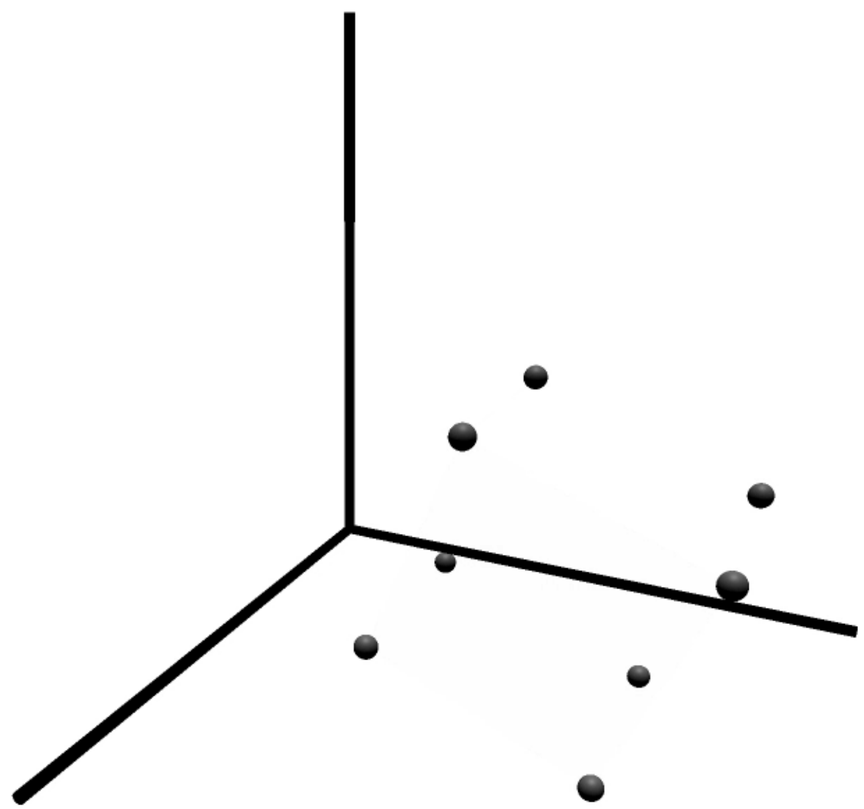
Spatial Transformations

Computer Graphics
CMU 15-462/15-662

Spatial Transformation

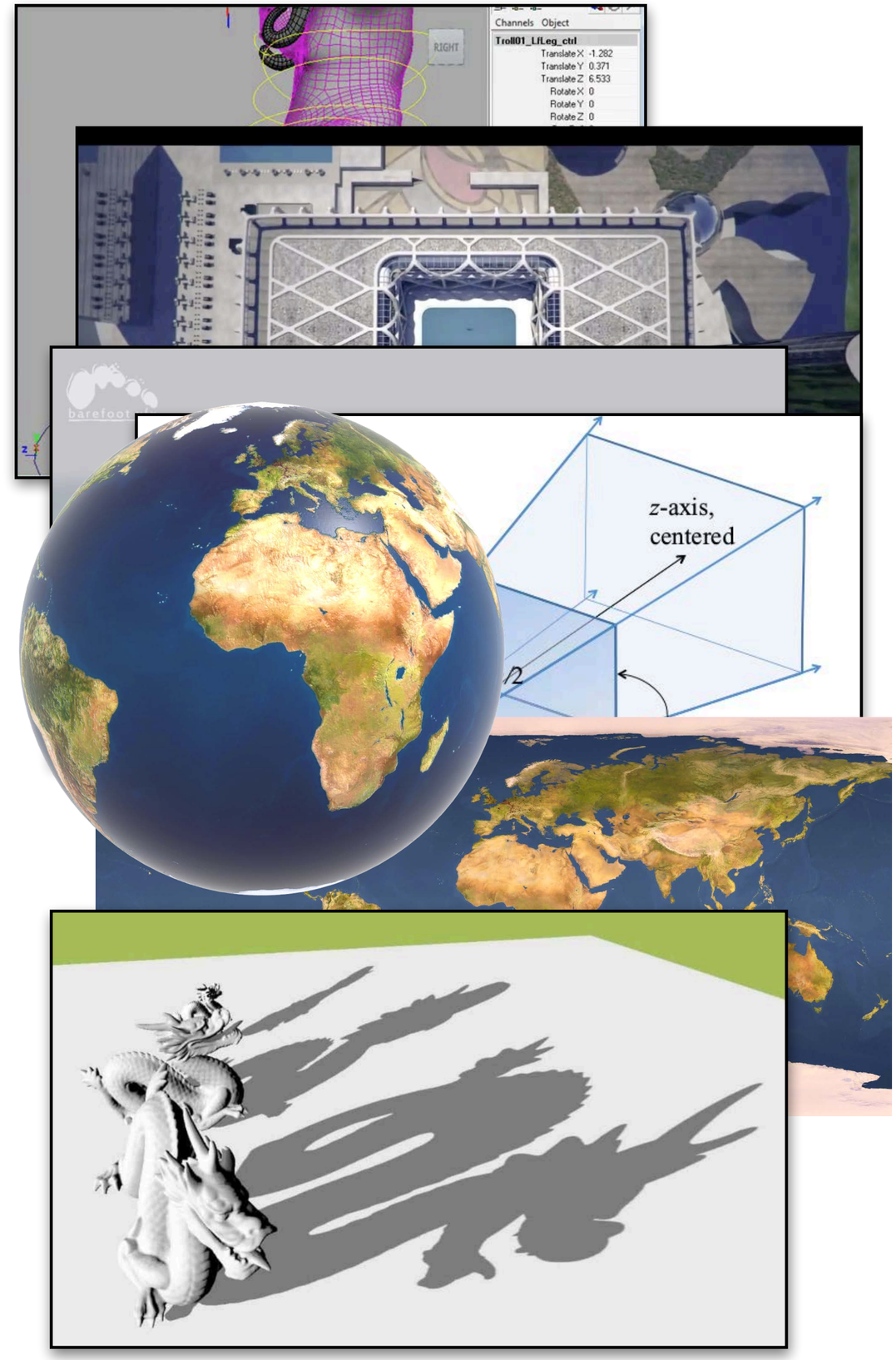
- Basically any function that assigns each point a new location
- Today we'll focus on common transformations of space (rotation, scaling, etc.) encoded by <u>linear</u> maps



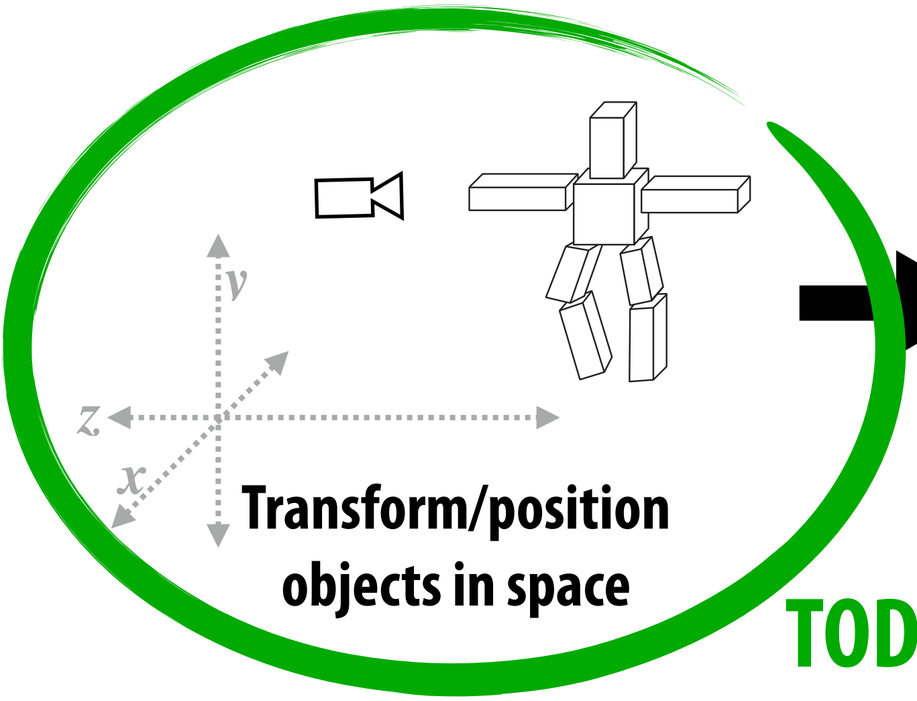
$$f : \mathbb{R}^n \to \mathbb{R}^n$$

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
  - …

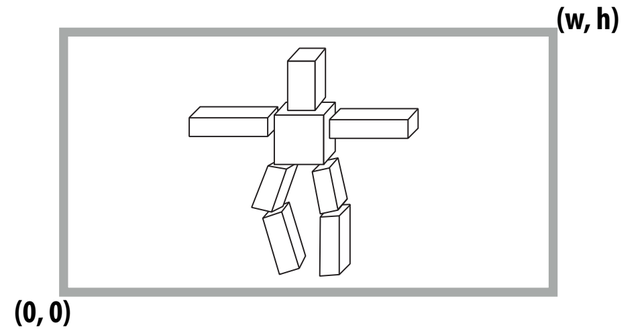


The Rasterization Pipeline

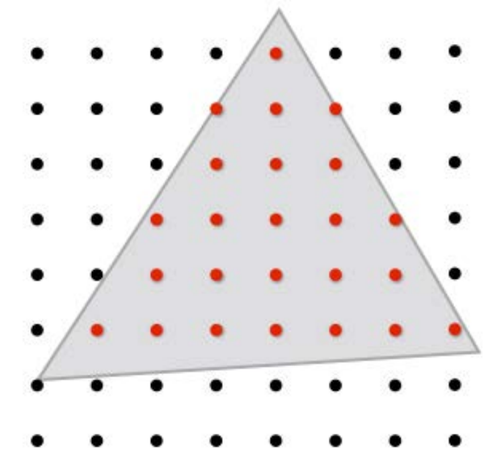


Transform/position objects in space

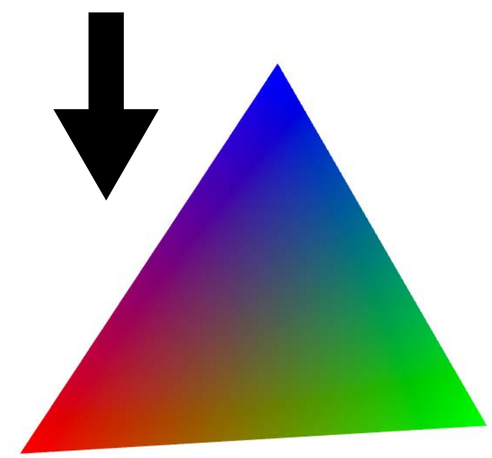
TODAY



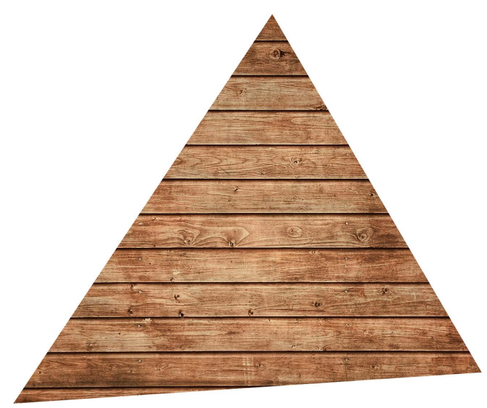
Project objects onto the screen



Sample triangle coverage



Interpolate triangle attributes at covered samples



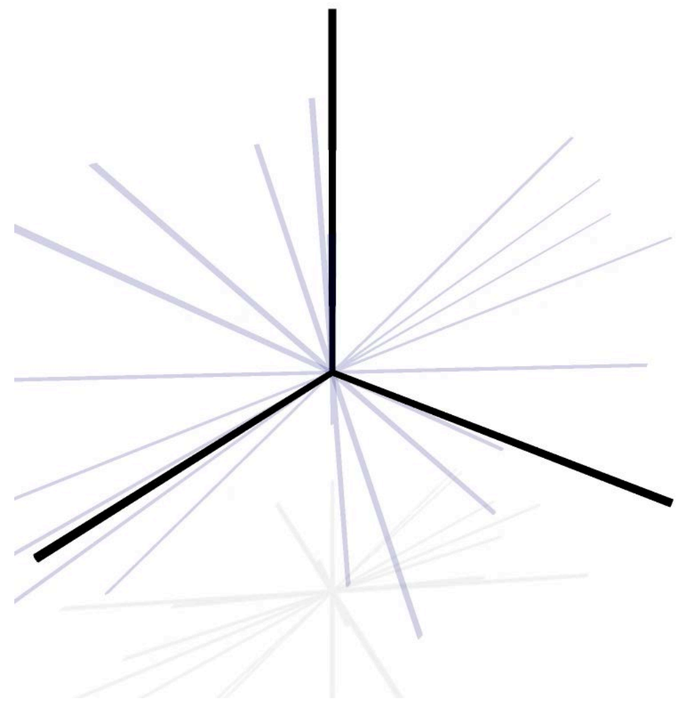
Sample texture maps / evaluate shaders

Combine samples into final image (depth, alpha, …)

Review: Linear Maps

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

Geometrically: it maps <u>lines</u> to <u>lines</u>, and preserves the origin



Algebraically: preserves vector space operations (addition & scaling)

$$\begin{tikzcd}
\mathbf{x}, \mathbf{y} \arrow[r, "\text{add first}"] \arrow[d, "\text{apply f first}"'] & \mathbf{x}+\mathbf{y} \arrow[d, "\text{then apply f}"] \\
f(\mathbf{x}), f(\mathbf{y}) \arrow[r, "\text{then add}"] & \begin{array}{c} f(\mathbf{x})+f(\mathbf{y}) \\ = \\ f(\mathbf{x}+\mathbf{y}) \end{array}
\end{tikzcd}$$

Why do we care about *linear* transformations?

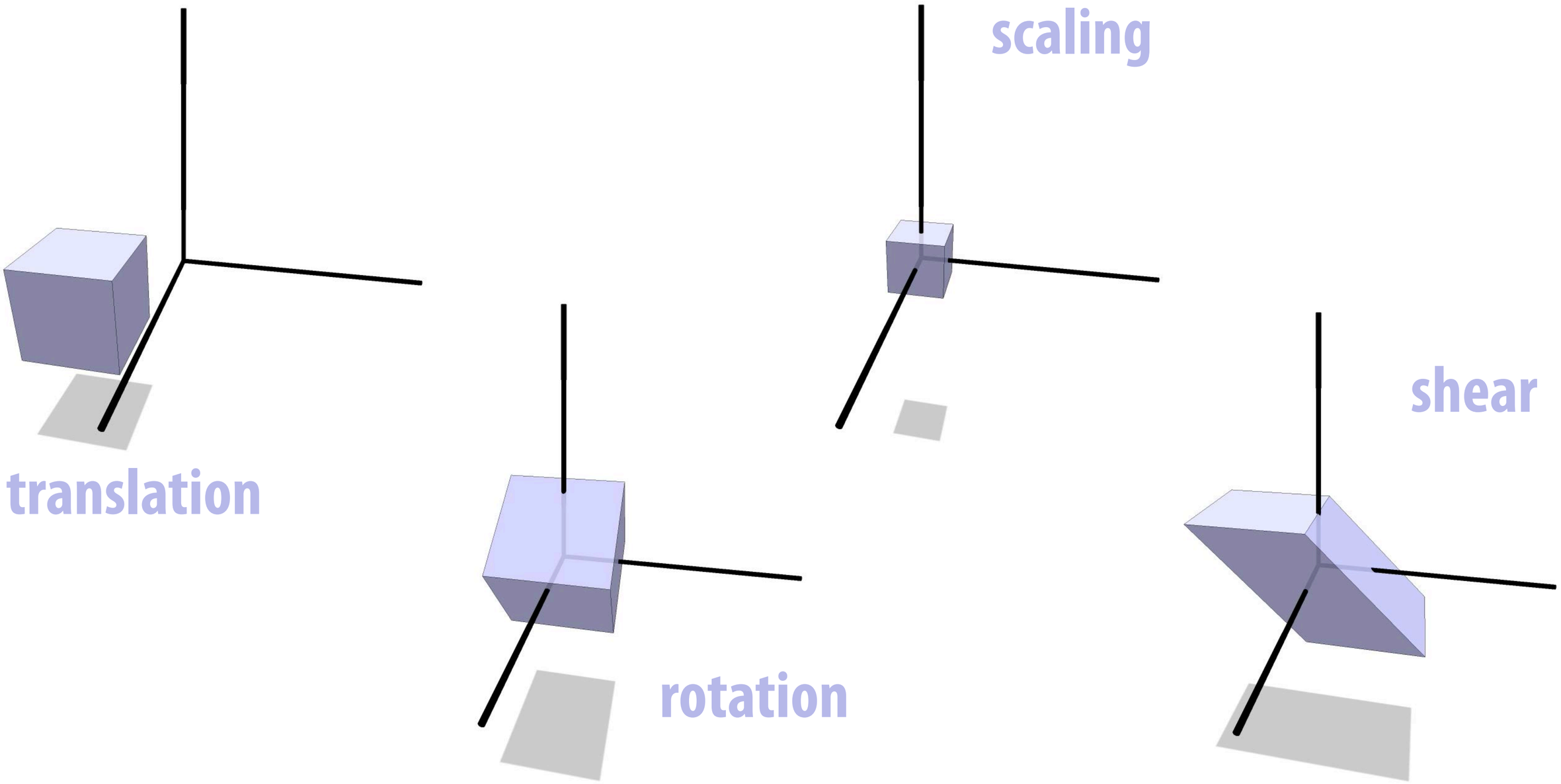
- Cheap to apply
- Usually pretty easy to solve for (linear systems)
- Composition of linear transformations is linear
  - product of <u>many</u> matrices is a <u>single</u> matrix
  - gives uniform representation of transformations
  - simplifies graphics algorithms, systems (e.g., GPUs & APIs)

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

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 <u>not</u> inspect a formula!)

Invariants of Transformation

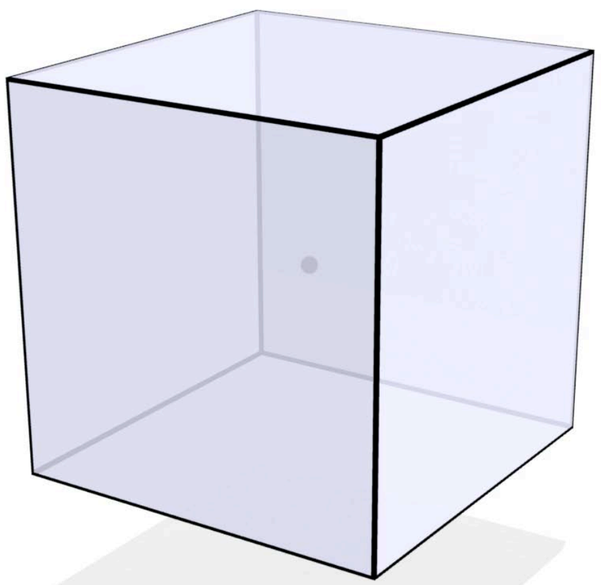
A transformation is determined by the <u>invariants</u> it preserves

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

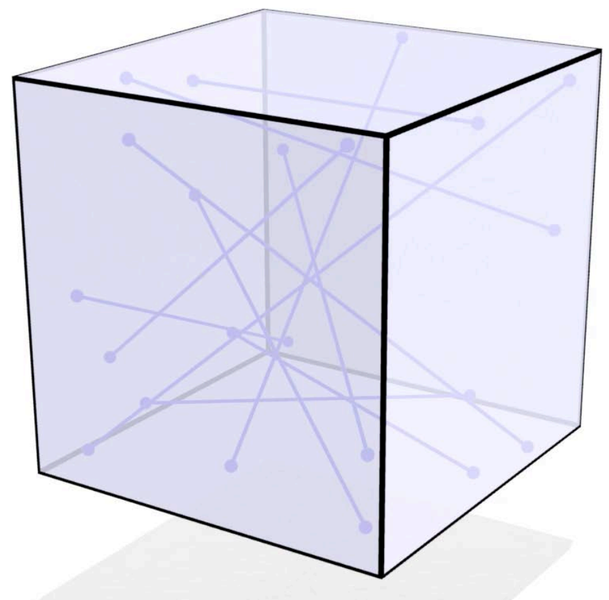
(Essentially how your brain "knows" what kind of transformation you're looking at…)

Rotation

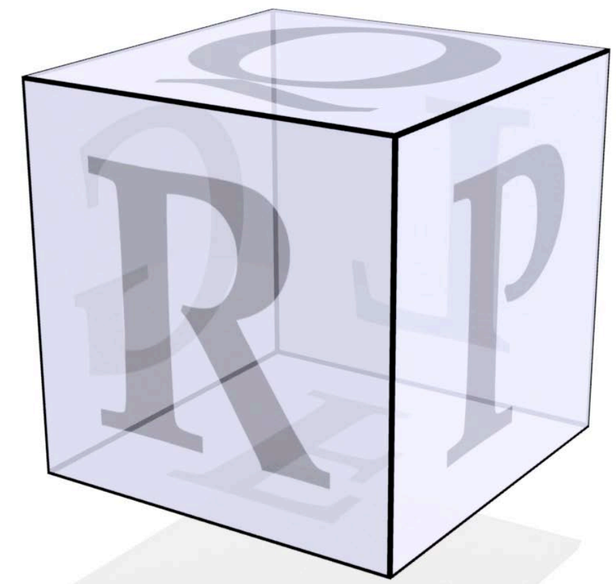
Rotations defined by three basic properties:



keeps origin fixed



preserves distances



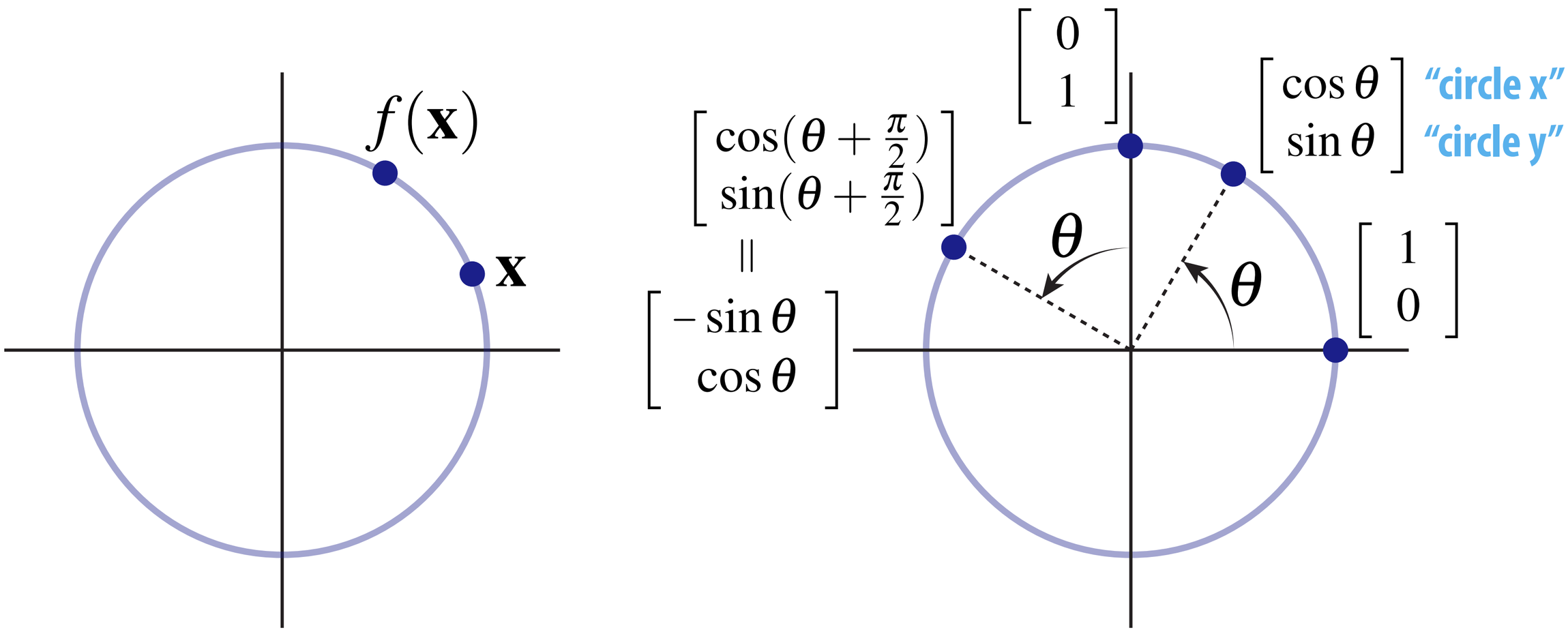
preserves orientation

First two properties together imply that rotations are <u>linear</u>.

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

2D Rotations—Matrix Representation

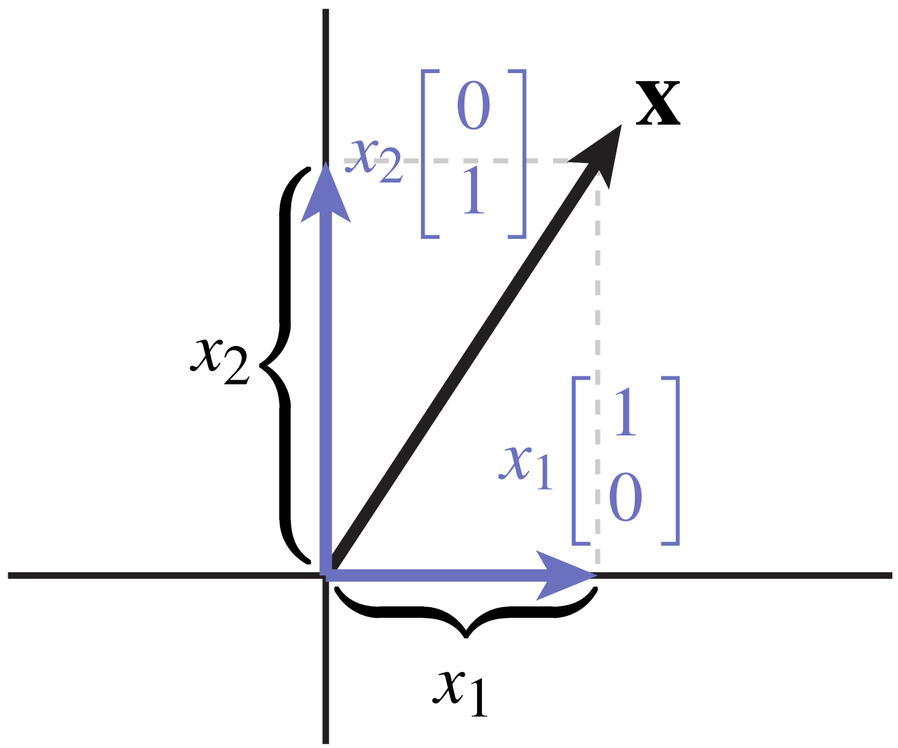
Rotations preserve distances and the origin—hence, a 2D rotation by an angle θ maps each point $\mathbf{x}$ to a point $f_\theta(\mathbf{x})$ on the circle of radius $|\mathbf{x}|$:

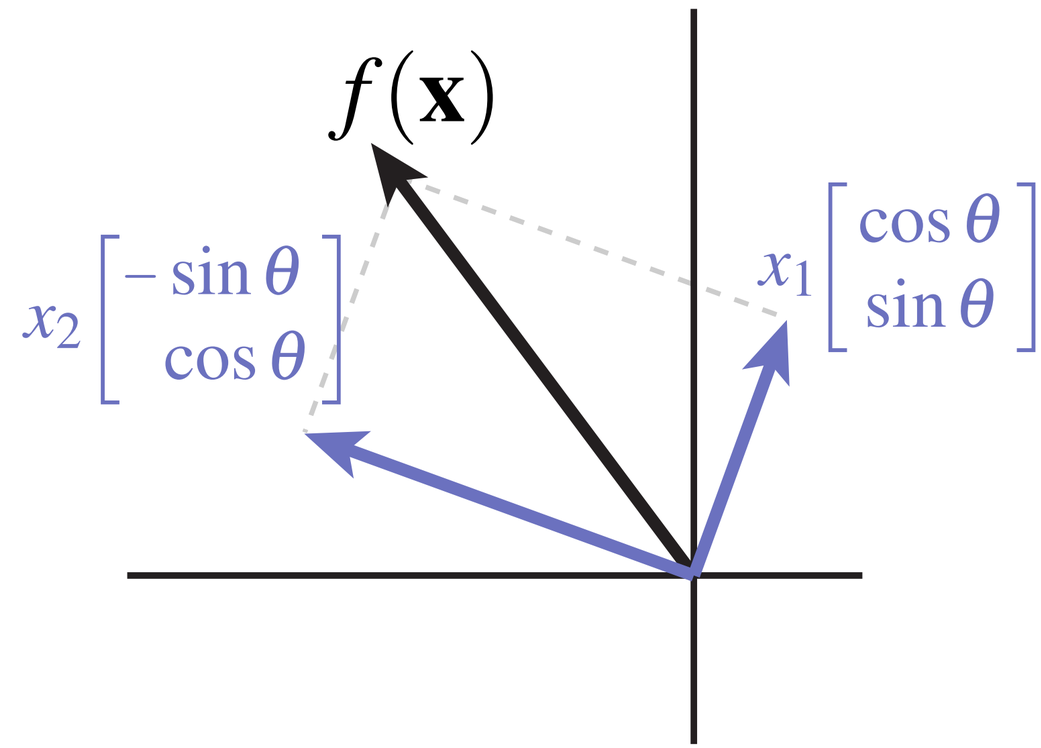


- **Where does $\mathbf{x} = (1,0)$ go if we rotate by θ (counter-clockwise)?**
- **How about $\mathbf{x} = (0,1)$?**

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

2D Rotations—Matrix Representation





$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = x_1 \begin{bmatrix} 1 \\ 0 \end{bmatrix} + x_2 \begin{bmatrix} 0 \\ 1 \end{bmatrix} \qquad f(\mathbf{x}) = x_1 \begin{bmatrix} \cos\theta \\ \sin\theta \end{bmatrix} + x_2 \begin{bmatrix} -\sin\theta \\ \cos\theta \end{bmatrix}$$

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

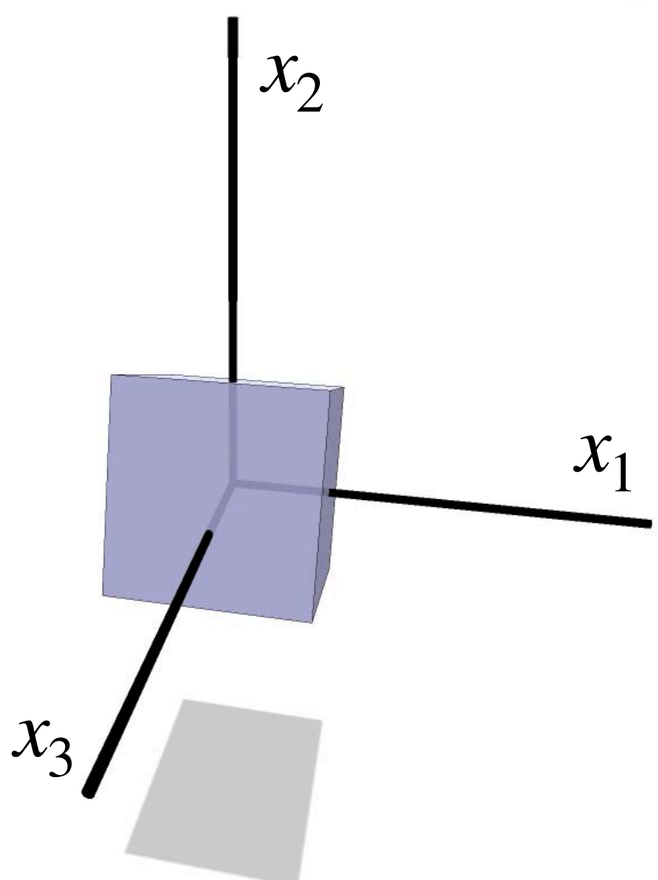
$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin(\theta) \\ 0 & \sin\theta & \cos(\theta) \end{bmatrix}$$

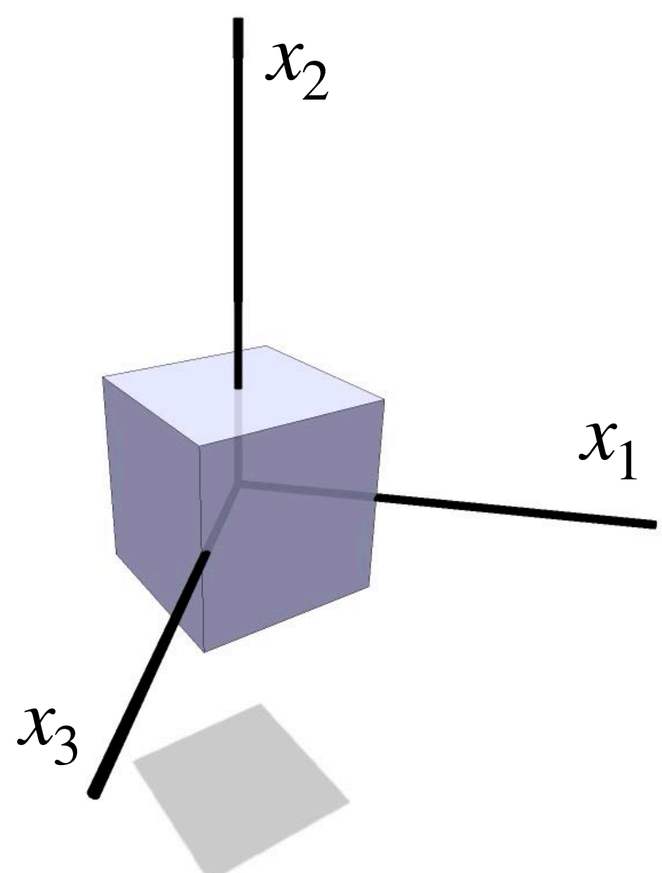
rotate around x_2

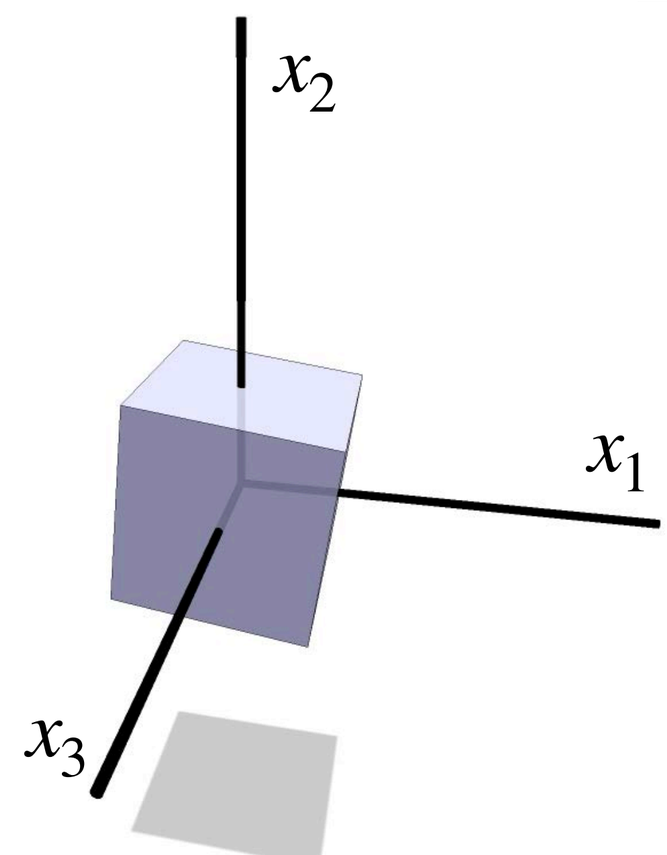
$$\begin{bmatrix} \cos\theta & 0 & \sin(\theta) \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos(\theta) \end{bmatrix}$$

rotate around x_3

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

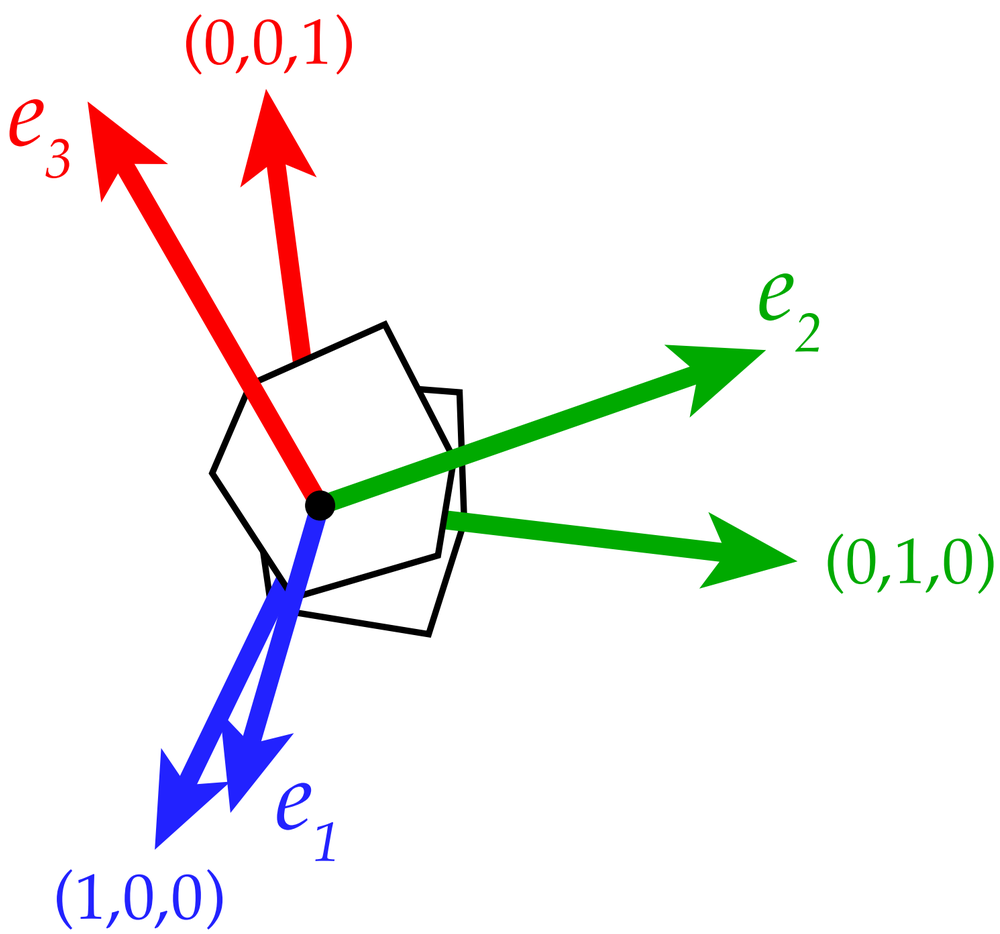






Rotations—Transpose as Inverse

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



$$\overset{R^\mathsf{T}}{\begin{bmatrix} \text{---} & e_1^T & \text{---} \\ \text{---} & e_2^T & \text{---} \\ \text{---} & e_3^T & \text{---} \end{bmatrix}} \overset{R}{\begin{bmatrix} | & | & | \\ e_1 & e_2 & e_3 \\ | & | & | \end{bmatrix}} = \underset{I}{\begin{bmatrix} e_1^T e_1 & e_1^T e_2 & e_1^T e_3 \\ e_2^T e_1 & e_2^T e_2 & e_2^T e_3 \\ e_3^T e_1 & e_3^T e_2 & e_3^T e_3 \end{bmatrix}} \begin{matrix} 1; & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{matrix}$$

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

Reflections

- Q: Does <u>every</u> matrix $Q^\mathsf{T} Q = I$ describe a rotation?
- Remember that rotations must preserve the <u>origin</u>, preserve <u>distances</u>, and preserve <u>orientation</u>
- Consider for instance this matrix:

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

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 <u>orientation</u>)

Orthogonal Transformations

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

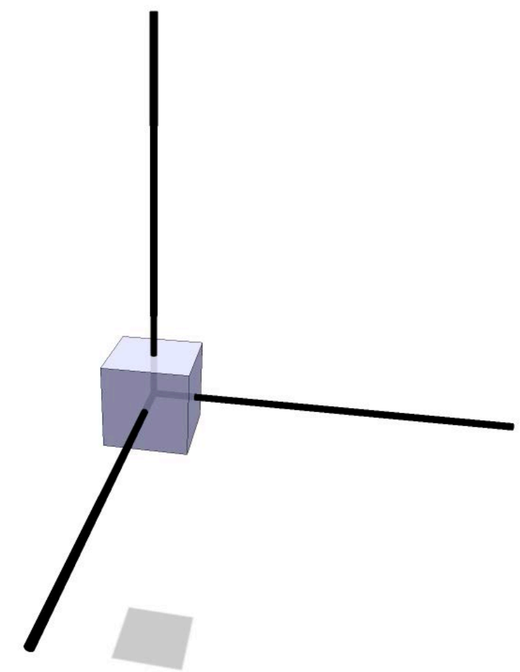
rotation

reflection

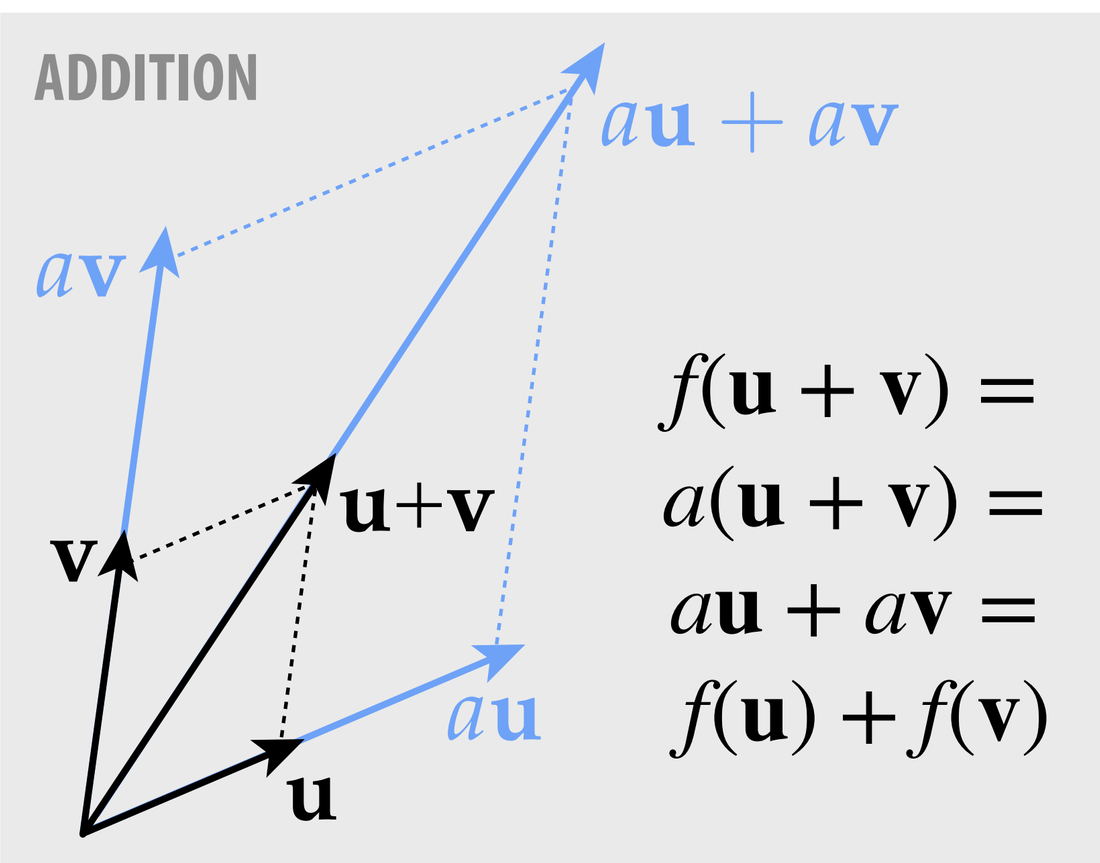
Scaling

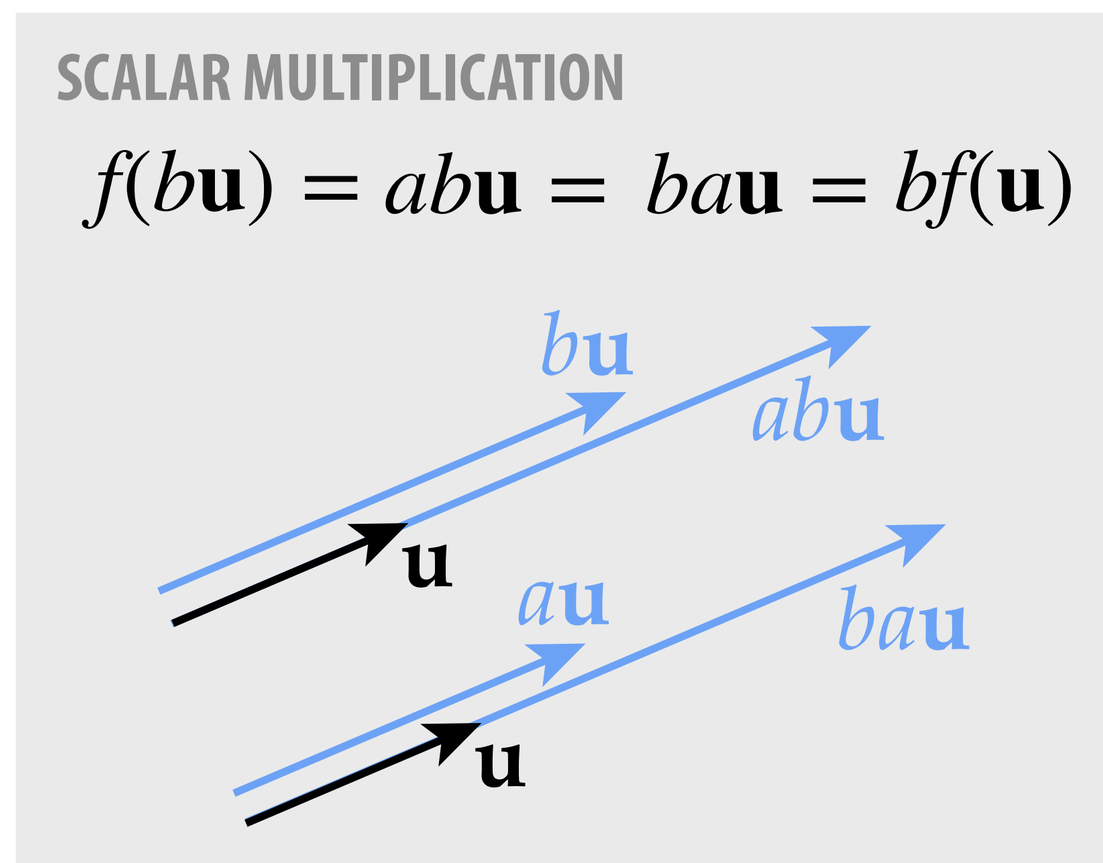
*assuming $a \neq 0,\ \mathbf{u} \neq 0$

- Each vector $\mathbf{u}$ gets mapped to a scalar multiple
  - $f(\mathbf{u}) = a\mathbf{u}, \quad a \in \mathbb{R}$
- Preserves the <u>direction</u> of all vectors*
  - $\dfrac{\mathbf{u}}{|\mathbf{u}|} = \dfrac{a\mathbf{u}}{|a\mathbf{u}|}$



- 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 <u>matrix</u>?

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 <u>negative</u>?

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 <u>preserved.</u>

What about 3D?

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

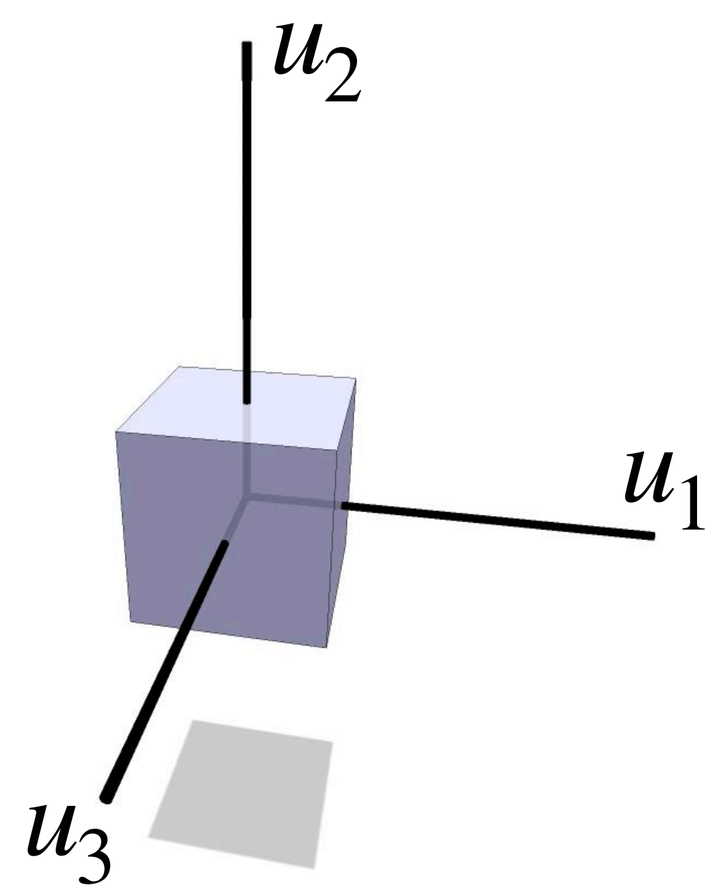
$$\begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{bmatrix}$$

Now we have *three* reflections, and so orientation is <u>reversed!</u>

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:**

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

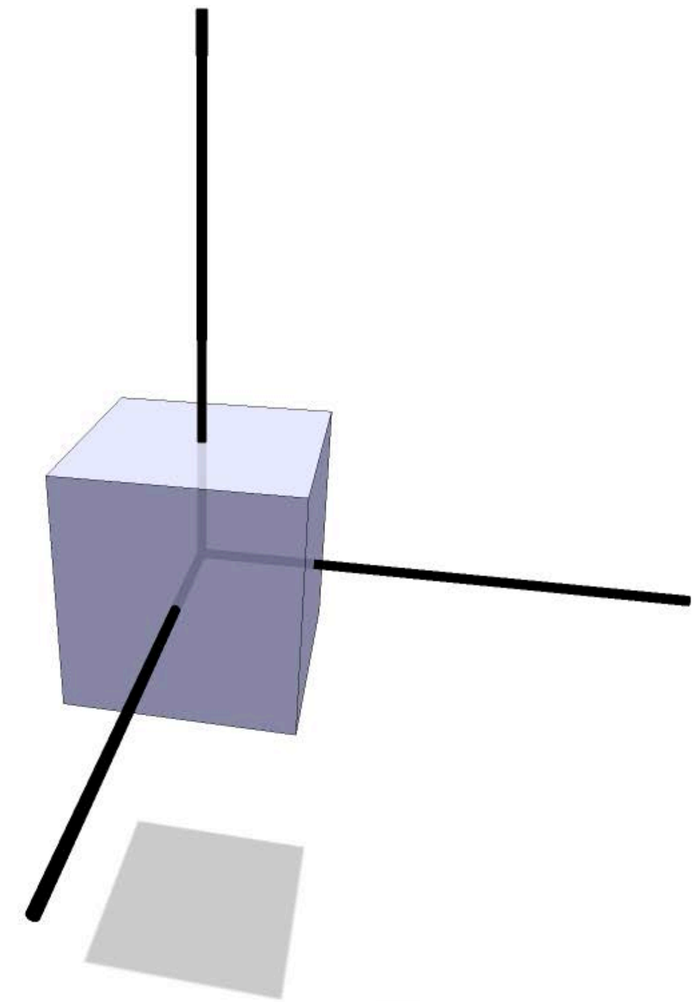


Ok, but what if we want to scale along some *other* axes?

Nonuniform Scaling

- <u>Idea.</u> We could:
  - rotate to the new axes (R)
  - apply a diagonal scaling (D)
  - rotate *back** to the original axes ($R^\top$)

- Notice that the overall transformation is represented by a <u>symmetric</u> matrix
$A := R^\top D R$



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

Q: Do <u>all</u> symmetric matrices represent nonuniform scaling (for some choice of axes)?

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

Spectral Theorem

- A: Yes! <u>Spectral theorem</u> says a symmetric matrix $A = A^\top$ has
  - orthonormal eigenvectors $e_1, \ldots, e_n \in \mathbb{R}^n$
  - real eigenvalues $\lambda_1, \ldots, \lambda_n \in \mathbb{R}$

$$\boxed{Ae_i = \lambda_i e_i}$$

- Can also write this relationship as $AR = RD$, where

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

- Equivalently, $A = RDR^\top$
- Hence, <u>every symmetric matrix performs a non-uniform scaling along some set of orthogonal axes</u>.
- 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

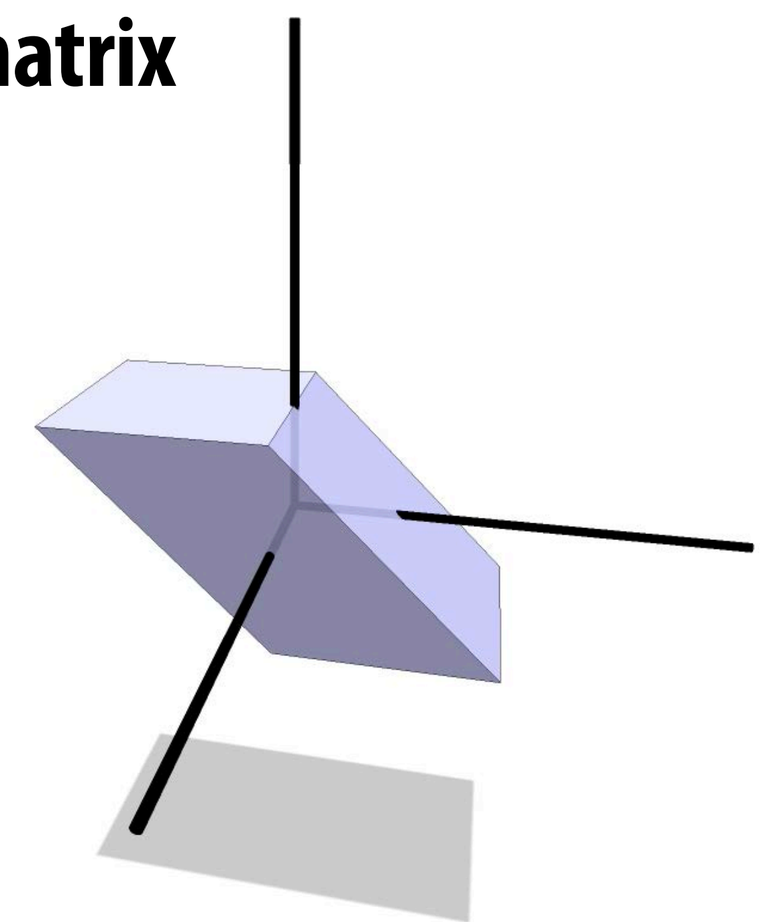
$$A_{\mathbf{u},\mathbf{v}} = I + \mathbf{u}\mathbf{v}^\mathsf{T}$$

Example.

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

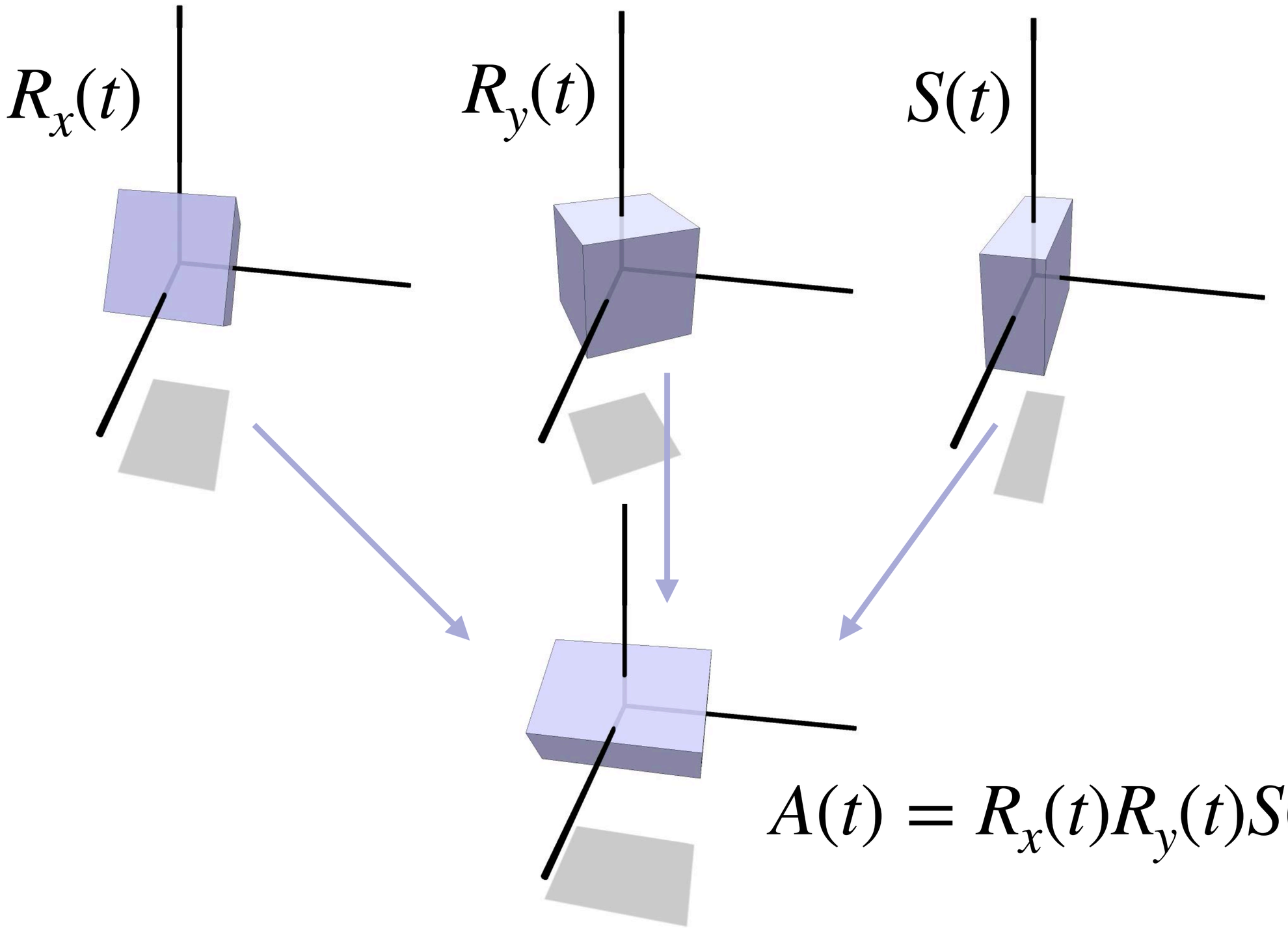
$$\mathbf{v} = (0,1,0)$$

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



Composite Transformations

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



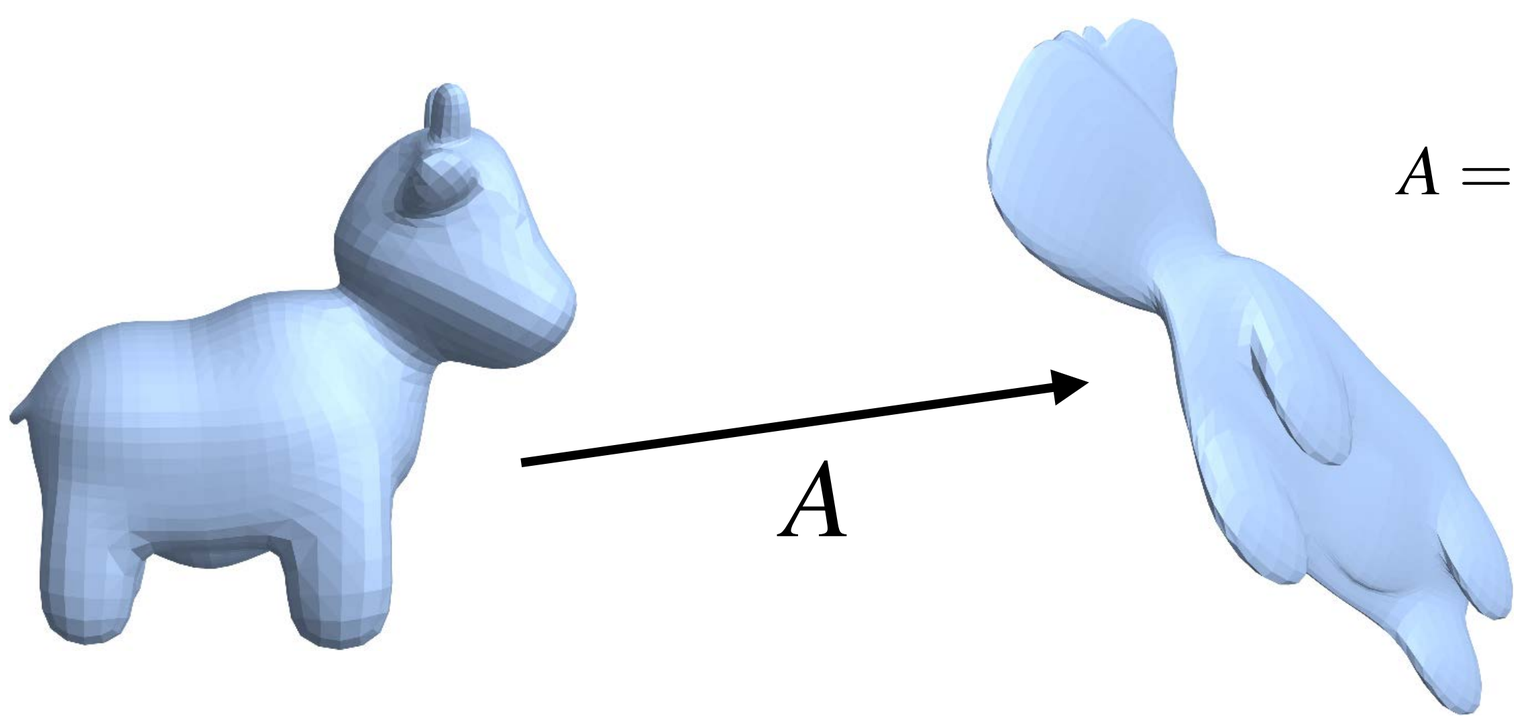
$$A(t) = R_x(t)R_y(t)S(t)$$

How do we <u>decompose</u> 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:



$$A = \begin{bmatrix} .34 & -.11 & -.89 \\ -.65 & .52 & -.70 \\ .25 & .23 & -.69 \end{bmatrix}$$

Polar & Singular Value Decomposition

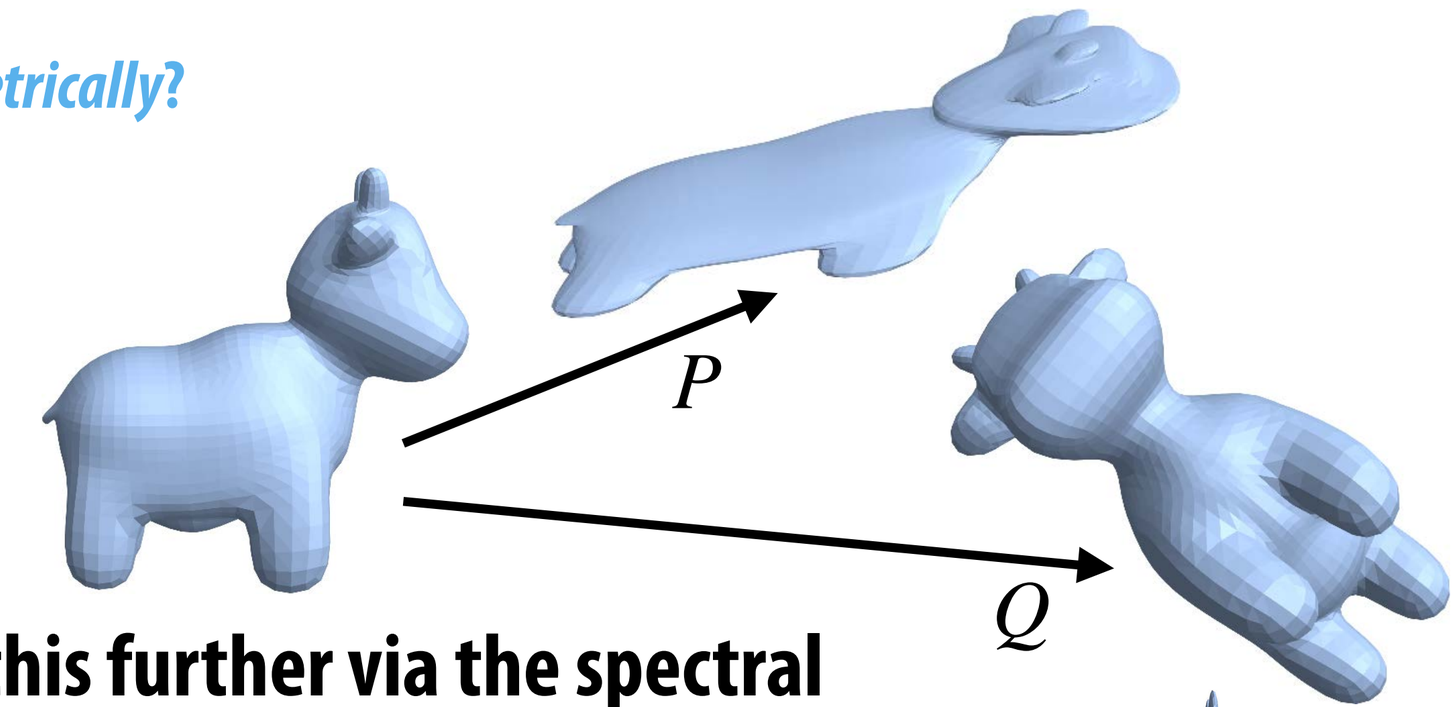
For example, <u>polar decomposition</u> decomposes any matrix A into orthogonal matrix Q and symmetric positive-semidefinite matrix P:

Q: What do each of the parts mean *geometrically*?

rotation/reflection

nonnegative, nonuniform scaling

$$A = QP$$



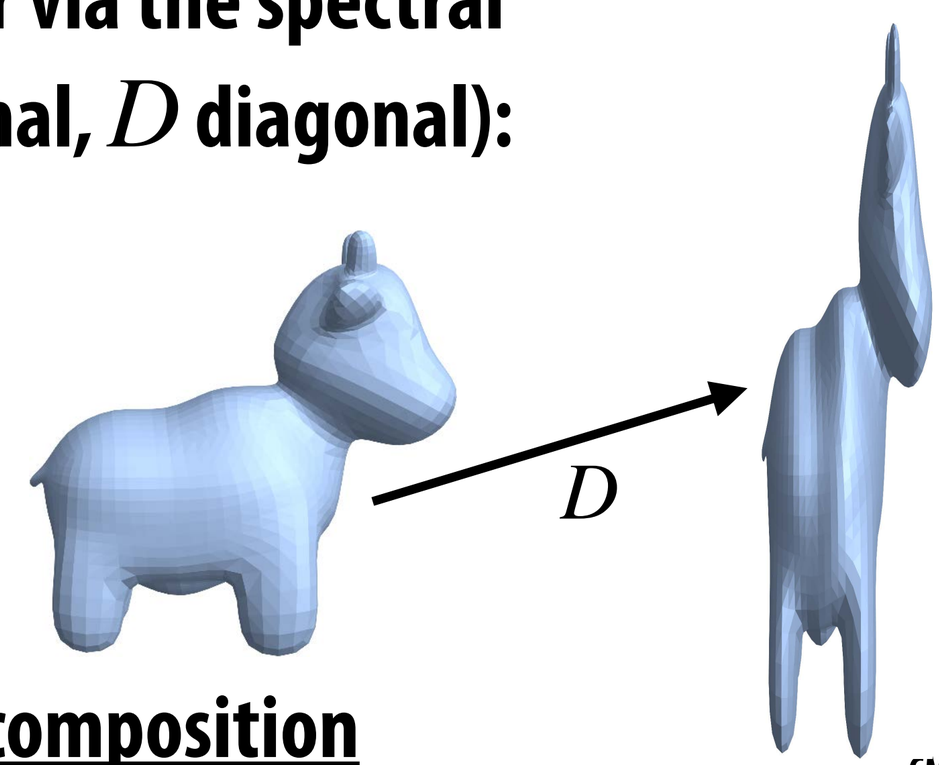
Since P is symmetric, can take this further via the spectral decomposition $P = VDV^\mathsf{T}$ (V orthogonal, D diagonal):

$$A = \underbrace{QV}_{U}DV^\mathsf{T} = UDV^\mathsf{T}$$

rotation

axis-aligned scaling

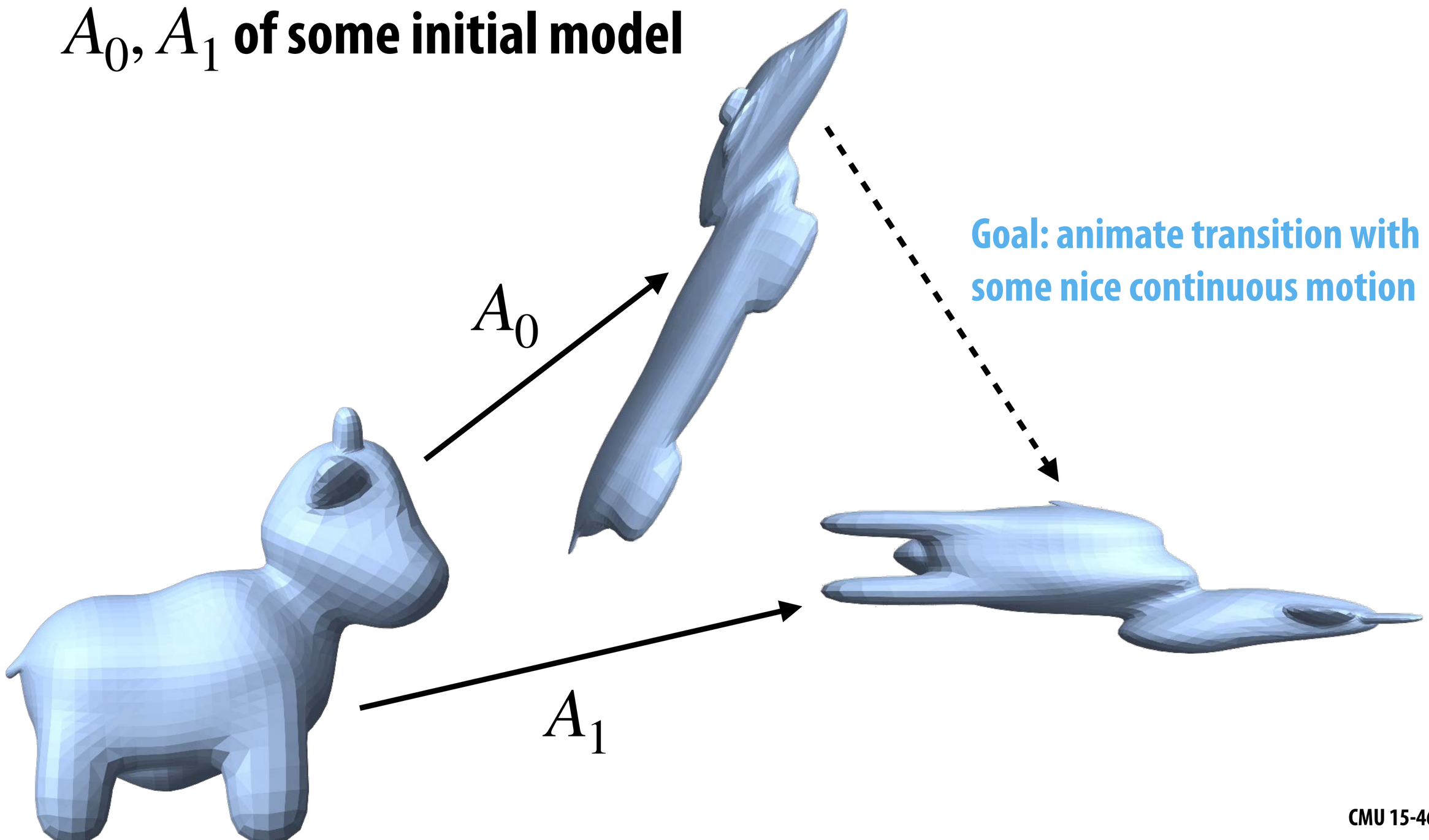
rotation



Result UDV^T is called the <u>singular value decomposition</u>

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 $t \in [0,1]$

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

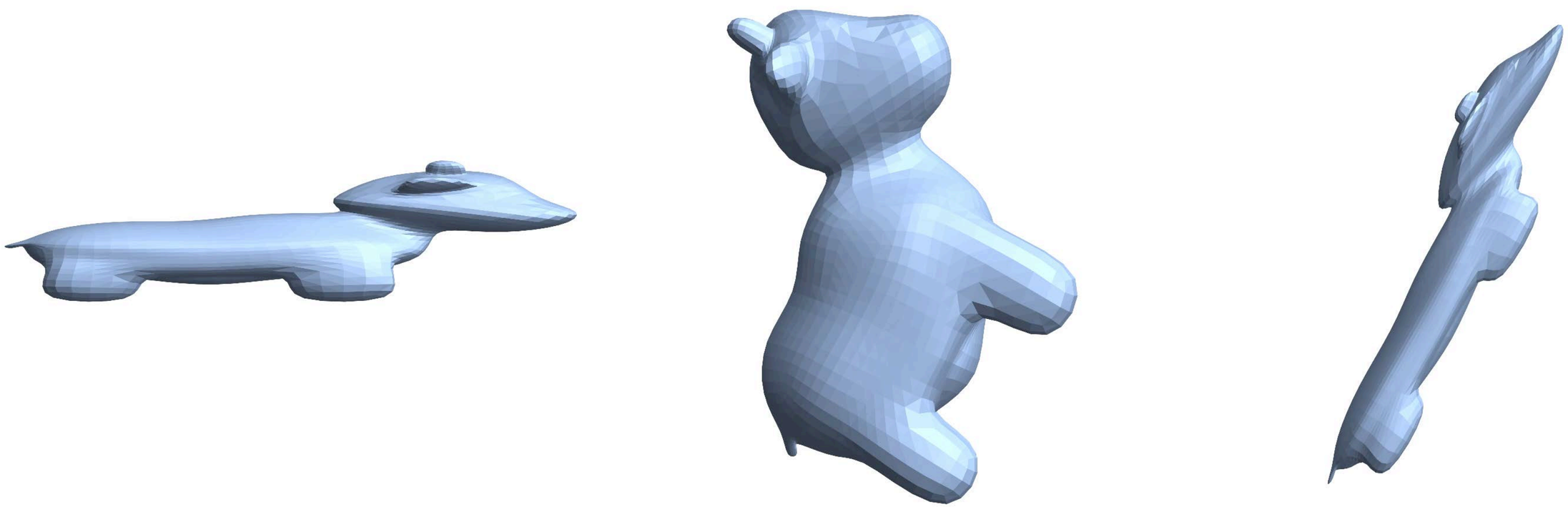
Hits the right start/endpoints… but looks awful in between!

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**



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

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

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

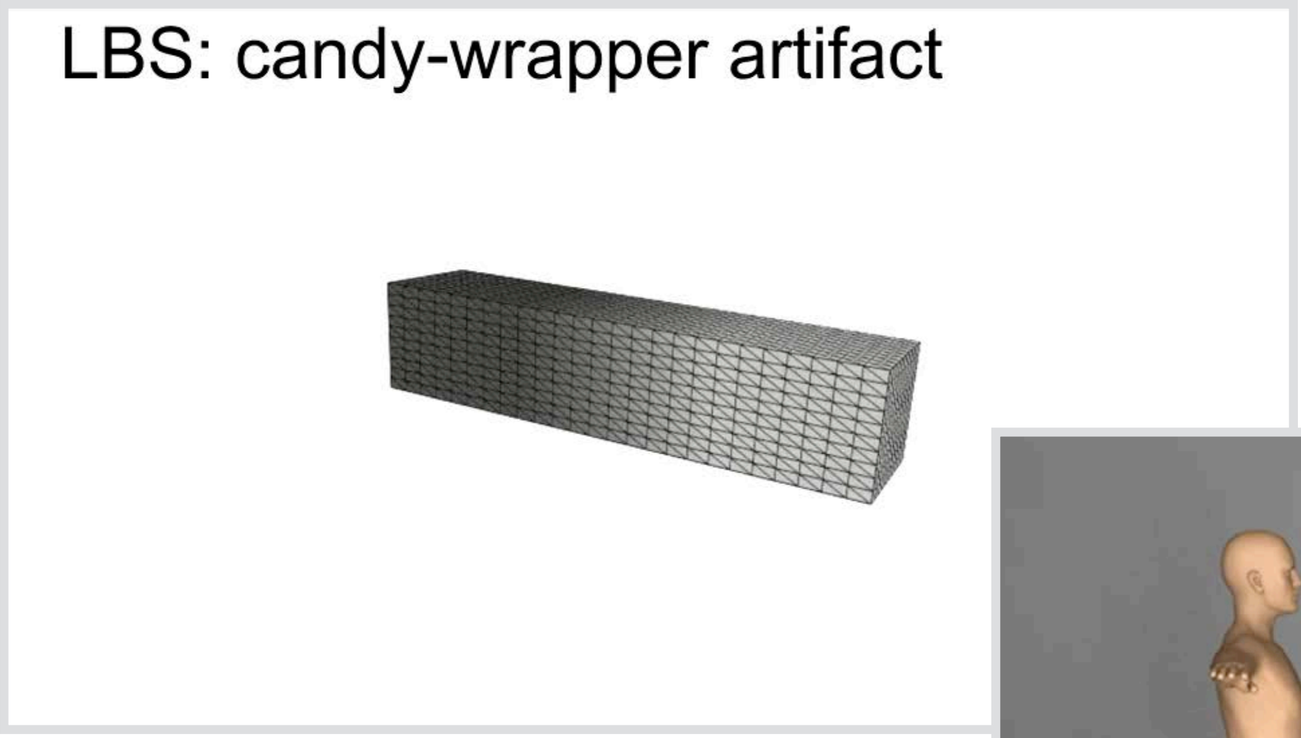
$$A(t) = Q(t)P(t)$$

…looks better!

See: Shoemake & Duff, "Matrix Animation and Polar Decomposition"

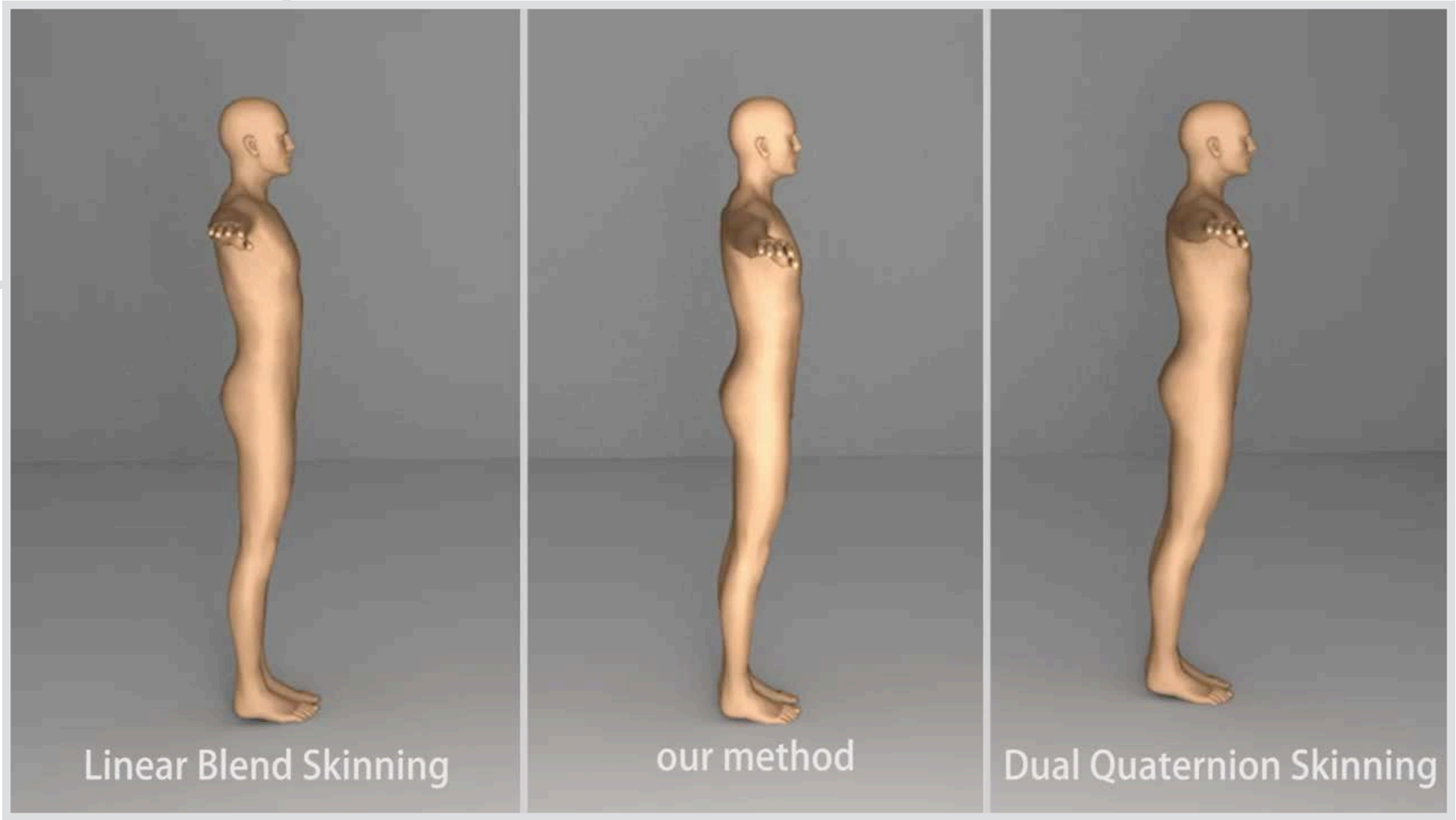
Example: Linear Blend Skinning

- 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…



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

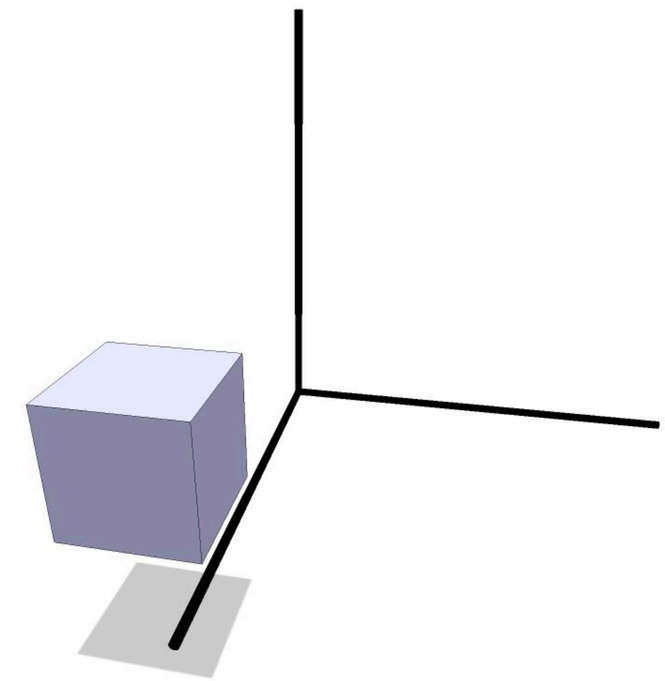
Rumman & Fratarcangeli (2015)
"Position-based Skinning for Soft Articulated Characters"



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 <u>linear</u>?
(Certainly seems to move us along a line…)

Let's carefully check the definition…

<u>additivity</u>

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

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

<u>homogeneity</u>

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

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

A: No! Translation is <u>affine</u>, *not* linear!

Composition of Transformations

- Recall we can compose <u>linear</u> transformations via matrix multiplication:

$$A_3(A_2(A_1\mathbf{x})) = (A_3A_2A_1)\mathbf{x}$$

- 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.)?

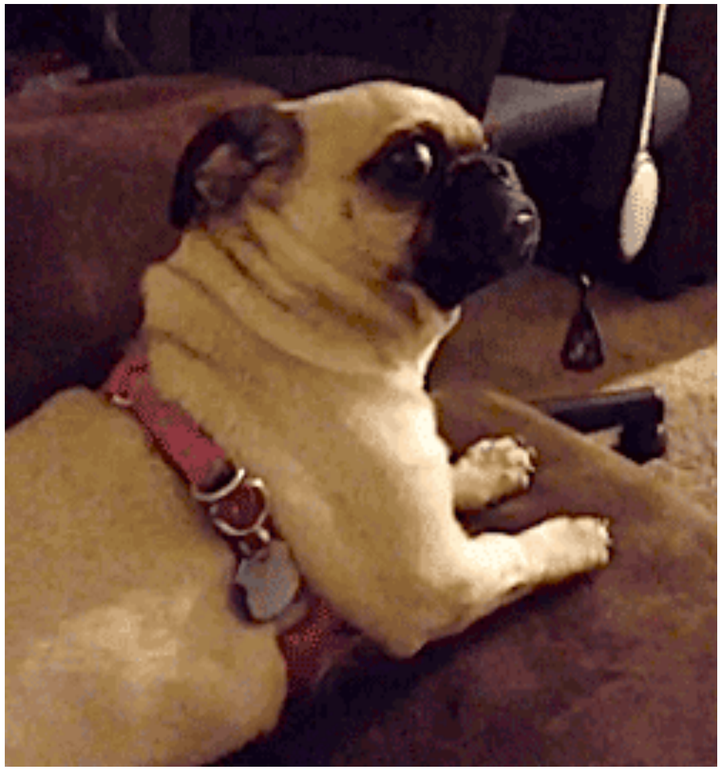
$$A_2(A_1\mathbf{x}+\mathbf{b}_1)+\mathbf{b}_2 = (A_2A_1)\mathbf{x}+(A_2\mathbf{b}_1+\mathbf{b}_2)$$

- 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…

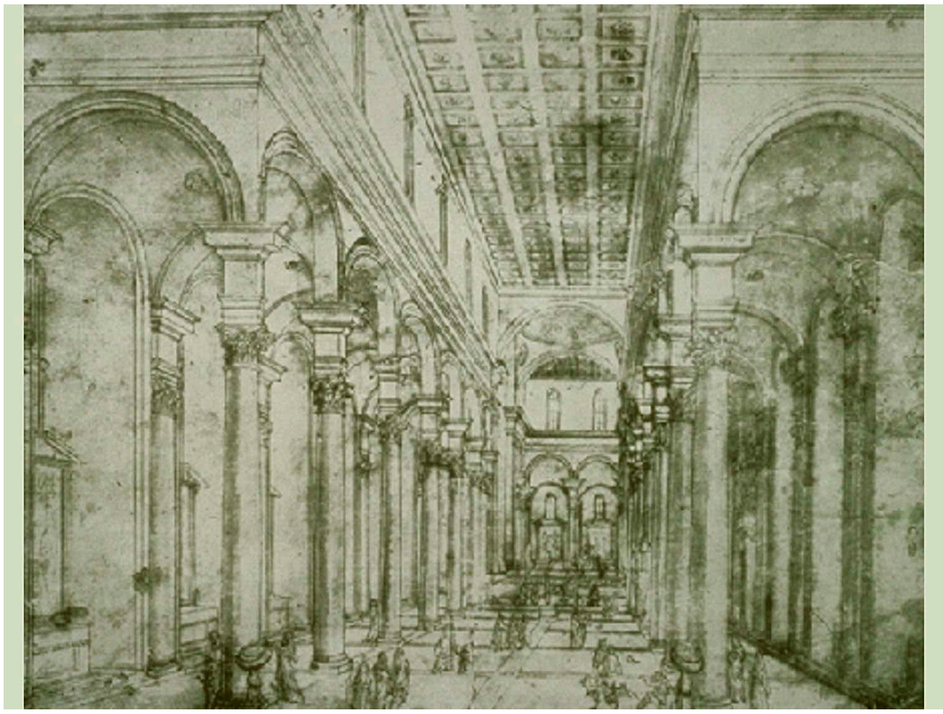
Strange idea:

Maybe translations turn into <u>linear</u> transformations if we go into the 4th dimension…!

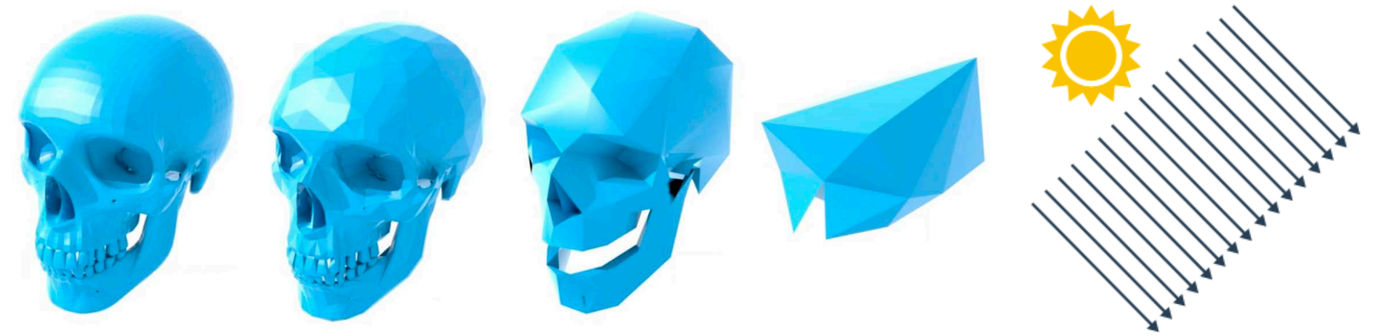


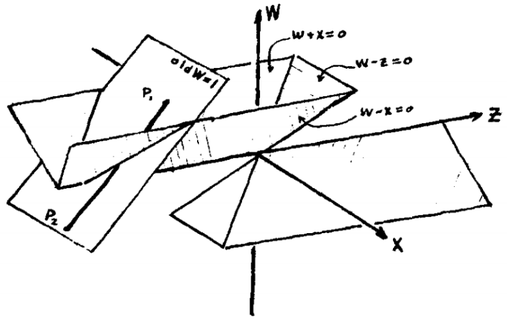
Homogeneous Coordinates

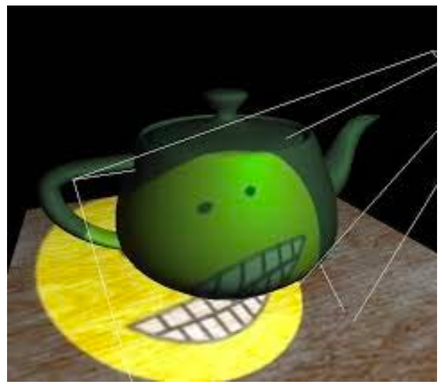
Filippo Brunelleschi, 1428

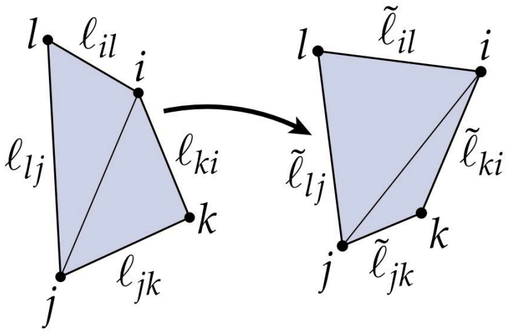


- Came from efforts to study <u>perspective</u>
- Introduced by Möbius as a natural way of assigning coordinates to <u>lines</u>
- Show up naturally in a surprising large number of places in computer graphics:
  - 3D transformations
  - perspective projection
  - quadric error simplification
  - premultiplied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





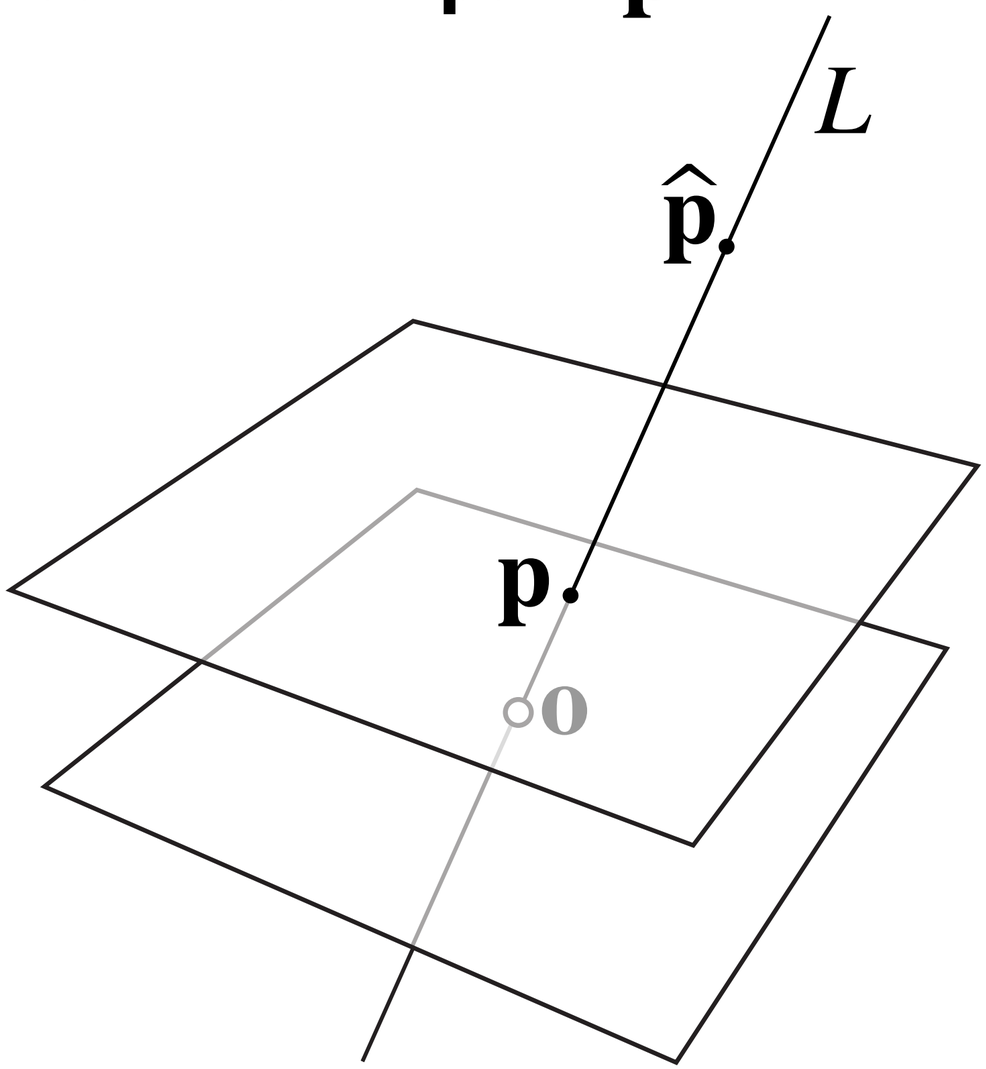


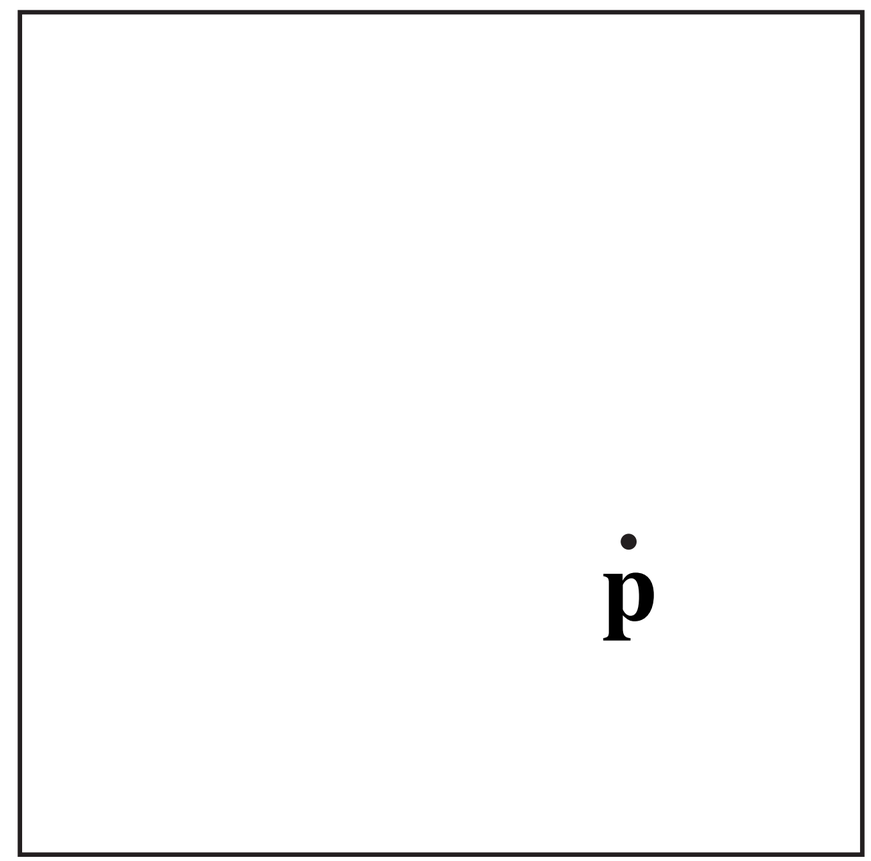


Probably worth understanding!

Homogeneous Coordinates—Basic Idea

- Consider any 2D plane that does not pass through the origin $\mathbf{o}$ in 3D
- Every <u>line</u> through the origin in 3D corresponds to a <u>point</u> in the 2D plane
  - Just find the point $\mathbf{p}$ where the line L pierces the plane



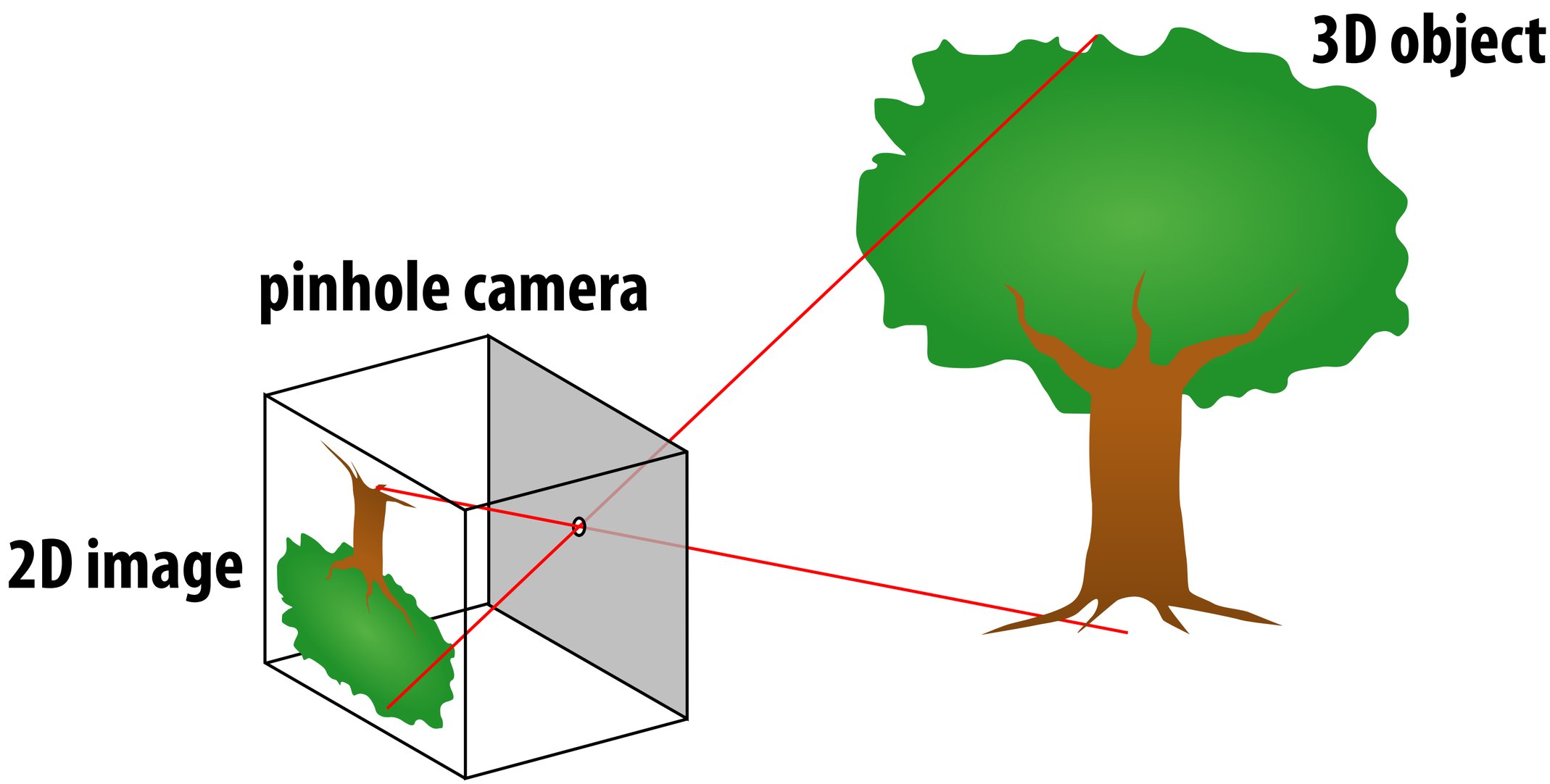


Hence, <u>any</u> point $\hat{\mathbf{p}}$ on the line L can be used to represent the point $\mathbf{p}$.

Q: What does this story remind you of?

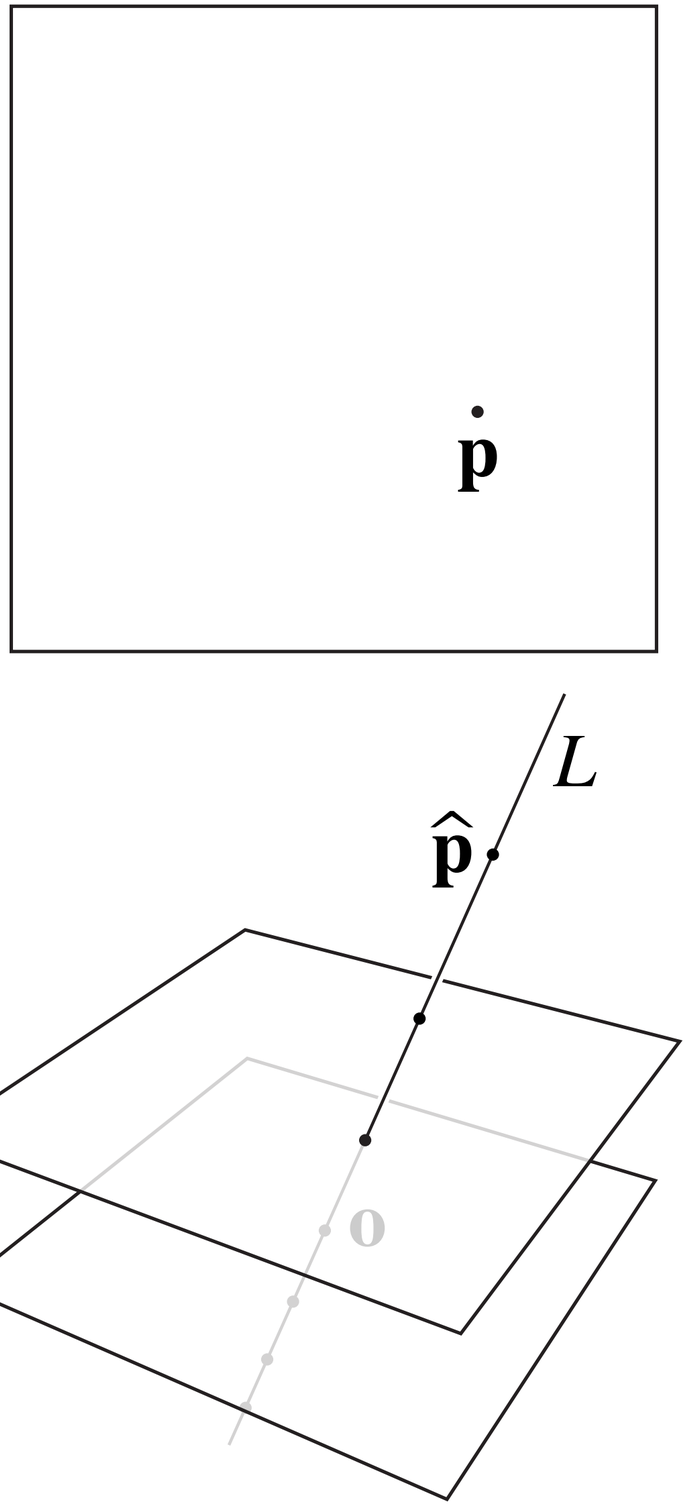
Review: Perspective projection

- 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)



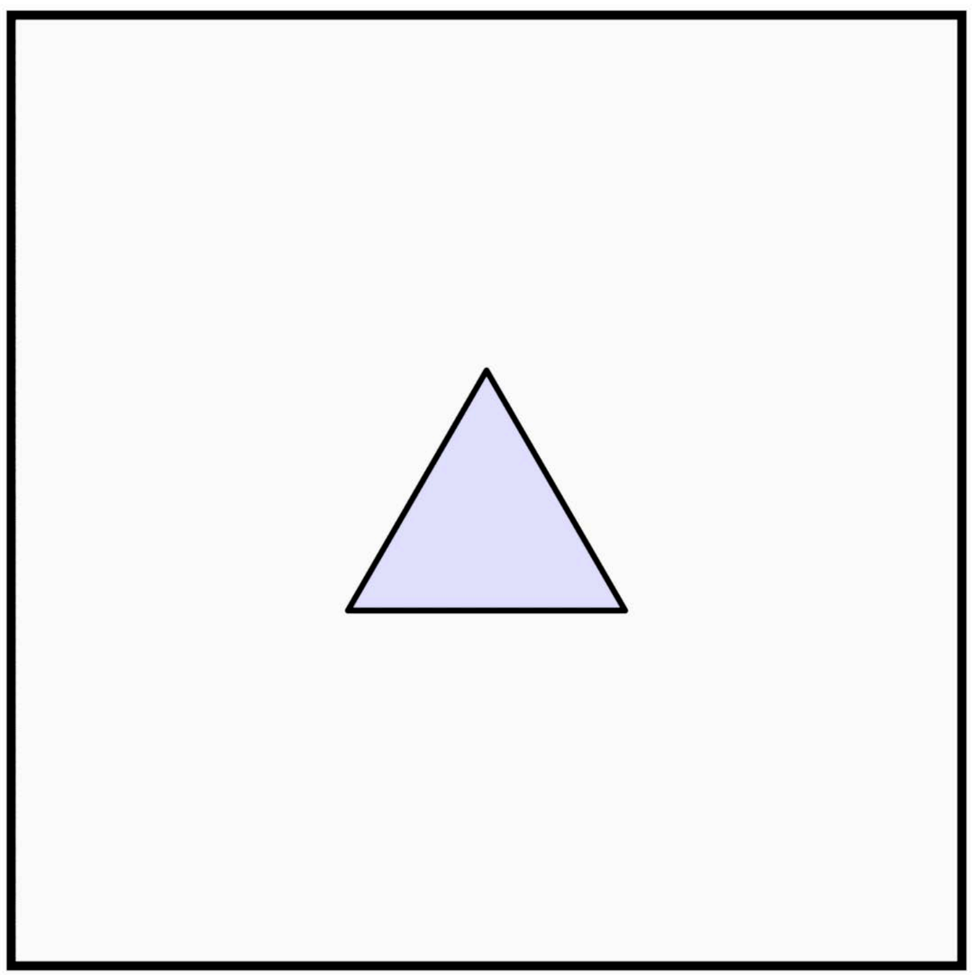
- More explicitly, consider a point $\mathbf{p} = (x, y)$, and the plane $z = 1$ in 3D
- Any three numbers $\hat{\mathbf{p}} = (a, b, c)$ such that $(a/c, b/c) = (x, y)$ are <u>homogeneous coordinates</u> for $\mathbf{p}$
  - E.g., $(x, y, 1)$
  - In general: (cx, cy, c) for $c \neq 0$
- Hence, two points $\hat{\mathbf{p}}, \hat{\mathbf{q}} \in \mathbb{R}^3 \backslash \{O\}$ describe the same point in 2D (and line in 3D) if $\hat{\mathbf{p}} = \lambda \hat{\mathbf{q}}$ for some $\lambda \neq 0$

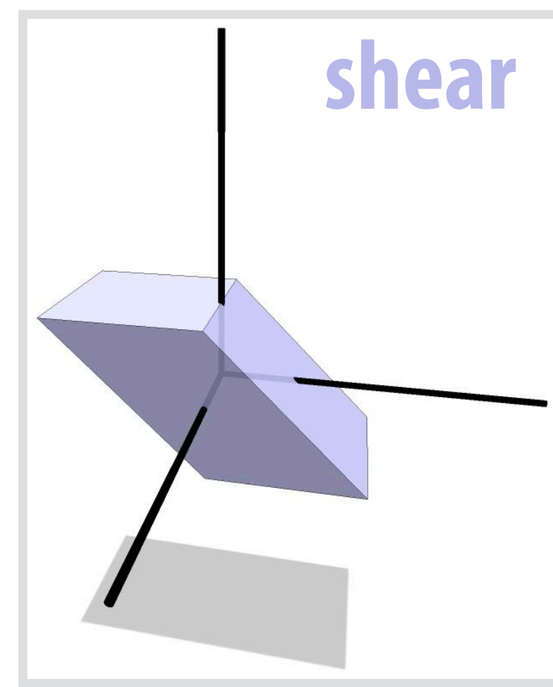
Great… but how does this help us with transformations?

Translation in Homogeneous Coordinates

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

2D coordinates





Q: What kind of transformation does this look like?

Translation in Homogeneous Coordinates

- But wait a minute—shear is a <u>linear</u> 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 <u>affine</u> transformation in 2D as a <u>linear</u> 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

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

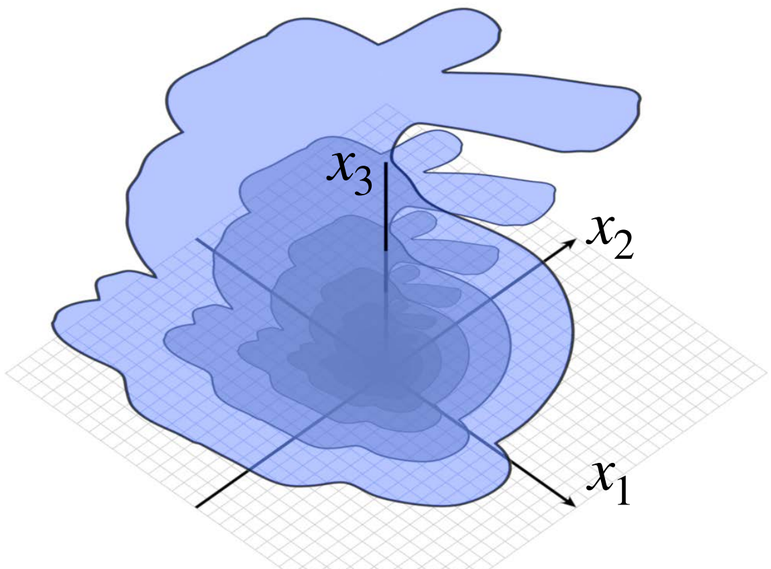
- In matrix form:

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

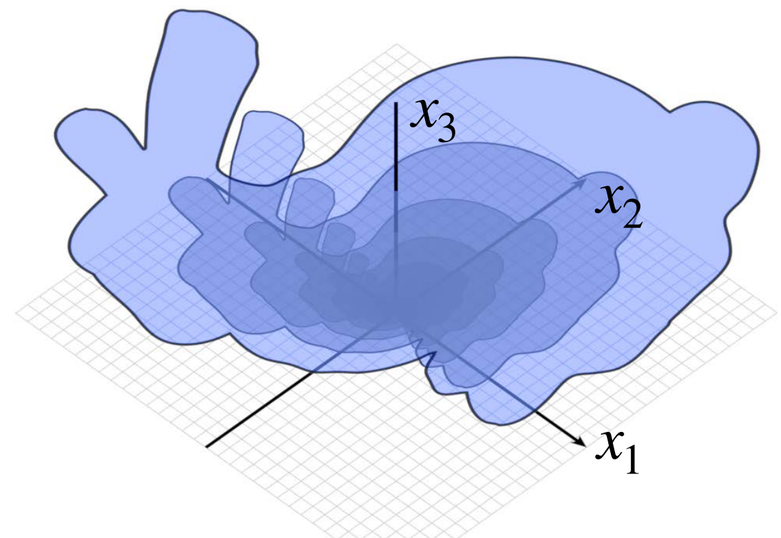
- In our case, $\mathbf{v} = (0,0,1)$ and so we get a matrix

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

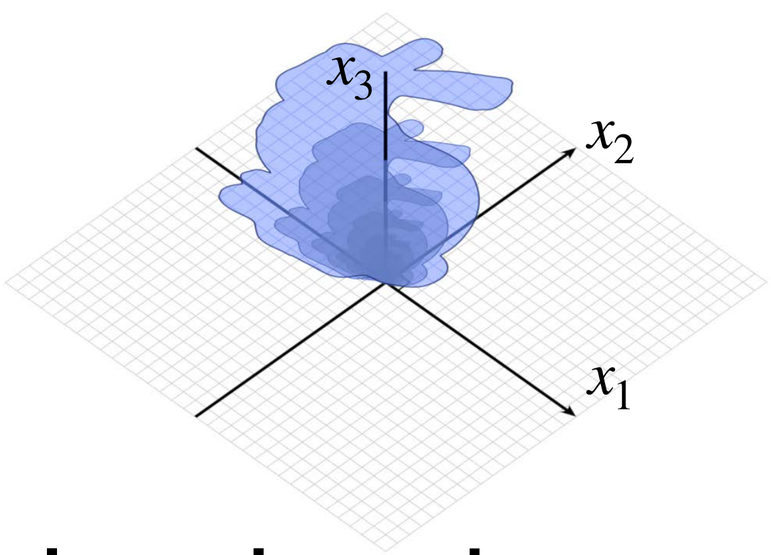
Other 2D Transformations in Homogeneous Coordinates



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

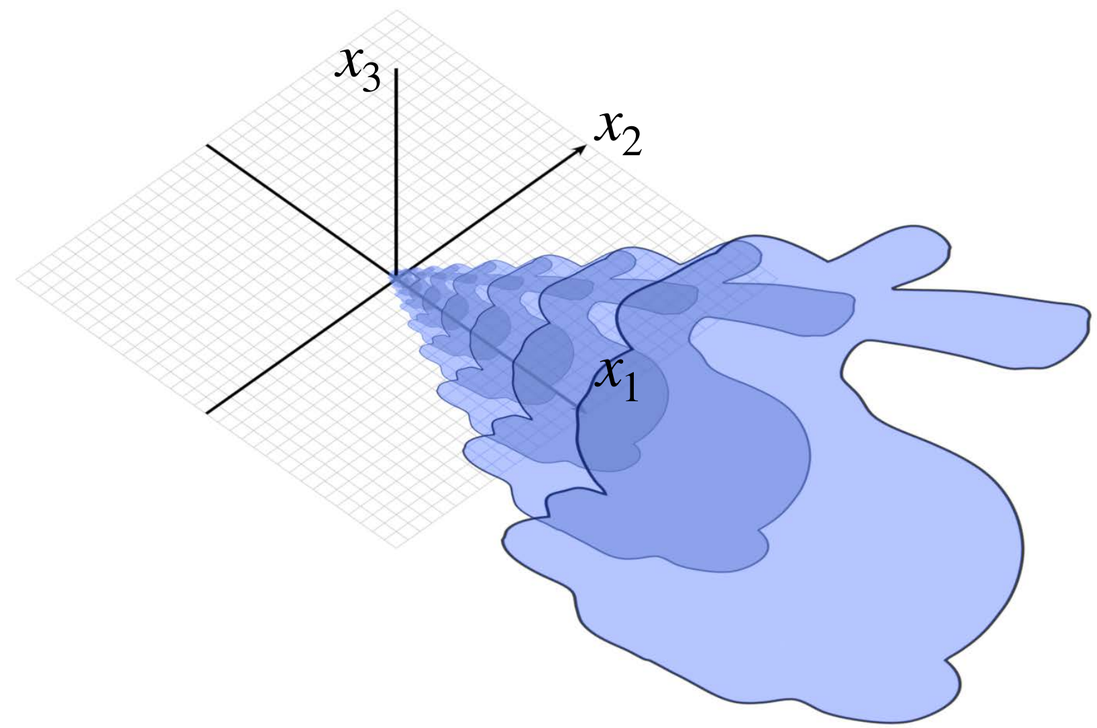


2D rotation ↔ rotate around x_3



2D scale ↔ scale x_1 and x_2; preserve x_3

(Q: what happens to 2D shape if you scale x_1, x_2, <u>and</u> x_3 uniformly?)



2D translate ↔ shear

Now easy to compose <u>all</u> 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 θ

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

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

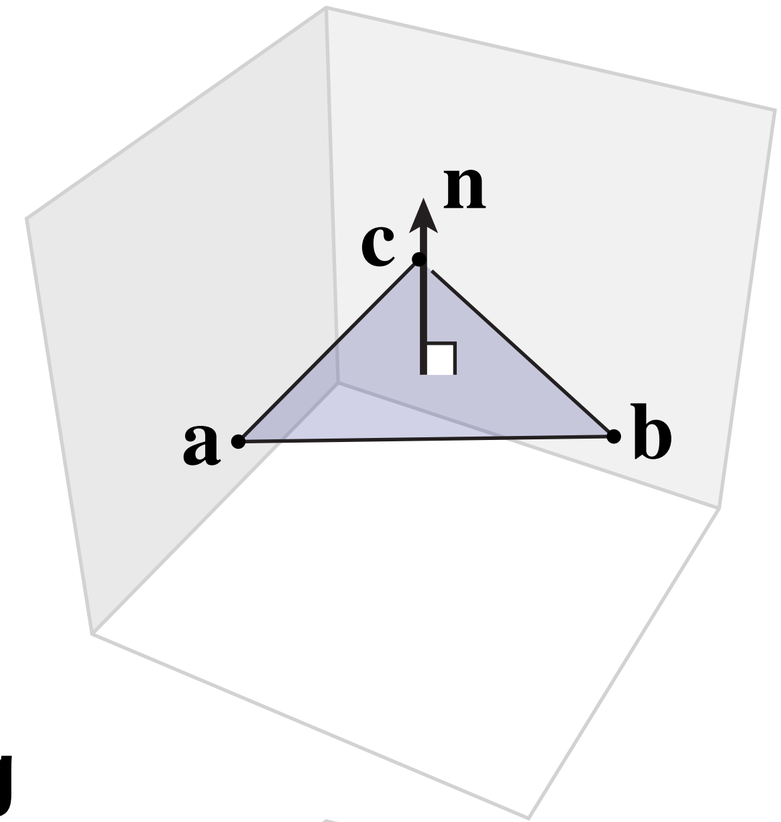
$$\begin{bmatrix} a & 0 & 0 & 0 \\ 0 & b & 0 & 0 \\ 0 & 0 & c & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

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

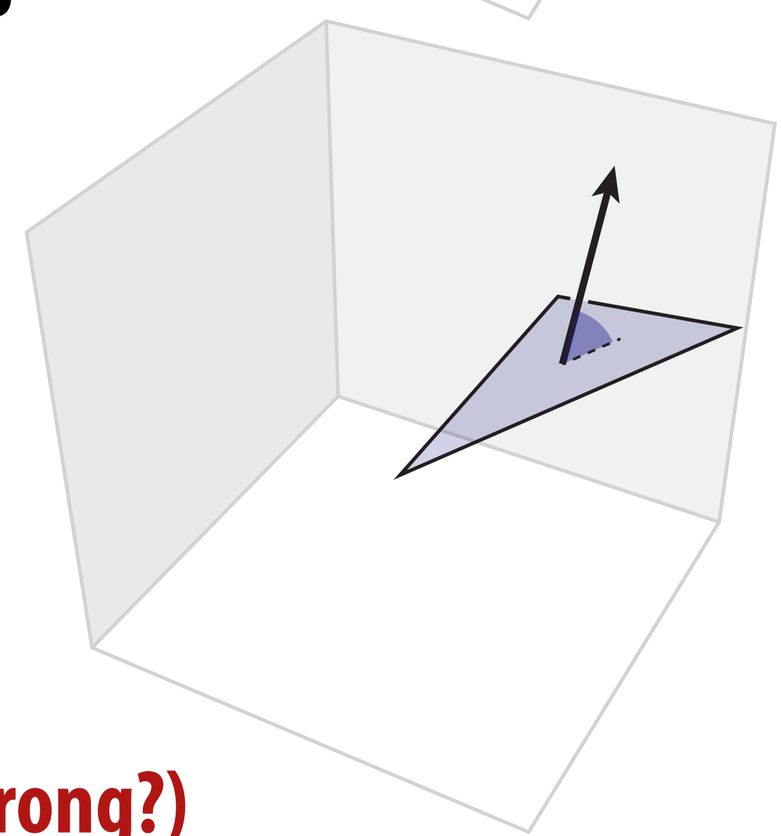
$$\begin{bmatrix} 1 & 0 & 0 & u \\ 0 & 1 & 0 & v \\ 0 & 0 & 1 & w \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Points vs. Vectors

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



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



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 θ

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

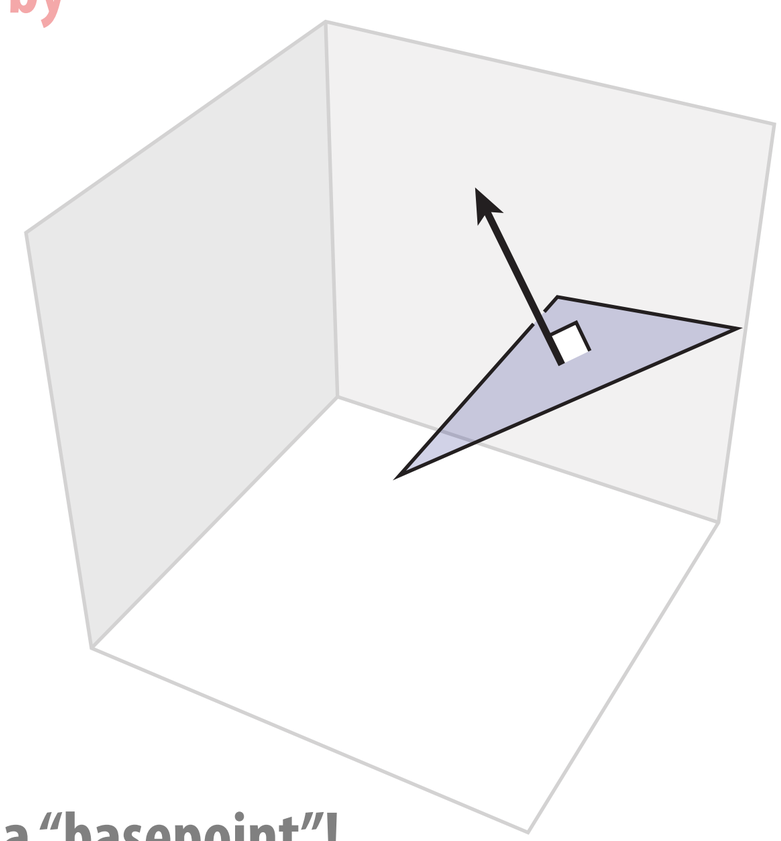
translate normal by (u, v, w)

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

- **Solution? Just set homogeneous coordinate to zero!**

- **Translation now gets ignored; normal is orthogonal to triangle**

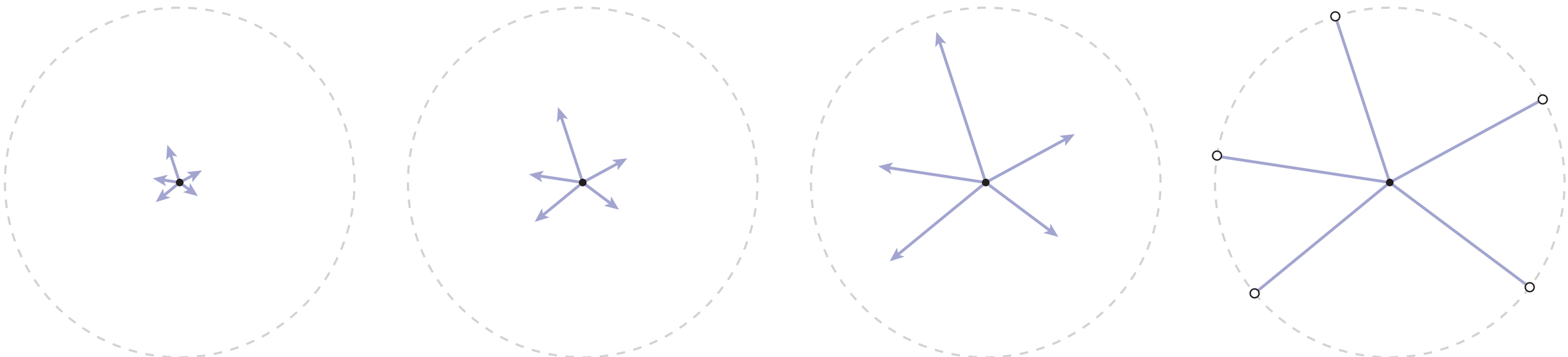
$$\begin{bmatrix} n_1 \\ n_2 \\ n_3 \\ 0 \end{bmatrix}$$



*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 <u>nonzero</u> homogeneous coordinate** $(c = 1)$
  - A ***vector*** **has a <u>zero</u> homogeneous coordinate** $(c = 0)$
- **But wait… what division by c mean when it's equal to zero?**
- **Well consider what happens as $c \to 0$…**



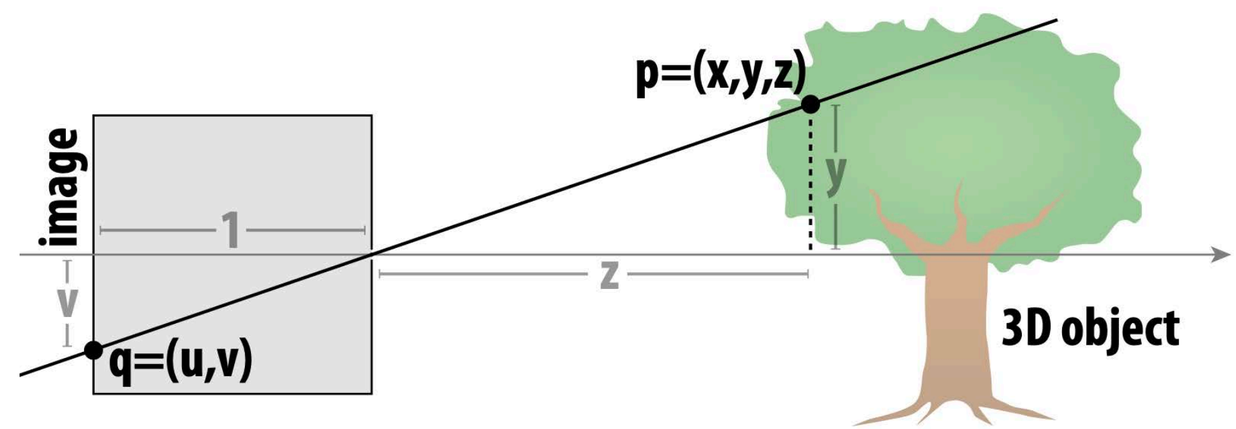
$(x, y)/1$ $(x, y)/0.5$ $(x, y)/0.25$ $(x, y)/0.001$

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 $z = 1$



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

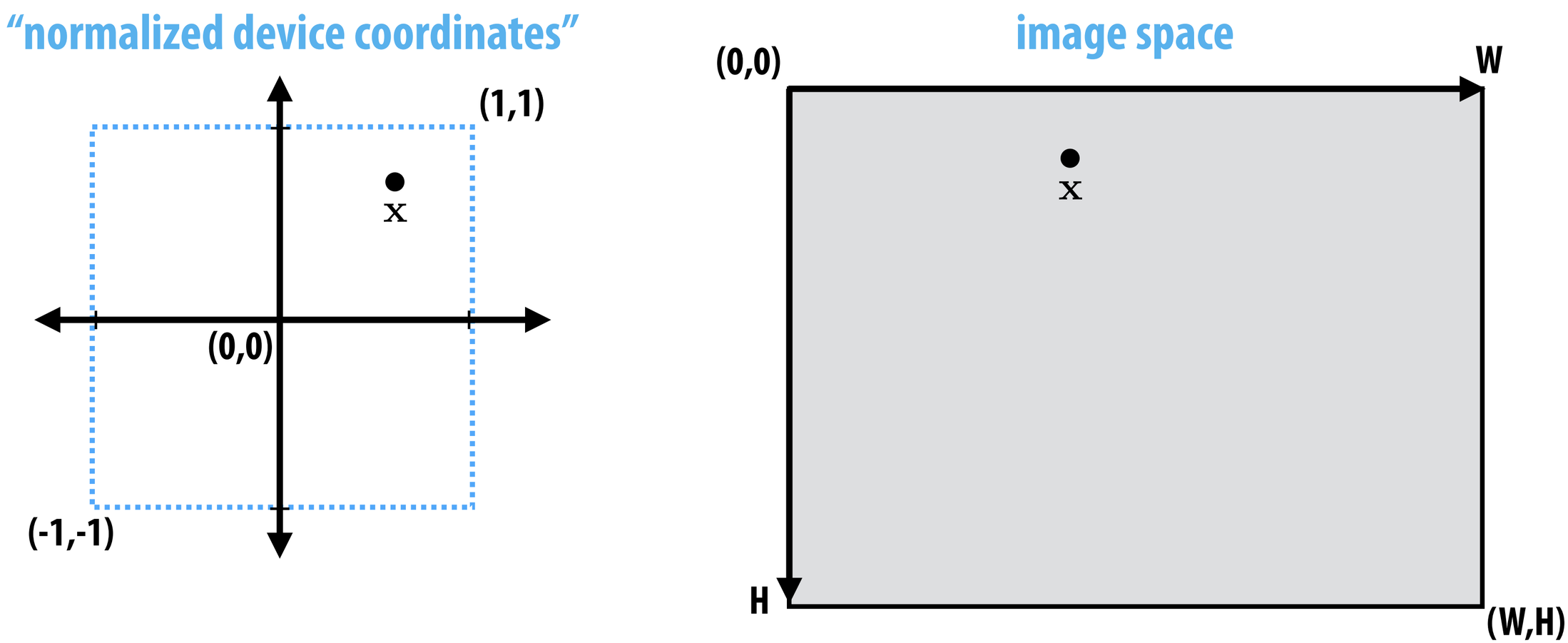
$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \\ z \end{bmatrix}$$

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

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

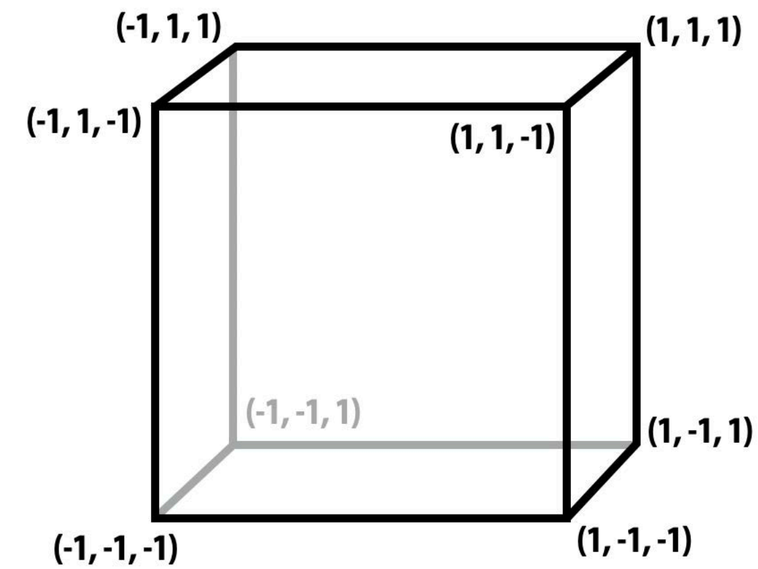
Screen Transformation

- 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 [0,1] x [0,1] on the z = 1 plane, into a W x H pixel image

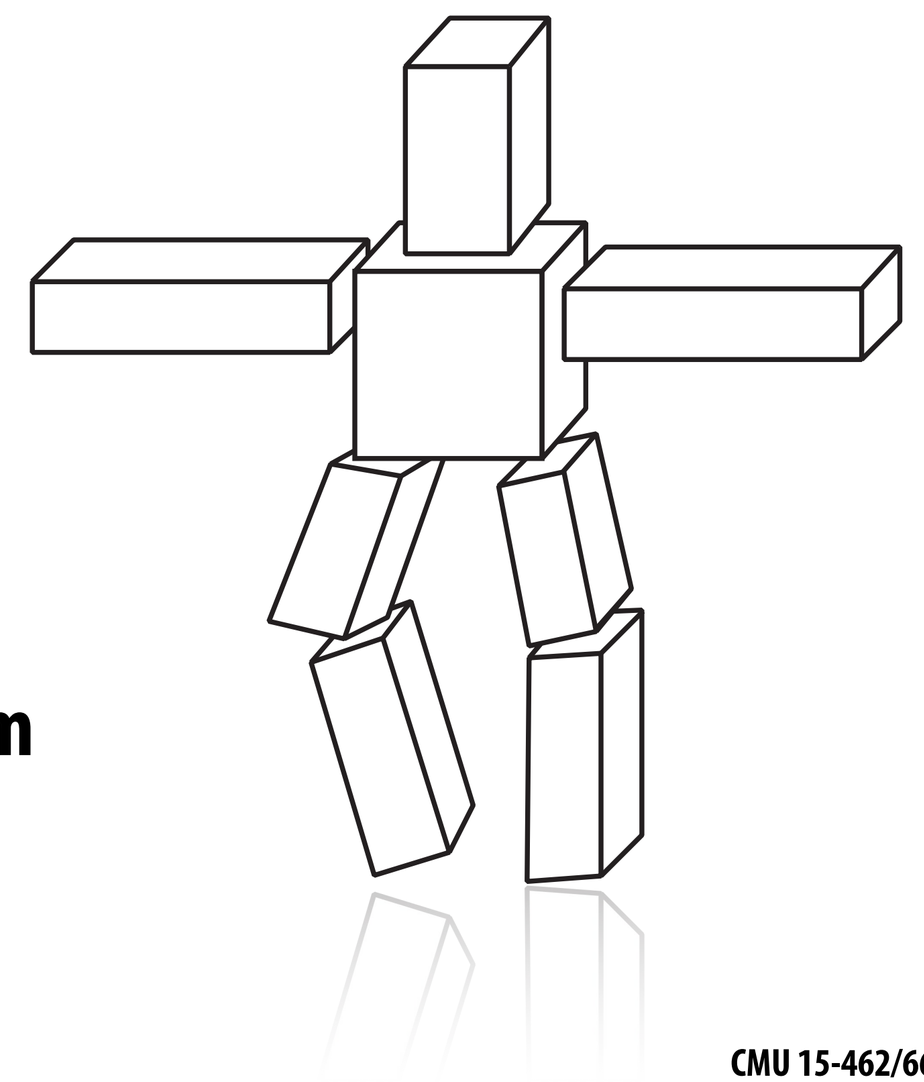


Q: What transformation(s) would you apply? (Careful: y is now *down!*)

Scene Graph

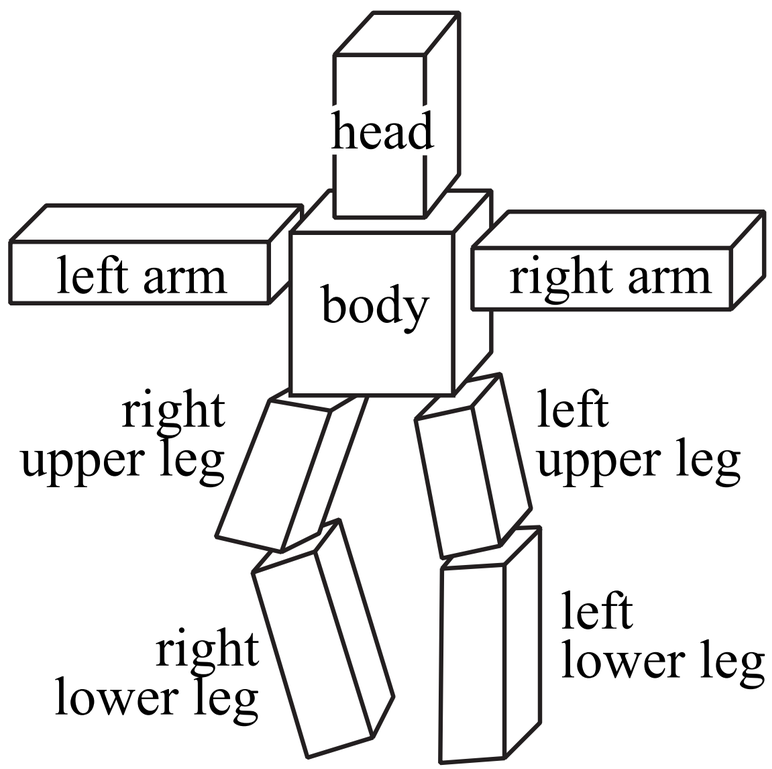


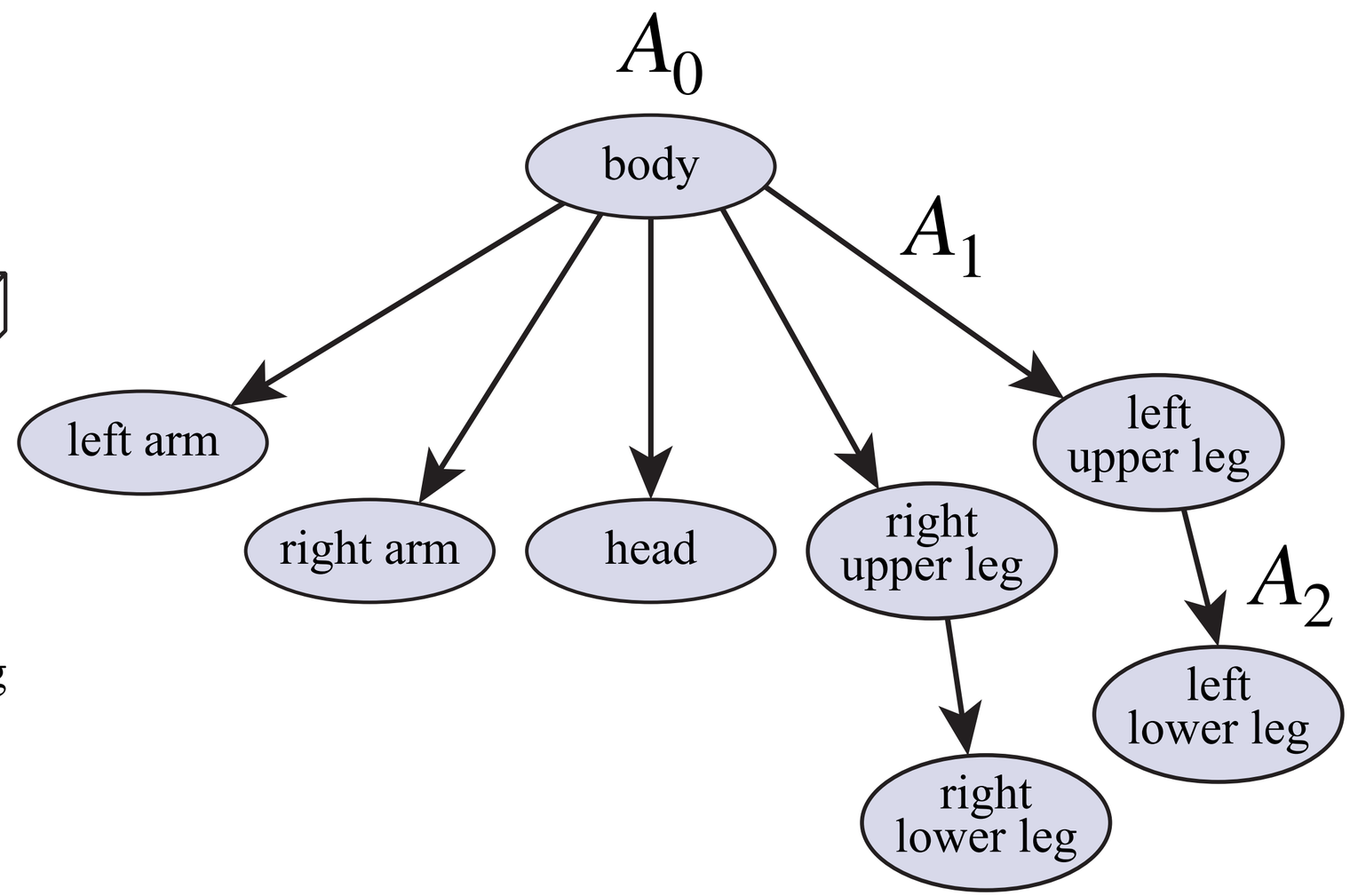
- For complex scenes (e.g., more than just a cube!) <u>scene graph</u> can help organize transformations
- Motivation: suppose we want to build a "cube creature" by transforming copies of the unit cube
- Difficult to specify each transformation directly
- Instead, build up transformations of "lower"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

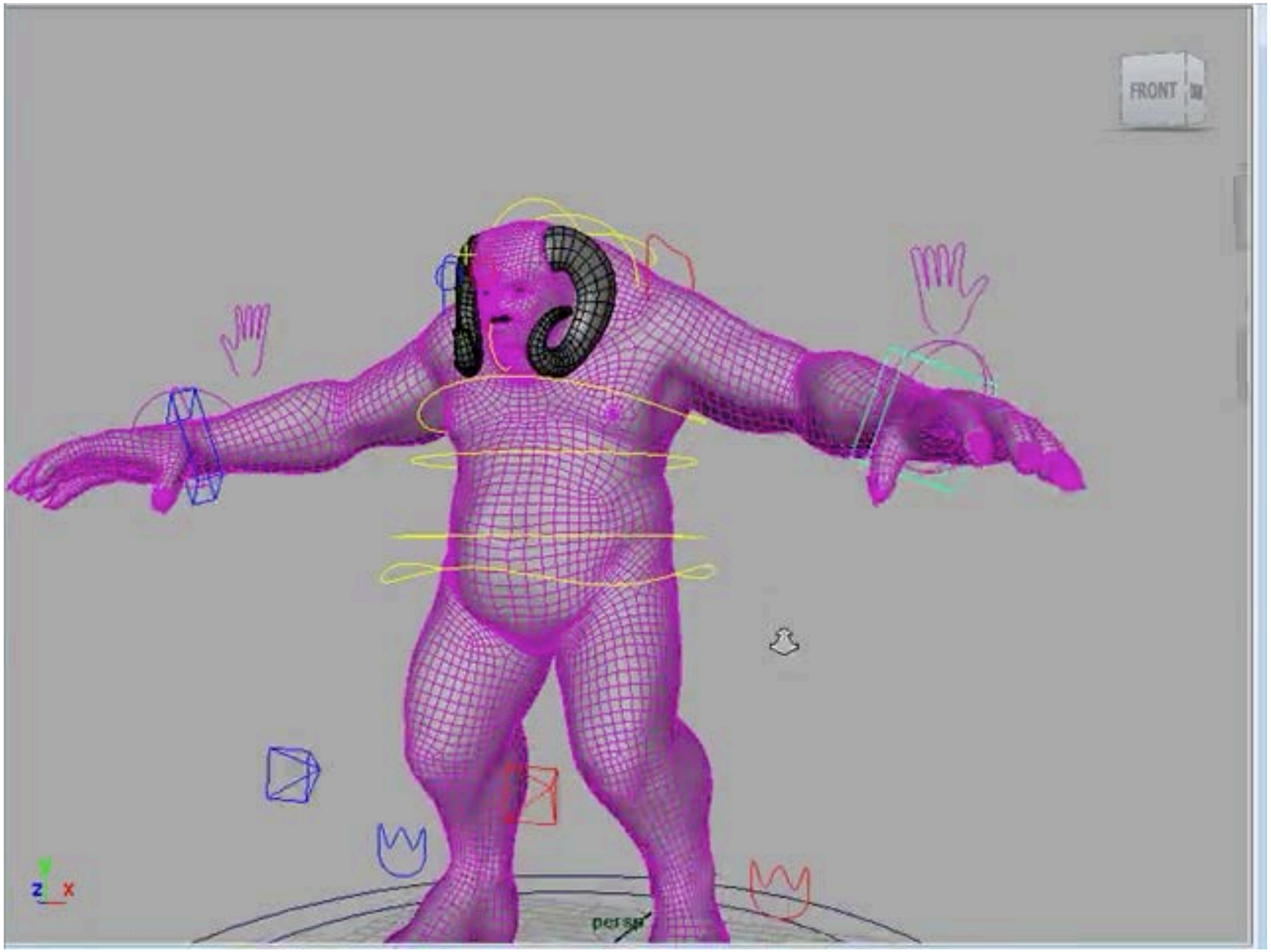




- E.g., $A_1 A_0$ gets applied to left upper leg; $A_2 A_1 A_0$ to left lower leg
- Keep transformations on a stack to reduce redundant multiplication

Scene Graph—Example

Often used to build up complex “rig”:

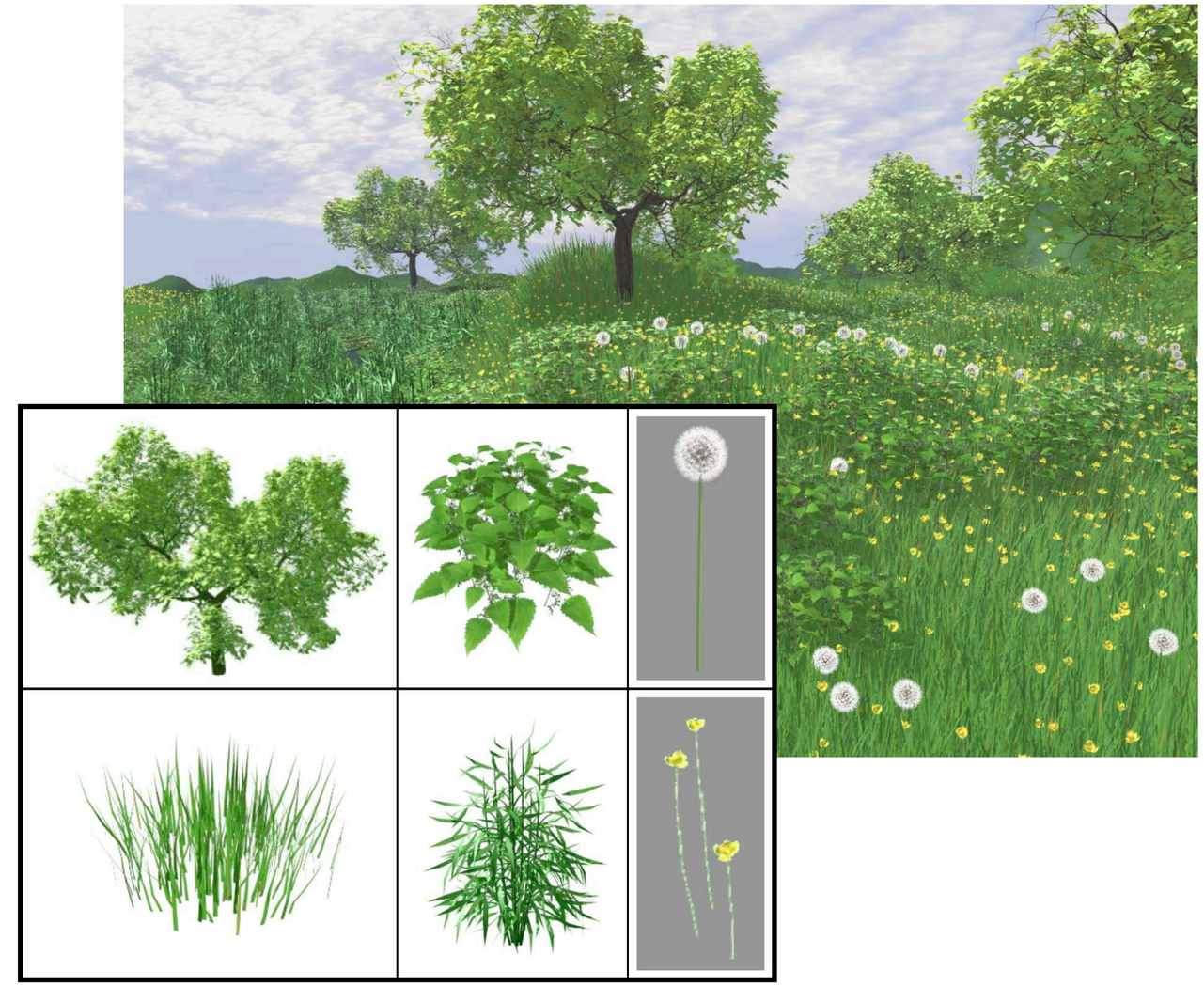


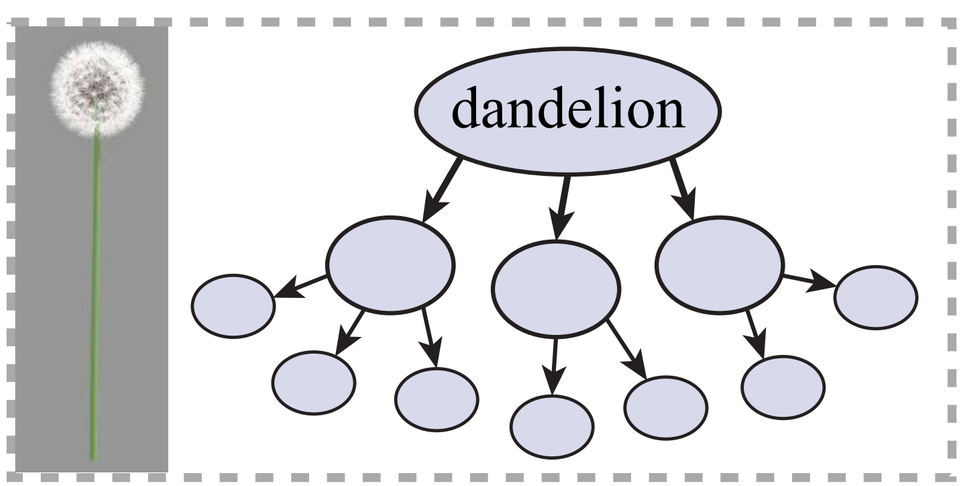
In general, scene graph also includes other models, lights, cameras, …

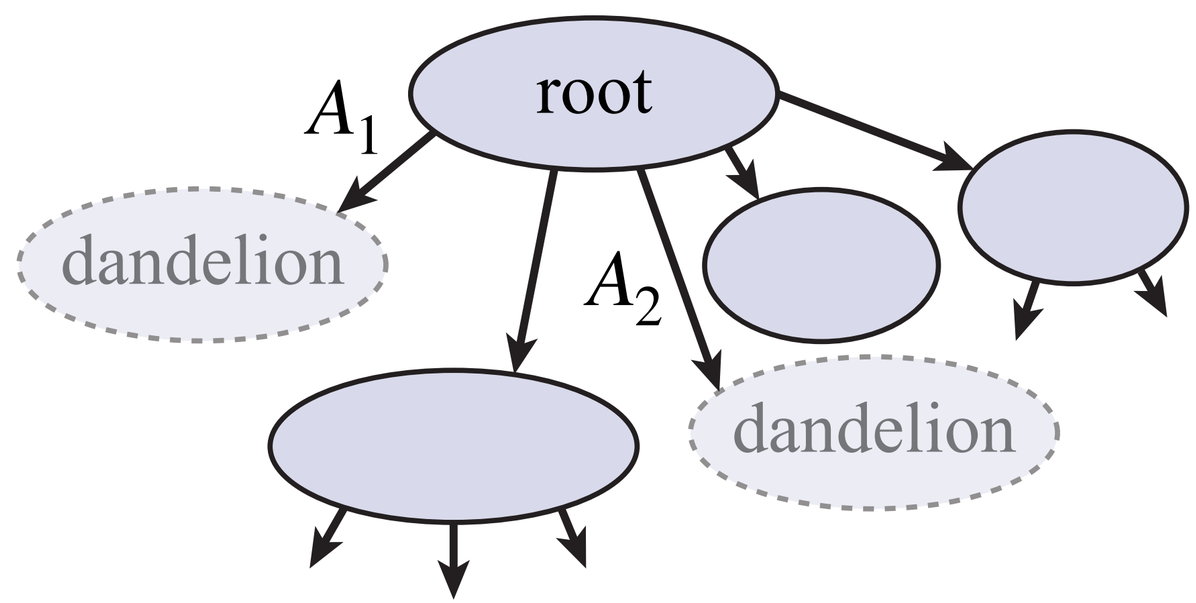
Instancing

- 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

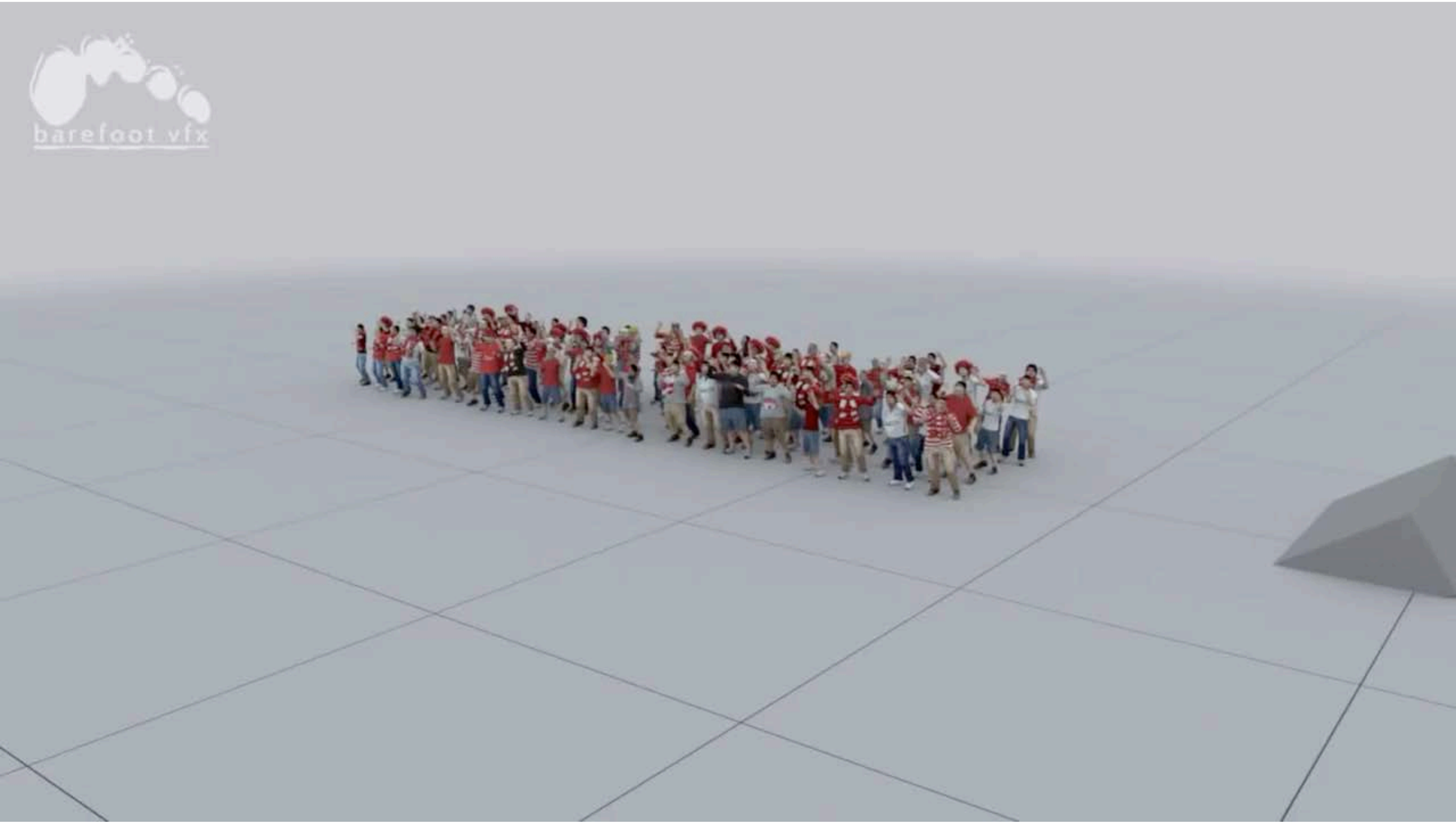
Deussen et al, "Realistic modeling and rendering of plant ecosystems"



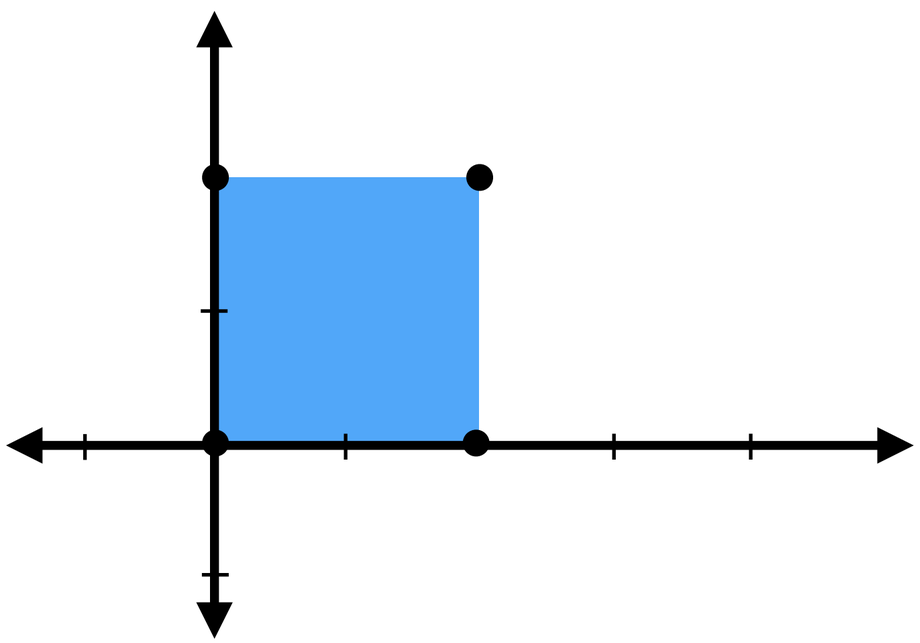




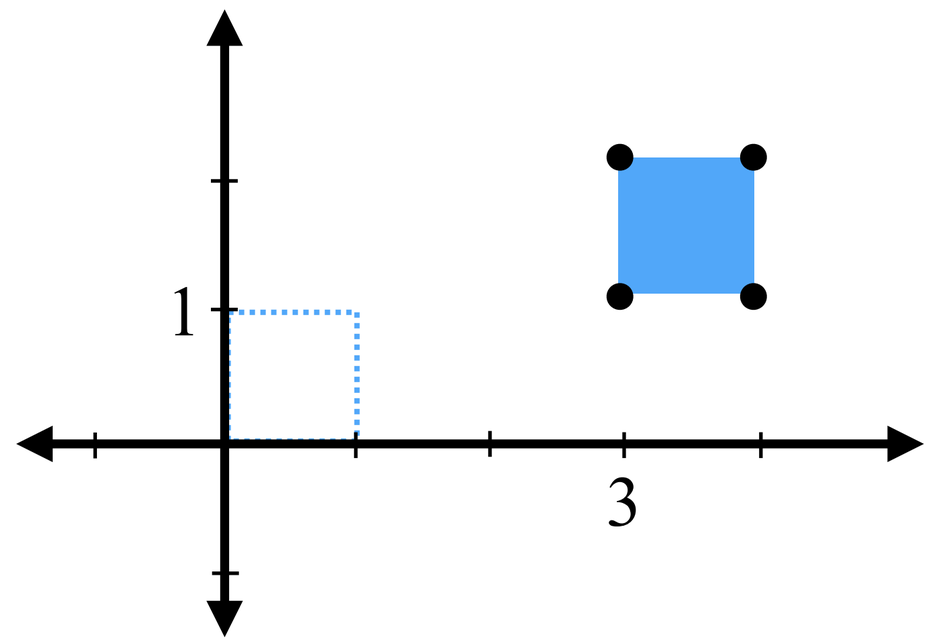
Instancing—Example



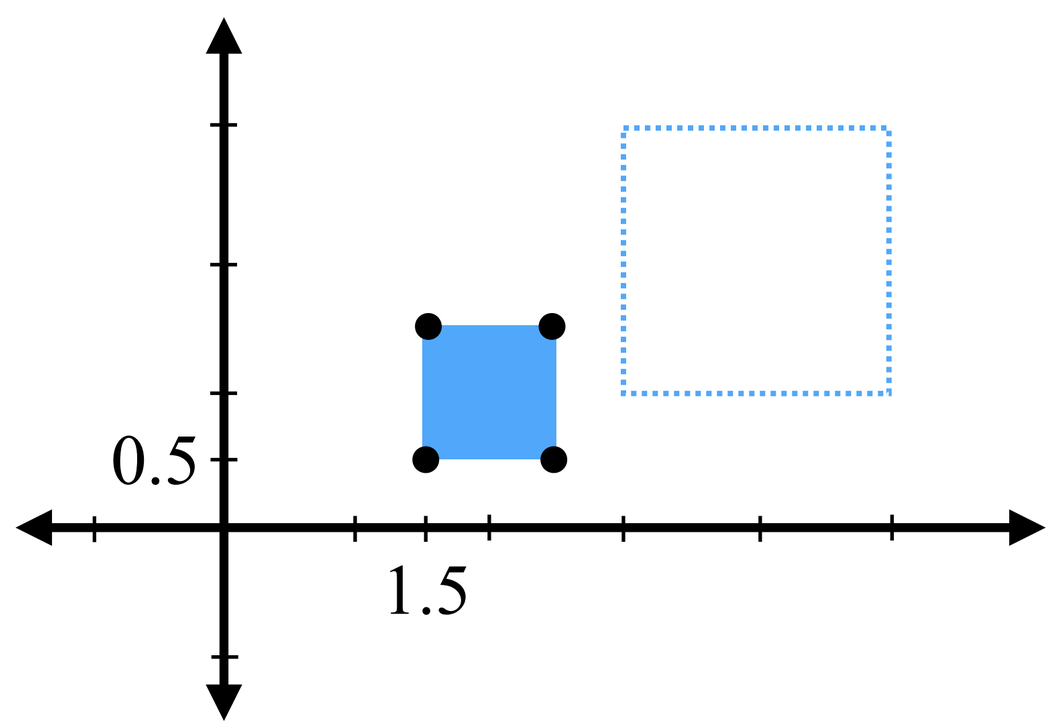
Order matters when composing transformations!



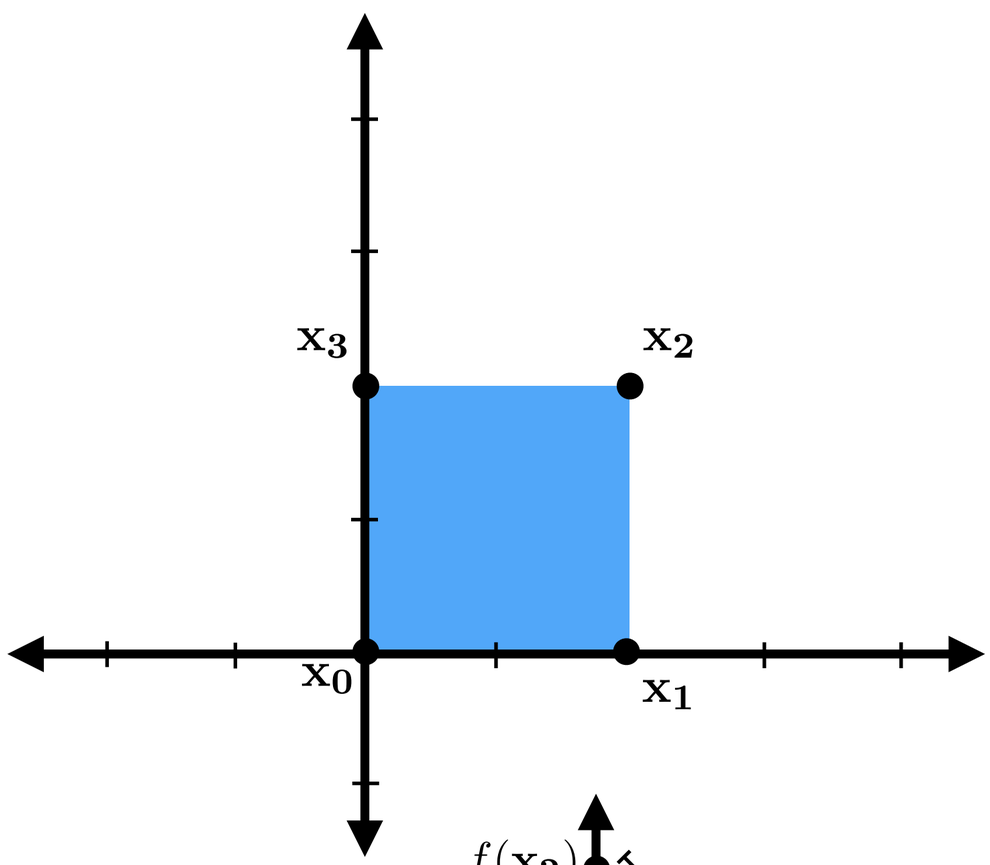
scale by 1/2, then translate by (3,1)

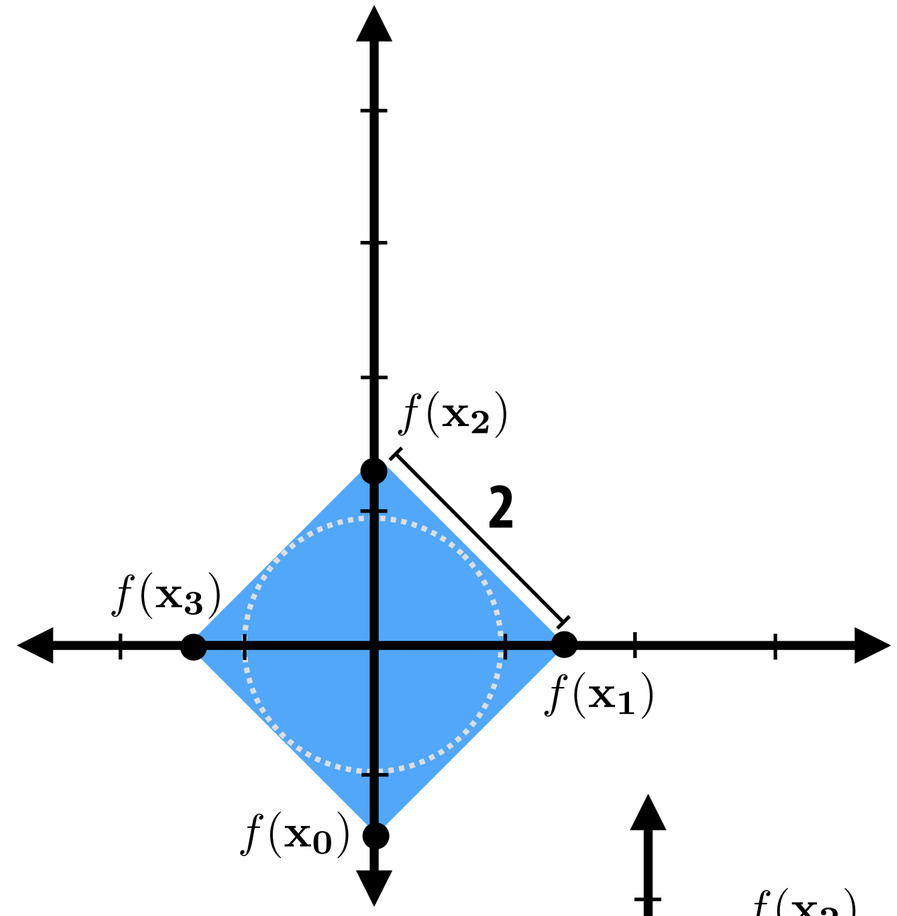


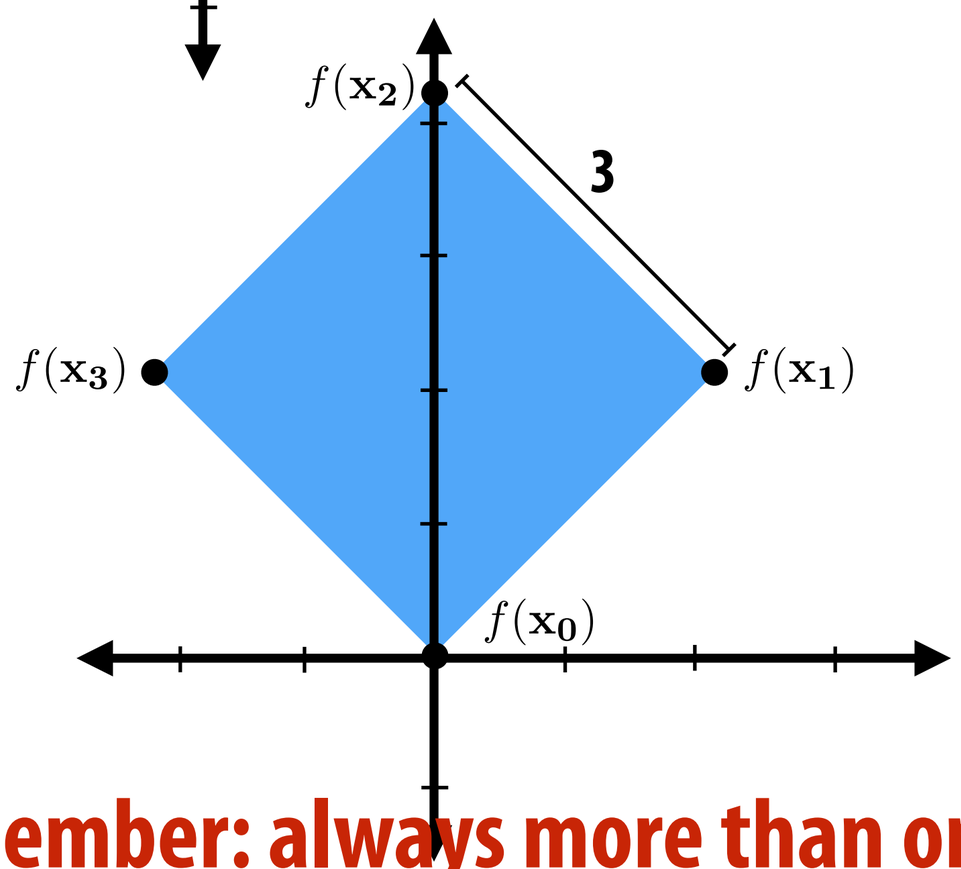
translate by (3,1), then scale by 1/2

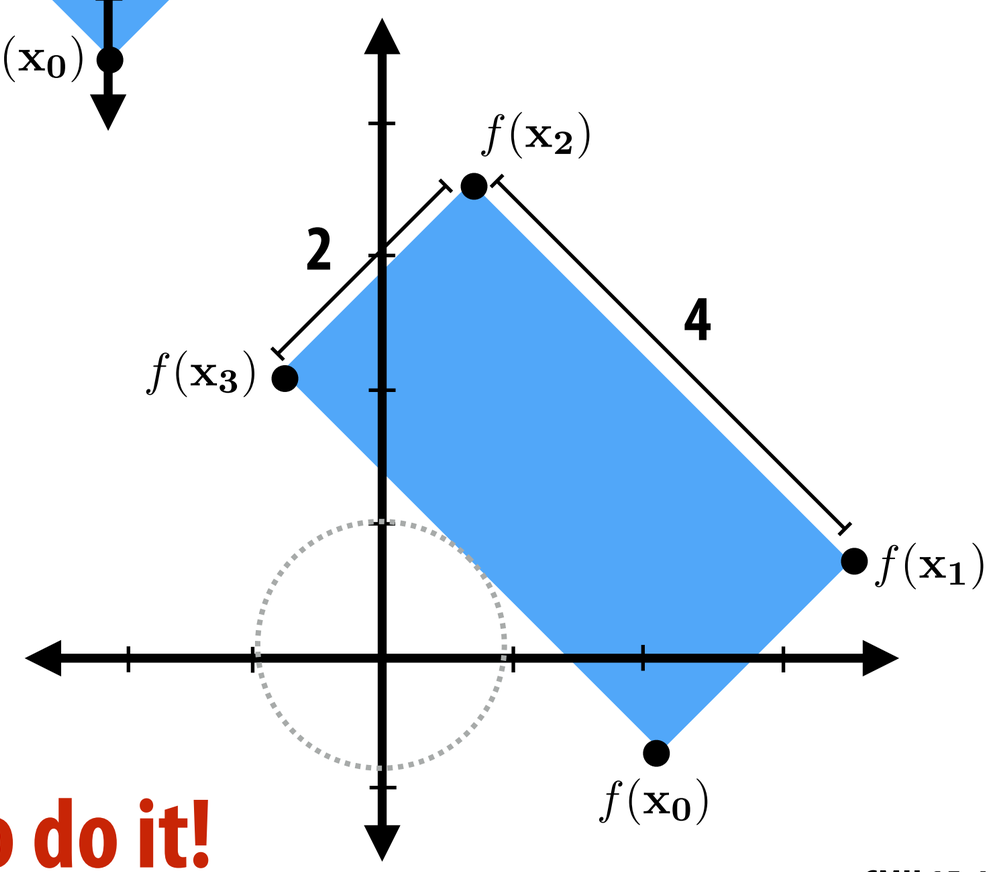


How would you perform these transformations?



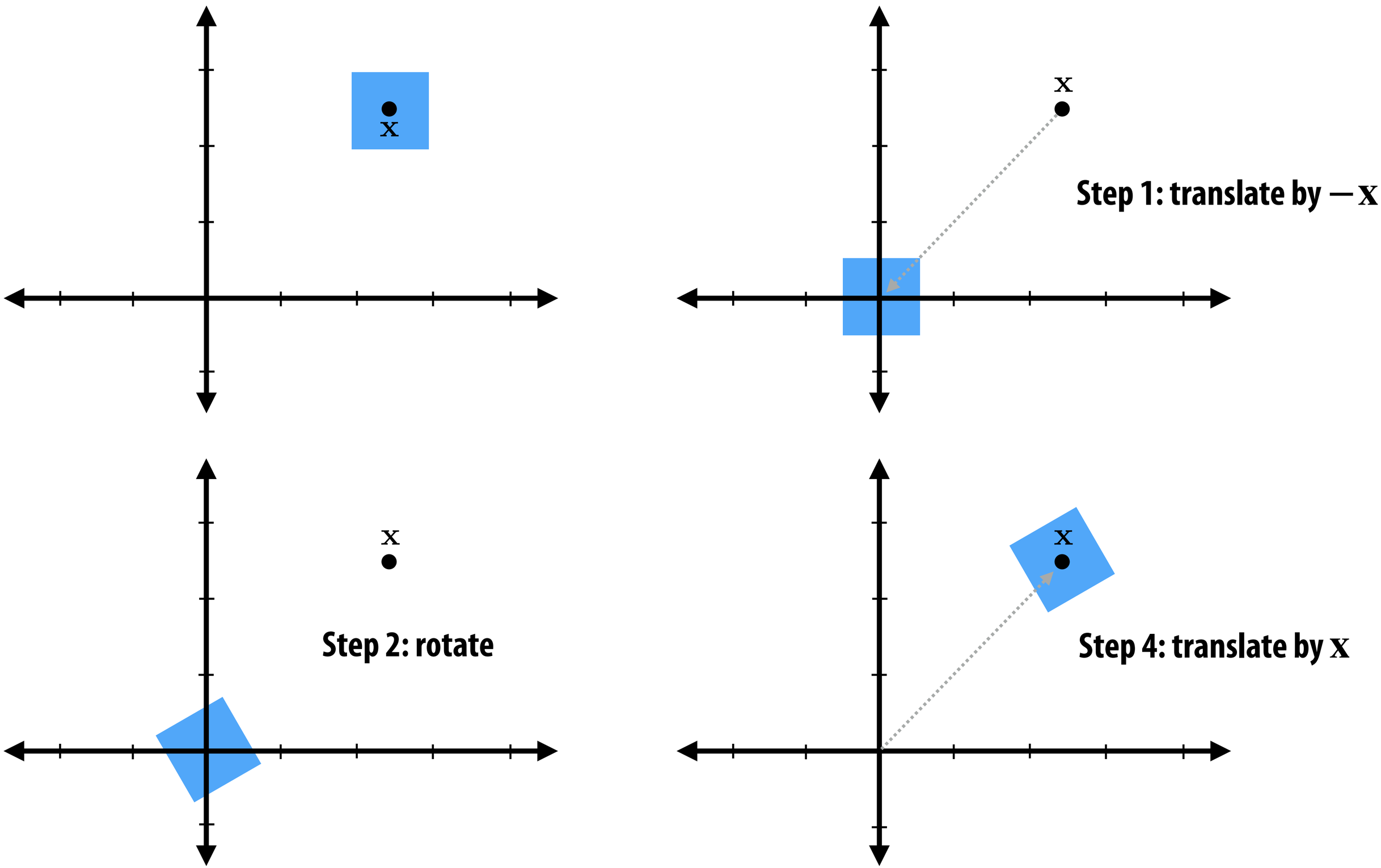






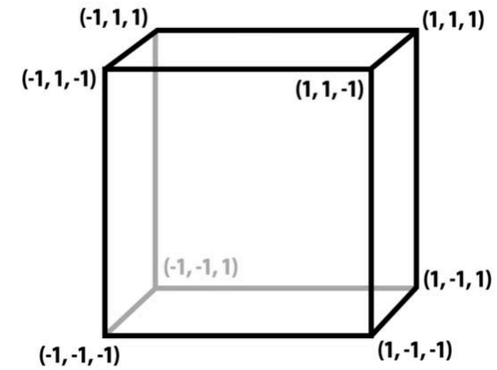
Remember: always more than one way to do it!

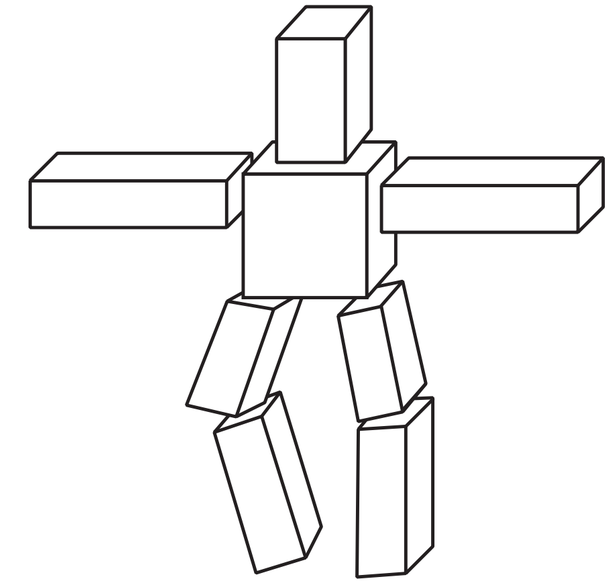
Common task: rotate about a point x



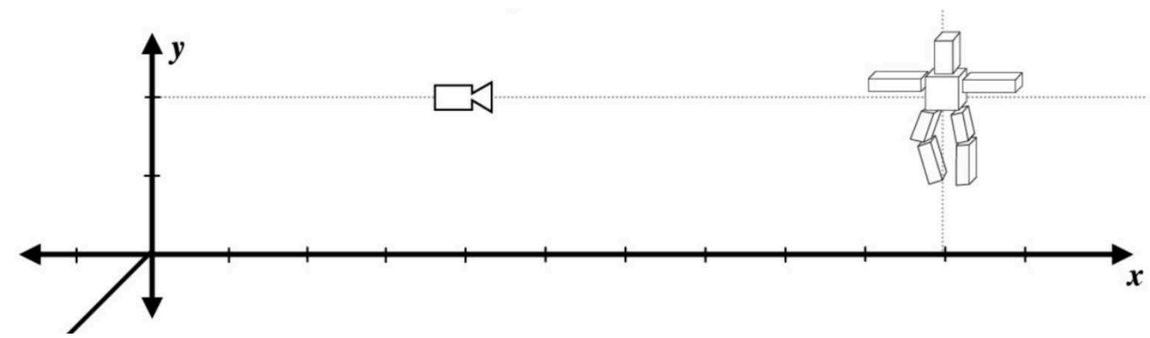
Q: What happens if we just rotate without translating first?

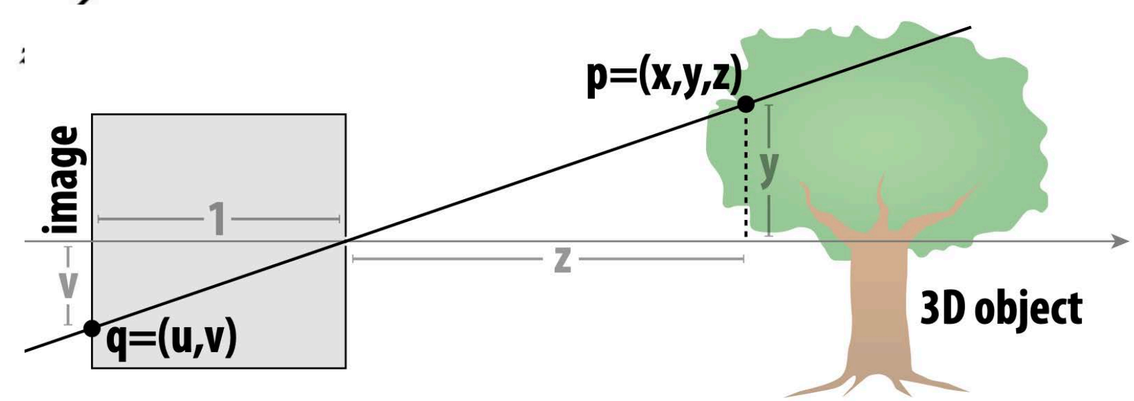
Drawing a Cube Creature

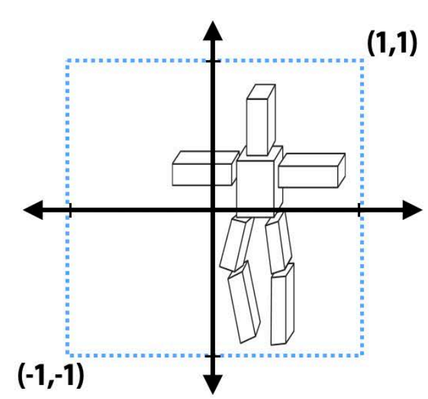


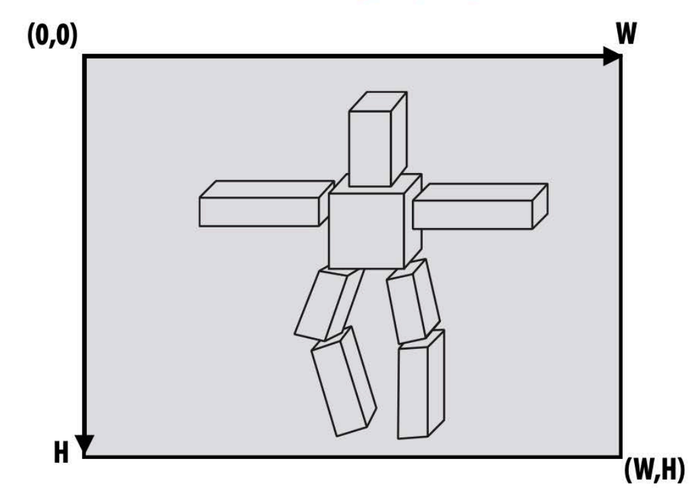


- Let's put this all together: starting with our 3D cube, we want to make a 2D, perspective-correct image of a "cube creature"
- First we use our scene graph to apply 3D transformations to several copies of our cube
- Then we apply a 3D transformation to position our camera
- Then a perspective projection
- 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

basic nonlinear transformations

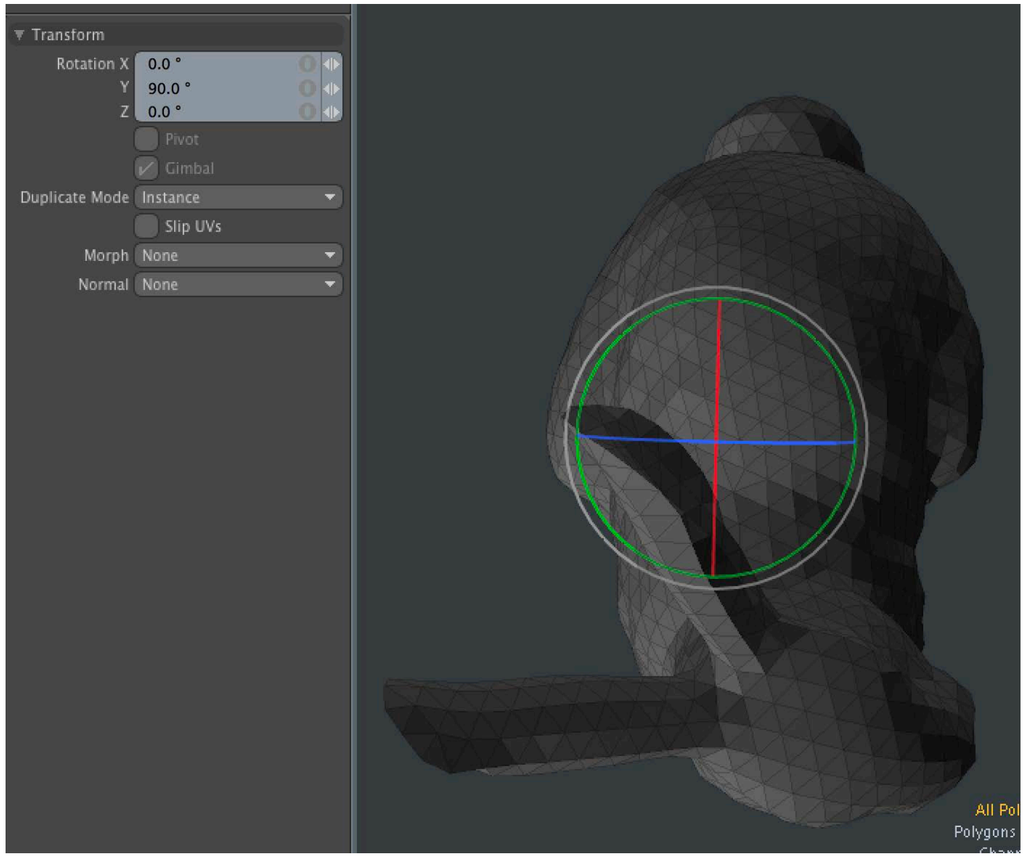
translation
perspective projection

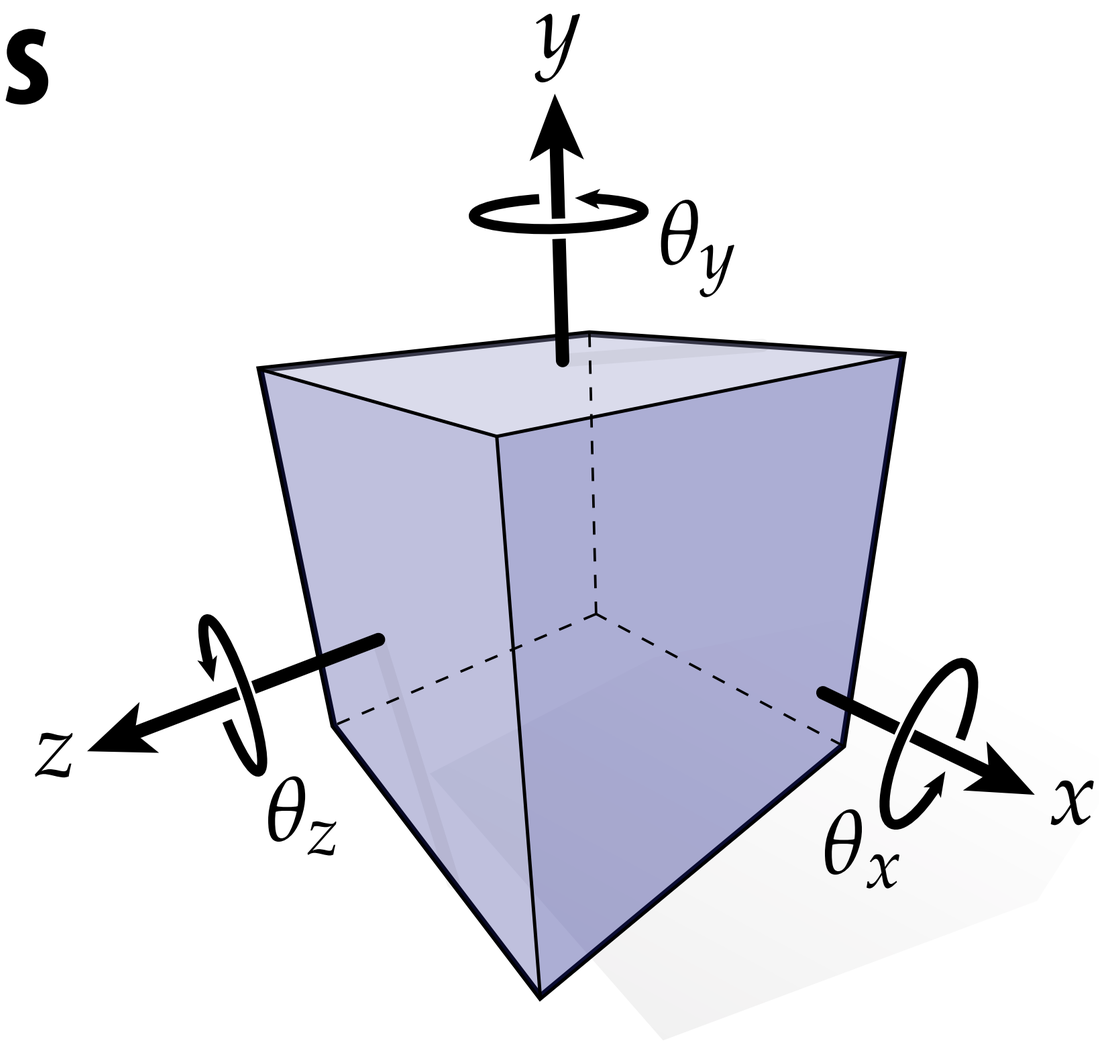
linear when represented via homogeneous coords
homogeneous coords also distinguish points & vectors

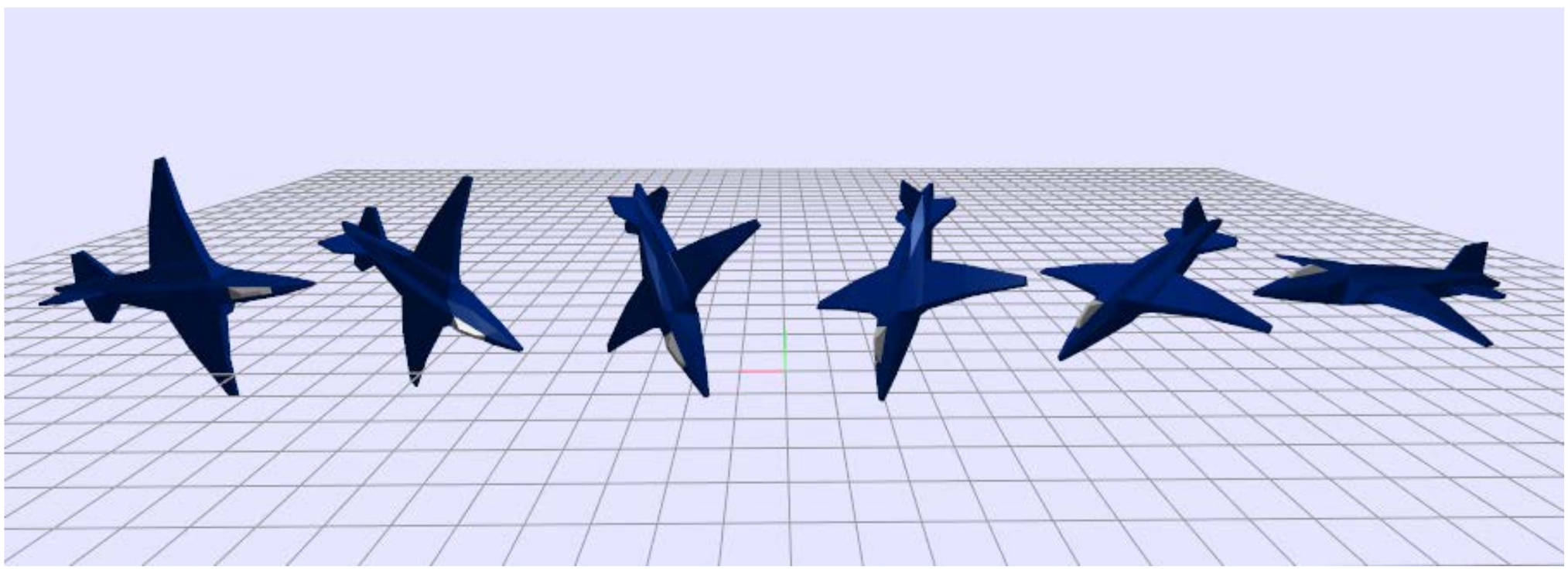
composite transformations

- 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

Next time: 3D Rotations





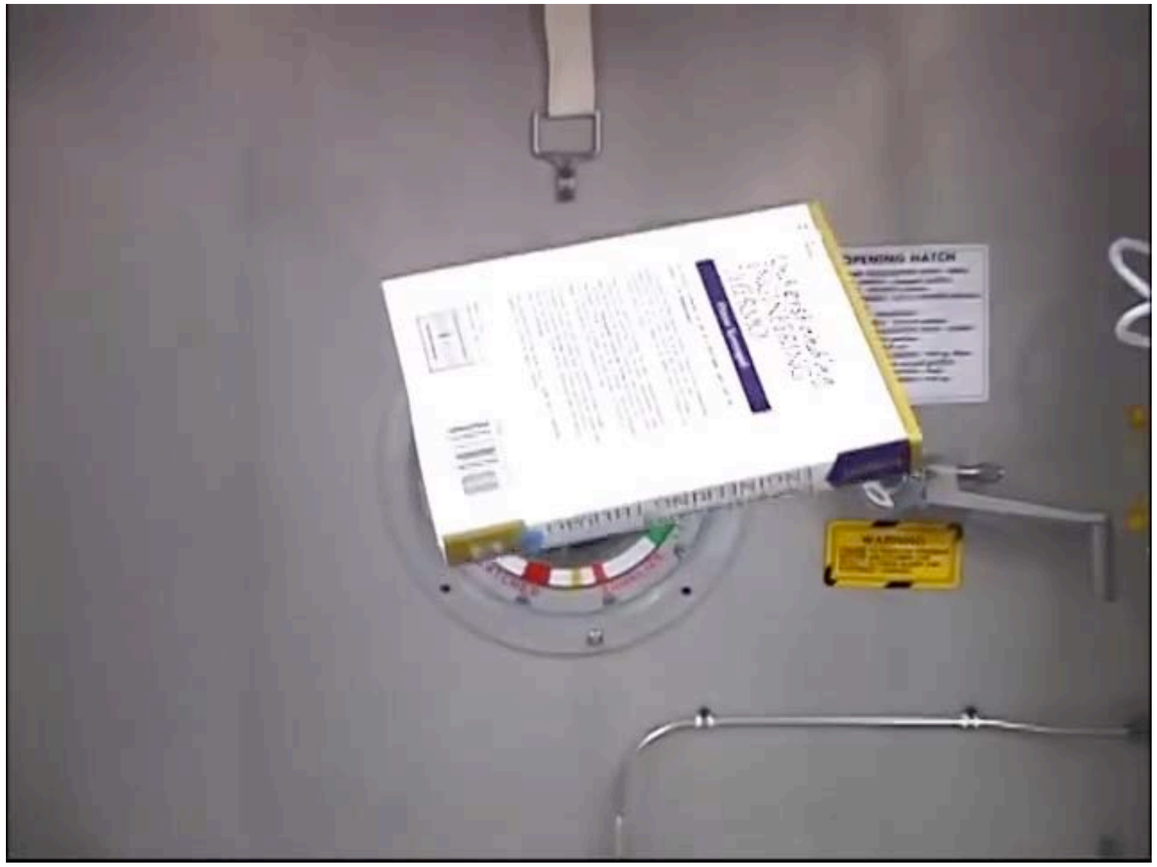


3D Rotations and Complex Representations

Computer Graphics
CMU 15-462/15-662

Rotations in 3D

- What is a rotation, intuitively?
- *How do you know a rotation when you see it?*
  - length/distance is preserved (no stretching/shearing)
  - orientation is preserved (e.g., text remains readable)
  - origin is preserved (otherwise it's a rotation + translation)



3D Rotations—Degrees of Freedom

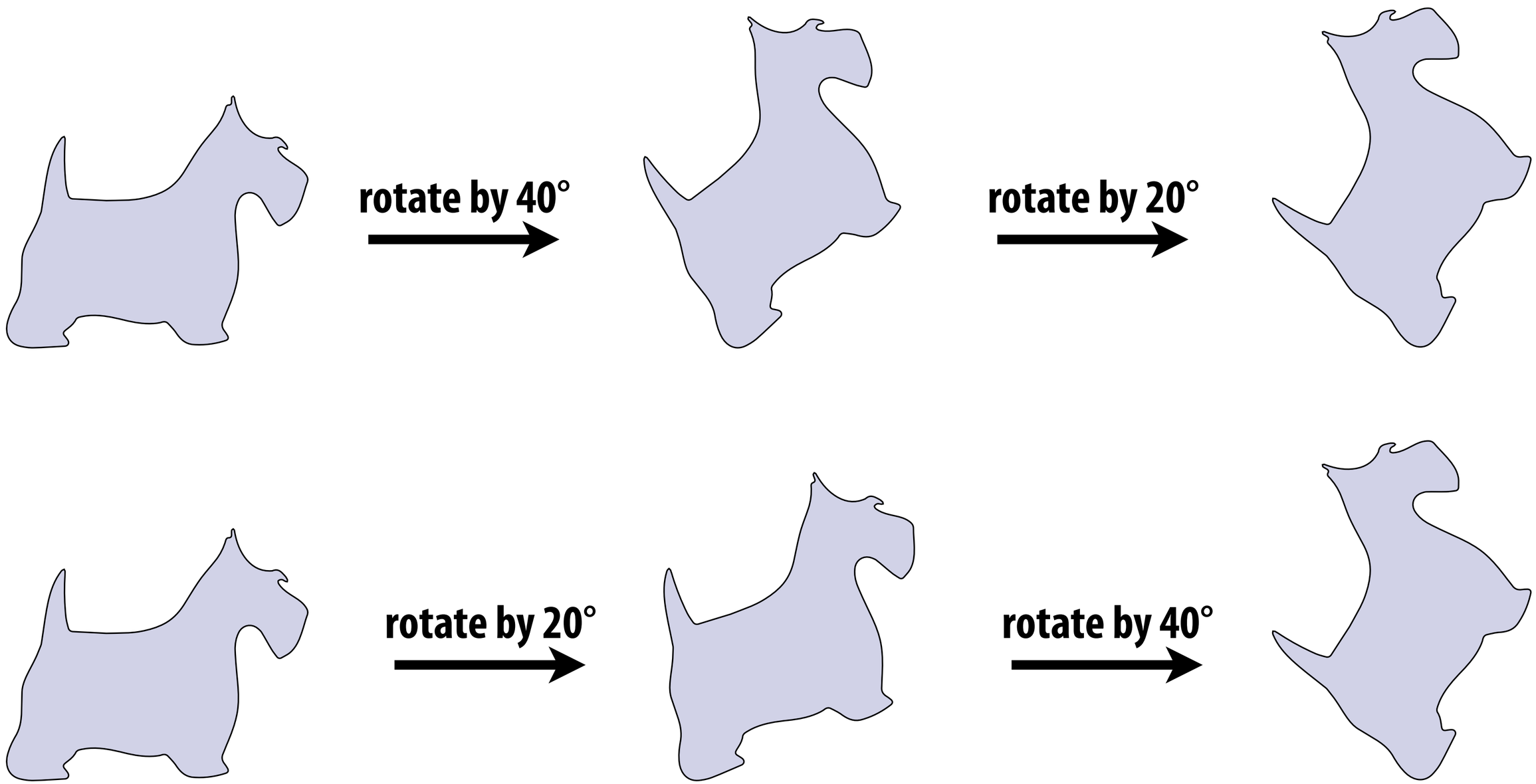
- How many numbers do we need to specify a rotation in 3D?
- For instance, we could use rotations around X, Y, Z. But do we *need* all three?
- Well, to rotate Pittsburgh to another city (say, São Paulo), we have to specify two numbers: latitude & longitude:
- Do we really need both latitude and longitude? Or will one suffice?
- Is that the *only* rotation from Pittsburgh to São Paulo? (How many more numbers do we need?)

NO: We can keep São Paulo fixed as we rotate the globe.

Hence, we MUST have three degrees of freedom.

Commutativity of Rotations—2D

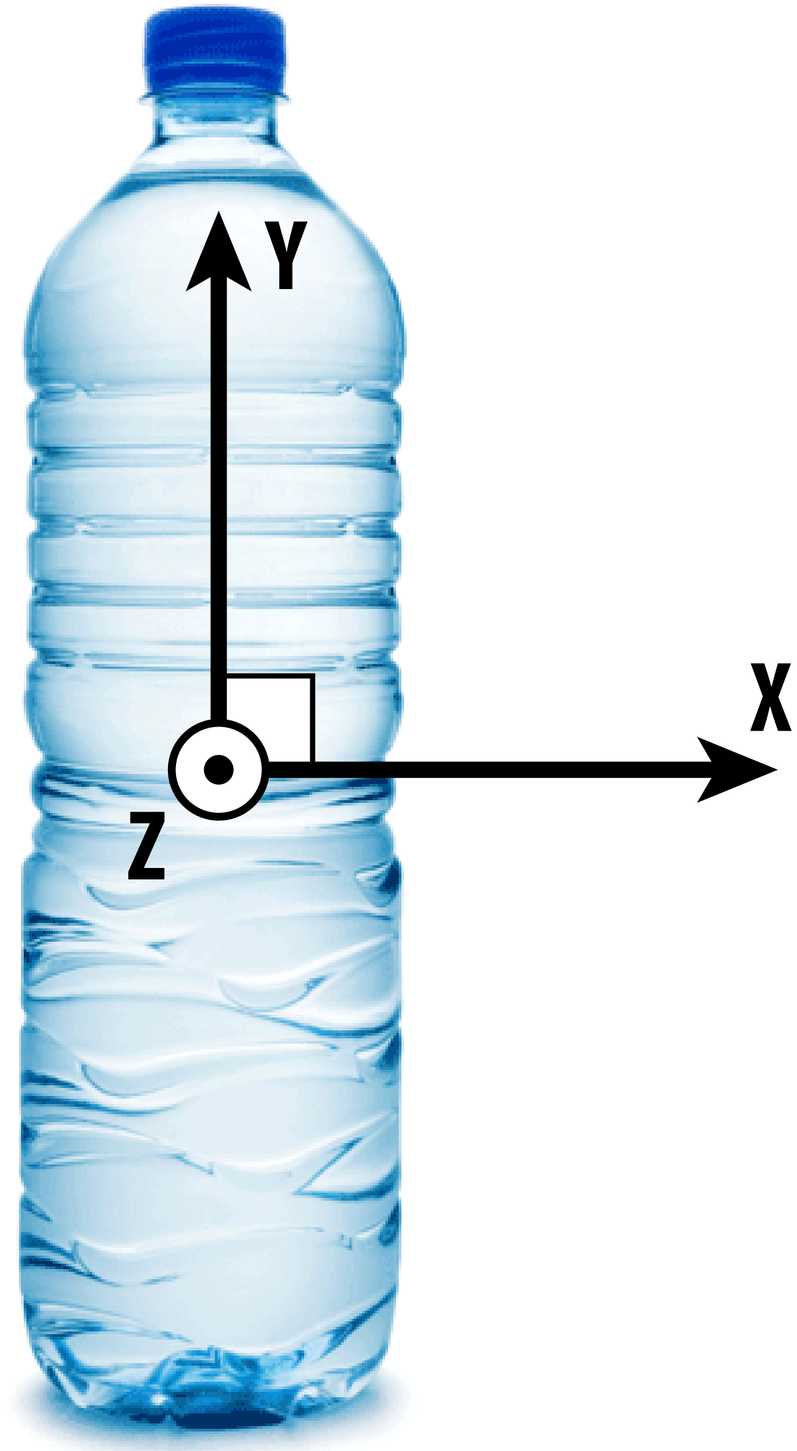
- In 2D, order of rotations doesn't matter:



Same result! ("2D rotations commute")

Commutativity of Rotations—3D

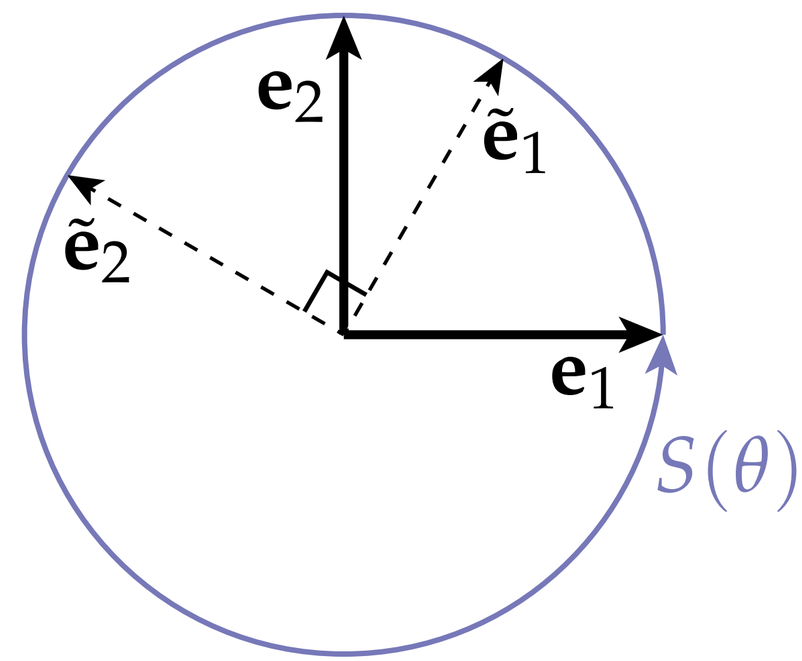
- What about in 3D?
- Try it at home—grab a water bottle!
  - Rotate 90° around Y, then 90° around Z, then 90° around X
  - Rotate 90° around Z, then 90° around Y, then 90° around X
  - (Was there any difference?)



CONCLUSION: bad things can happen if we're not careful about the order in which we apply rotations!

Representing Rotations—2D

- **First things first: how do we get a rotation matrix in 2D? (Don't just regurgitate the formula!)**
- **Suppose I have a function S(θ) that for a given angle θ gives me the point (x,y) around a circle (CCW).**
  - **Right now, I *do not care how this function is expressed!****
- **What's e1 rotated by θ?** $\tilde{\mathbf{e}}_1 = S(\theta)$
- **What's e2 rotated by θ?** $\tilde{\mathbf{e}}_2 = S(\theta + \pi/2)$
- **How about** $\mathbf{u} := a\mathbf{e}_1 + b\mathbf{e}_2$ **?**



$$\mathbf{u} := aS(\theta) + bS(\theta + \pi/2)$$

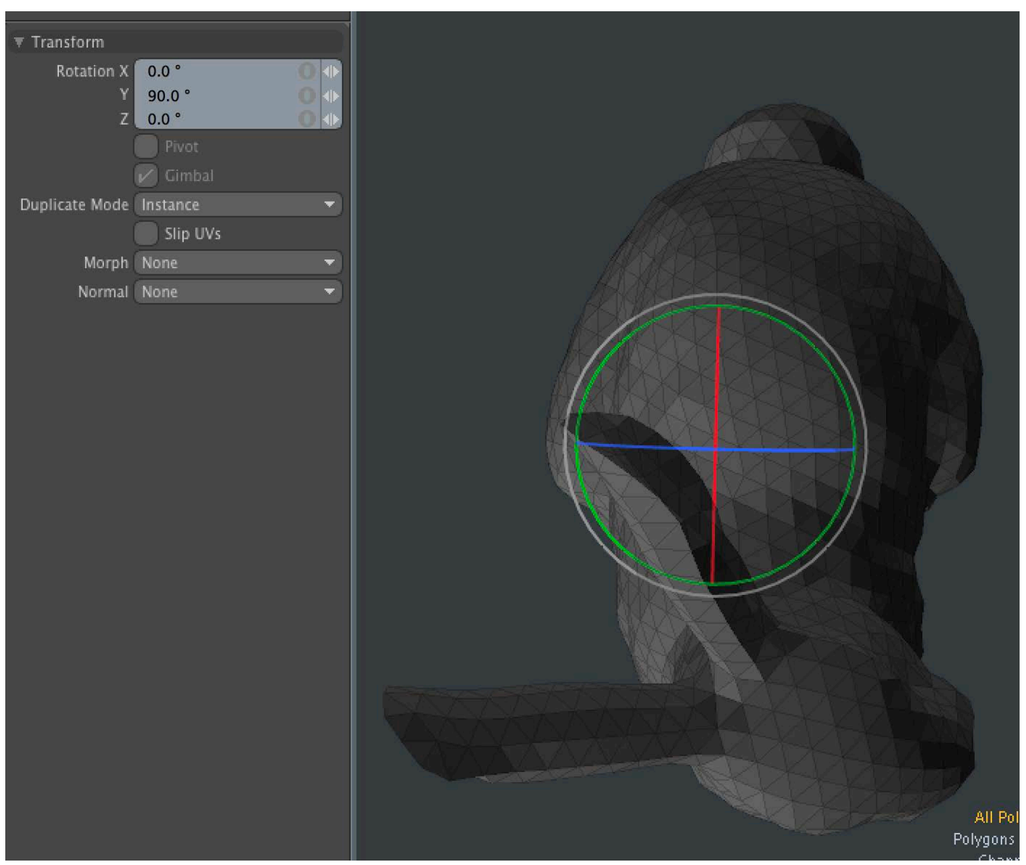
What then must the matrix look like?

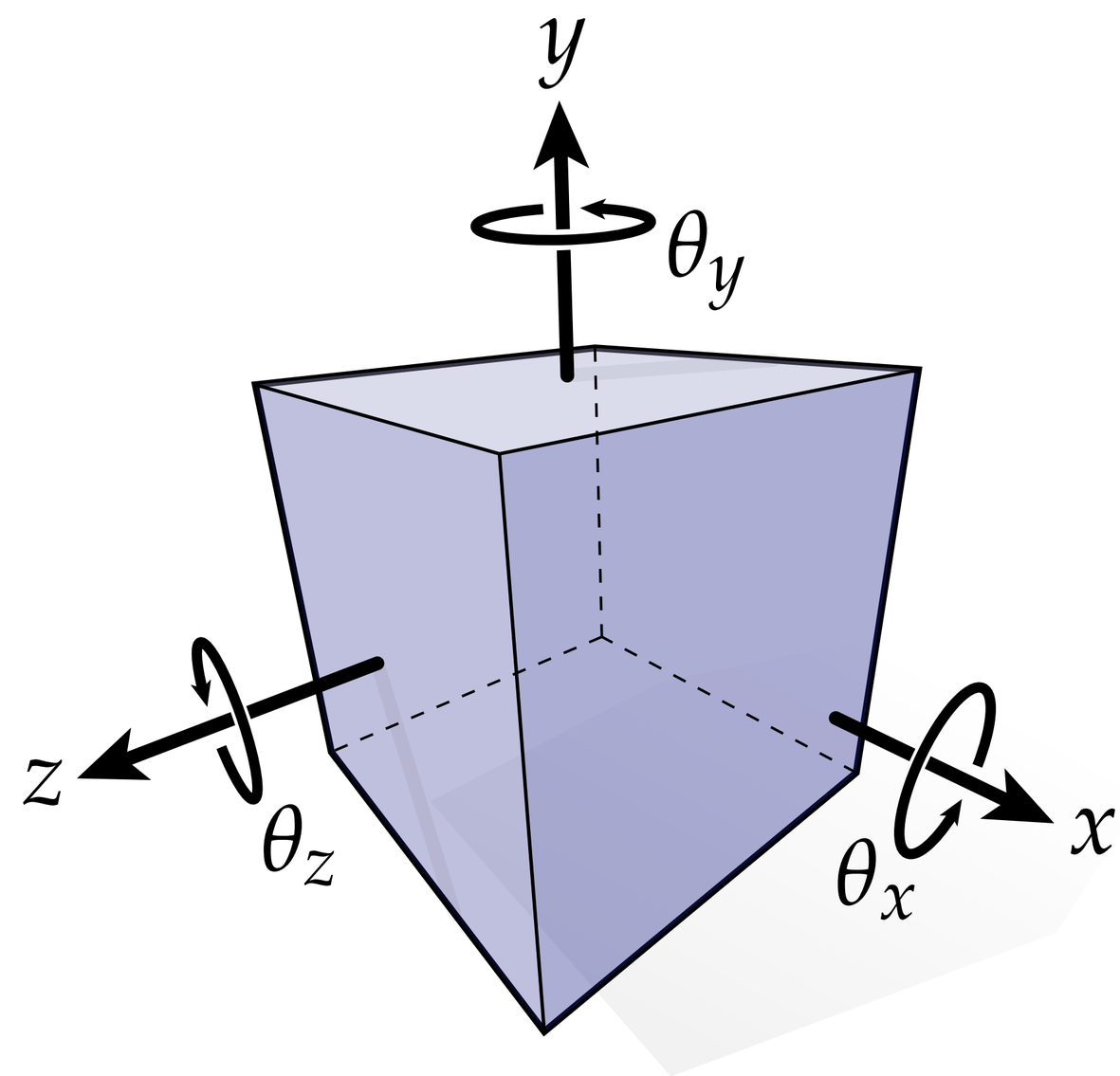
$$\begin{bmatrix} S(\theta) & S(\theta+\pi/2) \end{bmatrix} = \begin{bmatrix} \cos(\theta) & \cos(\theta+\pi/2) \\ \sin(\theta) & \sin(\theta+\pi/2) \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix}$$

***I.e., I don't yet care about sines and cosines and so forth.**

Representing Rotations in 3D—Euler Angles

- How do we express rotations in 3D?
- One idea: we know how to do 2D rotations.
- Why not simply apply rotations around the three axes? (X,Y,Z)
- Scheme is called *Euler angles*
- "Gimbal Lock"





Gimbal Lock

- **When using Euler angles θ_x, θ_y, θ_z, may reach a configuration where there is *no way to rotate around one of the three axes!***
- **Recall rotation matrices around three axes:**

$$R_x = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta_x & -\sin\theta_x \\ 0 & \sin\theta_x & \cos\theta_x \end{bmatrix} \quad R_y = \begin{bmatrix} \cos\theta_y & 0 & \sin\theta_y \\ 0 & 1 & 0 \\ -\sin\theta_y & 0 & \cos\theta_y \end{bmatrix} \quad R_z = \begin{bmatrix} \cos\theta_z & -\sin\theta_z & 0 \\ \sin\theta_z & \cos\theta_z & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

- **Product of these matrices represents rotation by Euler angles:**

$$R_x R_y R_z = \begin{bmatrix} \cos\theta_y \cos\theta_z & -\cos\theta_y \sin\theta_z & \sin\theta_y \\ \cos\theta_z \sin\theta_x \sin\theta_y + \cos\theta_x \sin\theta_z & \cos\theta_x \cos\theta_z - \sin\theta_x \sin\theta_y \sin\theta_z & -\cos\theta_y \sin\theta_x \\ -\cos\theta_x \cos\theta_z \sin\theta_y + \sin\theta_x \sin\theta_z & \cos\theta_z \sin\theta_x + \cos\theta_x \sin\theta_y \sin\theta_z & \cos\theta_x \cos\theta_y \end{bmatrix}$$

- **Consider special case $\theta_y = \pi/2$ (so, $\cos\theta_y = 0$, $\sin\theta_y = 1$):**

$$\Longrightarrow \begin{bmatrix} 0 & 0 & 1 \\ \cos\theta_z \sin\theta_x + \cos\theta_x \sin\theta_z & \cos\theta_x \cos\theta_z - \sin\theta_x \sin\theta_z & 0 \\ -\cos\theta_x \cos\theta_z + \sin\theta_x \sin\theta_z & \cos\theta_z \sin\theta_x + \cos\theta_x \sin\theta_z & 0 \end{bmatrix}$$

Gimbal Lock, continued

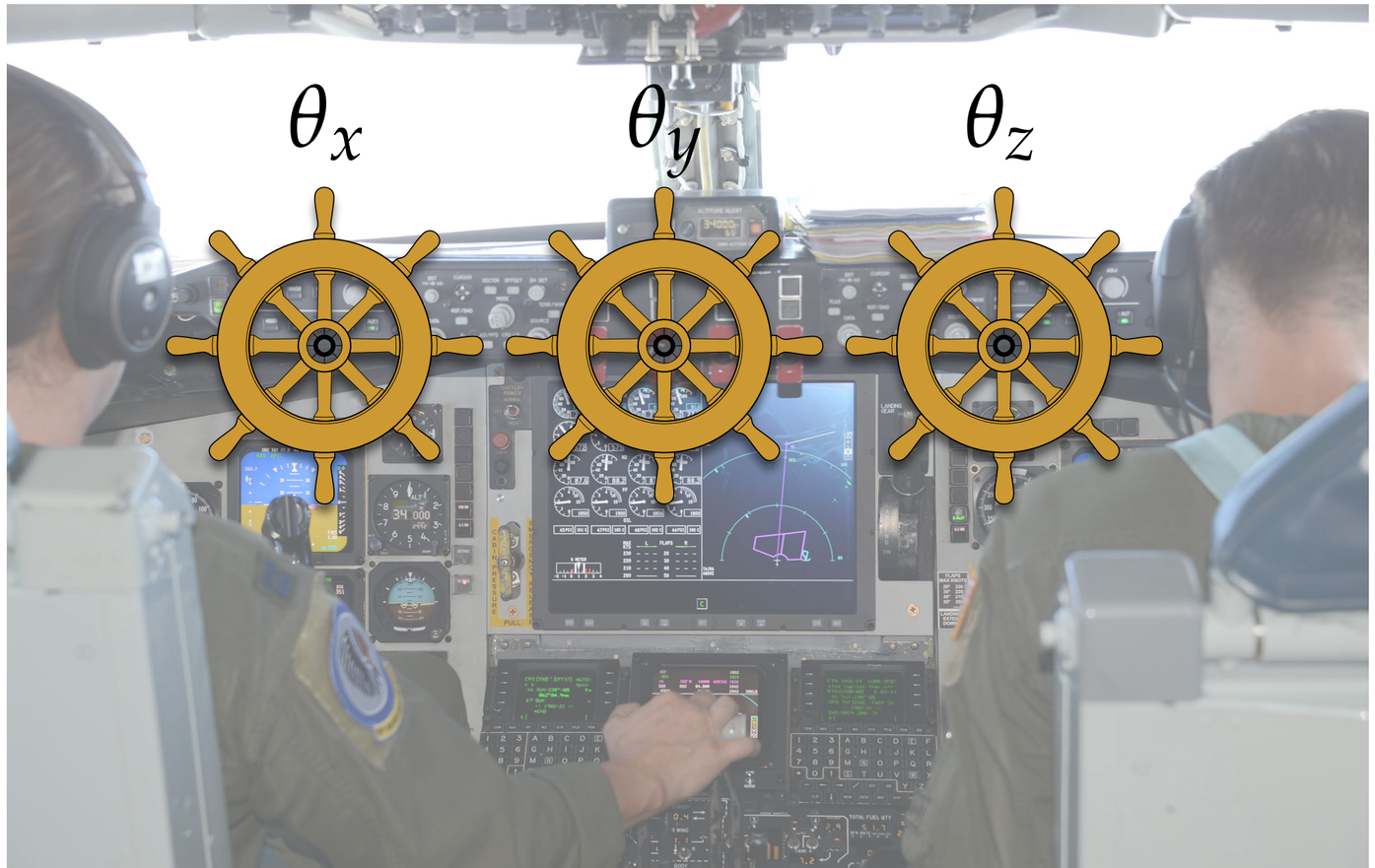
- Simplifying matrix from previous slide, we get

no matter how we adjust θ_x, θ_z, can only rotate in one plane!

$$\begin{bmatrix} 0 & 0 & 1 \\ \sin(\theta_x + \theta_z) & \cos(\theta_x + \theta_z) & 0 \\ -\cos(\theta_x + \theta_z) & \sin(\theta_x + \theta_z) & 0 \end{bmatrix}$$

Q: What does this matrix do?

