Raytracing

The Rendering Problem

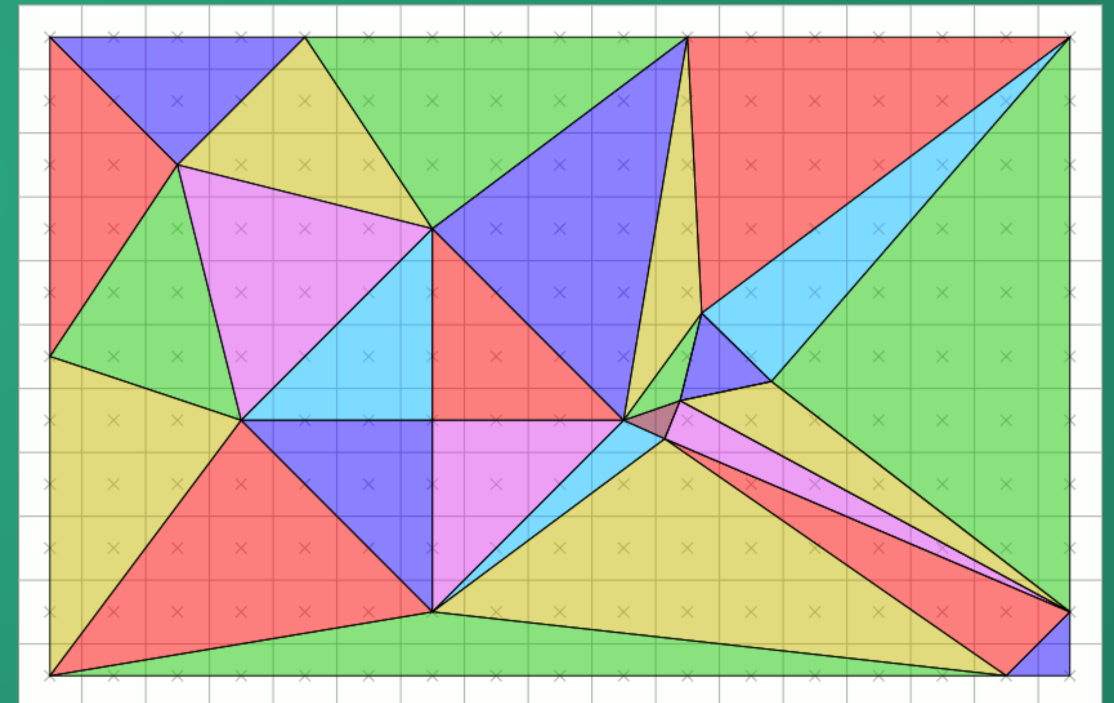
We have *shapes* and want to turn them into *pixels*. How do we do this?

Rasterization

What all GPUs do these days to render images

Idea: do some computations to directly express which pixels get filled in by particular shapes.

Bresenham's Line Algorithm

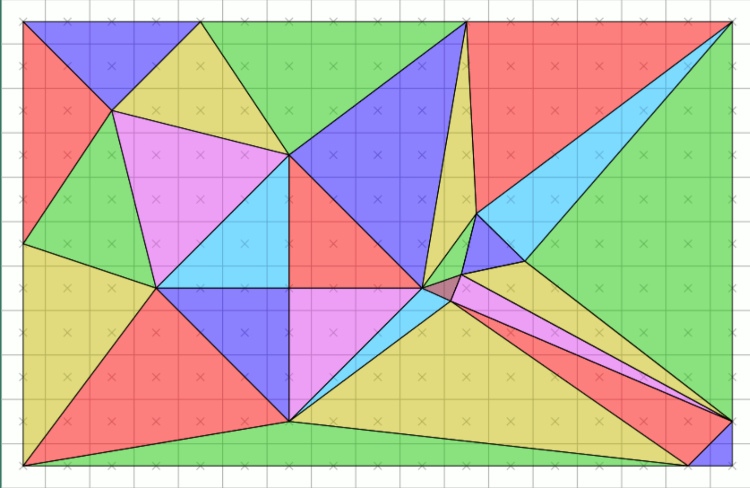


Top Left Rule

Crotalus Horridus, https://en.wikipedia.org/wiki/Bresenham%27s_line_algorithm#/media/File:Bresenham.svg

Drummyfish, https://en.wikipedia.org/wiki/Rasterisation#/media/File:Top-left_triangle_rasterization_rule.gif

Rasterization



Very fast, but does not carry any sort of physical information with it!

Just pushing shapes onto a grid.

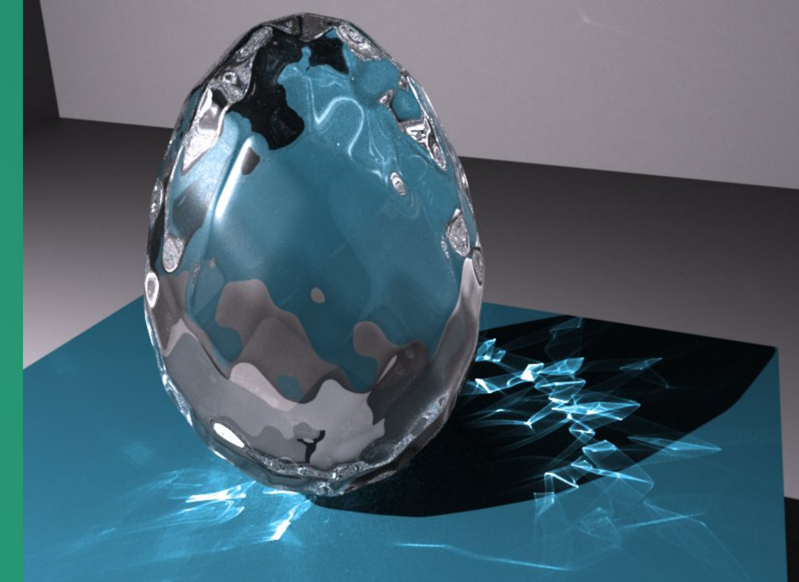
To compensate, we usually generate some information on the vertices (like light intensity) and rasterize that into the pixel grid as well. This is what you did in the shaders Hands-On.

But this still fails to capture non-local lighting!

Drummyfish, https://en.wikipedia.org/wiki/Rasterisation#/media/File:Top-left_triangle_rasterization_rule.gif

Things Rasterization Can't Do

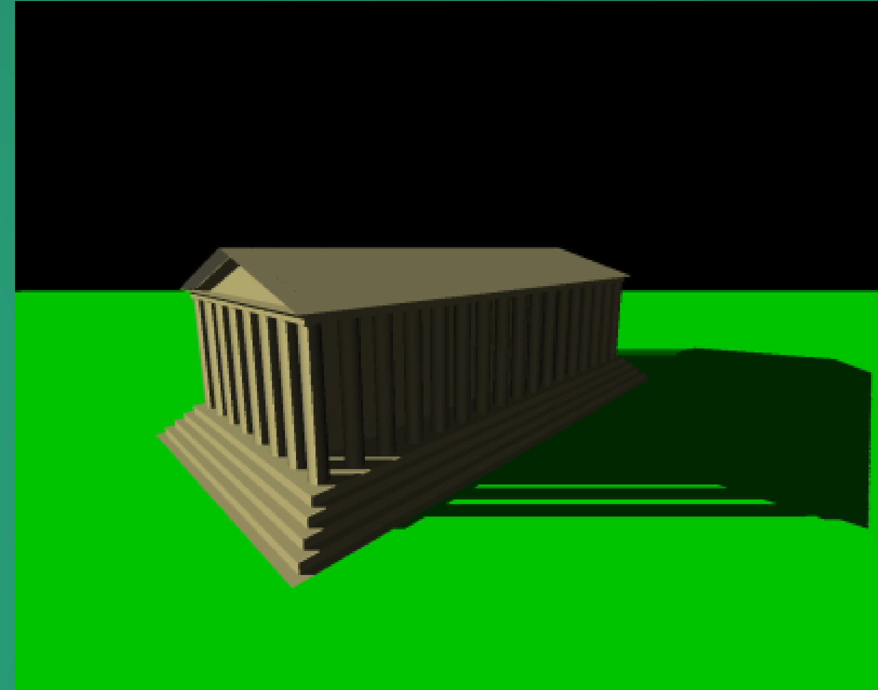
- Indirect Lighting
- Shadows
- Reflections
- Transparency/Translucency
- Caustics
- Ambient Occlusion

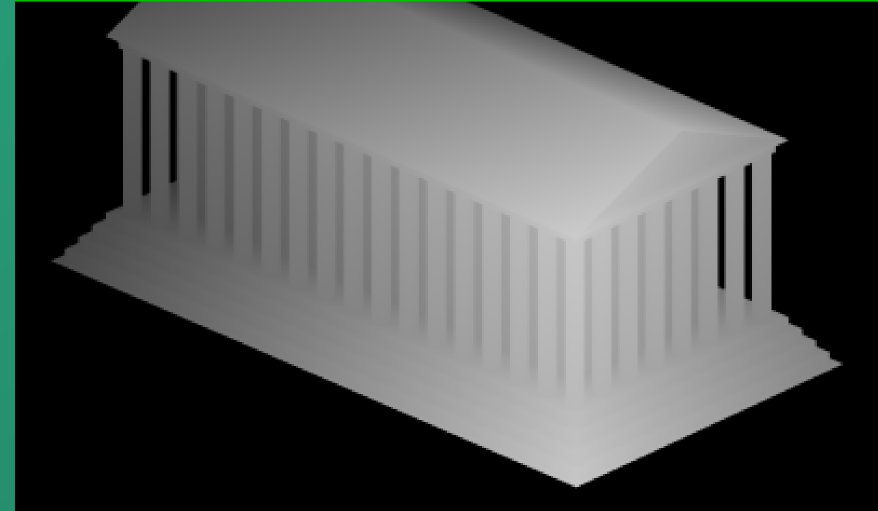


If you want to get these visual features with rasterization, you have to do a bunch of tricks!

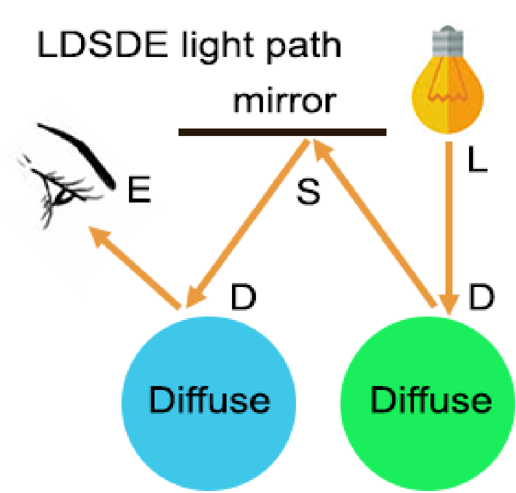
For example, to get shadows, we often use *shadow maps*.

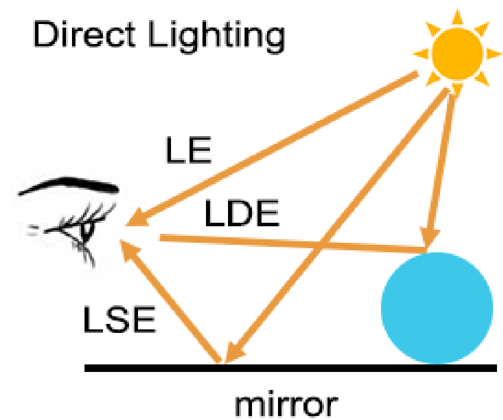
1. Render the scene from the perspective of the light.
2. Store this information in textures.
3. Render the primary scene using the shadow mapped textures.
4. Repeat for every light in the scene.

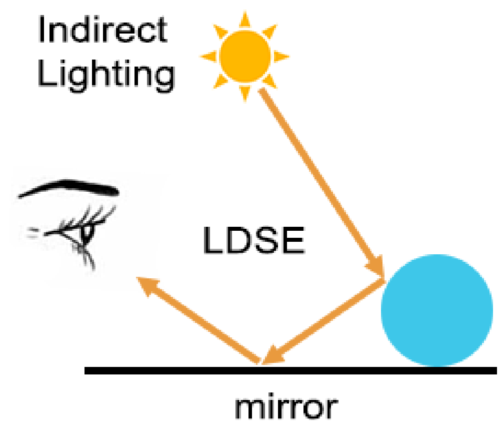




Pratetor Alpha, CCBYSA 3.0, https://en.wikipedia.org/wiki/Shadow_mapping







Raytracing attempts to capture this idea by simulating this process on the computer.

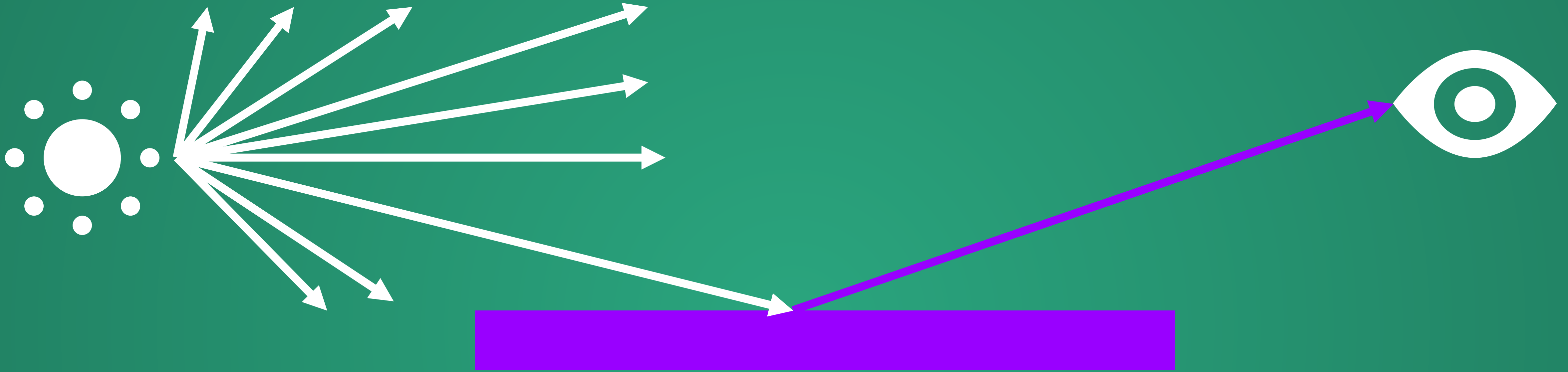
Instead of drawing shapes directly to a pixel grid, we're going to try to simulate light bouncing around the scene and eventually arriving at the pixel grid.

But we're going to almost immediately hit some serious issues trying this.

Raytracing Modes

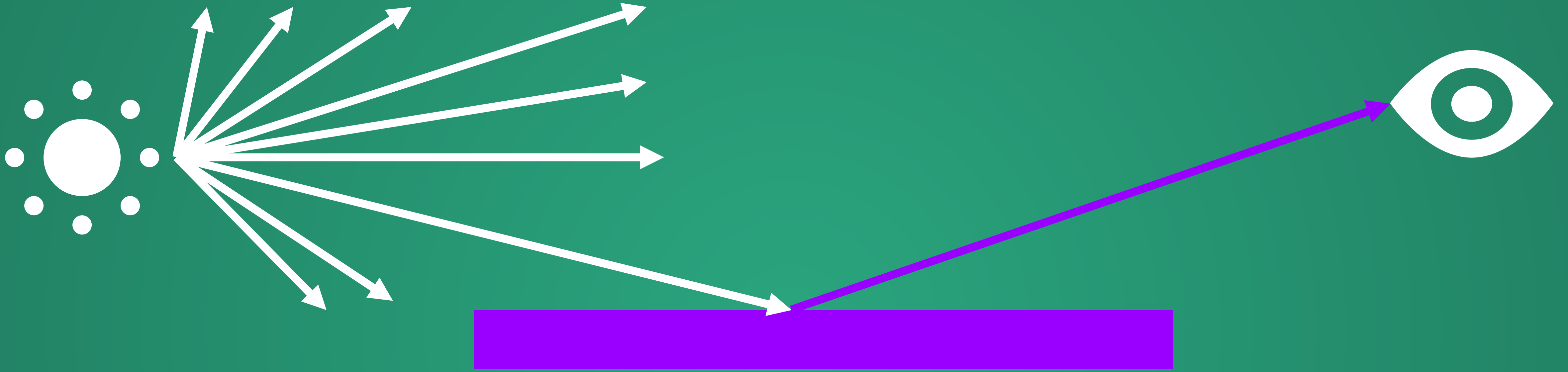
The simplest idea for raytracing is to trace rays from the light.

Known as forward raytracing.



1. Emit a photon from the source.
2. Trace its path, adding coloring as it hits surfaces.
3. If the photon hits the camera, register the color on the image.

The simplest idea for raytracing is to trace rays from the light.



But there is some loss, because the some photons never reach the camera.

When looking at the sky on a bright day, our eyes receive about 3×10^{14} photons per second.

A bright lightbulb emits on the order of 10^{20} photons per second.

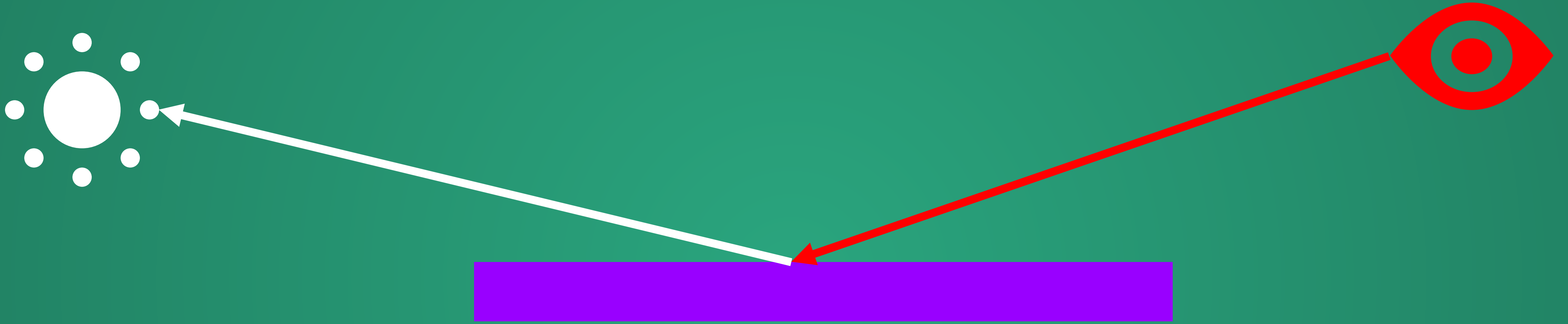
Fewer than 1 in 1,000,000 photons emitted by a light make it to our eyes.

