Raytracing

Adam Mally
CIS 560 Spring 2015
University of Pennsylvania

Before we begin

- Properly transforming surface normals from object space into world space is not as simple as multiplying them by the model matrix
  - Doing so skews them slightly, making them no longer normal (orthogonal) to the surface
- Given a point on a surface, there exists a surface normal n and some tangent vector t
  - $\text{Dot}(n, t) = n^T t = 0$
- When the object is transformed by a model matrix M, $Mt = t'$
  - The transformed normal n' must remain orthogonal to t', so we multiply n by some matrix S to get n'
  - $0 = (n')^T t'$
  - $0 = (Sn)^T Mt$
  - $0 = (n)^T S^T Mt = n^T t$
  - The above identity implies that $S^T M = I$, which in turn implies that $S^T = M^{-1}$, therefore $S = (M^{-1})^T$
- In summary, multiply the local-space surface normal by the *inverse transpose* of the model matrix to correctly bring it into world space

What is raytracing?

- Computer graphics technique used to simulate the way light bounces off surfaces
- Render fairly realistic-looking images without having to fully simulate the infinitely-many photons light sources give off
- Trace light paths backwards, from camera to light source(s)

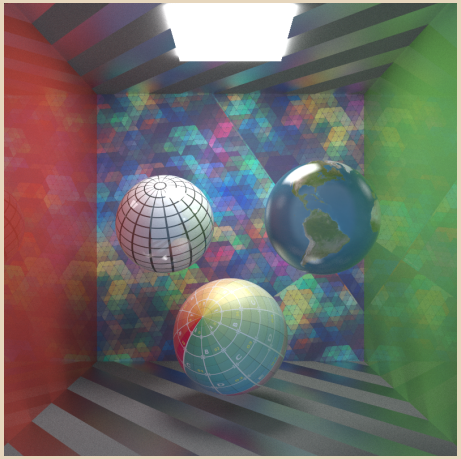


