

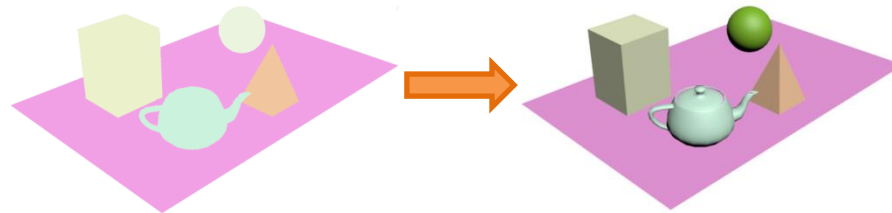
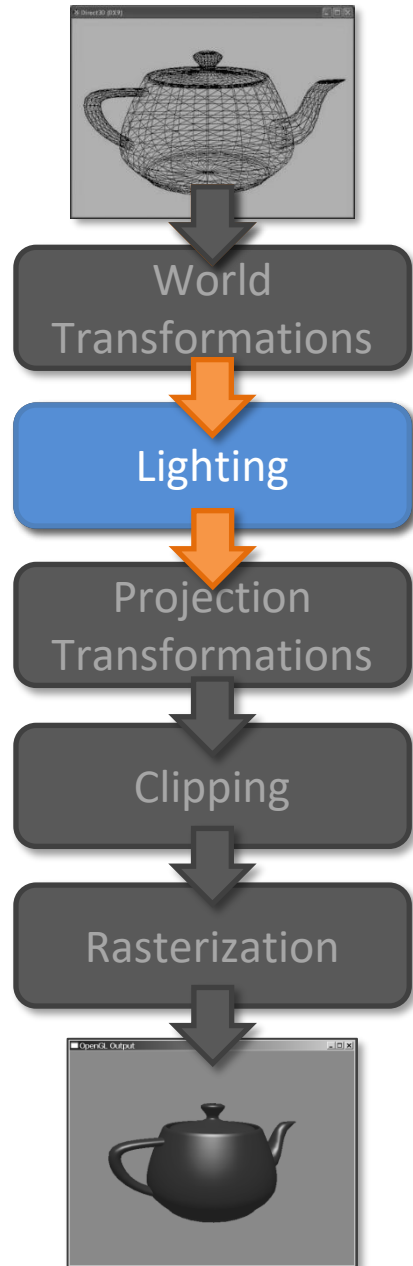
**SUPSI**

# Computer Graphics

## OpenGL (3): Lighting

Achille Peternier, adjunct professor







A diagram illustrating the concept of incidence angle. A red laser beam, labeled 'Light direction', originates from a red laser pointer on the right and strikes a vertical white rectangular surface. A horizontal blue arrow, labeled 'Surface orientation', points to the right, parallel to the surface. The angle between the incident red beam and the surface is labeled 'incidence angle' in red. The beam reflects off the surface, and a bright red spot is visible at the point of incidence. The background is dark, and a red carpet is visible at the bottom.

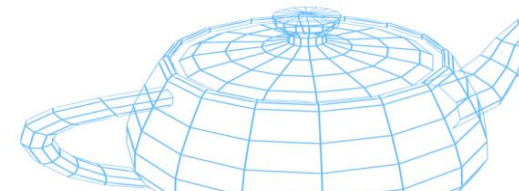
Surface orientation

Light direction

incidence  
angle

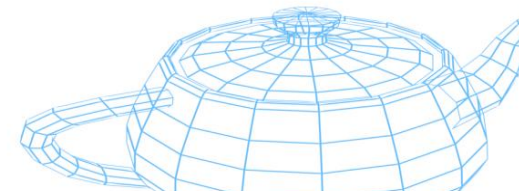
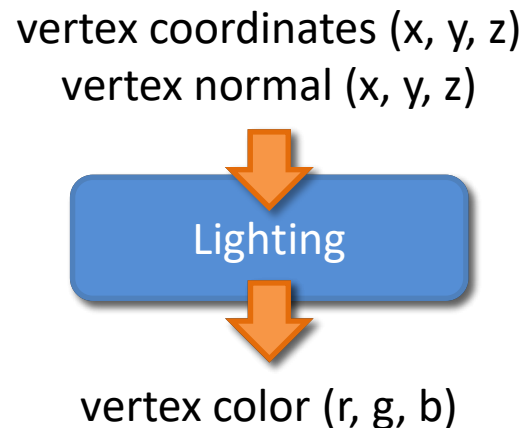
# Lighting

- So far, we specified vertex colors explicitly:
  - You can define vertex colors according to your own artistic taste or by using a custom lighting model.
  - ...or you can use OpenGL fixed-pipeline lighting implementation and let it compute vertex colors for you.



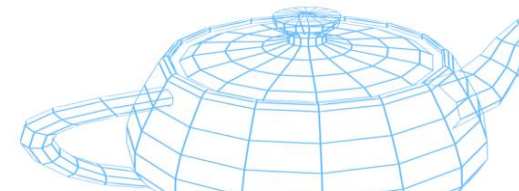
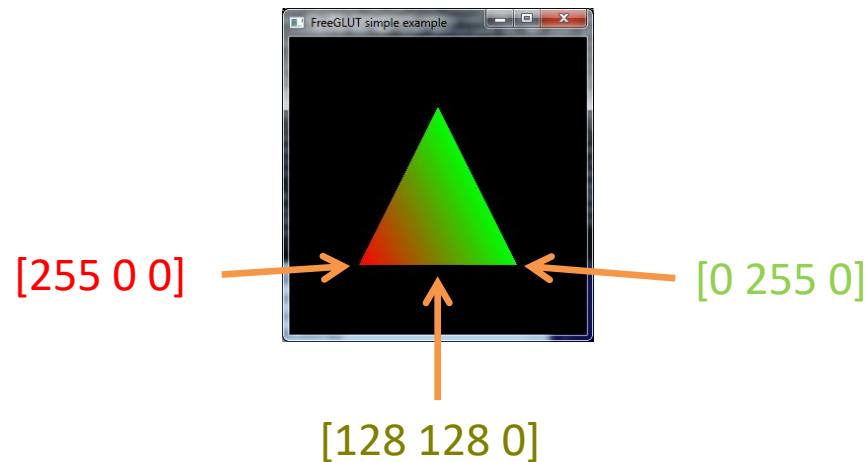
## Lighting

- For each vertex, you specify coordinates and the normal vector.
- OpenGL fills the vertex colors for you, according to the lighting settings you specified (e.g., light sources, material properties, etc.).



# Lighting

- The lighting contribution is computed for each vertex:
  - Each vertex will get a color:
    - OpenGL fixed-pipeline lighting works on **per-vertex** basis!
    - More advanced lighting implementations work on per-fragment basis.
  - Colors are interpolated during the rasterization step.



Rendering

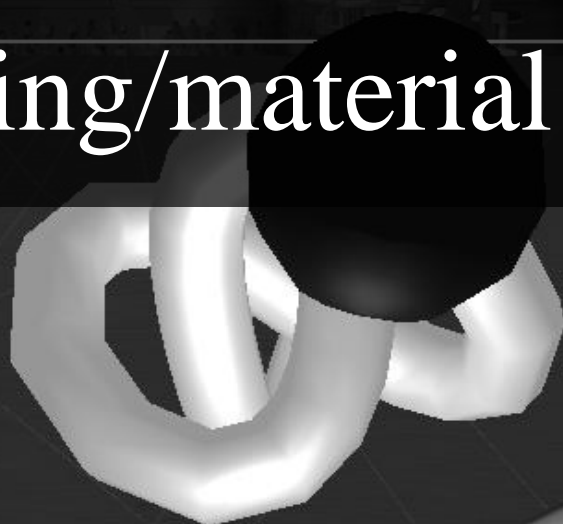
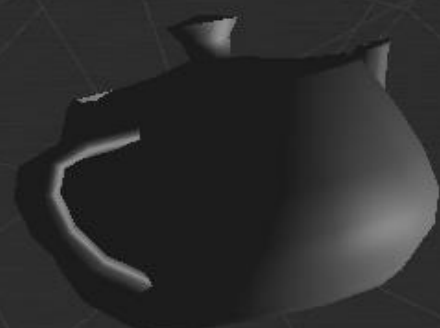
Exit module

Shading model

Flat

Gouraud

# Lighting/material demo



## Material settings

Object selected: Torus

Ambient:

RGB(36, 37, 21)

Diffuse:

RGB(249, 255, 71)

Specular:

RGB(55, 55, 55)

Shininess:

value: 12

Transparency:

value: 1.00

## Light settings

Ambient:

RGB(127, 127, 127)

Diffuse:

RGB(127, 127, 127)

Specular:

RGB(127, 127, 127)

Global ambient:

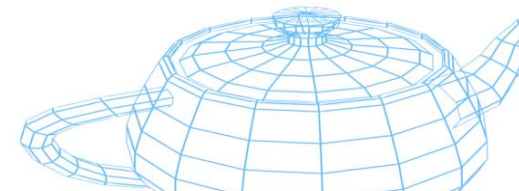
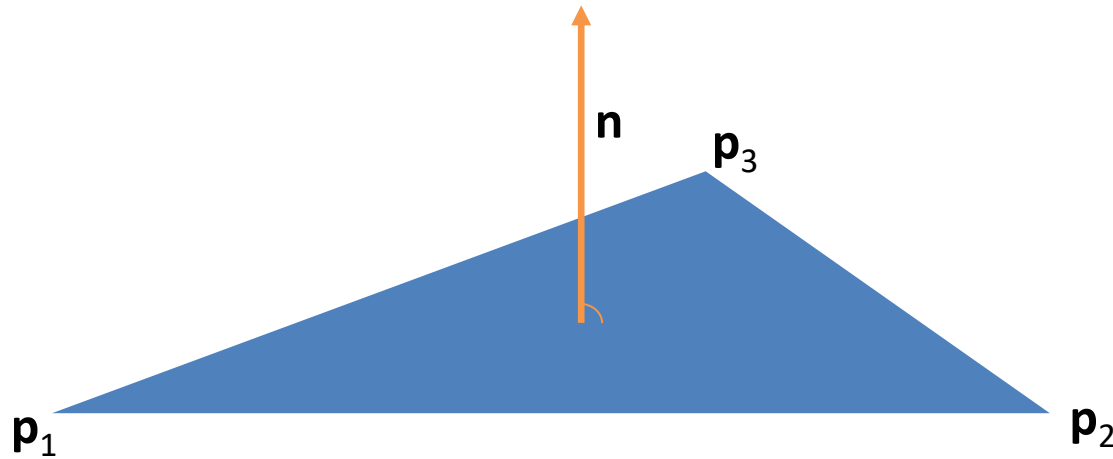
RGB(127, 127, 127)



Virtual Reality Laboratory

## Normal

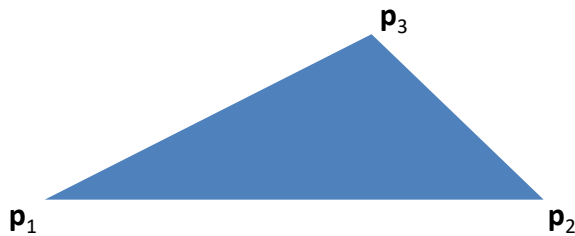
- A triangle's normal is a vector perpendicular to the plane defined by its three vertices.





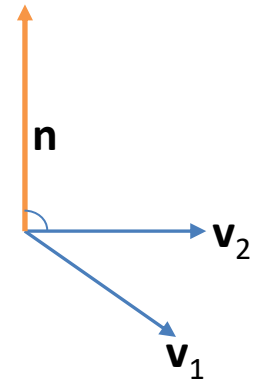
## Normal

- A triangle's normal is computed using the **cross** product between two vectors being part of the triangle's plane:

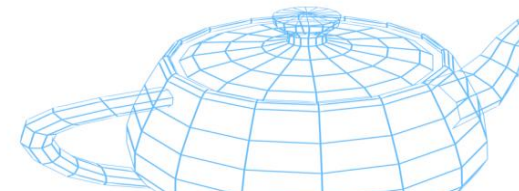


$$\mathbf{v}_1 = \mathbf{p}_2 - \mathbf{p}_1, \quad \mathbf{v}_2 = \mathbf{p}_3 - \mathbf{p}_1$$

$$\mathbf{n} = \mathbf{v}_1 \times \mathbf{v}_2$$

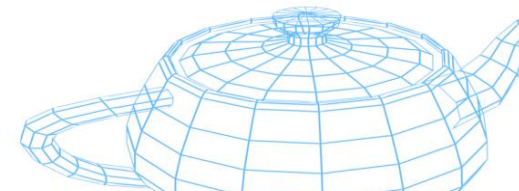
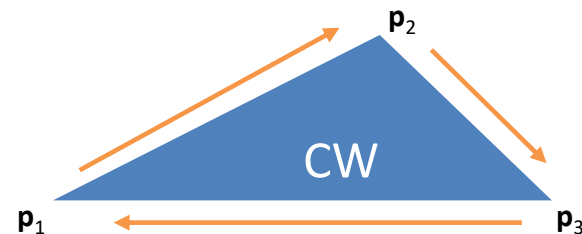
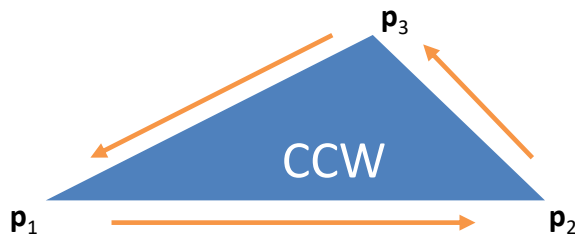


- The order used to specify vertices defines the direction of the normal (in/out).



## Face orientation

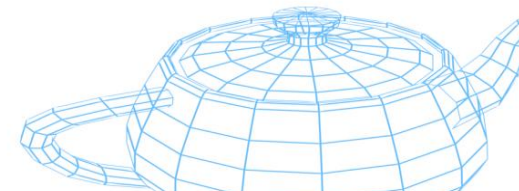
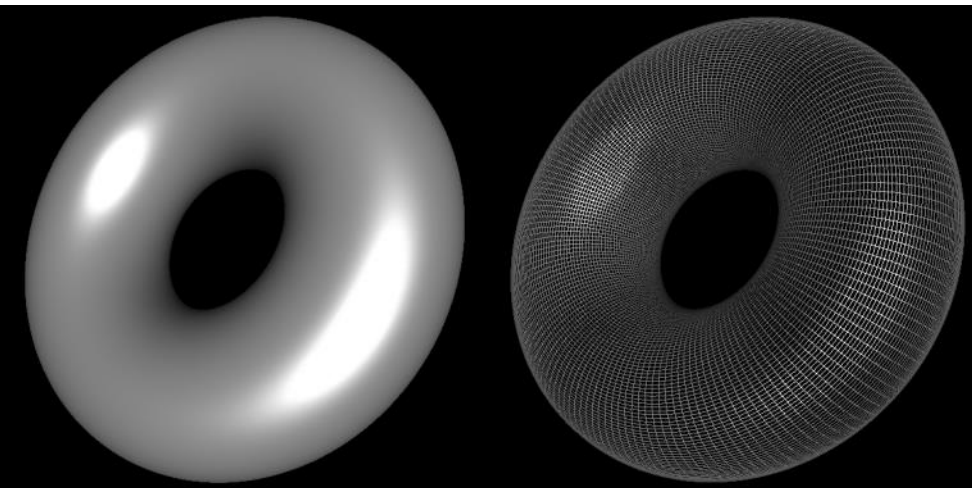
- Triangles have two faces (front and back).
- Their orientation is defined by the order used to pass vertices (vertex winding order).
- The front face is defined (by default) by passing vertices in counterclockwise (CCW) direction.

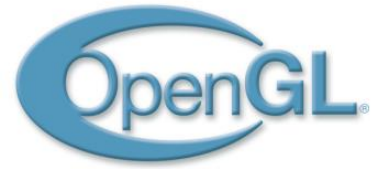




## Face orientation

- The way front/back face orientation is determined can be changed from counter-to clockwise:
  - `glFrontFace(GL_CW);` // default is GL\_CCW
- Use `glPolygonMode()` to change the way OpenGL renders front and back faces:
  - `glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);` // Wireframe
  - `glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);` // Solid



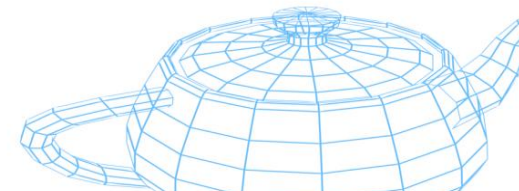


## Face culling

- Only renders front faces by discarding (culling) back faces.
- Good performance optimization when closed meshes are used, e.g., most solid geometric objects (cubes, spheres, pyramids, etc.) and 3D models:
  - Avoid to waste rasterization resources by rendering triangles that are always hidden behind front faces.
- Disabled by default, it is activated using:  
`glEnable(GL_CULL_FACE);`
- Culling mode is changed using:  
`glCullFace(flag);`



where possible *flags* are `GL_BACK` (default), `GL_FRONT`, and `GL_FRONT_AND_BACK`





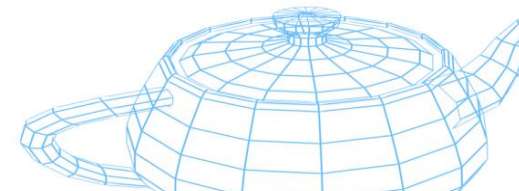
# Tutorial

Face orientation



## Per-vertex information

- Vertex position
  - $x, y, z[, w]$  (usually as *float*)
- Vertex color (e.g. RGB or RGBA)
  - $r, g, b[, a]$  (usually as *byte*)
- Vertex normal
  - $x, y, z$  (usually as *float*)



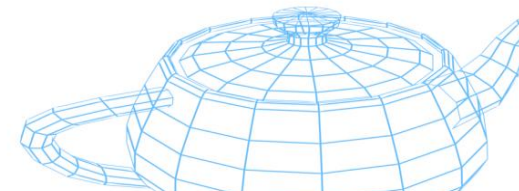


## Per-vertex information

- It's **up to you** to provide normal vectors to OpenGL via `glNormal3*()`.
- Normal vectors are defined per-vertex (each vertex has one normal).
- Normal vectors **must be normalized!**
  - Normal vectors are multiplied by the inverse transpose of the current matrix already loaded with `glLoadMatrix*()`:
    - Scaling requires to re-normalize normal vectors!
    - Activate `glEnable(GL_NORMALIZE)` to deal with that.