Waste 99.9999% of our computation simulating photons that never make it to the image

or

be smarter

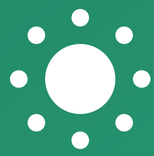
Solution: Grow Laser Eyes



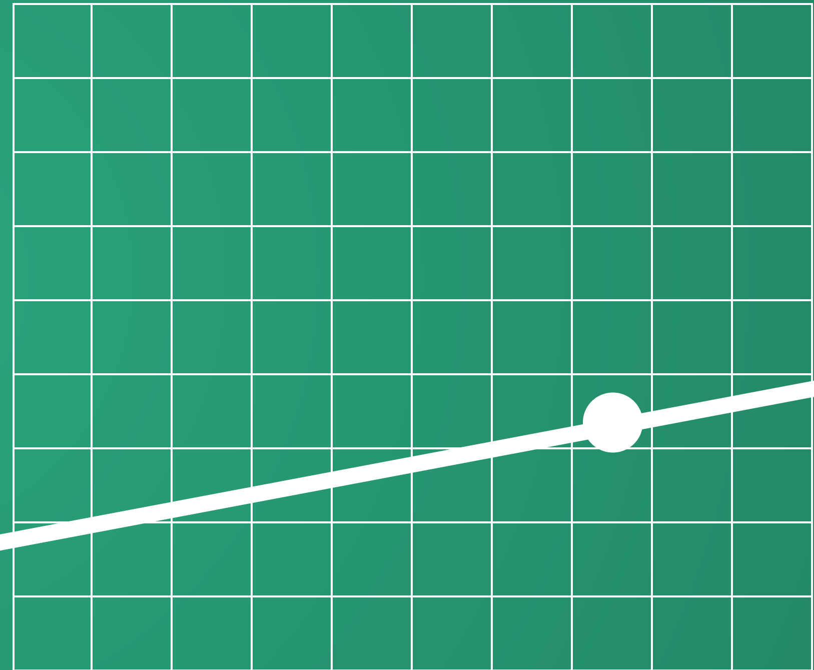
Known as backward raytracing

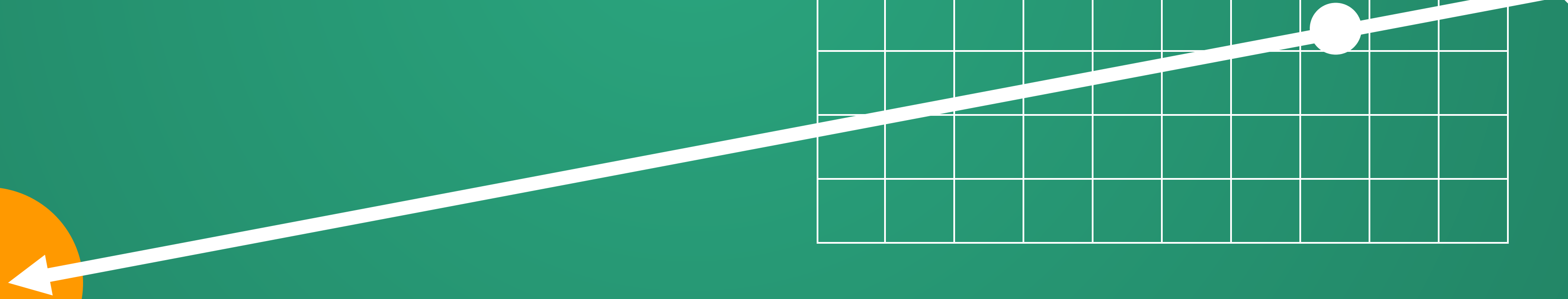
Instead of tracing photons from source (light) to sink (camera), we're going to shoot rays out from the camera.

Backward Raytracing

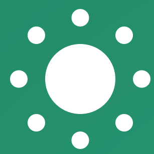
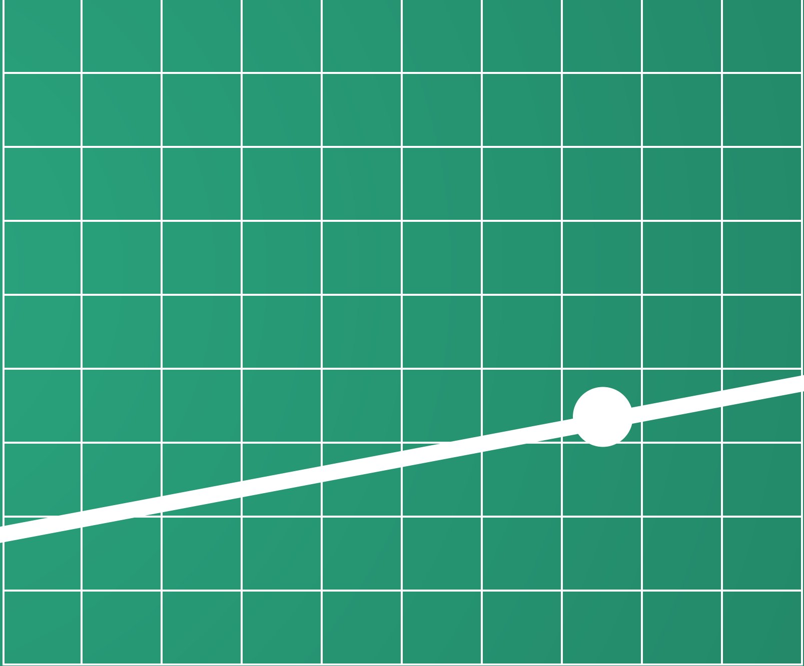


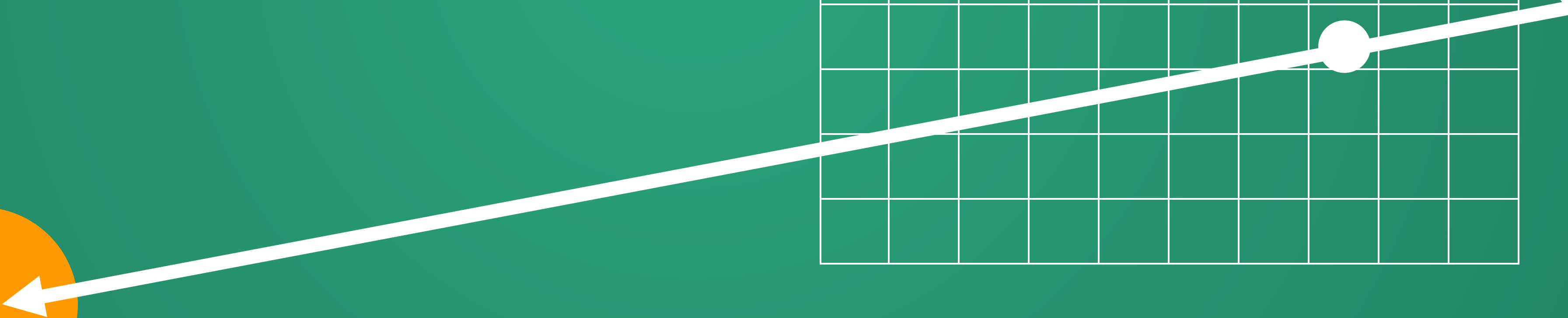
Can we see the ball?

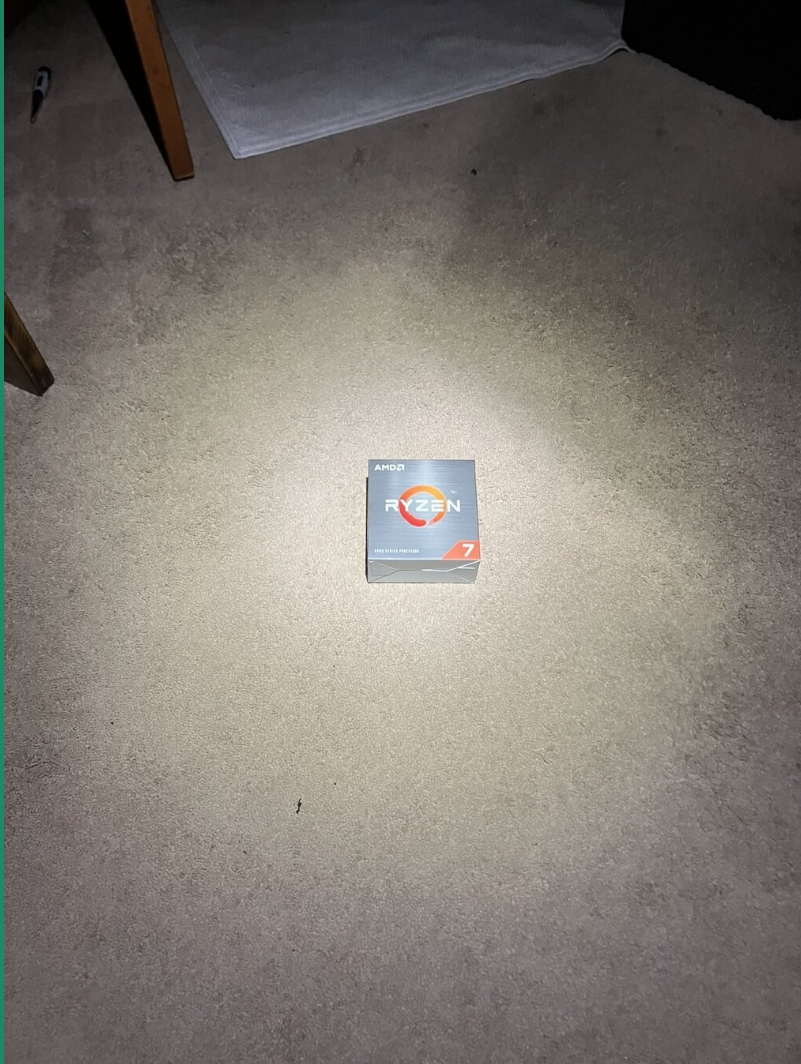


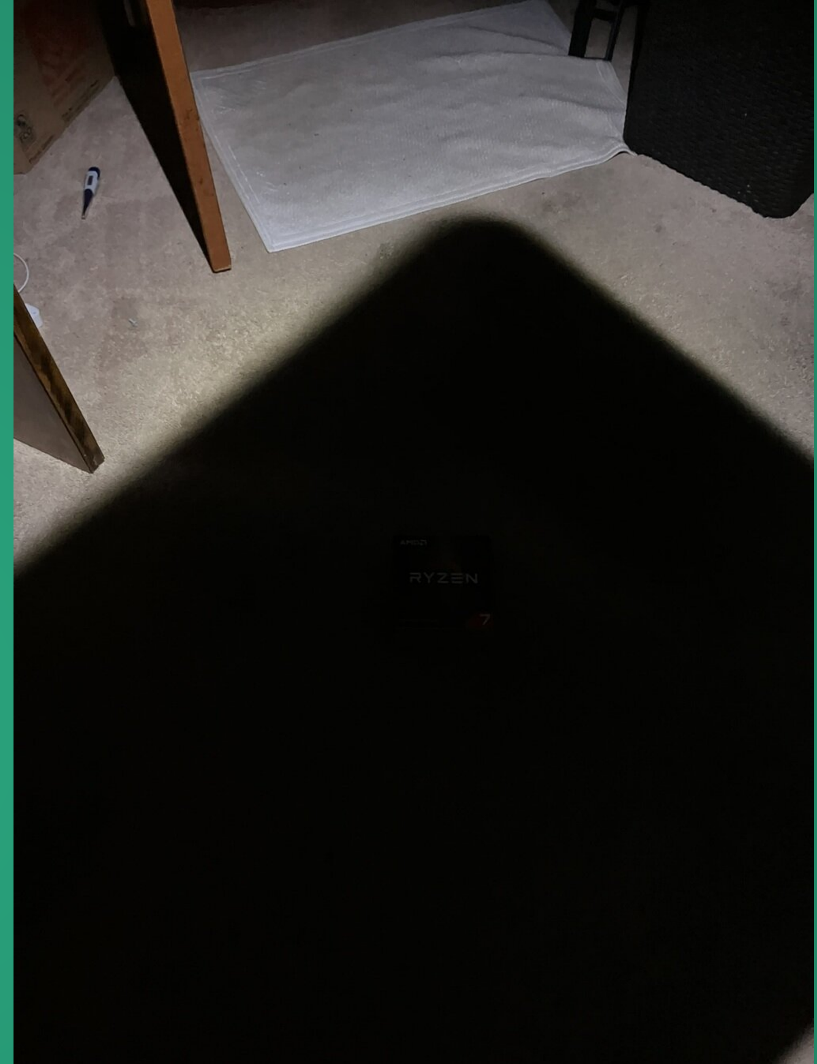


Can we see the ball now?

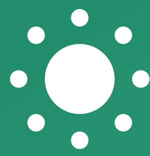





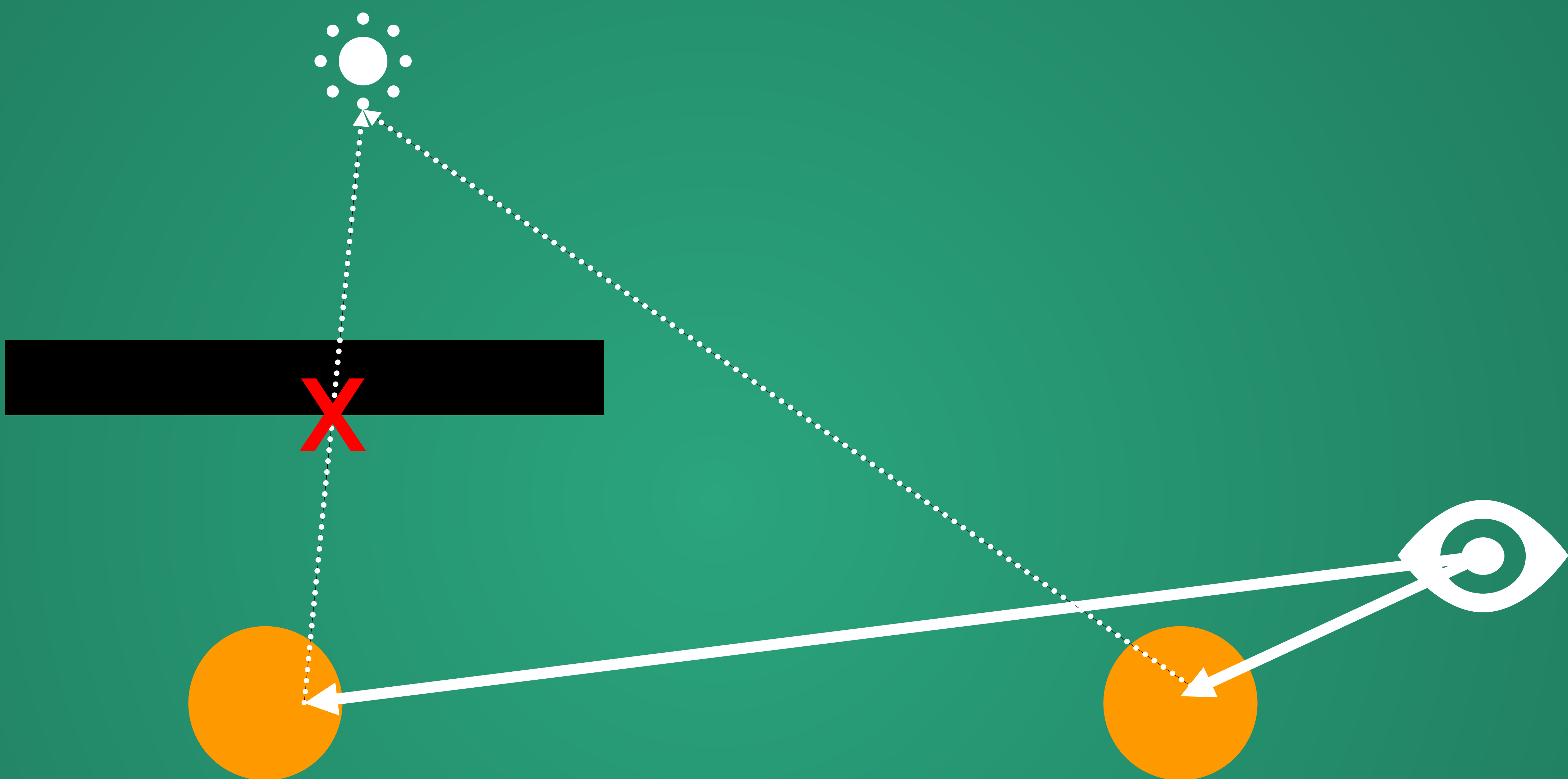




This is still with some ambient light reflection off the ceiling (which is white), walls (which are white), and floor (which is white carpet).



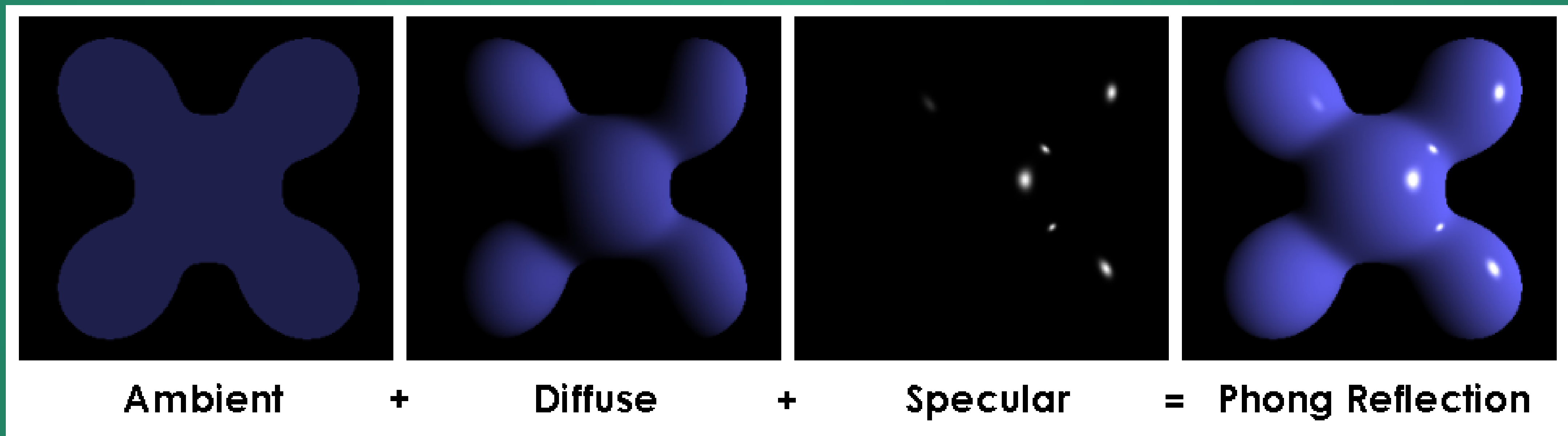
How do we model that one of these is visible, and one of these is not?

