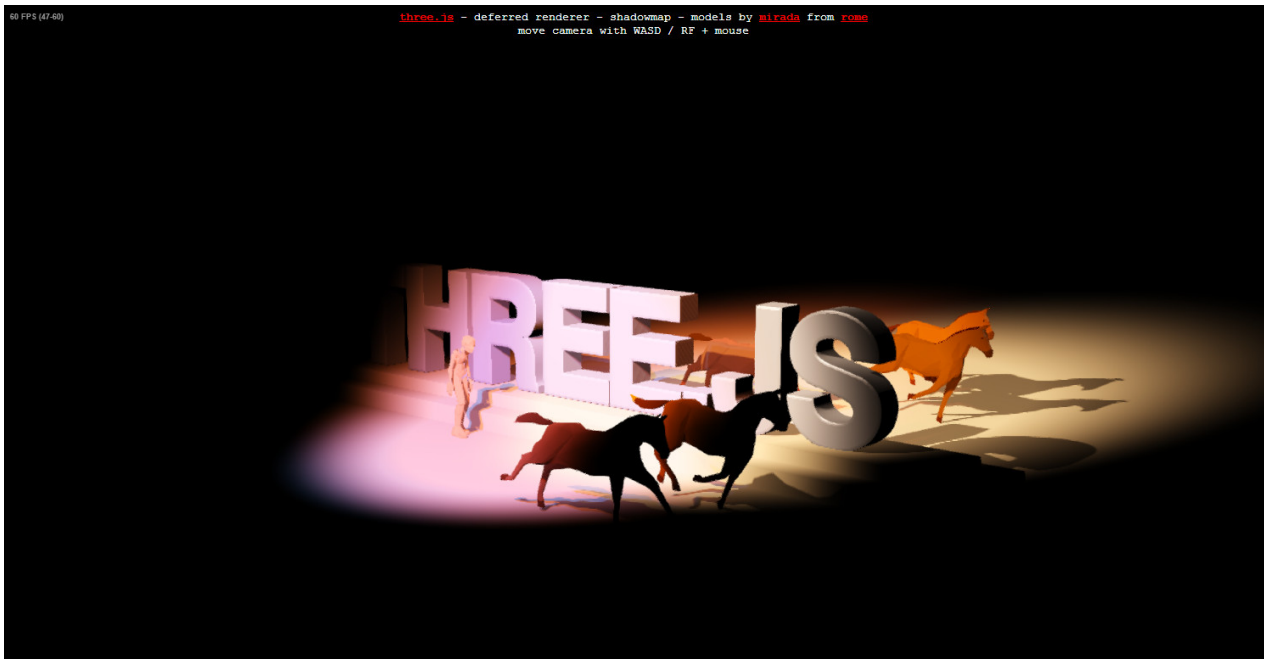
Lesson 6: Lights

Lesson: Lights

[recorded session 3/26 - session 1]

[start with one of the cool deferred lighting demos: http://mrdoob.github.com/three.js/examples/webgldeferred_animation.html - avoid “nude”, wait for her to walk by. Aha, better one:

http://alteredqualia.com/three/examples/webgl_deferred_shadowmap.html
 ]



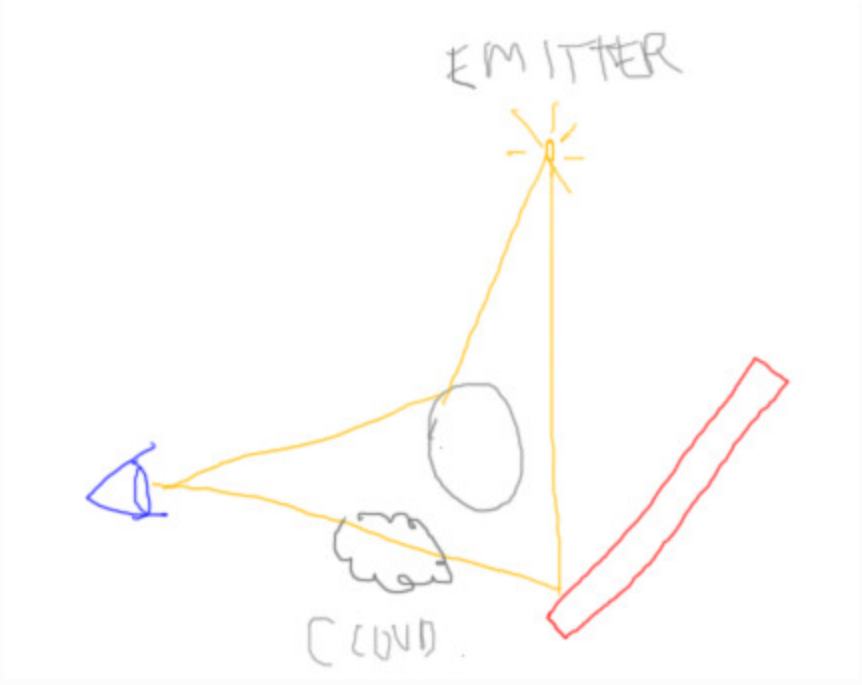
A 3D virtual world has three major components: objects, lights, and cameras. This unit is going to explore lights; the next, cameras. These components are on opposite ends of the rendering process: photons are generated at light sources, and the photons that reach the camera are what form the image. The objects in the scene are what ties these two ends of the process together.

[Additional Course Materials:
The demo run in this lesson is
[here](http://mrdoob.github.com/three.js/examples/webgldeferred_animation.html). It uses a technique called “deferred shading”, which we’ll talk about later.

As a reminder, course code, lesson scripts, and much else is available from [links on the wiki](https://www.udacity.com/wiki/cs291). All [Instructor Notes](https://www.udacity.com/wiki/cs291/instructor-comments) are also available in one place. Finally, don’t forget [the forums](http://forums.udacity.com/tags/cs291/#cs291).
]

Lesson: Photons as Particles

[image of light bouncing around, very general, and show fog.]



[“Light makes right.” - Andrew Glassner]

I love this little play on words that Andrew Glassner made up back in the 1980’s. It succinctly captures a kernel of truth about computer graphics: the more you can determine about how light moves through an environment, the better your results will be.

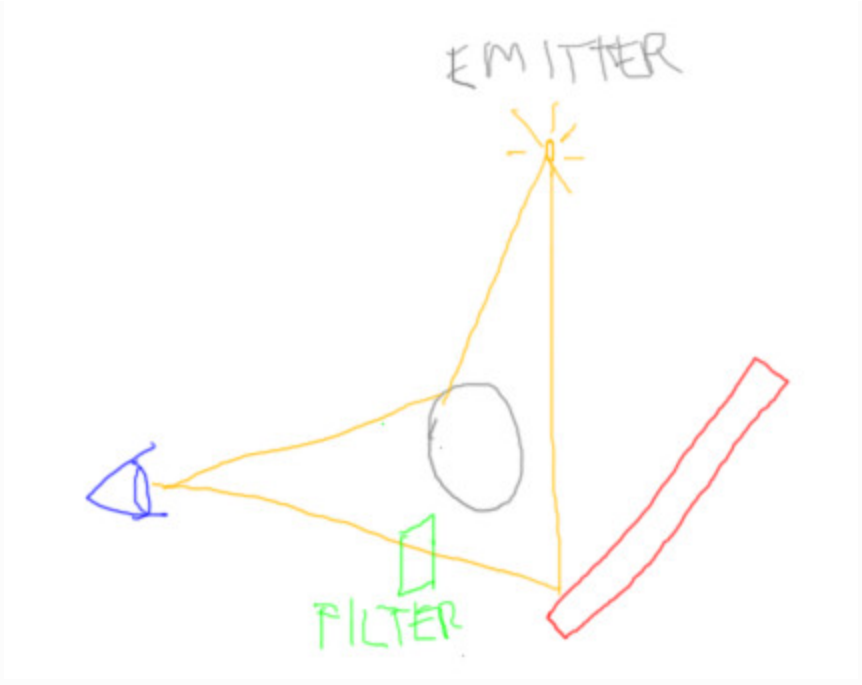
I’ve talked a bit before about light and how it bounces around a scene, eventually reaching the eye. The idea is that each light, often called an ***emitter***, sends photons out. The fates of each photon is that it’s absorbed as-is or reflected. An object can absorb light, and in fact almost

everything absorbs at least a little light - not even the highest quality mirror reflects 100% of the light hitting it, the numbers are usually more in the 90 to 95% range. Along the way photons can be absorbed by dust, water droplets, or other particles in the air or inside of translucent objects themselves.

This simple photon model ignores various effects, such as ***polarization*** and ***fluorescence,*** but that's generally fine in interactive computer graphics. Things are complex enough!

We can see objects precisely *because* photons ultimately travel from them to us. The reason objects have different colors is that each varies in how much it reflects various wavelengths of light.

[replace cloud with filter]

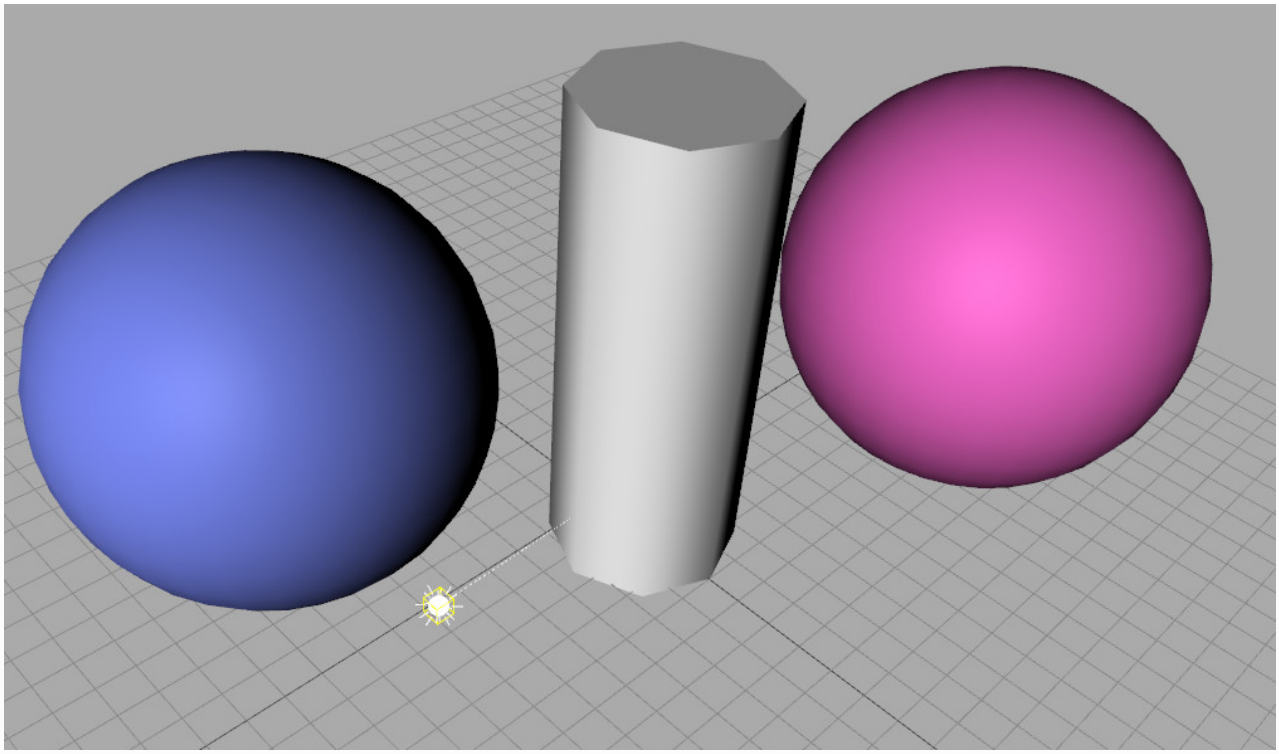


Where we do have some choice in computer graphics is what light paths we follow. The two paths we've used so far are light to surface, and surface to eye. We've been a bit more elaborate with surface to eye, in that we've allowed transparent filters along this path to modify the color.