Image source: http://www.yiningkarlli.com/

Basic Structure of a Raytracer

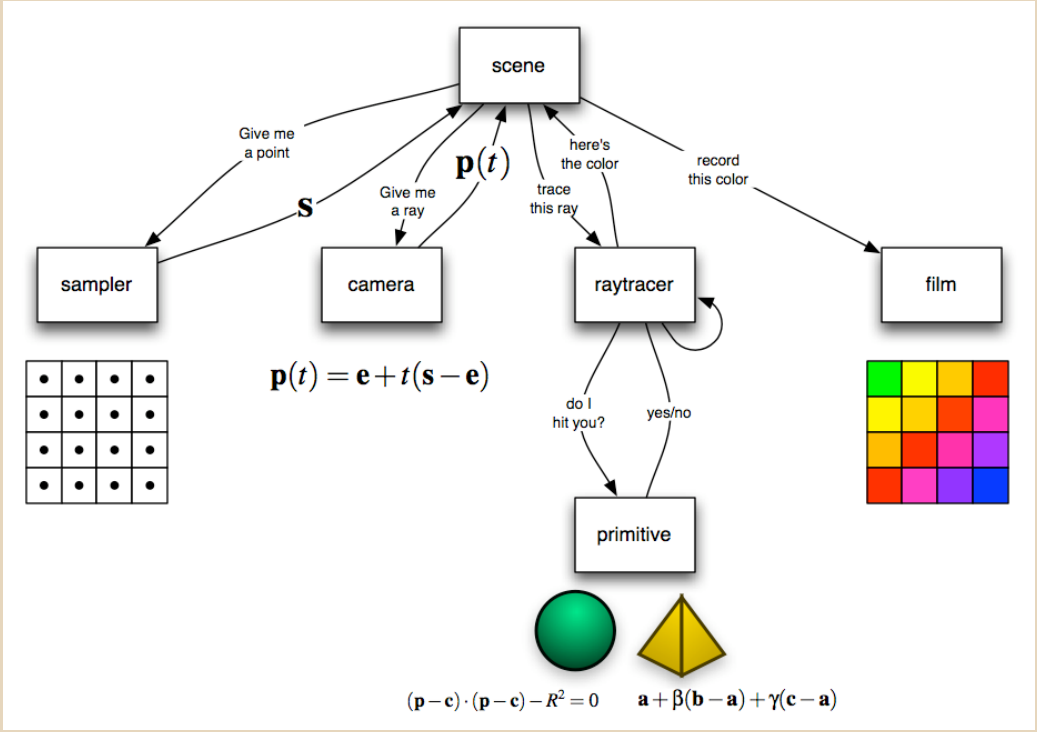


Image source: UC Berkeley CS184

Basic Structure of a Raytracer

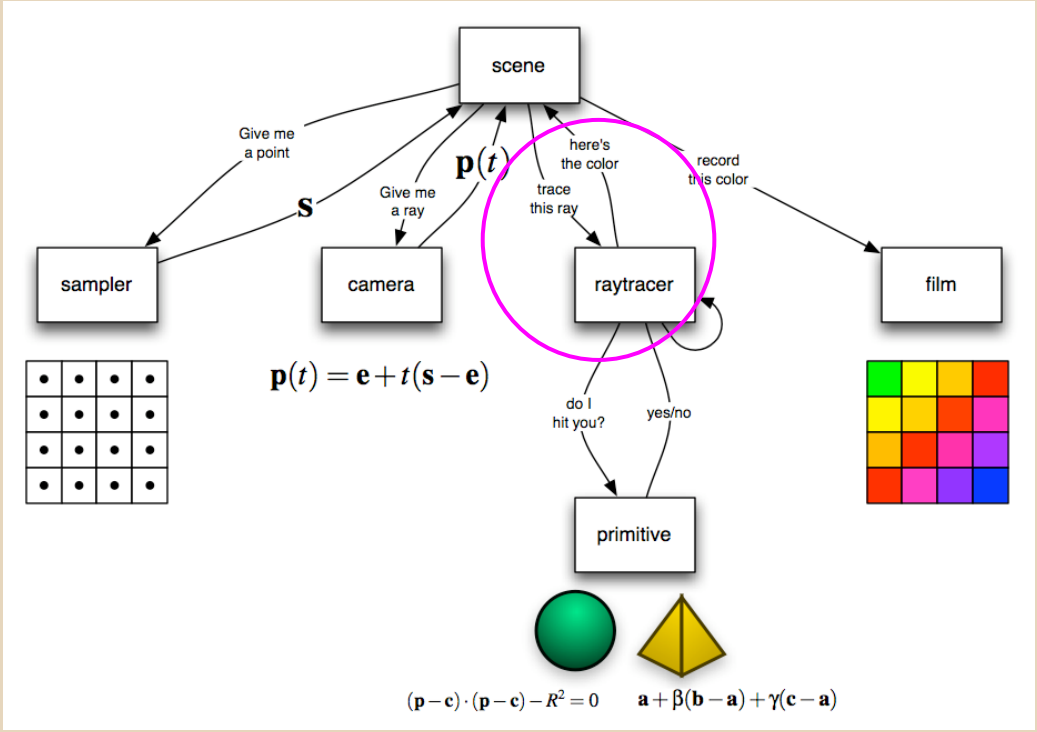
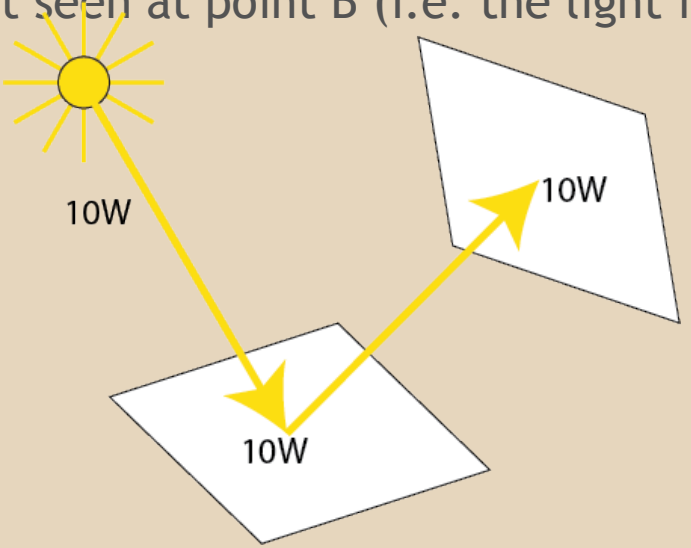


Image source: UC Berkeley CS184

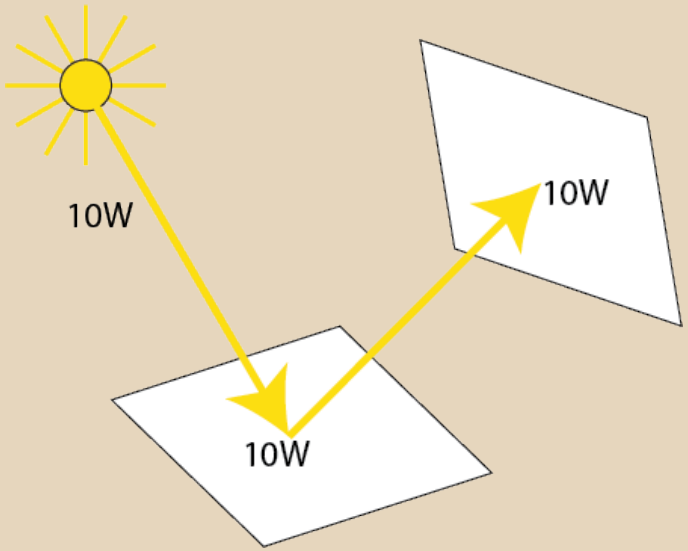
Radiometry

- There are several important properties of light that make up the foundation of raytracing (excluding certain phenomena such as black holes)
  - A light ray travels in a perfectly straight line from one point to another
  - Light rays do not interfere with one another if they intersect
  - Given two points in space that can directly see one another, the amount of light emitted from point A towards point B is the same amount of light seen at point B (i.e. the light is invariant)



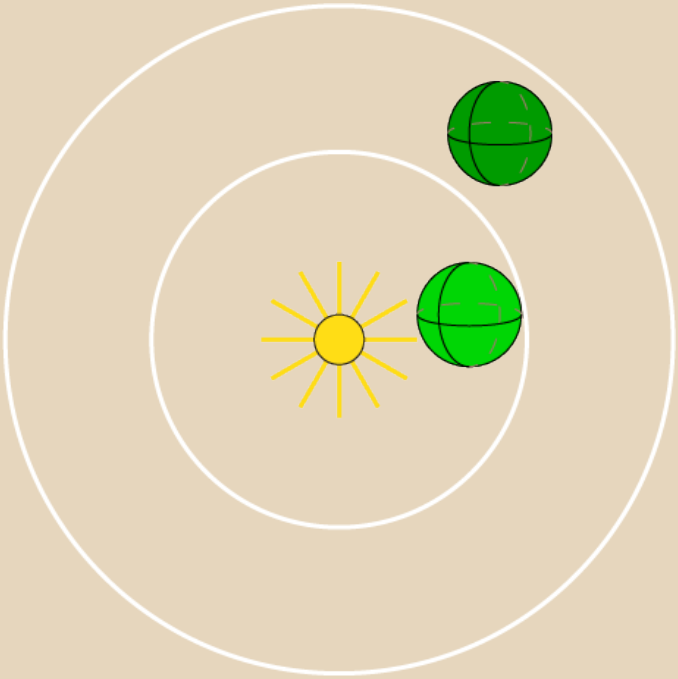
Radiometry

- Within radiometry, there is a system of units and measures for illumination
- Take an optics (geometric) approach to light
- Again, treat light as if it travels in perfectly straight lines
  - Excellent approximation when the light's wavelength is much smaller than the objects with which the light interacts
  - Cannot model diffraction, interference, etc.



