北京航空航天大學

人工智能学院（人工智能研究院）

# Computer Graphics
# Lecture 6: Visibility and Shading


潘成伟 (Chengwei Pan)

Email: pancw@buaa.edu.cn

Office: Room B1021, New Main Building

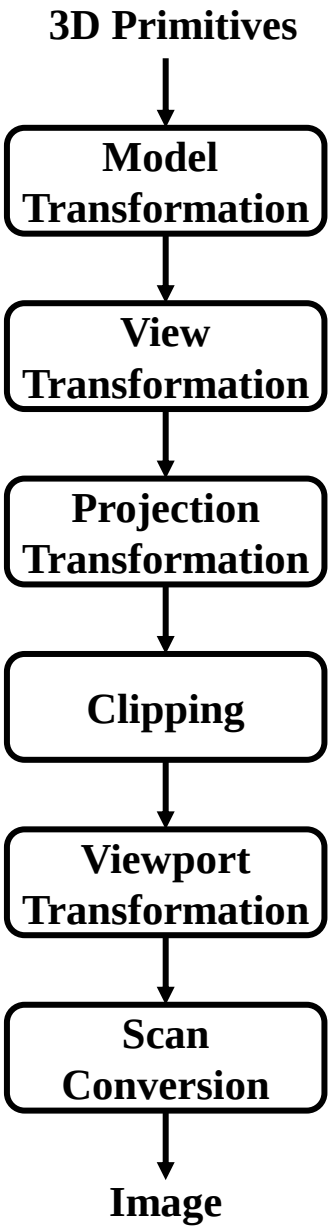
北京航空航天大学，人工智能学院

School of Artificial Intelligence, Beihang University

# Last Lecture

- Clipping
  - Line Clipping
  - Polygon Clipping
- Rasterization
  - Line Rasterization
  - Triangle Rasterization
- Aliasing and AntiAliasing

# Rasterization Pipeline




**Transform into world coordinate system**

**Transform into camera coordinate system**

**3D to 2D, Perspective projection**

**Clip primitives outside camera's view … in clip space**

**Transform into image coordinate system**

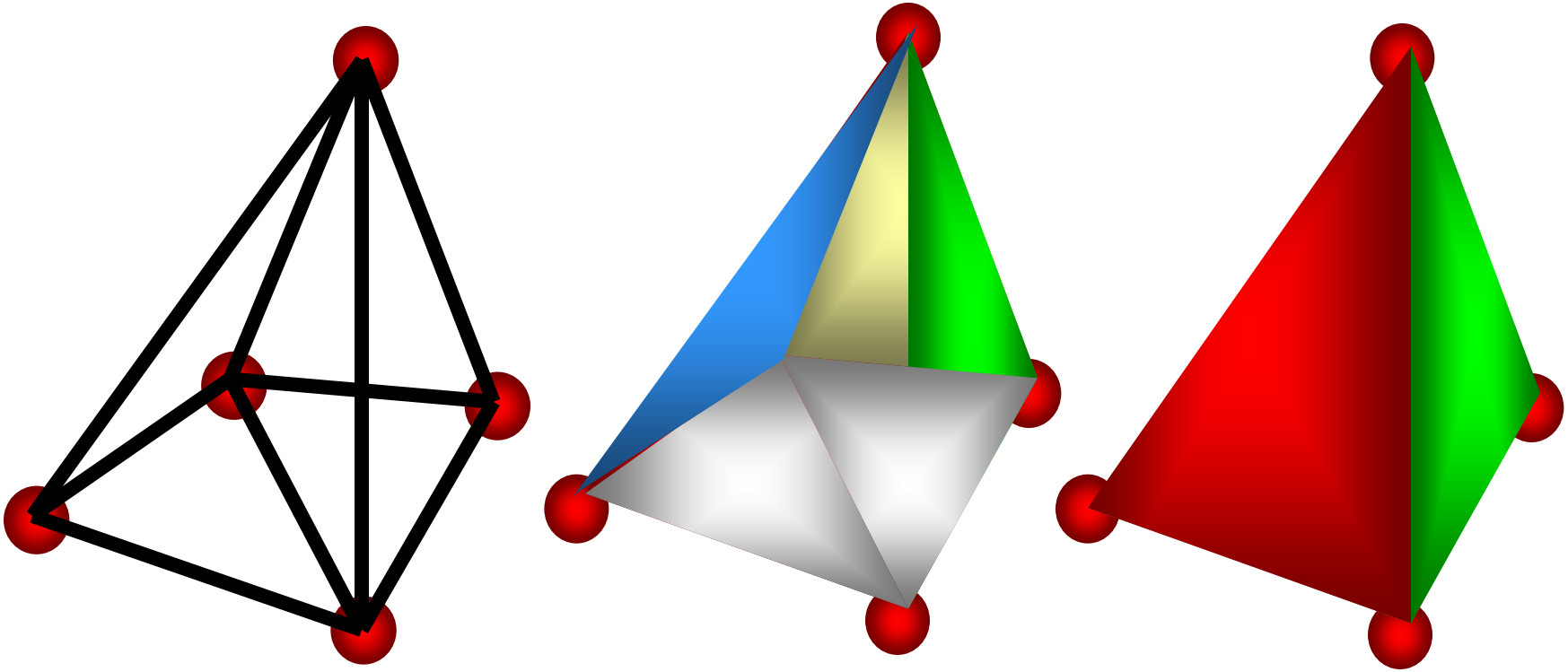
**Draw pixels**

# This Lecture

- Visibility
  - Back face culling
  - Hidden Surface Removal
    - Object space
    - Image space
- Shading
  - Phong shading model
  - Shading frequency

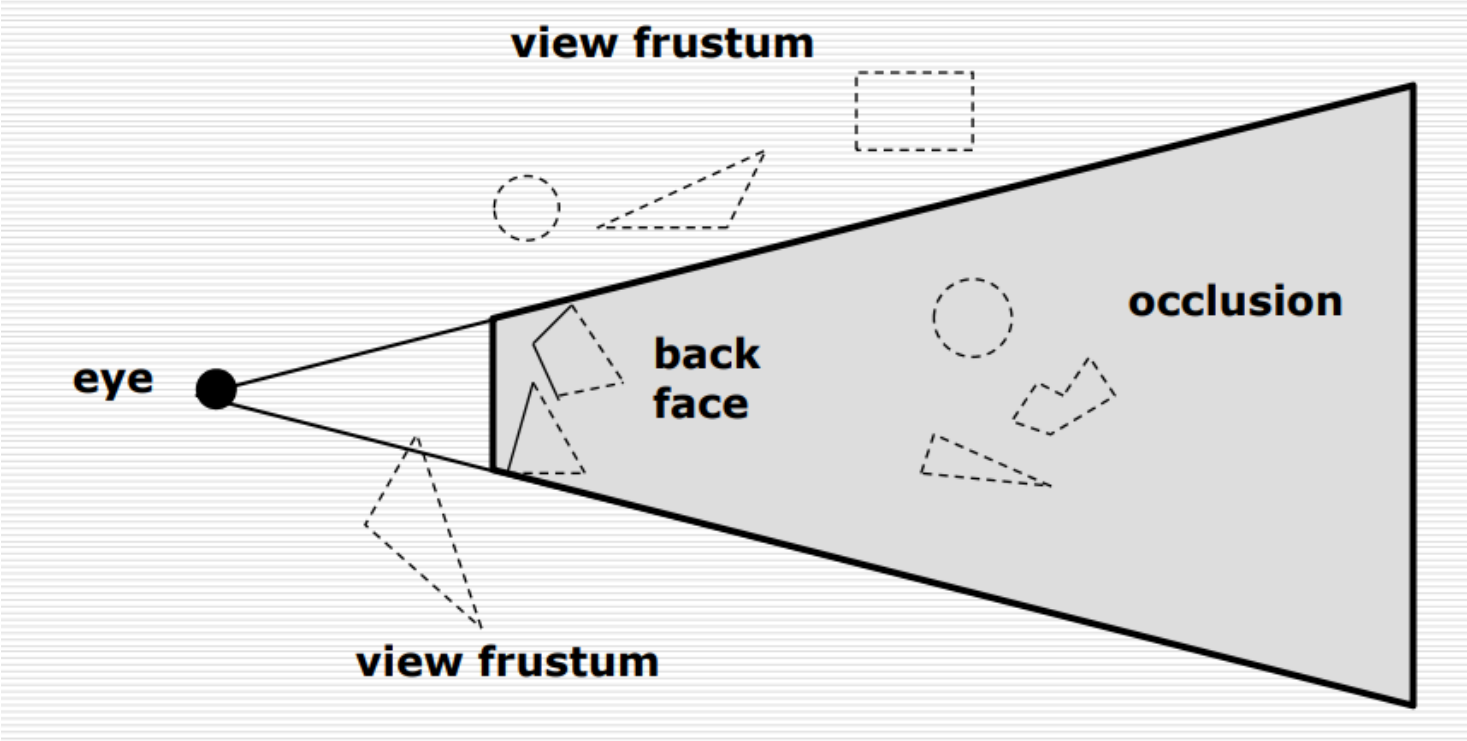
# Goal of Visible Surface Determination

- Draw only the surfaces(triangles) that are visible, given a view point and a view direction

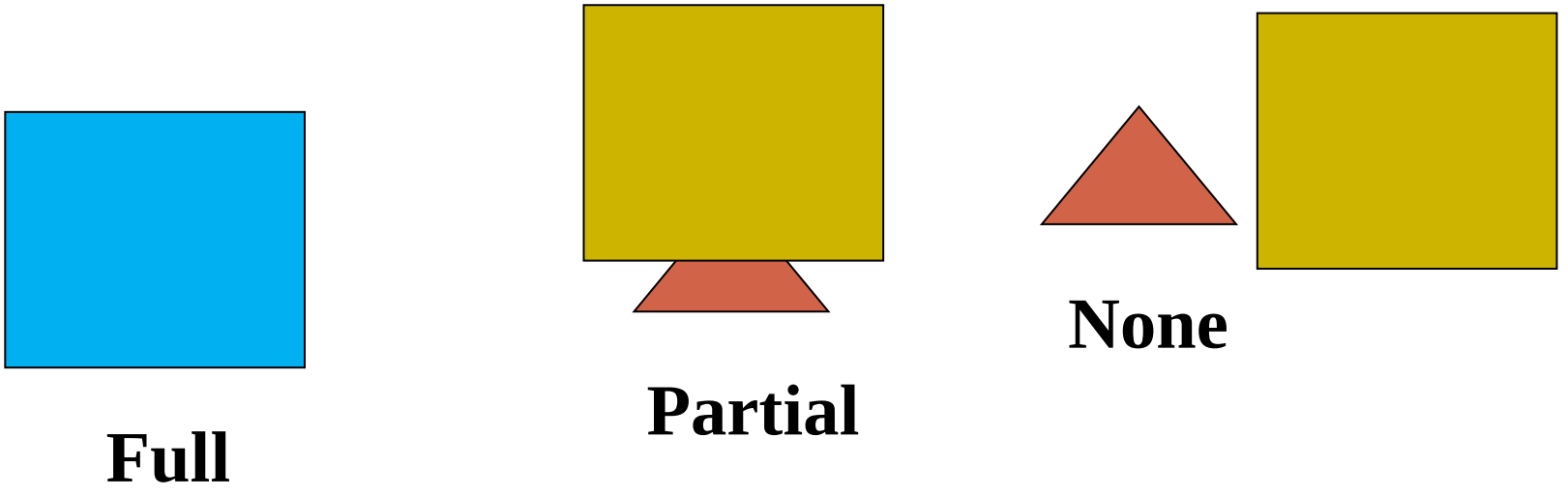


# Visible Determination

- view frustum (clipping)
- back face culling
- Occlusion (hidden surface removal)



# Occlusion: Full, Partial, None




- The rectangle is closer than the triangle
- Should appear in front of the triangle

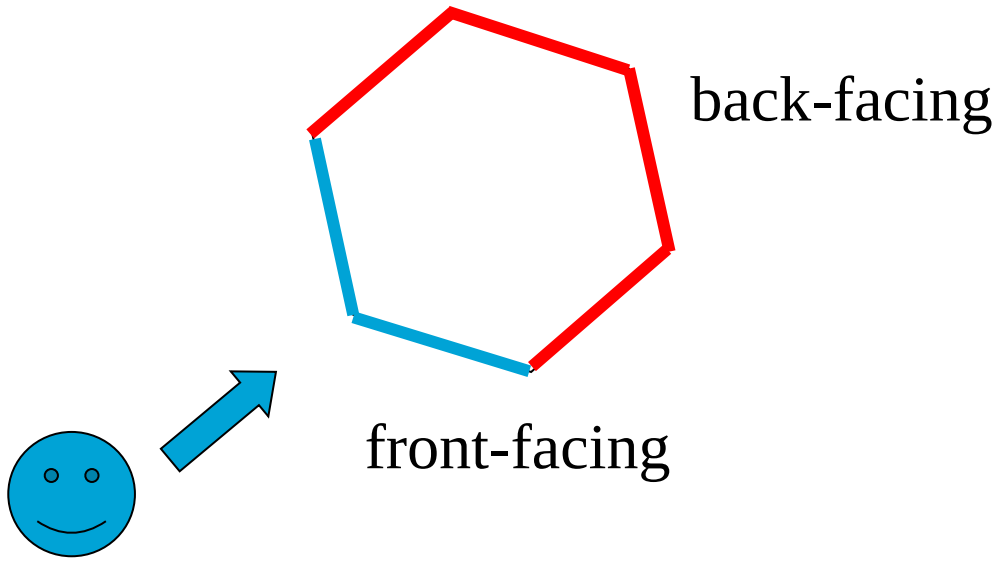
# This Lecture

- Visibility
  - Back face culling
  - Hidden Surface Removal
    - Object space
    - Image space

- Shading
  - Phong shading model
  - Shading frequency

# Back Face Culling

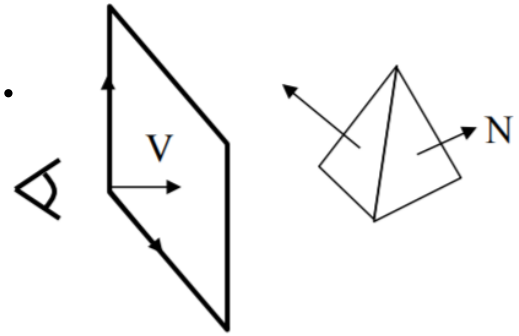
- Avoid drawing polygons facing away from the viewer
  - Front-facing polygons occlude these polygons in a closed polyhedron
- Test if a polygon is front- or back-facing



# Back Face Culling

- Of all the faces of an object, only the ones facing the viewer need to be rendered to the screen.
  - front-facing polygon points towards the viewer
  - back-facing polygon points away from the viewer

- Back face culling
  - The scalar product V·N must be positive.
  - The Z coordinate of N in eye space is negative.
  - Looking if the polygon vertices lies clockwise.

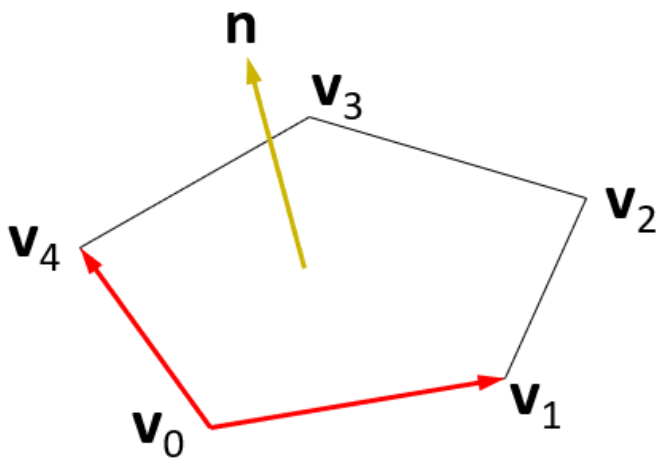


# Polygon Normal

- Let polygon vertices $\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_{n-1}$ be in counterclockwise order and co-planar
- Calculate normal with cross product:

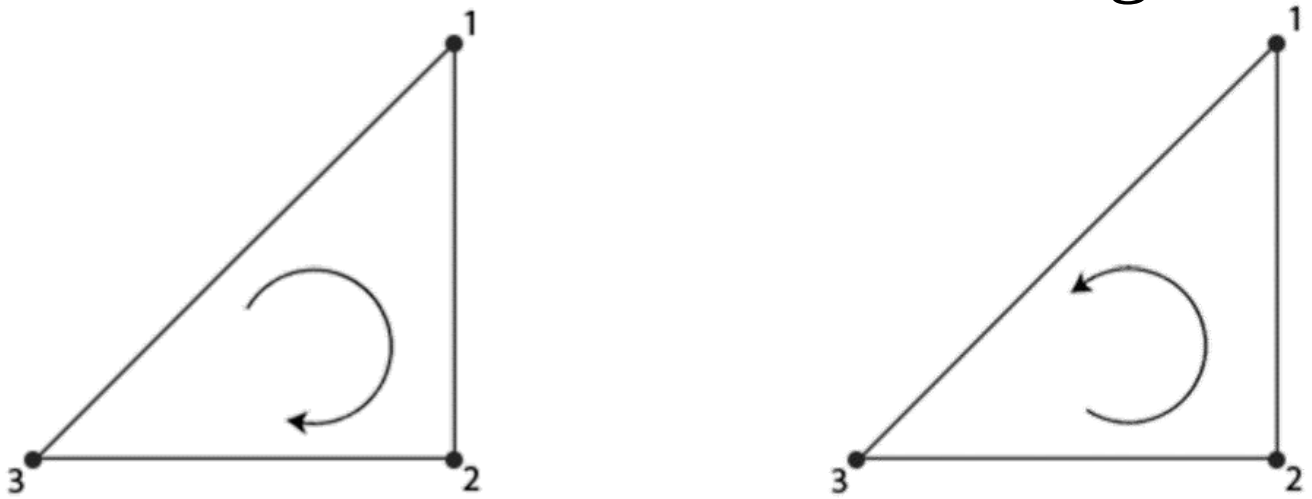
$$\mathbf{n} = (\mathbf{v}_1 - \mathbf{v}_0) \times (\mathbf{v}_{n-1} - \mathbf{v}_0)$$

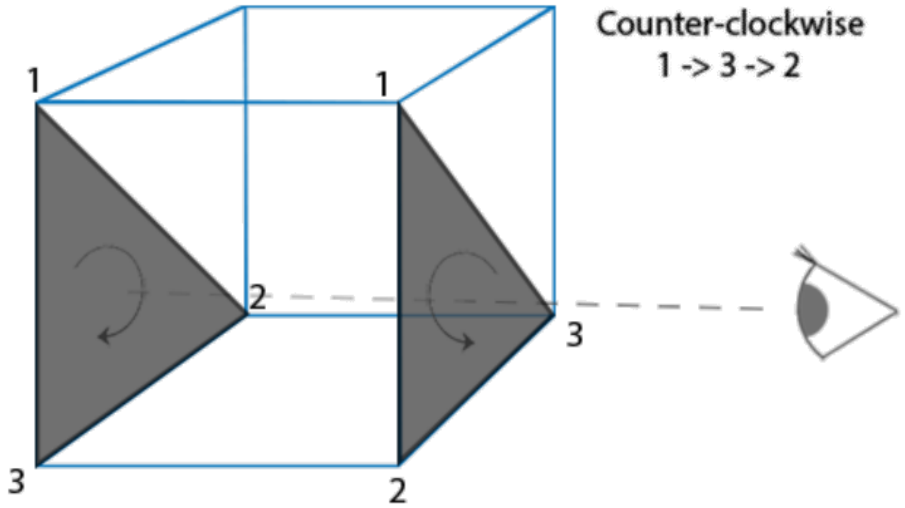
- Normalize to unit vector with $\mathbf{n} / \|\mathbf{n}\|$



