Lighting & Shading

What do we mean by “lighting” and “shading”?

Lighting vs. Shading

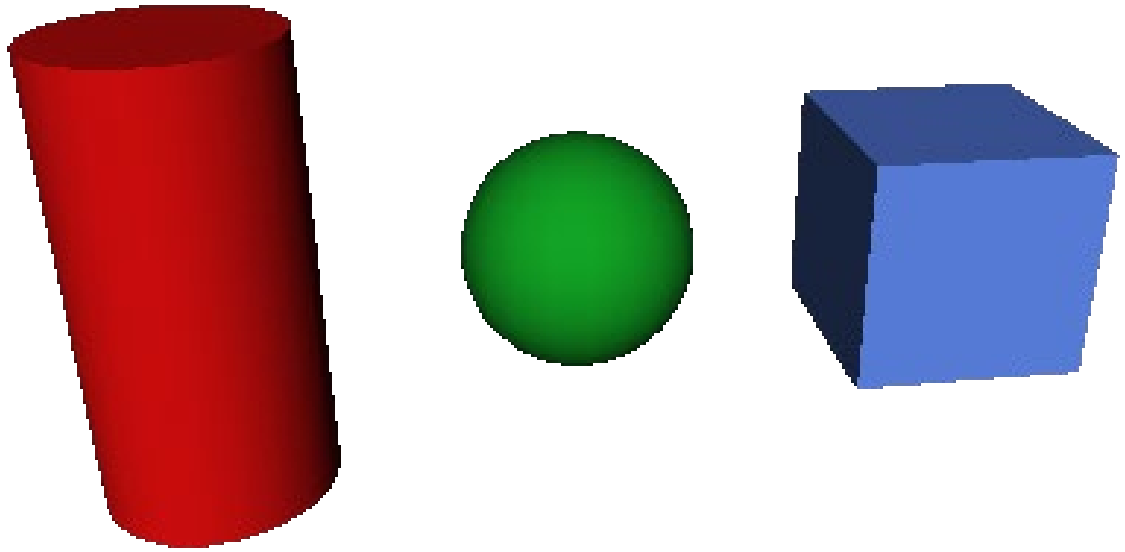
- **Lighting**: Approximating how light interacts with a surface
  - Mathematical equations
  - Based on physics

- **Shading**: Applying lighting results to the pixels of a mesh
  - Running the equations
  - Combining with surface materials
  - Getting the results on the screen

- Terms sometimes used interchangeably

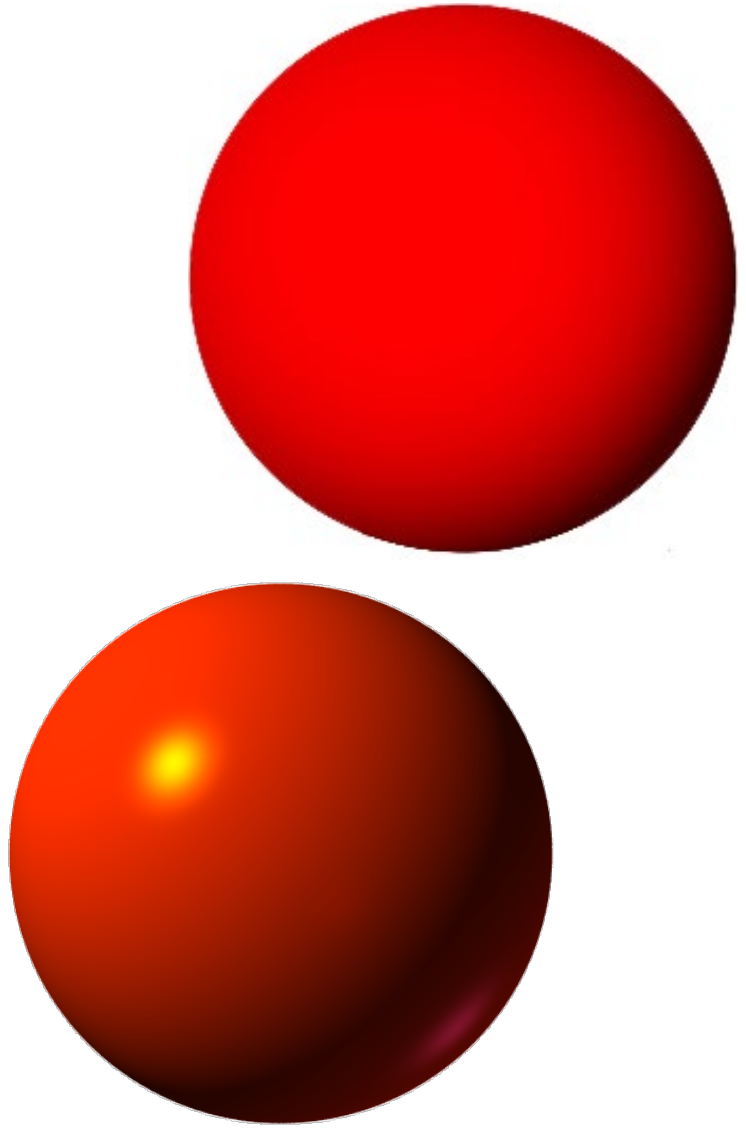
What we have so far

What we want (to start…)



Lighting

- We aren't *simulating* actual light
- We're *approximating* it
- Looking for plausible results
- Which are quick to calculate
- Need "simple" math functions to describe light/surface interactions



What is really happening with light?

- Actually lighting is very complex
- Incredible amount bouncing around
- Reflections everywhere
- Too much to calculate in real time!
- We focus on major components

Colors in a computer

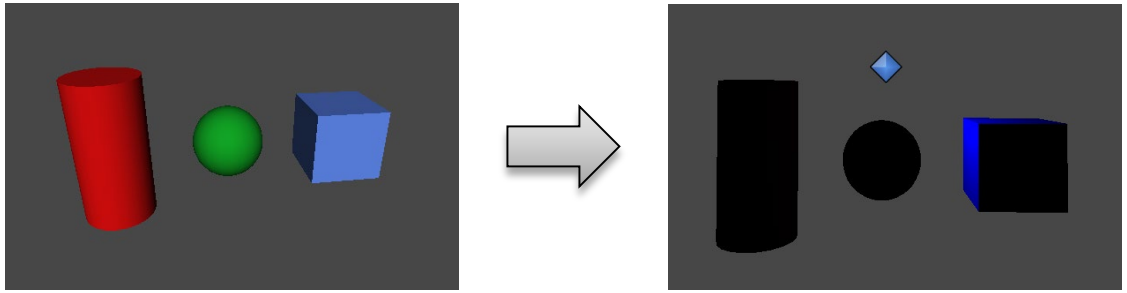
- Colors stored as three numeric components (channels)
  - Red, Green & Blue
  - Each is 0–255 (or 0.0f – 1.0f in a shader)
  - ~16.7 million combinations

Red	(1.0, 0.0, 0.0)	White	(1.0, 1.0, 1.0)
Blue	(0.0, 0.0, 1.0)	Black	(0.0, 0.0, 0.0)
Purple	(1.0, 0.0, 1.0)	Grey	(0.5, 0.5, 0.5)

Combining colors

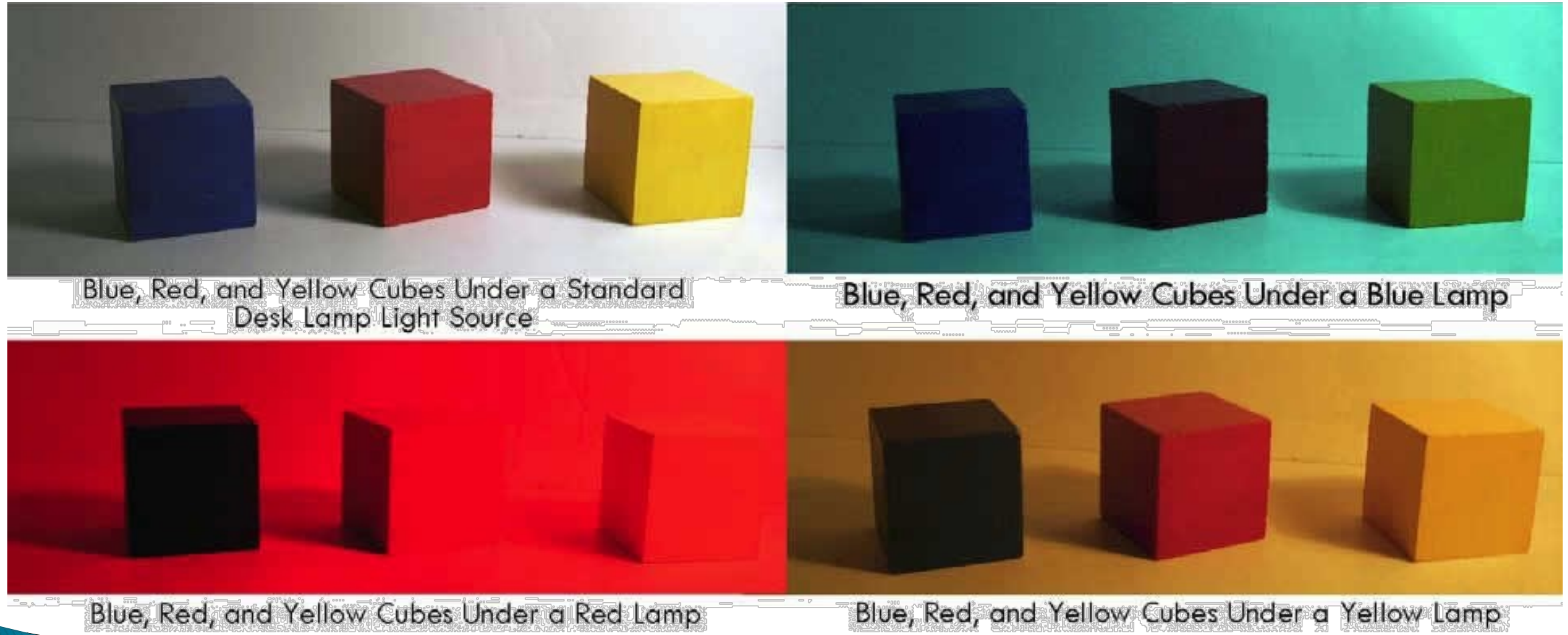
- Adding colors
  - Often used when applying multiple lights
  - More light = brighter!
  - Perceptual color may change (blue + red = purple)

- Multiplying colors
  - Often used when tinting colors
  - Light striking a surface is a tint
  - Blue * red = black



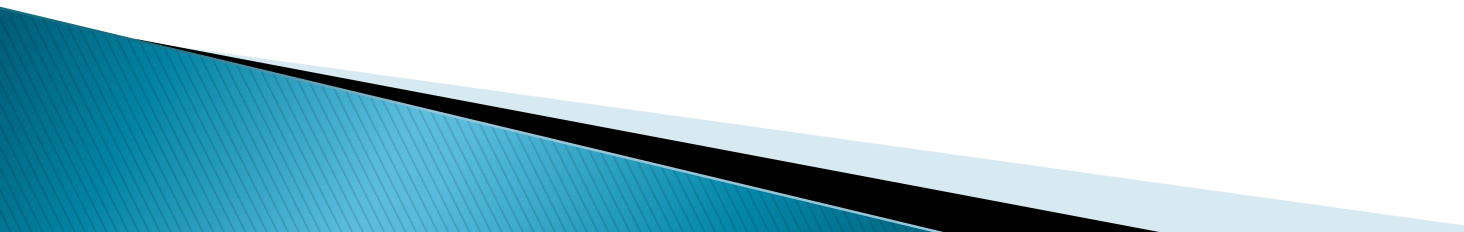
- Color * Scalar → Brighten or darken

Real example of lights & surfaces



Quick aside – Where do we put lighting code?

- It happens in a shader – but which one?
- Could technically do it in either
  - Lighting in vertex shader = “per-vertex lighting”
  - Lighting in pixel shader = “per-pixel lighting”
- Which is best?

