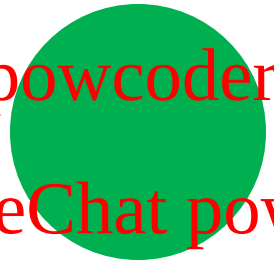
Chapter 3
CSCI-396
Jeff Bush

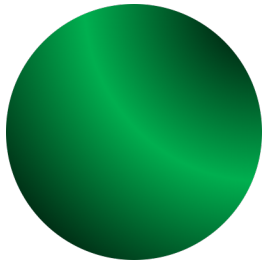
Lighting and Shading

Why do we need shading?

- So far our objects look unrealistic and "flat"
- If we draw a sphere we get:



- But we want something more like:



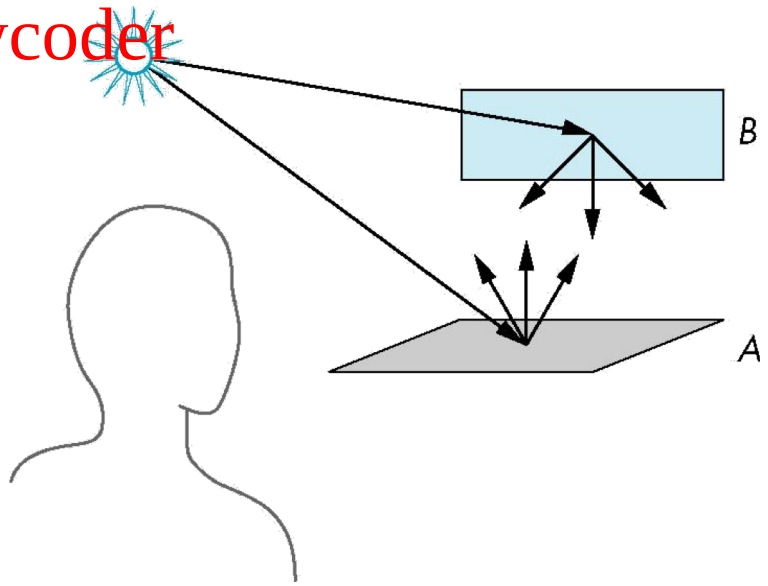
Shading

- The difference is that light-material interactions cause each point on the surface to visually have a different color or shade
  - The object itself is still one uniform color but the lighting causes it to *appear* as different shades
- Need to incorporate:
  - Light sources
  - Material of object
  - Location of viewer
  - Orientation of surface

Real World Lighting

Real world lighting

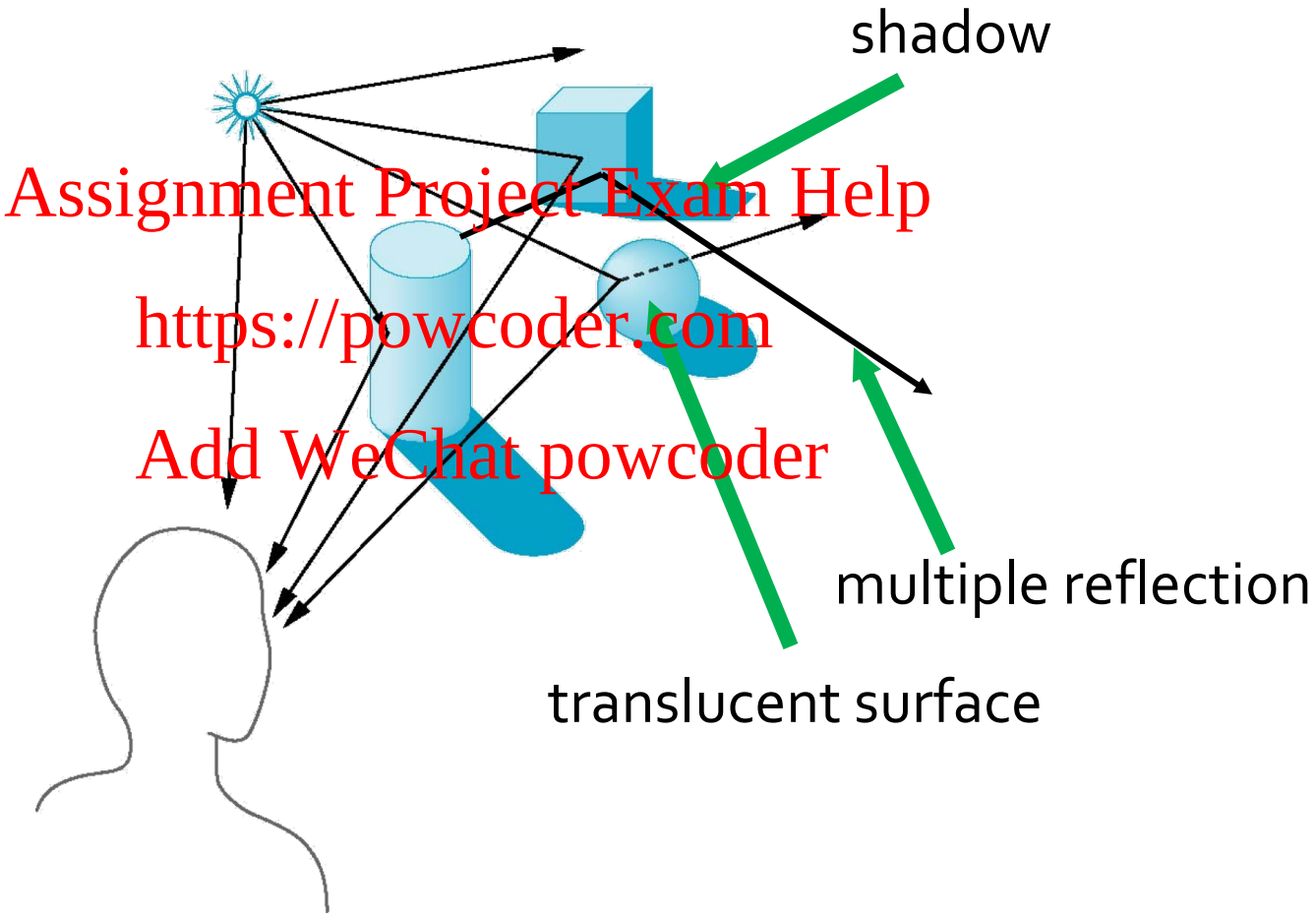
- Light is produced at some source and radiates outward
- Light strikes object A
  - Some of the light is scattered/reflected
  - Some is absorbed
- Some of the scattered light strikes B
  - Which scatters/absorbs it
- Some of that scattered light strikes A again…



Rendering Equation

- There is an infinite cycle of scattering and absorption of light
- It can be described by the *rendering* equation
  - Using integrals and limits the solution can be found
  - However no general solution and numerically solving it is too time intensive
  - Ray tracing is a special case for certain assumptions about the surfaces involved
- Rending equation is global however, including:
  - Shadows
  - Multiple scattering from object to object

Global Effects



Local vs Global Rendering

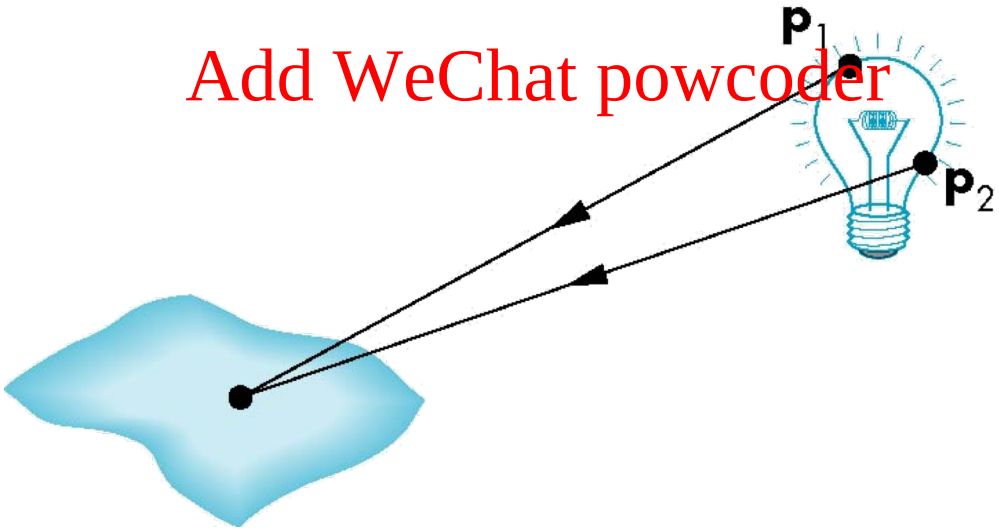
- Real-life shading requires a global calculation involving all objects and light sources simultaneously
  - Incompatible with pipeline which shades each triangle independently
- However, usually happy if things "look right" even if not perfect
  - Many techniques for approximating global effects