# Normal Direction

- Vertices counterclockwise ⇒ Front-facing
- Vertices clockwise ⇒ Back-facing



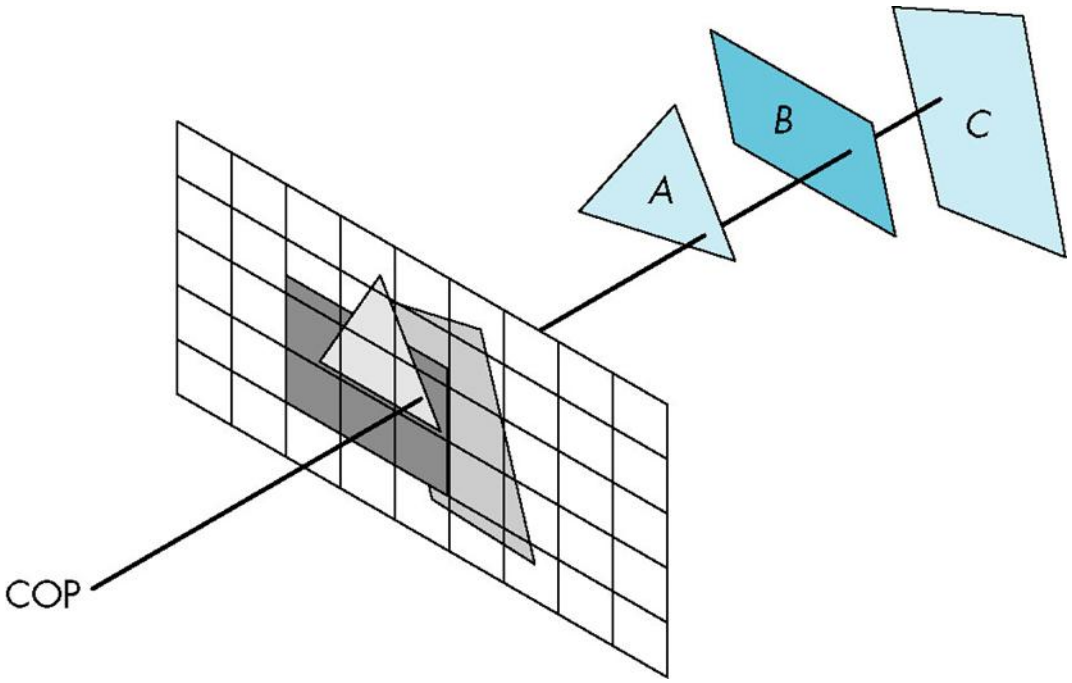



# This Lecture

- Visibility
  - Back face culling
  - Hidden Surface Removal
    - Object space
    - Image space
- Shading
  - Phong shading model
  - Shading frequency

# Visibility

- How do we ensure that closer polygons overwrite further ones in general?

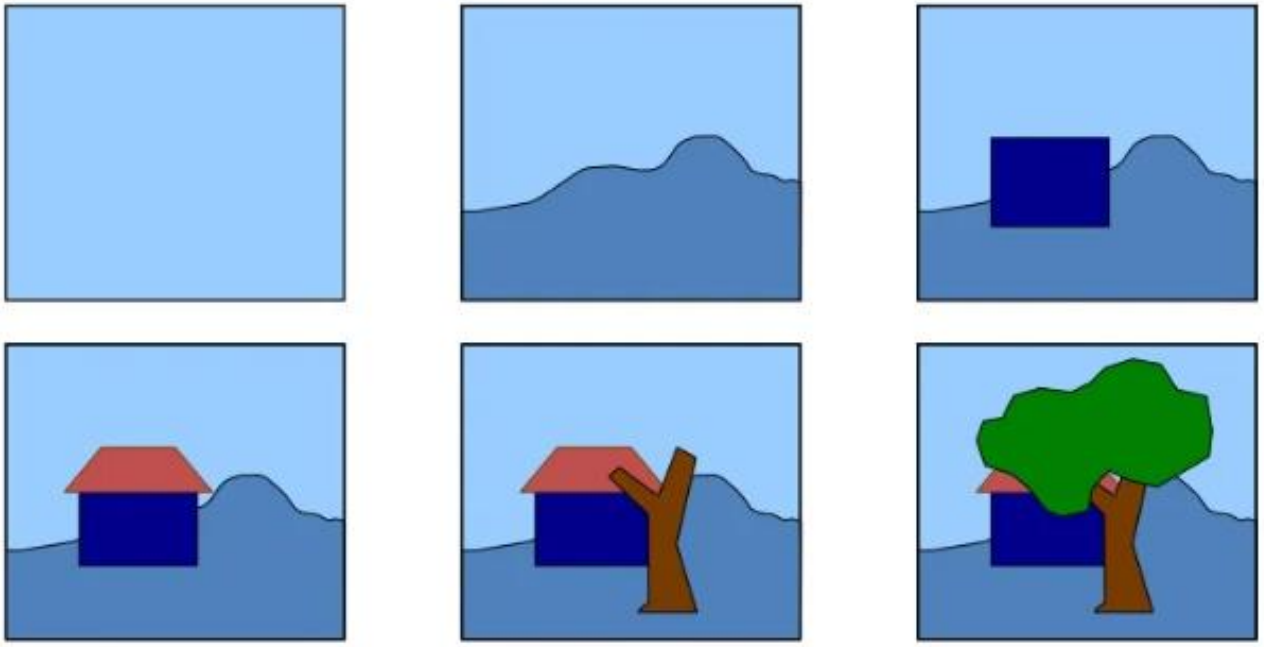


# Hidden Surface Removal

- Object space
  - Applied before vertices are mapped to pixels
  - Painter's algorithm, BSP tree
- Image space
  - Decide on visibility at each pixel position
  - Z-buffering

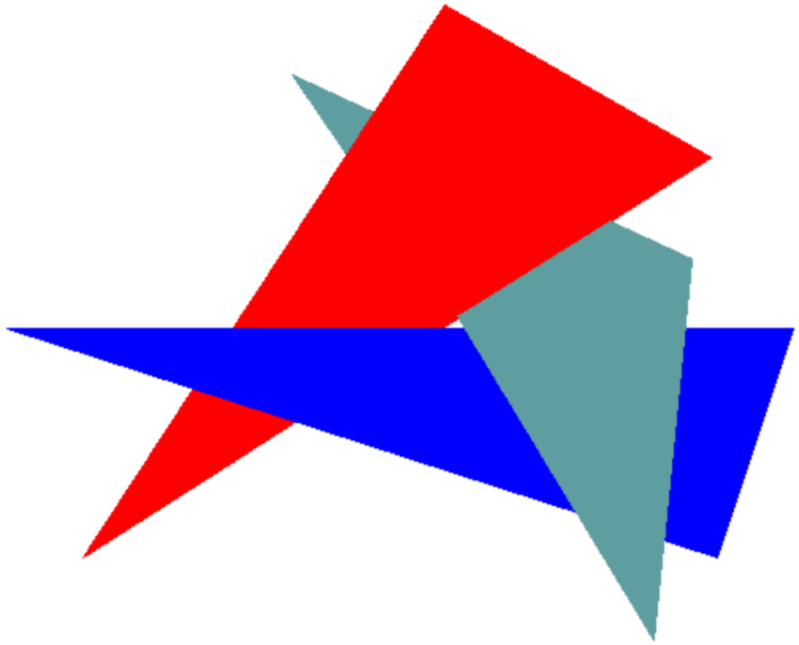
# Painter's Algorithm

- Draw surfaces from back to front
  - Sort polygons by their depth (z value)
  - Draw objects in order (farthest to closest)
  - Close objects paint over the top of farther away objects



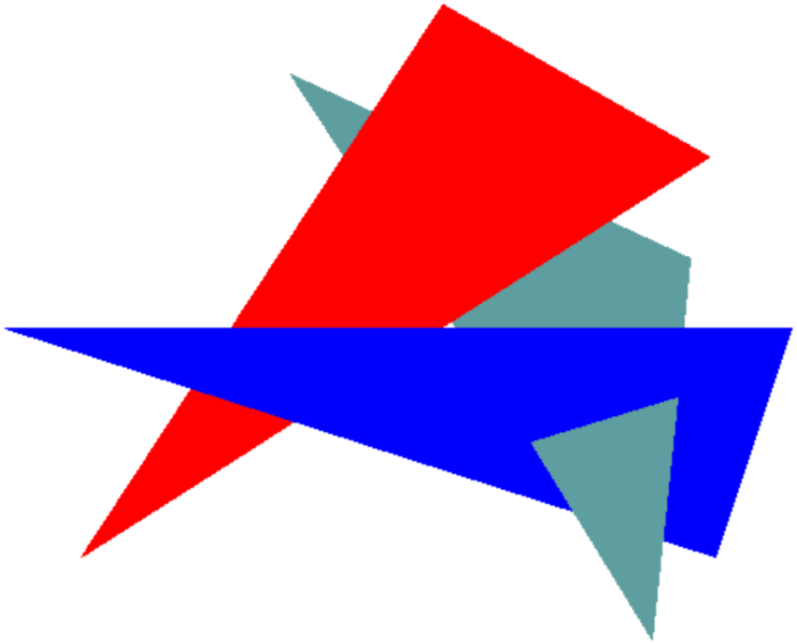
# Painter's algorithm

- Main issue is determining the order
- Doesn't always work



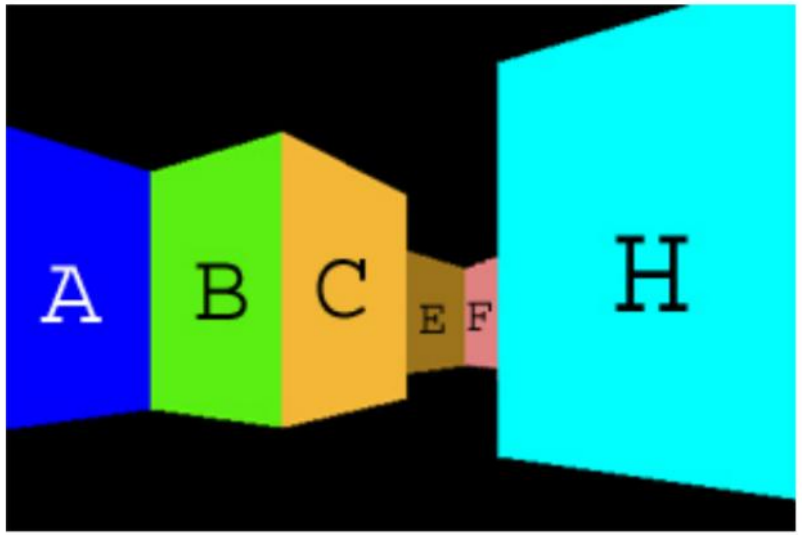
# Painter's algorithm

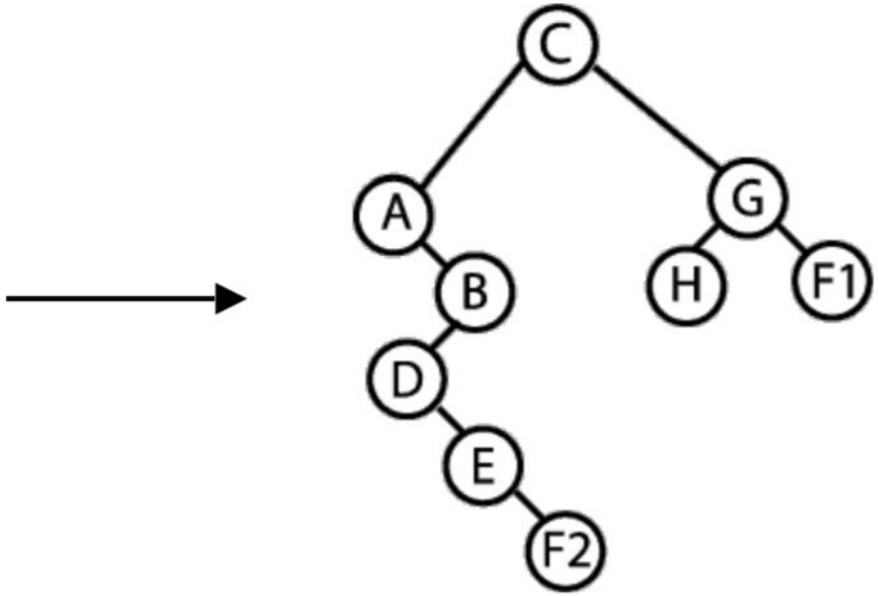
- Another problem case
- Need to segment the triangles so that they can be sorted



# BSP trees

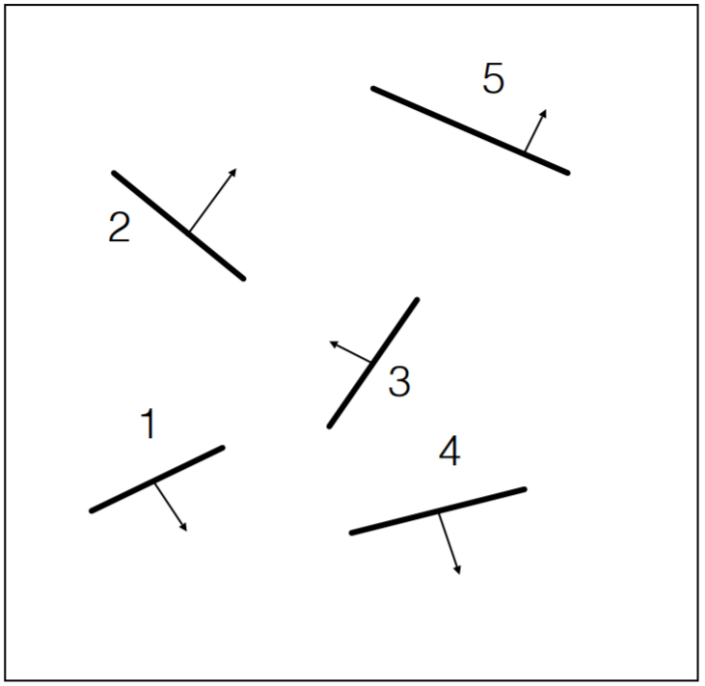
- Binary space partitioning tree
- Represents the scene with a tree
- Scene is drawn by traversing the tree
- Suitable for rendering static scenes



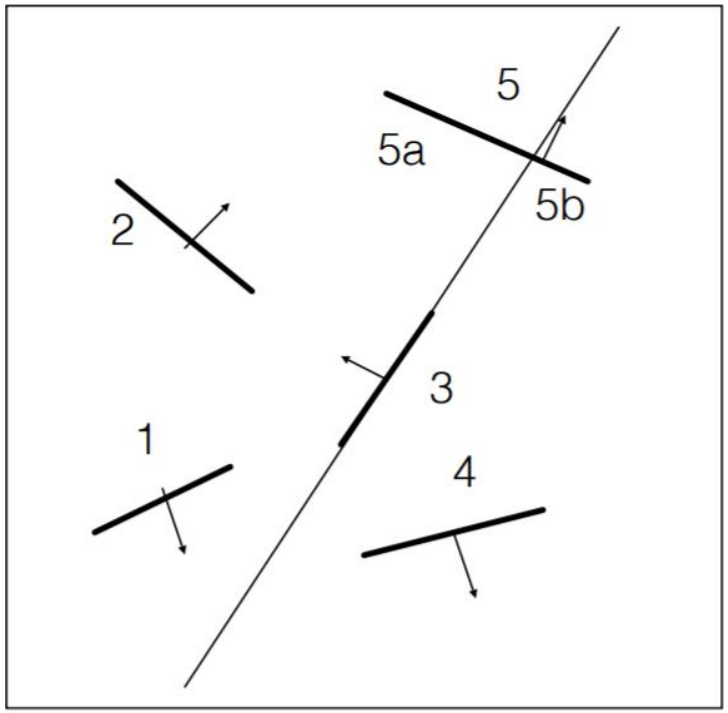
# BSP Tress

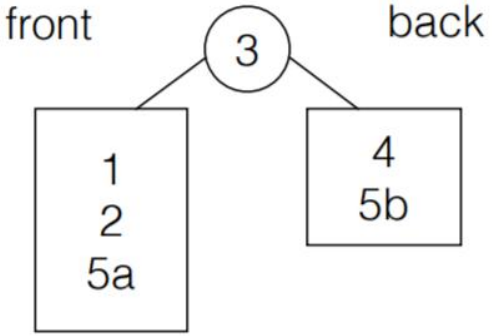
- Choose polygon arbitrarily



# BSP Tress

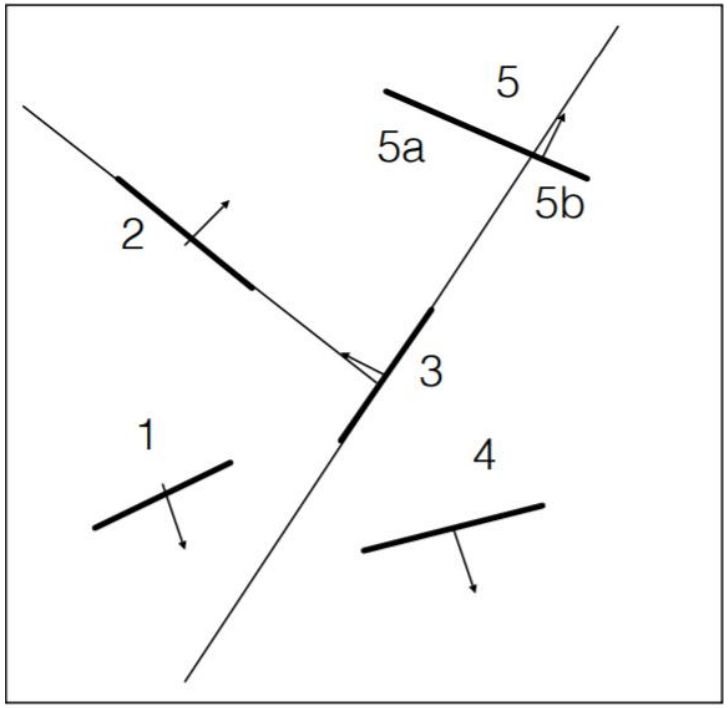
- Choose polygon arbitrarily
- Divide scene into front (relative to normal) and back half-spaces
- Split any polygon lying on both sides

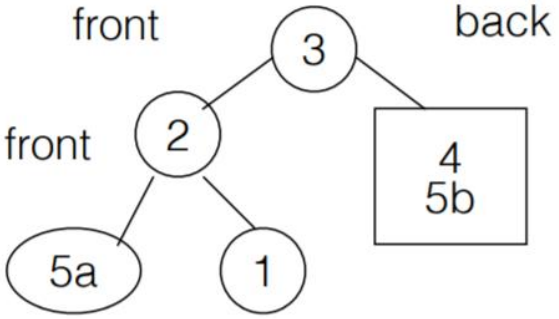


# BSP Tress

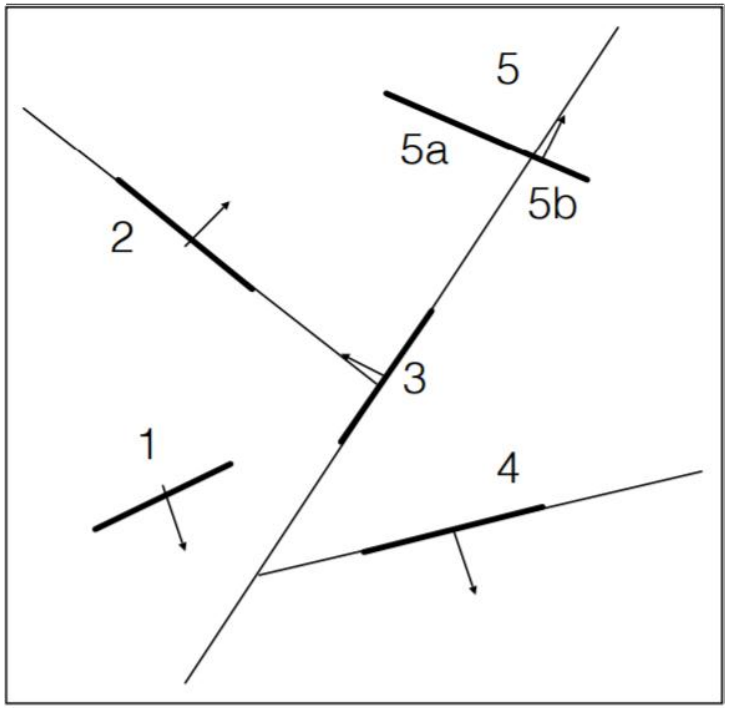
- Choose polygon arbitrarily
- Divide scene into front (relative to normal) and back half-spaces
- Split any polygon lying on both sides
- Choose a polygon from each side – split scene again.