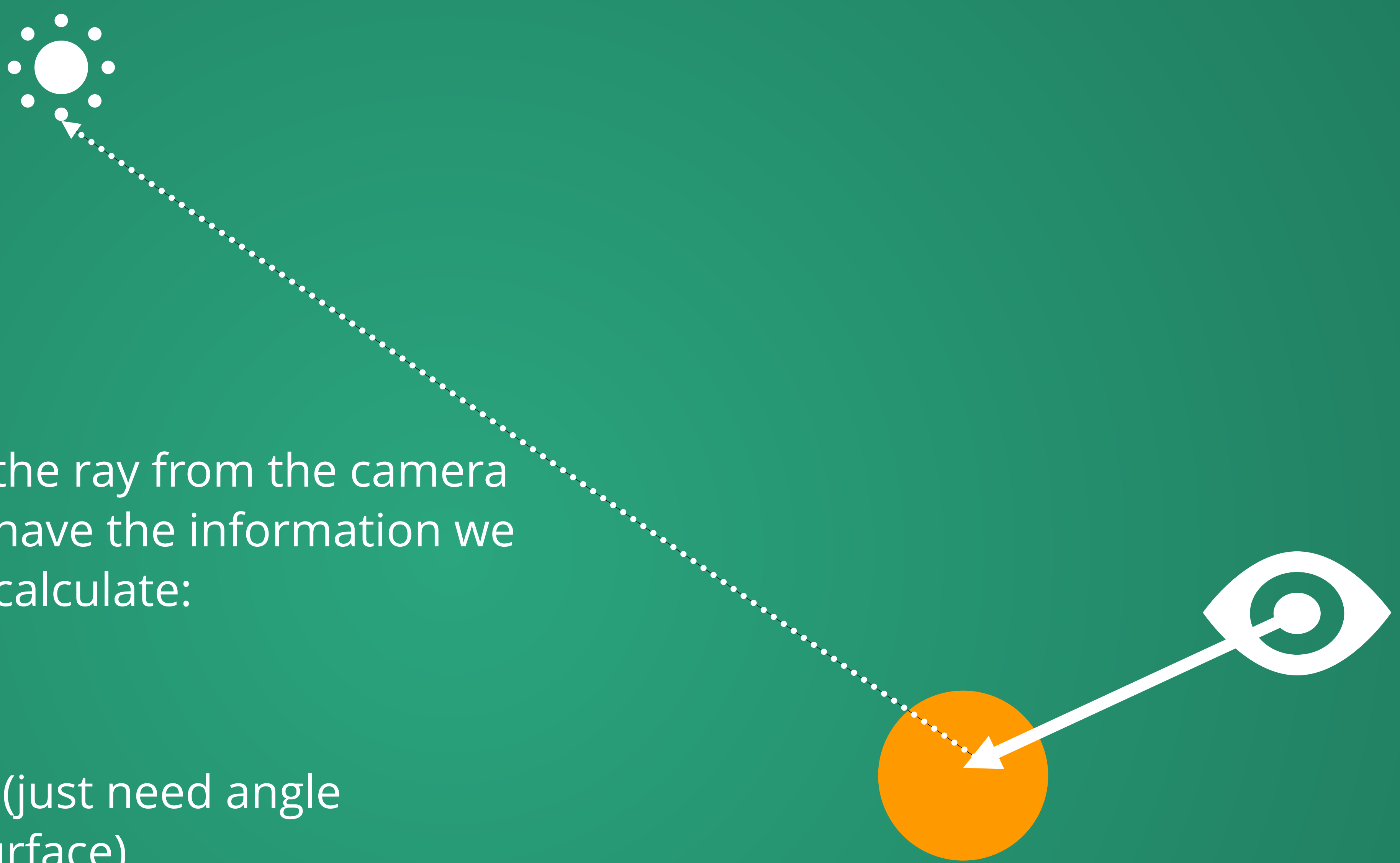


X

If there is no path from the light source to the surface of a diffuse object, there is no shading contribution.

Otherwise, we need to add some color. But how?



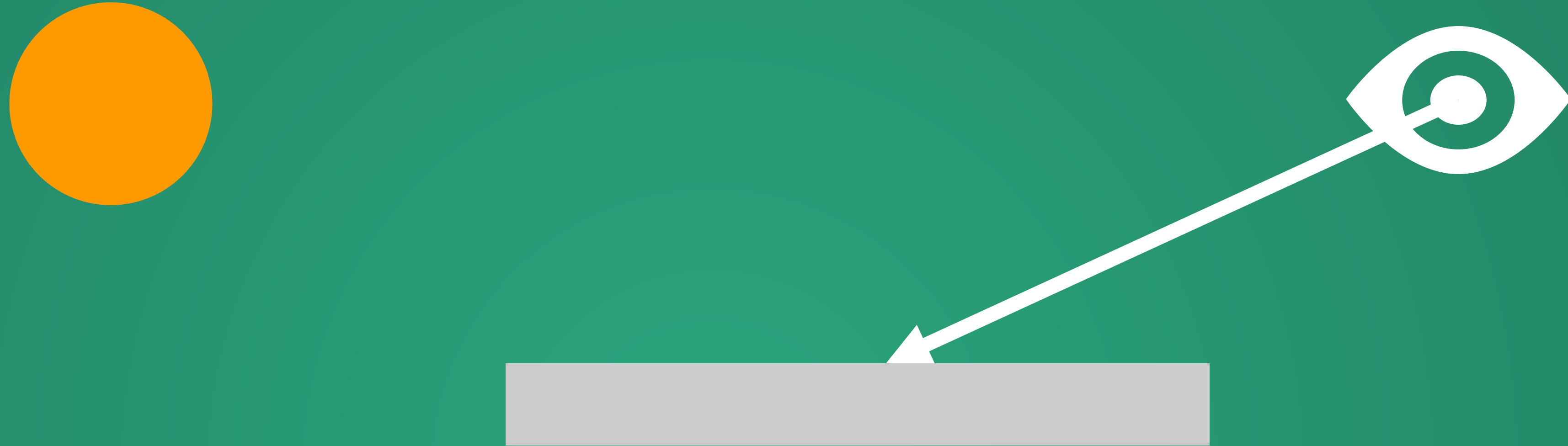


Because we have both the ray from the camera and ray to the light, we have the information we need to calculate:

1. Ambient Shading
2. Lambertian Shading (just need angle between light and surface)
3. Specular shading (need angles between light, surface, and camera)

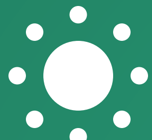
Non-Diffuse Objects

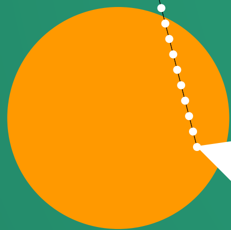
I have a mirror



When rendering using rasterization, we can get the info that these pixels correspond to a mirror (or reflective surface), but then we're stuck: we have no idea what that reflective surface is supposed to show!

How can we figure out what the mirror is supposed to show when raytracing?





Shoot a reflection ray off the mirror at the same angle as the incoming ray.

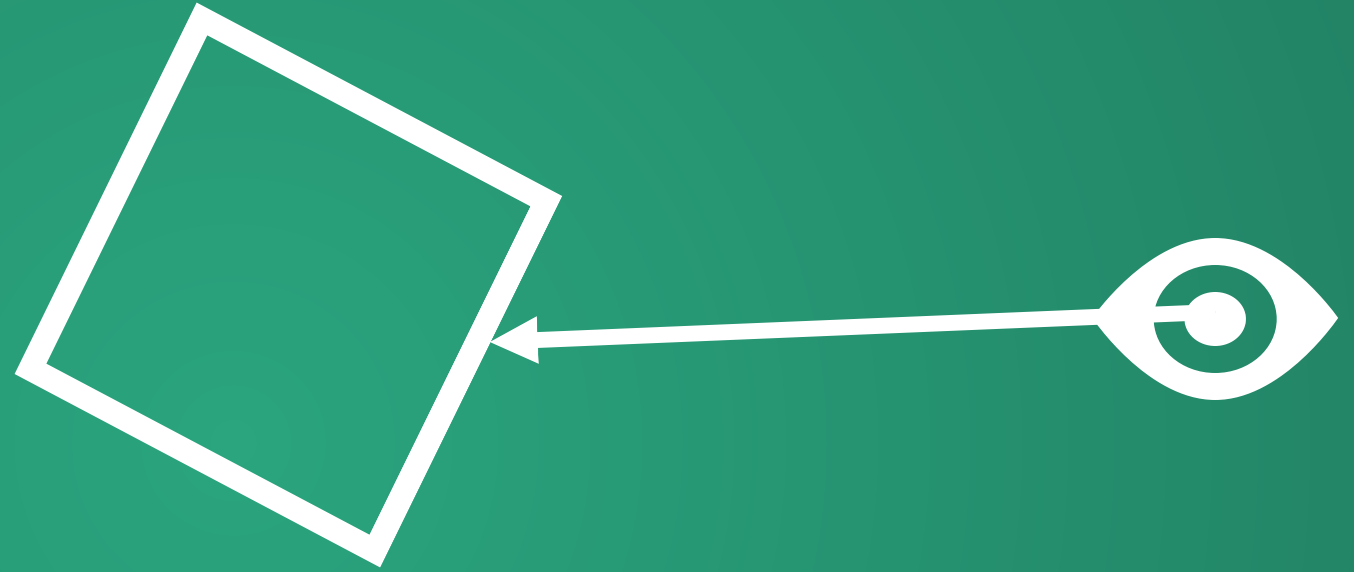
An example of *recursive* ray tracing.

Recursive Ray Tracing

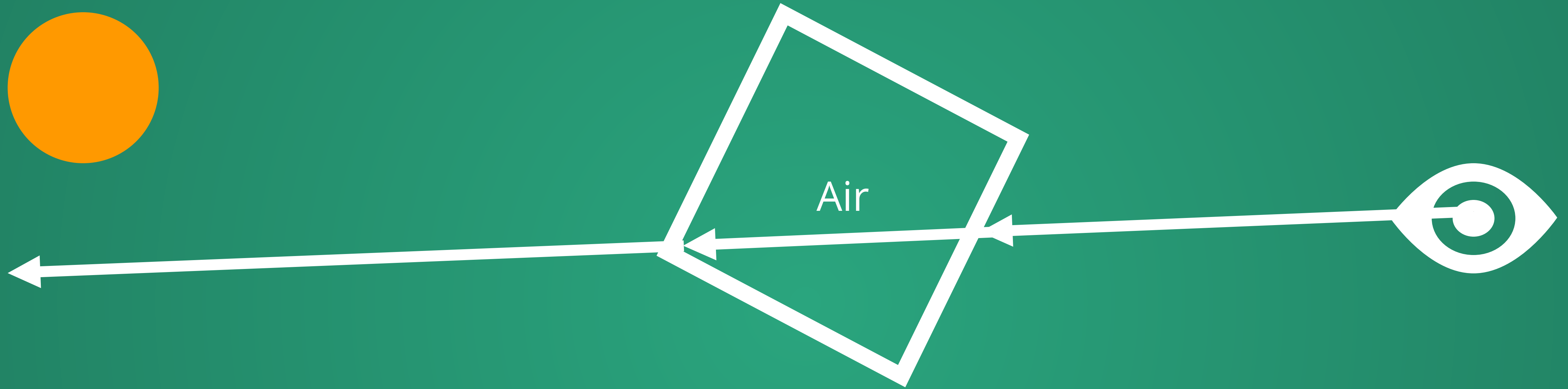
1. Our ray has hit an object
2. We're not quite sure what the color contribution should be
3. Shoot another ray according to some rules.
4. Continue until we do know what the color contribution will be

This will be the general framework by which we handle most of the advanced stuff!

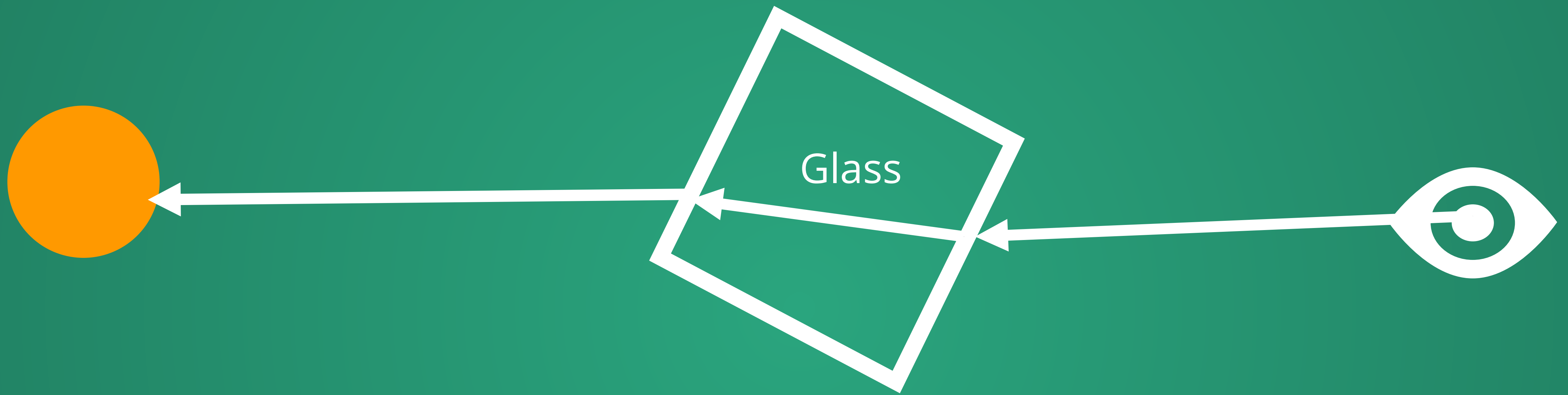
Transparent (Refractive) Object



Transparent (Refractive) Object



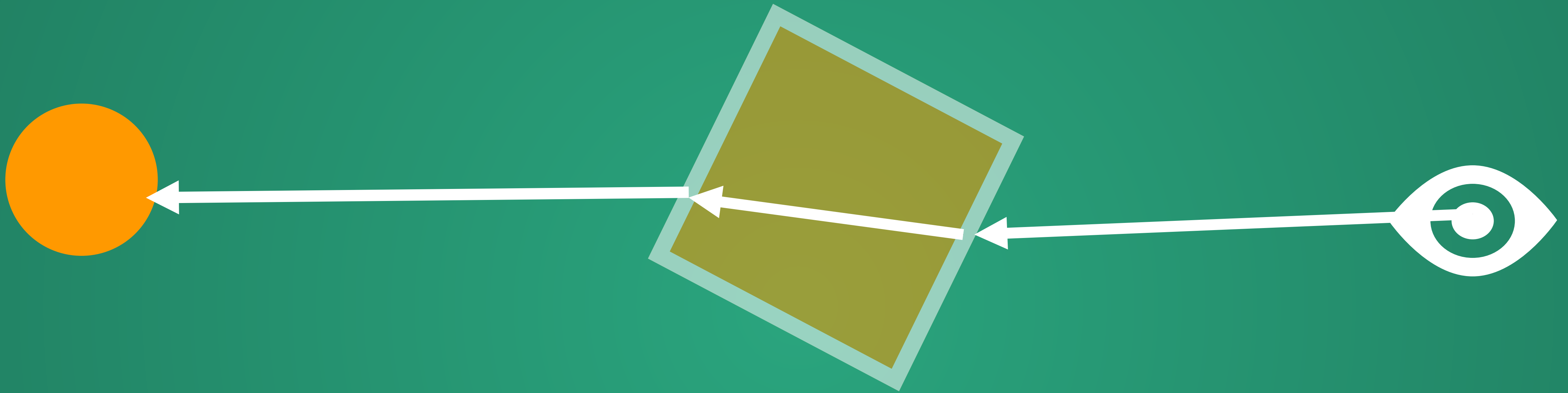
Transparent (Refractive) Object



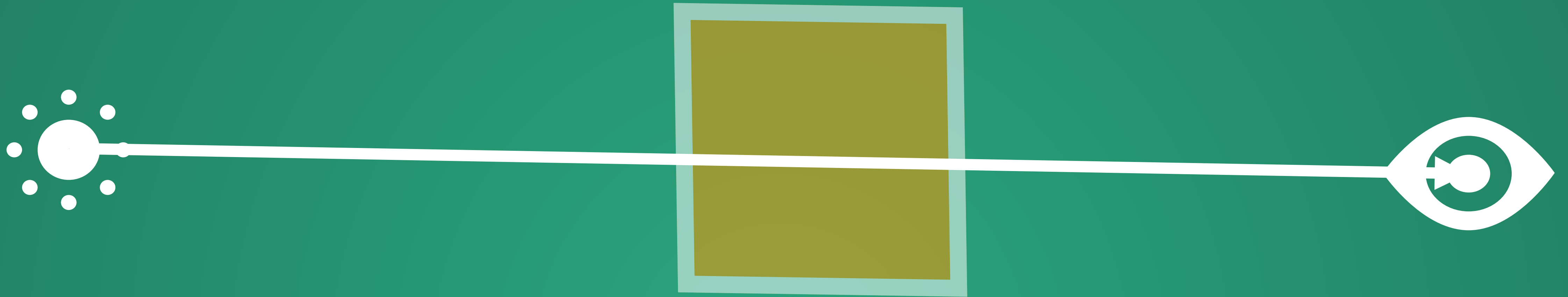
Amount of bend is controlled by Snell's Law of Refraction

$$\frac{\sin\theta_1}{\sin\theta_2} = \frac{n_2}{n_1}$$

Transparent Object with Tint

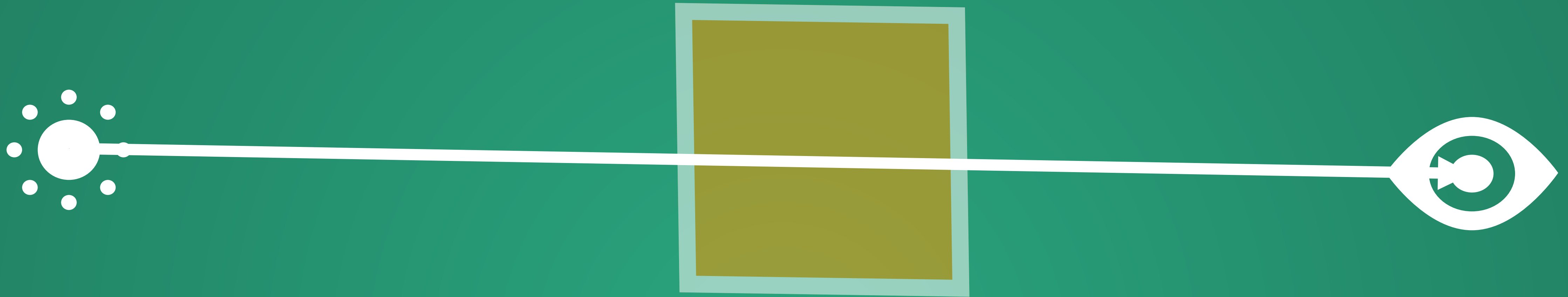


The ray that exits from this glass orange box will have some color attached to it, no matter what. But it still needs to get its color contributions from elsewhere. How do we compute this?



