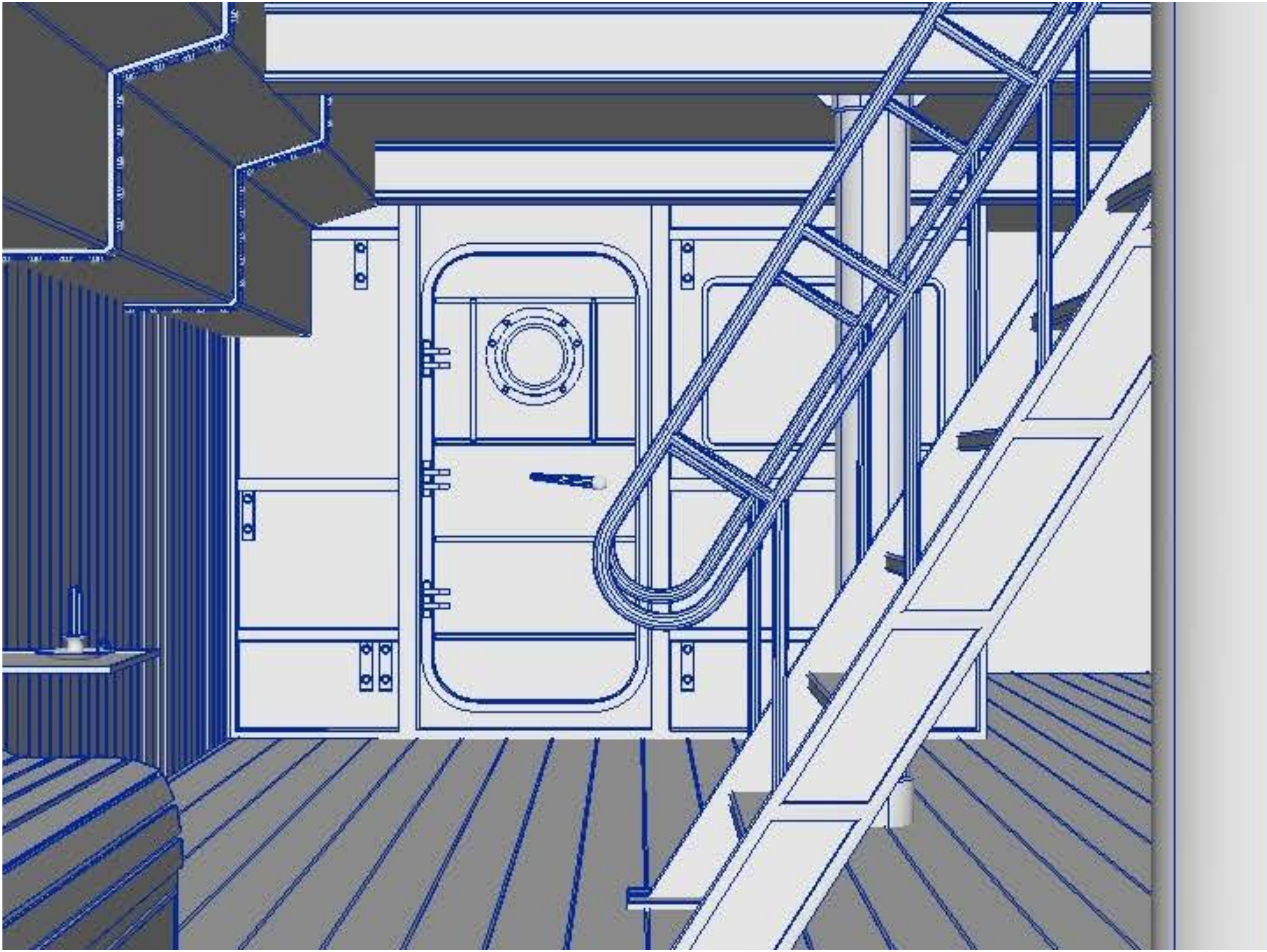
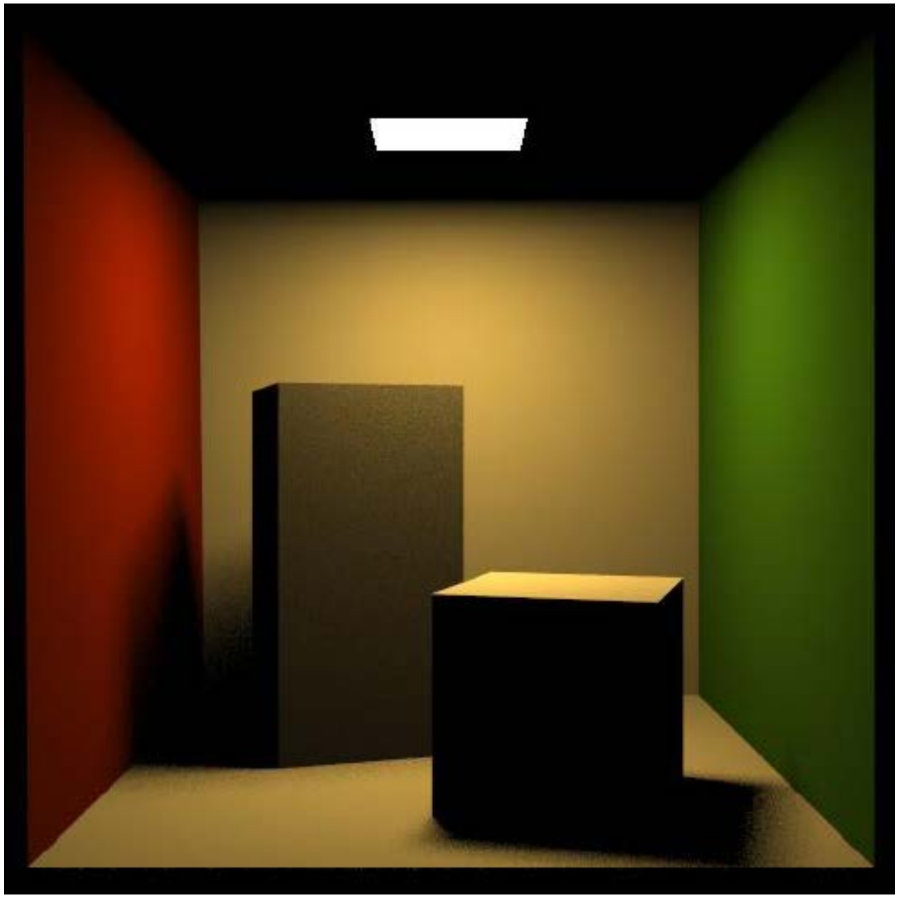
Shading
(introduction to rendering)



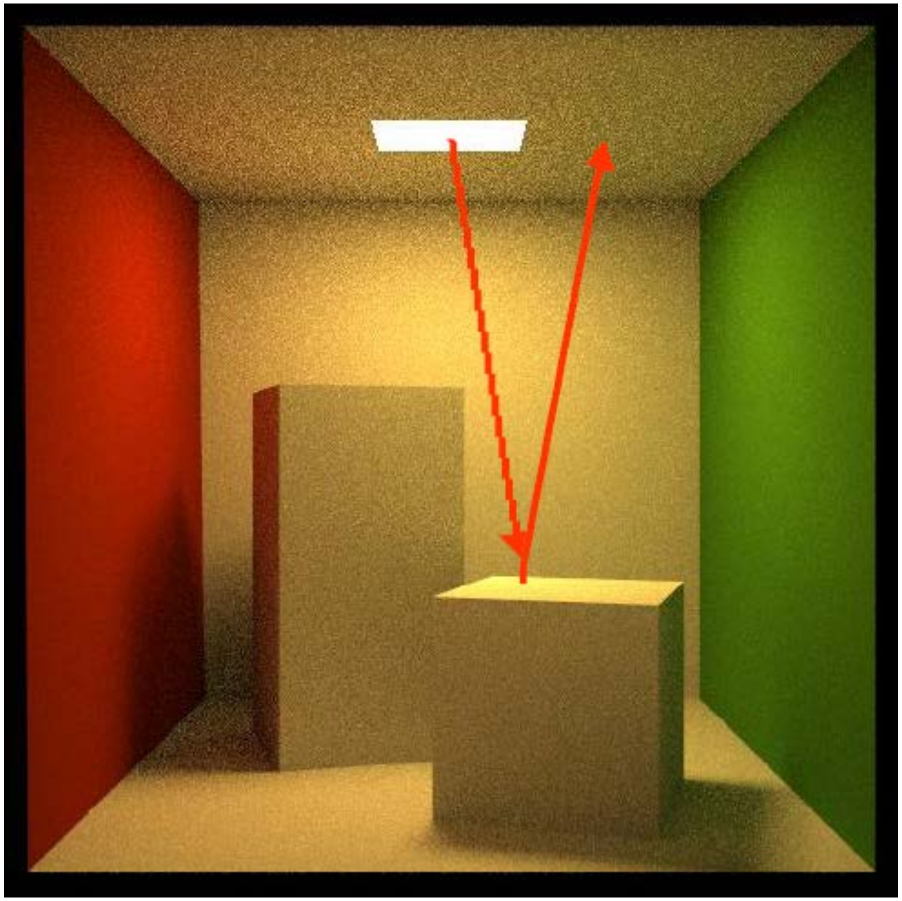
Rendering: simulation of light transport

- Diffuse scattering
    - matt surfaces
- Specular reflection
    - shiny surfaces
    - highlight
- Transparency
    - glass, water
    - penetrate the surface
- Global light transport
    - realism

Global illumination

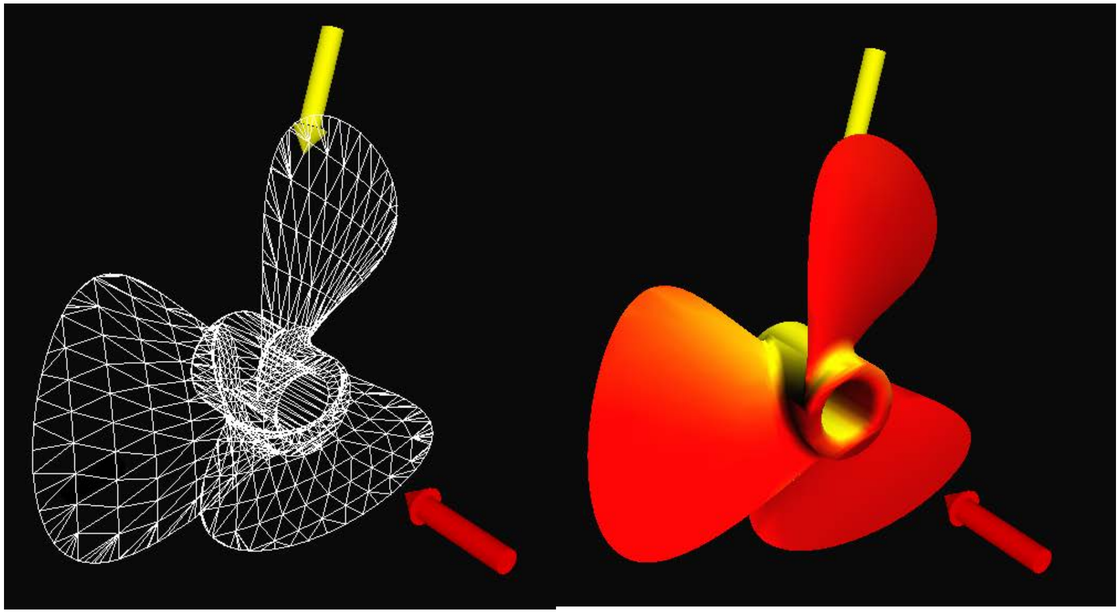


No multiple diffuse reflections



Multiple diffuse reflections

Local illumination



A (modest) example of shading

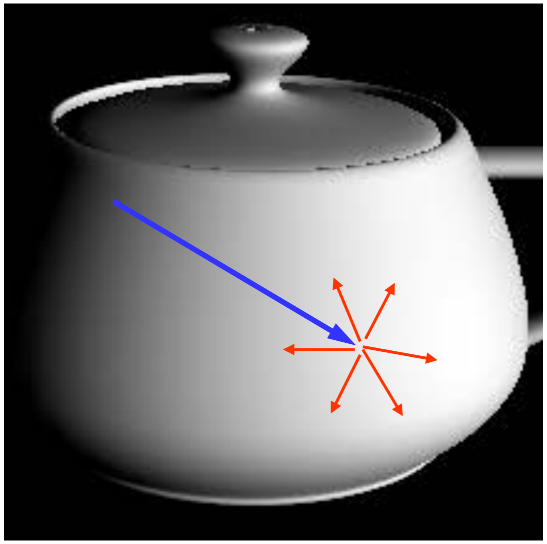
- Input:
  - a 3D object
  - Material and color of the object
  - Position and structure of the light source
  - “Intensity” of the light source
- Output:
  - Color and intensity of points of the given object

