1) Vertex load and transformation

2) Rasterisation

3) Fragment coloring
   - textures
   - materials & lights

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 wavelength, but RGB ratio
  - limited light sources (more lights → slower)
    - use max 10 (20) light sources
  - pipelined computation – one primitive at a time

    → no shadows

    → no reflections

    → no refraction

- Do not mix with Phong shading!

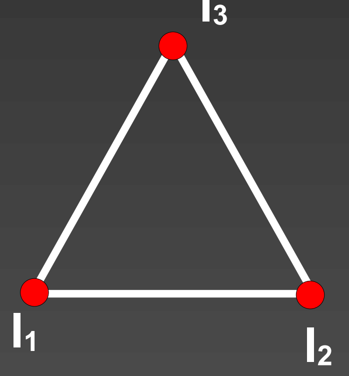
Phong Lighting Model

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

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

- 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?

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

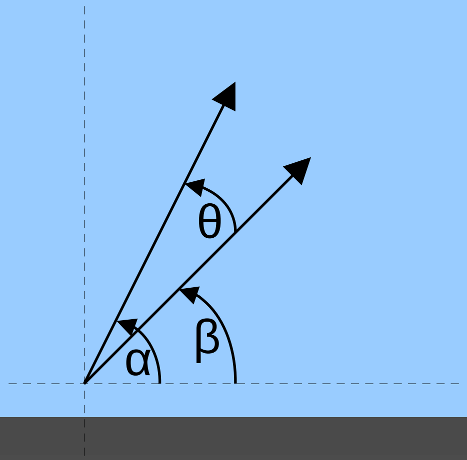
TARGET: Speed optimisation: replace transcendental with dot product

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

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



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

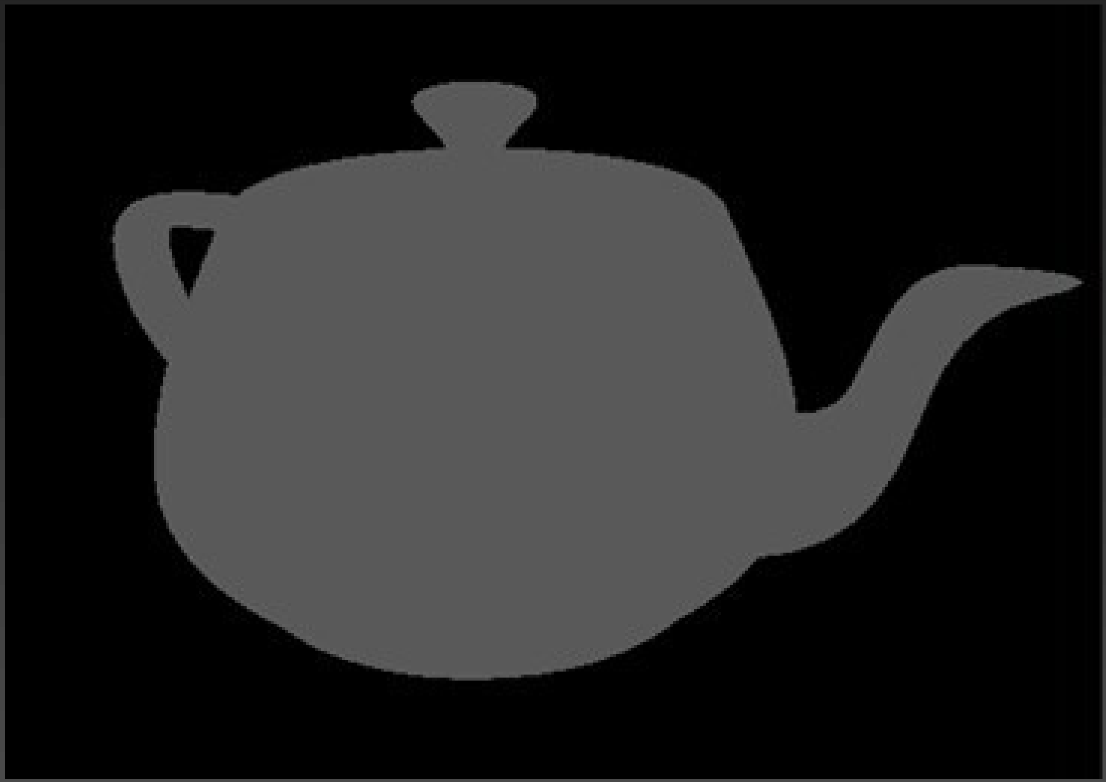
Phong Lighting Model

- 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!