Light-Material Interaction

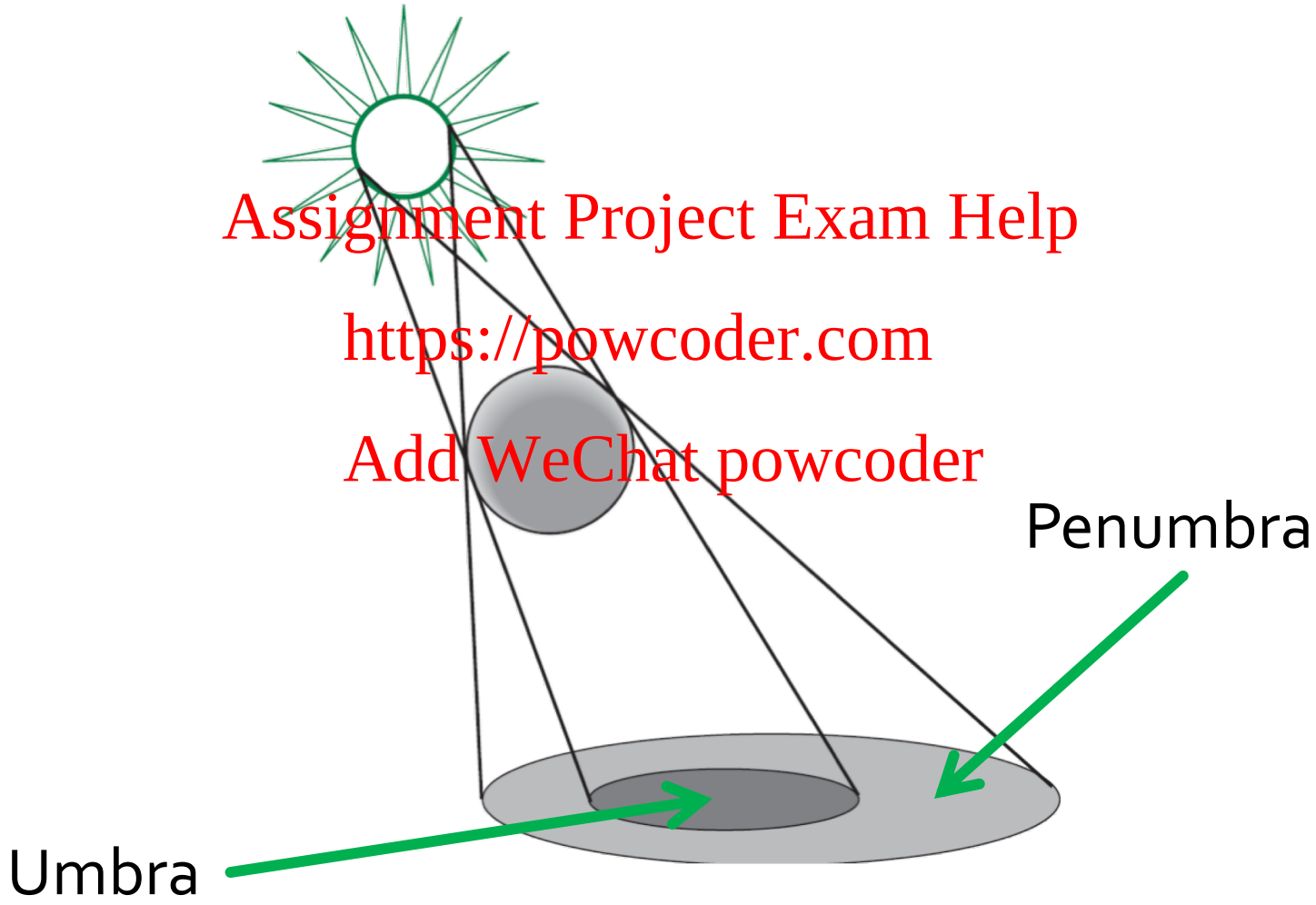
- When light strikes an object that light is partially absorbed (usually turns into heat) and partially scattered (reflected)
- Each color is not absorbed or reflected by the same amount for different surfaces
  - A red object under white light will look red since it reflects red and absorbs all other colors
- The reflected light is scattered in a manner that depends on the smoothness (at an atomic level) and orientation of the surface

Light Sources

- General/real light sources are difficult to work with because we must consider light coming from a range of points



General Light Sources



Our Light Model

Vector Math Notes

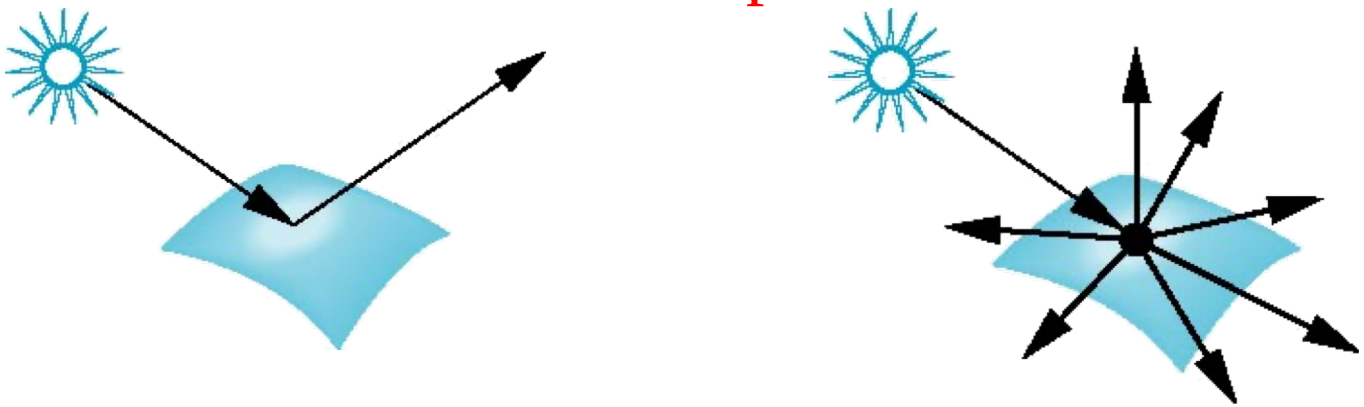
- Lighting models heavily use points and vectors so let's do some vector math review
- See "Scalars, Vectors, and Points" document
  - For those of you online I strongly recommend you have this handy

Light Sources

- Point source – like a light in a room
  - Has ***position*** and color
  - Radiates in all directions equally
  - Diminish amount based on distance
- Directional source – like the sun
  - Has ***direction*** and color
  - Essentially infinitely far away and bright light source
- Ambient light
  - Same amount of light everywhere in the scene
  - Used to model contribution of many sources and reflecting surfaces

Surface Types

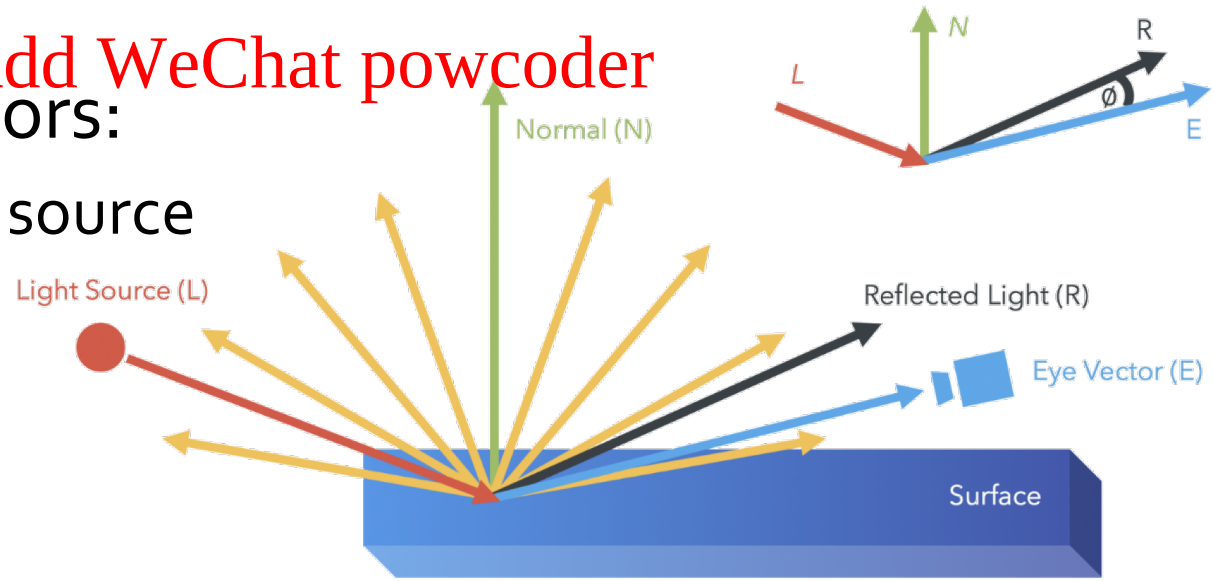
- Smoother surfaces mean more reflected light is concentrated in direction a perfect mirror would reflect
- A very rough surface scatters light in all directions