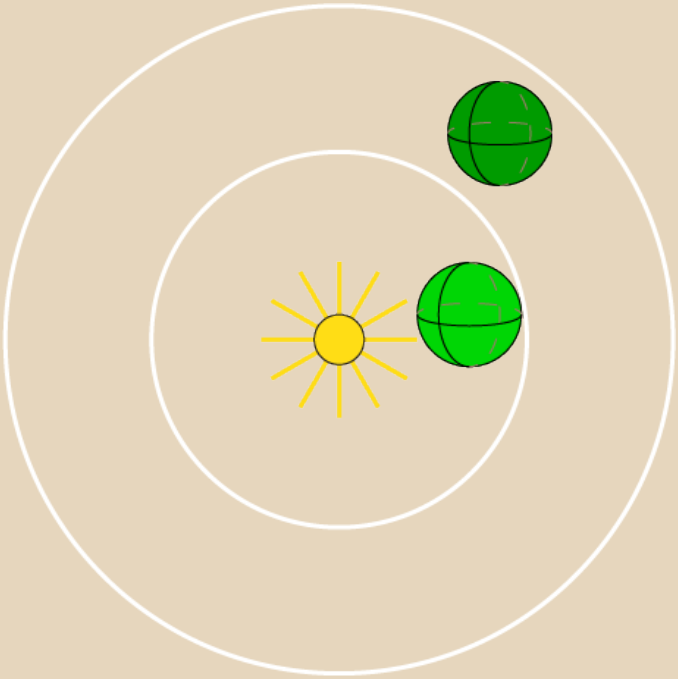
The physics of lighting: Flux

- We want to simulate the gradual shading that objects exhibit in real life
- By using various shading models (Lambert, Phong, Gouraud, etc.) we can approximate the energy reflected at a surface point
- A light's intensity reduces as it travels away from its source
  - **Radiant flux** is the amount of energy passing through a region of space per unit of time, measured in Joules/sec (aka Watts) and commonly denoted as Φ
  - Both spheres surrounding the point light have the same amount of total **flux**, but any one local area of the larger sphere has less flux than a local area on the smaller sphere
    - Hence objects further from the light being more weakly illuminated



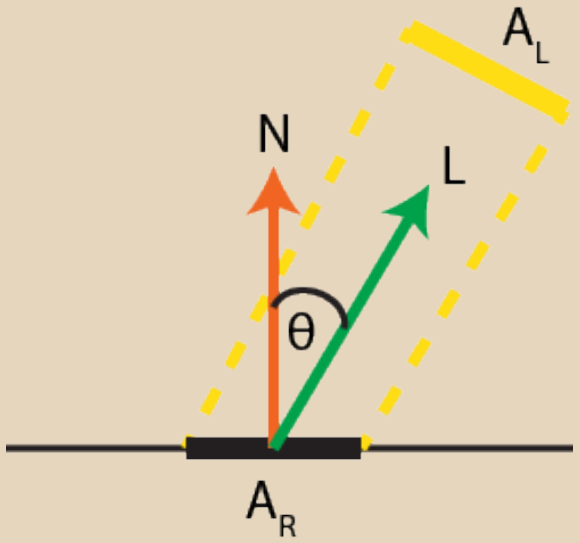
The physics of lighting: Irradiance

- We can represent the amount of **flux** arriving at a surface as **irradiance** (*E*), and the amount of **flux** leaving a surface as **radiant exitance** (*M*)
- Their unit of measurement is Watts/meter^2
- The **irradiance equation** for our point light emission sphere is:
  $E = \Phi/(4\pi r^2)$
  - This is just flux/surface area
- The amount of energy (illumination) from this light falls off proportionally to the squared distance from the light since it becomes diffused over an increasingly larger area



The physics of lighting: Lambert's law

- We can use the irradiance equation to help understand the origin of **Lambert's law**
  - Lambert's law: the amount of light arriving at a surface is proportional to the cosine of the angle between the light direction and surface normal
  - $E = \Phi cos(\theta)/A_L$
- As θ increases, the area of the lit surface increases, meaning the flux is distributed across a larger surface area, causing the irradiance to decrease for any one point on the lit surface
- Review: $dot(A, B) = |A||B|cos(\theta)$
  - θ = angle between A and B
- Can re-write Lambert's equation as $E = \Phi dot(N, L)/A_L$
  - Assuming N and L are normalized, of course
- The amount of light reaching a single point on a surface from a point light ($A_L = 0$) can be represented simply as $clamp(dot(N, L), 0, 1)$



Phong shading

- Developed by Bui Tuong Phong in his 1973 Ph.D. dissertation
- Combination of diffuse reflection (e.g. Lambert shading), specular highlights, and ambient light
- Each shading component is weighted by a constant (k_d, k_s, k_a) and the sum of the constants is 1
- The Phong shading equation is $k_dD + k_sS + k_aA$
  - D is the diffuse lighting calculation, S is the specular highlight calculation, and A is simply the color of the ambient light
- Not physically accurate!

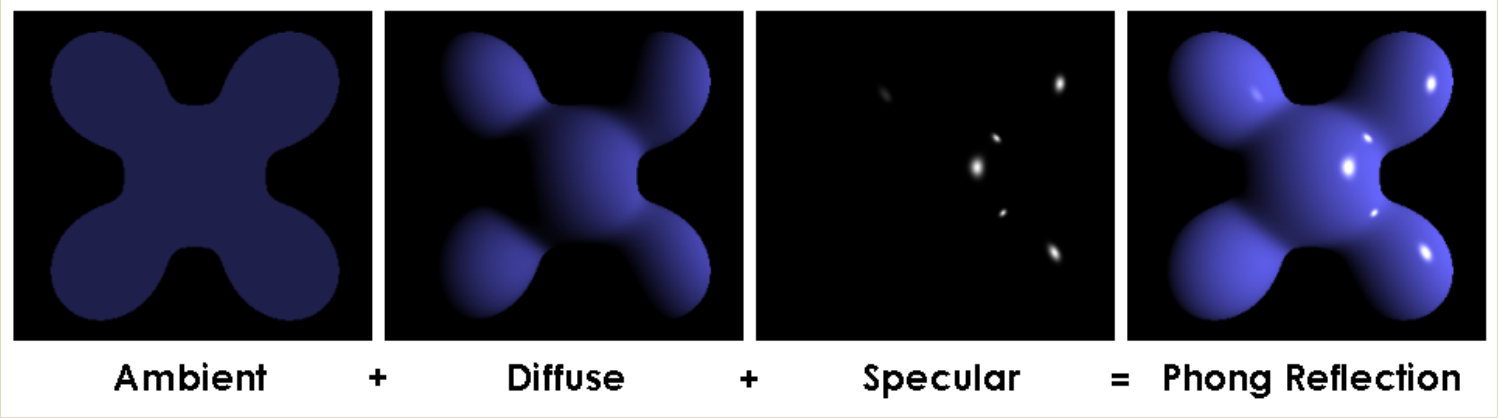
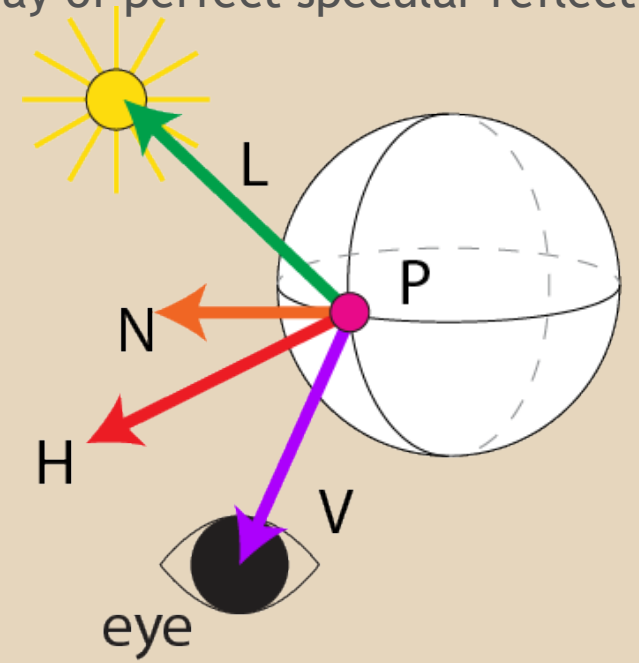


