

# MATERIALS AND LIGHTING

in OpenGL



## Logistics

- ◆ Programming Assignment 1 – Line Drawing
- ◆ Programming Assignment 2 – Polygon Fill
- ◆ Programming Assignment 3 – Clipping
- ◆ Programming Assignment 4 – Hello OpenGL
  - ◆ All done including resubmits
- ◆ Programming Assignment 5 - Tessellation
  - ◆ Resubmits due 2/4.
- ◆ Programming Assignment 6 – Transformations / Viewing
  - ◆ Due tonight.

# Logistics

- ◆ MIDTERM ASSIGNMENT
- ◆ Midterm Exam
  - ◆ All done.

# Logistics

- ◆ Grad report
  - ◆ List of papers due Friday, Jan 11<sup>th</sup>
    - ◆ All done
  - ◆ Actual Report due February 15<sup>th</sup>

## Future plans

- ◆ Tuesday:
  - ◆ Color / Lighting / Materials
- ◆ Today:
  - ◆ Lights and Materials and GLSL.
- ◆ Next Week:
  - ◆ Textures

## Illumination Models

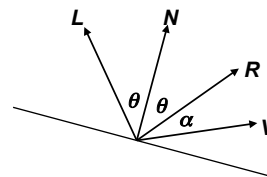
- ◆ Function or algorithm used to describe the reflective characteristics of a given surface
- ◆ Think about how that different surfaces reflect light in different ways
  - ◆ Glass vs. concrete vs. velvet
  - ◆ The way the light behaves/interacts will be based on viewpoint
- ◆ Sometimes known as shading

# The Phong Model

- ◆ *Ambient*
  - ◆ Light from no direction (due to scattering)
  - ◆ Surfaces illuminated by ambient light reflect in all directions
- ◆ *Diffuse*
  - ◆ Light from a certain direction
  - ◆ Reflected equally in all directions from the surface (causing the surface to look equally bright)
- ◆ *Specular*
  - ◆ Directional light which reflects off a surface in a particular direction
  - ◆ Causes the surface to have a shiny highlight
- ◆ *Emissive*
  - ◆ Light originating within an object
  - ◆ Does not affect other objects in the scene
  - ◆ We won't cover this in any detail

# Phong Illumination Model

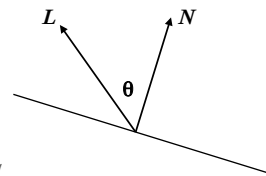
$$I_{\lambda} = \overset{\text{ambient}}{I_{a\lambda} k_a O_{a\lambda}} + f_{att} \overset{\text{diffuse}}{I_{d\lambda} [k_d O_{d\lambda} (L \cdot N)]} + \overset{\text{specular}}{k_s O_{s\lambda} (R \cdot V)^n}$$



# Diffuse Reflection

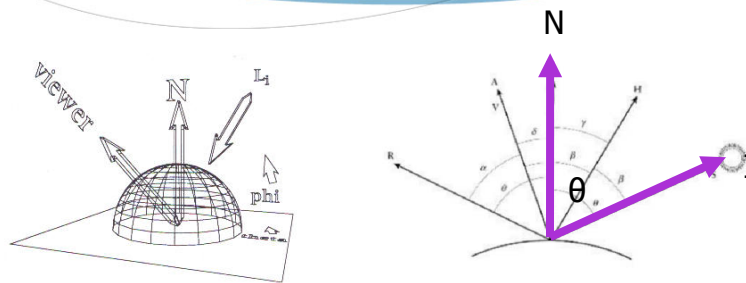
- ◆ Reflection from dull, matte surfaces (e.g., chalk)
  - ◆ Also called *Lambertian* reflection
  - ◆ Light comes from a point source
  - ◆ Light is reflected with equal intensity in all directions
- ◆ For a given surface, brightness depends only on the angle between the direction to the light source and the surface normal

$$D = I_d k_d \cos \Theta$$



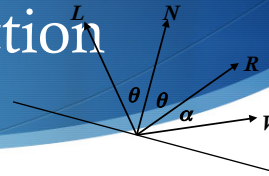
- ◆  $k_d$  is the *diffuse reflection coefficient*,  $0 \leq k_d \leq 1$ .

