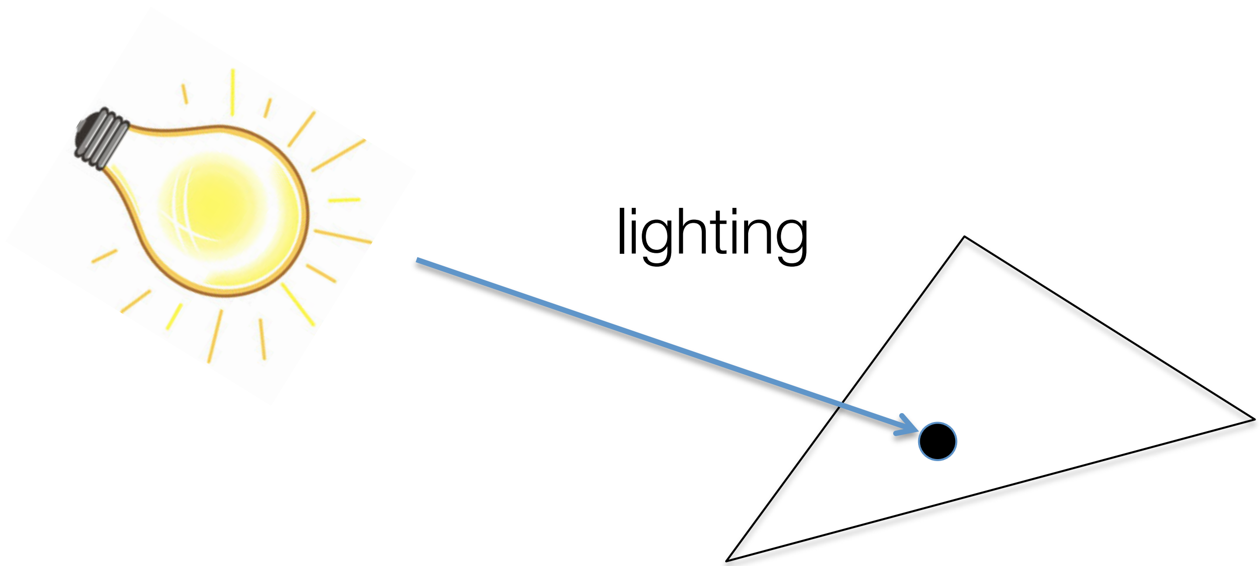
Learning Objectives

- Students completing this lecture will be able to
  - Explain the following terms: ambient, diffuse, specular lights; halfway vector
  - Explain the difference between local vs. global illumination, flat vs. smooth shading, Gouraud (per-vertex) vs. Phong (per-fragment) lighting
  - Derive the equation for Phong lighting model
  - Write WebGL code to implement lighting

Lighting and Shading

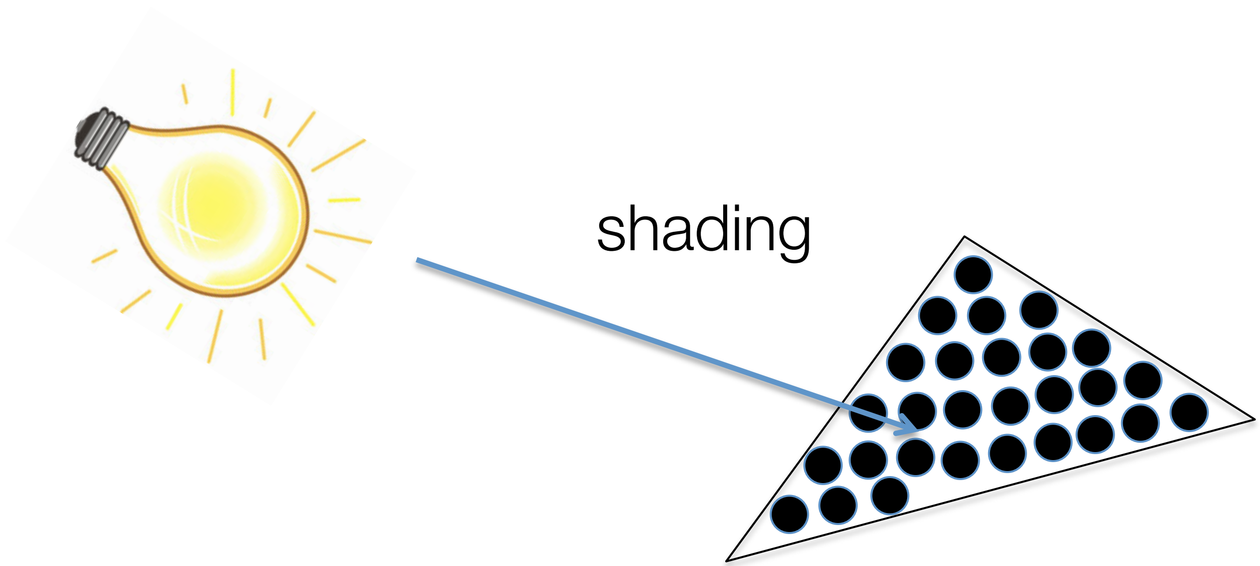
Illumination (Lighting)

- Model the interaction of light with surface points to determine their final color and brightness

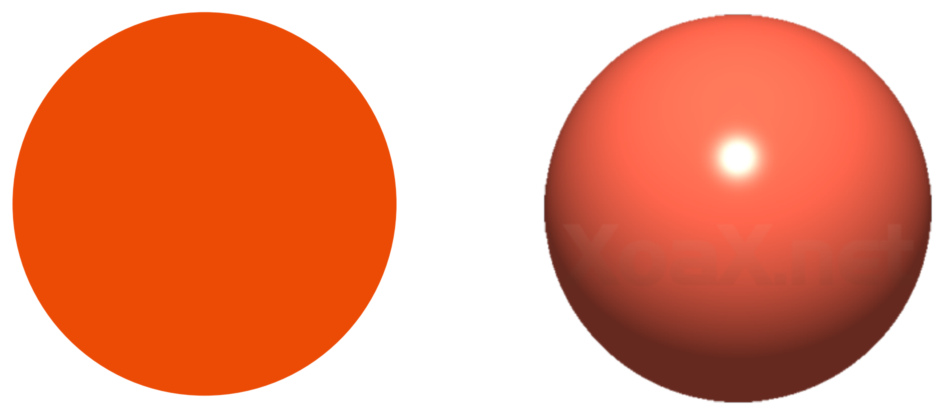


Shading

- Apply the lighting model at a set of points across the entire surface



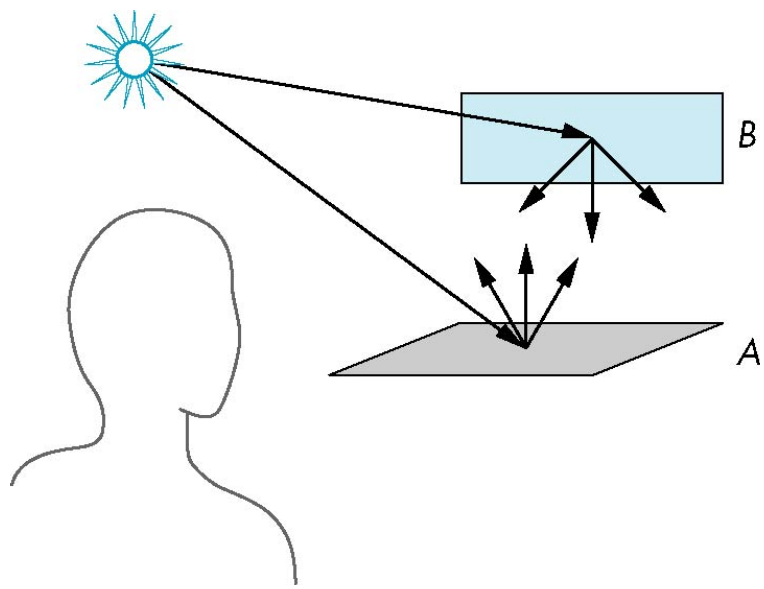
Why we need shading



- 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 (real scenario)