Dealing with color

- Three component intensity (red, green, blue)
- Luminance (intensity) of the source
    - Red component of source ⟹ red component of image
    - Green component of source ⟹ green component of image
    - Blue component of source ⟹ blue component of image
- Three similar but independent calculations
- We focus on one scalar value only

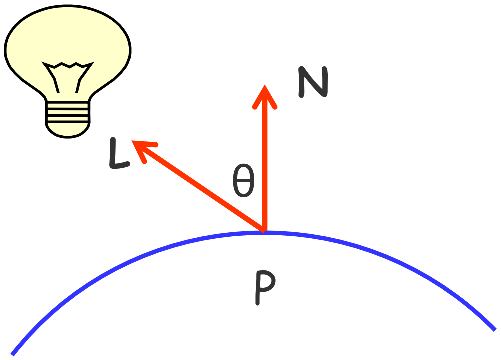
Diffuse reflection

- A perfect diffuse reflector (Lambertian surfacee) scatters the light equally in all directions
- Same appearance to all viewers
  - Material of the surface
  - The position of the light
- Same appearance to all viewers



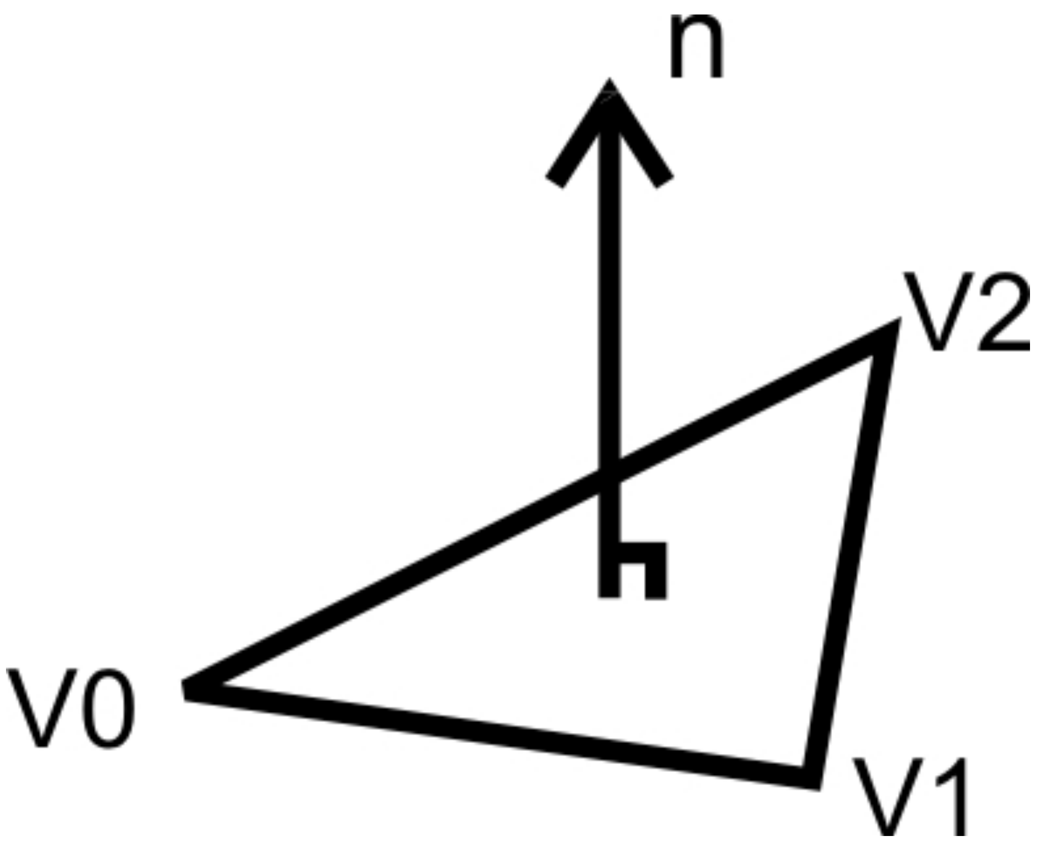
Diffuse: Two important vectors

- To compute the intensity at P, we need
  - The unit normal vector **N**,
  - The unit vector **L**, from P to the light

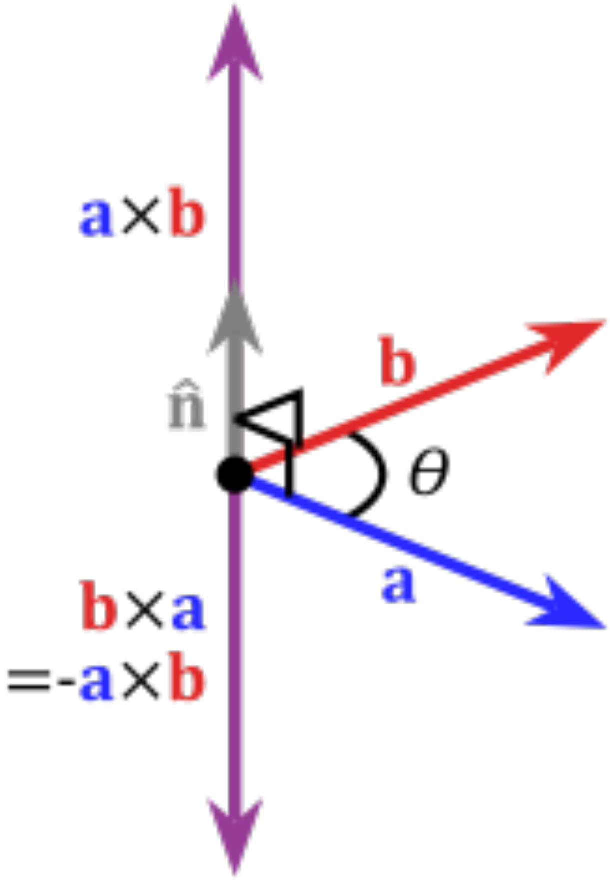


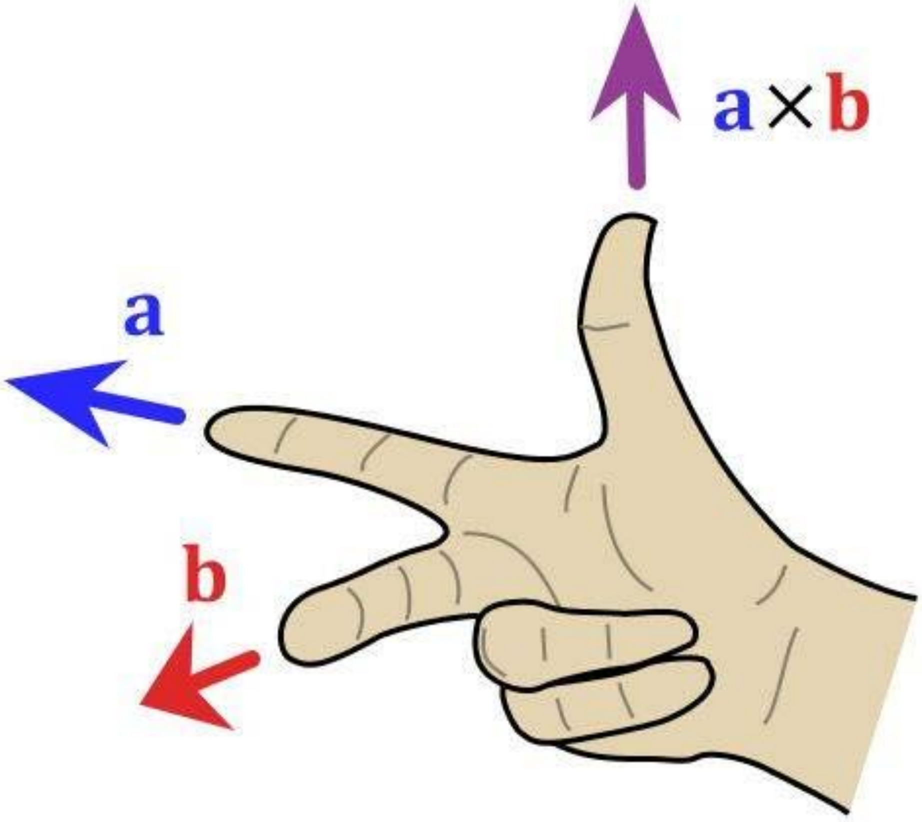
Normals

- What direction is the surface facing?



CrossProduct

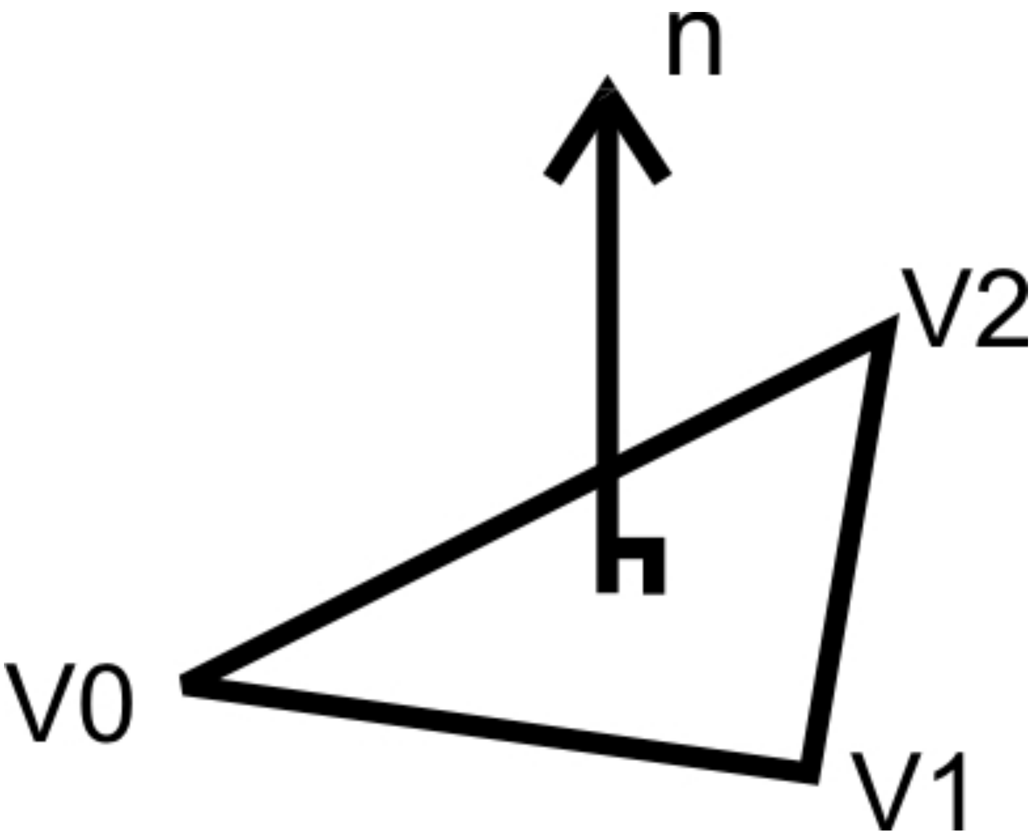




- n.x = a.y * b.z - a.z * b.y
- n.y = a.z * b.x - a.x * b.z
- n.z = a.x * b.y - a.y * b.x

