1) Vertex load and transformation

2) Rasterisation

3) Fragment coloring
   - materials & lights
   - textures

4) Transparency and depth computation

From Colors to
Materials, Lights and Shading

Lighting

- Phong lighting model (Bui Tuong Phong)
  - simplified model of real world light behavior
  - color is not continuous, but RGB ratio
  - limited light sources (8 guaranteed)
    - nowadays usually unlimited (more lights → slower)
  - pipelined computation – one primitive at a time

    → no shadows

    → no reflections

    → no refraction

- Do not mix with Phong shading!

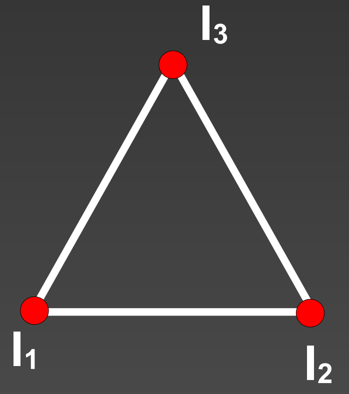
Phong lighting model

$$I_m = c_a \cdot i_a + c_d \cdot i_d \cdot (N \cdot L) + c_s \cdot i_s \cdot (V \cdot R)^n$$

$$I_{tot} = \sum_{m=0}^{srcs} I_m = c_a \cdot \sum_{m=0}^{srcs} i_{a,m} + \sum_{m=0}^{srcs} \left(c_d \cdot i_{d,m} \cdot (N \cdot L_m) + c_s \cdot i_{s,m} \cdot (V \cdot R_m)^n \right)$$

- Compute intensity for each point, source, RGB
- Total sum of all light sources
  - optimisation: source is too far → skip
- Depends on normal
  - MUST be normalised
  - vertex attribute, load from file or compute

```
glm::vec3 i1,i2,i3;
glm::normalize(glm::cross(i2-i1,i3-i1))
```



Math note: why normalised vectors?

$$I_m = c_a \cdot i_a + c_d \cdot i_d \cdot \cos\left(angle_{normal\_lightsource}\right) + c_s \cdot i_s \cdot \cos\left(angle_{viewer\_reflection}\right)^n$$

Speed optimisation: replace transcendental with dot product

$$I_m = c_a \cdot i_a + c_d \cdot i_d \cdot \left(N \cdot L\right) + c_s \cdot i_s \cdot \left(V \cdot R\right)^n$$

Equality: $\cos(\theta) = \vec{a} \cdot \vec{b}$ when $|\vec{a}| = 1, |\vec{b}| = 1$

Let: $\vec{\boldsymbol{a}} = (x_1, y_1) = (a\cos\alpha, a\sin\alpha), \vec{\boldsymbol{b}} = (x_2, y_2) = (b\cos\beta, b\sin\beta), a = |\vec{\boldsymbol{a}}|, b = |\vec{\boldsymbol{b}}|$

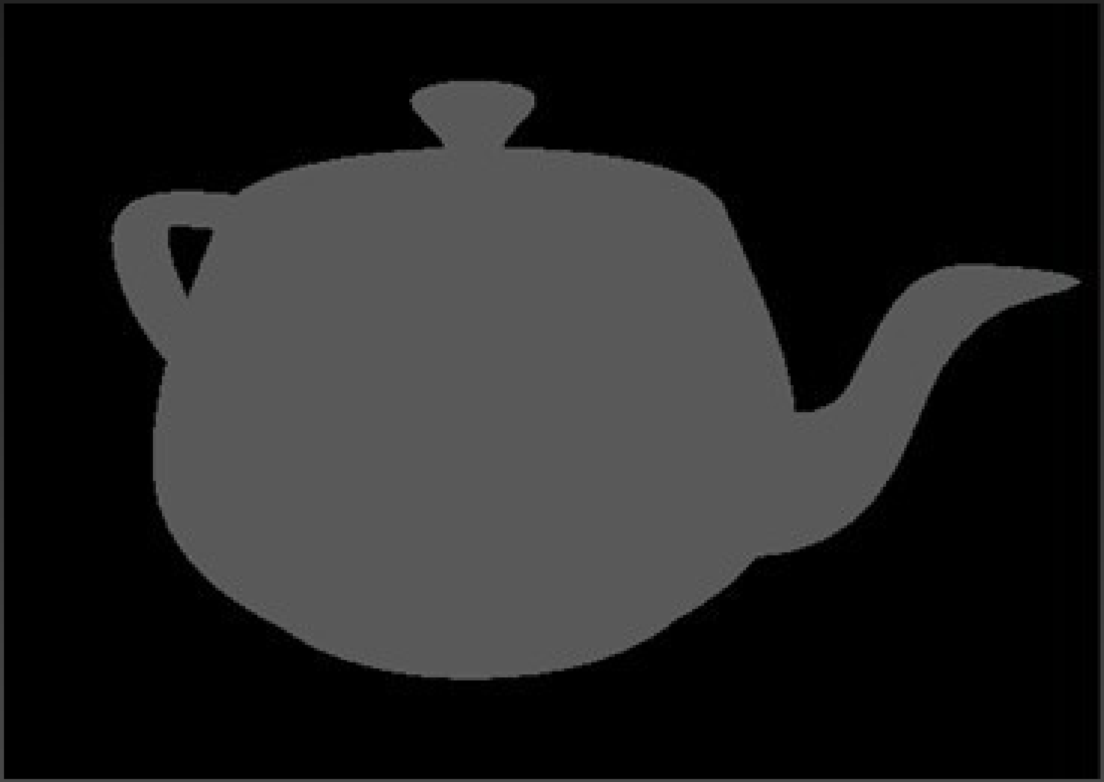
Then: $\theta = |\beta - \alpha|$

$$\begin{aligned}
\vec{\boldsymbol{a}} \cdot \vec{\boldsymbol{b}} &= x_1 x_2 + y_1 y_2 \\
&= ab\left(\cos\alpha\cos\beta + \sin\alpha\sin\beta\right) \\
&= ab\cos(\beta - \alpha) \\
&= ab\cos\theta \\
&= \cos\theta
\end{aligned}$$

