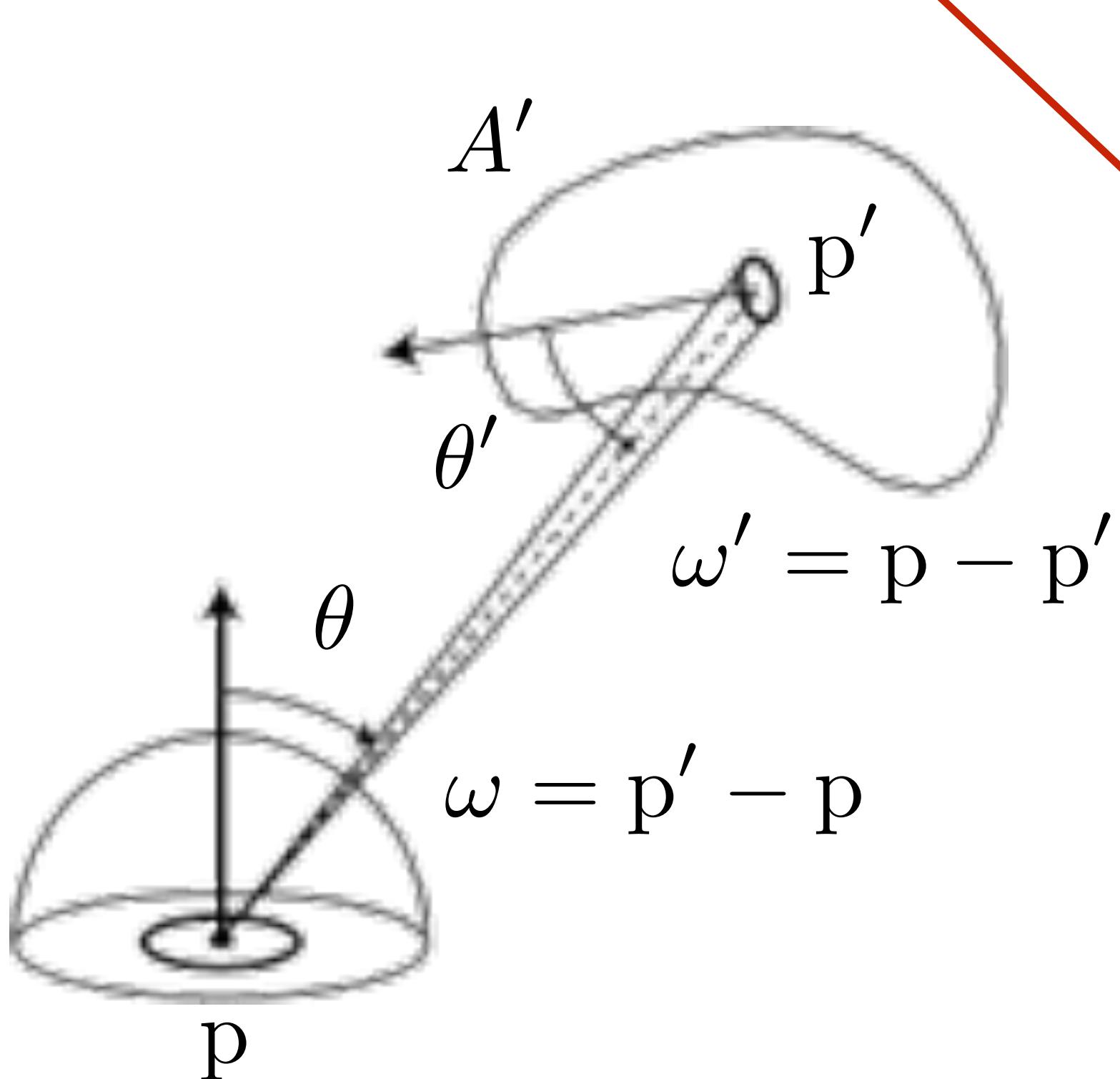
**Another idea:**

- Don't need to integrate over entire hemisphere of directions (incoming radiance is 0 from most directions).
- Just integrate over the area of the light (directions where incoming radiance is non-zero) and weight appropriately

# Direct lighting: area integral

$$E(p) = \int L(p, \omega) \cos \theta d\omega \quad \text{← Previously: just integrate over all directions}$$

$$E(p) = \int_{A'} L_o(p', \omega') V(p, p') \frac{\cos \theta \cos \theta'}{|p - p'|^2} dA' \quad \text{← Change of variables to integrate over area of light}$$



$$d\omega = \frac{dA}{|p' - p|^2} = \frac{dA' \cos \theta'}{|p' - p|^2}$$

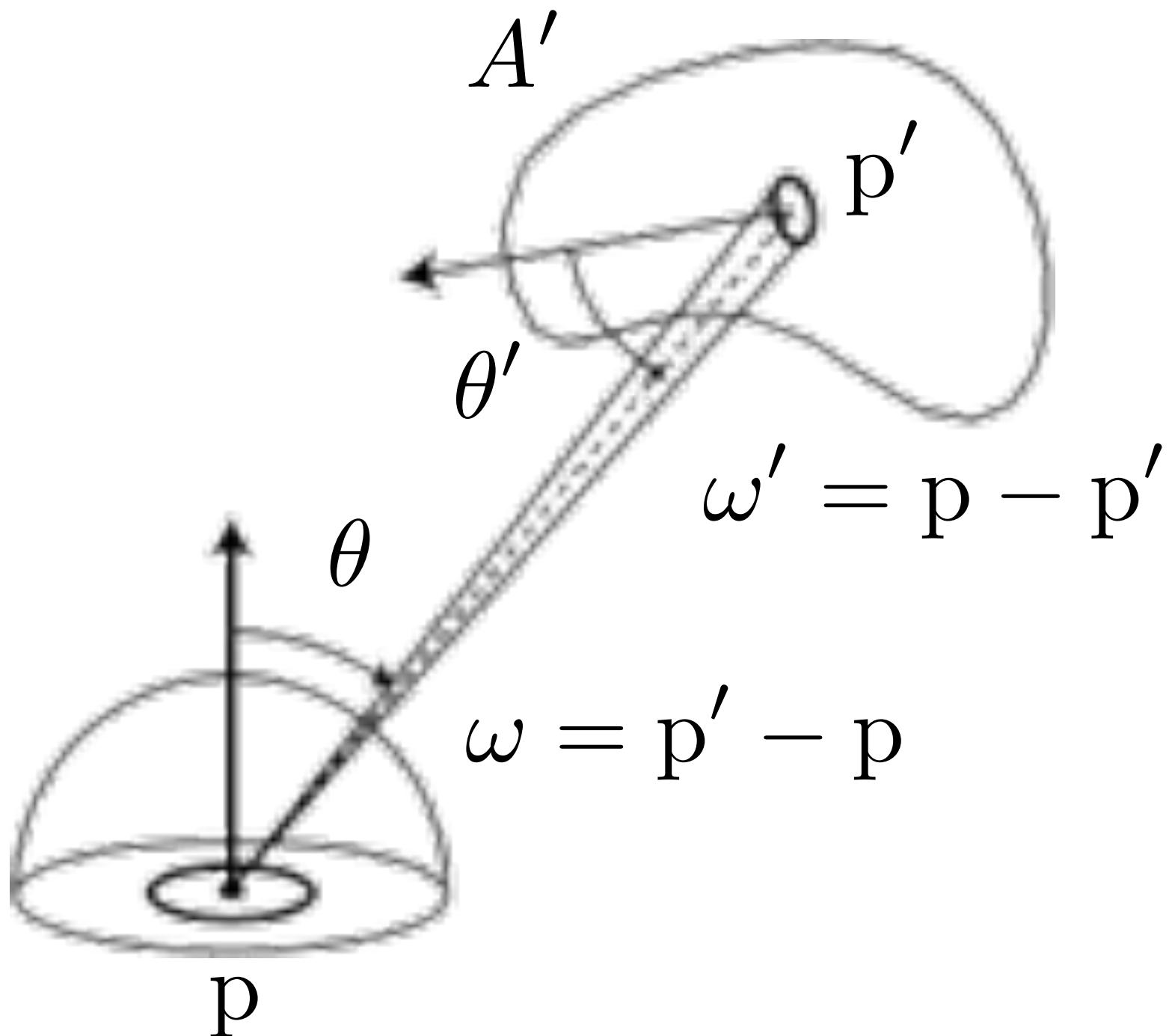
Binary visibility function:  
1 if  $p'$  is visible from  $p$ , 0 otherwise  
(accounts for light occlusion)