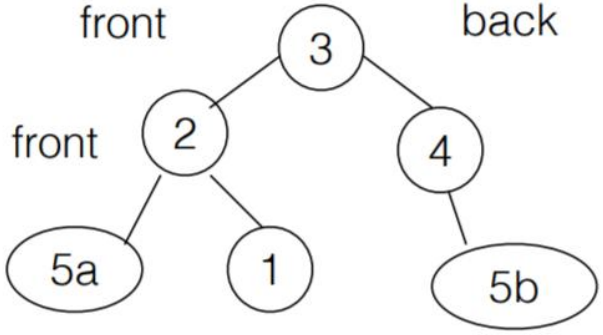


# BSP Tress

- Choose polygon arbitrarily
- Divide scene into front (relative to normal) and back half-spaces
- Split any polygon lying on both sides
- Choose a polygon from each side – split scene again.
- Recursively divide each side until each node contains only 1 polygon.

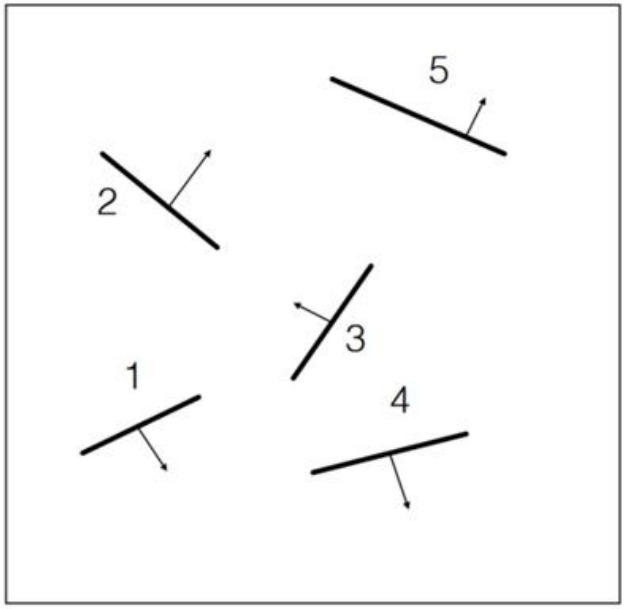


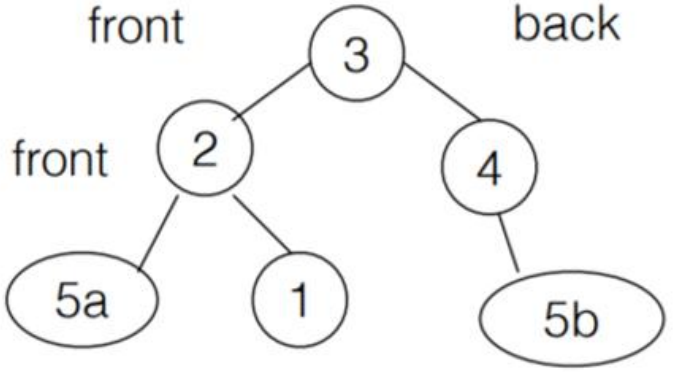



# Displaying a BSP tree

- The tree can be traversed to yield an ordering of the polygons for an arbitrary viewpoint

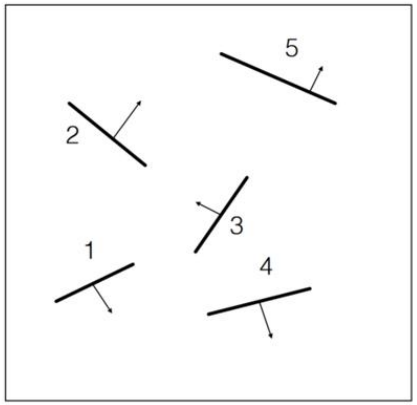
- Back to front using the painter's algorithm

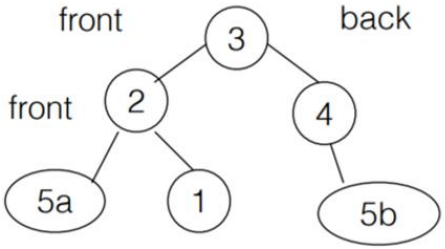


# Displaying a BSP tree: back to front

- Start at root polygon.
  - If viewer is in front half-space, draw polygons behind root first, then the root polygon, then polygons in front.
  - If viewer is in back half-space, draw polygons in front of root first, then the root polygon, then polygons behind.
  - Recursively descend the tree.




- Always drawing the opposite side from the viewer first

# This Lecture

- Visibility
  - Back face culling
  - Hidden Surface Removal
    - Object space
    - Image space

- Shading
  - Phong shading model
  - Shading frequency

# Z-buffer

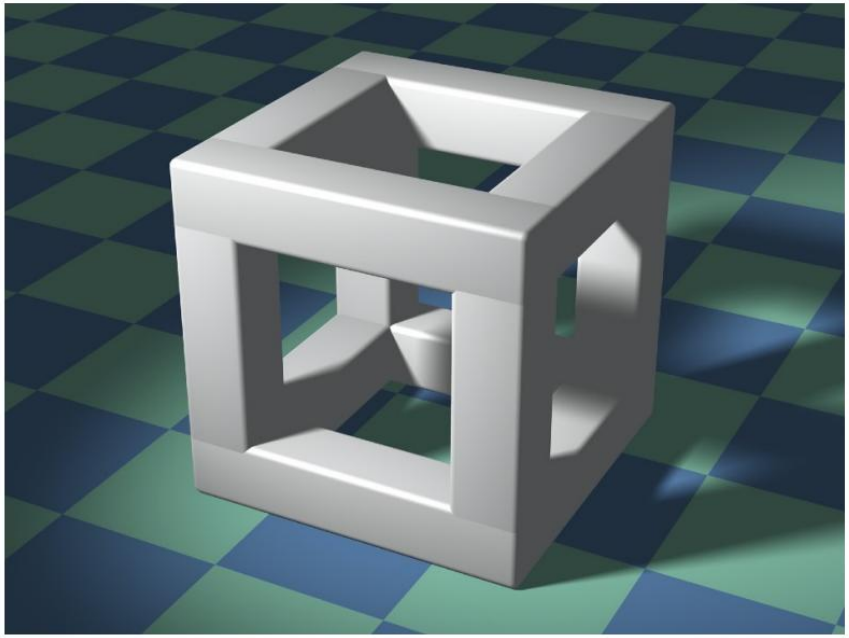
- An image-based method applied during rasterization
- Standard approach used in graphics hardware and libraries
- Easy to implement in hardware

# Z-buffer

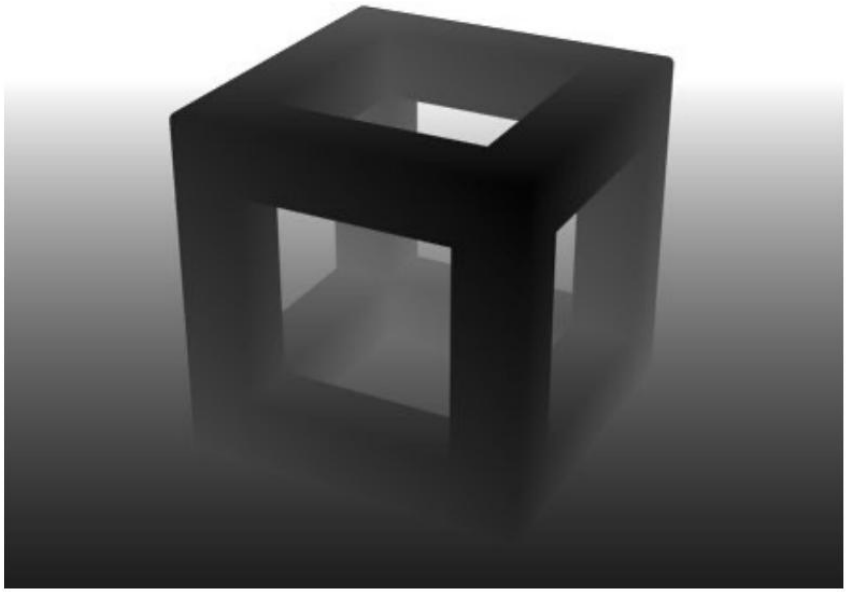
- Idea
  - Store current min. z-value for each sample (pixel)
  - Needs an additional buffer for depth values
    - Frame buffer stores color values
    - Depth buffer (z-buffer) stores depth

- Important: For simplicity we suppose
  - Z is always positive
  - Smaller z -> closer, larger z -> further

# Z-buffer example



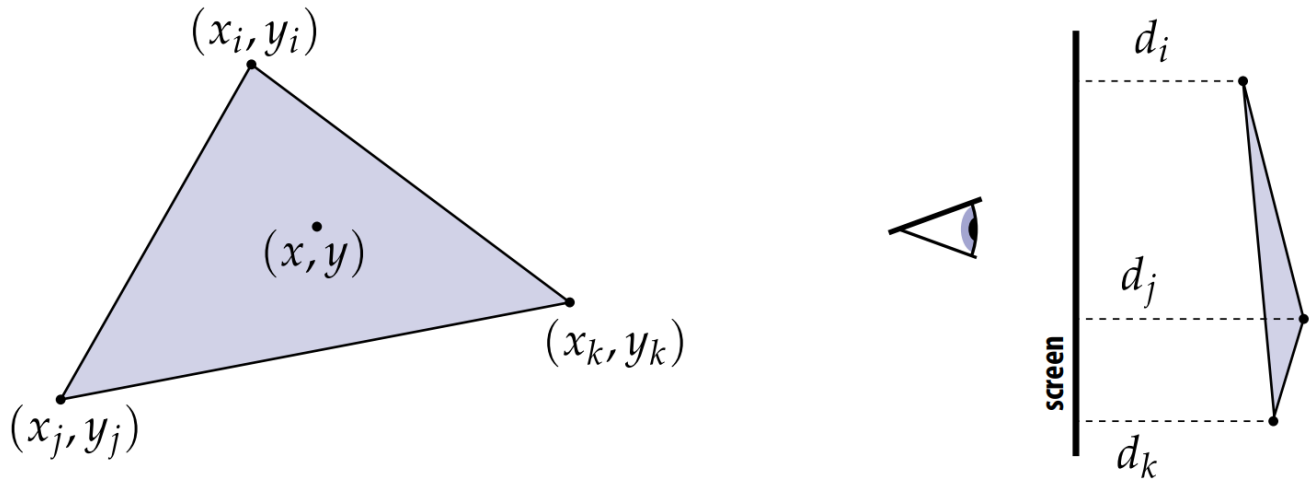
Rendering



Depth / Z buffer

# Sampling depth

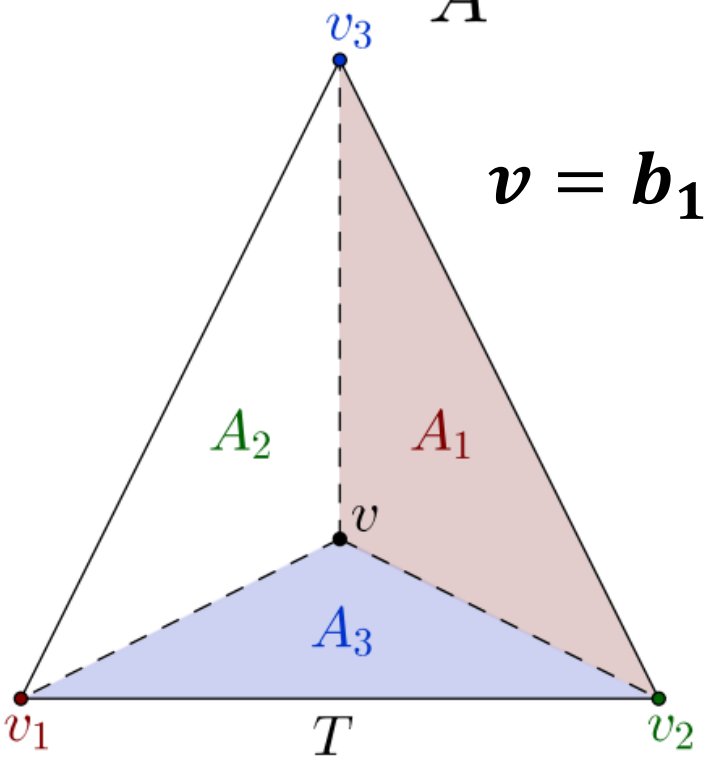
- Assume we have a triangle given by:
  - the projected 2D coordinates of each vertex
  - the depth of each vertex (i.e., distance from the viewer)




- compute the depth at a given sample point (x, y)
  - Interpolate it using barycentric coordinates

# Triangle Barycentric Coordinate

$$b_3 = \frac{A_3}{A}$$




$$\boldsymbol{v = b_1 v_1 + b_2 v_2 + b_3 v_3}$$

$$b_1 = \frac{A_1}{A} \qquad b_2 = \frac{A_2}{A}$$

$$A = A_1 + A_2 + A_3$$

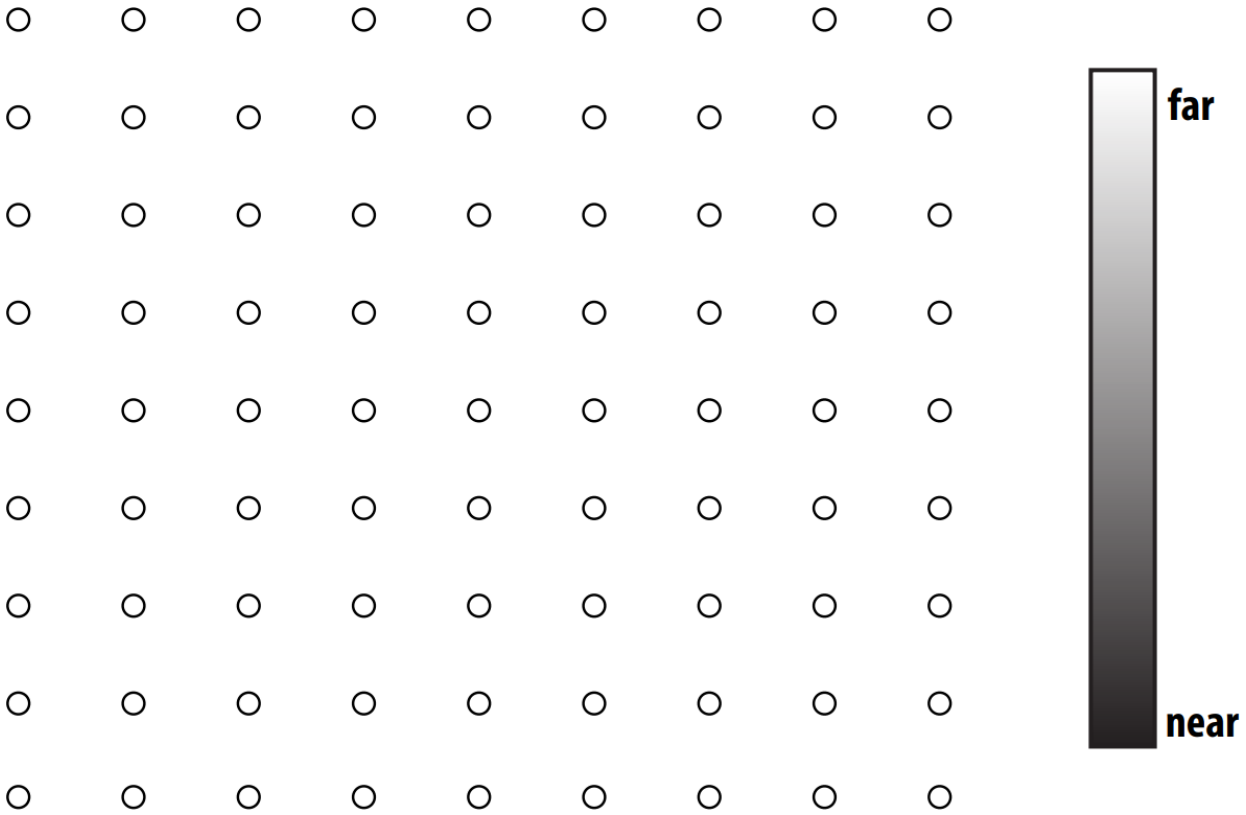
# Z-buffer Algorithm

- Initialize depth buffer to max. value
- During rasterization:

```
for (each triangle T)
    for (each sample (x,y,z) in T)
        if (z < zbuffer[x,y])                  // closest sample so far
            framebuffer[x,y] = rgb;            // update color
            zbuffer[x,y] = z;                  // update depth
        else
            ;                                  // do nothing, this sample is occluded
```