- We are now "locked" into a single axis of rotation
- Not a great design for airplane controls!



Rotation from Axis/Angle

- **Alternatively, there is a general expression for a matrix that performs a rotation around a given axis u by a given angle θ:**

$$\begin{bmatrix} \cos\theta + u_x^2\,(1-\cos\theta) & u_x u_y\,(1-\cos\theta) - u_z \sin\theta & u_x u_z\,(1-\cos\theta) + u_y \sin\theta \\ u_y u_x\,(1-\cos\theta) + u_z \sin\theta & \cos\theta + u_y^2\,(1-\cos\theta) & u_y u_z\,(1-\cos\theta) - u_x \sin\theta \\ u_z u_x\,(1-\cos\theta) - u_y \sin\theta & u_z u_y\,(1-\cos\theta) + u_x \sin\theta & \cos\theta + u_z^2\,(1-\cos\theta) \end{bmatrix}$$

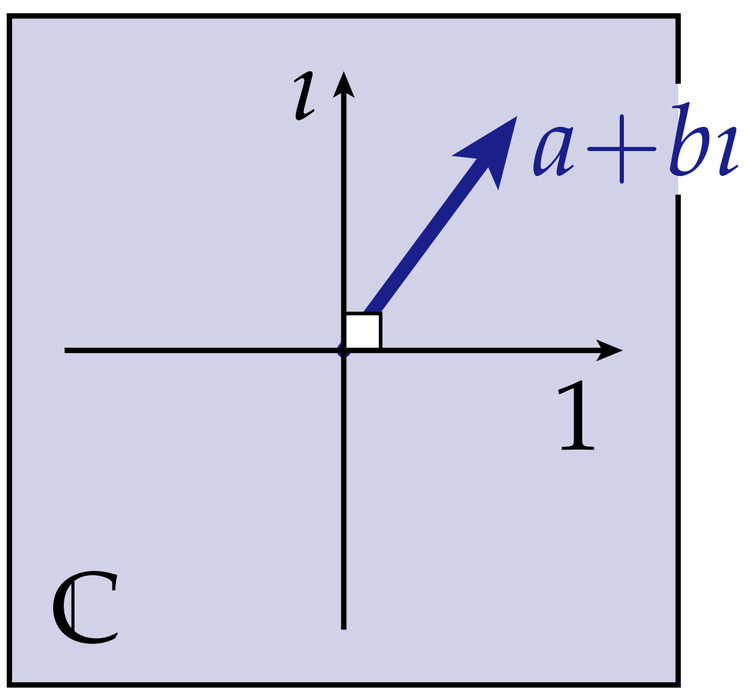
Just memorize this matrix! :-)

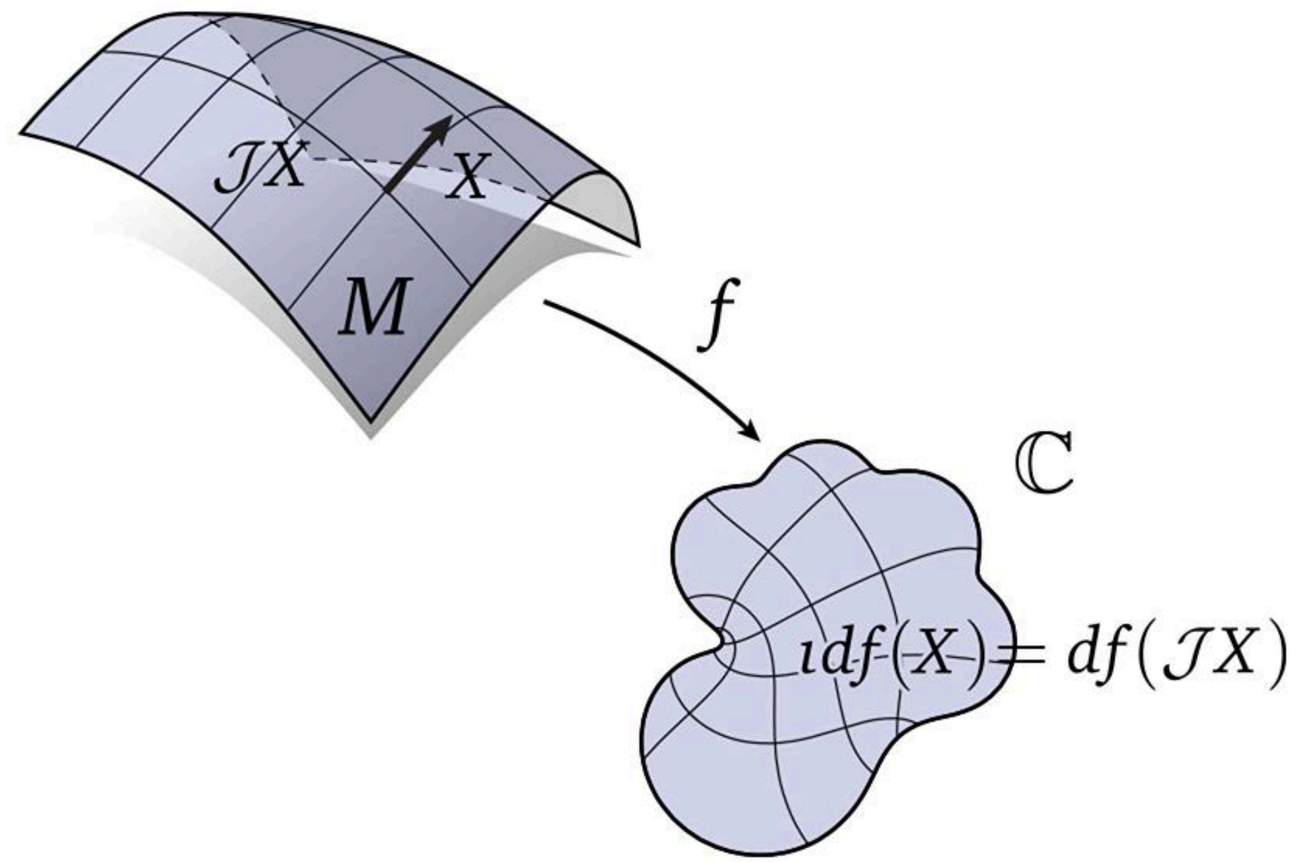
…we'll see a much easier way, later on.

Complex Analysis—Motivation

- Natural way to encode geometric transformations in 2D
- Simplifies code / notation / debugging / *thinking*
- *Moderate* reduction in computational cost/bandwidth/storage
- Fluency with complex analysis can lead into deeper/novel solutions to problems…

COMPLEX





Truly: no good reason to use 2D vectors instead of complex numbers…

DON'T: Think of these numbers as "complex."

DO: Imagine we're simply defining additional operations (like dot and cross).

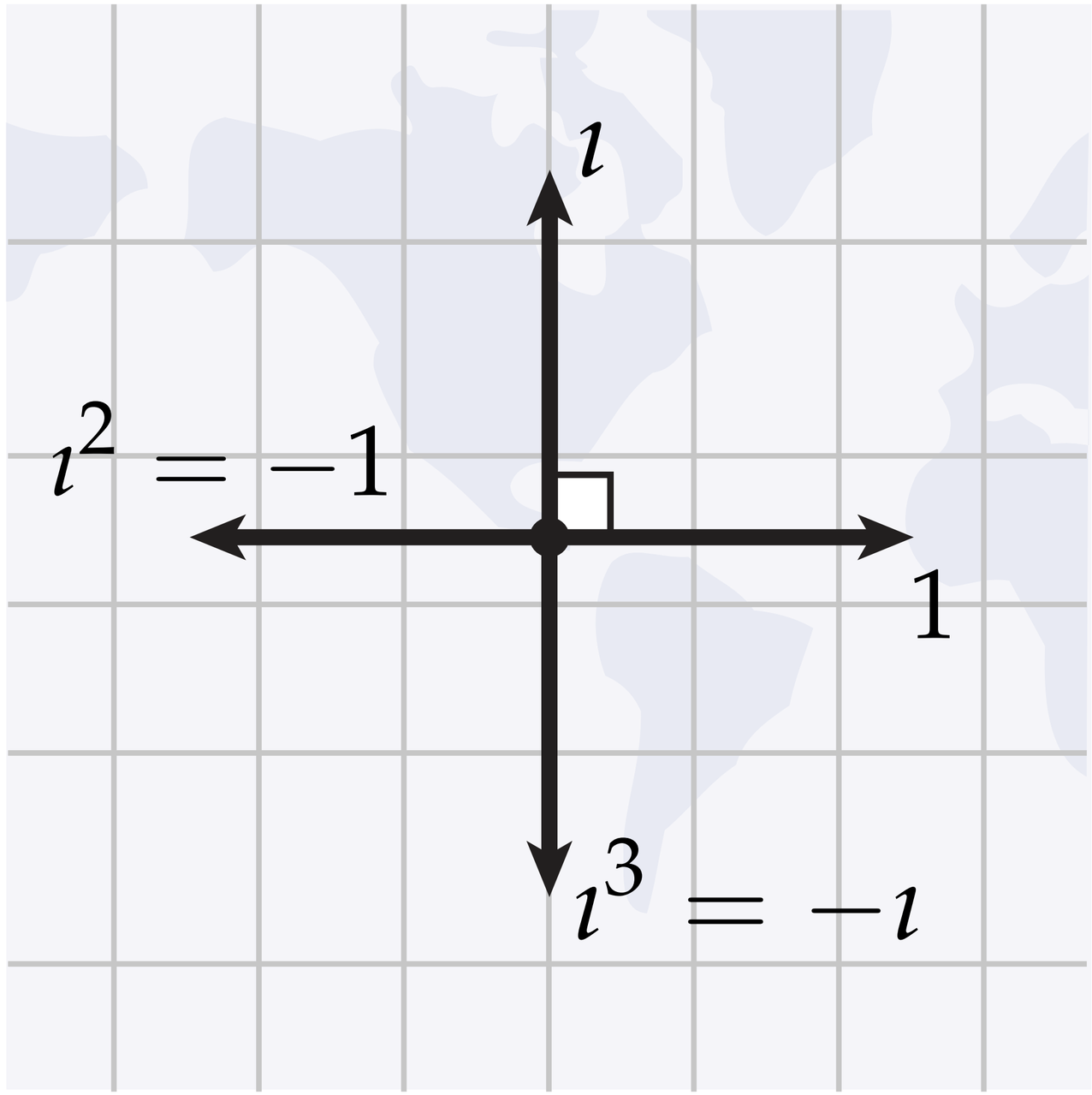
Imaginary Unit

$$\cancel{i := \sqrt{-1}}$$

nonsense!

More importantly: obscures geometric meaning.

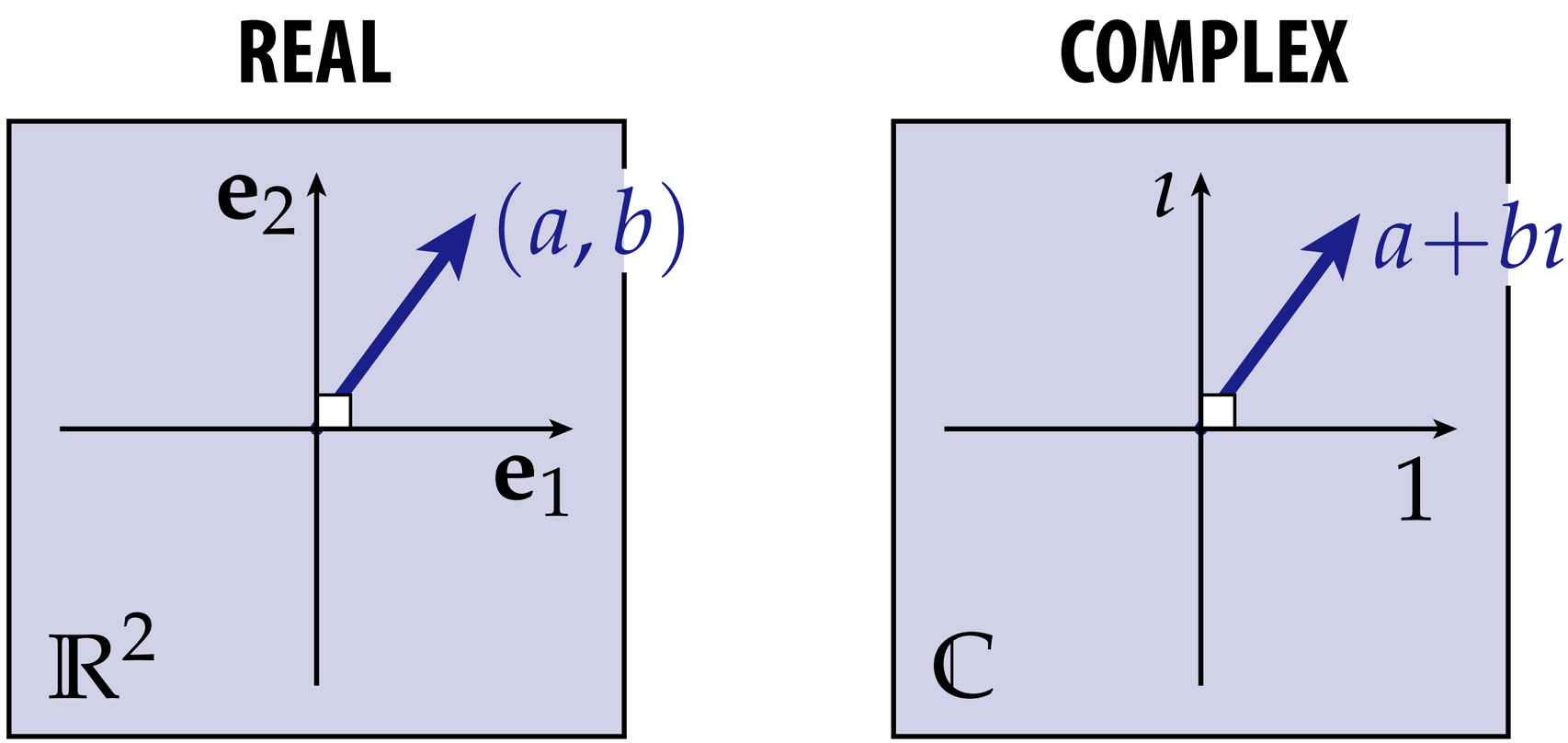
Imaginary Unit—Geometric Description



Imaginary unit is just a quarter-turn in the counter-clockwise direction.

Complex Numbers

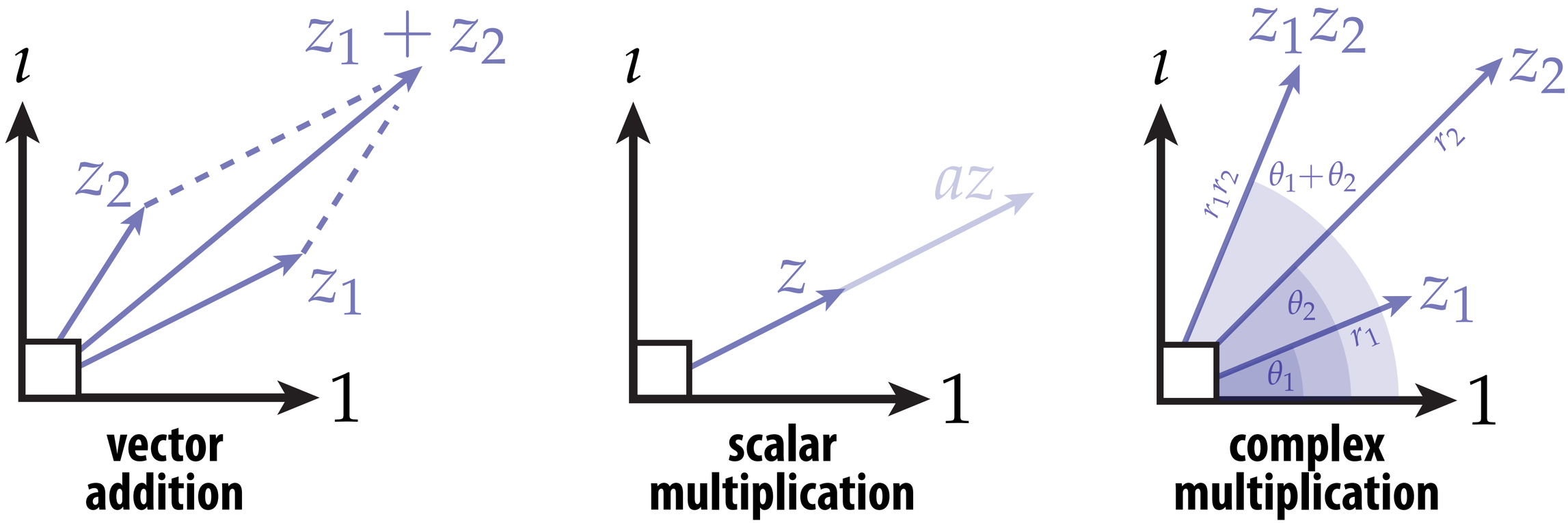
- Complex numbers are then just 2-vectors
- Instead of e_1,e_1, use "1" and "ι" to denote the two bases
- Otherwise, behaves exactly like a real 2-dimensional space



- …except that we're also going to get a very useful new notion of the *product* between two vectors.

Complex Arithmetic

- **Same operations as before, plus one more:**



- **Complex multiplication:**
  - **angles *add***
  - **magnitudes *multiply***

"POLAR FORM"*:

$$z_1 := (r_1, \theta_1)$$
$$z_2 := (r_2, \theta_2)$$
$$z_1 z_2 = (r_1 r_2, \theta_1 + \theta_2)$$

have to be more careful here!

***Not *quite* how it really works, but basic idea is right.**

Complex Product—Rectangular Form

- **Complex product in "rectangular" coordinates (1, ı):**

$$z_1 = (a + b\imath)$$

$$z_2 = (c + d\imath)$$

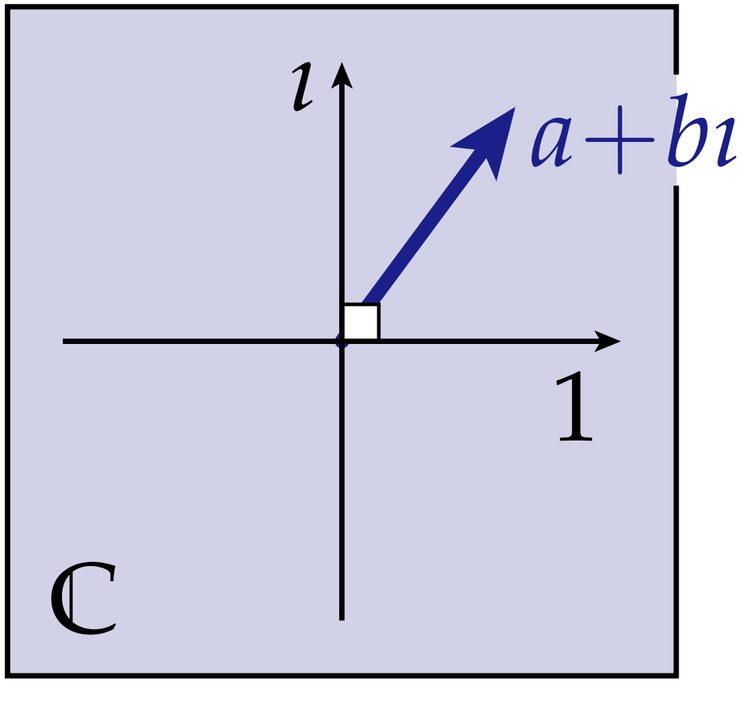
two quarter turns—same as -1

$$z_1 z_2 = ac + ad\imath + bc\imath + bd\imath^2 =$$

$$\boxed{(ac - bd) + (ad + bc)\imath.}$$

"real part" $\mathrm{Re}(z_1 z_2)$

"imaginary part" $\mathrm{Im}(z_1 z_2)$



- **We used a lot of "rules" here. Can you justify them geometrically?**
- **Does this product agree with our geometric description (last slide)?**

Complex Product—Polar Form

Leonhard Euler
(1707–1783)

- **Perhaps most beautiful identity in math:**

$$e^{\imath\pi} + 1 = 0$$

- **Specialization of *Euler's formula*:**

$$e^{\imath\theta} = \cos(\theta) + \imath \sin(\theta)$$

- **Can use to "implement" complex product:**

$$z_1 = ae^{\imath\theta}, \quad z_2 = be^{\imath\phi}$$

$$z_1 z_2 = abe^{\imath(\theta+\phi)}$$

(as with real exponentiation, exponents *add)*

Q: How does this operation differ from our earlier, "fake" polar multiplication?

2D Rotations: Matrices vs. Complex

- **Suppose we want to rotate a vector u by an angle θ, then by an angle ϕ.**

REAL / RECTANGULAR	COMPLEX / POLAR
$\mathbf{u} = (x, y)$ $\mathbf{A} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$ $\mathbf{B} = \begin{bmatrix} \cos\phi & -\sin\phi \\ \sin\phi & \cos\phi \end{bmatrix}$ $\mathbf{Au} = \begin{bmatrix} x\cos\theta - y\sin\theta \\ x\sin\theta + y\cos\theta \end{bmatrix}$ $\mathbf{BAu} = \begin{bmatrix} (x\cos\theta - y\sin\theta)\cos\phi - (x\sin\theta + y\cos\theta)\sin\phi \\ (x\cos\theta - y\sin\theta)\sin\phi + (x\sin\theta + y\cos\theta)\cos\phi \end{bmatrix}$ $= \cdots \text{some trigonometry} \cdots =$ $\mathbf{BAu} = \begin{bmatrix} x\cos(\theta+\phi) - y\sin(\theta+\phi) \\ x\sin(\theta+\phi) + y\cos(\theta+\phi) \end{bmatrix}.$	$u = re^{\imath\alpha}$ $a = e^{\imath\theta}$ $b = e^{\imath\phi}$ $abu = re^{\imath(\alpha+\theta+\phi)}.$

Pervasive theme in graphics:

Sure, there are often many "equivalent" representations.

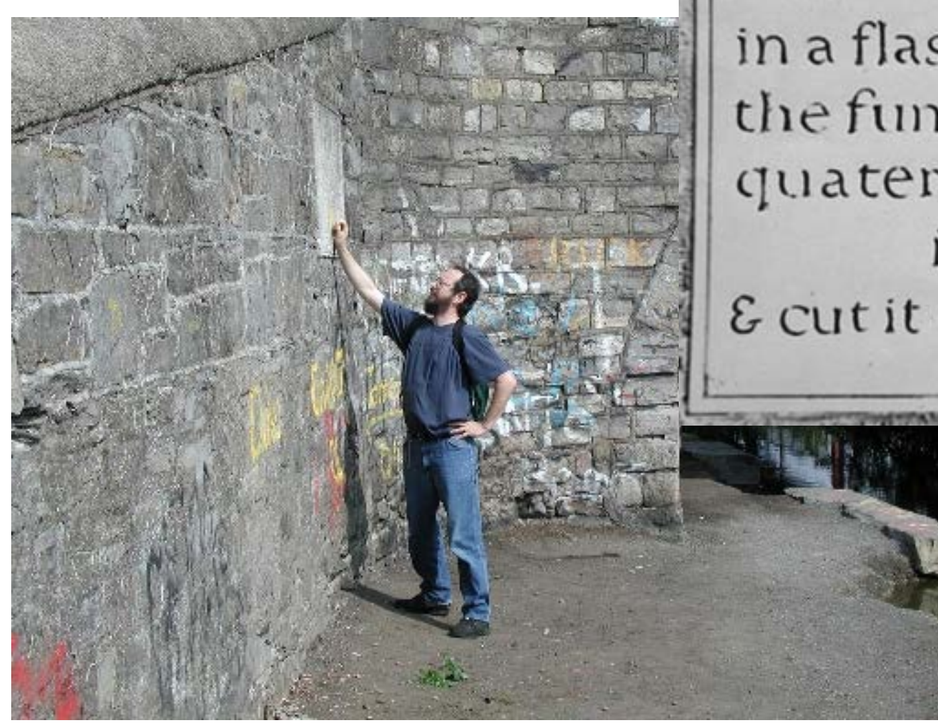
…But why not choose the one that makes life easiest*?

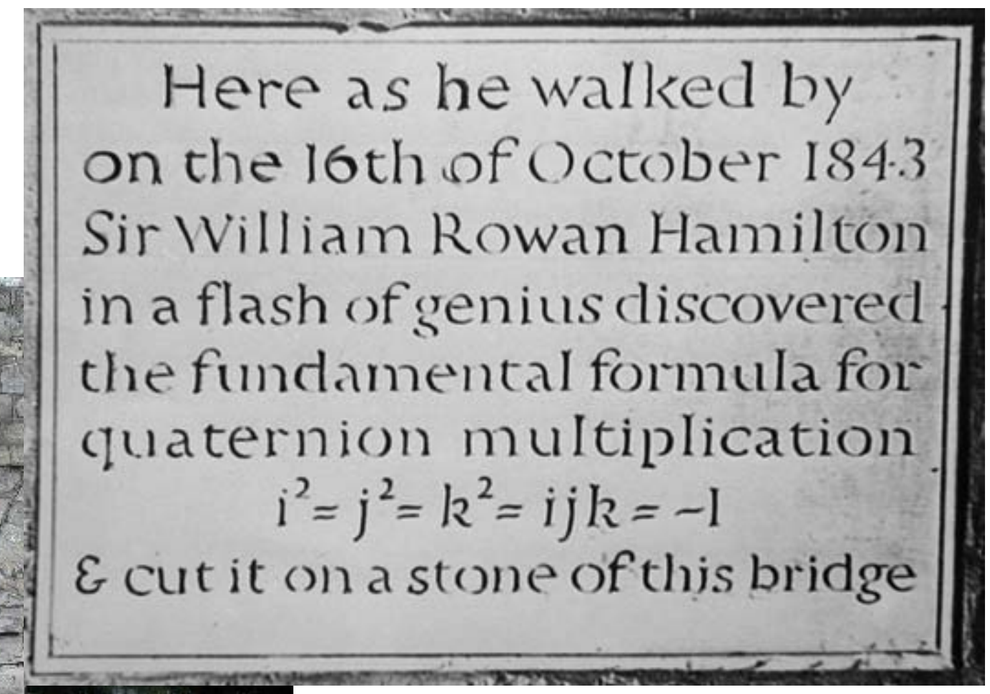
*Or most efficient, or most accurate…

Quaternions

- **TLDR: Kind of like complex numbers but for 3D rotations**
- **Weird situation: can't do 3D rotations w/ only 3 components!**

William Rowan Hamilton (1805-1865)





(Not Hamilton)

Quaternions in Coordinates

- **Hamilton's insight: in order to do 3D rotations in a way that mimics complex numbers for 2D, actually need FOUR coords.**
- **One real, *three* imaginary:**

$$\mathbb{H} := \operatorname{span}(\{1, \imath, \jmath, k\})$$

"H" is for *Hamilton!*

$$q = a + b\imath + c\jmath + dk \in \mathbb{H}$$

- **Quaternion product determined by**

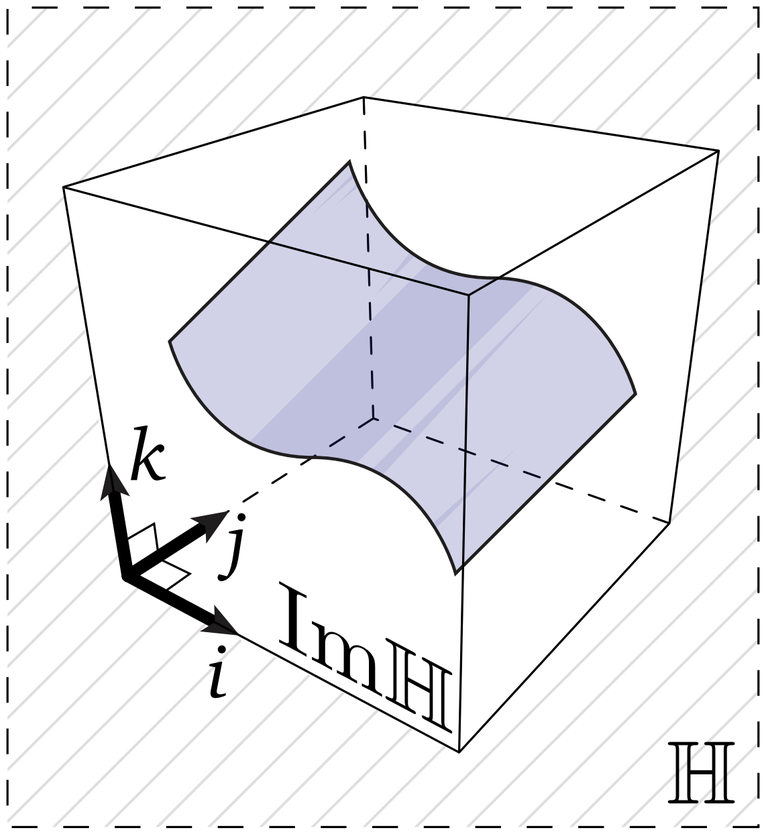
$$\imath^2 = \jmath^2 = k^2 = \imath\jmath k = -1$$

together w/ "natural" rules (distributivity, associativity, etc.)

- **WARNING: product no longer commutes!**

$$\text{For } q, p \in \mathbb{H}, \qquad qp \neq pq$$

(Why might it make sense that it doesn't commute?)



Quaternion Product in Components

- **Given two quaternions**

$$q = a_1 + b_1\imath + c_1\jmath + d_1k$$
$$p = a_2 + b_2\imath + c_2\jmath + d_2k$$

- **Can express their product as**

$$\begin{aligned} qp = {} & a_1a_2 - b_1b_2 - c_1c_2 - d_1d_2 \\ & +(a_1b_2 + b_1a_2 + c_1d_2 - d_1c_2)\imath \\ & +(a_1c_2 - b_1d_2 + c_1a_2 + d_1b_2)\jmath \\ & +(a_1d_2 + b_1c_2 - c_1b_2 + d_1a_2)k \end{aligned}$$

…fortunately there is a (much) nicer expression.

Quaternions—Scalar + Vector Form

- **If we have *four* components, how do we talk about pts in 3D?**
- **Natural idea: we have three imaginary parts—why not use these to encode 3D vectors?**

$$(x, y, z) \mapsto 0 + x\imath + y\jmath + zk$$

- **Alternatively, can think of a quaternion as a pair**

$$\begin{array}{c} (\text{ scalar, vector }) \in \mathbb{H} \\ \cap \qquad\quad \cap \\ \mathbb{R} \qquad\quad \mathbb{R}^3 \end{array}$$

- **Quaternion product then has simple(r) form:**

$$(a, \mathbf{u})(b, \mathbf{v}) = (ab - \mathbf{u} \cdot \mathbf{v}, a\mathbf{v} + b\mathbf{u} + \mathbf{u} \times \mathbf{v})$$

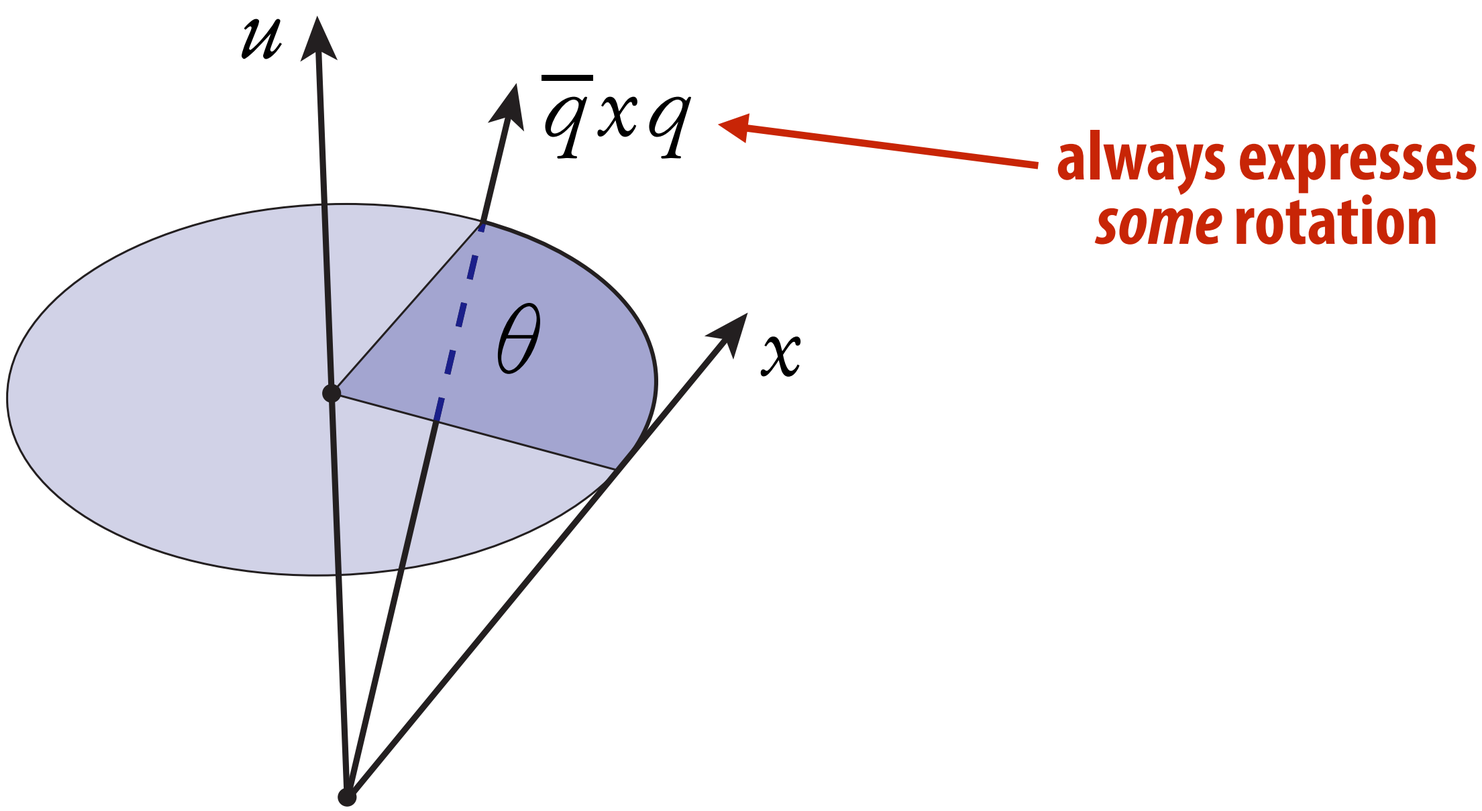
- **For vectors in R3, gets even simpler:**

$$\mathbf{uv} = \mathbf{u} \times \mathbf{v} - \mathbf{u} \cdot \mathbf{v}$$

3D Transformations via Quaternions

- **Main use for quaternions in graphics? *Rotations.***
- **Consider vector x ("pure imaginary") and *unit* quaternion q:**

$$x \in \mathrm{Im}(\mathbb{H})$$
$$q \in \mathbb{H}, \quad |q|^2 = 1$$



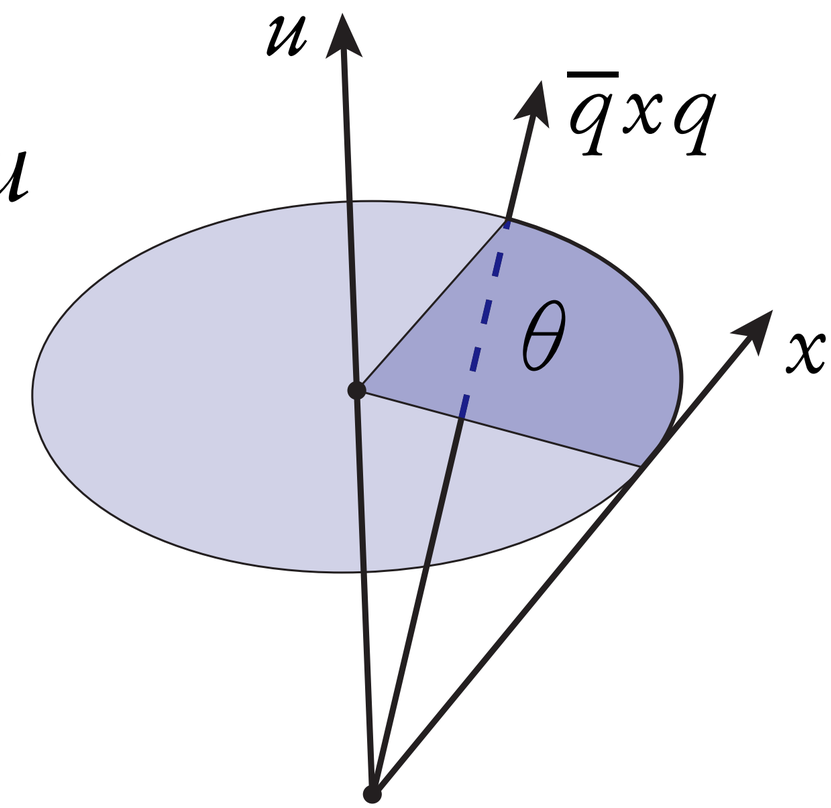
Rotation from Axis/Angle, Revisited

- Given axis u, angle θ, quaternion q representing rotation is

$$q = \cos(\theta/2) + \sin(\theta/2)u$$

angle

axis



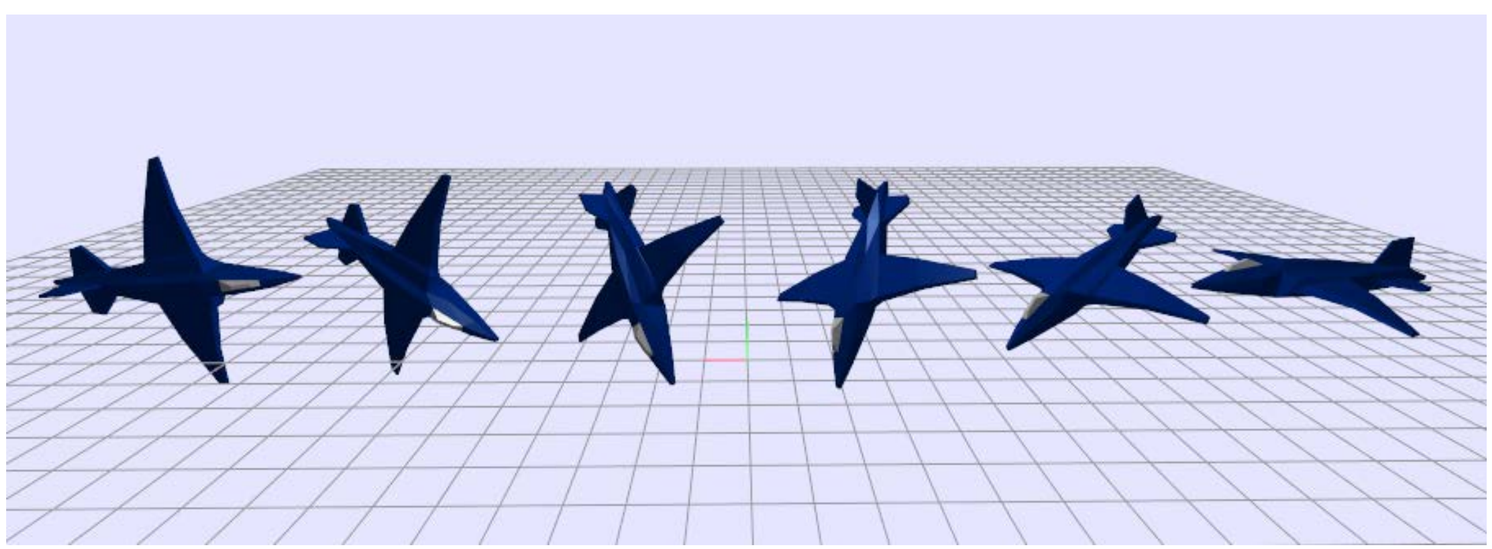
- Much easier to remember (and manipulate) than matrix!

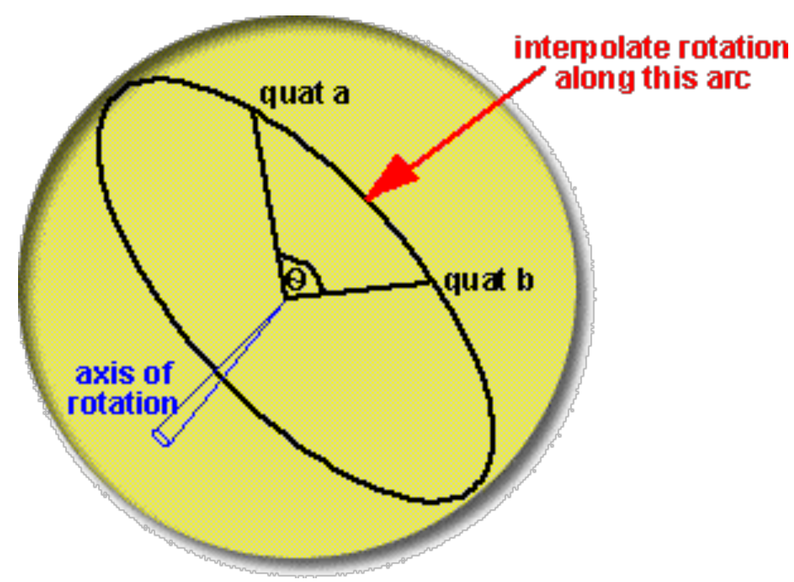
$$\begin{bmatrix} \cos\theta + u_x^2(1-\cos\theta) & u_x u_y(1-\cos\theta) - u_z\sin\theta & u_x u_z(1-\cos\theta) + u_y\sin\theta \\ u_y u_x(1-\cos\theta) + u_z\sin\theta & \cos\theta + u_y^2(1-\cos\theta) & u_y u_z(1-\cos\theta) - u_x\sin\theta \\ u_z u_x(1-\cos\theta) - u_y\sin\theta & u_z u_y(1-\cos\theta) + u_x\sin\theta & \cos\theta + u_z^2(1-\cos\theta) \end{bmatrix}$$

Interpolating Rotations

- **Suppose we want to smoothly interpolate between two rotations (e.g., orientations of an airplane)**
- **Interpolating Euler angles can yield strange-looking paths, non-uniform rotation speed, …**
- **Simple solution* w/ quaternions: "SLERP" (spherical linear interpolation):**

$$\mathrm{Slerp}(q_0, q_1, t) = q_0(q_0^{-1}q_1)^t, \quad t \in [0,1]$$



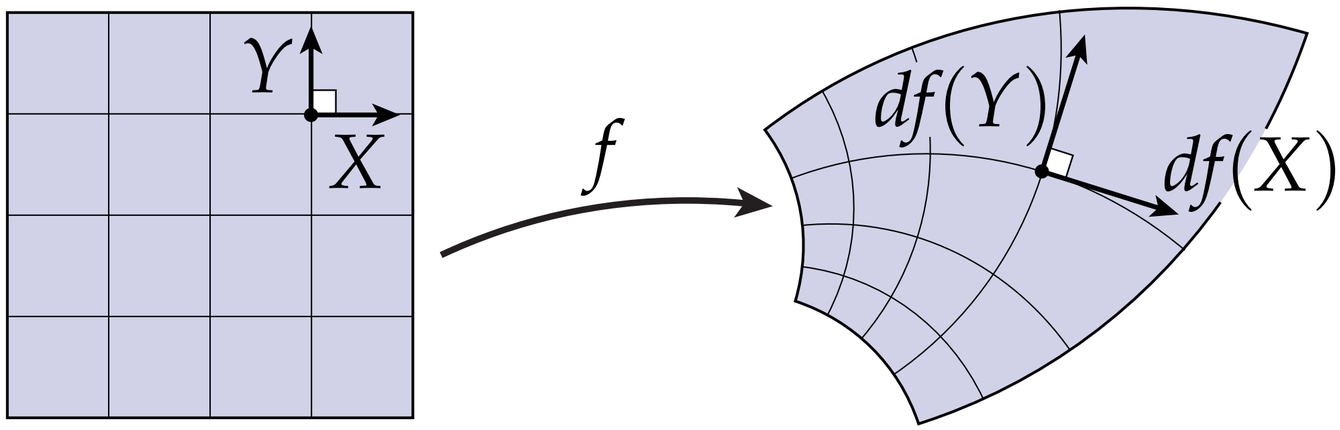


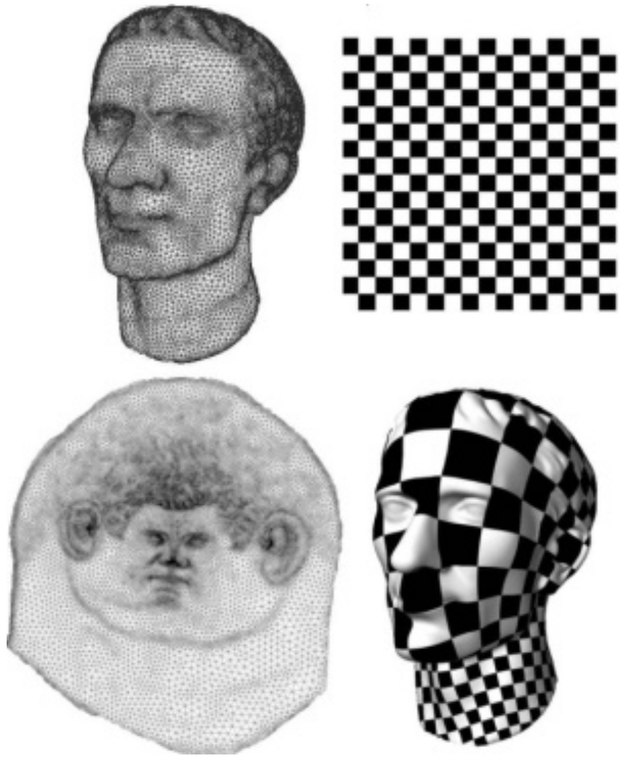
*Shoemake 1985, "Animating Rotation with Quaternion Curves"

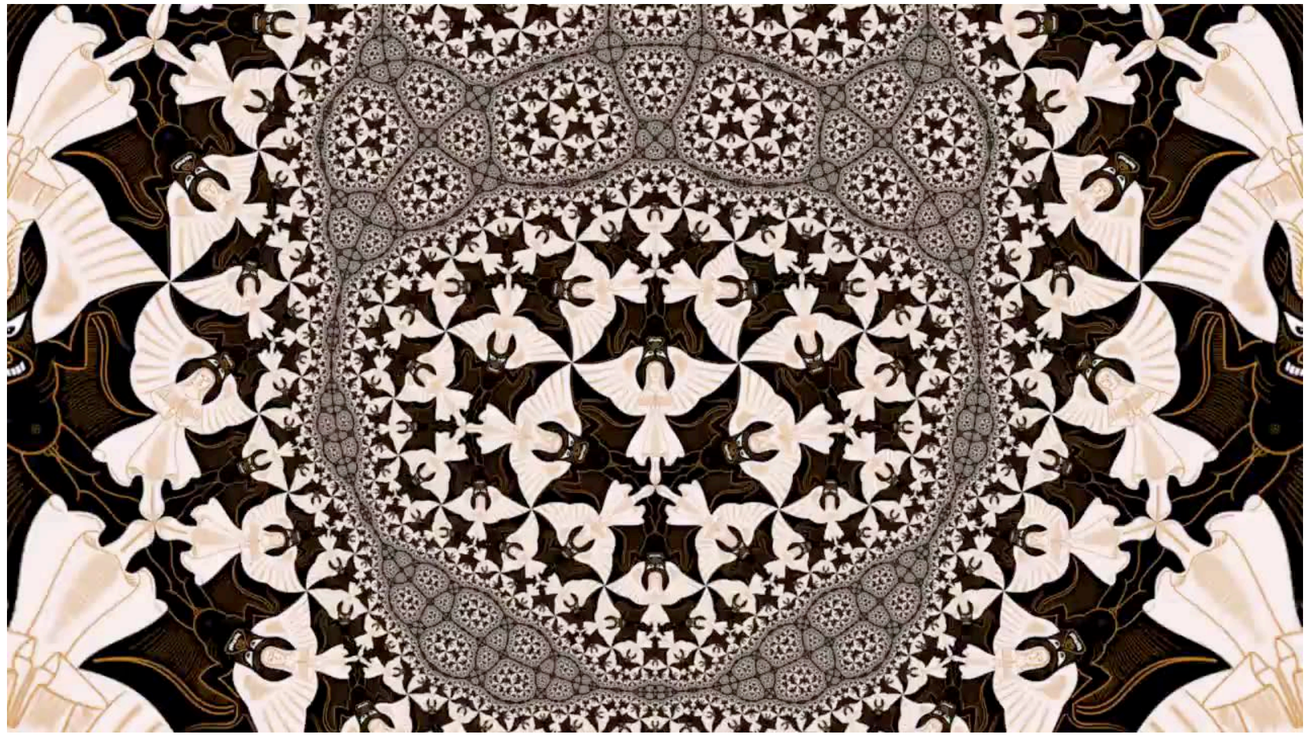
Where else are (hyper-)complex numbers useful in computer graphics?

Generating Coordinates for Texture Maps

Complex numbers are natural language for angle-preserving ("conformal") maps



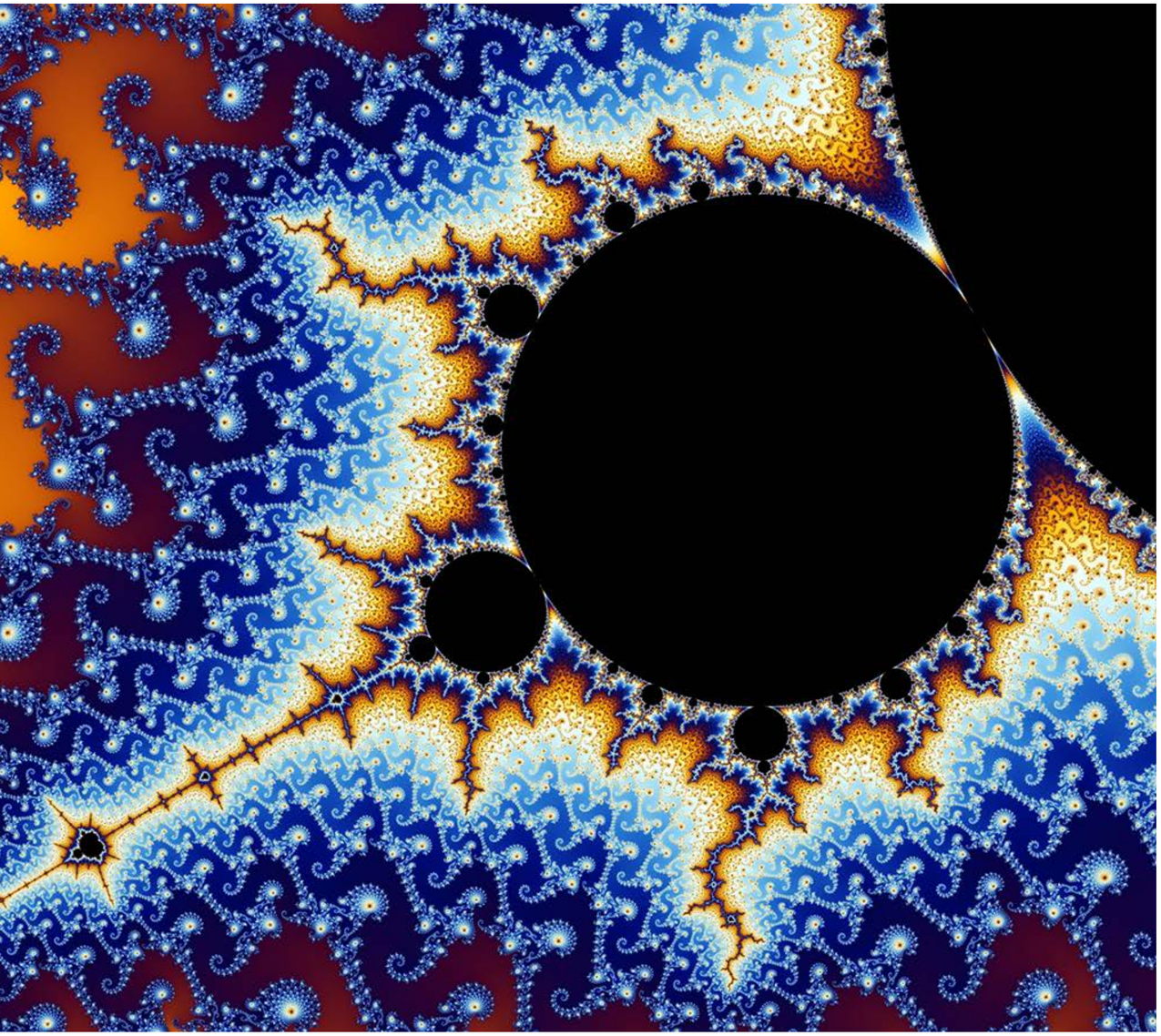


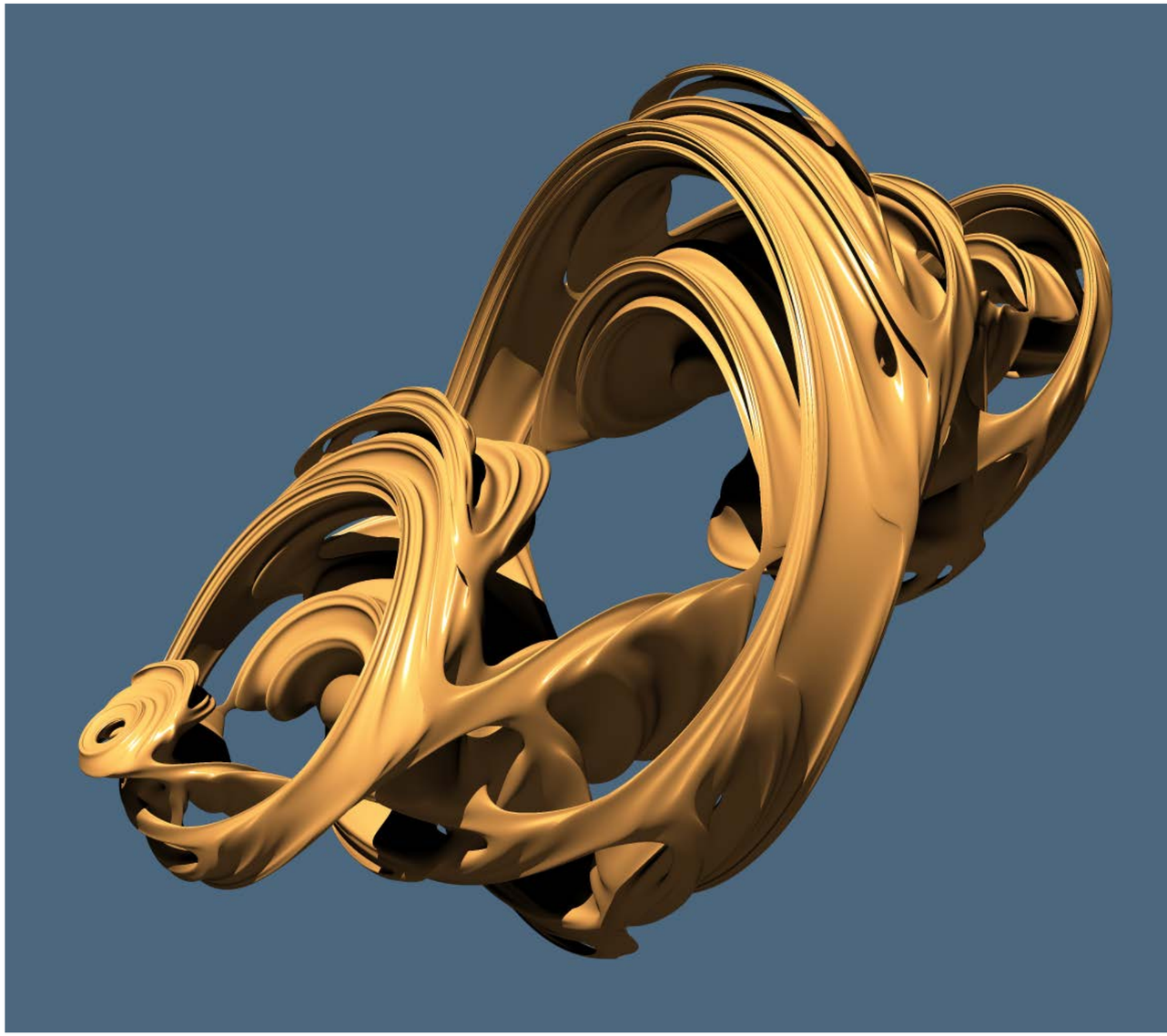


Preserving angles in texture well-tuned to human perception…

Useless-But-Beautiful Example: Fractals

- Defined in terms of iteration on (hyper)complex numbers:

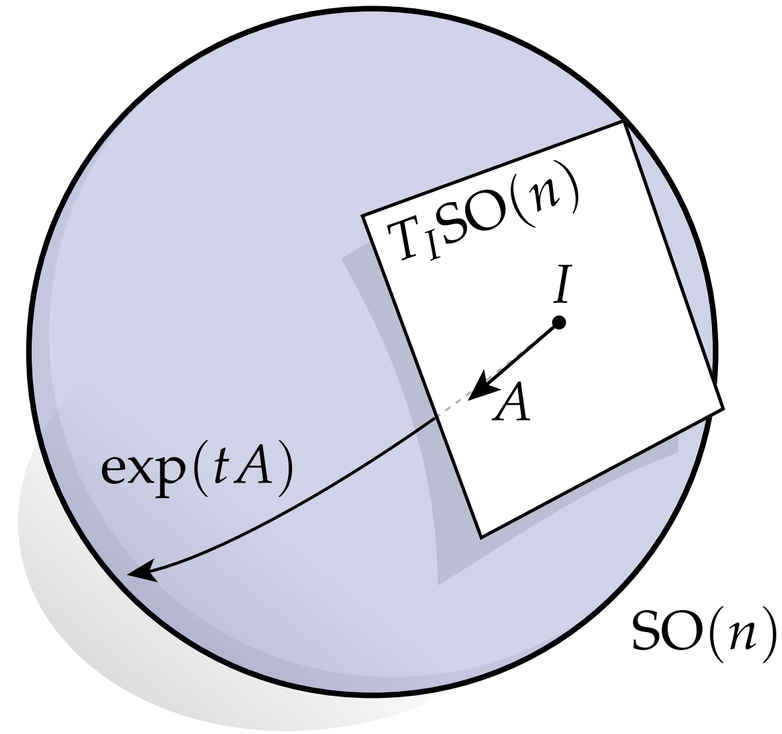


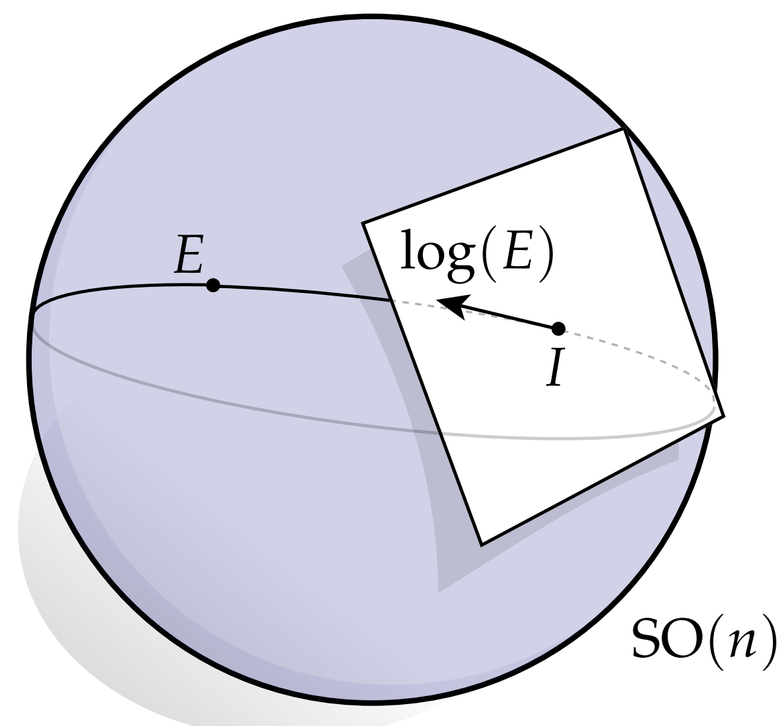


(Will see exactly how this works later in class.)

Not Covered: Lie algebras/Lie Groups

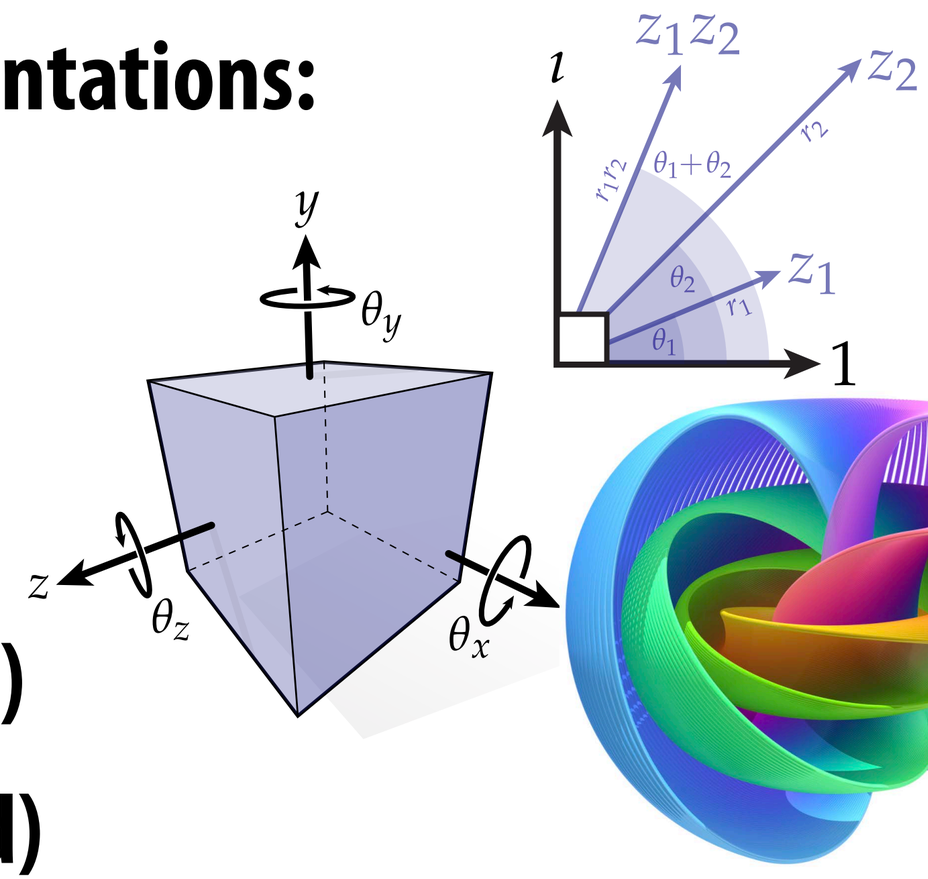
- Another super nice/useful perspective on rotations is via "Lie groups" and "Lie algebras"
- More than we have time to cover!
- Many benefits similar to quaternions (easy axis/angle representation, no gimbal lock, …)
- Nice for encoding angles bigger than 2π
- Also *very* useful for taking averages of rotations
- (Very) short story:
  - exponential map takes you from axis/angle to rotation matrix
  - logarithmic map takes you from rotation matrix to axis/angle





Rotations and Complex Representations—Summary

- Rotations are surprisingly complicated in 3D!
- Today, looked at how <u>complex</u> representations help understand/work with rotations in 3D (& 2D)
- In general, many possible representations:
  - Euler angles
  - axis-angle
  - quaternions
  - Lie group/algebra (not covered)
  - geometric algebra (not covered)



- There's no "right" or "best" way—the more you know, the more you'll be able to do!

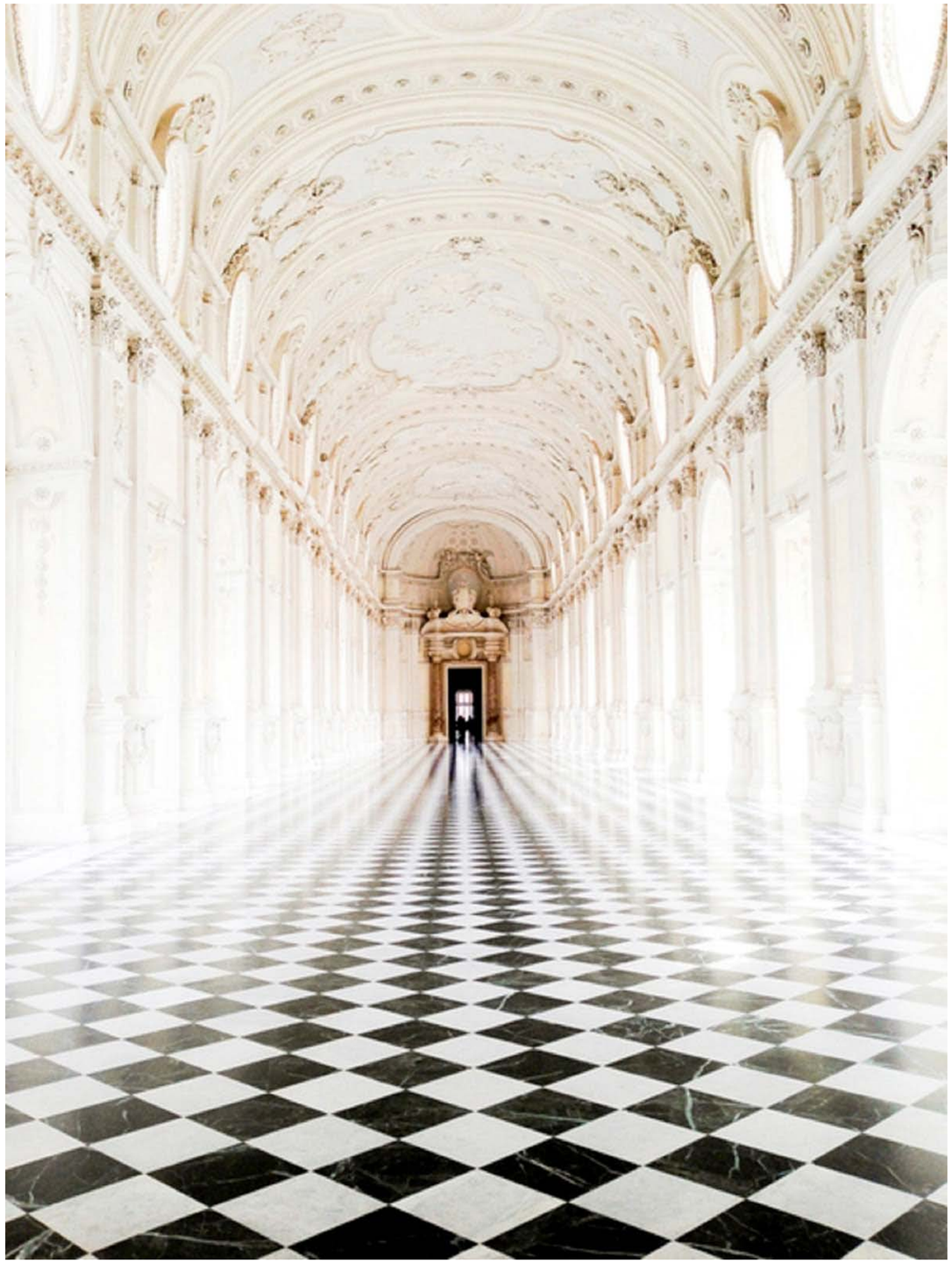
Next time: Perspective & Texture Mapping

Perspective Projection and Texture Mapping

Computer Graphics
CMU 15-462/15-662

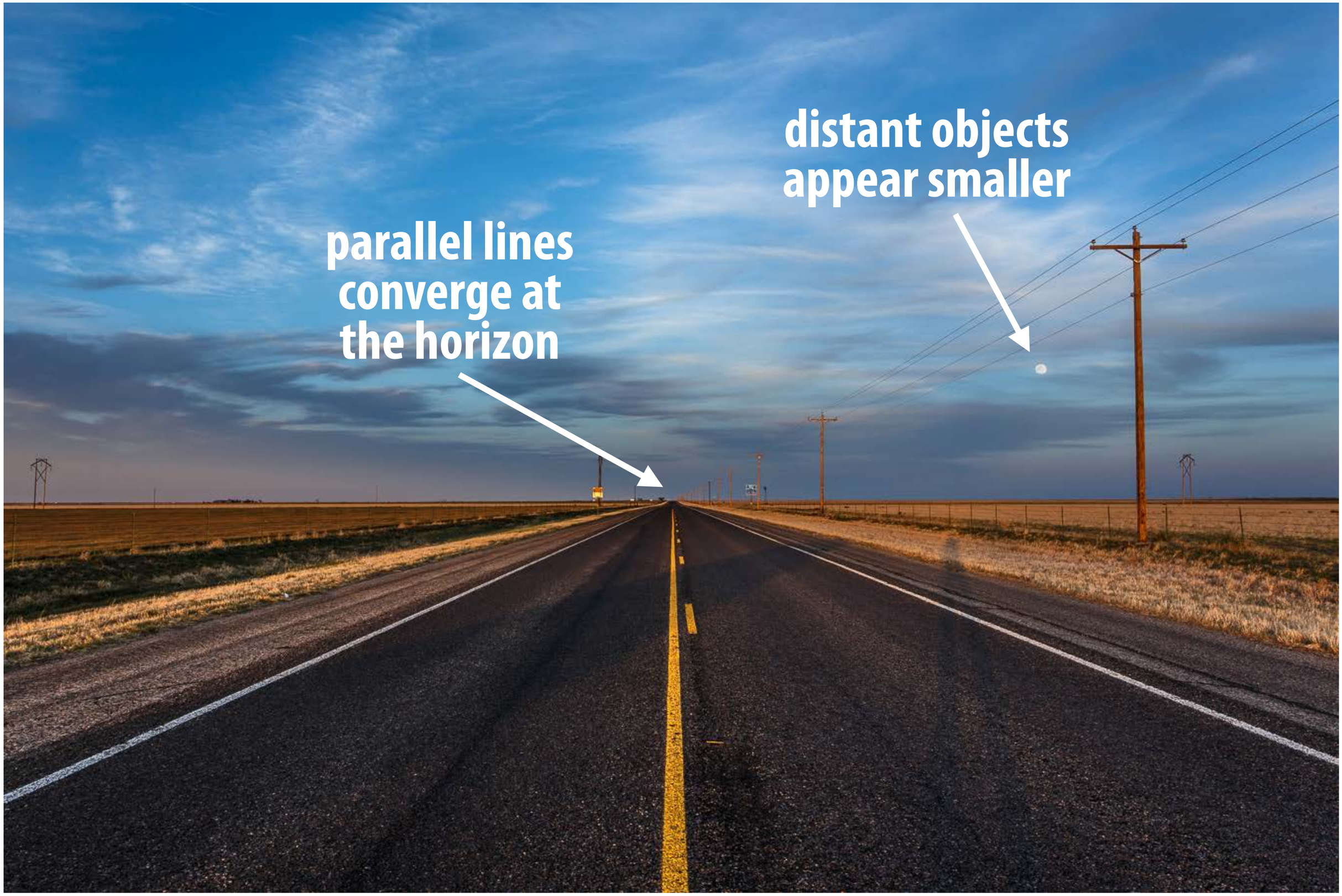
Perspective & Texture

- **PREVIOUSLY:**
  - ***rasterization***
    (how to turn primitives into pixels)
  - ***transformations***
    (how to manipulate primitives in space)
- **TODAY:**
  - **see where these two ideas come crashing together!**
  - **revisit *perspective* transformations**
  - **talk about how to map *texture* onto a primitive to get more detail**
  - **…and how perspective creates challenges for texture mapping!**



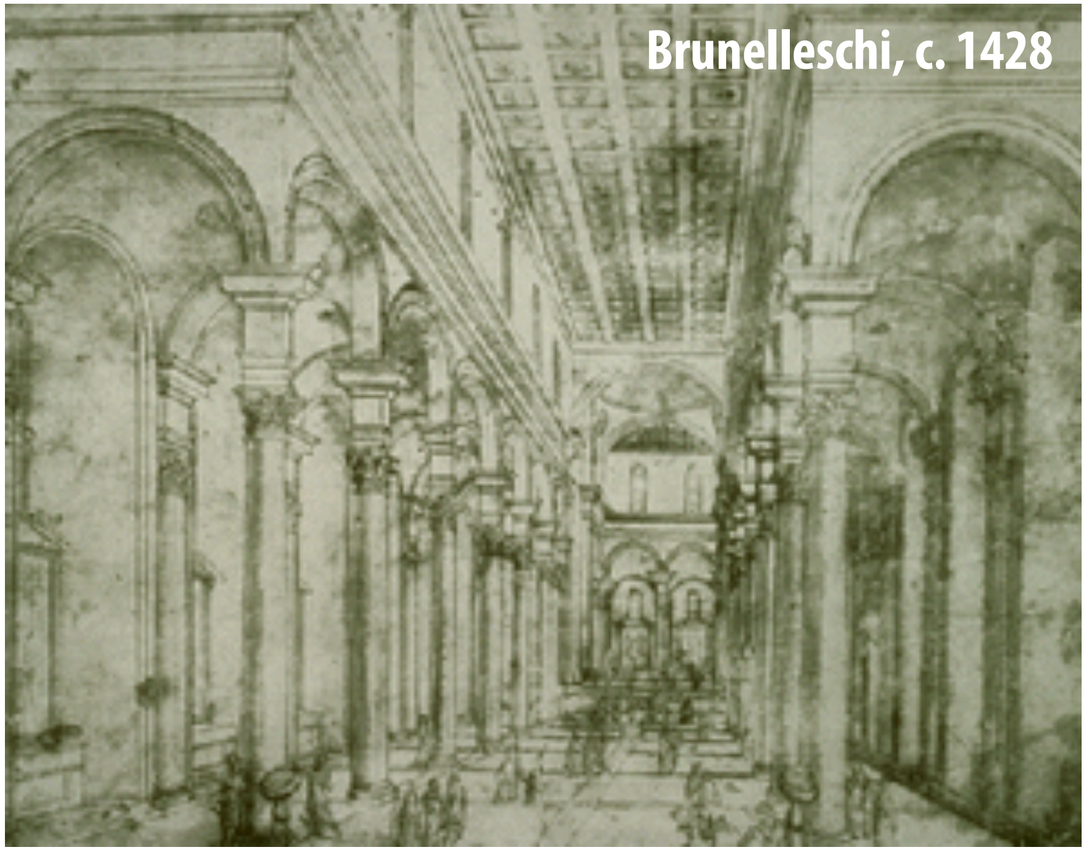
Perspective Projection

Perspective projection



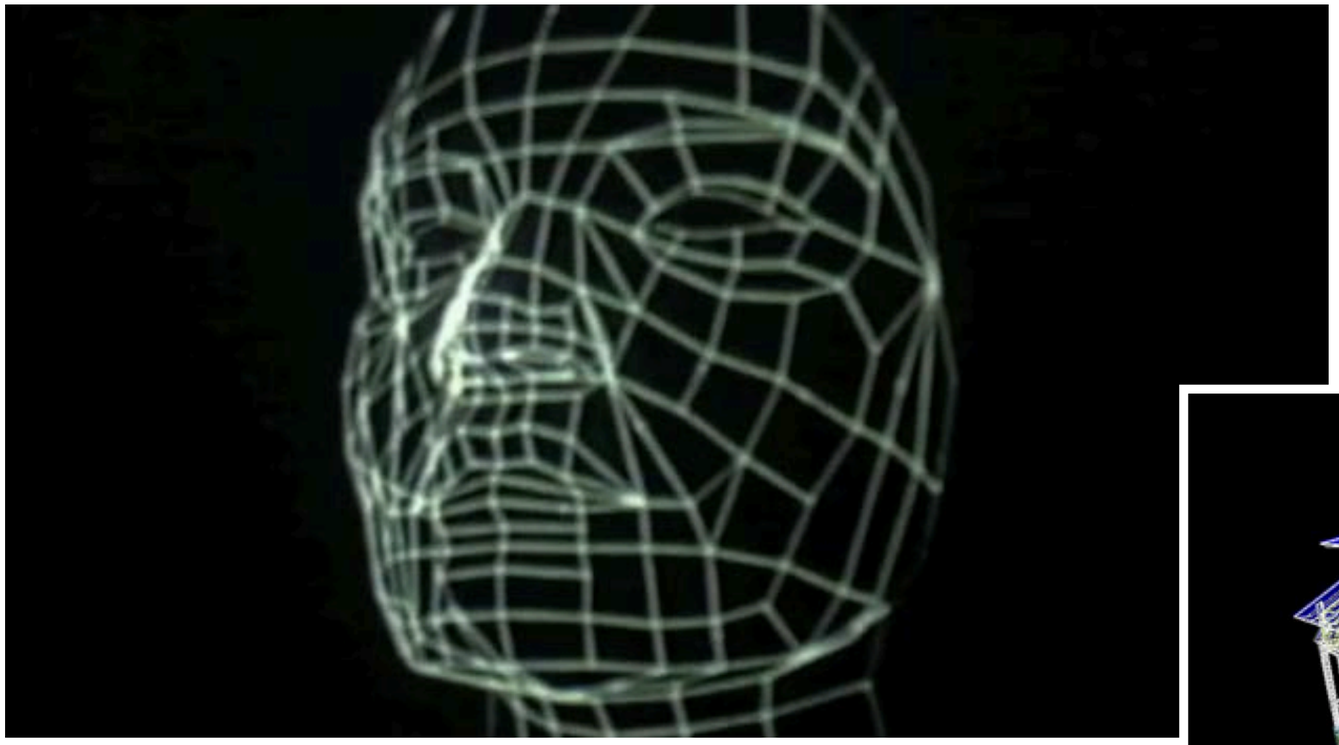
Early painting: incorrect perspective

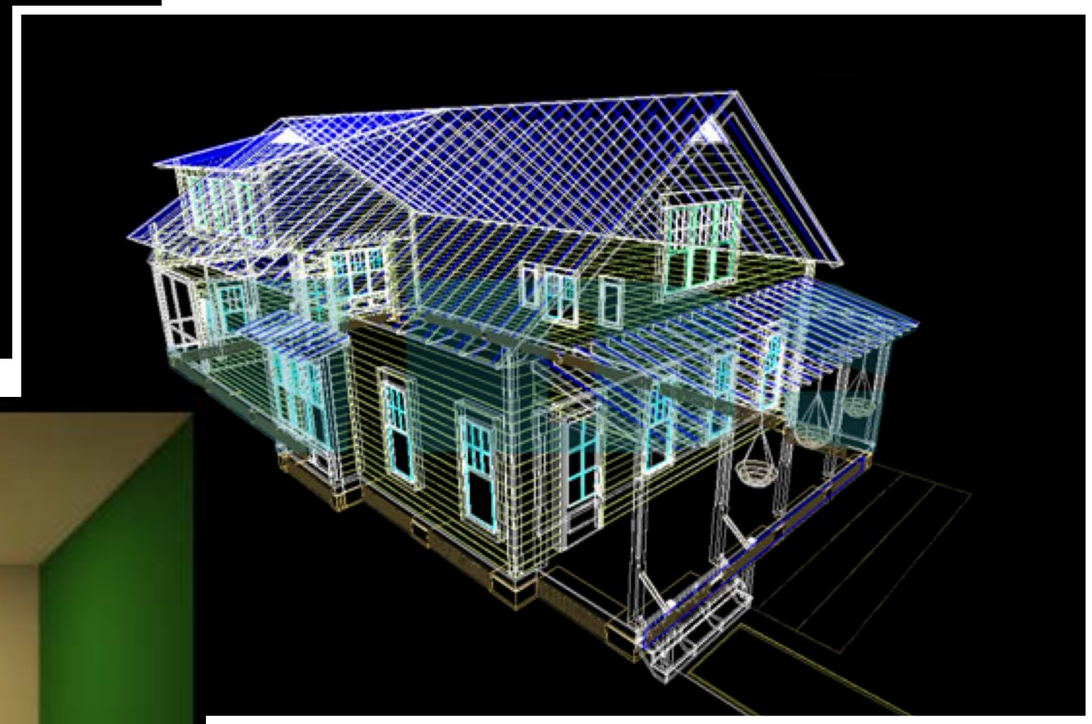
Evolution toward correct perspective



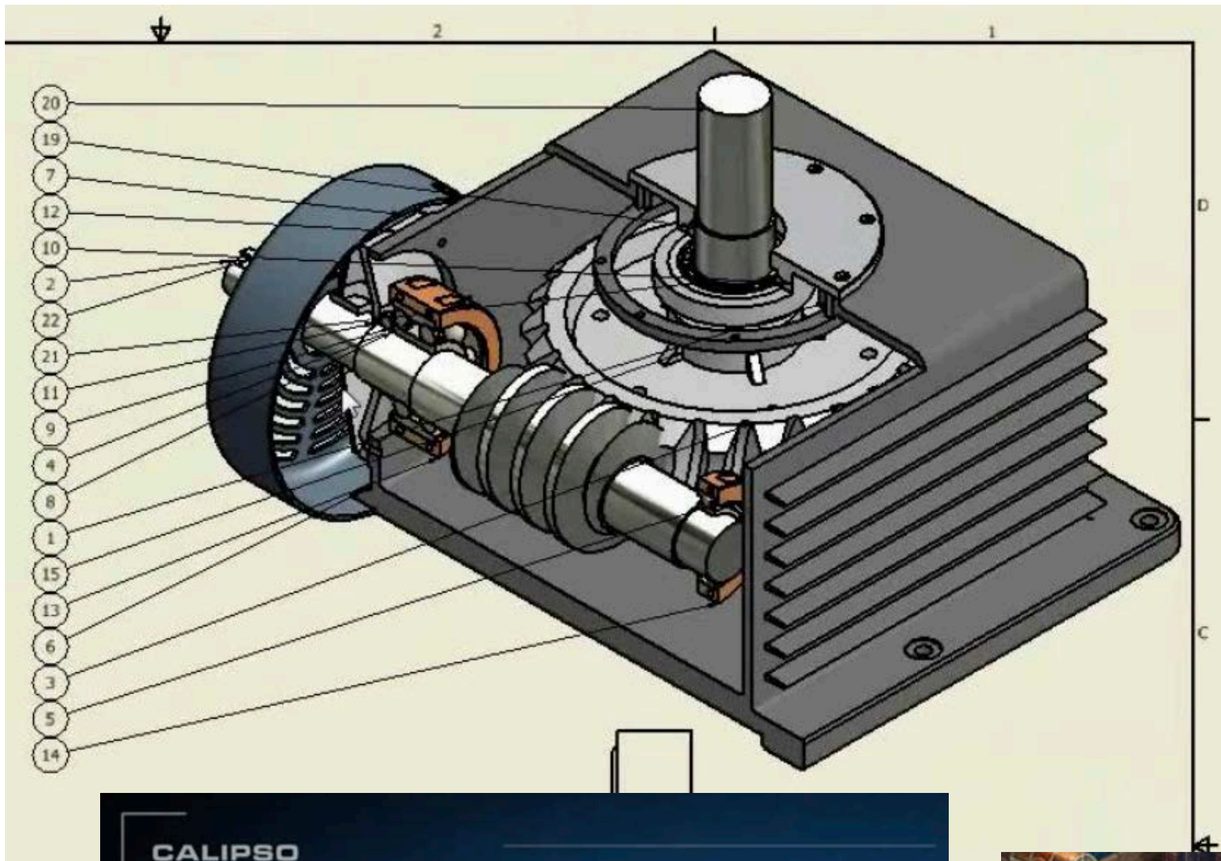
Later… rejection of proper perspective projection

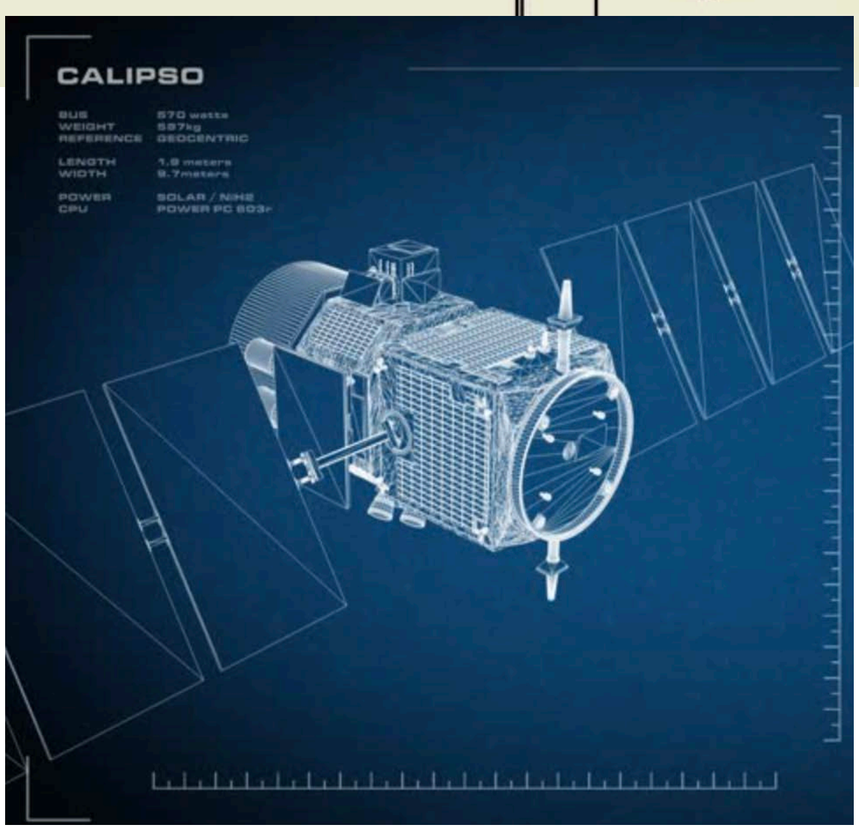
Return of perspective in computer graphics





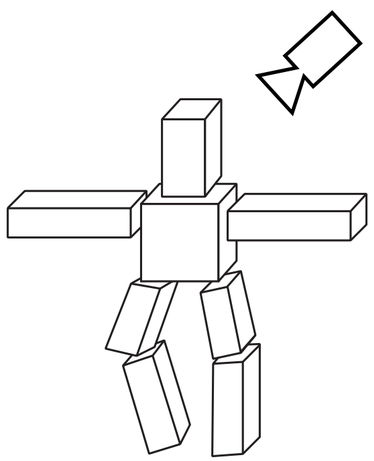
Rejection of perspective in computer graphics





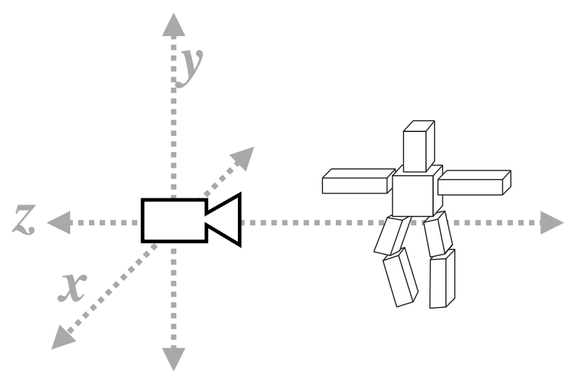
Transformations: From Objects to the Screen

[WORLD COORDINATES]



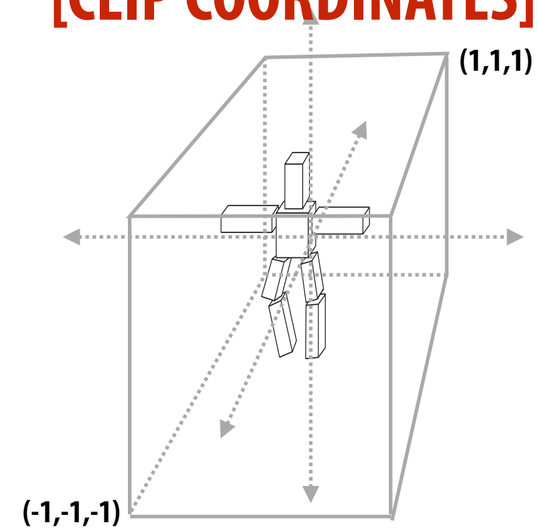
original description of objects

[VIEW COORDINATES]



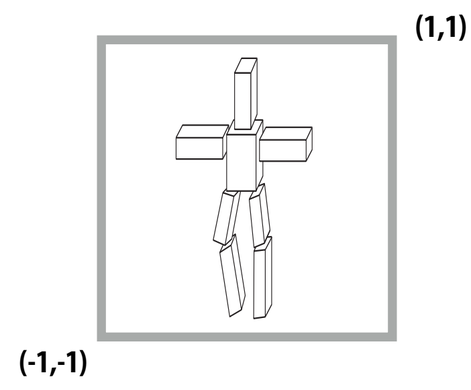
all positions now expressed relative to camera; camera is sitting at origin looking down -z direction

[CLIP COORDINATES]



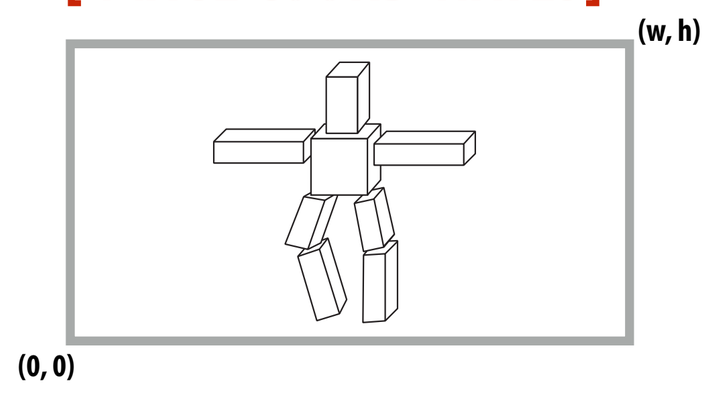
everything visible to the camera is mapped to unit cube for easy "clipping"

[NORMALIZED COORDINATES]



unit cube mapped to unit square via perspective divide

[IMAGE COORDINATES]

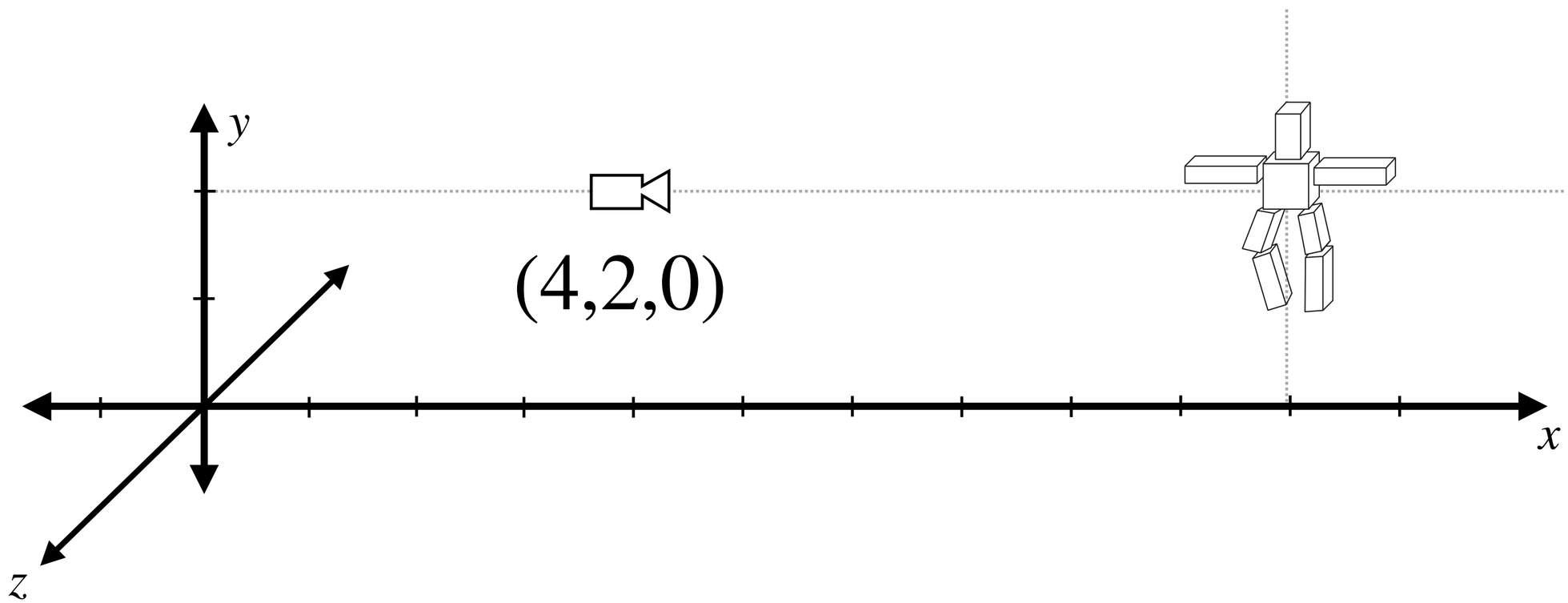


coordinates stretched to match image dimensions (and flipped upside-down)

2D primitives can now be drawn via rasterization

Review: simple camera transform

Consider camera at $(4,2,0)$, looking down x-axis, object given in world coordinates:

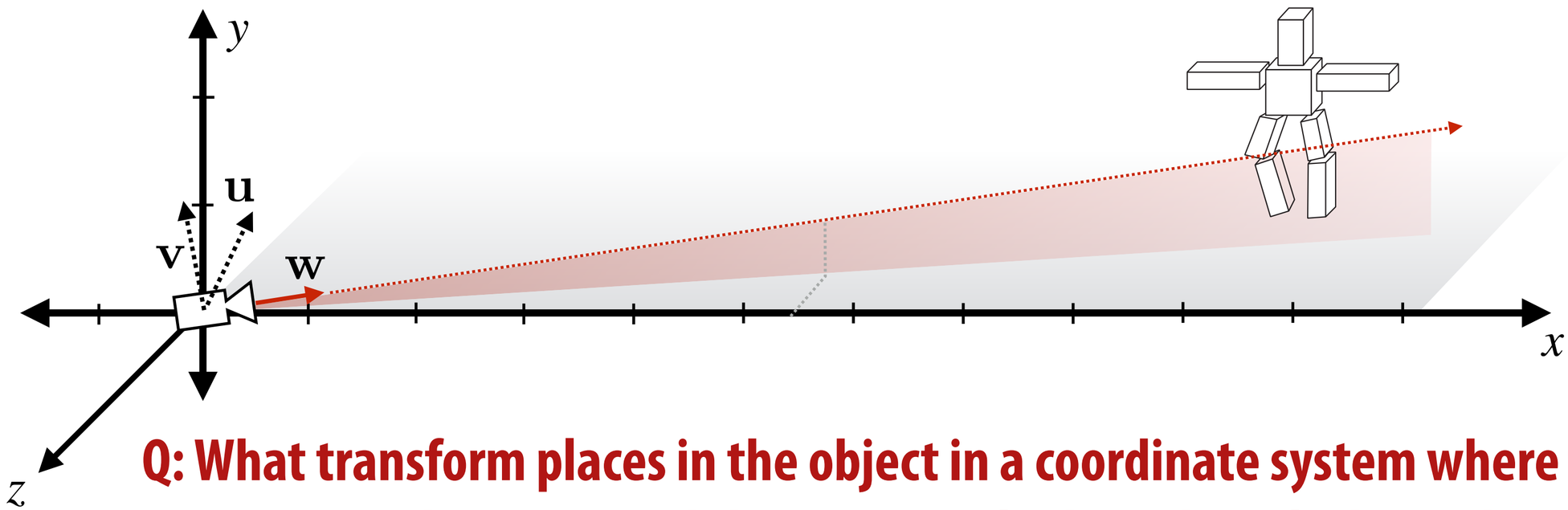


Q: What spatial transformation puts in the object in a coordinate system where the camera is at the origin, looking down the $-z$ axis?

- **Translating object vertex positions by (-4, -2, 0) yields position relative to camera**
- **Rotation about y by $\pi/2$ gives position of object in new coordinate system where camera's view direction is aligned with the $-z$ axis**

Camera looking in a different direction

Now consider a camera looking in a direction $\mathbf{w} \in \mathbb{R}^3$



Q: What transform places in the object in a coordinate system where the camera is at the origin and the camera is looking directly down the -z axis?

- **Construct vectors $\mathbf{u}$, $\mathbf{v}$ orthogonal to $\mathbf{w}$**
  - **e.g., pick an "up" vector $\mathbf{v}$, let $\mathbf{u} := \mathbf{w} \times \mathbf{v}$**
- **Build corresponding rotation matrix**

$$R = \begin{bmatrix} u_x & v_x & -w_x \\ u_y & v_y & -w_y \\ u_z & v_z & -w_z \end{bmatrix}$$

R maps x-axis to $\mathbf{u}$, $\mathbf{y}$-axis to $\mathbf{v}$, $\mathbf{z}$-axis to $-\mathbf{w}$

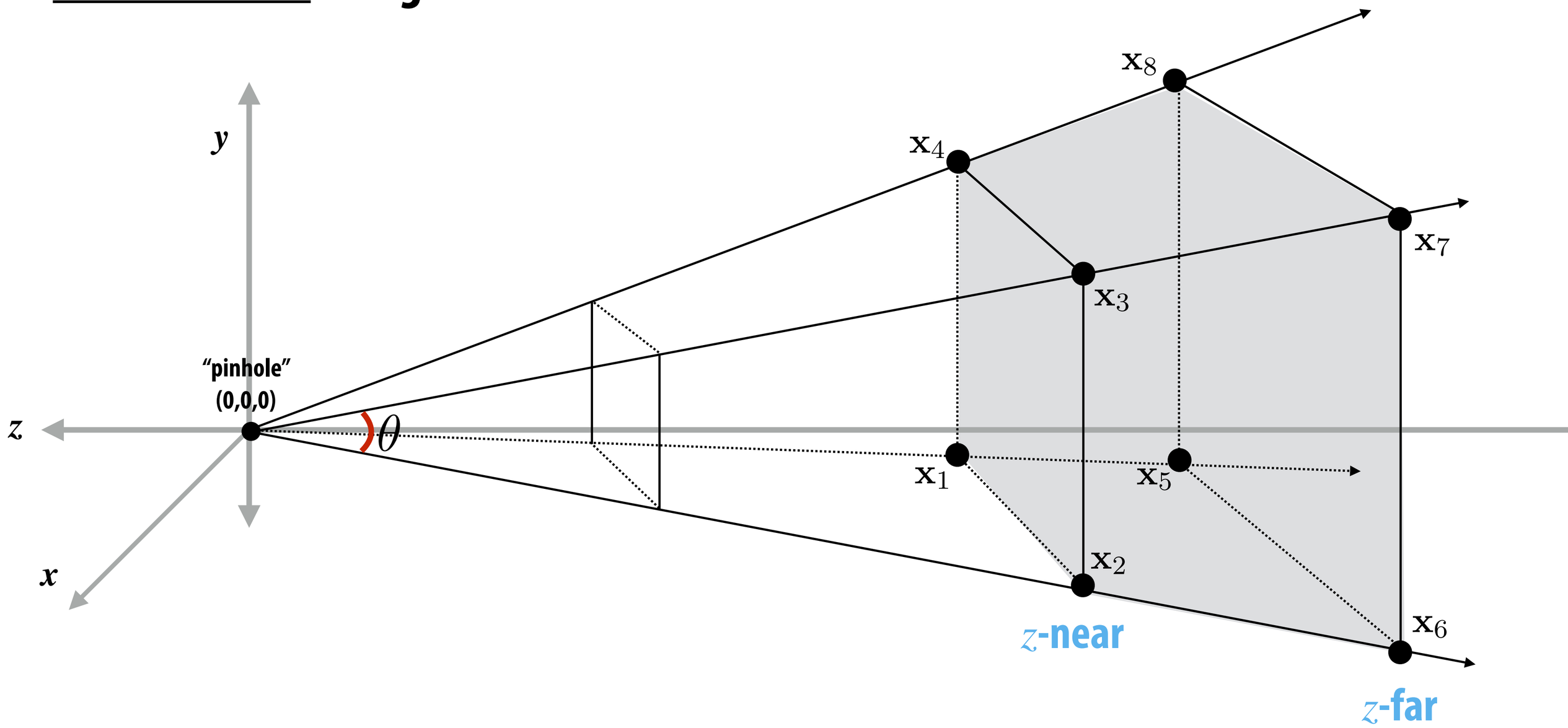
Now invert. (How do we do that?)

$$R^{-1} = R^{\mathsf{T}}$$

$$\begin{bmatrix} u_x & u_y & u_z \\ v_x & v_y & v_z \\ -w_x & -w_y & -w_z \end{bmatrix}$$

View frustum

<u>View frustum</u> is region the camera can see:



- Top / bottom / left / right planes correspond to four sides of the image
- Near / far planes correspond to closest/furthest thing we want to draw

Clipping

- "Clipping" eliminates triangles not visible to the camera / in view frustum
  - Don't waste time rasterizing primitives (e.g., triangles) you can't see!
  - Discarding individual fragments is expensive ("fine granularity")
  - Makes more sense to toss out whole primitives ("coarse granularity")
  - Still need to deal with primitives that are partially clipped…

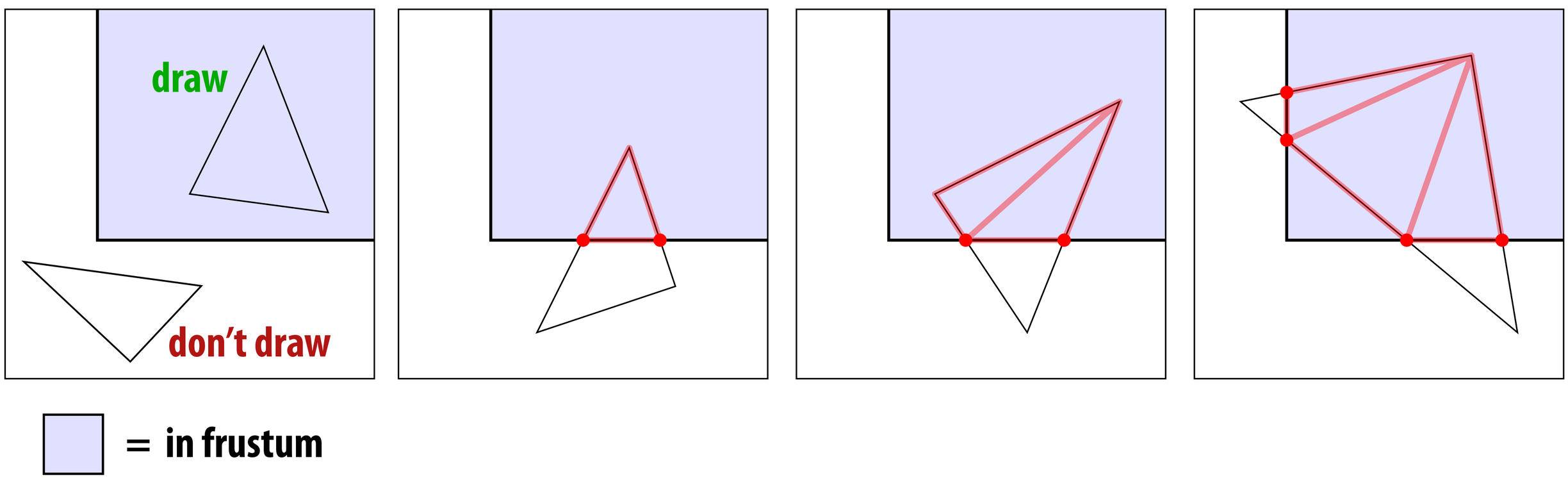
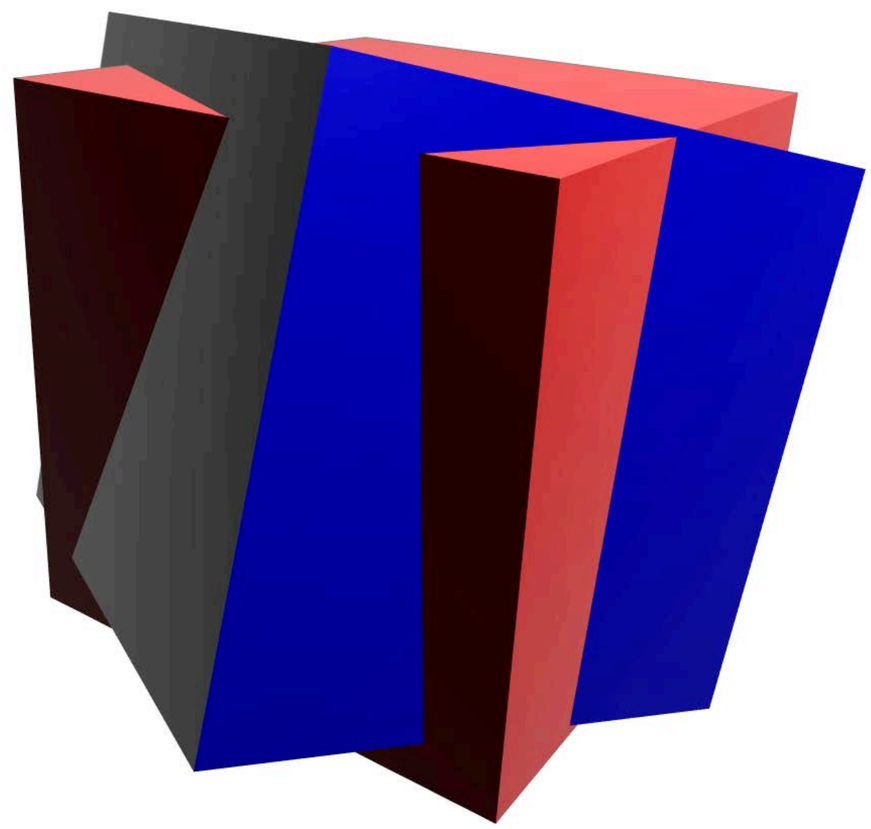


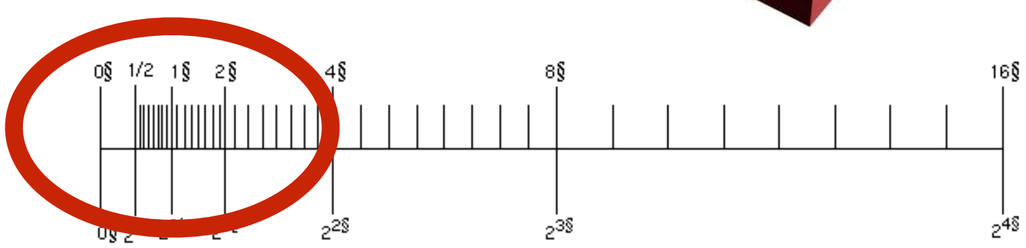
image credit: Jason L. McKesson (https://paroj.github.io/gltut/)

Near/Far Clipping

- Why have *near/far* clipping planes?
  - Some primitives (e.g., triangles) may have vertices both in front & behind eye! (Causes headaches for rasterization, e.g., checking if fragments are behind eye)
  - Also important for dealing with finite precision of depth buffer / limitations on storing depth as floating point values

near = 10^{-1}
far = 10^{3}

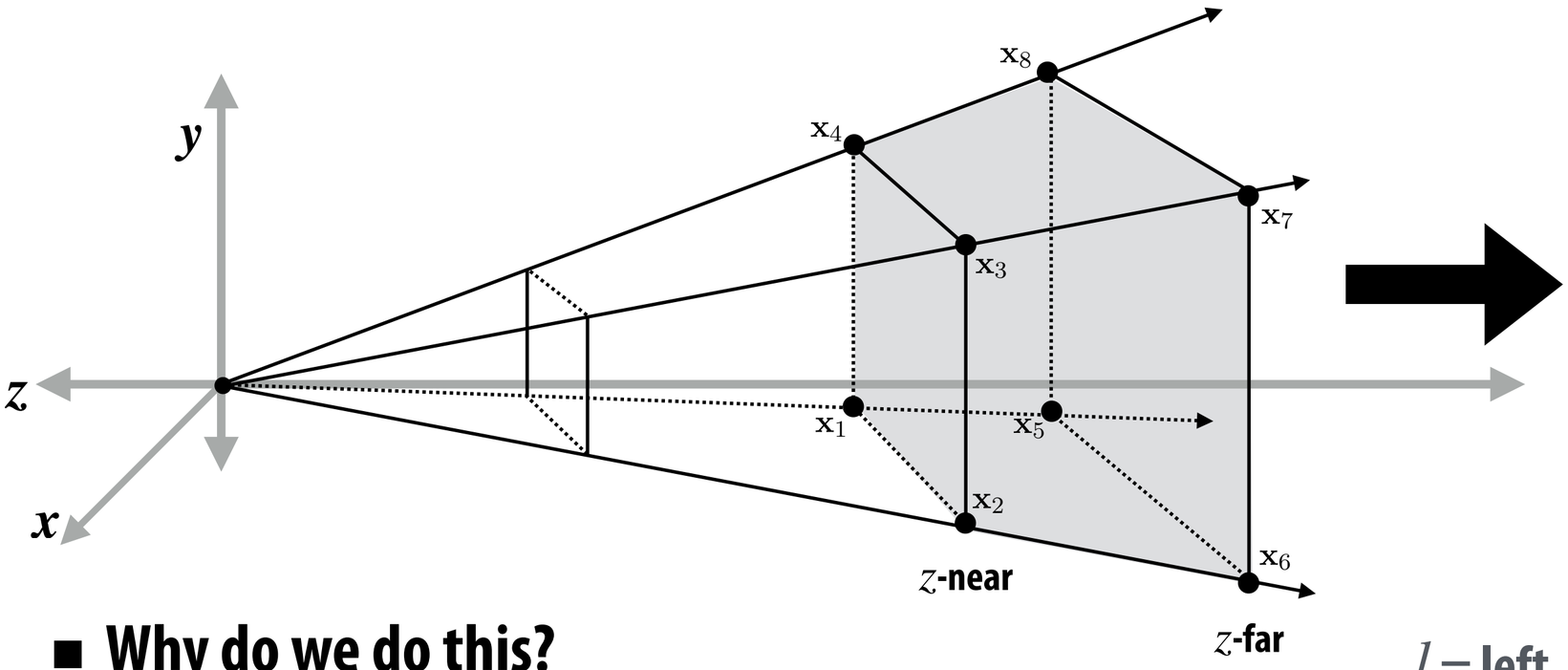


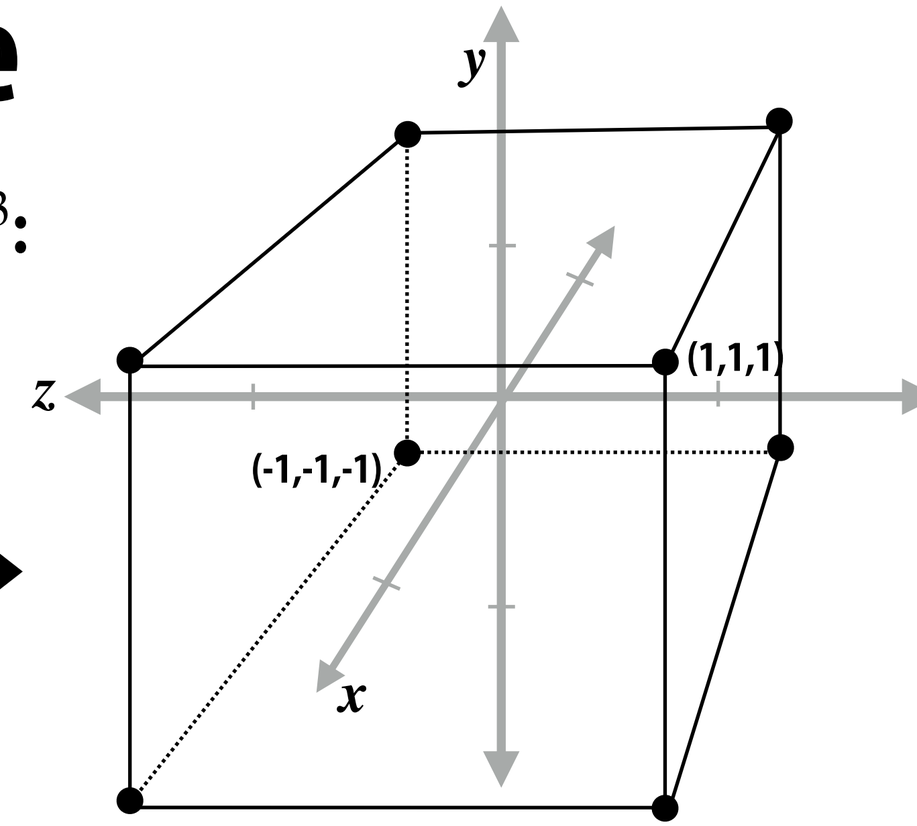


floating point has more "resolution" near zero—hence more precise resolution of primitive-primitive intersection

Mapping frustum to unit cube

Before projecting to 2D, map view frustum to cube $[-1,1]^3$:





- **Why do we do this?**
- **Makes clipping much easier!**
  - **just discard points outside range [-1,1]**
  - **need to think about partially-clipped triangles**
- **Q: How can we express this mapping as a matrix?**
- **A: Solve $A\mathbf{x}_i = \mathbf{y}_i$ for unknown entries of A**

scale to size 2

$$A = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & \frac{l+r}{l-r} \\ 0 & \frac{2}{t-b} & 0 & \frac{b+t}{b-t} \\ 0 & 0 & \frac{2}{n-f} & \frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

translate to origin

l = left $\quad b$ = bottom $\quad n$ = near

r = right $\quad t$ = top $\quad f$ = far

$$\begin{aligned} \mathbf{x}_1 &= \{l,b,n,1\} \\ \mathbf{x}_2 &= \{r,b,n,1\} \\ \mathbf{x}_3 &= \{r,t,n,1\} \\ \mathbf{x}_4 &= \{l,t,n,1\} \\ \mathbf{x}_5 &= \{l,b,f,1\} \\ \mathbf{x}_6 &= \{r,b,f,1\} \\ \mathbf{x}_7 &= \{r,t,f,1\} \\ \mathbf{x}_8 &= \{l,t,f,1\} \end{aligned}$$

$$\begin{aligned} \mathbf{y}_1 &= \{-1,-1,\ \ 1,1\} \\ \mathbf{y}_2 &= \{\ \ 1,-1,\ \ 1,1\} \\ \mathbf{y}_3 &= \{\ \ 1,\ \ 1,\ \ 1,1\} \\ \mathbf{y}_4 &= \{-1,\ \ 1,\ \ 1,1\} \\ \mathbf{y}_5 &= \{-1,-1,-1,1\} \\ \mathbf{y}_6 &= \{\ \ 1,-1,-1,1\} \\ \mathbf{y}_7 &= \{\ \ 1,\ \ 1,-1,1\} \\ \mathbf{y}_8 &= \{-1,\ \ 1,-1,1\} \end{aligned}$$

(orthographic projection)

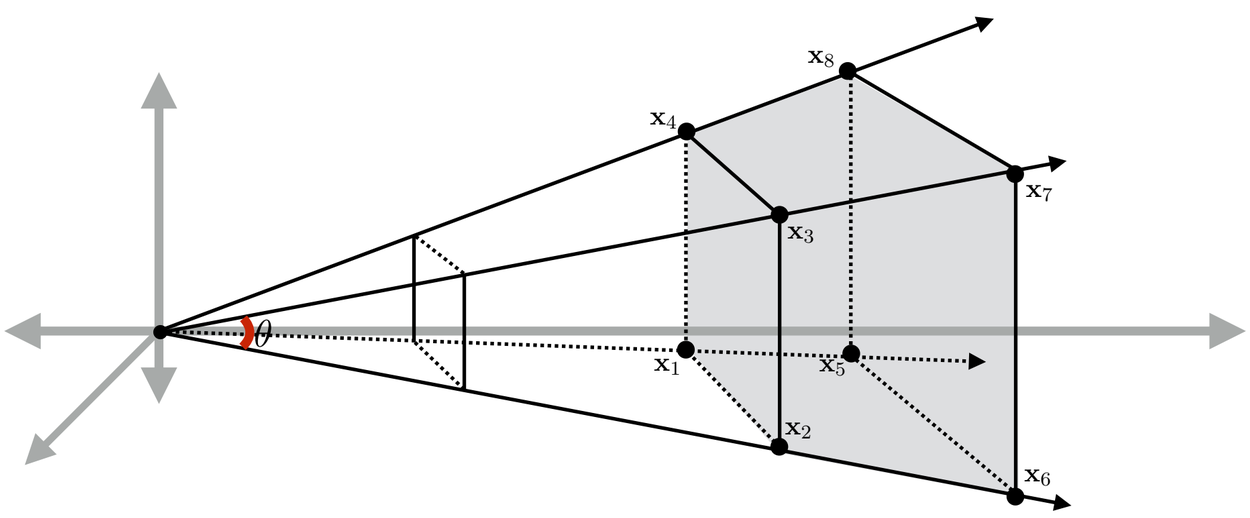
Matrix for Perspective Transform

Recall our basic perspective projection matrix

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

objects shrink in distance

Full perspective matrix takes geometry of view frustum into account:



$$\begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{-(f+n)}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

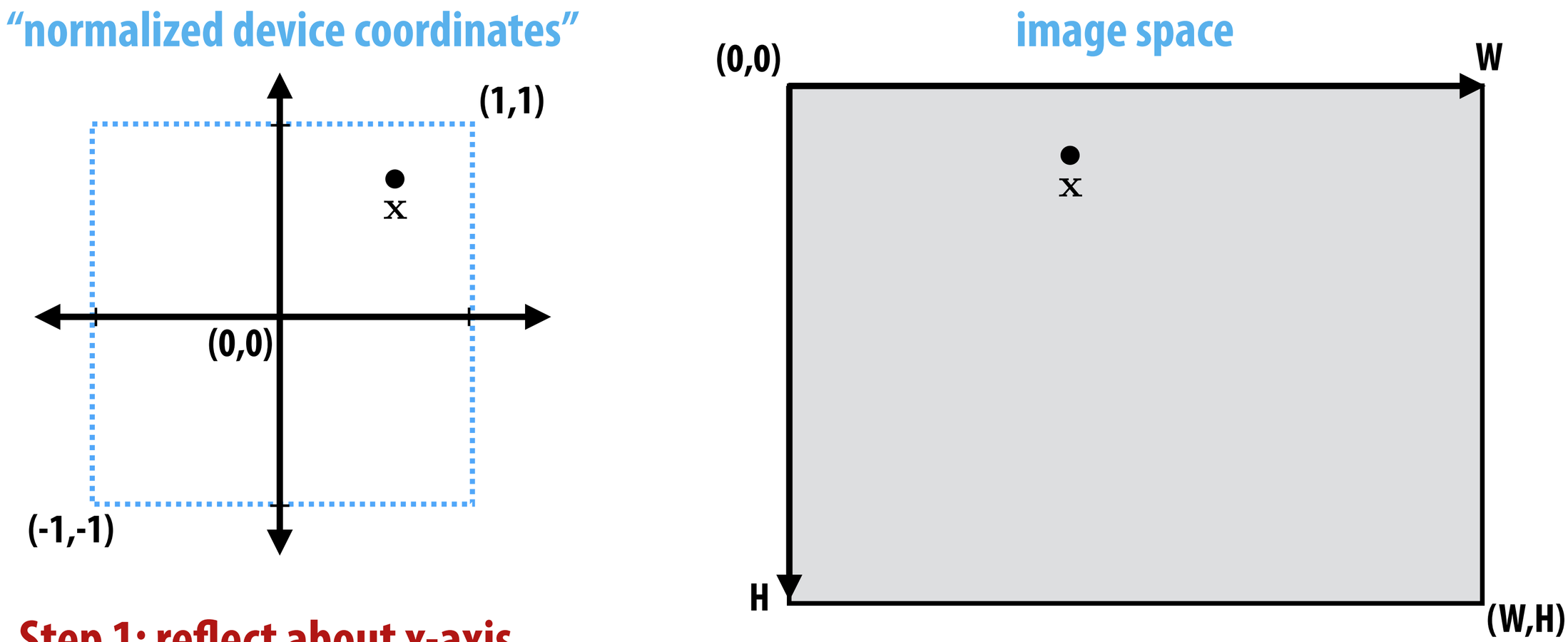
l = **left** b = **bottom** n = **near**

r = **right** t = **top** f = **far**

For a derivation: http://www.songho.ca/opengl/gl_projectionmatrix.html

Review: screen transformation

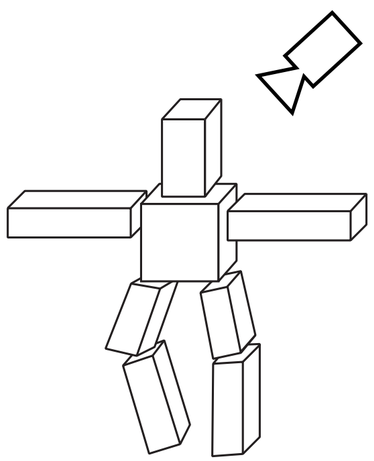
- Had one last transformation in the rasterization pipeline: transform from 2D viewing plane to pixel coordinates
- Projection will take points to [-1,1] x [-1,1] on the z = 1 plane; transform into a W x H pixel image



Step 1: reflect about x-axis
Step 2: translate by (1,1)
Step 3: scale by (W/2,H/2)

Transformations: From Objects to the Screen

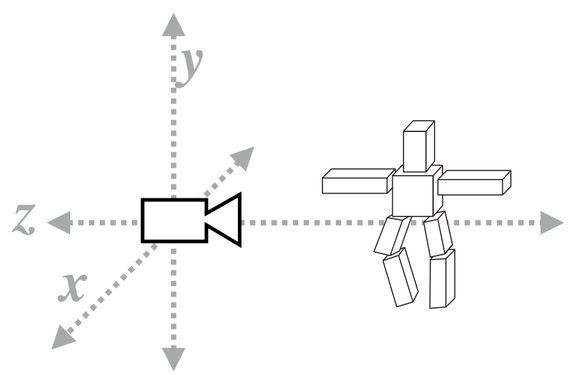
[WORLD COORDINATES]



original description of objects

view transform

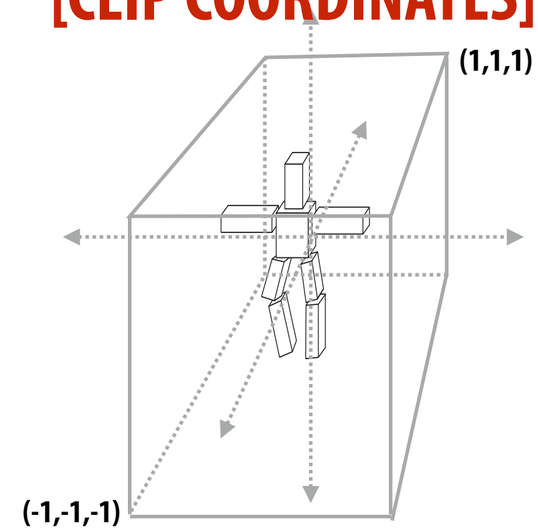
[VIEW COORDINATES]



all positions now expressed relative to camera; camera is sitting at origin looking down -z direction

projection transform

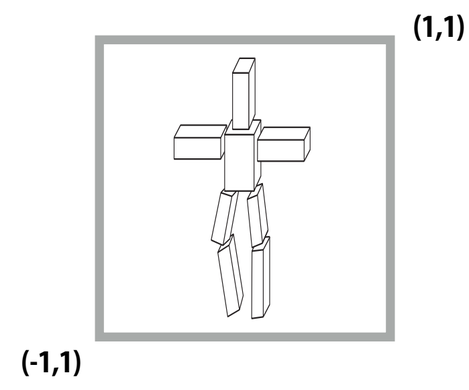
[CLIP COORDINATES]



everything visible to the camera is mapped to unit cube for easy "clipping"

perspective divide

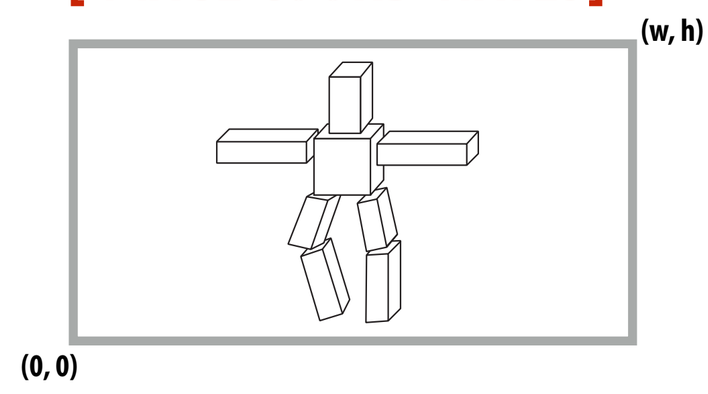
[NORMALIZED COORDINATES]



unit cube mapped to unit square via perspective divide

screen transform

[IMAGE COORDINATES]



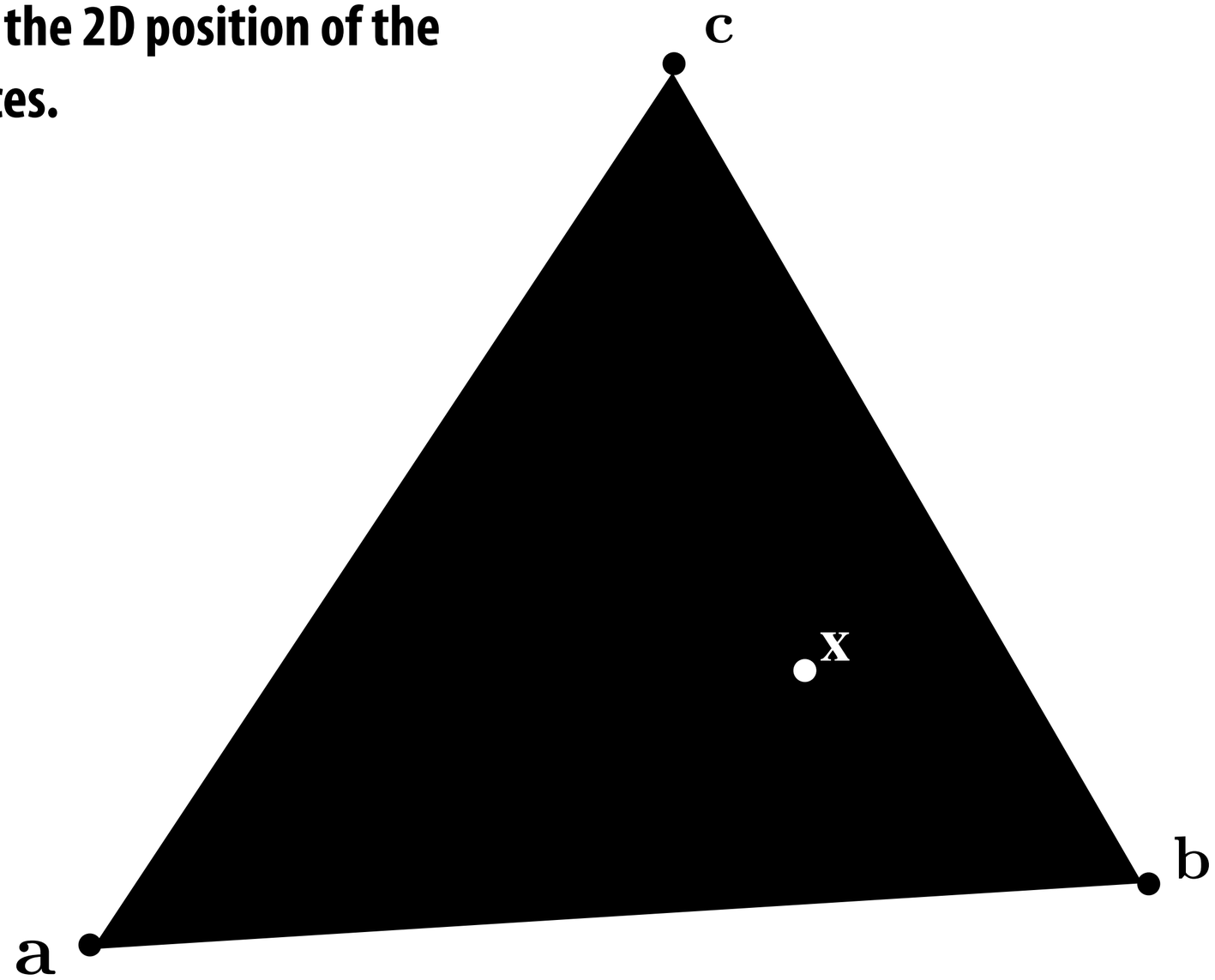
coordinates stretched to match image dimensions (and flipped upside-down)

2D primitives can now be drawn via rasterization

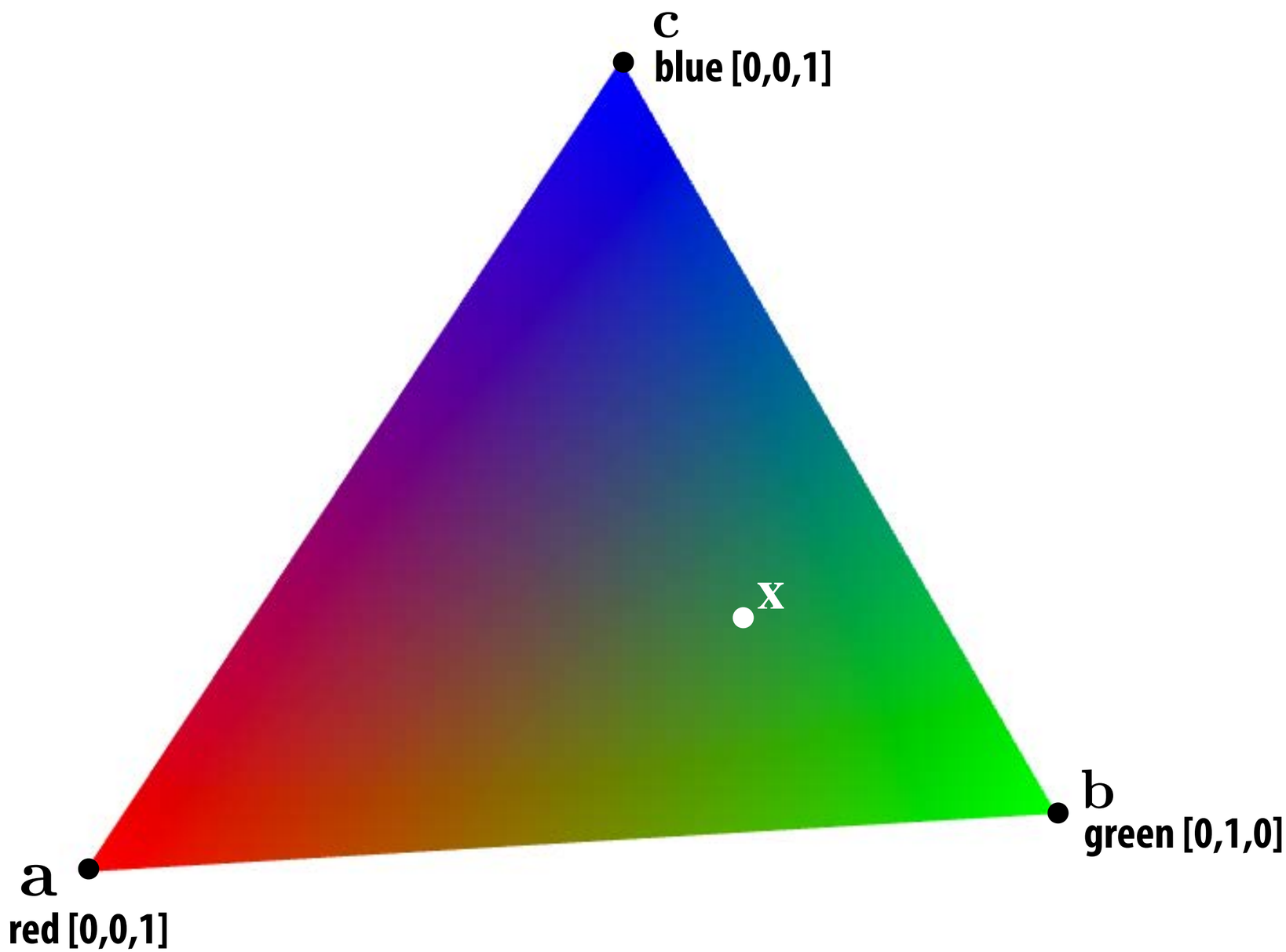
So, how do we draw nice primitives?

Coverage(x,y)

Previously discussed how to sample coverage given the 2D position of the triangle's vertices.



Consider sampling color(x,y)



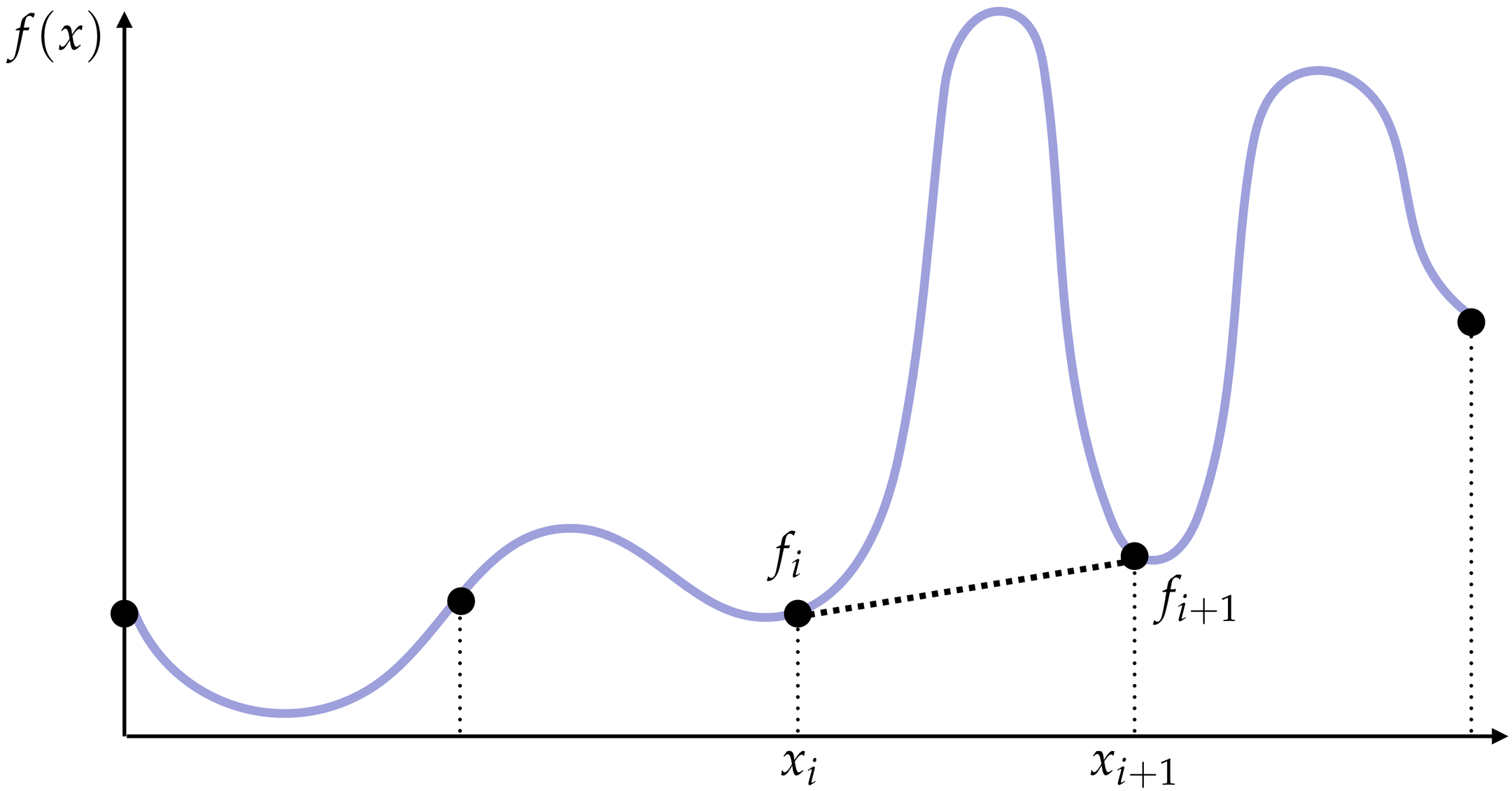
What is the triangle's color at the point x ?

Standard strategy: <u>interpolate</u> color values at vertices.

Linear interpolation in 1D

Suppose we've sampled values of a function f(x) at points x_i, i.e., $f_i := f(x_i)$

Q: How do we construct a function that "connects the dots" between x_i and x_{i+1}?



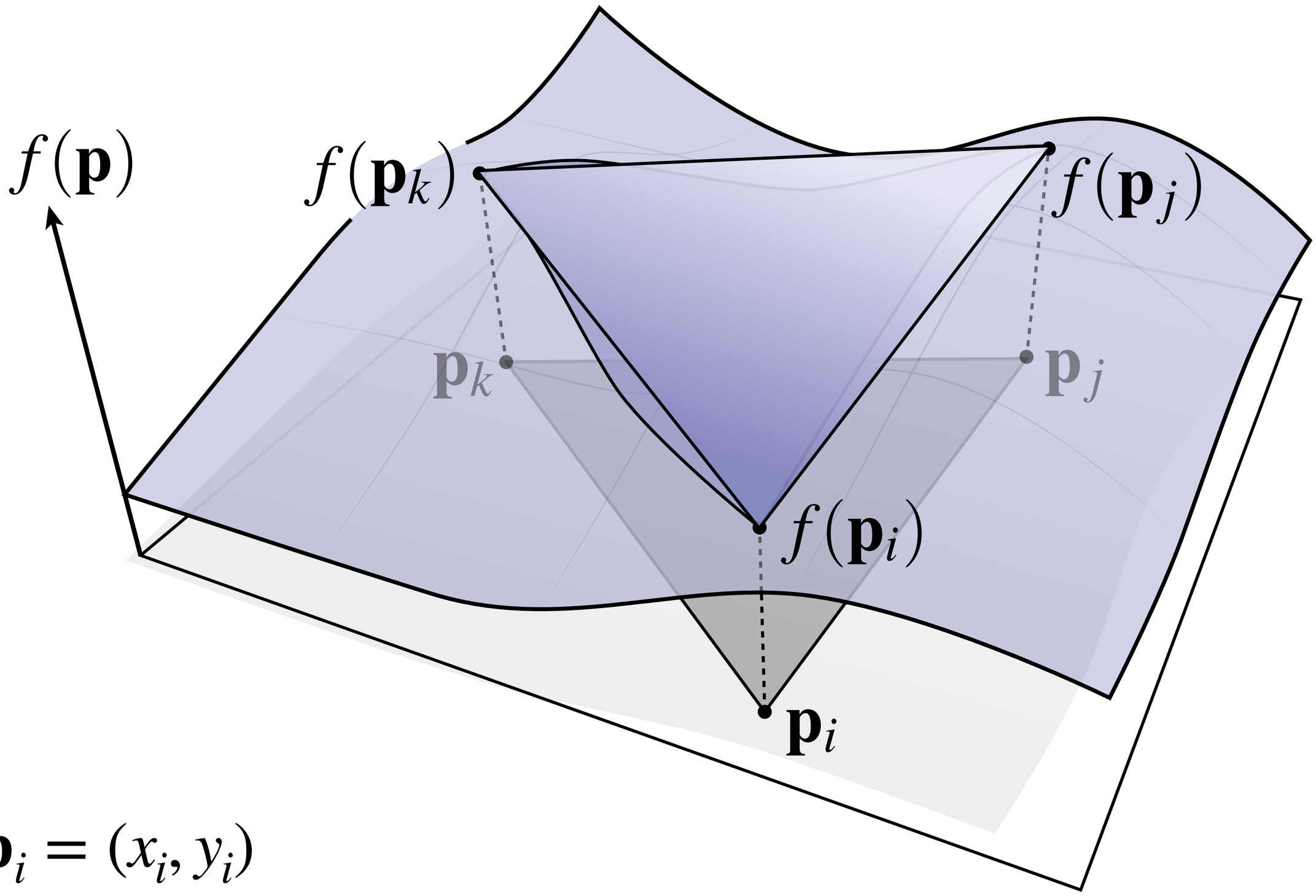
$$t := (x - x_i)/(x_{i+1} - x_i) \in [0, 1]$$

$$\hat{f}(t) = f_i + t(f_{i+1} - f_i) = (1 - t)f_i + tf_{i+1}$$

Linear interpolation in 2D

Suppose we've likewise sampled values of a function $f(\mathbf{p})$ at points $\mathbf{p}_i$, $\mathbf{p}_j$, $\mathbf{p}_k$ in 2D

Q: How do we "connect the dots" this time? E.g., how do we fit a plane?



$$\mathbf{p}_i = (x_i, y_i)$$

Linear interpolation in 2D

- **Want to fit a linear (really, *affine*) function to three values**
- **Any such function has three unknown coefficients a, b, and c:**

$$\hat{f}(x,y) = ax + by + c$$

- **To interpolate, we need to find coefficients such that the function matches the sample values at the sample points:**

$$\hat{f}(x_n, y_n) = f_n,\ n \in \{i, j, k\}$$

- **Yields three linear equations in three unknowns. Solution?**

$$\begin{bmatrix} a \\ b \\ c \end{bmatrix} = \frac{1}{(x_j y_i - x_i y_j) + (x_k y_j - x_j y_k) + (x_i y_k - x_k y_i)} \begin{bmatrix} f_i(y_k - y_j) + f_j(y_i - y_k) + f_k(y_j - y_i) \\ f_i(x_j - x_k) + f_j(x_k - x_i) + f_k(x_i - x_j) \\ f_i(x_k y_j - x_j y_k) + f_j(x_i y_k - x_k y_i) + f_k(x_j y_i - x_i y_j) \end{bmatrix}$$

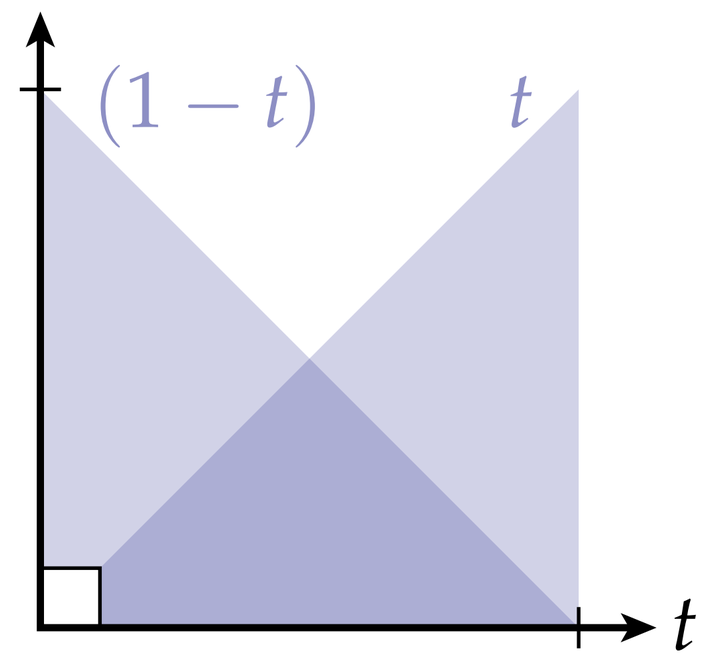
This is ugly. There <u>has</u> to be a better way to think about this…

1D Linear Interpolation, revisited

- Let's think about how we did linear interpolation in 1D:

$$\hat{f}(t) = (1-t)f_i + tf_j$$

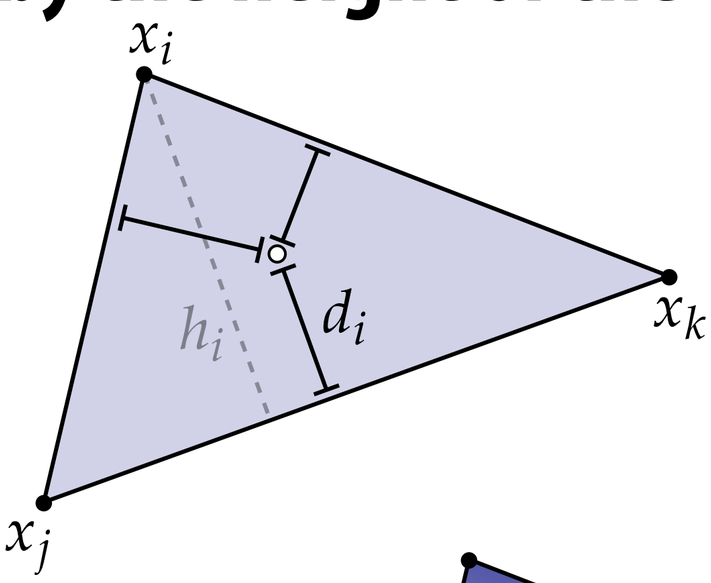
- Can think of this as a linear combination of two functions:



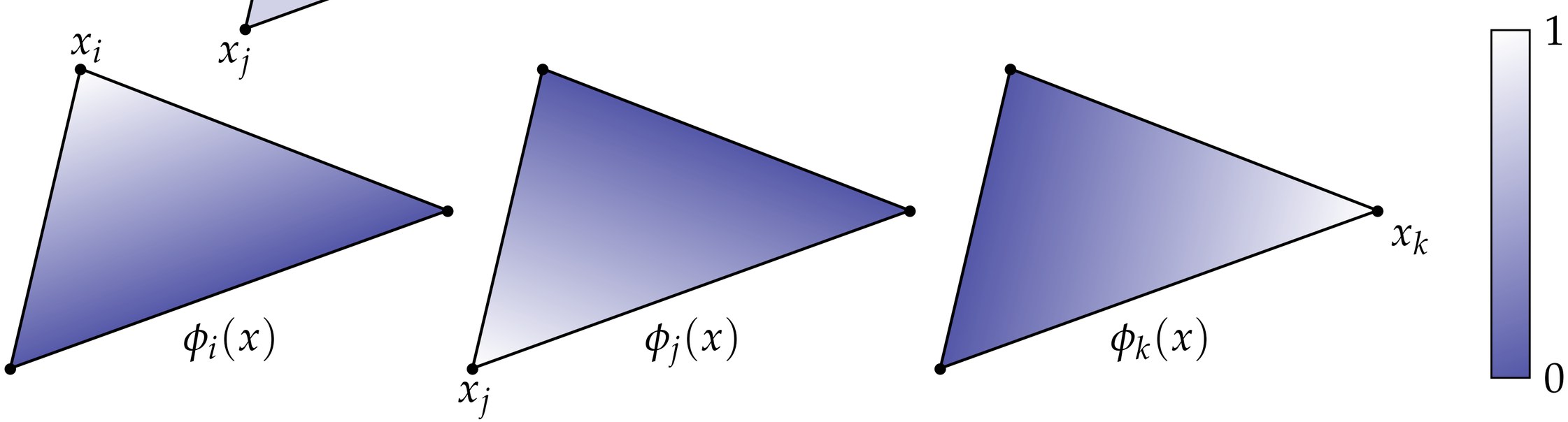
- As we move closer to t=0, we approach the value of f at x_i
- As we move closer to t=1, we approach the value of f at x_j

2D Linear Interpolation, revisited

- We can construct analogous functions for a triangle
- For a given point x, measure the distance to each edge; then divide by the height of the triangle:



$$\phi_i(x) = d_i(x)/h_i$$

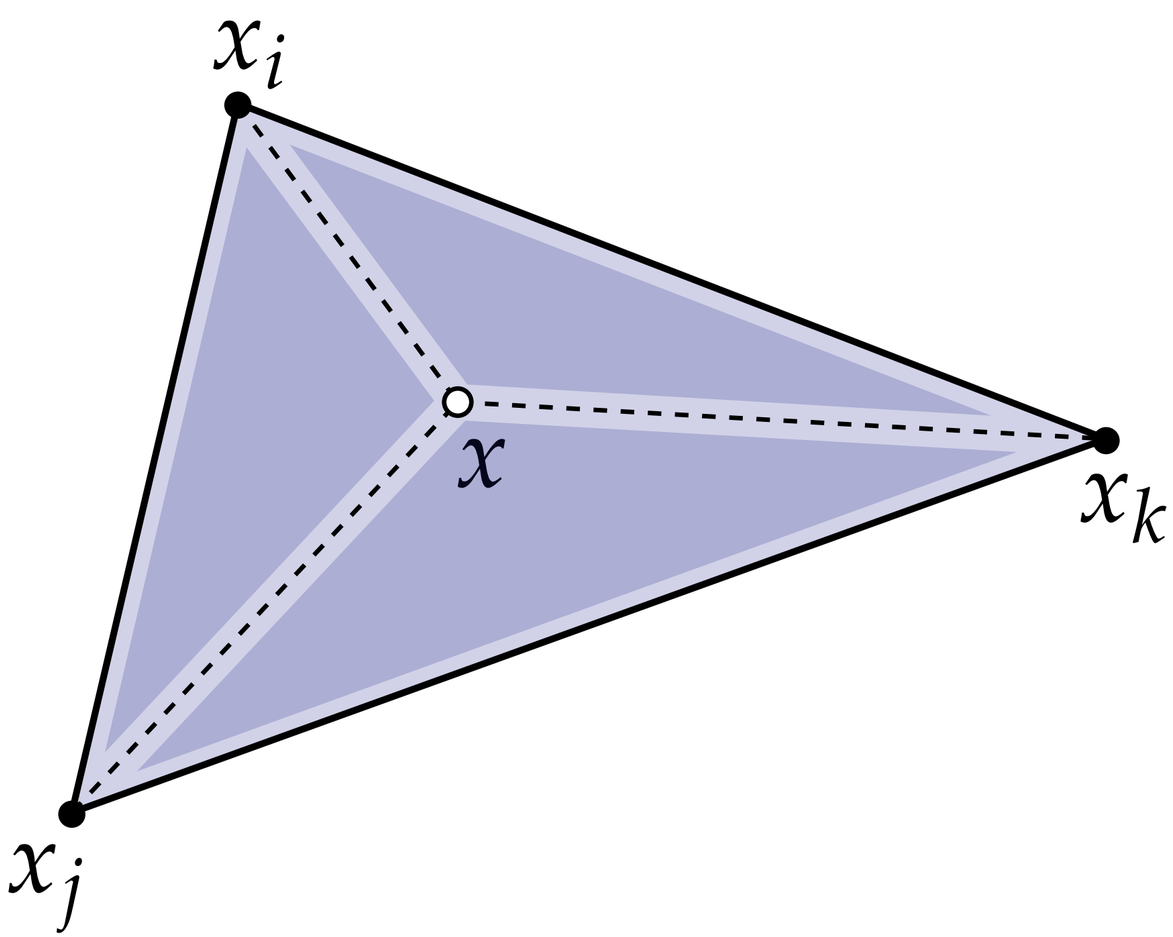


Interpolate by taking linear combination: $\hat{f}(x) = f_i\phi_i + f_j\phi_j + f_k\phi_k$

Q: Is this the same as the (ugly) function we found before?

2D Interpolation, another way

- **I claim we can *also* get the same three basis functions as a ratio of triangle areas:**



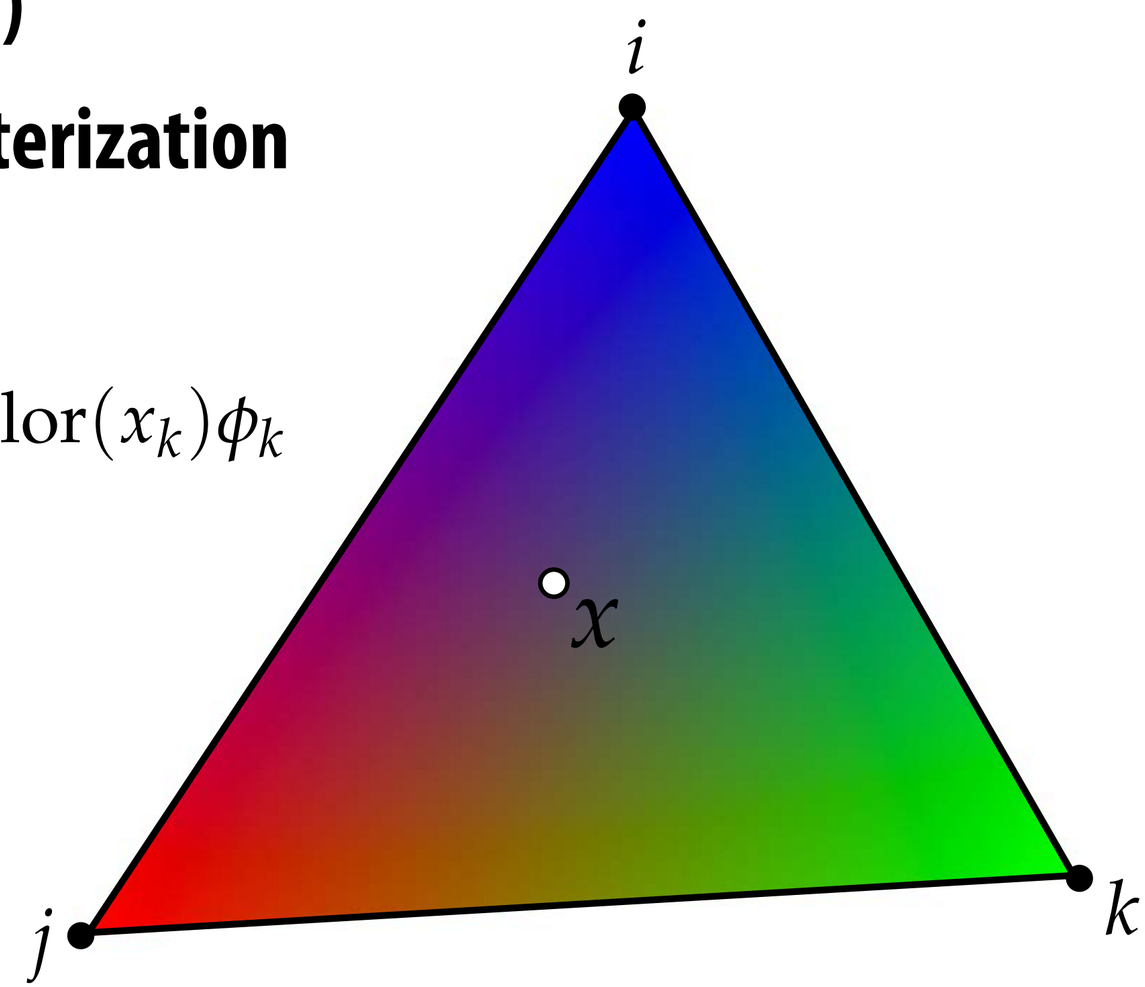
$$\phi_i(x) = \frac{\text{area}(x, x_j, x_k)}{\text{area}(x_i, x_j, x_k)}$$

Q: Do you buy it? (Why or why not?)

Barycentric Coordinates

- No matter how you compute them, the values of the three functions $\phi_i(\mathbf{x}), \phi_j(\mathbf{x}), \phi_k(\mathbf{x})$ for a given point are called <u>barycentric coordinates</u>
- Can be used to interpolate any attribute associated with vertices. (color*, texture coordinates, etc.)
- Importantly, these same three values fall out of the half-plane tests used for triangle rasterization! (Why?)
- Hence, get them for "free" during rasterization

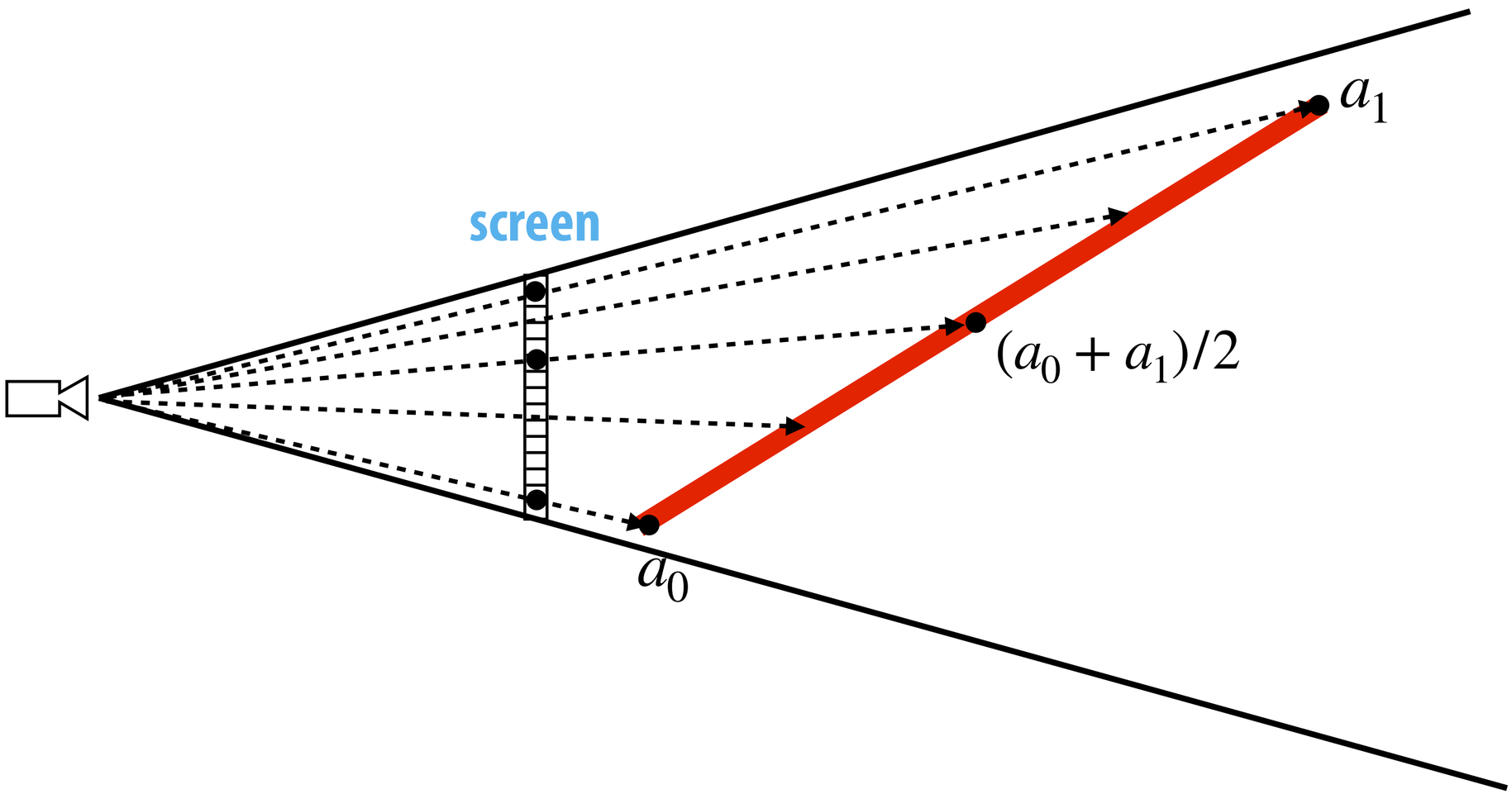
$$\text{color}(x) = \text{color}(x_i)\phi_i + \text{color}(x_j)\phi_j + \text{color}(x_k)\phi_k$$



*Note: we haven't explained yet how to encode colors as numbers! We'll talk about that in a later lecture…

Perspective-incorrect interpolation

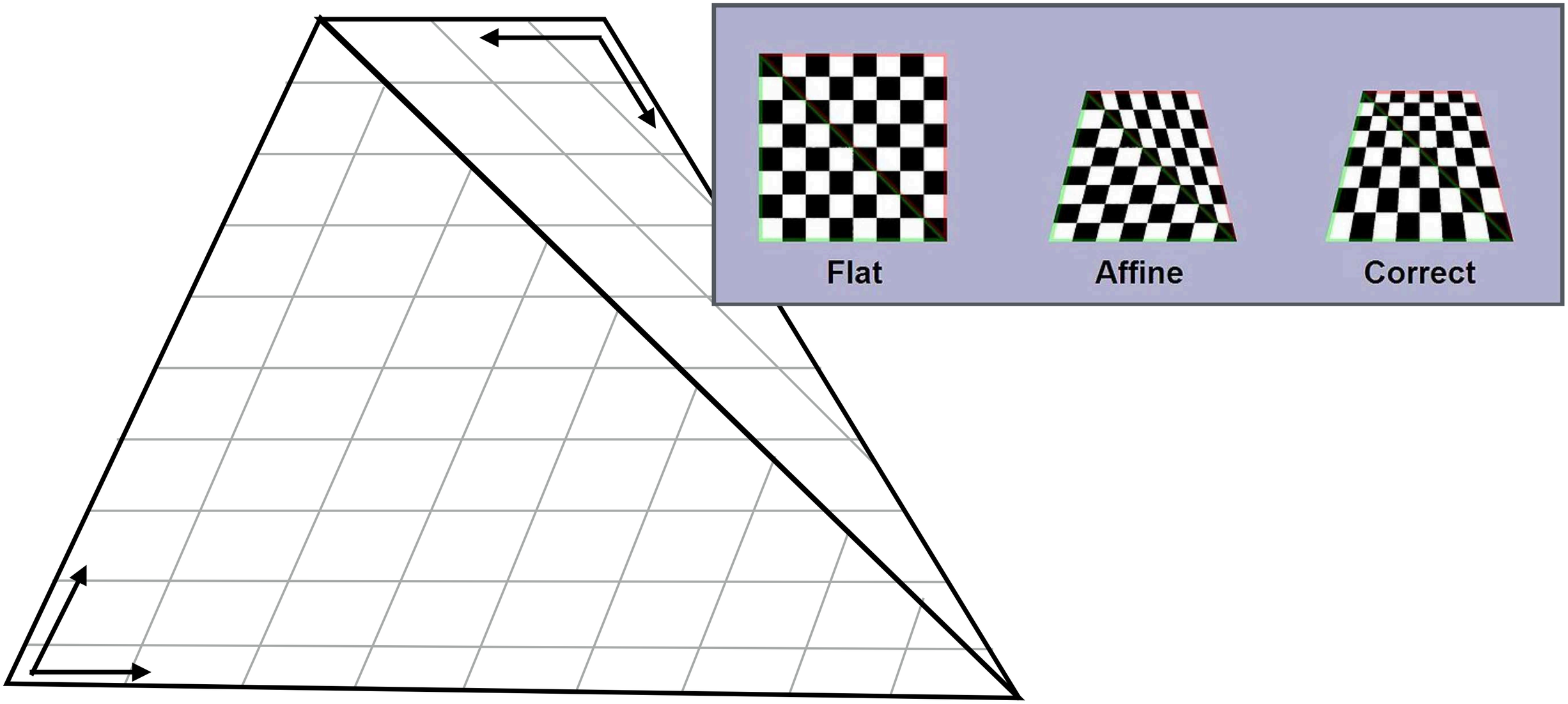
Due to perspective projection (homogeneous divide), barycentric interpolation of values on a triangle with different depths <u>is not</u> an affine function of screen XY coordinates



Want to interpolate attribute values linearly in <u>3D object space</u>, not image space.

Example: perspective *incorrect* interpolation

Consider a quadrilateral split into two triangles:



If we compute barycentric coordinates using 2D (projected) coordinates, leads to (derivative) discontinuity in interpolation where quad was split

Perspective Correct Interpolation

- Goal: interpolate some attribute ϕ at vertices
- Basic recipe:
  - Compute depth z at each vertex
  - Evaluate $Z := 1/z$ and $P := \phi/z$ at each vertex
  - Interpolate Z and P using standard (2D) barycentric coords
  - At each *fragment*, divide interpolated P by interpolated Z to get final value

For a derivation, see Low, "Perspective-Correct Interpolation"

Texture Mapping

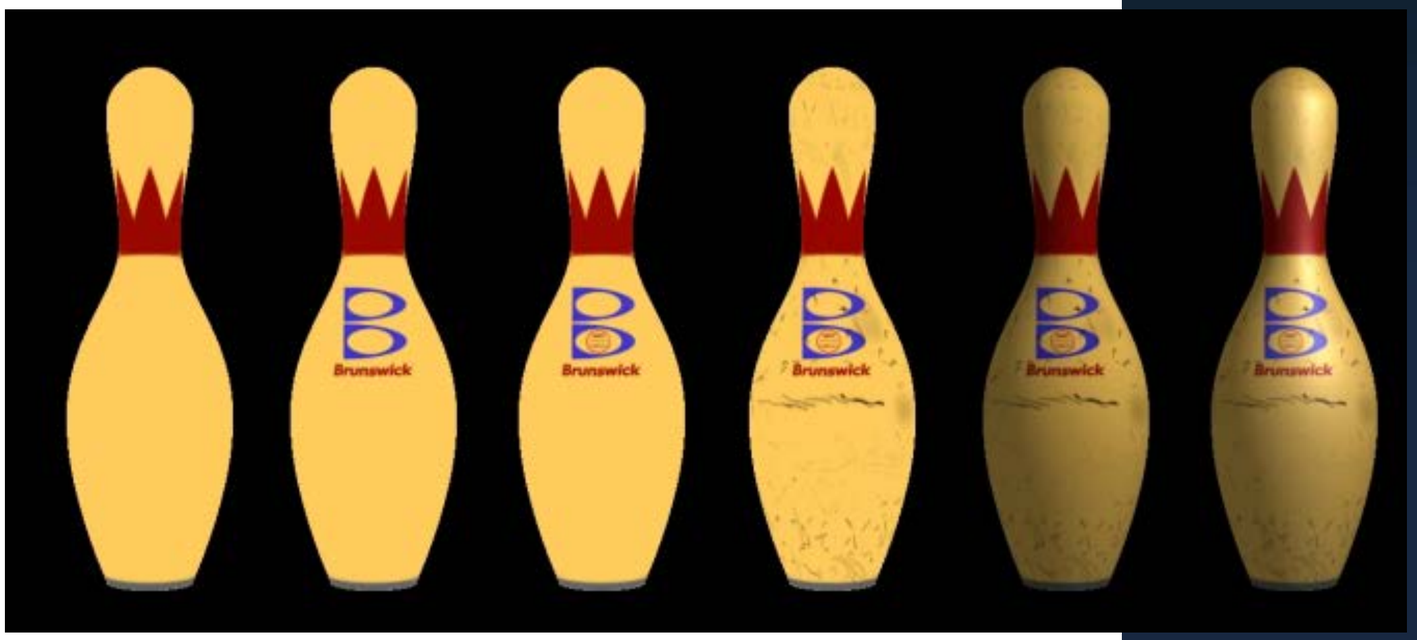
Many uses of texture mapping

Define variation in surface reflectance

Pattern on ball

Wood grain on floor

Describe surface material properties



Multiple layers of texture maps for color, logos, scratches, etc.

Normal & Displacement Mapping

normal mapping

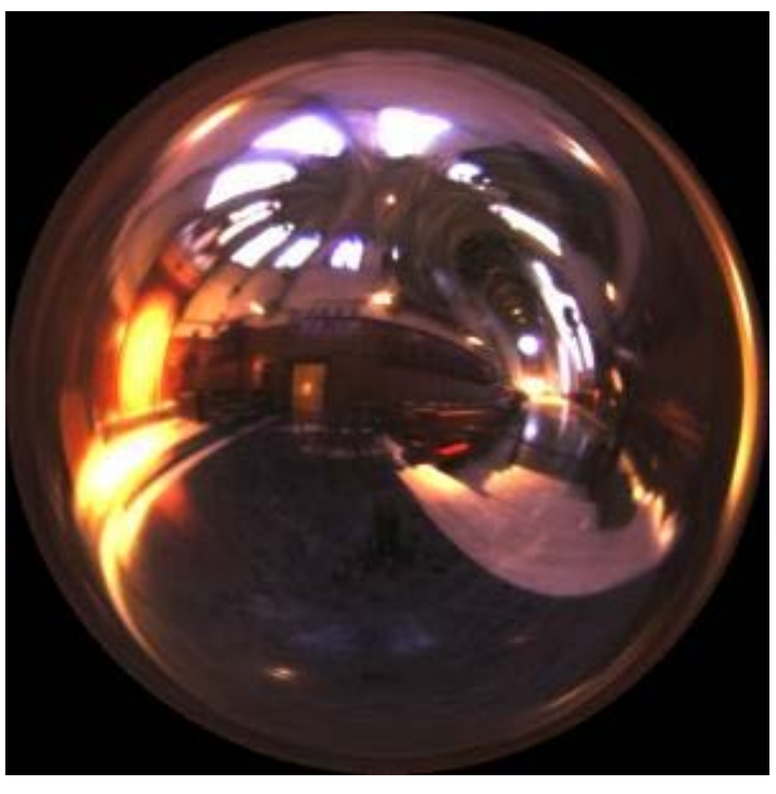
Use texture value to perturb surface normal to "fake" appearance of a bumpy surface

displacement mapping

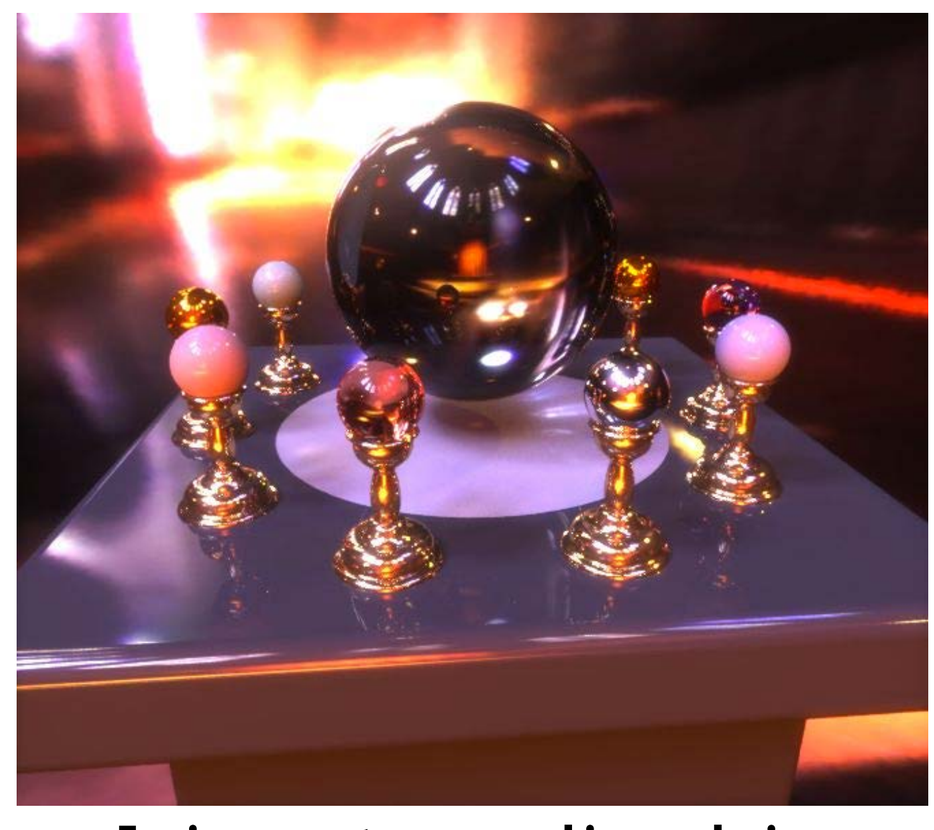
dice up surface geometry into tiny triangles & offset positions according to texture values (note bumpy silhouette and shadow boundary)

Represent precomputed lighting and shadows

Original model | With ambient occlusion | Extracted ambient occlusion map



Grace Cathedral environment map

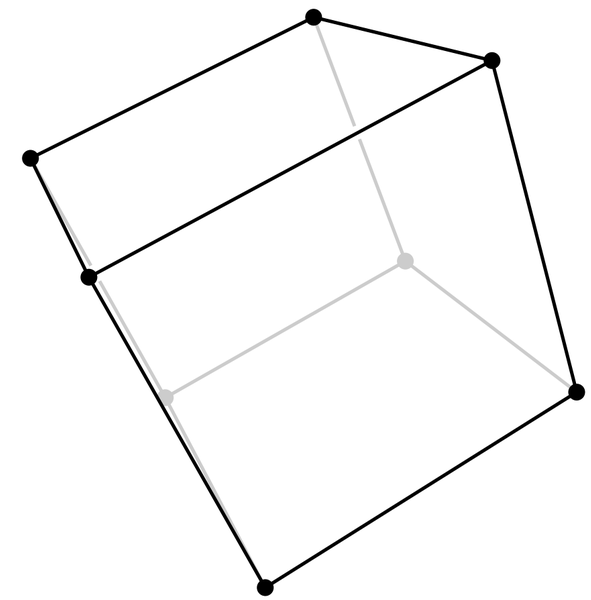


Environment map used in rendering

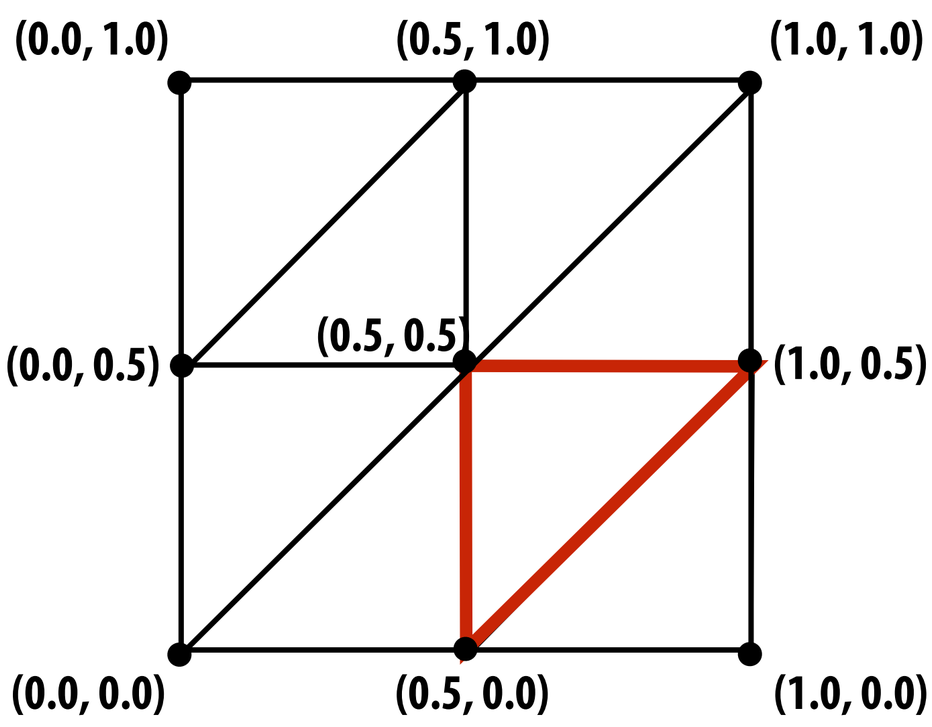
Texture coordinates

- "Texture coordinates" define a mapping from surface coordinates to points in texture domain
- Often defined by linearly interpolating texture coordinates at triangle vertices

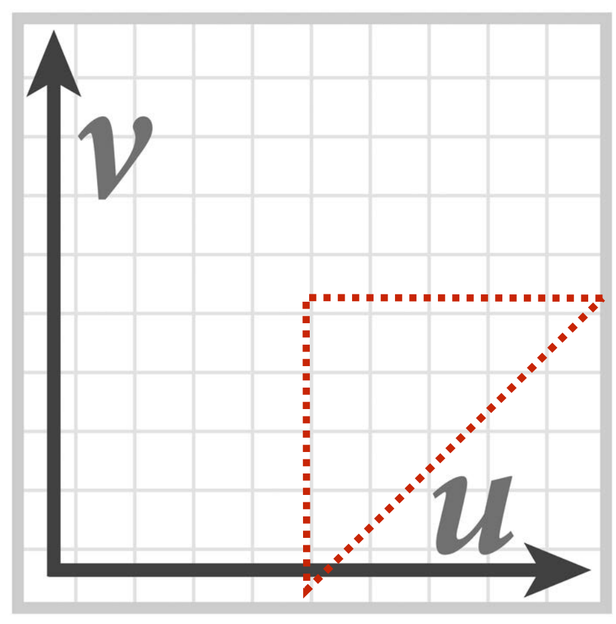
example: texture this cube



Suppose each cube face is split into eight triangles, with texture coordinates (u,v) at each vertex

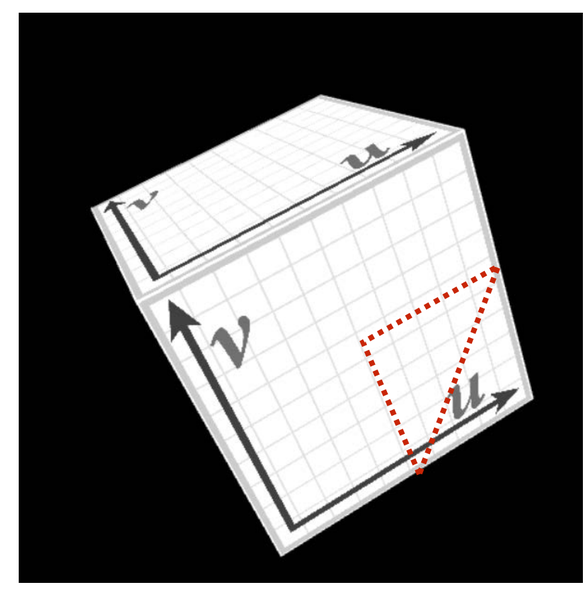


A texture on the $[0,1]^2$ domain can be specified by a 2048x2048 image



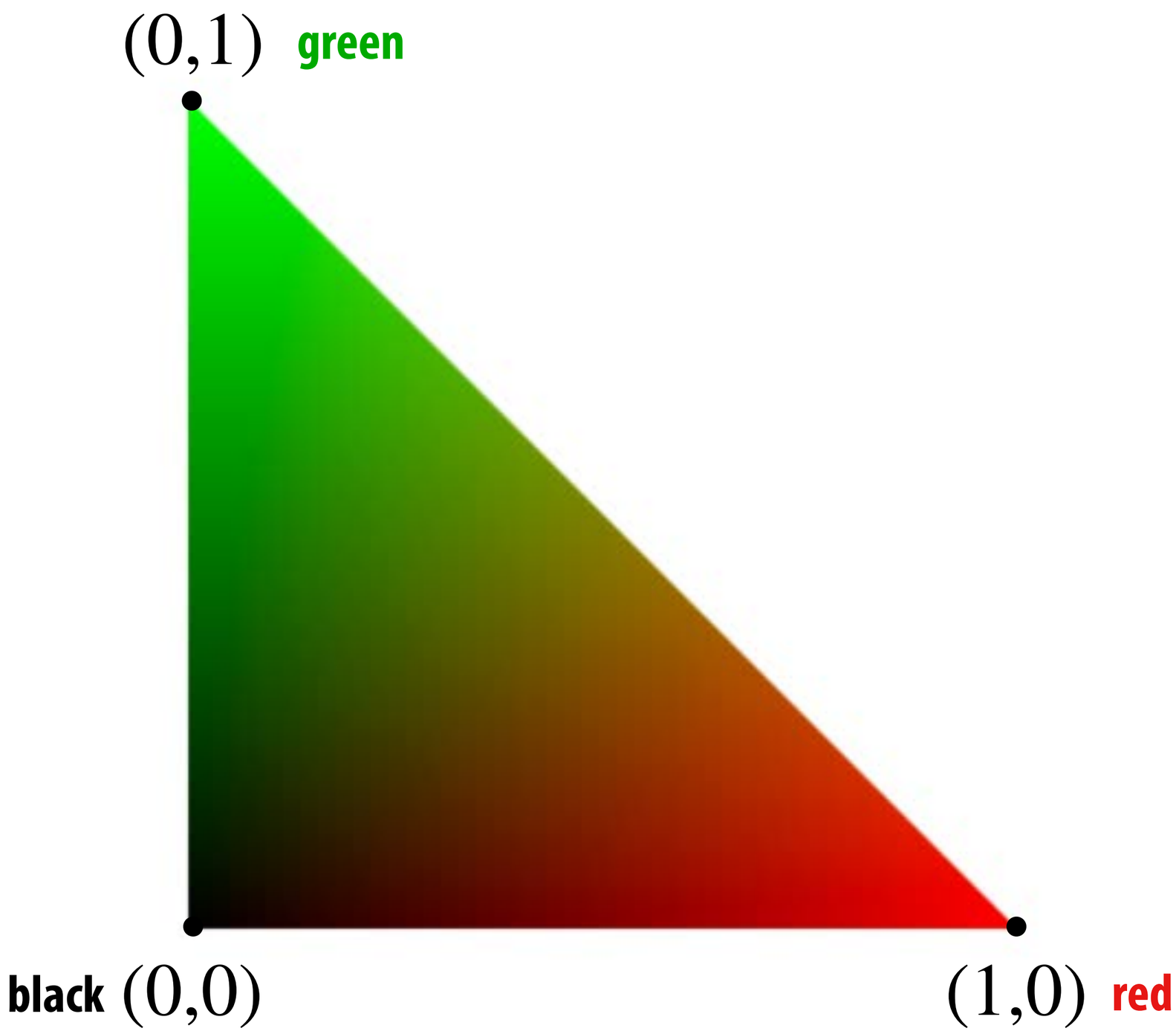
(location of highlighted triangle in texture space shown in red)

Linearly interpolating texture coordinates & "looking up" color in texture gives this image:



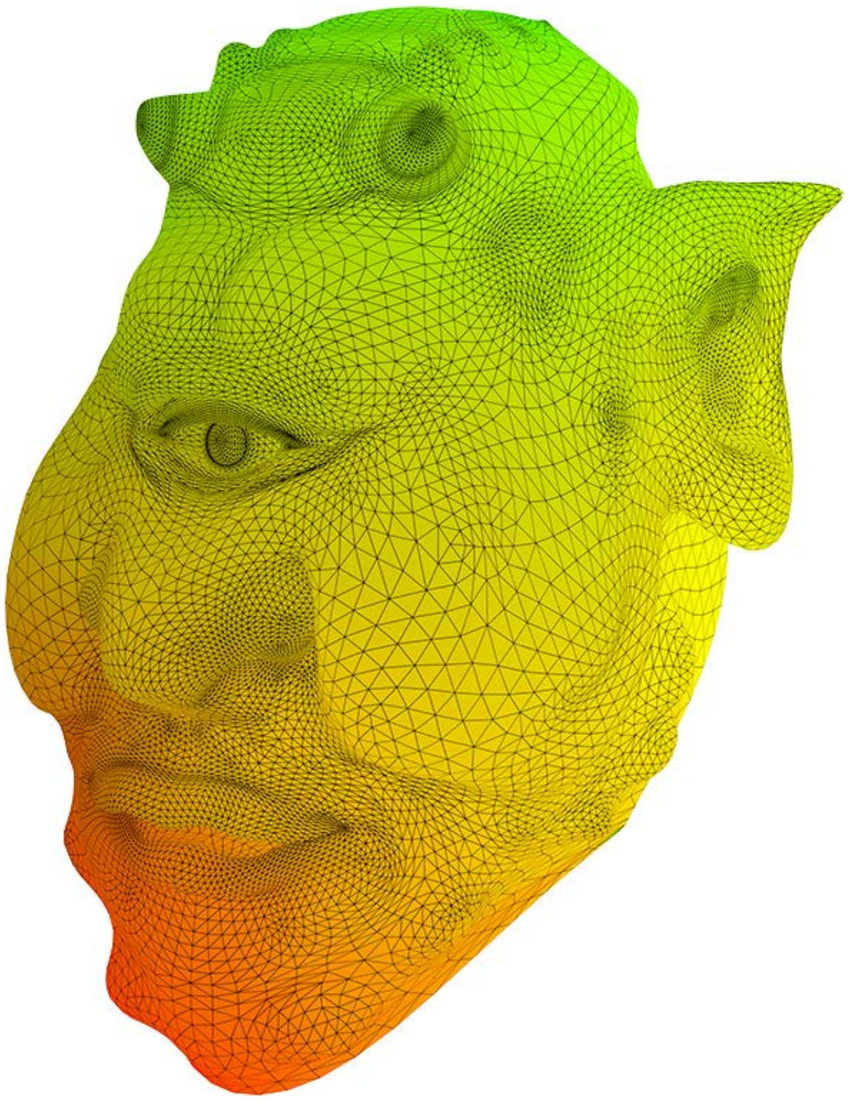
Visualization of texture coordinates

Associating texture coordinates (u, v) with colors helps to visualize mapping

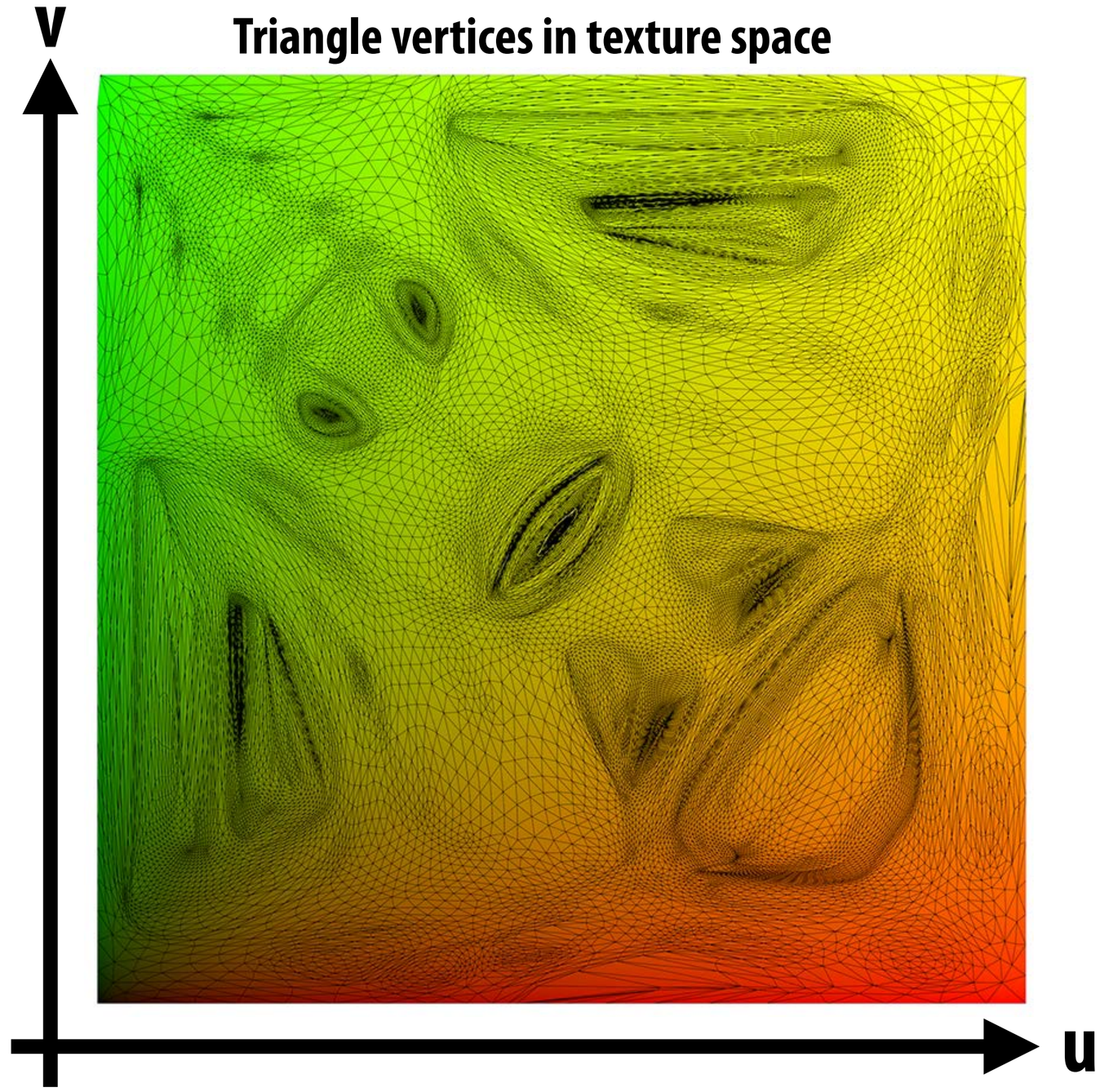


More complex mapping

Visualization of texture coordinates



Triangle vertices in texture space

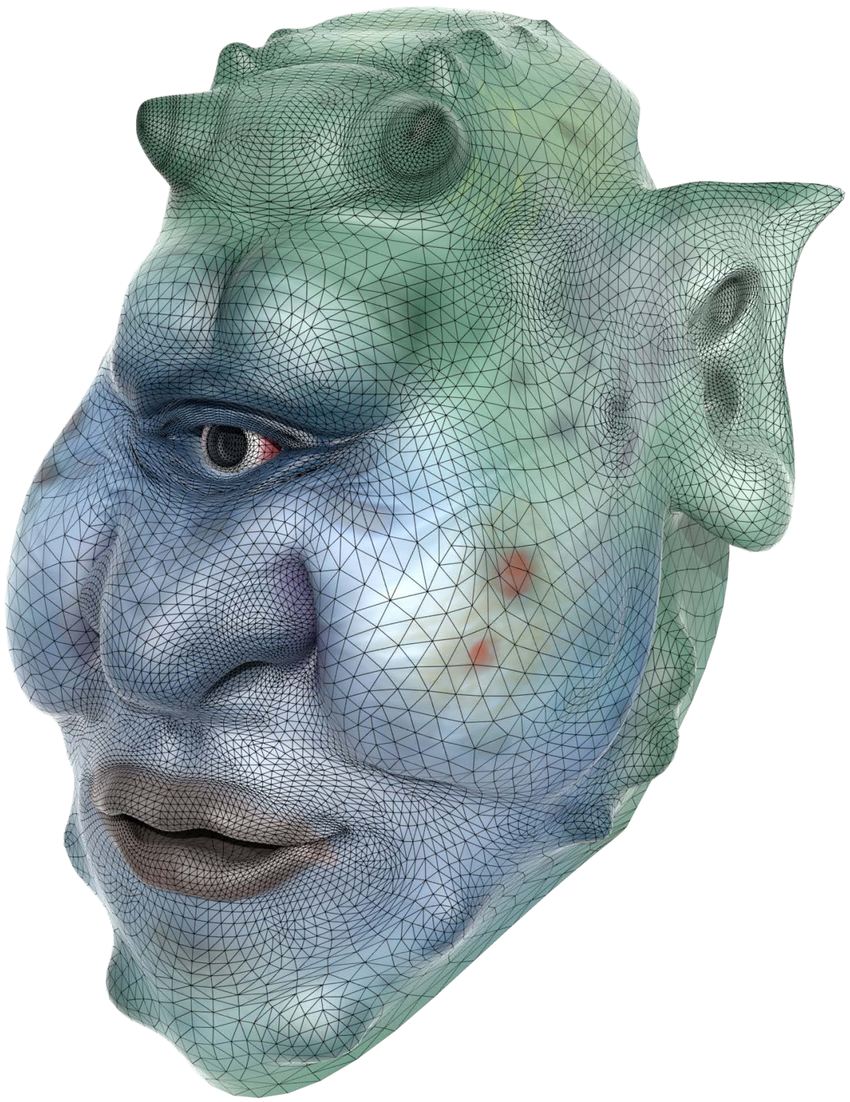


Each vertex has a coordinate (u,v) in texture space

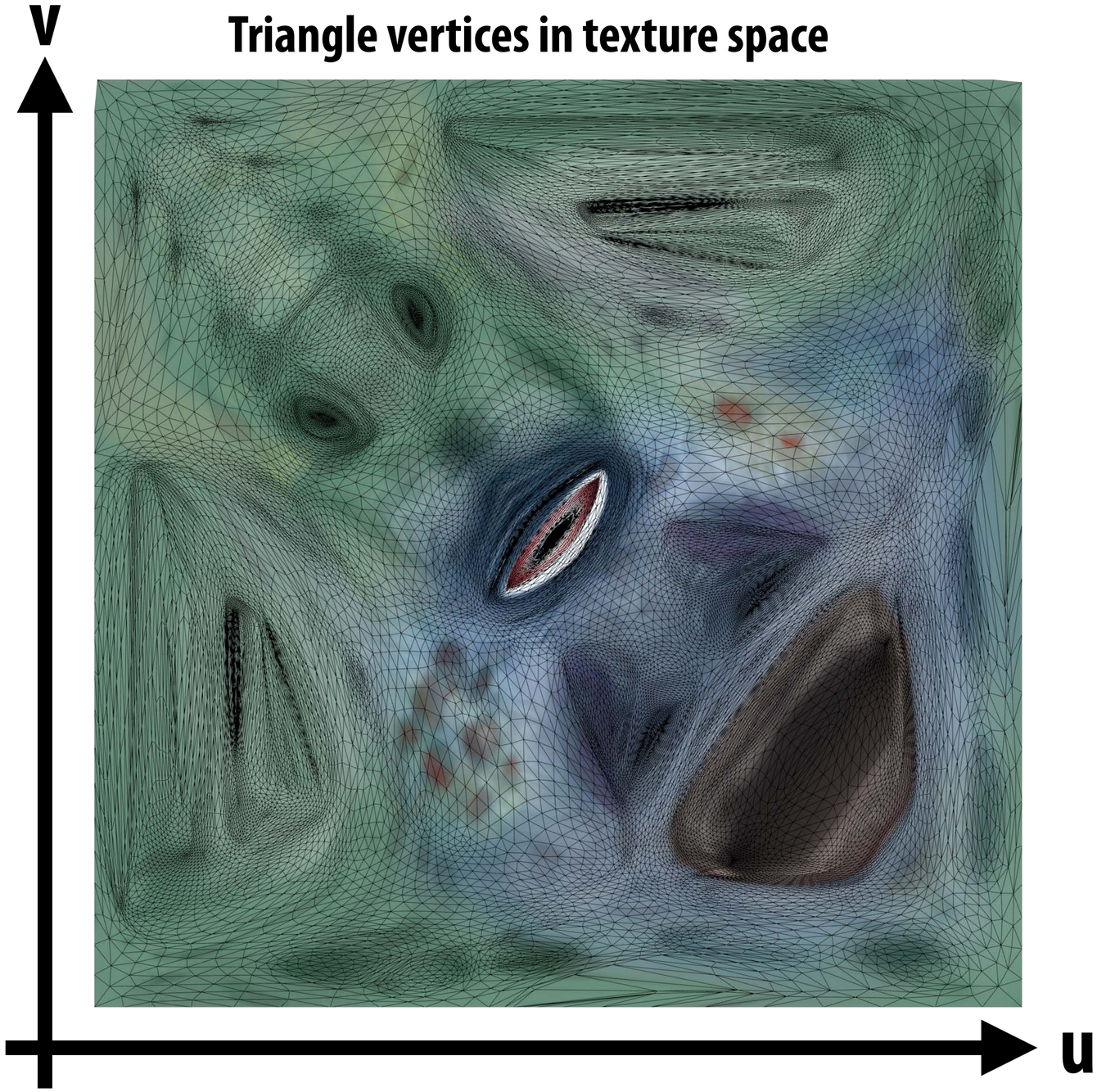
(Actually coming up with these coordinates is another story!)

Texture mapping adds detail

Rendered result



Triangle vertices in texture space



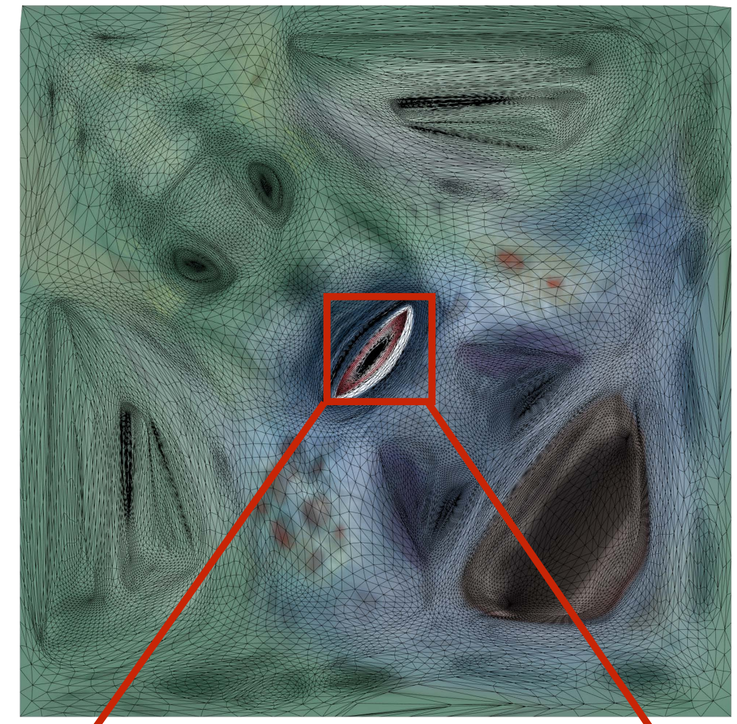
Each triangle "copies" a piece of the image back to the surface

Texture mapping adds detail

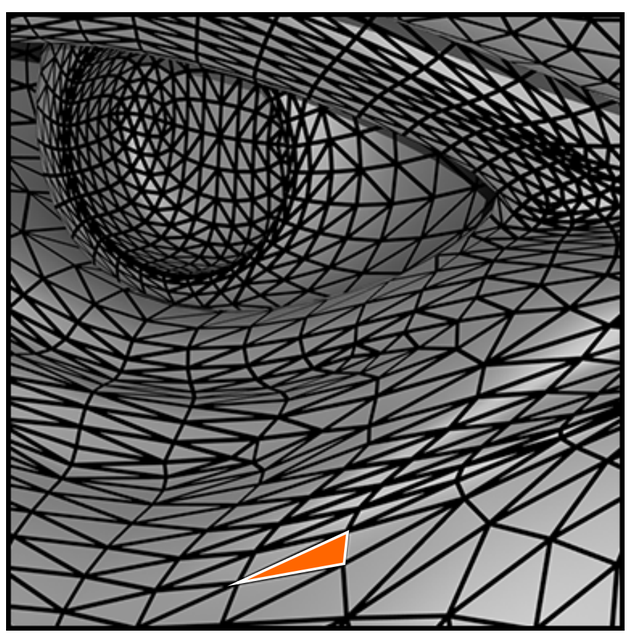
rendering without texture

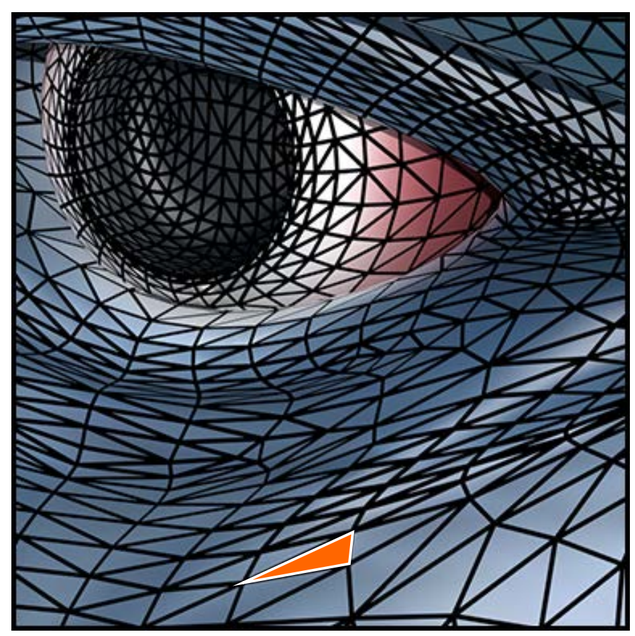
rendering with texture

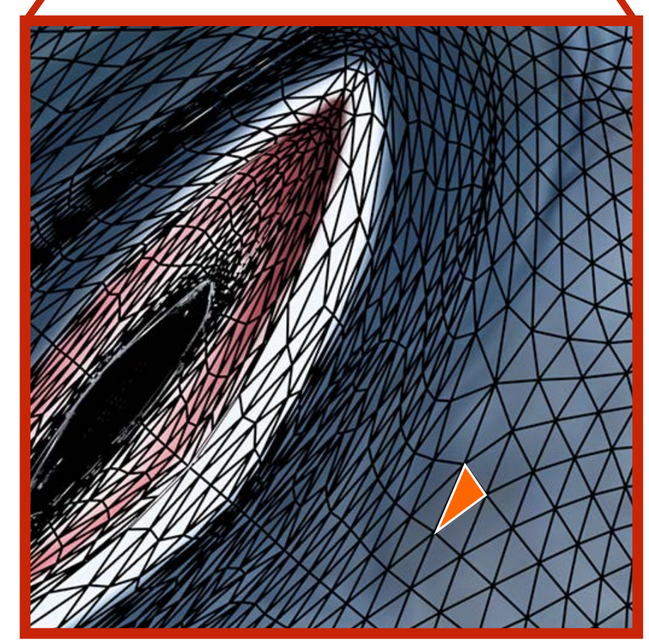
texture image



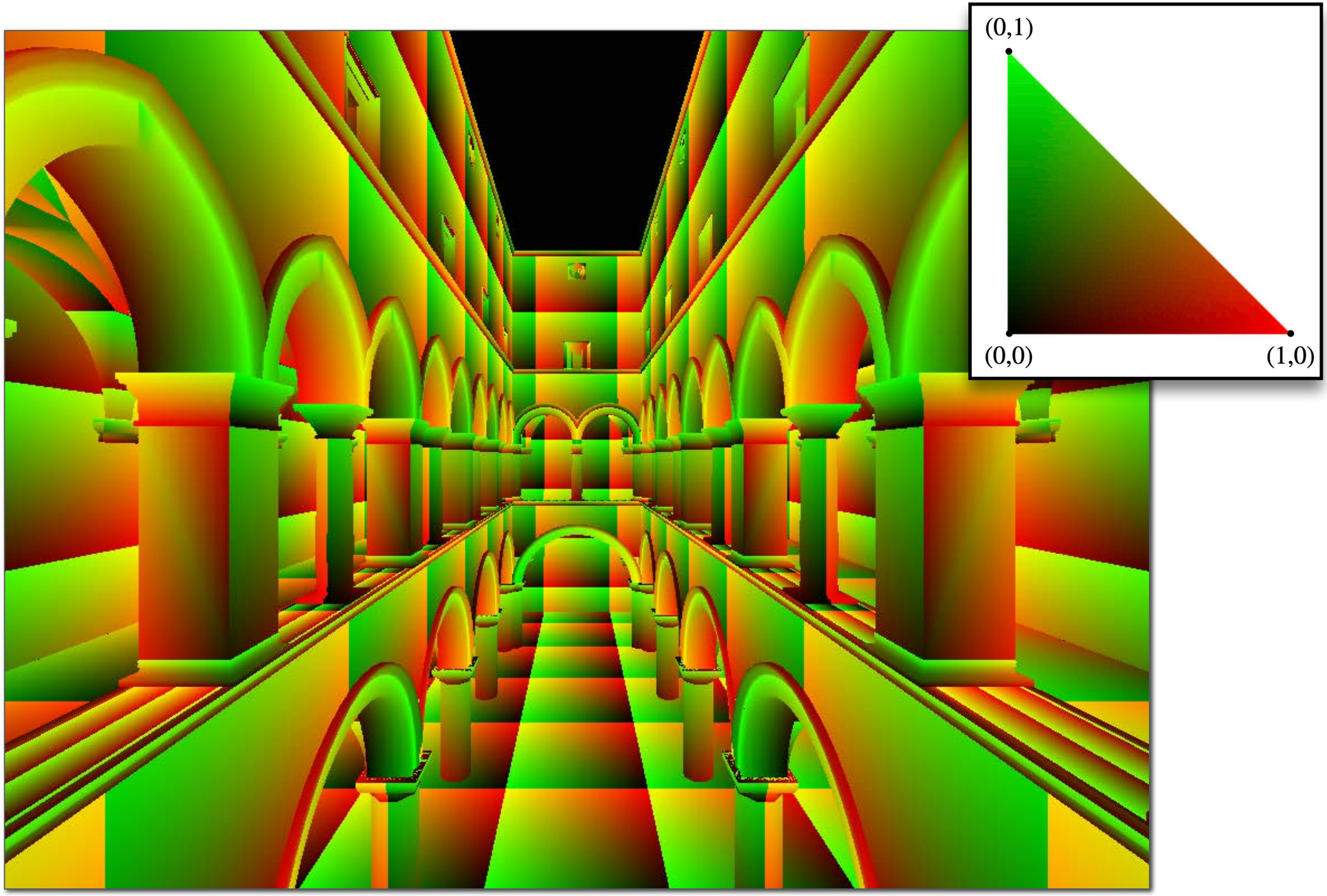
zoom





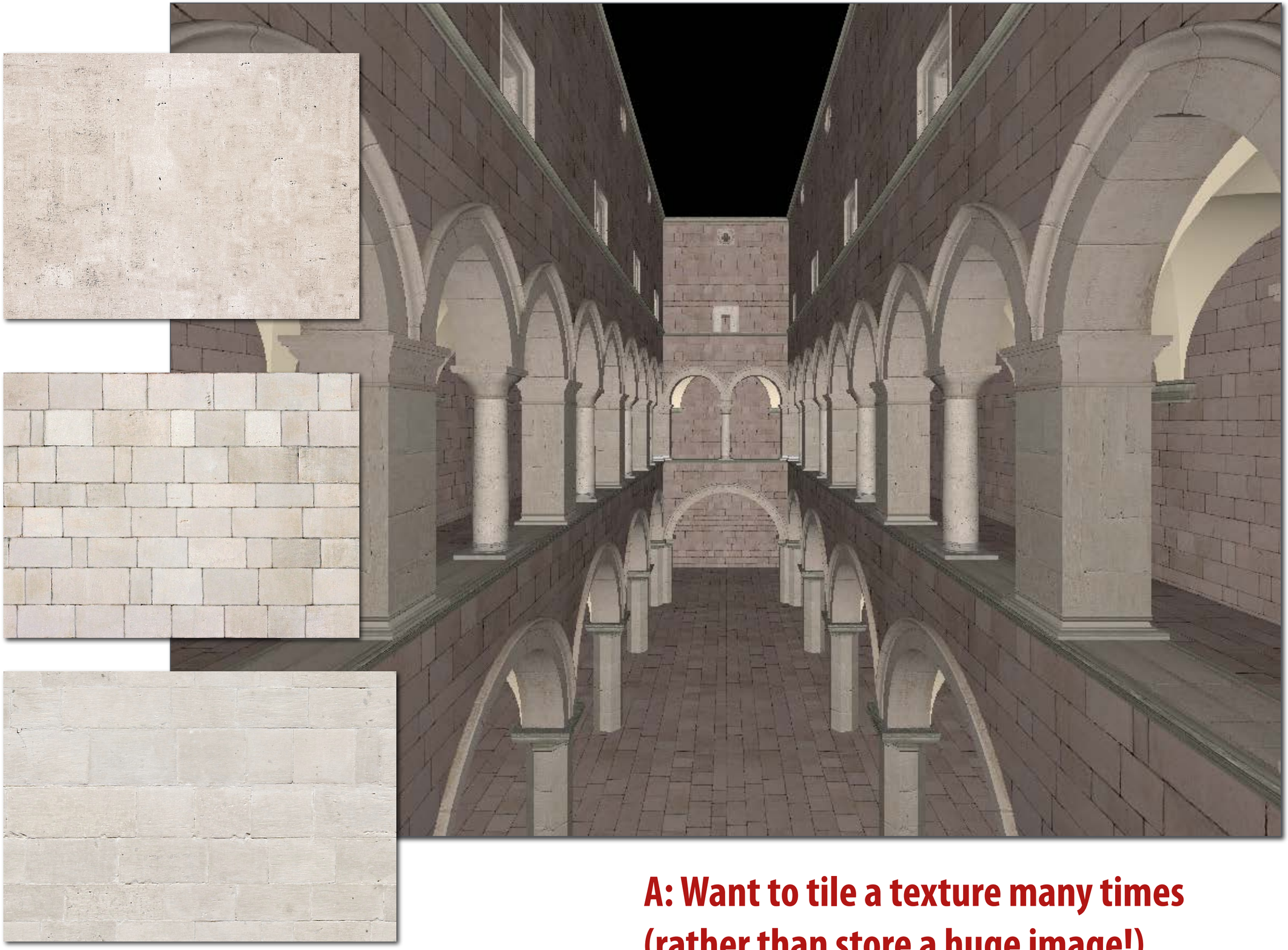


Another example: periodic coordinates



Q: Why do you think texture coordinates might repeat over the surface?

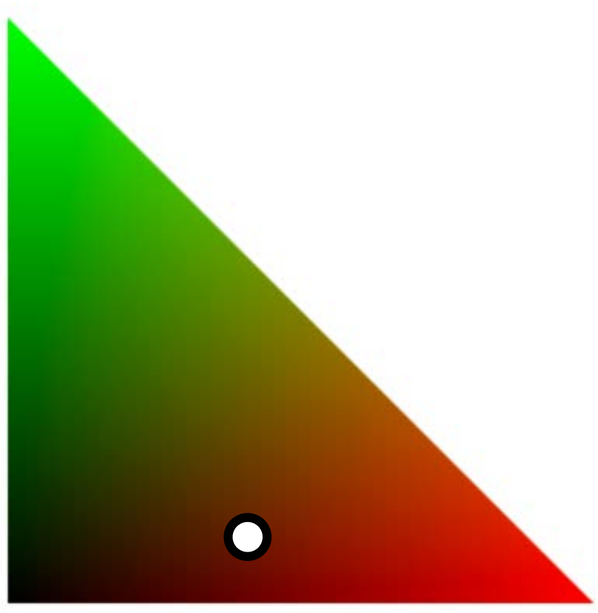
Textured Sponza

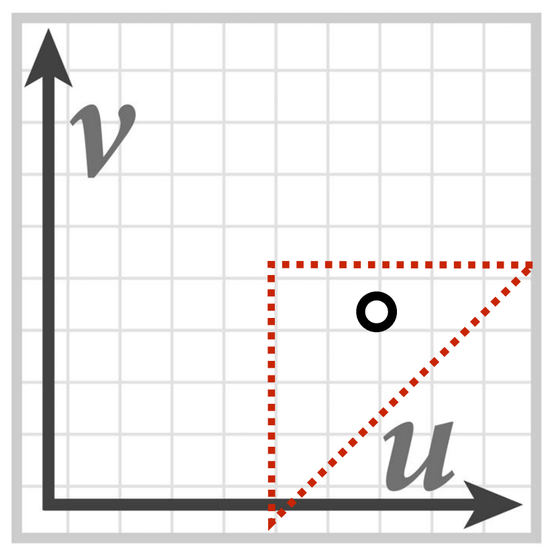


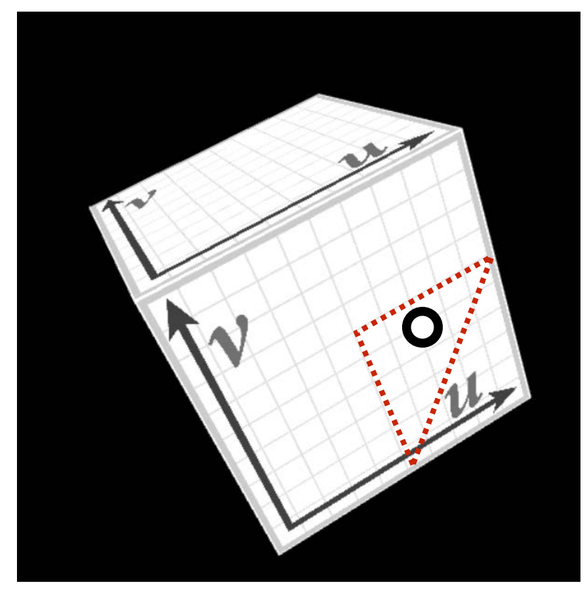
A: Want to tile a texture many times (rather than store a huge image!)

Texture Sampling 101

- **Basic algorithm for texture mapping:**
  - `for each pixel in the rasterized image:`
    - `interpolate` (u, v) `coordinates across triangle`
    - `sample (evaluate) texture at interpolated` (u, v)
    - `set color of fragment to sampled texture value`



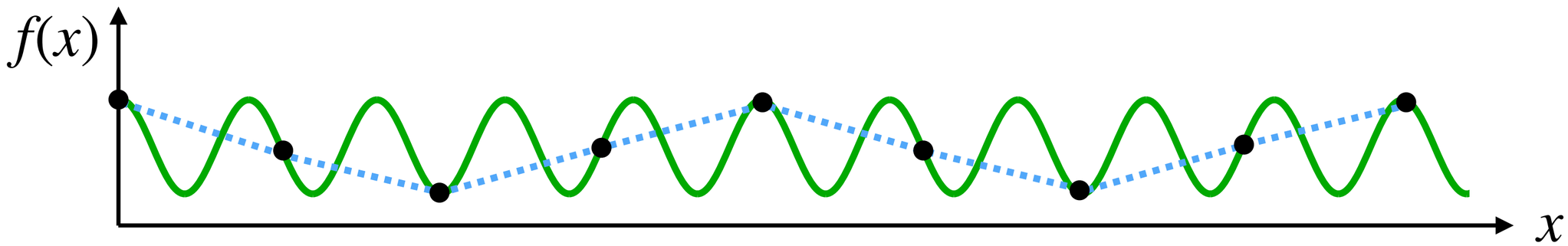


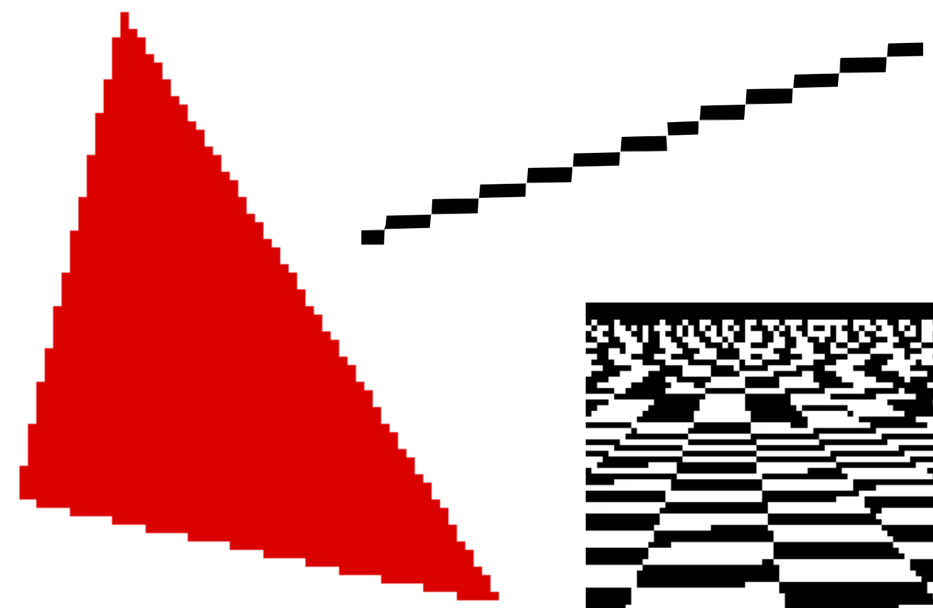


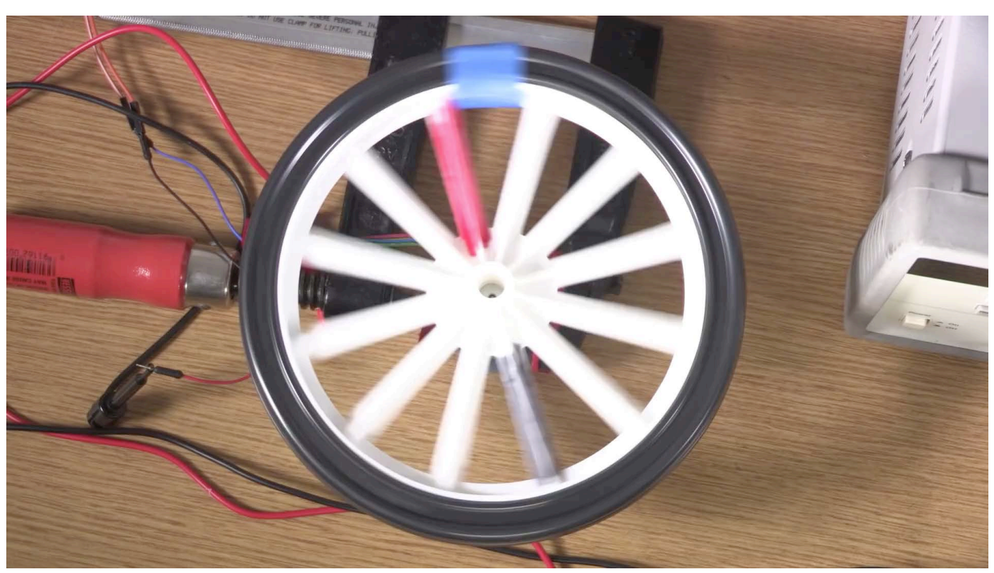
…sadly not this easy in general!

