Lighting and Shading Part 1

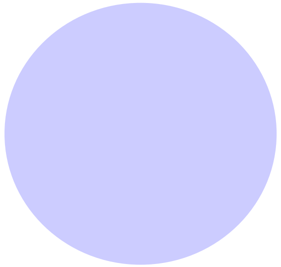
Sorin Babii
sorin.babii@cs.upt.ro

Objectives

- Learn to shade objects so their images appear three-dimensional
- Introduce the types of light-material interactions
- Build a simple reflection model – the Phong model – that can be used with real time graphics hardware

Why we need shading

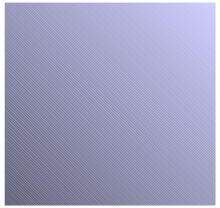
Suppose we build a model of a sphere using many polygons and color it with **`glColor`**. We get something like



But we want

Shading

- Why does the image of a real sphere look like

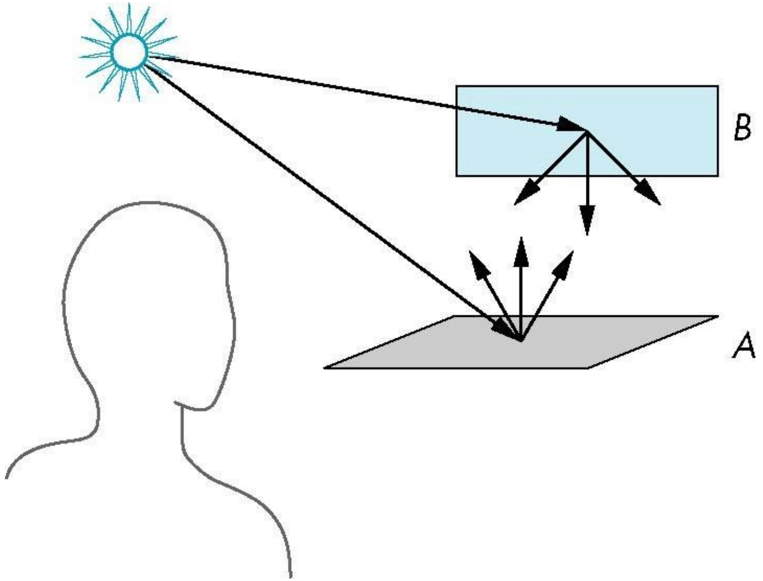


- Light-material interactions cause each point to have a different color or shade
- Need to consider
  - Light sources
  - Material properties
  - Location of viewer
  - Surface orientation

Scattering

- Light strikes A
  - Some scattered
  - Some absorbed
- Some of scattered light strikes B
  - Some scattered
  - Some absorbed
- Some of this scattered light strikes A

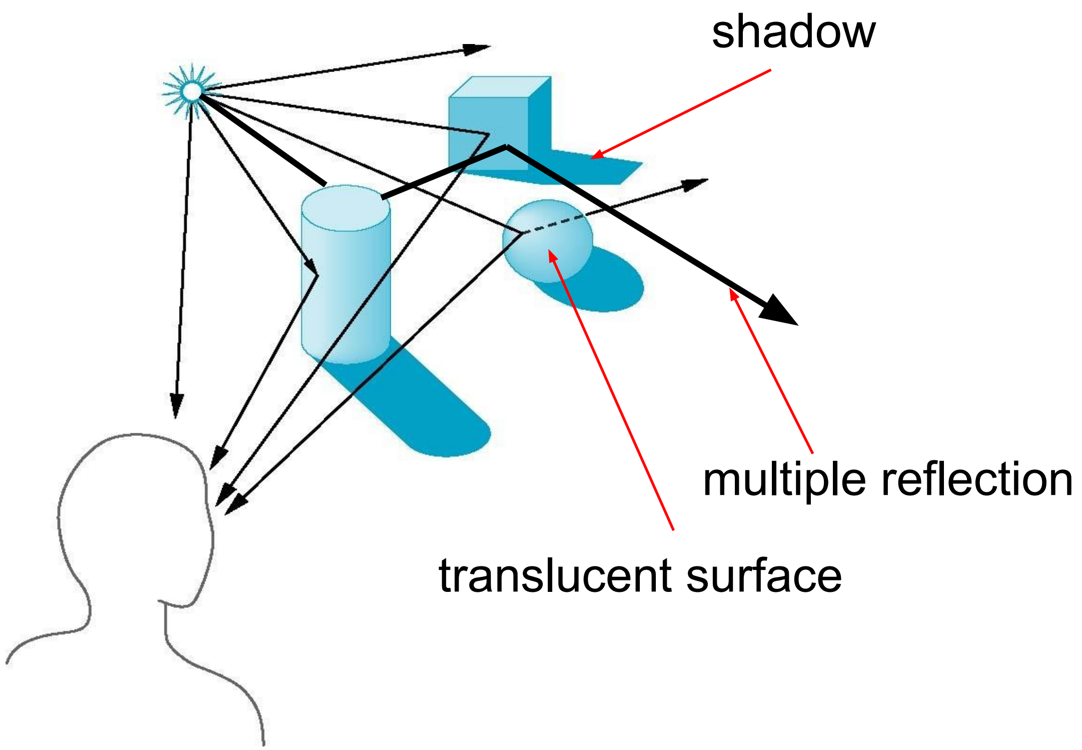
  and so on



Rendering Equation

- The infinite scattering and absorption of light can be described by the *rendering equation*
  - Cannot be solved in general
  - Ray tracing is a special case for perfectly reflecting surfaces
- Rendering equation is global and includes
  - Shadows
  - Multiple scattering from object to object

