Ray Tracing and an Introduction to Shading

Ashtan Mistal

November 29, 2022

A Brief Outline

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

Figure: Graphics from the video game *Minecraft*

Physics of Light

- The Basics of Light
- Reflection of light

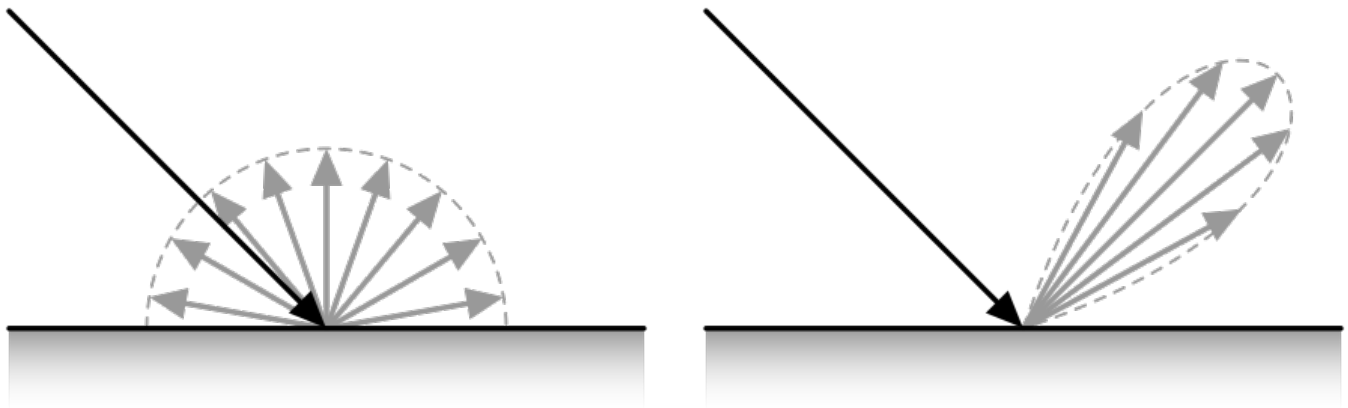


Figure: Diffuse (left) versus Specular reflection (right)

Reflection: Mathematically

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

$$\theta_i = \theta_r$$

- Surface normals:
  The normal of the surface is just a vector that is perpendicular to the surface. The mathematical details of how to find the normal for any surface aren't important for us today.
  We'll focus on simple objects.

Tracing light through an object

Let's start with a simple case, with no diffuse reflection.
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.
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.
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.
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.
That object that it hits first is the object that is going to be visible to the light source.

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 Demonstration" and click on "Scene Geometry".
Here, you can view the scene that we've recreated in real life and interact with it.

Shading

We've figured out how to determine which objects are visible to a light source - meaning, what objects - or faces of objects - are lit up by a light source. Now, we need to figure out how to determine the colour of those objects: how they look when they're lit up.

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

Once light hits an object, reflecting that light off of the object and onto other objects is a bit of a complicated process - let's look at that a bit later.

So let's think of ways that we can approximate the colour of an object just knowing what we mentioned above.

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.

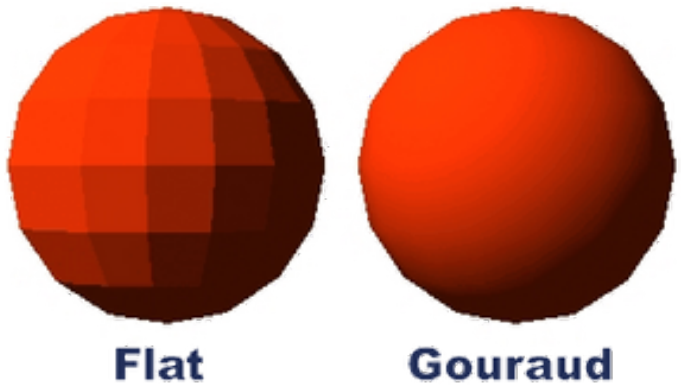


Figure: Simple approaches to shading

A Better Approximation

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".
We also talked about our diffuse and specular reflection before, so we can also take that into account.
In order to put diffuse reflection into practice, we can use the same approach as before with our simple model.
But specular is where we can take into account the direction of the camera. Only when the camera is in the same direction as the reflected ray will we see specular reflection.