Ambient component

$$I = c_a \cdot i_a + c_d \cdot i_d \cdot (N \cdot L) + c_s \cdot i_s \cdot (V \cdot R)^n$$

- c_a: material constant – reflectiveness for ambient light
- i_a: intensity of ambient component of light

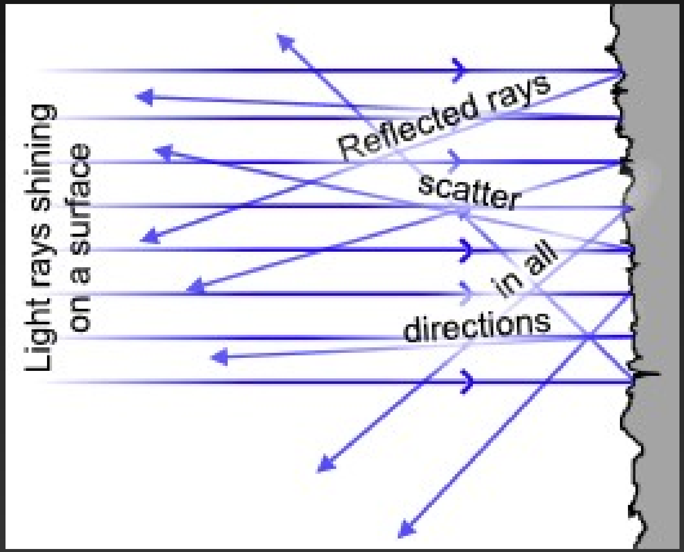


Phong lighting model

- three (four) independent components
  - ambient – scattered (omnidirectional) light, has no source or direction, from object scattered to all directions
  - diffuse – comes from single direction, from object scattered to all directions → depends on light source position only, not the viewer
  - specular – comes from single source, angle of incidence is the same as reflection (+ small scatter) → depends both on light source and viewer position
  - (radiation) – object radiates its own light, can be seen in absence of other light sources. It does not add light source to the scene, intensively radiating object does NOT light other objects!

Diffuse component

- Assumed ideal diffuse reflection
  - reflection is evenly distributed in all directions
- White wall paint, chalk, …



Diffuse component

$$I = c_a \cdot i_a + c_d \cdot i_d \cdot (N \cdot L) + c_s \cdot i_s \cdot (V \cdot R)^n$$

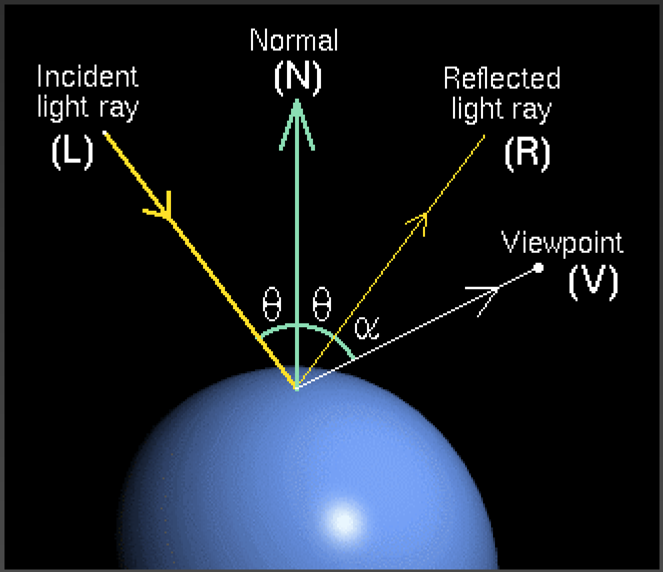
- c_d: material constant – reflectiveness for diffuse light
- i_d: intensity of diffuse component of light
- L – vector from point on object (vertex or rasterised point) towards light source
- N – normal vector

