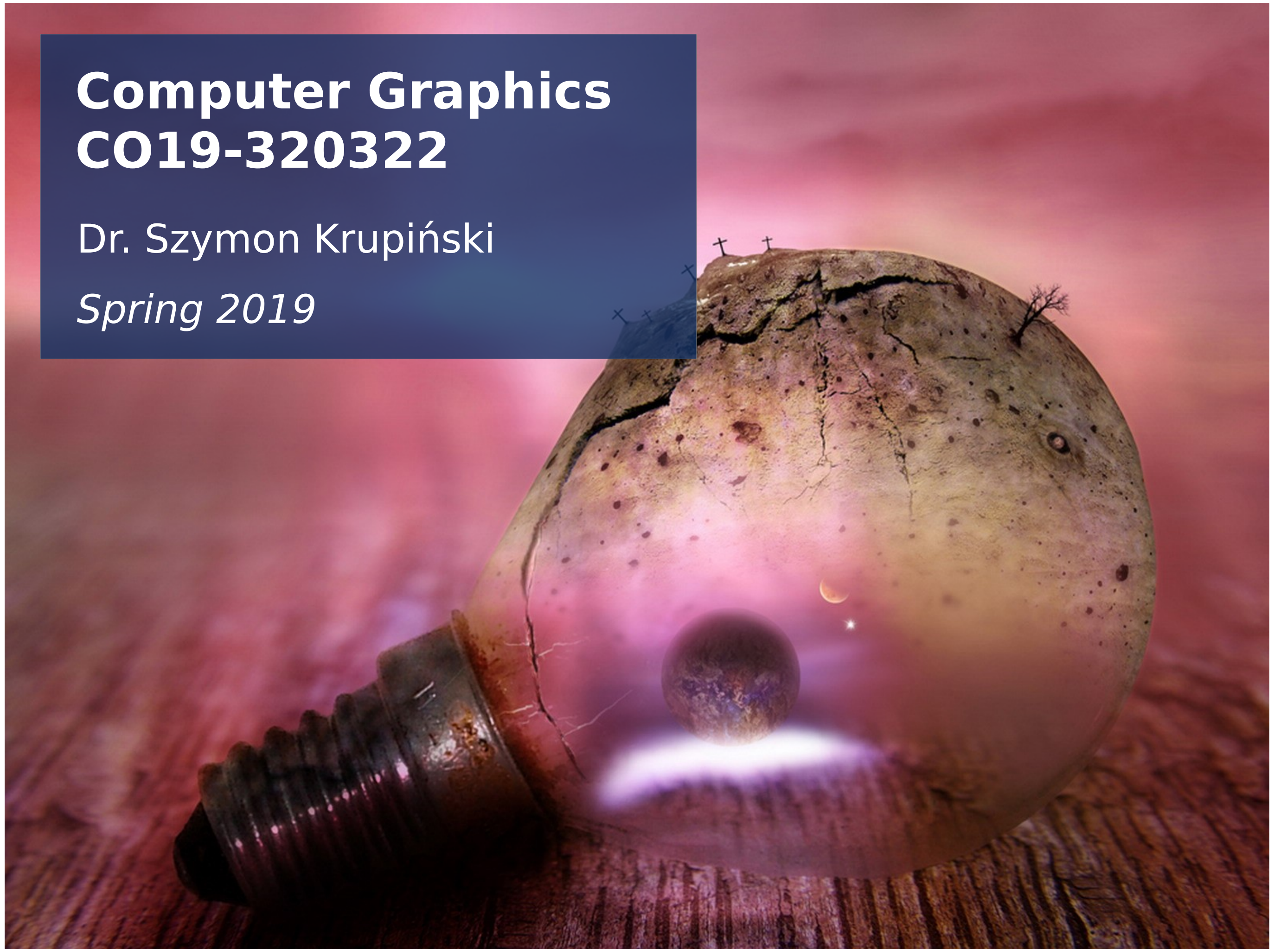
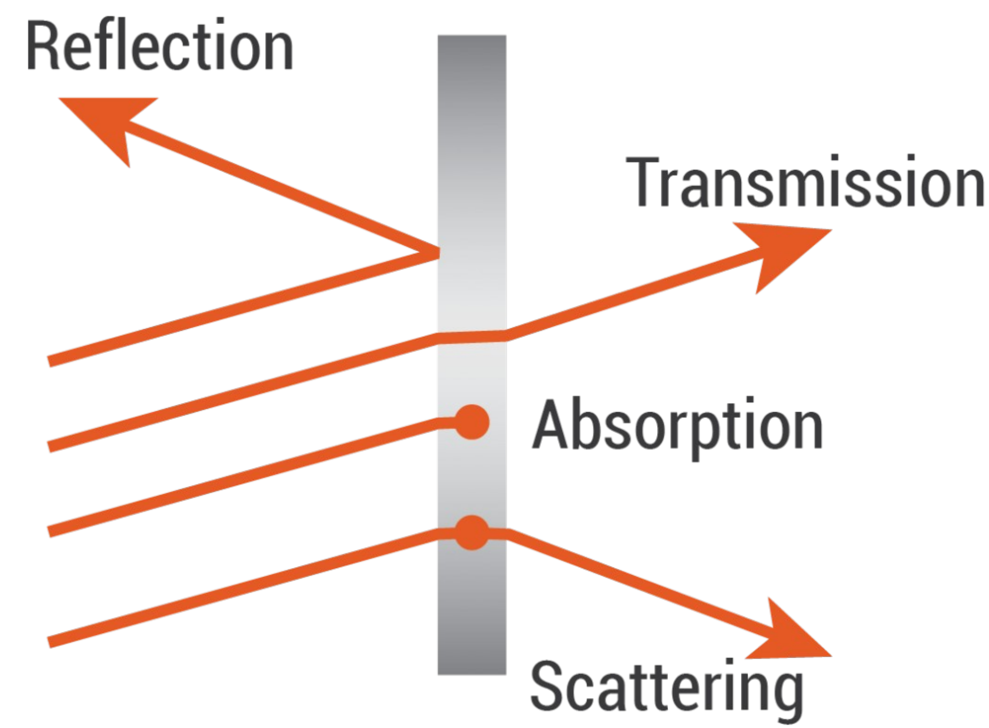
Lecture 15: OpenGL color, lighting and texturing