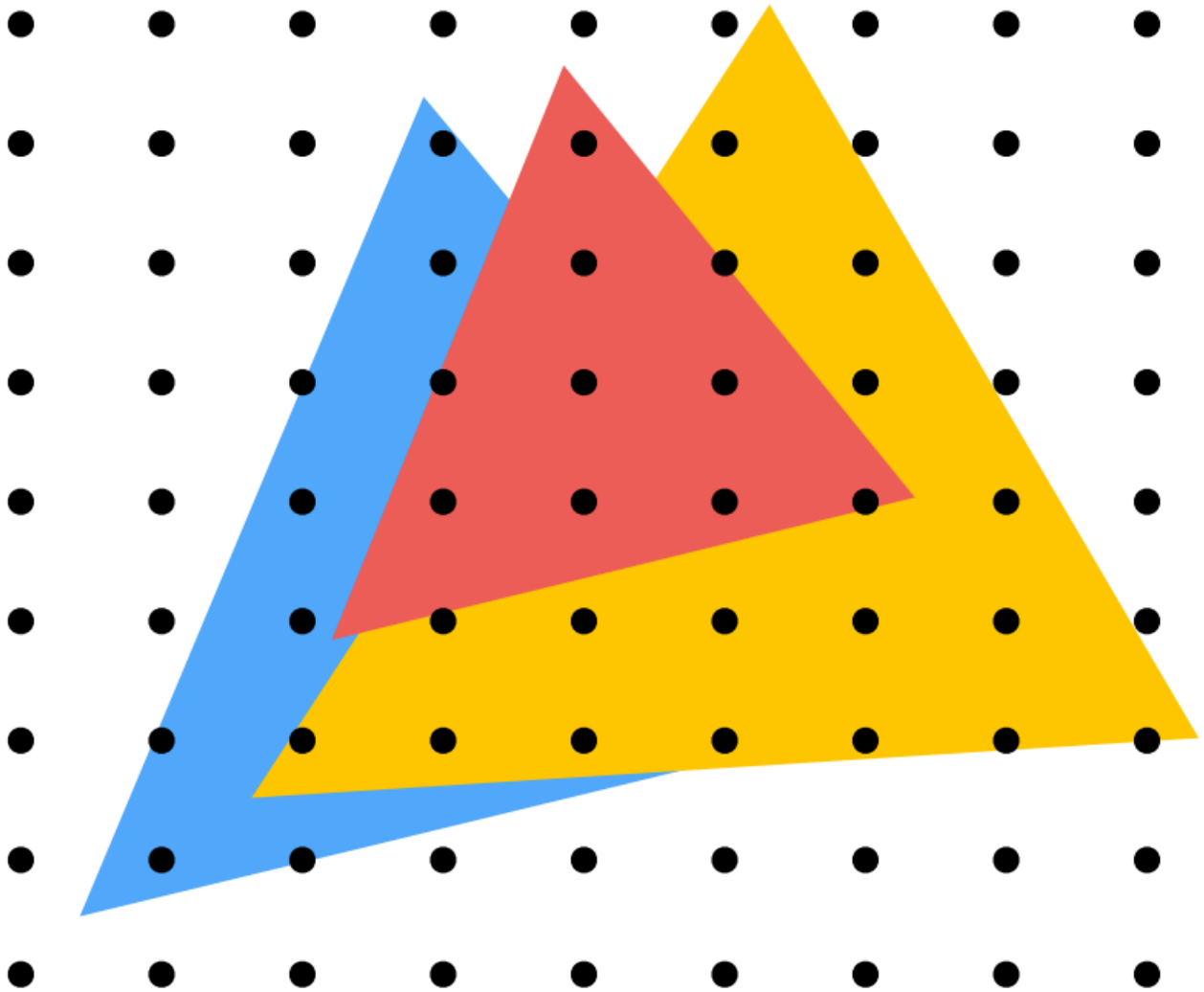
# Z-buffer

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




Initialize all depth buffer values to “infinity” (max value)

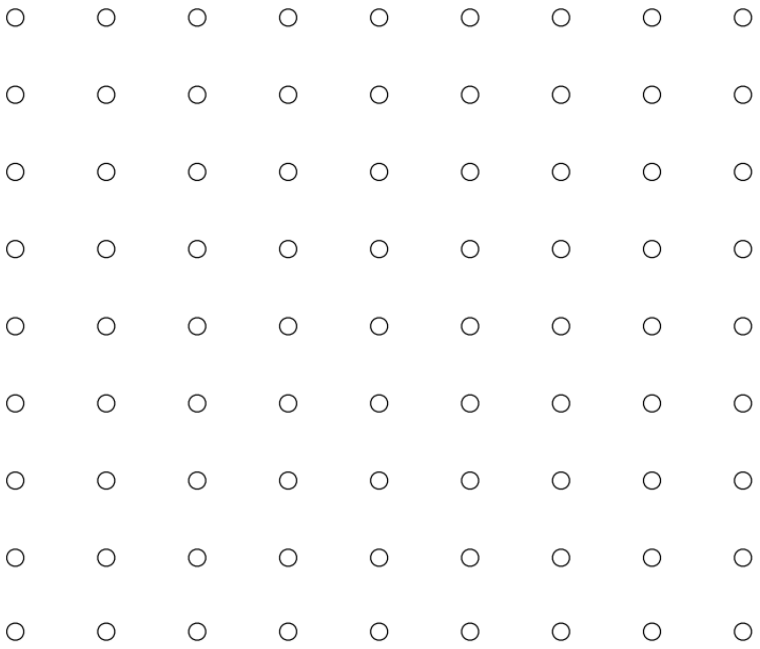
# Example: rendering three opaque triangles



