Intermediate Graphics & Animation Programming

GPR-300
Daniel S. Buckstein

Lighting & Shading Algorithms
Week 2

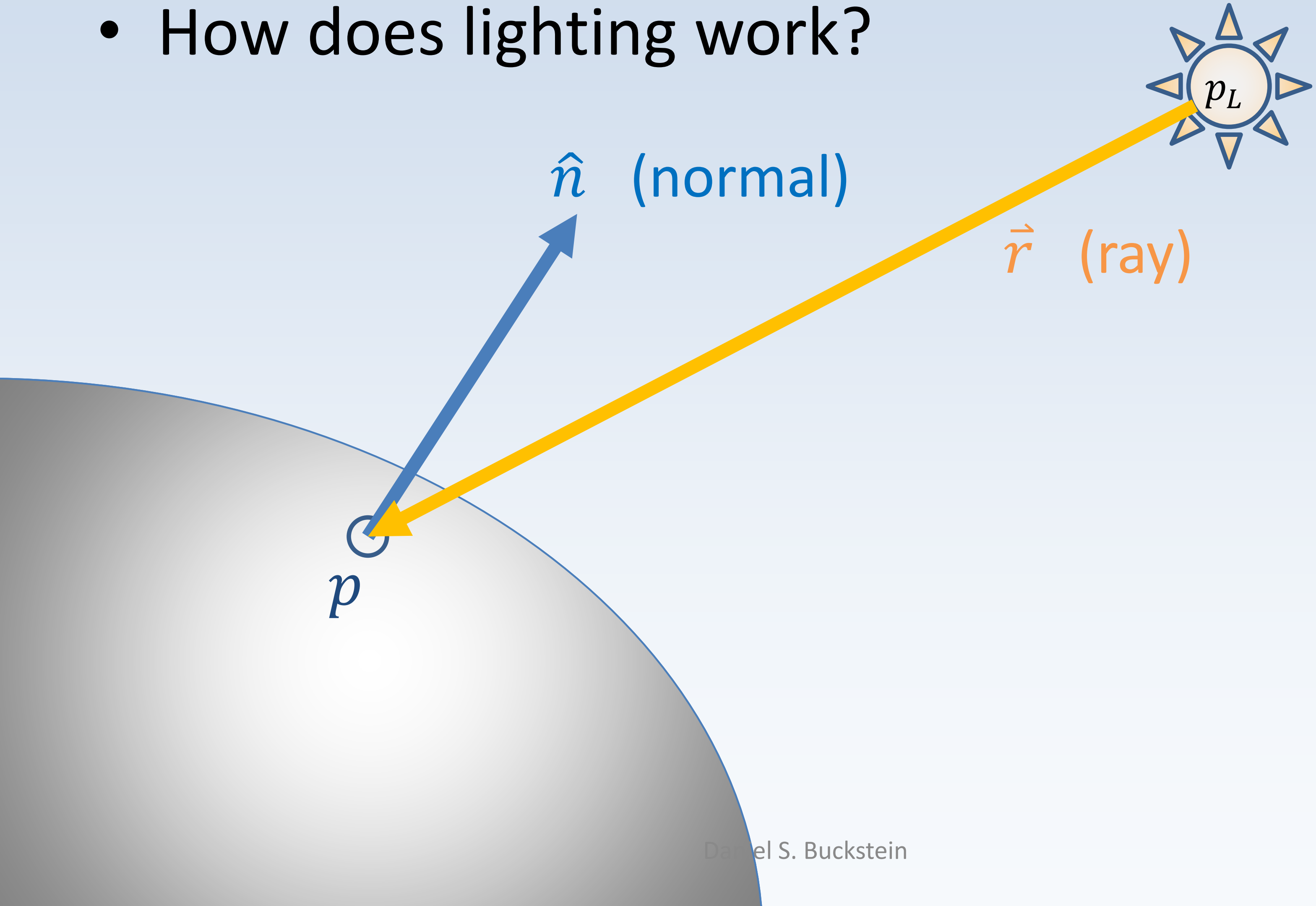
License

Lighting & Shading Algorithms

- Types of lights: point, directional, area
- Lighting and shading: Diffuse, Lambert, Phong
- Lighting precision: face, vertex, fragment
- Spaces: object, world, eye, screen
- Non-photorealistic shading: cel, gooch

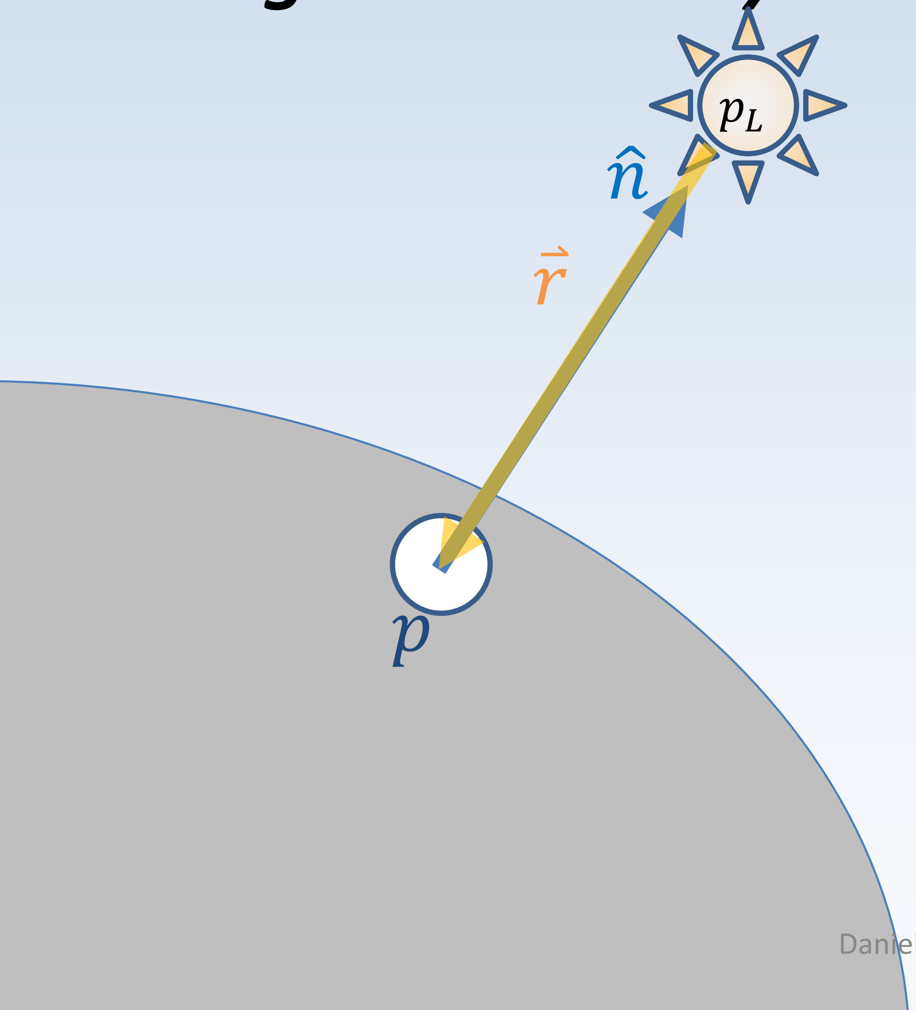
Lighting Algorithms

- How does lighting work?



Lighting Algorithms

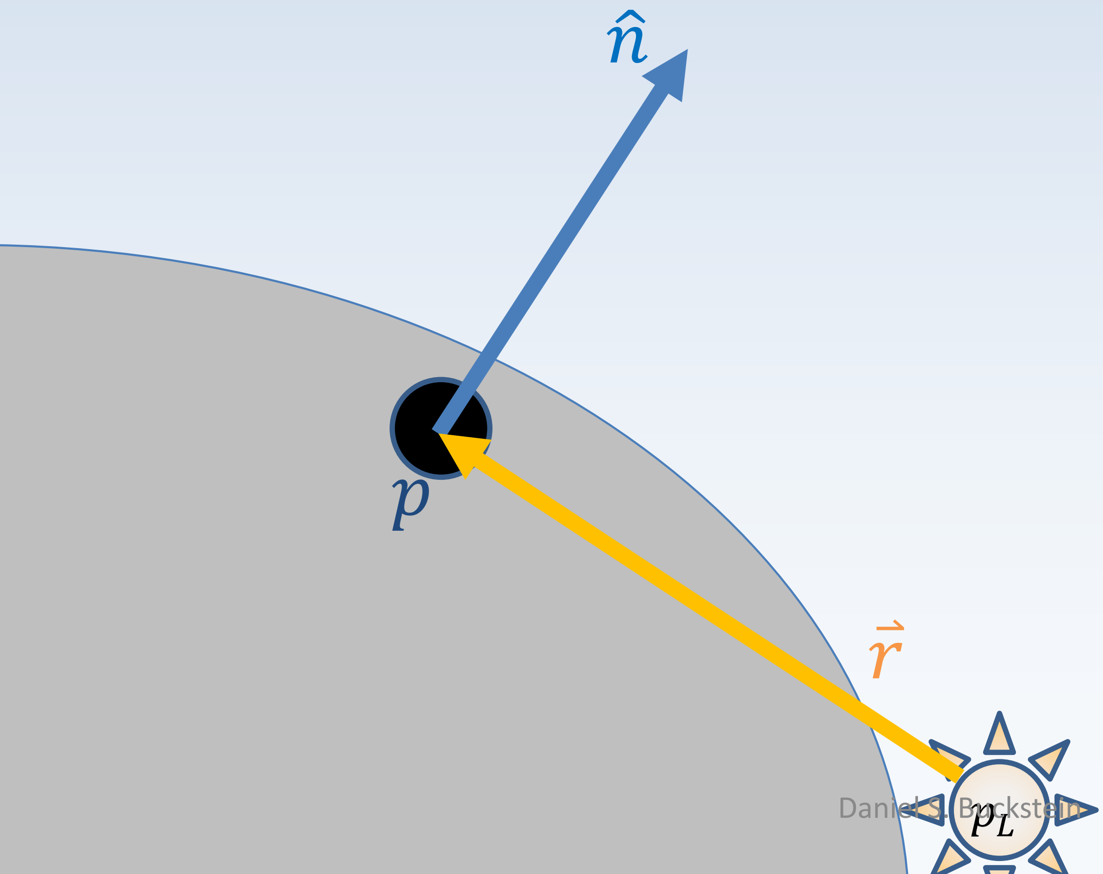
- ***Highest intensity***: normal and ray are *parallel*



(aligned or pretty close)

Lighting Algorithms

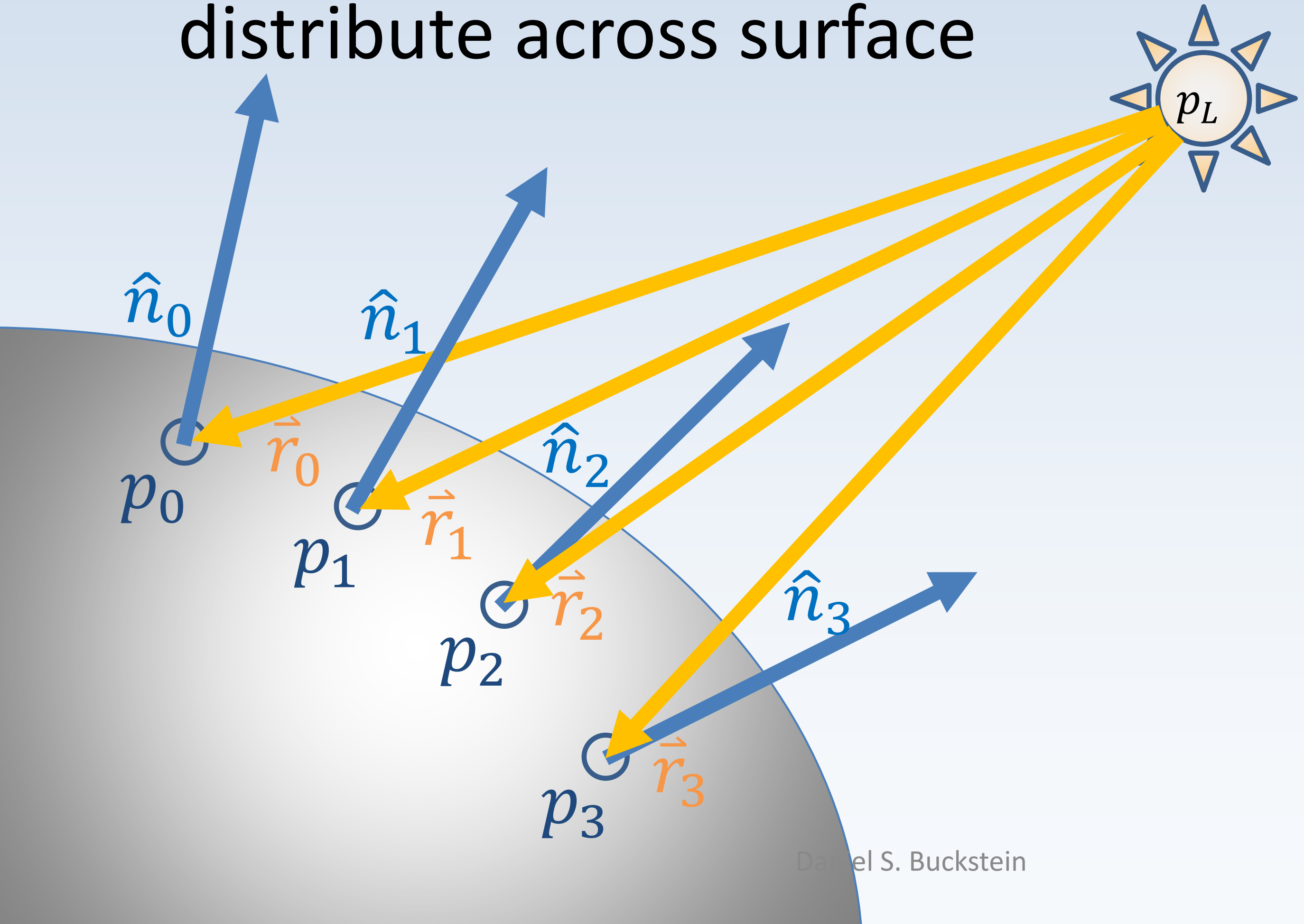
- ***Lowest intensity***: normal and ray are *perpendicular*



(not even close)