As the light passes through the glass cube, some part of its intensity is lost.

$$A = \varepsilon l c = \log_{10} \frac{I_0}{I}$$



After some rearranging, we find that some fraction of the color should come from the glass, and some fraction from the stuff beyond the glass.

The amount from beyond the glass falls off exponentially with how long we spend in the glass.

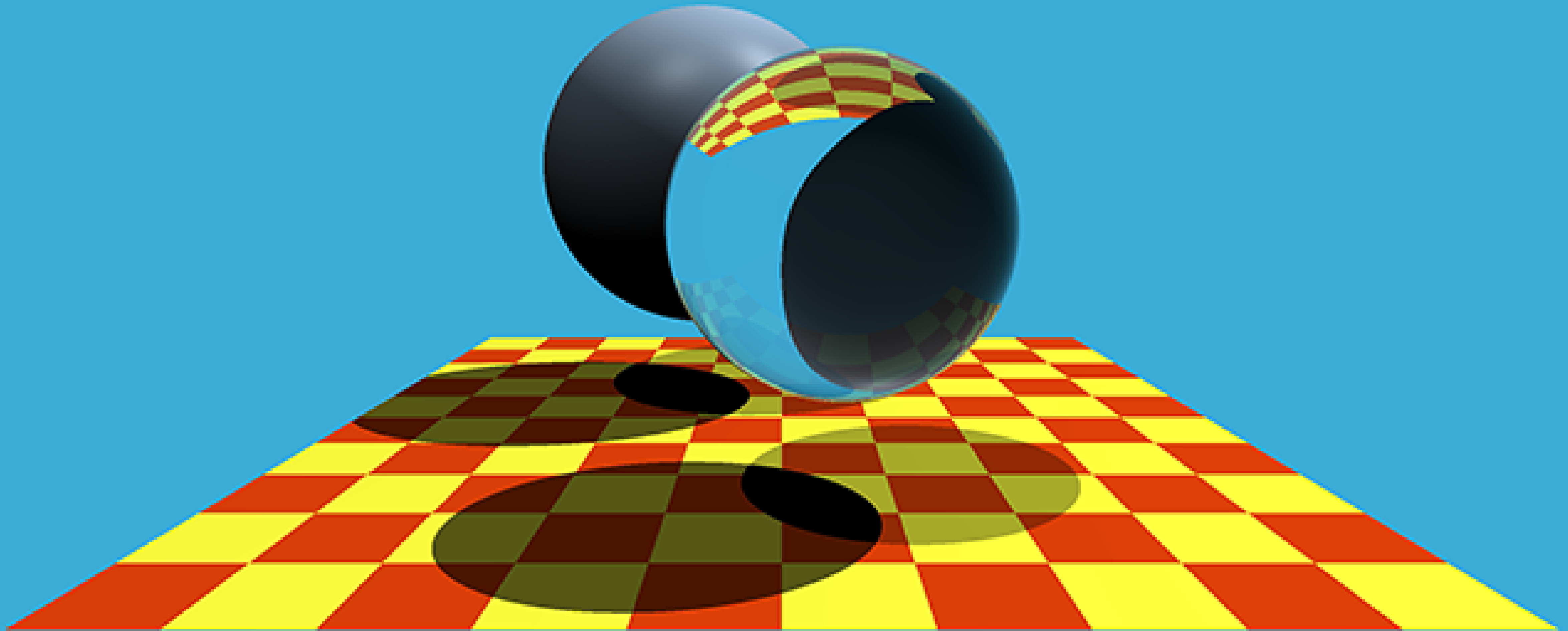
The Whitted Raytracer

These ideas form the core of what's called the Whitted Raytracer

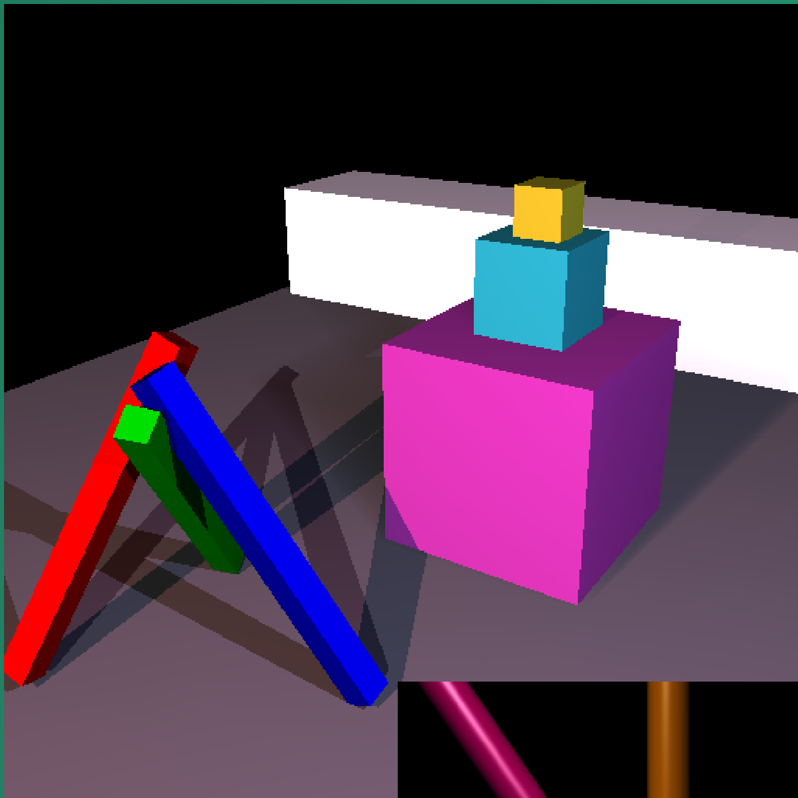
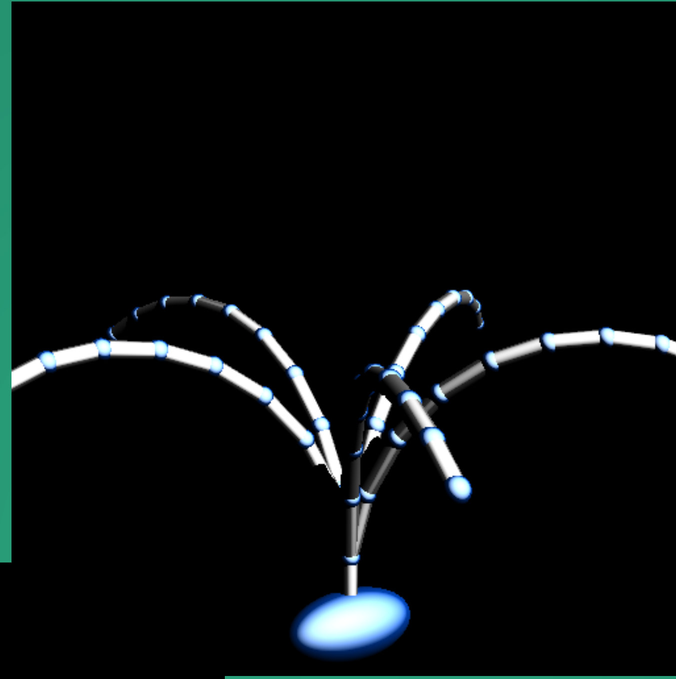
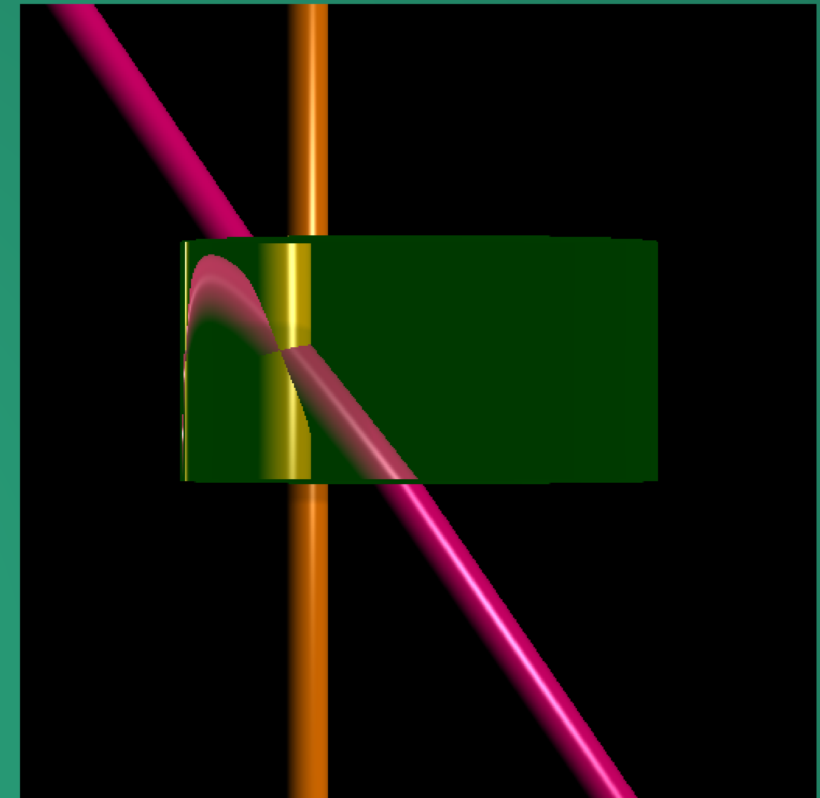
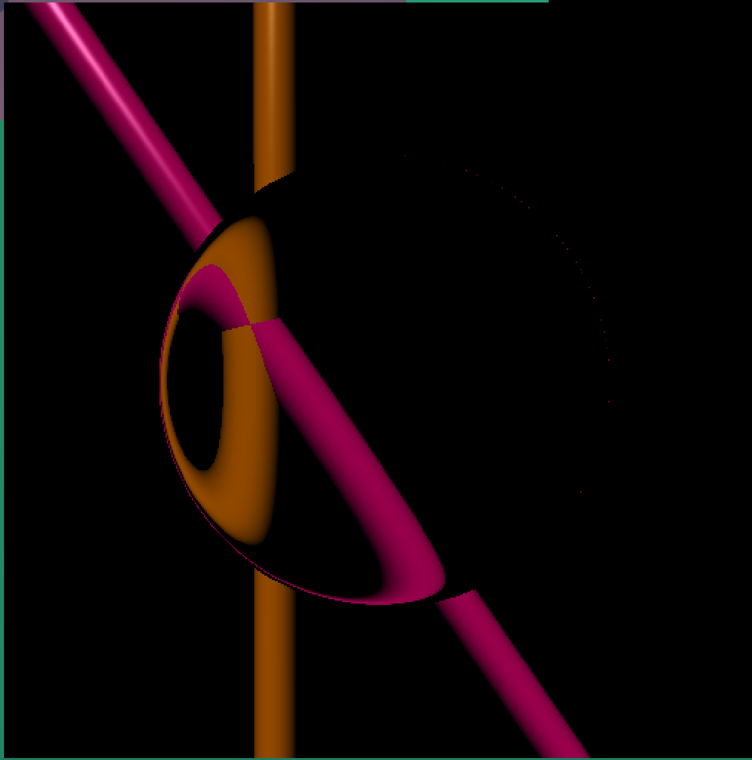
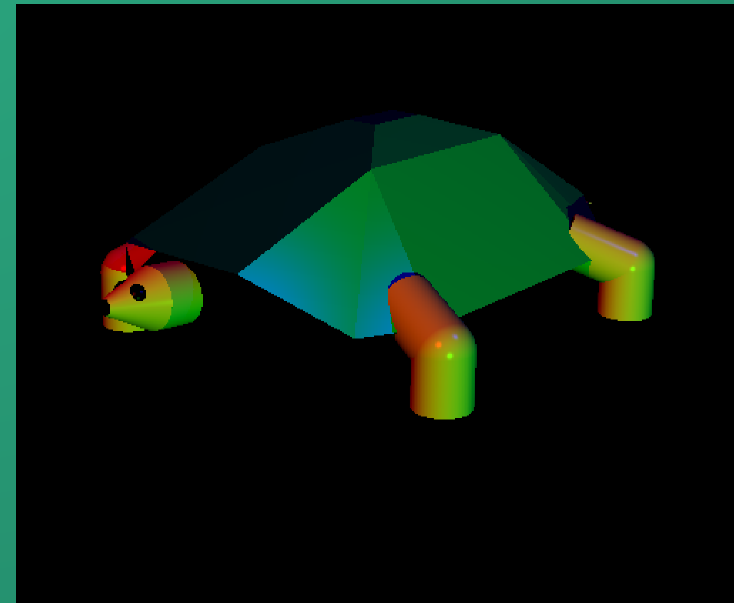
- Backwards tracing
- Phong model for opaque objects
- Recursive raytracing with appropriate rules for non-Lambertian surfaces
  - Snell's Law
  - Fresnel Equations
  - Beer-Lambert Law

Simple enough that it's sometimes as a weekender project
to get familiar with a new language!