Outgoing radiance from light point  $p$ , in direction  $w'$  towards  $p$

# Direct lighting: area integral

$$E(p) = \int_{A'} L_o(p', \omega') V(p, p') \frac{\cos \theta \cos \theta'}{|p - p'|^2} dA'$$

**Sample shape uniformly by area  $A'$**

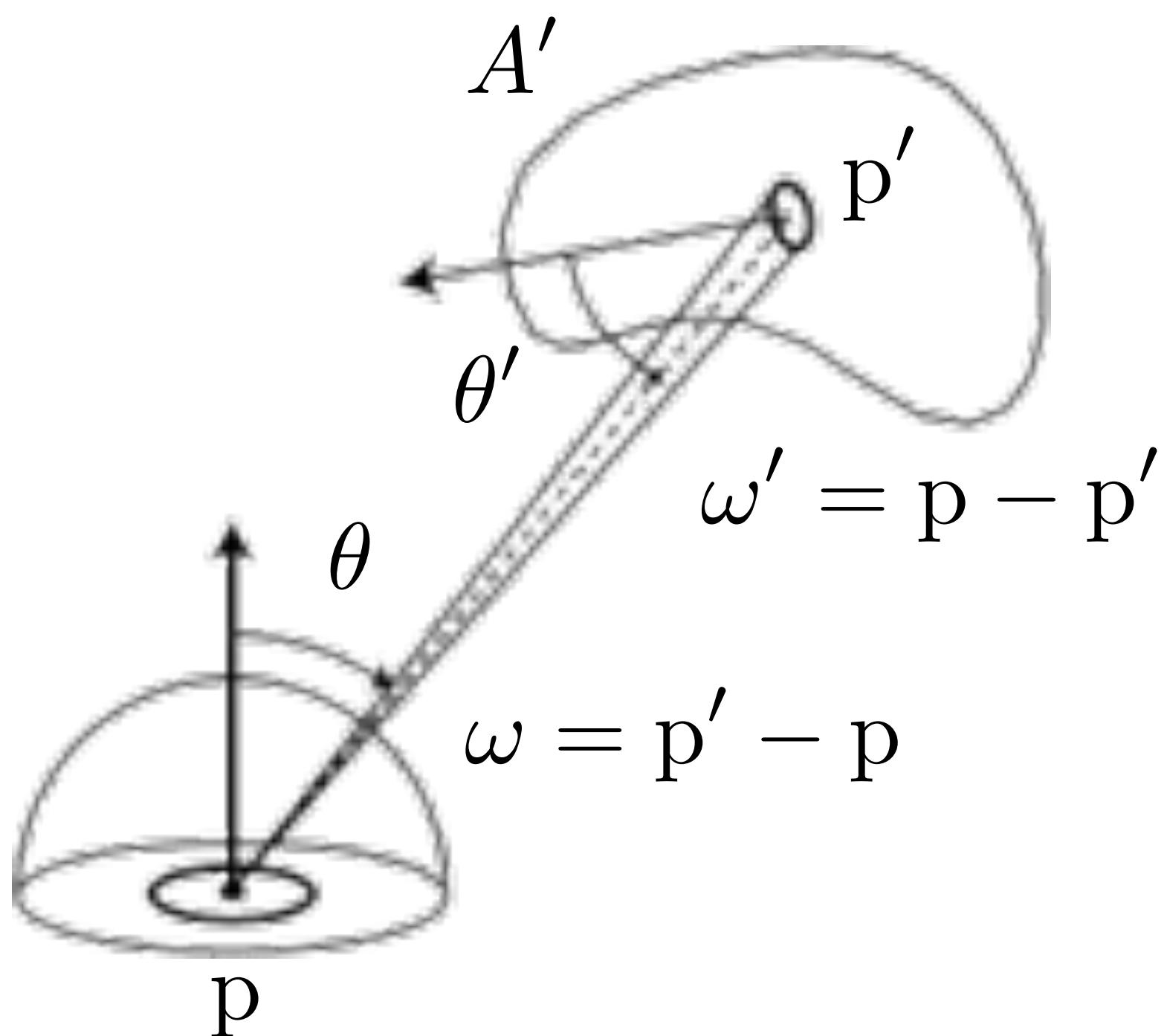


$$\int_{A'} p(p') dA' = 1$$

$$p(p') = \frac{1}{A'}$$

# Direct lighting: area integral

$$E(p) = \int_{A'} L_o(p', \omega') V(p, p') \frac{\cos \theta \cos \theta'}{|p - p'|^2} dA'$$



