Coding for Phong Lighting & Shading

Gouraud Shading

Phong Shading

Some slides modified from: David Kabala
Others from: Andries Van Damm, Brown Univ.

CAUTION! **WAY** too easy to get lost!

Shader Writing in GLSL:

- The bulk of your work in Project C;
  non-trivial vertex shader + fragment shader.
- **VERY unforgiving**: if your code is wrong, you see nothing; brief console err. msg.
  (be sure to open your web-browser's 'Console' to see that message!
  Right-Click(or Command-Click)→Inspect Element→select 'Console' tab.)
- ***No*** debugger! Not even a 'printf()' equivalent!
- ***THUS*** your prime strategy (and your only hope) is ***incremental development & version control:***
  Begin with a simple program that works. Improve it very slightly: test each and every new line of code, each tiny new step. After that step works, saving it as a higher-numbered version.

CAUTION! **WAY** too easy to get lost!

Build your Programs Incrementally!

- Spend all your time making and testing many tiny, quick improvements on a program that works perfectly, flawlessly.
- After each tiny improvement works, save a new version
- ***Never*** assemble a giant untested program,
  and then try to debug it. Hopeless: you may never fix it all!
- ***Big broken programs hide many big mistakes well (you may never find them all!)***
- ***Far faster to make*** *a long series of* ***tiny, tested improvements to*** *a really dumb simple* ***program that works*** *than to fix an 'almost finished' complicated program that doesn't.*
- Never waste too much time (>20-30 minutes) trying to fix a broken program. If its too puzzling, **STOP.**
  If last god version was 'ver027', save current as 'ver028BAD'.
  Copy 'ver027' to make 'ver029'. Make smaller improvements to ver029; you'll find the hidden bug!

CAUTION!   **WAY** too easy to get lost!

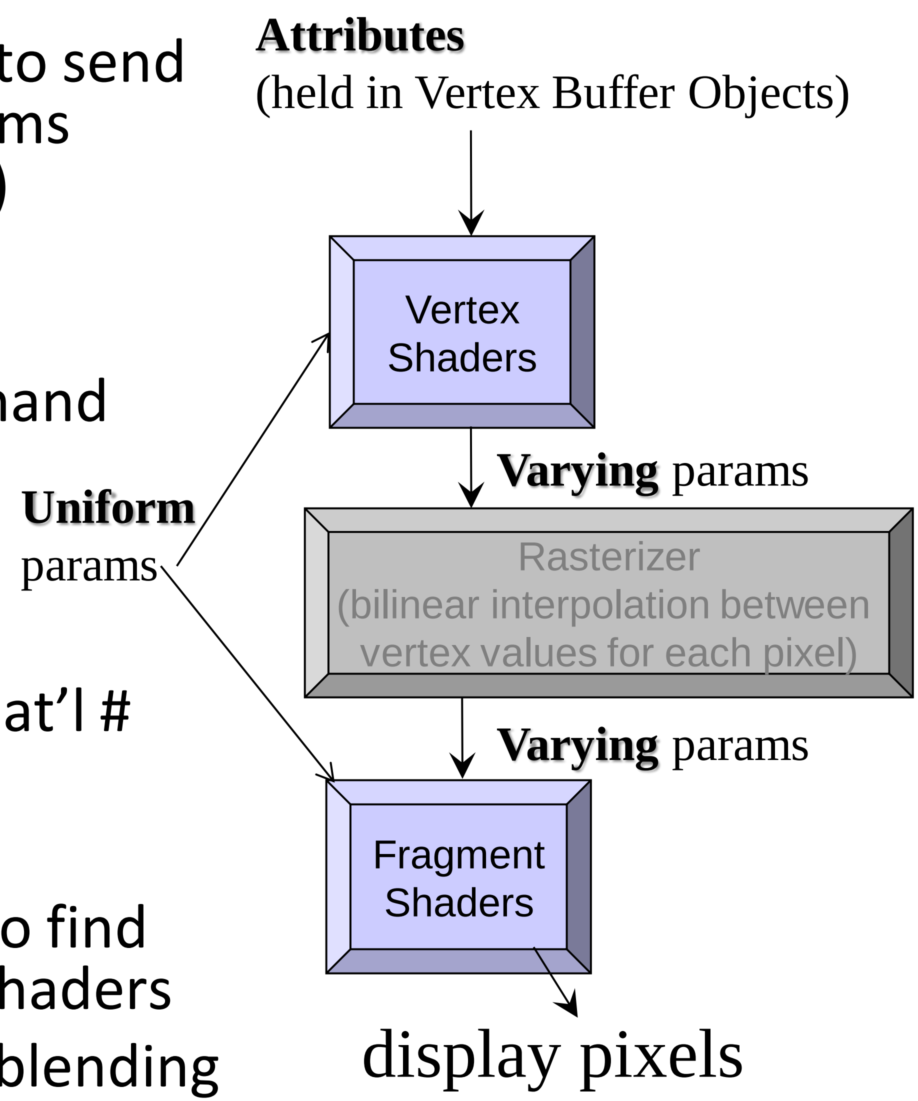
Version Control is Crucial.

