Ray Tracing and an Introduction to Shading

Ashtan Mistal

University of British Columbia

A Brief Outline

- Physics of Light
- Approximations and Traditional Shading Models
- Ray Tracing
- Putting it into perspective + optimization

What makes an image?

- Lighting conditions
- Scene geometry
- Surface properties
- Camera type and placement

Figure: Graphics from *Minecraft*

What kinds of lighting and graphics effects do you see in this image?

Figure: Graphics from the movie *Cars*

What about this image?

Figure: Graphics from *The Last of Us Part II*

What about this image?

Physics of Light

- The Basics of Light
- Reflection of light

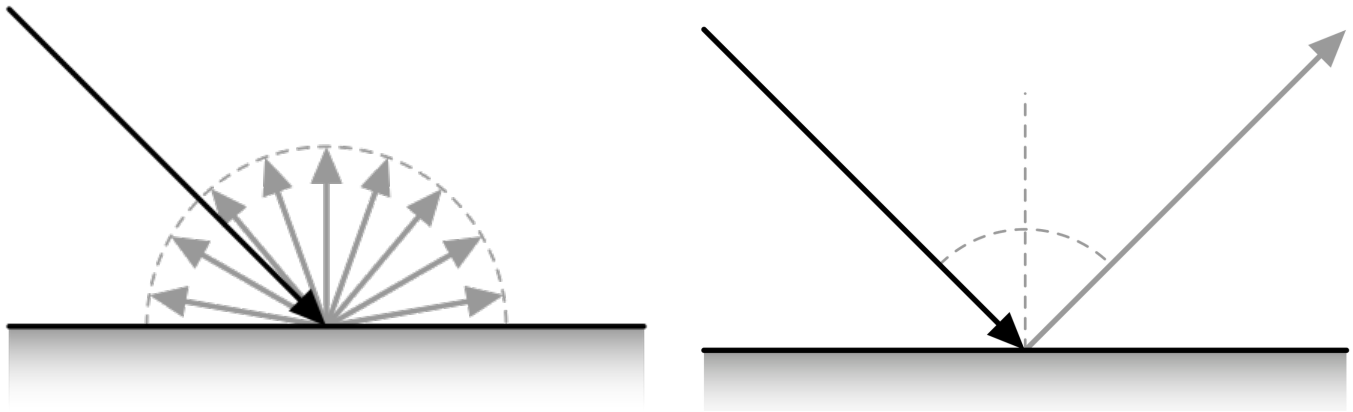


Figure: Diffuse (left) versus Specular reflection (right)

Reflection: Mathematically

- Angle of incidence and angle of reflection
  - Law of reflection: angle of incidence $=$ angle of reflection

$$\theta_i = \theta_r$$

- Surface normals: A vector perpendicular to the surface.

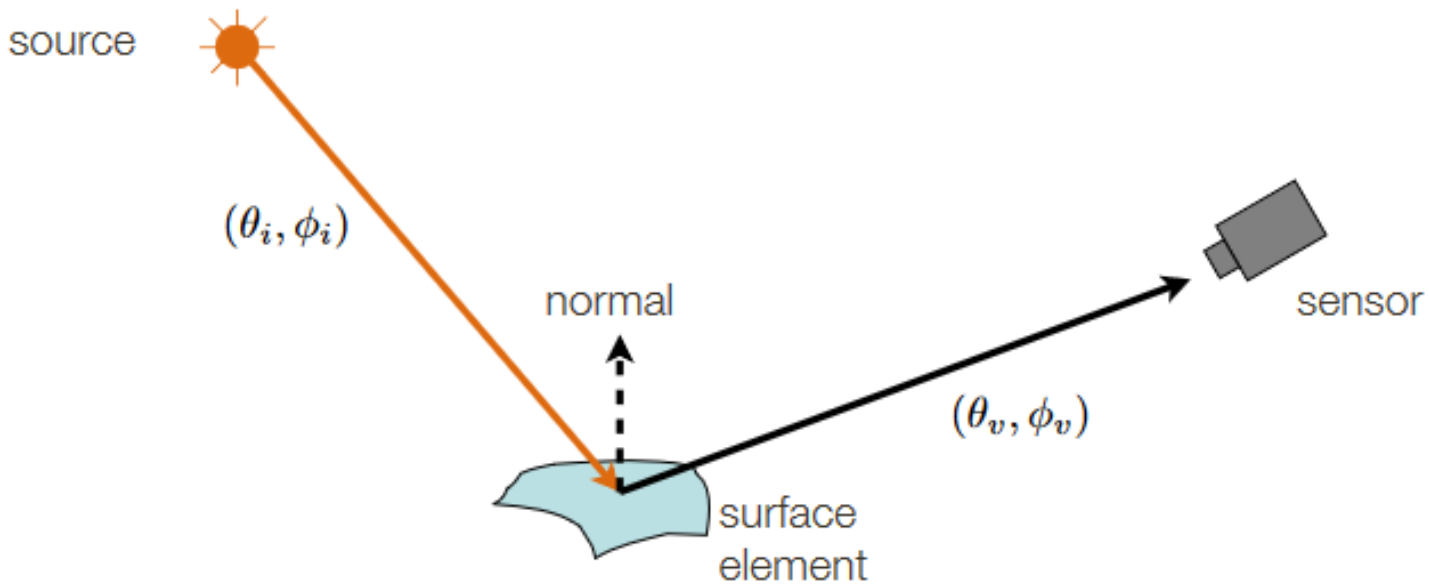
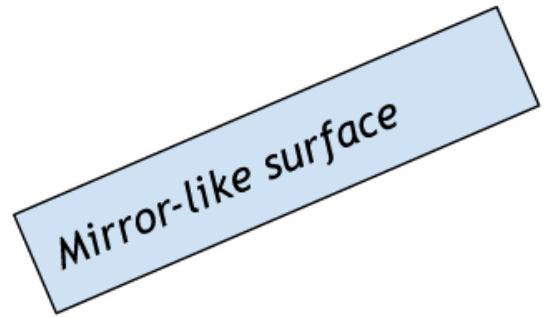