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

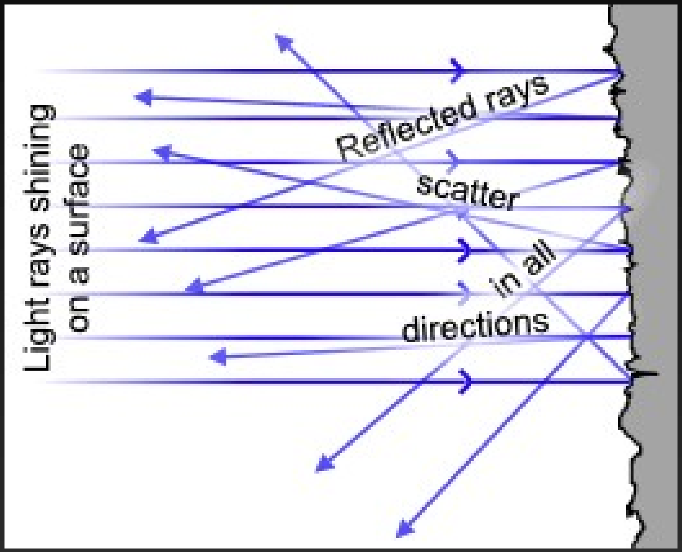


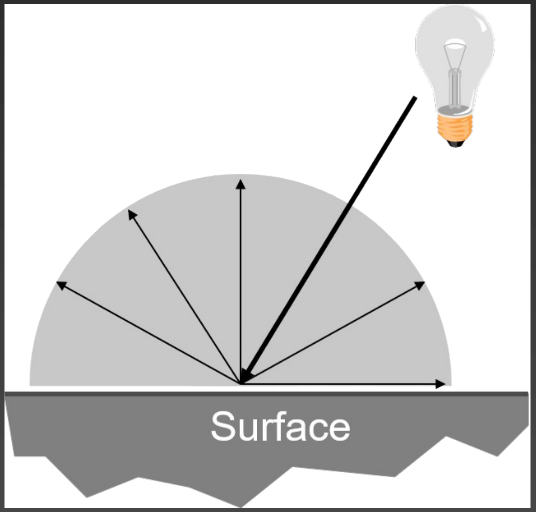
Phong Lighting Model

- 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

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

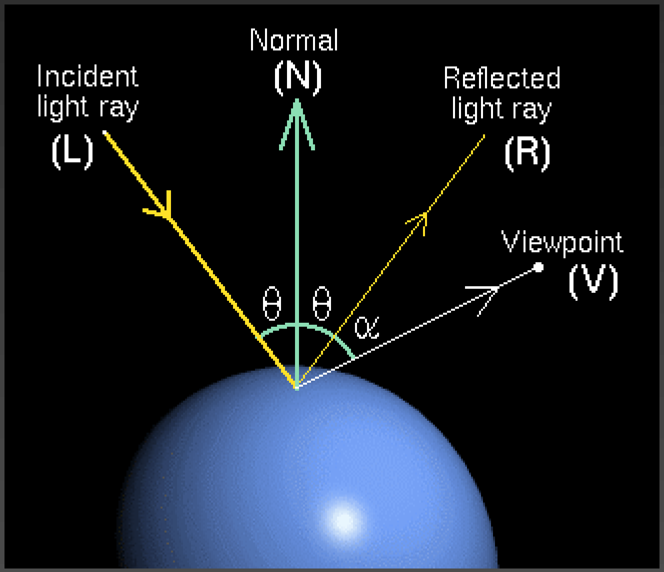
- c_d: material constant – reflectiveness for diffuse light
- i_d: intensity of diffuse component of light
- L – vector from point on object (vertex or rasterised point) towards light source
- N – normal vector

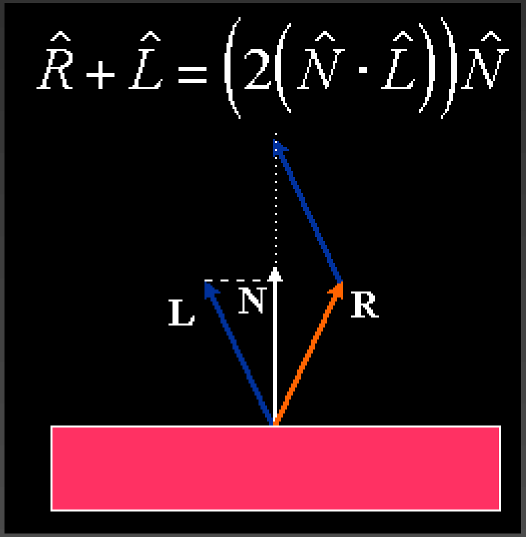
Phong Lighting Model

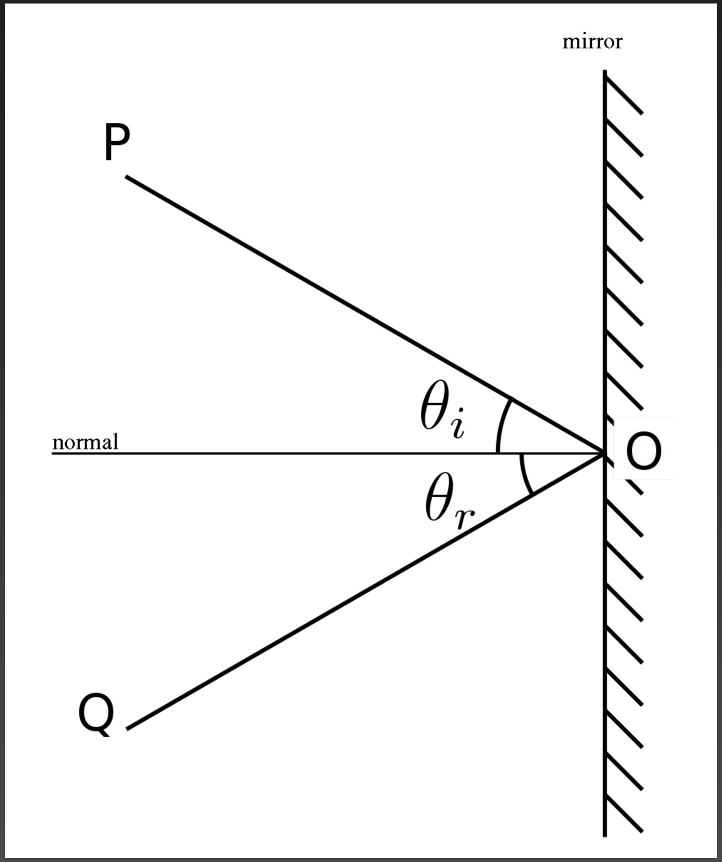
- three (four) independent components
  - ambient – scattered (omnidirectional) light, has no source or direction, from object scattered to all directions
  - diffuse – comes from single direction, from object scattered to all directions → depends on light source position only, not the viewer
  - specular – 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, …