Normals

- $A = V2 - V1$
- $B = V0 - V1$
- $N = A \times B$



Lambert's cosine law

- I : diffuse reflection at P
- $I = I_p k_d \cos(\vec{L}, \vec{N}) = I_p k_d \vec{L} \bullet \vec{N}$
- I_p: intensity of the light from source
- $0 \leq k_d \leq 1$: coefficient of diffuse reflection

Coefficient of diffuse reflection

- k_d is usually determined by a trial and error
- Examples:

Component	Gold	Black plastic	Silver
Red	0.75	0.01	0.5
Green	0.6	0.01	0.5
Blue	0.22	0.01	0.5

k_d=0.05

k_d=0.25

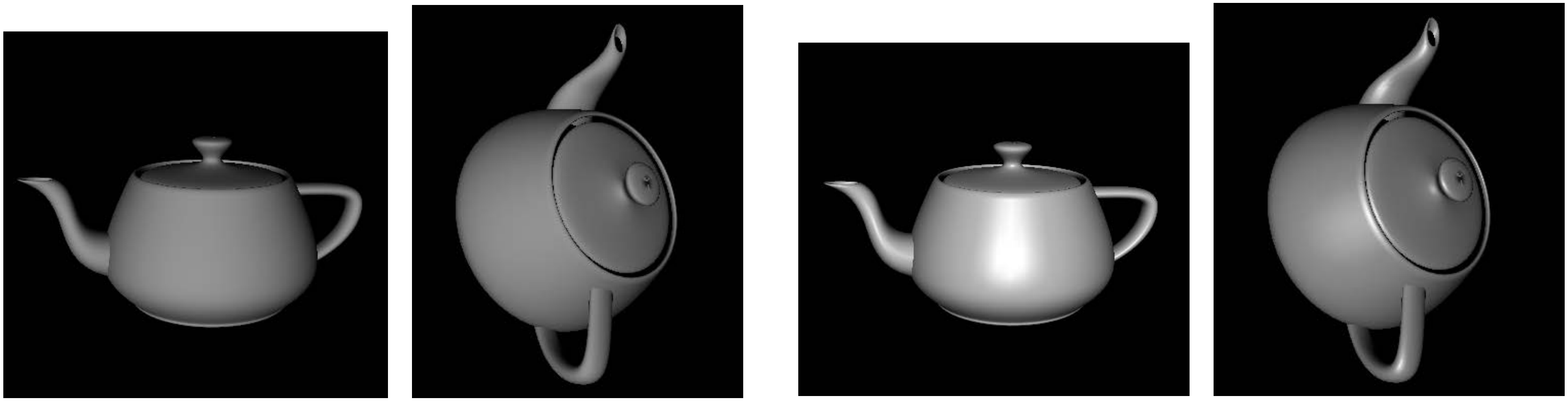
k_d=0.5

k_d=0.75

k_d=1

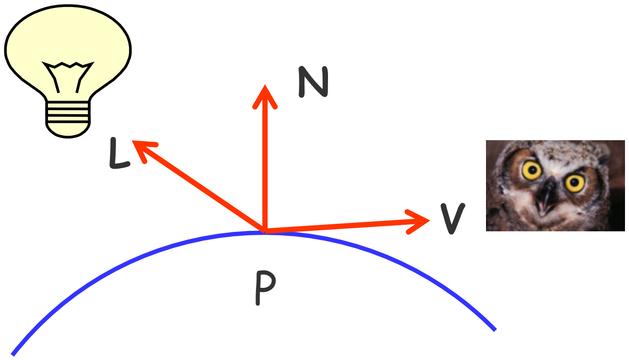
Specular reflection

- Diffusive reflection: no highlights, rough surface
- Specular reflection: highlights, shiny and smooth surfaces
- View dependent reflection



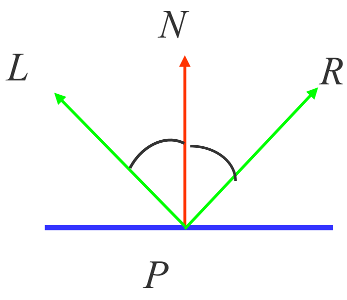
Specular: Three important vectors

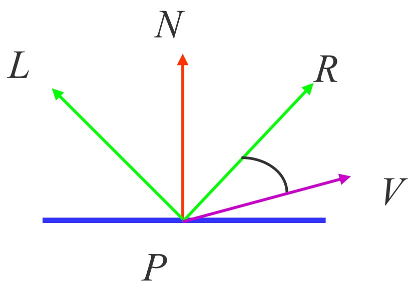
- To compute the intensity at P, we need
  - The unit normal vector **N**,
  - The unit vector **L**, from P to the light
  - The unit vector **V**, from P to the viewer

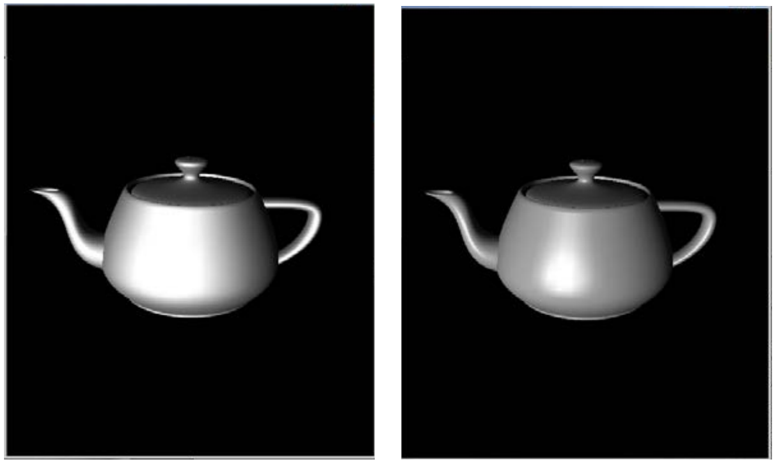


The Phong model for specular reflection

- I : specular reflection at P
- $I = I_p k_s \cos^n \quad = I_p k_s (\vec{R} \bullet \vec{V})^n$
- I_p: intensity of the light from source
- $0 \leq k_s \leq 1$: coefficient of specular reflection
- *n:* controls “shininess”

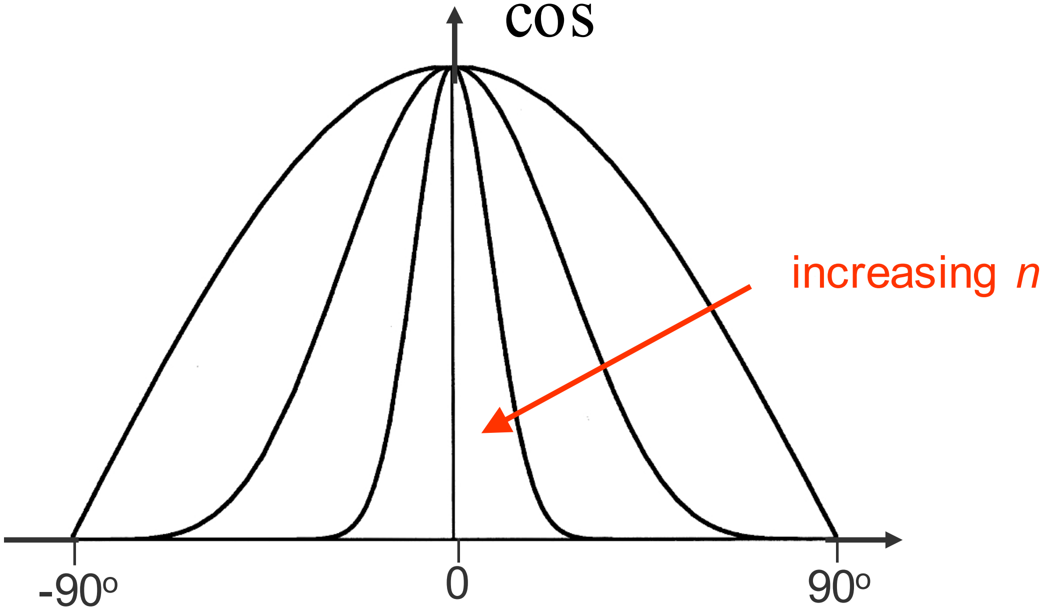






$\vec{R} = 2(\vec{L} \cdot \vec{N})\vec{N} - \vec{L}$ (why?)

The shininess coefficient



$n = 1$

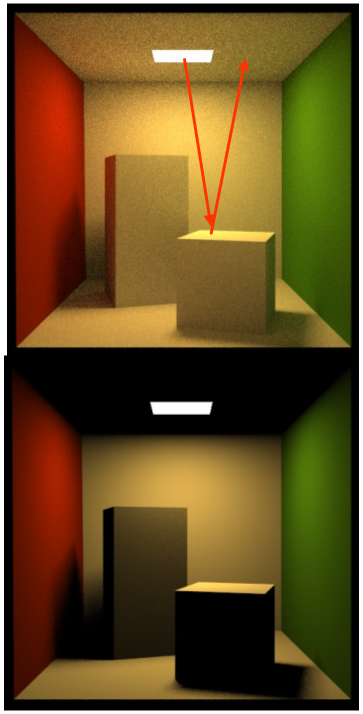
$n = 2$

$n = 4$

$n = 6$

Ambient light

- “Physical rules” are too simplified
- No indirect or global interaction of light



- A hack to overcome the problem: use “ambient light”

Ambient light specification

- Not situated at any particular point
- Spreads uniformly in all directions
- $I = k_a I_a$
- I_a : intensity of ambient light in the environment
- I : ambient light at a given point
- $0 \le k_a \le 1$: coefficient of ambient light reflection

k_a=0 k_a=0.5 k_a=1

A combined model
(The Phong local illumination model)