[Additional Course Material:
I mentioned it before, but a [wonderful short video](http://www.youtube.com/watch?v=1qQQXTMih1A) of Richard Feynman talking about electromagnetic waves is worth watching. Even most mirrors are [not fully reflective[(http://en.wikipedia.org/wiki/Mirror).
]

Lesson: Directional Lights

[Use editor and explain as we go http://mrdoob.github.com/three.js/editor/]



Light emitters can take whatever form you like, since ultimately the effect of a light is something computed in the vertex or fragment shader. That said, there are a few different common forms of lights used in applications. The simplest is called a directional light.

This type of light is defined by just that, a direction vector. A directional light is something like the sun. The direction to the sun is essentially the same for every object on earth. A directional light is considered infinitely far away. In many modelers, such as the three.js editor here, the directional light will be shown as some local point that you can move around. However, you're truly manipulating the direction, not the location of the light.

Since we often combine colors in various ways, there's not necessarily an upper limit of 1.0 to a color channel. It all depends. For example, if I want to clear the screen, I would want to use values from 0.0 to 1.0. There is an upper limit of each channel's intensity, so values larger than 1.0 are beyond the monitor's abilities. However, if I wanted to make a light bright, I might set its intensity to 11. When the distance from the light to the object and the object's color are taken into account, the final color may well be within the 0.0 to 1.0 range.

In some systems you can even set the light intensity to a negative number, meaning the light is somehow sucking photons from a surface. The light's color is just a number in an equation, after all. This is true for a material's color, too, that it could be set however you want, but we tend to leave material colors between zero and one, thinking of them as reflectivities. These basic illumination equations are not based on any physical units, such as watts or lumens, but rather are set in the zero to one range for our convenience.

Lesson: Directional Light in three.js

[recorded 3/26 session 2]

In three.js a directional light is created and added to a scene like this:

```
var light = new THREE.DirectionalLight( 0xFFFAAD, 0.7 );
light.position.set( 200, 500, 600 );
scene.add( light );
```

```
var light = new THREE.DirectionalLight( 0xFFFAAD, 0.7 );
light.position.set( 200, 500, 600 );
scene.add( light );
```

The first value is the color of the light, in this case a light yellow, the second is the intensity. Setting the light's position is how you actually set the vector. By setting 200, 500, 600, we're saying that the light, which is infinitely far away, comes from a direction that goes from point 200,500,600 to the origin. Note that the length of the direction vector does not matter.

```
light.position.set( 2, 5, 6 );
light.position.set( 0.02, 0.05, 0.06 );
```

```
light.position.set( 2, 5, 6 );
light.position.set( 0.02, 0.05, 0.06 );
```

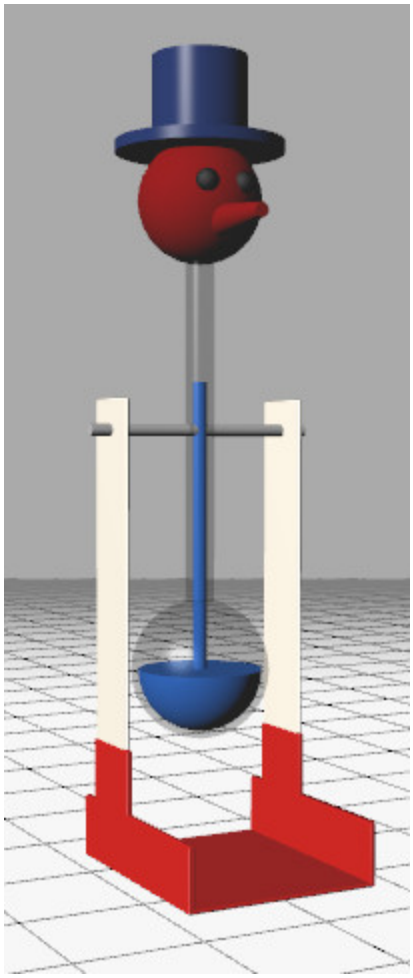
We could have just as easily set 2,5,6 or 0.02,0.05,0.06 for this so-called position and we'd get the same result.

Exercise: Set a Directional Light

[***direction to light: -200, 200, -400***
color: full white
intensity: 1.5
]

I've ripped the lighting out of the drinking bird scene, so that you can set it up. Your task is to put a directional light in fillScene(). The direction to the light from the origin passes through the point -200, 200, -400. Make the light white, with an intensity of 1.5.

When you're done, the scene should look like this.



[Exercise at unit6-dir_light_exercise.js]

Answer

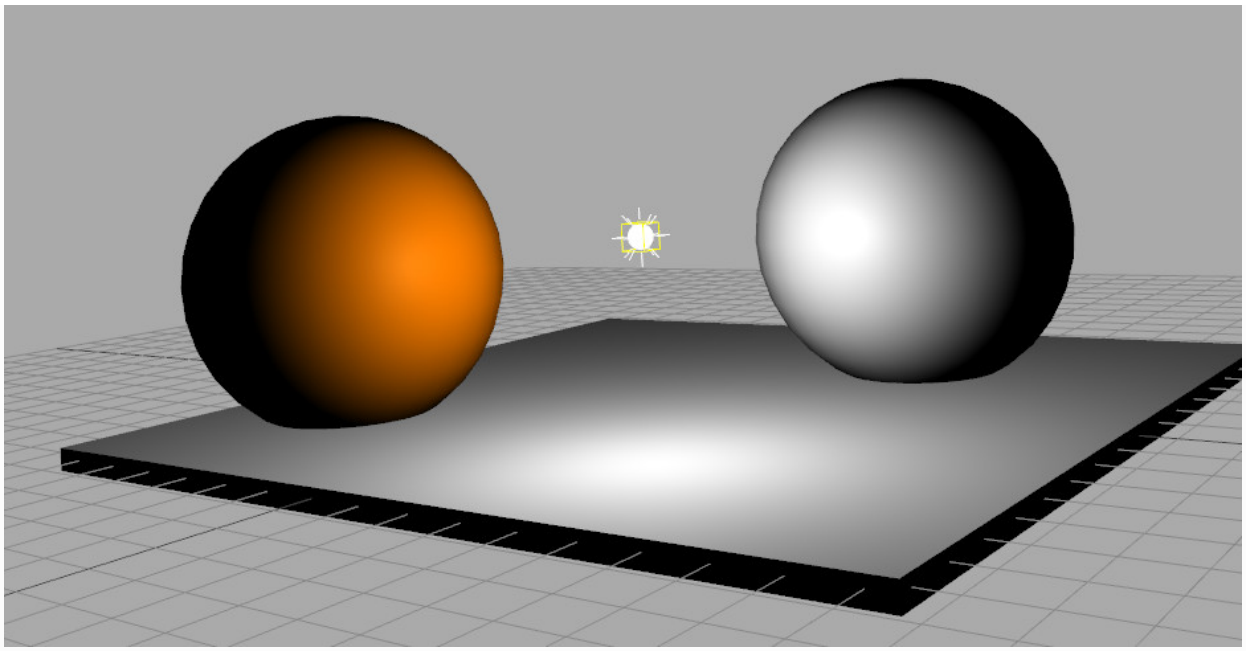
```
var light = new THREE.DirectionalLight( 0xFFFFFF, 1.5 );
light.position.set( -200, 200, -400 );

scene.add( light );
```

The code for this one is just this bit, in fillScene(). Create a directional light with a white color and intensity of 1.5, position it so it's coming from direction -200, 200, -400, and then add the light to the scene.

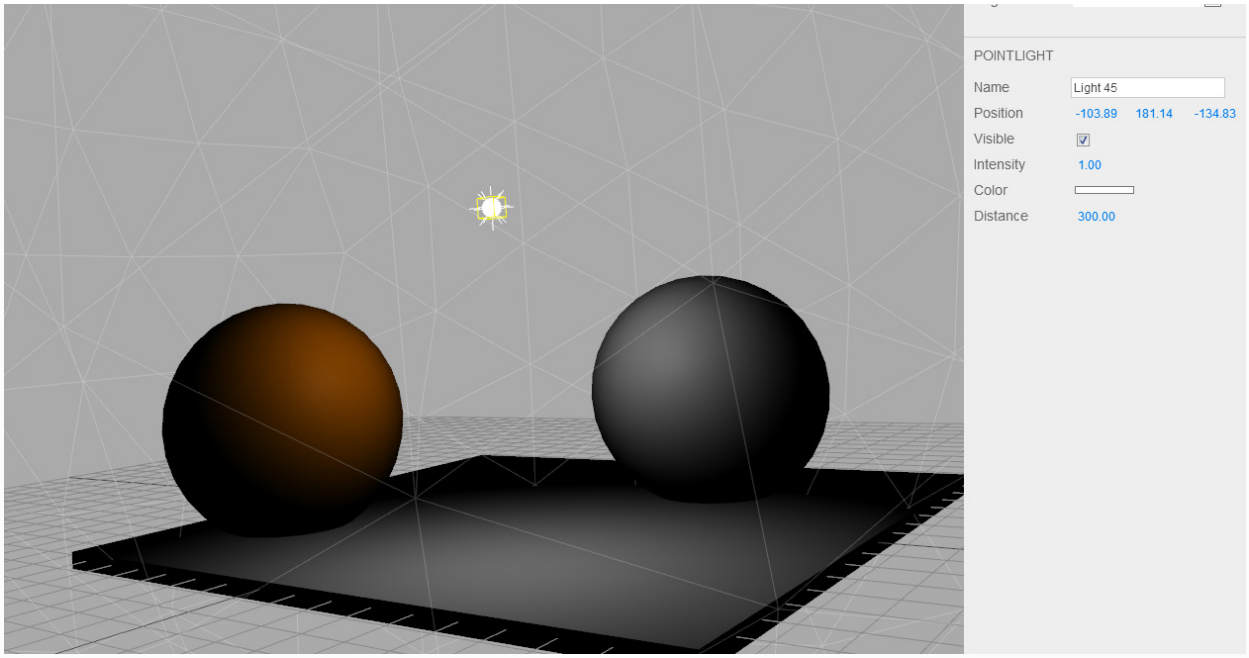
Lesson: A Point Light

[Use editor and explain as we go http://mrdoob.github.com/three.js/editor/]



A point light is what it sounds like: you define a position for it in space and it gives off light in all directions. You can set the color and intensity, just as you can with directional lights. These lights are different from real-world lights in that, by default, the distance from a light does not affect its brightness.

Attenuation is not done mostly because it's easier to quickly light a scene with point lights if there's no drop off with distance. It's entirely possible to use a different equation for how the light is attenuated by the distance. For example, you could divide the intensity by the distance, or the distance squared (which is how real lights work), or any other combination.