A Better Approximation

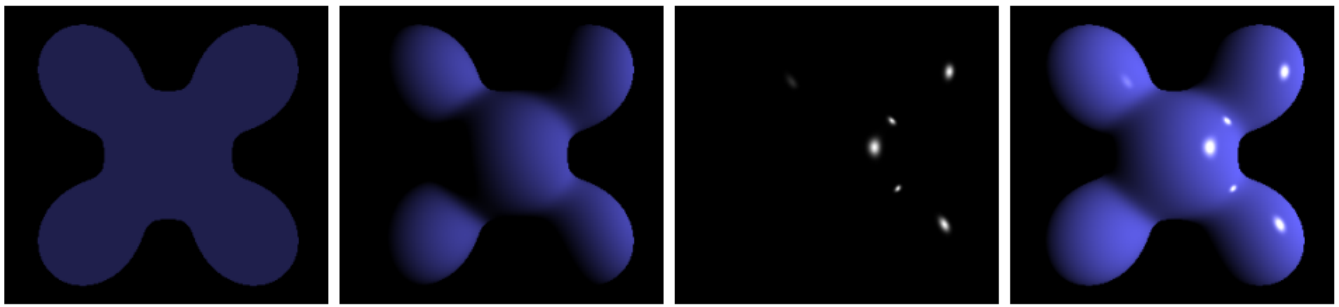
So, we can determine the level of specular reflection using the following formula:

$$\text{specular} = \max(\text{dot}(\text{normal}, \text{reflected ray}), 0)^{\text{shininess}} \quad (1)$$

where normal is the normal of the face, reflected ray is the reflected ray, and shininess is a constant that determines how shiny the object is (this changes based on the material of the object).

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/`

Tracing Rays

So far, we've been talking about how to determine the colour of an object given the position of the light source and the camera.
But we've been ignoring one key physical phenomena - we haven't actually done anything with the light rays that are being reflected off of the object.
From what we can see in the Phong shading demo, there aren't any shadows, and the scene doesn't look very realistic. The metallic objects aren't reflecting any of the light that hits them, and it just looks like a gray surface with some texture on it.
So, we need to figure out how to trace the light rays that are being reflected off of the object and determine what they hit.

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

Just as we mentioned before that the most simple objects to render are spheres, the most simple scene to render is a scene with only a sphere. So let's take a look at what that looks like. We've built up the scene using the algorithm we discussed above. Play around with the demo by clicking on "Ray Tracing - Sphere".

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?

We can make a very rough approximation that a light bulb (of around 100 watts) emits 10^{20} photons per second.
Regardless of if the actual number is higher or lower, we can see that this is a lot of rays of light.
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.
And, when we consider the size of the camera frame in comparison to the size of the scene, we can see that the probability of a ray of light exiting the scene without hitting the camera is very high.
So, we can draw the rays starting from the camera and going outward to the scene.

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.

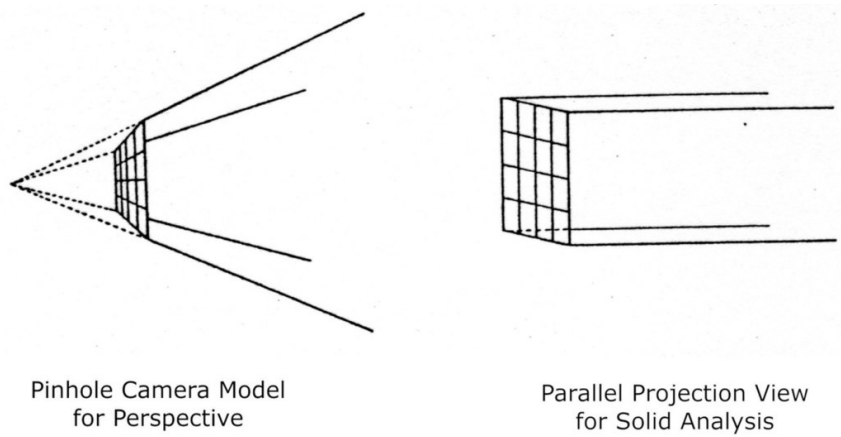


Figure: Different camera models and their field of view

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.