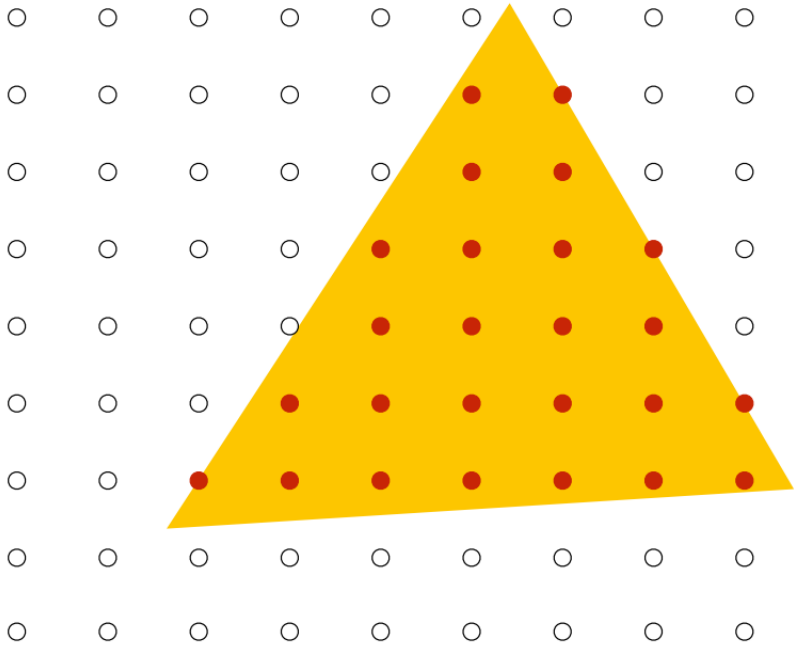
# Occlusion using the Z-buffer

**Processing yellow triangle:**

**depth = 0.5**

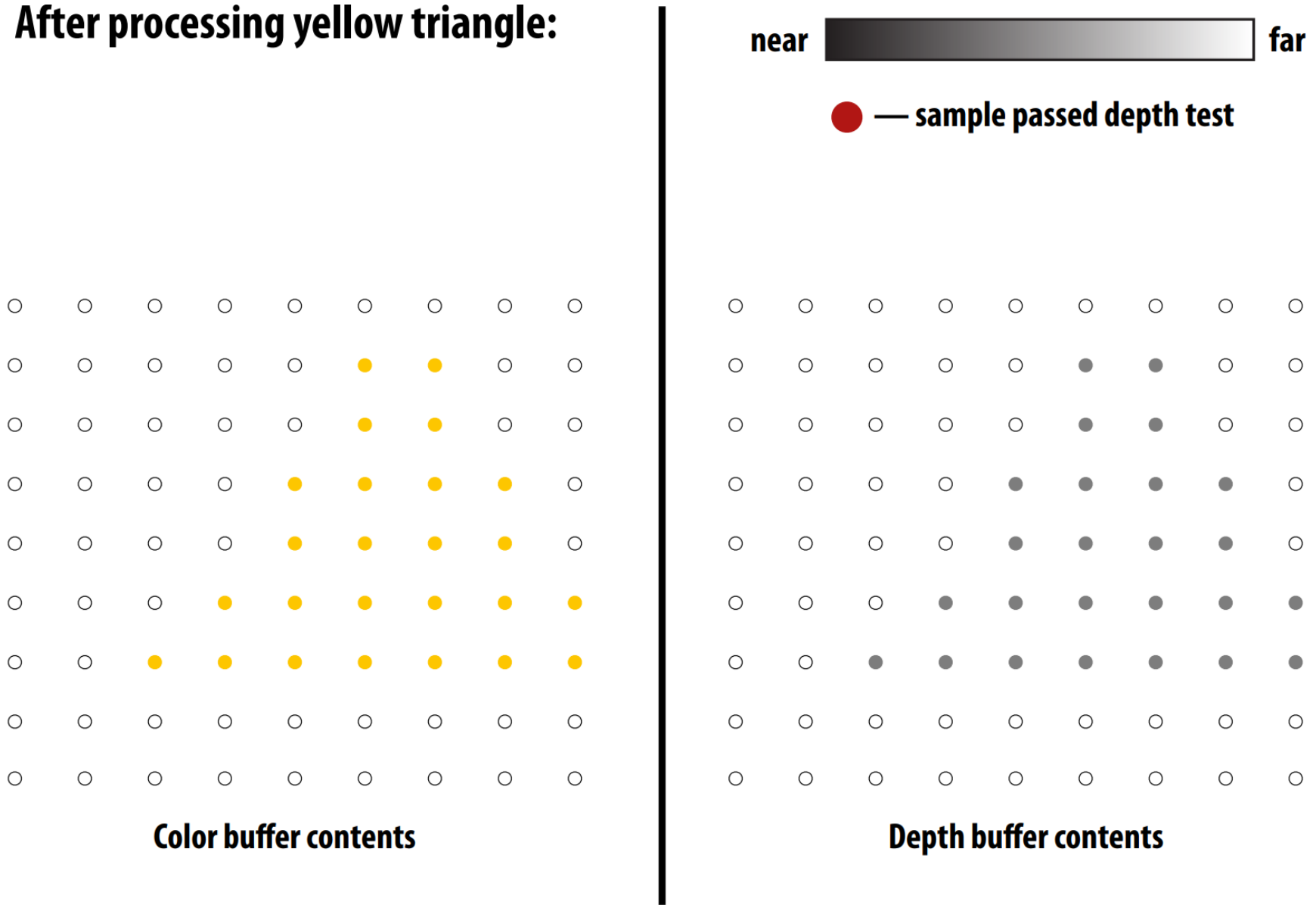


**Color buffer contents**

**Depth buffer contents**

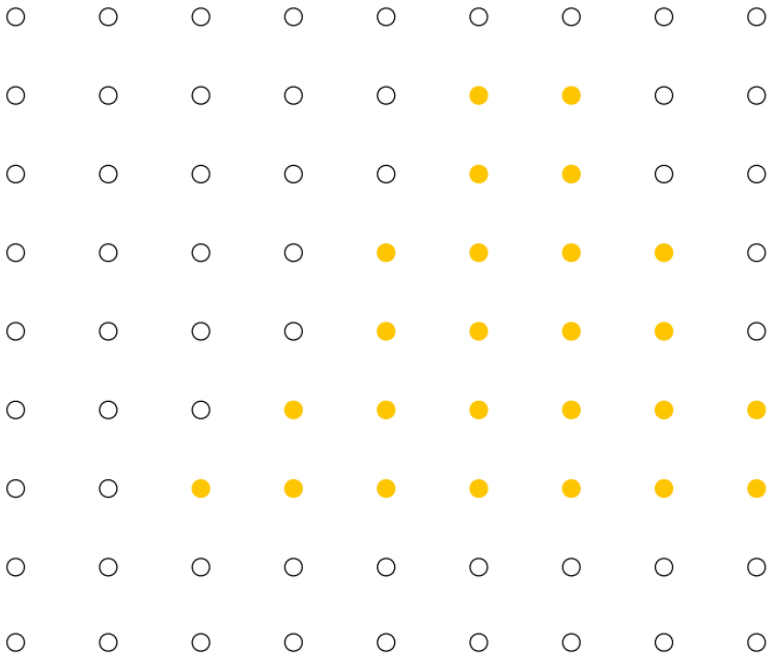
# Occlusion using the Z-buffer



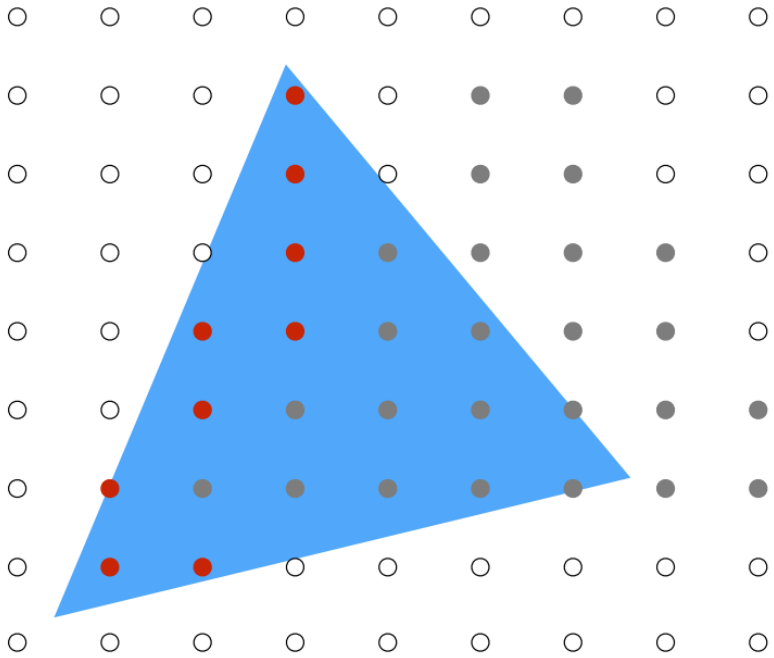
# Occlusion using the Z-buffer

**Processing blue triangle:**

**depth = 0.75**

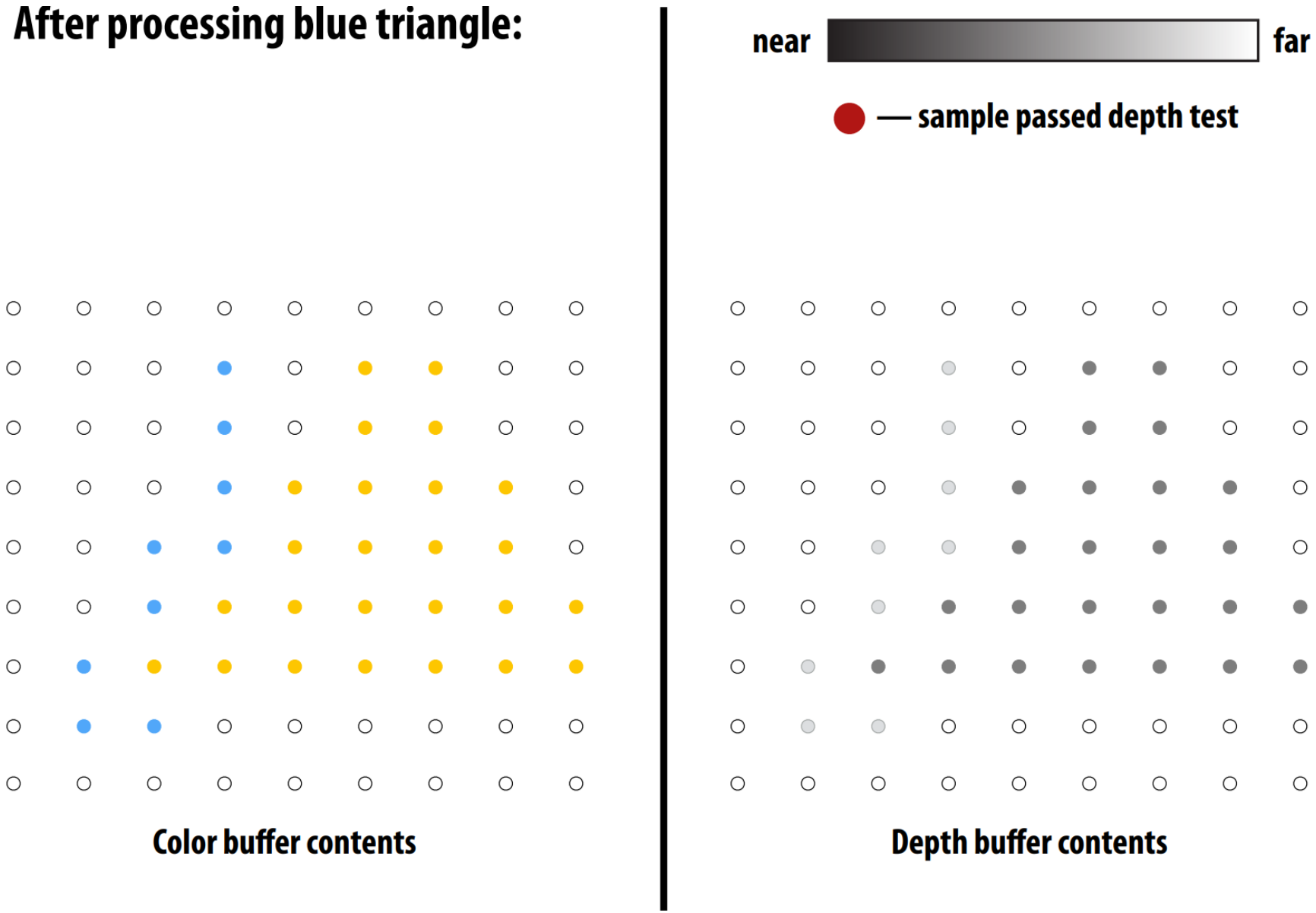


Color buffer contents

Depth buffer contents

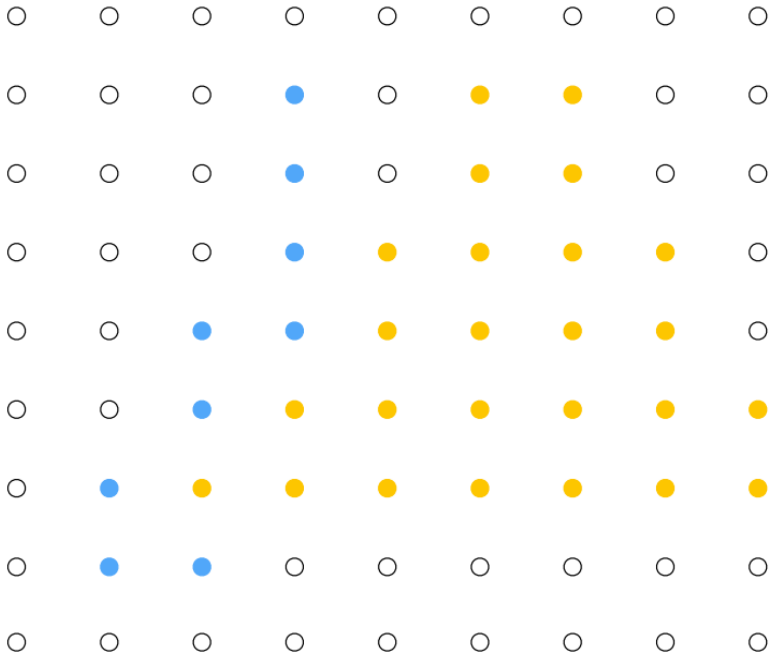
# Occlusion using the Z-buffer



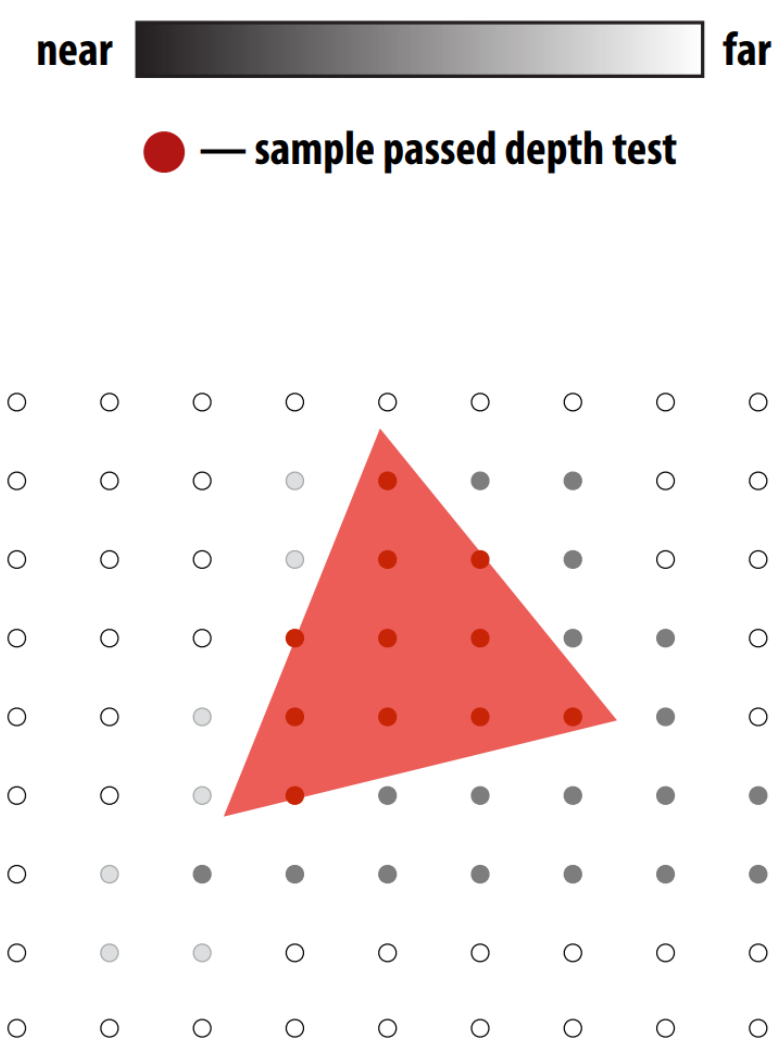
# Occlusion using the Z-buffer

**Processing red triangle:**

**depth = 0.25**

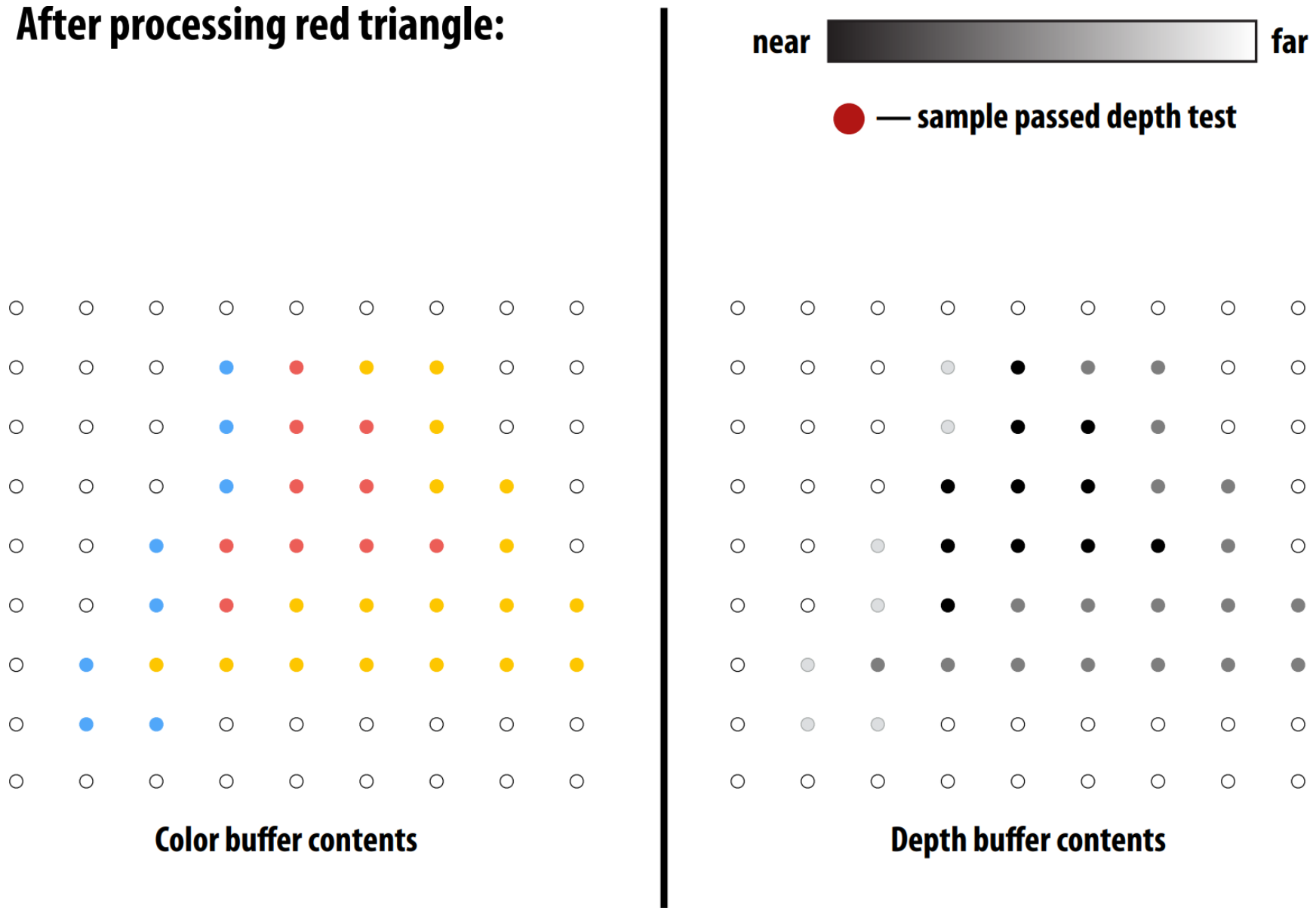


Color buffer contents

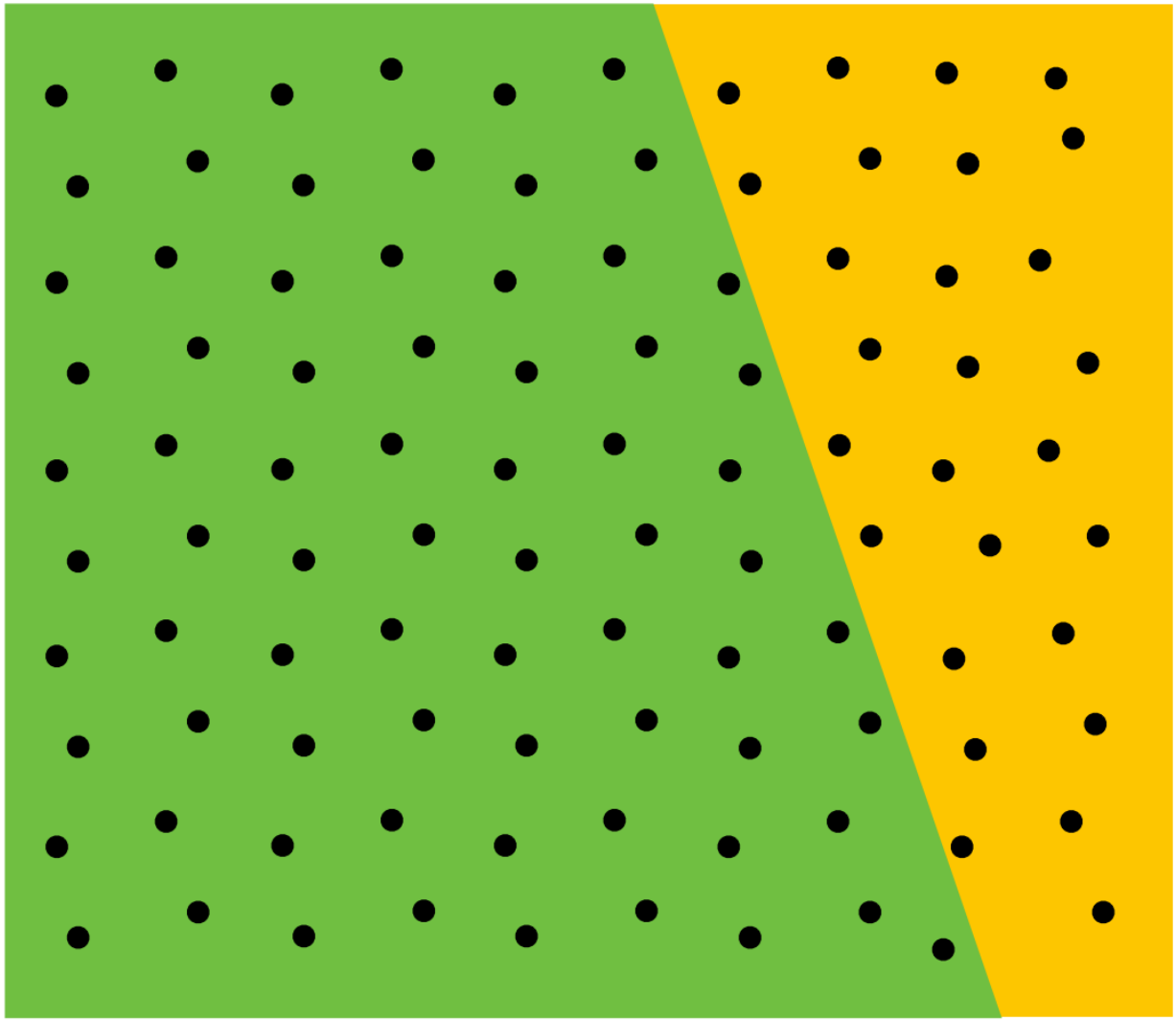



Depth buffer contents

# Occlusion using the Z-buffer

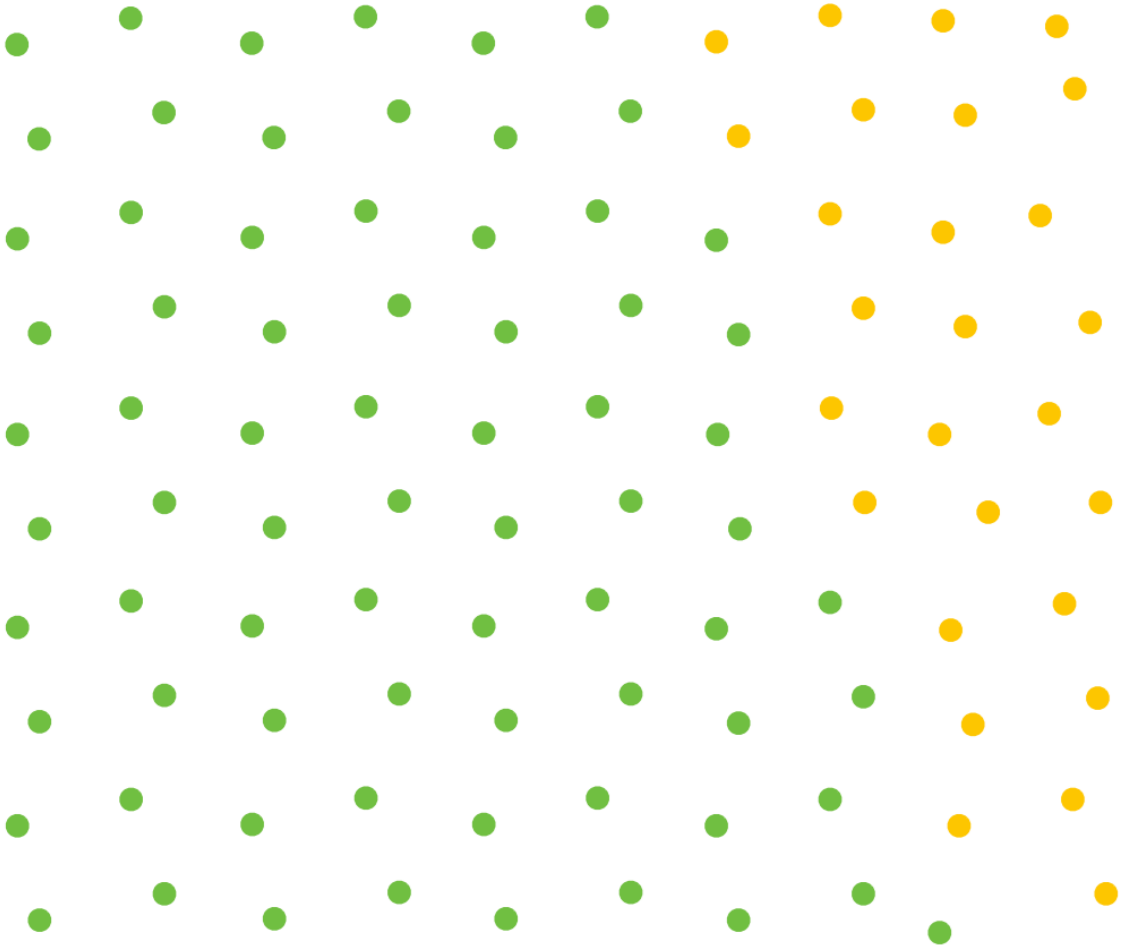


# Depth + Supersampling



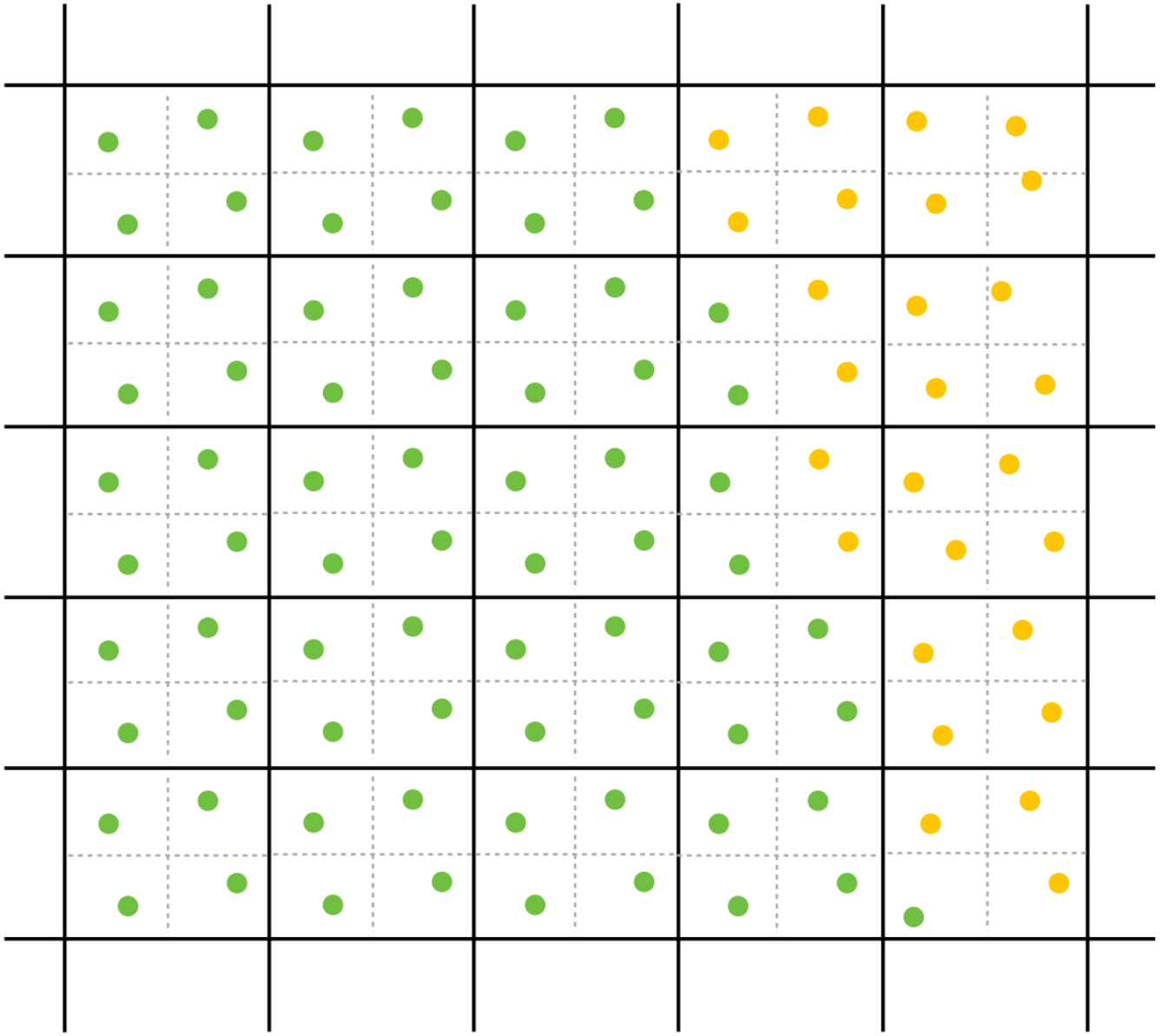
(Here: green triangle occludes yellow triangle)

# Depth + Supersampling

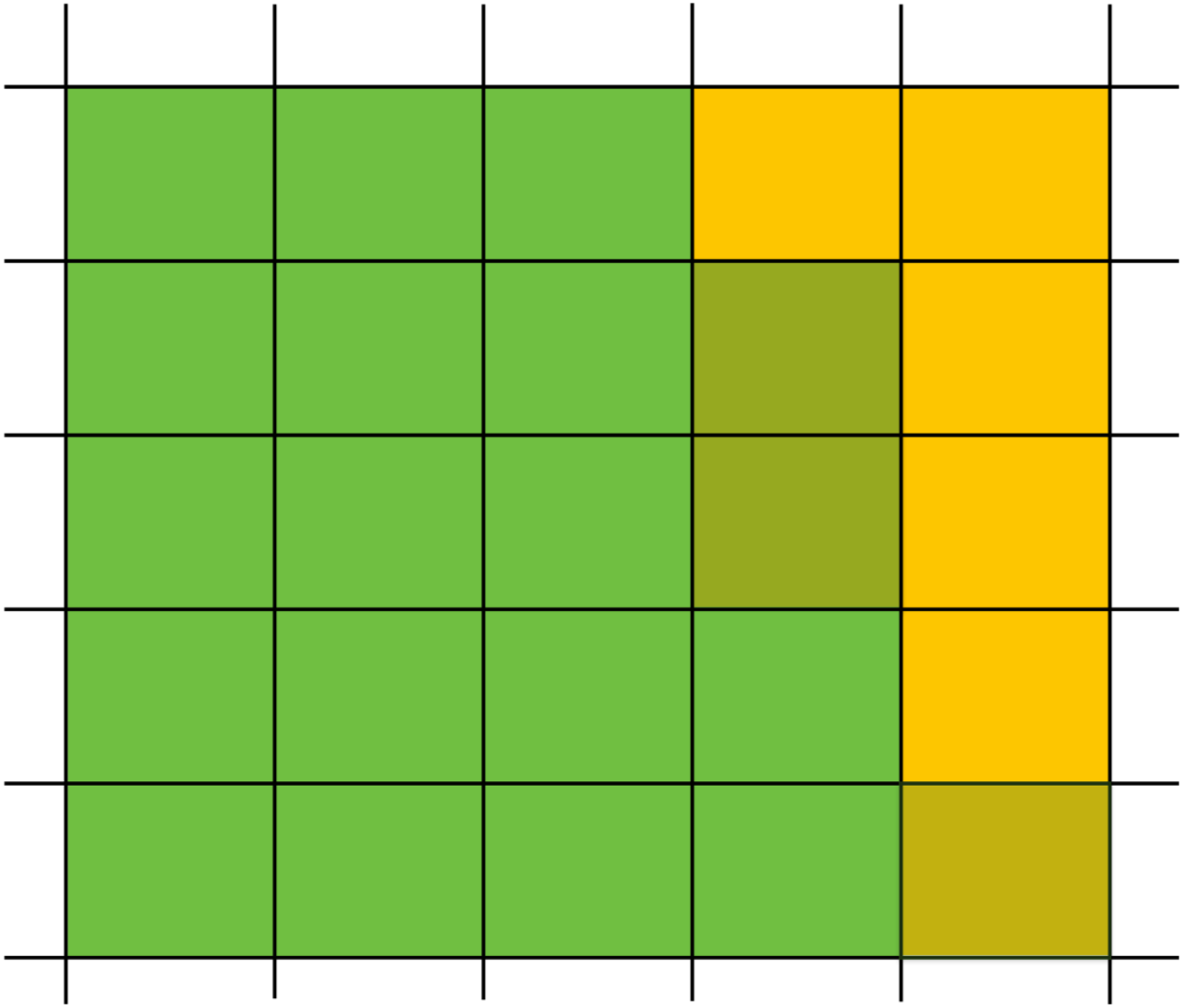


Color of super samples after rasterizing w/ depth buffer

# Color buffer contents (4 samples per pixel)



# Final resampled result



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

# Z-buffer Complexity

- Complexity
  - O(n) for n triangles (assuming constant coverage)
  - How is it possible to sort n triangles in linear time?
- Drawing triangles in different orders?
- Most important visibility algorithm
  - Implemented in hardware for all GPUs