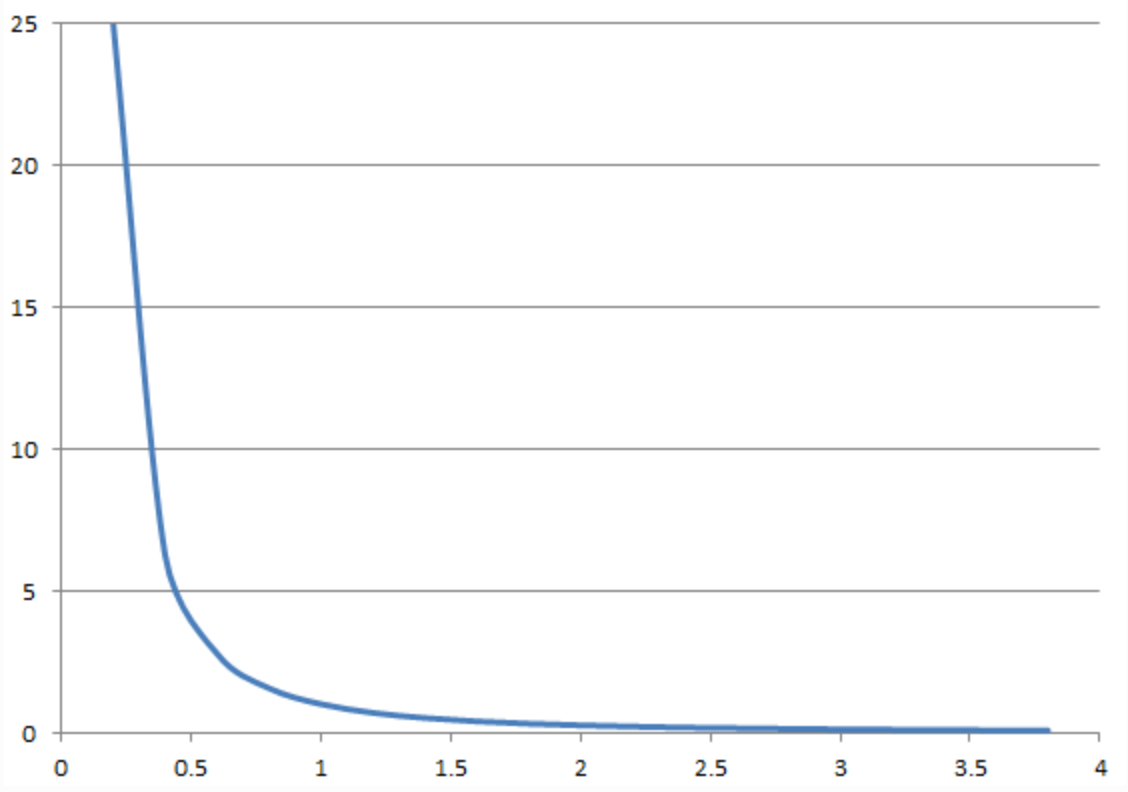


Three.js supports only one drop-off mode, which is to define a maximum distance. At this distance the light goes to zero. This is entirely non-physical, but is useful for the artistic control of lighting. You can put point lights in a scene and have each affect a local area, without affecting the lighting in other parts of the scene.

Lesson: Ambient Lighting

[show drop off equation distance squared -

label axes: Y is light intensity
X is distance



indirect illumination
ambient lighting
]

Think about being in a room at night with a lamp on your desk. You drop your pencil under the desk. The only reason you can see the pencil is that light from your desk lamp bounces off the walls, ceiling, and other surfaces. Some of this light makes it to beneath your desk and then gets to your eye.

One reason that we don't use lights that drop off with the square of the distance is that light doesn't bounce around in our scenes but only affects what it directly hits. Since there's no light bouncing around, it dies off very quickly within our simplified system.

We compensate for this lack of reflected light, called "***indirect illumination***", by adding a fudge factor, called ***ambient lighting***. In three.js we set this light by giving it a color.

```
scene.add( new THREE.AmbientLight( 0x222222 ) );
```

scene.add(new THREE.AmbientLight(0x222222));

The color is multiplied by the material's ambient color to give a solid amount of fill color:

```
var someMaterial = new THREE.MeshLambertMaterial( );
someMaterial .color.setRGB( 0.8,0.2,0.1);
someMaterial .ambient.copy( someMaterial .color );
```

```
var someMaterial = new THREE.MeshLambertMaterial( );
someMaterial.color.setRGB( 0.8,0.2,0.1);
someMaterial.ambient.copy( someMaterial.color );
```

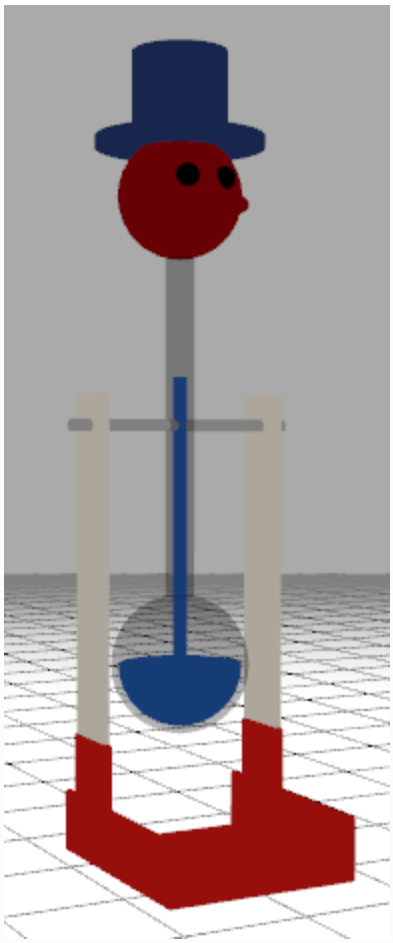
Remember that the ambient color on the material is actually a separate color in three.js. If you don't set it, the color is white and the ambient light will simply add some gray to the object as a whole. Gray's OK, but it's generally better if you set the ambient color to match the diffuse color. You can then use the ambient light's color to change the feel of the scene as a whole.

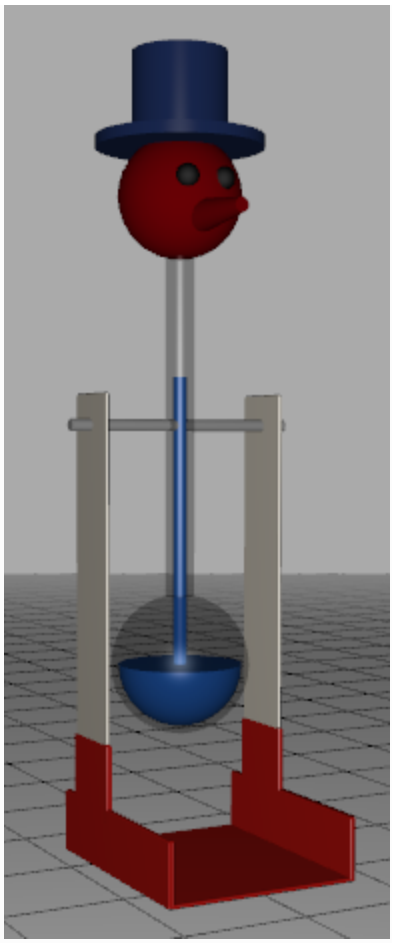
Exercise: Head Light

```
var light = new THREE.PointLight( 0xFFFFFF, 1.0 );
light.position.set( 1000, 1000, 1000 );
scene.add( light );
```

```
var light = new THREE.PointLight( 0xFFFFFF, 1.0 );
light.position.set( 1000, 1000, 1000 );
scene.add( light );
```

[put just code above and first image to start - second image will follow]





Creating a point light in three.js is simple enough, and you can look at the documentation page for more details. In this exercise, the code is set up so that the scene is lit by only ambient lighting. It's quite flat, as you can see, and detail is lost.

[add second image]

What I want you to do is turn off the ambient lighting and replace it with what I call a "head light". Think of a miner's light, something you put on your head to look around in the dark. Instead of a flat ambient light, you have a light that is guaranteed to illuminate everything you see, since it is located wherever you are.

If you search for the word "headlight" in the code you'll see where you need to modify things. There are two tasks you'll need to do. The first is to add the proper type of light to the fillScene() method. Use the ***headlight*** variable, which I've declared at the top of the code for you. Declaring it at the top makes this variable usable in any of the methods in the code.

Make the headlight a full white color and intensity of 1.0. The second task is to make the light's position always match that of the camera for every frame. If you look in the render() method you'll

see this code:

```
function render() {
	var delta = clock.getDelta();
	cameraControls.update(delta);

	renderer.render(scene, camera);
}
```

```
function render() {
    var delta = clock.getDelta();
    cameraControls.update(delta);

    renderer.render(scene, camera);
}
```

This method is called every time a new image is to be rendered. Don't worry about the delta and cameraControls calls, leave those alone. The key line here is the renderer object's render() call. This call is what actually renders the scene. What you want to do is make the headlight's position match the camera's position before rendering. By doing so, the headlight will then shine on everything visible, bathing it with illumination. The surfaces then are not all the same dull solid ambient color but have some definition to them, which looks more realistic.

Both the light and the camera are derived from Object3D(), so you can access the position of both of these in the same way. Bon chance! Good luck!

[exercise at unit6-headlight_exercise.js]

Answer

```
	// scene.add( new THREE.AmbientLight( 0xFFFFFF ) );
	headlight = new THREE.PointLight( 0xFFFFFF, 1.0 );
	scene.add( headlight );
```

```
// scene.add( new THREE.AmbientLight( 0xFFFFFF ) );
headlight = new THREE.PointLight( 0xFFFFFF, 1.0 );
scene.add( headlight );
```

In fillScene() you need to delete the ambient light (or at least comment it out), then create a point light and add it to the scene. The position of this light initially doesn't matter, since it will be moved to match the camera's position.

```
headlight.position.copy( camera.position );

renderer.render(scene, camera);
```

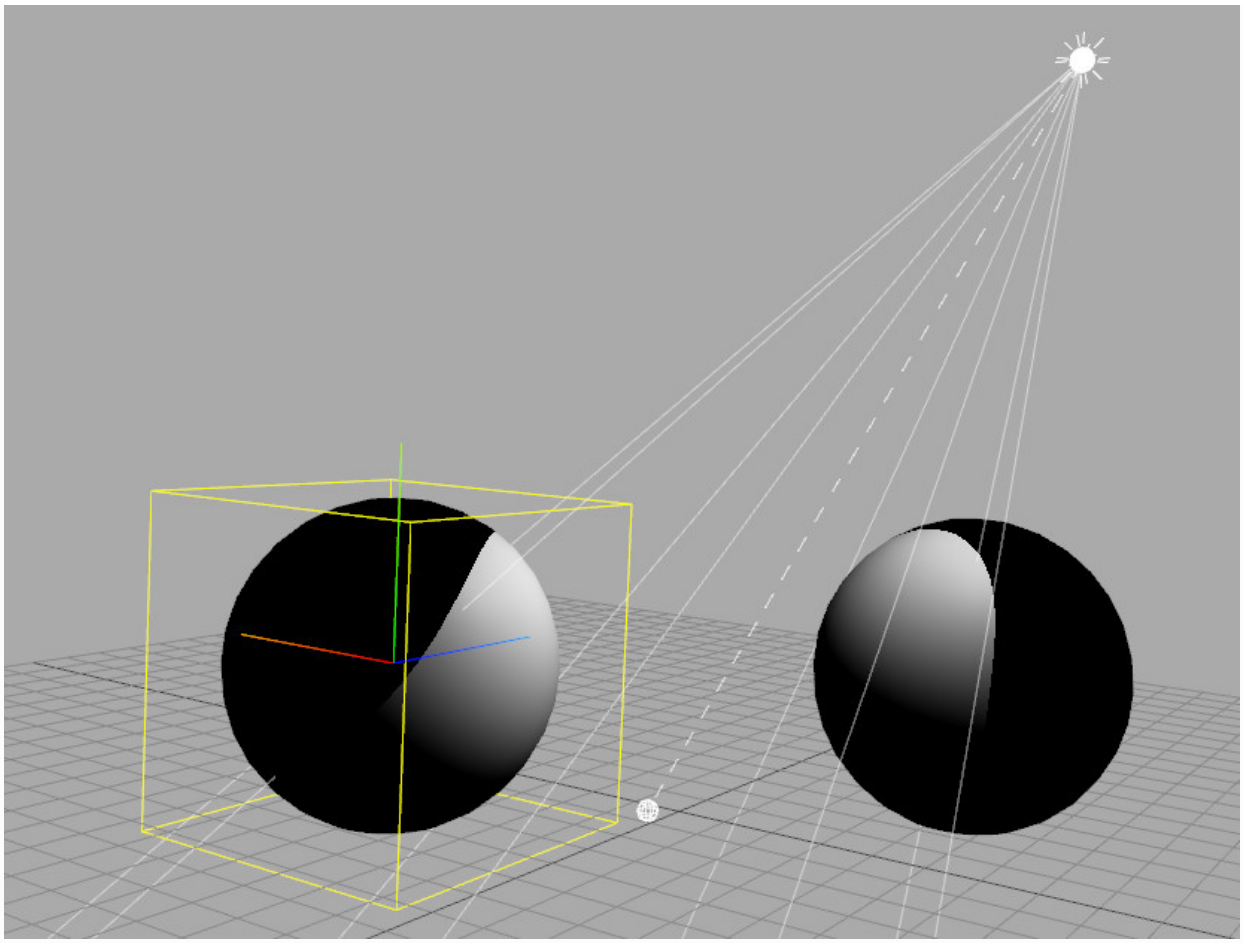

```
headlight.position.copy( camera.position );

renderer.render(scene, camera);
```