Global Effects



Local vs Global Rendering

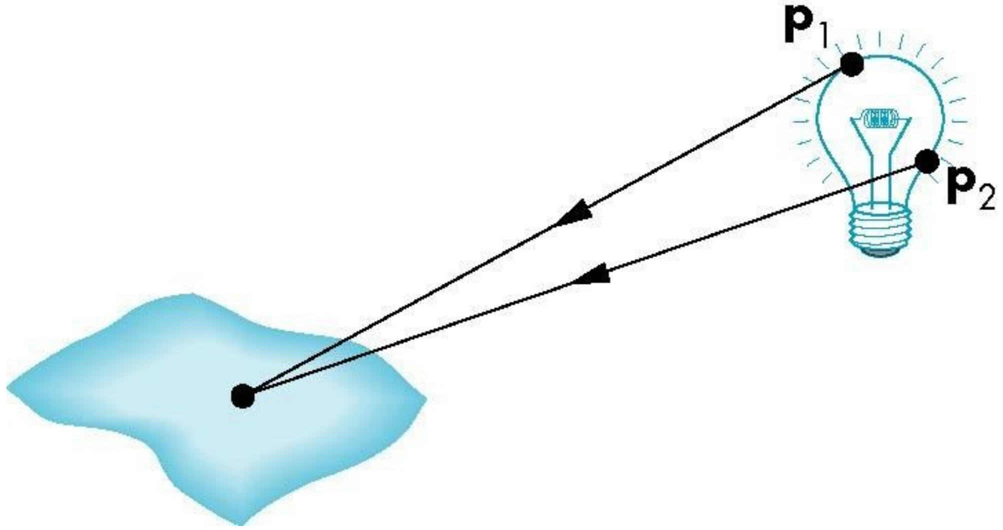
- Correct shading requires a global calculation involving all objects and light sources
  - Incompatible with pipeline model, which shades *each polygon independently* (local rendering)
- However, in computer graphics, especially real time graphics, we are happy if things “look right”
  - There are many techniques for *approximating* global effects

Light-Material Interaction

- Light that strikes an object is partially absorbed and partially scattered (reflected)
- The amount reflected determines the color and brightness of the object
    - A surface appears red under white light because the red component of the light is reflected and the rest is absorbed
- The reflected light is scattered in a manner that depends on the smoothness and orientation of the surface

Light Sources

General light sources are difficult to work with because we must integrate light coming from all points on the source

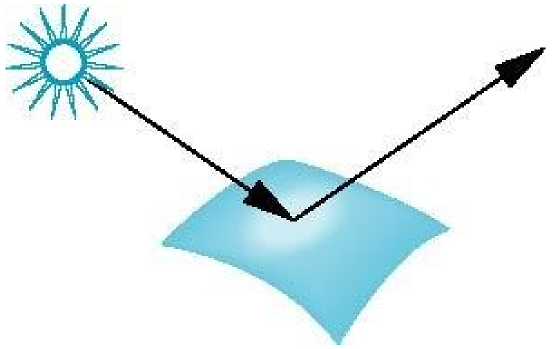


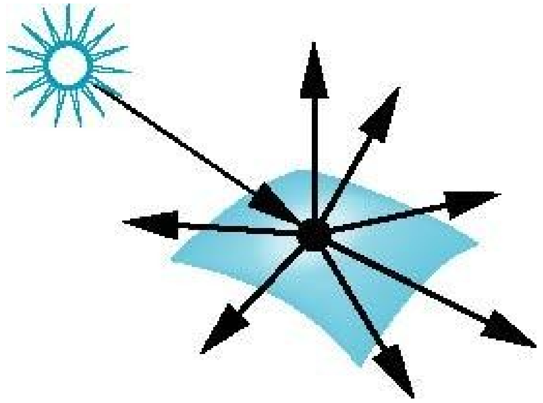
Simple Light Sources

- Point source
  - Model with position and color
  - Distant source = infinite distance away (parallel)
- Spotlight
  - Restrict light from ideal point source
- Ambient light
  - Same amount of light everywhere in scene
  - Can model contribution of many sources and reflecting surfaces

Surface Types

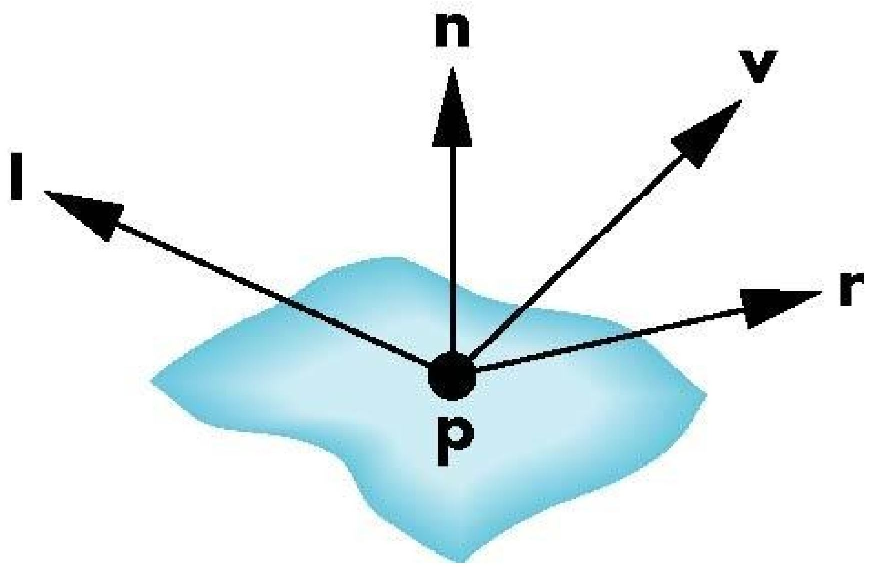
- The smoother a surface, the more reflected light is concentrated in the direction a perfect mirror would reflected the light
- A very rough surface scatters light in all directions


smooth surface


rough surface

Phong Model

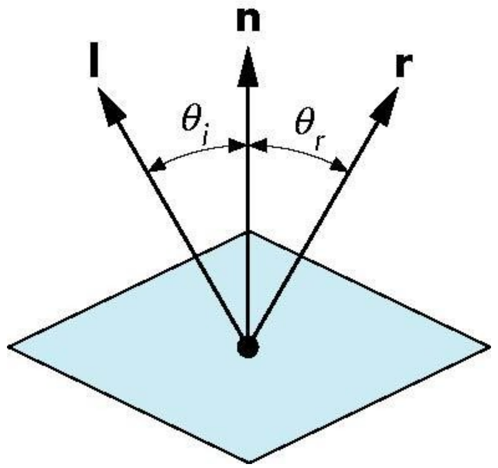
- A simple model that can be computed rapidly
- Has three components
  - Diffuse
  - Specular
  - Ambient
- Uses four vectors
  - To source
  - To viewer
  - Normal
  - Perfect reflector