# Diffuse Reflection



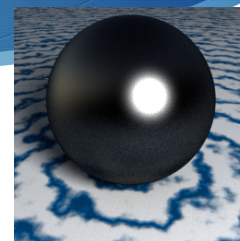
$$diffuse = L_i R_d \cos \theta$$

# Specular Reflection



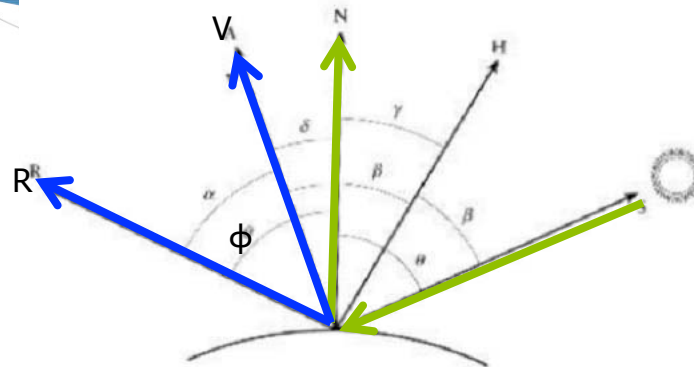
- ◆ Observable on any shiny surface
  - ◆ “Highlights” are caused by *specular reflection*
- ◆ On a perfectly reflective surface, light is reflected only in one direction
  - ◆ Angle of reflection = angle of incidence
- ◆ The angle between the reflection and the viewpoint,  $\alpha$ , determines the amount of reflection seen
  - ◆ Also affected by the *specular reflection exponent* of the material
  - ◆ For a perfect mirror, can only see reflection when  $\alpha$  is zero

# Specular reflection



- ◆ Reflection in the perfect “reflective” direction.
- ◆ Provides the “highlight”
- ◆ Effect is view dependent based on how well the view vector aligns with the perfectly reflective direction.
- ◆ Differences in the determination of specular reflection is the main distinguisher between different illumination models.

## Specular reflection: Phong



$$specular = L_i R_s \cos \phi$$

## Specular Reflection

- ◆ Maximum reflection occurs when  $\alpha$  is zero
- ◆ Falloff is approximated by  $\cos^n \alpha$ 
  - ◆  $n$  is the specular reflection exponent
  - ◆ Low values of  $n$  give gentle falloff; high values give sharp, focused highlights
- ◆ Note that the color of the reflected light is determined mostly by the color of the light.

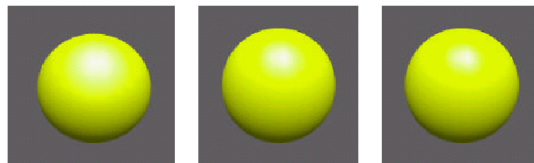
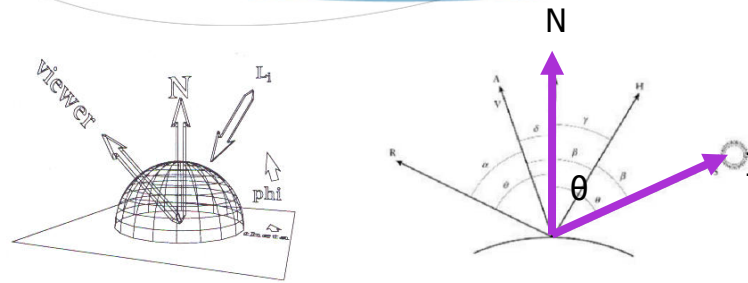


Figure 9.3: specular highlights with specular coefficients 20, 50, and 80 (left, center, and right), respectively

## Diffuse Reflection



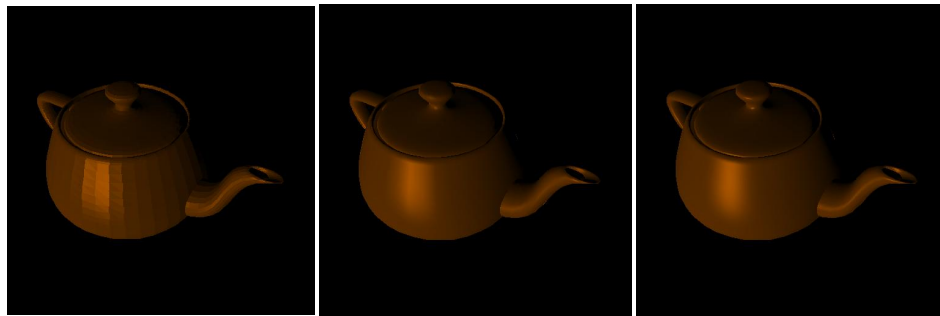
$$diffuse = L_i R_d \cos \theta$$

## Diffuse reflection in GLSL

- ◆ What we need to determine color
  - ◆ Position of a light source
  - ◆ Color of that light source
- ◆ Normal of surface
- ◆ Diffuse reflection “color” of object.