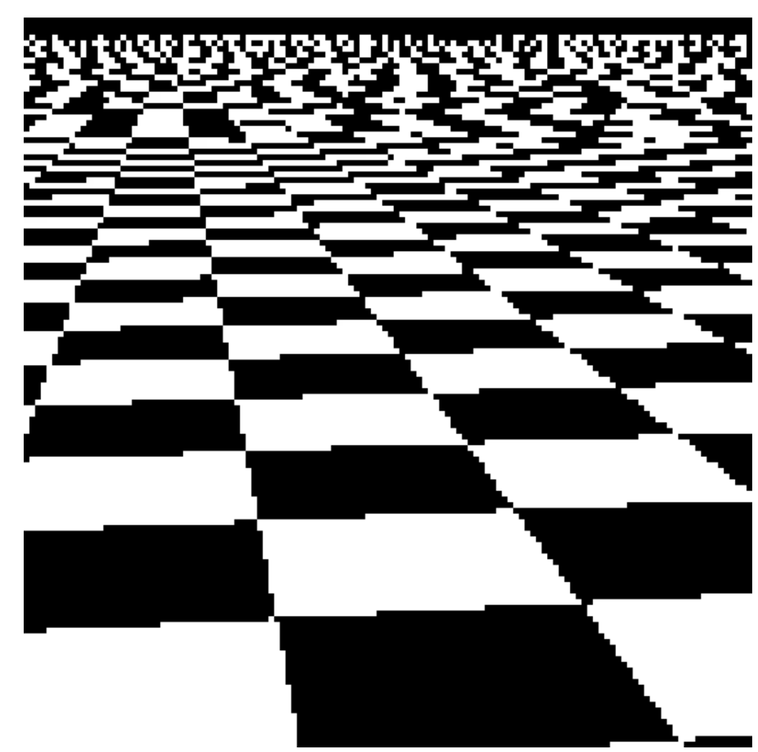
Recall: aliasing

Undersampling a high-frequency signal can result in aliasing





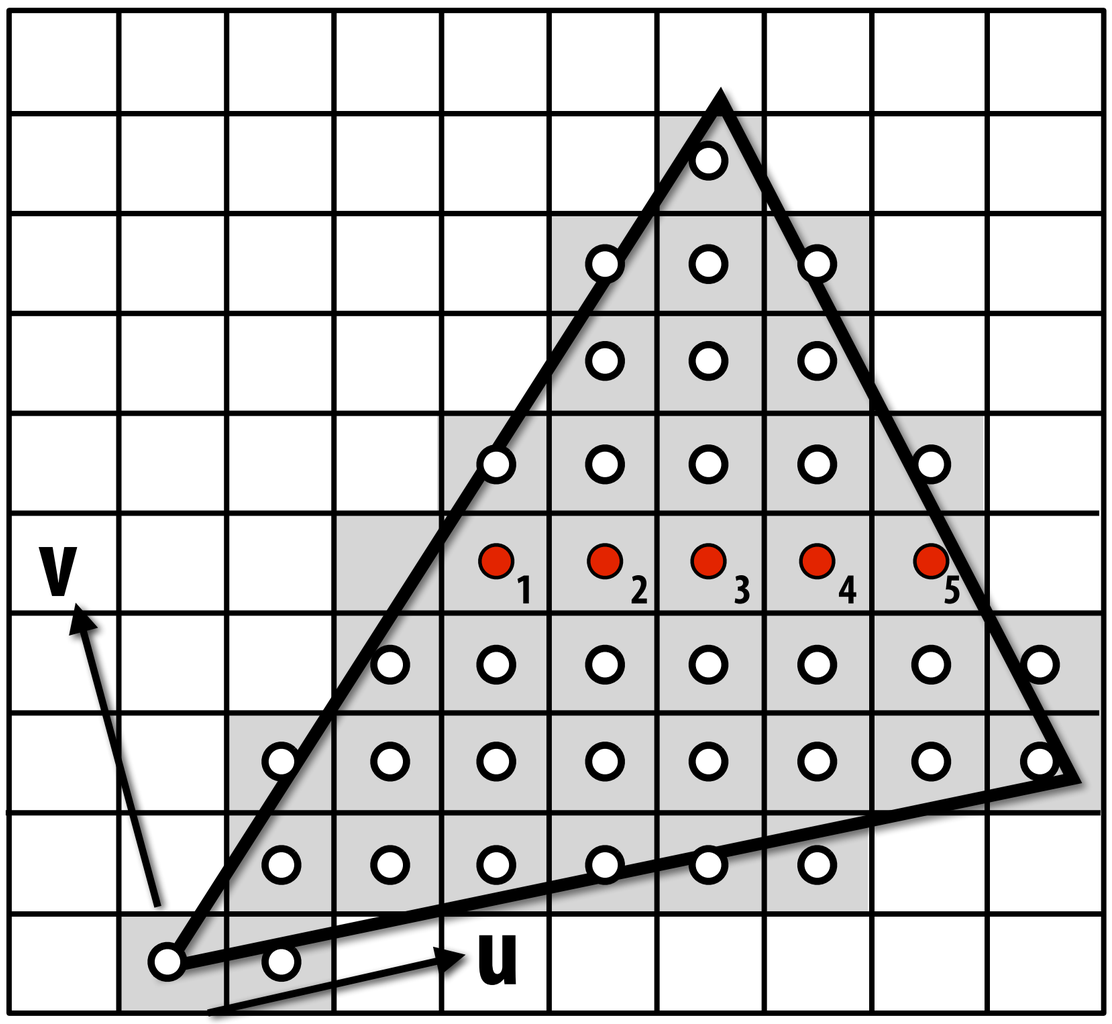




Visualizing texture samples

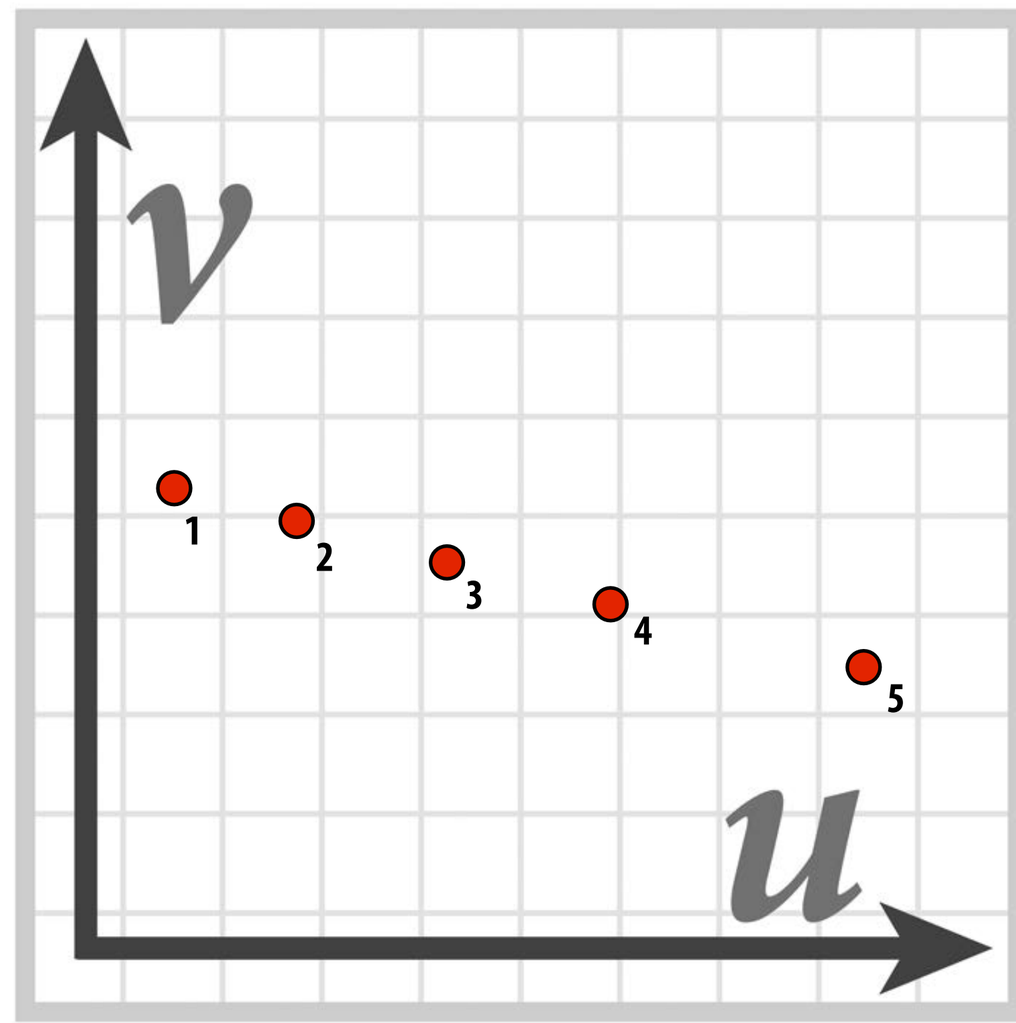
Since triangles are projected from 3D to 2D, pixels in screen space will correspond to regions of varying size & location in texture

sample positions in screen space



Sample positions are uniformly distributed in screen space (rasterizer samples triangle's appearance at these locations)

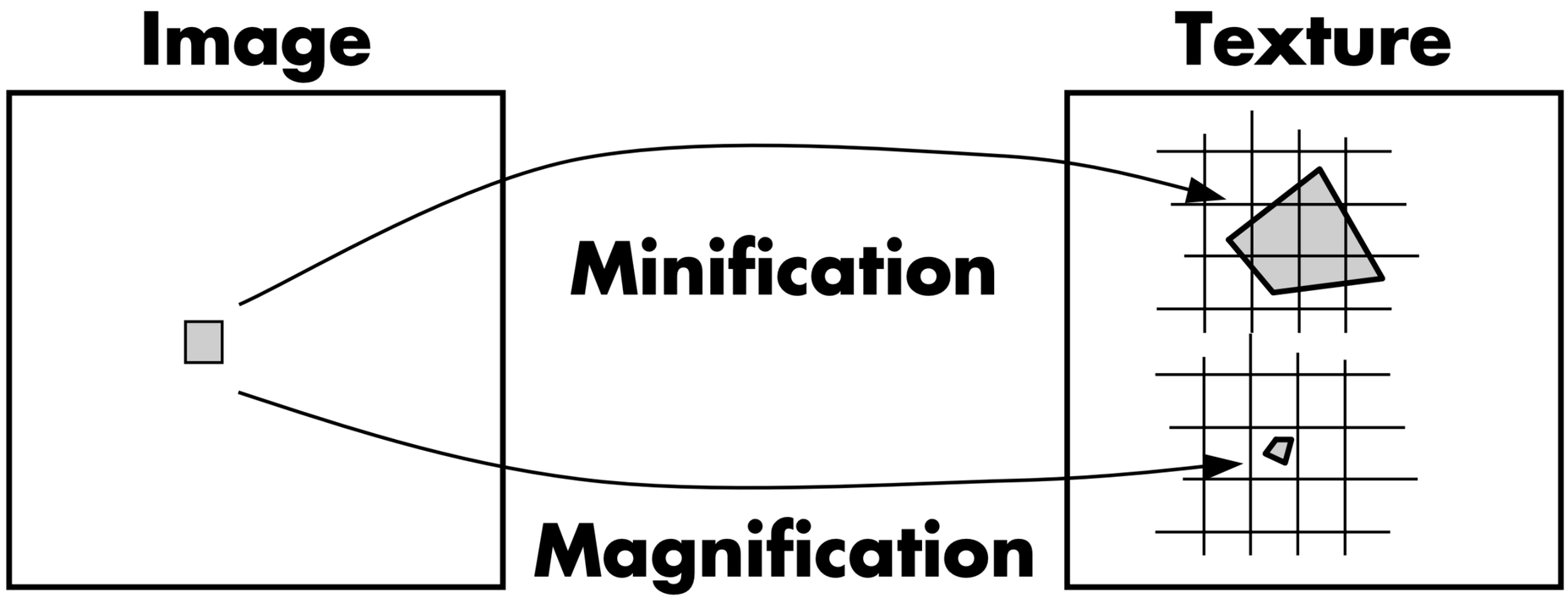
sample positions in texture space



Sample positions in texture space are not uniform (texture function is sampled at these locations)

Irregular sampling pattern makes it hard to avoid aliasing!

Magnification vs. Minification

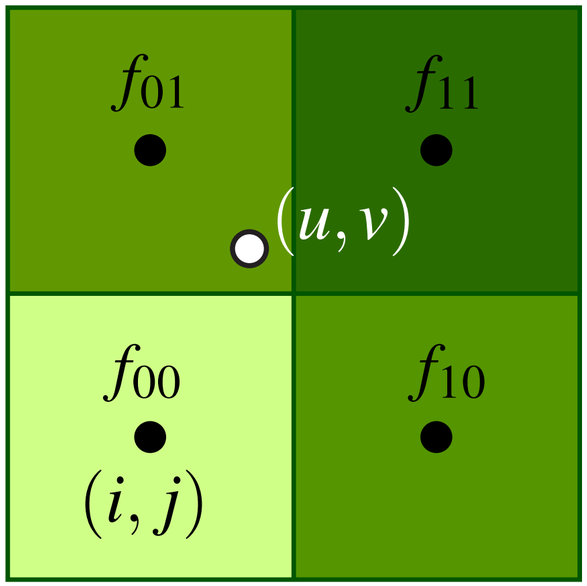


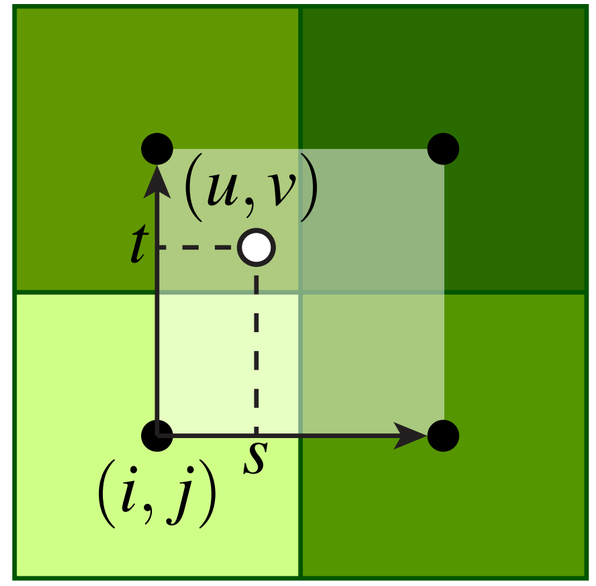
- **Magnification (easier):**
  - Example: camera is <u>very</u> close to scene object
  - Single screen pixel maps to tiny region of texture
  - Can just interpolate value at screen pixel center
- **Minification (harder):**
  - Example: scene object is <u>very</u> far away
  - Single screen pixel maps to large region of texture
  - Need to compute average texture value over pixel to avoid aliasing

Figure credit: Akeley and Hanrahan

Bilinear interpolation (magnification)

How can we "look up" a texture value at a non-integer location (u, v)?





linear (each row)

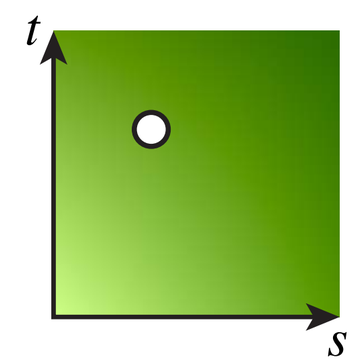
$$(1-s)f_{01} + sf_{11}$$

$$(1-s)f_{00} + sf_{10}$$

$$i = \left\lfloor u - \tfrac{1}{2} \right\rfloor \qquad s = u - \left(i + \tfrac{1}{2}\right) \in [0, 1]$$

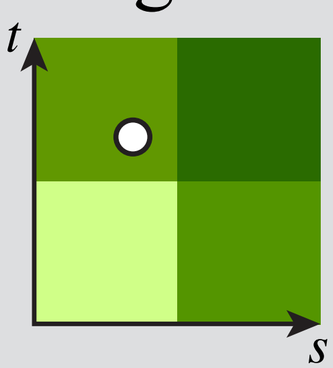
$$j = \left\lfloor v - \tfrac{1}{2} \right\rfloor \qquad t = v - \left(j + \tfrac{1}{2}\right) \in [0, 1]$$

bilinear



$$(1-t)\left((1-s)f_{00} + sf_{10}\right) + t\left((1-s)f_{01} + sf_{11}\right)$$

nearest neighbor

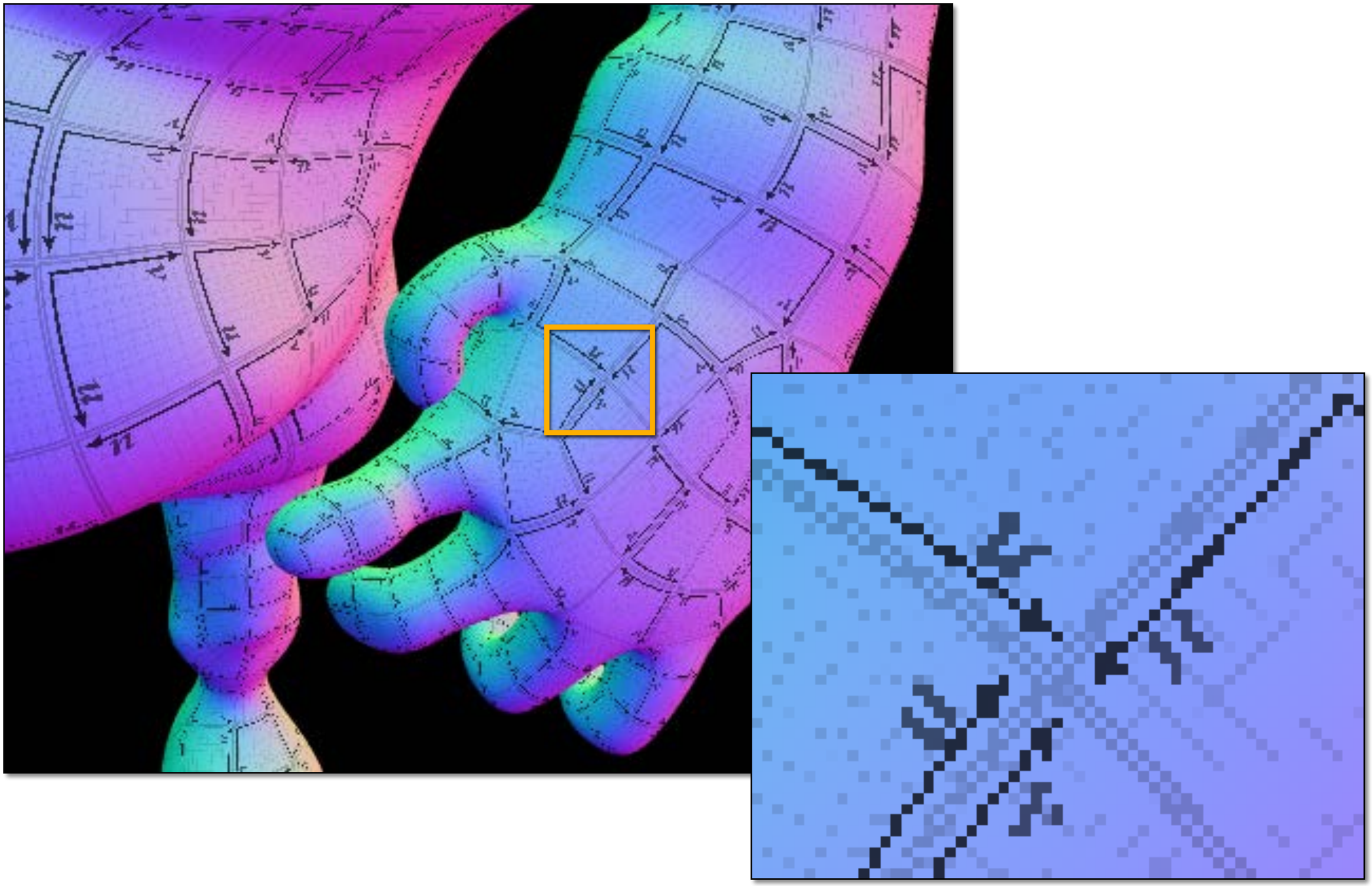


fast but ugly:

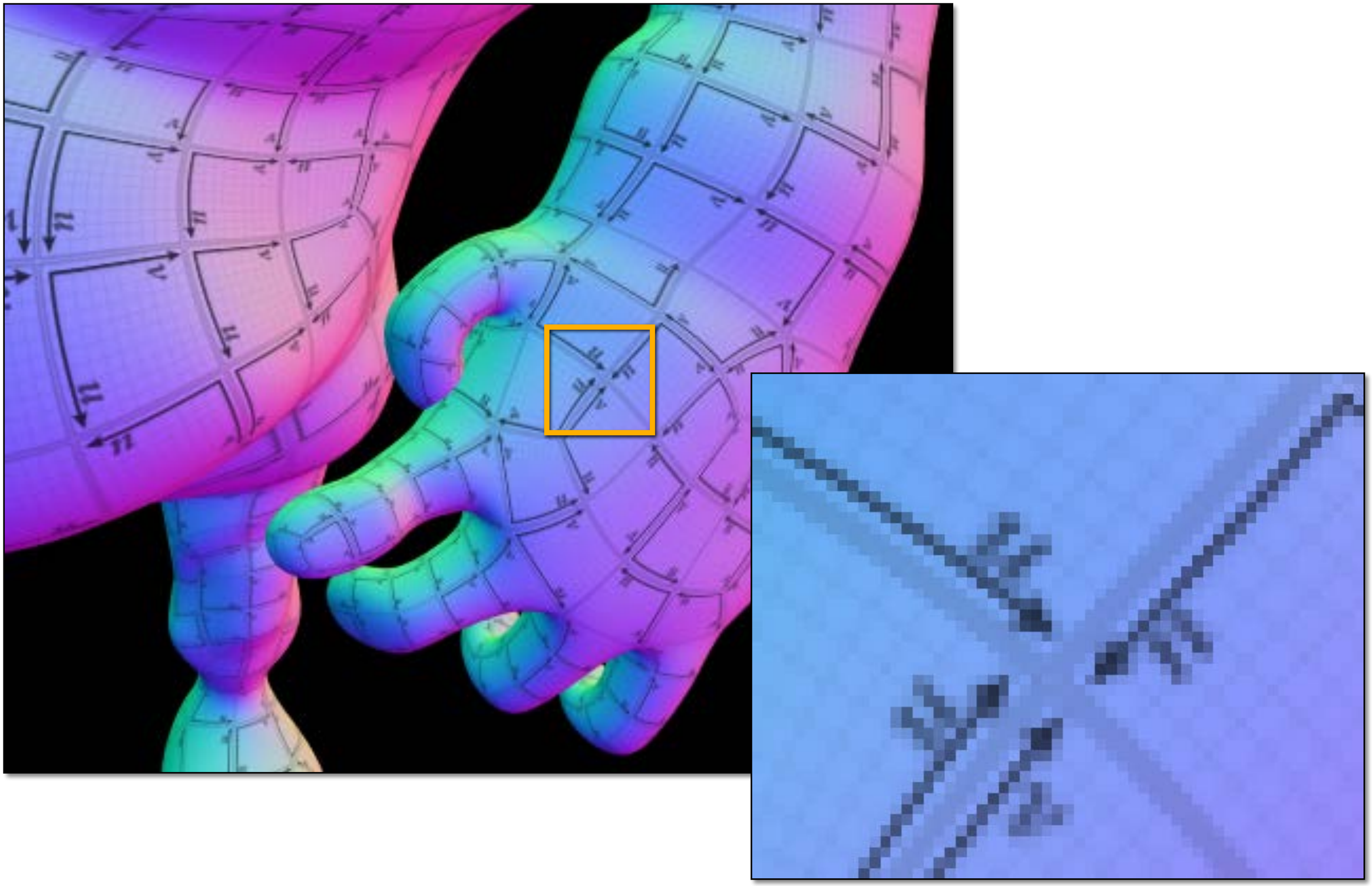
just grab value of nearest "texel" (texture pixel)

Q: What happens if we interpolate vertically <u>first</u>?

Aliasing due to minification

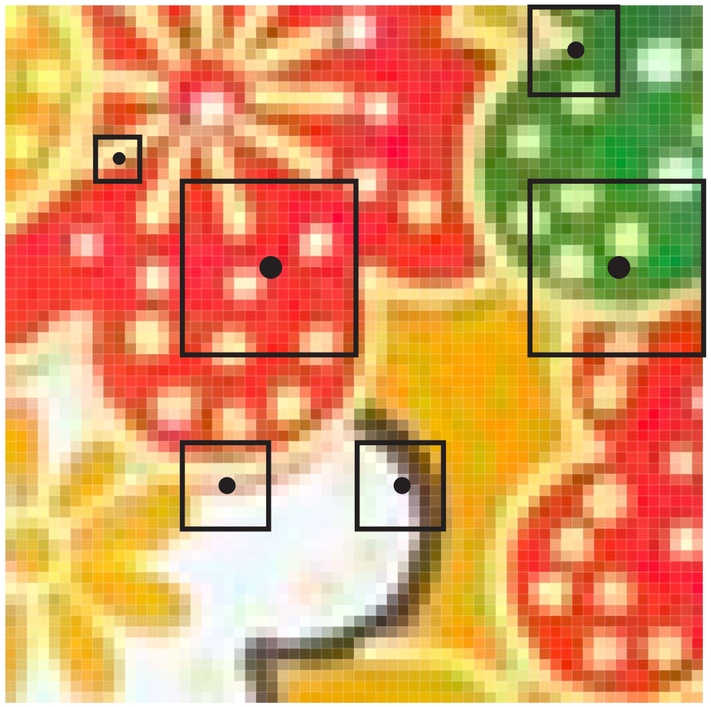


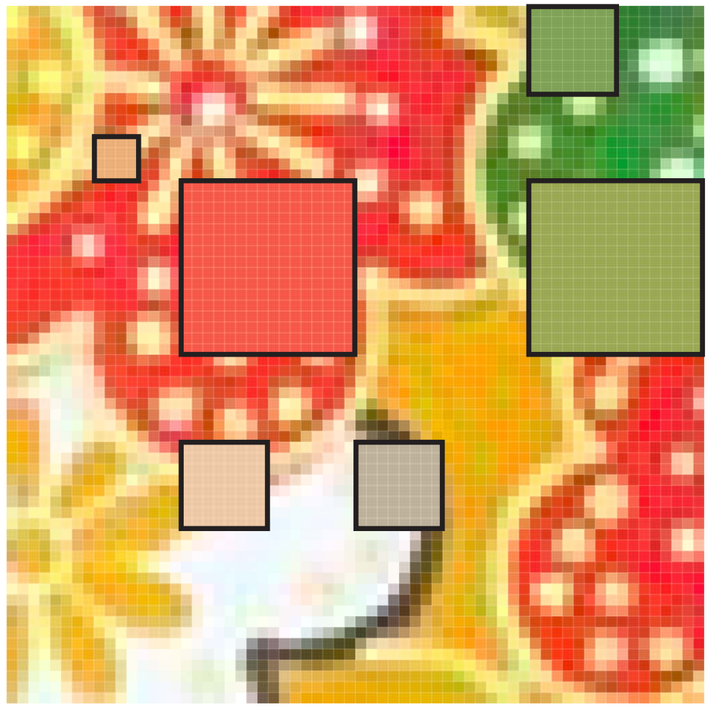
“Pre-filtering” texture (minification)



Texture prefiltering — basic idea

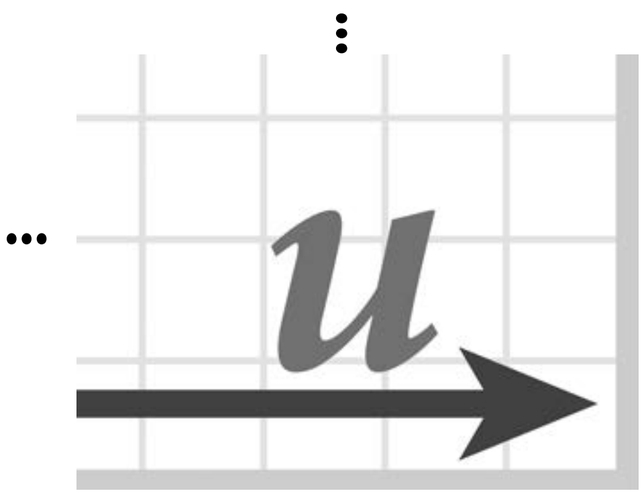
- Texture aliasing often occurs because a single pixel on the screen covers many pixels of the texture
- If we just grab the texture value at the center of the pixel, we get aliasing (get a "random" color that changes if the sample moves even very slightly)
- Ideally, would use the *average* texture value—but this is expensive to compute
- Instead, we can pre-compute the averages (once) and just look up these averages (many times) at run-time



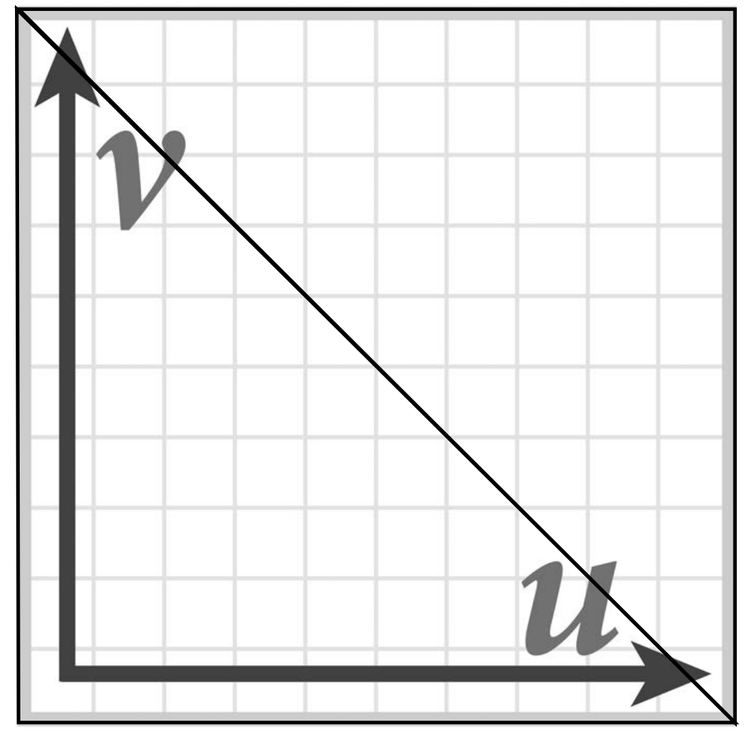


But *which* averages should we store? Can't precompute them all!

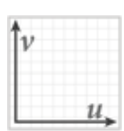
Prefiltered textures



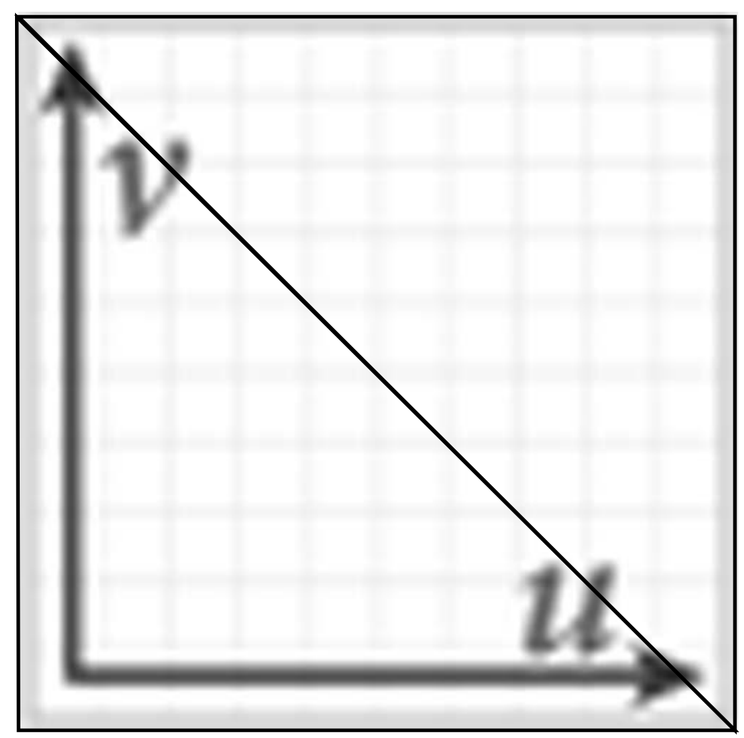
Actual texture: 700x700 image
(only a crop is shown)



Texture minification



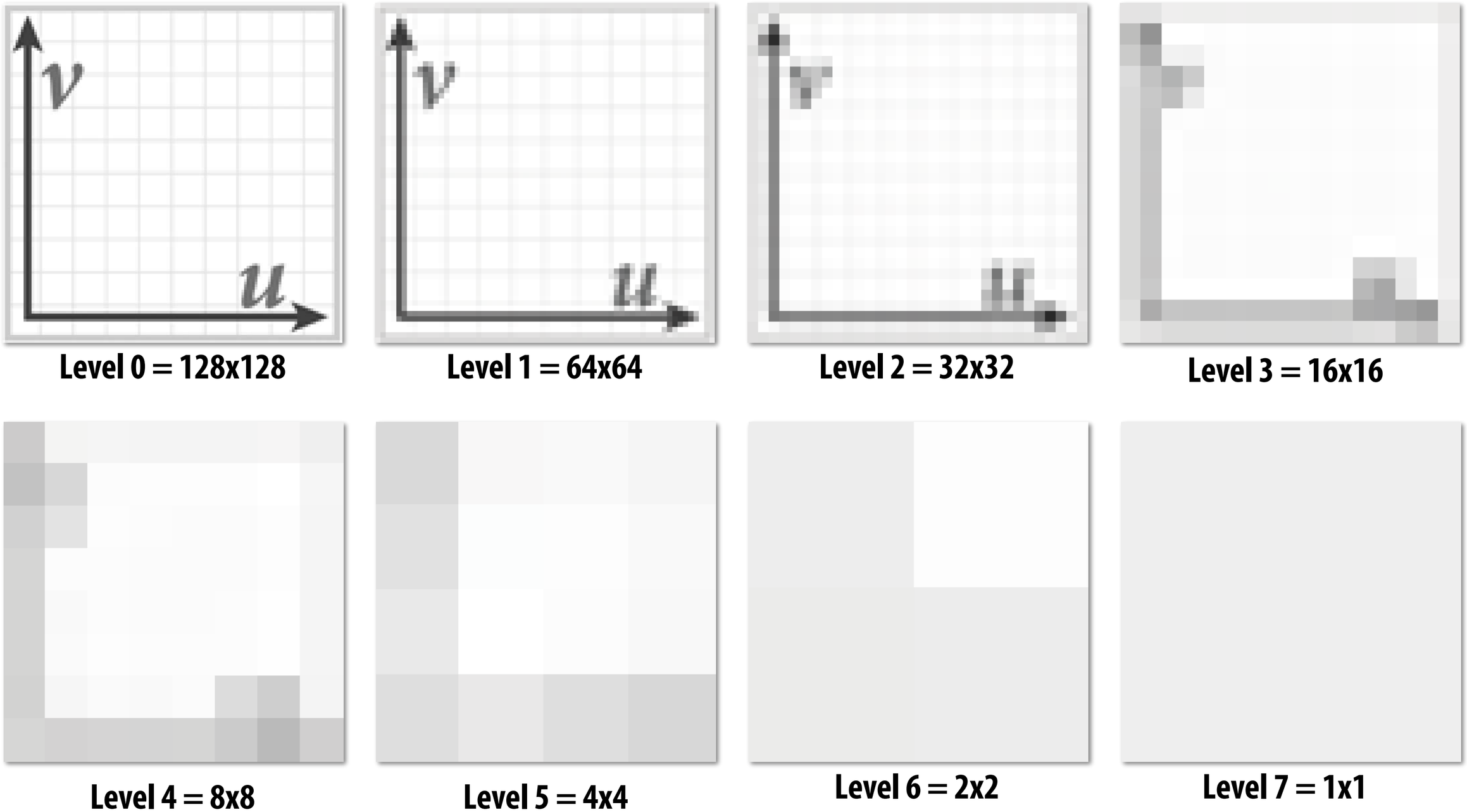
Actual texture: 64x64 image



Texture magnification

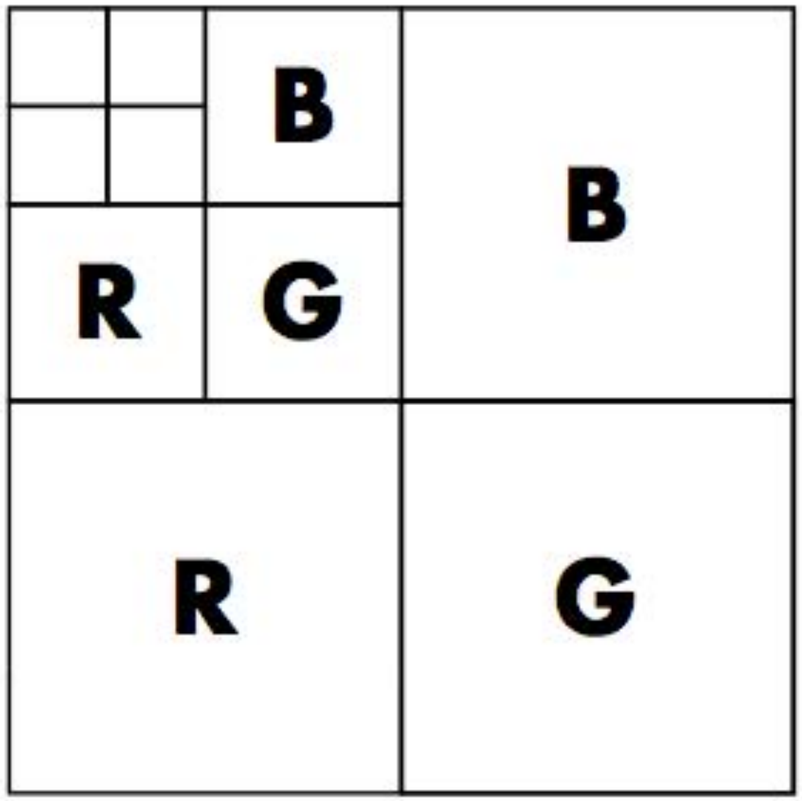
Q: Are two resolutions enough? A: No…

MIP map (L. Williams 83)

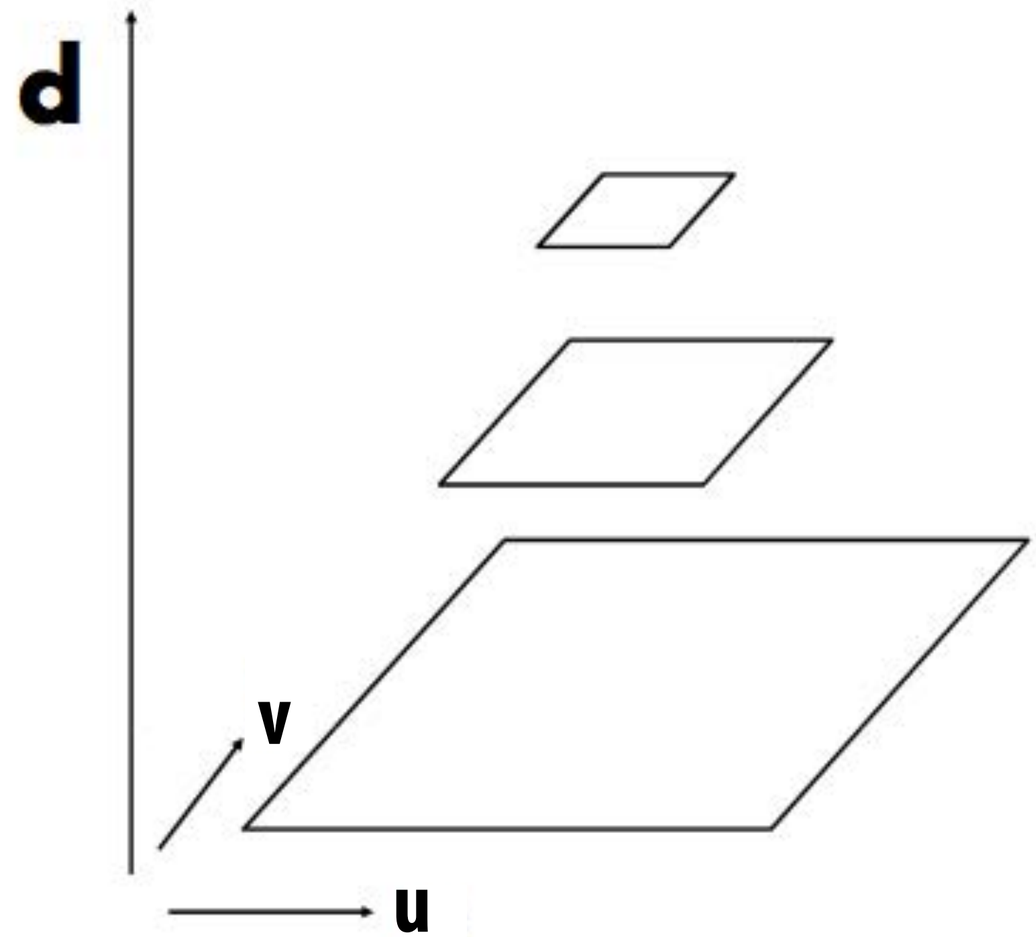


- Rough idea: store prefiltered image at "every possible scale"
- Texels at higher levels store average of texture over a region of texture space (downsampled)
- Later: look up a <u>single</u> pixel from MIP map of appropriate size

Mipmap (L. Williams 83)



Williams' original proposed mip-map layout



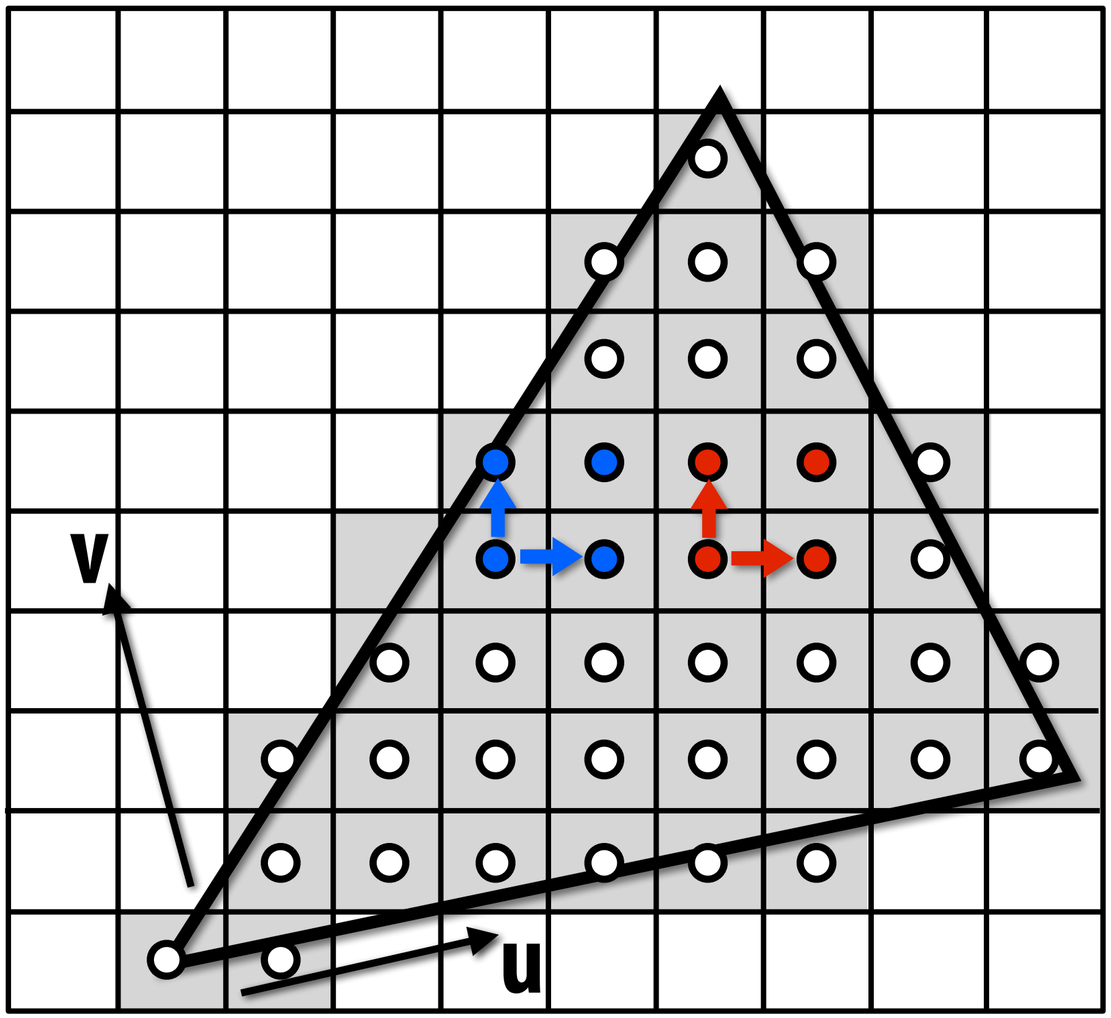
"Mip hierarchy"
level = *d*

Q: What's the storage overhead of a mipmap?

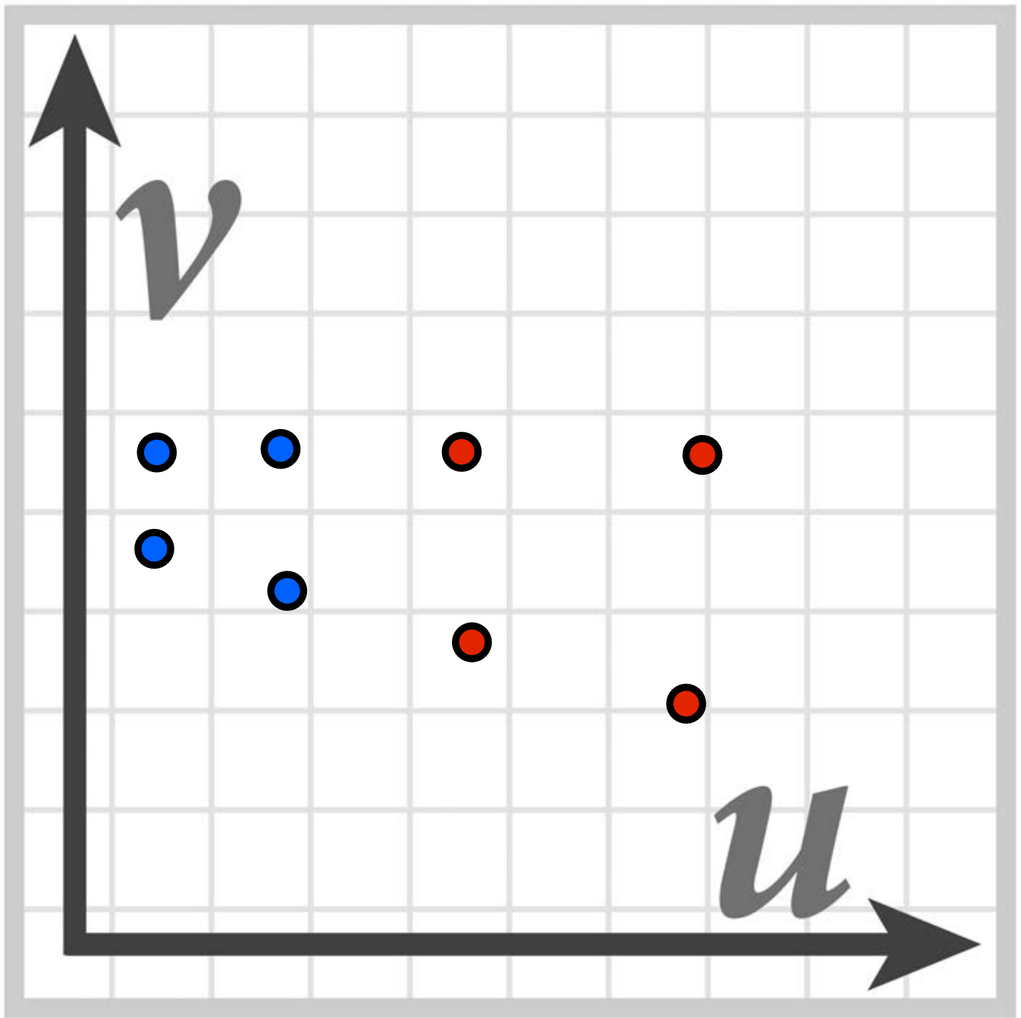
Slide credit: Akeley and Hanrahan

Computing MIP Map Level

Even within a single triangle, may want to sample from different MIP map levels:



Screen space

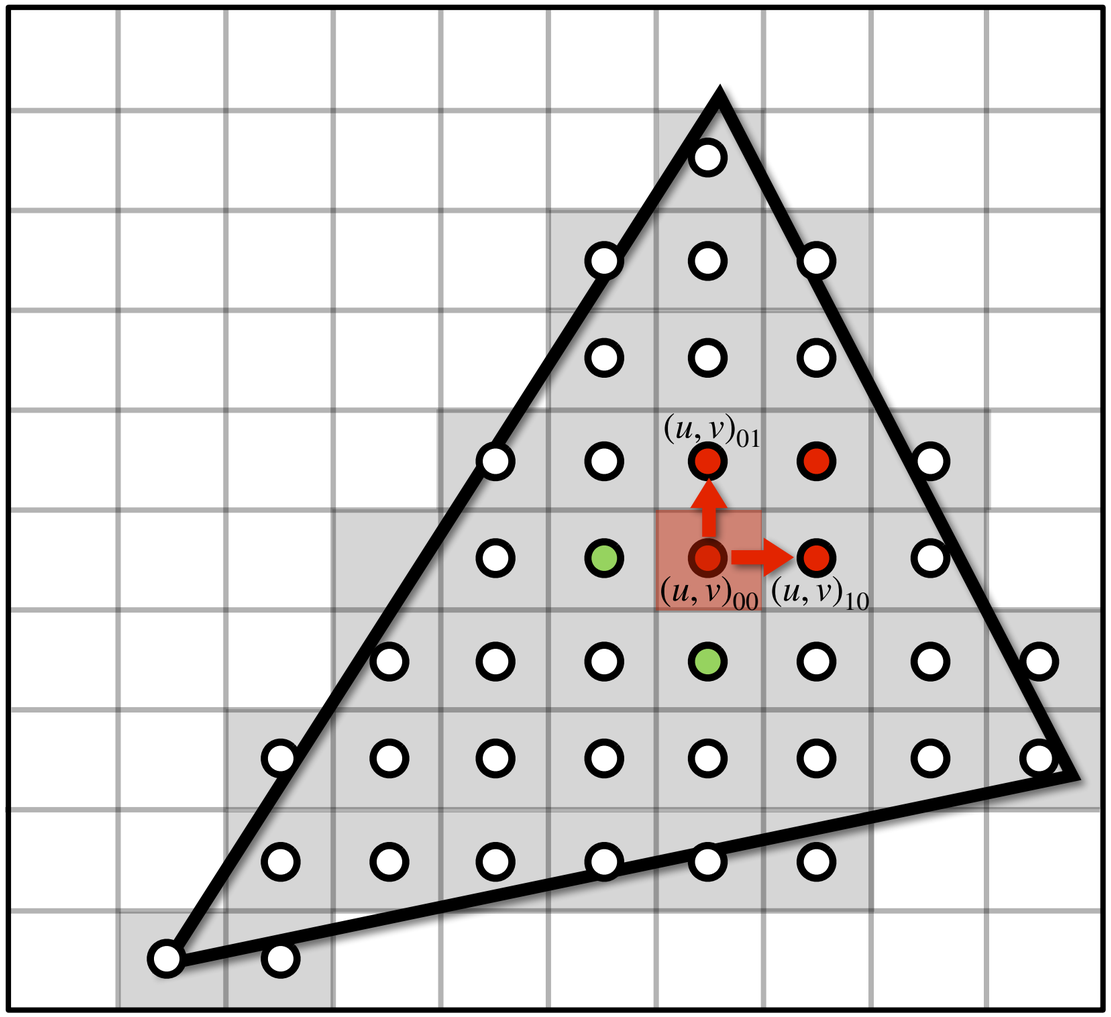


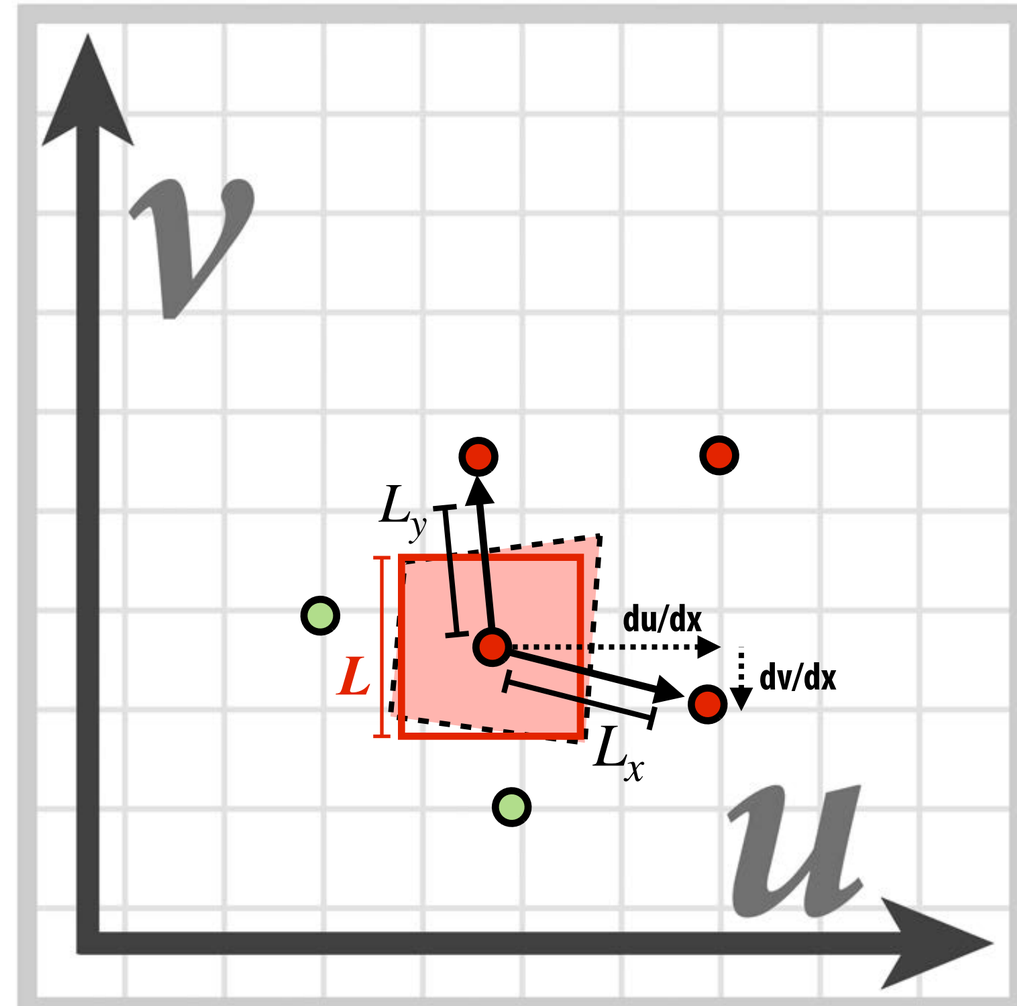
Texture space

Q: Which pixel should sample from a <u>coarser</u> MIP map level: the blue one, or the red one?

Computing Mip Map Level

Compute differences between texture coordinate values at neighboring samples





$$\frac{du}{dx} = u_{10} - u_{00} \qquad \frac{dv}{dx} = v_{10} - v_{00}$$

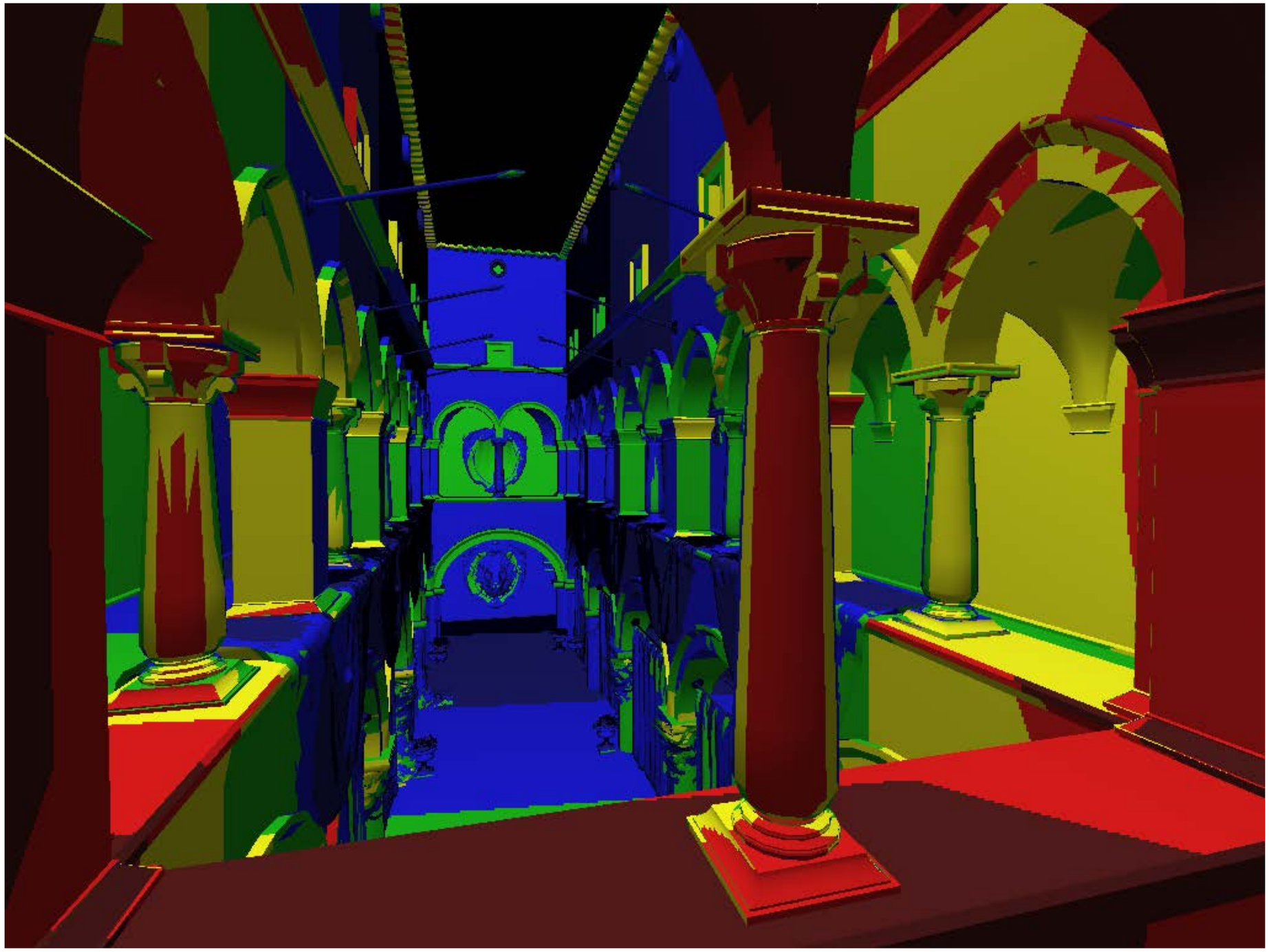
$$\frac{du}{dy} = u_{01} - u_{00} \qquad \frac{dv}{dy} = v_{01} - v_{00}$$

$$L_x^2 = \left(\frac{du}{dx}\right)^2 + \left(\frac{dv}{dx}\right)^2 \qquad L_y^2 = \left(\frac{du}{dy}\right)^2 + \left(\frac{dv}{dy}\right)^2$$

$$L = \sqrt{\max(L_x^2, L_y^2)}$$

mip-map level: $d = \log_2 L$

Visualization of mip-map level
(d clamped to nearest level)



Sponza (bilinear resampling at level 0)

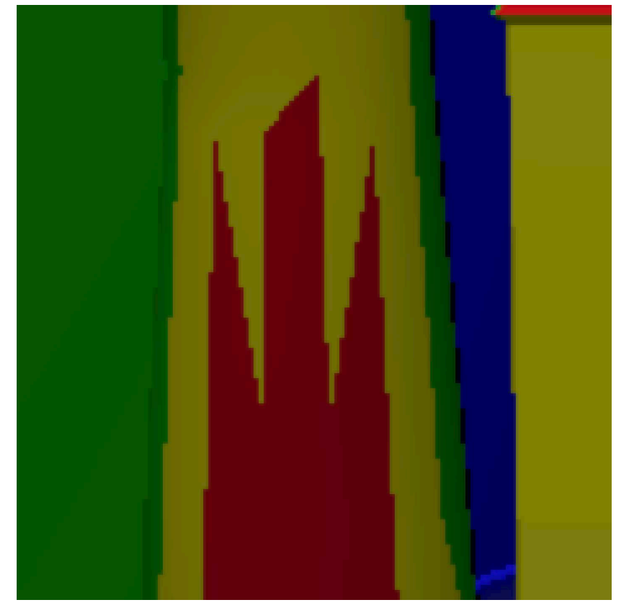
Sponza (bilinear resampling at level 2)

Sponza (bilinear resampling at level 4)

Sponza (MIP mapped)

Problem with basic MIP mapping

- If we just use the nearest level, can get artifacts where level "jumps"—appearance sharply transitions from detailed to blurry texture
- **IDEA**: rather than clamping the MIP map level to the closest integer, use the original (continuous) MIP map level d
- **PROBLEM**: we only computed a fixed number of MIP map levels. How do we interpolate between levels?



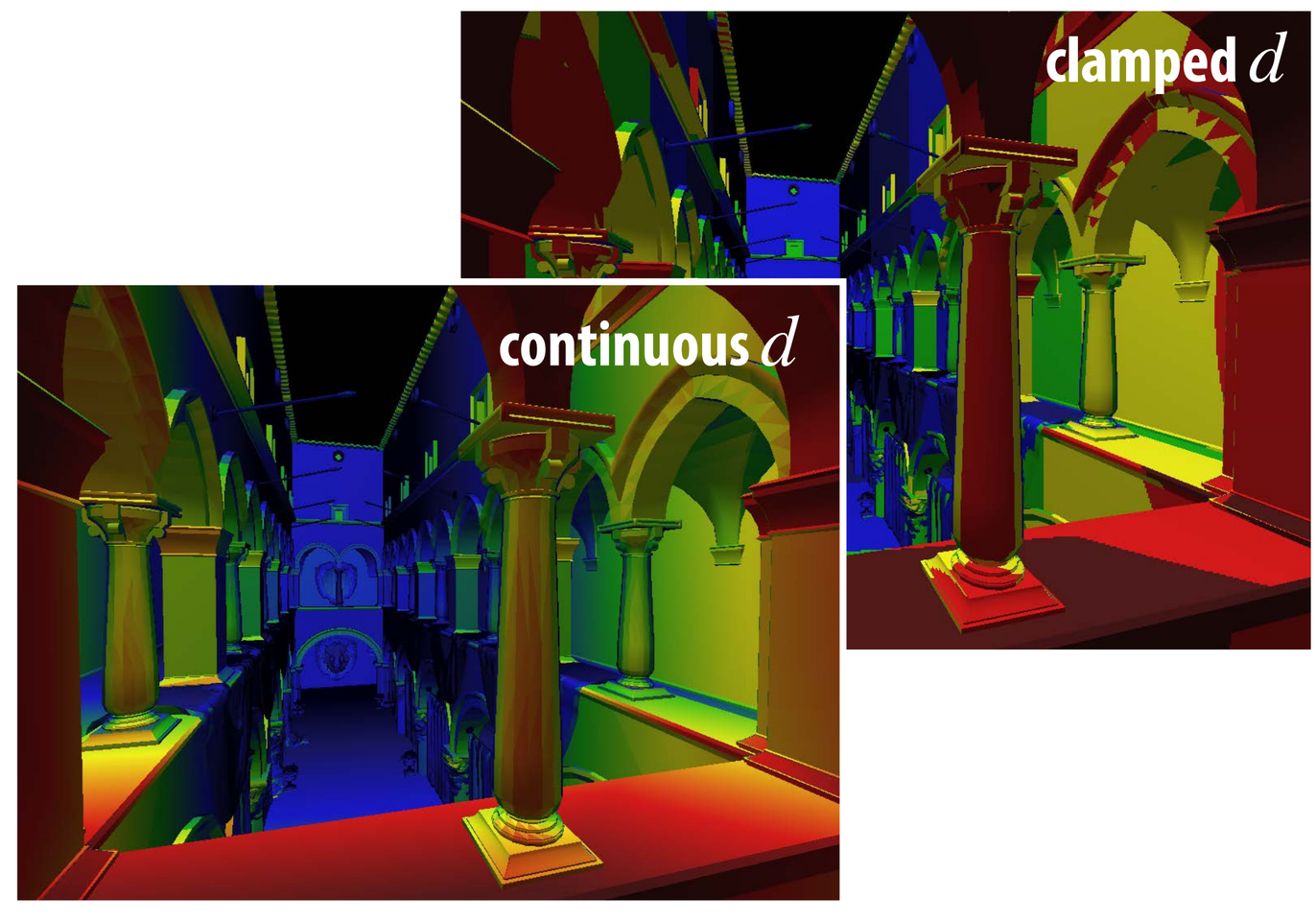
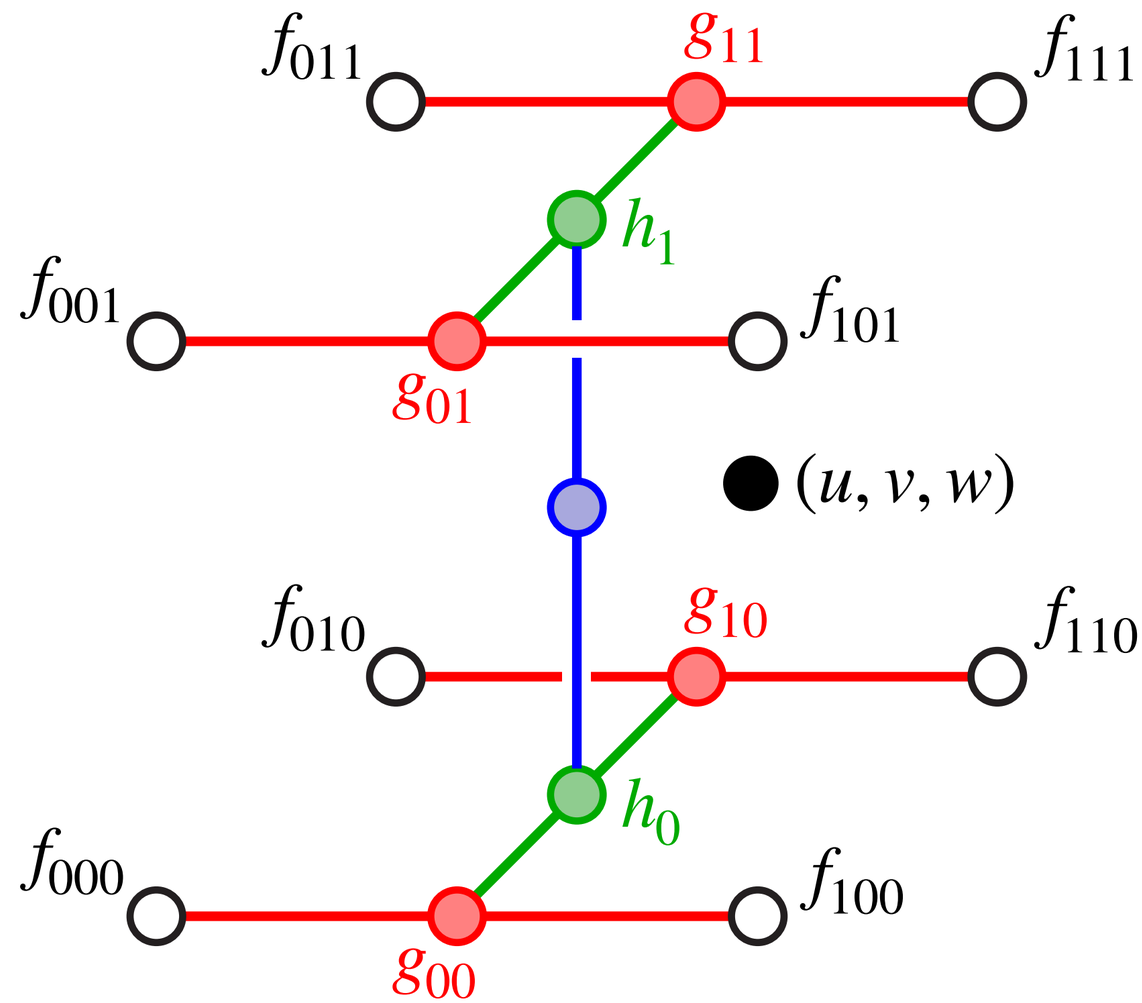


image adapted from: Akeley and Hanrahan

Trilinear Filtering

- Used <u>bilinear</u> filtering for 2D data; can use <u>trilinear</u> filtering for 3D data
- Given a point $(u, v, w) \in [0,1]^3$, and eight closest values f_{ijk}
- Just iterate linear filtering:
  - weighted average along u
  - weighted average along v
  - weighted average along w



$$g_{00} = (1-u)f_{000} + uf_{100}$$
$$g_{10} = (1-u)f_{010} + uf_{110}$$
$$g_{01} = (1-u)f_{001} + uf_{101}$$
$$g_{11} = (1-u)f_{011} + uf_{111}$$

$$h_0 = (1-v)g_{00} + vg_{10}$$
$$h_1 = (1-v)g_{01} + vg_{11}$$

$$(1-w)h_0 + wh_1$$

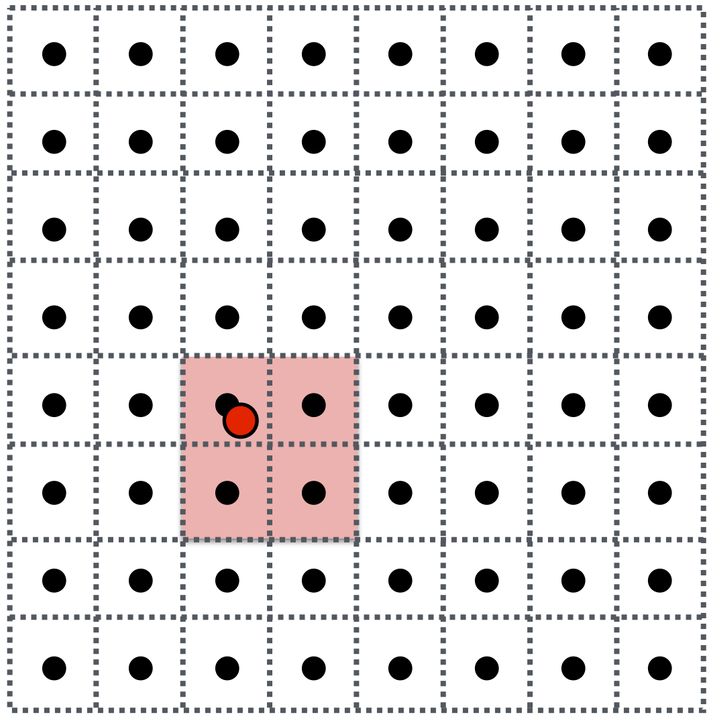
MIP Map Lookup

- MIP map interpolation works essentially the same way
  - not interpolating from 3D grid
  - interpolate from two MIP map levels closest to $d \in \mathbb{R}$
  - perform bilinear interpolation independently in each level
  - interpolate between two bilinear values using $w = d - \lfloor d \rfloor$

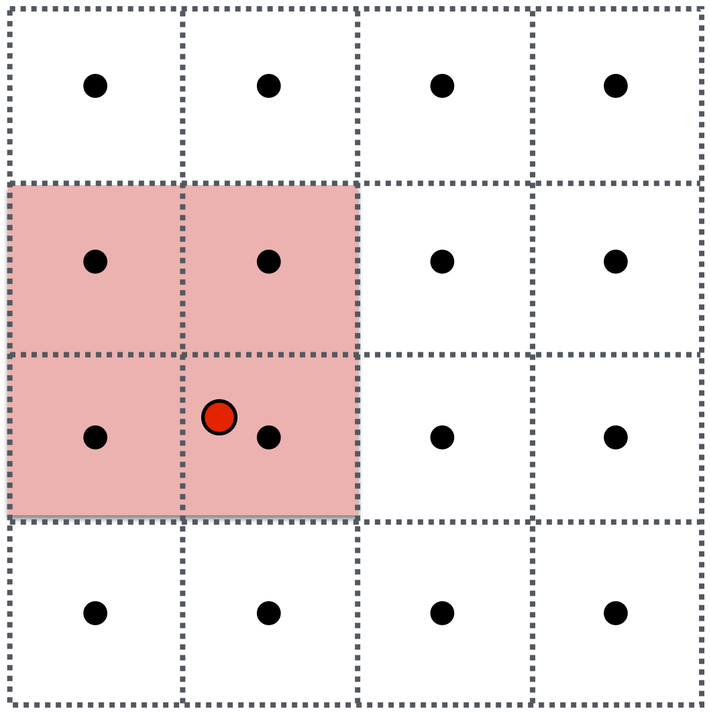
Starts getting expensive! (➟ specialized hardware)

<u>Bilinear interpolation:</u>
four texel reads
3 linear interpolations (3 mul + 6 add)

<u>Trilinear/MIP map interpolation:</u>
eight texel reads
7 linear interpolations (7 mul + 14 add)



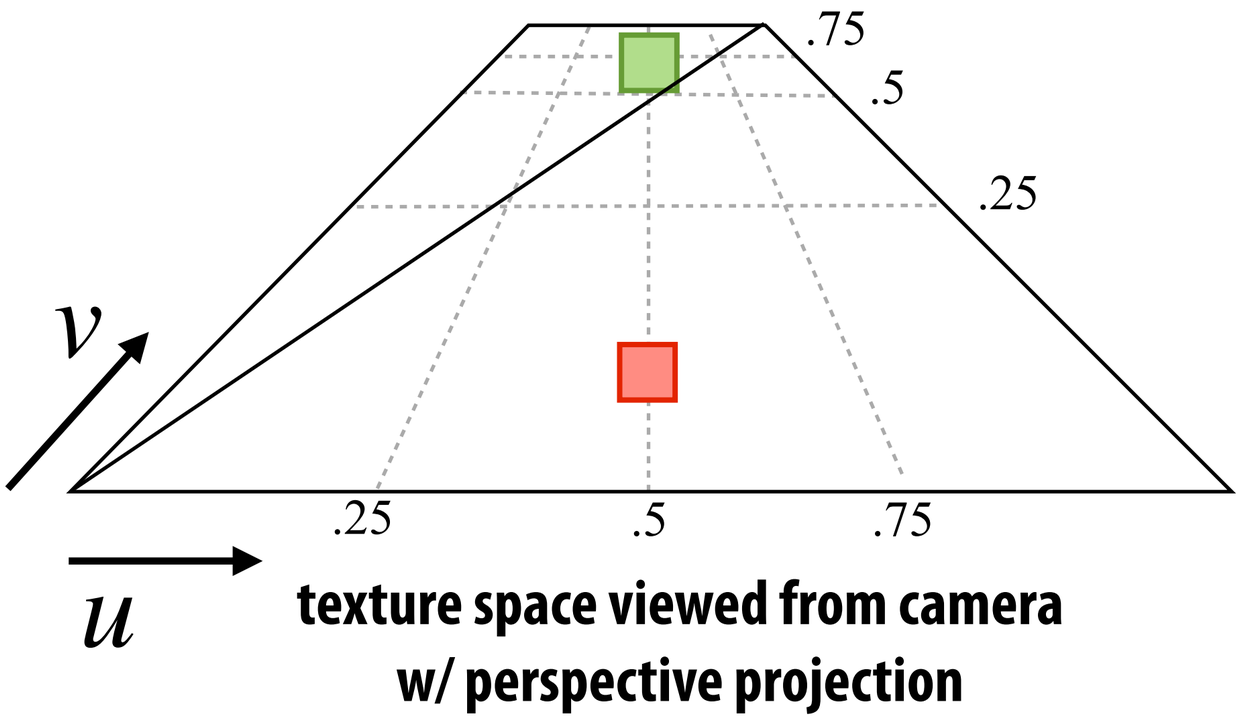
mip-map texels: level $\lfloor d \rfloor$



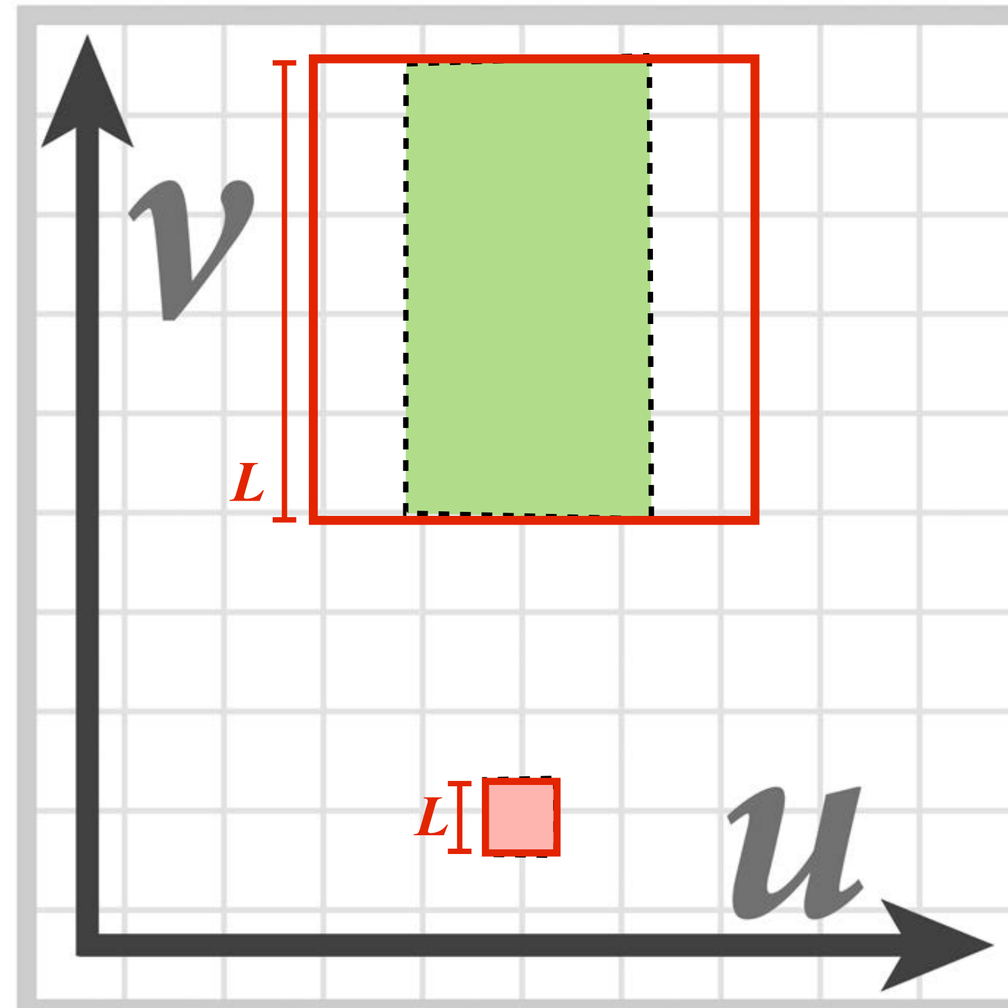
mip-map texels: level $\lfloor d \rfloor + 1$

Anisotropic Filtering

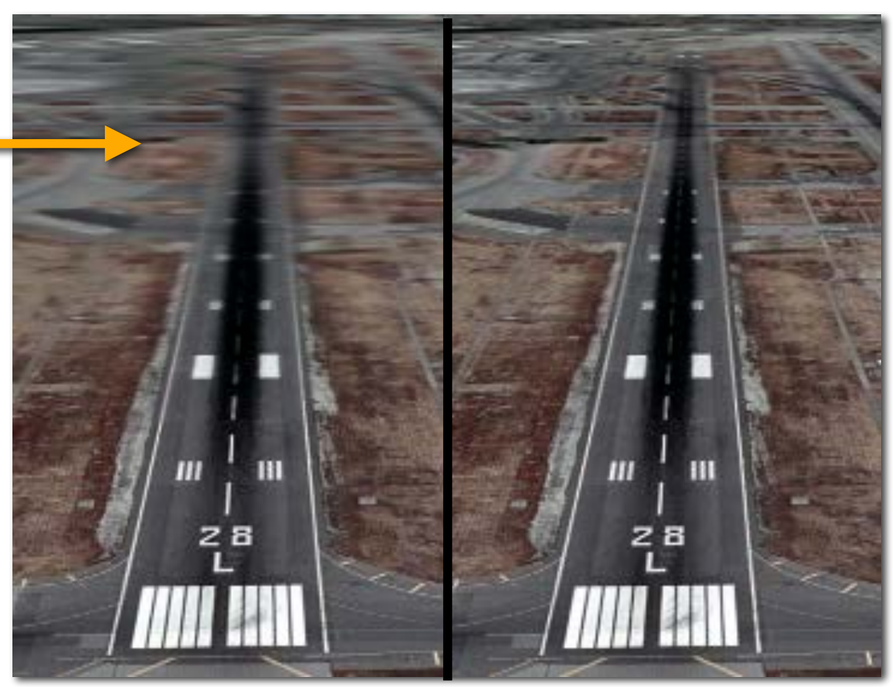
At grazing angles, samples may be stretched out by (very) different amounts along u and v



texture space viewed from camera
w/ perspective projection



Overblurring in u direction



isotropic Filtering (trilinear)

anisotropic Filtering

Common solution: combine multiple MIP map samples
(even more arithmetic/bandwidth!)

Texture Sampling Pipeline

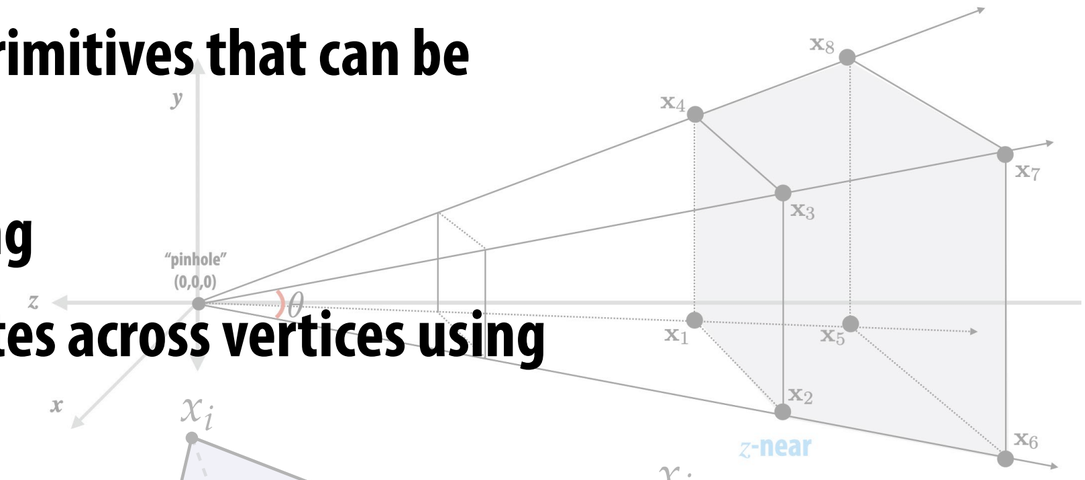
1. Compute u and v from screen sample (x, y) via barycentric interpolation
2. Approximate $\frac{du}{dx}, \frac{du}{dy}, \frac{dv}{dx}, \frac{dv}{dy}$ by taking differences of screen-adjacent samples
3. Compute mip map level d
4. Convert normalized $[0,1]$ texture coordinate (u, v) to pixel locations $(U, V) \in [W, H]$ in texture image
5. Determine addresses of texels needed for filter (e.g., eight neighbors for trilinear)
6. Load texels into local registers
7. Perform tri-linear interpolation according to (U, V, d)
8. (…even more work for anisotropic filtering…)

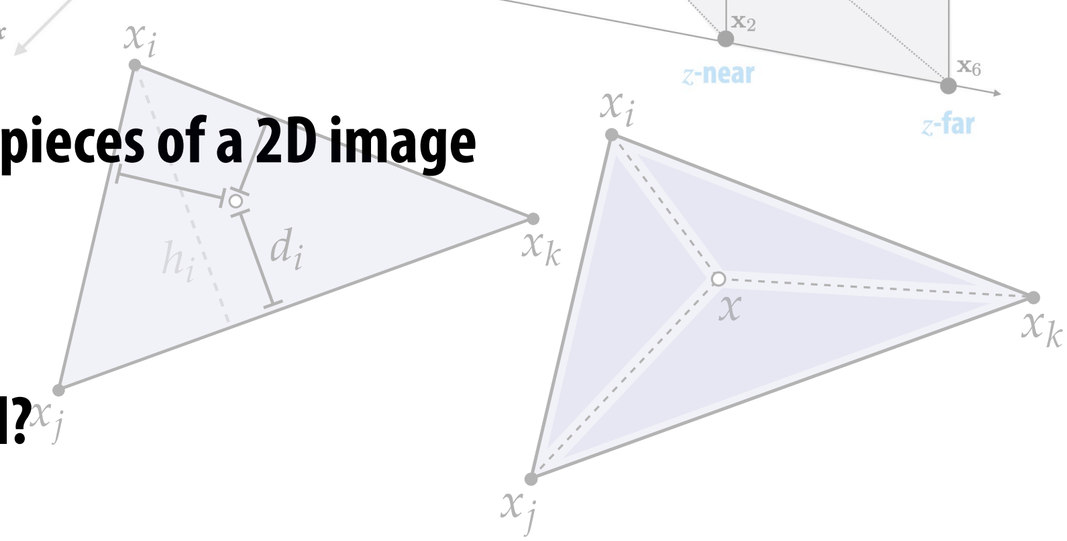
Takeaway: high-quality texturing requires <u>far</u> more work than just looking up a pixel in an image! Each sample demands significant arithmetic & bandwidth

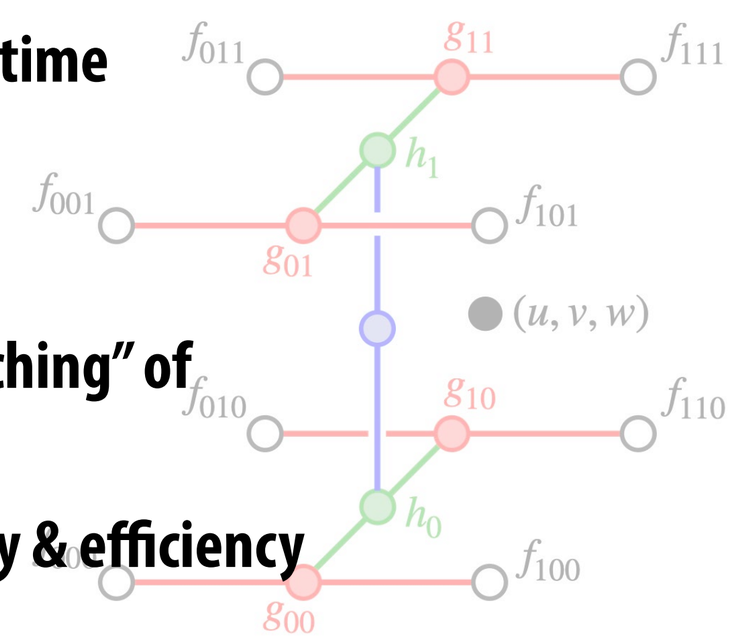
For this reason, graphics processing units (GPUs) have dedicated, fixed-function hardware support to perform texture sampling operations

Perspective & Texture Mapping—Summary

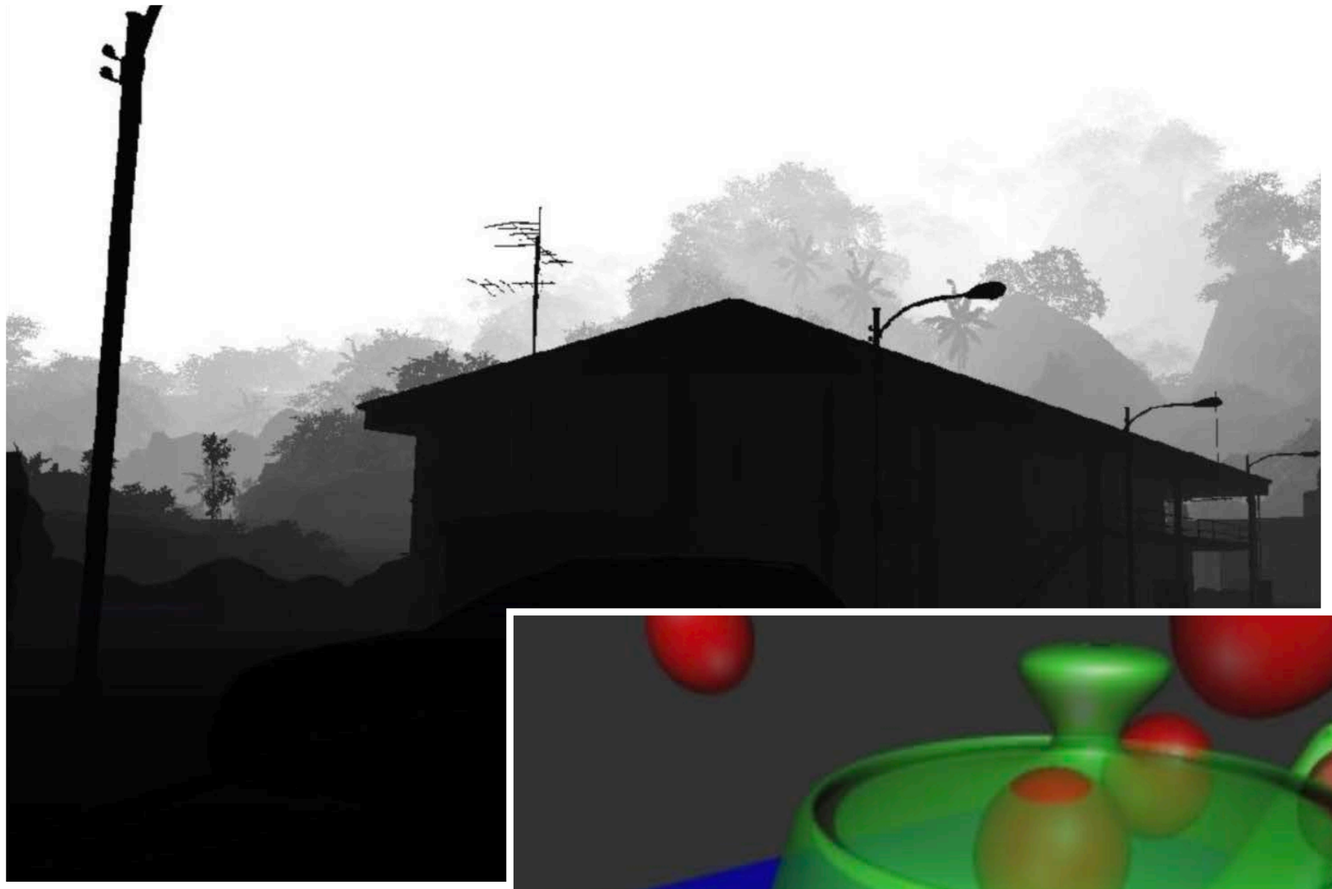
- Perspective projection turns 3D primitives into 2D primitives that can be rasterized
  - View frustum used to manage clipping, Z-fighting
- Once we have 2D primitives, can interpolate attributes across vertices using barycentric coordinates
- Important example: texture coordinates, used to copy pieces of a 2D image onto a 3D surface
- Careful texture filtering is needed to avoid aliasing
  - Key idea: what's the <u>average</u> color covered by a pixel?
  - For magnification, can just do a bilinear lookup
  - For minification, use prefiltering to compute averages ahead of time
    - a MIP map stores averages at different levels
    - blend between levels using trilinear filtering
  - At grazing angles, anisotropic filtering needed to deal w/ "stretching" of samples
  - In general, no perfect solution to aliasing! Try to balance quality & efficiency

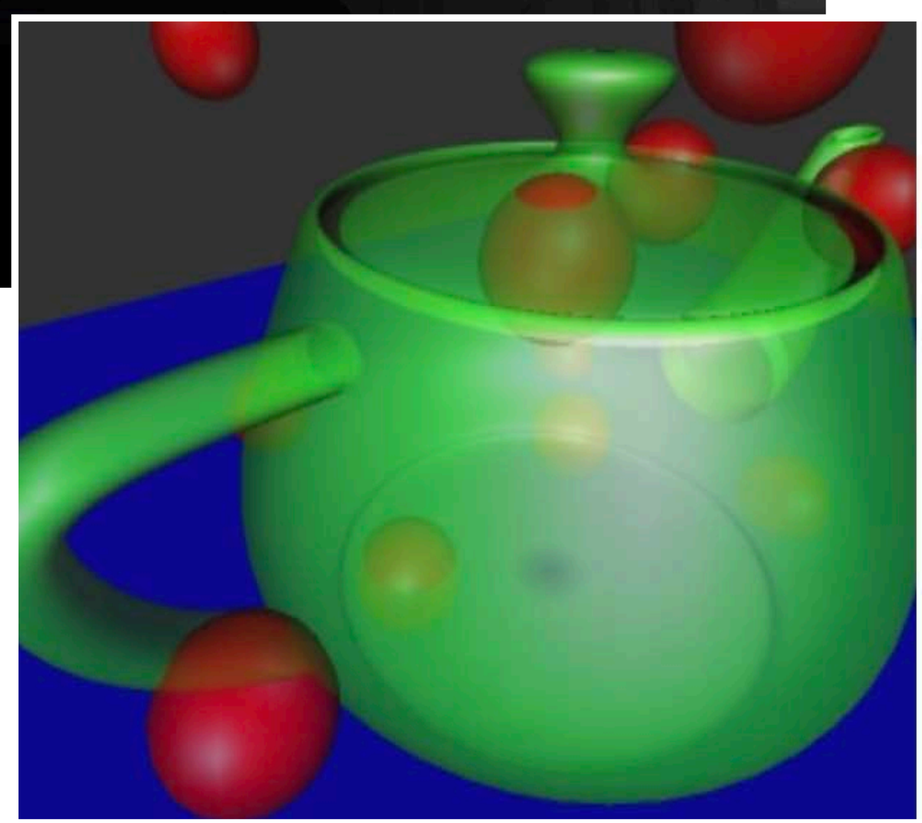


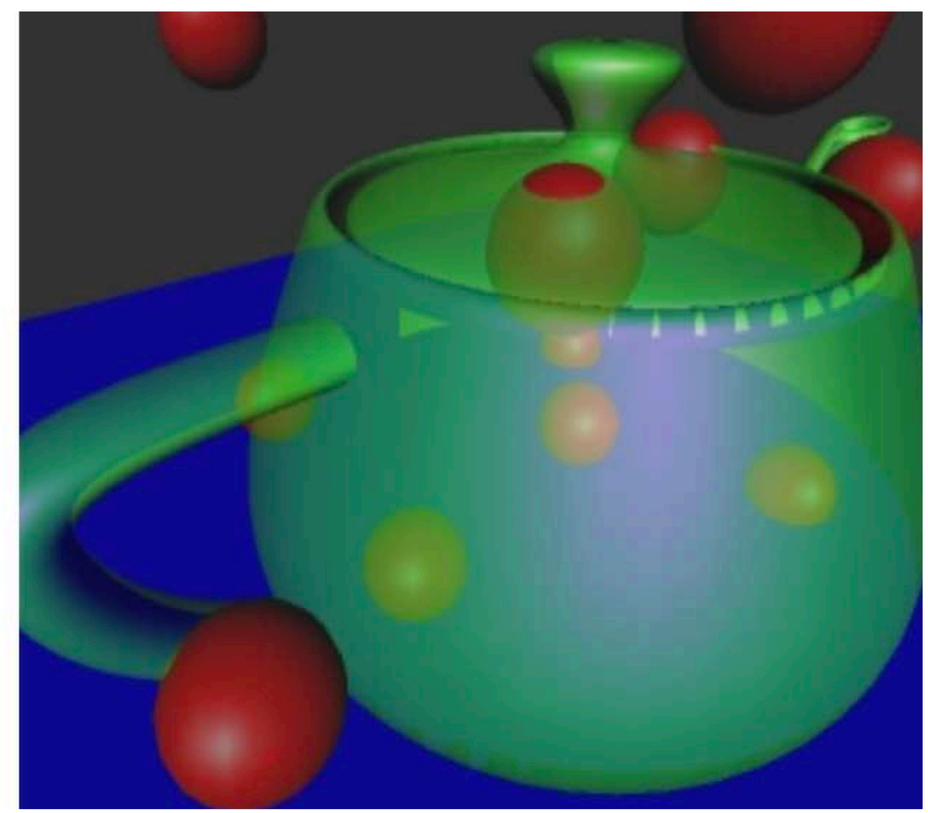




Next Time: Depth & Transparency







Depth and Transparency

Computer Graphics
CMU 15-462/15-662

Today: Wrap up the rasterization pipeline!

Remember our goal:

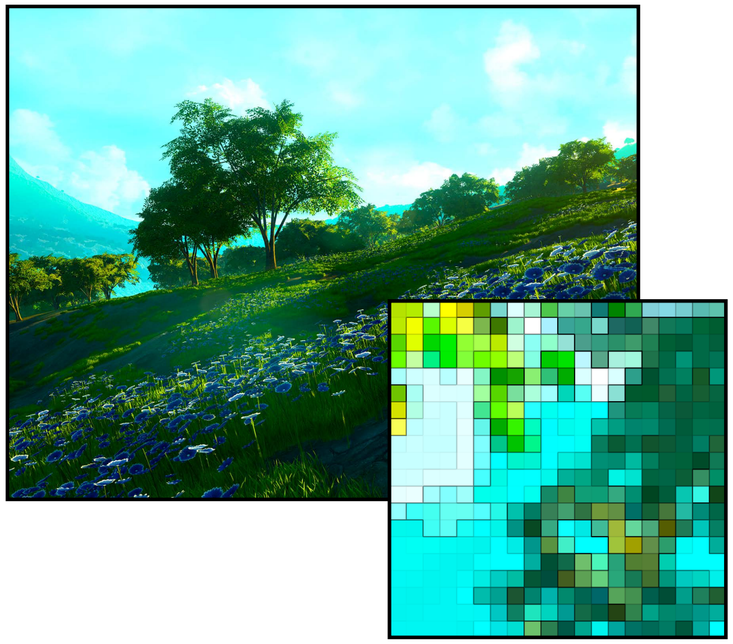
- Start with INPUTS (triangles)
  - –possibly w/ other data (e.g., colors or texture coordinates)
- Apply a series of transformations: STAGES of pipeline
- Produce OUTPUT (final image)

INPUT (TRIANGLES)

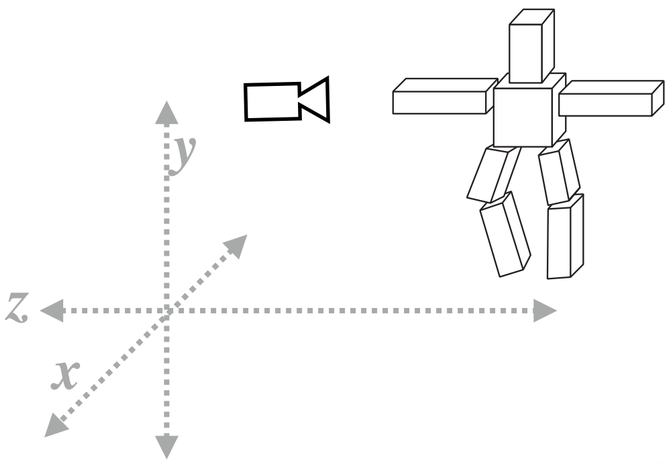
```
VERTICES
A: ( 1, 1, 1 )   E: ( 1, 1,-1 )
B: (-1, 1, 1 )   F: (-1, 1,-1 )
C: ( 1,-1, 1 )   G: ( 1,-1,-1 )
D: (-1,-1, 1 )   H: (-1,-1,-1 )

TRIANGLES
EHF, GFH, FGB, CBG,
GHC, DCH, ABD, CDB,
HED, ADE, EFA, BAF
```

RASTERIZATION PIPELINE

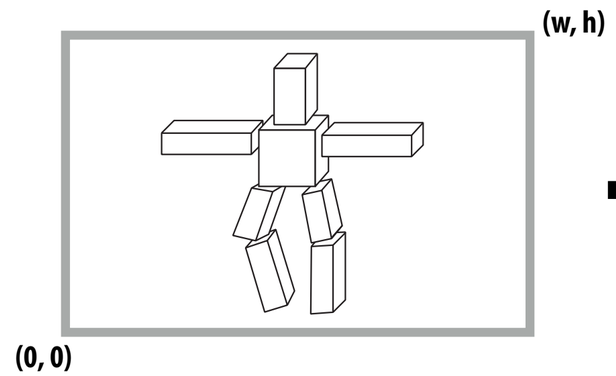
OUTPUT (BITMAP IMAGE)



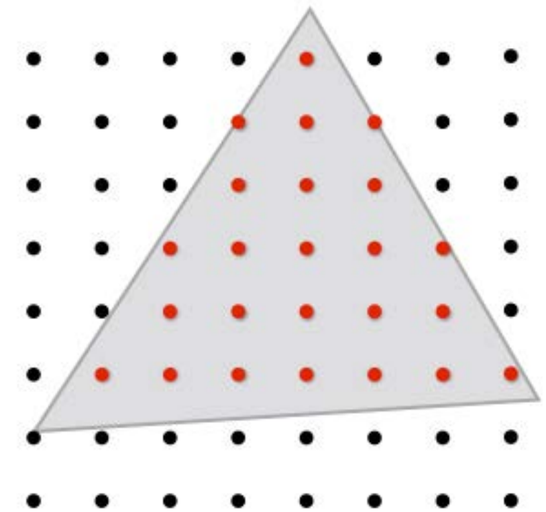
What we know how to do so far…



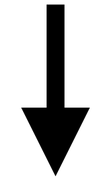
position objects in the world
(3D transformations)

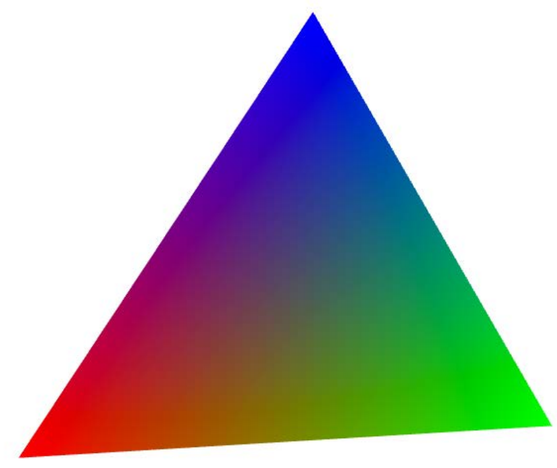


project objects onto the screen
(perspective projection)

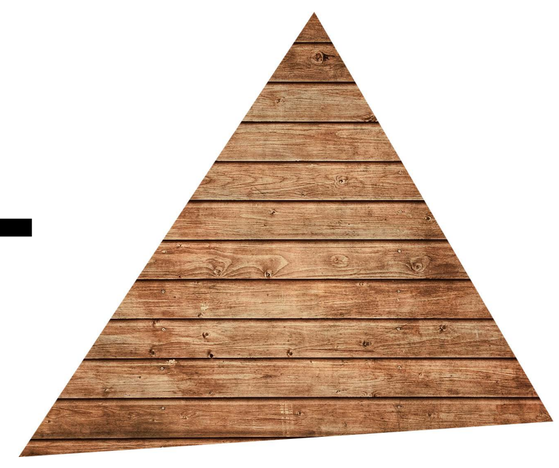


sample triangle coverage
(rasterization)

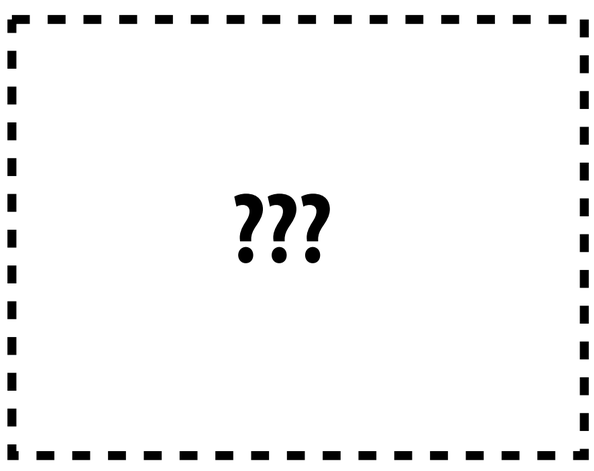




interpolate vertex attributes
(barycentric coodinates)



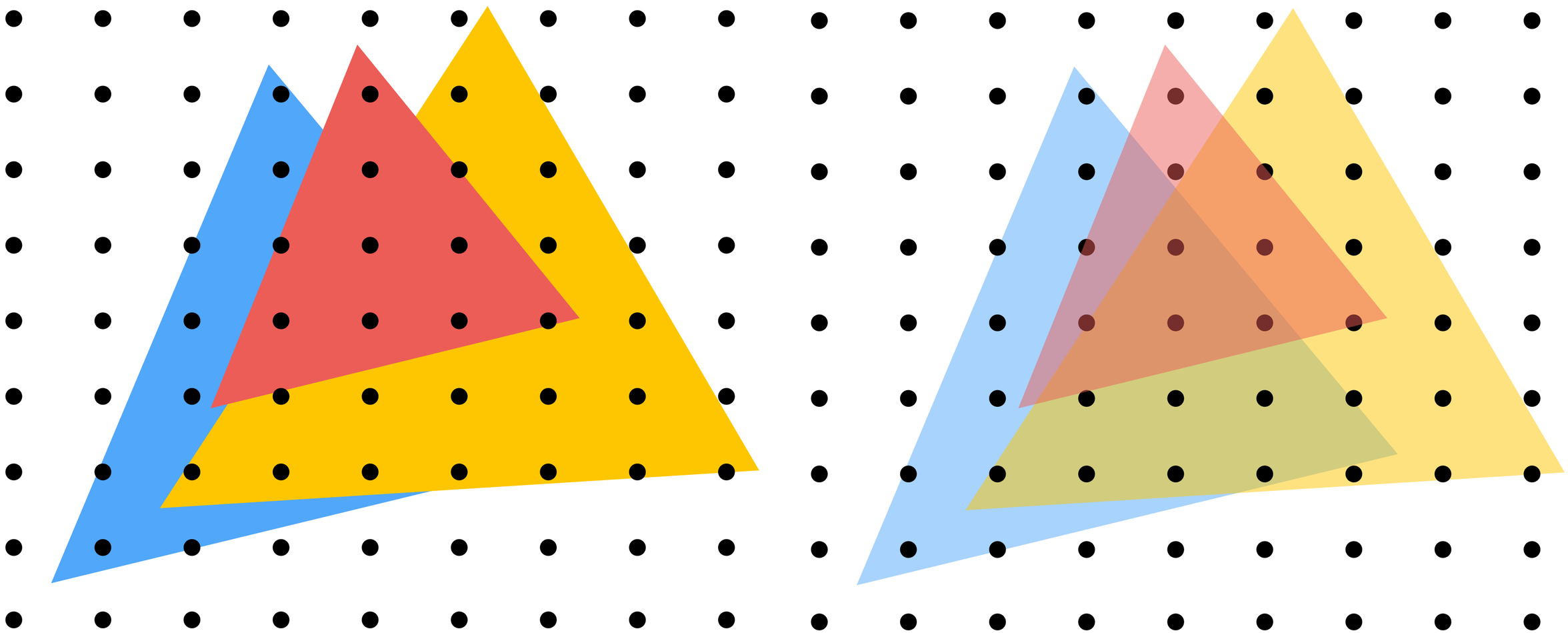
sample texture maps
(filtering, mipmapping)



put samples into frame buffer
(depth & alpha)

Occlusion

Occlusion: which triangle is visible at each covered sample point?



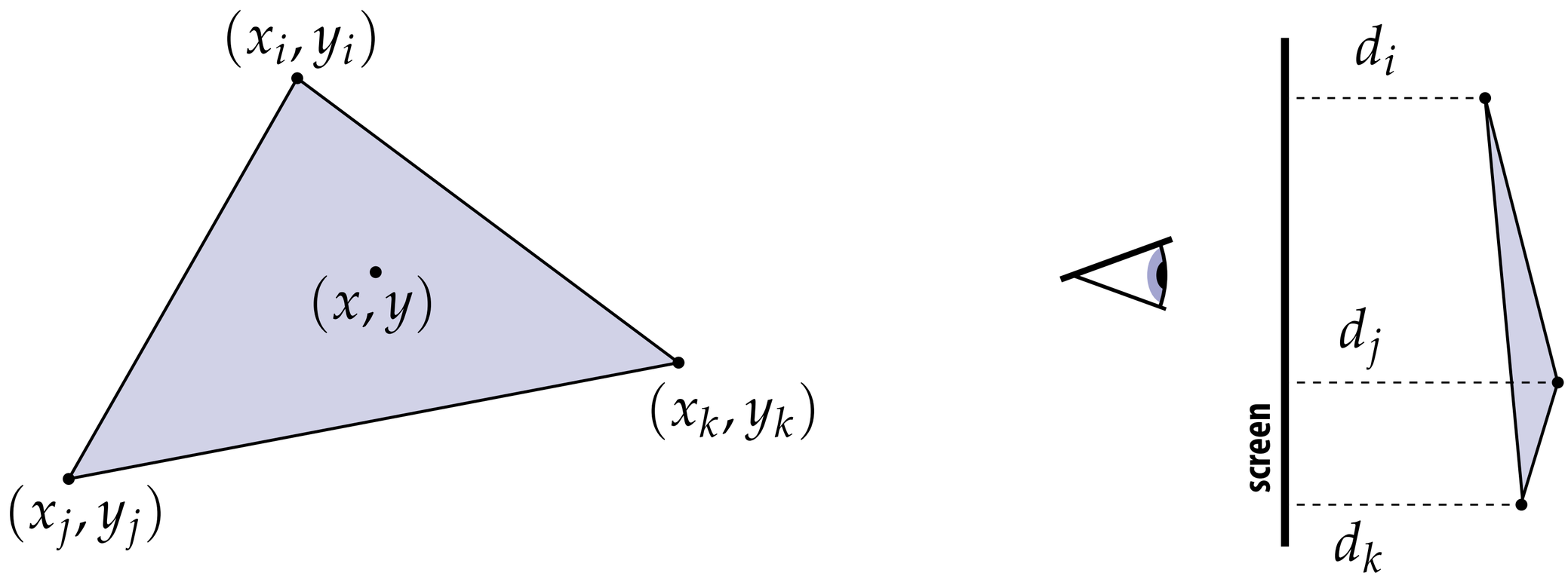
Opaque Triangles

50% transparent triangles

Sampling Depth

Assume we have a triangle given by:

- **the projected 2D coordinates (x_i, y_i) of each vertex**
- **the "depth" d_i of each vertex (i.e., distance from the viewer)**

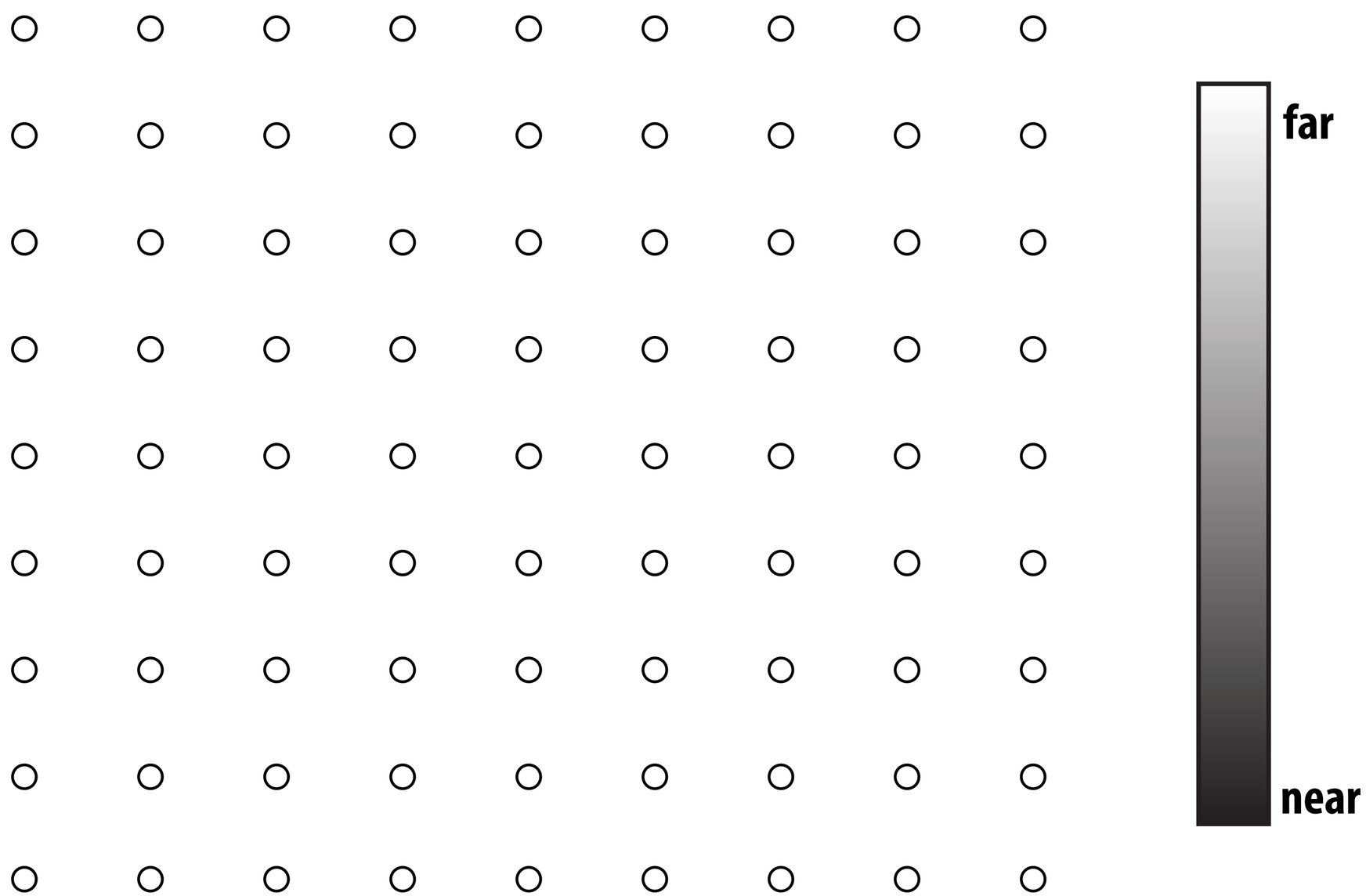


Q: How do we compute the depth d at a given sample point (x, y)?

A: Interpolate it using barycentric coordinates—just like any other attribute that varies linearly over the triangle

The depth-buffer (Z-buffer)

For each sample, *depth-buffer* stores the depth of the closest triangle seen so far

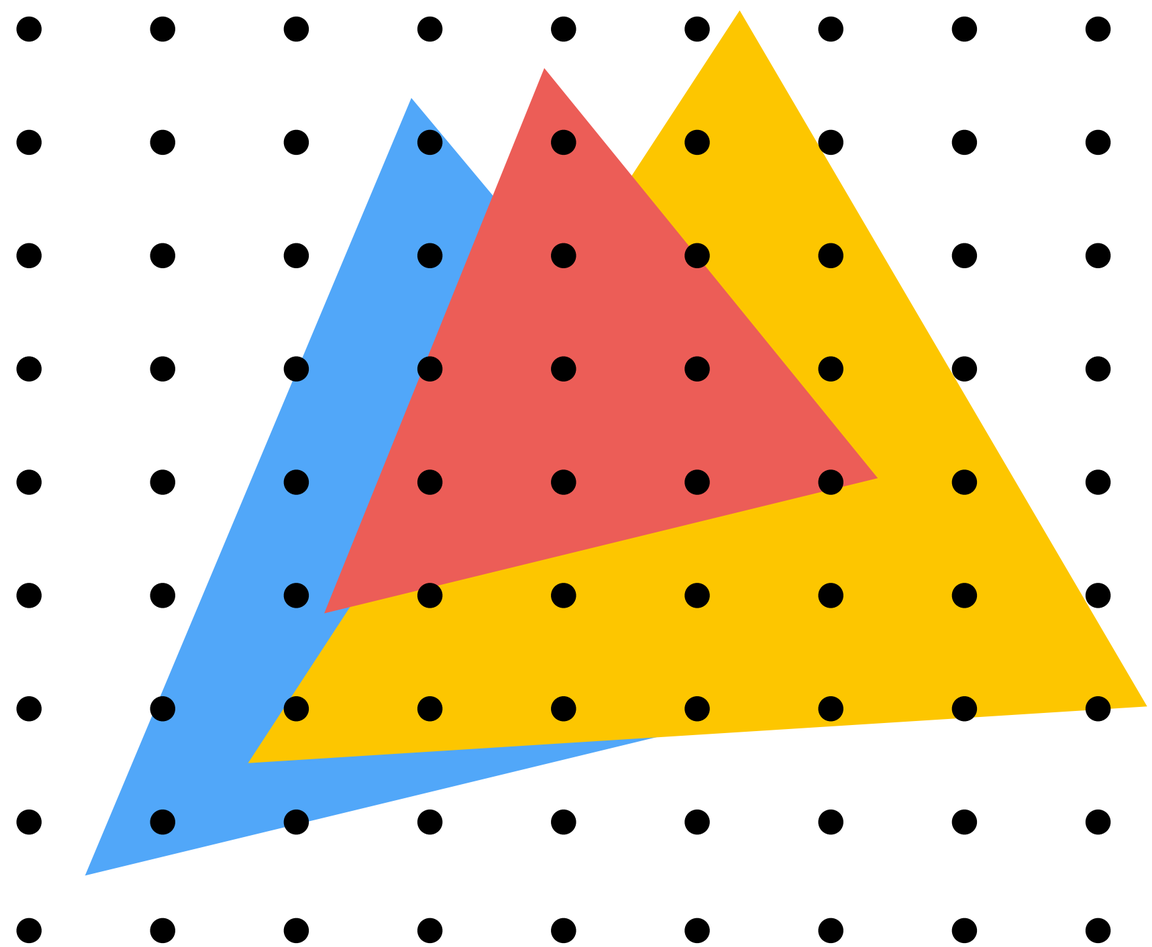


Initialize all depth buffer values to "infinity" (max value)

Depth buffer example

near far

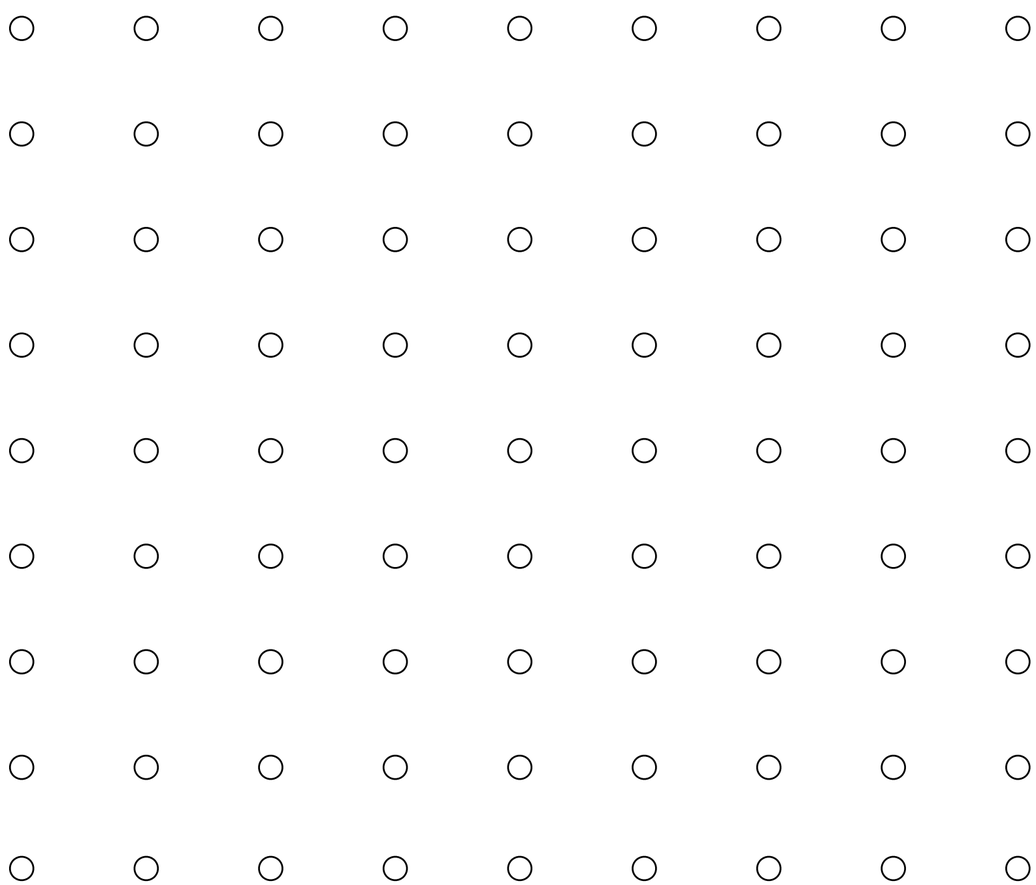
Example: rendering three opaque triangles



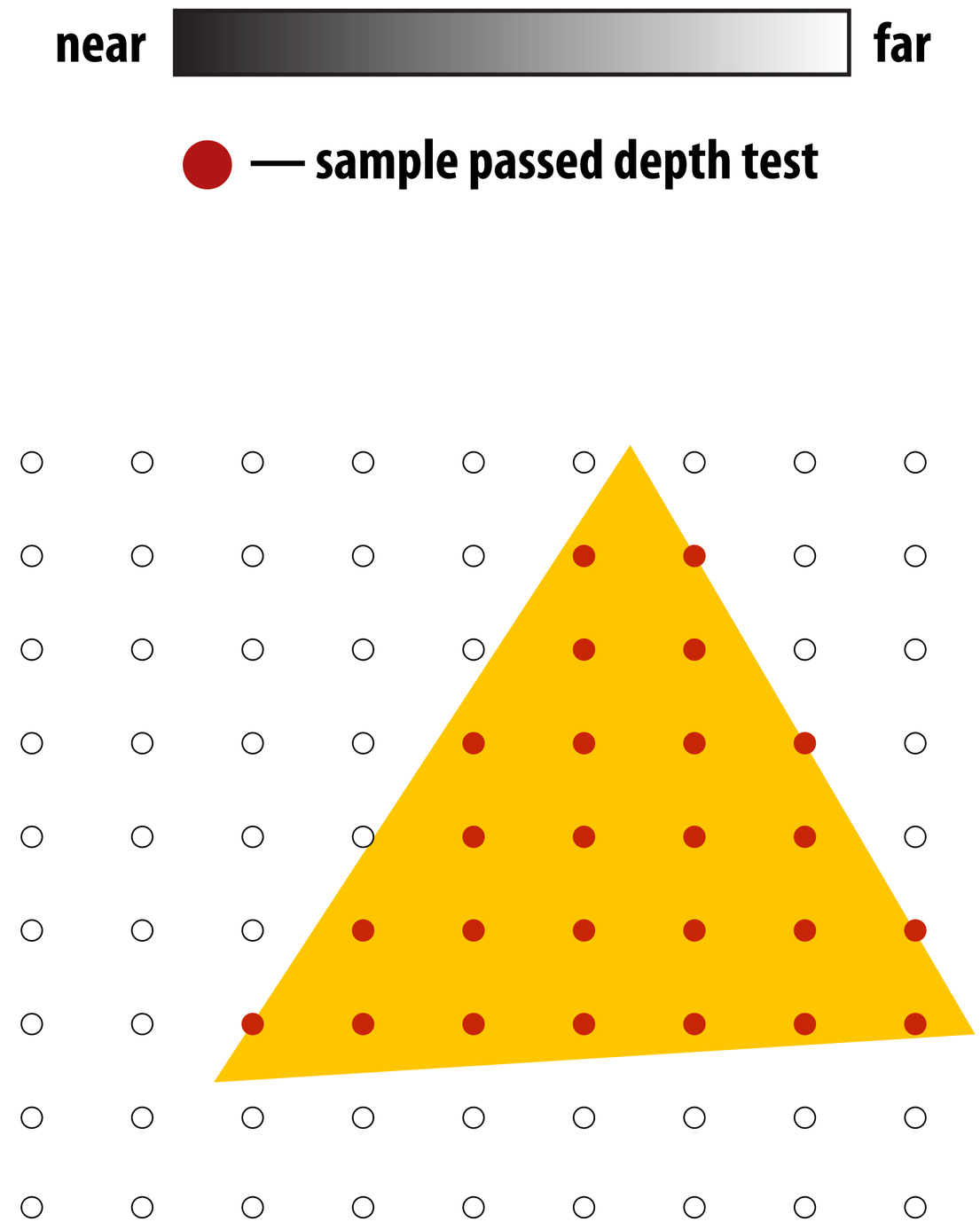
Occlusion using the depth-buffer (Z-buffer)

Processing yellow triangle:

depth = 0.5



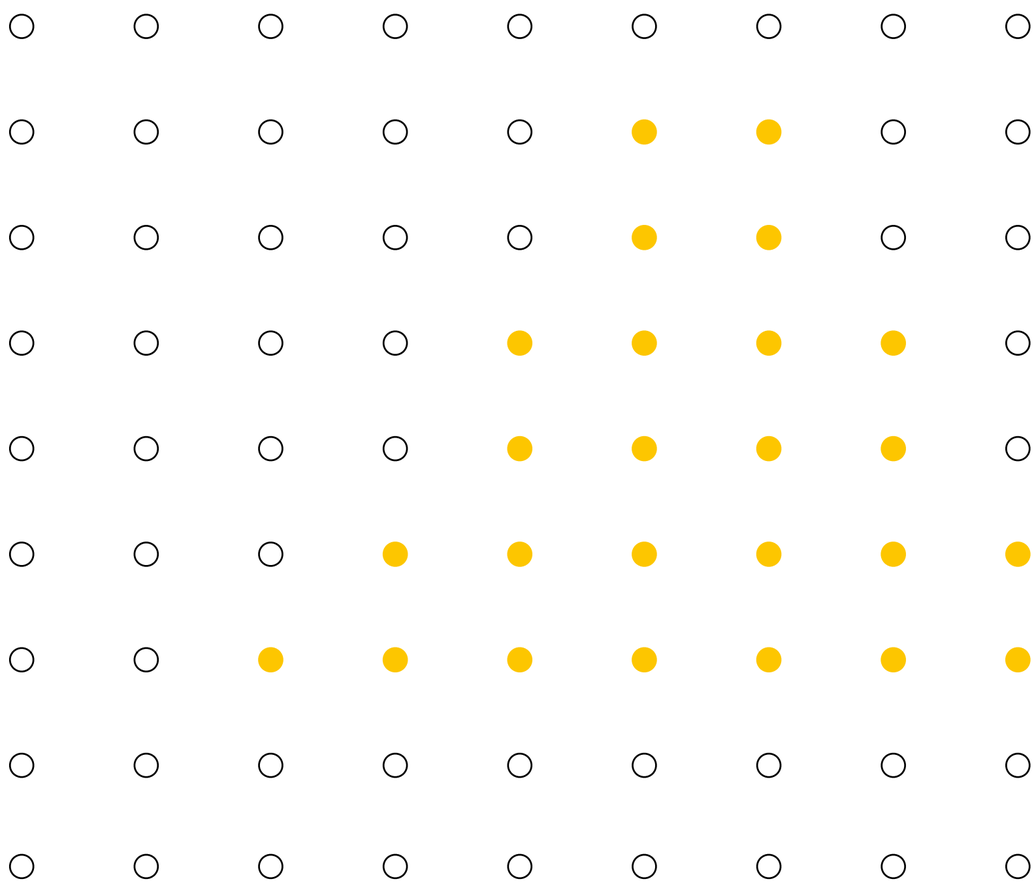
Color buffer contents



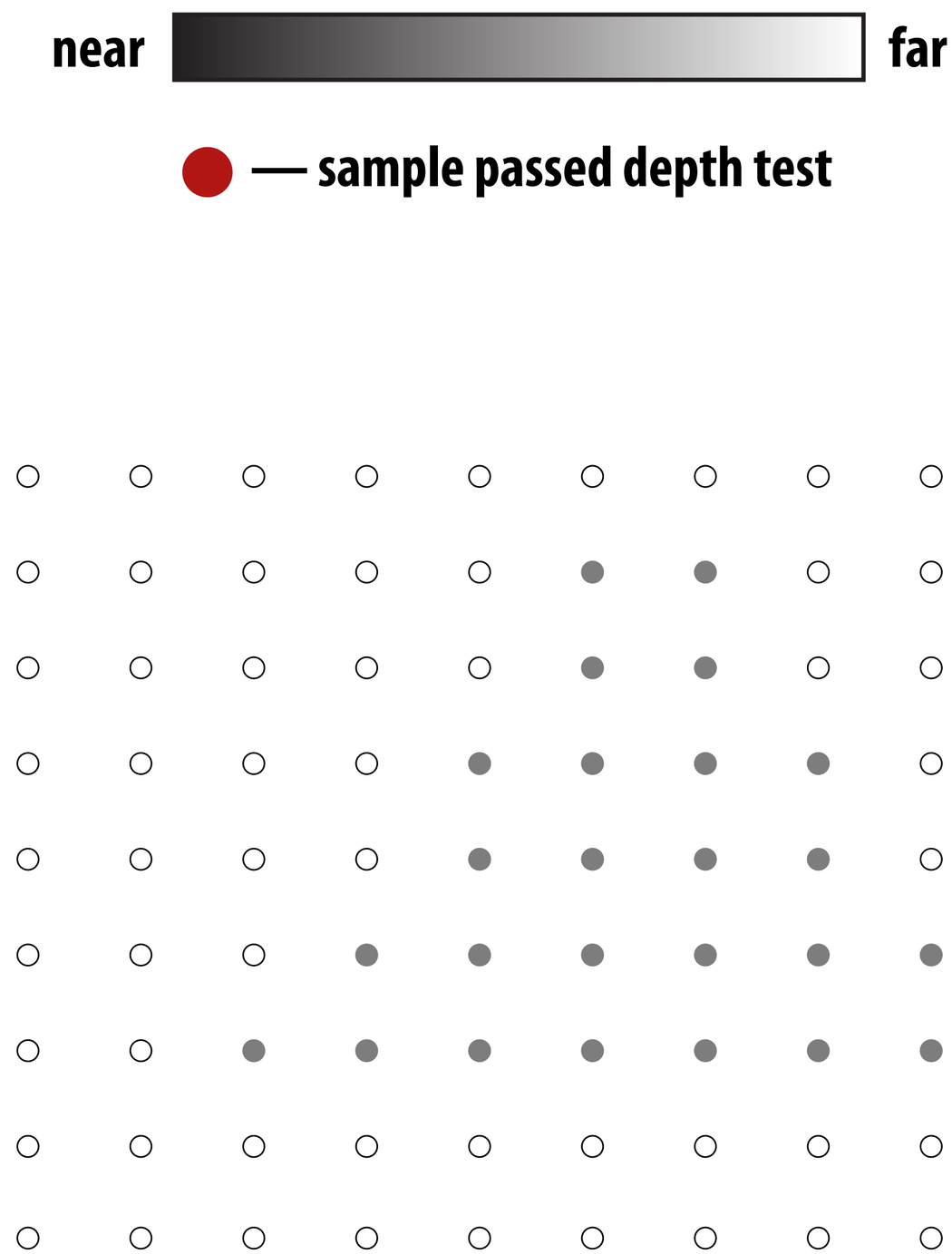
Depth buffer contents

Occlusion using the depth-buffer (Z-buffer)

After processing yellow triangle:



Color buffer contents

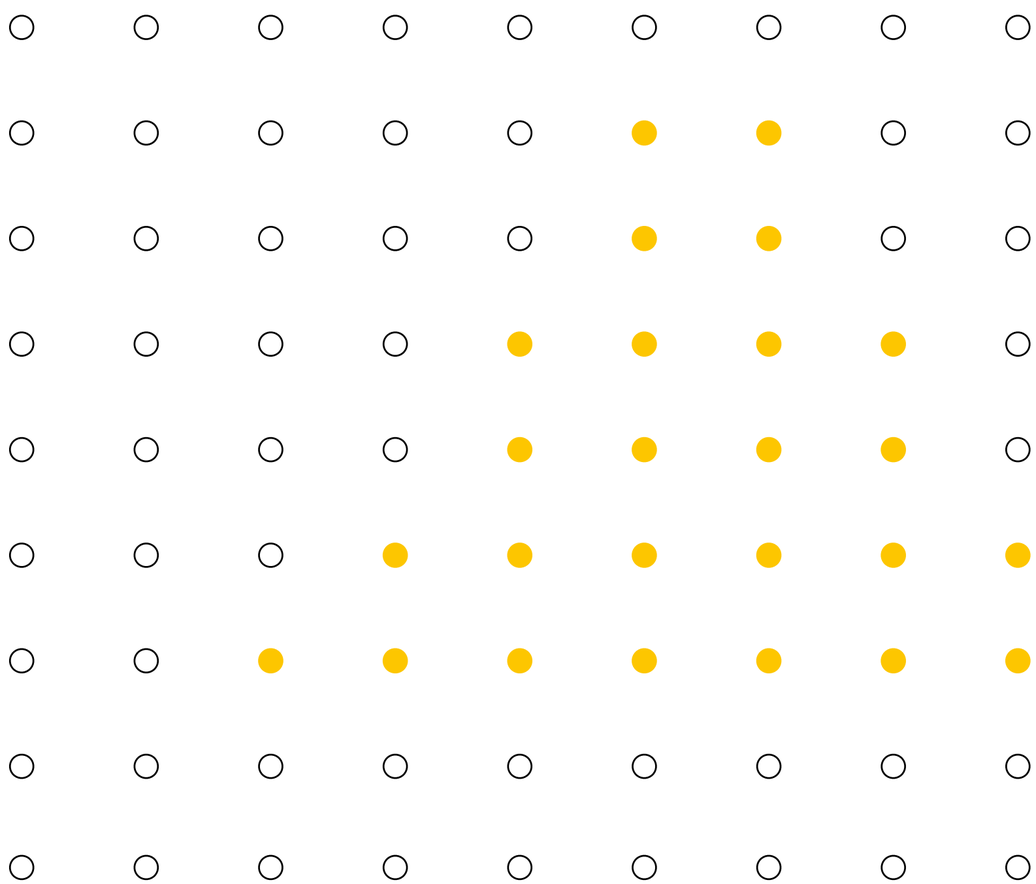


Depth buffer contents

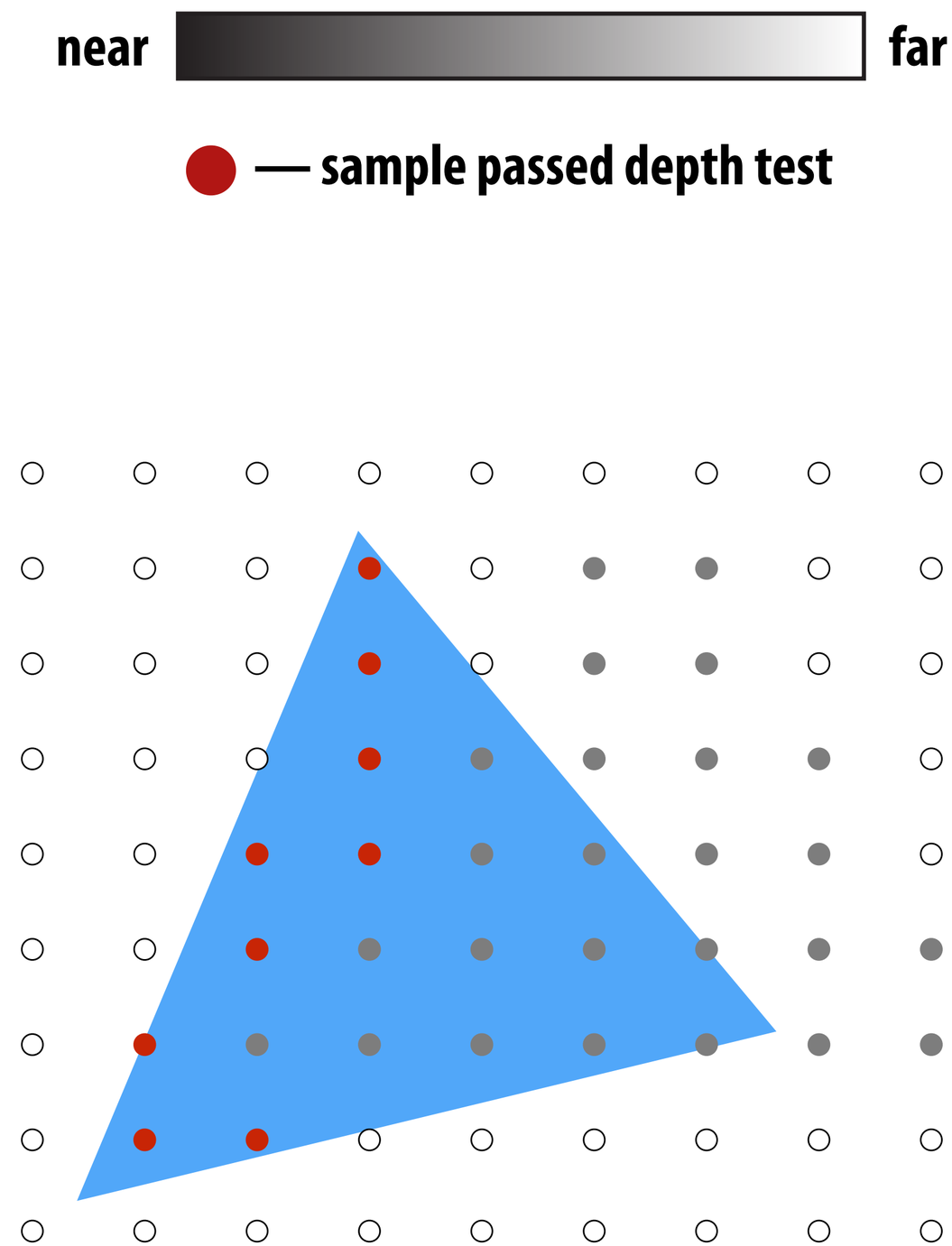
Occlusion using the depth-buffer (Z-buffer)

Processing blue triangle:

depth = 0.75



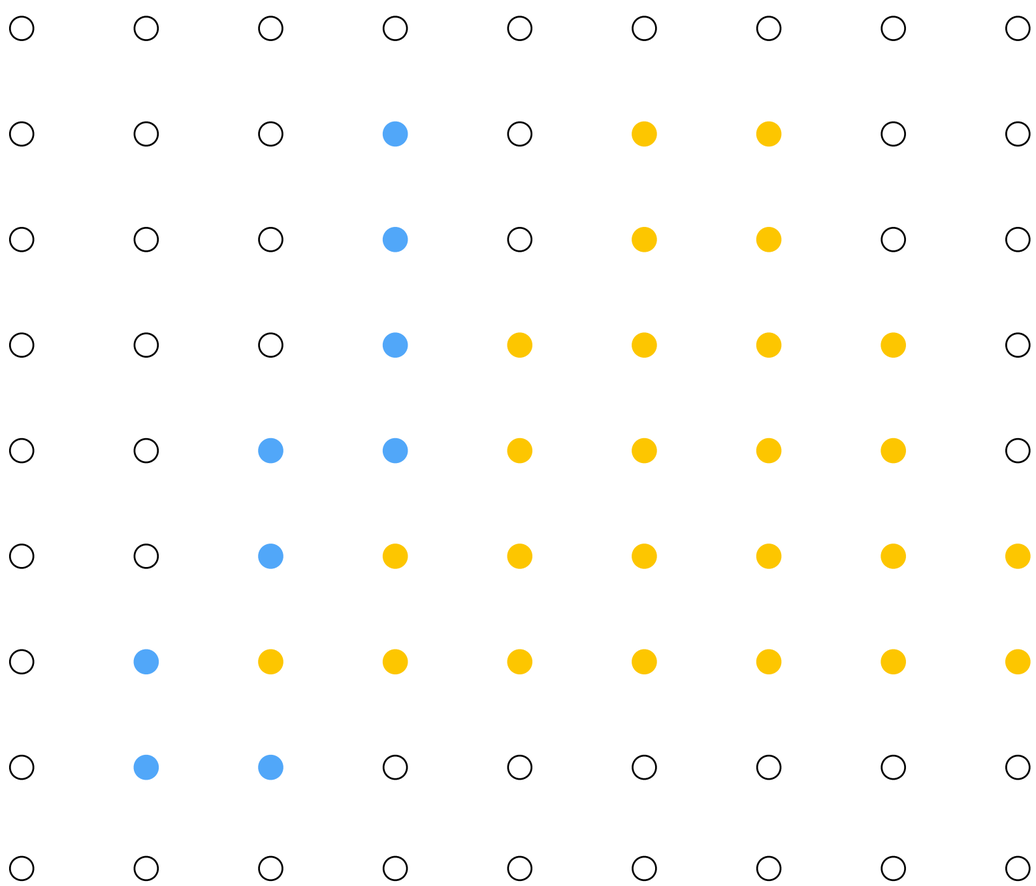
Color buffer contents



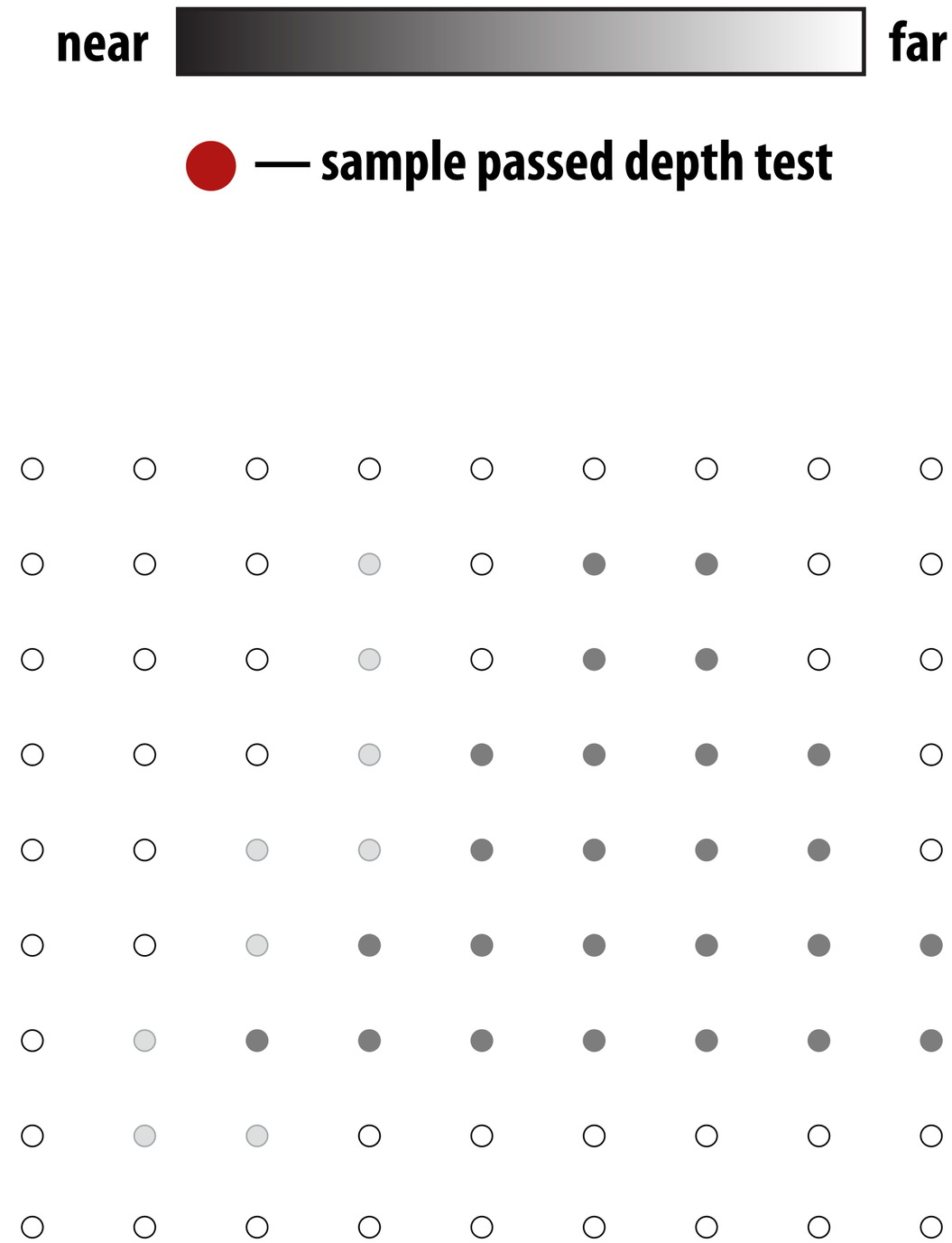
Depth buffer contents

Occlusion using the depth-buffer (Z-buffer)

After processing blue triangle:



Color buffer contents

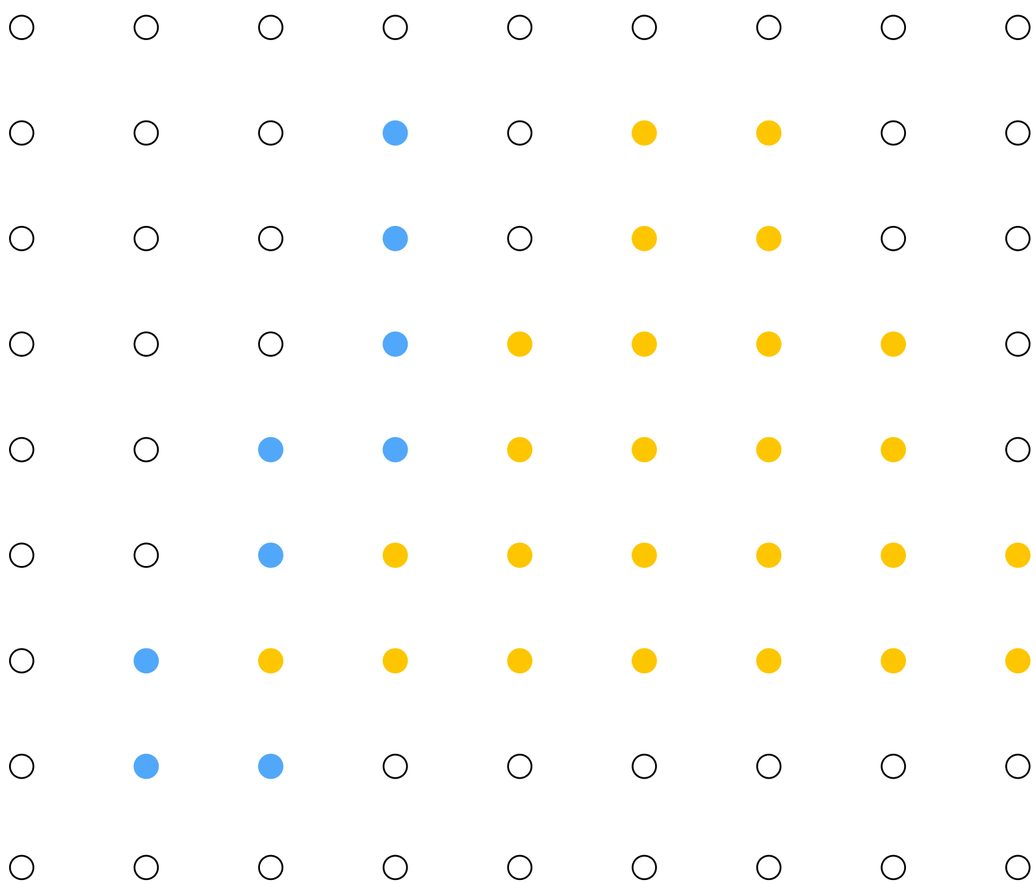


Depth buffer contents

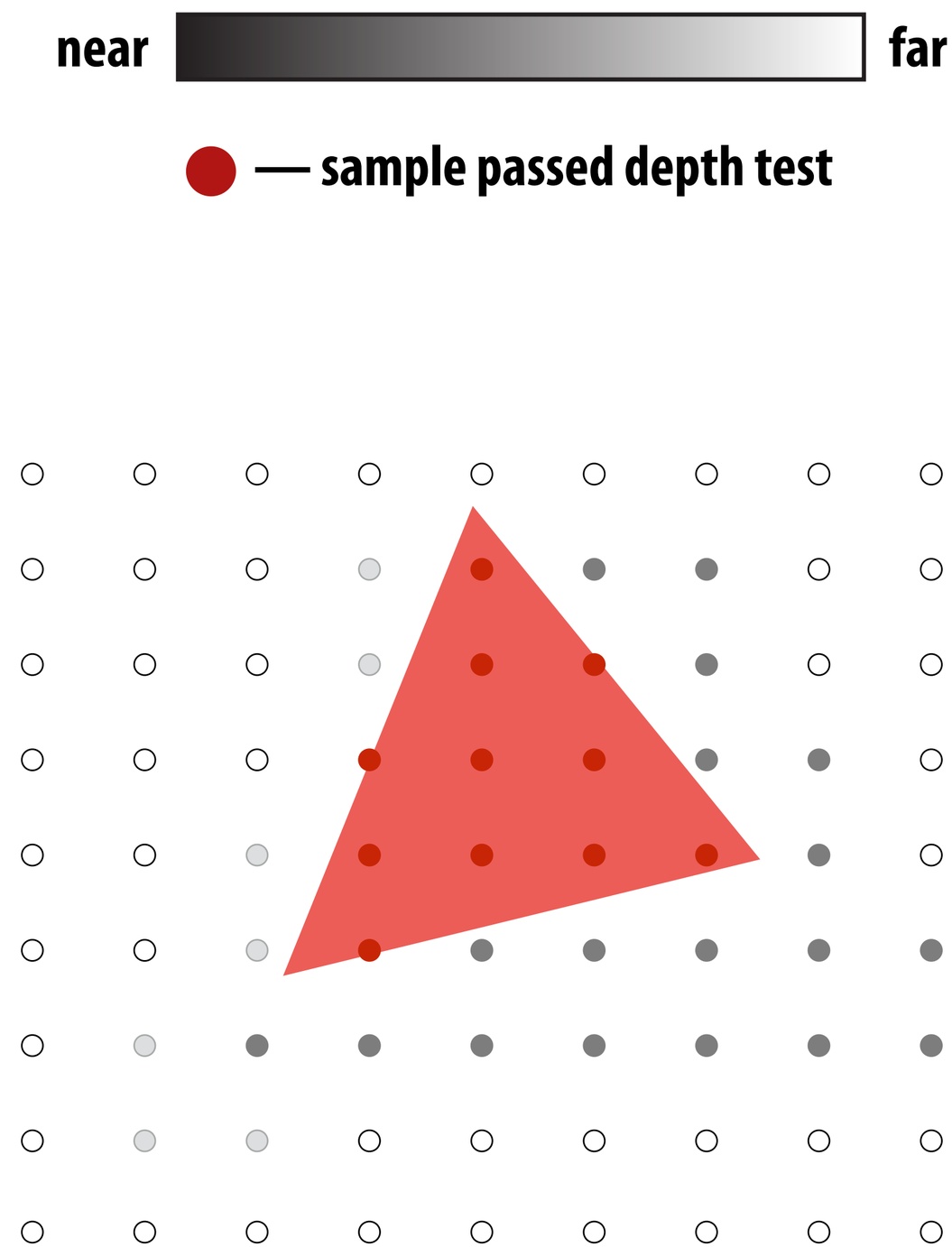
Occlusion using the depth-buffer (Z-buffer)

Processing red triangle:

depth = 0.25



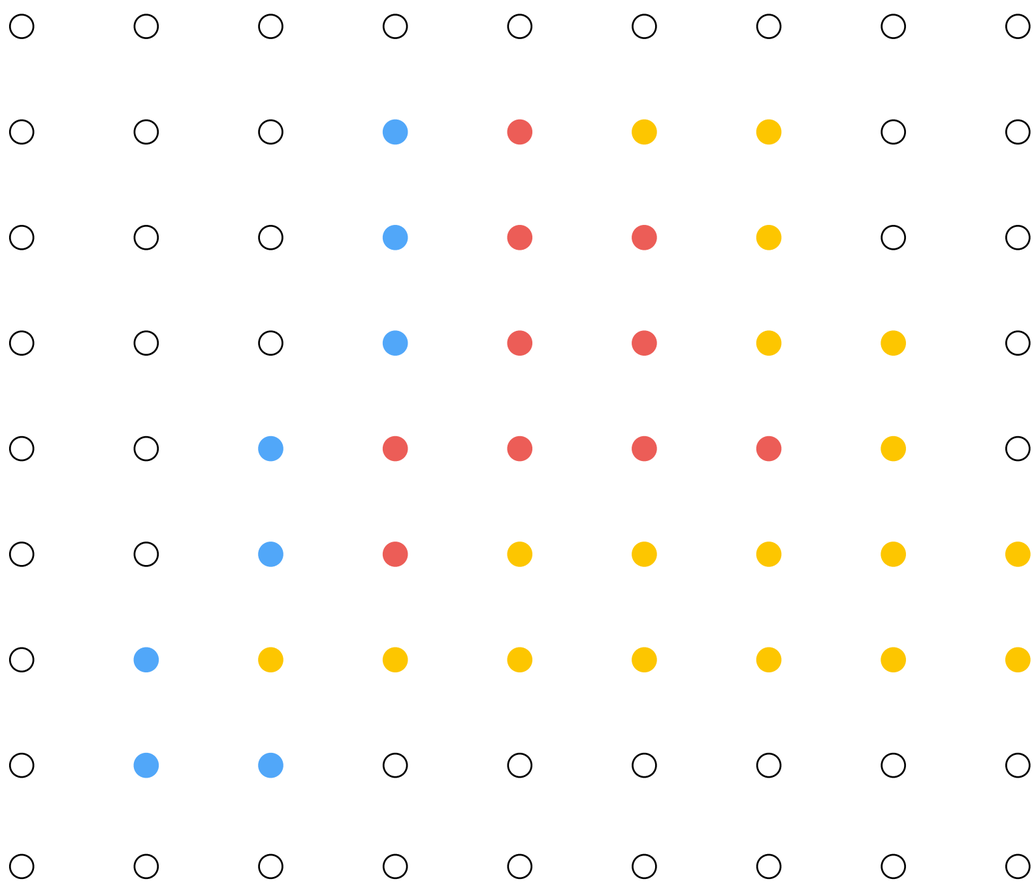
Color buffer contents



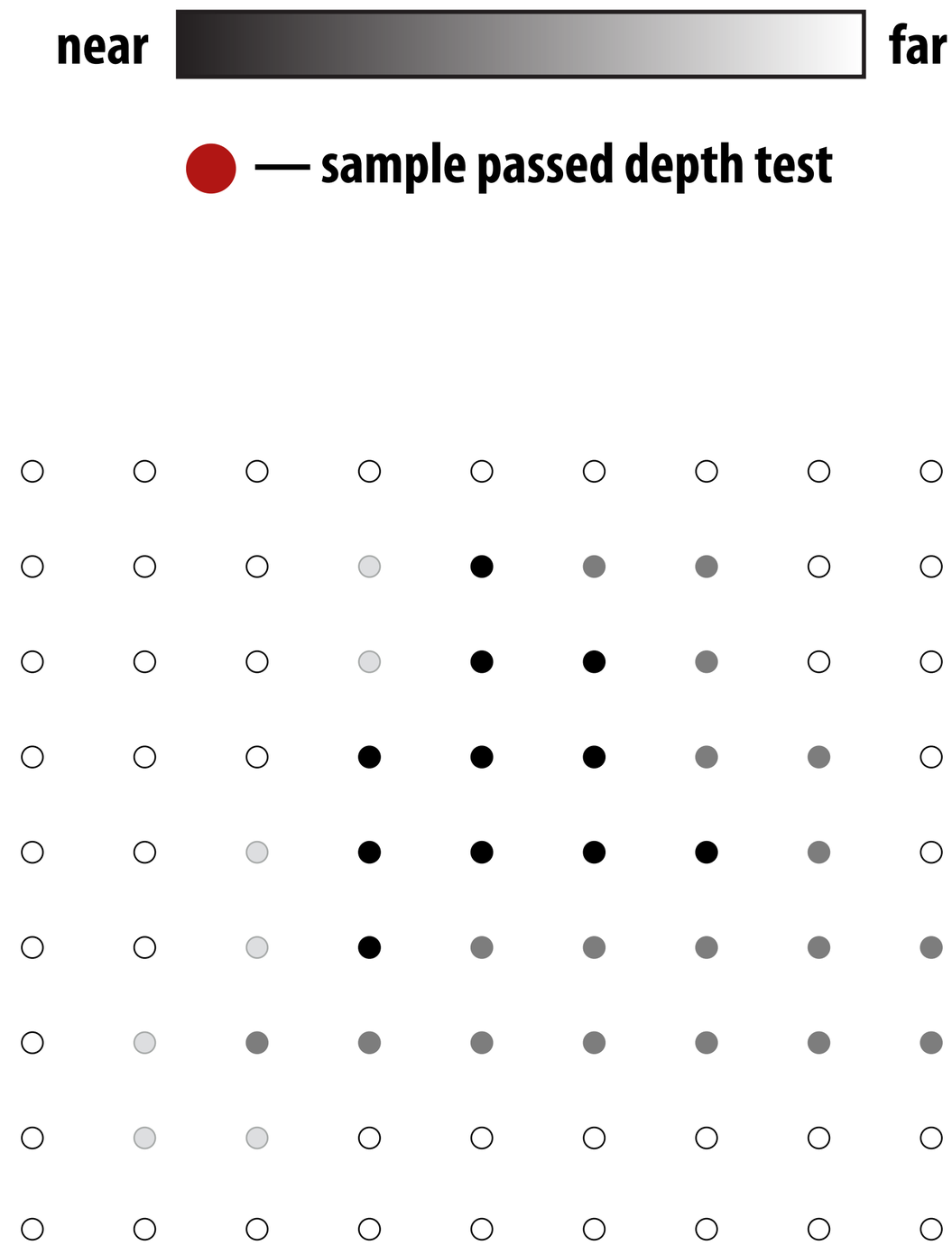
Depth buffer contents

Occlusion using the depth-buffer (Z-buffer)

After processing red triangle:



Color buffer contents



Depth buffer contents

Occlusion using the depth buffer

```
bool pass_depth_test(d1, d2)
{
    return d1 < d2;
}
```

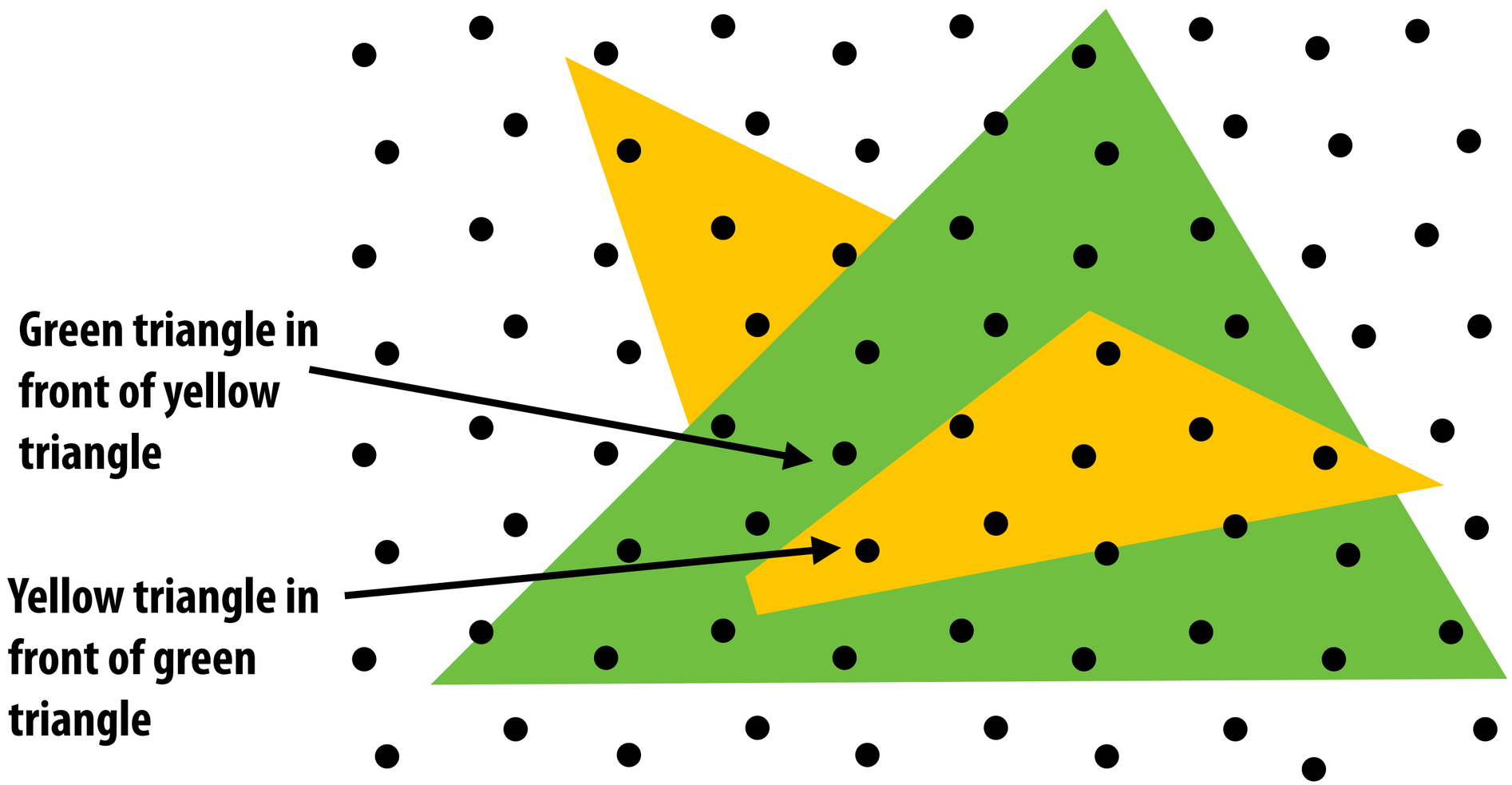
```
draw_sample(x, y, d, c) //new depth d & color c at (x,y)
{
  if( pass_depth_test( d, zbuffer[x][y] ))
  {
    // triangle is closest object seen so far at this
    // sample point. Update depth and color buffers.
    zbuffer[x][y] = d;  // update zbuffer
    color[x][y] = c;    // update color buffer
  }
  // otherwise, we've seen something closer already;
  // don't update color or depth
}
```

Depth + Intersection

Q: Does depth-buffer algorithm handle interpenetrating surfaces?

A: Of course!

Occlusion test is based on depth of triangles <u>at a given sample point</u>.
Relative depth of triangles may be different at different sample points.

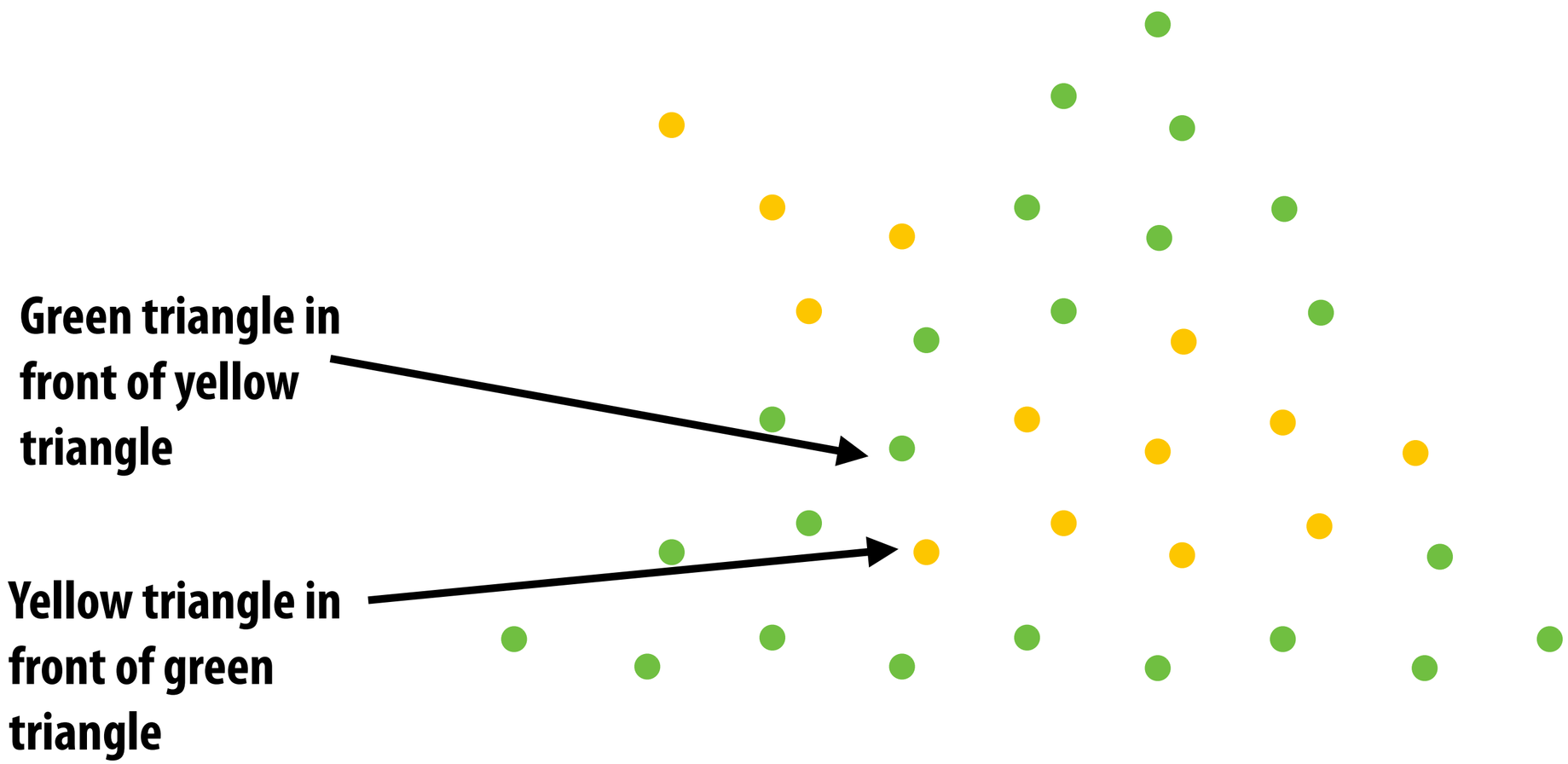


Intersection

Q: Does depth-buffer algorithm handle interpenetrating surfaces?

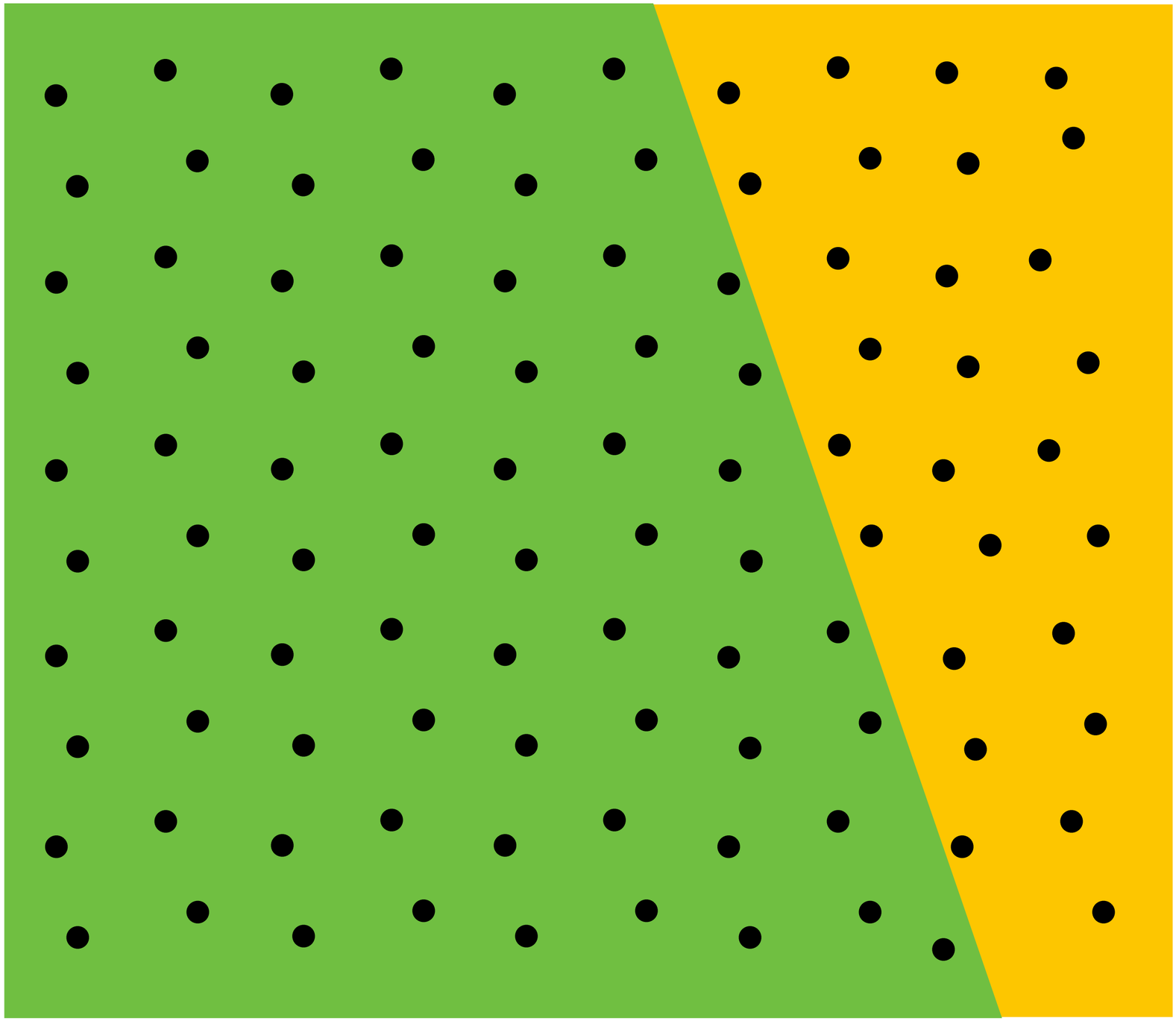
A: Of course!

Occlusion test is based on depth of triangles <u>at a given sample point</u>.
Relative depth of triangles may be different at different sample points.



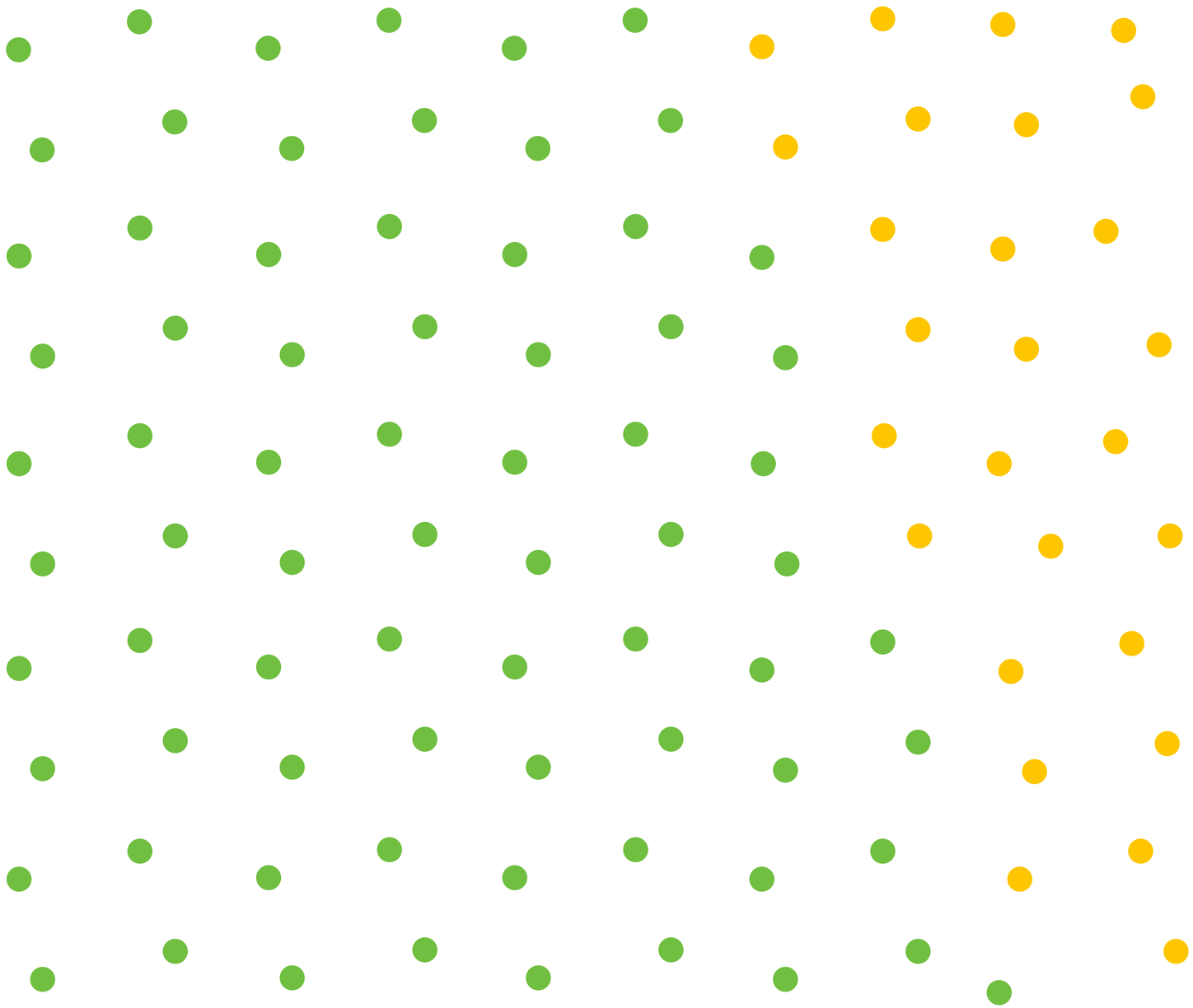
Depth + Supersampling

Q: Does depth buffer work with super sampling? **A: Yes! If done per (super) sample.**



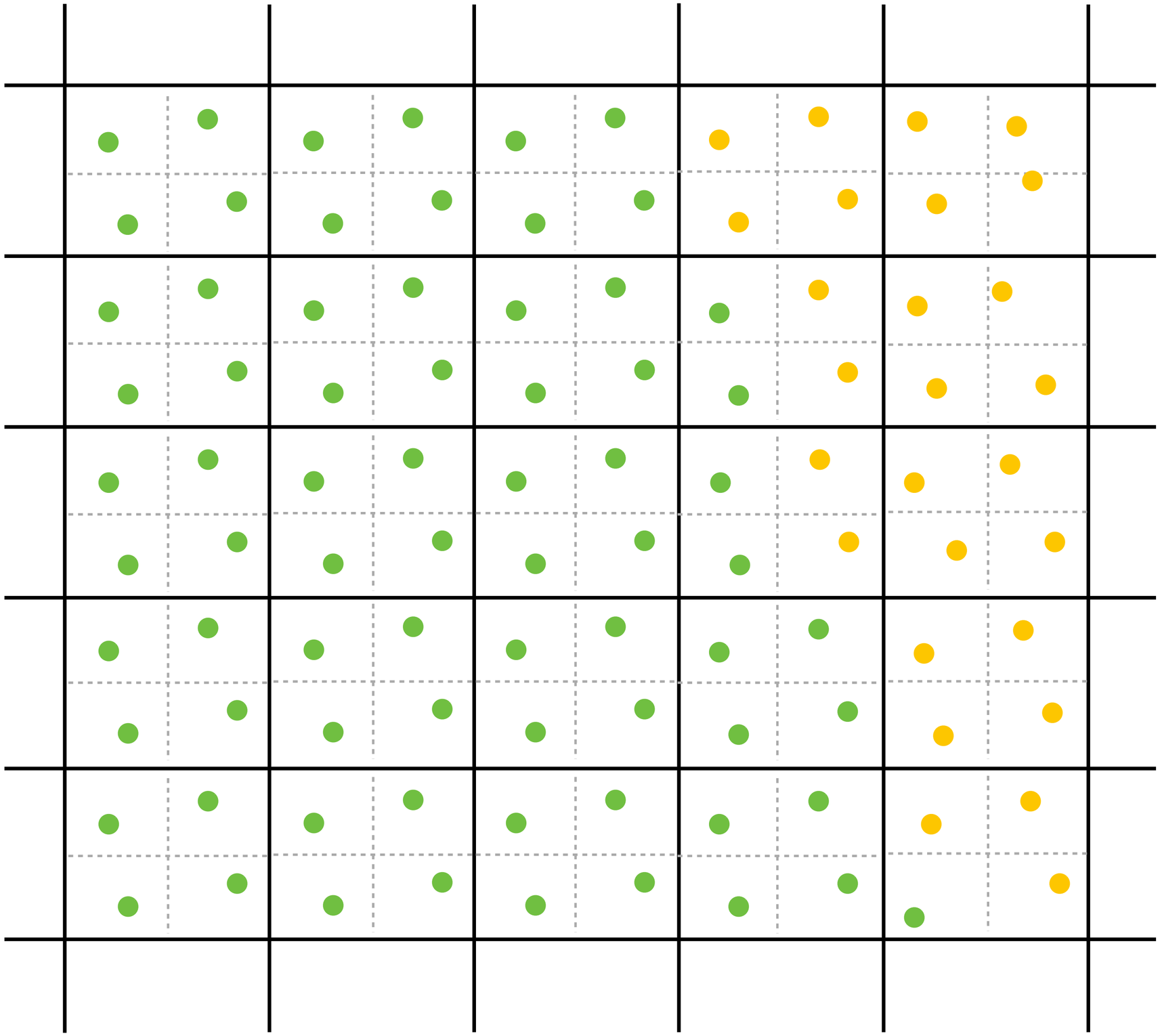
(Here: green triangle occludes yellow triangle)

Depth + Supersampling

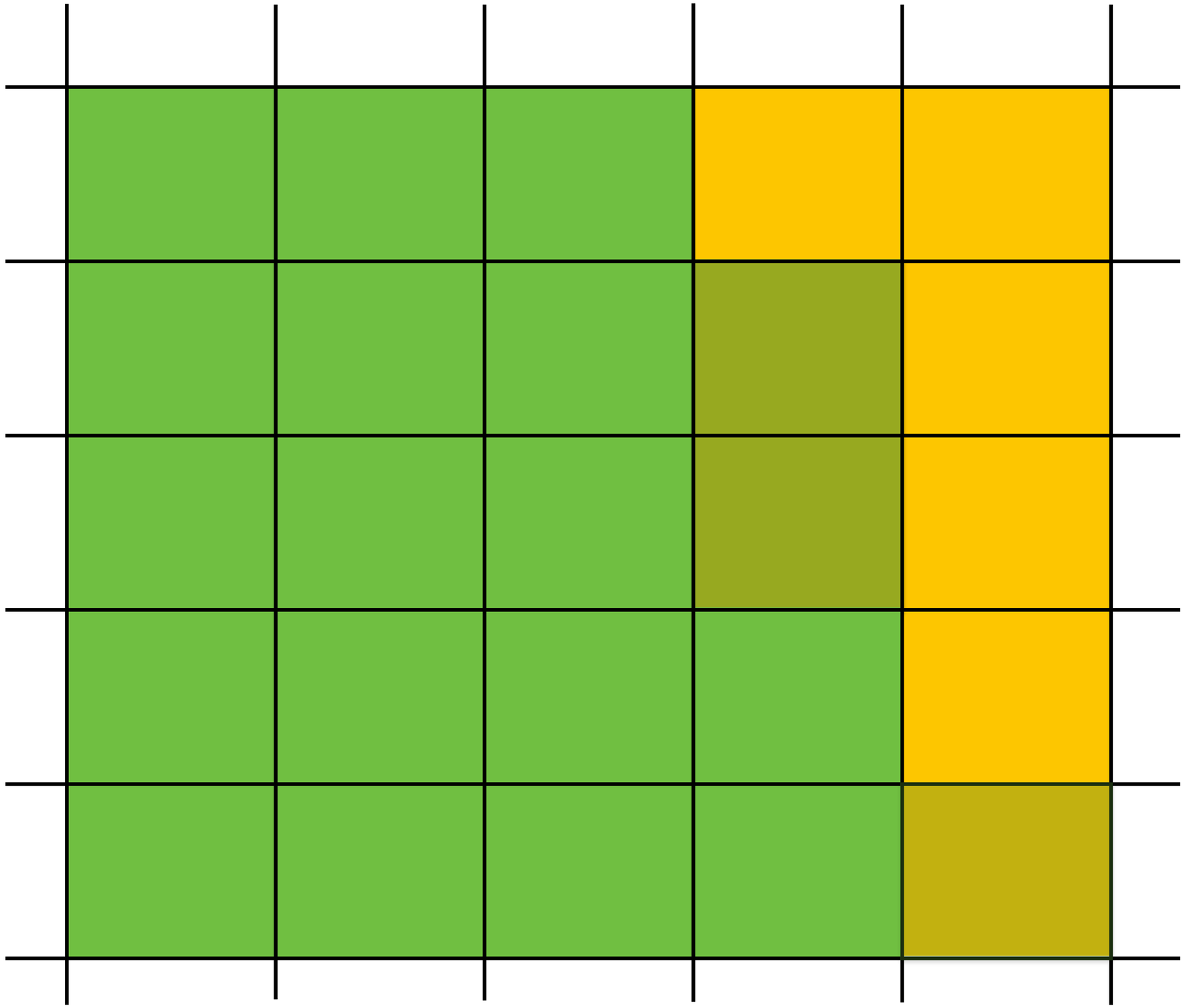


Color of super samples after rasterizing w/ depth buffer

Color buffer contents (4 samples per pixel)



Final resampled result



Note anti-aliasing of edge due to filtering of green and yellow samples

Summary: occlusion using a depth buffer

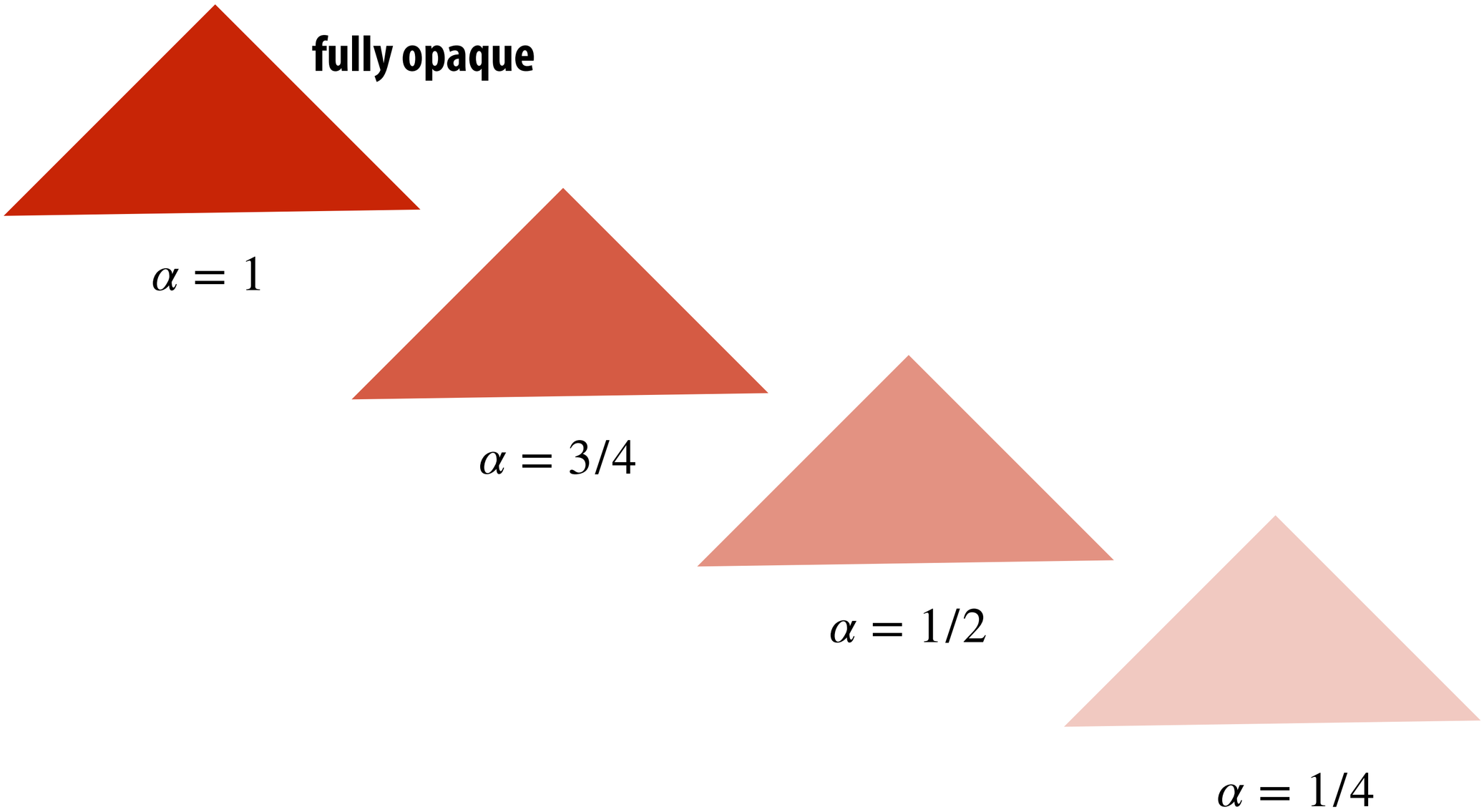
- Store one depth value per (super) sample—not one per pixel!
- Constant additional space per sample
  - Hence, constant space for depth buffer
  - Doesn't depend on number of overlapping primitives!
- Constant time occlusion test per covered sample
  - Read-modify write of depth buffer if "pass" depth test
  - Just a read if "fail"
- Not specific to triangles: only requires that surface depth can be evaluated at a screen sample point

But what about semi-transparent surfaces?

Compositing

Representing opacity as alpha

An "alpha" value $0 \leq \alpha \leq 1$ describes the <u>opacity</u> of an object

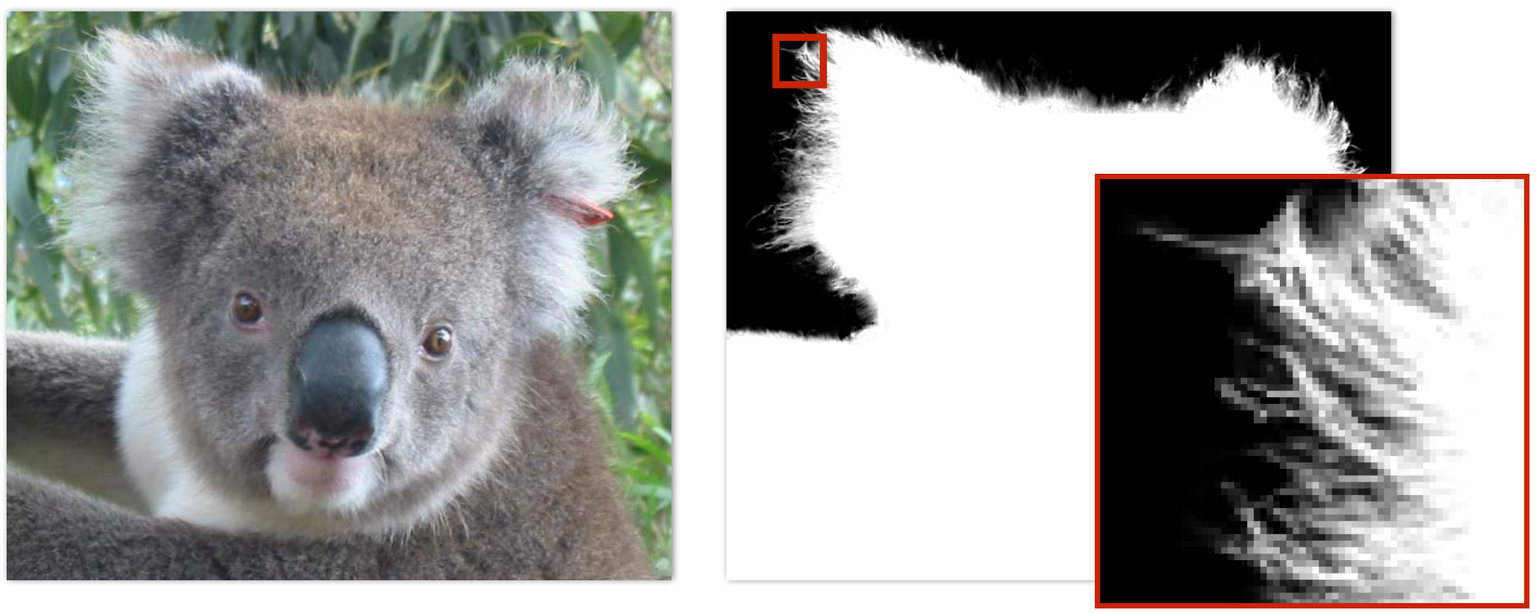


$\alpha = 0$

fully transparent

Alpha channel of an image

color channels | **α channel**

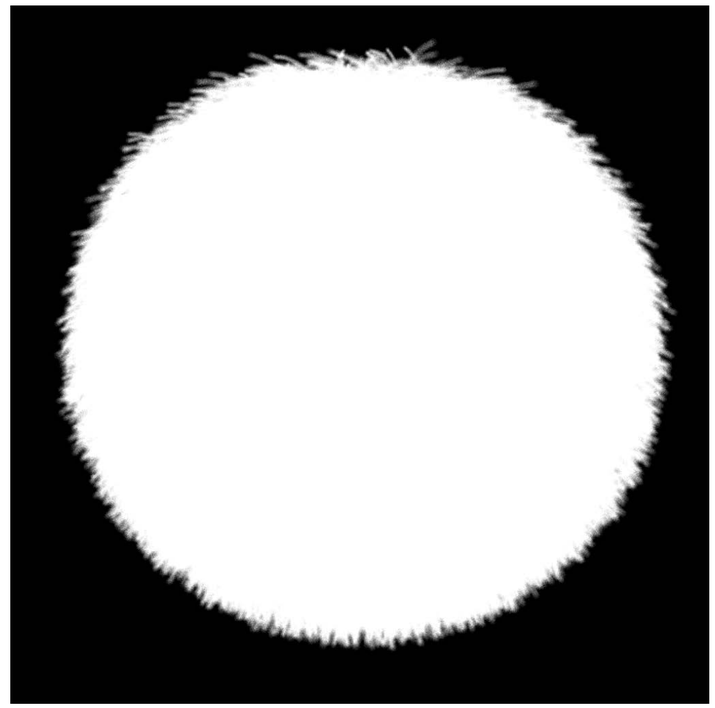


Key idea: can use α channel to composite one image on top of another.

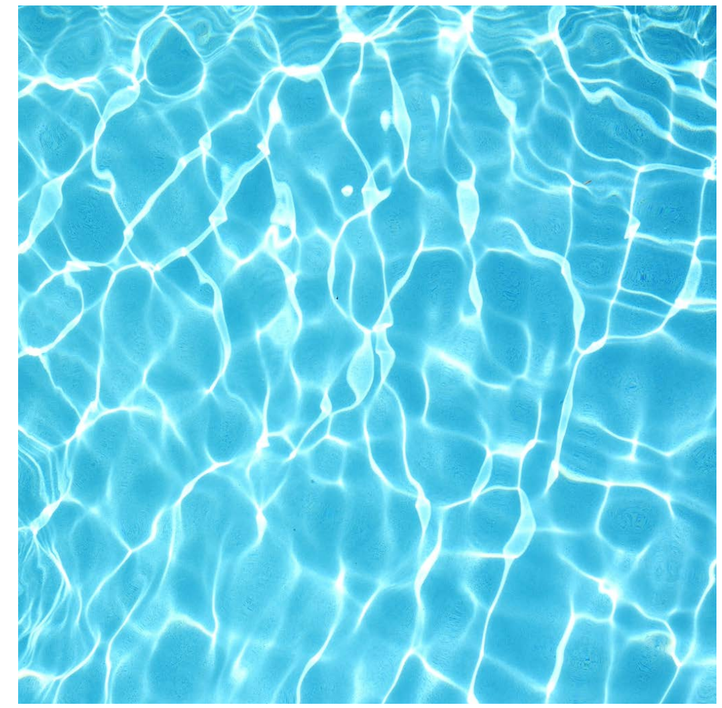
Fringing

Poor treatment of color/alpha can yield dark "fringing":

foreground color



foreground alpha

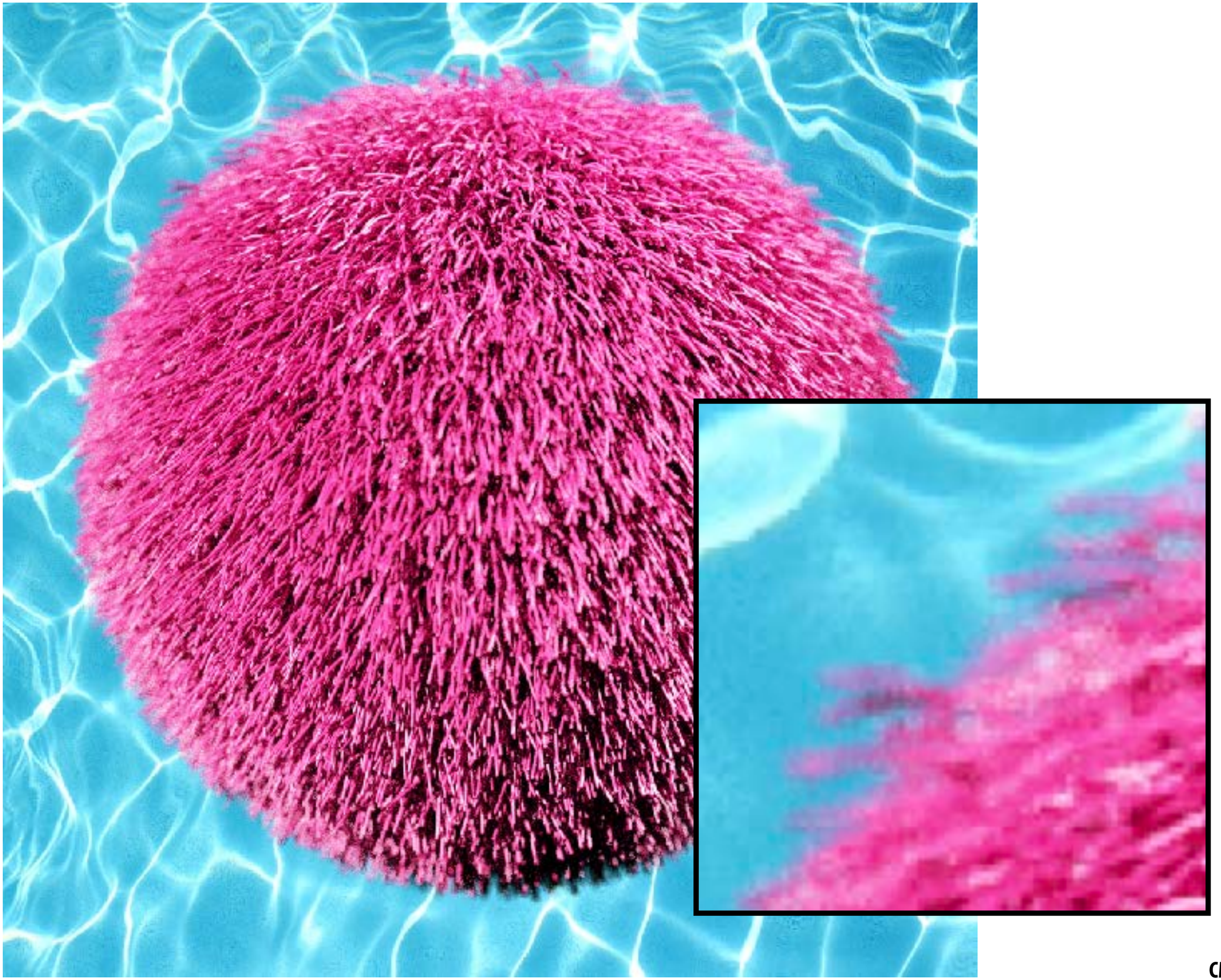


background color

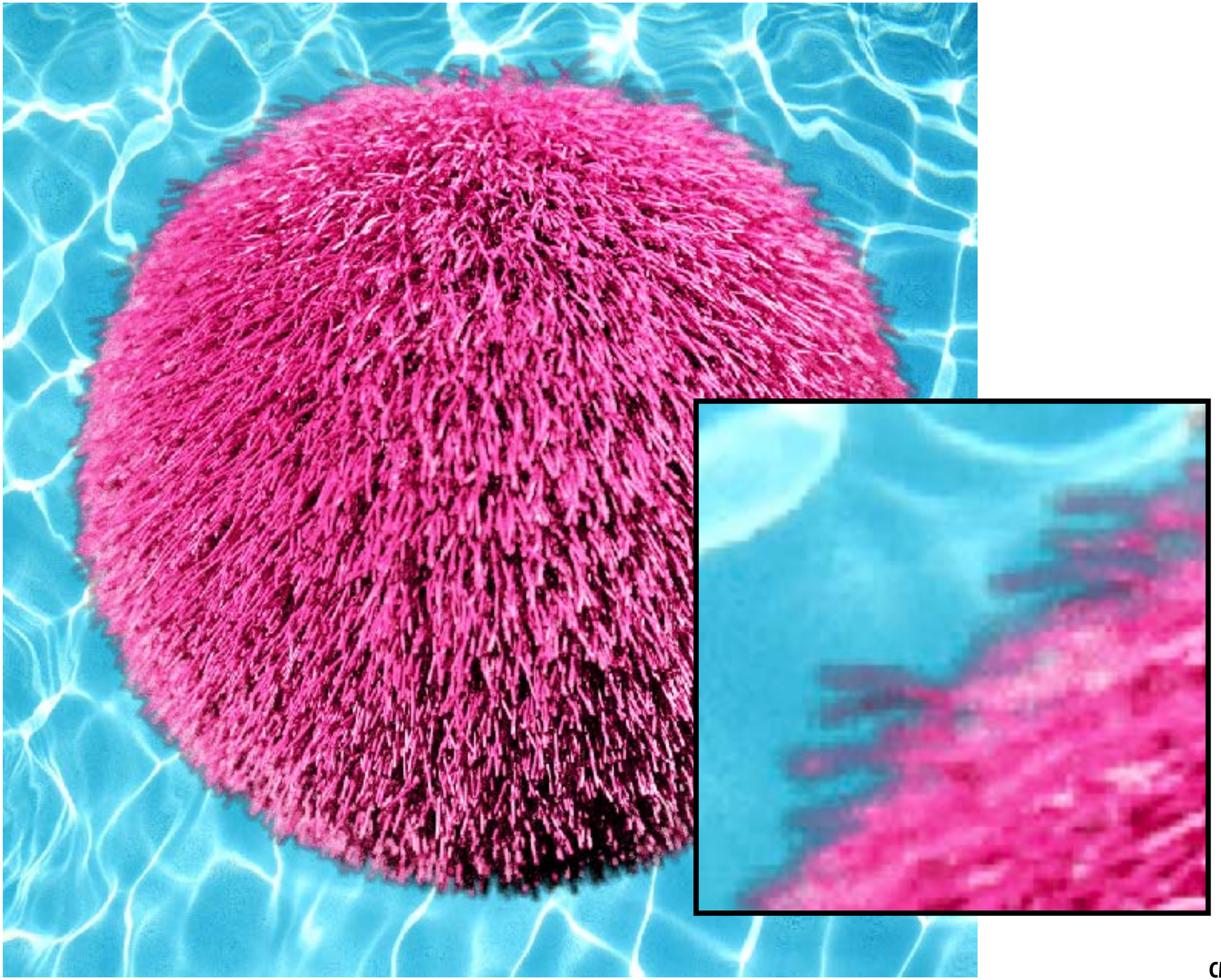
fringing

no fringing

No fringing



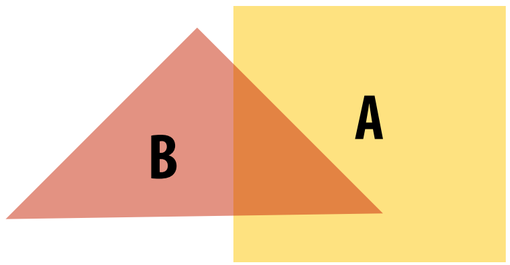
Fringing (…why does this happen?)



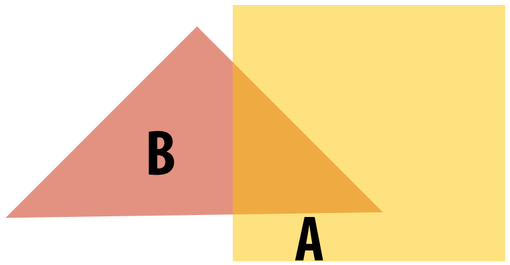
Over operator:

Composites image B with opacity α_B <u>over</u> image A with opacity α_A

Informally, captures behavior of "tinted glass"



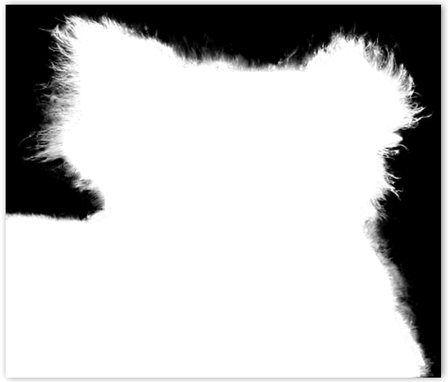
B **over** A



A **over** B

Notice: "over" is <u>not</u> commutative

A **over** $B \neq B$ **over** A



Koala over NYC

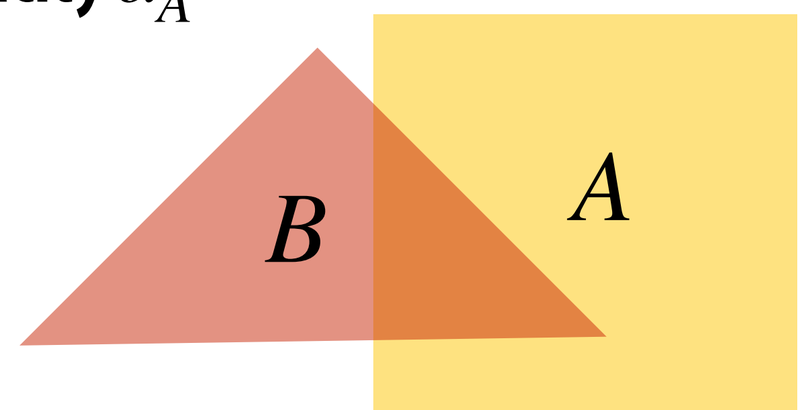
Porter & Duff "Compositing Digital Images" (1984)

Over operator: non-premultiplied alpha

Composite image B with opacity α_B over image A with opacity α_A

A first attempt:

$$A = (A_r, A_g, A_b)$$
$$B = (B_r, B_g, B_b)$$



B **over** A

Composite color:

what B lets through

$$C = \alpha_B B + (1 - \alpha_B)\alpha_A A$$

appearance of semi-transparent B

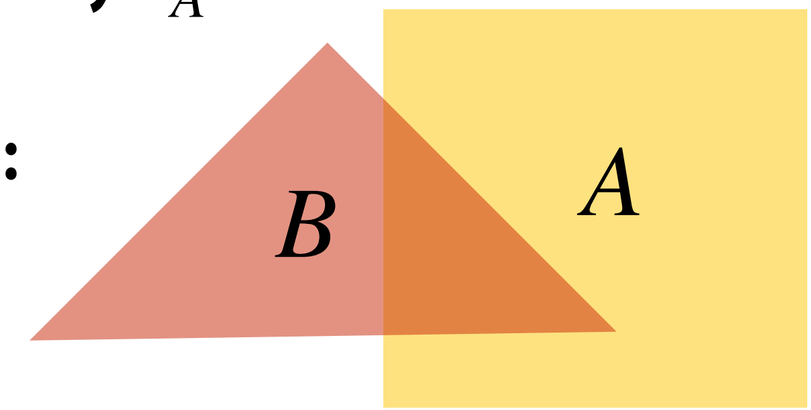
appearance of semi-transparent A

Composite alpha:

$$\alpha_C = \alpha_B + (1 - \alpha_B)\alpha_A$$

Over operator: premultiplied alpha

Composite image B with opacity α_B over image A with opacity α_A



B **over** A

Premultiplied alpha—multiply color by α, <u>then</u> composite:

$$A' = (\alpha_A A_r,\ \alpha_A A_g,\ \alpha_A A_b, \alpha_A)$$

$$B' = (\alpha_B B_r,\ \alpha_B B_g,\ \alpha_B B_b, \alpha_B)$$

$$C' = B' + (1 - \alpha_B)A'$$

Notice premultiplied alpha composites alpha just like how it composites rgb.
(Non-premultiplied alpha composites alpha differently than rgb.)

"Un-premultiply" to get final color:

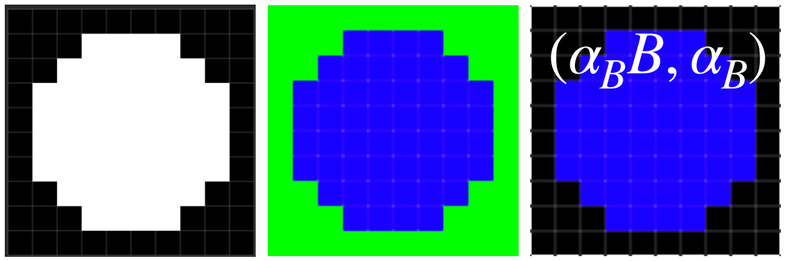
$$(C_r,\ C_g,\ C_b,\ \alpha_C) \Longrightarrow (C_r/\alpha_C, C_g/\alpha_C, C_b/\alpha_C)$$

Q: Does this division remind you of anything?

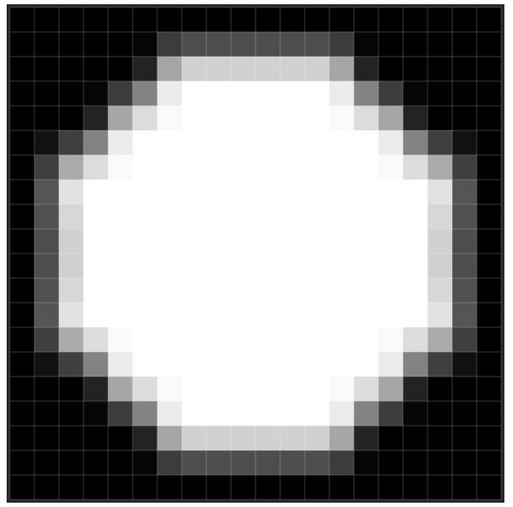
Compositing with & without premultiplied α

Suppose we upsample an image w/ an α channel, then composite it onto a background:

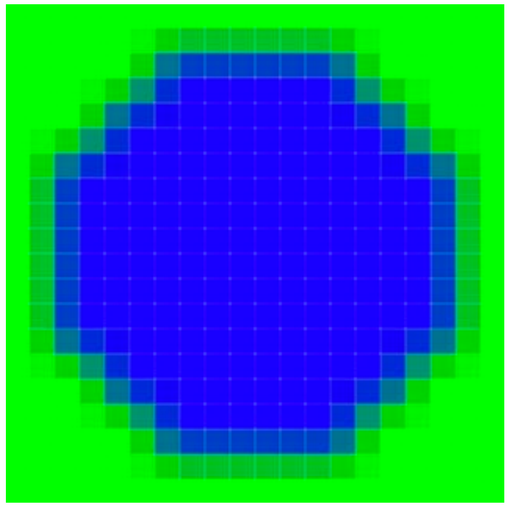
image B



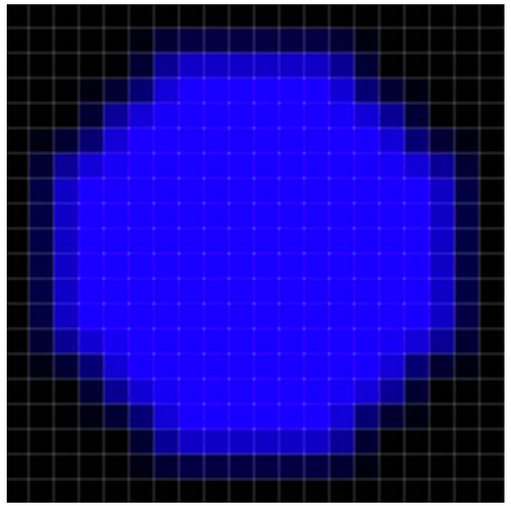
original alpha | original color | premultiplied color



upsampled alpha

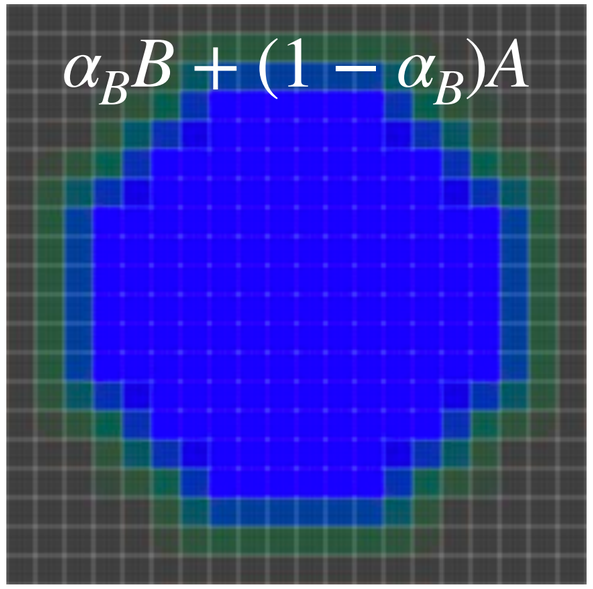


upsampled color

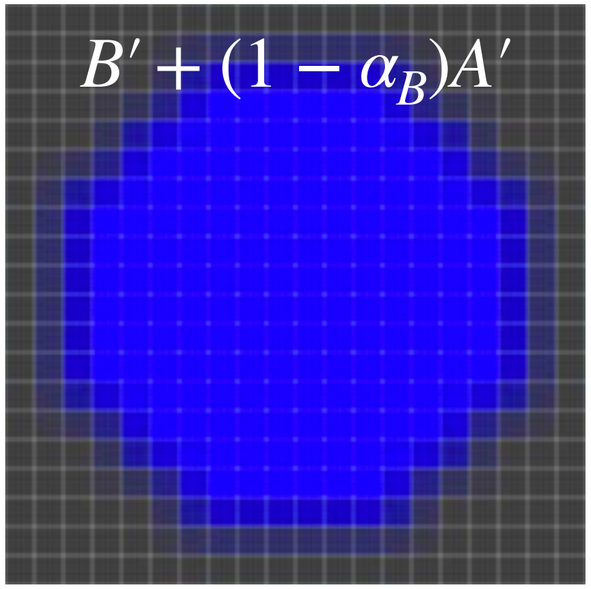


upsampled premultiplied color

new background A
$(\alpha_A = 1)$



B over A
non-premultiplied



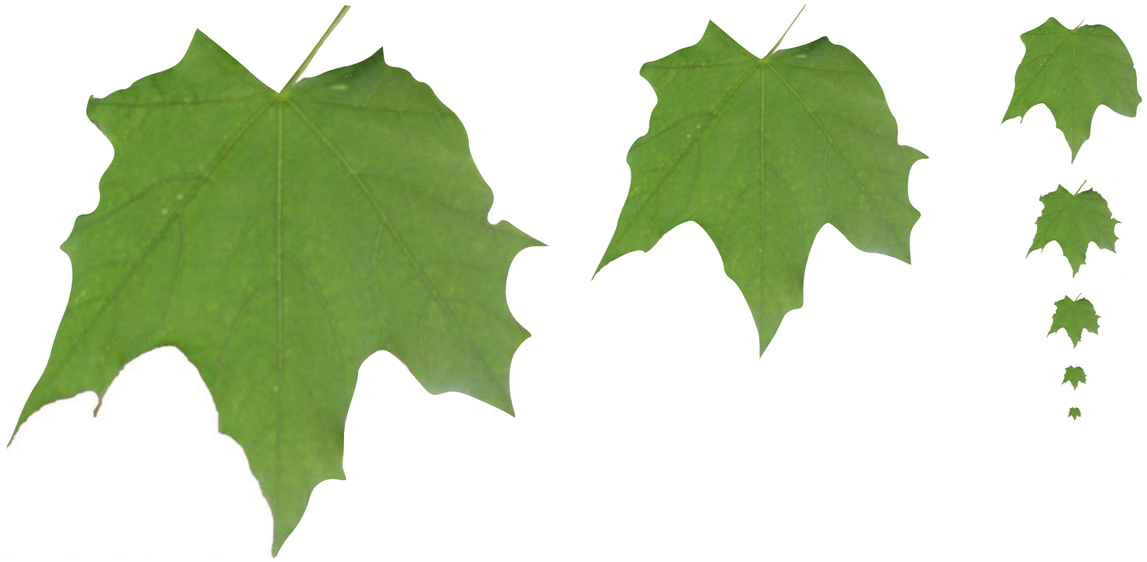
B over A
premultiplied

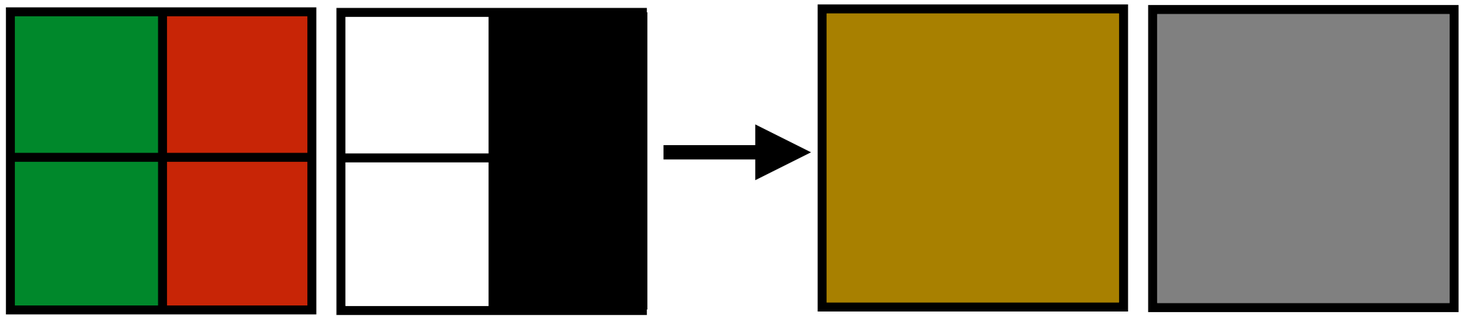
Q: Why do we get the "green fringe" when we don't premultiply?

Similar problem with non-premultiplied α

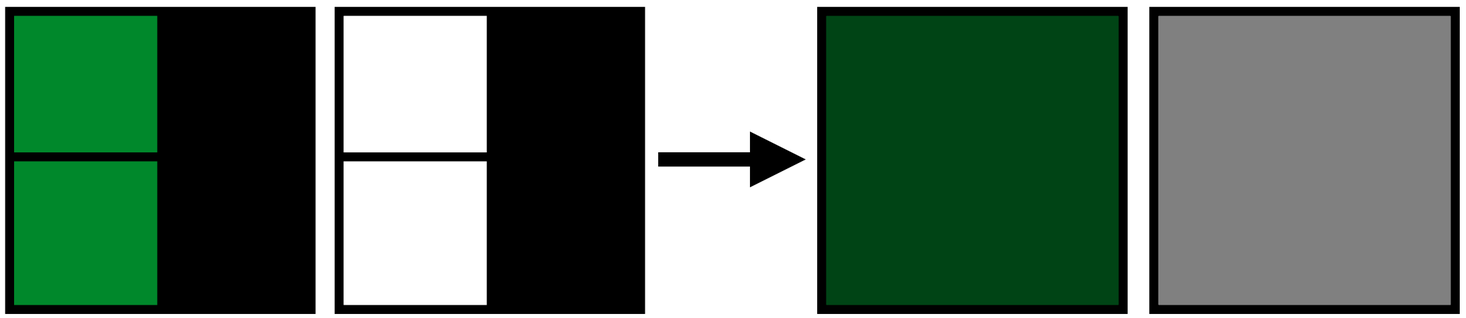
Consider pre-filtering (downsampling) a texture with an alpha matte

desired downsampled result





input color | input α | filtered color | filtered α | composited over white

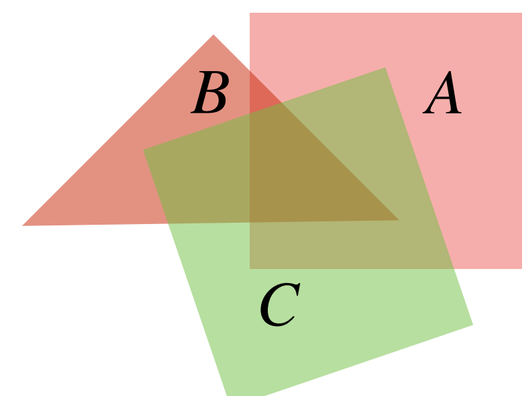


premultiplied color | premultiplied α | filtered color | filtered α | composited over white

More problems: applying "over" repeatedly

Composite image C with opacity α_C over B with opacity α_B over image A with opacity α_A

Premultiplied alpha is closed under composition; non-premultiplied alpha is not!



C **over** B **over** A

Example: composite 50% bright red over 50% bright red (where "bright red" = $(1,0,0)$, and $\alpha = 0.5$)

non-premultiplied

color

$$.5(1,0,0) + (1-.5).5(1,0,0)$$

$$\Downarrow$$

$$(0.75,0,0)$$ too dark!

alpha

$$.5 + (1-.5).5 = .75$$

premultiplied

color

$$(.5,0,0,.5)+(1-.5)(.5,0,0,.5)$$

$$\Downarrow$$

$$(.75,0,0.75)$$

$$\Downarrow$$ divide by α

bright red $(1,0,0)$

alpha

$$\alpha = 0.75$$

Summary: advantages of premultiplied alpha

- Compositing operation treats all channels the same (color and α)
- Fewer arithmetic operations for "over" operation than with non-premultiplied representation
- Closed under composition (repeated "over" operations)
- Better representation for filtering (upsampling/downsampling) images with alpha channel
- Fits naturally into rasterization pipeline (homogeneous coordinates)

Strategy for drawing semi-transparent primitives

Assuming all primitives are semi-transparent, and color values are encoded with premultiplied alpha, here's a strategy for rasterizing an image:

```
over(c1, c2)
{
   return c1.rgba + (1-c1.a) * c2.rgba;
}
```

```
update_color_buffer( x, y, sample_color, sample_depth )
{
   if (pass_depth_test(sample_depth, zbuffer[x][y])
   {
       // (how) should we update depth buffer here??
       color[x][y] = over(sample_color, color[x][y]);
  }
}
```

Q: What is the assumption made by this implementation?

Triangles must be rendered in back to front order!