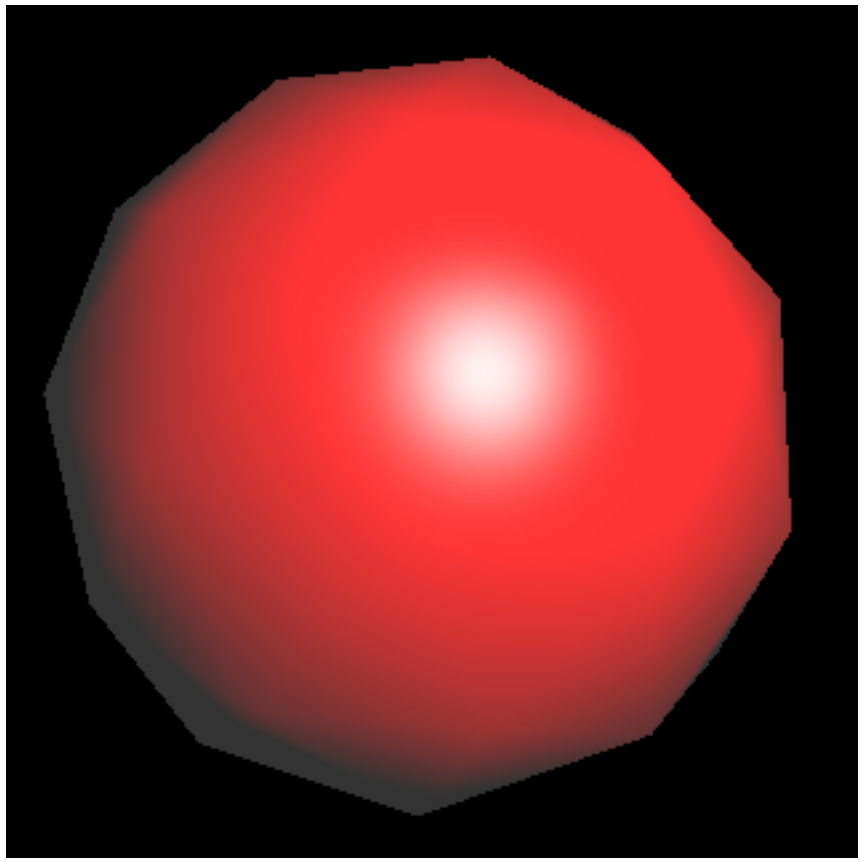


Per-vertex lighting

- All lighting done in vertex shader
  - Results interpolated by rasterizer
  - Known as Gouraud shading
- Results in very obvious geometry
- Faster, but *ugly*
  - Fewer vertices than pixels
  - Fewer overall calculations

Per-pixel lighting

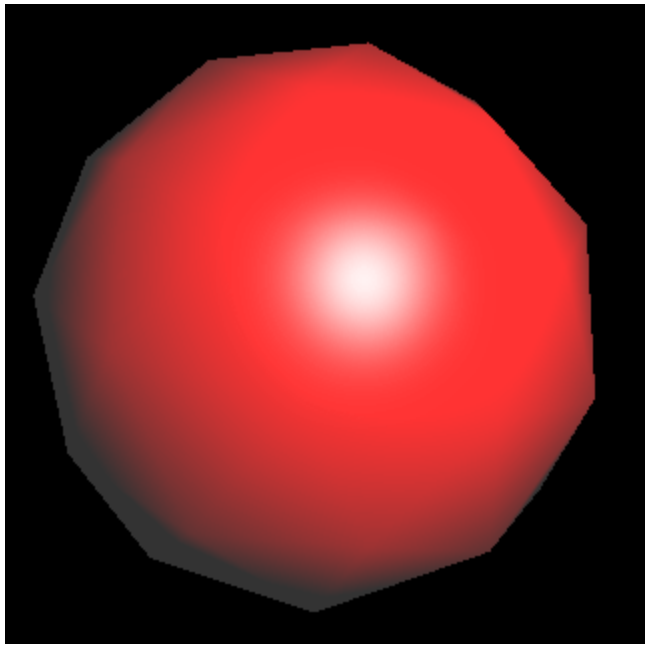
- Lighting done at the *pixel level*
  - Pass normal to pixel shader
  - Rasterizer interpolates normal
- Much more accurate!
  - Changes non-linearly across each triangle
  - Small differences noticeable
- The only way to fly!



Comparison

Per-Vertex Lighting

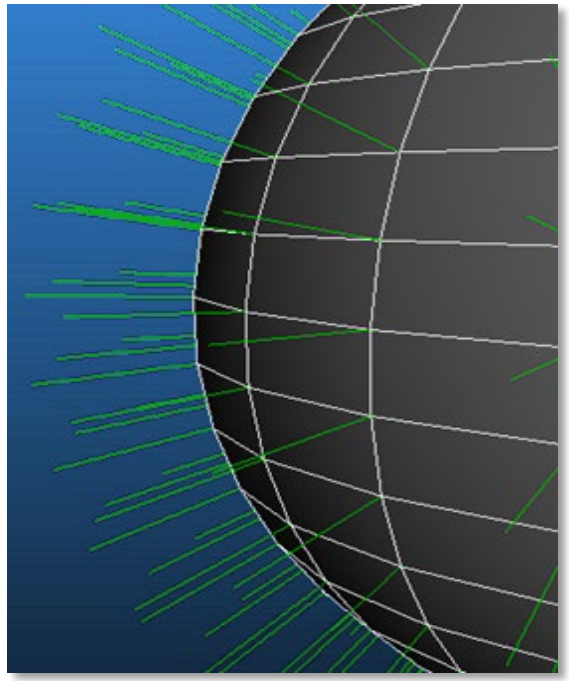
Per-Pixel Lighting



Shading Requirements

Requirement 1: *Surface information*

- **Normals**
  - Direction surface faces
  - Generally “outward”
  - Defined at each vertex
- **Shininess**
  - Material property
  - Shiny ← → Matte
- **Color**
  - Solid RGB color or texture
  - Tinted by light

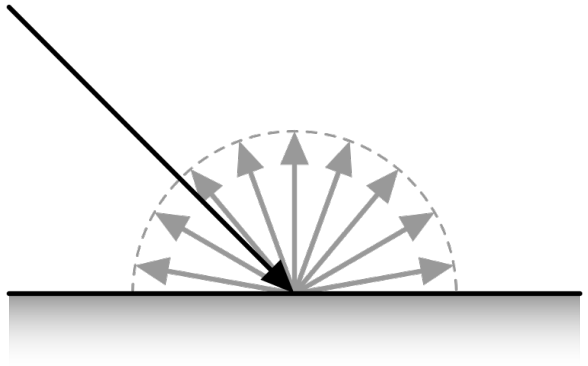


Requirement 2: *Light source*

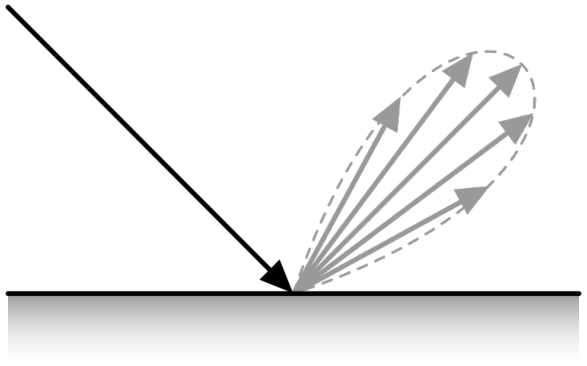
- **Position** of the light source
  - If it has one
- **Direction** light is traveling
  - From light source to surface
- **Color** – single RGB color
- **Intensity** – a simple multiplier

Requirement 3: *Lighting equations*

- Bidirectional Reflectance Distribution Functions (BRDFs)
- Functions that describe how light is reflected



Diffusion
(Lambert)



Glossy Reflection
(Specular / Phong)

1: Surface Information

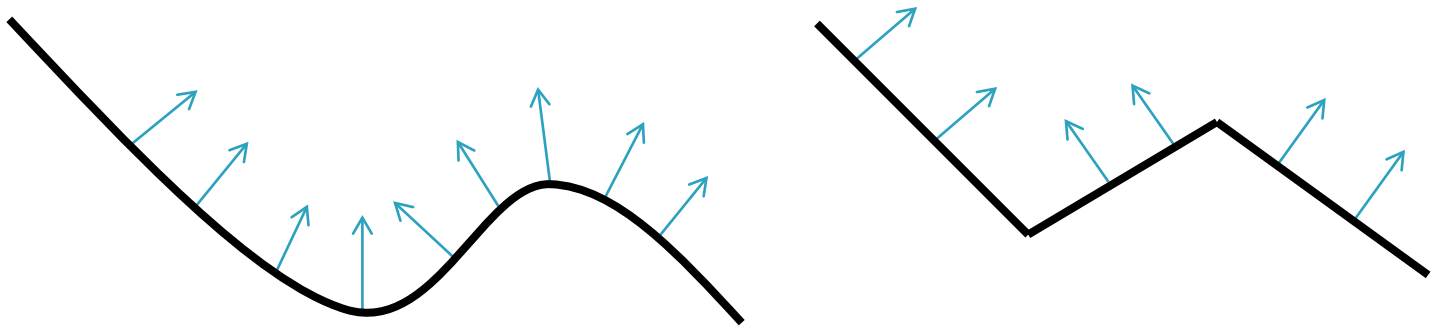
Surface information

- Already have a “surface” – a mesh
- Need the following:
  - Normals – Defined in the 3D model file
  - Color – Material property (pick a color or use a texture)
  - Shininess – Material property (pick an amount)
- Let’s take a closer look at normals

Normals

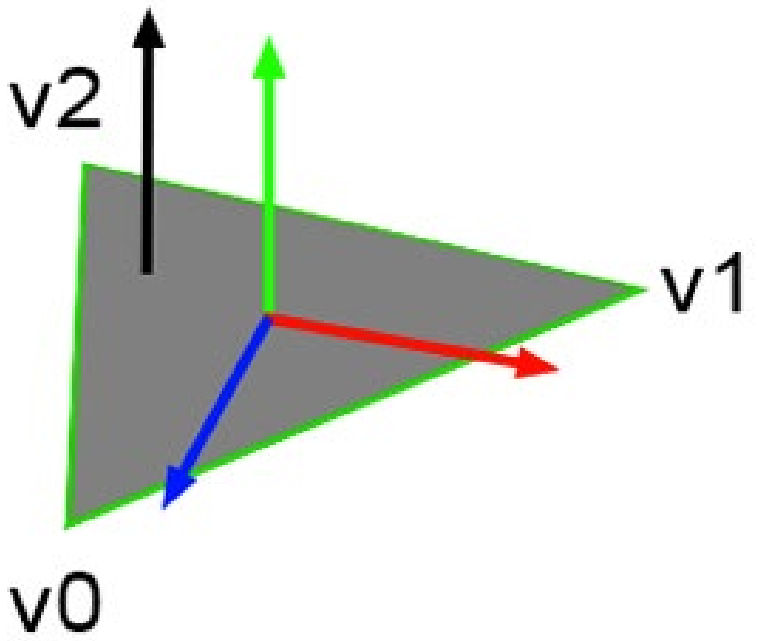
- A vector that is perpendicular to the tangent plane to a surface at a particular point

- Which way the surface “faces” at that point



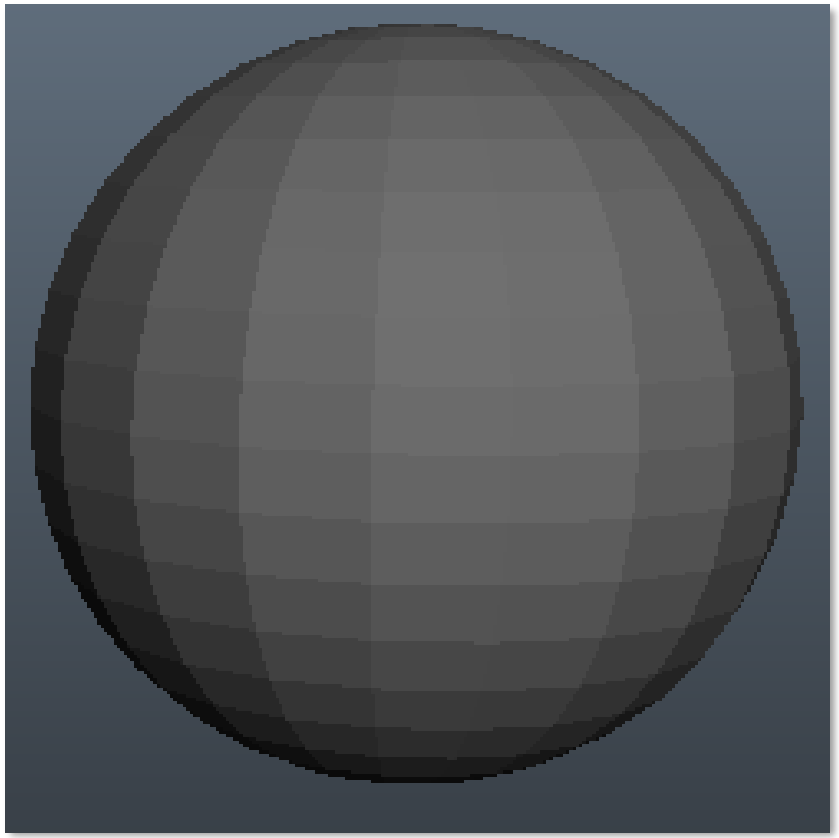
Calculating normal of a triangle

- Cross product
  - Cross two edge of the triangle
  - (v0,v1) x (v0,v2)
- This gives a uniform normal for the entire face
- Is that what we want?