Ideal Reflector

- Normal is determined by local orientation
- Angle of incidence = angle of reflection
- The three vectors must be coplanar

$$\mathbf{r} = 2\,(\mathbf{l} \cdot \mathbf{n})\,\mathbf{n} - \mathbf{l}$$

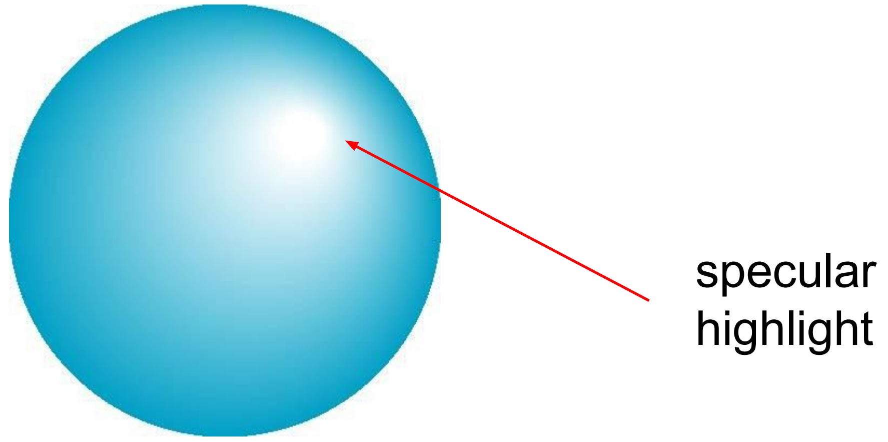


Lambertian Surface

- Perfectly diffuse reflector
- Light scattered equally in all directions
- Amount of light reflected is proportional to the vertical component of incoming light
  - reflected light ~$\cos\theta_i$
  - $\cos\theta_i$ = **l** · **n** if vectors normalized
  - There are also three coefficients, k_r, k_b, k_g that show how much of each color component is reflected

Specular Surfaces

- Most surfaces are neither ideal diffusers nor perfectly specular (ideal reflectors)
- Smooth surfaces show specular highlights due to incoming light being reflected in directions concentrated close to the direction of a perfect reflection



Modeling Specular Reflections

Phong proposed using a term that dropped off as the angle between the viewer and the ideal reflection increased

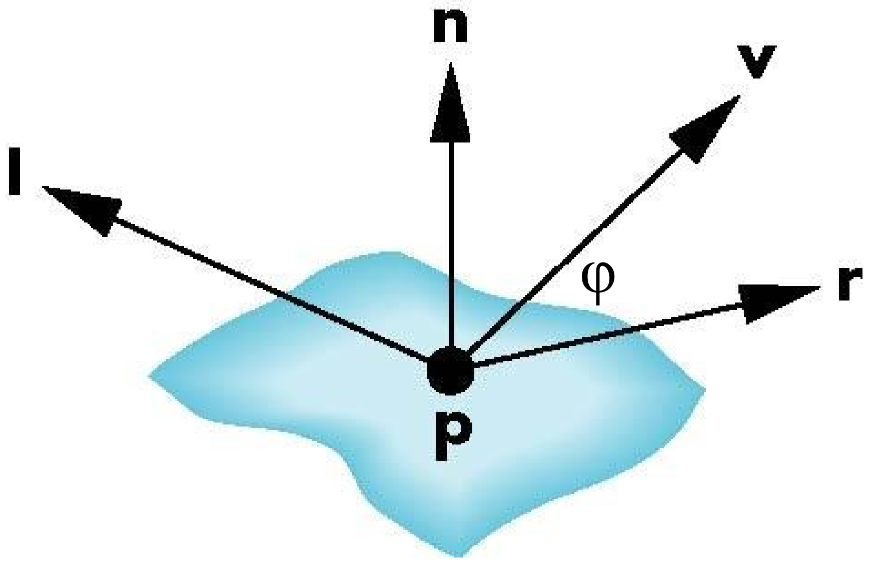
$$\mathbf{I}_r \sim k_s \, \mathbf{I} \cos^{\alpha}\varphi$$

reflected intensity

absorption coef

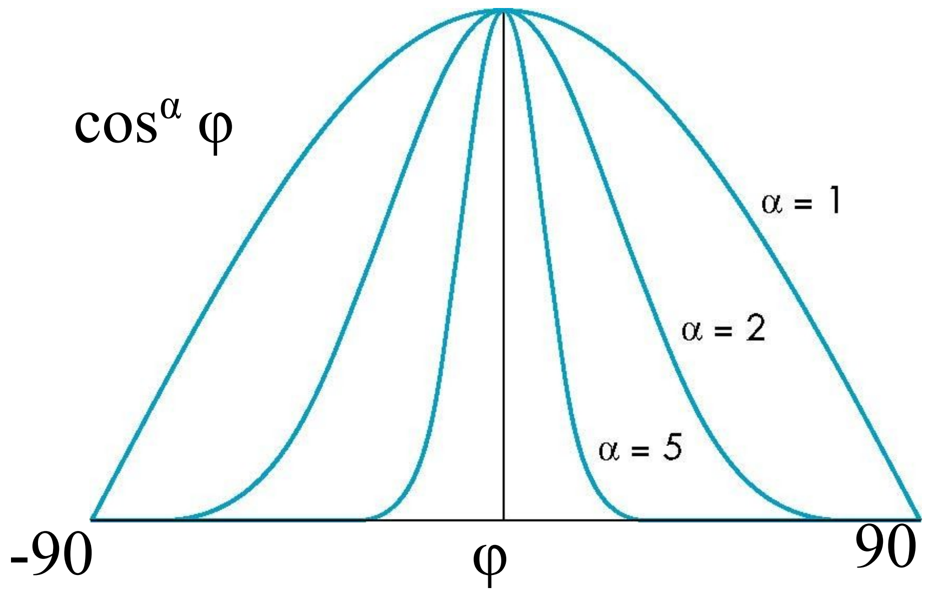
incoming intensity

shininess coef



The Shininess Coefficient

- Values of α between 100 and 200 correspond to metals
- Values between 5 and 10 give surface that look like plastic



Lighting and Shading II

Objectives

- Continue discussion of shading
- Introduce modified Phong model
- Consider computation of required vectors

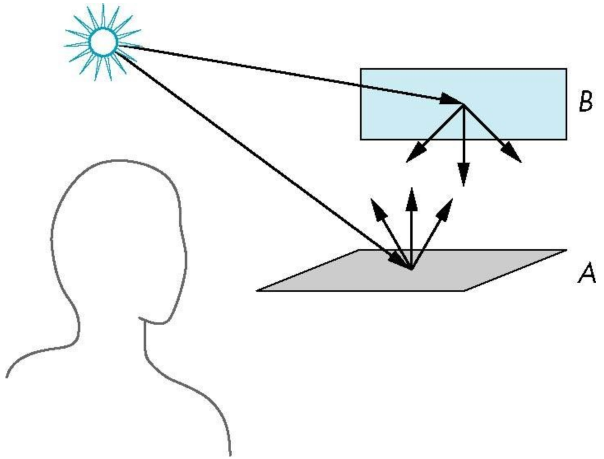
Ambient Light

- Ambient light is the result of multiple interactions between (large) light sources and the objects in the environment
- Amount and color depend on both the color of the light(s) and the material properties of the object
- Add $k_a \, I_a$ to diffuse and specular terms

reflection coef intensity of ambient light

Distance Terms

- The light from a point source that reaches a surface is inversely proportional to the square of the distance between them
- We can add a factor of the form $1/(a + bd + cd^2)$ to the diffuse and specular terms
- The constant and linear terms soften the effect of the point source