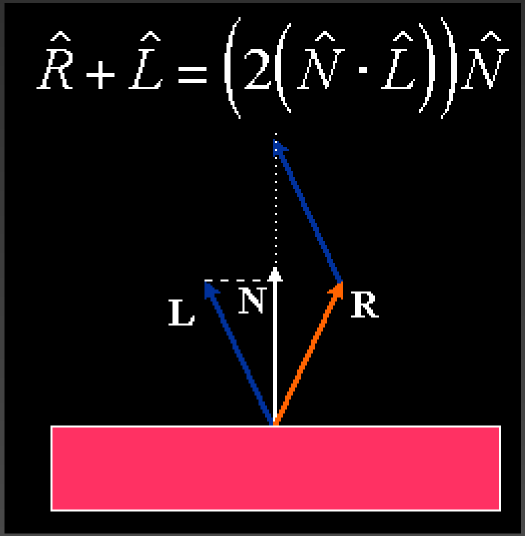
Phong lighting model

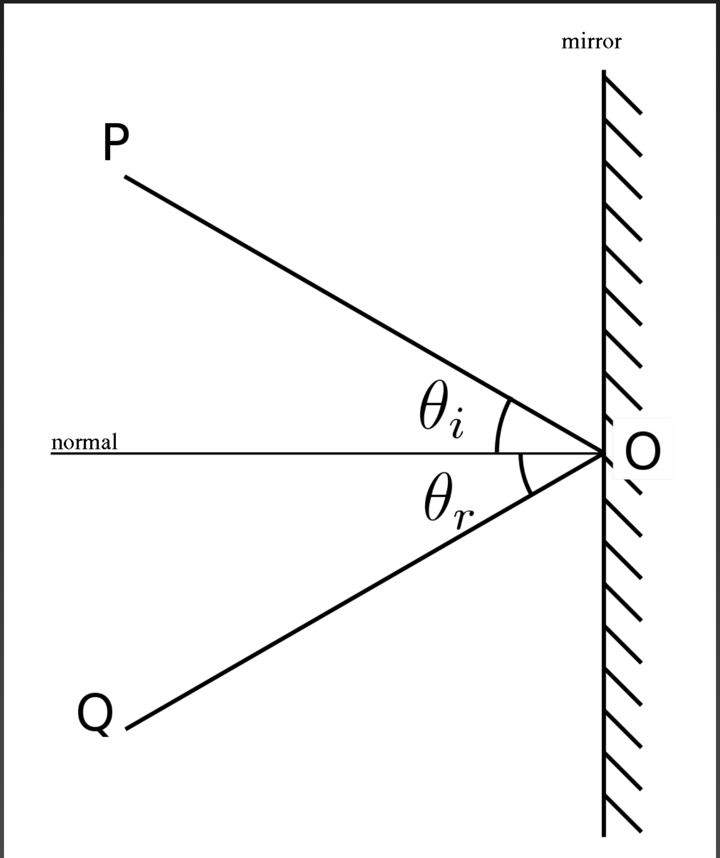
- three (four) independent components
  - ambient – scattered (omnidirectional) light, has no source or direction, from object scattered to all directions
  - diffuse – comes from single direction, from object scattered to all directions → depends on light source position only, not the viewer
  - specular – comes from single source, angle of incidence is the same as reflection (+ small scatter) → depends both on light source and viewer position
  - (radiation) – object radiates its own light, can be seen in absence of other light sources. It does not add light source to the scene, intensively radiating object does NOT light other objects!

Specular component

- Theoretically perfect mirror
- Angle on incidence and reflection
- Metal plate, water, glass, …



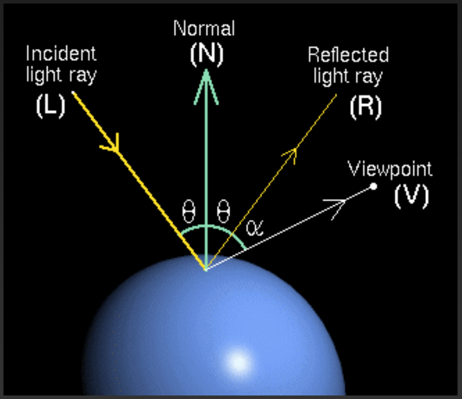


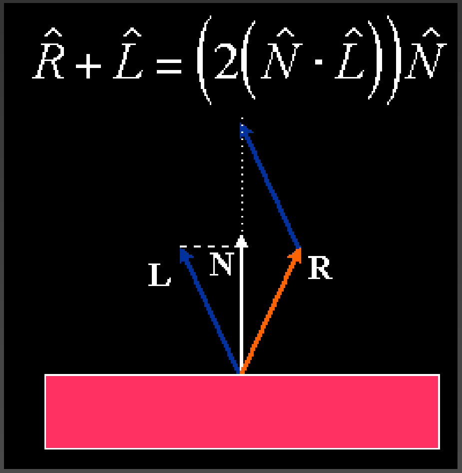


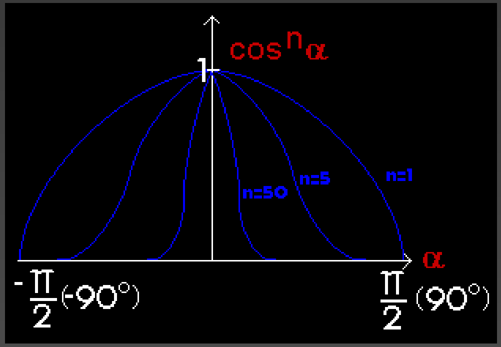
Specular component

$$I = c_a \cdot i_a + c_d \cdot i_d \cdot (N \cdot L) + c_s \cdot i_s \cdot (V \cdot R)^n$$

- c_s: material constant – reflectiveness for specular light
- i_s: intensity of specular component of light
- R – vector of perfect reflection
- V – vector towards viewer
- n – „shininess“, material constant (bigger = more intensive reflection with smaller diameter)
  - usually 0.0f to 128.0f (higher value → more intensive reflection with smaller diameter)