**Probability:**

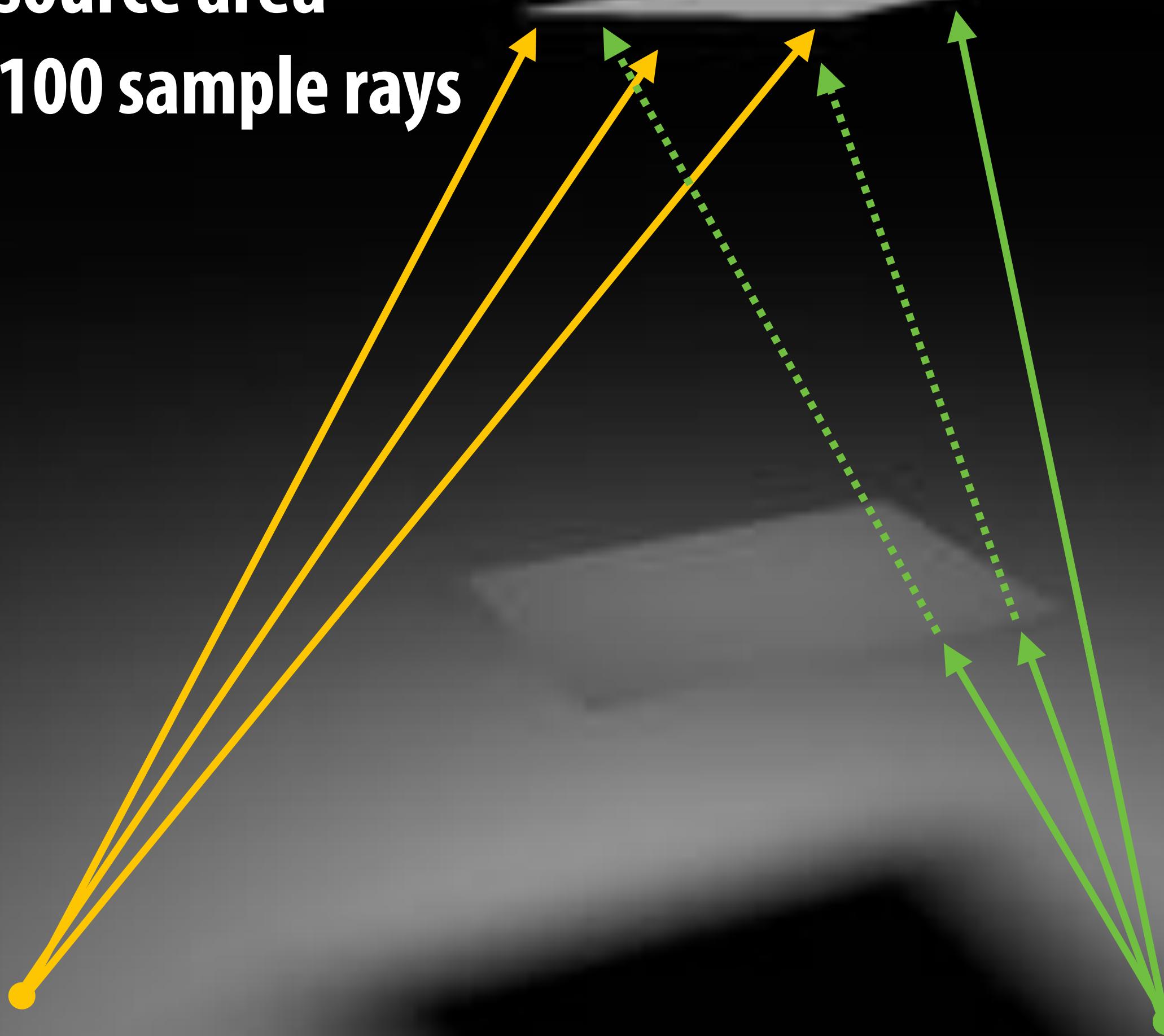
$$p(p') = \frac{1}{A'}$$

**Estimator**

$$Y_i = L_o(p'_i, \omega'_i) V(p, p'_i) \frac{\cos \theta_i \cos \theta'_i}{|p - p'_i|^2}$$

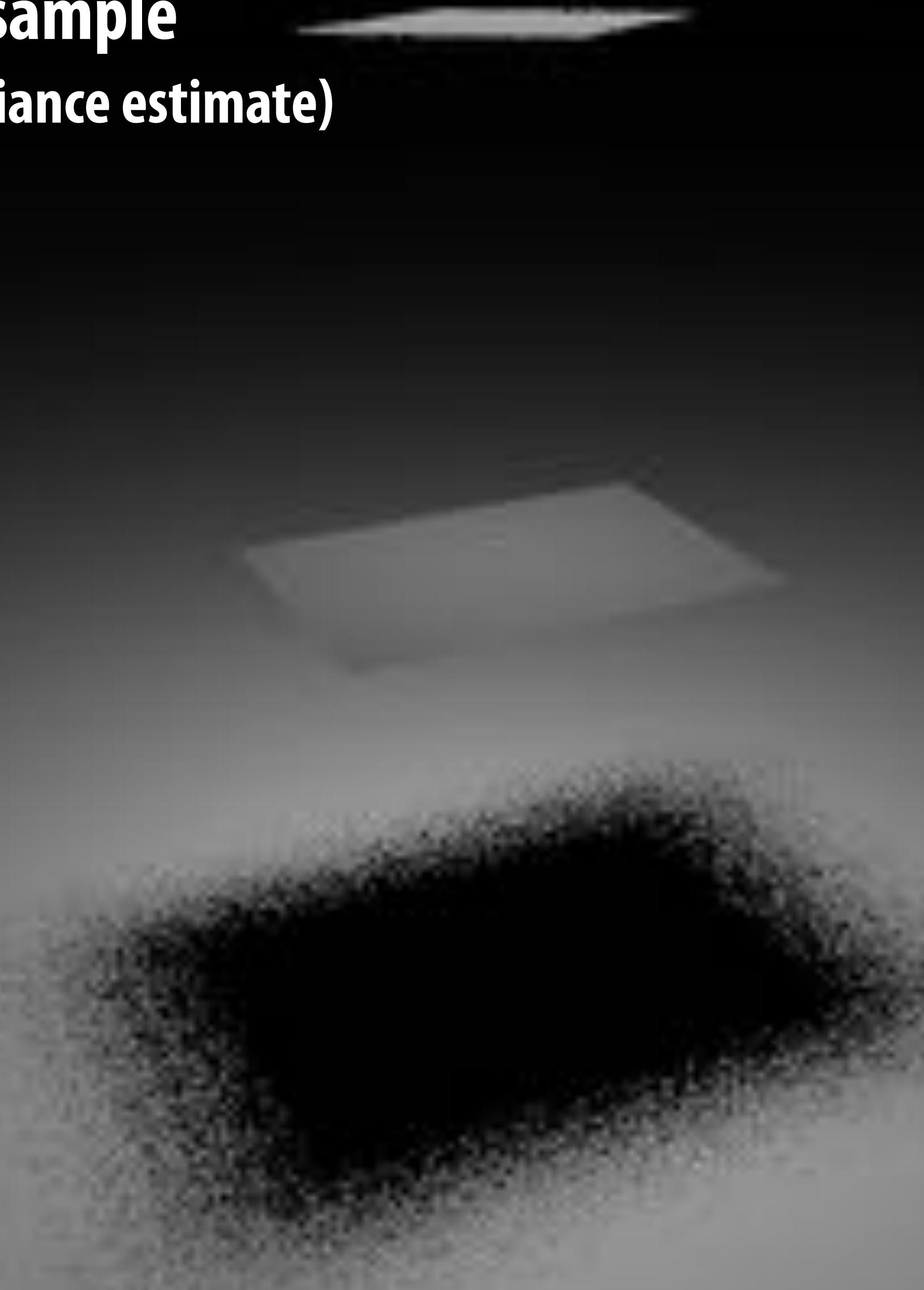
$$F_N = \frac{A'}{N} \sum_{i=1}^N Y_i$$

**Light source area  
sampling, 100 sample rays**



If no occlusion is present, all directions chosen in computing estimate “hit” the light source.  
(Choice of direction only matters if portion of light is occluded from surface point p.)