Putting it all together

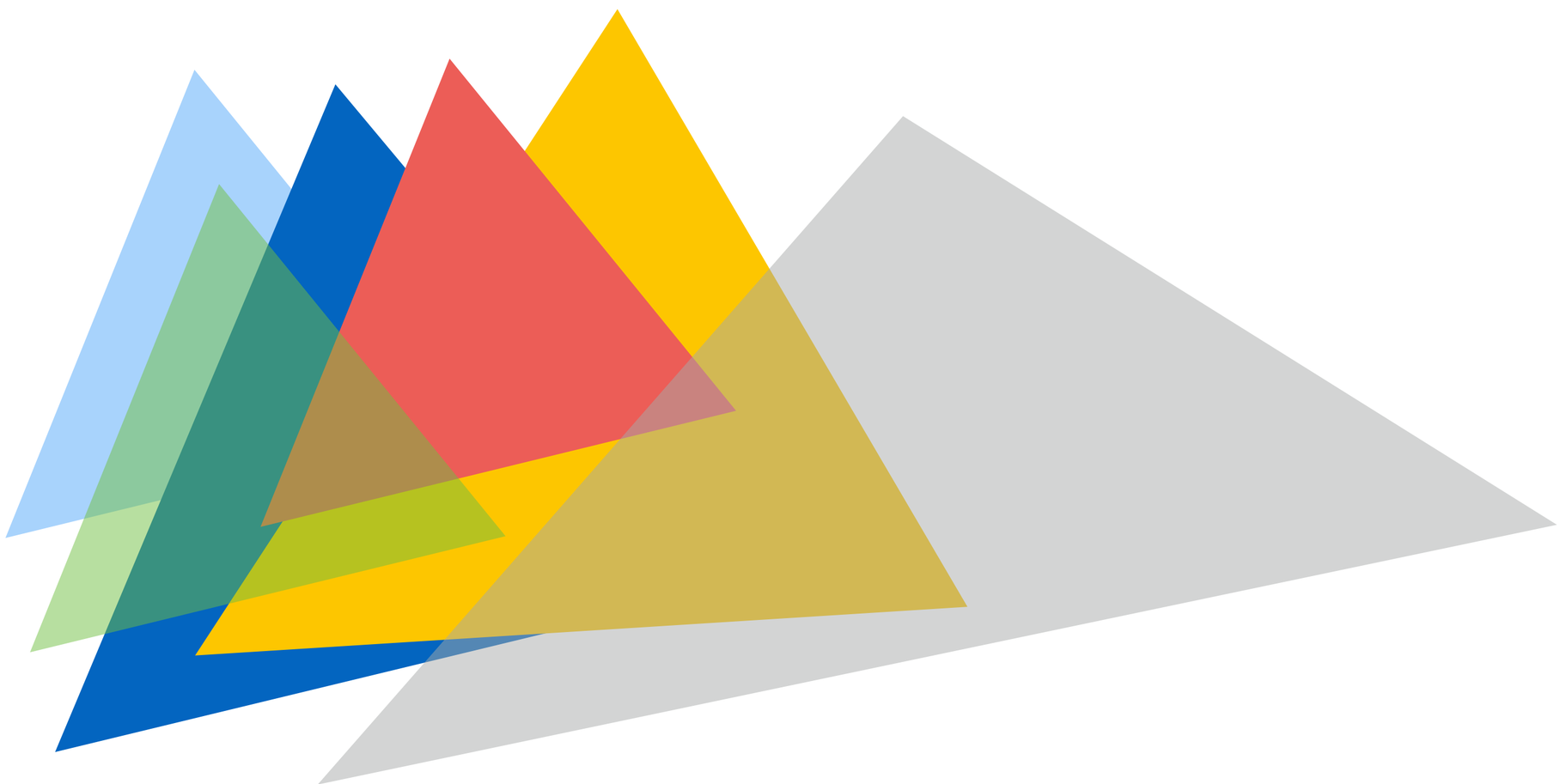
What if we have a mixture of opaque and transparent triangles?

<u>Step 1:</u> render opaque primitives (in any order) using depth-buffered occlusion

If pass depth test, triangle overwrites value in color buffer at sample

<u>Step 2:</u> disable depth buffer update, render semi-transparent surfaces in back-to-front order.

If pass depth test, triangle is composited OVER contents of color buffer at sample



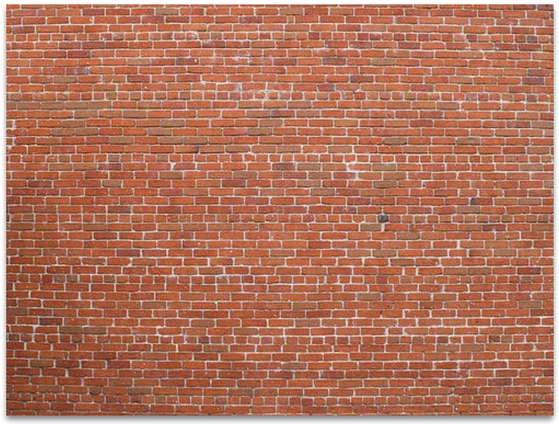
End-to-end rasterization pipeline

Goal: turn inputs into an image!

Inputs:

```
positions = {
    v0x, v0y, v0z,
    v1x, v1y, v1x,
    v2x, v2y, v2z,
    v3x, v3y, v3x,
    v4x, v4y, v4z,
    v5x, v5y, v5x
};
```

```
texcoords ={
    v0u, v0v,
    v1u, v1v,
    v2u, v2v,
    v3u, v3v,
    v4u, v4v,
    v5u, v5v
};
```



texture map

Object-to-camera-space transform $T \in \mathbb{R}^{4\times4}$

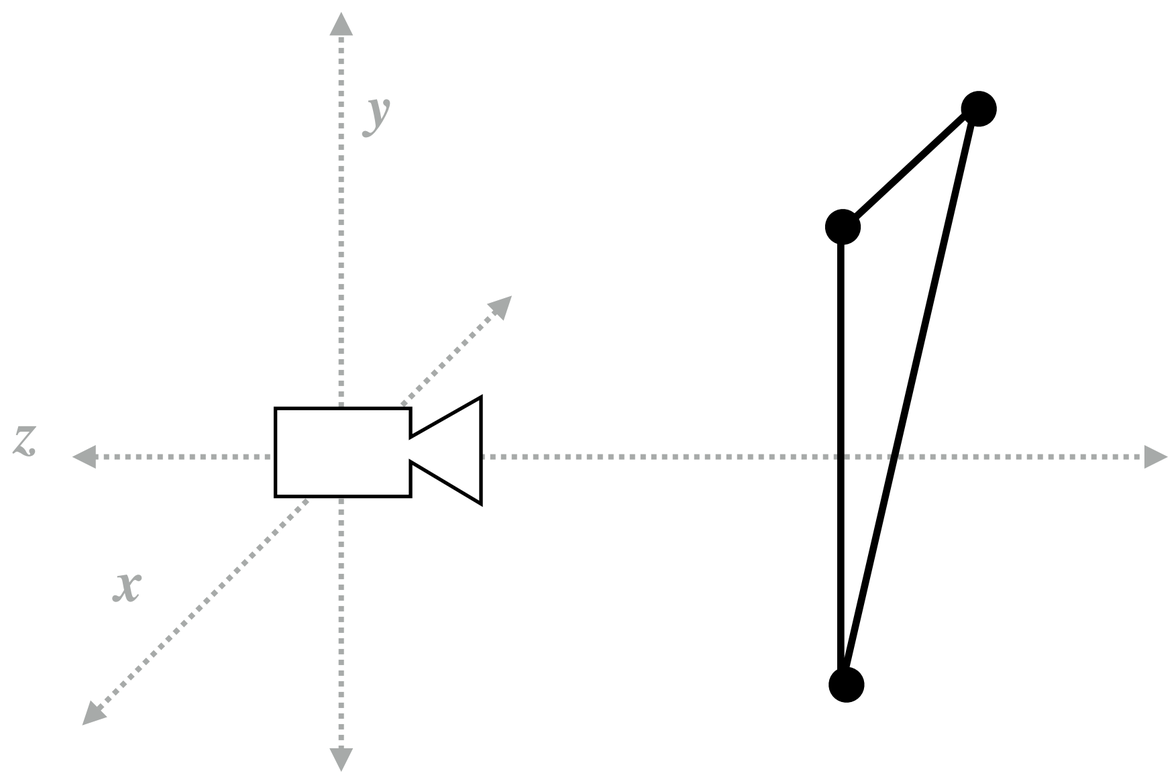
Perspective projection transform $P \in \mathbb{R}^{4\times4}$

Size of output image (W, H)

At this point we have all the tools we need to make an image…
Let's review!

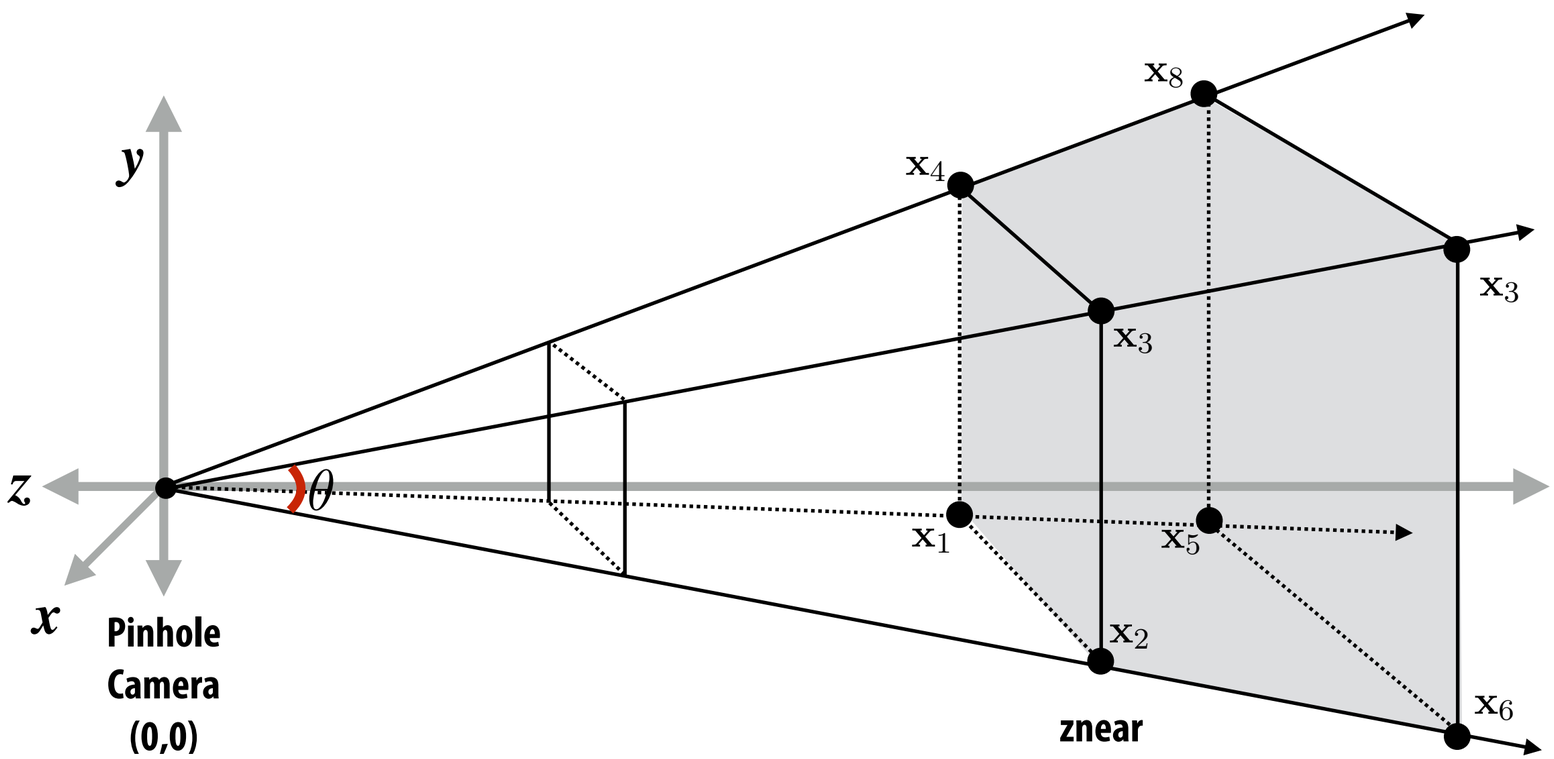
Step 1:

Transform triangle vertices into camera space

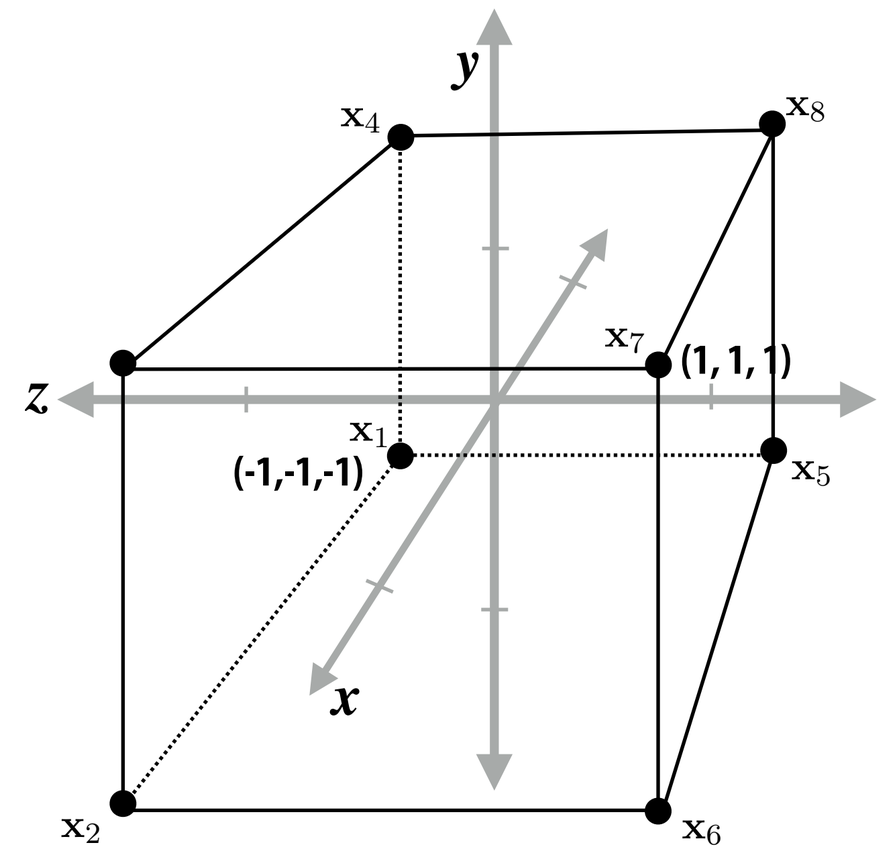


Step 2:

Apply perspective projection transform to transform triangle vertices into normalized coordinate space



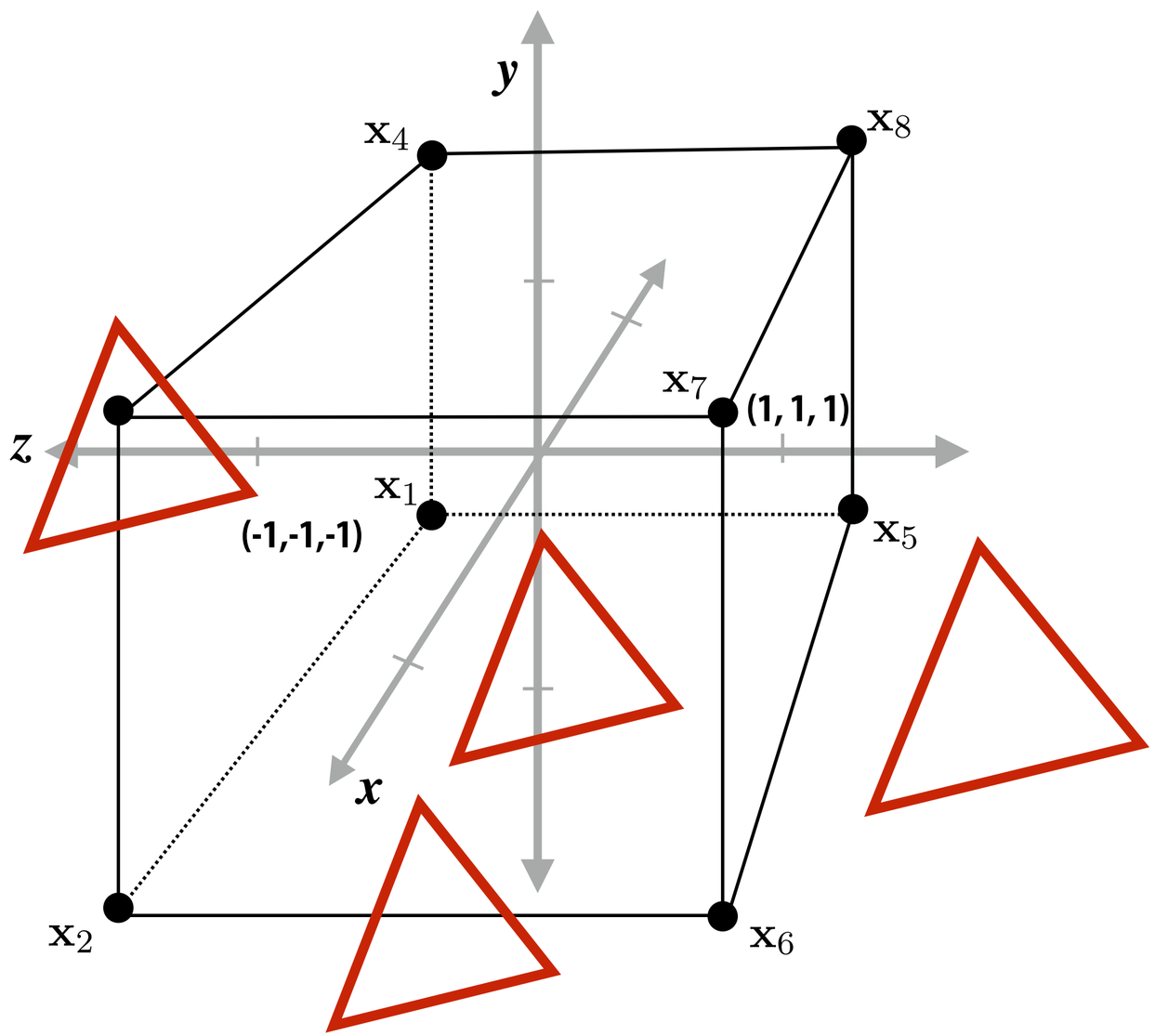
Camera-space positions: 3D



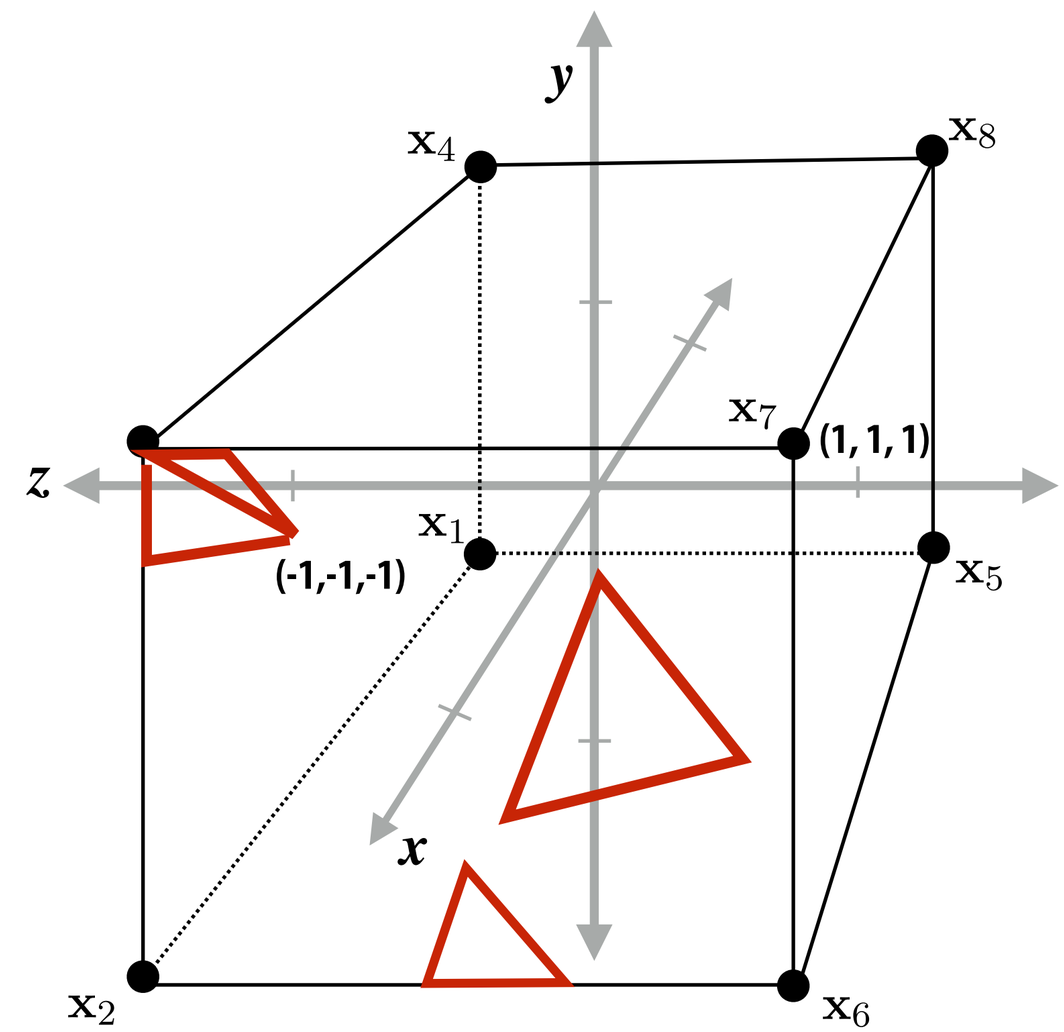
Normalized space positions

Step 3: clipping

- Discard triangles that lie complete outside the unit cube (culling)
  - They are off screen, don't bother processing them further
- Clip triangles that extend beyond the unit cube to the cube
  - (possibly generating new triangles)



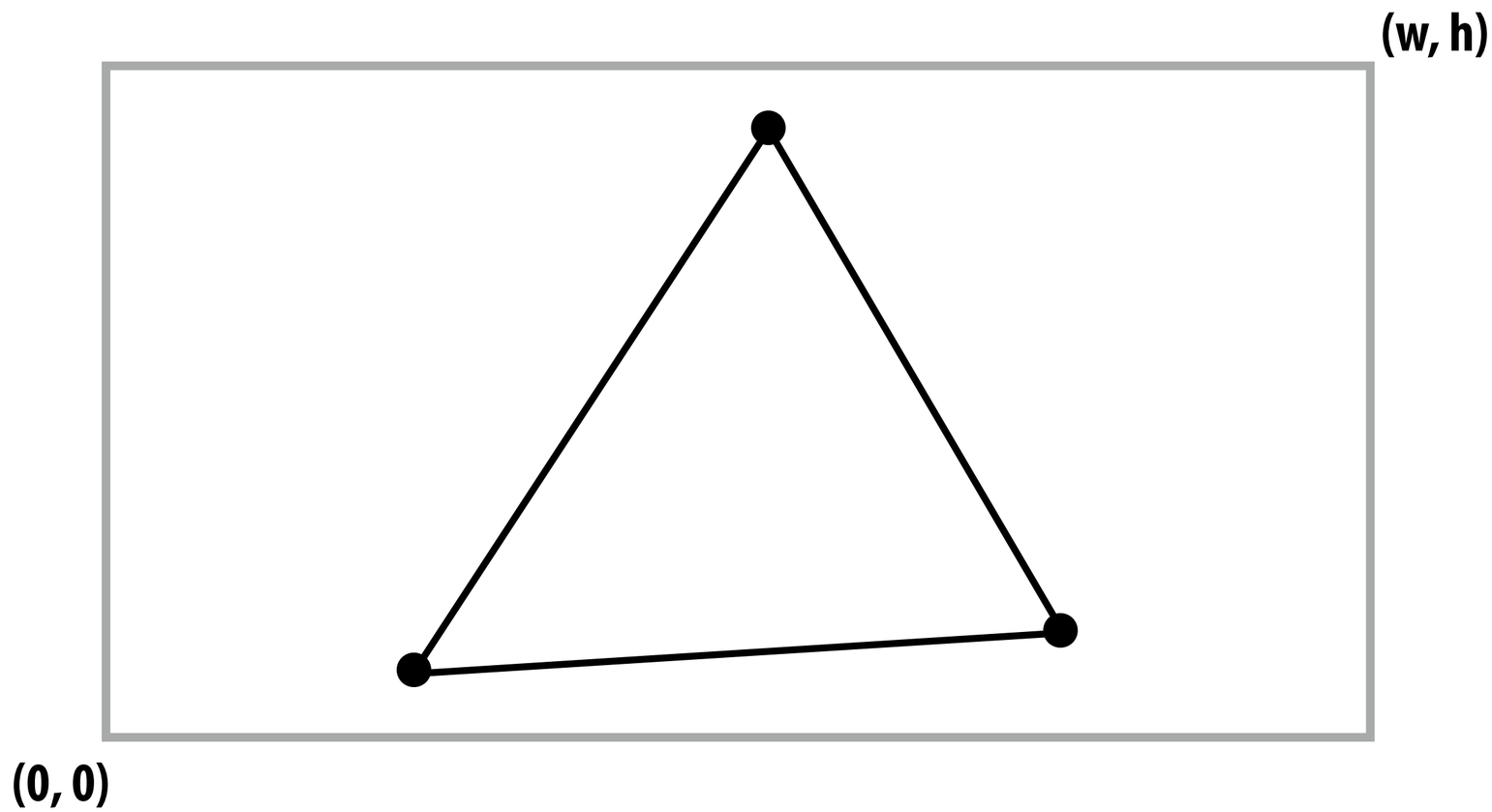
Triangles before clipping



Triangles after clipping

Step 4: transform to screen coordinates

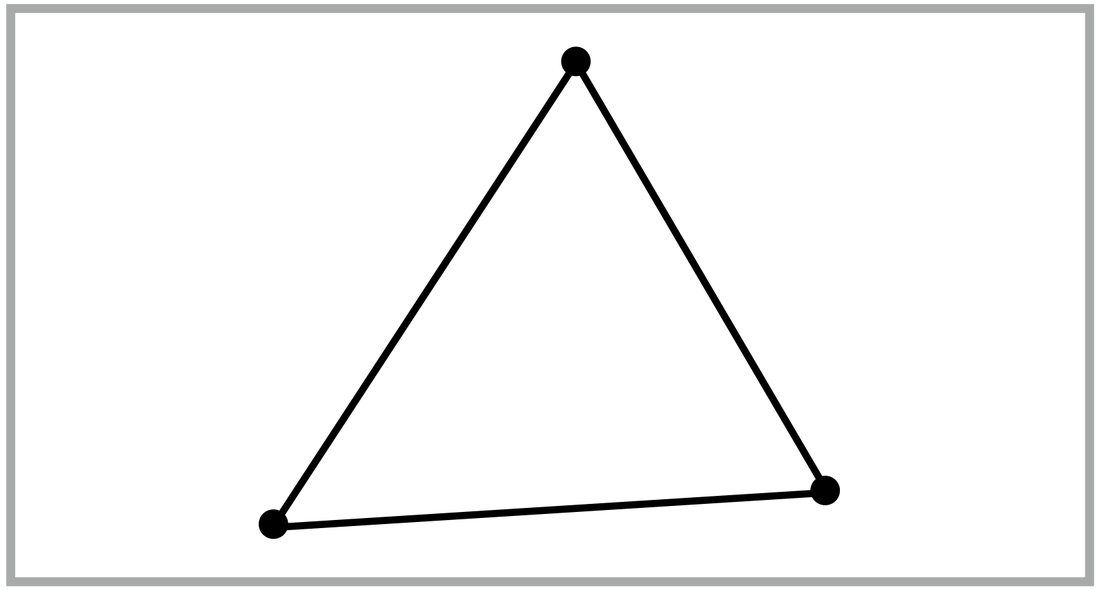
Perform homogeneous divide, transform vertex xy positions from normalized coordinates into screen coordinates (based on screen w,h)



Step 5: setup triangle (triangle preprocessing)

Before rasterizing triangle, can compute a bunch of data that will be used by all fragments, e.g.,

- **triangle edge equations**
- **triangle attribute equations**
- **etc.**



$$\mathbf{E}_{01}(x,y) \qquad \mathbf{U}(x,y)$$

$$\mathbf{E}_{12}(x,y) \qquad \mathbf{V}(x,y)$$

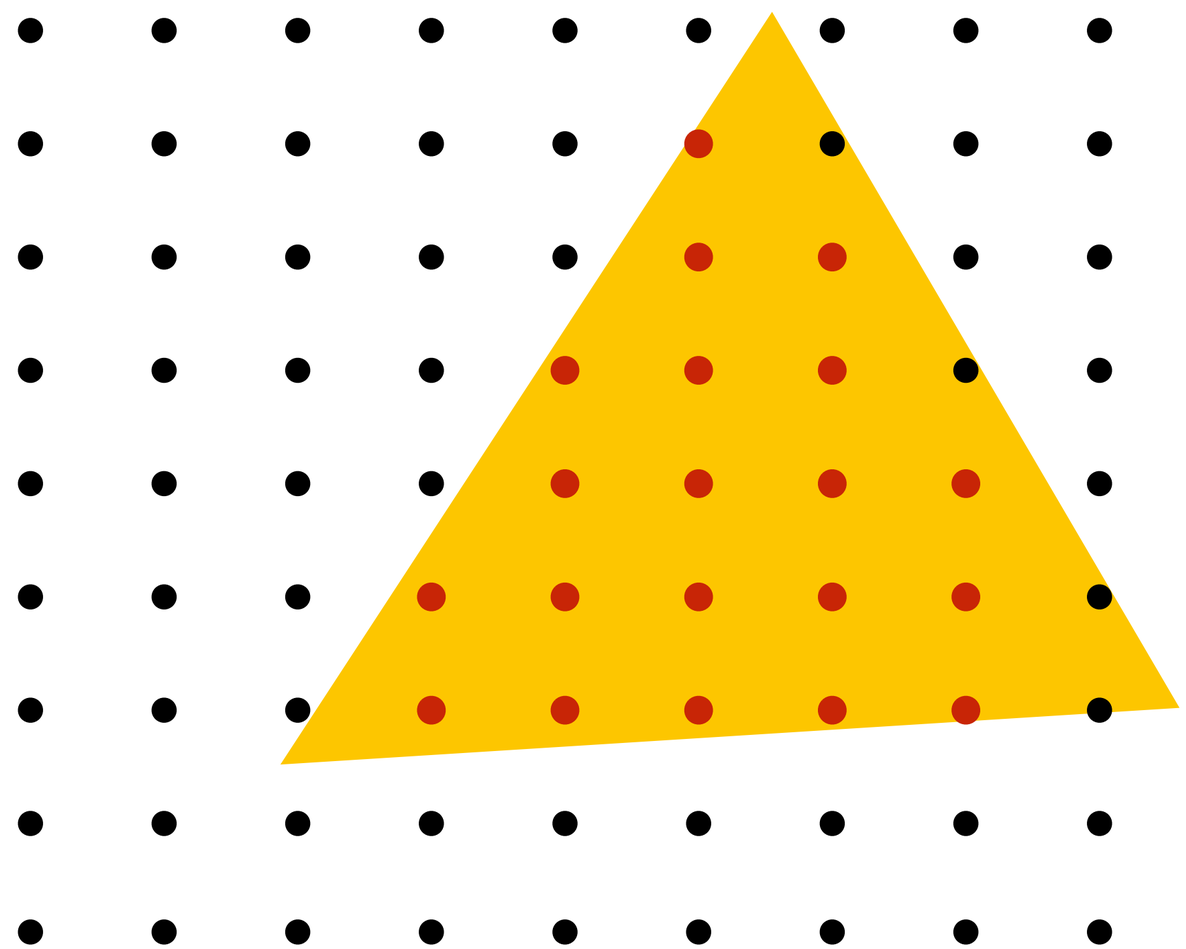
$$\mathbf{E}_{20}(x,y)$$

$$\frac{1}{\mathbf{w}}(x,y)$$

$$\mathbf{Z}(x,y)$$

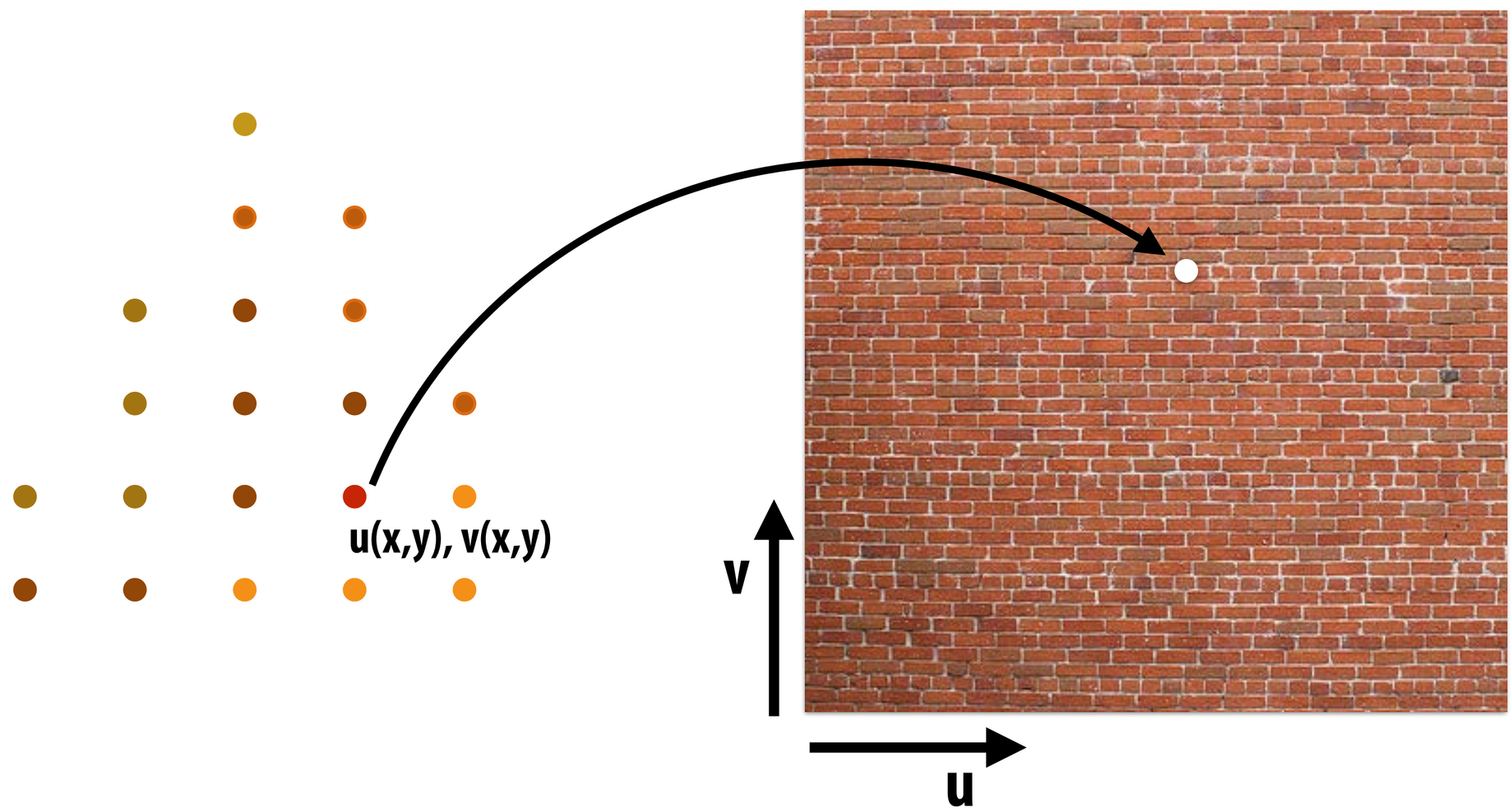
Step 6: sample coverage

Evaluate attributes z, u, v at all covered samples



Step 6: compute triangle color at sample point

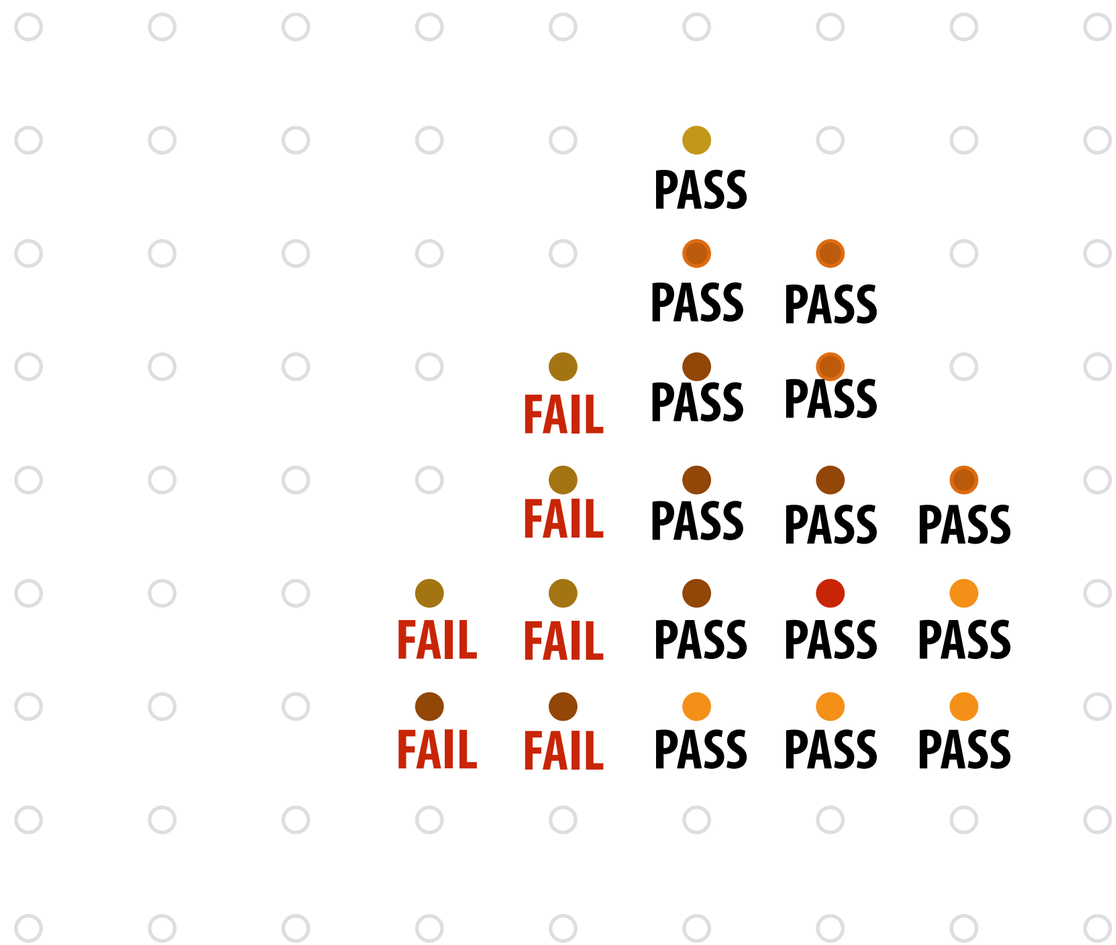
e.g., sample texture map *



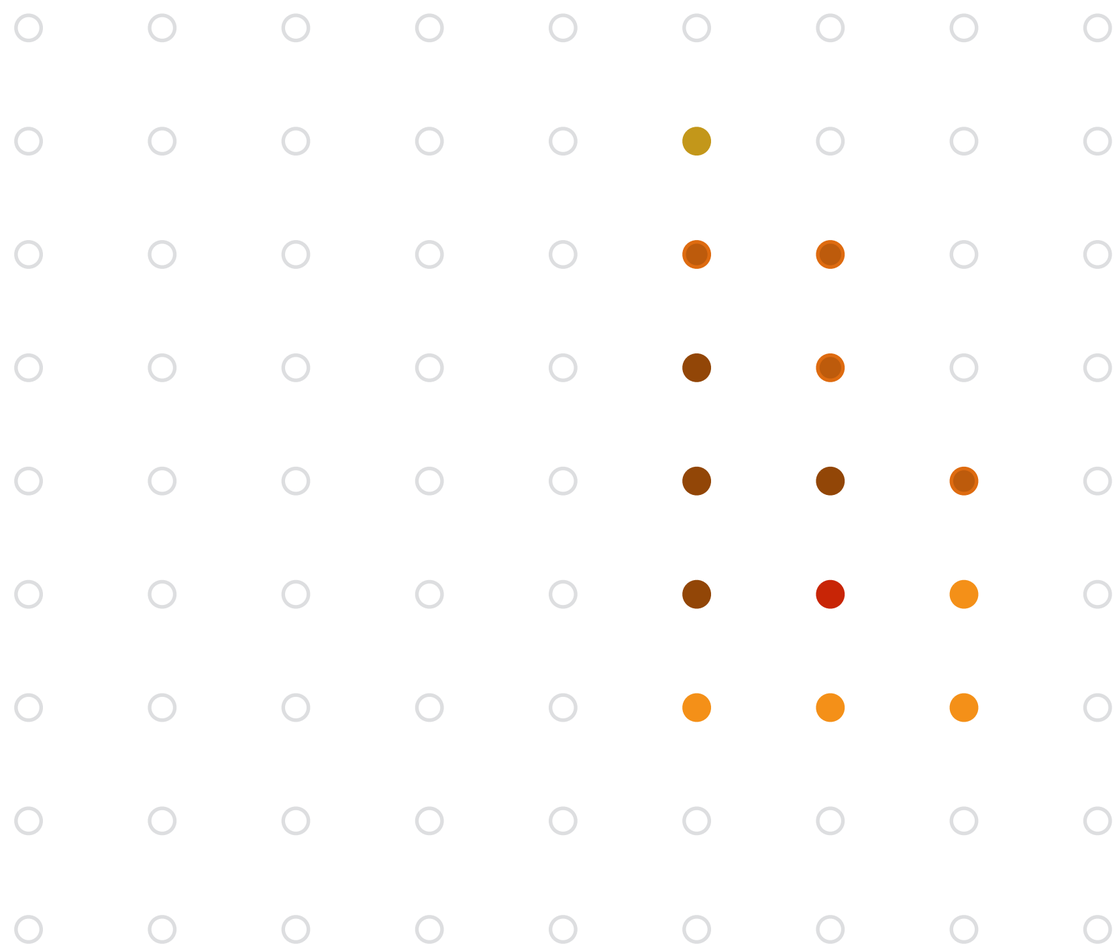
*Not the only way to get a color! Later we'll talk about more general models of materials…

Step 7: perform depth test (if enabled)

Also update depth value at covered samples (if necessary)



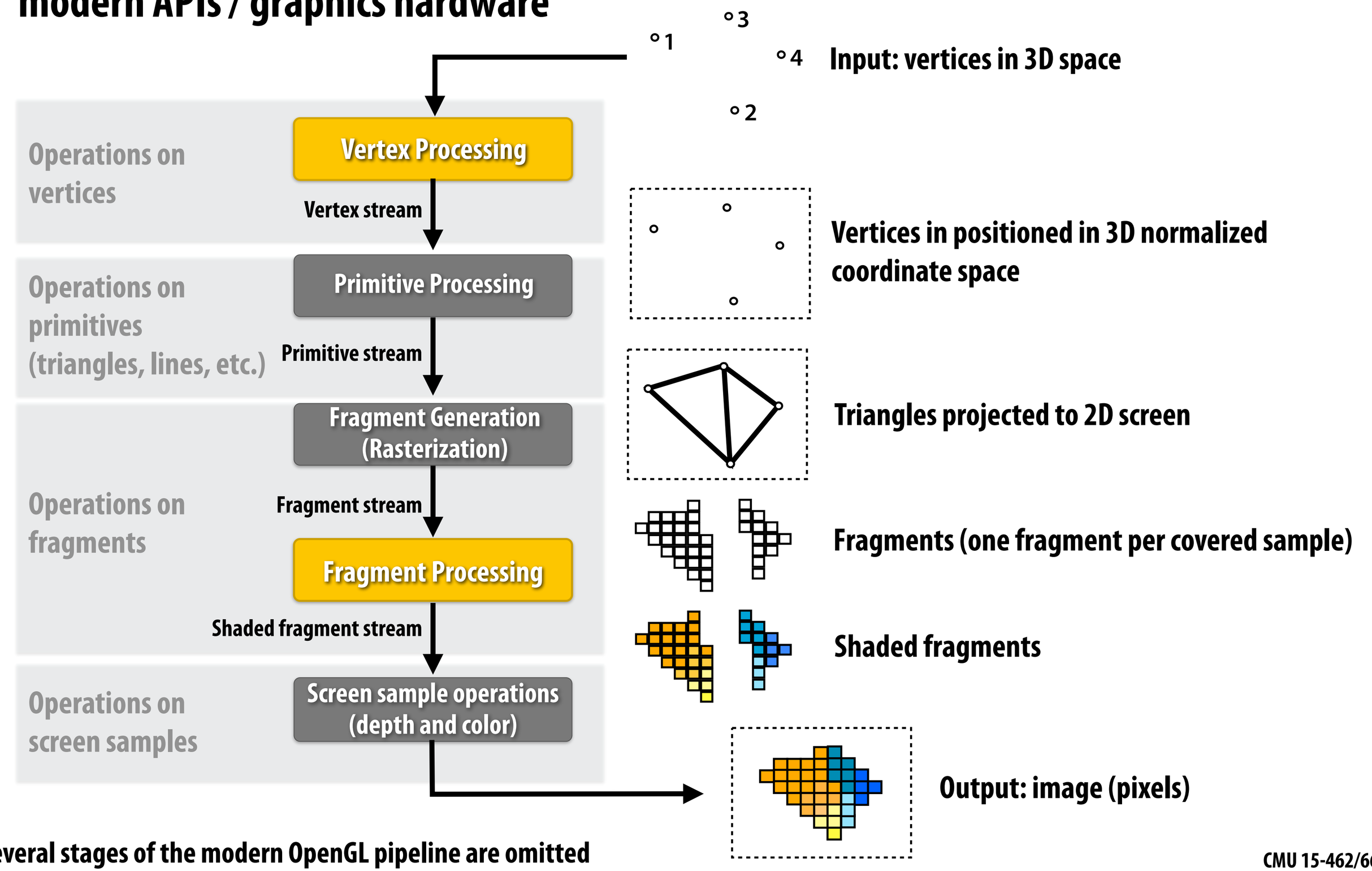
Step 8: update color buffer* (if depth test passed)



* Possibly using OVER operation for transparency

OpenGL/Direct3D graphics pipeline

Our rasterization pipeline doesn't look much different from "real" pipelines used in modern APIs / graphics hardware



* Several stages of the modern OpenGL pipeline are omitted

Goal: render very high complexity 3D scenes

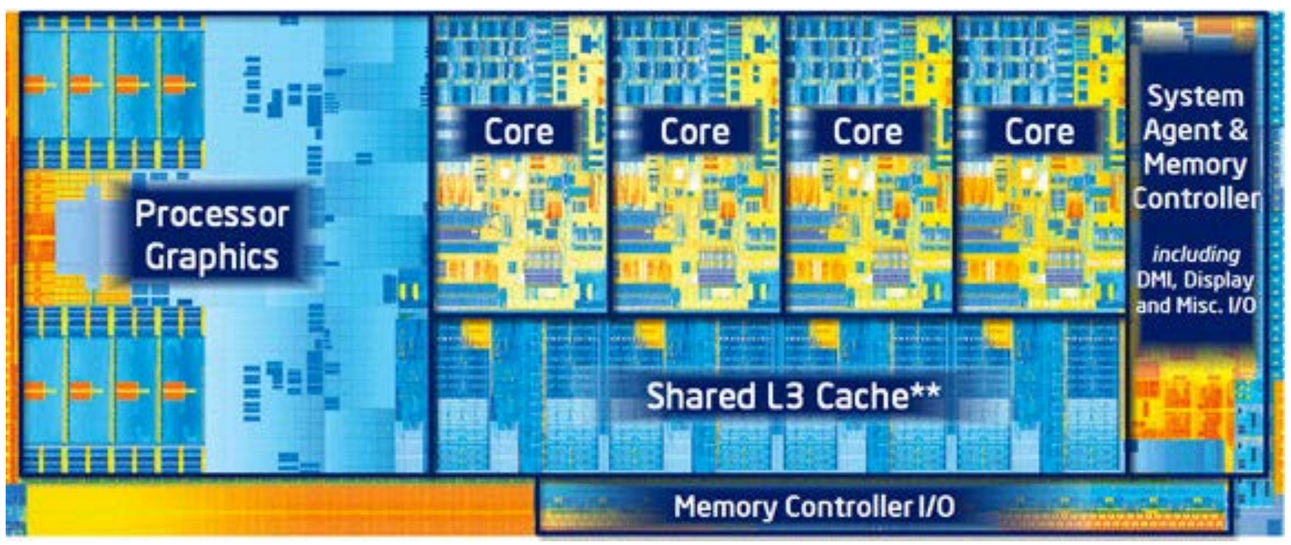
- 100's of thousands to millions of triangles in a scene
- Complex vertex and fragment shader computations
- High resolution screen outputs (~10Mpixel + *supersampling*)
- 30-120 fps

Unreal Engine Kite Demo (Epic Games 2015)

Graphics pipeline implementation: GPUs

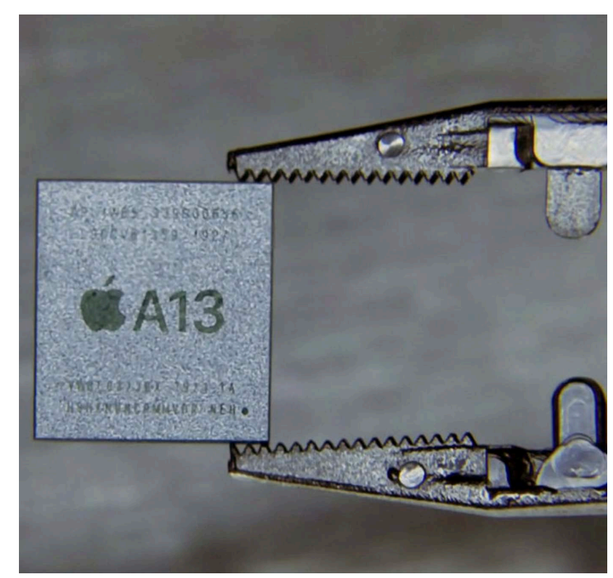
Specialized processors for executing graphics pipeline computations

discrete GPU card



integrated GPU: part of modern CPU die

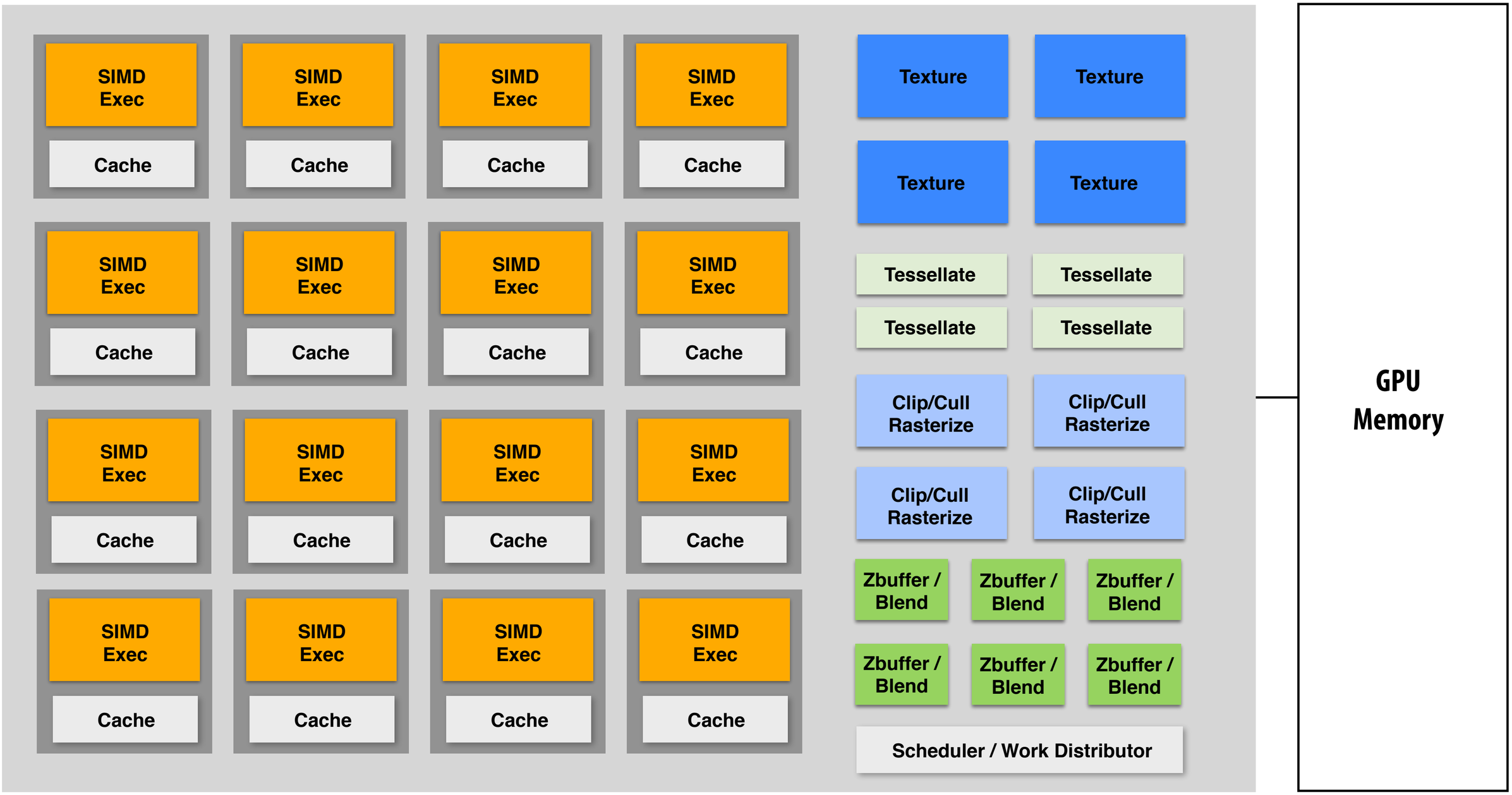
smartphone GPU (integrated)



GPU: heterogeneous, multi-core processor

Modern GPUs offer ~35 TFLOPs of performance for generic vertex/fragment programs ("compute")

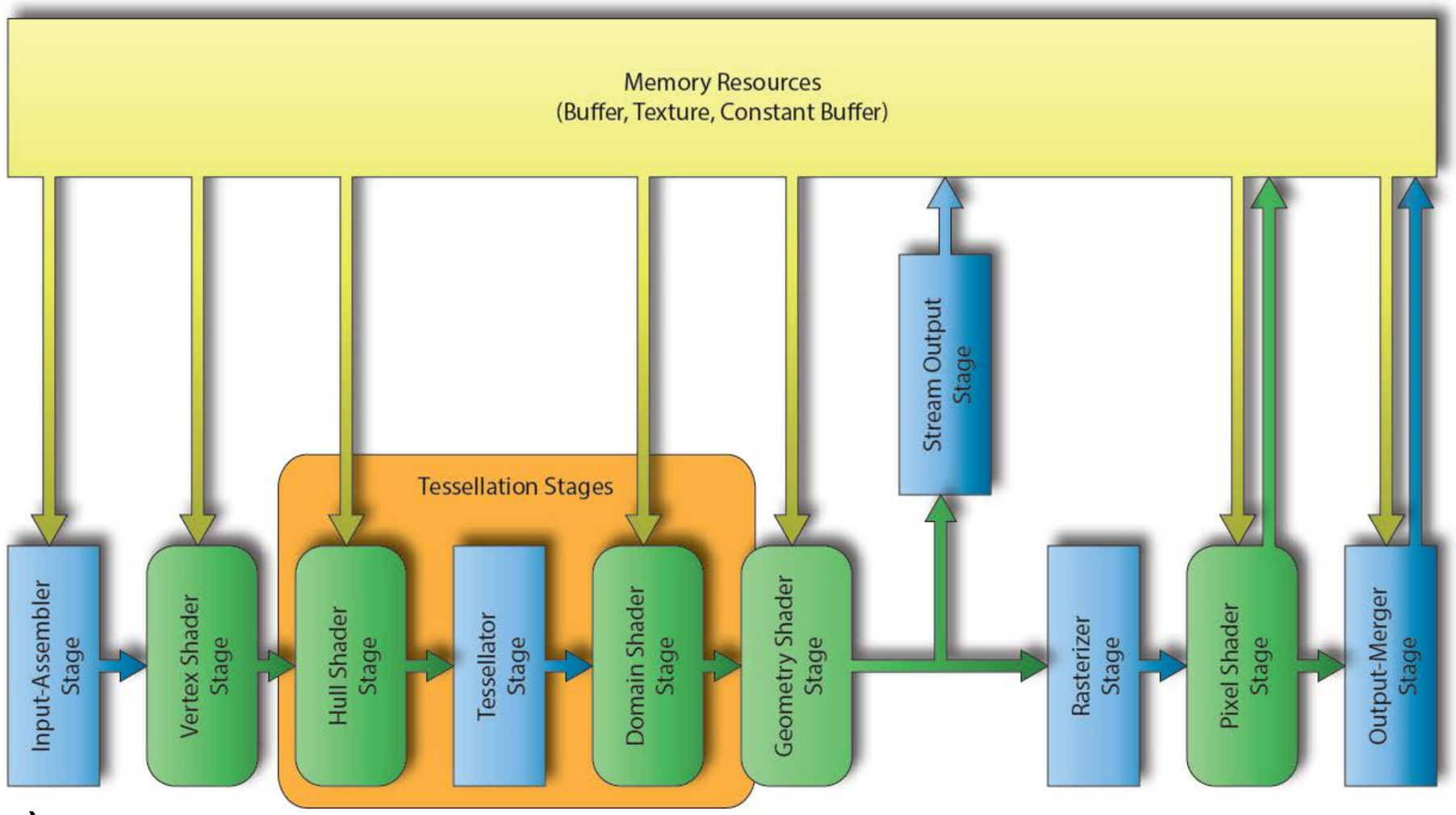
still enormous amount of *fixed-function* compute over here



This part (mostly) not used by CUDA/OpenCL; raw graphics horsepower still greater than compute!

Modern Rasterization Pipeline

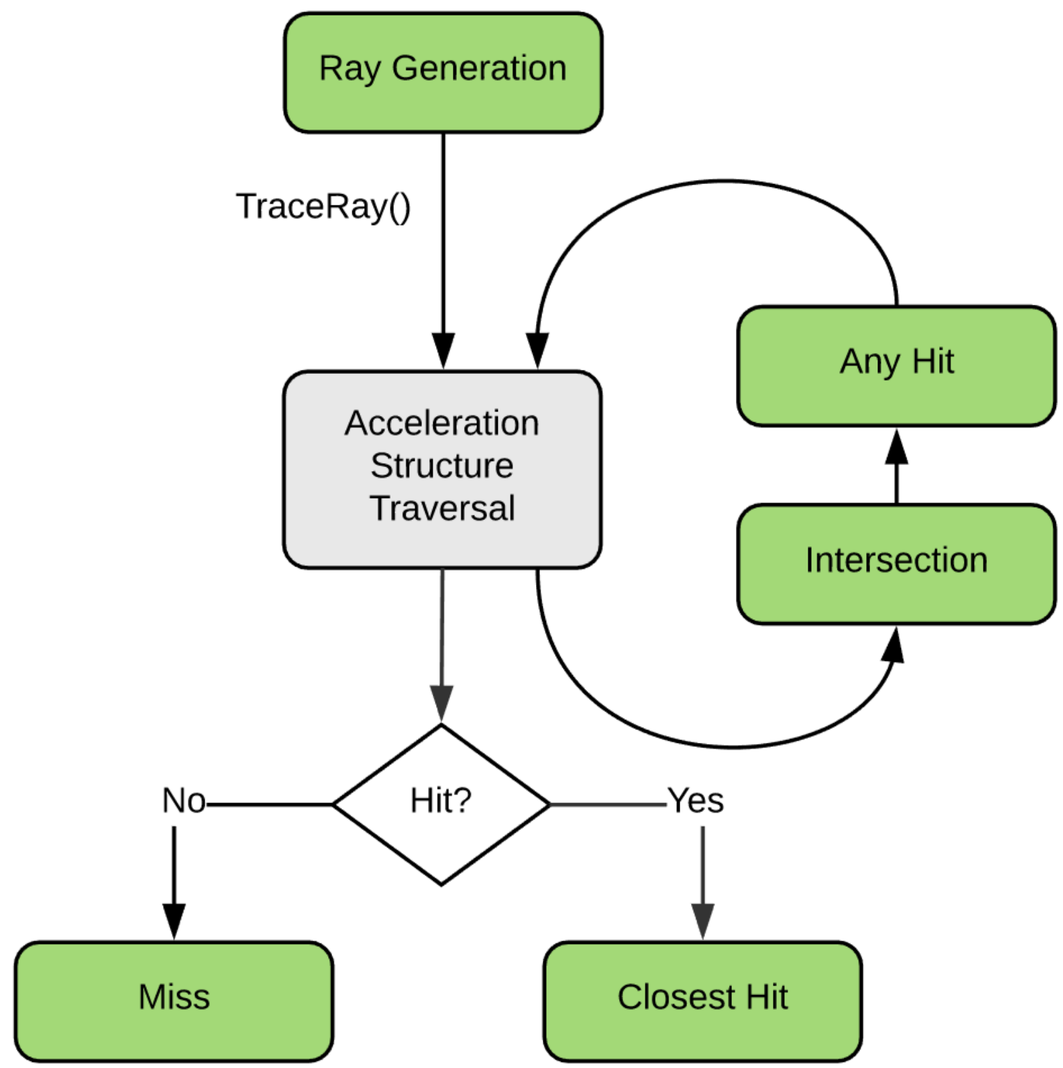
- Trend toward more generic (but still highly parallel!) computation:
  - make stages programmable
    - replace fixed function vertex, fragment processing
    - add geometry, tessellation shaders
    - generic "compute" shaders (whole other story…)
  - more flexible scheduling of stages



(DirectX 12 Pipeline)

Ray Tracing in Graphics Pipeline

- **More recently: specialized pipeline for ray tracing (NVIDIA RTX)**



https://devblogs.nvidia.com/introduction-nvidia-rtx-directx-ray-tracing/

GPU Ray Tracing Demo ("Marbles at Night")

What else do we need to know to generate images like these?

GEOMETRY

How do we describe complex shapes (so far just triangles…)

RENDERING

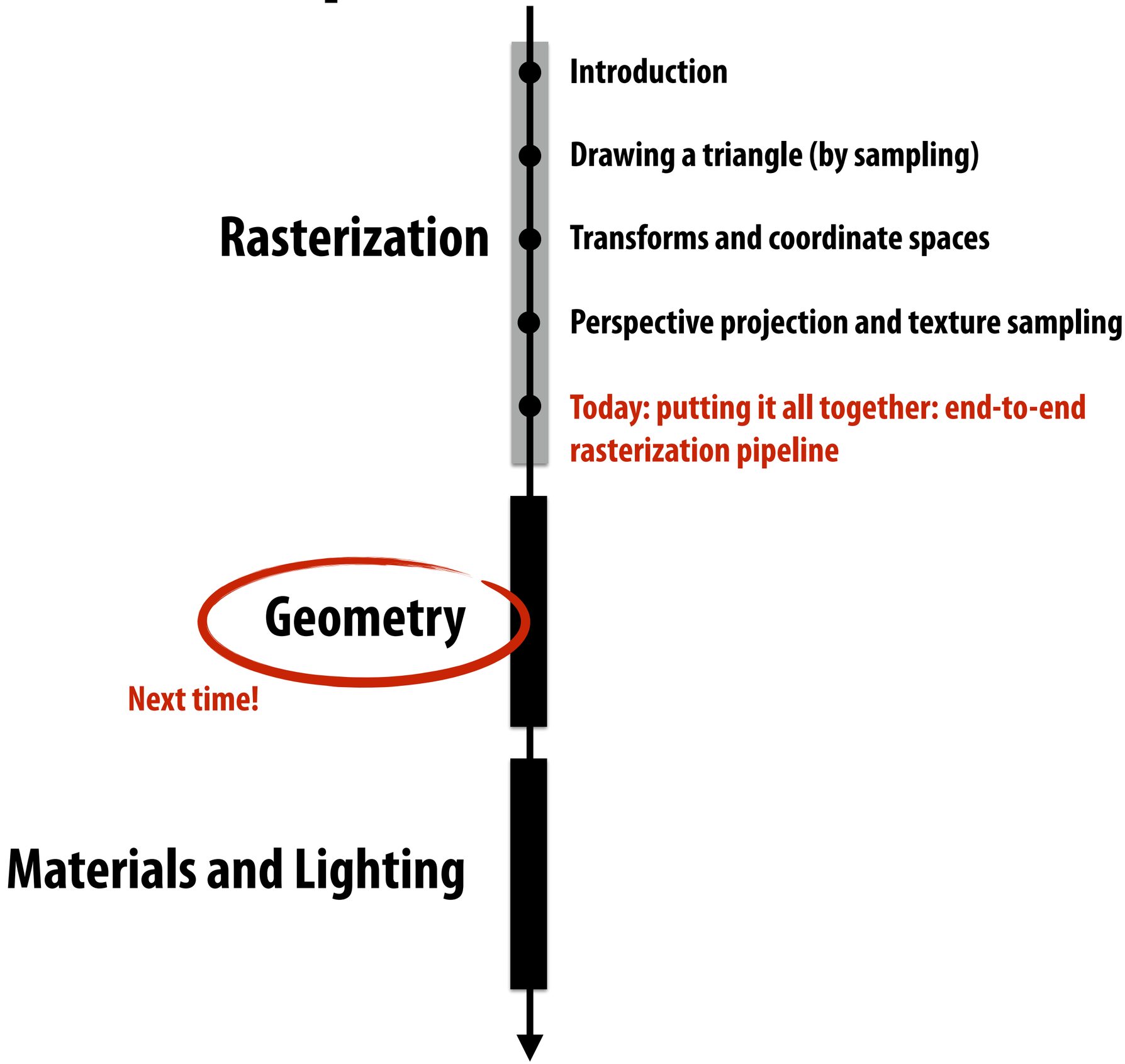
How does light interact w/ materials to produce color?

ANIMATION

How do we describe the way things move?

("Moana", Disney 2016)

Course roadmap

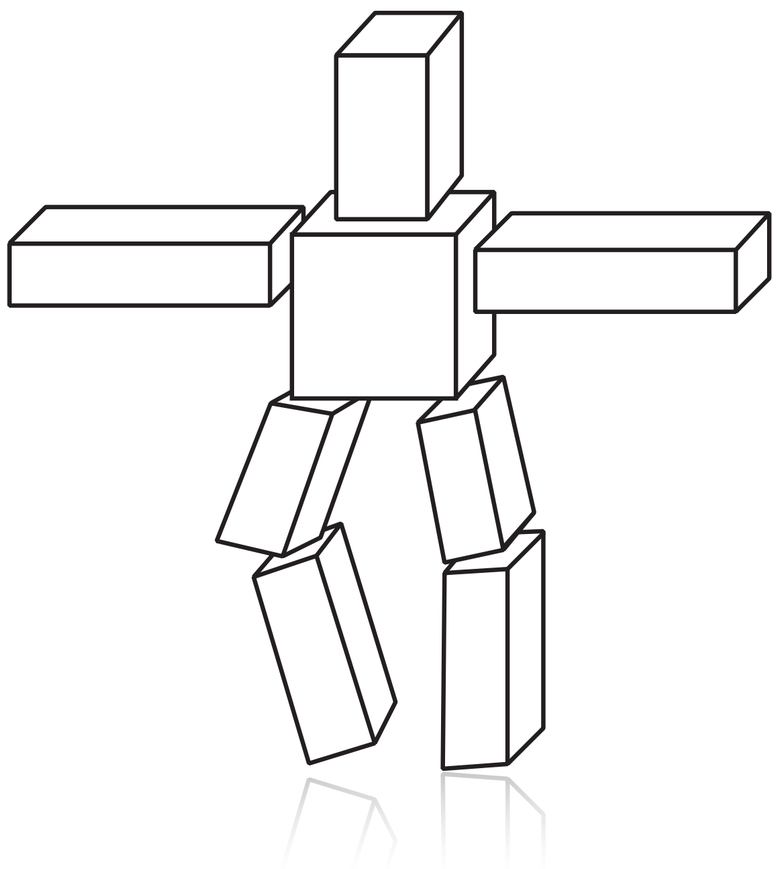


Introduction to Geometry

Computer Graphics
CMU 15-462/15-662

Increasing the complexity of our models

Transformations

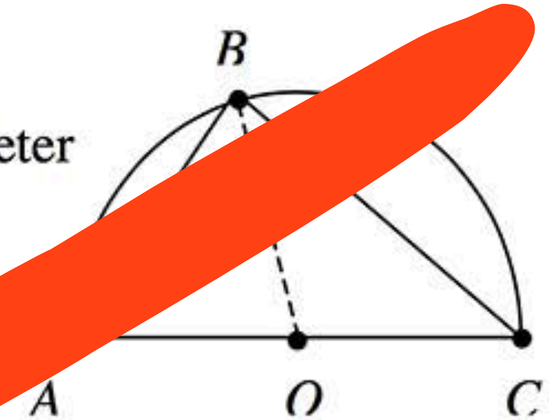


Geometry

Materials, lighting, ...

Q: What is geometry?

A: Geometry is the study of two-column proofs.



TH[illegible] 9.5. Let ΔABC be inscribed in a semicircle with diameter $\overline{A}\ \overline{C}$.

Then $\angle ABC$ [illegible] angle.

Proof:

Statement	[illegible]
1. Draw radius OB. Then $OB = OC = O$[illegible]	Given
2. $m\angle OBC = m\angle BCA$	[illegible]sceles Triangle Theorem
$m\angle OBA = m\angle BAC$	
3. $m\angle ABC = m\angle OBA +$ [illegible]	3. Angl[illegible]ostulate
4. $m\angle ABC + m\angle BC$[illegible] $\angle BAC = 180$	4. The sum o[illegible]gles of a triangle is 180
5. $m\angle ABC + m$[illegible] $+ m\angle OBA = 180$	5. Substitution (lin[illegible]
6. $2\, m\angle$[illegible] 180	6. Substitution (line 3)
7. m[illegible] $= 90$	7. Division Property of Equality
8. $\angle ABC$ is a right angle	8. Definition of Right Angle

Ceci n'est pas géométrie.

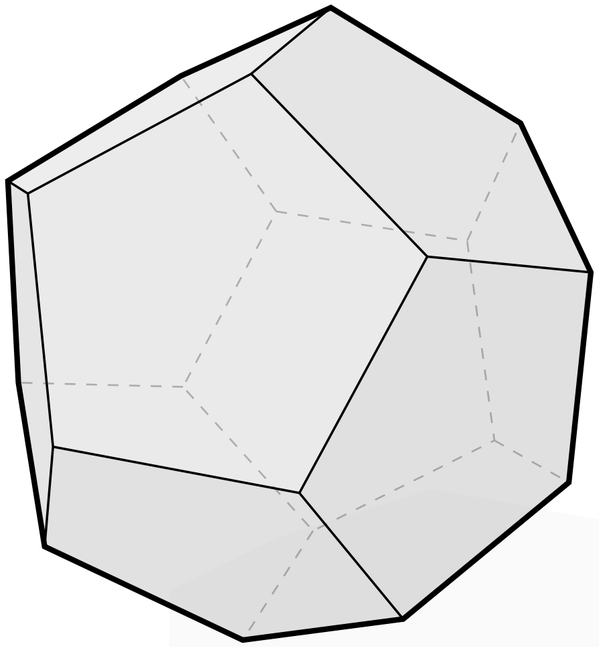
See: Paul Lockhart, "A Mathematician's Lament"

What is geometry?

"Earth" "measure"

ge•om•et•ry /jēˈämətrē/ *n.*

1. The study of shapes, sizes, patterns, and positions.
2. The study of spaces where some quantity (lengths, angles, etc.) can be *measured*.



Plato: "...the earth is in appearance like one of those balls which have leather coverings in twelve pieces..."

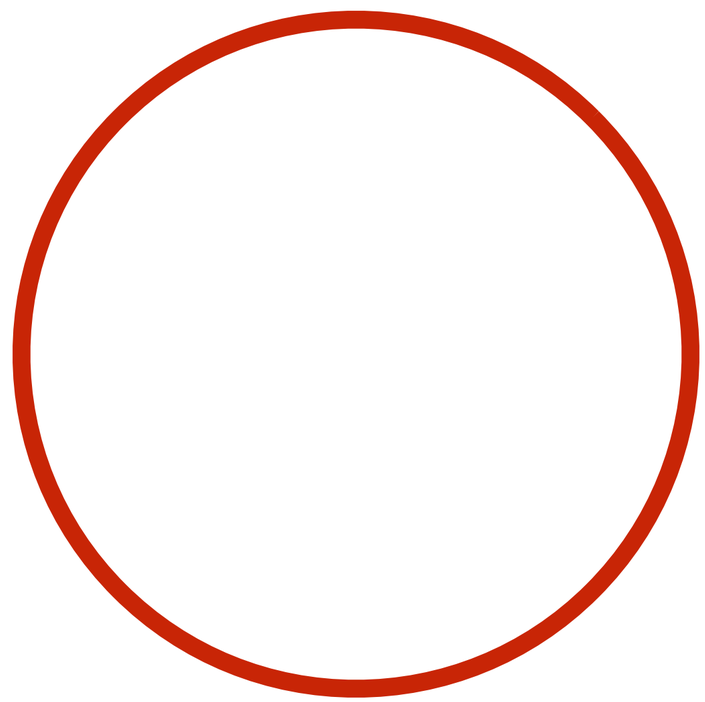
How can we describe geometry?

IMPLICIT

$$x^2+y^2=1$$

LINGUISTIC

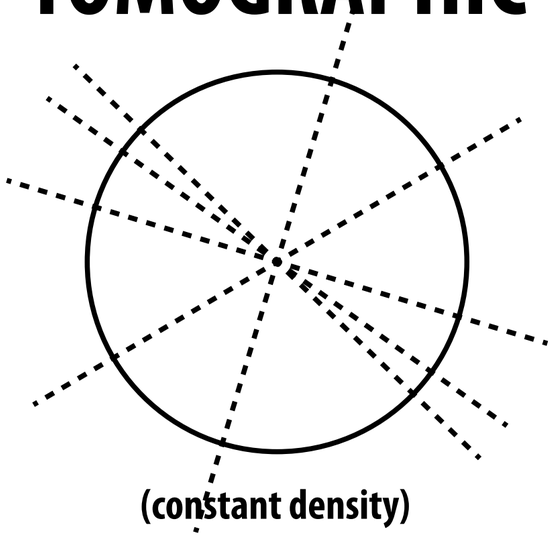
"unit circle"



EXPLICIT

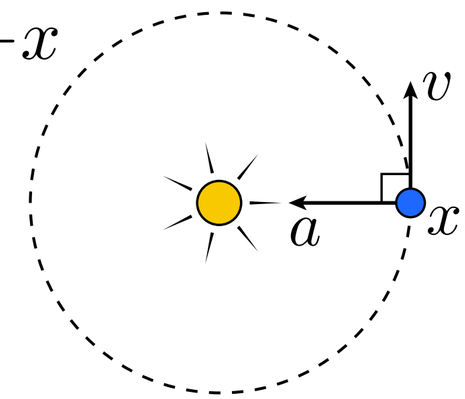
$$(\underbrace{\cos\theta}_{x}, \underbrace{\sin\theta}_{y})$$

TOMOGRAPHIC



DYNAMIC

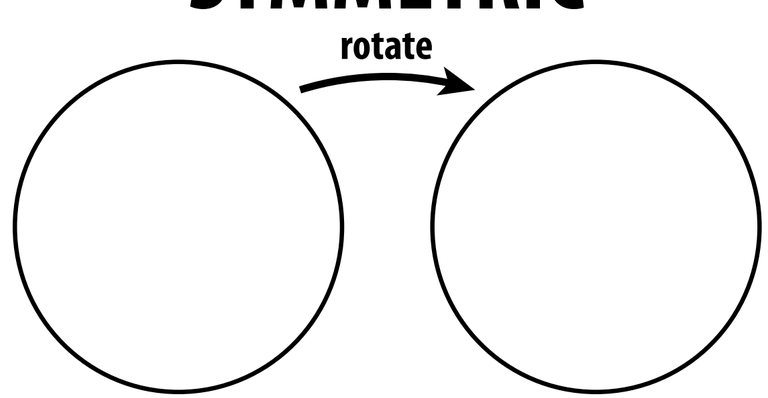
$$\frac{d^2}{dt^2}x = -x$$



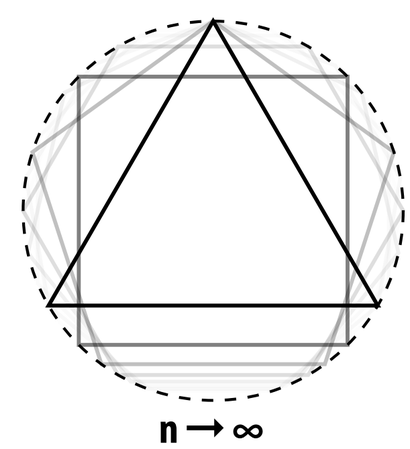
CURVATURE

$$\kappa = 1$$

SYMMETRIC



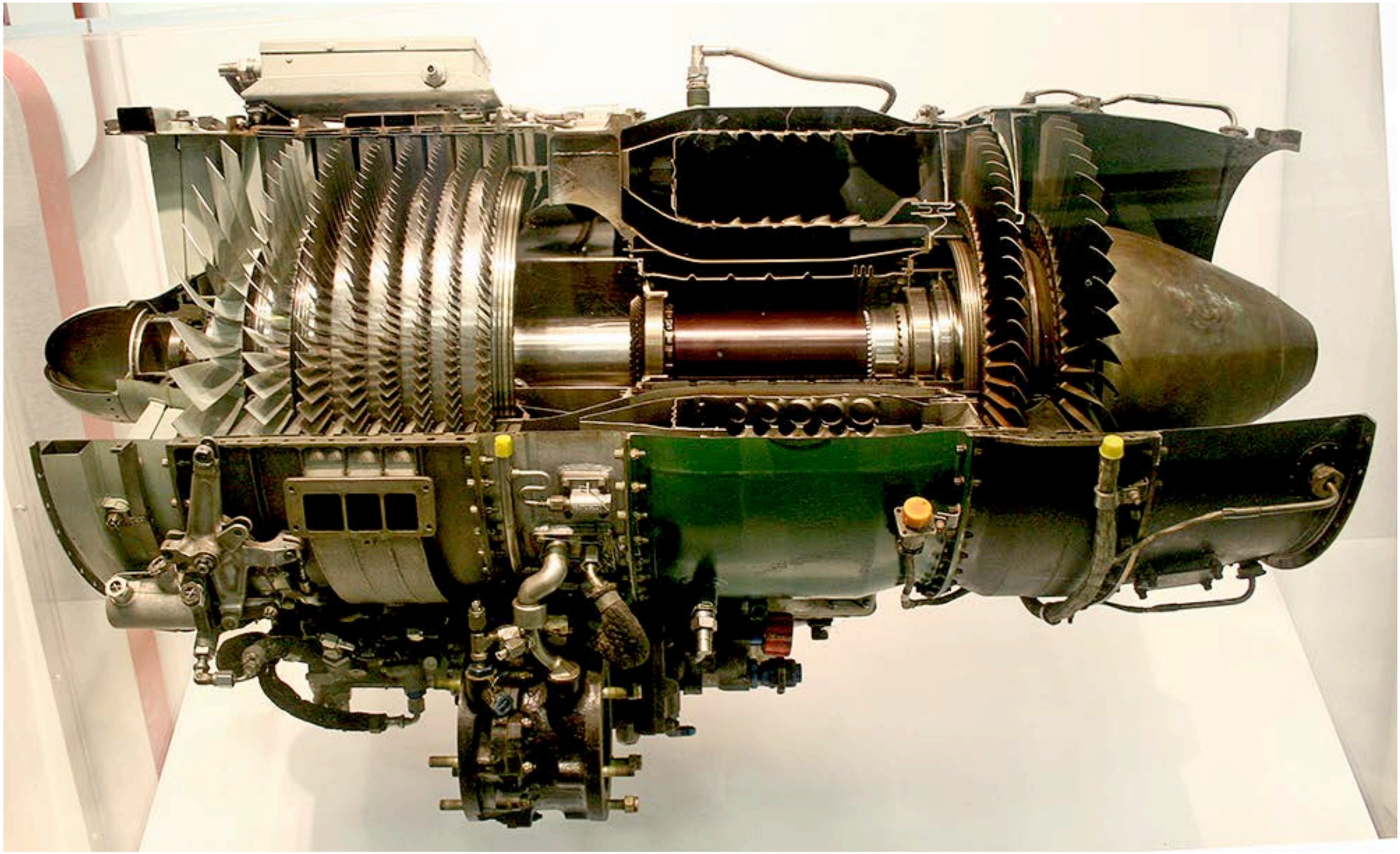
DISCRETE



Given all these options, what's the <u>best</u> way to encode geometry on a computer?

Examples of geometry

Examples of geometry



Examples of geometry

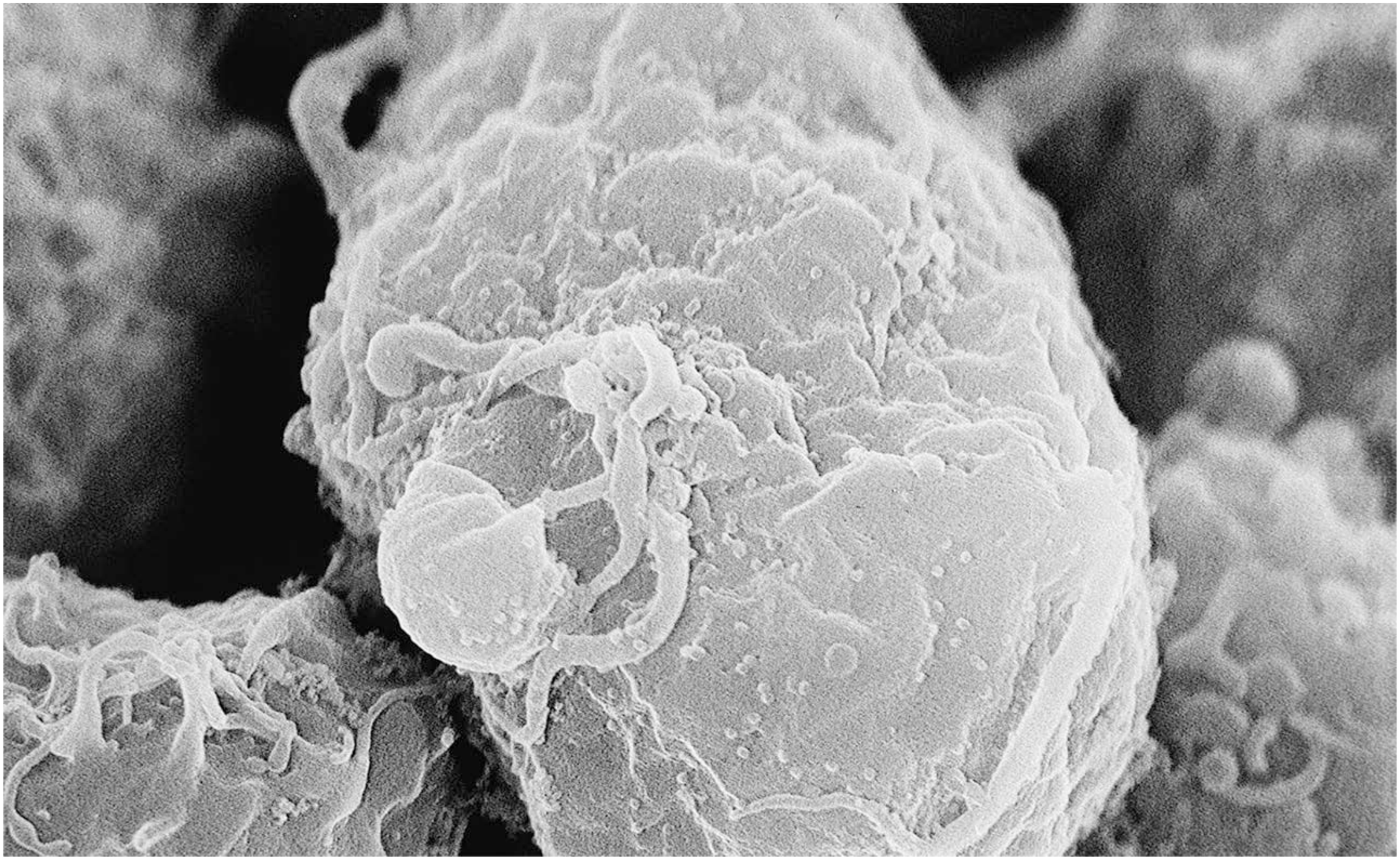
Examples of geometry

Examples of geometry

Examples of geometry

Examples of geometry

Examples of geometry



It’s a Jungle Out There!

No one "best" choice—geometry is hard!

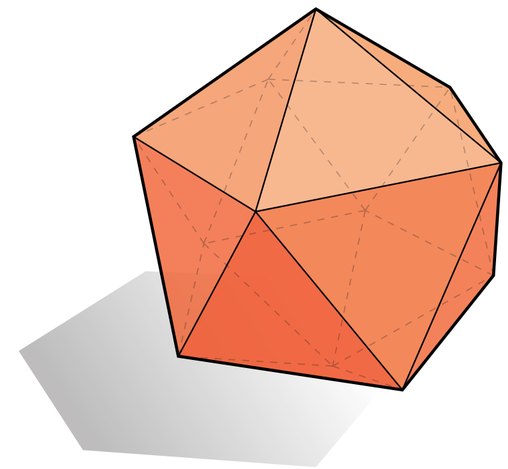
"I hate meshes.
I cannot believe how hard this is.
Geometry is hard."

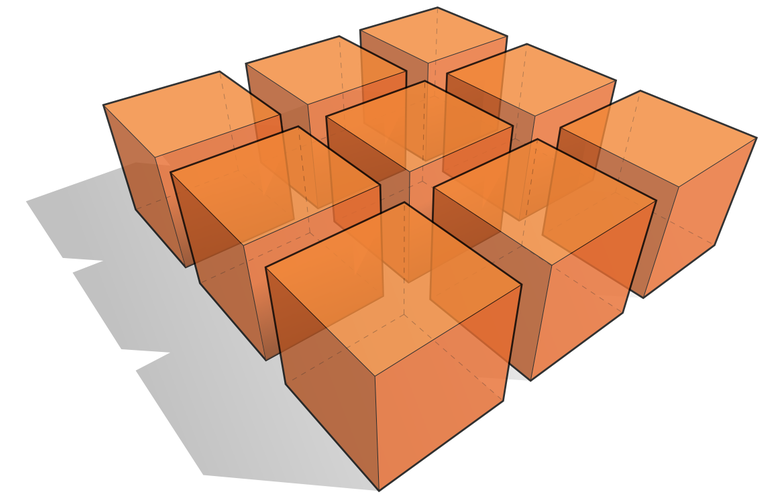
—David Baraff

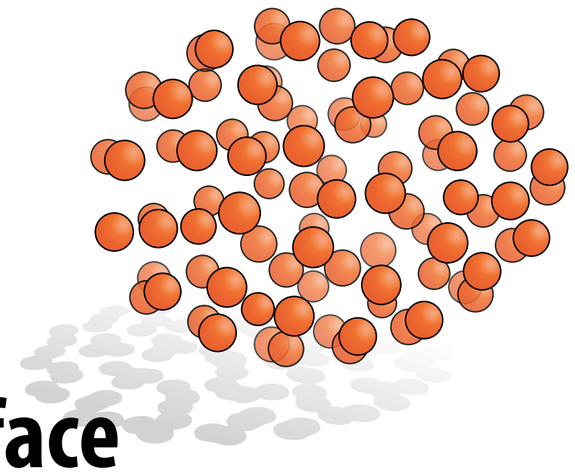
Senior Research Scientist
Pixar Animation Studios

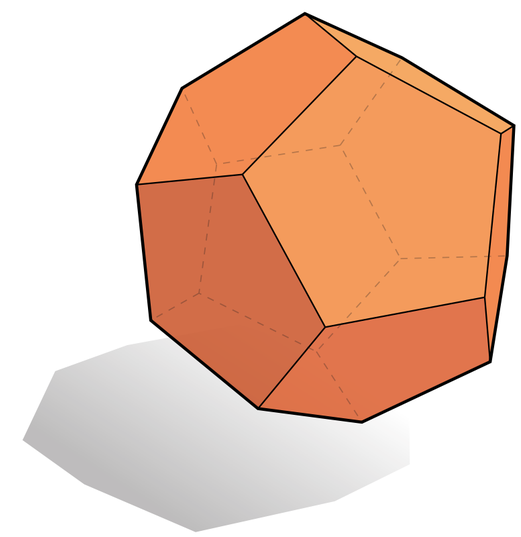
Many ways to digitally encode geometry

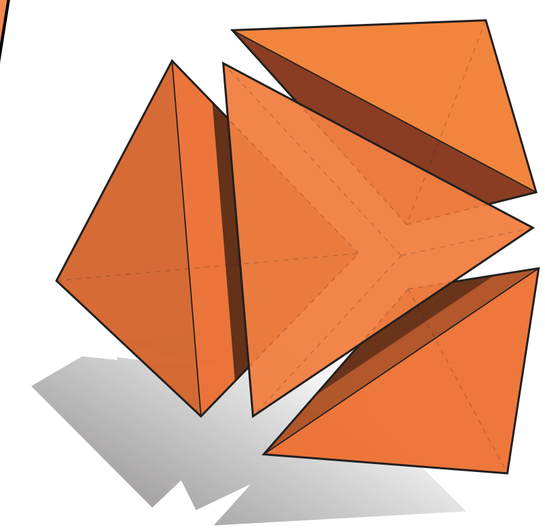
- **EXPLICIT**
  - point cloud
  - polygon mesh
  - subdivision, NURBS
  - ...
- **IMPLICIT**
  - level set
  - algebraic surface
  - L-systems
  - ...
- **Each choice best suited to a different task/type of geometry**





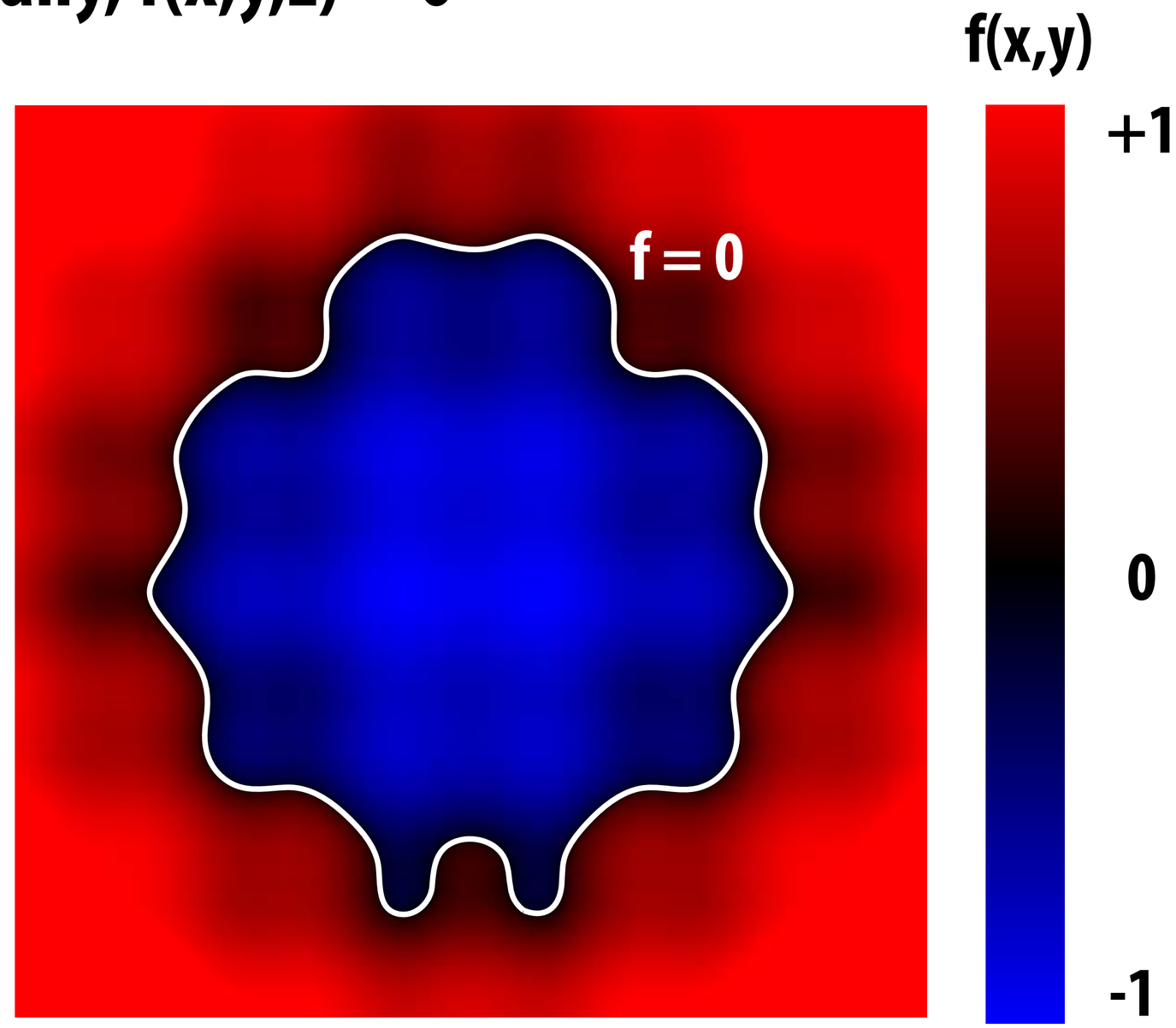






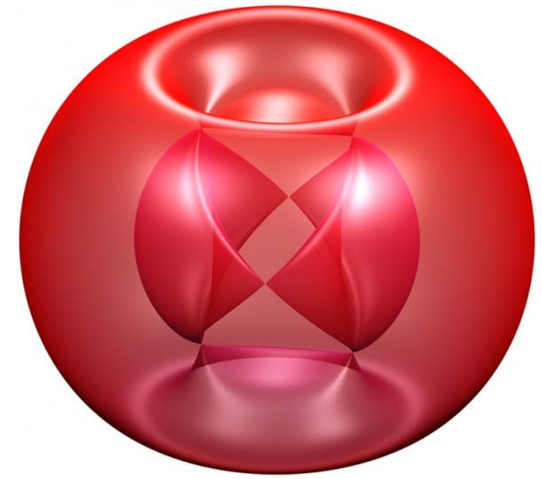
"Implicit" Representations of Geometry

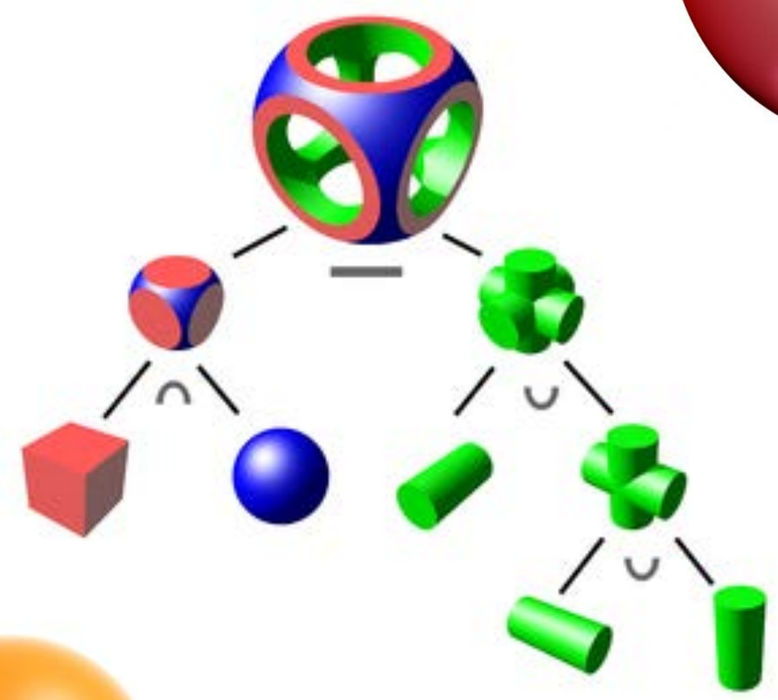
- Points aren't known directly, but satisfy some relationship
- E.g., unit sphere is all points such that $x^2+y^2+z^2=1$
- More generally, $f(x,y,z) = 0$

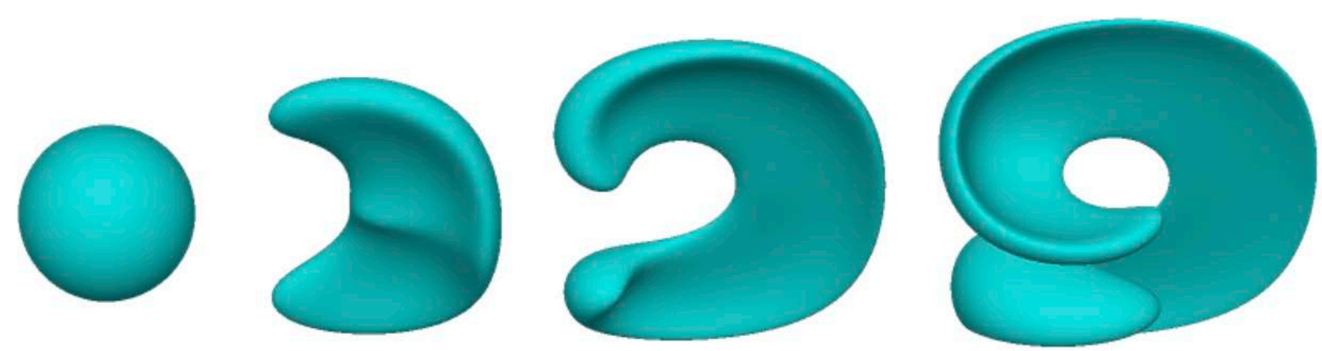


Many implicit representations in graphics

- algebraic surfaces
- constructive solid geometry
- level set methods
- blobby surfaces
- fractals
- ...







(Will see some of these a bit later.)

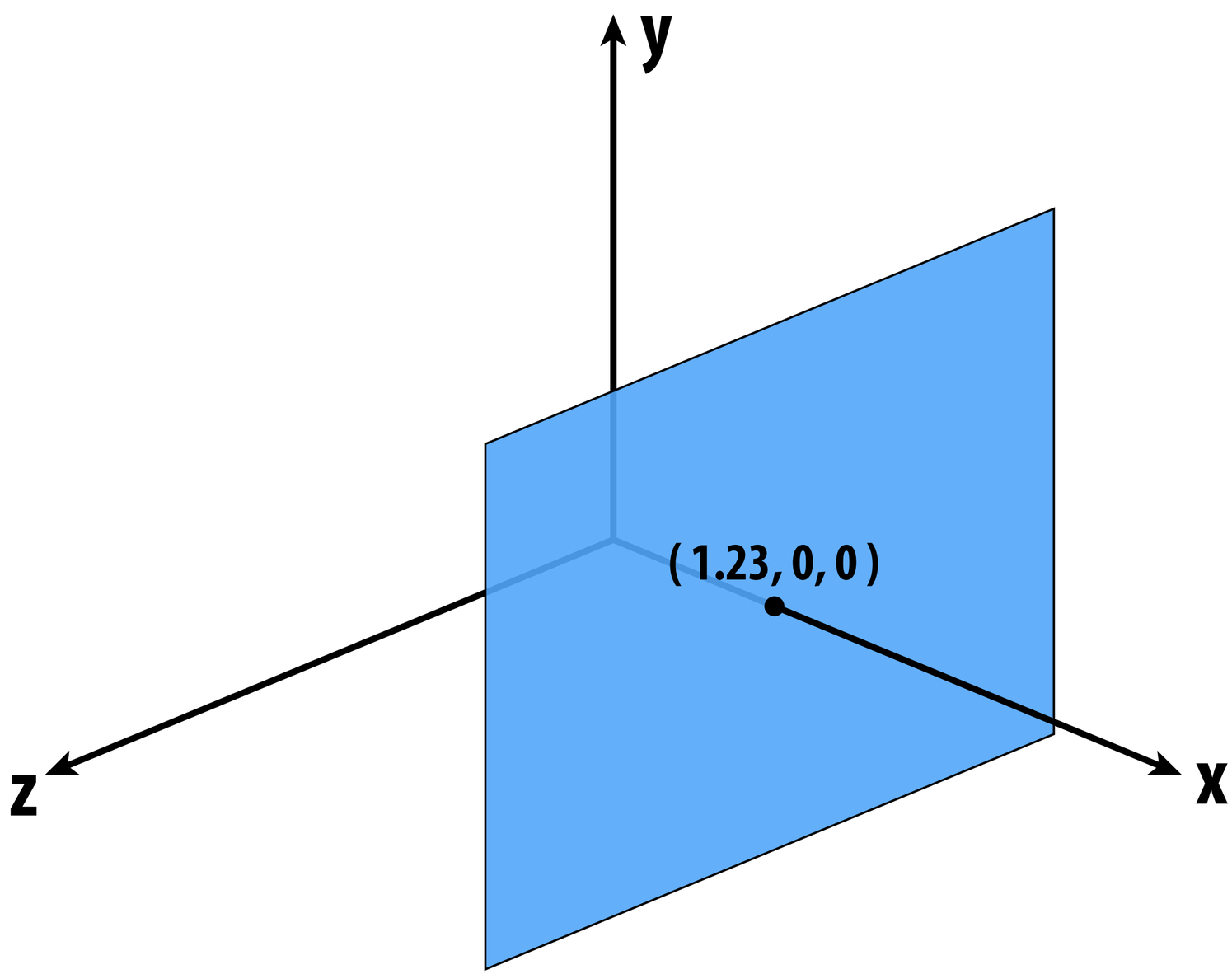
But first, let's play a game:

I'm thinking of an implicit surface f(x,y,z)=0.

Find *any* point on it.

Give up?

My function was f(x,y,z) = x - 1.23 (a plane):



Observation: implicit surfaces make some tasks hard (like sampling)

Let's play another game.

I have a new surface $f(x,y,z) = x^2 + y^2 + z^2 - 1$.

I want to see if a point is *inside* it.

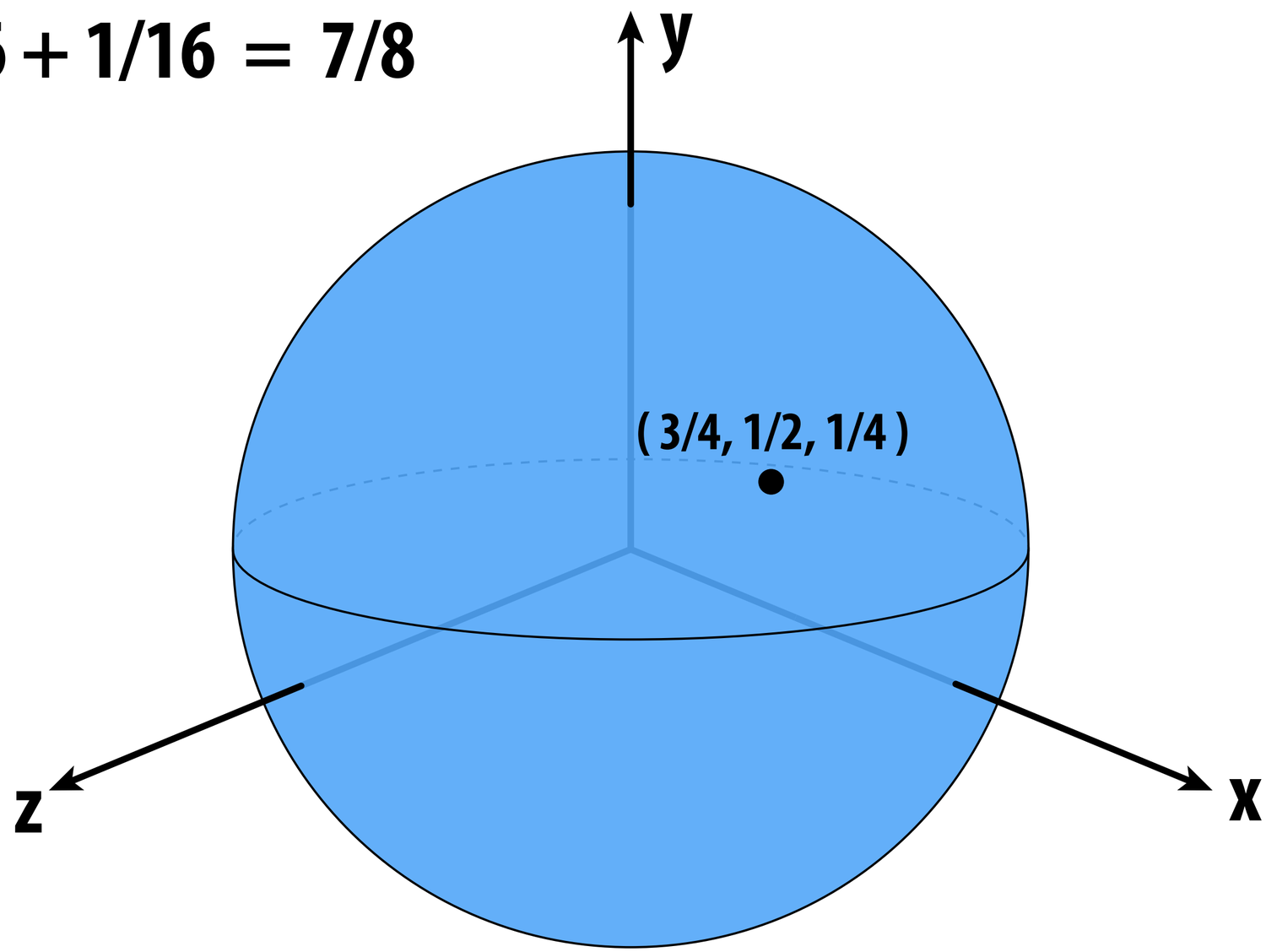
Check if this point is inside the unit sphere

How about the point (3/4, 1/2, 1/4)?

$9/16 + 4/16 + 1/16 = 7/8$

$7/8 < 1$

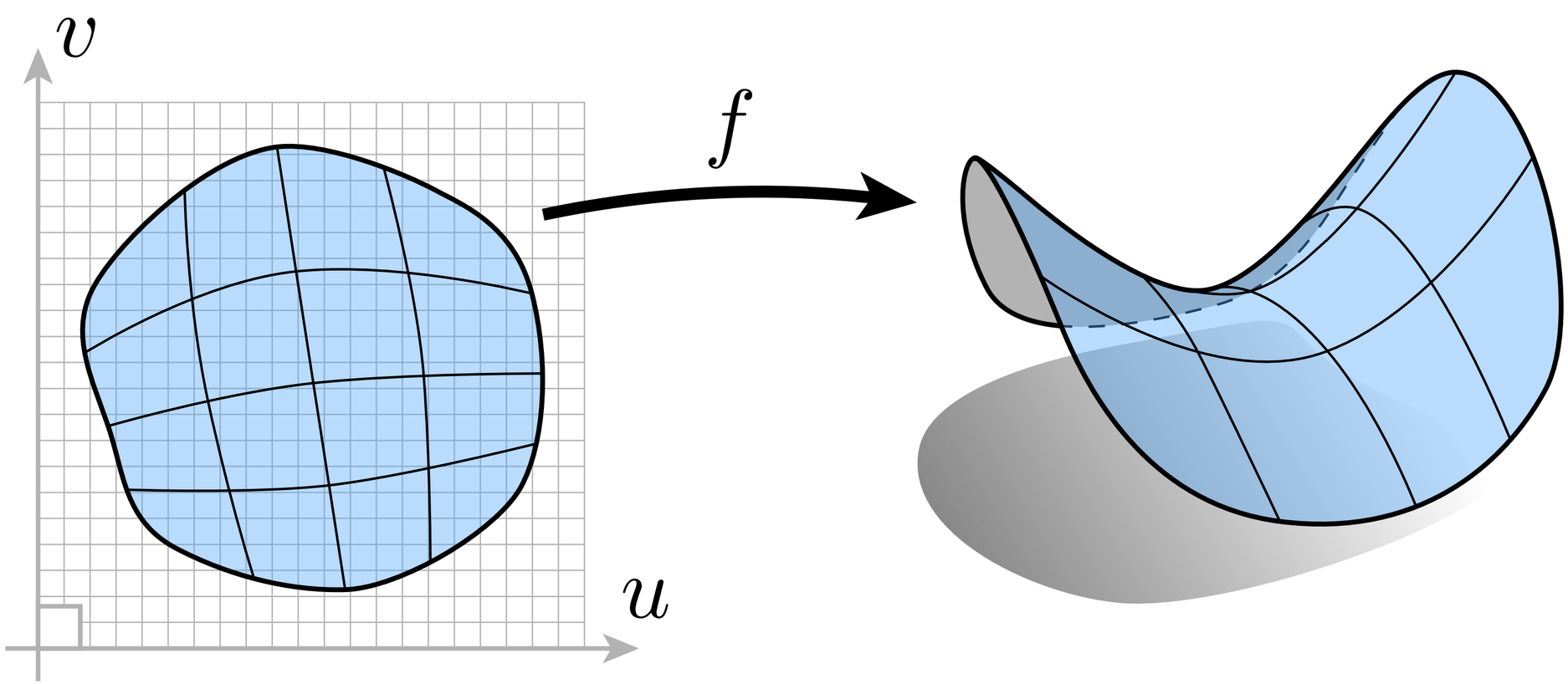
YES.



Implicit surfaces make other tasks easy (like inside/outside tests).

"Explicit" Representations of Geometry

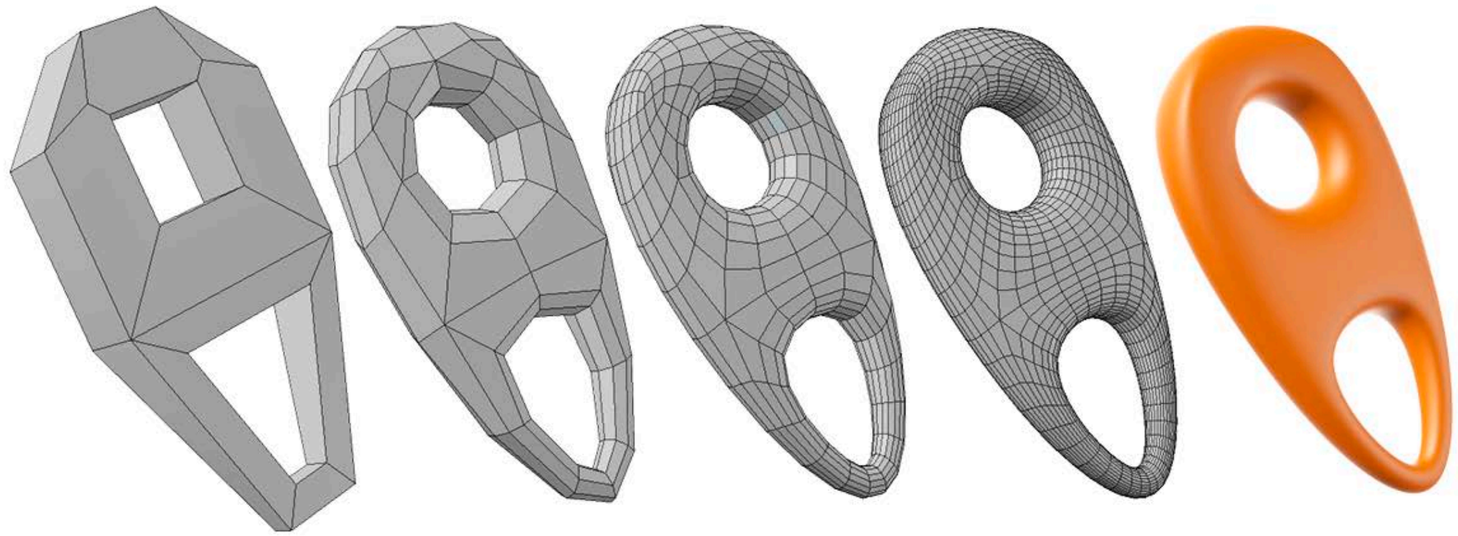
- **All points are given directly**
- **E.g., points on sphere are** $(\cos(u)\sin(v), \sin(u)\sin(v), \cos(v))$,
  $$\text{for } 0 \leq u < 2\pi \text{ and } 0 \leq v \leq \pi$$
- **More generally:** $f : \mathbb{R}^2 \to \mathbb{R}^3; (u, v) \mapsto (x, y, z)$

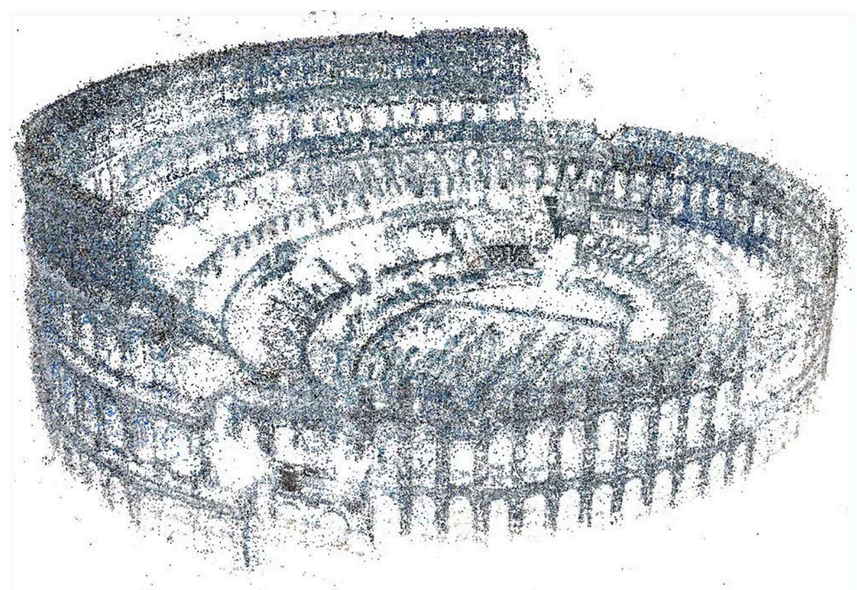


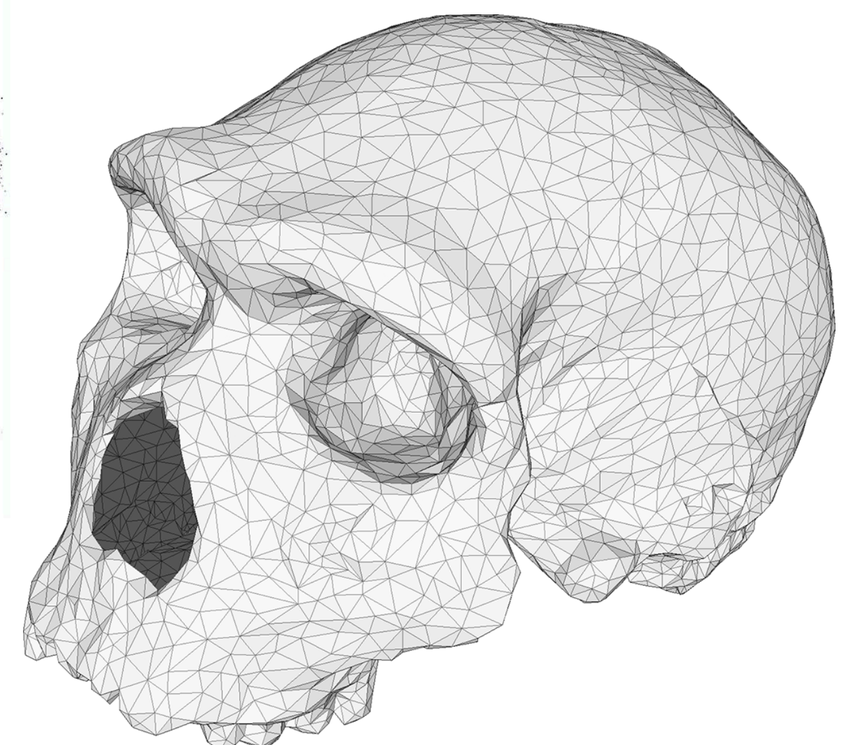
- **(Might have a bunch of these maps, e.g., one per triangle!)**

Many explicit representations in graphics

- triangle meshes
- polygon meshes
- subdivision surfaces
- NURBS
- point clouds
- ...







(Will see some of these a bit later.)

But first, let's play a game:

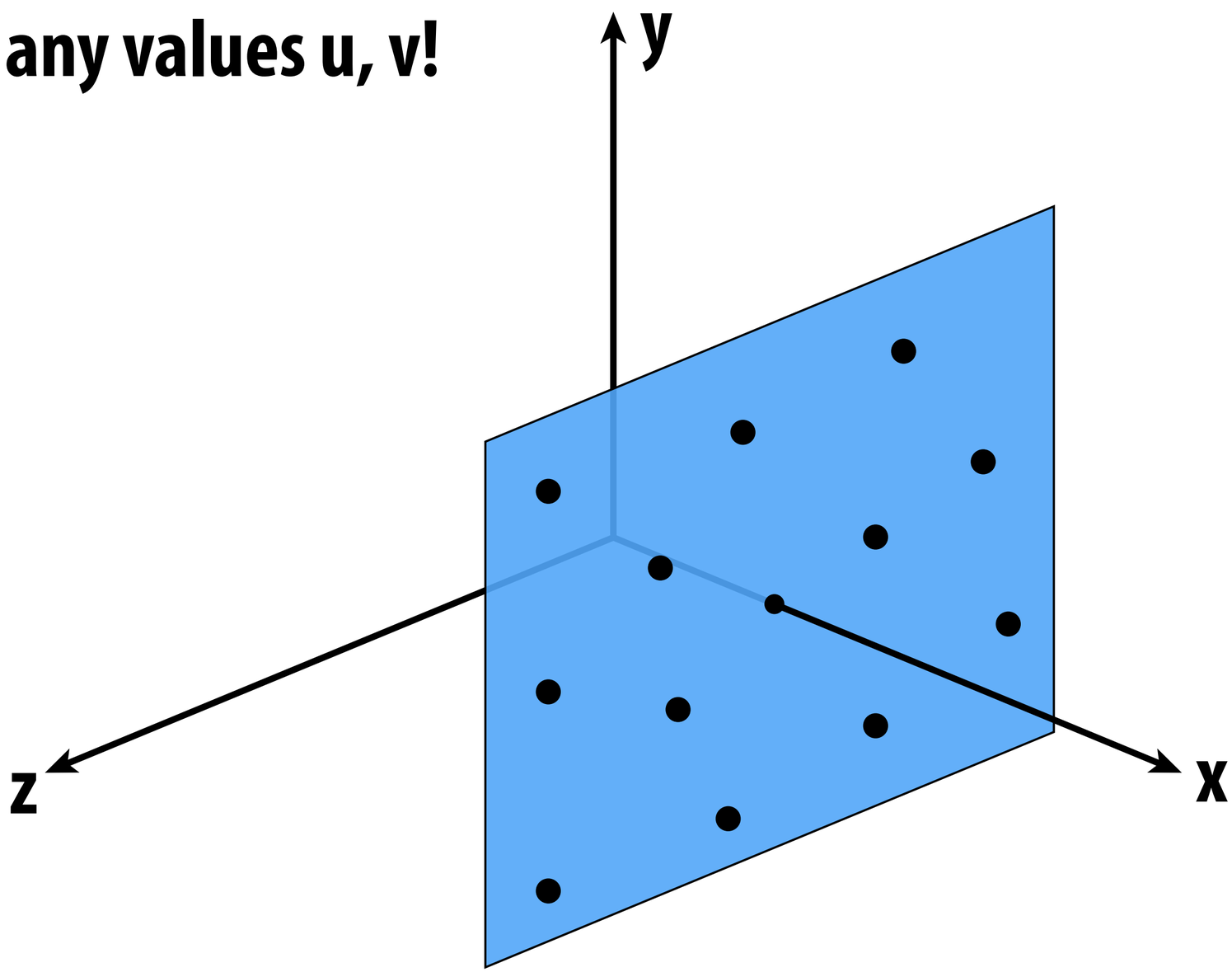
I'll give you an explicit surface.

You give me some points on it.

Sampling an explicit surface

My surface is f(u, v) = (1.23, u, v).

Just plug in any values u, v!



Explicit surfaces make some tasks easy (like sampling).

Let's play another game.

I have a new surface f(u,v).

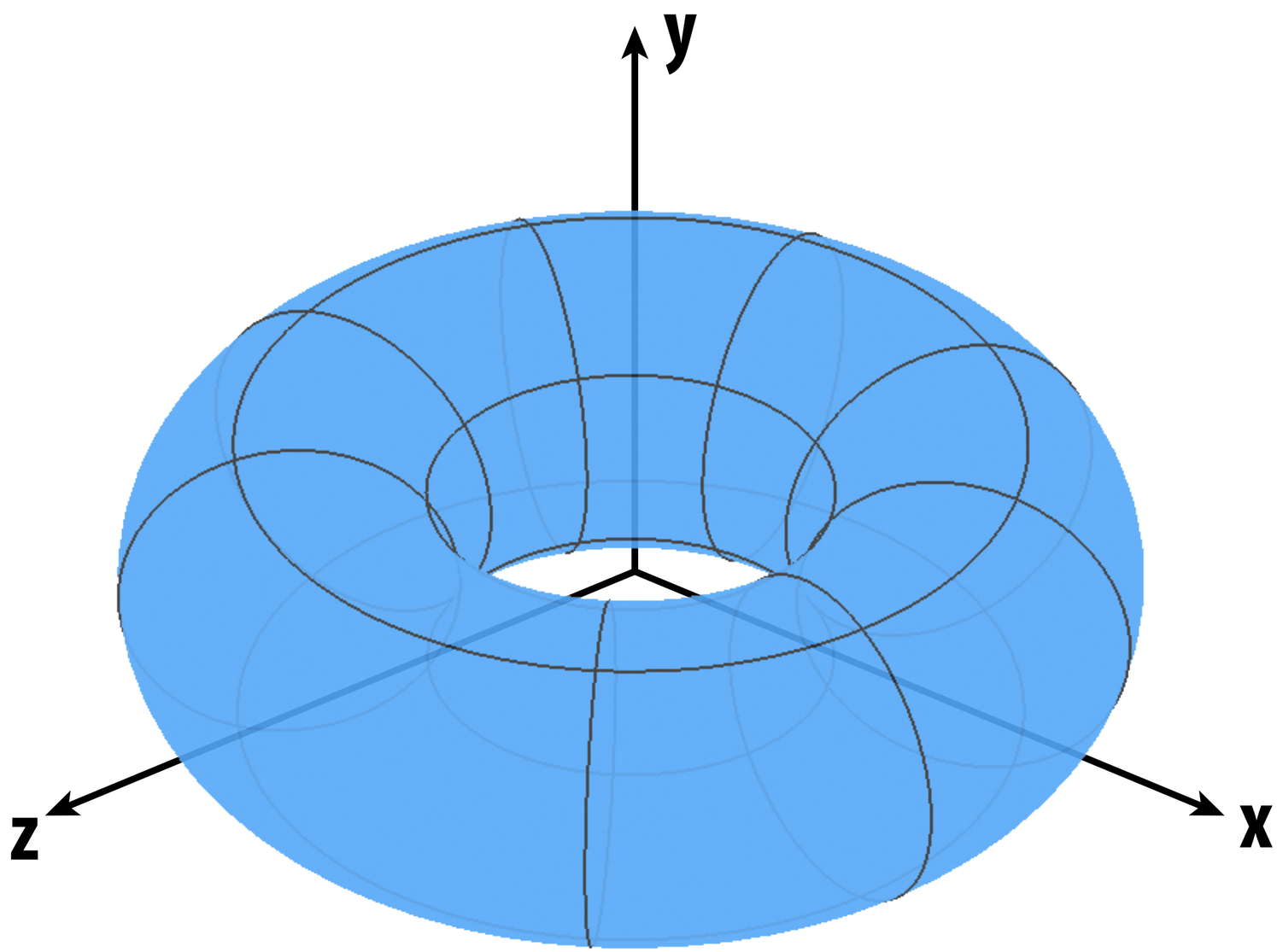
I want to see if a point is *inside* it.

Check if this point is inside the torus

My surface is f(u,v) = ((2+cos u)cos v, (2+cos u)sin v, sin u)

How about the point (1.96, -0.39, 0.9)?

...NO!



Explicit surfaces make other tasks hard (like inside/outside tests).

CONCLUSION:

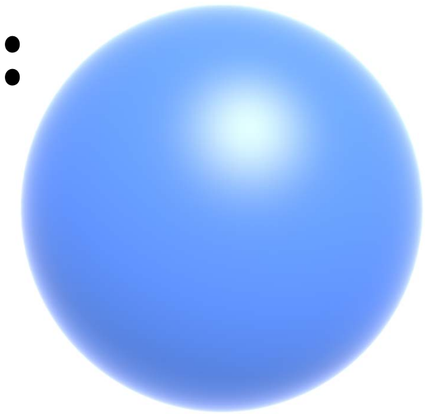
Some representations work better than others—depends on the task!

Different representations will also be better suited to different types of geometry.

Let's take a look at some common representations used in computer graphics.

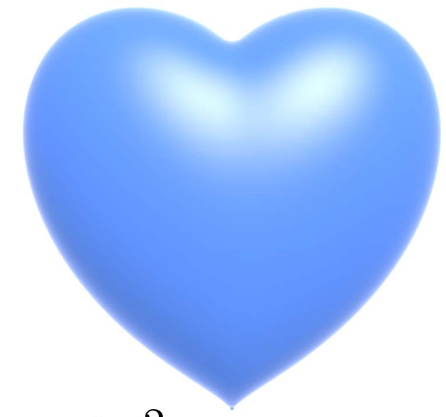
Algebraic Surfaces (Implicit)

- **Surface is zero set of a polynomial in x, y, z**
- **Examples:**



$x^2+y^2+z^2=1$

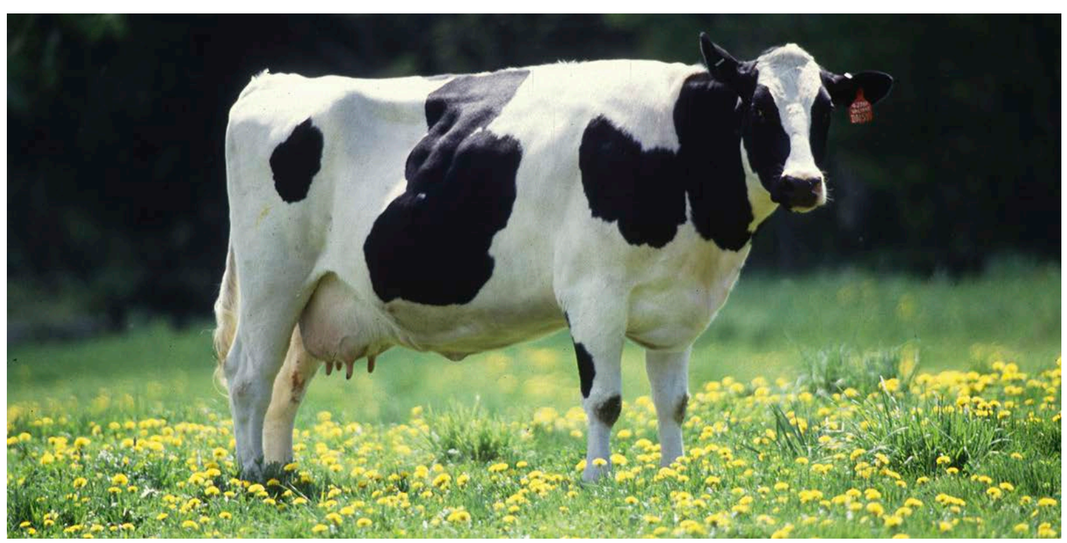
$(R-\sqrt{x^2+y^2})^2+z^2=r^2$



$(x^2+\frac{9y^2}{4}+z^2-1)^3=$

$x^2z^3+\frac{9y^2z^3}{80}$

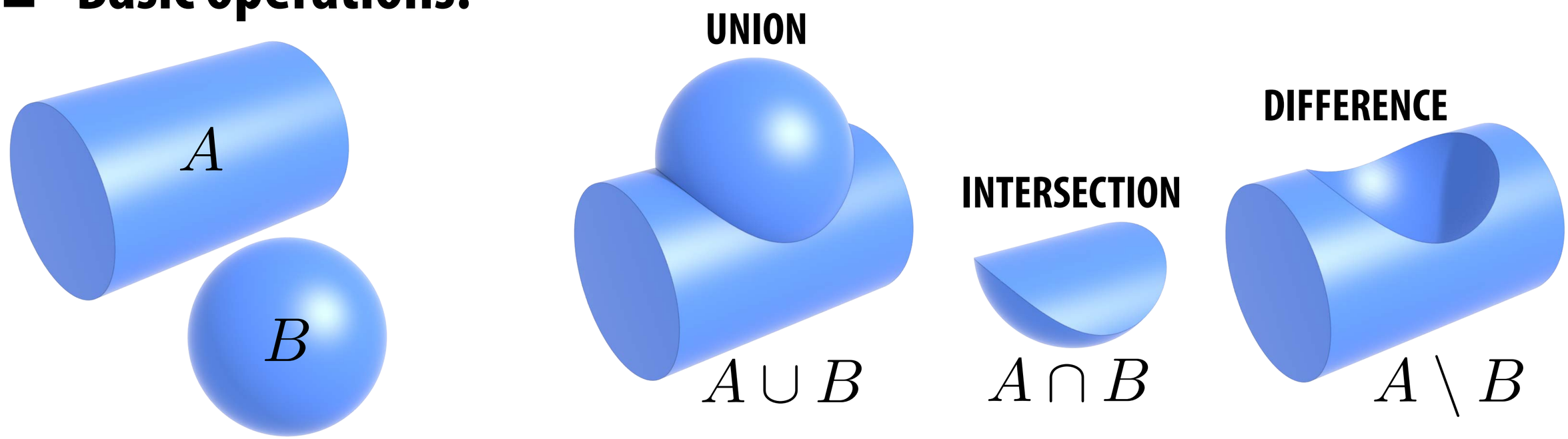
- **What about more complicated shapes?**



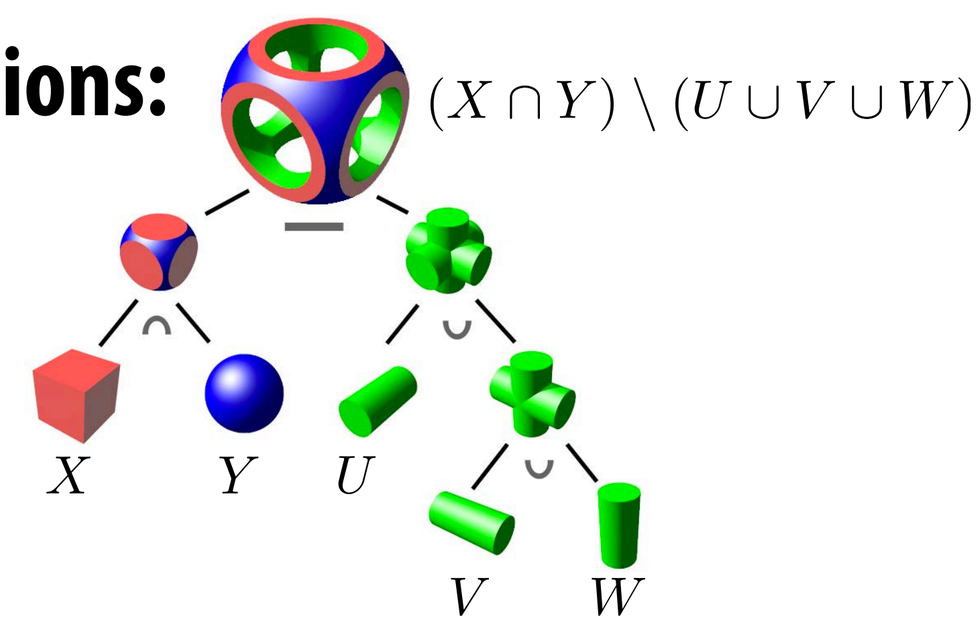
- **Very hard to come up with polynomials!**

Constructive Solid Geometry (Implicit)

- Build more complicated shapes via Boolean operations
- Basic operations:

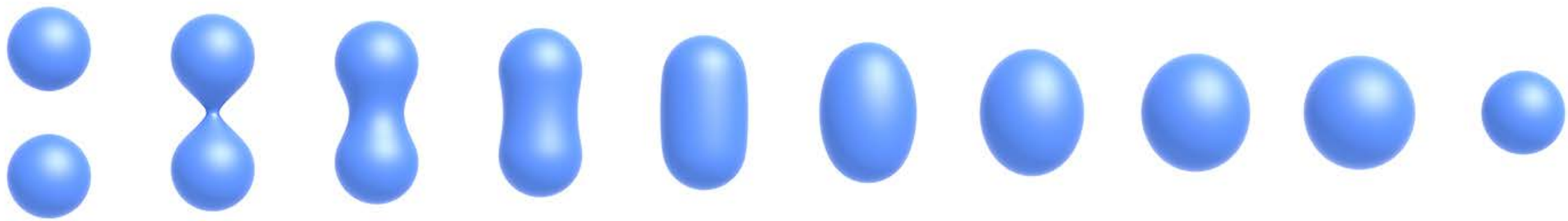


- Then chain together expressions:



Blobby Surfaces (Implicit)

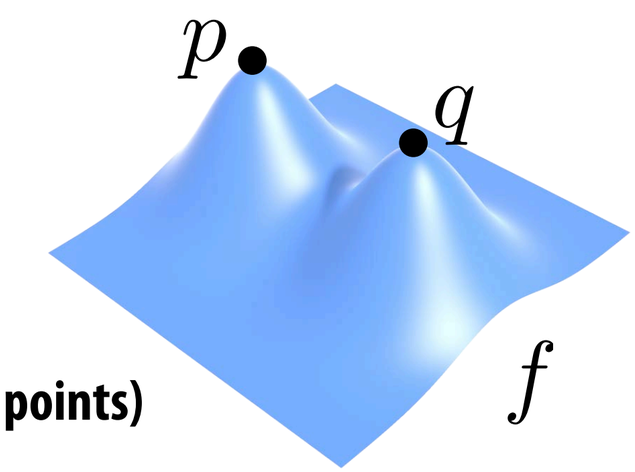
- **Instead of Booleans, gradually blend surfaces together:**

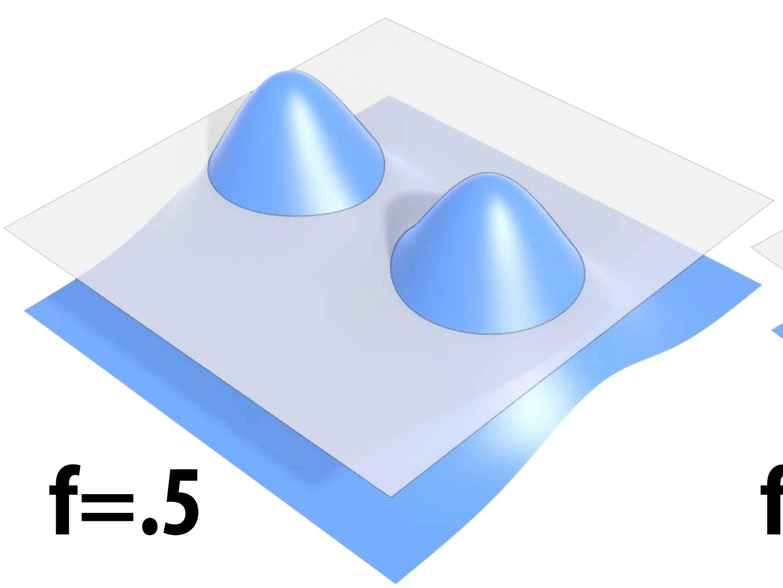


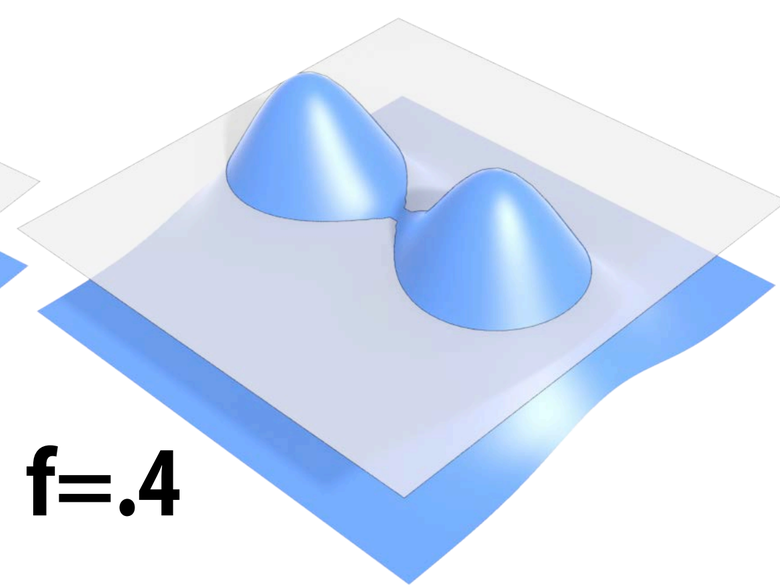
- **Easier to understand in 2D:**

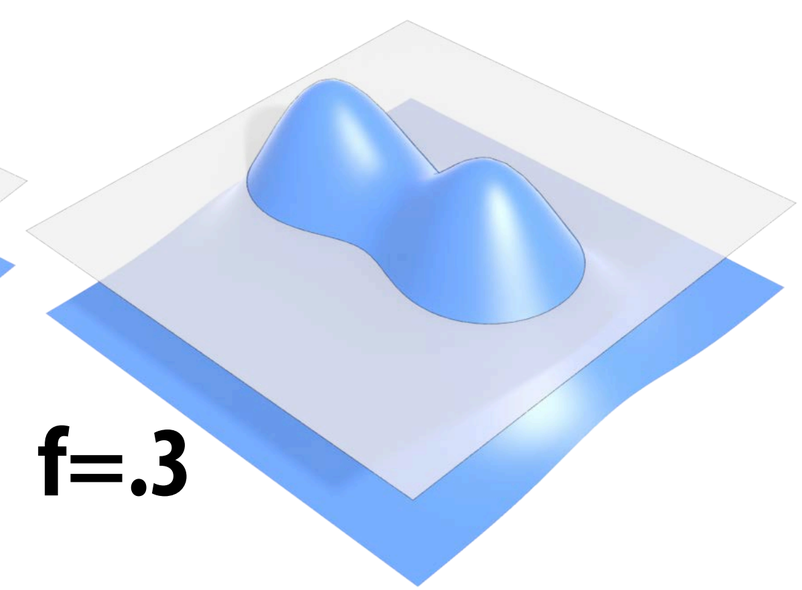
$$\phi_p(x) := e^{-|x-p|^2}$$ **(Gaussian centered at p)**

$$f := \phi_p + \phi_q$$ **(Sum of Gaussians centered at different points)**



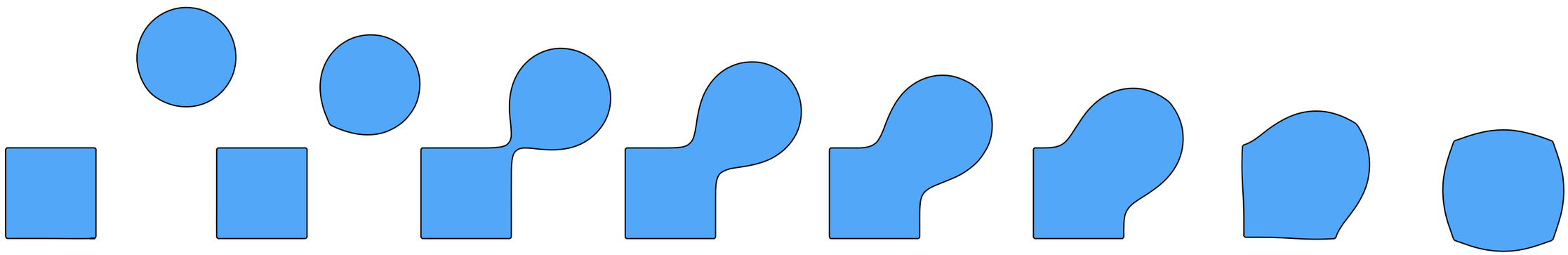






Blending Distance Functions (Implicit)

- A *distance function* gives distance to closest point on object
- Can blend any two distance functions d_1, d_2:

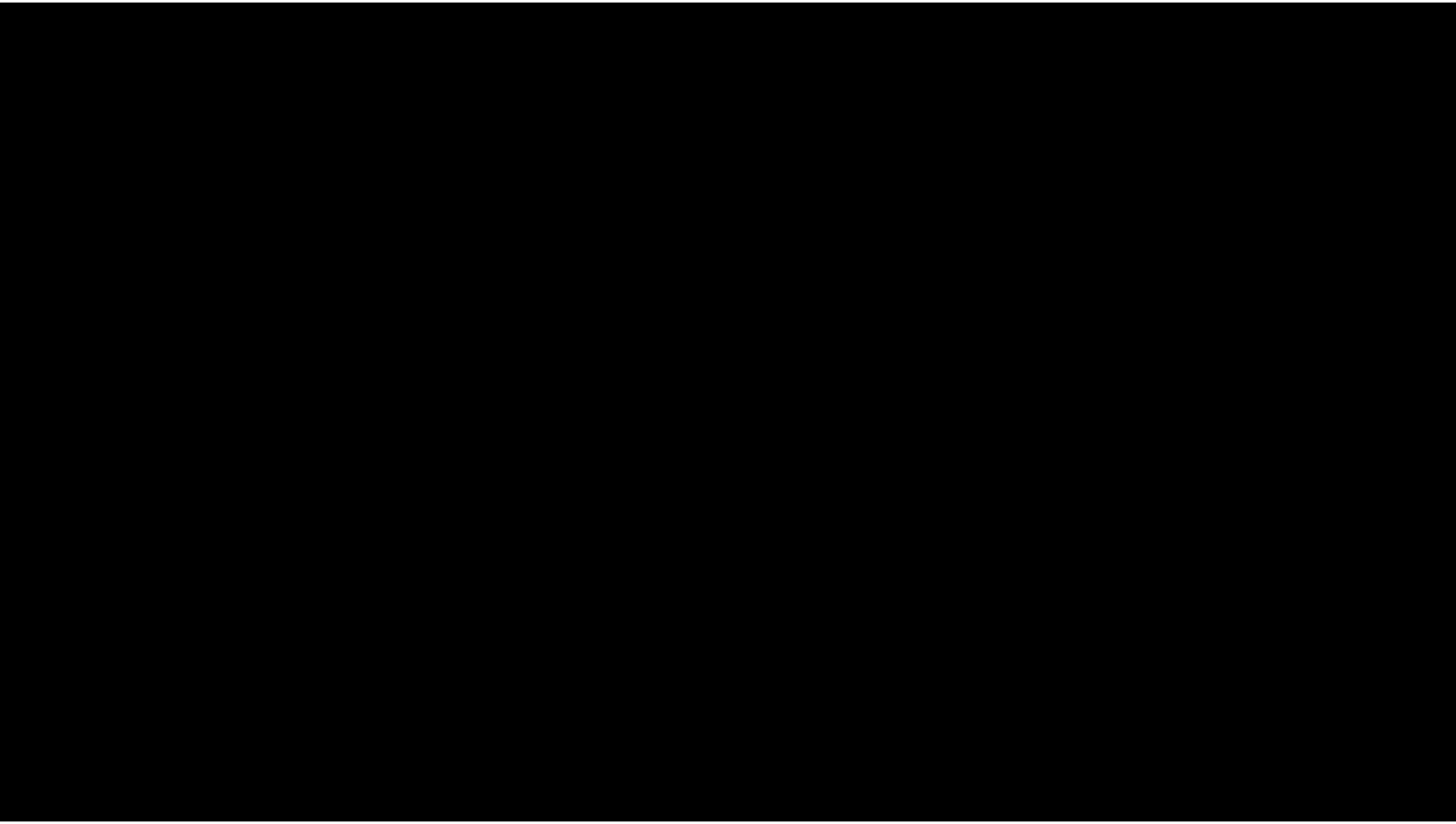


- Similar strategy to points, though many possibilities. E.g.,

$$f(x) := e^{d_1(x)^2} + e^{d_2(x)^2} - \frac{1}{2}$$

- Appearance depends on how we combine functions
- Q: How do we implement a Boolean union of $d_1(x), d_2(x)$?
- A: Just take the minimum: $f(x) = \min(d_1(x), d_2(x))$

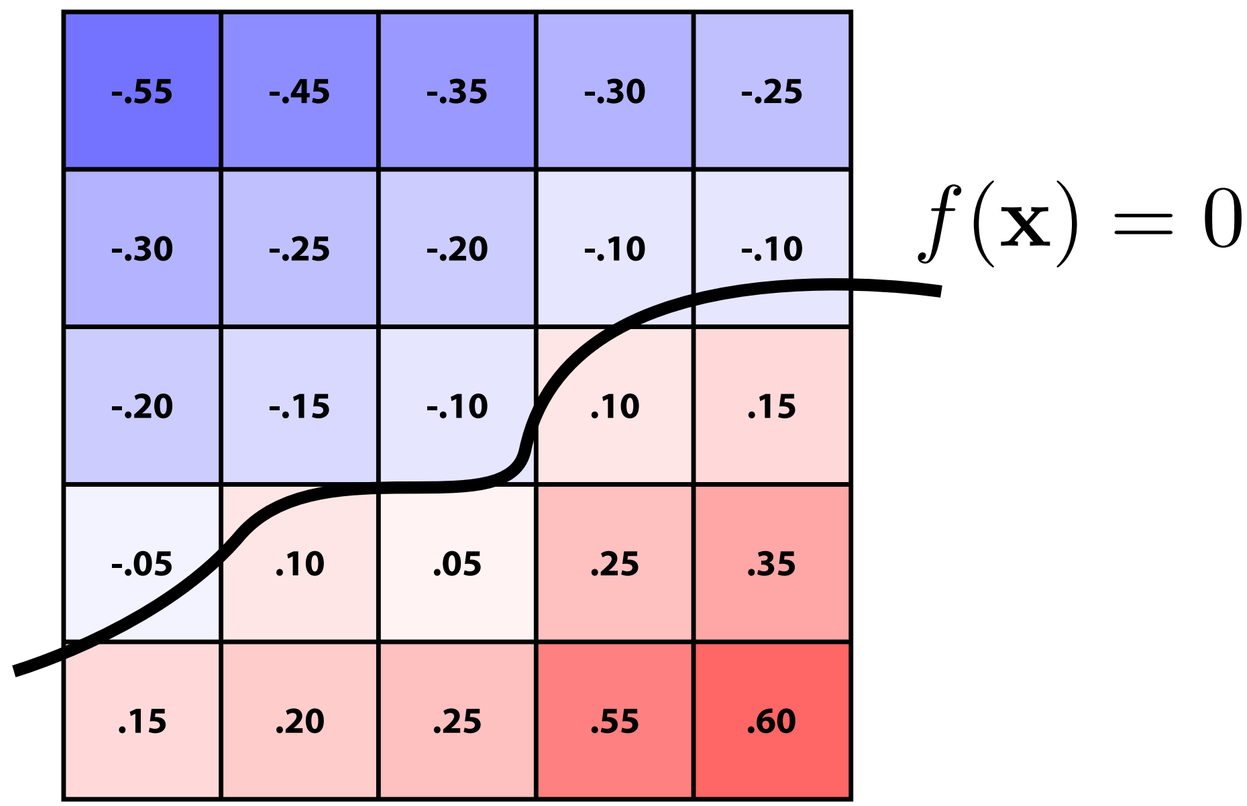
Scene of pure distance functions (not easy!)



see http://iquilezles.org/

Level Set Methods (Implicit)

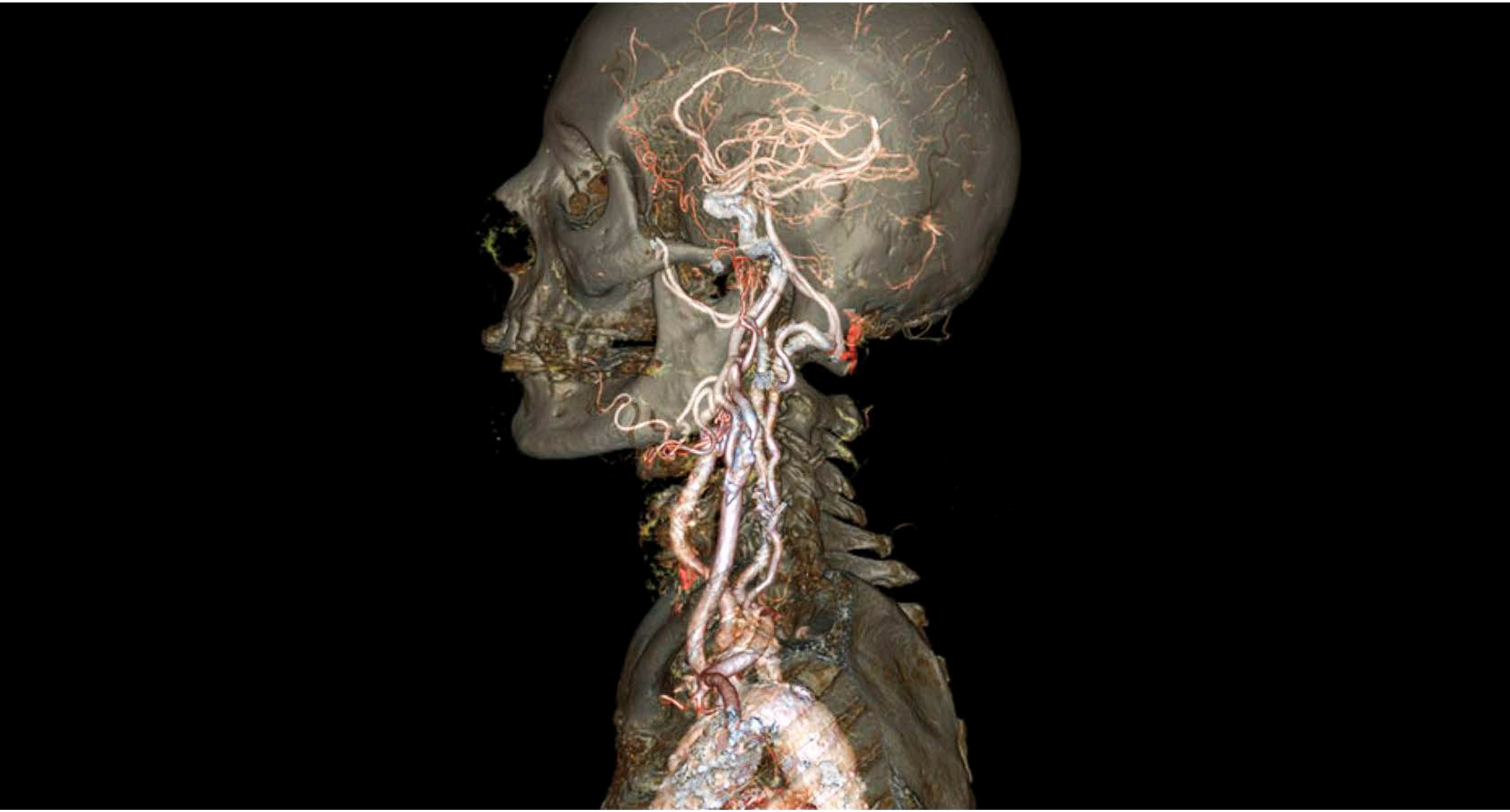
- Implicit surfaces have some nice features (e.g., merging/splitting)
- But, hard to describe complex shapes in closed form
- Alternative: store a grid of values approximating function



- Surface is found where *interpolated* values equal zero
- Provides much more explicit control over shape (like a texture)
- Unlike closed-form expressions, run into problems of <u>aliasing</u>!

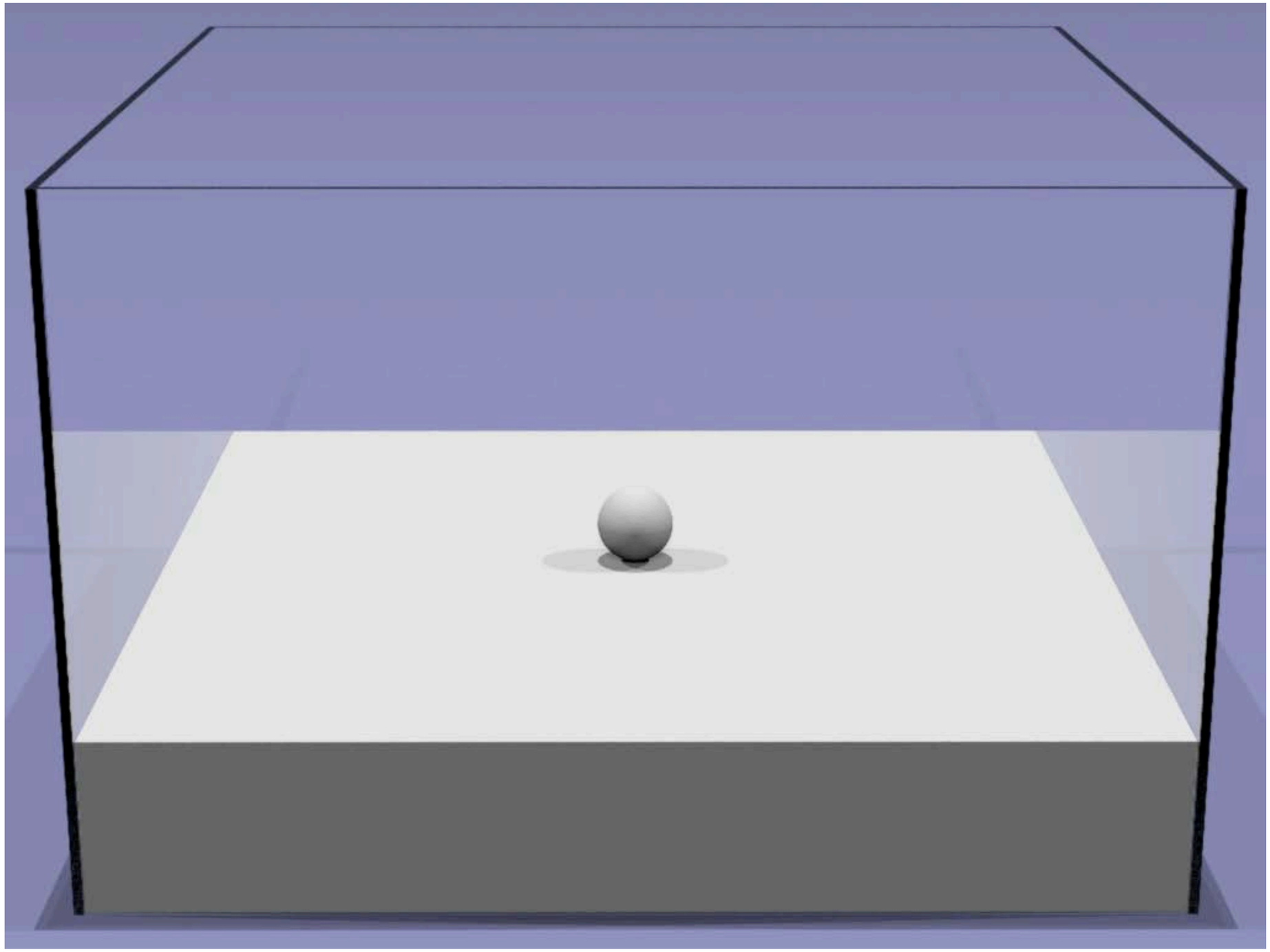
Level Sets from Medical Data (CT, MRI, etc.)

- Level sets encode, e.g., constant tissue density



Level Sets in Physical Simulation

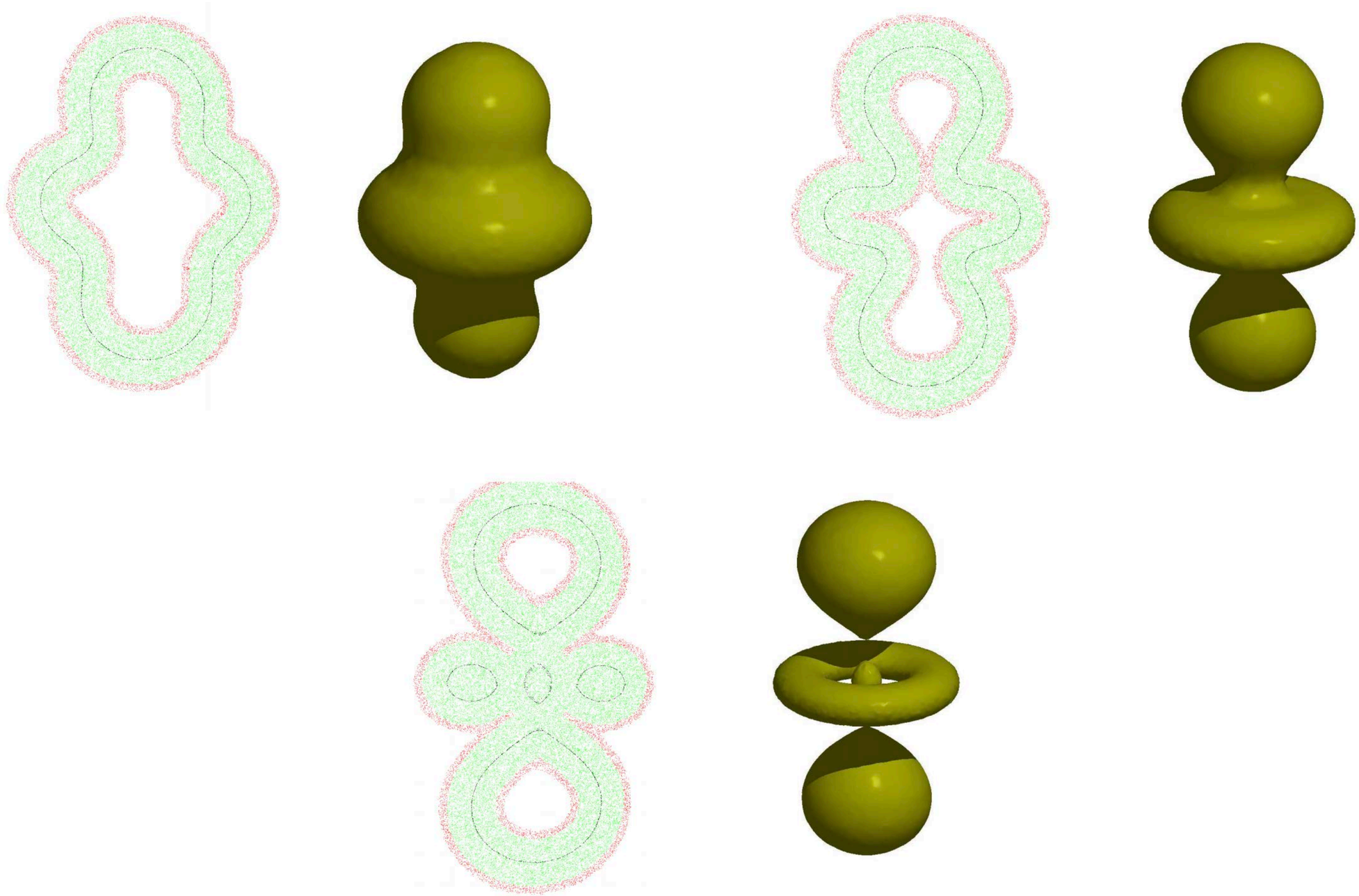
Level set encodes distance to air-liquid boundary:



see http://physbam.stanford.edu

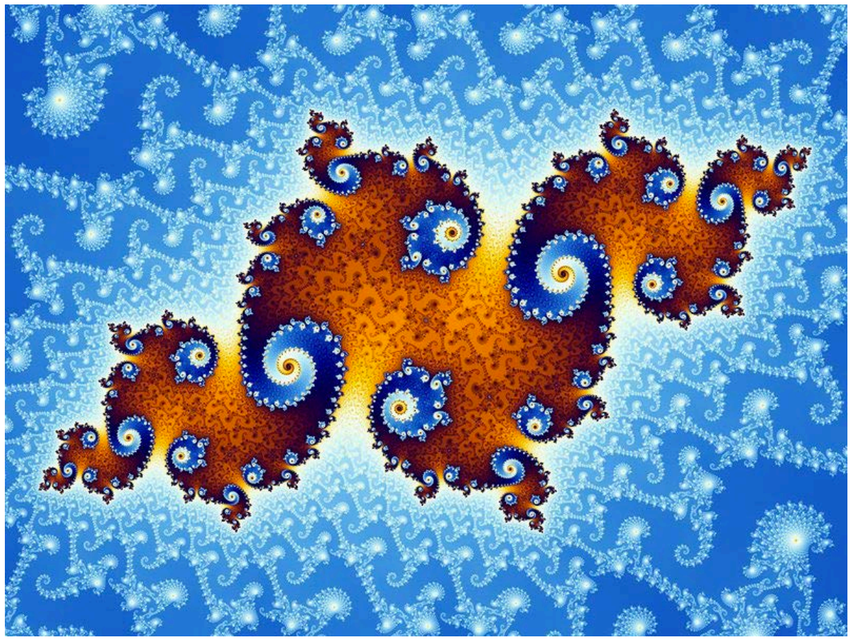
Level Set Storage

- Drawback: storage for 2D surface is now $O(n^3)$
- Can reduce cost by storing only a narrow band around surface:



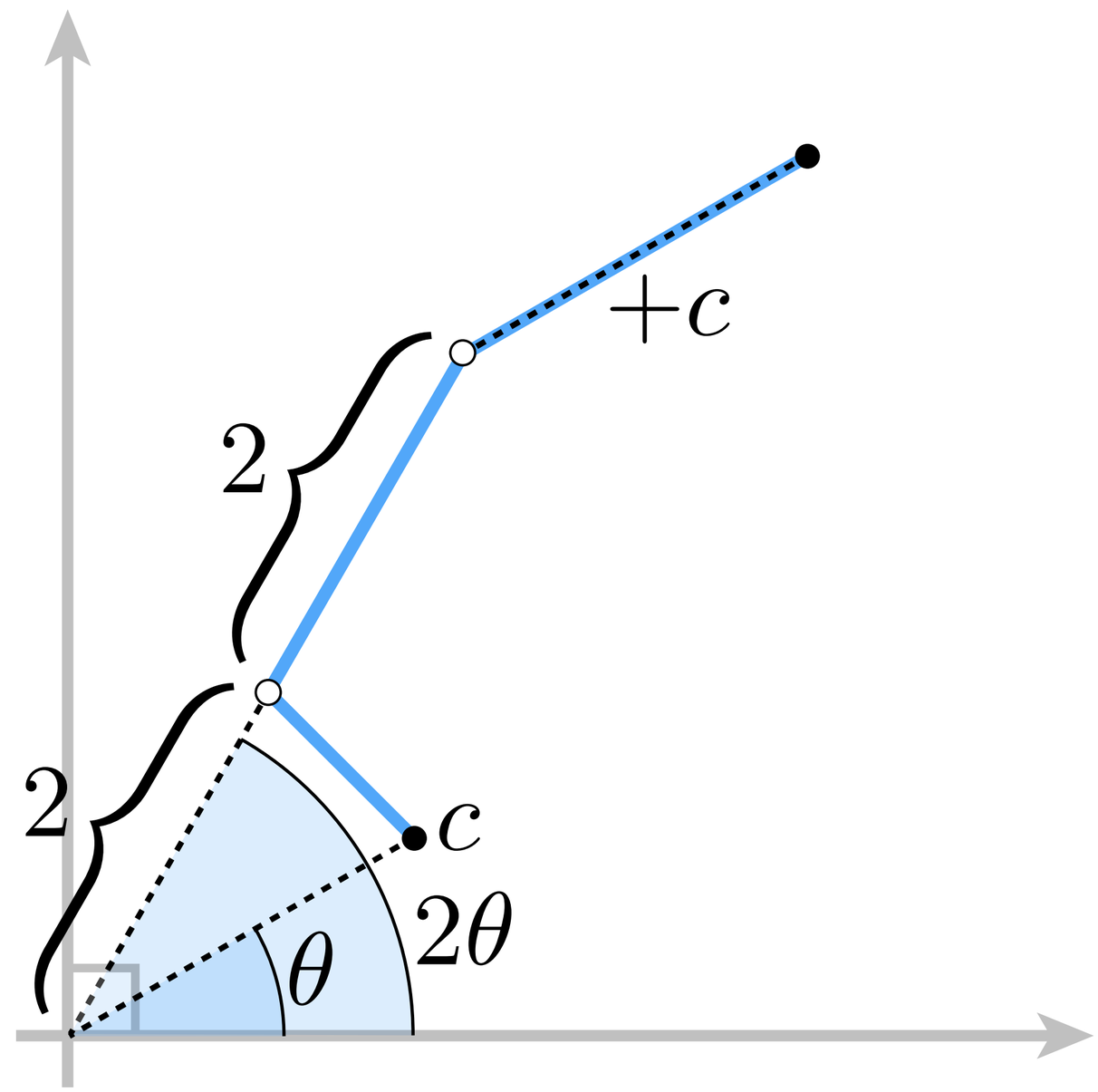
Fractals (Implicit)

- No precise definition; exhibit self-similarity, detail at all scales
- New "language" for describing natural phenomena
- Hard to control shape!



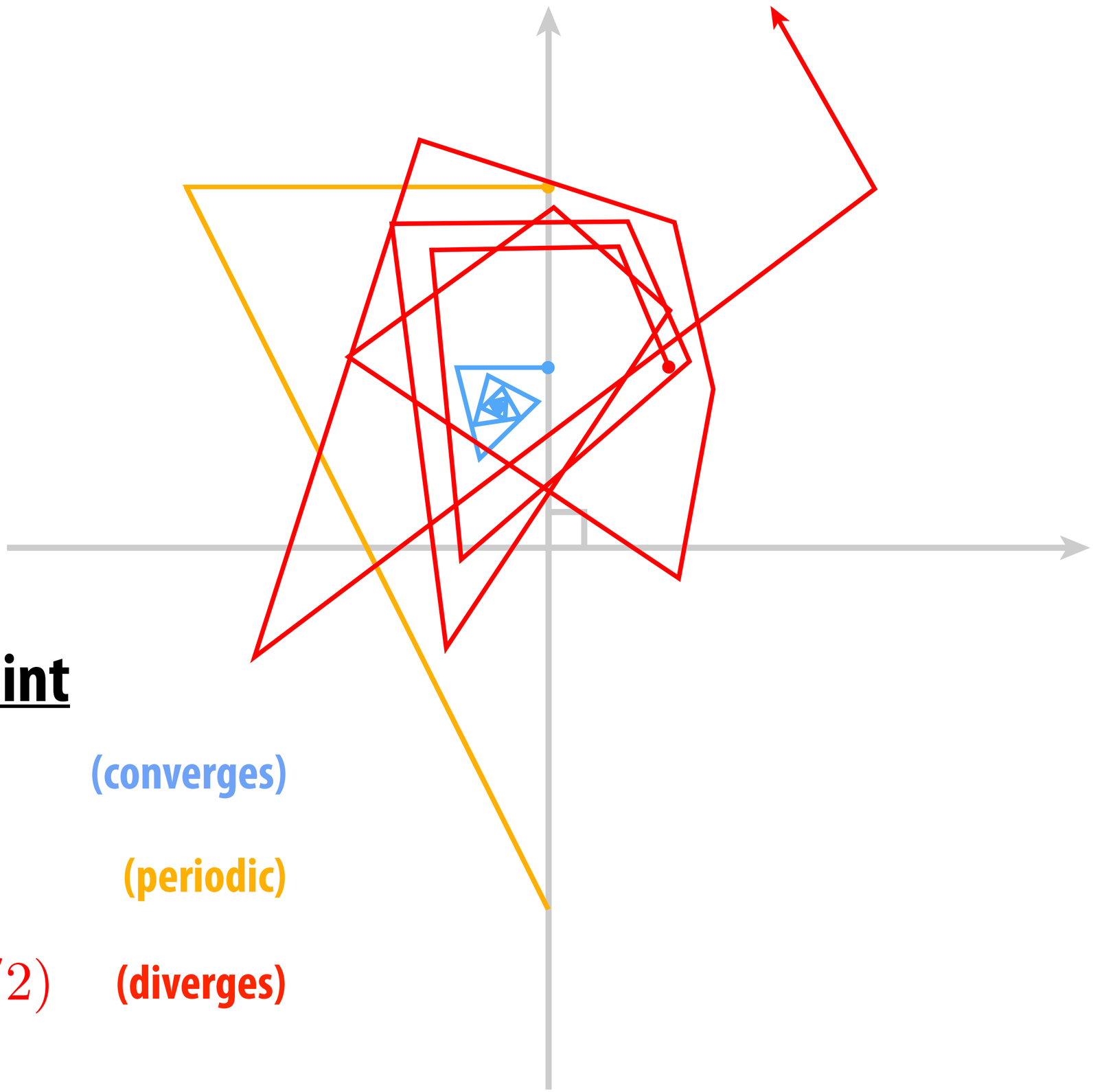
Mandelbrot Set - Definition

- For each point c in the plane:
  - double the angle
  - square the magnitude
  - add the original point c
  - repeat
- Complex version:
  - *Replace* z *with* $z^2 + c$
  - repeat



If magnitude remains bounded (never goes to ∞), it's in the Mandelbrot set.

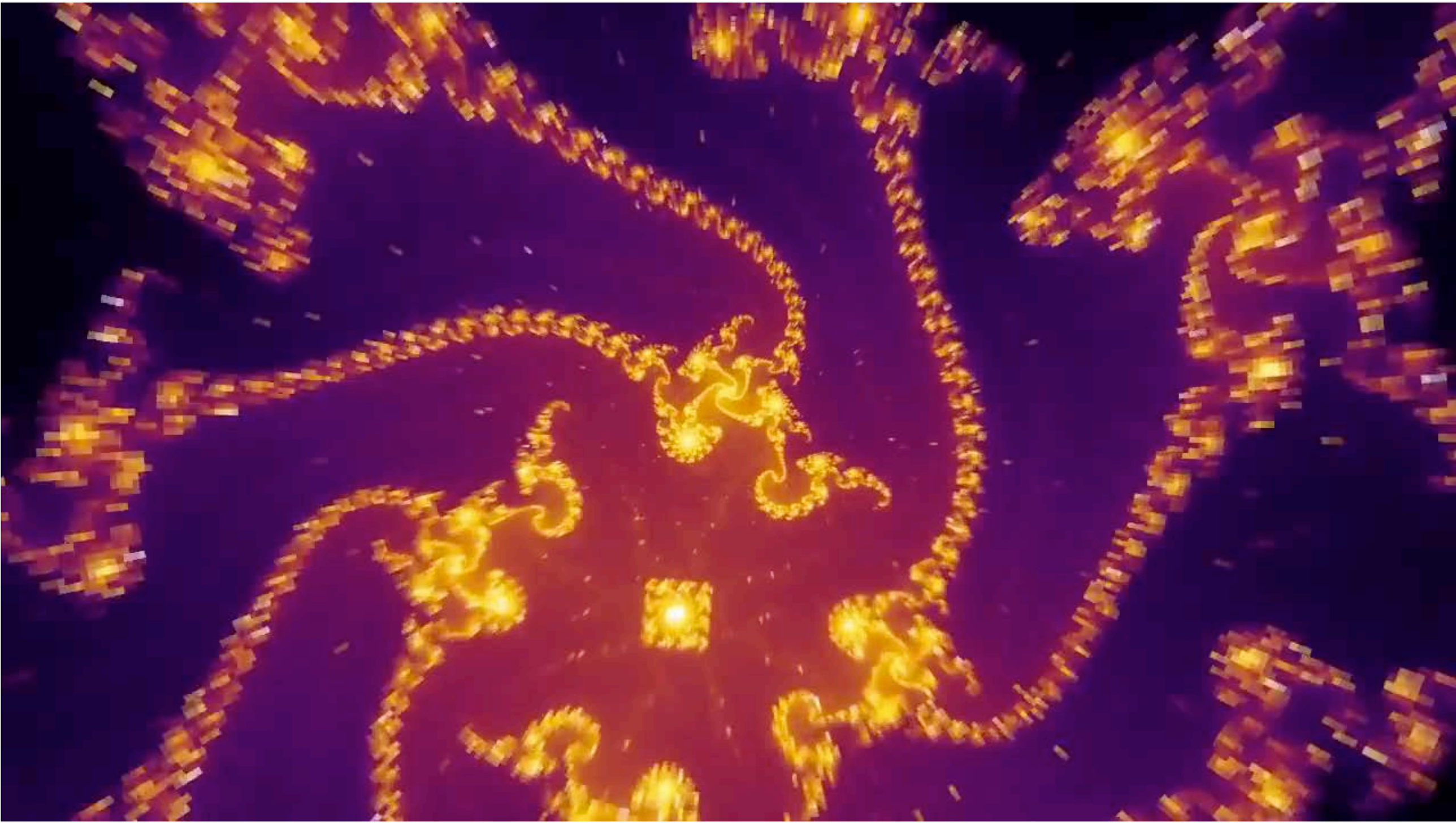
Mandelbrot Set - Examples



starting point

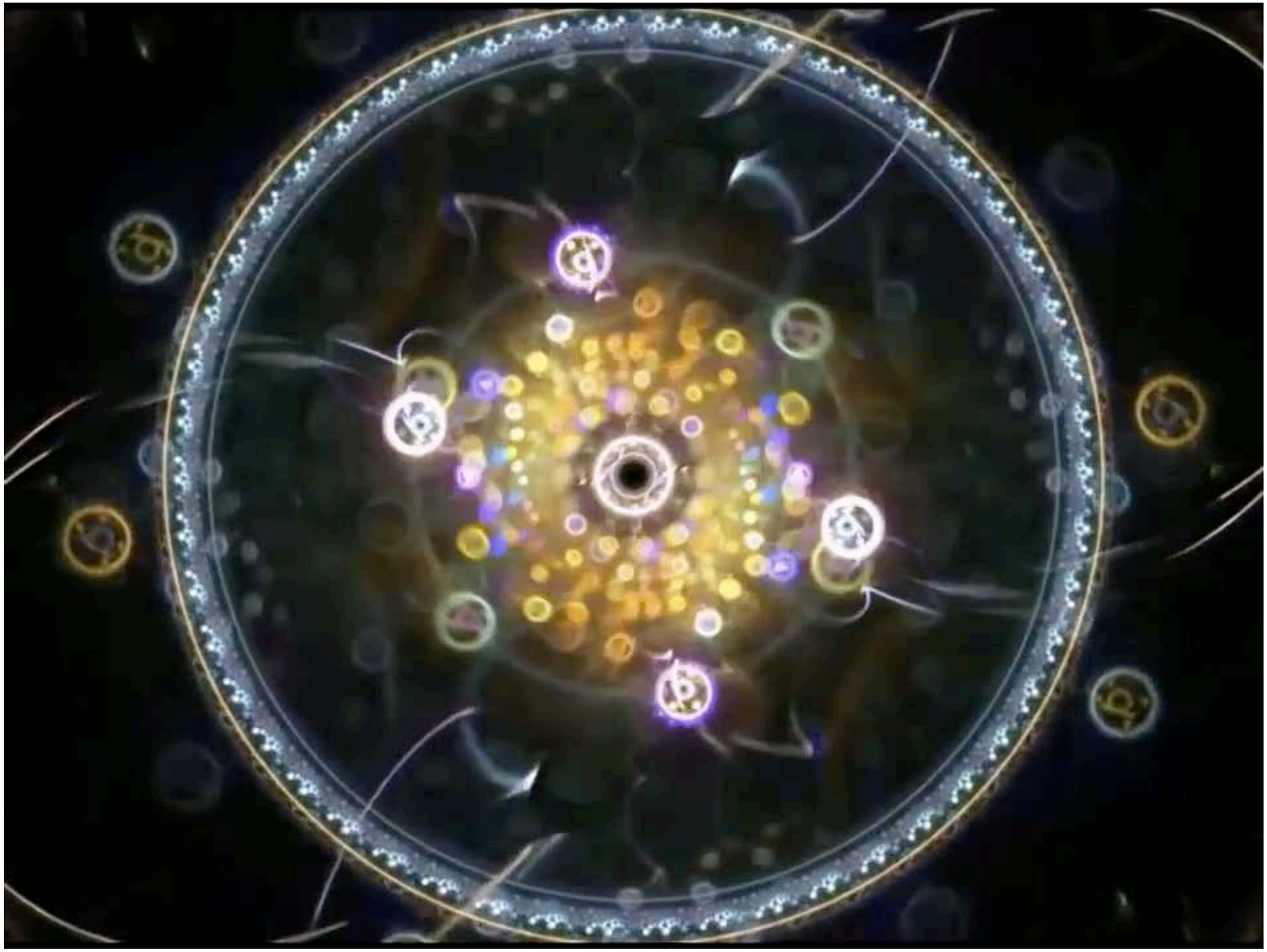
- $(0, 1/2)$ (converges)
- $(0, 1)$ (periodic)
- $(1/3, 1/2)$ (diverges)

Mandelbrot Set - Zooming In



(Colored according to how quickly each point diverges/converges.)

Iterated Function Systems



Scott Draves (CMU alumn) - see http://electricsheep.org

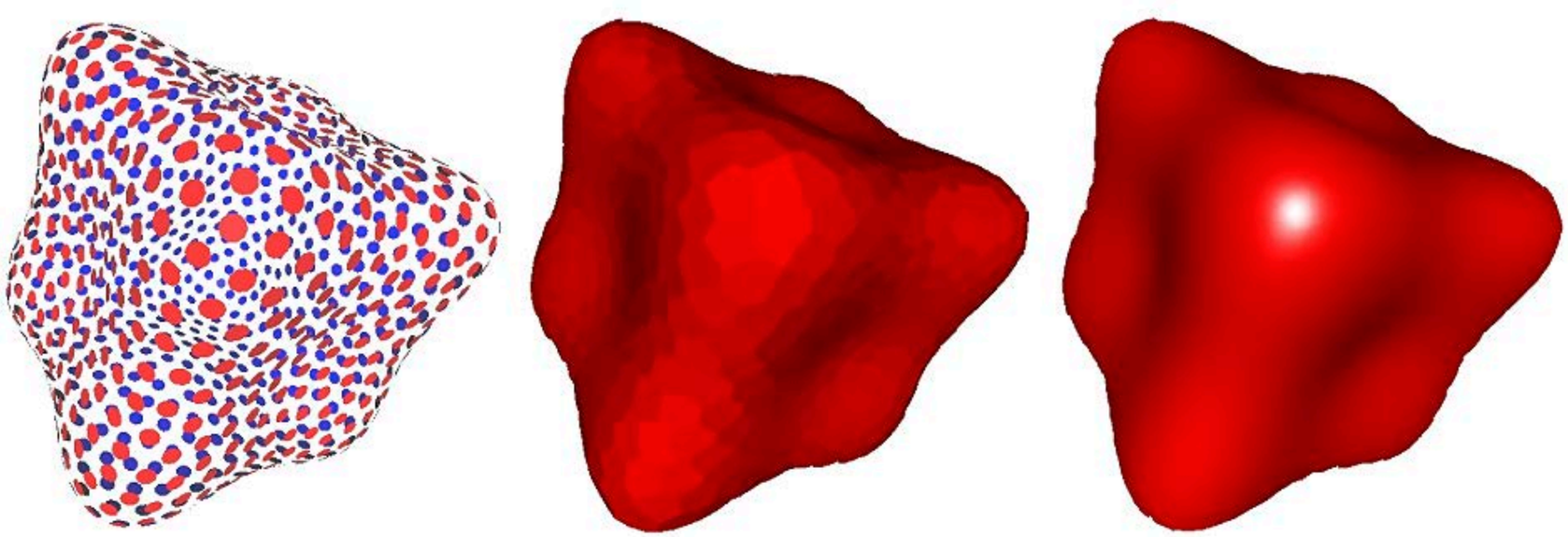
Implicit Representations - Pros & Cons

- **Pros:**
  - **description can be very compact (e.g., a polynomial)**
  - **easy to determine if a point is in our shape (just plug it in!)**
  - **other queries may also be easy (e.g., distance to surface)**
  - **for simple shapes, exact description/no sampling error**
  - **easy to handle changes in topology (e.g., fluid)**
- **Cons:**
  - **expensive to find all points in the shape (e.g., for drawing)**
  - ***very difficult to model complex shapes***

What about explicit representations?

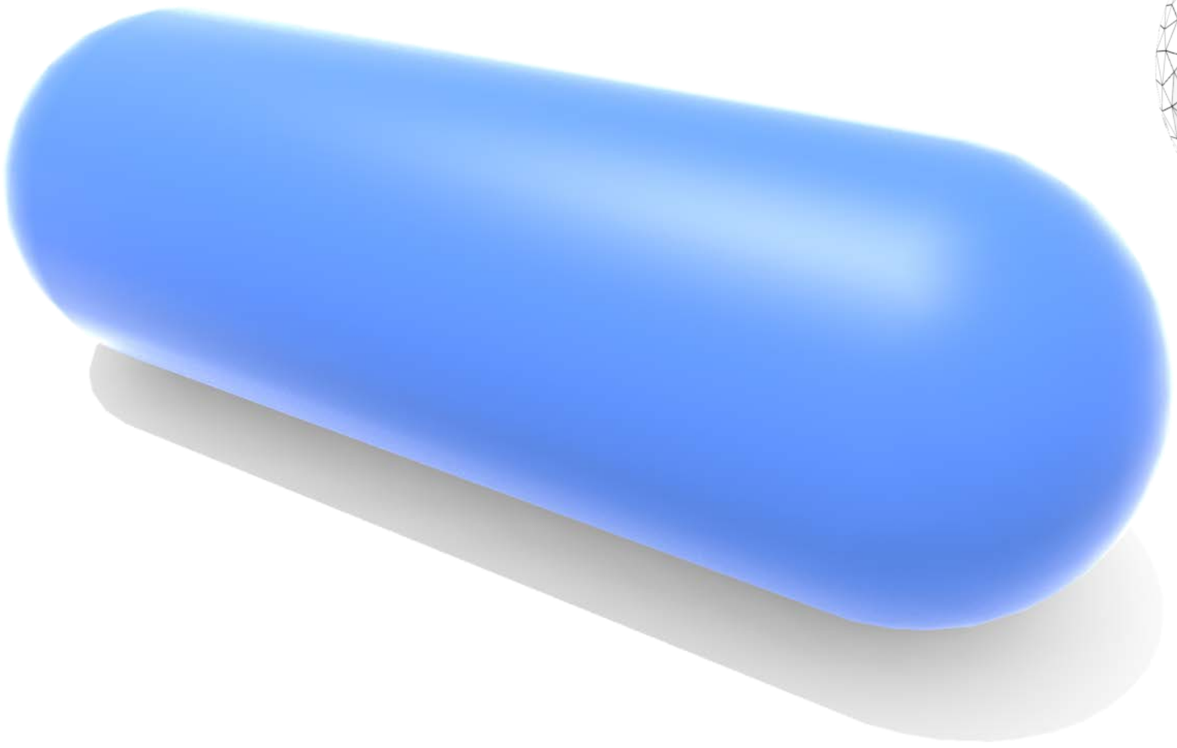
Point Cloud (Explicit)

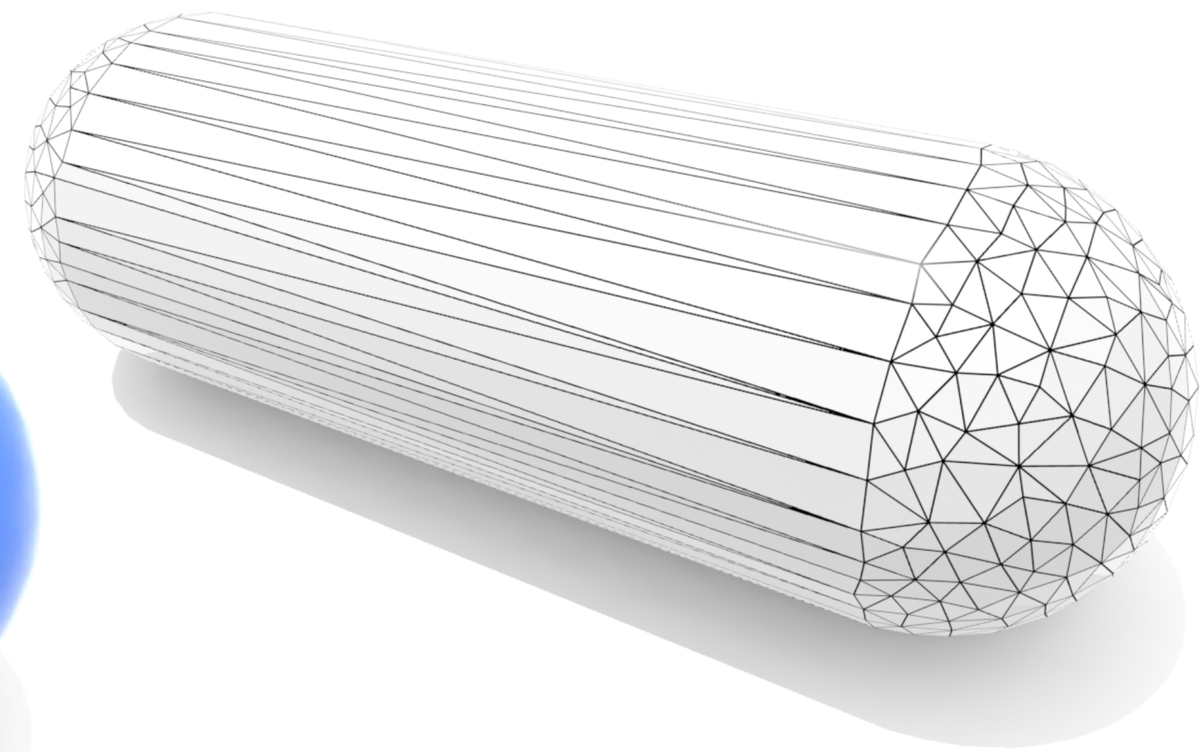
- Easiest representation: list of points (x,y,z)
- Often augmented with *normals*
- Easily represent any kind of geometry
- Easy to draw dense cloud (>>1 point/pixel)
- Hard to interpolate undersampled regions
- Hard to do processing / simulation / …



Polygon Mesh (Explicit)

- Store vertices *and* polygons (most often triangles or quads)
- Easier to do processing/simulation, adaptive sampling
- More complicated data structures
- Irregular neighborhoods





(Much more about polygon meshes in upcoming lectures!)

Triangle Mesh (Explicit)

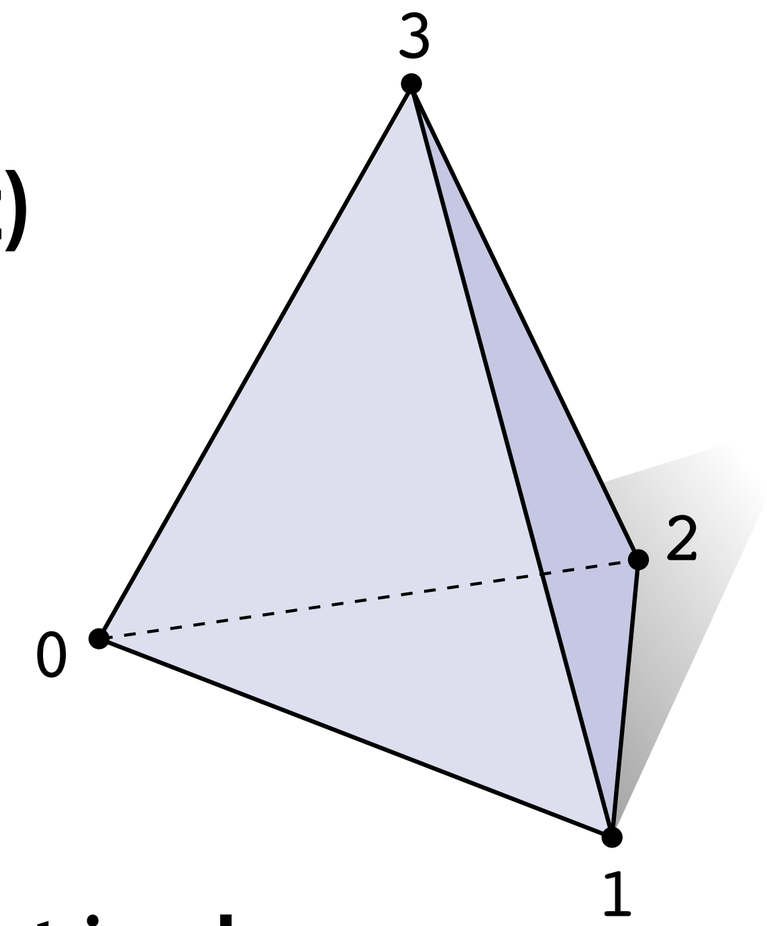
- Store vertices as triples of coordinates (x,y,z)
- Store triangles as triples of indices (i,j,k)
- E.g., tetrahedron:

VERTICES

	x	y	z
0:	-1	-1	-1
1:	1	-1	1
2:	1	1	-1
3:	-1	1	1

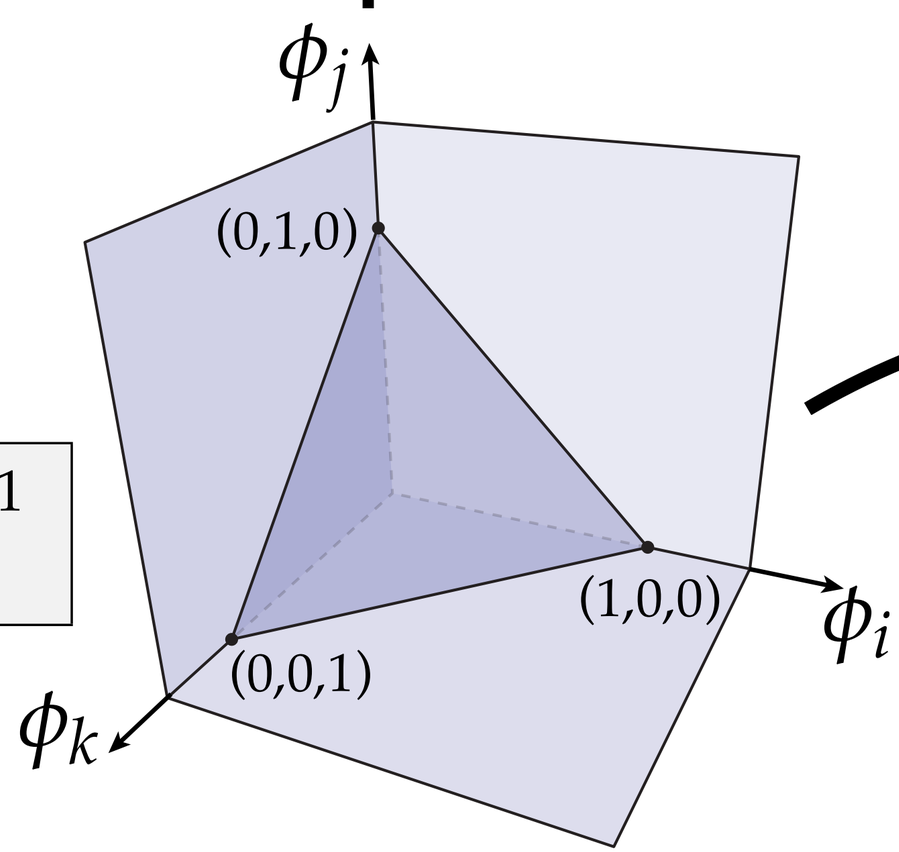
TRIANGLES

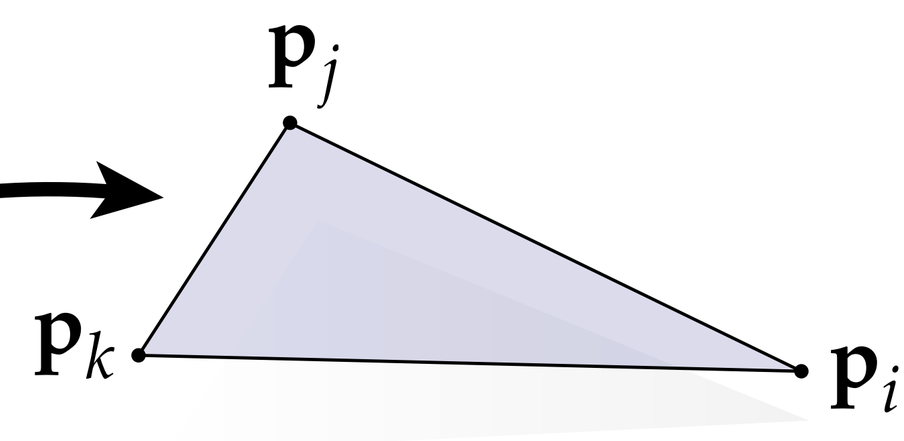
i	j	k
0	2	1
0	3	2
3	0	1
3	1	2



- Use barycentric interpolation to define points inside triangles:

$$\phi_i + \phi_j + \phi_k = 1$$
$$\phi_i, \phi_j, \phi_k > 0$$





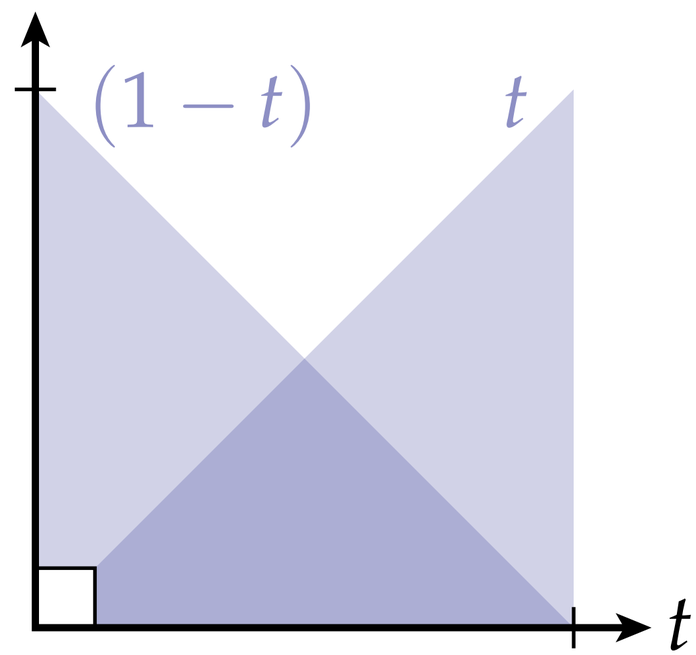
$$p = \phi_i \mathbf{p}_i + \phi_j \mathbf{p}_j + \phi_k \mathbf{p}_k$$

Recall: Linear Interpolation (1D)

- Interpolate values using *linear interpolation*; in 1D:

$$\hat{f}(t) = (1-t)f_i + tf_j$$

- Can think of this as a linear combination of two functions:



- Why limit ourselves to <u>linear</u> basis functions?
- Can we get more interesting geometry with other bases?

Bernstein Basis

- Linear interpolation essentially uses 1st-order polynomials
- Provide more flexibility by using higher-order polynomials
- Instead of usual basis (1, x, x^2, x^3, ...), use Bernstein basis:

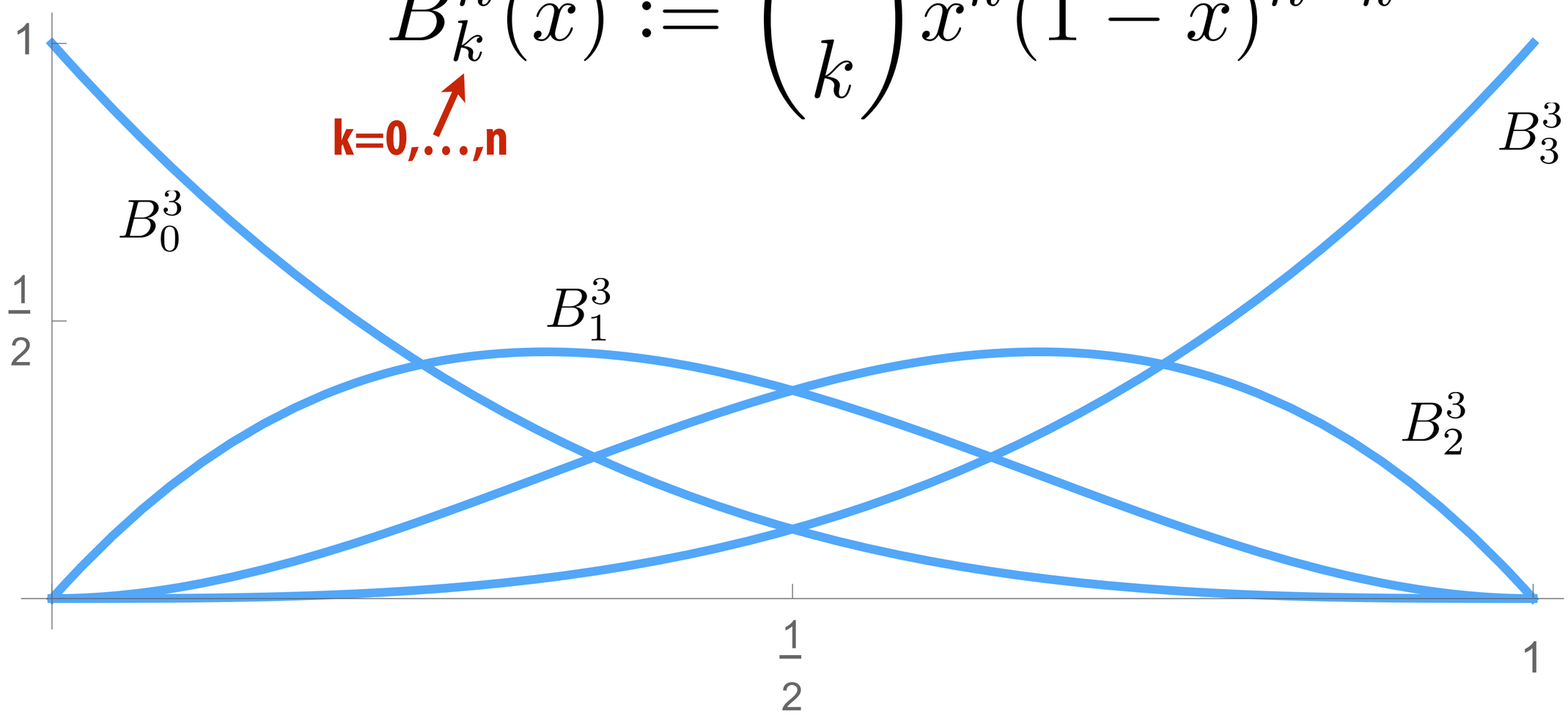
degree

0≤x≤1

"n choose k"

$$B_k^n(x) := \binom{n}{k} x^k (1-x)^{n-k}$$

k=0,...,n



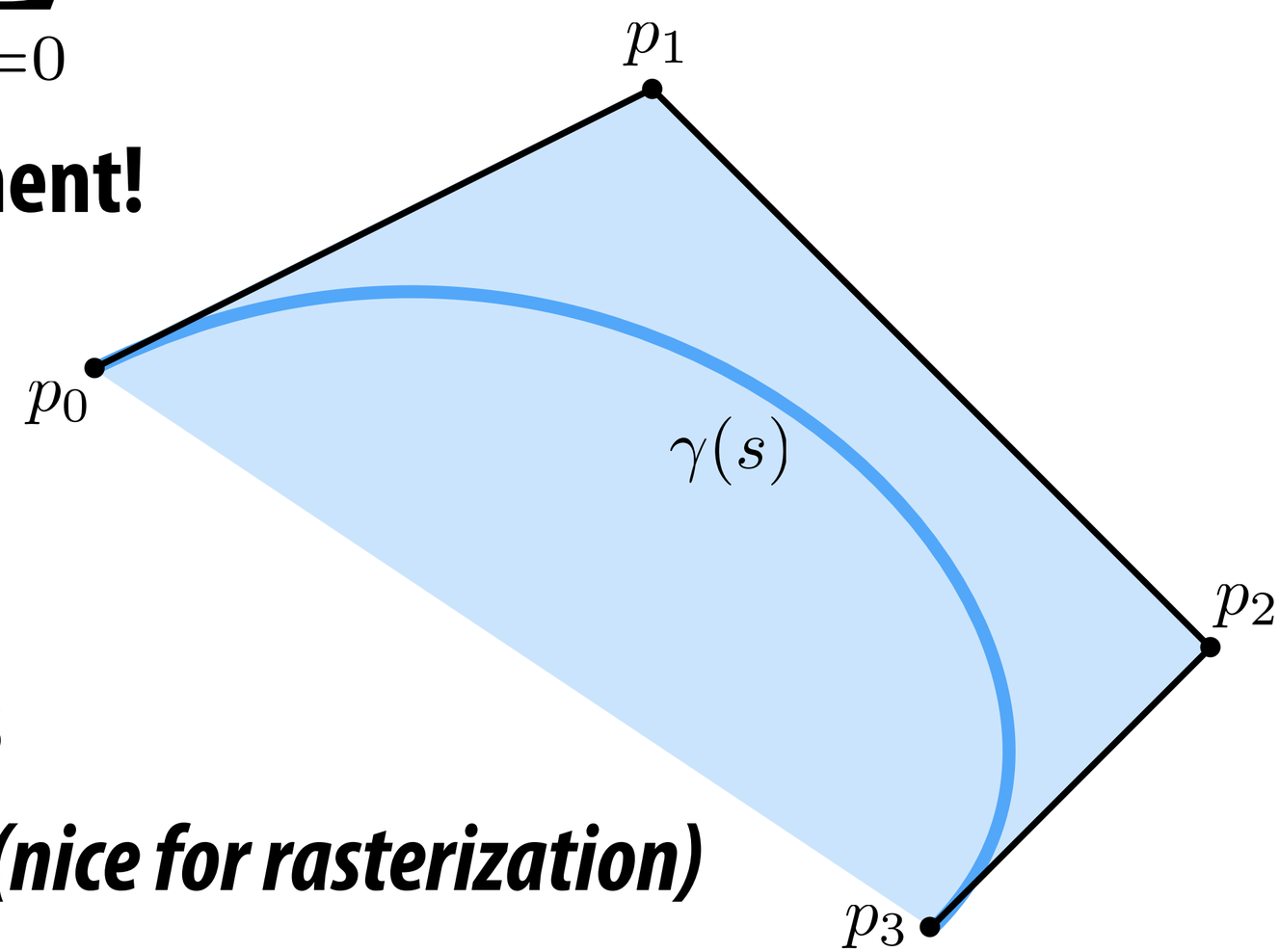
Bézier Curves (Explicit)

- A Bézier curve is a curve expressed in the Bernstein basis:

$$\gamma(s) := \sum_{k=0}^{n} B_{n,k}(s) p_k$$

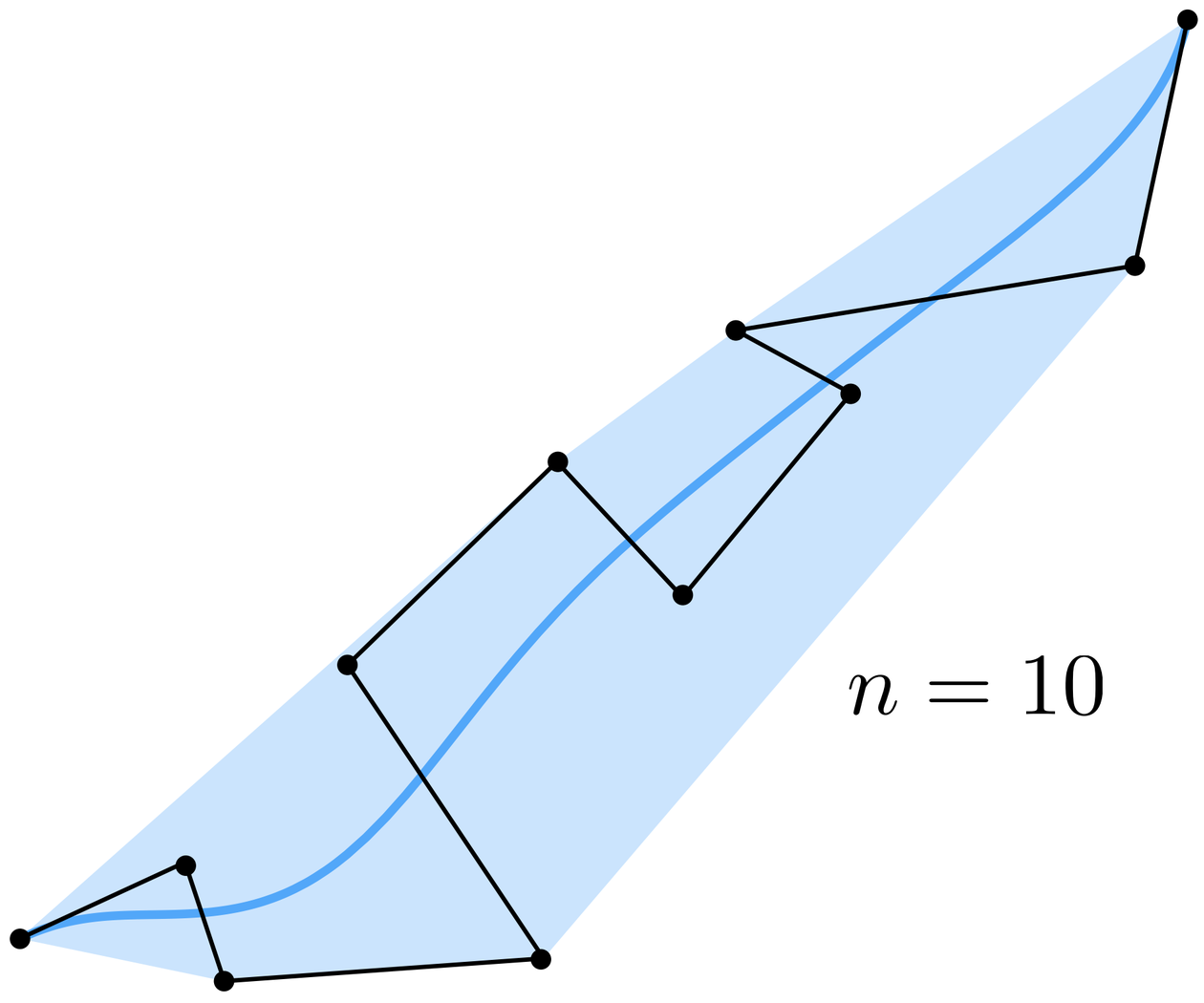
control points

- For n=1, just get a line segment!
- For n=3, get "cubic Bézier":
- Important features:
  1. interpolates endpoints
  2. tangent to end segments
  3. contained in convex hull *(nice for rasterization)*



Just keep going…?

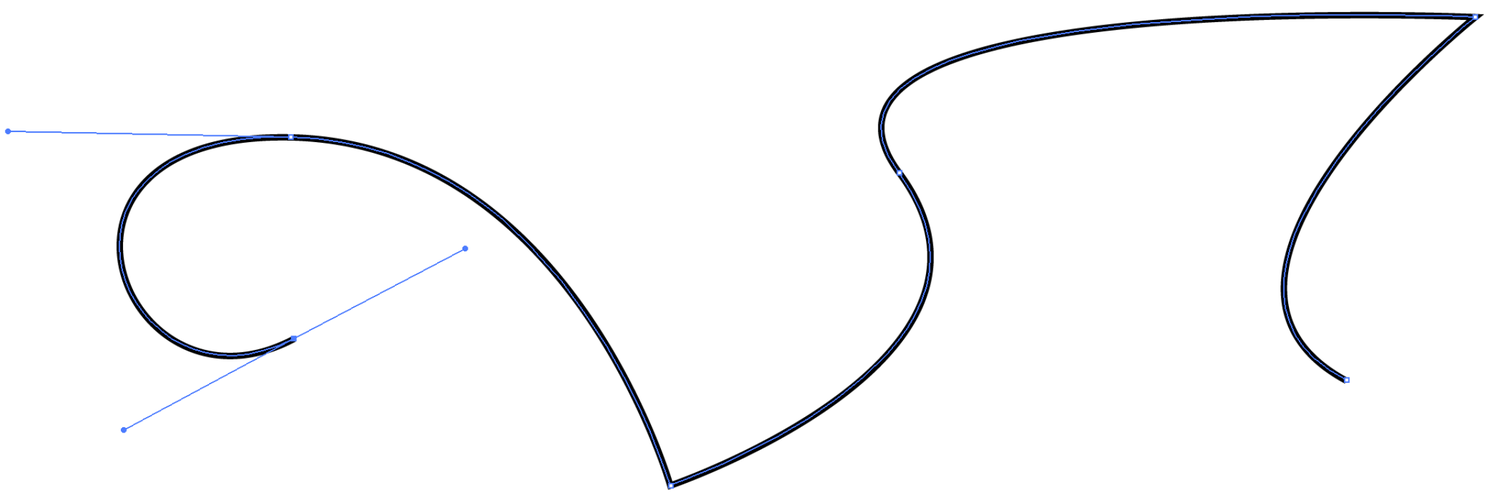
- What if we want an even more interesting curve?
- High-degree Bernstein polynomials don't interpolate well:

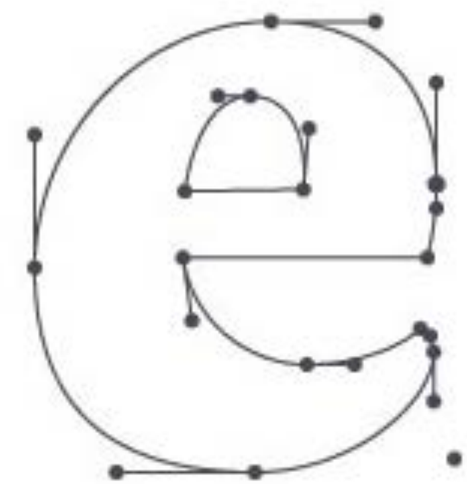


Very hard to control!

Piecewise Bézier Curves (Explicit)

- Alternative idea: piece together many Bézier curves
- Widely-used technique (Illustrator, fonts, SVG, etc.)





- Formally, piecewise Bézier curve:

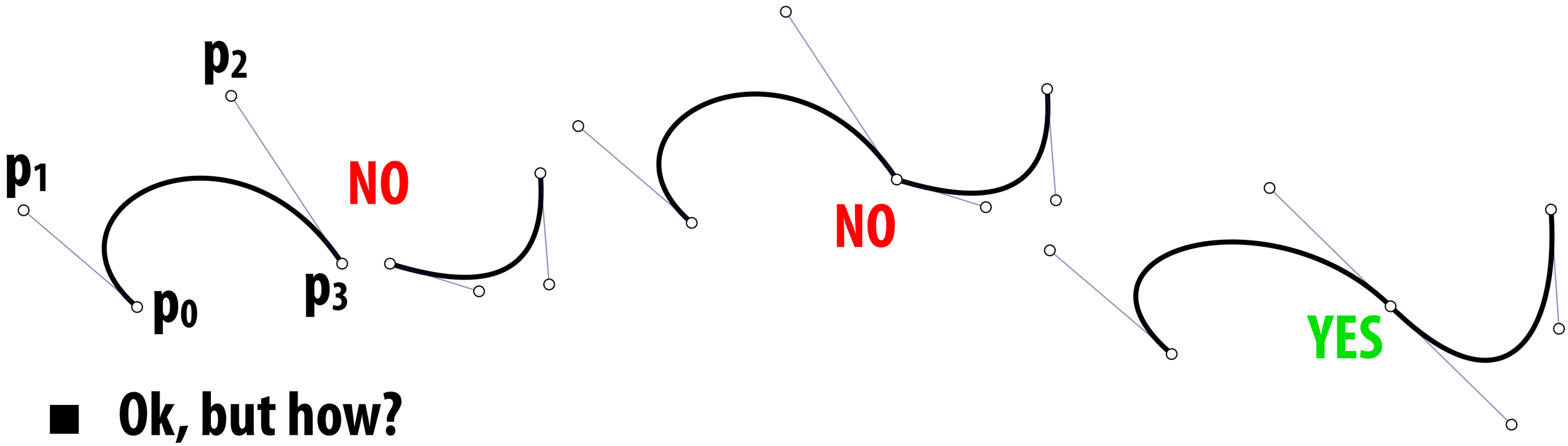
piecewise Bézier

$$\gamma(u) := \gamma_i\left(\frac{u - u_i}{u_{i+1} - u_i}\right), \qquad u_i \leq u < u_{i+1}$$

single Bézier

Bézier Curves — tangent continuity

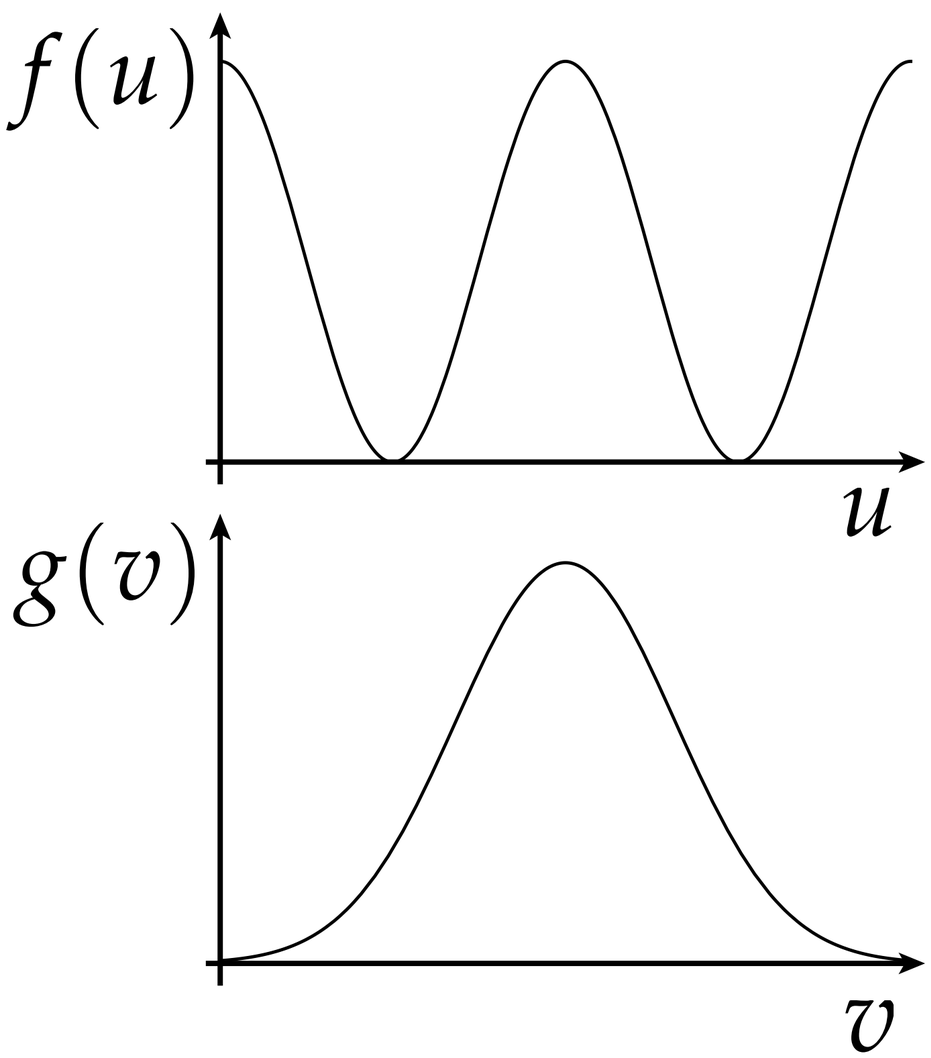
- To get "seamless" curves, need *points* <u>and</u> *tangents* to line up:

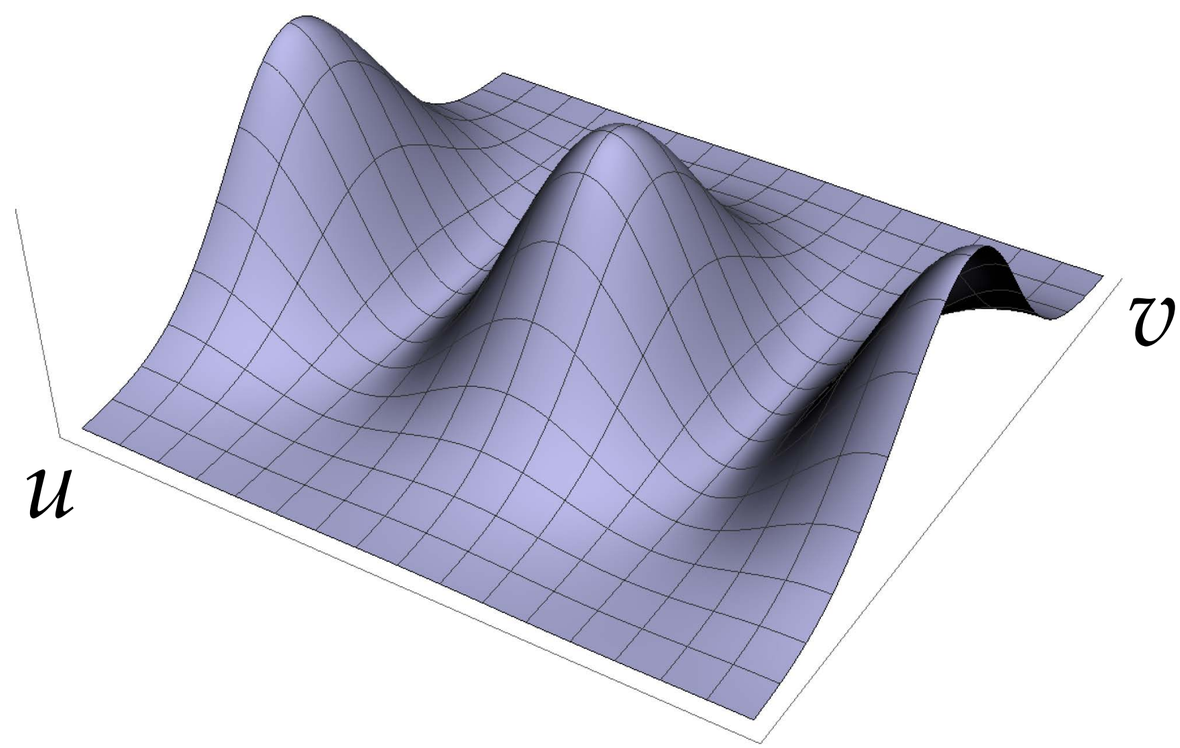


- Ok, but how?
- Each curve is cubic: $u^3p_0 + 3u^2(1-u)p_1 + 3u(1-u)^2p_2 + (1-u)^3p_3$
- Want endpoints of each segment to meet
- Want tangents at endpoints to meet
- Q: How many constraints vs. degrees of freedom?
- Q: Could you do this with *quadratic* Bézier? *Linear* Bézier?

