# Foundations of Computer Graphics

Online Lecture 7: OpenGL Shading

*Motivation*

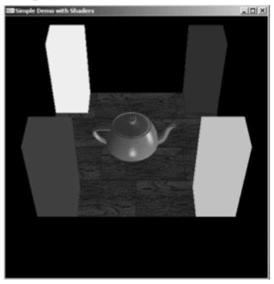
Ravi Ramamoorthi

## Motivation for Lecture

- Lecture deals with lighting (DEMO for HW 2)
- Briefly explain shaders used for mytest3
  - Do this before explaining code fully so you can start HW 2
  - Primarily explain with reference to source code

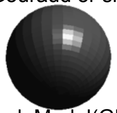
## Demo for mytest3

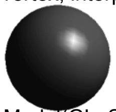
- Lighting on teapot
- Blue, red highlights
- Diffuse shading
- Texture on floor
- Update as we move



## Importance of Lighting

- Important to bring out 3D appearance
- Important for correct shading under lights
- The way shading is done also important
  - Flat: Entire face has single color (normal) from one vertex
  - Gouraud or smooth: Colors at each vertex, interpolate


glShadeModel(GL_FLAT)


glShadeModel(GL_SMOOTH)

## Brief primer on Color

- Red, Green, Blue primary colors
  - Can be thought of as vertices of a color cube
  - R+G = Yellow, B+G = Cyan, B+R = Magenta, R+G+B = White
  - Each color channel (R,G,B) treated separately
- RGBA 32 bit mode (8 bits per channel) often used
  - A is for alpha for transparency if you need it
- Colors normalized to 0 to 1 range in OpenGL
  - Often represented as 0 to 255 in terms of pixel intensities

## Outline

- *Gouraud and Phong shading (vertex vs fragment)*
- Types of lighting, materials and shading
  - Lights: Point and Directional
  - Shading: Ambient, Diffuse, Emissive, Specular
- Fragment shader for mytest3
  - HW 2 requires a more general version of this
- Source code in display routine

## Vertex vs Fragment Shaders

- Can use vertex or fragment shaders for lighting
- Vertex computations interpolated by rasterizing
  - *Gouraud (smooth) shading*
  - *Flat shading*
- Either compute colors at vertices, interpolate
  - This is standard in old-style OpenGL
  - Can be implemented with vertex shaders
- Or interpolate normals etc. at vertices
- And then shade at each pixel in fragment shader
  - *Phong shading* (different from Phong illumination)
  - More accurate

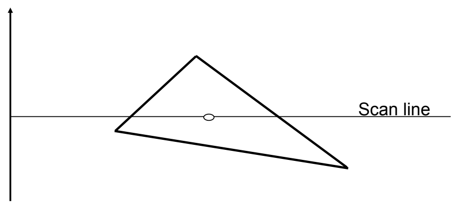
## Foundations of Computer Graphics

Online Lecture 7: OpenGL Shading

*Gouraud and Phong Shading*

Ravi Ramamoorthi

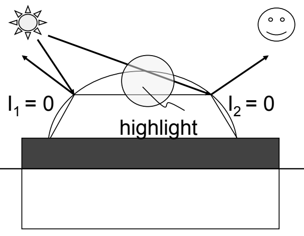
## Gouraud Shading – Details



Actual implementation efficient: difference equations while scan converting

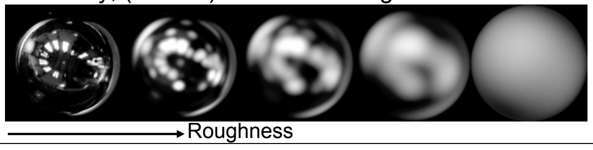
## Gouraud and Errors

- $I_1 = 0$ because (N dot E) is negative.
- $I_2 = 0$ because (N dot L) is negative.
- Any interpolation of $I_1$ and $I_2$ will be 0.



