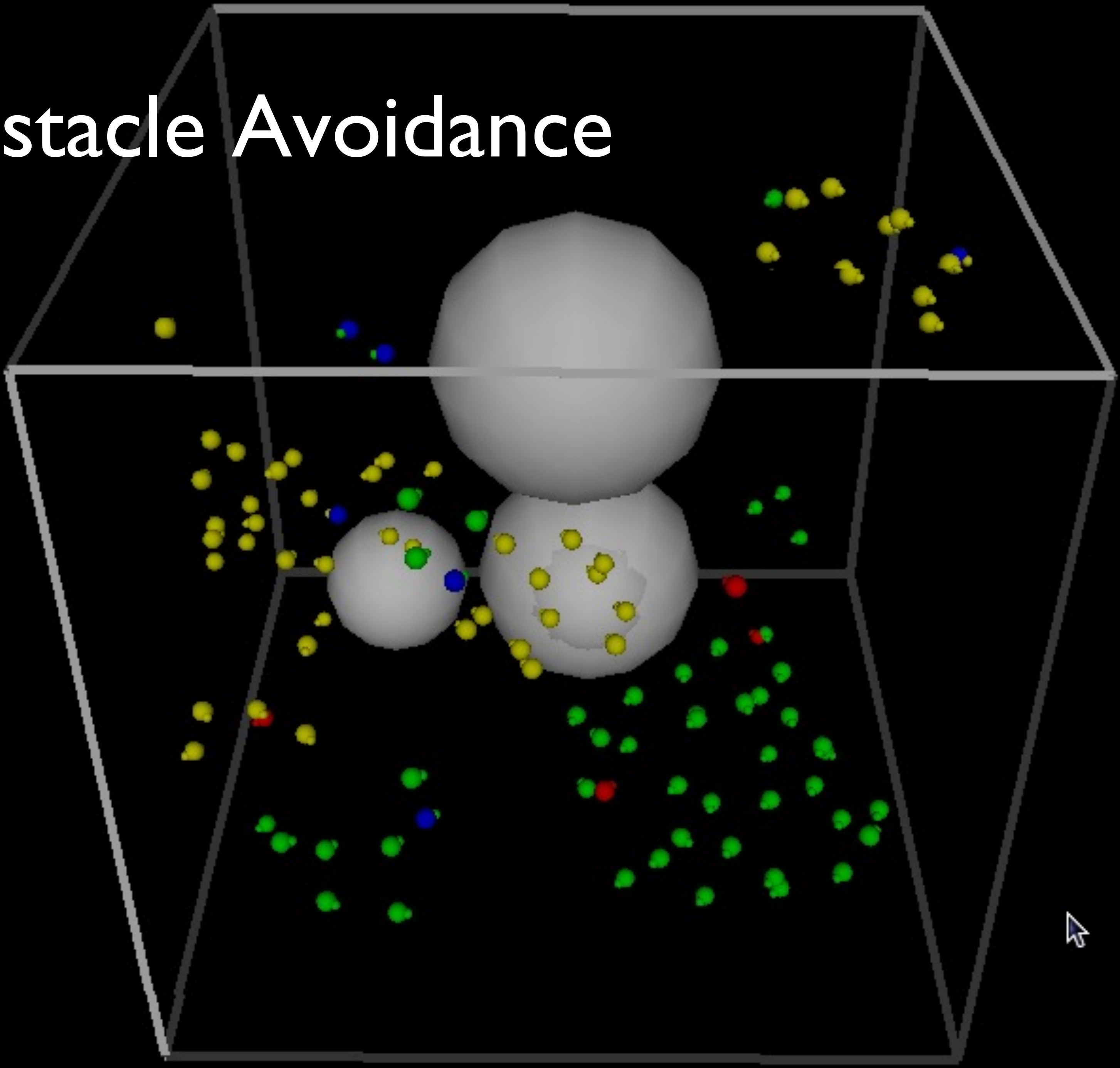
Lecture 16-17:
Boids recap and introduction to ray tracing

CGRA 354 : Computer Graphics Programming

Instructor: Alex Doronin
Cotton Level 3, Office 330
alex.doronin@vuw.ac.nz

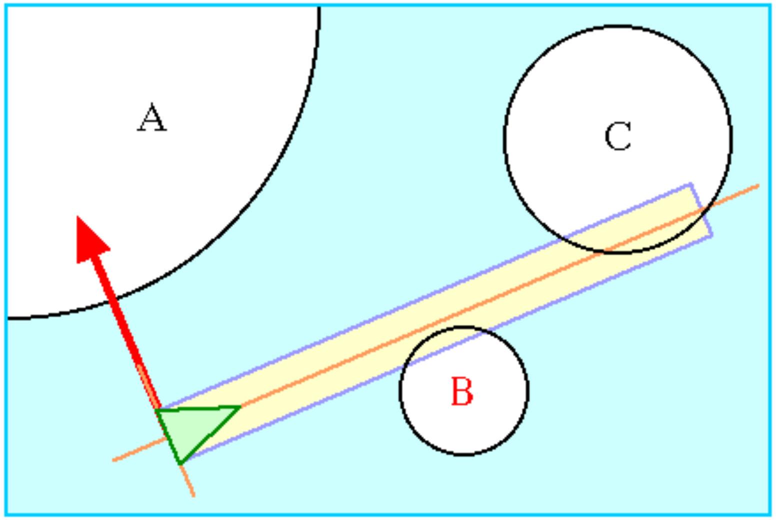
With slides from: Holly Rushmeier, Yale; Steve Marschner, Cornell; Taehyun Rhee, CMIC; Zohar Levi, Marc Gluyas, ECS;

Obstacle Avoidance



Avoiding large spheres

- Spheres are easy. Approximate a shape with spheres.
- Project a cylinder forward from the boid
- If it intersects a sphere steer to avoid



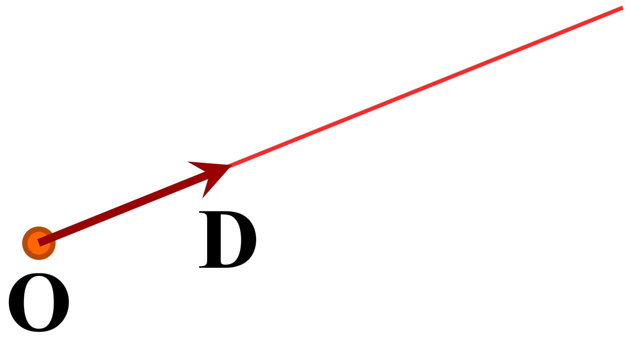
Craig Reynolds has written extensively about various strategies for obstacle avoidance in this informal paper: https://www.red3d.com/cwr/nobump/nobump.html

The diagram is from:
http://www.red3d.com/cwr/steer/gdc99/

Intersect a line and a sphere?

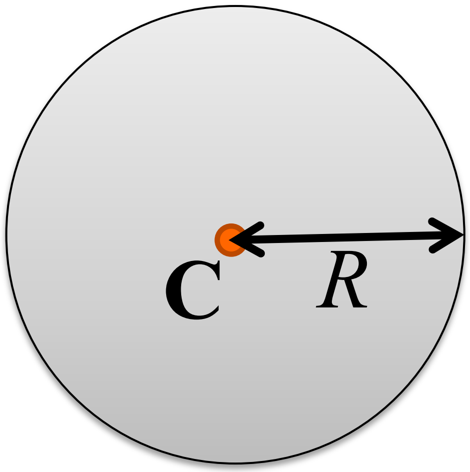
- Equation of a line

$$\mathbf{P}(t) = \mathbf{O} + t\mathbf{D}$$



- Equation of a sphere

$$|\mathbf{P} - \mathbf{C}| = R$$

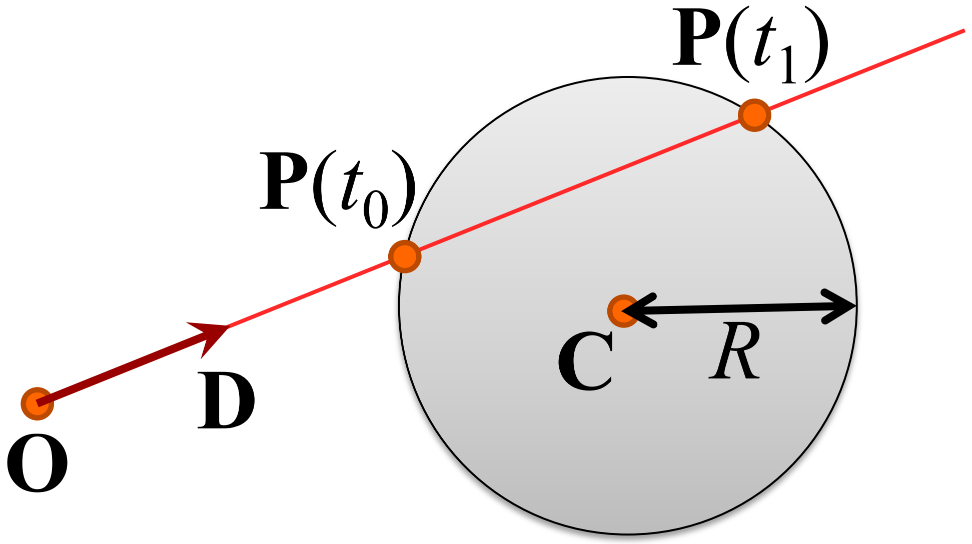


Solve that quadratic equation

$$t^2(\mathbf{D}\bullet\mathbf{D}) + 2t([\mathbf{O}-\mathbf{C}]\bullet\mathbf{D}) + ([\mathbf{O}-\mathbf{C}]\bullet[\mathbf{O}-\mathbf{C}]) = R^2$$

$$t^2(\mathbf{D}\bullet\mathbf{D}) + t(2[\mathbf{O}-\mathbf{C}]\bullet\mathbf{D}) + ([\mathbf{O}-\mathbf{C}]\bullet[\mathbf{O}-\mathbf{C}] - R^2) = 0$$