Tensor Product

- Can use a pair of curves to get a surface
- Value at any point (u,v) given by product of a curve f at u and a curve g at v (sometimes called the *"tensor product"*):

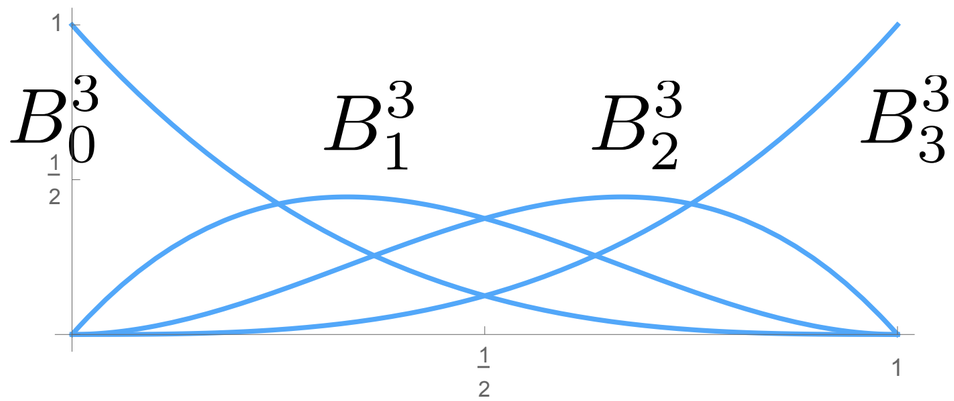


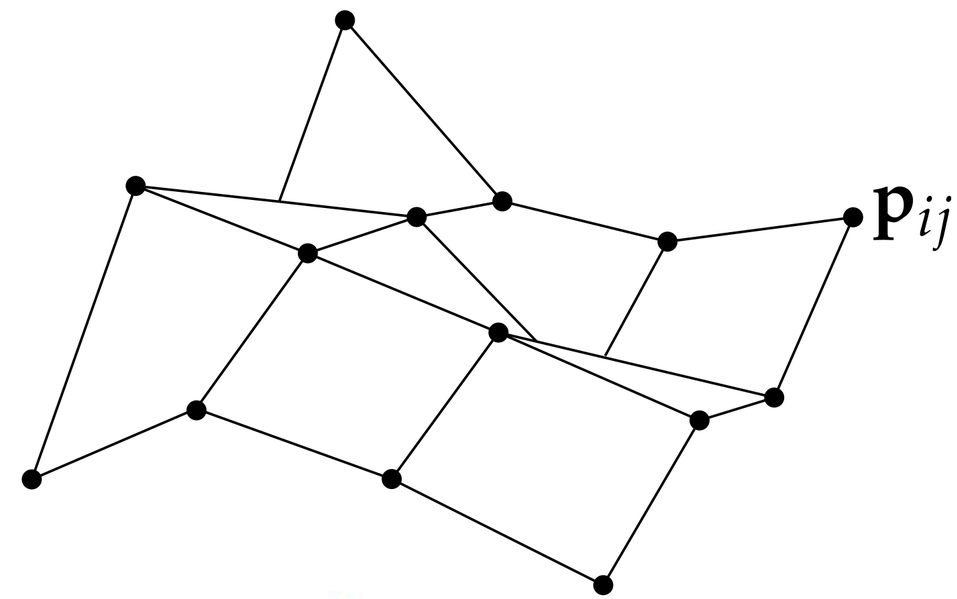


$$(f \otimes g)(u,v) := f(u)g(v)$$

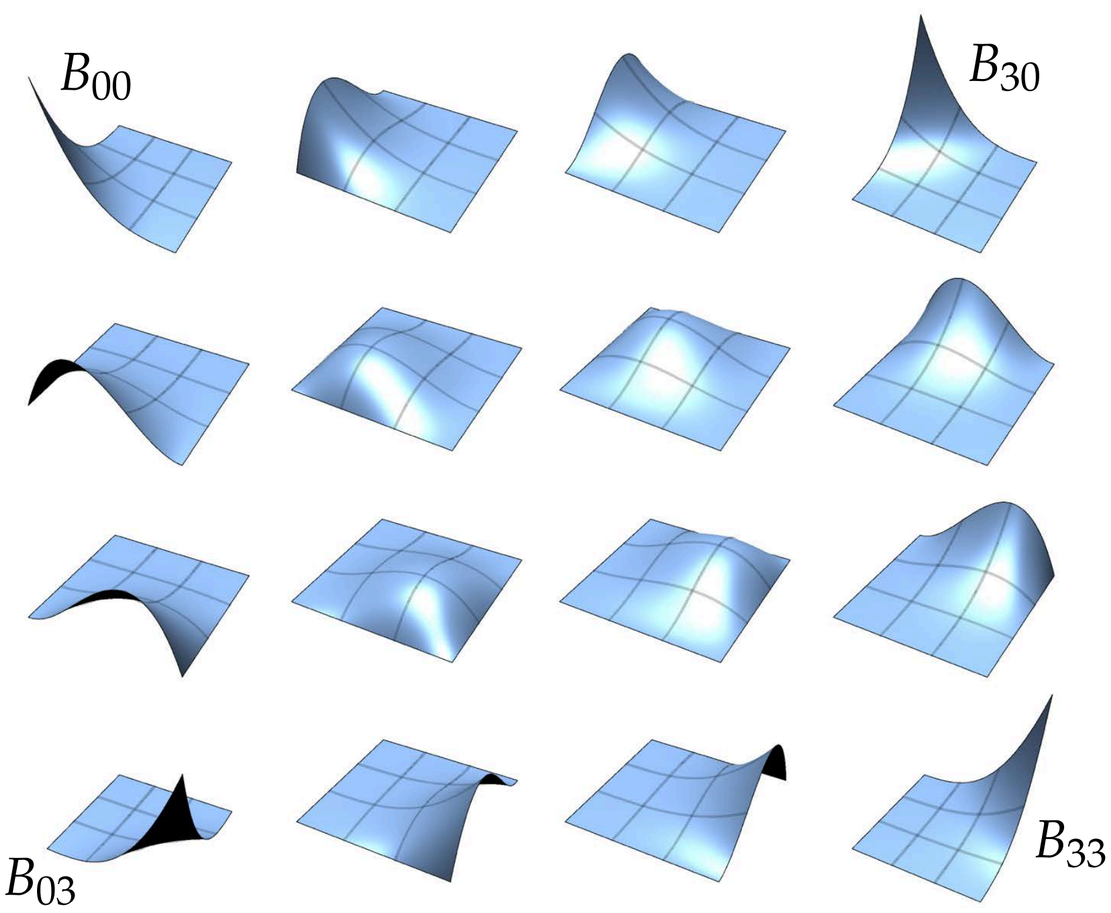
Bézier Patches

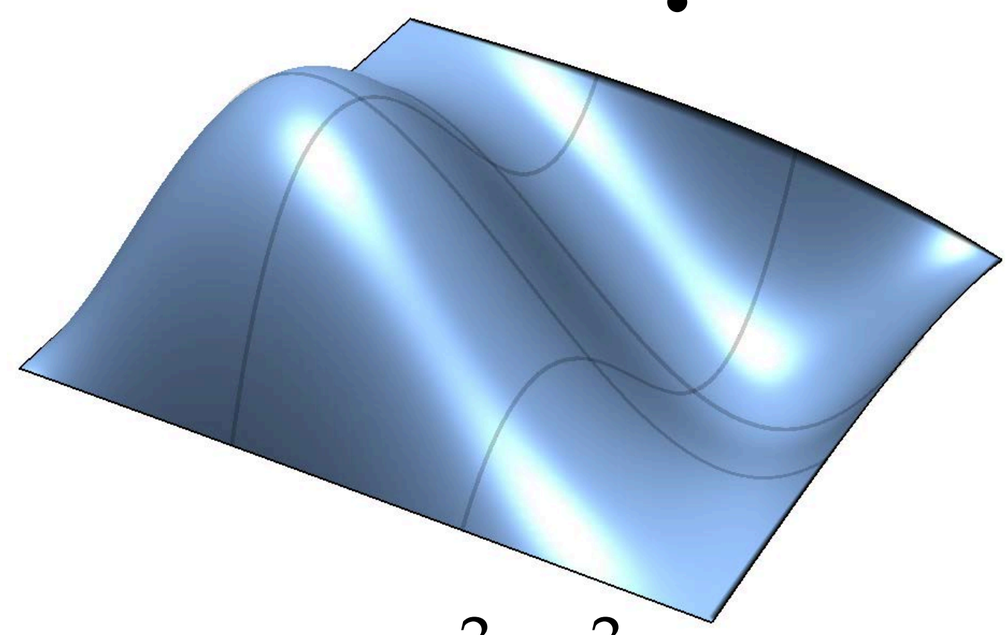
- ***Bézier patch*** **is sum of (tensor) products of Bernstein bases**





$$B_{i,j}^3(u,v) := B_i^3(u)B_j^3(v)$$

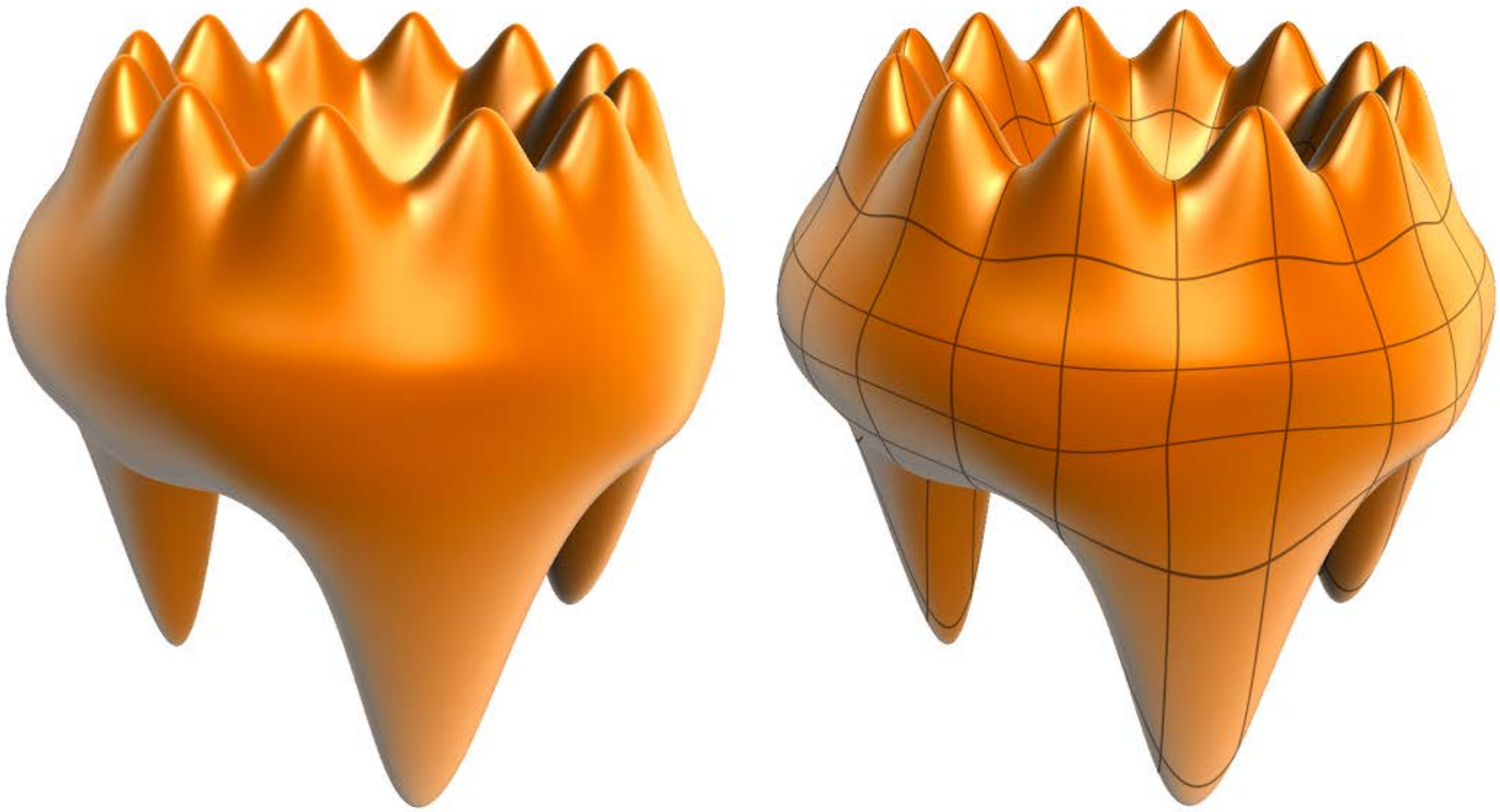




$$S(u,v) := \sum_{i=0}^{3}\sum_{j=0}^{3} B_{i,j}^3(u,v)\mathbf{p}_{ij}$$

Bézier Surface

- Just as we connected Bézier *curves*, can connect Bézier *patches* to get a surface:

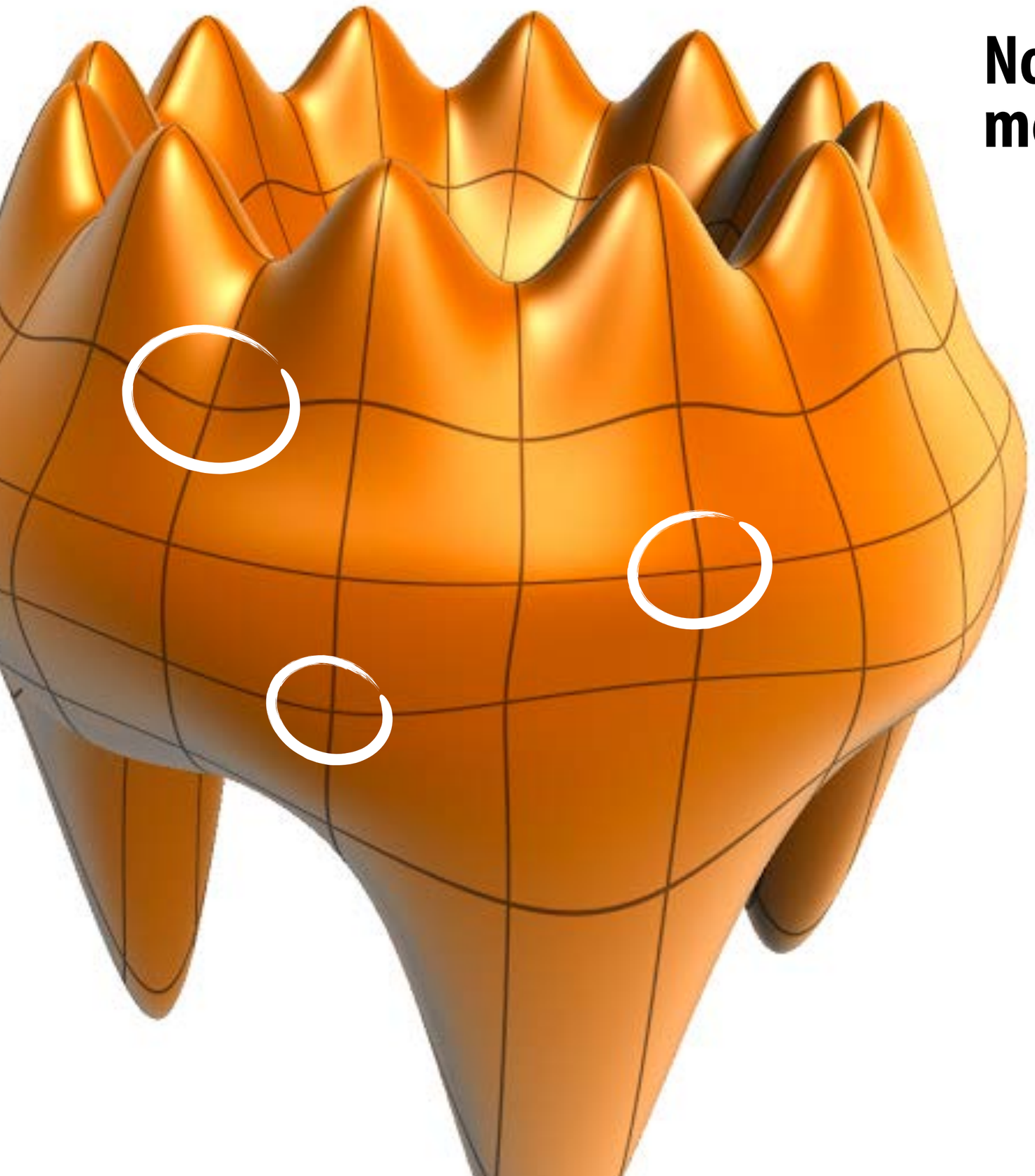


- *Very* easy to draw: just dice each patch into regular (u,v) grid!

Q: Can we always get tangent continuity?
(Think: how many constraints? How many degrees of freedom?)

Notice anything fishy about the last picture?

Bézier Patches are Too Simple



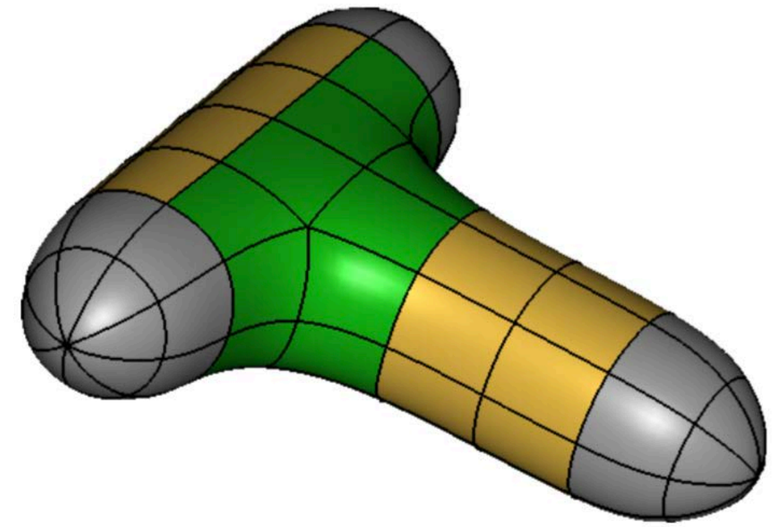
Notice that exactly four patches meet around *every* vertex!

In practice, far too constrained.

To make interesting shapes (with good continuity), we need patches that allow more interesting connectivity...

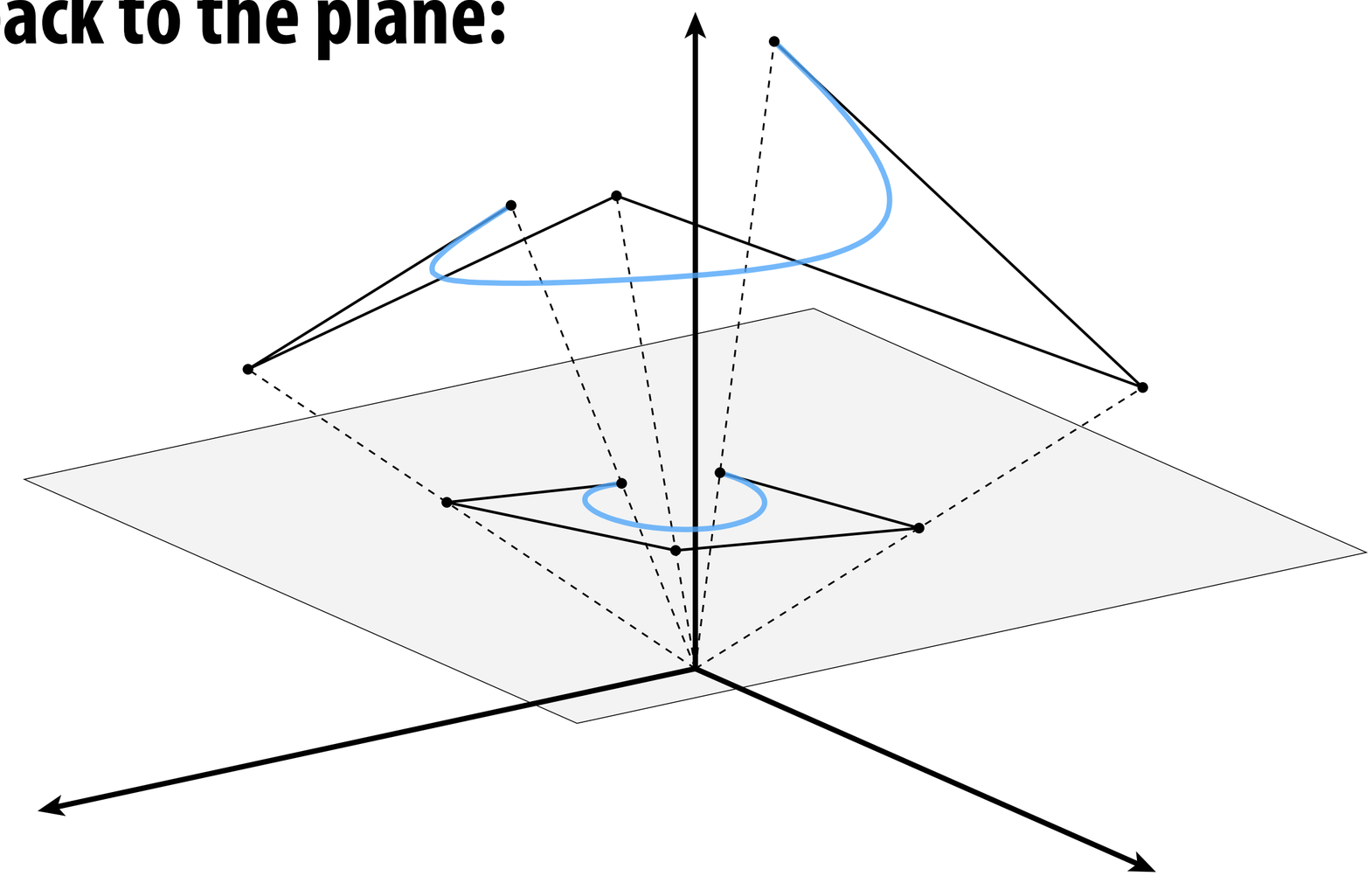
Spline patch schemes

- There are *many* alternatives!
- NURBS, Gregory, Pm, polar…
- Tradeoffs:
  - degrees of freedom
  - continuity
  - difficulty of editing
  - cost of evaluation
  - generality
  - …
- As usual: ***pick the right tool for the job!***



Rational B-Splines (Explicit)

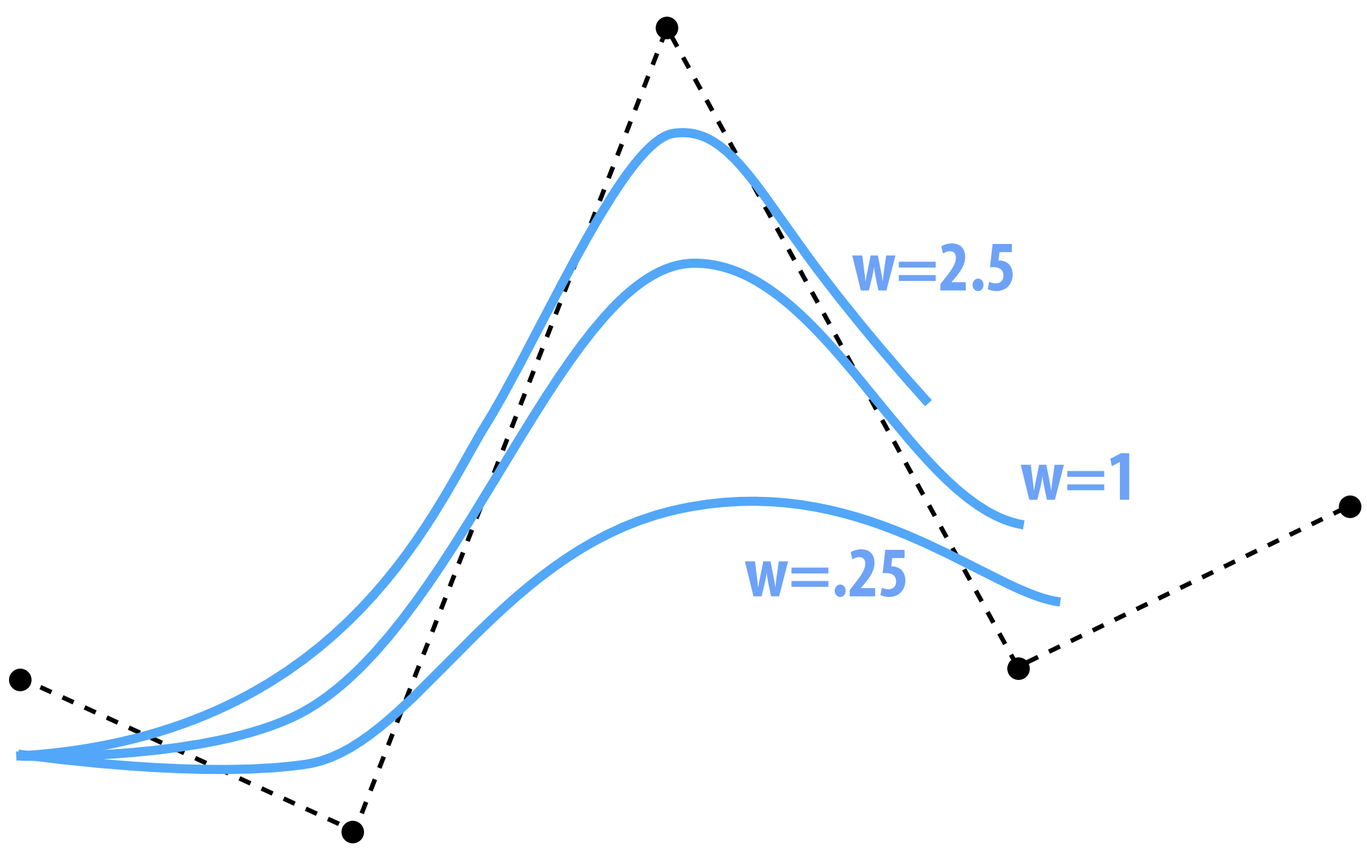
- Bézier can't exactly represent *conics*—not even the circle!
- Solution: interpolate in homogeneous coordinates, then project back to the plane:



Result is called a *rational* B-spline.

NURBS (Explicit)

- (N)on-(U)niform (R)ational (B)-(S)pline
  - knots at arbitrary locations (non-uniform)
  - expressed in homogeneous coordinates (rational)
  - piecewise polynomial curve (B-Spline)
- Homogeneous coordinate w controls "strength" of a vertex:

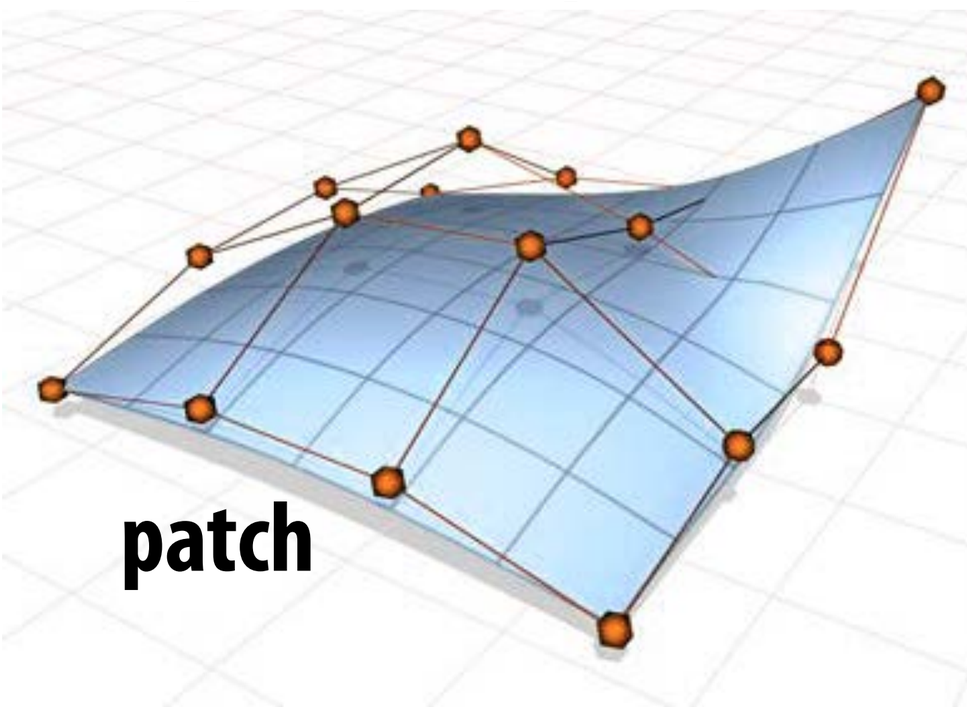


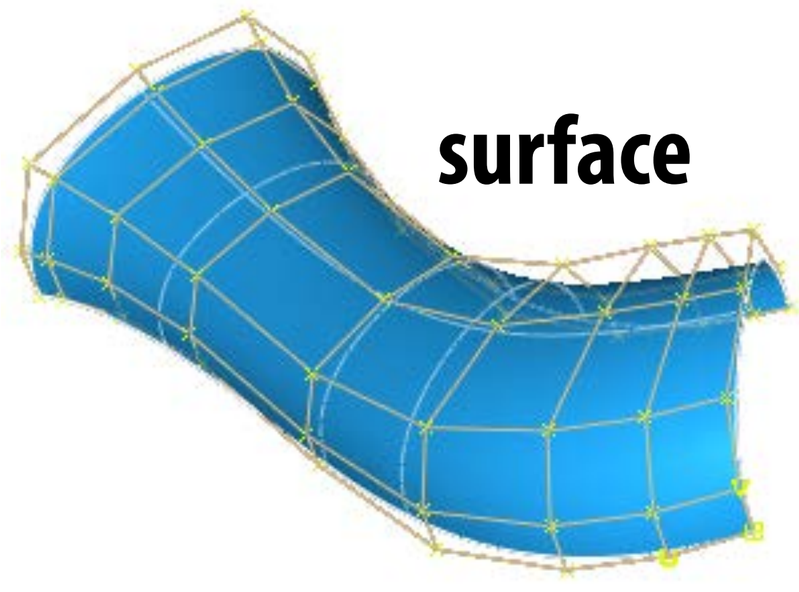
NURBS Surface (Explicit)

- How do we go from curves to surfaces?
- Use *tensor product* of NURBS curves to get a patch:

$$S(u,v) := N_i(u)N_j(v)p_{ij}$$

- Multiple NURBS patches form a surface

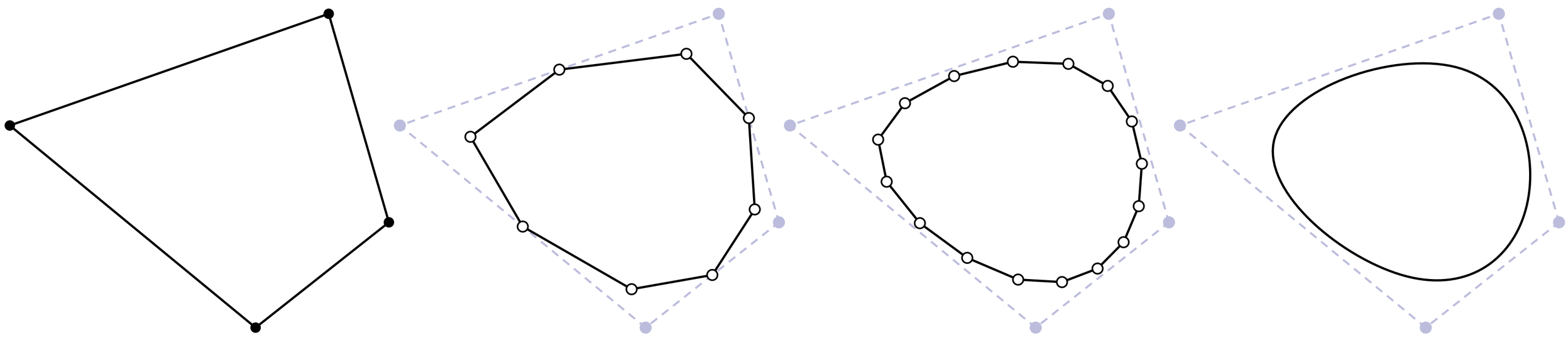




- Pros: easy to evaluate, exact conics, high degree of continuity
- Cons: Hard to piece together patches / hard to edit (many DOFs)

Subdivision

- Alternative starting point for curves/surfaces: *subdivision*
- Start with "control curve"
- Repeatedly split, take weighted average to get new positions
- For careful choice of averaging rule, approaches nice limit curve
  - *Often exact same curve as well-known spline schemes!*

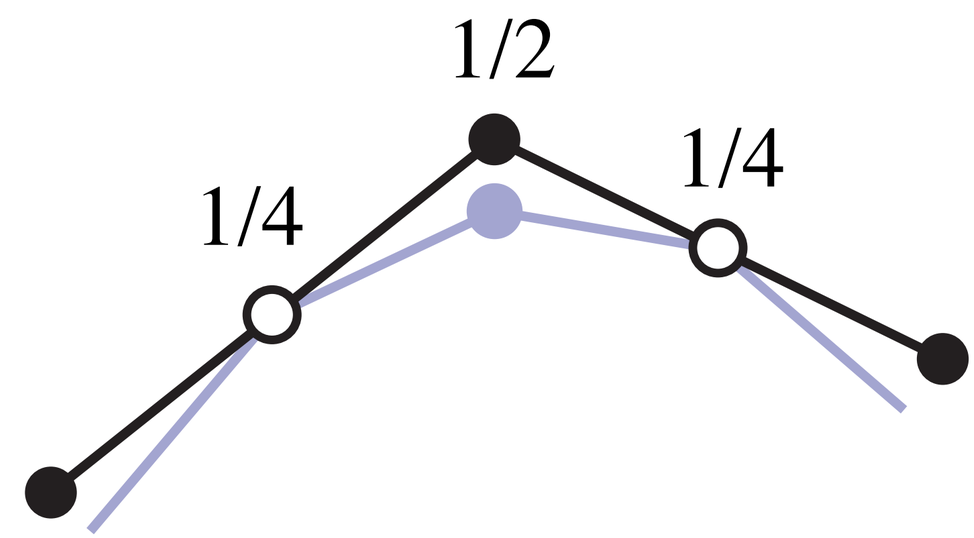


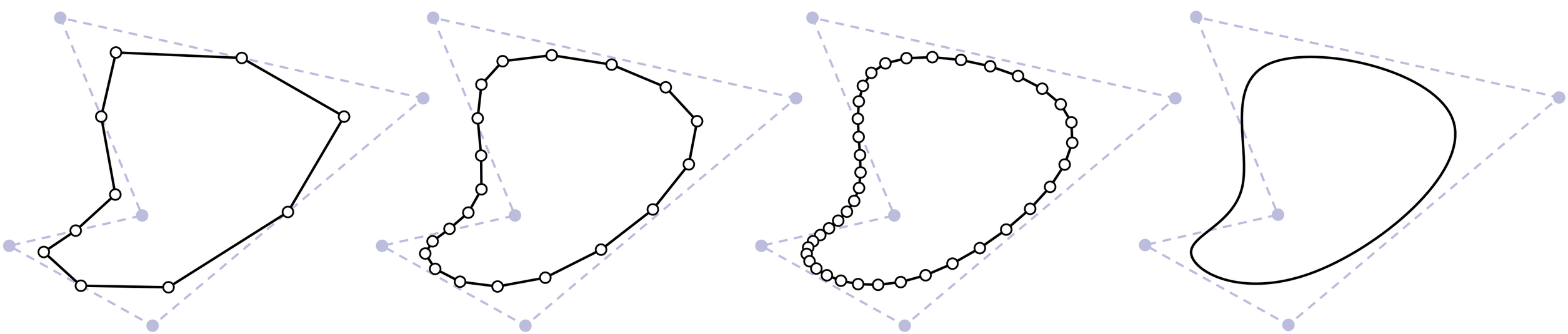
Q: Is subdivision an explicit or implicit representation?

Subdivision—Example

- **One possible scheme: *Lane-Riesenfeld***
  - **insert midpoint of each edge**
  - **use row k of Pascal's triangle (normalized to 1) as weights for neighbors**
  - **e.g., $k = 2$, get weights** (1/4,1/2,1/4)
  - **limit is B-spline of degree $k + 1$**

$k = 0$:				1			
$k = 1$:			1		1		
$k = 2$:		1		2		1	
$k = 3$:	1		3		3		1

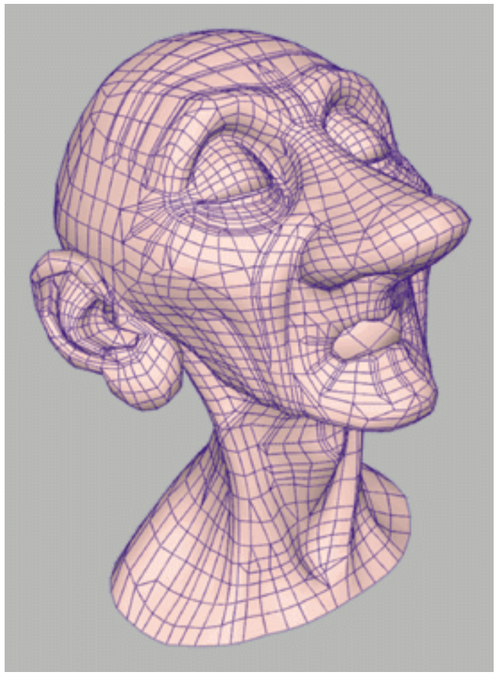


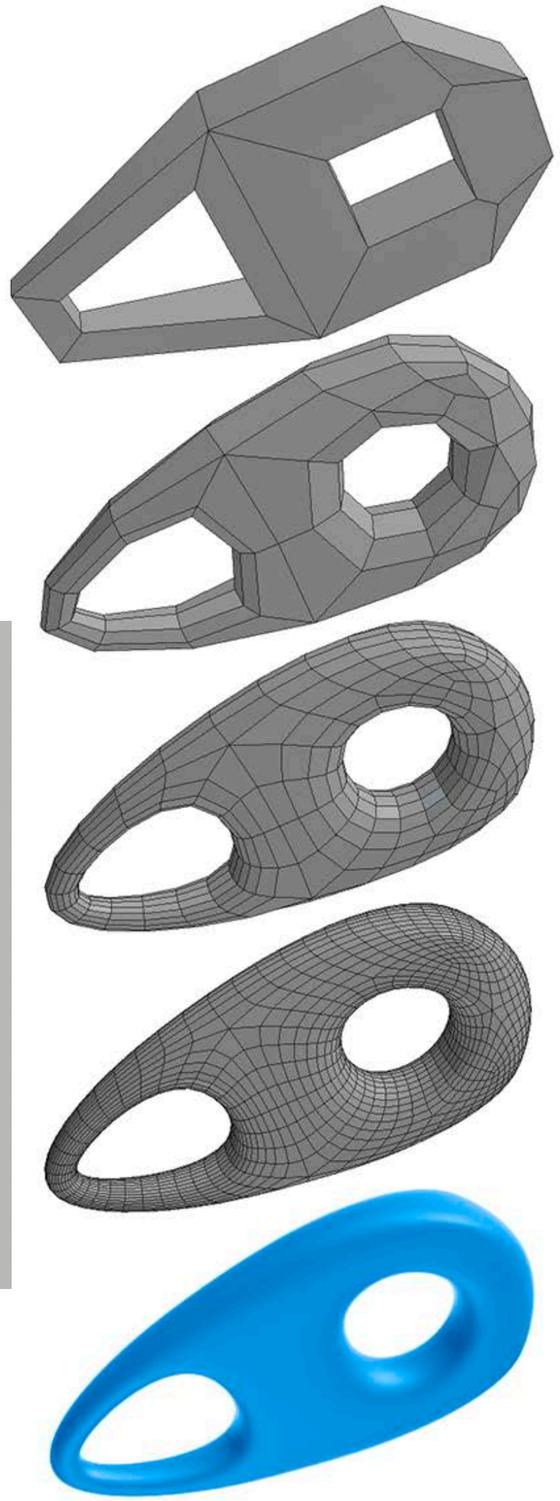


iteration 1 **iteration 2** **iteration 3** **limit curve**

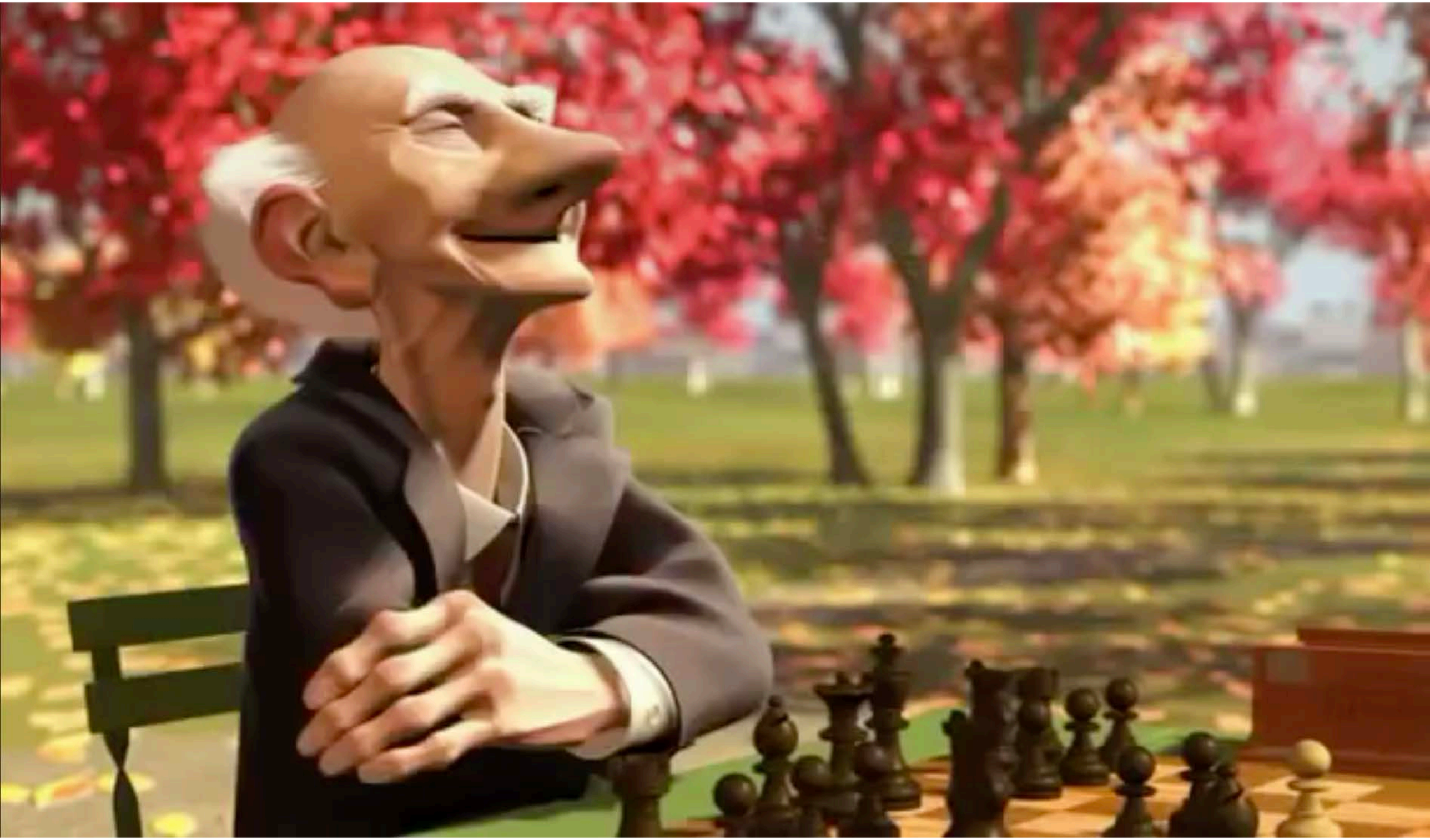
Subdivision Surfaces (Explicit)

- Start with coarse polygon mesh ("control cage")
- Subdivide each element
- Update vertices via local averaging
- Many possible rules:
  - *Catmull-Clark* (quads)
  - *Loop* (triangles)
  - ...
- Common issues:
  - interpolating or approximating?
  - continuity at vertices?
- Easier than splines for modeling; harder to evaluate pointwise
- Widely used in practice (2019 Academy Awards!)



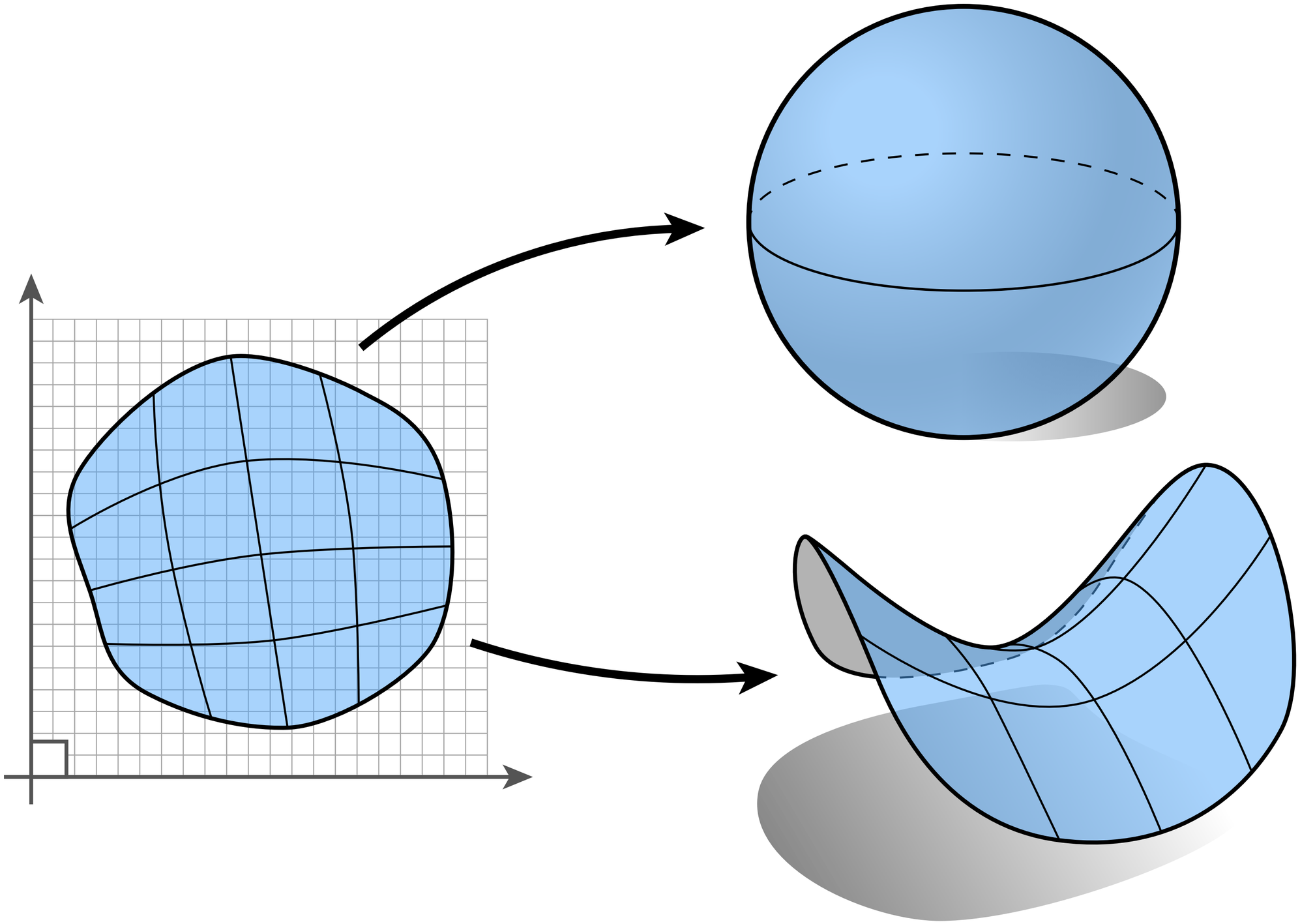


Subdivision in Action (Pixar's "Geri's Game")



see: de Rose et al, "Subdivision Surfaces in Character Animation"

Next time: Curves, Surfaces, & Meshes



Meshes and Manifolds

Computer Graphics
CMU 15-462/15-662

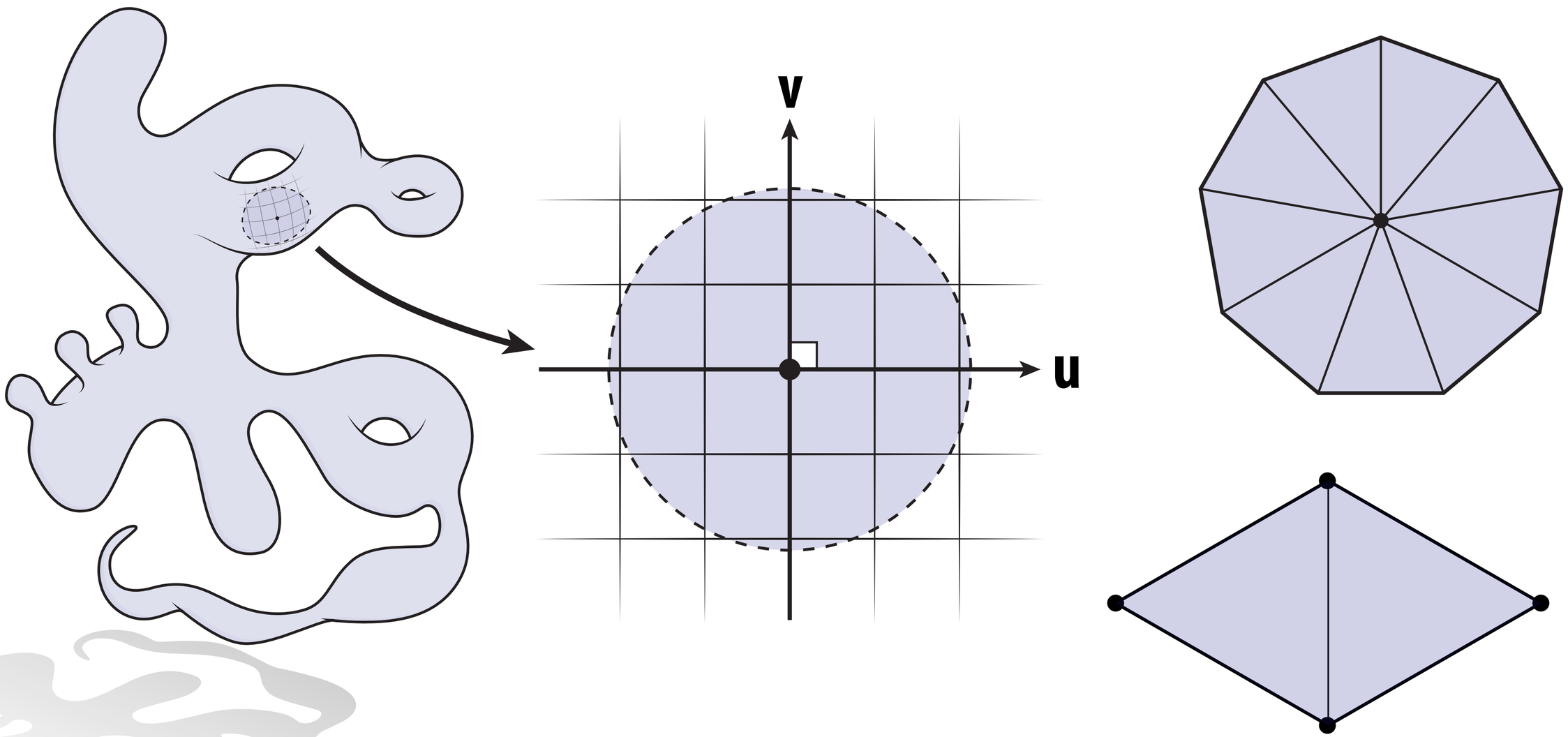
Last time: overview of geometry

- 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
- Today:
  - what is a surface, anyway?
  - nuts & bolts of polygon meshes
  - geometry processing / resampling

Geometry

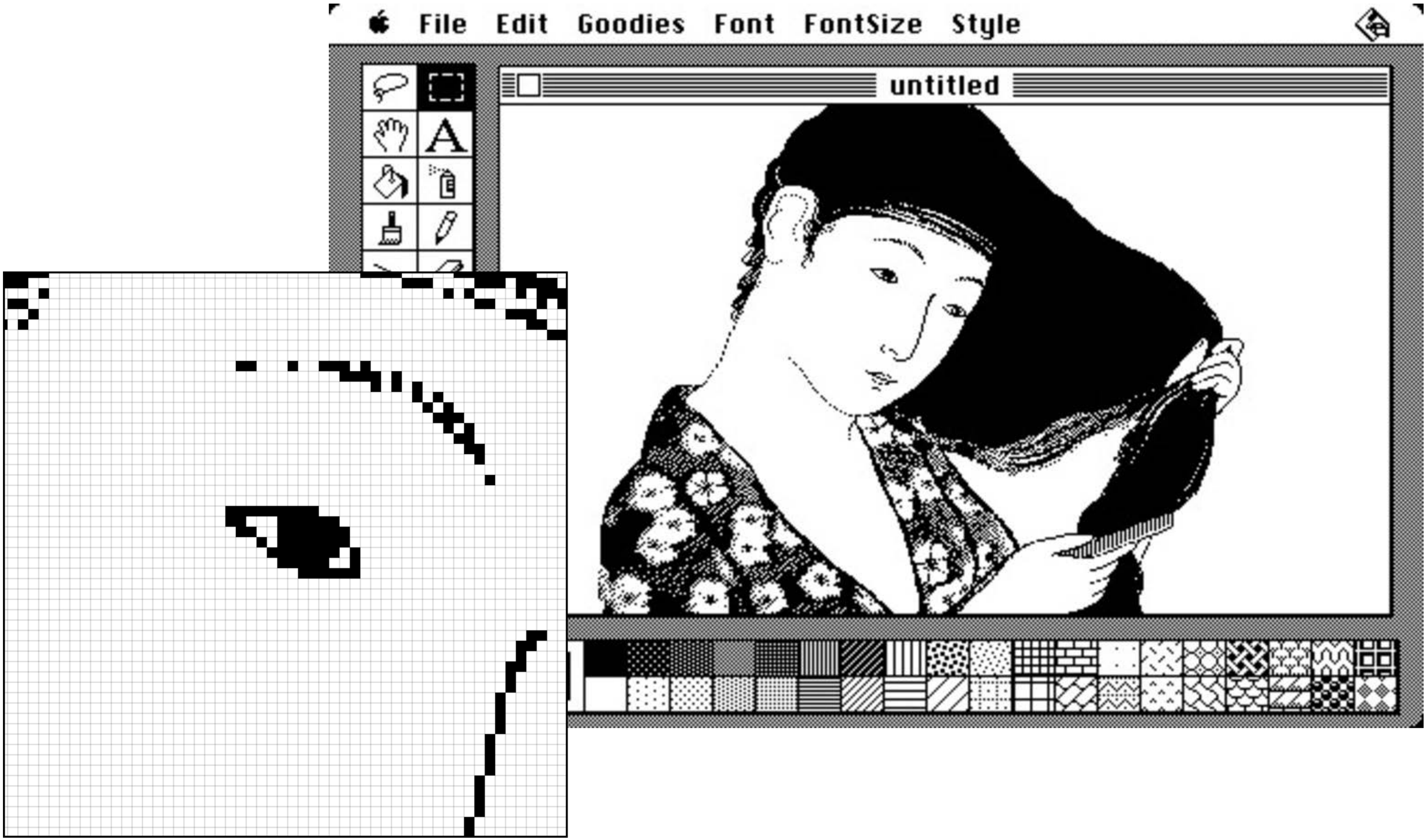
Manifold Assumption

- Today we're going to introduce the idea of *manifold* geometry
- Can be hard to understand motivation at first!
- So first, let's revisit a more familiar example...



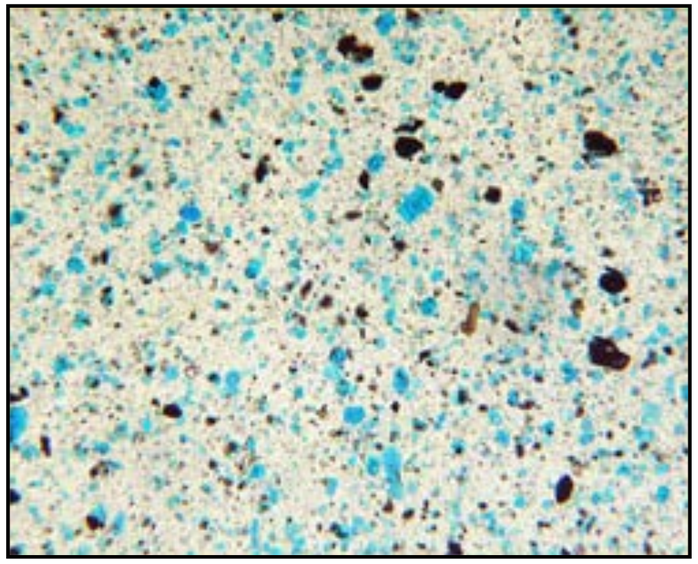
Bitmap Images, Revisited

To encode images, we used a *regular grid* of pixels:



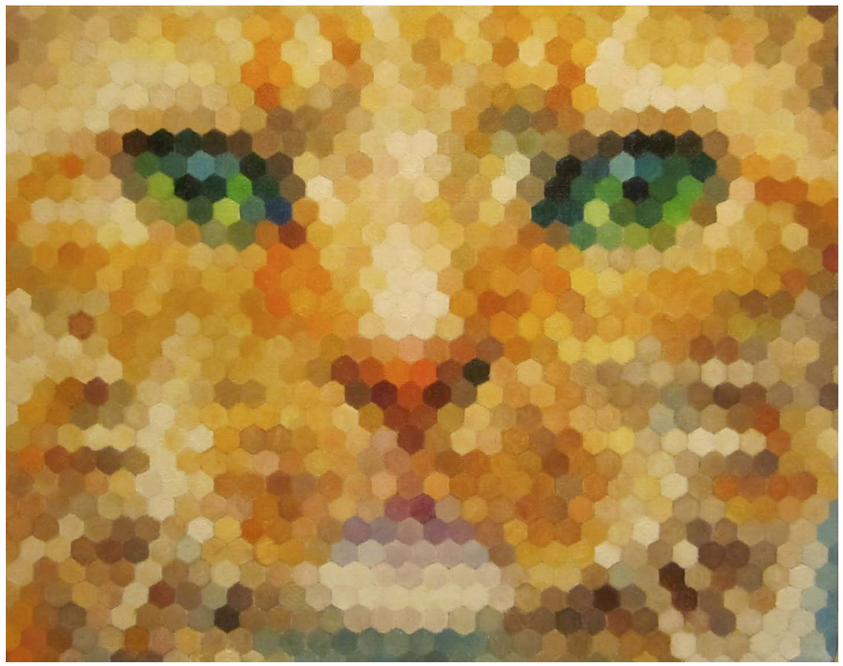
But images are not fundamentally made of little squares:

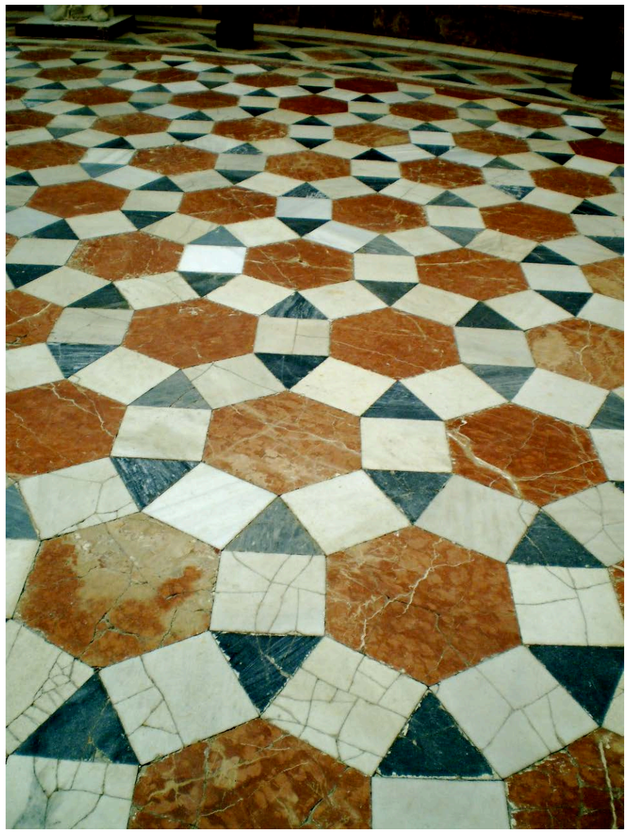
Goyō Hashiguchi, *Kamisuki* (ca 1920)



photomicrograph of paint

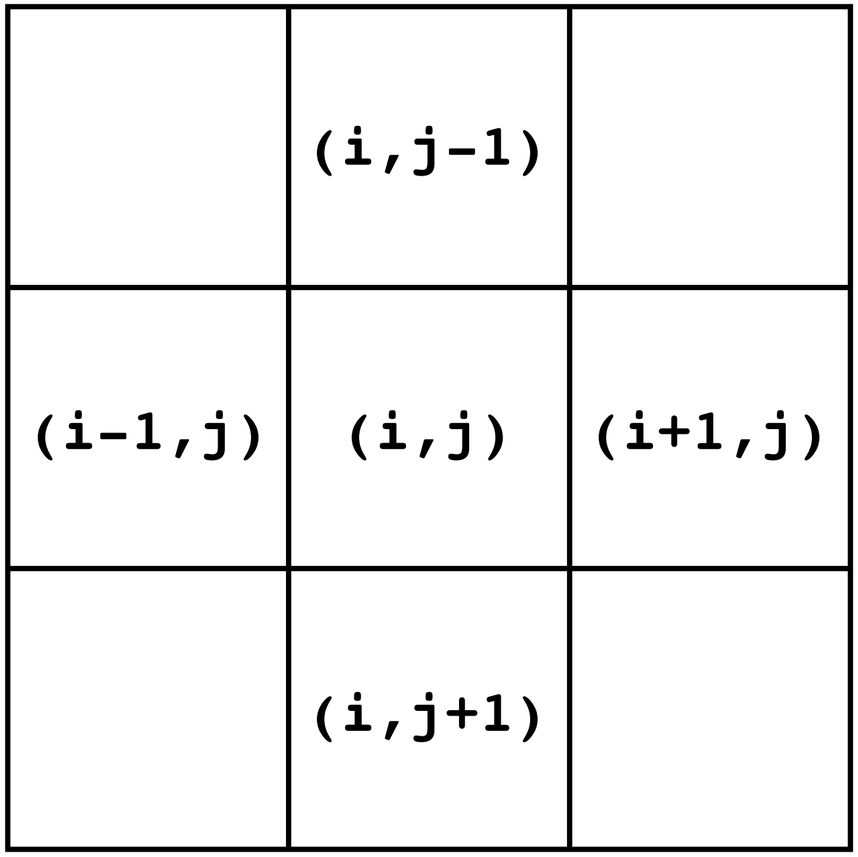
So why did we choose a square grid?





…rather than dozens of possible alternatives?

Regular grids make life easy

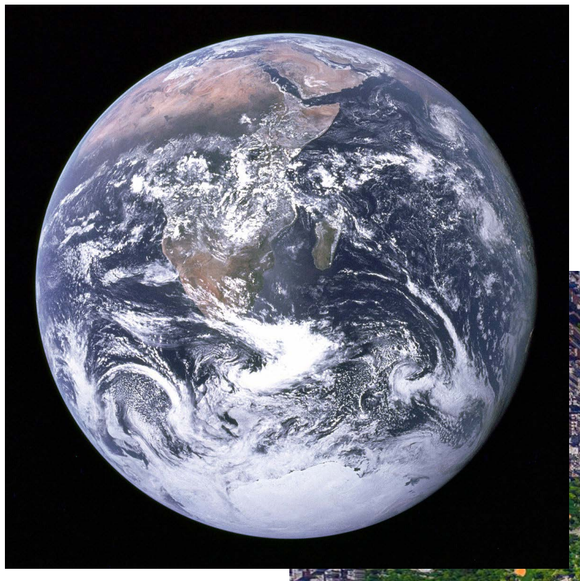


- One reason: SIMPLICITY / EFFICIENCY
  - E.g., always have four neighbors
  - Easy to index, easy to filter…
  - Storage is just a list of numbers
- Another reason: GENERALITY
  - Can encode basically any image
- Are regular grids *always* the best choice for bitmap images?
  - No! E.g., suffer from anisotropy, don't capture edges, ...
  - But *more often than not* are a pretty good choice
- Will see a similar story with geometry...

So, how should we encode surfaces?

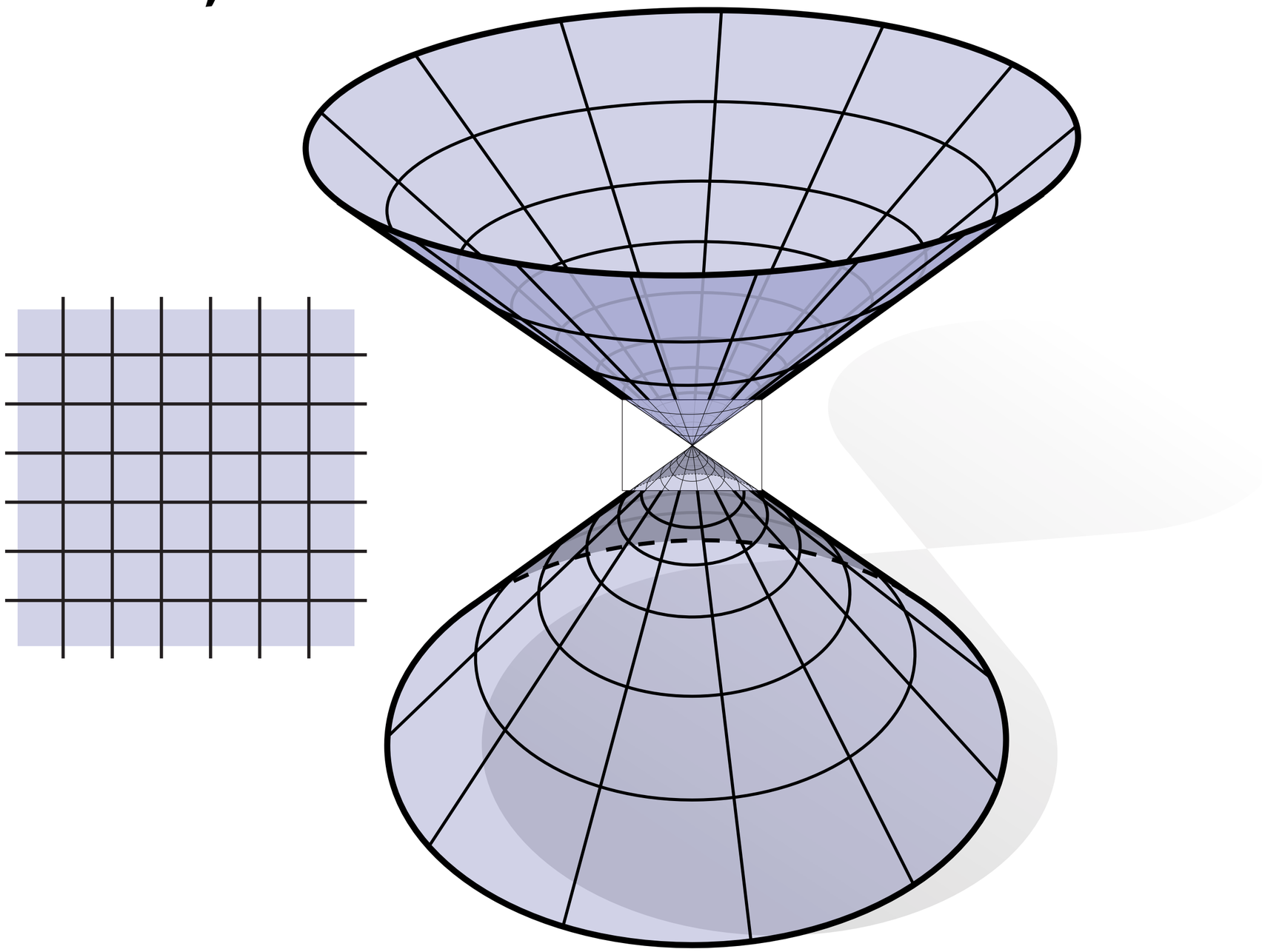
Smooth Surfaces

- Intuitively, a *surface* is the boundary or "shell" of an object
- (Think about the candy shell, not the chocolate.)
- Surfaces are *manifold*:
  - If you zoom in far enough, can draw a regular coordinate grid
  - E.g., the Earth from space vs. from the ground



Isn’t every shape manifold?

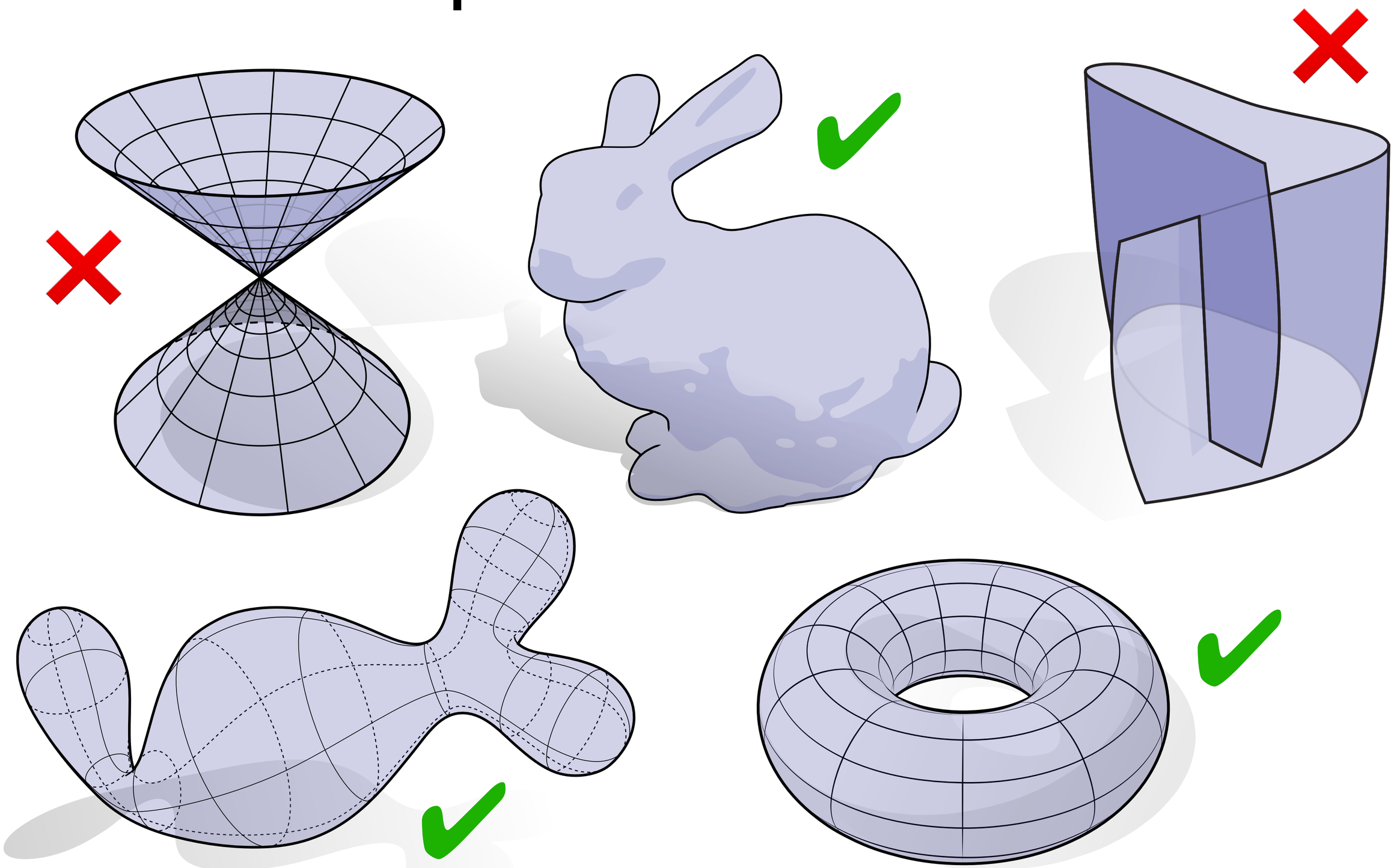
- **No, for instance:**



Can’t draw ordinary 2D grid at center, no matter how close we get.

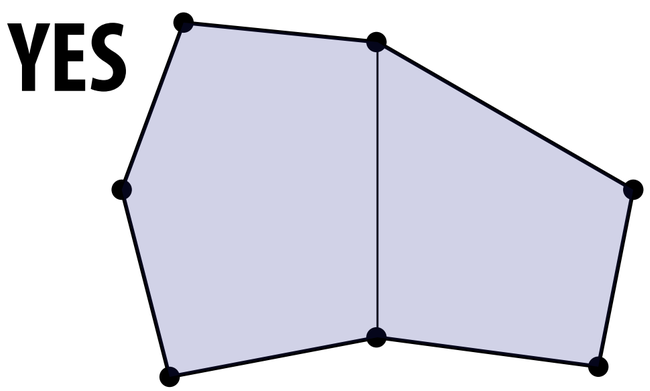
Examples—Manifold vs. Nonmanifold

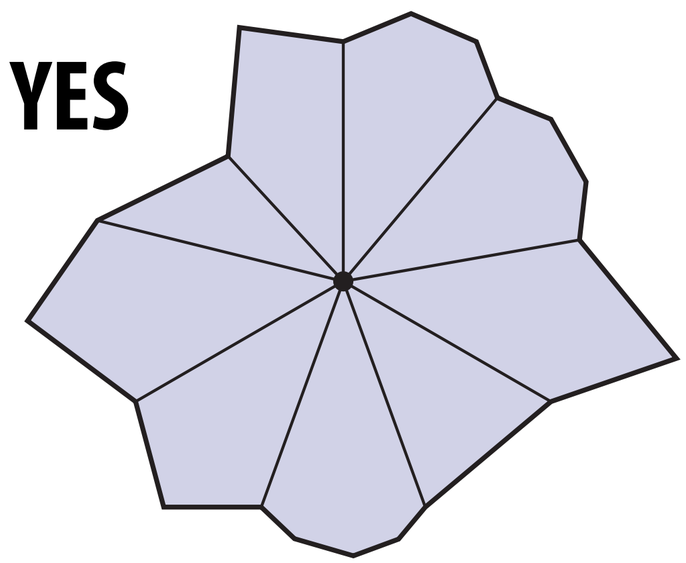
- **Which of these shapes are manifold?**

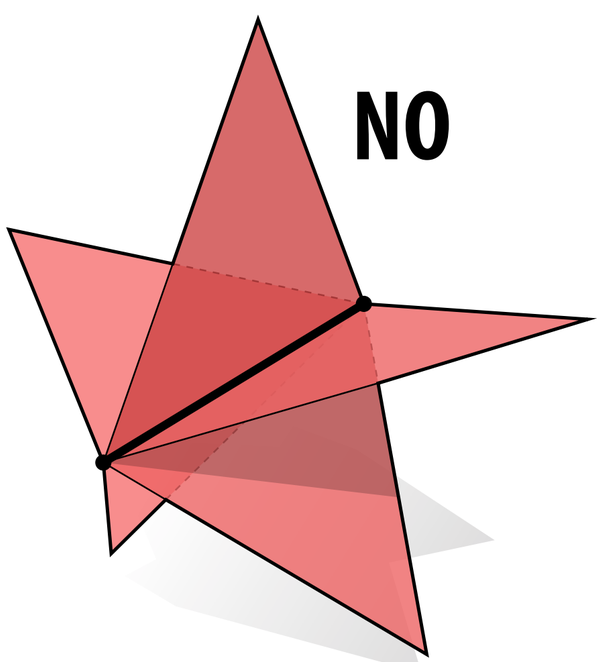


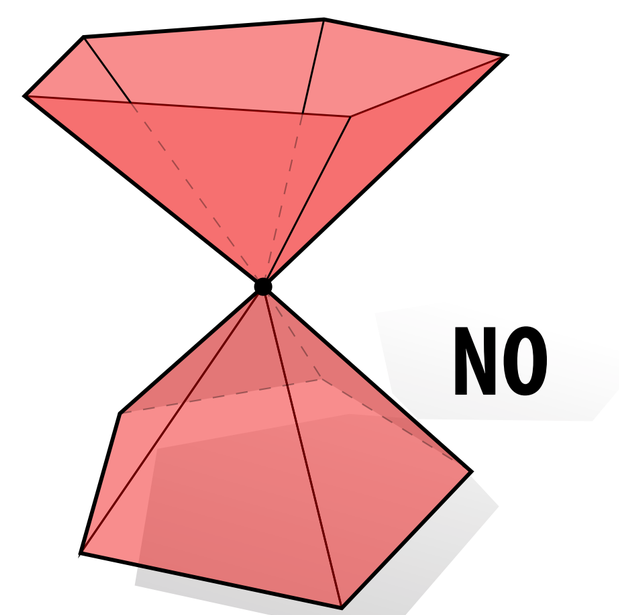
A manifold polygon mesh has fans, not fins

- 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"



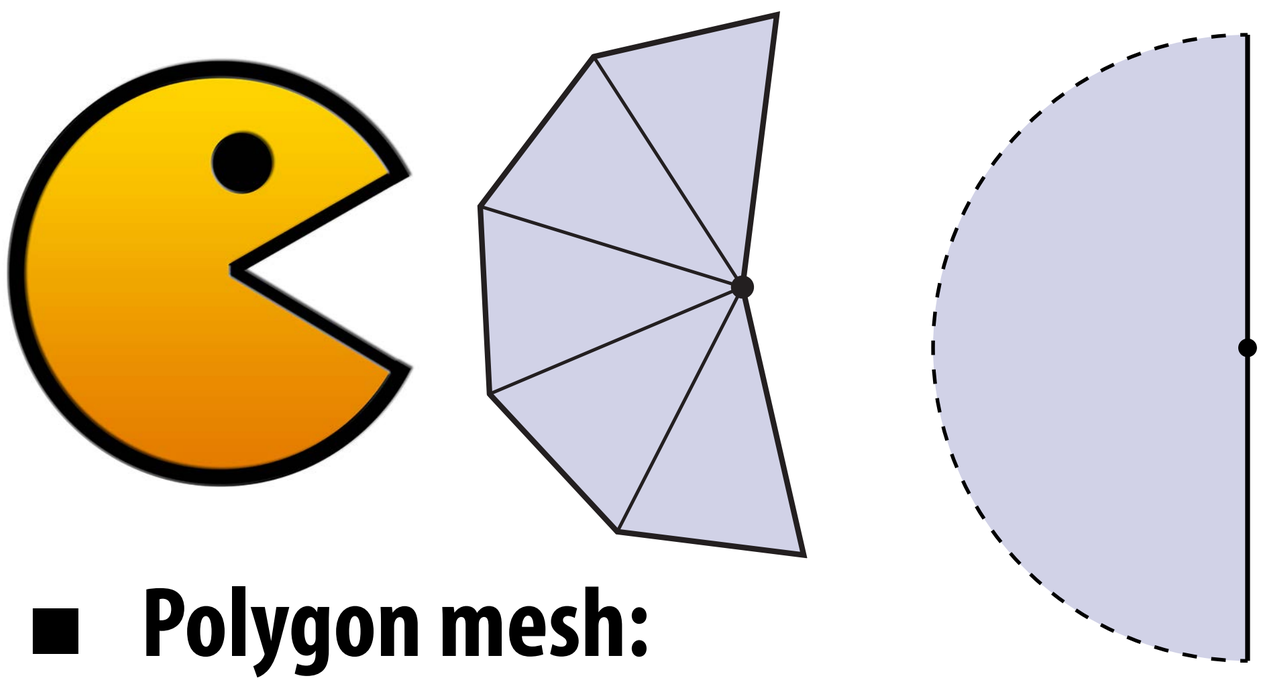






What about boundary?

- The boundary is where the surface "ends."
- E.g., waist & ankles on a pair of pants.
- Locally, looks like a *half* disk
- Globally, each boundary forms a loop



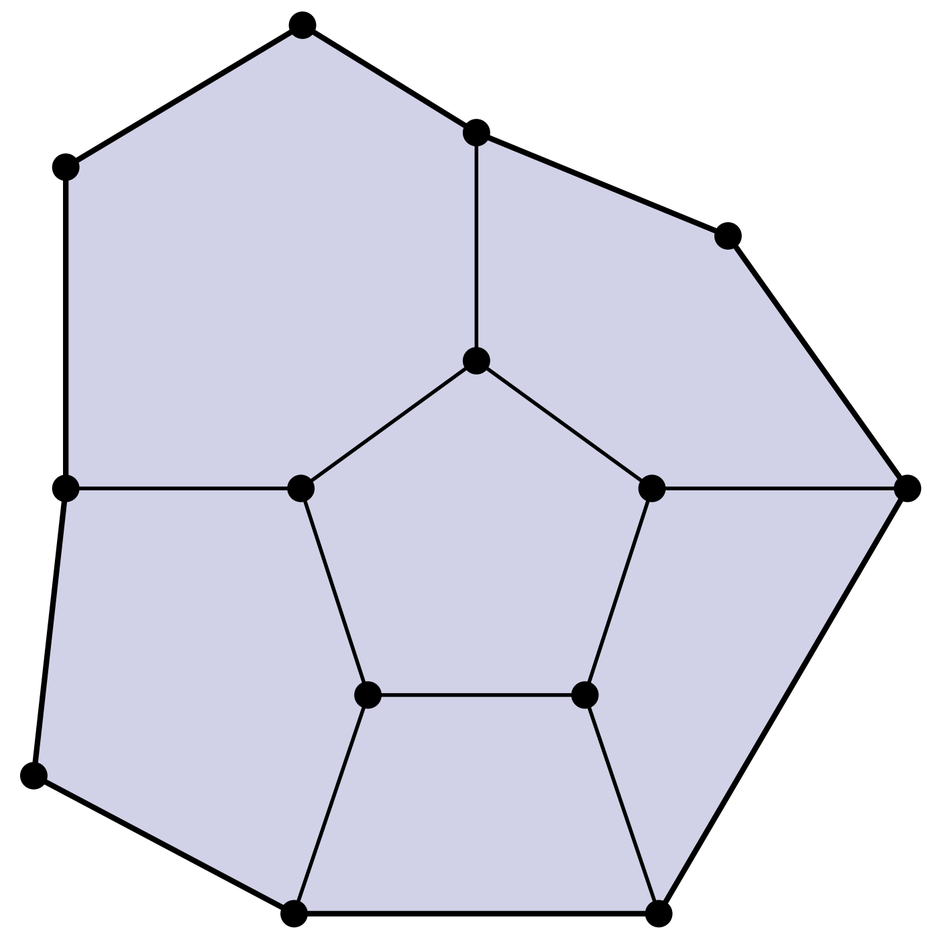
- Polygon mesh:
  - one polygon per boundary edge
  - boundary vertex looks like "pacman"

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**

	(i,j-1)	
(i-1,j)	(i,j)	(i+1,j)
	(i,j+1)	



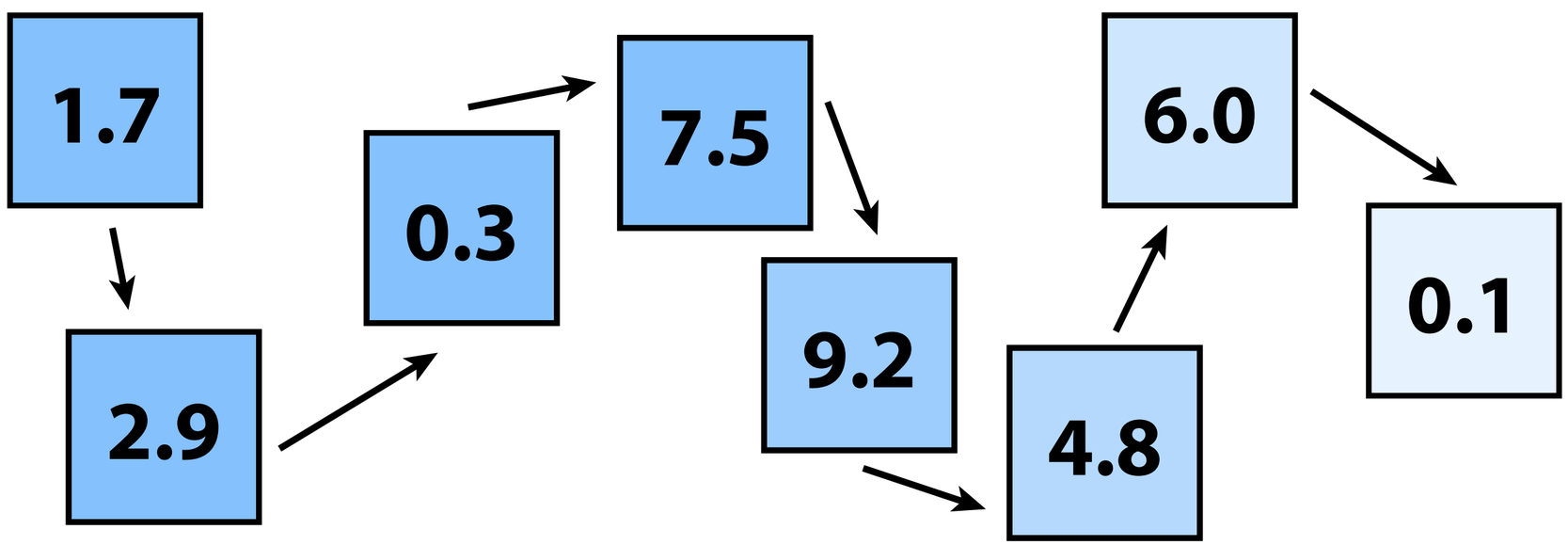
How do we actually encode all this data?

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)

1.7	2.9	0.3	7.5	9.2	4.8	6.0	0.1

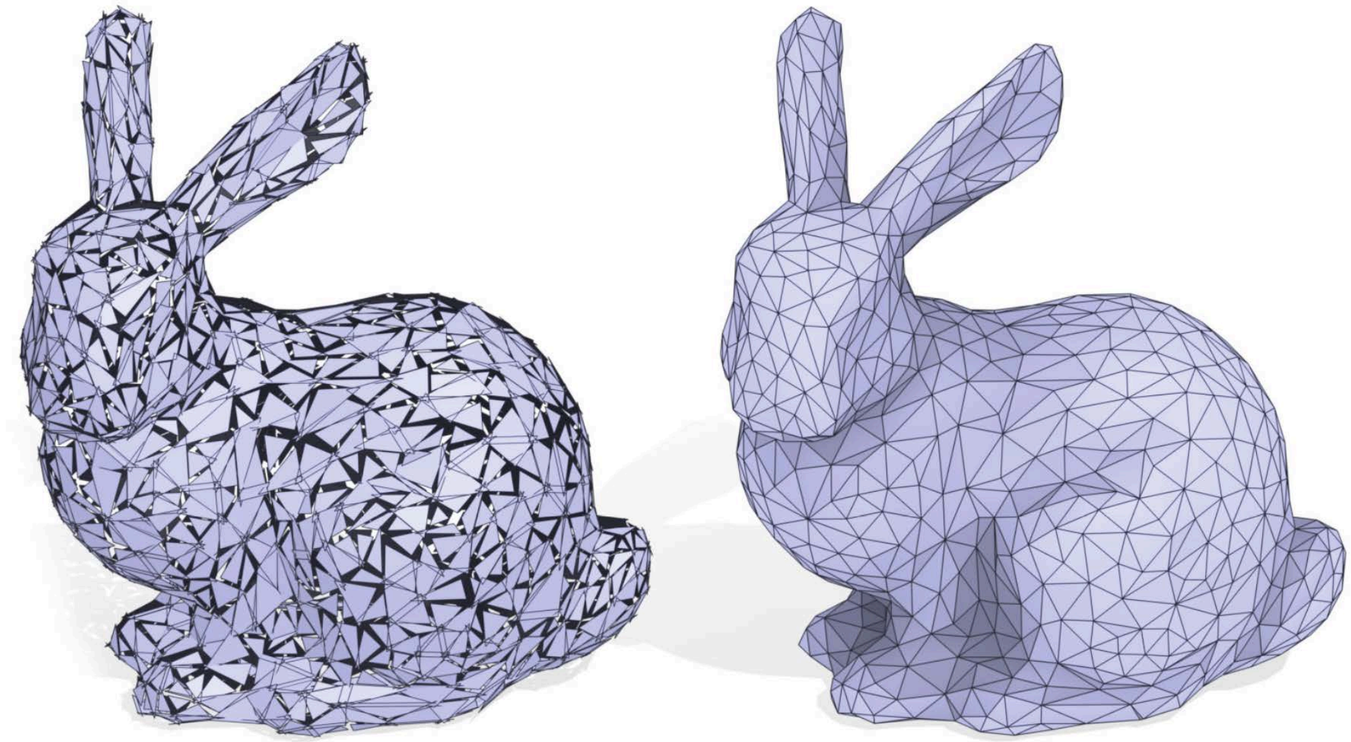
- Alternative: use a linked list (linear lookup, incoherent access)

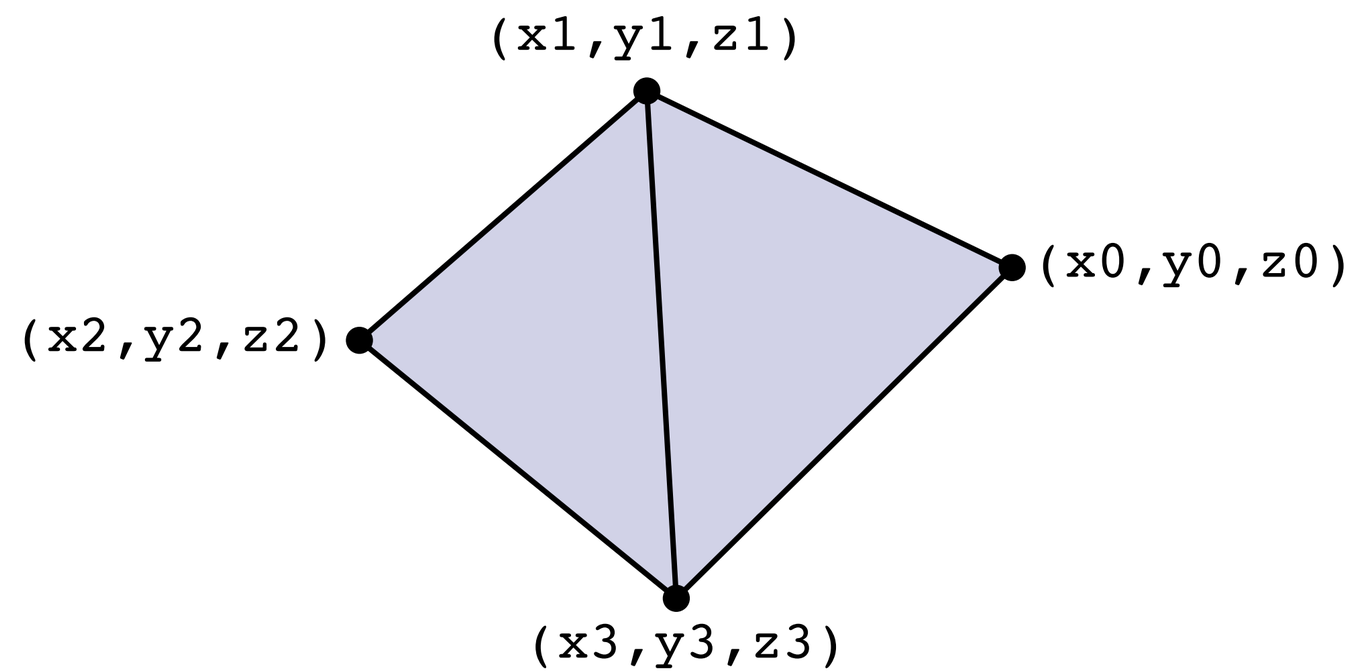


- Q: Why bother with the linked list?
- A: For one, we can easily insert numbers wherever we like...

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





x0,y0,z0	x1,y1,z1	x3,y3,z3
x1,y1,z1	x2,y2,z2	x3,y3,z3

Adjacency List (Array-like)

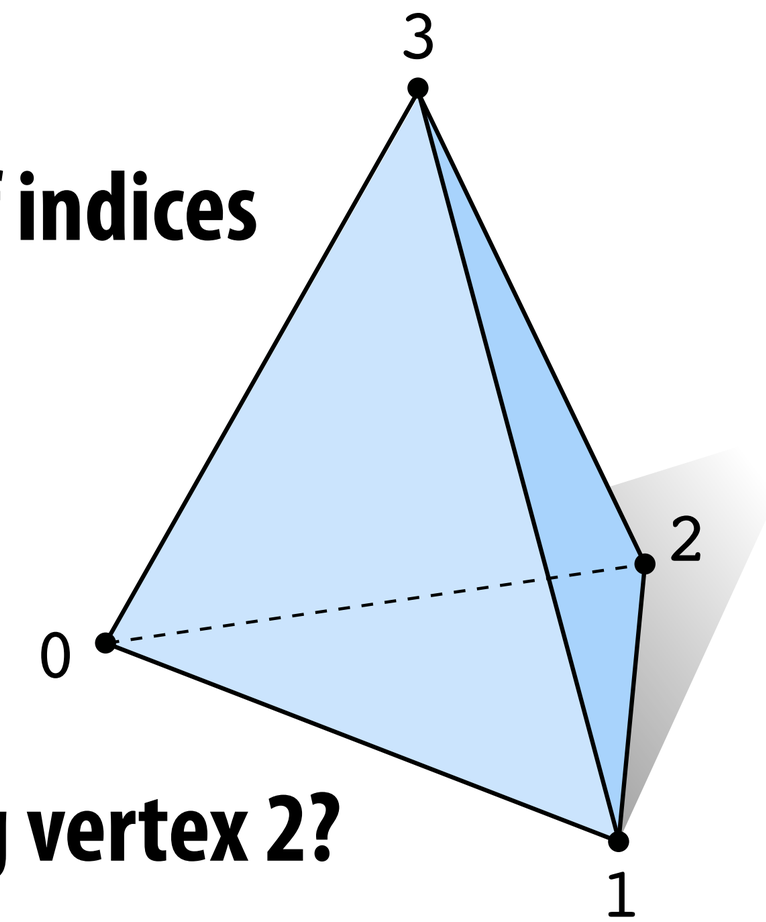
- Store triples of coordinates (x,y,z), tuples of indices
- E.g., tetrahedron:

VERTICES

	x	y	z
0:	-1	-1	-1
1:	1	-1	1
2:	1	1	-1
3:	-1	1	1

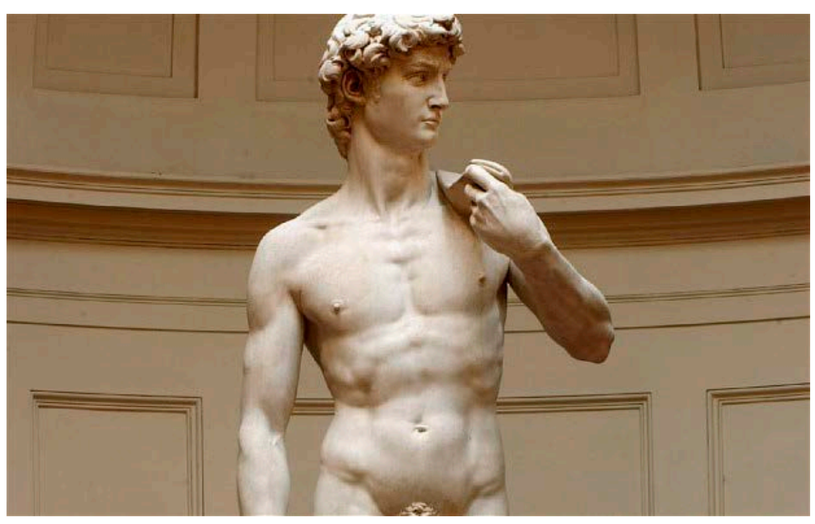
POLYGONS

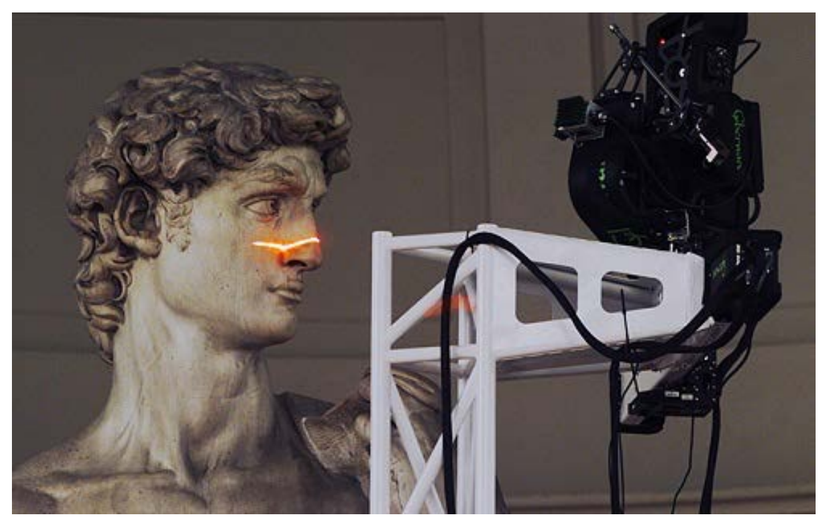
i	j	k
0	2	1
0	3	2
3	0	1
3	1	2

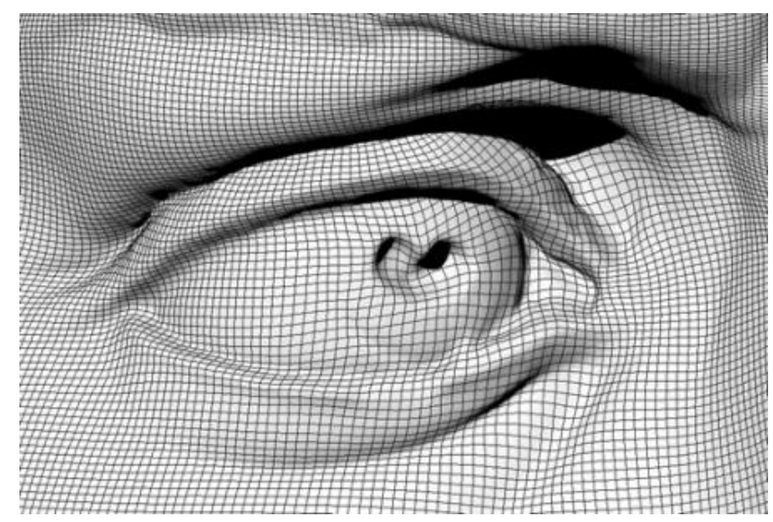


- Q: How do we find all the polygons touching vertex 2?
- Ok, now consider a more complicated mesh:

~1 *billion* polygons







Very **expensive to find the neighboring polygons! (What's the cost?)**

Incidence Matrices

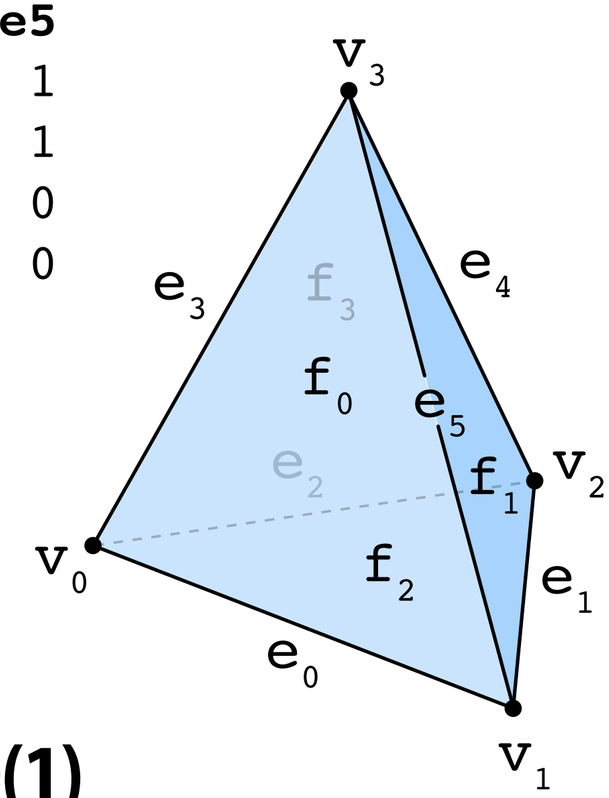
- If we want to know who our neighbors are, why not just store a list of neighbors?
- Can encode all neighbor information via *incidence matrices*
- E.g., tetrahedron:

VERTEX↔EDGE

	v0	v1	v2	v3
e0	1	1	0	0
e1	0	1	1	0
e2	1	0	1	0
e3	1	0	0	1
e4	0	0	1	1
e5	0	1	0	1

EDGE↔FACE

	e0	e1	e2	e3	e4	e5
f0	1	0	0	1	0	1
f1	0	1	0	0	1	1
f2	1	1	1	0	0	0
f3	0	0	1	1	1	0



- 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

Aside: Sparse Matrix Data Structures

$$\begin{array}{c|ccc} & 0 & 1 & 2 \\ \hline 0 & 4 & 2 & 0 \\ 1 & 0 & 0 & 3 \\ 2 & 0 & 7 & 0 \end{array}$$

- Ok, but how do we actually store a "sparse matrix"?
- Lots of possible data structures:
  - <u>Associative array</u> from `(row, column)` to `value`
    - easy to lookup/set entries, fast (e.g., hash table)
    - harder to do matrix operations (e.g., multiplication)
  - <u>Array of linked lists</u> (one per row)
    - conceptually simple
    - slow access time, incoherent memory access
  - <u>Compressed column format</u>—pack entries in list
    - hard to add/modify entries
    - fast for actual matrix operations
- In practice: often build up entries using an "easier" data structure, convert to compressed format for computation

```
(row,col)  val
(0,0) -> 4
(0,1) -> 2
(1,2) -> 3
(2,1) -> 7
```

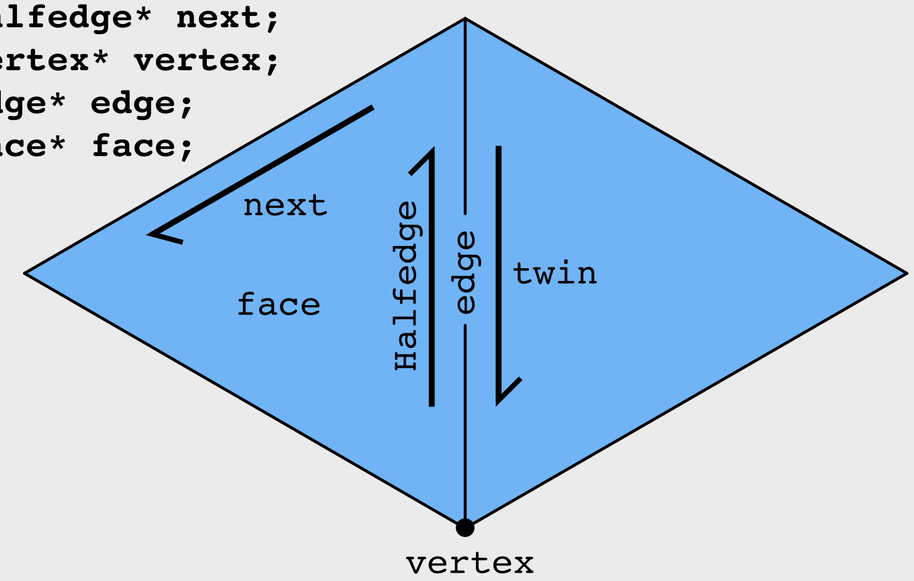
```
                (col,val)   (col,val)
row 0:   (0,4) ---------> (1,2)
    1:   (2,3)
    2:   (1,7)
```

```
values          4,2,7,3
row indices     0,0,2,1
cumulative      1,3,4
 # entries
 by column
```

Halfedge Data Structure (Linked-list-like)

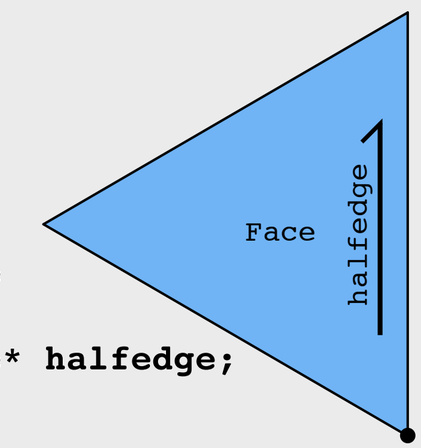
- Store *some* information about neighbors
- Don't need an exhaustive list; just a few key pointers
- Key idea: two *halfedges* act as "glue" between mesh elements:

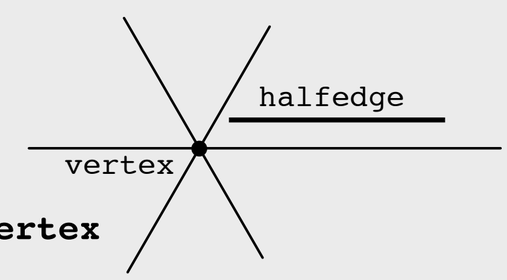
```
struct Halfedge
{
   Halfedge* twin;
   Halfedge* next;
   Vertex* vertex;
   Edge* edge;
   Face* face;
};
```




```
struct Edge
{
   Halfedge* halfedge;
};
```

```
struct Face
{
   Halfedge* halfedge;
};
```





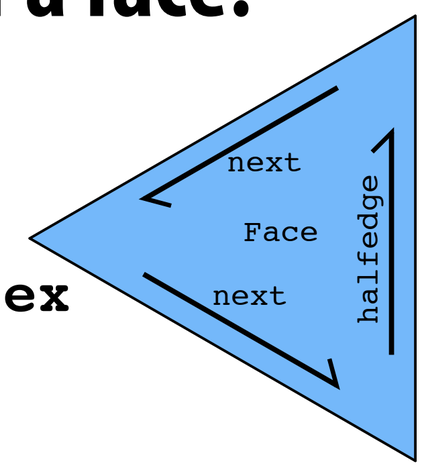

```
struct Vertex
{
   Halfedge* halfedge;
};
```

- Each vertex, edge face points to just *one* of its halfedges.

Halfedge makes mesh traversal easy

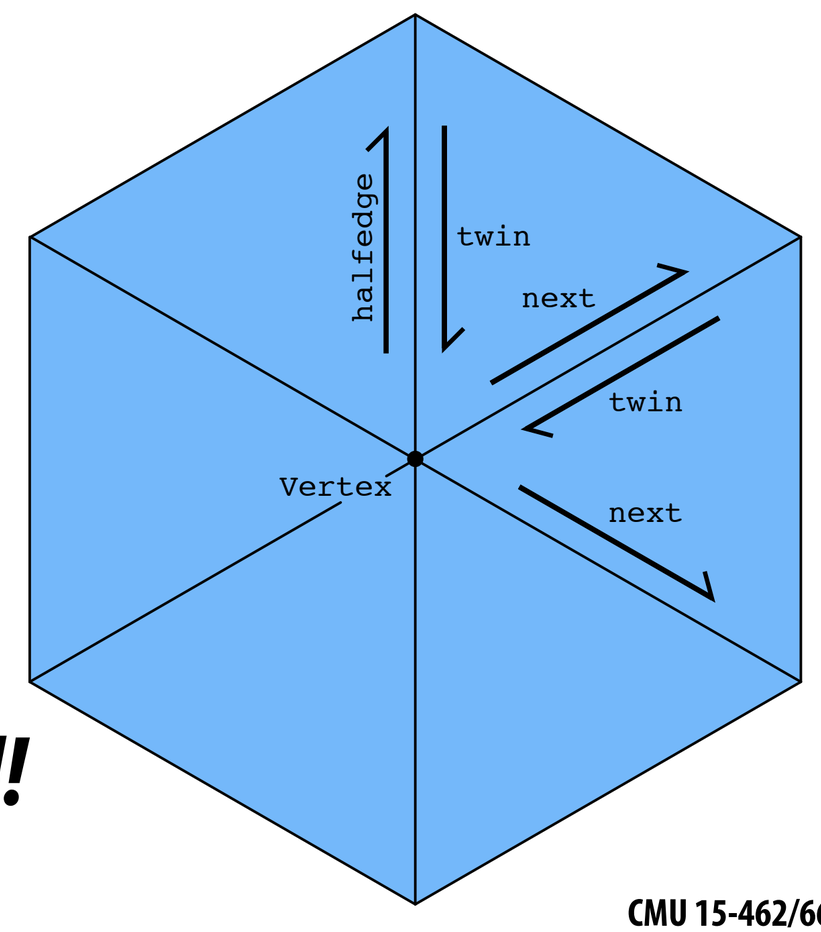
- Use "twin" and "next" pointers to move around mesh
- Use "vertex", "edge", and "face" pointers to grab element
- Example: visit all vertices of a face:

```
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

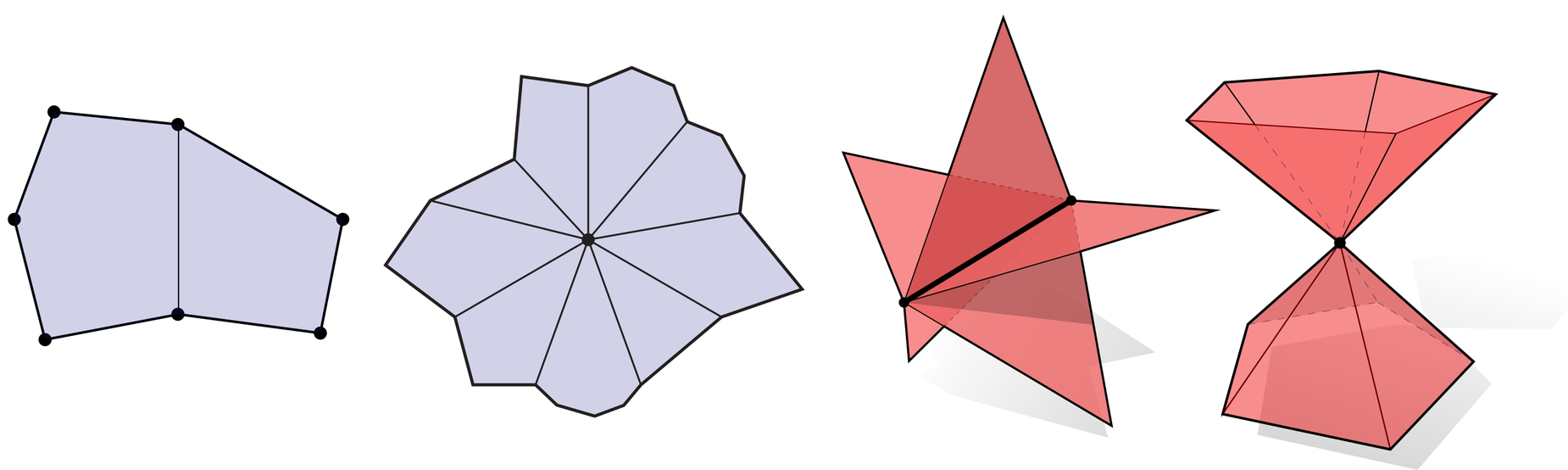
- Consider simplified halfedge data structure
- Require only "common-sense" conditions

(pointer to yourself!)