Image source: https://en.wikipedia.org/wiki/File:Phong_components_version_4.png

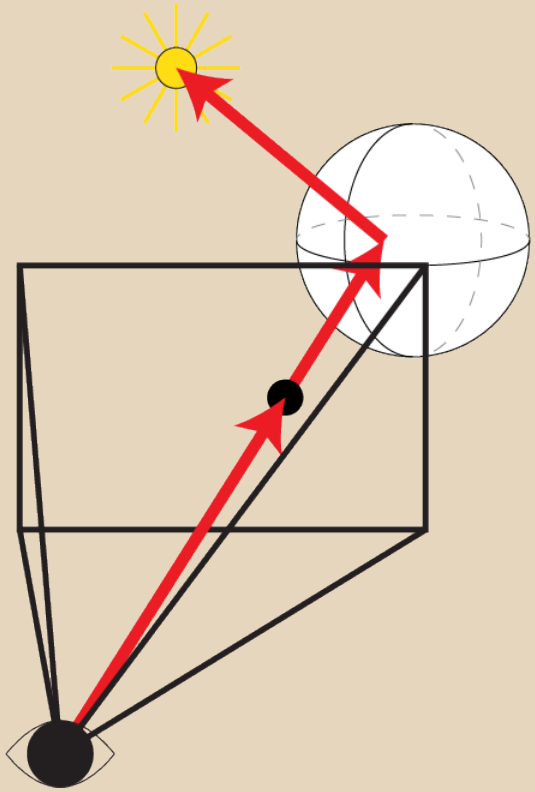
Blinn-Phong specular highlights

- S = max(pow(dot(H, N), shininess), 0) * specular_color
- H = (V + L) / 2
  - H is halfway between the view vector and the light direction
- shininess controls how diffuse the highlight is, with smaller values being more diffuse
- Mathematically, the highlight is strongest when V aligns with the ray formed by reflecting L about N (the ray of perfect specular reflection)



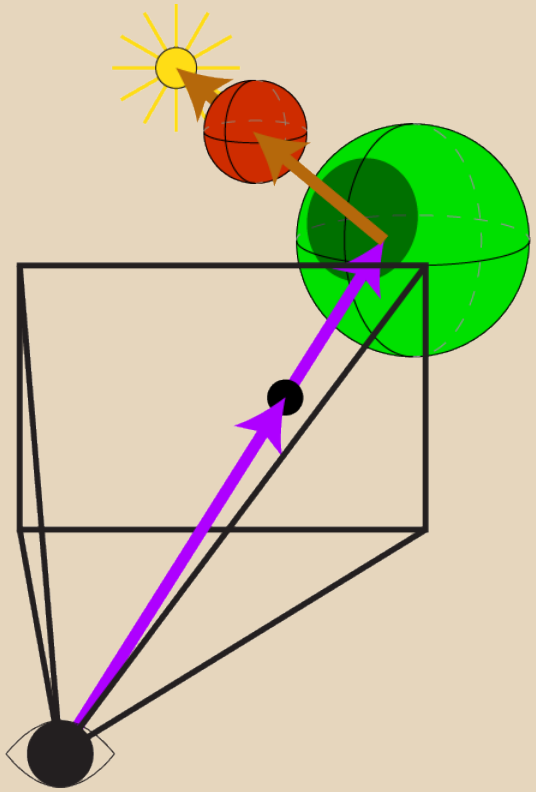
Backward Raytracing

- In the real world, photons are emitted by light sources and bounce off surfaces
- A small fraction of reflected photons reach our eyes, meaning the majority of emitted photons have no bearing on what we see
- To avoid computing the paths of all these extra photons, we trace the paths of photons in reverse
  - From each pixel of our camera screen into the scene, and back to the light source



Basic Raytracing

- The most basic raytracing algorithm involves two main ray types:
  camera rays and **light-feeler rays**
- Camera rays are emitted through each pixel of the view screen and are tested against all geometry in the scene
- When a camera ray hits a scene object, a light-feeler ray is cast from the point of intersection to each light source in the scene
  - If a light-feeler ray reaches its light source then it contributes a portion of light to the overall color of the camera ray that created it
  - If a light-feeler ray is obstructed by an object in the scene, then it contributes pure black to the overall camera ray color



Light-feeler issue: floating point error

- When computing the point of intersection with a surface, floating point error often results in the POI being slightly inside the object intersected
- When casting the light-feeler ray, it will intersect the object for which we are trying to compute the lighting
- This causes an effect commonly known as “shadow acne”

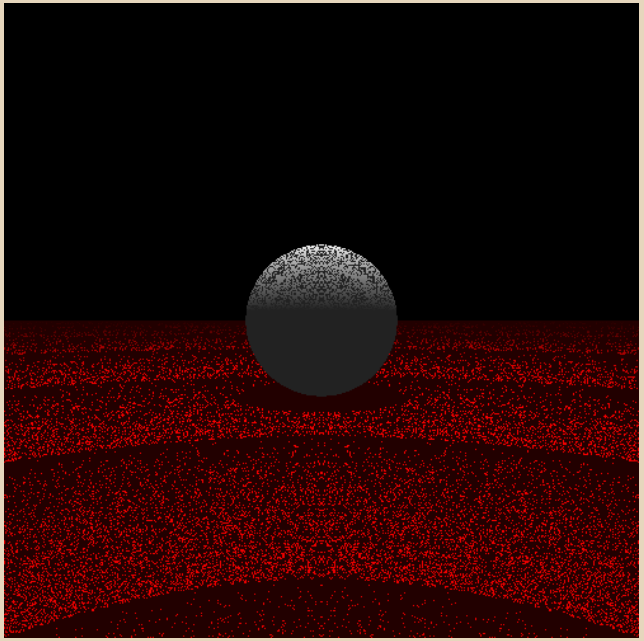


Image source: http://joedoliner.com/wp-content/uploads/2011/01/surface_acne_sphere_plane.png

Shadow acne: what to do?