Smooth Rough

Phong Reflection Model

- Very simple model that can be computed rapidly
- Has three components:
  - Diffuse
  - Specular
  - Ambient
- Uses four vectors:
  - Direction to **L**ight source
  - Direction to **E**ye
  - Surface **N**ormal
  - Ideal **R**eflector



Ideal Reflector (Specular Surface)

An ideal reflector has an equivalent angle of incidence and angle of reflection and the light, reflector, and normal vectors are coplanar

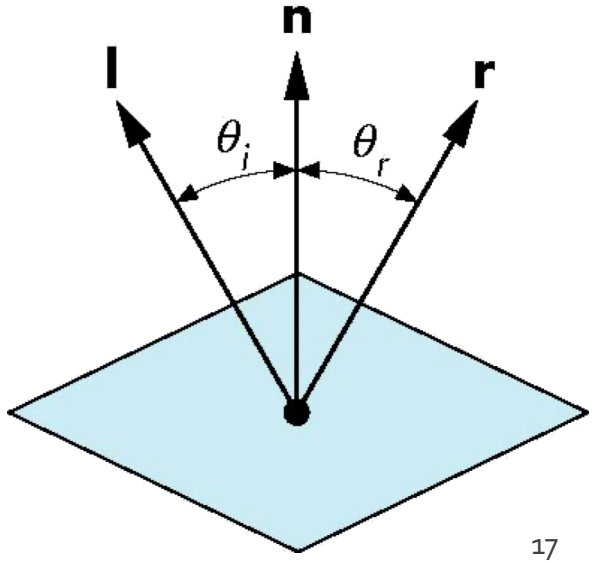
$$R = L - 2(L \cdot N)N$$

$$\theta_i = \theta_r = \cos^{-1}(L \cdot N)$$

assuming $\|L\| = \|N\| = 1$

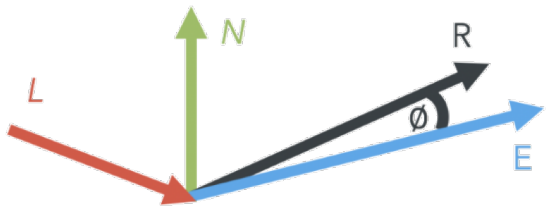
GLSL has `reflect(L, N)`

To compute R for us



Phong Specular Reflection Model

- Most objects aren't ideal reflectors (perfect mirrors)
- Phong model uses cos of angle θ between E and R
  - $\cos(\theta) = 1$ when $\theta = 0$ (E and R are same direction)
  - $\cos(\theta)$ decreases when E and R further apart
    - Actually $\cos^{\alpha}(\theta)$ with $\alpha > 1$ to decrease faster
    - α is "shininess"



How to efficiently compute $\cos(\theta)$?

Phong Specular Reflection Model

$$(R \cdot E)^{\alpha}$$

(assuming R and E are normalized)

- Gives a value from 0 to 1 for the proportion (percent) of the specular light we see*

- What else do we need to get the actual specular light ***color*** that we see?

* Technically could be −1 to 1 but we will simply treat negative values as 0

Phong Specular Reflection Model

$$(R \cdot E)^{\alpha}$$

(assuming R and E are normalized)

- Gives a value from 0 to 1 for the proportion (percent) of the specular light we see*

- What else do we need to get the actual specular light ***color*** that we see?
  - The color and intensity of the light
  - The color and reflectiveness of the material

* Technically could be −1 to 1 but we will simply treat negative values as 0

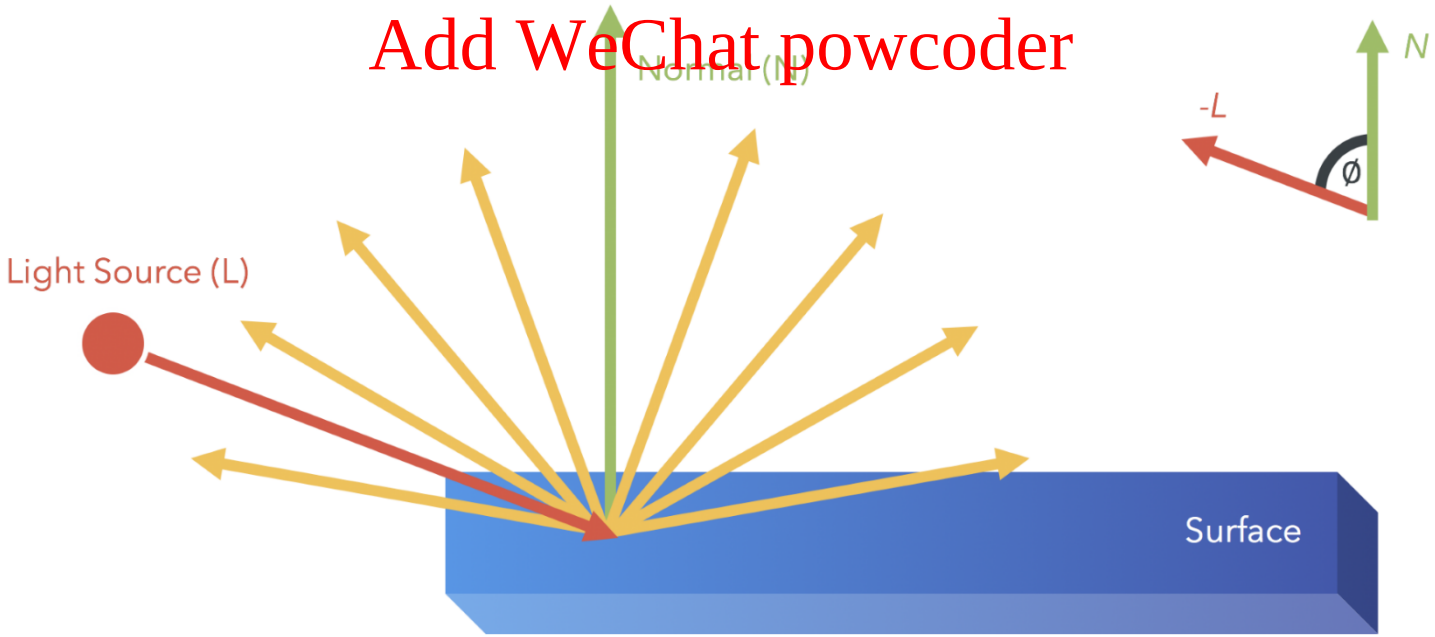
Phong Specular Reflection Model

- Light (color and intensity): represented as a single color
  - Same for all objects in the scene
    - Except may have distance incorporated
- Material (color, reflectiveness, and shininess): represented as a color plus a shininess scalar
  - Shininess: Metals: 100-200, Plastic: 5-10
  - Usually same for all vertices of entire object
- What kind and type of variable in the shader for these?
- Final equation: $I_s K_s (R \cdot E)^{\alpha}$

  where I_s is specular color/intensity of light, K_s is specular material color/reflectiveness, and α is shininess
  - Book uses C_l for I_s, C_m for K_s, and n for α but no one else does