```
struct Halfedge {
   Halfedge *next, *twin;
};
```

```
twin->twin == this
twin != this
every he is someone's "next"
```

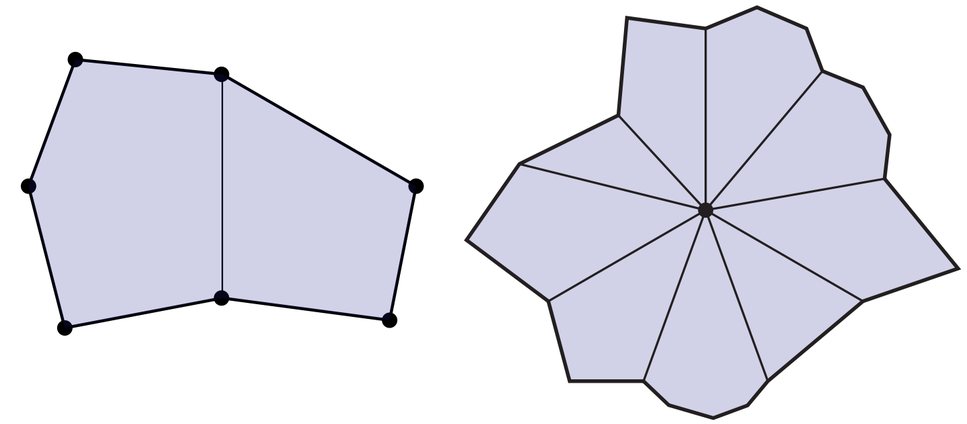
- Keep following `next`, and you'll get faces.
- Keep following `twin` and you'll get edges.
- Keep following `next->twin` and you'll get vertices.

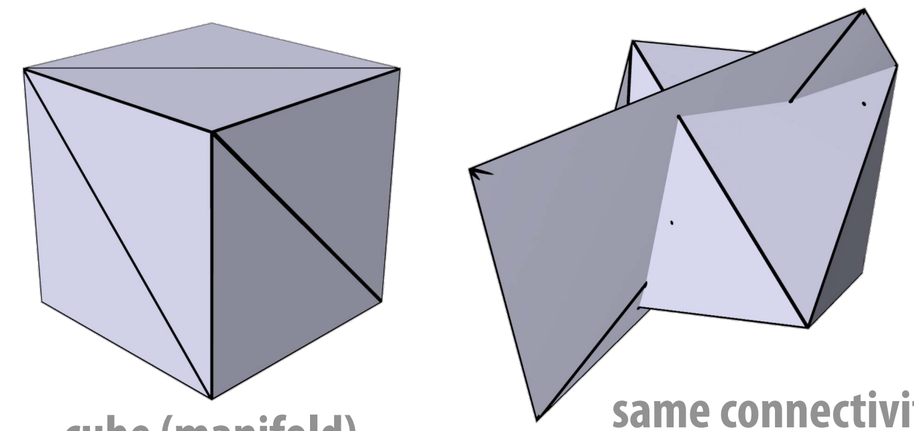


Q: Why, therefore, is it impossible to encode the red figures?

Connectivity vs. Geometry

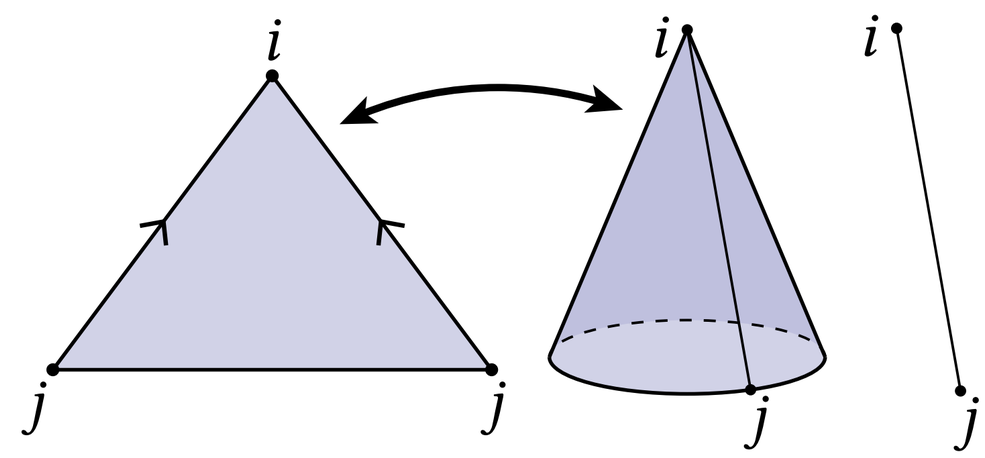
- Recall manifold conditions (fans not fins):
  - every edge contained in two faces
  - every vertex contained in one fan
- These conditions say <u>nothing</u> about vertex positions! Just connectivity
- Hence, can have perfectly good (manifold) connectivity, even if geometry is awful
- In fact, sometimes you can have perfectly good manifold connectivity for which <u>any</u> vertex positions give "bad" geometry!
- Can lead to confusion when debugging: mesh looks "bad", even though connectivity is fine

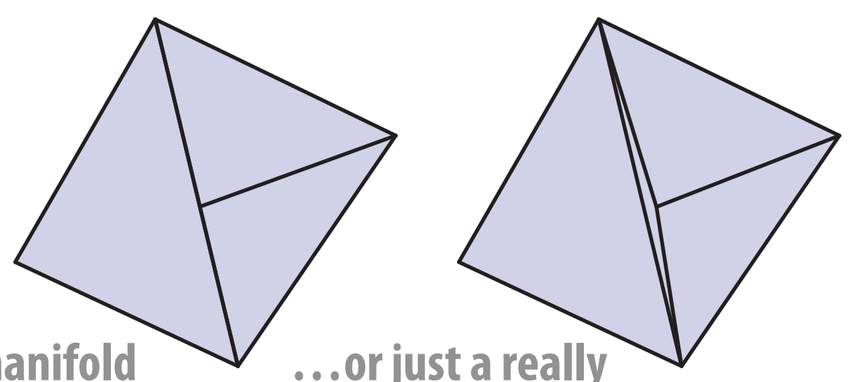




cube (manifold)

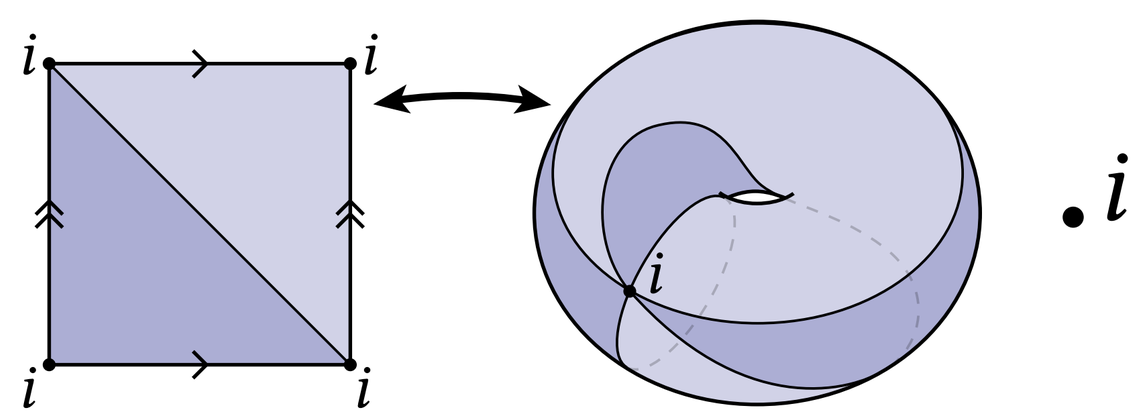
same connectivity, random vertex positions





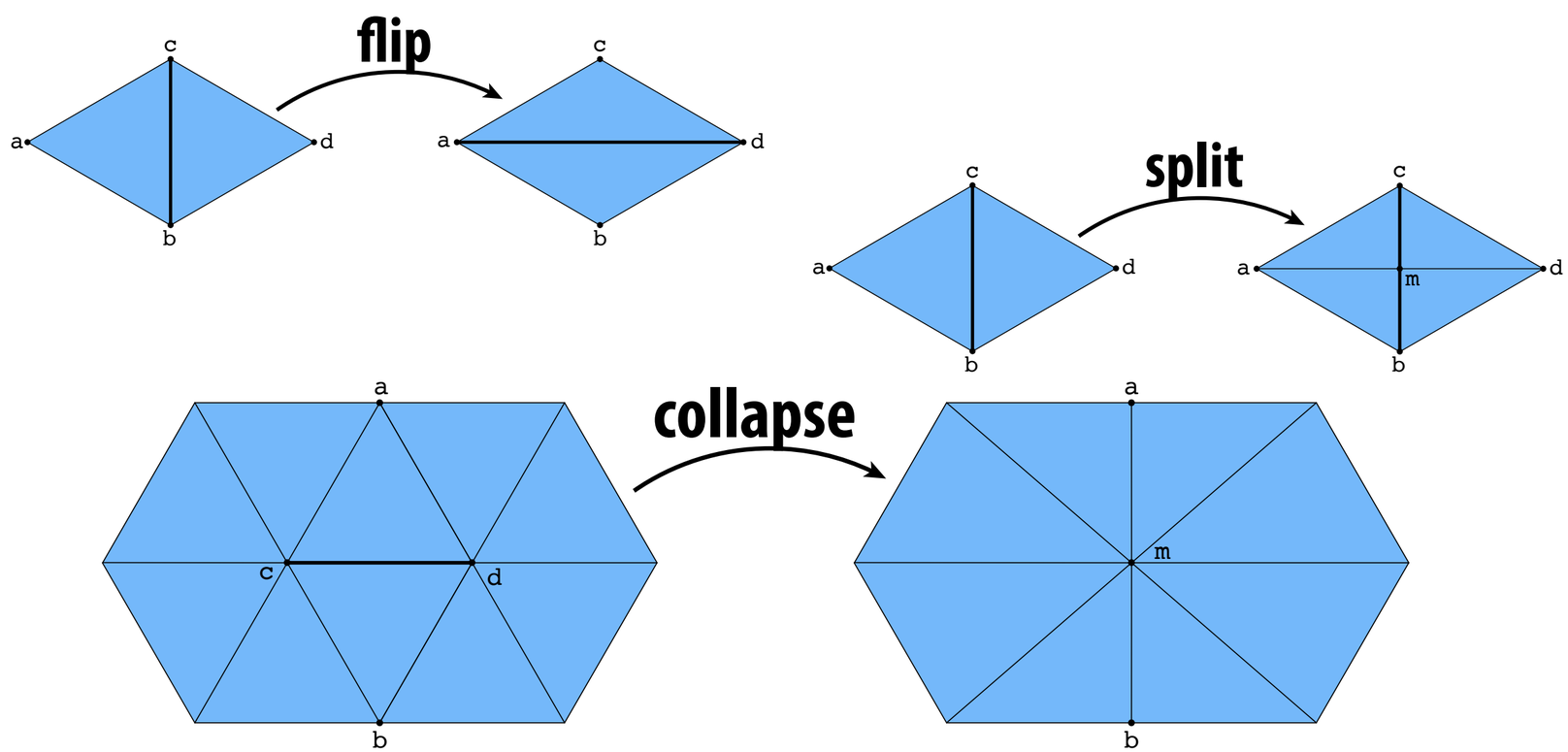
non manifold connectivity?

…or just a really skinny triangle?



Halfedge meshes are easy to edit

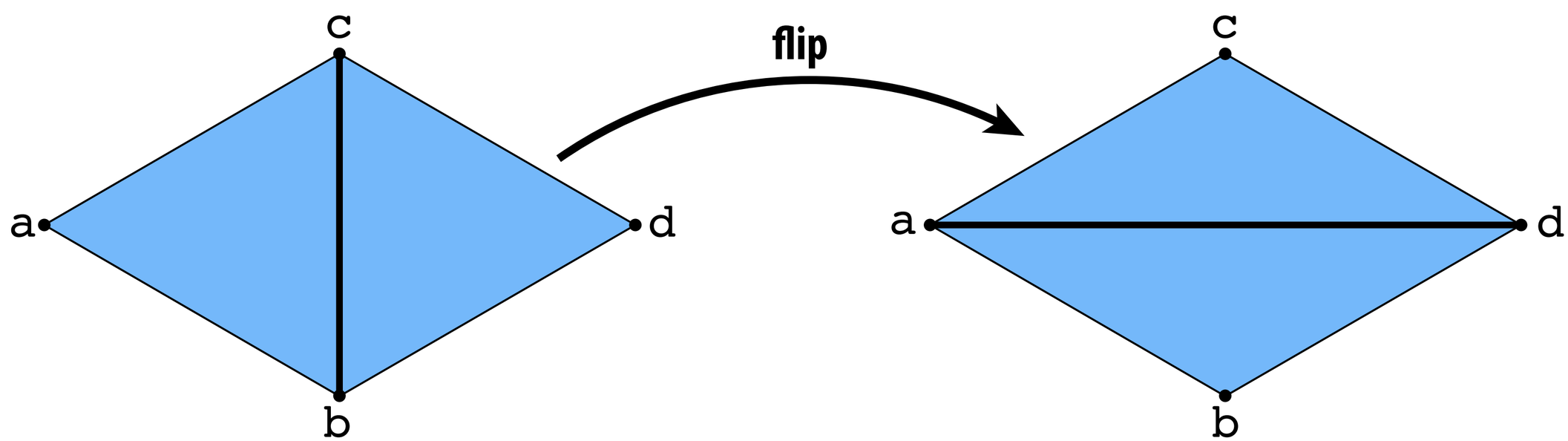
- Remember key feature of linked list: insert/delete elements
- Same story with halfedge mesh ("linked list on steroids")
- E.g., for triangle meshes, several atomic operations:



- How? Allocate/delete elements; reassigning pointers.
- Must be careful to preserve manifoldness!

Edge Flip (Triangles)

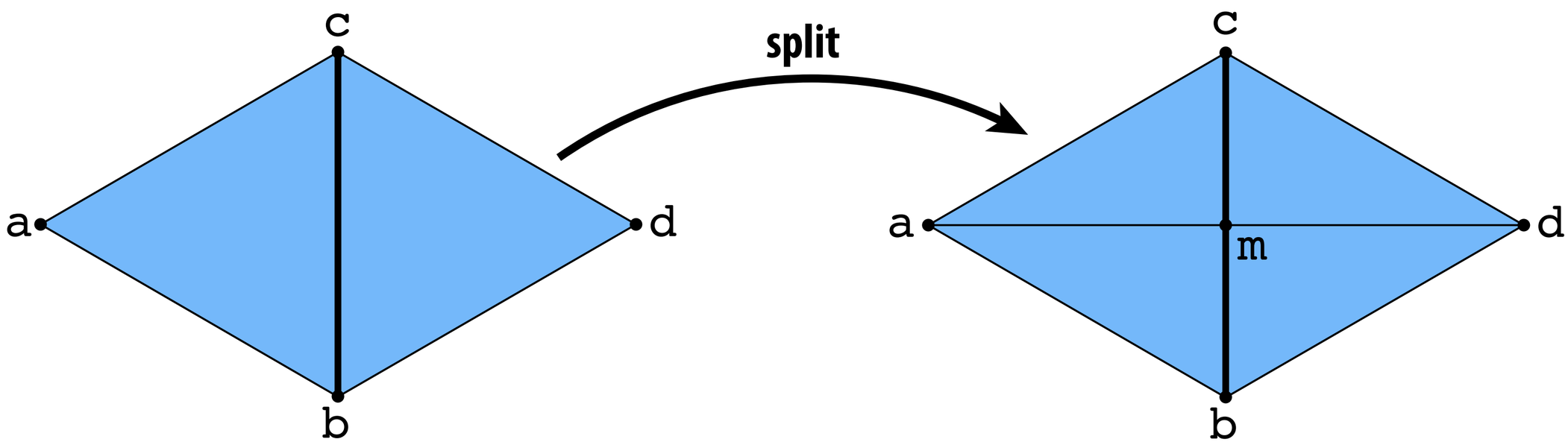
- Triangles (a,b,c), (b,d,c) become (a,d,c), (a,b,d):



- Long list of pointer reassignments (`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)

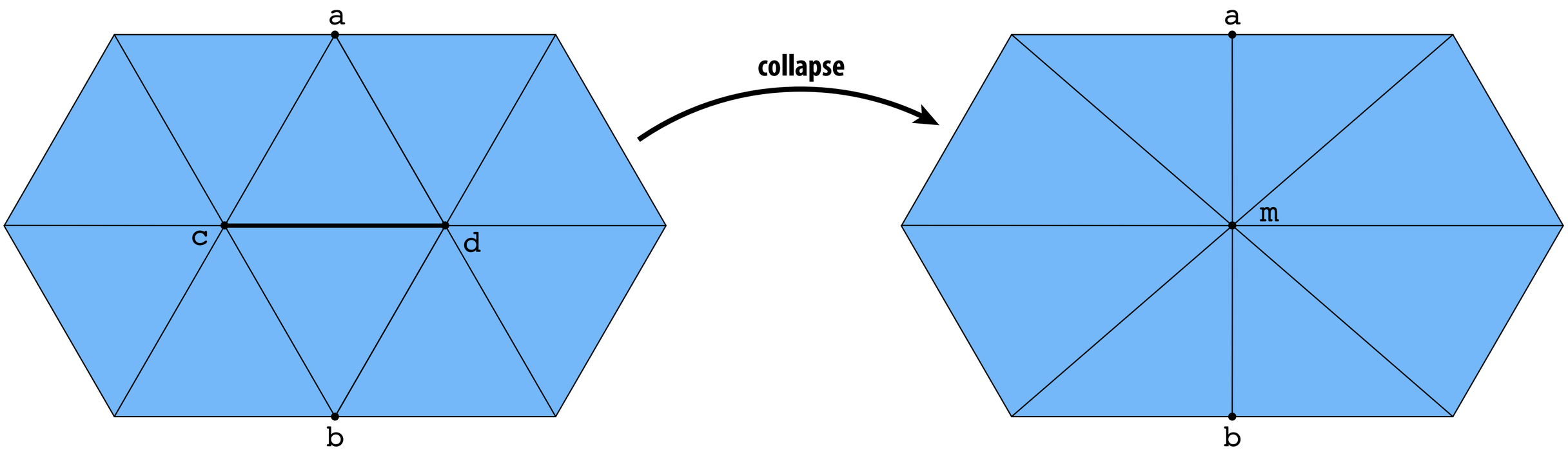
- Insert midpoint m of edge (c,b), connect to get four triangles:



- This time, have to *add* new elements.
- Lots of pointer reassignments.
- Q: Can we "reverse" this operation?

Edge Collapse (Triangles)

- Replace edge (b,c) with a single vertex m:

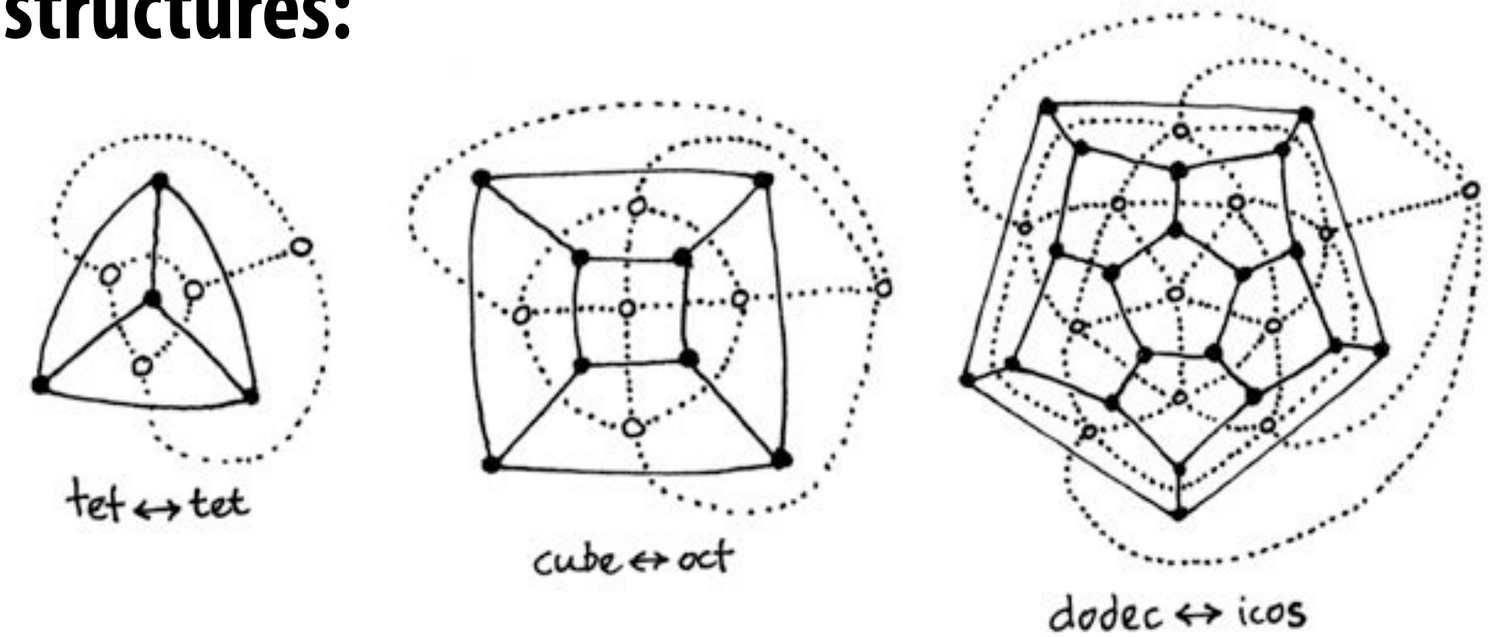


- Now have to *delete* elements.
- Still lots of pointer assignments!
- Q: How would we implement this with an adjacency list?
- Any other good way to do it? (E.g., different data structure?)

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/

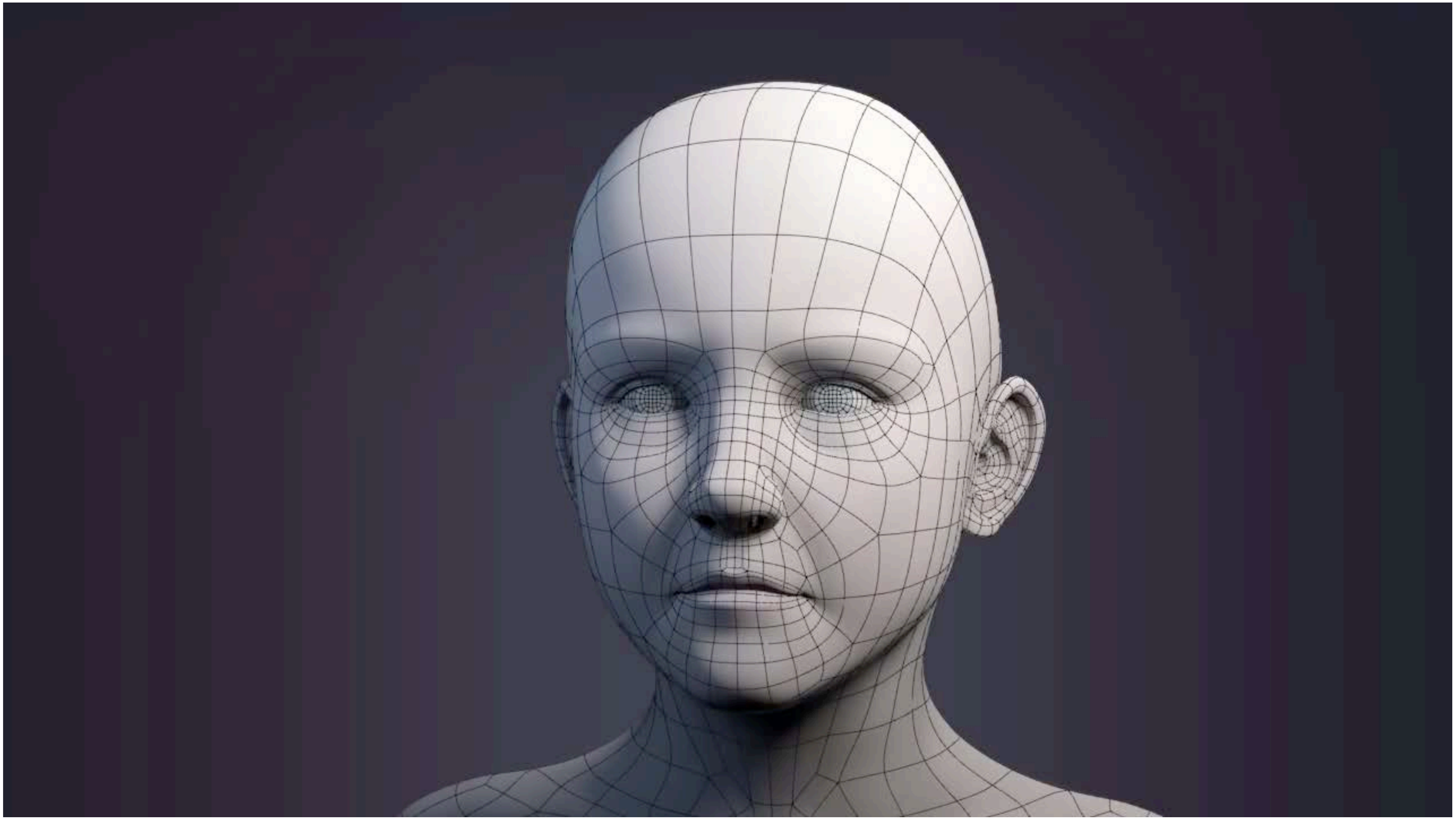
Comparison of Polygon Mesh Data Strucutres

	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!

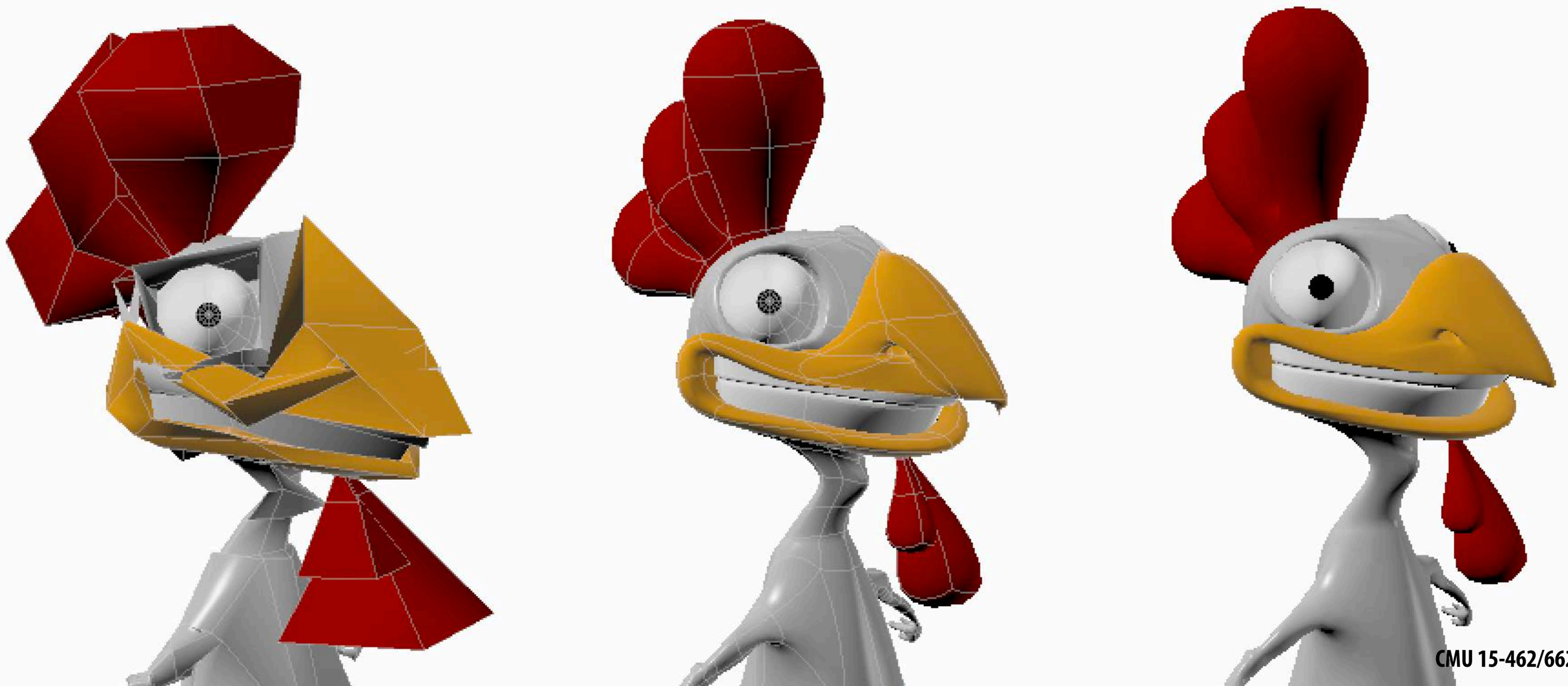
Ok, but what can we actually *do* with our fancy new data structures?

Subdivision Modeling



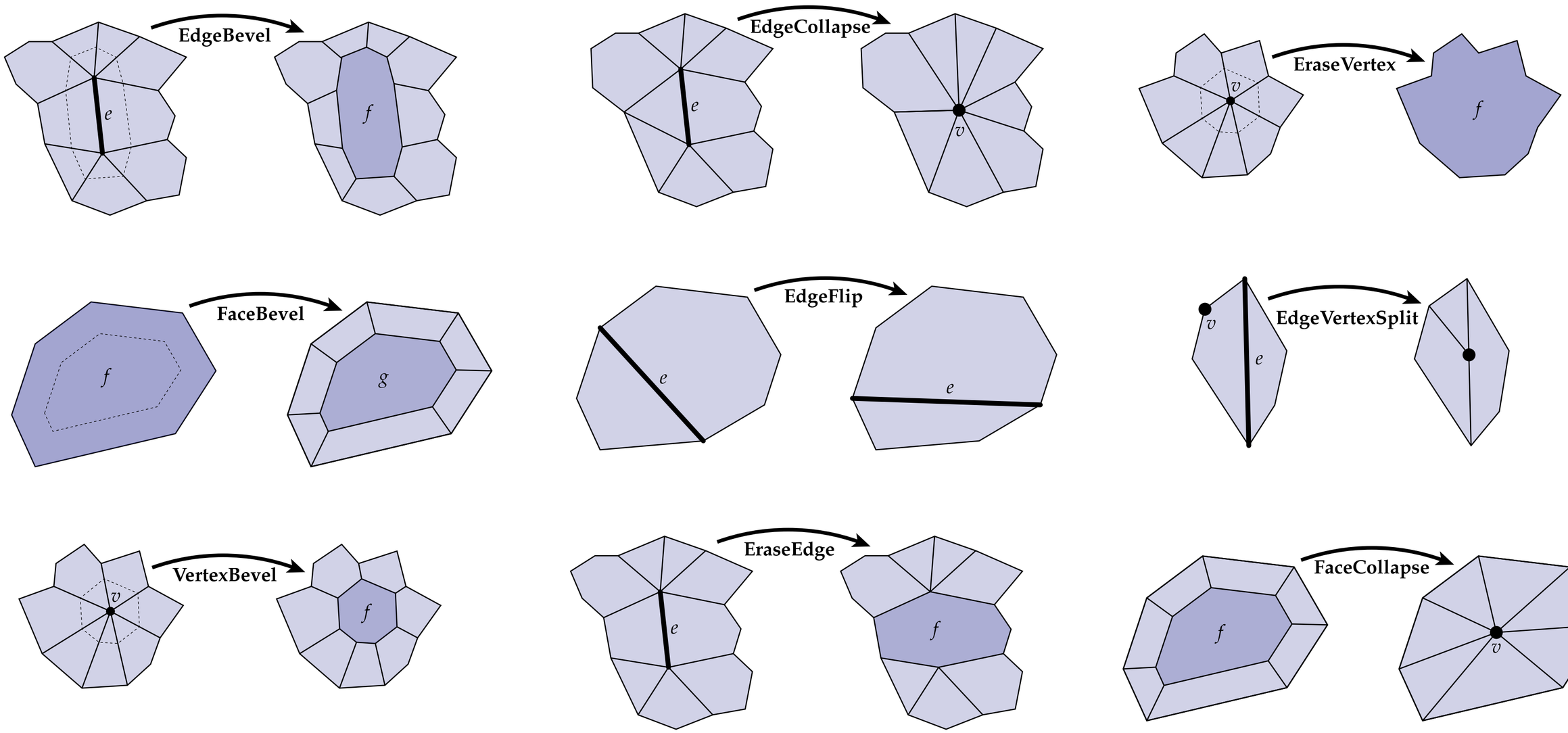
Subdivision Modeling

- 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

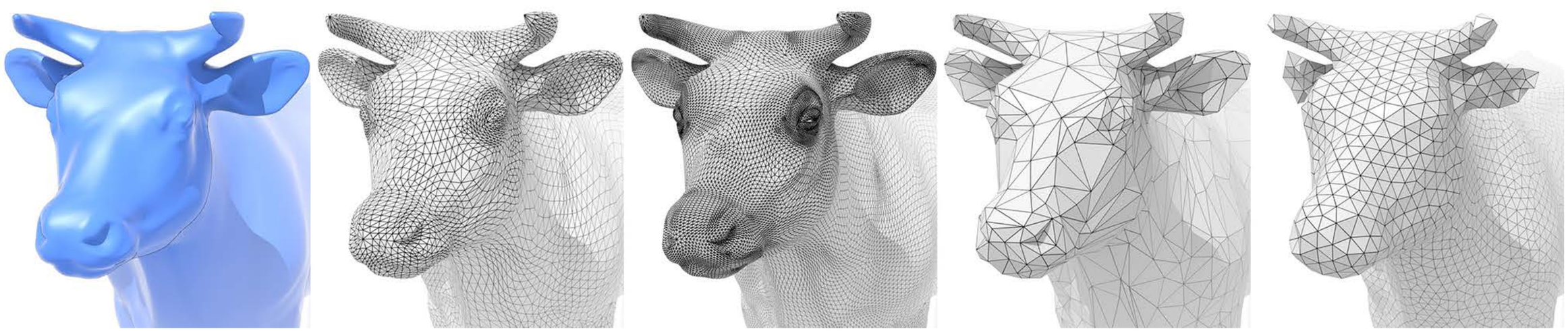
- **For general polygon meshes, we can dream up lots of local mesh operations that might be useful for modeling:**



…and many, many more!

Next Time: Digital Geometry Processing

- Extend traditional digital signal processing (audio, video, etc.) to deal with *geometric* signals:
  - upsampling / downsampling / resampling / filtering ...
  - aliasing (reconstructed surface gives "false impression")

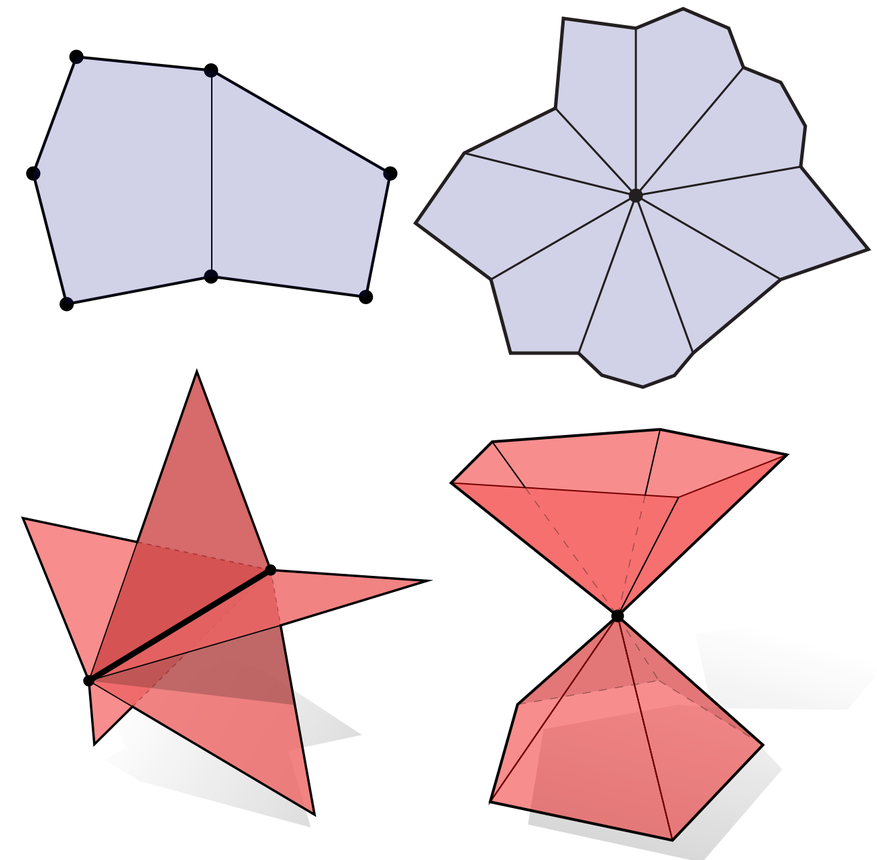


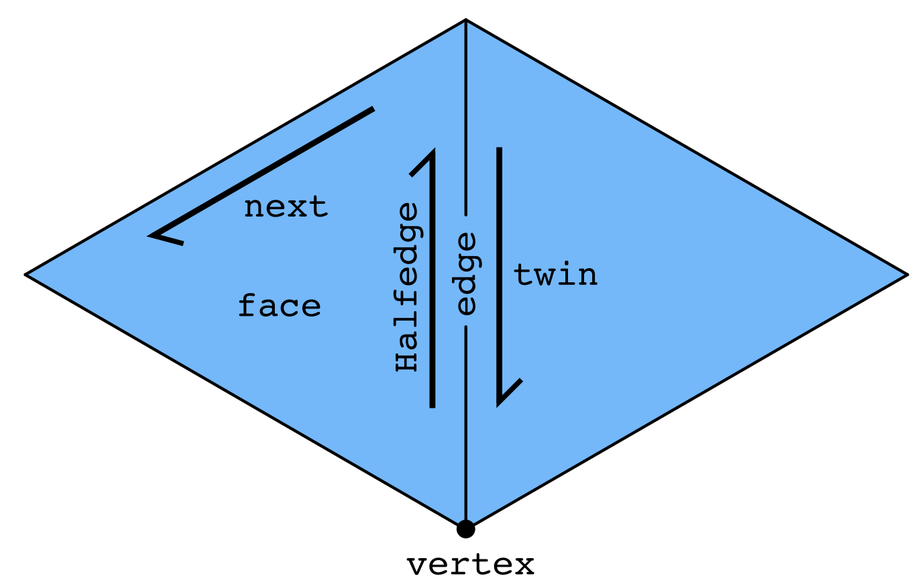
Digital Geometry Processing

Computer Graphics
CMU 15-462/15-662

Last time: Meshes & Manifolds

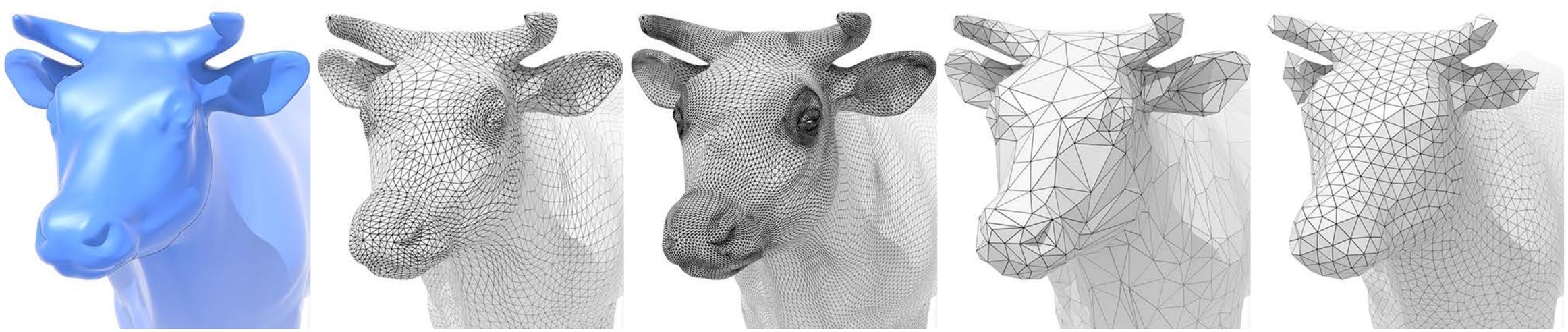
- Mathematical description of geometry
  - simplifying assumption: *manifold*
  - for polygon meshes: "fans, not fins"
- Data structures for surfaces
  - polygon soup
  - halfedge mesh
  - storage cost vs. access time, etc.
- Today:
  - how do we manipulate geometry?
  - geometry processing / resampling





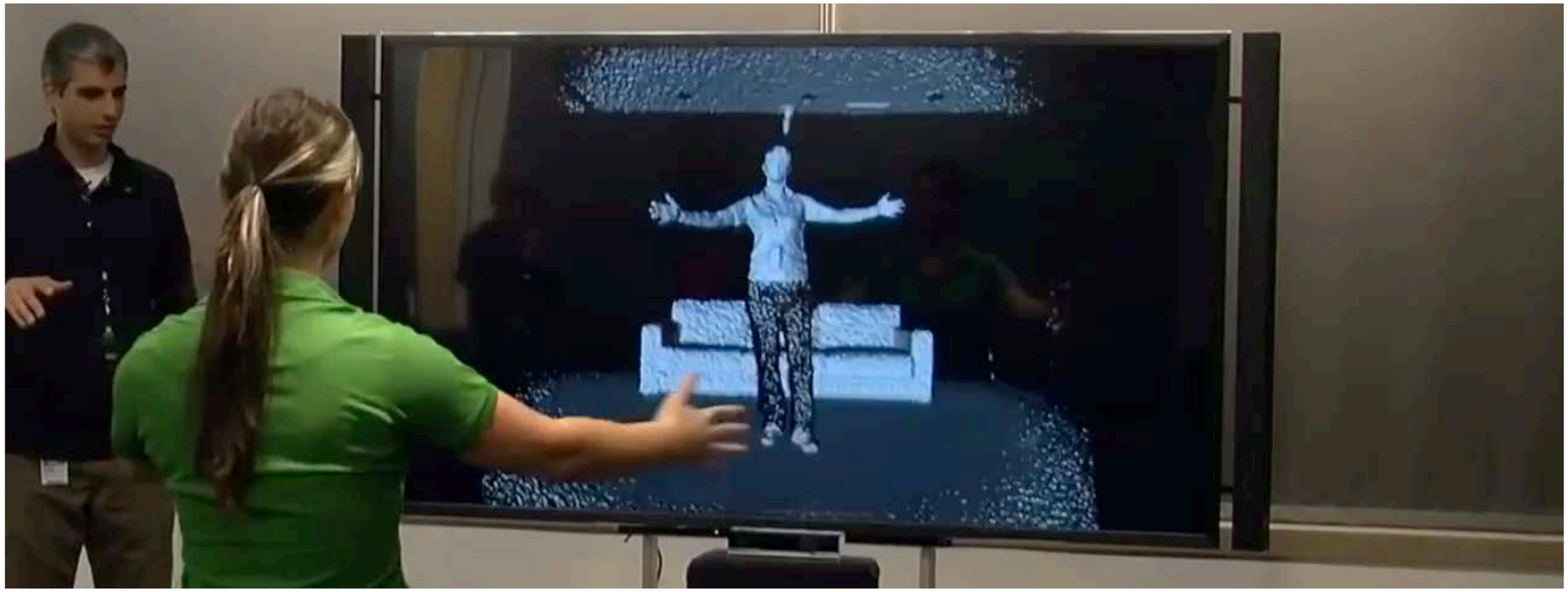
Today: Geometry Processing

- Extend traditional digital signal processing (audio, video, etc.) to deal with *geometric* signals:
  - upsampling / downsampling / resampling / filtering ...
  - aliasing (reconstructed surface gives "false impression")
- Beyond pure geometry, these are basic building blocks for many areas/algorithms in graphics (rendering, animation...)

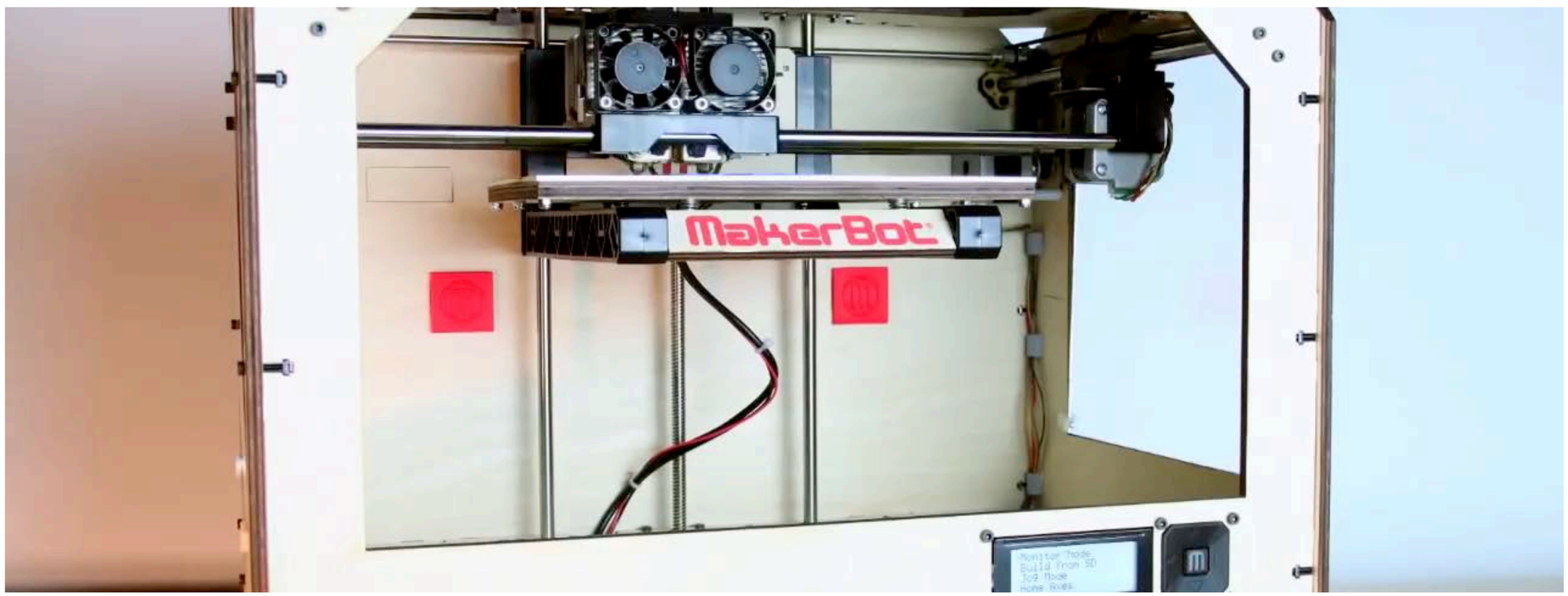


Digital Geometry Processing: Motivation

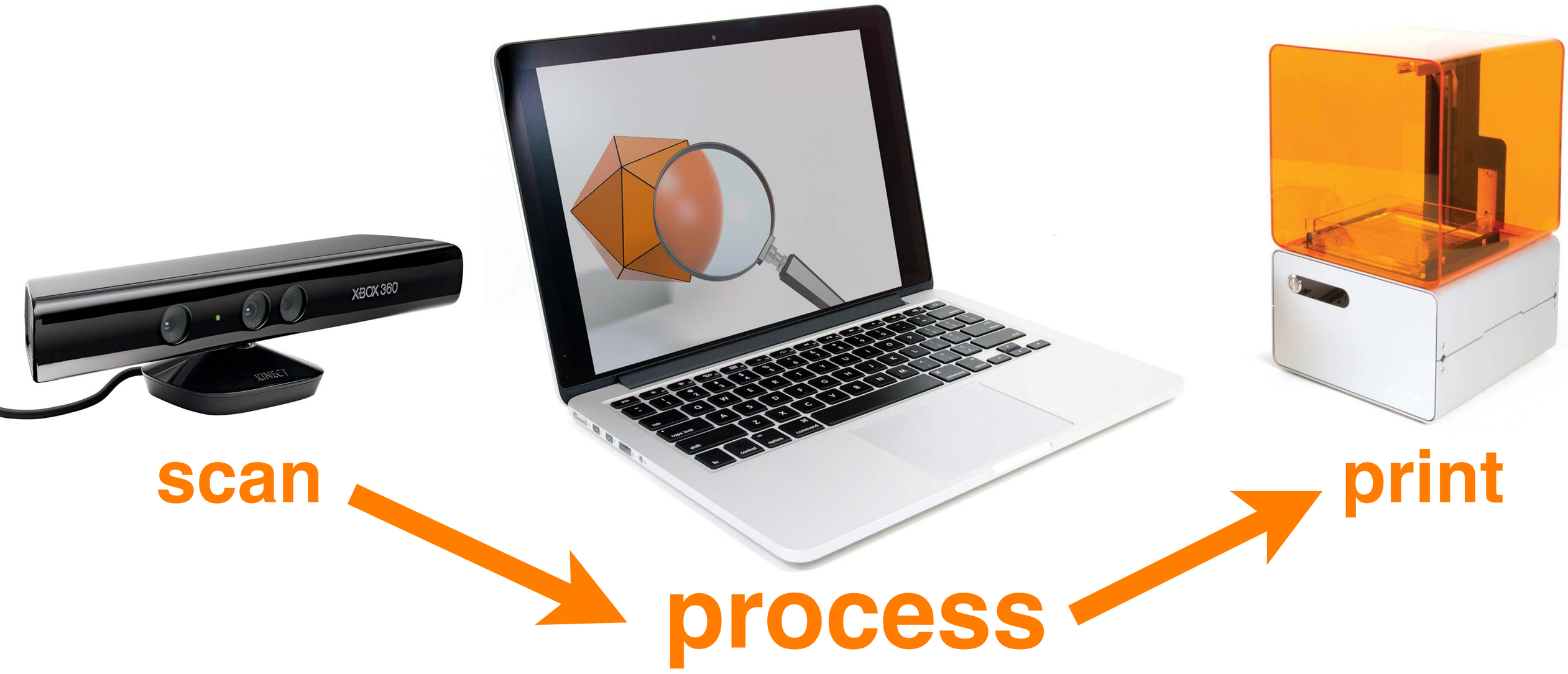
3D Scanning



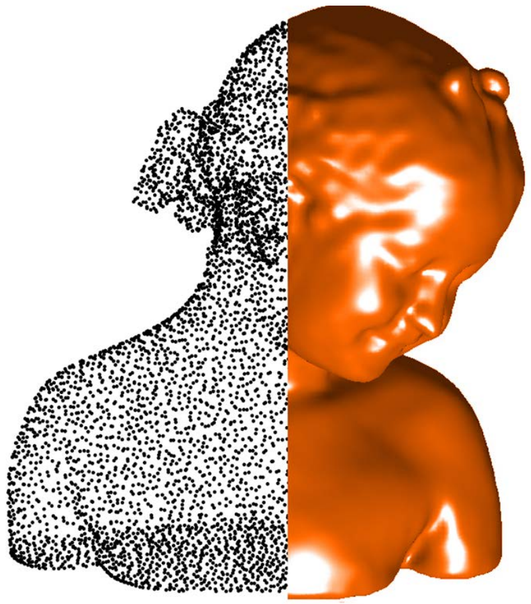
3D Printing



Geometry Processing Pipeline

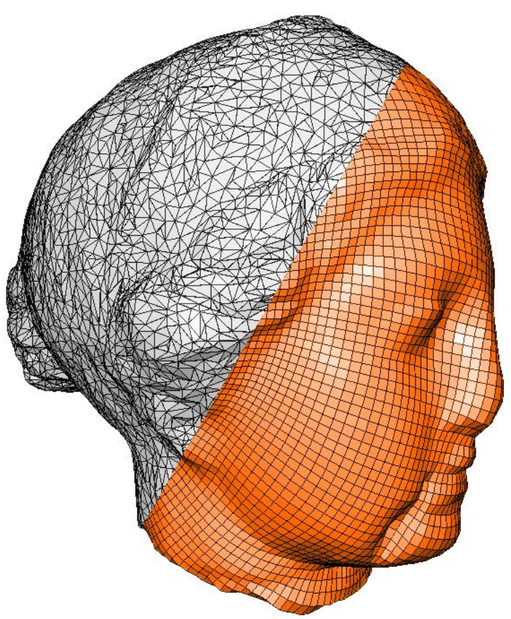


Geometry Processing Tasks

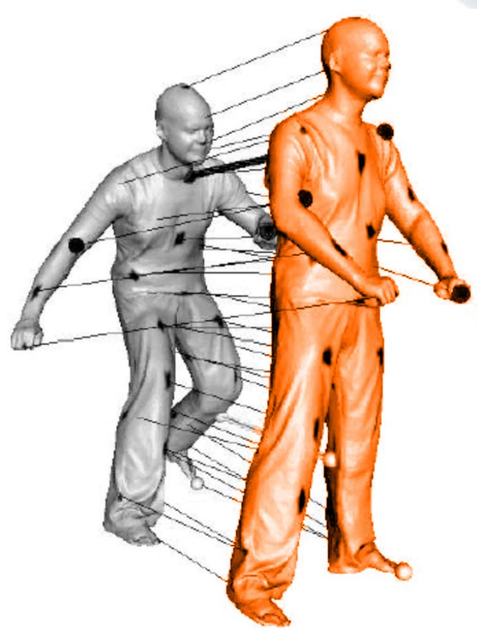


reconstruction

filtering

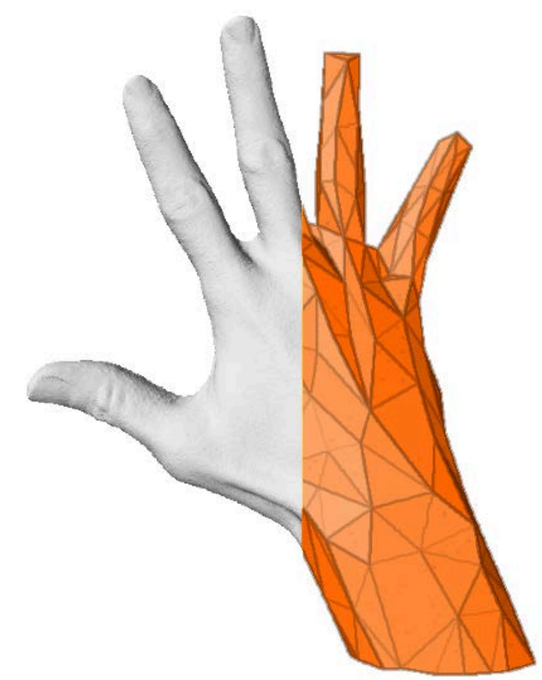


remeshing



shape analysis

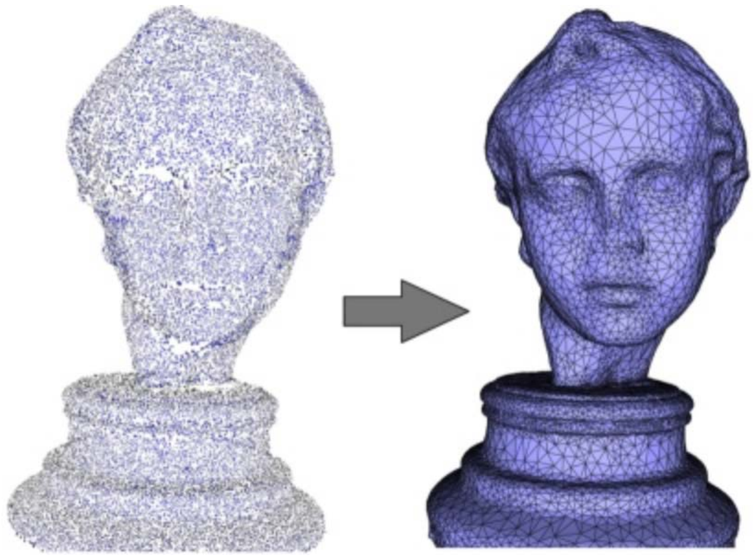
parameterization



compression

Geometry Processing: Reconstruction

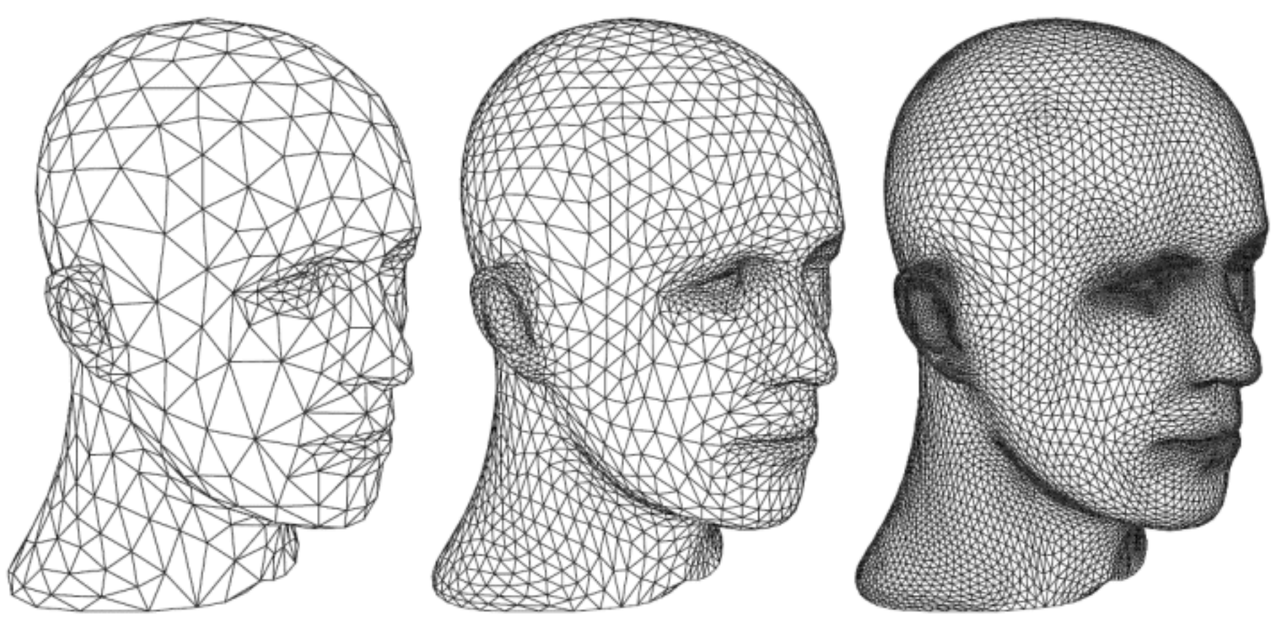
- Given samples of geometry, reconstruct surface
- What are "samples"? Many possibilities:
  - points, points & normals, ...
  - image pairs / sets (multi-view stereo)
  - line density integrals (MRI/CT scans)



- 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)

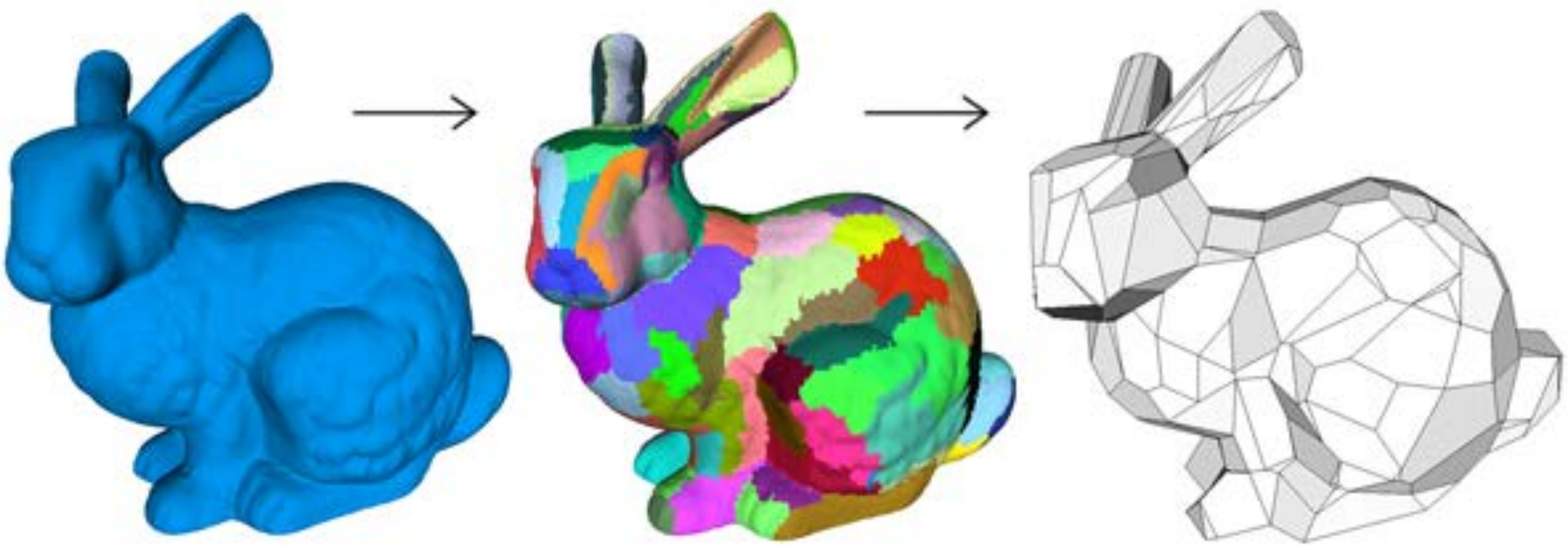
Geometry Processing: Upsampling

- Increase resolution via interpolation
- Images: e.g., bilinear, bicubic interpolation
- Polygon meshes:
  - subdivision
  - bilateral upsampling
  - ...



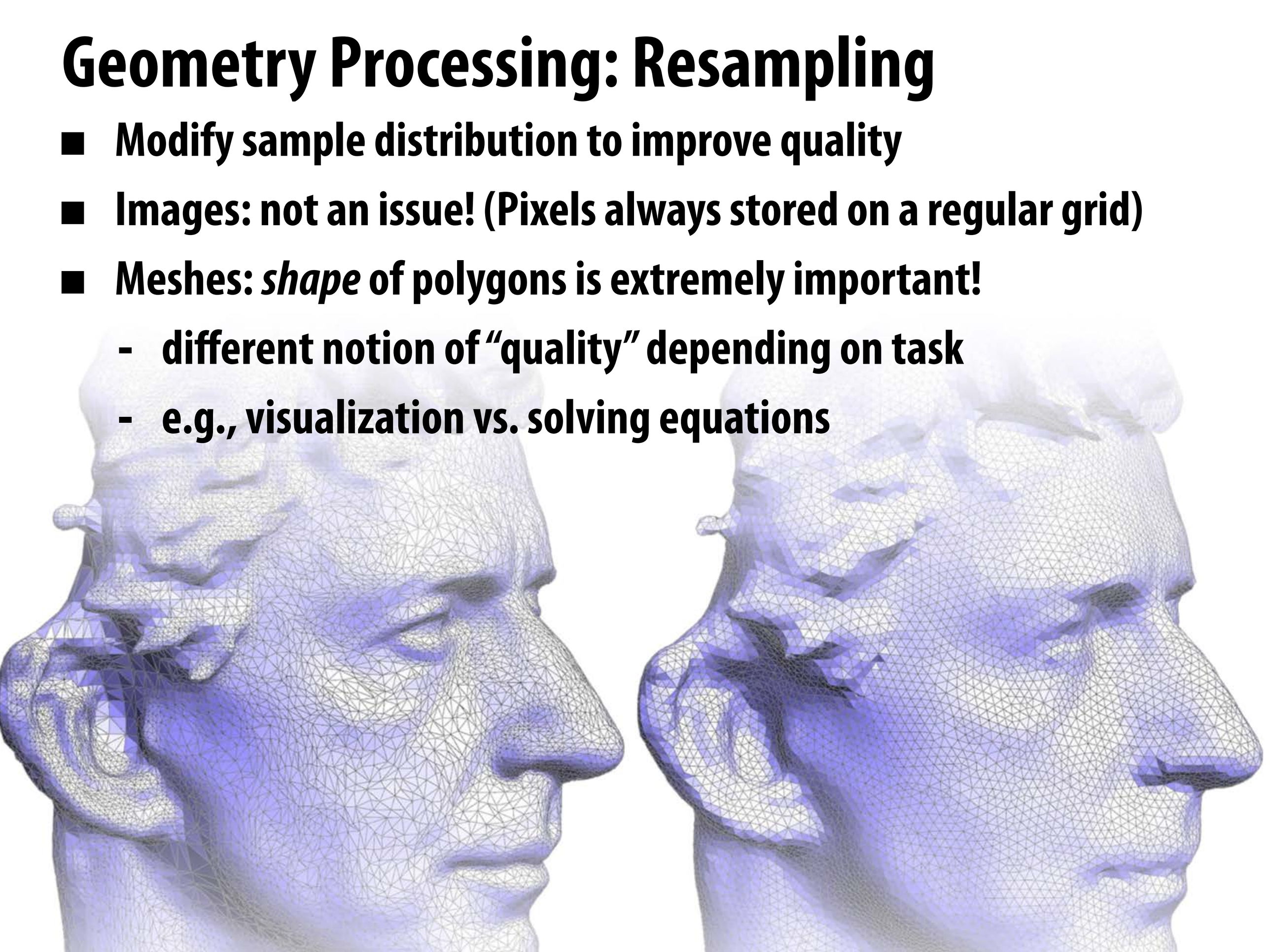
Geometry Processing: Downsampling

- 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, ...



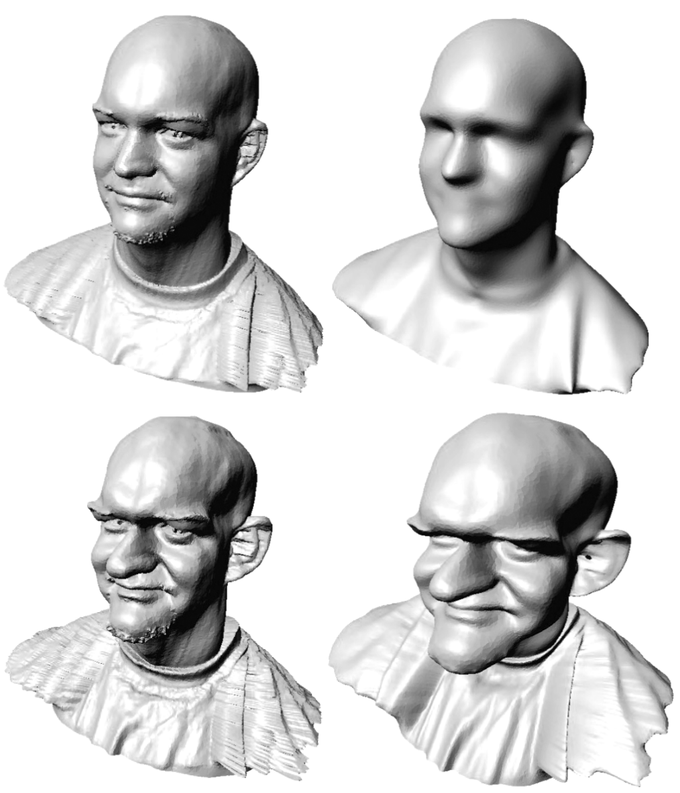
Geometry Processing: Resampling

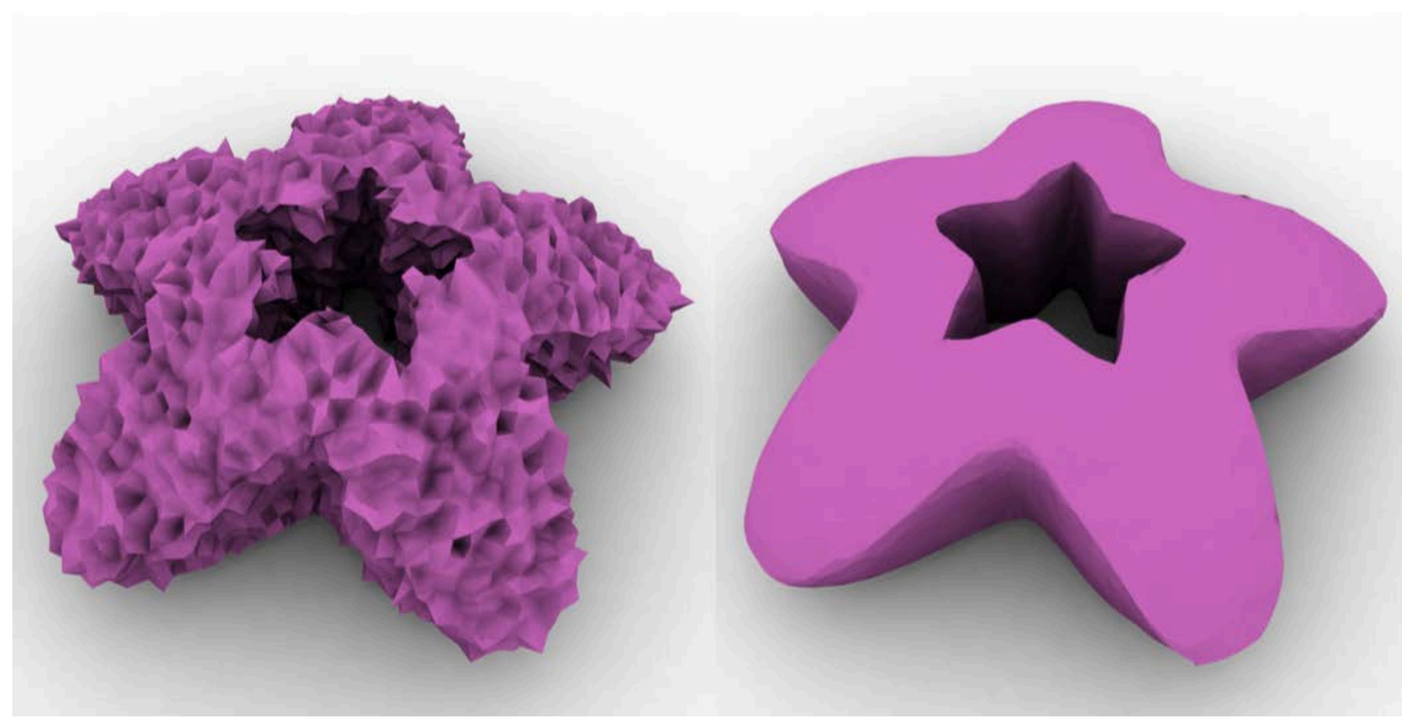
- Modify sample distribution to improve quality
- Images: not an issue! (Pixels always stored on a regular grid)
- Meshes: *shape* of polygons is extremely important!
  - different notion of "quality" depending on task
  - e.g., visualization vs. solving equations



Geometry Processing: Filtering

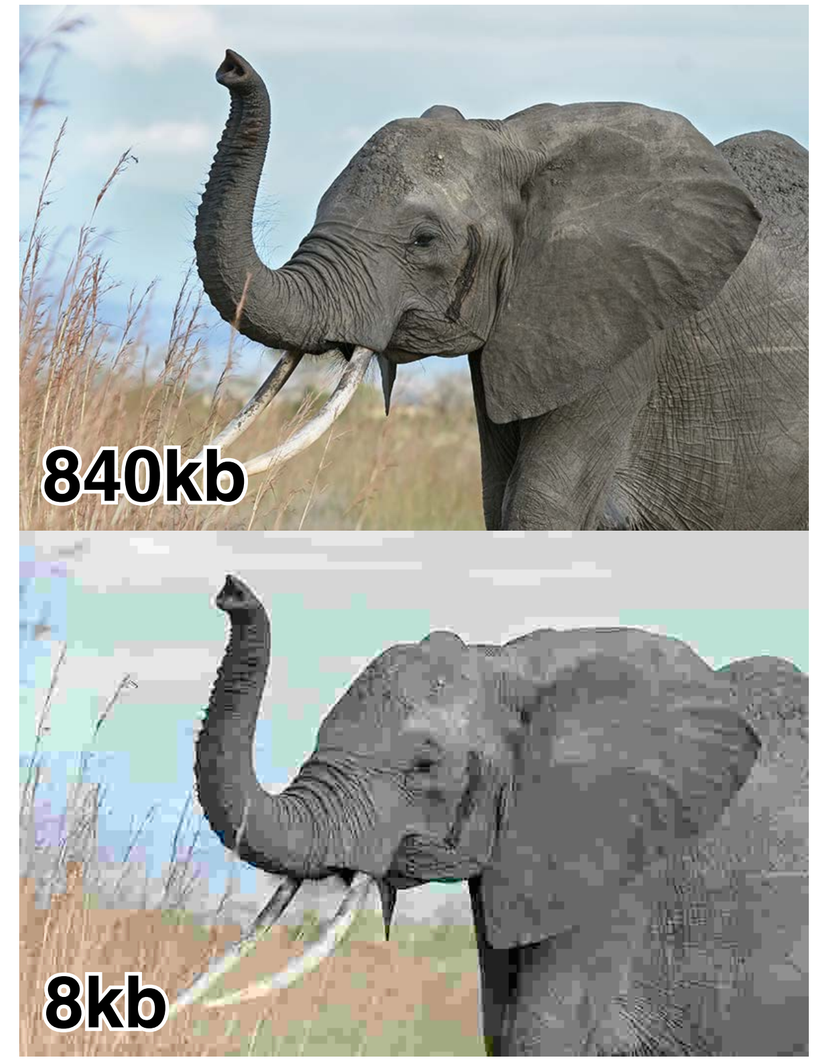
- Remove noise, or emphasize important features (e.g., edges)
- Images: blurring, bilateral filter, edge detection, ...
- Polygon meshes:
  - curvature flow
  - bilateral filter
  - spectral filter

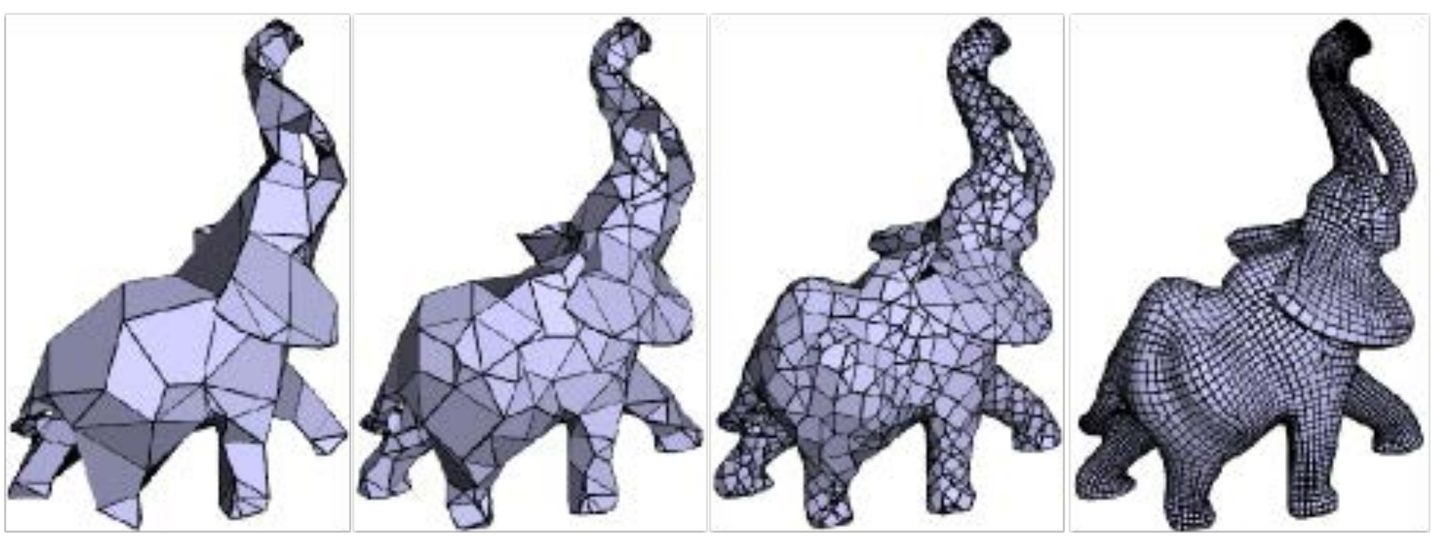




Geometry Processing: Compression

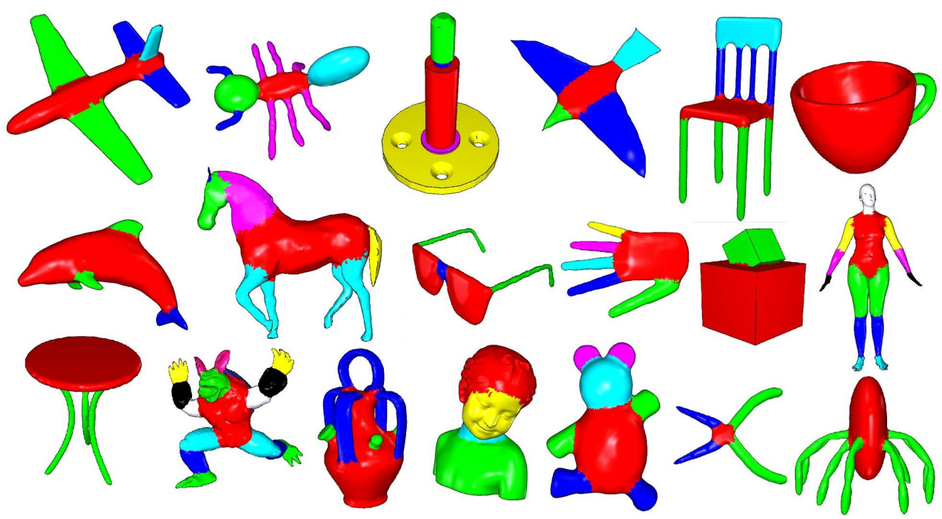
- Reduce storage size by eliminating redundant data/ approximating unimportant data
- Images:
  - run-length, Huffman coding - *lossless*
  - cosine/wavelet (JPEG/MPEG) - *lossy*
- Polygon meshes:
  - compress geometry *and* connectivity
  - many techniques (lossy & lossless)

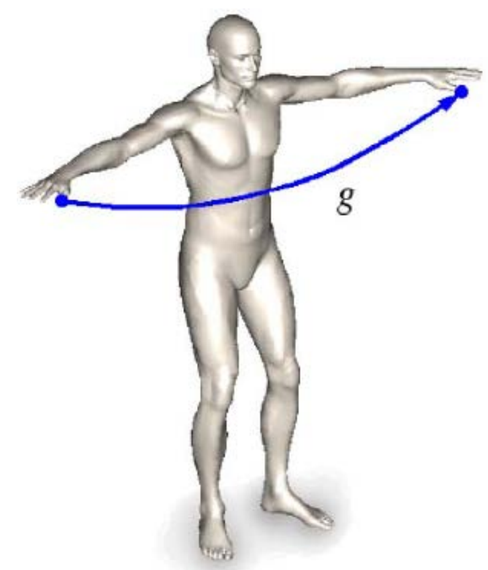




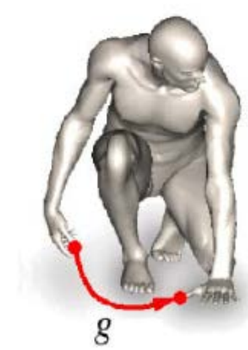
Geometry Processing: Shape Analysis

- Identify/understand important semantic features
- Images: computer vision, segmentation, face detection, ...
- Polygon meshes:
  - segmentation, correspondence, symmetry detection, ...

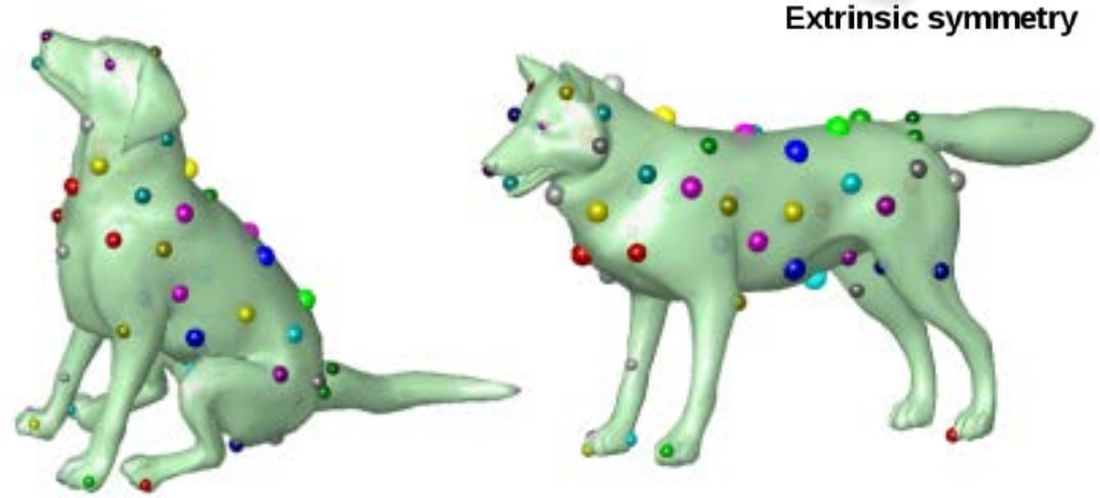




Extrinsic symmetry



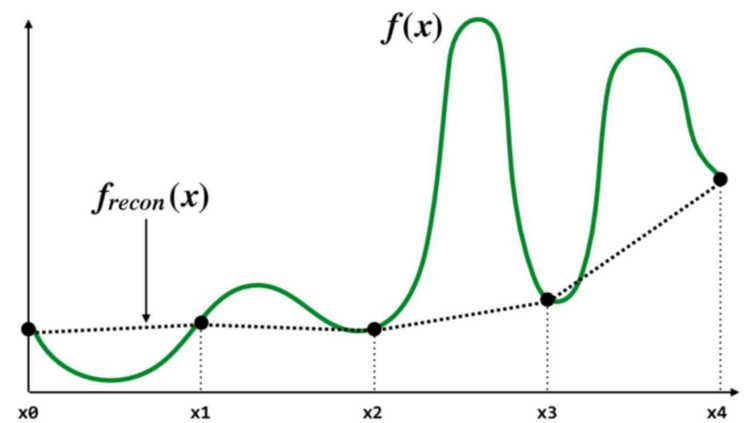
Intrinsic symmetry

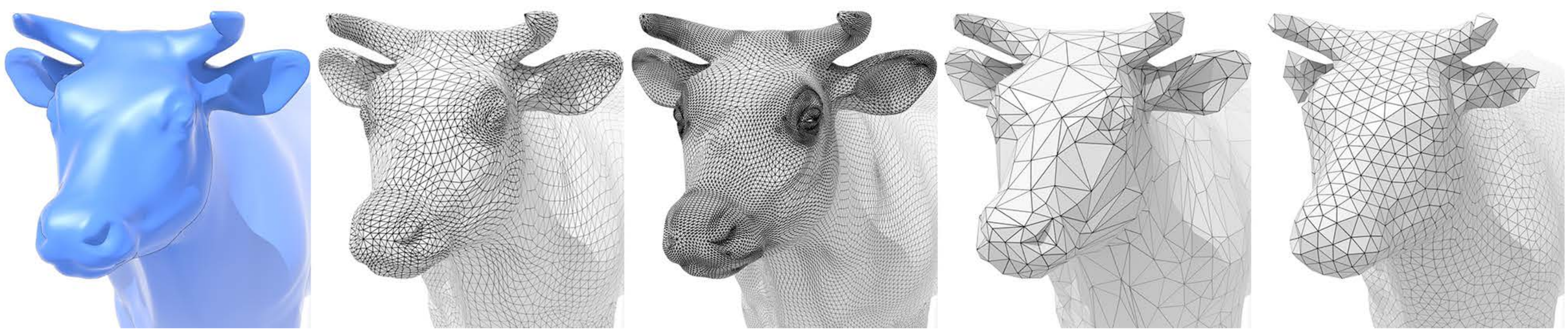


Enough overview—
Let’s process some geometry!

Remeshing as resampling

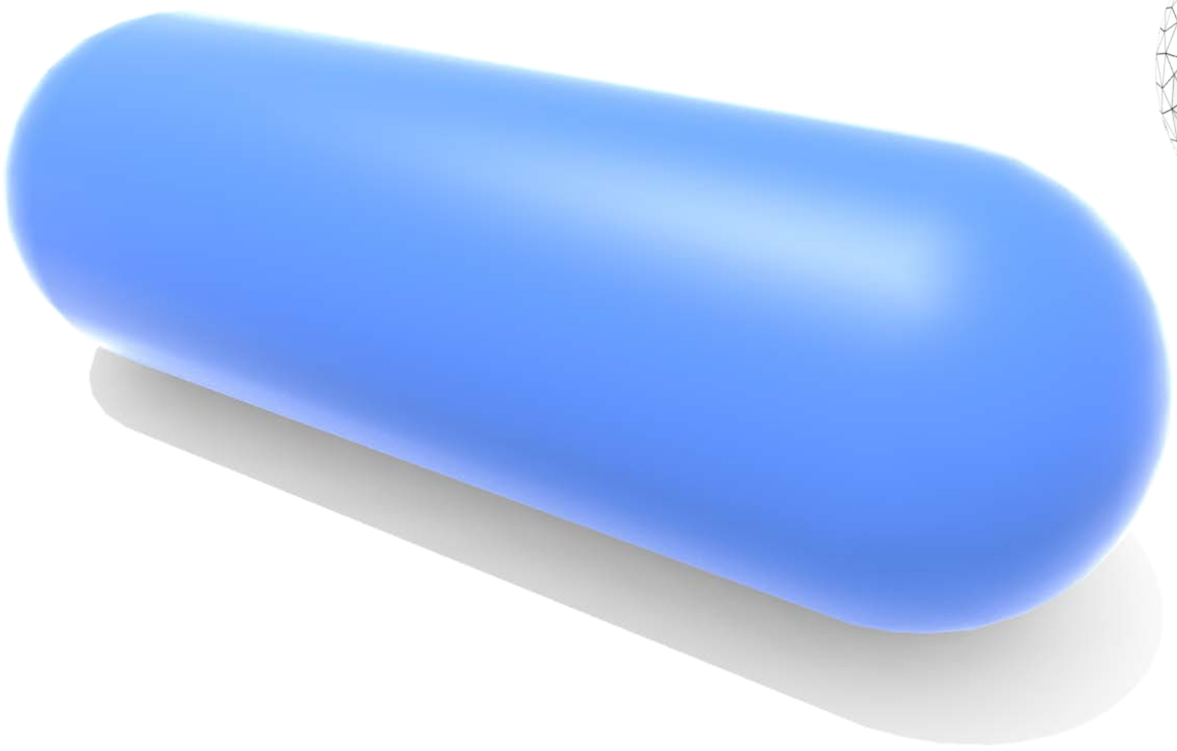
- Remember our discussion of *aliasing*
- Bad sampling makes signal appear different than it really is
- E.g., undersampled curve looks flat
- Geometry is no different!
  - undersampling destroys features
  - oversampling bad for performance

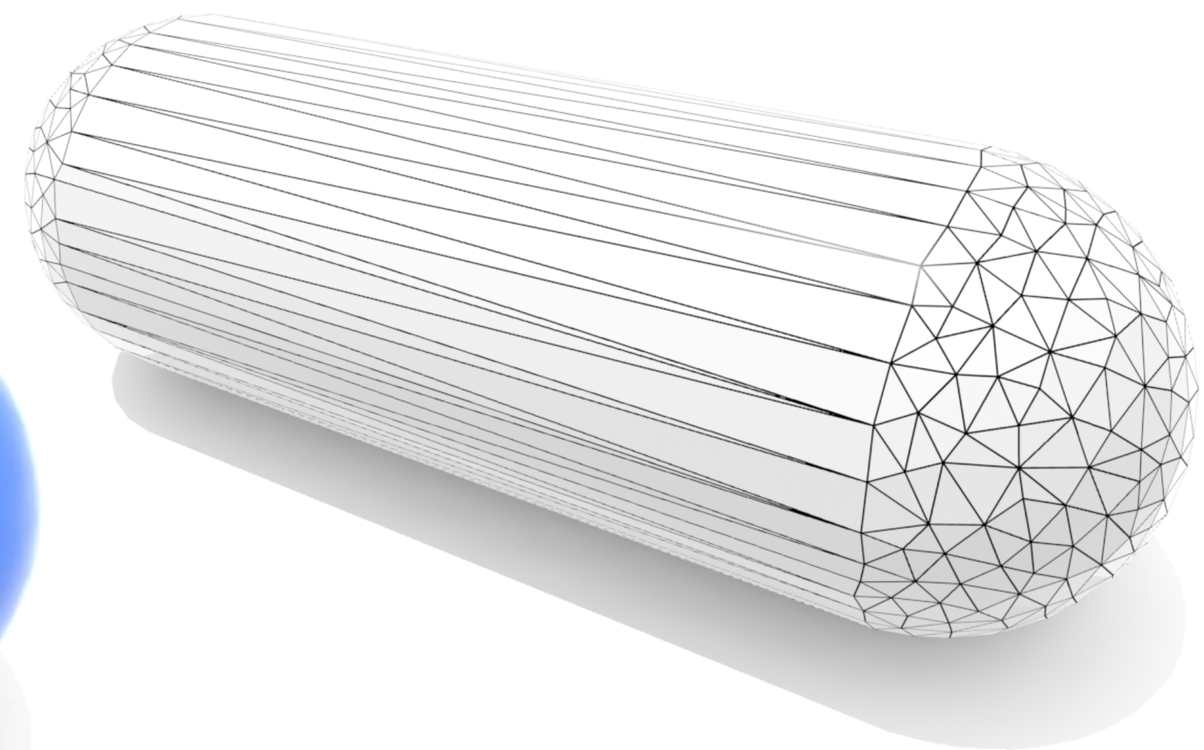




What makes a "good" mesh?

- One idea: good approximation of original shape!
- Keep only elements that contribute *information* about shape
- Add additional information where, e.g., *curvature* is large

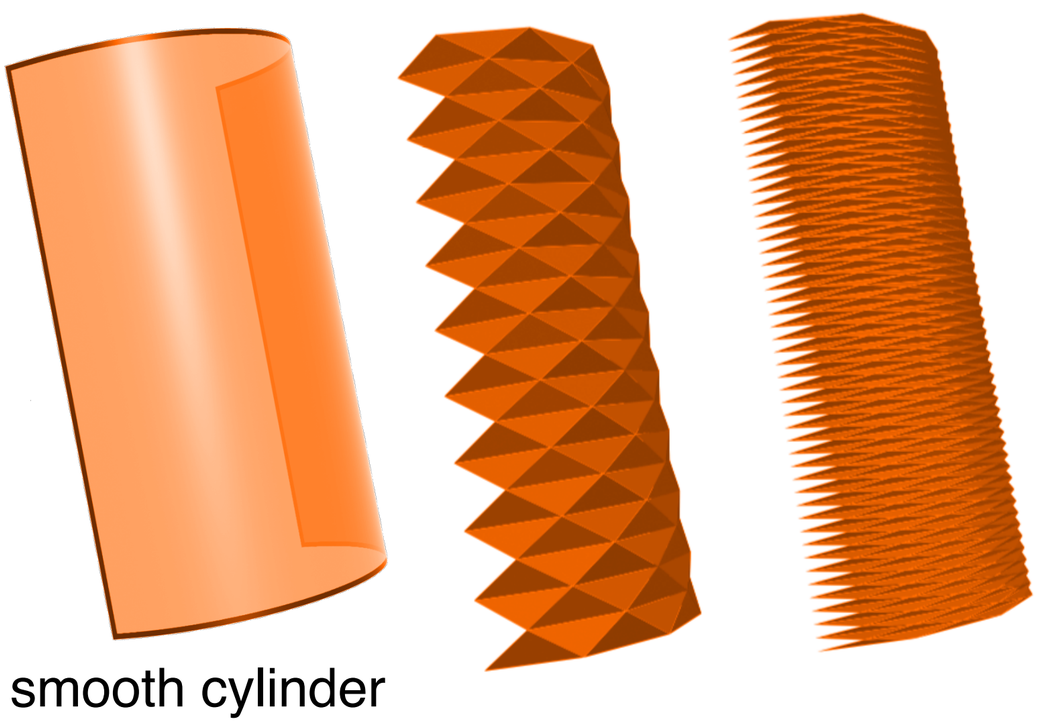




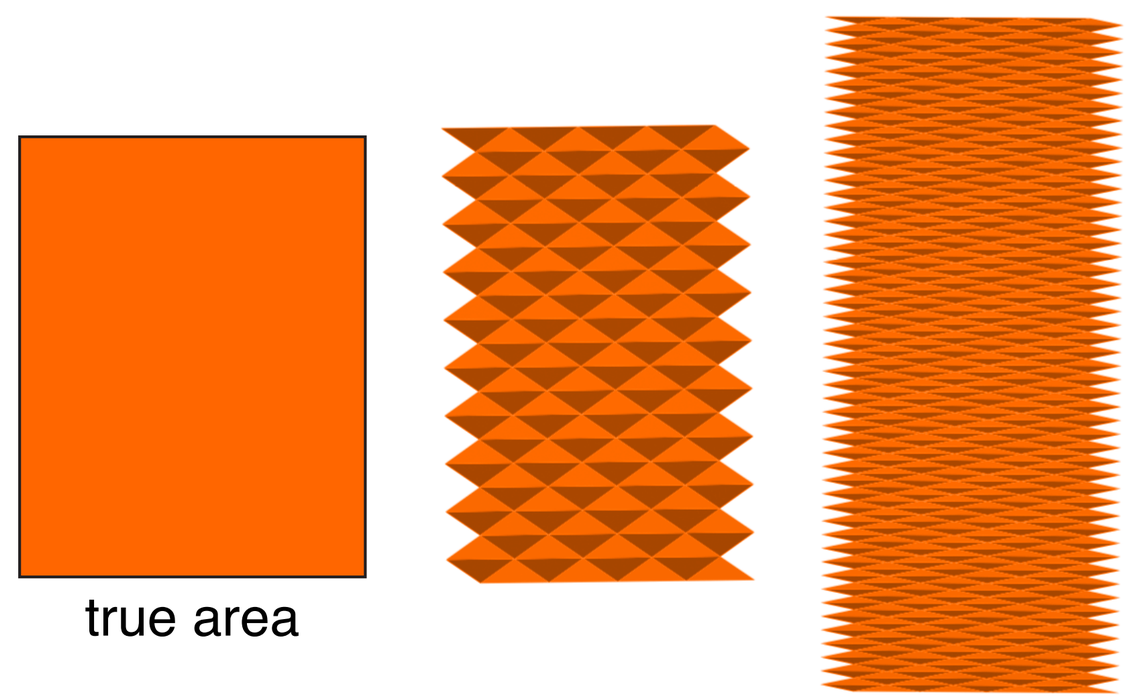
Approximation of position is not enough!

- **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 <u>exactly</u> on smooth cylinder



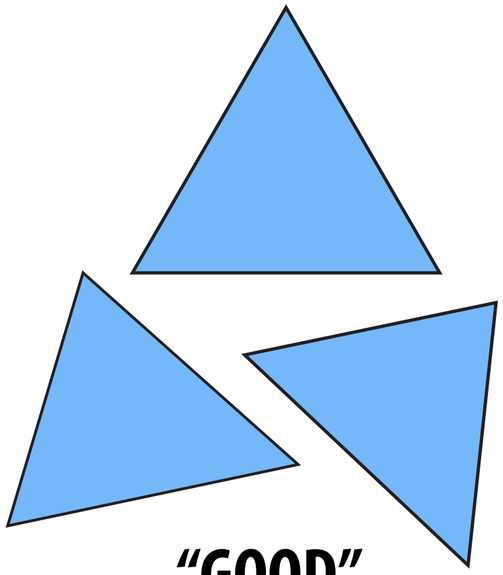
flattening of smooth cylinder & meshes



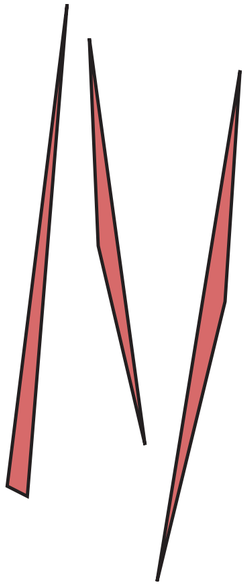
*See Hildebrandt et al (2007), "On the convergence of metric and geometric properties of polyhedral surfaces"

What else makes a "good" triangle mesh?

- Another rule of thumb: *triangle shape*



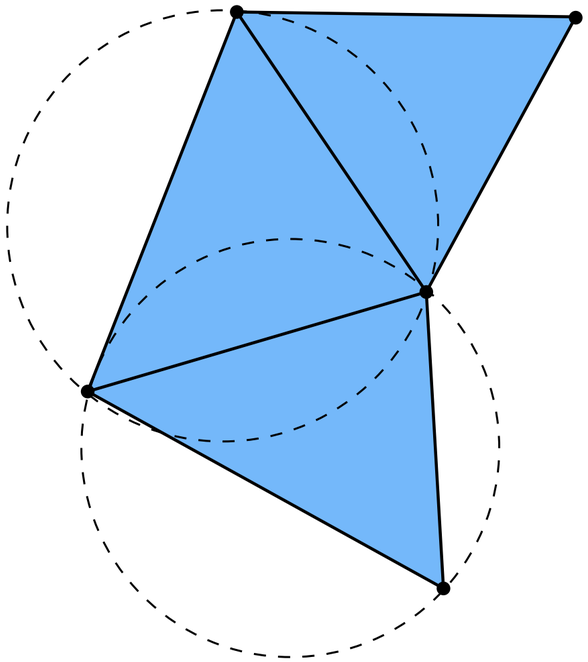
"GOOD"



"BAD"

pronunciation:

DELAUNAY

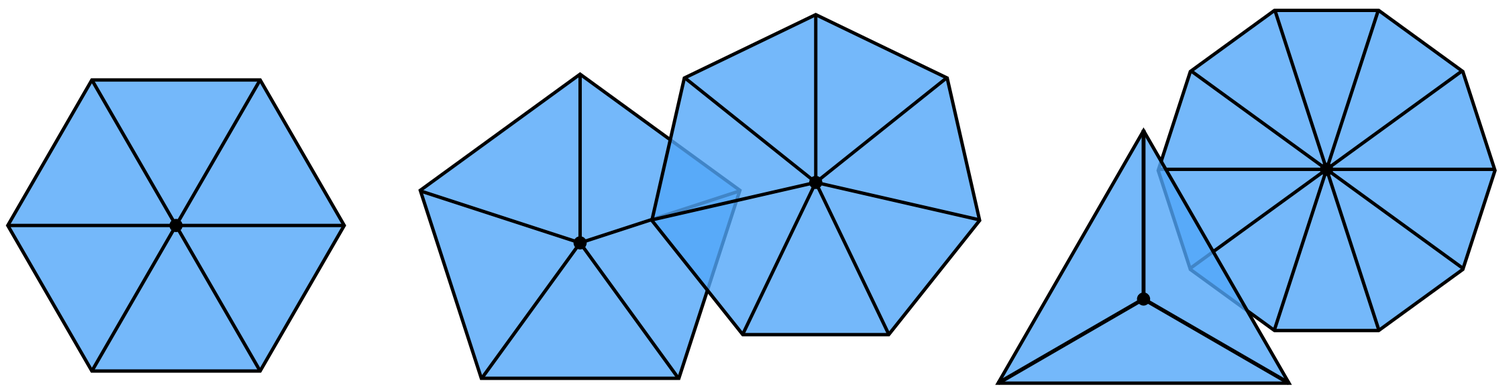


- E.g., all angles close to 60 degrees
- More sophisticated condition: *Delaunay* (empty circumcircles)
  - often helps with numerical accuracy/stability
  - coincides with <u>shockingly</u> many other desirable properties
    (maximizes minimum angle, provides smoothest interpolation, guarantees maximum principle…)
- Tradeoffs w/ good geometric approximation*
  - e.g., long & skinny might be "more efficient"

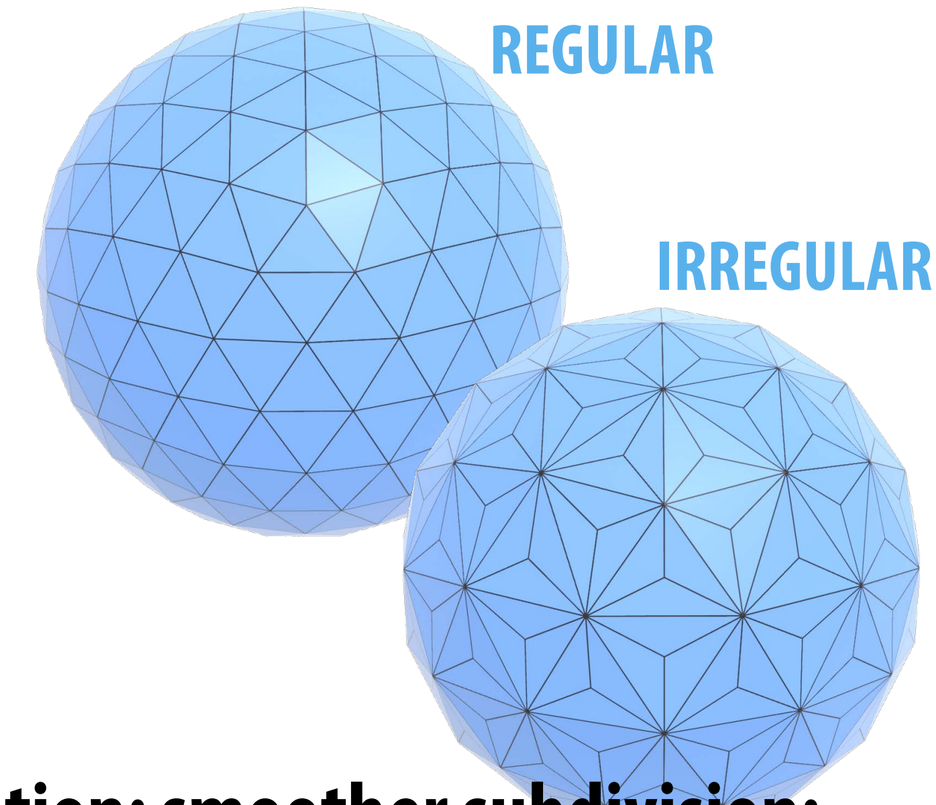
*see Shewchuk, "What is a Good Linear Element"

What else constitutes a "good" mesh?

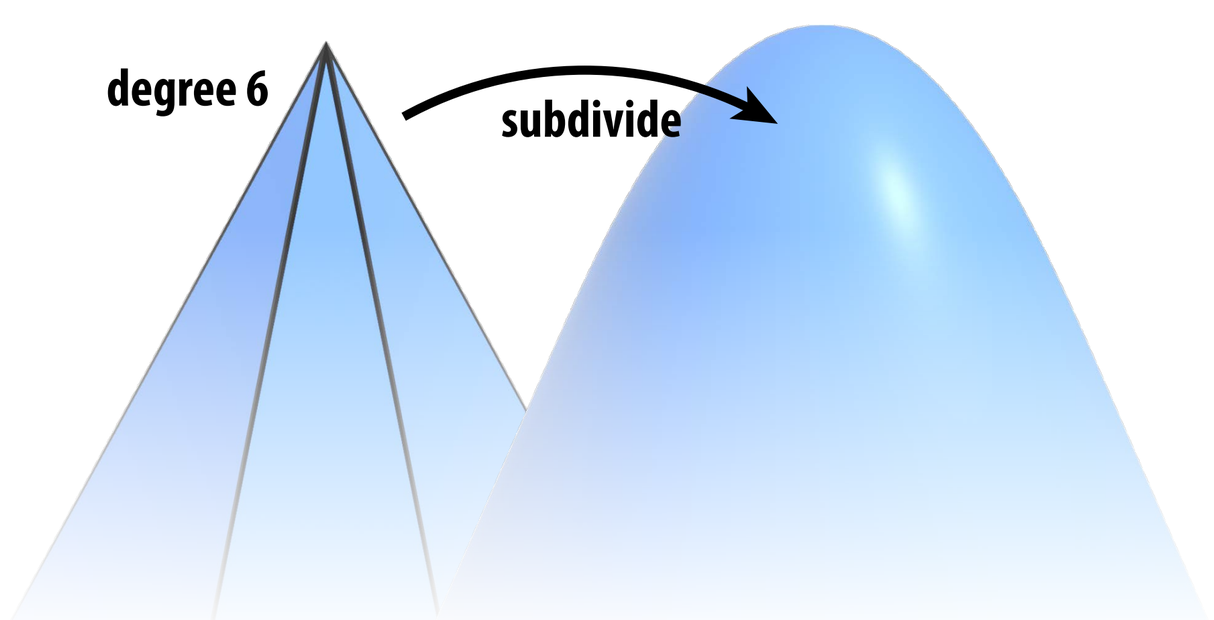
- Another rule of thumb: *regular vertex degree*
- Degree 6 for triangle mesh, 4 for quad mesh

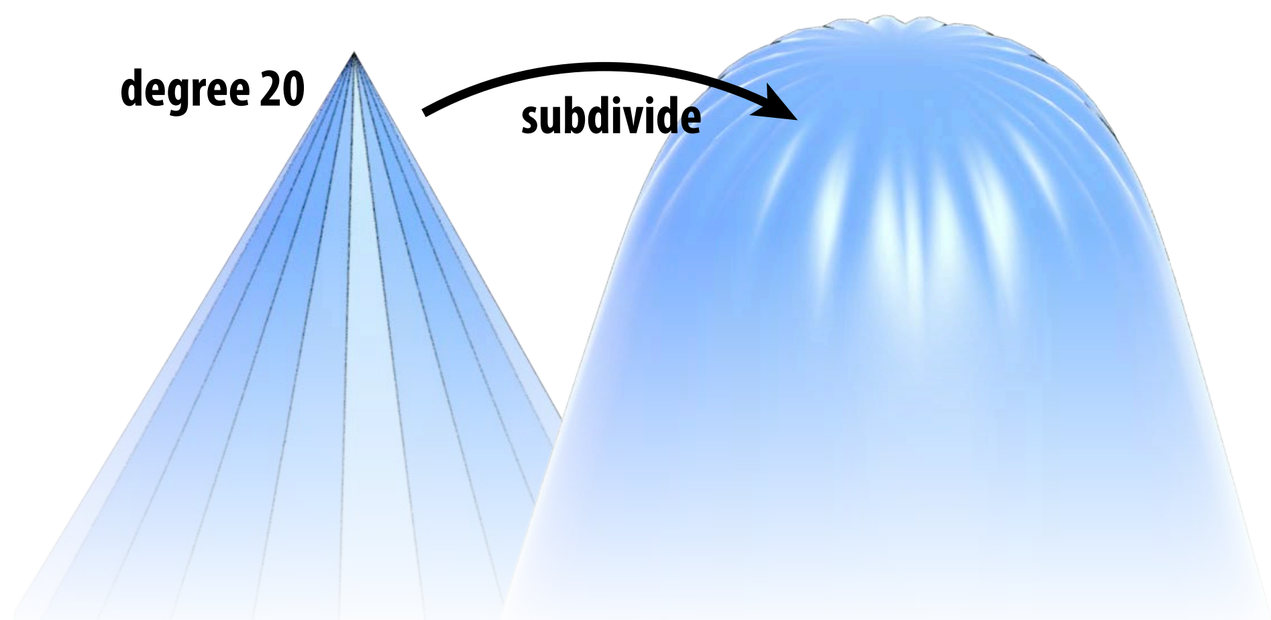


"GOOD" "OK" "BAD"



Why? Better polygon shape; more regular computation; smoother subdivision:

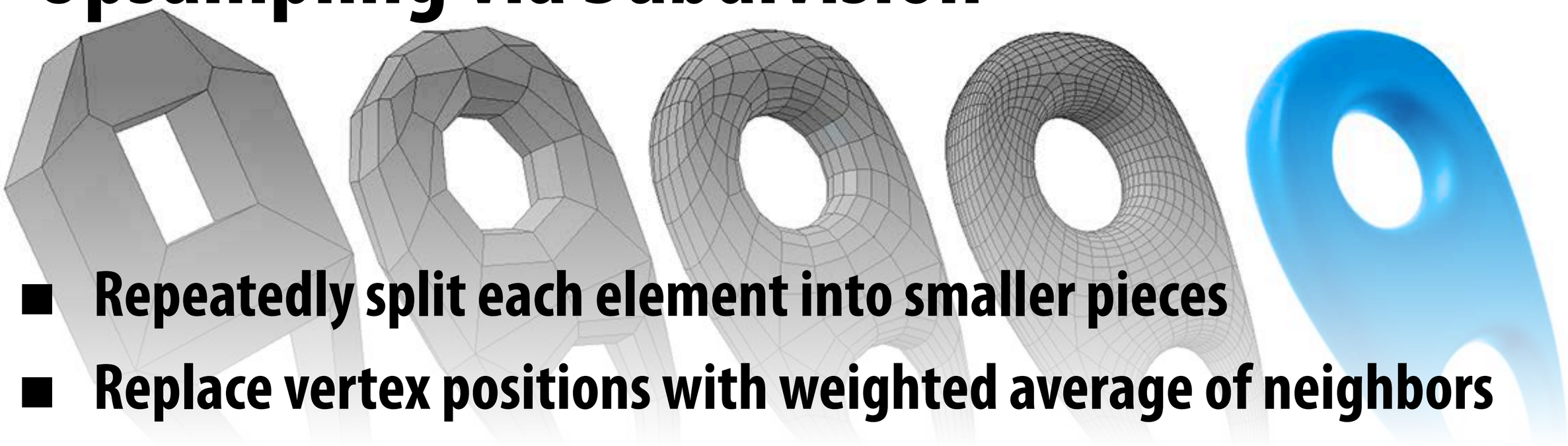




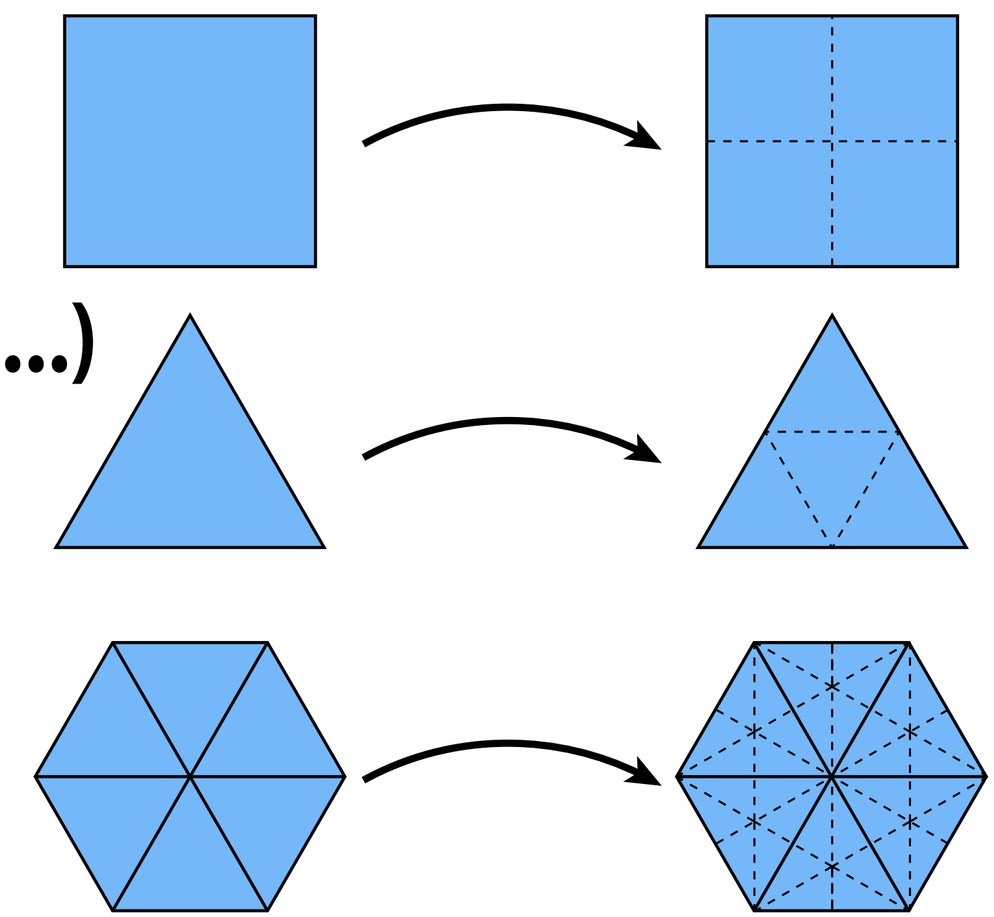
Fact: in general, can't have regular vertex degree everywhere!

How do we upsample a mesh?

Upsampling via Subdivision

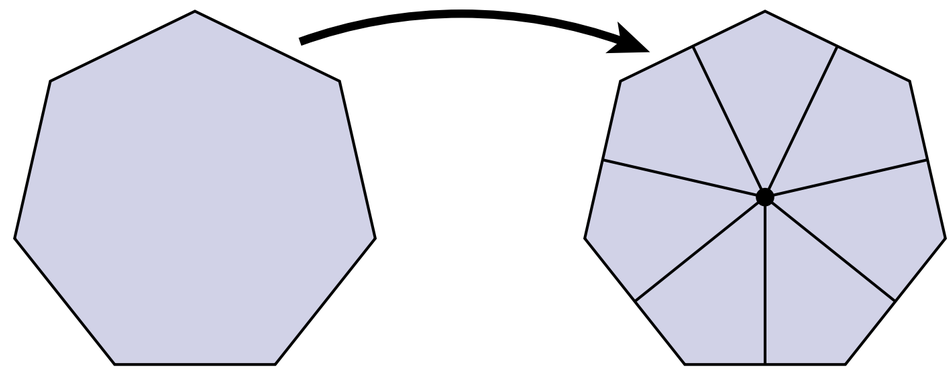


- Repeatedly split each element into smaller pieces
- Replace vertex positions with weighted average of neighbors
- Main considerations:
  - interpolating vs. approximating
  - limit surface continuity ($C^1, C^2, \ldots$)
  - behavior at irregular vertices
- Many options:
  - Quad: Catmull-Clark
  - Triangle: Loop, Butterfly, Sqrt(3)



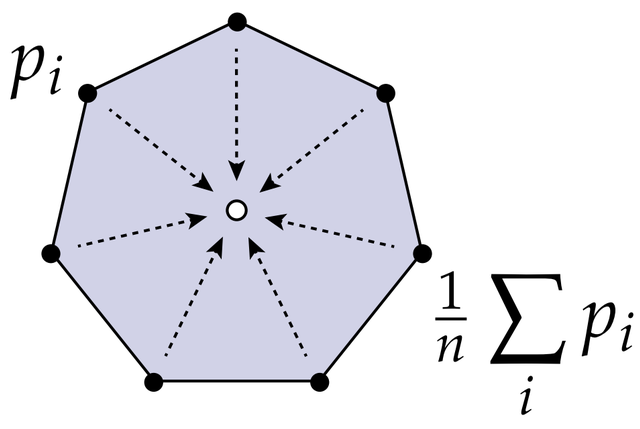
Catmull-Clark Subdivision

- **Step 0: split every polygon (any # of sides) into quadrilaterals:**

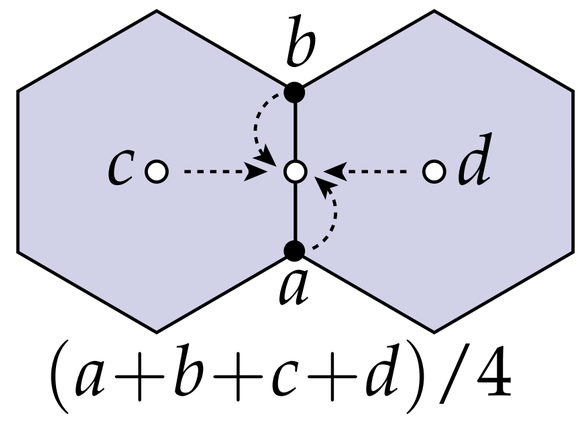


- **New vertex positions are weighted combination of old ones:**

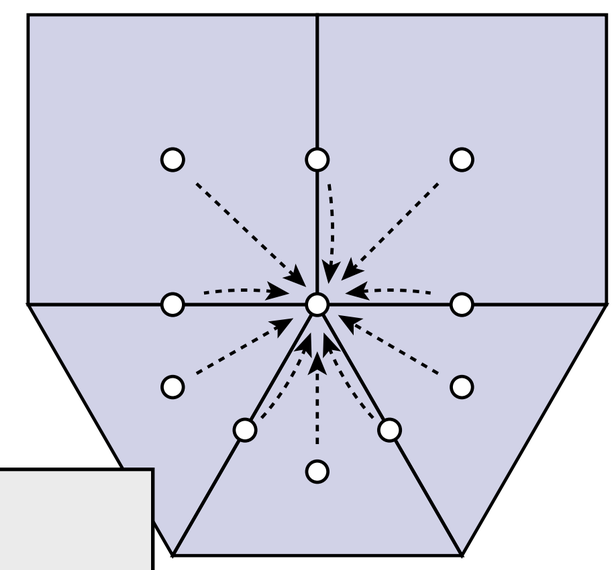
STEP 1: Face coords



STEP 2: Edge coords



STEP 3: Vertex coords



New vertex coords:

$$\frac{Q + 2R + (n-3)S}{n}$$

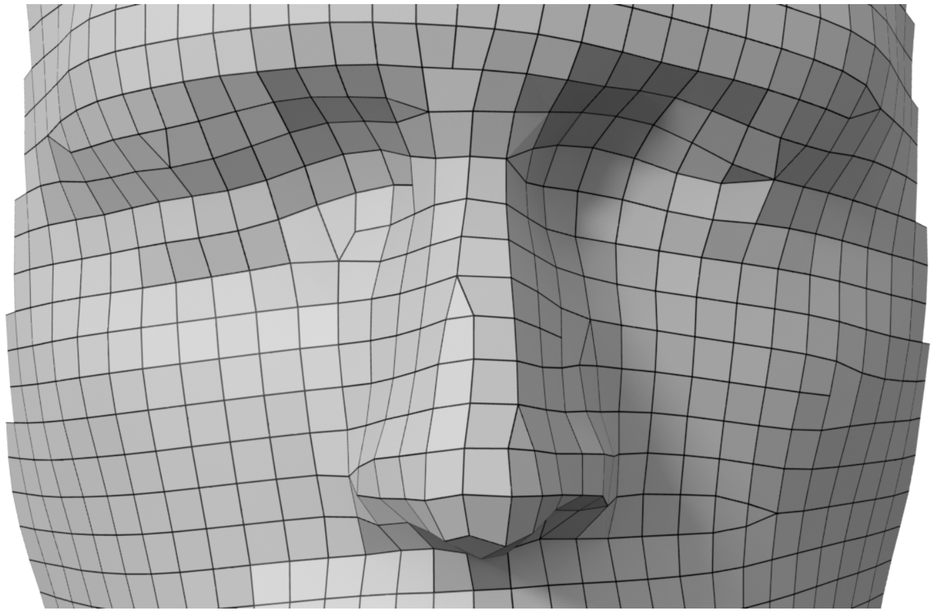
n – vertex degree

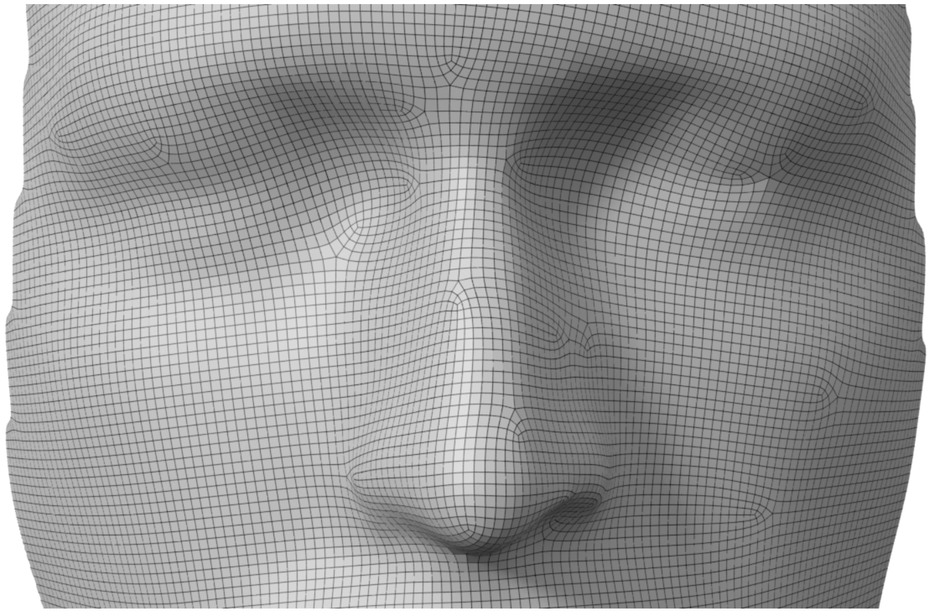
Q – average of face coords around vertex

R – average of edge coords around vertex

S – original vertex position

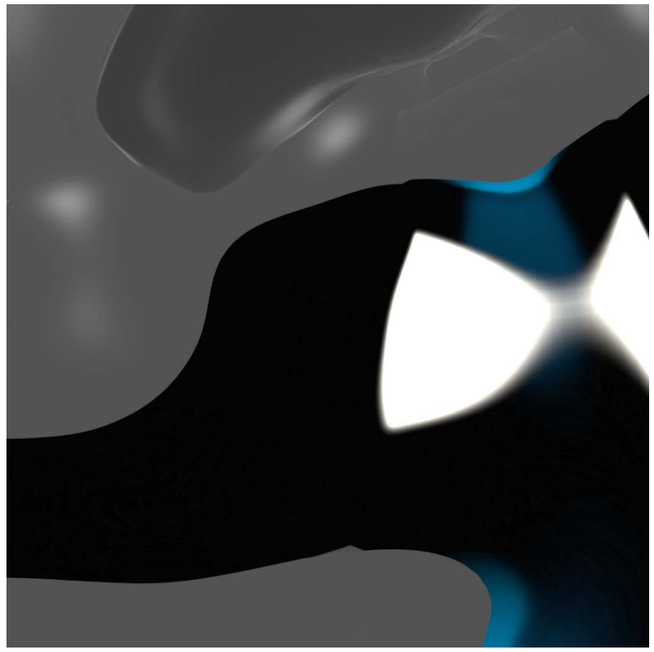
Catmull-Clark on quad mesh



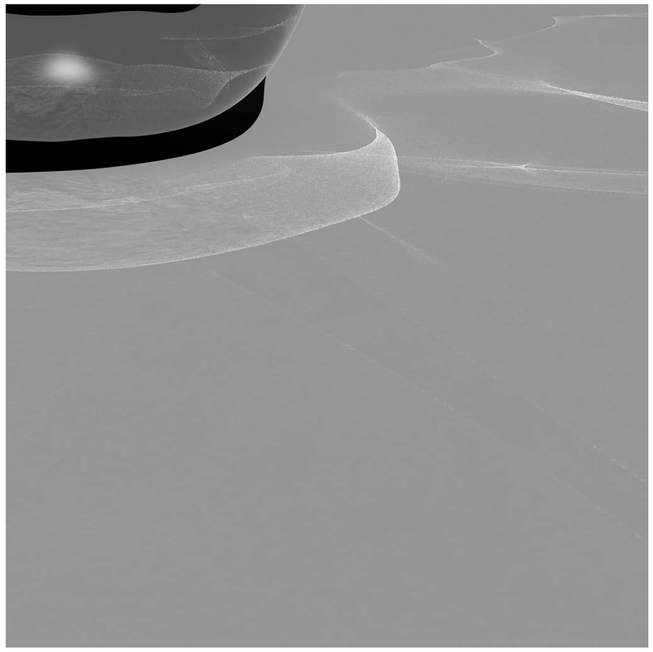


few irregular vertices

$\Longrightarrow$ smoothly-varying surface normals

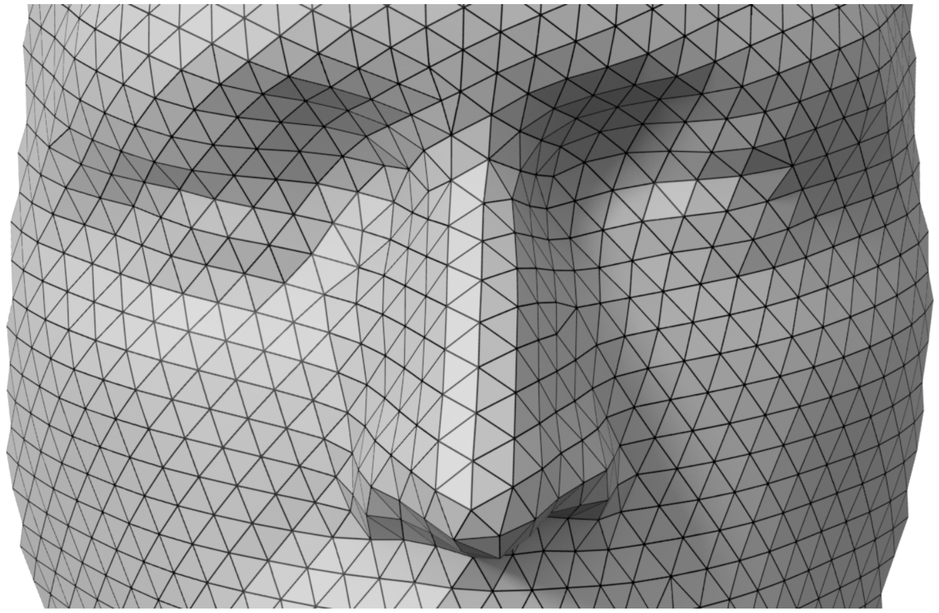


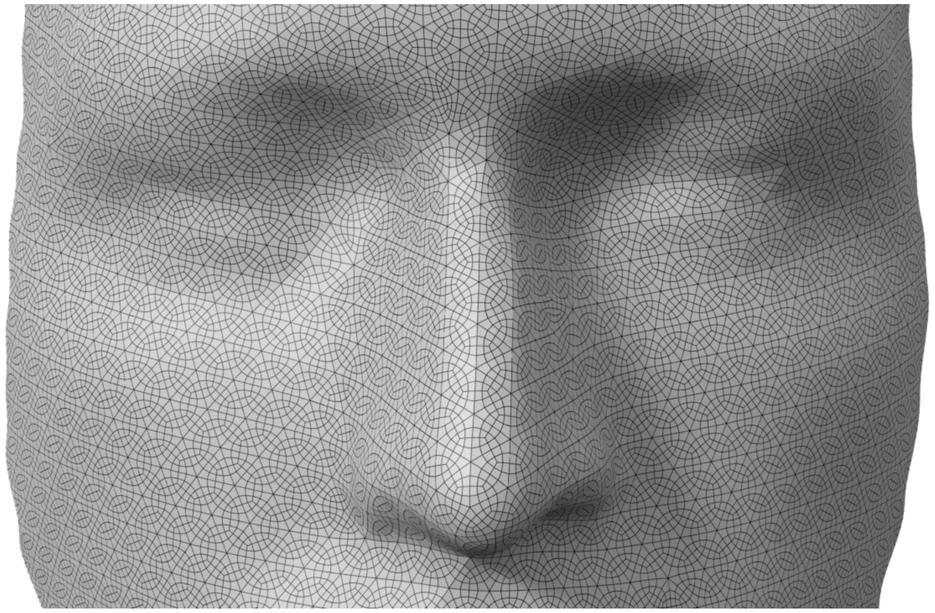
smooth
reflection lines



smooth
caustics

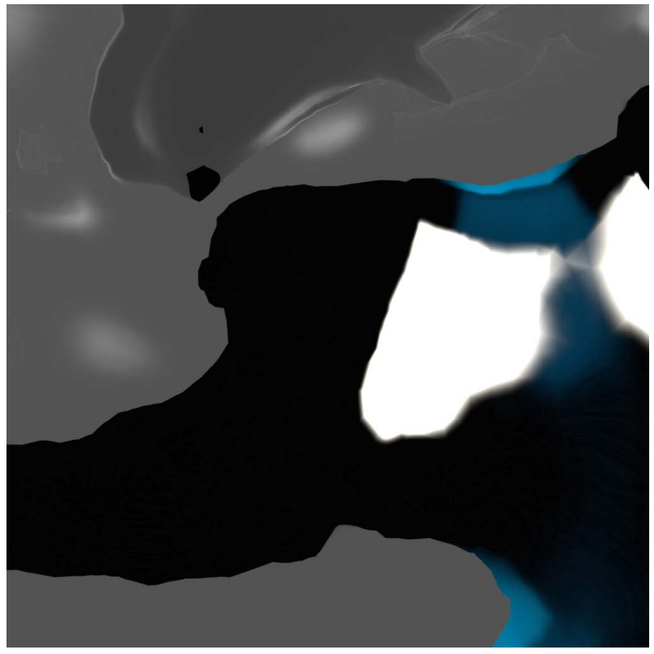
Catmull-Clark on triangle mesh



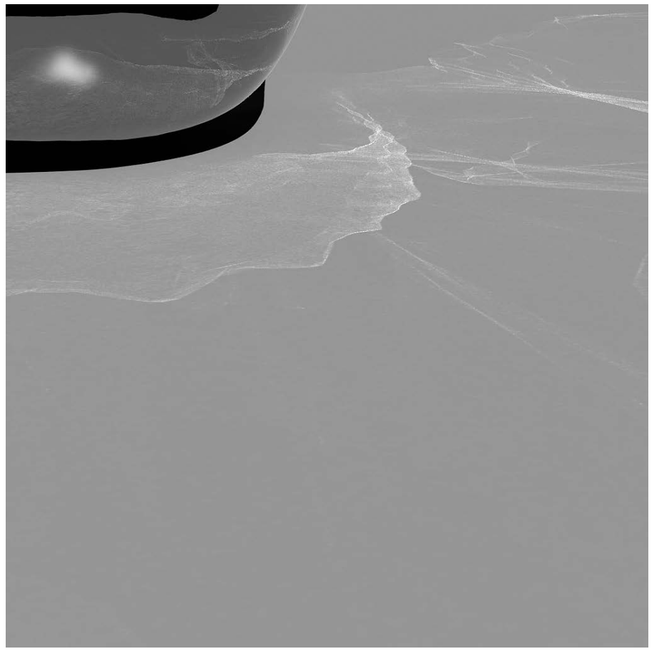


many irregular vertices

$\Longrightarrow$ erratic surface normals



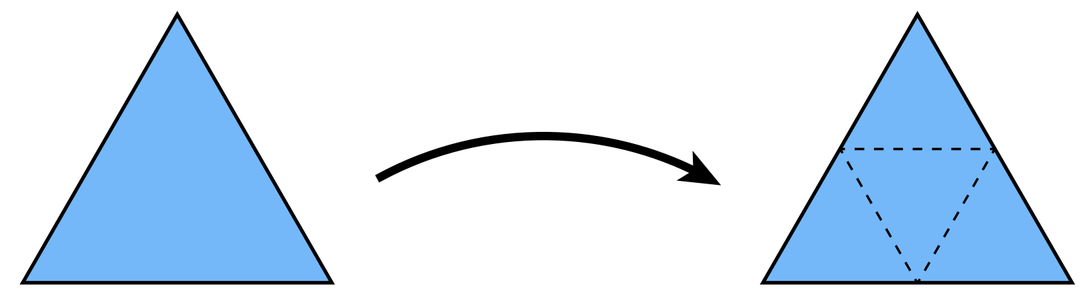
jagged
reflection lines



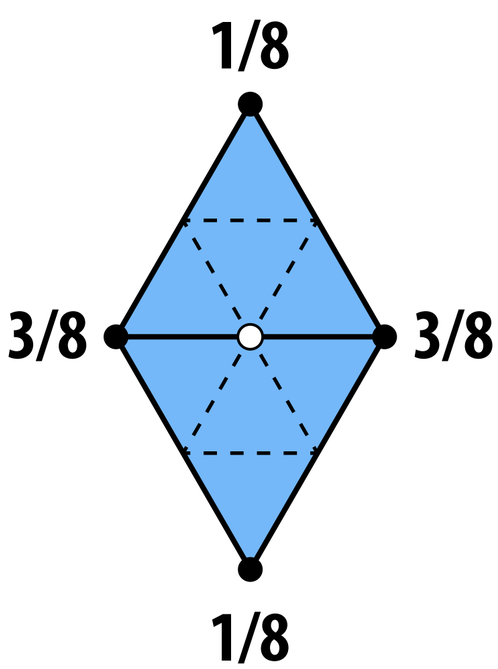
jagged
caustics

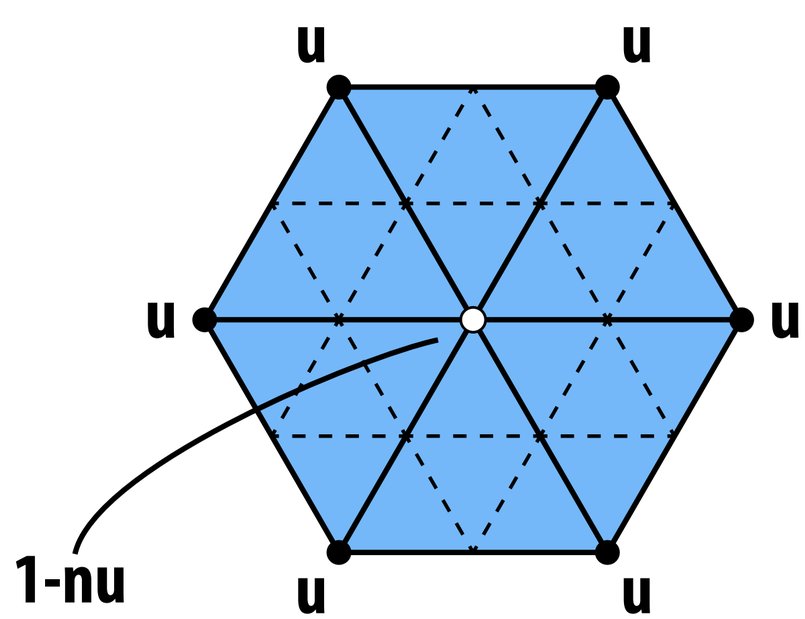
Loop Subdivision

- Alternative subdivision scheme for triangle meshes
- Curvature is continuous away from irregular vertices ("C^2")
- Algorithm:
  - Split each triangle into four



  - Assign new vertex positions according to weights:



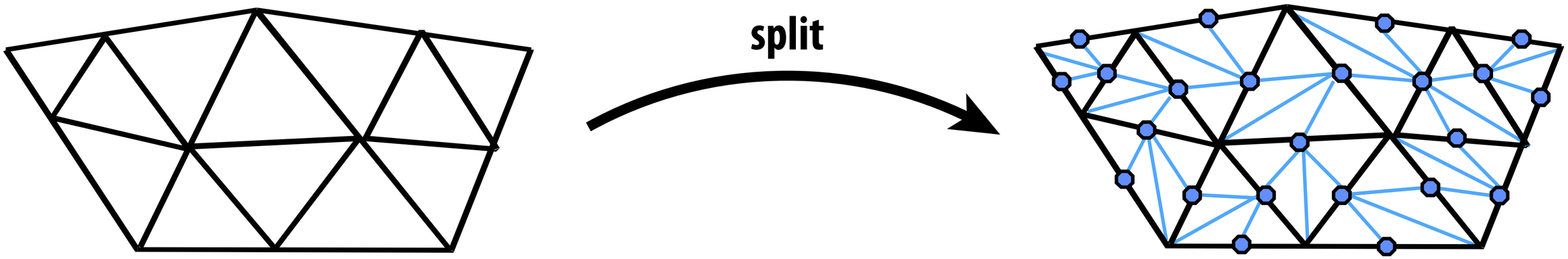


n: vertex degree

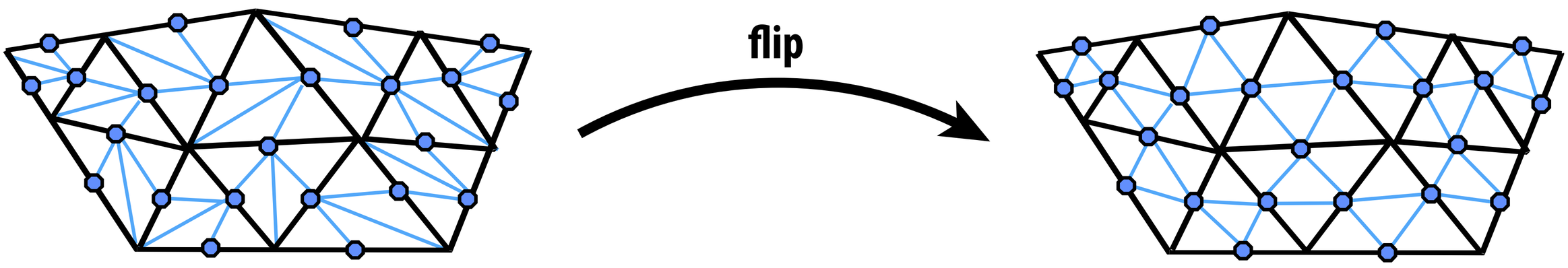
u: 3/16 if n=3, 3/(8n) otherwise

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!)

Images cribbed from Denis Zorin.

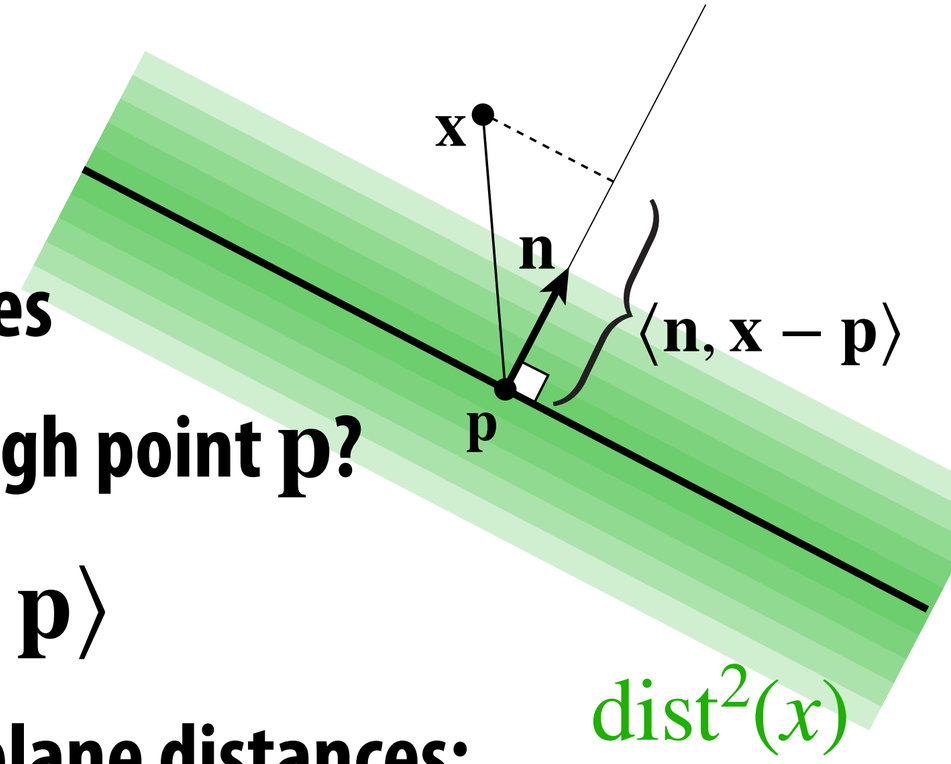
What if we want *fewer* triangles?

Simplification via Edge Collapse

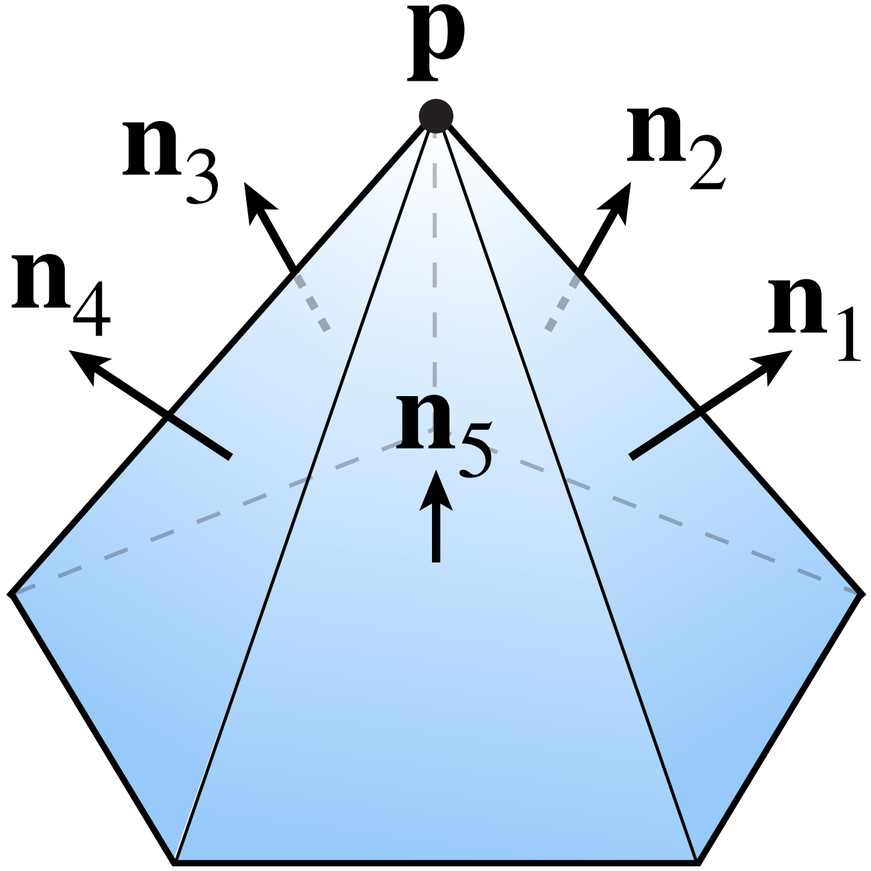
- One popular scheme: iteratively collapse edges
- Greedy algorithm:
  - assign each edge a cost
  - collapse edge with least cost
  - repeat until target number of elements is reached
- Particularly effective cost function: *quadric error metric**

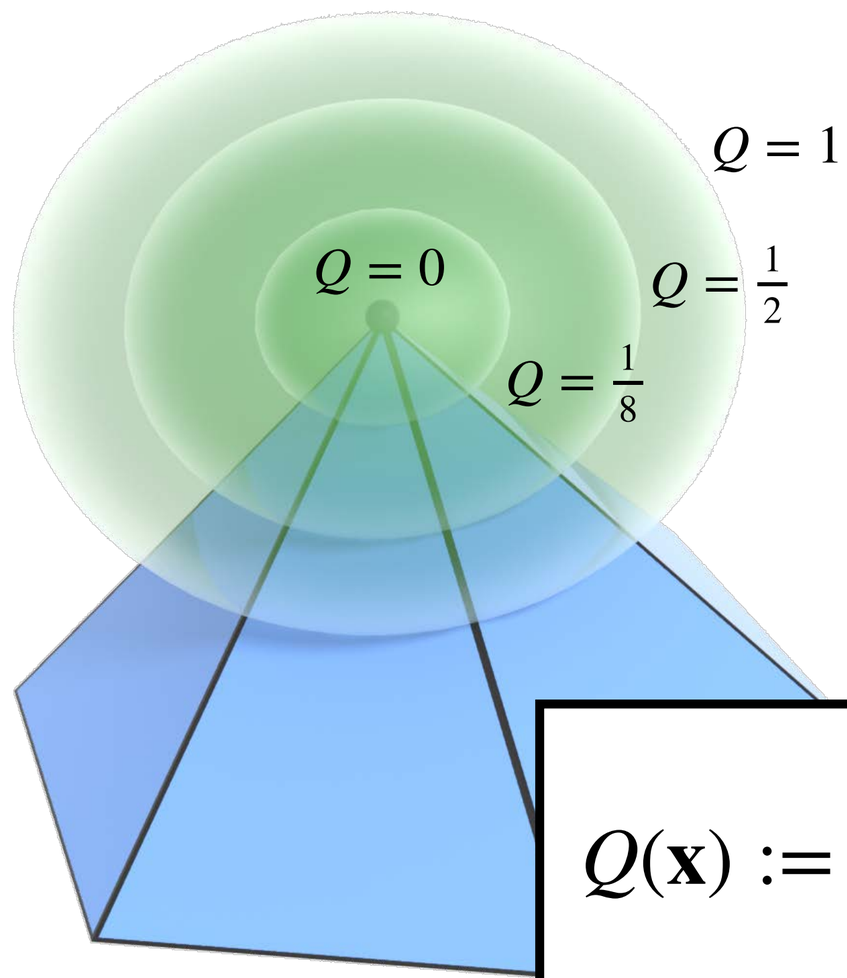
*invented at CMU (Garland & Heckbert 1997)

Quadric Error Metric



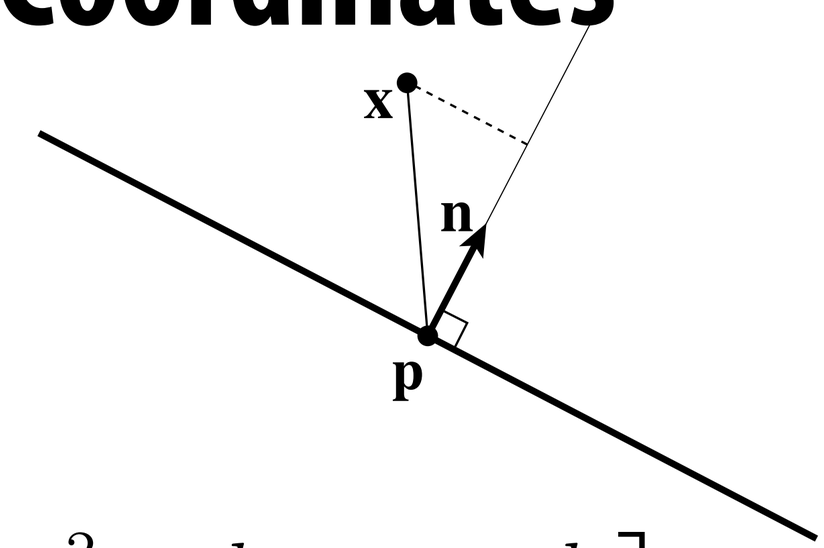
- Approximate distance to a collection of triangles
- Q: Distance to plane w/ normal $\mathbf{n}$ passing through point $\mathbf{p}$?
- A: $\mathrm{dist}(\mathbf{x}) = \langle \mathbf{n}, \mathbf{x} \rangle - \langle \mathbf{n}, \mathbf{p} \rangle = \langle \mathbf{n}, \mathbf{x} - \mathbf{p} \rangle$
- 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
  - $\mathbf{u} := (x, y, z, 1)$
  - $\mathbf{v} := (a, b, c, d)$

$$K = \begin{bmatrix} a^2 & ab & ac & ad \\ ab & b^2 & bc & bd \\ ac & bc & c^2 & cd \\ ad & bd & cd & d^2 \end{bmatrix}$$

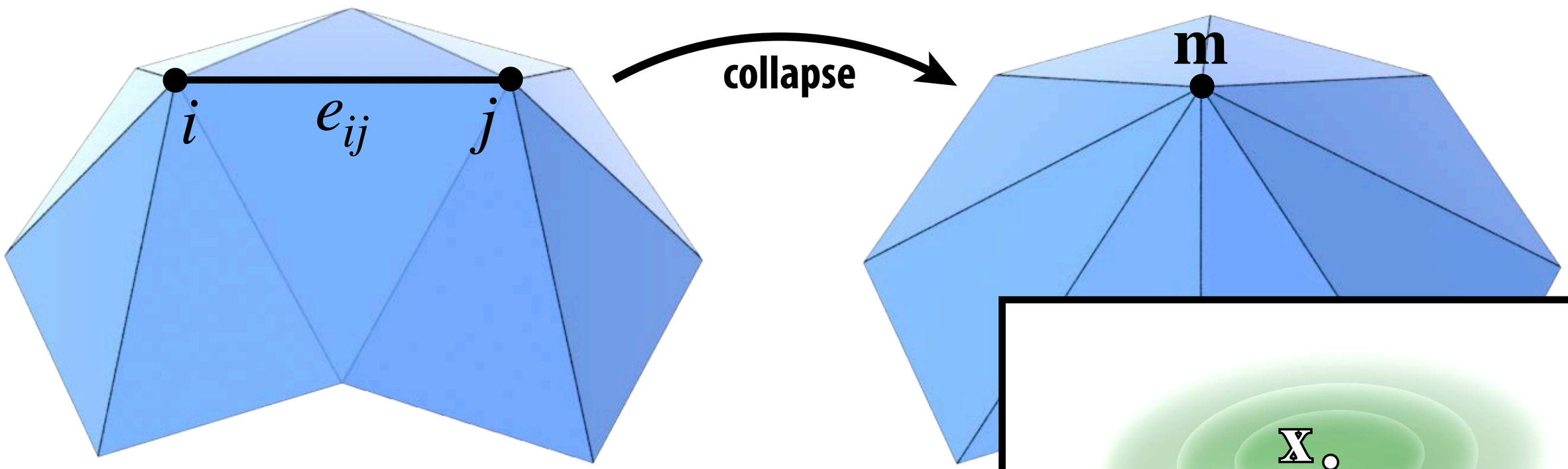
- Signed distance to plane is then just $\langle \mathbf{u}, \mathbf{v} \rangle = ax + by + cz + d$
- Squared distance is $\langle \mathbf{u}, \mathbf{v} \rangle^2 = \mathbf{u}^\top (\mathbf{v}\mathbf{v}^\top)\mathbf{u} =: \mathbf{u}^\top K \mathbf{u}$
- Matrix $K = \mathbf{v}\mathbf{v}^T$ encodes squared distance to plane

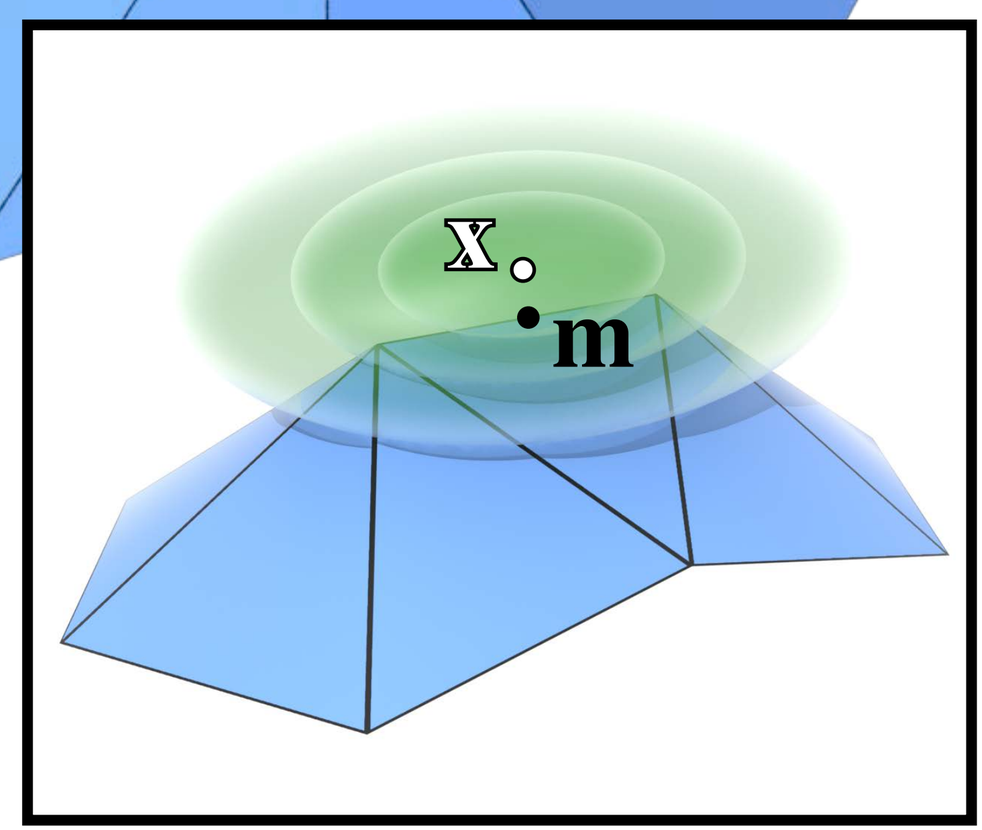
Key idea: <u>sum</u> of matrices $K \Longleftrightarrow$ distance to <u>union</u> of planes

$$\mathbf{u}^\top K_1 \mathbf{u} + \mathbf{u}^\top K_2 \mathbf{u} = \mathbf{u}^\top (K_1 + K_2)\mathbf{u}$$

Quadric Error of Edge Collapse

- How much does it cost to collapse an edge e_{ij}?
- Idea: compute midpoint $\mathbf{m}$, measure error $Q(\mathbf{m}) = \mathbf{m}^\mathsf{T}(K_i + K_j)\mathbf{m}$
- Error becomes "score" for e_{ij}, determining priority

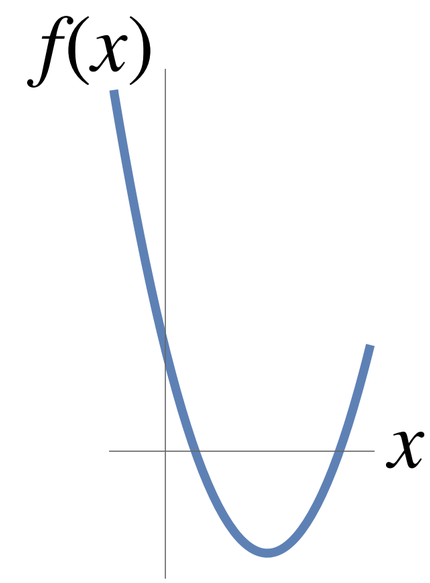


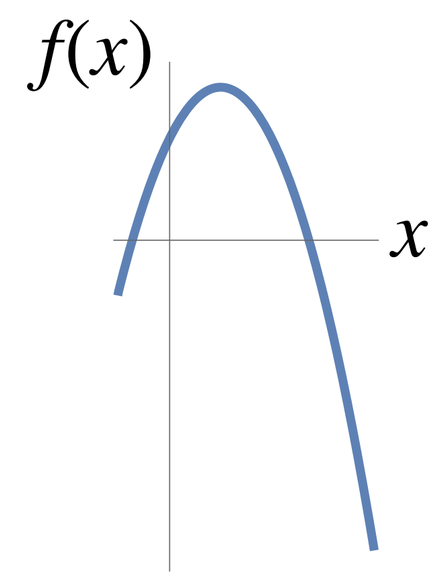


- Better idea: find point $\mathbf{x}$ that *minimizes error*!
- Ok, but how do we minimize quadric error?

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: $f(x, y) = ax^2 + bxy + cy^2 + dx + ey + g$

$$\mathbf{x} = \begin{bmatrix} x \\ y \end{bmatrix} \quad A = \begin{bmatrix} a & b/2 \\ b/2 & c \end{bmatrix} \quad \mathbf{u} = \begin{bmatrix} d \\ e \end{bmatrix}$$

$$f(x, y) = \mathbf{x}^\mathsf{T} A \mathbf{x} + \mathbf{u}^\mathsf{T} \mathbf{x} + g$$

(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)

$$x = -b/2a$$

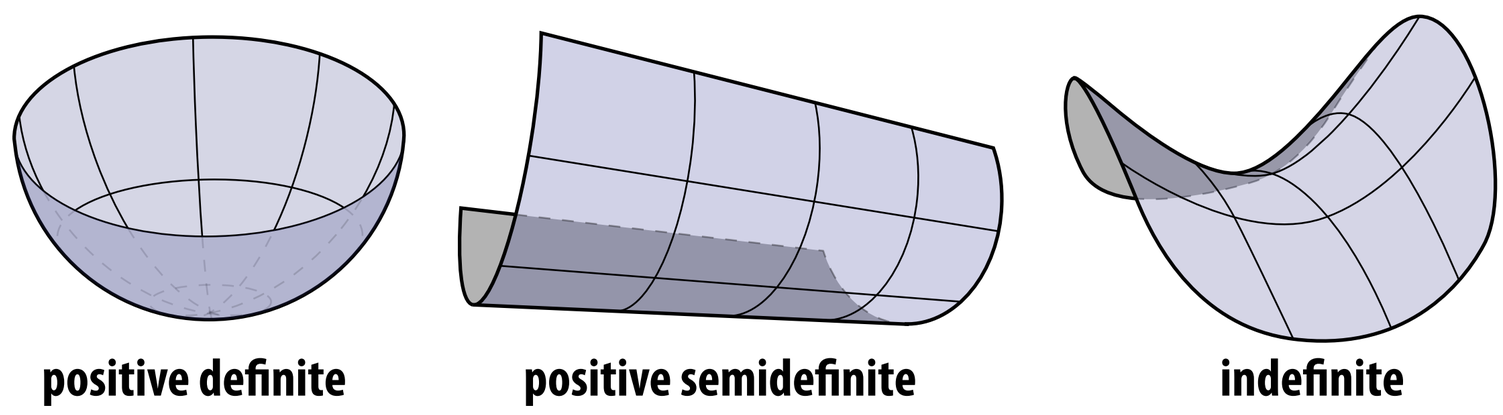
(Can you show this is true, at least in 2D?)

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:*

$$\mathbf{x}^\top A\,\mathbf{x} > 0 \qquad \forall\,\mathbf{x}$$

- 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 <u>linear</u> 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:

$$\begin{bmatrix} \mathbf{x}^\mathsf{T} & 1 \end{bmatrix} \begin{bmatrix} B & \mathbf{w} \\ \mathbf{w}^\mathsf{T} & d^2 \end{bmatrix} \begin{bmatrix} \mathbf{x} \\ 1 \end{bmatrix}$$

$$= \mathbf{x}^\mathsf{T} B \mathbf{x} + 2\mathbf{w}^\mathsf{T}\mathbf{x} + d^2$$

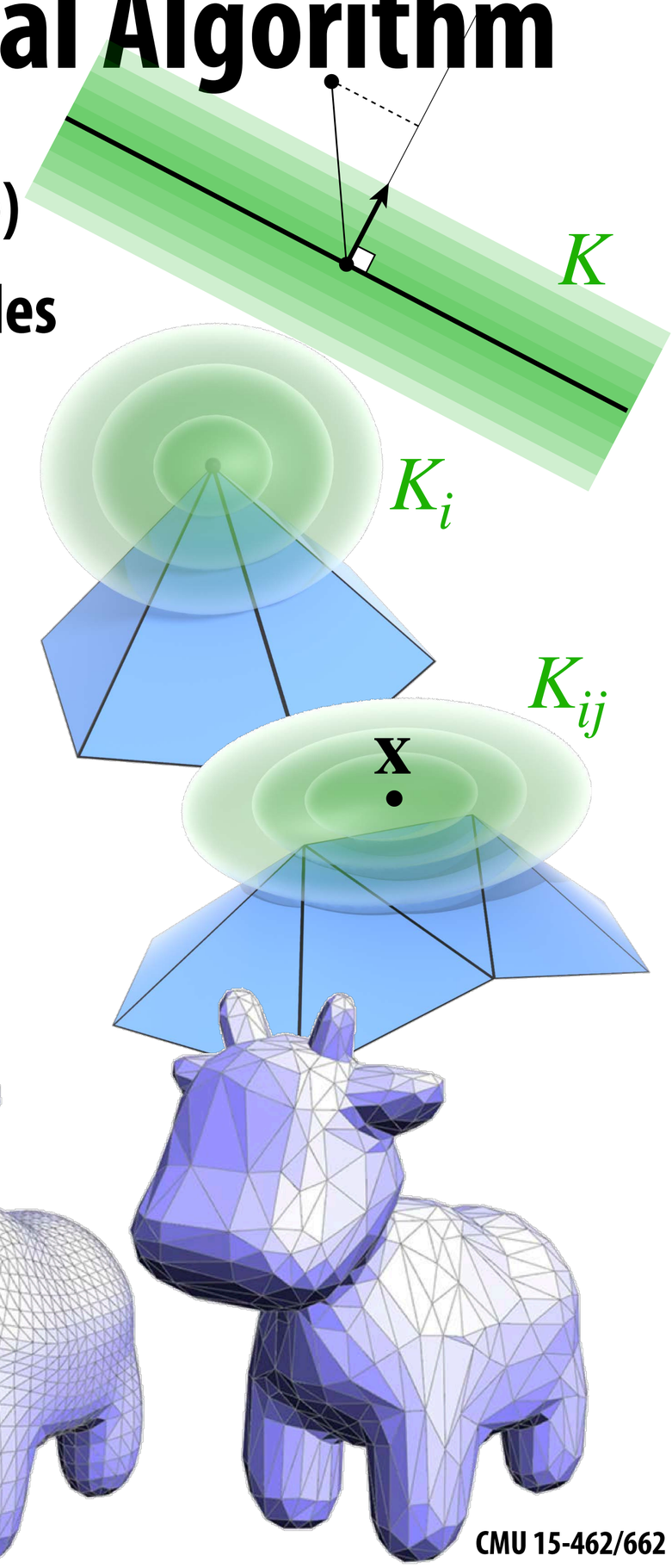
- Now we have a quadratic polynomial in the unknown position $\mathbf{x} \in \mathbb{R}^3$
- Can minimize as before:

$$2B\mathbf{x} + 2\mathbf{w} = 0 \qquad \Longleftrightarrow \qquad \mathbf{x} = -B^{-1}\mathbf{w}$$

Q: Why should B be positive-definite?

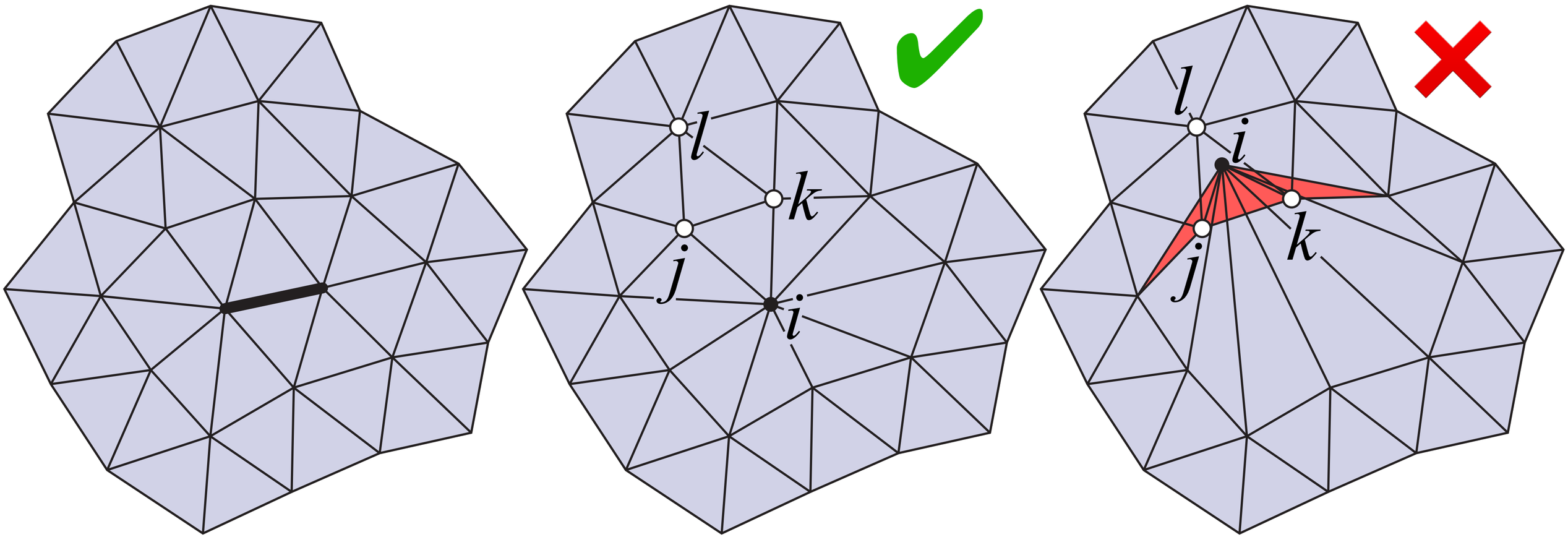
Quadric Error Simplification: Final Algorithm

- Compute K for each triangle (squared distance to plane)
- Set K_i at each vertex to sum of Ks 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):

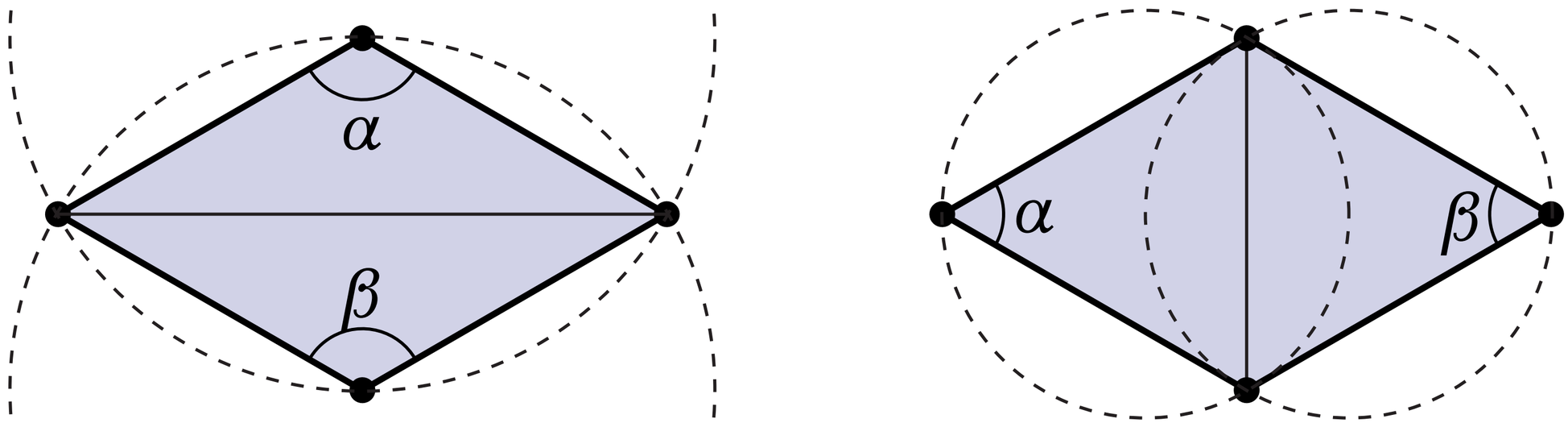


- Easy solution: for each triangle ijk touching collapsed vertex i, consider normals N_{ijk} and N_{kjl} (where kjl is other triangle containing edge jk)
- If $\langle N_{ijk}, N_{kjl} \rangle$ is negative, don't collapse this edge!

What if we're happy with the *number* of triangles, but want to improve *quality?*

How do we make a mesh "more Delaunay"?

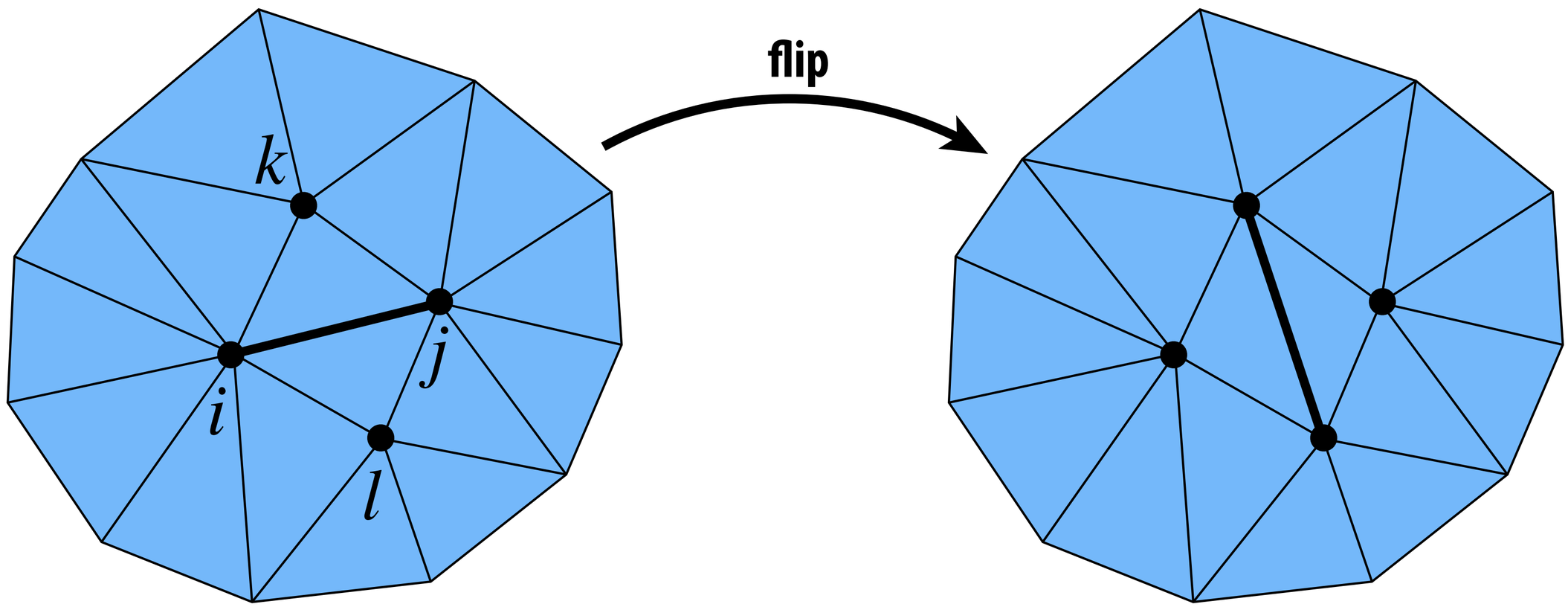
- **Already have a good tool: edge flips!**
- **If $\alpha + \beta > \pi$, flip it!**



- **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!

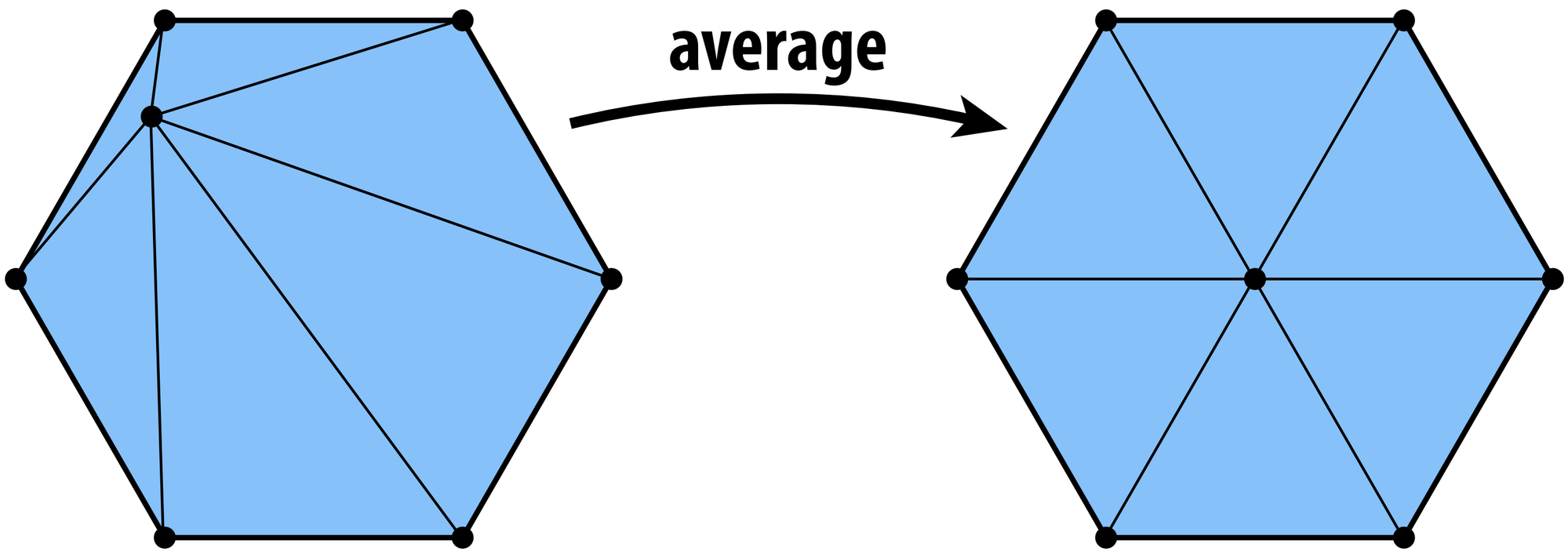


total deviation: $|d_i - 6| + |d_j - 6| + |d_k - 6| + |d_l - 6|$

- 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"?

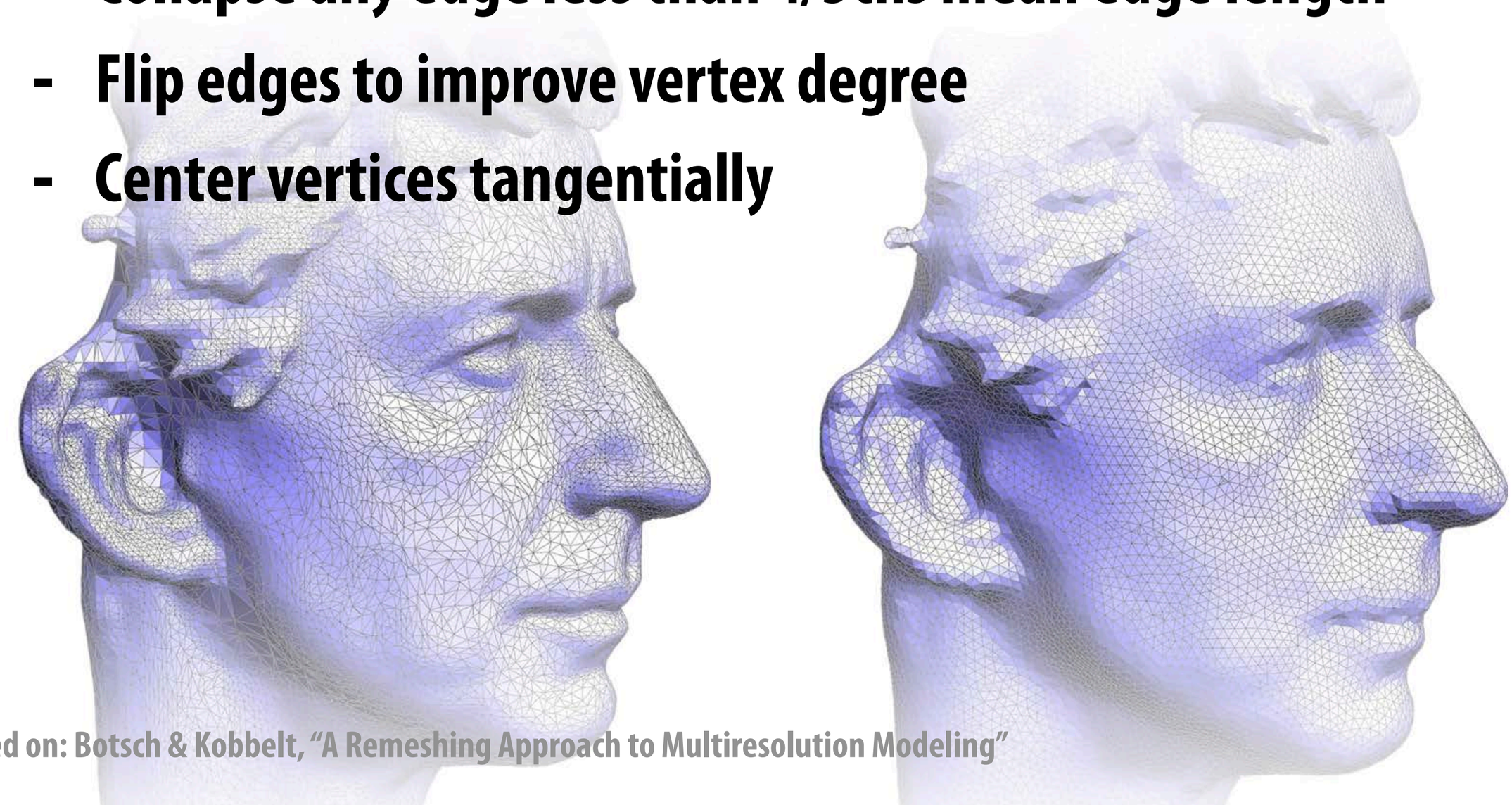
- Delaunay doesn't guarantee triangles are "round" (angles near 60°)
- Can often improve shape by centering vertices:



- 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

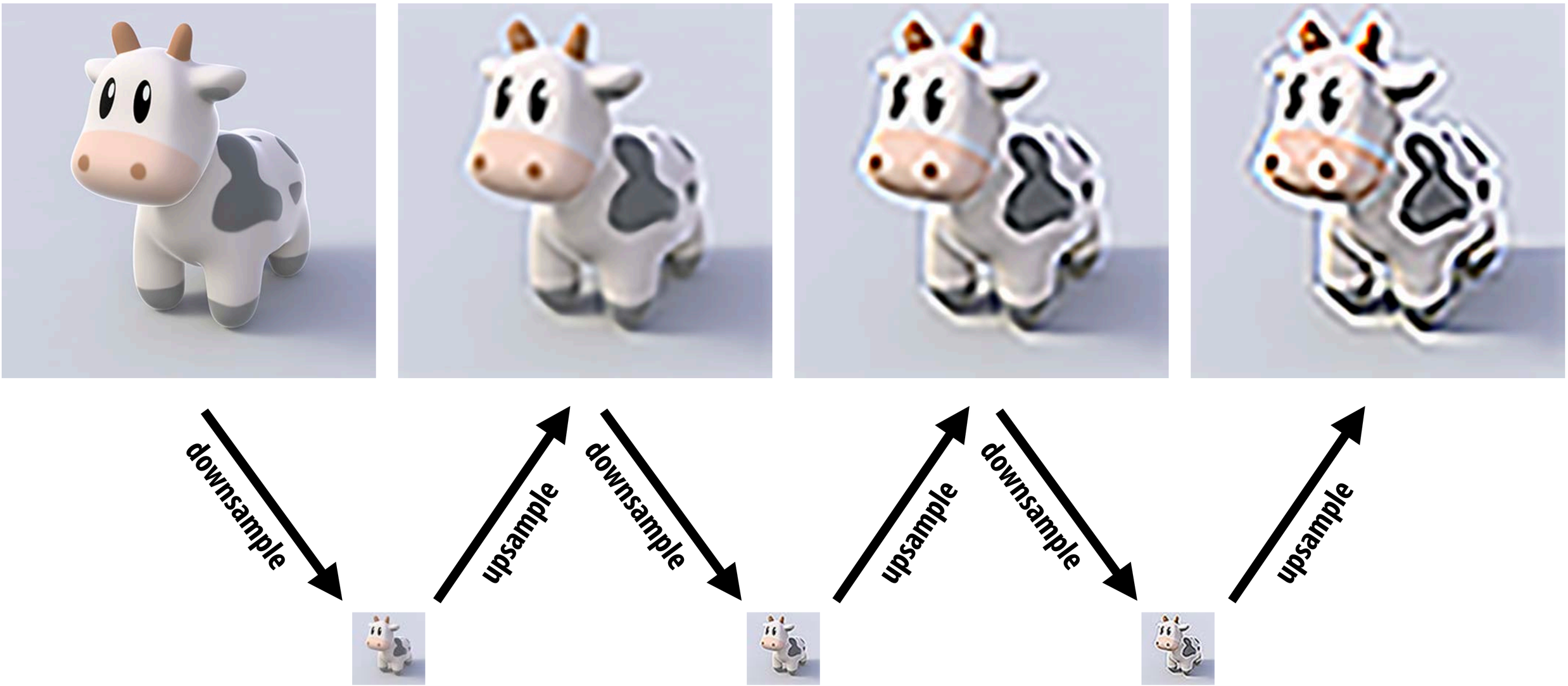


Based on: Botsch & Kobbelt, "A Remeshing Approach to Multiresolution Modeling"

What can go wrong when you resample a signal?

Danger of Resampling

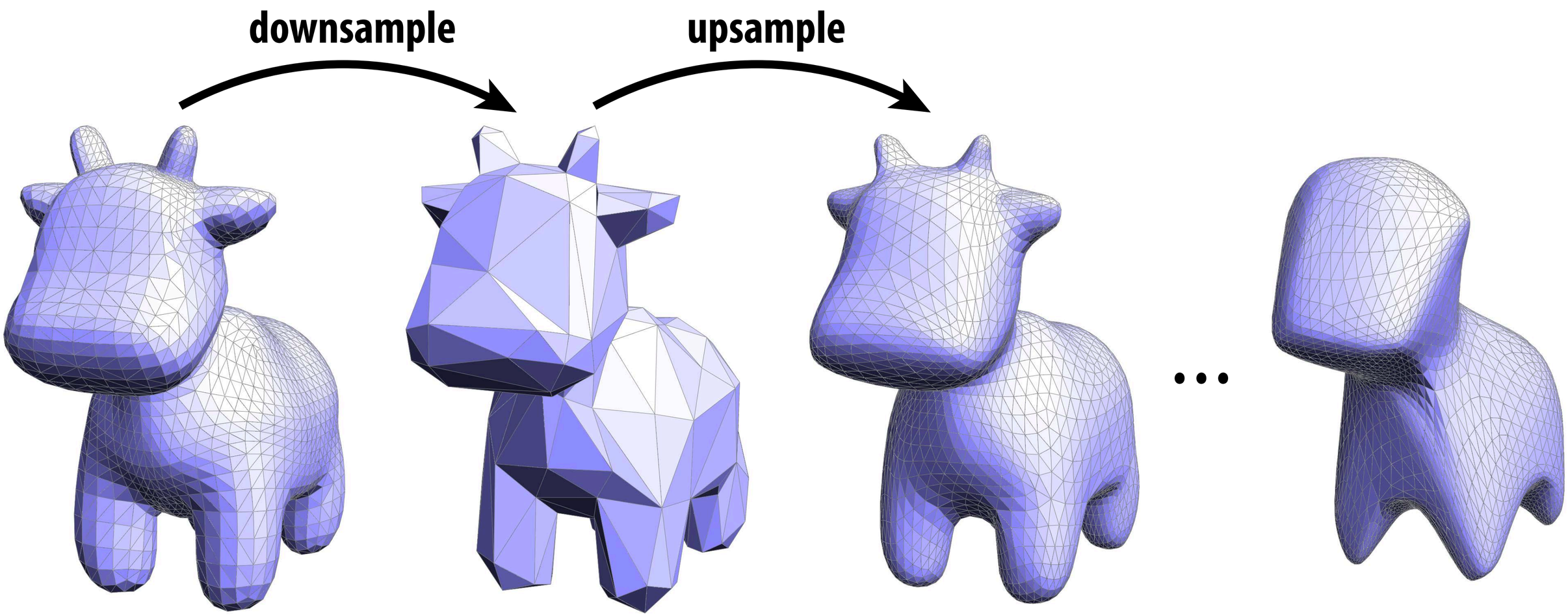
Q: What happens if we repeatedly resample an image?



A: Signal quality degrades!

Danger of Resampling

Q: What happens if we repeatedly resample a mesh?



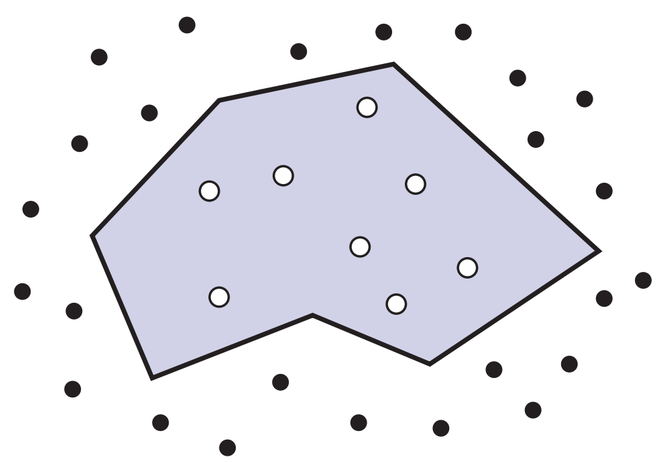
A: Signal also degrades!

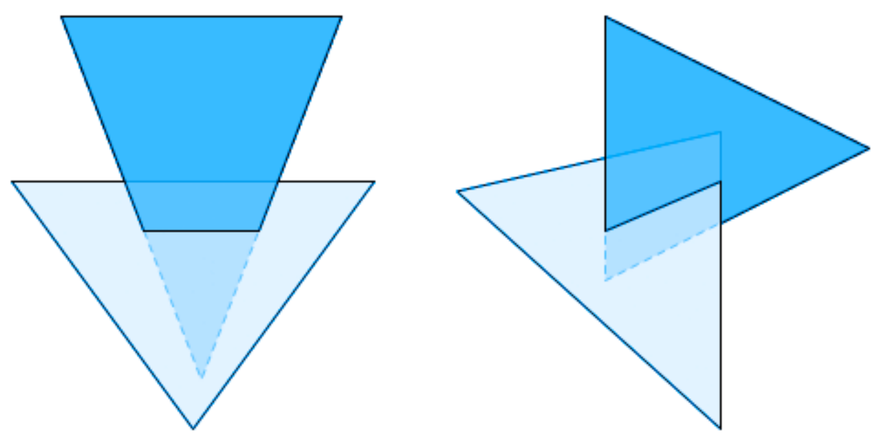
But wait: we have the original signal (mesh).

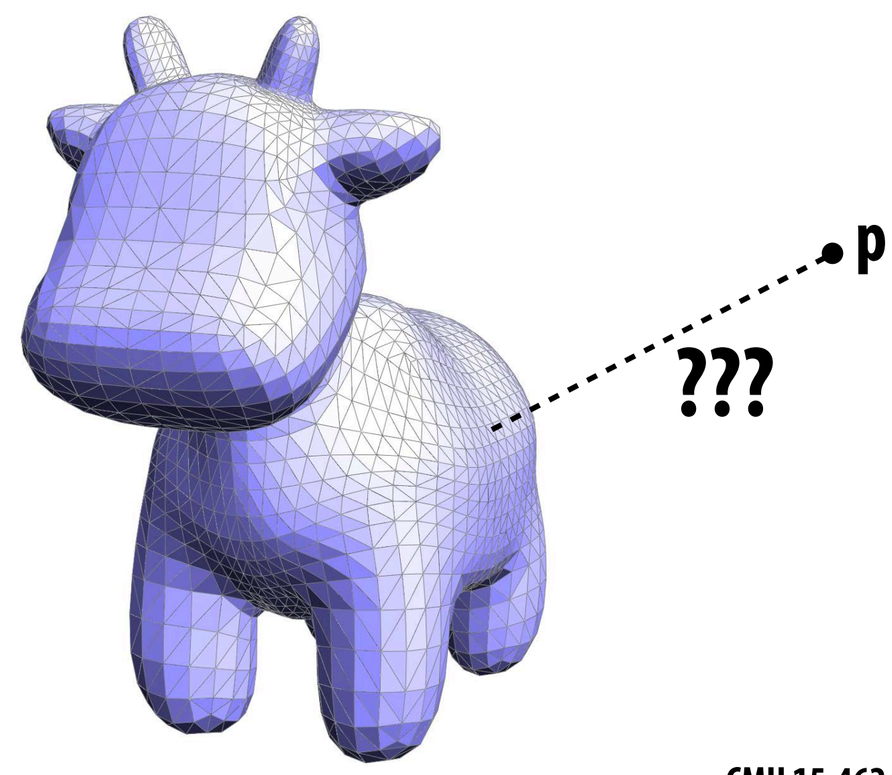
Why not just project each new sample point onto the closest point of the original mesh?

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 <u>accelerate</u> such queries for large meshes?
- So many questions!



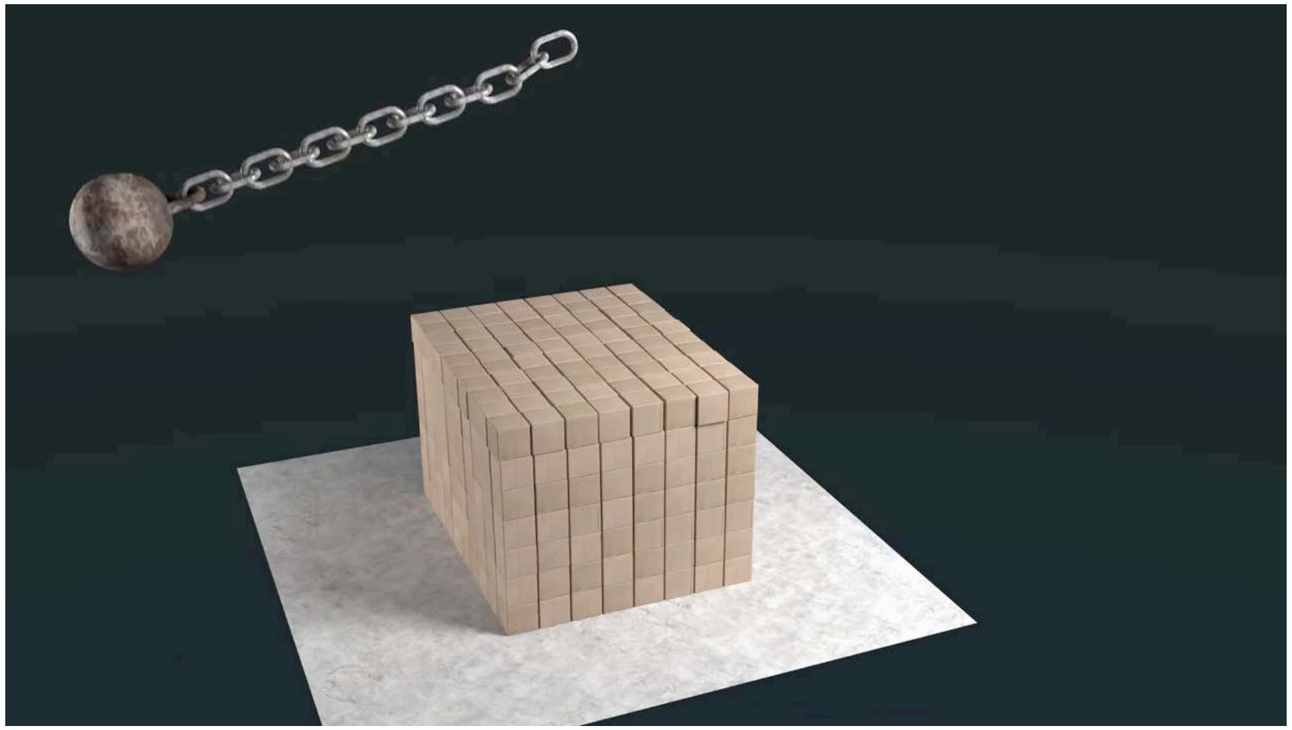


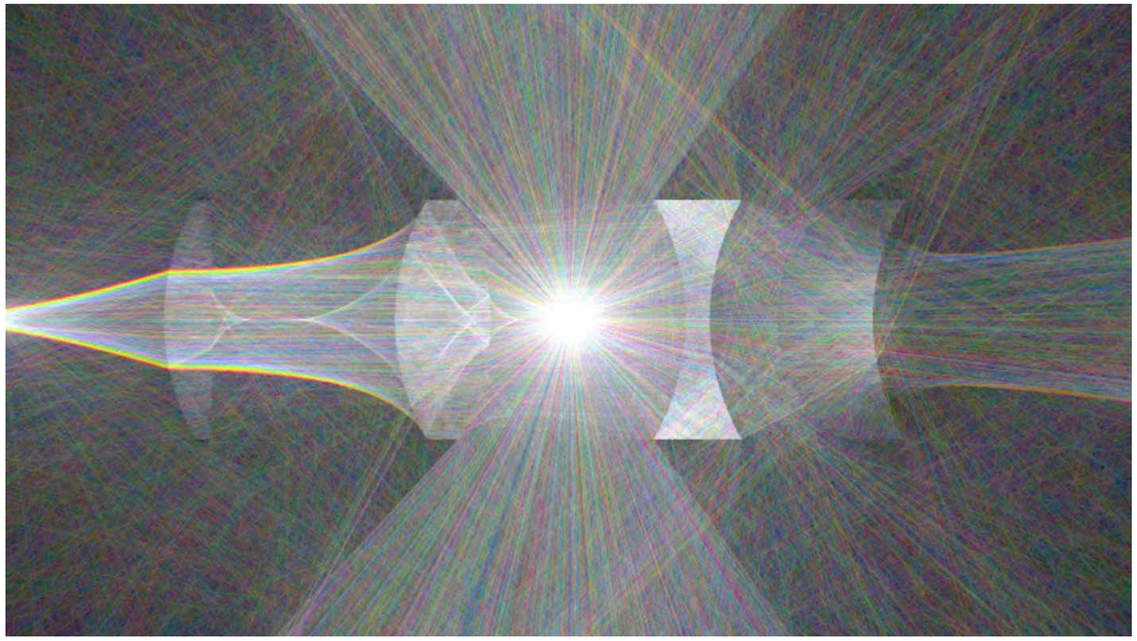


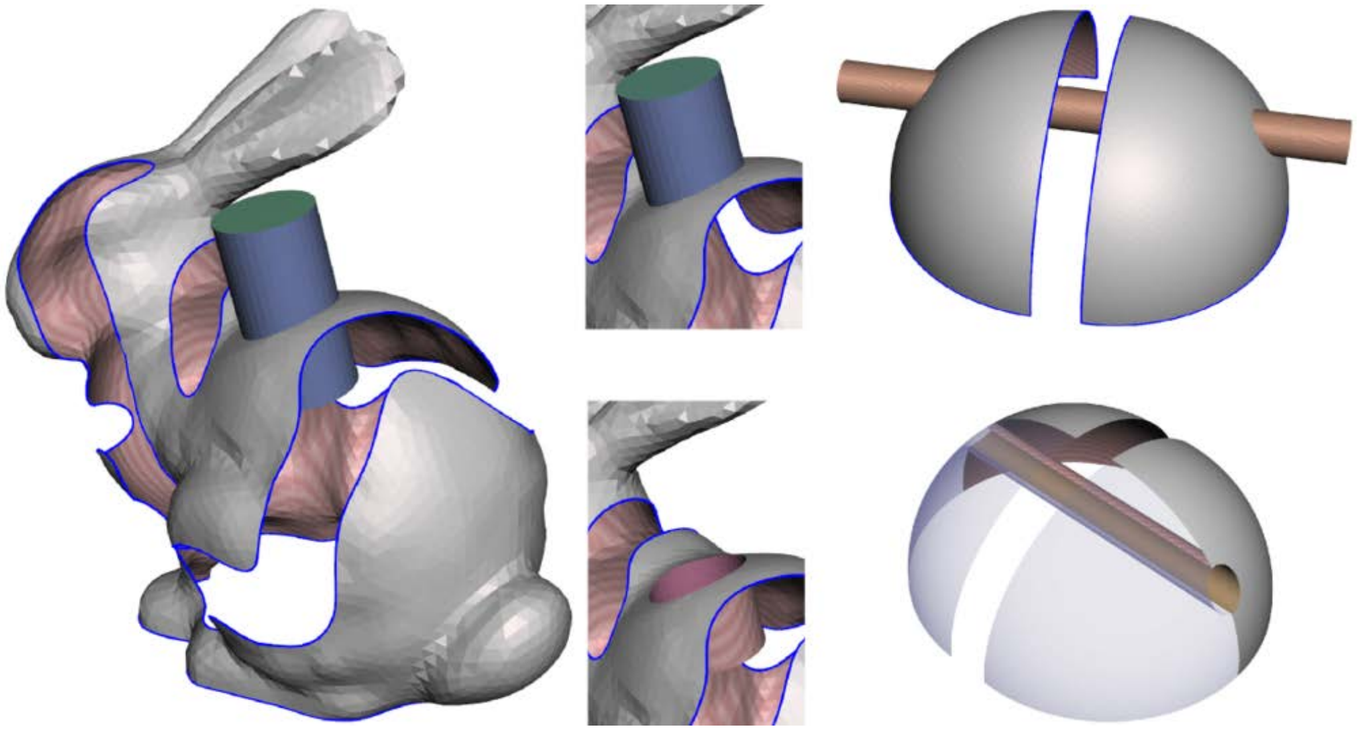
Geometric Queries

Computer Graphics
CMU 15-462/15-662

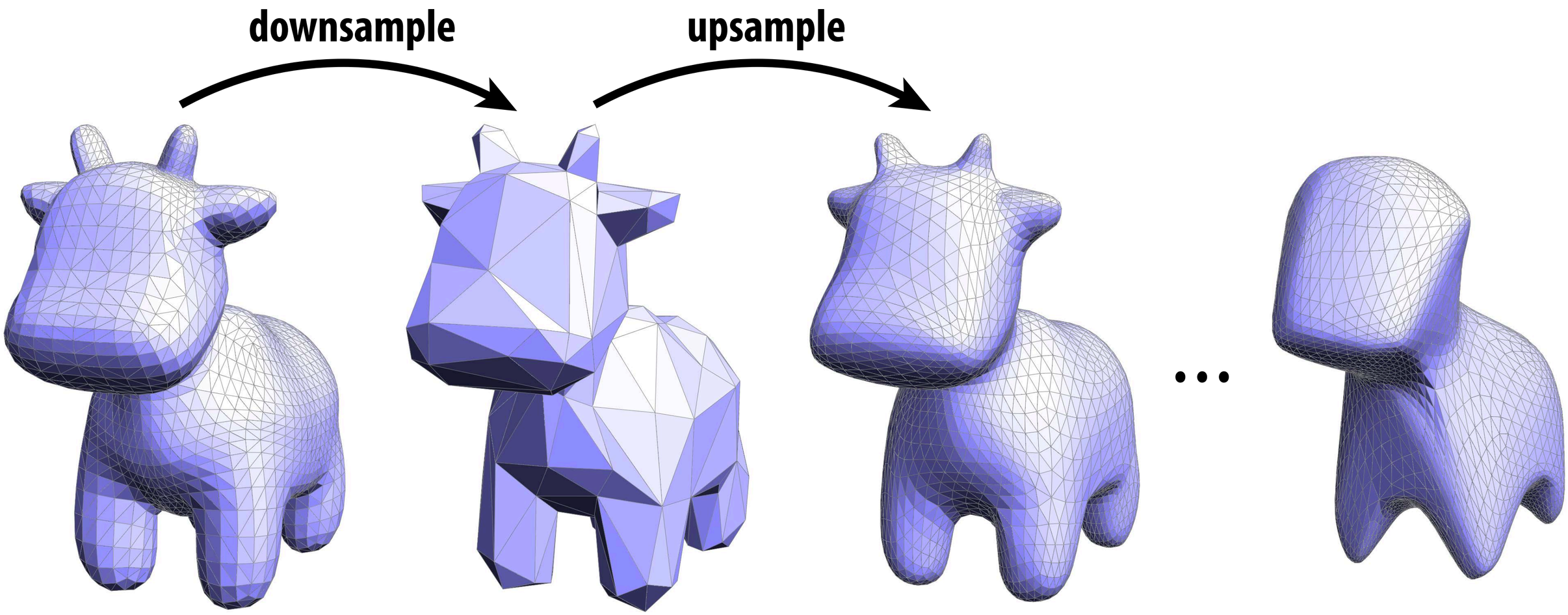
Geometric Queries—Motivation





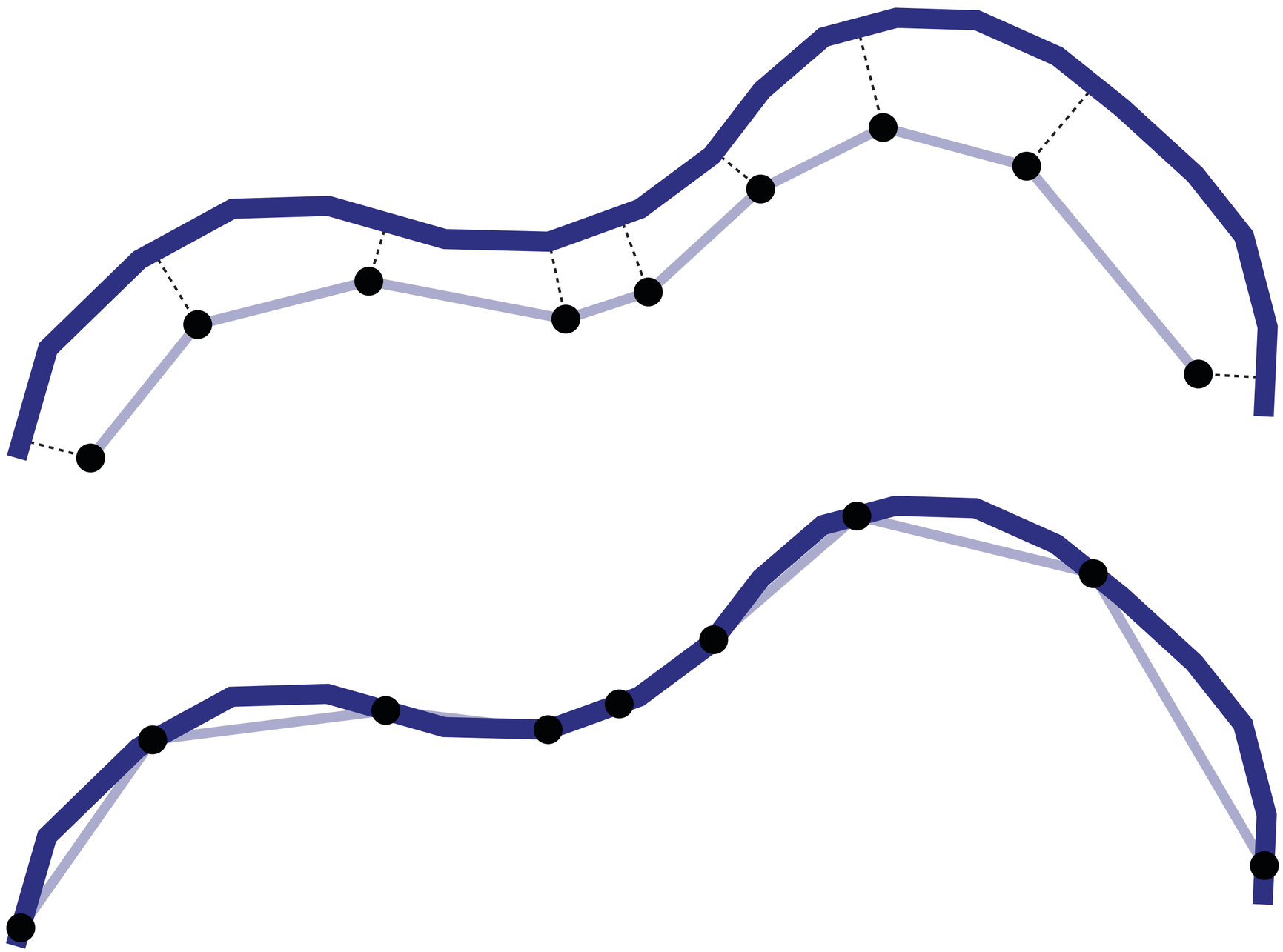


Motivating Example:
Signal Degradation in Geometry Processing



Q: How can we do a better job of preserving the original signal?

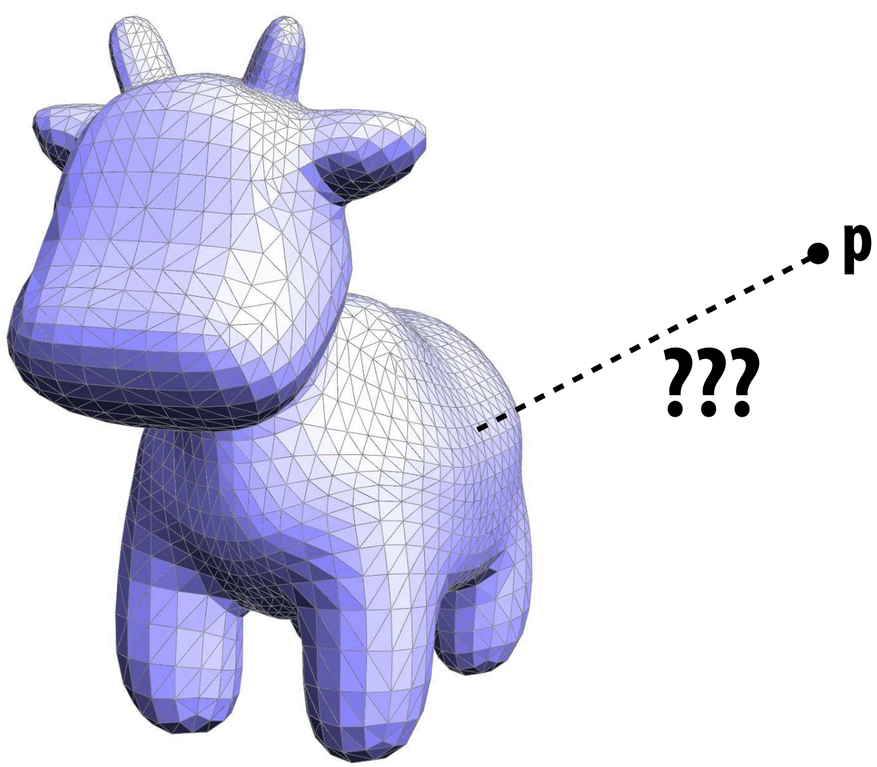
Recovering Fidelity via Closest Point Projection



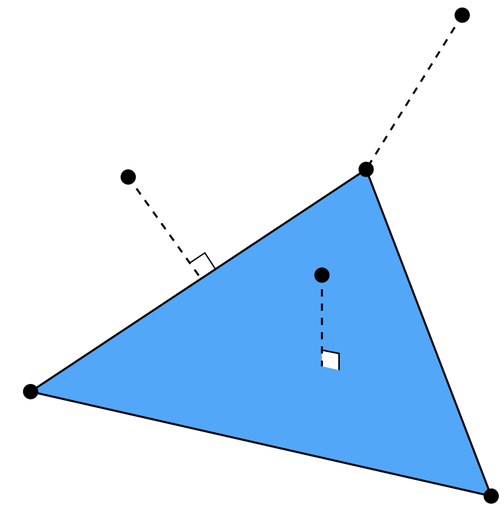
Idea: after resampling, project each vertex onto original mesh

Closest Point Queries

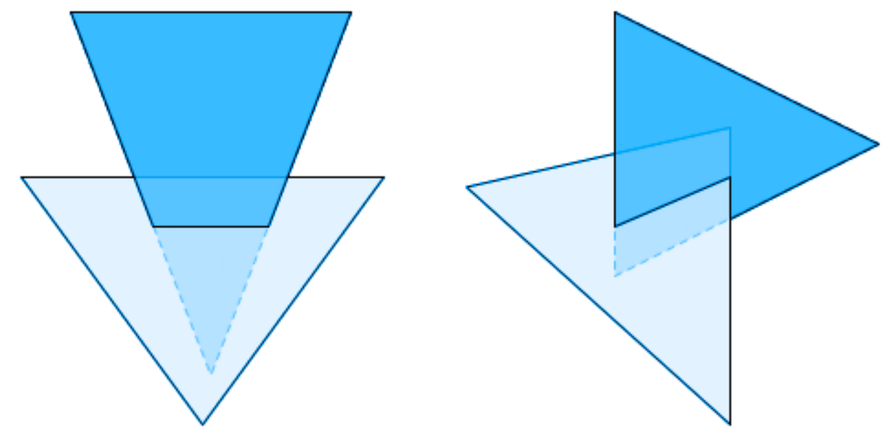
- Q: Given a point, in space (e.g., a new sample point), how do we find the closest point on a given surface?
- Q: Does implicit/explicit representation make this easier?
- Q: Does our choice of mesh data structure make a difference?
- Q: What's the cost of the naïve algorithm?
- Q: How do we find the distance to a single triangle anyway?
- So many questions!



Many types of geometric queries



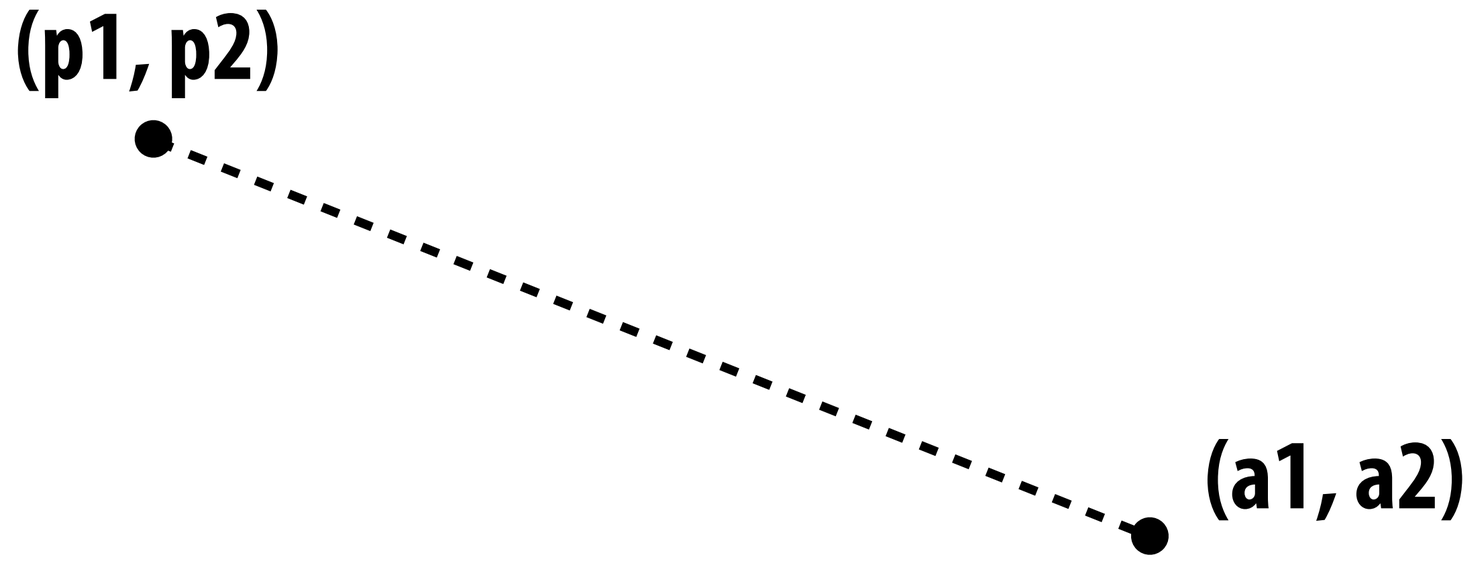
- Already identified need for "closest point" query
- Plenty of other things we might like to know:
  - Do two triangles intersect?
  - Are we inside or outside an object?
  - Does one object contain another?
  - ...



- Data structures we've seen so far not really designed for this...
- Need some new ideas!
- TODAY: come up with simple (read: slow) algorithms.
- NEXT TIME: intelligent ways to accelerate geometric queries.

Warm up: closest point on point

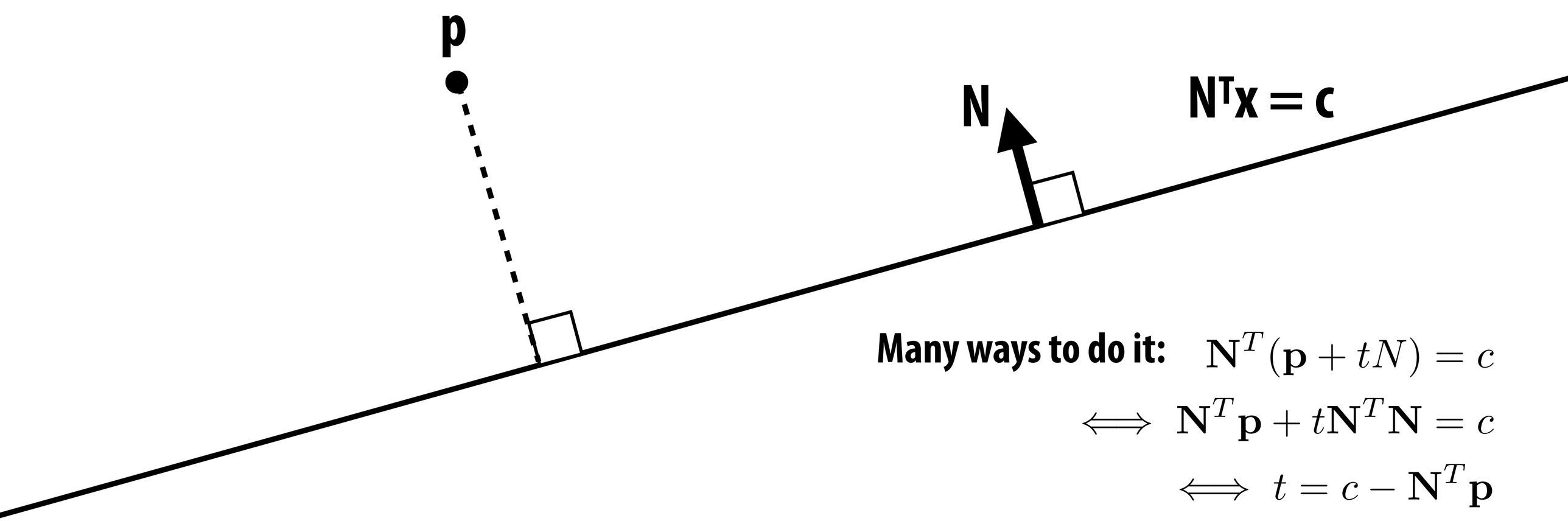
- Goal is to find the point on a mesh closest to a given point.
- *Much* simpler question: given a query point (p1,p2), how do we find the closest point on the point (a1,a2)?



Bonus question: what's the distance?

Slightly harder: closest point on line

- Now suppose I have a line $N^T x = c$, where N is the unit normal
- How do I find the point closest to my query point p?



Many ways to do it:

$$\mathbf{N}^T(\mathbf{p} + tN) = c$$

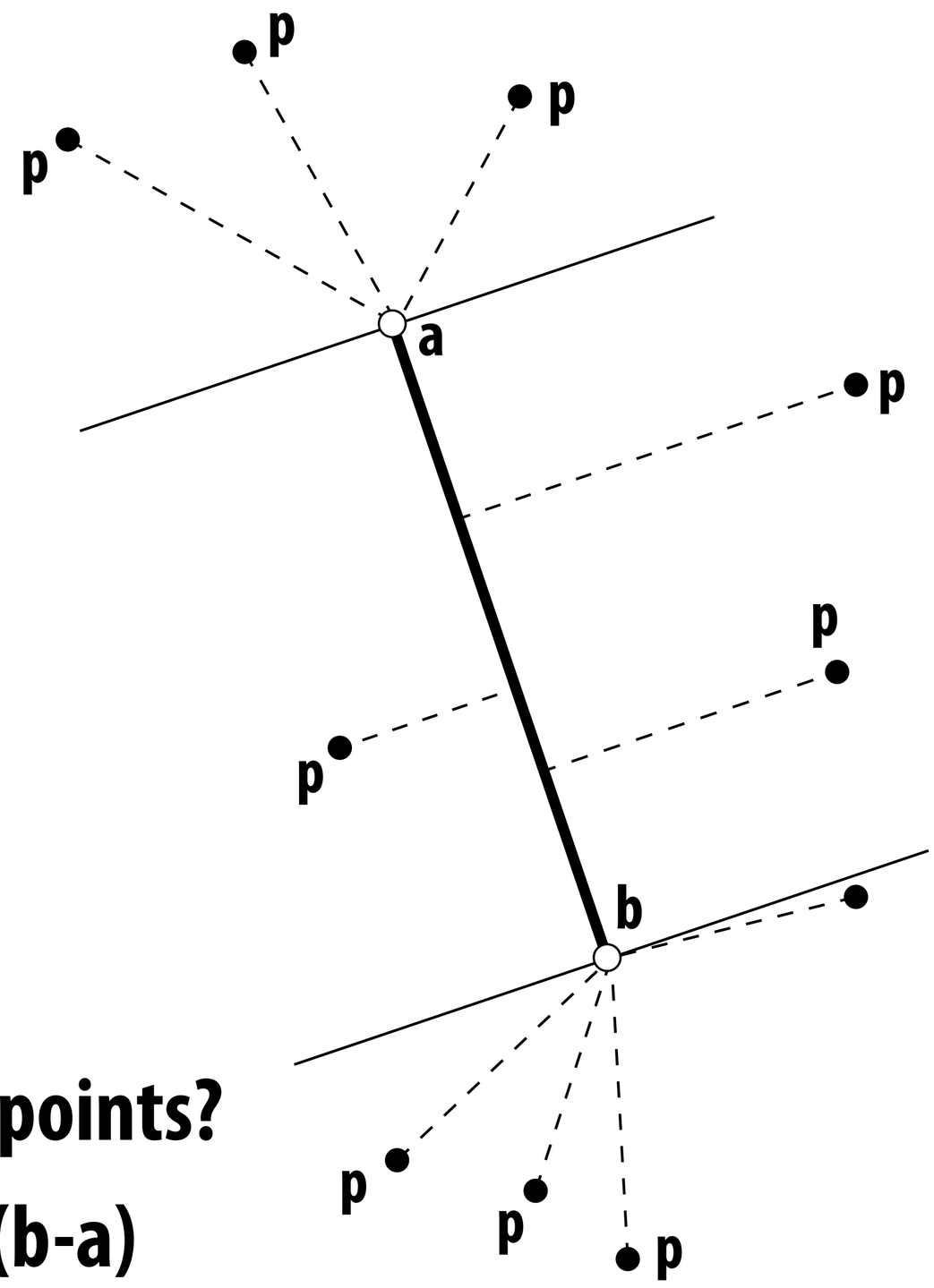
$$\iff \mathbf{N}^T\mathbf{p} + t\mathbf{N}^T\mathbf{N} = c$$

$$\iff t = c - \mathbf{N}^T\mathbf{p}$$

$$\Rightarrow \mathbf{p} + t\mathbf{N} = \boxed{\mathbf{p} + (c - \mathbf{N}^T\mathbf{p})\mathbf{N}}$$

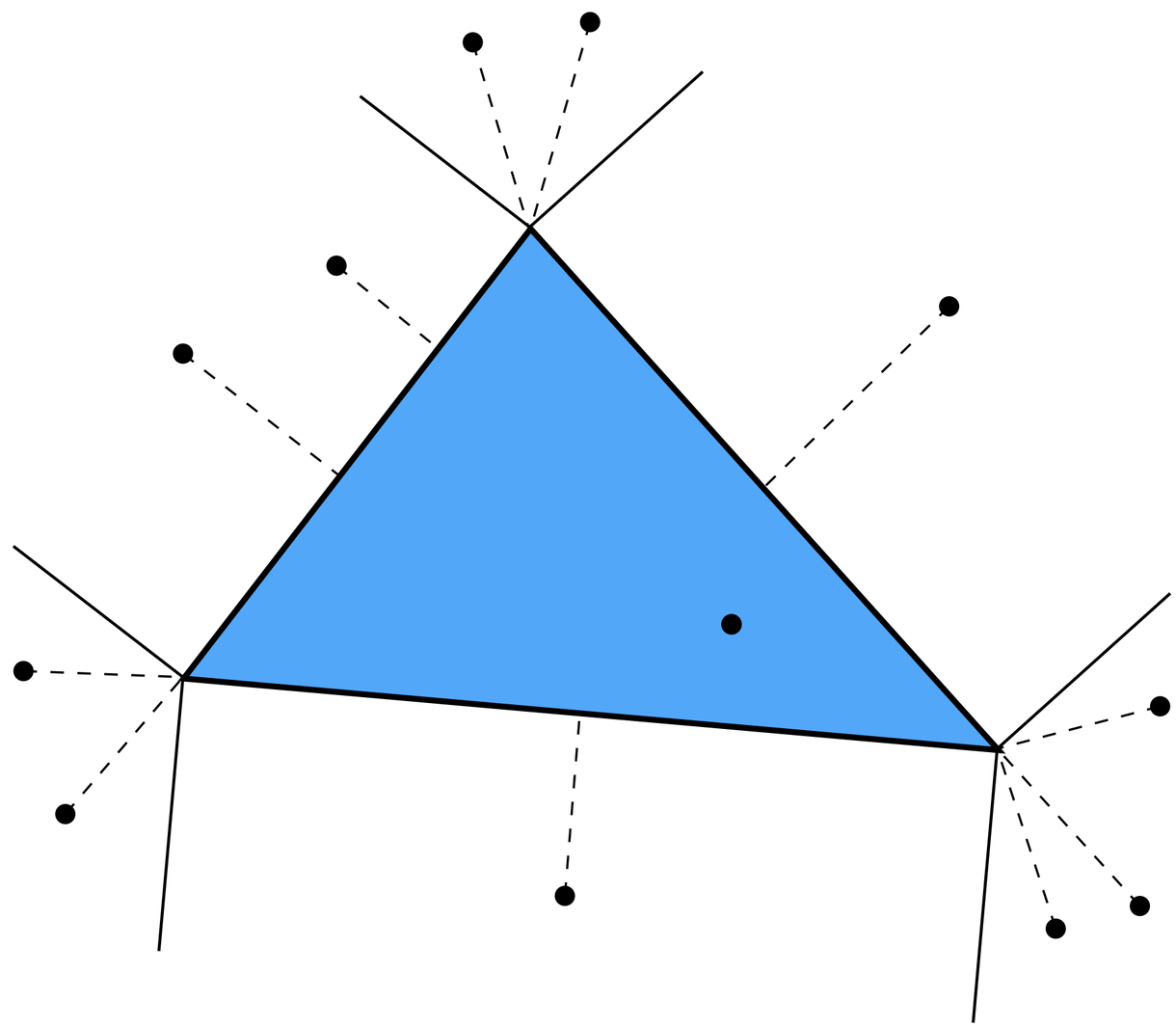
Harder: closest point on line segment

- Two cases: endpoint or interior
- Already have basic components:
  - point-to-point
  - point-to-line
- Algorithm?
  - find closest point on line
  - check if it's between endpoints
  - if not, take closest endpoint
- How do we know if it's between endpoints?
  - write closest point on line as a+t(b-a)
  - if t is between 0 and 1, it's inside the segment!



Even harder: closest point on triangle

- What are all the possibilities for the closest point?
- Almost just minimum distance to three segments:



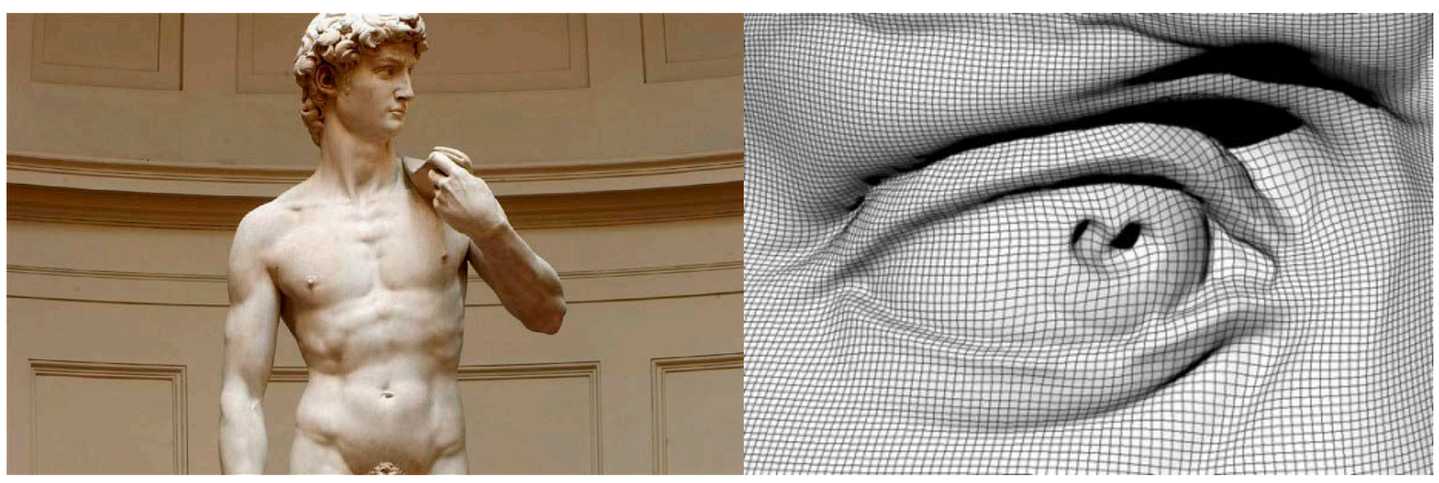
Q: What about a point inside the triangle?

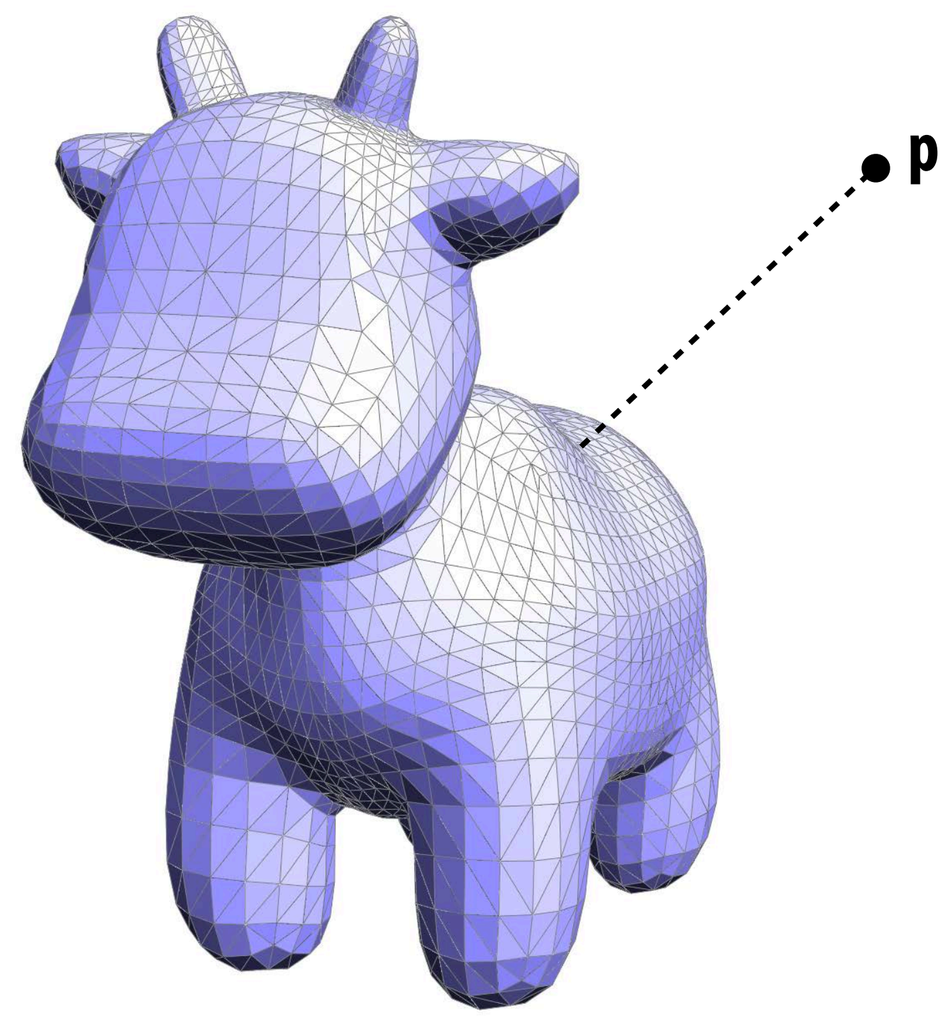
Closest point on triangle in 3D

- Not so different from 2D case
- Algorithm?
  - project onto plane of triangle
  - use half-*space* tests to classify point (vs. half plane)
  - if inside the triangle, we're done!
  - otherwise, find closest point on associated vertex or edge
- By the way, how do we find closest point on plane?
- Same expression as closest point on a line!
- E.g., $p + (c - N^T p) N$

Closest point on triangle *mesh* in 3D?

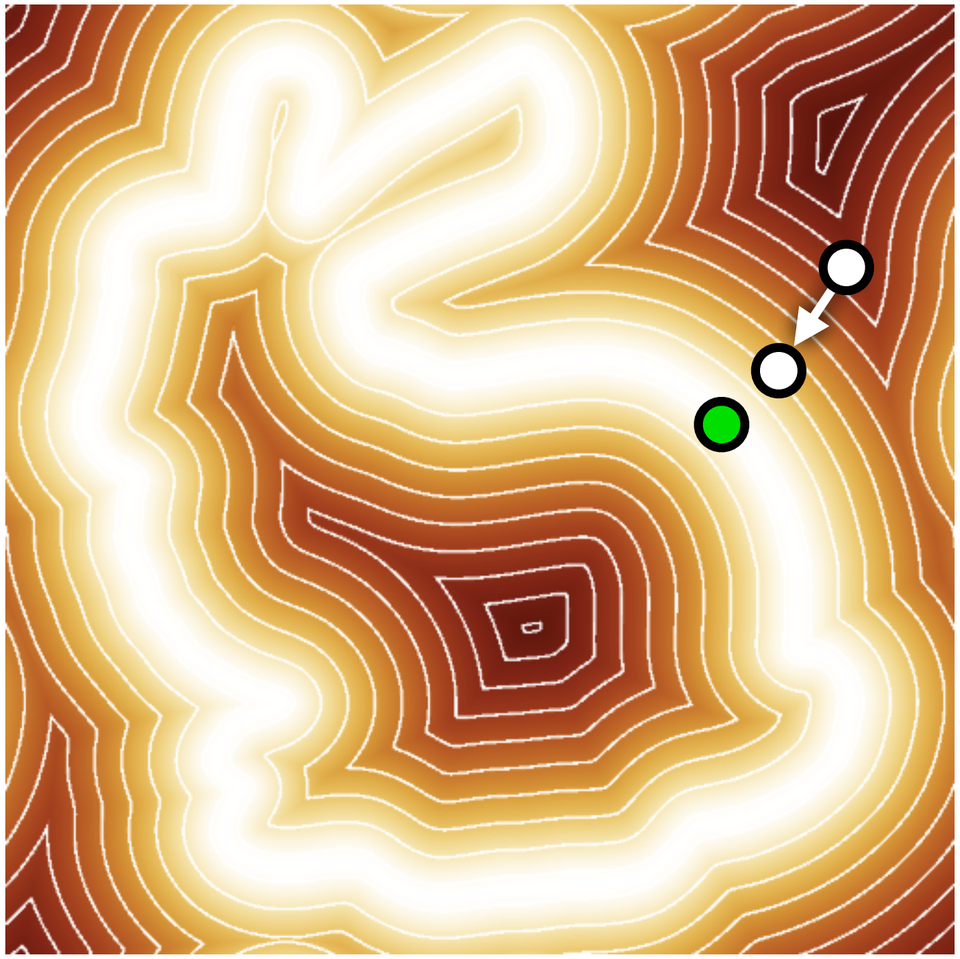
- Conceptually easy:
  - loop over all triangles
  - compute closest point to current triangle
  - keep globally closest point
- Q: What's the cost?
- What if we have *billions* of faces?
- NEXT TIME: Better data structures!





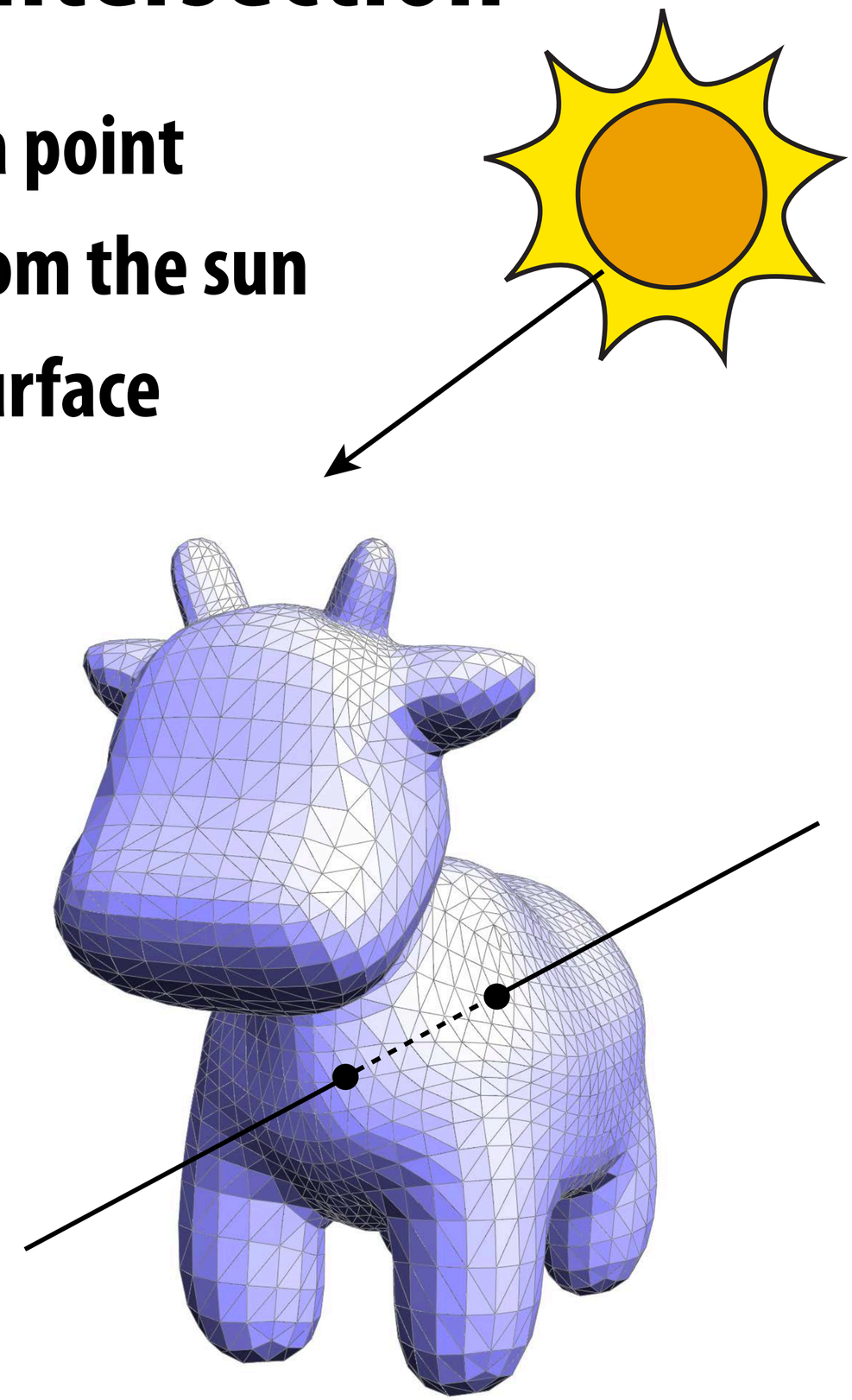
Closest point to *implicit* surface?

- If we change our representation of geometry, algorithms can change completely
- E.g., how might we compute the closest point on an implicit surface described via its distance function?
- One idea:
  - start at the query point
  - compute gradient of distance (using, e.g., finite differences)
  - take a little step (decrease distance)
  - repeat until we're at the surface (zero distance)
- Better yet: just store closest point for each grid cell! (speed/memory trade off)



Different query: ray-mesh intersection

- A "ray" is an oriented line starting at a point
- Think about a ray of light traveling from the sun
- Want to know where a ray pierces a surface
- Why?
  - GEOMETRY: inside-outside test
  - RENDERING: visibility, ray tracing
  - ANIMATION: collision detection
- Might pierce surface in many places!



Ray equation

- Can express ray as

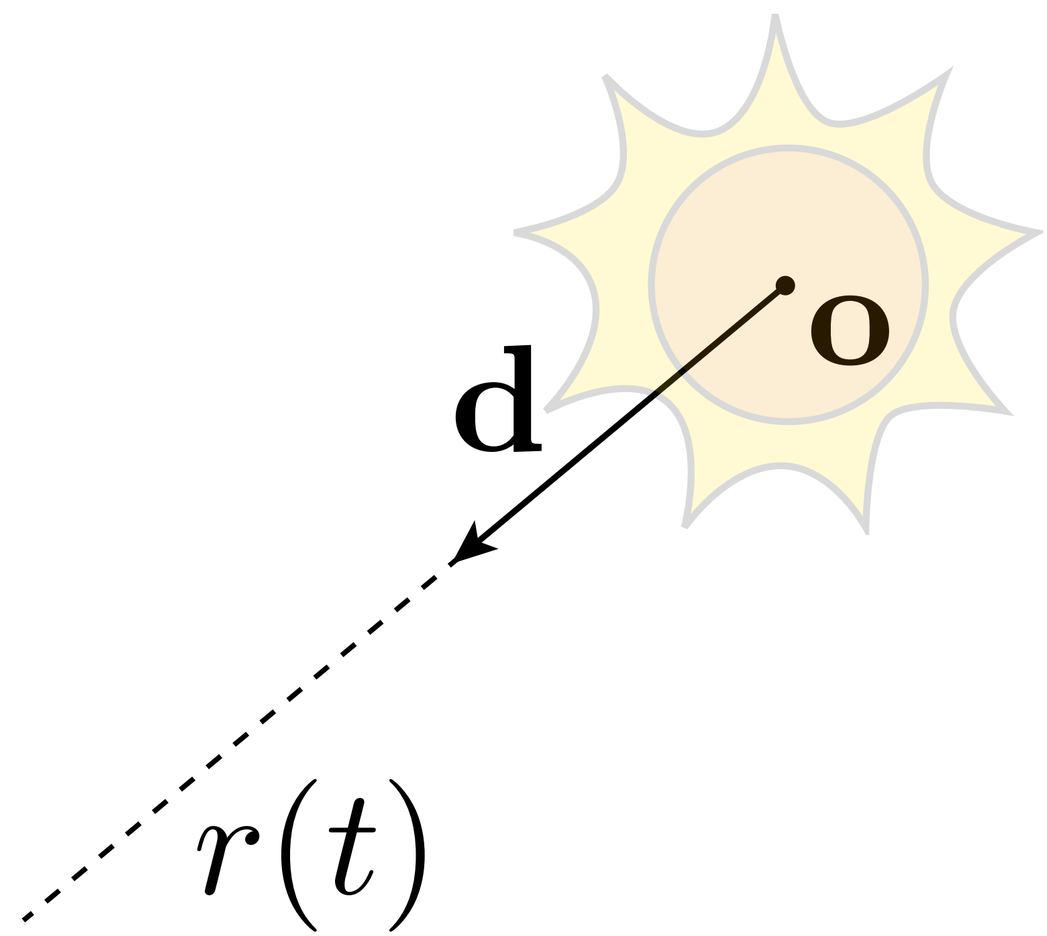
origin

unit direction

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

point along ray

"time"



Intersecting a ray with an implicit surface

- Recall implicit surfaces: all points x such that f(x) = 0
- Q: How do we find points where a ray pierces this surface?
- Well, we know all points along the ray: r(t) = o + td
- Idea: replace "x" with "r" in 1st equation, and solve for t
- Example: unit sphere

quadratic formula:

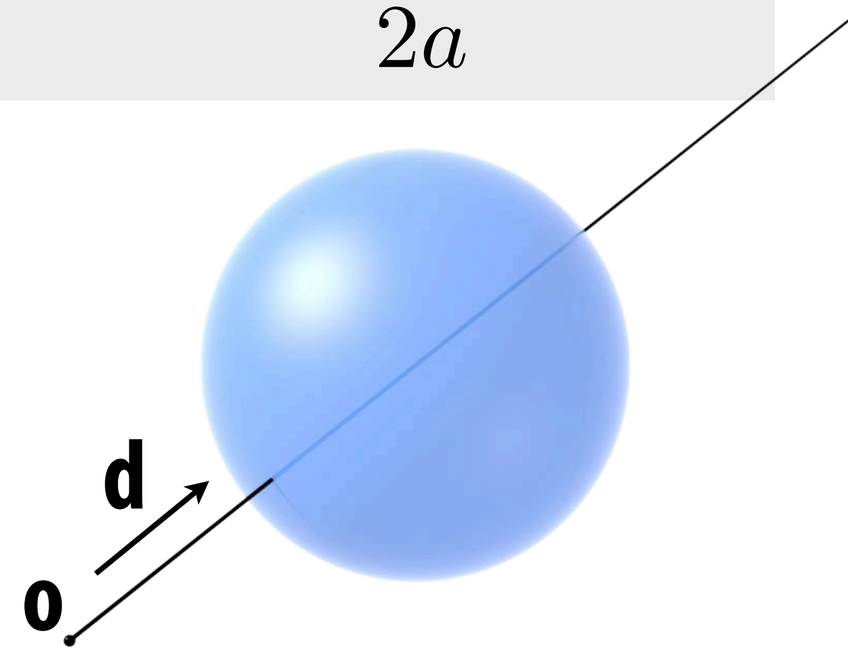
$$t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

$$f(\mathbf{x}) = |\mathbf{x}|^2 - 1$$

$$\Rightarrow f(\mathbf{r}(t)) = |\mathbf{o} + t\mathbf{d}|^2 - 1$$

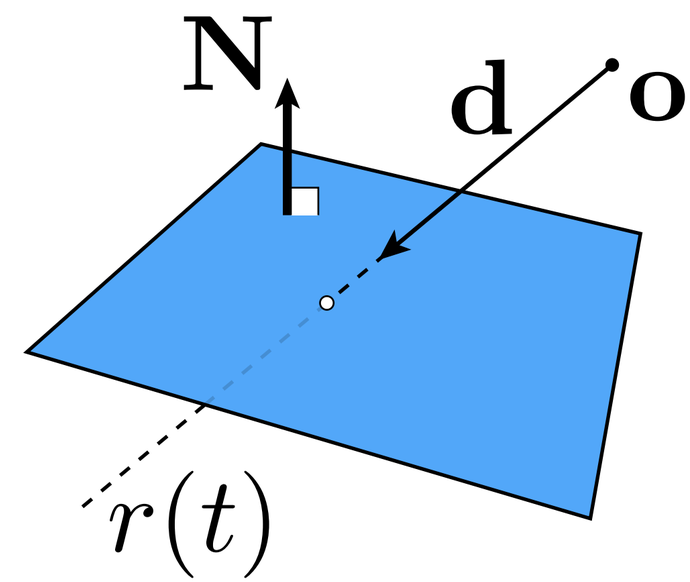
$$\underbrace{|\mathbf{d}|^2}_{a} t^2 + \underbrace{2(\mathbf{o} \cdot \mathbf{d})}_{b} t + \underbrace{|\mathbf{o}|^2 - 1}_{c} = 0$$

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



Why two solutions?

Ray-plane intersection



- Suppose we have a plane $\mathbf{N}^\mathsf{T}\mathbf{x} = c$
  - N - unit normal
  - c - offset
- How do we find intersection with ray $\mathbf{r}(t) = \mathbf{o} + t\mathbf{d}$?
- *Key idea:* again, replace the point x with the ray equation t:

$$\mathbf{N}^\mathsf{T}\mathbf{r}(t) = c$$

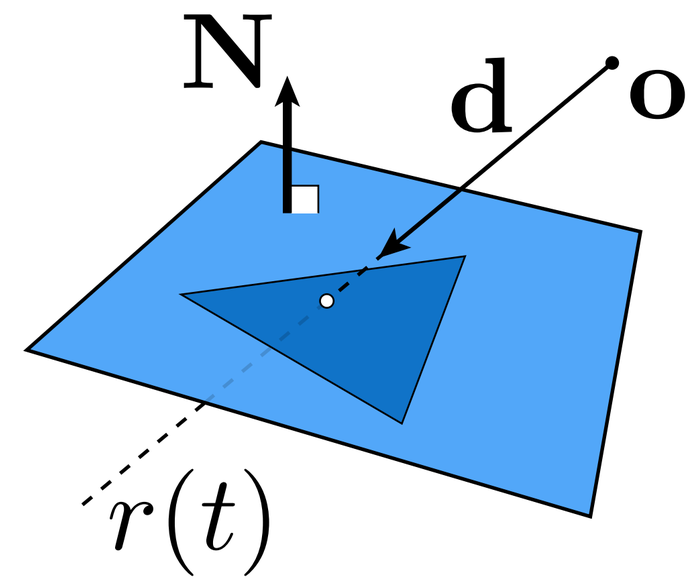
- Now solve for t:

$$\mathbf{N}^\mathsf{T}(\mathbf{o} + t\mathbf{d}) = c \qquad \Rightarrow t = \frac{c - \mathbf{N}^\mathsf{T}\mathbf{o}}{\mathbf{N}^\mathsf{T}\mathbf{d}}$$

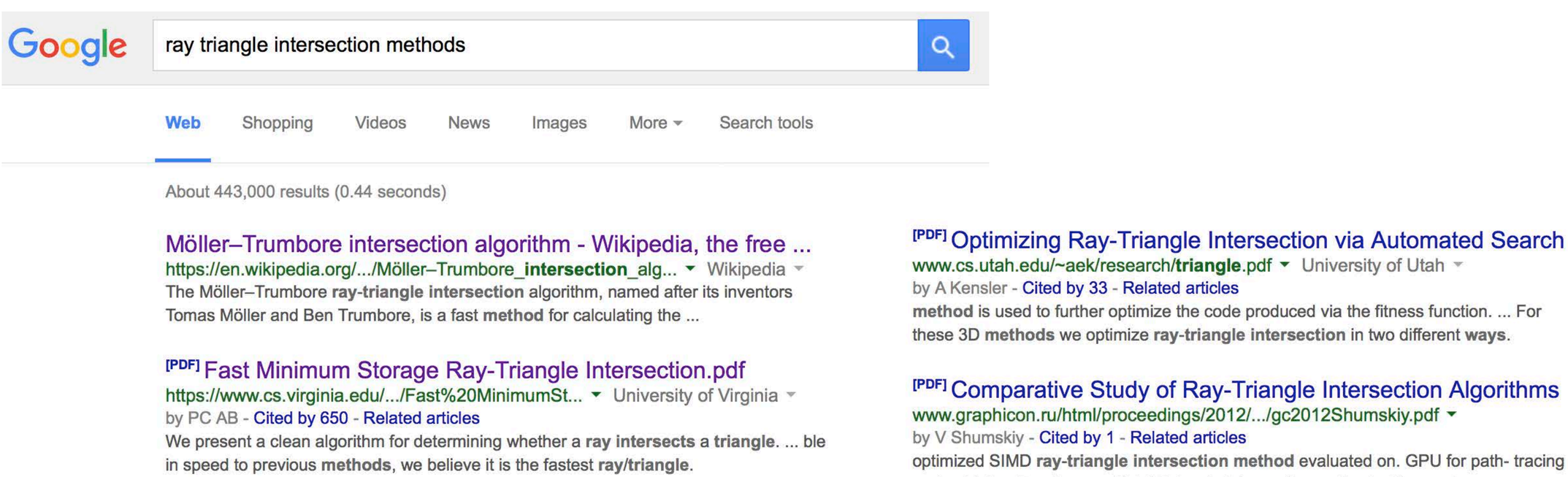
- And plug t back into ray equation:

$$r(t) = \mathbf{o} + \frac{c - \mathbf{N}^\mathsf{T}\mathbf{o}}{\mathbf{N}^\mathsf{T}\mathbf{d}}\mathbf{d}$$

Ray-triangle intersection



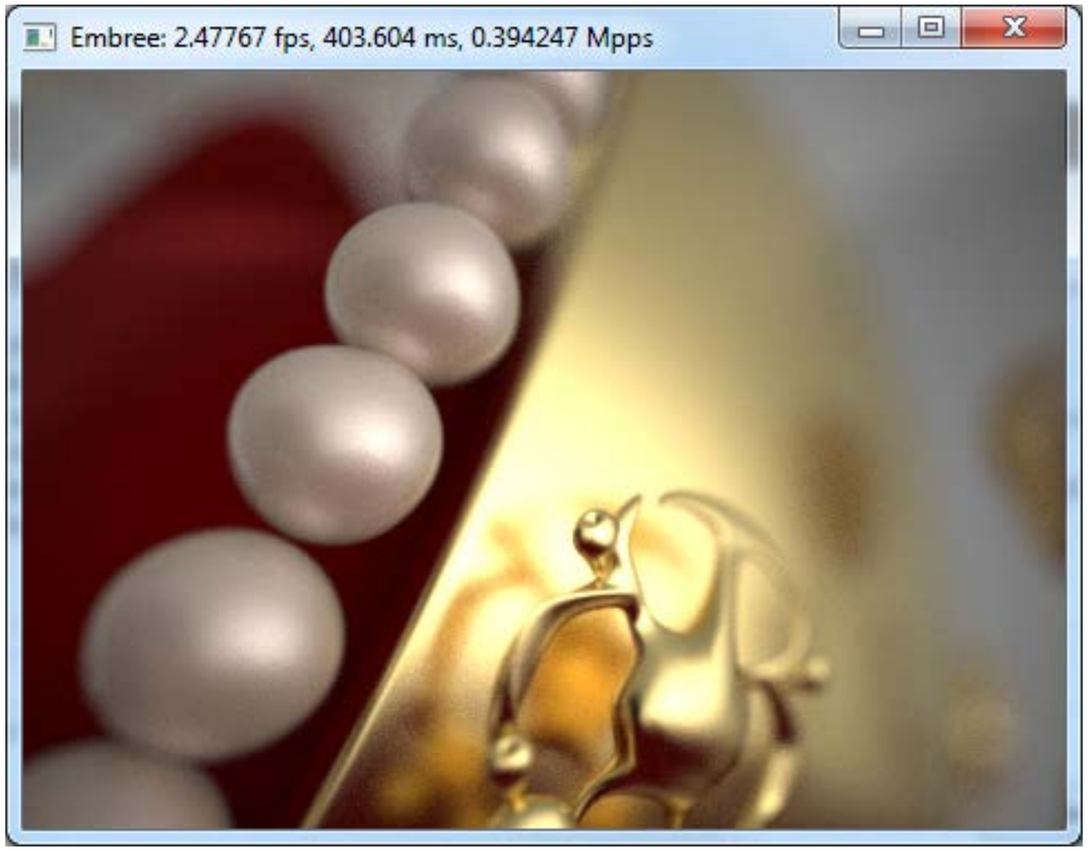
- Triangle is in a plane...
- Not much more to say!
  - Compute ray-plane intersection
  - Q: What do we do now?
  - A: Why not compute barycentric coordinates of hit point?
  - If barycentric coordinates are all positive, point in triangle
- Actually, a *lot* more to say... if you care about performance!



Why care about performance?

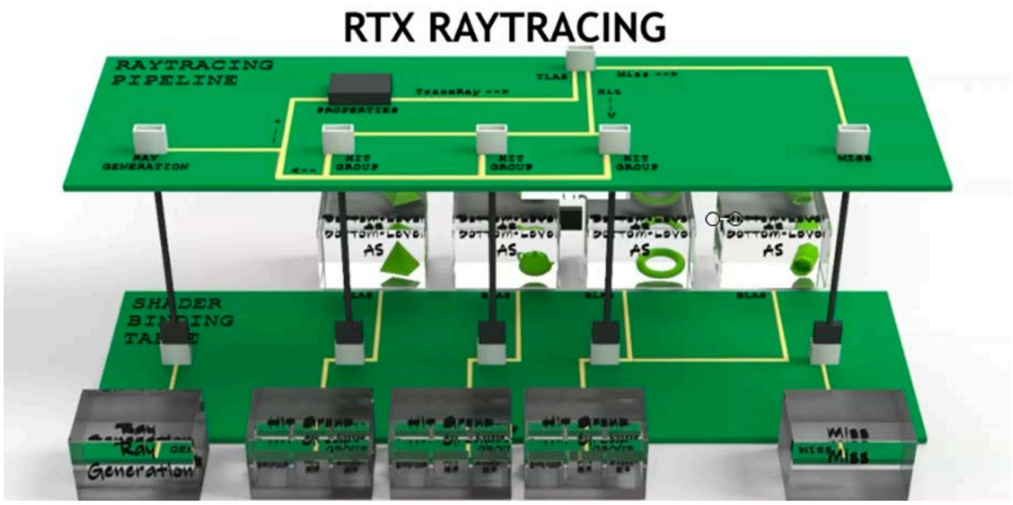
Pixar's "Coco" — about 50 <u>hours</u> per frame (@24 frames/sec)

High-performance ray tracing



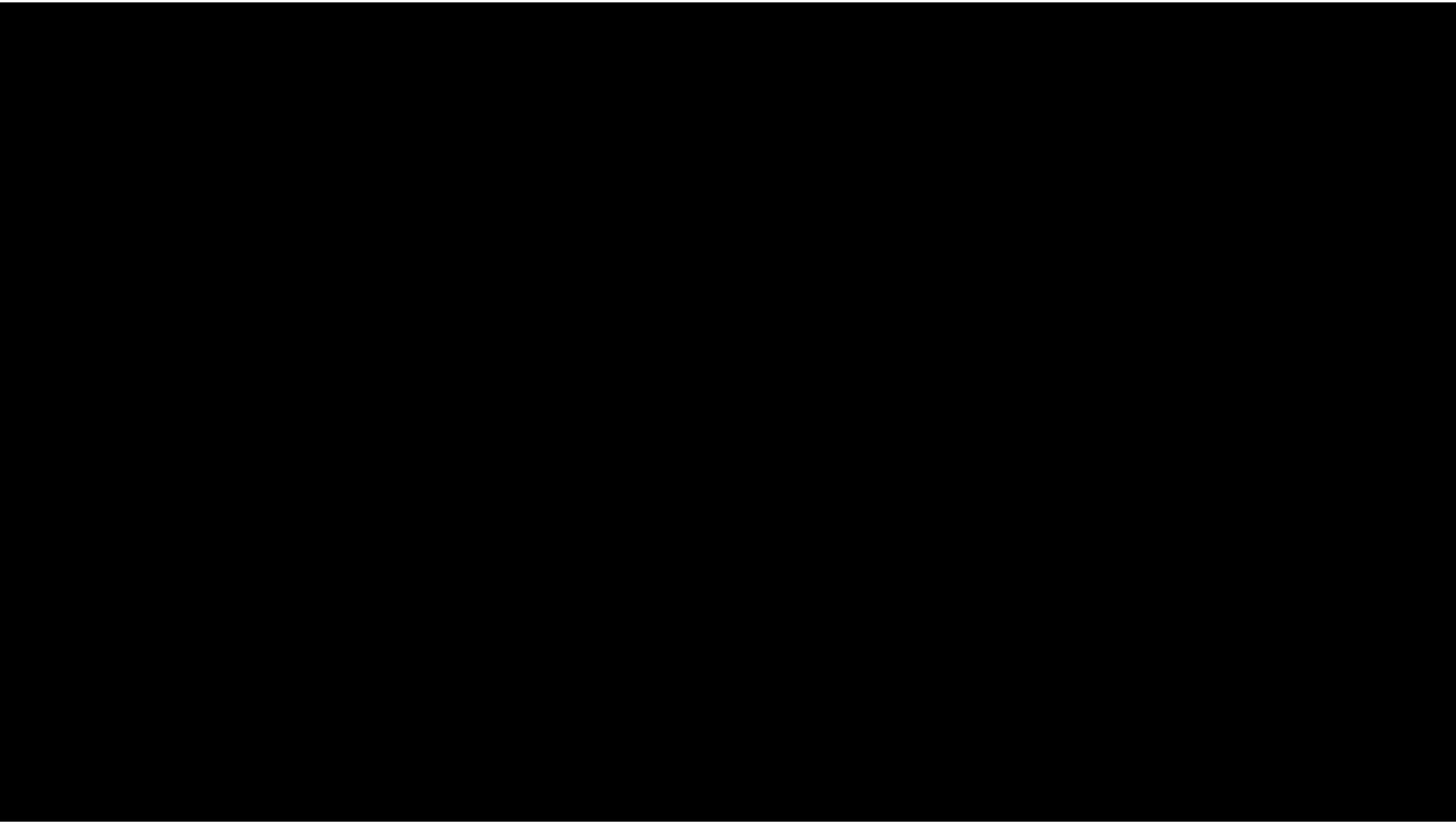
Intel Embree

NVIDIA OptiX



Hardware-accelerated ray tracing (RTX)

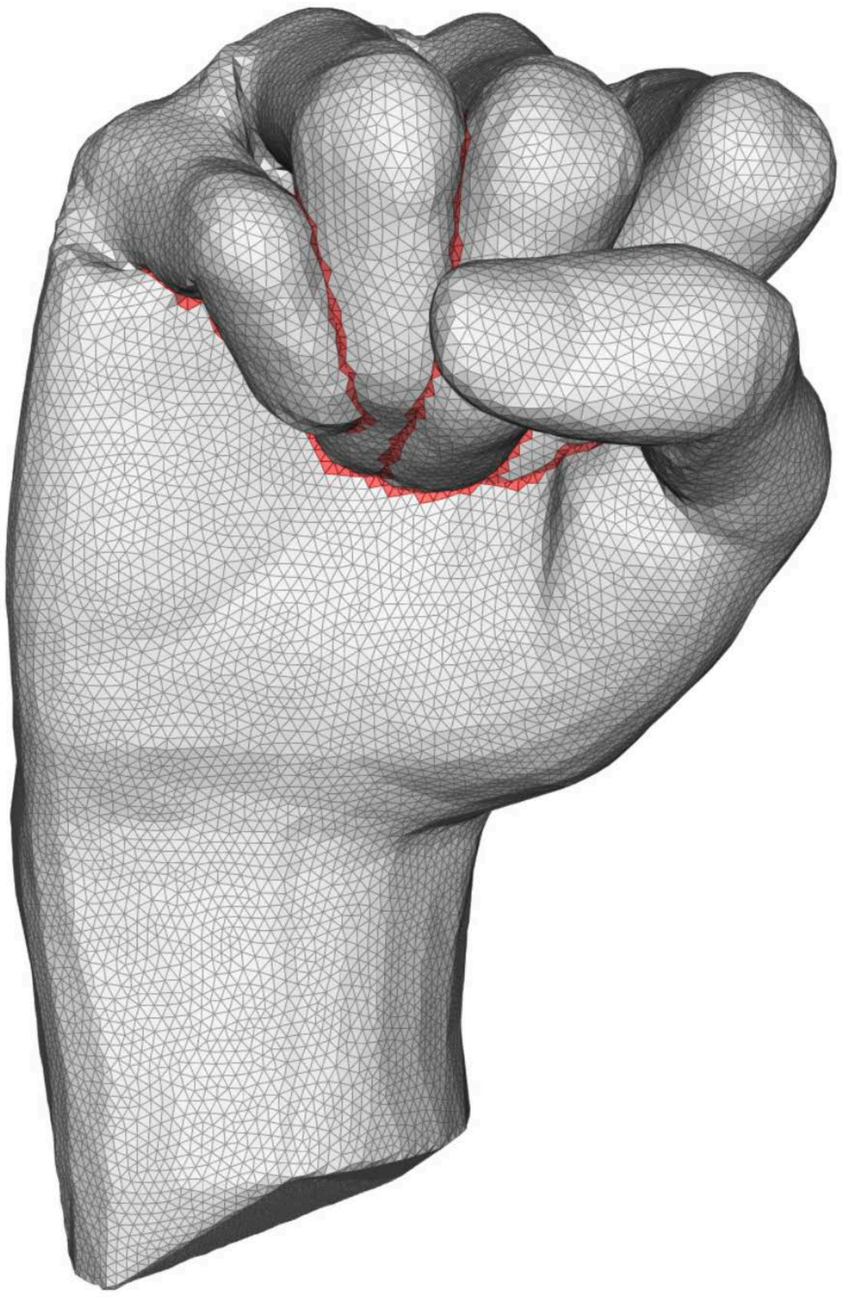
High-performance ray tracing



NVIDIA's "Project Sol" — real time ray tracing demo (RTX)

One more query: mesh-mesh intersection

- **GEOMETRY: How do we know if a mesh intersects itself?**
- **ANIMATION: How do we know if a collision occurred?**



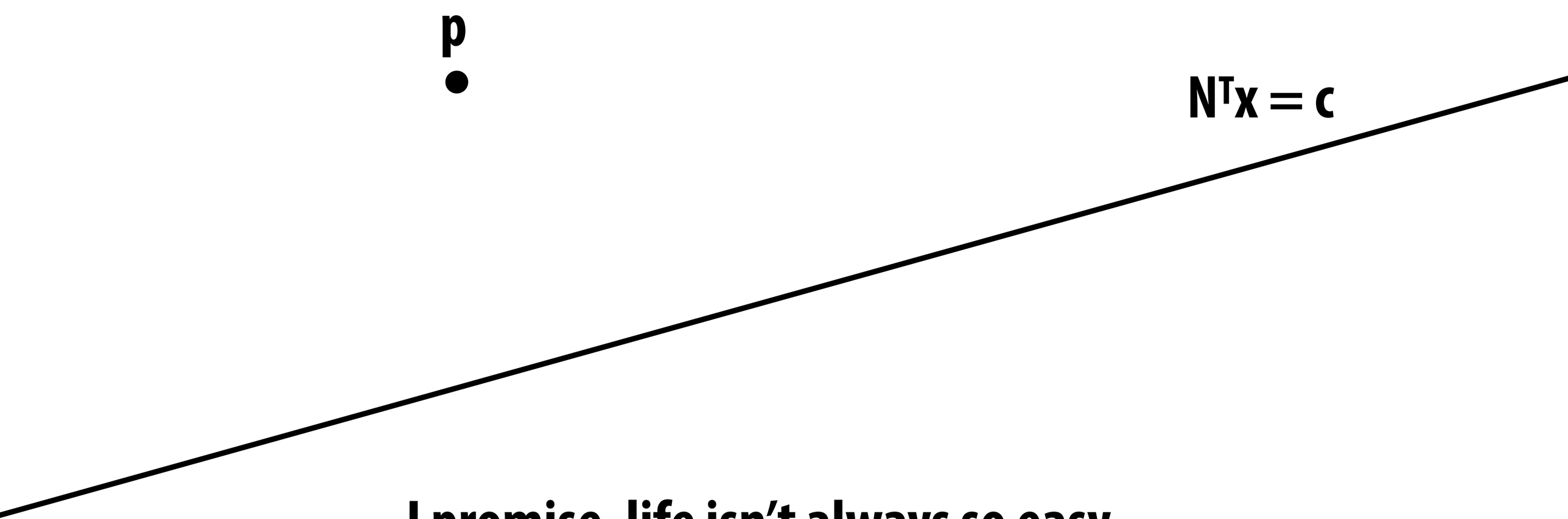
Warm up: point-point intersection

- Q: How do we know if p intersects a?
- A: ...check if they're the same point!

Sadly, life is not always so easy.

Slightly harder: point-line intersection

- Q: How do we know if a point intersects a given line?
- A: ...plug it into the line equation!



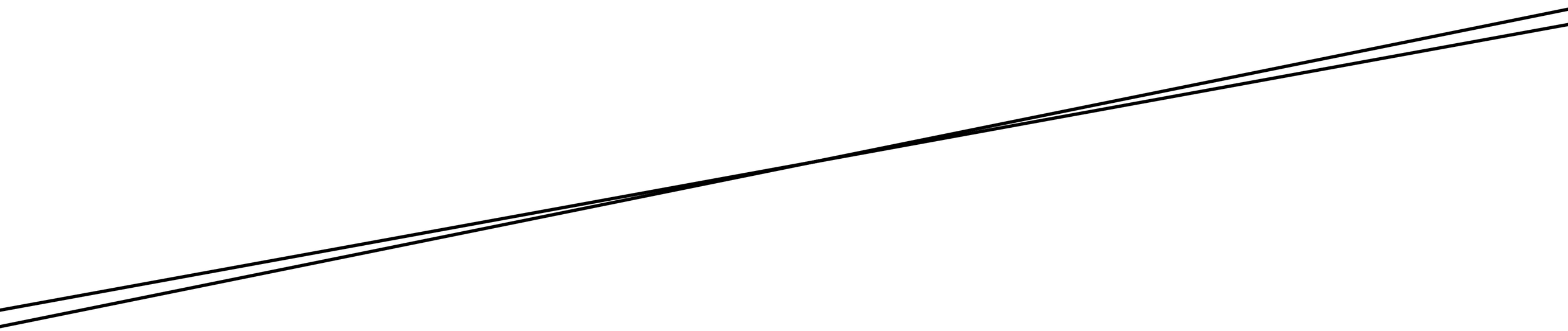
I promise, life isn't always so easy.

Finally interesting: line-line intersection

- Two lines: ax=b and cx=d
- Q: How do we find the intersection?
- A: See if there is a simultaneous solution
- Leads to linear system: $\begin{bmatrix} a_1 & a_2 \\ c_1 & c_2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} b \\ d \end{bmatrix}$

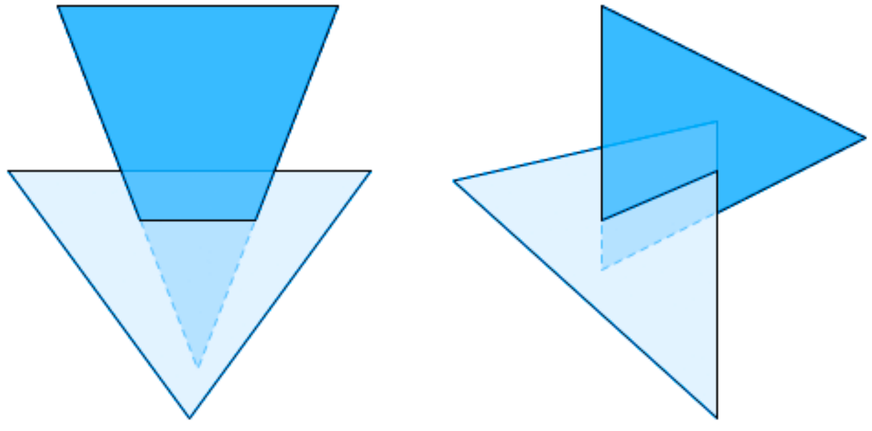
Degenerate line-line intersection?

- What if lines are almost parallel?
- Small change in normal can lead to big change in intersection!
- Instability very common, very important with geometric predicates. Demands special care (e.g., analysis of matrix).

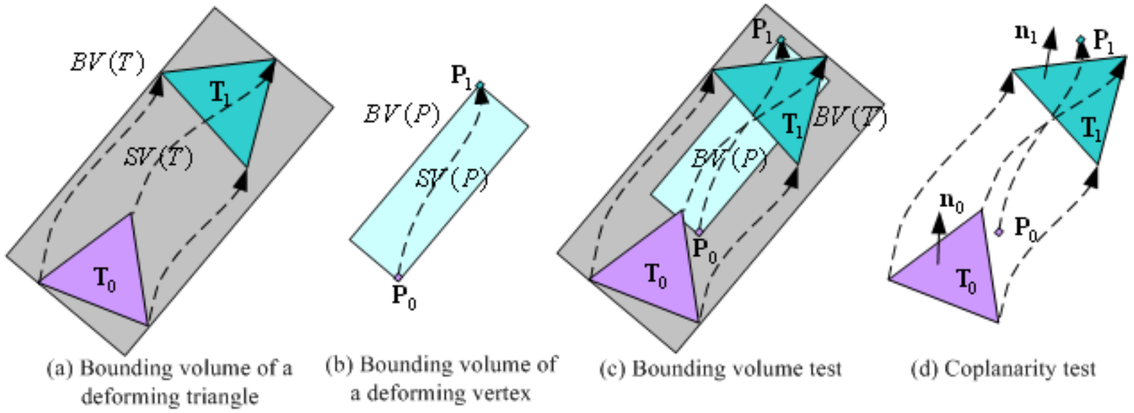


See for example Shewchuk, "*Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates*"

Triangle-Triangle Intersection?



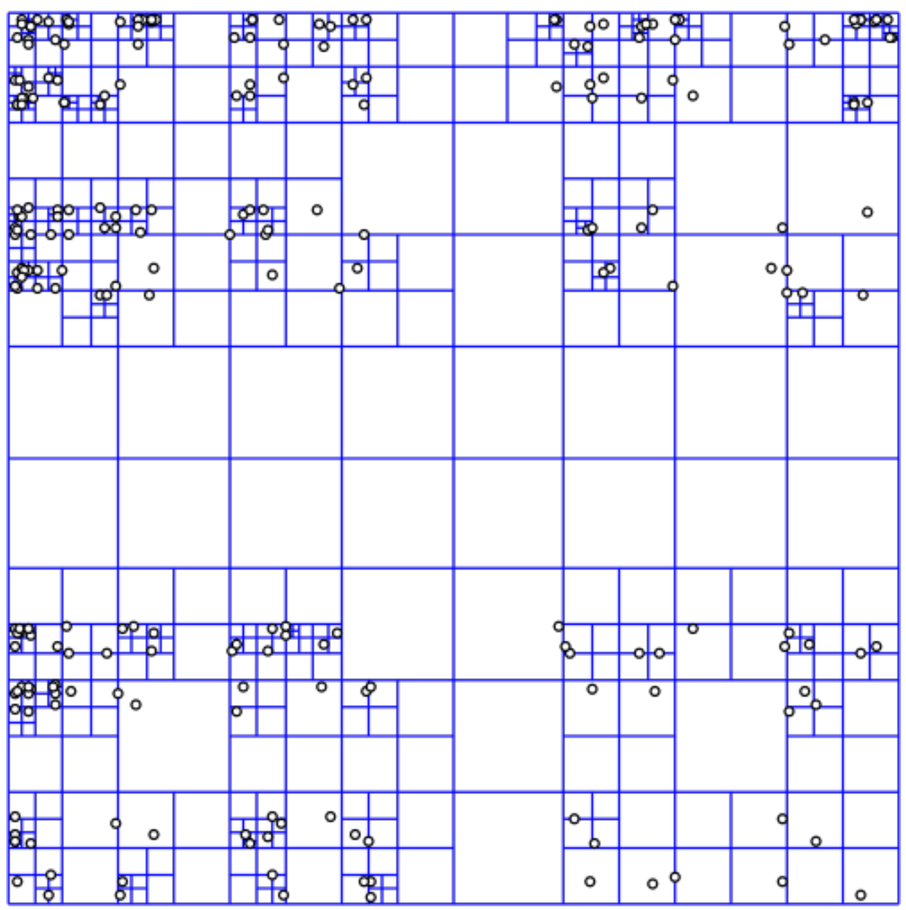
- Lots of ways to do it
- Basic idea:
  - Q: Any ideas?
  - One way: reduce to edge-triangle intersection
  - Check if each line passes through plane
  - Then do interval test
- What if triangle is *moving*?
  - Important case for animation
  - Can think of triangles as *prisms* in time
  - Turns dynamic problem (nD + time) into purely geometric problem in (n+1)-dimensions

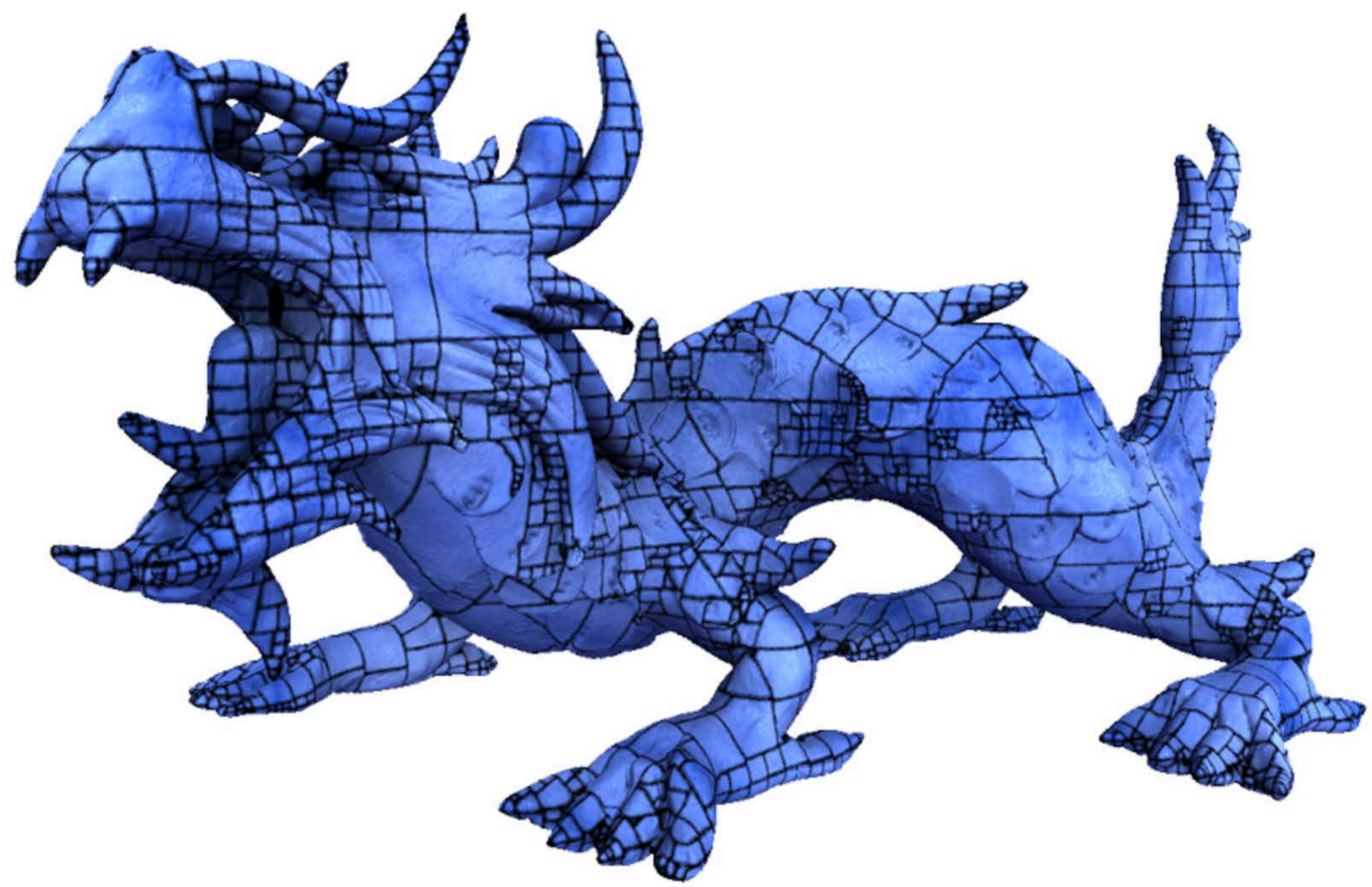


(a) Bounding volume of a deforming triangle (b) Bounding volume of a deforming vertex (c) Bounding volume test (d) Coplanarity test

Up Next: Spatial Acceleration Data Strucutres

- Testing every element is *slow!*
- E.g., linearly scanning through a list vs. binary search
- Can apply this same kind of thinking to geometric queries





Spatial Data Structures

Computer Graphics
CMU 15-462/662

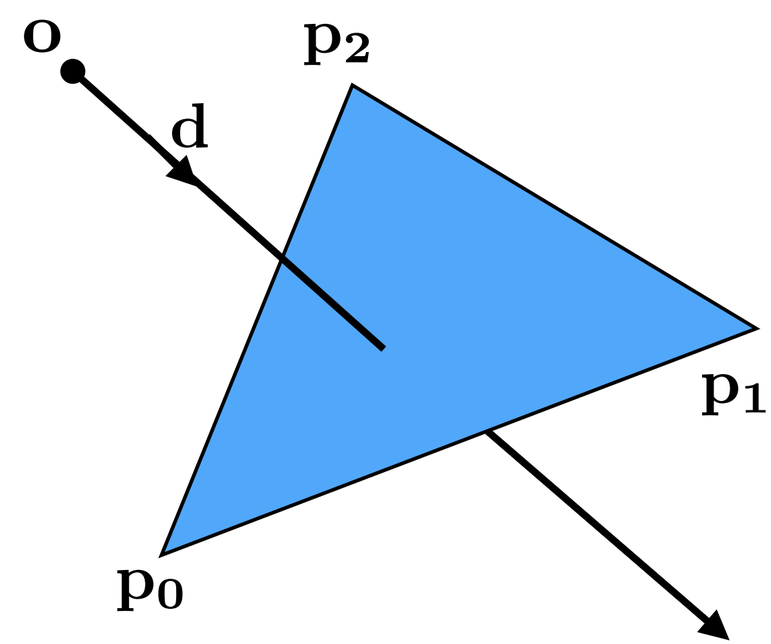
Complexity of geometry

How can we <u>efficiently</u> perform a geometric query on a scene of this complexity?

Important use case: ray tracing

Review: ray-triangle intersection

- **Find ray-plane intersection**



Parametric equation of a ray:

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

ray origin

normalized ray direction

Plug equation for ray into implicit plane equation:

$$\mathbf{N}^{\mathbf{T}}\mathbf{x} = c$$

$$\mathbf{N}^{\mathbf{T}}(\mathbf{o} + t\mathbf{d}) = c$$

Solve for *t* corresponding to intersection point:

$$t = \frac{c - \mathbf{N}^{\mathbf{T}}\mathbf{o}}{\mathbf{N}^{\mathbf{T}}\mathbf{d}}$$

- **Determine if point of intersection is within triangle**

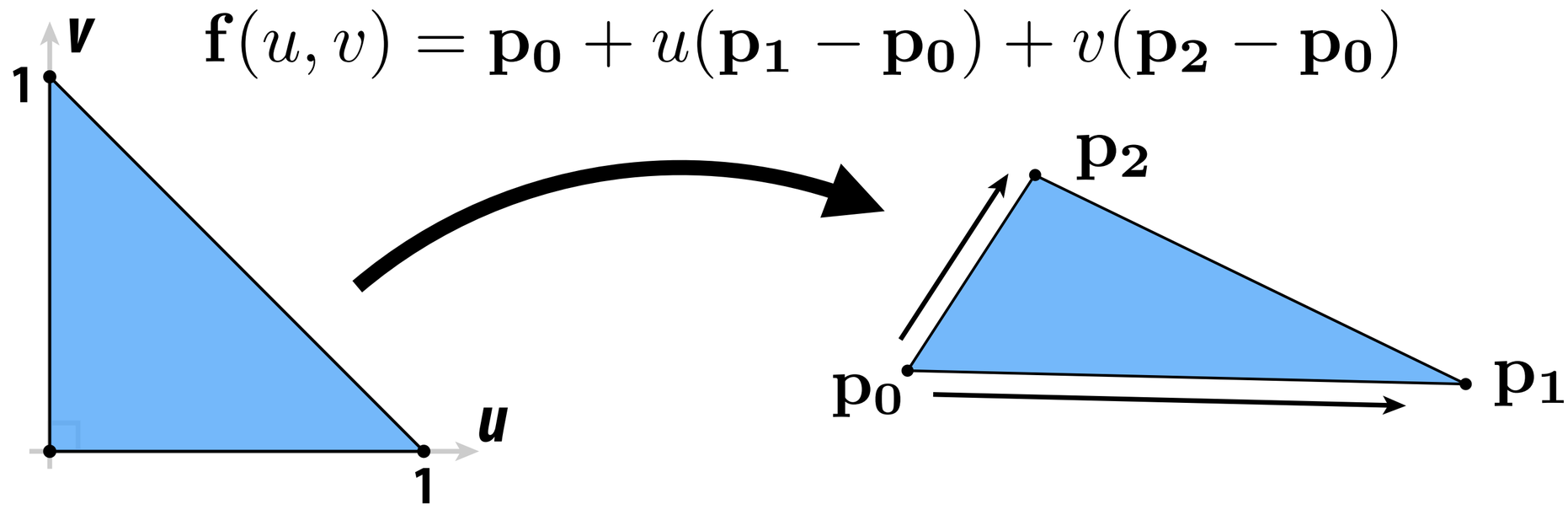
Ray-triangle intersection—a different way

- **Parameterize triangle given by vertices** $\mathbf{p_0}, \mathbf{p_1}, \mathbf{p_2}$ **using barycentric coordinates**

$$f(u,v) = (1-u-v)\mathbf{p_0} + u\mathbf{p_1} + v\mathbf{p_2}$$

- **Can think of a triangle as an affine map of the unit triangle**

$$\mathbf{f}(u,v) = \mathbf{p_0} + u(\mathbf{p_1} - \mathbf{p_0}) + v(\mathbf{p_2} - \mathbf{p_0})$$



Ray-triangle intersection—a different way

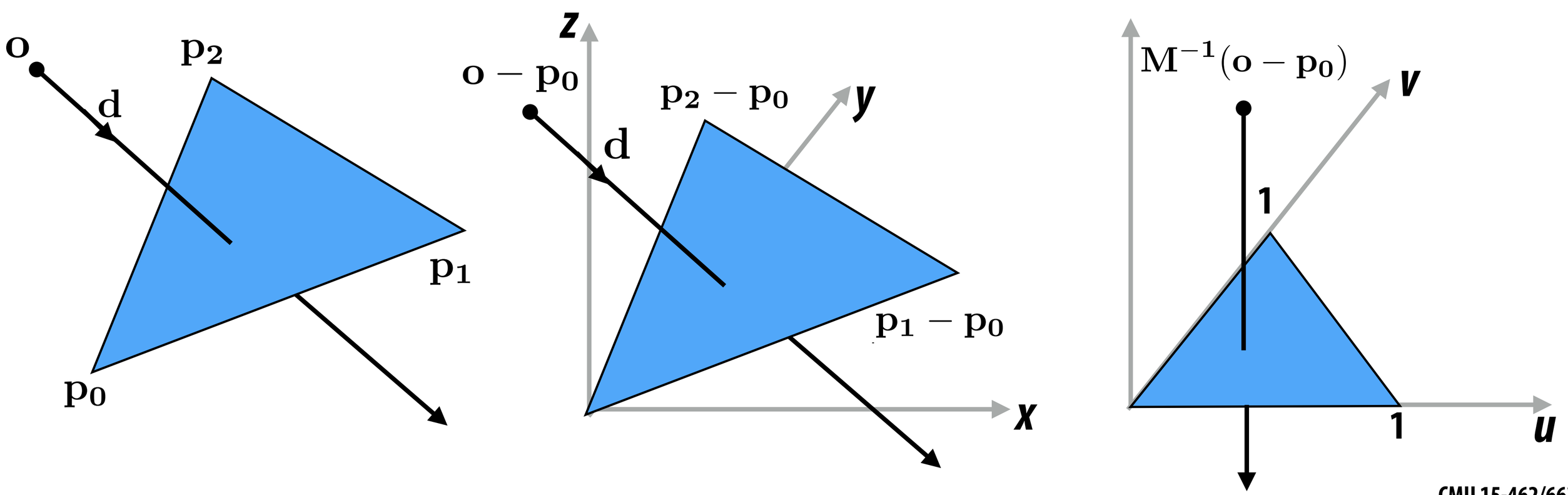
Plug parametric ray equation directly into equation for points on triangle:

$$\mathbf{p_0} + u(\mathbf{p_1} - \mathbf{p_0}) + v(\mathbf{p_2} - \mathbf{p_0}) = \mathbf{o} + t\mathbf{d}$$

Solve for u, v, t:

$$\underbrace{\begin{bmatrix} \mathbf{p_1} - \mathbf{p_0} & \mathbf{p_2} - \mathbf{p_0} & -\mathbf{d} \end{bmatrix}}_{\mathbf{M}} \begin{bmatrix} u \\ v \\ t \end{bmatrix} = \mathbf{o} - \mathbf{p_0}$$

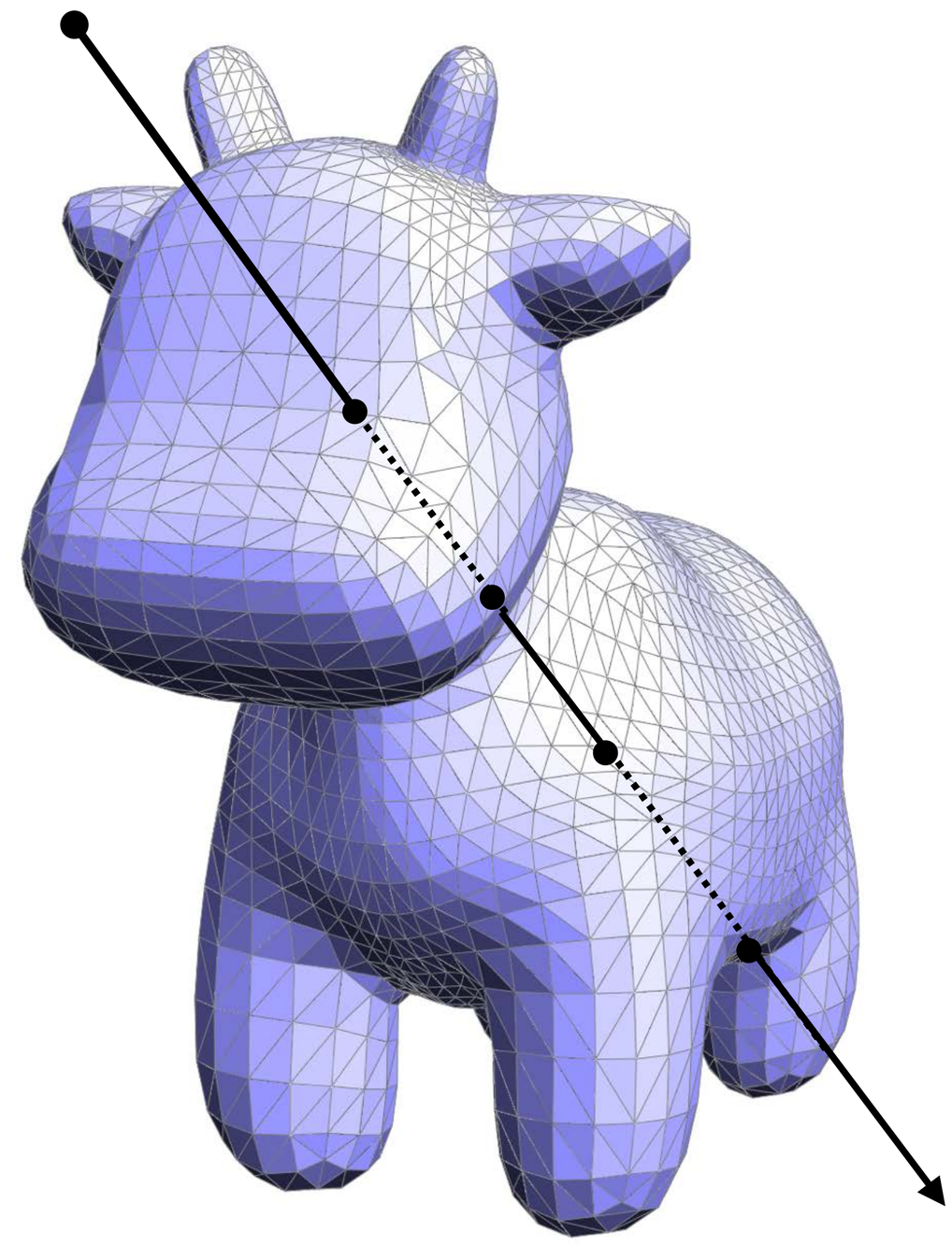
$\mathbf{M}^{-1}$ **transforms triangle back to unit triangle in u,v plane, and transforms ray's direction to be orthogonal to plane**



First Hit Problem

Given a scene defined by a set of *N* primitives and a ray *r*, find the <u>closest</u> point of intersection of *r* with the scene

"Find the first primitive the ray hits"



Naïve algorithm?

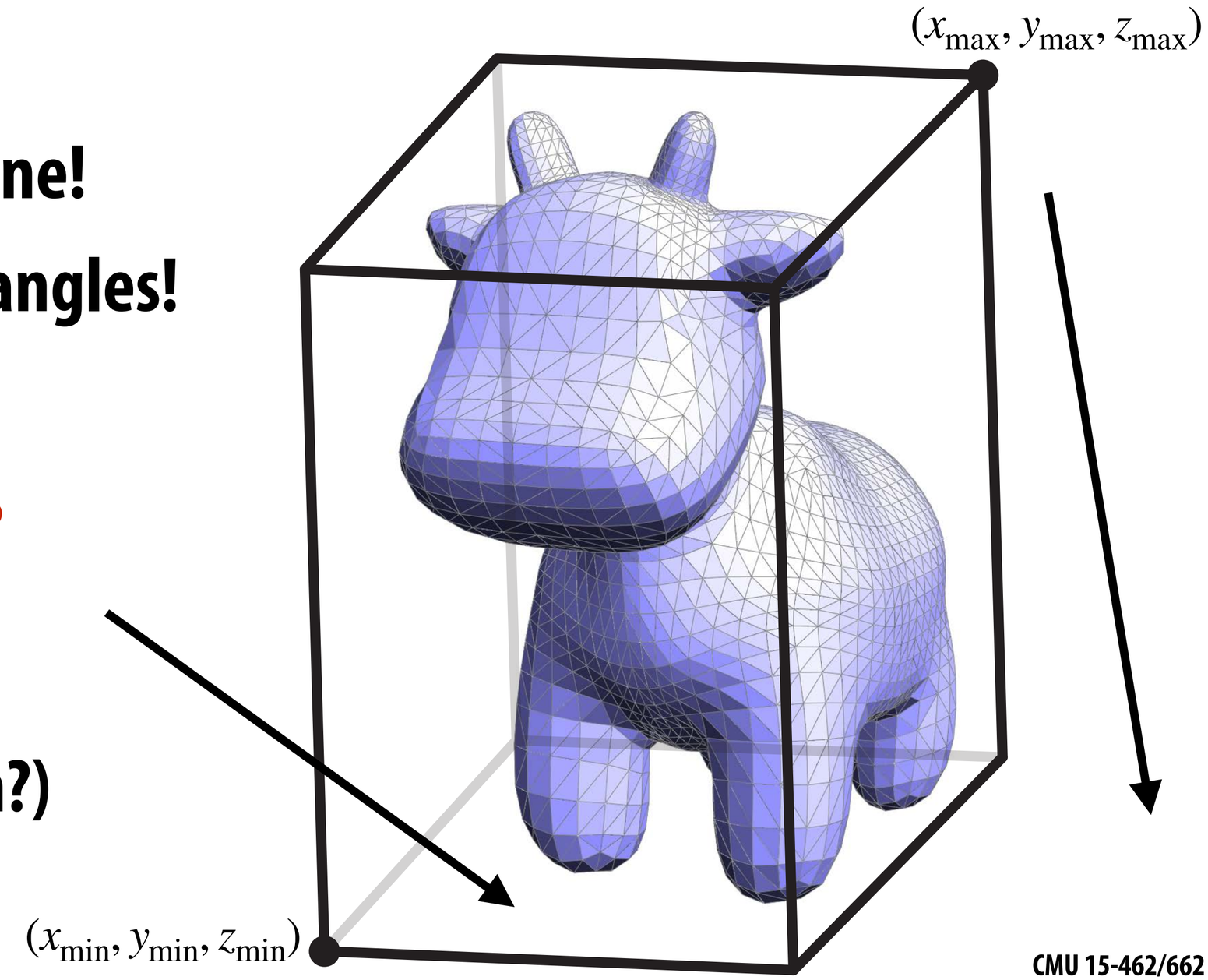
1. **Intersect ray with <u>every</u> triangle**
2. **Keep the closest hit point**

Complexity? $O(N)$

Can we do better?

Bounding Box

- Precompute smallest "bounding box" around all primitives
  - Q: How?
  - A: Loop over vertices; keep max/min (x,y,z) coordinates
- Intersect ray with box
  - If it misses, we're done!
  - If it hits…try all triangles!



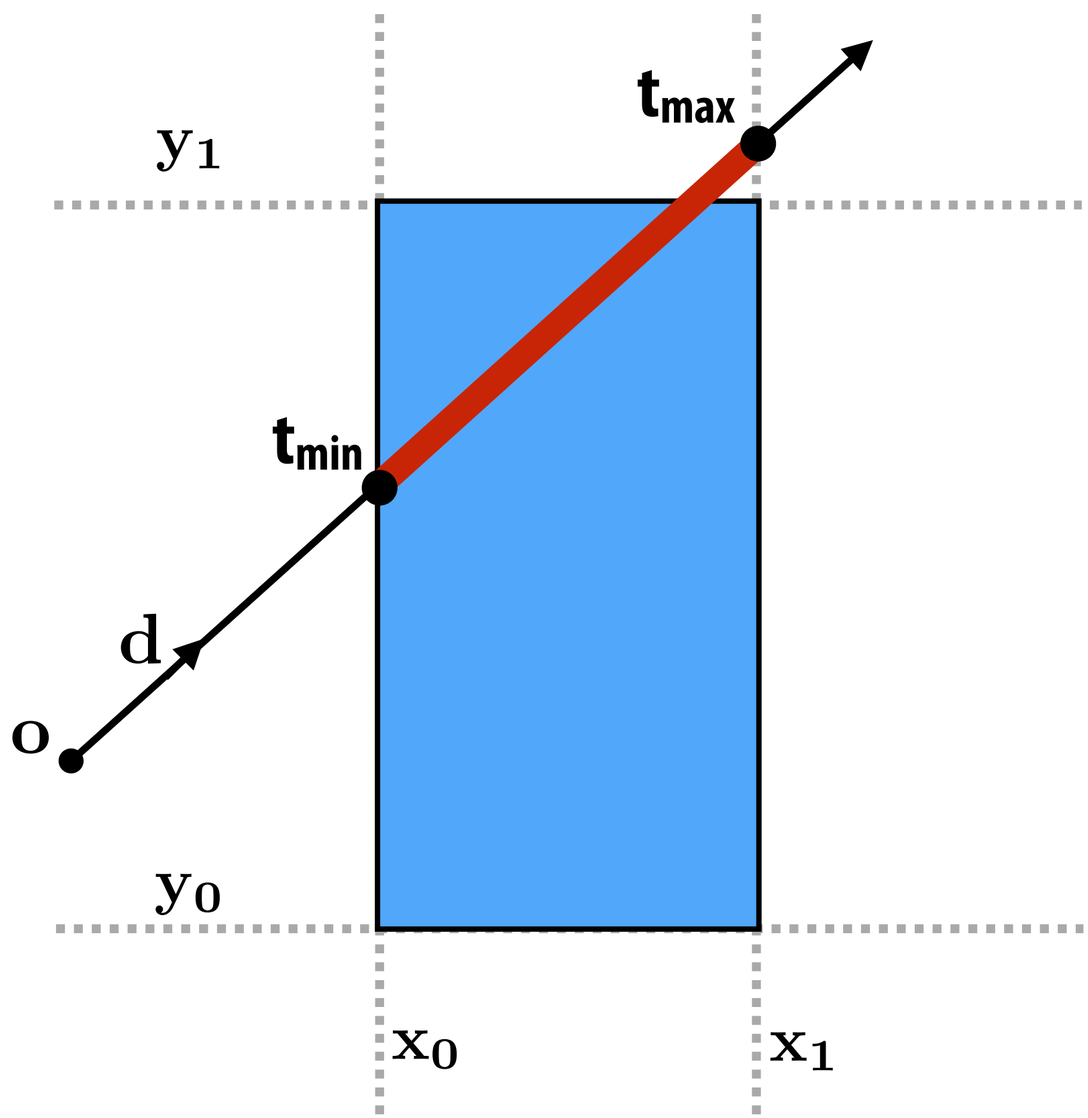
Did we actually do better?

No! Worst case is still O(N)

(Also: ray-box intersection?)

Ray-axis-aligned-box intersection

What is ray's closest/farthest intersection with axis-aligned box?



Find intersection of ray with all planes of box:

$$\mathbf{N}^{\mathbf{T}}(\mathbf{o} + t\mathbf{d}) = c$$

Math simplifies greatly since plane is axis aligned (consider x=x_0 plane in 2D):

$$\mathbf{N}^{\mathbf{T}} = \begin{bmatrix} 1 & 0 \end{bmatrix}^T$$

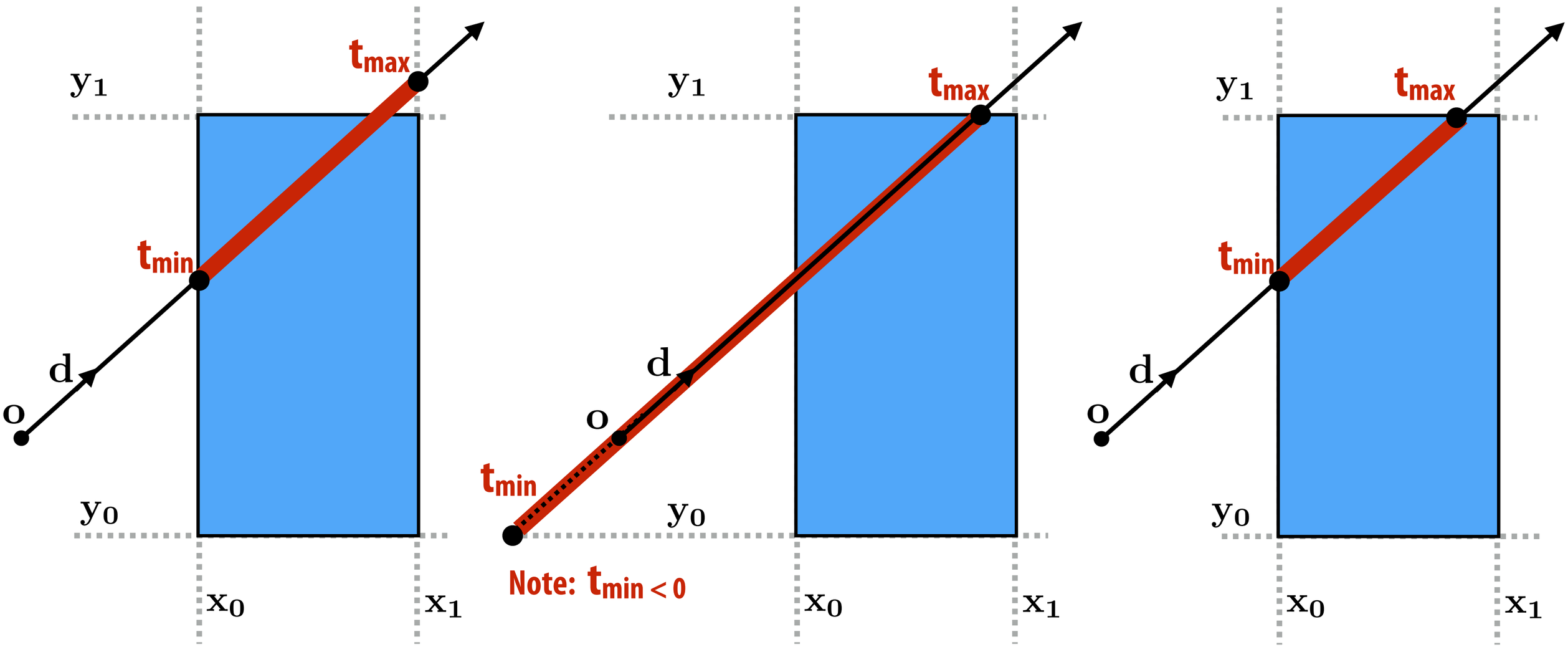
$$c = x_0$$

$$t = \frac{x_0 - \mathbf{o_x}}{\mathbf{d_x}}$$

Figure shows intersections with x=x_0 and x=x_1 planes.

Ray-axis-aligned-box intersection

Compute intersections with all planes, take intersection of t_{min}/t_{max} intervals



Intersections with x planes

Intersections with y planes

Final intersection result

How do we know when the ray misses the box?

Ok, but we still didn't make it any faster!

How do we speed things up?

A simpler problem…

- Imagine I have a set of integers S
- Given an integer, say *k*=18, find the element of S closest to *k*:

10 123 2 100 6 25 64 11 200 30 950 111 20 8 1 80

What's the cost of finding *k* in terms of the size N of the set?

Can we do better?

Suppose we first *sort* the integers:

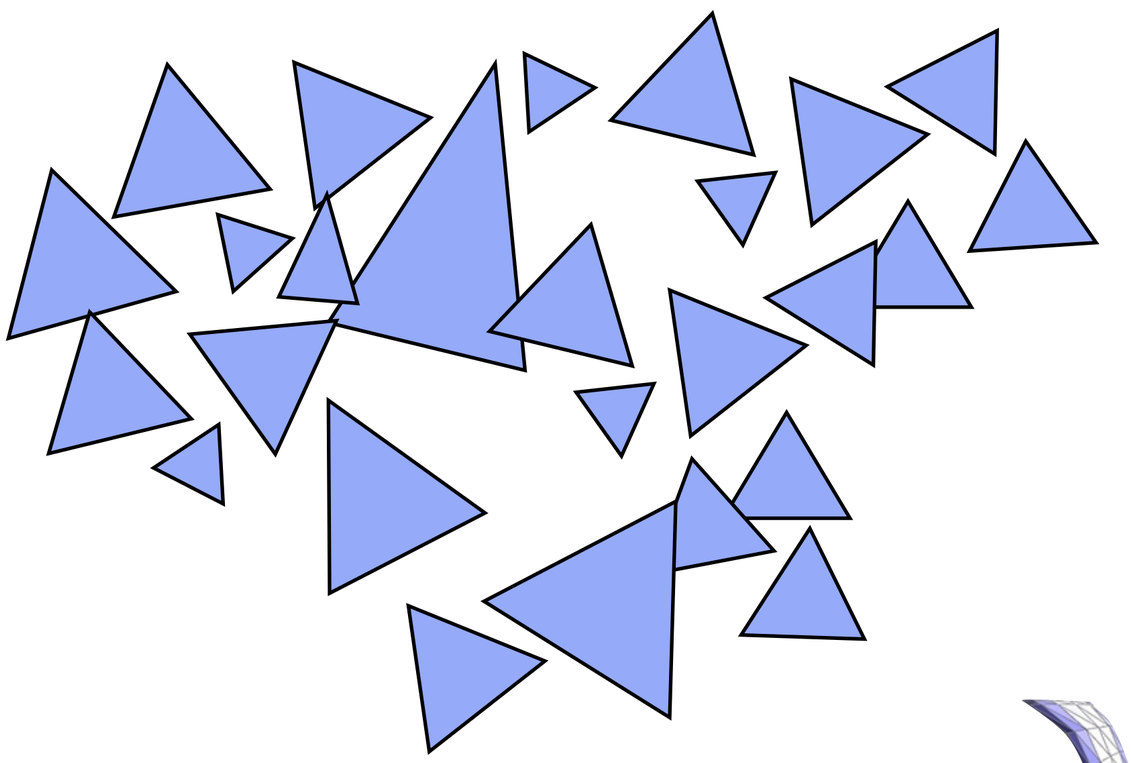
1 2 6 8 10 11 20 25 30 64 80 100 111 123 200 950

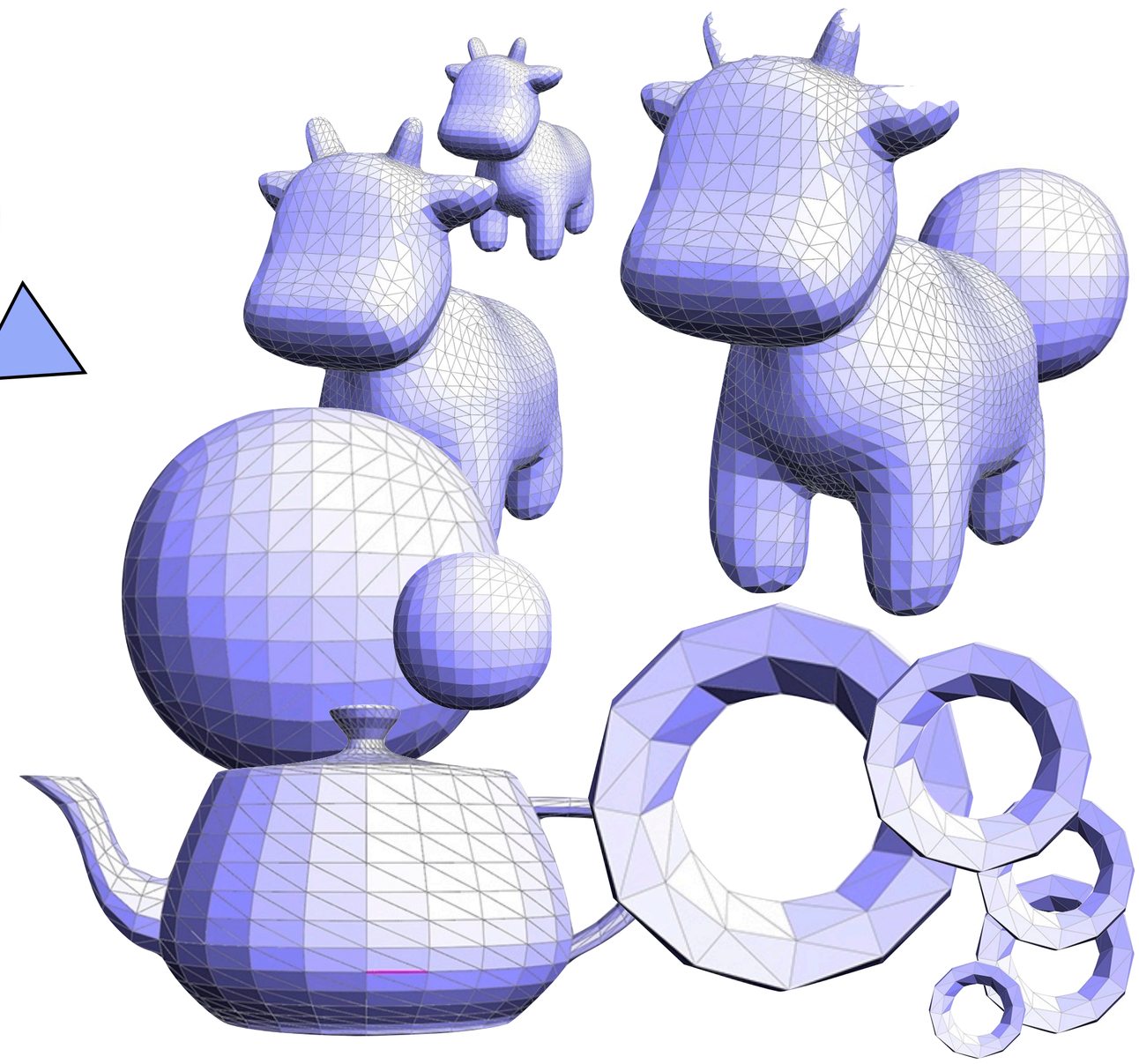
How much does it now cost to find k (*including sorting*)?

Cost for just ONE query: O(n log n) *worse* than before! :-)

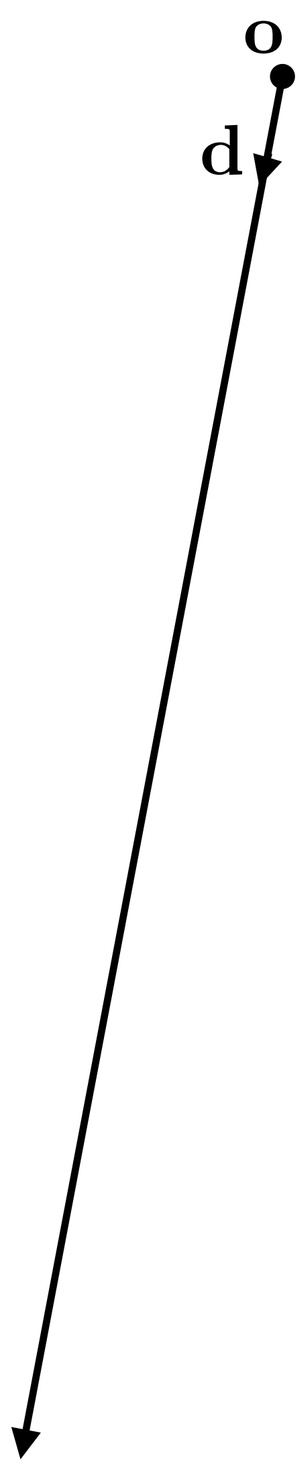
Amortized cost: O(log n) …*much* better!

Can we also reorganize scene primitives to enable fast ray-scene intersection queries?

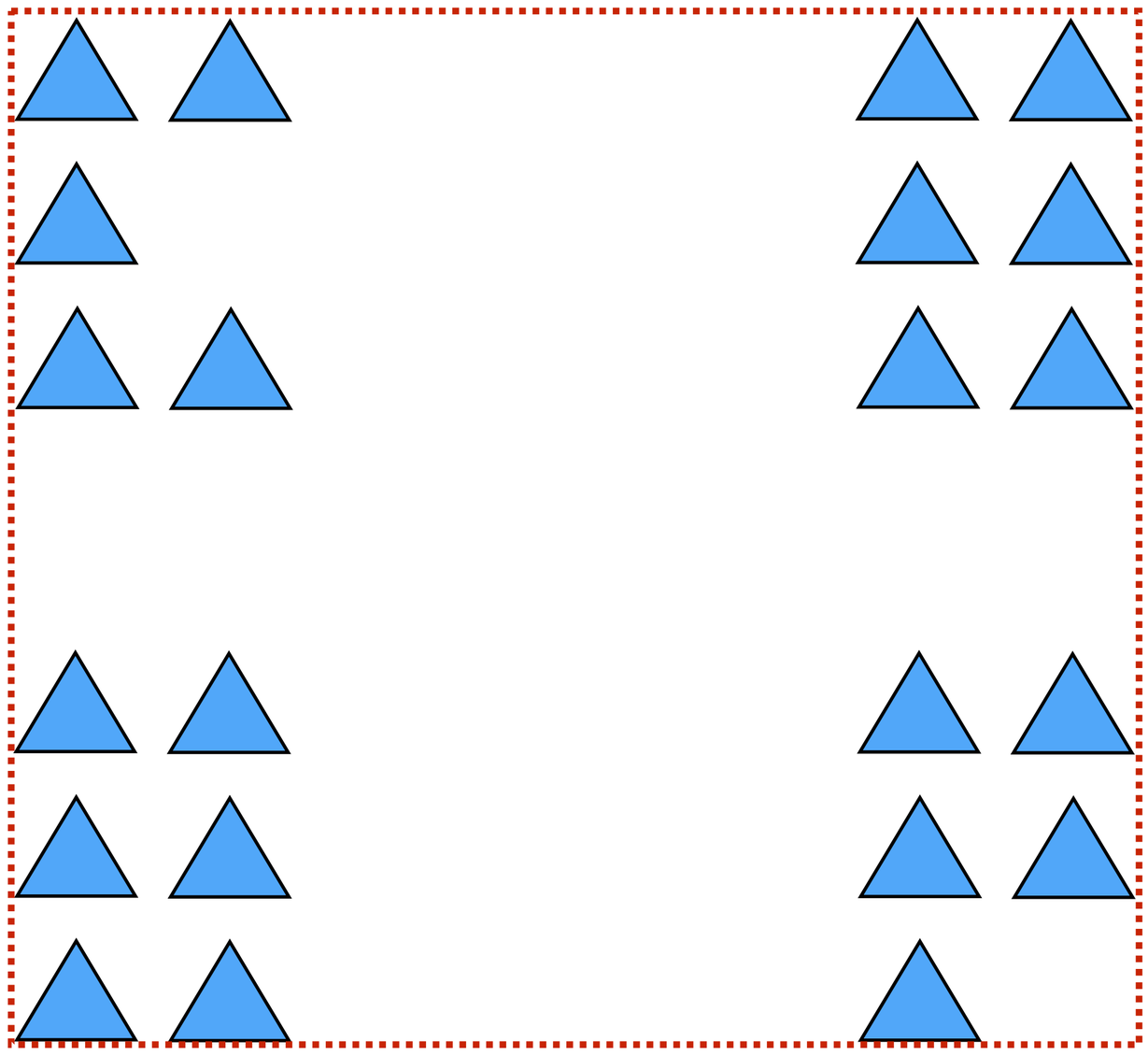




Simple case



Ray misses bounding box of all primitives in scene



<u>Cost (misses box):</u>

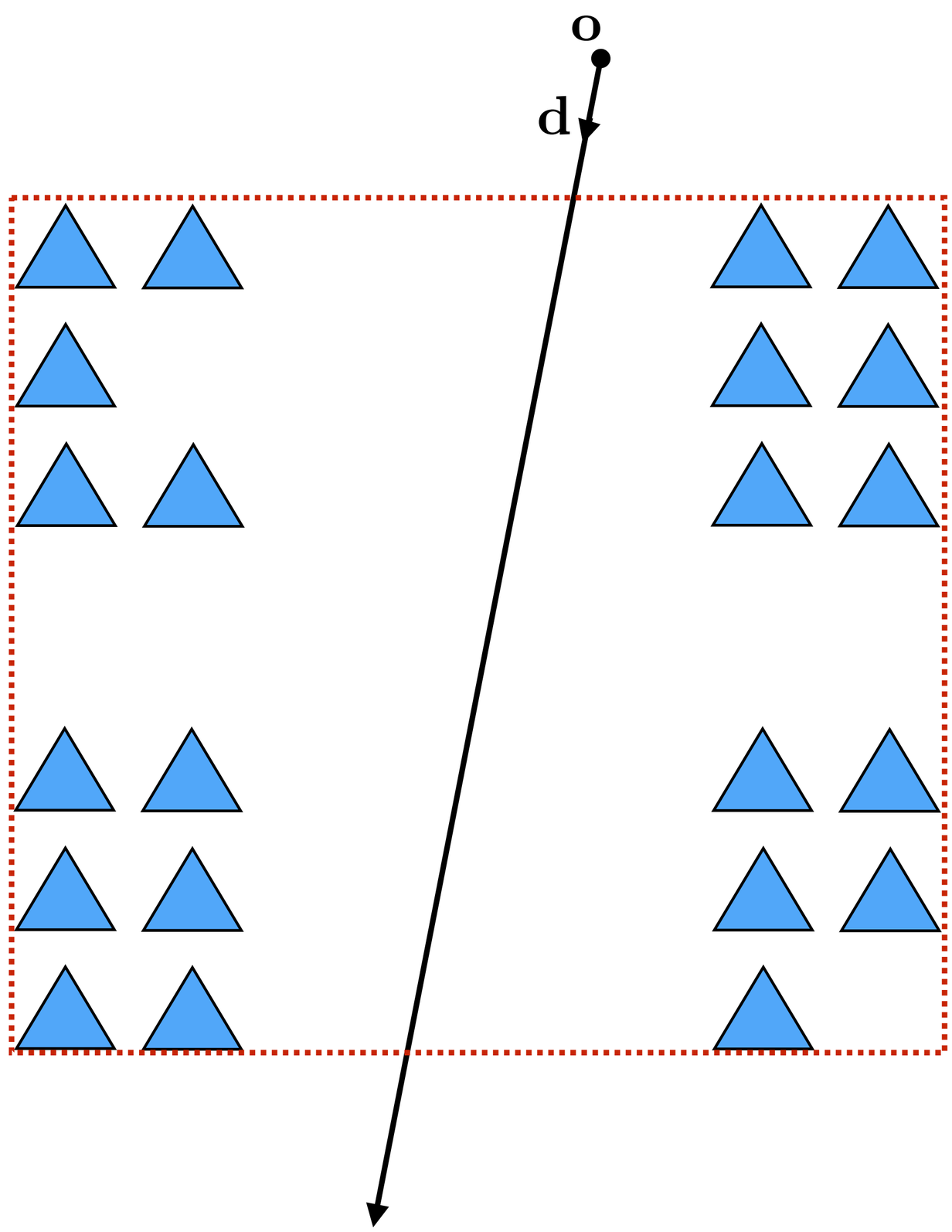
preprocessing: O(n)

ray-box test: O(1)

amortized cost*: O(1)

*over *many* ray-scene intersection tests

Another (should be) simple case



Cost (hits box):

preprocessing: O(n)
ray-box test: O(1)
triangle tests: O(n)
amortized cost*: O(n)

Still no better than naïve algorithm (test all triangles)!

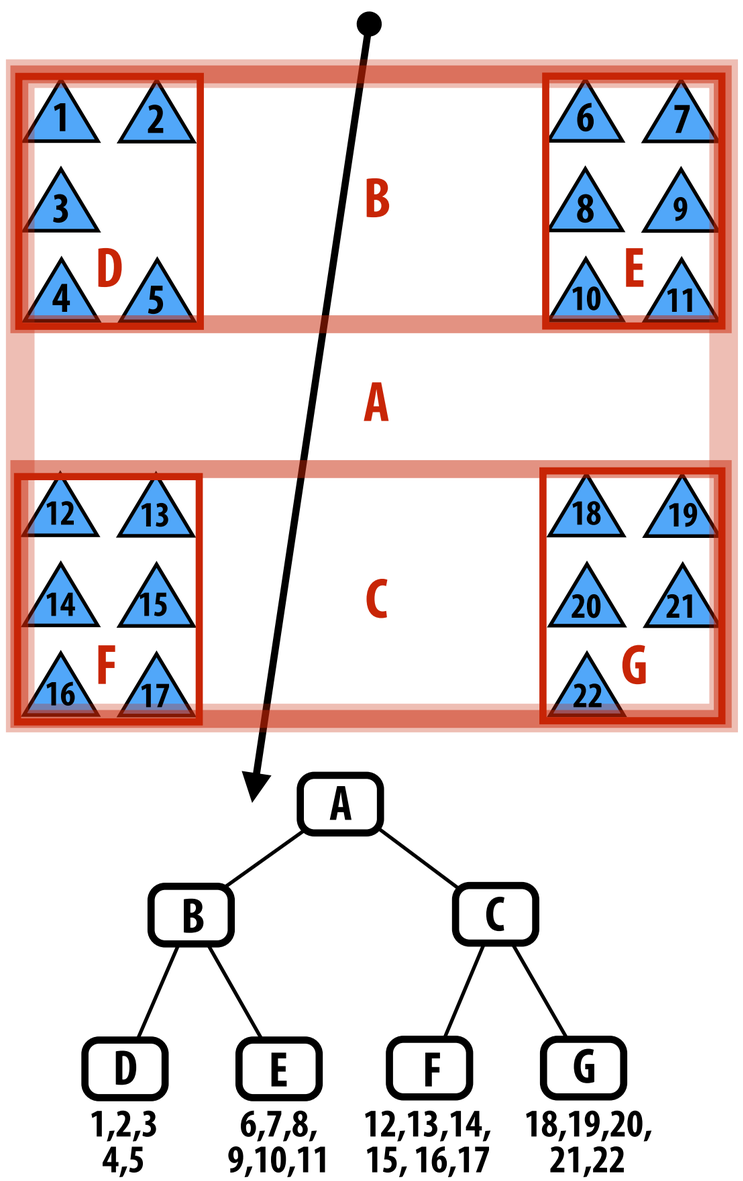
*over *many* ray-scene intersection tests

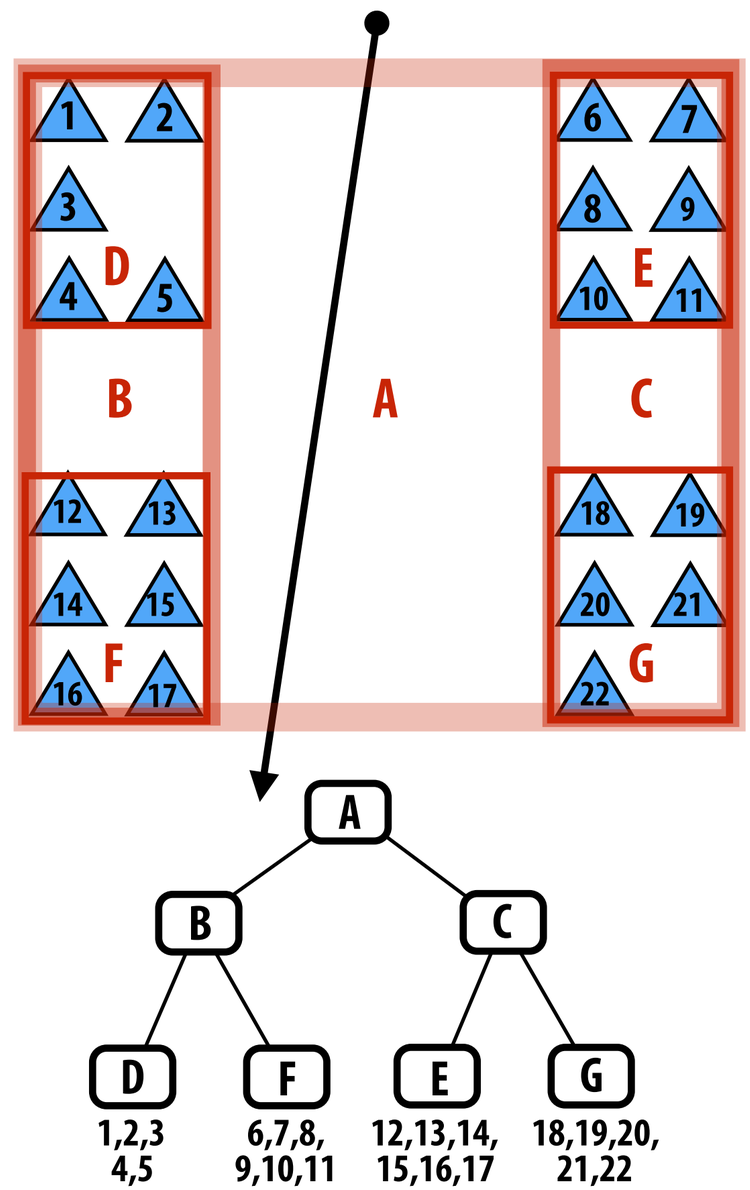
Q: How can we do better?

A: Apply this strategy hierarchically.

Bounding volume hierarchy (BVH)

- **Leaf nodes:**
  - Contain *small* list of primitives
- **Interior nodes:**
  - Proxy for a *large* subset of primitives
  - Stores bounding box for all primitives in subtree



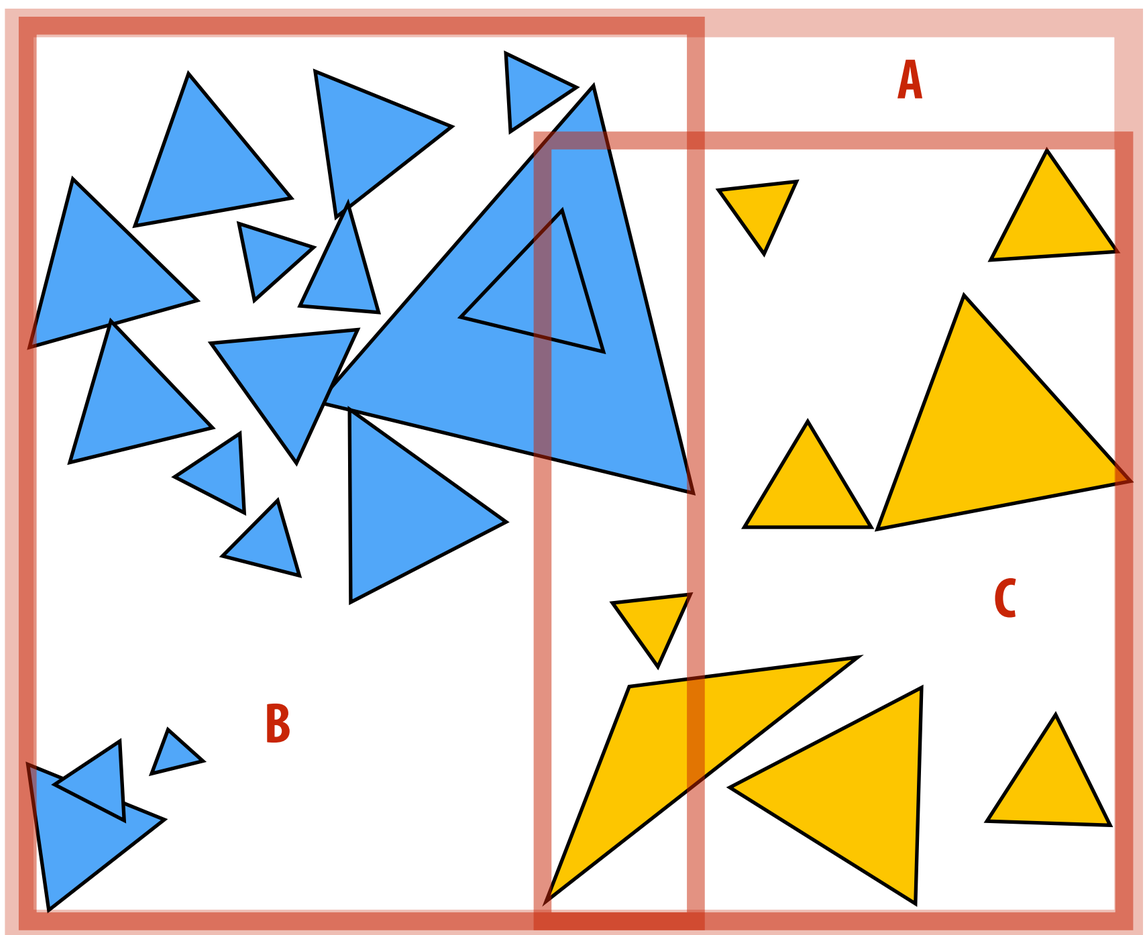


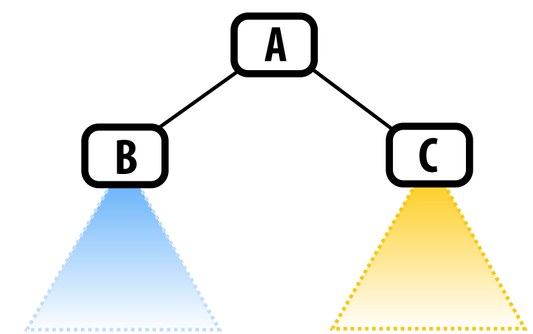
Left: two different BVH organizations of the same scene containing 22 primitives.

Is one BVH better than the other?

Another BVH example

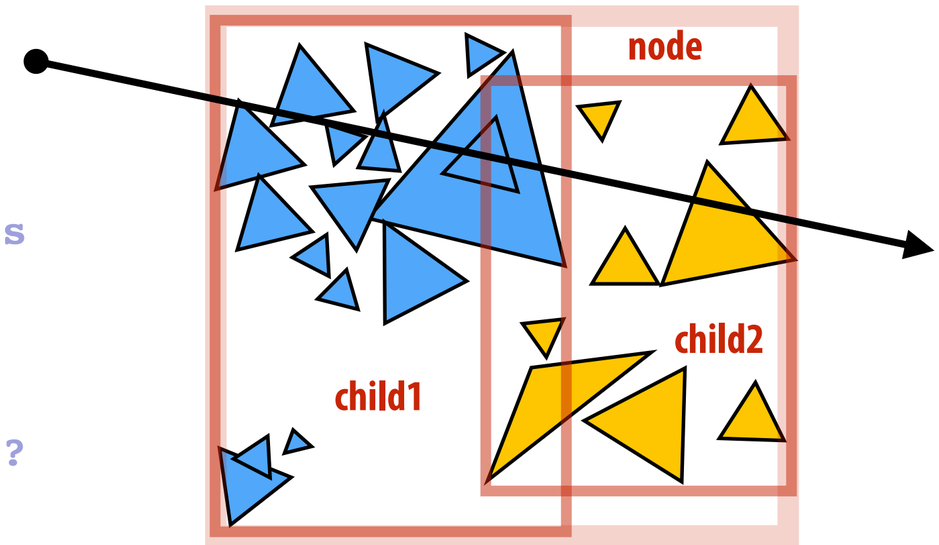
- BVH partitions each node's primitives into disjoints sets
  - Note: The sets can still be overlapping in space (below: child bounding boxes may overlap in space)





Ray-scene intersection using a BVH

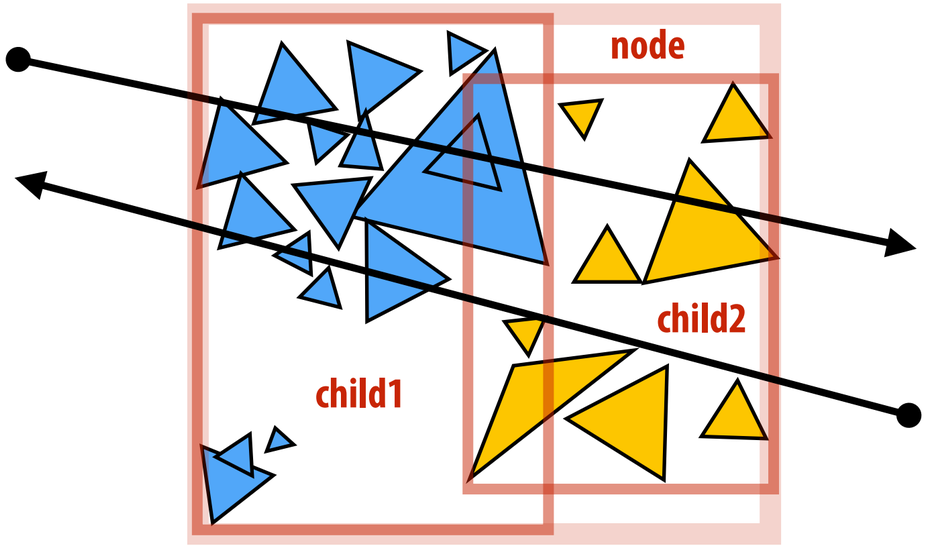
```
struct BVHNode {
   bool leaf;  // am I a leaf node?
   BBox bbox;  // min/max coords of enclosed primitives
   BVHNode* child1; // "left" child (could be NULL)
   BVHNode* child2; // "right" child (could be NULL)
   Primitive* primList; // for leaves, stores primitives
};

struct HitInfo {
   Primitive* prim;  // which primitive did the ray hit?
   float t;   // at what t value?
};

void find_closest_hit(Ray* ray, BVHNode* node, HitInfo* closest) {
   HitInfo hit = intersect(ray, node->bbox);  // test ray against node's bounding box
   if (hit.prim == NULL || hit.t > closest.t))
      return; // don't update the hit record

   if (node->leaf) {
      for (each primitive p in node->primList) {
         hit = intersect(ray, p);
         if (hit.prim != NULL && hit.t < closest.t) {
            closest.prim = p;
            closest.t = t;
         }
      }
   } else {
      find_closest_hit(ray, node->child1, closest);
      find_closest_hit(ray, node->child2, closest);
   }}
```



Improvement: "front-to-back" traversal

General strategy for improving performance:

Do traversal in a way that is likely to terminate "early"