$$at^2 + bt + c = 0$$



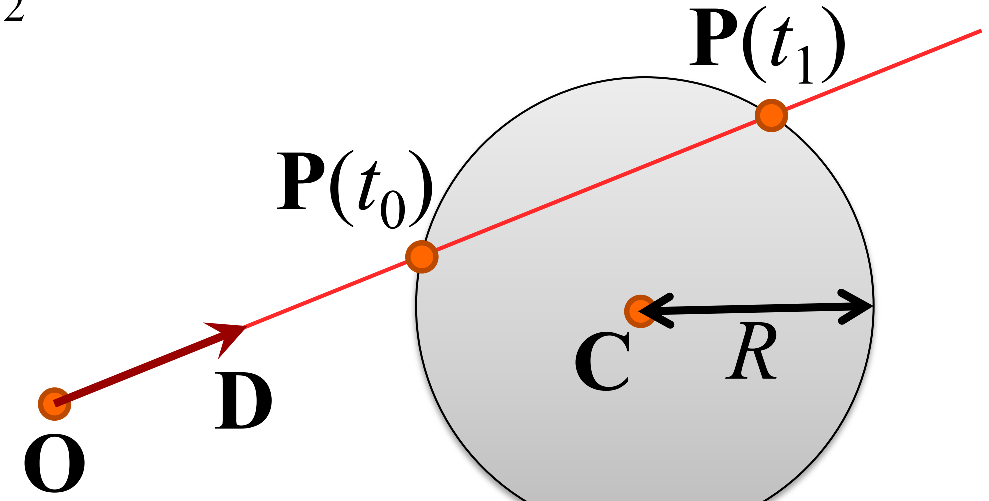
Solve that quadratic equation

$$at^2 + bt + c = 0$$

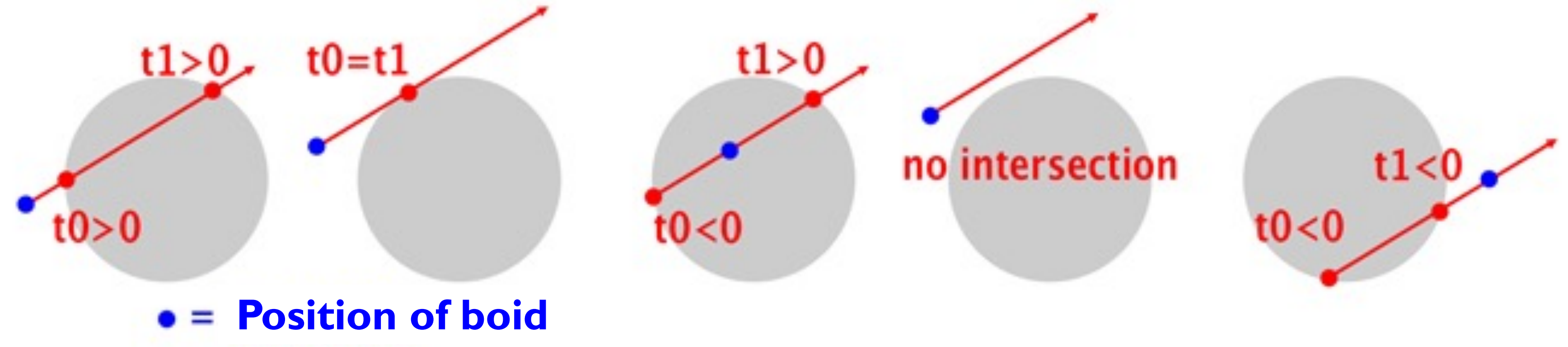
$$a = \mathbf{D} \bullet \mathbf{D}$$

$$b = 2[\mathbf{O} - \mathbf{C}] \bullet \mathbf{D}$$

$$c = [\mathbf{O} - \mathbf{C}] \bullet [\mathbf{O} - \mathbf{C}] - R^2$$



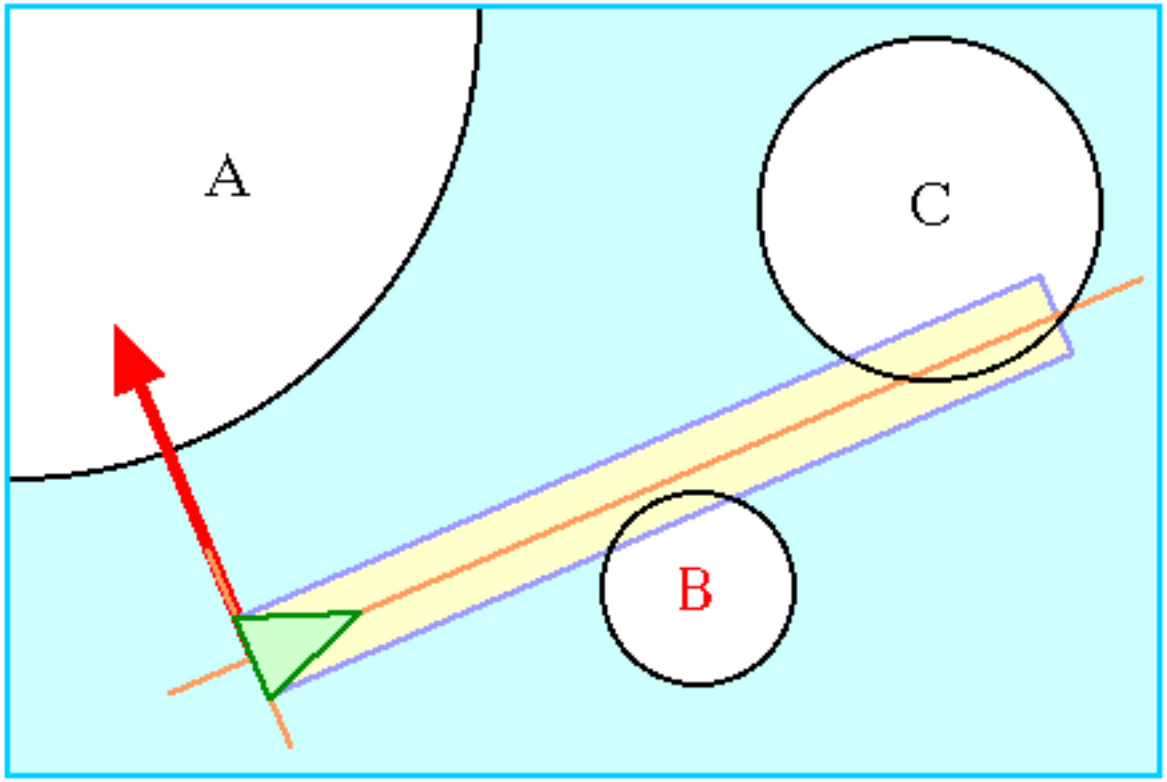
But wait, there's more to consider…



- Boid is going to hit sphere
- Boid is going to just skim sphere
- Boid is inside sphere
- Boid is going to miss sphere
- Sphere is behind boid

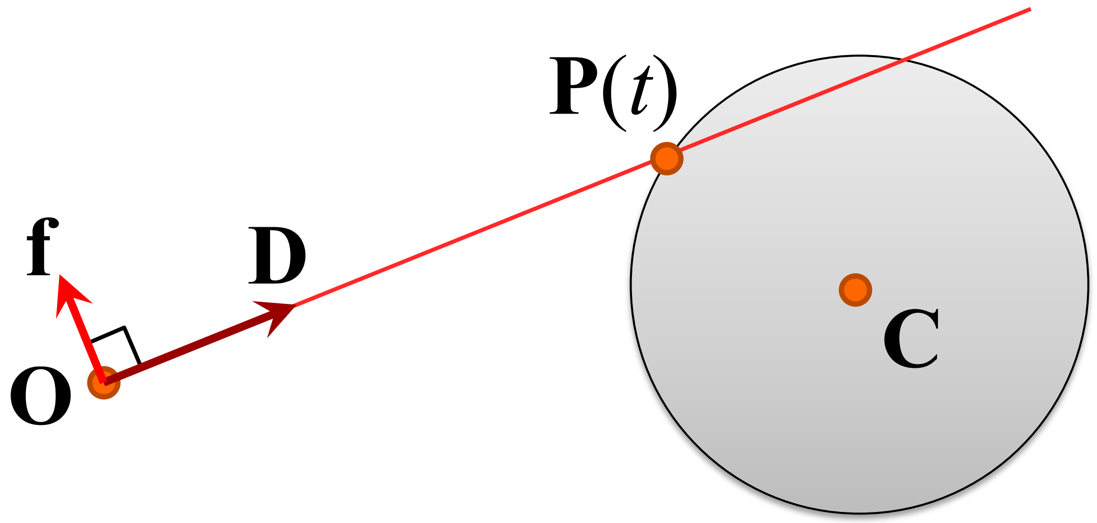
How do we calculate that force?

- The force must push us away from the sphere that we are going to hit



What do we know?

- We know **O**, **D**, **C**, t, **P**(t)
- We want **f**



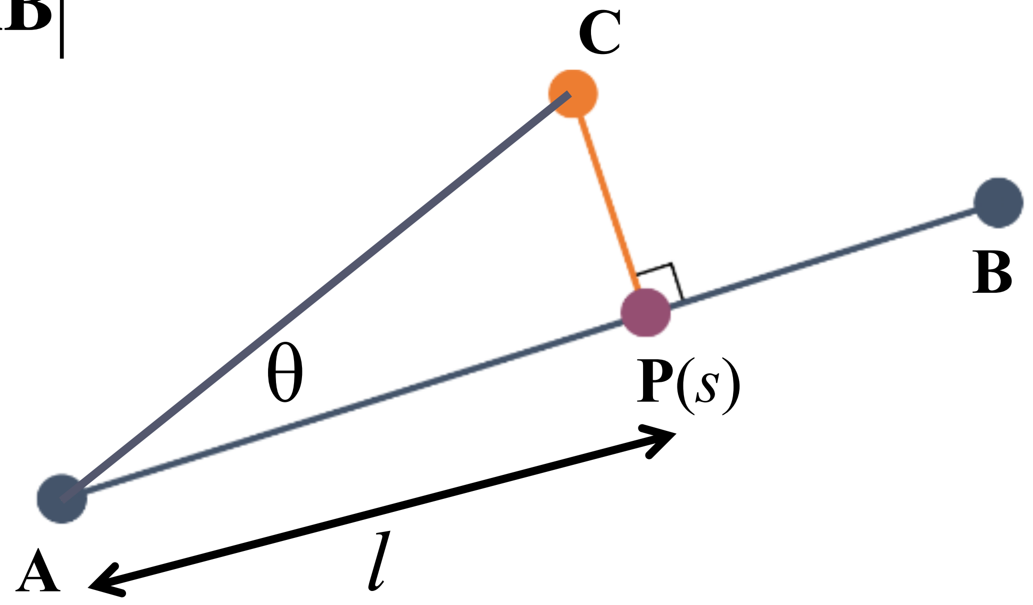
Closest point on a line

- $\mathbf{P}(s)$ is the point on line $\mathbf{AB}$ that is closest to point $\mathbf{C}$

$$l = |\mathbf{AC}|\cos\theta = \frac{\mathbf{AC} \bullet \mathbf{AB}}{|\mathbf{AB}|}$$

$$s = \frac{l}{|\mathbf{AB}|}$$

$$\mathbf{P}(s) = (1 - s)\mathbf{A} + s\mathbf{B}$$

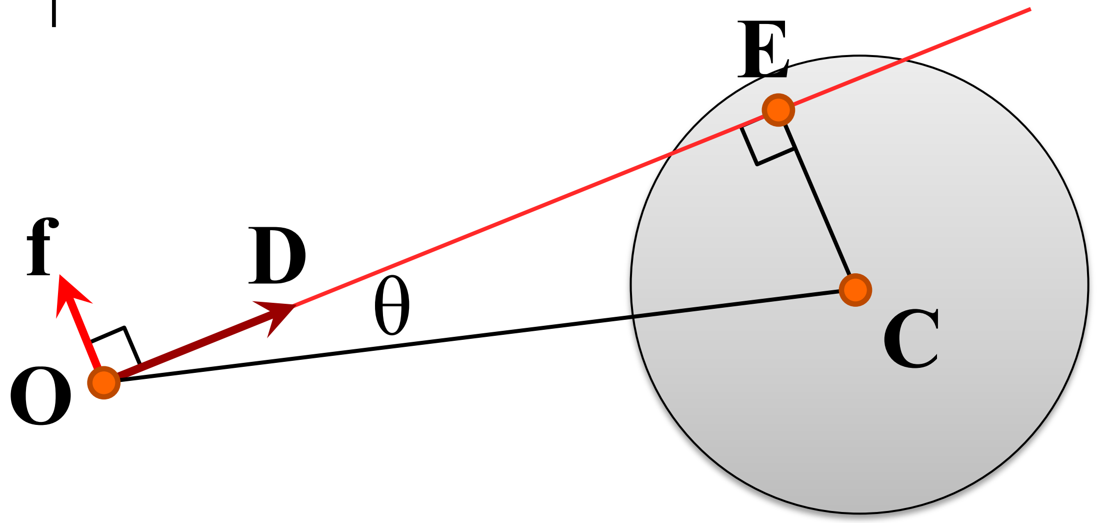


Getting a vector at right angles to **D**

$$|\mathbf{OE}| = |\mathbf{OC}|\cos\theta = \frac{\mathbf{OC} \bullet \mathbf{D}}{|\mathbf{D}|}$$

$$\mathbf{E} = \mathbf{O} + |\mathbf{OE}|\frac{\mathbf{D}}{|\mathbf{D}|}$$

$$\mathbf{f} = \frac{\mathbf{CE}}{|\mathbf{CE}|}$$



- Our force **f** is a vector that points in the same direction as **CE**

Ray Tracing I

fundamentals of ray tracing

ray/sphere intersection

calculating normal vectors

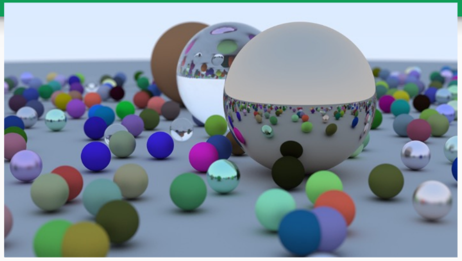
how illumination works

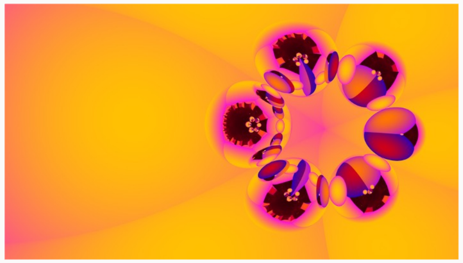
What is Raytracing?