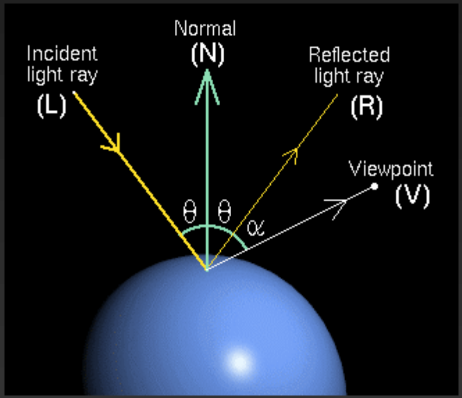


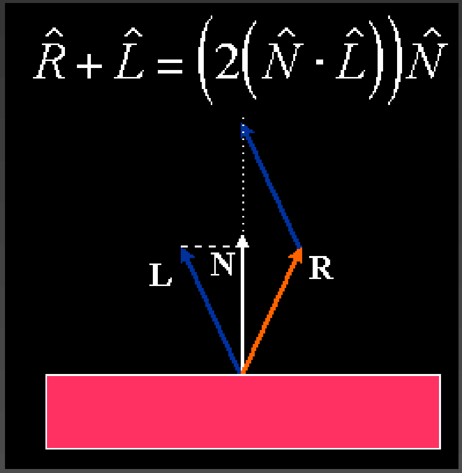


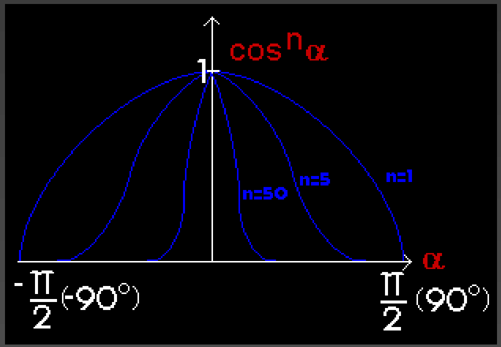
Specular Component

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

- 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)