Lighting Algorithms

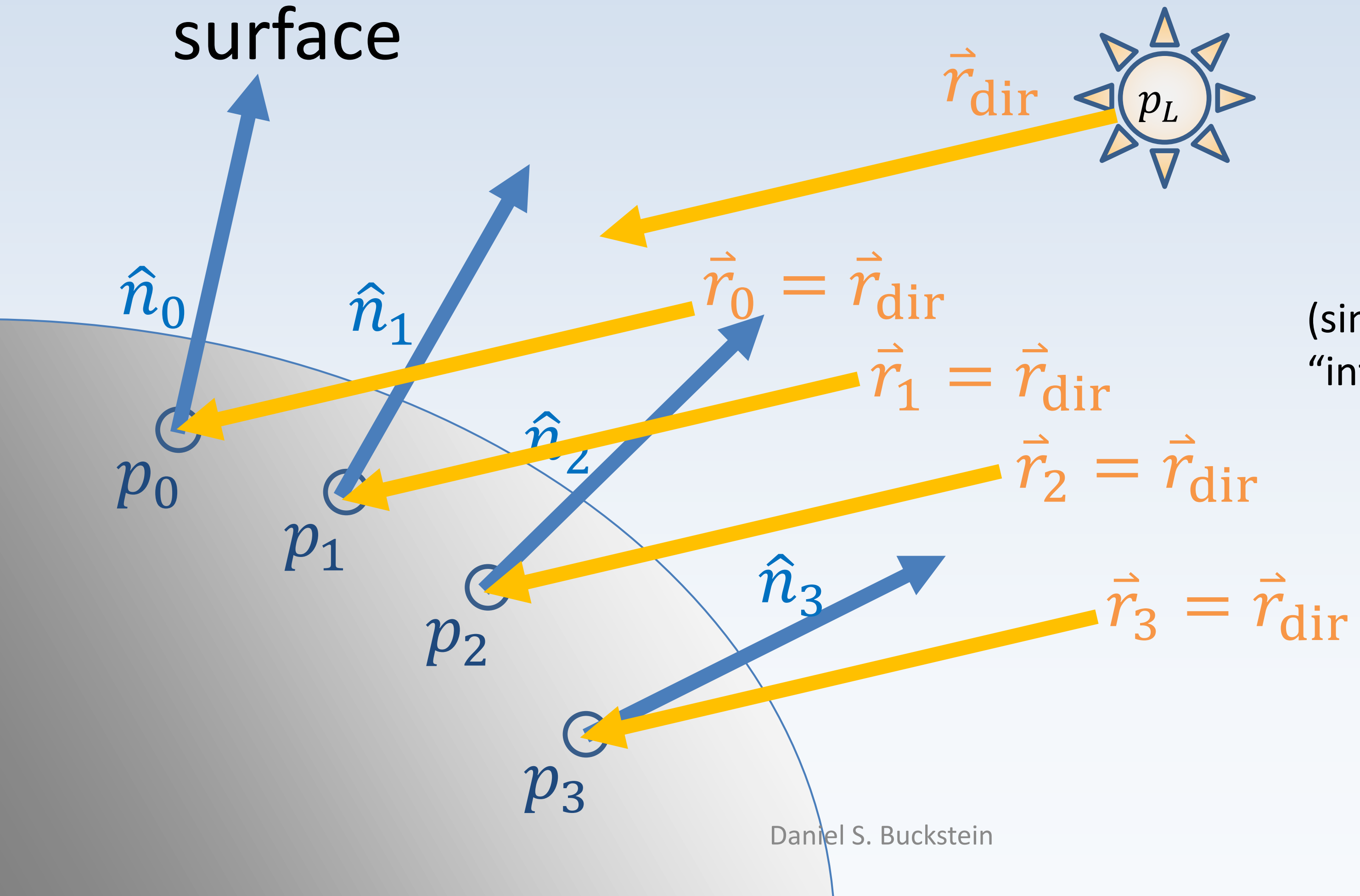
- ***Point light***: rays *directly from light source* distribute across surface



(example: a light bulb)

Lighting Algorithms

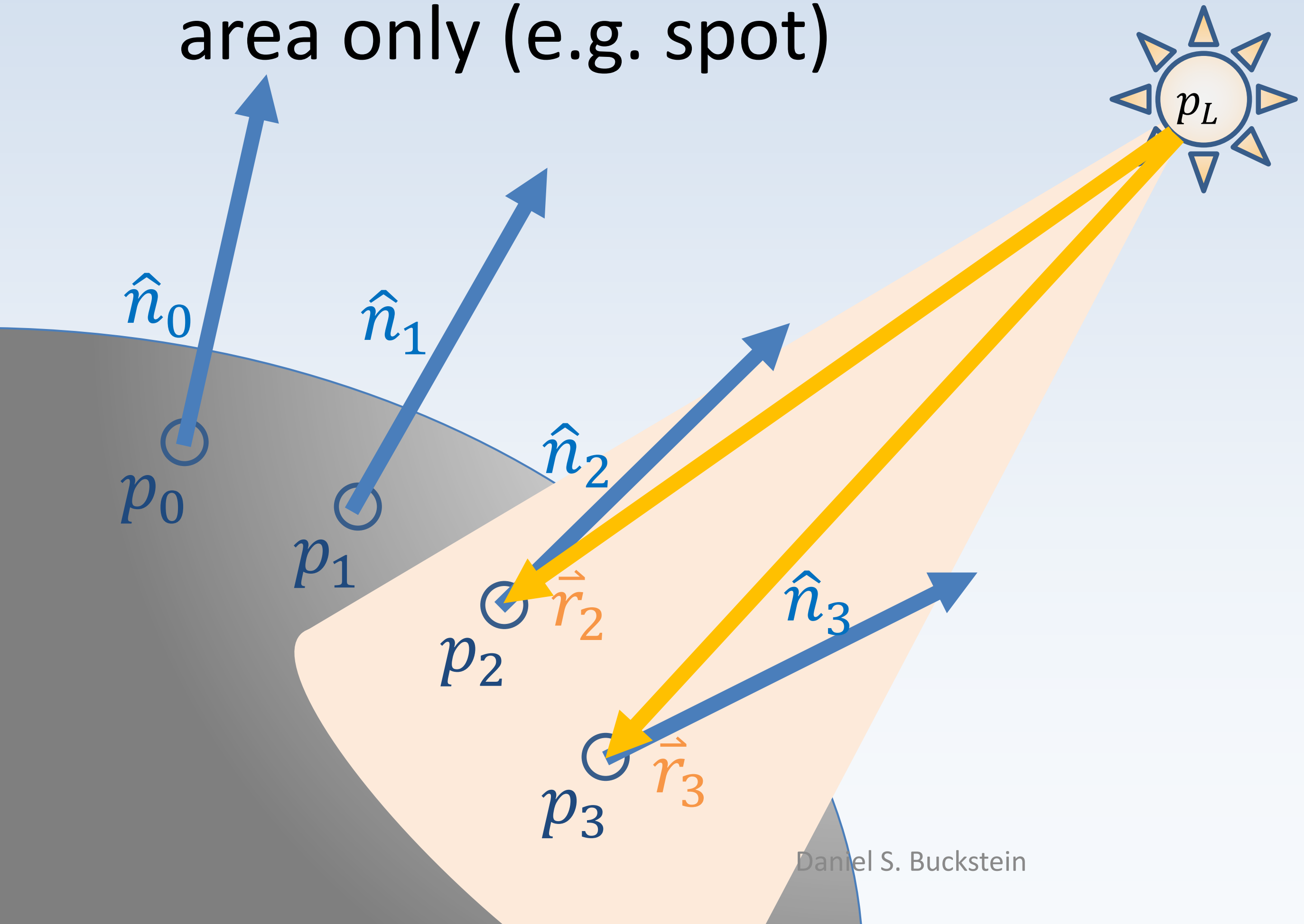
- ***Directional light***: rays *always the same* across surface



(simulates a light source "infinitely" far away)

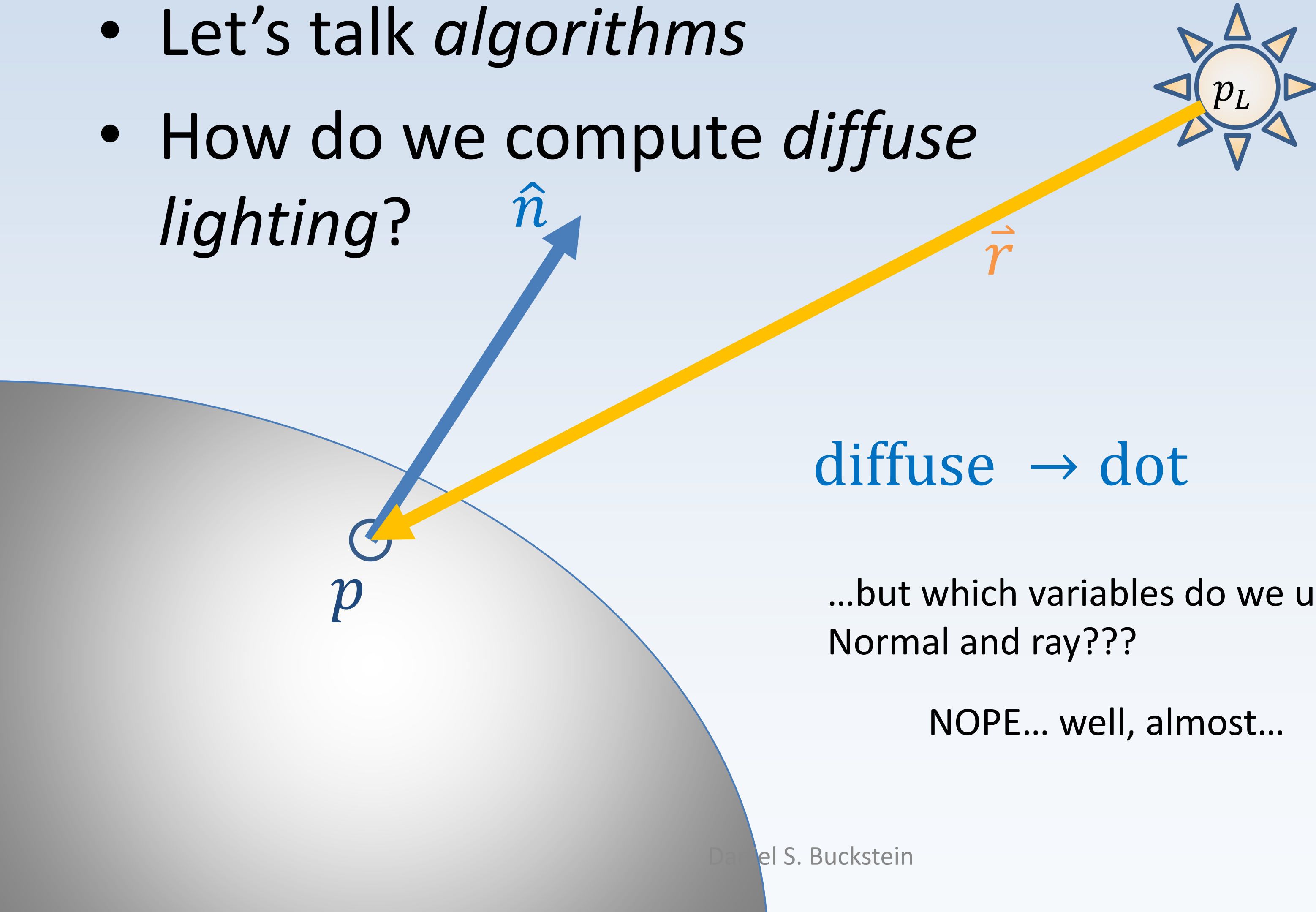
Lighting Algorithms

- ***Area light***: rays affect surface contained in area only (e.g. spot)



Lighting Algorithms

- Let's talk *algorithms*
- How do we compute *diffuse lighting*?



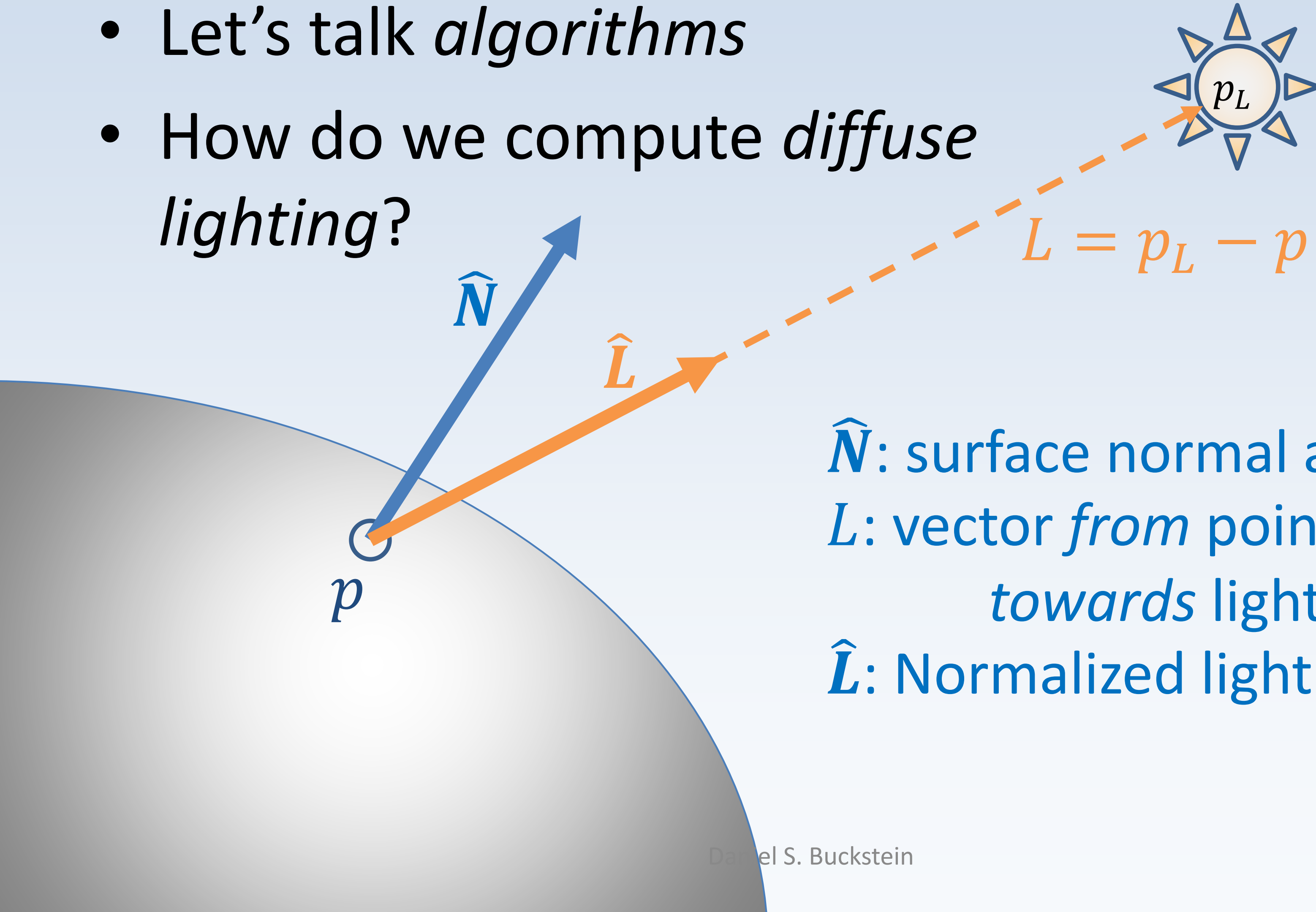
$\text{diffuse} \rightarrow \text{dot}$

…but which variables do we use?
Normal and ray???