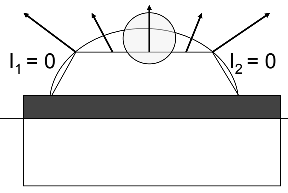
## Phong Illumination Model

- Specular or glossy materials: highlights
  - Polished floors, glossy paint, whiteboards
  - For plastics highlight is color of light source (not object)
  - For metals, highlight depends on surface color
- Really, (blurred) reflections of light source



## 2 Phongs make a Highlight

- Phong Shading (not illumination) model.
- First interpolate the ***normals,*** not colors.
- The entire lighting calculation is performed for each pixel, based on the interpolated normal. (Old OpenGL doesn't do this, but you can and will with current *fragment shaders*)



## Simple Vertex Shader in mytest3

```
#version 330 core  // Do not use any version older than 330!

// Inputs
layout (location = 0) in vec3 position;
layout (location = 1) in vec3 normal;
layout (location = 2) in vec2 texCoords;

// Extra outputs, if any
out vec4 myvertex;
out vec3 mynormal;
out vec2 texcoord;
```

## Simple Vertex Shader in mytest3

```
#version 330 core  // Do not use any version older than 330!
// ...Inputs and extra outputs seen earlier
// Uniform variables
uniform mat4 projection;
uniform mat4 modelview;
uniform int istex ;
void main() {
   gl_Position = projection * modelview * vec4(position, 1.0f);
   mynormal = mat3(transpose(inverse(modelview))) * normal ;
   myvertex = modelview * vec4(position, 1.0f) ;
   texcoord = vec2 (0.0, 0.0); // Default value just to prevent errors
   if (istex != 0){ texcoord = texCoords; }
}
```

## Outline

- Gouraud and Phong shading (vertex vs fragment)
- *Types of lighting, materials and shading*
  - *Lights: Point and Directional*
  - *Shading: Ambient, Diffuse, Emissive, Specular*
- Fragment shader for mytest3
  - HW 2 requires a more general version of this
- Source code in display routine

## Foundations of Computer Graphics

Online Lecture 7: OpenGL Shading

*Lighting and Shading*

Ravi Ramamoorthi

## Lighting and Shading

- Rest of this lecture considers lighting
- In real world, complex lighting, materials interact
- For now some basic approximations to capture key effects in lighting and shading
- Inspired by old OpenGL fixed function pipeline
  - But remember that’s not physically based

## Types of Light Sources

- Point
  - Position, Color
  - Attenuation (quadratic model)
- Attenuation

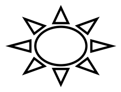
## Types of Light Sources

- Point
  - Position, Color
  - Attenuation (quadratic model)
- Attenuation
  - Usually assume no attenuation (not physically correct)
  - Quadratic inverse square falloff for point sources
  - Linear falloff for line sources (tube lights). Why?
  - No falloff for distant (directional) sources. Why?
- Directional (w=0, infinite far away, no attenuation)

## Material Properties

- Need normals (to calculate how much diffuse, specular, find reflected direction and so on)
  - *Usually specify at each vertex, interpolate*
  - GLUT used to do it automatically for teapots etc
  - Can do manually for parametric surfaces
  - Average face normals for more complex shapes
- Four terms: Ambient, Diffuse, Specular, Emissive

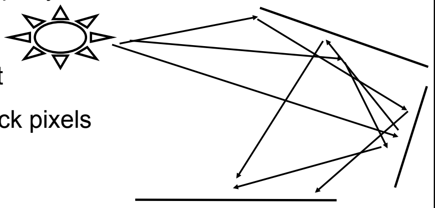
## Emissive Term



Only relevant for light sources when looking directly at them
- Gotcha: must create geometry to actually see light
- Emission does not in itself affect other lighting calculations

## Ambient Term

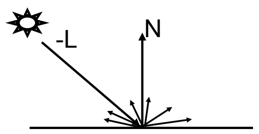
- Hack to simulate multiple bounces, scattering of light
- Assume light equally from all directions



- Global constant
- Never have black pixels

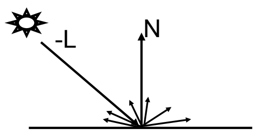
## Diffuse Term

- Rough matte (technically Lambertian) surfaces
- Light reflects equally in all directions



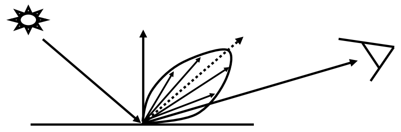
## Diffuse Term

- Rough matte (technically Lambertian) surfaces
- Light reflects equally in all directions



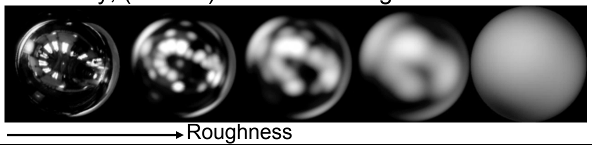
## Specular Term

- Glossy objects, specular reflections
- Light reflects close to mirror direction