Light Sources

- In the Phong Model, we add the results from each light source
- Each light source has separate diffuse, specular, and ambient terms to allow for maximum flexibility even though this form does not have a physical justification
- Separate red, green and blue components
- Hence, 9 coefficients for each point source:

$$I_{dr}, I_{dg}, I_{db}, I_{sr}, I_{sg}, I_{sb}, I_{ar}, I_{ag}, I_{ab}$$

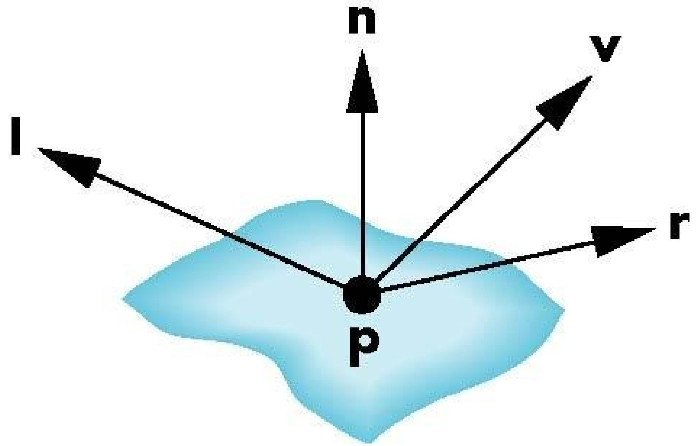
Material Properties

- Material properties match light source properties
  - Nine absorption coefficients
    - k_{dr}, k_{dg}, k_{db}, k_{sr}, k_{sg}, k_{sb}, k_{ar}, k_{ag}, k_{ab}
  - Shininess coefficient α

Adding up the Components

For each light source and each color component, the Phong model can be written (without the distance terms) as

$$I = k_d I_d \, \mathbf{l} \cdot \mathbf{n} + k_s I_s (\mathbf{v} \cdot \mathbf{r})^{\alpha} + k_a I_a$$



For each color component we add contributions from all sources

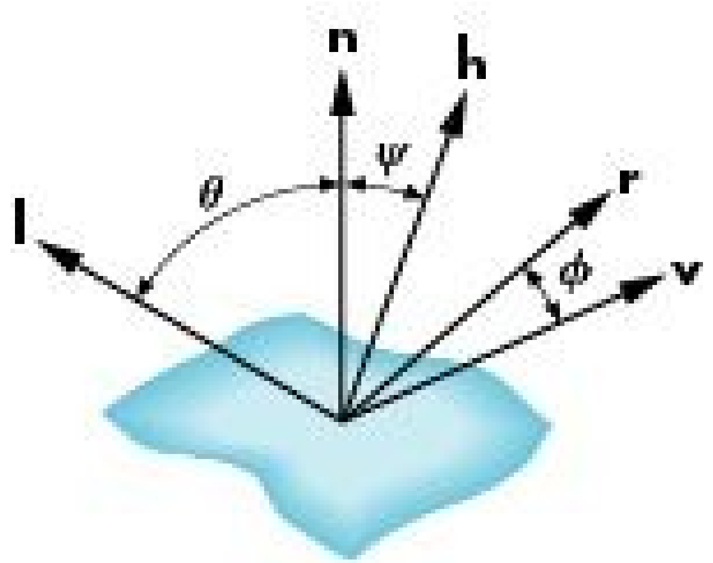
Modified Phong Model

- The specular term in the Phong model is problematic because it requires the calculation of a new reflection vector and view vector for each vertex
- **Blinn** suggested an approximation using the halfway vector that is more efficient

The Halfway Vector

h is normalized vector halfway between **l** and **v**

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



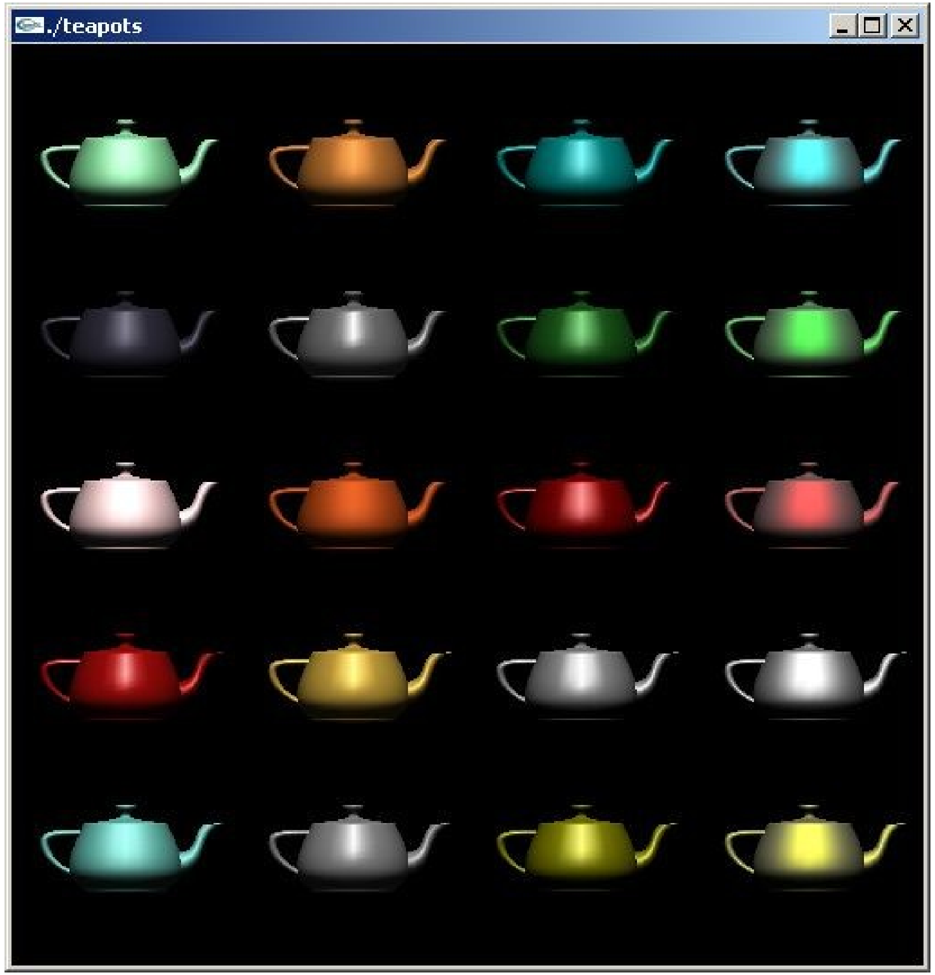
Using the halfway vector

- Replace $(\mathbf{v} \cdot \mathbf{r})^{\alpha}$ by $(\mathbf{n} \cdot \mathbf{h})^{\beta}$
- β is chosen to match shininess
- Note that halfway angle is half of angle between **r** and **v** if vectors are coplanar
- Resulting model is known as the *modified* Phong or Phong-Blinn lighting model