# Shading Models



Flat

Gouraud

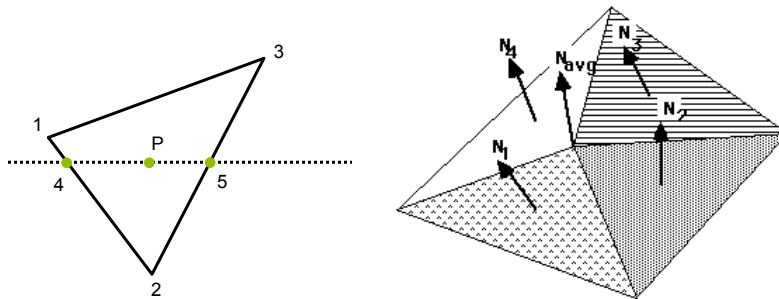
Phong

# Shading Models

- ◆ Gouraud Shading
  - ◆ Illumination is interpolated across each polygon
  - ◆ Normals required at each polygon vertex, calculated as an average of the normals of each face that shares that vertex
  - ◆ Illumination is calculated for each polygon vertex
  - ◆ Interior points interpolated from endpoint illumination intensities

# Shading Models

- Gouraud Shading – interpolating normals



## Diffuse – GLSL – Gouraud vertex shader

```
// Vertex position (in model space)
attribute vec4 vPosition;

// Normal vector at vertex (in model space)
attribute vec3 vNormal;

// Light position is given in world space
uniform vec4 lightPosition;

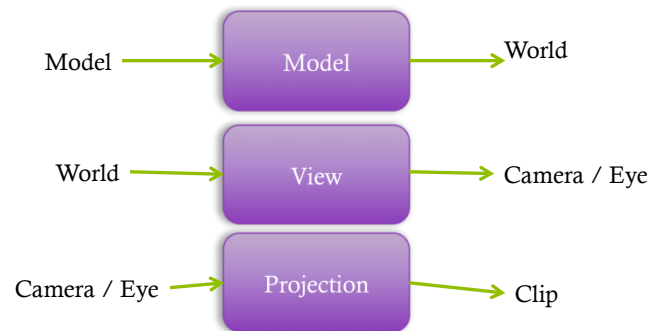
// Light color
uniform vec4 lightColor;

// Diffuse reflection color
uniform vec4 diffuseColor;

// Color to pass to fragment shader for interpolation
varying vec4 color;
```

# Vector spaces

- When performing lighting calculations, all vectors need to be in the same space



# Vector spaces

- Converting normal
  - Normals must be handles special (not affected by model translations)...If M is your ModelView transformation use

$$\left(M^{-1}\right)^T$$

- To transform normals
- [http://www.songho.ca/opengl/gl\\_normaltransform.html](http://www.songho.ca/opengl/gl_normaltransform.html)

## Diffuse – GLSL - Gouraud vertex shader

```
// View and projection matrices. Set to identity for now
// For non-default, this, or parameters to form these, would be
// passed in as uniform variables.
mat4 modelMatrix = mat4 (1.0,  0.0,  0.0,  0.0,
                        0.0,  1.0,  0.0,  0.0,
                        0.0,  0.0,  1.0,  0.0,
                        0.0,  0.0,  0.0,  1.0);

mat4 viewMatrix = mat4 (1.0,  0.0,  0.0,  0.0,
                       0.0,  1.0,  0.0,  0.0,
                       0.0,  0.0,  1.0,  0.0,
                       0.0,  0.0,  0.0,  1.0);

mat4 projMatrix = mat4 (1.0,  0.0,  0.0,  0.0,
                       0.0,  1.0,  0.0,  0.0,
                       0.0,  0.0,  1.0,  0.0,
                       0.0,  0.0,  0.0,  1.0);

mat4 modelViewMatrix = viewMatrix * modelMatrix;

//mat4 normalMatrix = transpose (inverse (modelViewMatrix));
```