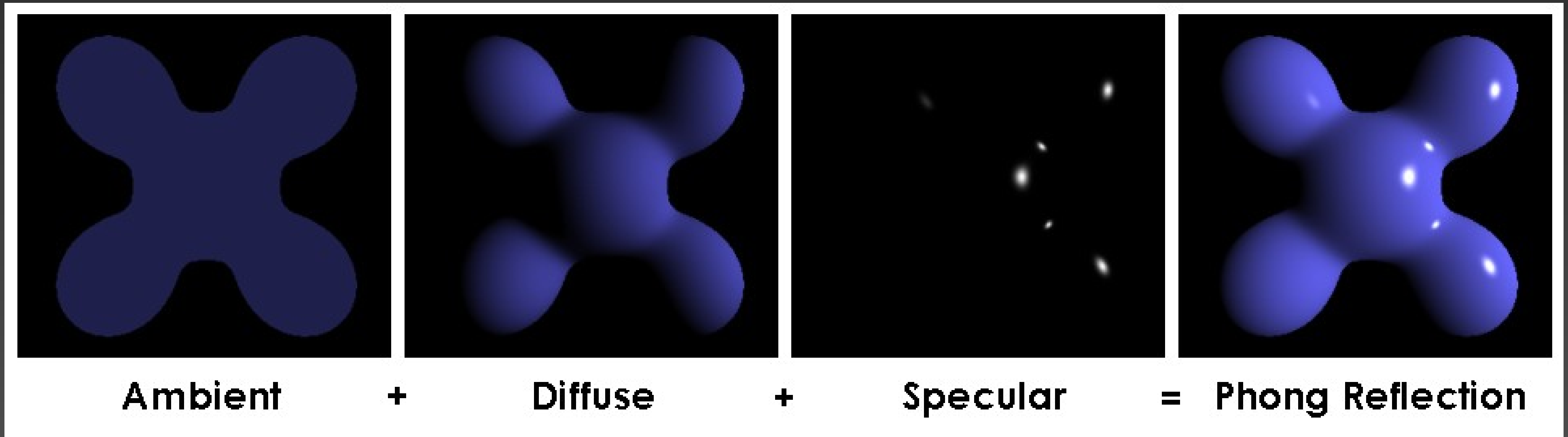






Specular component

Phong Light Model Examples



Material color

- Final color value depends on color of material and color of light sources
  - white light and red material
  - green light and red material
- Different color can be set for ambient, diffuse, specular component of material
  - ambient and diffuse usually same value
  - specular usually white (grey) – reflection has color of light source, just less intensive
- Intensity in range of 0.0f-1.0f, allows direct multiplication
  - lights: radiated color (LR,LG,LB)
  - material: reflected color (MR,MG,MB)
  - result: (LR*MR, LG*MG, LB*MB)

```
glm::vec3 light_rgb; glm::vec4 material_rgba;
glm::vec4 out_color = material_rgba * glm::vec4(light_rgb,1.0f);
```

Final color in vertex

- Phong model (for each of R,G,B and each light source separately)

$$I_m = c_a \cdot i_a + c_d \cdot i_d \cdot (N \cdot L) + c_s \cdot i_s \cdot (V \cdot R)^n$$

\+ radiation

$$I_{tot} = I_r + \sum_{m=0}^{srcs} I_m$$

Shading

- Filling of line or polygon by
  - single color – constant (flat) shading
    - attributes (e.g. colors) set by "provoking vertex" – last (closing) vertex of primitive
      FS: flat in vec4 myrgba;
  - interpolated color – Gouraud shading
    - linear interpolation of colors from vertices
    - color in vertices computed by Phong model
  - interpolated normal – Phong shading

Gouraud and Phong shading

- Gouraud
  - use Phong lighting model to compute color in vertices
  - inside polygon (FS) – linear interpolation of colors
  - simple HW, lower quality

- Phong (per-fragment lighting)
  - nothing computed for vertices, just pass data to fragments
  - linear interpolation of normal for each fragment in polygon
  - for each fragment compute color by complete Phong lighting model
  - higher quality (especially specular component)
  - GPU intensive, usually simplified

Per-vertex point light: Vertex shader

```
#version 430 core

// Vertex attributes
layout (location = 0) in vec4 aPosition;
layout (location = 1) in vec3 aNormal;

// Matrices
uniform mat4 m_m, v_m, p_m;

// Light and material properties
uniform vec3 light_position;
uniform vec3 ambient_intensity, diffuse_intensity, specular_intensity;
uniform vec3 ambient_material, diffuse_material, specular_material;
uniform float specular_shinines;

// Outputs to the fragment shader
out VS_OUT
{
    vec3 color;
} vs_out;

void main(void)
{
    // Create Model-View matrix
    mat4 mv_m = v_m * m_m;
    // Calculate view-space coordinate - in P point we are computing the color
    vec4 P = mv_m * aPosition;

    // Calculate normal in view space
    vec3 N = mat3(mv_m) * aNormal;
    // Calculate view-space light vector
    vec3 L = light_position - P.xyz;
    // Calculate view vector (negative of the view-space position)
    vec3 V = -P.xyz;

    // Normalize all three vectors
    N = normalize(N);
    L = normalize(L);
    V = normalize(V);
    // Calculate R by reflecting -L around the plane defined by N
    vec3 R = reflect(-L, N);

    // Calculate the ambient, diffuse and specular contributions
    vec3 ambient = ambient_material * ambient_intensity;
    vec3 diffuse = max(dot(N, L), 0.0) * diffuse_material * diffuse_intensity;
    vec3 specular = pow(max(dot(R, V), 0.0), specular_shinines) * specular_material * specular_intensity;

    // Send the color output to the fragment shader
    vs_out.color = ambient + diffuse + specular;

    // Calculate the clip-space position of each vertex
    gl_Position = p_m * P;
}
```

$$I_m = c_a \cdot i_a + c_d \cdot i_d \cdot (N \cdot L) + c_s \cdot i_s \cdot (V \cdot R)^n$$

Per-vertex point light: Frag. shader

```
#version 430 core
out vec4 color;

// Input from vertex shader
in VS_OUT
{
    vec3 color;
} fs_in;

void main(void)
{
    color = vec4(fs_in.color, 1.0);
}
```

