Local lighting and surface models

COM3404

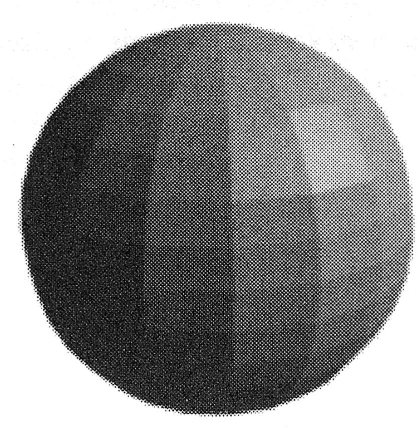
Richard Everson

School of Engineering, Computer Science and Mathematics
University of Exeter

R.M.Everson@exeter.ac.uk
http://www.secamlocal.ex.ac.uk/studyres/COM304

Outline

1. Camera and viewing
2. Rendering polygonal models with shading
3. Lights and shading
4. Interpolative shading



References

- Fundamentals of 3D Computer Graphics. Watt. Chapters 4, 5 & 6.
- Computer Graphics: Principles and Practice. Foley et al (1995). Chapters 15 & 16.

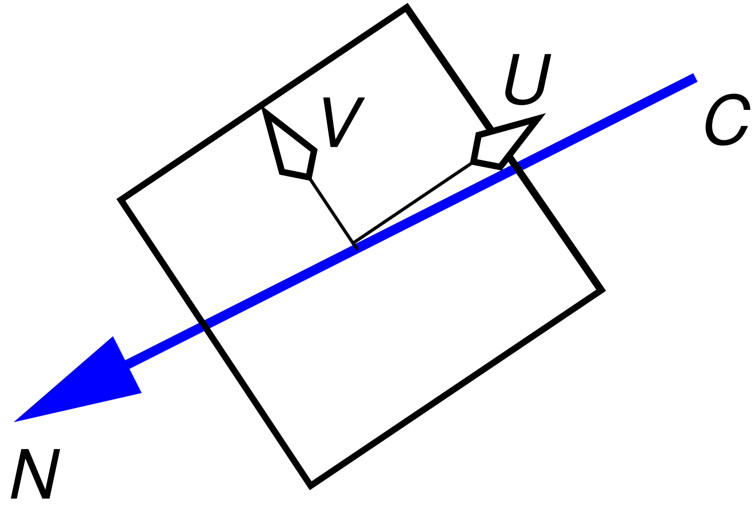
Camera and viewing

- Scene constructed in world coordinates independently of camera
- Location of the camera determines view of the scene

Camera parameters

Specified in world coordinates

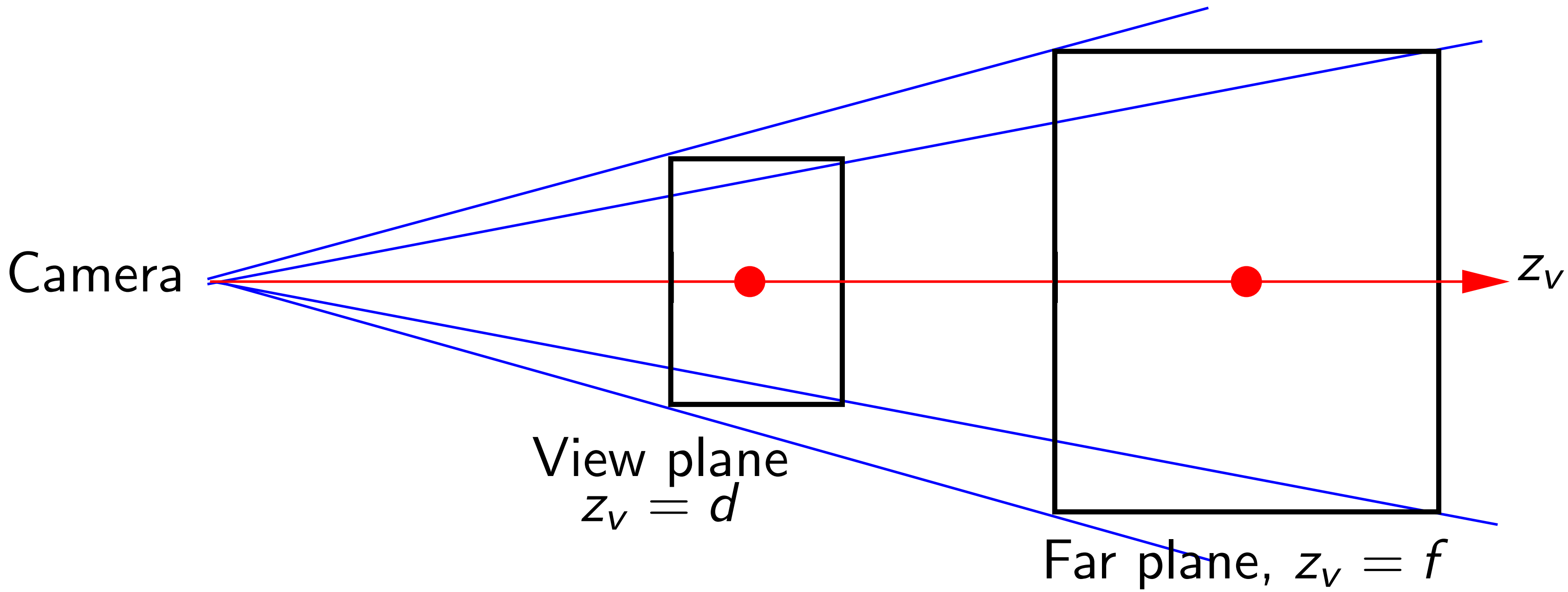
- Location, **C**
- Direction: **N** normal to viewing plane along line of sight
- Angle of viewing plane set by orientation of **U** and **V** vectors.
- Field of view: set by viewing frustrum
- Depth of field – the area around the focal length that is in focus.



Animation

Location and other viewing parameters may be animated like other objects in the scene.