In the render() method you copy the camera's position to the headlight's. Now when rendering takes place you have a headlight giving a varying shade to the visible areas. In practice you may want to set the headlight's intensity lower, since its effect is fairly obvious.

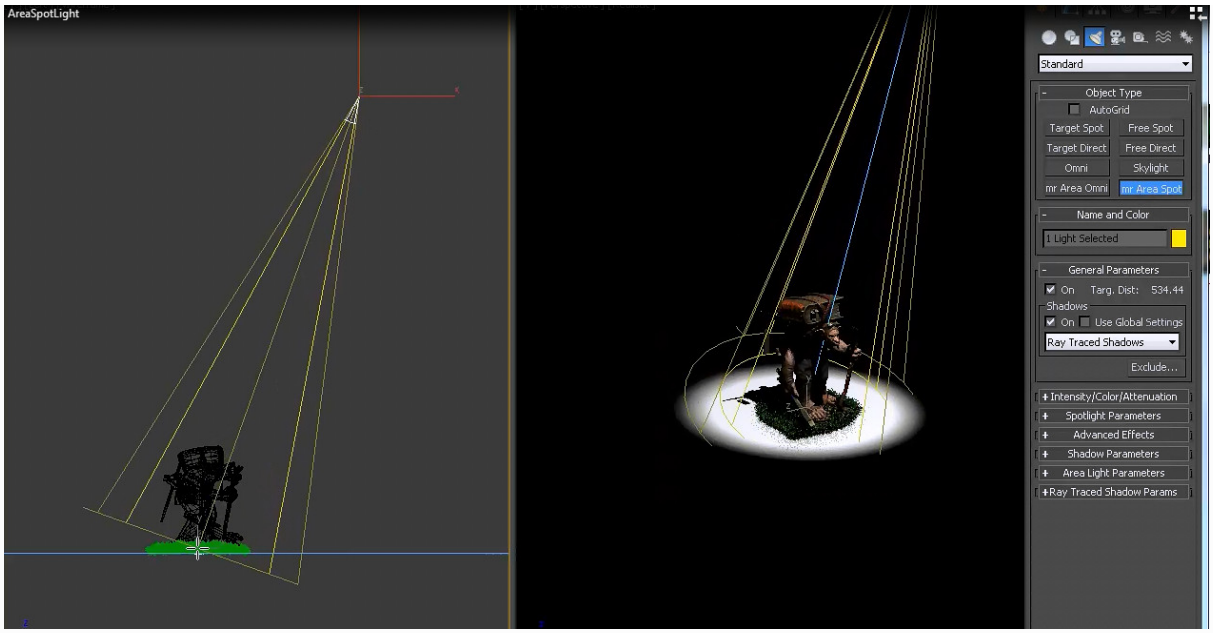
Lesson: Spot Light

[Use editor and explain as we go http://mrdoob.github.com/three.js/editor/ - add a ground plane for sure!!! Show angle and exponent effect.]



A spotlight in three.js is similar to a spotlight in the real-world. You shine this sort of light on something to make it the focus of attention. This emitter is like a point light, in that it has a position. It is also like a directional light, in that you point it somewhere. However, there's one important addition: control of the cone of light it forms. The main control in three.js is the angle at which the spotlight ends. There is also a falloff exponent that is similar to how specular highlighting works: as you increase this number the spotlight becomes tighter.

[video in Unit6_SpotLight, AreaSpotLight.mp4, run from beginning until I stop talking. Does *not* have to finish, probably better if it ends early.]



More elaborate spotlights are possible. For example, this is 3DStudio Max's spotlight. It uses two angles to control the outer cone and a separate inner cone it calls a "hotspot". The spotlight can also be circular or rectangular. There are no rules about what constitutes a spotlight, and with shader programming you can create any sort of spotlight you can imagine.

[http://mrdoob.github.com/three.js/editor/ continue to run three.js editor and add a point and directional light to a scene.]

I should also mention at this point that you can put any number of lights in a scene. There's only one ambient light setting, but you can have as many spot, point, and directional lights as you want. Each light's contribution is added in to the fragment's final color.

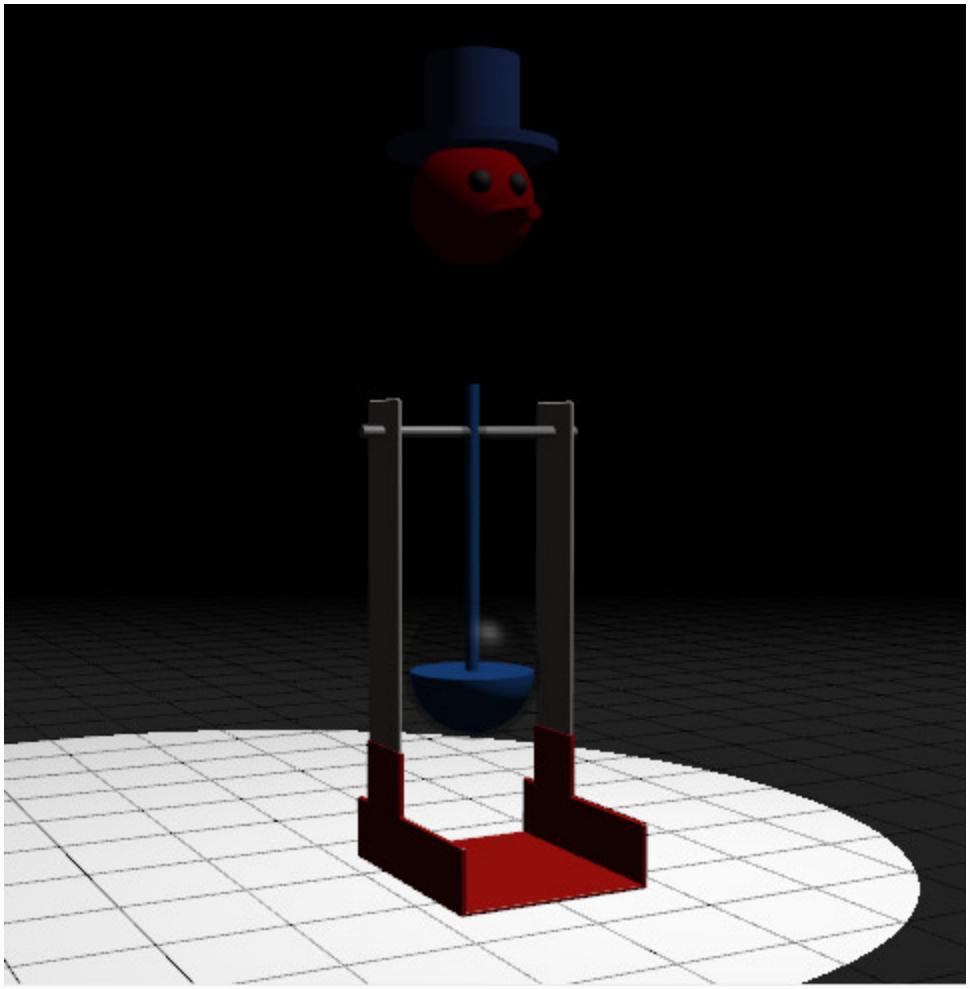
Exercise: Spot Light in three.js

Let's show off the drinking bird. Replace the directional light in this scene with a spotlight having the following characteristics:

color: full white
intensity: 1.5
location: -400, 1200, 300
angle: 20 degrees
exponent: 1

target position: 0, 200, 0

I recommend reading the documentation for how to set the target position, which is where the spotlight is pointing.



[just show static image, not the running demo.]

When you're done the scene should look like this.

It's not a huge change, but does look more dramatic. Things will get even more interesting in a minute.

[Additional Course Materials:
The spotlight documentation is
[here](http://mrdoob.github.com/three.js/docs/56/#Reference/Lights/SpotLight).
]

Answer

[recorded 3/26 start of session 3 - fixed 7/17, intensity changed to 1.5]

```
var light = new THREE.SpotLight( 0xFFFFFF, 1.5 );
light.position.set( -400, 1200, 300 );
light.angle = 20 * Math.PI / 180;
light.exponent = 1;
light.target.position.set( 0, 200, 0 );
scene.add(light);
```

```
var light = new THREE.SpotLight( 0xFFFFFF, 1.5 );
light.position.set( -400, 1200, 300 );
light.angle = 20 * Math.PI / 180;
light.exponent = 1;
light.target.position.set( 0, 200, 0 );
scene.add(light);
```

Here's my solution. One thing to realize is that the angle must be set in radians. The other is looking up how to access the target position. I added this to the online documentation after I had problems myself. I recommend the same to you: if you find a problem in the three.js documentation, fix it and submit it using github. You'll help at least a hundred other people.

At this point you might want to experiment with the spotlight and play with its various parameters, either in this program or using the three.js editor.

[Additional Course Materials:
Instructions for contributing to three.js are
[here](https://github.com/mrdoob/three.js/wiki/How-to-contribute-to-three.js).