Raytracing is the other main rendering algorithm in Computer Graphics. Compared to rasterization, raytracing easily supports much more complex effects such as reflection, refraction, and ambient lighting, at a significant cost to performance.

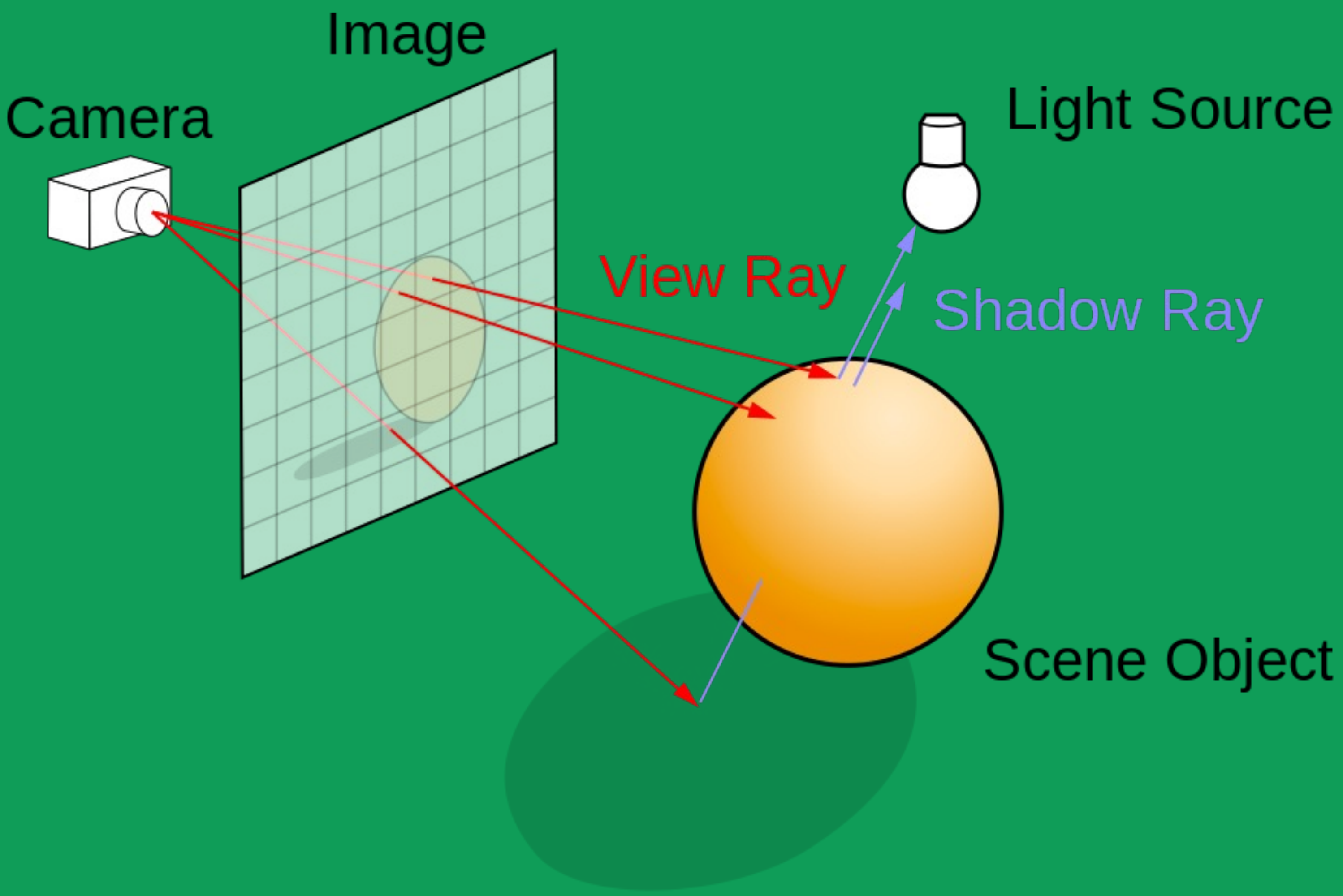
Raytracing is the basis of physically-based rendering and is our best tool for rendering photorealistic images. It's also widely used in creative coding and generative artwork due to its flexibility.

Historically, raytracing has only been used for films and other offline media due to its computation time. But it has recently become infamous in gaming with the introduction of hardware-accelerated raytracing into modern GPUs (RTX, RDNA2), which enable real-time performance for limited raytracing. It's expected that raytracing will play a larger and larger role in real-time graphics in years to come.