View space to screen space



- Scene clipped to near and far clipping planes
- View plane and near clipping plane usually coincide.
- View plane extends from $-h \leq x_s, y_s \leq h$
- Transformation to screen (in view plane) accomplished by perspective transformation $\mathbf{T}_{pers}$

Viewing transformation

Transform points from world coordinates to viewing space

$$\begin{bmatrix} x_v \\ y_v \\ z_v \\ 1 \end{bmatrix} = \mathbf{T}_{view} \begin{bmatrix} x_w \\ y_w \\ z_w \\ 1 \end{bmatrix} = \mathbf{RT} \begin{bmatrix} x_w \\ y_w \\ z_w \\ 1 \end{bmatrix}$$

$$\mathbf{R} = \begin{bmatrix} U_x & U_y & U_x & 0 \\ V_x & V_y & V_x & 0 \\ N_x & N_y & N_x & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad \mathbf{T} = \begin{bmatrix} 1 & 0 & 0 & -C_x \\ 0 & 1 & 0 & -C_y \\ 0 & 0 & 1 & -C_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

View space operations

Back face culling

Remove any polygons that face away from the camera. If

- $\mathbf{n}$ is polygon normal
- $\mathbf{v}$ is vector from polygon centre to camera (line of sight vector)

then polygon is visible if

$$\mathbf{v} \cdot \mathbf{n} = |\mathbf{n}||\mathbf{v}| \cos\theta > 0$$

Clipping to viewing frustrum

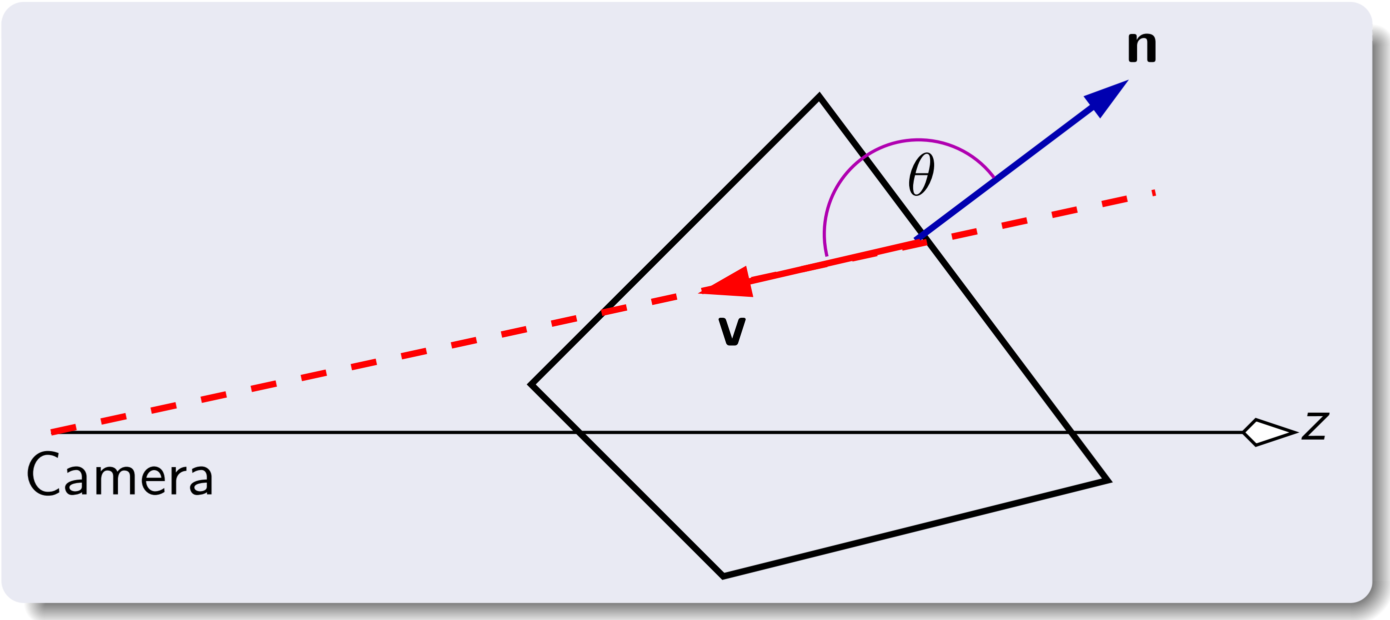
- Clip polygons that are too near $z_v < d$ or too far $z_v > f$
- Clip polygons that lie outside the planes

$$x_v = \pm \frac{hz_v}{d} \qquad y_v = \pm \frac{hz_v}{d}$$

- Can also be carried out more efficiently in screen space

Back face culling

Remove any polygons that face away from the camera.



- **n** is polygon normal
- **v** is vector polygon centre to camera (line of sight vector)

Polygon is visible if

$$\mathbf{v} \cdot \mathbf{n} = |\mathbf{n}||\mathbf{v}| \cos\theta > 0$$

Perspective transformation

Transform to *3D screen space* (x_s, y_s, z_s)

- x_s, y_s coordinates on the screen $-1 \leq x_s, y_s \leq 1$
- z_s is the apparent depth of the point.

$$\begin{bmatrix} x_s \\ y_s \\ z_s \\ 1 \end{bmatrix} = \mathbf{T}_{pers} \begin{bmatrix} x_v \\ y_v \\ z_v \\ 1 \end{bmatrix} = \begin{bmatrix} d/h & 0 & 0 & 0 \\ 0 & d/h & 0 & 0 \\ 0 & 0 & f/(f-d) & -df/(f-d) \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x_v \\ y_v \\ z_v \\ 1 \end{bmatrix}$$

Overall transform from world to screen coordinates is

$$\begin{bmatrix} x_s \\ y_s \\ z_s \\ 1 \end{bmatrix} = \mathbf{T}_{pers} \mathbf{T}_{view} \begin{bmatrix} x_w \\ y_w \\ z_w \\ 1 \end{bmatrix}$$

- Render point at (x_s, y_s)
- Use apparent depth to determine what is visible

Coordinates and transformations

$$\mathbf{x}_s = \mathbf{T}_{pers}\mathbf{T}_{view}\mathbf{x}_w$$
$$\mathbf{x}_s = \mathbf{T}_{pers}\mathbf{x}_v$$

World coordinates $\mathbf{x}_w = (x_w, y_w, z_w)$

Coordinates in which scene is constructed.