- *Start with some simple code that you **KNOW** is working correctly. Doesn't matter how simple – even 'hello world' is good.*
- *Comment it first – write your INTENT for the code, your current plans, from start to finish, everything. This helps you find big problems before you code them, helps you think through the entire problem.*
- *Save that as your first version. Make sure it still works. Make a higher-numbered copy.*
- *Never modify your earlier versions –they're your record of current thinking, including mistakes, so you can find out 'what was I thinking?!?!' later.*
- *Save copious versions; memory is trivially cheap, but your time is not. I routinely write 10-12 progressively better versions for Project- C-sized programs*

RECALL:
How Shader Programs Communicate

Only 3 ways for your JavaScript code to send data ***into*** the GPU's shader programs (and no way to retrieve that data!)

- ***Uniform*** *parameters*
  - Set before each drawing command
  - Ex: modelView Matrix
- ***Attribute*** *parameters*
  - Set per vertex
  - Ex: position, surface normal, mat'l #
- ***Varying*** *parameters*
  - Passed from Vertex Shaders to rasterizer, which interpolates to find per-pixel values for fragment shaders
  - Ex: triangle with smooth color blending



Part 1: Gouraud Shading ("Goorr-Rowe")

one lighting calc per **Vertex**

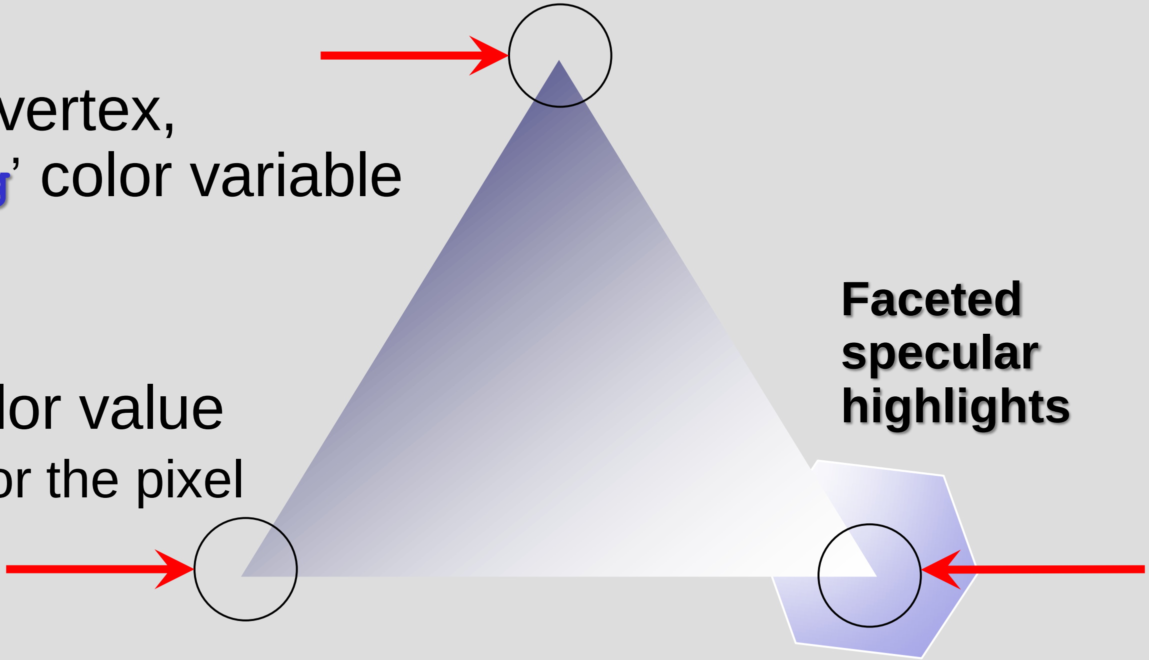
- **For each vertex,**
  compute on-screen RGB color
- **For each pixel,**
  bilinearly **interpolate** on-screen RGB color:

Vertex Shader

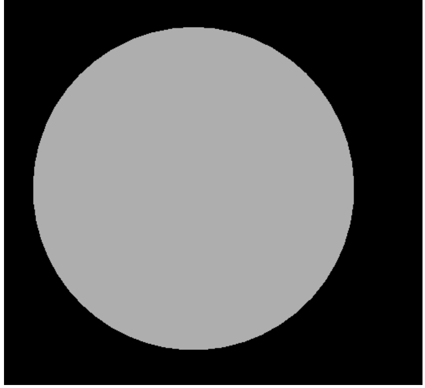
computes Phong lighting for vertex,
export result as one '`varying`' color variable

Fragment Shader

gets a rasterized `varying` color value
which sets `gl_fragColor` for the pixel

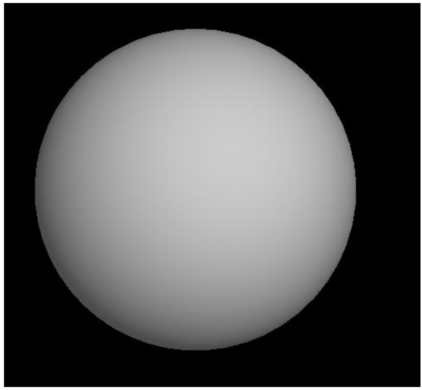


Gouraud Step-by-Step Goals:
1) Surface Normal Attributes



- Set dark-color background
  (e.g. (0.0,0.2,0.1) why? surfaces won't vanish)
- Just one object: unit sphere at world origin
- Sphere fills screen: put camera on world +z axis
- Add surface normal **attribute** to sphere vertices
- **TEST:** do your surface-normal attribute values actually arrive at your Vertex Shader?
  - How? In Vertex Shader, compute vertex color as (normal vector + 1,1,1)/2
  - send that color to Fragment Shaders as a 'varying' variable to interpolate colors between vertices
    (Gouraud shading!)
  - In Fragment Shader, use that 'varying' variable to set the final color of the pixel in gl_FragColor

Gouraud Step-by-Step Goals:
2) ADD Light Source Uniforms



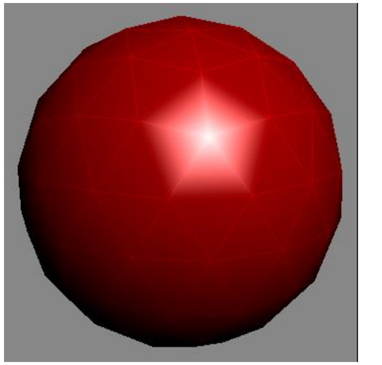
Add ‘uniform’ vars for one light source:

- At first, just the light-source position (convert from world to CAM coords in Javascript: don’t repeat the same transform for each vertex!)
- Vertex Shader:t ransform normals to CAM coords, compute (N·L), use result as color.
- Then add other light-source uniforms: Ia, Id, Is, etc. (RGB ambient, diffuse, specular illum; all [0-1])

TEST: do your light-source uniform values actually arrive at your Vertex Shader?

- Create ***one*** uniform, test it, then the next uniform…
- How? Let the newest uniform set the vertex color

Gouraud Step-by-Step Goals:
3) ADD Materials Uniforms



Add ‘uniform’ vars for just one material:

- Start with diffuse reflectance Kd;
- Vertex Shader: compute diffuse color: Kd*Id*Att*(N·L) (note entirely black shadows)
- Then add weak ambient lighting & reflectance (lightens the shadows: no longer entirely black)
- Then add specular term:
  - initially, try Se = 20, Ks = Is = (0.9, 0.9, 0.9)
  - GLSL-ES: use the ‘pow()’ and ‘reflect()’ functions
  - Note the faceted, hexagonal specular highlights!

