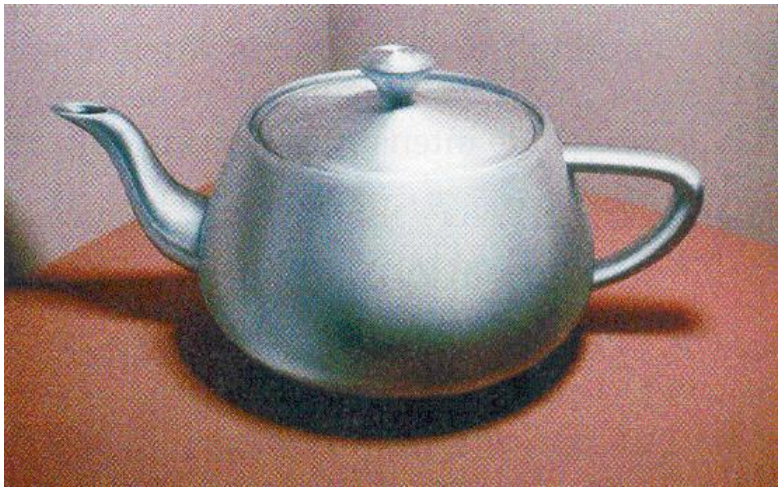
Illumination Models
Surface-rendering Methods



Overview

- Introduction
- Phong’s Reflection Model
- Lighting in WebGL
- Transparencies and Shadows
- Shading Methods
- Shading in WebGL
- Ray-Tracing

INTRODUCTION

3D visualization pipeline

- Create a scene and instantiate models
  - Position, orientation, size
- Establish viewing parameters
  - Camera position and orientation
- Perform clipping
- Compute illumination and shade polygons – where ?
- Project into 2D
- Rasterize

3D visualization pipeline

- Each object is processed separately
  - 3D triangles
- Object / triangle inside the view volume ?
  - No : go to next object / triangle
- Rasterization
  - Compute the location on the screen of each triangle
  - Compute the color of each pixel

Lighting or ***illumination***

- The process of computing the intensity and color of a sample point in a scene as seen by a viewer

- It is a function of the geometry of scene
  - Models, lights and camera, and their spatial relationships

- And of material properties
  - Reflection, absorption, …

Shading

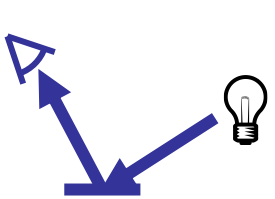
- The process of *interpolation* of color at points in-between those with known lighting or illumination
  - Vertices of triangles in a mesh
- Used in many real time graphics applications (e.g., games)
  - Calculating illumination at a point is usually expensive !
- BUT, in ray-tracing only do lighting for samples
  - Based on pixels (or sub-pixel samples for super-sampling)
  - No shading rule

GPU

- Lighting is usually calculated by a ***vertex shader***
- While shading is done by a ***fragment or pixel shader***
  - the term shader is ambiguous, unfortunately

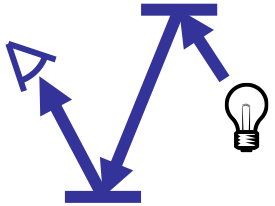
COMPUTING ILLUMINATION

Global Illumination



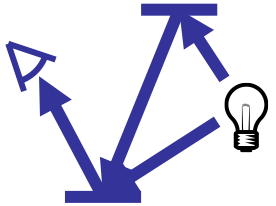
Direct illumination

+

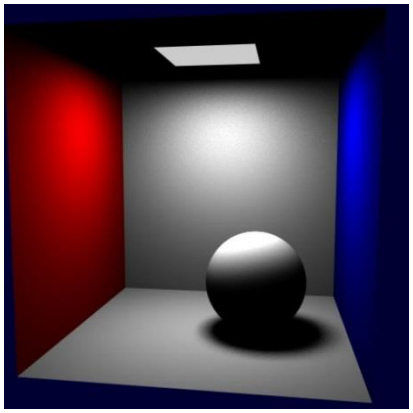


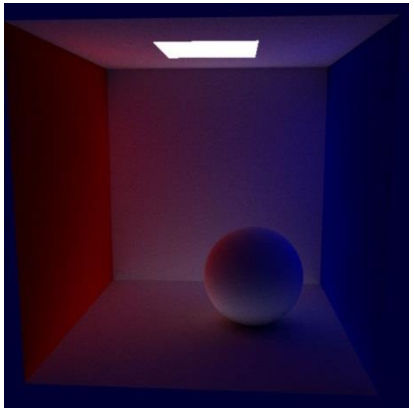
Indirect illumination

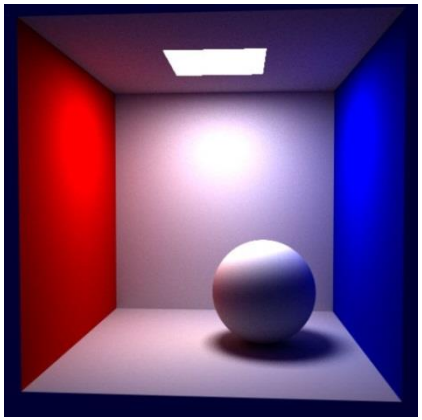
=



Total illumination







[Andy Van Dam]

Non-global vs. Global Illumination

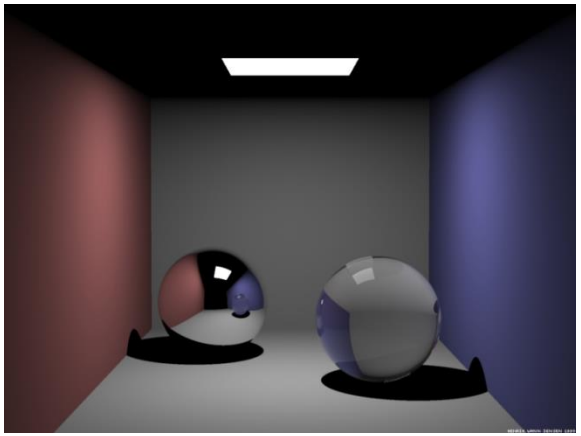
Direct (diffuse + specular) lighting
+ indirect specular reflection

Full global illumination

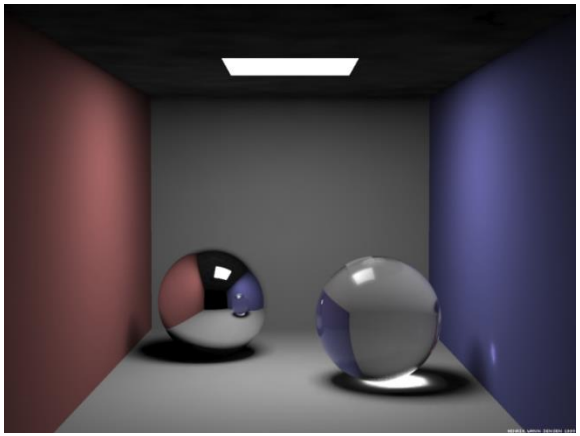
[Andy Van Dam]

Examples of Global Models

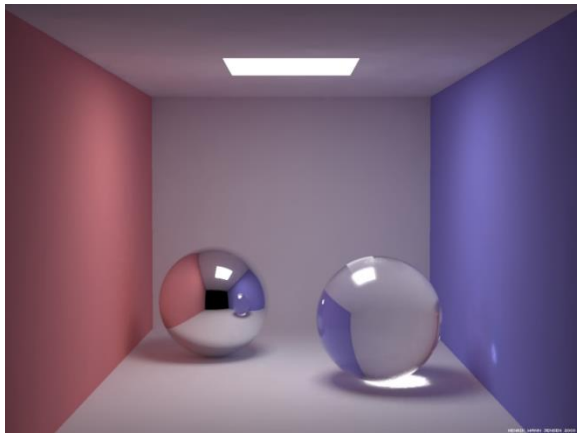
- Take into account global information of both direct (from emitters) and indirect illumination (inter-object reflections)
- Different approximations
  - Advantages and disadvantages; resource requirements
  - More computation gives better results…



Direct illumination + specular reflection
Ray trace



+ soft shadows and caustics
Ray trace + caustic photon map



+ diffuse reflection (color bleeding)
Ray trace + caustic and diffuse photon maps

http://graphics.ucsd.edu/~henrik/images/global.html

[Andy Van Dam]

Computing Illumination – *Light transport simulation*

- Evaluate illumination with enough samples to produce final images without any guessing / shading
- Often used for high quality renderers, e.g., those used in FX movies
  - Can take days for a single frame, even on modern render farms
- Some implementations can run in real time on the GPU
  - But more complex lighting models that are difficult to parallelize are still run on the CPU
- Many simulations use stochastic sampling
  - Path tracing, photon mapping, Metropolis light transport

Computing Illumination – Polygon rendering

- Evaluate illumination at several samples
- Shade (using a shading rule) in between to produce pixels in the final image
- Often used in real-time applications such as computer games
- Done in the GPU
- Lower quality than light transport simulation !!
  - But satisfactory results with various additions such as maps (bump, displacement, environment)

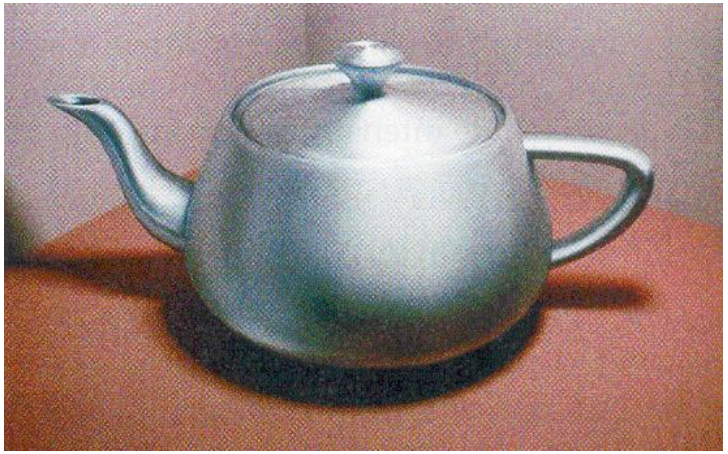
- Get realistic images by :
  - using perspective projections of the scene models
  - applying natural illumination effects on the visible surfaces
- Natural illumination effects are obtained using:
  - an illumination model – allows computing the color to be assigned to each visible surface point
  - a surface-rendering method that applies an illumination model and assigns a color to every pixel

- Photorealistic images require:

  - a precise representation of the properties of each surface

  - a good description of the scene's illumination

- And might imply modeling:
  - surface texture
  - transparency
  - reflections
  - shadows
  - etc.



Lighting – Polygon rendering

- Compute surface color based on
  - Type and number of light sources
  - Illumination model
    - Phong: ambient + diffuse + specular components
  - Reflective surface properties
  - Atmospheric effects
    - Fog, smoke
- Polygons making up a model surface are shaded
  - Realistic representation

- Illumination models used in Computer Graphics

  - are often an approximation to the Laws of Physics

  - that describe the interaction light-surface

- There are different types of illumination models

  - simple models, based on simple photometric computations
    (to reduce the computational cost)

  - more sophisticated models, based on the propagation of radiant energy
    (computationally more complex)

LIGHT SOURCES

Light Sources

- Objects radiating light and conttributing to the illumination of a scene's objects

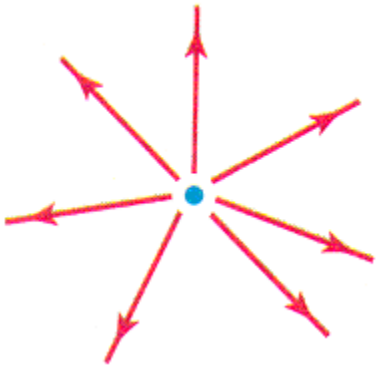
- Can be defined by several features:

  - Location

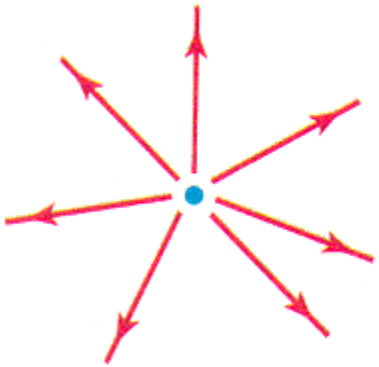
  - Color of emitted light

  - Emission direction

  - Shape

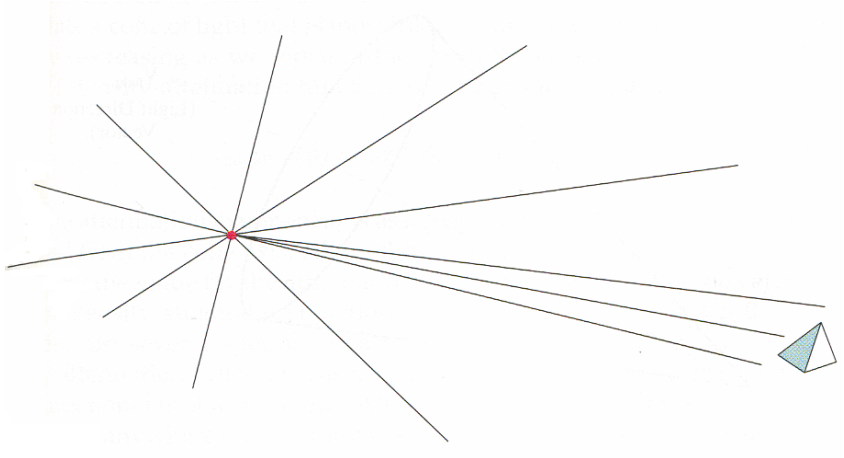


Simplified Light Sources



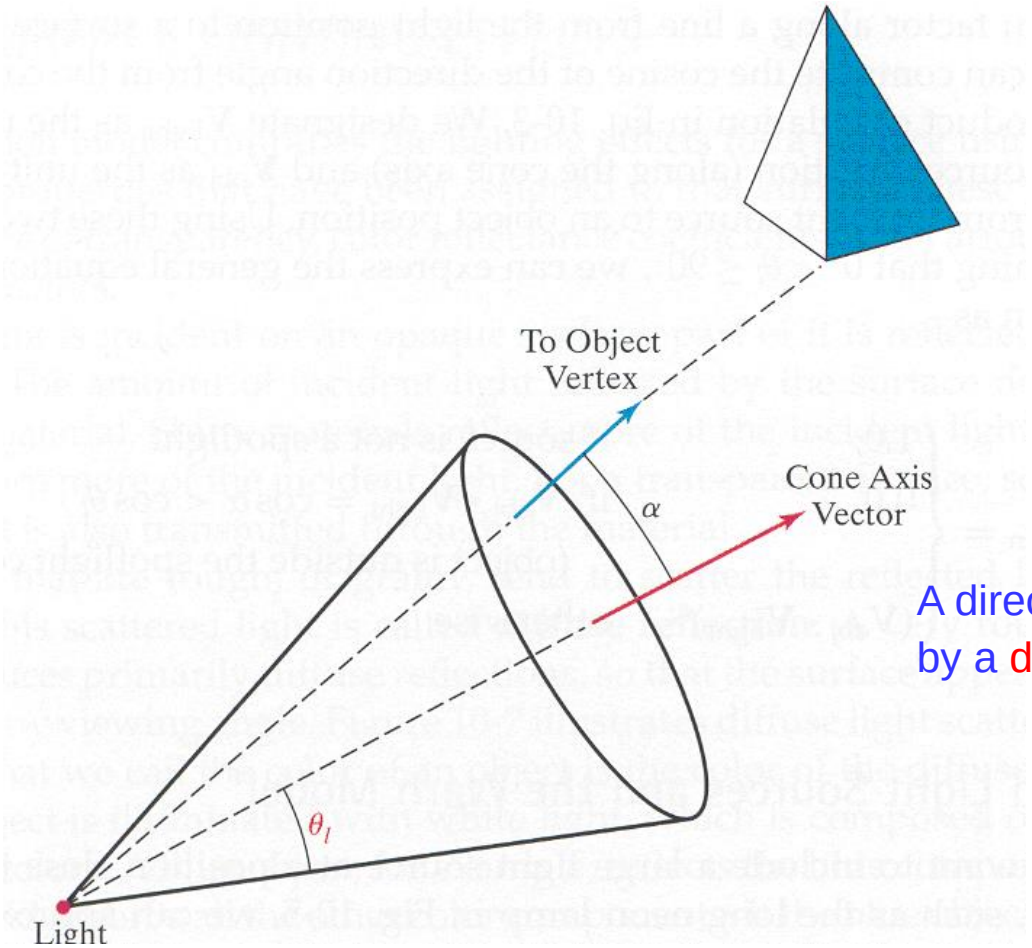
Isotropic point light source

Light source at an indefinite distance



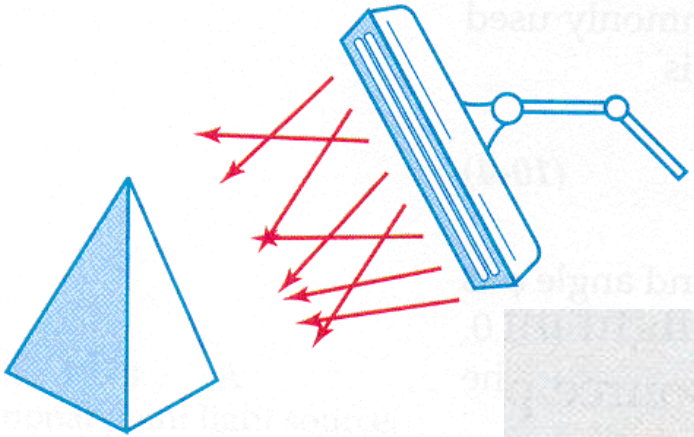
Rays emitted by a light source at a far-away location can be considered as parallel

Spotlight – Directional Light Source



A directional light source is defined by a direction and an emission angle

More sophisticated light sources



Illumination effects when close to the object

SURFACE FEATURES

Surface Features

- An illumination model takes into account a surface's optical properties:

  - reflection coefficients for each color

  - degree of transparency

  - texture parameters

- When light is incident on an opaque surface:

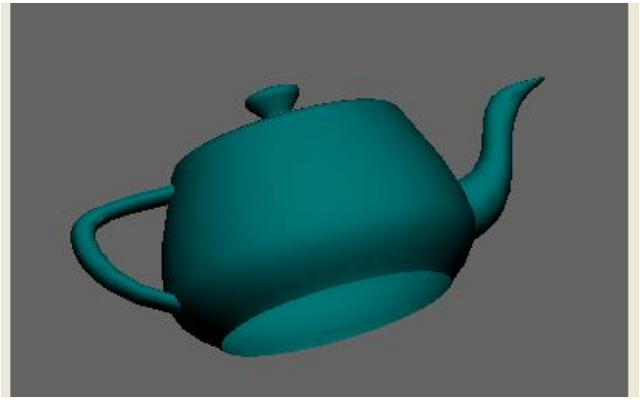
  - part is absorbed

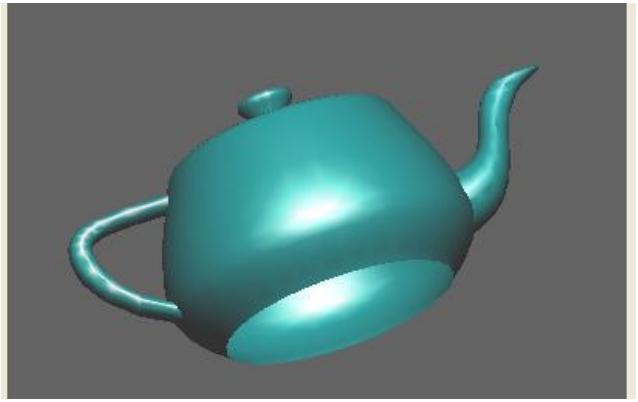
  - part is reflected

- The amount of reflected light depends on the surface's features

  - Shiny surfaces reflect more light

  - Mate / dull surfaces reflect less light

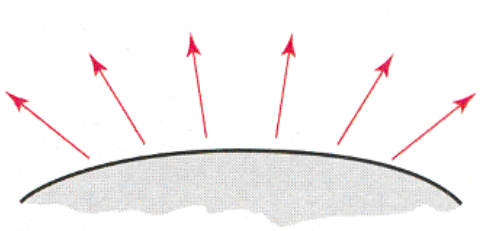




- Transparent surfaces transmit some light

- Rough surfaces tend to spread the reflected light in all directions

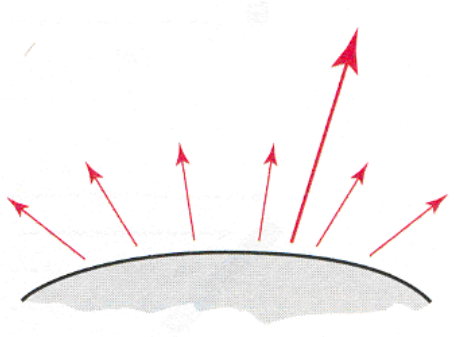
  - **diffuse reflection**



And look equally shiny from any viewpoint

- Smooth surfaces reflect more light in particular directions

  - **specular reflection (*highlight*)**



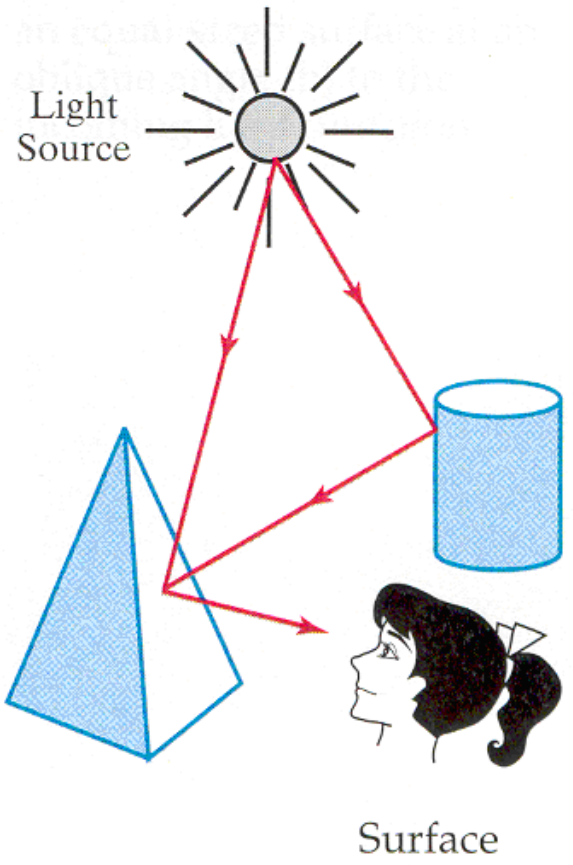
And present some shinier areas

- Another component to consider in an illumination model is

  - **ambient illumination**

- A surface might not be directly illuminated and still be visible, due to light reflected by other objects in the scene

- The amount of light reflected by a surface is the sum of all contributions from the light sources and the ambiente illumination



PHONG'S REFLECTION MODEL

Basic illumination models

- Sophisticated illumination models precisely compute the interaction efects between the radiating energy and the surface material

- Basic models use approximations to represent the physical processes producing the illumination effects