NOPE… well, almost…

Lighting Algorithms

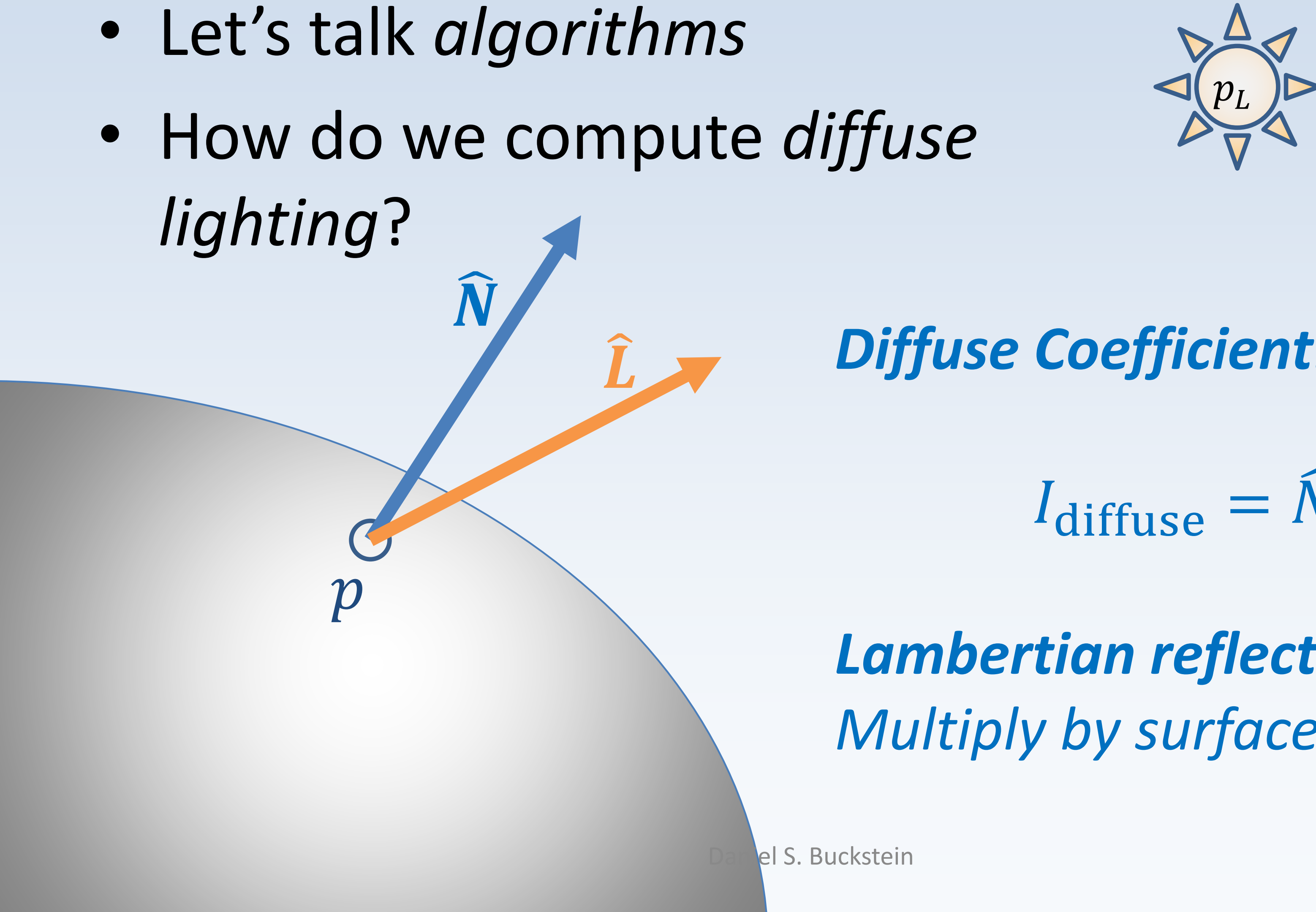
- Let's talk *algorithms*
- How do we compute *diffuse lighting*?



$\hat{\boldsymbol{N}}$: surface normal at point p
L: vector *from* point p *towards* light source p_L
$\hat{\boldsymbol{L}}$: Normalized light vector

Lighting Algorithms

- Let's talk *algorithms*
- How do we compute *diffuse lighting*?



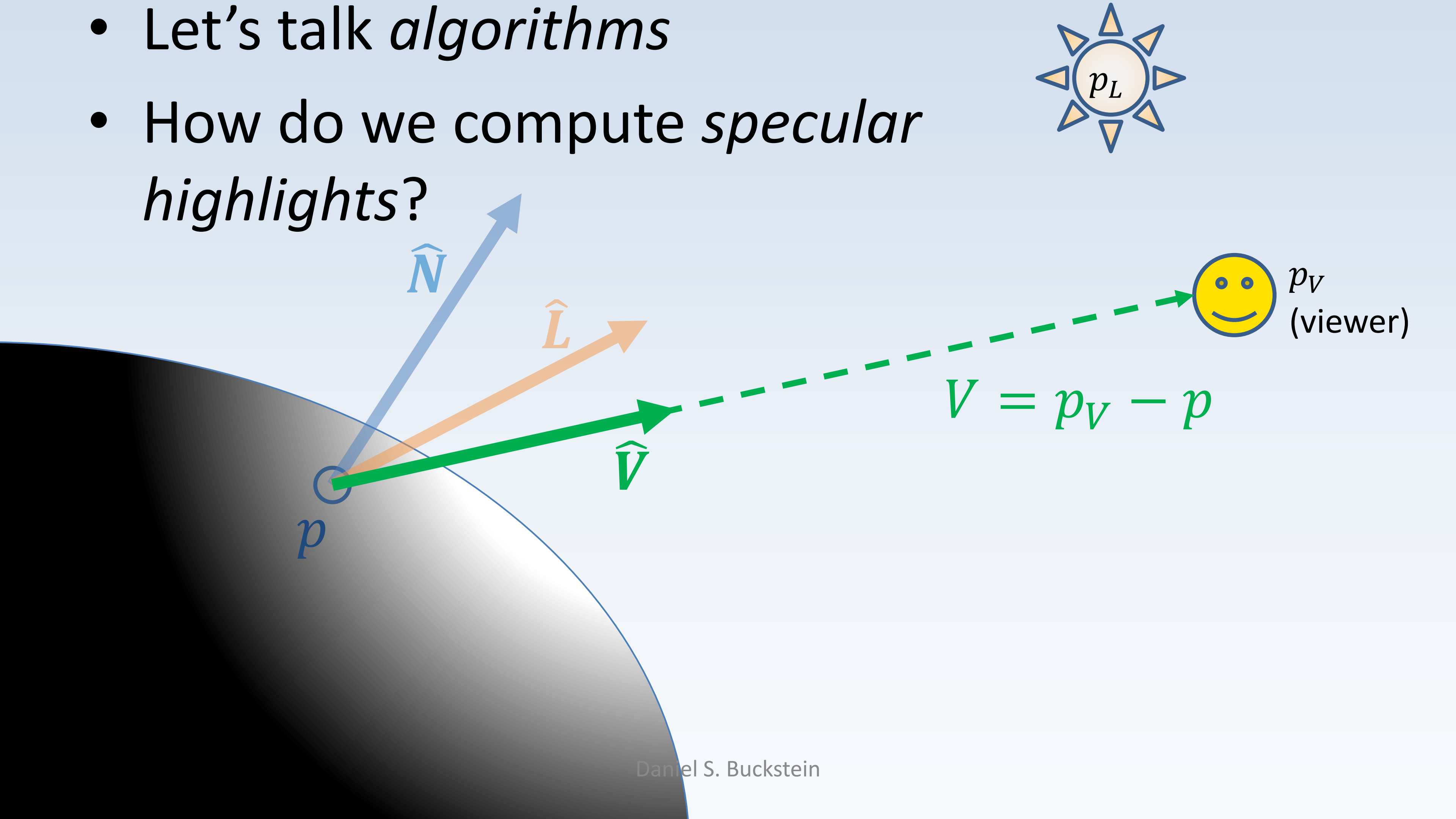
Diffuse Coefficient:

$$I_{\text{diffuse}} = \hat{N} \cdot \hat{L}$$

Lambertian reflection model:
Multiply by surface color!

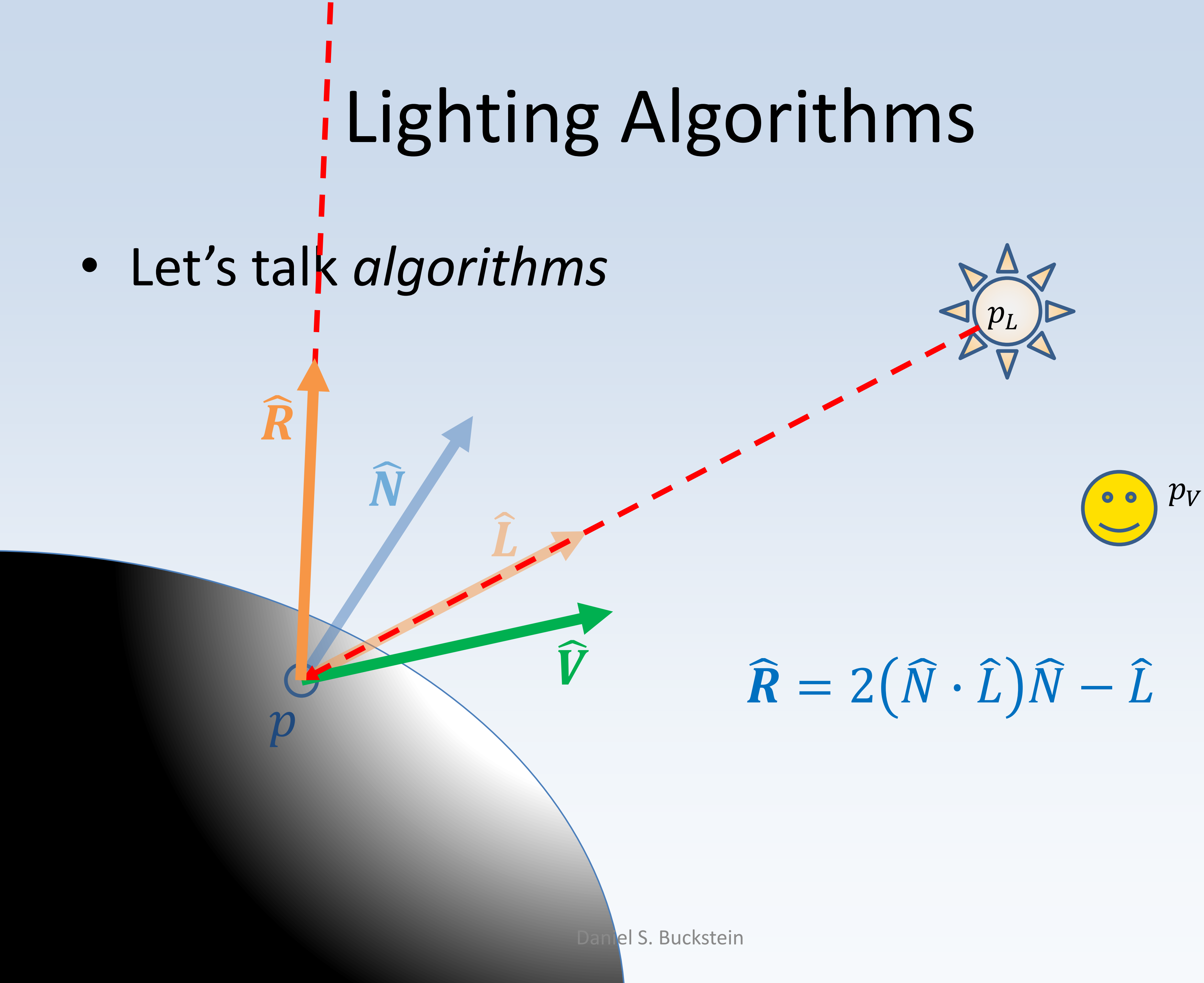
Lighting Algorithms

- Let's talk *algorithms*
- How do we compute *specular highlights*?