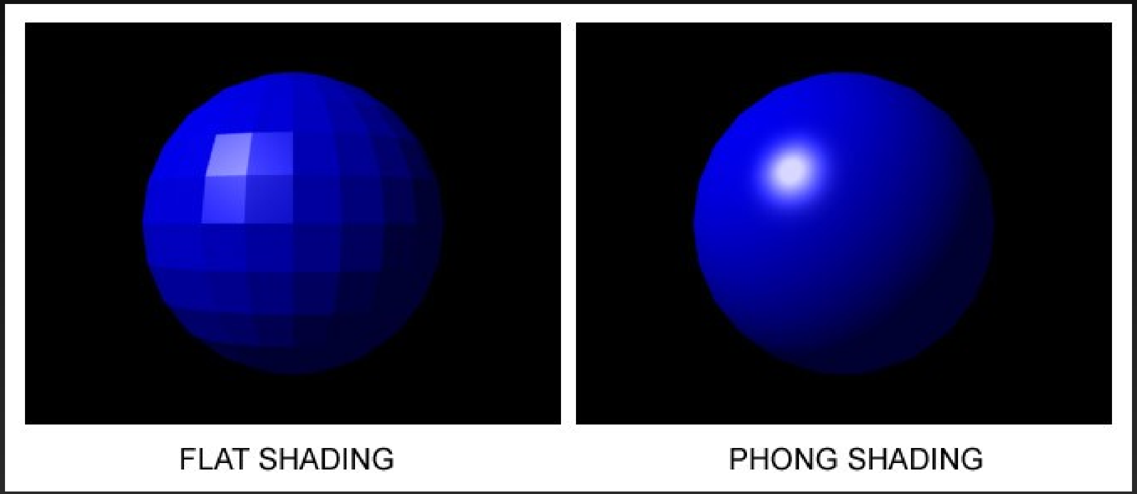
Per-fragment point light: Vert. shader

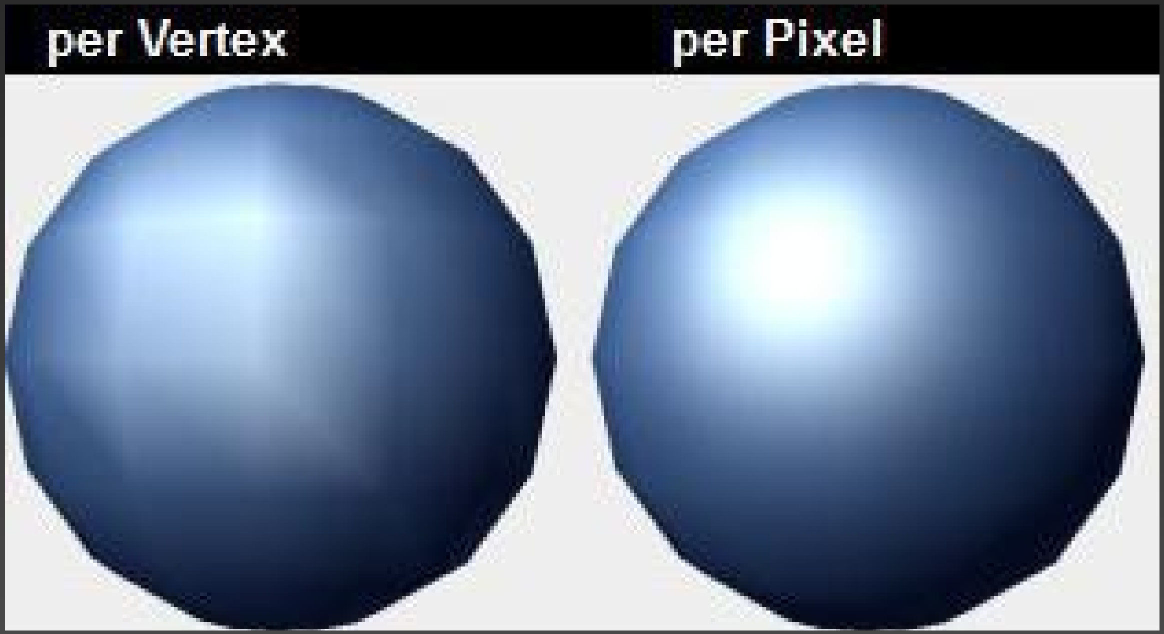
```
#version 430 core

// Vertex attributes
layout (location = 0) in vec4 aPosition;
layout (location = 1) in vec3 aNormal;

// Matrices
uniform mat4 m_m, v_m, p_m;

// Light properties
uniform vec3 light_position;

// Outputs to the fragment shader
out VS_OUT
{
    vec3 N;
    vec3 L;
    vec3 V;
} vs_out;

void main(void)
{
    // Create Model-View matrix
    mat4 mv_m = v_m * m_m;
    // Calculate view-space coordinate - in P point we are computing the color
    vec4 P = mv_m * aPosition;

    // Calculate normal in view space
    vs_out.N = mat3(mv_m) * aNormal;
     // Calculate view-space light vector
    vs_out.L = light_position - P.xyz;
    // Calculate view vector (negative of the view-space position)
    vs_out.V = -P.xyz;

    // Calculate the clip-space position of each vertex
    gl_Position = p_m * P;
}
```