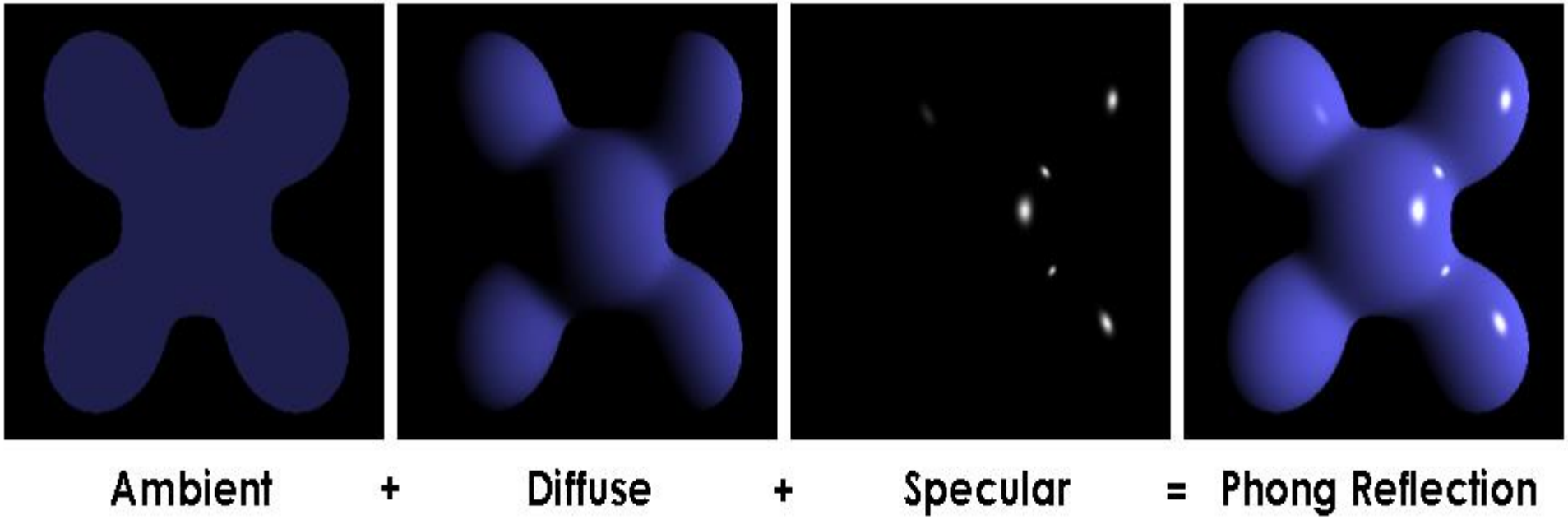
- The empirical model described next computes good enough results for most situations and includes:

  - ambient illumination

  - diffuse reflection

  - specular reflection

Phong reflection model – 1973



[Wikipedia]

Ambient illumination

- Ambient illumination is included as a constant value for the whole scene:

$$I_a$$

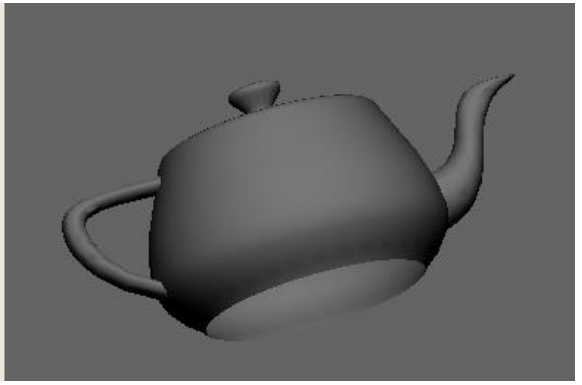
which entails a uniform illumination for all objects

- Therefore, the reflections produced by the surfaces are:

  - independent of viewing direction

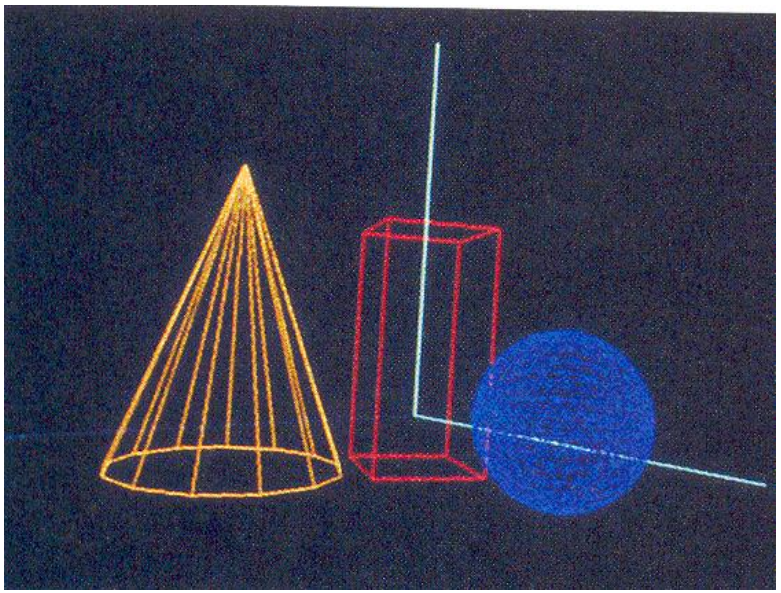
  - independent of surface orientation

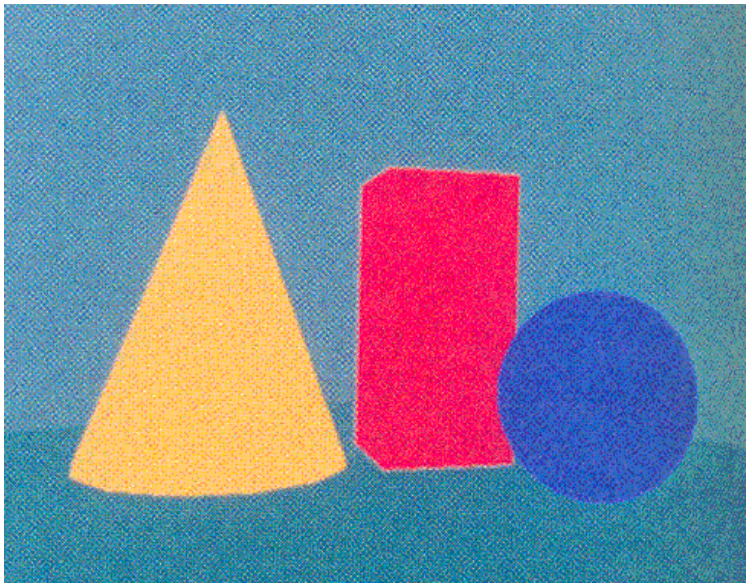
  - just depend on the optical properties of the surface



- Just by itself, ambient illumination produces uninteresting results.
- Why ?

$$I_{ambdiff} = K_d \, I_a$$



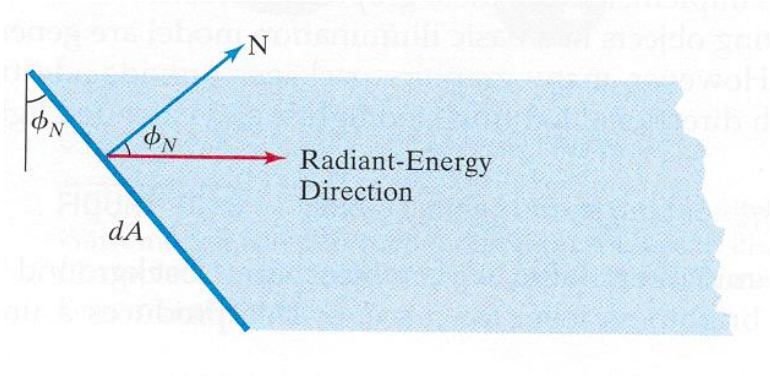


Just ambient illumination

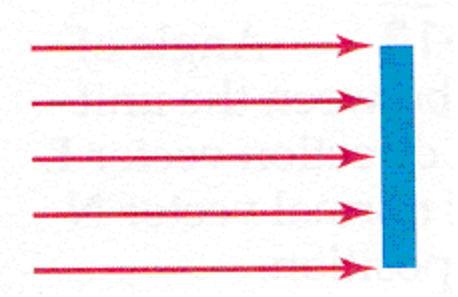
Diffuse reflection

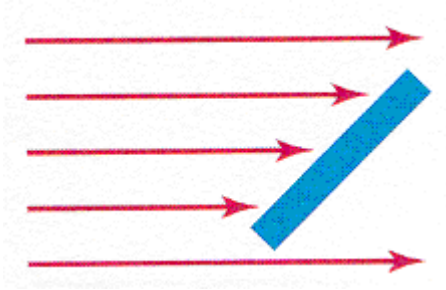
- It is considered that incident light is spread with equal intensity in all directions, regardless of the viewing direction
- Surfaces with that feature are called Lambertian reflectors or ideal diffuse relectors
- The intensity of reflected light is computed by Lambert's Law:

$$Intensity = \frac{Radiant\text{-}Energy\ per\ time\ unit}{Projected\ Area}$$



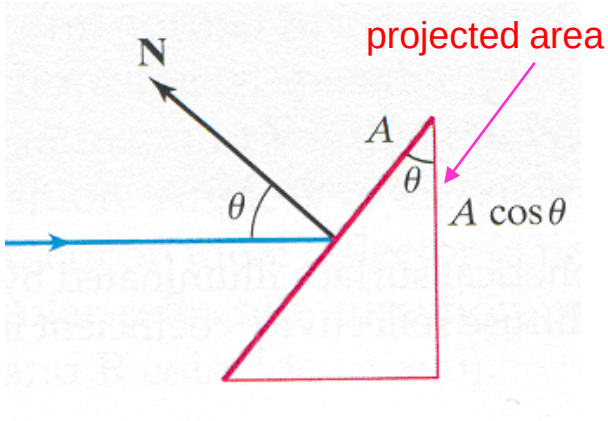
- There is, at least, one point light source (usually located at the viewpoint)
- The amount of incident light depends on the surface orientation regarding the direction of the light source
- A surface that is orthogonal to the light direction is “more illuminated” than an oblique surface, with the same area
- This effect can be seen by varying the orientation of a white paper-sheet relative to light direction





- θ is the incidence angle (between the surface normal and the light direction)

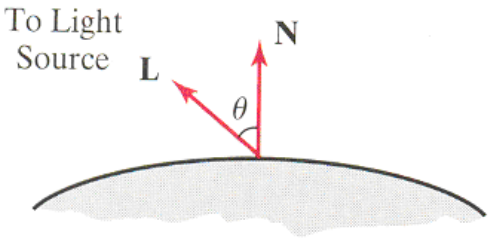
- Given a light source I_l, the amount of diffusely reflected light by a surface is:



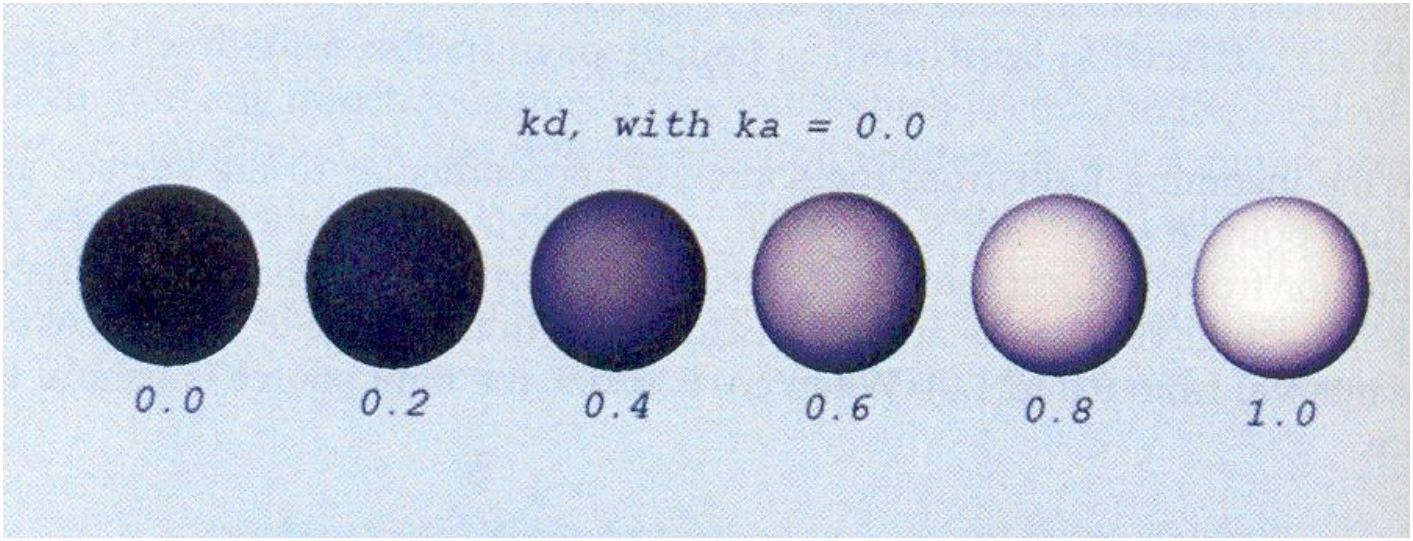
$$I_{l,\text{diff}} = k_d I_l \cos\theta$$

- Using unit vectors:

  N → surface normal
  L → light source direction



$$I_{l,\text{diff}} = \begin{cases} k_d I_l(\mathbf{N}\cdot\mathbf{L}), & \text{se } \mathbf{N}\cdot\mathbf{L} > 0 \\ 0.0, & \text{se } \mathbf{N}\cdot\mathbf{L} \le 0 \end{cases}$$

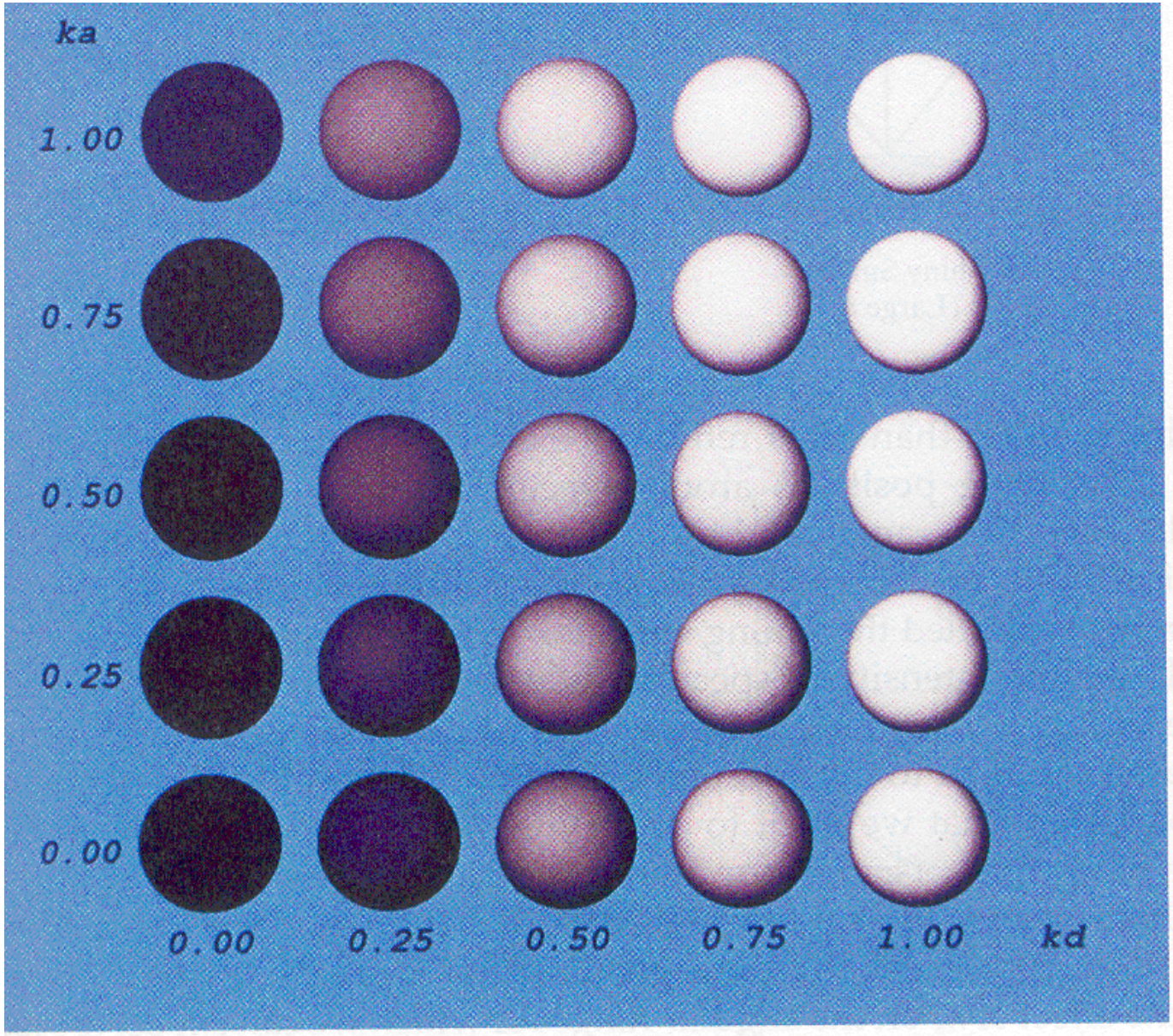


Diffuse reflection for a sphere illuminated by a white point light source, with $0 < K_d < 1$, and without ambient illumination ($K_a = 0$)

Adding the ambient and diffuse components:

$$I_{\text{diff}} = \begin{cases} k_a I_a + k_d I_l (\mathbf{N} \cdot \mathbf{L}), & \text{se } \mathbf{N} \cdot \mathbf{L} > 0 \\ k_a I_a, & \text{se } \mathbf{N} \cdot \mathbf{L} \leq 0 \end{cases}$$

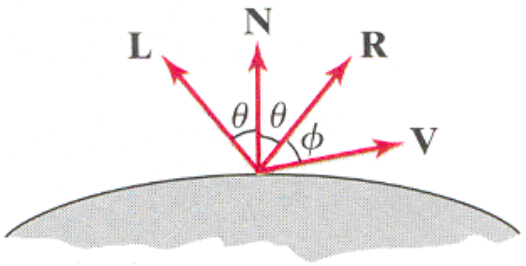
Ambient illumination



Diffuse reflection

Specular Reflection and Phong's Model

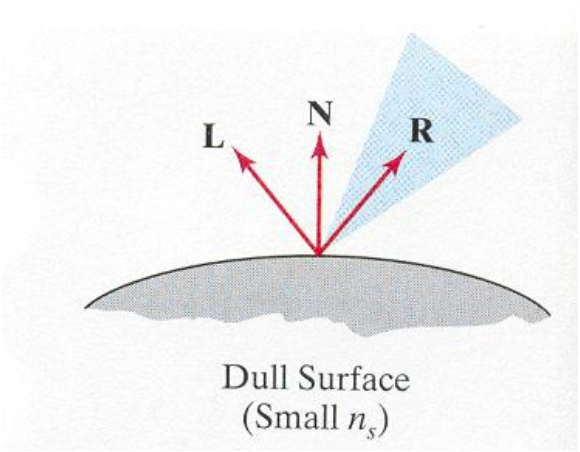
- Te shinier areas, specular reflections or *highlights*, that can be seen on shiny surfaces, result from the reflection of most light around concentrated areas



- The specular reflection angle is equal to the incidence angle (relative to the normal)
- **R** is the unit vector defining the ideal specular reflection direction
- **V** is the unit vector defining the viewing diretion
- An ideal reflector reflects light only in the specular reflection direction (the viewer perceives the specular reflection only if **V** and **R** are coincident $\Phi = 0$)

- Objects that are not ideal reflectors produce specular reflections in a finite set of directions around vector **R**

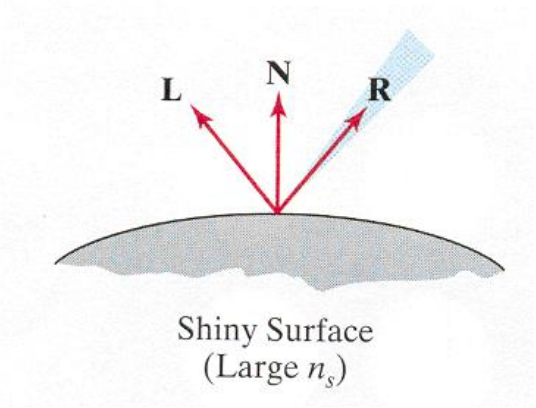
- Shiny surfaces have a narrow set of reflection directions

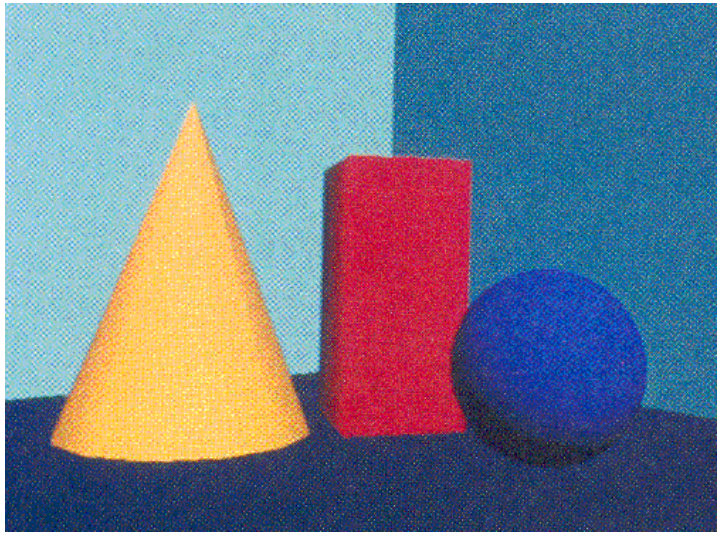


- The empirical Phong specular reflection model sets the intensity of the specular reflections as proportional to $\cos^{n_s}\phi$

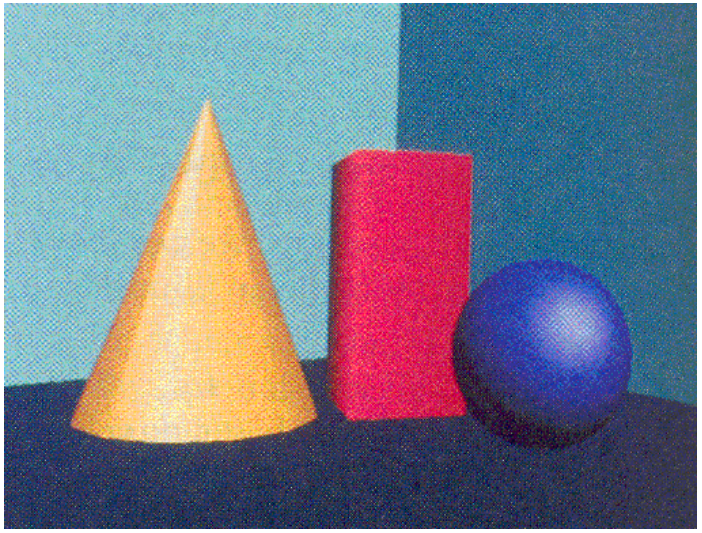
$$I_{l,\text{spec}} = W(\theta) I_l \cos^{n_s}\phi$$