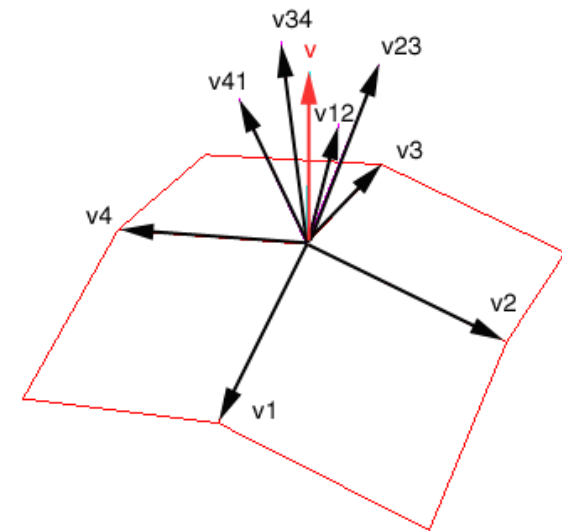


```
glBegin(GL_TRIANGLES);  
    glNormal3f(0.0f, 0.0f, 1.0f);  
        glVertex3f(0.0f, 0.0f, 0.0f);  
        glVertex3f(10.0f, 0.0f, 0.0f);  
        glVertex3f(5.0f, 5.0f, 0.0f);  
glEnd();
```



## Shared normal vectors

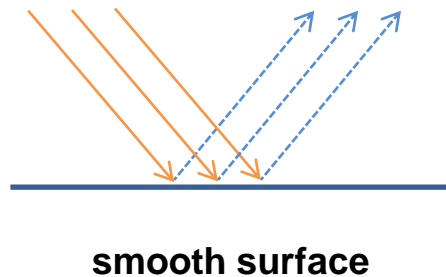
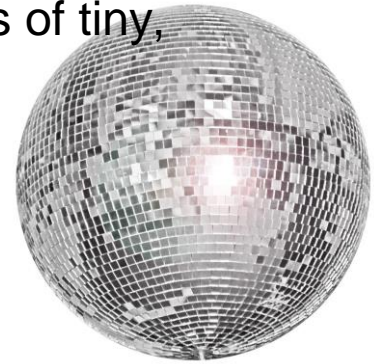
- When one same vertex is shared by more than one single triangle, its normal can be computed as the **average** of the normal vectors of the faces that share it.
- It is common to define a threshold angle or normal group to prevent artifacts, e.g., when drawing a cube corner:
  - One different normal should be specified for each face, although some vertices are shared among neighbor faces.
- $$v = (v_{12} + v_{23} + v_{34} + v_{41}) / |v_{12} + v_{23} + v_{34} + v_{41}|$$



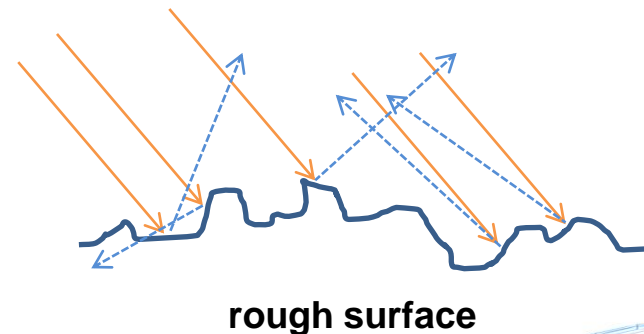


## The microfacet theory

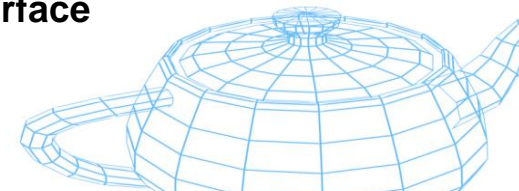
- At microscopic scale, any surface can be represented by a series of tiny, perfectly reflective mirrors called **microfacets**.
- The alignment of these tiny mirrors depends on the roughness of a surface, which might reflect light into specific directions or scatter it all around:



smooth surface

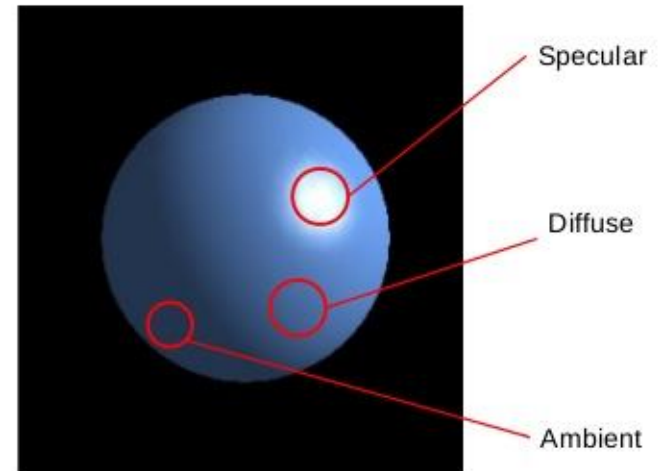


rough surface

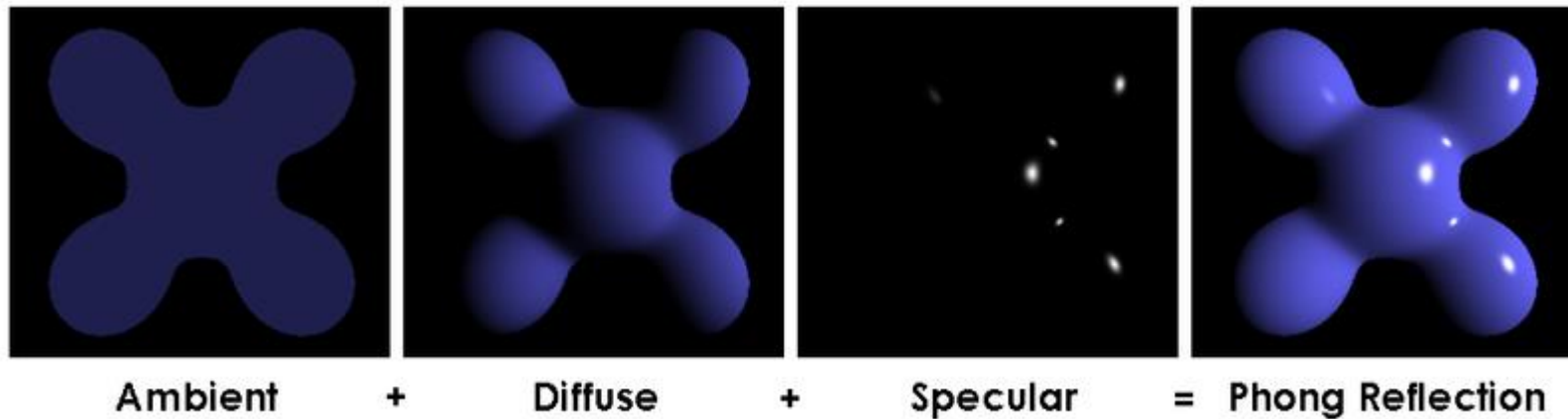


## Lighting model

- Empirical model:
  - Combines diffuse reflection of rough surfaces with specular reflection of shiny surfaces.
  - An additional ambient component simulates indirect light scattering.
- Simple and computationally-light model:
  - Cannot simulate all existing materials.
  - Direct illumination only:
    - No shadows, no global illumination.



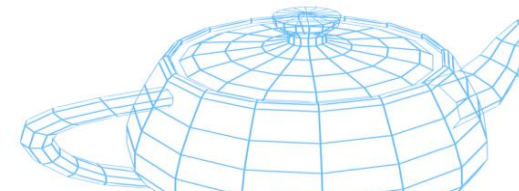
## Lighting model



- Phong's reflection model, PhD thesis, Utah University, 1973
  - OpenGL uses a slightly different variant called Blinn-Phong reflection model.
- Not to be confused with Phong shading.