View space $\mathbf{x}_v = (x_v, y_v, z_v)$

Coordinates of the scene relative to the camera's point of view

Viewing transformation $\mathbf{T}_{view}$

Transformation describes the camera's view on the scene.

Maps world coordinates to view space.

3D screen space $\mathbf{x}_s = (x_s, y_s, z_s)$

Coordinates of points in the viewing plane, together with apparent depth.

Perspective transformation $\mathbf{T}_{pers}$

Maps view space to 3D screen space

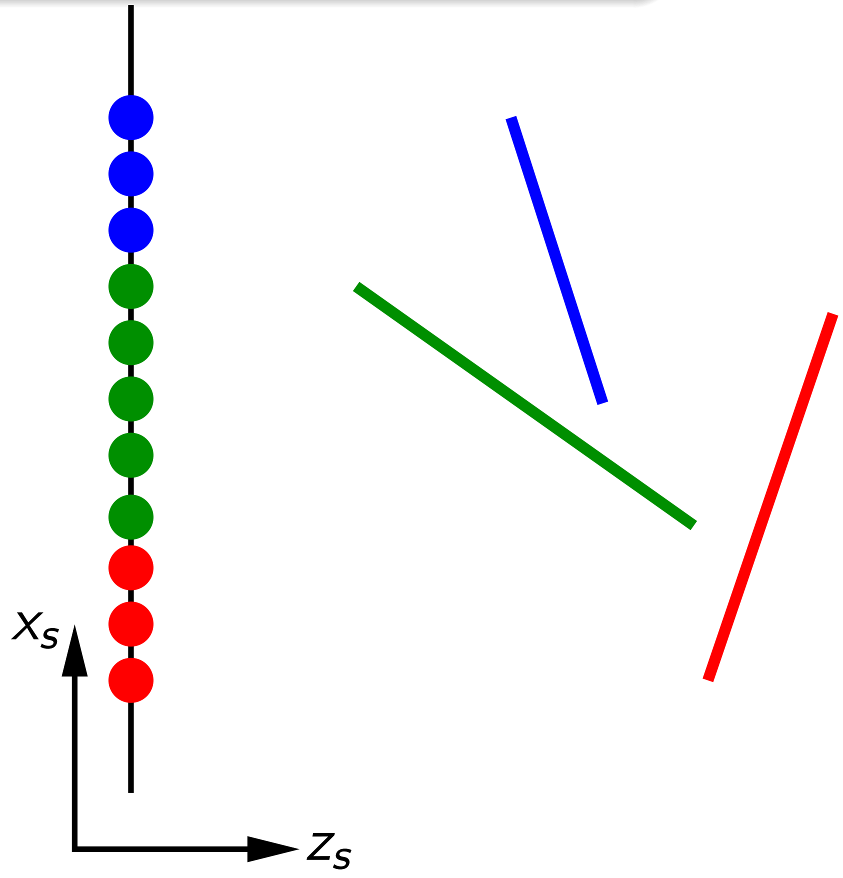
Rendering polygonal models with shading

Determine rendered colour/shade of each polygon from:

- light falling *directly* on it from a light source;
- surface properties of the polygon.

Visibility Many polygons may have the same screen coordinates (x_s, y_s) but different z_s: how to determine which is rendered?

Lighting How to determine the rendered colour and intensity of a polygon of a particular colour illuminated by a light of a particular colour located in a particular position?



Z-buffer algorithm

Z-buffer

Buffer with an entry for every pixel on the screen.
Contents of (x_s, y_s) element is the transformed surface that is closest to the screen at (x_s, y_s)

1. initialise all elements of Z_{buf} to max-depth

2. foreach polygon P in the model
3.   transform vertices of P to 3D screen coordinates (x_s, y_s, z_s)
4.   if $z_s < Z_{buf}(x_s, y_s)$ for a vertex
5. $Z_{buf}(x_s, y_s) = z_s$
6.   shade the polygon (in screen coordinates)
               by interpolating vertex shades
7.   end
8. end

Z-buffer

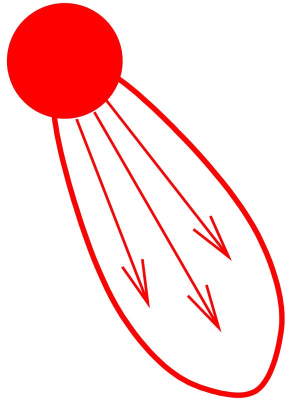
- Z-buffer usually supported in hardware
- Quality of rendering is dependent on resolution of buffer; 20-32 bits is usual. Depth (z) values scaled to use full range of Z-buffer.
- Polygons can be processed in any order.
- Independent of model representation (CSG, volumetric, polygonal, etc).
- Scan-line versions exist: less memory, but considerably more complex.

- Cannot cope with transparent or partially transparent polygons, because if the transparent polygon is at the front a list of those behind it must be kept.