The three.js scene editor is [here](http://mrdoob.github.com/three.js/editor/).
]

Lesson: Deferred Rendering

[http://mrdoob.github.com/three.js/examples/webgldeferred_pointlights.html]

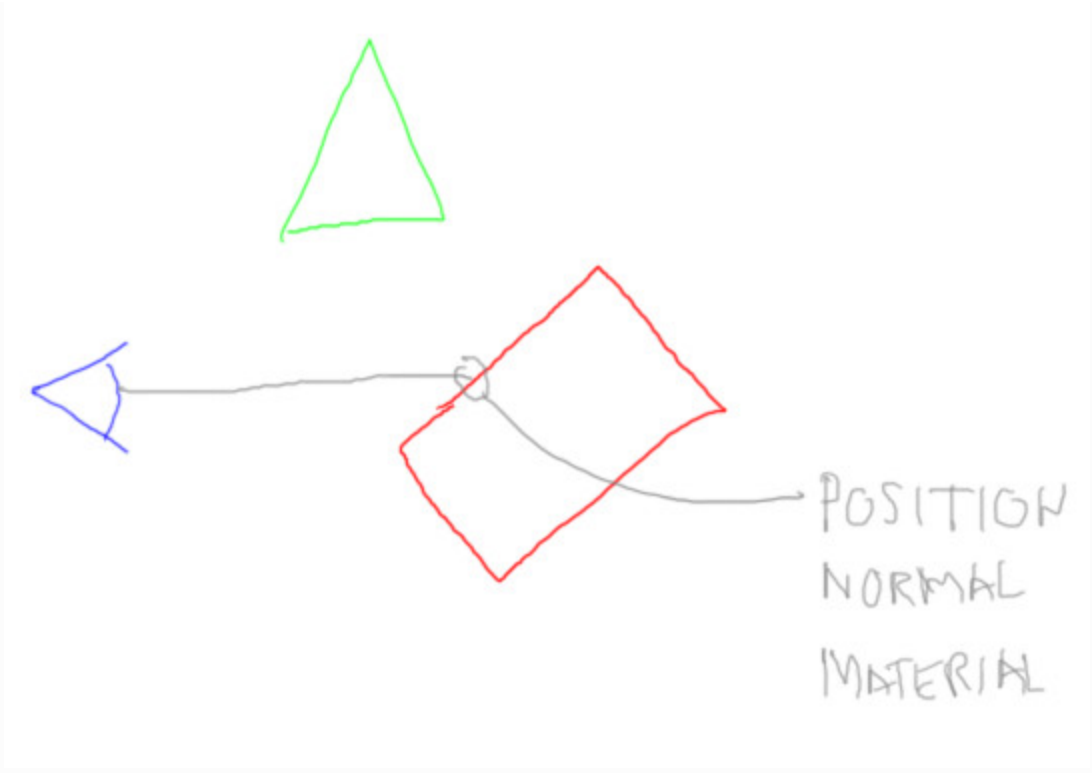
[**Delete**, as this is incorrect: One thing you'll notice if you try to put too many lights in a scene is that three.js will eventually display none of them - the object will be black. If you add four spotlights all surfaces with the Phong Material will not have these applied, though LambertMaterial objects will.

[**Delete** What is happening internally is that three.js attempts to make a fragment shader for the surface, but it's running out of instructions! OpenGL ES 2.0 doesn't have a lot of instruction memory available. If you run DirectX 9, which is comparable in its capability level, you have just 64 arithmetic instructions total per fragment shader. [and 32 texture instructions].

[**Delete** The reason this happens to only the PhongMaterial objects is that The PhongMaterial uses the fragment shader to compute illumination. The LambertMaterial does its computations in the vertex shader, which usually has more instruction space. Also, Phong illumination has additional computations for the specular highlight that Lambert does not. If you added more and more lights, eventually even the vertex shader would run out of instructions and the LambertMaterial objects would go black, too.]

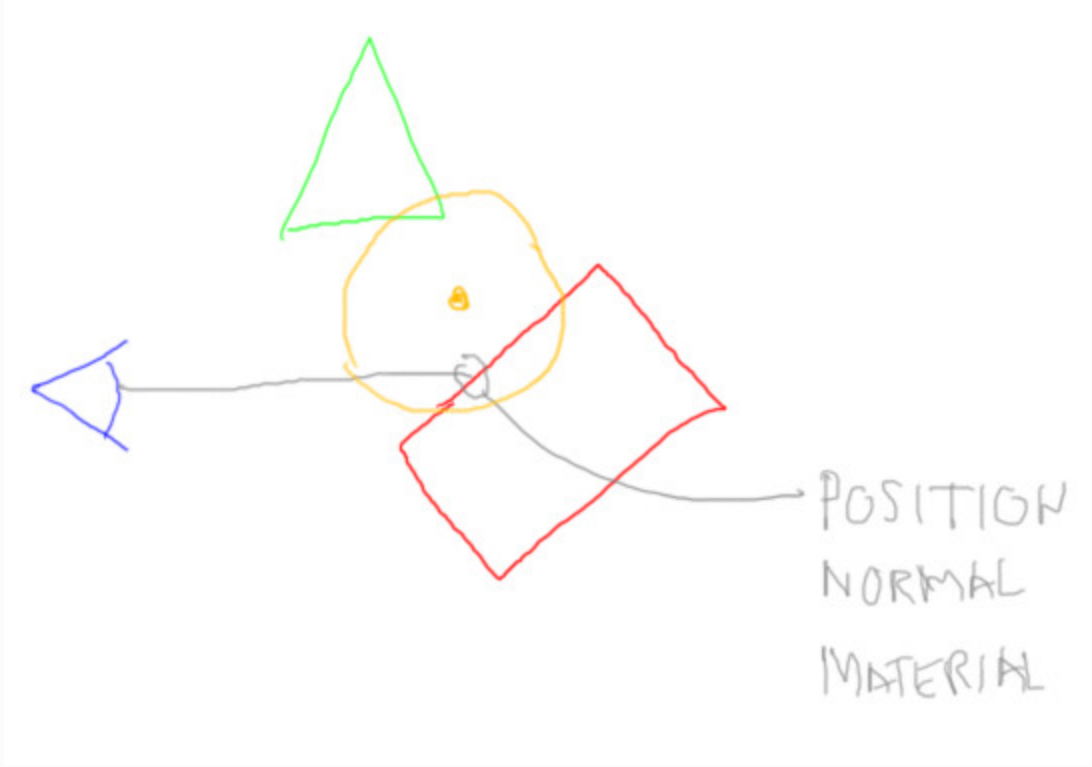
A problem with adding more and more lights to a scene is the expense. Every light you add means yet another light that must be evaluated for the surface. One way around this is deferred rendering. Look at this demo: there are 50 lights in this scene and it runs just fine.

[Draw a deep pixel structure for a surface pixel, then show light getting rendered]



Normally you render a surface and the fragment color for each pixel is stored, if it's the closest visible object. In a deferred rendering algorithm you instead store data of some sort at each pixel.

There are many variations, with names such as deferred shading versus deferred lighting, and here's just one: you could store the position, normal, and material color and shininess of the closest surface at each pixel. You also draw into the z-buffer, as usual. I'm not going to get into the details of *how* you store these various pieces of data; the point is that you *can* do so. It's just image data in another format.



With deferred rendering, every point light in the scene has an upper limit as to how far its light goes. This distance forms a sphere, so a sphere is drawn in a special way for each light. Another way of saying this is that each light can affect a volume in space. Whatever surfaces we find inside the sphere are affected by the light. Each light affects a fairly small number of pixels on the screen, namely whatever area the light's sphere covers. This means that a huge number of lights with a limited radius can be evaluated in this way.

By drawing a sphere, we are telling the GPU which pixels on the screen are covered by the light and so should be evaluated. There are variants on what shapes you draw - a circle, a screen-aligned rectangle. Whatever is drawn, the idea is that the geometry's purpose is to test only a limited set of pixels potentially in range of the light. This is as opposed to standard lights, where every light is evaluated for every surface.

I hope this gives you a flavor of how deferred rendering works. I'm really jumping the gun here - you need to know about shader programming to implement these - but the idea is to treat lights as objects that are rendered into the scene after all object surface values are recorded. There are some problem cases for deferred rendering techniques, such as transparency, but it offers a way to have an incredible number of lights in a scene.

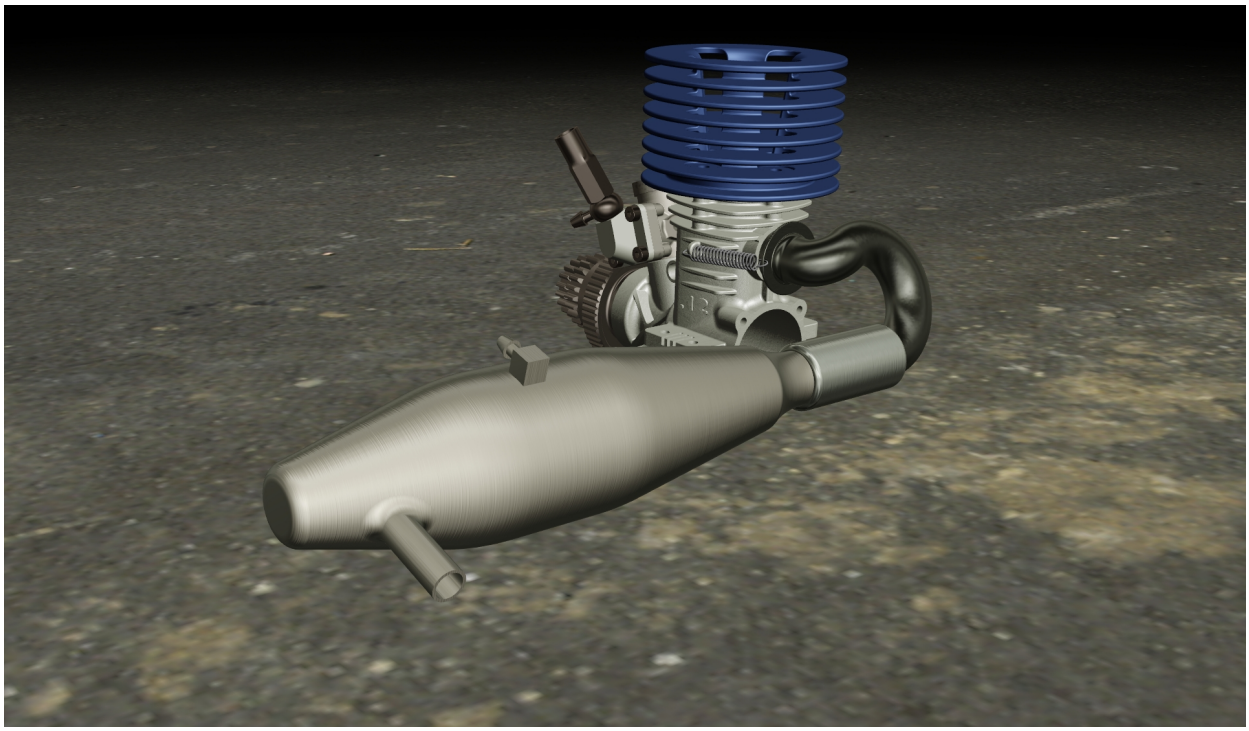
[Additional Course Materials:
The three.js distribution has three deferred rendering examples,
[here](http://mrdoob.github.com/three.js/examples/webgldeferred_pointlights.html),