Lighting Algorithms

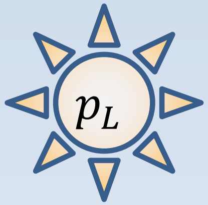
- Let's talk *algorithms*

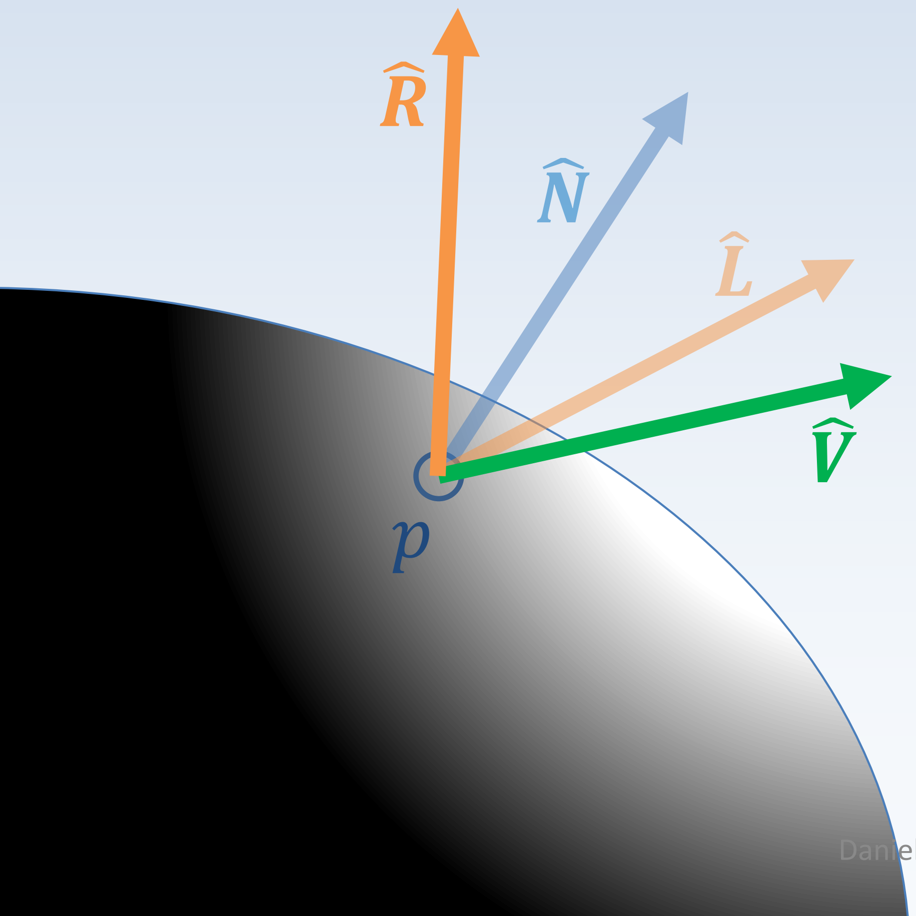


$$\hat{\boldsymbol{R}} = 2\left(\hat{N} \cdot \hat{L}\right)\hat{N} - \hat{L}$$

Lighting Algorithms

- Let’s talk *algorithms*



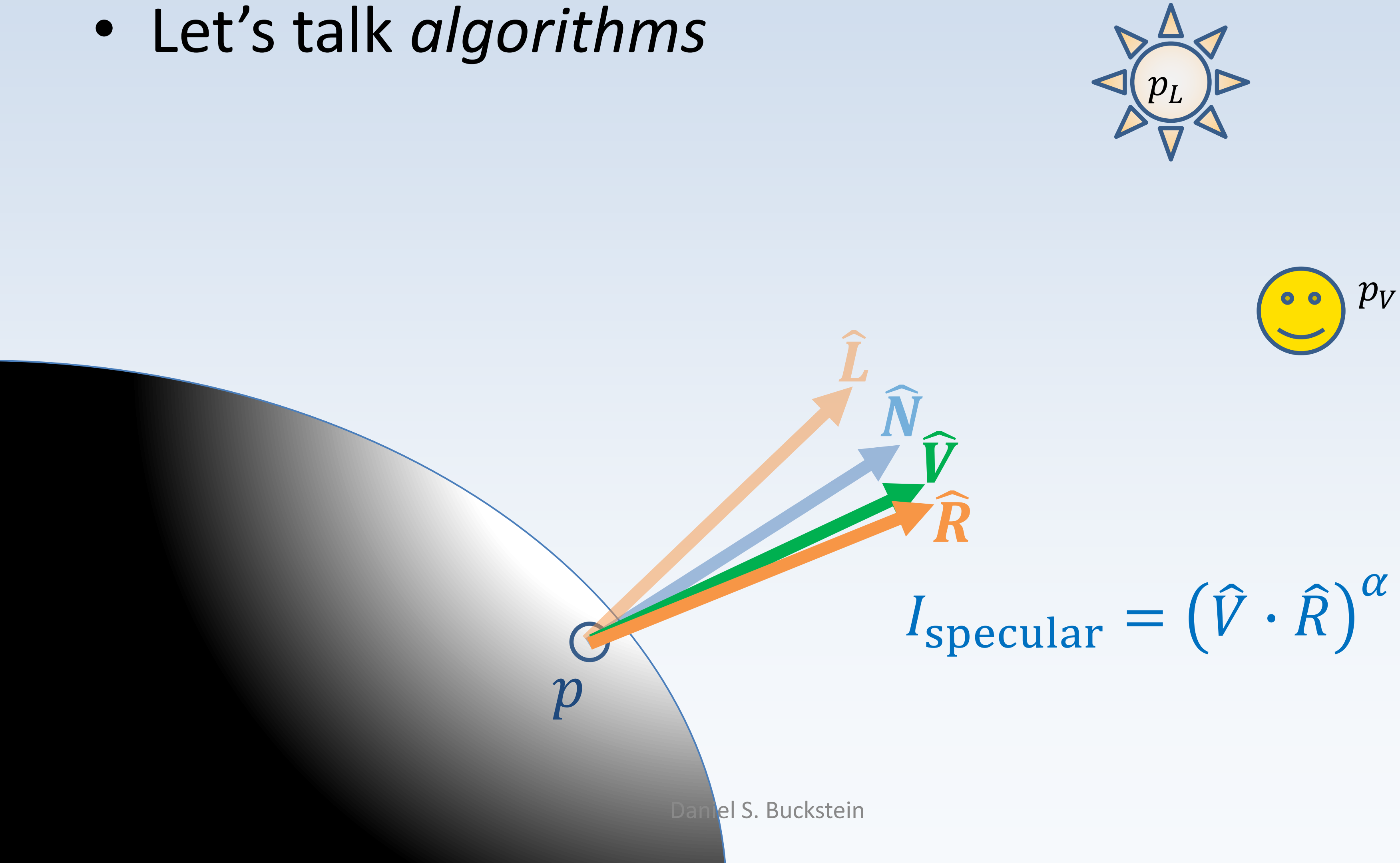


Specular coefficient:

$$I_{\text{specular}} = \left(\hat{V} \cdot \hat{R}\right)^{\alpha}$$

Lighting Algorithms

- Let’s talk *algorithms*



$$I_{\text{specular}} = \left(\hat{V} \cdot \hat{R}\right)^{\alpha}$$

Lighting Algorithms

- ***Phong reflection model***:
- Sum of diffuse, specular and ambient:

$$
\begin{aligned}
I_{\text{Phong}} &= \text{diffuse} + \text{specular} + \text{ambient} \\
&= k_d I_{\text{diffuse}} + k_s I_{\text{specular}} + k_a \\
&= k_d\left(\hat{N} \cdot \hat{L}\right) + k_s\left(\hat{V} \cdot \hat{R}\right)^{\alpha} + k_a
\end{aligned}
$$

Lighting Algorithms

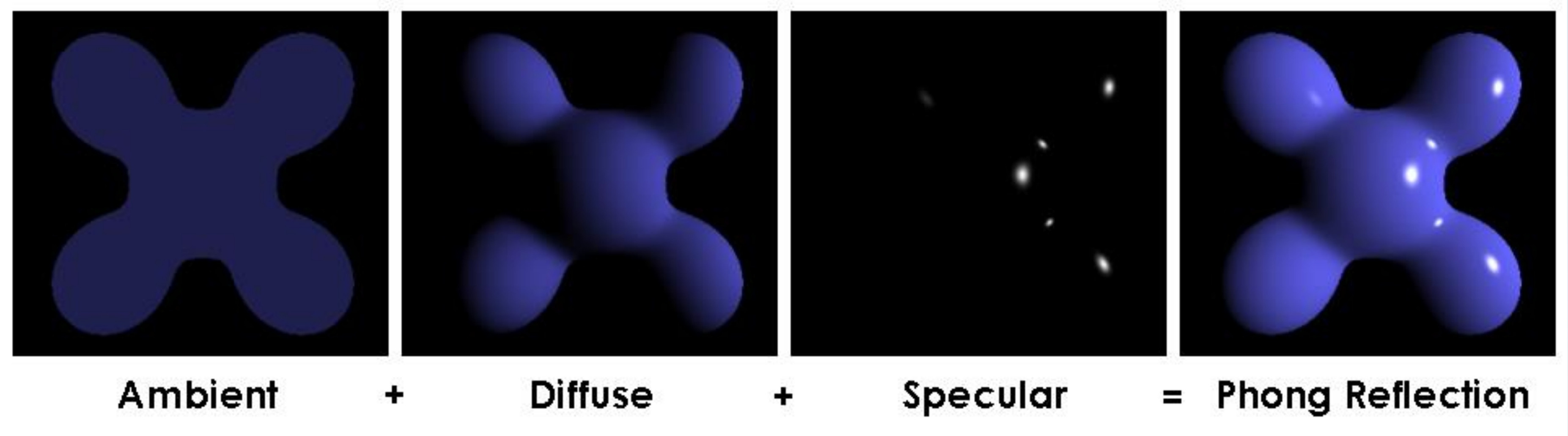
- ***Phong reflection model***:
- Similar formula for multiple lights

$$I_{\text{Phong}} = \sum_{m \,\in\, \text{lights}} (\text{diffuse}_m + \text{specular}_m) + k_a$$

$$= \sum_{m \,\in\, \text{lights}} \left(k_{d_m}(\hat{N} \cdot \hat{L}_m) + k_{s_m}(\hat{V} \cdot \hat{R}_m)^{\alpha} \right) + k_a$$

Lighting Algorithms

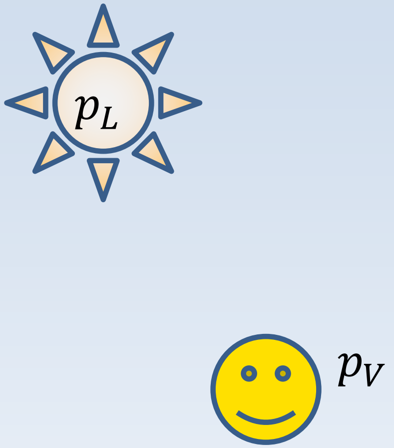
- ***Phong reflection model***:

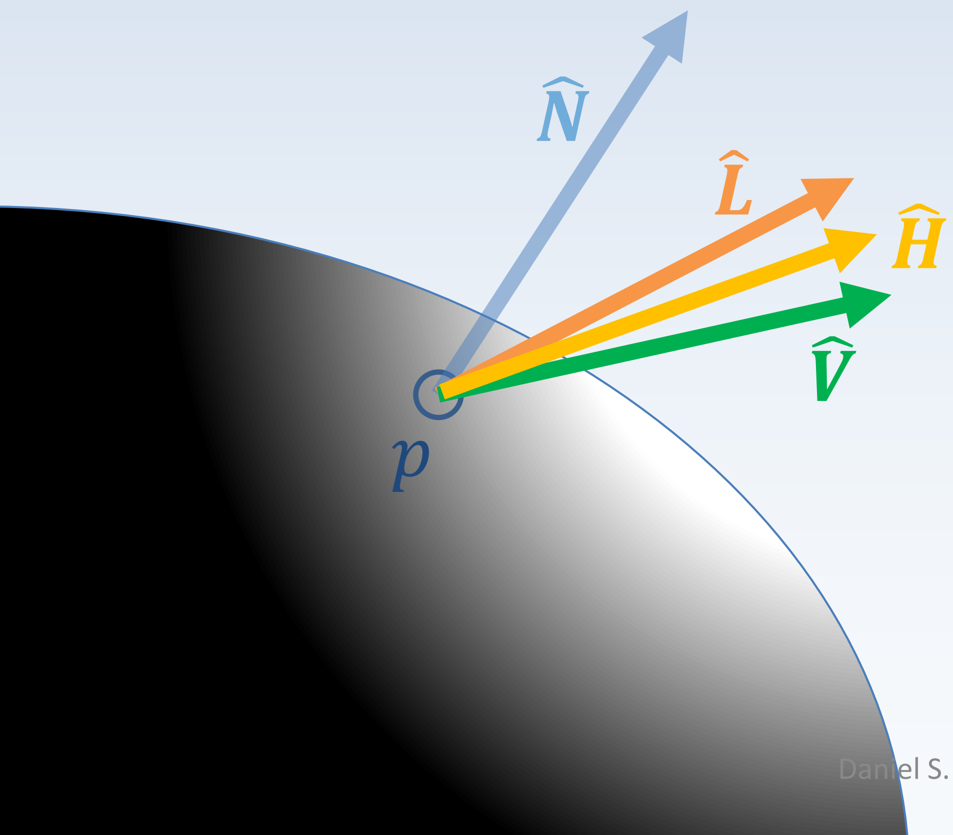


https://en.wikipedia.org/wiki/Phong_reflection_model

Lighting Algorithms

- Phong has some improvements!
- ***Blinn-Phong shading model***:





Half-vector:

$$\hat{H} = \frac{\hat{L} + \hat{V}}{\left\| \hat{L} + \hat{V} \right\|}$$

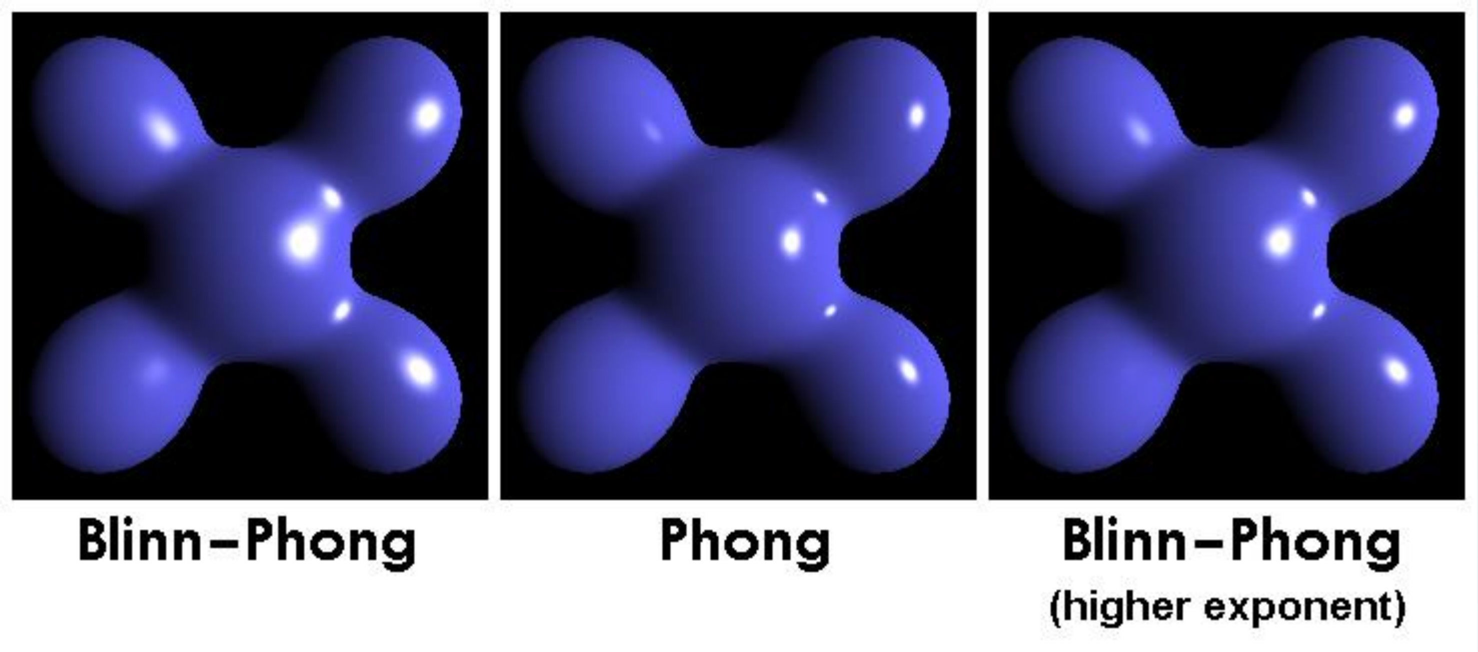
Lighting Algorithms

- ***Blinn-Phong shading model***:
- Sum of diffuse, specular and ambient, where the specular component is a bit different:

$$
\begin{aligned}
I_{\text{Phong}} &= \text{diffuse} + \text{specular} + \text{ambient} \\
&= k_d I_{\text{diffuse}} + k_s I_{\text{specular}} + k_a \\
&= k_d\left(\hat{N} \cdot \hat{L}\right) + k_s\left(\hat{N} \cdot \hat{H}\right)^{4\alpha} + k_a
\end{aligned}
$$

Lighting Algorithms

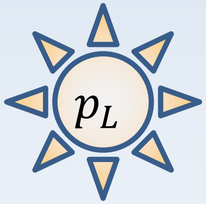
- ***Blinn-Phong shading model***:



https://en.wikipedia.org/wiki/Blinn-Phong_shading_model

Lighting Precision

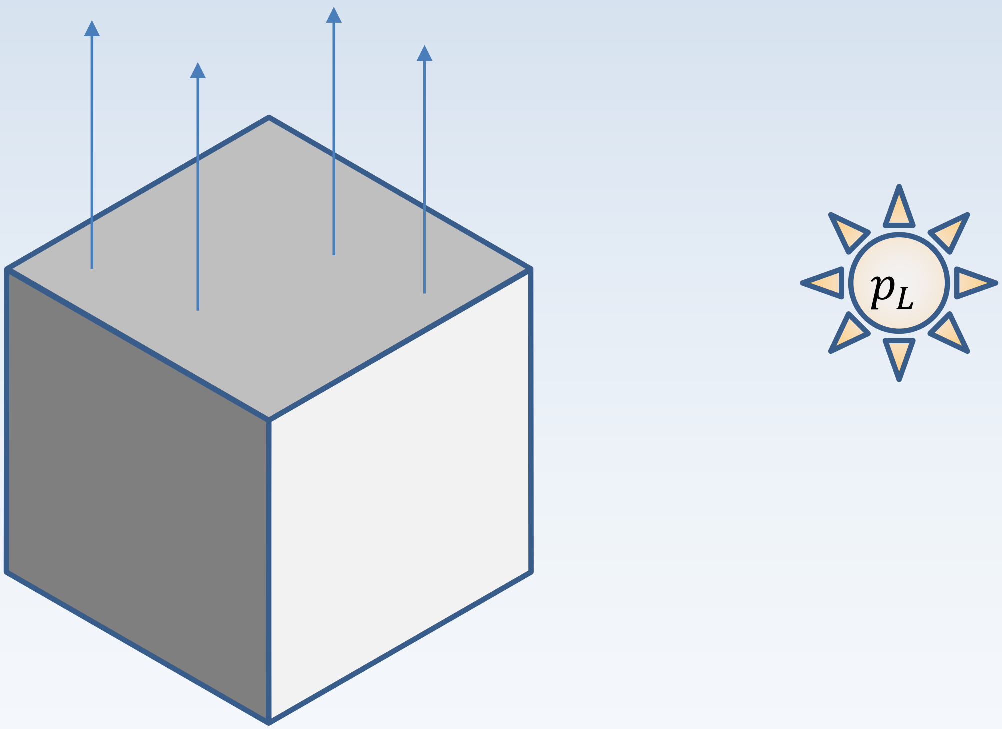
- Depending on ***when*** and ***how*** we perform lighting calculations, our lighting might look more or less precise