$W(\theta)$ is the specular reflection coefficient

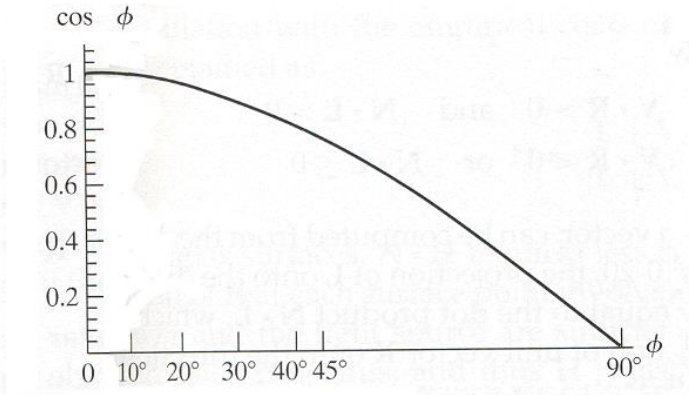




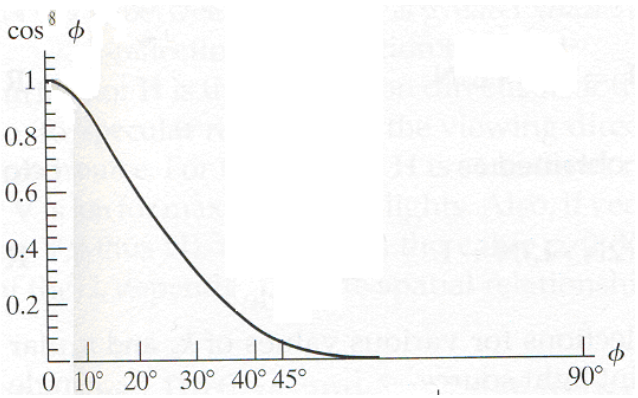
Diffuse reflection

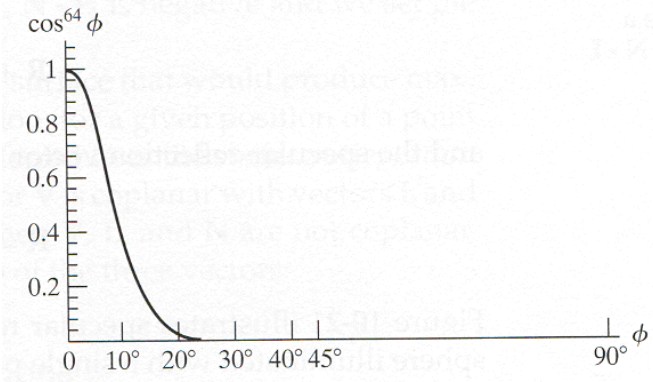


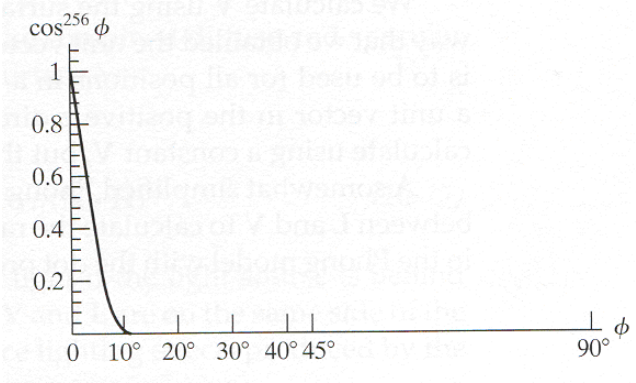
Diffuse and specular reflection



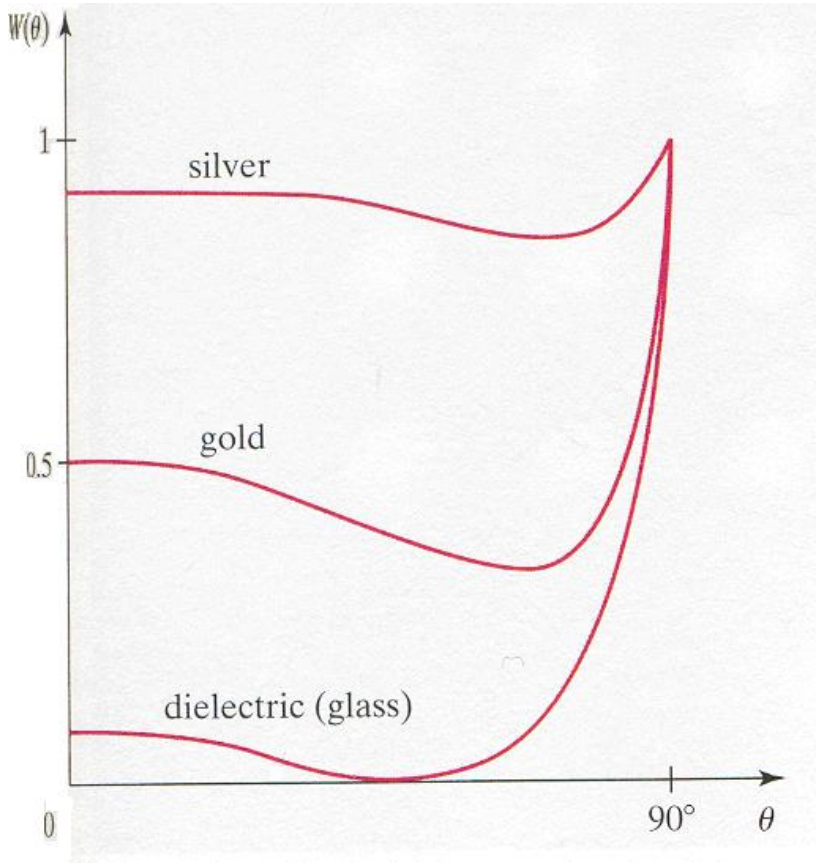
Less shinier surface







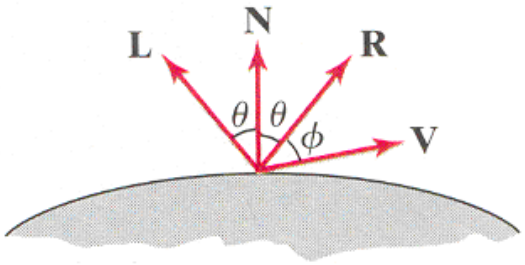
More shinier surface



For different materials, variation of $W(\theta)$ as a function of the incidence angle

- The value of cos Φ can be computed by the scalar product of unit vectors **V**·**R**

- There are no specular reflections, whenever:
  - The light source is behind the surface
  - **V** and **L** "are on the same side" of vector **N**



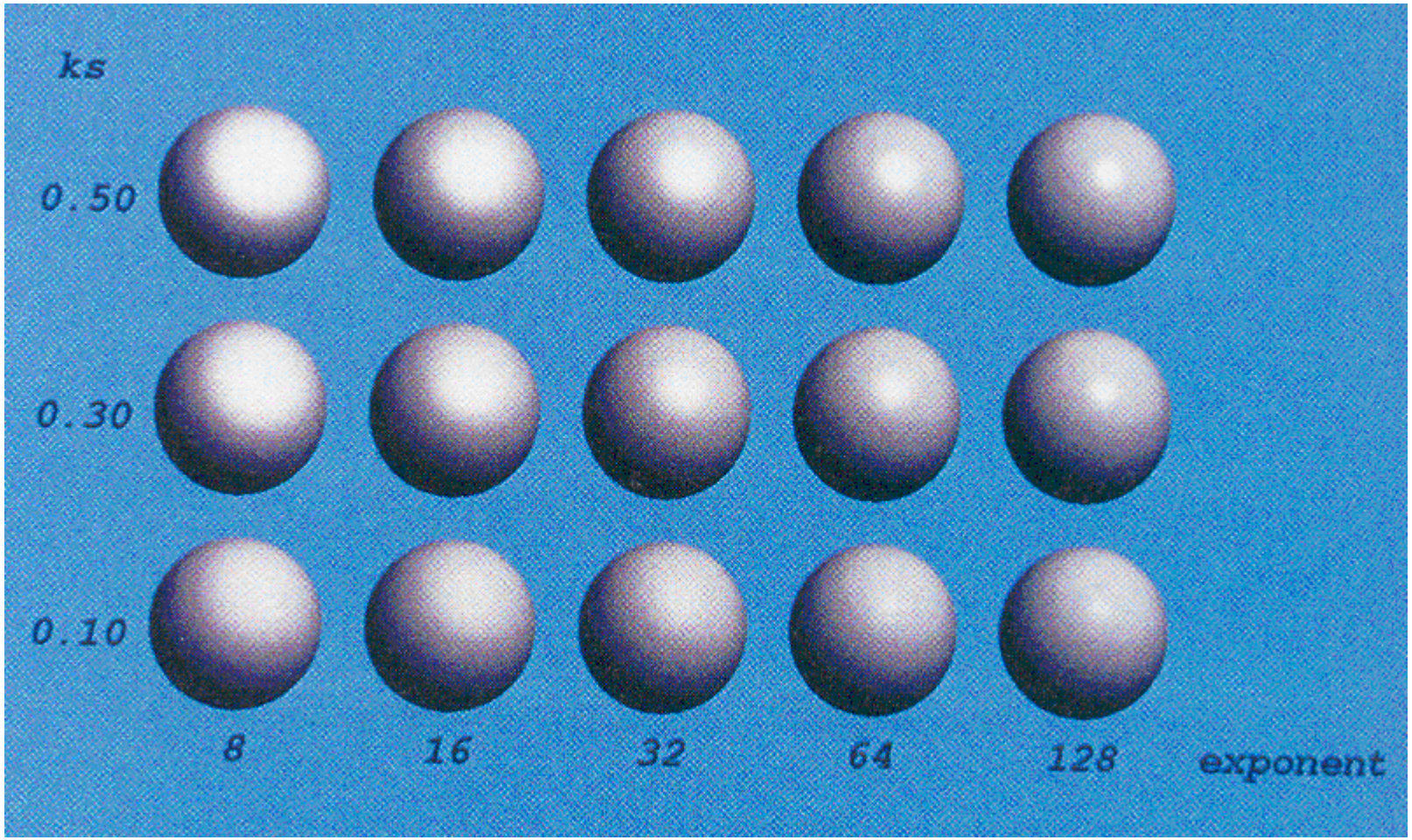
- Assuming that the specular reflection coefficent is a constant for each material, the intensity of the specular reflection, for a single point light source, can be computed as:

cos Φ

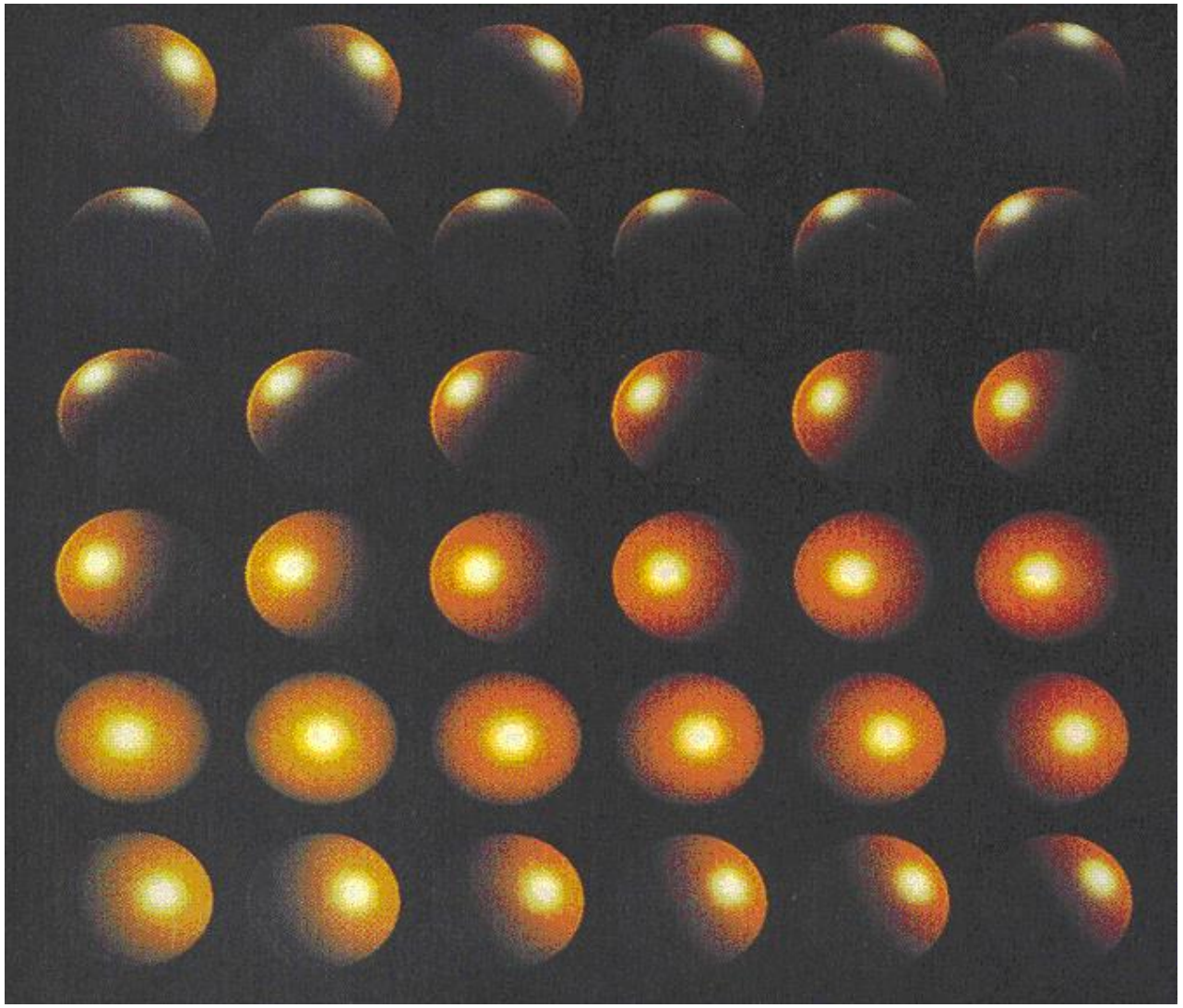
$$I_{l,\text{spec}} = \begin{cases} k_s I_l (\mathbf{V} \cdot \mathbf{R})^{n_s}, & \text{if } \mathbf{V} \cdot \mathbf{R} > 0 \quad \textit{and} \quad \mathbf{N} \cdot \mathbf{L} > 0 \\ 0.0, & \text{if } \mathbf{V} \cdot \mathbf{R} < 0 \quad \textit{or} \quad \mathbf{N} \cdot \mathbf{L} \leq 0 \end{cases}$$

R and V on opposite sides

Light source behind the surface



Specular reflections due to a single point light source, for several specular parameters



Light source moving around a sphere; no ambient illumination

More than one point light-source

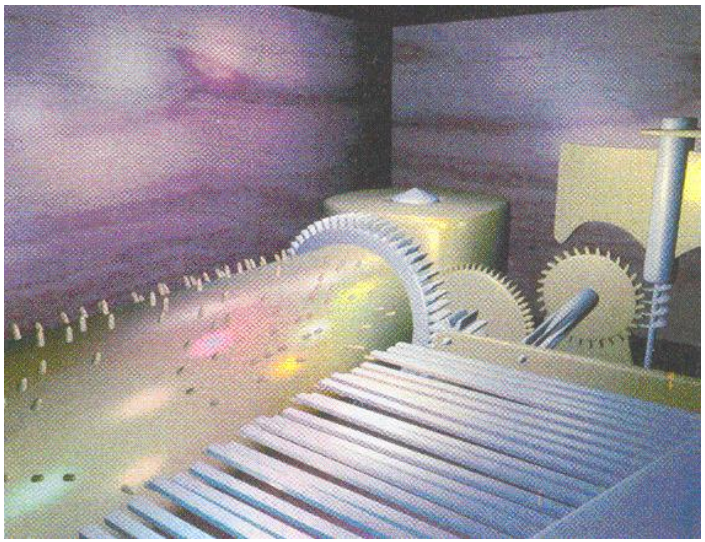
- For a single point light-source, we add the three components:
  - ambient illumination
  - diffuse reflection
  - specular reflection

$$I = k_a I_a + k_d I_l (\mathbf{N} \cdot \mathbf{L}) + k_s I_l (\mathbf{N} \cdot \mathbf{H})^{n_s}$$

- If there is more than one light-source:

$$I = k_a I_a + \sum_{l=1}^{n} I_l [k_d (\mathbf{N} \cdot \mathbf{L}) + k_s (\mathbf{N} \cdot \mathbf{H})^{n_s}]$$

Various light-sources, with different colors

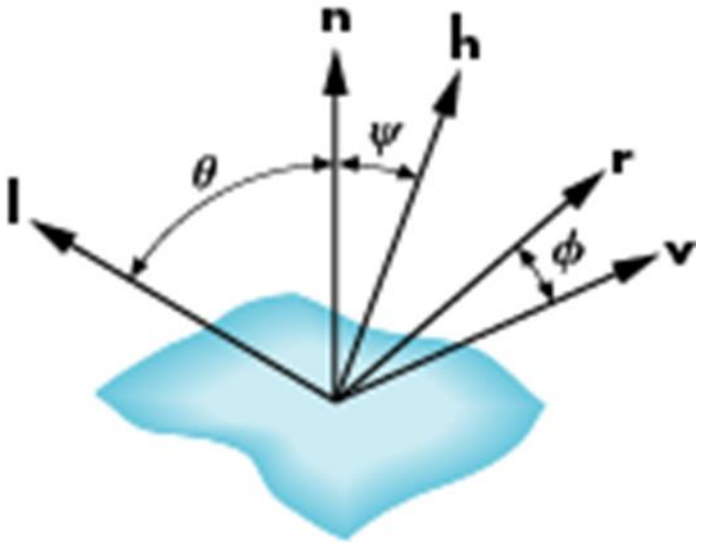


The *halfway vector* **H** (between **L** and **V**) is sometimes used as an approximation to **R,** in a version of Phong's model simpler to Compute for non-planar surface (with varying **N**)

The Halfway Vector

- **h** is the unit vector “halfway” between **l** and **v**

$$\mathbf{h} = (\mathbf{l} + \mathbf{v}) / \|\mathbf{l} + \mathbf{v}\|$$



[Angel]

RGB color model

- When using the RGB model, light intensity is specified in the illumination model as a RGB-vector:

$$I_l = (I_{lR}, I_{lG}, I_{lB})$$

- The same for the reflection coefficients:

$$k_a = (k_{aR}, k_{aG}, k_{aB}) \quad k_d = (k_{dR}, k_{dG}, k_{dB}) \quad k_s = (k_{sR}, k_{sG}, k_{sB})$$

- For instance, the blue component for the diffuse reflection is:

$$I_{lB,\text{diff}} = k_{dB} I_{lB} (\mathbf{N} \cdot \mathbf{L}_l)$$

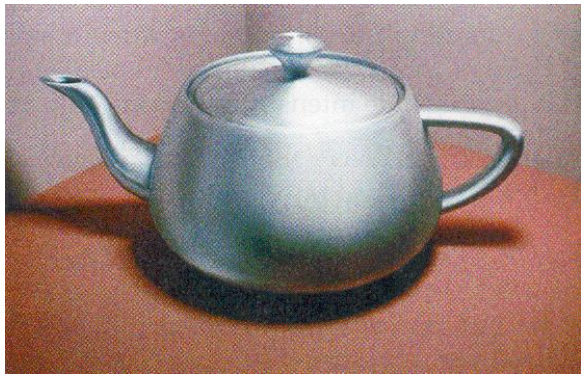
- It is possible to use other color models (e.g., CMY)

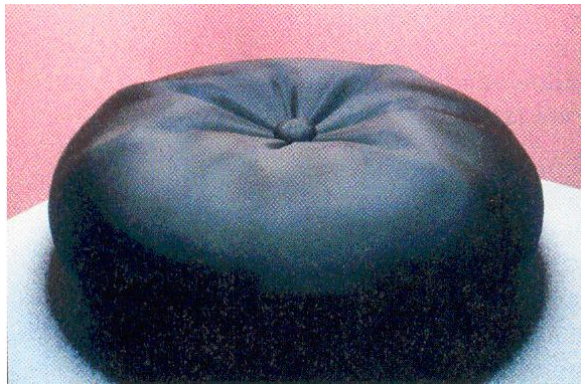
- In the original Phong model K_s is a constant and independent form surface color

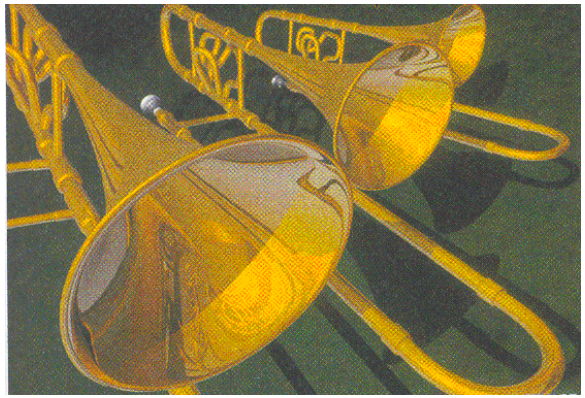
- That originates specular reflections with the same color as the light-source, and gives a “plastic-look” to the objects

- There are more sophisticated color models, that produce more realistic images for other kinds of materials

Images obtained with more sophisticated color models







TASK

Application problem (see PDF)

1 – Consider a sphere with center (0, 0, 0) and radius 2. The viewer is located at (0, 20, 0) and looks at point $P = (0, 2, 0)$ on the sphere's surface.

The sphere is illuminated by two point light sources:

- a red point light source, F_1, located at (-10, 12, 0),
- a blue point light source, F_2, located at (10, 12, 0).

In addition to the light sources, there is an ambient illumination component: $I_a = (0.1, 0.1, 0.1)$.

The sphere has the following properties:

- The ambient and diffuse reflection coefficients have the same value:

$$k_a = k_d = (0.5, 0.5, 0.5).$$

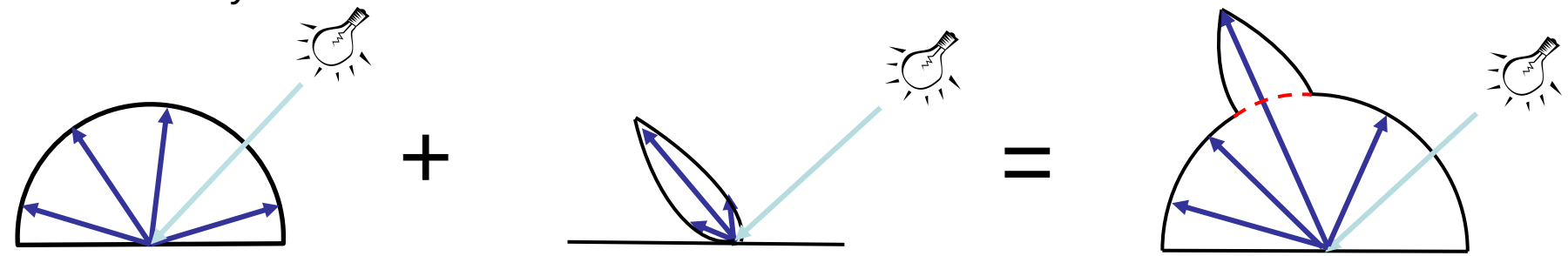
- The specular reflection coefficient is $k_s = (0.6, 0.6, 0.6)$.
- The *Phong* coefficient is $n = 10$.

Compute, using the *Phong Illumination Model*, the light intensity perceived by the viewer as reflected by point P on the sphere's surface.

MORE SOPHISTICATED REFLECTION MODELS

The BRDF

- Light arriving at a surface can scatter in many directions
  - The direction of scattering is determined by the material
  - Intensity at a given outgoing direction is dependent on incoming direction and material properties
- Model of reflectance is called the ***bidirectional reflectance distribution function*** (***BRDF***)
  - for any incoming light ray how much energy is reflected for any outgoing light ray



Diffuse Reflections | Specular Reflections | Total Scattering Distribution (*BRDF*)

BRDF for simple model of diffuse + specular reflection, e.g., the Phong Lighting Model

[Andy Van Dam]

A more sophisticated illumination model

Cook and Torrence, 1982

Other Lighting Models

- Models you have seen before are empirical, but look okay
  - Phong Model
  - Blinn-Phong Model
- More complex models are physically based
  - Cook-Torrance Model
  - Oren-Nayer Model
- Performance vs. accuracy tradeoff

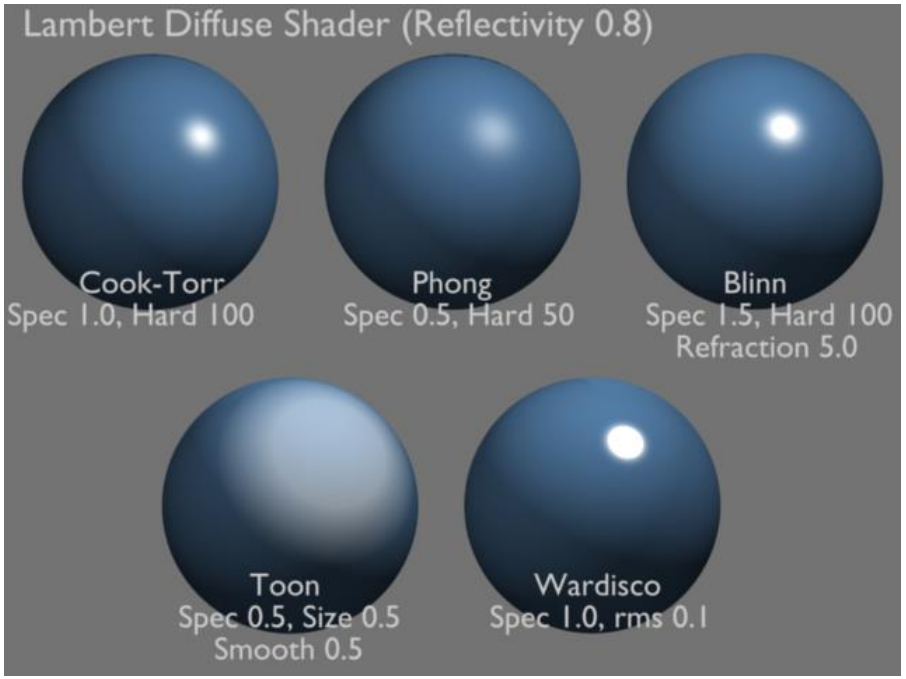


Image credit:
http://wiki.blender.org/index.php/File:Manual-Shaders-Lambert.png

[Andy Van Dam]

LIGHTING IN WEBGL

OpenGL (Pre-3.1) – *Lighting*

- The Phong illumination model is used
- Light source features and material properties are represented using the RGB or RGBA model
  - Emitted and reflected light
  - Ambient, diffuse and specular reflection coefficients
- Define several light sources, that can be independently controlled
- Programmers can implement and use more sophisticated color models !!
- But have to carry out all required computations !!