**1 area light sample  
(high variance in irradiance estimate)**



**16 area light samples**

**(lower variance in irradiance estimate)**



# Comparing different techniques

- Variance in an estimator manifests as noise in rendered images
- Estimator efficiency measure:

$$\text{Efficiency} \propto \frac{1}{\text{Variance} \times \text{Cost}}$$

- If one integration technique has twice the variance of another, then it takes twice as many samples to achieve the same variance
- If one technique has twice the cost of another technique with the same variance, then it takes twice as much time to achieve the same variance

# Example—Cosine-Weighted Sampling

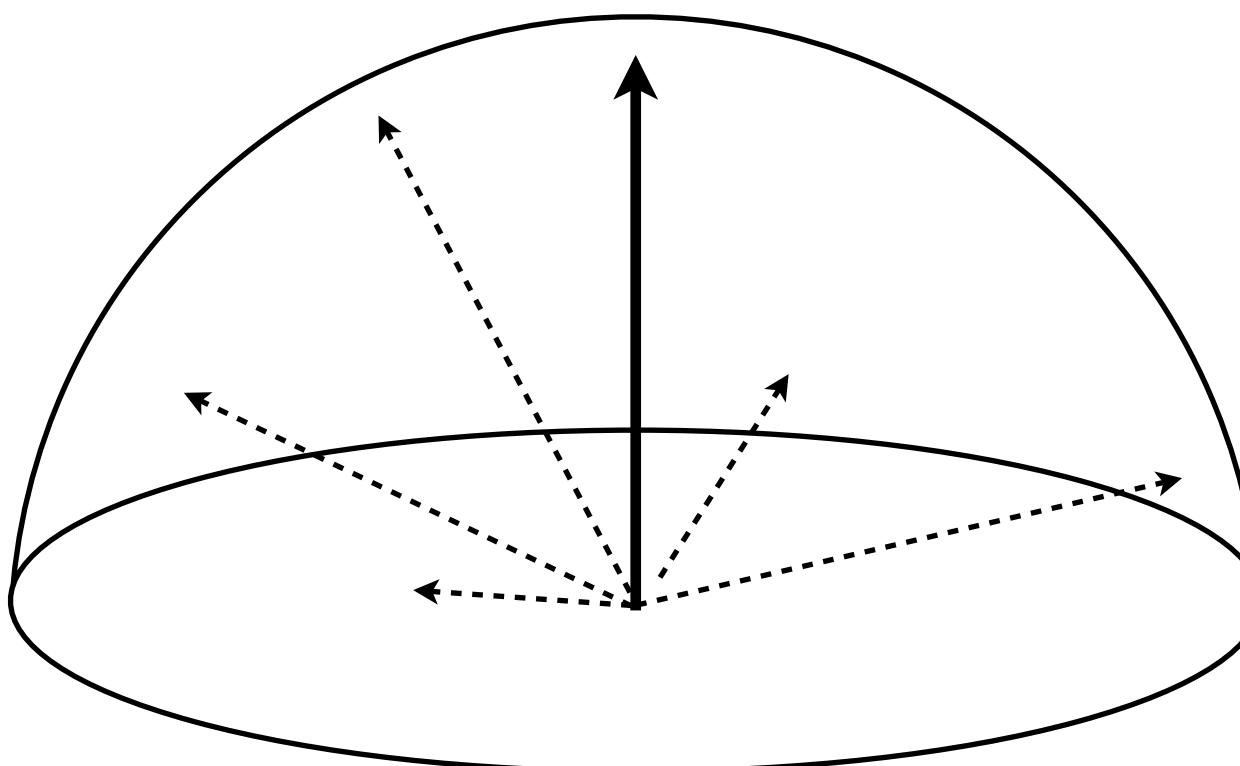
Consider uniform hemisphere sampling in irradiance estimate:

$$f(\omega) = L_i(\omega) \cos \theta$$

$$p(\omega) = \frac{1}{2\pi}$$

$$(\xi_1, \xi_2) = (\sqrt{1 - \xi_1^2} \cos(2\pi\xi_2), \sqrt{1 - \xi_1^2} \sin(2\pi\xi_2), \xi_1)$$

$$\int_{\Omega} f(\omega) d\omega \approx \frac{1}{N} \sum_i^N \frac{f(\omega)}{p(\omega)} = \frac{1}{N} \sum_i^N \frac{L_i(\omega) \cos \theta}{1/2\pi} = \frac{2\pi}{N} \sum_i^N L_i(\omega) \cos \theta$$



# Example—Cosine-Weighted Sampling

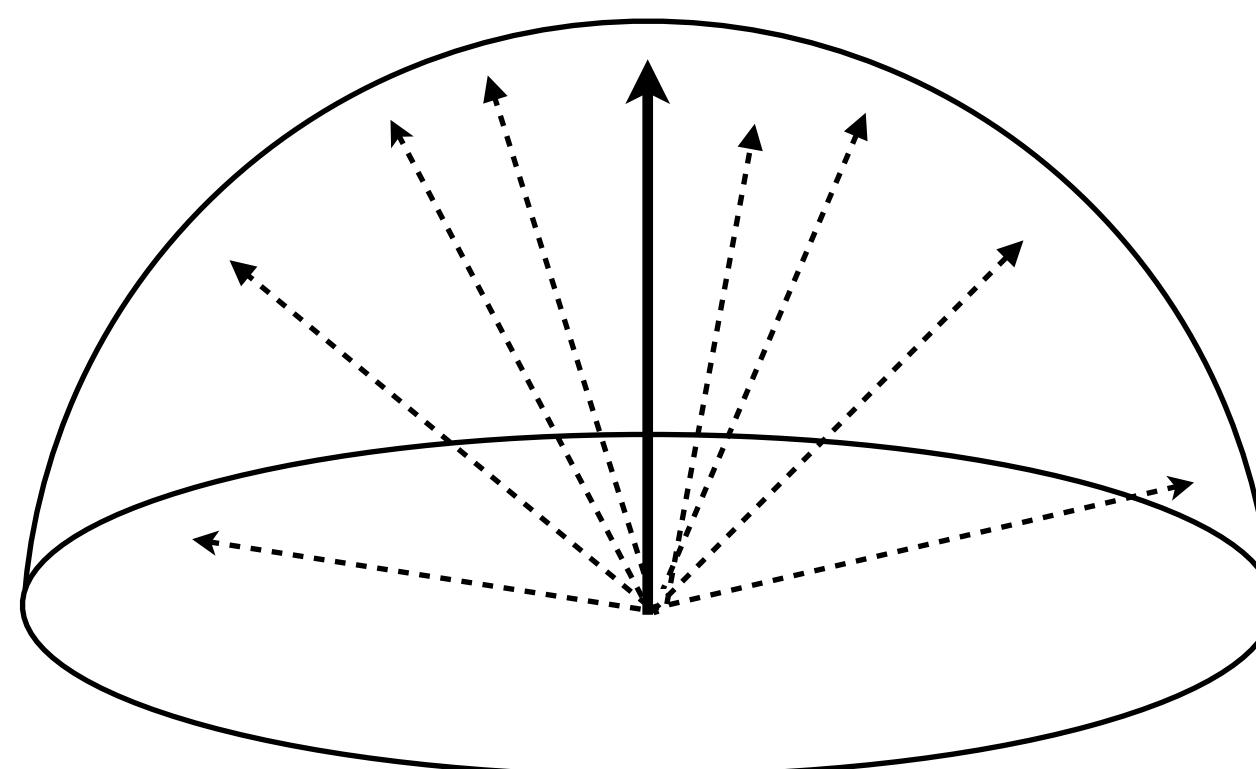
Cosine-weighted hemisphere sampling in irradiance estimate:

$$f(\omega) = L_i(\omega) \cos \theta$$

$$p(\omega) = \frac{\cos \theta}{\pi}$$

$$\int_{\Omega} f(\omega) d\omega \approx \frac{1}{N} \sum_i^N \frac{f(\omega)}{p(\omega)} = \frac{1}{N} \sum_i^N \frac{L_i(\omega) \cos \theta}{\cos \theta / \pi} = \frac{\pi}{N} \sum_i^N L_i(\omega)$$