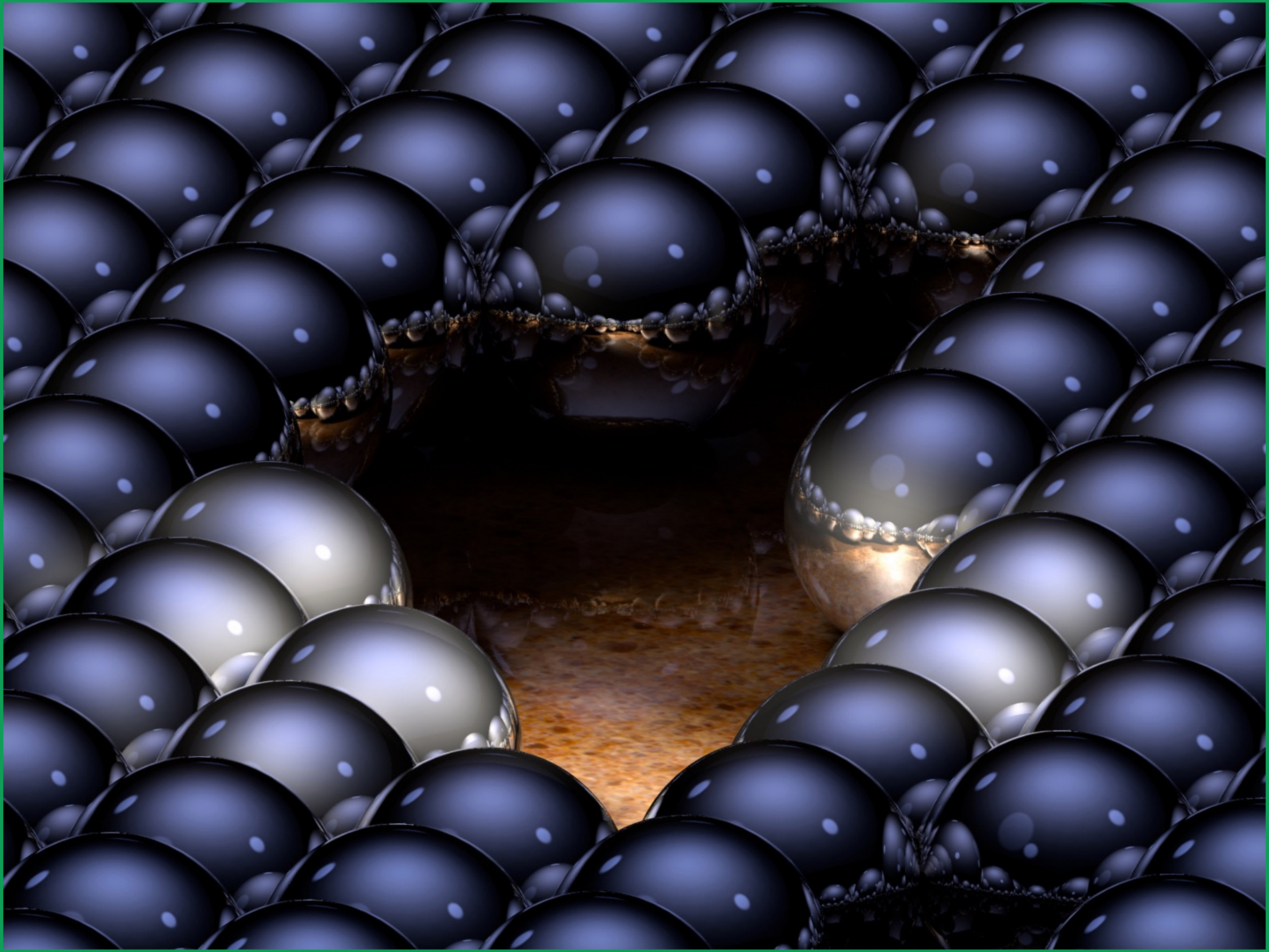




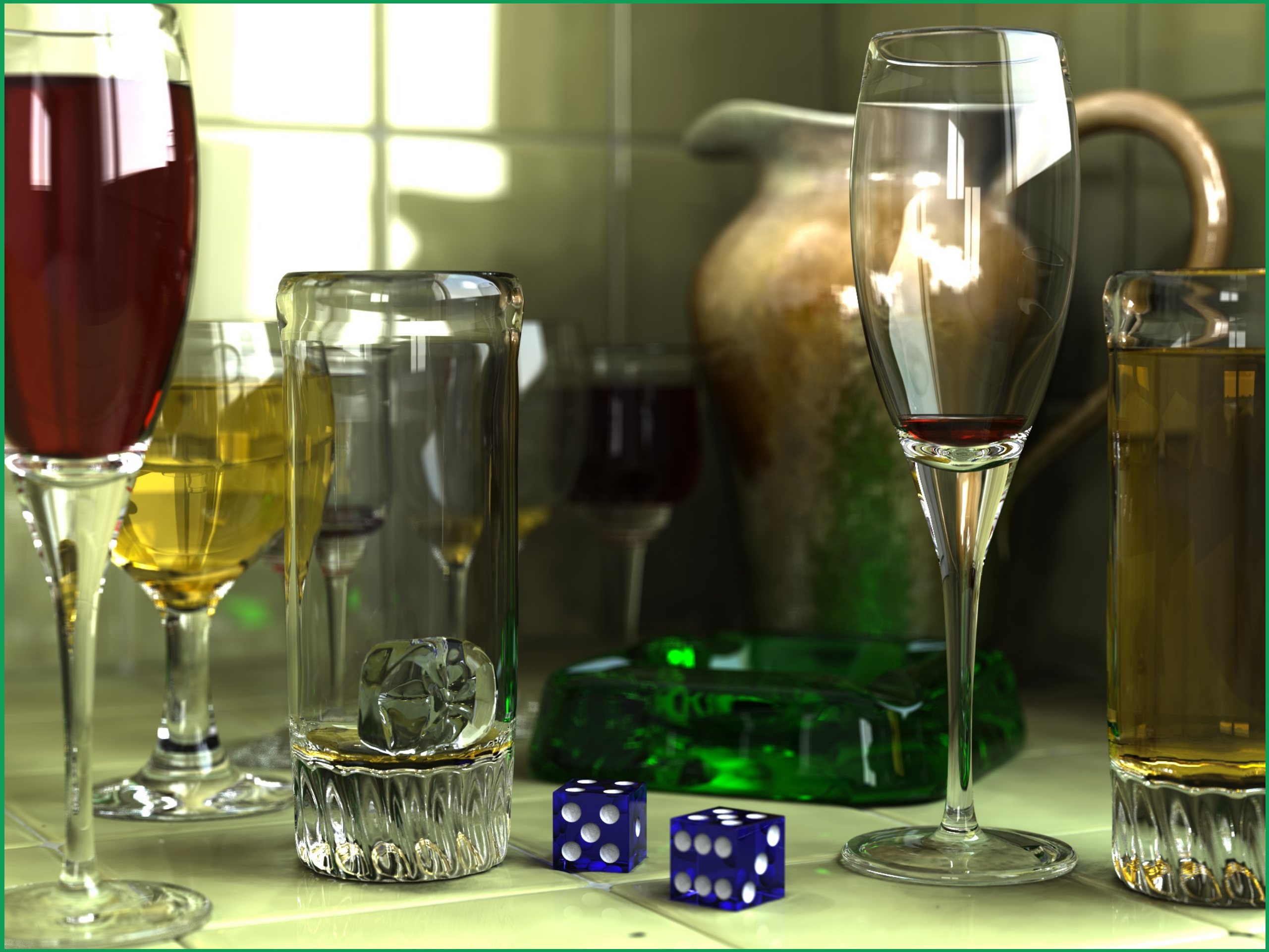
Ray tracing



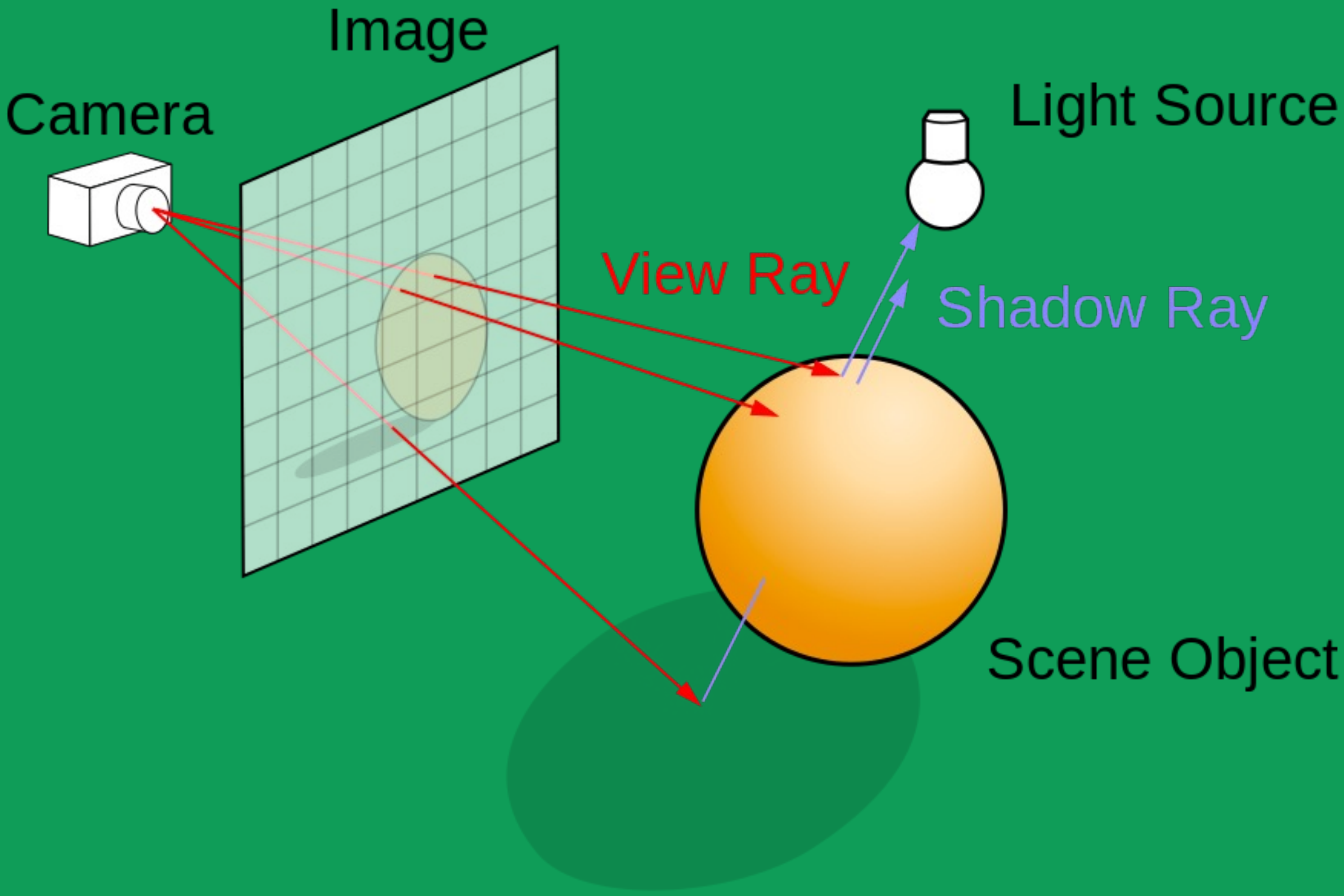
Multiple reflections



- Effects: reflection, refraction, transparency, depth-of-field blur, and bumpy surfaces
- Ray tracing has become more dominant in recent years



More details on the basic idea



- Specification of a camera (position, orientation)
- Specification of an image plane (distance from camera, aspect ratio, resolution in pixels)
- Specification of a scene comprising one or more objects and one or more light sources
- Ability to cast view rays from the camera position through the centre of each pixel in the image
- Ability to intersect a ray with all objects in the scene
- Ability to find the normal at the intersection point
- Ability to cast shadow rays from the intersection point to every light source in the scene
- Ability to calculate the colour at the intersection point, given all of the details about the object and its material

The basic algorithm

```
for each pixel (i,j) do {
   view_ray = CreateRayForPixel( i, j ) ;
   tNear = INFINITY
   objNear = NULL
   for each object in objects do {
      t = FindIntersection( view_ray, object )
      if( t < tnear ) {
         tNear = t
         objNear = object
      }
   }
   if( objNear ≠ NULL) {
      pixel_colour(i,j) = FindColour( tNear, objNear ) ;
   }
}
```

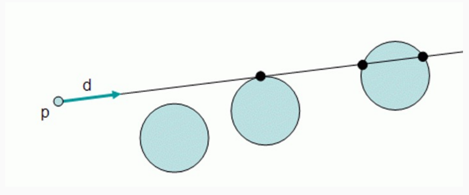
Rays

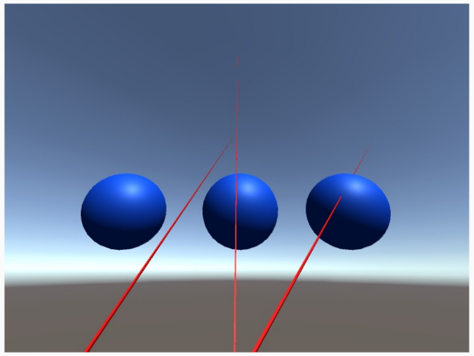
A ray is a line through space which has a starting point, but no end point. A ray has an origin and a direction.

We can generate every point on a ray using the parametric formula
p + td, for every value of t >= 0. This equation will help us find algorithms to find intersections of rays with solid objects.

We usually think of rays as the path taken by photon packets emitted by a light source.

Radiance is defined as colour and brightness of the light energy travelling along a ray. Radiance is the primary quantity that we work with in physically-based rendering.



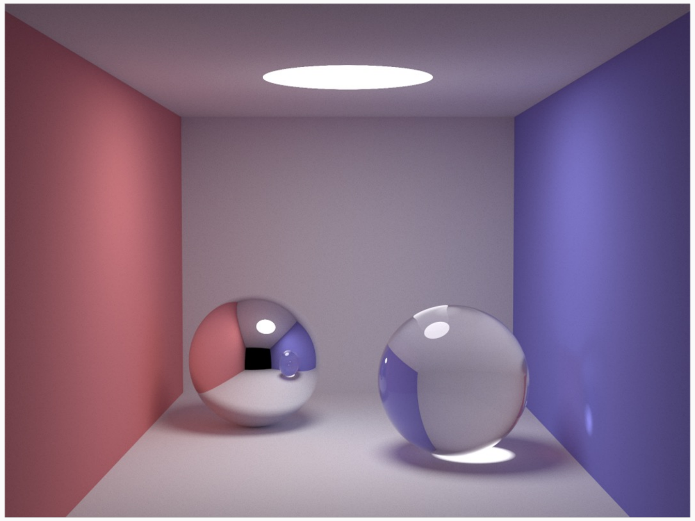


Global Illumination

In real life, light that’s reflected off objects does all go directly to our eyes. For the majority of materials, light is scattered in random directions, often multiple times between objects in the world.

This creates much of the visual complexity of a scene: the corners of the room are slightly darker then the walls, and nearby objects of different colours have their colours mix slightly. No surface in the scene is completely black, even if there is no direct light hitting it.

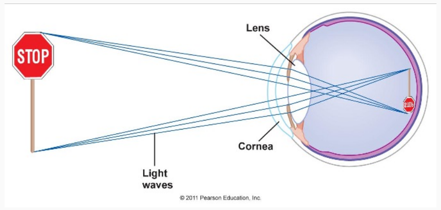
Accurately accounting for all light bouncing around in a scene is called global illumination, and is the main benefit of raytracing. Phong shading is not capable of global illumination; its ambient term is intended to approximate indirect lighting.

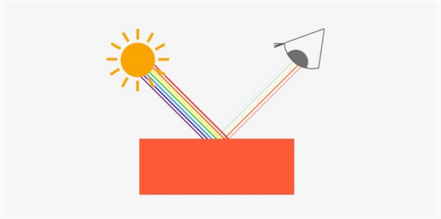


Raytracing Intuition

Raytracing is an intuitive and simple algorithm based on a small set of common-sense observations about the world.

- The image you see is based on the pattern of light hitting your retina.
  - The same applies for a camera - we'll often use camera terminology such as lens and film.
- Light travels in straight lines (in >99.9% of observable cases), and can be reflected or refracted by objects.
- When you see an opaque object, you are seeing light which has been reflected off its surface and towards your eyes.
  - Shadows and dark-coloured objects are the reduction or absence of light hitting your retina at that particular position.
- The colour of reflected light depends on both the colour of the material and the colour of the light before it was reflected.