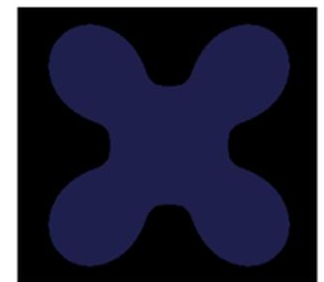
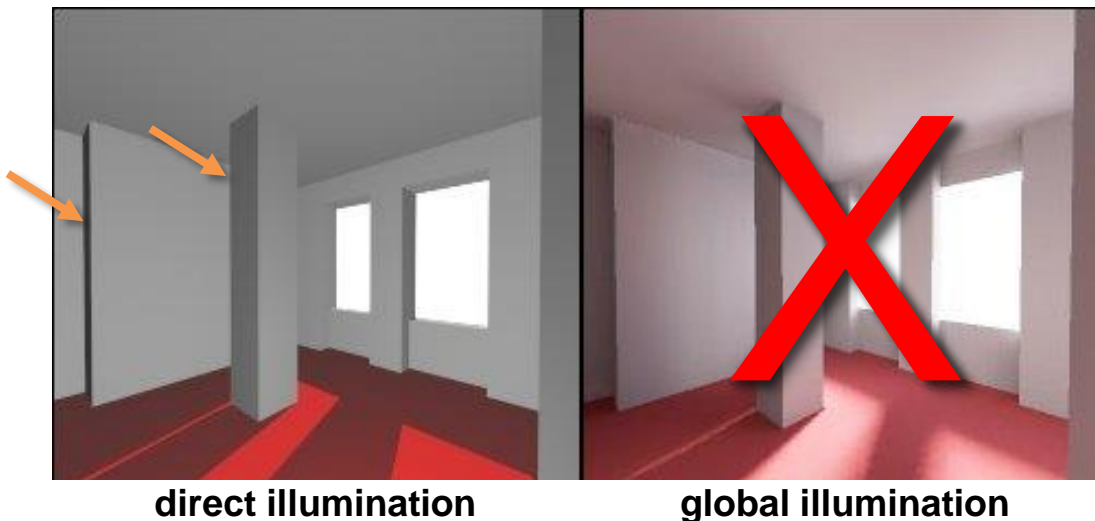


Bui Tuong Phong  
1942 - 1975

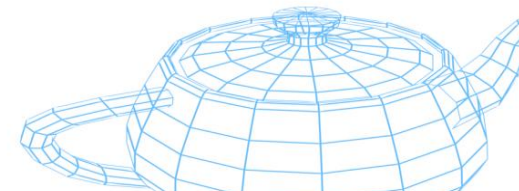


## Ambient term

- Simulates global illumination:
  - When you switch a light on, some rays bounce on the wall one or more times and reach areas that are not directly facing the light source.
  - It is impossible to tell the original direction from scattered light that bounced several times.
  - If you switch the light off, the indirect light disappears.
  - Without ambient light, back faces would be always dark.



Ambient

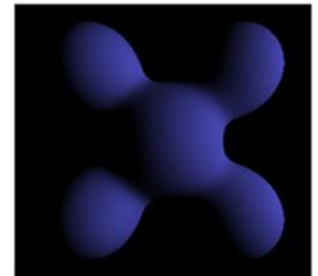
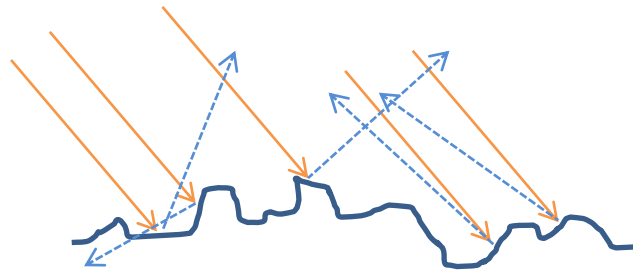


## Diffuse term

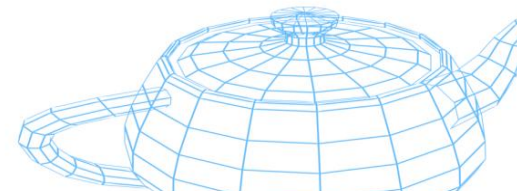
- Is the light that comes from one direction:
  - It is brighter if it comes squarely down on a surface than if it barely glances off the surface.
  - Once it hits a surface, it is scattered equally in all directions, so it appears equally bright, no matter where the eye is located.
  - Based on the Lambert cosine law.



Johann Heinrich Lambert  
1728 - 1777

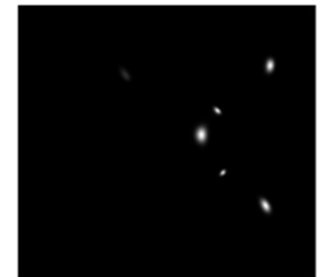
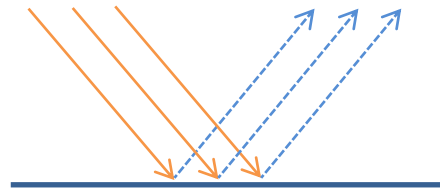
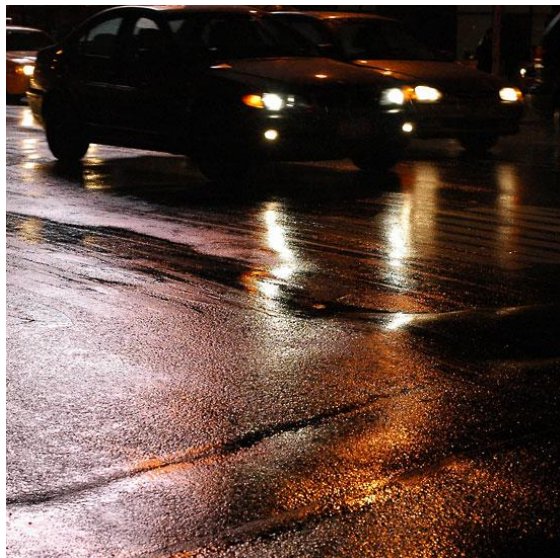


Diffuse

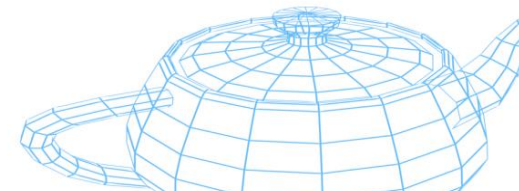


## Specular term

- Is the light that comes from a particular direction, and it tends to bounce off the surface in a preferred direction:
  - A well-collimated laser beam bouncing off a high-quality mirror produces almost 100 percent specular reflection.
  - Shiny metal or plastic has a high specular component, and chalk or carpet has almost none.
  - You can think of specular as shininess.

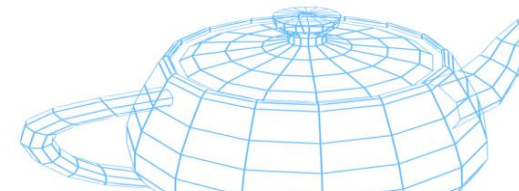


Specular



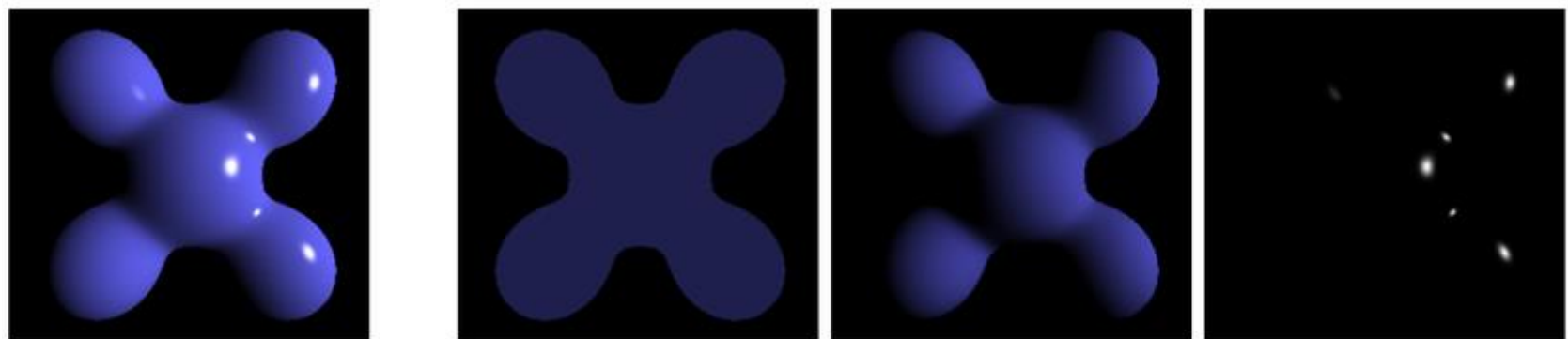
## Emissive term

- Additional term used to simulate light originating from an object:
  - In the OpenGL lighting model, the emissive color of a surface adds a specific, constant intensity to the object, but is unaffected by any light sources:
    - It works like `glColor*()`
  - The emissive color does not introduce any additional light into the overall scene.





## Lighting model (Phong)



Phong Reflection

Ambient

+

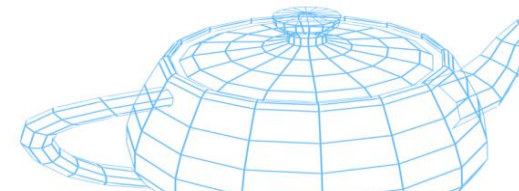
Diffuse

+

Specular

$$I_p = k_e + k_a i_a + \sum_{m \in \text{lights}} (k_d (\hat{L}_m \cdot \hat{N}) i_{m,d} + k_s (\hat{R}_m \cdot \hat{V})^\alpha i_{m,s})$$

Emissive

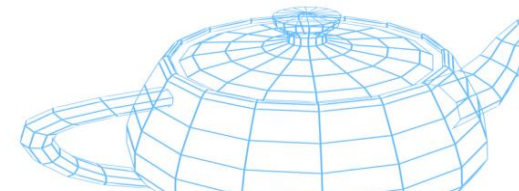




## Lighting model (Phong)

$$I_p = k_e + k_a i_a + \sum_{m \in \text{lights}} (k_d (\hat{L}_m \cdot \hat{N}) i_{m,d} + k_s (\hat{R}_m \cdot \hat{V})^\alpha i_{m,s})$$