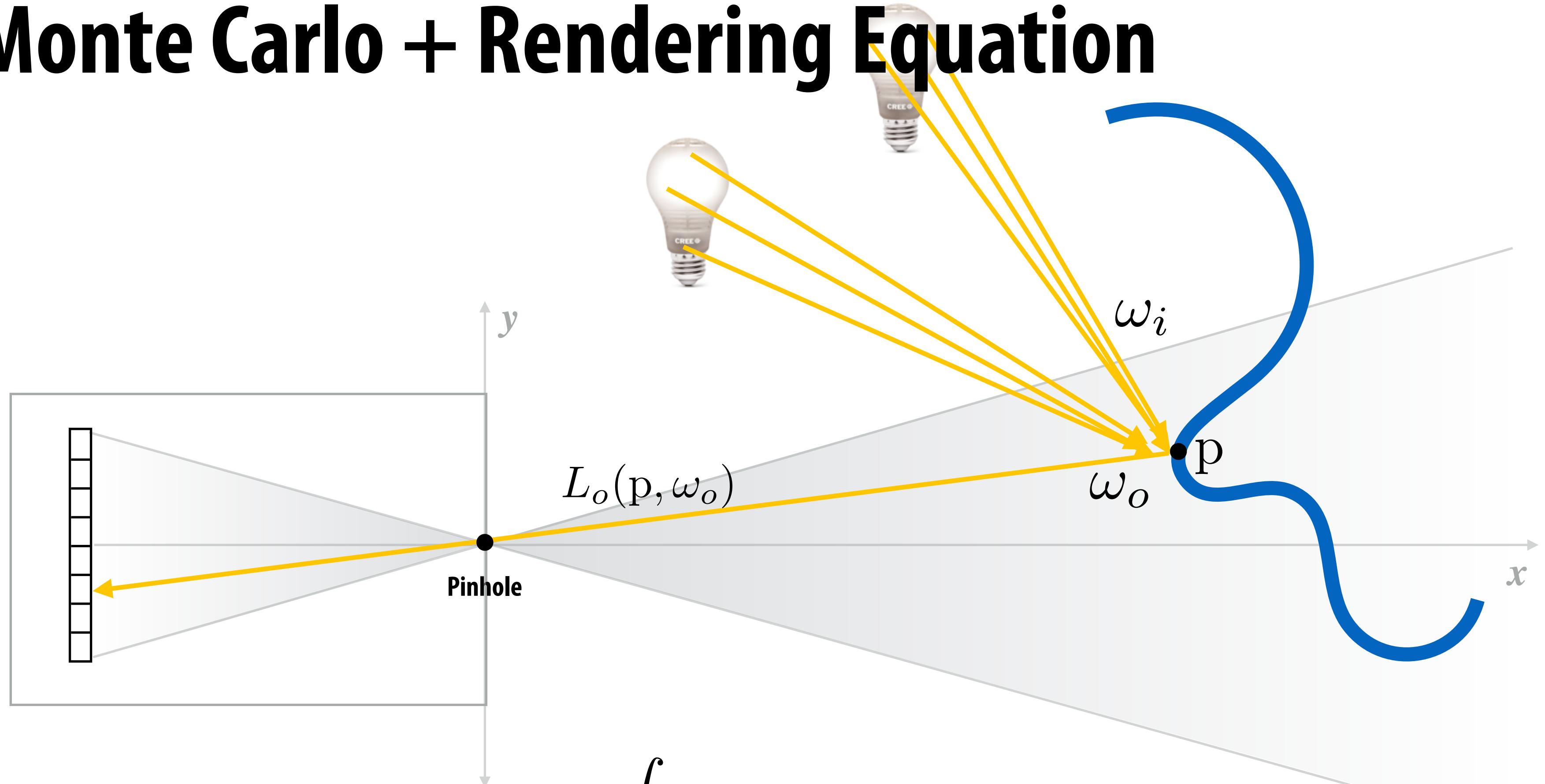
Idea: bias samples toward directions where  $\cos \theta$  is large  
(if  $L$  is constant, then these are the directions that contribute most)



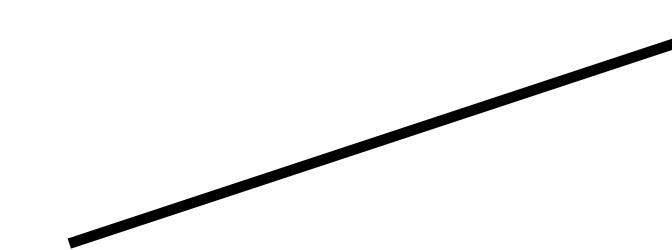
**So far we've considered light coming directly from light sources, scattered once.**

**How do we use Monte Carlo integration to get the final color values for each pixel?**

# Monte Carlo + Rendering Equation



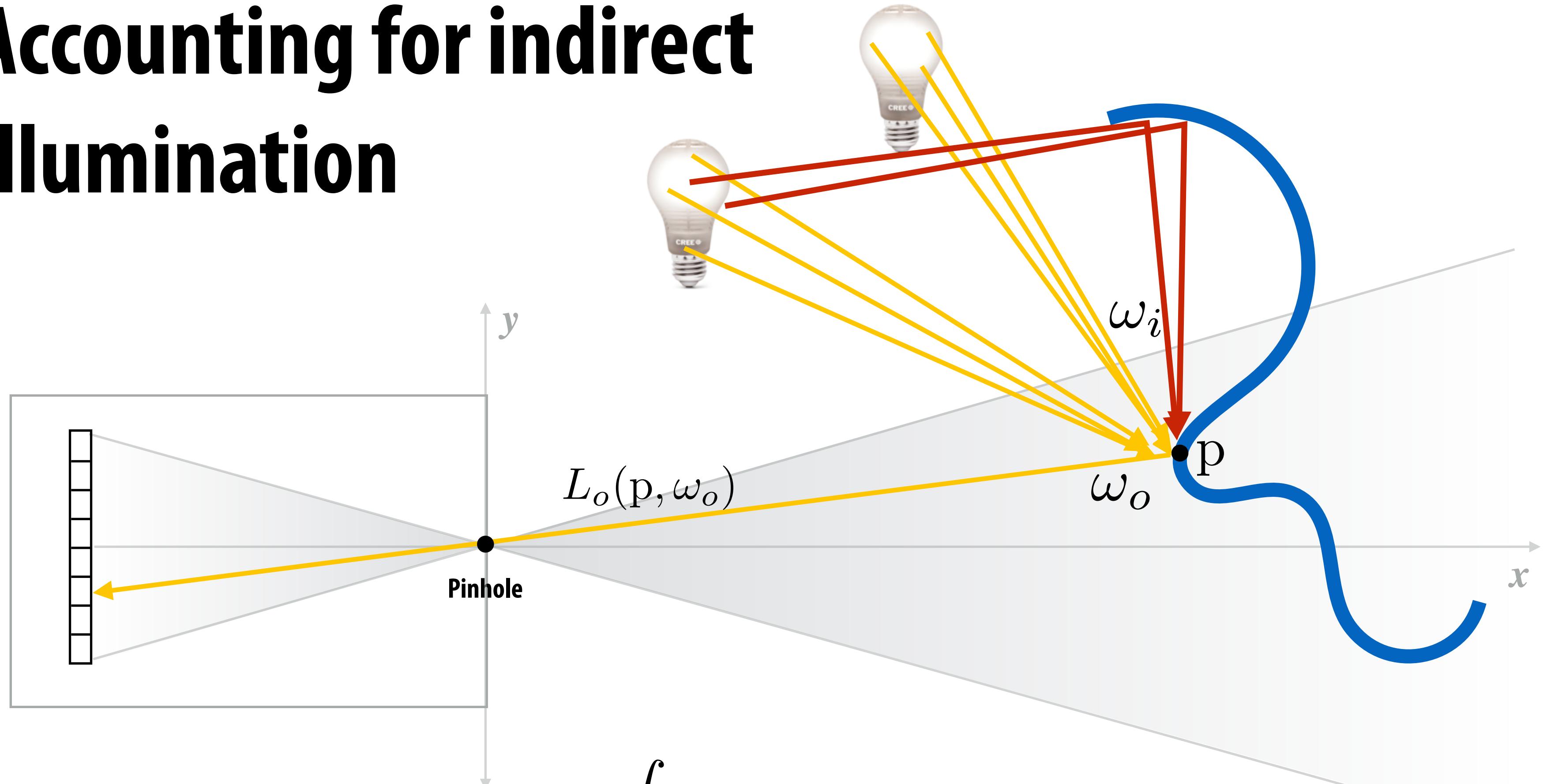
$$L_o(p, \omega_o) = L_e(p, \omega_o) + \int_{H^2} f_r(p, \omega_i \rightarrow \omega_o) L_i(p, \omega_i) \cos \theta_i d\omega_i$$