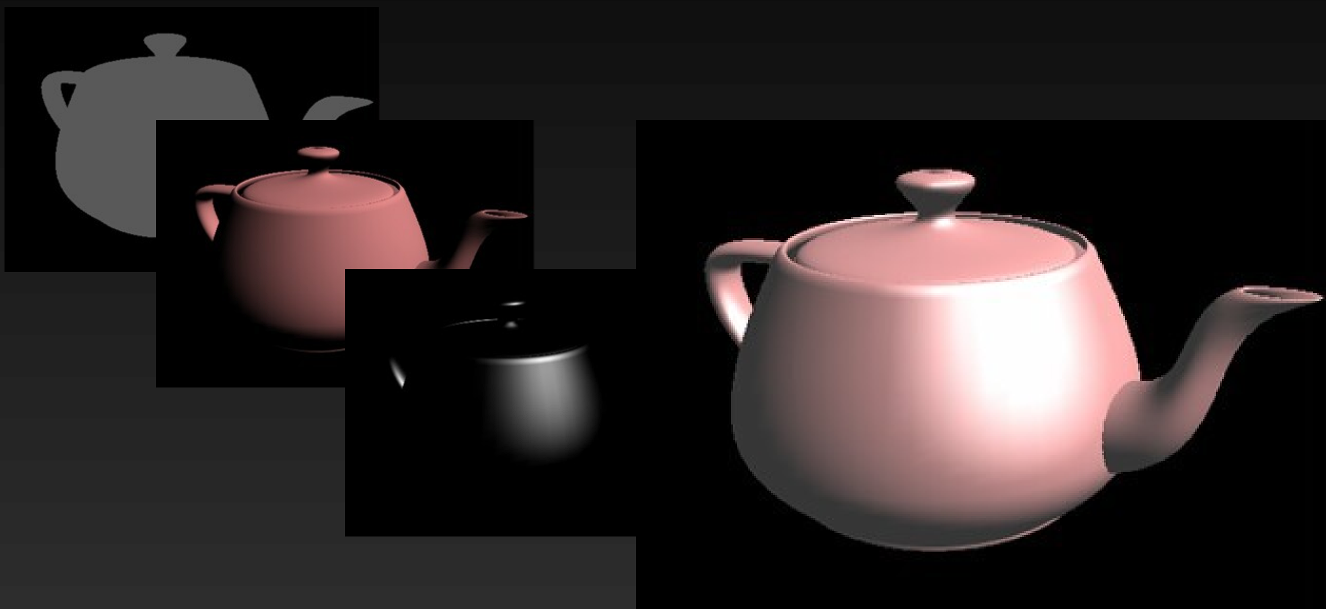


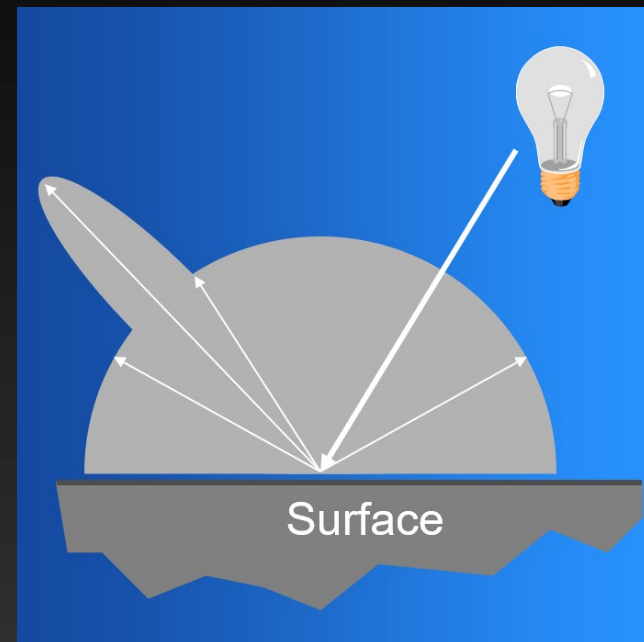


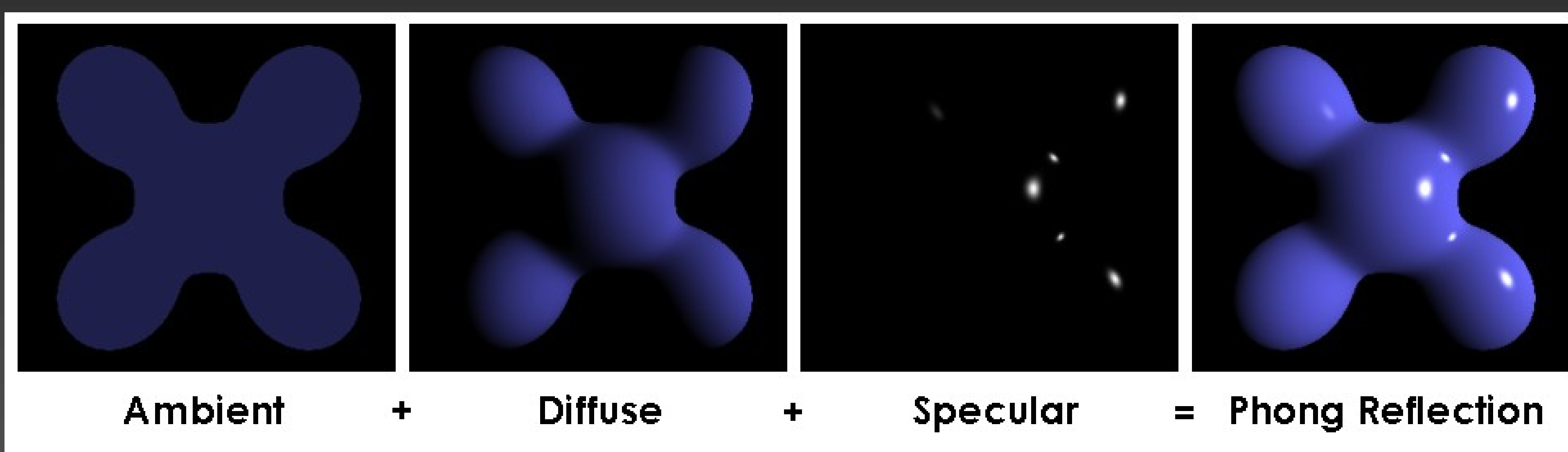


Specular Component

Phong Light Model







Material Color

- Final color value depends on combination of
  - 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 are usually same value
  - specular is 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 FragColor = 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;`
    - lowest quality, fastest
  - Gouraud shading (per vertex lighting)
    - use Phong lighting model to compute color in vertices
    - inside polygon (FS) – linear interpolation of colors from VS
    - simple HW, lower quality
  - Phong shading (per-fragment lighting)
    - starting vectors set in vertex shader
    - inside polygon (FS) – linear interpolation of vectors, used to compute Phong model for each fragment separately
    - higher quality (especially specular component)

Per-Vertex Point Light

```glsl
#version 460 core

// Vertex attributes
in vec4 aPosition;
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;
}
```

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

```glsl
#version 460 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

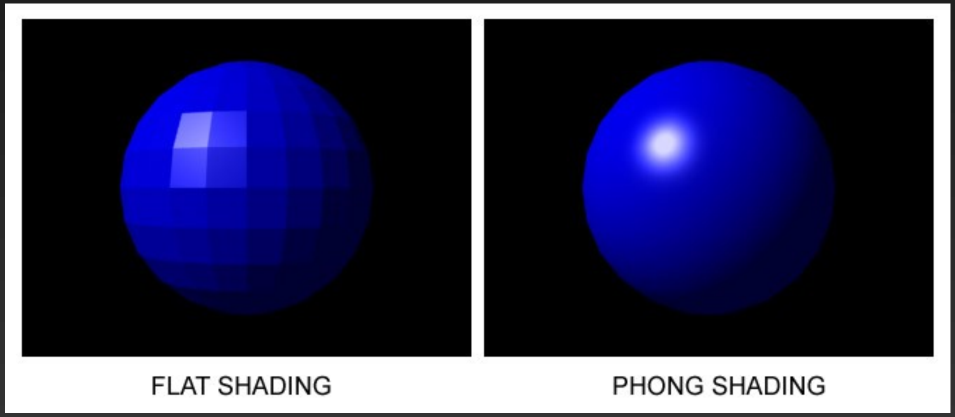
```
#version 460 core

// Vertex attributes
in vec4 aPosition;
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;
}
```

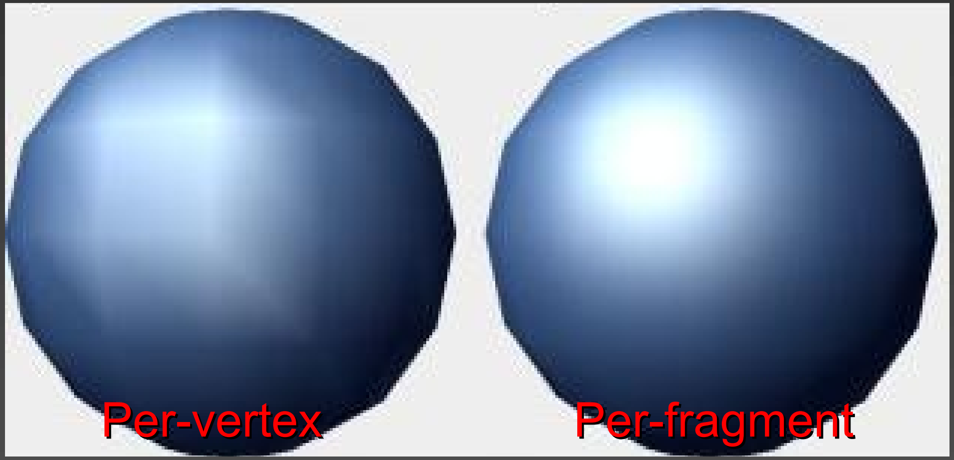
```
#version 460 core
out vec4 color;