## Phong Illumination Model

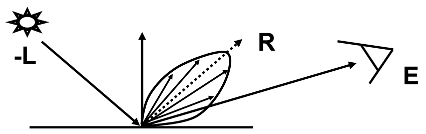
- Specular or glossy materials: highlights
  - Polished floors, glossy paint, whiteboards
  - For plastics highlight is color of light source (not object)
  - For metals, highlight depends on surface color
- Really, (blurred) reflections of light source



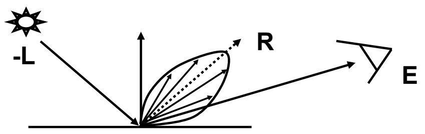
## Idea of Phong Illumination

- Simple way for view-dependent highlights
  - Not physically based
- Use dot product (cosine) of eye and reflection of light direction about surface normal
- Alternatively, dot product of half angle and normal
  - Has greater physical backing. We use this form
- Raise cosine lobe to some power to control sharpness or roughness

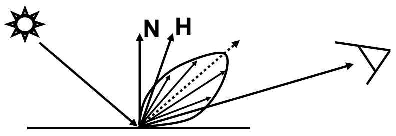
## Phong Formula



## Phong Formula

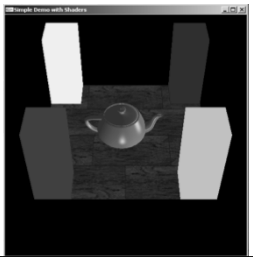


## Alternative: Half-Angle (Blinn-Phong)



## Demo in mytest3

- What happens when we make surface less shiny?



## Outline

- Gouraud and Phong shading (vertex vs fragment)
- Types of lighting, materials and shading
  - Lights: Point and Directional
  - Shading: Ambient, Diffuse, Emissive, Specular
- *Fragment shader for mytest3*
  - HW 2 requires a more general version of this
- Source code in display routine

**Foundations of Computer Graphics**

Online Lecture 7: OpenGL Shading

*Fragment Shader Example (HW 2 more general)*

Ravi Ramamoorthi

## Fragment Shader Setup

```
#version 330 core // Do not use any version older than 330!

// Inputs fragment shader are outputs of same name of vertex shader
in vec4 myvertex;
in vec3 mynormal;
in vec2 texcoord;

// Output the frag color
out vec4 fragColor;

uniform sampler2D tex ;
uniform int istex ;
uniform int islight ; // are we lighting.
uniform vec3 color;
```

## Fragment Shader Variables

```
// Assume light 0 is directional, light 1 is a point light.
// Actual light values are passed from the main OpenGL program.
// This could be fancier.  My goal is to illustrate a simple idea.
uniform vec3 light0dirn ;
uniform vec4 light0color ;
uniform vec4 light1posn ;
uniform vec4 light1color ;
```

## Fragment Shader Variables

```
// Now, set the material parameters.  These could be bound to
// a buffer.  But for now, I'll just make them uniform.
// I use ambient, diffuse, specular, shininess.
// Ambient is just additive and doesn't multiply the lights.
uniform vec4 ambient ;
uniform vec4 diffuse ;
uniform vec4 specular ;
uniform float shininess ;
```

## Fragment Shader Compute Lighting

```
vec4 ComputeLight (const in vec3 direction, const in vec4
   lightcolor, const in vec3 normal, const in vec3 halfvec, const
   in vec4 mydiffuse, const in vec4 myspecular, const in float
   myshininess) {

        float nDotL = dot(normal, direction)  ;
        vec4 lambert = mydiffuse * lightcolor * max (nDotL, 0.0) ;

        float nDotH = dot(normal, halfvec) ;
        vec4 phong = myspecular * lightcolor * pow (max(nDotH,
   0.0), myshininess) ;

        vec4 retval = lambert + phong ;
        return retval ;
}
```

## Fragment Shader Main Transforms

```
void main (void) {
    if (istex > 0) fragColor = texture(tex, texcoord);
    else if (islight == 0) fragColor = vec4(color, 1.0f) ;
    else {
        // They eye is always at (0,0,0) looking down -z axis
        // Also compute current fragment position, direction to eye

        const vec3 eyepos = vec3(0,0,0) ;
        vec3 mypos = myvertex.xyz / myvertex.w ; // Dehomogenize
        vec3 eyedirn = normalize(eyepos - mypos) ;

        // Compute normal, needed for shading.
        vec3 normal = normalize(mynormal) ;
```