Per-fragment point light: Frag. shader

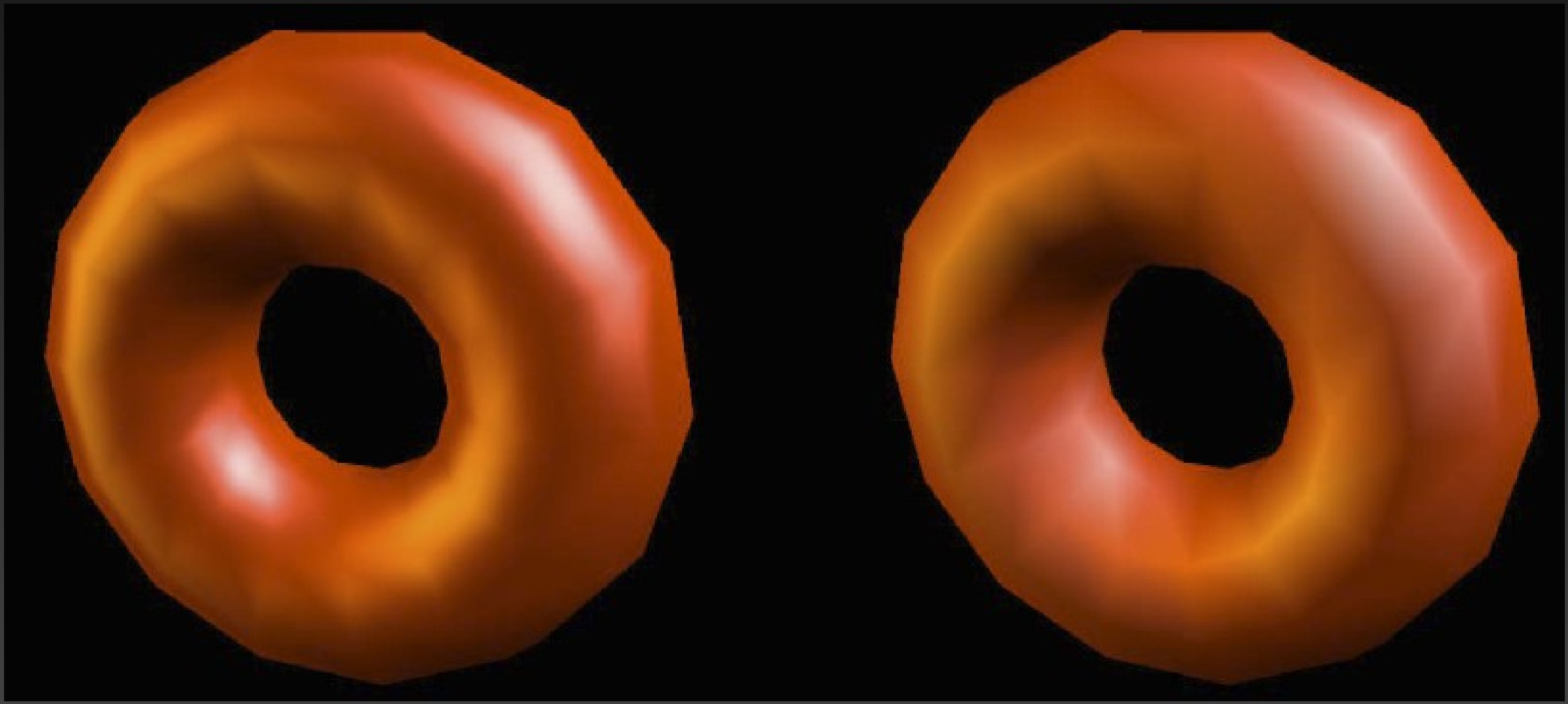
```
#version 430 core
out vec4 color;

// Material properties
uniform vec3 ambient_material, diffuse_material, specular_material;
uniform float specular_shinines;

// Input from vertex shader
in VS_OUT
{
    vec3 N;
    vec3 L;
    vec3 V;
} fs_in;

void main(void)
{
    // Normalize the incoming N, L and V vectors
    vec3 N = normalize(fs_in.N);
    vec3 L = normalize(fs_in.L);
    vec3 V = normalize(fs_in.V);

    // Calculate R by reflecting -L around the plane defined by N
    vec3 R = reflect(-L, N);

    // Calculate the ambient, diffuse and specular contributions
    vec3 ambient = ambient_material * vec3(1.0);
    vec3 diffuse = max(dot(N, L), 0.0) * diffuse_material;
    vec3 specular = pow(max(dot(R, V), 0.0), specular_shinines) * specular_material;

    color = vec4(ambient + diffuse + specular, 1.0);
}
```

Flat vs. interpolated shading
Per-fragment vs. Gouraud





Per-fragment vs. Gouraud



Lighting HOWTO

- Set normals for vertices
  - normalised vectors – length = 1.0f
- Light sources – set properties, position, etc.
- Lighting model – create shaders, set uniforms
- Materials – define material constants
  - ambient, diffuse, specular, shininess
  - usually per object – uniforms, not vertex attributes

- Lighting computation takes time
  - partition to static and dynamic lights
  - static lights and static parts of scene can be baked together
    - light + dark corridor textures → light corridor textures
    - choose light/dark texture – no lights → fast

Normalisation

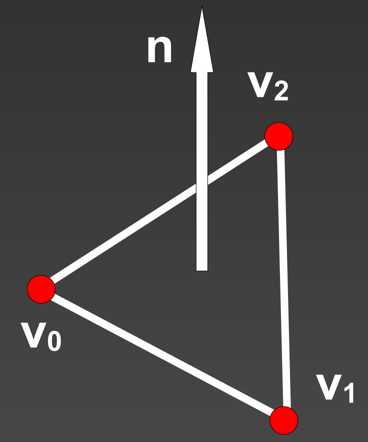
- Normals are scaled with transformations glm::scale() !!!
  - i.e. avoid dynamic scaling
  - try to precompute normals
- Normal vectors must be normalised
  - shaders
    - divide by length L ()

      vec3 nn = normalize(n);

$$\vec{n} = (v_2 - v_0) \times (v_1 - v_0)$$

$$L = \sqrt{(n_x^2 + n_y^2 + n_z^2)}$$

$$\vec{n}_n = \frac{\vec{n}}{L}$$



Create and set light source

- CoreP
  - Create uniforms
    - bools to control model
    - int to set light count
    - arrays of light properties
      - Vec3 for position, direction, diffuse color, specular color,…
      - Scalar for shininess, …
  - Create shaders with Phong light model, that uses defined parameters