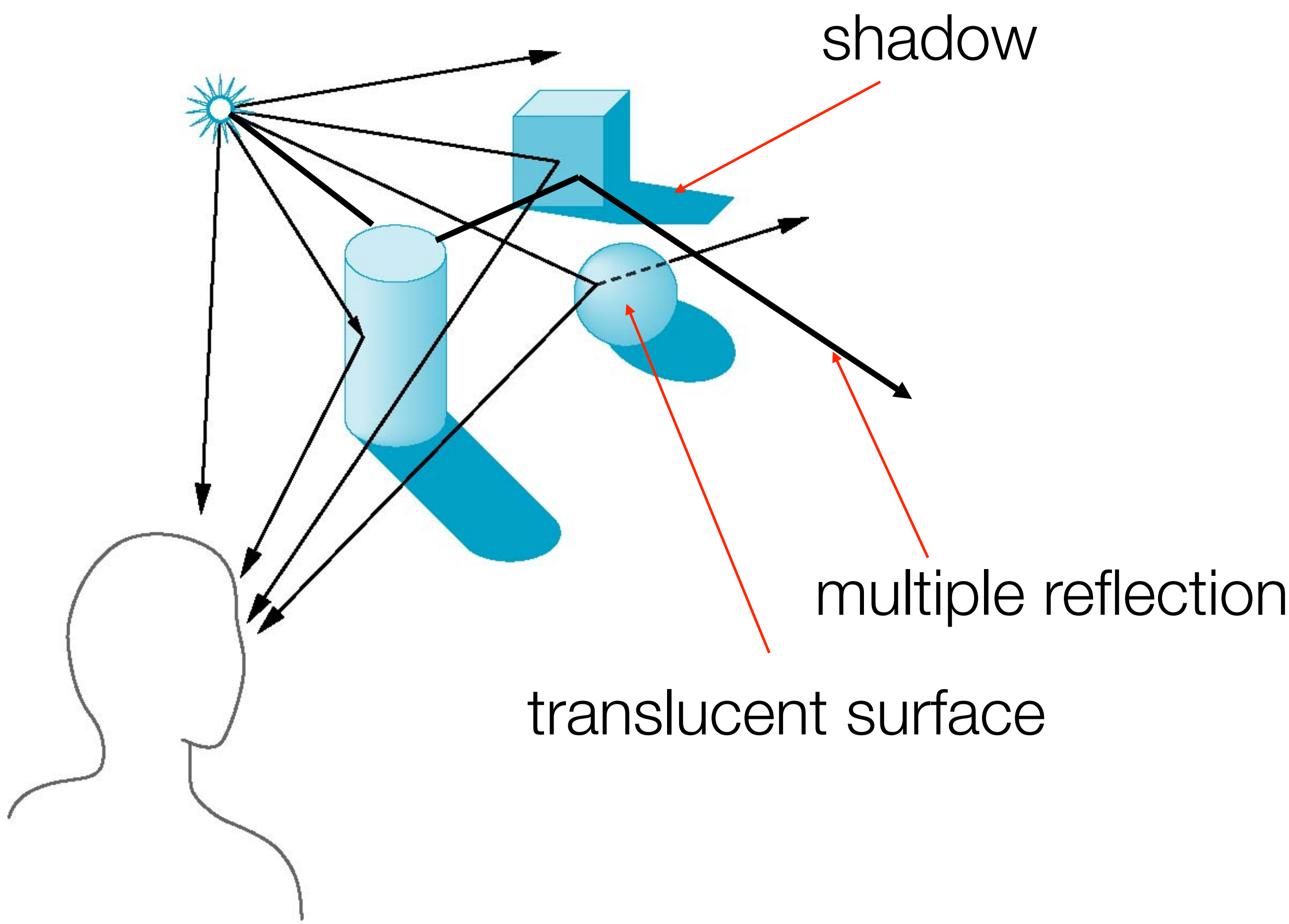
- Light strikes A
  - Some scattered
  - Some absorbed
- Some of scattered light strikes B
  - Some scattered
  - Some absorbed
- Some of this scattered light strikes A and so on



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 Illumination

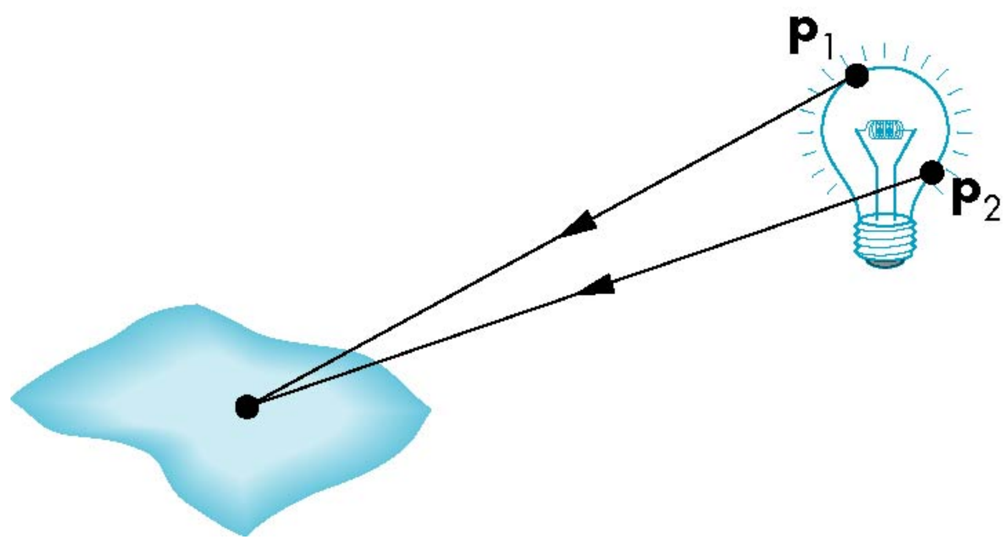
- Correct shading requires a global calculation involving all objects and light sources
  - Incompatible with pipeline model which shades each polygon independently (local illumination)
- Local illumination only considers the light, the observer position, and the object material properties
- However, in computer graphics, especially real time graphics, we are happy if things “look right”

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



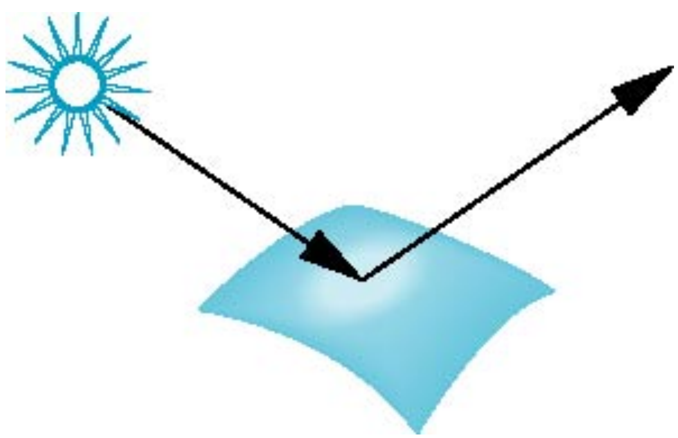
Simple Light Sources

- 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 the scene
  - Can model contribution of many sources and reflecting surfaces