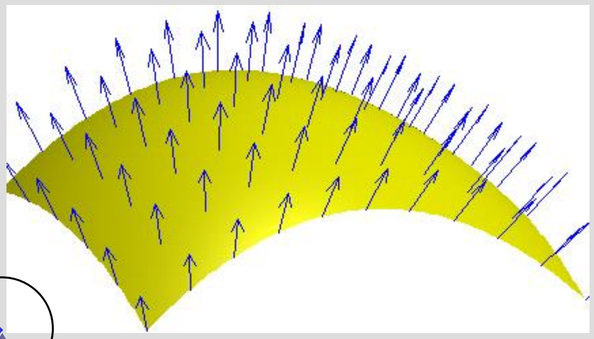
TEST: do your Materials uniform values actually arrive at your Vertex Shader?

- Create ***one*** uniform, test it, then the next uniform…
- How? Let the newest uniform set the vertex color

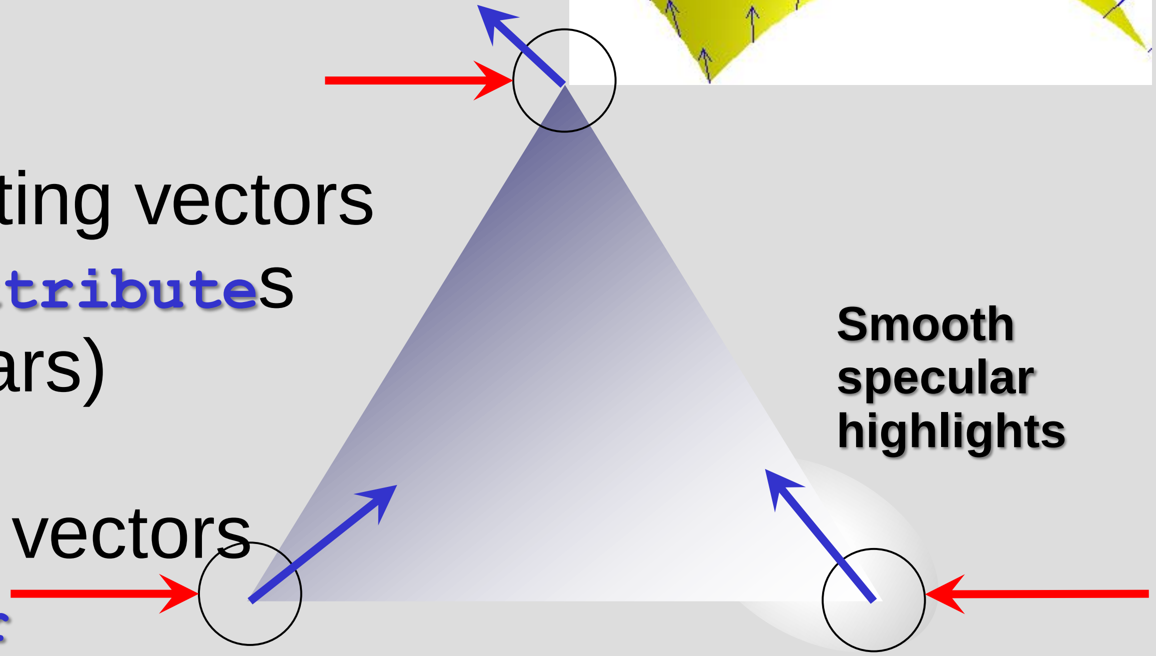
Phong Shading

One lighting calc per **Pixel**

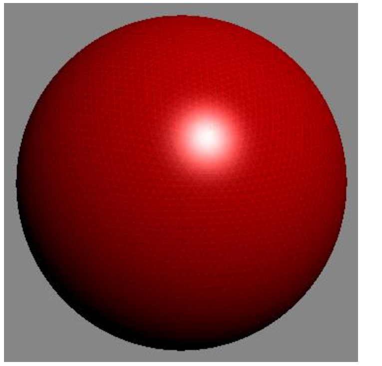
- **For each vertex,**
  compute lighting vectors (**N**orm,**L**ight,**V**iew)
- **For each pixel,**
  bilinearly **interpolate** vectors,
  compute lighting, set color