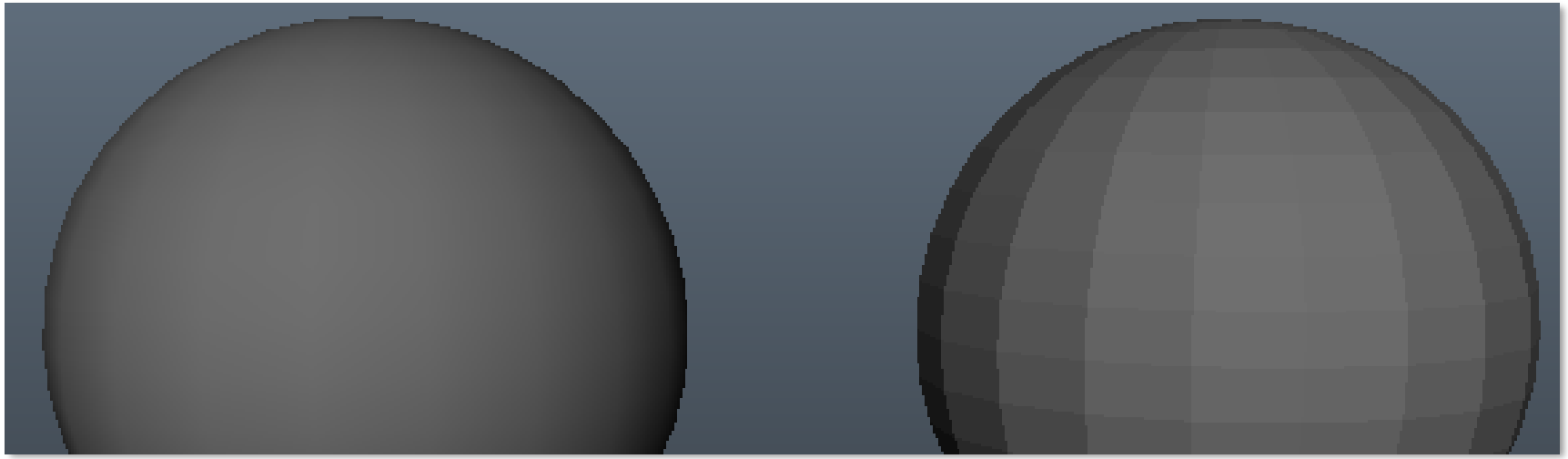


Uniform face normals

- When each pixel of a triangle has the same normal
- “Hard shading”
- Results in flat-looking triangles
- And hard edges between neighboring triangles



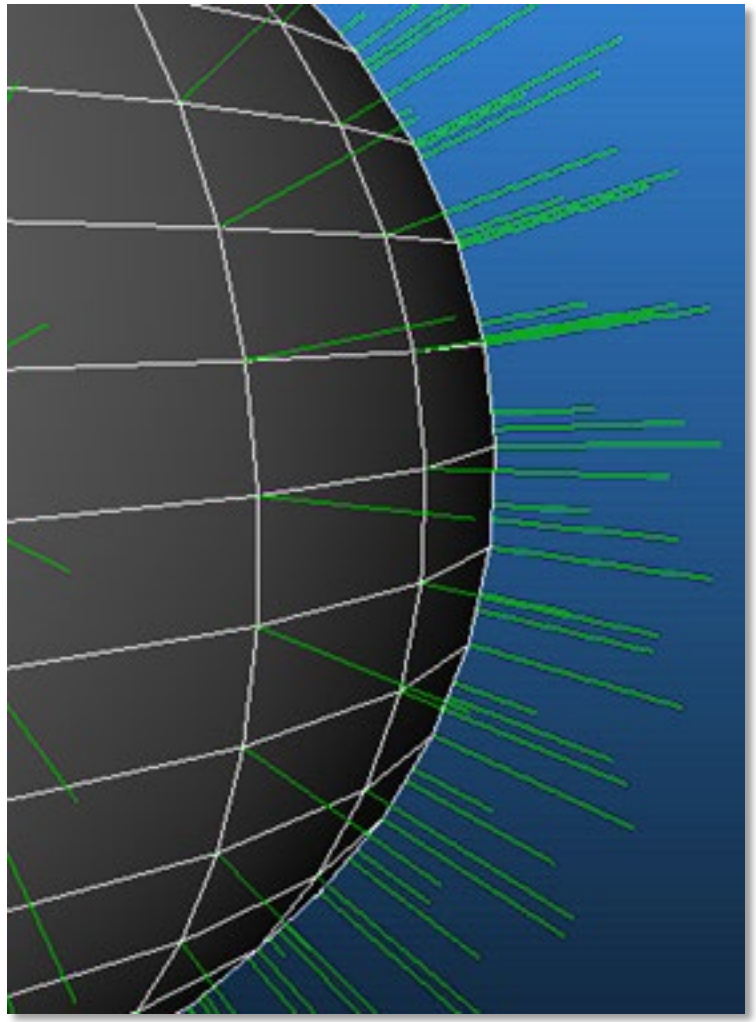
Smooth vs. Hard Shading



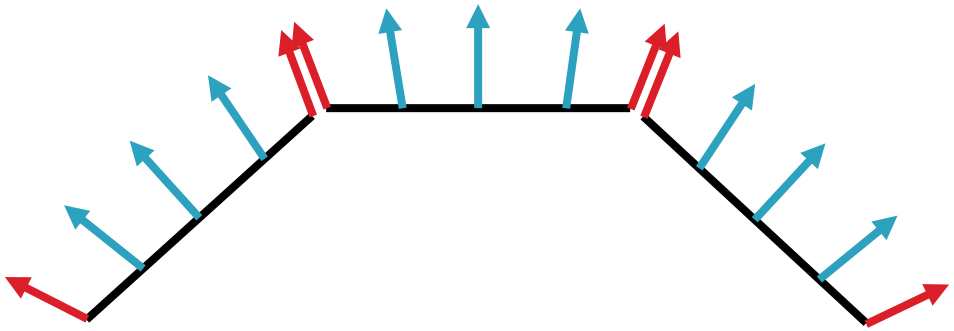
- Normal is *different* at each pixel of a face
- Each vertex of a triangle has a *different* normal
- Normal is the *same* at each pixel of a face
- Each vertex of a triangle has the *same* normal

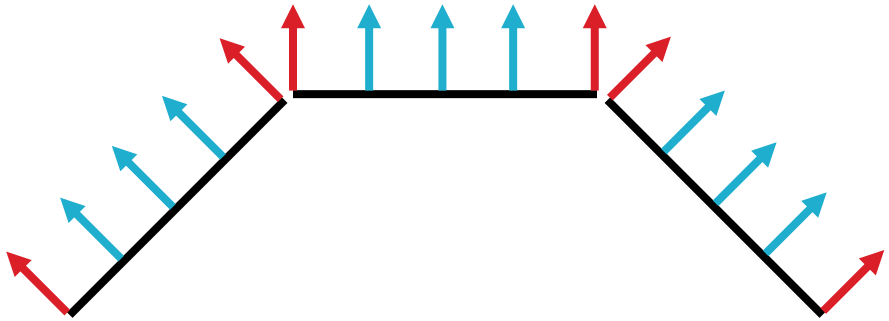
Normals

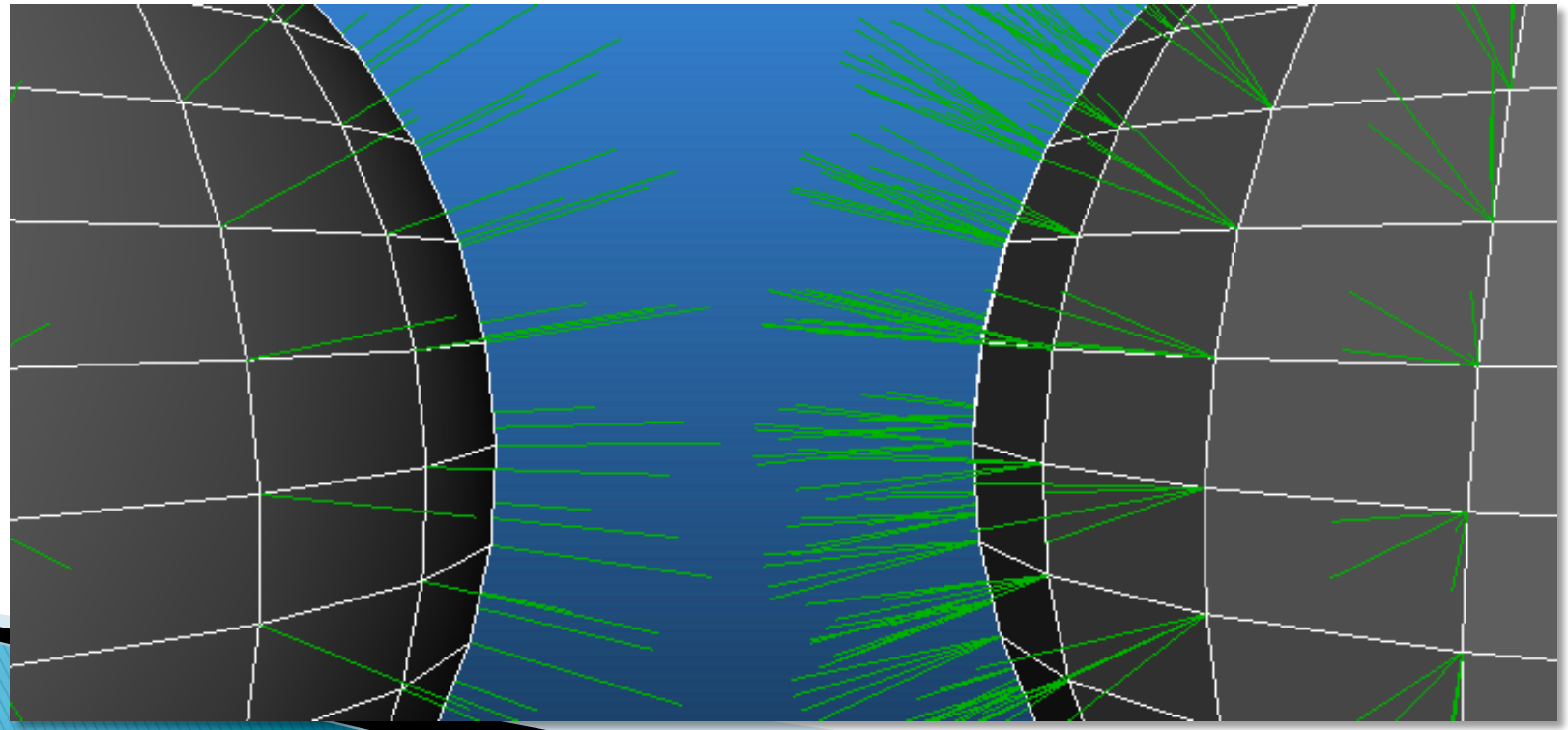
- Stored at each vertex
- Define which way the surface “faces”
- Normal of each *pixel* is interpolated from normals at *vertices*



Smooth vs. Hard Shading





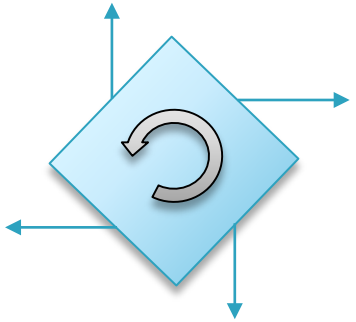


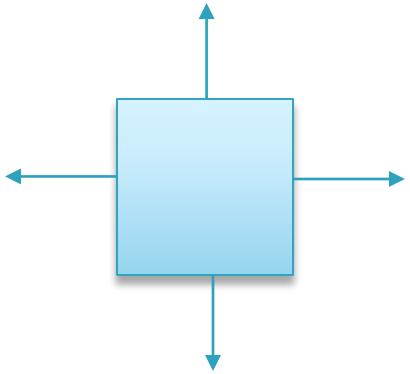
Normals in shaders

- VS must send normals to pixel shader
  - Update VertexToPixel
  - `float3 normal : NORMAL;`
- Just pass normals through? Fine if there's *no rotation*
- If an object rotates, its normals must be rotated, too!

Rotations & normals

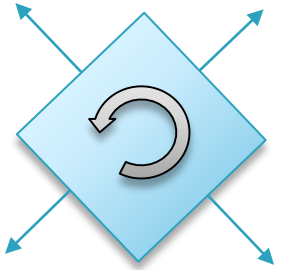
Only rotating vertices
Normals are incorrect





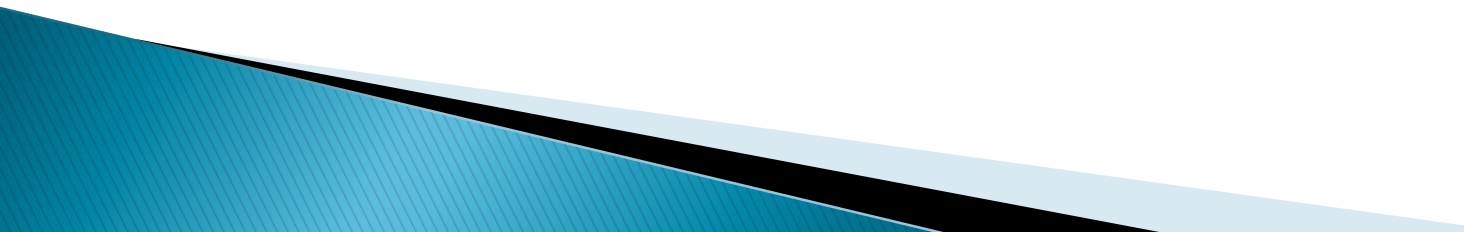
Object with normals
(No rotation)

Rotating both
Normals are correct



Normals in world space

- Multiply the input normal by world matrix
  - Works fine if there's rotation
  - Works fine if there's *uniform* scaling
- Problematic if there's translation!
- Shouldn't translate normal!