## Diffuse – GLSL – Gouraud vertex shader

```
// All vectors need to be converted to "eye" space
// All vectors should also be normalized
vec4 vertexInEye = modelViewMatrix * vPosition;
vec4 lightInEye = viewMatrix * lightPosition;
vec4 normalInEye = normalize(modelViewMatrix * vec4(vNormal, 0.0));

vec3 L = normalize ((lightInEye - vertexInEye).xyz);
vec3 N = normalize (normalInEye.xyz);

// convert to clip space (like a vertex shader should)
gl_Position = projMatrix * modelViewMatrix * vPosition;

// Calculate color and pass to fragment shader
color = lightColor * diffuseColor * (dot(N, L));
}
```

## Diffuse – GLSL – Gouraud fragment shader

```
// color passed in from vertex shader
varying vec4 color;

void main()
{
    // set the final color
    gl_FragColor = color;
}
```

## Shading Models

- ◆ Phong Shading
  - ◆ Normal vectors are interpolated across each polygon
  - ◆ Averaged normal (per Gouraud required at each polygon vertex)
  - ◆ Illumination is calculated for each polygon interior point by applying illumination model directly using interpolated normal

## Diffuse – GLSL – Phong vertex shader

```
// Vertex position (in model space)
attribute vec4 vPosition;

// Normal vector at vertex (in model space)
attribute vec3 vNormal;

// Light position is given in world space
uniform vec4 lightPosition;

// Vectors to "attach" to vertex and get sent to fragment
// shader
// Vectors and points will be passed in "eye" space
varying vec3 lPos;
varying vec3 vPos;
varying vec3 vNorm;
```

## Diffuse – GLSL – Phong vertex shader

```
void main()
{
    // View and projection matrices. Set to identity for now
    // For non-default, this, or parameters to form these, would be
    // passed in as uniform variables.
    mat4 modelMatrix = mat4 (1.0, 0.0, 0.0, 0.0,
                             0.0, 1.0, 0.0, 0.0,
                             0.0, 0.0, 1.0, 0.0,
                             0.0, 0.0, 0.0, 1.0);

    mat4 viewMatrix = mat4 (1.0, 0.0, 0.0, 0.0,
                            0.0, 1.0, 0.0, 0.0,
                            0.0, 0.0, 1.0, 0.0,
                            0.0, 0.0, 0.0, 1.0);

    mat4 projMatrix = mat4 (1.0, 0.0, 0.0, 0.0,
                            0.0, 1.0, 0.0, 0.0,
                            0.0, 0.0, 1.0, 0.0,
                            0.0, 0.0, 0.0, 1.0);

    mat4 modelViewMatrix = viewMatrix * modelMatrix;

    //mat4 normalMatrix = transpose (inverse (modelViewMatrix));
```

## Diffuse – GLSL – Phong vertex shader

```
// All vectors need to be converted to "eye" space
// All vectors should also be normalized
vec4 vertexInEye = modelViewMatrix * vPosition;
vec4 lightInEye = viewMatrix * lightPosition;
vec4 normalInEye = normalize(modelViewMatrix * vec4(vNormal, 0.0));

// pass our vertex data to the fragment shader
lPos = lightInEye;
vPos = vertexInEye;
vNorm = normalInEye;

// convert to clip space (like a vertex shader should)
gl_Position = projMatrix * modelViewMatrix * vPosition;
}
```

## Diffuse – GLSL – Phong fragment shader

```
// Light color
uniform vec4 lightColor;

// Diffuse reflection color
uniform vec4 diffuseColor;

// Vectors "attached" to vertex and get sent to fragment shader
varying vec3 lPos;
varying vec3 vPos;
varying vec3 vNorm;

void main()
{
    // calculate your vectors
    vec3 L = normalize (lPos - vPos);
    vec3 N = vNorm;

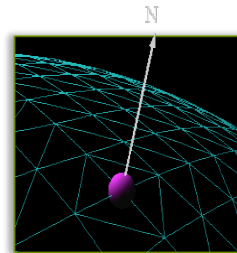
    // set the final color
    gl_FragColor = lightColor * diffuseColor * (dot(N, L));
}
```

## On the OpenGL (CPU side)

- ◆ To do list:
  - ◆ Calculate normals at each vertex
  - ◆ “Attach” normal to each vertex and send as attribute.
- ◆ Send diffuse parameters as uniform variables
- ◆ Draw.