Types of light

Spot lights

- Directed
- Intensity decays away from source
- Parameters: location, direction, colour, intensity, decay, drop-off

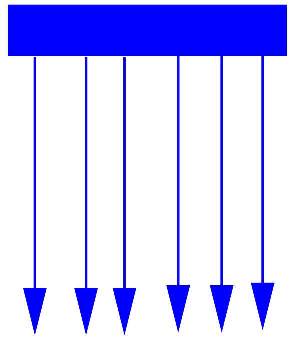


Area or rectangular light

- Rectangular beam of parallel rays
- Parameters: location, colour, intensity, direction

Ambient light

- Simulates the overall light in a space
- Not directed or located
- Parameters: colour, intensity



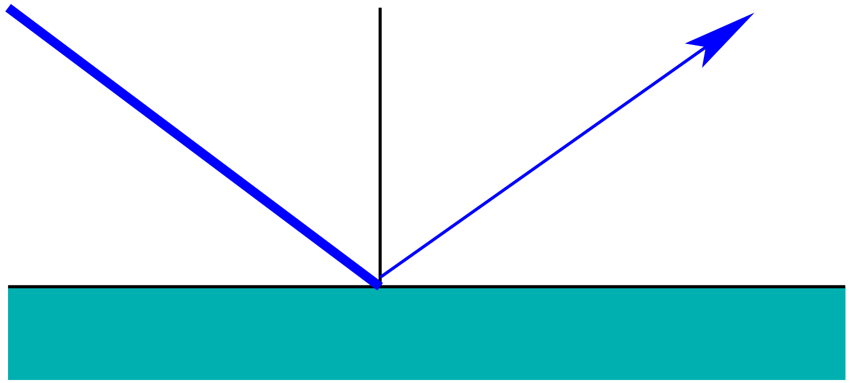
Directional light

- Located at infinity
- Illuminates scene uniformly from one direction
- Used to simulate sunlight

Specular reflection

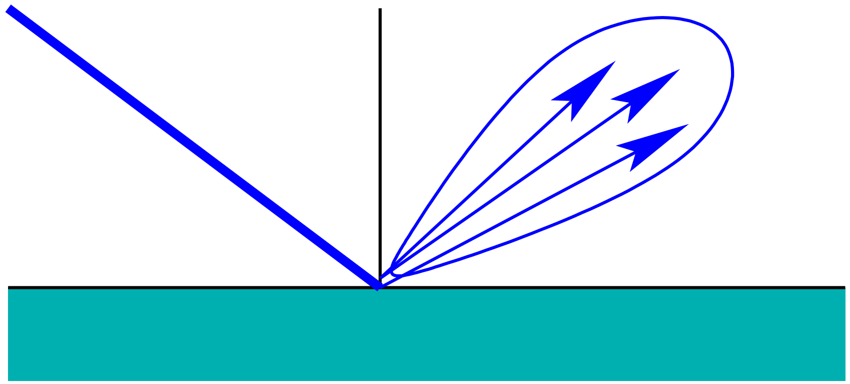
Perfect specular reflection

- Reflection from a perfect mirror.
- Incident and reflected light make equal and opposite angles with surface normal.



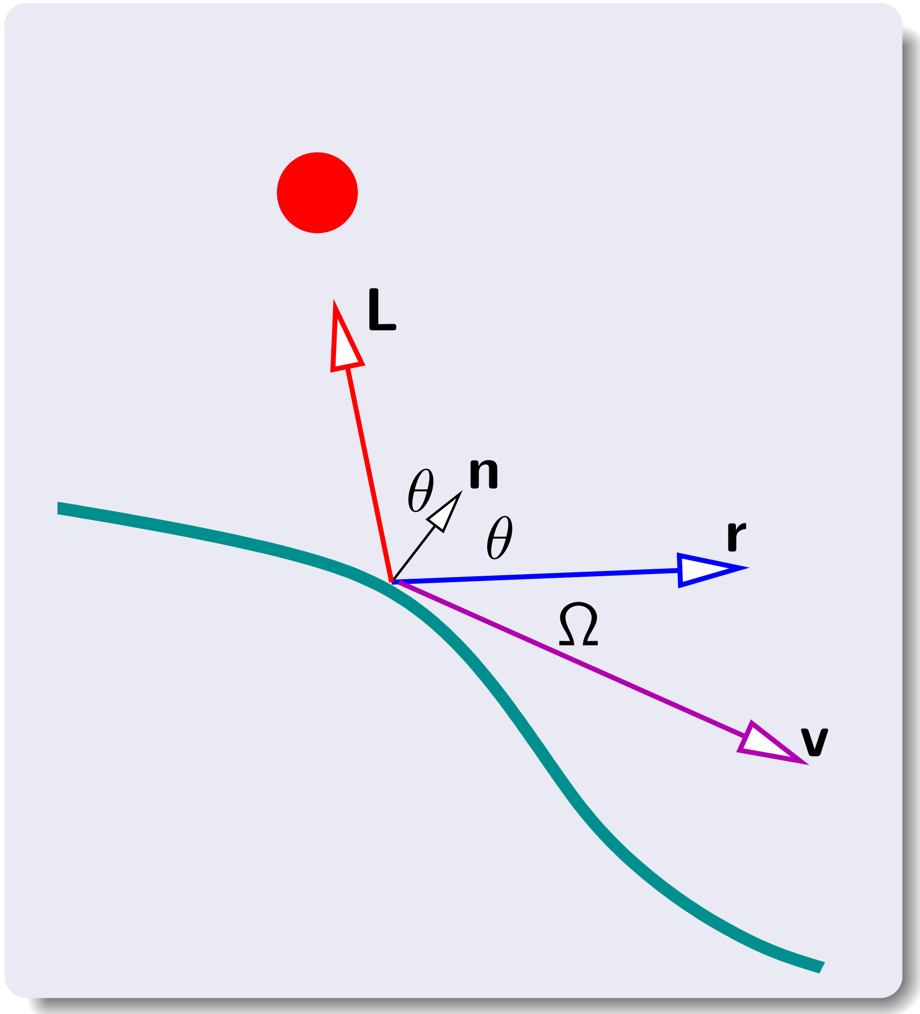
Imperfect specular reflection

- Some light is scattered away from principal reflected direction.