OpenGL (Pre-3.1) – *Lighting*

- Main steps:
  - Associate a “normal vector” to every vertex, which determines its orientation regarding each light source.
    - How to ?
    - Representation of polyhedra vs. approximation of curved surfaces
  - Create light sources, set their features and position them
  - Define material properties for the various models in the scene
  - Set some illumination model features
    - Distance from viewer to scene
    - Global ambient illumination
    - …

OpenGL (Pre-3.1) – *Lighting Model*

- Global ambient illumination

```
GLfloat lmodel_ambient[] = { 0.2, 0.2, 0.2, 1.0};
glLightModelfv( GL_LIGHT_MODEL_AMBIENT,
                                        lmodel_ambient );
```

- Distance to viewer

```
glLightModeli( GL_LIGHT_MODEL_LOCAL_VIEWER, GL_TRUE );

glLightModeli( GL_LIGHT_MODEL_LOCAL_VIEWER,
                                          GL_FALSE );
```

Default! Why?

- Enable / Disable

```
glEnable( GL_LIGHTING );
```

OpenGL (Pre-3.1) – *Light Sources*

- Examplo: point light-source

```
GLfloat light_ambient[] = { 0.0, 0.0, 0.0, 1.0 };
GLfloat light_diffuse[] = { 1.0, 1.0, 1.0, 1.0 };
GLfloat light_specular[] = { 1.0, 1.0, 1.0, 1.0 };
GLfloat light_position[] = { 1.0, 1.0, 1.0, 0.0 };

glLightfv(GL_LIGHT0, GL_AMBIENT, light_ambient);
glLightfv(GL_LIGHT0, GL_DIFFUSE, light_diffuse);
glLightfv(GL_LIGHT0, GL_SPECULAR, light_specular);
glLightfv(GL_LIGHT0, GL_POSITION, light_position);

glEnable( GL_LIGHT0 );
```

OpenGL (Pre-3.1) – *Light Sources*

- Example: "spotlight"

```
…
GLfloat spot_direction[] = { -1.0, -1.0, 0.0 };

glLightfv(GL_LIGHT1, GL_AMBIENT, light1_ambient);
glLightfv(GL_LIGHT1, GL_DIFFUSE, light1_diffuse);
glLightfv(GL_LIGHT1, GL_SPECULAR, light1_specular);
glLightfv(GL_LIGHT1, GL_POSITION, light1_position);
…
glLightf(GL_LIGHT1, GL_CONSTANT_ATTENUATION, 1.5);
glLightf(GL_LIGHT1, GL_LINEAR_ATTENUATION, 0.5);
…
glLightf(GL_LIGHT1, GL_SPOT_CUTOFF, 45.0);
glLightfv(GL_LIGHT1, GL_SPOT_DIRECTION,
                                    spot_direction);
glLightf(GL_LIGHT1, GL_SPOT_EXPONENT, 2.0);

glEnable( GL_LIGHT1 );
```

OpenGL (Pre-3.1) – *Material Properties*

- Different properties / coefficients

```
GLfloat no_mat[] = { 0.0, 0.0, 0.0, 1.0 };
GLfloat mat_ambient[] = { 0.7, 0.7, 0.7, 1.0 };
GLfloat mat_ambient_color[] = { 0.8, 0.8, 0.2, 1.0
  };
GLfloat mat_diffuse[] = { 0.1, 0.5, 0.8, 1.0 };
GLfloat mat_specular[] = { 1.0, 1.0, 1.0, 1.0 };

GLfloat no_shininess[] = { 0.0 };
GLfloat low_shininess[] = { 5.0 };
GLfloat high_shininess[] = { 100.0 };

GLfloat mat_emission[] = {0.3, 0.2, 0.2, 0.0};
```

OpenGL (Pré-3.1) – *Material Properties*

- Just diffuse reflection

```
glMaterialfv(GL_FRONT, GL_AMBIENT, no_mat);

glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);

glMaterialfv(GL_FRONT, GL_SPECULAR, no_mat);
glMaterialfv(GL_FRONT, GL_SHININESS, no_shininess);

glMaterialfv(GL_FRONT, GL_EMISSION, no_mat);
```

OpenGL (Pre-3.1) – *Material Properties*

- Diffuse and specular reflection, with high Phong coefficient

```
glMaterialfv(GL_FRONT, GL_AMBIENT, no_mat);

glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);

glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
glMaterialfv(GL_FRONT, GL_SHININESS,
                                    high_shininess);

glMaterialfv(GL_FRONT, GL_EMISSION, no_mat);
```

OpenGL (Pre-3.1) - *Shading*

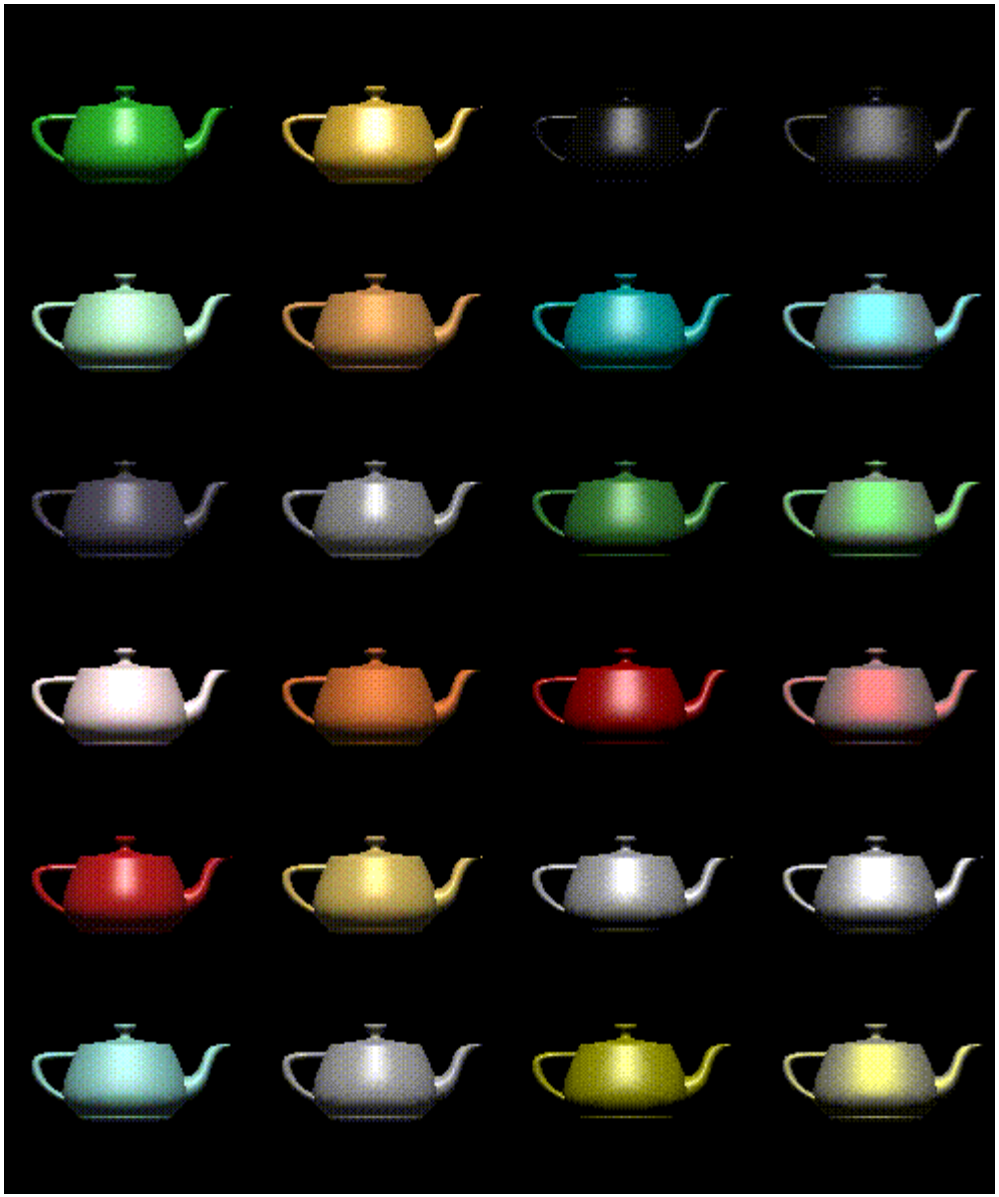
- Color assigned to primitives (line segments and polygons)
  - Constant Color (*Flat-Shading*), using the color assigned to one of the primitive's vertices (which one?)

```
glShadeModel( GL_FLAT );
```

  - Computed by interpolation (*Gouraud Shading*) – Default !

```
glShadeModel( GL_SMOOTH );
```

OpenGL – *Material Properties*



- Teapots with different “materials”:
  - Emerald, jade, …
  - Brass, bronze, …
  - Plastic
  - Rubber

[OpenGL – The Red Book]

Material properties

Material	ambient (k_a)	diffuse (k_d)	specular (k_s)	specular exponent (m)	translucency (d)
Brass	0.329412 0.223529 0.027451	0.780392 0.568627 0.113725	0.992157 0.941176 0.807843	27.8974	1.0
Bronze	0.2125 0.1275 0.054	0.714 0.4284 0.18144	0.393548 0.271906 0.166721	25.6	1.0
Polished Bronze	0.25 0.148 0.06475	0.4 0.2368 0.1036	0.774597 0.458561 0.200621	76.8	1.0
Chrome	0.25 0.25 0.25	0.4 0.4 0.4	0.774597 0.774597 0.774597	76.8	1.0
Copper	0.19125 0.0735 0.0225	0.7038 0.27048 0.0828	0.256777 0.137622 0.086014	12.8	1.0
Polished Copper	0.2295 0.08825 0.0275	0.5508 0.2118 0.066	0.580594 0.223257 0.0695701	51.2	1.0
Gold	0.24725 0.1995 0.0745	0.75164 0.60648 0.22648	0.628281 0.555802 0.366065	51.2	1.0
Polished Gold	0.24725 0.2245 0.0645	0.34615 0.3143 0.0903	0.797357 0.723991 0.208006	83.2	1.0

[https://people.eecs.ku.edu/~jrmiller/Courses/672/InClass/3DLighting/MaterialProperties.html]

OpenGL / WebGL – *Lighting*

- And, nowadays, how to do it?
- What calculations are done by the application ?
- What calculations are done by the *shaders* ?

- Per vertex vs. per fragment shading

OpenGL / WebGL – *Lighting*

- Compute in the application and/or send attributes to *shaders*
  - Material properties
  - Normal vectors
  - Light-source features
- Use unit vectors !!
  - Pay attention to applied transformations !!
  - GLSL normalization function
- Compute scalar products !!

OpenGL / WebGL – *Light Sources*

- Example: point light-source

```
vec4 diffuse0 = vec4(1.0, 0.0, 0.0, 1.0);

vec4 ambient0 = vec4(1.0, 0.0, 0.0, 1.0);

vec4 specular0 = vec4(1.0, 0.0, 0.0, 1.0);

vec4 light0_pos = vec4(1.0, 2.0, 3.0, 0.0);
```

OpenGL / WebGL – *Material Properties*

- Diffuse and specular reflection, with high Phong coefficient

```
vec4 ambient = vec4(0.2, 0.2, 0.2, 1.0);

vec4 diffuse = vec4(1.0, 0.8, 0.0, 1.0);

vec4 specular = vec4(1.0, 1.0, 1.0, 1.0);

GLfloat shine = 100.0
```

OpenGL / WebGL – *Per Vertex Shading*

- Alternatives
  - The application computes a color (*shade*) for each vertex of the triangle mesh
  - And sends it to the *vertex shader*

- OR
  - The application sends all parameters to the *vertex shader*
  - The shader computes the color (*shade*) for each vertex

OpenGL / WebGL – *Per Vertex Shading*

- Smooth Shading → Default
  - Color interpolated along each primitive
  - If passed to the *fragment shader* as *varying*
- Flat Shading
  - Constant color for each primitive
  - If passed to the *fragment shader* as *uniform*

OpenGL / WebGL – *Per Vertex Shading*

```
// vertex shader
attribute vec4 vPosition;
attribute vec3 vNormal;
varying vec4 color;  //vertex shade

// light and material properties
uniform vec4 AmbientProduct, DiffuseProduct, SpecularProduct;
uniform mat4 ModelView;
uniform mat4 Projection;
uniform vec4 LightPosition;
uniform float Shininess;
```

OpenGL / WebGL – *Per Vertex Shading*

```
void main()
{
    // Transform vertex  position into eye coordinates
    vec3 pos = (ModelView * vPosition).xyz;

    vec3 L = normalize( LightPosition.xyz - pos );
    vec3 E = normalize( -pos );
    vec3 H = normalize( L + E );

    // Transform vertex normal into eye coordinates
    vec3 N = normalize( ModelView*vec4(vNormal, 0.0) ).xyz;
```

OpenGL / WebGL – *Per Vertex Shading*

```
    // Compute terms in the illumination equation
    vec4 ambient = AmbientProduct;

    float Kd = max( dot(L, N), 0.0 );
    vec4  diffuse = Kd*DiffuseProduct;
    float Ks = pow( max( dot(N, H), 0.0), Shininess );
    vec4  specular = Ks * SpecularProduct;
    if( dot(L, N) < 0.0 )  specular = vec4(0.0, 0.0, 0.0, 1.0);
    gl_Position = Projection * ModelView * vPosition;

    color = ambient + diffuse + specular;
    color.a = 1.0;
}
```

OpenGL / WebGL– *Per Vertex Shading*

```
// fragment shader

precision mediump float;

varying vec4 color;

void main()
{
    gl_FragColor = color;
}
```

TRANSPARENCIES AND SHADOWS

Transparency

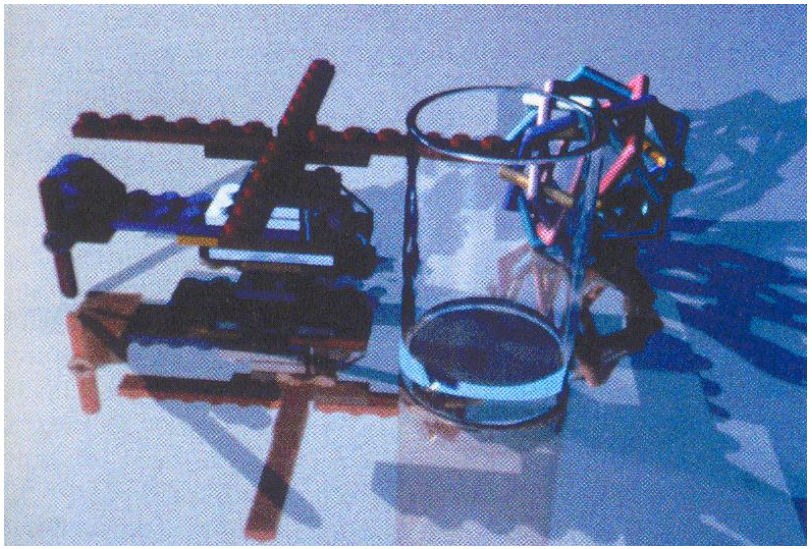
- A transparent object allows seeing other objects behind it

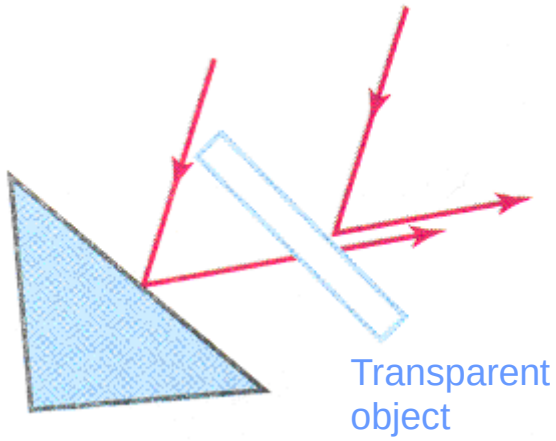
- There are also translucent objects, which transmit light, but scatter it in all directions

- Transparency can be modeled with different degrees of realism:

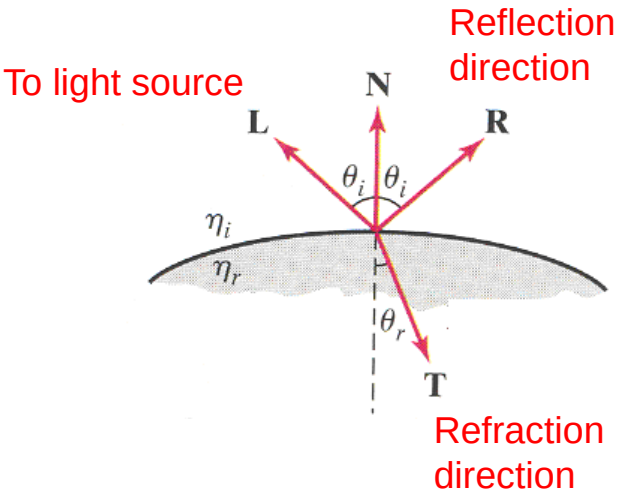
  - with refraction (light rays change direction)

  - without refraction





- For greater realism, light refraction has to be considered

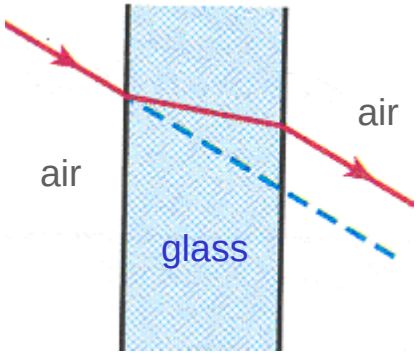


- The direction of refracted light is different from the direction of incident light

- The refraction angle varies with:

  - the media refraction indices $\eta_i\ \eta_r$

  - the incidence angle θ_i



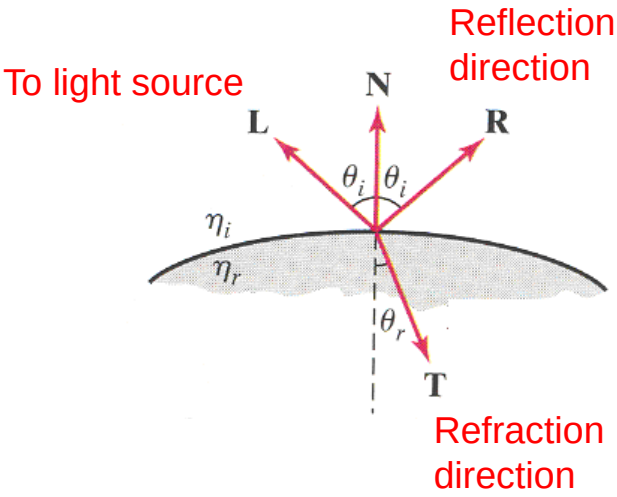
- According to Snell's Law :

$$\sin\theta_r = \frac{\eta_i}{\eta_r}\sin\theta_i$$

The emerging ray is parallel to the incident light ray

- The index of refraction varies with different factors:

  - light wavelength
  - material temperature
  - direction (for anisotropic materials)
  - …



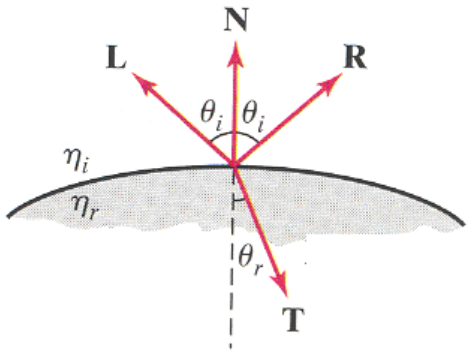
- In general, we can use na average value

- Indices of refraction for different materials:

vacuum / air	- 1.00
ice	- 1.31
water	- 1.33
common glass	- 1.52
quartz	- 1.54

- To compute T (unit vector for the refraction direction):

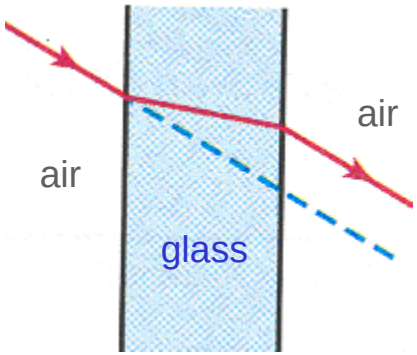
$$\mathbf{T} = \left(\frac{\eta_i}{\eta_r} \cos\theta_i - \cos\theta_r \right) \mathbf{N} - \frac{\eta_i}{\eta_r} \mathbf{L}$$



N – surface unit normal vector

L – unit vector towards the light source

- The general effect of light passing through a glass slice is na emerging light ray parallel to incident light ray



The emerging ray is parallel to the incident light ray

- In most applications, less realistic and faster approximations are used

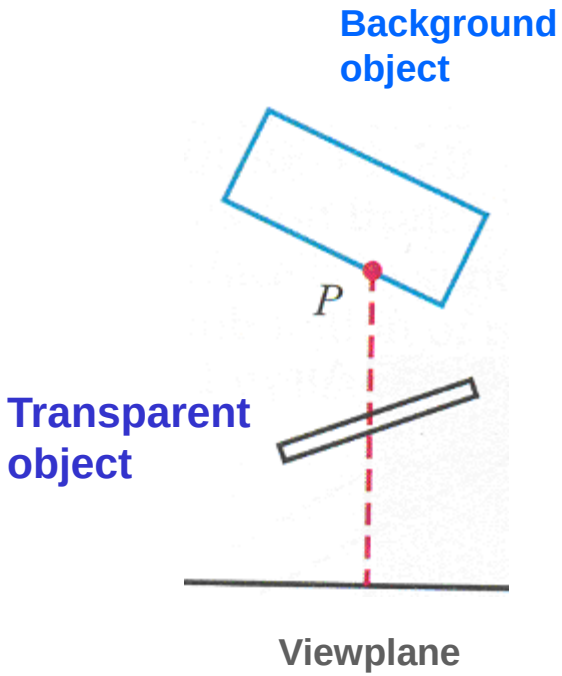
- A simple approximation ignores direction changes between materials of the transmitted light rays

- This approximation:

  - is fast

  - produces acceptable results

for transparent surfaces of small thickness



- It is possible to combine light transmitted through a transparent surface with light reflected byt that surface:

$$I = (1 - k_t) I_{\text{refl}} + k_t I_{\text{trans}}$$

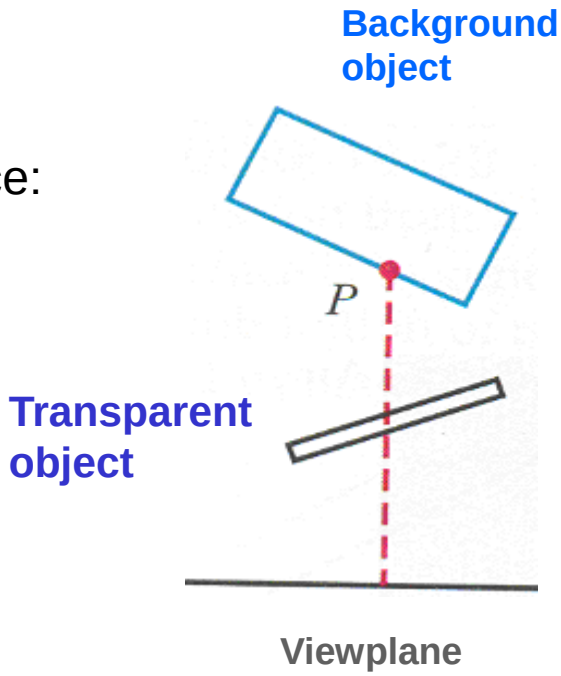
k_t - transparency coefficient [0, 1]

