SHADING

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
- Introduce modified Phong model
- Consider computation of required vectors

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

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

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

Slides modified from Angel book 6e
The infinite scattering and absorption of light can be described by the rendering equation:

\[ L_i(x, \omega_i) = \int f_s(x, \omega_i \rightarrow \omega_i) L_e(x, \omega_i) \cos \theta \, d\omega_i \]

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"
- Exist 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 reflect the light.
- A very rough surface scatters light in all directions.

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.

Lambertian Surface

- Perfectly diffuse reflector.
- Light scattered equally in all directions.
- Amount of light reflected is proportional to the vertical component of incoming light.
  \[ I_{\text{reflected}} \propto \cos \theta_i \]
  \[ \cos \theta_i = l \cdot n \] 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.

\[ I_r = k_s I \cos^\phi \]

reflected intensity
incoming intensity
shininess coef
absorption coef
**The Shininess Coefficient**
- Values of $\alpha$ 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 $k_a I_a$ to diffuse and specular terms.

**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 $1/(a + b d + c d^2)$ 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.

**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 $\alpha$.

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

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

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 a normalized vector halfway between $l$ and $v$.

Using the halfway vector
- Replace $(v \cdot r)^{\beta}$ by $(n \cdot h)^{\beta}$.
- $\beta$ is chosen to match shininess.
- Note that the halfway angle is half of the angle between $r$ and $v$ if vectors are coplanar.
- Resulting model is known as the modified Phong or 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, $n$ 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 (Chapter 11).

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

$$ r = 2(l \cdot n)n - l $$
**Plane Normals**

- Equation of plane: \( ax + by + cz + d = 0 \)
- From Chapter 3 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

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

**Normal to Sphere**

- Implicit function \( f(x,y,z) = 0 \)
- Normal given by gradient
- Sphere \( f(\mathbf{p}) = \mathbf{p} \cdot \mathbf{p} - 1 \)

\[
\mathbf{n} = \left[ \frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z} \right]^T = \mathbf{p}
\]

**Parametric Form**

- For sphere
  \[
  \begin{align*}
  x &= \cos u \sin v \\
  y &= \cos u \cos v \\
  z &= \sin u
  \end{align*}
  \]
- Tangent plane determined by vectors
  \[
  \frac{\partial \mathbf{p}}{\partial u} = \left[ \frac{\partial x}{\partial u}, \frac{\partial y}{\partial u}, \frac{\partial z}{\partial u} \right]^T \\
  \frac{\partial \mathbf{p}}{\partial v} = \left[ \frac{\partial x}{\partial v}, \frac{\partial y}{\partial v}, \frac{\partial z}{\partial v} \right]^T
  \]
- Normal given by cross product
  \[
  \mathbf{n} = \frac{\partial \mathbf{p}}{\partial u} \times \frac{\partial \mathbf{p}}{\partial v}
  \]

**General Case**

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

**Objectives**

- Introduce the OpenGL shading methods
  - per vertex vs per fragment shading
  - Where to carry out
- Discuss polygonal shading
  - Flat
  - Smooth
  - Gouraud
OpenGL shading

- Need
  - Normals
  - material properties
  - Lights
- State-based shading functions have been deprecated (glNormal, glMaterial, glLight)
- Get computed in application or send attributes to 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

\[
\text{plane } \mathbf{n} \cdot (\mathbf{p} - \mathbf{p}_0) = 0
\]
\[
\mathbf{n} = (\mathbf{p}_2 - \mathbf{p}_0) \times (\mathbf{p}_1 - \mathbf{p}_0)
\]
\[
\text{normalize } \mathbf{n} \leftarrow \mathbf{n} / ||\mathbf{n}||
\]

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
  \[
  \text{vec4 diffuse0} = \text{vec4}(1.0, 0.0, 0.0, 1.0);
  \text{vec4 ambient0} = \text{vec4}(1.0, 0.0, 0.0, 1.0);
  \text{vec4 specular0} = \text{vec4}(1.0, 0.0, 0.0, 1.0);
  \text{vec4 light0_pos} = \text{vec4}(1.0, 2.0, 3.0, 1.0);
  \]
- This gives options, but light bounces differently due to the object, not the light!
  - Better to have a light color, and set material properties to do rest