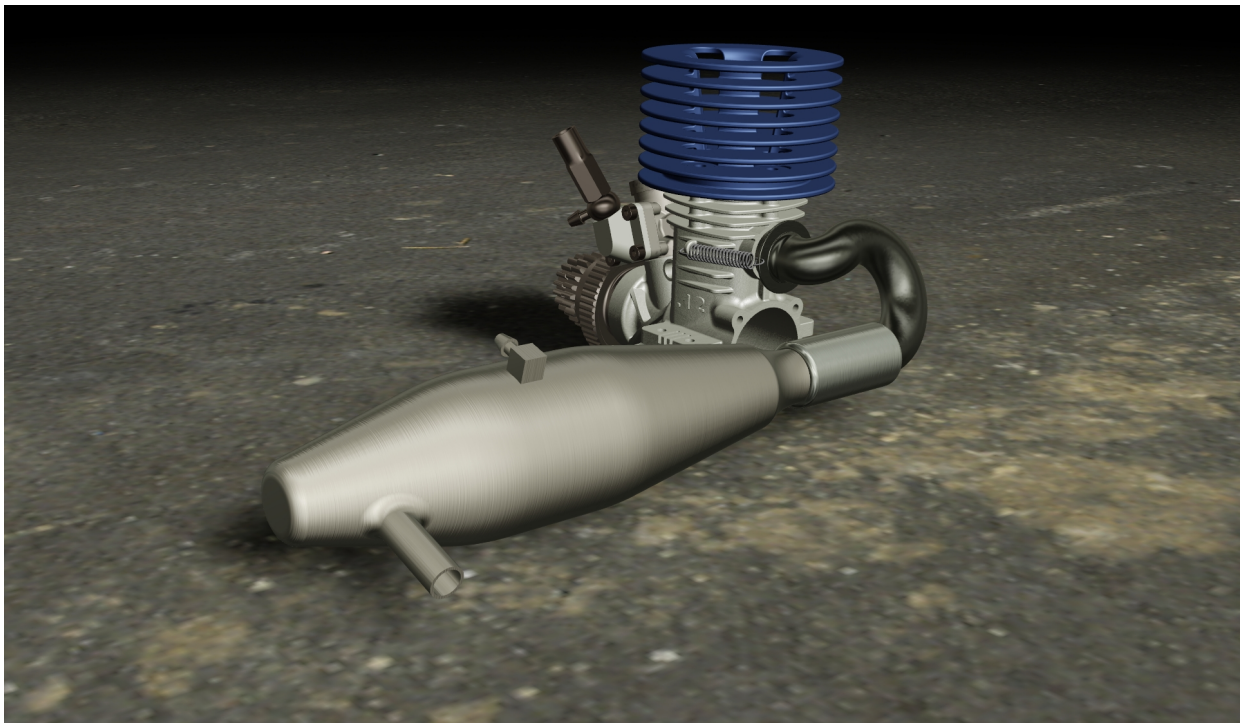
[here](http://mrdoob.github.com/three.js/examples/webgldeferred_arealights.html), and
[here](http://mrdoob.github.com/three.js/examples/webgldeferred_animation.html).

AlteredQualia's work in this area is great, see [his site](http://alteredqualia.com/) for more examples.
]

Lesson: Shadow Mapping

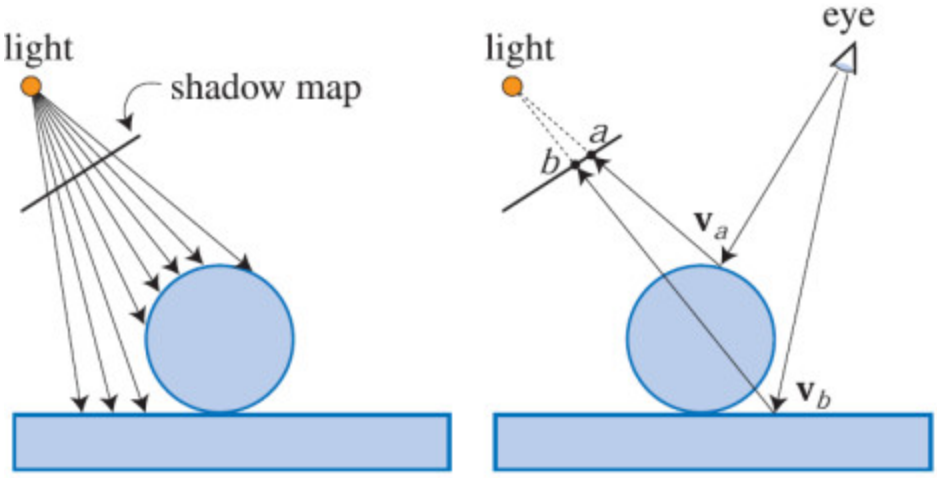
[video: directory Unit6_Shadows, NoShadowsVsShadows.wmv - I want just the non-shadow vs. shadow clips, namely: 0:00 to 0:04 (without ambient shadows) and 0:12 to 0:17, just repeat between those two clips.]





In both Egyptian and Greek mythology, it's said that art was invented by someone tracing the shadow of a person on a cave wall. Shadows give a scene more realism and solidity. An object's shadow firmly establishes its location in relation to its surroundings.

[shadow buffer drawing, showing two objects and light w/shadow buffer.



]

Rasterization is focused on triangles rendered from the eye, so we have to work a bit to generate shadows. One popular method of adding these is called ***shadow mapping***. This algorithm dates way back, it was first described by Lance Williams in 1978. The idea is to render the

scene from the point of view of the light. Whatever the light “sees” is what gets lit. Remember our firefly with human eyeballs? This is where that answer comes into play. If a light doesn’t see a surface, that means the surface is in shadow.

In fact, this is exactly how the basic algorithm is done: the scene is rendered from the view of the light and the distance of the closest object is stored at each pixel.

Because the light has to look in a particular direction and has a limited area it can view, the light source used for shadow mapping is often a spotlight of some sort. A spotlight can be pointed a direction and has a constrained view of the world. In three.js the only lights capable of casting shadows are spotlights and directional lights. For directional lights you specify the limits of how “wide” the light extends.

[Additional Course Materials:
[Here’s](http://cabinetmagazine.org/issues/24/stoichita.php) an interesting article on the history of the shadow, when children first understand shadows, and more.

Lance William’s original article on this algorithm is, “[Casting Curved Shadows on Curved Surfaces](http://artis.imag.fr/~Cyril.Soler/DEA/Ombres/Papers/William.Sig78.pdf)”.
]

Quiz: Shadow Buffer Characteristics

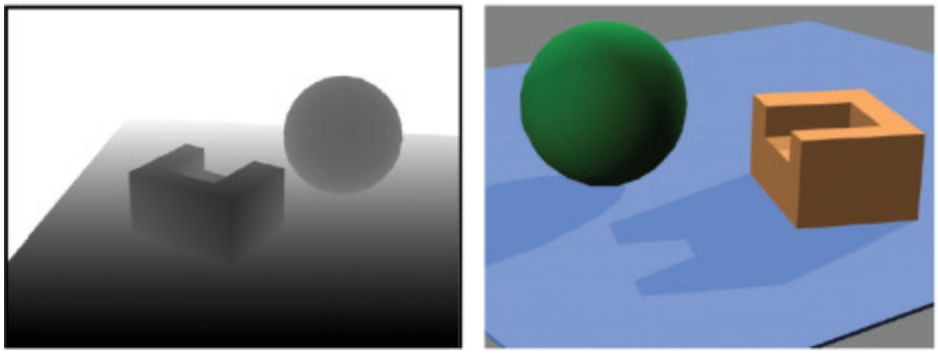
The shadow map algorithm creates a shadow “image” of the scene from the light’s view, called the shadow buffer.

What is this image most like?

A regular scene’s...
( ) ... transparency channel.
( ) ... fragment shader.
( ) ... Z-buffer.
( ) ... color buffer.

Answer

[Add to right of question:]



[z-buffer]

The depths of objects in the scene are stored in the shadow map. This image is most like a regular scene's z-buffer, which stores the distances to objects. On the left is an example of a shadow buffer's stored depths, with black being closer to the light. On the right this shadow map is used to detect from the camera's view which pixels are in shadow and in light.

Exercise: Shadows in three.js

```
renderer.shadowMapEnabled = true;

spotlight.castShadow = true;

cube.castShadow = true;
cube.receiveShadow = true;
```

```
renderer.shadowMapEnabled = true;

spotlight.castShadow = true;

cube.castShadow = true;
cube.receiveShadow = true;
```

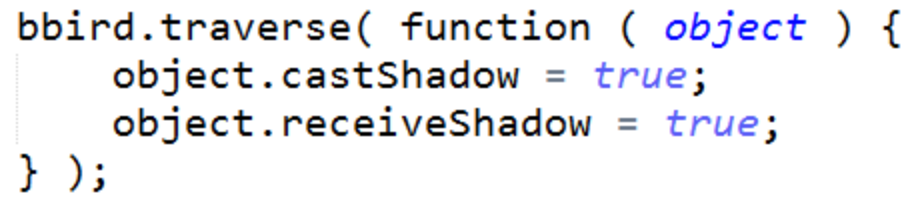
Turning on shadows for a scene is pretty straightforward in three.js. You enable the shadow map on the renderer itself, and tell the spotlight to cast shadows. For each object, you have a choice: you can have it cast or not cast a shadow, or receive or not receive shadows from other objects. For example, if you know a ground plane is never going to cast a shadow, you can save a little processing by not setting the castShadow flag to true for it.

One feature of three.js is that you need to set castShadow and receiveShadow for every piece of

geometry. If you have an Object3D that contains three meshes, you need to set the flags for each of the meshes - setting the flags on the Object3D itself will do nothing. This is actually a constant struggle in scene graph systems: what wins? If I set an attribute on a parent and on a child, does the child's setting win over the parent or vice versa? Three.js avoids the issue by simply not letting the parent have any say.

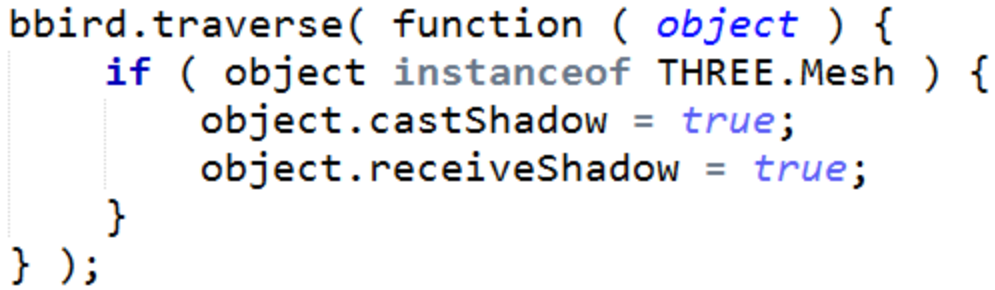
It's tedious to go through every object and have to set these flags, especially for something such as the drinking bird. Happily, three.js has a traversal method you can call that will walk through an object and all its descendants - children, grandchildren, and so on - and do whatever you want with them. Here's the snippet I use:

```
bbird.traverse( function ( object ) {
        object.castShadow = true;
        object.receiveShadow = true;
} );
```



This code sets the castShadow and receiveShadow flag true on every object that makes up the drinking bird. It's actually a doing a bit too much, as setting these flags on objects without meshes doesn't do anything. Here's better code:

```
bbird.traverse( function ( object ) {
        if ( object instanceof THREE.Mesh ) {
                object.castShadow = true;
                object.receiveShadow = true;
        }
} );
```