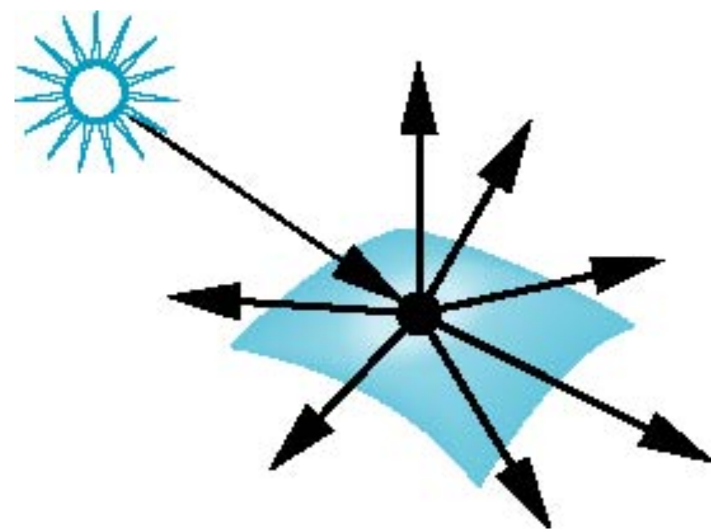
Surface Types

- 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

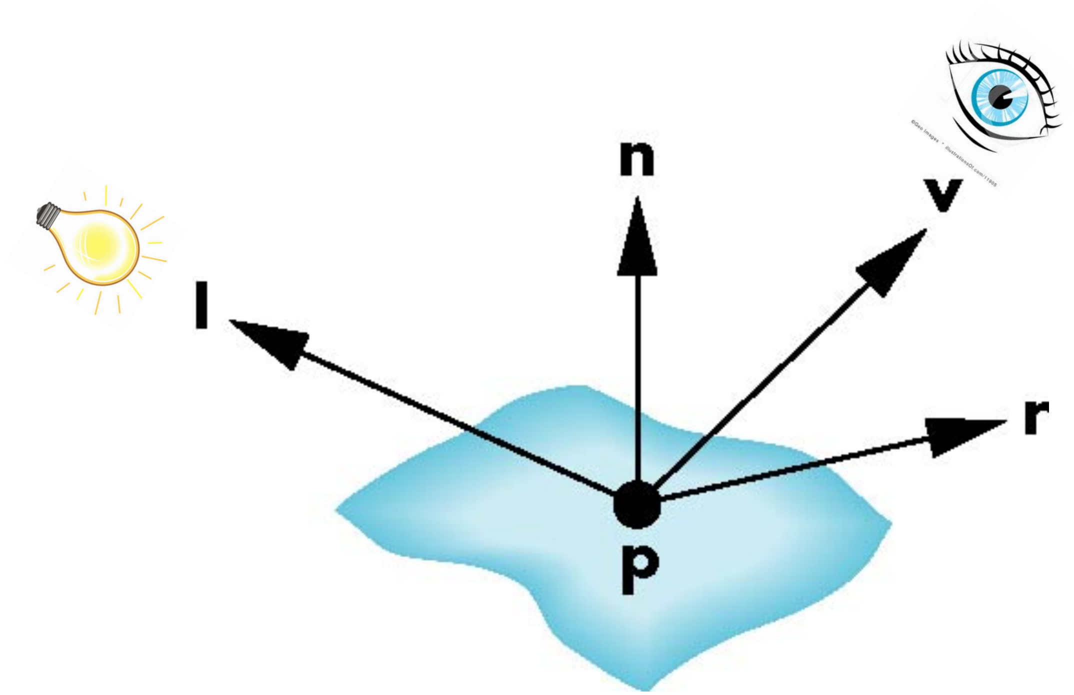


rough surface



Phong Model

- A simple model that can be computed rapidly
- Has three components
  - Diffuse
  - Specular
  - Ambient
- Uses four vectors
  - To source (**l**)
  - To viewer (**v**)
  - Normal (**n**)
  - Perfect reflector (**r**)



Ideal Reflector

- Normal is determined by local orientation
- Angle of incidence = angle of reflection: $\theta_i = \theta_r$
- The three vectors ($\mathbf{l}$, $\mathbf{n}$, and $\mathbf{r}$) must be coplanar
- Assume $\mathbf{l}$, $\mathbf{n}$, and $\mathbf{r}$ are all normalized

$$\cos\theta_i = \mathbf{l} \cdot \mathbf{n} = \mathbf{n} \cdot \mathbf{r} = \cos\theta_r$$

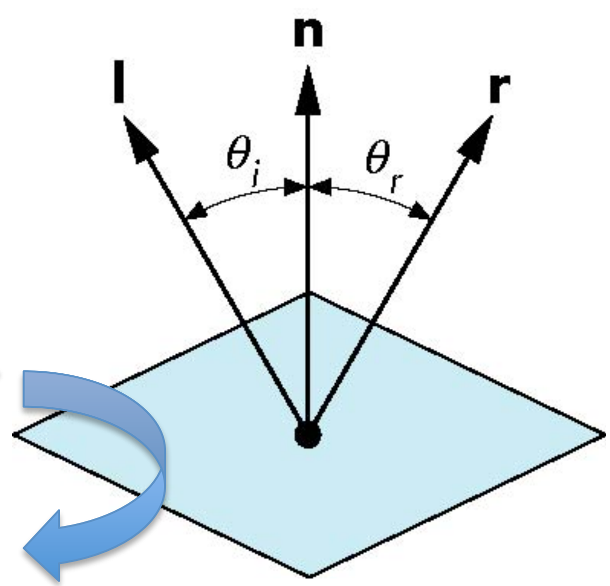
$$\mathbf{r} = \alpha\mathbf{l} + \beta\mathbf{n}$$

$$\mathbf{n} \cdot \mathbf{r} = \alpha\mathbf{l} \cdot \mathbf{n} + \beta = \mathbf{l} \cdot \mathbf{n}$$

$$1 = \mathbf{r} \cdot \mathbf{r} = \alpha^2 + 2\alpha\beta\mathbf{l} \cdot \mathbf{n} + \beta^2$$

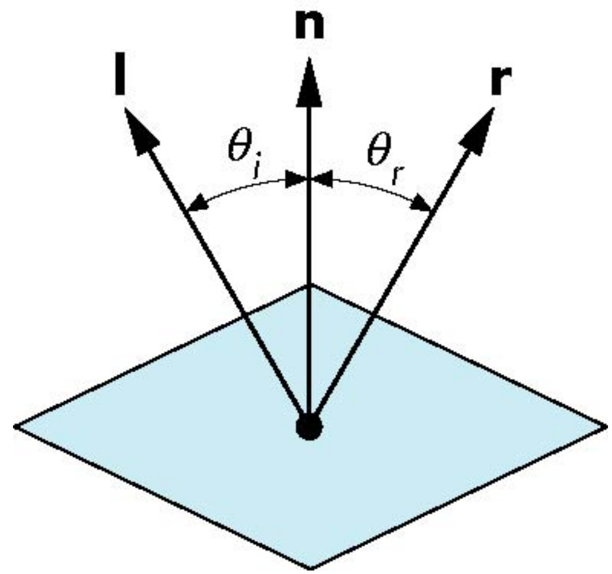
$$\mathbf{r} = 2\,(\mathbf{l} \cdot \mathbf{n})\,\mathbf{n} - \mathbf{l}$$

compute α and β



$$\mathbf{r} = \alpha\mathbf{l} + \beta\mathbf{n}$$

Ideal Reflector



$$\mathbf{n} \cdot \mathbf{r} = \alpha\mathbf{l} \cdot \mathbf{n} + \beta = \mathbf{l} \cdot \mathbf{n}$$

$$\beta = \mathbf{l} \cdot \mathbf{n} - \alpha\mathbf{l} \cdot \mathbf{n}$$

$$1 = \mathbf{r} \cdot \mathbf{r} = \alpha^2 + 2\alpha\beta\mathbf{l} \cdot \mathbf{n} + \beta^2$$