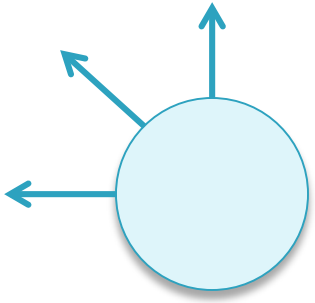
Ignoring translation

- We just want scaling and rotation
- Cast the world matrix to a float3x3
  - Keeps top-left 3x3 portion
  - That's the part with scaling and rotation
- Normal is 3-component anyway
  - So this makes the math easy

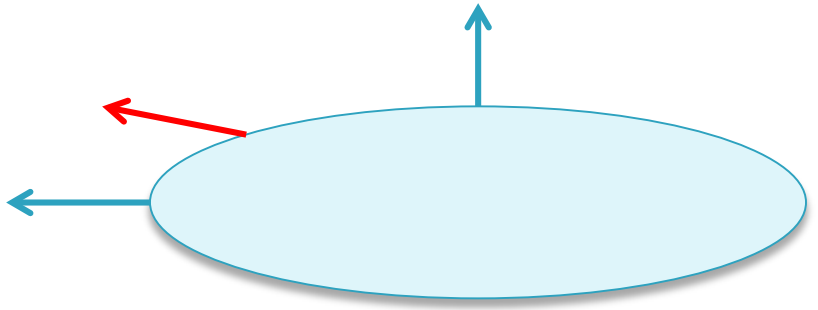
Normals & non-uniform scale

- Problem when multiplying:
  - Normal
  - World matrix with a *non-uniform scale*
- Non-Uniform Scale
  - Scale with a different value on at least one axis
- Incorrect results for world space normals

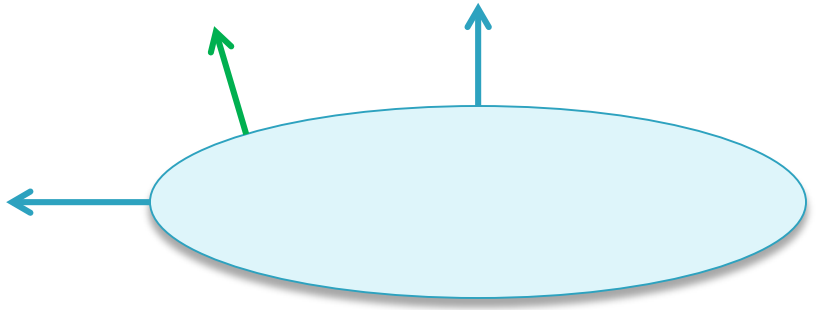
Scaling Normals



Non-scaled sphere
(with normals)



Scaled on only one axis
(Normals scaled **incorrectly**)



Normals scaled with
inverse transpose

Fixes for non-uniform scale

- **Solution 1**: Don't use non-uniform scales!
- **Solution 2**: Use the *inverse transpose* of the world matrix when transforming normals
  - Rotation remains the same
  - Scale becomes inverted
- Requires sending one more matrix to shader
  - Already part of your transform class!

Shader code for normals

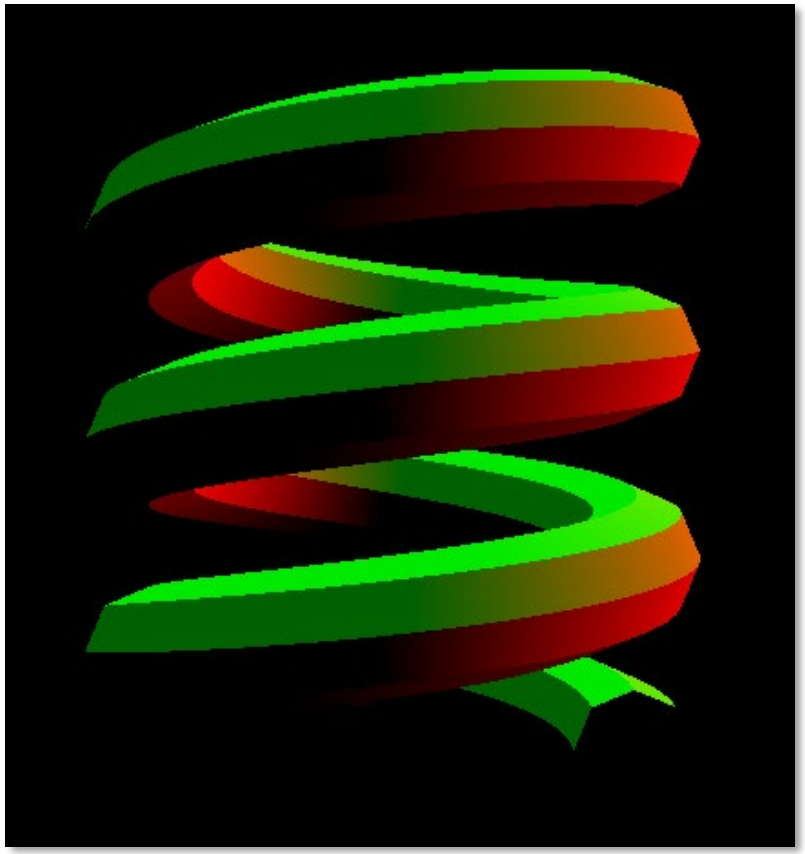
```
// VERTEX SHADER --------------
// Cast matrix to a 3x3 – removing translation
// Then normalize result
output.normal = mul(input.normal, (float3x3)worldInvTrans);
output.normal = normalize(output.normal);


// PIXEL SHADER --------------
// Must re-normalize interpolated normals!  Interpolation results in
// non-uniform vectors, since each component is interpolated independently.
input.normal = normalize(input.normal);
```

Normals in the Pixel Shader

- Normal passed in from vertex shader
  - But…how do we know if it worked?
  - How do we check the data?

- Return the normal as if it were a color!
  - Normal is float3 (x,y,z)
  - Return as our (r,g,b)

```
// Temporarily for testing...
return float4(input.normal, 1);
```



2: Light Sources

Light sources

- What about potential sources of light?
  - How do we simulate a large source of light like the sun?
  - Is everything just a light bulb?
  - What about flash lights?

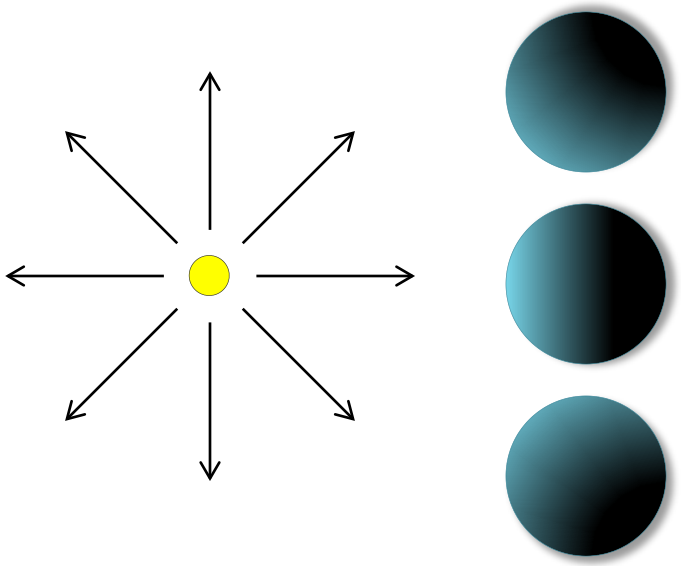
- We use *punctual* light sources
  - “Pertaining to or of the nature of a point”

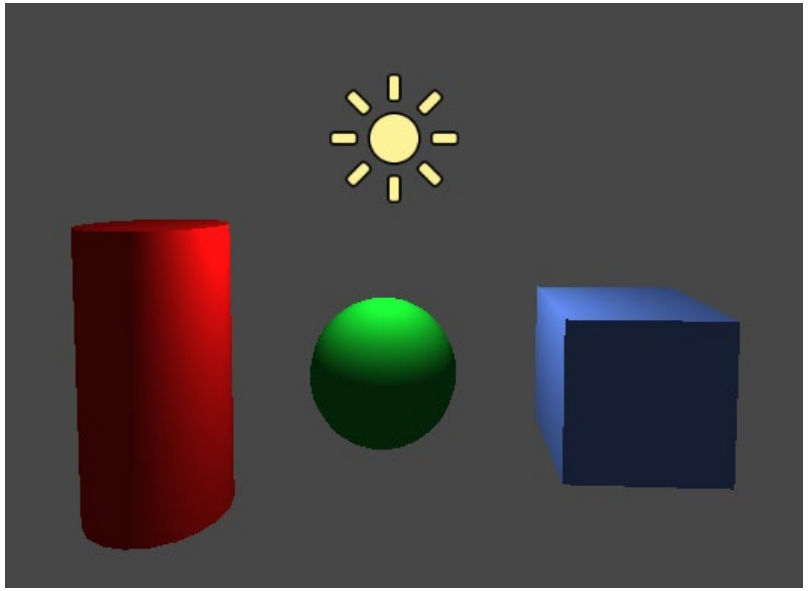
Light source types

- Directional Light
  - All rays are parallel
  - Often simulates sunlight
- Point Light
  - Like a light bulb
  - Light “emits” from a point (in all directions)
- Spot Light
  - Like a flash light
  - Light “emits” from a point in a cone shape

Point light

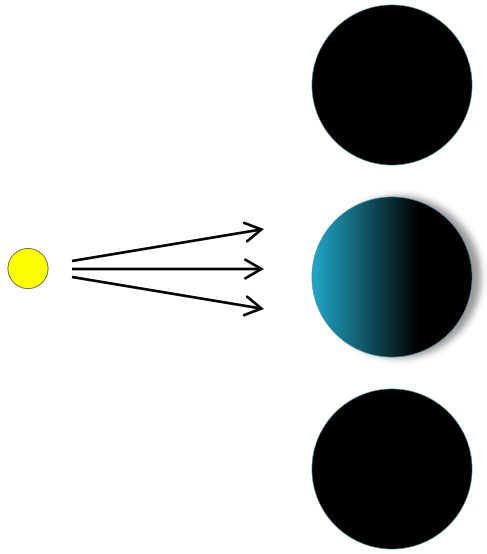
- All light rays come from a single point
  - Direction is calculated at each pixel
  - Vector between the light and the surface
  - Must pass *world* (not screen) position into pixel shader

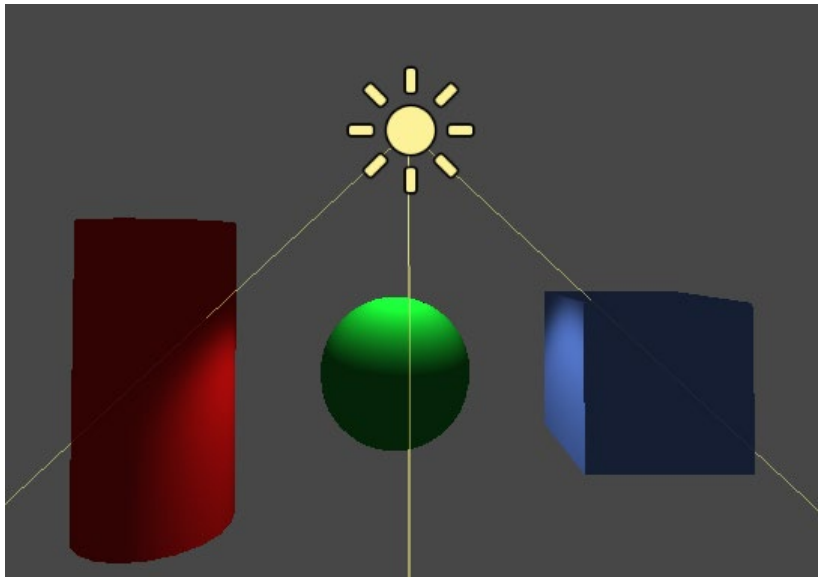




Spot light

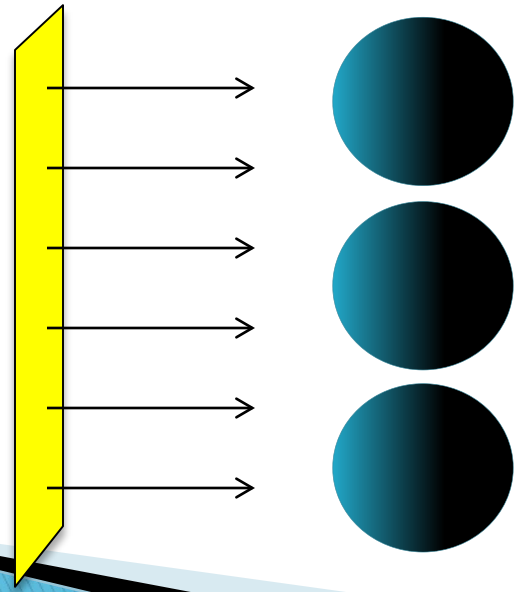
- Like a flash light – Conical shape
- Similar to a point light, with restrictions on the angle