Need to know incident radiance.

So far, have only computed incoming radiance from scene light sources.

# Accounting for indirect illumination



Incoming light energy from direction  $\omega_i$  may be due to light reflected off another surface in the scene (not an emitter)

# Path tracing: indirect illumination

$$\int_{H^2} f_r(\omega_i \rightarrow \omega_o) L_{o,i}(tr(p, \omega_i), -\omega_i) \cos \theta_i d\omega_i$$

- **Sample incoming direction from some distribution (e.g. proportional to BRDF):**
$$\omega_i \sim p(\omega)$$
- **Recursively call path tracing function to compute incident indirect radiance**

•p

Direct illumination



• p

One-bounce global illumination

• p

Two-bounce global illumination

• p

Four-bounce global illumination

• p

Eight-bounce global illumination

• p

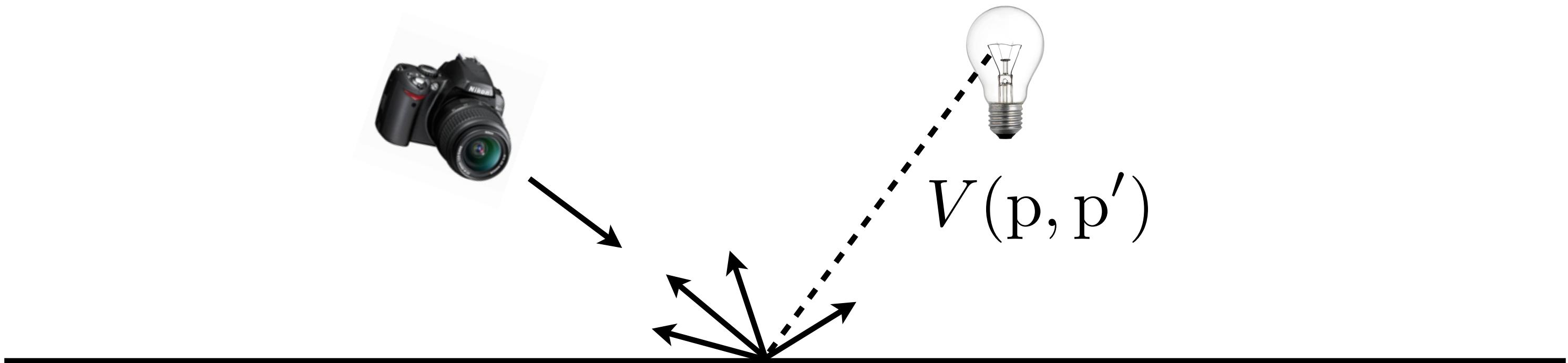
Sixteen-bounce global illumination

**Wait a minute...  
When do we stop?!**

# Russian roulette

- Idea: want to avoid spending time evaluating function for samples that make a **small contribution** to the final result
- Consider a low-contribution sample of the form:

$$L = \frac{f_r(\omega_i \rightarrow \omega_o) L_i(\omega_i) V(p, p') \cos \theta_i}{p(\omega_i)}$$



# Russian roulette

$$L = \frac{f_r(\omega_i \rightarrow \omega_o) L_i(\omega_i) V(p, p') \cos \theta_i}{p(\omega_i)}$$



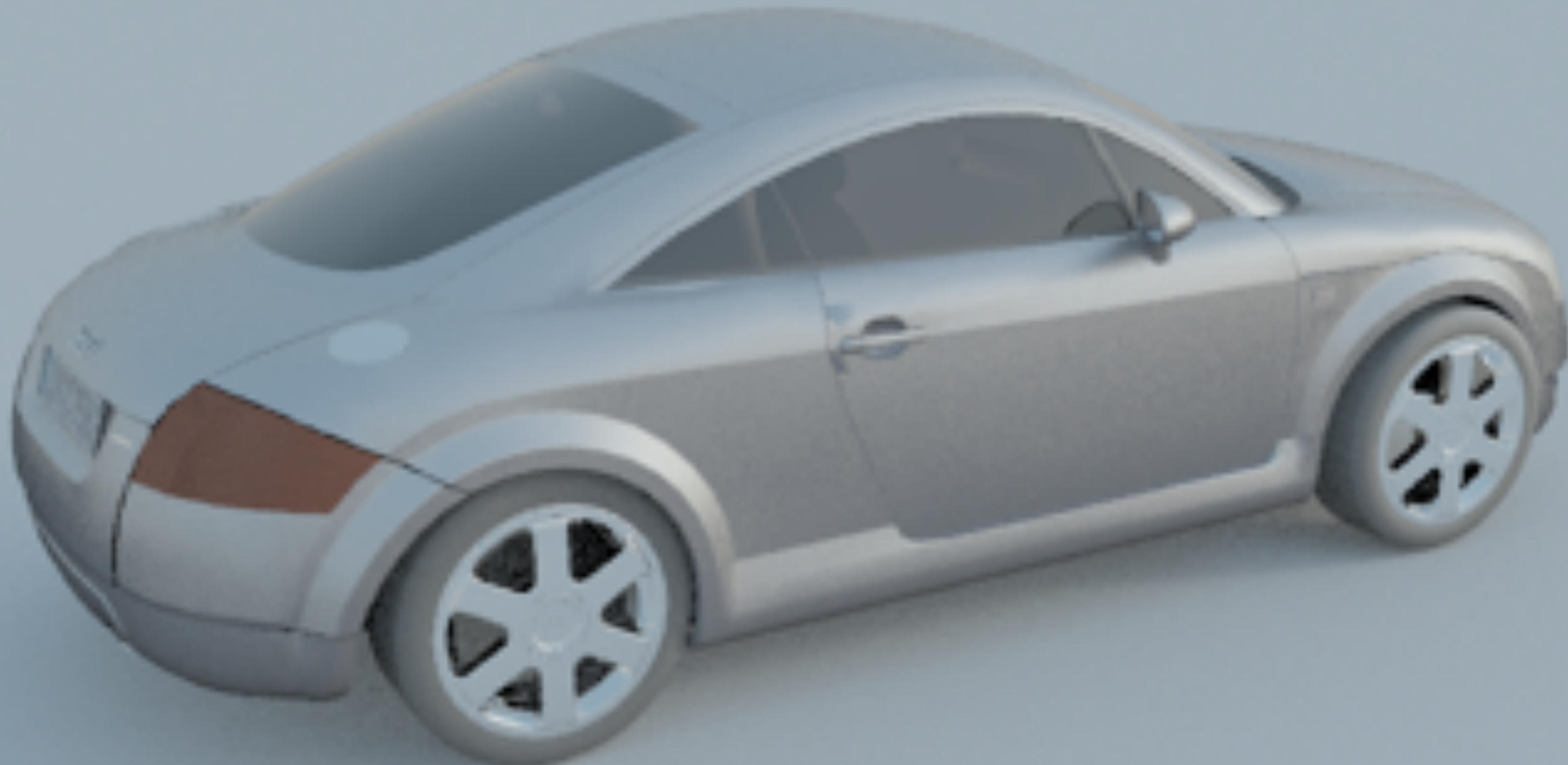
$$L = \left[ \frac{f_r(\omega_i \rightarrow \omega_o) L_i(\omega_i) \cos \theta_i}{p(\omega_i)} \right] V(p, p')$$