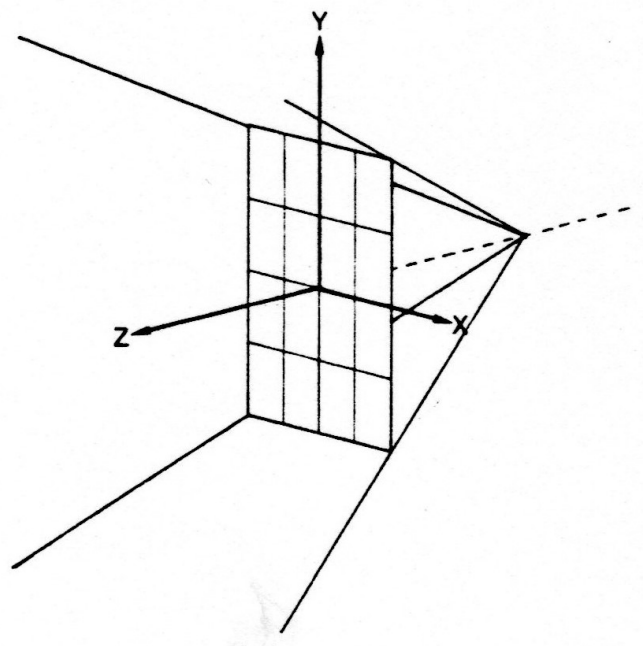


Figure: Camera local coordinate system with the "screen" in the Z=0 plane

Ray tracing our scene: Putting it into perspective

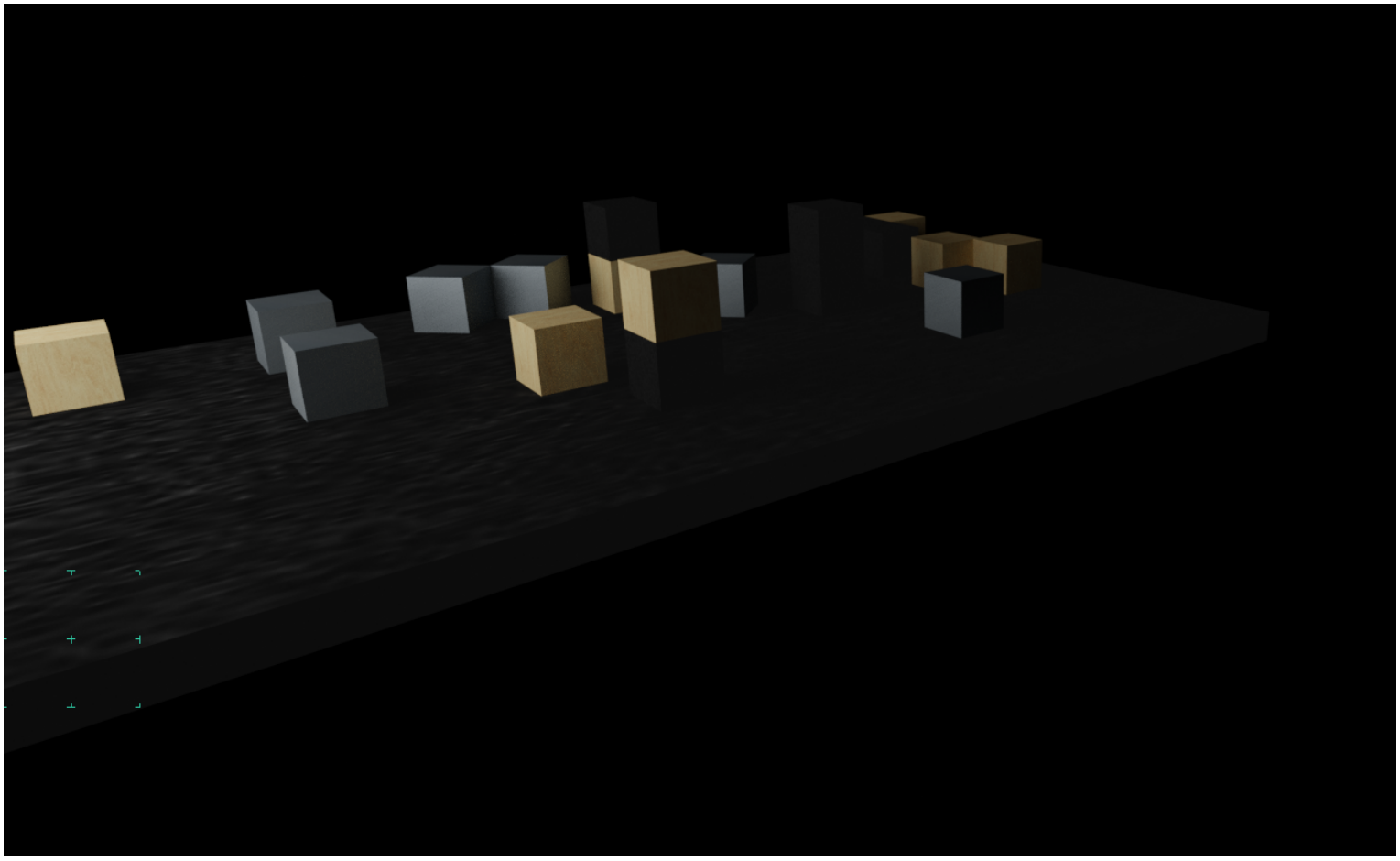


Figure: Ray tracing our scene

Check out a rendered video of this scene 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. With what we talked about before about the field of view, we can see that the number of rays we have to trace 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. but would be more computationally expensive.

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 7)

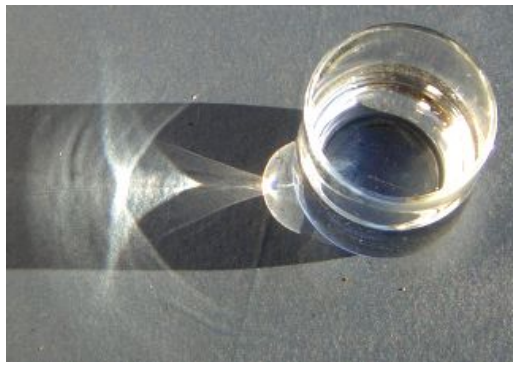


Figure: Caustics produced by a water glass

- 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
- 3d models and 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?