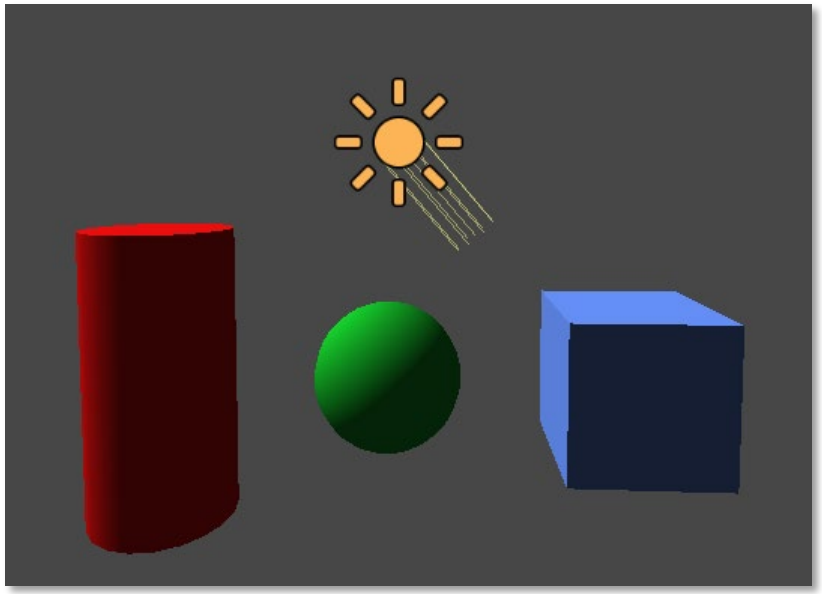




Directional light

- All light “travels” in the same direction (parallel)
- Uniform light direction (no calculation necessary)
  - Directional lights have no position!

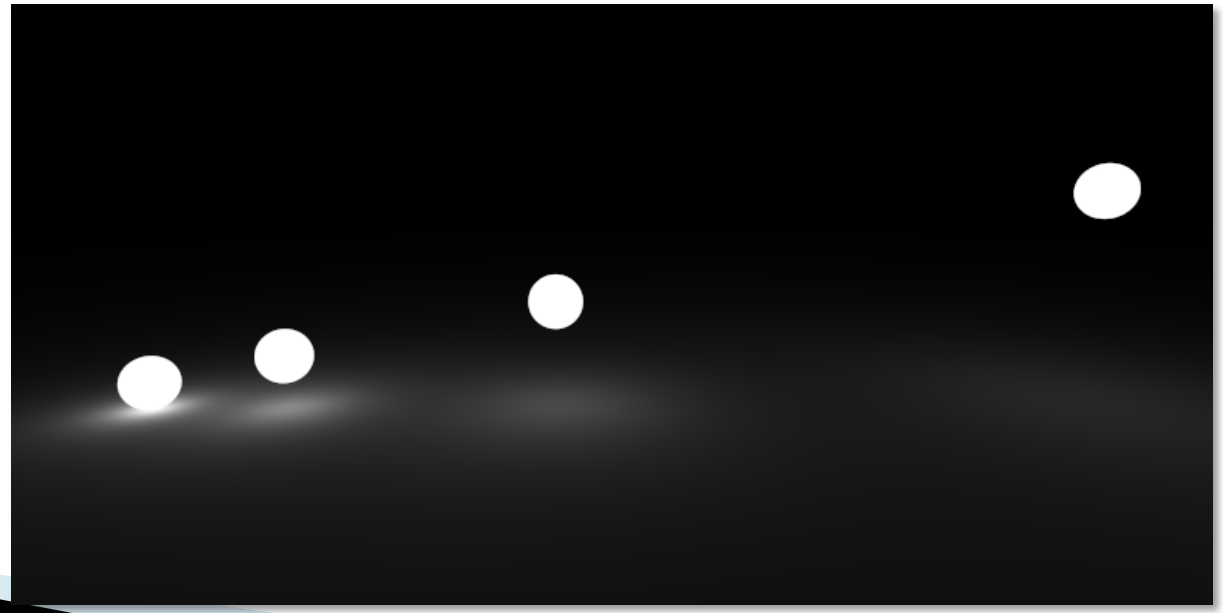




Attenuation

- Light intensity should **attenuate** (weaken) with distance
- Otherwise light is always equally bright regardless of distance

- More distance = less light



Which lights should attenuate?

- **Point** & **Spot** lights
  - Have a position
  - Need a range, too
  - Can properly attenuate

- **Directional** lights
  - All light in same direction
  - No “source position”
  - Assumes the light goes on forever
  - At uniform intensity
  - Never attenuate

3. Lighting Equations

BRDFs – Our lighting equations

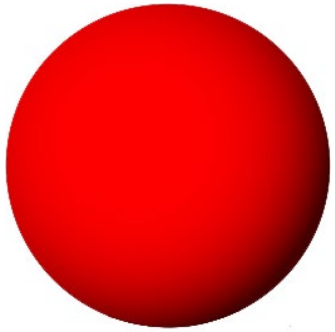
- Mathematical functions to calculate how much light reflects
- Bidirectional Reflectance Distribution Functions (BRDFs)
- Approximates how light reflects, given:
  - The incoming light direction
  - The surface normal
  - Our eye position (sometimes)

Where do BRDF's come from?

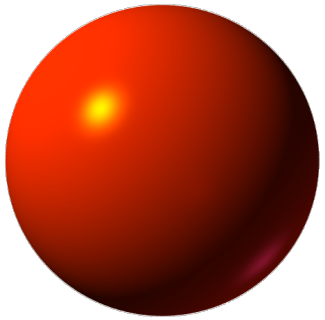
- Modeled after the physical world
- Each approximates a single property of light
  - Diffusion
  - Specularity
  - Anisotropy
  - Etc.
- We'll use a combination to make objects look realistically lit

Examples of BRDFs

- **Lambertian**: Basic, diffuse lighting



- **Phong**: Specularity (bright reflections)
  - **Blinn-Phong**: Similar to Phong
  - Less accurate but faster to compute



- **Cook-Torrance** (Microfacet)
  - Physically-based
  - Specular BRDF
  - Handles more surface properties

“The Rendering Equation”

$$L_{\mathrm{o}}(\mathbf{x}, \omega_{\mathrm{o}}, \lambda, t) = L_{\mathrm{e}}(\mathbf{x}, \omega_{\mathrm{o}}, \lambda, t) + \int_{\Omega} f_{\mathrm{r}}(\mathbf{x}, \omega_{\mathrm{i}}, \omega_{\mathrm{o}}, \lambda, t) L_{\mathrm{i}}(\mathbf{x}, \omega_{\mathrm{i}}, \lambda, t)(\omega_{\mathrm{i}} \cdot \mathbf{n}) \, \mathrm{d}\omega_{\mathrm{i}}$$

- Introduced in 1986
  - Literally called “The Rendering Equation”
  - Original papers here and here
- Calculates the radiance leaving a point as the sum of emitted and reflected radiance
- Spoiler: We’re not doing all this math!!!

Deciphering the Rendering Equation: Output

$$L_{\mathrm{o}}(\mathbf{x}, \omega_{\mathrm{o}}, \lambda, t) = L_{\mathrm{e}}(\mathbf{x}, \omega_{\mathrm{o}}, \lambda, t) + \int_{\Omega} f_{\mathrm{r}}(\mathbf{x}, \omega_{\mathrm{i}}, \omega_{\mathrm{o}}, \lambda, t) L_{\mathrm{i}}(\mathbf{x}, \omega_{\mathrm{i}}, \lambda, t)(\omega_{\mathrm{i}} \cdot \mathbf{n}) \, \mathrm{d}\omega_{\mathrm{i}}$$

$$L_{\mathrm{o}}(\mathbf{x}, \omega_{\mathrm{o}}, \lambda, t)$$

Outgoing light
(spectral radiance)

Position in space

Direction of outgoing light

Light wavelength

Time

Deciphering: Emitted light

$$L_{\mathrm{o}}(\mathbf{x}, \omega_{\mathrm{o}}, \lambda, t) = L_{\mathrm{e}}(\mathbf{x}, \omega_{\mathrm{o}}, \lambda, t) + \int_{\Omega} f_{\mathrm{r}}(\mathbf{x}, \omega_{\mathrm{i}}, \omega_{\mathrm{o}}, \lambda, t) L_{\mathrm{i}}(\mathbf{x}, \omega_{\mathrm{i}}, \lambda, t)(\omega_{\mathrm{i}} \cdot \mathbf{n}) \, \mathrm{d}\, \omega_{\mathrm{i}}$$

$$L_{\mathrm{e}}(\mathbf{x}, \omega_{\mathrm{o}}, \lambda, t)$$

Emitted light
(spectral radiance)

Position in space

Direction of outgoing light

Light wavelength

Time

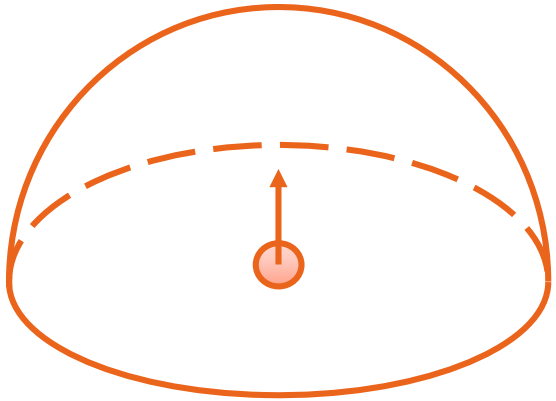
Deciphering: All incoming light

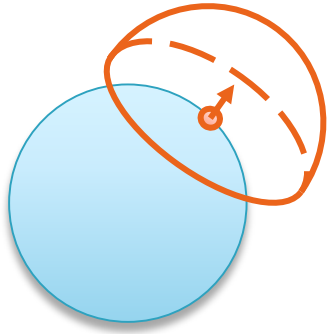
$$L_{\mathrm{o}}(\mathbf{x}, \omega_{\mathrm{o}}, \lambda, t) = L_{\mathrm{e}}(\mathbf{x}, \omega_{\mathrm{o}}, \lambda, t) + \int_{\Omega} f_{\mathrm{r}}(\mathbf{x}, \omega_{\mathrm{i}}, \omega_{\mathrm{o}}, \lambda, t) L_{\mathrm{i}}(\mathbf{x}, \omega_{\mathrm{i}}, \lambda, t)(\omega_{\mathrm{i}} \cdot \mathbf{n}) \, \mathrm{d}\, \omega_{\mathrm{i}}$$

$$\int_{\Omega}$$

Integral over Ω

Ω is a **hemisphere** encompassing *directions of all incoming light* (ω_i) that can strike this position in space





Deciphering: BRDF

$$L_{\mathrm{o}}(\mathbf{x}, \omega_{\mathrm{o}}, \lambda, t) = L_{\mathrm{e}}(\mathbf{x}, \omega_{\mathrm{o}}, \lambda, t) + \int_{\Omega} f_{\mathrm{r}}(\mathbf{x}, \omega_{\mathrm{i}}, \omega_{\mathrm{o}}, \lambda, t) L_{\mathrm{i}}(\mathbf{x}, \omega_{\mathrm{i}}, \lambda, t)(\omega_{\mathrm{i}} \cdot \mathbf{n}) \, \mathrm{d}\omega_{\mathrm{i}}$$

$$f_{\mathrm{r}}(\mathbf{x}, \omega_{\mathrm{i}}, \omega_{\mathrm{o}}, \lambda, t)$$

BRDF

Position in space

Negative direction*
of incoming light
(direction TO the light)

* *All* directions over Ω

Direction of outgoing light

Light wavelength

Time

Deciphering: Incoming light

$$L_{\mathrm{o}}(\mathbf{x}, \omega_{\mathrm{o}}, \lambda, t) = L_{\mathrm{e}}(\mathbf{x}, \omega_{\mathrm{o}}, \lambda, t) + \int_{\Omega} f_{\mathrm{r}}(\mathbf{x}, \omega_{\mathrm{i}}, \omega_{\mathrm{o}}, \lambda, t) L_{\mathrm{i}}(\mathbf{x}, \omega_{\mathrm{i}}, \lambda, t)(\omega_{\mathrm{i}} \cdot \mathbf{n}) \, \mathrm{d}\omega_{\mathrm{i}}$$

$$L_{\mathrm{i}}(\mathbf{x}, \omega_{\mathrm{i}}, \lambda, t)$$

Incoming light
(spectral radiance)