$$1 = \alpha^2 + 2\alpha(\mathbf{l} \cdot \mathbf{n} - \alpha\mathbf{l} \cdot \mathbf{n})\mathbf{l} \cdot \mathbf{n} + (\mathbf{l} \cdot \mathbf{n} - \alpha\mathbf{l} \cdot \mathbf{n})^2$$

$$1 = \alpha^2 + 2\alpha(\mathbf{l} \cdot \mathbf{n})^2 - 2\alpha^2(\mathbf{l} \cdot \mathbf{n})^2 + (\mathbf{l} \cdot \mathbf{n})^2 - 2\alpha(\mathbf{l} \cdot \mathbf{n})^2 + \alpha^2(\mathbf{l} \cdot \mathbf{n})^2$$

$$1 = \alpha^2 + (1 - (\mathbf{l} \cdot \mathbf{n})^2) + (\mathbf{l} \cdot \mathbf{n})^2$$

$$1 = \alpha^2$$

$$\cancel{\alpha = 1} \qquad \cancel{\beta = \mathbf{l} \cdot \mathbf{n} - \alpha\mathbf{l} \cdot \mathbf{n} = 0}$$

$$\alpha = -1 \qquad \beta = \mathbf{l} \cdot \mathbf{n} - \alpha\mathbf{l} \cdot \mathbf{n} = 2(\mathbf{l} \cdot \mathbf{n})$$

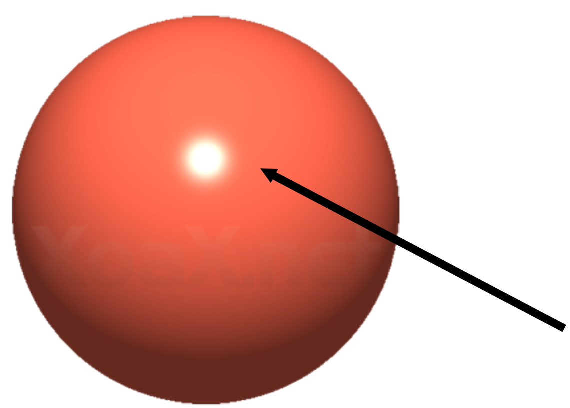
$$\mathbf{r} = 2\ (\mathbf{l} \cdot \mathbf{n})\ \mathbf{n} - \mathbf{l}$$

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 ~$\cos \theta_i$
  - $\cos \theta_i = \mathbf{l} \cdot \mathbf{n}$ if vectors normalized
  - There are also three coefficients, k_r, k_b, k_g that show how much of each color component (R, G, and B) 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



specular highlight

Modeling Specular Reflections

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

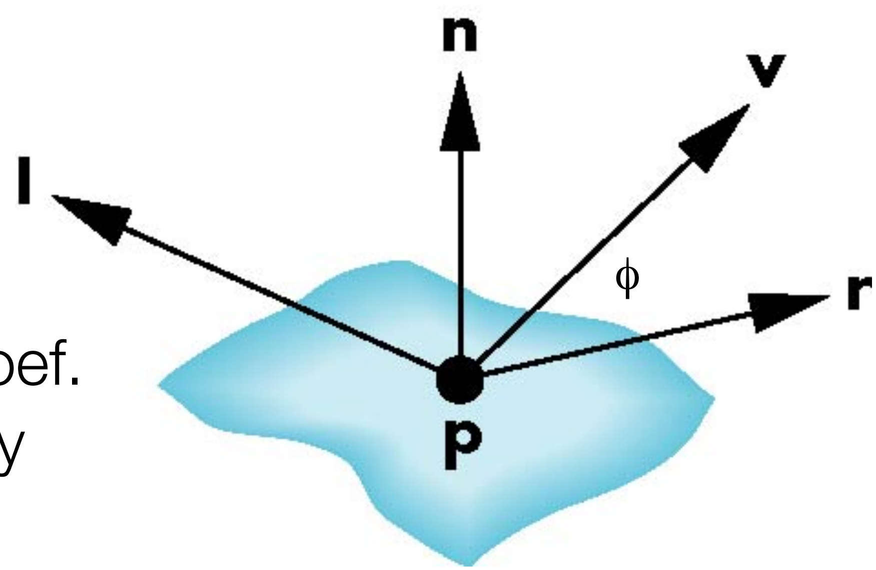
$$\mathbf{I}_r \sim k_s\ \mathbf{I}\ \cos^{\alpha}\phi$$

reflected intensity

absorption coef.

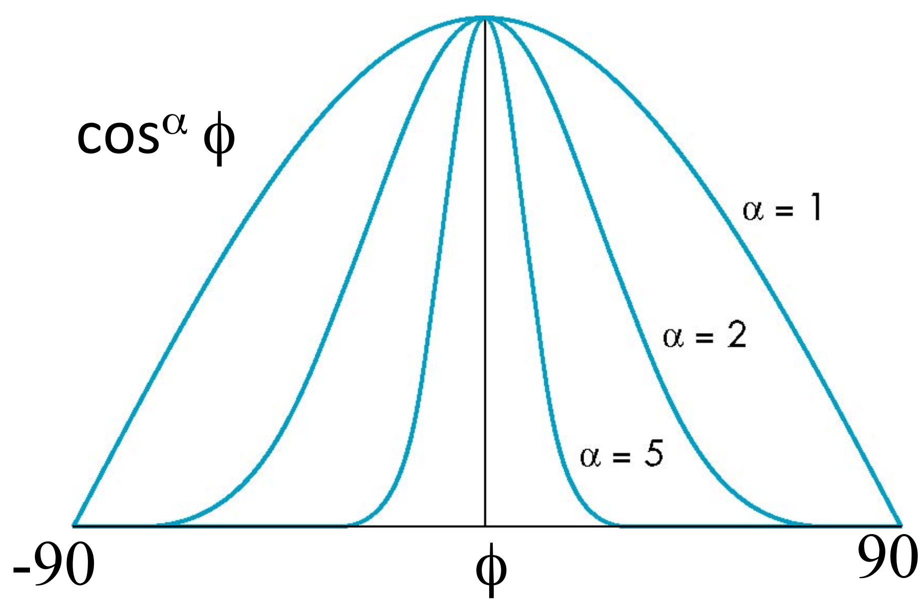
incoming intensity

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



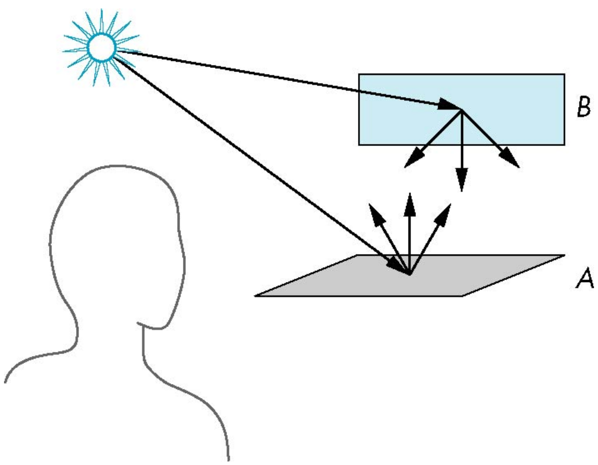
Ambient Light

- 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 $k_a \, \mathbf{I}_a$ 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 (a) and linear (bd) 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

