PROFESSOR: All right. We're going to get started now. For those of you who don't know me, I'm Josh Slocum, one of your TAs. And today's lecture is going to be a primer on ray tracing to help you get started on the final project.

So first, we're going to start with a little background. Tell you about ray tracing. Ray tracing is essentially a physics simulation. You simulate the way photons bounce around and interact with materials. And from that, you generate an image of a scene.

It's extremely parallel because you can trace a whole bunch of photons bouncing around at the same time. And it has a huge capacity for photorealism. So like this sample seen here, you have all sorts of reflections and light working in ways that you don't see in a normal rendering program.

The final project, as you know if you've read the handout, is going to be groups of two or three people each. And you'll be given a working ray tracer program. A very slow one, but one that works. And your objective is to make it run as fast as possible.

You can do pretty much anything you want with a few restrictions we'll talk about later. You can make algorithmic improvements, parallelize it, change the way the rendering works slightly. All sorts of things. Pretty much anything you can think of.

The only deliverables for the final project are the preliminary report due next week, in which you'll have to show us that you've profiled the project, tell us what hot spots you've identified, and ideas you have for speeding up the ray tracer, as well as any actual work you've already done.

The final project comes with two built-in scenes that can be ray traced. Scene one, this one here, is rendered much faster than scene two. And you're going to want to work on that one first. And then, there's scene two, which is slightly different. It has water and one less sphere.

These are the restrictions on what you can do for the final project. The purpose of these restrictions is basically just to make sure that everyone's ray tracer is doing the same thing. So you're all working on the same problem, and you're not achieving speed ups by simplifying out some functionality of the program.

Does anyone have any questions about the final project before I move on? All right. Go ahead.

AUDIENCE: [INAUDIBLE] we're going to do a third scene similar to the first two, right?

PROFESSOR: Oh. Yeah. In order to make sure that you're following these rules, we're going to test your ray tracer on one or more other scenes and make sure that they render properly. But these are the two scenes that you will be benchmarked on for your performance grade.

All right. Next, we're going to go over how ray tracing works and the big concepts used in the ray tracer you'll be working on. So the basic idea behind the rendering is you're going to construct a geometric model of the scene you want to render, and so it will be made of polygons, or spheres, or planes.

And then within that model, you insert a grid of pixels. They'll form the image you're going to render. And then, you just project the scene onto the grid of pixels, and you have some image. The way you do this with ray tracing is you use a method called ray casting where you take some camera and cast rays through each pixel onto the scene. And then, whatever color object the ray hits is the color that that pixel gets shaded in the scene.

This method of tracing rays is called backwards ray tracing, because you're going in opposite direction that photons go in when you're actually observing something. So if this was an actual scene, you'd have light bouncing here and into you're eye.

Whereas what you're doing is you're tracing backwards from the eye to the object.

Ray casting-- this image here is a ray image-- is very primitive. It doesn't really get you much. So what you can do is you can improve your physics simulation by adding a whole bunch more things on top of it, like shadows, and reflections, and global illumination. All sorts of fun stuff.

So if you want to start simulating shadows, and reflections, and defraction, you need to start recursively casting rays, which is called ray tracing. So when your ray bounces off of an object, you can cast a reflection ray at the same angle mirrored from the perpendicular of the object and see what color object that ray hits, and then, shade this object with that color.

You can also cast a refracted ray, and use Snell's a lot to tell which direction it goes in and do the same thing. And then, you can cast a shadow ray at all the light sources in the scene. And if the shadow ray hits an object before hitting the light source, then you know that this object is in the shade and make it darker.

And obviously, you have to limit the recursion depth for your recursive tracing, because otherwise, you'll never finish rendering. So, first way of improving lighting is to start using direct illumination, which is you have a light source at the top of the scene. And when you hit something like a wall here, you then bounce a ray at the light source.

And depending on how far this intersection A is from the light source, you shade either brighter or darker. So it might be hard to see in this light, but the wall gets darker the further away it is from the light. And then, you do shadows the same way, except at point B here, the rate it bounces from the ground hits this ball. And so, point B is in the shade.

Now, you can improve on shadows by making what's called soft shadows. Soft shadows are shadows that form a gradient from darker to lighter. They happen when you have a light source that's not a point light. So if you look at the shadows around you in the room, you'll see there are gradients, not just a solid block of

darkness.

And the way you achieve that is when you get an intersection, when your ray intersects a point, you cast multiple shadow rays at different points along the light source. And depending on how many of these rays hit something before hitting the light source, you shade some proportion of the light being casted, or the light coming from this light source.