Results



Results

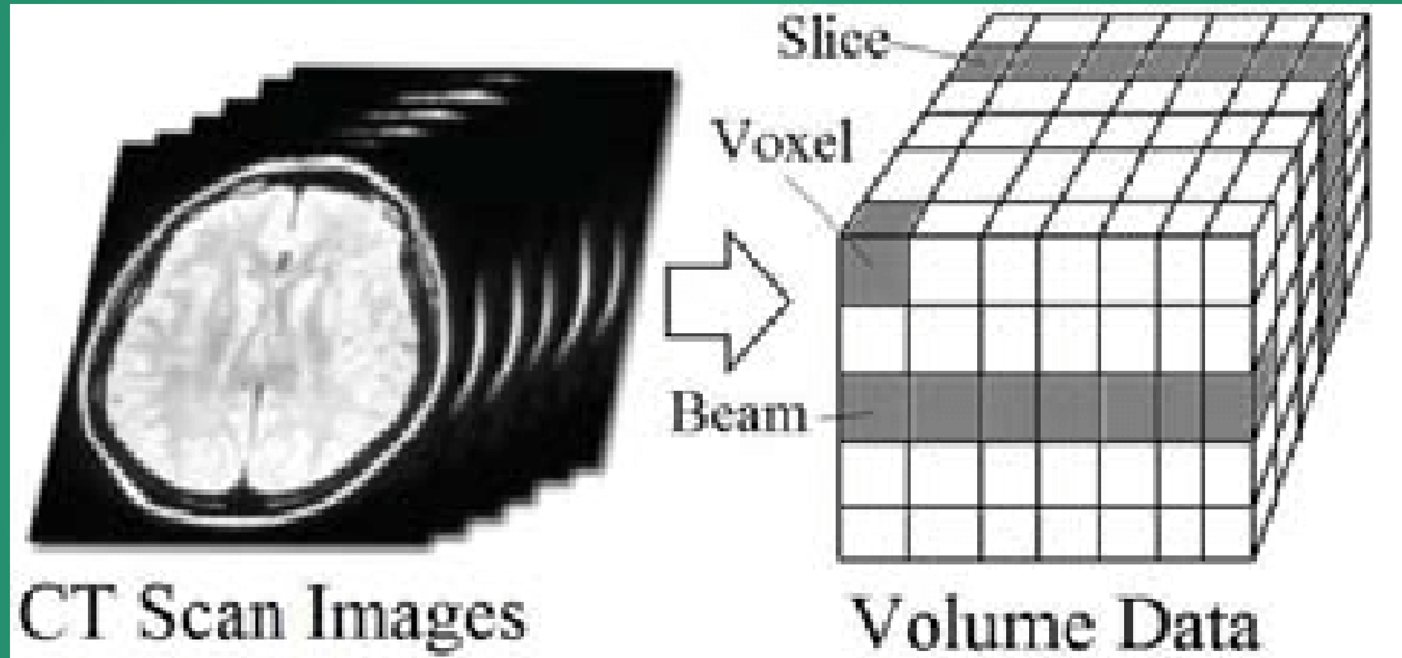






Direct Volume Rendering

An application of raytracing to
data visualization

X Ray Imaging

A CT scan consists of "voxels" stacked into a 3D grid

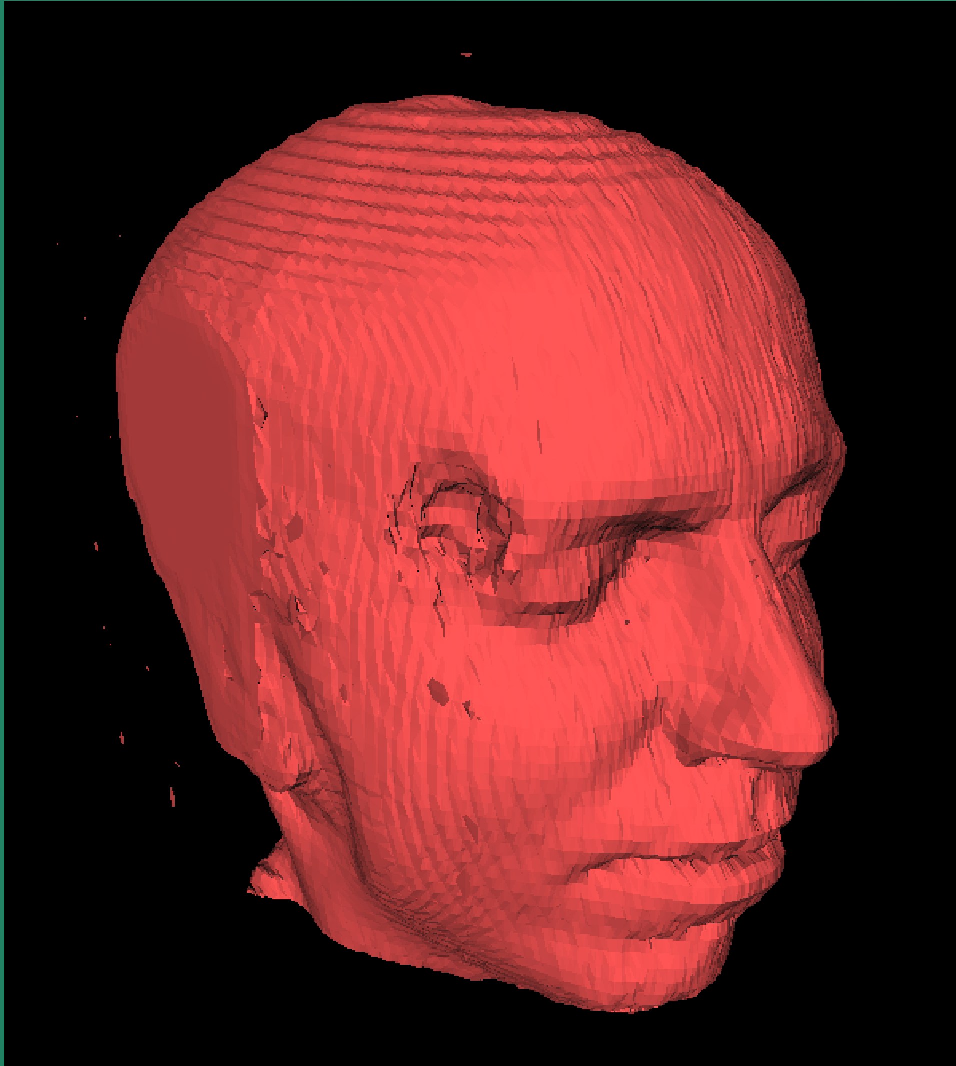


You can think of these as stacks of images: the top layer is an image, then there's a second image just below that, and another just below that, etc.

How do we visualize this data?

Option 1: As 2D slices. Rely on a UI to move slices up and down and rely on doctor to be able to reconstruct the scene in their head.

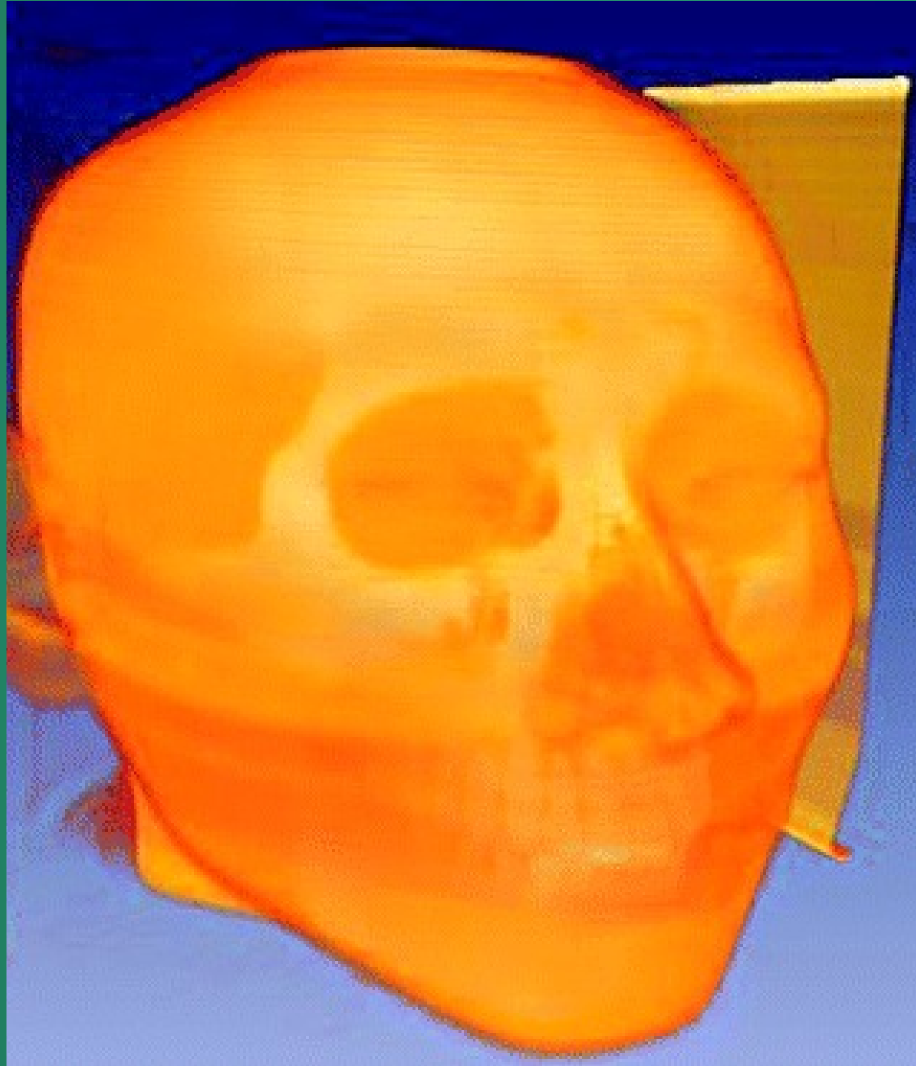
How do we visualize this data?

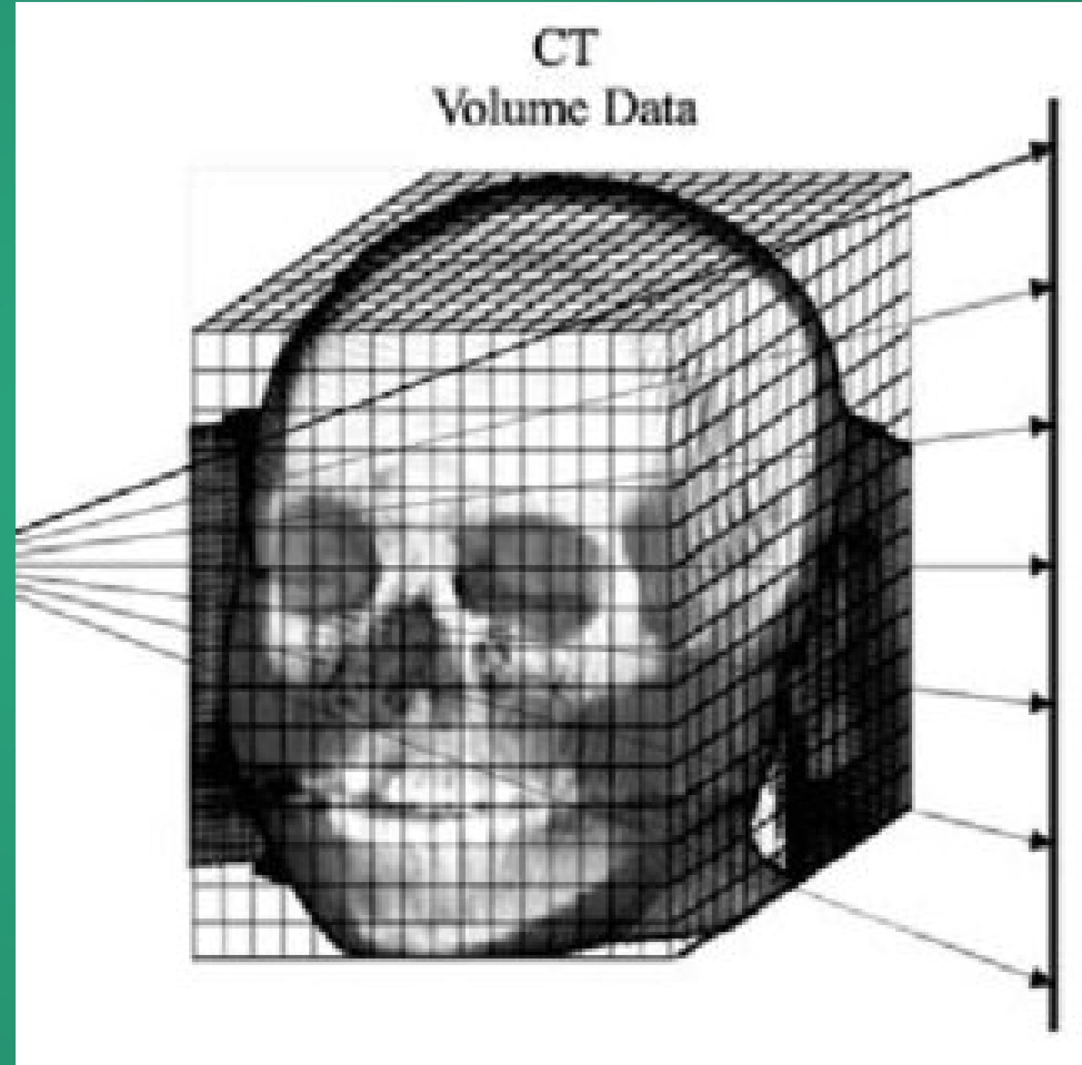


Option 2: As a surface. Rely on algorithms like marching cubes to generate a surface (triangles) from the voxelized data, then show this.

https://en.wikipedia.org/wiki/Marching_cubes

How do we visualize this data?

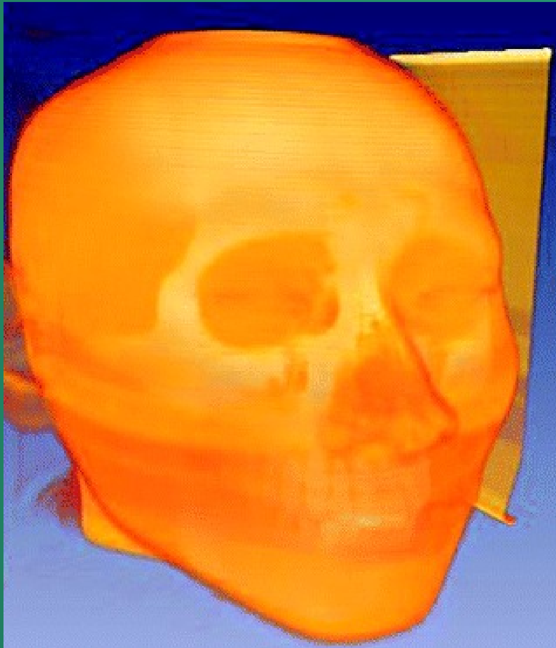


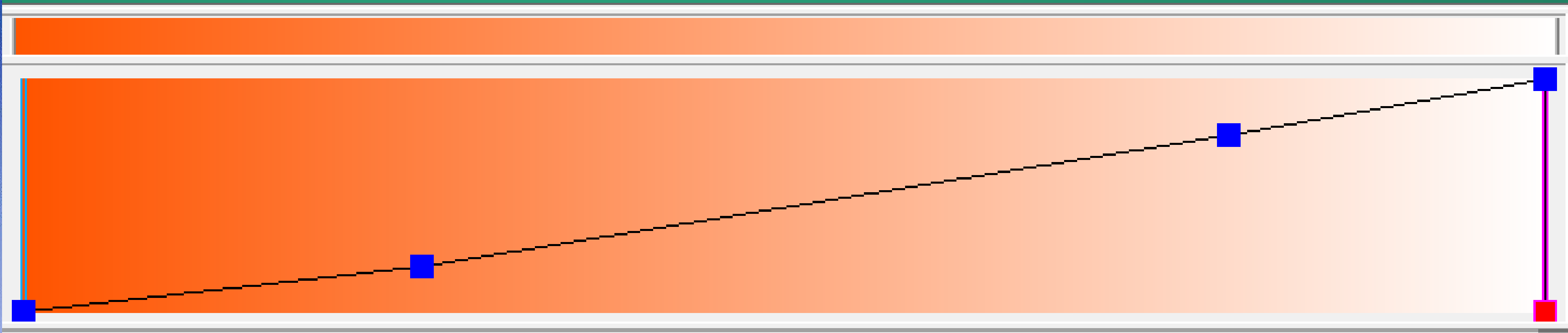


How do we visualize this data?

Specify a *transfer function* which gives each intensity of the source voxels a color and a transparency. Example: low-intensity voxels are orange and high transparency, high-intensity voxels are white and low-transparency.

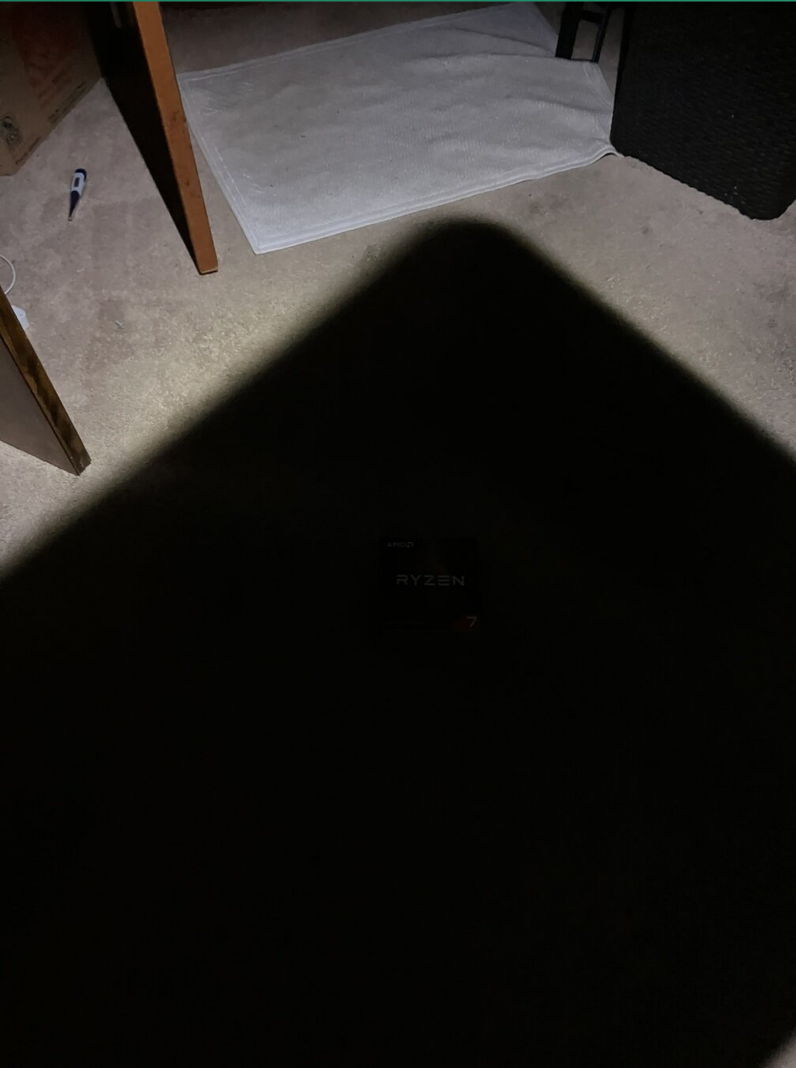
Then raytrace the voxels as if they're cubes with the color/transparency values set by the transfer function.





Path Tracing and Global Illumination

Why can we still see the box?