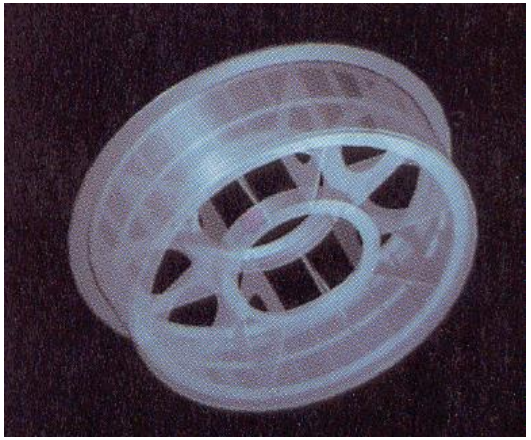
1 – fully transparent

0 – totally opaque

- Can also define the opacity coefficient:

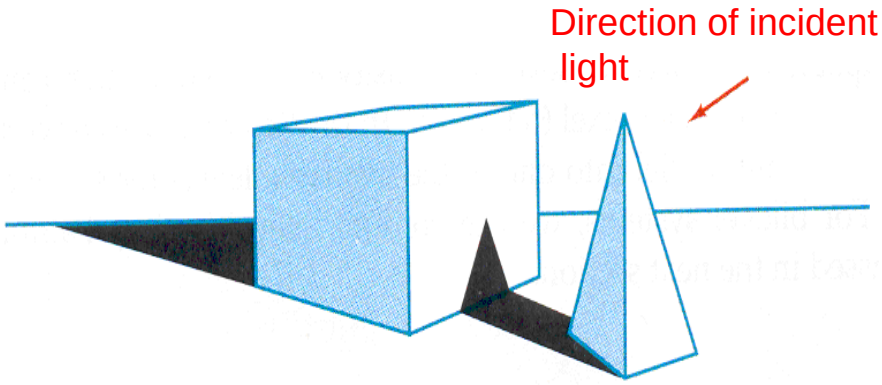
$$(1 - k_t)$$





transparency without refraction

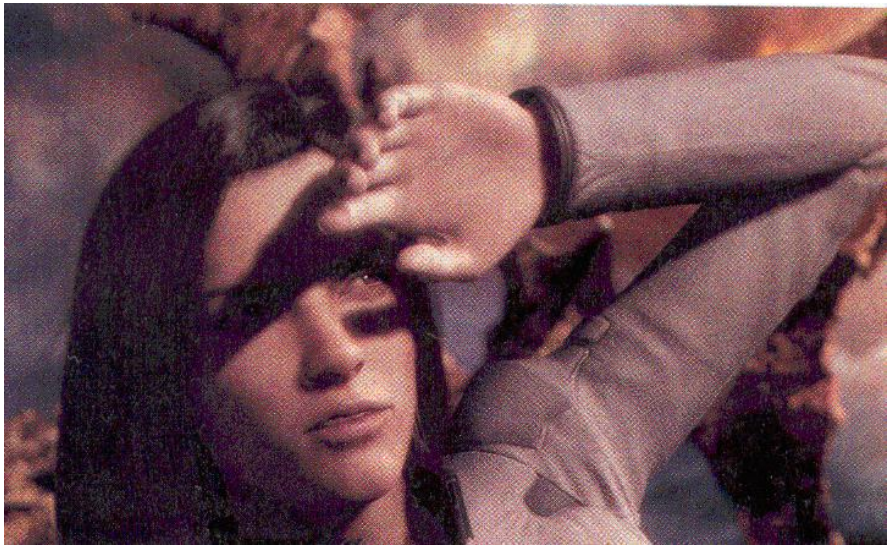
Shadows



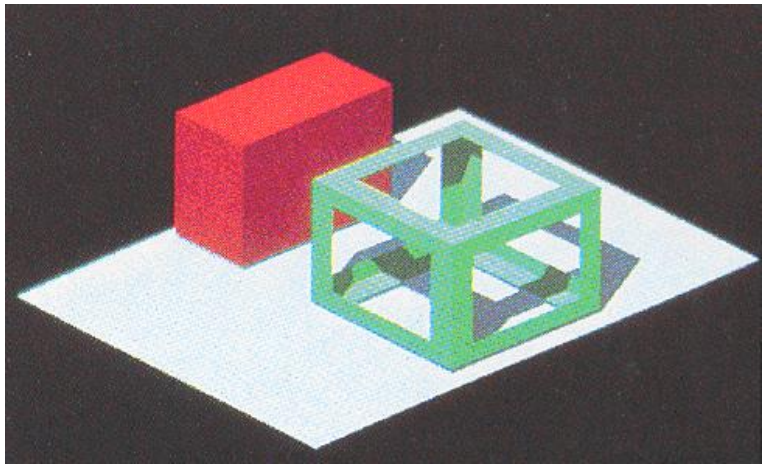
- Areas that are not illuminated by light sources can be identified by visibility detection methods

- Identify the scene's polygonal faces that are not visible form the location of each light source

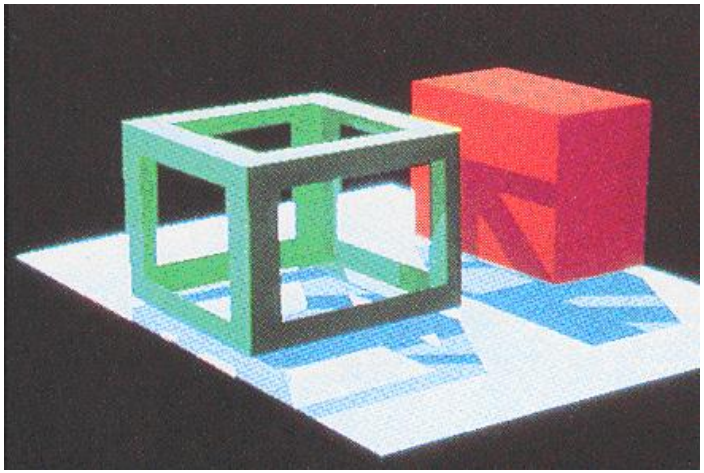
- Then, just use the light sources that contribute to the illumination of each polygonal face



Example: Shadows due to



one light source



two light sources

SHADING

Illumination and Shading

- How to optimize?
  - Fewer light sources
  - Simple shading method
- BUT, less computations mean less realism
  - Wireframe representation
  - Flat-shading
  - Gouraud shading
  - Phong shading

Polygon Rendering Methods – *Shading*

- Color values resulting from an illumination model can be used for surface rendering in different ways:

  - Compute the color values for each and every pixel corresponding to the projected surface

  - Compute the color values just for a few chosen pixels, and compute approximate color values for the rest

- In general, graphics APIs use *scan-line* algorithms and use the illumination model to compute color values at mesh vertices

  - Some interpolate color values along the *scan-lines*

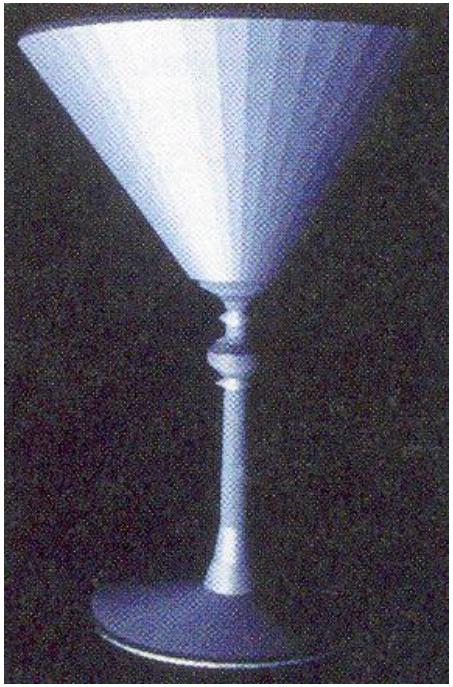
  - Other use more precise methods

Polygon Rendering Methods – *Shading*

- The most common shading methods are:
  - Constant color intensity method or *flat-shading*
  - Gouraud's method
  - Phong's method

Flat-Shading

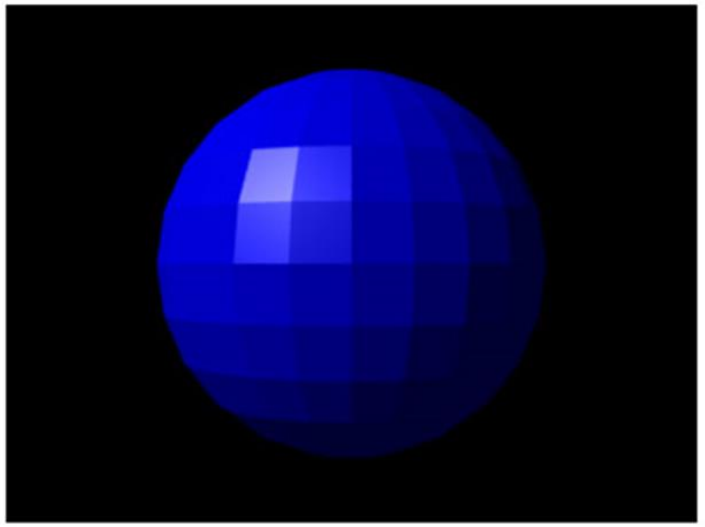
- Uses the illumination model to compute RGB color values at a chosen point for each polygon (first vertex, centroid, etc.)
- Assigns the same color to all pixels in the projection of each polygon
- It is simple and fast !
- Appropriate for simple cases and useful to obtain the general appearance of a curved object



- In general, *flat-shading* is adequate when all the following conditions are verified:
  - The polygon is a face of a polyhedron and not part of a polygonal mesh approximating a curved surface
  - All light sources are very far-away (indefinite distance):
    (**N.L** constant for the entire polygon)
  - The viewpoint is very far-away (indefinite distance)
    (**V.R** constant for the entire polygon)
- Otherwise, the approximation is reasonable if the object's curved surface is represented by a mesh composed of very small area polygons

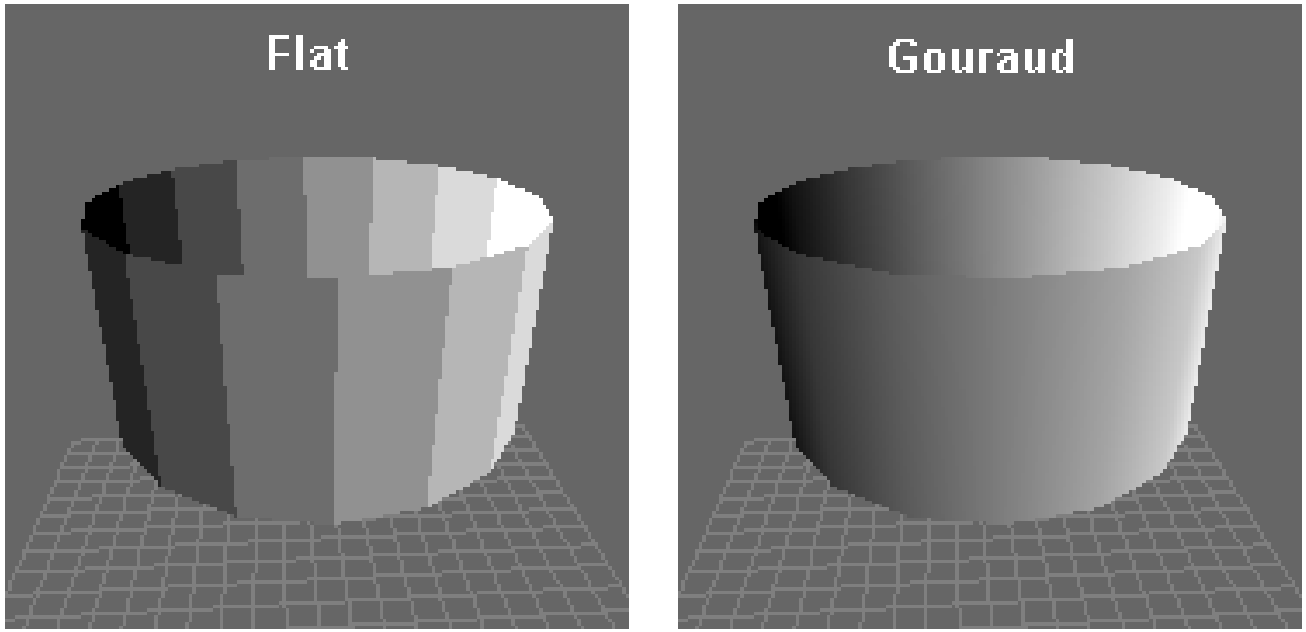
Flat-Shading

- For each triangle / polygon
  - Apply the illumination model just once !
  - All pixels have the same color
- Fast !
- But objects seem “blocky”



FLAT SHADING

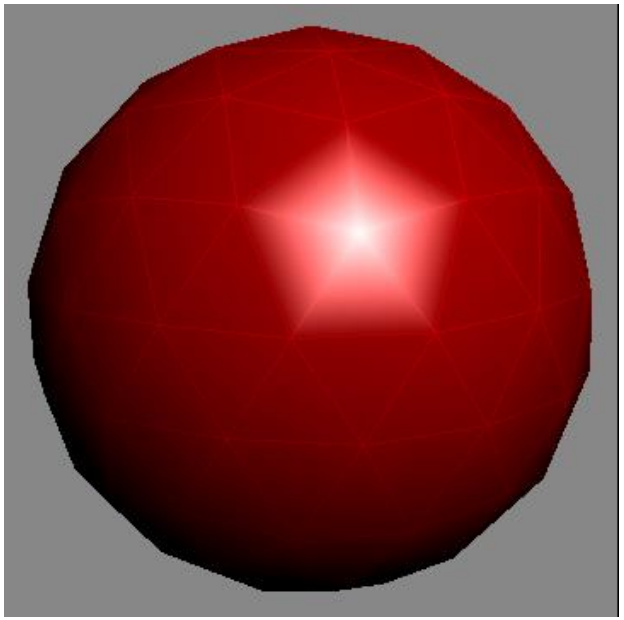
Flat-Shading vs Gouraud Shading



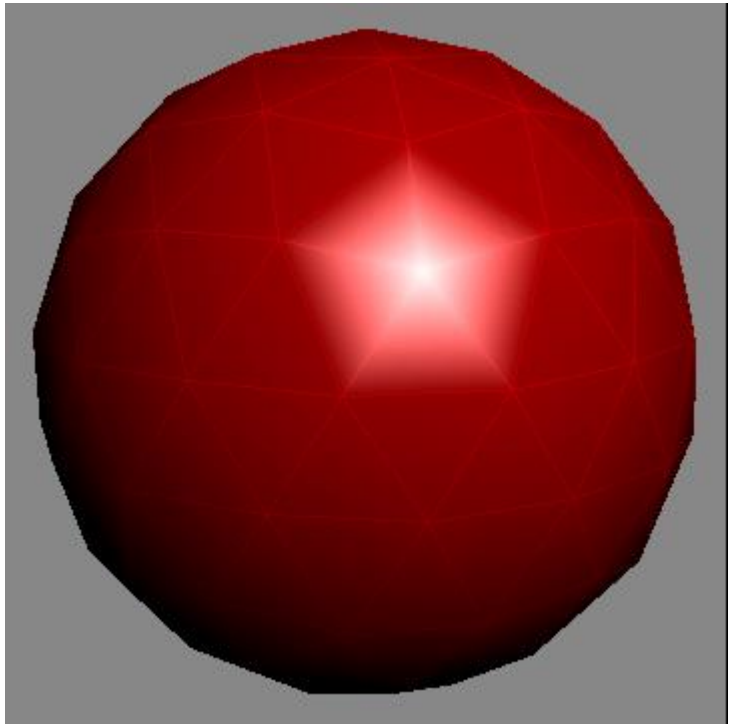
[Wikipedia]

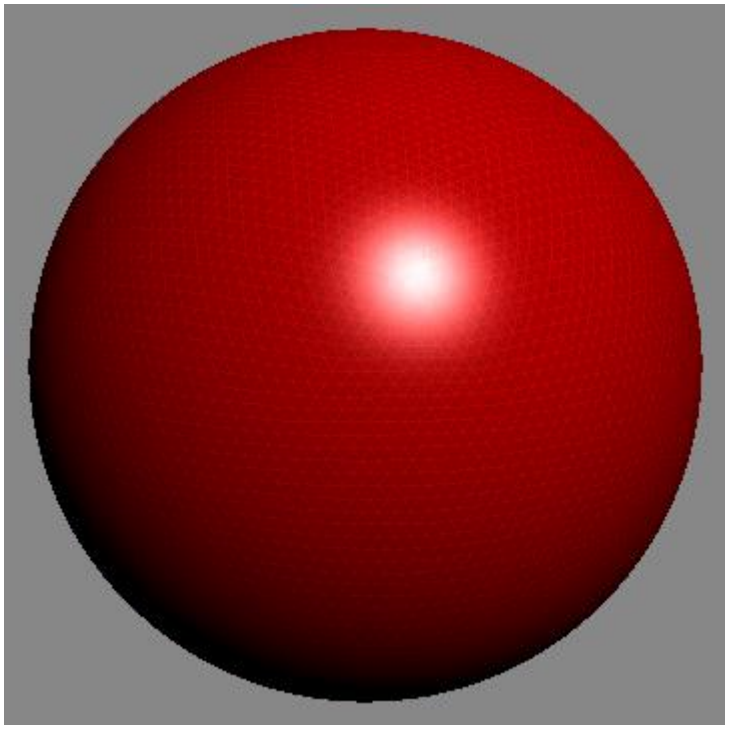
Gouraud Shading – 1971

- For each triangle / polygon
  - Apply the illumination model at each vertex
  - Interpolate color to shade each pixel
- Better than flat-shading
- Problems with highlights
- Mach-effect



Gouraud Shading – More triangles !





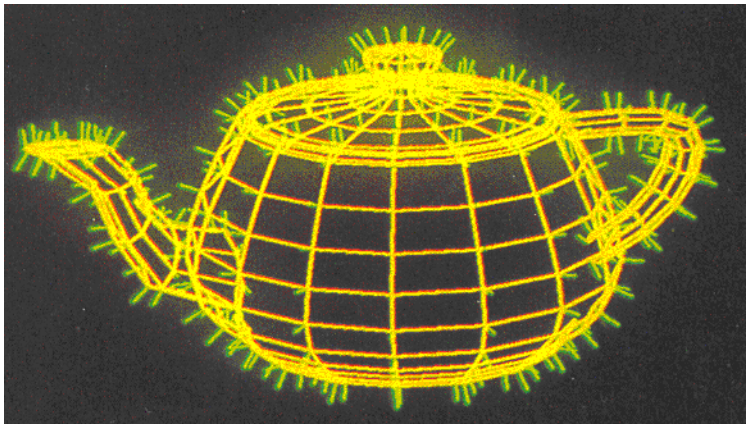
[Wikipedia]

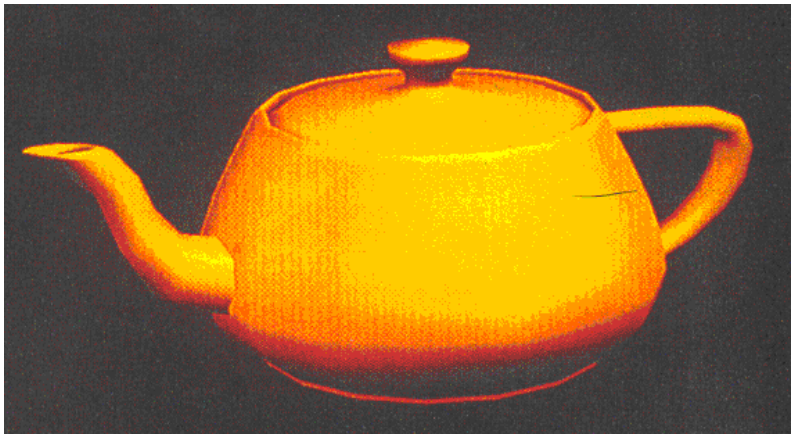
Gouraud Shading – Color-Interpolation Shading

- Computes color intensity values at mesh vertices and linearly interpolates color along the mesh polygons

- Developed by Henri Gouraud to render the surface of a curved object approximated by a polygonal mesh

- Achieves a smooth transition of intensity values among adjacent polygons by estimating, for each vertex, its "average" normal vector



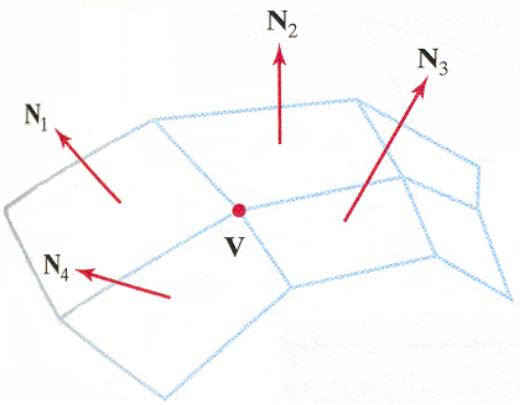


- Each mesh polygon is processed in three steps:

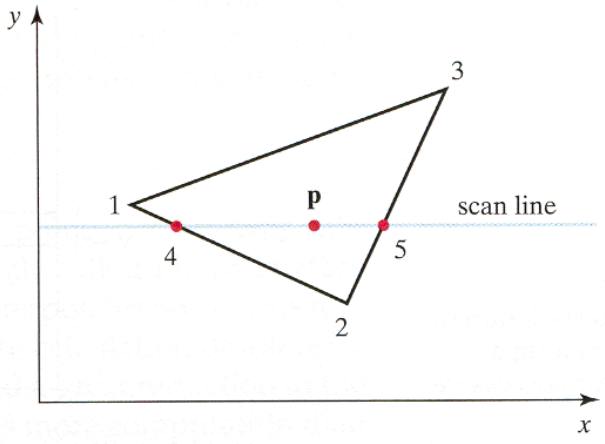
  1 – compute the “average” normal vector for each polygon vertex

  2 – apply the illumination model at each vertex to compute color intensity values

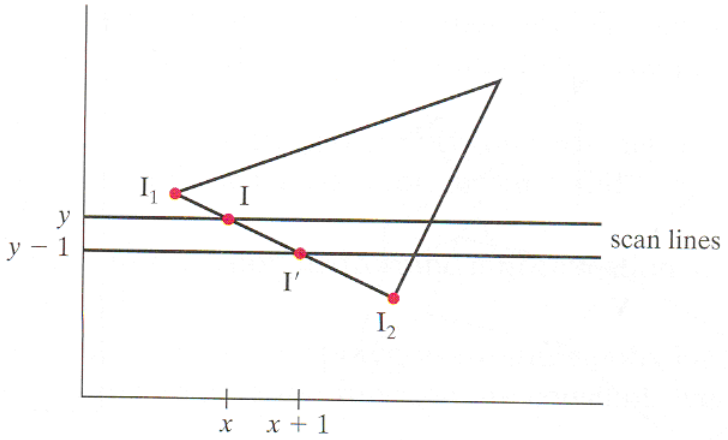
  3 – linearly interpolate color intensity values along the projected polygonal area