Vertex Shader
computes ‘`varying`’ lighting vectors
(from vertex `attribute`s
&/or `uniform` vars)
Fragment Shader
gets rasterized `varying` vectors
computes `gl_fragColor`



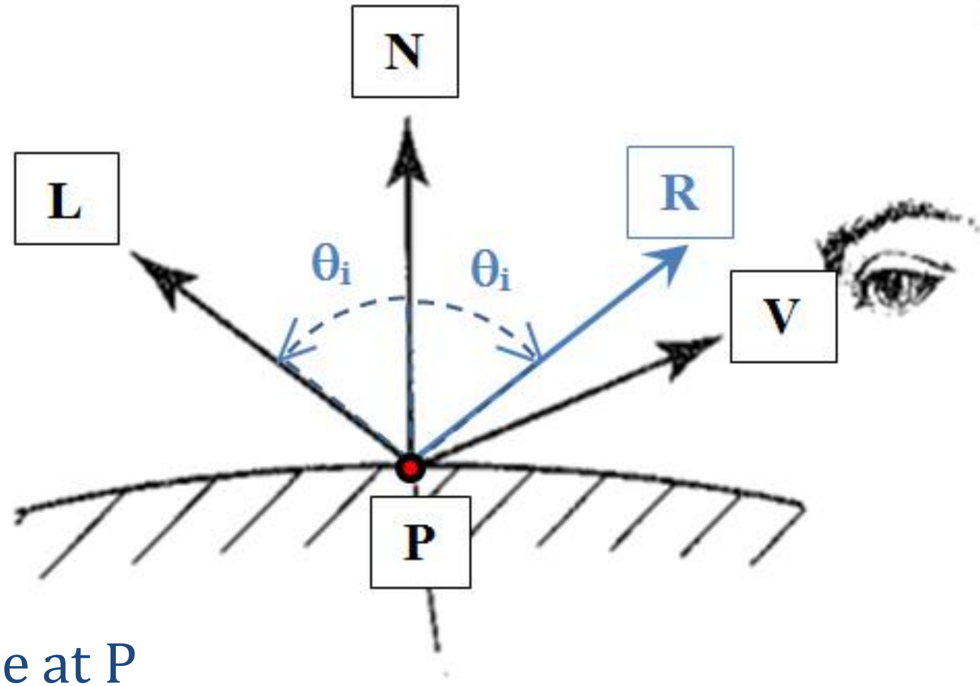
Phong Shading Step-by-Step Goals:
4) Per-pixel Vectors



- Add new 'varying' vars to Vertex Shader to interpolate all vectors for per-pixel lighting
- Move lighting calcs to Frag Shader. Diffuse 1st:
  - Start by adding a 'varying' var to interpolate the surface normal N, and the vertex position P.
  - In Fragment Shader, find unit light direction vec L (light-position uniform - vertex position varying)
  - In Fragment Shader, compute (N·L) to set color.
- Then specular:
  - Compute reflected direction R or half-vector H (test it – use R or H to set sphere color, move light)
  - Complete the specular term: compare results to earlier Gouraud-shaded version – they should look similar, but Phong Shading highlights are round with no faceting: looks perfectly smooth, flawless

Phong Lighting
Step 1: Find Scene Vectors

To find ***On-Screen RGB Color***
at point ***P (start):***



1) Find all 3D scene vectors first:

   a) Light Vector **L**:
      unit vector towards light source

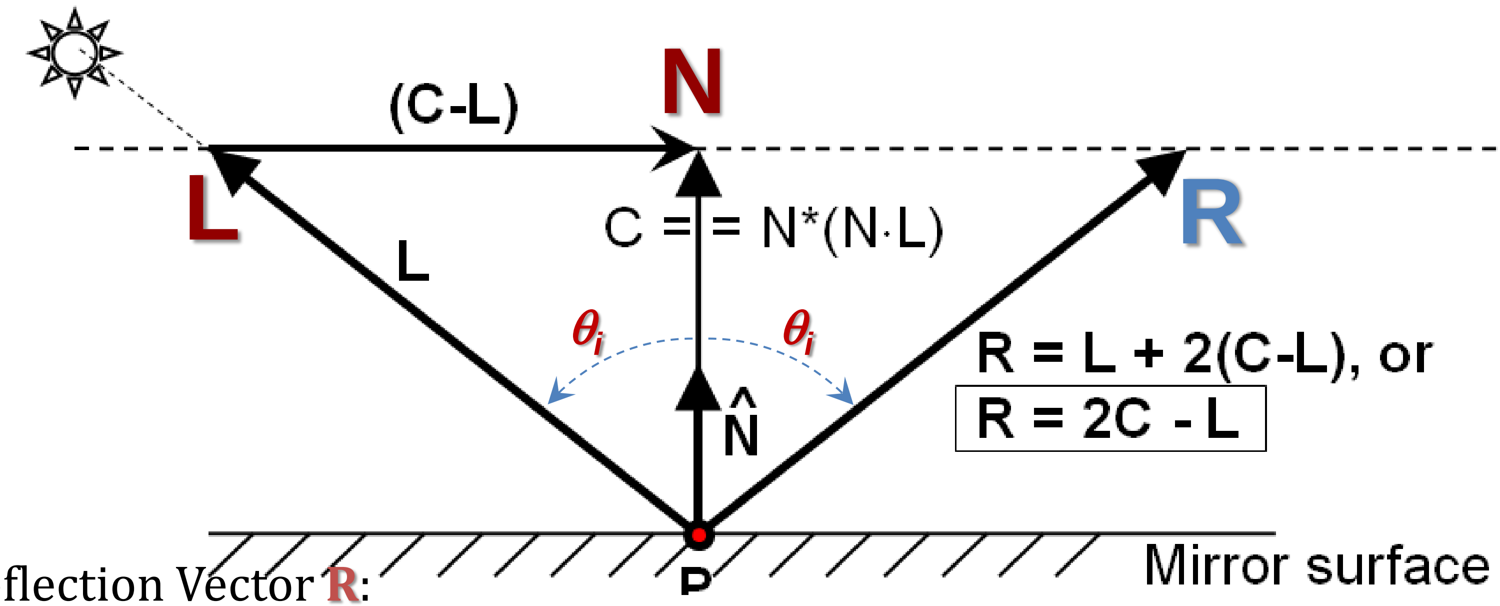
   b) Normal Vector **N**:
      unit vector perpendicular to surface at P

   c) View Vector **V**:
      unit vector towards camera eye-point

On to step 2: how do we find the Reflected-light Vector **R**?

Phong Lighting
Step 2: Find reflection Vector R

To find ***On-Screen RGB Color*** at point ***P (cont'd):***



2) COMPUTE the Light Reflection Vector **R**:

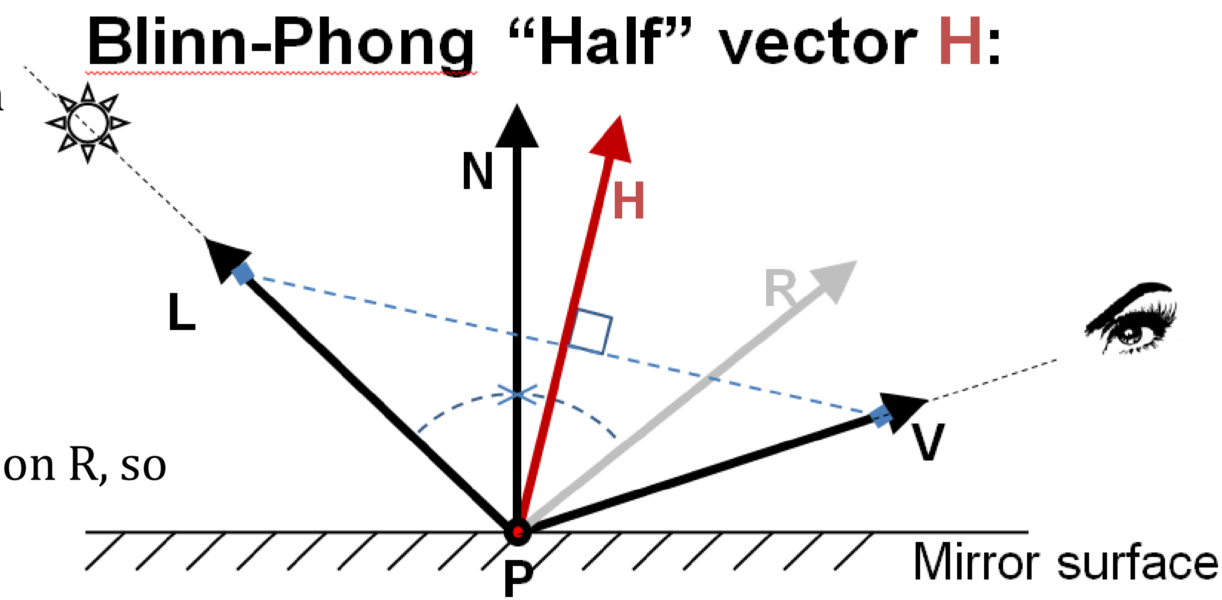
- Given unit light vector **L**, find lengthened normal **C**
  C = N (L·N)
- In diagram, if we add vector 2*(C-L) to L vector we get R vector. Simplify:
  R = 2C – L
- **Result:** unit-length **R** vector GLSL-ES→ See built-in '`reflect()`' function
  (If **N** is a unit-length vector, then **R** vector length matches **L** vector length)