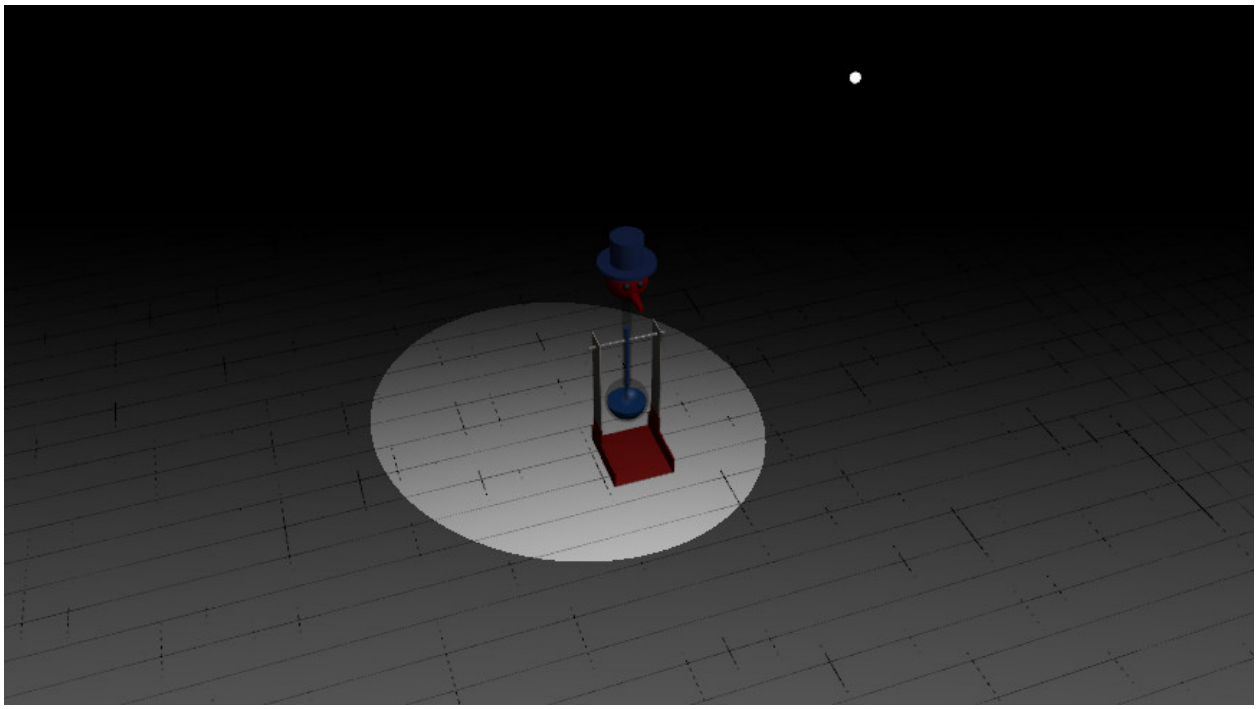
This uses some real-live JavaScript, checking if an object has a mesh. If it does, then the shadow flags have meaning for it, so we set them.

[show solution, and zoom out to show spotlight location - shown at end]

Give it a try right now, turning on shadows for the spotlight you added to the drinking bird scene. I've kindly set the cast and receive shadow flags for the drinking bird itself, but not for the ground plane. Also, I'm leaving it up to you to figure out where the various calls to enable shadowing go.

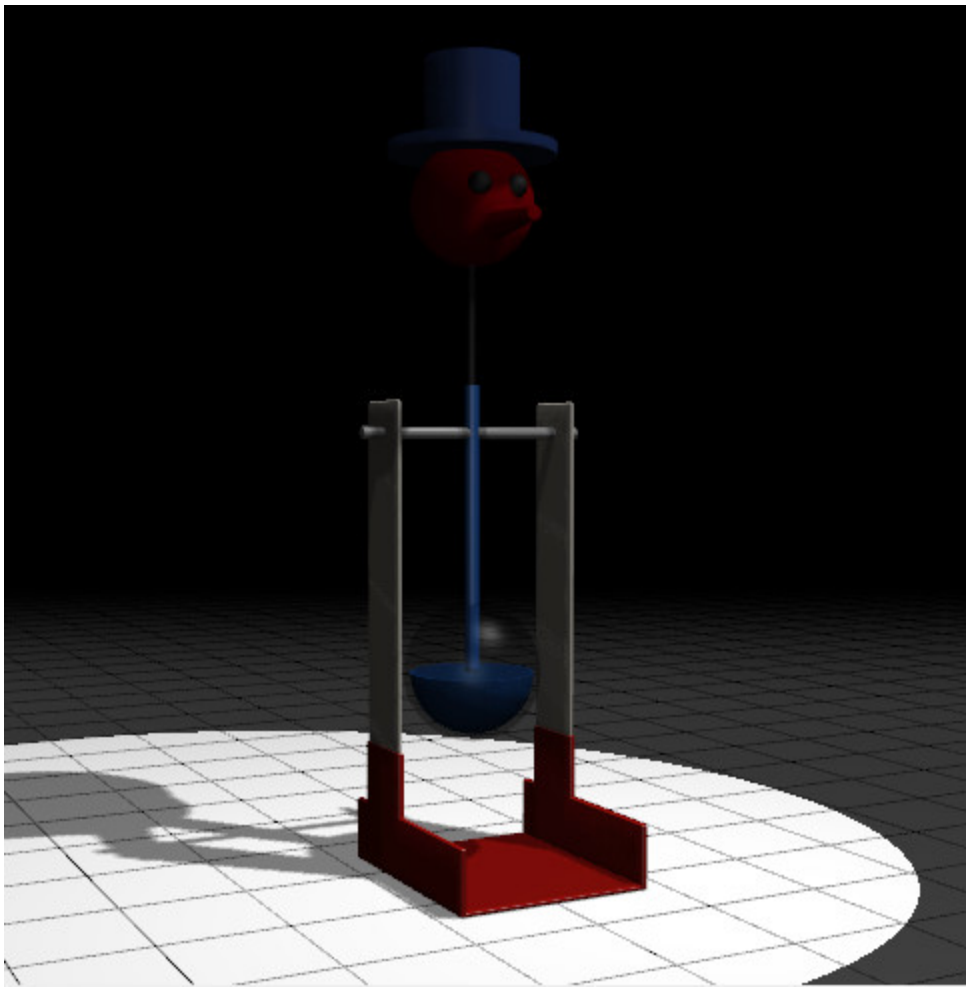
For this exercise I've also added a little sphere representing the light source in the scene. If you move out you'll see it hanging in the air. Doing this for your lights is a handy way to get a better sense of where you're placing them.

[Exercise at unit6-shadow_exercise.js]



[Additional Course Materials:
Try out the "shadow bias" slider once you have this exercise working. I'll discuss this value briefly in the next lesson, but it's a good one to get a feel for by trying out yourself.
]

Answer



There are a few small bits of code to be added. To the spotlight itself you need to add:

light.castShadow = true;

```
light.castShadow = true;
```

The solidGround object was not receiving any shadows, so you need to add:

solidGround.receiveShadow = true;

```
solidGround.receiveShadow = true;
```

You don’t need to have the ground cast a shadow, since it’s below everything else.

Finally, the renderer itself must have shadows enabled:

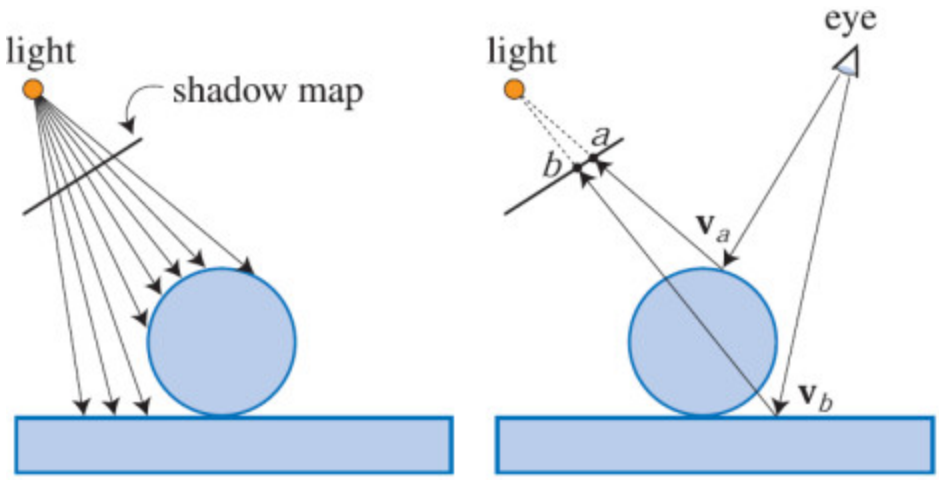
renderer.shadowMapEnabled = true;

```
renderer.shadowMapEnabled = true;
```

which is added to the list of renderer initialization settings.

Take a look and examine the legs of the model, I expect you'll see some odd shading on them. If you make the headlight brighter the effect will become more obvious. Moving the spotlight is likely to shift these patterns around.

Lesson: Shadow Buffer Limitations



The shadow mapping algorithm works by seeing the world from the light's point of view. The depth image formed is then checked during normal rendering. The depth of the surface visible at a point is compared with the transformed depth that the shadow map buffer holds. And that's where all the trouble begins.

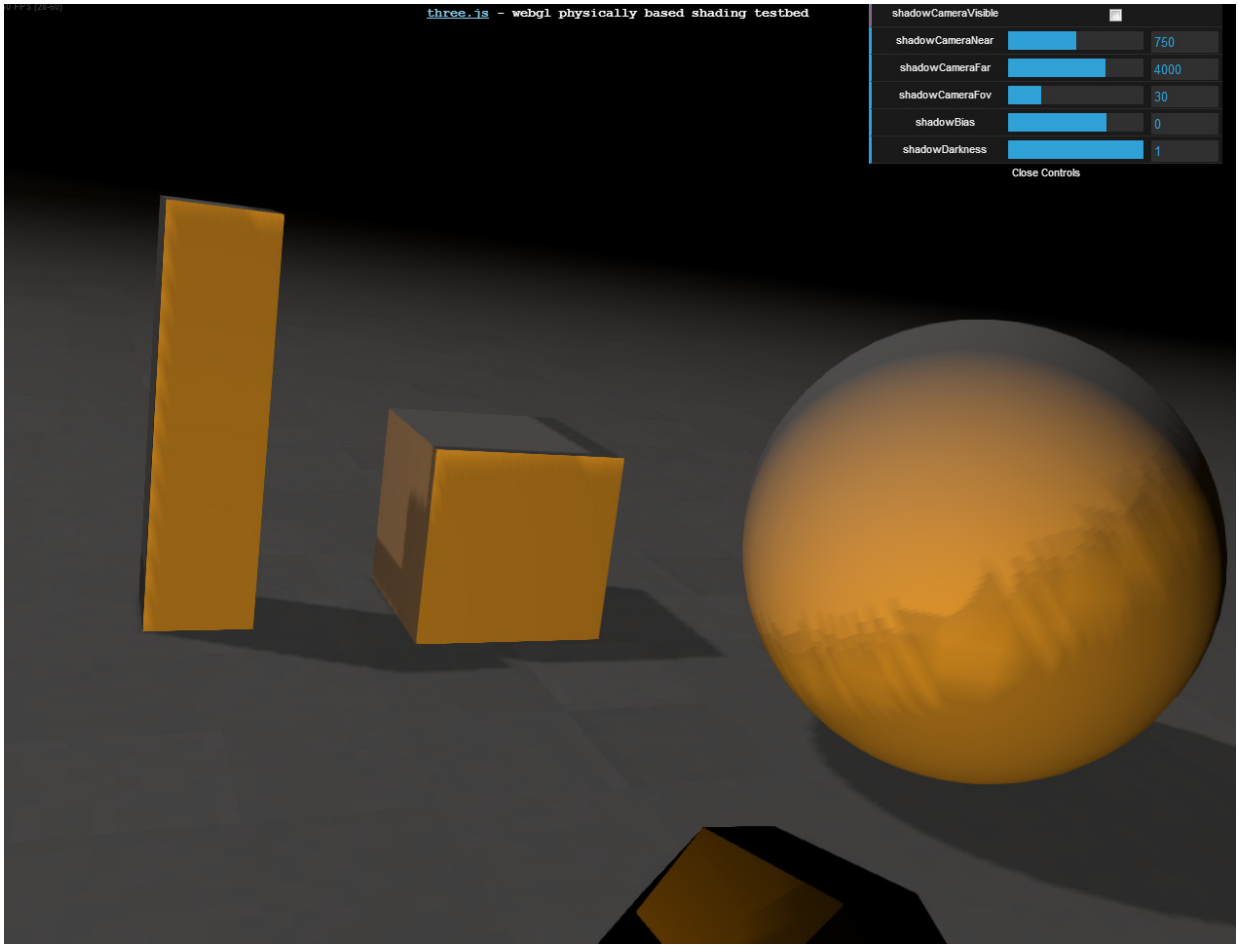
The pixels in the light's image of the world don't exactly match up with those in the camera's view. These cause some form of what's called "surface acne", where little dark dots or patches appear in areas that should be fully lit.