- Ignore intersection with last object hit
  - Problem: prevents self-shadowing
- Use a minimum *t* value for the shadow test
  - Problem: A “good” *t* value depends on the scale of the scene and the distance the intersection is from the camera
- Use *doubles* instead of *floats* for extra precision
  - Problem: Increases memory needed to store the scene
- Offset the computed intersection by moving it along the surface normal
  - Should only move a *very* small amount (e.g. a factor above the smallest possible increment of a floating point, ~10^{-6})

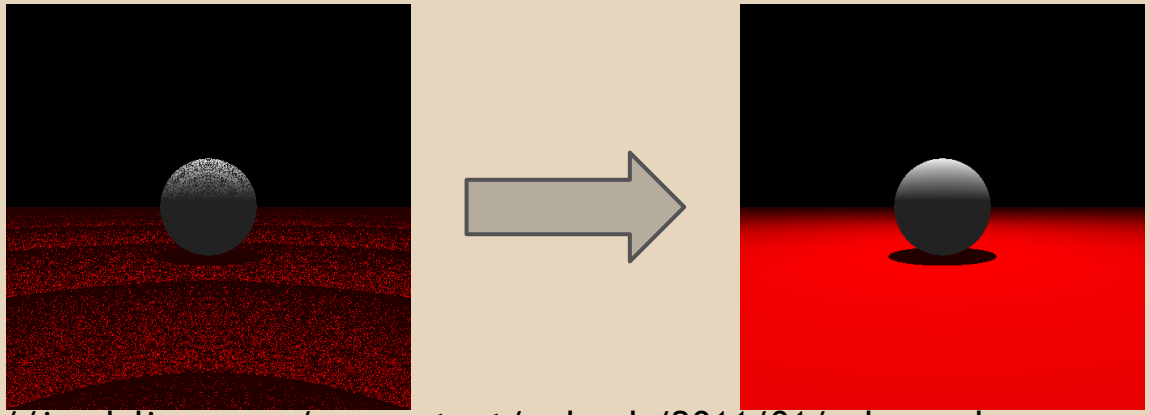


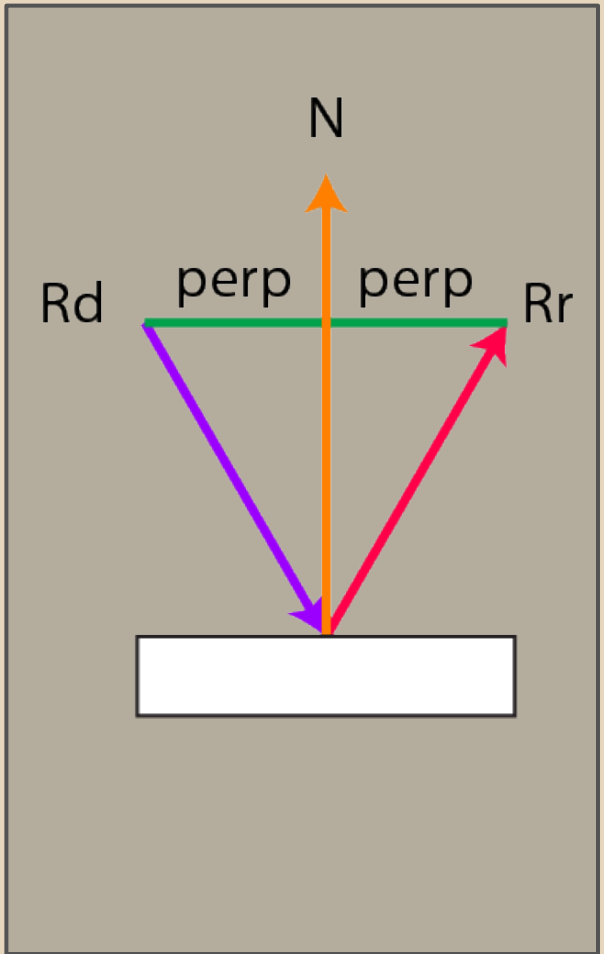
Image source: http://joedoliner.com/wp-content/uploads/2011/01/sphere-plane.png

Computing the total color at a point

- *Color = [For each light]Σ (surface_color * light_color)/[# of lights]*
  - *surface_color = ((1-Kr)*local_illumination + Kr * reflected_color)*
  - *local_illumination is different depending on whether or not the material is transparent or opaque*
    - *Transparent: material_color * light_refracted_through_object*
    - *Opaque: material_color * material.EvaluateReflectedEnergy * shadow*
  - ***reflected_color*** is the color obtained by reflecting the incident ray over the surface normal and tracing the path of that ray, computing the total color at the point in the scene it intersects. It should be multiplied by the reflective object's *material_color*.
  - *Kr* is the reflectivity coefficient, and can be any value from 0 to 1.
- Note that in order to properly compute the ***local_illumination*** of an *opaque material*, we must also perform a light feeler check as described on previous slides
  - If the light source is obscured by a *transparent object*, rather than making the shadow black, the shadow should be the *material_color* of the transparent object
  - While the above approach may not be physically accurate, within a simple raytracer it's too costly to evaluate which light rays, when refracted through the transparent object, would illuminate the opaque object

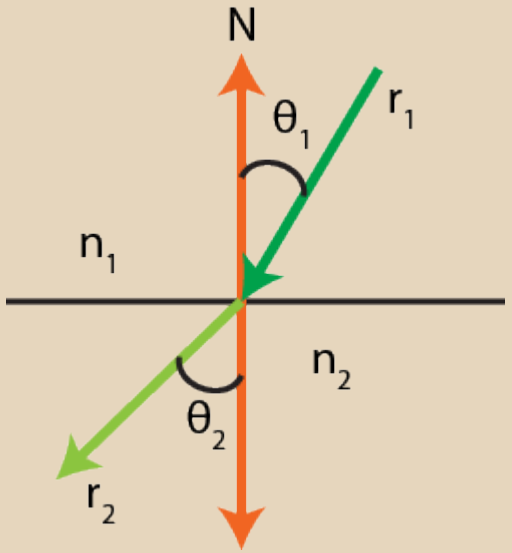
Review: Ray reflection

- Rr = Rd − 2 * N * dot(Rd, N)
- perp = N * dot(-Rd, N) + Rd
  - Perp is the line perpendicular to N that intersects the starting point of the ray
- Remember that a dot product computes the projection of one ray onto another, so to find perp we project -Rd onto N, then add Rd to the resultant vector (which in this case would be N traveling in the opposite direction) to get perp
- In practice, you can just use glm::reflect