Reflection geometry

- I incident intensity
- I_s specular reflected intensity
- **r** 'mirror' direction
- **v** viewing direction
- **L** direction to light
- **n** surface normal



Specular reflection

Model specular reflection as

$$I_s = I(\mathbf{r} \cdot \mathbf{v})^n = I(cos\Omega)^n$$

$n \rightarrow \infty$ for perfect specular reflection

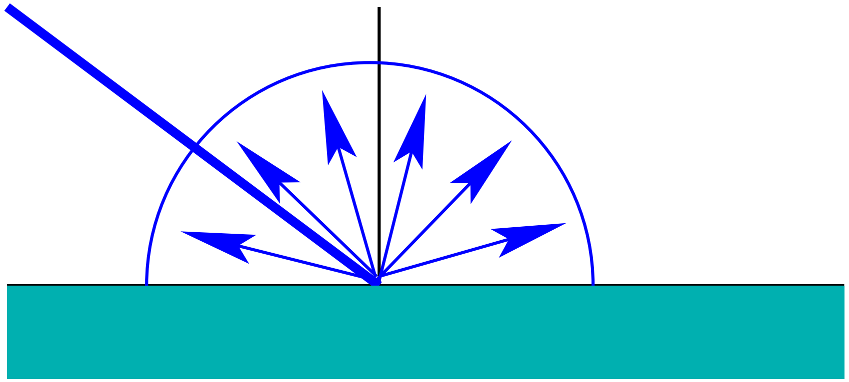
Imperfect specular reflection from two sources

Diffuse reflection

- Light reflected with equal intensity in all directions
- Modelled as

$$I_d = I \cos\theta = I\mathbf{n} \cdot \mathbf{L}$$

where **n** is the direction of the surface normal.



Phong reflection model

Model the combined contributions to light intensity of a surface as

$$
\begin{aligned}
I_r &= k_a I_a + k_s I_s + k_d I_d \\
&= k_a I_a + I[k_s(\mathbf{r} \cdot \mathbf{v})^n + k_d \mathbf{n} \cdot \mathbf{L}]
\end{aligned}
$$