- $I_p$  = surface point color
- $k_*$  = material properties
- $i_*$  = light properties
- $a, d, s, e$  = ambient, diffuse, specular, emissive (typically as RGB vectors)
- $N$  = surface normal
- $V$  = surface direction towards the viewer
- $L_m$  = surface direction towards the light
- $R_m$  = perfect reflection ray vector, defined as  $\hat{R}_m = 2(\hat{L}_m \cdot \hat{N})\hat{N} - \hat{L}_m$
- $\alpha$  = shininess constant (larger for mirror-like surfaces)

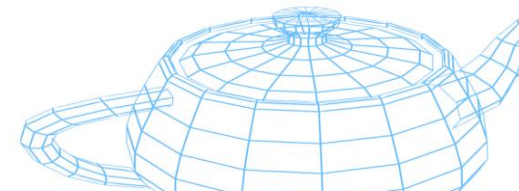
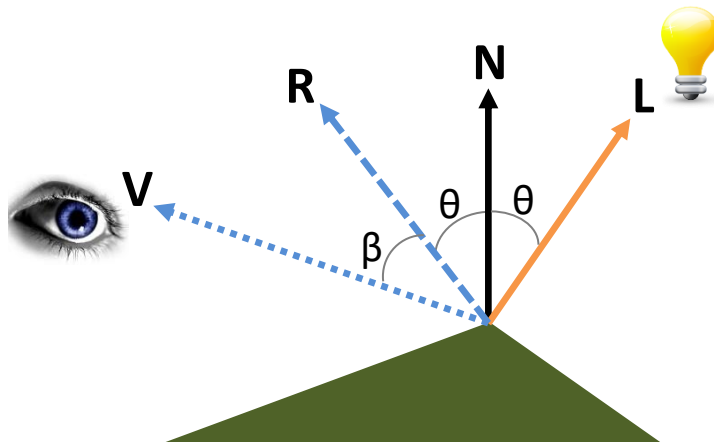


## Lighting model (Phong)

$$I_p = k_e + k_a i_a + \sum_{m \in \text{lights}} (k_d (\hat{L}_m \cdot \hat{N}) i_{m,d} + k_s (\hat{R}_m \cdot \hat{V})^\alpha i_{m,s})$$

$$\hat{R}_m = 2(\hat{L}_m \cdot \hat{N})\hat{N} - \hat{L}_m$$

- According to Phong's model, the specular component is proportional to the cosine of the angle  $\beta$  between the light reflection vector and the eye vector:
  - The specular effect is higher when R and V are closer.
  - The specular decay is determined by  $\alpha$  (higher values mean faster decay).





## Lighting model (Blinn-Phong)

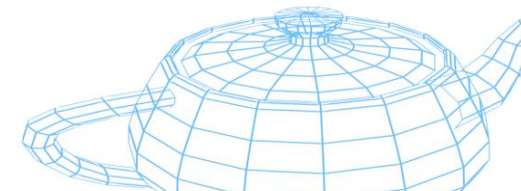
$$(Phong, 1973) \quad I_p = k_e + k_a i_a + \sum_{m \in \text{lights}} (k_d (\hat{L}_m \cdot \hat{N}) i_{m,d} + k_s (\hat{R}_m \cdot \hat{V})^\alpha i_{m,s})$$

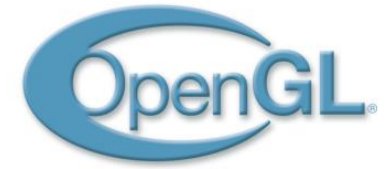
~~$$\hat{R}_m = 2(\hat{L}_m \cdot \hat{N})\hat{N} - \hat{L}_m$$~~

$$(Blinn, 1977) \quad I_p = k_e + k_a i_a + \sum_{m \in \text{lights}} (k_d (\hat{L}_m \cdot \hat{N}) i_{m,d} + k_s (\hat{N} \cdot \hat{H}_m)^\alpha i_{m,s})$$

$$\hat{H}_m = \frac{\hat{L}_m + \hat{V}}{\|\hat{L}_m + \hat{V}\|}$$

- $H$  = half-vector between the viewer and the light vector



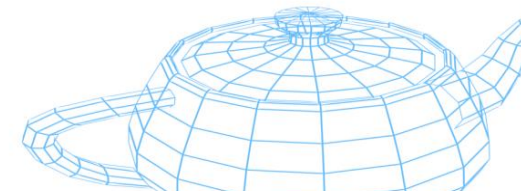
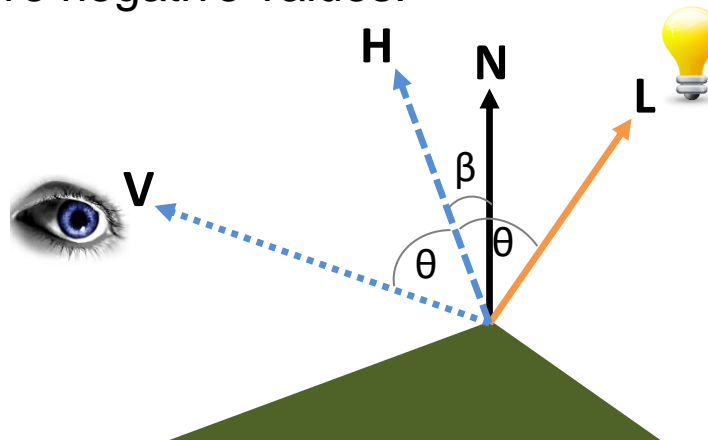


## Lighting model (Blinn-Phong)

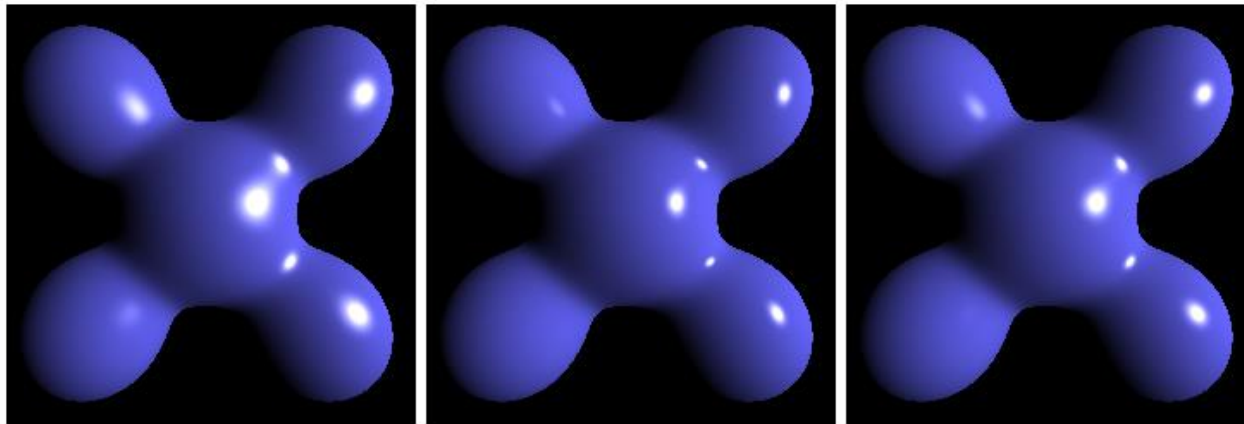
$$I_p = k_e + k_a i_a + \sum_{m \in \text{lights}} (k_d (\hat{L}_m \cdot \hat{N}) i_{m,d} + k_s (\hat{N} \cdot \hat{H}_m)^\alpha i_{m,s})$$

$$\hat{H}_m = \frac{\hat{L}_m + \hat{V}}{\|\hat{L}_m + \hat{V}\|}$$

- This variant is computationally slightly less expensive than the standard Phong's model.
- The half-angle H is always below 90°
  - With standard Phong, the angle  $\beta$  could be larger than 90° under some circumstances and give negative values.




## Lighting model



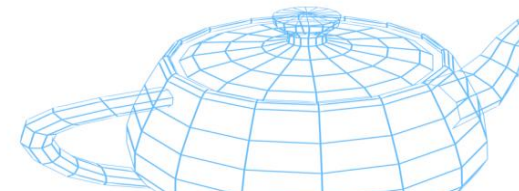
**Blinn-Phong**

**Phong**

**Blinn-Phong**  
(higher exponent)

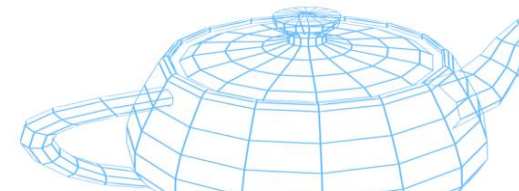
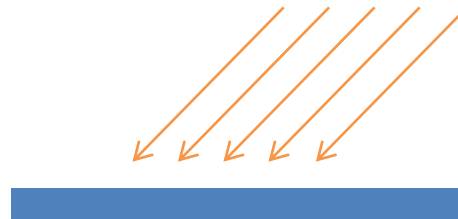
$$I_p = k_e + k_a i_a + \sum_{m \in \text{lights}} (k_d (\hat{L}_m \cdot \hat{N}) i_{m,d} + k_s (\hat{N} \cdot \hat{H}_m)^\alpha i_{m,s})$$


Similar models: just use a higher exponent (in Blinn-Phong) to make it looking more like Phong