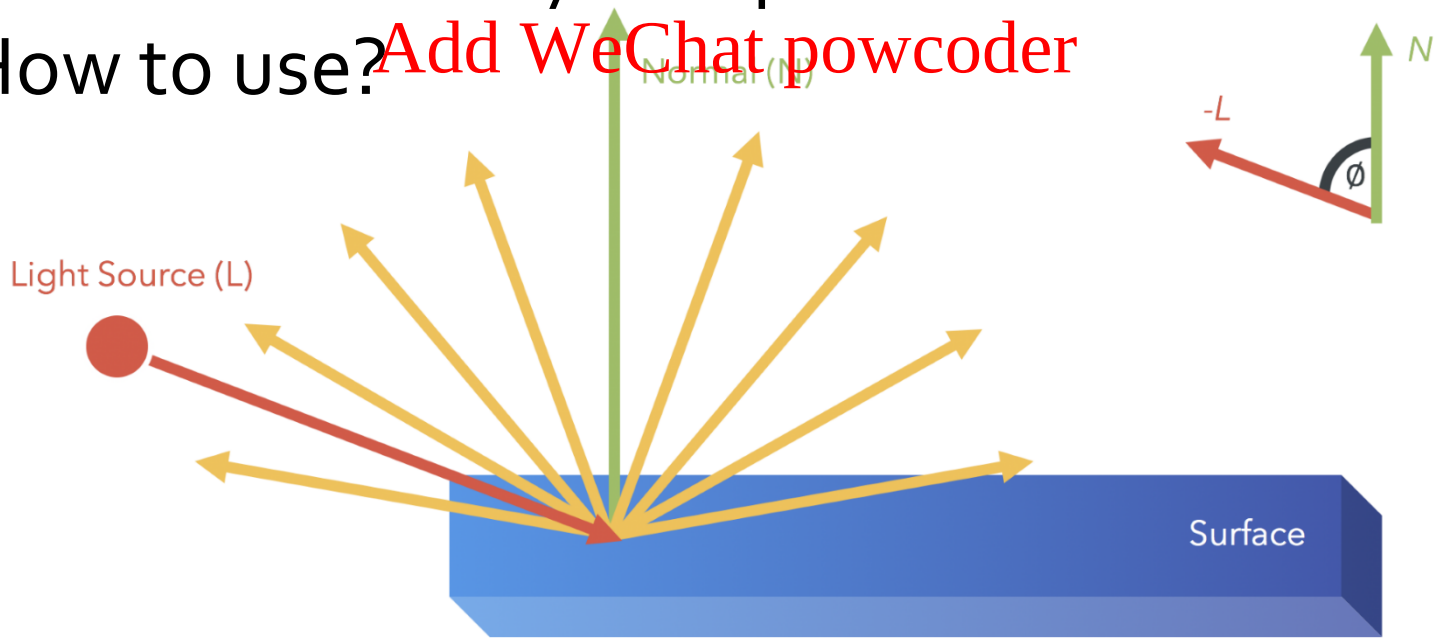
Lambert Diffuse Reflection Model

- Lambertian surfaces are perfectly diffuse
  - Light is scattered equally in all directions
- Amount of light reflected is proportional to the vertical component of incoming light
  - Called ***cosine emission law***



Lambert Diffuse Reflection Model

- Proportion of reflected light is $\cos(\theta)$
  - Another value from 0 to 1 but this time angle θ is between $-L$ and N and no shininess factor
- How to efficiently compute?
- How to use?



Lambert Diffuse Reflection Model

- Final diffuse equation:

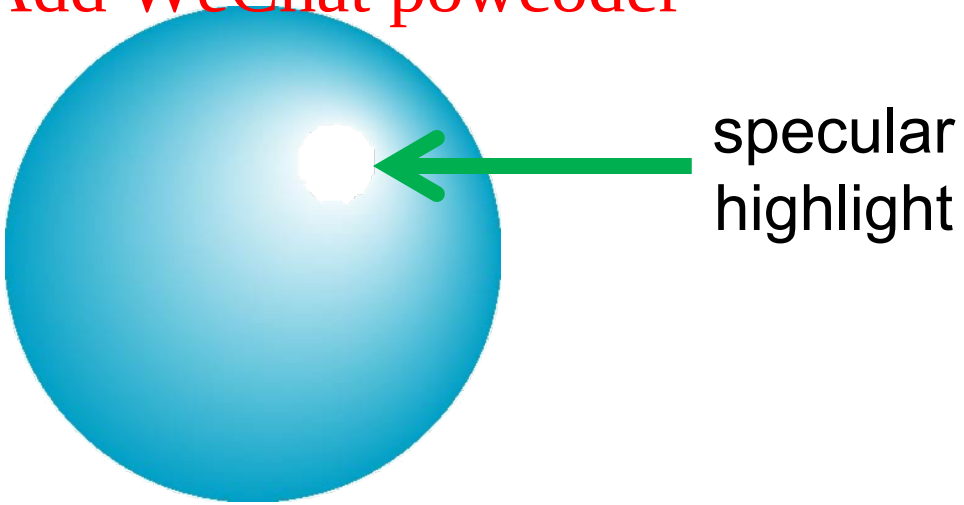
$$I_d K_d (-L \cdot N)$$

where I_d is diffuse color/intensity of light, K_d is diffuse material color/reflectiveness

  - Again the book uses C_l for I_d, C_m for K_d…

Combination: Specular and Diffuse

- Most surfaces are neither perfectly specular or diffuse
- Smooth surfaces show specular highlights due to incoming light being reflected in directions concentrated close to the direction of a perfect reflection



Combination : Specular and Diffuse

- Simply add the two results together
  - This can result in having more than 100% red, green, and/or blue color – think "blinding light"
    - GLSL will automatically fix colors but "clamping" all values to the 0 to 1 range
      - Clamp means if < 0 makes it 0 and if > 1 makes it 1

- Combination equation:

$$I_d K_d(-L \cdot N) + I_s K_s(R \cdot E)^{\alpha}$$

with $R = \text{reflect}(L, N)$ and normalized L, N, E vectors

Ambient Light

- Ambient light is the result of multiple interactions between (large) light sources and the objects in the environment
  - This is a "hack" to overcome limitations of the pipeline architecture of OpenGL
- Still have a light color/intensity and material color/reflectiveness but no scaling
- Final ambient equation:

$$I_a K_a$$

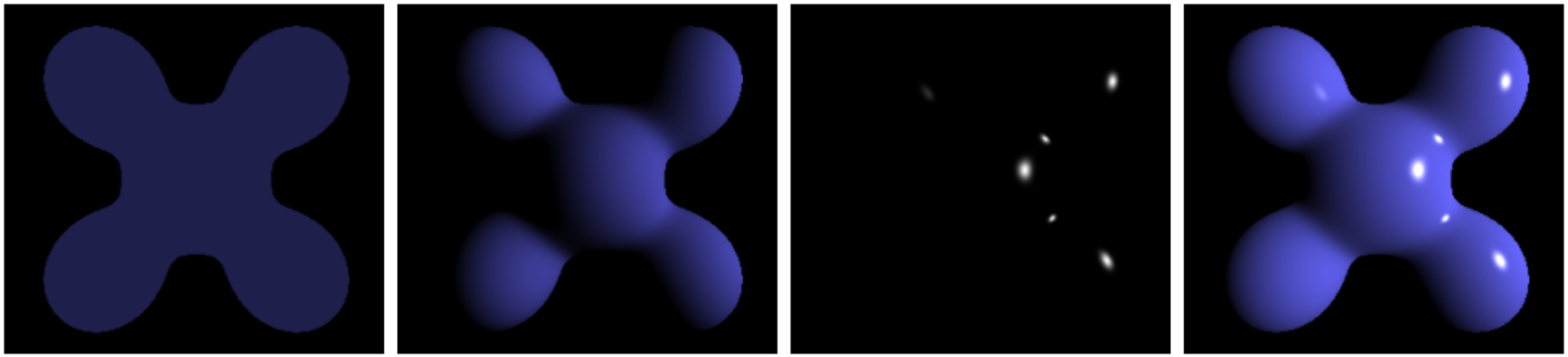
Phong Reflection Model – Complete

- Phong Reflection Model using the specular, diffuse, and ambient components
- Final equation:

$$I_a K_a + I_l K_d (-L \cdot N) + I_l K_s (R \cdot E)^{\alpha}$$

$$R = \text{reflect}(L, N)$$

L, N, and E are normalized



Light and Material Parameters

- I_a, I_d, I_s – light color/intensity
  - Determined by the type of light we want
  - Most lights we have in the world are (off-)white
    - Sometimes a bit bluer or yellower (warmer)
  - In many cases we will simply use pure, bright, white for all 3 of these
- K_a, K_d, K_s, α – material color/reflectance and shininess
  - Determined by the material of the object we want
  - Typically:
    - K_d will be adjusted the most based on color of object
    - K_a is dark version of K_d (low amount of ambient)
    - K_s is a shade of white/gray (regardless of actual object color)