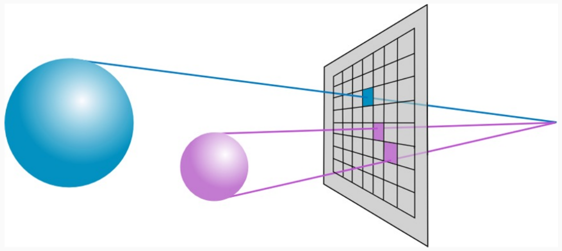


Key Idea

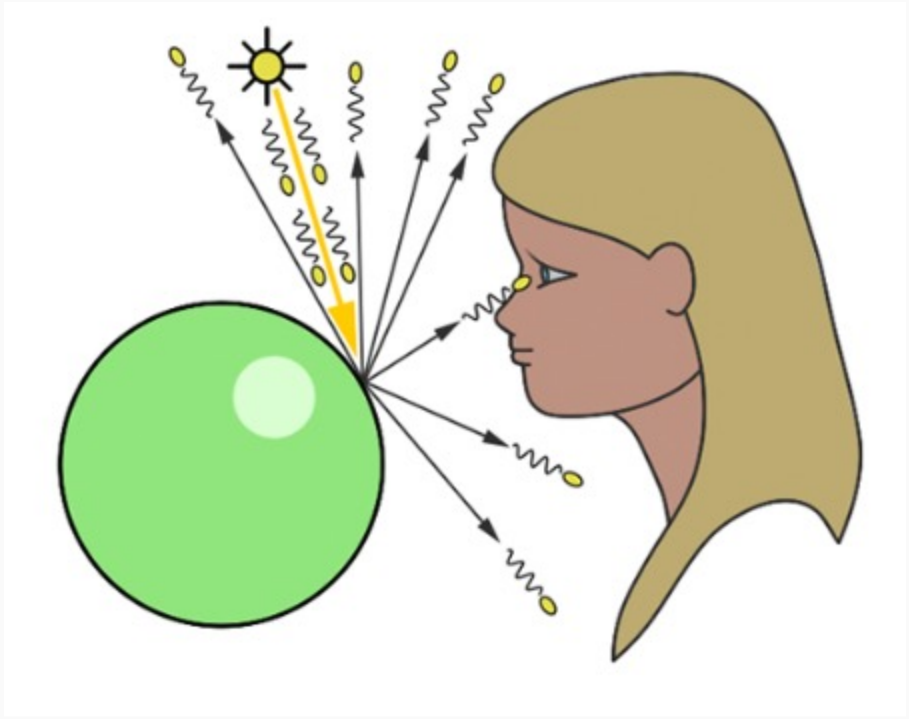
To photorealistically recreate an image, all we need to do is accurately figure out how much (and what colour) light is arriving at each point on our retina.

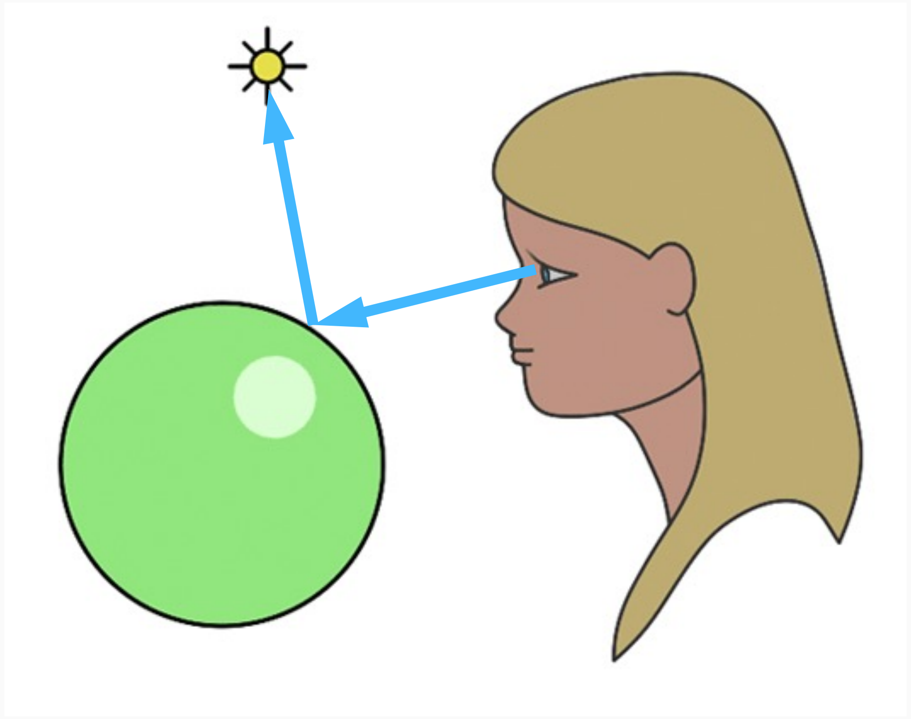
The naive approach would be to simulate all of the light rays emitted by every light source in the scene, and count how many enter your eye. The vast majority do not, so this is inefficient.

Instead, we'll shoot a ray back out away from the eye. This is counter intuitive, but it has the huge advantage that we only calculate light which eventually ends up in the final image. We'll do this for every pixel of our screen, to build a complete image.



Forward or backward ray tracing?





Forward ray tracing:
trace rays from lights

Backward ray tracing:
trace rays from eye

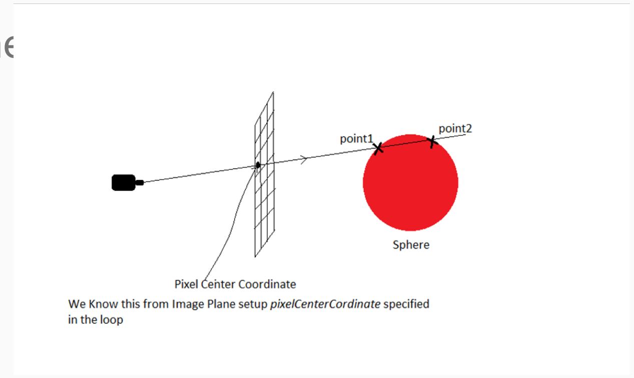
Firing the Primary Ray

The first step of raytracing is to “shoot” a primary ray out of the pixel, and find where it intersects the scene geometry. This is significant - it tells us the specific point on the specific object which is responsible for the colour of this pixel.

We can use our knowledge about the object this to calculate the light which reaches the retina.

If the object is a light source (eg: a light bulb), then the light received by this pixel is simply the light emitted by the bulb.

If the object is opaque, we need to do a bit more work...

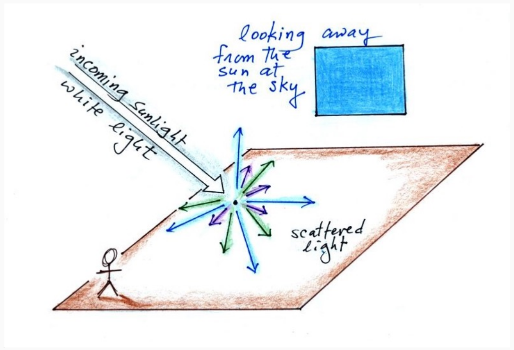

point1
point2
Sphere
Pixel Center Coordinate
We Know this from Image Plane setup *pixelCenterCordinate* specified in the loop

Surface Reflectance

Opaque objects do not emit light themselves - they are visible because they reflect (and usually scatter) light from other sources.

We can calculate the light going back along the primary ray (which will determine the pixel colour) if we know two things:

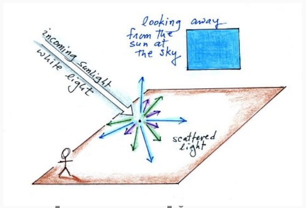
- The locations of all light sources in the scene
- **How light, coming from a specific direction, is reflected by the surface back along the primary ray.** This depends entirely on the material.



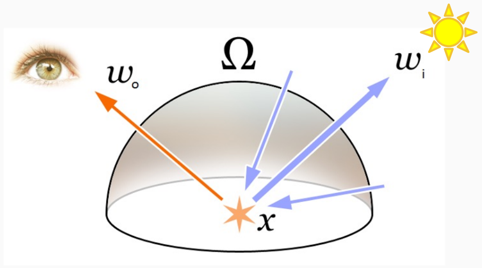
To calculate the pixel colour value we iterate through every light source. We then calculate how much of that light is being reflected by the object back along the primary ray. Finally, we take the sum of these individual contributions to give us the final result.

Example: Single Light Source

For the simplest example we use a single light source with a diffuse (aka Lambertian) material. A diffuse material scatters incoming light equally in all directions, according to the cosine law.



$$L_o = (\mathbf{n} \cdot \omega_i) L_i$$



L refers to radiance values - Li for incident light (ie: from the light source) and Lo for outgoing light (ie: back along the primary ray). In other words, they are the quantities of light travelling along the wi and wo rays.

Example: Multiple Light Sources

With multiple light sources and a single diffuse material, it’s much the same. We iterate over every light source, and take the sum of their cosine laws.

$$L_o = (\mathbf{n} \cdot \omega_i) L_i \longrightarrow L_o = \sum_{j=1}^{N} (\mathbf{n} \cdot \omega_i^{(j)}) L_i^{(j)}$$

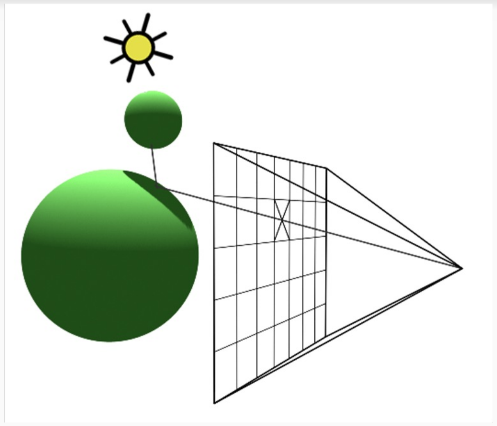
This accounts for the individual colours and brightnesses of each light, as well as their relative angles to the surfaces. Each wi(j) and Li(j) refers to the vector to, and radiance emitted by the j’th light source.

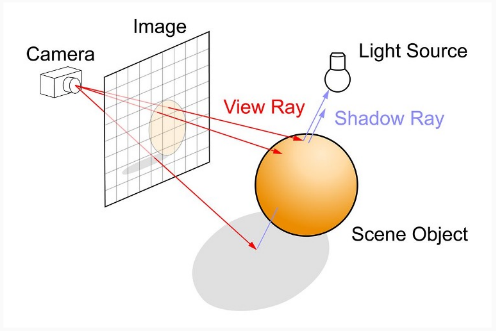
Secondary Rays (aka Shadow Rays)

So far, we've assumed that all light sources will affect the final colour of the pixel. But we need to account for occlusion - where the light source is obstructed by another object in the scene. This is what creates shadows in real life.

We need to ensure that there is an unobstructed path from each light source to the point on the surface that was hit by the primary ray. If there isn't, we discard that light source's contribution.

We can test this by shooting a secondary ray from the point of intersection towards the light. If the ray intersects another object before the light source, then it is occluded.



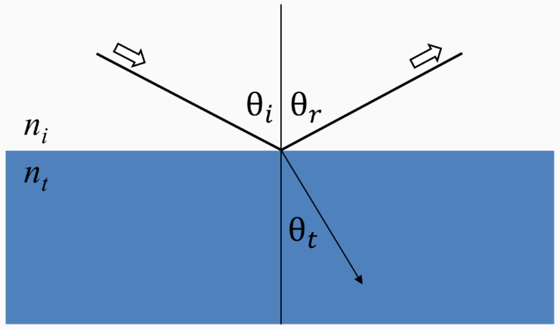


Reflection and Refraction

The first area where raytracing really shines is with mirror-like and glass-like materials.

To implement a mirror, we can simply shoot another ray from the point that the primary ray hit, in the direction that it would be reflected.