# This Lecture

- Visibility
  - Back face culling
  - Hidden Surface Removal
    - Object space
    - Image space

- Shading
  - Phong shading model
  - Shading frequency

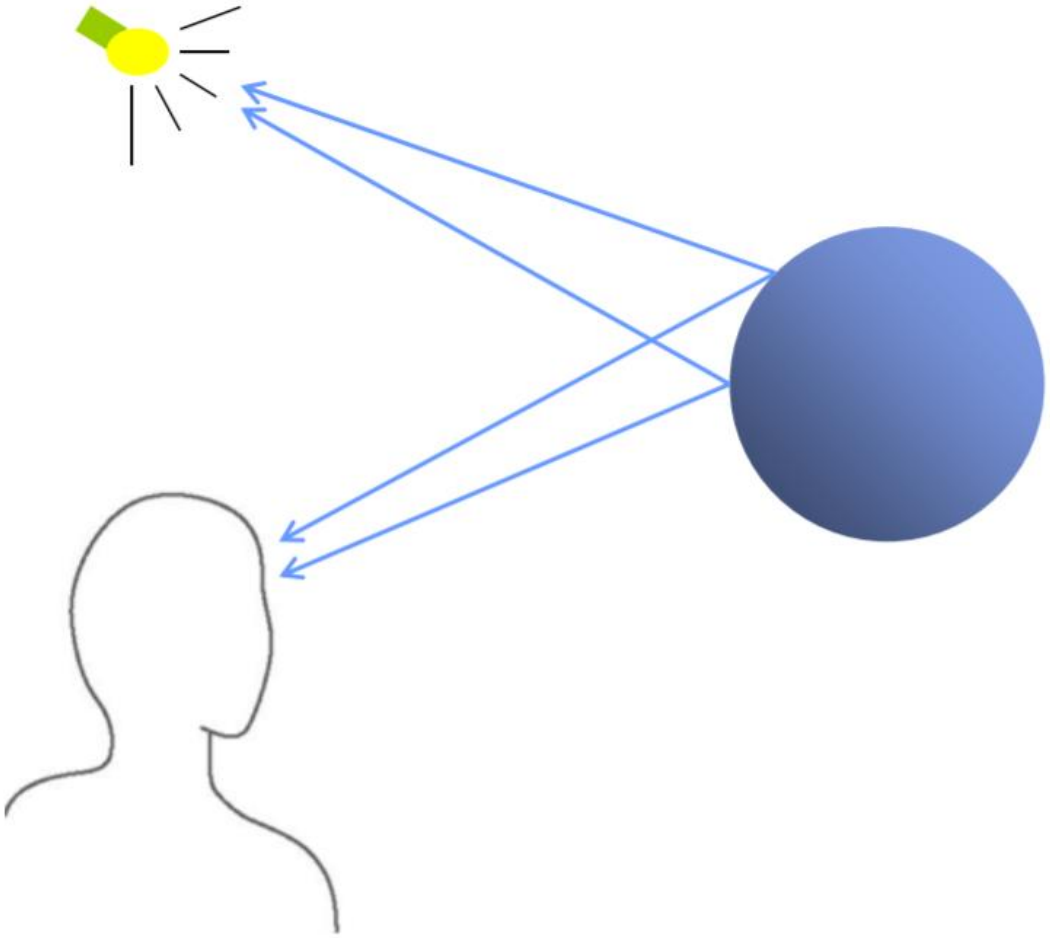
# Why need Lighting & Shading?

- Sphere without lighting & shading

- Sphere with shading
  - Has visual cues for humans (shape, light position, viewer position, surface orientation, material properties, etc)

# What causes shading

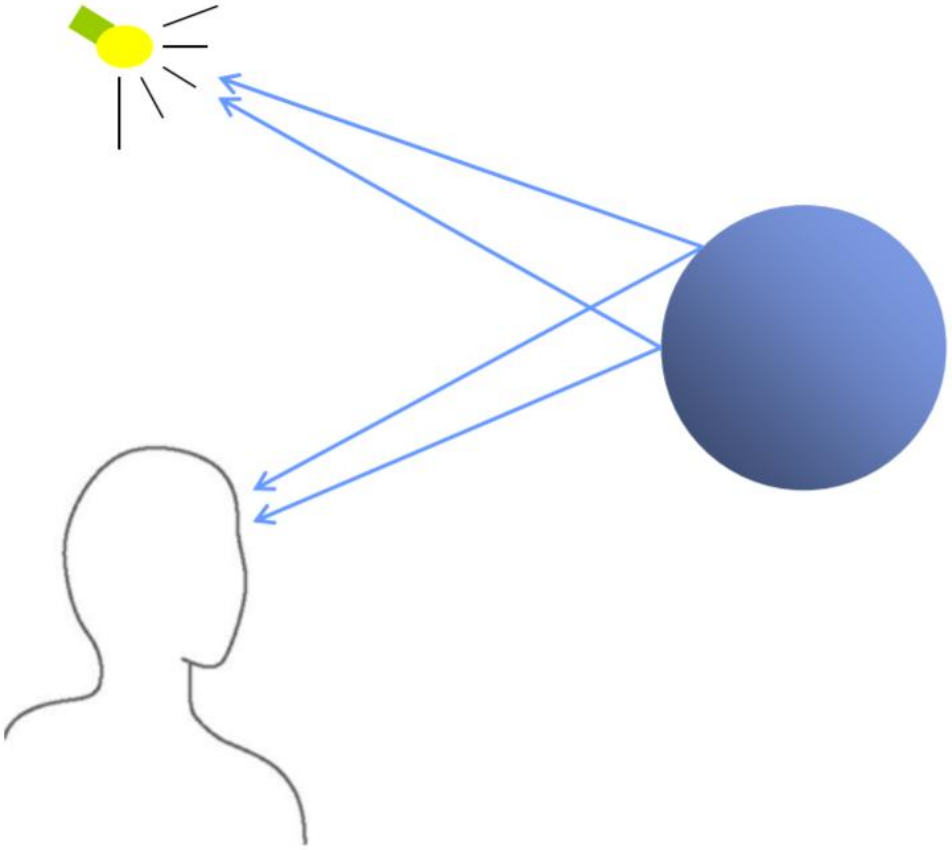
- Shading caused by different angles with light, camera at different points



# Shading

- Light-material interactions cause each point to have a different color or shade

- Need to consider
  - Light sources
  - Material properties
  - Location of viewer
  - Surface orientation



# This Lecture

- Visibility
  - Back face culling
  - Hidden Surface Removal
    - Object space
    - Image space
- Shading
  - Phong shading model
  - Shading frequency

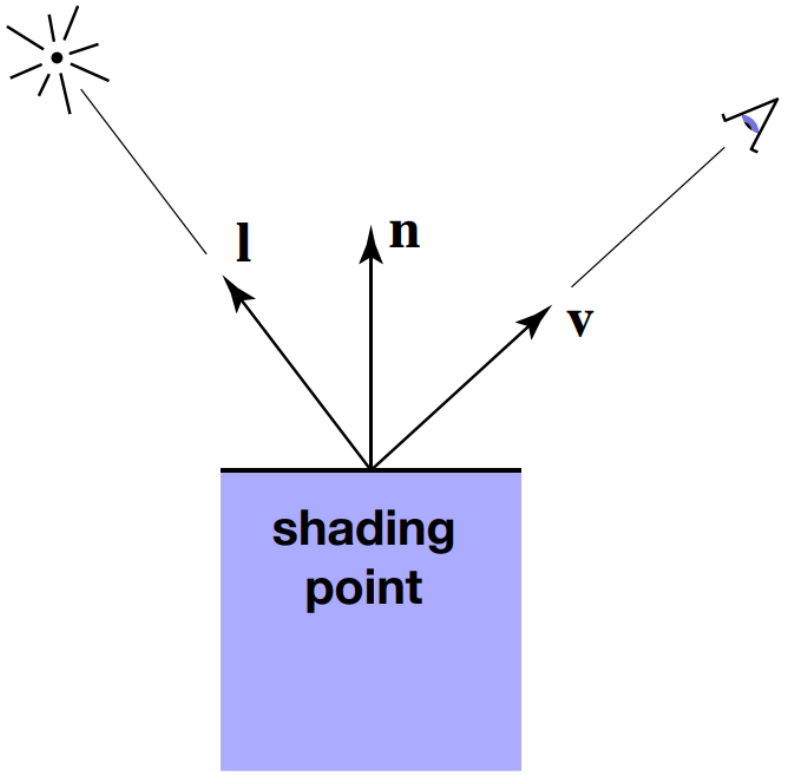
# Blinn-Phong Reflectance Model

- A simple shading model that can be computed rapidly
- Has three components
  - Diffuse
  - Specular
  - Ambient

# Shading is Local

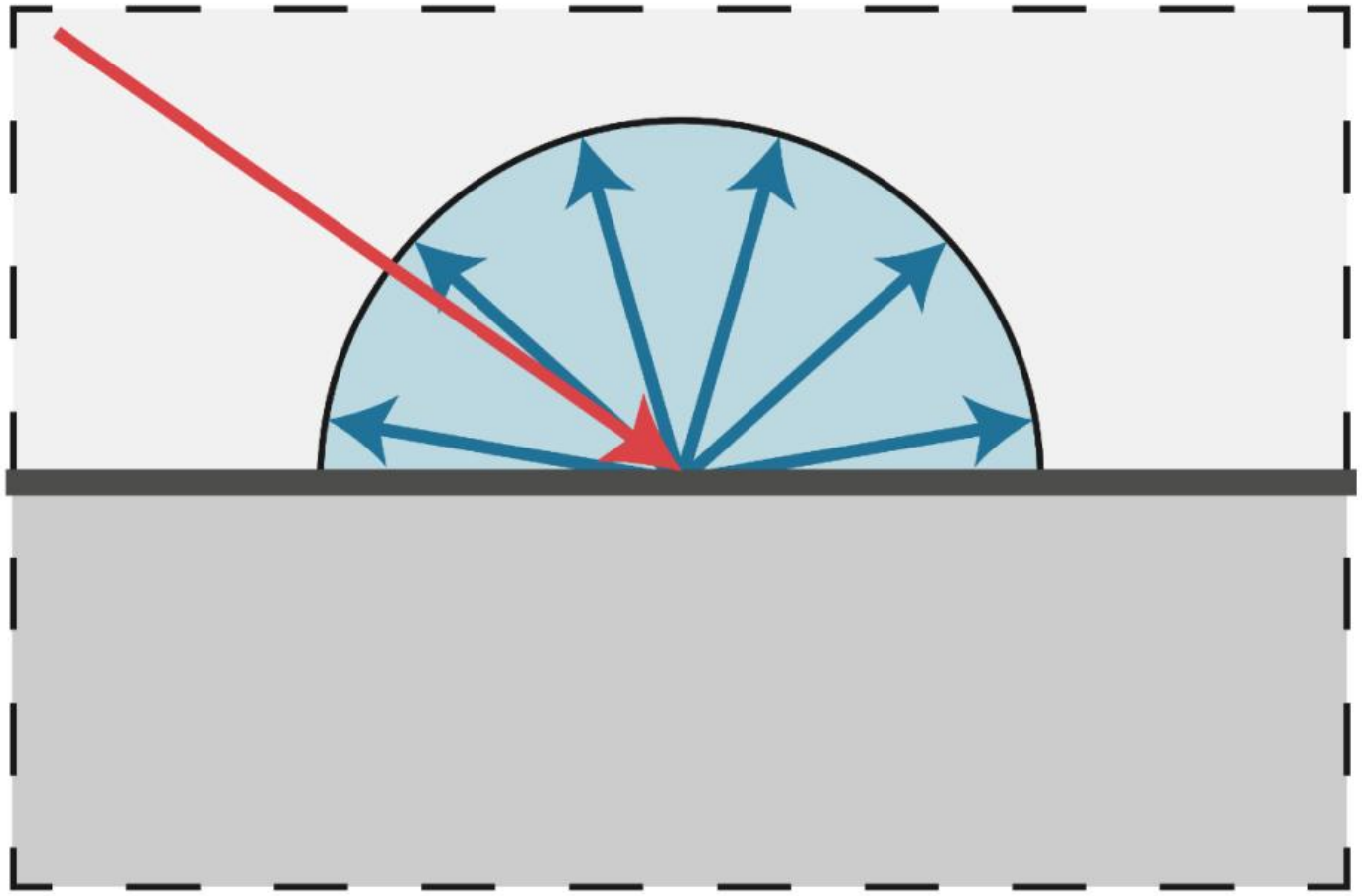
- Compute light reflected toward camera at a specific shading point

- Inputs
  - Viewer direction, v
  - Surface normal, n
  - Light direction, l
  - Surface parameters (color, shininess, …)



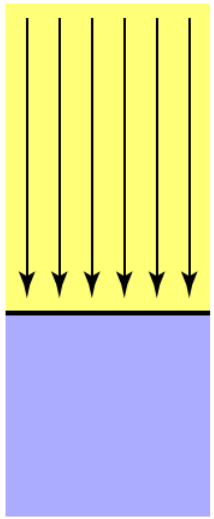
# Diffuse Reflection

- Light is scattered uniformly in all directions
  - Surface color is the same for all viewing directions

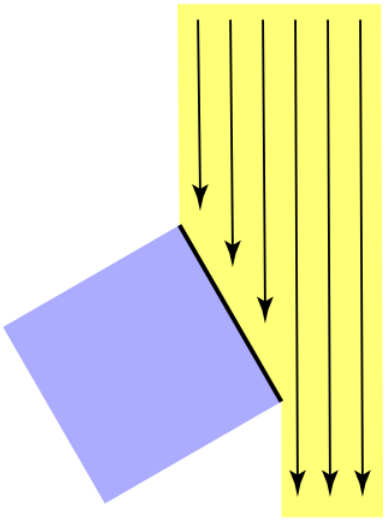


# Diffuse Reflection

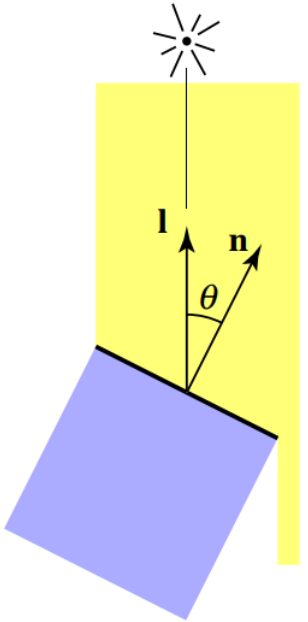
- But how much light (energy) is received?
  - Lambert's cosine law