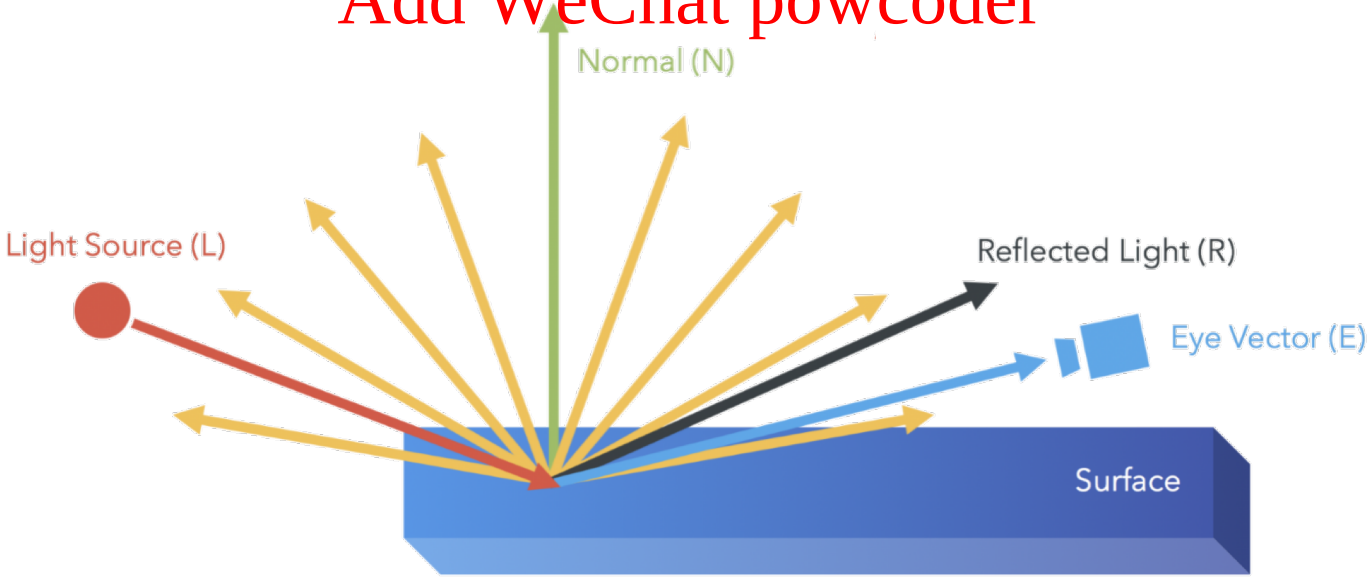
Phong Reflection Model Vectors

$$I_a K_a + I_d K_d(-L \cdot N) + I_s K_s (R \cdot E)^{\alpha}$$

$$R = \text{reflect}(L, N)$$

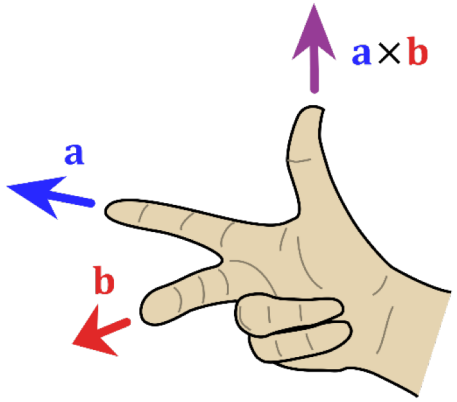
- Need to get L, N, and E vectors

  - Also need to normalize them but that is easy



Normal Vector

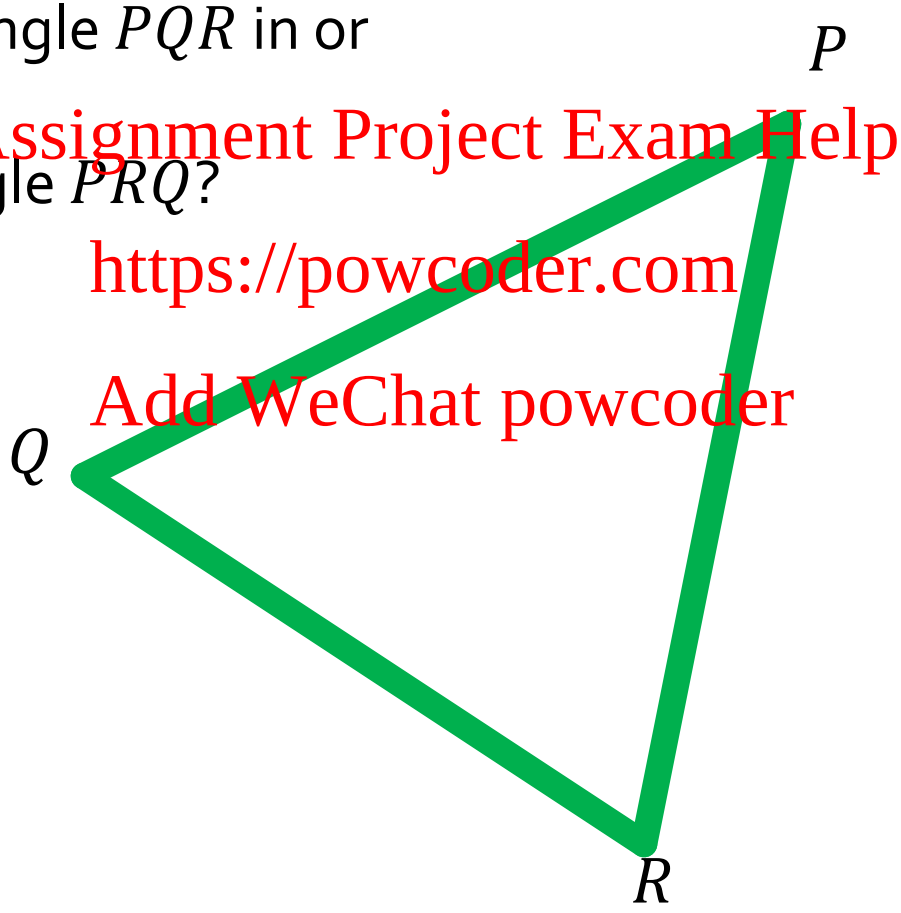
- The "normal vector" is perpendicular to a surface
  - Frequently just called ***the normal***
- Determined by the right-hand-rule
  - Have the fingers of your right hand follow the vertices around a triangle your thumb is the normal
    - Counter-clockwise around triangle
  - Mathematically this is the cross product of two vectors



Normal Vector

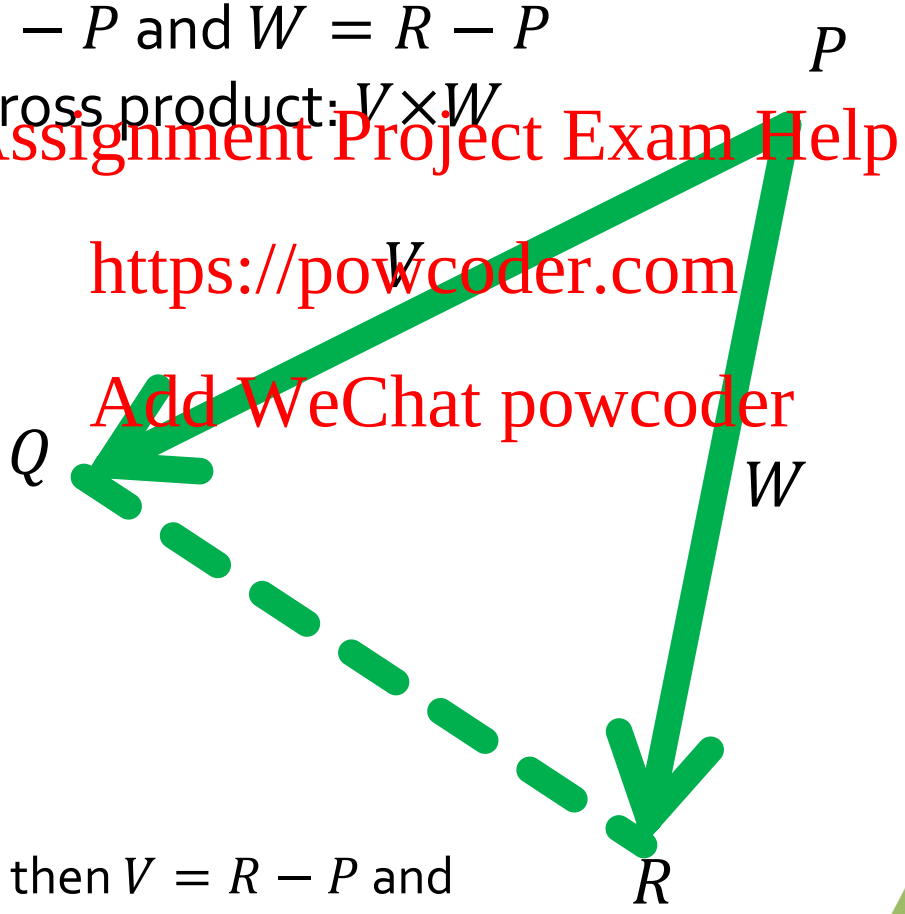
Using the right-hand rule, is the normal of the triangle PQR in or out of the screen?

What about triangle PRQ?