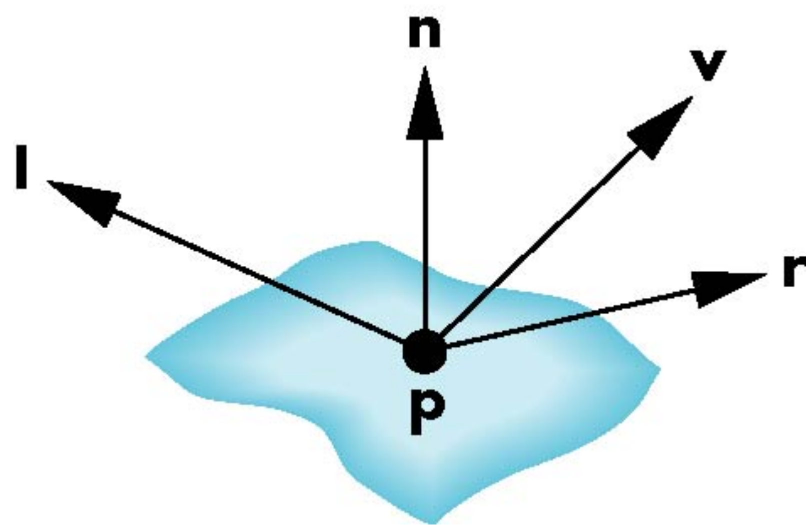
- Material properties match light source properties
  - –Nine absorbtion coefficients
    - $k_{dr}, k_{dg}, k_{db}; k_{sr}, k_{sg}, k_{sb}; k_{ar}, k_{ag}, k_{ab}$
  - –Shininess coefficient α

Adding up the Components

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

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

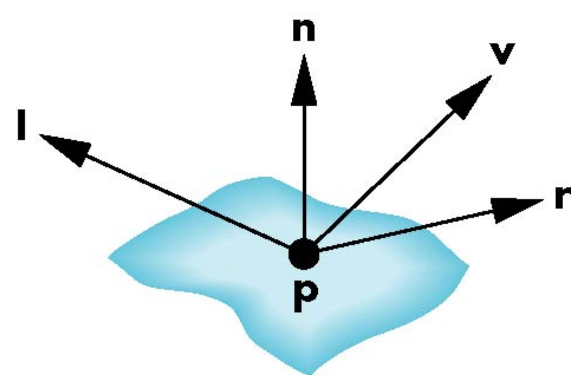
- For each color component we add contributions from all sources



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

Modified Phong Model

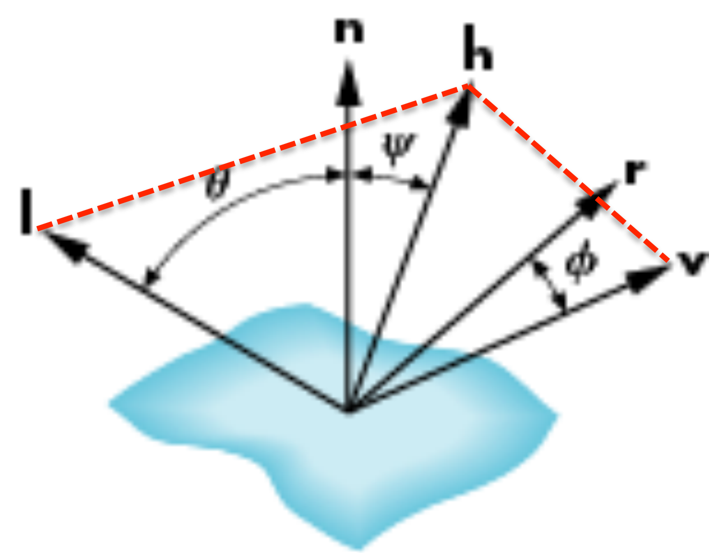
- The specular term in the Phong model is problematic because it requires the calculation of a new reflection vector (**r**) and view vector (**v**) for each vertex
- Blinn suggested an approximation using the halfway vector that is more efficient



The Halfway Vector

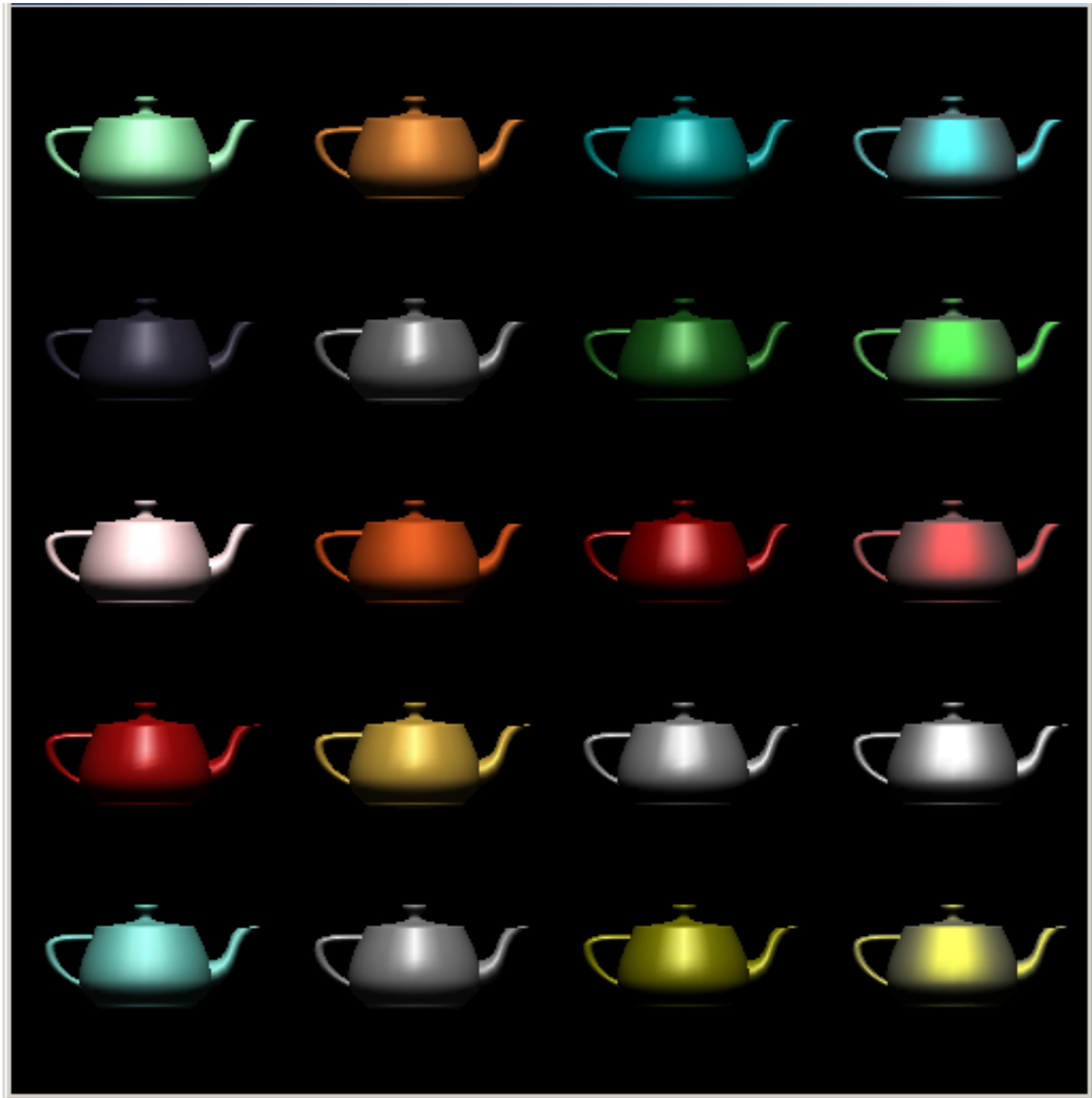
- **h** is normalized vector halfway between **l** and **v**