- The final model = diffuse + specular + ambient

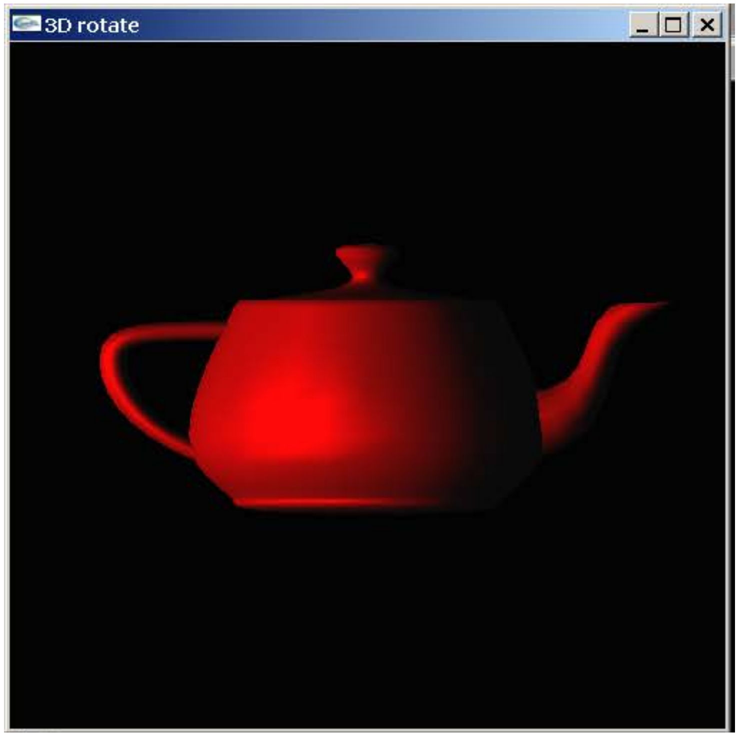
- $$I = I_p k_d (\vec{L} \bullet \vec{N}) + I_p k_s (\vec{R} \bullet \vec{V})^n + I_a k_a$$

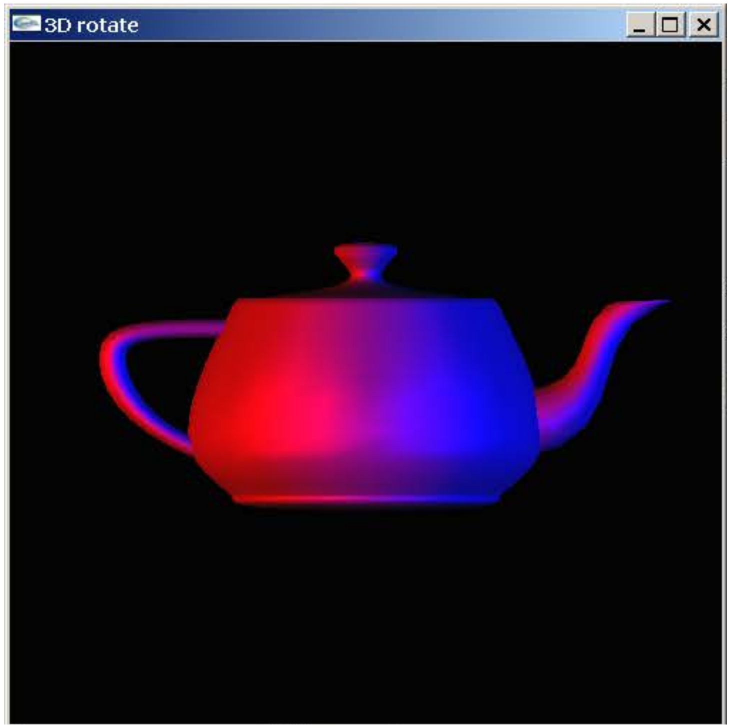
Example: two light sources

- Right Light=(1.0,0.0,0.0)
- Left Light=(1.0,1.0,1.0)

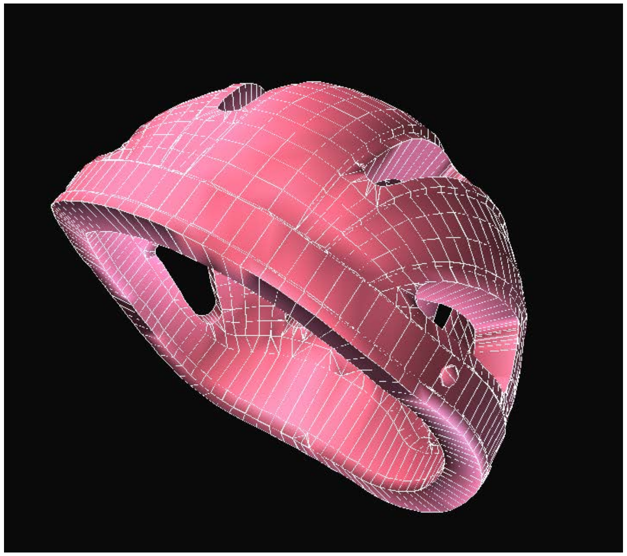
Multiple light sources

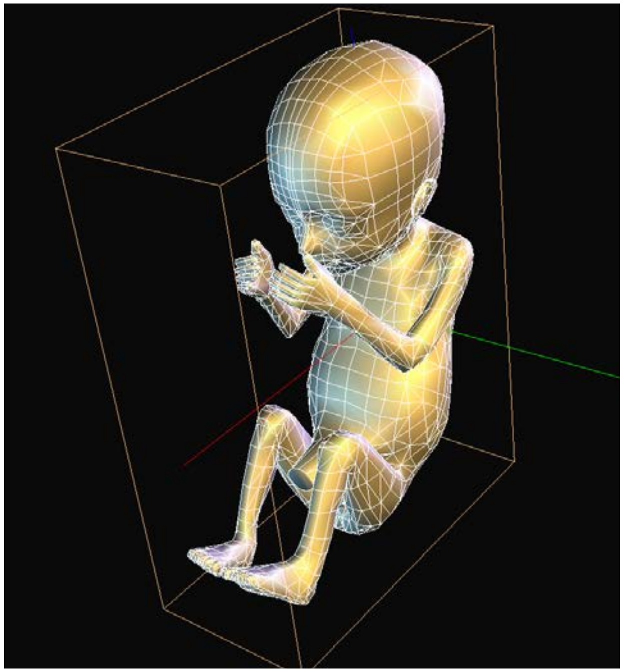
- The total reflection at p is the
  sum of all contributed intensities from all sources
- “Standard” OpenGL supports up to 8 light sources





Shading polygon meshes

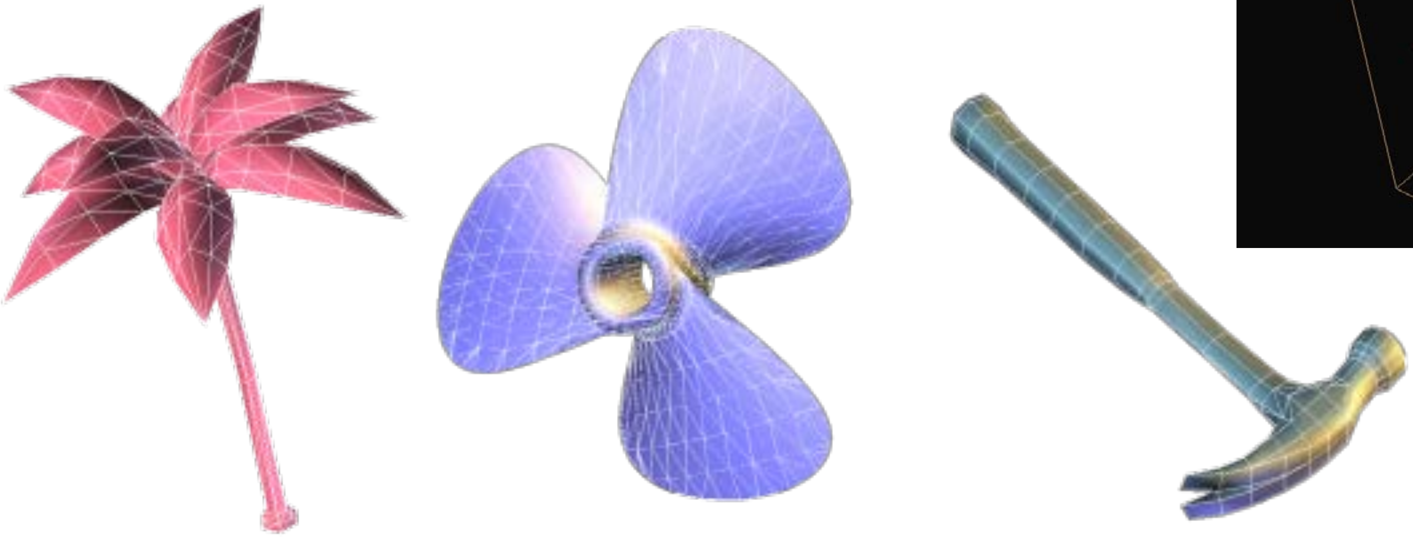




Brute-force idea:

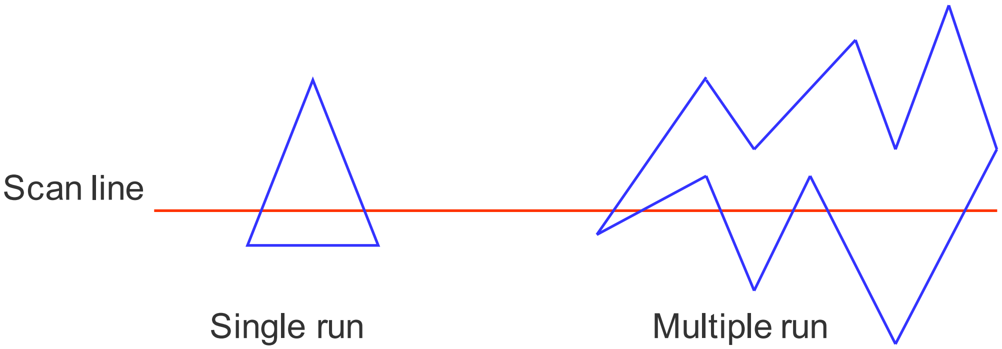
```
for each face in the mesh
  for each point on the face
      find normal at this point
      use Phong model to find the color
```

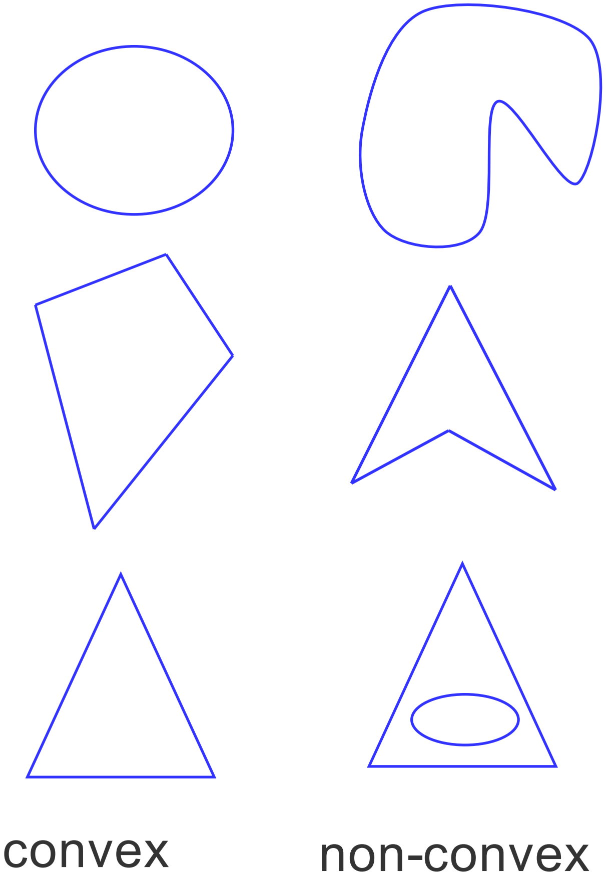
- ❖ These two steps require large a (relatively) amount of computations
- ❖ Interpolated polygon shading is a computationally efficient alternative