// Material properties
uniform vec3 ambient_intensity, diffuse_intensity, specular_intensity;
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 * 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;



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

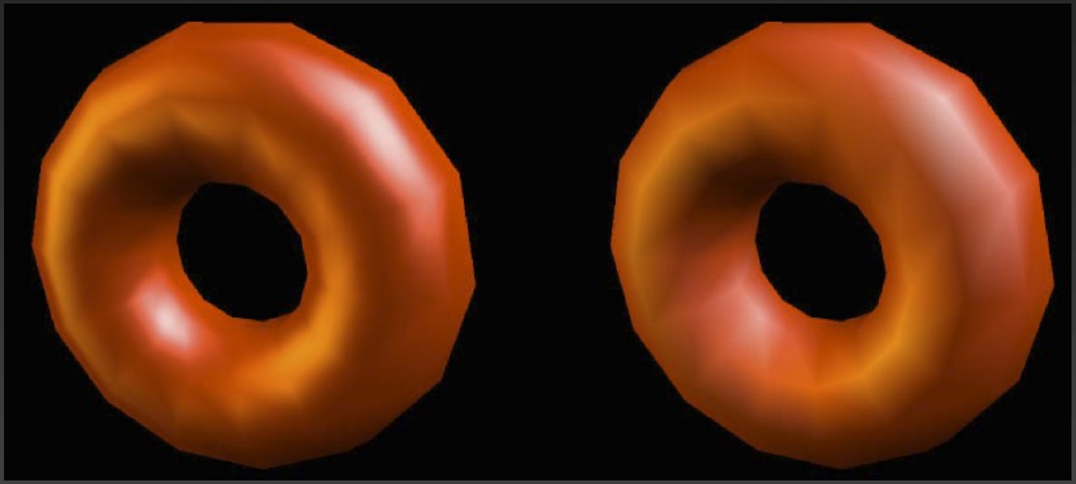
$$I_m = c_a \cdot i_a + c_d \cdot i_d \cdot (N \cdot L) + c_s \cdot i_s \cdot (V \cdot R)^n$$

Shading Types

- Flat vs. Interpolated
  - Interp.: per-vertex = Gouraud, per-fragment = Phong

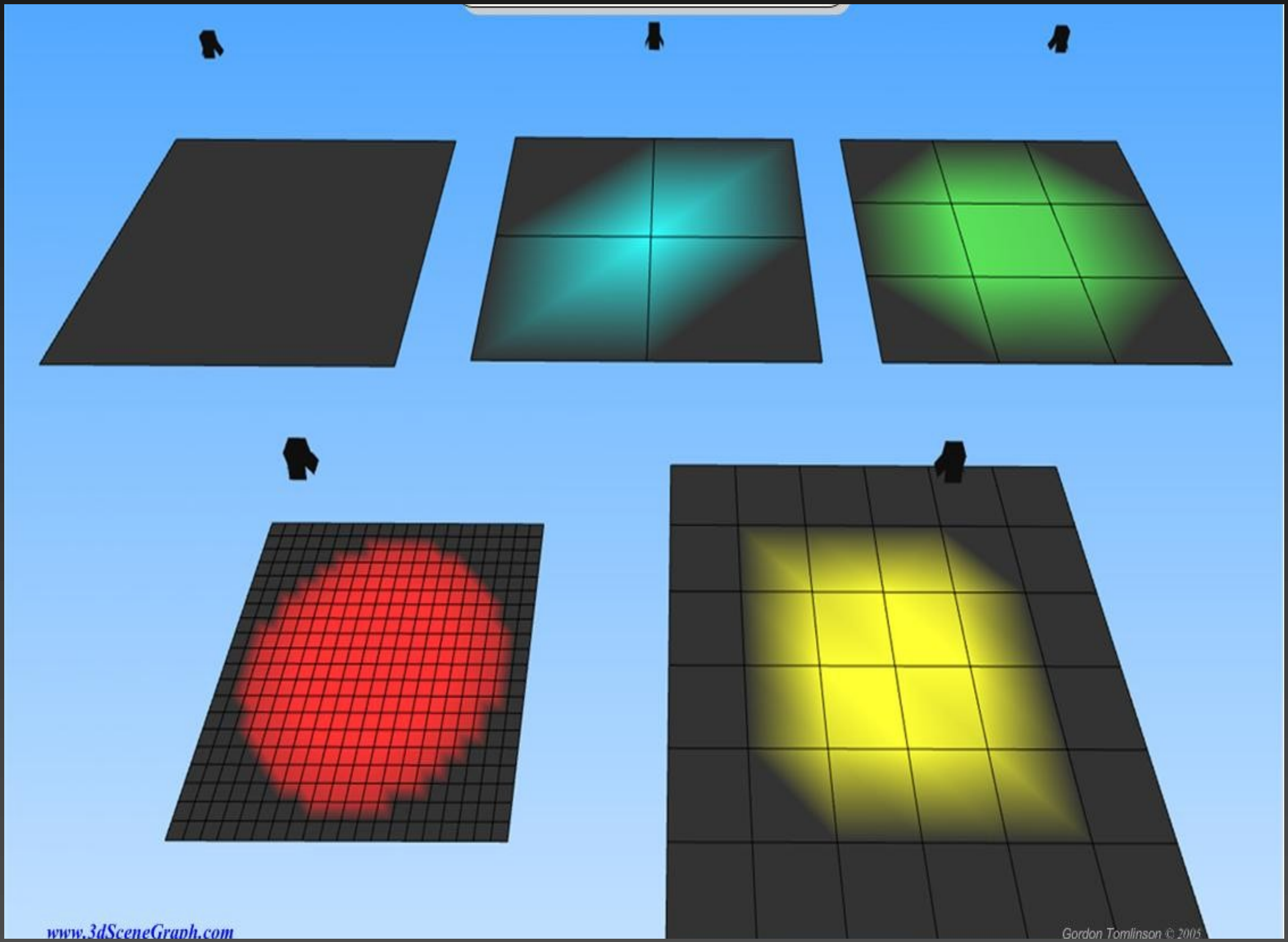






Lighting Problem

- Per-vertex lighting is fast, but innacurate
  - for HQ: per-fragment by shaders