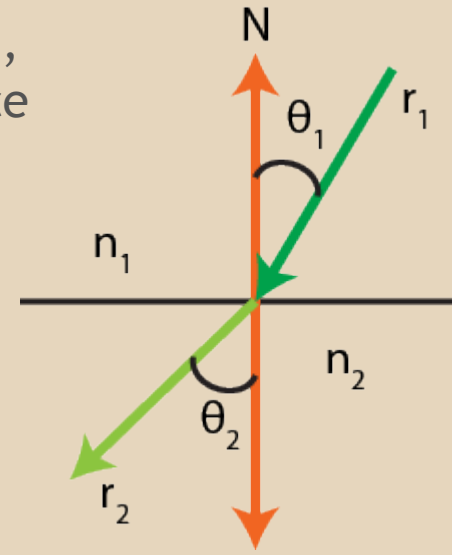
Raytracing: reflection and refraction

- Snell's law: $(n_1/n_2)sin(\theta_1)= sin(\theta_2)$
  - n_1 and n_2 are the indices of refraction of the first medium and second medium
- How do we use Snell's law to find the direction of our ray after it has been refracted?
- $A \times B = |A||B|sin(\theta)\eta$
  - η is a vector perpendicular to A and B
- Trigonometric identity: $cos^2(\theta) = 1 - sin^2(\theta)$
- Component of the refracted ray along negative normal: $dot(-N, r_2) = |N||r_2|cos(\theta_2)$
- Component of refracted ray on incident surface: $sin(\theta_2)(N \times \eta)$
- Combine all of these to get: $r_2 = n(N \times (-N \times r_1)) - N \sqrt{(1-(n^2 dot(N \times r_1, N \times r_1)))}$
  - $n = n_1/n_2$
- In practice, you can just use glm::refract



Transparent objects: critical angle

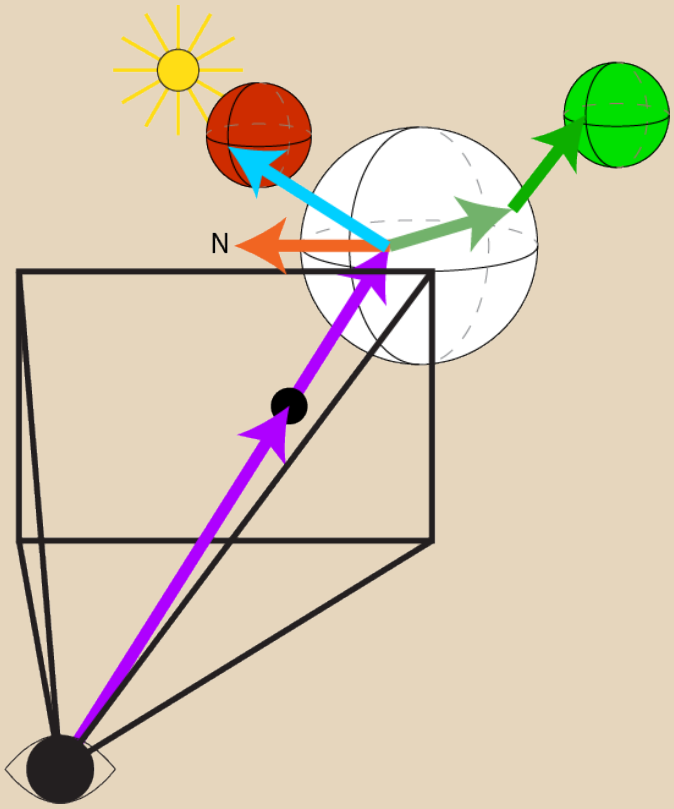
- When a ray leaves a medium and enters a less dense medium, depending on the angle between the ray direction and surface normal the ray may be ***reflected*** back into the transparent object rather than being refracted outwards (also known as total internal reflection)
- Given the following equation for computing a refracted ray and the following identities, we want to find instances in which the **radicand** is negative:
  - $r_2 = n*r_1-(n*\cos(\theta_1)+\sqrt{1-\sin^2(\theta_2)})*N$
    - $n = n_1/n_2$
  - $\sin^2(\theta_2) = (n_1/n_2)^2*\sin^2(\theta_1) = (n_1/n_2)^2(1-\cos^2(\theta_1))$



- As we see in the equation for r_2, the radicand is $1-\sin^2(\theta_2)$
- In order to determine that the radicand is less than 0, all we really need to check is that $\sin^2(\theta_2)>1$
- Using the identity above, we just need to check that $(n_1/n_2)^2(1-\text{dot}(r_1, N)^2) > 1$, assuming r_1 and N are normalized

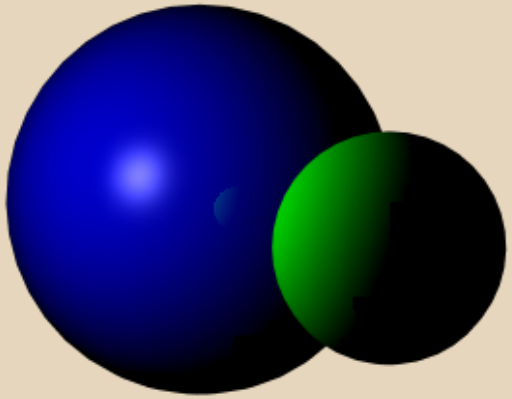
Raytracing: reflection and refraction

- In addition to computing basic diffuse lighting and occlusion shadows, we can compute the paths light takes when bouncing off reflective surfaces and moving through refractive surfaces
- Our color formula now becomes:
  K_r * *reflection_color* + K_t * *refraction_color*
  - *reflection_color* and *refraction_color* can be computed by using the color formula on the previous slide based on a **reflected ray** or a **refracted ray** rather than a camera ray
  - This means our raytracer has just become recursive!
    - e.g. if our reflected ray hits another reflective surface, we compute a color for that surface by reflecting our reflected ray again