Controlling light model

- Uniform bool controls shader

```
uniform bool use_lighting;
if (use_lighting)
    {… light model …}
    else { constant color }
```

Light types and properties

- Each light has all components
  - ambient, diffuse, specular
  - different values of RGBA and other parameters (A is usually ignored)
- Use w to determine light type
- Directional light source
  - in infinity – [x, y, z, 0.0]
    → position vector
    VS: vs_out.L = vec3(direction)
  - parallel rays = sun
- Point light source or SpotLight
  - inside scene – [x, y, z, 1.0] – position point
  - VS: vs_out.L = light_pos - P.xyz

- (array of) uniform parameters passed to shaders

```
// Loop through enabled lights
for (int i = 0; i < lightsNumber; i++) {
    if (lightSource[i].position.w == 0.0)
        DirectionalLight(i, normalize(n), amb, diff,
                         spec);
    else if (lightSource[i].spotCutoff == 180.0)
        PointLight(i, eye, Position, normalize(n),
                   amb, diff, spec);
    else
        SpotLight(i, eye, Position, normalize(n),
                  amb, diff, spec);
}
```

Visual improvement, light types

- Point source + Spotlight
  - attenuation by distance (constant, linear, quadratic)

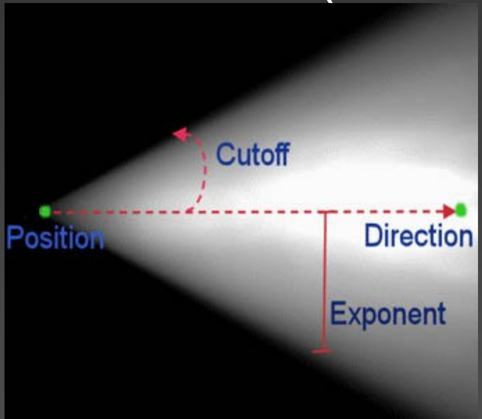
$$\left(\frac{1}{(k_c + k_l \cdot d + k_q \cdot d^2)}\right)$$

```
d = length(L) ; //vector to light source
dist_attenuation = 1.0 / (constantAttenuation +
                    linearAttenuation * d +
                    quadraticAttenuation * d * d);

color = ambient + dist_attenuation * (diffuse + specular)
```

Visual improvement, light types

- Spotlight
  - position as point source + direction + cone
  - light direction uniform glm::vec3 spotDirection;
  - light cone (angle) uniform float cosCutoff; (= glm::cos(cutoff_angle))
  - light distribution in cone uniform float spotExponent;

```
float spotEffect = dot(normalize(spotDirection), -L);
if (spotEffect > cosCutoff)
    full_attenuation =  dist_atten * pow(spotEffect, spotExponent);

color = ambient + full_attenuation * (diffuse + specular)
```