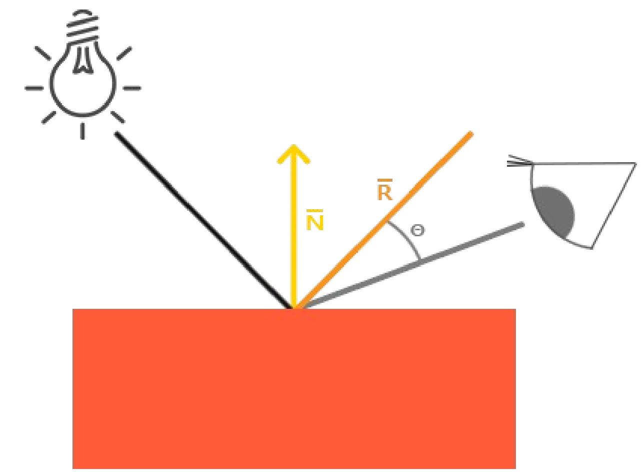


Recent subjects

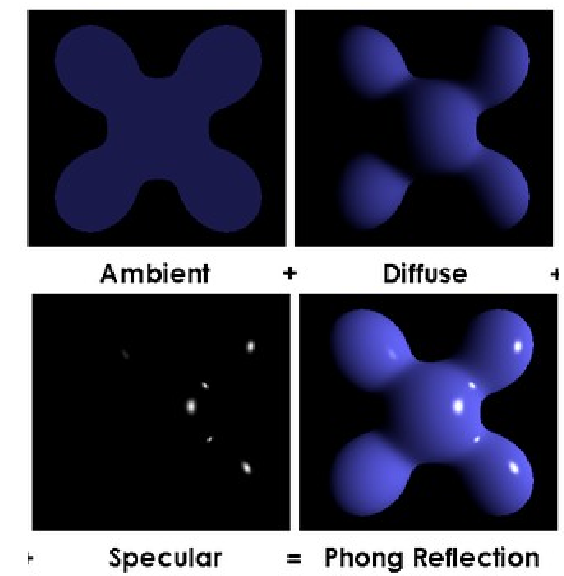
- How do we simulate the way light behaves in our 3-D scene?



Beyond the reflection model



- Reflection model allows us to compute the intensity (and color!) of light “produced” by one pixel



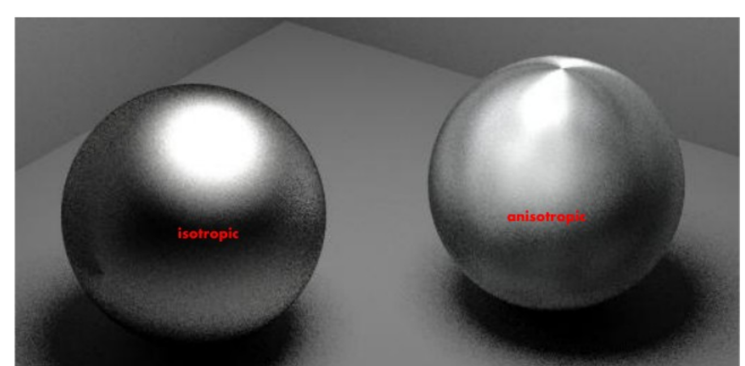
- Phong model is “good enough” for most of the situations

$$L = \sum_{j \epsilon lights} L_i^j (k_a + k_d(\hat{\mathbf{N}} \cdot \omega_i^j)_+ + k_s(\hat{\mathbf{N}} \cdot \hat{\mathbf{H}}_i^j)_+^s)$$

ambient diffuse specular

sum all lights

  - Some not uncommon details cannot be produces such as subsurface scattering or anisotropic specular highlights