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



Using the Halfway Vector

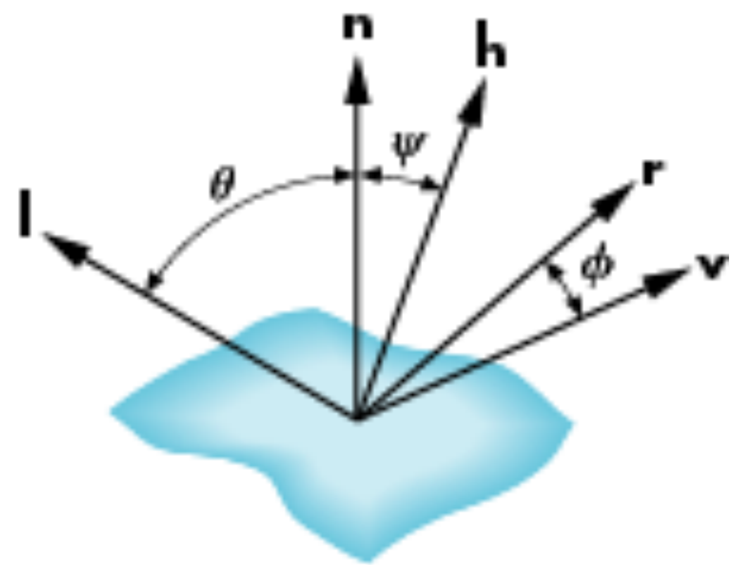
- Replace $(\mathbf{v} \cdot \mathbf{r})^{\alpha}$ by $(\mathbf{n} \cdot \mathbf{h})^{\beta}$
- β is chosen to match shineness
- Note that halfway angle (ψ) is half of angle between $\mathbf{r}$ and $\mathbf{v}$ (ϕ) if vectors are coplanar ($2\psi=\phi$)
- Resulting model is known as the modified Phong or Blinn lighting model
  - Specified in OpenGL standard



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 computer **r** from **l** and **n**
- Problem is determining **n**
- How we determine **n** differs depending on underlying representation of surface
- WebGL leaves determination of normal to application

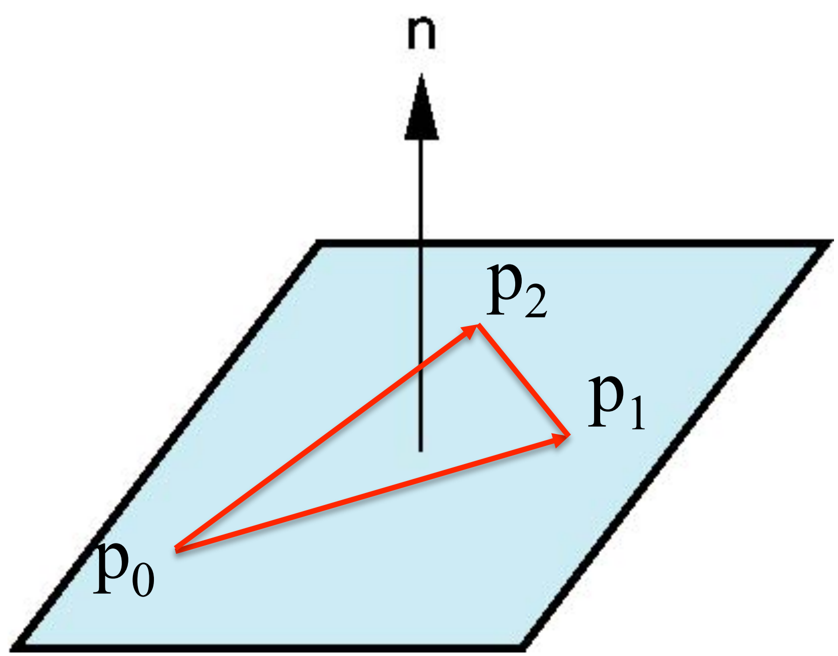


$$\mathbf{r} = 2\,(\mathbf{l} \cdot \mathbf{n})\,\mathbf{n} - \mathbf{l}$$

Plane Normals

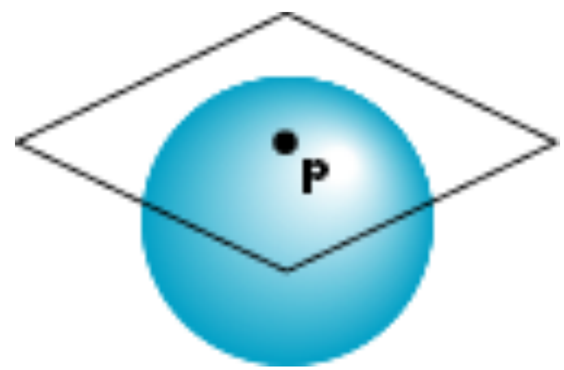
- Equation of plane: $ax + by + cz + d = \mathbf{0}$
- We know that plane is determined by three points p_0, p_1, p_2 or normal $\mathbf{n}$ and p_0
- Normal can be obtained by

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



Normal to Sphere

- Implicit function $f(x, y, z) = x^2+y^2+z^2-1 = 0$
- Normal given by gradient
- Sphere $f(\mathbf{p}) = \mathbf{p} \cdot \mathbf{p} - 1 = 0$
- $\mathbf{n} = [\partial f/\partial x, \partial f/\partial y, \partial f/\partial z]^T = [2x, 2y, 2z] = 2\mathbf{p}$
- The normal at every point in the surface of the sphere points directly out of the sphere



Lighting and Shading in WebGL

WebGL Lighting

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

Normalization

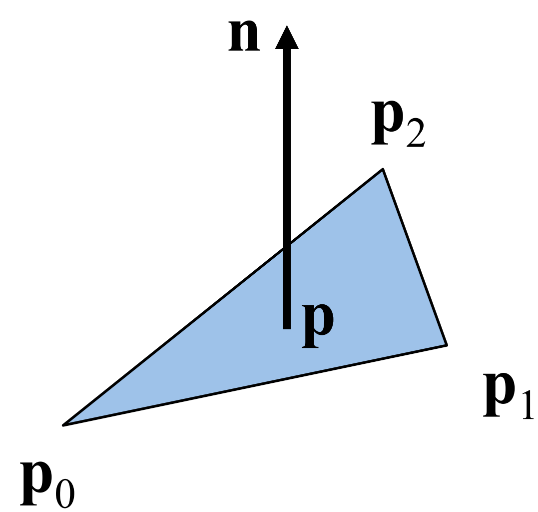
- Cosine terms in lighting calculations can be computed using dot product
- Unit length vectors simplify calculation
- Usually we want to set the magnitudes to have unit length but
  - Length can be affected by transformations
  - Note that scaling does not preserved length
- GLSL has a normalization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

Normal for Triangle

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

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

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

    ```
    vec4 light0_pos = vec4(1.0, 2.0, 3,0, 1.0);
    ```

  - If $w = 0.0$, we are specifying a parallel source with the given direction vector

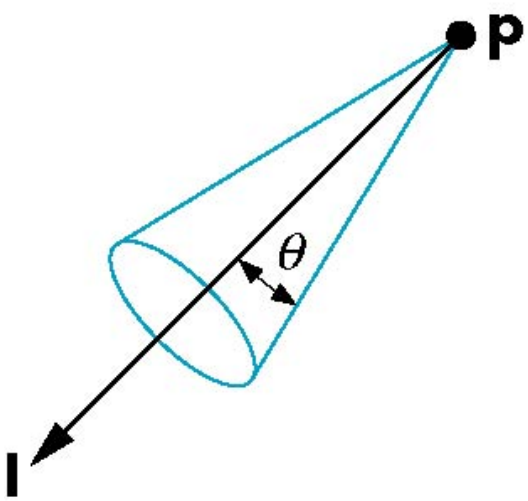
    ```
    vec4 light0_pos = vec4(1.0, 2.0, 3,0, 0.0);
    ```