Specified in OpenGL standard

Example

Only differences in these teapots are the parameters in the modified Phong model



Computation of Vectors

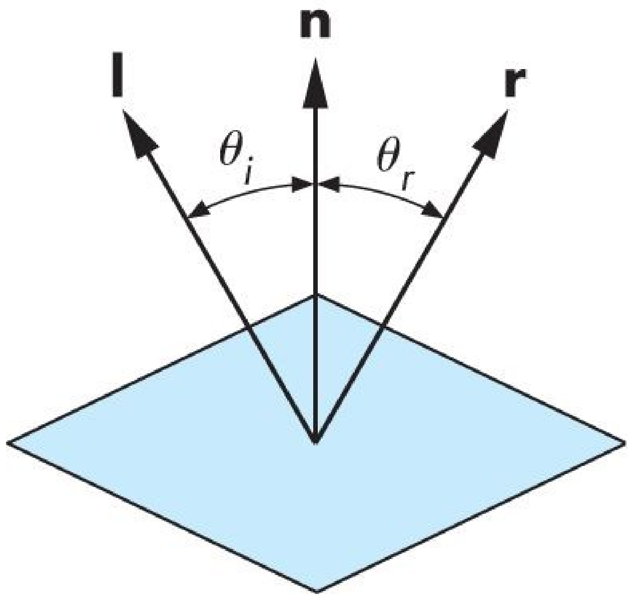
- **l** and **v** are specified by the application
- Can compute **r** from **l** and **n**
- Problem is determining **n**
- For simple surfaces can be determined but how we determine **n** differs depending on underlying representation of surface
- OpenGL leaves determination of normal to *application*

  Exception for GLU quadrics and Bezier surfaces was deprecated

Computing Reflection Direction

- Angle of incidence = angle of reflection
- Normal, light direction and reflection direction are coplanar
- Want all three to be unit length

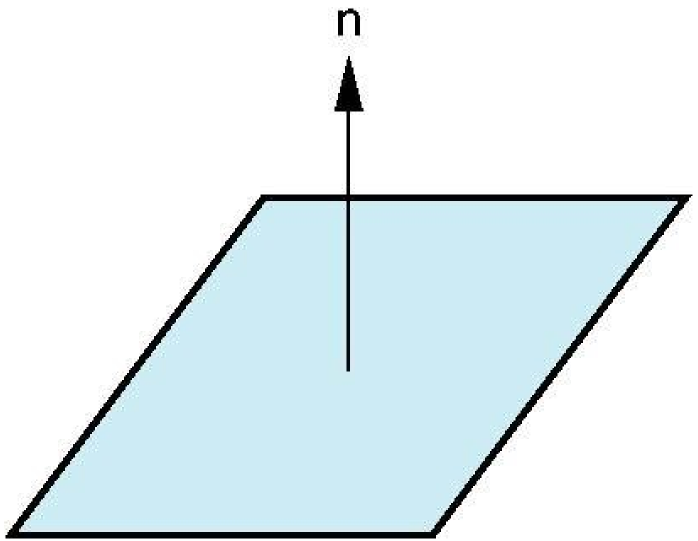
$$\boldsymbol{r} = 2\,(\boldsymbol{l} \cdot \boldsymbol{n})\,\boldsymbol{n} - \boldsymbol{l}$$



Plane Normals

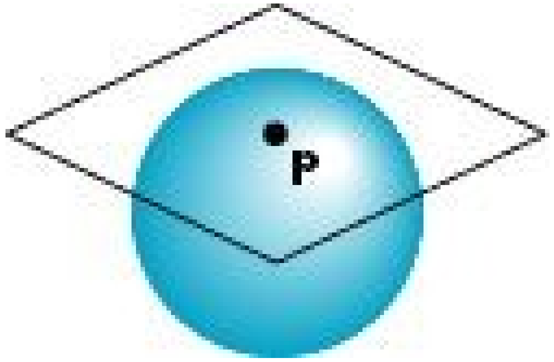
- Equation of plane: ax+by+cz+d = 0
- From Chapter 4 we know that plane is determined by three points p_0, p_2, p_3 or normal **n** and p_0
- Normal can be obtained by

$$\mathbf{n} = (p_2 - p_0) \times (p_1 - p_0)$$



Normal to Sphere

- Implicit function $f(x,y,z)=0$
- Normal given by gradient
- Sphere $f(\mathbf{p})=\mathbf{p}\cdot\mathbf{p}-1$
- $n = [\partial f/\partial x, \partial f/\partial y, \partial f/\partial z]^T=\mathbf{p}$



Parametric Form

- For sphere

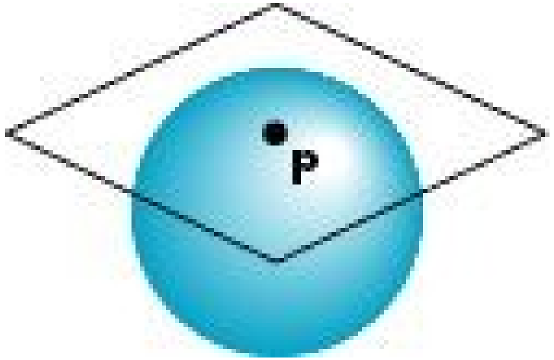
$$x=x(u,v)=\cos u \sin v$$
$$y=y(u,v)=\cos u \cos v$$
$$z=z(u,v)=\sin u$$

- Tangent plane determined by vectors

$$\partial \mathbf{p}/\partial u = [\partial x/\partial u, \partial y/\partial u, \partial z/\partial u]^T$$
$$\partial \mathbf{p}/\partial v = [\partial x/\partial v, \partial y/\partial v, \partial z/\partial v]^T$$