- Per-face
- Per-vertex lighting
- Per-fragment lighting

Lighting Precision

- ***Per-face*** lighting: normals uniform across face



Lighting Precision

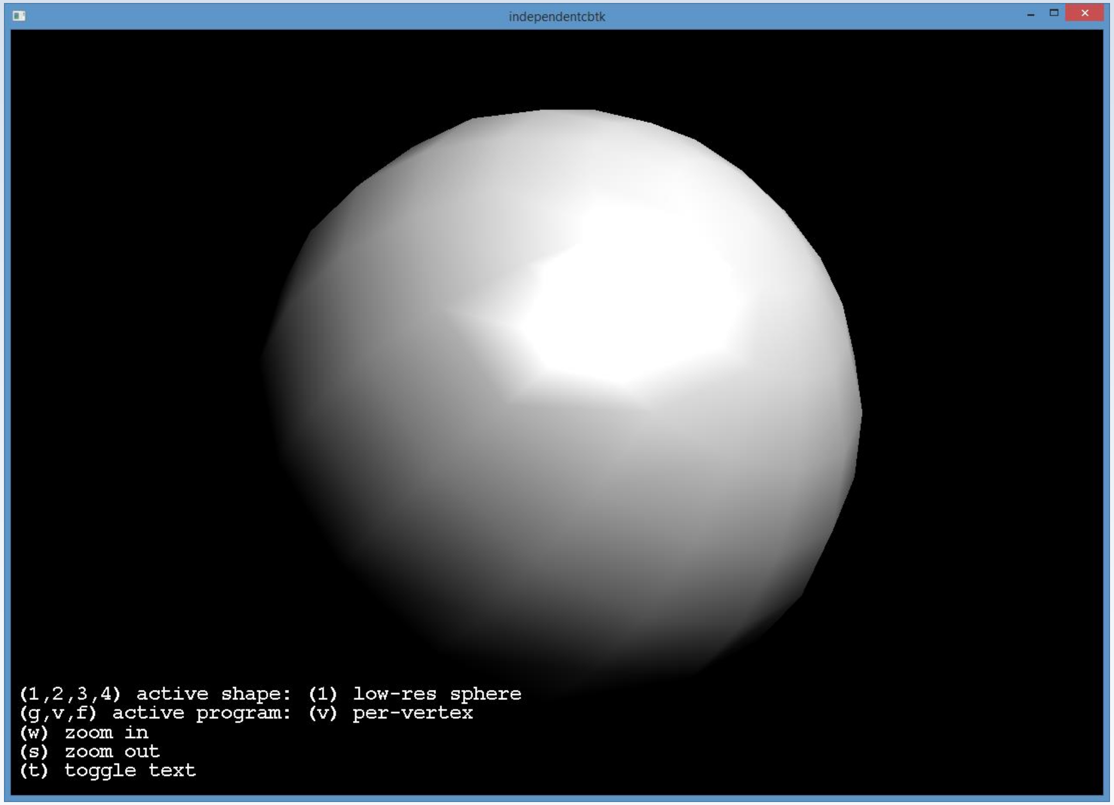
- ***Per-vertex*** and ***per-fragment*** are more relevant

- Per-vertex: calculate lighting in ***vertex shader***
- Per-fragment: calculate in ***fragment shader***

- Both work because of ***Gouraud shading***

Lighting Precision

- ***The old way: per-vertex lighting (low-precision)***

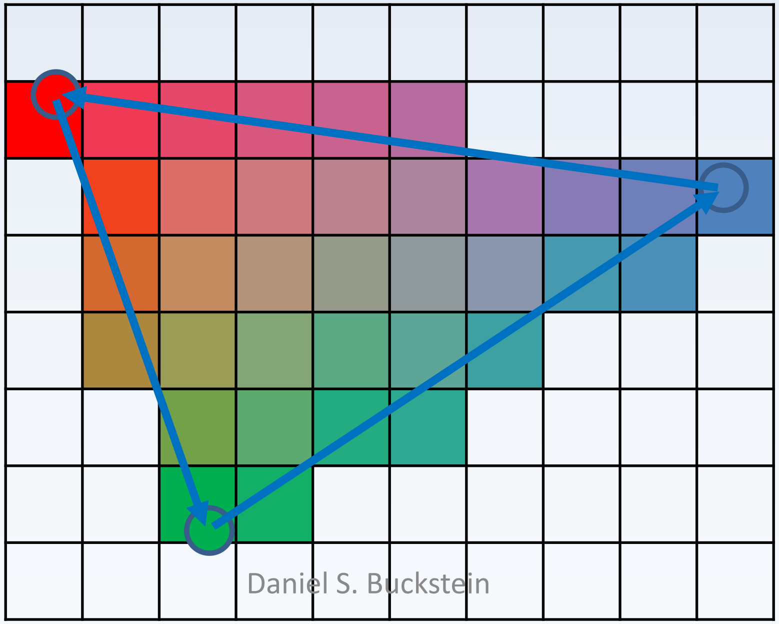


This image shows the effect of *per-vertex* lighting

(for future reference)

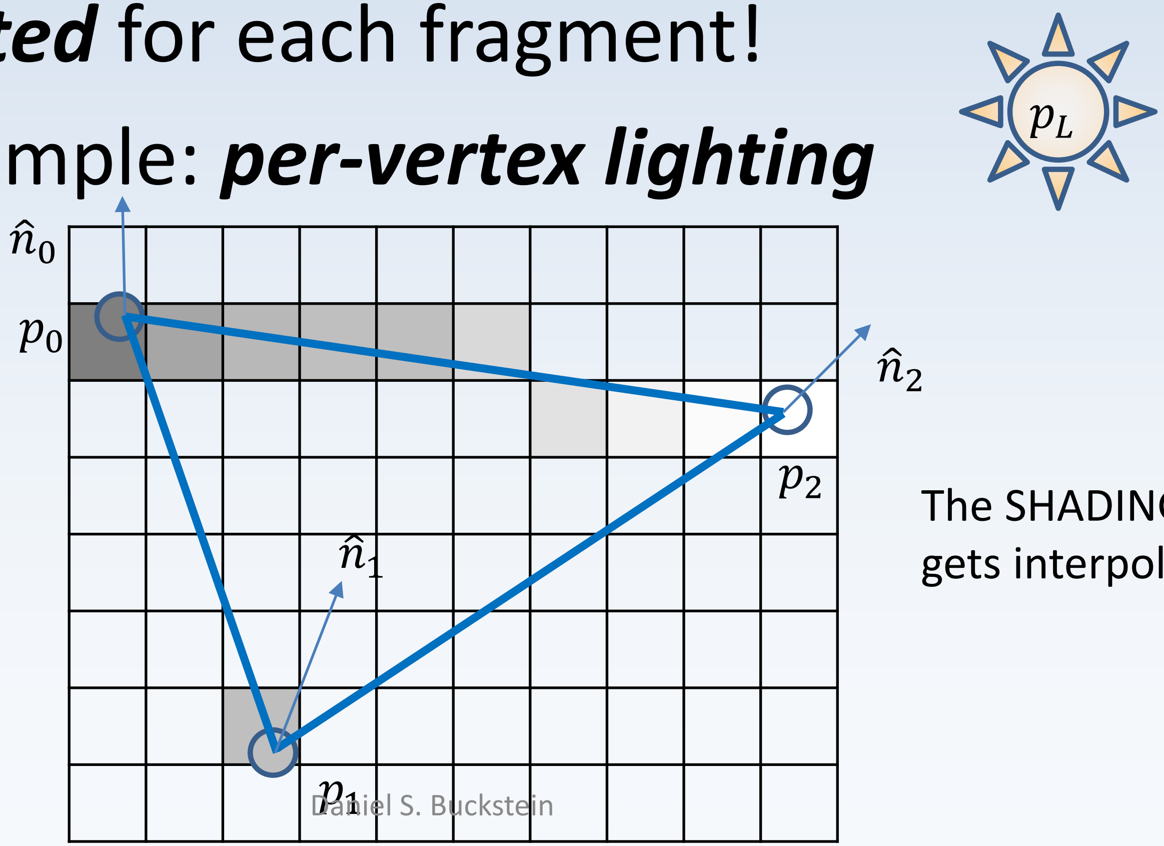
Lighting Precision

- ***Gouraud shading***: whatever values are computed in the vertex shader are ***linearly interpolated*** for each fragment!
- Visual example: ***vertex colour***



Lighting Precision

- ***Gouraud shading***: whatever values are computed in the vertex shader are ***linearly interpolated*** for each fragment!
- Visual example: ***per-vertex lighting***



The SHADING RESULT gets interpolated!

Lighting Precision

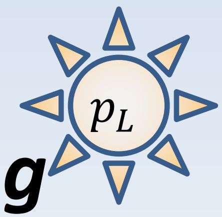
- ***Gouraud shading*** applies to ***all values*** passed from the vertex shader to the fragment shader
- Therefore if we compute a colour from lighting in the *vertex shader*…
- … the *fragment shader* only receives an *interpolated SHADING VALUE*!

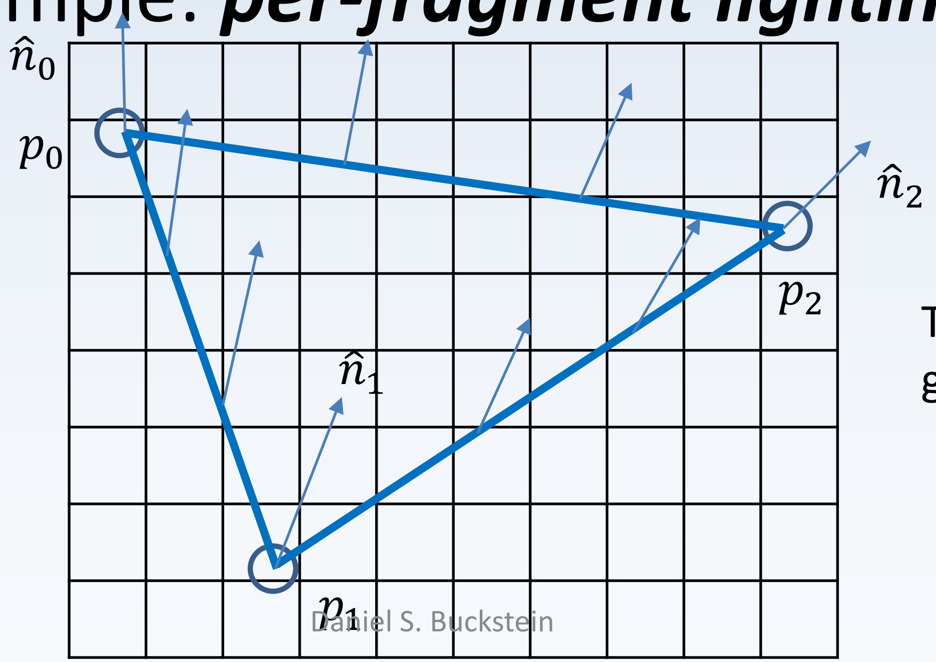
Lighting Precision

- ***Gouraud shading*** applies to ***all values*** passed from the vertex shader to the fragment shader
- Furthermore, if we simply *pass attributes*…
- …then the fragment shader computes lighting on a *per-fragment basis* (maximum precision!)

Lighting Precision

- ***Gouraud shading***: what happens if we *don't* compute lighting in the vertex shader…
- Just pass the normals through?
- Visual example: ***per-fragment lighting***





The ATTRIBUTES get interpolated!

Lighting Precision

- GLSL 1.2 example: per-vertex lighting (vert. s.)

```
attribute vec4 position;   // input vertex position
attribute vec3 normal;     // input vertex normal
uniform vec3 lightPos;     // input light position
varying float passShading; // pass-thru result, will be lerpd

void main() {
    gl_Position = gl_ModelViewProjectionMatrix * position;
    vec3 N = normalize(normal);
    vec3 L = normalize(lightPos – position.xyz);
    passShading = MY_DIFFUSE_FUNC (N, L);
}
```

Lighting Precision

- GLSL 1.2 example: per-vertex lighting (frag. s.)

```
varying float passShading; // input shading result, pre-lerpd

void main() {
    gl_FragColor.rgb = vec3 (passShading);
}
```

Lighting Precision

- GLSL 1.2 example: per-fragment lighting (vert)

```
attribute vec4 position;    // input vertex position
attribute vec3 normal;      // input vertex normal
varying vec3 passPosition; // pass-thru position, will be lerpd
varying vec3 passNormal;   // pass-thru normal, will be lerpd

void main() {
    gl_Position = gl_ModelViewProjectionMatrix * position;
    passPosition = position.xyz;
    passNormal = normal;
}
```

Lighting Precision

- GLSL 1.2 example: per-fragment lighting (frag)

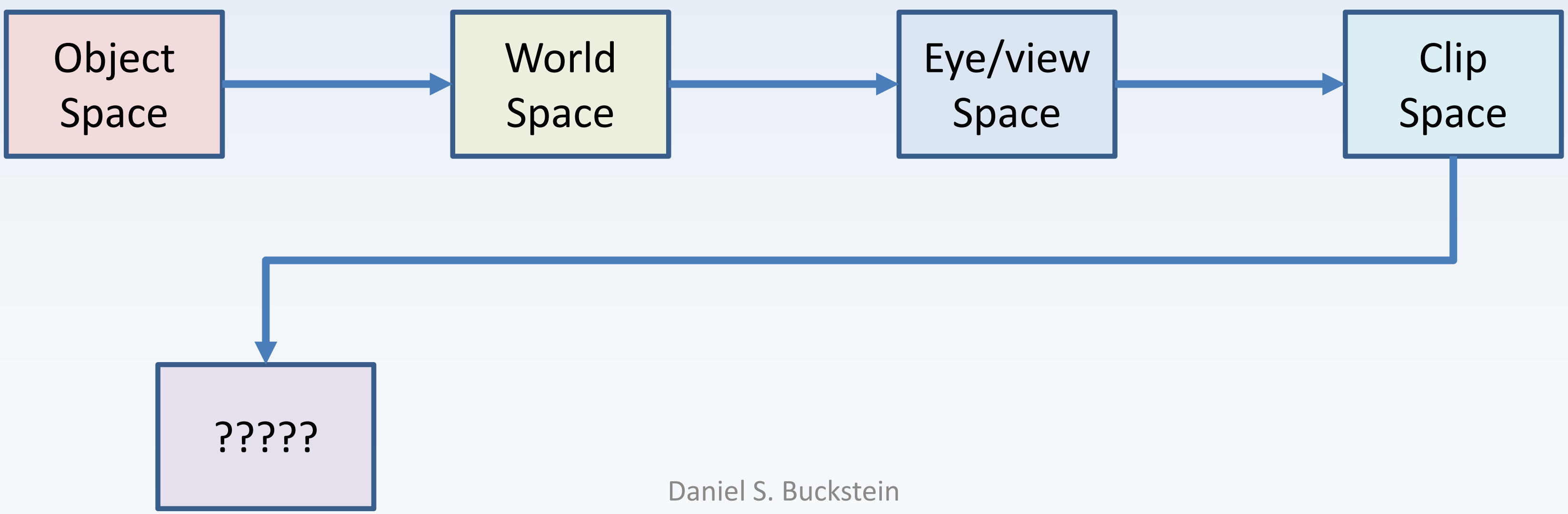
```
varying vec3 passPosition; // position of THIS FRAG, pre-lerpd
varying vec3 passNormal;   // normal at THIS FRAG, pre-lerpd
uniform vec3 lightPos;

void main() {
    vec3 N = normalize(passNormal); // WHAT IS THIS OP CALLED???
    vec3 L = normalize(lightPos – passPosition);
    float perFragShading = MY_DIFFUSE_FUNC (N, L);
    gl_FragColor.rgb = vec3 (perFragShading);
}
```