1

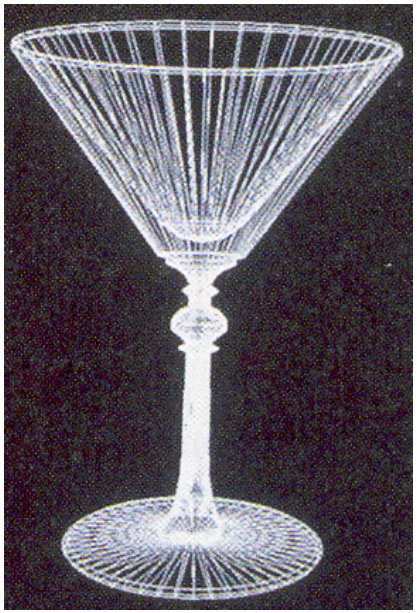


2

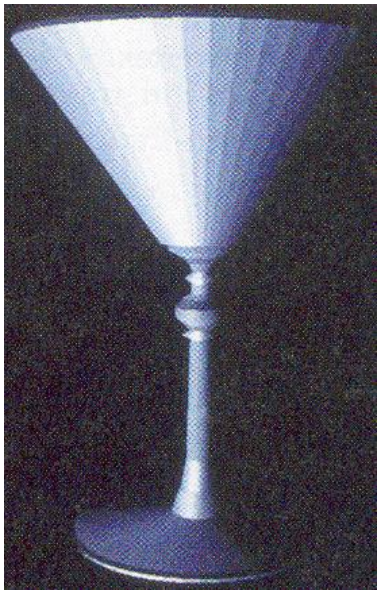


3

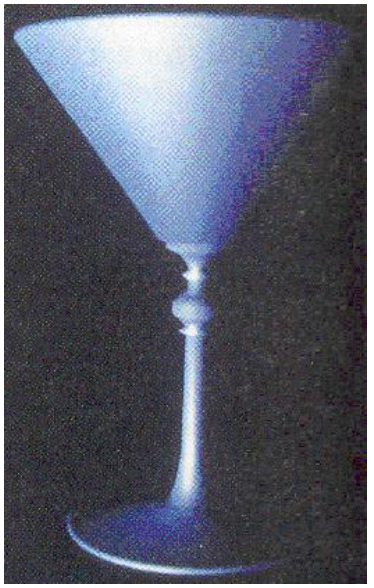
Smoother transitions



Polygonal mesh

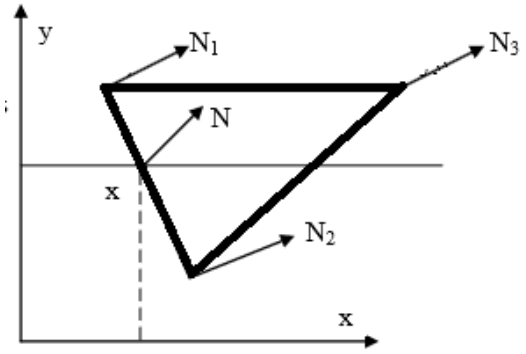


Flat-shading



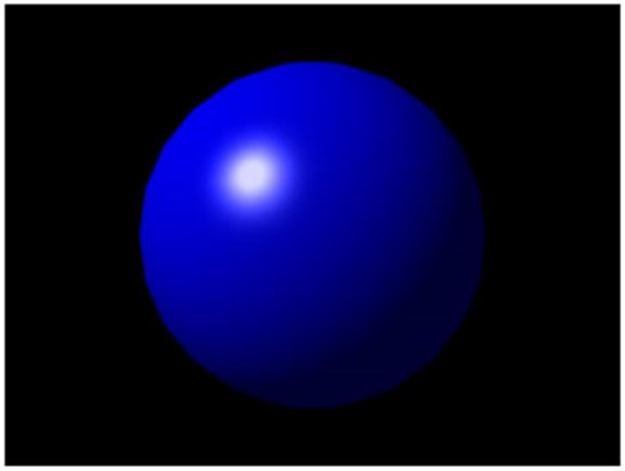
Gouraud shading

Phong Shading – 1973



- For each triangle / polygon
  - Interpolate normal vectors across rasterized polygons

- Better than Gouraud shading
- BUT, more time consuming



PHONG SHADING

Phong Shading – Normal-Interpolation Shading

- More sophisticated approach, developed by Phong Bui Tuong (1975)
- Linearly interpolates normal vectors (instead of color intensity values) and repeatedly applies the illumination model
- Produces more realistic images with:
  - Better specular reflections
  - Attenuated Mach Bands
  - But it requires more computations

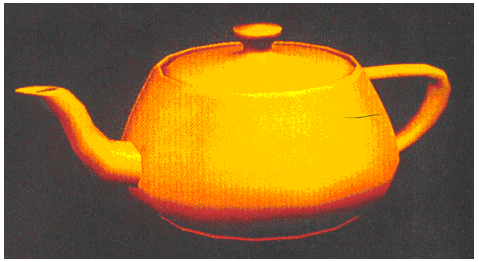
• Each mesh polygon is processed in three steps:

1 – compute the "average" normal vector for each polygon vertex

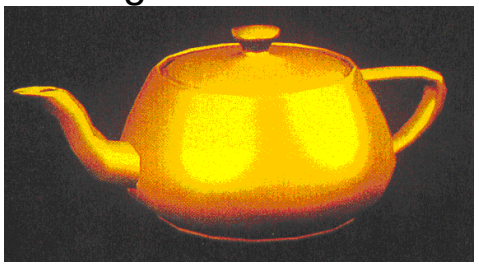
2 – linearly interpolate normal vectors along the projected polygonal area

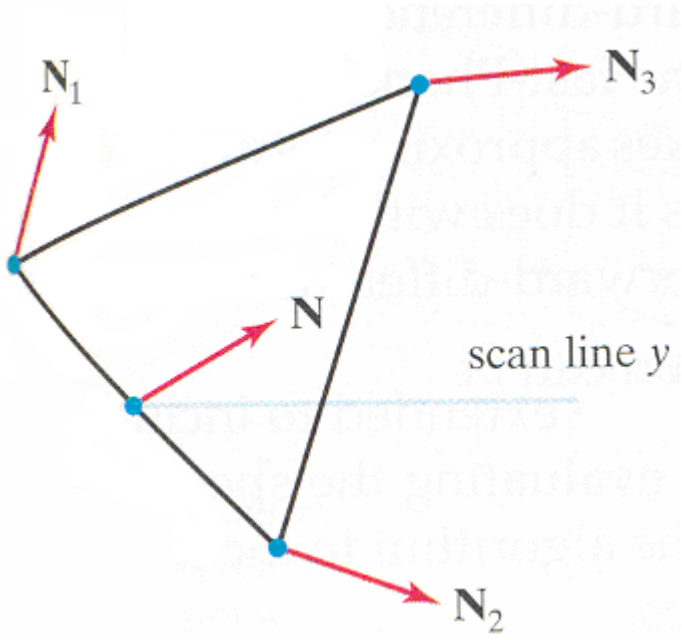
3 – apply the illumination model along the scan-lines to compute color intensity values using the interpolated normal vectors

Gouraud

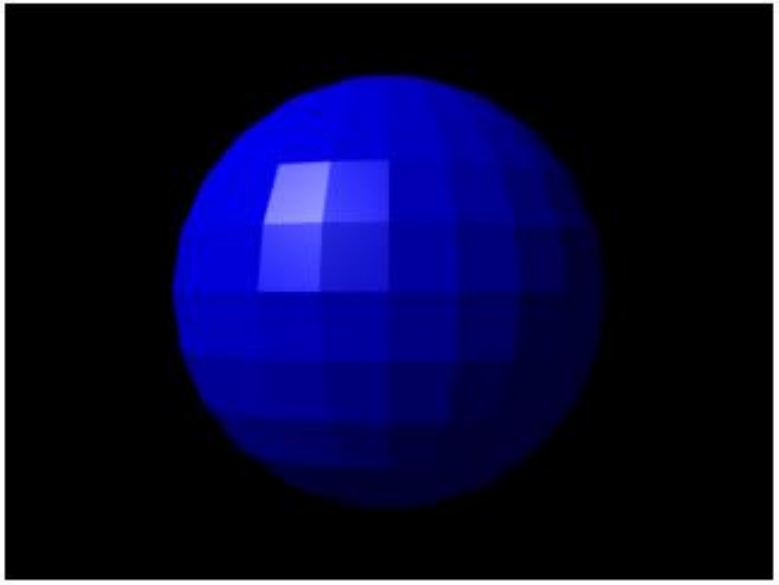


Phong

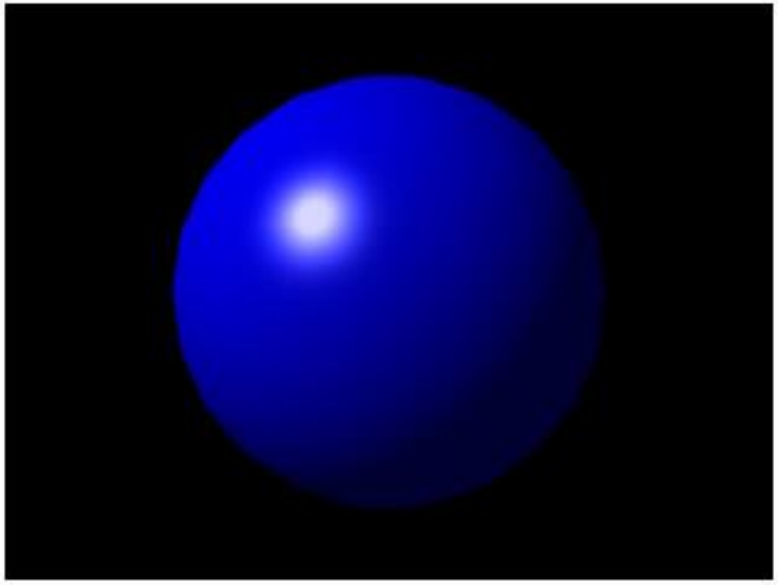




Flat-Shading vs Phong Shading



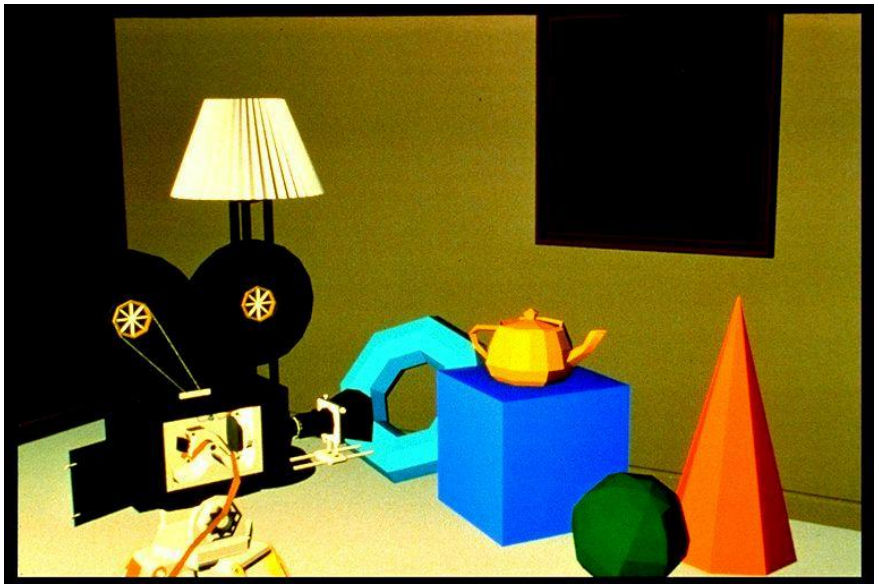
FLAT SHADING



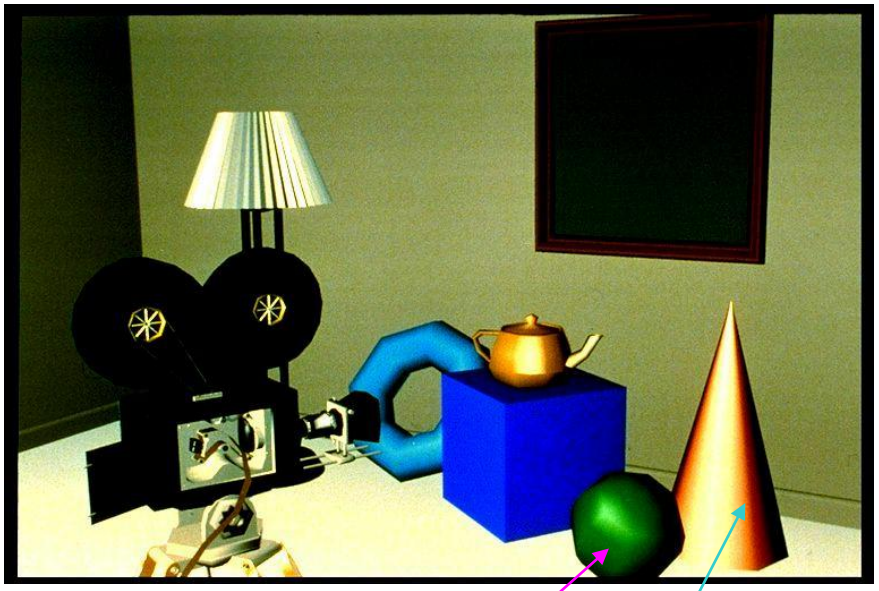
PHONG SHADING

[Wikipedia]

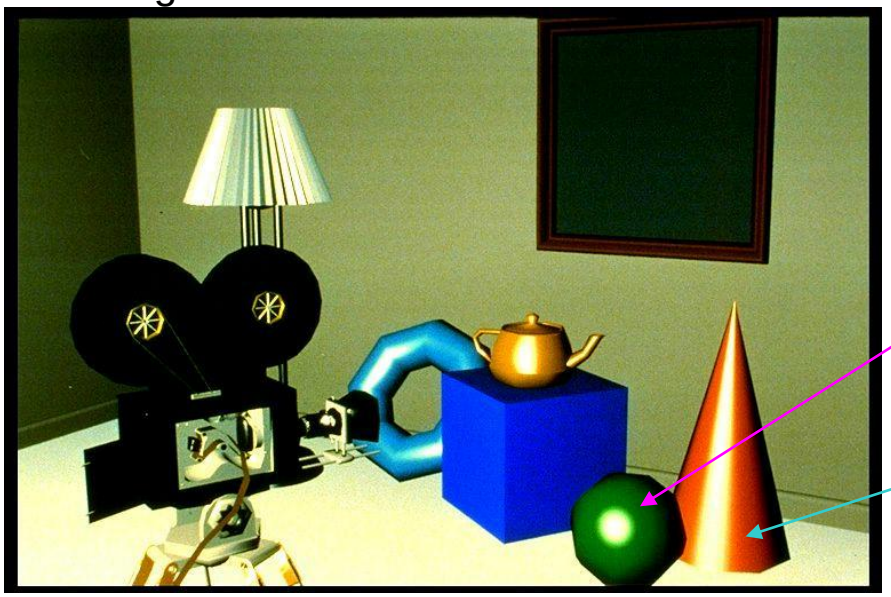
Flat



Gouraud



Phong



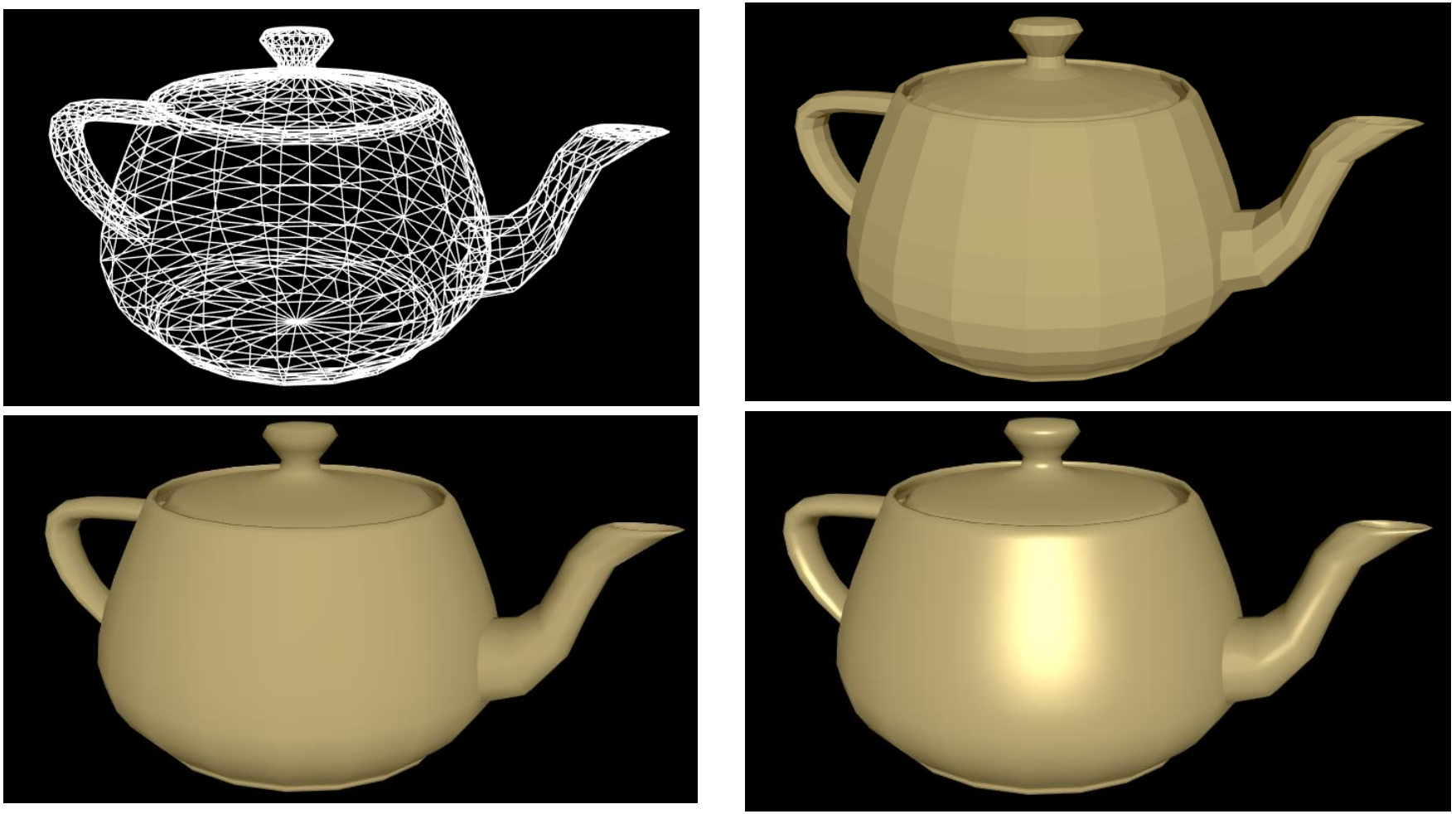
Improved *Highlights*

Less perceptible *Mach Bands*

Three.js – Example

[https://threejs.org/examples/#webgl_geometry_teapot]

Three.js – Examples



[https://threejs.org/examples/#webgl_geometry_teapot]

SHADING IN WEBGL

OpenGL / WebGL – *Per Fragment Shading*

```
// vertex shader
attribute vec4 vPosition;
attribute vec3 vNormal;

// output values that will be interpolated per-fragment
varying vec3 fN;
varying vec3 fE;
varying vec3 fL;

uniform mat4 ModelView;
uniform vec4 LightPosition;
uniform mat4 Projection;
```

OpenGL / WebGL– *Per Fragment Shading*

```
void main()
{
    fN = vNormal;
    fE = vPosition.xyz;
    fL = LightPosition.xyz;

    if( LightPosition.w != 0.0 ) {
            fL = LightPosition.xyz - vPosition.xyz;
    }

    gl_Position = Projection*ModelView*vPosition;
}
```

OpenGL / WebGL – *Per Fragment Shading*

```
// fragment shader
precision mediump float;
// per-fragment interpolated values from the vertex shader
varying vec3 fN;
varying vec3 fL;
varying vec3 fE;

uniform vec4 AmbientProduct, DiffuseProduct, SpecularProduct;
uniform mat4 ModelView;
uniform vec4 LightPosition;
uniform float Shininess;
```

OpenGL / WebGL – *Per Fragment Shading*

```
void main()
{
    // Normalize the input lighting vectors

   vec3 N = normalize(fN);
    vec3 E = normalize(fE);
    vec3 L = normalize(fL);

    vec3 H = normalize( L + E );
    vec4 ambient = AmbientProduct;
```

OpenGL / WebGL – *Per Fragment Shading*

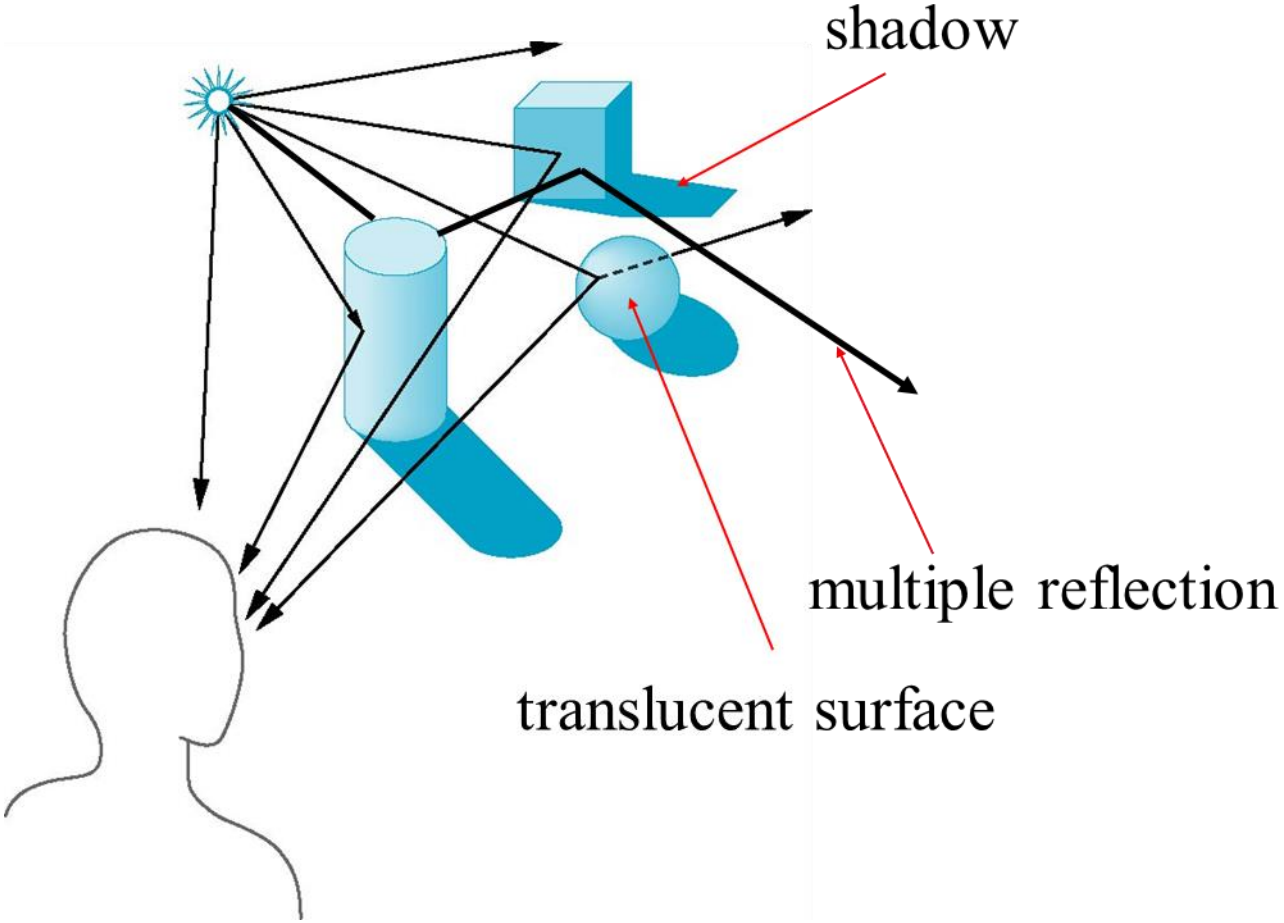
```
    float Kd = max(dot(L, N), 0.0);
    vec4 diffuse = Kd*DiffuseProduct;

    float Ks = pow(max(dot(N, H), 0.0), Shininess);
    vec4 specular = Ks*SpecularProduct;

    // discard the specular highlight if the light's behind the vertex
    if( dot(L, N) < 0.0 )
        specular = vec4(0.0, 0.0, 0.0, 1.0);

    gl_FragColor = ambient + diffuse + specular;
    gl_FragColor.a = 1.0;
}
```

Teapot Examples

[Angel]