Blinn-Phong Lighting
Fast (but Approximate) Specular Reflection

- Skip reflection-vector ***R*** calculation
- Instead, define the 'half angle' ***H***:

  $$\boldsymbol{H} = \frac{\boldsymbol{L}+\boldsymbol{V}}{|\boldsymbol{L}+\boldsymbol{V}|}$$

  'halfway' between light and eye.
- H==N when eye aligns with reflection R, so



- Replace Phong specular term **(L·R)**^{Se} with Phong-Blinn specular term **(N·H)**^{Se}
- CAREFUL! must have equal-length L,V (e.g. normalize both L and V first)
- Should we use Phong or Blinn-Phong?
  - Blinn-Phong slightly simpler, faster to compute
  - slight difference on-screen, but hard to see
  - implemented in original OpenGL for simplicity C

Phong Lighting
Step 3: Gather Light & Material Data

To find ***On-Screen RGB Color***
at point ***P (cont'd):***

3) For each light source, gather:

- RGB triplet for **Ambient** Illumination **Ia** $0 \leq Iar, Iag, Iab \leq 1$
- RGB triplet for **Diffuse** Illumination **Id** $0 \leq Idr, Idg, Idb \leq 1$
- RGB triplet for **Specular** Illumination **Is** $0 \leq Isr, Isg, Isb \leq 1$

For each surface material, gather:

- RGB triplet for **Ambient** Reflectance **Ka** $0 \leq Kar, Kag, Kab \leq 1$
- RGB triplet for **Diffuse** Reflectance **Kd** $0 \leq Kdr, Kdg, Kdb \leq 1$
- RGB triplet for **Specular** Reflectance **Ks** $0 \leq Kar, Kag, Kab \leq 1$
- RGB triplet for **Emissive** term(often zero) **Ke** $0 \leq Ker, Keg, Keb \leq 1$
- Scalar 'shinyness' or 'specular exponent' term **Se** $1 \leq Se \leq \sim 100$

Phong Lighting
Step 4: Sum of Light Amounts

To find ***On-Screen RGB Color***
at point ***P (cont'd):***

sum of each kind of light at ***P***:
Phong Lighting = Ambient + Diffuse + Specular + Emissive
SUMMED for all light sources

4) For the i-th light source, find:

RGB= **Ke** +	// 'emissive' material; it glows!
Ia***Ka** +	// ambient light * ambient reflectance
Id***Kd*****Att***max(0,(**N**·**L**))	// diffuse light * diffuse reflectance
Is***Ks*****Att***(max(0,**R**·**V**))^{Se},	// specular light * specular reflectance

- Distance Attenuation scalar: $0 \leq$ **Att** ≤ 1
  - Fast, OK-looking default value: **Att =** 1.0
  - Physically correct value: **Att(d)** = 1/(**d**istance to light)^2 (too dark too fast!)
  - Faster, Nice-looking 'Hack': **Att(d)** = 1/(**d**istance to light)
  - OpenGL compromise: **Att(d) = min(1, 1/(c1 + c2*d + c3*d^2))**
- 'Shinyness' or 'specular exponent' $1 \leq Se \leq \sim100$ (large for sharp, small highlights)

!END!