Position in space

Negative direction*
of incoming light
(direction TO the light)

Time

Light wavelength

* *All* directions over Ω

Deciphering: The rest

$$L_{\mathrm{o}}(\mathbf{x}, \omega_{\mathrm{o}}, \lambda, t) = L_{\mathrm{e}}(\mathbf{x}, \omega_{\mathrm{o}}, \lambda, t) + \int_{\Omega} f_{\mathrm{r}}(\mathbf{x}, \omega_{\mathrm{i}}, \omega_{\mathrm{o}}, \lambda, t) L_{\mathrm{i}}(\mathbf{x}, \omega_{\mathrm{i}}, \lambda, t) (\omega_{\mathrm{i}} \cdot \mathbf{n}) \, \mathrm{d}\, \omega_{\mathrm{i}}$$

$$(\omega_{\mathrm{i}} \cdot \mathbf{n}) \, \mathrm{d}\, \omega_{\mathrm{i}}$$

Negative direction* of incoming light (direction TO the light)

* *All* directions over Ω

normal

Dot product

Differential of negative direction of incoming light (solid angle)

The Rendering Equation for us

$$L_{\mathrm{o}}(\mathbf{x}, \omega_{\mathrm{o}}, \lambda, t) = L_{\mathrm{e}}(\mathbf{x}, \omega_{\mathrm{o}}, \lambda, t) \; + \int_{\Omega} f_{\mathrm{r}}(\mathbf{x}, \omega_{\mathrm{i}}, \omega_{\mathrm{o}}, \lambda, t) L_{\mathrm{i}}(\mathbf{x}, \omega_{\mathrm{i}}, \lambda, t)(\omega_{\mathrm{i}} \cdot \mathbf{n}) \, \mathrm{d}\, \omega_{\mathrm{i}}$$

- We’re obviously not using the whole thing
- Where are we simplifying?

The Rendering Equation for us

$$L_o(\mathbf{x}, \omega_o, \cancel{\lambda}, \cancel{t}) = \cancel{L_e(\mathbf{x}, \omega_o, \lambda, t)} + \cancel{\int_\Omega} f_r(\mathbf{x}, \omega_i, \omega_o, \cancel{\lambda}, \cancel{t}) L_i(\mathbf{x}, \omega_i, \cancel{\lambda}, \cancel{t})(\omega_i \cdot \mathbf{n}) \, \cancel{d}\omega_i$$

- **Wavelength?** We assume RGB are similar
- **Time?** Single point in time
- **Emitted light?** Nope (but could be emissive texture)
- **Integral** over a hemisphere?
  - WAY too expensive for a shader!
  - We only handle light coming directly from a light source
  - And only a single direction per light
- Don't need that **solid angle** anymore, either

The Rendering Equation – What's left?

$$L_o(\mathbf{x}, \omega_o) = f_r(\mathbf{x}, \omega_i, \omega_o) L_i(\mathbf{x}, \omega_i)(\omega_i \cdot \mathbf{n})$$

BRDF (f), light source (L) and a dot product

- However, need to account for *all lights* in a scene
  - Multiple lights linearly increase brightness
  - So this is a simple sum of all results

$$L_o(\mathbf{x}, \omega_o) = \sum_{light=1}^{numLights} f_r(\mathbf{x}, \omega_i, \omega_o) L_i(\mathbf{x}, \omega_i)(\omega_i \cdot \mathbf{n})$$

The Rendering Equation as pseudocode

$$L_o(\mathbf{x}, \omega_o) = \sum_{light=1}^{numLights} f_r(\mathbf{x}, \omega_i, \omega_o) L_i(\mathbf{x}, \omega_i)(\omega_i \cdot \mathbf{n})$$

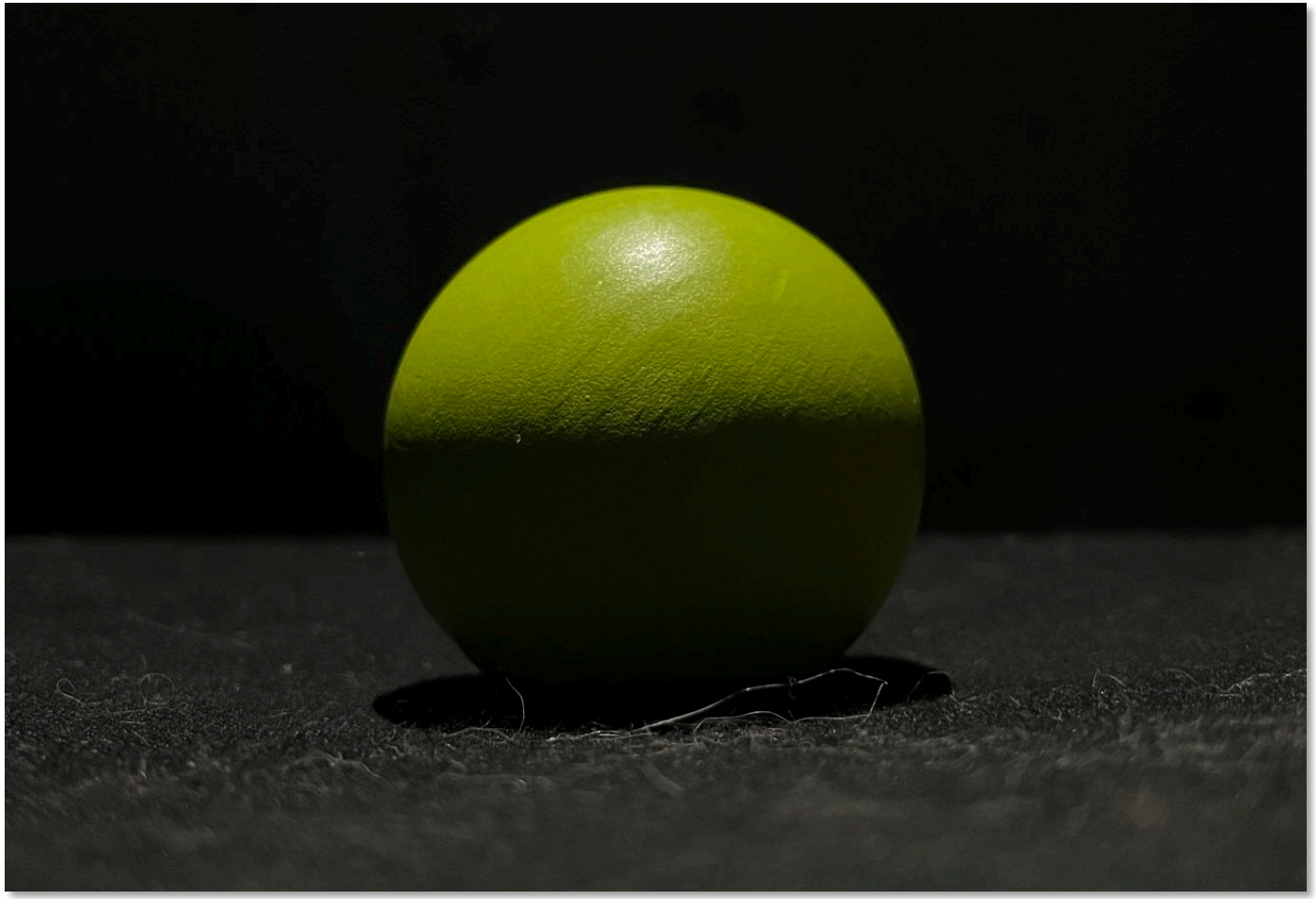
- HLSL pseudocode based on the above formula:

```
float3 total = float3(0,0,0);
for (int i = 0; i < numLights; i++)
{
   Light light = lights[i];
   total += BRDF(light, surface) * LightColor(light) * dot(-light.Direction, normal);
}
return total;
```

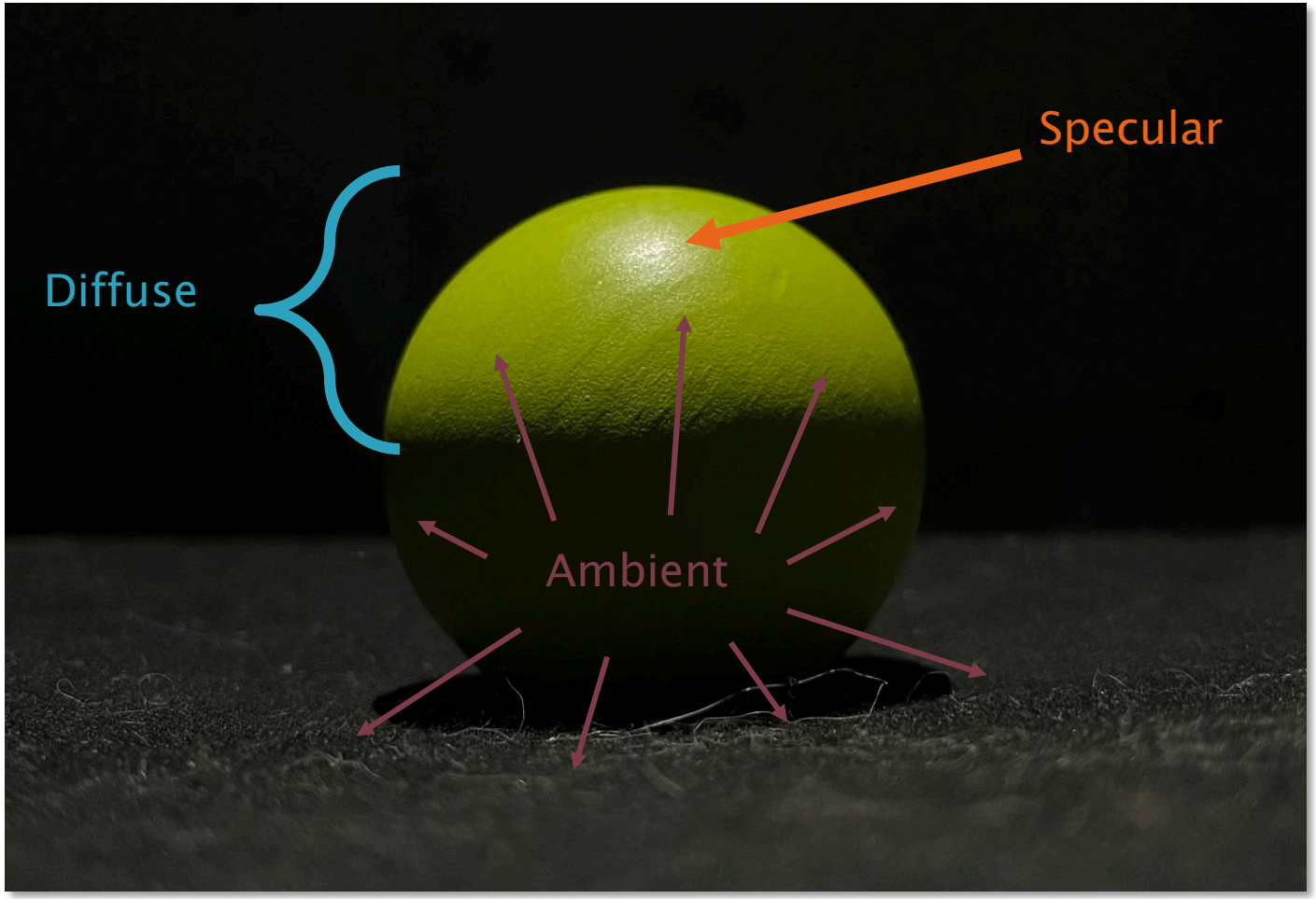
Lighting Approximations

And their BRDFs

Photo – What should we approximate?