## Surface Normals

- ◆ Normals define how a surface reflects light
  - ◆ Application usually provides normals as a vertex attribute
  - ◆ Use **unit normals** for proper lighting
    - ◆ scaling affects a normal's length
  - ◆ Cross product of polygon edges





## Vector Review

### ◆ Cross Product

$$\mathbf{u} \times \mathbf{v} = \hat{\mathbf{x}} (u_y v_z - u_z v_y) + \hat{\mathbf{y}} (u_z v_x - u_x v_z) + \hat{\mathbf{z}} (u_x v_y - u_y v_x),$$

$$|\mathbf{A} \times \mathbf{B}| = |\mathbf{A}| |\mathbf{B}| \sin \Theta$$

Perpendicular to both A and B

## Calculating Surface Normals

```
Begin Function CalculateSurfaceNormal (Input Triangle)
Returns Vector
```

```
Set Vector U to (Triangle.p2 minus Triangle.p1)
Set Vector V to (Triangle.p3 minus Triangle.p1)
```

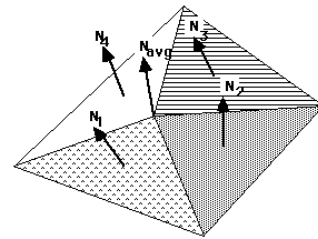
```
Set Normal.x to (multiply U.y by V.z) minus (multiply U.z by V.y)
Set Normal.y to (multiply U.z by V.x) minus (multiply U.x by V.z)
Set Normal.z to (multiply U.x by V.y) minus (multiply U.y by V.x)
Returning Normal
```

```
End Function
```

OpenGL.org

## Calculating and adding normals

- ◆ Normals should be generated when vertices are generated.
- ◆ Must be a normal for every vertex.
- ◆ Normals passed to GPU as attributes



## Attributes and OpenGL

- ◆ Sending data over to GPU with vertex positions




# Attributes and OpenGL

- ◆ glBufferSubData — updates a subset of a buffer object's data store

```
void glBufferSubData
```

```
(GLenum target,
 GLintptr offset,
 GLsizeiptr size,
 const GLvoid * data);
```



Offset within data buffer

Number of bytes

Array of data

# Attributes and OpenGL

```
// get the points for your shape
float *points = getVertices();
int dataSize = nVertices() * 4 * sizeof (float);

float *normals = getNormals();
int ndataSize = nVertices() * 3 * sizeof (float);

GLushort *elements = getElements();
int edataSize = nVertices() * sizeof (GLushort);
```

# Attributes and OpenGL

```
//generate the buffer
glGenBuffers( 1 , &buffer[0] );

//bind the buffer
glBindBuffer( GL_ARRAY_BUFFER , buffer[0] );

//buffer data
glBufferData( GL_ARRAY_BUFFER, dataSize + ndataSize , 0 ,
GL_STATIC_DRAW );

glBufferSubData ( GL_ARRAY_BUFFER, 0, dataSize, points);
glBufferSubData ( GL_ARRAY_BUFFER, dataSize, ndataSize,
normals);
```

# Attributes and GLSL

```
attribute vec4 vPosition;
attribute vec3 vNormal;
```



How does the shader know what part of the buffer corresponds to which attribute variable?

# Attributes and GLSL

```
int dataSize = numVerts[0] * 4 * sizeof (float);

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

GLuint vNormal = glGetAttribLocation( program , "vNormal" );
glEnableVertexAttribArray( vNormal );
glVertexAttribPointer( vNormal , 3 , GL_FLOAT , GL_FALSE, 0 ,
                      BUFFER_OFFSET(dataSize) );
```



## Summary

- ◆ On GLSL side
  - ◆ Must pass in normals as attribute data.
  - ◆ Construct vectors for illumination model (in common coordinate system)
  - ◆ Calculate color based on phong equation
- ◆ Gouraud Shading
  - ◆ Color computation done in vertex shader
  - ◆ Color passed to fragment shader and interpolated
- ◆ Phong Shading
  - ◆ Color computation done in fragment shader
  - ◆ Vectors / points required for computation passed to fragment shader.

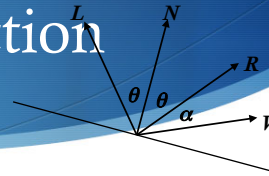
## Summary

- ◆ On OpenGL side
  - ◆ Must include normal data with vertex data in `ARRAY_BUFFER`.
  - ◆ Must tell GLSL what part of buffer corresponds to GLSL attributes.
- ◆ Questions.