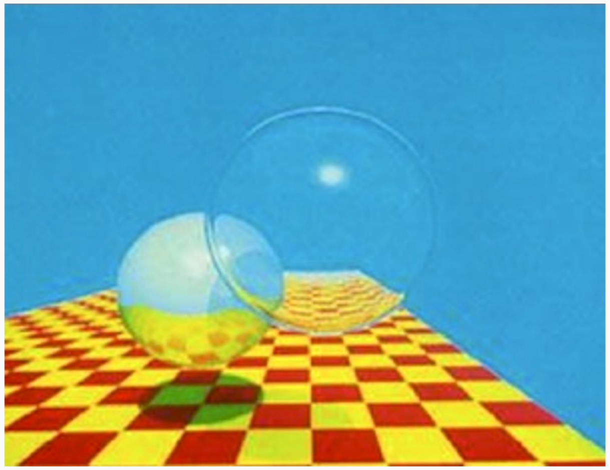
To implement refractions, we shoot a ray *inside* the object at the angle it would refracted at. We also need to take into account the small amount of light which is reflected off the surface, so we combine this with a reflected ray too.



Whitted Raytracer

So far, this covers everything you need to implement a “Whitted” raytracer. This is the model developed for the first published paper on raytracing by Turner Whitted in 1979.

At the time, this was a groundbreaking result, but with modern computation power we can easily extend this model to produce much more interesting images.



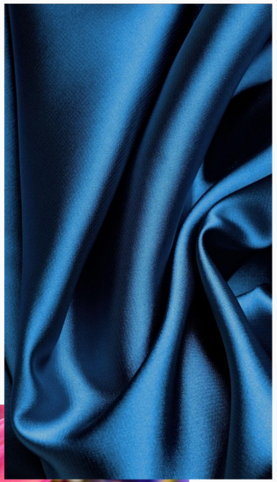
Advanced Materials

So far we've only looked at the most basic material. But this approach works for *any* material we can find a mathematical model for.

In the previous slides, we used a Lambertian model, which is good for simple materials like paper and wall paint.

Extending the Lambertian model, we could also use the Phong model, which adds specular highlights. This is great for materials like plastic and rubber.

We can extend this technique with more advanced material models to render *any* object. We just have to come up with an answer to the important question: *how is light reflected off this surface?*



Bidirectional Reflectance Distribution Function (BRDF)

The BRDF describes exactly how light is reflected by a material. Each material has its own BRDF, and developing new ones is an active area of research. Given two direction vectors (incident light direction, and viewer direction), the BRDF tells us what proportion of the radiance along the incident ray is reflected along the outgoing ray.

$$L_o = f(\omega_o, \omega_i) L_i$$

$$L_o = \sum_{j=1}^{N} f(\omega_o, \omega_i^{(j)}) L_i^{(j)}$$

The Lambertian and Phong illumination models are both examples of BRDFs, and there are many, many more, which will be covered in further CGRA courses.

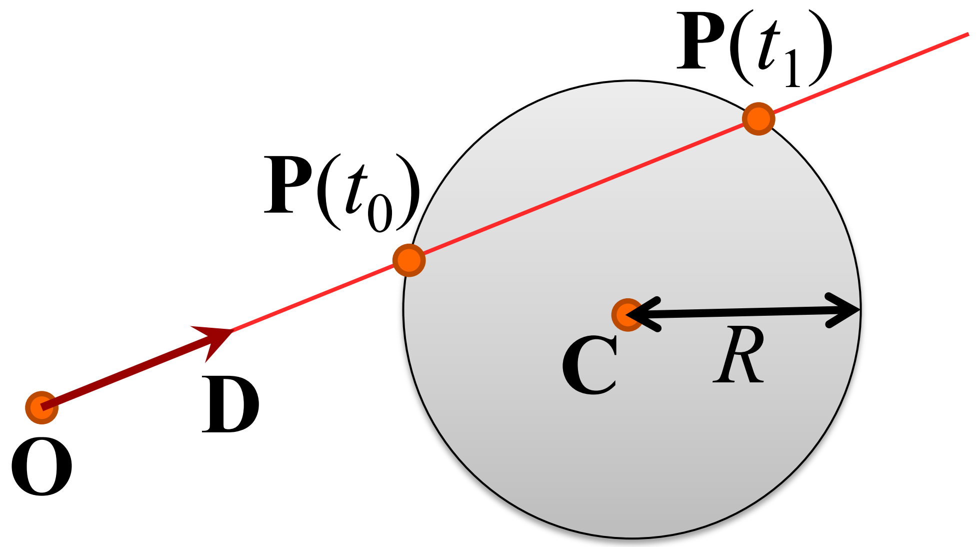
Sphere-ray intersection

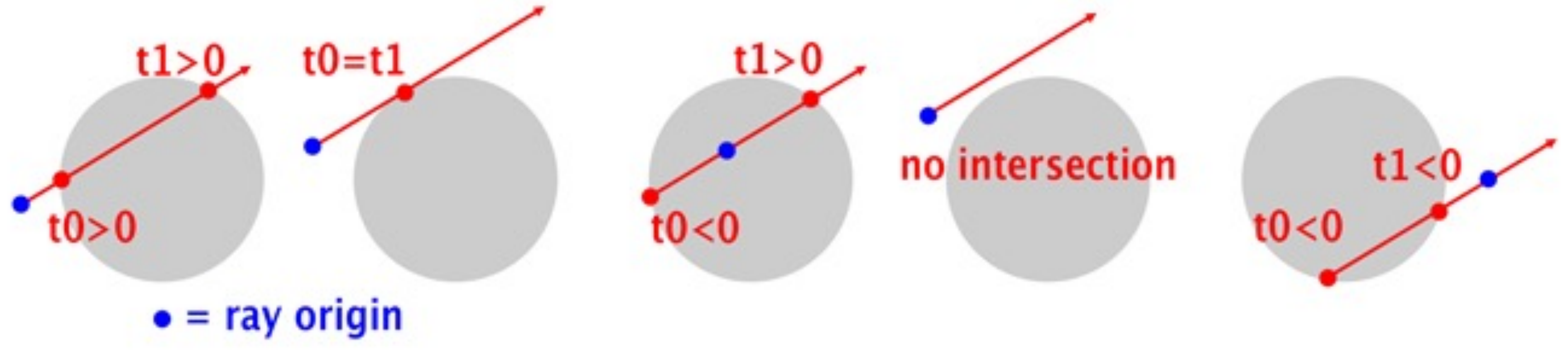
$$at^2 + bt + c = 0$$

$$a = \mathbf{D} \bullet \mathbf{D}$$

$$b = 2[\mathbf{O} - \mathbf{C}] \bullet \mathbf{D}$$

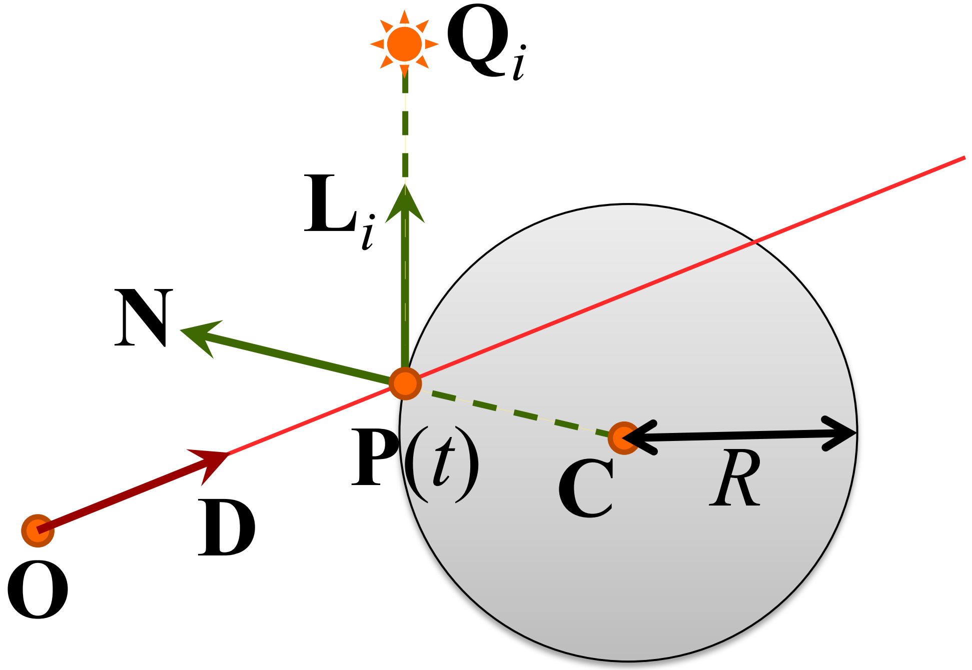
$$c = [\mathbf{O} - \mathbf{C}] \bullet [\mathbf{O} - \mathbf{C}] - R^2$$





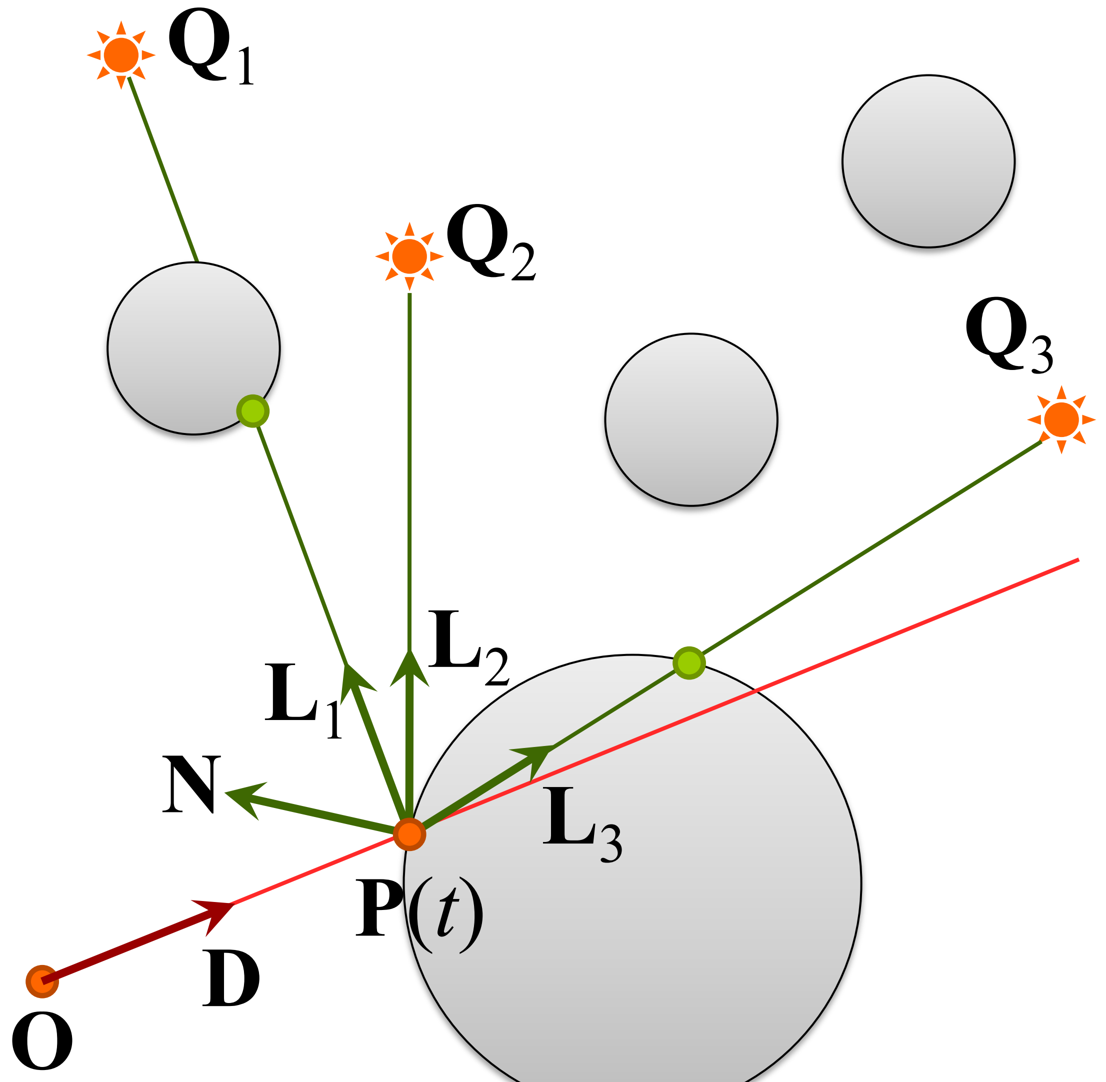
What information do we need?