Image Credit: Leonid Sigal, CPSC 425 Lecture Slides

Tracing light through an object

Let's start with a point light source interacting with a mirror. If I want to trace the light from a light source to a point on a surface, I can do so by first finding the direction of the light. This is pretty arbitrary, as point light sources scatter in all directions. So, we can just pick one. Let's call this our ray.

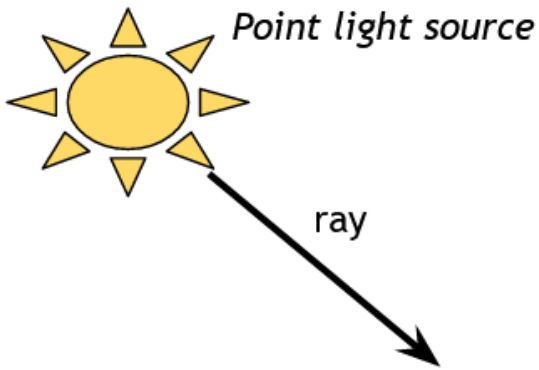
Point light source

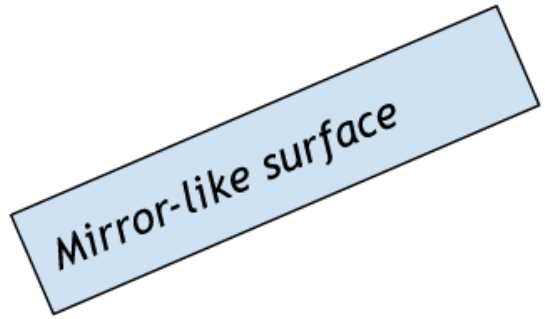


Mirror-like surface

Tracing light through an object - continued

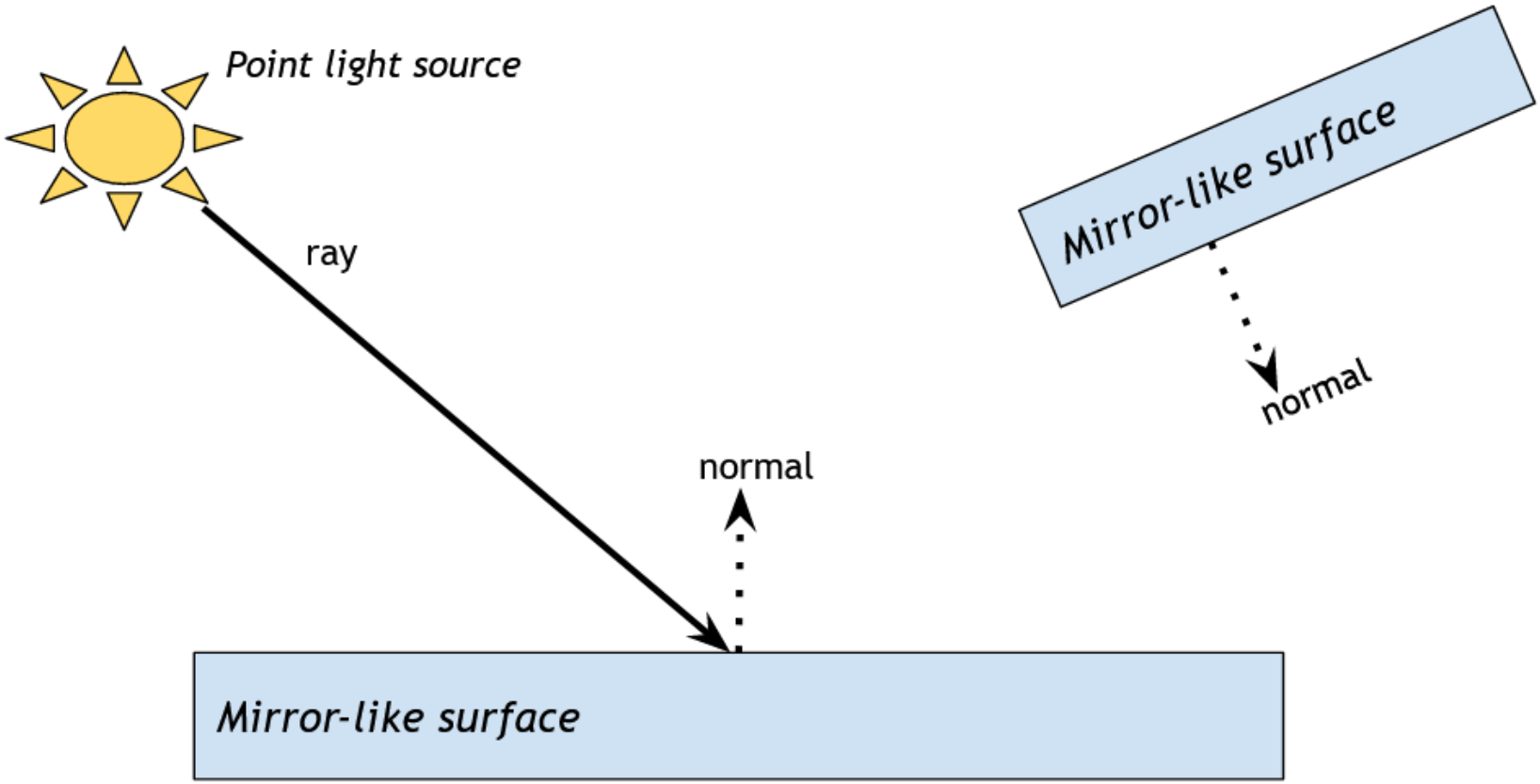
I then want to follow that ray until it hits a surface.