- Normal given by cross product

$$\mathbf{n} = \partial \mathbf{p}/\partial u \times \partial \mathbf{p}/\partial v$$



General Case

- We can compute parametric normals for other simple cases
    - Quadrics
    - Parametric polynomial surfaces
        - Bezier surface patches (Chapter 11)

Lighting and Shading in WebGL

Sorin Babii, Călin Popa
sorin.babii@cs.upt.ro, calin.popa@cs.upt.ro

Objectives

- Introduce the WebGL shading methods
  - Light and material functions on MV.js
  - Comparison: per vertex vs per fragment shading
  - Where to carry out

WebGL lighting

- Need
  - Normals
  - Material properties
  - Lights
- State-based shading functions have been deprecated (`glNormal`, `glMaterial`, `glLight`)
- Compute in application or in shaders

Normalization

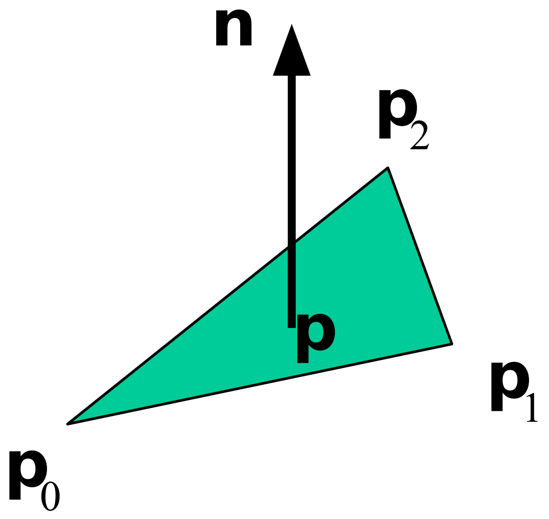
- Cosine terms in lighting calculations can be computed using dot product
- Unit length vectors simplify calculation
- Usually we want to set the magnitudes to have unit length but
  - Length can be affected by transformations
  - Note that scaling *does not preserve length*
- GLSL has a normalization function

Normal for Triangle

plane $\mathbf{n} \cdot (\mathbf{p} - \mathbf{p}_0) = 0$

$\mathbf{n} = (\mathbf{p}_2 - \mathbf{p}_0) \times (\mathbf{p}_1 - \mathbf{p}_0)$

normalize $\mathbf{n} \leftarrow \mathbf{n} / |\mathbf{n}|$



Note that right-hand rule determines outward face

Specifying a Point Light Source

For each light source, we can set an RGBA for the diffuse, specular, and ambient components, and for the position

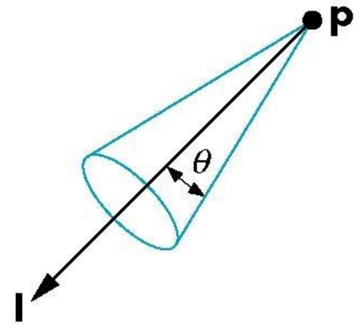
```
var diffuse0 = vec4(1.0, 0.0, 0.0, 1.0);
var ambient0 = vec4(1.0, 0.0, 0.0, 1.0);
var specular0 = vec4(1.0, 0.0, 0.0, 1.0);
var light0_pos = vec4(1.0, 2.0, 3,0, 1.0);
```

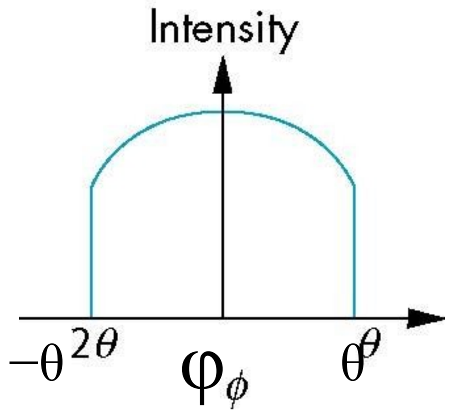
Distance and Direction

- The source colors are specified in RGBA
- The position is given in homogeneous coordinates
    - If w =1.0, we are specifying a finite location
    - If w =0.0, we are specifying a parallel source with the given direction vector
- The coefficients in distance terms are usually quadratic (1/(a+b*d+c*d*d)) where d is the distance from the point being rendered to the light source

Spotlights

- Derive from point source
  - Direction
  - Cutoff
  - Attenuation - proportional to $\cos^{\alpha}\varphi$





Global Ambient Light

- Ambient light depends on color of light sources

    A red light in a white room will cause a red ambient term that disappears when the light is turned off

- A global ambient term that is often helpful for testing

Moving Light Sources

- Light sources are geometric objects whose positions or directions are affected by the model-view matrix
- Depending on where we place the position (direction) setting function, we can
    - Move the light source(s) with the object(s)
    - Fix the object(s) and move the light source(s)
    - Fix the light source(s) and move the object(s)
    - Move the light source(s) and object(s) independently

Light Properties

```
var lightPosition = vec4(1.0, 1.0, 1.0, 0.0 );
var lightAmbient = vec4(0.2, 0.2, 0.2, 1.0 );
var lightDiffuse = vec4( 1.0, 1.0, 1.0, 1.0 );
var lightSpecular = vec4( 1.0, 1.0, 1.0, 1.0 );
```

Material Properties

- Material properties should match the terms in the light model
- Reflectivities
- w component gives opacity

```
var materialAmbient = vec4( 1.0, 0.0, 1.0, 1.0 );
var materialDiffuse = vec4( 1.0, 0.8, 0.0, 1.0);
var materialSpecular = vec4( 1.0, 0.8, 0.0, 1.0 );
var materialShininess = 100.0;
```

Using MV.js for Products

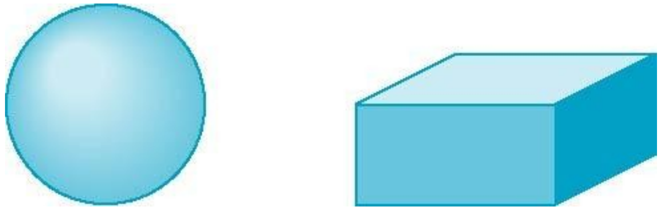
```
var ambientProduct = mult(lightAmbient, materialAmbient);
var diffuseProduct = mult(lightDiffuse, materialDiffuse);
var specularProduct = mult(lightSpecular, materialSpecular);

 gl.uniform4fv(gl.getUniformLocation(program,
                 "ambientProduct"),       flatten(ambientProduct));
 gl.uniform4fv(gl.getUniformLocation(program,
                 "diffuseProduct"),       flatten(diffuseProduct) );
 gl.uniform4fv(gl.getUniformLocation(program,
                  "specularProduct"),        flatten(specularProduct) );
 gl.uniform4fv(gl.getUniformLocation(program,
                  "lightPosition"),        flatten(lightPosition) );
 gl.uniform1f(gl.getUniformLocation(program,
                 "shininess"),materialShininess);
```