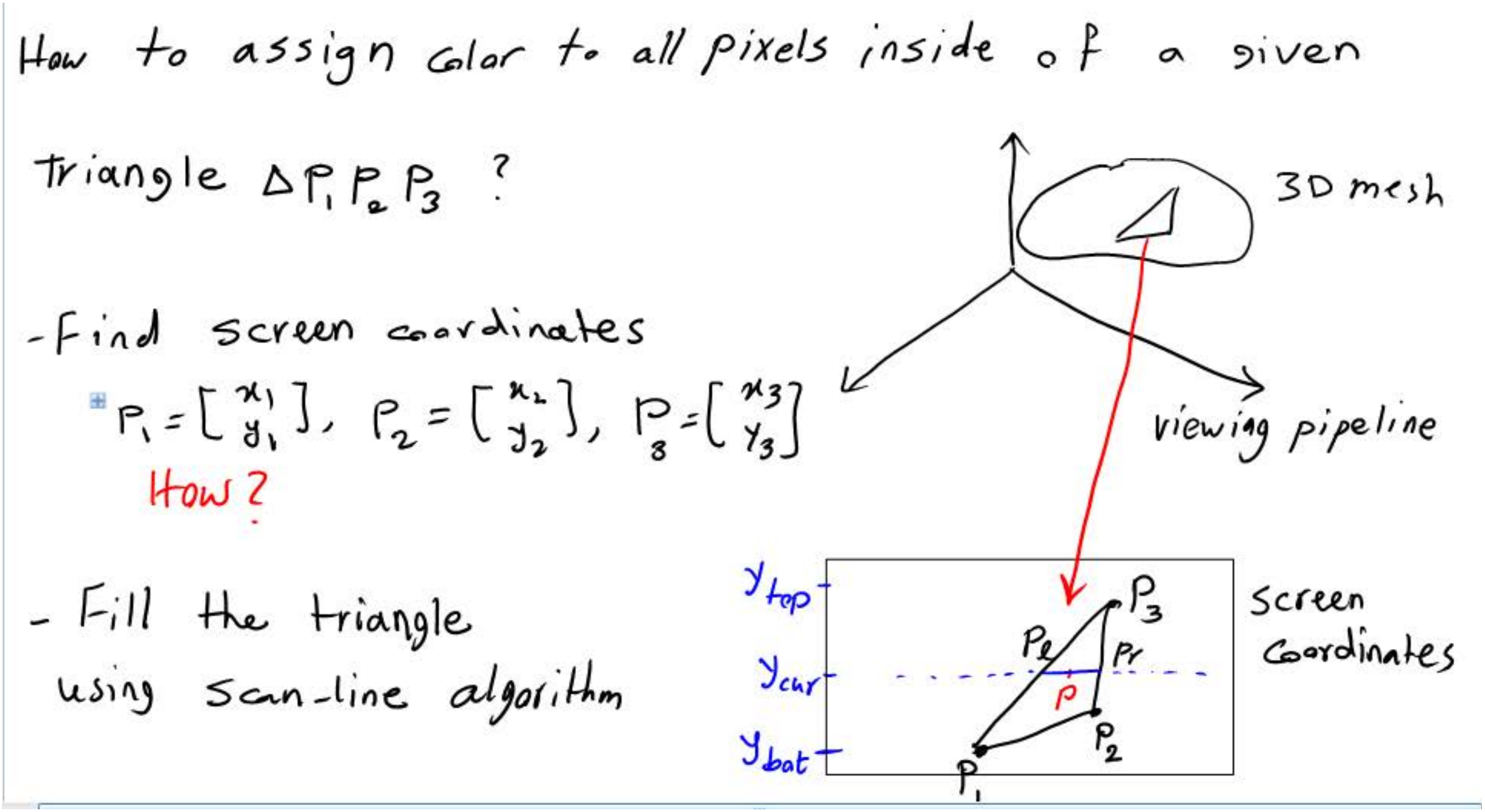
Scan-converting polygons

- Polygon- fill routine
- Convex polygons can be filled particularly efficiently
- Convex object definition





In which space should polygons be filled?



How to assign color to all pixels inside of a given triangle $\Delta P_1 P_2 P_3$?

- Find screen coordinates

$P_1 = \begin{bmatrix} x_1 \\ y_1 \end{bmatrix}$, $P_2 = \begin{bmatrix} x_2 \\ y_2 \end{bmatrix}$, $P_3 = \begin{bmatrix} x_3 \\ y_3 \end{bmatrix}$

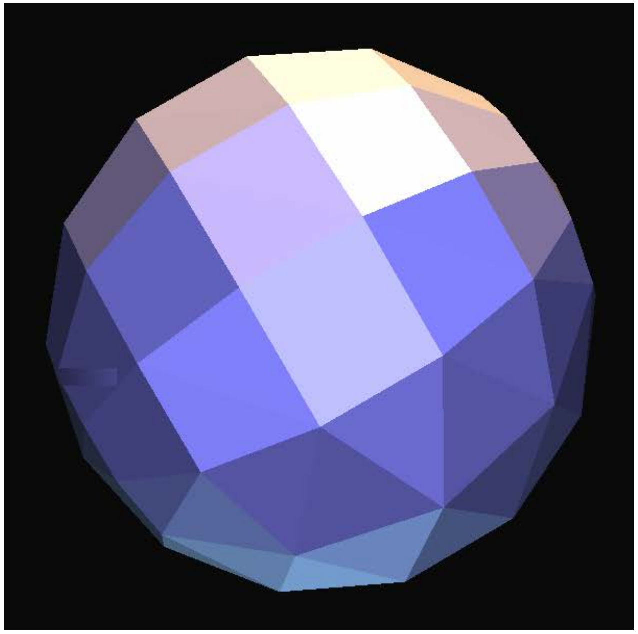
How?

- Fill the triangle using scan-line algorithm

Scan-converting convex polygons: Flat shading

```
for each face in the mesh {
    find color $c$ for the pixel at $(x,y)$
    for ($y=y_{bottom}$; $y<=y_{top}$; $y$++) {
        find $x_{left}$ and $x_{right}$
        for ($x=x_{left}$; $x<= x_{right}$; $x$++)
            set the color of fragment at $(x,y)$ to $c$
    }
}
```

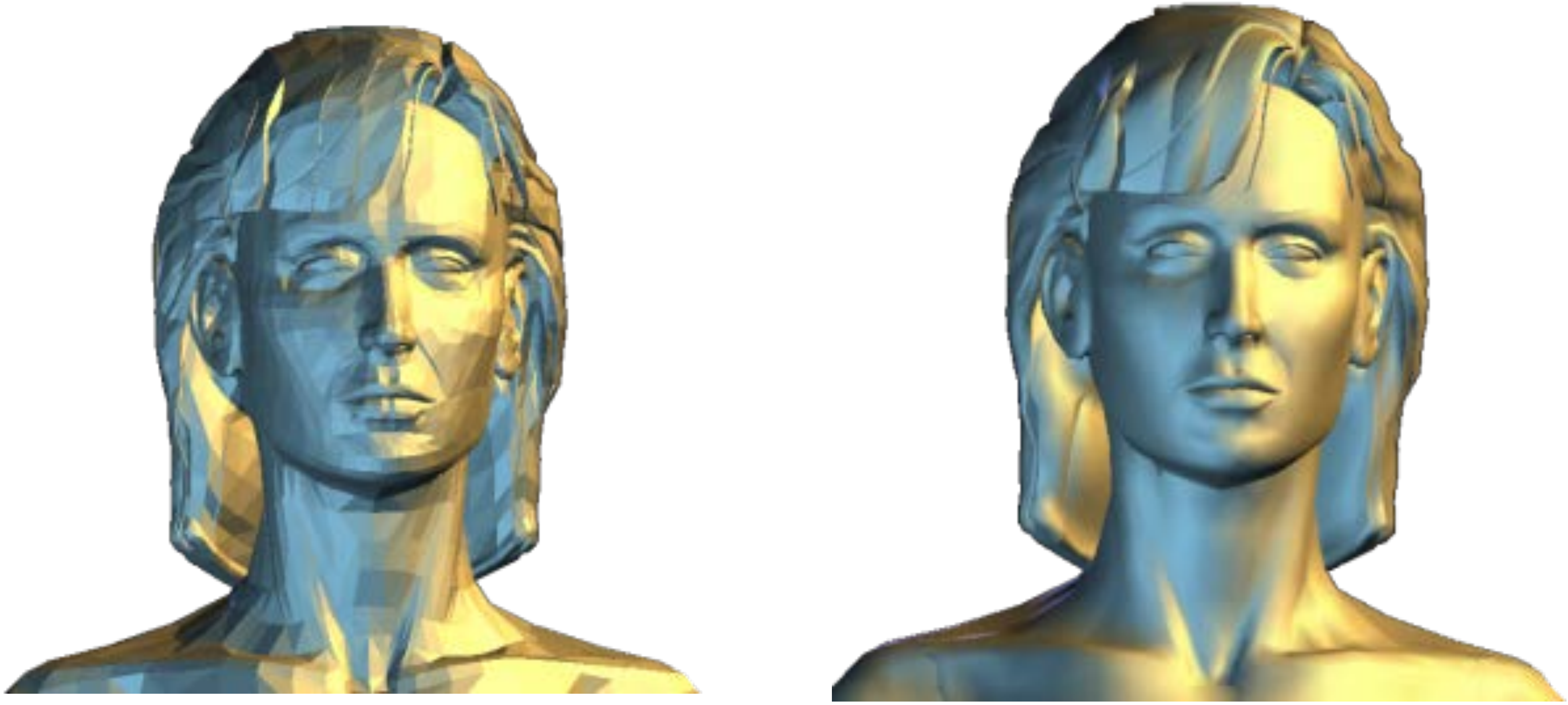
Flat shading



- Individual facets are visualized
- Same color for any point of the face
- OpenGL: glShadeModel(GL_FLAT)