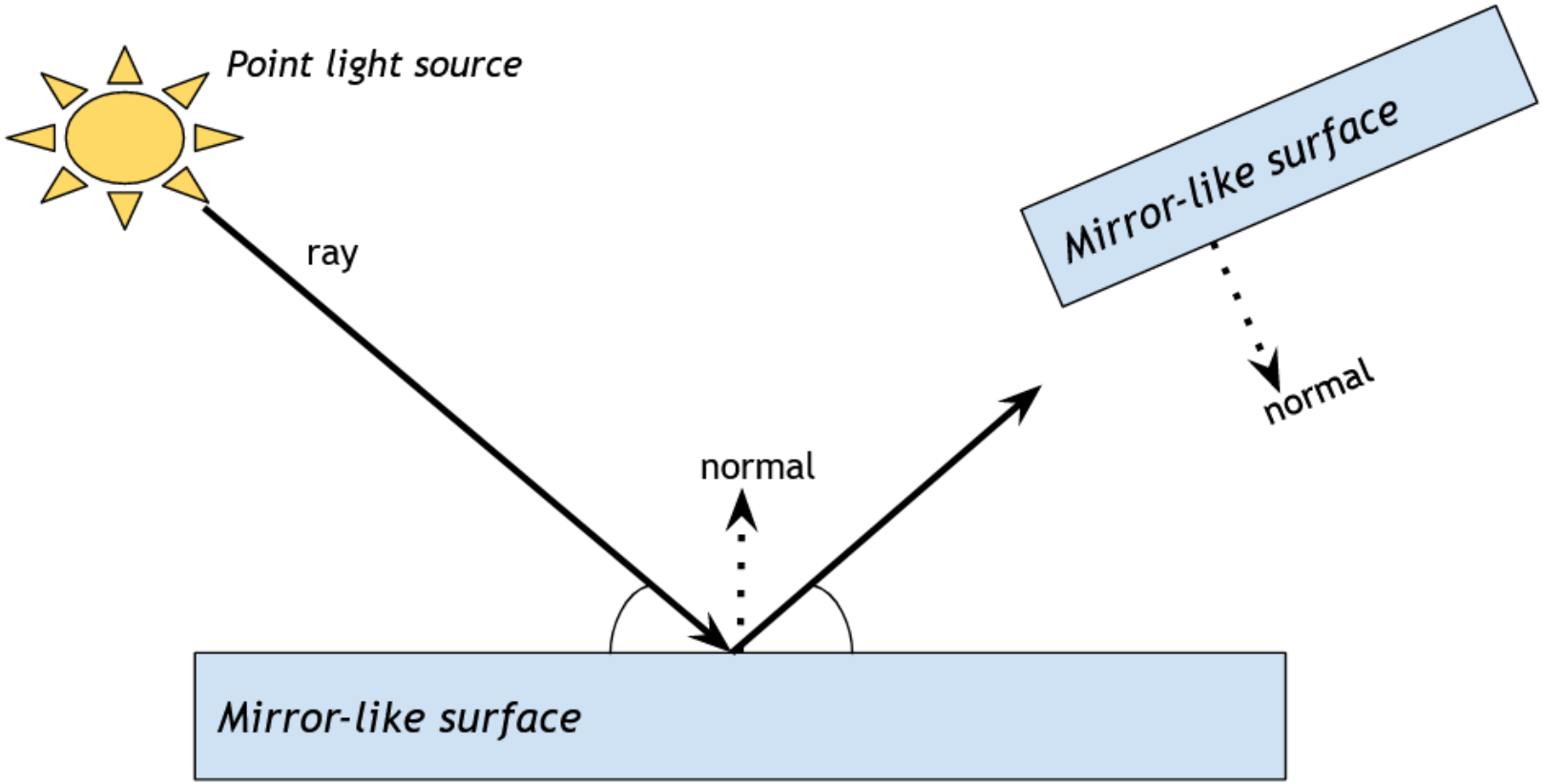
Tracing light through an object - continued