So you can see point D here, none of the rays hit the light source, and so, it's in a very dark area. Whereas point C, one of the rays hits the light source, and so it's in a slightly brighter area.

And then reflection and refraction work very similarly to shadows, except instead of bouncing from here to the light source, you bounce off this metal ball at the angle of reflection. And you bounce through this glass ball using the angle of refraction. You can calculate using Snell's law.

Any questions before we move on? All right. So as you can see pretty clearly in this image, there's still some problems. The ceiling is completely black. Shadows are mostly black. And there's no light being transmitted through this glass ball either into the shadow here. So there's definitely some improvements we can make on our physics simulation.

The first thing we're going to look at is called global illumination, which is pretty much any sort of illumination that doesn't come directly from light sources. Now the reason this sort of illumination is important is that there's actually a lot of light that gets bounced around the room that doesn't come directly from the light source.

Like that white wall, if I shine this laser pointer at it, as you can see, there's still a lot of light bouncing off of it. It's not all being absorbed. So what you can do is you can take your ray intersection, and you can use what's called the Monte Carlo method, which is you take a whole bunch of random samples and interpolate those samples to get some value that approximates what the illuminant should be here.

The problem with that, though, is that when you trace backwards, you get this

exponential explosion of recursively cast rays. And you have no guarantee that any of these paths will ever get to this light source. So what you have to do is trace forwards, so you start at the light source, and you cast photons in random directions and let them bounce all around the scene until they get absorbed.

Now, when a photon bounces, a bunch of light gets absorbed. But not all the light. And so, the rest of the light keeps reflecting. And depending on the kind of material, different amounts and colors of light will be reflected.

So as you can see, this ray here bounces off the orange wall. And so, orange light gets reflected. This ray bounces off the blue wall. And so, blue light gets reflected. And this ray bounces off this metal ball, and so it gets completely reflected up towards the ceiling.

So if you remember each bounce in something called a photon map, then later, you can come back and say, oh, there's so many photons in this area. And that's what these yellow circles are. So the illuminance is going to be approximately this.

The problem with that is unless you're simulating an extremely large number of photons, you'll have an uneven distribution. So what you can do is you can hybridize forwards ray tracing and the Monte Carlo method.

So you construct your photon map, which is represented in this image by all the red dots scattered everywhere. And then you do backwards ray tracing. Then when you reach a point, you randomly sample different reflection rays and average their illuminance to get the illuminance at this point.

So you have blue illuminance from the wall here. And then light gray from here and darker gray from here. And the result is this sort of pale blue, which you can't really see in the image because the light's bad. Yeah, so backwards trace, randomly sample, illumination. And any questions before we go on?

So it turns out even with building a photon map and then randomly sampling the photon map, calculating the irradiance can still be very expensive. So what we've implemented to speed that up is what's called an irradiance cache. And the way the

irradiance cache works is there are certain areas on objects that will have pretty much the same irradiance. It's pretty much the same irradiance levels. So you can interpolate from nearby points.

So what we do with the irradiance cache is as the scene is rendering, if there is a cache miss, then we calculate the irradiance for some pixel which corresponds to a point in the scene. And then, if we then trace a ray very near to that same pixel, you can then interpolate from nearby cached irradiances and get something that looks right.

So if you take a look at this image, where the green dots represent cache misses on these big planer objects here. There's very few cache misses. And it's almost entirely cache hits. Whereas at the interface between different objects, the cache misses become much more common. But overall, the number of irradiance calculations is greatly reduced.

And then, the third kind of illumination that we implement is called caustics. Caustics are just light being lensed into patterns of brighter and darker spaces, such as you get from a light pointing towards a glass ball. Or if you ever see the sort of wavy pattern on the bottom of a pool, that's also from caustics.

Caustics are also implemented with the photon map, except it's created slightly differently. Instead of bouncing photons everywhere, photons are absorbed as soon as they hit a diffuse object. So a photon might bounce off of this metal sphere and then hit the wall, and it will be immediately absorbed.

Whereas if it hits this sphere, it'll bounce, and bounce, and then be absorbed. So the objective of caustics is to trace how photons are grouped based on the optical properties of materials in the scene.

And then, caustics are rendered pretty simply. Because there's much fewer photons than in the photon map for global illumination, you don't need to use an irradiance cache. So you simply trace a ray to a point and then sample the number of photons nearby. And you end up with something like this if you just do caustics shading.