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 \(1/(a+b*d+c*d^2))\) where \(d\) is the distance from the point being rendered to the light source

Spotlights

- Derive from point source
  - Direction
  - Cutoff
  - Attenuation Proportional to \(\cos^\theta\)
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

Material Properties

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

```c
vec4 ambient = vec4(0.2, 0.2, 0.2, 1.0);
vec4 diffuse = vec4(1.0, 0.8, 0.0, 1.0);
vec4 specular = vec4(1.0, 1.0, 1.0, 1.0);
GLfloat shine = 100.0
```

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

Emissive Term

- We can simulate a light source in OpenGL 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 regardless of A
  - Later we will enable blending and use this feature
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 $n = p$
  - 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

\[ \mathbf{n} = \frac{\mathbf{n}_1 + \mathbf{n}_2 + \mathbf{n}_3 + \mathbf{n}_4}{|\mathbf{n}_1 + \mathbf{n}_2 + \mathbf{n}_3 + \mathbf{n}_4|} \]

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
Examples

• From Chapter 5 in book
• Designed to work like old OpenGL
• Each light has a diffuse, specular, and ambient component
• Each object has a material
• You multiply the material properties times the light properties
• Can do in CPU so done once per object

Per Vertex Lighting - Vertex Shader

```glsl
// vertex shader
in vec4 vPosition;
in vec3 vNormal;
out vec4 color;
// light and material props
uniform vec4 AmbientProduct;
uniform vec4 DiffuseProduct;
uniform vec4 SpecularProduct;
uniform mat4 ModelView;
uniform mat4 Projection;
uniform vec4 LightPosition;
uniform float Shininess;

void main() {  
  // Transform to eye coordinates
  vec3 pos = ModelView * vPosition.xyz;
  vec3 L = normalize(LightPosition.xyz-pos);
  vec3 E = normalize(vPosition.xyz);
  vec3 H = normalize(L+E);
  // Transform normal to eye coordinates
  vec4 N = normalize(ModelView*vec4(vNormal,0.0)).xyz;

  // 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;
  vec4 total = ambient + diffuse + specular;
  if(dot(L,N) < 0.0) // Is light below surface?
    total = vec4(0.0, 0.0, 0.0, 1.0);

gl_Position = Projection*ModelView*vPosition;
color = total;
color.a = 1.0;
}
```

Per Vertex Lighting – Fragment Shader

```glsl
// fragment shader
in vec4 color;
void main() {  
  gl_FragColor = color;
}
```

Per Fragment Lighting: Vertex Shader

```glsl
// vertex shader
in vec4 vPosition;
in vec3 vNormal;
out vec3 fN;  // Rasterizer will interpolate these
out vec3 fE;
out vec3 fL;
uniform mat4 ModelView;
uniform vec4 LightPosition;
uniform mat4 Projection;
void main() {  
  fN = vNormal;
  fE = vPosition.xyz;
  fL = LightPosition.xyz;
  if( LightPosition.w != 0.0 ) {  
    fL = LightPosition.xyz . vPosition.xyz;
  }  
gl_Position = Projection*ModelView*vPosition;
}
```

Per Fragment Lighting: Fragment Shader

```glsl
// Declarations
in vec3 fN;
in vec3 fL;
in vec3 fE;
uniform vec4 AmbientProduct;
uniform vec4 DiffuseProduct;
uniform vec4 SpecularProduct;
uniform mat4 ModelView;
uniform vec4 LightPosition;
uniform float Shininess;
void main() {  
  vec3 N = normalize(fN);  // Normalize vectors
  vec3 E = normalize(fE);
  vec3 L = normalize(fL);
  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 )  // Is light below surface?
    specular = vec4(0.0, 0.0, 0.0, 1.0);
  vec4 total = ambient + diffuse + specular;
  gl_FragColor = total;
  gl_FragColor.a = 1.0;
}
```