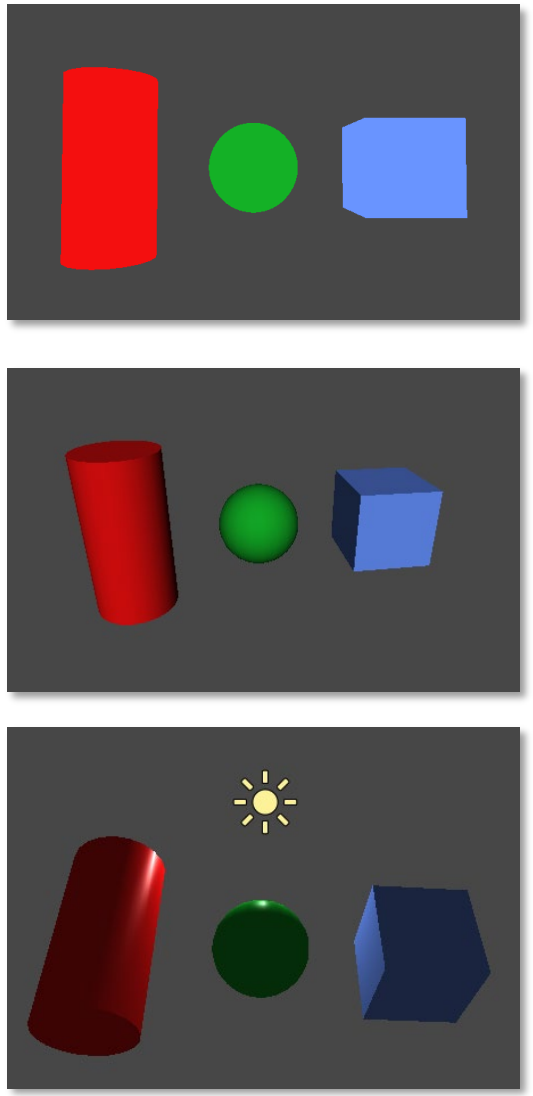


Classic computer graphics light approximations



The big three

- Ambient
  - Approximates light bouncing around
  - *Incredibly* simplistic model of complex system
- Diffuse
  - Reflectance (brightness) of the surface
- Specular
  - Reflection of the light source



Ambient

- Approximation of light bouncing around
- Minimum light level & color of scene
- Implementation:
  - Pick a subtle color that matches the scene
  - That's your ambient color
- No actual "calculation" here
  - Gotta go fast!

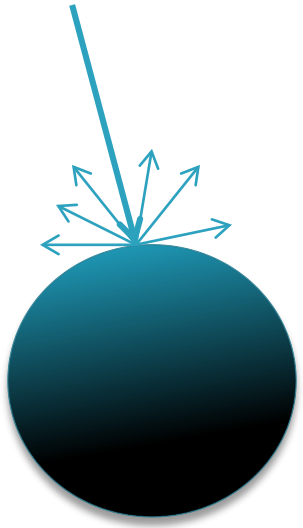
Shader – Ambient implementation

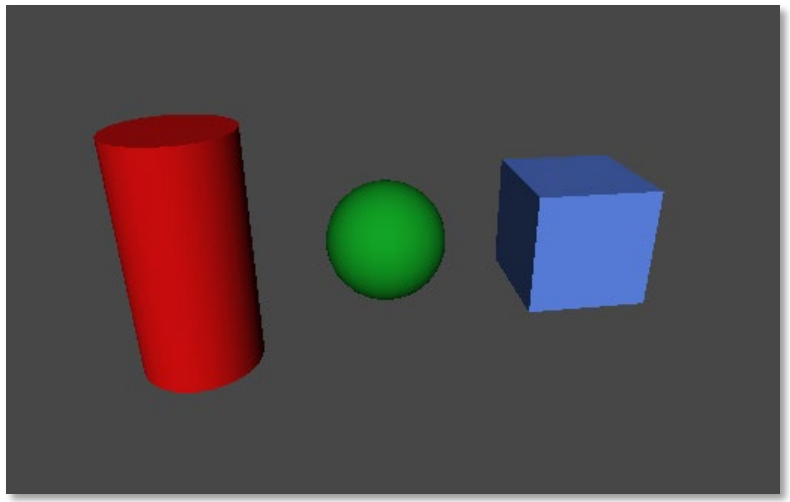
```
// Pretty simple
float3 ambientTerm = ambientColor * surfaceColor;


// ambientColor is coming from a constant buffer
// surfaceColor is currently object's color tint
//   - Will eventually come from a texture
```

Diffuse light

- Hits & spreads out in all directions

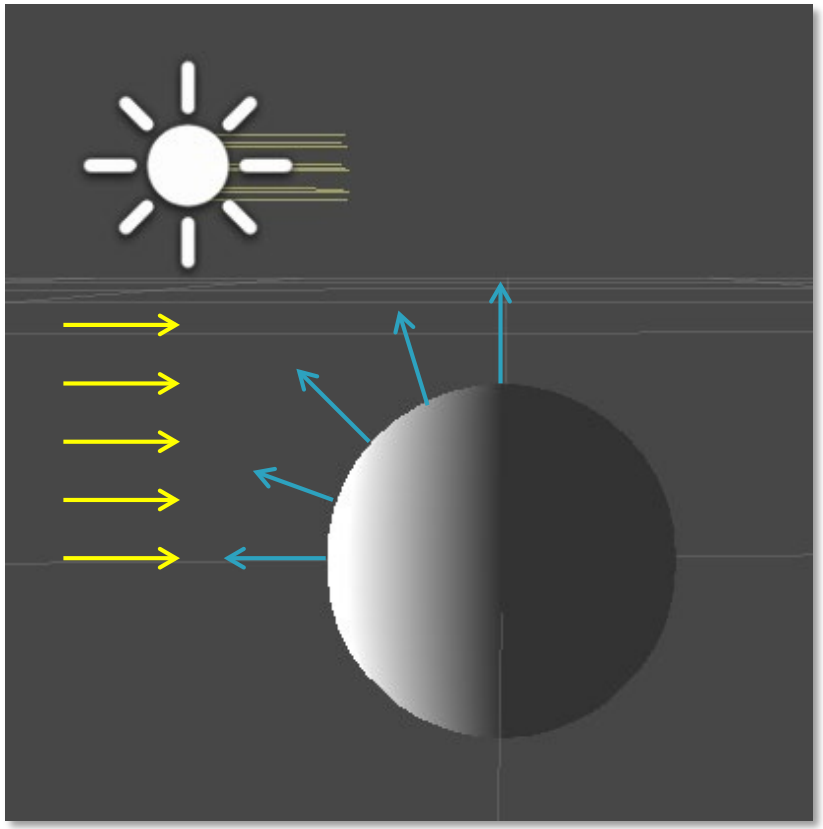
- Looks the same from any viewing angle
  - Camera's position doesn't matter
  - Only light direction and surface normal matter





Diffuse lighting

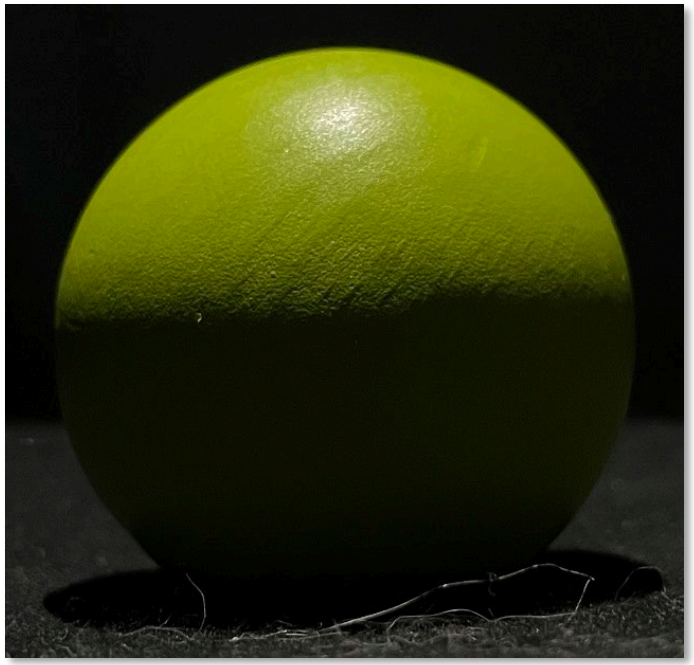
- Compare **surface normal** & **light direction**
- Brighter where light hits “head on”
- Less bright where light “glances off”
- Lambert’s Cosine Law
  - “N dot L”



Diffuse – Lambertian reflectance

- Approximates light leaving a perfectly diffuse surface

- BRDF: $I_D = \mathbf{L} \cdot \mathbf{N} C I_L$
  - L is normalized vector from surface to light
  - N is surface normal
  - C is surface color
  - I_L is light color & intensity

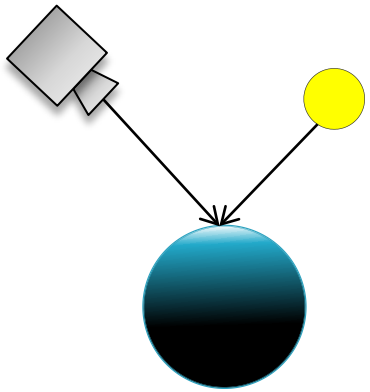


- Implementation:
  - Dot product between two unit vectors
  - Multiply by light (color & intensity) and surface color!

Shader – Diffuse implementation

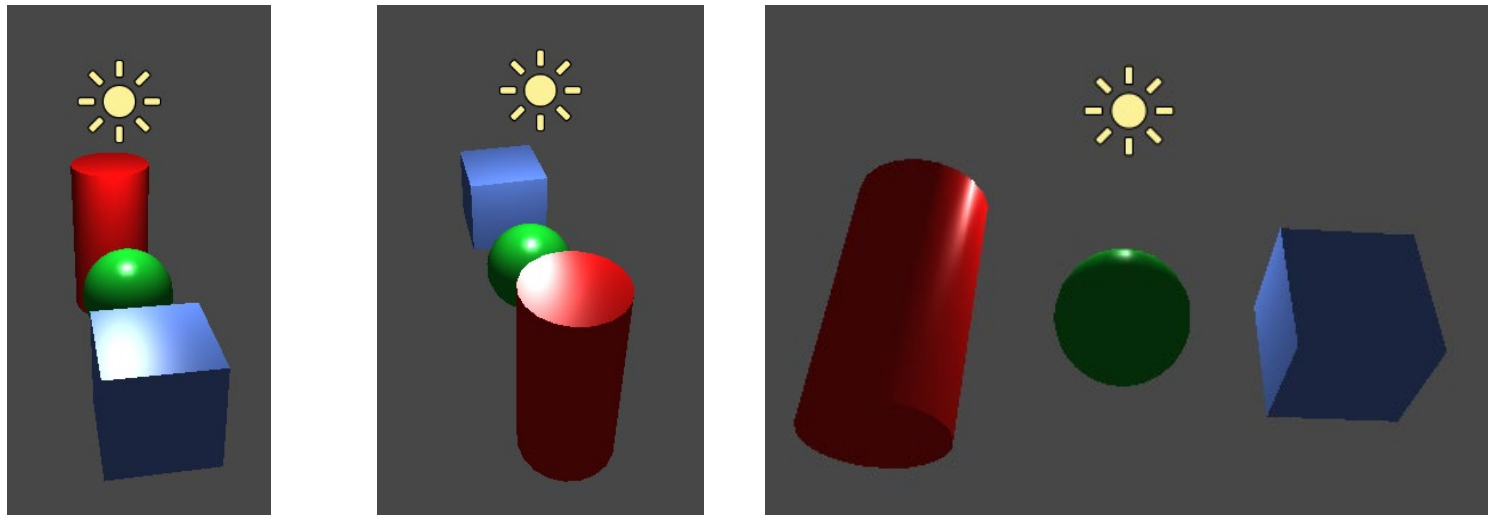
```
// Get direction TO the light (opposite of light's direction)
float3 dirToLight = -lightDirection;

// Lambertian reflectance is really just a dot product
float3 diffuseTerm =
   saturate(dot(input.normal, dirToLight)) * // Diffuse intensity, clamped 0-1
   lightColor * lightIntensity *   // Light's overall
   surfaceColor;                   // Light tints the surface

// normal is pixel shader input (from rasterizer)
// light's direction, color & intensity from cbuffer
// surfaceColor is the material's color tint (Will eventually come from a texture)
```

Specular light



- Simulates “shine” on a surface
  - Basic reflection of the light source

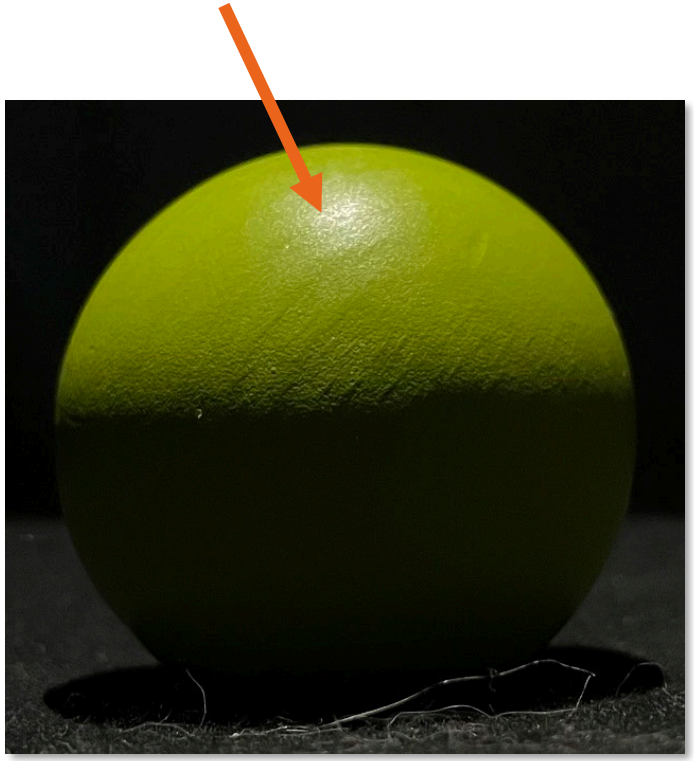
- Looks different based on your viewing angle