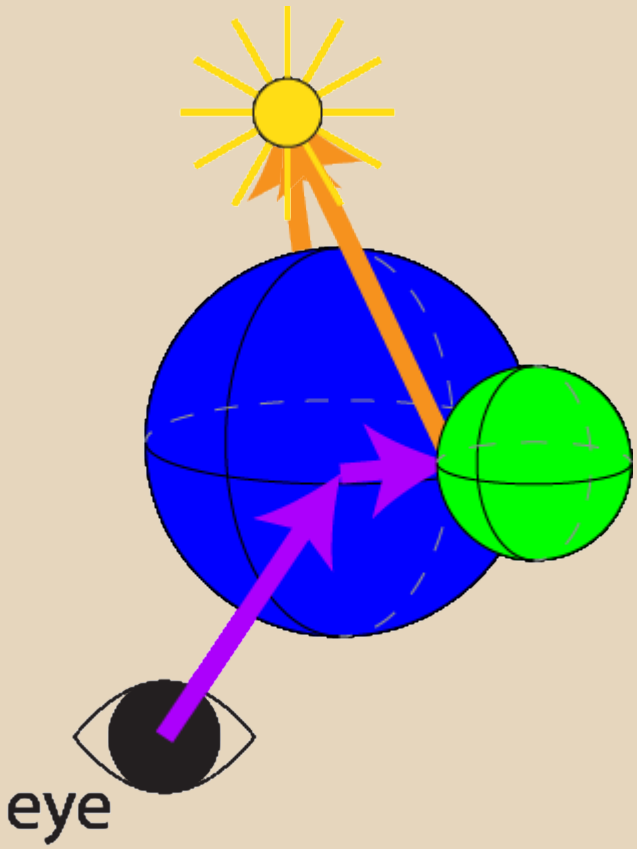
Recursive raytracing example

- In the image below we have a blue sphere with a Phong material that has a reflectivity coefficient of 0.5
  - We can see the green Lambert sphere reflected in the blue sphere



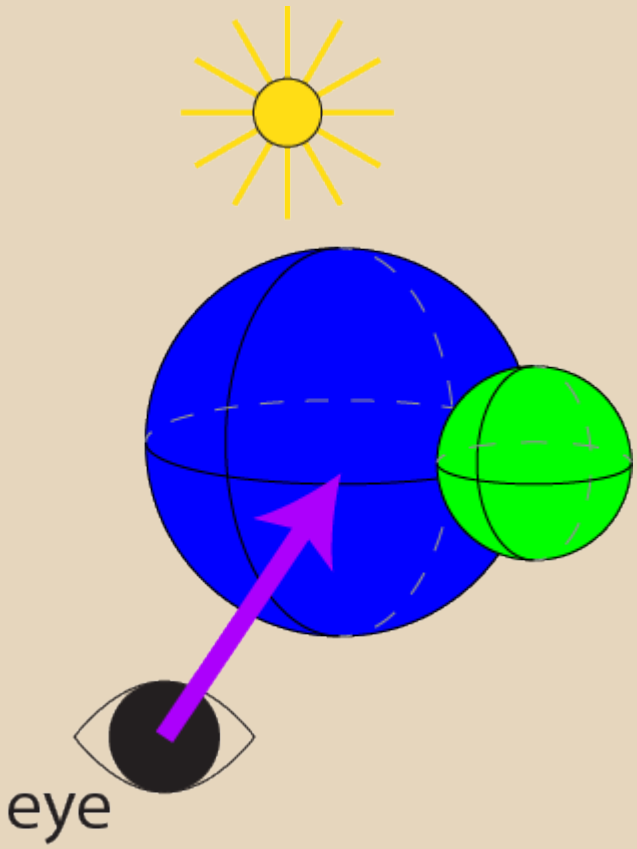
Recursive raytracing example

1. Cast a ray from the camera “through” a pixel and check where it intersects the scene



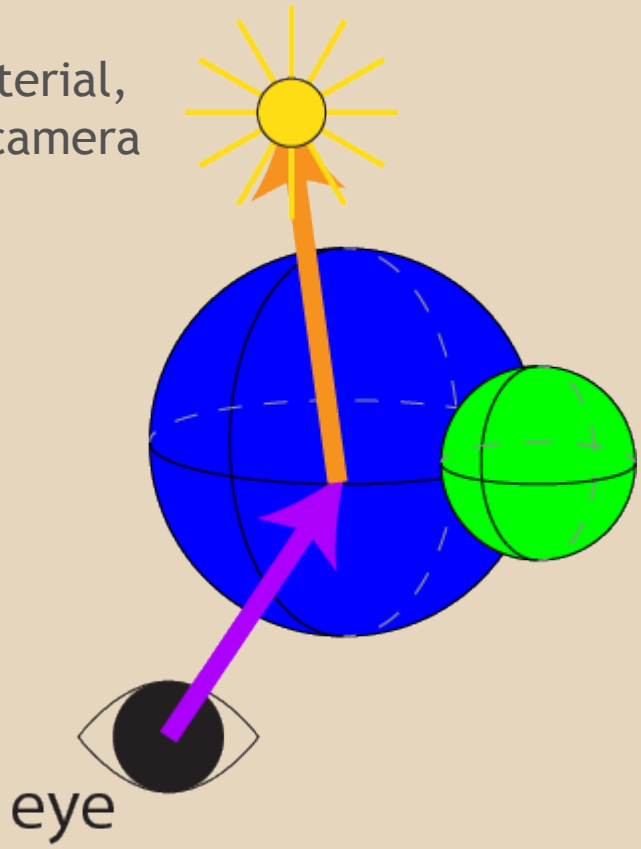
Recursive raytracing example

1. Cast a ray from the camera “through” a pixel and check where it intersects the scene
2. Check if the point of intersection can see the scene’s light source (it can!)