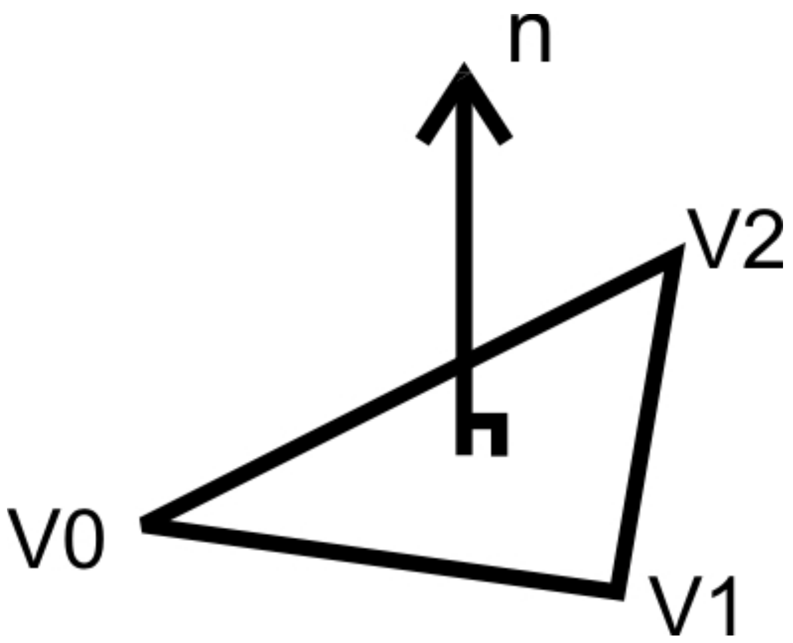
Flat versus smooth shading

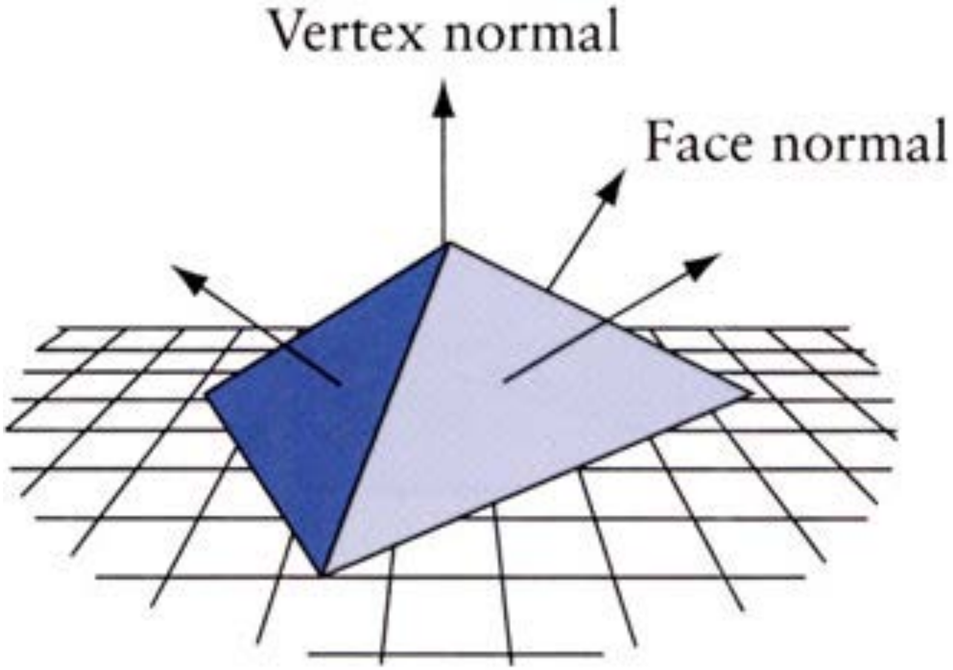
- Flat shading is particularly efficient
- Not suitable for smooth objects
  (Mach band effect)



Face vs. "vertex" normals

- For each triangle we can define a normal for the face
- For each vertex we an define a normal by interpolating normals of attached faces



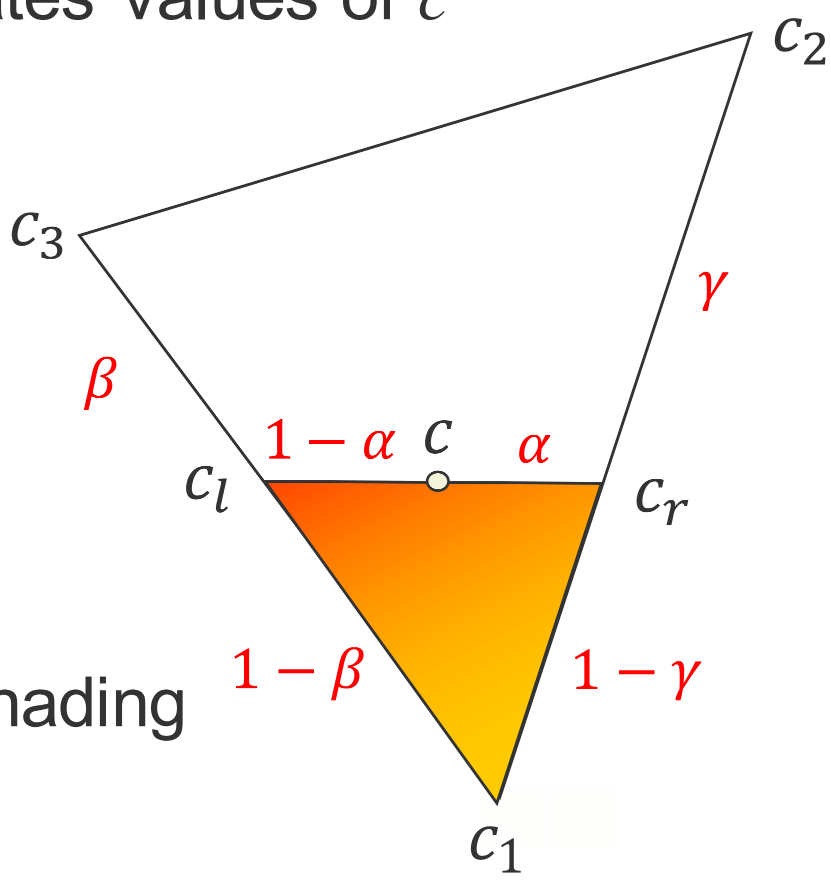


Smooth shading (Gouraud)

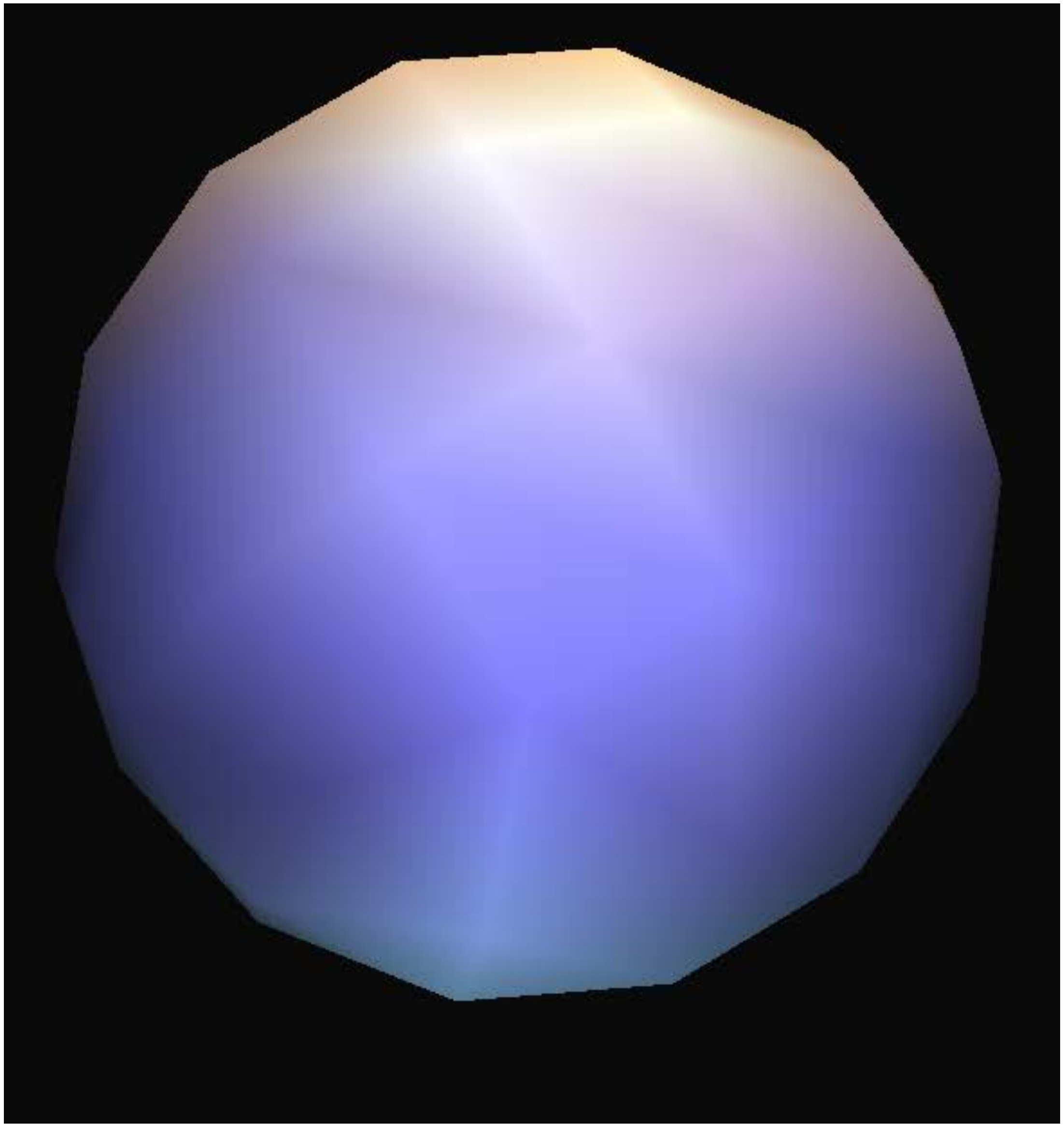
- Gouraud shading: interpolates values of c
- Bilinear interpolation
- $c_l = (1-\beta)c_1 + \beta c_3$

  $c_r = (1-\gamma)c_1 + \gamma c_2$

  $c = (1-\alpha)c_l + \alpha c_r$
- More expensive than flat shading



Gouraud shading



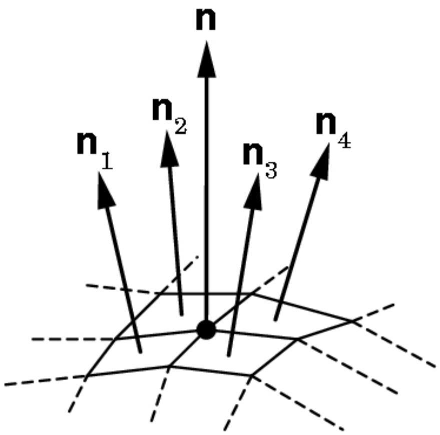
Toward the Phong interpolation: interpolating vertex normals

- Polygonal meshes don't have normal at the vertices
- But they (often) approximate a smooth underlying surface
- A simple estimate for vertex normal:

  the nomalized average of the normals of the faces

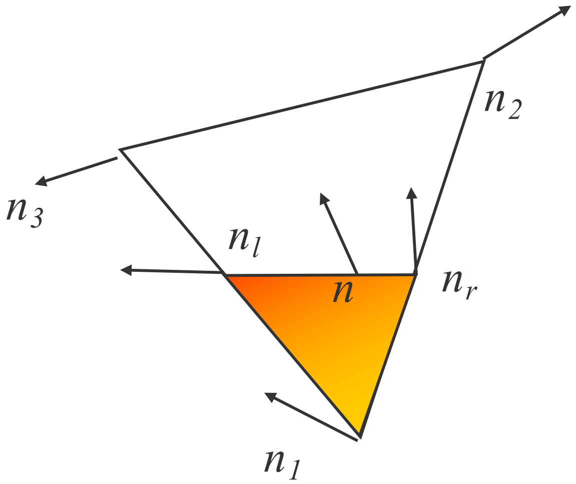
$$m = n_1 + n_2 + n_3 + n_4$$

$$n = \frac{m}{|m|}$$

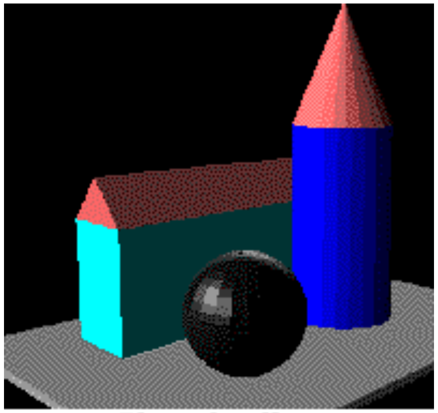


Phong shading (interpolation)