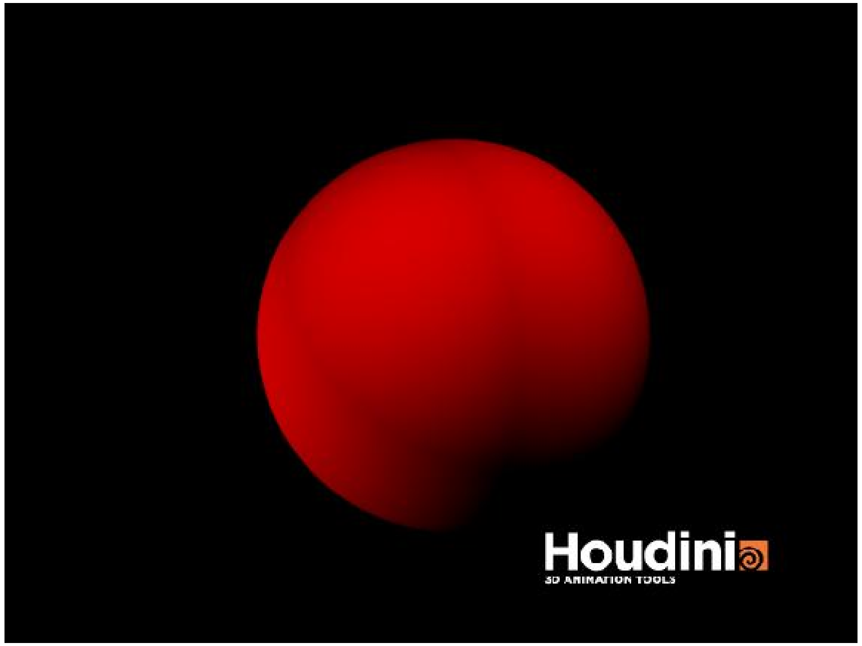
where

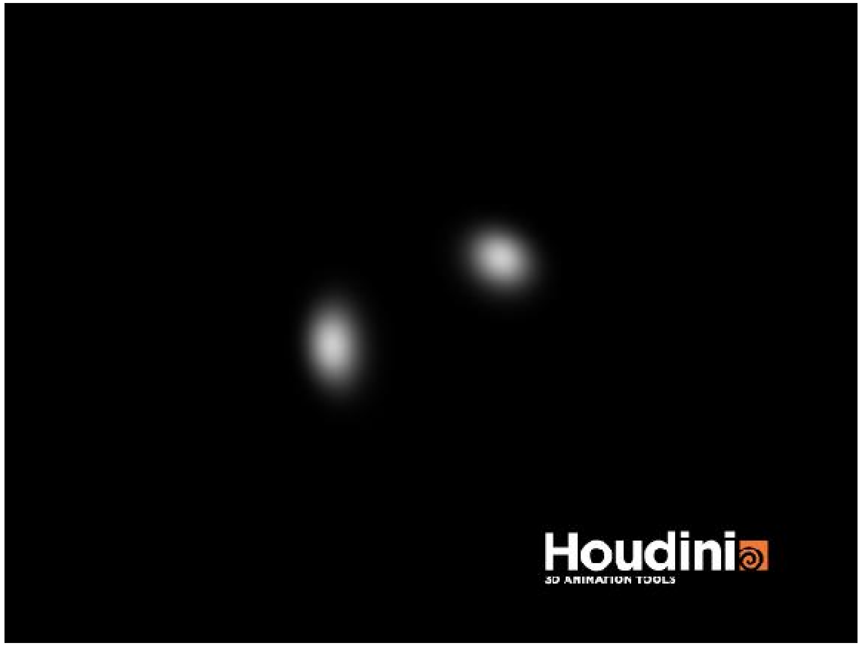
$$k_s + k_a \leq 1$$

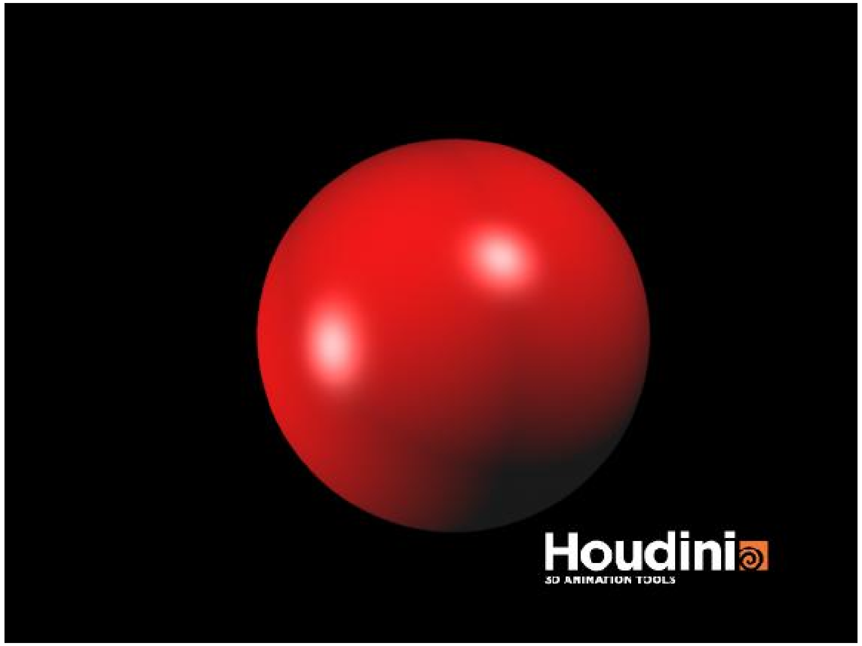
set characteristics of the surface.

- Reflected light from all sources must be summed.
- Each R, G, B component is treated separately.
- Model is a *local* model: light is not reflected from surface to surface.
- Light source itself is taken as a point source at infinity, but the spatial variation of the light can be modelled:
  $$I = I_0(\cos\phi)^m = I_0(-\mathbf{L} \cdot \mathbf{L}_s)^m$$
  where $\mathbf{L}_s$ is the principal direction of the light source and ϕ is the angle between $\mathbf{L}$ and $\mathbf{L}_s$

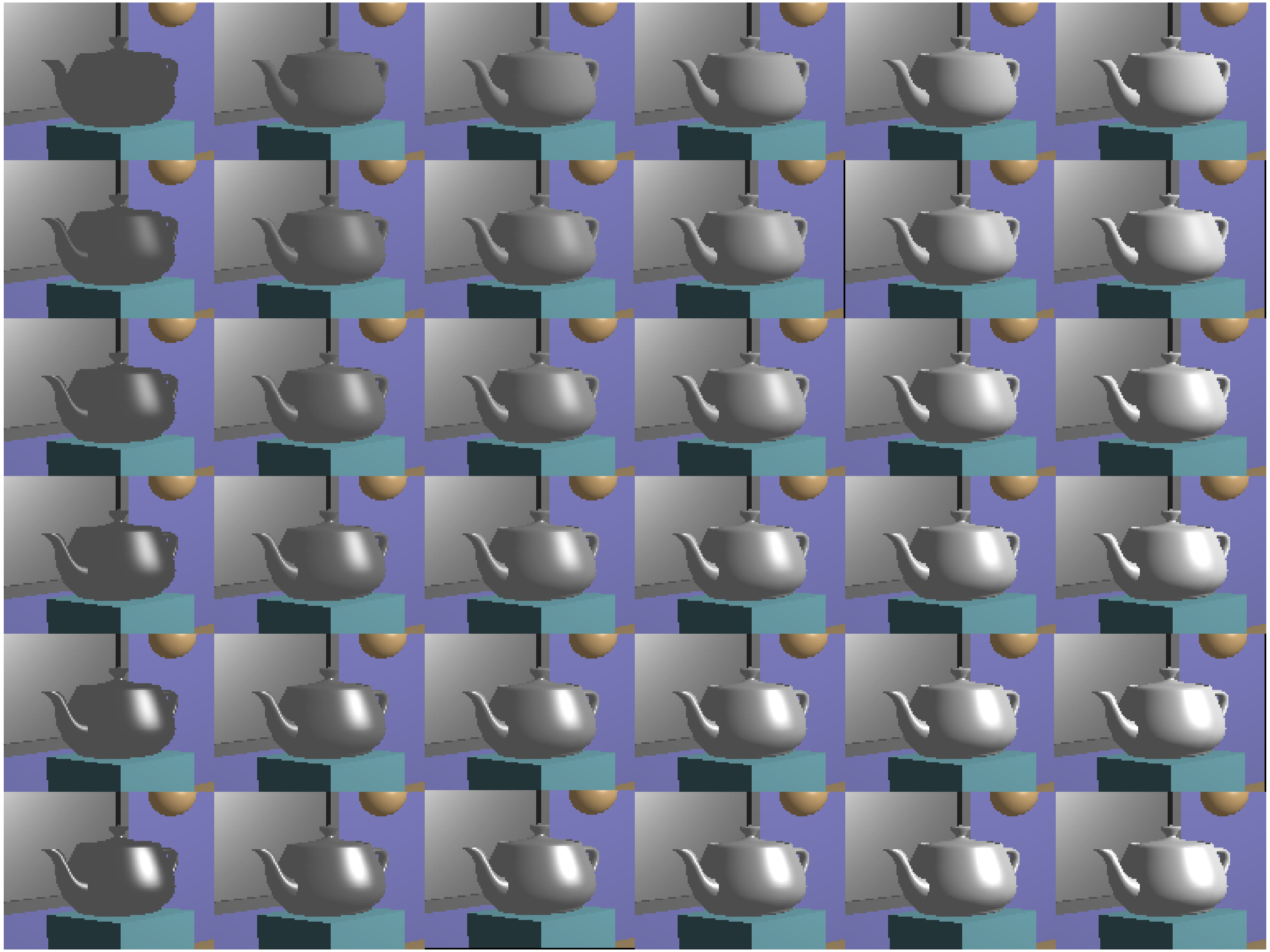
Phong reflection model







Phong reflection



Horizontally $K_s = 0, 0.2, 0.4, 0.6, 0.8, 1.0$; Vertically $K_d = 0, 0.2, 0.4, 0.6, 0.8, 1.0$; $K_a = 0.7, n = 10$