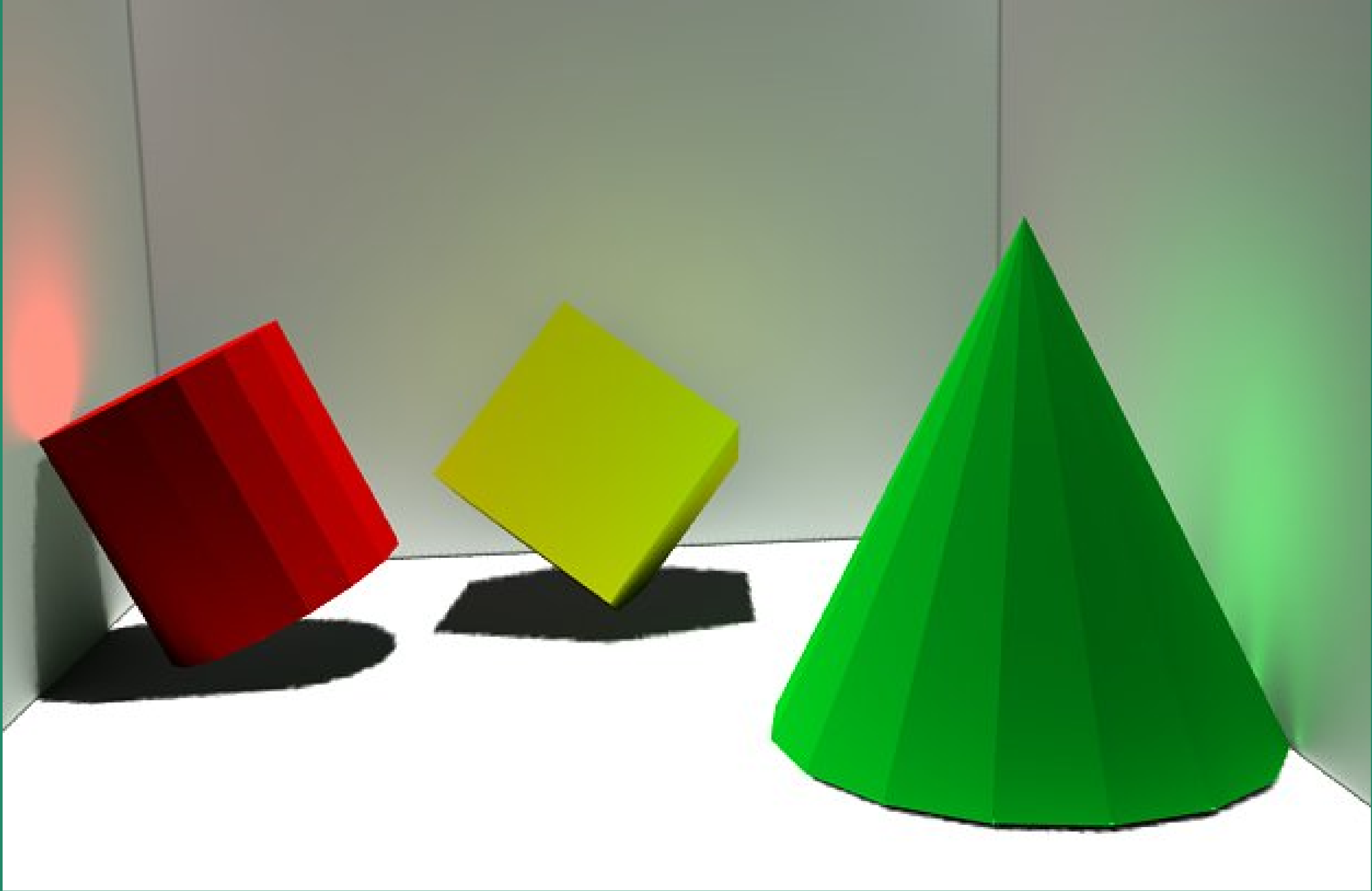


Light doesn't necessarily have to travel in once bounce

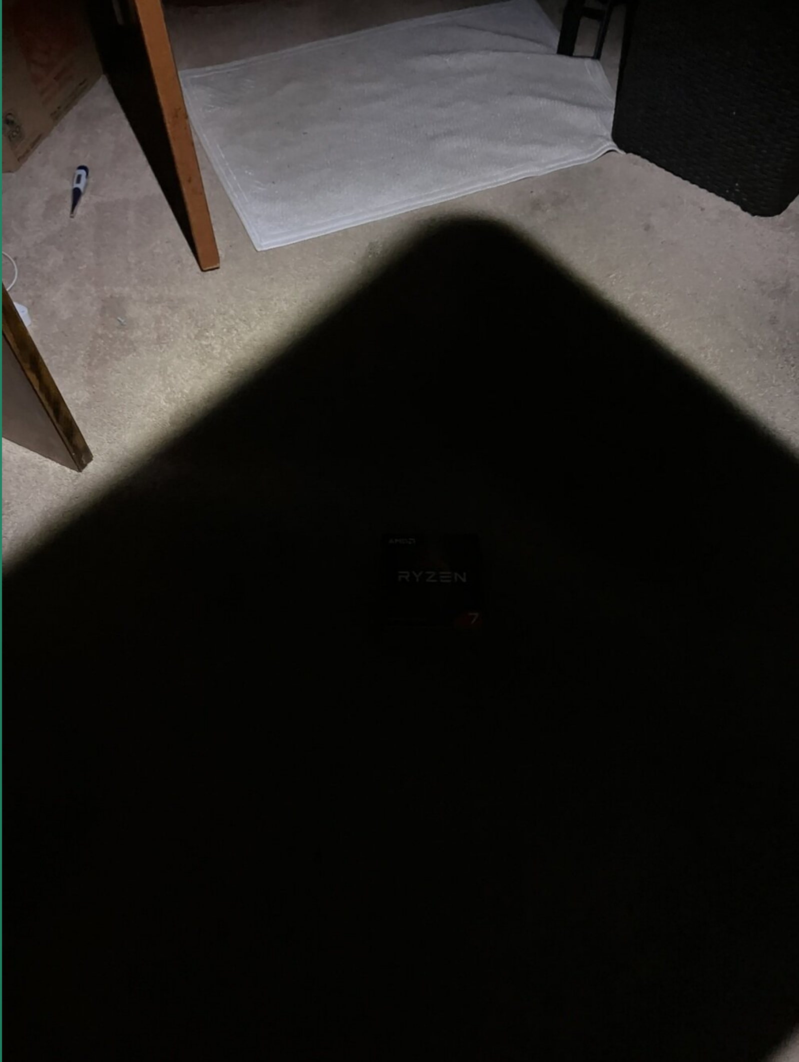
Known as Global Illumination

Ambient Occlusion

Color Bleed



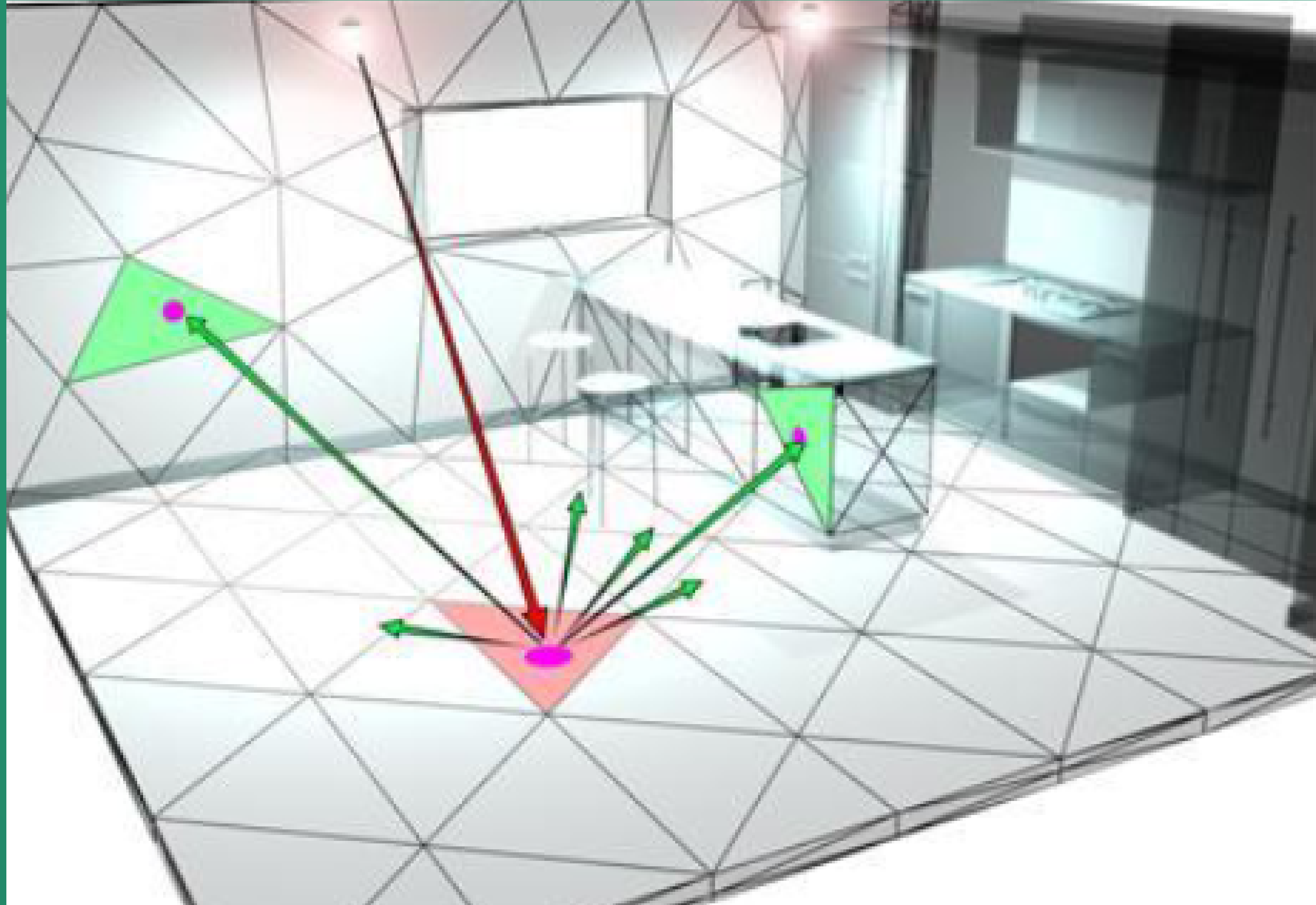
Things visible in shadows

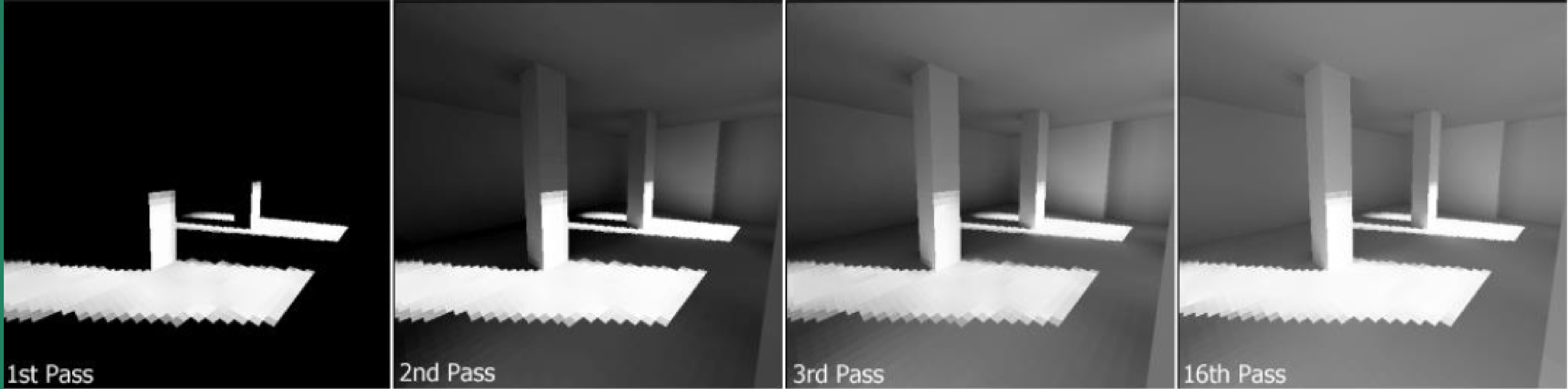


Subsurface Scattering

So how do we get global illumination?

Radiosity





Propogate light around the scene until we reach a steady state, i.e. doing more iterations doesn't change the scene much. Then, raytrace the result.

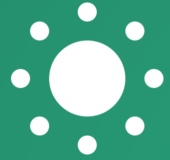
Reasonable speed if everything is diffuse, but cannot handle reflective or transparent materials.

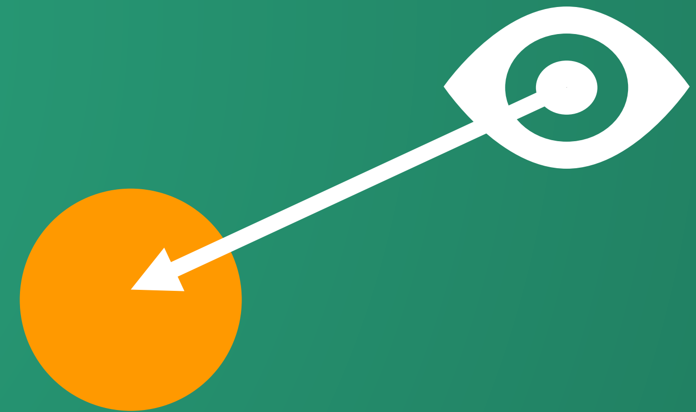
Example: Radiosity in Battlefield 3

How do we get something more general?

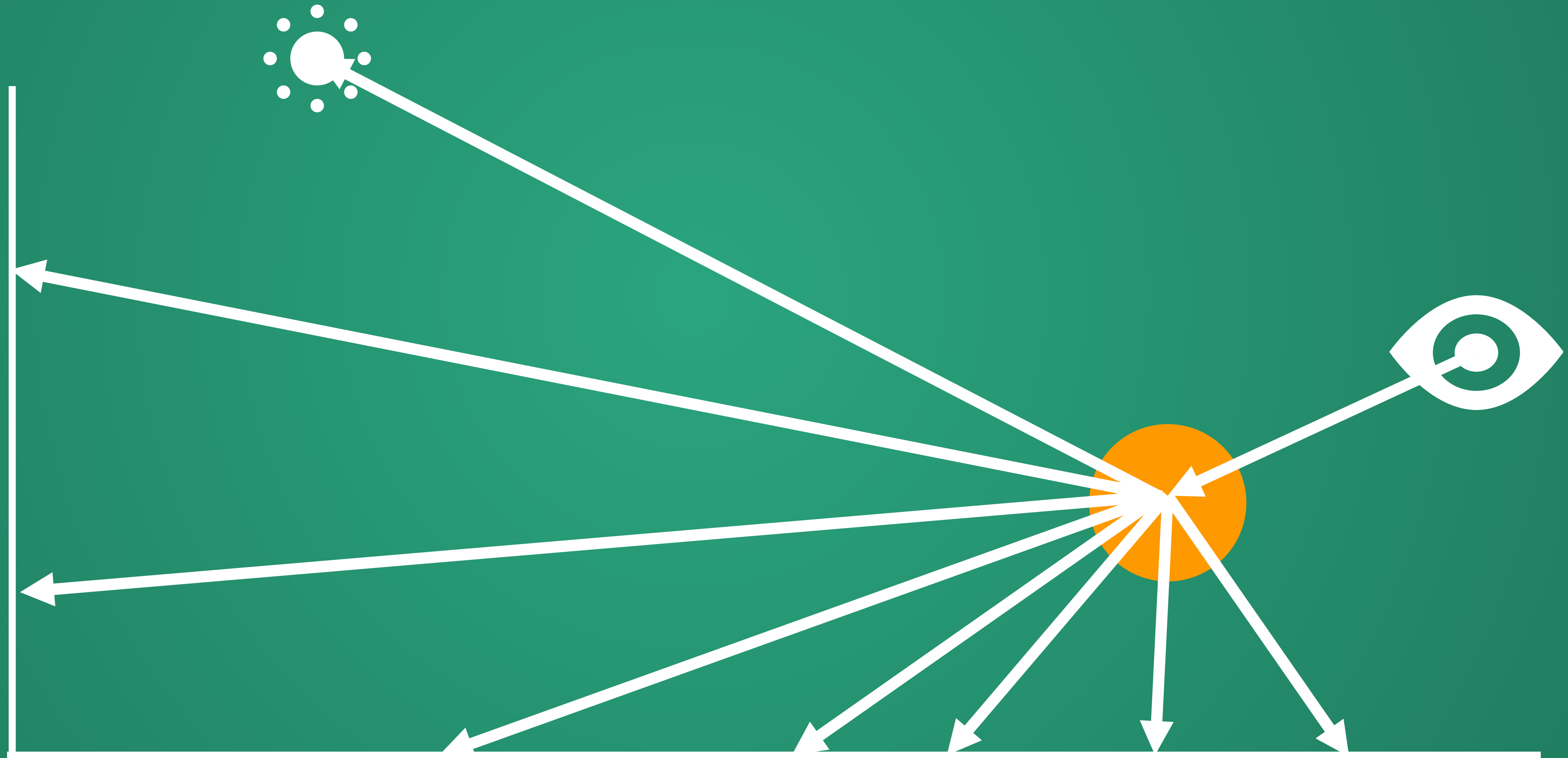
http://blog.wolfire.com/2011/03/GDC-session-summary-Battlefield-3-Radiosity

Where did this light come from?

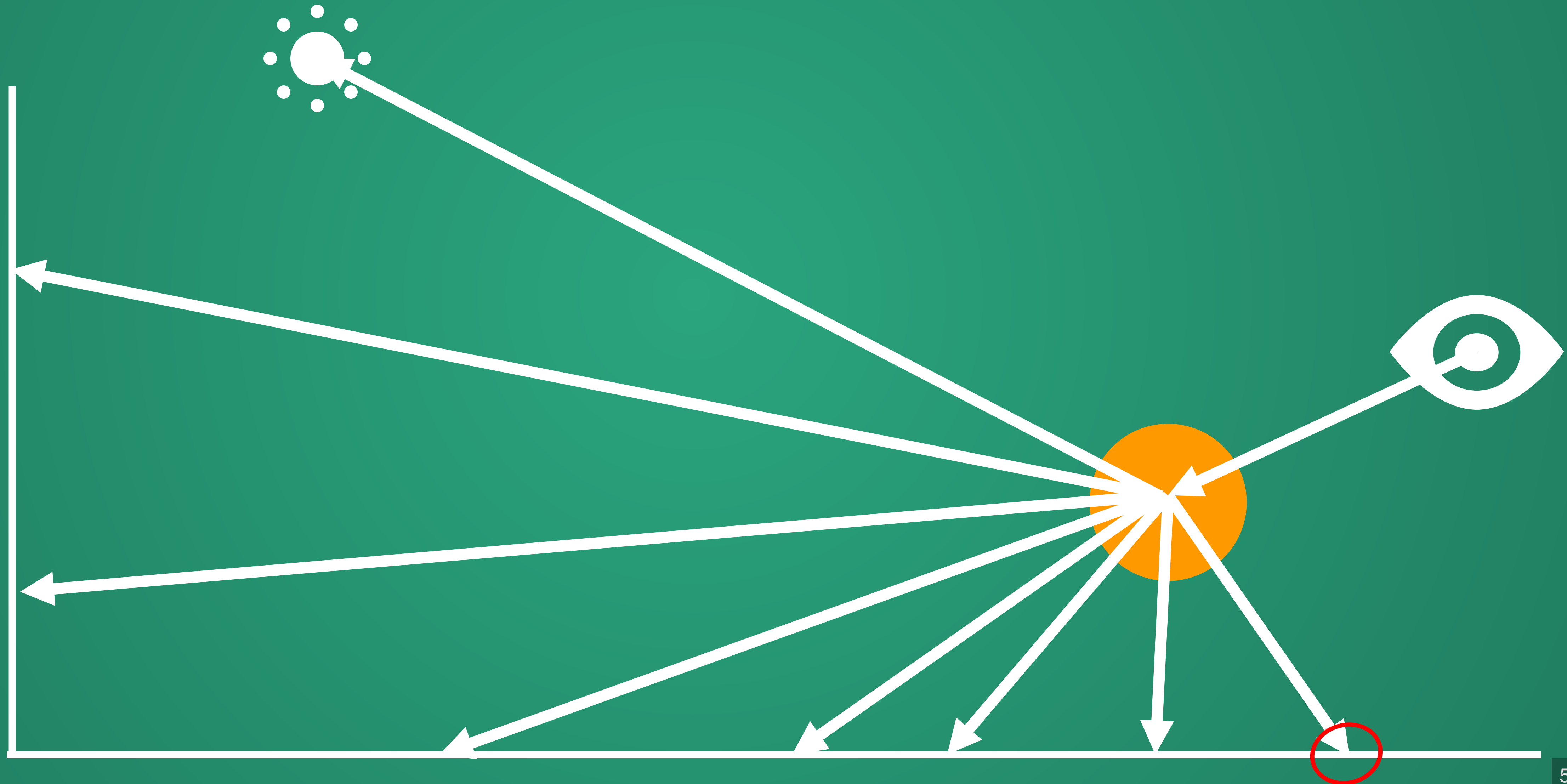




Where did this light come from?