Once it hits that surface, I find the normal of the surface at the point of intersection.



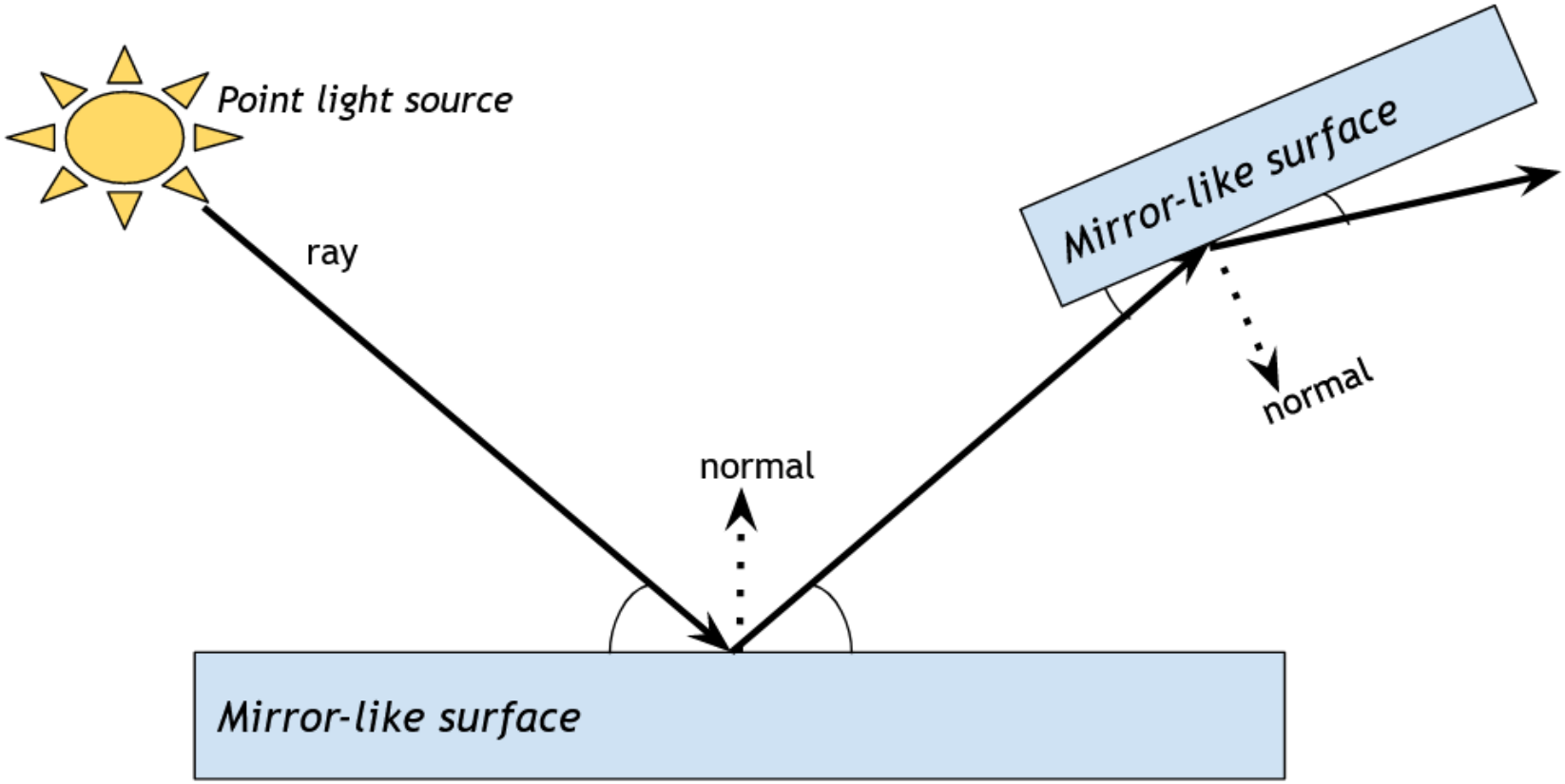
Tracing light through an object - continued

I then want to find the angle of incidence between the normal and the ray, and then bounce the ray off the surface according to the law of reflection.