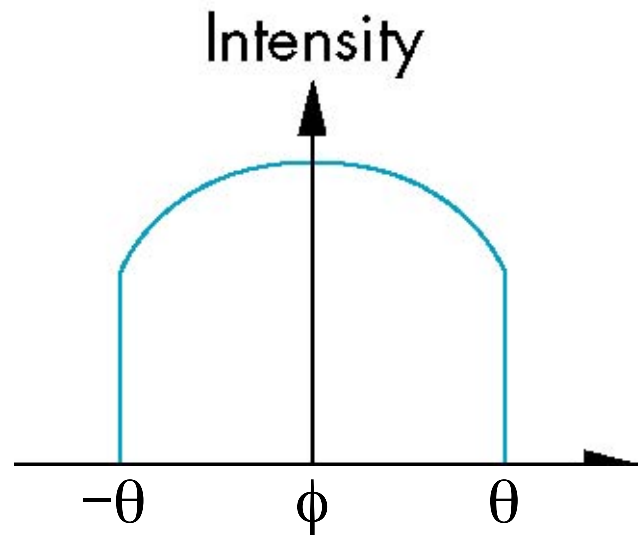
Distance and Direction

- The coefficients in the distance terms $1/(a + bd + cd^2)$ where d is the distance from the point being rendered to the light source
- Use three floats for these values (constant, linear and quadratic terms)

Spotlights

- Derived from point source
  - Direction (the direction of the light in homogeneous object coordinates)
  - Cutoff (the maximum spread angle of a light source)
  - Attenuation (Proportional to $\cos^{\alpha}\phi$)





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 modelview 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
- `w` component gives opacity

```
var materialAmbient = vec4(0.2, 0.2, 0.2, 1.0);
var materialDiffuse = vec4(1.0, 0.8, 0.0, 1.0);
var materialSpecular = vec4(1.0, 1.0, 1.0, 1.0);
var materialShininess = 100.0
```

Using `MV.js` for Products

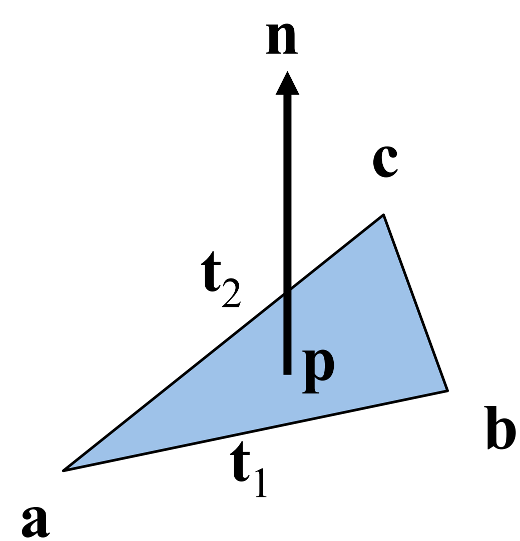
```
var ambientProduct = mult(lightAmbient, materialAmbient);
var diffuseProduct = mult(lightDiffuse, materialDiffuse);
var specularProduct = mult(lightSpecular, materialSpecular);

 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

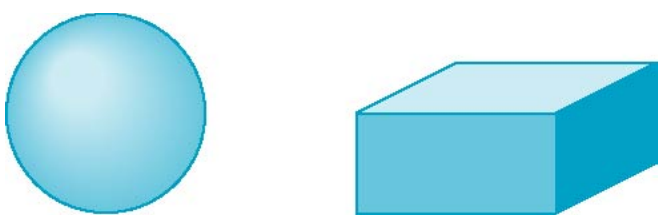
```
function quad(a, b, c, d) {
     var t1 = subtract(vertices[b], vertices[a]);
     var t2 = subtract(vertices[c], vertices[b]);
     var normal = cross(t1, t2);
     normal = normalize(vec3(normal));

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

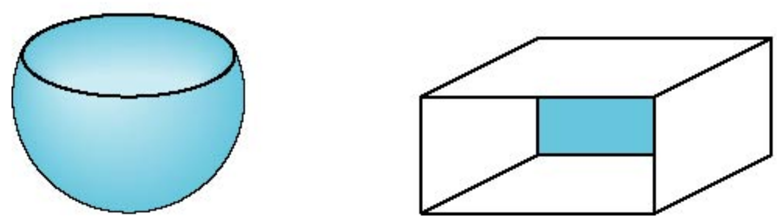


Front and Back Faces

- Every face has a front and back
- For may objects, we never see the back face so we do not care how or if it is rendered
- If it maters, we can handle in shader



back faces not visible



back faces visible

Polygonal Shading

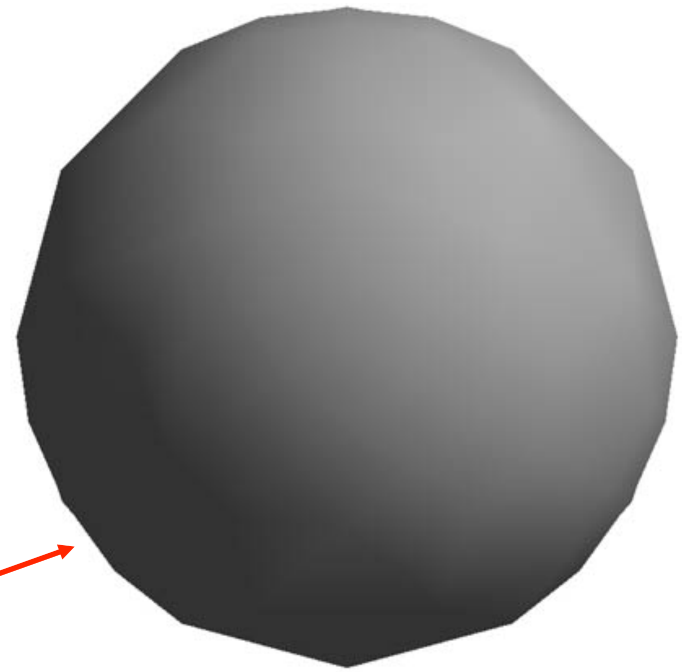
- In per vertex shading, shading calculations are done for each vertex
  - Vertex colors become vertex shades and can be sent to the vertex shader as a vertex attribute
  - Alternately, we can send the parameters to the vertex shader and have it compute the shade
- By default, vertex shades are interpolated across an object if passed to the fragment shader as a varying variable (smooth shading)
- We can also 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 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

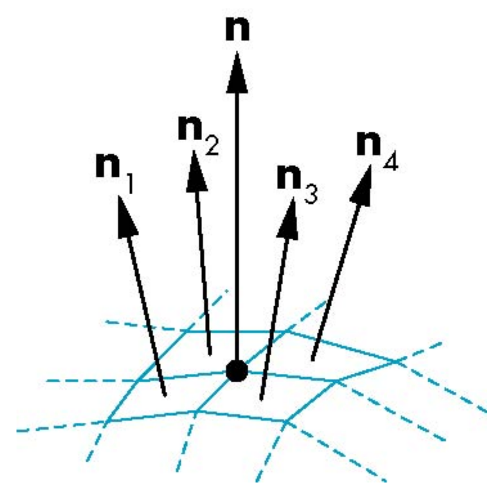
- We can set a new normal at each vertex
- Easy for sphere model
  - If centered at origin $\mathbf{n} = \mathbf{p}$
- Now smooth shading works
- Note silhouette edge