RAY-TRACING

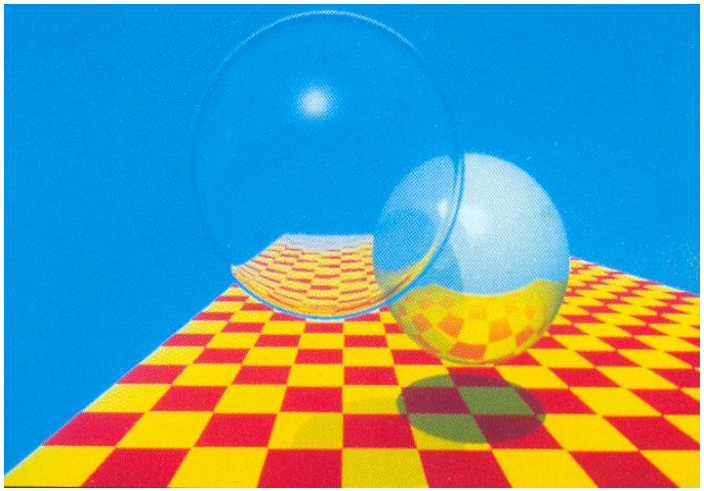
Global illumination

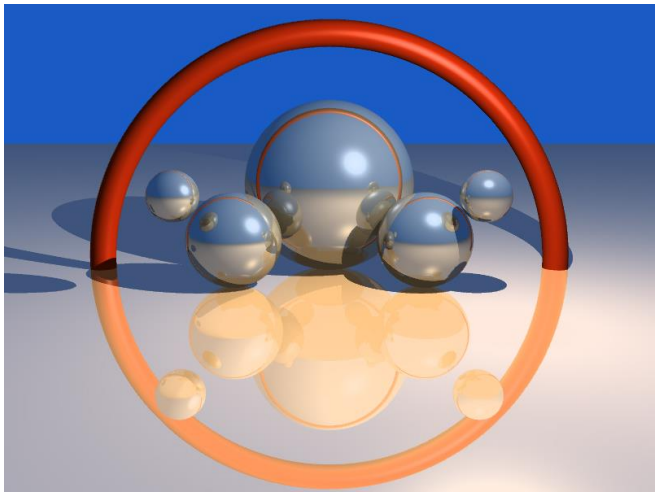


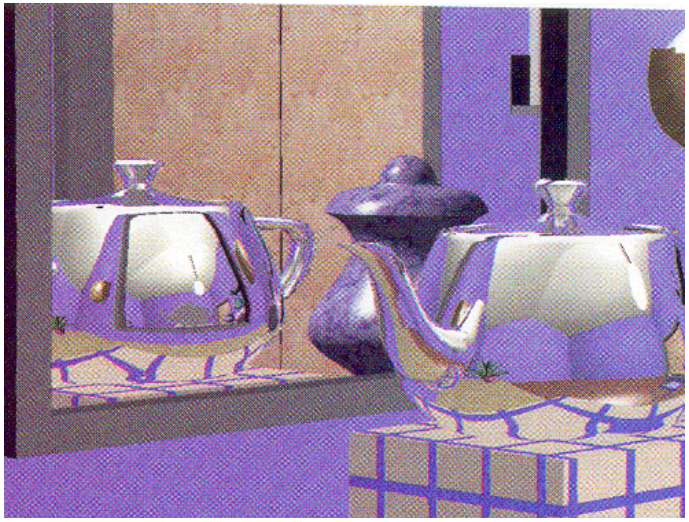
[Angel]

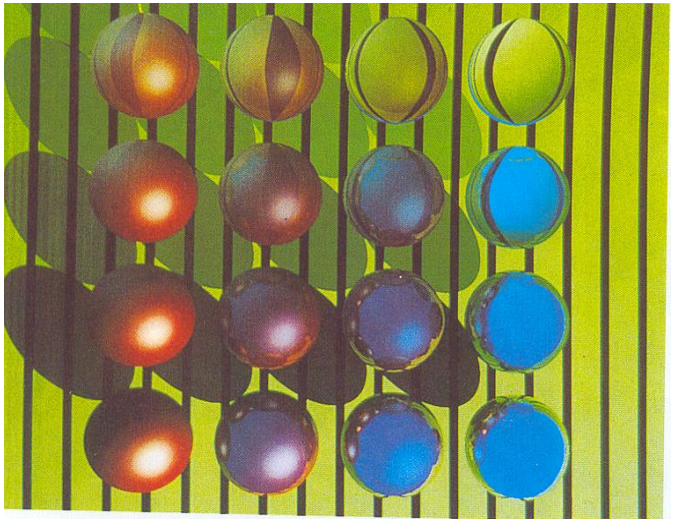
Global illumination:

Example images generated with *ray-tracing*





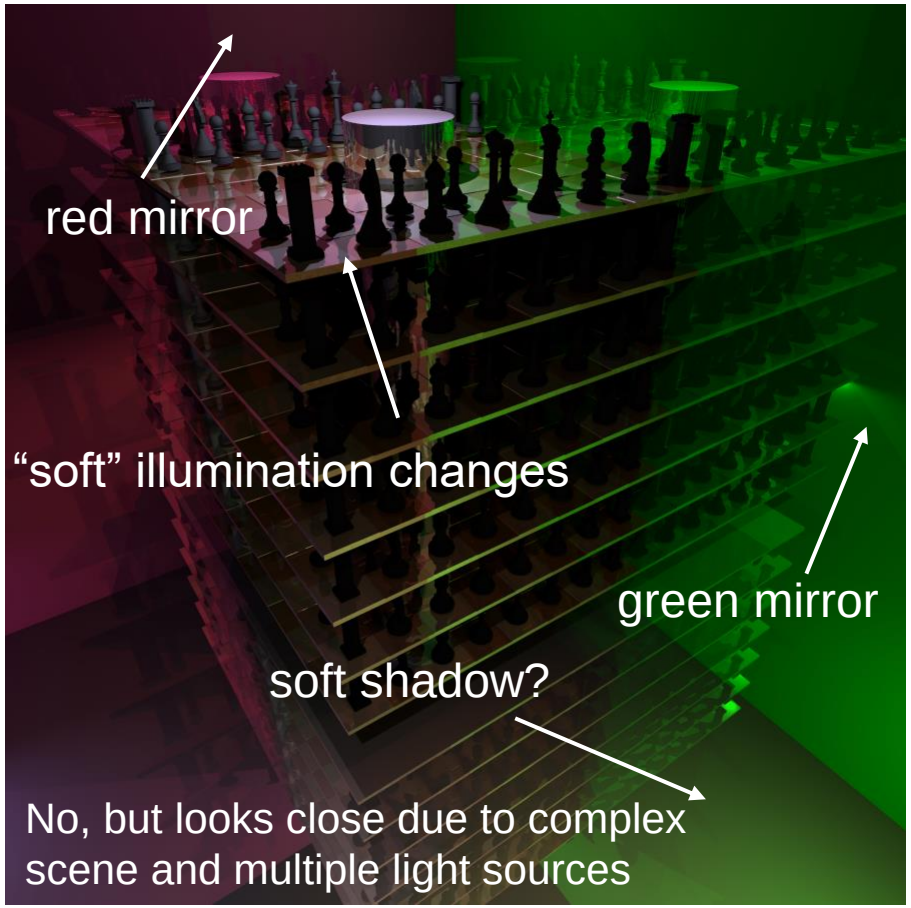




Ray-Tracing

What “effects” do you see?

Rendered in a matter of seconds with Travis Fischer’s ’09 ray tracer

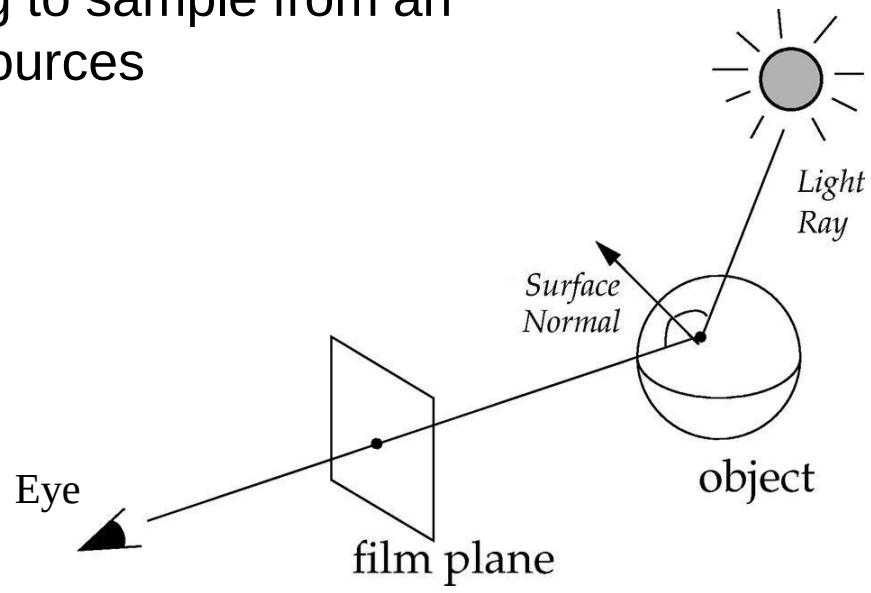


[Andy Van Dam]

What is a Ray-tracer?

- A finite back-mapping of rays from camera (eye) through each sample (pixel or subpixel) to objects in scene

- To avoid forward solution of having to sample from an infinite number of rays from light sources



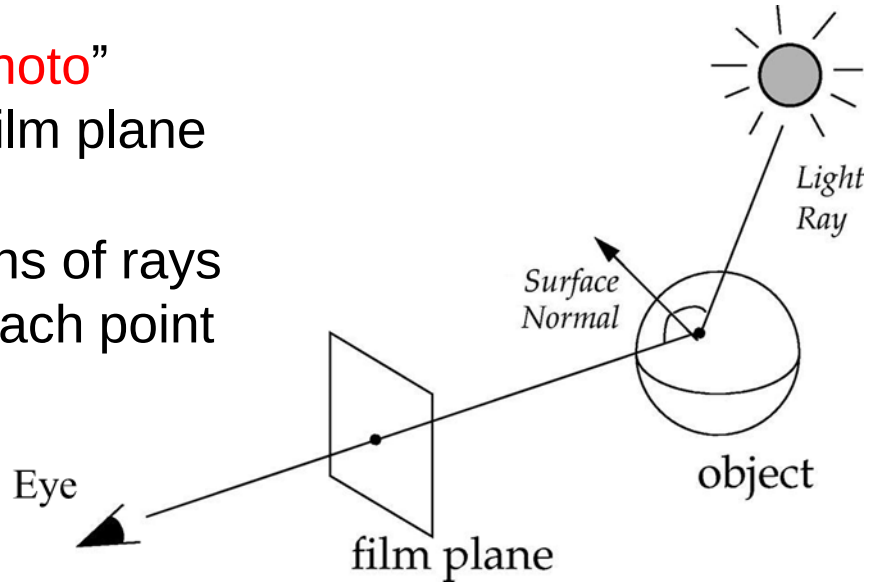
[Andy Van Dam]

What is a Ray-tracer?

- Each pixel represents either:
  - a ray intersection with an object / light source in scene
  - no intersection

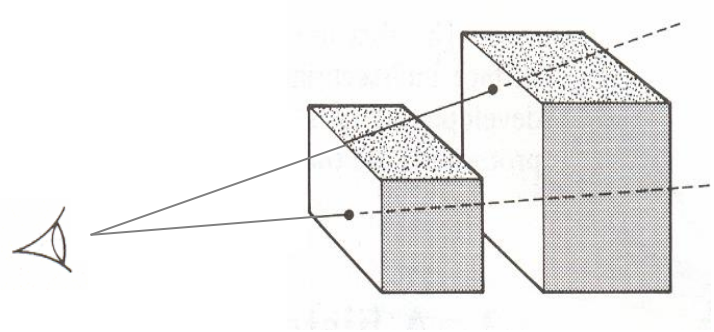
- A ray traced scene is a "virtual photo" comprised of many samples on film plane

- Generalizing from one ray, millions of rays are shot from eye, one through each point on film plane



[Andy Van Dam]

Ray-Casting

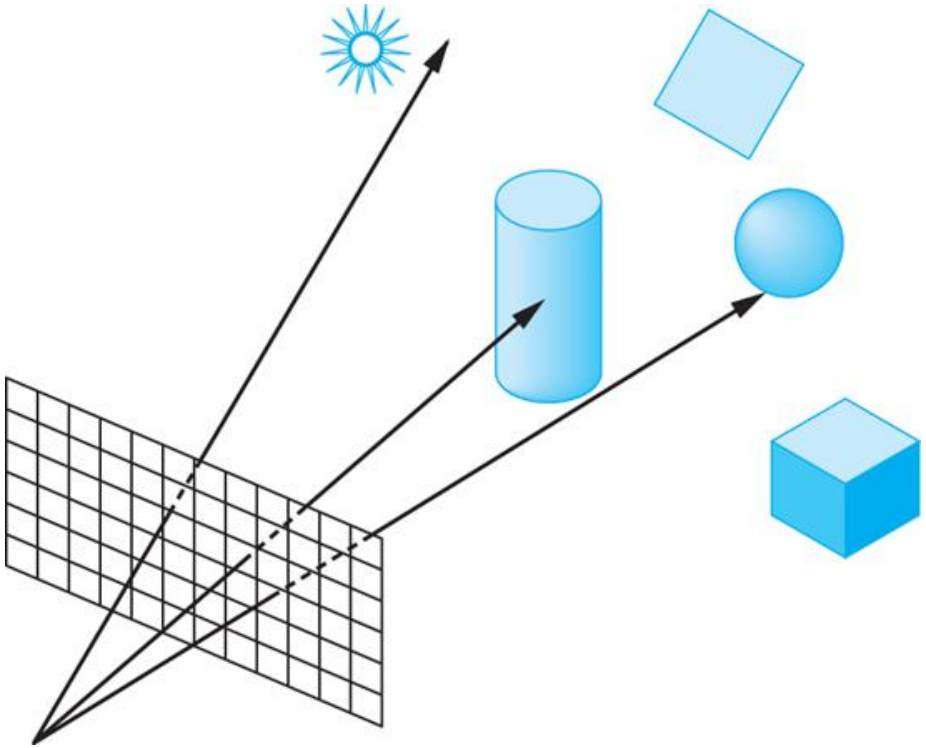


- Image-space method

- Which models are intersected by a ray cast from the viewpoint that passes through the center of a pixel ?

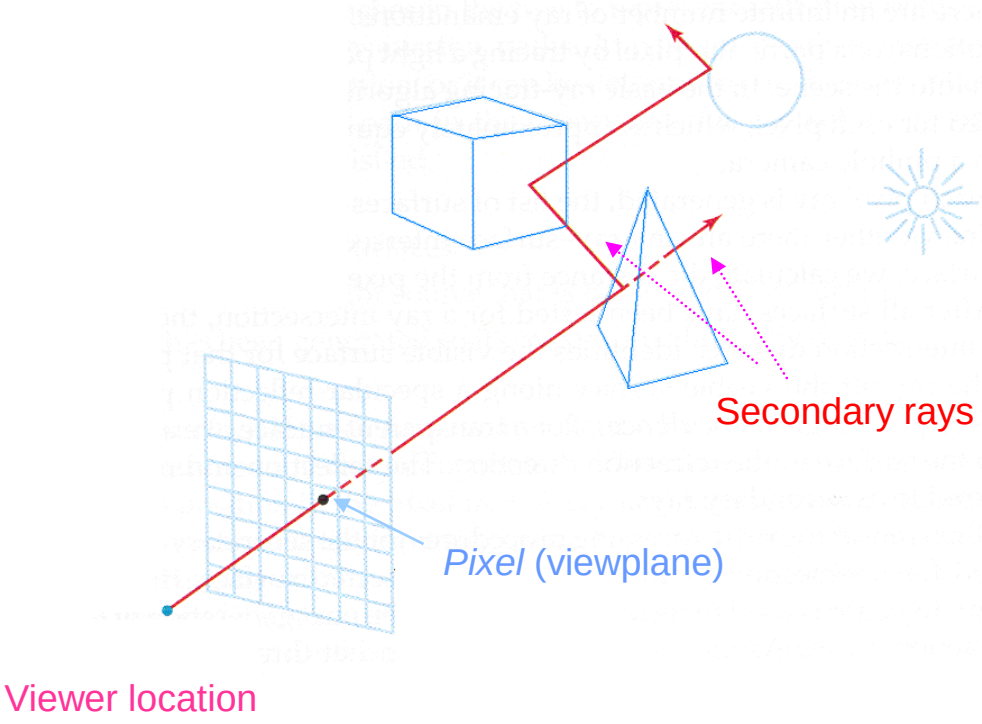
- If models are opaque, the intersection point closer to the viewpoint determines the color (i.e., visible surface) for the corresponding pixel

Ray Casting

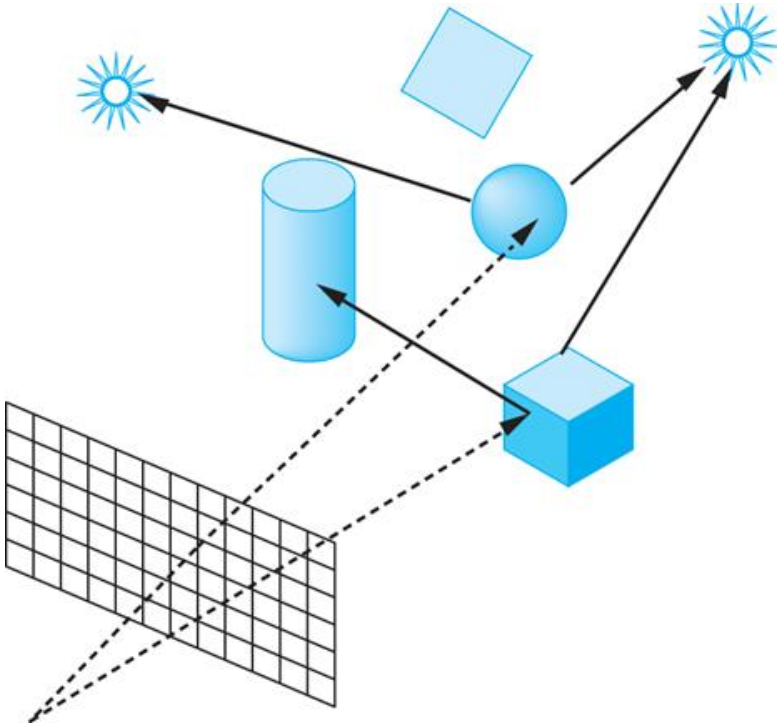


[Angel]

- In addition, might cast **secondary rays** (reflected or transmited)
- Whenever there are secondary rays: ***ray-tracing***

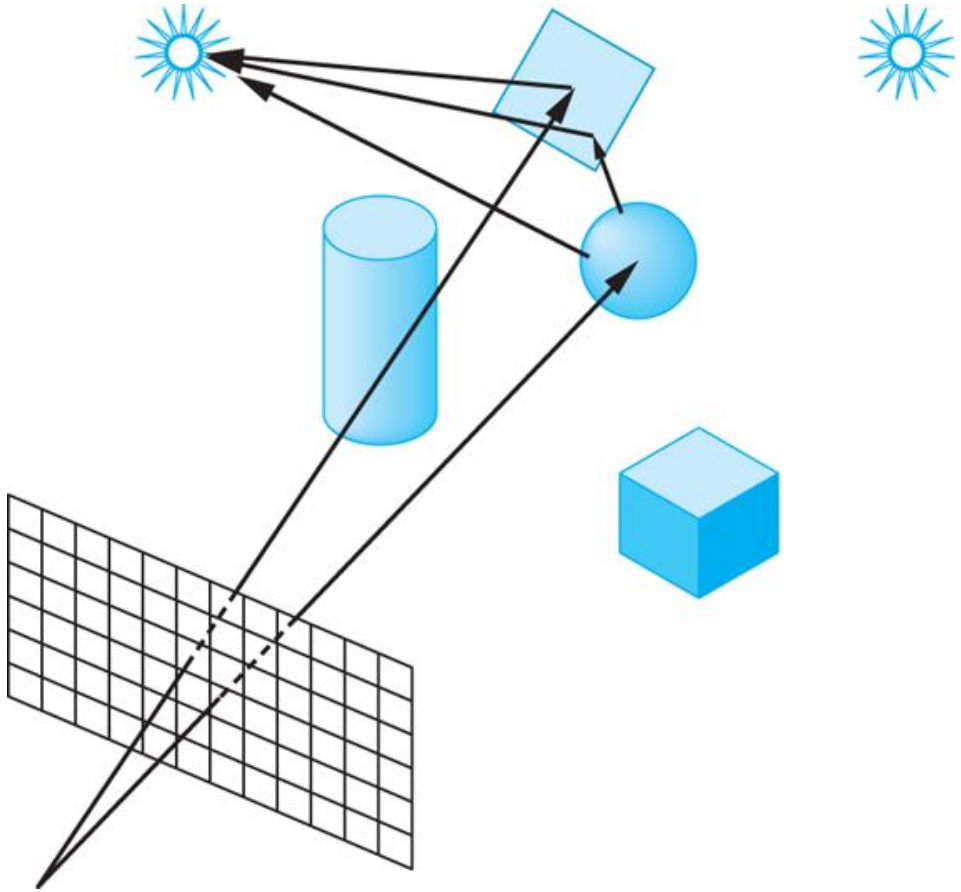


Ray Tracing: Shadow or feeler rays



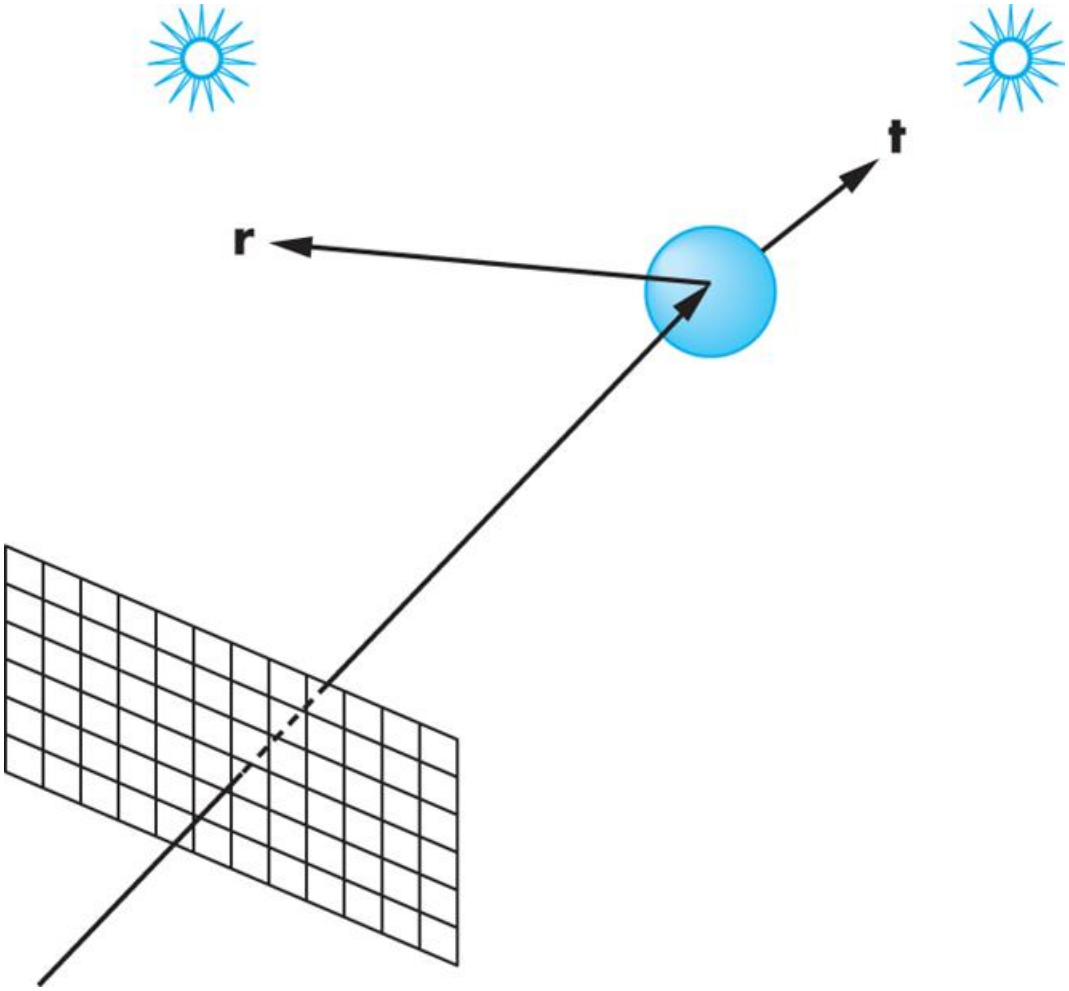
[Angel]

Ray Tracing: Reflecting surfaces



[Angel]

Ray Tracing: Transmitting surfaces



[Angel]