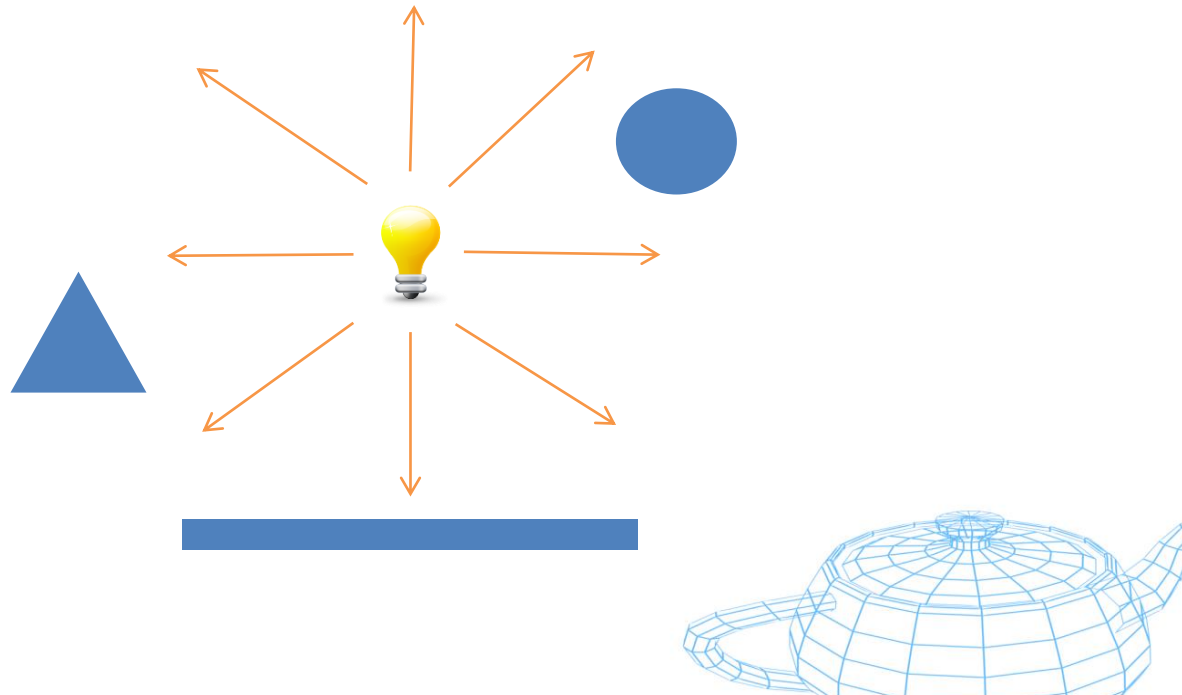
## Directional light

- Also called “infinite light”.
- Light is coming from an infinite distance, with all its rays perfectly parallel.
- E.g.: sun rays (on earth).
- Easiest and fastest light to simulate and compute:
  - ray vectors are all the same.



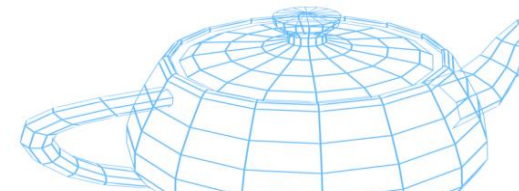
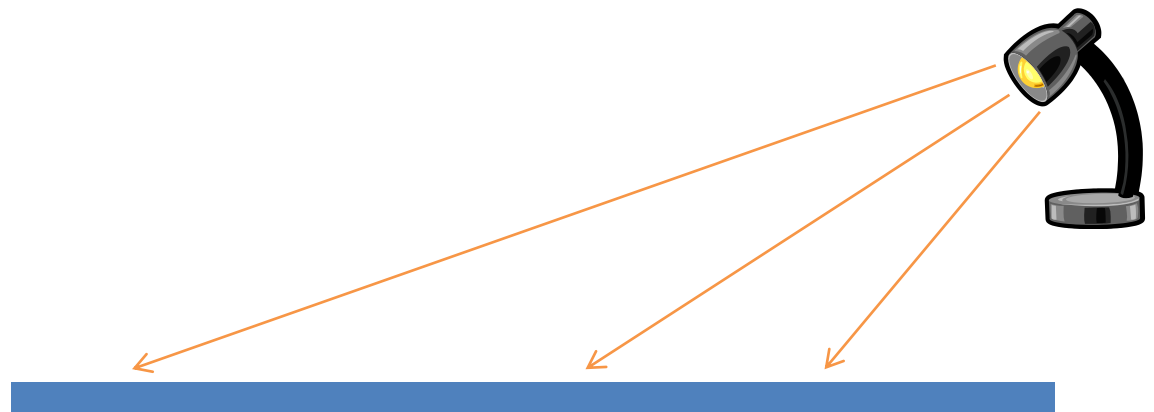
## Omnidirectional light

- Also called “point light”.
- Light irradiates equally in all directions, originating from a single point.
- E.g.: candle, emergency signal flare, sun rays (in space).



## Spot light

- Light source with a cone of effect, optionally fading its intensity when closer to the border.
- E.g.: torch, desk lamp, stage light, projector.





## Attenuation

$$I_f = I_p \frac{1}{k_c + k_l d + k_q d^2}$$

- $I_f$  = final illumination after attenuation
- $I_p$  = illumination computed through the lighting model
- $d$  = distance of the light source
- $k_c$  = constant factor
- $k_l$  = linear factor
- $k_q$  = quadratic factor

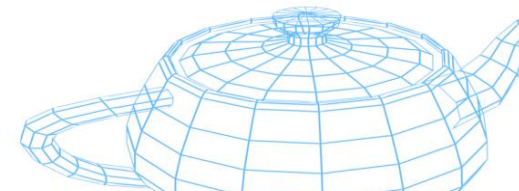




## Lighting model

- Lighting is deactivated by default and must be enabled using:
  - `glEnable(GL_LIGHTING);`
- Each light source must be activated independently, using:
  - `glEnable(GL_LIGHT#);`  
Where # is in the range 0 to some maximum number of supported lights minus one.
  - You can query the maximum number of supported lights via see `glGetIntegerv(GL_MAX_LIGHTS, &maxNrOfLights);`

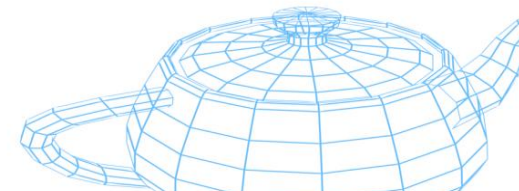
...I've never seen `GL_MAX_LIGHTS` returning more than 8 ☺

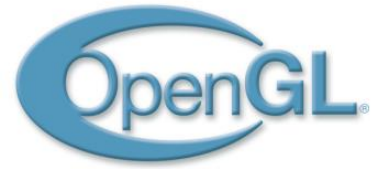




## Lighting model

- `GL_LIGHT0` default parameters are different from `GL_LIGHT1` to `GL_LIGHT_N`.
- Usually, only the first 3-4 lights are hardware accelerated:
  - Use multipass rendering when many light sources are necessary.
- Do not activate more than the minimum number of lights required for rendering your object:
  - Visually useless.
  - Potential saturation, unless you use high dynamic range.
  - Unnecessarily slow.





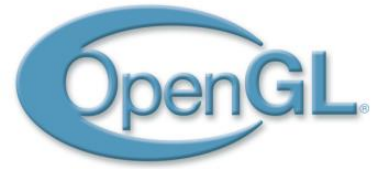
## Material properties

$$I_p = k_e + k_a i_a + \sum_{m \in \text{lights}} (k_d (\hat{L}_m \cdot \hat{N}) i_{m,d} + k_s (\hat{N} \cdot \hat{H}_m)^\alpha i_{m,s})$$

```
glm::vec4 emission(0.0f, 0.0f, 0.0f, 1.0f);  
glm::vec4 ambient(0.2f, 0.2f, 0.2f, 1.0f);  
glm::vec4 diffuse(0.8f, 0.8f, 0.8f, 1.0f);  
glm::vec4 specular(0.5f, 0.5f, 0.5f, 1.0f);  
int shininess = 128;
```

```
glMaterialfv(GL_FRONT_AND_BACK, GL_EMISSION, glm::value_ptr(emission));  
glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, glm::value_ptr(ambient));  
glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, glm::value_ptr(diffuse));  
glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, glm::value_ptr(specular));  
  
glMaterialf(GL_FRONT_AND_BACK, GL_SHININESS, shininess);
```



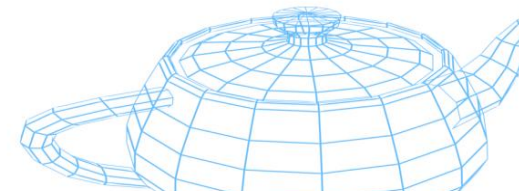


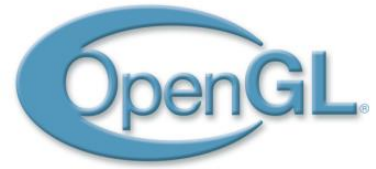
## Light properties

$$I_p = k_e + k_a i_a + \sum_{m \in \text{lights}} (k_d (\hat{L}_m \cdot \hat{N}) i_{m,d} + k_s (\hat{N} \cdot \hat{H}_m)^\alpha i_{m,s})$$

```
glm::vec4 ambient(1.0f, 1.0f, 1.0f, 1.0f);  
glm::vec4 diffuse(1.0f, 1.0f, 1.0f, 1.0f);  
glm::vec4 specular(1.0f, 1.0f, 1.0f, 1.0f);
```

```
glLightfv(GL_LIGHT#, GL_AMBIENT, glm::value_ptr(ambient));  
glLightfv(GL_LIGHT#, GL_DIFFUSE, glm::value_ptr(diffuse));  
glLightfv(GL_LIGHT#, GL_SPECULAR, glm::value_ptr(specular));
```