HOWTO: Lighting

- Set (load or compute) 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

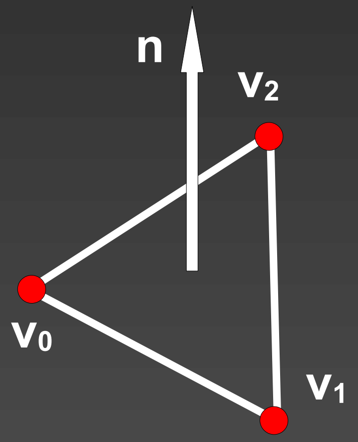
- Lighting computation takes time
  - partition to static and dynamic lights
  - static lights and static parts of scene can be baked together
    - light + dark corridor textures → light corridor textures
    - choose light/dark texture – no lights → fast

Normalisation

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

```
vec3 nn = normalize(n);
```

$$\vec{n} = (v_2 - v_0) \times (v_1 - v_0)$$
$$L = \sqrt{(n_x^2 + n_y^2 + n_z^2)}$$
$$\vec{n}_n = \frac{\vec{n}}{L}$$



Normal Transformation

- Homogeneous glm::scale()
  - easier
  - only for special cases

- Non-homogeneous glm::scale()
  - universal, slower, harder

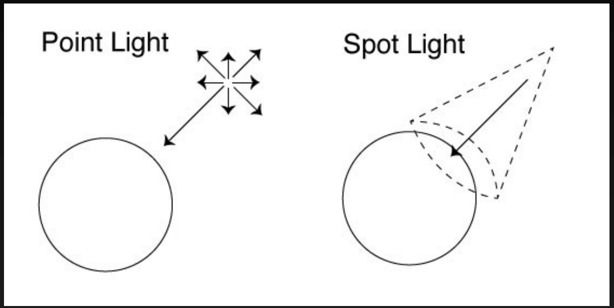
```
// C++

// normal_matrix is constant for all vertices.
// Do not waste GPU power - precompute and send uniform

glm::mat3 normal_matrix =
       glm::mat3(glm::inverseTranspose(model_matrix));

myshader.setUniform("uNm", normal_matrix));
```

```
#version 460 core
in vec3 aPos;
in vec3 aNormal;

uniform mat4 uMm = mat4(1.0);
uniform mat4 uVm = mat4(1.0);
uniform mat4 uPm = mat4(1.0);

out vec3 normal;

// VERTEX shader
void main() {
    // Outputs the positions/coordinates of all vertices
    gl_Position = uPm * uVm * uMm * vec4(aPos, 1.0f);

    normal = vec3(uMm * vec4(aNormal, 1.0));
}
```

```
#version 460 core
in vec3 aPos;
in vec3 aNormal;

uniform mat4 uMm = mat4(1.0);
uniform mat4 uVm = mat4(1.0);
uniform mat4 uPm = mat4(1.0);

uniform mat3 uNm = mat3(1.0); // precomputed

out vec3 normal;

// VERTEX shader
void main() {
    // Outputs the positions/coordinates of all vertices
    gl_Position = uPm * uVm * uMm * vec4(aPos, 1.0f);

    normal = uNm * aNormal;
}
```

Light Types And Properties

- Each light can have all components
  - ambient, diffuse, specular
  - different values of RGBA and other parameters (A is usually ignored)
  - Usually
    - specialized lights, same color of components
- Possible simplification
  - use w, angle 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, SpotLight
  - inside scene – [x, y, z, 1.0]
    → position = point
    VS: `vs_out.L = light_pos - P.xyz`




```
#version 460 core
// FRAGMENT SHADER

out vec4 FragColor;

// ... uniform parameters

vec4 DirectionalLight(...params...) {
    /* compute light and return result */
    return vec4(...);
}

vec4 PointLight(...params...) {
    /* compute light and return result */
    return vec4(...);
}

vec4 SpotLight(...params...) {
    /* compute light and return result */
    return vec4(...);
}

void main()
{

    if (light_position.w == 0.0)
        FragColor = DirectionalLight(...params...);
    else if (spotCutoff[i] == 180.0)
        FragColor = PointLight(...params...);
    else
        FragColor = SpotLight(...params...);
}
```

Visual Improvement, Light Types

- Point light + Spotlight
  - attenuation by distance
    - constant, linear, quadratic

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

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

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

Visual Improvement, Light Types

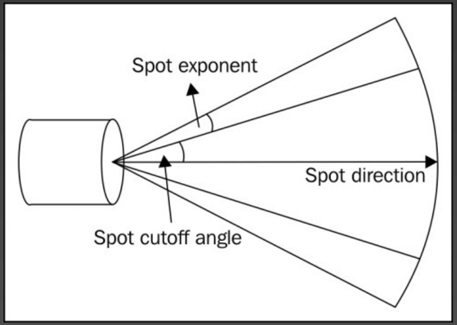
- Spotlight
  - position same as point source
  - light direction

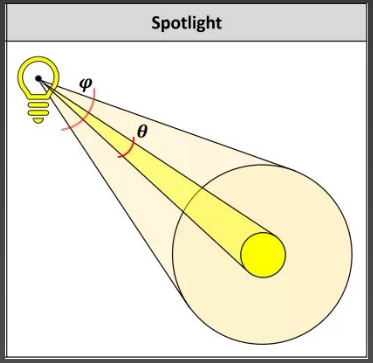
    `uniform glm::vec3 spotDirection;`
  - light cone (angle)