- If tentative contribution (in brackets) is small, total contribution to the image will be small regardless of  $V(p, p')$
- Ignoring low-contribution samples introduces systematic error
  - No longer converges to correct value!
- Instead, randomly discard low-contribution samples in a way that leaves estimator unbiased

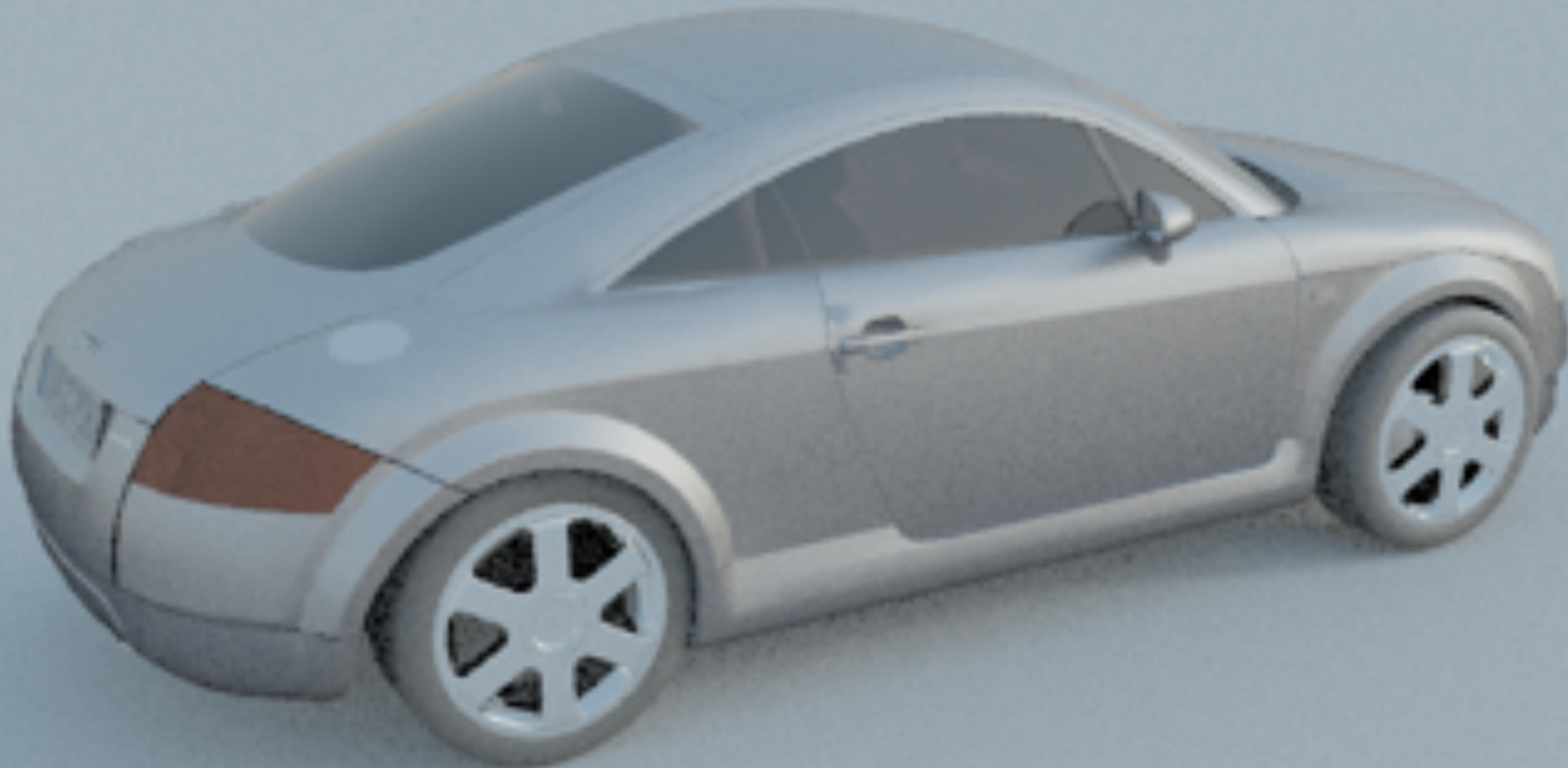
# Russian roulette

- New estimator: evaluate original estimator with probability  $p_{\text{rr}}$ , reweight. Otherwise ignore.
- Same expected value as original estimator:

$$p_{\text{rr}}E\left[\frac{X}{p_{\text{rr}}}\right] + E[(1 - p_{\text{rr}})0] = E[X]$$