Top face of cube receives a certain amount of light

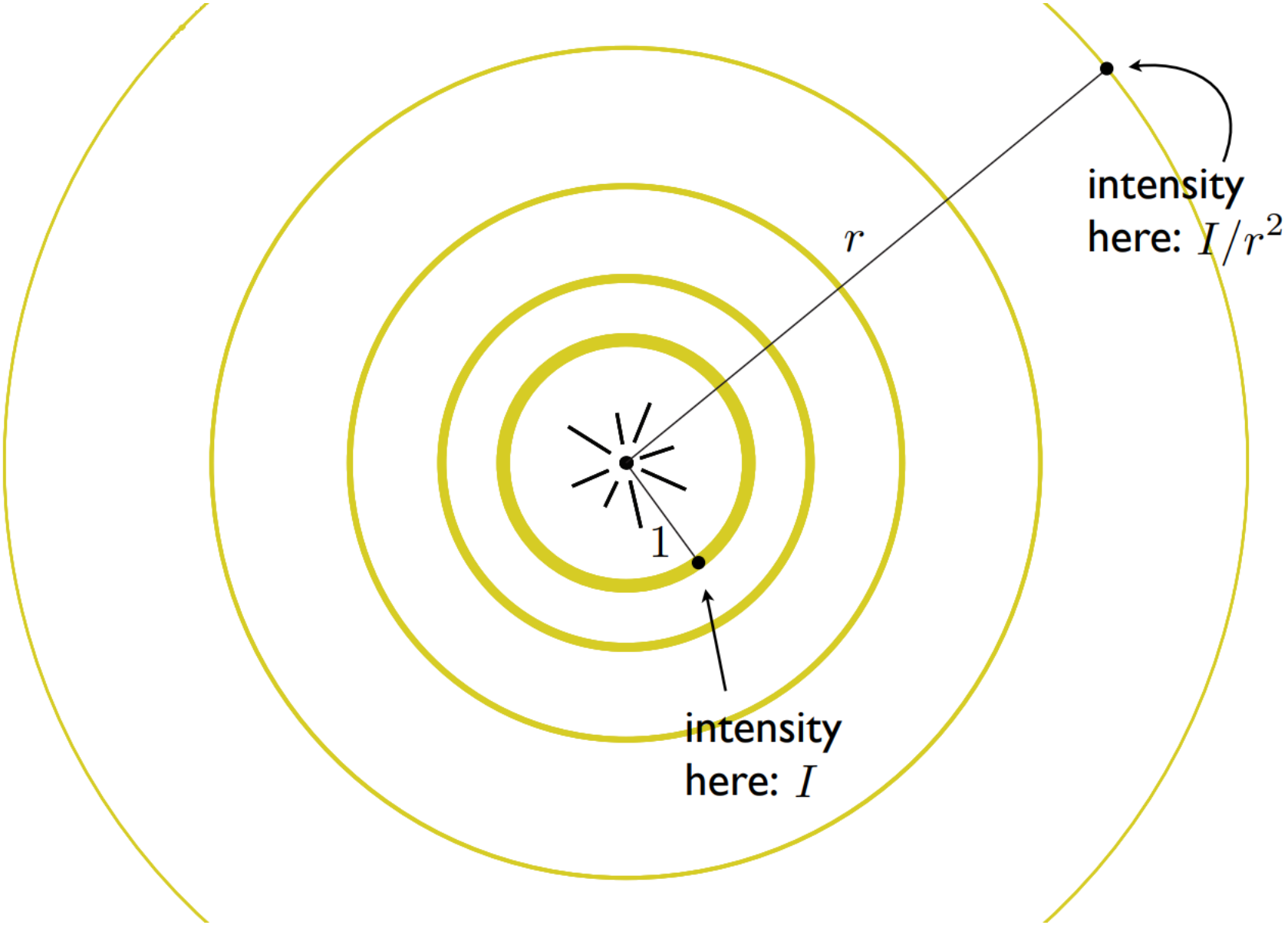


Top face of 60° rotated cube intercepts half the light



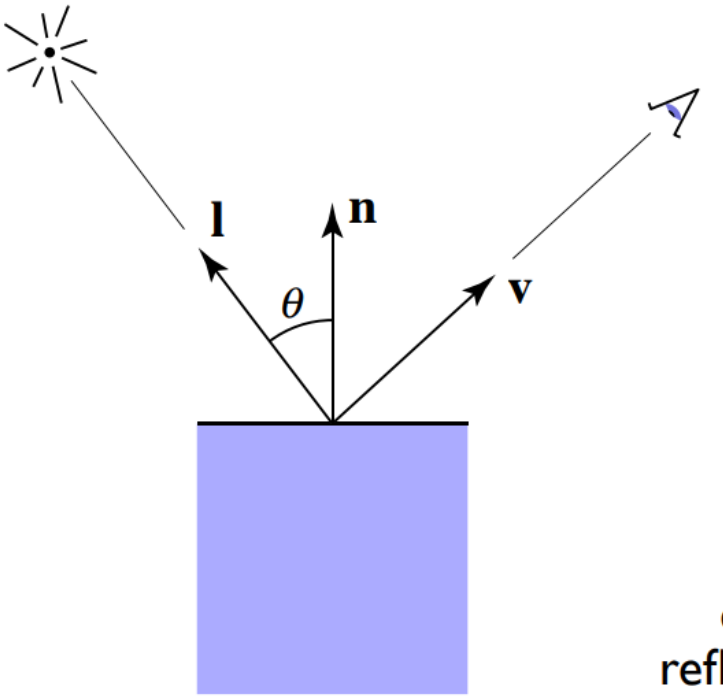

In general, light per unit area is proportional to $\cos \theta = l \cdot n$

# Light Fall Off



# Lambertian (Diffuse) Shading

- Shading independent of view direction




energy arrived
at the shading point

$$L_d = k_d \, (I/r^2) \max(0, \mathbf{n} \cdot \mathbf{l})$$

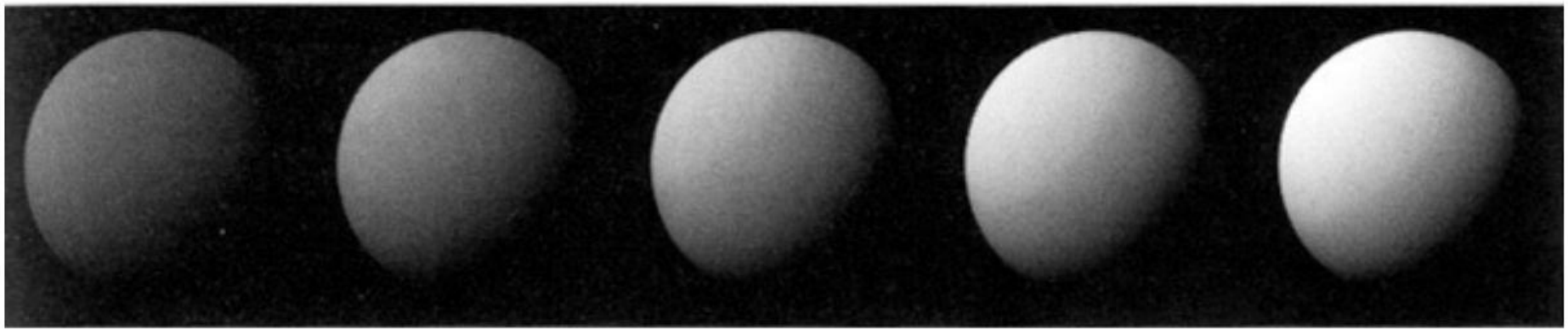
diffusely
reflected light

diffuse
coefficient
(color)

energy received
by the shading point

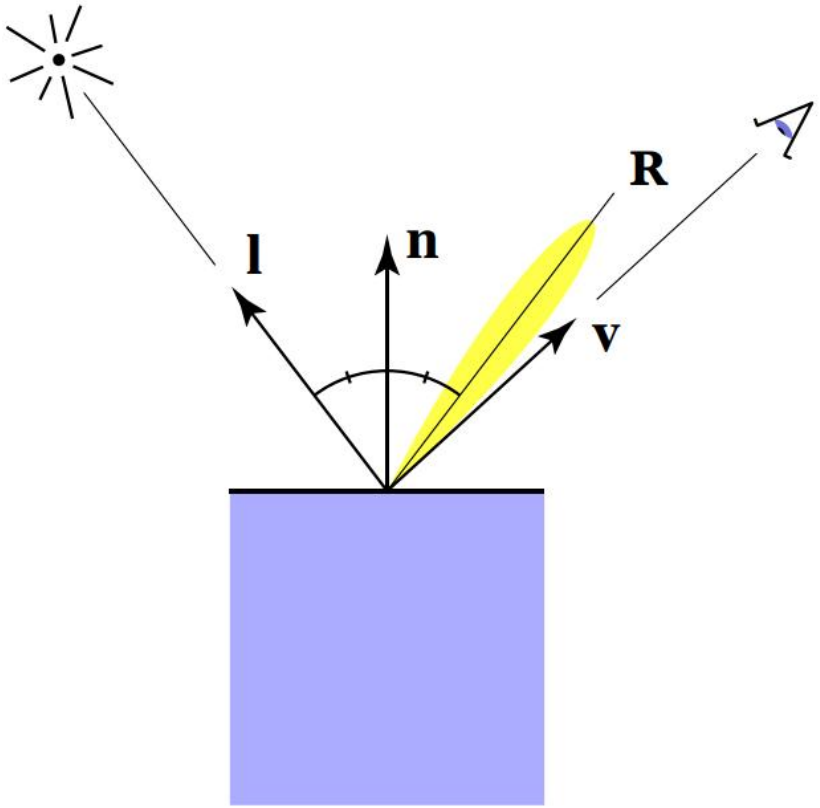
# Lambertian (Diffuse) Shading

- Produces diffuse appearance



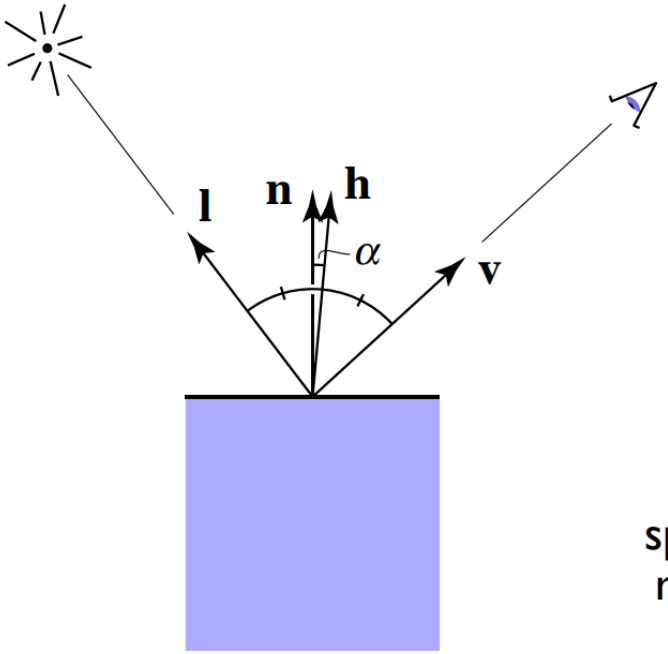
# Specular Term

- Intensity depends on view direction
  - Bright near mirror reflection direction



# Specular Term

- V close to mirror direction $\Leftrightarrow$ half vector near normal
  - Measure "near" by dot product of unit vectors




$$\mathbf{h} = \text{bisector}(\mathbf{v}, \mathbf{l})$$

(半程向量)

$$= \frac{\mathbf{v} + \mathbf{l}}{\|\mathbf{v} + \mathbf{l}\|}$$

$$L_s = k_s \left(I/r^2\right) \max(0, \cos\alpha)^p$$

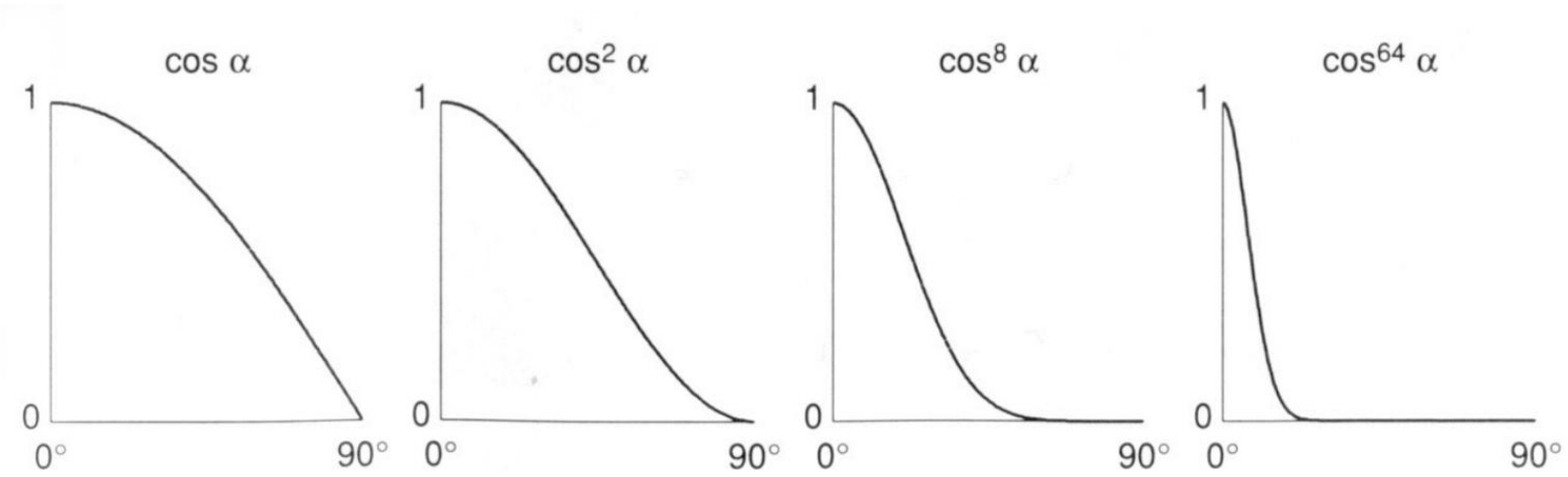
$$= k_s \left(I/r^2\right) \max(0, \mathbf{n} \cdot \mathbf{h})^p$$

specularly reflected light

specular coefficient

# Cosine Power Plots

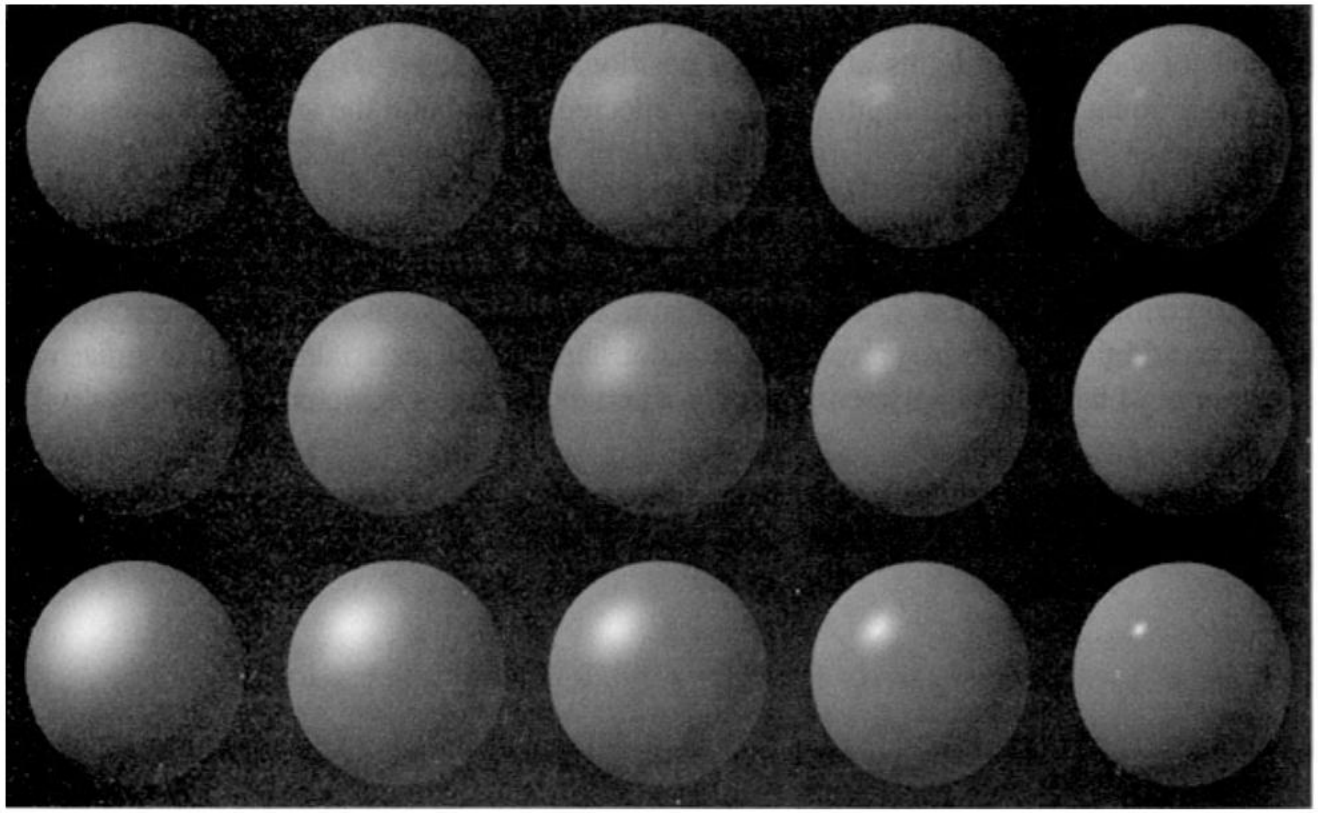
- Increasing p narrows the reflection lobe



# Specular Term

Blinn-Phong

$$L_s = k_s \left(I/r^2\right) \max(0, \mathbf{n} \cdot \mathbf{h})^p$$



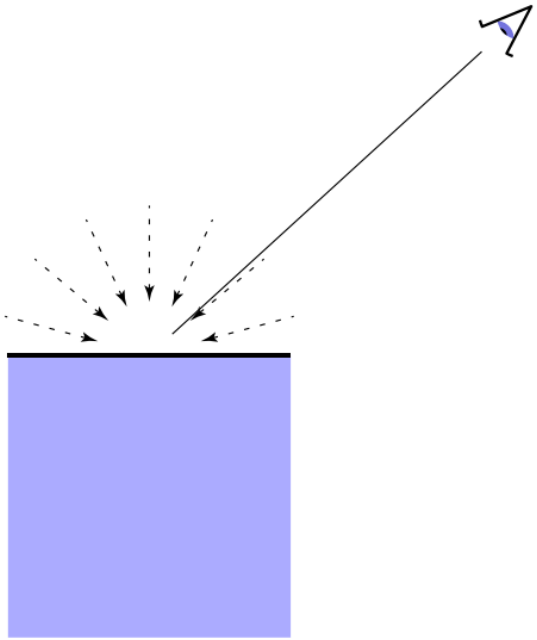
$k_s$

$p$

Note: showing
Ld + Ls together

# Ambient Term

- Shading that does not depend on anything
  - Add constant color to account for disregarded illumination and fill in black shadows
  - This is approximate / fake!

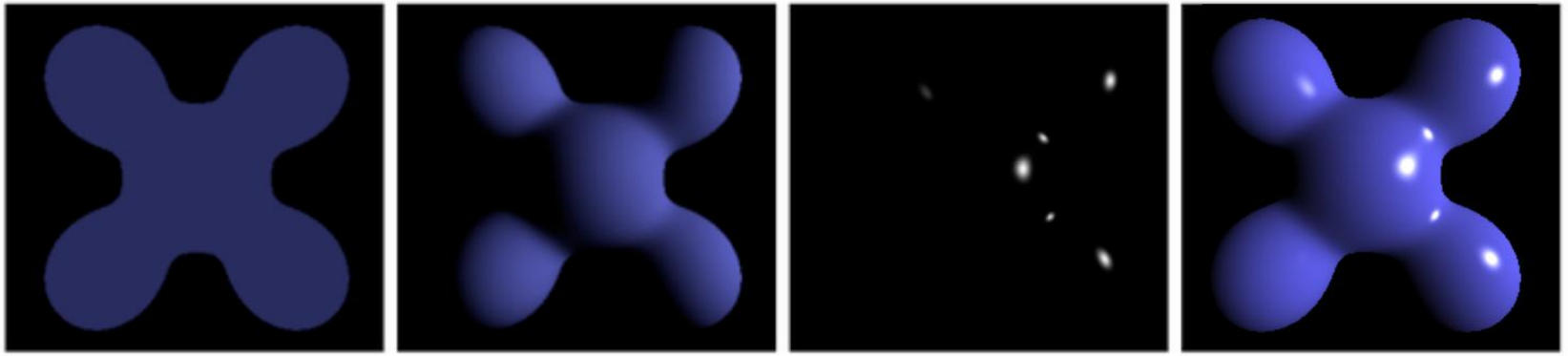


$$L_a = k_a \, I_a$$

ambient coefficient

reflected ambient light

# Blinn-Phong Reflection Model

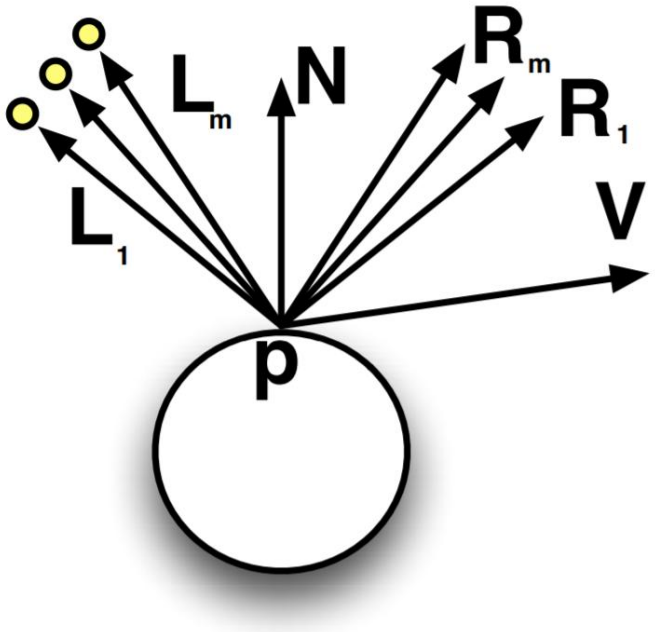



$$L = L_a + L_d + L_s$$
$$= k_a\, I_a + k_d\, (I/r^2) \max(0, \mathbf{n} \cdot \mathbf{l}) + k_s\, (I/r^2) \max(0, \mathbf{n} \cdot \mathbf{h})^p$$

# Multiple Light sources

- For multiple light sources we simply compute the illumination from each source and sum them

$$I = k_a I_a + \sum_{l=1}^{m} \left( k_d \left( \frac{I_l}{{r_l}^2} \right) \max(0, \mathbf{n} \cdot \mathbf{l}) + k_s \left( \frac{I_l}{{r_l}^2} \right) \max(0, \mathbf{n} \cdot \mathbf{h})^p \right)$$