- We know:
  $\mathbf{O}$, $\mathbf{D}$, $\mathbf{C}$, R, t
- Easy to calculate:
  $\mathbf{P}(t)$
- We also need:
  - $\mathbf{N}$ – the normal
  - $\mathbf{L}$ – vector to light



Shadow rays

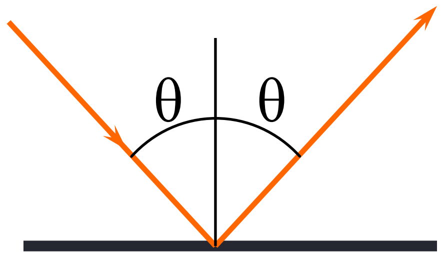
- Determine which lights are visible
- Use ray tracing method with origin at $\mathbf{P}(t)$ and vector $\mathbf{L}_i$
  $\mathbf{Q}(s)=\mathbf{P}(t)+s\mathbf{L}_i$
- L_3 can be discarded based on testing the normal



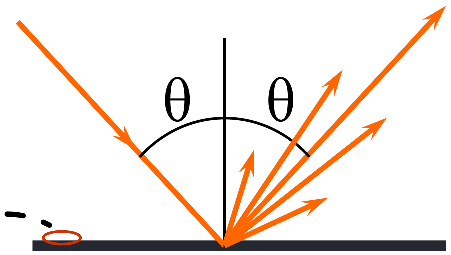
Intersection with the object itself

- When intersecting that shadow ray with the object itself, we may get the same intersection point at time t close to 0. This can be caused due to numerical (precision) issues.

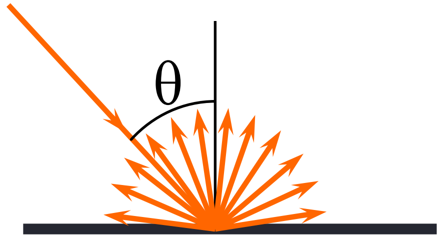
How do surfaces reflect light?



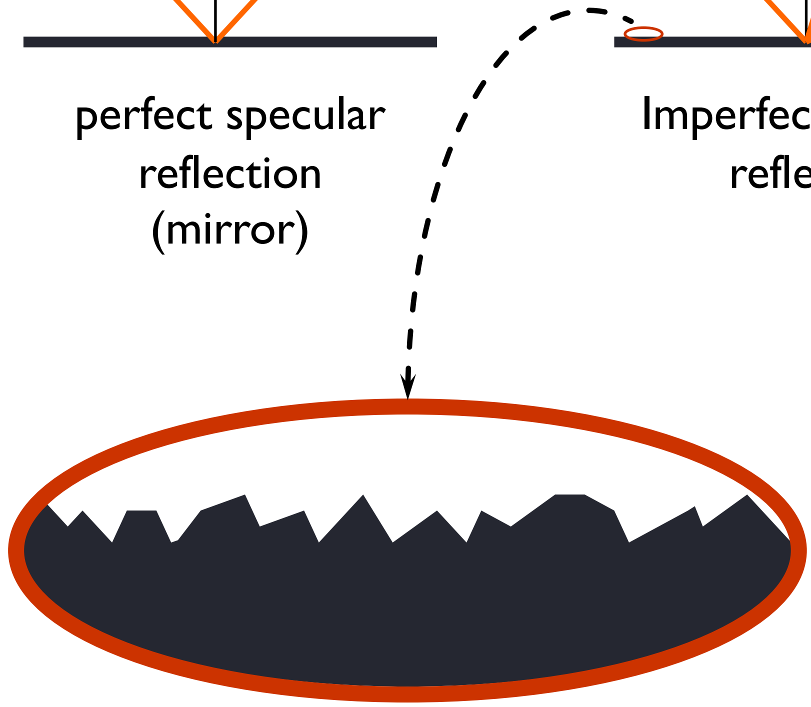
perfect specular reflection (mirror)



Imperfect specular reflection

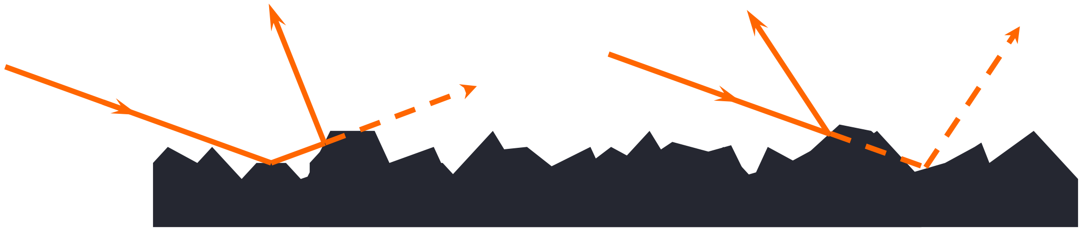


diffuse reflection (Lambertian reflection)



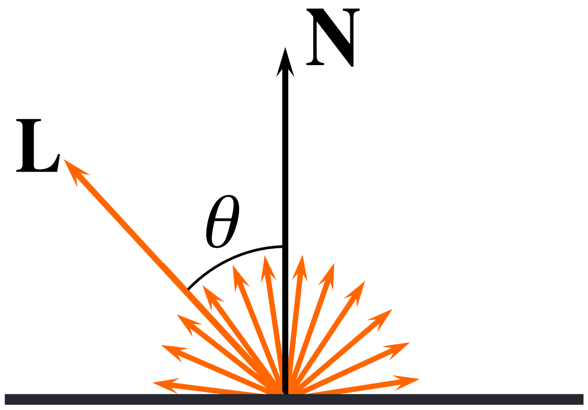
Comments on reflection

- the surface can absorb some wavelengths of light
  - e.g. shiny gold or shiny copper
- specular reflection has “interesting” properties at glancing angles owing to occlusion of micro-facets by one another



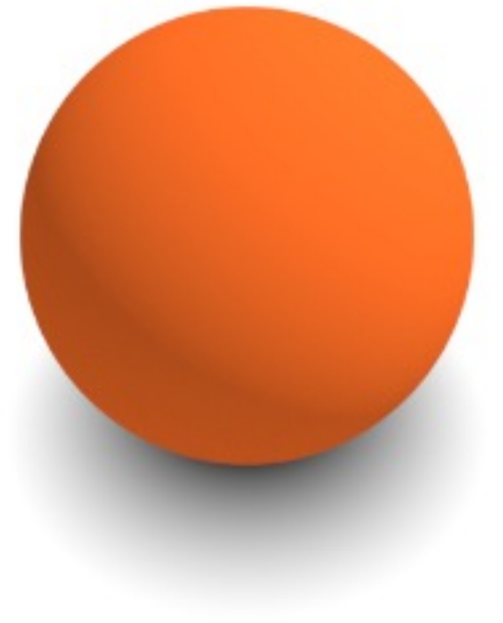
- plastics are good examples of surfaces with:
  - specular reflection in the light’s colour
  - diffuse reflection in the plastic’s colour

Diffuse shading calculation



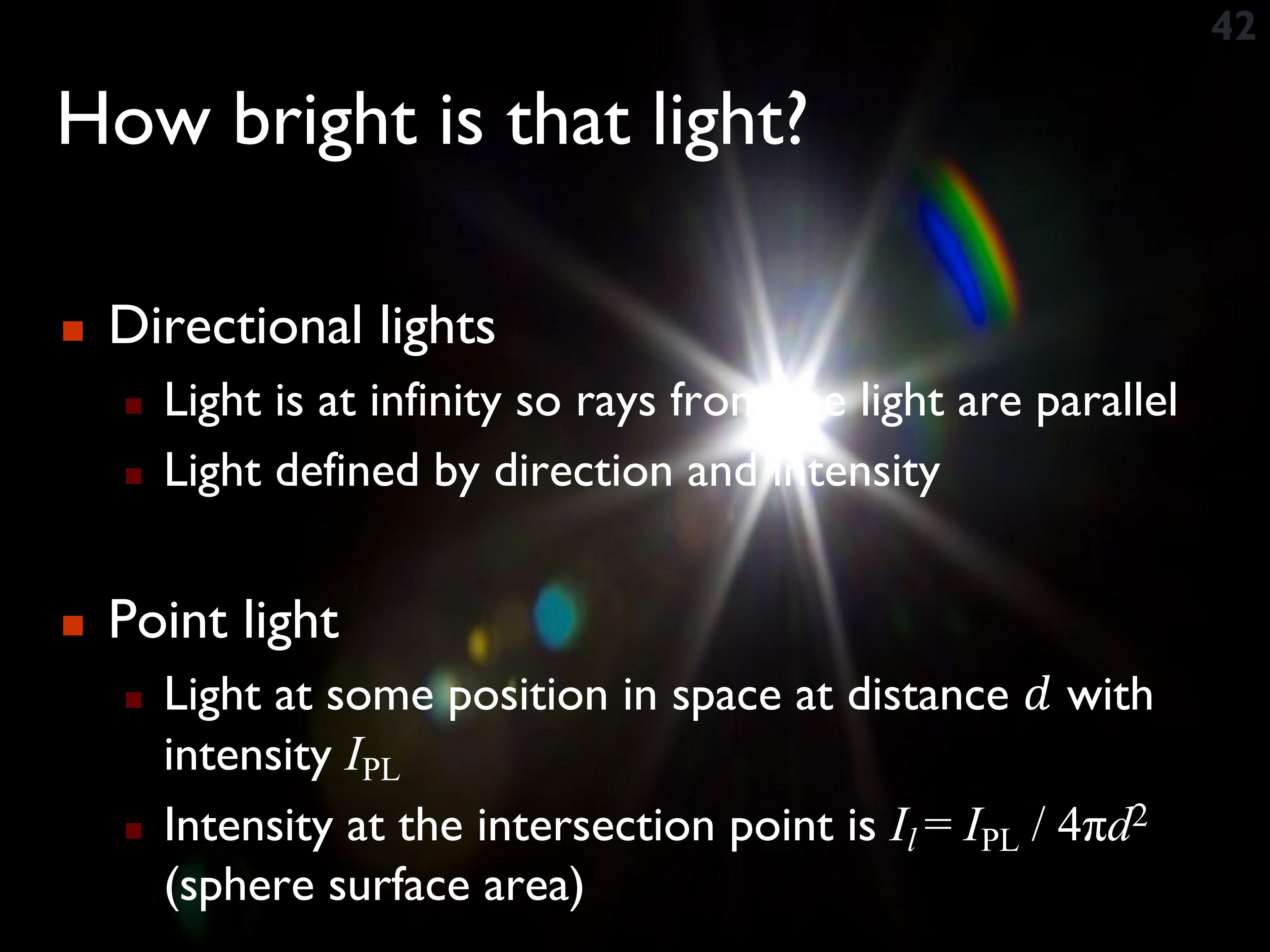
$$I = I_l k_d \cos\theta$$
$$= I_l k_d (\mathbf{N} \bullet \mathbf{L})$$

- Lambertian reflection
  - The incoming light is reflected evenly in all directions
  - The intensity depends on the angle

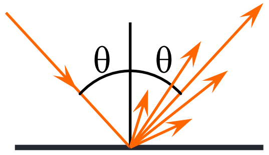


- L is a normalised vector pointing in the direction of the light source
- N is the normal to the object at the intersection point
- I_l is the intensity of the light source as seen from the intersection point
- k_d is the proportion of light which is diffusely reflected by the surface, it is a property of the surfaces material

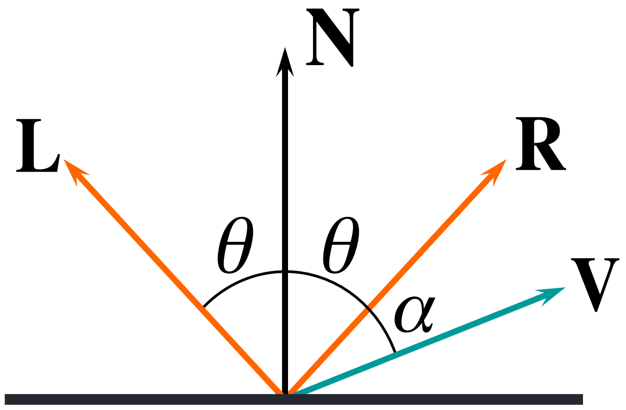
How bright is that light?