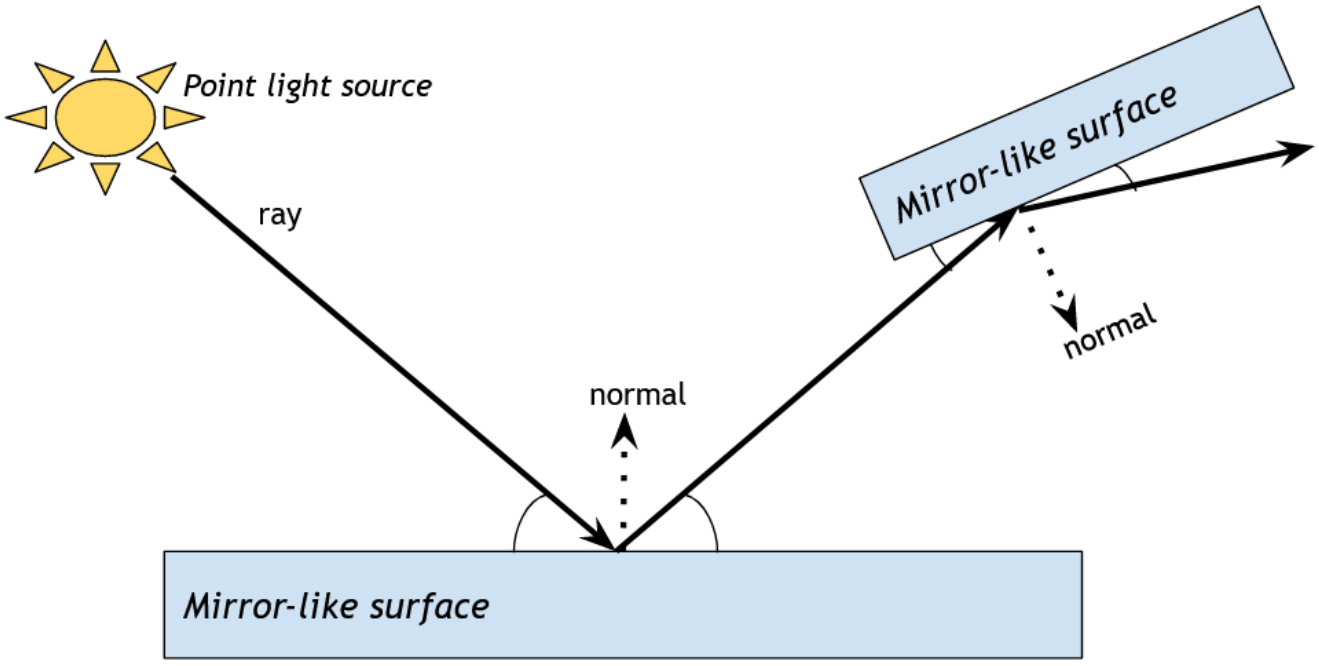
Tracing light through an object - continued

We can then move on to the next surface - whatever the light is heading towards next - and repeat the process.



This process of casting a ray from a light source to a surface and determining where it first hits a surface is called ray casting.

Where the light ends up



Once the light hits the camera, we now know the path of the light up until it hits the camera. The object that it was at immediately before will be visible in our scene.

Scene Geometry

To view the scene we're looking at today, navigate to the following website:
`https://phas.ubc.ca/~ashtan/`

From there, go to "WebGL Demonstrations" and click on "Scene Geometry".
Here, you can view the scene that we've recreated in real life and interact with it.

Shading

Say we know where the light source is, where we're viewing the object from, and where the object is in space.

How can we approximate the colour of an object, just knowing what we talked about?

A Simple Approach to Shading

From where the light source is and the face of the object that's being lit up, we can determine the angle of incidence between the light source and the face of the object.
A simple approach is to just make the object brighter the smaller the angle of incidence is.

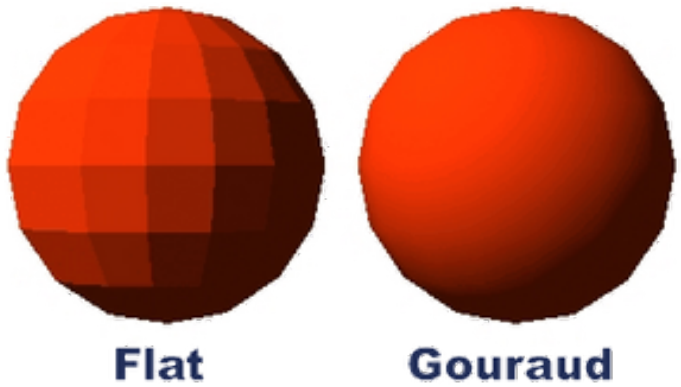


Figure: Simple approaches to shading

An Equation to Model Light

Let's start to build something that we can use to calculate light at a given point.

It'll be a function, that will be in terms of the angle of the incoming light (θ_i, ϕ_i) and the angle of the camera, i.e. the viewing angle (θ_v, ϕ_v).