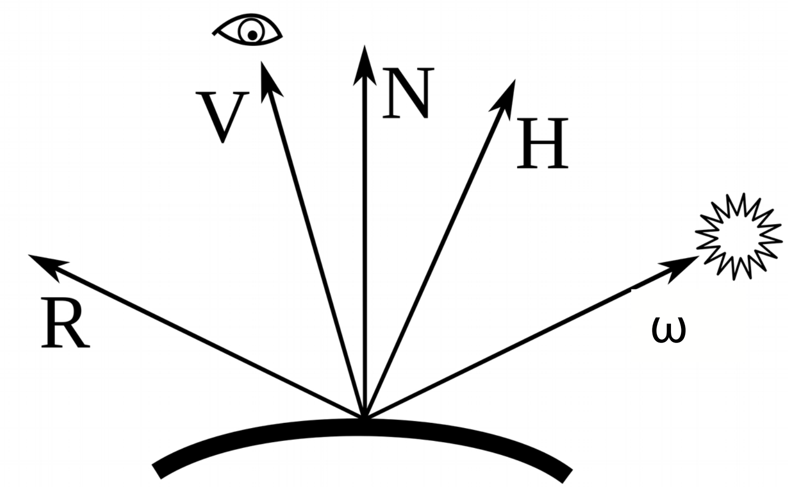
Phong details

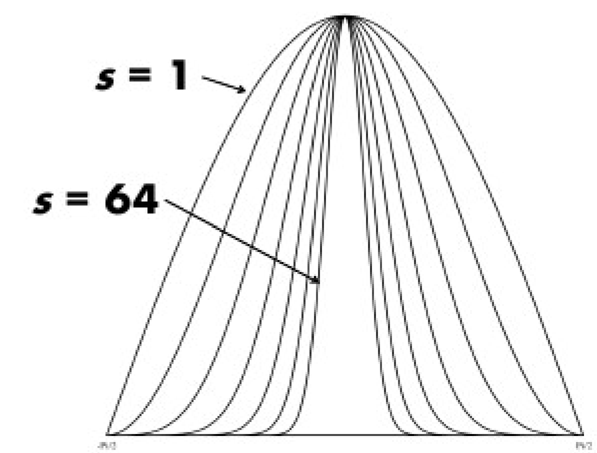
- Specular term

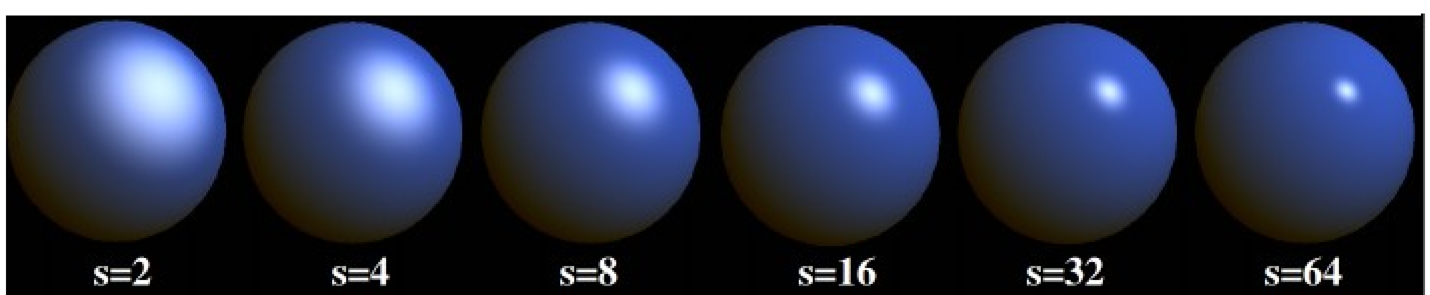
$$L_o = \sum_{j \in lights} \Big(\underbrace{k_a \hat{I}^j_{i,a}}_{\text{Ambient}} + \underbrace{k_d \hat{I}^j_{i,d} \max(0, \omega_{i,d} \cdot \hat{\mathrm{N}})}_{\text{Diffuse}} + \underbrace{\boxed{k_s \hat{I}^j_{i,s} \max(\mathrm{V} \cdot \mathrm{R}^j, 0)^s}}_{\text{Specular}} \Big)$$

cos() around the viewing reflection angle

- "shininess coefficient" - the exponent decides how shiny the object appears and the reflection converges to mirror reflection for higher powers





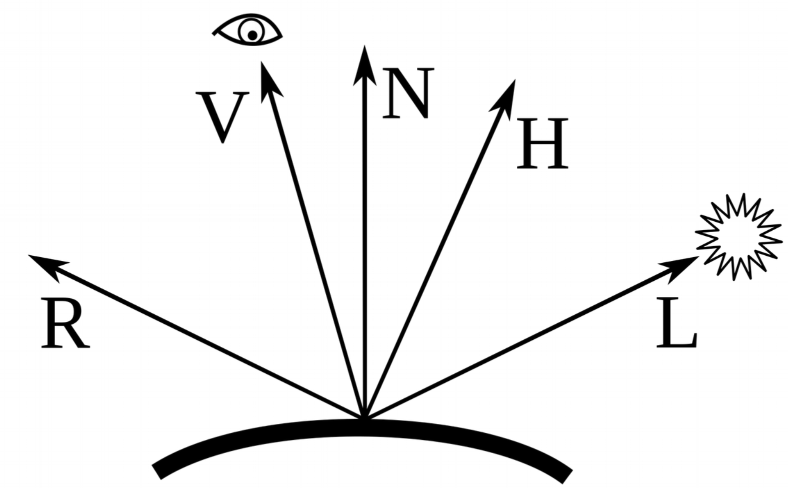


Blinn–Phong optimisation

- A way to optimise the calculation of the diffuse term:

$$L_o = k_s \hat{I}_{i,s} (N \cdot \mathrm{H})^s$$

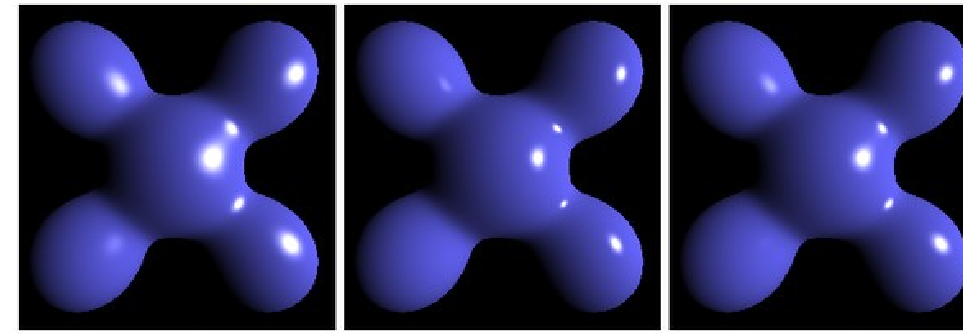
- H is the “half-way” angle
  - still viewing direction dependent, since H depends on V
  - if the viewer and the light source are very (infinitely) far away, L and V are unchanged for all pixels and H only need be computed once for the entire image



$$\mathbf{H} = (\mathbf{L} + \mathbf{V}) \,/\, |\,\mathbf{L} + \mathbf{V}\,|$$

Blinn–Phong optimisation

- Blinn–Phong was the default shading model used in OpenGL and Direct3D in the old pipeline model (“fixed-function”) for per-vertex shading
  - in per-**vertex lighting** the color is computed for each vertex and then it is interpolated between vertices. In per-**pixel lighting** normals are interpolated between vertices and the color is computed on each **fragment**
  - A **fragment** is a collection of values produced by the rasterizer. Each fragment represents a sample-sized segment of a rasterized triangle.
    - size covered is related to the pixel area
    - rasterization can produce multiple fragments from the same triangle per-pixel, depending on various multisampling parameters and OpenGL state
    - there will be at least one fragment produced for every pixel area covered by the primitive being rasterized



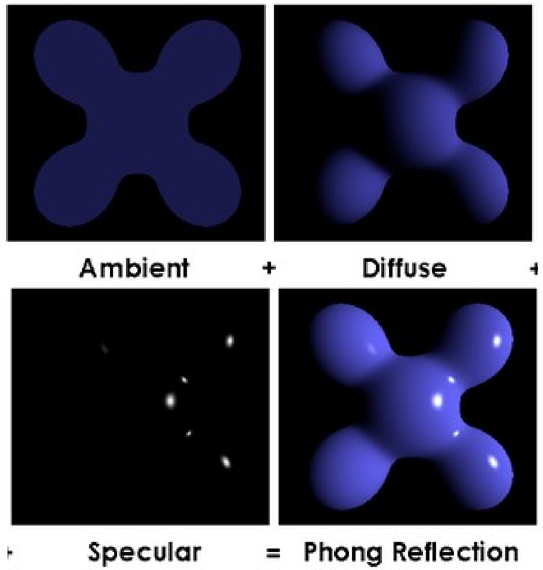
Blinn–Phong | Phong | Blinn–Phong (higher exponent)

- Screen position
- Vertex x,y and z from projection
- From front or back of triangle?
- ...

Applied in OpenGL

- Old style: all done for us if we accept the "default"

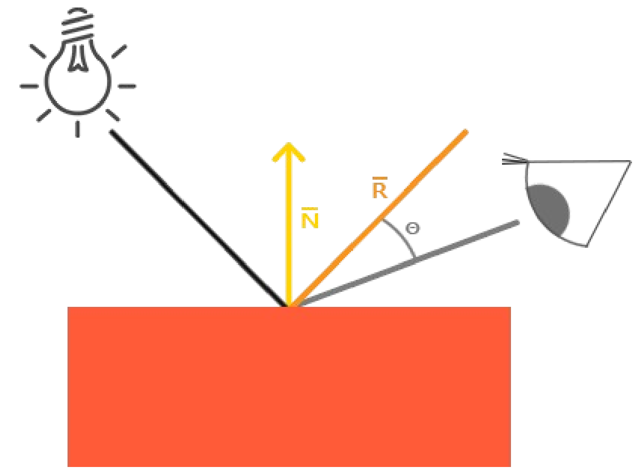
```
float lightZeroPosition[] = {10.0, 4.0, 10.0, 1.0};
float lightZeroColor[] = {0.8, 1.0, 0.8, 1.0};
glLightfv(GL_LIGHT0, GL_POSITION, lightZeroPosition);
glLightfv(GL_LIGHT0, GL_DIFFUSE, lightZeroColor);
glEnable(GL_LIGHT0);
glEnable(GL_LIGHTING);
float ambColor[] =  {1.0, 0.0, 0.0, 1.0};
float difColor[] =  {1.0, 0.0, 0.0, 1.0};
glMaterialfv(GL_FRONT, GL_AMBIENT, ambColor);
glMaterialfv(GL_FRONT, GL_DIFFUSE, difColor);
```