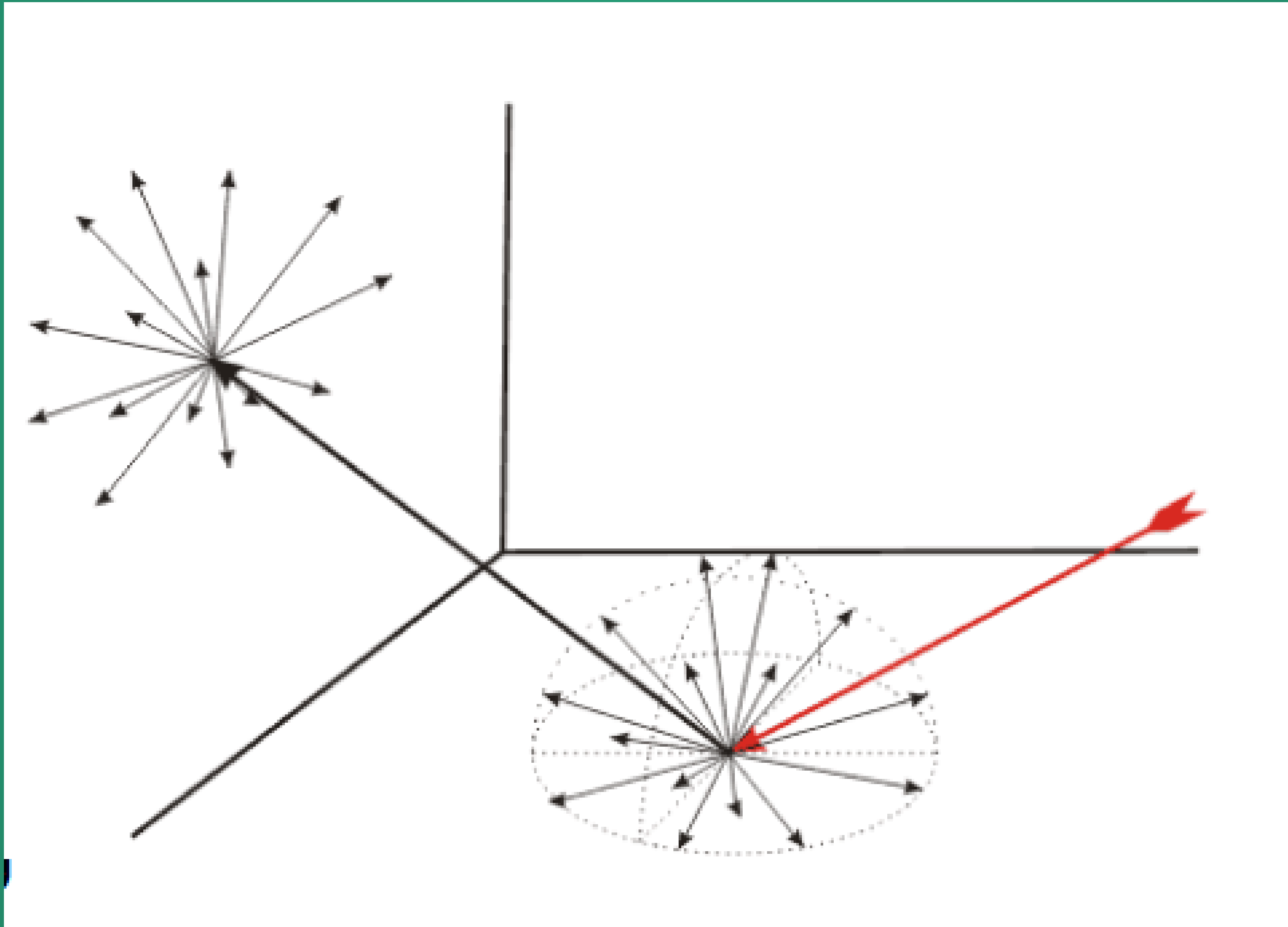


Where did this light come from?



Where did this light come from?

Whenever we get a ray-geometry intersection, fire off additional rays from that intersection to see where that light came from



Path Tracing

Like Whitted raytracer, but at every intersection, fire off additional rays and see what *their* shading contribution is.

Stop once we're a certain number of levels deep.

Suppose we do 20 recursive rays per collision and stop once we're 5 levels deep. How many rays do we need to shoot per pixel?

3.2 million rays. More rays than you'd need to raytrace an entire image in a Whitted raytracer!

Global Illumination is *gorgeous*.

But path tracing has issues with noise.

We can mostly get rid of the noise if we use thousands of rays per sampling but...

$$2048^5 \approx 3.6028797 \times 10^{16}$$

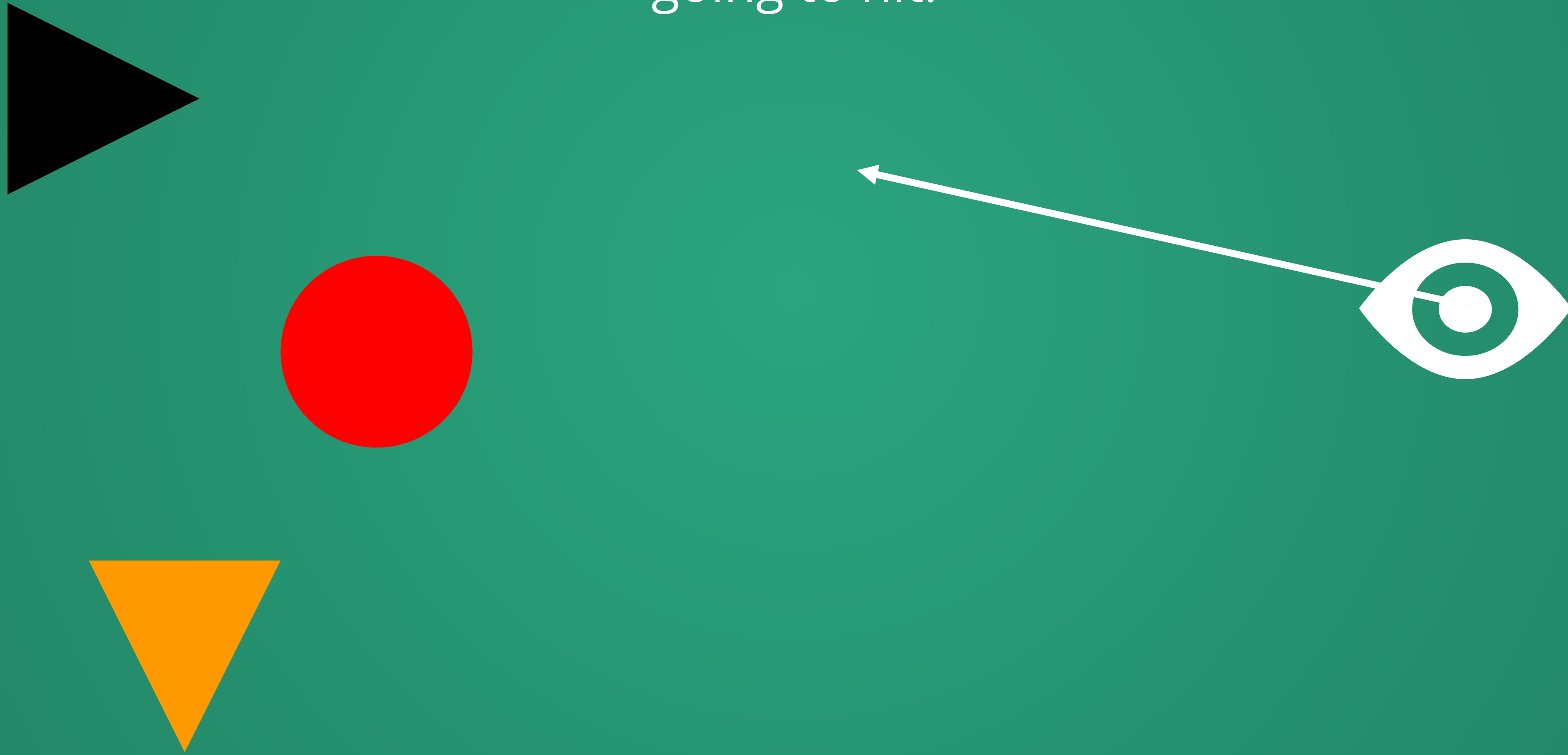
The time cost becomes intolerable.

Modern "Raytracing"

What do we need to be able to do to get real-time raytraced animation?

Problem 1: Collisions are slow

When we look at a ray in a scene, we intuitively know what it's going to hit.



But computationally, we have to check every single object.

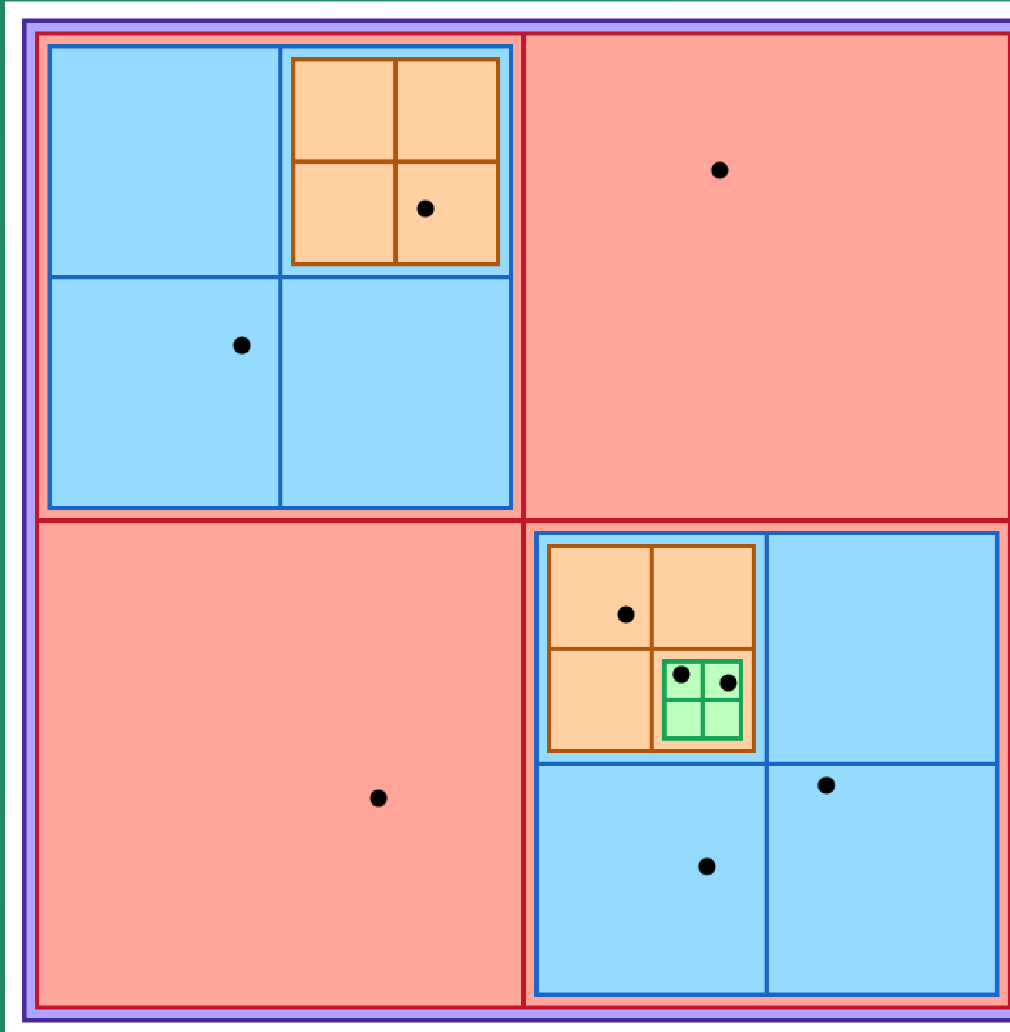
Let's say we have a scene with 1000 objects (very mild)

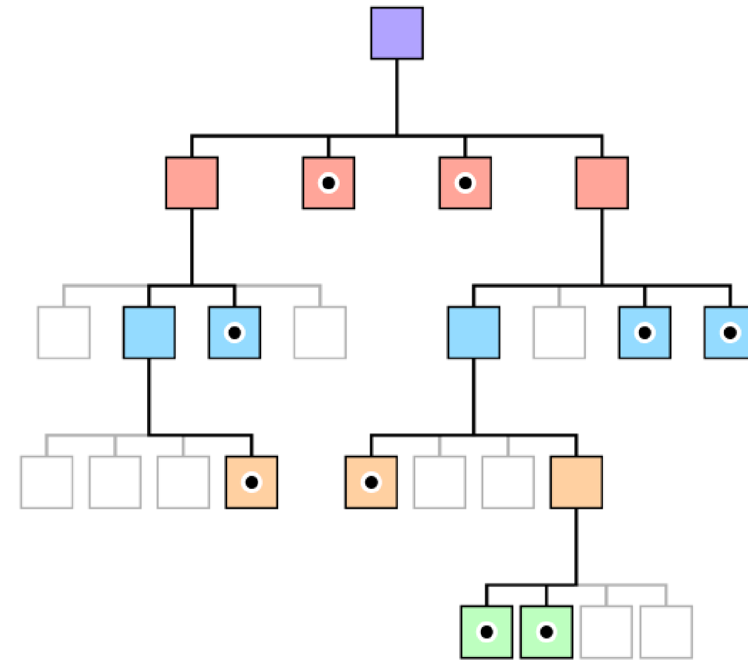
We're going to shoot 10 recursive rays when pathtracing, to a depth of 5. This is *very* minimal and will certainly result in tons of noise in the output image.

How many ray-object intersections do we need to do to trace a single pixel?

100,000 rays in total. Each needs to be checked against 1000 objects. 10^8 (100 million) ray-object checks to trace a single pixel.

Solution: Spatial Acceleration Structures!





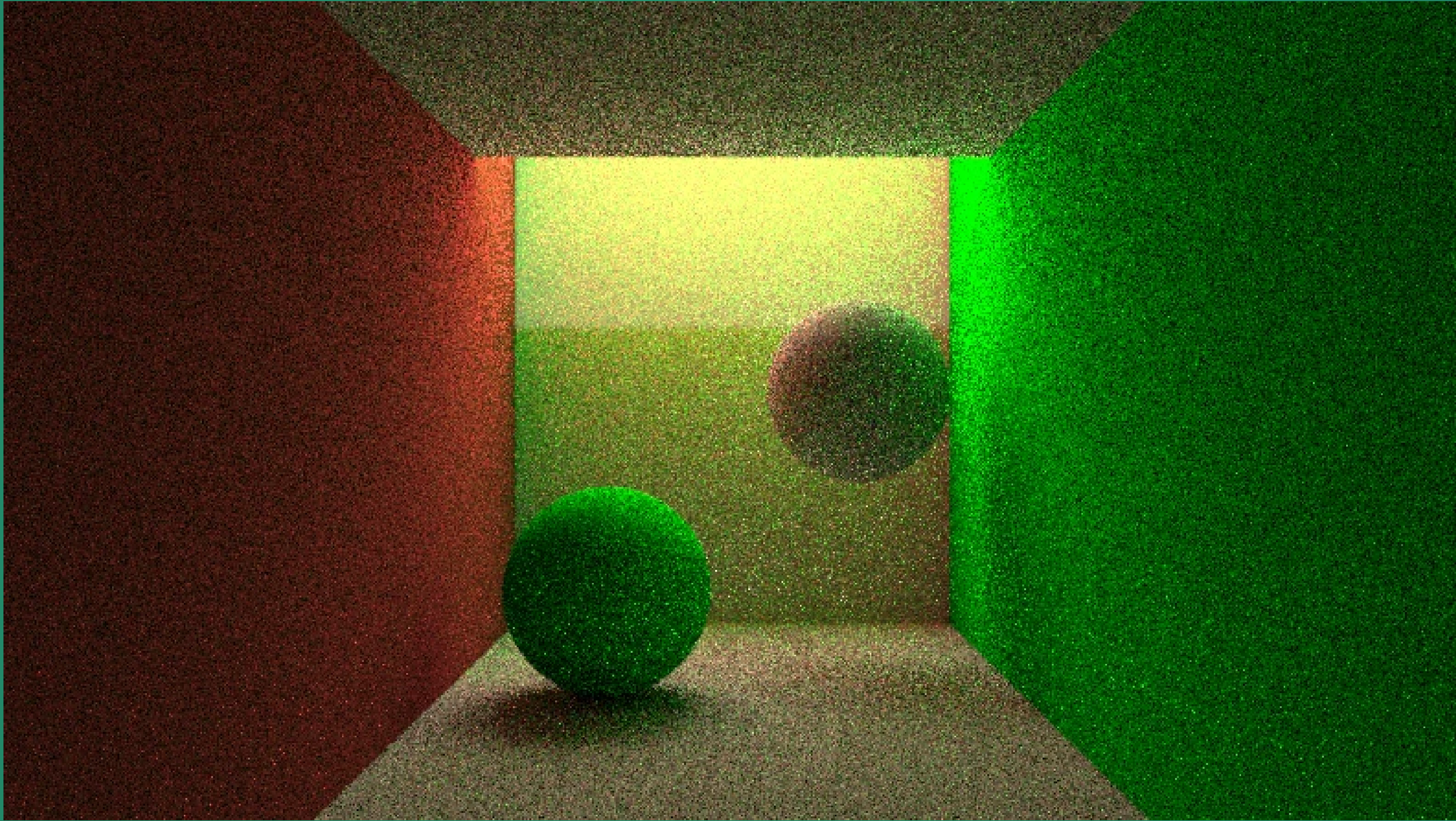
How do we find an object in the tree?

```
1 Object acceleratedRayIntersection(Ray ray, TreeNode node){
2   if (ray.intersectsWith(node)){
3     for(child in node.children){
4       acceleratedRayIntersection(ray, child);
5     }
6   }
7
8   // Do something to gather all the intersected objects here
9 }
```

How many branches?