We'll call it the **Bidirectional Reflection Distribution Function**, or **BRDF** for short.

Building up the BRD

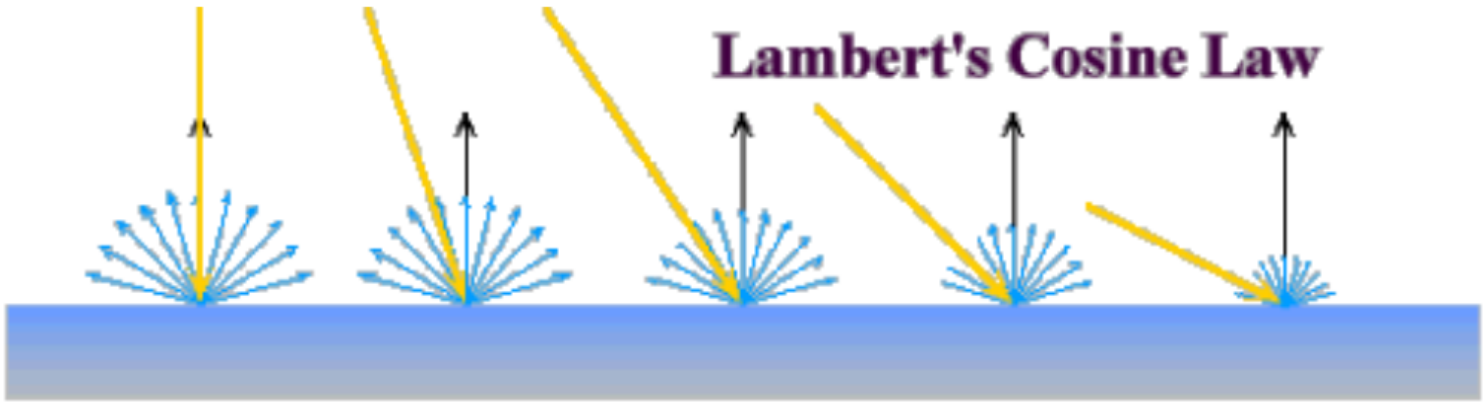
We know from our real-life experiences that light is also reflected off of other objects: If we go outside, even if the sun isn't directly shining on us, we can still see other things around us. Let's call that "ambient light".

Ambient light can be approximated in our BRDF with a simple constant representing the amount of ambient light in the scene:

$$\boldsymbol{BRDF}() = \alpha_{ambient}$$

Building up the BRDF

We talked about diffuse reflection before. We know it only depends on the incoming angle θ_i - this is called **Lambert's Cosine Law**:



$$BRDF\,(\theta_i) = \alpha_{ambient} + \alpha_{diffuse}\cos(\theta_i)$$

A Better Approximation

Lastly, we just need to add a term that depends on the viewing angle as well to determine the specular component.

It'll be brighter when the viewing angle is close to the reflected angle, so we have the following:

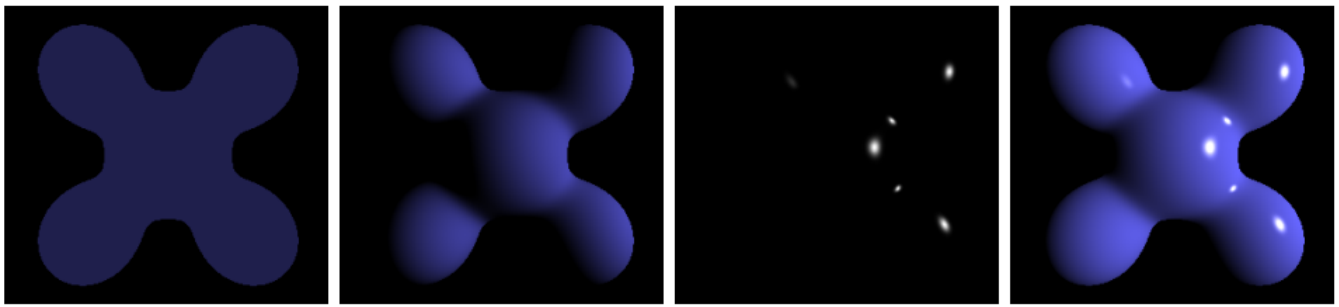
$$\textbf{\textit{BRDF}}\,(\theta_i, \theta_v) = \alpha_{ambient} + \alpha_{diffuse}\cos(\theta_i) + \alpha_{specular}\cos^{\beta}(\theta_v)$$

Here, β is the shininess of the object.

This is for two dimensions. In three dimensions the cos() terms are replaced by dot products with the normal vector, giving dependence on ϕ angles also.

A Better Approximation

So we can add up all of these components - ambient, diffuse, and specular - to determine the overall shading of the object. This is then done for all the faces of the object, and all objects in the scene.