# This Lecture

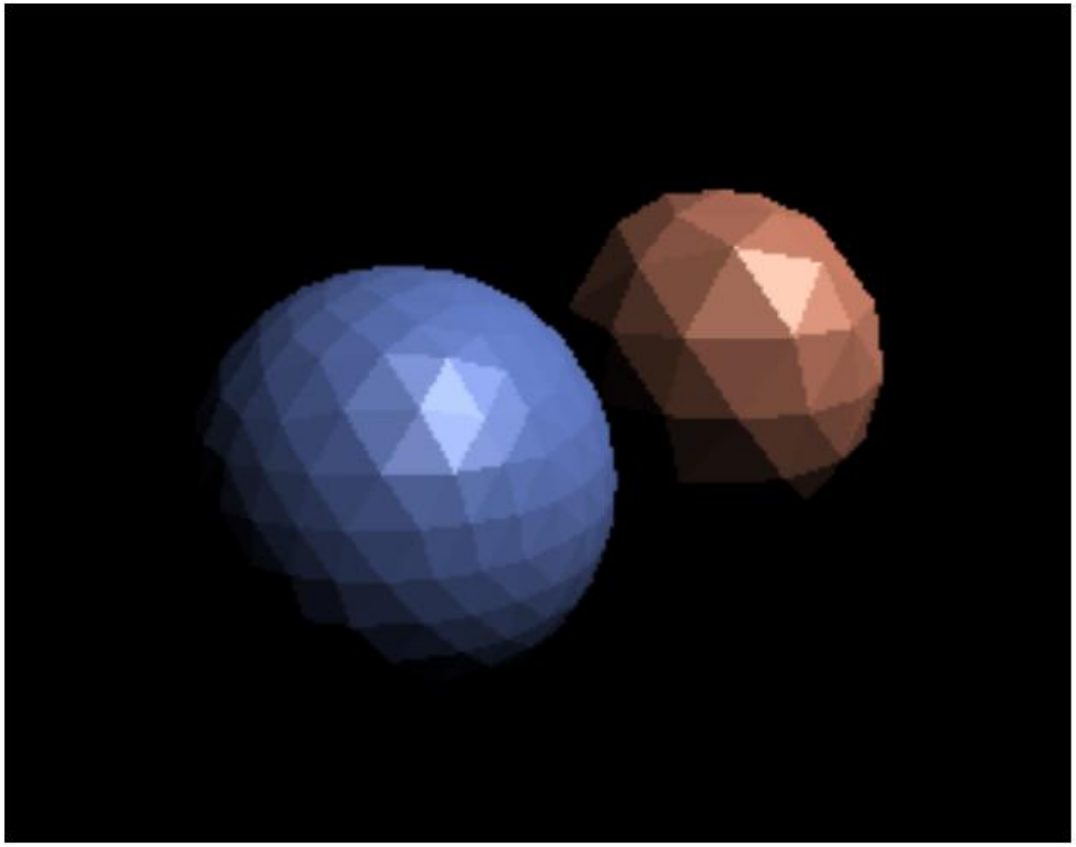
- Visibility
  - Back face culling
  - Hidden Surface Removal
    - Object space
    - Image space
- Shading
  - Phong shading model
  - Shading frequency

# Shading Frequencies

- What caused the shading difference?

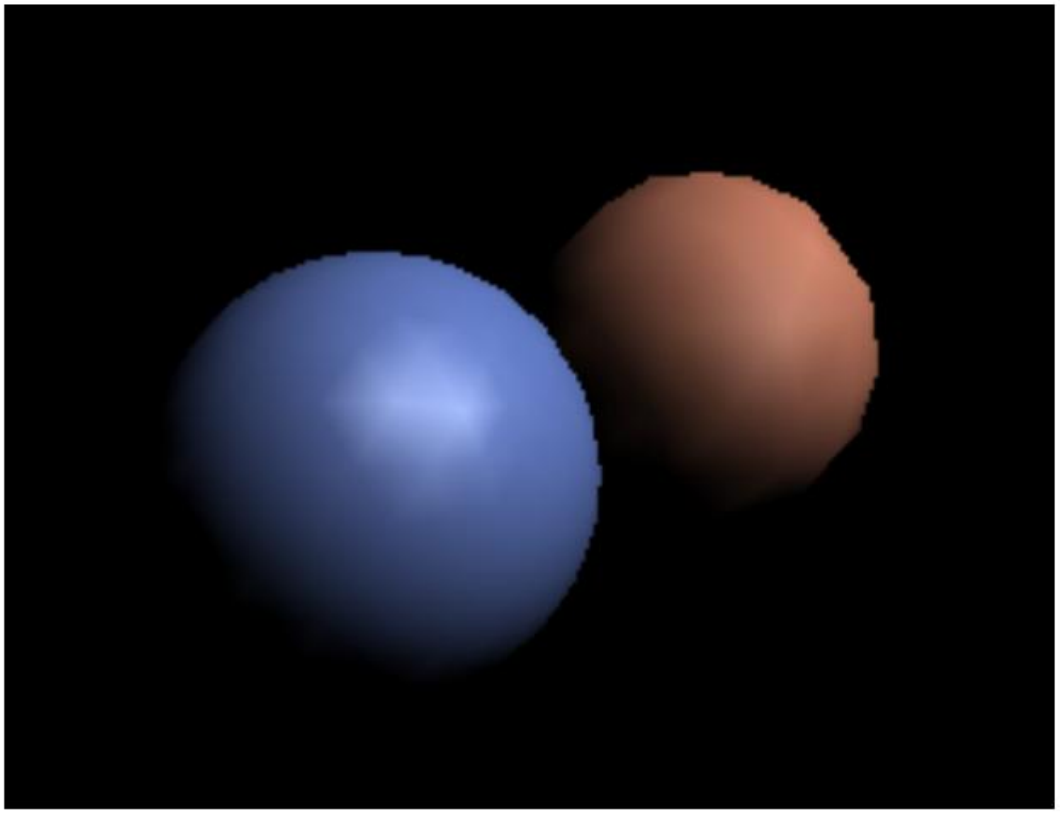
# Shade each triangle (flat shading)

- Flat shading
  - Triangle face is flat — one normal vector
  - Not good for smooth surfaces



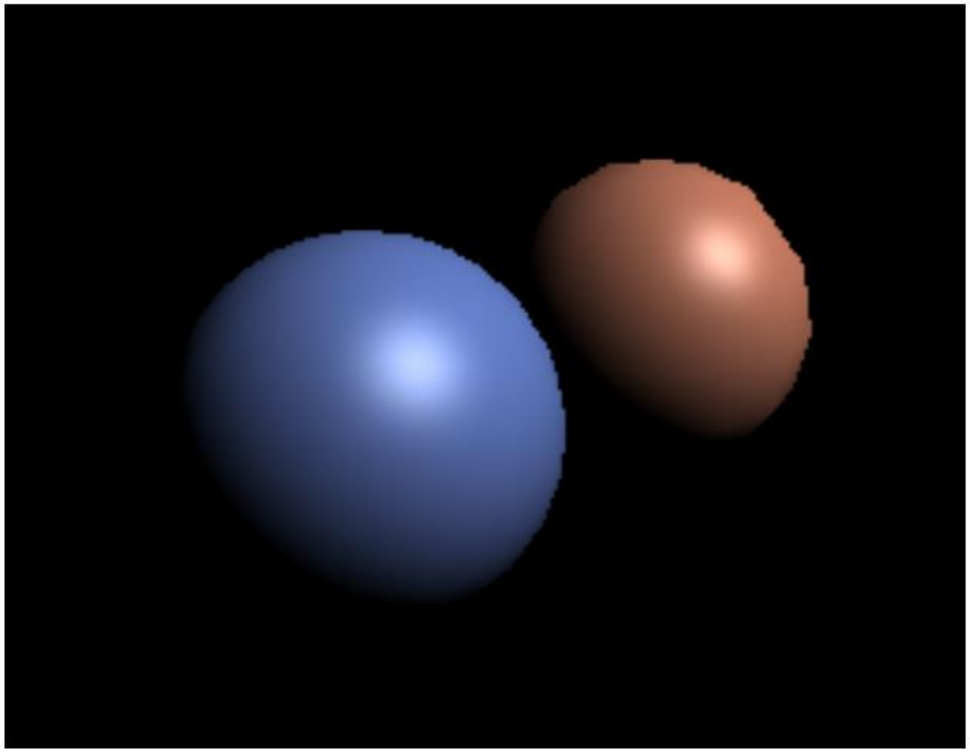
# Shade each vertex (Gouraud shading)

- Gouraud shading
  - Interpolate colors from vertices across triangle
  - Each vertex has a normal vector (how?)

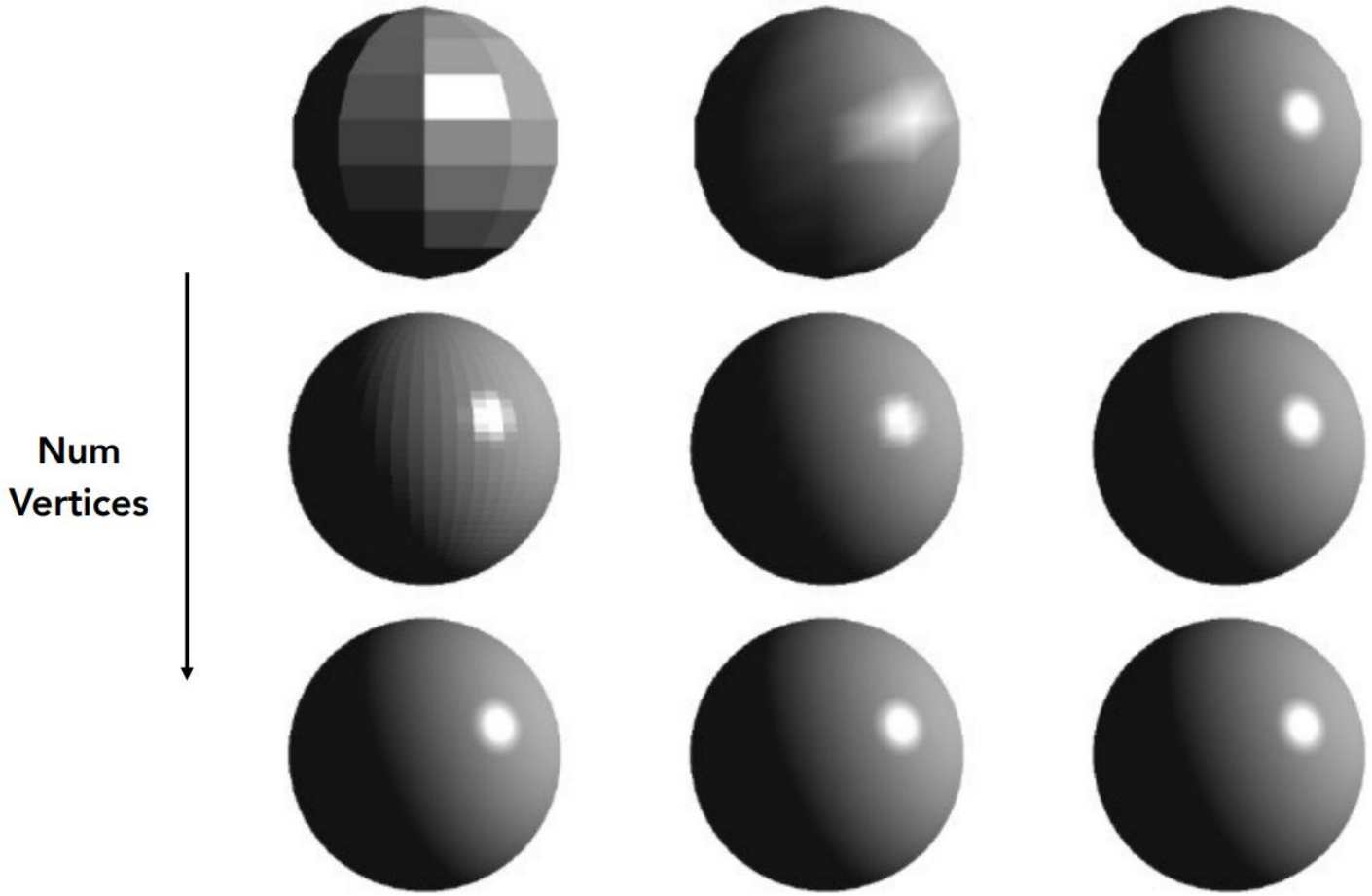


# Shade each pixel (Phong shading)

- Phong shading
  - Interpolate normal vectors across each triangle
  - Compute full shading model at each pixel



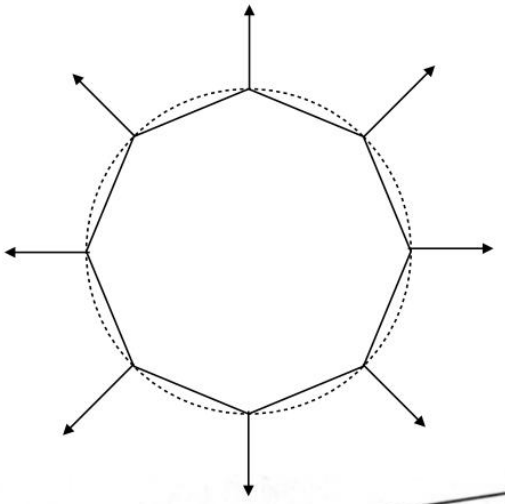
# Shading Frequency: Face, Vertex or Pixel



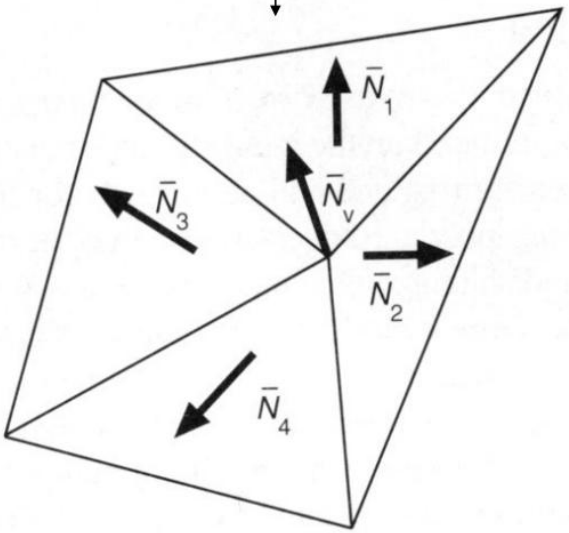

| | | | |
|---|---|---|---|
| Shading freq. : | Face | Vertex | Pixel |
| Shading type : | Flat | Gouraud | Phong |

# Defining Per-Vertex Normal Vectors

- Best to get vertex normals from the underlying geometry
  - e.g. consider a sphere



- Otherwise have to infer vertex normals from triangle faces
  - Simple scheme: average surrounding face normals




$$N_v = \frac{\sum_i N_i}{\| \sum_i N_i \|}$$

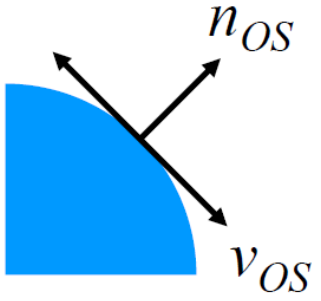
# Computing vertex normal

**Dot product** $n_{OS}^{\mathbf{T}}\, v_{OS} = 0$

$$n_{OS}^{\mathbf{T}}\ (\mathbf{M}^{-1}\ \mathbf{M})\ v_{OS} = 0$$

$$(n_{OS}^{\mathbf{T}}\ \mathbf{M}^{-1})\ (\mathbf{M}\ \ v_{OS}) = 0$$

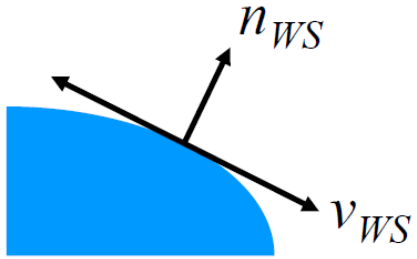
$$(n_{OS}^{\mathbf{T}}\ \mathbf{M}^{-1})\ v_{WS} = 0$$




is perpendicular to normal

$$n_{WS}^{\mathbf{T}} = n_{OS}^{\mathbf{T}}\,(\mathbf{M}^{-1})$$

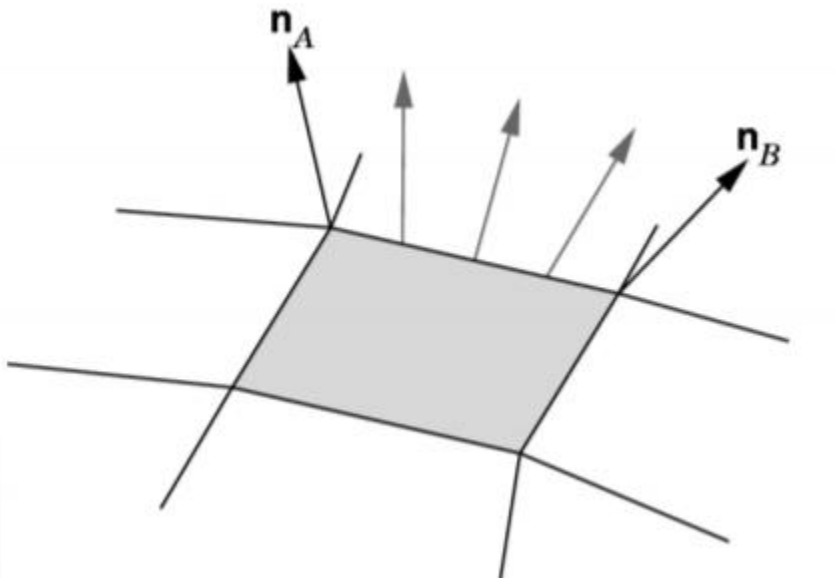
$$\boxed{n_{WS} = (\mathbf{M}^{-1})^{\mathbf{T}}\, n_{OS}}$$

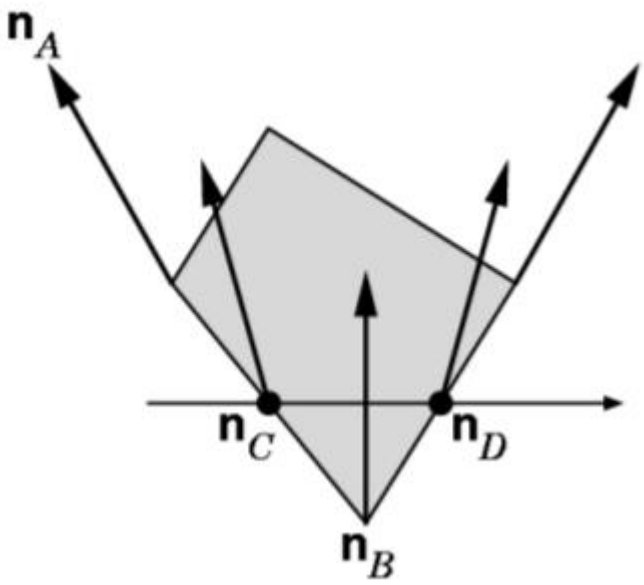



The matrix is the transpose of inverse of **M**

# Defining Per-Pixel Normal Vectors

- Interpolate normals at vertices and then edges, then at every interior point



# 谢谢


北京航空航天大學

人工智能学院(人工智能研究院)