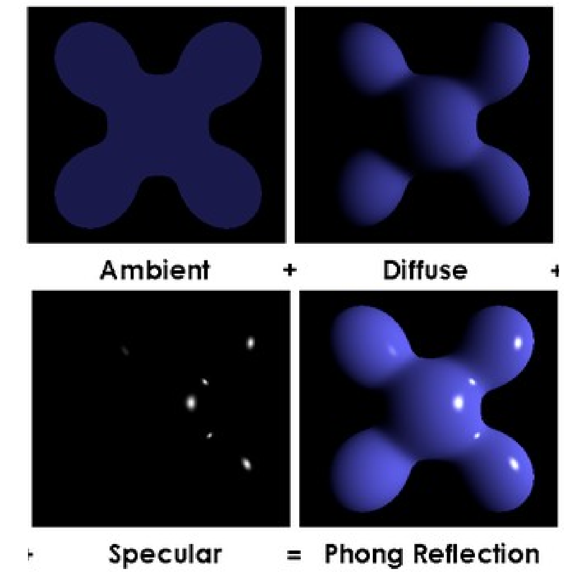


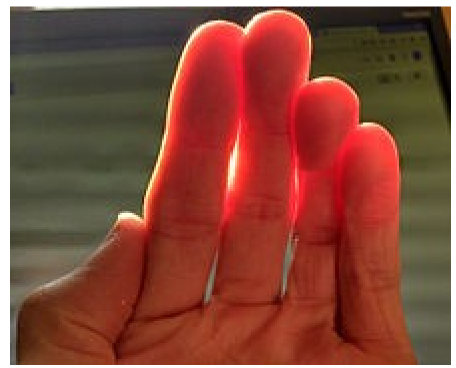
Normals

- The value of normal vectors is one of the critical information for the lighting calculations
- Can be manually defined for each vertex separately

```
glBegin (GL_POLYGON);
   glNormal3fv(n0);
   glVertex3fv(v0);
   glNormal3fv(n1);
   glVertex3fv(v1);
   glNormal3fv(n2);
   glVertex3fv(v2);
   glNormal3fv(n3);
   glVertex3fv(v3);
glEnd();
```

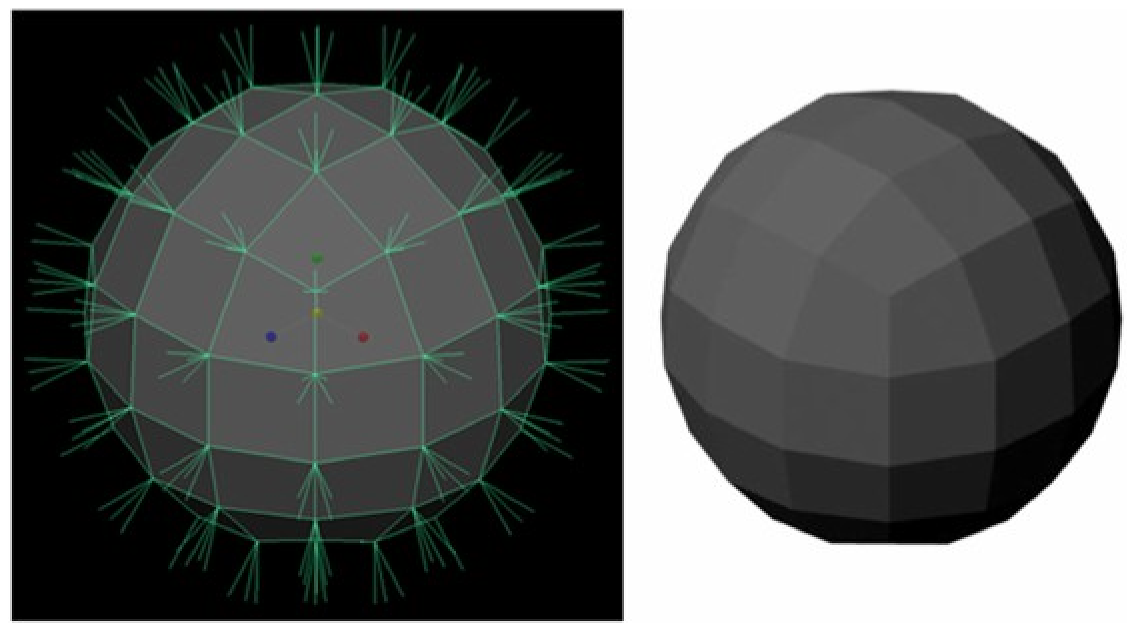






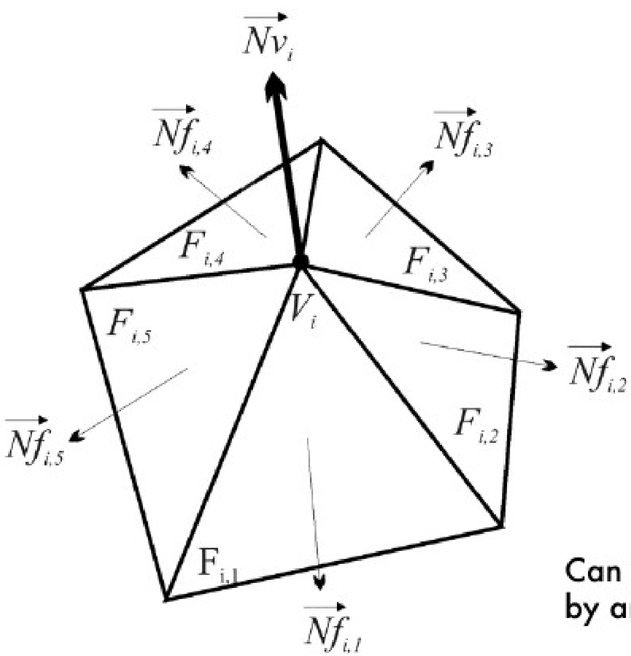
Normals

- If this would be the only available information, the light simulation would be rather crude…