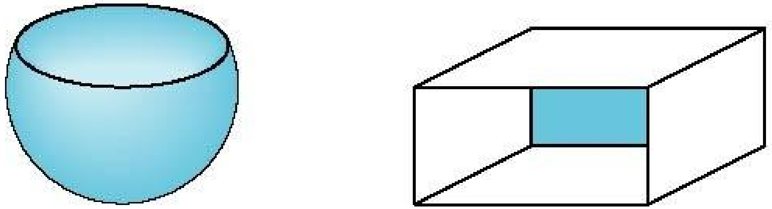
Adding Normals for Quads

```
function quad(a, b, c, d) {
     var t1 = subtract(vertices[b], vertices[a]);
     var t2 = subtract(vertices[c], vertices[b]);
     var normal = cross(t1, t2); // vec4
     var normal = vec3(normal);
     normal = normalize(normal);

     pointsArray.push(vertices[a]);
     normalsArray.push(normal);
         .
         .
         .
```

Front and Back Faces

- Every face has a front and back
- For many objects, we never see the back face so we don't care how or if it's rendered
- If it matters, we can handle in shader


back faces not visible


back faces visible

Emissive Term

- We can simulate a light source in WebGL by giving a material an emissive component
- This component is unaffected by any sources or transformations

Transparency

- Material properties are specified as RGBA values
- The A value can be used to make the surface translucent
- The default is that all surfaces are opaque
- Later we will enable blending and use this feature
- However with the HTML5 canvas, A<1 will mute colors

Video

http://www.cs.upt.ro/~sorin/webgl/Code/w06/shadedCube.html

http://www.cs.upt.ro/~sorin/webgl/Code/w06/shadedSphere1.html

Shading in vertex shader vs fragment shader will be discussed later

Polygonal Shading

Sorin Babii, Călin Popa
sorin.babii@cs.upt.ro, calin.popa@cs.upt.ro

Polygonal Shading

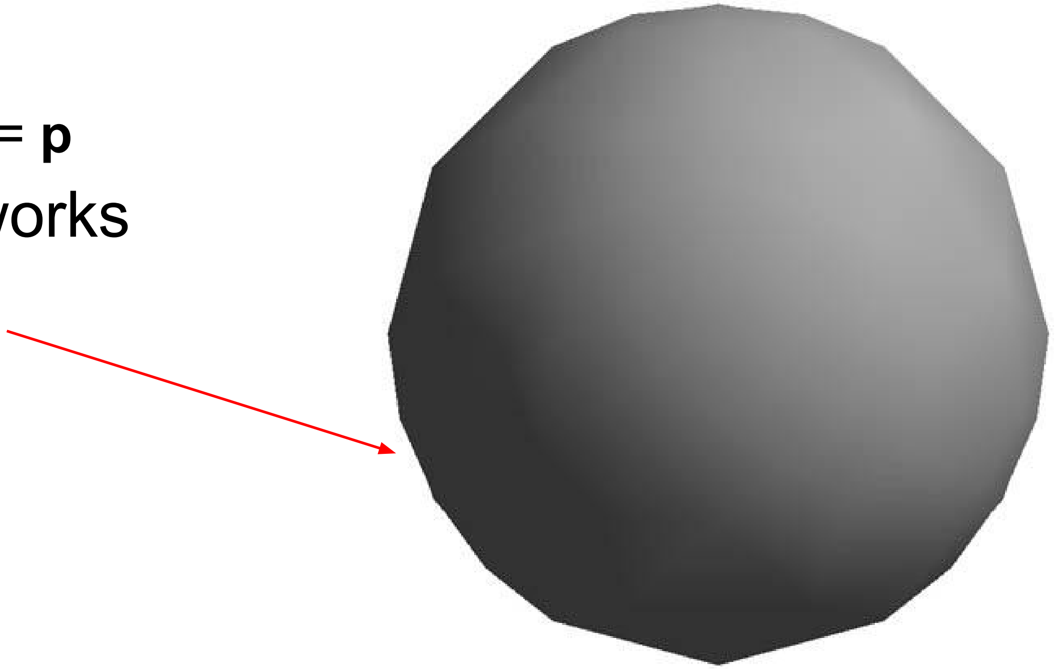
- In per vertex shading, shading calculations are done for each vertex
  - Vertex colors become *vertex shades* and can be sent to the vertex shader as a *vertex attribute*
  - Alternately, we can *send the parameters* to the vertex shader and have it *compute the shade*
- By default, vertex shades are interpolated across an object <u>if passed to the fragment shader</u> as a varying variable (smooth shading)
- Use uniform variables to shade with a single shade (flat shading)

Polygon Normals

- Triangles have a single normal
- Shades at the vertices as computed by the modified Phong model can be almost same
- Identical for a distant viewer (default) or if there is no specular component
- Consider model of sphere
- Want *different normals* at each vertex even though this concept is not quite correct mathematically

Smooth Shading

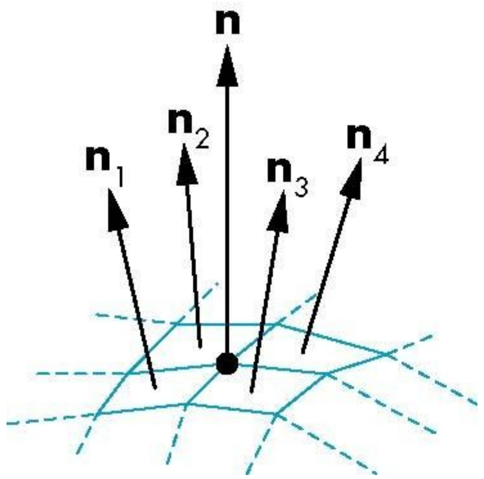
- We can set a new normal at each vertex
- Easy for sphere model
  - If centered at origin **n** = **p**
- Now smooth shading works
- Note *silhouette edge*



Mesh Shading

- The previous example is not general because we knew the normal at each vertex analytically
- For polygonal models, Gouraud proposed we use the *average* of the normals *around* a mesh vertex

$$\mathbf{n} = (\mathbf{n}_1+\mathbf{n}_2+\mathbf{n}_3+\mathbf{n}_4) / |\mathbf{n}_1+\mathbf{n}_2+\mathbf{n}_3+\mathbf{n}_4|$$