Some Notes on My Version

• Use Blinn/Phong model
• Lights only have a color, bounces determines solely by object material properties
• Ambient light is a property of scene not each light source
• If light position is vector (w=0) then it becomes a direction
• Use a normal matrix
• If light behind surface can’t be seen at all
Phong Shading – Vertex Shader

```glsl
out vec3 fN;
out vec3 fL;
out vec3 fE;
uniform mat4 Projection, ModelView, NormalMatrix;
uniform vec4 LightPosition;

void main() {
    vec3 N = normalize(vNormal.xyz); // N in world space.
    vec3 fE = ModelView*vec4(0, 1, 0, 1).xyz; // fE in clip space.
    gl_Position = ModelView*vec4(N.xyz, 1.0).xyz; // our eye vector (since camera at origin?)
    if (fE.w != 0.0) {
        // if it is a direction
        // The vector to the eye is light location - vertex (in same space)
        fL = LightPosition.xyz - gl_Position.xyz;
    } else {
        // In case it is a position
        fL = normalize(vNormal.xyz - gl_Position.xyz);
    }
    fN = normalize(fL); // put vertex into clip space.
}
```

Phong Shading - Fragment Shader

```glsl
in vec3 fN;
in vec3 fL;
in vec3 fE;
out vec4 fColor;

uniform vec4 ambientLight, lightColor;
uniform mat4 ModelView;
uniform vec4 LightPosition;
uniform float Shininess;

void main() {
    vec4 diffuse = vec4(0, 0, 0, 0);
    vec4 ambient = vec4(ambientLight.xyz, 1.0);
    vec4 specular = vec4(0, 0, 0, 0);
    float spec = 0.0, diff = dot(N, L);
    if (diff > 0) {
        diffuse = diffuse*kd;
        spec = dot(N, H);
        if (spec > 0) {
            specular = ks*pow(spec, Shininess); // specular reflectance
        }
    } else {
        diffuse = ambient+specular*lightColor;
        fColor = diffuse;
    }
}
```

Useful Helper Functions

- `setMaterial(....)`
- `pointLight(...)`

Review

- Steps for an OpenGL program
  1. Create shaders
  2. Create buffer objects and load data into them
  3. Vertex data is stored in Vertex Buffer Objects (VBOs)
  4. VBOs are stored in Vertex Array Objects (VAOs)
  5. Connect data locations with shader variables
  6. Render

Create Shaders

- Develop the source code
- Then
  - Create a program – `glCreateProgram()`
  - Create a shader – `glCreateShader()` (vertex, fragment, geometry)
  - Load Shader Source – `glShaderSource()`
  - Compile Shader – `glCompileShader()`
  - Attach Shader to Program – `glAttachShader()`
  - Link Program – `glLinkProgram()`
  - Use Program – `glUseProgram()`
- Do this with the `initShaders()` function

Vertex Objects

- Vertex Array Objects (VAOs)
  1. Generate a VAO name by calling `glGenVertexArrays()`
  2. Bind a VAO for initialization with `glBindVertexArray()` (make current)
  3. Update VBOs associated with the VAO
  4. Bind VAO for use in rendering
- Vertex Buffer Objects (VBOs)
  1. Generate VBO names by calling `glGenBuffers()`
  2. Bind a specific VBO for initialization by call `glBindBuffer()`
  3. Load data into VBO using `glBufferData()`
  4. Bind VAO for use in rendering with `glBindVertexArray()`

```glsl
GLuint vao;
glGenVertexArrays(1, &vao);
glBindVertexArray(vao);
```
Connecting Vertex Data to Shaders

- Vertex Data is passed to the Vertex Shader
  1. Find where a `in` vertex shader variable is with `glGetAttribLocation`
  2. Turn it on with `glEnableVertexAttribArray`
  3. Connect other pipe end to the VAO with `glVertexAttribPointer`

- These things are done to the current program (`glUseProgram`):

```c
GLuint vPosition = glGetUniformLocation(program, "vPosition");
glEnableVertexAttribArray(vPosition);
glVertexAttribPointer(vPosition, 4, GL_FLOAT, GL_FALSE, 0, BUFFER_OFFSET(0));
```

Render

- `glDrawArrays(GL_TRIANGLES, 0, NumVertices);`

- What about changing shader behavior with uniform parameters?
  - Need to get location with `glGetUniformLocation`
  - Set it with `glUniformX`

```c
glUniform4fv(glGetUniformLocation(program, "AmbientProduct"), 1, ambient_product);
glUniform4fv(glGetUniformLocation(program, "DiffuseProduct"), 1, diffuse_product);
glUniform4fv(glGetUniformLocation(program, "SpecularProduct"), 1, specular_product);
glUniform4fv(glGetUniformLocation(program, "LightPosition"), 1, light_position);
glUniform1f(glGetUniformLocation(program, "Shininess"), material_shininess);
glUniformMatrix4fv(glGetUniformLocation(program, "model_view"), 1, GL_TRUE, model_view);
```