Recursive raytracing example

1. Cast a ray from the camera “through” a pixel and check where it intersects the scene
2. Check if the point of intersection can see the scene’s light source (it can!)
3. Compute the surface shading at the POI
   a. Since the blue sphere has a reflective material, as part of our calculation we reflect the camera ray about the surface normal and check what it intersects



Recursive raytracing example

1. Cast a ray from the camera “through” a pixel and check where it intersects the scene
2. Check if the point of intersection can see the scene’s light source (it can!)
3. Compute the surface shading at the POI
   a. Since the blue sphere has a reflective material, as part of our calculation we reflect the camera ray about the surface normal and check what it intersects
4. The reflected ray hits the green sphere, so we need to compute the shading at this new POI

Recursive raytracing example

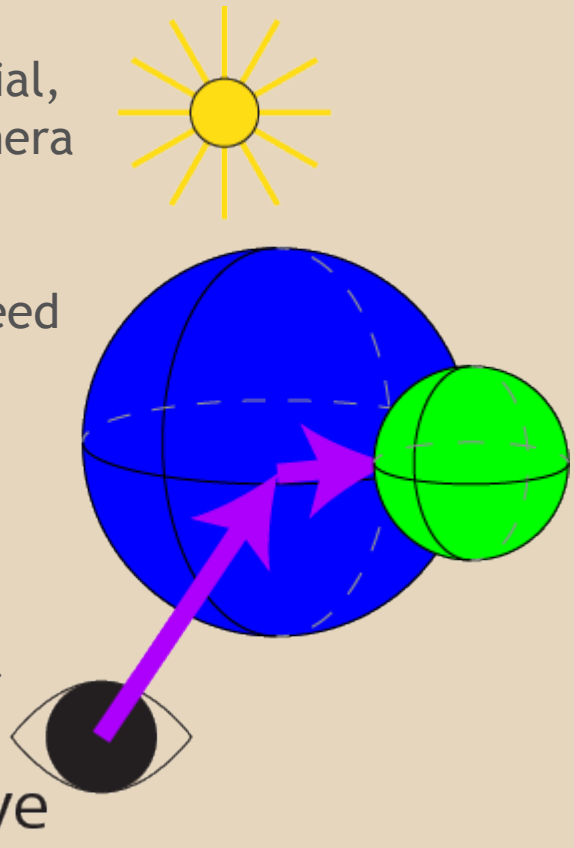
1. Cast a ray from the camera “through” a pixel and check where it intersects the scene
2. Check if the point of intersection can see the scene’s light source (it can!)
3. Compute the surface shading at the POI
   a. Since the blue sphere has a reflective material, as part of our calculation we reflect the camera ray about the surface normal and check what it intersects
4. The reflected ray hits the green sphere, so we need to compute the shading at this new POI
5. Again, perform a light-feeler check (this new POI is also lit!)

Recursive raytracing example

1. Cast a ray from the camera “through” a pixel and check where it intersects the scene
2. Check if the point of intersection can see the scene’s light source (it can!)
3. Compute the surface shading at the POI
   a. Since the blue sphere has a reflective material, as part of our calculation we reflect the camera ray about the surface normal and check what it intersects
4. The reflected ray hits the green sphere, so we need to compute the shading at this new POI
5. Again, perform a light-feeler check (this new POI is also lit!)
6. The green sphere has a Lambert shader, so we do not need to recurse to compute its shading

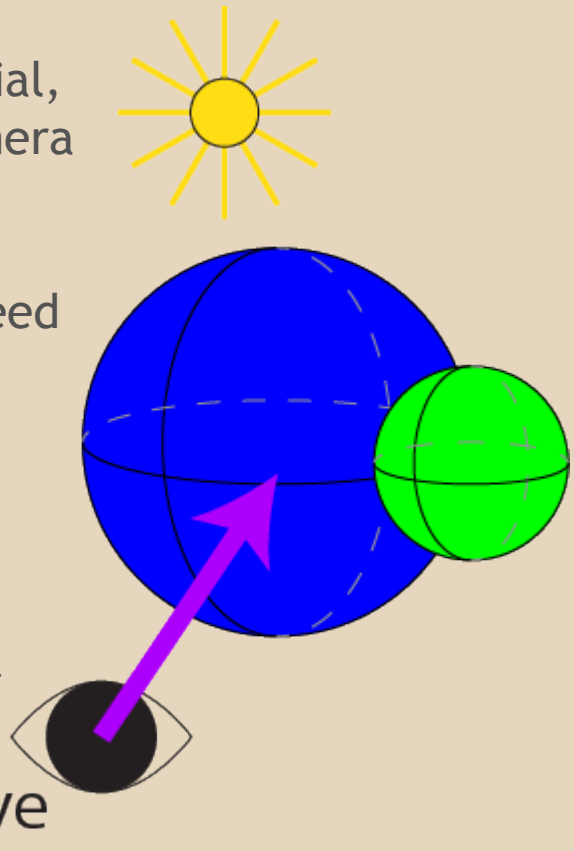
Recursive raytracing example

1. Cast a ray from the camera “through” a pixel and check where it intersects the scene
2. Check if the point of intersection can see the scene’s light source (it can!)
3. Compute the surface shading at the POI
   a. Since the blue sphere has a reflective material, as part of our calculation we reflect the camera ray about the surface normal and check what it intersects
4. The reflected ray hits the green sphere, so we need to compute the shading at this new POI
5. Again, perform a light-feeler check (this new POI is also lit!)
6. The green sphere has a Lambert shader, so we do not need to recurse to compute its shading
7. Return the green sphere’s shading to the formula that computes the blue sphere’s shading



Recursive raytracing example

1. Cast a ray from the camera “through” a pixel and check where it intersects the scene
2. Check if the point of intersection can see the scene’s light source (it can!)
3. Compute the surface shading at the POI
   a. Since the blue sphere has a reflective material, as part of our calculation we reflect the camera ray about the surface normal and check what it intersects
4. The reflected ray hits the green sphere, so we need to compute the shading at this new POI
5. Again, perform a light-feeler check (this new POI is also lit!)
6. The green sphere has a Lambert shader, so we do not need to recurse to compute its shading
7. Return the green sphere’s shading to the formula that computes the blue sphere’s shading
8. We’ve computed the blue sphere’s shading, so we send that information back to the camera



Overview of recursive raytracing

- Create a ray R for every pixel in your output image
- Find the intersection of R with your scene's geometry
- If the intersected object is reflective, compute its reflected color with another ray trace, where the ray originates at the point of intersection and has a reflected direction based on the intersection normal
- If the intersected object is refractive, compute its refracted color in a similar manner as reflected color
- If the intersected surface is not fully reflective, compute its surface color using shadow rays and (for example) the Lambert equation
- Compute the final color that the intersection produces based on the three previous computations
- Set the pixel's color to the final color

Rasterization: aliasing

- Computing the color of the scene relative to the center of each of our pixels gives us a relatively small number of samples of the lighting in our scene
- Causes an effect called aliasing, which occurs any time one discretizes continuous space
- Can try to mitigate aliasing in two ways:
  - Increase the resolution of our image so that each pixel is smaller than a face in the scene (very costly)
  - Increase the number of ray samples per pixel and average the result of multiple raycasts (still costly, but less so than making an enormous image)