Gouraud and Phong Shading

- Gouraud Shading
  - Find average normal at each vertex (vertex normals)
  - Apply modified Phong model at each vertex
  - Interpolate vertex shades across each polygon
- Phong shading
  - Find vertex normals
  - Interpolate vertex normals across edges
  - Interpolate edge normals across polygon
  - Apply modified Phong model at each fragment

Comparison

- If the polygon mesh approximates surfaces with a high curvatures, Phong shading may look smooth while Gouraud shading may show edges
- Phong shading requires much more work than Gouraud shading
  - Until recently not available in real time systems
  - Now can be done using fragment shaders
- Both need data structures to represent meshes so we can obtain vertex normals

Per Vertex and Per Fragment Shaders

Sorin Babii, Călin Popa
sorin.babii@cs.upt.ro, calin.popa@cs.upt.ro

Vertex Lighting Shaders I

```
// vertex shader

attribute vec4 vPosition;
attribute vec4 vNormal;
varying vec4 fColor;
uniform vec4 ambientProduct, diffuseProduct, specularProduct;
uniform mat4 modelViewMatrix;
uniform mat4 projectionMatrix;
uniform vec4 lightPosition;
uniform float shininess;

void main()
{
```

Vertex Lighting Shaders II

```
vec3 pos = -(modelViewMatrix * vPosition).xyz;
vec3 light = lightPosition.xyz;
vec3 L = normalize( light - pos );
vec3 E = normalize( -pos );
vec3 H = normalize( L + E );

// Transform vertex normal into eye coordinates

   vec3 N = normalize( (modelViewMatrix*vNormal).xyz);

// Compute terms in the illumination equation
```

Vertex Lighting Shaders III

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

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

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

Vertex Lighting Shaders IV

```
// fragment shader

precision mediump float;

varying vec4 fColor;

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

Fragment Lighting Shaders I

```
// vertex shader

attribute vec4 vPosition;
attribute vec4 vNormal;
varying vec3 N, L, E;
uniform mat4 modelViewMatrix;
uniform mat4 projectionMatrix;
uniform vec4 lightPosition;
```

Fragment Lighting Shaders II

```
// vertex shader…

void main()
{
    vec3 pos = -(modelViewMatrix * vPosition).xyz;
    vec3 light = lightPosition.xyz;
    L = normalize( light - pos );
    E =  -pos;
    N = normalize( (modelViewMatrix * vNormal).xyz);
    gl_Position = projectionMatrix * modelViewMatrix * vPosition;
}
```

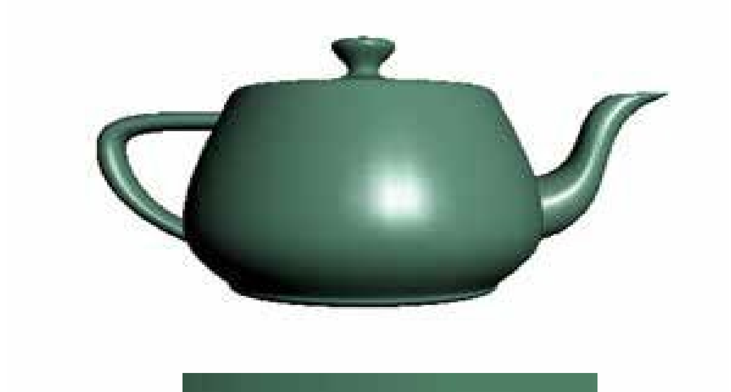
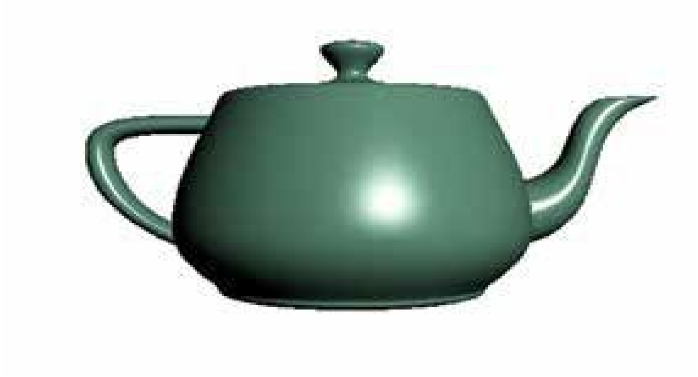
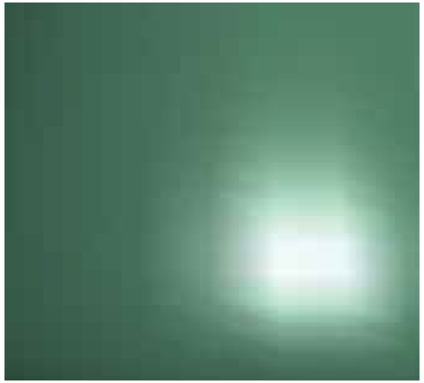
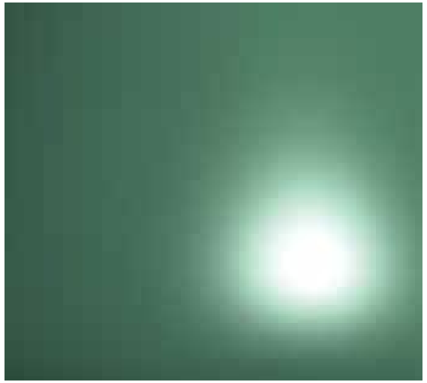
Fragment Lighting Shaders III

```
// fragment shader

precision mediump float;

uniform vec4 ambientProduct;
uniform vec4 diffuseProduct;
uniform vec4 specularProduct;
uniform float shininess;
varying vec3 N, L, E;

void main()
{
```

Fragment Lighting Shaders IV

```
    vec4 fColor;
    vec3 H = normalize( L + E );
    vec4 ambient = ambientProduct;
    float Kd = max( dot(L, N), 0.0 );
    vec4  diffuse = Kd * diffuseProduct;
    float Ks = pow( max(dot(N, H), 0.0), shininess );
    vec4  specular = Ks * specularProduct;
    if( dot(L, N) < 0.0 ) specular = vec4(0.0, 0.0, 0.0, 1.0);
    fColor = ambient + diffuse +specular;
    fColor.a = 1.0;
    gl_FragColor = fColor;
}
```

Teapot Examples

Video

http://www.cs.upt.ro/~sorin/webgl/Code/w06/shadedSphere2.html

vertex shading

http://www.cs.upt.ro/~sorin/webgl/Code/w06/shadedSphere4.html

fragment shading