- Directional lights
  - Light is at infinity so rays from the light are parallel
  - Light defined by direction and intensity
- Point light
  - Light at some position in space at distance d with intensity I_{PL}
  - Intensity at the intersection point is $I_l = I_{\mathrm{PL}} / 4\pi d^2$ (sphere surface area)

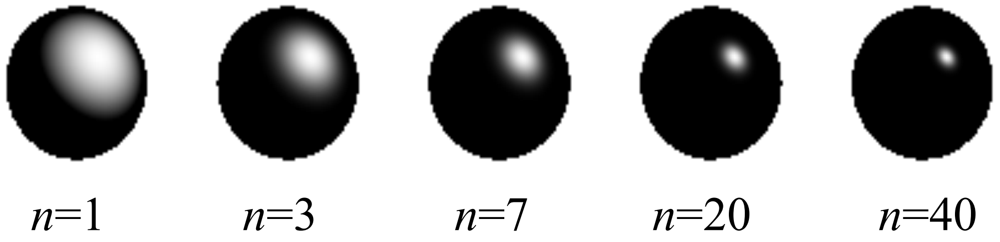


Phong specular reflection

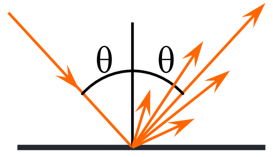


$$I = I_l k_s \cos^n \alpha$$
$$= I_l k_s (\mathbf{R} \bullet \mathbf{V})^n$$

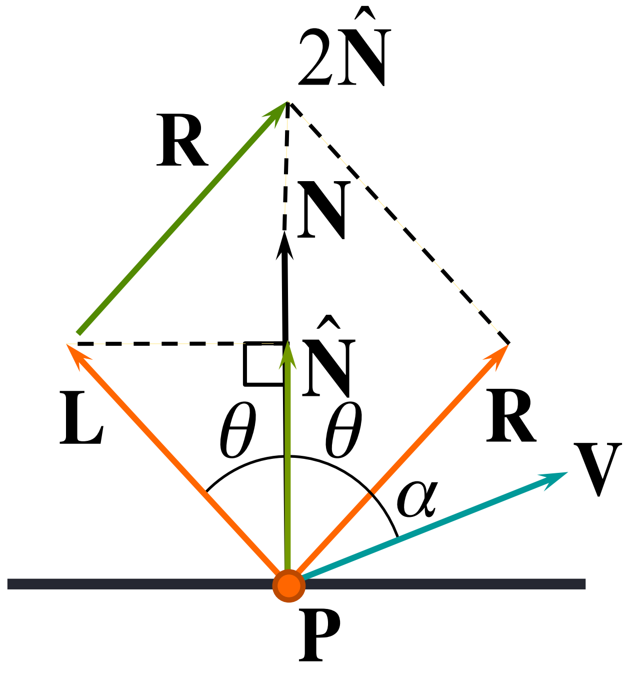
- The Phong model is an easy-to-calculate *approximation* of specular reflection



- L is a normalised vector pointing in the direction of the light source
- R is the normalised vector of perfect reflection
- N is the normal to the polygon
- V is a normalised vector pointing at the viewer (D) (changes as the camera moves)
- I_l is the intensity of the light source as seen from the intersection point
- k_s is the proportion of light which is specularly reflected by the surface
- *n* is Phong's *ad hoc* "roughness" coefficient
- *I* is the intensity of the specularly reflected light



Calculating **R** and **V**

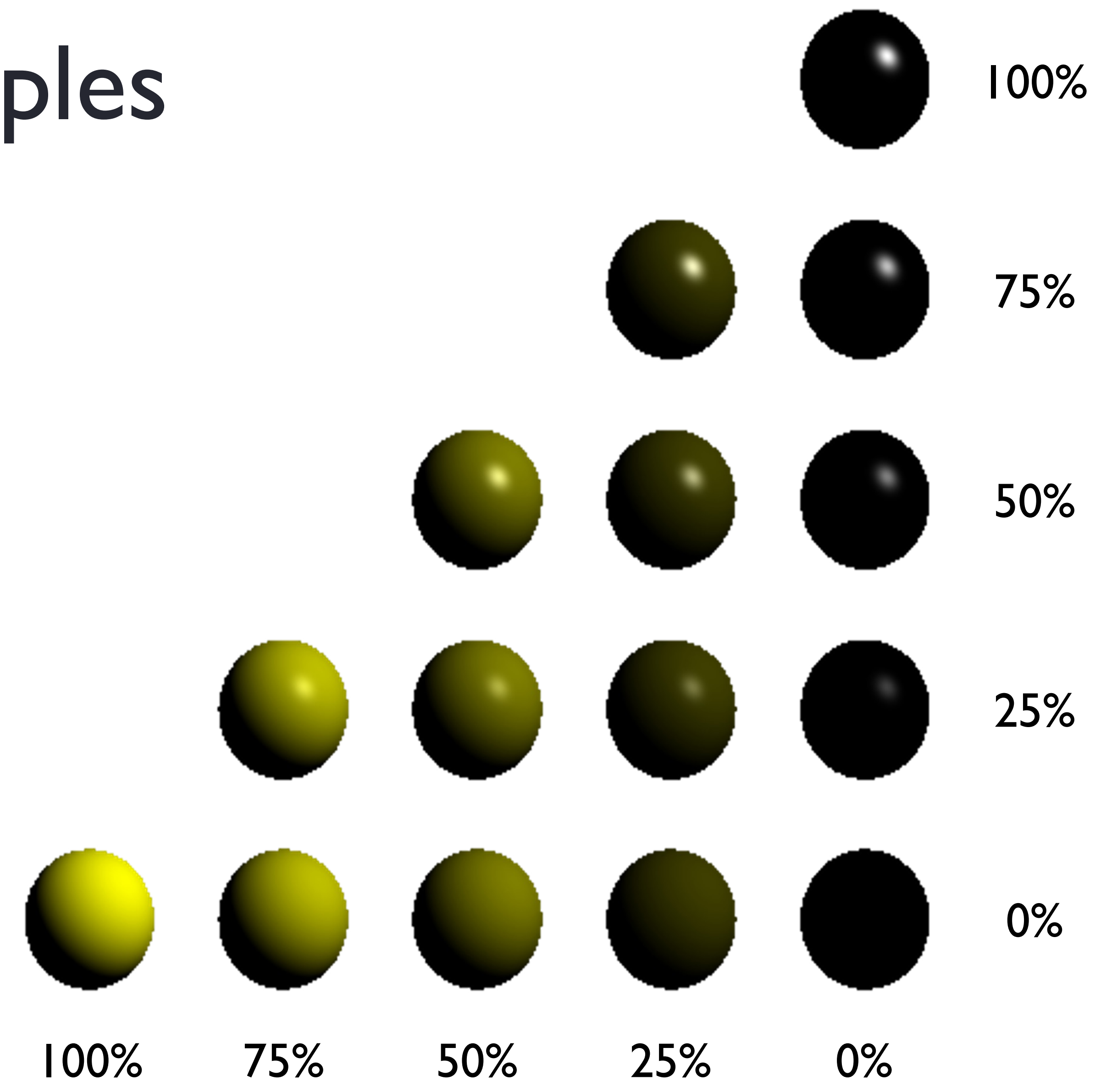


$$\hat{\mathbf{N}} = \mathbf{N}(\mathbf{L} \bullet \mathbf{N})$$

$$\mathbf{L} + \mathbf{R} = 2\hat{\mathbf{N}}$$

$$\mathbf{V} = \frac{-\mathbf{D}}{|\mathbf{D}|}$$

Examples



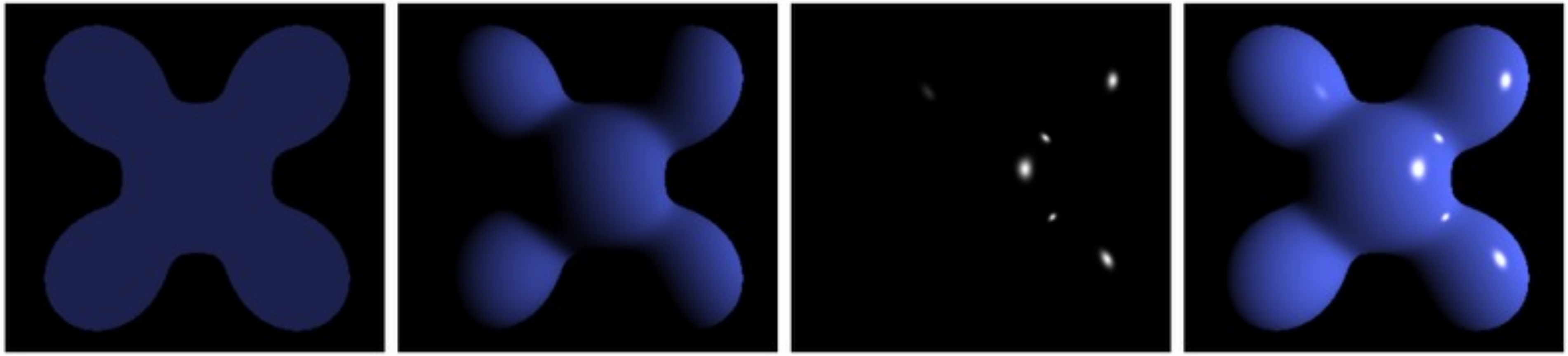
specular reflection

diffuse reflection

How good an approximation is this?

- Lambertian diffuse reflection
  - a good approximation to physical reality
- Phong specular reflection
  - a rough approximation to physical reality
- but no consideration of inter-object reflections
  - all illumination is directly from the light
  - so we cheat!
  - assume that all light reflected off all other surfaces can be amalgamated (joined) into a single constant term: “ambient illumination”

Ambient illumination

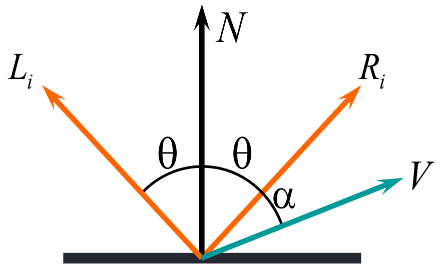


ambient + diffuse + specular = simple shading

- Ambient illumination is not realistic but it makes objects look better
- Without it, any part of the object that isn’t lit would be black

Shading: overall equation

- the overall shading equation is thus ambient illumination plus diffuse and specular reflections from each light source



$$I = I_a k_a + \sum_i I_i k_d (\mathbf{L}_i \bullet \mathbf{N}) + \sum_i I_i k_s (\mathbf{R}_i \bullet \mathbf{V})^n$$

- the more lights there are in the scene, the longer this calculation will take