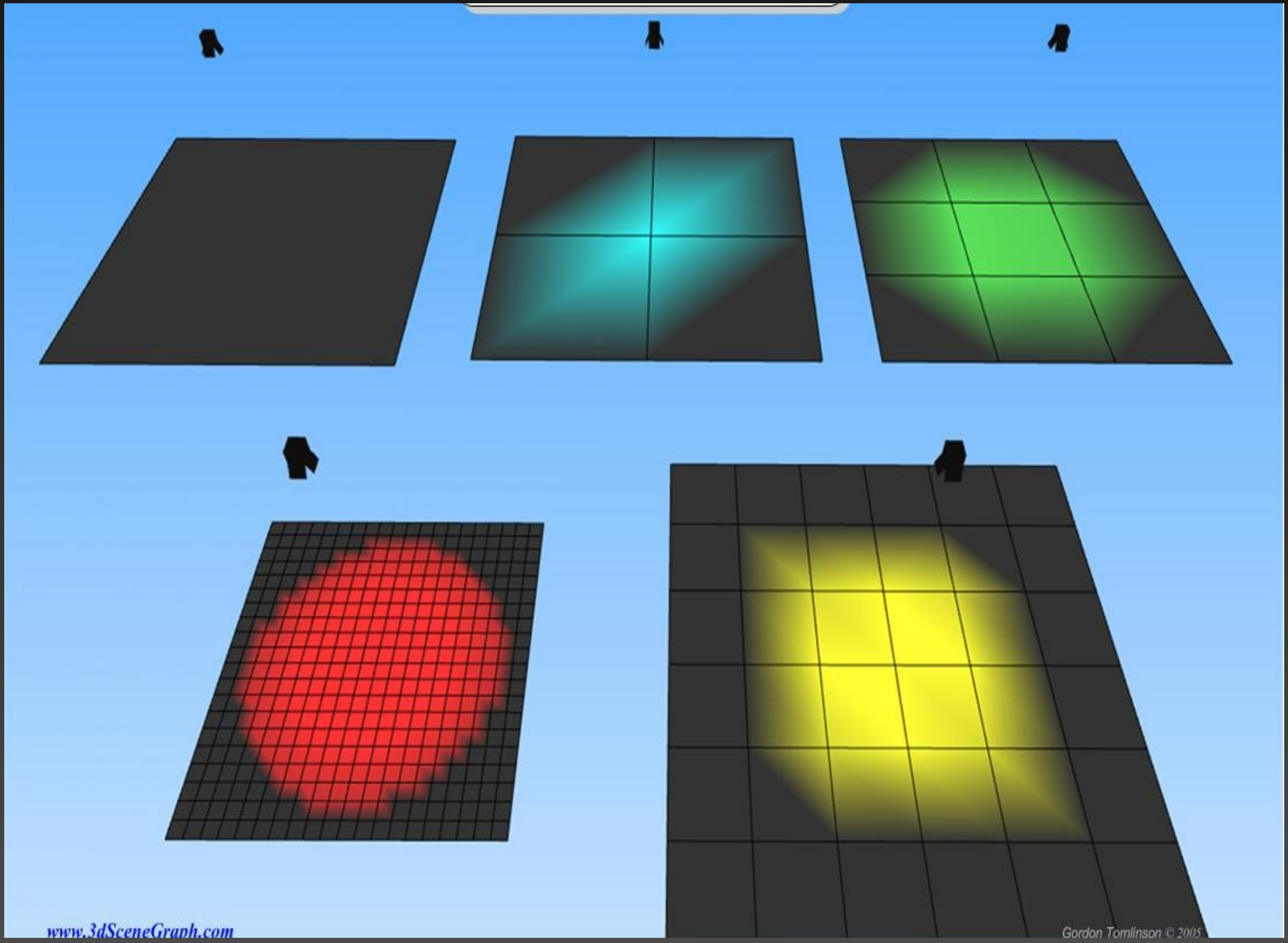


Per-fragment components

```
uniform sampler2D tex0; //diffuse texture
uniform sampler2D tex1; //specular map

ambient = …
diffuse = …
specular = …

color = texture(tex0, texcoord0) * (diffuse  + ambient) +
          + texture(tex1,texcoord0) * specular;
```

Lighting problem

- Per-vertex lighting is innacurate
  - need per-fragment by shaders



Materials

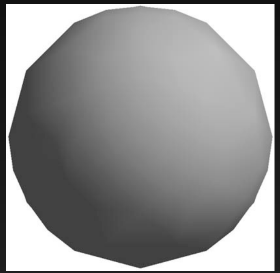
- define reaction (albedo) to ambient, diffuse and specular component of light source
- self-light = radiation, independent on any light source
  - set as uniform for object

color = radiation + ambient + full_attenuation * (diffuse + specular)

$$I_{tot} = I_r + \sum_{m=0}^{srcs} I_m$$

Shading of connected polygons

- Implicit normal for polygon
  - compute normalize(cross(v_2-v_1,v_3-v_2)) and set for all vertices of triangle
  - Phong model calculates same color in all vertices → flat look



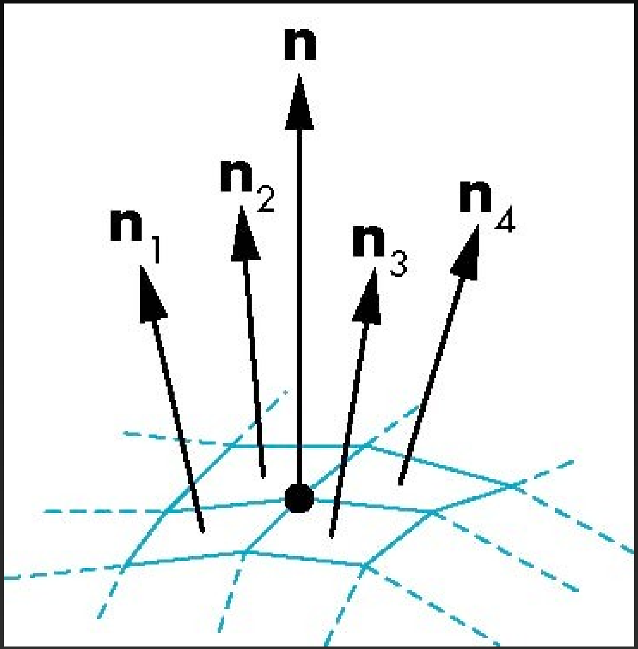
- Independent normal for each vertex
  - best: load from file 😎
  - manual computation: normal in shared vertex is average of implicit normals of connected polygons → continuous interpolation → smooth look
  - Be careful: not mathematically correct! (plane has only one normal by definition)

Averaging Normals

- Average of normals in vertex
  - precompute on model create (load)

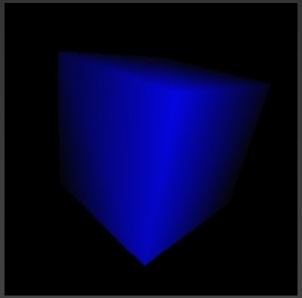
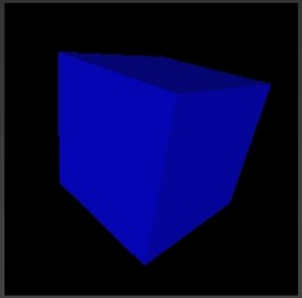
```
for each primitive
    for each vertex of primitive
        for each vertex of model
            if vertex_shared { n=avg(n1,n2n,…) }
```

$$n = (n1+n2+n3+n4)/\ |n1+n2+n3+n4|$$



- Phong model calculates different color in all polygon vertices …
  - … but connected vertices of different polygons have same color
    - smooth connection without visible edges
    - geometry is the same → rough contour stays
- we may need sharp edge
  - user defined angle limit for averaging

    if (normal_difference < 70°) { do_avg; }

Online Example

http://www.cs.toronto.edu/~jacobson/phong-demo/