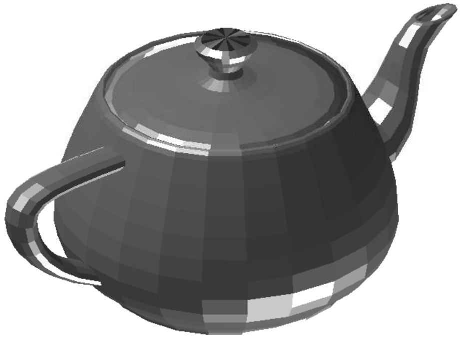
Interpolative shading

1. Use Z-buffer algorithm to determine screen locations of vertices
2. Shade polygons in *screen space* by

   Flat shading Shade determined by polygon normal and colour at ‘centre’

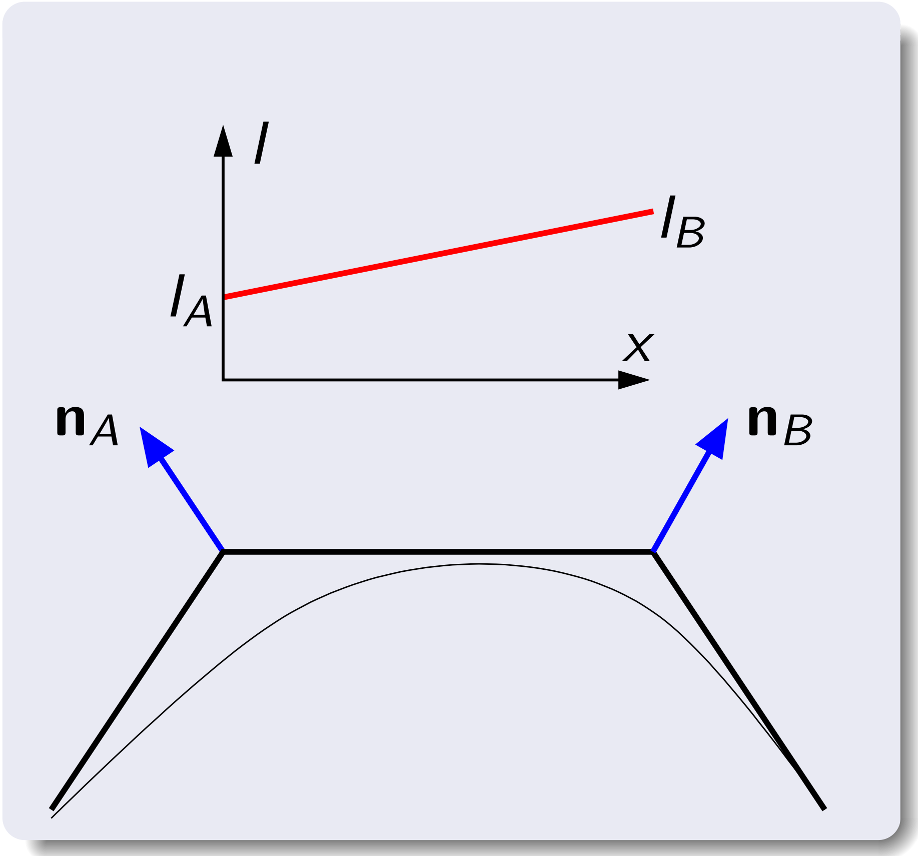
   Gouraud shading Calculate shades at vertices and interpolate to interior pixels

   Phong shading Interpolate normal direction to interior pixels and then calculate shade



Gouraud shading

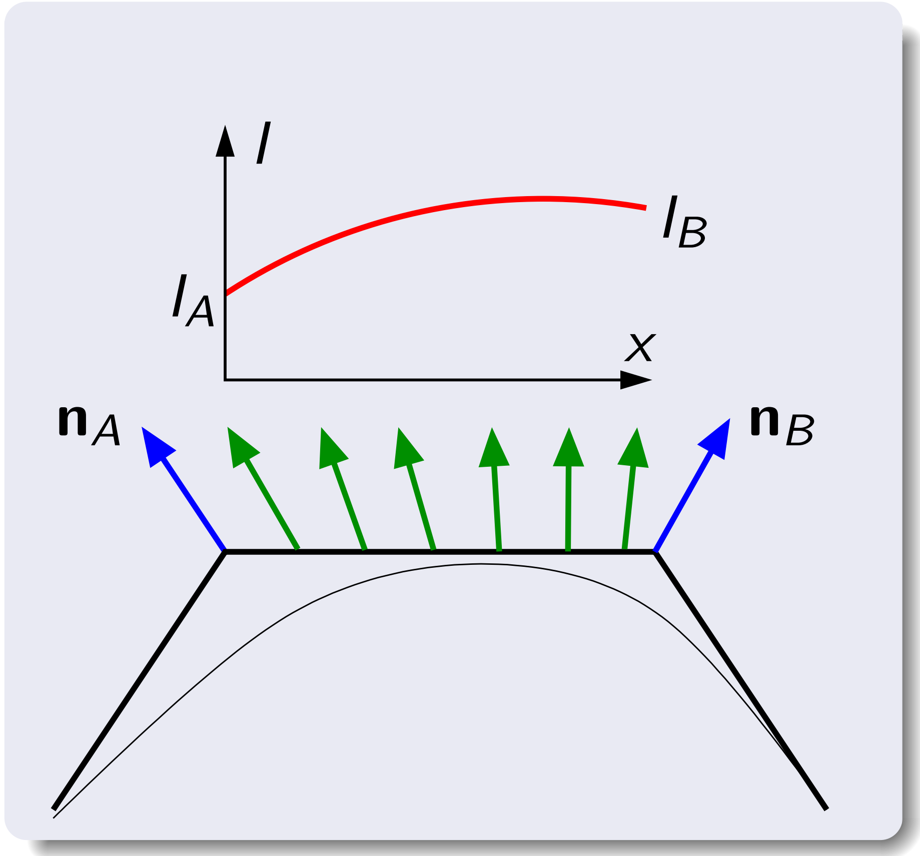
- Calculate vertex normals by (possibly weighted) averaging of adjacent polygons
- (Bi)linearly interpolate intensity between vertices



Efficient scan-line implementation of interpolation.