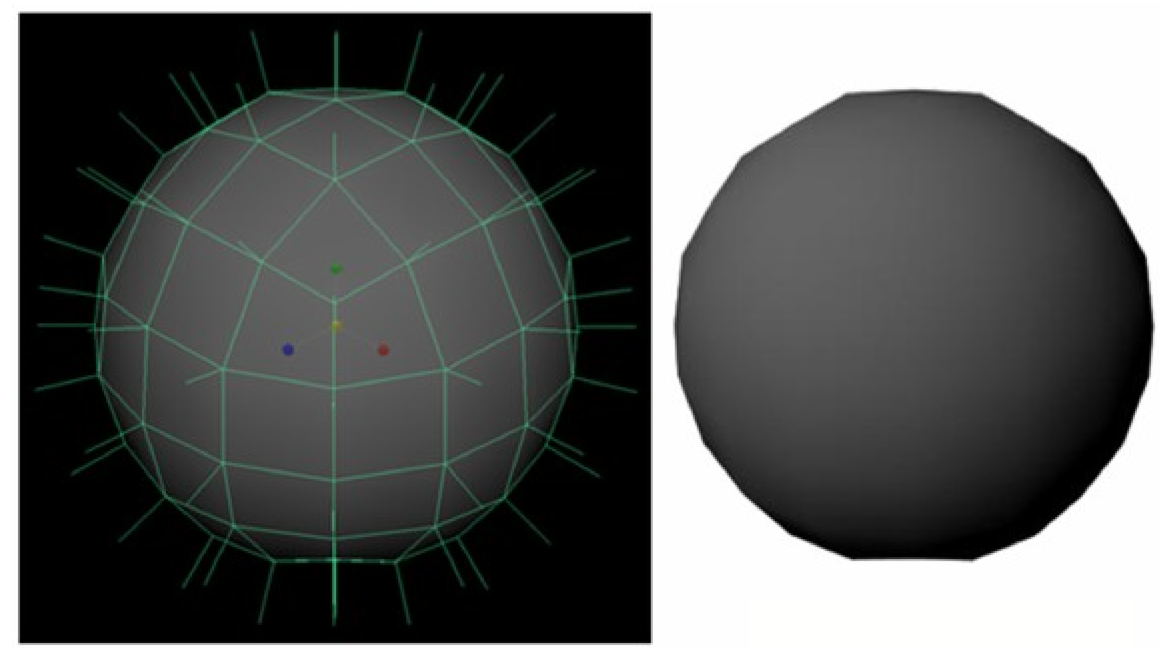
- Normals can also be interpolated!

Normals interpolation



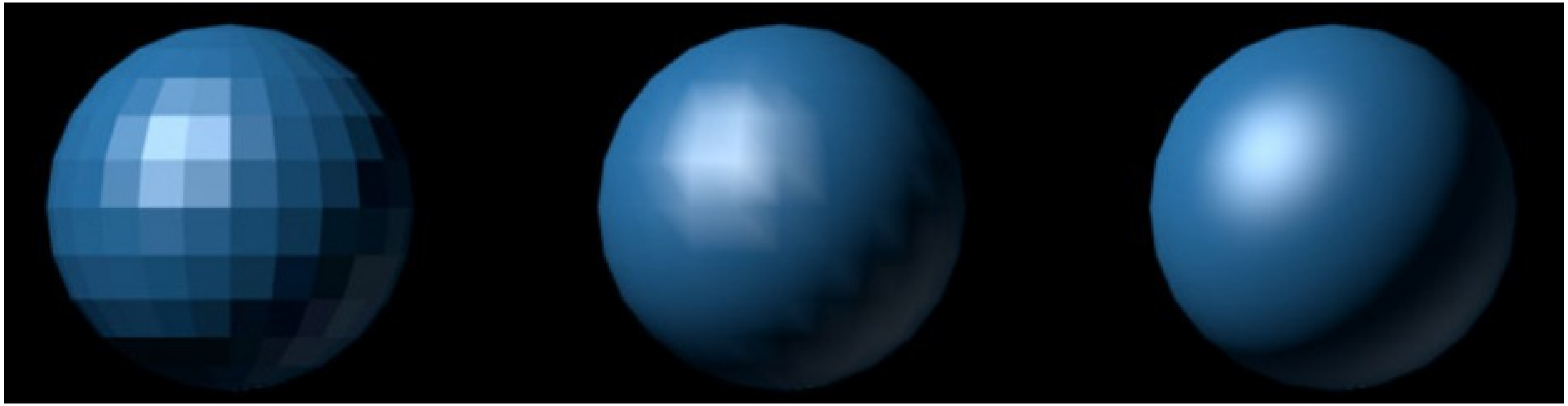
$$\overrightarrow{Nv_i} = \frac{\sum_{k=1}^{n} \overrightarrow{Nf_{i,k}}}{n}$$

Can weight triangle normals by angle, area, etc.