Specular – Phong

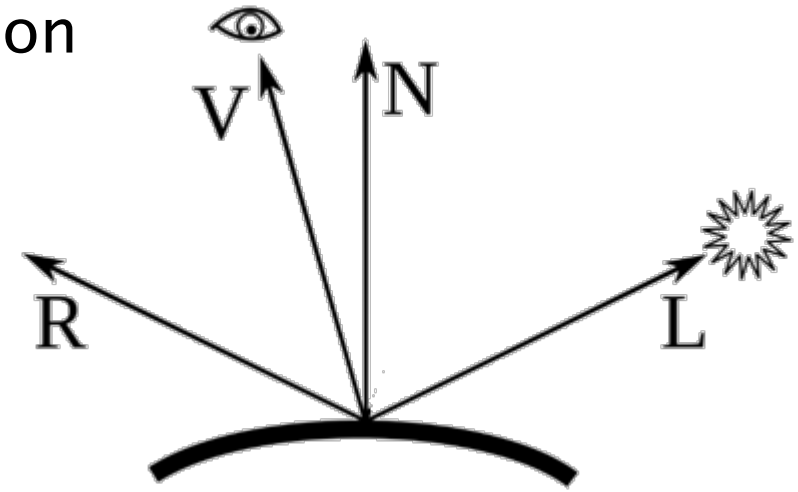
- Approximates shiny reflection
- BRDF: $k_s(\hat{R}_m \cdot \hat{V})^{\alpha} i_{m,s}$
  - k_s is specular constant (1.0 usually)
  - R_m is normalized light reflection vector
  - V is normalized vector from surface to viewer
  - α is material's shininess constant
  - i_m is light intensity
  - i_s is specular intensity
- Implementation
  - Dot product of vectors, raised to a power
  - Multiply by intensities

Just the shiny bits



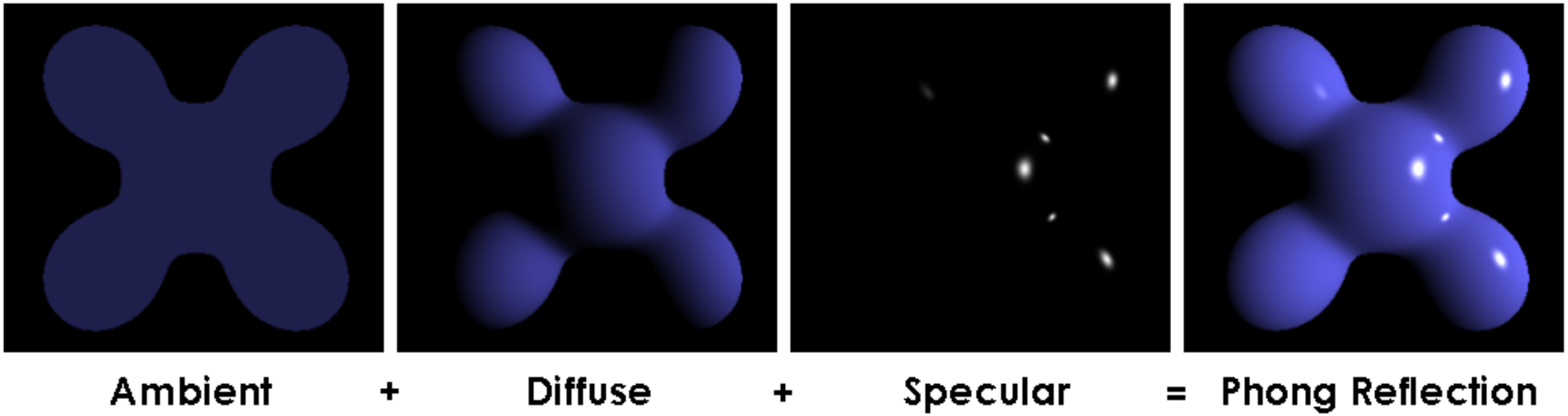
Phong reflection model

- Calculate the reflection vector (R)
  - Based on the Normal (N) and Light (L) vectors
- Compare reflection vector to view vector
  - Use dot product to get cosine of angle
  - Within certain angle, we should “see” reflection
- Raise result to a power for a smooth falloff



Phong reflectance results visualized

- Three separate approximations, added together



Shader – Phong implementation

```
// Calculate the reflection of the INCOMING light dir
// using the surface normal (ensure both are normalized)
float3 refl = reflect( lightDir, input.normal );

// Calculate cosine of the reflection and camera vector
// - saturate() ensures it's between 0 and 1
float RdotV = saturate( dot( refl, dirToCamera ) );

// Raise the that to a power equal to some "shininess"
// factor – try large numbers here (64, 200, etc.)
float3 specularTerm = pow(RdotV, shininess) *
   lightColor * lightIntesity * surfaceColor;
```

Total light for a single pixel

- Overall calculation looks like this:

$$I_{\mathrm{p}} = i_{\mathrm{a}} + \sum_{m \in \text{lights}} \left((\hat{L}_m \cdot \hat{N}) i_{m,\mathrm{d}} + (\hat{R}_m \cdot \hat{V})^{\alpha} i_{m,\mathrm{s}} \right)$$

Final light intensity for a pixel

Ambient term

Add up the following for each light

Diffuse (Lambert)

Specular (Phong)

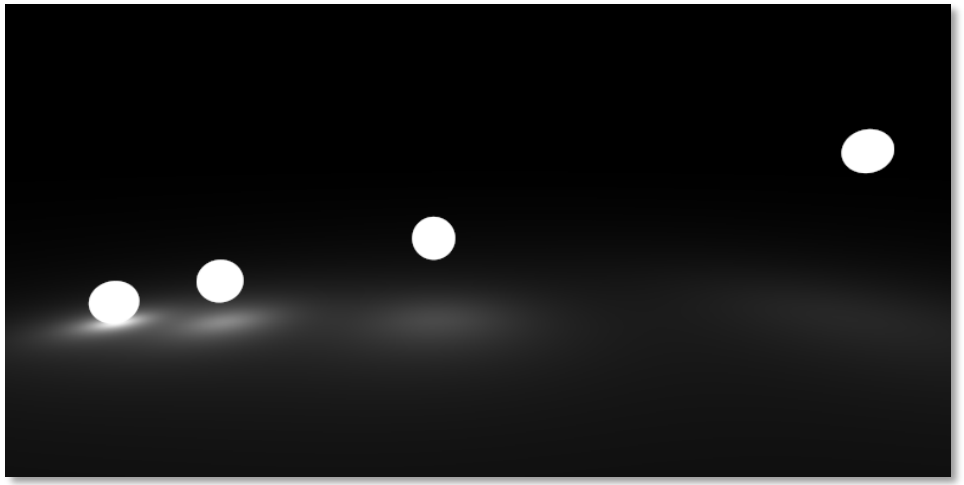
Shader – Final shading value

```
float3 totalShading = ambientTerm;

// Do this once PER LIGHT
// Each light will have its own diffuse and specular result
totalShading += diffuseTerm + specularTerm;

// Return this from our pixel shader
return float4(totalShading, 1);
```

Attenuation

Attenuation

- Light intensity should **attenuate** (weaken) further from source
  - Otherwise light is always equally intense
  - Greater distance = less light



- Attenuate **point** & **spot** lights
- Can't attenuate **directional**
  - Has no position
  - Can't measure distance

- But how much?

Attenuation functions

- Many exist, all have the same effect:
  - Result decreases as distance increases

- 1.0 / (1.0 + a * dist + b * dist * dist))
  - Classic computer graphics attenuation function
  - “a” is linear falloff value
  - “b” is quadratic falloff value
  - “dist” is distance between light and pixel

  - This one *isn’t great* for our purposes!

Common attenuation example

- Tweakable and looks fine, but…
  - Parameters don't represent exact range
  - Doesn't hit zero at a finite distance

Better attenuation function?

- We need a *range-based* function

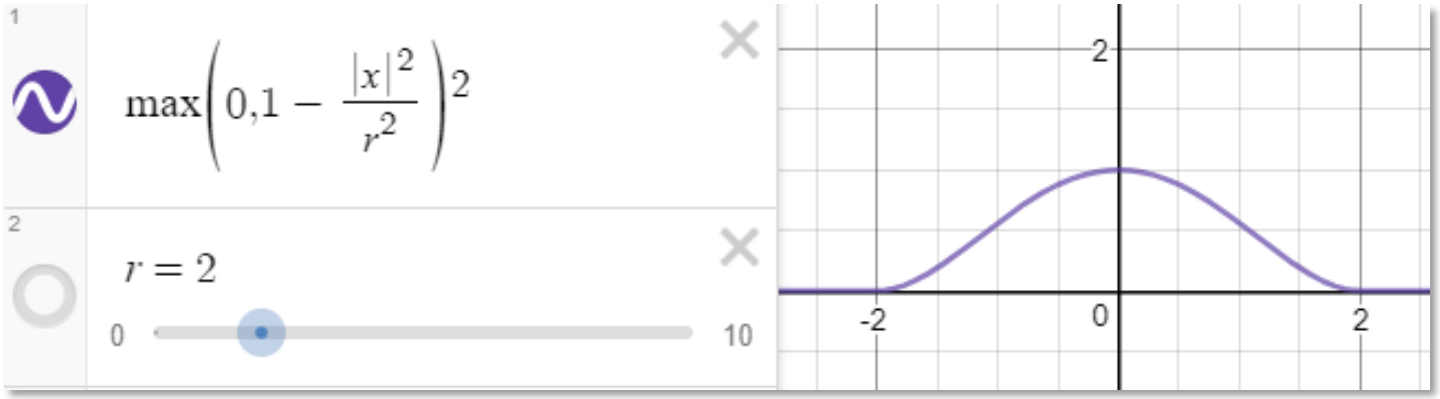
- An attenuation function that…
  - Goes to zero at a *specific distance* (range)
  - Still has a nice smooth falloff
  - Ideally isn't linear

$$\max(0, 1.0 - \text{dist}^2 / \text{range}^2)^2$$

Range-based attenuation

- Example in HLSL
  - `dist` is actual distance between pixel and light
  - `radius` is light's "range"

```
att = saturate(1.0-dist*dist/(radius*radius));
att *= att;
```



Applying attenuation

```
float3 totalShading = ambientTerm;

// Do this once PER LIGHT
// Each light will have its own diffuse, specular and atten.
totalShading += (diffuseTerm + specularTerm) * attenuation;

// Return this from our pixel shader
return float4(totalShading, 1);
```

Representing Lights in Code

Sending light data to the shader

- A single light is made up of multiple pieces of data

Directional Light:
- Color
- Intensity
- Direction

Point Light:
- Color
- Intensity
- Position
- Range

Spot Light:
- Color
- Intensity
- Position
- Direction
- Range
- Spot Falloff

- How do we represent and send all this data to the shader?

Sending light data…poorly

```
cbuffer ExternalData : register(b0)
{
    float3 dirLight1Color;
    float  dirLight1Intensity;
    float3 dirLight1Direction;
    float3 dirLight2Color;
    float  dirLight2Intensity;
    float3 dirLight2Direction;
    float3 pointLight1Color;
    float  pointLight1Intensity;
    float3 pointLight1Position;
    float  pointLight1Range;
    // And on and on and on...
}
```

Please, I beg you, no…

Use a common struct for all lights!

```
#define LIGHT_TYPE_DIR     0
#define LIGHT_TYPE_POINT   1
#define LIGHT_TYPE_SPOT    2

// Make a matching struct and #defines on CPU (C++) side
struct LightStruct
{
	int    Type;
	float3 Direction;    // 16 bytes
	float  Range;
	float3 Position;     // 32 bytes
	float  Intensity;
	float3 Color;        // 48 bytes
	float  SpotFalloff;
	float3 Padding;      // 64 bytes
};
```

Can use these in your code when checking a light's Type

Has all properties to represent any light we want, and a Type so we can dynamically handle each one.

Padding at the end ensures it doesn't cross the 16-byte boundary.

Sending light data with structs

```
cbuffer ExternalData : register(b0)
{
    Light dirLight1;
    Light dirLight2;
    Light dirLight3;
    Light pointLight1;
    Light pointLight2;
}

// Meanwhile, in C++...
ps->SetData("dirLight1", &myLight, sizeof(Light));
```

You know, an array would make this even easier!

More on that in the future

Spot Light Example

Spot Light Calculation

```
// Start with the POINT LIGHT calculation

// Calculate the falloff from the center of the
// spot light's
// true direction (center of the cone) to the
// direction from the light to this pixel.
// Remember: lightDir is direction to THIS PIXEL.
float angleFromCenter = max(dot(-lightDir, light.direction), 0.0f);

// Raise to a power for a nice "falloff".
// Multiply the diffuse and specular results by this:
float spotAmount = pow(angleFromCenter, light.spotPower);
```