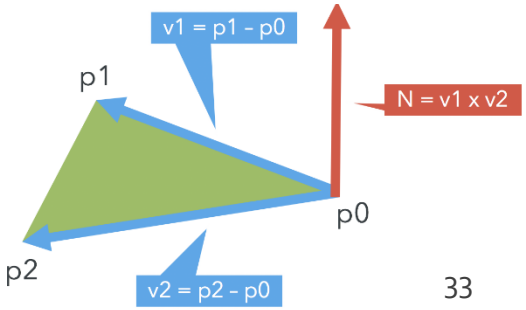


Normal Vector

To do this mathematically, need to get the vectors $V = Q - P$ and $W = R - P$ and the take the cross product: $V \times W$

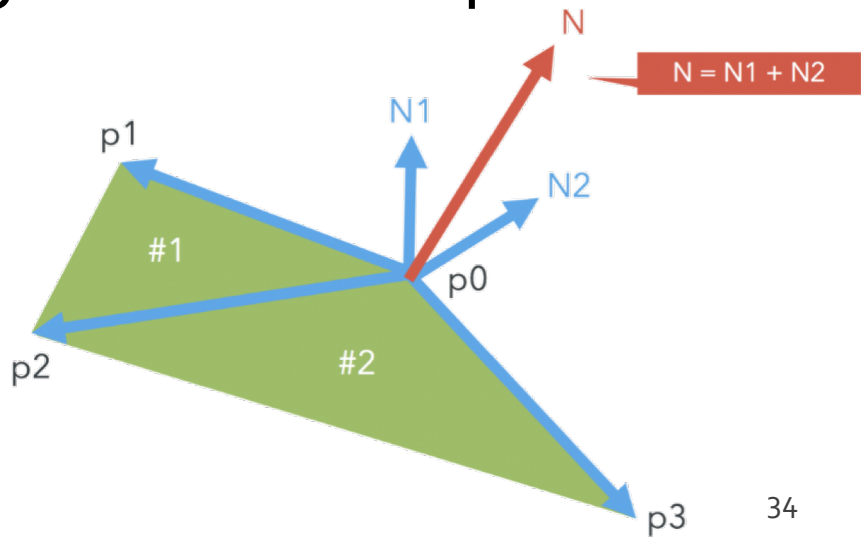


If doing triangle PRQ then $V = R - P$ and $W = Q - P$ instead and normal would flip



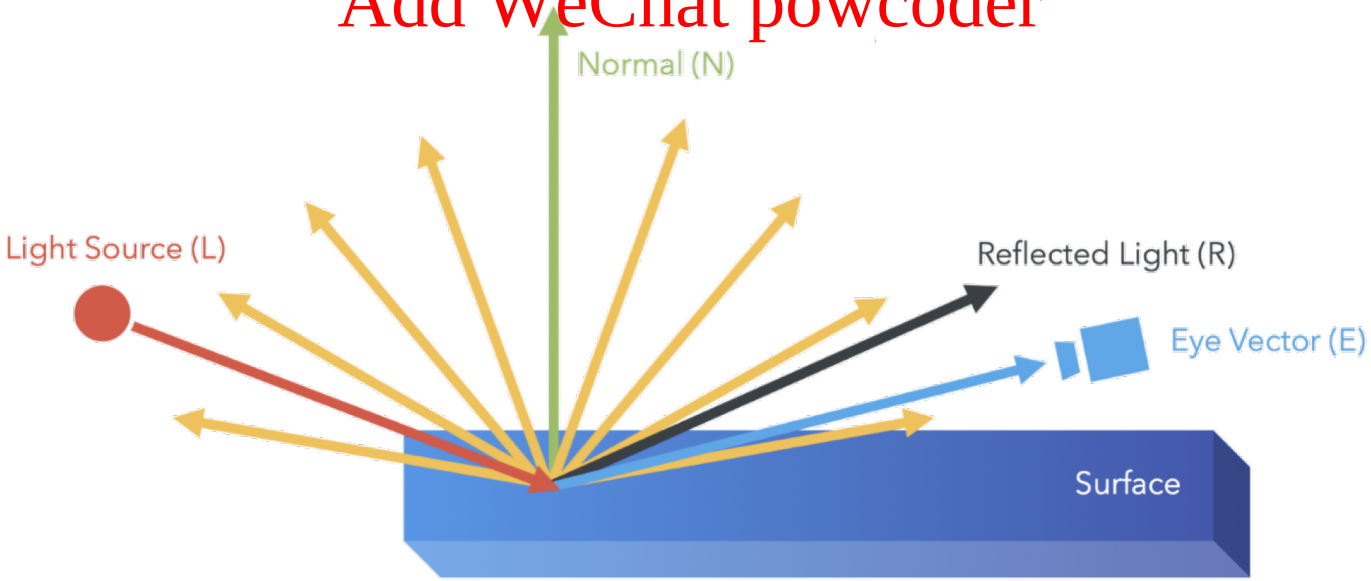
Normal Vector

- That allows us to compute the normal of a triangle, but our vertex shader works on vertices and we can only point attributes to vertices
- For vertices we take the sum (or average) of the normals of all triangles that share that vertex
- Since the normal for a single vertex is dependent on many vertices, where must this be computed? What kind and type of shader variable?



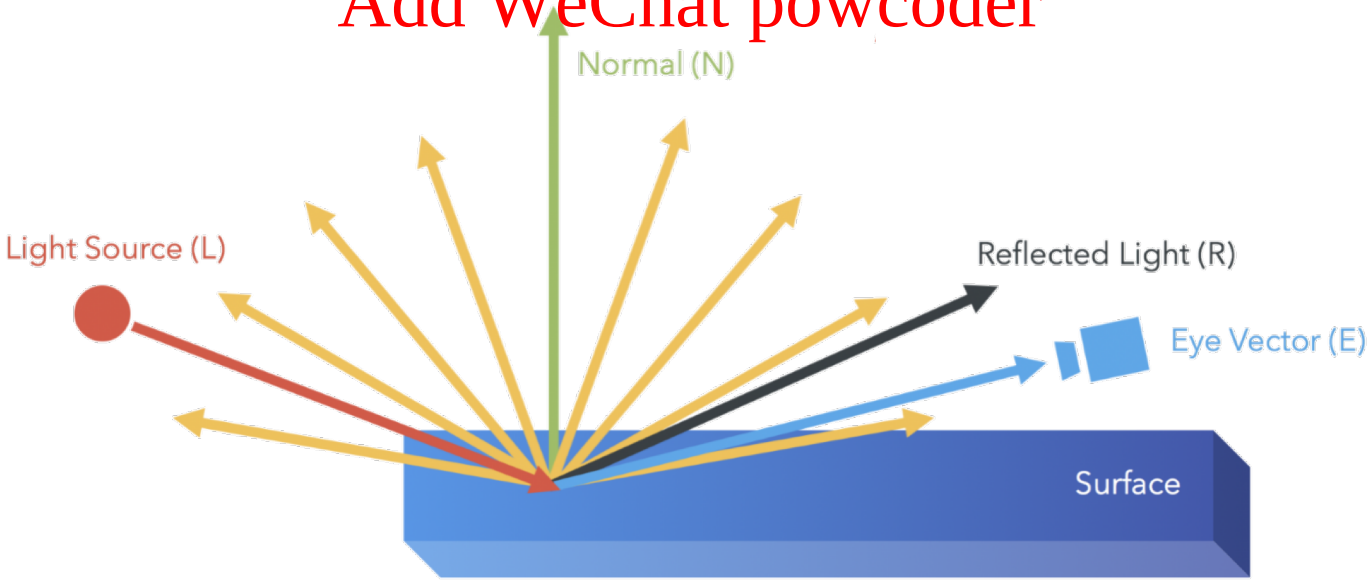
Light Vector

- L is the direction light is travelling when it reaches the vertex/surface
  - Point light: point P with light at P_L has $L = P - P_L$
  - Directional light: L is direction of light source
    - No work needed!



Eye Vector

- E is the direction from the vertex/surface to the eye (i.e. the viewer/camera)
  - OpenGL camera is always at 0,0,0 thus if we are at point P then $E = \langle 0,0,0 \rangle - P = -P$



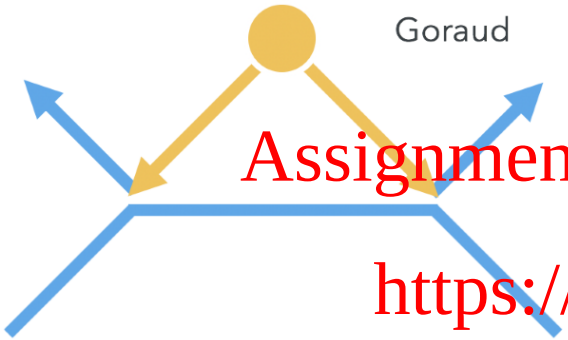
Bringing it All Together

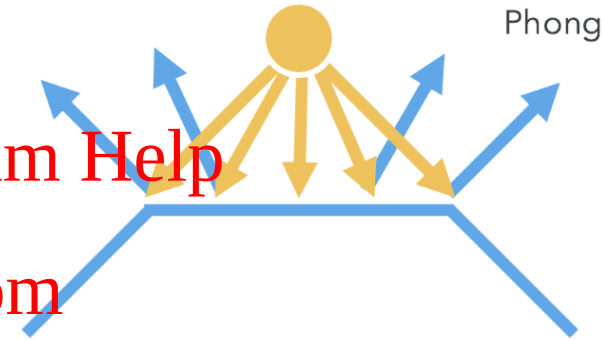
$$I_a K_a + I_d K_d(-L \cdot N) + I_s K_s (R \cdot E)^\alpha$$

$$R = \text{reflect}(L, N) \qquad L = P - P_L \qquad E = -P$$

- Uniforms:
  - I_a, I_d, I_s – lighting properties (each a `vec3`)
  - K_a, K_d, K_s – material properties (each a `vec3`)
  - α – material shininess (`float`)
  - P_L – position of light (`vec3`)
    - For flexibility can use a `vec4` with `w` as `1` for a point source and `0` for a directional source and then adjust L based on that
- Attributes:
  - P – vertex position (`vec4` with `w` as `1`)
  - N – vertex normal (`vec3`)

Which Fragment Gets What Code?