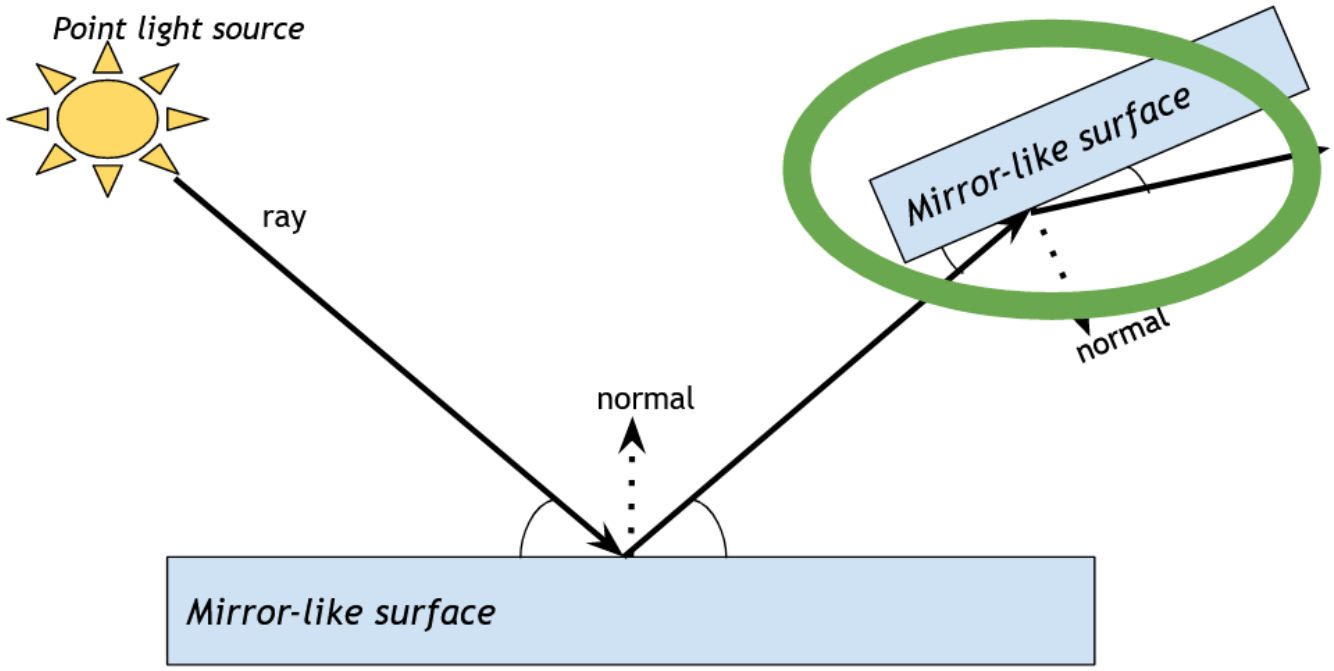
You can see what this looks like in our WebGL demo by navigating to the same website as before, clicking on WebGL demos, and click the "Phong Shading" button.
`https://phas.ubc.ca/~ashtan/`

Phong Shading in *Half-Life 2*

Figure: Phong shading used in *Half-Life 2*. Note the reflection of the light source on the rocks.

What's missing in our BRDF?

We now know how to determine the shading of an object given the position of the light source and the camera. But we haven't actually done anything *more* with the light rays that are being reflected off of the initial object.



So, we need to figure out how to follow the reflected light throughout the scene until it hits our camera.

A Naive Approach

Intuitively, light travels from the light source, interacts with a scene, and then travels either to the camera, gets fully absorbed, or exits the scene.
So we can build an algorithm that does exactly that.

```
for each light source:
    for each angle:
        cast a ray from light source at angle
        while the ray hasn't hit anything:
            if ray hits an object:
                determine the colour of obj
                add obj color to light source color
                Bounce ray off object and continue
            if ray hits camera:
                add ray color to pixel in camera frame
            If ray exits scene: break
```

In practice - A simple model

Using the algorithm we just discussed, let's render a sphere. Just one object.

Play around with the demo by clicking on "Ray Tracing - Sphere".

(website: `https://phas.ubc.ca/~ashtan/`)

Real Life Demonstration

Let's see what a ray trace would look like in real life using the same scene as before, this time using the real-life scene as a reference. We'll see why we're using our real life demonstration to trace the rays to begin with in a moment.

- Starting with a single light ray (laser)
- Point light source (flashlight)

How many rays of light do you think there are in this room?

A light bulb (of around 100 watts) emits 10^{20} photons per second (and that's *one light source*).

And we have to trace the path for **every** ray of light? It's an understatement to say that this is a lot of computation.

We need to find a way to reduce the number of rays of light that we have to trace.
First, let's change our perspective on this problem. (pun intended)

A change in perspective

- We know light will travel in a given path from the light source to the camera.
- We also know that light will travel in a given path from the camera to the light source.
- These paths are the same!

When we have light exiting the scene without hitting the camera, it's a waste of computational power to bother calculating the path of that ray.

Outwards from the camera

How do we do this?
We first need to take a look at how we're representing our camera. This is dependent on whether we're looking for accurate perspective, or if we're looking to analyze a selection of a scene.