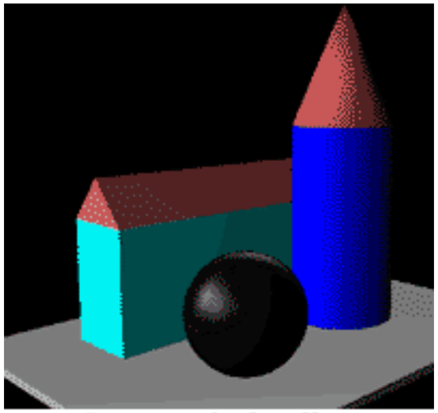
- Better realism for highlights
- Use normal of vertices to interpolate normal of interior points
- Linear interpolation of n_1 and n_3 $\Longrightarrow$ n_l
- Linear interpolation of n_1 and n_2 $\Longrightarrow$ n_r
- Linear interpolation of n_l and n_r $\Longrightarrow$ n
- Normalize n
- Drawback: relatively slow



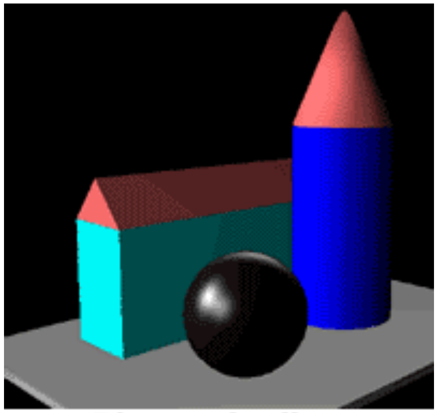
A comparison



Flat shading

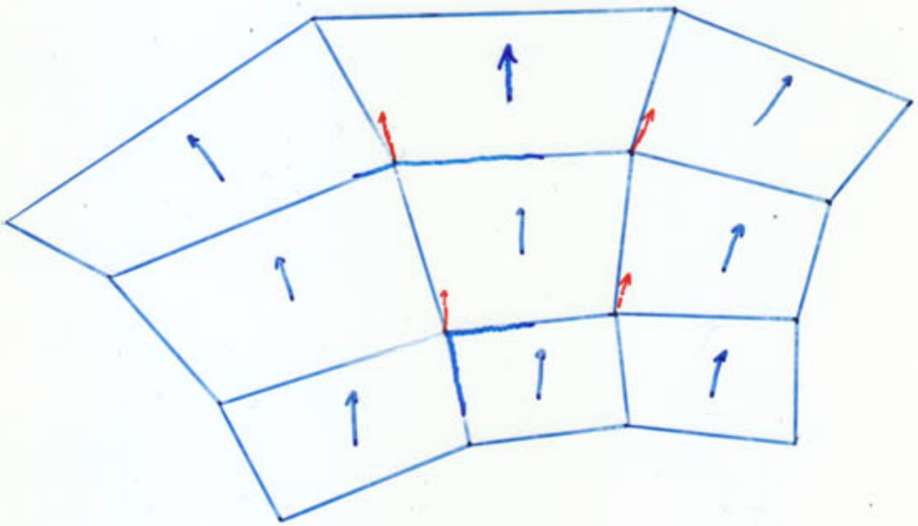


Gouraud shading



Phong shading

Shading: local illumination mode vs. interpolation

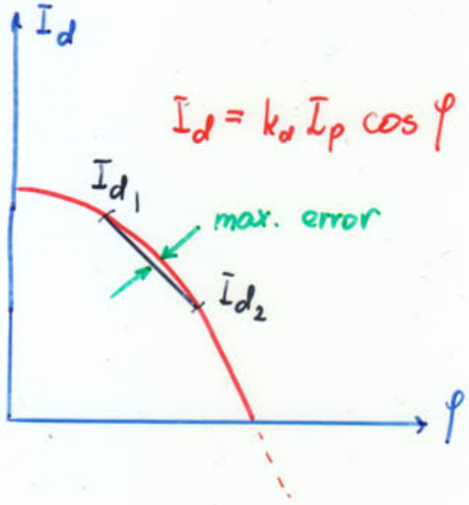


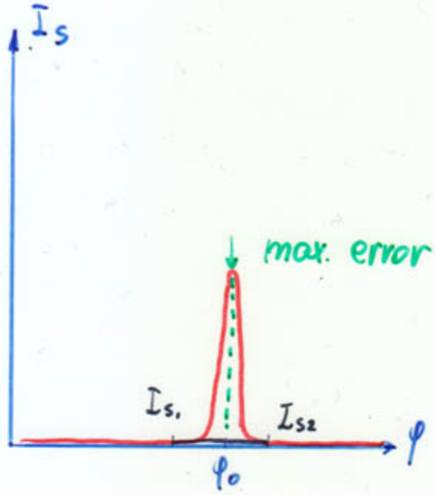
↑ average normal vector

Approaches:

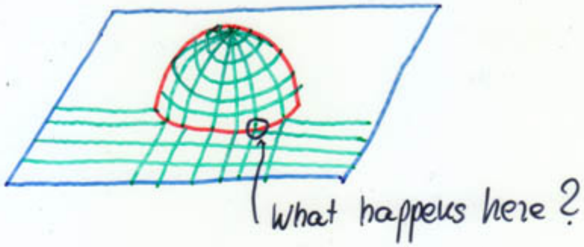
1) Shade each polygon separately (facettes easily perceivable)
2) Use average normal vectors at the vertices, interpolate in the color space, first along the edges, then along the scan lines (Gouraud)
3) Use average normal vectors at the vertices, interpolate in the space of normal vectors (Phong)

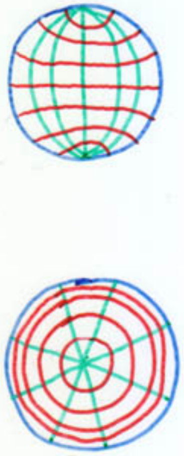
Why are we using Gouraud interpolation for diffuse shading and Phong interpolation for shading with specular reflection?

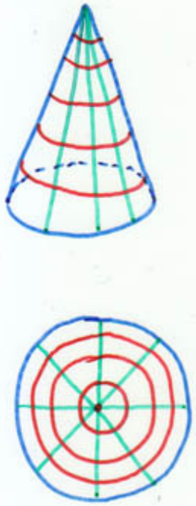




Pitfalls – what should be interpolated where




Sphere

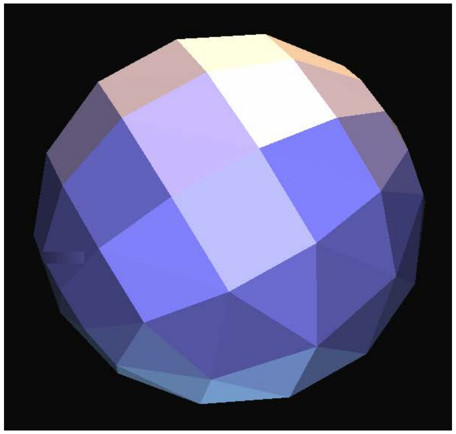

Cone


Bishop's hat

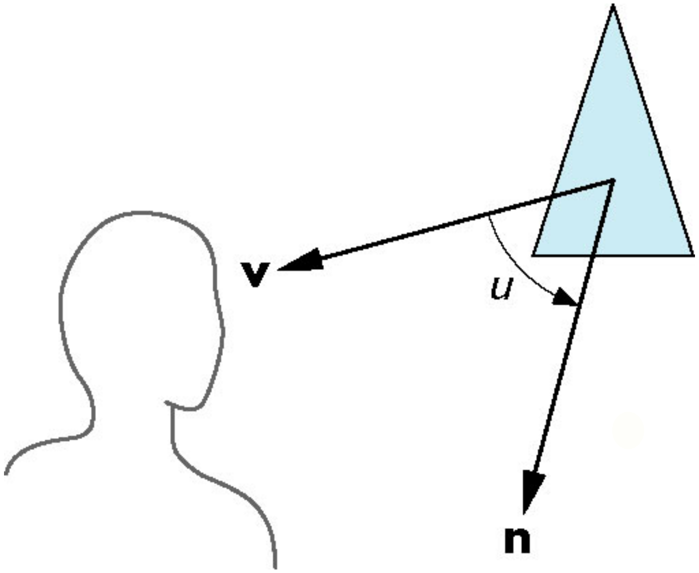
Hidden surface/line removal

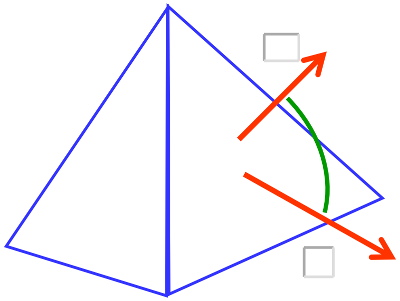
- Visible surface
  - Parts of scene that are visible from a chosen viewpoint
- Hidden surface
  - Parts of scene that are not visible from a chosen viewpoint

Back-face removal



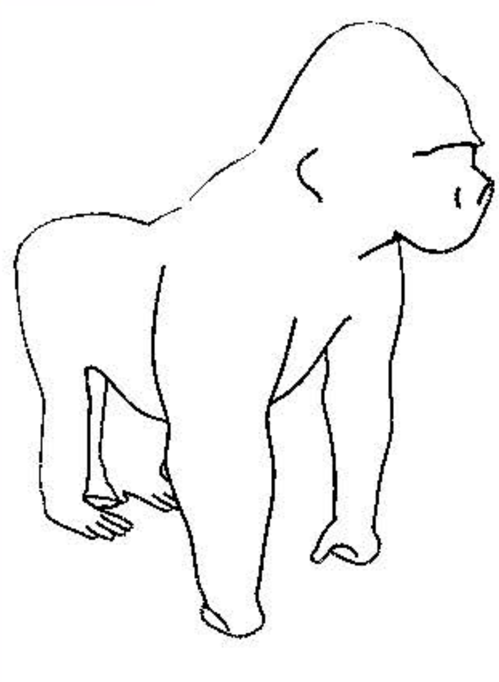
- Also called back-face culling
- We see a polygon if its normal is pointed toward the viewer
- Condition: $\cos\theta \geq 0 \; or \; n.v \geq 0$





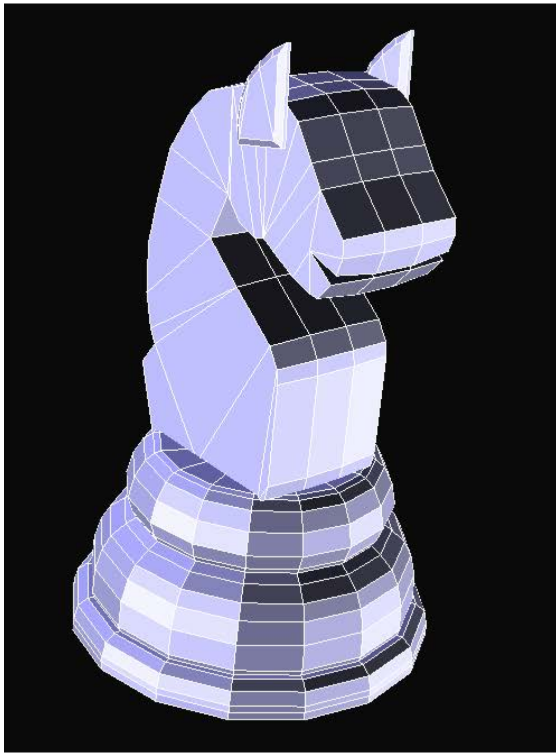
Digression: Silhouette (contour) extraction

- Silhouette lines are very important for visualizing objects(very useful in the traditional art)
- Any edge shared by a front-facing polygon and a back-facing polygon is a silhouette edge (but may be hidden)
- Sample application: NPR



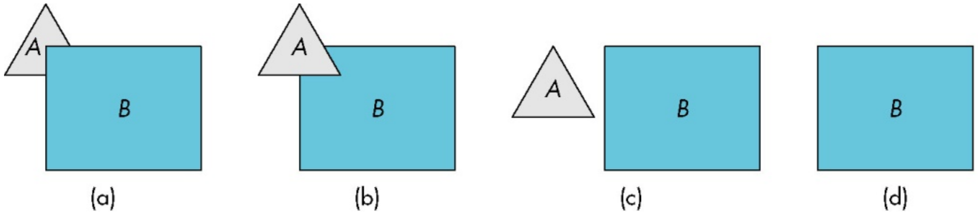
Is back-face removal enough?