Per Vertex Lighting - Shaders

```c
// vertex shader
in vec4 vPosition;
in vec3 vNormal;
out vec4 color; // vertex shade

// light and material properties
uniform vec4 AmbientProduct, DiffuseProduct, SpecularProduct;
uniform mat4 ModelView;
uniform mat4 Projection;
uniform vec4 LightPosition;
uniform float Shininess;

void main()
{
  // Transform vertex position into eye coordinates
  vec3 pos = (ModelView * vPosition).xyz;

  vec3 L = normalize(LightPosition.xyz - pos);
  vec3 E = normalize(-pos);
  vec3 H = normalize(L + E);

  // Transform vertex normal into eye coordinates
  vec3 N = normalize(ModelView*(vec4(vNormal, 0.0)).xyz;
```

Per Vertex Lighting - Shaders II VS

```c
void main()
{
  // Transform vertex position into eye coordinates
  vec3 pos = (ModelView * vPosition).xyz;

  vec3 L = normalize(LightPosition.xyz - pos);
  vec3 E = normalize(-pos);
  vec3 H = normalize(L + E);

  // Transform vertex normal into eye coordinates
  vec3 N = normalize(ModelView*(vec4(vNormal, 0.0)).xyz;
```

Per Vertex Lighting - Shaders III VS

```c
// 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;
}```
**Per Vertex Lighting - Shaders IV FS**

// fragment shader
in vec4 color;

void main()
{
    gl_FragColor = color;
}

**Per Fragment Lighting - Shaders I VS**

// vertex shader
in vec4 vPosition;
in vec3 vNormal;

// output values that will be interpolated per-fragment
out vec3 fN;
out vec3 fE;
out vec3 fL;

uniform mat4 ModelView;
uniform vec4 LightPosition;
uniform mat4 Projection;

**Per Fragment Lighting - Shaders II VS**

void main()
{
in vec4 vNormal;
in vec3 vPosition.xyz;
in vec3 LightPosition.xyz;
if( LightPosition.w != 0.0 ) {
    fL = LightPosition.xyz - vPosition.xyz;
}
gl_Position = Projection*ModelView*vPosition;
}

**Per Fragment Lighting - Shaders III FS**

// fragment shader
// per-fragment interpolated values from the vertex shader
in vec3 fN;
in vec3 fE;
in vec3 fL;

uniform vec4 AmbientProduct, DiffuseProduct, SpecularProduct;
uniform mat4 ModelView;
uniform vec4 LightPosition;
uniform float Shininess;

float Kd = max(dot(fL, fN), 0.0);
vec4 diffuse = Kd*DiffuseProduct;
float Ks = pow(max(dot(fN, fH), 0.0), Shininess);
vec4 specular = Ks*SpecularProduct;

// discard the specular highlight if the light's behind the vertex
if( dot(fL, fN) < 0.0 )
    specular = vec4(0.0, 0.0, 0.0, 1.0);

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

**Per Fragment Lighting - Shaders IV FS**

float Kd = max(dot(fL, fN), 0.0);
vec4 diffuse = Kd*DiffuseProduct;
float Ks = pow(max(dot(fN, fH), 0.0), Shininess);
vec4 specular = Ks*SpecularProduct;

// discard the specular highlight if the light's behind the vertex
if( dot(fL, fN) < 0.0 )
    specular = vec4(0.0, 0.0, 0.0, 1.0);

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

**Per Fragment Lighting - Shaders V VS**

float Kd = max(dot(fL, fN), 0.0);
vec4 diffuse = Kd*DiffuseProduct;
float Ks = pow(max(dot(fN, fH), 0.0), Shininess);
vec4 specular = Ks*SpecularProduct;

// discard the specular highlight if the light's behind the vertex
if( dot(fL, fN) < 0.0 )
    specular = vec4(0.0, 0.0, 0.0, 1.0);

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