```
void find_closest_hit(Ray* ray, BVHNode* node, HitInfo* closest)
{
   if (node->leaf) {
      // same as before
   } else {
      HitInfo hit1 = intersect(ray, node->child1->bbox);
      HitInfo hit2 = intersect(ray, node->child2->bbox);

      NVHNode* first = (hit1.t <= hit2.t) ? child1 : child2;
      NVHNode* second = (hit2.t <= hit1.t) ? child2 : child1;

      find_closest_hit(ray, first, closest);
      if (hit2.t < closest.t)
         find_closest_hit(ray, second, closest); // why might we still need to do this?
   }
}
```

"Front to back" traversal. Traverse to closest child node first. Why?

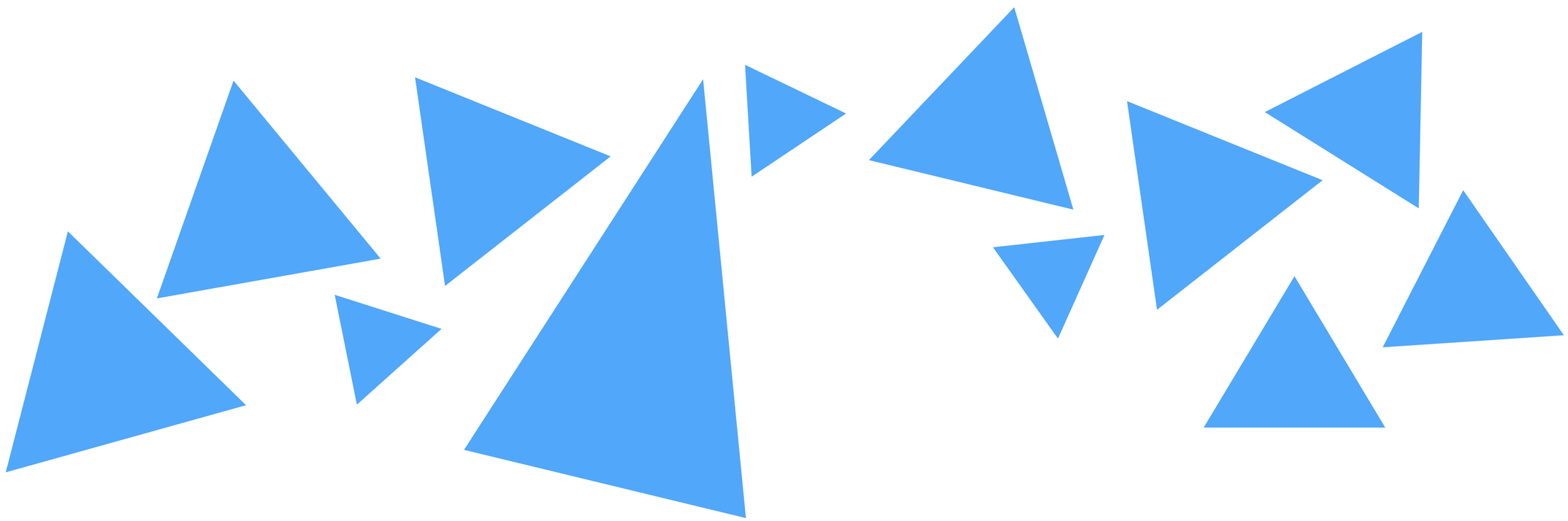
Other strategy for improving performance: Build a "better" BVH!

But for a given set of primitives, there are *many* possible BVHs…

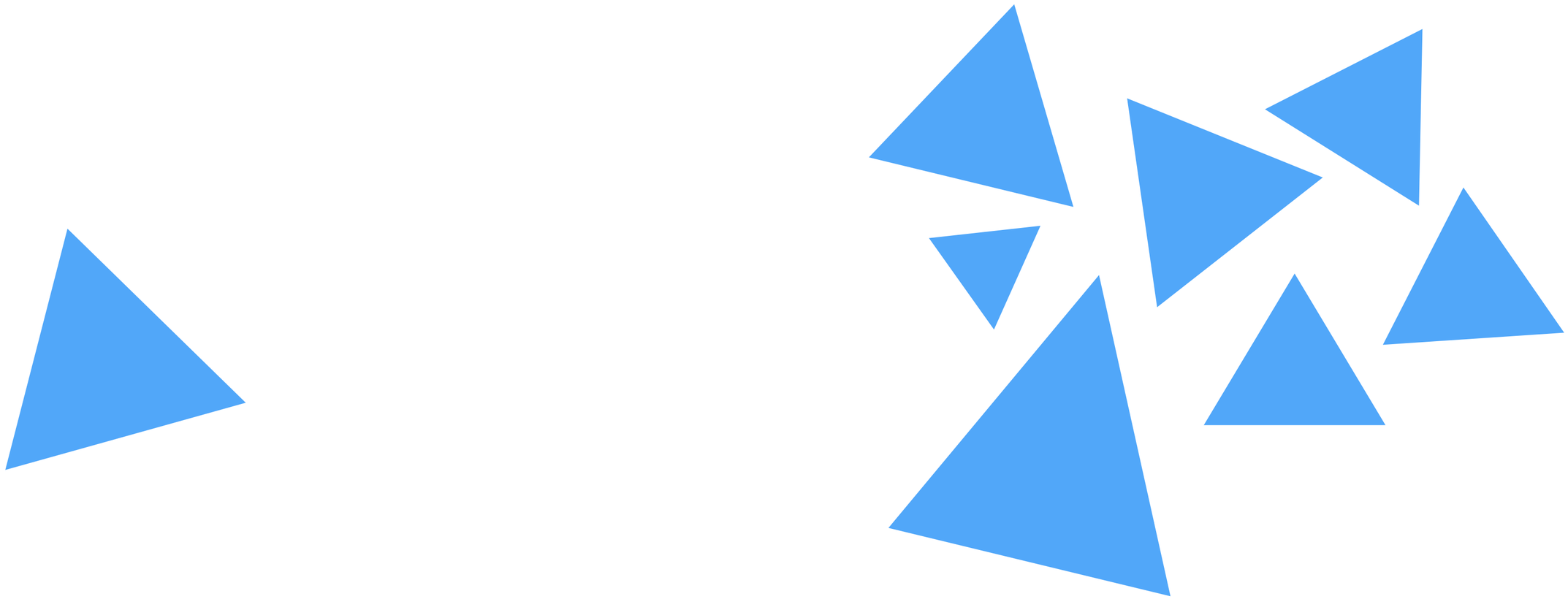
($2^N/2$ ways to partition N primitives into two groups)

Q: How do we <u>quickly</u> build a high-quality BVH?

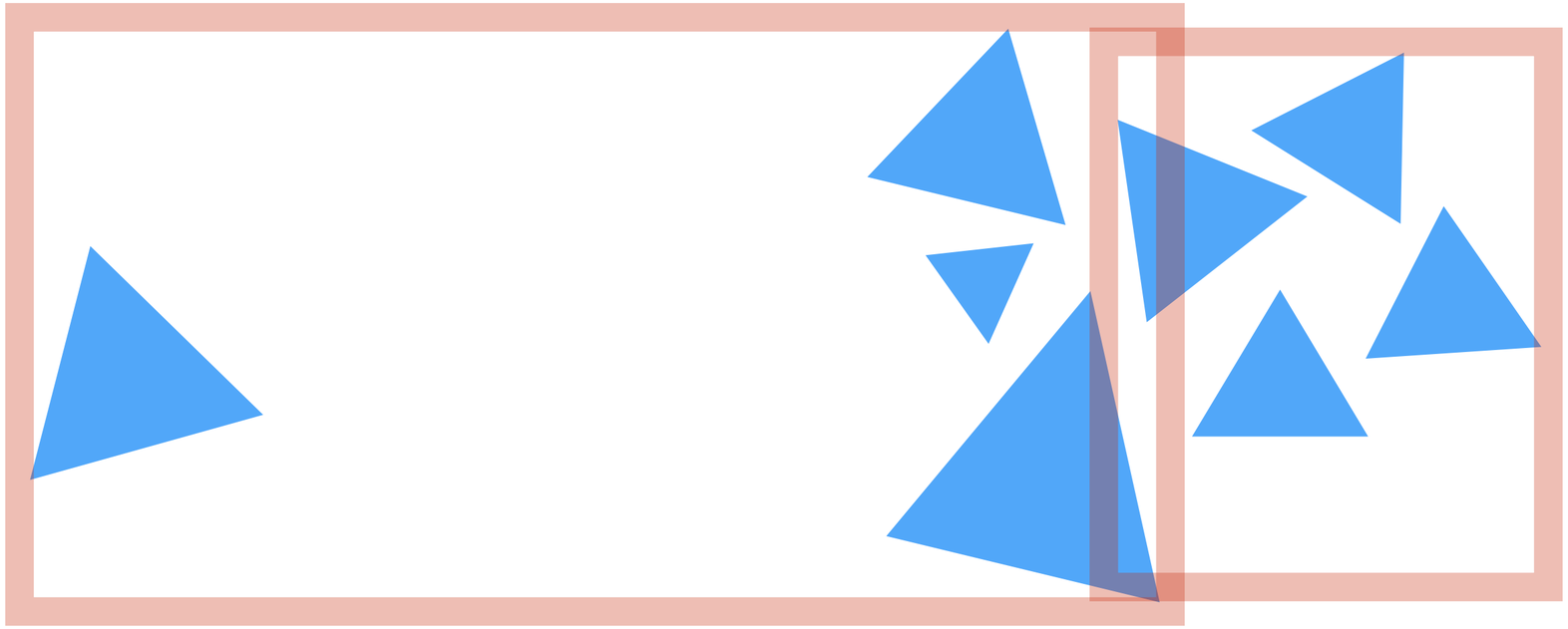
How would you partition these triangles into two groups?



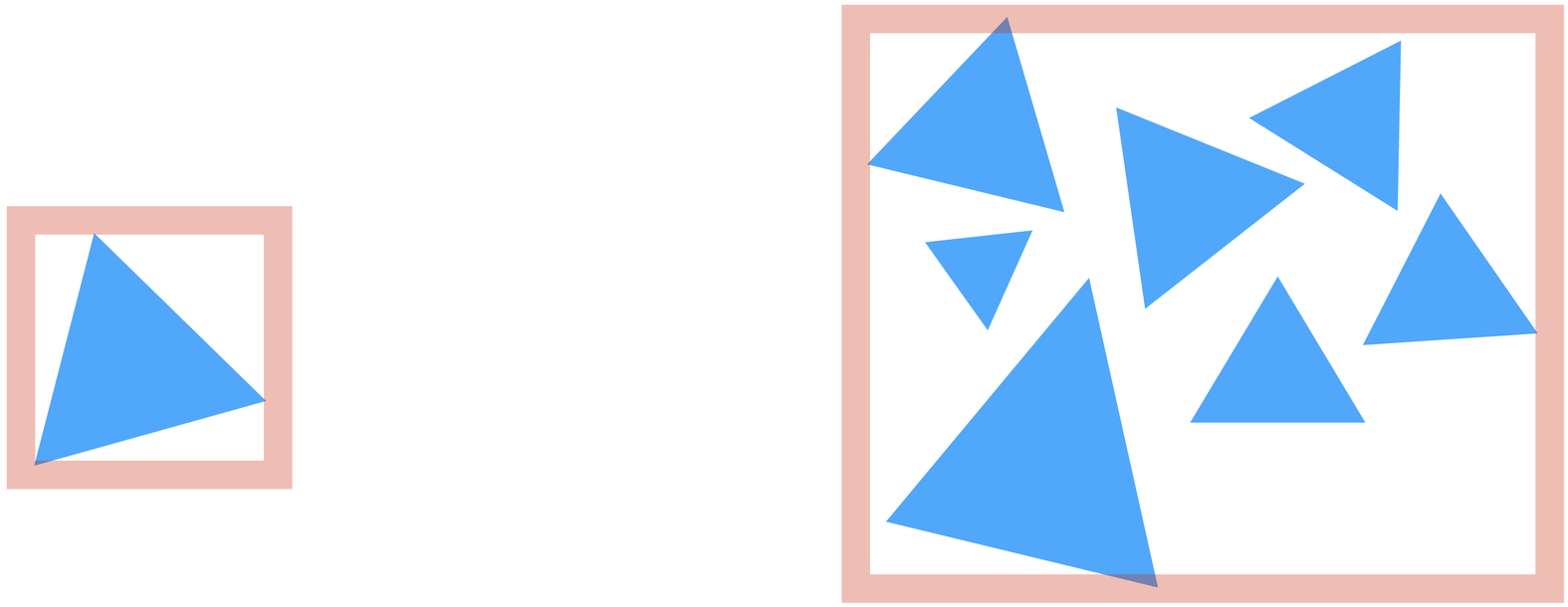
What about these?



Intuition about a "good" partition?



Partition into child nodes with equal numbers of primitives



Better partition

Intuition: want small bounding boxes (minimize overlap between children, avoid empty space)

What are we really trying to do?

A good partitioning minimizes the <u>cost</u> of finding the closest intersection of a ray with primitives in the node.

EASY CASE—for a leaf node:

$$C = \sum_{i=1}^{N} C_{\text{isect}}(i)$$

$$= N C_{\text{isect}}$$

Where $C_{\text{isect}}(i)$ is the cost of ray-primitive intersection for primitive *i* in the node.

(Common to assume all primitives have the same cost)

Cost of making a partition

HARDER CASE—the <u>expected cost</u> of intersecting an interior node, given that the node's primitives are partitioned into child sets A and B:

$$C = C_{\text{trav}} + p_A C_A + p_B C_B$$

C_{trav} **is the cost of traversing an interior node (e.g., bounding box test)**

C_A **and** C_B **are the costs of intersection with the resultant child subtrees**

p_A **and** p_B **are the probability a ray intersects the bbox of the child nodes A and B**

Primitive count is common heuristic for child node costs:

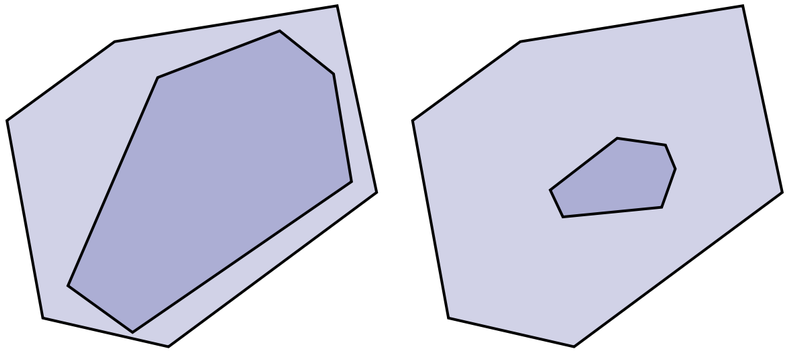
$$C = C_{\text{trav}} + p_A N_A C_{\text{isect}} + p_B N_B C_{\text{isect}}$$

Remaining question: how do we get the probabilities p_A, p_B?

Estimating probabilities

- **For convex object A inside convex object B, the probability that a random ray that hits B also hits A is given by the ratio of the surface areas S_A and S_B of these objects.**

$$P(\text{hit}A|\text{hit}B) = \frac{S_A}{S_B}$$



Leads to surface area heuristic (SAH):

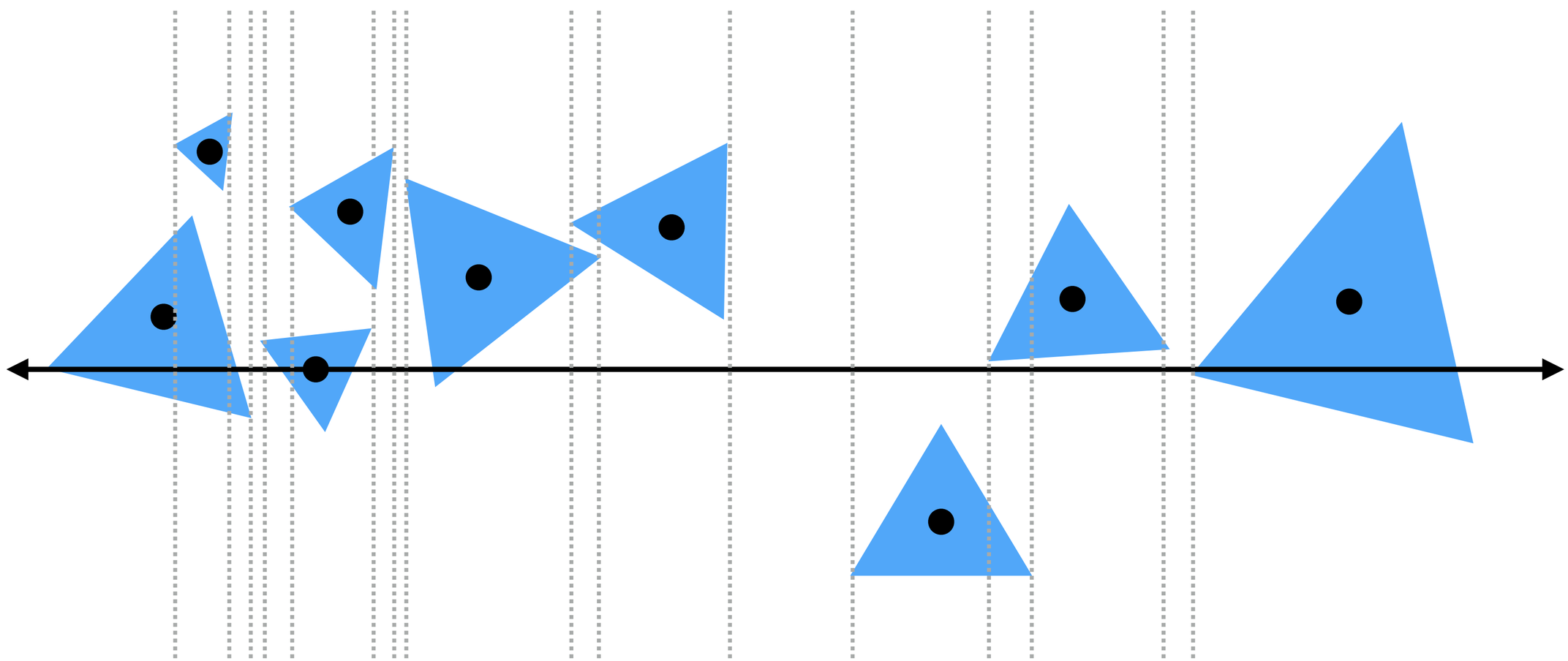
$$C = C_{\text{trav}} + \frac{S_A}{S_N} N_A C_{\text{isect}} + \frac{S_B}{S_N} N_B C_{\text{isect}}$$

Assumptions of the SAH (*which may not hold in practice!*):

- **Rays are randomly distributed**
- **No occlusion (i.e., one object blocking another)**

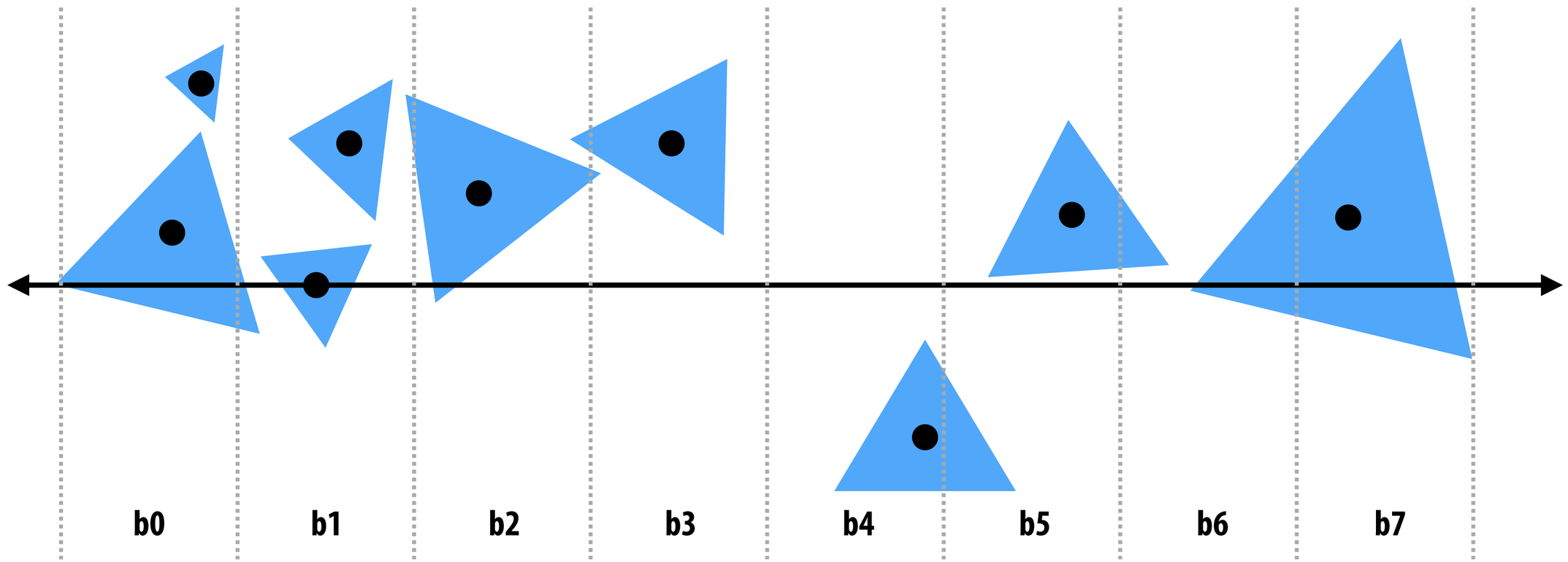
Implementing partitions

- Constrain search for good partitions to axis-aligned spatial partitions
  - Choose an axis; choose a split plane on that axis
  - Partition primitives by the side of splitting plane their centroid lies
  - Cost estimate changes only when plane moves past triangle boundary
  - Have to consider rather large number of possible split planes…



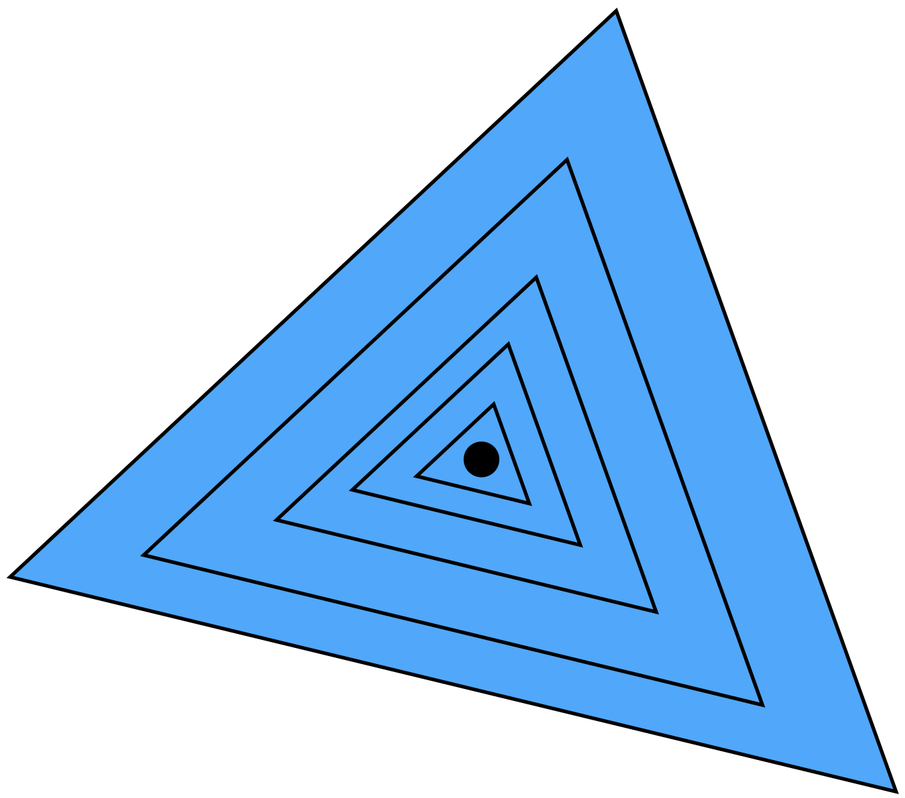
Efficiently implementing partitioning

- **Efficient modern approximation: split spatial extent of primitives into B buckets (B is typically small: B < 32)**

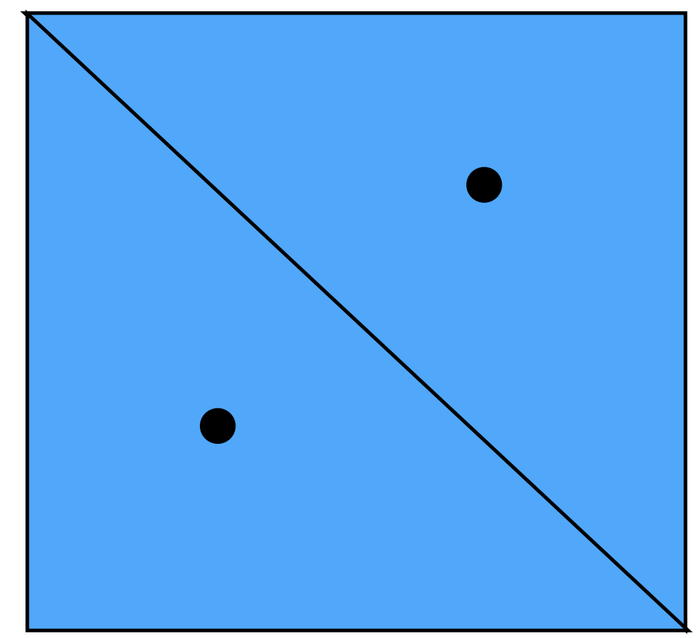



```
For each axis x,y,z:
   initialize buckets
   For each primitive p in node:
      b = compute_bucket(p.centroid)
      b.bbox.union(p.bbox);
      b.prim_count++;
   For each of the B-1 possible partitioning planes
      Evaluate cost, keep track of lowest cost partition
Recurse on lowest cost partition found (or make node a leaf)
```

Troublesome cases



All primitives with same centroid (all primitives end up in same partition)

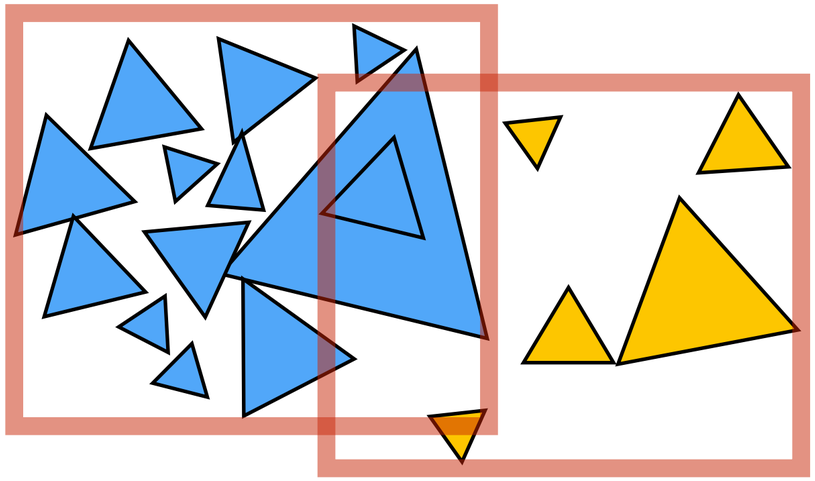


All primitives with same bbox (ray often ends up visiting both partitions)

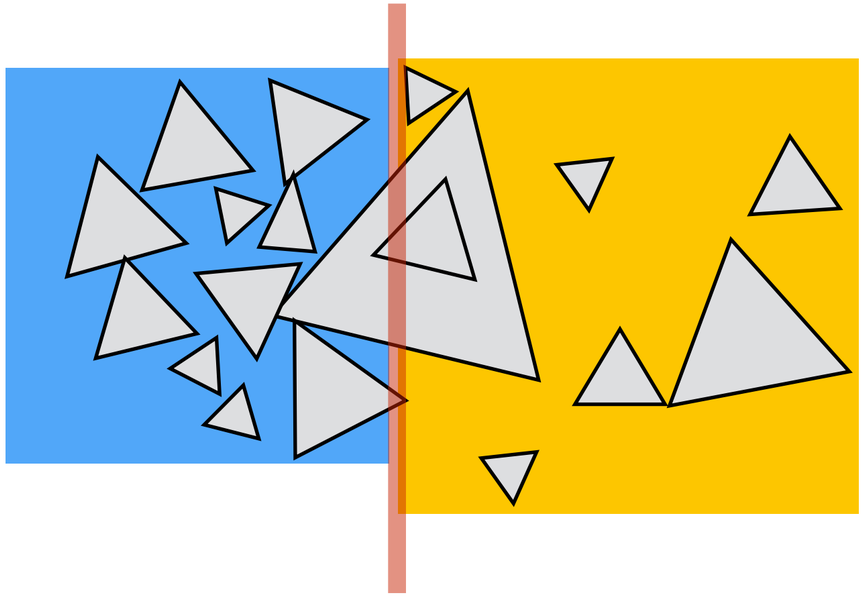
In general, different strategies may work better for different types of geometry / different distributions of primitives…

Primitive-partitioning acceleration structures vs. space-partitioning structures

- Primitive partitioning (bounding volume hierarchy): partitions node's primitives into disjoint sets (but sets may overlap in space)

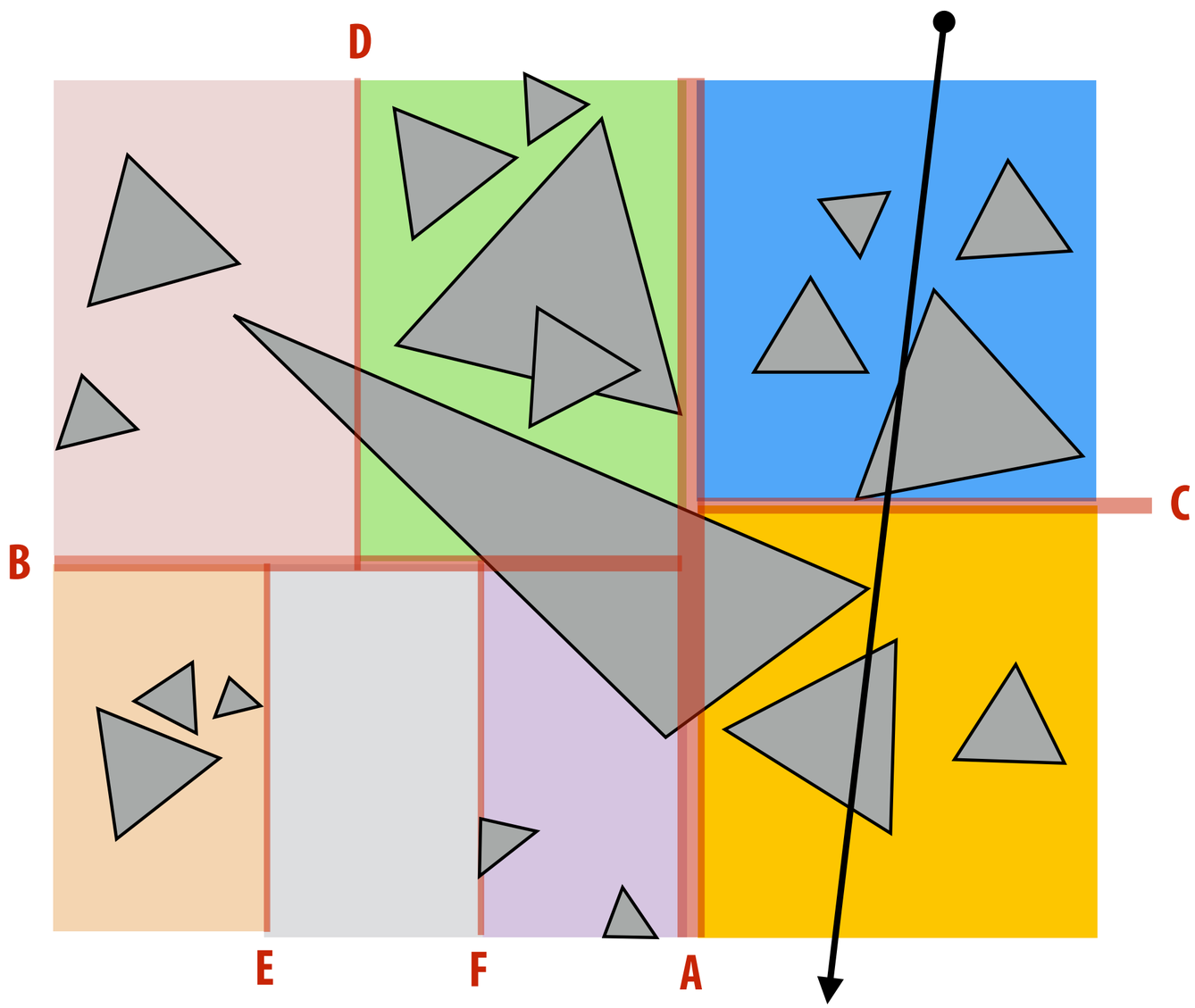


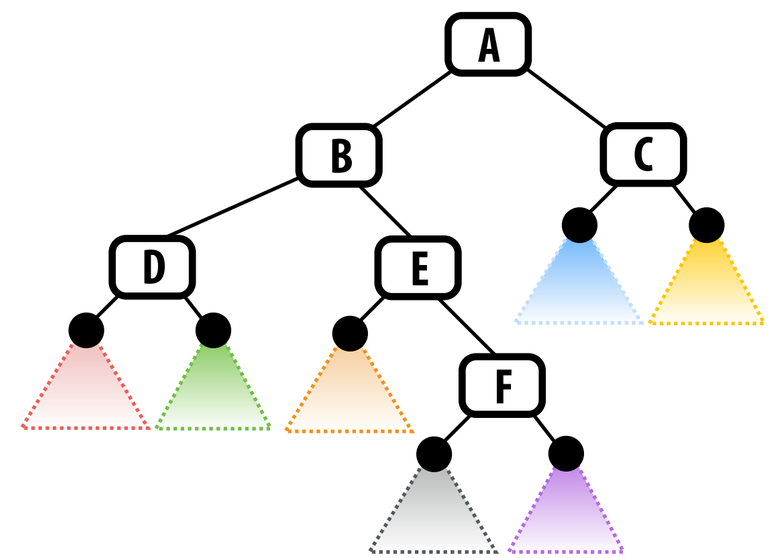
- Space-partitioning (grid, K-D tree) partitions space into disjoint regions (primitives may be contained in multiple regions of space)



K-D tree

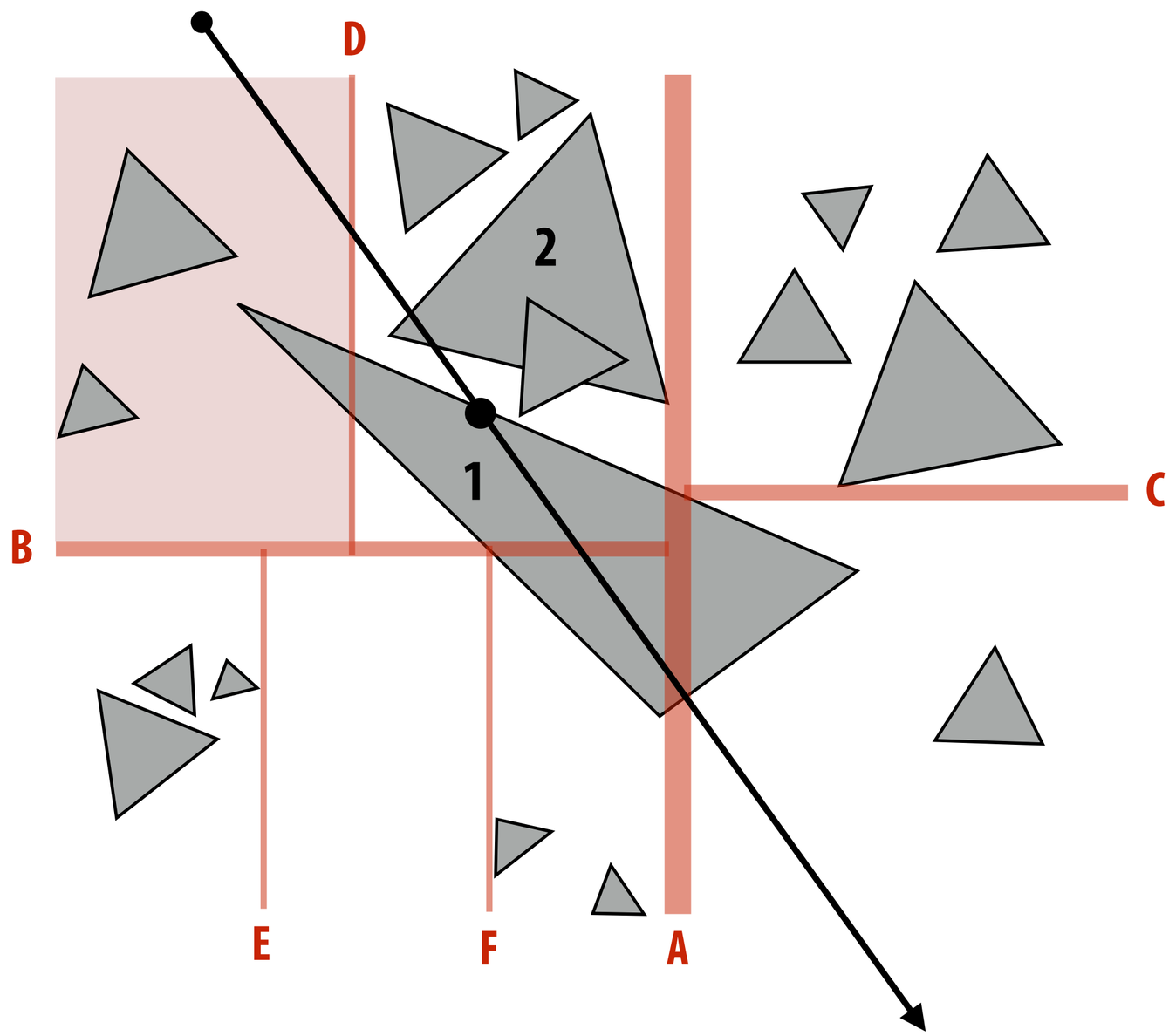
- Recursively partition <u>space</u> via axis-aligned partitioning planes
  - Interior nodes correspond to spatial splits
  - Node traversal can proceed in front-to-back order
  - Unlike BVH, can terminate search after first hit is found.

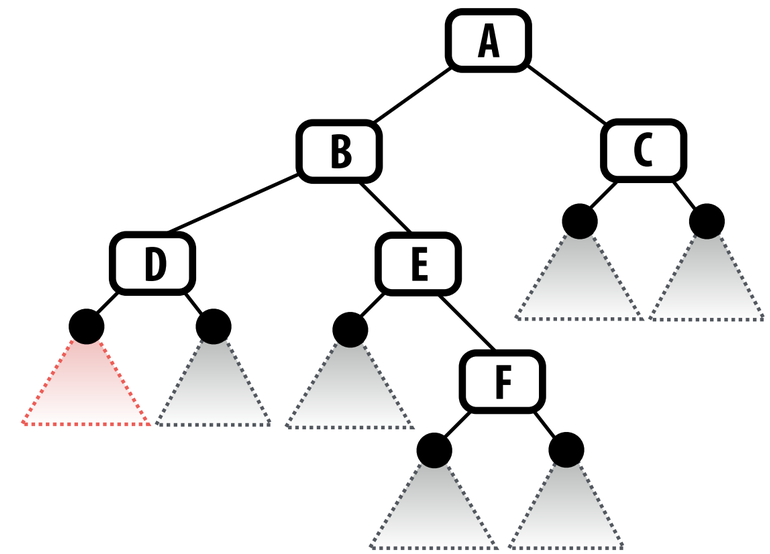




Challenge: objects overlap multiple nodes

- Want node traversal to proceed in front-to-back order so traversal can terminate search after first hit found





Triangle 1 overlaps multiple nodes.

Ray hits triangle 1 when in highlighted leaf cell.

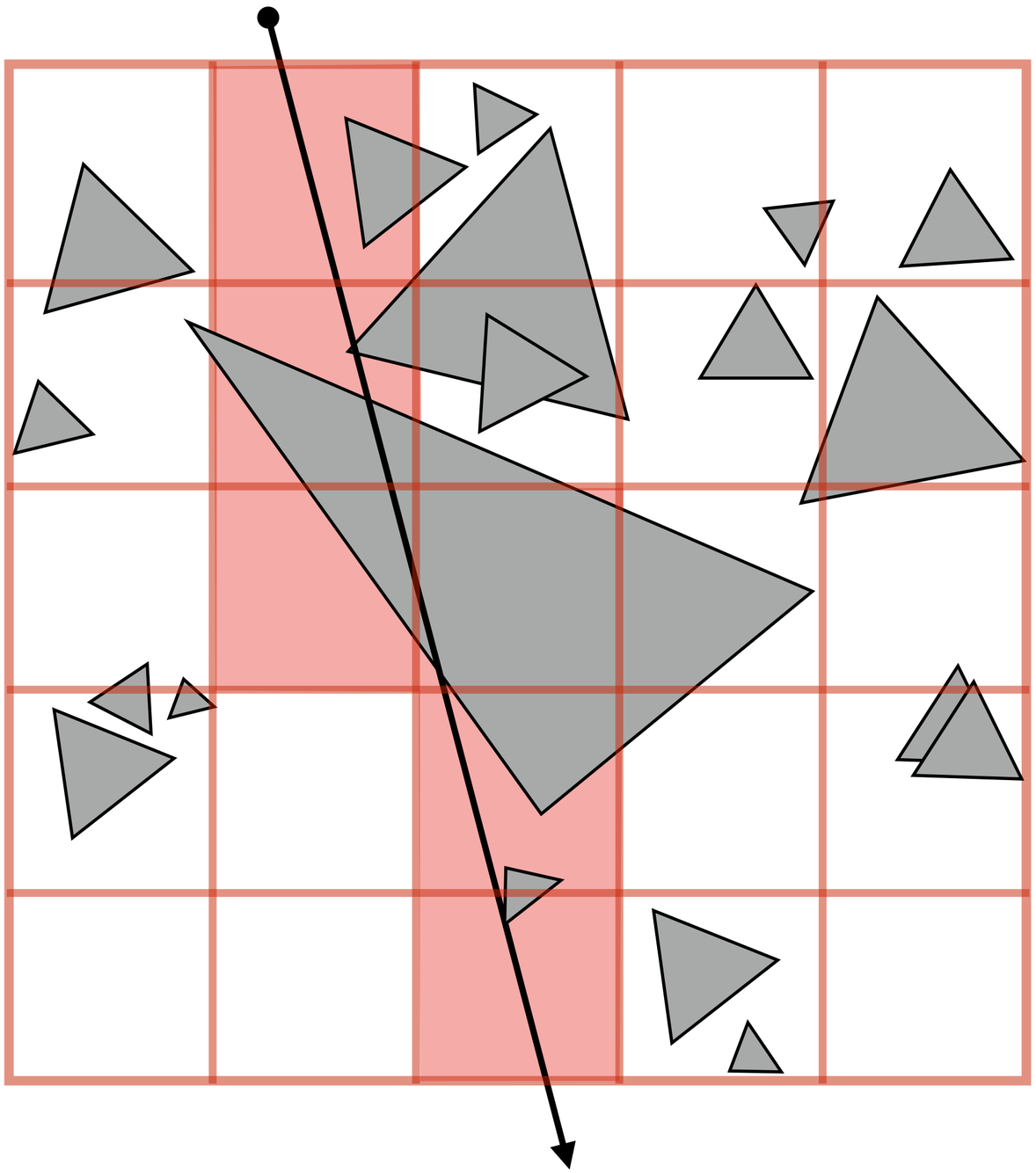
But intersection with triangle 2 is closer! (Haven't traversed to that node yet)

Solution: require primitive intersection point to be within current leaf node.

(primitives may be intersected multiple times by same ray *)

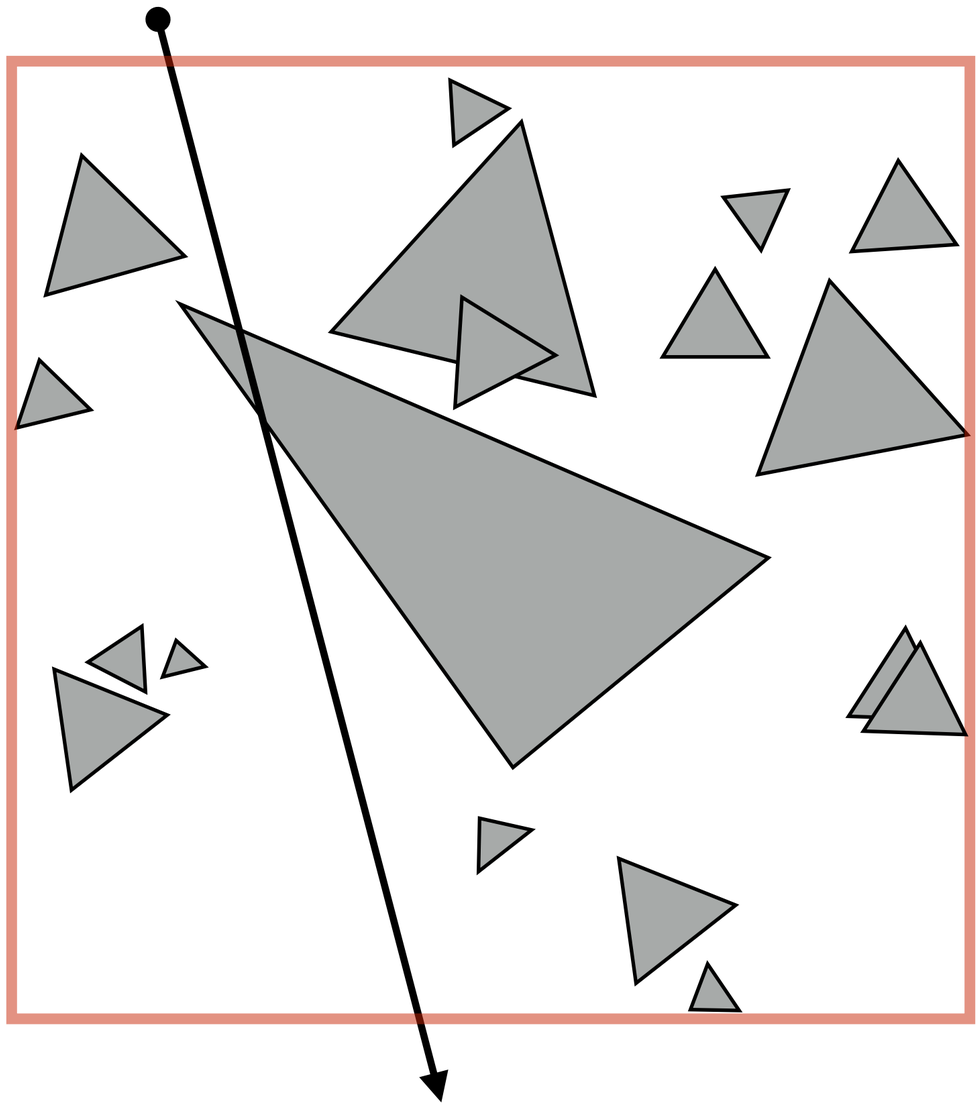
* Caching or "mailboxing" can be used to avoid repeated intersections

Uniform grid

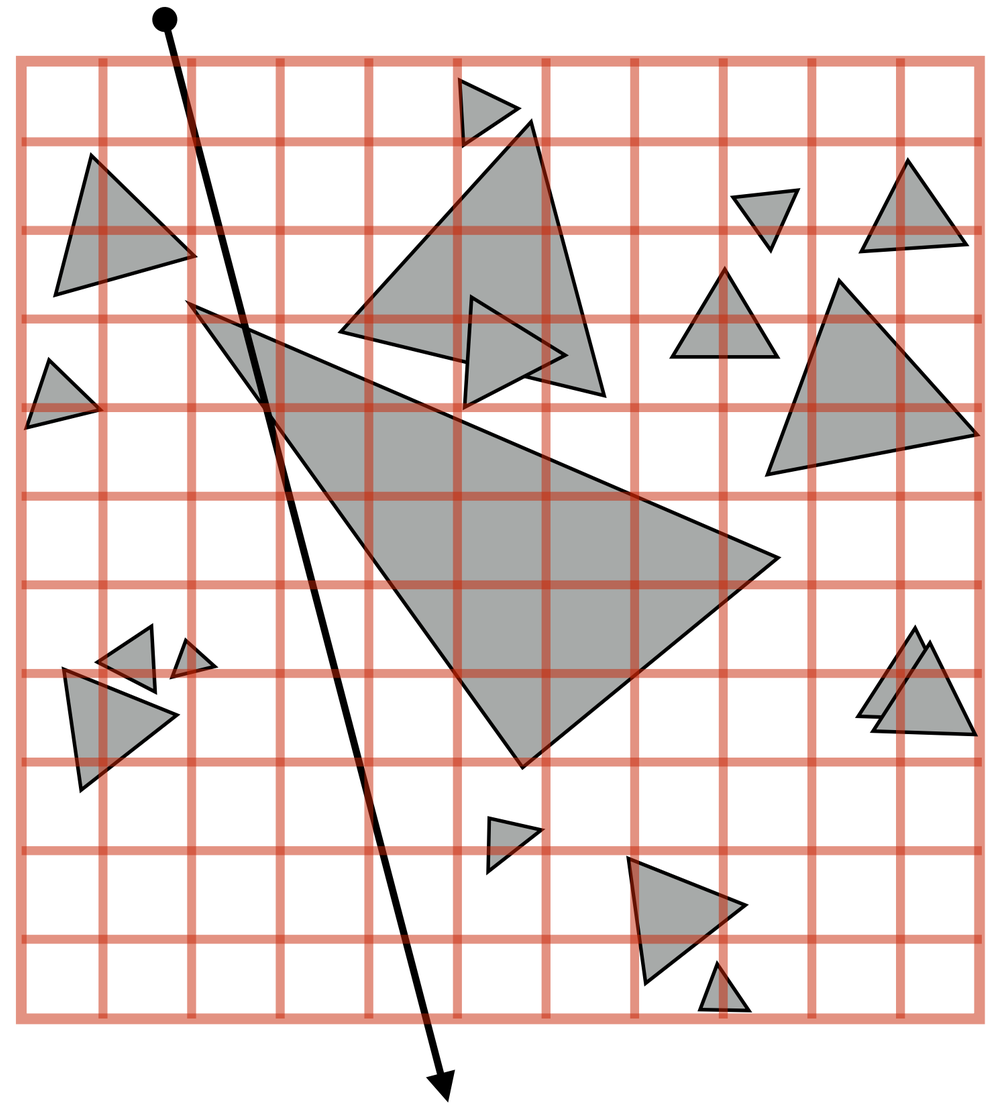


- Partition space into equal sized volumes (volume-elements or "voxels")
- Each grid cell contains primitives that overlap voxel. (very cheap to construct acceleration structure)
- Walk ray through volume in order
  - Very efficient implementation possible (think: *3D line rasterization*)
  - Only consider intersection with primitives in voxels the ray intersects

What should the grid resolution be?



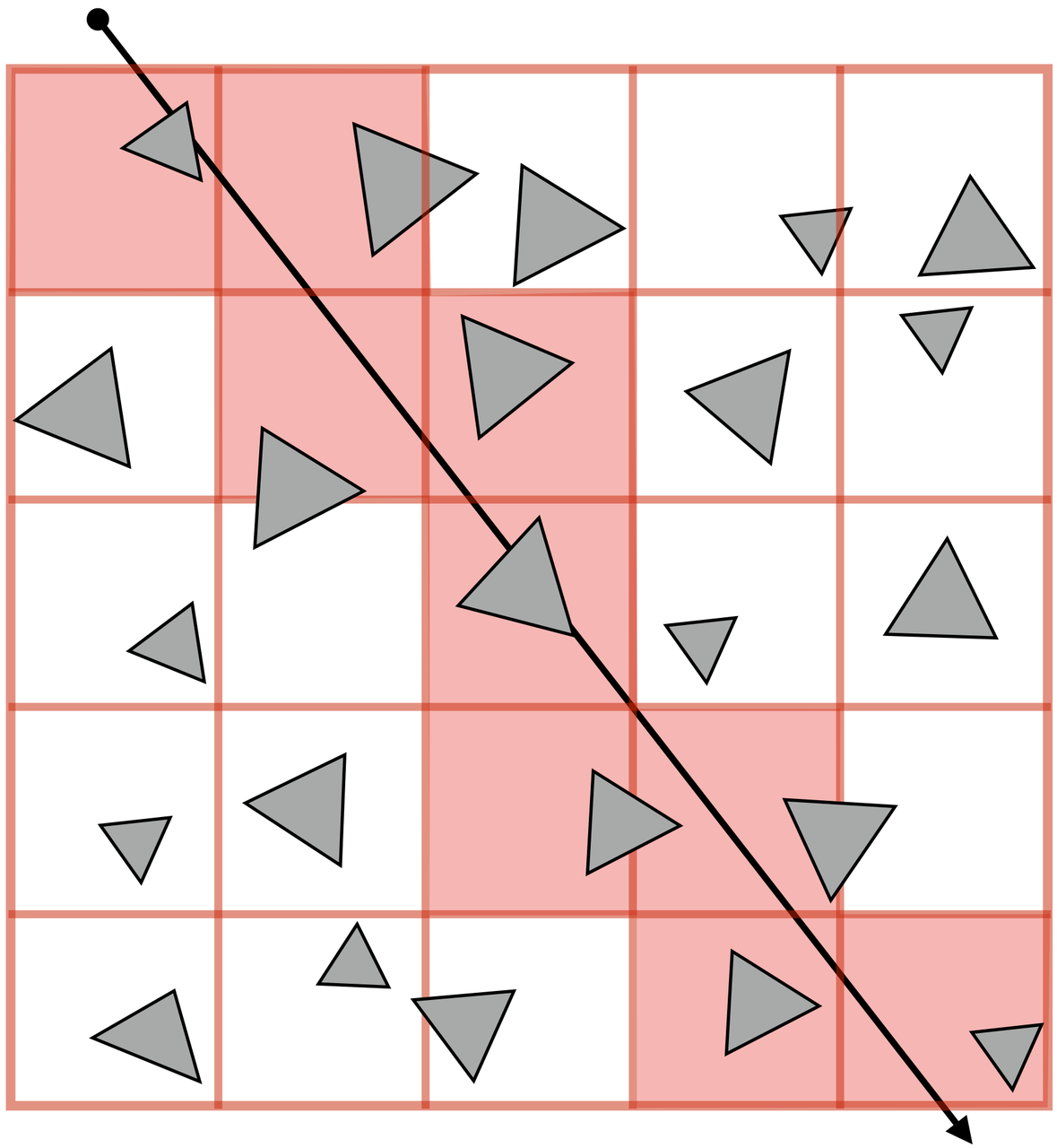
Too few grid cells: degenerates to brute-force approach



Too many grid cells: incur significant cost traversing through cells with empty space

Heuristic

- **Choose number of voxels ~ total number of primitives**
  (constant primitives per voxel — assuming uniform distribution)



Intersection cost: $O(\sqrt[3]{N})$

(Q: Which grows faster, cube root of N or log(N)?

Uniform distribution of primitives

Terrain / height fields:

[Image credit: Misuba Renderer]

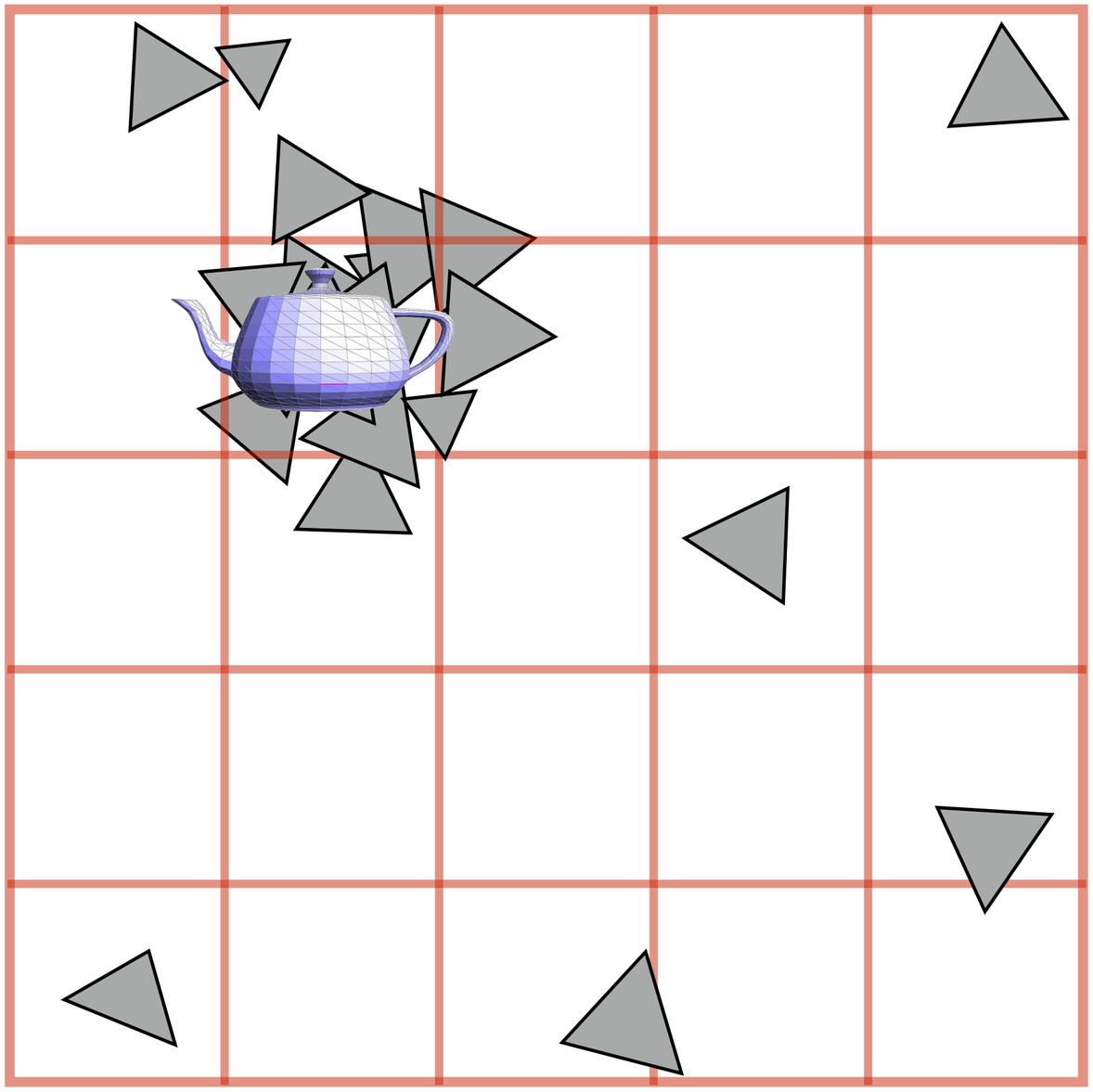
Grass:

[Image credit: www.kevinboulanger.net/grass.html]

Example credit: Pat Hanrahan

Uniform grid cannot adapt to non-uniform distribution of geometry in scene

(Unlike K-D tree, location of spatial partitions is not dependent on scene geometry)



"Teapot in a stadium problem"

- Scene has large spatial extent.
- Contains a high-resolution object that has small spatial extent (ends up in one grid cell)

Non-uniform distribution of geometric detail

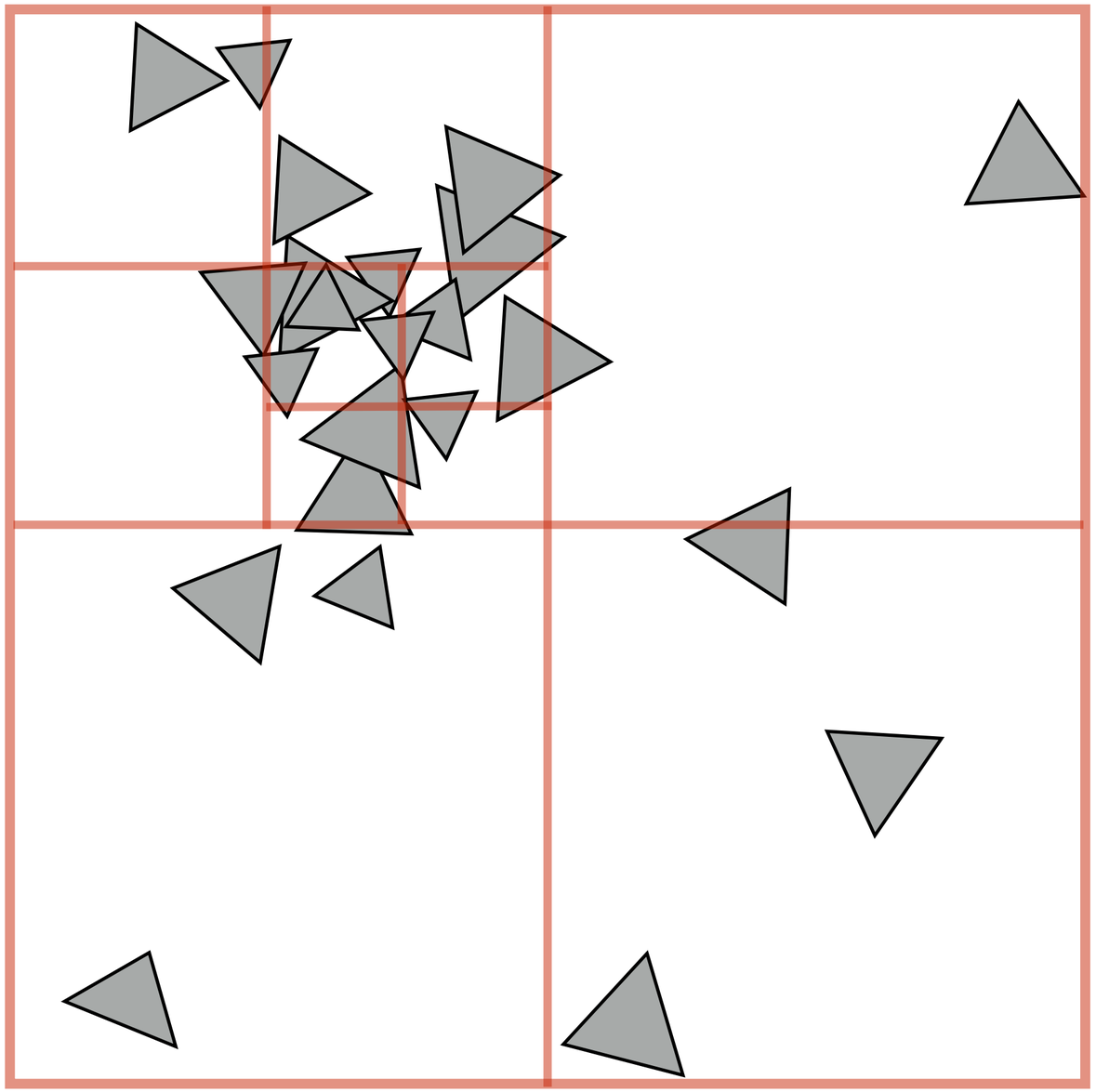
[Image credit: Pixar]

Quad-tree / octree

Like uniform grid: easy to build (don't have to choose partition planes)

Has greater ability to adapt to location of scene geometry than uniform grid.

But lower intersection performance than K-D tree (only limited ability to adapt)



Quad-tree: nodes have 4 children (partitions 2D space)

Octree: nodes have 8 children (partitions 3D space)

Summary of spatial acceleration structures:
Choose the right structure for the job!

- **Primitive vs. spatial partitioning:**
  - **Primitive partitioning: partition sets of objects**
    - **Bounded number of BVH nodes**
    - **Simpler to update if primitives in scene change position**
  - **Spatial partitioning: partition space**
    - **Traverse space in order (first intersection is closest intersection)**
    - **May intersect primitive multiple times**

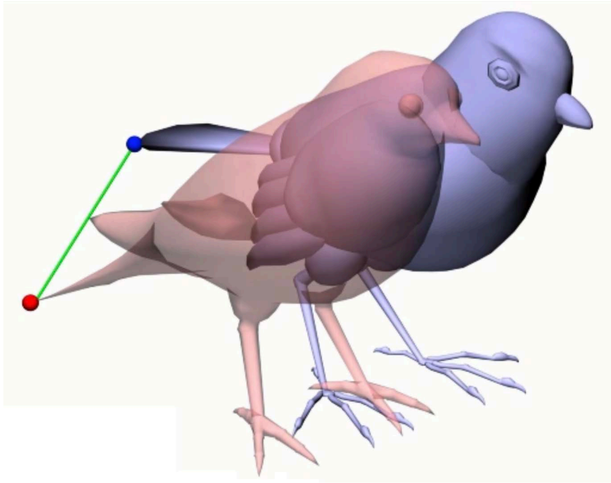
- **Adaptive structures (BVH, K-D tree)**
  - **More costly to construct (must be able to amortize cost over many geometric queries)**
  - **Better intersection performance under non-uniform distribution of primitives**

- **Non-adaptive accelerations structures (uniform grids)**
  - **Simple, cheap to construct**
  - **Good intersection performance if scene primitives are uniformly distributed**

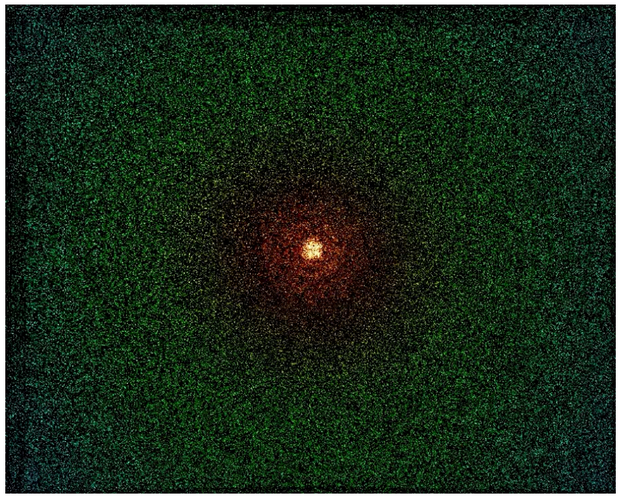
- **Many, many combinations thereof…**

Hierarchical Acceleration in Graphics

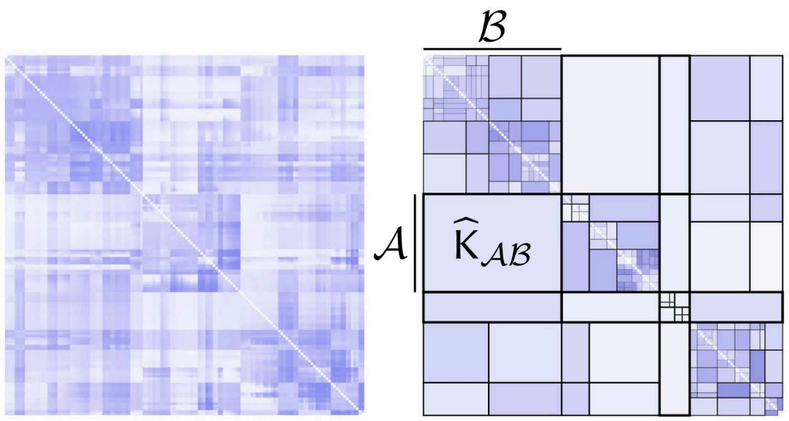
- **GEOMETRY**
  - Inside-outside tests (e.g., meshing)
  - Closest point tests (e.g., Hausdorff distance)



- **ANIMATION/SIMULATION**
  - "Particle systems"
  - N-body dynamics, fluid simulation, …
  - Barnes-Hut algorithm
  - fast multipole method

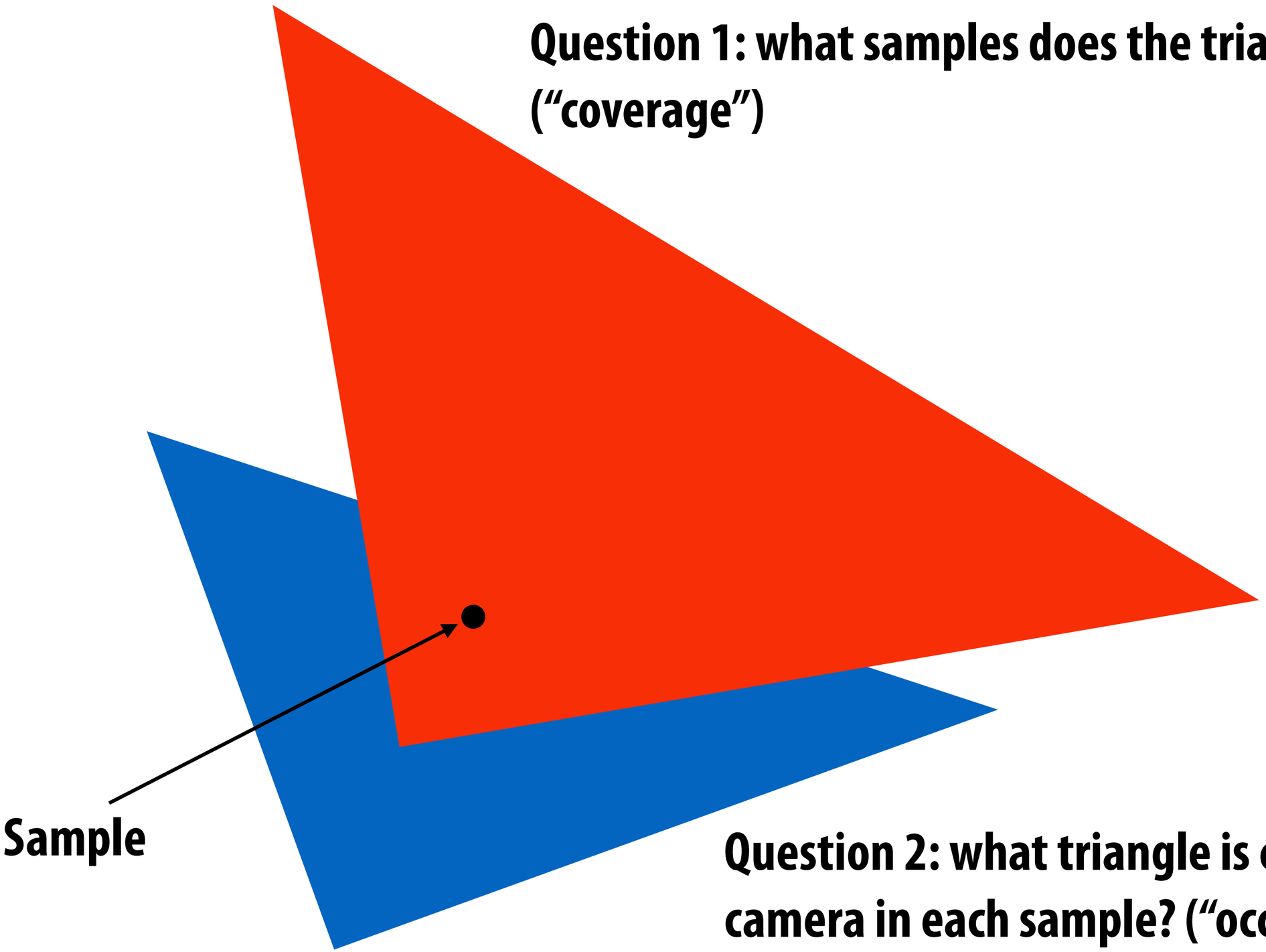


- **RENDERING**
  - Visibility
  - Physically-based ray tracing



Q: How can we use ray intersection queries to generate an image?

Recall triangle visibility problem:



Question 1: what samples does the triangle overlap? ("coverage")

Question 2: what triangle is closest to the camera in each sample? ("occlusion")

Before, we solved this problem using rasterization + depth buffering

But we can also do it via ray queries!

Basic rasterization algorithm

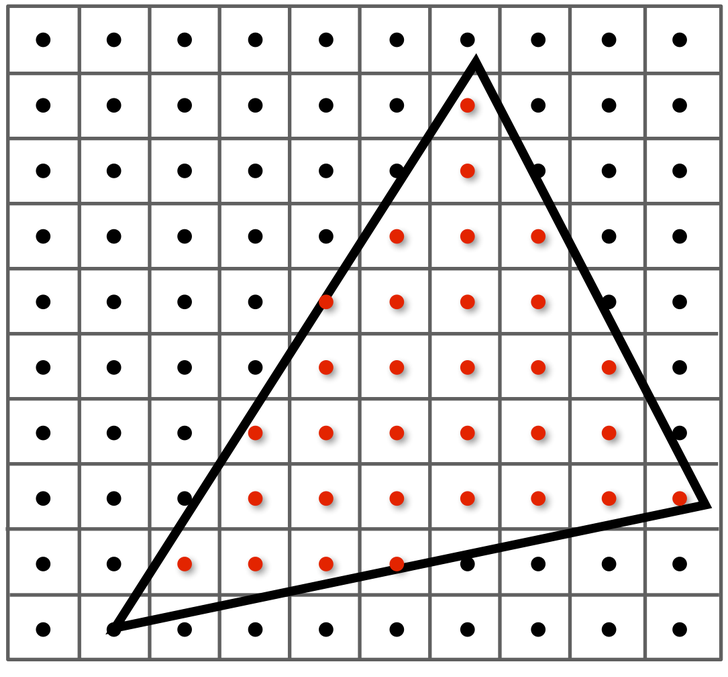
"For each triangle, find the samples it covers"

Sample = 2D point

Coverage: 2D triangle/sample tests (does projected triangle cover 2D sample point?)

Occlusion: depth buffer

```
initialize z_closest[] to ∞.                    // store closest-surface-so-far for all samples
initialize color[]                              // store scene color for all samples
for each triangle t in scene:                   // loop 1: triangles
    t_proj = project_triangle(t)
    for each 2D sample s in frame buffer:   // loop 2: visibility samples
        if (t_proj covers s)
            compute color of triangle at sample
            if (depth of t at s is closer than z_closest[s])
                update z_closest[s] and color[s]
```



Basic ray casting algorithm

"For each sample, find the primitives it's covered by"

Sample = a ray in 3D

Coverage: 3D ray-triangle intersection tests (does ray "hit" triangle)

Occlusion: closest intersection along ray

```
initialize color[]                                  // store scene color for all samples
for each sample s in frame buffer:                  // loop 1: visibility samples (rays)
    r = ray from s on sensor through pinhole aperture
    r.min_t = ∞                                     // only store closest-so-far for current ray
    r.tri = NULL;
    for each triangle tri in scene:                 // loop 2: triangles
        if (intersects(r, tri)) {                   // 3D ray-triangle intersection test
            if (intersection distance along ray is closer than r.min_t)
                update r.min_t and r.tri = tri;
        }
    color[s] = compute surface color of triangle r.tri at hit point
```

Both schemes use further acceleration:

RASTERIZATION — limit tests to bounding box of triangle

RAY TRACING — use hierarchical acceleration (as we saw today!)

Basic rasterization vs. ray casting

- **Rasterization:**
  - Proceeds in triangle order
  - Store depth buffer (random access to regular structure of fixed size)
  - Don't have to store entire scene in memory, naturally supports unbounded size scenes
- **Ray casting:**
  - Proceeds in screen sample order
    - Don't have to store closest depth so far for the entire screen (just current ray)
    - Natural order for rendering transparent surfaces (process surfaces in the order the are encountered along the ray: front-to-back or back-to-front)
  - Must store entire scene
  - Performance more strongly depends on distribution of primitives in scene
- **High-performance implementations embody similar techniques:**
  - Hierarchies of rays/samples
  - Hierarchies of geometry
  - Deferred shading
  - …

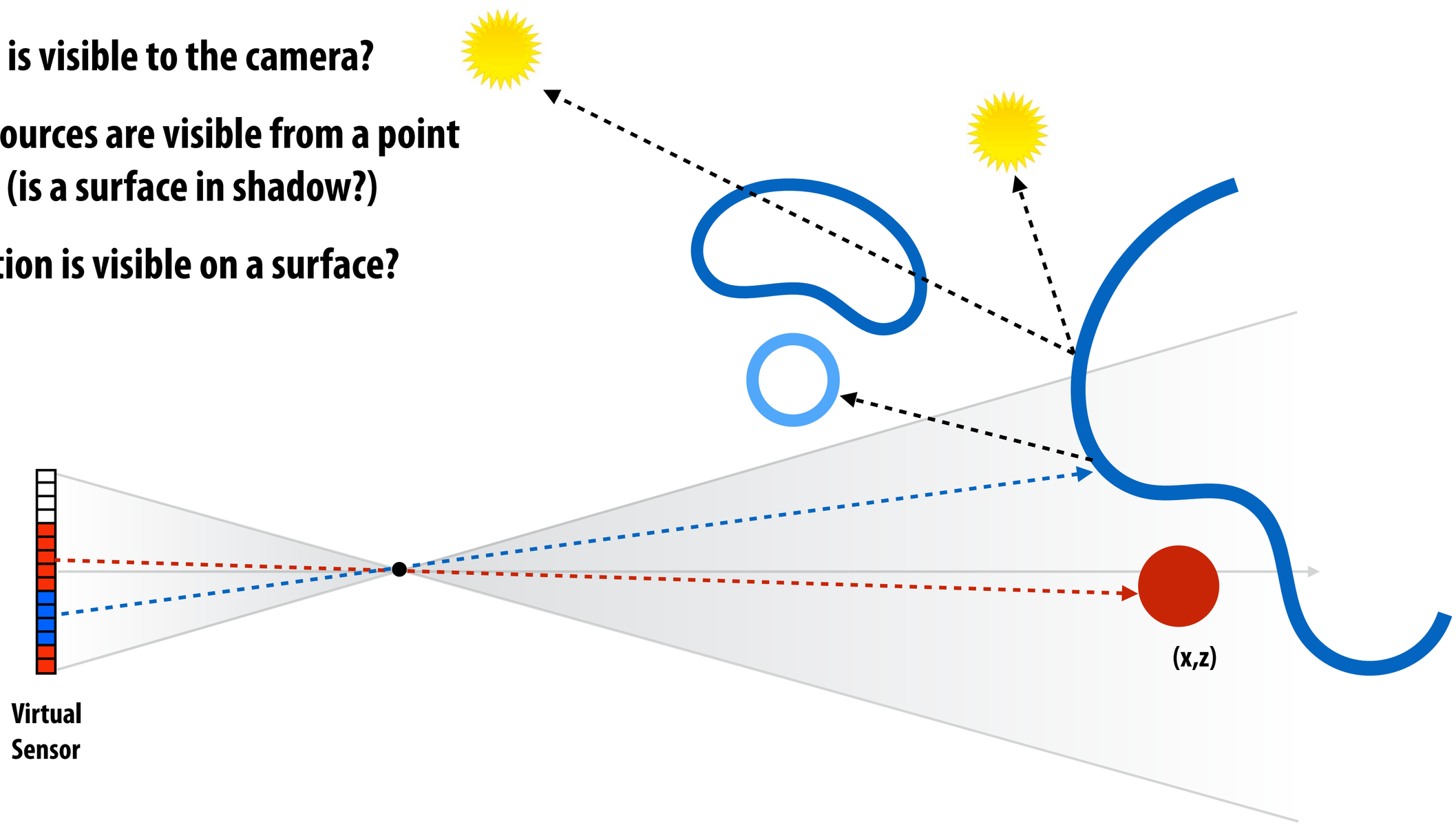
There is an important difference…

Ray casting can be used for many tasks:

What object is visible to the camera?

What light sources are visible from a point on a surface (is a surface in shadow?)

What reflection is visible on a surface?



In contrast, rasterization is a highly-specialized solution for computing visibility for a set of uniformly distributed rays originating from the same point (most often: the camera)

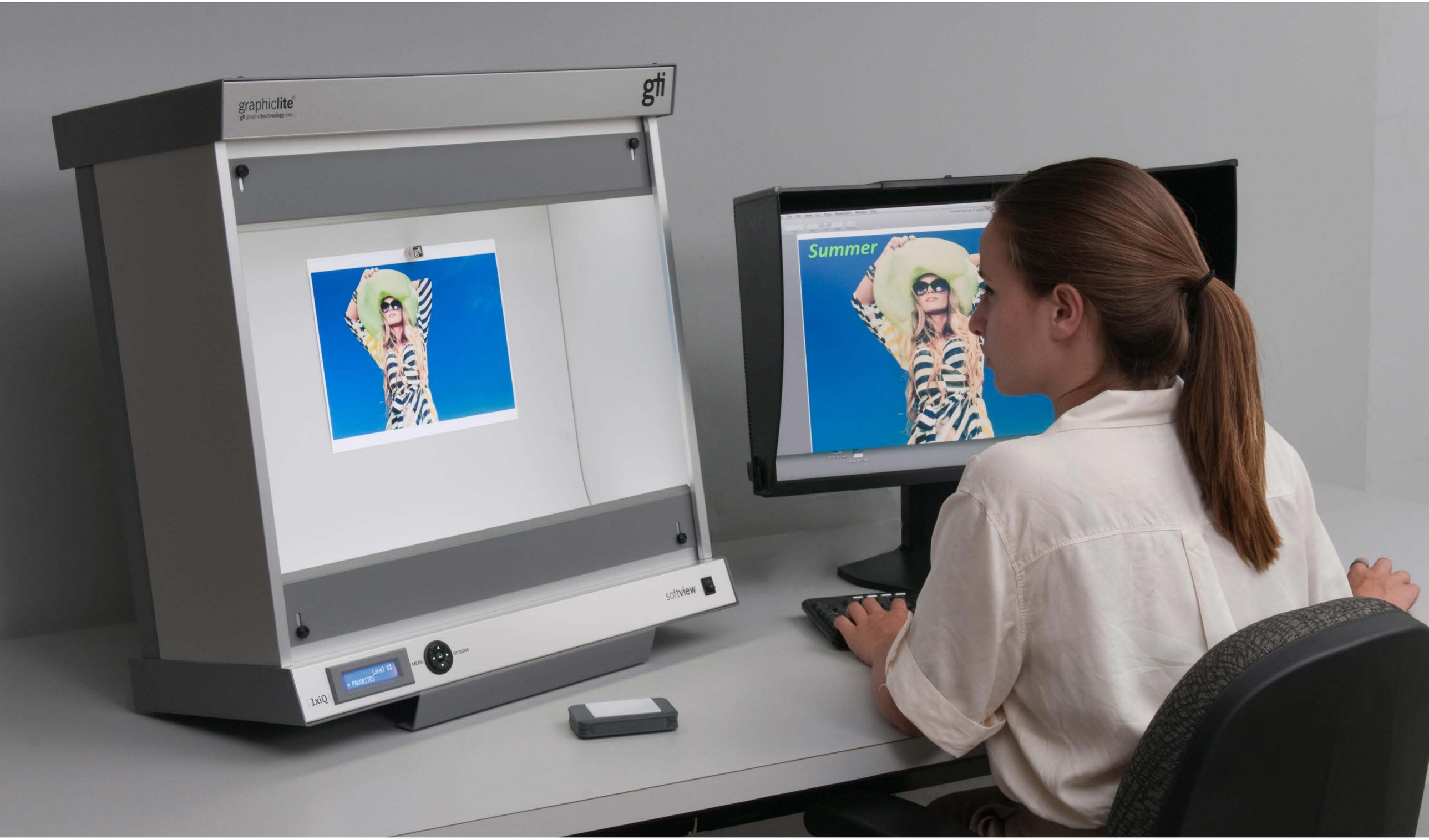
Next time: Color

Color

Computer Graphics
CMU 15-462/15-662

Why do we need to be able to talk precisely about color?

WIN
SONY
CONIA
WIN

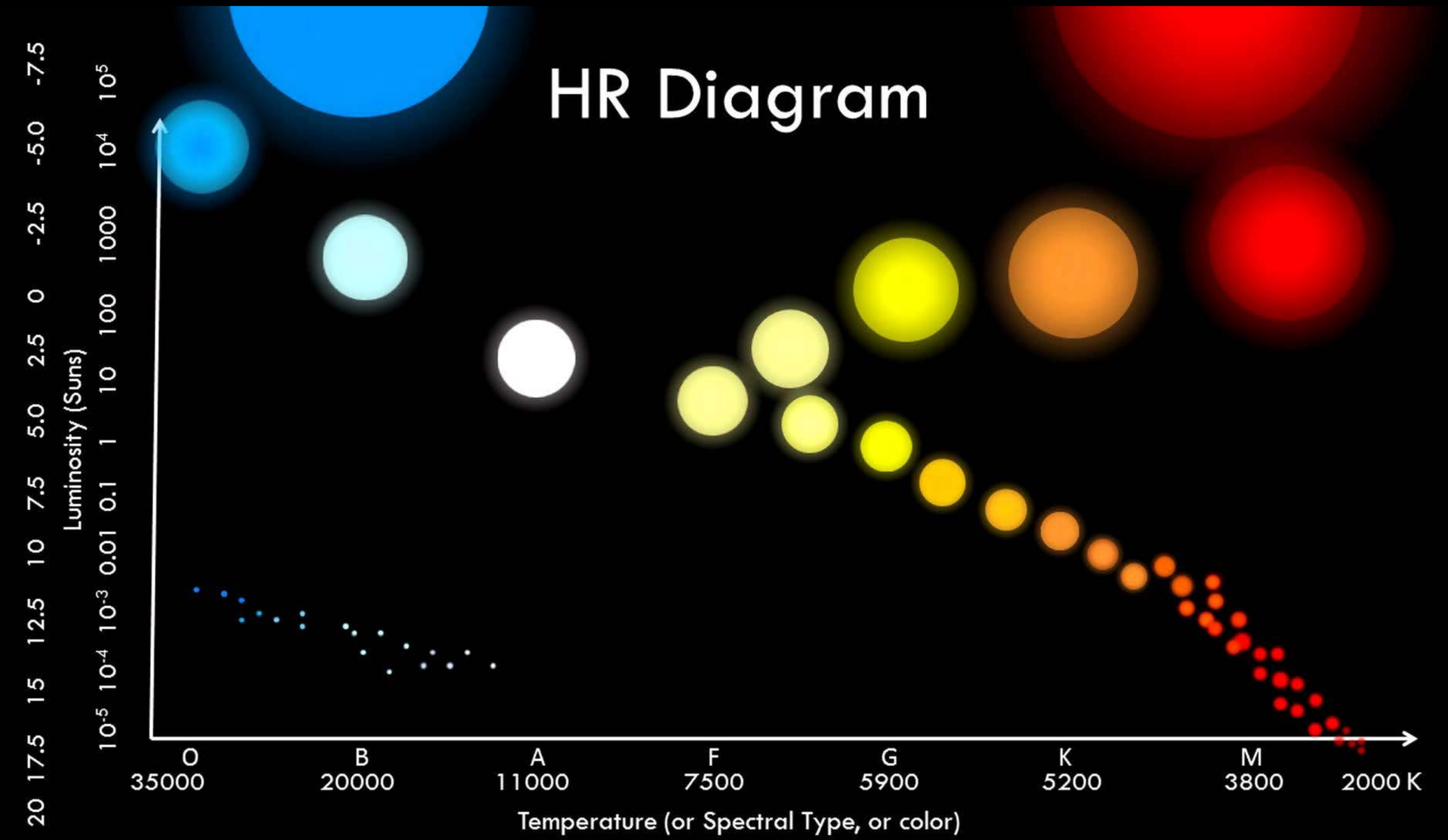

graphiclite
gti graphictechnology, inc.
gti
softview
1xiQ
MENU
OPTIONS
Summer

on screen

printed

Zhangye Danxia Geological Park, China

Credit: http://parade.com/63549/linzlowe/where-in-the-world-are-these-incredible-rainbow-mountains



Hertzsprung-Russell diagram

Credit: Adric Riedel https://www.youtube.com/watch?v=588oz17Xj8o

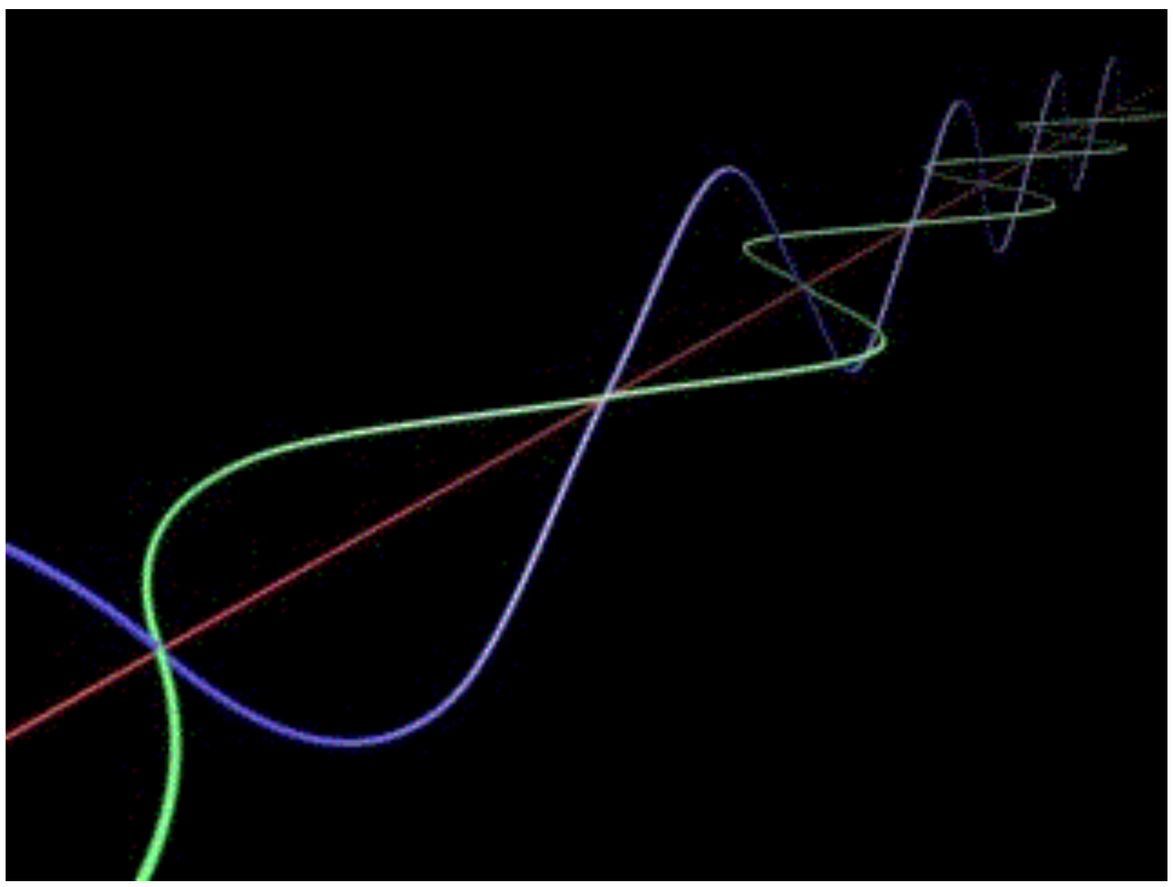
Starry Night, Van Gogh

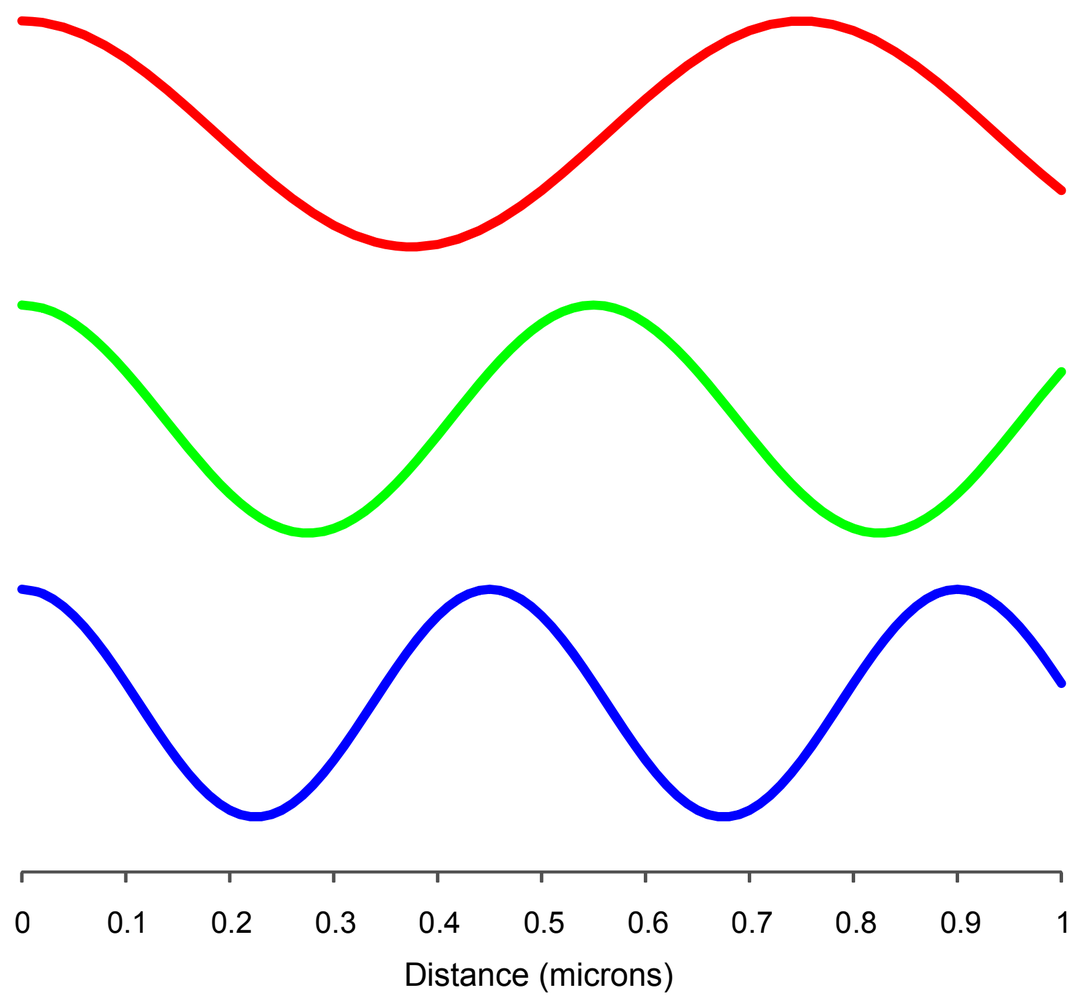
Cannon Beach, Oregon

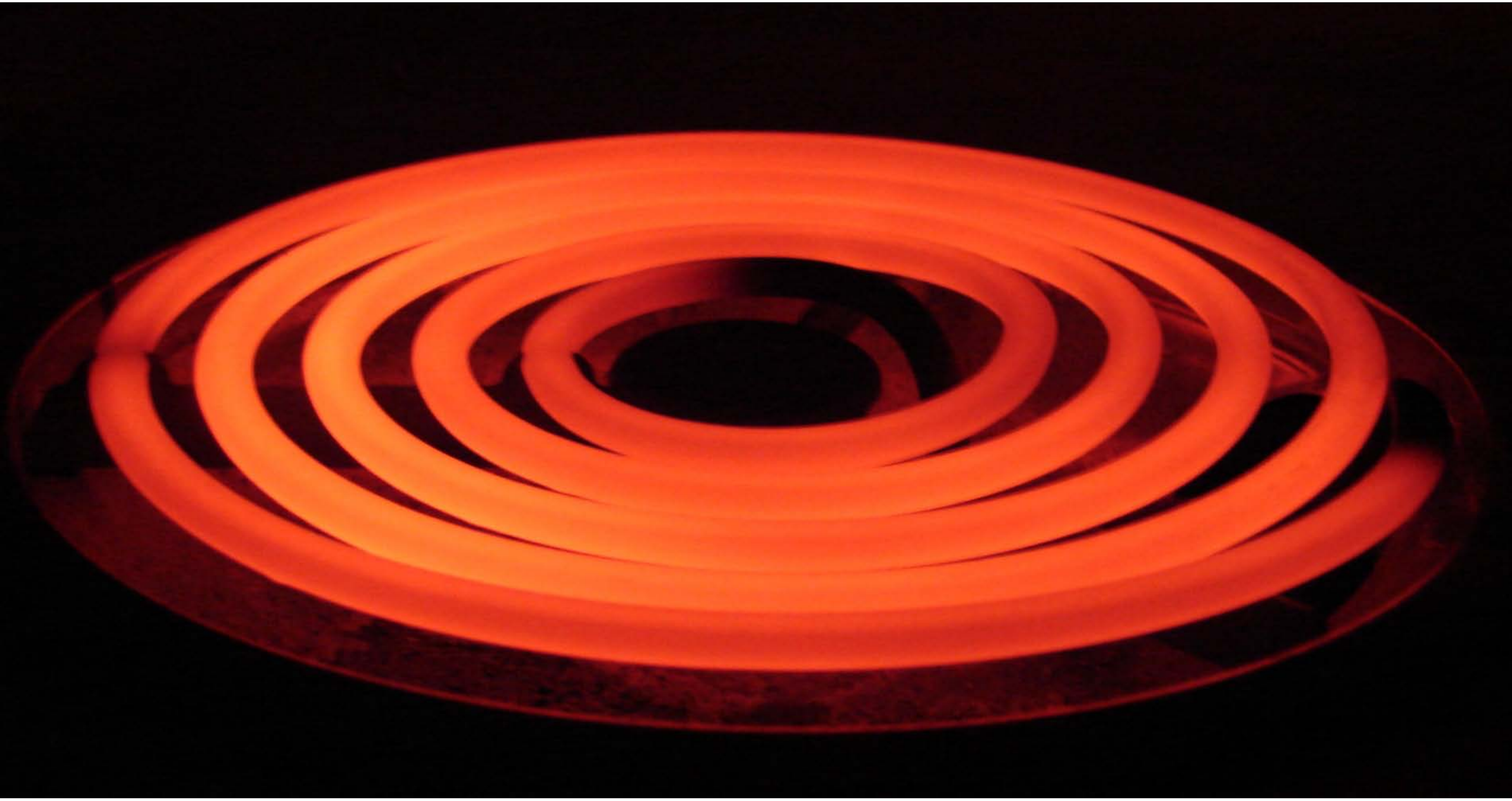
What is color?

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*?



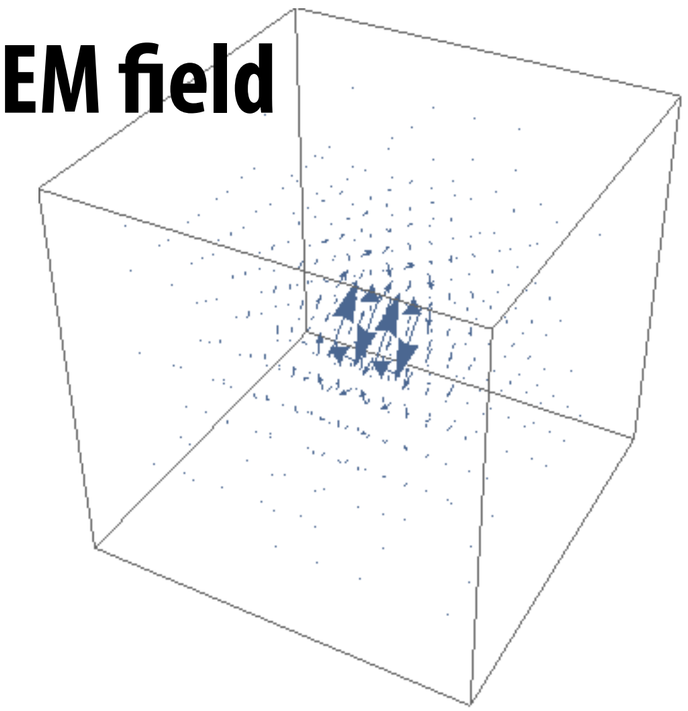


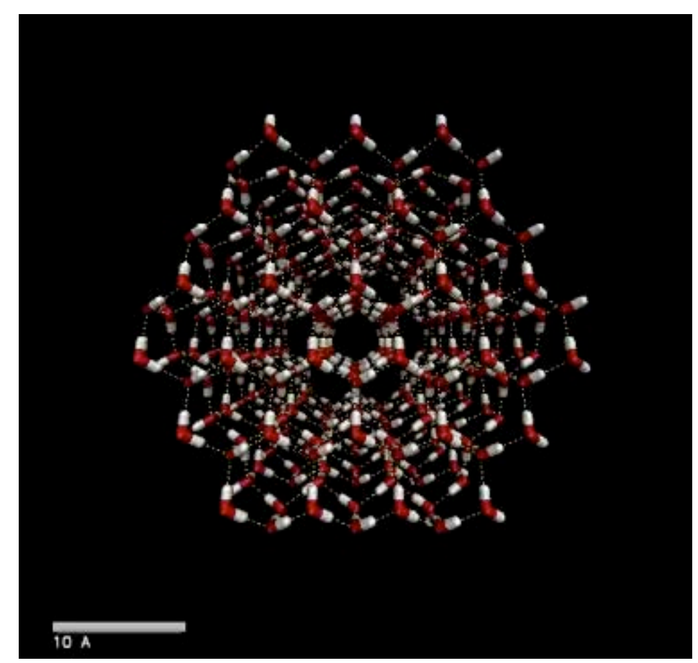


Q: Why does your stove turn red when it heats up?

Heat generates light

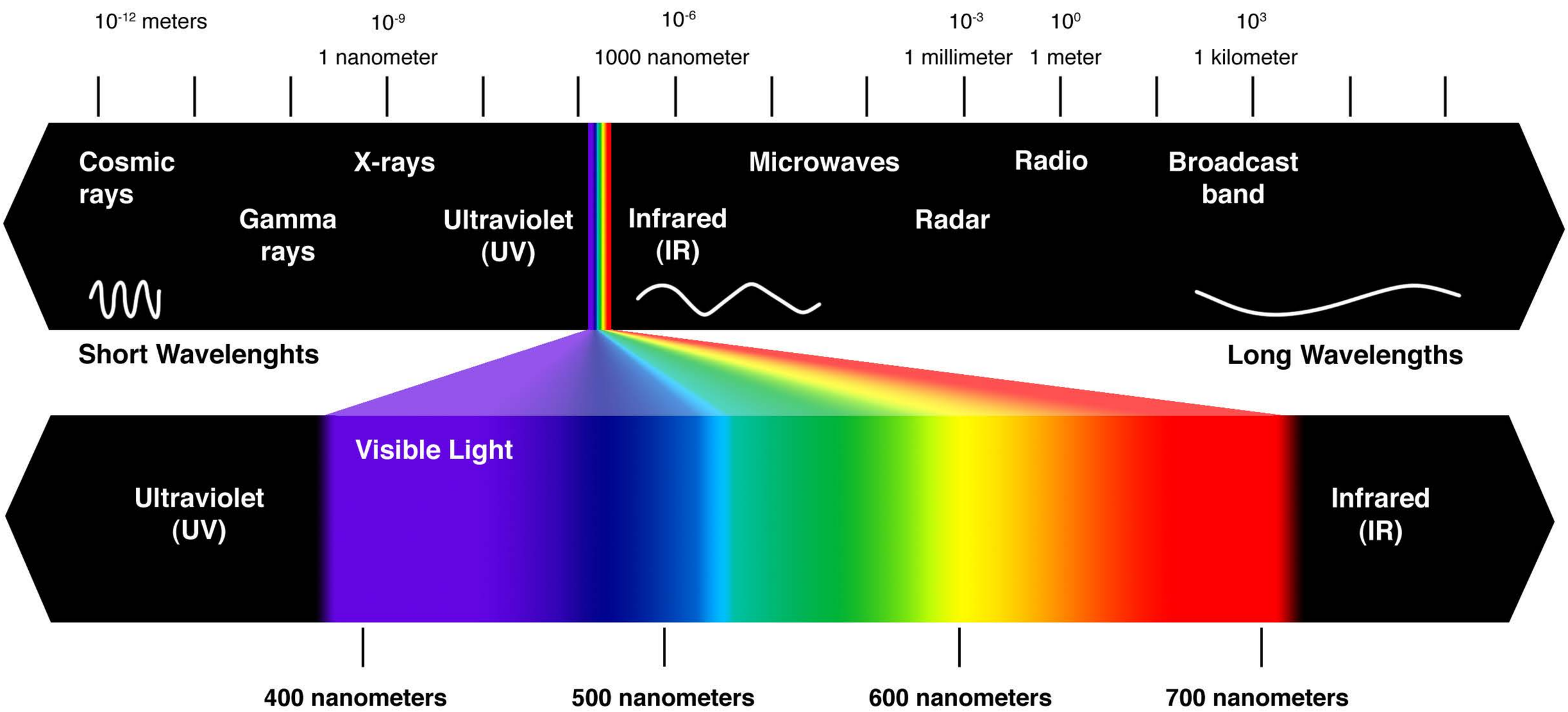
- One of *many* ways light is produced:
- Maxwell: motion of charged particles creates EM field
- Thermodynamics: …particles jiggle around!
- Hence, anything moving generates light
- In other words:
  - *every* object around you is producing color!
  - frequency determined by temperature





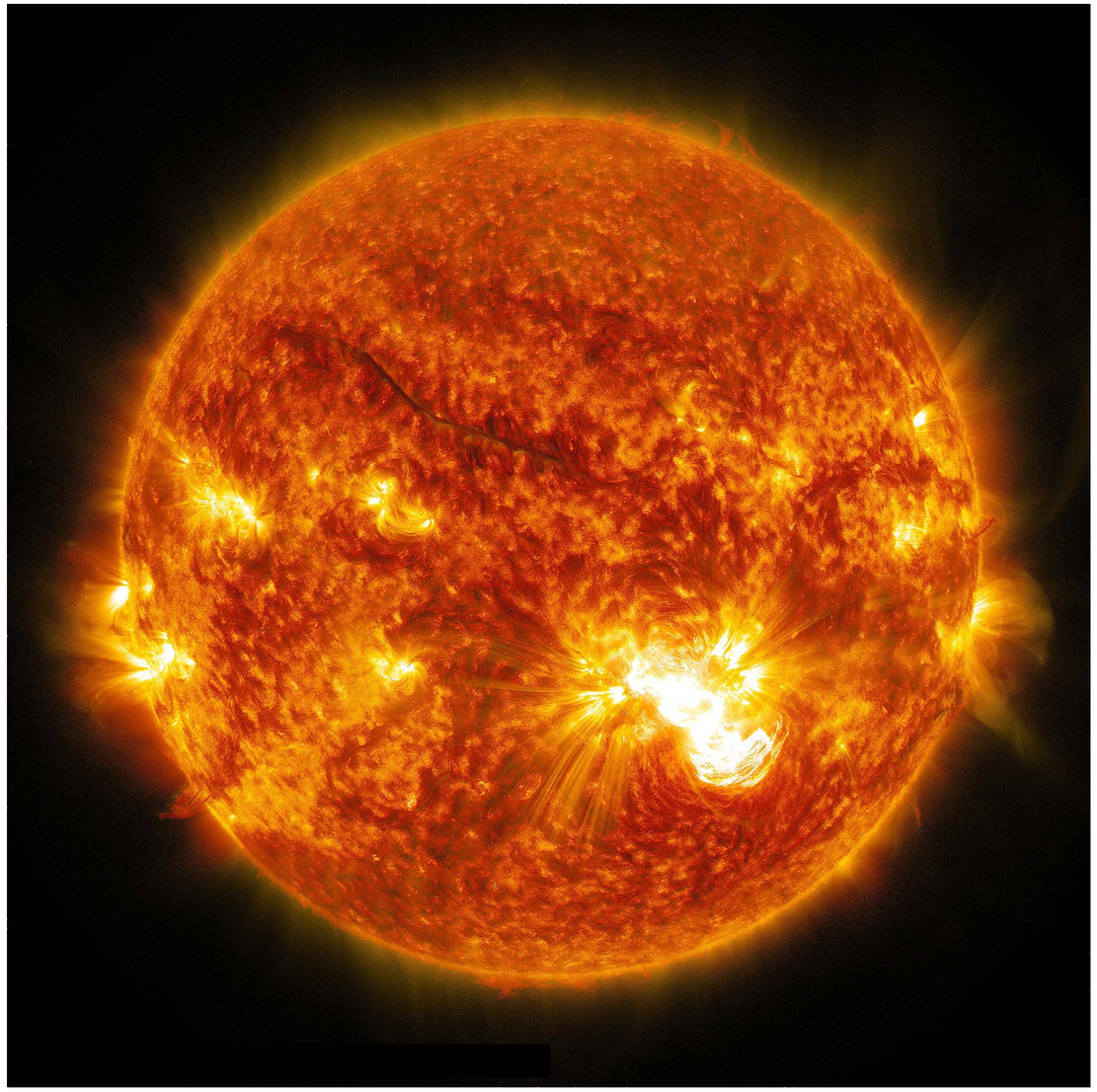
Most light is not visible!

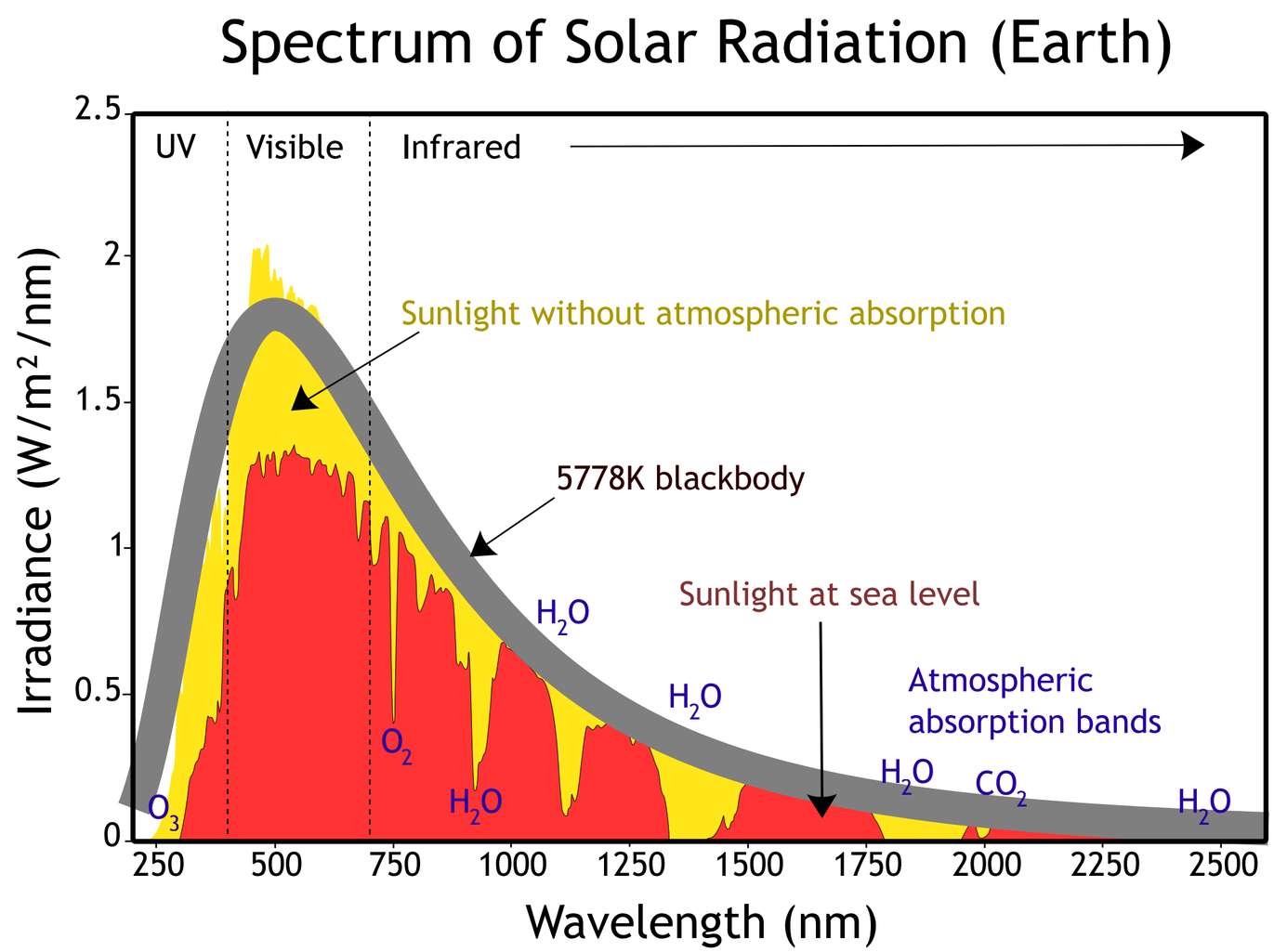
- Frequencies visible by human eyes are called "visible spectrum"
- These frequencies what we normally think of as "color"



Natural light is a mixture of frequencies

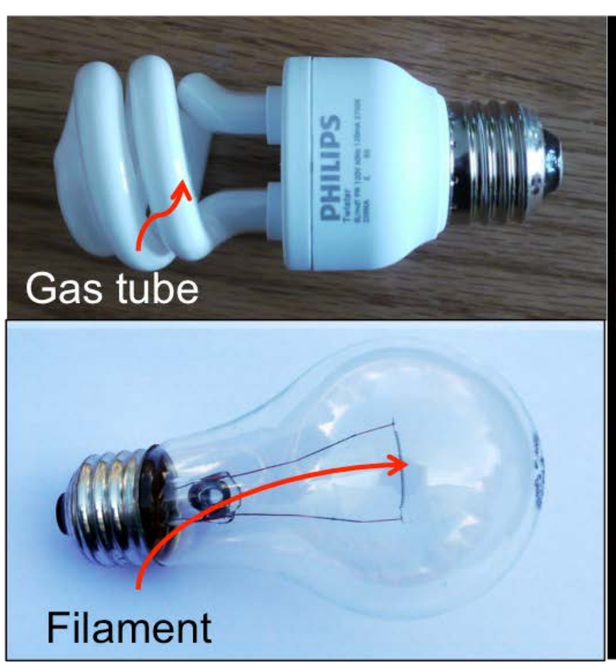
- "White" light is really a mixture of all (visible) frequencies
- E.g., the light from our sun

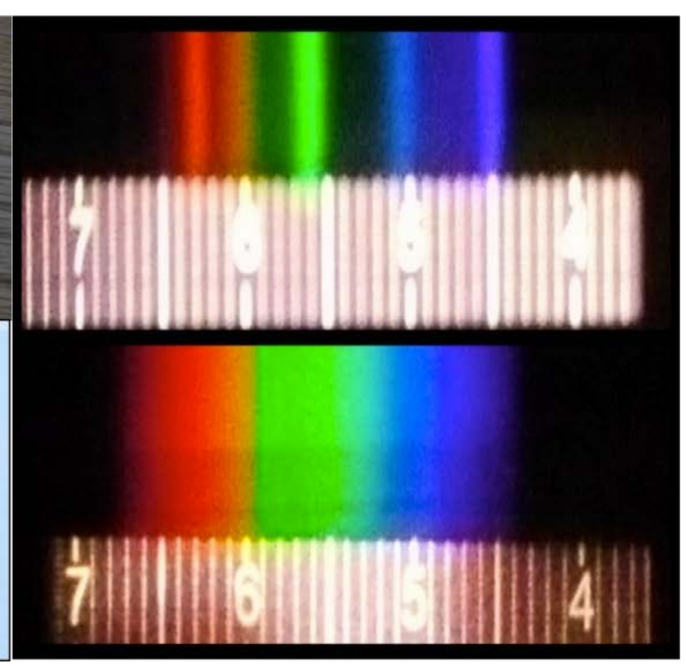


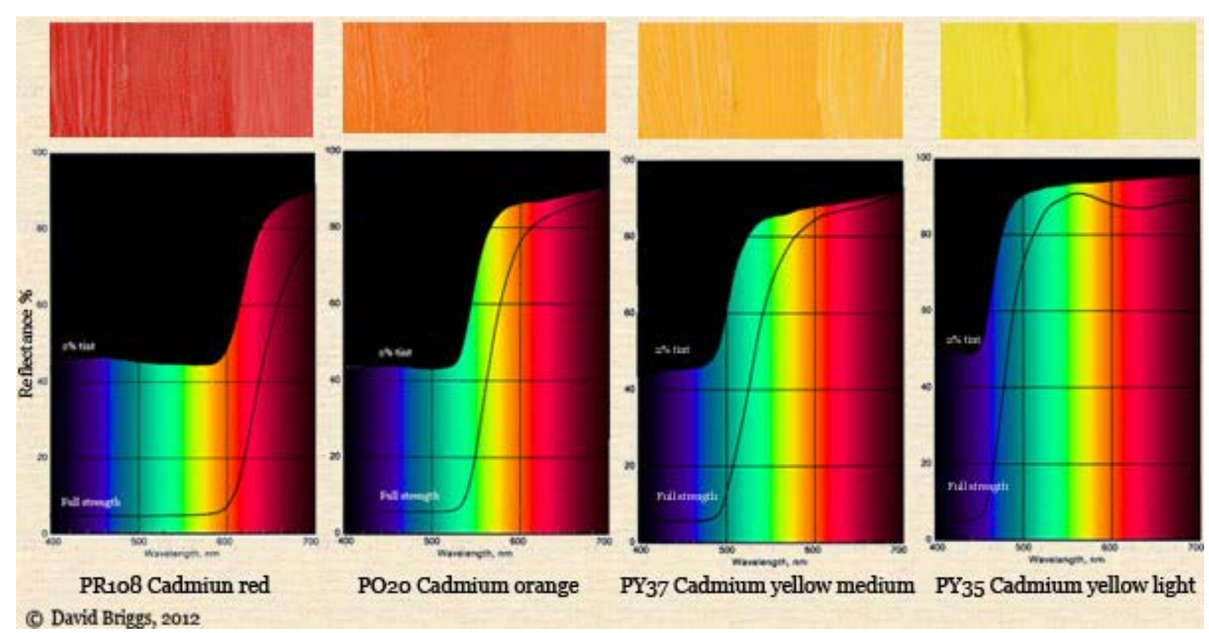


Additive vs. Subtractive Models of Light

- 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: *"absorbtion 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:

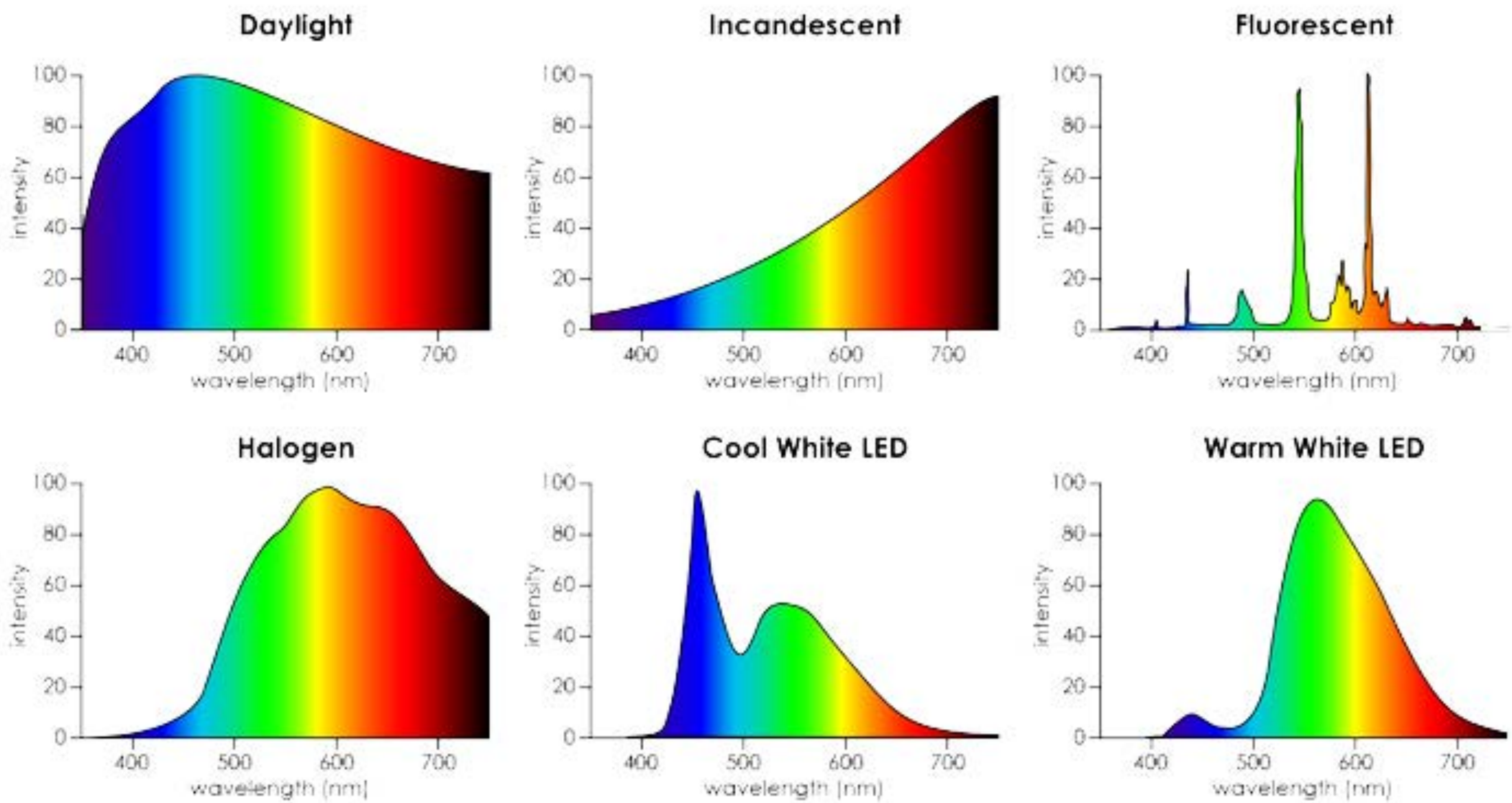
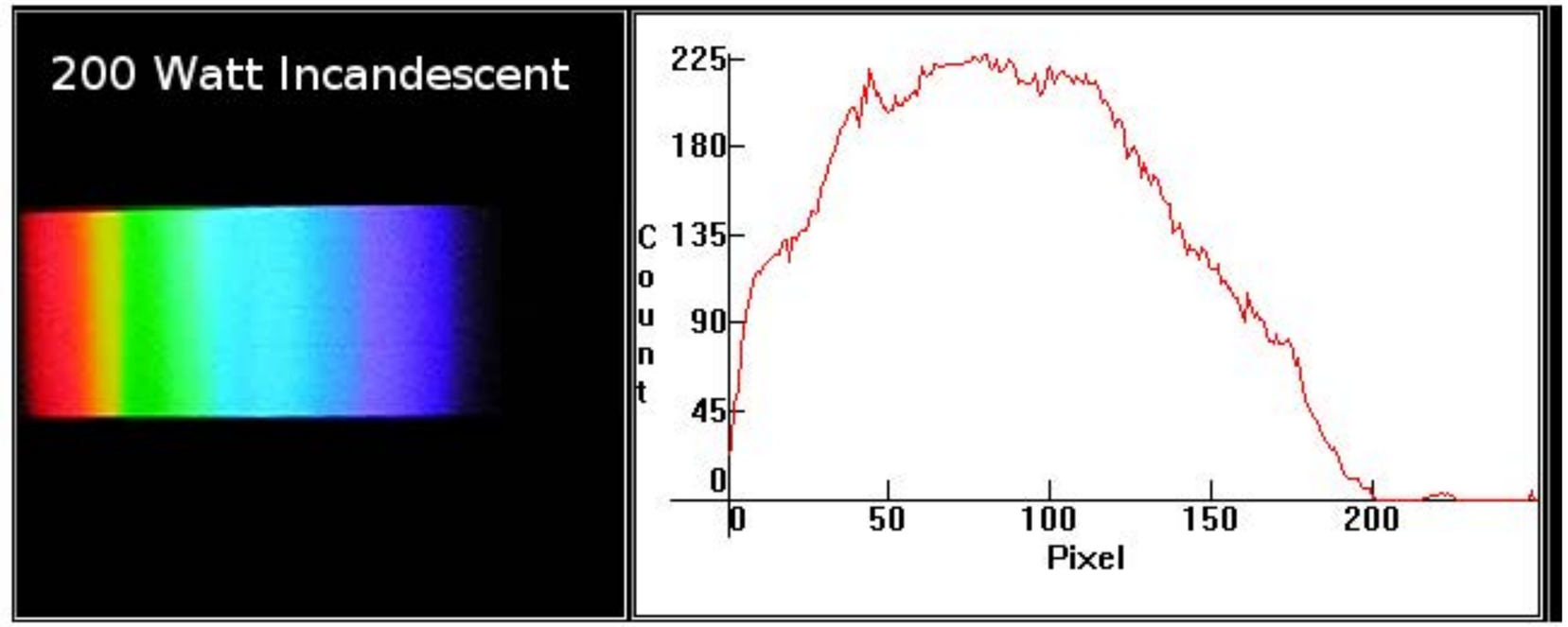


Figure credit:

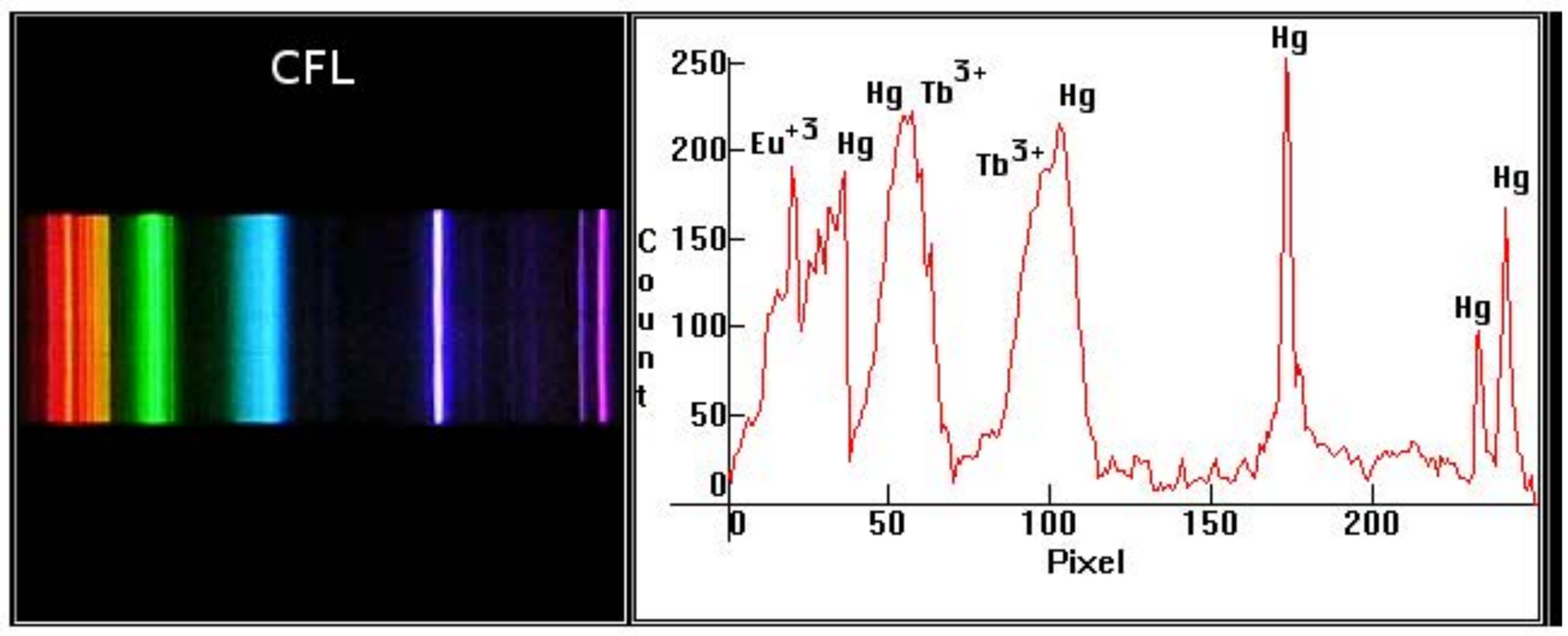
admesy
ADVANCED MEASUREMENT SYSTEMS

Emission Spectrum—Example

- Why so many different kinds of lightbulbs on the market?
- "Quality" of light:



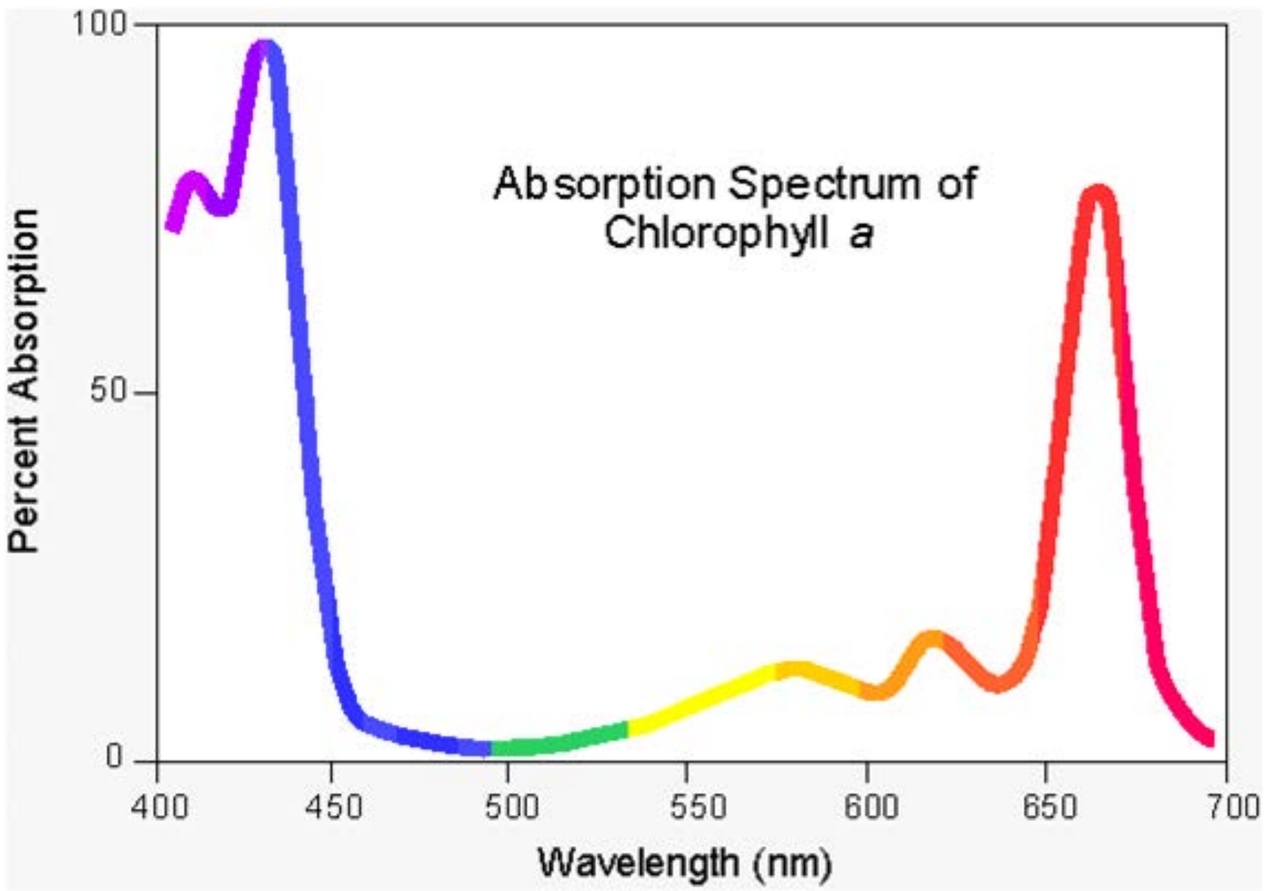
Incandescent:
+more sun-like
-power-hungry



CFL:
-"choppy" spectrum
+power efficient

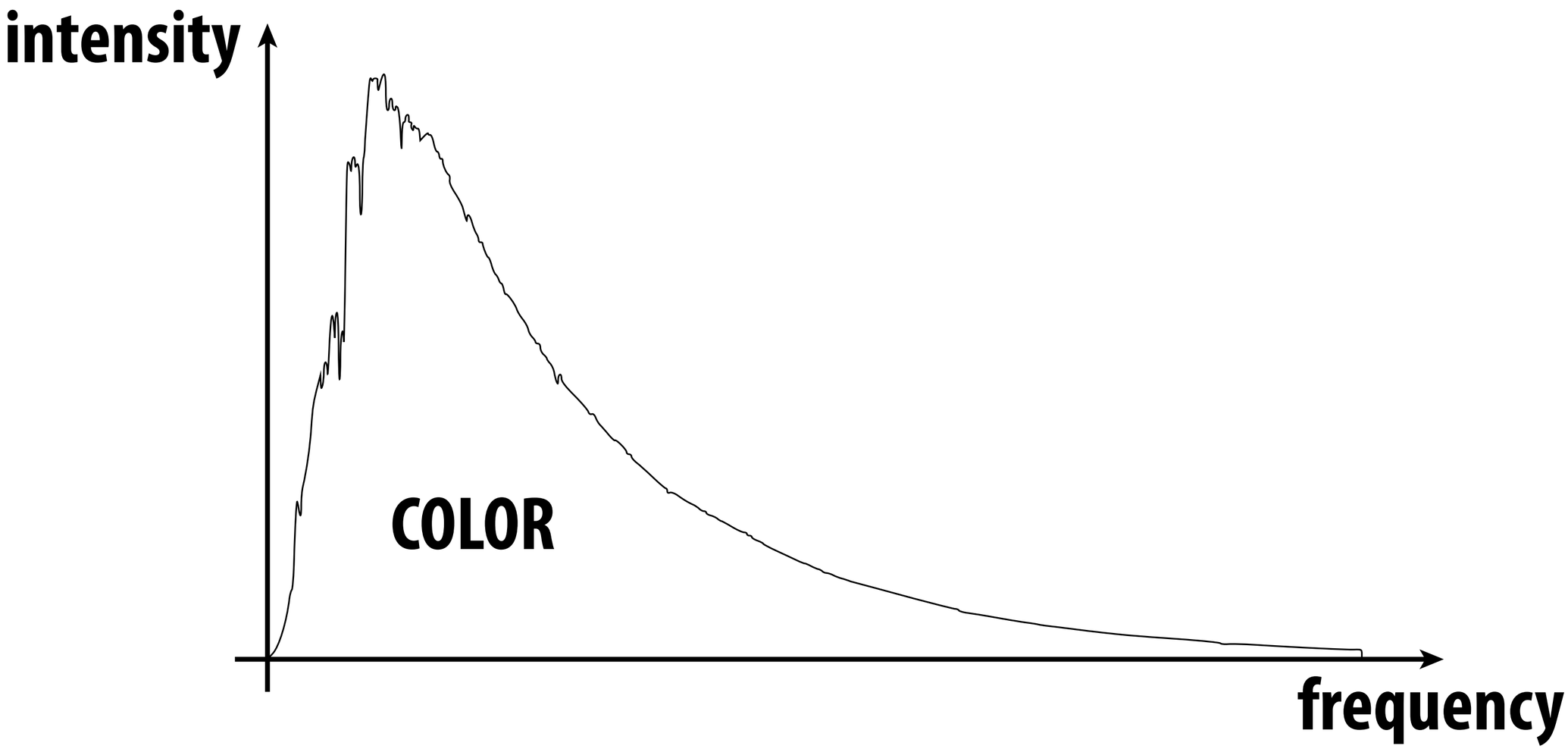
Absorption Spectrum

- 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!**

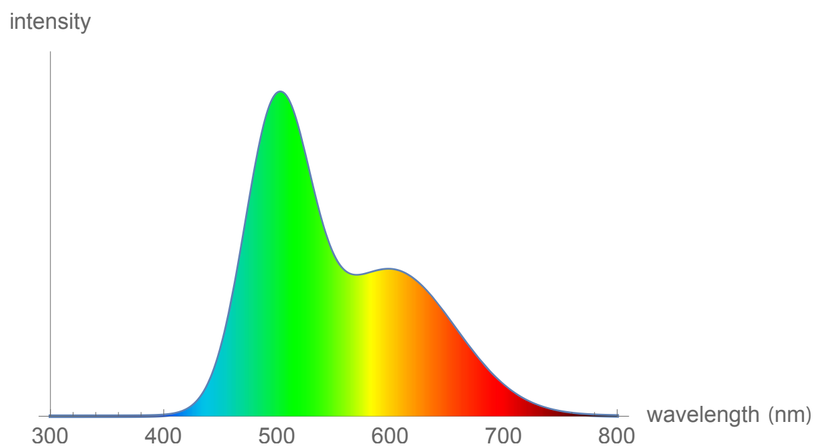
If you remember to use spectral description as a starting point, the issues surrounding color theory/ practice will make a *lot* more sense!

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!

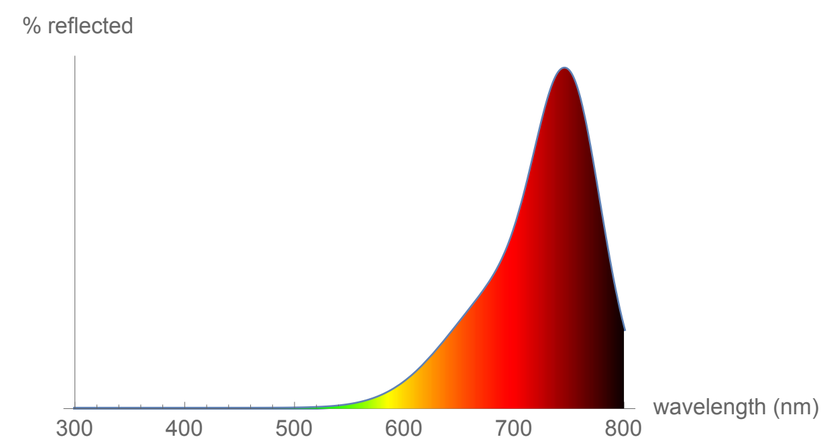
Interaction of emission and reflection

- Toy model for what happens when light gets reflected
  - ν—frequency (Greek "nu")
  - Light source has emission spectrum f(ν)
  - Surface has reflection spectrum g(ν)
  - Resulting intensity is the *product* f(ν)g(ν)

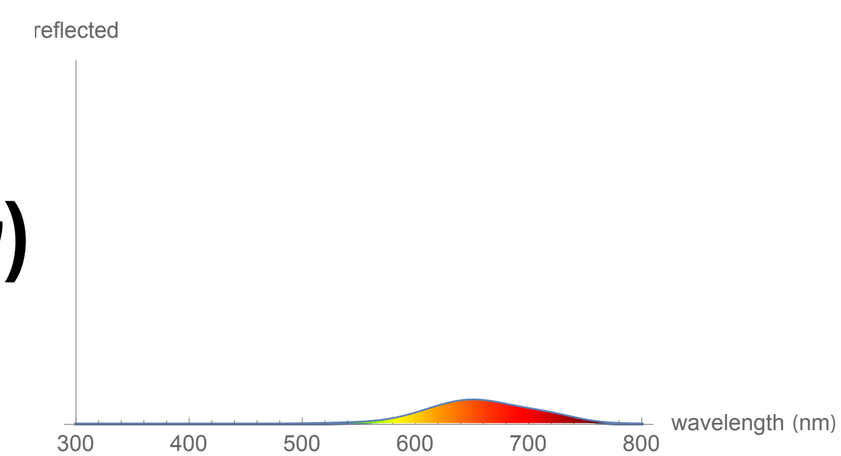
f(ν)



g(ν)

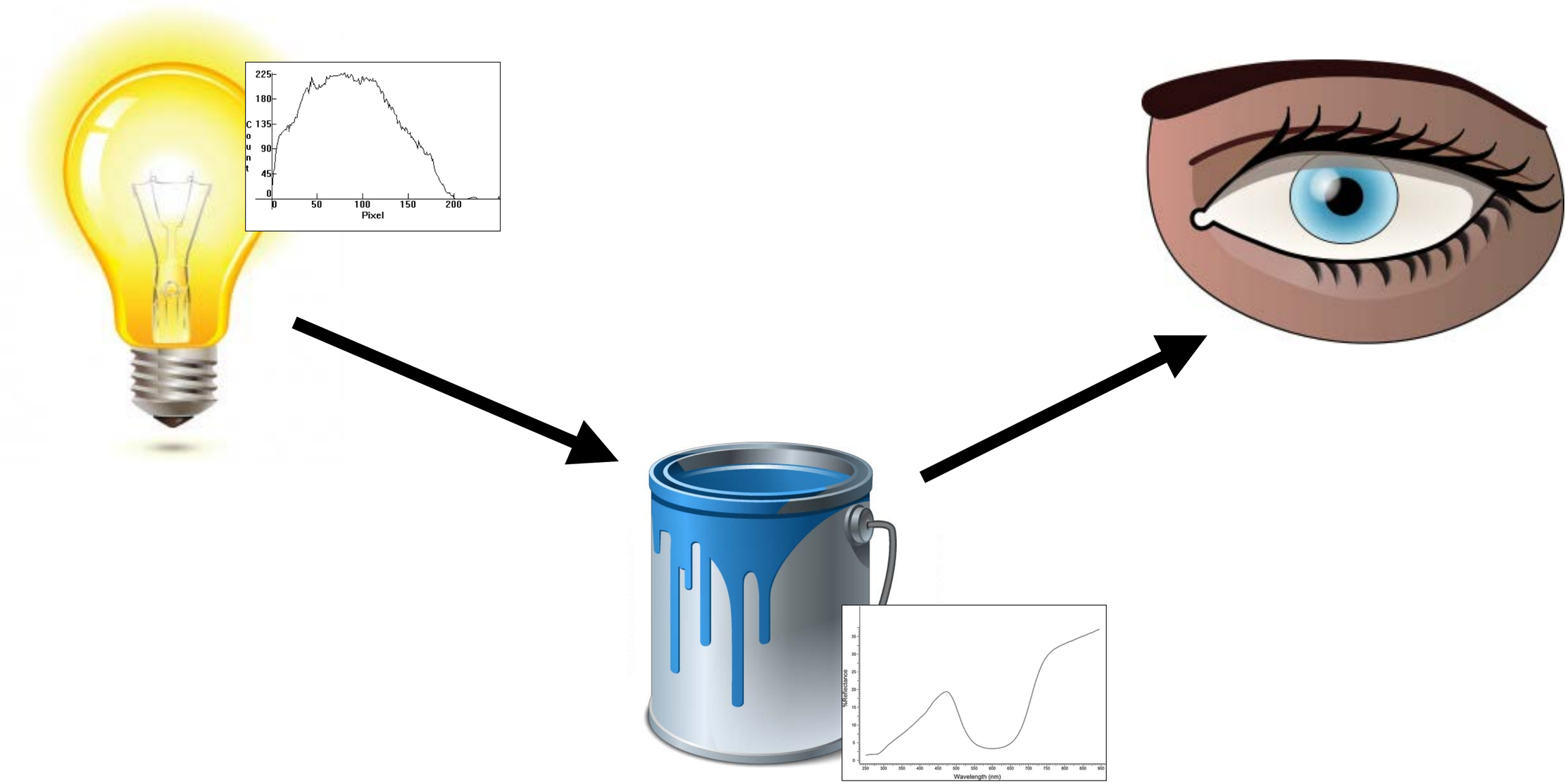


fg(ν)



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

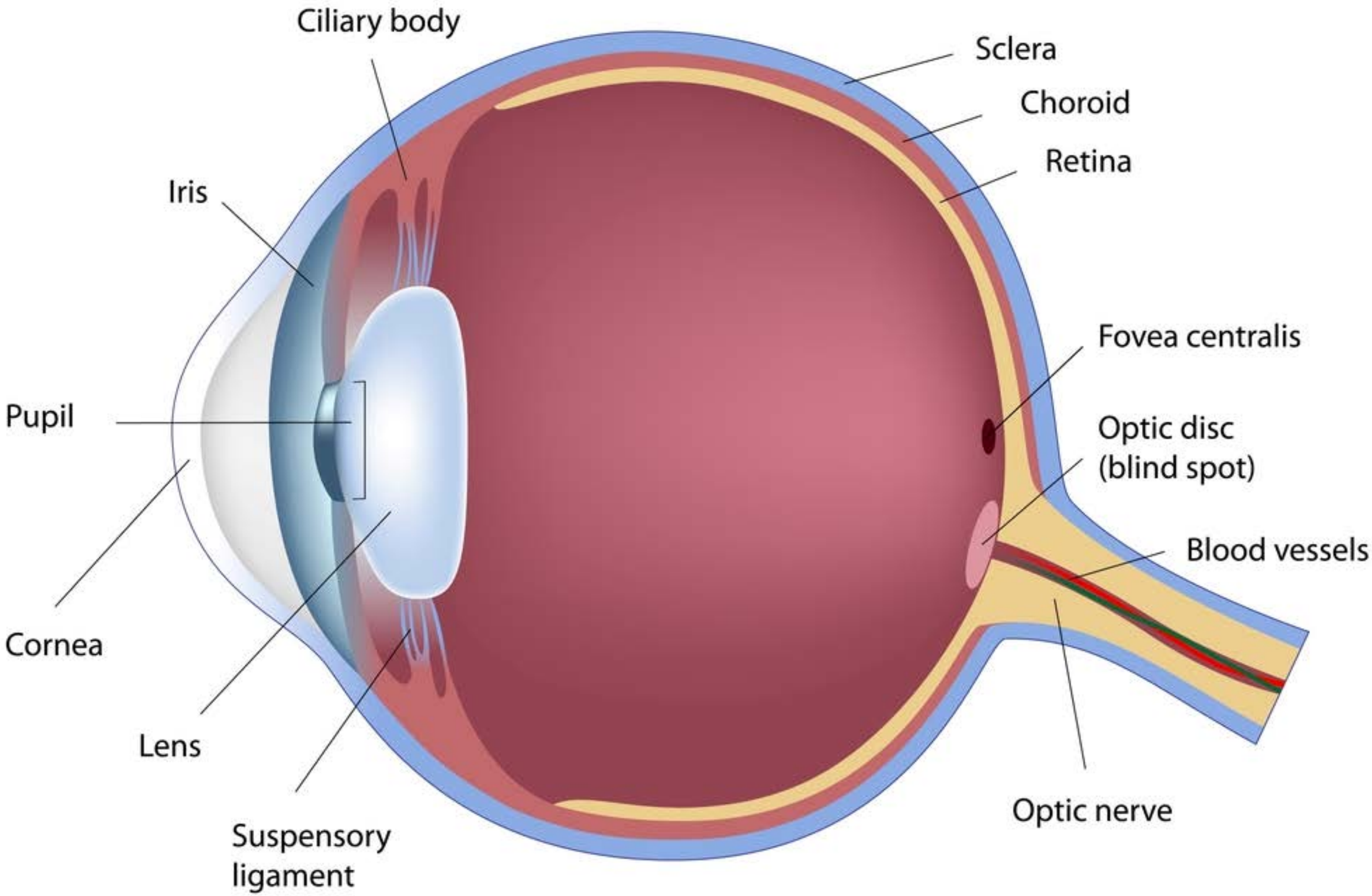


Image credit: Georgia Retina (http://www.garetina.com/about-the-eye)

The eye (optics)

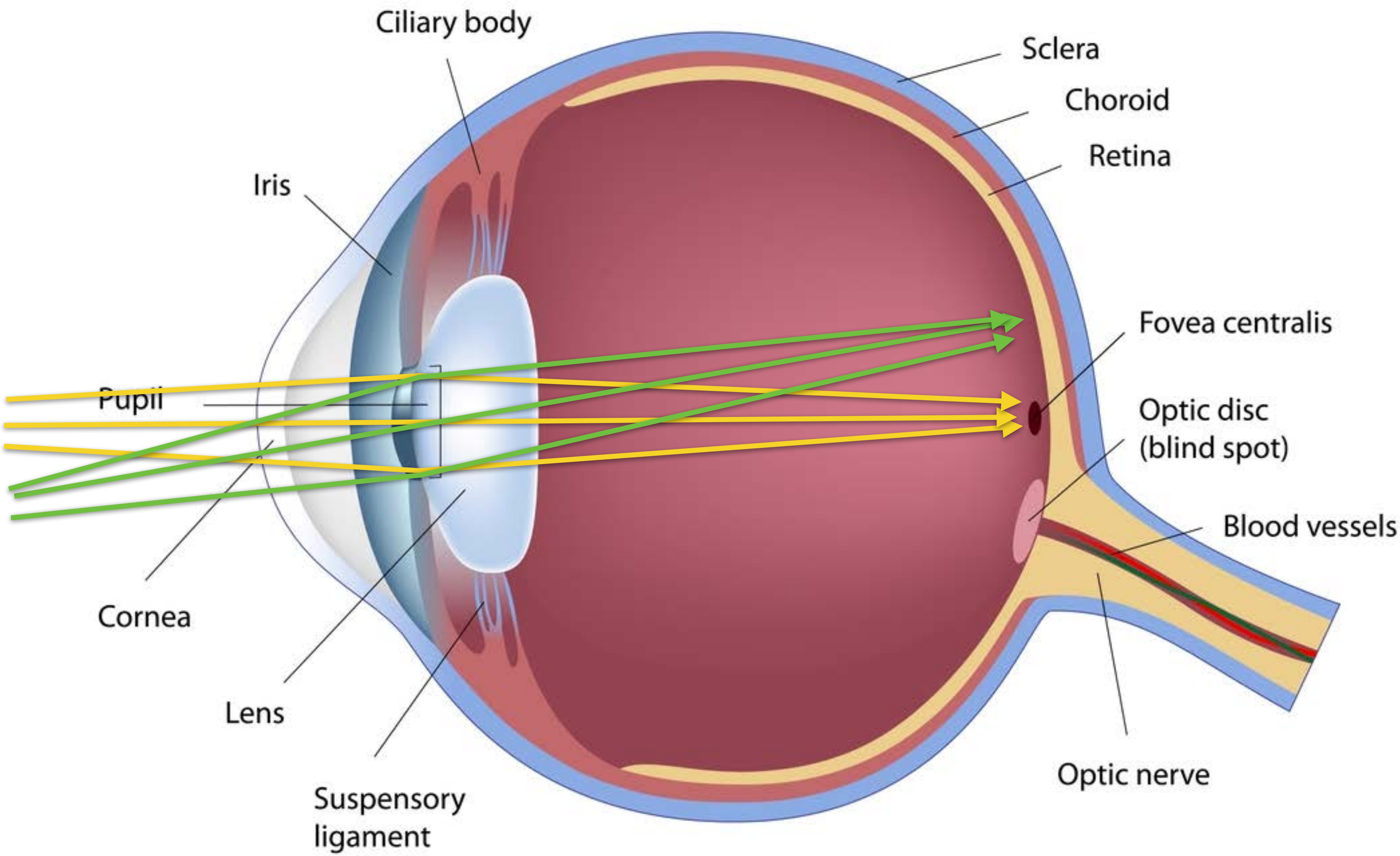


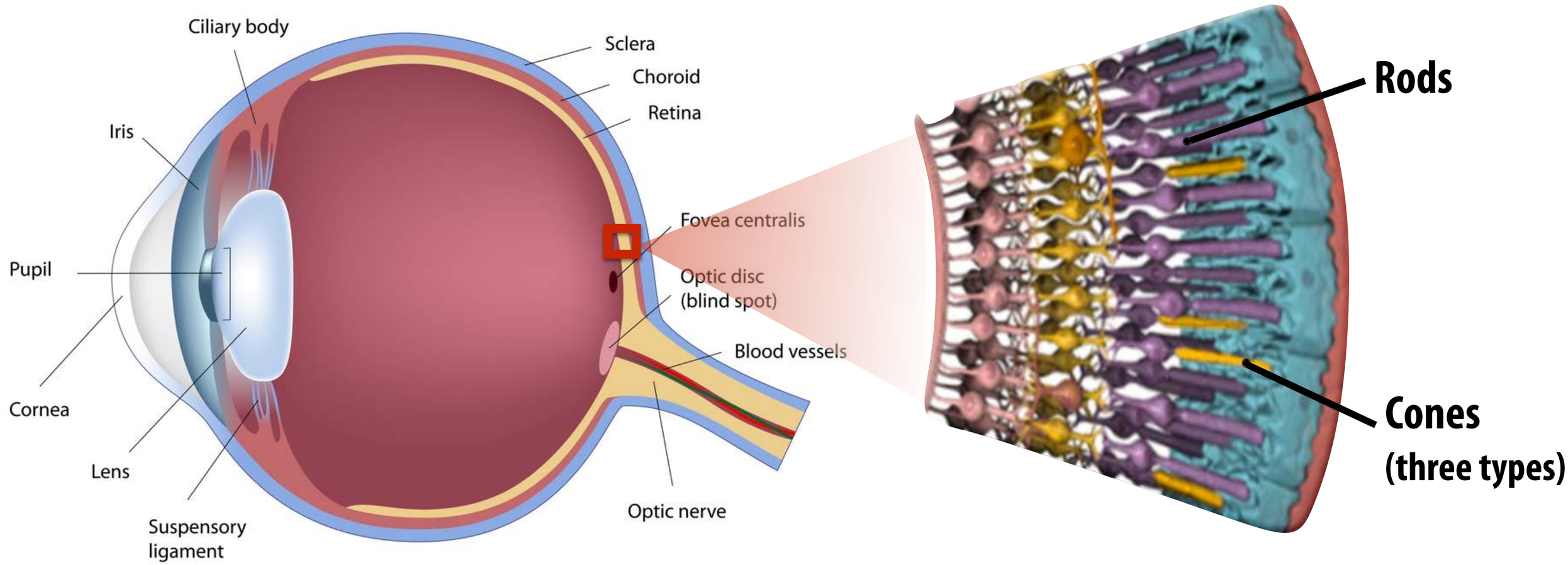
Image credit: Georgia Retina (http://www.garetina.com/about-the-eye)

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 a efficient sensor (when $f(\lambda)$ is large, a small amount of light at wavelength λ will trigger a large sensor response)
- **Total response of photosensor:**

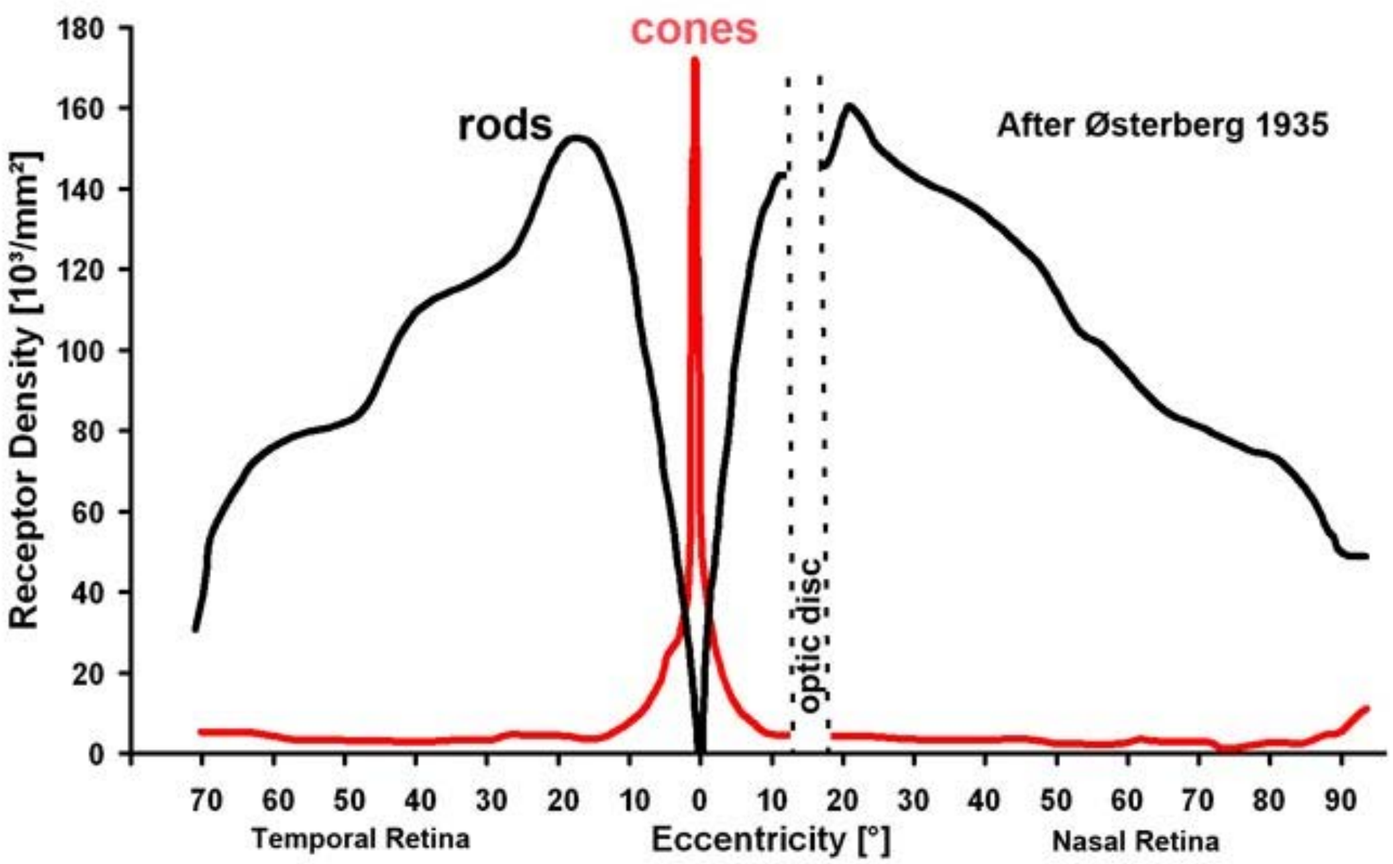
$$R = \int_{\lambda} \Phi(\lambda) f(\lambda) d\lambda$$

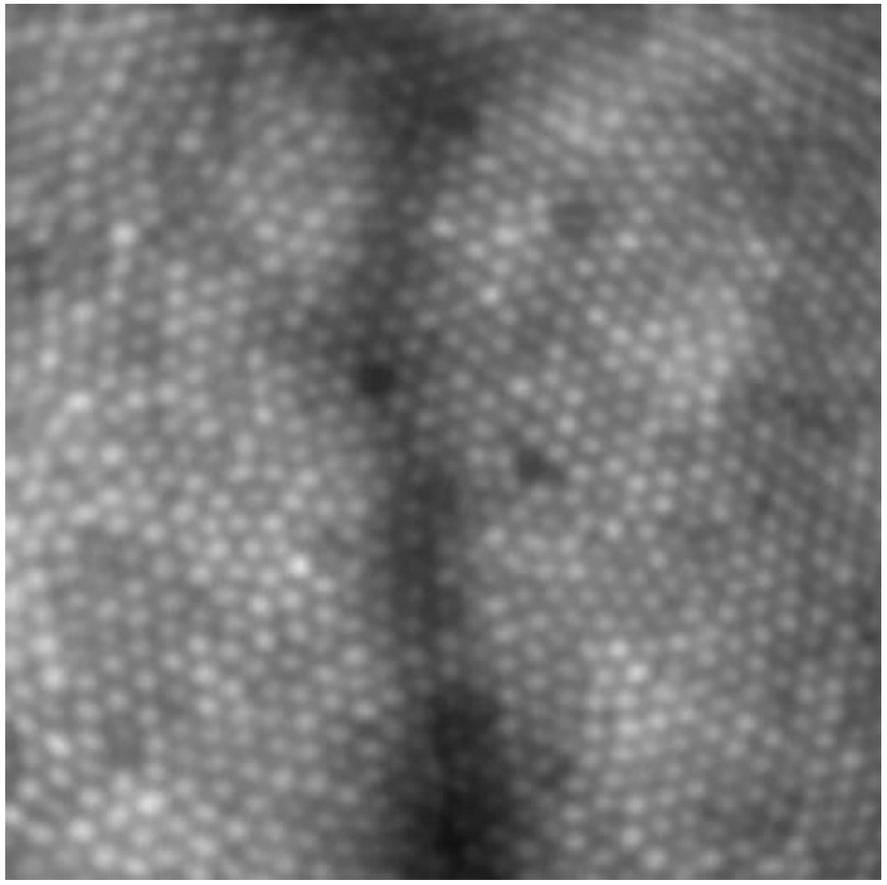
The eye's photoreceptor cells: rods & cones



- 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

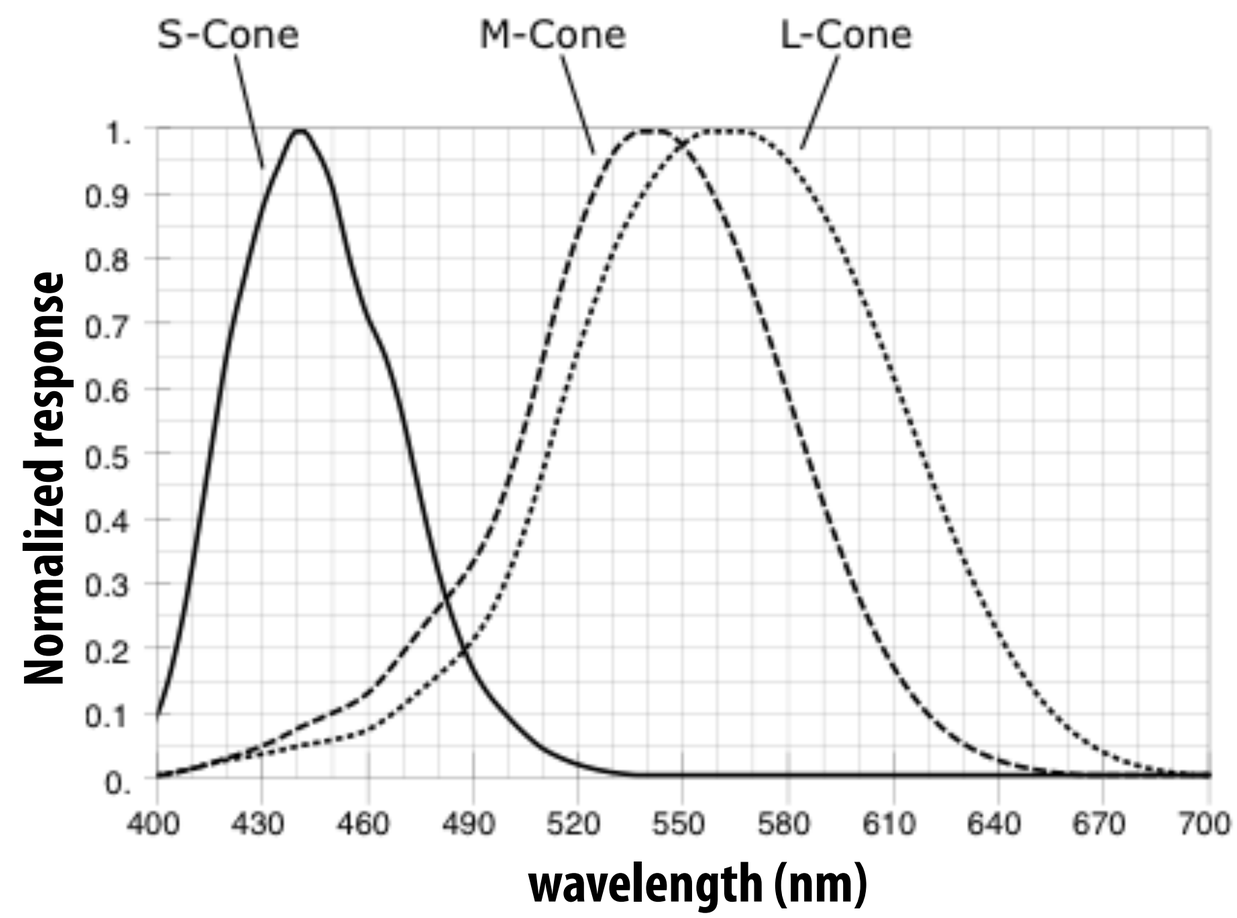
Three types of cones: S, M, and L cones (corresponding to peak response at short, medium, and long wavelengths)

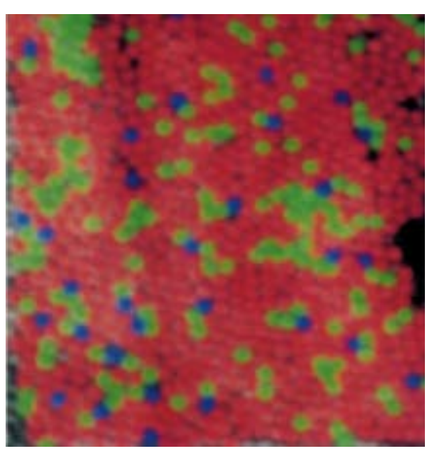
$$S = \int_\lambda \Phi(\lambda) S(\lambda) d\lambda$$

$$M = \int_\lambda \Phi(\lambda) M(\lambda) d\lambda$$

$$L = \int_\lambda \Phi(\lambda) L(\lambda) d\lambda$$

Response functions for S, M, and L cones



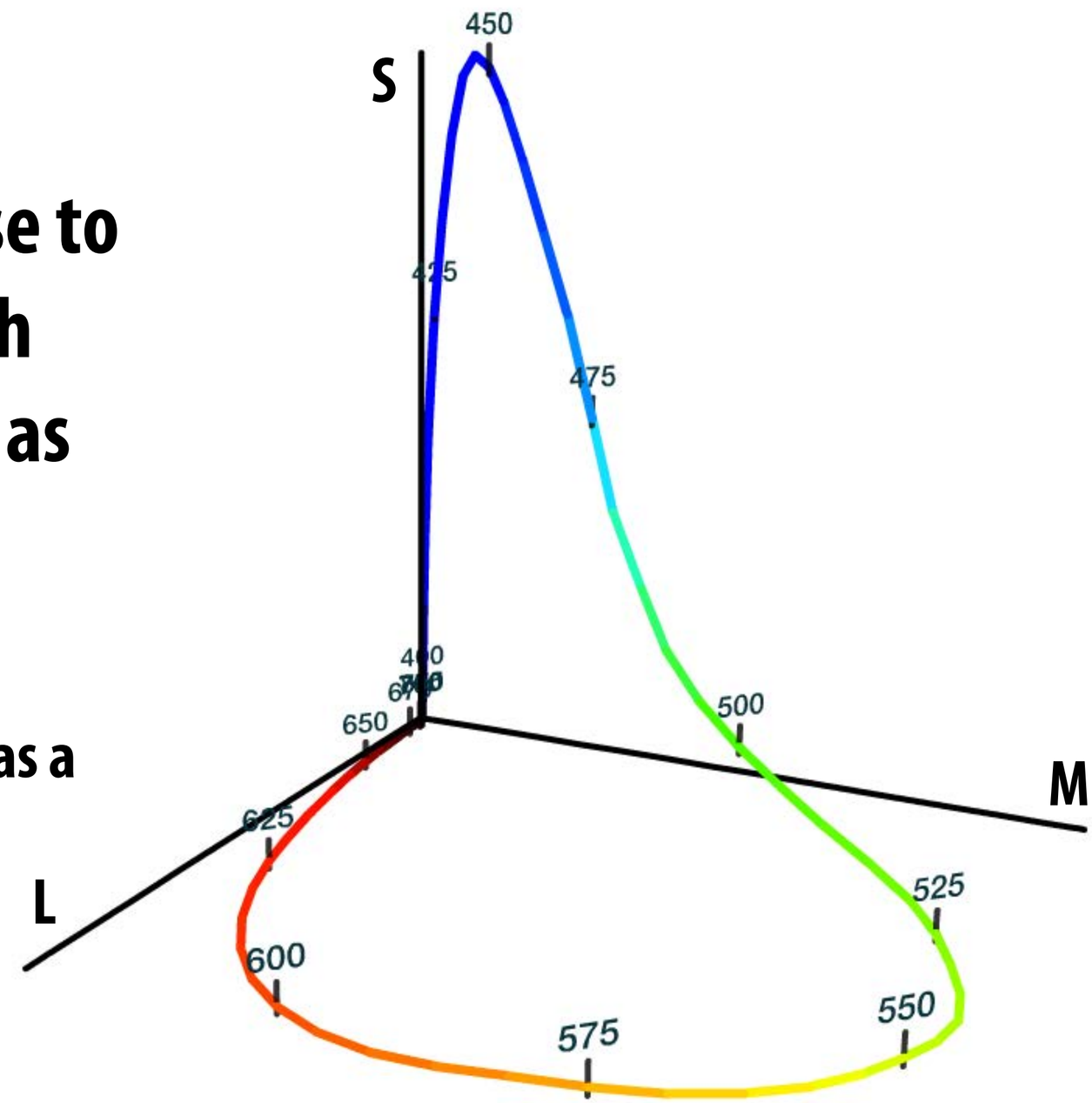


Uneven distribution of cone types in eye
~64% of cones are L cones, ~ 32% M cones

Response of S,M,L cones to monochromatic light

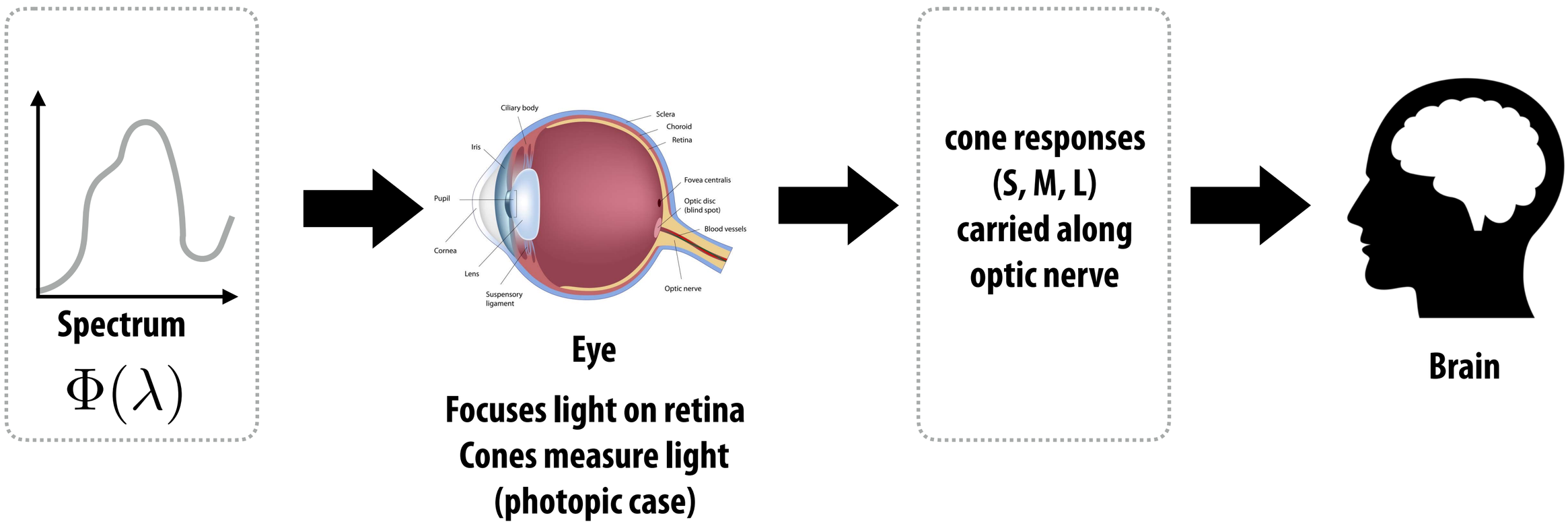
Figure visualizes cone's response to monochromatic light (light with energy in a single wavelength) as points in 3D space

(plots value of S, M, L response functions as a point in 3D space)



The human visual system

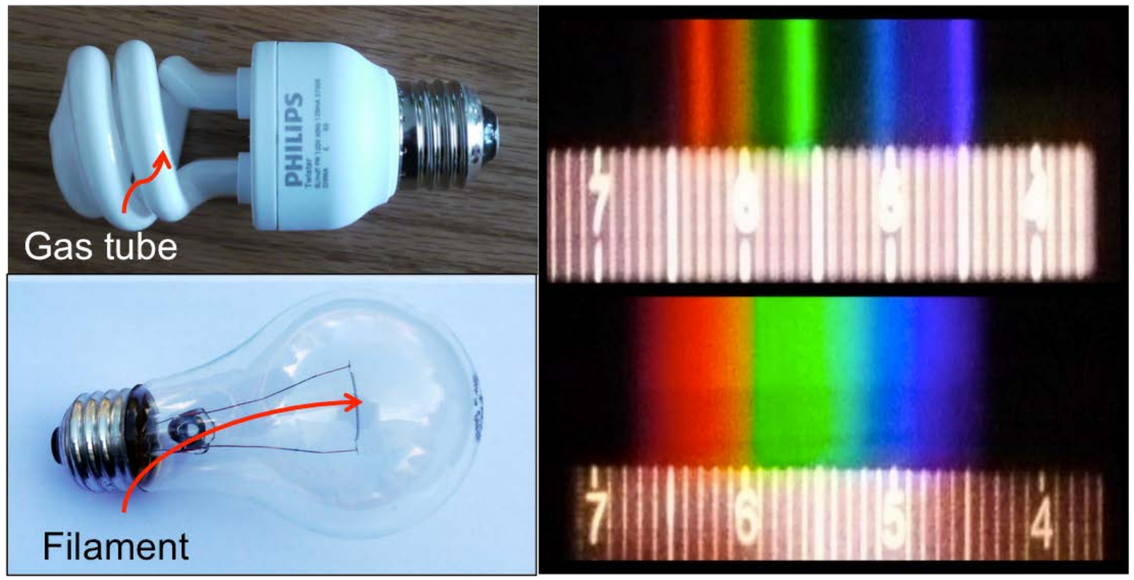
- Human eye does not directly measure the spectrum of incoming light
  - i.e., the brain does not receive "a spectrum" from the eye
- The eye measures three response values = (S, M, L). The result of integrating the incoming spectrum against response functions of S, M, L-cones



Q: Is it possible for two functions to integrate to the same value?

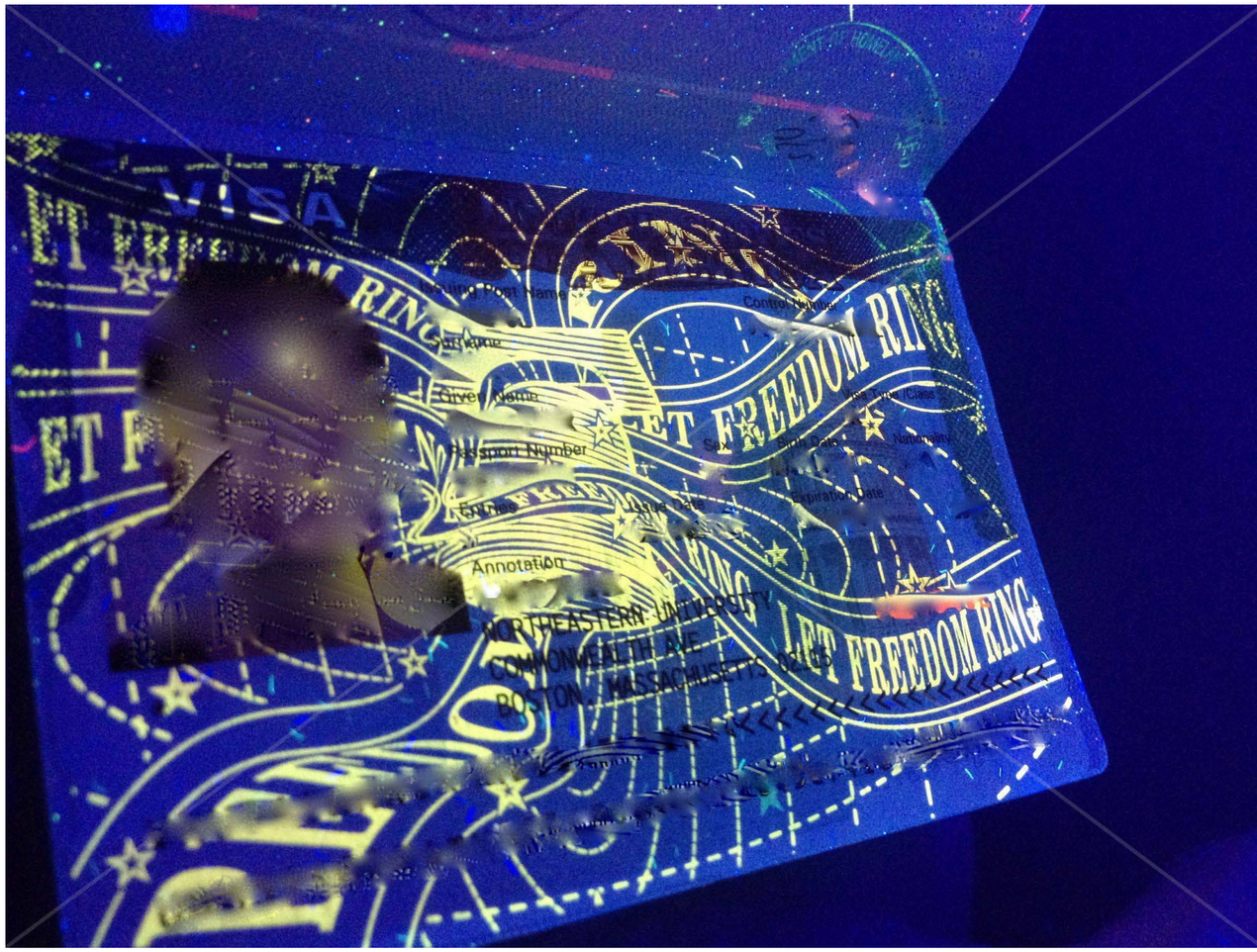
Metamers

- 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:

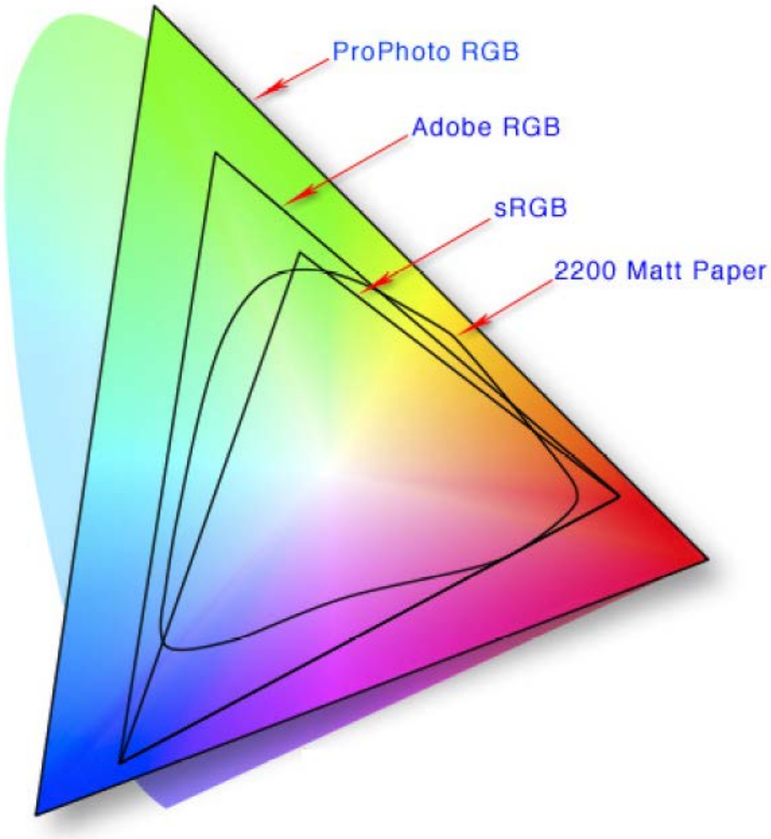


Ok, so color can get pretty complicated!

How do we encode it in a simple(r) way?

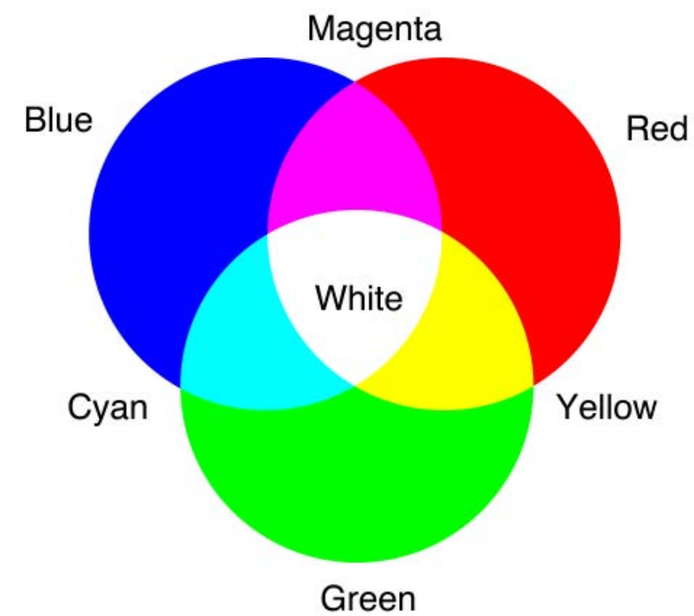
Color Spaces and Color Models

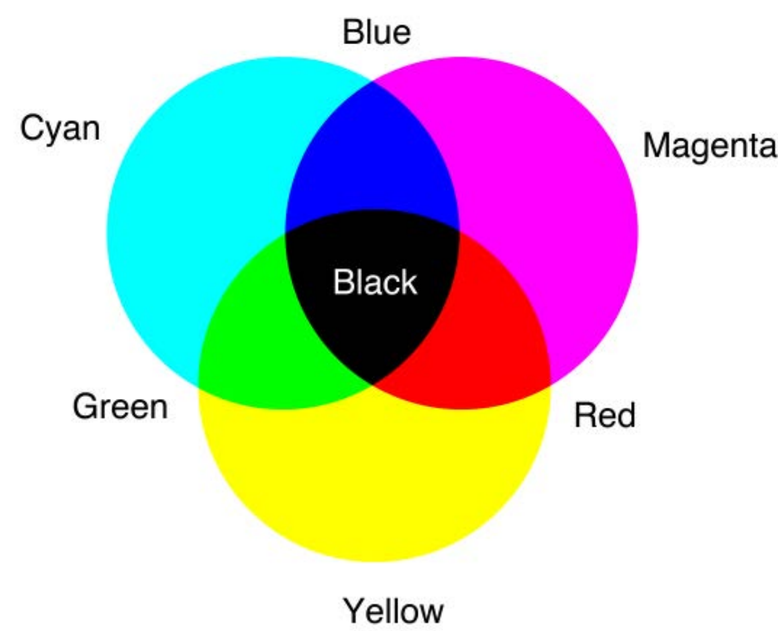
- Many ways to specify a color
  - storage
  - convenience
- In general, specify a color from some *color space* using a *color model*
- *Color <u>space</u>* is like artist's palette: full range of colors we can choose from
- *Color <u>model</u>* is the way a particular color in a color space is specified:
  - artist's palette: "yellow ochre"
  - RGB color model: 204, 119, 34



Additive vs. Subtractive Color Models

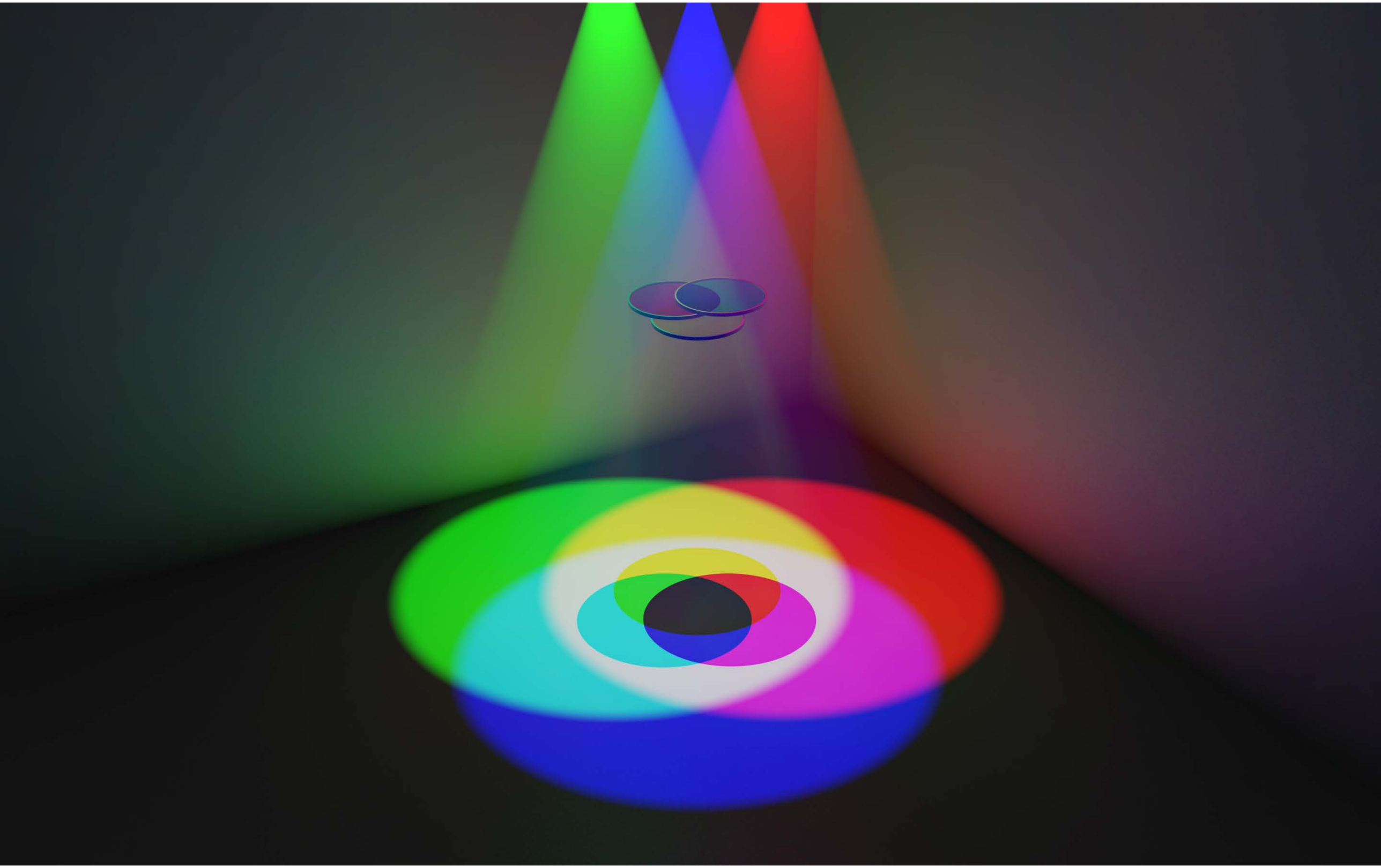
- Just like we had emission & absorption spectra, we have *additive* and *subtractive** color models
- Additive
  - Used for, e.g., combining colored lights
  - Prototypical example: RGB
- Subtractive
  - Used for, e.g., combining paint colors
  - Prototypical example: CMYK





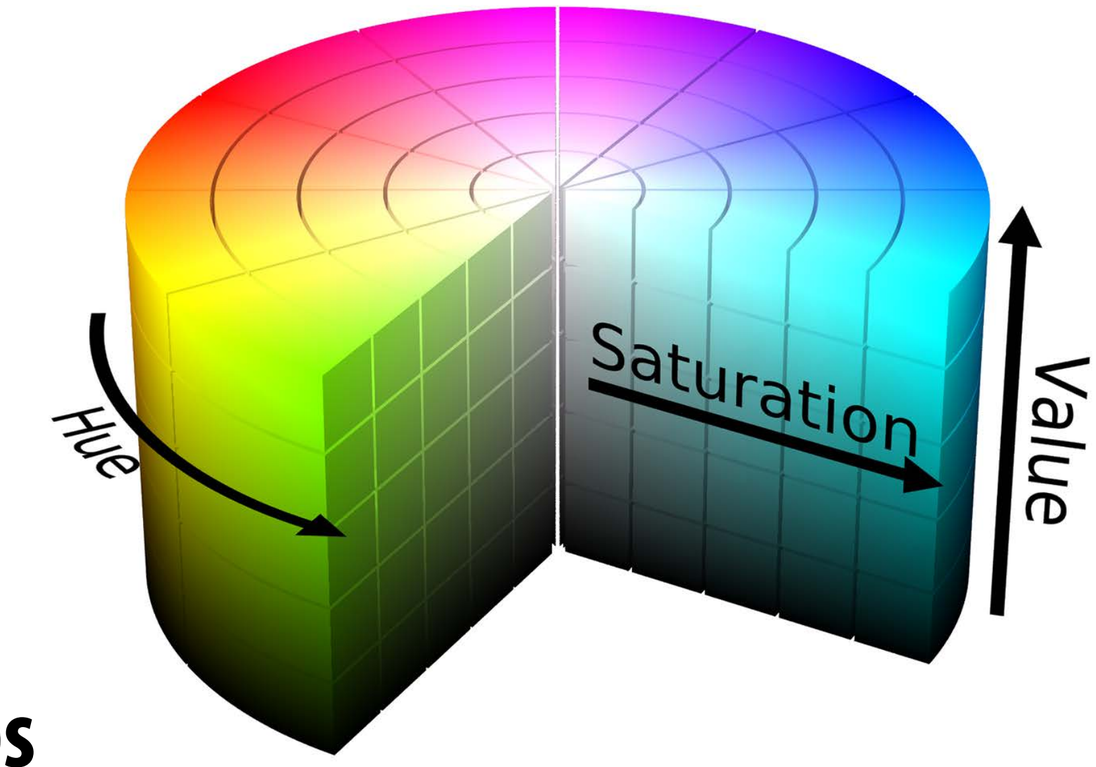
*A better name than subtractive might be *multiplicative*, since we *multiply* to get the final color!

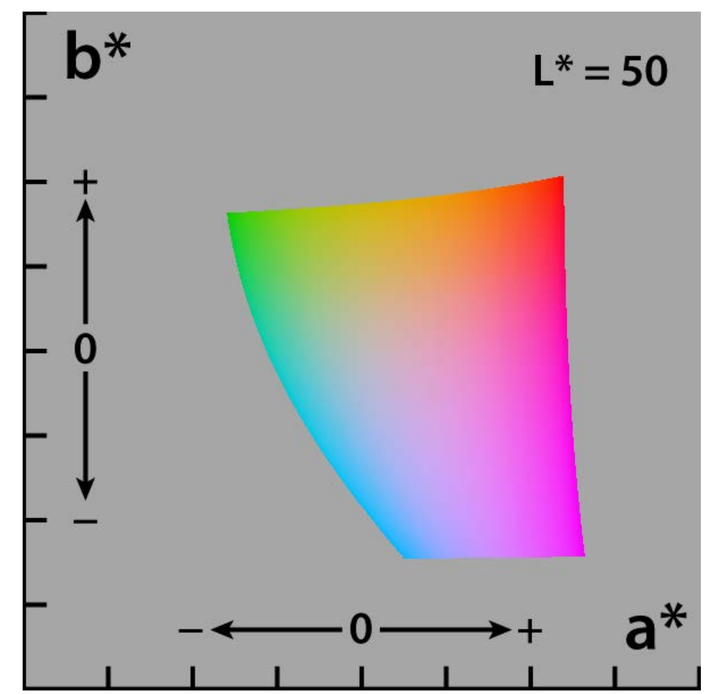
Let’s shed some light on this picture…



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—preceptually-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*16 = 256 values
- Q: Roughly what color is `#ff6600`?

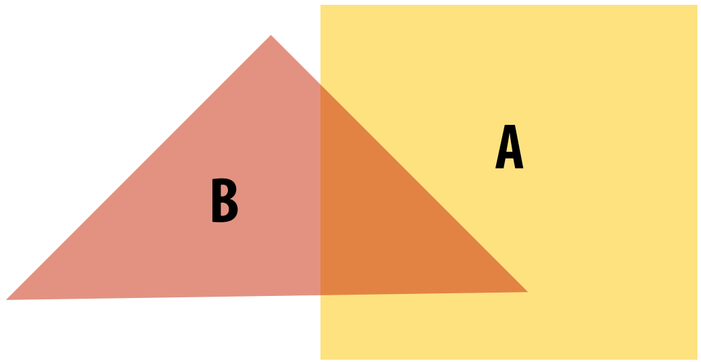
*Upper vs. lowercase letters? Makes absolutely *no* difference!

Other Ways of Specifying Color?

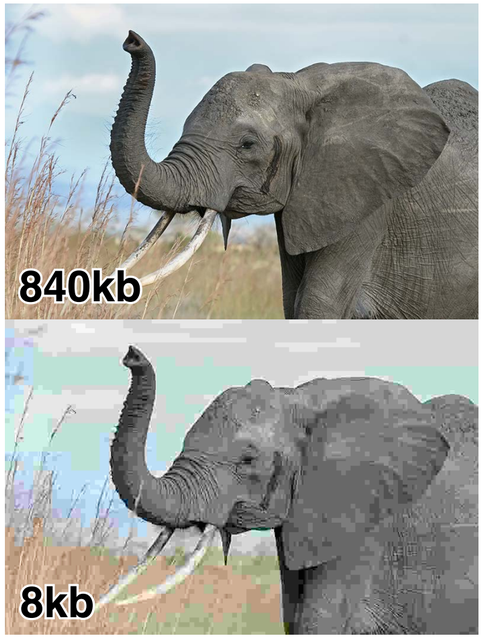
- Other color specifications not based on continuous color space
- E.g., Pantone Matching System
  - industry standard (proprietary)
  - 1,114 colors
  - Combination of 13 base pigments
- And not to forget…

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!

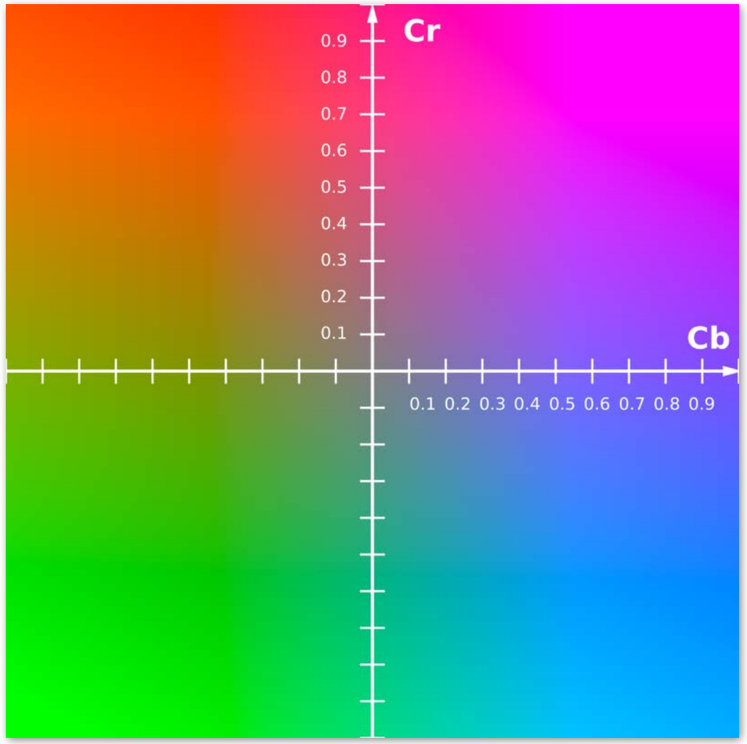


B over A



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)

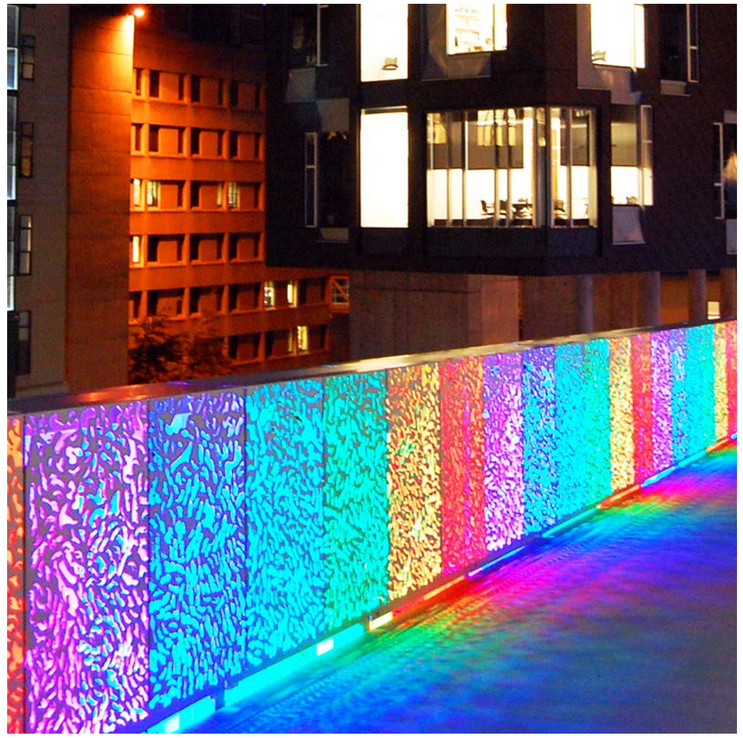
Original picture

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

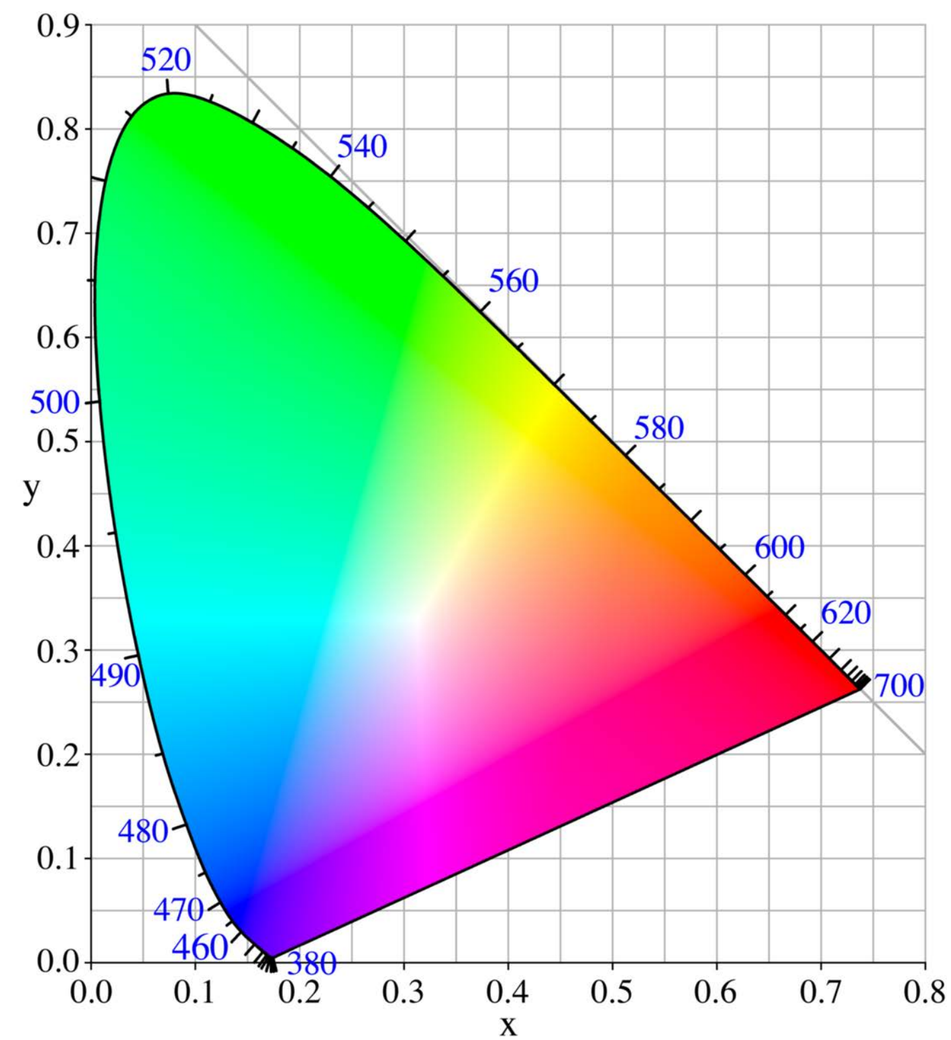
CMYK

Which raises a very important question:

Which actual colors (i.e., *spectra*) do these values get mapped to?

CIE 1931* Color Space

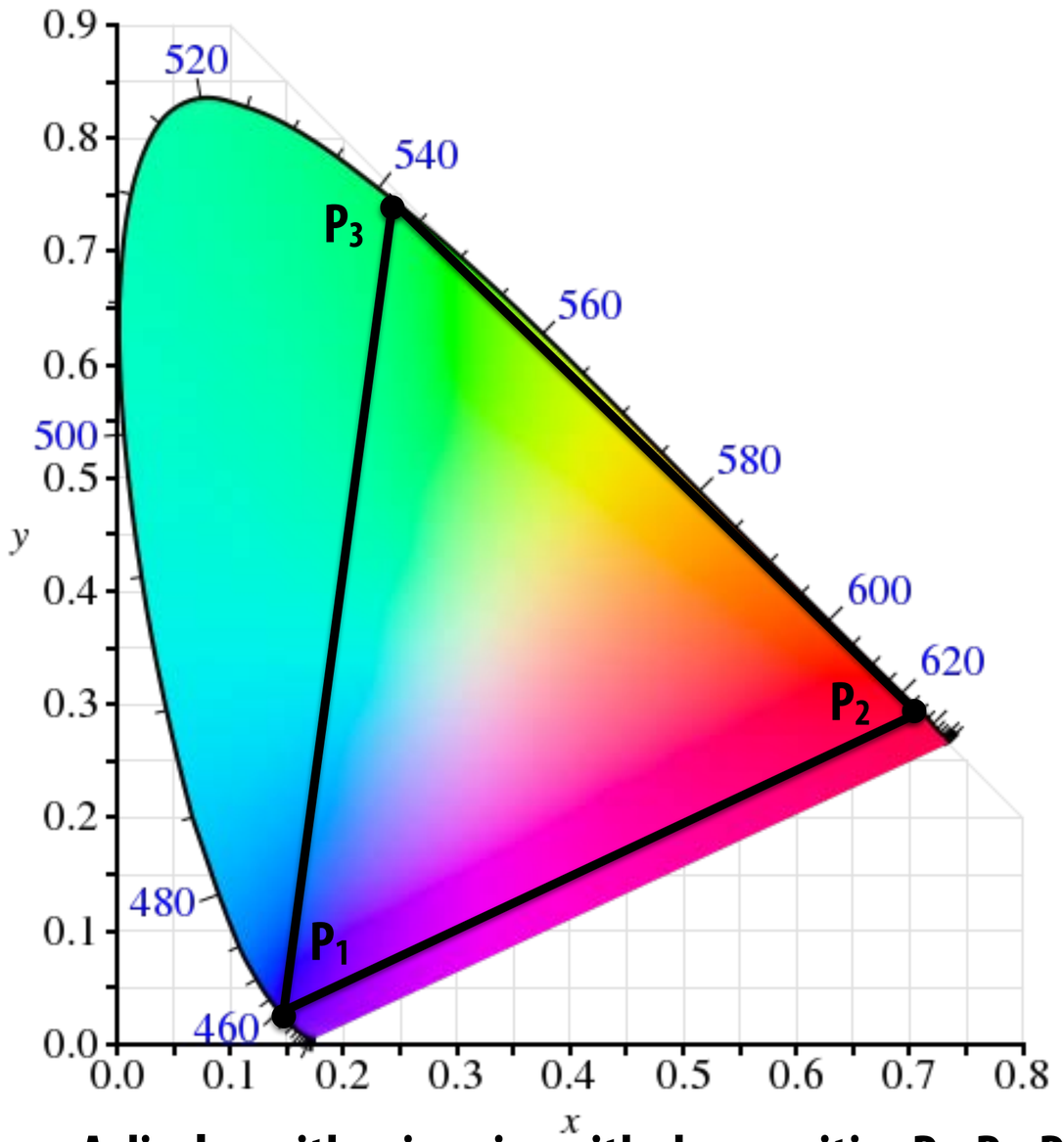
- Standard "reference" color space
- Encompasses all colors visible by "most" human observers
  - associated color *model* (XYZ) captures perceptual effects
  - e.g., perception of color ("chromaticity") changes w/ brightness ("luminosity")
  - different from specifying direct simulation of cones (SML)



*CIE 1931 does not mean anything important: "created in 1931 by the Commission Internationale de l'Éclairage"

Chromaticity Diagrams

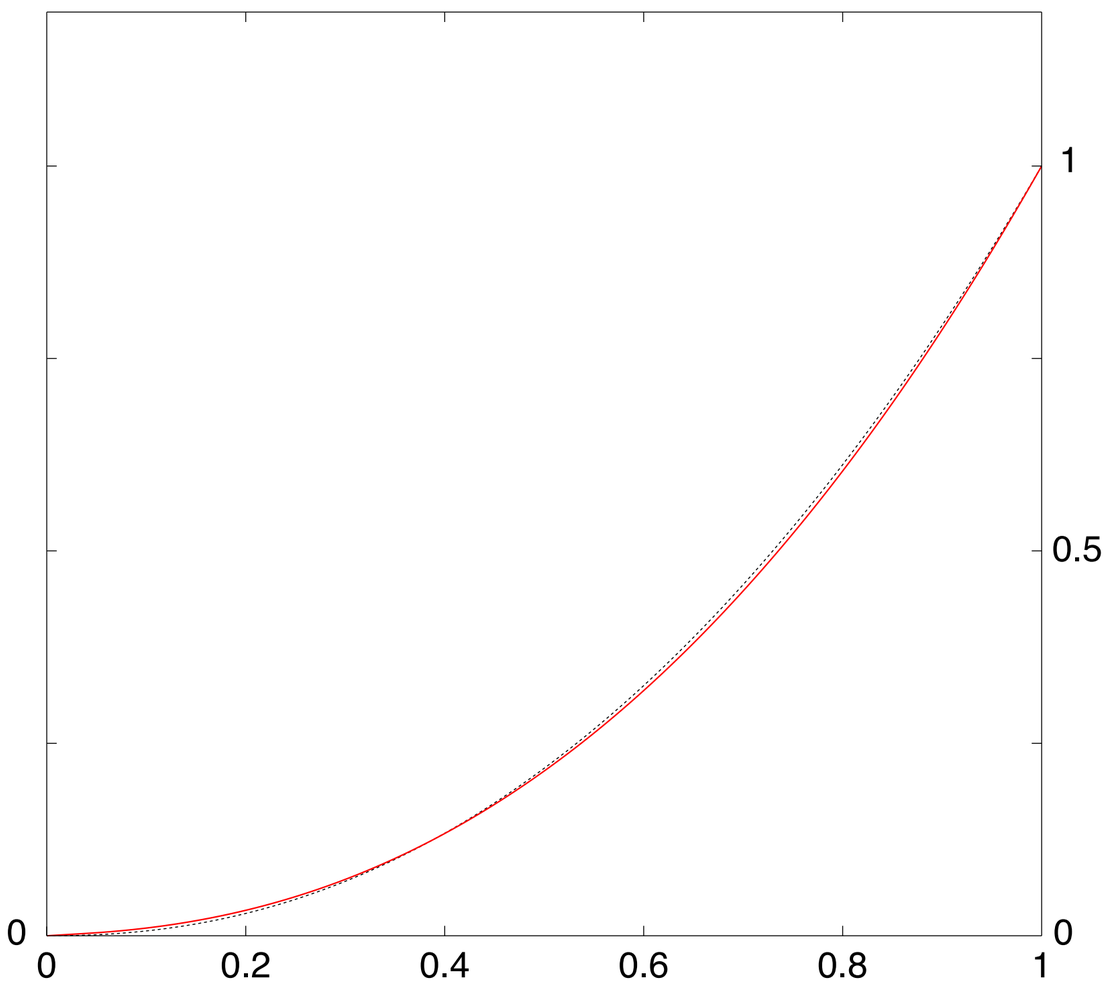
- Chromaticity is the intensity-independent component of a color
- Chromaticity diagram used to visualize extent of a color space

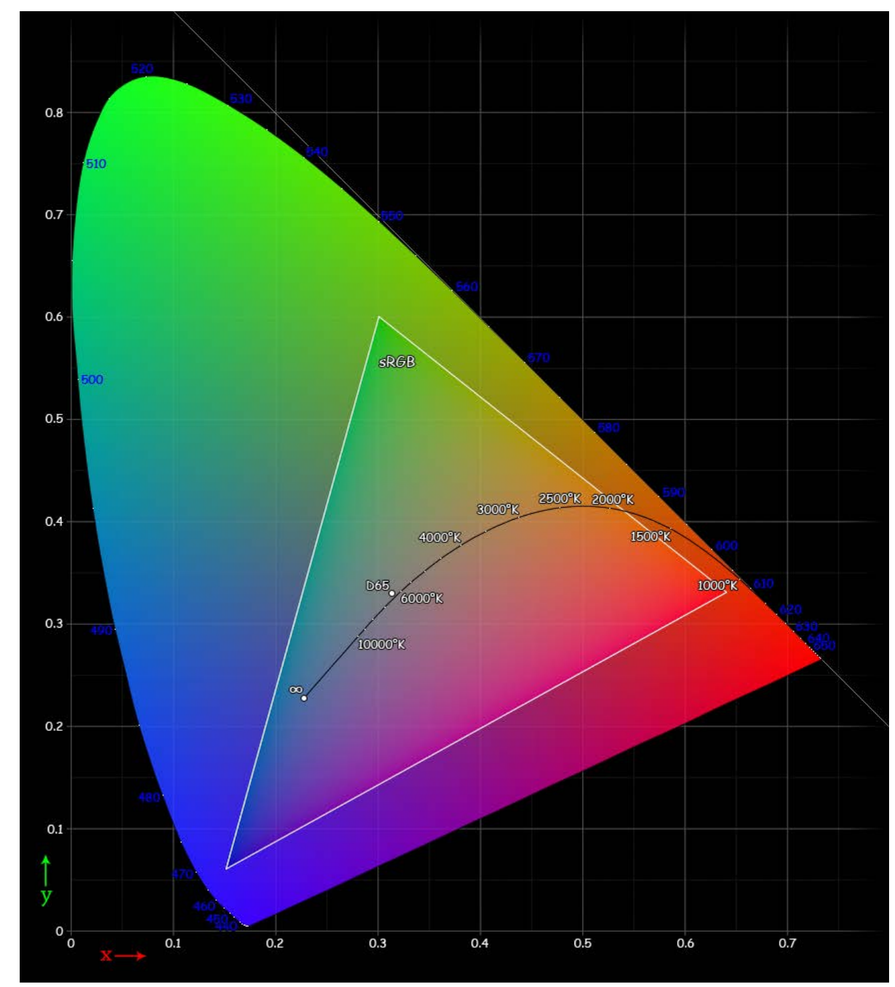


A display with primaries with chromacities P_1, P_2, P_3 can create colors that are combinations of these primaries (colors that fall within the triangle)

sRGB Color Space

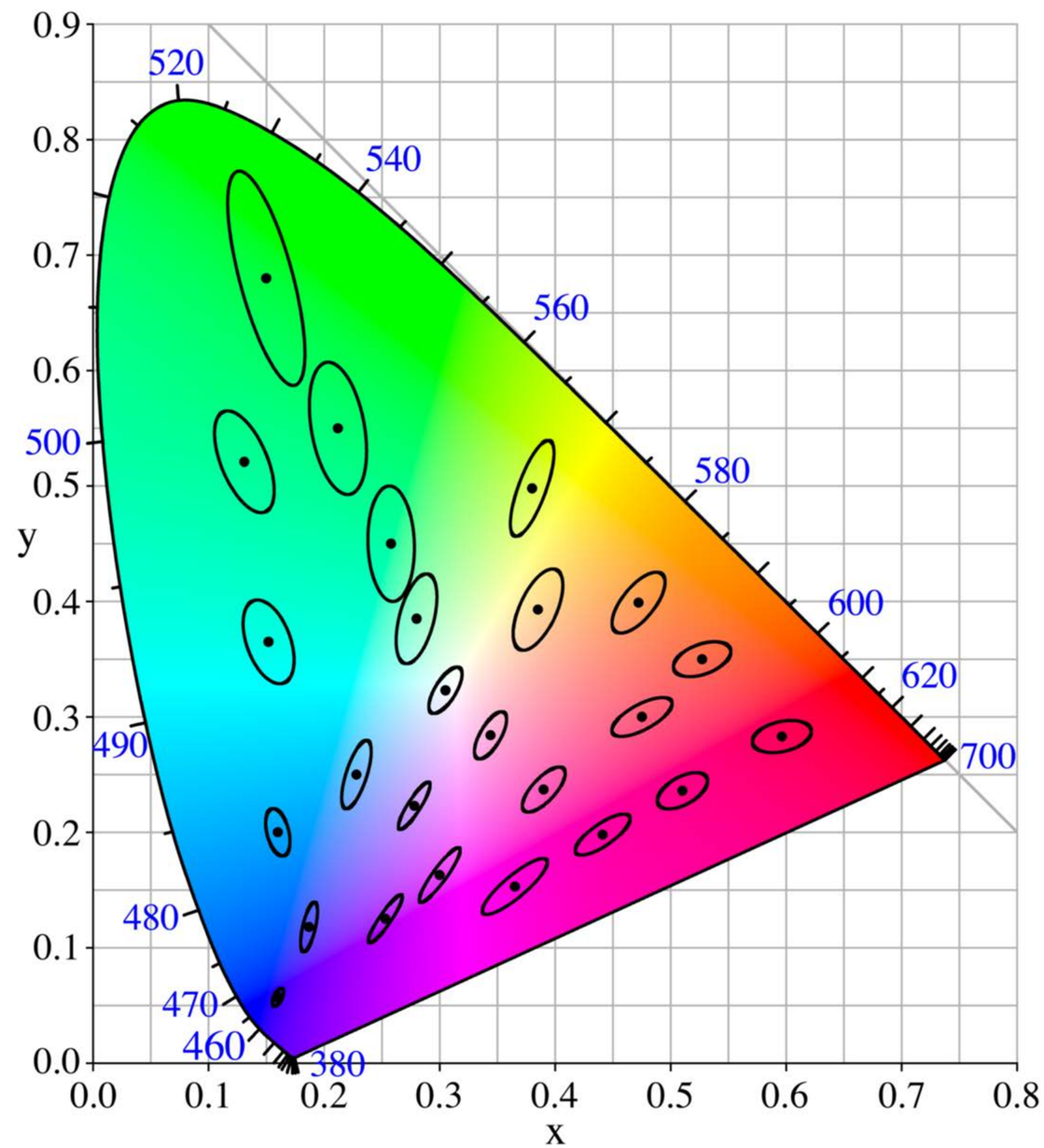
- CIE 1934 captured all possible human-visible colors
- sRGB (roughly) subset of colors available on displays, printers, …
- Nonlinear relationship between stored RGB values & intensity
  - Makes better use of limited set of numerical values





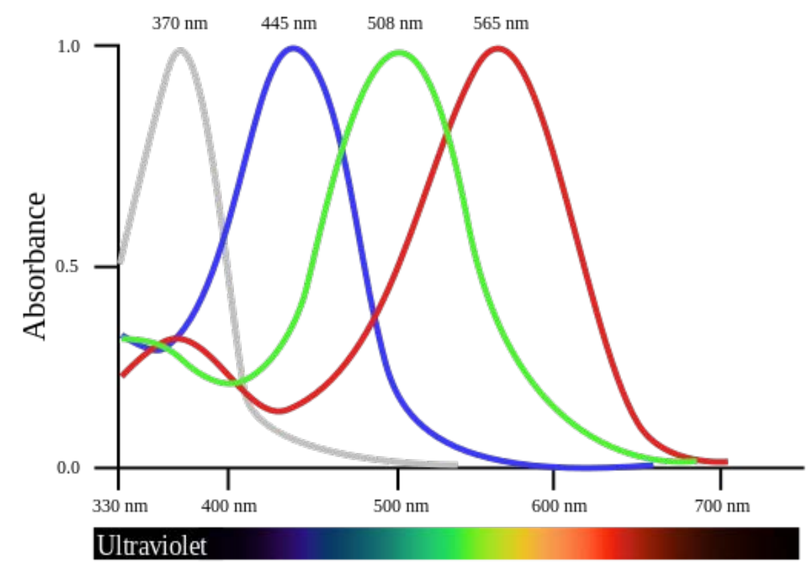
Color Acuity (MacAdam Ellipse)

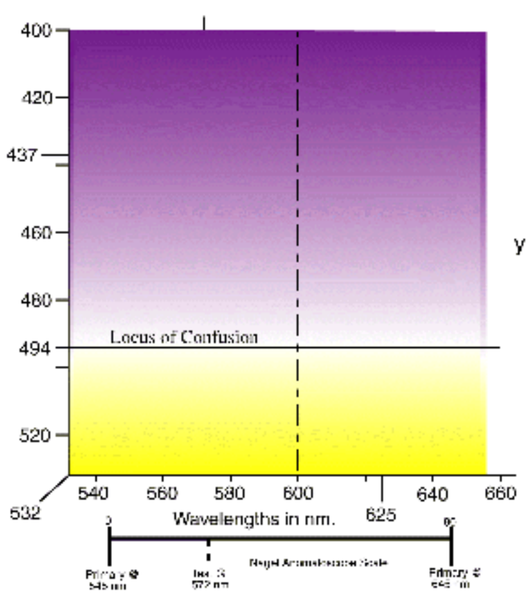
- In addition to *range* of colors visible, one might be interested in how *sensitive* people are to changes in color
- Each ellipse corresponds to a region of "just noticeable differences" of color (chromaticity)
- So, if you want to make two colors distinct, at bare minimum should avoid overlapping ellipses…

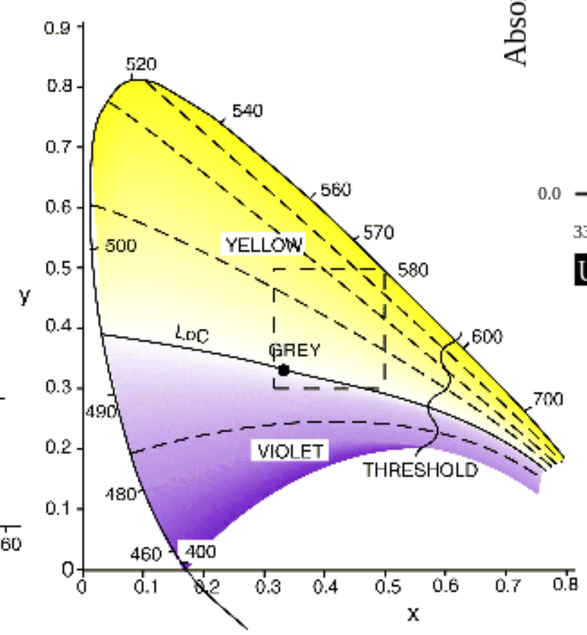


Nonstandard Color Vision

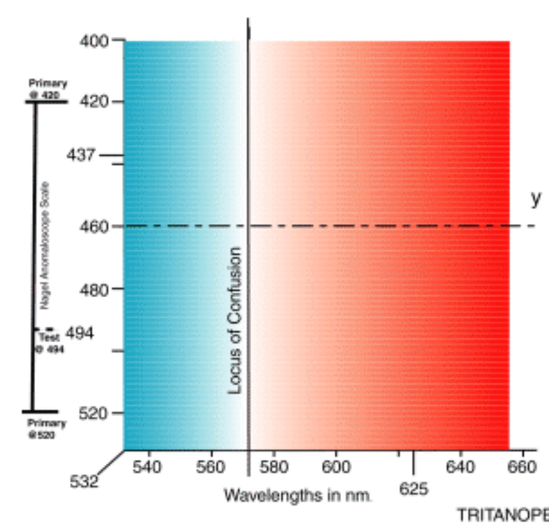
- Morphological differences in eye can cause people (& animals) to see different ranges of color (e.g., more/fewer cone types)
- Alternative chromaticity diagrams help visualize color gamut, useful for designing, e.g., widely-accessible interfaces

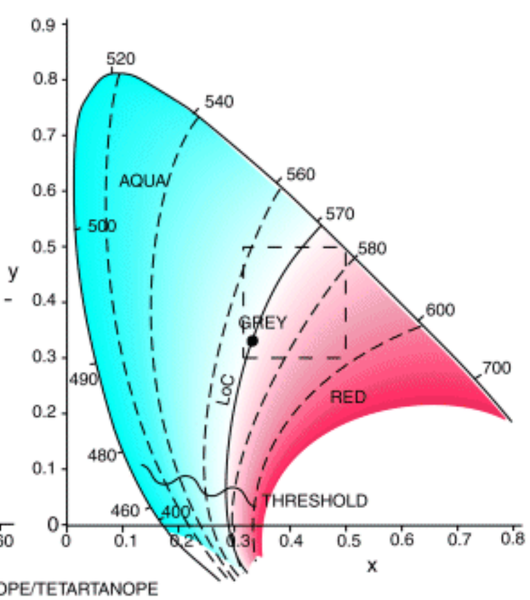






PROTANOPE/DUETRANOPE
Failure in the P-channel





TRITANOPE/TETARTANOPE
Failure in the Q-channel

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
  - Adobe Color Management, …

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: $\gamma \approx 2.5$

(Doesn't apply to modern LCD displays, whose luminance output is linearly proportional to input; DOES still apply to other devices, like sensors, etc.)

So if pixels store Y, what will the display's output look like?

Fix: pixels sent to display must store:

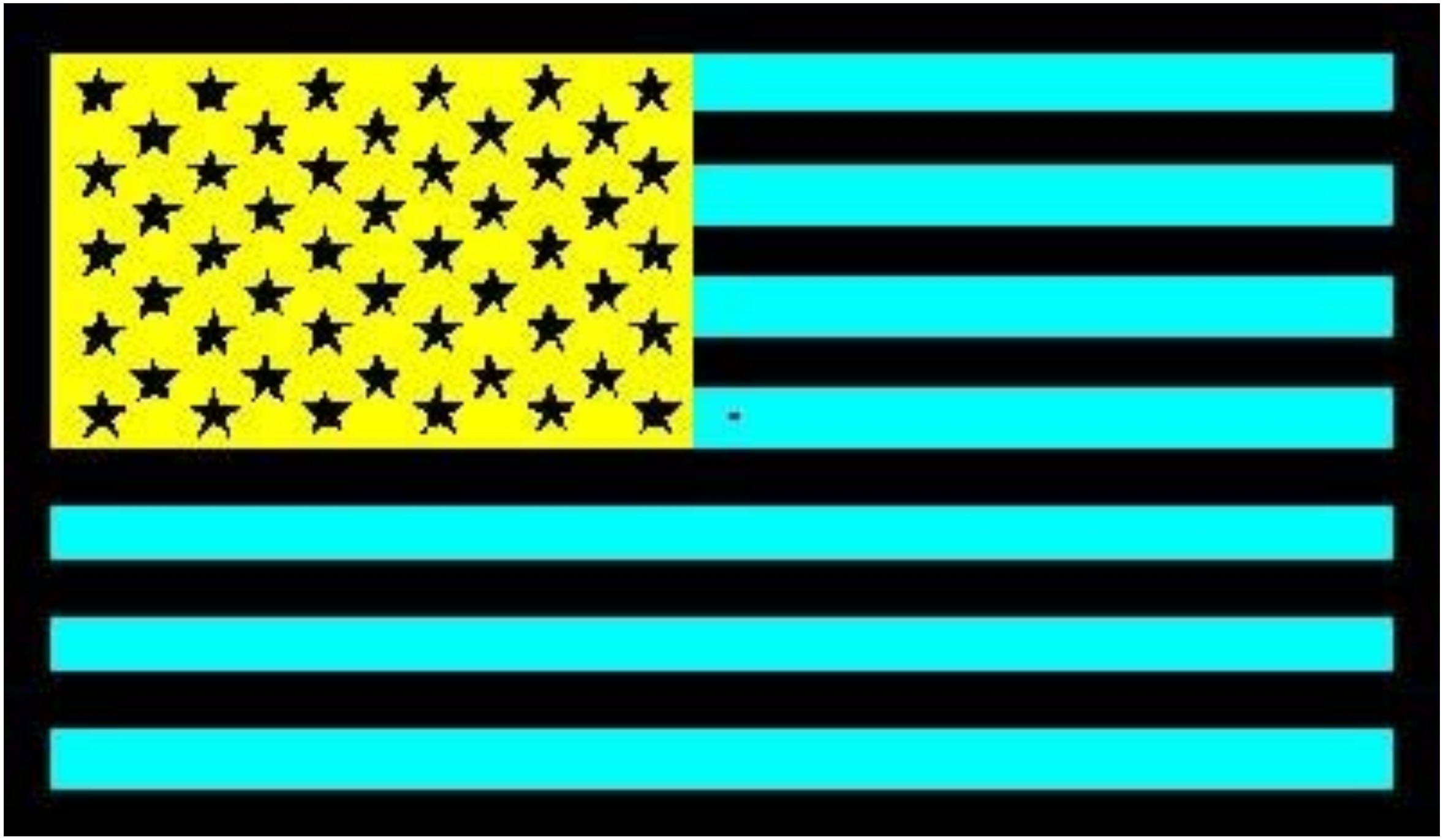
$$Y^{1/2.5} = Y^{0.4}$$

Observed display output

Desired display output

Image credit: http://creativebits.org/mac_os_x/windows_vs_mac_monitor_gamma

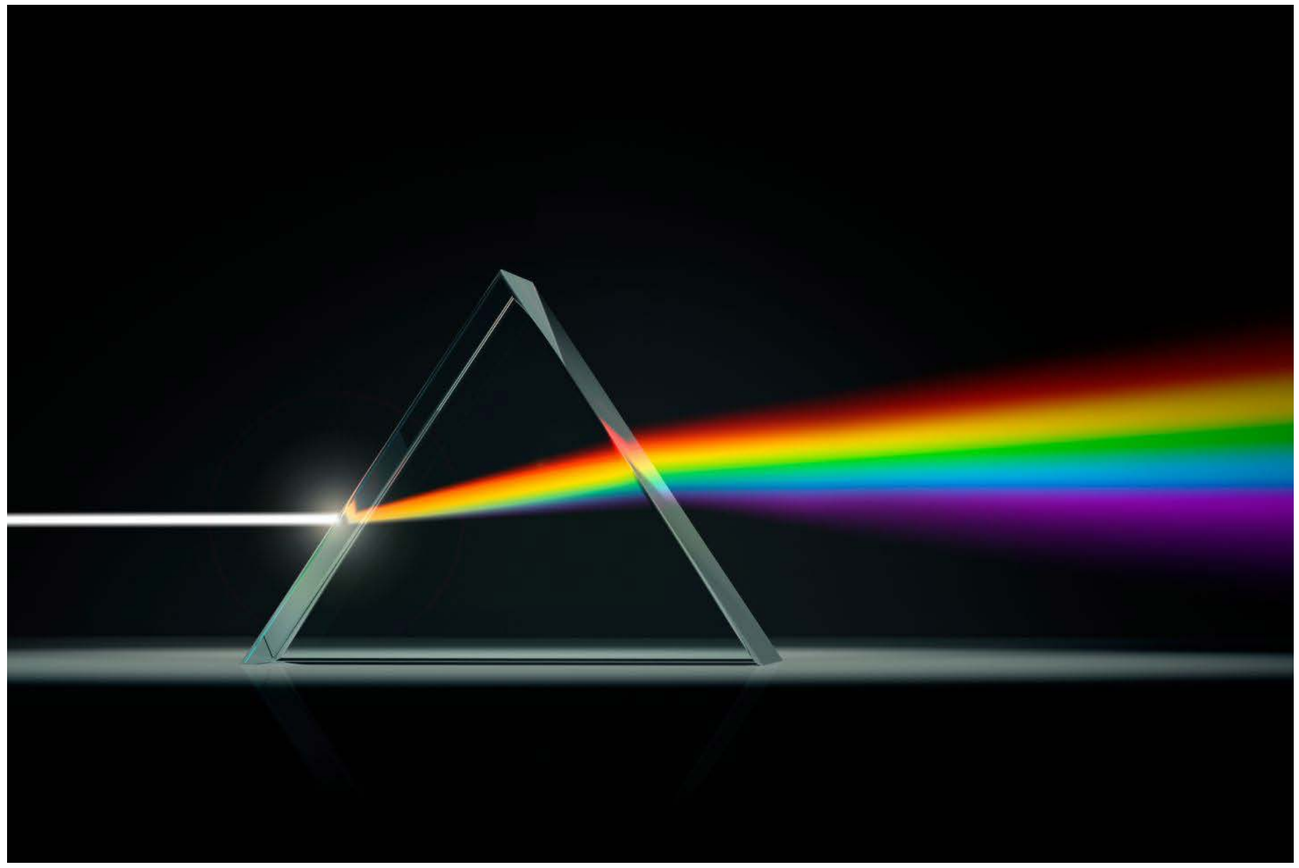
Human Perception—Acommodation Effect



Human Perception—Acommodation 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

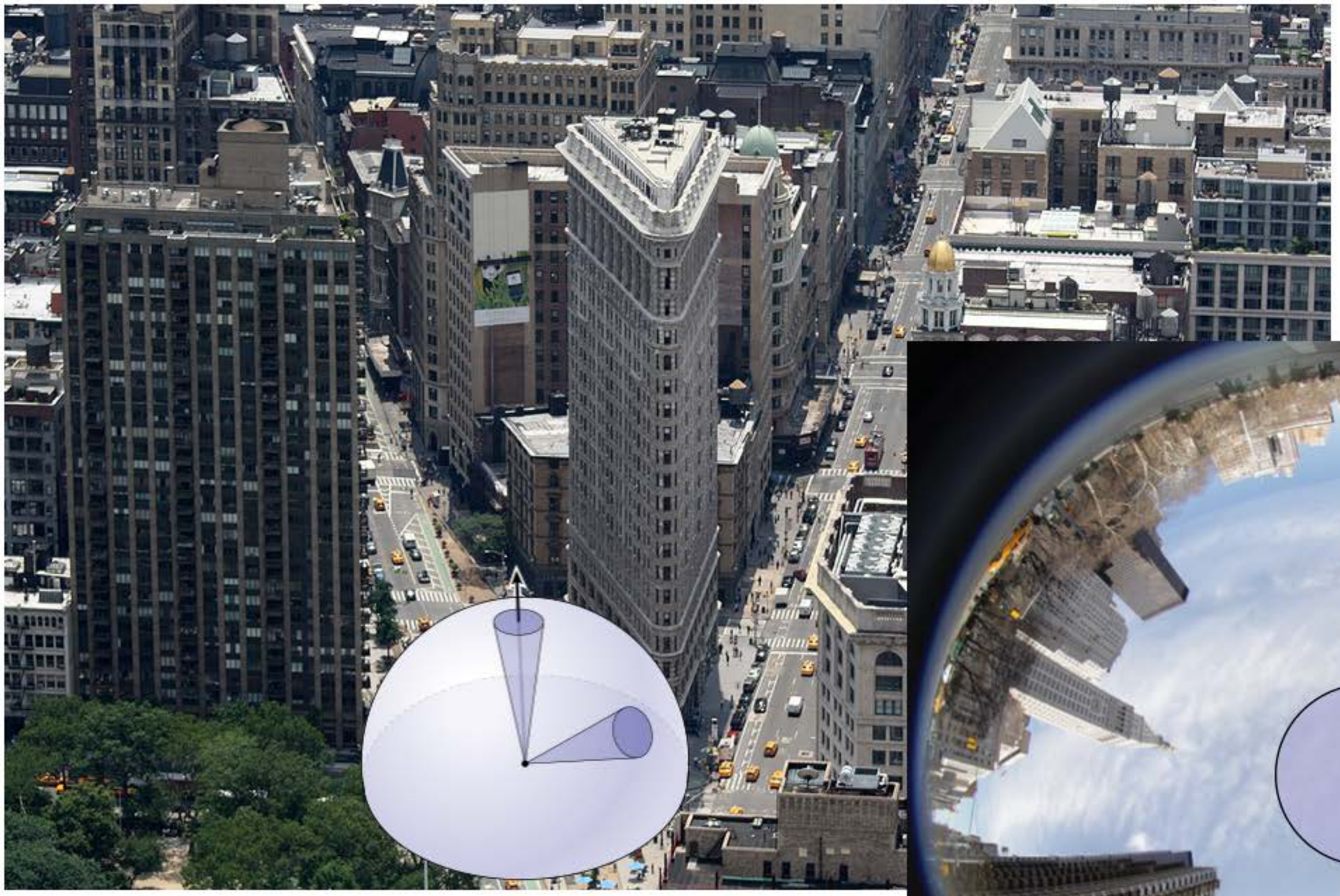
Recap: Incident vs. Exitant Radiance

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



irradiance

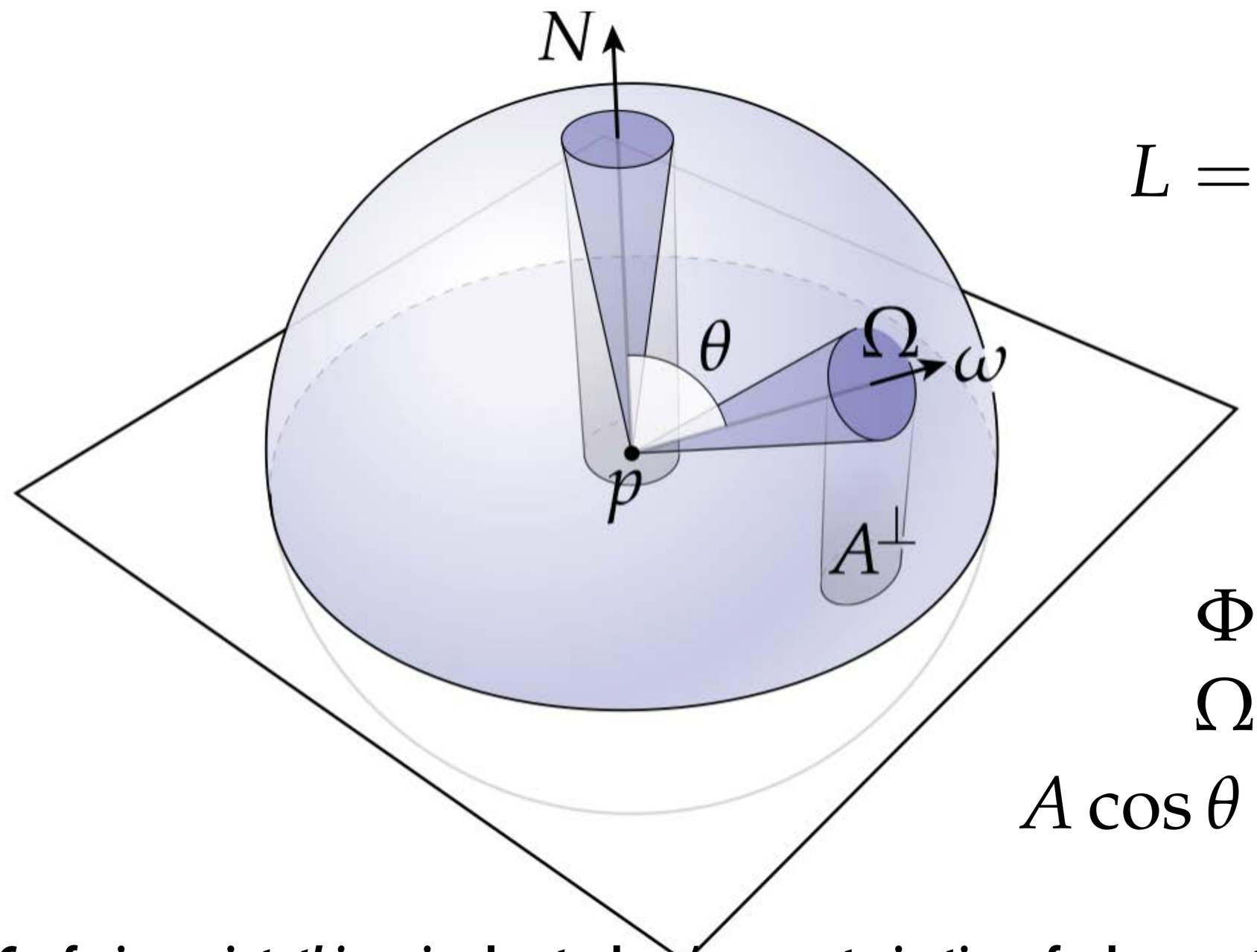
radiance in direction ω

$$E = \int_{\mathrm{H}^2} L(\omega) \cos\theta \, d\omega$$

angle between ω and normal

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.**



$$L = \frac{\partial^2 \Phi}{\partial \Omega \, \partial A \cos\theta}$$

Φ — **radiant flux**

Ω — **solid angle**

$A \cos\theta$ — **projected area***

***Confusing point: *this* cosine has to do w/ parameterization of sphere, not Lambert's law**

Aside: A Tale of Two Cosines

- **Confusing point first time you study photorealistic rendering: "cos θ" shows up for two <u>completely unrelated</u> reasons**

LAMBERT'S LAW

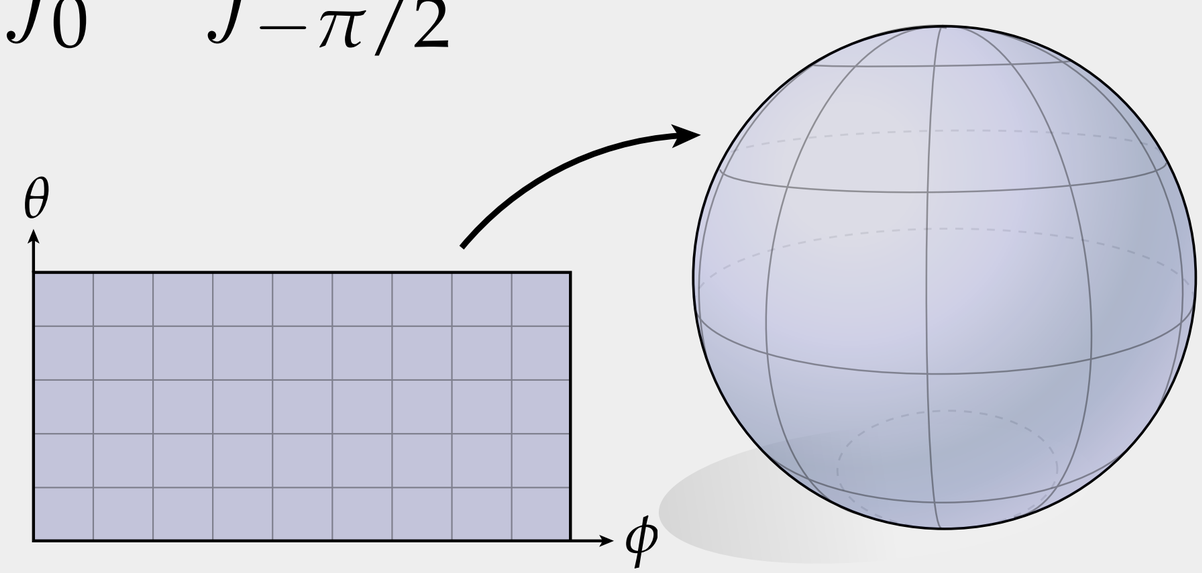
$$A = A' \cos\theta$$

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 ω_o
- Summed up by the *rendering equation* (Kajiya):

outgoing/observed radiance

emitted radiance (e.g., light source)

incoming radiance

angle between incoming direction and normal

$$L_o(\mathbf{p},\omega_o) = L_e(\mathbf{p},\omega_o) + \int_{\mathcal{H}^2} f_r(\mathbf{p},\omega_i \to \omega_o) L_i(\mathbf{p},\omega_i) \cos\theta \, d\omega_i$$

point of interest

direction of interest

all directions in hemisphere

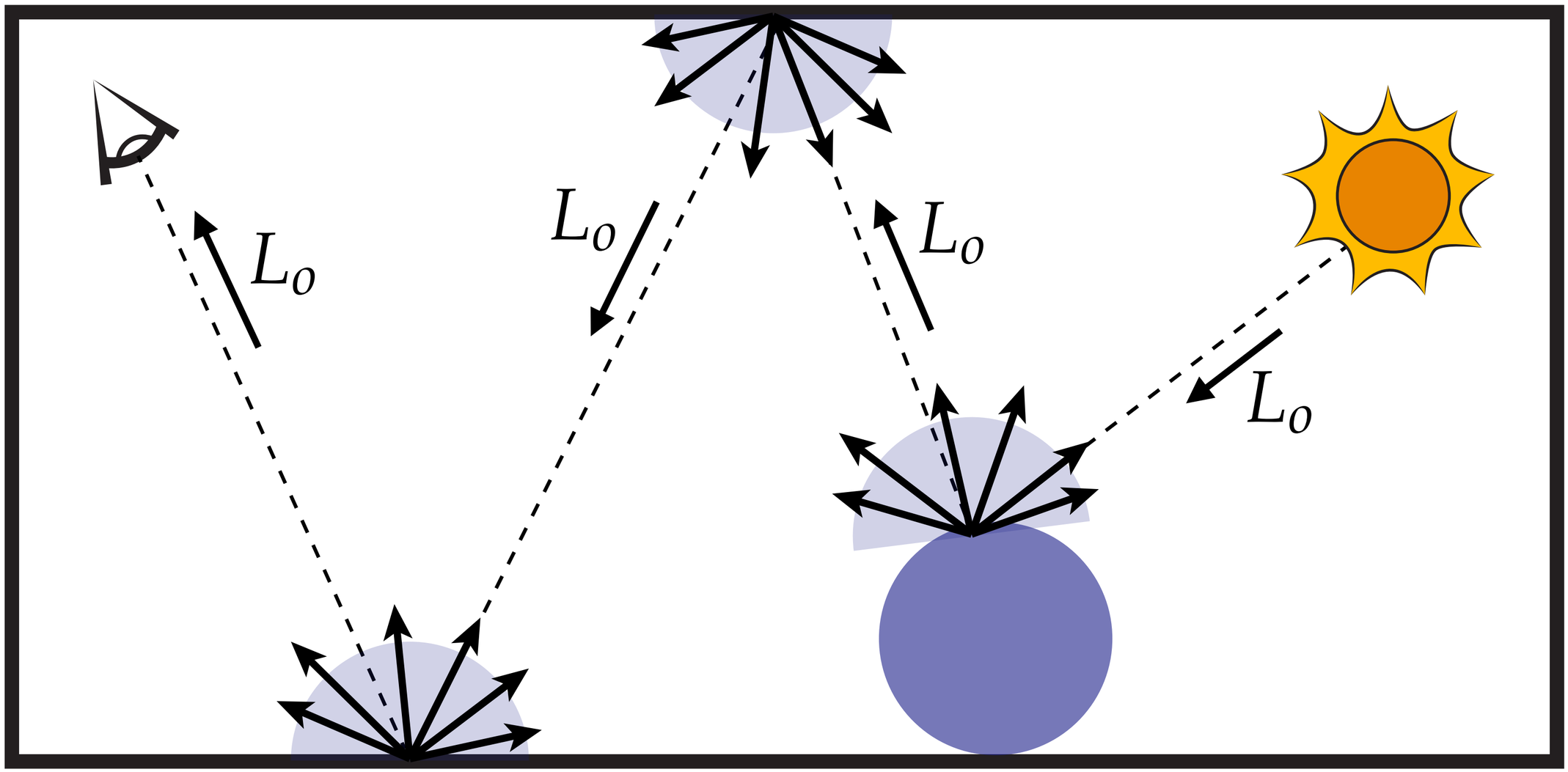
scattering function

incoming direction

Key challenge: to evaluate incoming radiance, we have to compute yet another integral. I.e., rendering equation is *recursive*.

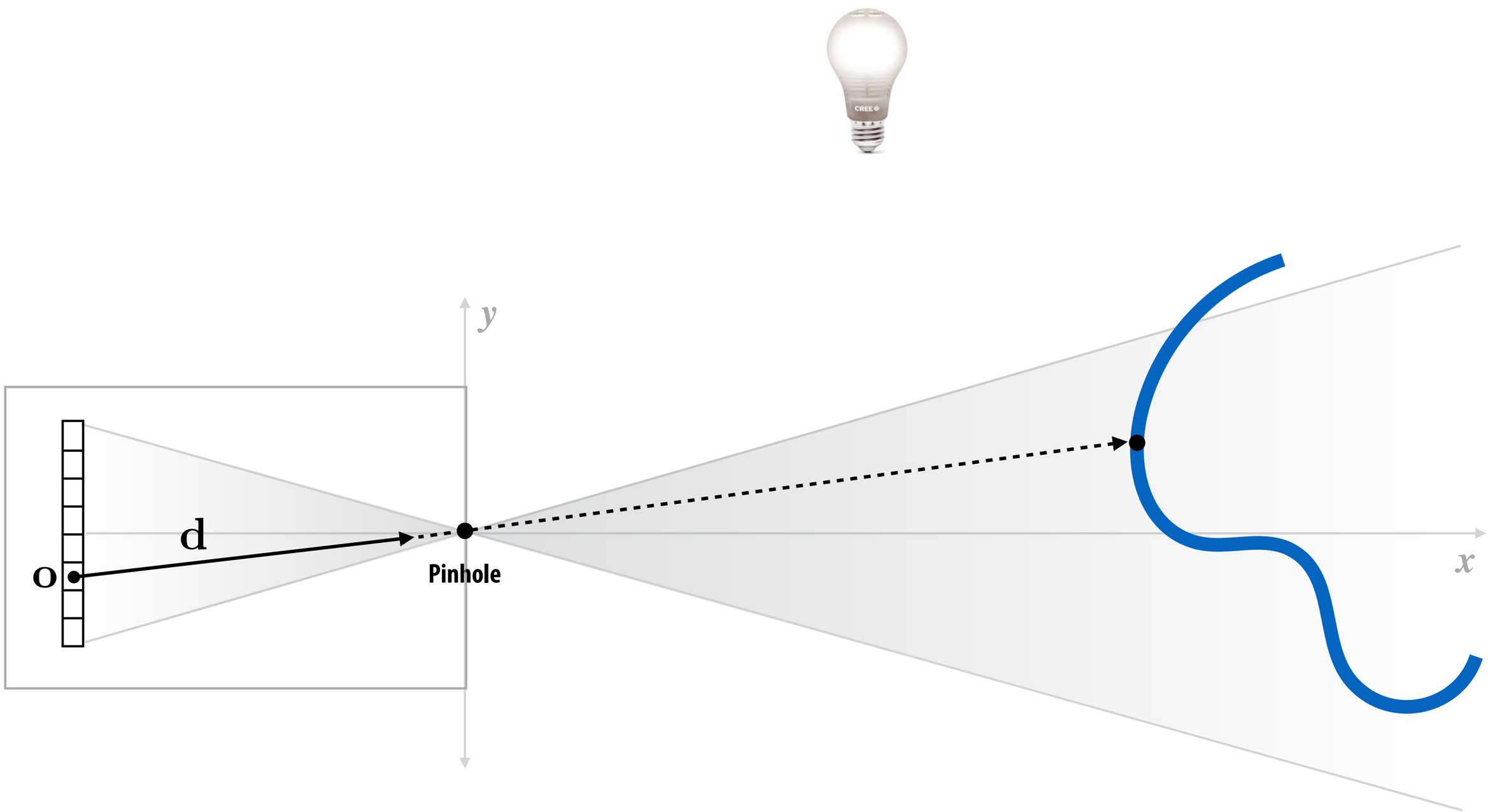
Recursive Raytracing

- **Basic strategy: recursively evaluate rendering equation!**



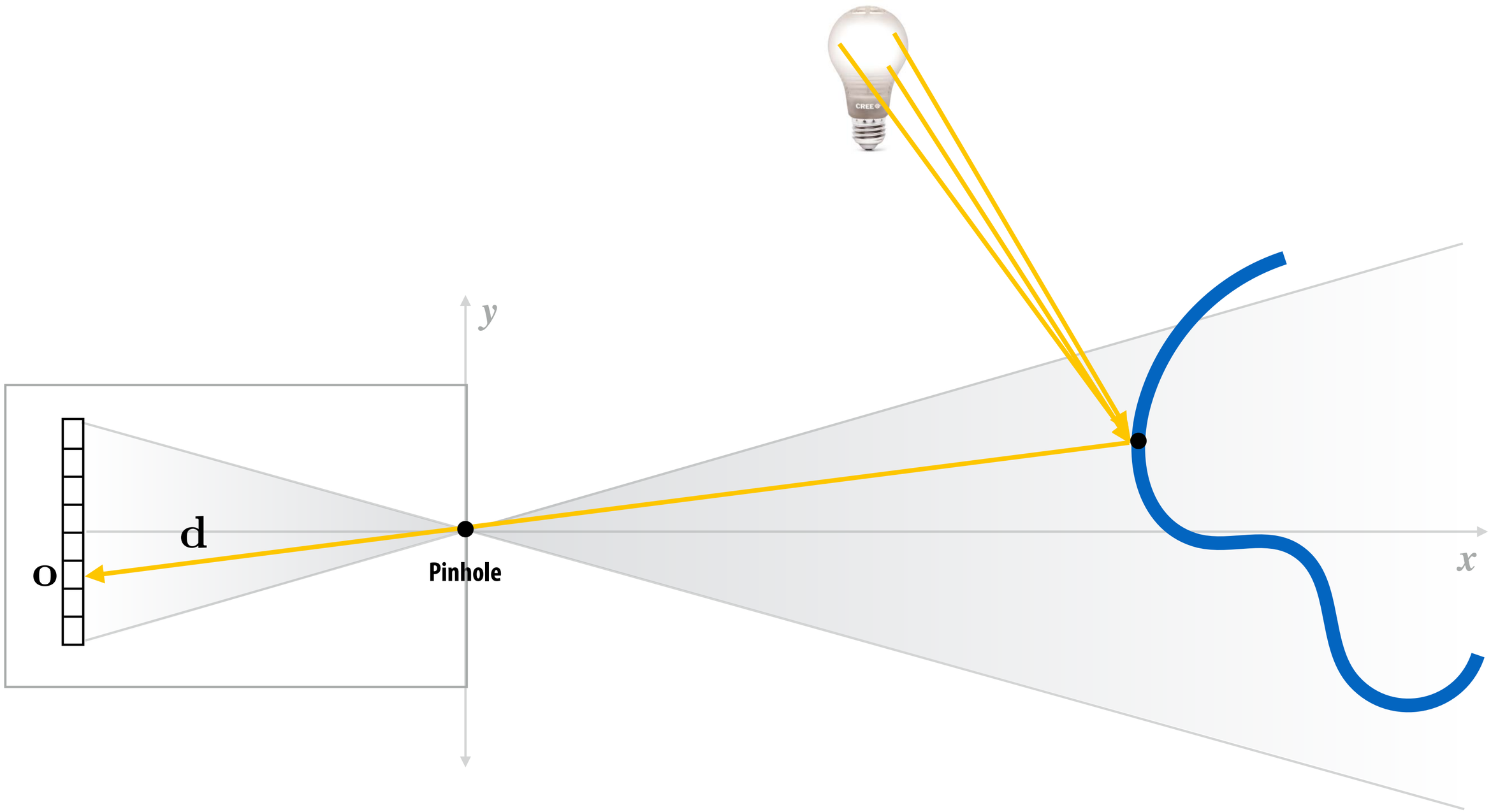
(*This* is why you're writing a ray tracer—rasterizer isn't enough!)

Renderer measures radiance along a ray

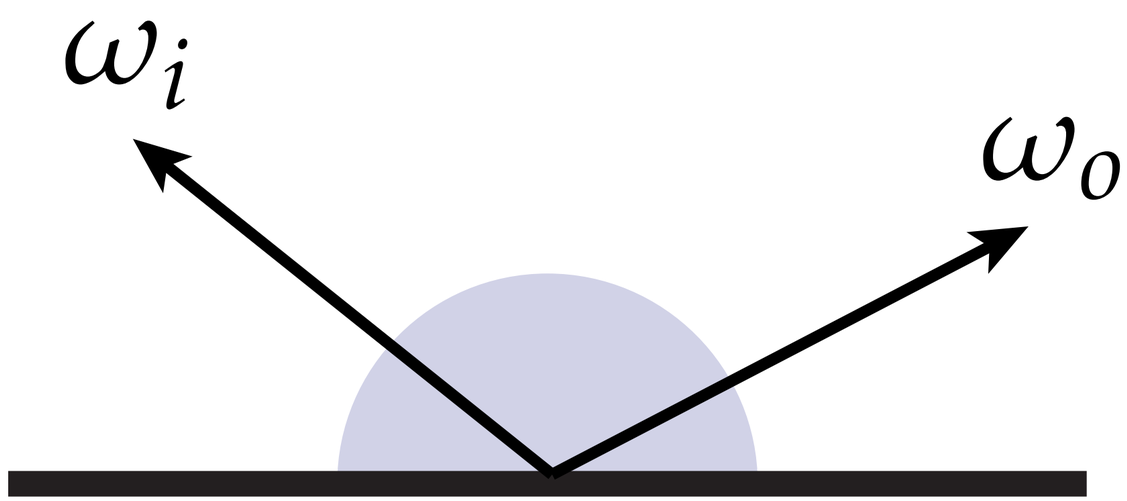


At each "bounce," want to measure radiance traveling in the direction opposite the ray direction.

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?

$$L_o(\mathbf{p}, \omega_o) = \int_{\mathcal{H}^2} f_r(\mathbf{p}, \omega_i, \omega_o) L_i(\mathbf{p}, \omega_i) \cos\theta \, \mathrm{d}\,\omega_i$$

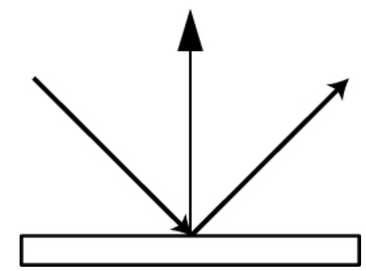
Reflection models

- *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

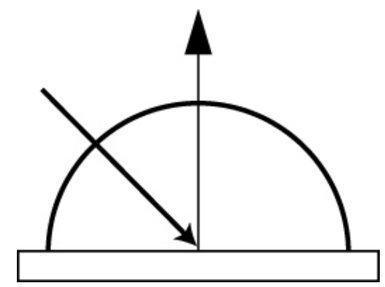
- **Ideal specular**

  Perfect mirror



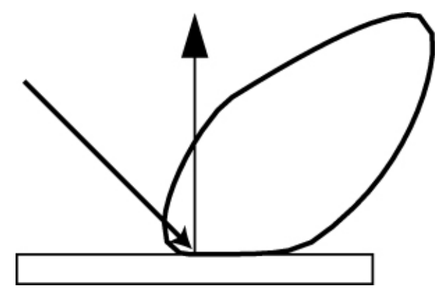
- **Ideal diffuse**

  Uniform reflection in all directions



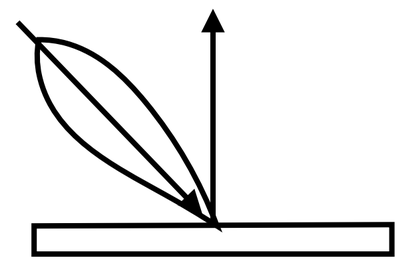
- **Glossy specular**

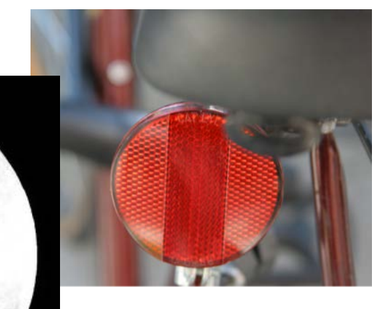
  Majority of light distributed in reflection direction

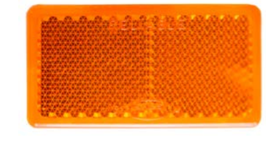


- **Retro-reflective**

  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

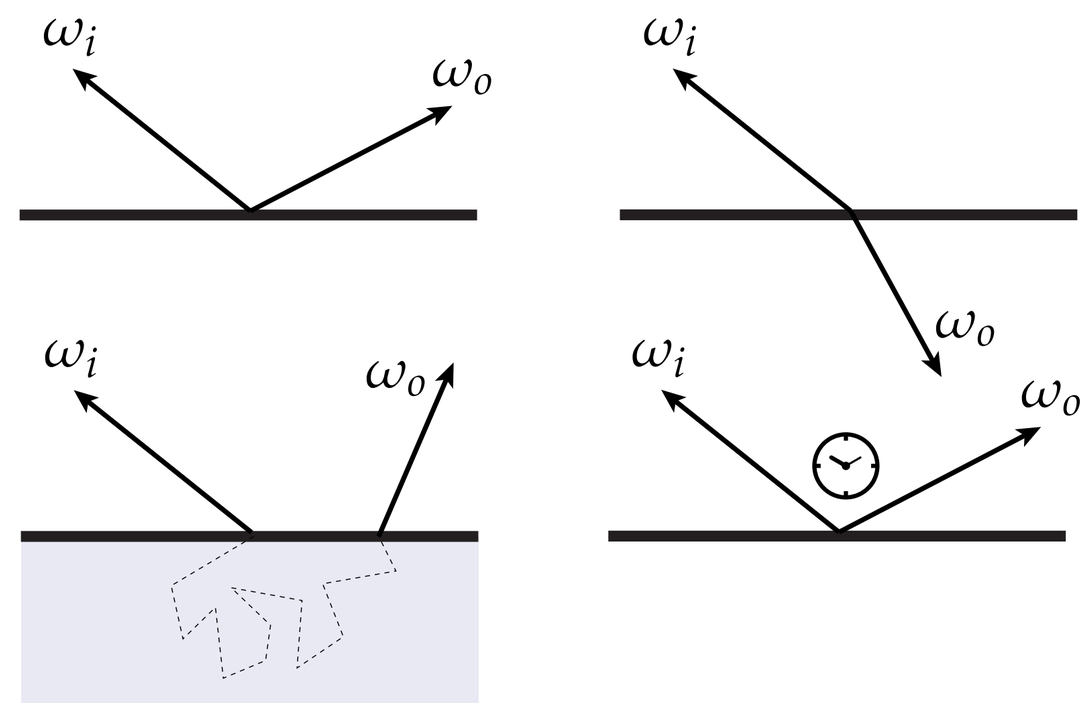
Materials: mirror

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
  - …

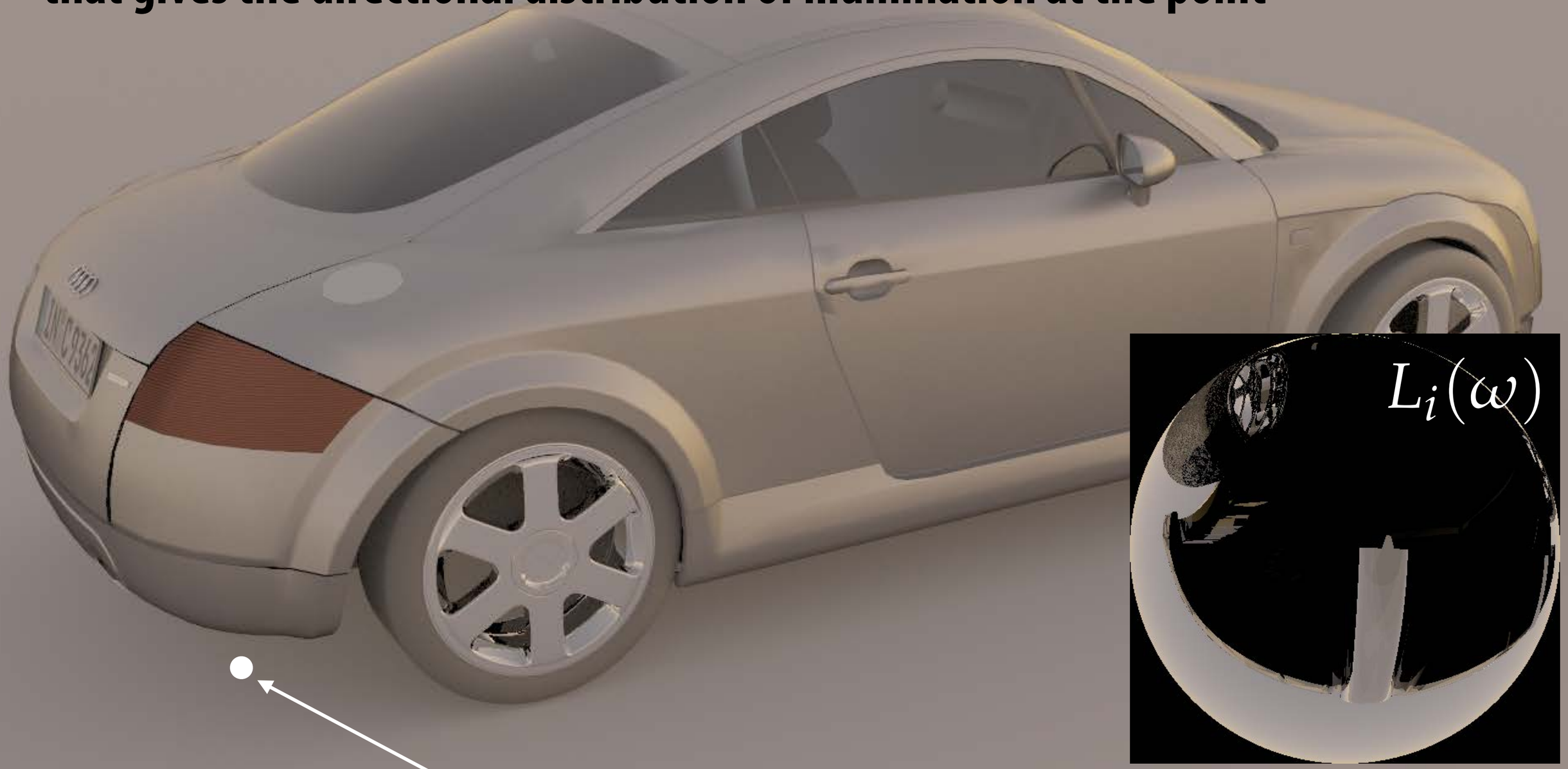


- What goes in must come out! (Total energy must be conserved)
- In general, can talk about *"probability*"* a particle arriving from a given direction is scattered in another direction

*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

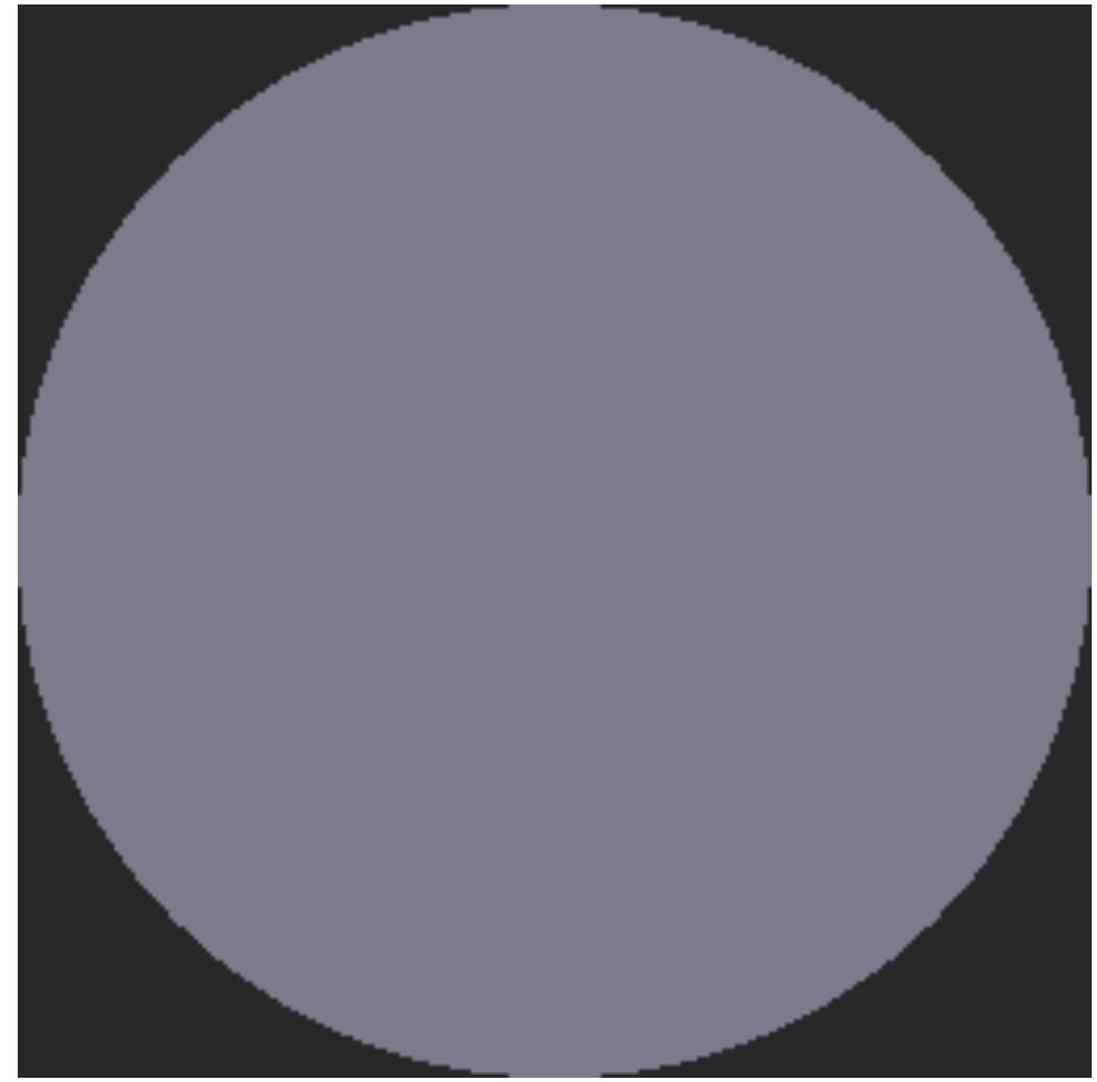


Consider view of hemisphere from this point

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"

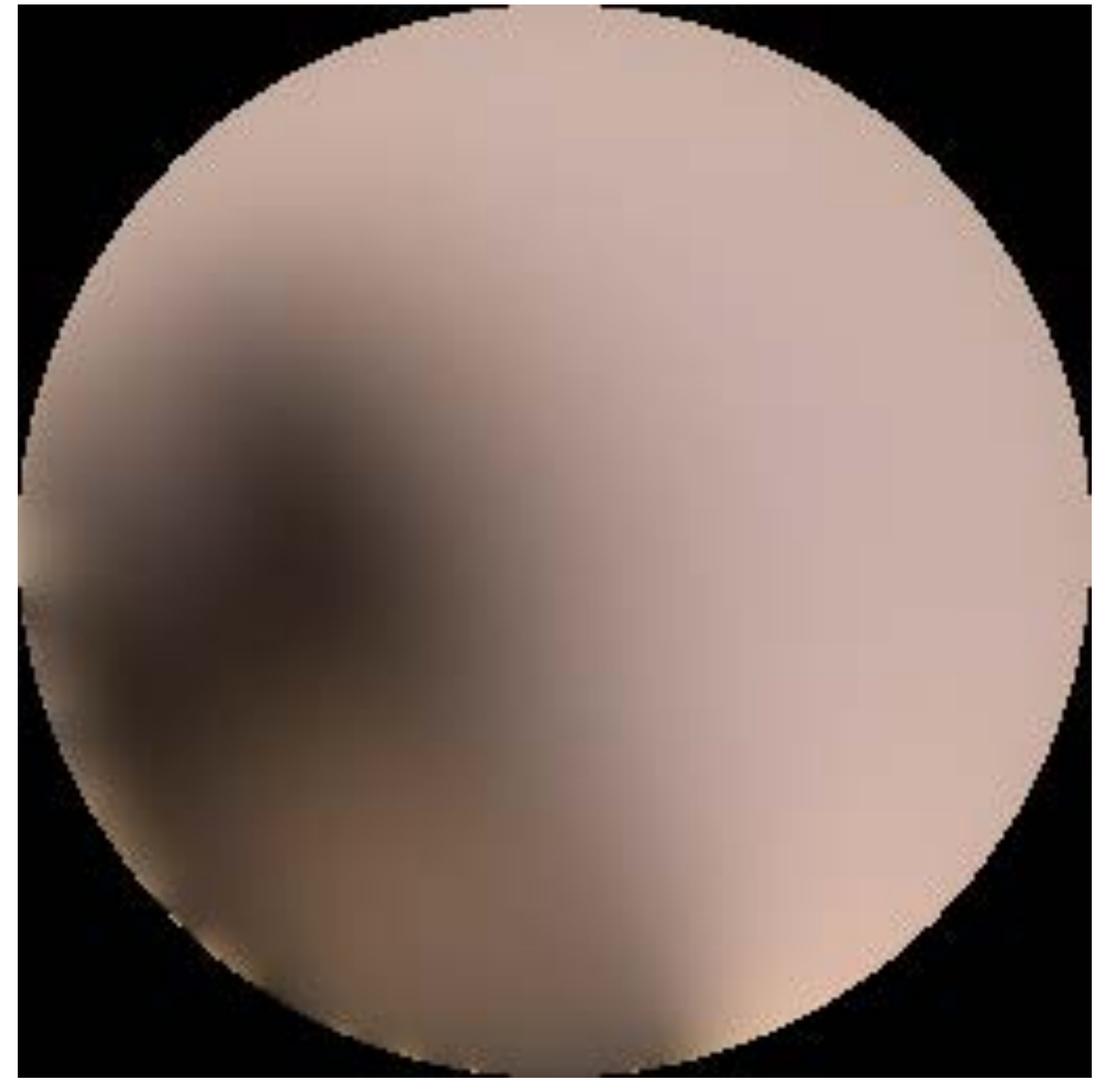
Incident radiance

Exitant radiance

Copper

More blurring, plus coloration (nonuniform absorption across frequencies)

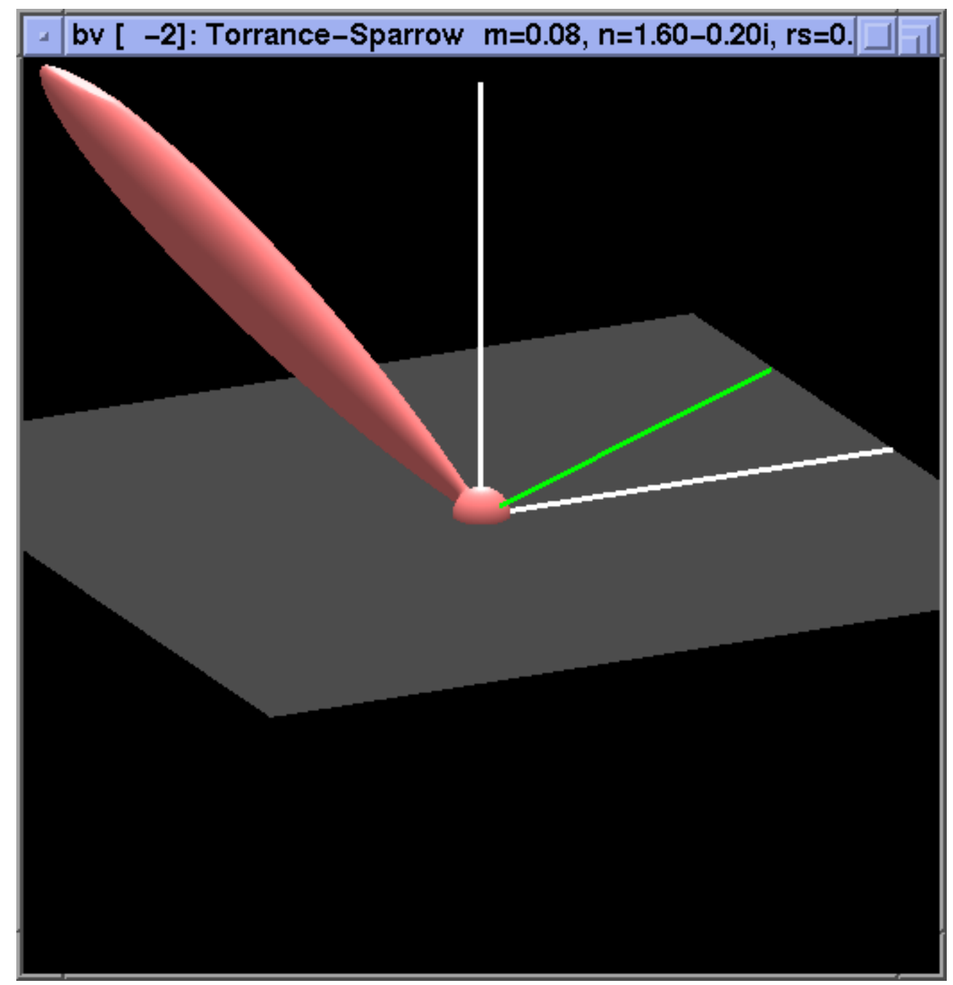
Incident radiance



Exitant radiance

Scattering off a surface: the BRDF

- "Bidirectional reflectance distribution function"
- Encodes behavior of light that "bounces off" surface
- Given incoming direction ω_i, how much light gets scattered in any given outgoing direction ω_o?
- Describe as distribution $f_r(\omega_i \rightarrow \omega_o)$



bv (Szymon Rusinkiewicz)

$$f_r(\omega_i \rightarrow \omega_o) \geq 0$$

why *less* than or equal?

$$\int_{\mathcal{H}^2} f_r(\omega_i \rightarrow \omega_o) \cos\theta \, d\omega_i \leq 1$$

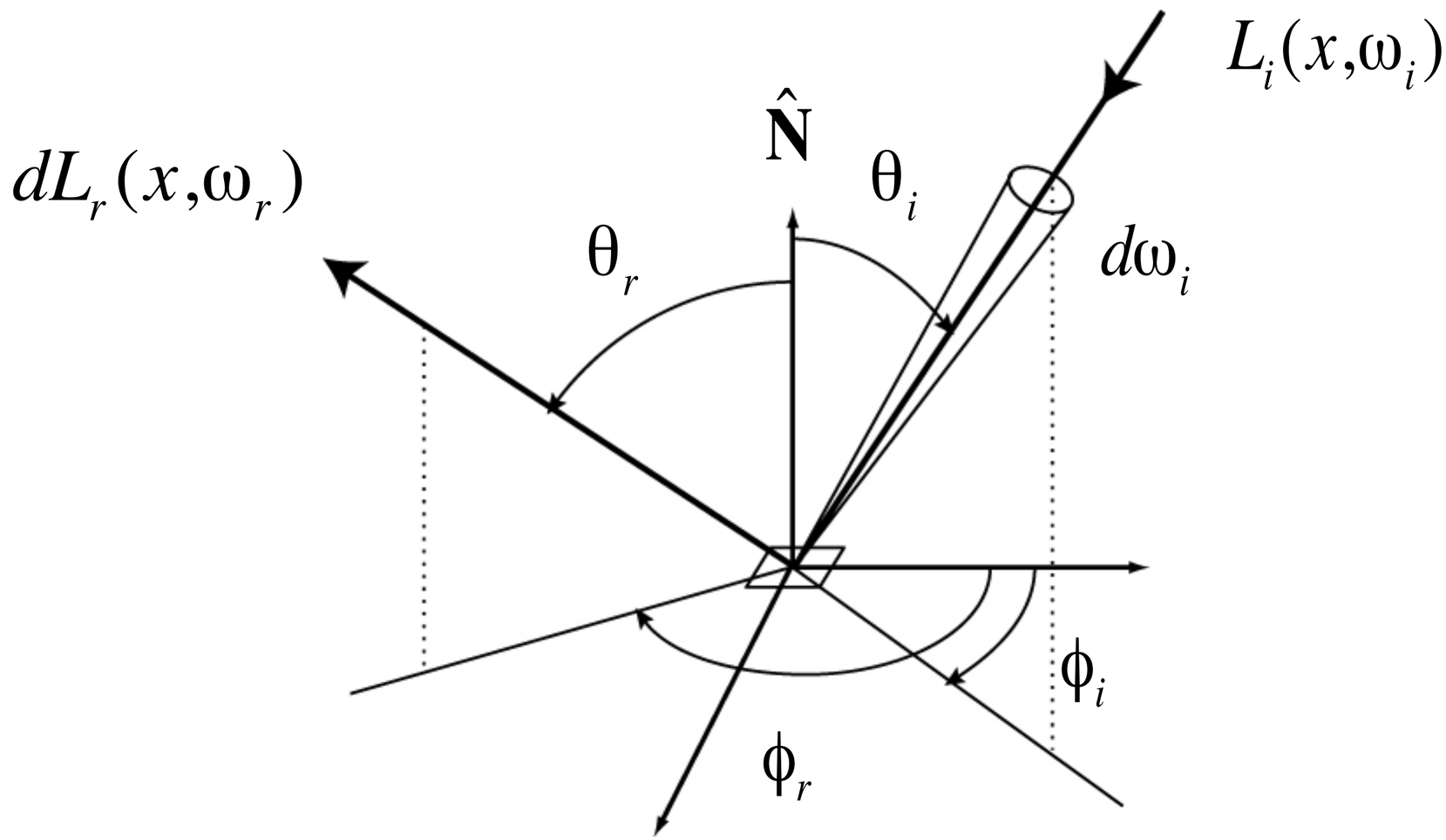
where did the rest of the energy go?!

$$f_r(\omega_i \rightarrow \omega_o) = f_r(\omega_o \rightarrow \omega_i)$$

"Helmholtz reciprocity"

Q: Why should Helmholtz reciprocity hold? Think about little mirrors…

Radiometric description of BRDF

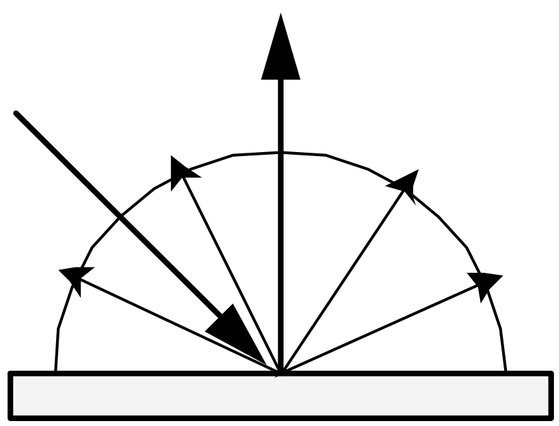


$$f_r(\omega_i \rightarrow \omega_o) = \frac{dL_o(\omega_o)}{dE_i(\omega_i)} = \frac{dL_o(\omega_o)}{dL_i(\omega_i)\cos\theta_i}\left[\frac{1}{sr}\right]$$

"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



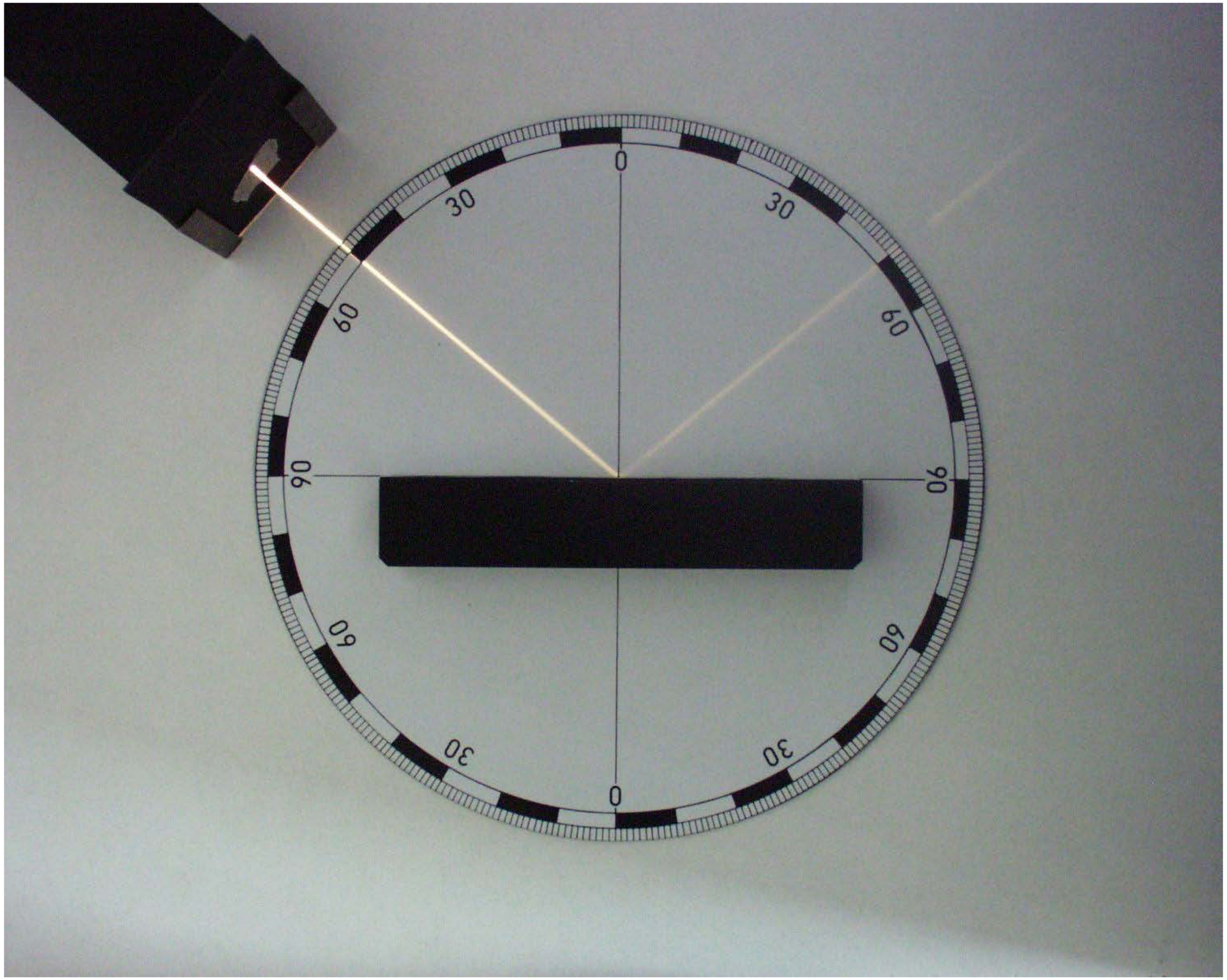
$$f_r = c$$

$$\begin{aligned} L_o(\omega_o) &= \int_{H^2} f_r \, L_i(\omega_i) \, \cos\theta_i \, \mathrm{d}\omega_i \\ &= f_r \int_{H^2} L_i(\omega_i) \, \cos\theta_i \, \mathrm{d}\omega_i \\ &= f_r E \end{aligned}$$

"albedo" (between 0 and 1)

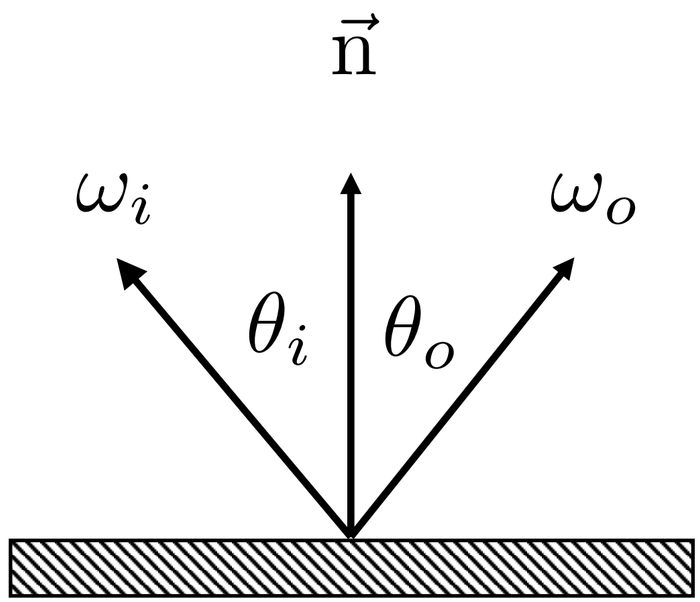
$$f_r = \frac{\rho}{\pi}$$

Example: perfect specular reflection



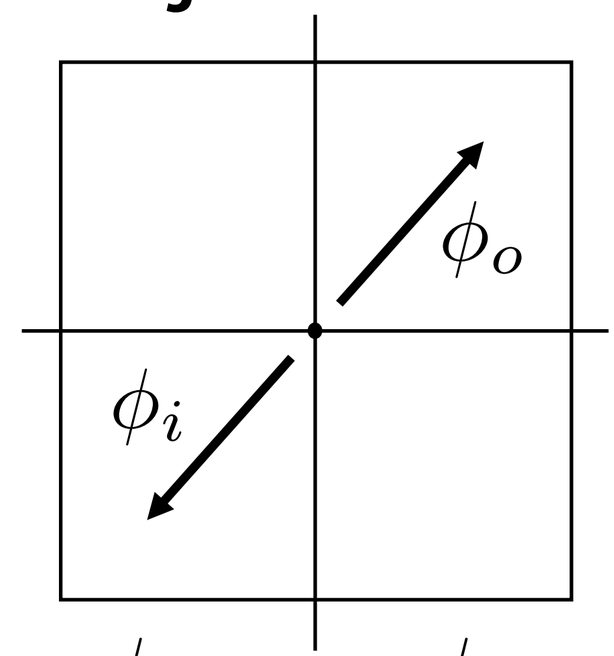
[Zátonyi Sándor]

Geometry of specular reflection



$$\theta = \theta_o = \theta_i$$

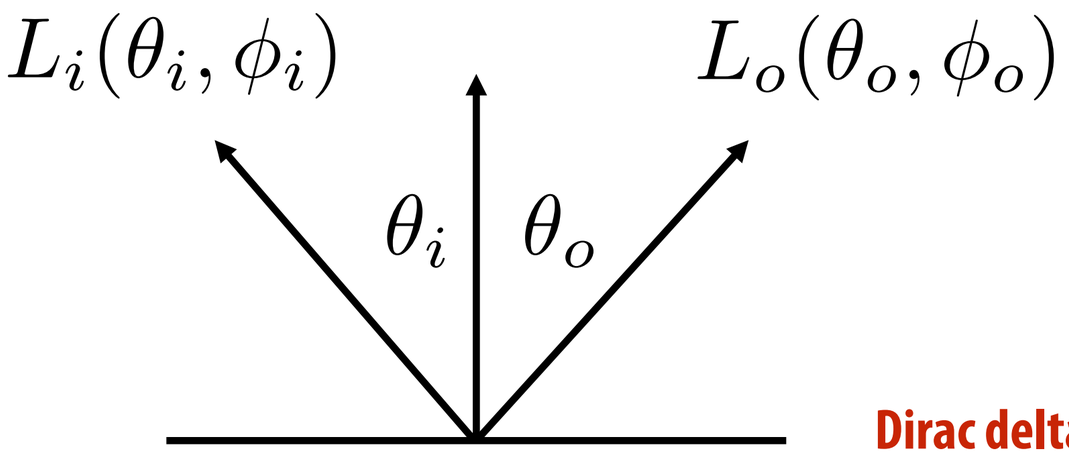
Top-down view
(looking down on surface)



$$\phi_o = -\phi_i$$

$$\omega_o = -\omega_i + 2(\omega_i \cdot \vec{n})\vec{n}$$

Specular reflection BRDF



$$L_o(\theta_o, \phi_o) = L_i(\theta_i, \phi_i)$$

Dirac delta

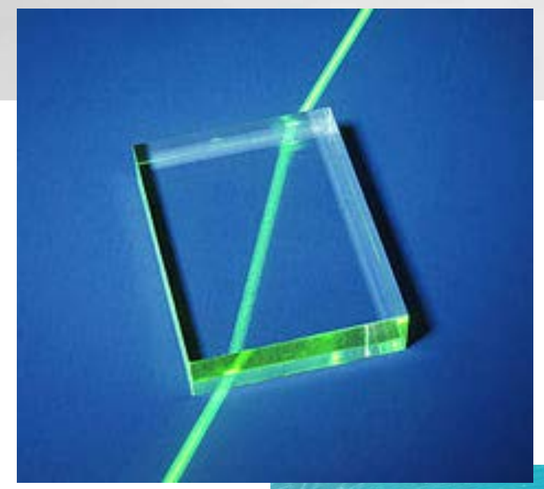
$$f_r(\theta_i, \phi_i; \theta_o, \phi_o) = \frac{\delta(\cos\theta_i - \cos\theta_o)}{\cos\theta_i}\delta(\phi_i - \phi_o \pm \pi)$$

- **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

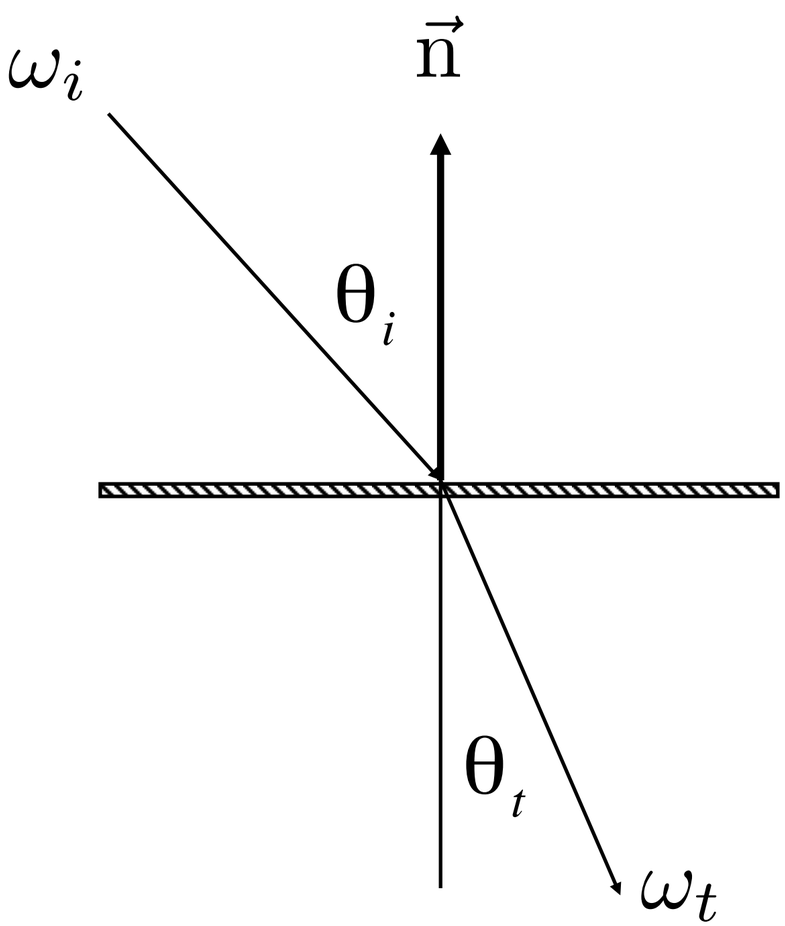
In addition to reflecting off surface, light may be transmitted through surface.

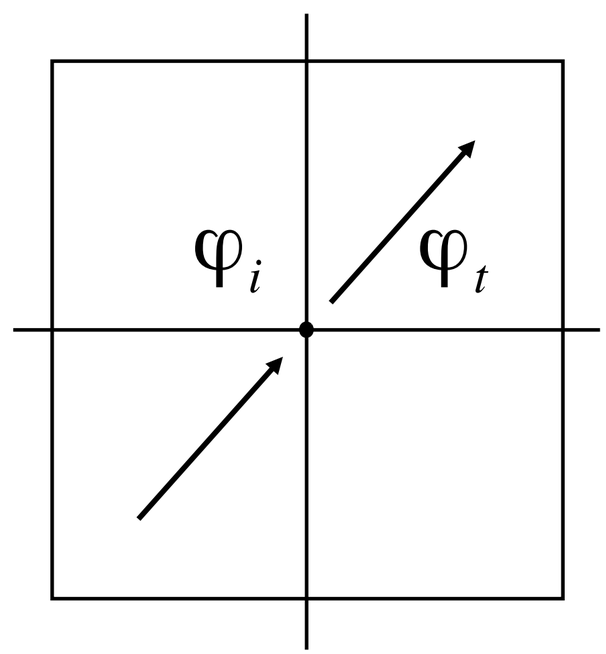
Light refracts when it enters a new medium.



Snell's Law

Transmitted angle depends on relative index of refraction of material ray is leaving/entering.



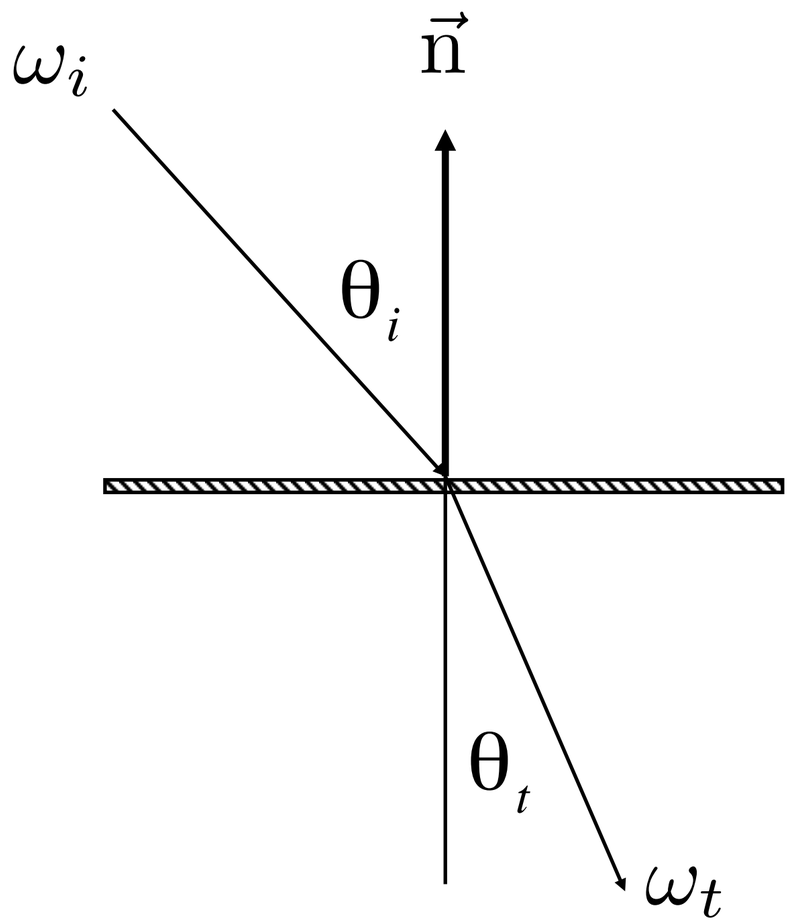


Medium	η^*
Vacuum	1.0
Air (sea level)	1.00029
Water (20°C)	1.333
Glass	1.5-1.6
Diamond	2.42

* index of refraction is wavelength dependent (these are averages)

$$\eta_i \sin\theta_i = \eta_t \sin\theta_t$$

Law of refraction



$$\eta_i \sin\theta_i = \eta_t \sin\theta_t$$

$$\cos\theta_t = \sqrt{1 - \sin^2\theta_t}$$

$$= \sqrt{1 - \left(\frac{\eta_i}{\eta_t}\right)^2 \sin^2\theta_i}$$

$$= \sqrt{1 - \left(\frac{\eta_i}{\eta_t}\right)^2 (1 - \cos^2\ \theta_i)}$$

Total internal reflection:

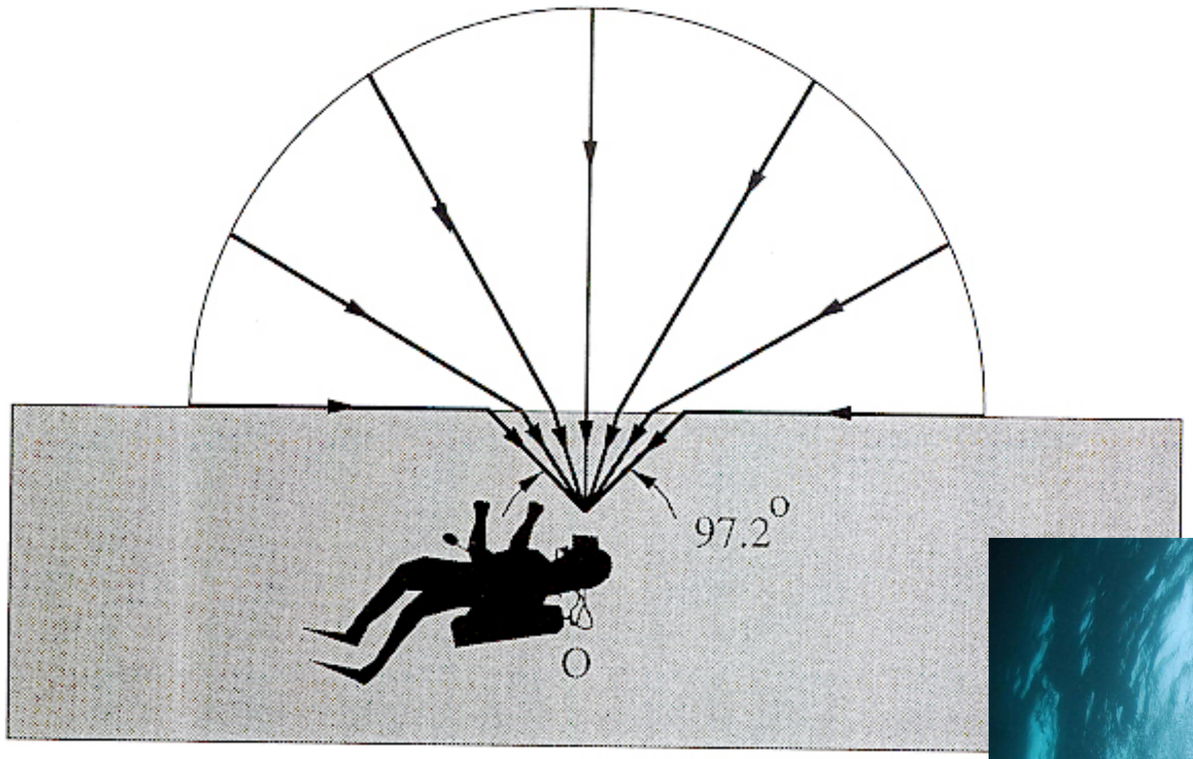
$$1 - \left(\frac{\eta_i}{\eta_t}\right)^2 (1 - \cos^2\theta_i) < 0$$

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.

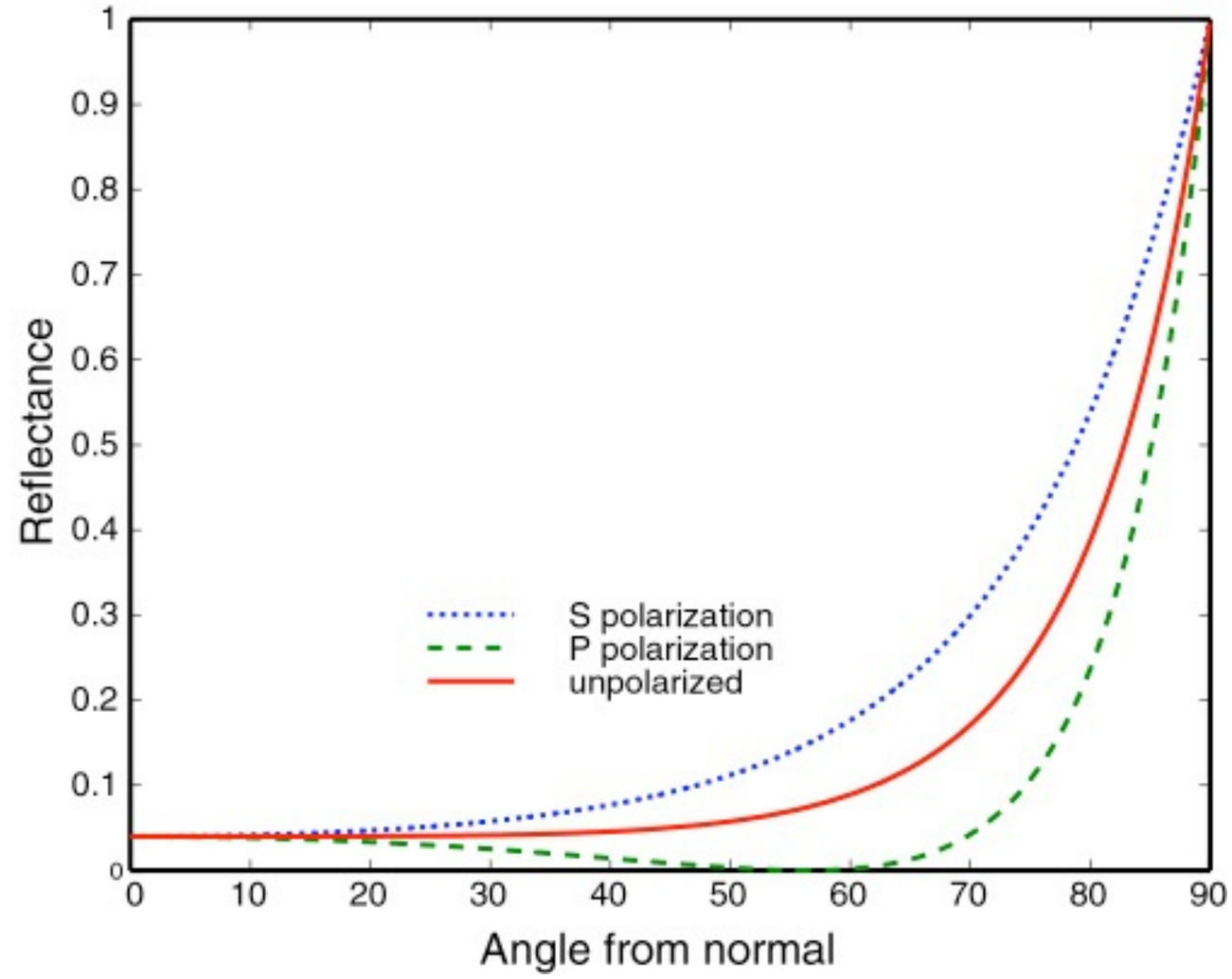
Optical manhole

Only small “cone” visible, due to total internal reflection (TIR)



Fresnel reflection

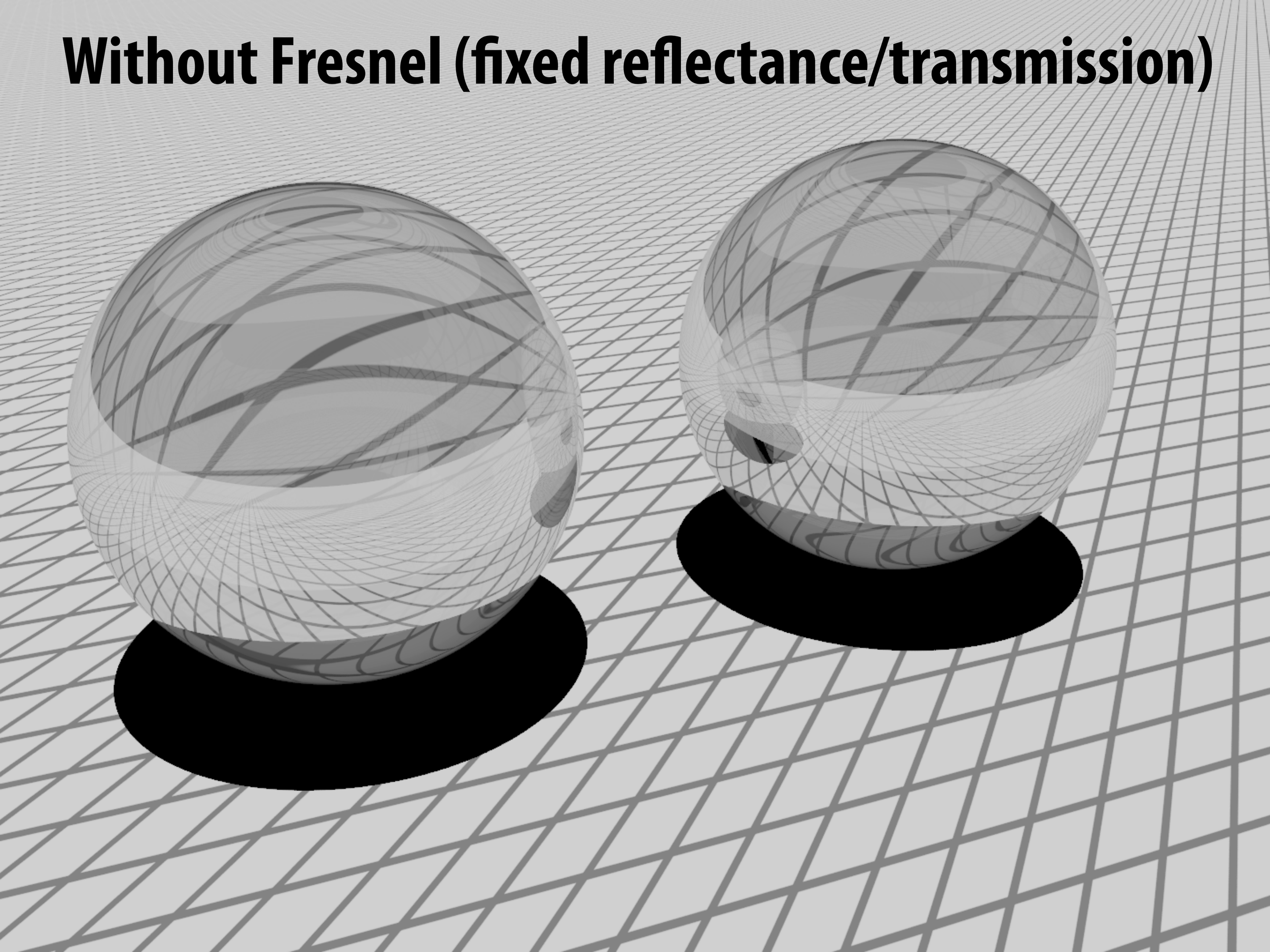
Many real materials: reflectance increases w/ viewing angle



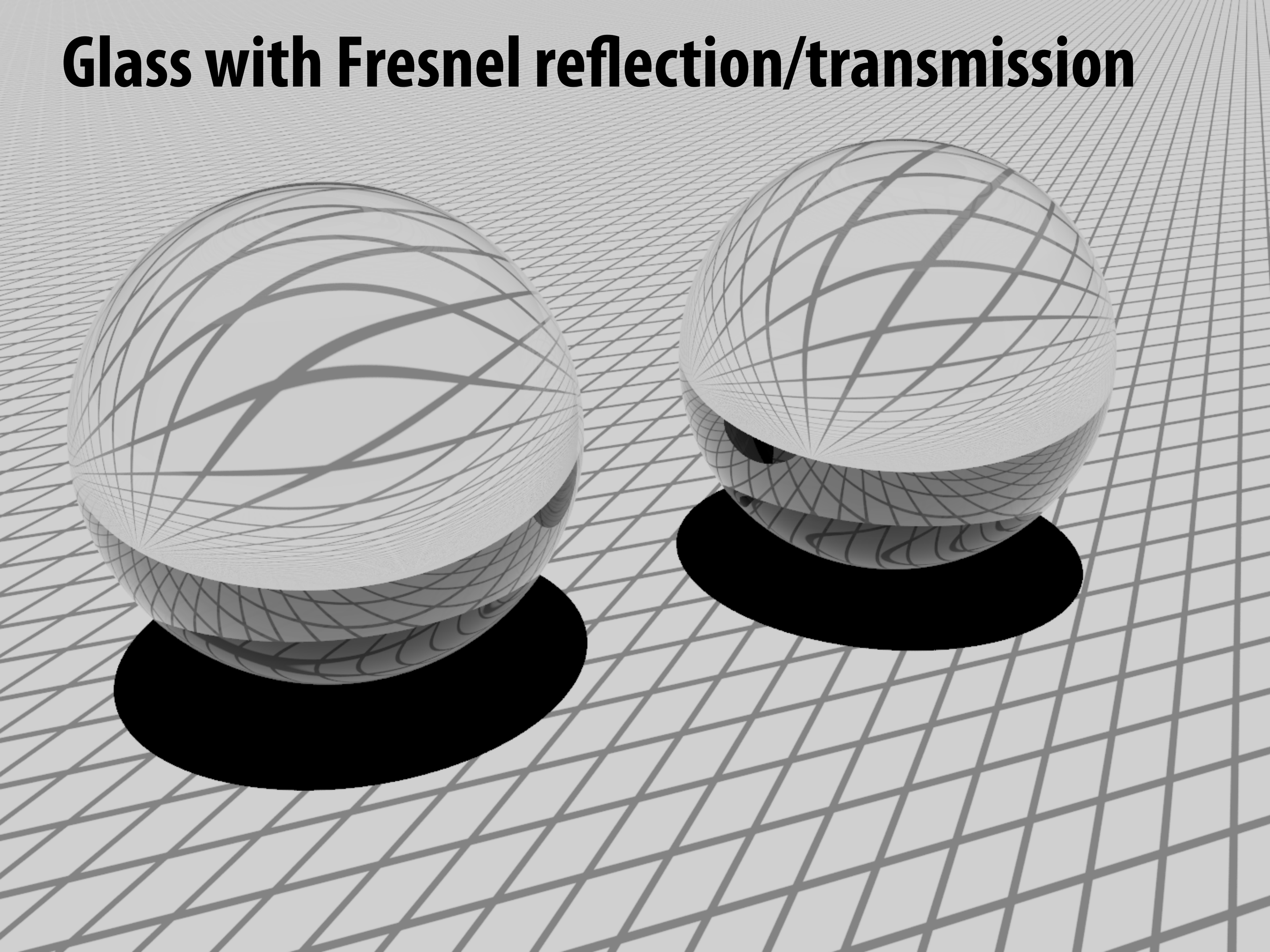
[Lafortune et al. 1997]

Snell + Fresnel: Example

Without Fresnel (fixed reflectance/transmission)



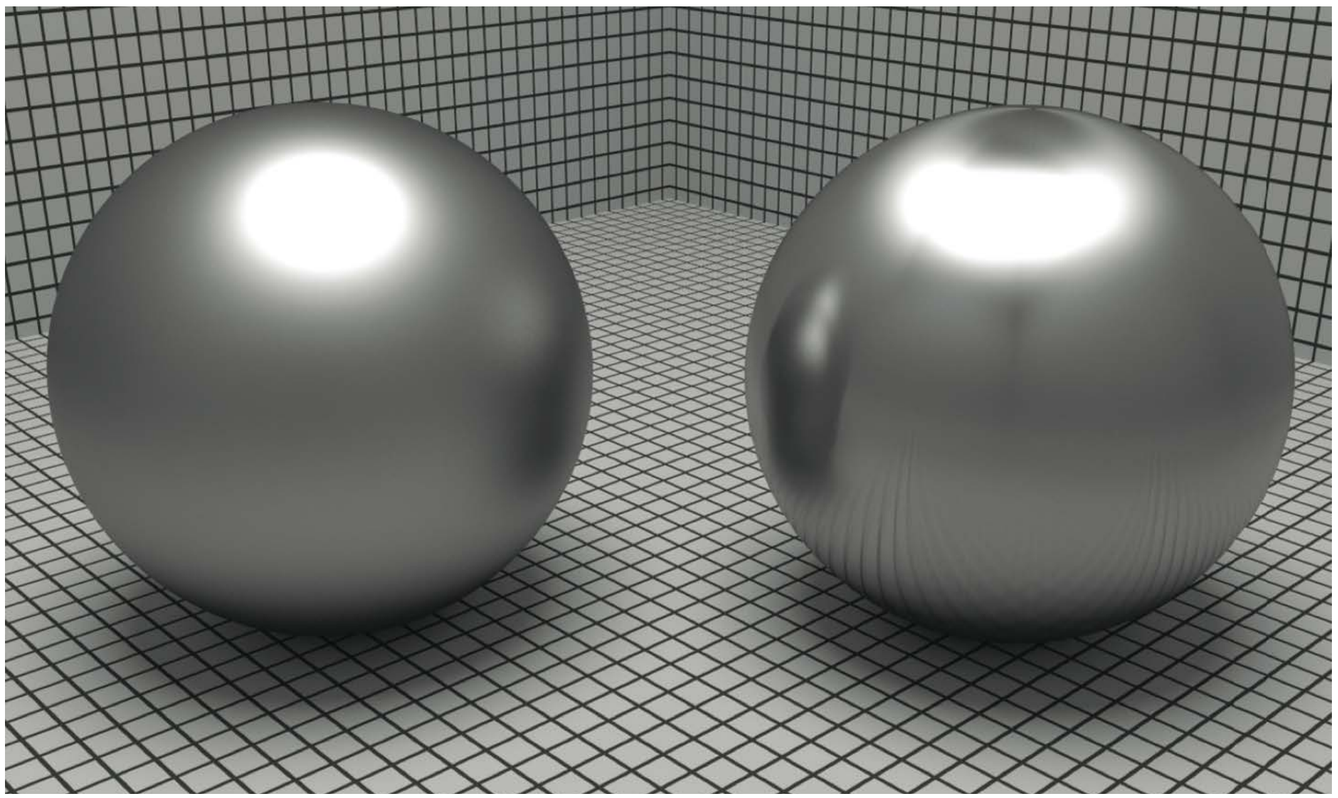
Glass with Fresnel reflection/transmission



Anisotropic reflection

Reflection depends on azimuthal angle ϕ

Results from oriented microstructure of surface e.g., brushed metal



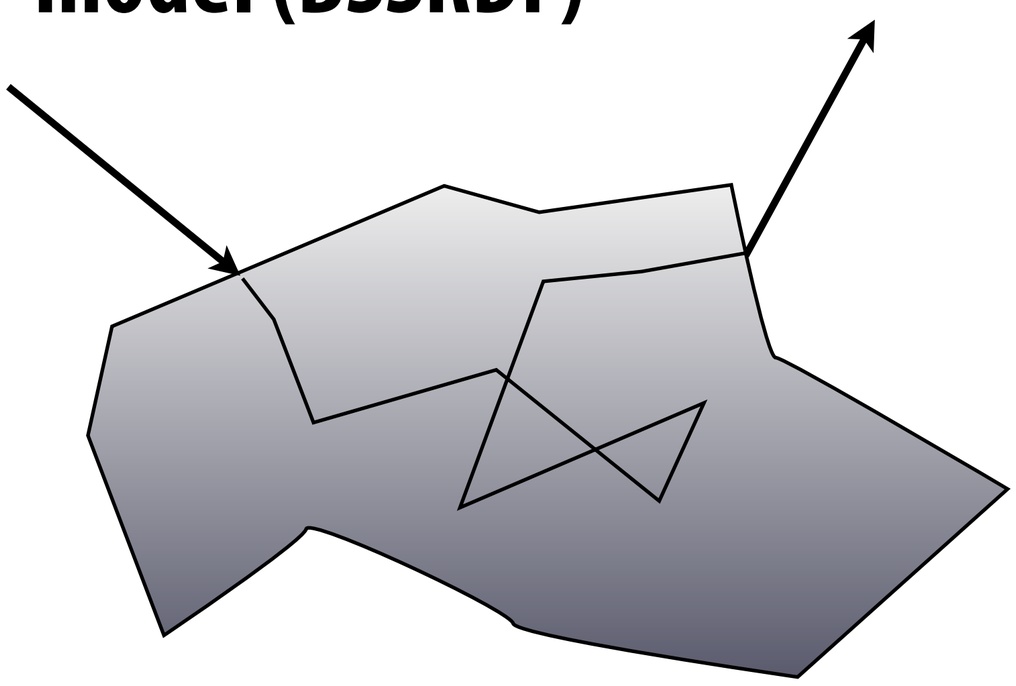
Translucent materials: Jade

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]

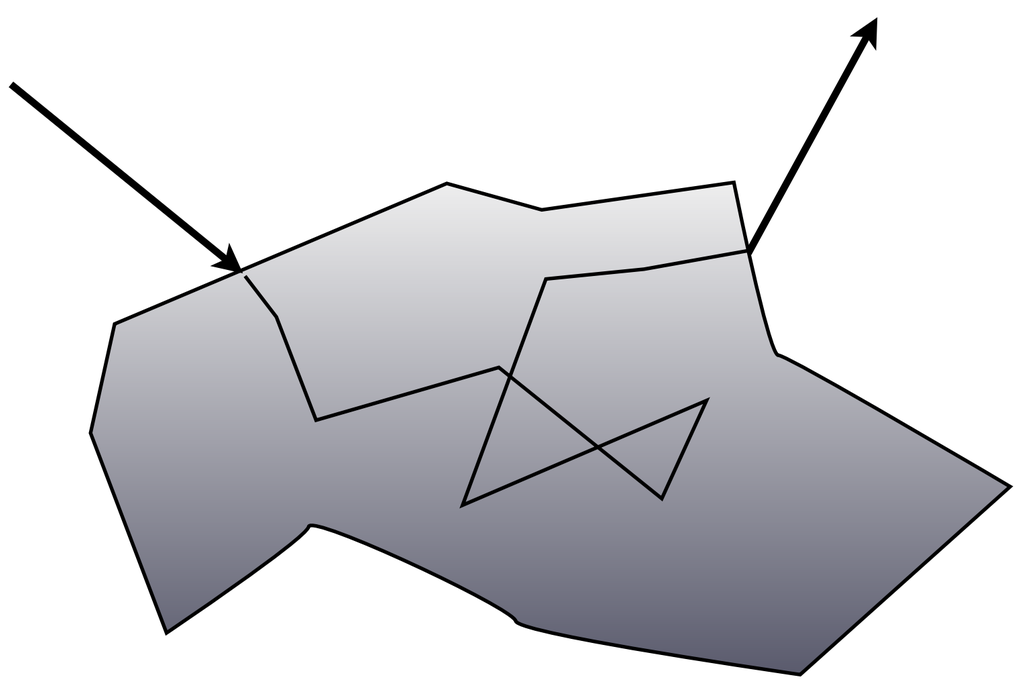
Scattering functions

- **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(!)**

$$L(x_o, \omega_o) = \int_A \int_{H^2} S(x_i, \omega_i, x_o, \omega_o)\, L_i(x_i, \omega_i) \cos\theta_i \, \mathrm{d}\omega_i \, \mathrm{d}A$$



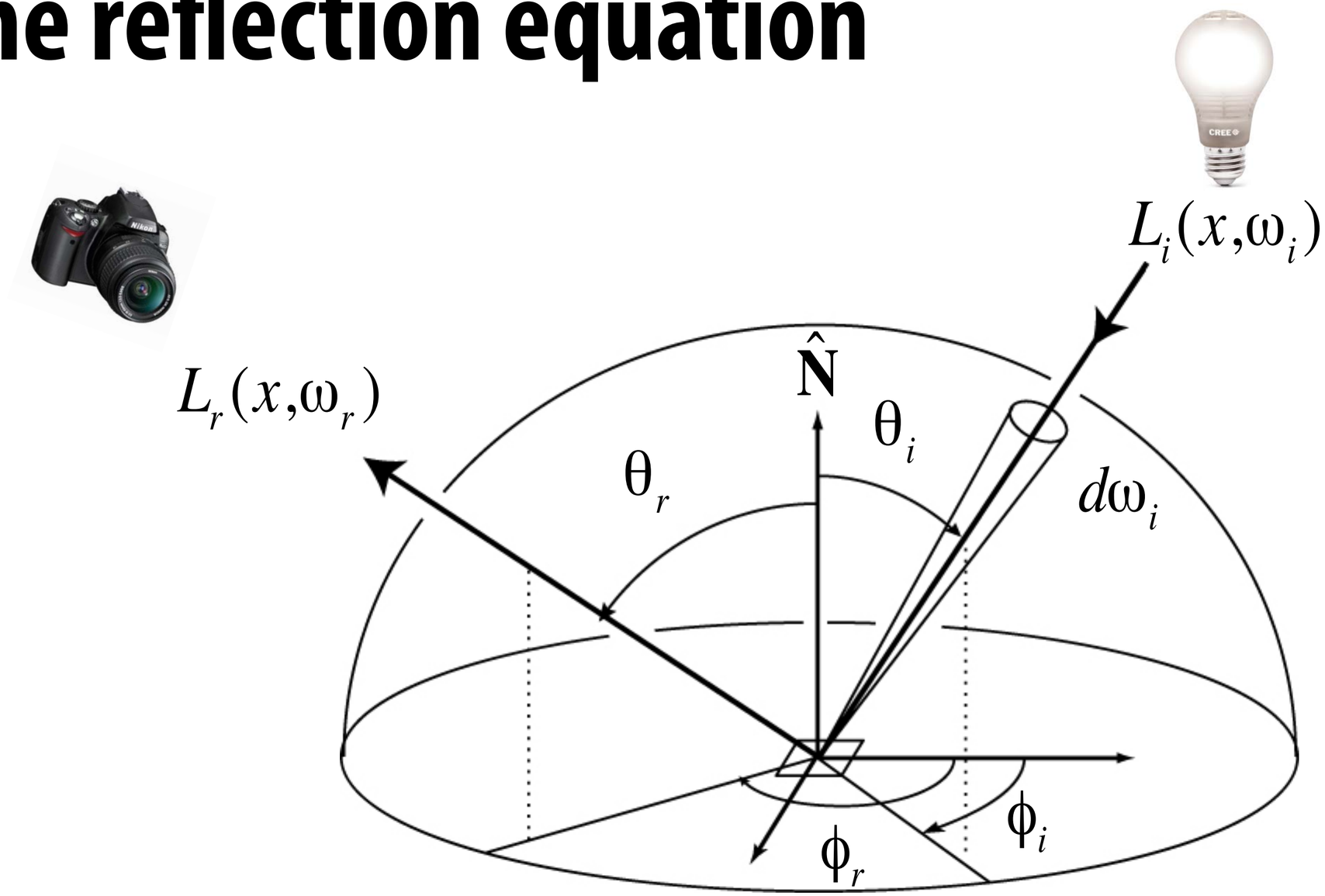
BRDF

BSSRDF

Ok, so scattering is *complicated!*

What's a (relatively simple) algorithm that can capture all this behavior?

The reflection equation



$$\mathrm{d}L_r(\omega_r) = f_r(\omega_i \to \omega_r)\,\mathrm{d}L_i(\omega_i)\,\cos\theta_i$$

$$L_r(\mathrm{p},\omega_r) = \int_{H^2} f_r(\mathrm{p},\omega_i \to \omega_r)\,L_i(\mathrm{p},\omega_i)\,\cos\theta_i\,\mathrm{d}\omega_i$$

The reflection equation

- **Key piece of overall rendering equation:**

$$L_r(\mathrm{p}, \omega_r) = \int_{H^2} f_r(\mathrm{p}, \omega_i \to \omega_r)\, L_i(\mathrm{p}, \omega_i)\, \cos\theta_i \, \mathrm{d}\omega_i$$

  - **Approximate integral via *Monte Carlo integration***
  - **Generate directions ω_j sampled from some distribution $p(\omega)$**
  - **Compute the estimator**

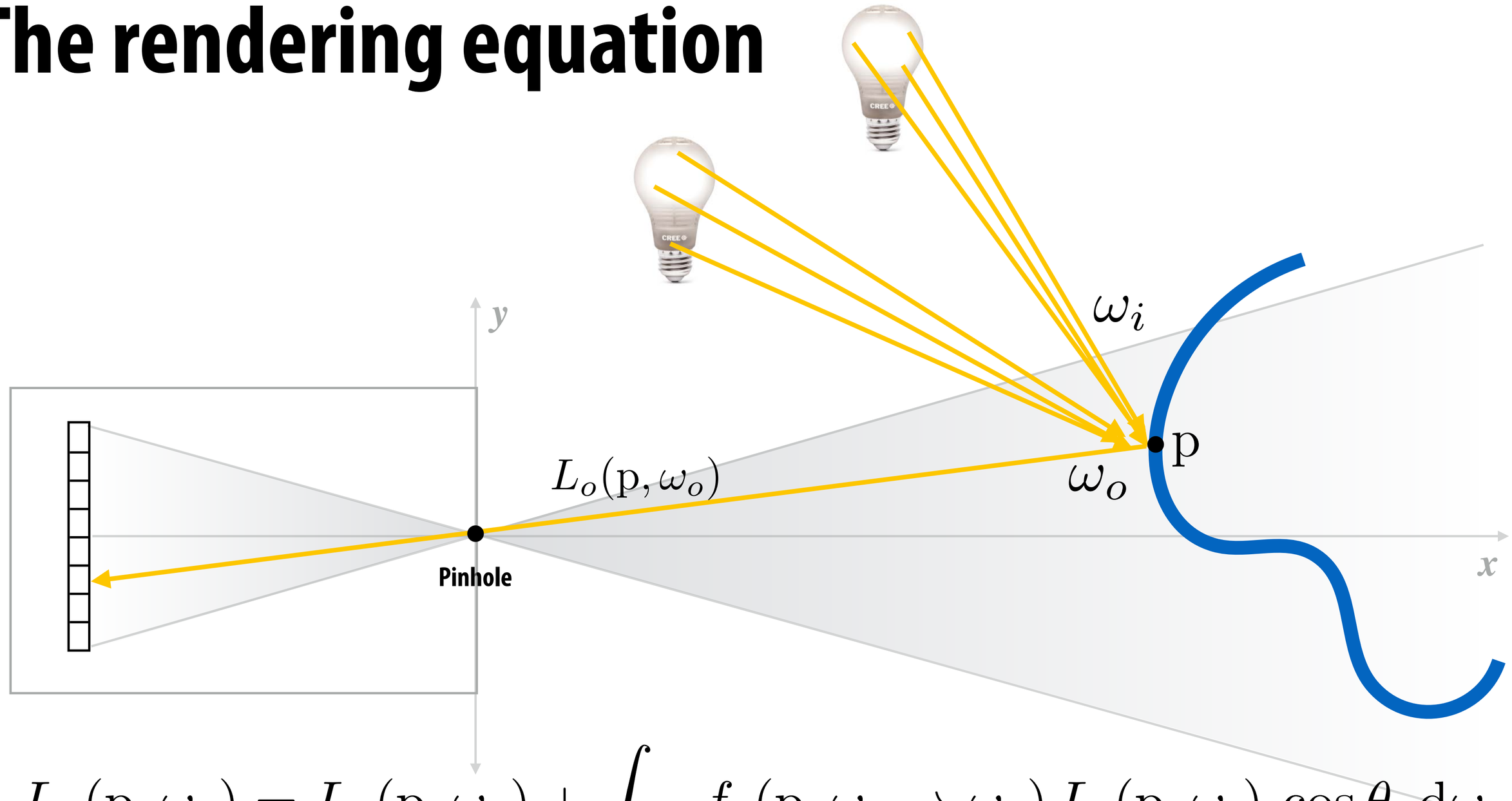
$$\frac{1}{N} \sum_{j=1}^{N} \frac{f_r(\mathrm{p}, \omega_j \to \omega_r)\, L_i(\mathrm{p}, \omega_j)\, \cos\theta_j}{p(\omega_j)}$$

  - **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



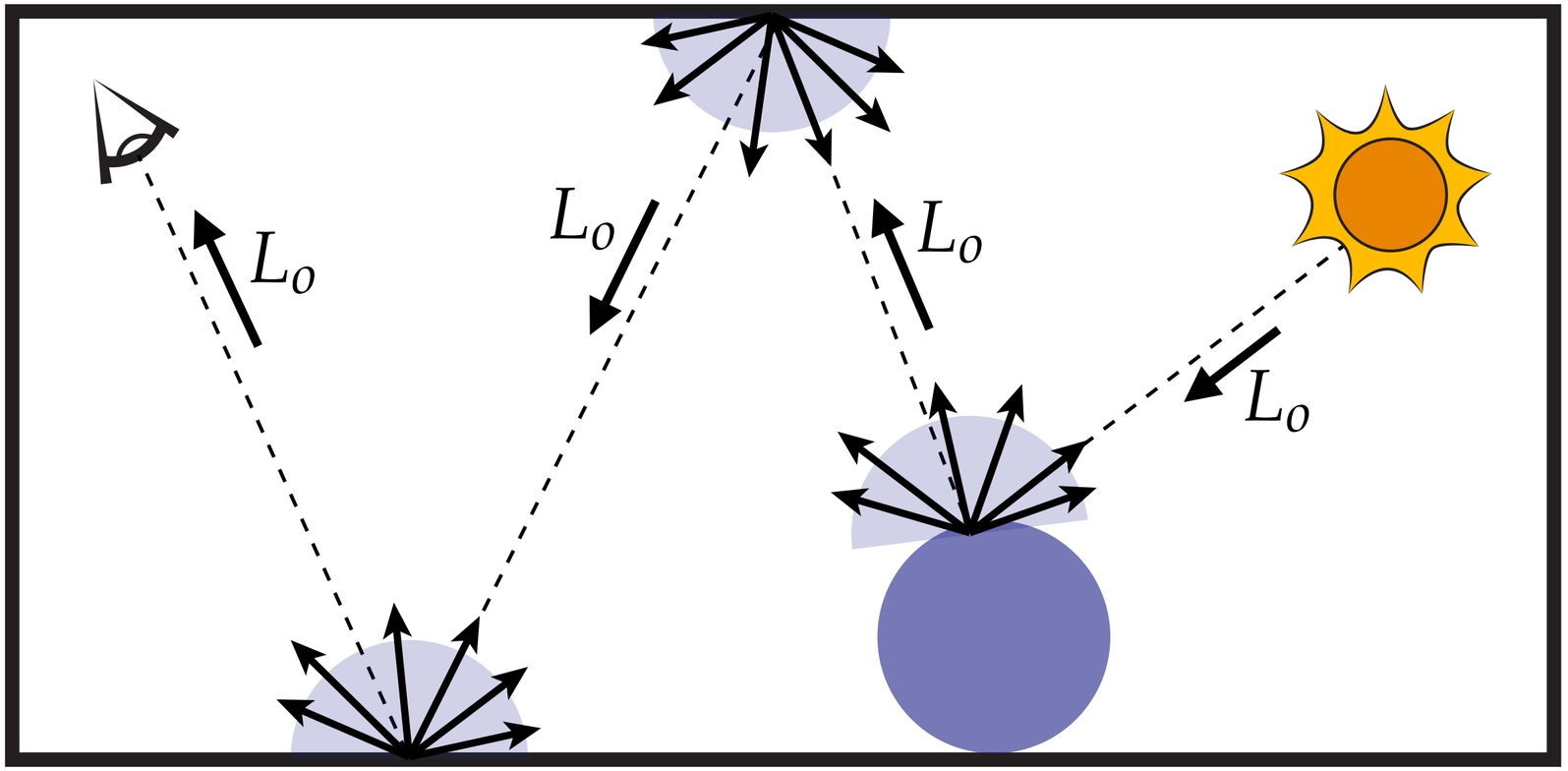
$$L_o(\mathrm{p}, \omega_o) = L_e(\mathrm{p}, \omega_o) + \int_{H^2} f_r(\mathrm{p}, \omega_i \to \omega_o)\, L_i(\mathrm{p}, \omega_i)\, \cos\theta_i \, \mathrm{d}\omega_i$$

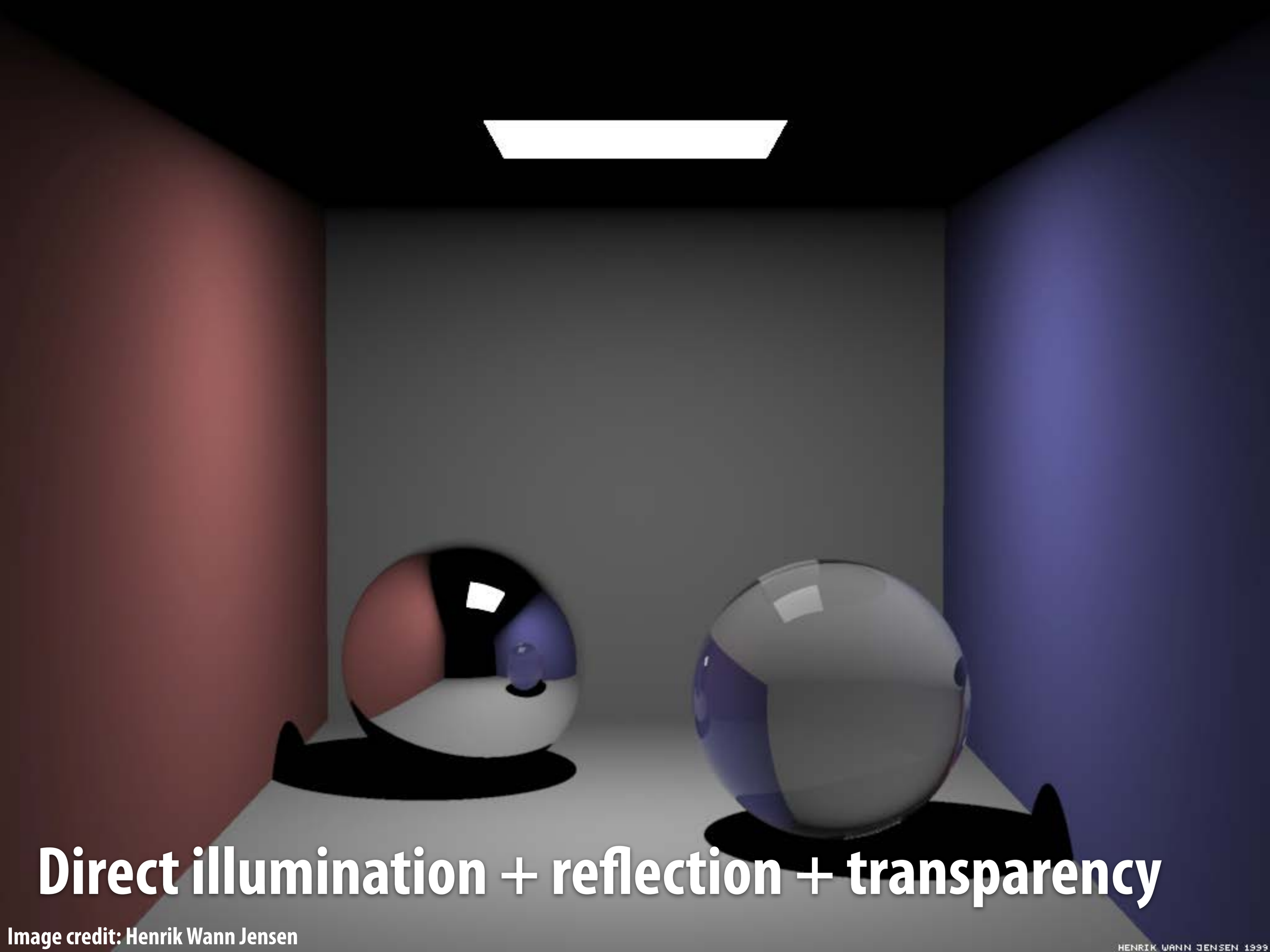
Now that we know how to handle reflection, how do we solve the full rendering equation? Have to determine incident radiance…

Key idea in (efficient) rendering: take advantage of special knowledge to break up integration into “easier” components.

Path tracing: overview

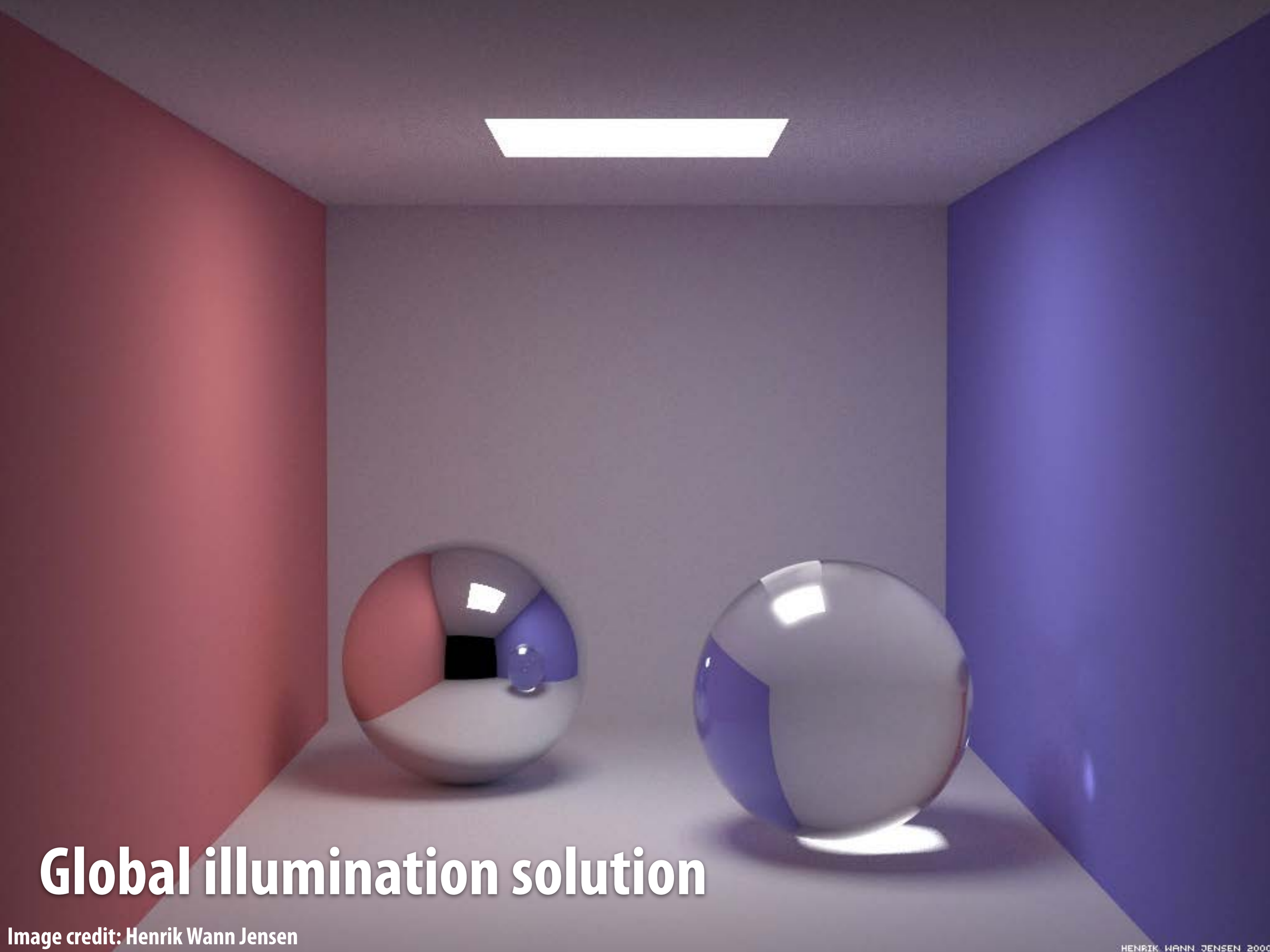
- Partition the rendering equation into direct and indirect illumination
- Use *Monte Carlo* to estimate each partition separately
  - One sample for each
  - Assumption: 100s of samples per pixel
- Terminate paths with *Russian roulette*





Direct illumination + reflection + transparency

Image credit: Henrik Wann Jensen



Global illumination solution

Image credit: Henrik Wann Jensen

Next Time: Monte Carlo integration

$$\int_{\Omega} f(p)\ dp \quad \approx \quad \mathrm{vol}(\Omega)\frac{1}{N}\sum_{i=1}^{N} f(X_i)$$

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

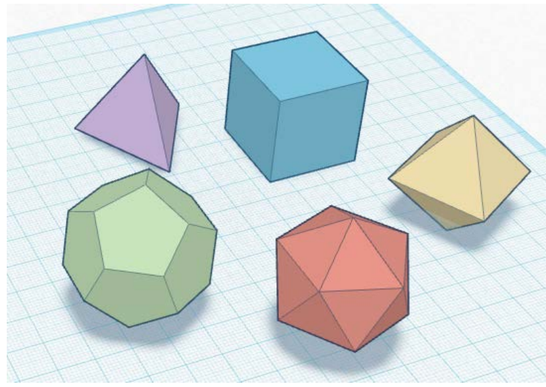
- Combine into final *Monte Carlo ray tracing* algorithm
- Alternative to rasterization, lets us generate *much* more realistic images (usually at much greater cost…)

Photorealistic Rendering—Basic Goal

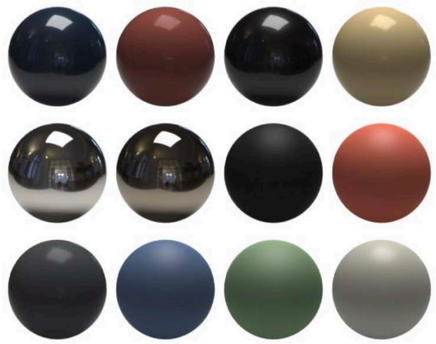
What are the INPUTS and OUTPUTS?

camera

geometry



materials



lights

("scene")

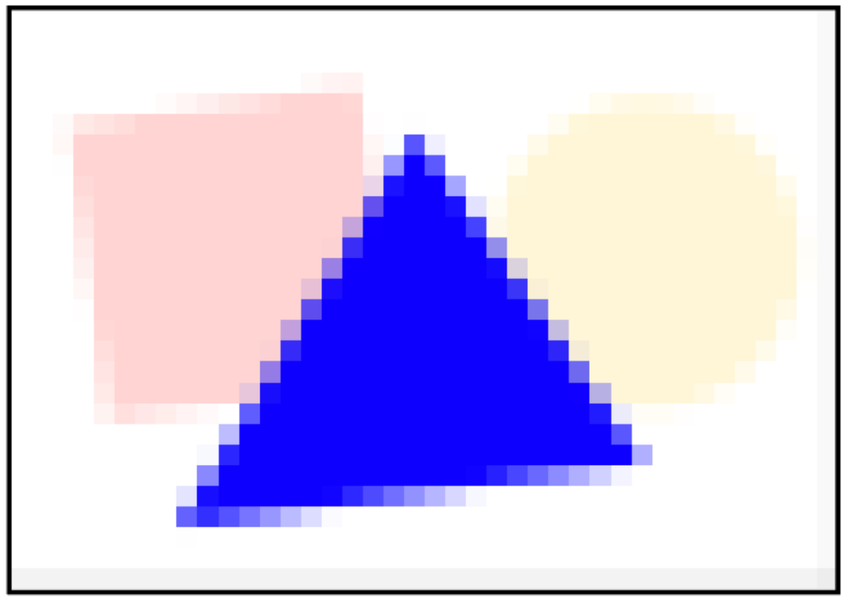
Ray Tracer

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

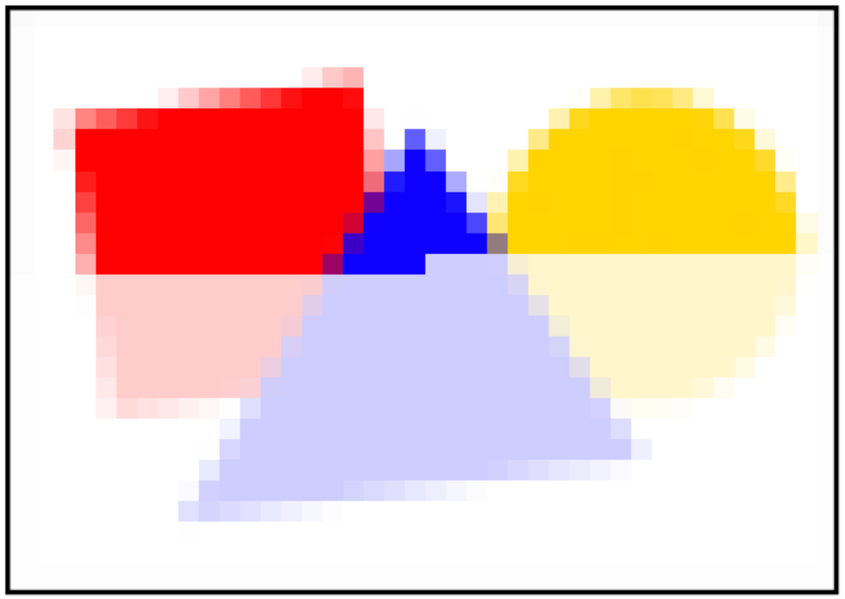
RASTERIZATION