Lighting in Different Spaces

- Vertex transformations:
- Vertex shader responsible for taking a vertex from *object space* to *clip space*

$$\vec{v}_{\text{object}} \times M = \vec{v}_{\text{world}} \times V = \vec{v}_{\text{eye}} \times P = \vec{v}_{\text{clip}}$$



Lighting in Different Spaces

- Animation principles and graphic principles are BFFs <3
- ***Model matrix***: transformation from ***object's local coordinates*** to ***world coordinates***
- Ultimately computed using ***forward kinematics***

Lighting in Different Spaces

- The Model Matrix:

$$M_{\text{object}} = {}^{\text{world}}T_{\text{object}}$$

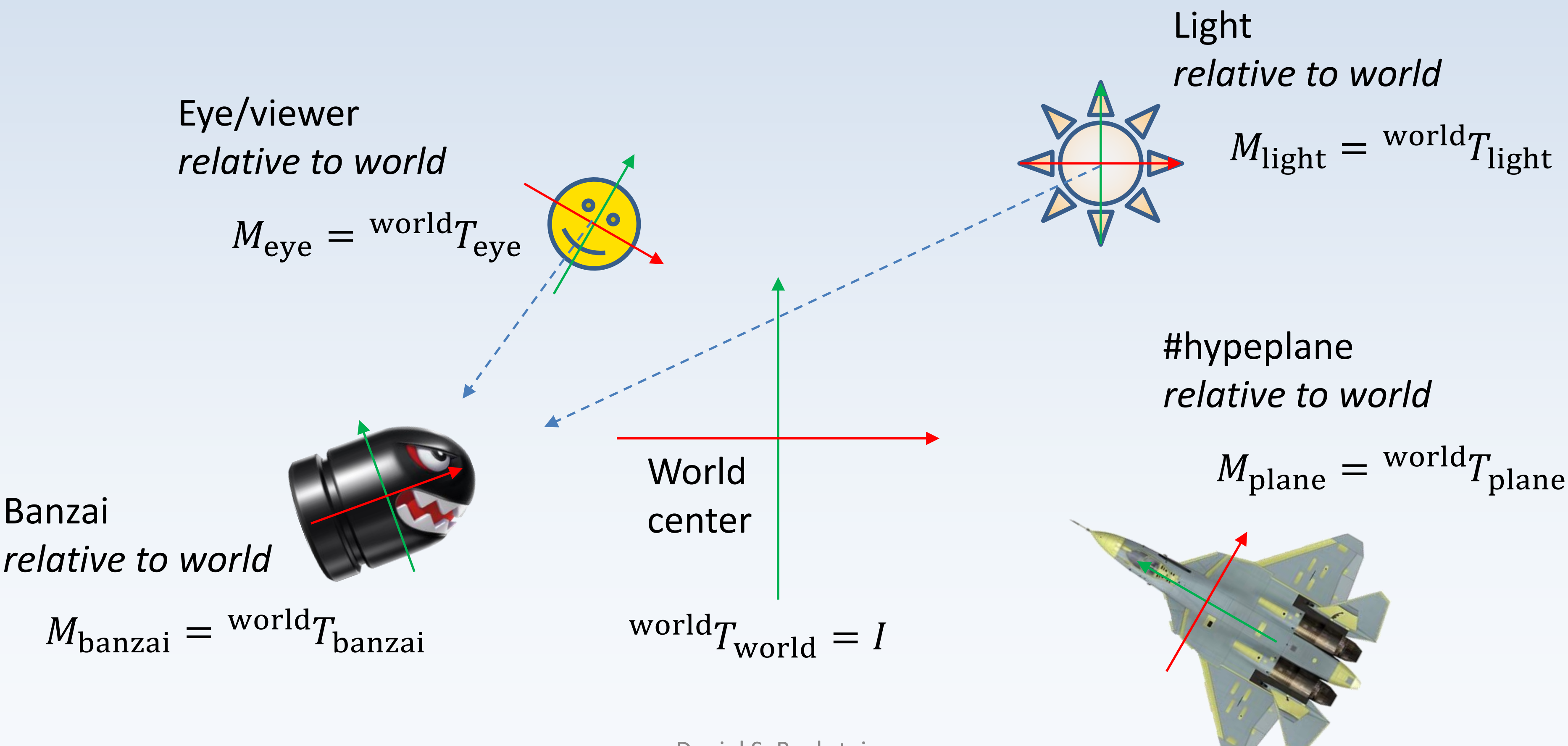
- Brings all scene objects into a *common space*

- For all objects in the scene:

$$\begin{aligned} M_{\text{object}} &= {}^{\text{world}}T_{\text{object}} \\ &= {}^{\text{world}}T_{\text{parent}}\,{}^{\text{parent}}T_{\text{object}} \end{aligned}$$

Lighting in Different Spaces

- Example scene (world-space):

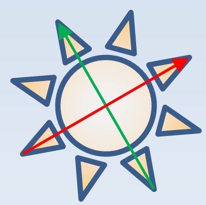


http://www.mariowiki.com/images/thumb/2/2b/BanzaibillNSMBU.png/640px-BanzaibillNSMBU.png
http://www.gamefromscratch.com/image.axd?picture=Jet.png

Lighting in Different Spaces

- Moving into someone else's space:
- First, the viewer…
- Definition of the ***view matrix*** and ***modelview matrix***

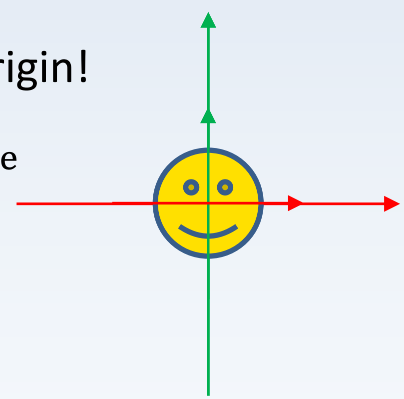
Light
relative to eye



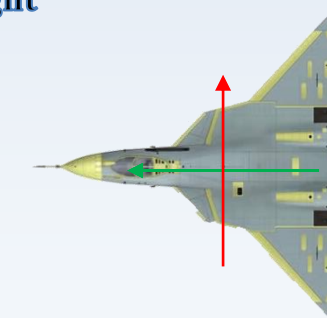
$$\boldsymbol{MV}_{\mathbf{light}} = {}^{\text{world}}T_{\text{eye}}^{-1}\,{}^{\text{world}}T_{\text{light}}$$
$$= \boldsymbol{V}\cdot\boldsymbol{M}_{\mathbf{light}}$$

View center is the new origin!

$${}^{\text{eye}}T_{\text{eye}} = {}^{\text{world}}T_{\text{eye}}^{-1}\,{}^{\text{world}}T_{\text{eye}}$$
$$= V\cdot M_{\text{eye}} = I$$



World center
relative to eye



Banzai
relative to eye

$$\mathbf{MV}_{\mathbf{banzai}} = {}^{\text{world}}T_{\text{eye}}^{-1}\,{}^{\text{world}}T_{\text{banzai}}$$
$$= \boldsymbol{V}\cdot\boldsymbol{M}_{\mathbf{banzai}}$$

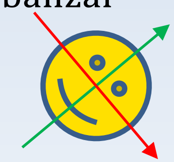
$$\boldsymbol{V} = {}^{\text{eye}}T_{\text{world}}$$
$$= {}^{\text{world}}T_{\text{eye}}^{-1} = \boldsymbol{M}_{\mathbf{eye}}^{-1}$$

Lighting in Different Spaces

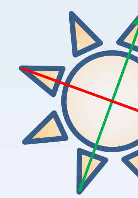
- Moving into someone else's space: (object-space)

Eye *relative to banzai*

$$ {}^{\text{banzai}}T_{\text{eye}} = {}^{\text{world}}T_{\text{banzai}}^{-1}\,{}^{\text{world}}T_{\text{eye}} $$
$$ = M_{\text{banzai}}^{-1} \cdot M_{\text{eye}} $$

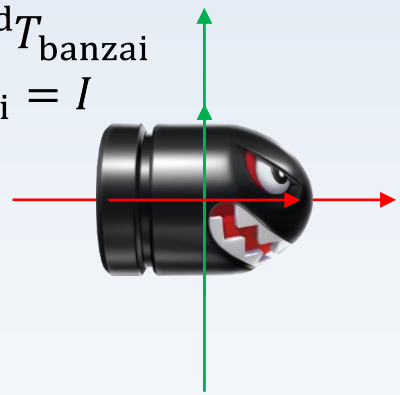


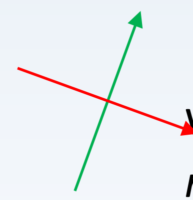
Light *relative to banzai*



Banzai is the new origin!

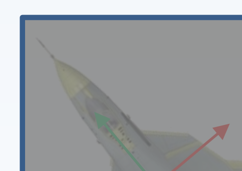
$$ {}^{\text{banzai}}T_{\text{banzai}} = {}^{\text{world}}T_{\text{banzai}}^{-1}\,{}^{\text{world}}T_{\text{banzai}} $$
$$ = M_{\text{banzai}}^{-1} \cdot M_{\text{banzai}} = I $$





World center *relative to banzai*

$$ {}^{\text{banzai}}T_{\text{world}} = {}^{\text{world}}T_{\text{banzai}}^{-1} = \boldsymbol{M_{\text{banzai}}^{-1}} $$

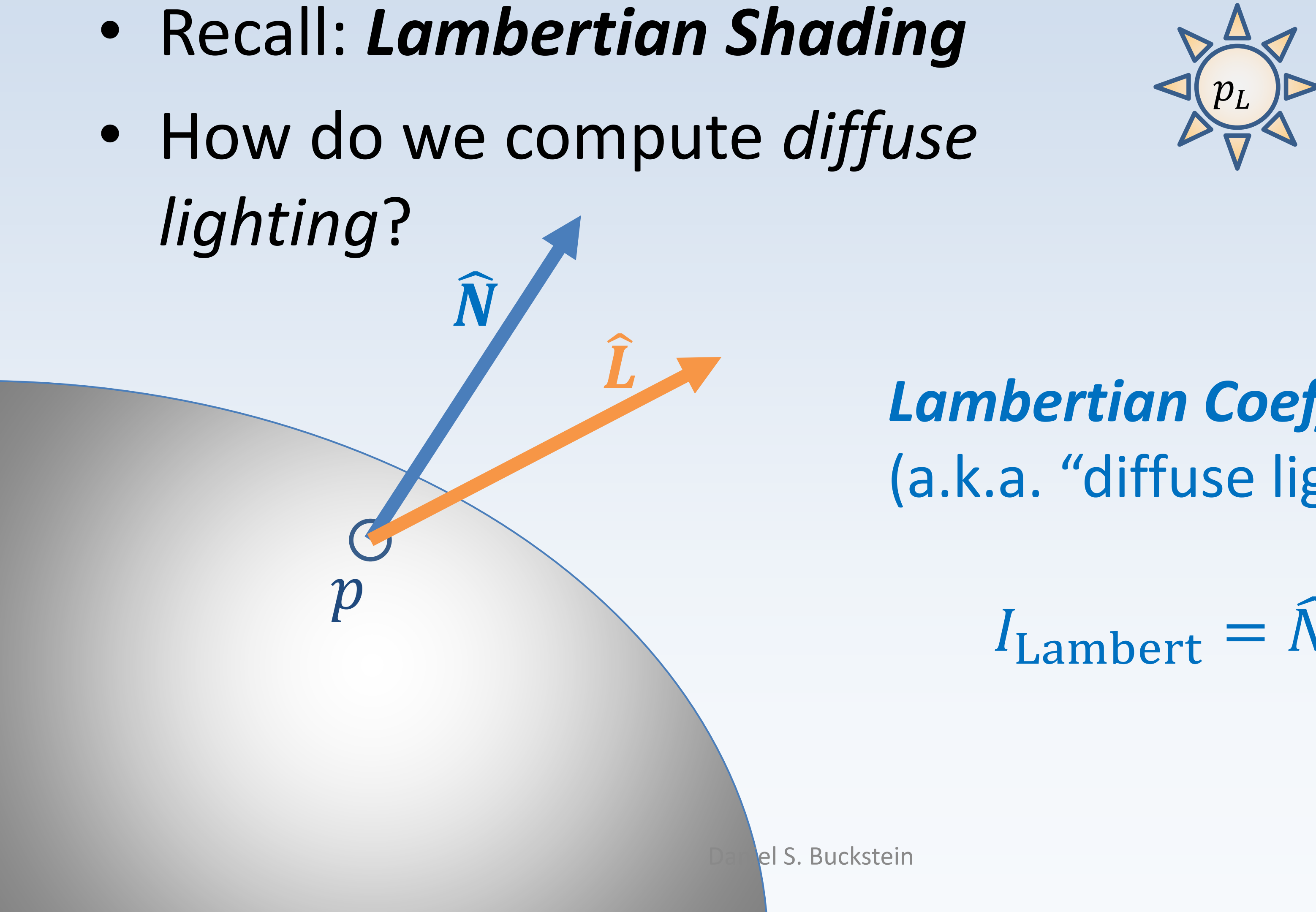


Non-Photorealistic Rendering

- Lighting and shading algorithms
  - Gooch shading, cel shading
- Outline algorithms
  - Some fast and terrible methods
  - Some precise methods, edge detection
- Tips and tricks
- Advanced algorithms

Non-Photorealistic Shading

- Recall: ***Lambertian Shading***
- How do we compute *diffuse lighting*?