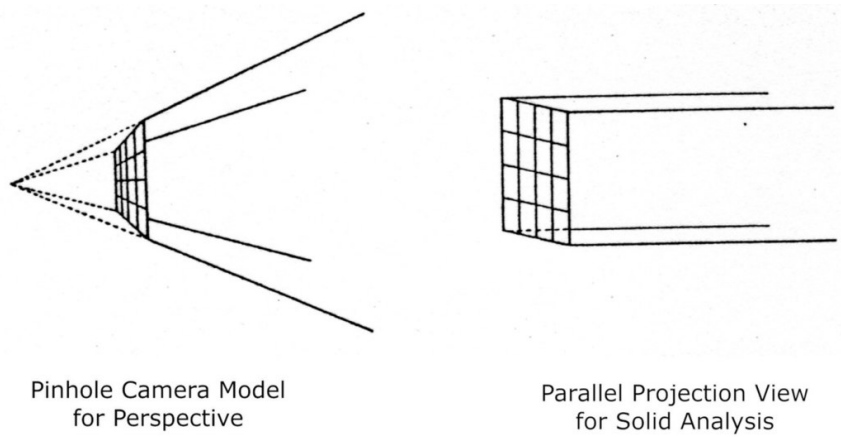


Figure: Different camera models and their field of view

Choosing the angle of the ray

If we draw a line from the pinhole to the pixel we're interested in, it forms the beginning of the ray that's already in the direction we want it to go.

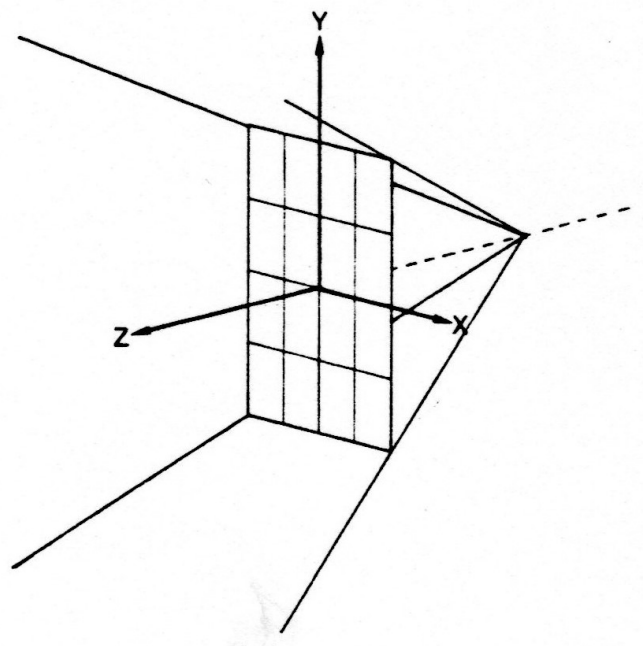


Figure: Camera local coordinate system with the "screen" in the Z=0 plane

Ray tracing our scene: Putting it into perspective

Check out a rendered video of this scene on the Demos page, under "Ray Tracing - Scene".

We don't actually trace 10^{20} rays

We can't actually trace 10^{20} rays, of course. That'd be crazy.

The actual number is comparatively small.

When we choose, say, a 1920x1080 camera frame, we have 2,073,600 pixels - and therefore 2,073,600 rays to trace. We can also choose to cast multiple rays per pixel, which will increase the accuracy of the scene.

We're almost there! What's didn't we cover today?

- Multiple reflections coming from the same point
  - e.g. a light ray partially reflecting off of a surface and partially being transmitted through (see Figure 9)

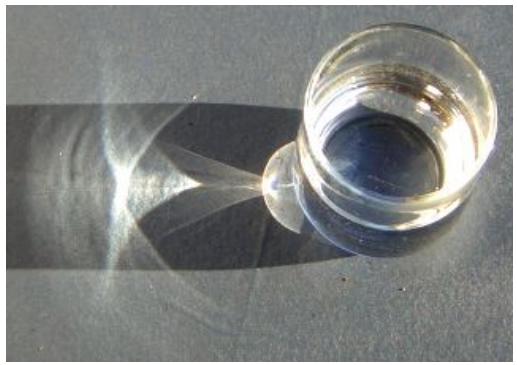


Figure: Caustics produced by a water glass

- Recursive ray tracing is a solution to this problem

We're almost there! What's didn't we cover today?

- De-noising the image
  - One ray per image leads to a lot of noise caused by jagged edges
  - there are a few de-noising techniques that can be used to reduce the noise
- Creating textures
  - This is a complicated computer graphics topic! You could have a whole course on geometric modelling.

What benefits does ray tracing have?

- It's a very accurate way of rendering a scene - it can be used to create photorealistic images
- It allows us to model complex phenomena, such as soft shadows, volume rendering, and caustics
- It's a very flexible rendering technique

But we have to ask - Are these benefits worth the drastically increased computational cost?