## Fragment Shader Main Routine

```
// Light 0, directional
        vec3 direction0 = normalize (light0dirn) ;
        vec3 half0 = normalize (direction0 + eyedirn) ;
        vec4 col0 = ComputeLight(direction0, light0color, normal,
   half0, diffuse, specular, shininess) ;
        // Light 1, point
        vec3 position = light1posn.xyz / light1posn.w ;
        vec3 direction1 = normalize (position - mypos) ;
       // no attenuation
        vec3 half1 = normalize (direction1 + eyedirn) ;
        vec4 col1 = ComputeLight(direction1, light1color, normal,
   half1, diffuse, specular, shininess) ;
        fragColor = ambient + col0 + col1 ; }
}
```

## Outline

- Gouraud and Phong shading (vertex vs fragment)
- Types of lighting, materials and shading
  - Lights: Point and Directional
  - Shading: Ambient, Diffuse, Emissive, Specular
- Fragment shader for mytest3
  - HW 2 requires a more general version of this
- *Source code in display routine*

## Light Set Up (in display)

```
/* New for Demo 3; add lighting effects */
   {
     const GLfloat one[] = {1,1,1,1} ;
    const GLfloat medium[] = {0.5f, 0.5f, 0.5f, 1};
    const GLfloat small[] = {0.2f, 0.2f, 0.2f, 1};
    const GLfloat high[] = {100} ;
    const GLfloat zero[] = {0.0, 0.0, 0.0, 1.0} ;
    const GLfloat light_specular[] = {1, 0.5, 0, 1};
    const GLfloat light_specular1[] = {0, 0.5, 1, 1};
    const GLfloat light_direction[] = {0.5, 0, 0, 0}; // Dir lt
    const GLfloat light_position1[] = {0, -0.5, 0, 1};
    GLfloat light0[4], light1[4] ;
    // Set Light and Material properties for the teapot
    // Lights are transformed by current modelview matrix.
    // The shader can't do this globally. So we do so manually.
    transformvec(light_direction, light0) ;
    transformvec(light_position1, light1) ;
```

## Moving a Light Source

- Lights transform like other geometry
- Only modelview matrix (not projection). One of only real applications where the distinction is important
- Types of light motion
  - Stationary: set the transforms to identity before specifying it
  - Moving light: Push Matrix, move light, Pop Matrix
  - Moving light source with viewpoint (attached to camera). Can simply set light to 0 0 0 so origin wrt eye coords (make modelview matrix identity before doing this)

## Modelview Light Transform

```
/* New helper transformation function to transform vector by
   modelview */
void transformvec (const GLfloat input[4], GLfloat output[4]) {
  glm::vec4 inputvec(input[0], input[1], input[2], input[3]);
  glm::vec4 outputvec = modelview * inputvec;
  output[0] = outputvec[0];
  output[1] = outputvec[1];
  output[2] = outputvec[2];
  output[3] = outputvec[3];
}
```

## Set up Lighting for Teapot

```
glUniform3fv(light0dirn, 1, light0) ;
glUniform4fv(light0color, 1, light_specular) ;
glUniform4fv(light1posn, 1, light1) ;
glUniform4fv(light1color, 1, light_specular1) ;
glUniform4fv(ambient,1,small) ;
glUniform4fv(diffuse,1,medium) ;
glUniform4fv(specular,1,one) ;
glUniform1fv(shininess,1,high) ;
// Enable and Disable everything around the teapot
// Generally, we would also need to define normals etc.
// But the teapot object file already defines these for us.
if (DEMO > 4)
    glUniform1i(islight,lighting) ; // lighting only teapot.
```

## Shader Mappings in init

```
vertexshader = initshaders(GL_VERTEX_SHADER, "shaders/light.vert") ;
fragmentshader = initshaders(GL_FRAGMENT_SHADER, "shaders/light.frag") ;
shaderprogram = initprogram(vertexshader, fragmentshader) ;

// * NEW * Set up the shader parameter mappings properly for lighting.
islight = glGetUniformLocation(shaderprogram,"islight") ;
light0dirn = glGetUniformLocation(shaderprogram,"light0dirn") ;
light0color = glGetUniformLocation(shaderprogram,"light0color") ;
light1posn = glGetUniformLocation(shaderprogram,"light1posn") ;
light1color = glGetUniformLocation(shaderprogram,"light1color") ;
ambient = glGetUniformLocation(shaderprogram,"ambient") ;
diffuse = glGetUniformLocation(shaderprogram,"diffuse") ;
specular = glGetUniformLocation(shaderprogram,"specular") ;
shininess = glGetUniformLocation(shaderprogram,"shininess") ;
```