    `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);
else
    full_attenuation = 0.0; //out of cone

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





Per-Fragment Components, i.e. (Multi)Textures

- Need to compute partial light intensities separatelly

```
vec3 ambient_L = …
vec3 diffuse_L = …
vec3 specular_L = …
```

- than combine with per-fragment material constant
  - only diffuse texture

```
uniform sampler2D tex_diffuse;  // diffuse texture
FragColor = texture(tex_diffuse, texcoord0) * (full_attenuation*diffuse_L + ambient_L)
            + full_attenuation * specular_L;
```

  - multitexturing – diffuse and specular texture

```
uniform sampler2D tex_diffuse;  // diffuse texture
FragColor = texture(tex_diffuse, texcoord0) * (full_attenuation*diffuse_L + ambient_L)
          + texture(tex_specular, texcoord0) * full_attenuation * specular_L;
```

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 whole object

```
FragColor = radiation + (...lights, materials, textures...)
```

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

Uniforms: Subelements One-by-one, AoS (Array of Structures)

```
// C++
struct s_light {
    glm::vec4 position;
    glm::vec3 ambient;
    glm::vec3 diffuse;
    glm::vec3 specular;
    //... other params
};

GLint myLoc = glGetUniformLocation(progID, "light.ambient");
```

```
#version 460 core
struct s_light {
    vec4 position;
    vec3 ambient;
    vec3 diffuse;
    vec3 specular;
    //... other params
};

uniform s_light light;
```

```
// C++
struct s_light {
    glm::vec4 position;
    glm::vec3 ambient;
    //... other params
};

// AoS = Array of Structures ======>>>>> Comlicated, too many LOC 😥
constexpr int MAX_LIGHTS = 16;
s_light lights[MAX_LIGHTS];

GLint myLoc = glGetUniformLocation(progID, "lights[0].ambient");
GLint myLoc = glGetUniformLocation(progID, "lights[0].diffuse");
//...
GLint myLoc = glGetUniformLocation(progID, "lights[1].ambient");
//...all other params for all other lights...
```

```
#version 460 core
struct s_light {
    vec4 position;
    vec3 ambient;
    vec3 diffuse;
    vec3 specular;
    //... other params
};

#define MAX_LIGHTS 16 // MUST be know at compile time
uniform s_light lights[MAX_LIGHTS];
```

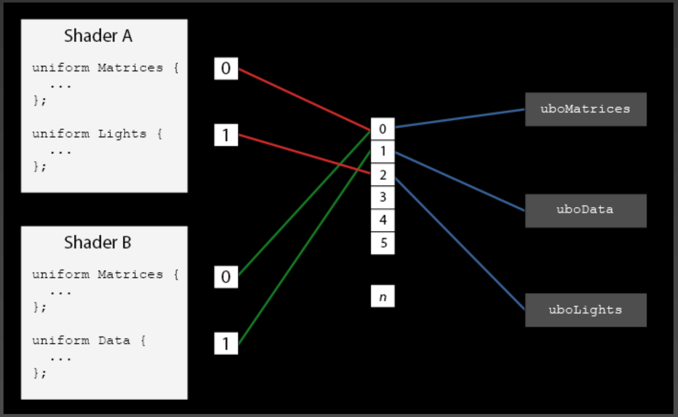
```
// C++
struct s_light {
    glm::vec4 position;
    glm::vec3 ambient;
    //... other params
};

//  repeatedly generated names  ====>>>>> SLOW & UGLY 😥
for (int i = 0; i < MAX_LIGHTS; i++) {
  std::string number = std::to_string(i);

  glGetUniformLocation(progID, ("pointLight[" + number + "].position").c_str());
  glGetUniformLocation(progID, ("pointLight[" + number + "].ambient").c_str());
  //...other params...
}
```

Uniforms: SoA (Structure of Arrays)

```cpp
// C++

// SoA = Structure of Arrays (C like, compile time allocated)
#define MAX_LIGHTS = 16;
struct s_lights {
    glm::vec4 position[MAX_LIGHTS];
    glm::vec3 ambient[MAX_LIGHTS];
    glm::vec3 diffuse[MAX_LIGHTS];
    glm::vec3 specular[MAX_LIGHTS];
    //... other params
};
s_lights lights;

//-------------------------------------------
// SoA = Structure of Arrays (C++ like, compile time allocated)
constexpr int MAX_LIGHTS = 16;

struct s_lights {
    std::array<glm::vec4, MAX_LIGHTS> position;
    std::array<glm::vec3, MAX_LIGHTS> ambient;
    std::array<glm::vec3, MAX_LIGHTS> diffuse;
    std::array<glm::vec3, MAX_LIGHTS> specular;    //... other params
};
s_lights lights;

//==============================================================================
// send per attribute, but WHOLE ARRAY AT ONCE 😎
void send_data() {
    GLint myLoc = glGetUniformLocation(progID, "lights.position");
    glUniform4fv(myLoc, MAX_LIGHTS, glm::value_ptr(lights.position[0]));

    GLint myLoc = glGetUniformLocation(progID, "lights.ambient");
    glUniform3fv(myLoc, MAX_LIGHTS, glm::value_ptr(lights.ambient[0]));

    //...
}
```

```glsl
#version 460 core

#define MAX_LIGHTS 16

struct s_lights {
    vec4 position[MAX_LIGHTS];
    vec3 ambient[MAX_LIGHTS];
    vec3 diffuse[MAX_LIGHTS];
    vec3 specular[MAX_LIGHTS];
    //... other params
};

uniform s_lights lights;
```

Best solution: UBO
Uniform Buffer Objects
(directly map C++ mem to uniforms)