- It fails for a non-convex surface
- It can’t recognize partly obscured faces



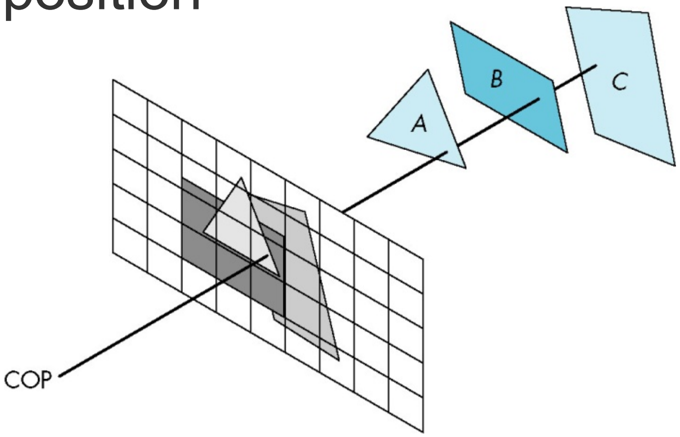
Hidden-surface algorithms

- Object-space



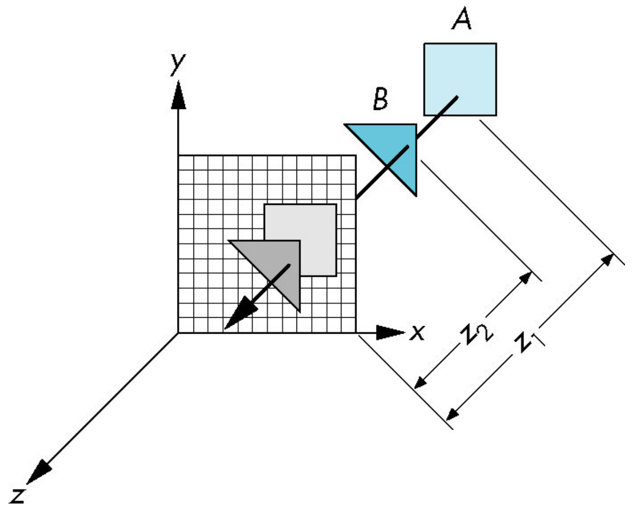
  - Comparison within real 3D scene
  - Works best for scenes that contain few polygons

- Image-space
  - Decide on visibility at each pixel’s position



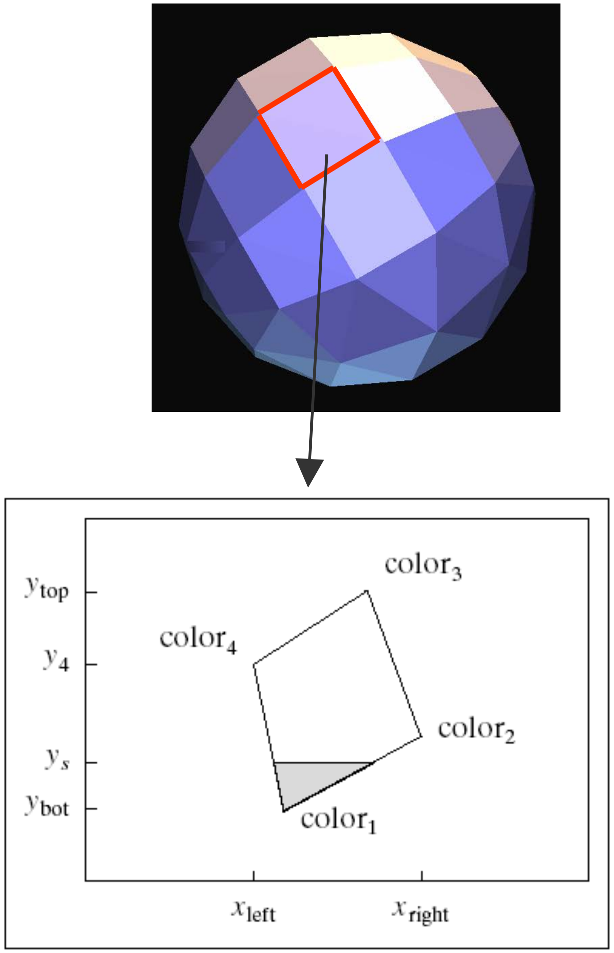
Z-buffer (depth-buffer)

- A commonly used image-space approach to hidden-surface removal
- Each location in the z-buffer contains the distance of the closest 3D point.
- Use the intensity (color) of the nearest 3D point for each pixel



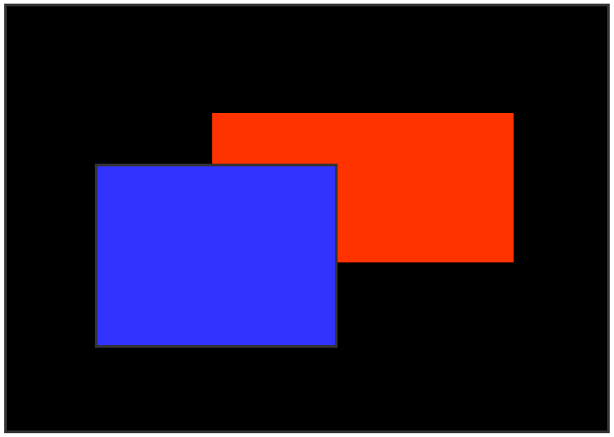
Recall polygon fill algorithm

```
for each face in the mesh
    for ($y=y_{bottom}$; $y<=y_{top}$; $y$++) {
        find $x_{left}$ and $x_{right}$
        for ($x=x_{left}$; $x<= x_{right}$; $x$++)
            find color $c$ for the pixel at ($x,y$)
    }
```



Screen space

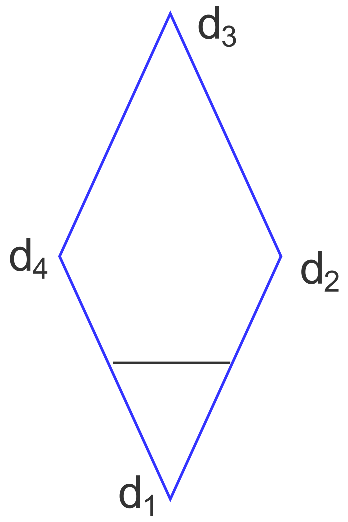
The Z-buffer algorithm



```
for all positions (x,y) on the screen
    frame(x,y) = background
    depth(x,y) = max_distance
end
for each polygon in the mesh
     for each point(x,y) in the polygon-fill algorithm
    ★ compute z, the distance of the corresponding 3D-point from COP
       if depth(x,y) > z    // current point is closer
          depth(x,y) = z
          frame(x,y) = I(p) //shading
       endif
      endfor
endfor
```

Some facts about the z-buffer algorithm

- After the algorithm
  - Frame buffer contains intensity values of the visible surface
  - z-buffer contains depth values for all visible points
- For the step ★ in algorithm
  - We know d_1, d_2, d_3 and d_4 from vertices of the mesh
  - Use linear interpolation for other points



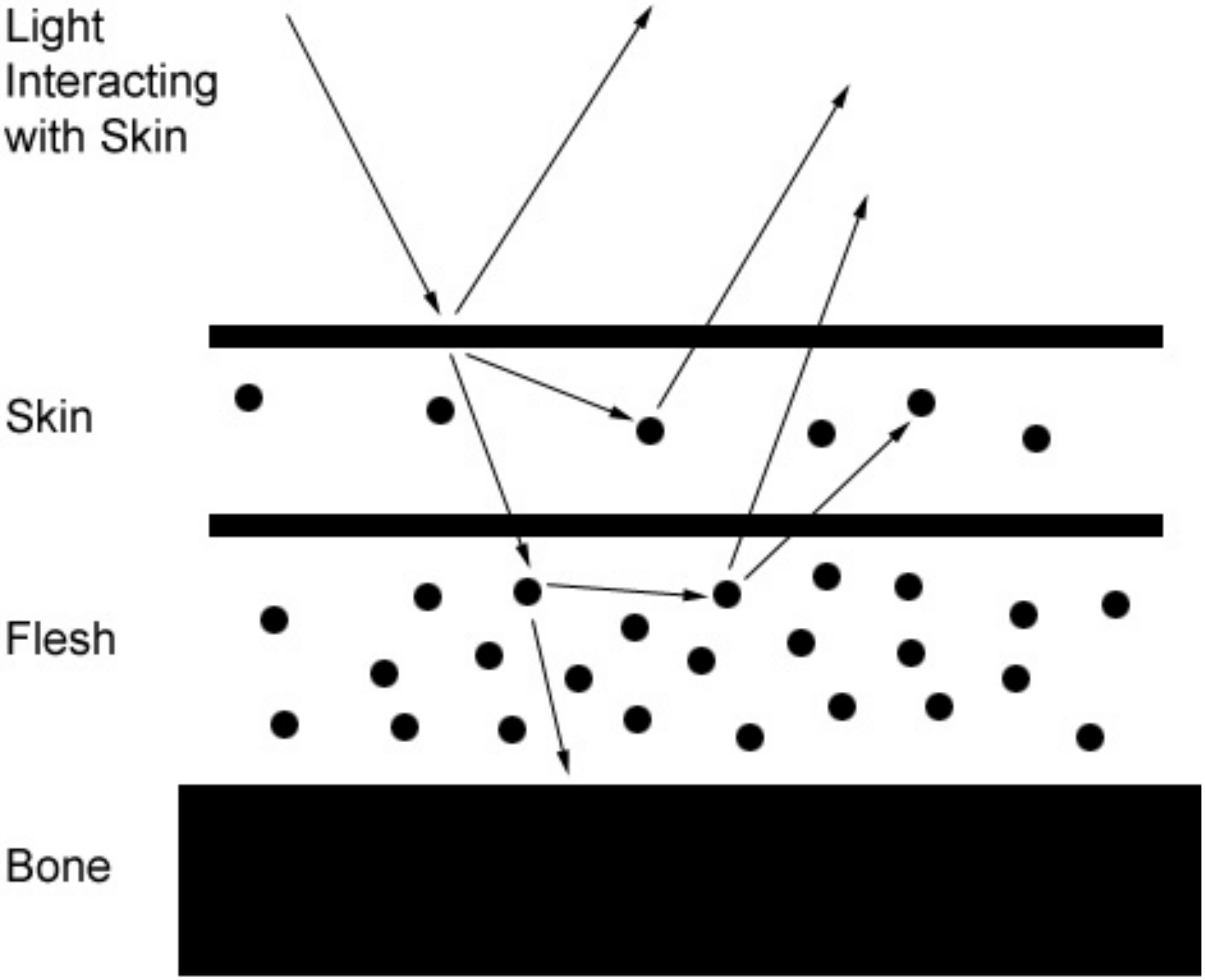
Beyond what we just learned...

Transparency + refraction

Caustics

Traslucency

Subsurface scattering



Subsurface scattering

Global illumination

Participating media