Vertex Shader

Computes the final color for the vertex using the vertex normal. Then, it passes the calculated color to the fragment shader in a varying variables.

Varying Color

Fragment Shader

Assigns the color for the fragment using the interpolated varying color.

Passes the vertex normal to the fragment shader in a varying variable.

Varying Normal

Computes the final color for the fragment using the respective interpolated normal.

Which Fragment Gets What Code?

- Flat:
  - Each triangle is a single, solid, color – lighting it applied per *triangle* (if at all)
  - Can be computed in JS or vertex shader
- Gouraud:
  - Calculate colors/lighting for each vertex in ***vertex shader*** and let them interpolate across the polygons
- Phong:
  - Calculate vectors in the vertex shader and let them interpolate across the polygon
  - Calculate the actual color/lighting per-fragment in the ***fragment shader***
- What do you think are the advantages/disadvantages of each?

Which Fragment Gets What Code?

- Flat:
  - Fastest
  - Each polygon is noticeable
- Gouraud:
  - Fastest
  - Specular highlights look bad, otherwise lighting is okay
- Phong:
  - Slowest
  - Usually highest quality but can lose sharpness of corners
    - Since vectors are interpolated across the shape and the normal along edges are angled

Bending of Light in Phong Model

How to fix this situation?

Bending of Light in Phong Model

- Can duplicate every vertex and have vertices with the same position but a different normal
  - For any object that we want "smooth" surfaces this won't be necessary (and actually counterproductive)
  - For sharp object we need to do this – this means limited ability reuse vertices in IBOs / triangle strips

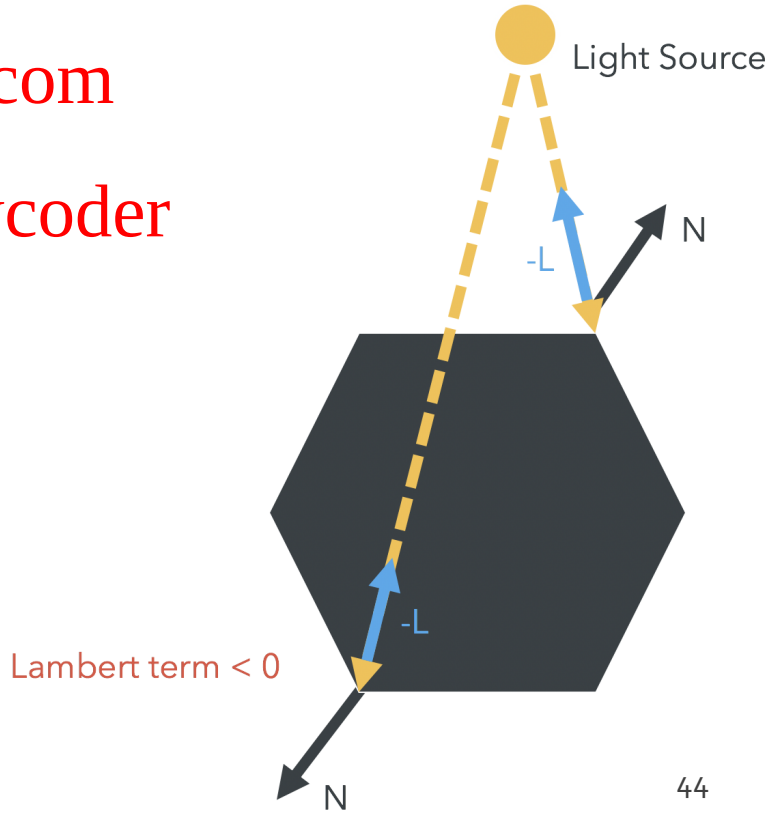
Let's Start Programming: Diffuse (Lambertian) Only at First

$$I_d K_d(-L \cdot N)$$
$$L = P - P_L$$

- Uniforms:
  - I_d – diffuse lighting (`vec3`) – name `uLightDiffuse`
  - K_d – diffuse material (`vec3`) – name `uMaterialDiffuse`
  - P_L – position of light (`vec4`) – name `uLight`
    - `w=1` for a point source and `0` for a directional source
- Attributes:
  - P – vertex position (`vec4`, `w=1`) – named `aPosition` and `P`
  - N – vertex normal (`vec3`) – named `aNormal`
- Use Phong Shading: vectors are computed in vertex shader and sent to the fragment shader where all other calculations are done
  - Need to be normalized in the fragment shader

Negative Light?

- Technically our current model can produce the diffuse coefficient from -1 to 1
  - Only 0 to 1 makes sense
  - If less than 0 just set to 0



Let's Start Programming: Add Ambient Lighting

$$I_a K_a + I_d K_d(-L \cdot N)$$

$$L = P - P_L$$

- Uniforms:
  - I_a, I_d – lighting properties (each a `vec3`)
  - K_a, K_d – material properties (each a `vec3`)
  - P_L – position of light (`vec3`)
- Attributes:
  - P – vertex position (`vec4`, `w=1`) – named `aPosition` and `P`
  - N – vertex normal (`vec3`)

Let's Start Programming: Add Specular Lighting

$$I_a K_a + I_d K_d(-L \cdot N) + I_s K_s (R \cdot E)^{\alpha}$$

$$R = \text{reflect}(L, N) \qquad L = P - P_L \qquad E = -P$$

- Uniforms:
  - I_a, I_d, I_s – lighting properties (each a `vec3`)
  - K_a, K_d, K_s – material properties (each a `vec3`)
  - α – material shininess (`float`) – name `uMaterialShininess`
  - P_L – position of light (`vec3`)
- Attributes:
  - P – vertex position (`vec4` with `w` as `1`)
  - N – vertex normal (`vec3`)
- Note: R vector is computed in fragment shader

Light Position Controls

- Currently the position of the light is fixed in the `init()` function
  - Add three number boxes to control the position of the light
  - Add event handlers for them
  - Add default values in `init()`

Light Position or Light Direction

- With a very simple piece of code we can support both light direction and light position
  - Our `vec4` for the light will use `w=1` meaning it is a point light and `w=0` meaning it is a directional light
    - This matches with the definitions on your sheet, `w=0` means a vector, i.e. a direction
- Previously we had $L = P - P_L$ to compute the light direction from a point light (i.e. if `w=1`), if `w=0` what should we do instead?

Light Position or Light Direction

- With a very simple piece of code we can support both light direction and light position
  - Our `vec4` for the light will use `w=1` meaning it is a point light and `w=0` meaning it is a directional light
    - This matches with the definitions on your sheet, `w=0` means a vector, i.e. a direction
- Previously we had $L = P - P_L$ to compute the light direction from a point light (i.e. if `w=1`), if `w=0` what should we do instead?

$$L = uLight.xyz$$

- Add this to the shaders and add a checkbox to the HTML to control if light is a point or direction

Attenuation of Light

- In the real world amount of light is proportional to the square of the distance to the light source:

$$I \sim \frac{1}{d^2}$$

  - However real world lights also also not point sources
- Use the following to reduce effects of a point light source:

$$\frac{1}{a + bd + cd^2}$$

  - This is applied to the diffuse and specular terms
  - Why not the ambient term?
- Add this into the shaders where d is the distance between the position and the light and a, b, and c are constants given by the user (need some more HTML controls)

Attenuation of Light

- When using real-world values of 0,0,1 for the attenuation coefficients a, b, c how far away can you get before the light is gone?

Attenuation of Light

- When using real-world values of 0,0,1 for the attenuation coefficients a, b, c how far away can you get before the light is gone?
- We need a much stronger light!
  - The color `vec3(1.0, 1.0, 1.0)` represents the strongest white light your ***monitor*** can produce
    - Can your monitor light up a room or the outdoors?
  - We can have light that has values greater than 1.0!
    - Monitor: $50 - 300\ cd/m^2$
    - Office Space: $1500\ cd/m^2$
    - Sun: $1.6 \times 10^9\ cd/m^2$

Assignment Project Exam Help

https://powcoder.com

Add WeChat powcoder

Problem: Light Bending

- Using "smooth" shading since the normals are smoothly interpolated between the vertices
  - This is typically what we want as it allows us to using less faces to render a smooth surface
- However, legitimate sharp edges have an unrealistic bending of light
  - We can accomplish flat shading in 3 ways:
    - Duplicate our vertices, each with a different normal
      - Bad due to duplicated vertices (this is the current solution in demo)
    - Tell OpenGL to use flat shading
      - Very tricky to get right but is possible
    - Compute the normals in the fragment shader:
      - Use the derivative of the position to get two tangents to the surface
      - Use cross product to get normal from tangents