Multiple Switchable Lights (two options)

😊 simple: C++ always sends all uniforms
😒 slower: send unused, branch in shader

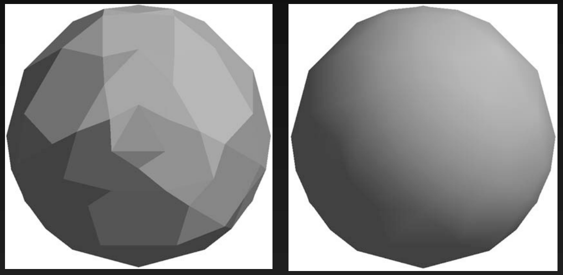
```
#version 460 core
#define MAX_LIGHTS 16

struct s_lights {
    bool active[MAX_LIGHTS]; // per-light activation
    vec4 position[MAX_LIGHTS];
    vec3 ambient[MAX_LIGHTS];
    vec3 diffuse[MAX_LIGHTS];
    vec3 specular[MAX_LIGHTS];
    float spotCutoff[MAX_LIGHTS];
    //... other params
};

uniform s_lights lights;

vec4 DirectionalLight(int i) {
    /* compute light and return result */
    return vec4(0.0);
}
vec4 PointLight(int i) {
    /* compute light and return result */
    return vec4(0.0);
}
vec4 SpotLight(int i) {
    /* compute light and return result */
    return vec4(0.0);
}
void main() {
    vec3 accumulator = vec3(0.0);

    // Loop through enabled lights
    for (int i = 0; i < MAX_LIGHTS; i++) {
        if (lights.active[i]) {
            if (light.position[i].w == 0.0)
                accumulator += DirectionalLight(i);
            else if (spotCutoff[i] == 180.0)
                accumulator += PointLight(i);
            else
                accumulator += SpotLight(i);
        }
    }

  FragColor = vec4(accumulator, 1.0);
}
```

😊 faster: no branch in shader, send only used
😒 complex: C++ must reorder and count lights

```
#version 460 core
#define MAX_LIGHTS 16

struct s_lights {
    vec4 position[MAX_LIGHTS];
    vec3 ambient[MAX_LIGHTS];
    vec3 diffuse[MAX_LIGHTS];
    vec3 specular[MAX_LIGHTS];
    float spotCutoff[MAX_LIGHTS];
    //... other params
};

uniform s_lights lights;
uniform int active_lights; // light count

vec4 DirectionalLight(int i) {
    /* compute light and return result */
    return vec4(0.0);
}
vec4 PointLight(int i) {
    /* compute light and return result */
    return vec4(0.0);
}
vec4 SpotLight(int i) {
    /* compute light and return result */
    return vec4(0.0);
}
void main() {
    vec3 accumulator = vec3(0.0);

    // Loop through enabled lights
    for (int i = 0; i < active_lights; i++) {
        if (light.position[i].w == 0.0)
            accumulator += DirectionalLight(i);
        else if (spotCutoff[i] == 180.0)
            accumulator += PointLight(i);
        else
            accumulator += SpotLight(i);
    }

  FragColor = vec4(accumulator, 1.0);
}
```

Shading of Connected Polygons

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



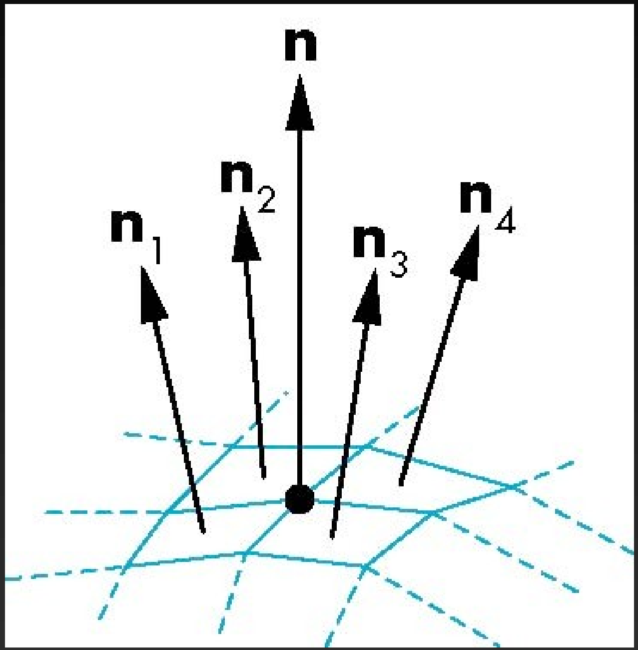
- Independent normal for each vertex
  - best: load from file 😎
  - manual computation: normal in shared vertex is average of implicit normals of connected polygons → continuous interpolation → smooth look
  - Be careful: not mathematically correct! (plane has only one normal by definition)

Averaging Normals

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

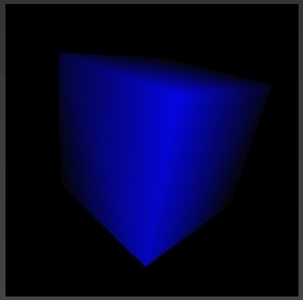
```
for each primitive
    for each vertex of primitive
        for each vertex of model
            if vertex_shared { n=avg(n1,n2n,…) }
```

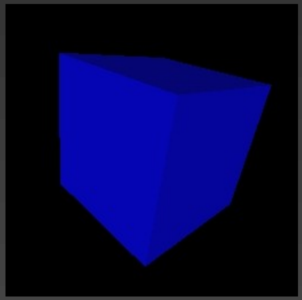
$$n = (n1+n2+n3+n4)/ |n1+n2+n3+n4|$$



- 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 want sharp edge
  - user defined angle limit for averaging

```
if (normal_difference < 70°) {
     n=avg(n1, n2, n3, n4);
}
```





Online Example

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