```
for each primitive:
    for each sample:
        determine coverage
        evaluate color
```

(Use *Z-buffer* to determine which primitive is visible)

RAY TRACING



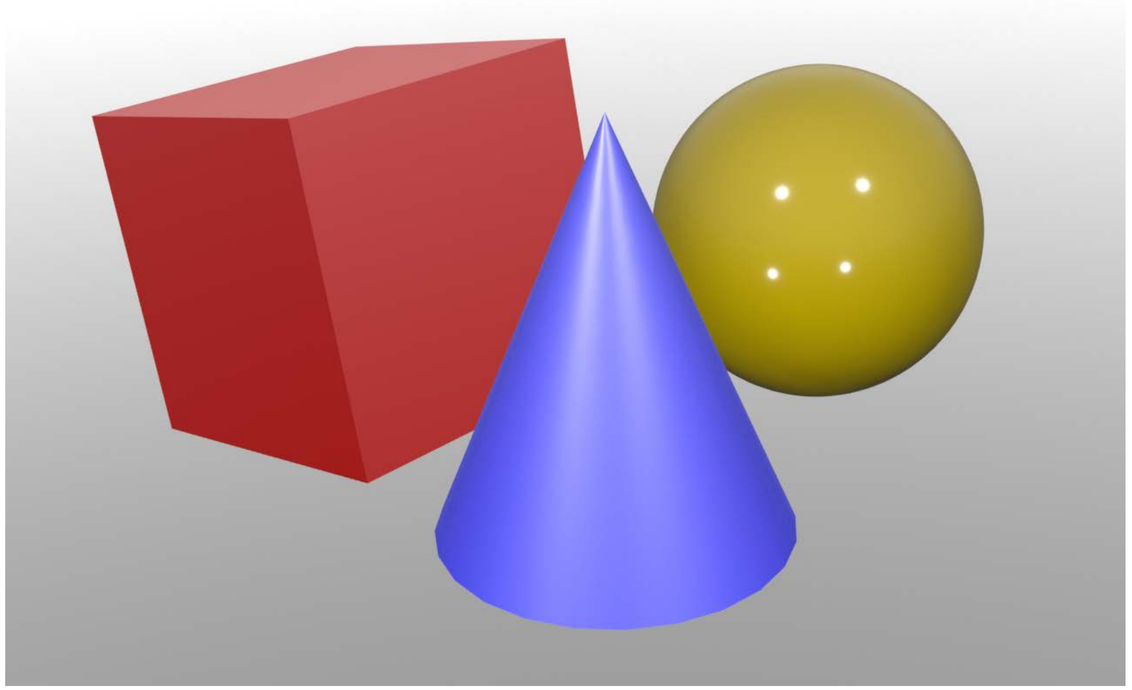
```
for each sample:
    for each primitive:
        determine coverage
        evaluate color
```

(Use spatial data structure like BVH to determine which primitive is visible)

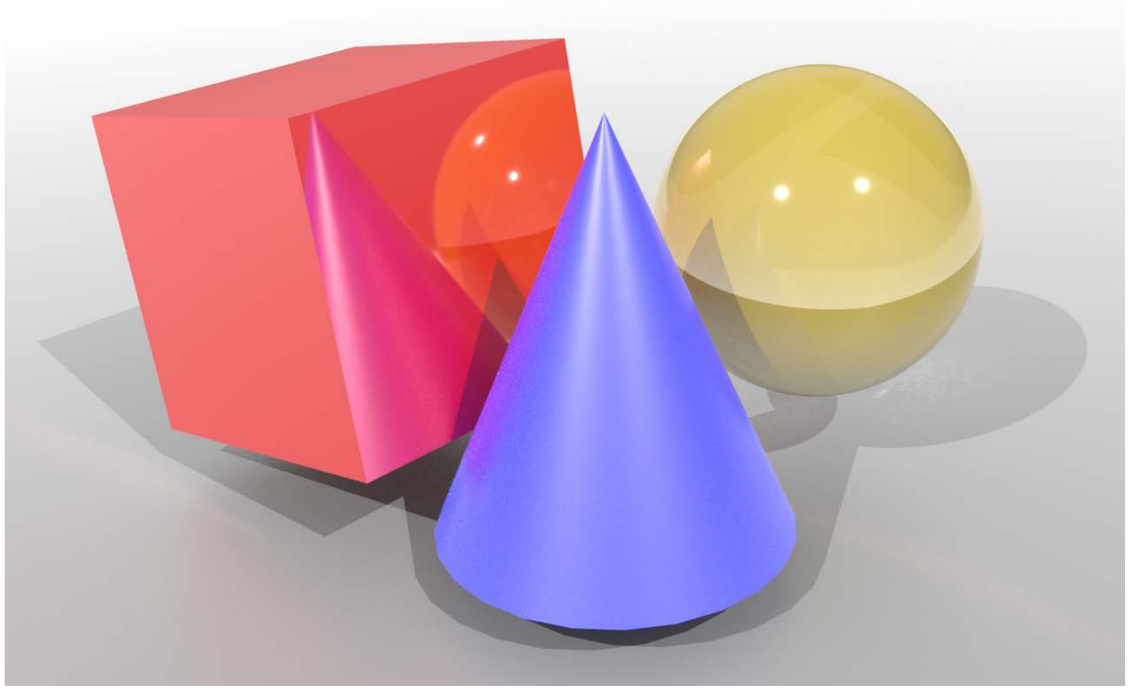
Ray Tracing vs. Rasterization—Illumination

- 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

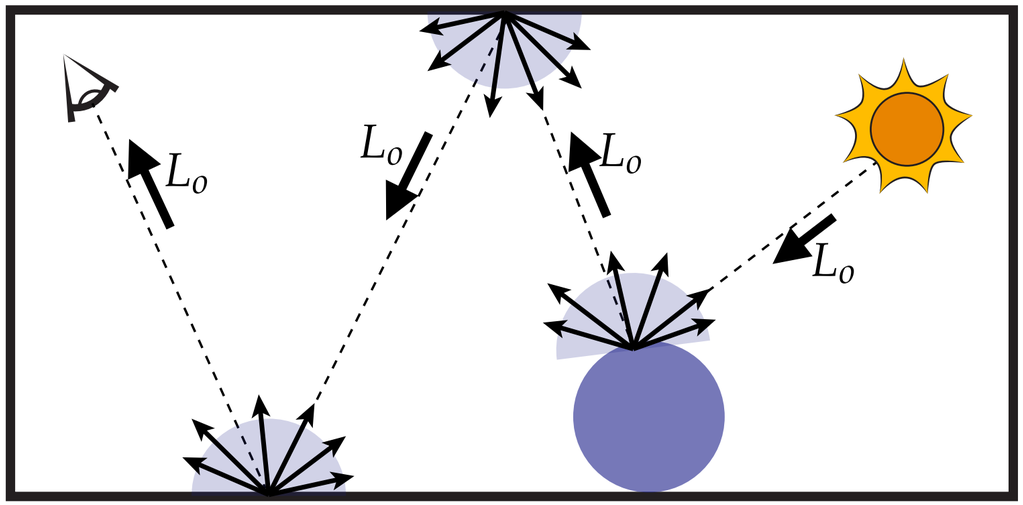


Q: What illumination effects are missing from the image on the left?

*But not *impossible* to do <u>some</u> 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



$$L_o(\mathbf{p},\omega_o) = L_e(\mathbf{p},\omega_o) + \int_{\mathcal{H}^2} f_r(\mathbf{p},\omega_i \to \omega_o) L_i(\mathbf{p},\omega_i) \cos\theta \, d\omega_i$$

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?

$$\boxed{\lim_{N\to\infty} \frac{|\Omega|}{N} \sum_{i=1}^{N} f(X_i) = \int_\Omega f(x)\, dx}$$

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)•1 + (1/2)•0 = 1/2

expected value of random variable Y

number of possible outcomes

$$E(Y) := \sum_{i=1}^{k} p_i y_i$$

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?)

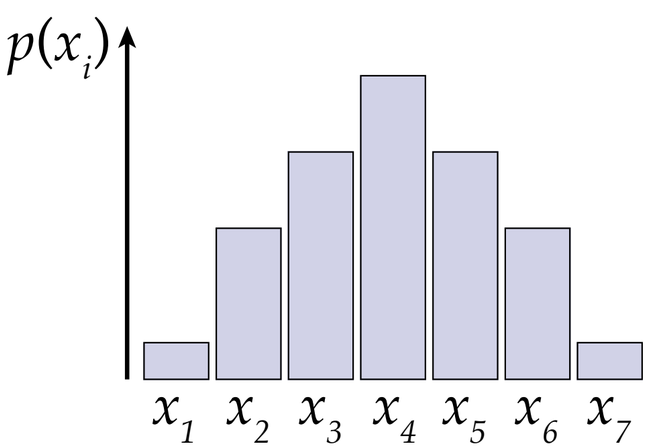
Variance

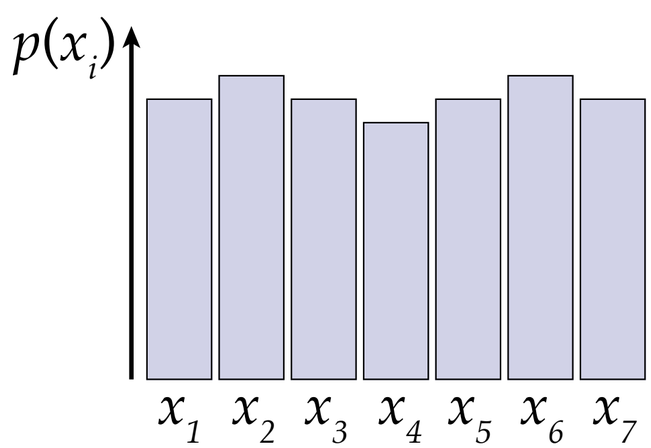
Intuition: how far are our samples from the average, on average?

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?)

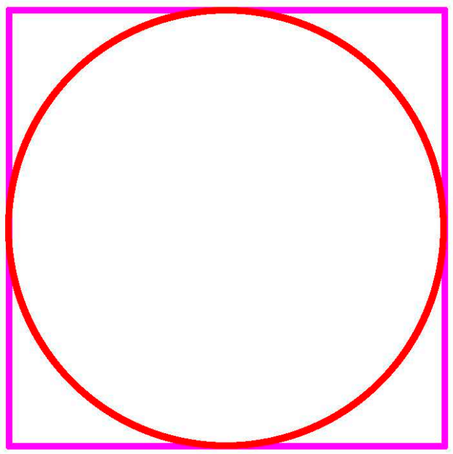
Law of Large Numbers

- **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 π
1	4.000000
10	3.200000
100	3.240000
1000	3.112000
10000	3.163600
100000	3.139520
1000000	3.141764



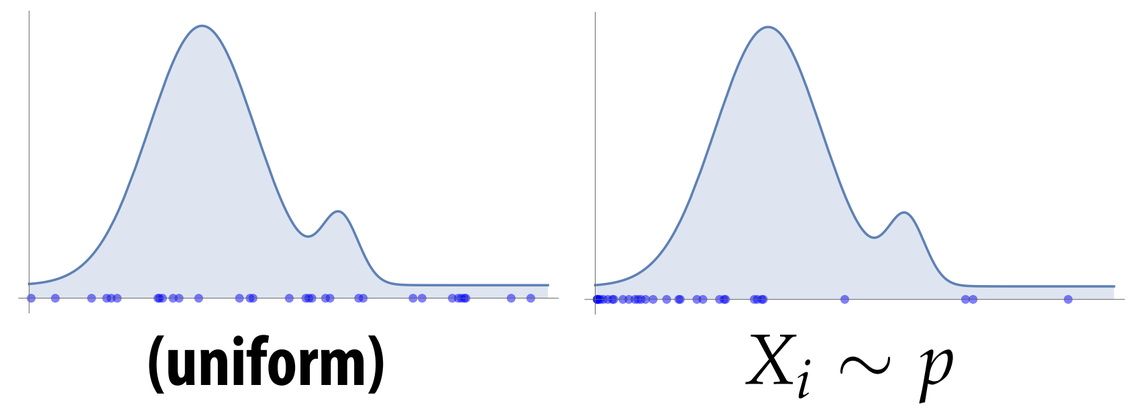
Q: Why is the law of large numbers important for Monte Carlo ray tracing?

A: No matter how hard the integrals are (crazy lighting, geometry, materials, etc.), can *always get the right image by taking more samples.**

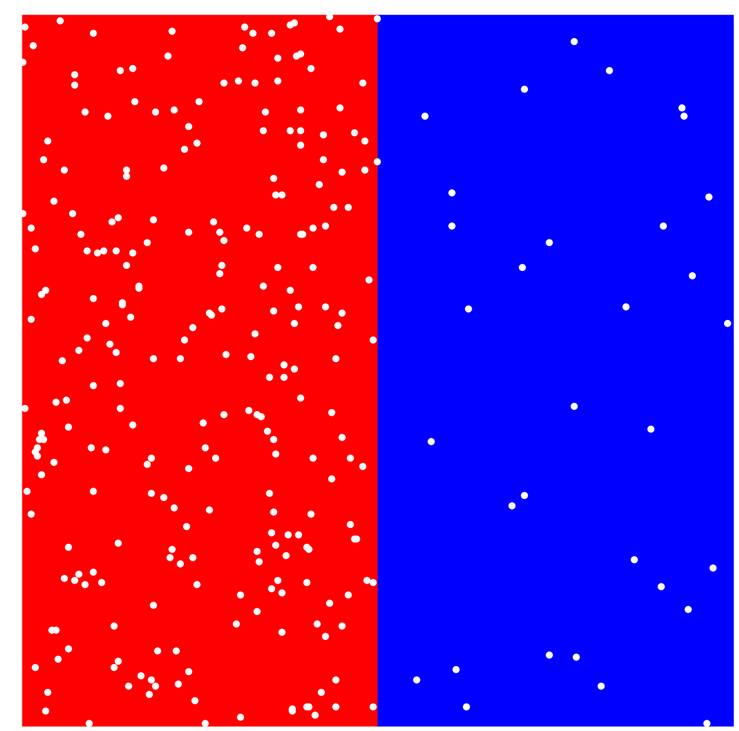
*As long as we make sure to sample all possible kinds of light paths…

Biasing

- 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(Xi) 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

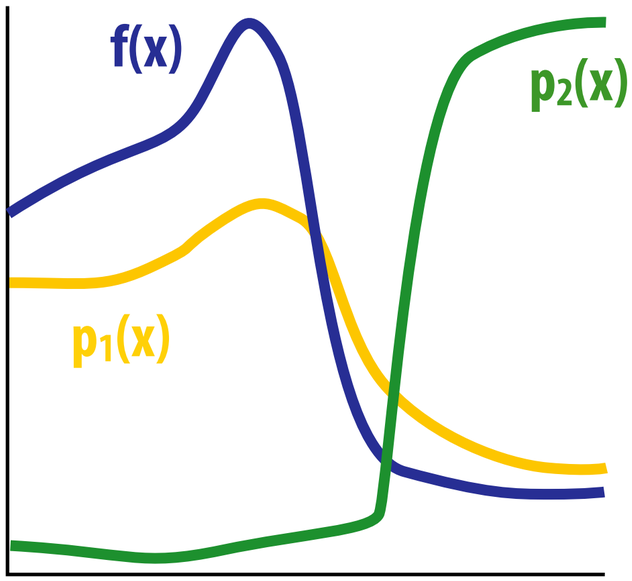


$$\int_\Omega f(x)\,dx \approx \frac{1}{N}\sum_{i=1}^{N}\frac{f(X_i)}{p(X_i)}$$



Importance sampling

Q: Ok, so then WHERE is the best place to take samples?

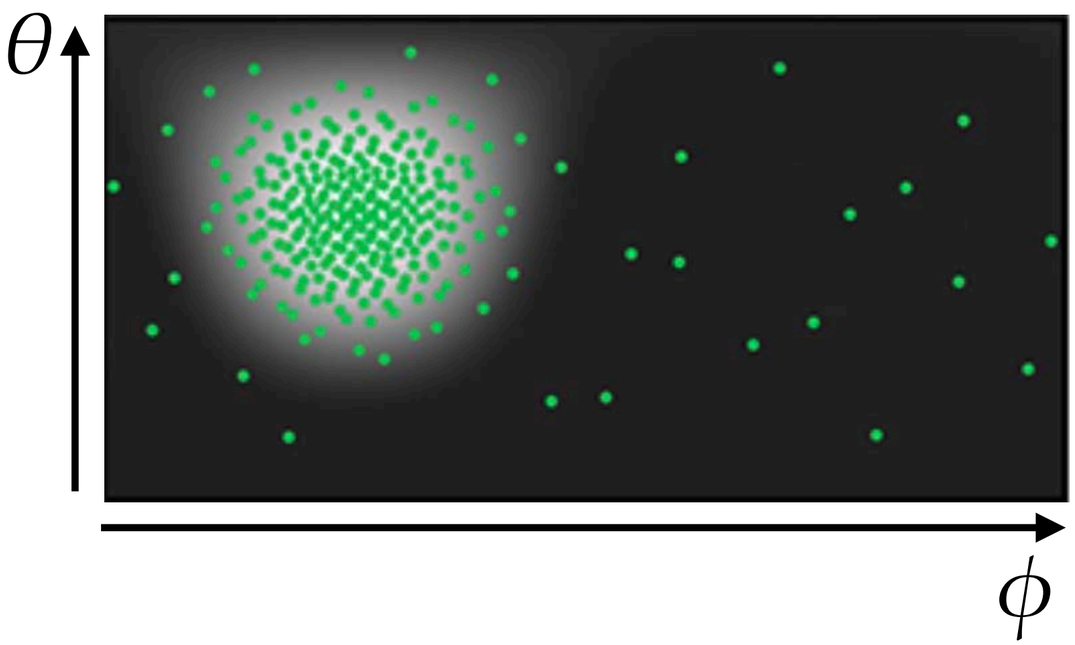


Think:

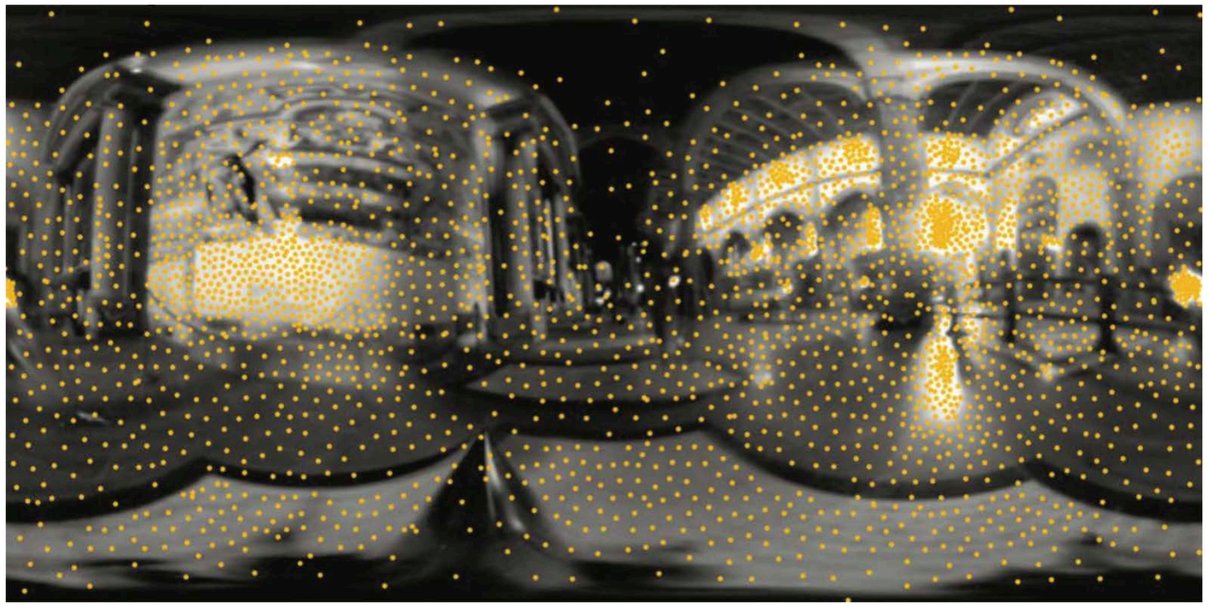
- **What is the behavior of $f(x)/p_1(x)$? $f(x)/p_2(x)$?**
- **How does this impact the variance of the estimator?**

Idea: put *more* where integrand is *large* ("most useful samples"). E.g.:

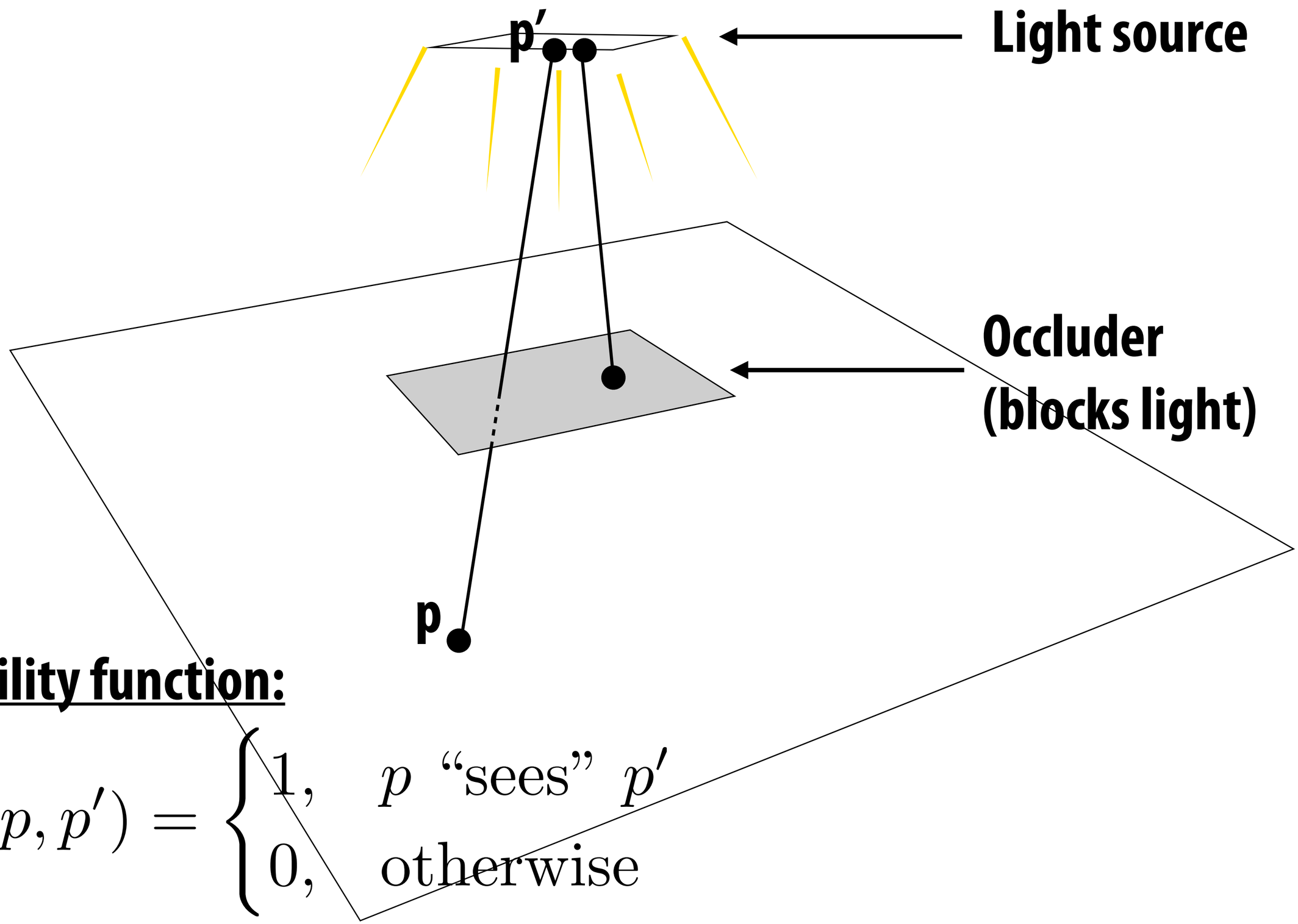
(BRDF)



(image-based lighting)



Example: Direct Lighting



Visibility function:

$$V(p, p') = \begin{cases} 1, & p \text{ “sees” } p' \\ 0, & \text{otherwise} \end{cases}$$

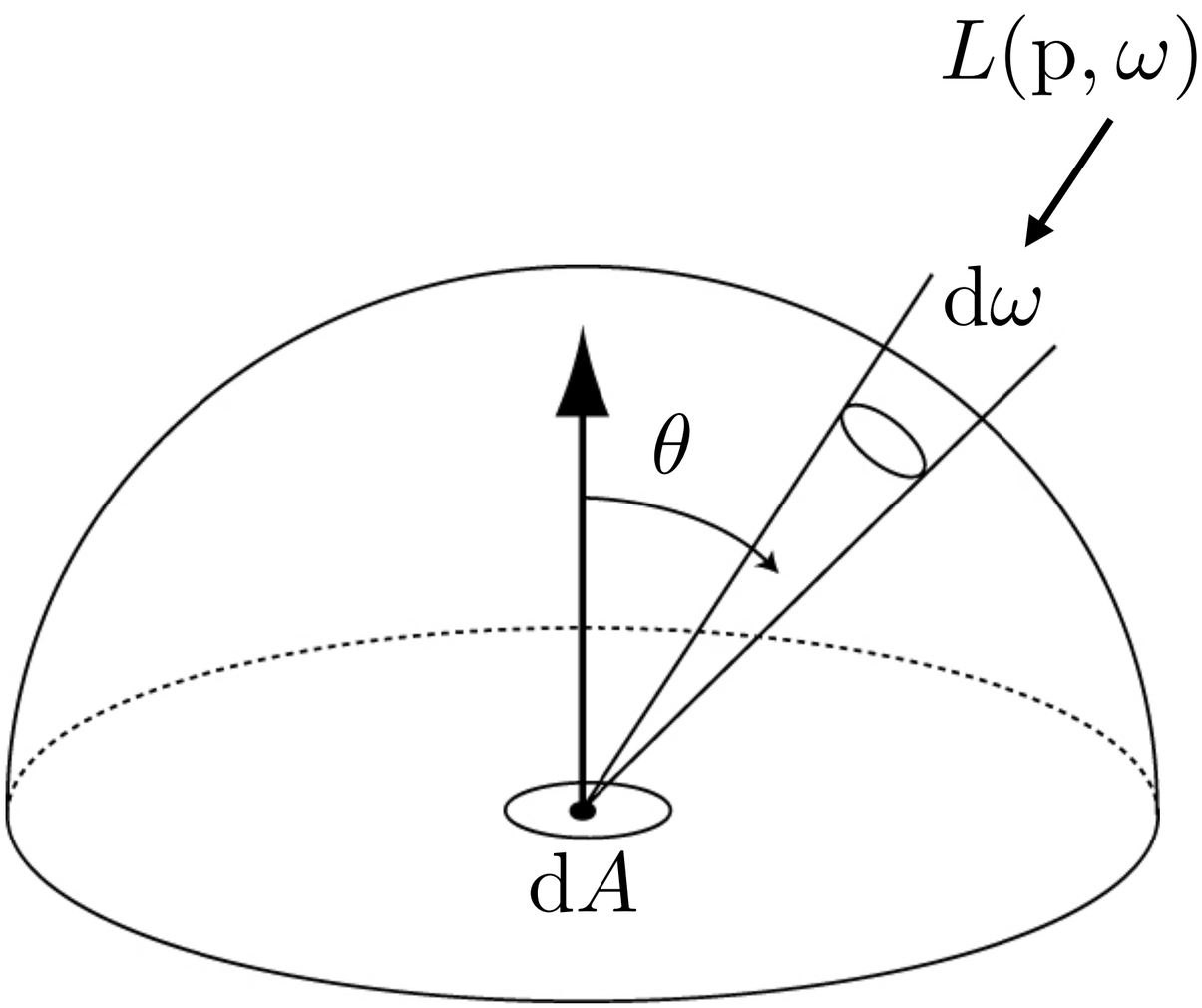
How bright is each point on the ground?

Direct lighting—uniform sampling

Uniformly-sample hemisphere of directions with respect to solid angle

$$p(\omega) = \frac{1}{2\pi}$$

$$E(\mathrm{p}) = \int L(\mathrm{p}, \omega)\, \cos\theta \, \mathrm{d}\omega$$



Estimator:

$$X_i \sim p(\omega)$$

$$Y_i = f(X_i)$$

$$Y_i = L(\mathrm{p}, \omega_i) cos\, \theta_i$$

$$F_N = \frac{2\pi}{N} \sum_{i=1}^{N} Y_i$$

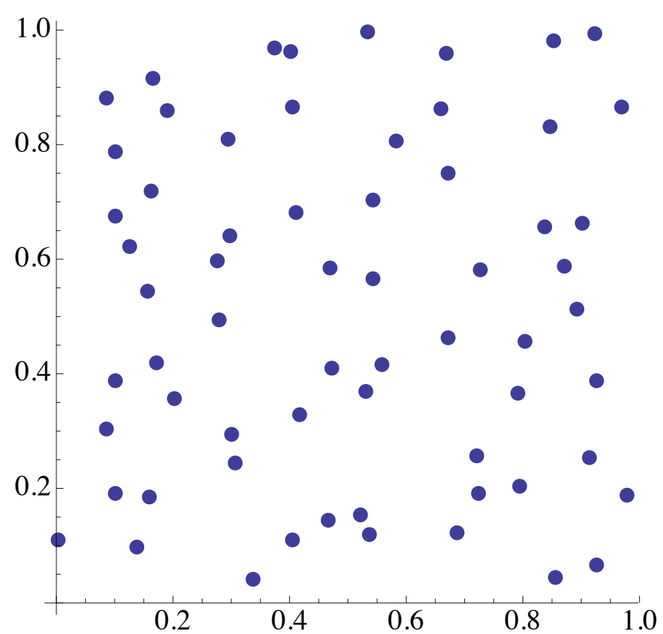
Aside: Picking points on unit hemisphere

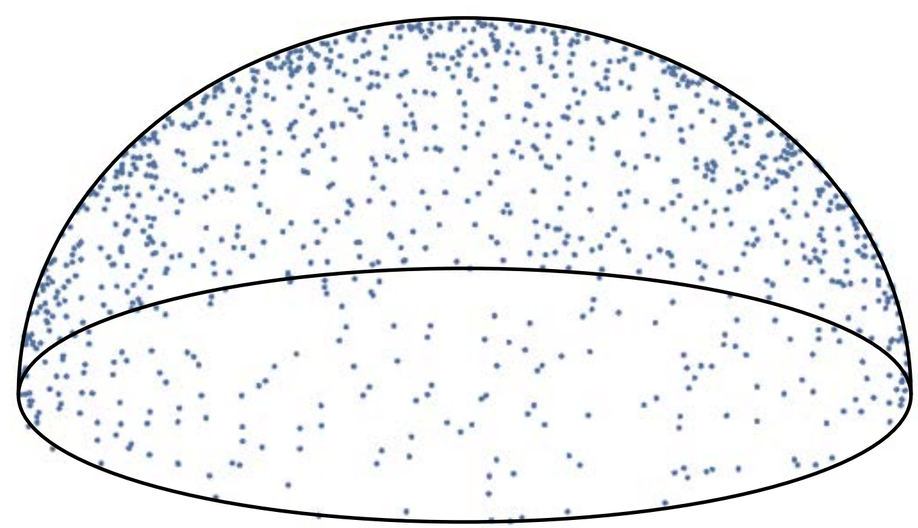
How do we <u>uniformly</u> sample directions from the hemisphere?

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

$$p(\omega) = \frac{1}{2\pi}$$

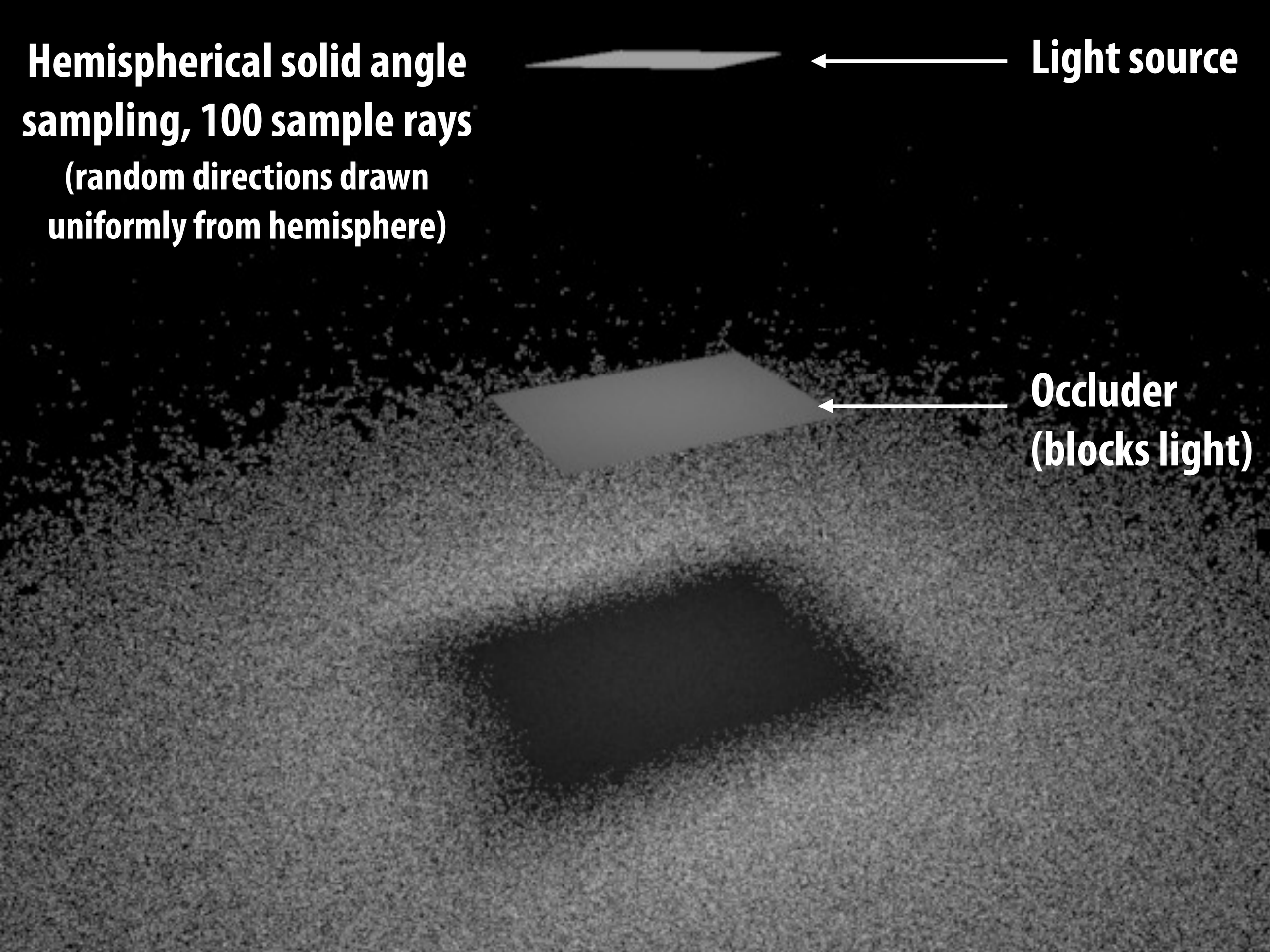
$$E(\mathrm{p}) = \int L(\mathrm{p}, \omega)\, \cos\theta \, \mathrm{d}\omega$$

Given surface point *p*

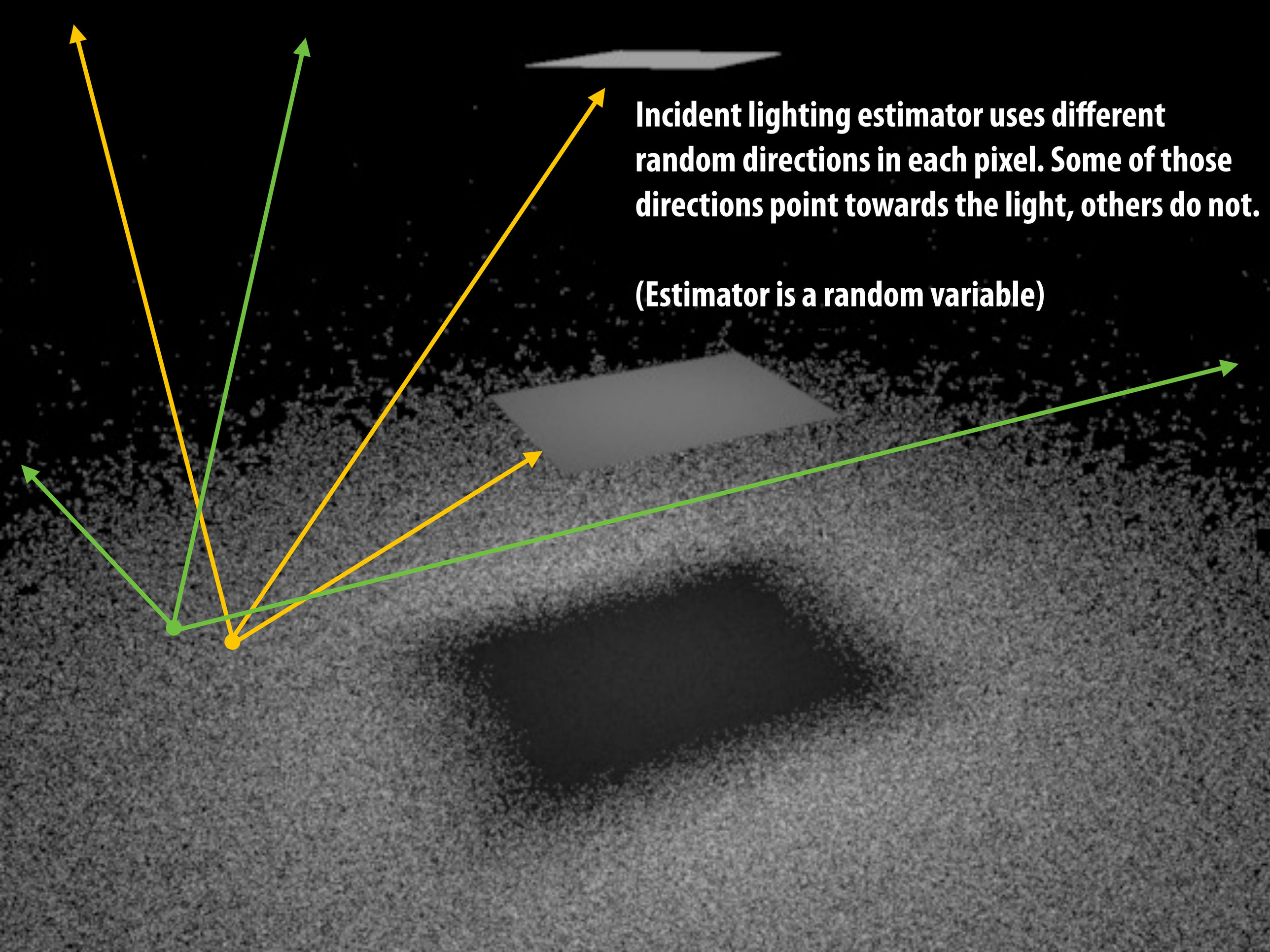
A ray tracer evaluates radiance along a ray (see Raytracer::trace_ray() in raytracer.cpp)

For each of N samples:

- **Generate random direction:** ω_i
- **Compute incoming radiance arriving** L_i **at *p* from direction:** ω_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)
Light source
Occluder (blocks light)

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

$$E(\mathrm{p}) = \int L(\mathrm{p}, \omega)\, \cos\theta \, \mathrm{d}\omega$$

Previously: just integrate over all directions

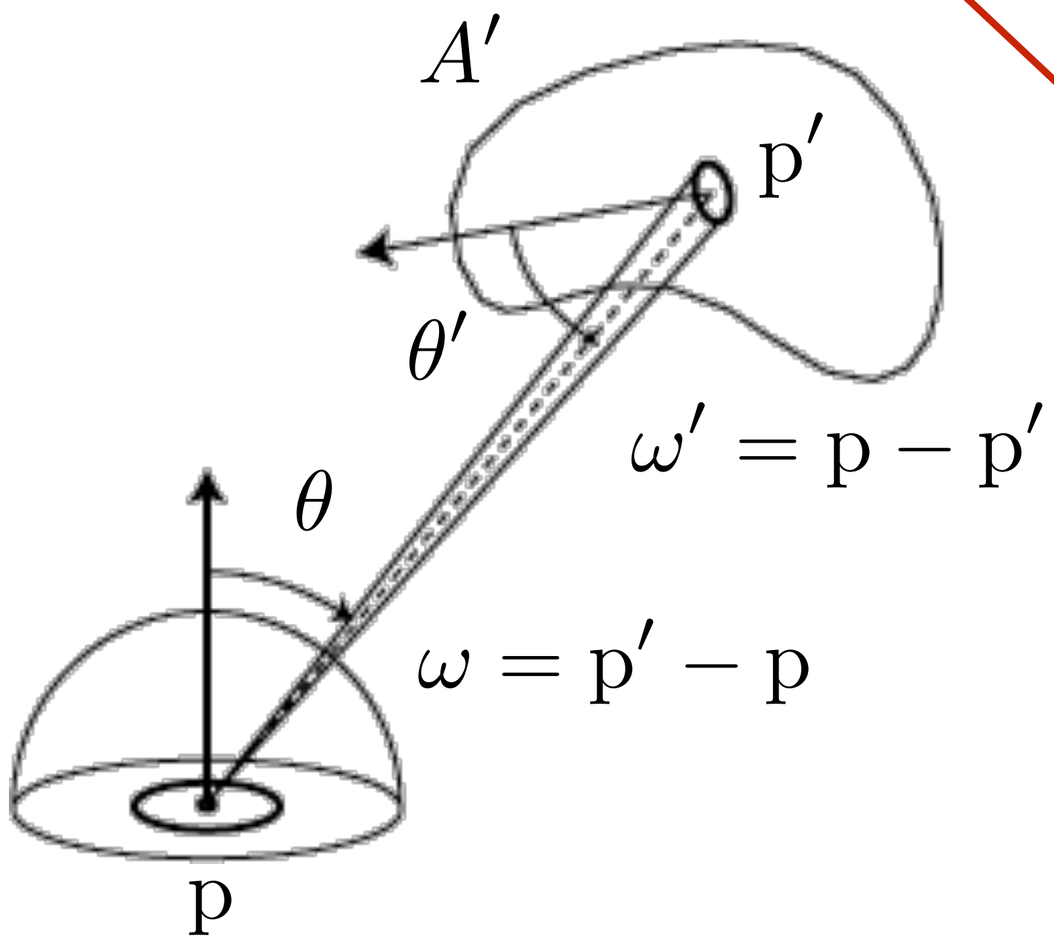
$$E(\mathrm{p}) = \int_{A'} L_o(\mathrm{p}', \omega')\, V(\mathrm{p}, \mathrm{p}')\, \frac{\cos\theta \cos\theta'}{|\mathrm{p} - \mathrm{p}'|^2}\, \mathrm{d}A'$$

Change of variables to integrate over area of light *

$$d\omega = \frac{dA}{|\mathrm{p}' - \mathrm{p}|^2} = \frac{dA' \cos\theta}{|\mathrm{p}' - \mathrm{p}|^2}$$

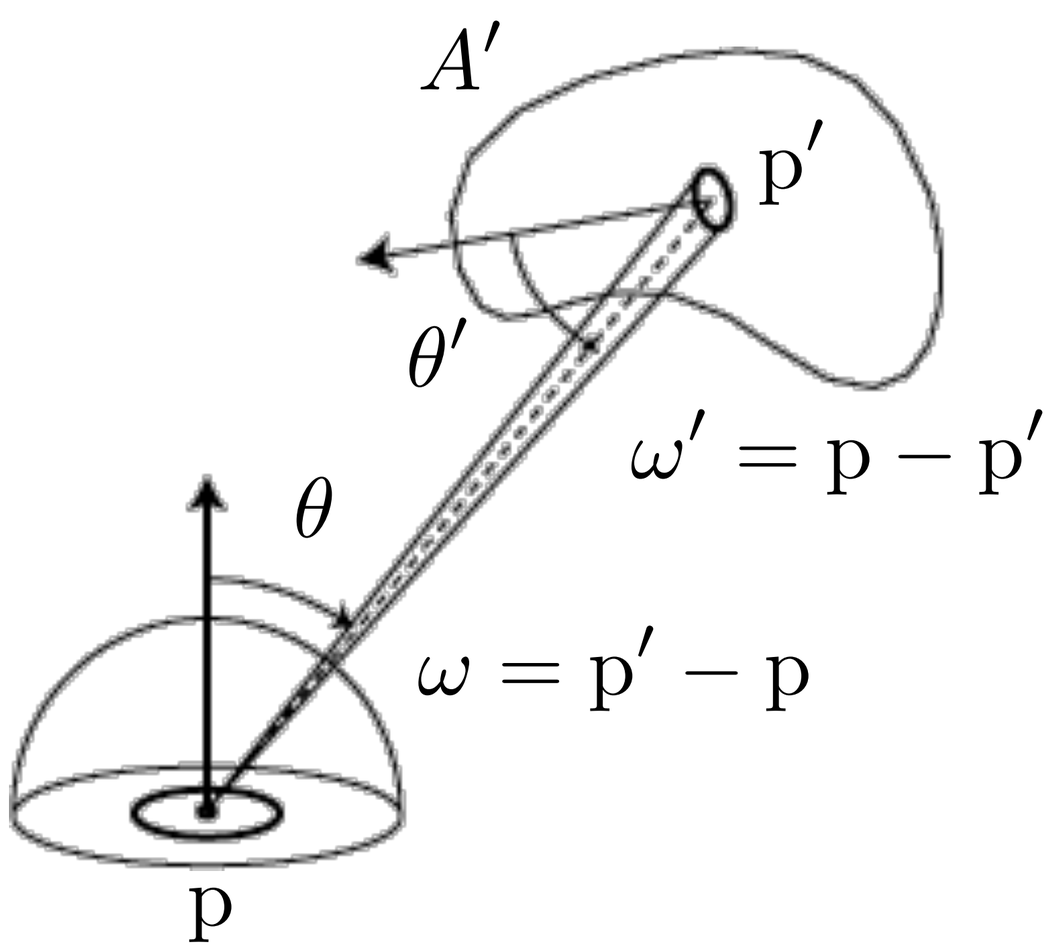
Binary visibility function:
1 if p′ is visible from p, 0 otherwise
(accounts for light occlusion)

Outgoing radiance from light point p, in direction w′ towards p



Direct lighting: area integral

$$E(\mathrm{p}) = \int_{A'} L_o(\mathrm{p}', \omega')\, V(\mathrm{p}, \mathrm{p}')\, \frac{\cos\theta \cos\theta'}{|\mathrm{p} - \mathrm{p}'|^2}\, \mathrm{d}A'$$



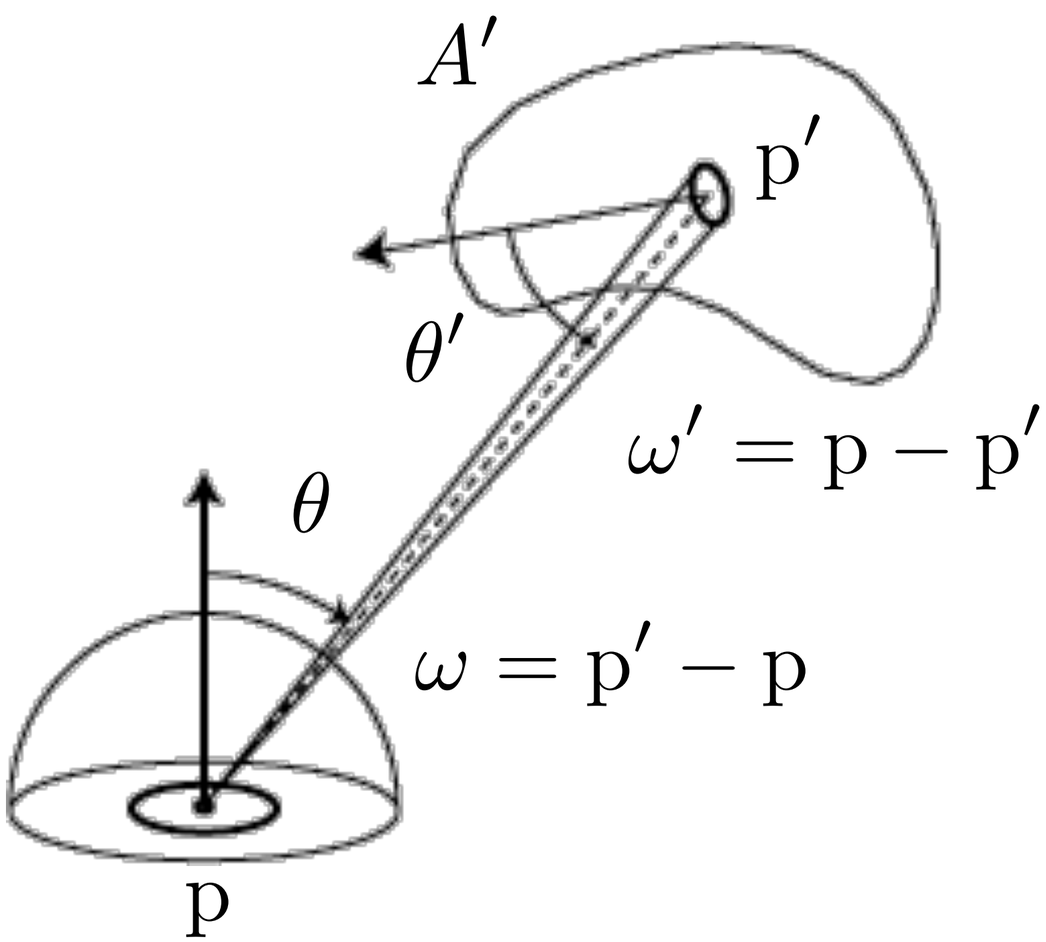
Sample shape uniformly by area A′

$$\int_{A'} p(\mathrm{p}')\, \mathrm{d}A' = 1$$

$$p(\mathrm{p}') = \frac{1}{A'}$$

Direct lighting: area integral

$$E(\mathrm{p}) = \int_{A'} L_o(\mathrm{p}', \omega')\, V(\mathrm{p}, \mathrm{p}')\, \frac{\cos\theta \cos\theta'}{|\mathrm{p} - \mathrm{p}'|^2}\, \mathrm{d}A'$$



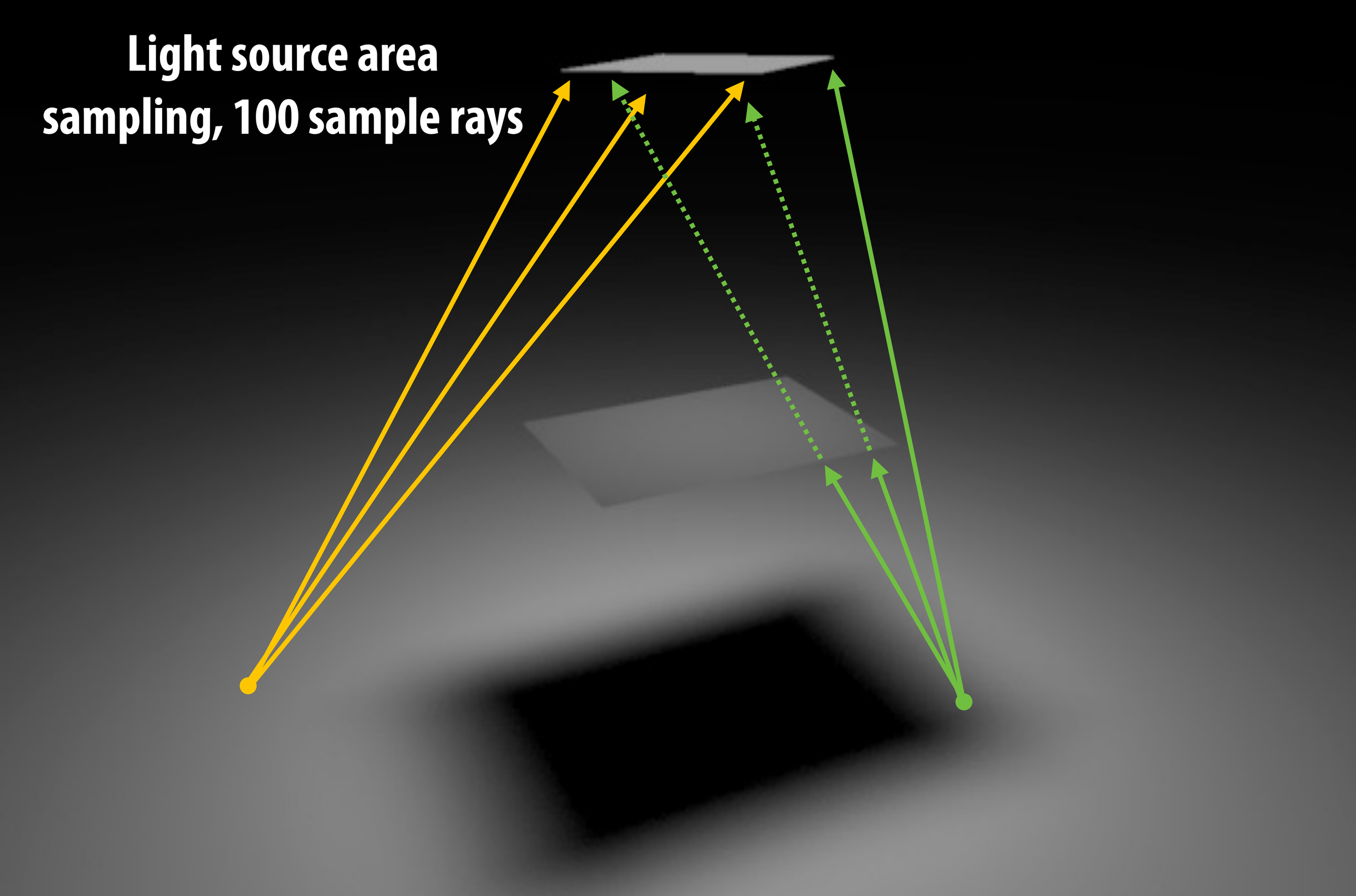
Probability:

$$p(\mathrm{p}') = \frac{1}{A'}$$

Estimator

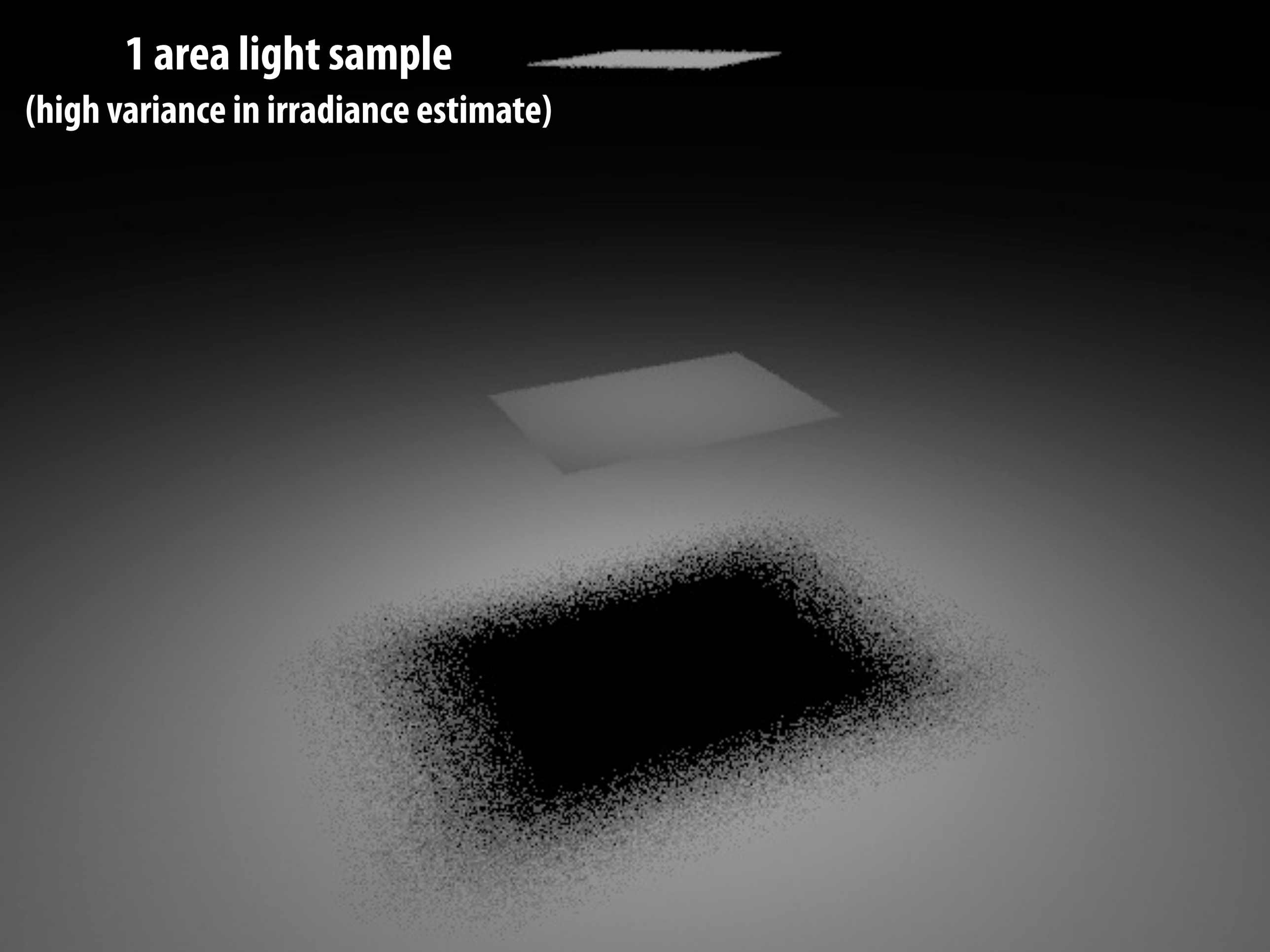
$$Y_i = L_o(\mathrm{p}'_i, \omega'_i)\, V(\mathrm{p}, \mathrm{p}'_i)\, \frac{\cos\theta_i \cos\theta'_i}{|\mathrm{p} - \mathrm{p}'_i|^2}$$

$$F_N = \frac{A'}{N} \sum_{i=1}^{N} Y_i$$

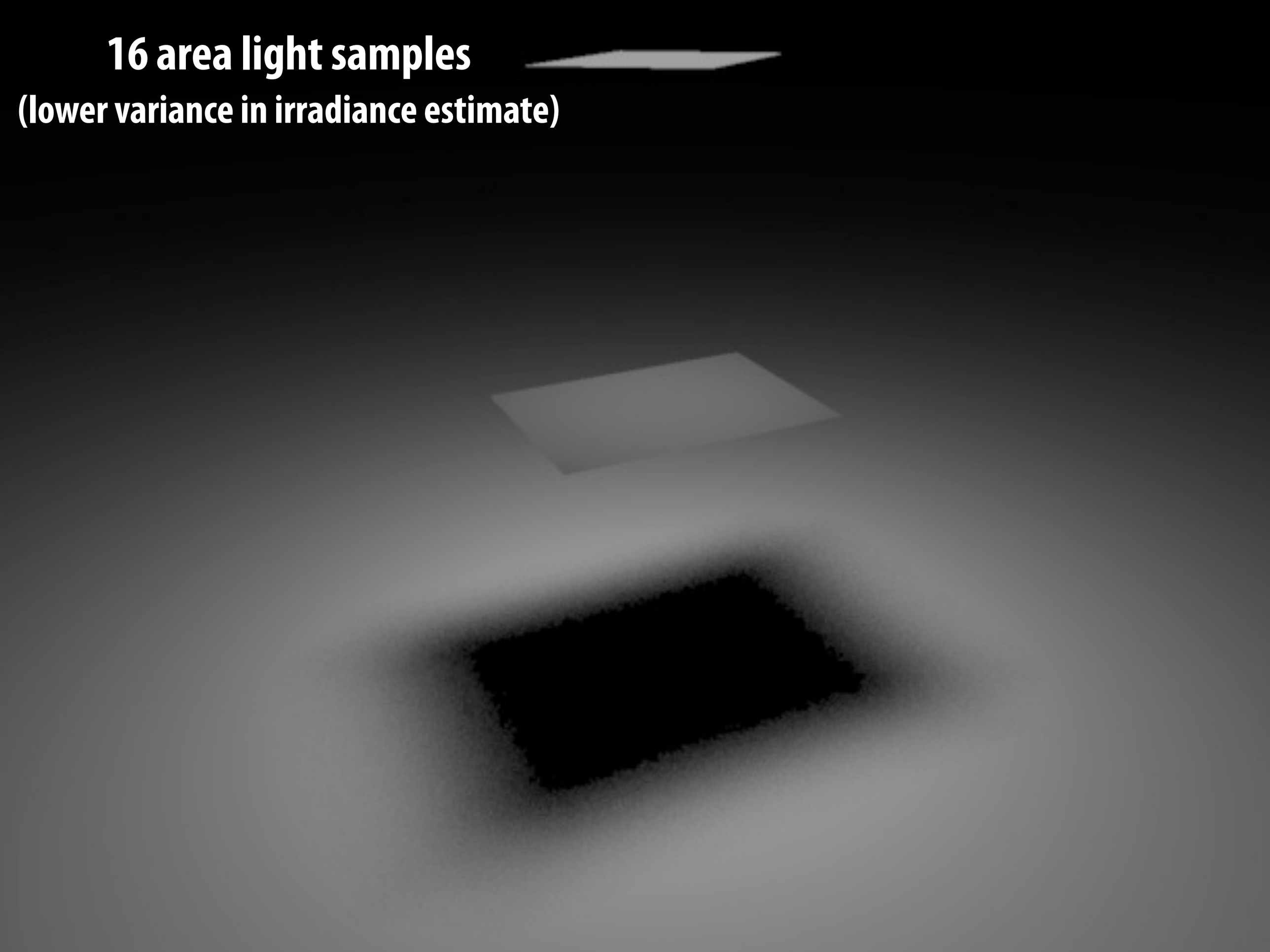


If no occlusion is present, all directions chosen in computing estimate "hit" the light source.
(Choice of direction only matters if portion of light is occluded from surface point *p*.)

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

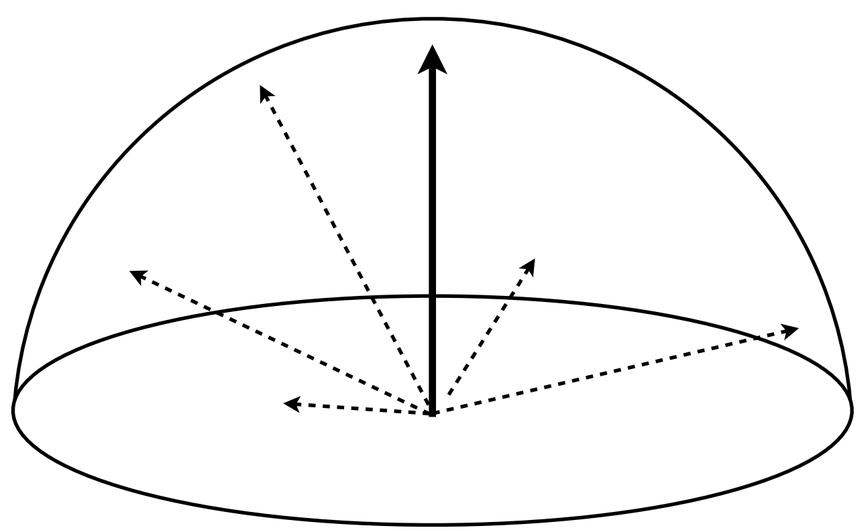
Example—Cosine-Weighted Sampling

Consider <u>uniform</u> hemisphere sampling in irradiance estimate:

$$f(\omega) = L_i(\omega)\cos\theta \qquad\qquad p(\omega) = \frac{1}{2\pi}$$

$$(\xi_1, \xi_2) = (\sqrt{1-\xi_1^2}\cos(2\pi\xi_2), \sqrt{1-\xi_1^2}\sin(2\pi\xi_2), \xi_1)$$

$$\int_\Omega f(\omega)\,\mathrm{d}\omega \approx \frac{1}{N}\sum_i^N \frac{f(\omega)}{p(\omega)} = \frac{1}{N}\sum_i^N \frac{L_i(\omega)\ \cos\theta}{1/2\pi} = \frac{2\pi}{N}\sum_i^N L_i(\omega)\cos\theta$$



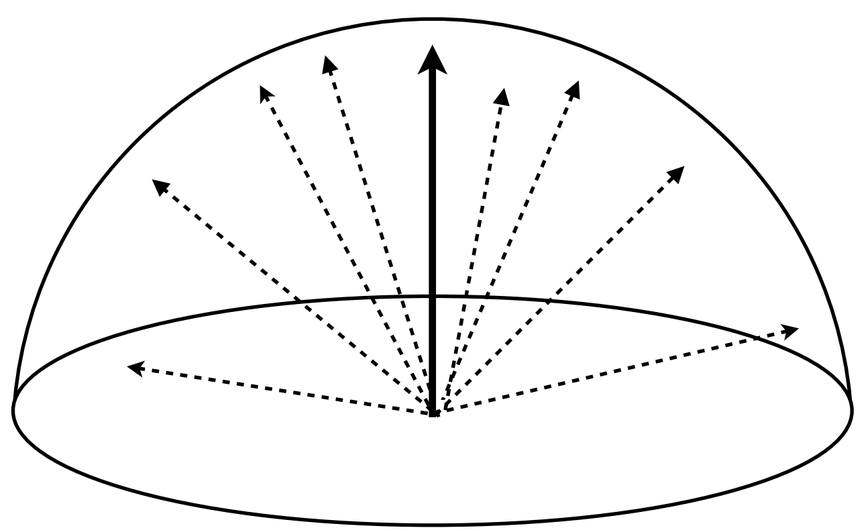
Example—Cosine-Weighted Sampling

<u>Cosine-weighted</u> hemisphere sampling in irradiance estimate:

$$f(\omega) = L_i(\omega)\cos\theta \qquad\qquad p(\omega) = \frac{\cos\theta}{\pi}$$

$$\int_\Omega f(\omega)\,\mathrm{d}\omega \approx \frac{1}{N}\sum_i^N \frac{f(\omega)}{p(\omega)} = \frac{1}{N}\sum_i^N \frac{L_i(\omega)\ \cos\theta}{\cos\theta/\pi} = \frac{\pi}{N}\sum_i^N L_i(\omega)$$

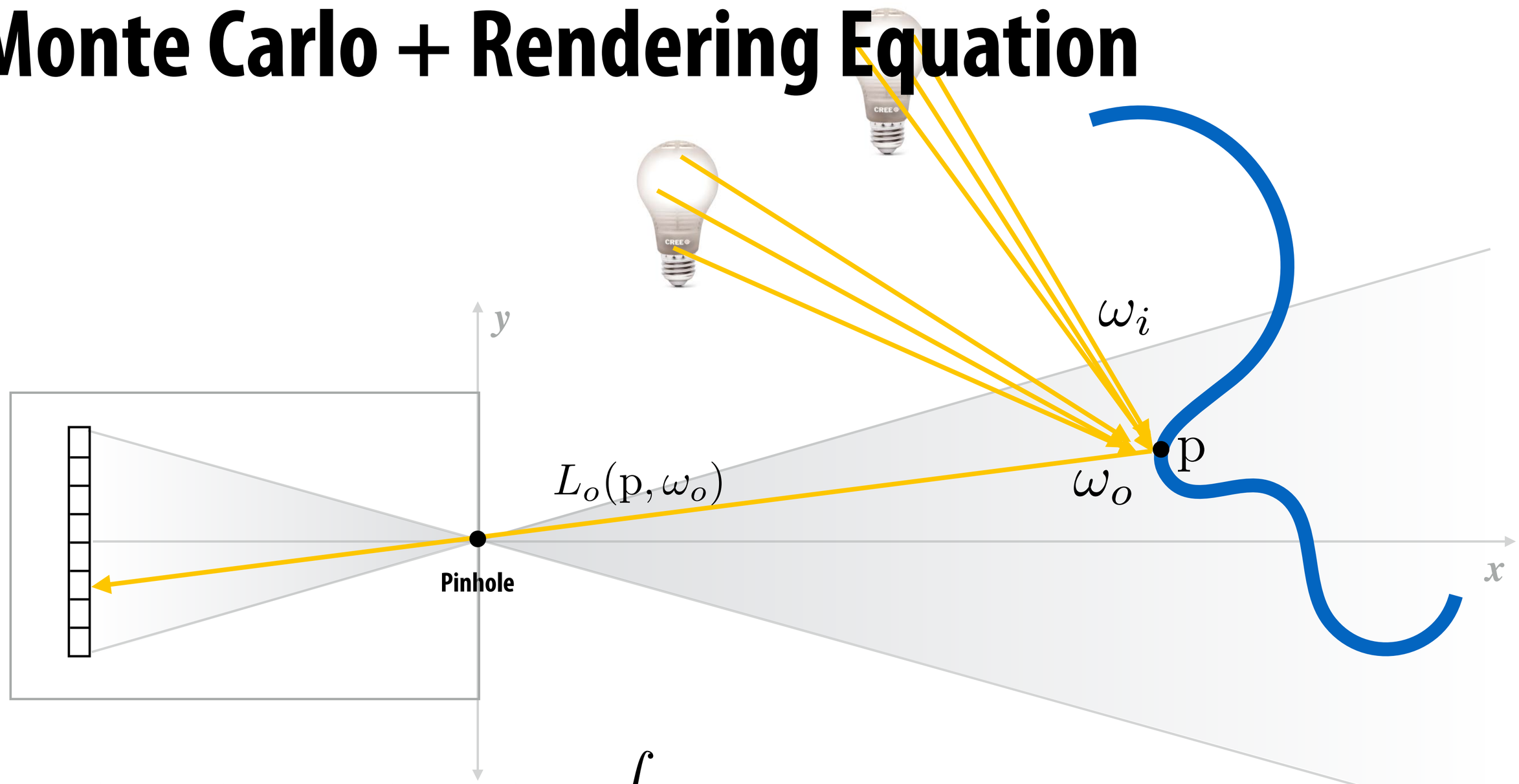
Idea: bias samples toward directions where $\cos\theta$ is large
(if L is constant, then these are the directions that contribute most)



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

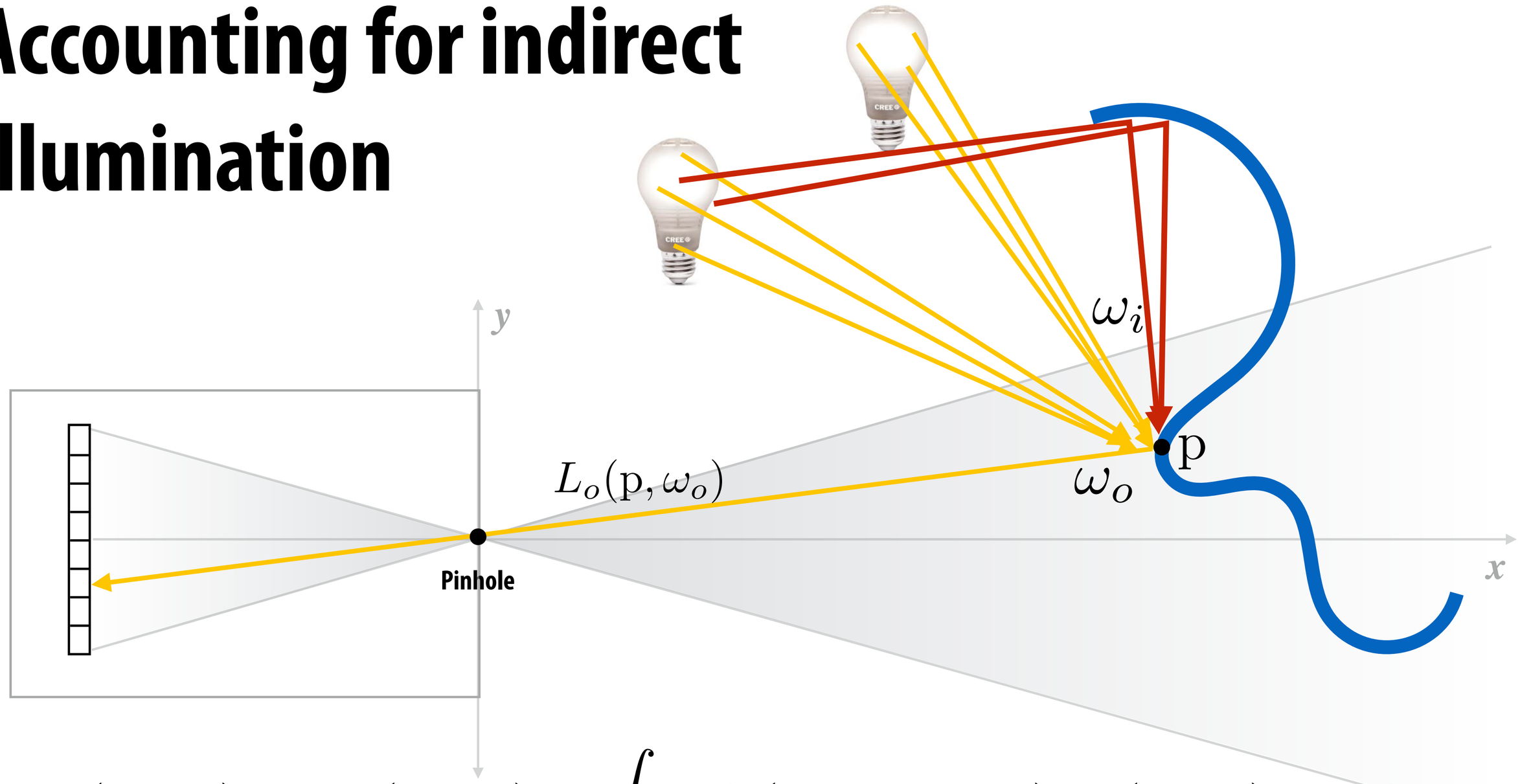


$$L_o(\mathrm{p}, \omega_o) = L_e(\mathrm{p}, \omega_o) + \int_{H^2} f_r(\mathrm{p}, \omega_i \to \omega_o)\, L_i(\mathrm{p}, \omega_i)\, \cos\theta_i \, \mathrm{d}\omega_i$$

Need to know incident radiance.

So far, have only computed incoming radiance from scene light sources.

Accounting for indirect illumination



$$L_o(\mathrm{p}, \omega_o) = L_e(\mathrm{p}, \omega_o) + \int_{H^2} f_r(\mathrm{p}, \omega_i \to \omega_o)\, L_i(\mathrm{p}, \omega_i)\, \cos\theta_i \, \mathrm{d}\omega_i$$

Incoming light energy from direction ω_i may be due to light reflected off another surface in the scene (not an emitter)

Path tracing: indirect illumination

$$\int_{H^2} f_r(\omega_i \to \omega_o)\, L_{o,i}(tr(\mathrm{p}, \omega_i), -\omega_i)\, \cos\theta_i \, \mathrm{d}\omega_i$$

- **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**

Direct illumination

One-bounce global illumination

Two-bounce global illumination

Four-bounce global illumination

Eight-bounce global illumination

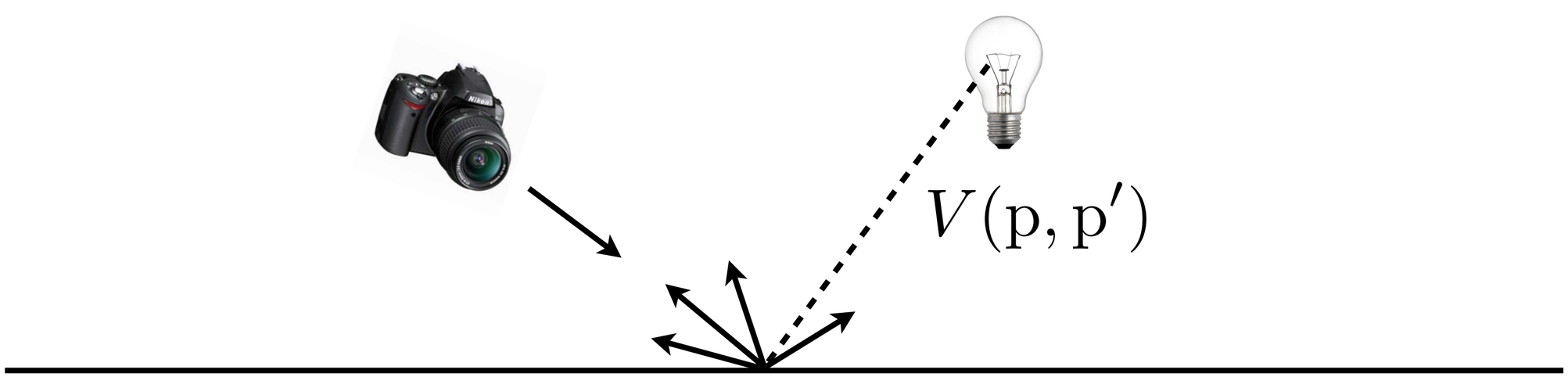
Sixteen-bounce global illumination

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:**

$$L = \frac{f_r(\omega_i \to \omega_o)\, L_i(\omega_i)\, V(\mathrm{p}, \mathrm{p}')\, \cos\theta_i}{p(\omega_i)}$$



Russian roulette

$$L = \frac{f_r(\omega_i \to \omega_o)\, L_i(\omega_i)\, V(\mathrm{p}, \mathrm{p}')\, \cos\theta_i}{p(\omega_i)}$$

$$\downarrow$$

$$L = \left[\frac{f_r(\omega_i \to \omega_o)\, L_i(\omega_i)\, \cos\theta_i}{p(\omega_i)}\right] V(\mathrm{p}, \mathrm{p}')$$

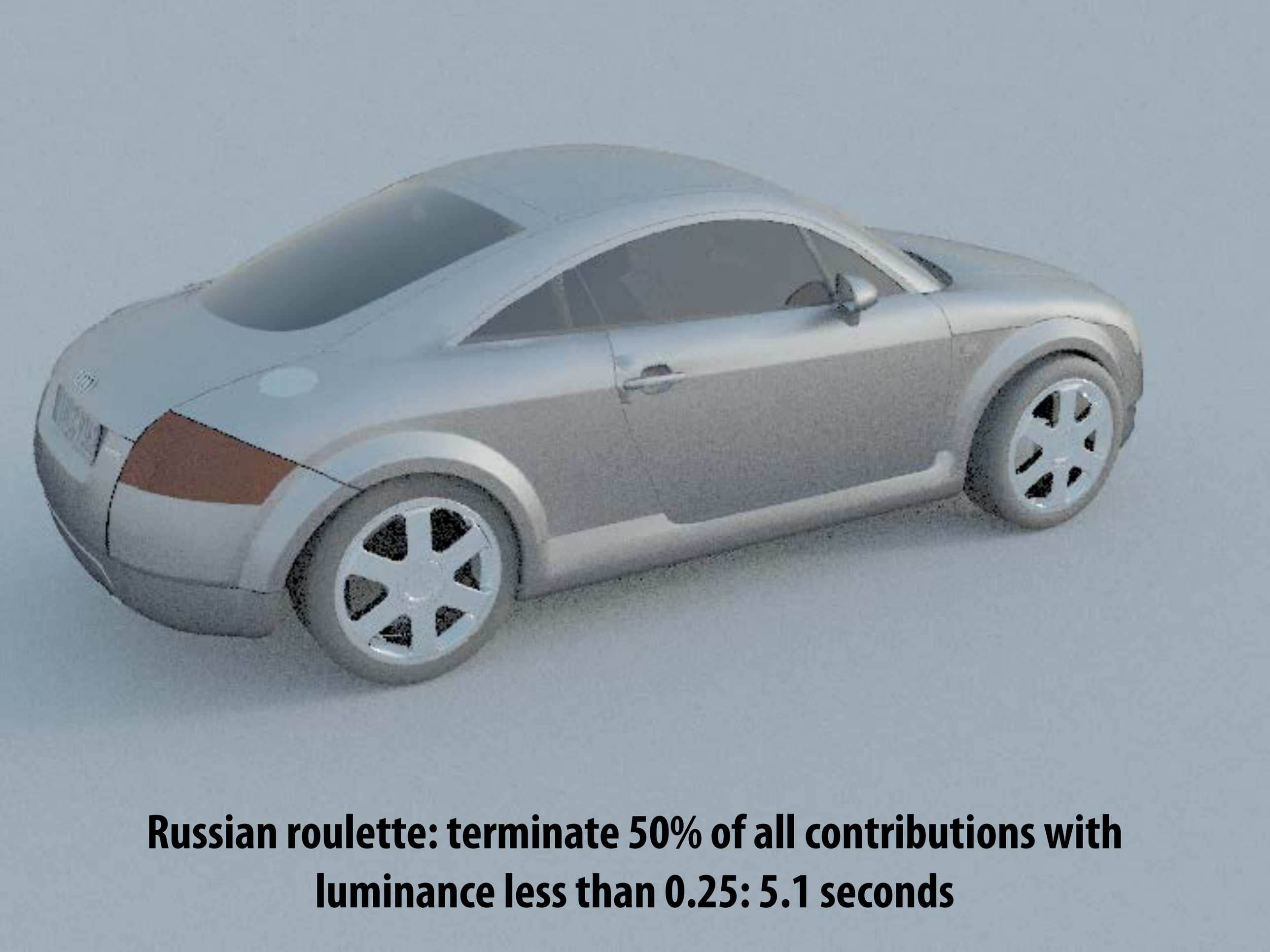
- **If tentative contribution (in brackets) is small, total contribution to the image will be small regardless of** $V(\mathrm{p}, \mathrm{p}')$
- **Ignoring low-contribution samples introduces systematic error**
  - **No longer converges to correct value!**
- **Instead, randomly discard low-contribution samples in a way that leaves estimator unbiased**

Russian roulette

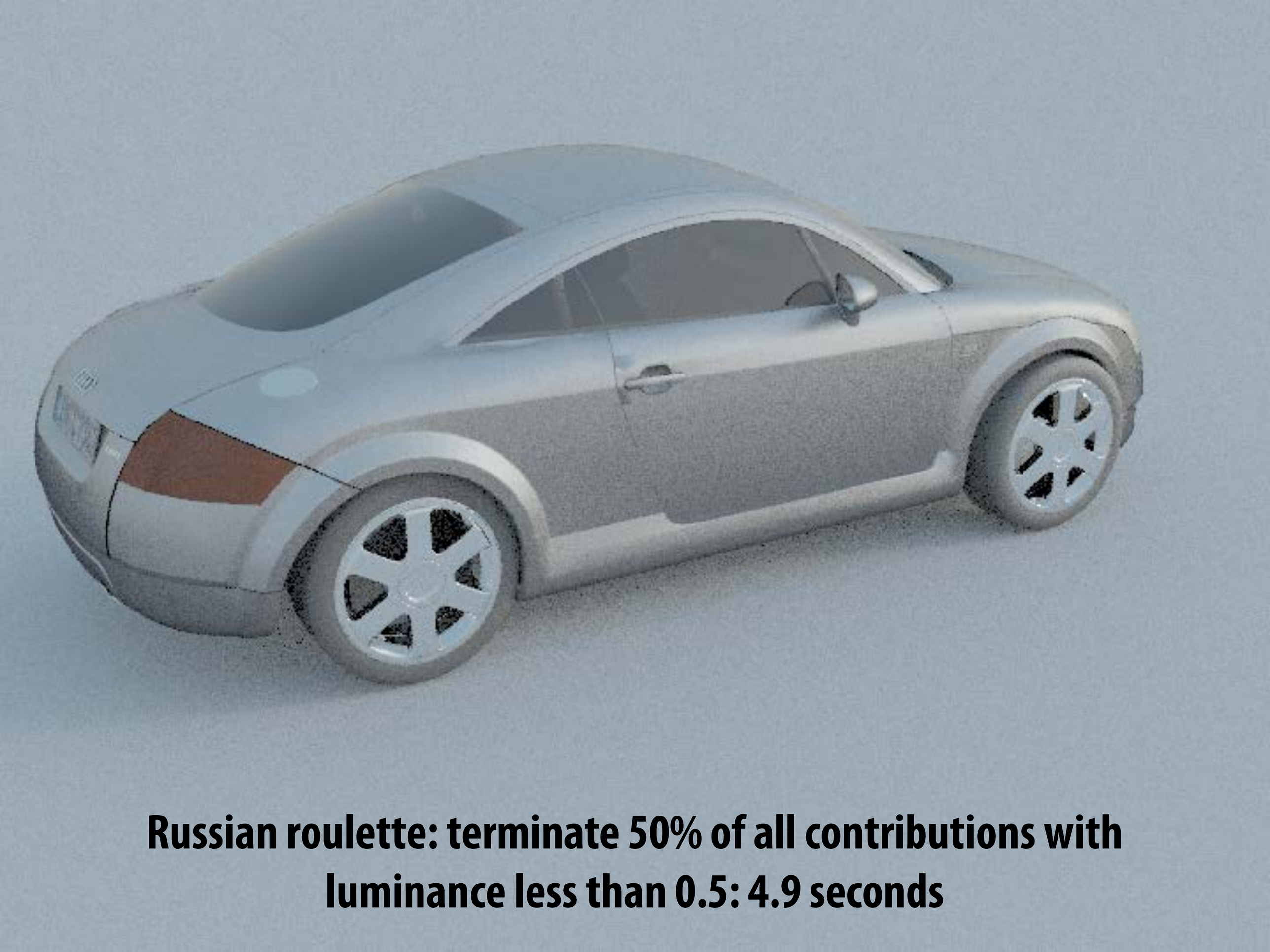
- **New estimator: evaluate original estimator with probability p_{rr}, reweight. Otherwise ignore.**
- **Same *expected value* as original estimator:**

$$p_{\mathrm{rr}} E\left[\frac{X}{p_{\mathrm{rr}}}\right] + E[(1-p_{\mathrm{rr}})0] = E[X]$$

No Russian roulette: 6.4 seconds

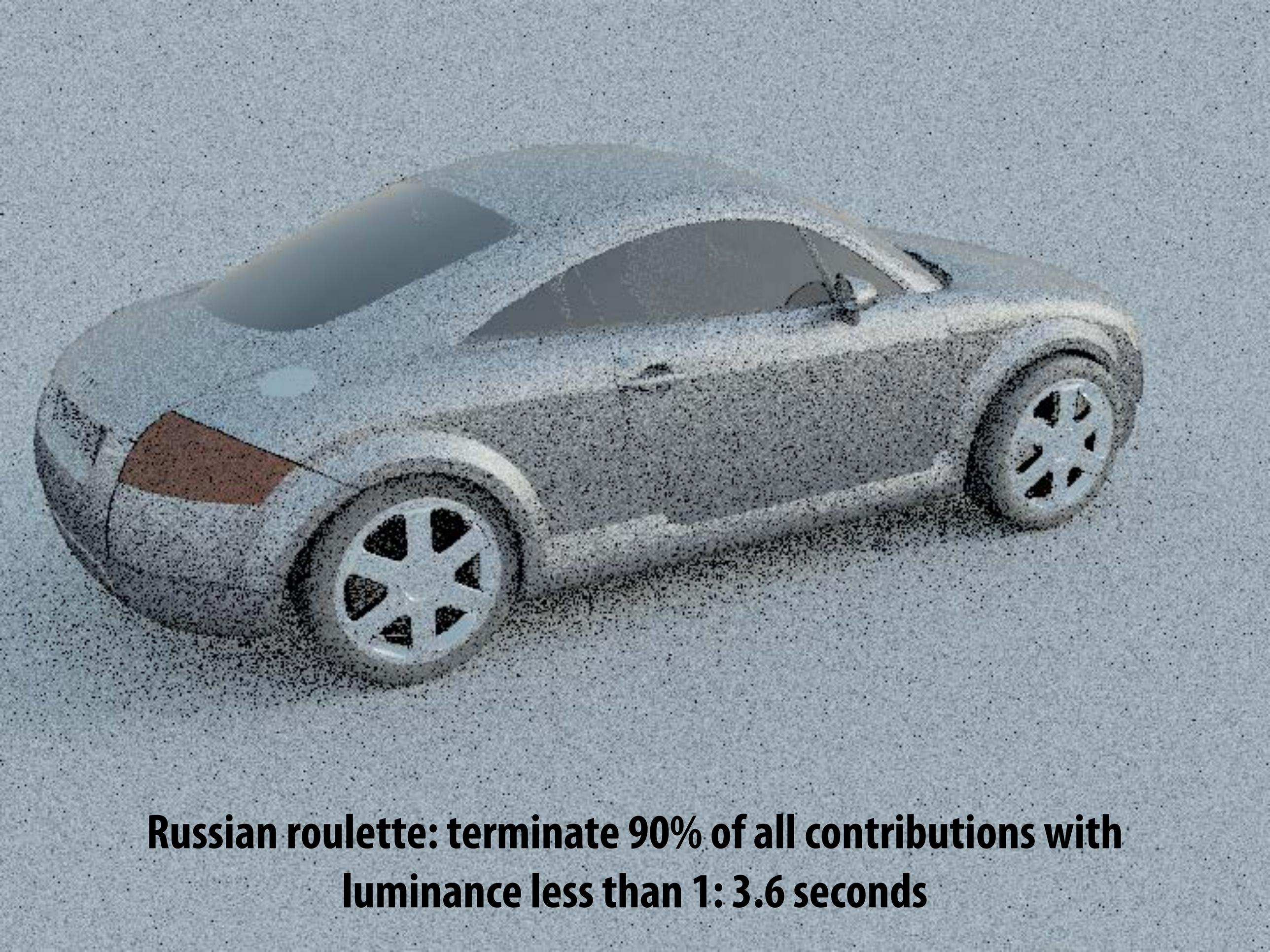


Russian roulette: terminate 50% of all contributions with luminance less than 0.25: 5.1 seconds



Russian roulette: terminate 50% of all contributions with luminance less than 0.5: 4.9 seconds

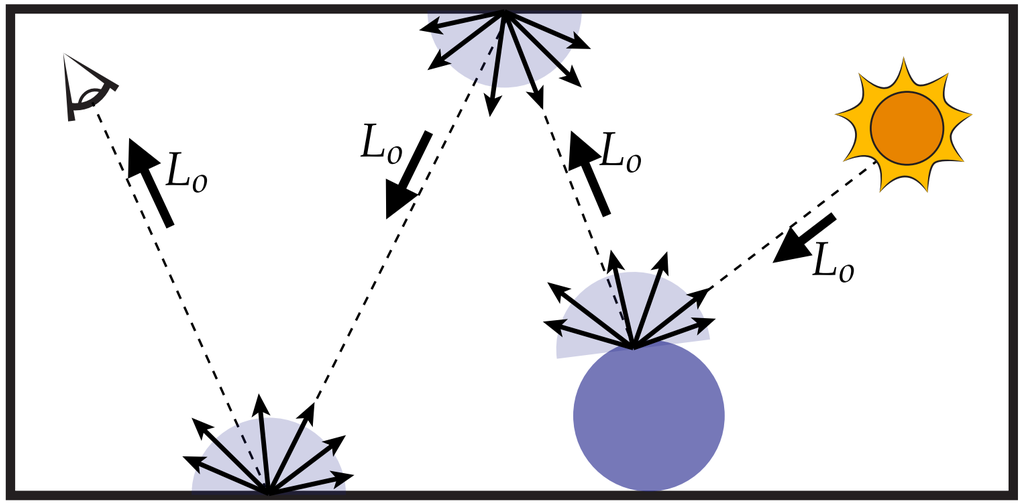
Russian roulette: terminate 90% of all contributions with luminance less than 0.125: 4.8 seconds



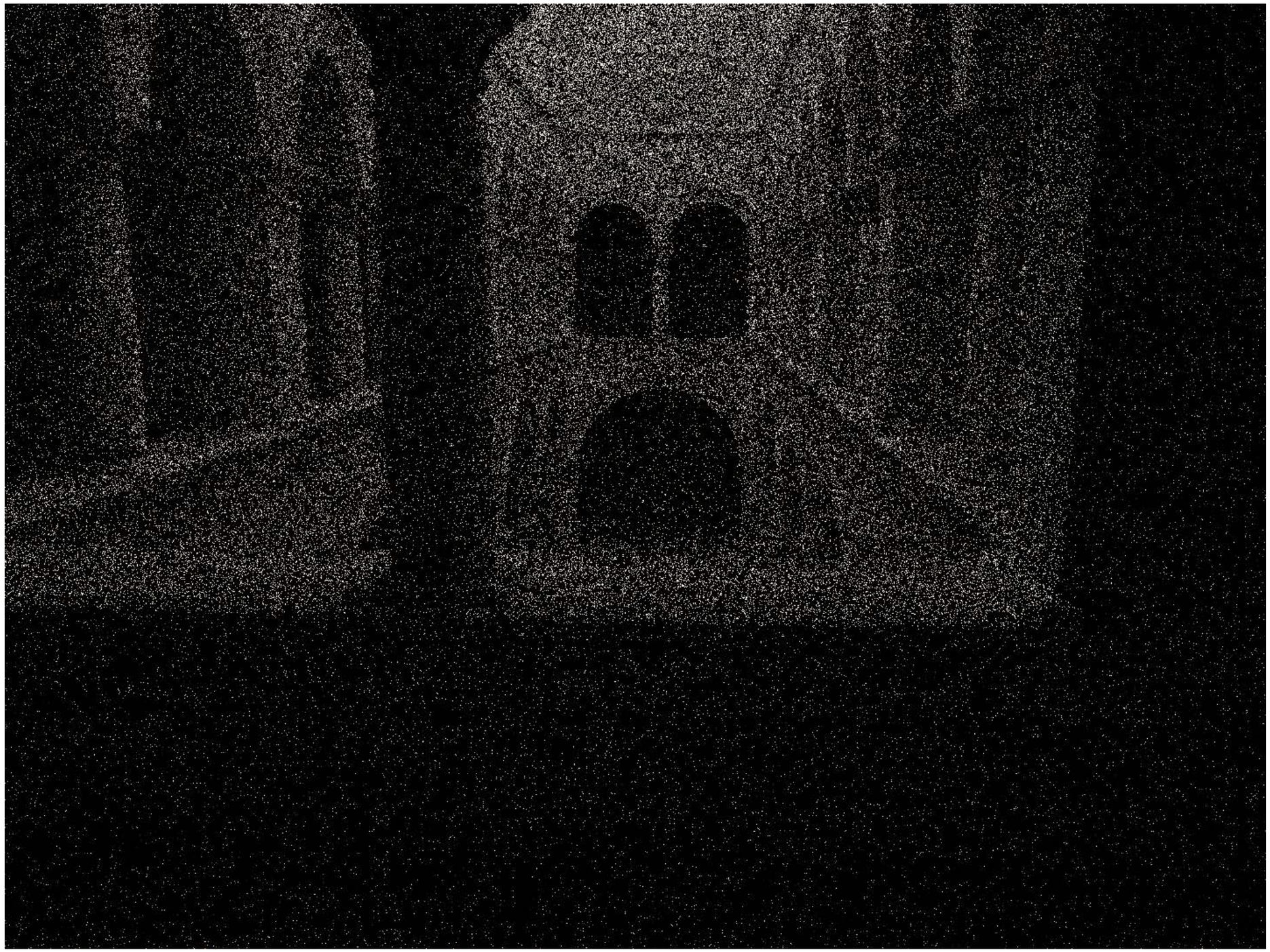
Russian roulette: terminate 90% of all contributions with luminance less than 1: 3.6 seconds

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!**



$$L_o(\mathbf{p},\omega_o) = L_e(\mathbf{p},\omega_o) + \int_{\mathcal{H}^2} f_r(\mathbf{p},\omega_i \to \omega_o) L_i(\mathbf{p},\omega_i) \cos\theta \, d\omega_i$$

Next time:

- **Variance reduction—how do we get the most out of our samples?**