Lambertian Coefficient:
(a.k.a. "diffuse lighting")

$$I_{\text{Lambert}} = \hat{N} \cdot \hat{L}$$

Non-Photorealistic Shading

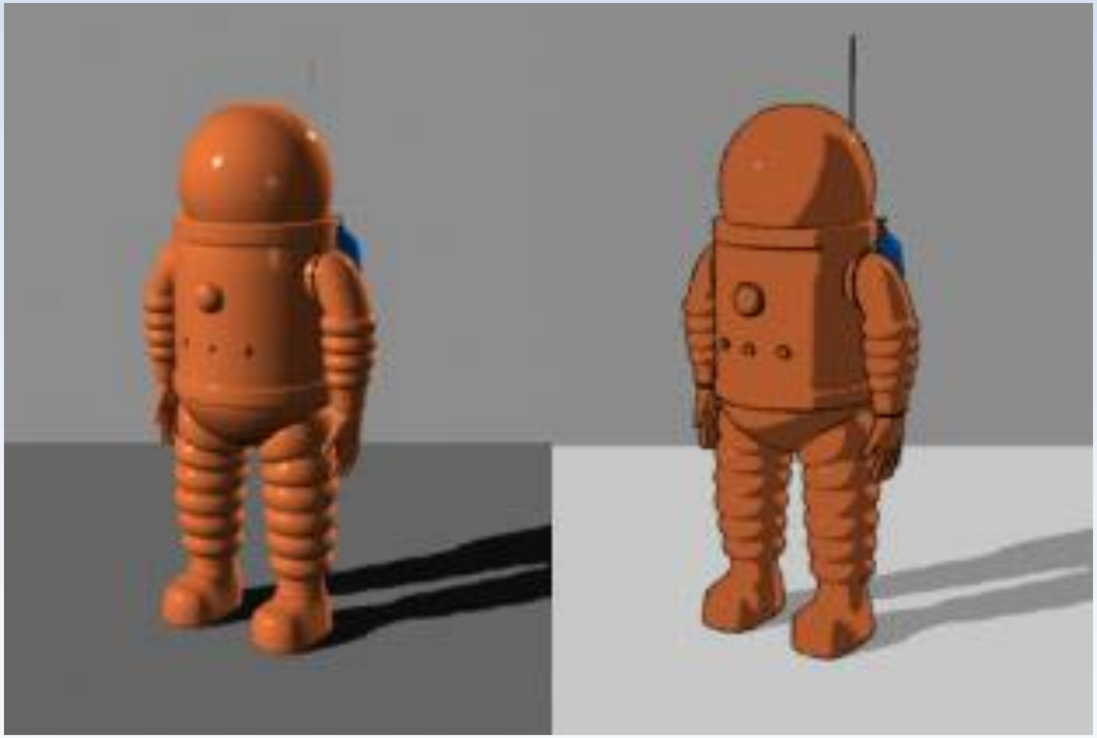
- Dot product has a range of $[-1, 1]$
- We can clamp to get a range of $[0, 1]$ using

$$I'_{\text{diffuse}} = \max(0.0, I_{\text{diffuse}}) \\ = \max\left(0.0, \hat{N} \cdot \hat{L}\right)$$

- Half of the range is just thrown out ☹
- Solution:
- $I'_{\text{diffuse}} = \text{serialize}(I_{\text{diffuse}})$

Non-Photorealistic Shading

- ***Cel Shading***:



https://upload.wikimedia.org/wikipedia/commons/b/b7/Toon-shader.jpg

Non-Photorealistic Shading

- ***Cel Shading***: Similar technique, but we want blocky shading instead of smooth
- *Method 1*: use if/else statements (SLOWWW)

```
float kdiff = dot(N, L);
     if (kdiff <= 0.00) kdiff = 0.00;
else if (kdiff <= 0.25) kdiff = 0.25;
else if (kdiff <= 0.50) kdiff = 0.50;
else if (kdiff <= 0.75) kdiff = 0.75;
else                    kdiff = 1.00;
// proceed with lighting using new kdiff
```

Non-Photorealistic Shading

- ***Cel Shading***:
- *Method 2*: Instead, let's make use of the serialized Lambertian coefficient: [0, 1]
- The serialized Lambertian coefficient gradient:

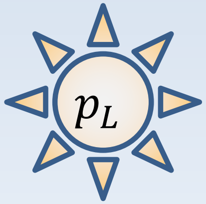
$I' = [0, 1]$

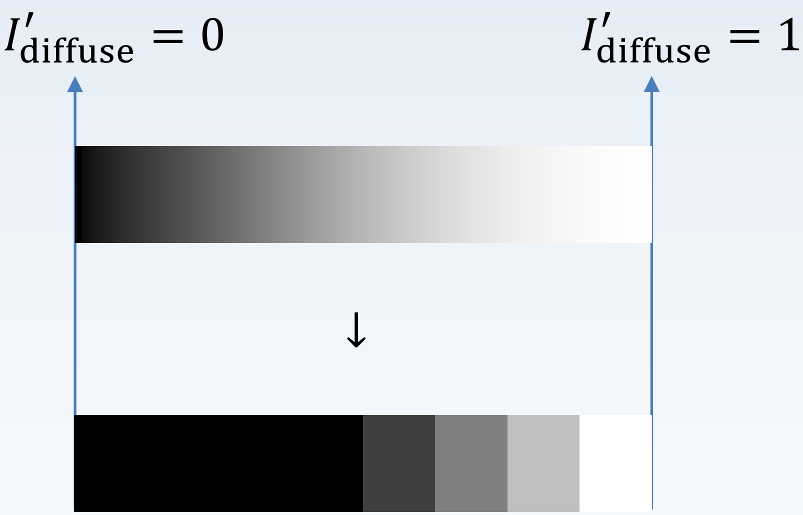
- What can we do with this???
- "Ramp": colour re-mapping technique

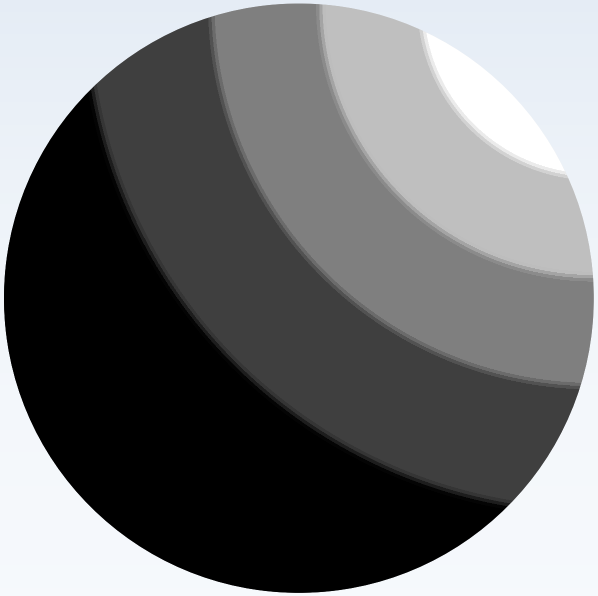
$I' = [0, 1] \rightarrow$  $\rightarrow I'' = [0, 1]$

Non-Photorealistic Shading

- Final result:
- Serialized lighting coefficient used as input to ramp

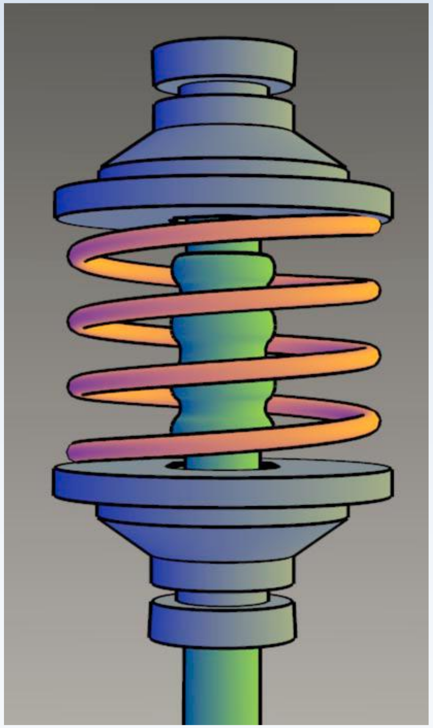


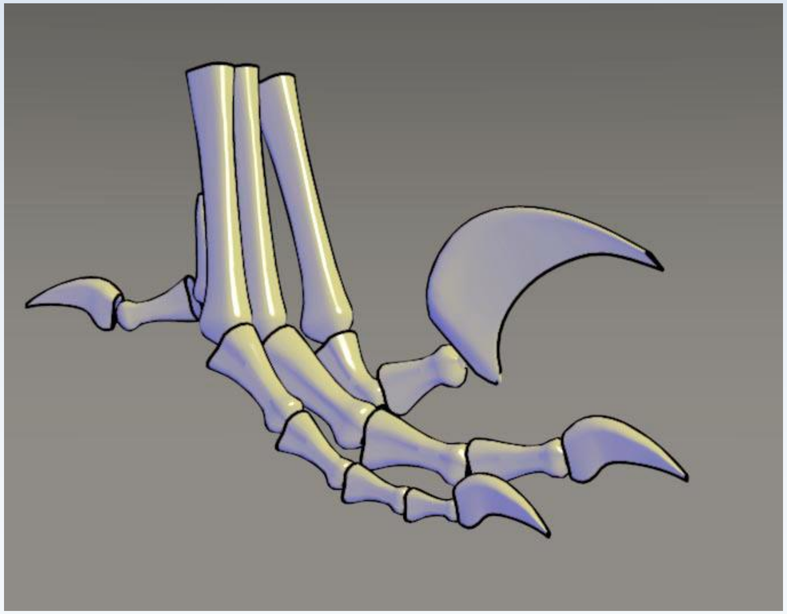




Non-Photorealistic Shading

- ***Gooch shading***:
- Created by Amy & Bruce Gooch et al (1998)





http://www.cs.northwestern.edu/~bgooch/PDFs/gooch98.pdf

Non-Photorealistic Shading

- ***Gooch Shading***: the core concept:
- Use *colour* to simulate *light*
- Cool colour = dark, warm colour = light:

$\hat{N} \cdot \hat{L} = -1$

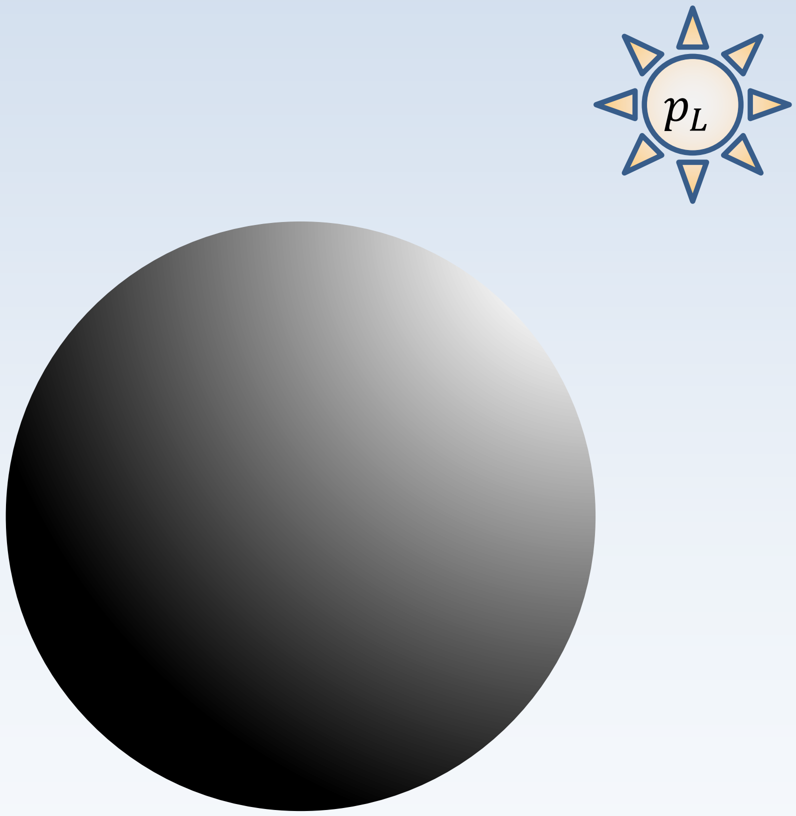
$\rightarrow I'_{\text{diffuse}} = 0$

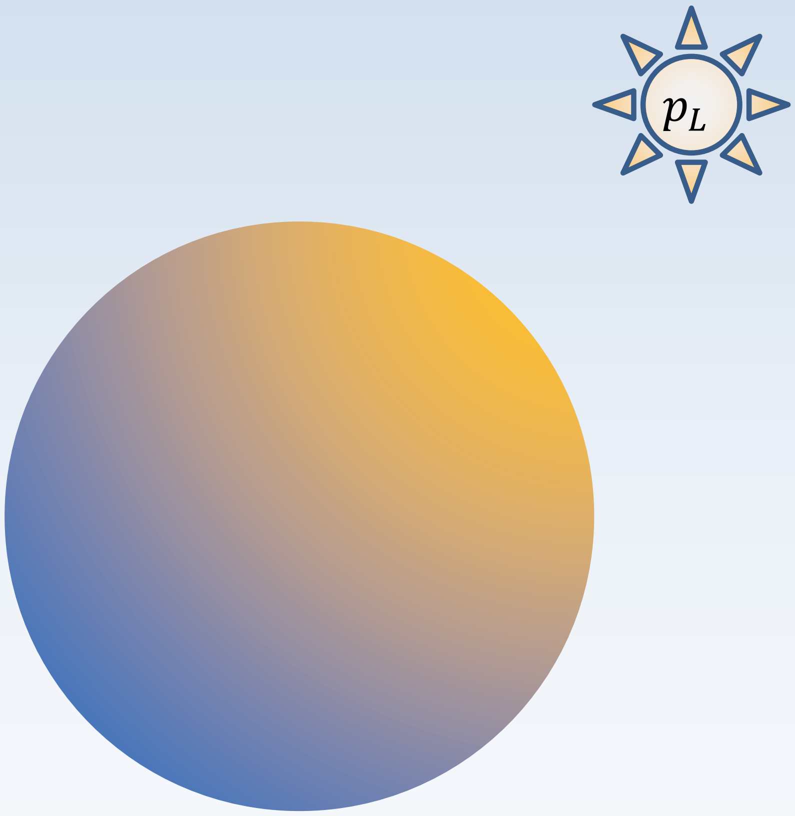
$\hat{N} \cdot \hat{L} = +1$

$\rightarrow I'_{\text{diffuse}} = 1$

Non-Photorealistic Shading

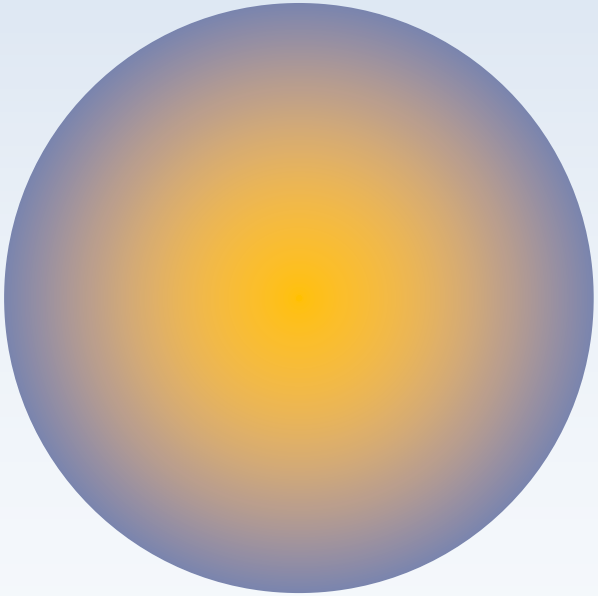
- Normal shading vs. Gooch shading:





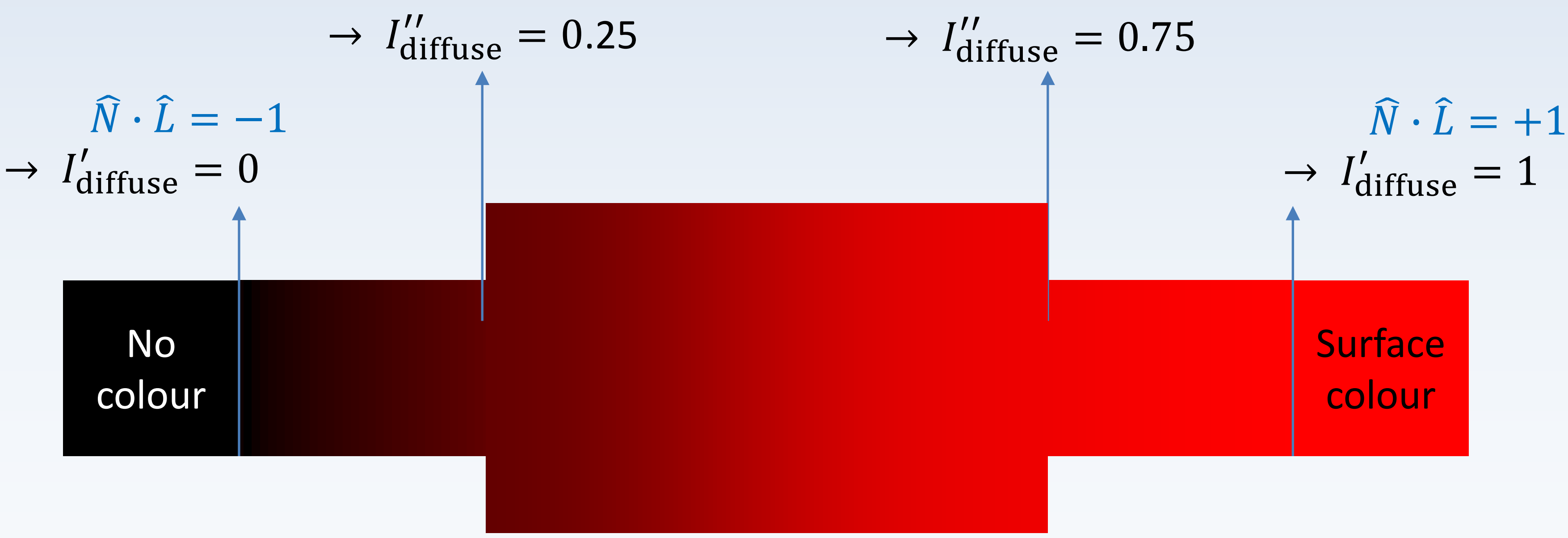
Non-Photorealistic Shading

- Normal shading vs. Gooch shading:



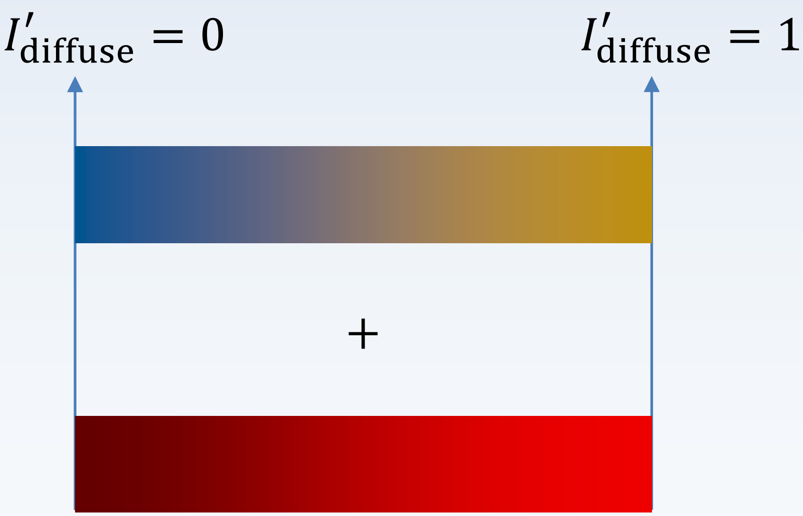
Non-Photorealistic Shading

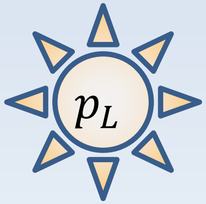
- ***Gooch Shading***: that’s not all… also takes the surface colour into account
  - Select a portion of the gradient → $$I''_{\text{diffuse}} = \text{reserialize}(I'_{\text{diffuse}}) = 0.5(I'_{\text{diffuse}}) + 0.25$$



Non-Photorealistic Shading

- Final result:
- Darken (scale) the lighting sample
  - (blue to yellow)
- Add surface colour sample!

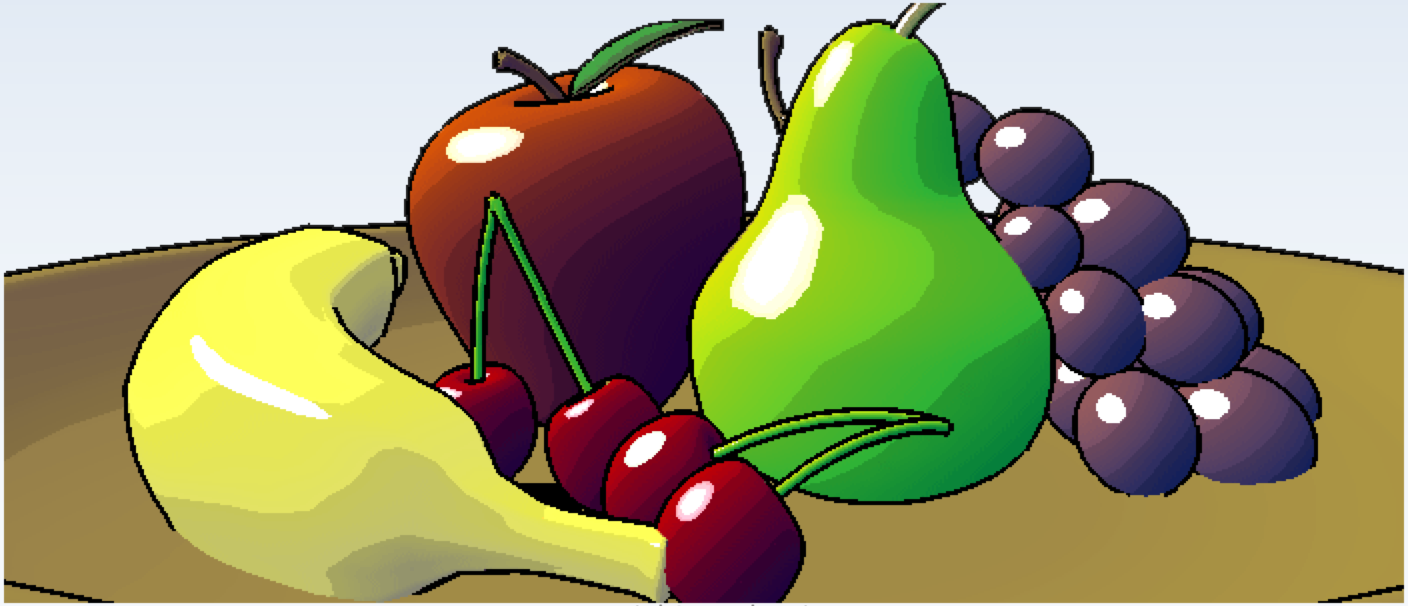




Non-Photorealistic Shading

- *Non-photorealistic shading pro tips*!!!

1) Cel+Gooch combo
   - Use these two shading techniques in tandem for maximum non-photo action*



*until we add outlines

http://marctenbosch.com/npr_shading/

Non-Photorealistic Shading

- *Non-photorealistic shading pro tips*!!!

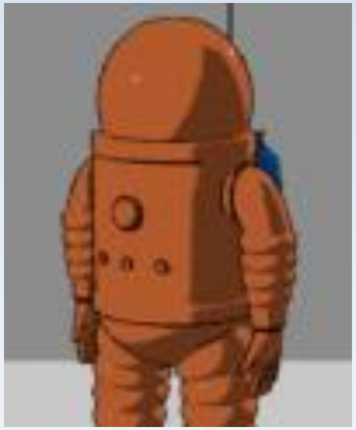
2) The albedo map is the key ;)

  - Flat colours are winner/gagnant

http://www.dailyemerald.com/wp-content/uploads/2013/09/Wind-Waker-Windfall.jpg

Non-Photorealistic Shading

- Gooch & cel shading are nice…
- …what if we want a more cartoony feel?

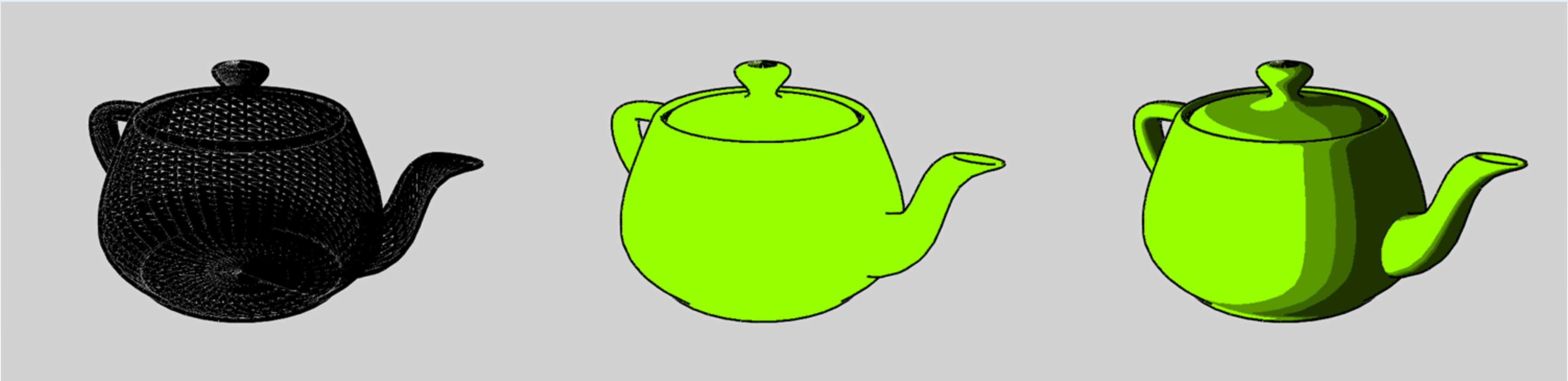


- "***Toon shading***"
- Add lines to make scene look hand-drawn

http://s1211.photobucket.com/user/ponrhartman/media/WIP/goku-0015.jpg.html

Outline Algorithms

- *Method 1*: Second geometry pass
- Front-face culling, re-draw object in wireframe
- Shader program:
- Fragment shader: ***black*** (or any solid colour)

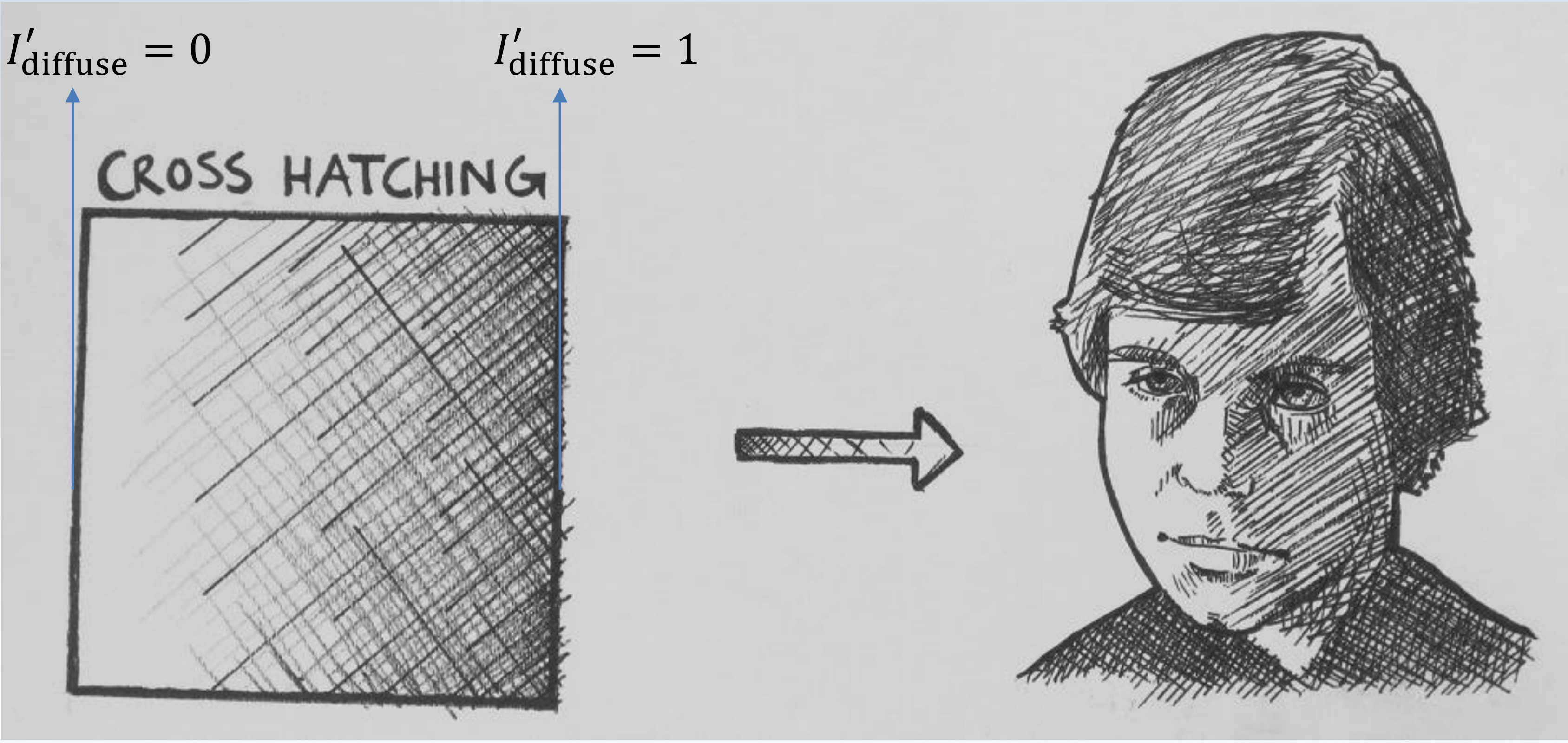


https://en.wikipedia.org/wiki/Cel_shading#/media/File:Celshading_teapot_large.png

Outline Algorithms

- *Method 2*: ***Post-processing***
- Image processing techniques
- “Convolution” algorithms
- Sobel filter
- Canny edge detection (the best)
- …
- We’ll get to this in a few weeks!

Intermediate NPR Algorithms



Advanced NPR Algorithms

- Ryan Woodward's "Thought of You"

- 'Nuff said.

https://www.youtube.com/watch?v=OBk3ynRbtsw

More Lighting!!!

- An *intermediate* technique:

http://www.valvesoftware.com/publications/2007/NPAR07_IllustrativeRenderingInTeamFortress2.pdf

The end.

- Questions? Comments? Concerns?