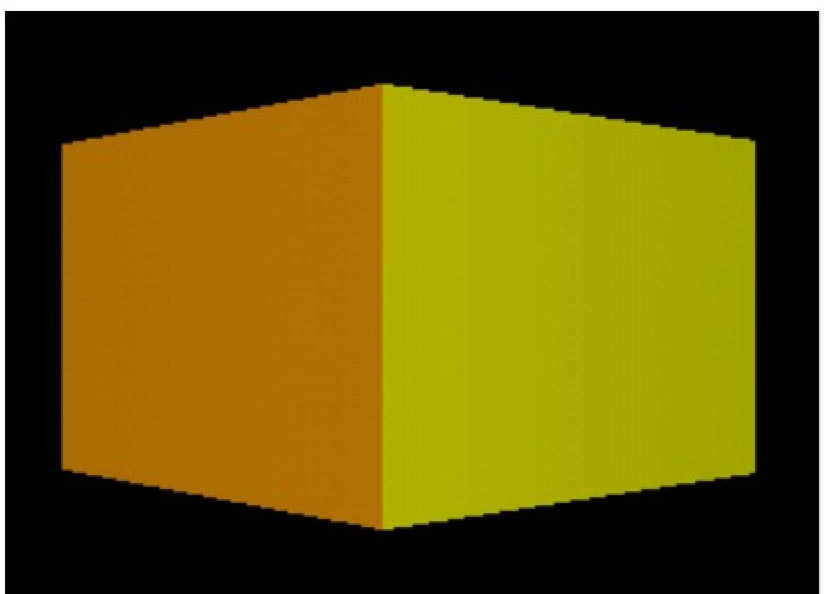
Normals interpolation

- This is one of the key questions of shading!
  - **Flat** shading (uses actual triangle normals): one luminance evaluation per triangle. Every pixel in the triangle gets the same color
  - **Gouraud** shading (uses vertex normals): one luminance evaluation per vertex. The resulting vertex colors are interpolated to the interior pixels of each triangle
  - **Phong** shading (uses vertex normals): one luminance evaluation per pixel. Each pixel uses a vertex normal interpolated to the pixel location

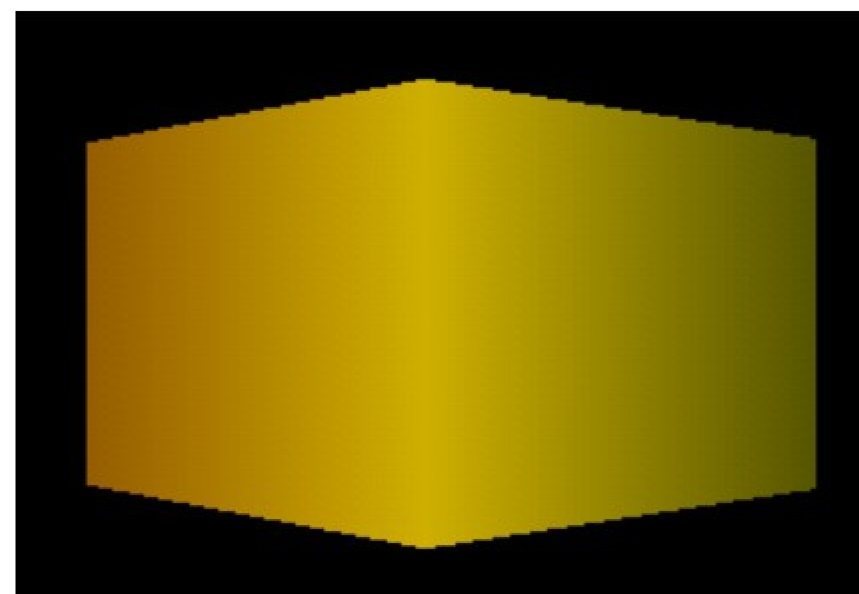


“Corner cases”

- Normals are poorly defined and difficult to compute at corners
- If we use averaged vertex normals to shade a cube, the edges have unrealistic lighting
- Need to specify what type of shader to use for different parts of the object (the same triangle may need both flat and smooth shading!)



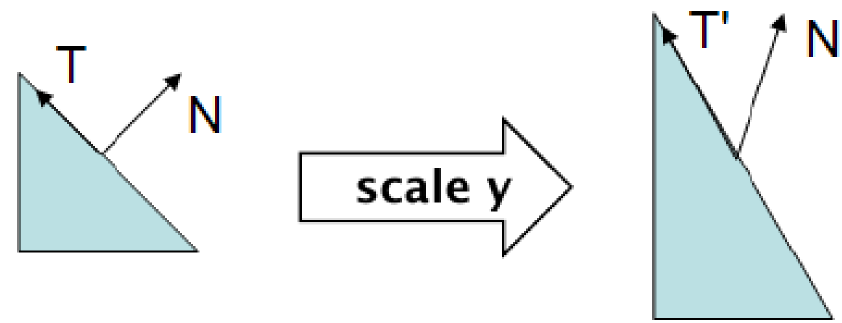
flat shading



smooth shading

Transforming normals?