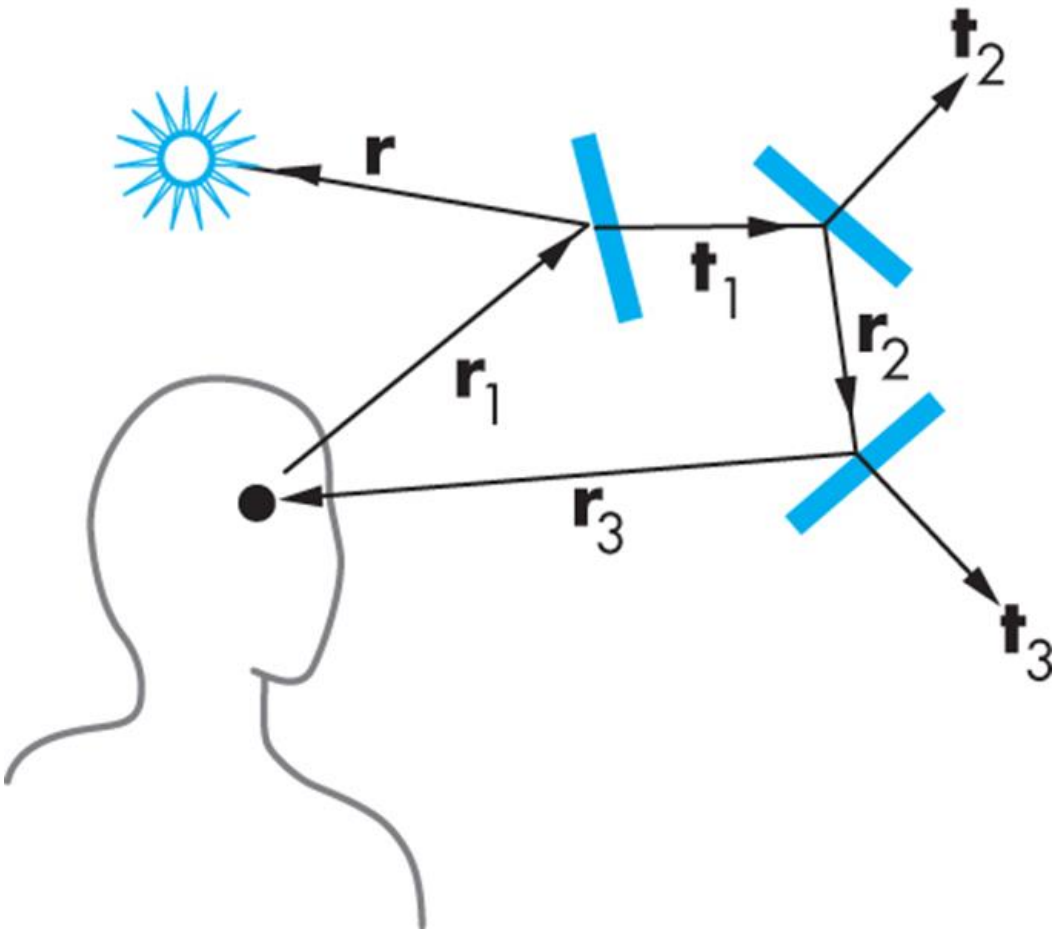
Ray Tracing Fundamentals

- Generate primary ray
  - shoot rays from eye through sample points on film plane
  - sample point is typically center of a pixel
- Ray-object intersection
  - find first object in scene that ray intersects with (if any)
  - solves VSD/HSR problem

Ray Tracing Fundamentals

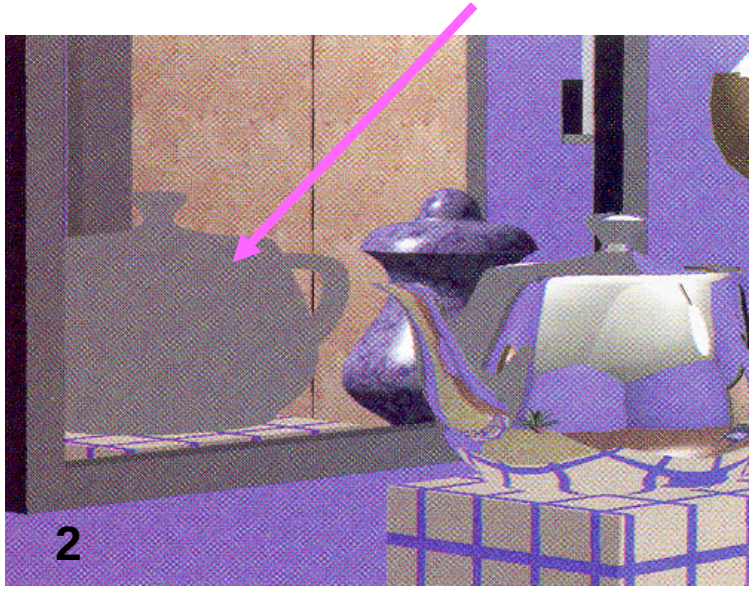
- Calculate lighting (i.e., color)
  - use illumination model to determine direct contribution from light sources (light rays)
  - reflective objects recursively generate secondary rays (indirect contribution) that also contribute to color
- Sum of contributions determines color of sample point
- RT uses specular reflection rays
- No diffuse reflection rays !
- RT is limited approximation to global illumination

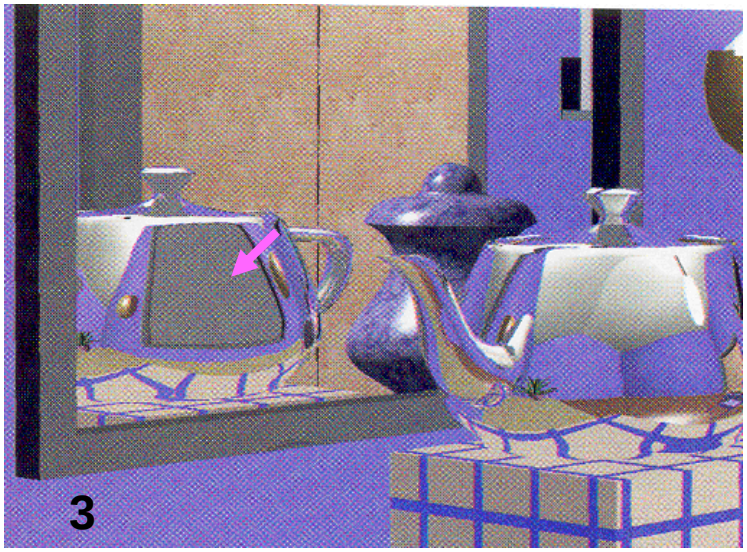
Ray Tracing: Ray trees



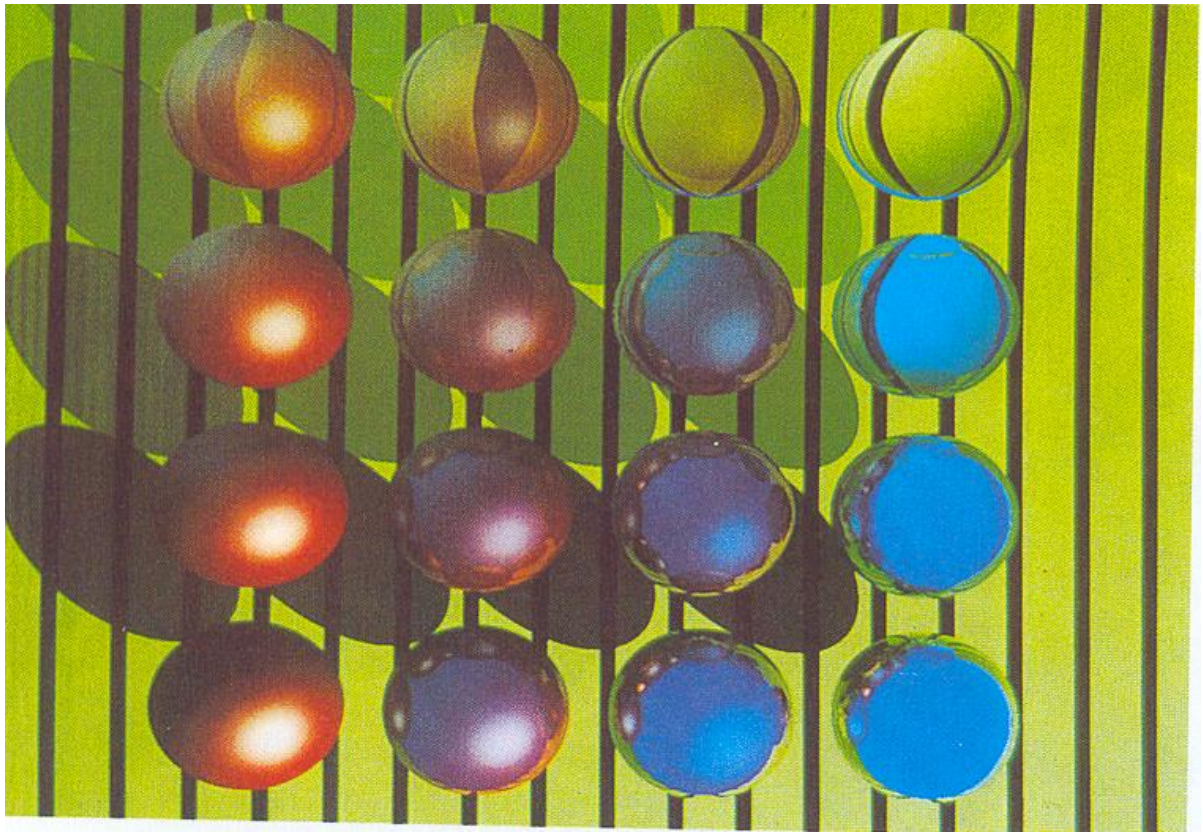
[Angel]

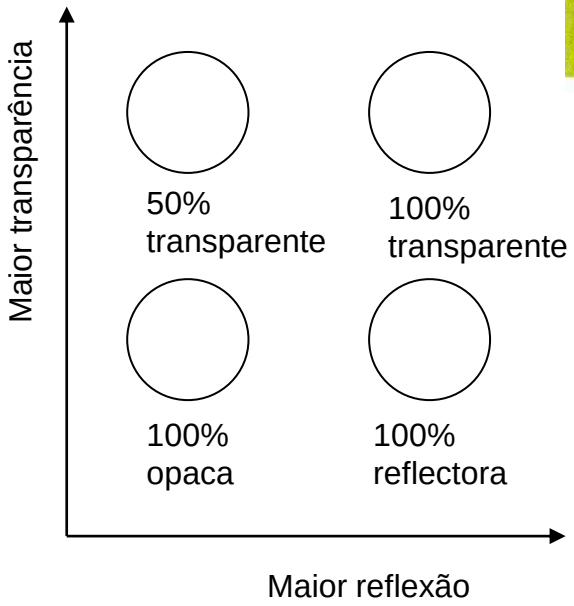
Pixels without an assigned color



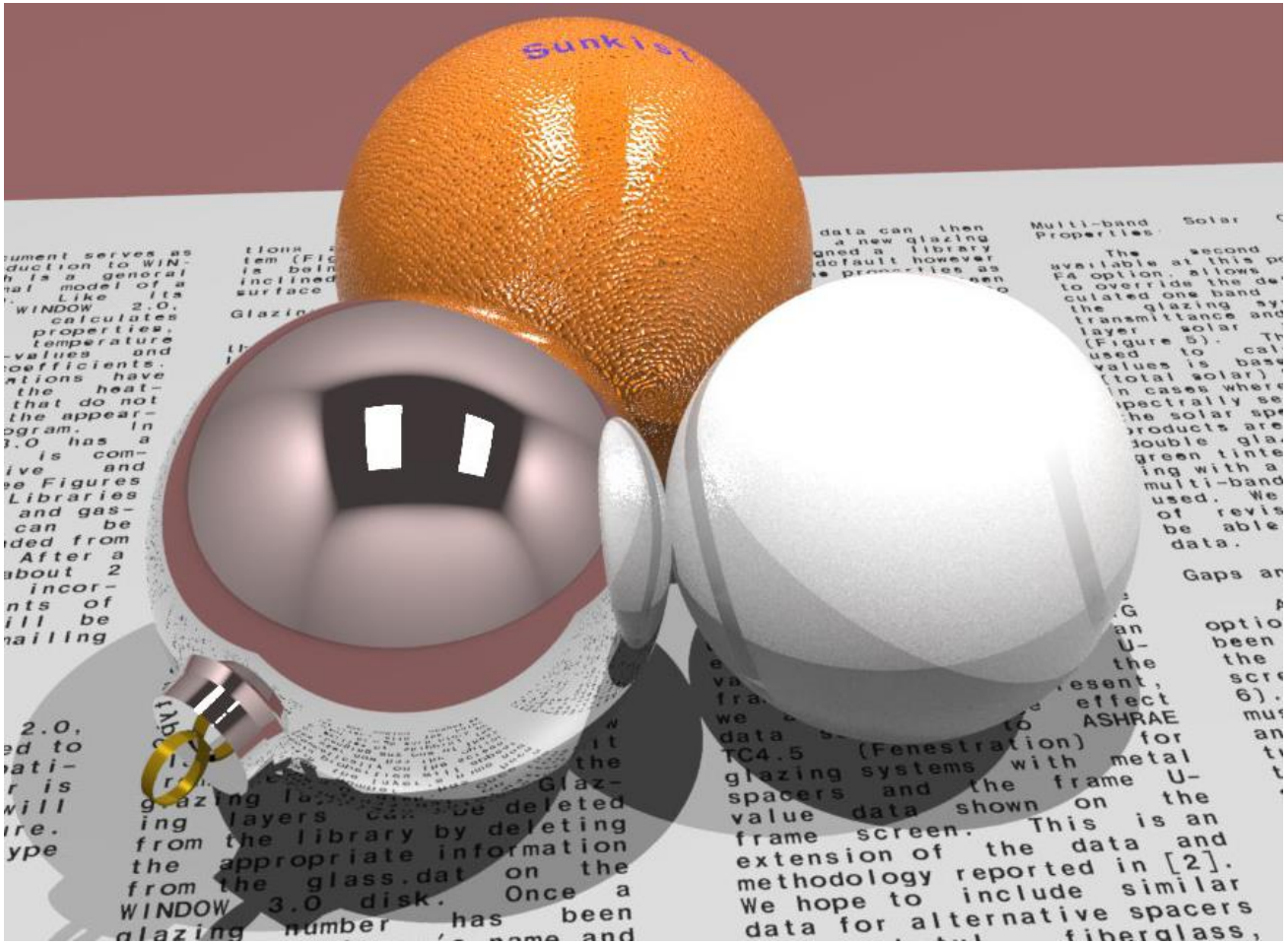


Ray tracing with varying depth:
2, 3 and 4 secondary rays



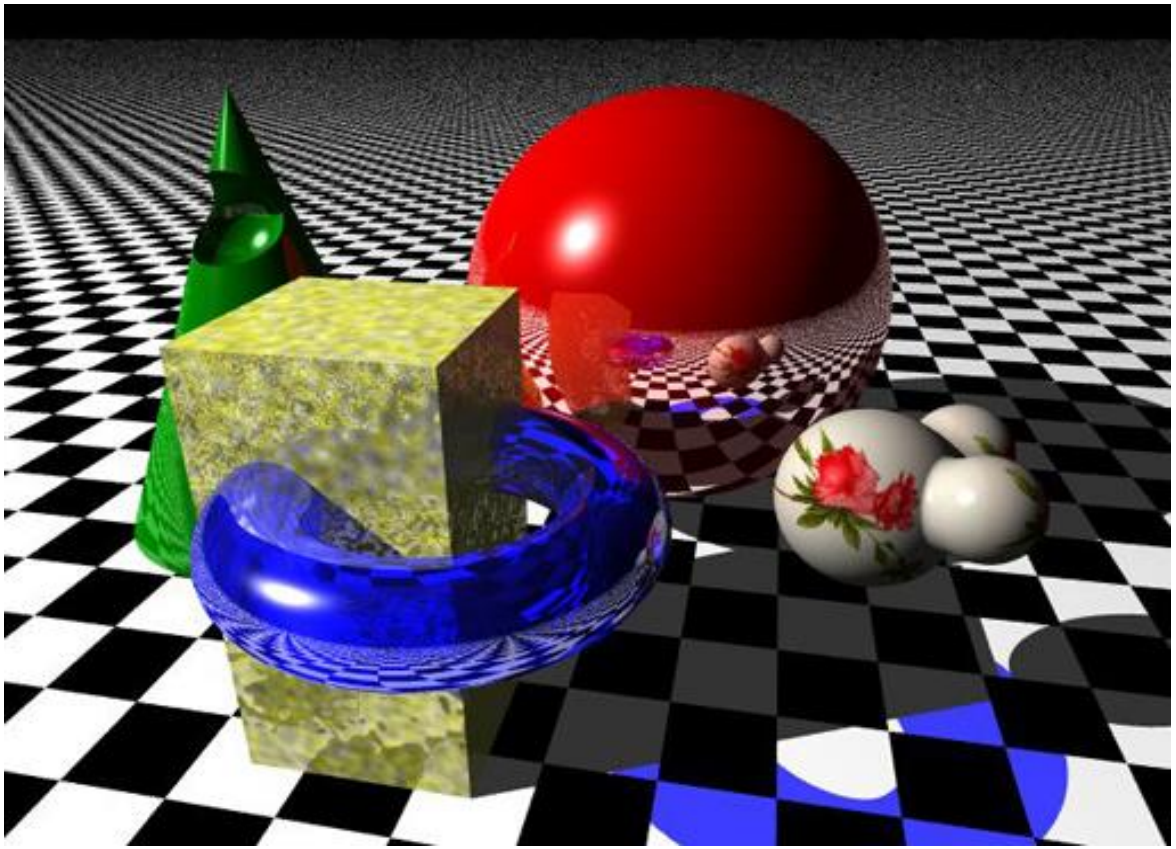

Maior transparência
50%
transparente
100%
transparente
100%
opaca
100%
reflectora
Maior reflexão

Ray-Tracing example



http://radsite.lbl.gov/radiance/book/img/plate10.jpg

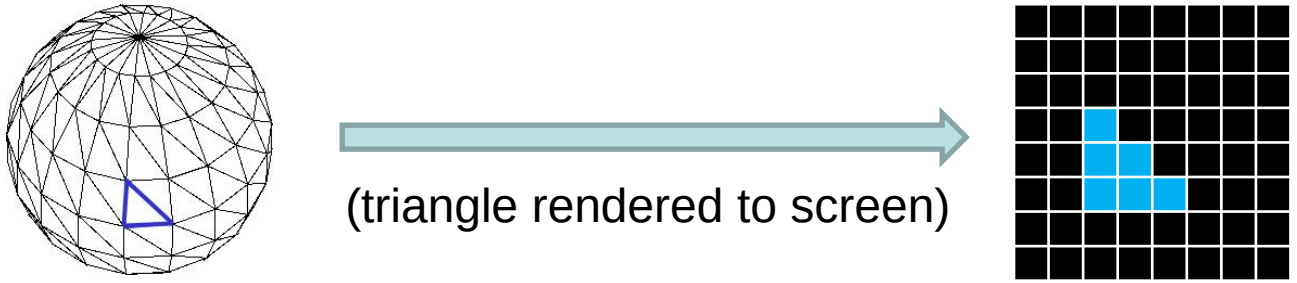
Another Ray-Tracing example



http://www.tjhsst.edu/~dhyatt/superap/samplex.jpg

Ray Tracing vs. Triangle Scan Conversion

```
for each object in scene:
    for each triangle in object:
        pass vertex geometry, camera matrices, and
        lights to the shader program,
        which renders each triangle (using z-buffer)
        into framebuffer; use simple or complex
              illumination model
```

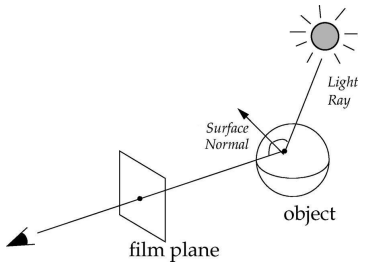


[Andy Van Dam]

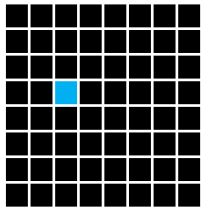
Ray Tracing vs. Triangle Scan Conversion

```
for each sample in film plane:
    determine closest object in scene hit by
        a ray going through that sample from eye point
    set color based on calculation of simple/complex
        illumination model for intersected object
```

- No need to mesh curved objects; we can use an implicit equation directly



(ray intersection rendered to screen)



[Andy Van Dam]

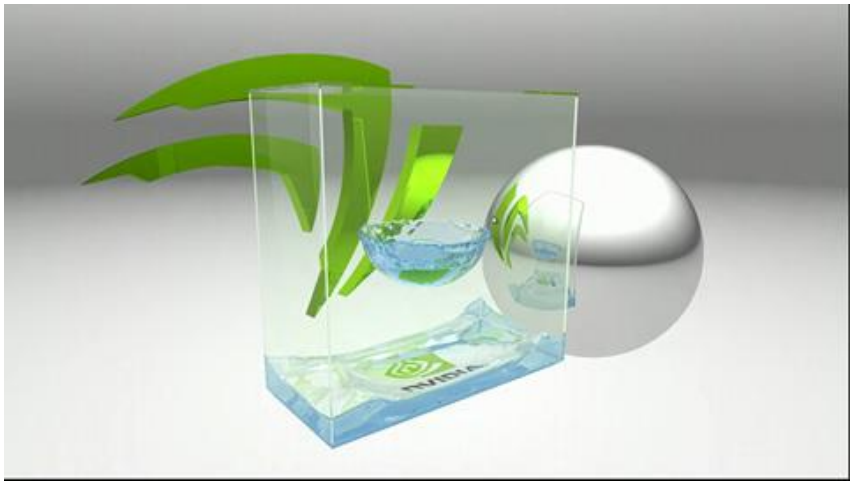
Real-time ray tracing

- It was impossible to do ray tracing in real-time !
- Still using “rendering farms” for non-real-time CG – movies and animations

Real-time ray tracing

- BUT, GPUs can be used to speed up ray tracing
- The architecture of recent GPUs is capable of real-time ray tracing !!

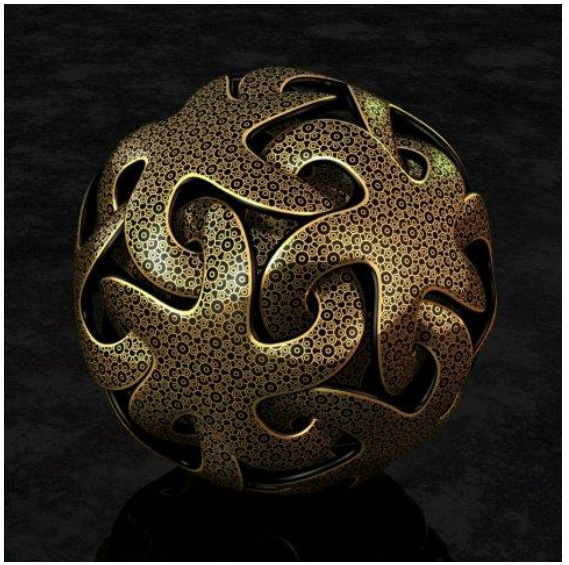
[van Dam]

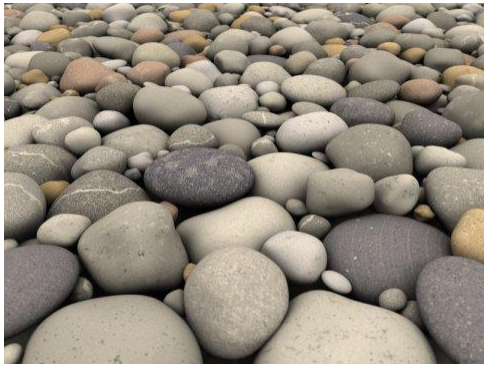


Scene ray traced in real time using NVIDIA Kepler

POV-Ray

- Free advanced ray-tracer !!
  - povray.org
- See the hall-of-fame !!
  - hof.povray.org





REFERENCES

References

- D. Hearn and M. P. Baker, *Computer Graphics with OpenGL*, 3rd Ed., Addison-Wesley, 2004

- E. Angel and D. Shreiner, *Introduction to Computer Graphics,* 7th Ed., Pearson Education, 2012

- J. Foley et al., *Introduction to Computer Graphics*, Addison-Wesley, 1993

Acknowledgments

- Some ideas and figures have been taken from slides of other CG courses.

- In particular, from the slides made available by Ed Angel and Andy van Dam.

- Thanks!