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

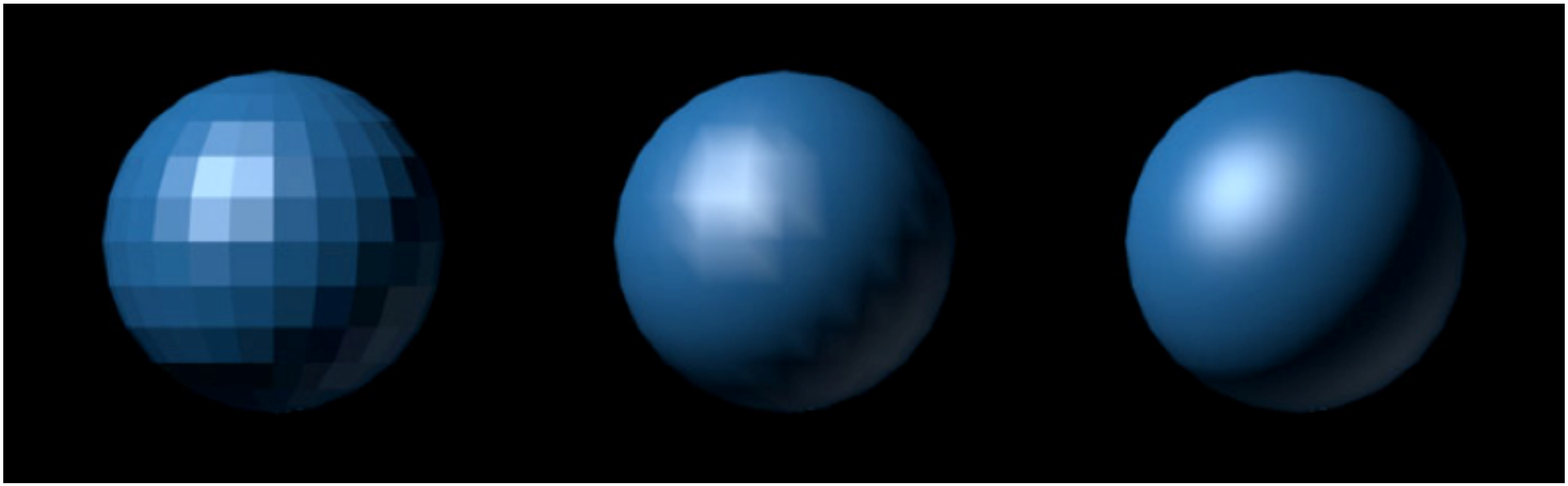
$$\mathbf{n} = (\mathbf{n}_1+\mathbf{n}_2+\mathbf{n}_3+\mathbf{n}_4) \,/\, |\mathbf{n}_1+\mathbf{n}_2+\mathbf{n}_3+\mathbf{n}_4|$$



Gouraud and Phong Shading

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

Comparison



flat   Gouraud shading   Phong shading

Comparison

- If the polygon mesh approximates surfaces with high curvatures, Phong shading may look smooth while Gouraud shading may show edges
- Phong shading requires much more work than Gouraud shading
  - 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 Lighting Shaders

Per-Vertex Lighting Shaders (use halfway vector)

```
// vertex shader

attribute vec4 vPosition;
attribute vec4 vNormal;
varying vec4 fColor;
uniform vec4 ambientProduct;
uniform vec4 diffuseProduct;
uniform vec4 specularProduct;
uniform mat4 modelViewMatrix; // 4 by 4
uniform mat4 projectionMatrix;
uniform vec4 lightPosition;
uniform float shininess;
```

```
void main()
{
  // transform vertex position from object space
     to eye space
  vec3 pos = (modelViewMatrix * vPosition).xyz;
  vec3 light = lightPosition.xyz;
  // assumed light position is already in the eye space
  vec3 L = normalize( light - pos );
  vec3 E = normalize( -pos ); // eye is at (0, 0, 0)
  vec3 H = normalize( L + E );

  vec4 NN = vec4(vNormal,0); // normalized normal
  // transform vertex normal into the eye space
  // assume no non-uniform scaling!
  // otherwise, compute and use normalMatrix
  // normalMatrix = transpose(inverse(modelViewMatrix))
  vec3 N = normalize( (modelViewMatrix * NN).xyz );
```

```
  // compute terms in the illumination equation
  // ambient
  vec4 ambient = AmbientProduct;
  float Kd = max( dot(L, N), 0.0 );
  // diffuse
  vec4 diffuse = Kd*DiffuseProduct;
  // specular
  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 = ProjectionMatrix * modelViewMatrix *
                 vPosition;
   fColor = ambient + diffuse + specular;
   fColor.a = 1.0; // opacity
}
```

```
// fragment shader

precision mediump float;
varying vec4 fColor;

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

Per-Fragment Lighting Shaders (use halfway vector)

```
// vertex shader
attribute vec4 vPosition;
attribute vec4 vNormal;
varying vec3 N, L, E; // sent to fragment shader
uniform mat4 modelViewMatrix;
uniform mat4 projectionMatrix;
uniform vec4 lightPosition;
void main()
{
    vec3 pos = (modelViewMatrix * vPosition).xyz;
    vec3 light = lightPosition.xyz;
    L = normalize( light - pos );
    E = normalize ( -pos);
    vec4 NN = vec4(vNormal,0);
    N = normalize( (modelViewMatrix*NN).xyz );
    gl_Position = projectionMatrix * modelViewMatrix *
                  vPosition;
};
```

```
// fragment shader

precision mediump float;

uniform vec4 ambientProduct;
uniform vec4 diffuseProduct;
uniform vec4 specularProduct;
uniform float shininess;
varying vec3 N, L, E; // sent from vertex shader

void main()
{
  vec4 fColor;
  vec3 NN, NL, NE;
  NN = normalize(N); // normalize per-fragment N
  NL = normalize(L); // normalize per-fragment L
  NE = normalize(E); // normalize per-fragment E
```

```
    vec3 NH = normalize( NL + NE );

    vec4 ambient = ambientProduct;

    float Kd = max( dot(NL, NN), 0.0 );
    vec4  diffuse = Kd*diffuseProduct;

    float Ks = pow(max(dot(NN, NH), 0.0), shininess);
    vec4  specular = Ks * specularProduct;
    if( dot(NL, NN) < 0.0 )
        specular = vec4(0.0, 0.0, 0.0, 1.0);

    fColor = ambient + diffuse +specular;
    fColor.a = 1.0;
    gl_FragColor = fColor;
}
```

Per-Vertex Lighting Shaders (use reflect vector)

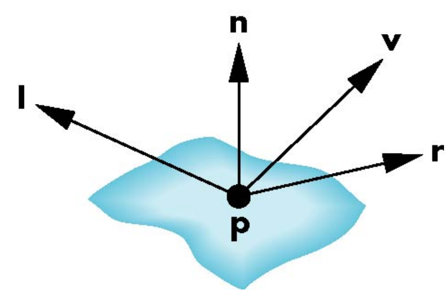
```
// vertex shader

attribute vec4 vPosition;
attribute vec4 vNormal;
varying vec4 fColor;
uniform vec4 ambientProduct;
uniform vec4 diffuseProduct;
uniform vec4 specularProduct;
uniform mat4 modelViewMatrix; // 4 by 4
uniform mat4 projectionMatrix;
uniform vec4 lightPosition;
uniform float shininess;
```

```
void main()
{
  // transform vertex position from object space
     to eye space
  vec3 pos = (modelViewMatrix * vPosition).xyz;
  vec3 light = lightPosition.xyz;
  // assumed light position is already in the eye space
  vec3 L = normalize( light - pos );
  vec3 E = normalize( -pos ); // eye is at (0, 0, 0)

  vec4 NN = vec4(vNormal,0); // normalized normal
  // transform vertex normal into the eye space
  // assume no non-uniform scaling!
  // otherwise, compute and use normalMatrix
  // normalMatrix = transpose(inverse(modelViewMatrix))
  vec3 N = normalize( (modelViewMatrix * NN).xyz );
  // note that the reflect function returns: L – 2(L·N)*N
  vec3 R = normalize(-reflect(L, N));
```

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



```
    // compute terms in the illumination equation
    // ambient
    vec4 ambient = AmbientProduct;
    float Kd = max( dot(L, N), 0.0 );
    // diffuse
    vec4 diffuse = Kd*DiffuseProduct;
    // specular
    float Ks = pow( max(dot(E, R), 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 = ProjectionMatrix * modelViewMatrix *
                   vPosition;
     fColor = ambient + diffuse + specular;
     fColor.a = 1.0; // opacity
}
```

```
// fragment shader

precision mediump float;
varying vec4 fColor;

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

Teapot Examples

Per-vertex

Per-fragment