Assignment Project Exam Help

https://powcoder.com

Add WeChat powcoder

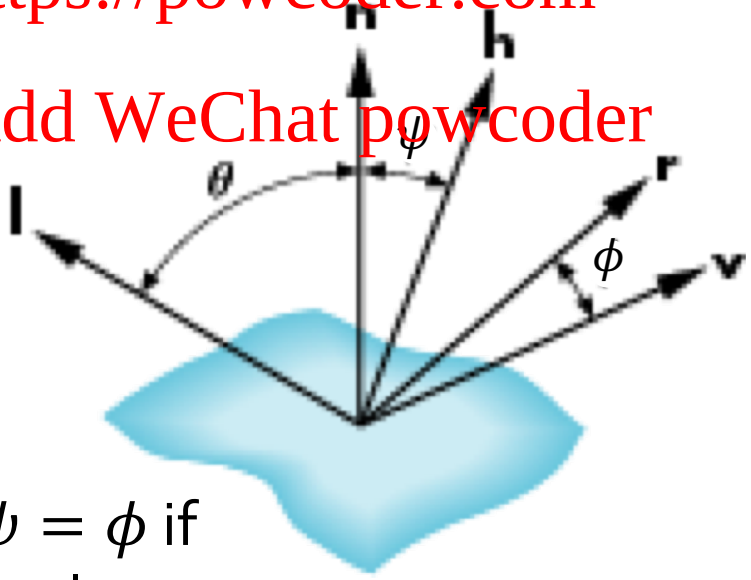
Modified Phong Model

- 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 unit vector halfway between **l** and **v**

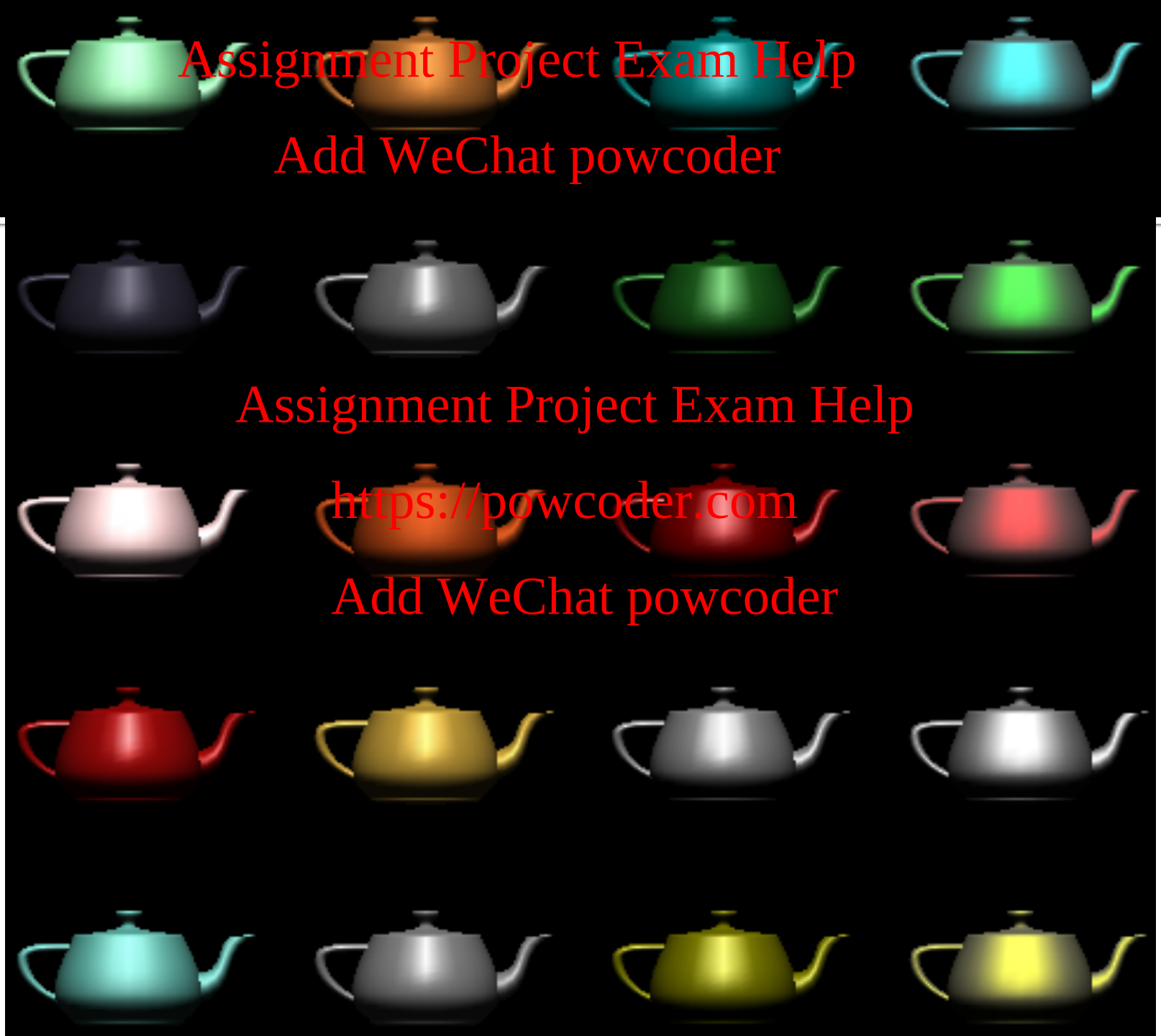
$$\mathbf{h} = \frac{\mathbf{l} + \mathbf{v}}{\|\mathbf{l} + \mathbf{v}\|}$$



Note that $2\psi = \phi$ if vectors are coplanar

Using the halfway vector

- Replace $(\mathbf{v} \cdot \mathbf{r})^{\alpha}$ with $(\mathbf{n} \cdot \mathbf{h})^{\beta}$
- Have to chose a new β that gives us the same results as the old α (if β is given the same value as α the specular highlights will be smaller)
- Resulting model is known as the modified Phong or Phong-Blinn lighting model


Assignment Project Exam Help
Add WeChat powcoder
Assignment Project Exam Help
https://powcoder.com
Add WeChat powcoder

Vertex Normals

- Lets write `calc_normal(verts, inds)` which calculates the normals for each vertex
- Process:
  - Start with all normals as <0,0,0>
  - Calculate the normal of each triangle and add this to each vertex's normal
  - Normalize all normals

Using Normals

- Start with the unlit-cube example
  - Draws a solid-colored cube using an element array and a triangle-strip
    - Color can be changed by user
    - Also includes a tetrahedron function for element array triangle-strip tetrahedron
  - Has all features of rotating-cube-perspective
  - Includes the full `calc_normals` function
- After we have all vertices and indices we get the array of normals with `calc_normals`
- Each vertex will have a normal associated with it
  - How do we associate a new value with each vertex?
  - How does the vertex shader give its value to the fragment shader?

Add Ambient Lighting

- We just need a new parameter for k_a
  - Make a new range or number input:
    - Initial value of 0.2
    - Goes from 0.0 to 1.0 with a step of 0.05 and
    - Has a label of `<i>k<sub>a</sub></i>`
  - Add a listener for it in the JS code that will send the value of k_a to the fragment shader when changed

- For the moment the light and object material colors will be the same for all of ambient, diffuse, and specular lighting

Light Movement

- Should the light move with the box or not? Depends on the situation!
  - What I am rotating/scaling/translating? Just the box or the entire world including the light?
- Model-view only applies to box:


```
L = light_src;
if (light_src.w == 1.0) L -= model_view*vPos;
L = normalize(L);
```

- Model-view applies to entire world:

```
L = model_view*light_src;
if (light_src.w == 1.0) L -= model_view*vPos;
L = normalize(L);
```

- We could embed this in an `if` with a `uniform bool`

Explore

- Some starting points:
  - What happens when you put a point light inside the box?
  - What happens when you translate/scale the box with a direction vs point light?
  - Can you create unrealistic views with light going where it shouldn't?
  - How does the distance of a point light source effect the lighting of the cube?
  - Can your material color and light color both be bright but the box still be black?
  - What amount of shininess is actually making a difference?

Fixing Problems

- A few problems:
  - Light "bends" around sharp edges due to interpolation of normal across face
  - Light intensity doesn't diminish with distance realistically
  - Specular light isn't as pronounced as it maybe should be

Flat Shading

- Check out flat shading in the latest code
- Compare effects of flat shading on diffuse vs. specular lighting
  - Explain

Other Additions

- Many systems:
  - Allow the material color to be different for specular, diffuse, and ambient light
    - We did fix the material color of specular to the light color but otherwise they are the same
  - Allow light color/brightness to be different for specular, diffuse, and ambient light
    - Would fix some issues we are having with brightness and ambient light with the current setup

Nonphotorealistic Shading

- Sometimes the goal is to not look real!
  - Maybe we want to be like a cartoon
- We can add a black outline by finding locations where the surface normal is perpendicular to the viewer direction
- We can "posterize" our shading by lighting up areas higher than a specific value and leaving things the same that are below that