So then at the end, you take your direct illumination, and then add it to caustics, and then add it to global illumination. And you get something that looks like this. Any questions before we go to the code overview? Go ahead.

AUDIENCE: Do we weight them equally?

PROFESSOR: So the question was, do we weight them equally? And the answer is yes. And you just sum them together. Ideally, you were clever and constructed your ray tracer so that they should be weighted equally, because there's weighting done internally, basically.

So next, we're going to do a quick overview of the code that will be in your repositories just get you started and help you understand how it works. First, we're going to start by explaining some of the classes you'll find. First is SceneObject, which is the parent class for all of the objects that will be used to construct the scenes you'll be rendering.

Every scene object has a ray intersection method that's called whenever you trace a ray. And it'll return where the ray intersection is and the object. And SceneObject is inherited by these objects here, which all of the objects will inherit from.

Then we've got the Raytracer class, which contains the main method. So it starts the program. And it also contains methods for constructing the scene and then rendering the scene once you've constructed the scene first.

We have the LightSource class, which is a base class for any light sources. In our ray tracer, the only subclass is square photon light, which is this light at the top of the scene here. And what that does, or what the light class does, is it contains methods for constructing the photon maps and getting illumination from either global, or caustic, or direct sources.

And then there's also the iCache class, which essentially just implements the iCache for you. Any questions? So basic execution overview.

You have the main method for Raytracer. Parses the command line. Constructs the

scene. And the scene is just a command line switch. So if you say S1, it'll construct scene one. If you say S2, it'll construct scene two. Then for each light source, it tells the light source to make photon maps for the global and caustic illumination.

GUEST SPEAKER: I should mention that we actually moved the main functionality into [INAUDIBLE] so you can replace the data. You don't have to touch that. It just constructs the scene, so that we can then add a scene [INAUDIBLE] from that. So you shouldn't need to do that.

PROFESSOR: OK. So that's changed then. Then, main calls the render function of Raytracer. What the render function does-- excuse me, method does-- is it casts a ray for each pixel in the scene you're going to render, finds the nearest intersection, and then computes the shading at that intersection by recursively bouncing rays and [? painting ?] the illuminance cache for illumination.

And that's basically what the Raytracer does. It's pretty simple. So we're also going to quickly go over how the light source traces photons for direct and global illumination. The difference is kind of settled, but pretty important.

With global illumination, you want some total number of photons to be put in the map. And you put a photon in the map whenever it hits some diffuse surface. And then, that photon then continues bouncing around until it's absorbed.

For caustic illumination, it's pretty similar. You have some set number of photons you want to get into the map. You cast a ray in some random direction. And then, the ray intersects with a diffuse material. You store the intersection in the photon map. And then you break. You don't keep bouncing the photons around.

So that's basically it for the code overview. This slide has some resources that you can use if you want to learn more about ray tracing. Some of these will explain concepts that we haven't implemented. They're an interesting read, but not needed if you don't want to read them.

And also, there's no standard way of doing global illumination or direct illumination.

So you may come across resources that do things slightly different than our reference implementation. That doesn't mean either one is wrong. But you should be aware that there will almost certainly be differences. Does anyone have any questions? Go ahead.

AUDIENCE: So we can use one of those alternatives?

PROFESSOR: Can you say that louder please?

AUDIENCE: So we can use one of those alternatives?

PROFESSOR: These aren't implementations. They're descriptions of how ray tracing works. You should not change your code to work in this way. These are just to help with clarification if you want to learn more about what's being done in the code. Go ahead.

AUDIENCE: [INAUDIBLE PHRASE] what specifically are we allowed to change for the [INAUDIBLE]?

PROFESSOR: So what specifically are you allowed to change?

AUDIENCE: Yeah. Which ever one's smaller [INAUDIBLE PHRASE].

PROFESSOR: Anything not prohibited by these rules, which are paraphrased from the project handout, you can do. So basically, as long as you're still performing the same essential calculations, you're fine. Hmm?

GUEST SPEAKER: I would say the criteria for correctness is to look at the images, and say that they look more or less the same. They don't give you a pixel by pixel accurate. This isn't a graphics class that actually does that. This should be almost like a [INAUDIBLE].

PROFESSOR: They should visually look the same. If you find some approximation that looks almost the same and runs 100 times faster, go for it. You can't say, for example, simplify your ray tracer though by saying, OK, we're only going to be able to render these two scenes, but we'll be able to do it really fast.