Traversing trees inherently requires a lot of branches

Problem 2: Noise

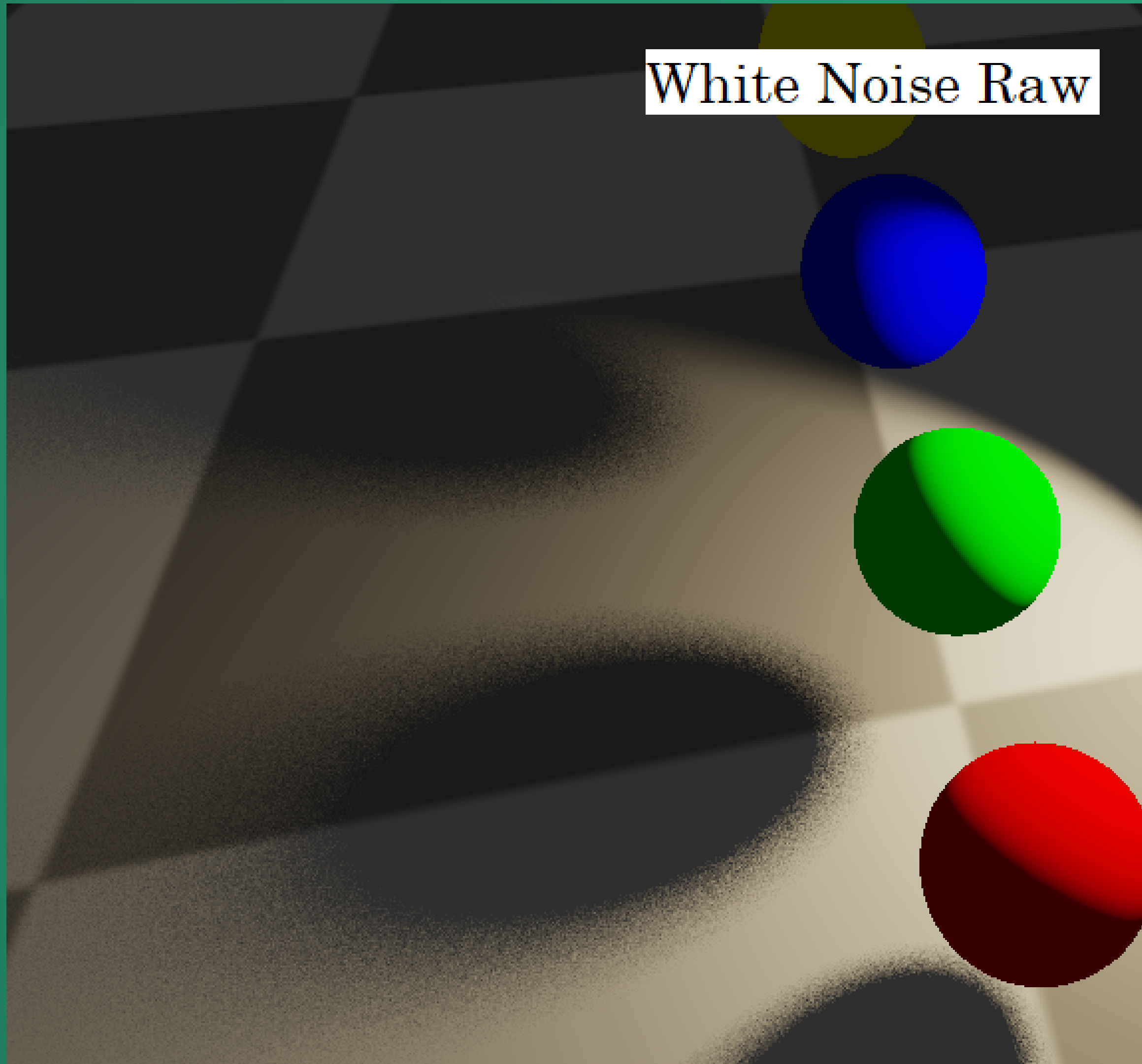


Who wants to watch a movie that looks like this?

Image Filtering

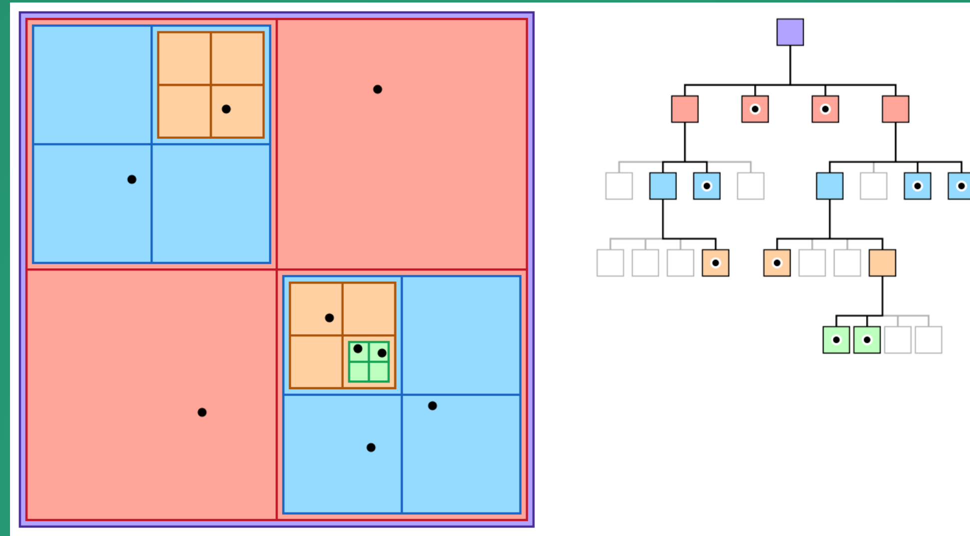
Apply an *image filter* to the image which attempts to remove noise.

Note: almost always a convolutional image kernel!



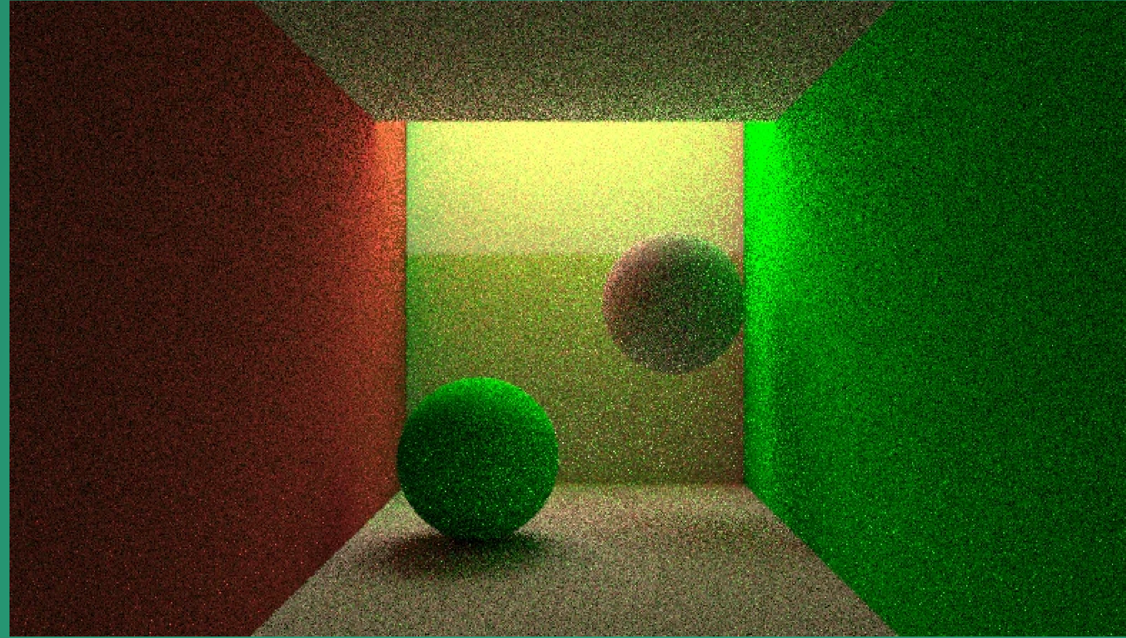
White Noise TAA

GPU "Raytracing Hardware"



Add a custom compressed spatial acceleration structure to a special memory location on the GPU, along with specialized hardware to quickly traverse it.

GPU "Raytracing Hardware"



NVidia has developed lots of algorithms that operate in screen space. Combine that expertise along with their massive machine learning abilities to create deep neural filters which can effectively denoise raytraced images.

That's it.

Spatial data structure + specialized traversal hardware

Solve problem that ray-geometry is slow and GPUs can't do branching well

Improved machine-learning-based image filters

Solve problem that raytracing is inherently a noisy procedure.

In fact, games still don't do true real-time raytracing!

Even with the spatial acceleration structure and improved image filtering, true raytracing is **still** too expensive to do in realtime.

> *shadows, AO, reflections, translucency and global illumination --UE4*

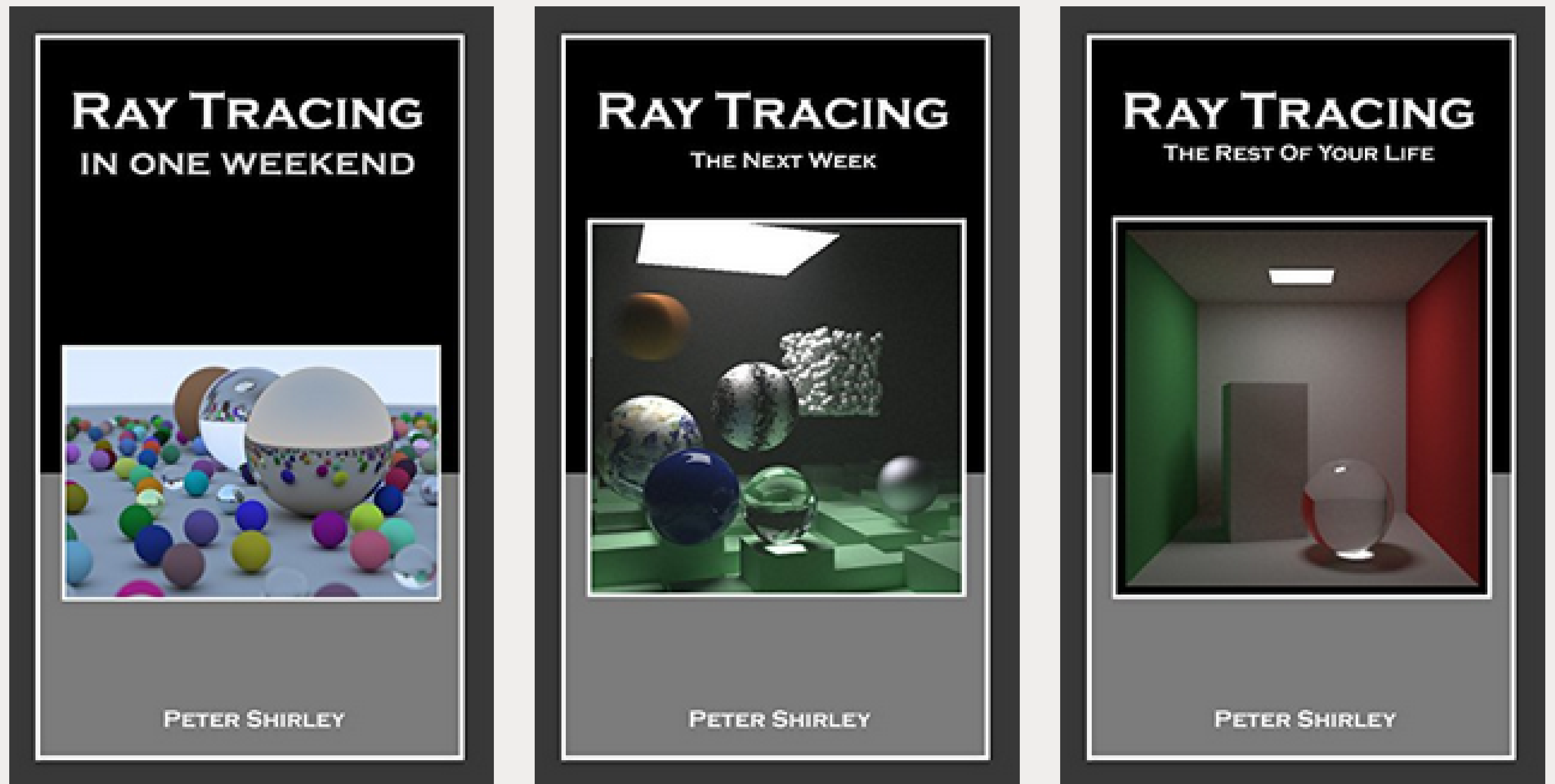
https://www.youtube.com/embed/J3ue35ago3Y?enablejsapi=1

FIN

More Info

Ray Tracing in One Weekend

https://raytracing.github.io/

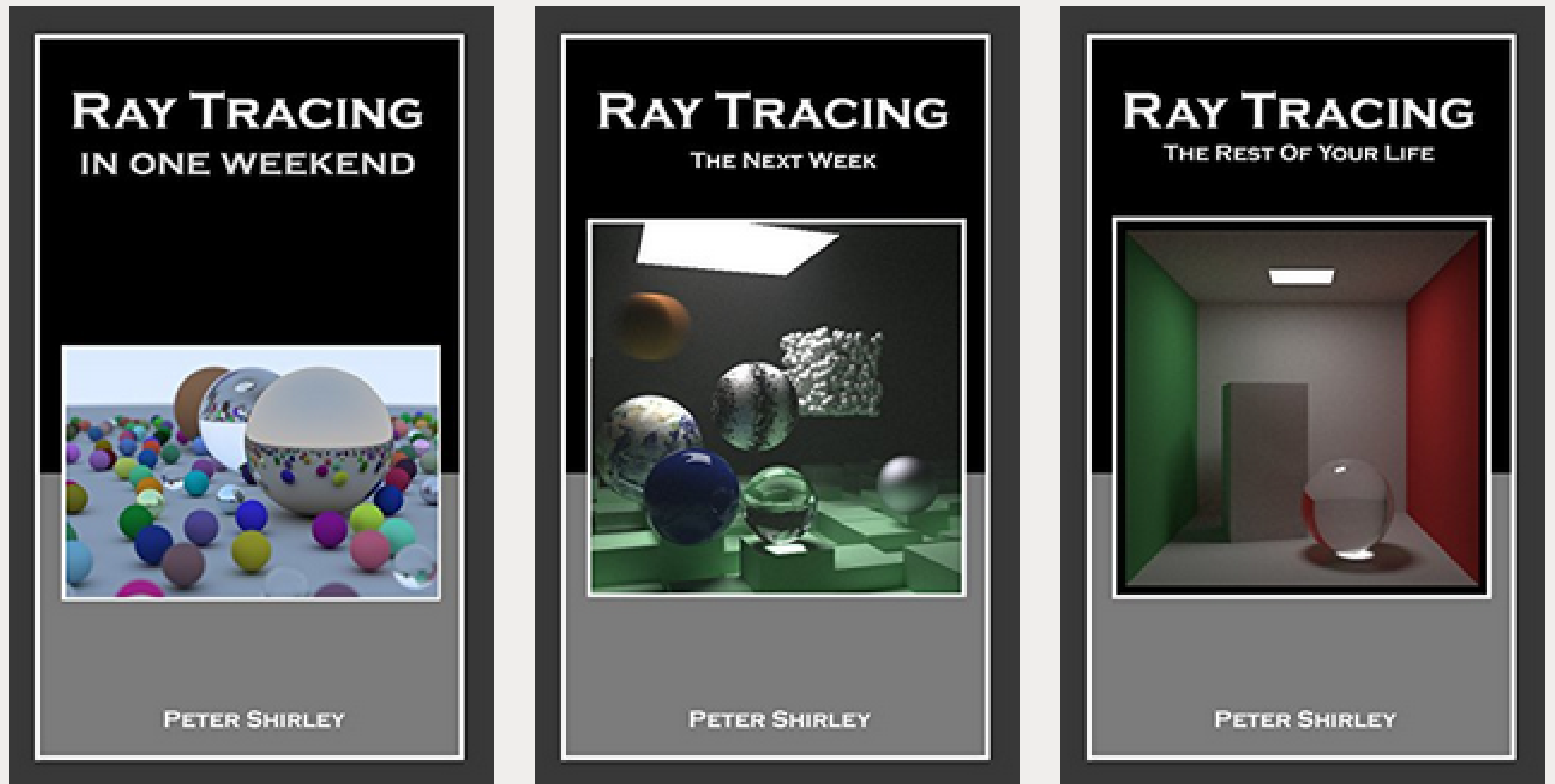


A series of three free books that are designed to get you into raytracing relatively quickly.

Uses C++---pretty similar to Java (in fact, Java syntax was built partially to emulate C++)

Ray Tracing in One Weekend

https://raytracing.github.io/



A series of three free books that are designed to get you into raytracing relatively quickly.

Uses C++---pretty similar to Java (in fact, Java syntax was built partially to emulate C++)

Scratch-a-Pixel

An excellent site for an introduction to 3D rendering that tries to be light-handed with the math. Excellent raytracing references.

https://www.scratchapixel.com/index.html

Also has a tiny bit on Perlin noise, if you're into that!

Unreal Docs

https://docs.unrealengine.com/4.26/en-US/RenderingAndGraphics/RayTracing/

An example of what working with raytracing in an existing graphics engine looks like (along with some absolutely gorgeous screenshots)