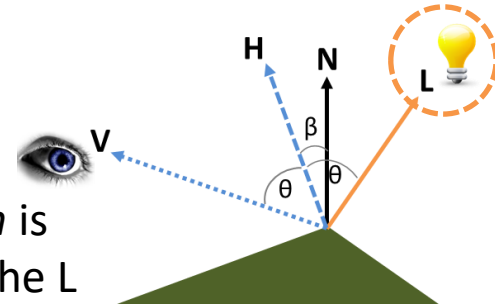
## Light properties

### Directional/infinite light settings

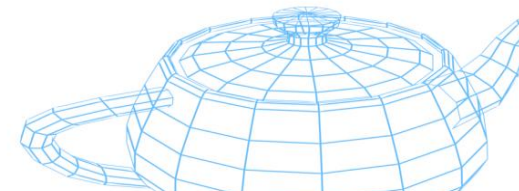
*w is zero!*

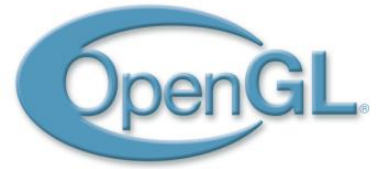
*position is  
used as the L  
vector!*

```
glm::vec4 position(x, y, z, 0.0f);  
glLightfv(GL_LIGHT#, GL_POSITION, glm::value_ptr(position));
```



- Since the position is specified in homogeneous coordinates, it is computed as  $x/w$ ,  $y/w$ , and  $z/w$ .
- When  $w$  approaches zero,  $x$ ,  $y$ , and  $z$  approach infinity. You can verify it by dividing  $x$ ,  $y$ , and  $z$  by a very small value.
- Diffuse and specular lighting calculations take the light's direction but not its actual position into account, and attenuation is disabled.





## Light properties

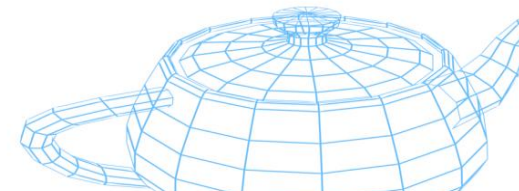
### Omnidirectional/point light settings

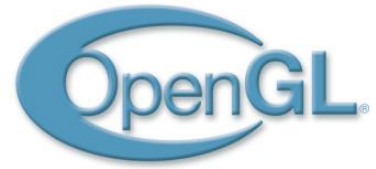
w is one!

```
glm::vec4 position(x, y, z, 1.0f);  
float cutoff = 180.0f; // special value
```

```
glLightfv(GL_LIGHT#, GL_POSITION, glm::value_ptr(position));  
glLightfv(GL_LIGHT#, GL_SPOT_CUTOFF, &cutoff);
```

- A special cutoff value of  $180^\circ$  is used to tell OpenGL that we want an omnidirectional light.





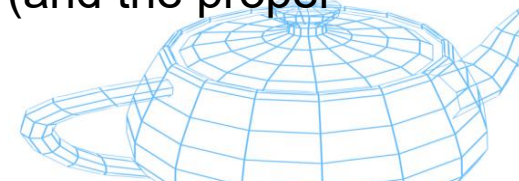
## Light properties

### Spot light settings

w is one!

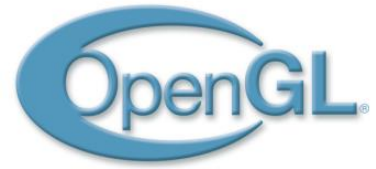
```
glm::vec4 position(x, y, z, 1.0f);  
float cutoff = 15.0f;    // Any angle between 0° and 90°  
glm::vec3 direction(dx, dy, dz);  
  
glLightfv(GL_LIGHT#, GL_POSITION, glm::value_ptr(position));  
glLightfv(GL_LIGHT#, GL_SPOT_CUTOFF, &cutoff);  
glLightfv(GL_LIGHT#, GL_SPOT_DIRECTION, glm::value_ptr(direction));
```

- Keep in mind that (any) light position and direction is multiplied by the current modelview matrix, like any other geometric primitive.
  - ...you can then specify lights both in object coordinates (and the proper transformation matrices) or in world coordinates.



**WARNING!**





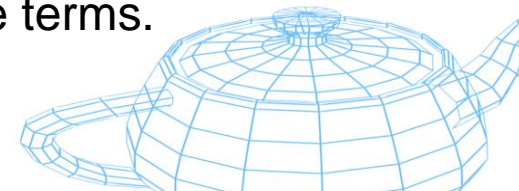
## Light properties

### Attenuation

$$I_f = I_p \frac{1}{k_c + k_l d + k_q d^2}$$

```
glLightf(GL_LIGHT#, GL_CONSTANT_ATTENUATION, value); // default 1  
glLightf(GL_LIGHT#, GL_LINEAR_ATTENUATION, value);    // default 0  
glLightf(GL_LIGHT#, GL_QUADRATIC_ATTENUATION, value); // default 0
```

- Attenuation only works for omnidirectional and spot lights.
- Attenuation does not affect the global ambient and emissive terms.





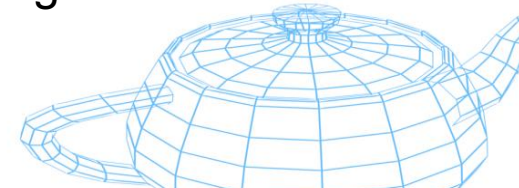
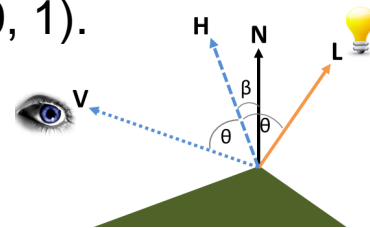
## Light properties

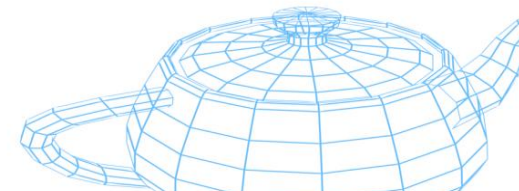
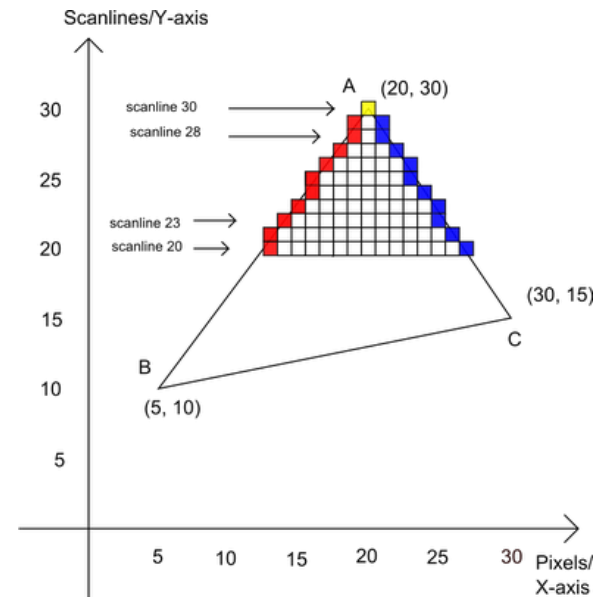
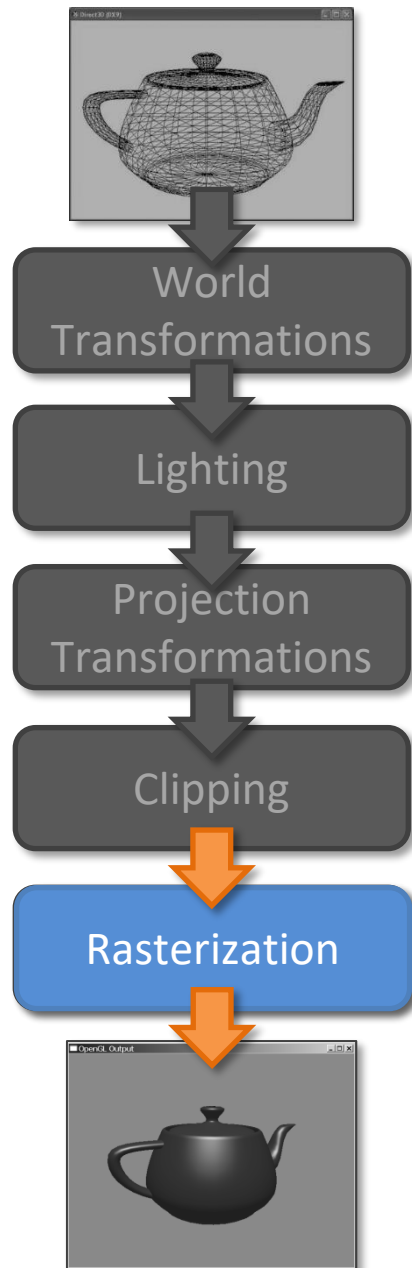
### Global settings

```
glm::vec4 gAmbient(0.2f, 0.2f, 0.2f, 1.0f);

glLightModelf(GL_LIGHT_MODEL_LOCAL_VIEWER, 1.0f); // default 0
glLightModelfv(GL_LIGHT_MODEL_AMBIENT, glm::value_ptr(gAmbient));
```

- Activate the local viewer position for a more accurate computation of specular highlights (recommended). Otherwise, the V vector is always (0, 0, 1).
- The global ambient term is multiplied by each light ambient term:
  - Default is [0.2 0.2 0.2 1.0]: that's the reason why even bright ambient terms look so dark.