Phong shading

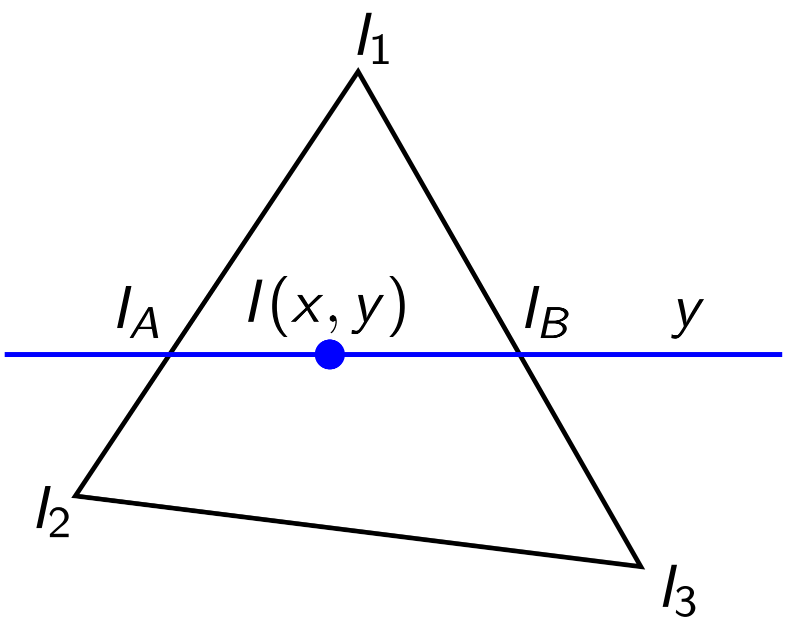
- Calculate vertex normals by (possibly weighted) averaging of adjacent polygons
- (Bi)linearly interpolate *normals* between vertices

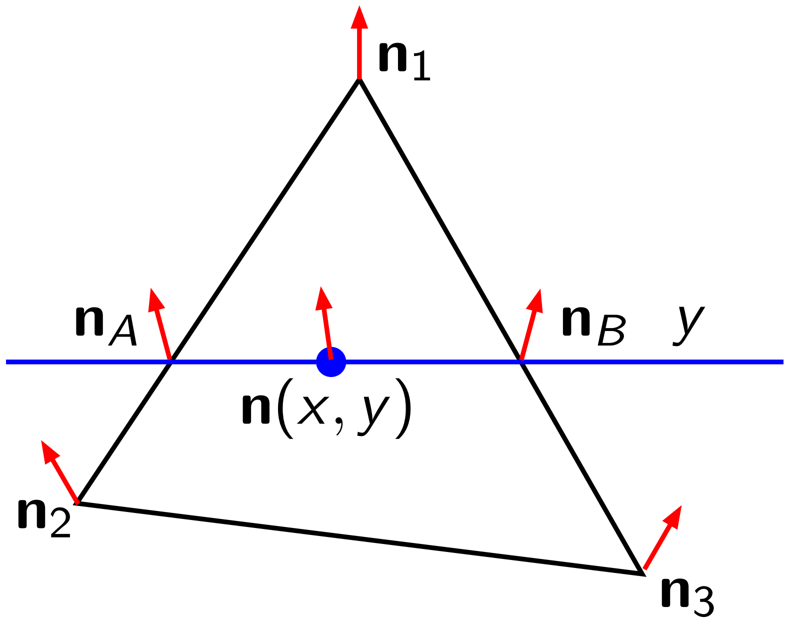


Interpolation of normals restores some 'curvature'

Scanline interpolation

- Intensities or normals interpolated from vertices along vertices
- Interpolate along scanline from edges





$$I_A = I_{Aprev} + \Delta_{2,1}$$
$$I_B = I_{Bprev} + \Delta_{3,1}$$
$$I(x_i, y) = I(x_{i-1}, y) + \Delta_x$$

$$\mathbf{n}_A = \mathbf{n}_{Aprev} + \boldsymbol{\Delta}_{2,1}$$
$$\mathbf{n}_B = \mathbf{n}_{Bprev} + \boldsymbol{\Delta}_{3,1}$$
$$\mathbf{n}(x_i, y) = \mathbf{n}(x_{i-1}, y) + \boldsymbol{\Delta}_x$$

Z-buffer algorithm

- for all x, y
- $\quad Z_{buf}(x, y) := \text{max-depth}$

- for each polygon
- $\quad$ Convert to edge-based representation in screen coordinates

- for $y := y_{min}$ to y_{max}
- $\quad$ for each segment in $EdgeList[y]$
- $\quad\quad$ Interpolate $X_{left}, X_{right}, Z_{left}, Z_{right}, \mathbf{n}_{left}, \mathbf{n}_{right}$ from segment ends
- $\quad\quad$ for $x := X_{left}$ to X_{right}
- $\quad\quad\quad$ interpolate z and $\mathbf{n}$
- $\quad\quad\quad$ if $z < Z_{buf}(x, y)$
- $\quad\quad\quad\quad Z_{buf}(x, y) := z$
- $\quad\quad\quad\quad F_{buf}(x, y) := shading(\mathbf{n})$

Shading

Wireframe Just draw the edges.

- Very fast, but not beautiful.

Flat Use the polygon normal.

- Fast, avoids interpolation.

Gouraud

- Basic shading, good for diffuse reflections.
- Poor rendering of highlights and specular reflections.

Phong Interpolate normals.

- Highest quality without volumetric rendering
- 4-5 times slower than Gouraud.

Phong and Gouraud

- Use Phong for surfaces with specular reflection (large k_s).
- Use Gouraud for diffuse surfaces ($k_s \approx 0$).