## For the assignment

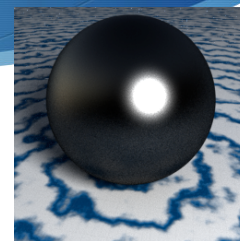
- ◆ Add specular component to complete the Phong model.

# Specular Reflection



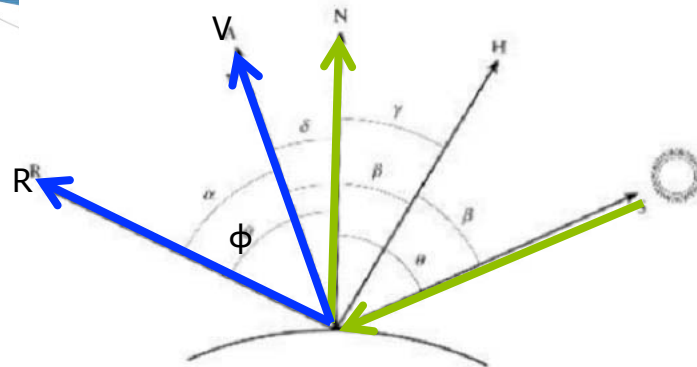
- ◆ Observable on any shiny surface
  - ◆ “Highlights” are caused by *specular reflection*
- ◆ On a perfectly reflective surface, light is reflected only in one direction
  - ◆ Angle of reflection = angle of incidence
- ◆ The angle between the reflection and the viewpoint,  $\alpha$ , determines the amount of reflection seen
  - ◆ Also affected by the *specular reflection exponent* of the material
  - ◆ For a perfect mirror, can only see reflection when  $\alpha$  is zero

# Specular reflection



- ◆ Reflection in the perfect “reflective” direction.
- ◆ Provides the “highlight”
- ◆ Effect is view dependent based on how well the view vector aligns with the perfectly reflective direction.
- ◆ Differences in the determination of specular reflection is the main distinguisher between different illumination models.

## Specular reflection: Phong



$$specular = L_i R_s \cos \phi$$

## Specular Reflection

- Maximum reflection occurs when  $\alpha$  is zero
- Falloff is approximated by  $\cos^n \alpha$ 
  - $n$  is the specular reflection exponent
  - Low values of  $n$  give gentle falloff; high values give sharp, focused highlights
- Note that the color of the reflected light is determined mostly by the color of the light.

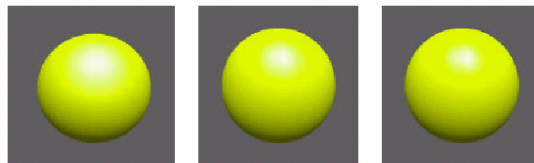
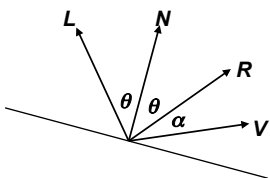


Figure 9.3: specular highlights with specular coefficients 20, 50, and 80 (left, center, and right), respectively



# Phong Illumination Model

$$I_{\lambda} = \overset{\text{ambient}}{I_{a\lambda} k_a O_{a\lambda}} + f_{att} I_{d\lambda} \left[ \overset{\text{diffuse}}{k_d O_{d\lambda} (L \cdot N)} + \overset{\text{specular}}{k_s O_{s\lambda} (R \cdot V)^n} \right]$$


break

## Adding specular

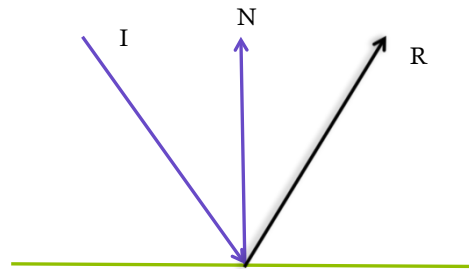
- ◆ What you will need:
  - ◆ Color of specular highlight
  - ◆ Specular exponent
  - ◆ View Vector (vector to camera)
  - ◆ Reflection vector
    - ◆ Can be calculated from light vector and normal

## Tips

- ◆ View Vector
  - ◆ Default camera at (0,0,0)
  - ◆ Can use `vPosition` in eye space to calculate
- ◆ Reflection Vector
  - ◆ Use GLSL `reflect` function

## GLSL reflect

◆ `R = reflect(I, N);`



Be sure to normalize vectors

# Questions?

- ◆ Due next Thursday.

- ◆ Next week:

- ◆ Textures.