## Shading

- Shading is decoupled from the lighting model.
- Three main shading options:



### Flat shading

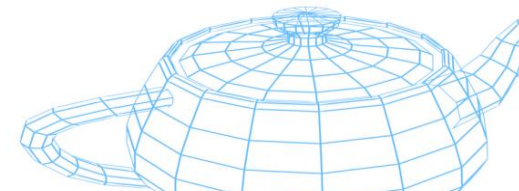
```
glShadeModel(GL_FLAT);
```

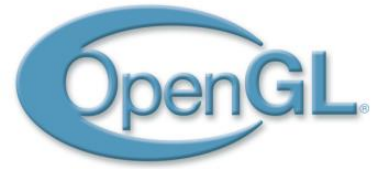
### Gouraud shading

```
glShadeModel(GL_SMOOTH);
```

### Phong shading

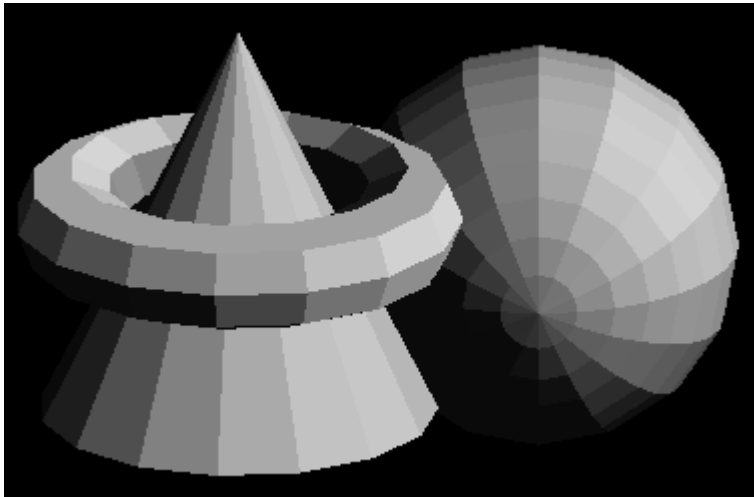
*(use shaders)*



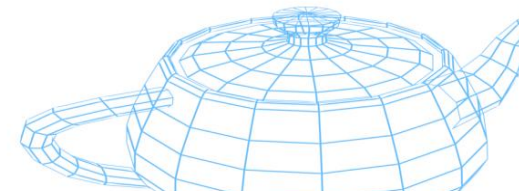


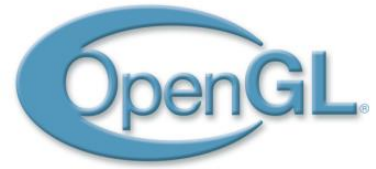
## Flat shading

- Also called “constant shading”.
- One single color is used for drawing all the pixels of the primitive.
- Very fast, poor quality:
  - Activated using `glShadeModel (GL_FLAT)` ;
- Last vertex color is used for the whole surface.



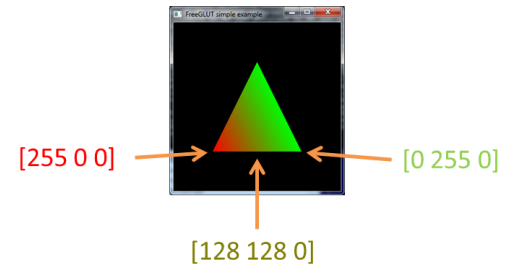
[source: John T. Bell]



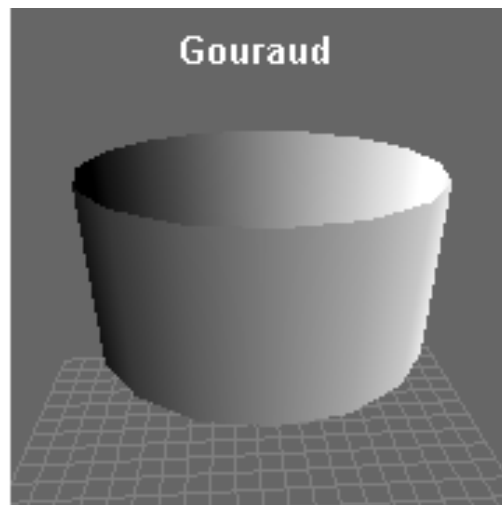
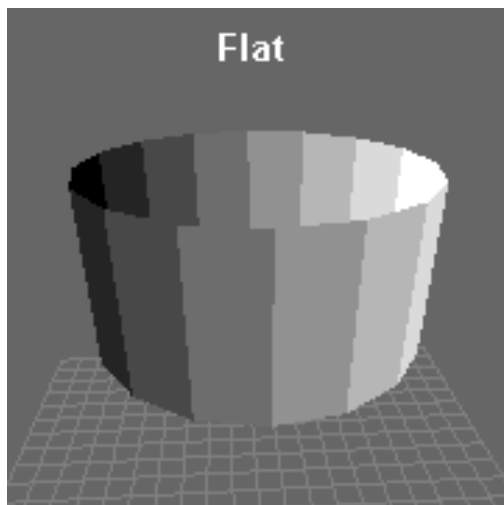


## Gouraud shading (1974)

- Vertex colors are linearly interpolated over the surface.
- Still very fast and with better quality than flat shading:
  - When normal vectors are properly set, edges disappear.
- Default shading used by the OpenGL fixed pipeline:
  - Activated using `glShadeModel(GL_SMOOTH)`.

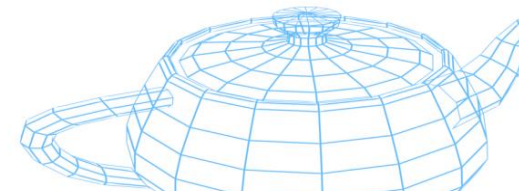
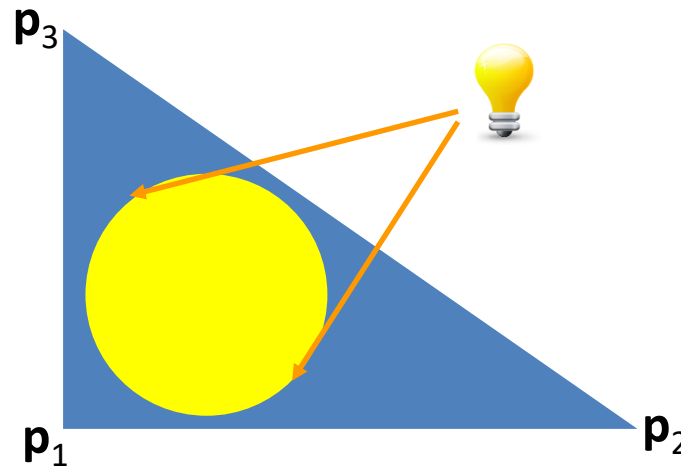


Henri Gouraud  
1944



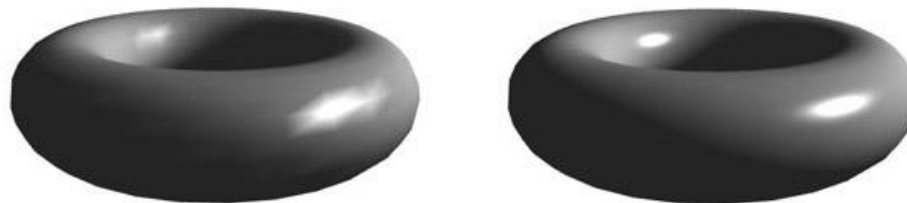
## Missed highlights

- Since the lighting model is computed at each vertex, a large triangle may completely miss one light's contribution:
  - Use a higher level of tessellation.
  - Use shaders to compute the lighting model on per-pixel basis.

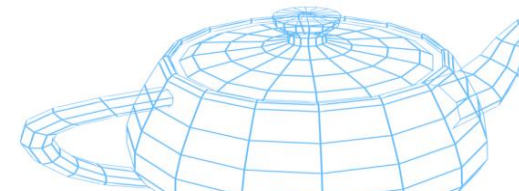


## Phong shading

- Also referred to as “per-pixel lighting”.
- The normal is interpolated over the surface and the lighting equation is computed locally within each fragment:
  - Requires much more computational power:
    - Normalization occurring at each fragment.
- When high tessellation is used, Gouraud shading looks like Phong shading:
  - One triangle per pixel  $\rightarrow$  Gouraud = Phong
- Phong shading requires programmable shaders (OpenGL 2.0+).



[source: [www.lighthouse3d.com](http://www.lighthouse3d.com)]







# Tutorial

Lighting example