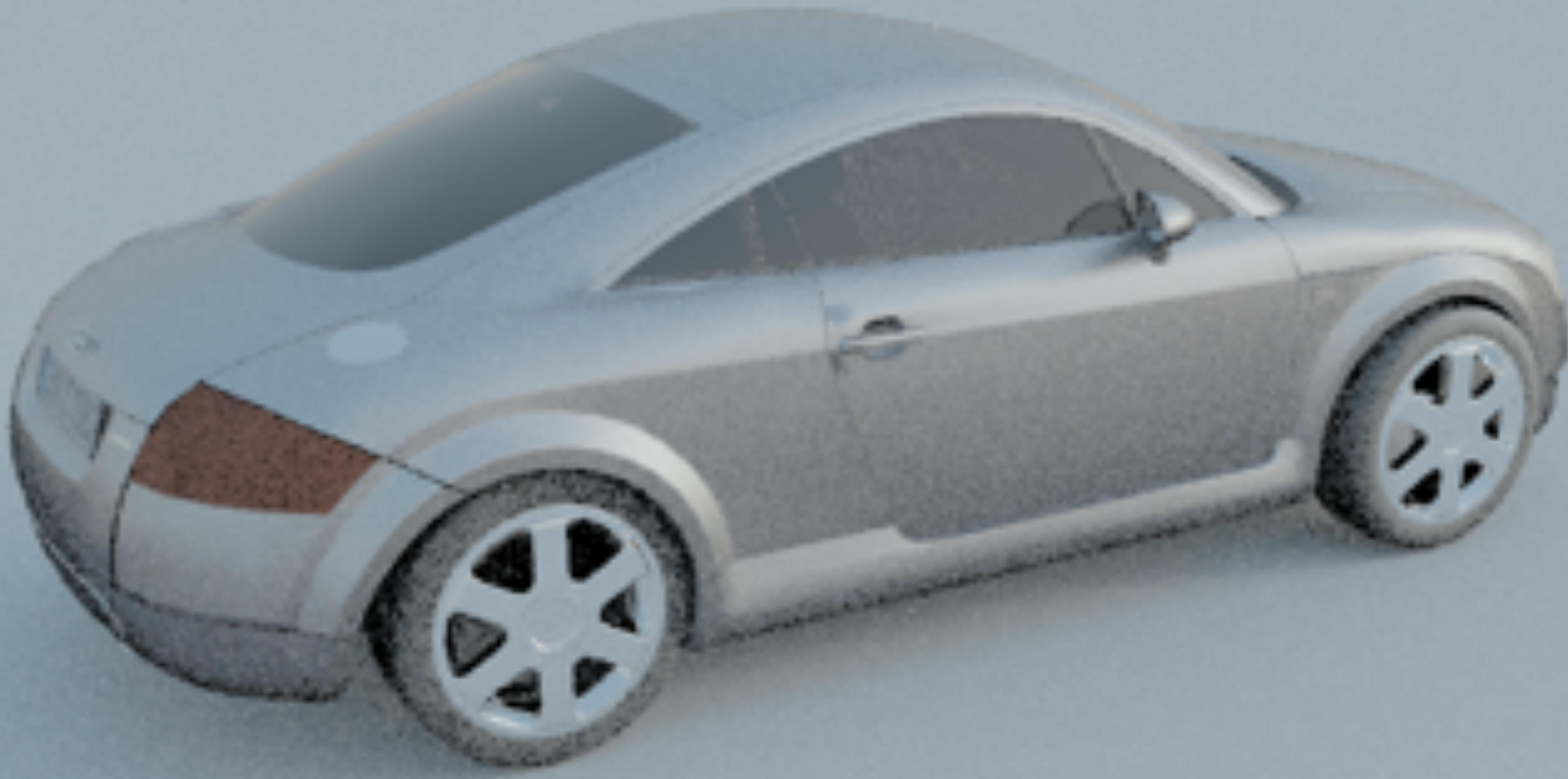
**No Russian roulette: 6.4 seconds**



**Russian roulette: terminate 50% of all contributions with  
luminance less than 0.25: 5.1 seconds**



**Russian roulette: terminate 50% of all contributions with  
luminance less than 0.5: 4.9 seconds**



**Russian roulette: terminate 90% of all contributions with  
luminance less than 0.125: 4.8 seconds**

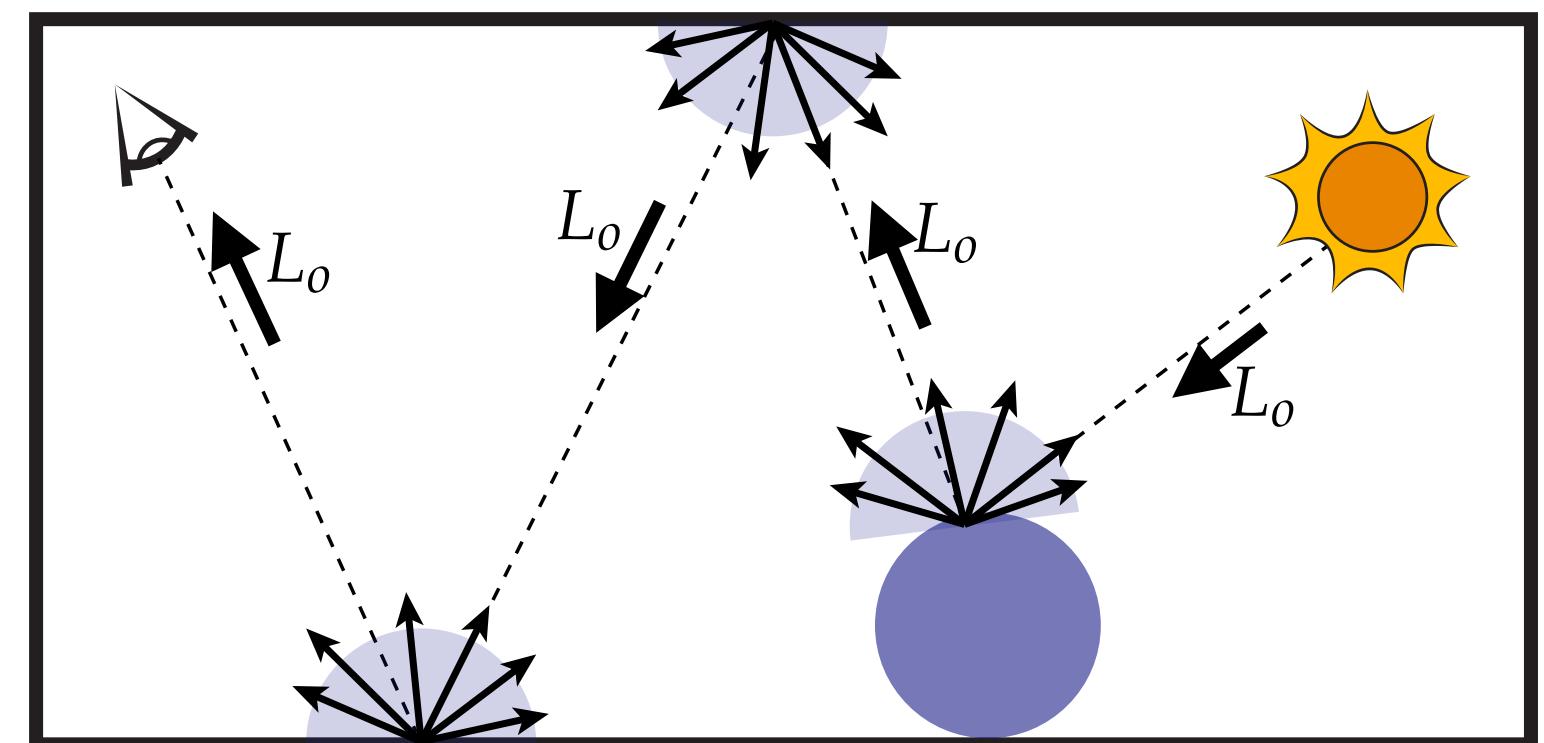


**Russian roulette: terminate 90% of all contributions with  
luminance less than 1: 3.6 seconds**

# Monte Carlo Rendering—Summary

- Light hitting a point (e.g., pixel) described by rendering equation
  - Expressed as recursive integral
  - Can use Monte Carlo to estimate this integral
  - Need to be intelligent about how to sample!

$$L_o(\mathbf{p}, \omega_o) = L_e(\mathbf{p}, \omega_o) + \int_{\mathcal{H}^2} f_r(\mathbf{p}, \omega_i \rightarrow \omega_o) L_i(\mathbf{p}, \omega_i) \cos \theta d\omega_i$$



# Next time:

- Variance reduction—how do we get the most out of our samples?