- Can't just multiply the normal by the model/view matrix
  - If the model/view matrix is non-orthogonal, e.g. contains a non-uniform scaling, then the normal will be wrong
  - Translation and rotation preserve lengths and angles
  - Scale, shear, etc. do not preserve lengths and angles
- We can preserve dot products N·V for arbitrary V
  - Perspective transforms do not preserve lengths and angles and cannot preserve dot products N·V
- Thus, lighting calculations that depend on angles must be done before the perspective transformation
  - … unless we keep all the necessary data for calculations somewhere on the side (for shaders)

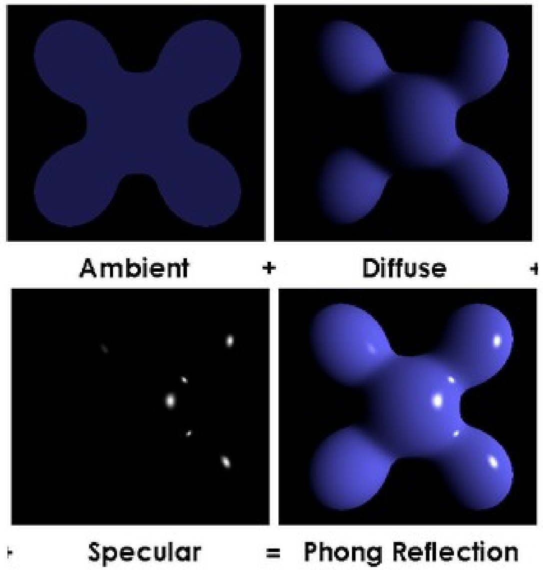


New style OpenGL shading?

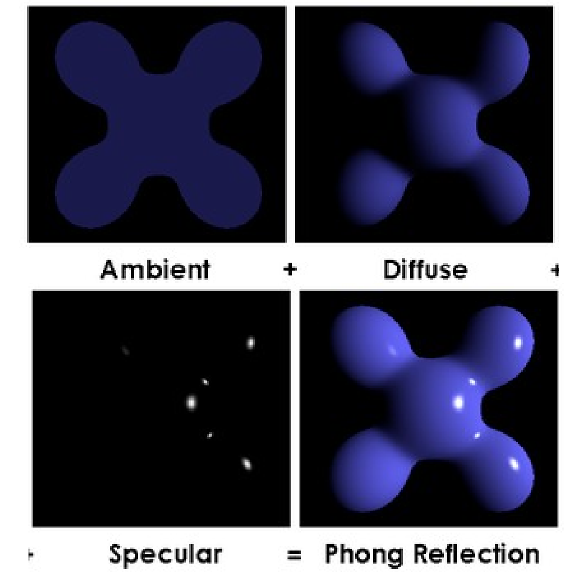
- We are getting dangerously close to the GLSL territory!
  - Two mini-programs typically used to evaluate every luminance evaluation:
    - Vertex shader
    - Fragment shader

Vertex shader

```
attribute vec3 inputPosition;
attribute vec2 inputTexCoord;
attribute vec3 inputNormal;

uniform mat4 projection, modelview, normalMat;

varying vec3 normalInterp;
varying vec3 vertPos;

void main(){
    gl_Position = projection * modelview * vec4(inputPosition, 
1.0);
    vec4 vertPos4 = modelview * vec4(inputPosition, 1.0);
    vertPos = vec3(vertPos4) / vertPos4.w;
    normalInterp = vec3(normalMat * vec4(inputNormal, 0.0));
}
```



Fragment shader




```
precision mediump float;

varying vec3 normalInterp;
varying vec3 vertPos;

uniform int mode;

const vec3 lightPos = vec3(1.0,1.0,1.0);
const vec3 lightColor = vec3(1.0, 1.0, 1.0);
const float lightPower = 40.0;
const vec3 ambientColor = vec3(0.1, 0.0, 0.0);
const vec3 diffuseColor = vec3(0.5, 0.0, 0.0);
const vec3 specColor = vec3(1.0, 1.0, 1.0);
const float shininess = 16.0;
const float screenGamma = 2.2; // Assume the
monitor is calibrated to the sRGB color space

void main() {

  vec3 normal = normalize(normalInterp);
  vec3 lightDir = lightPos - vertPos;
  float distance = length(lightDir);
  distance = distance * distance;
  lightDir = normalize(lightDir);

  float lambertian = max(dot(lightDir,normal),
0.0);
  float specular = 0.0;

  if (lambertian > 0.0) {

    vec3 viewDir = normalize(-vertPos);

    // this is blinn phong
    vec3 halfDir = normalize(lightDir + viewDir);
    float specAngle = max(dot(halfDir, normal), 0.0);
    specular = pow(specAngle, shininess);

    // this is phong (for comparison)
    if(mode == 2) {
      vec3 reflectDir = reflect(-lightDir, normal);
      specAngle = max(dot(reflectDir, viewDir), 0.0);
      // note that the exponent is different here
      specular = pow(specAngle, shininess/4.0);
    }
  }
  vec3 colorLinear = ambientColor +
         diffuseColor * lambertian * lightColor * lightPower / distance +
         specColor * specular * lightColor * lightPower / distance;
  // apply gamma correction (assume ambientColor, diffuseColor and specColor
  // have been linearized, i.e. have no gamma correction in them)
  vec3 colorGammaCorrected = pow(colorLinear, vec3(1.0/screenGamma));
  // use the gamma corrected color in the fragment
  gl_FragColor = vec4(colorGammaCorrected, 1.0);
}
```

Thank you!

- Questions?