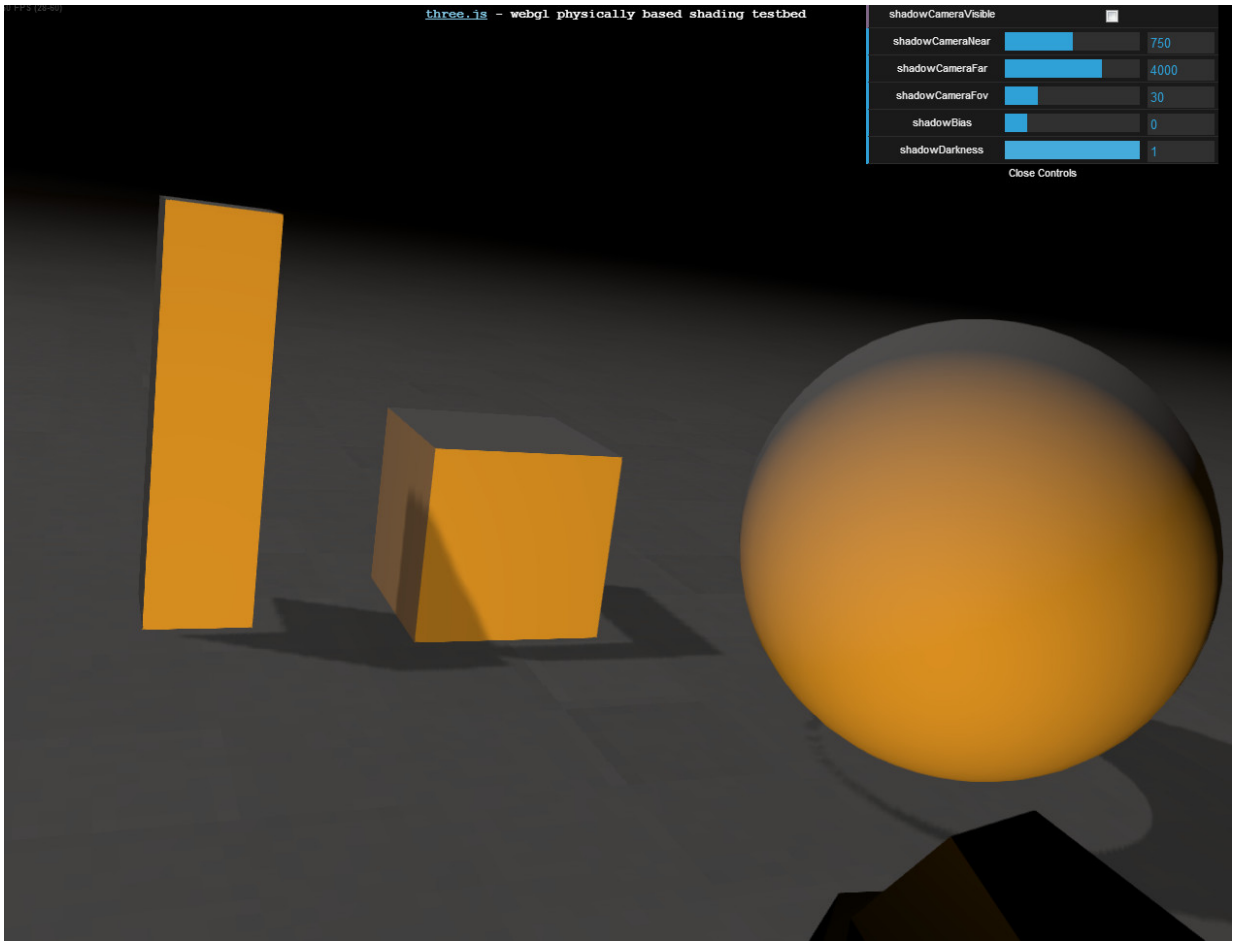
[draw distances on image]

Take this example. The light's view of the surface says it's 3.80 meters away from the light. The camera's view of the surface says the location is 3.81 meters away from the light. Since this value is slightly higher than the light's view, the surface is considered in shadow. The surface

does something that should be impossible: it shadows itself, as the depths don't compare as well as we would like.

[http://mrdoob.github.com/three.js/examples/webgl_shading_physical.html - EXCELLENT demo for doing shadow map demo and for showing the various problems. Turn on view frustum. Play with all the values (the defaults don't show shadows on). Crank shadows to darkest.
]





Here's a demo showing the problem, as well as some solutions. On these surfaces you can see patterns of acne, where the geometry is shadowing itself. One obvious solution is to add some tolerance value, called a shadow bias: if the light's depth plus this bias is greater than the camera's computed distance along the light's view, then consider the surface lit.

Unfortunately, a simple bias leads to another problem, called "Peter Panning". Just like Peter Pan, the object starts to look as if it's becoming detached from it shadow and sometimes as if its hovering above its true location.

In the 35 or so years this algorithm has been around, many different solutions have been proposed to fix this problem. There are other problems with shadow mapping, such as the edges of shadows looking pixelated, and solutions such as percentage closer filtering and cascaded shadow maps. I won't explain these here except to say that three.js has support for both of these solutions built in.

[Additonal Course Materials:
To try out the three.js demo and see how various shadow map settings affect the quality, [try this

demo](http://mrdoob.github.com/three.js/examples/webgl_shading_physical.html) yourself. For tutorial code showing how to set shadows, see [Lee Stemkoski's program](http://stemkoski.github.com/Three.js/Shadow.html). For more elaborate shadow map settings, see the (incomplete) documentation for the three.js [ShadowMapPlugin](http://mrdoob.github.com/three.js/docs/57/#Reference/Renderers/WebGLRenderer).

One promising technique to fix self-shadowing problems is to use [normal offsetting](http://www.dissidentlogic.com/#Normal).

The book "[Real-Time Shadows](http://www.amazon.com/Real-Time-Shadows-Michael-Wimmer/dp/1568814380?tag=realtimerenderin)" is entirely about this subject; there is also [a free precursor of the book](http://www.mpi-inf.mpg.de/resources/ShadowCourse). Our own book "[Real-Time Rendering](http://www.amazon.com/Real-Time-Rendering-Tomas-MOller/dp/1568814240?tag=realtimerenderin)" has a fair bit about shadow algorithms and their problems and solutions.

You can read more about the surface acne problem and many others in this article, "[It's Really Not a Rendering Bug, You See...](http://www.geekshavefeelings.com/x/wp-content/uploads/2010/03/Its-Really-Not-a-Rendering-Bug-You-see....pdf)". My contribution to computer graphics is coining the term "[surface acne](http://tog.acm.org/resources/RTNews/html/rtnews2c.html#art1)".

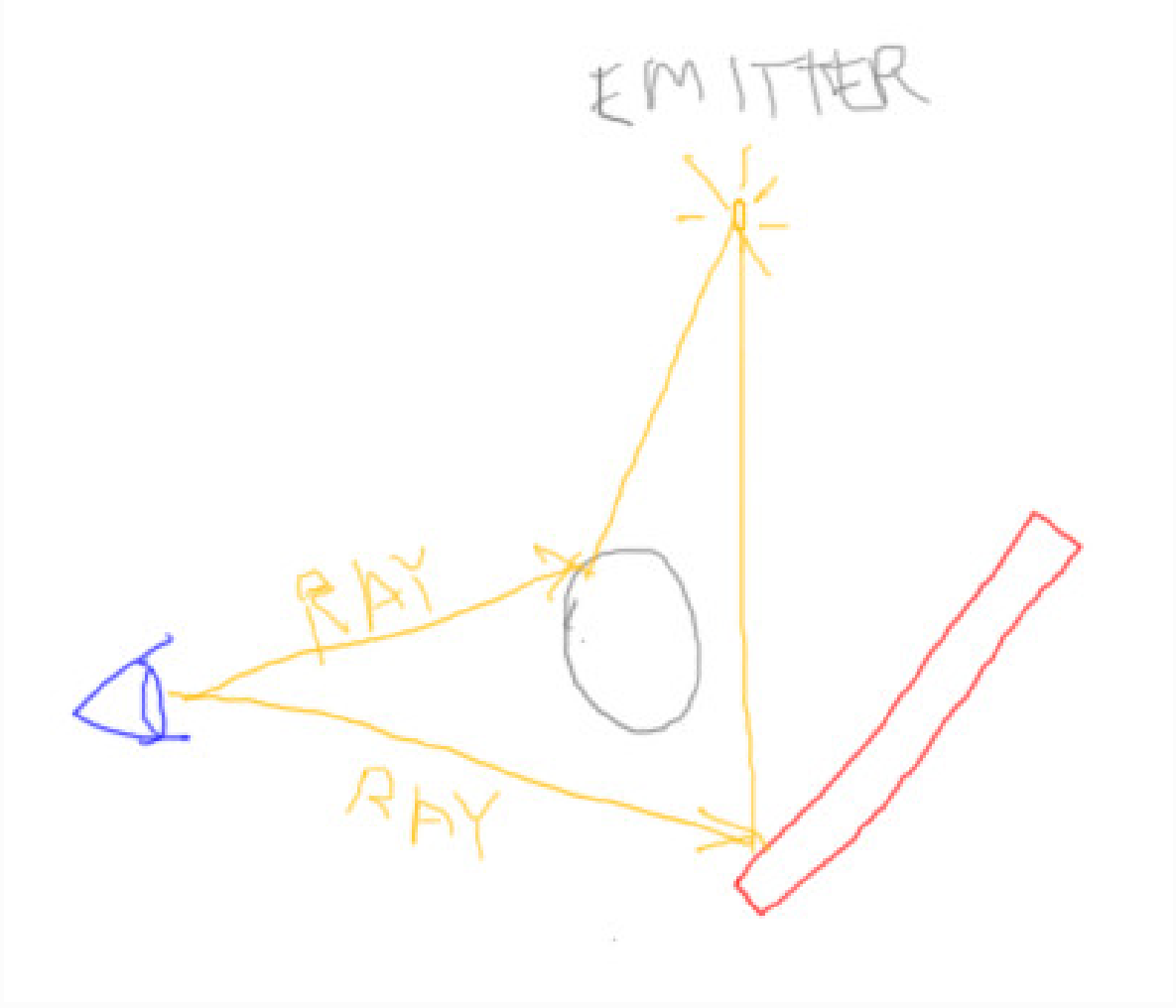
A nice quick summary of the various shadowing algorithms used in interactive rendering is [here](http://cis565-fall-2012.github.com/lectures/10-24-Shadows.pdf).
]

[recorded session 3 end 3/26]

Lesson: Ray Tracing

[recorded session 4 starts, 3/26]

[add paths to various given paths from previous lesson.]



So far we've examined what are called "***local illumination***" models, where an object is affected by a light and the result is sent to the eye. Light comes from only, well, light sources. No light is reflected from other objects.

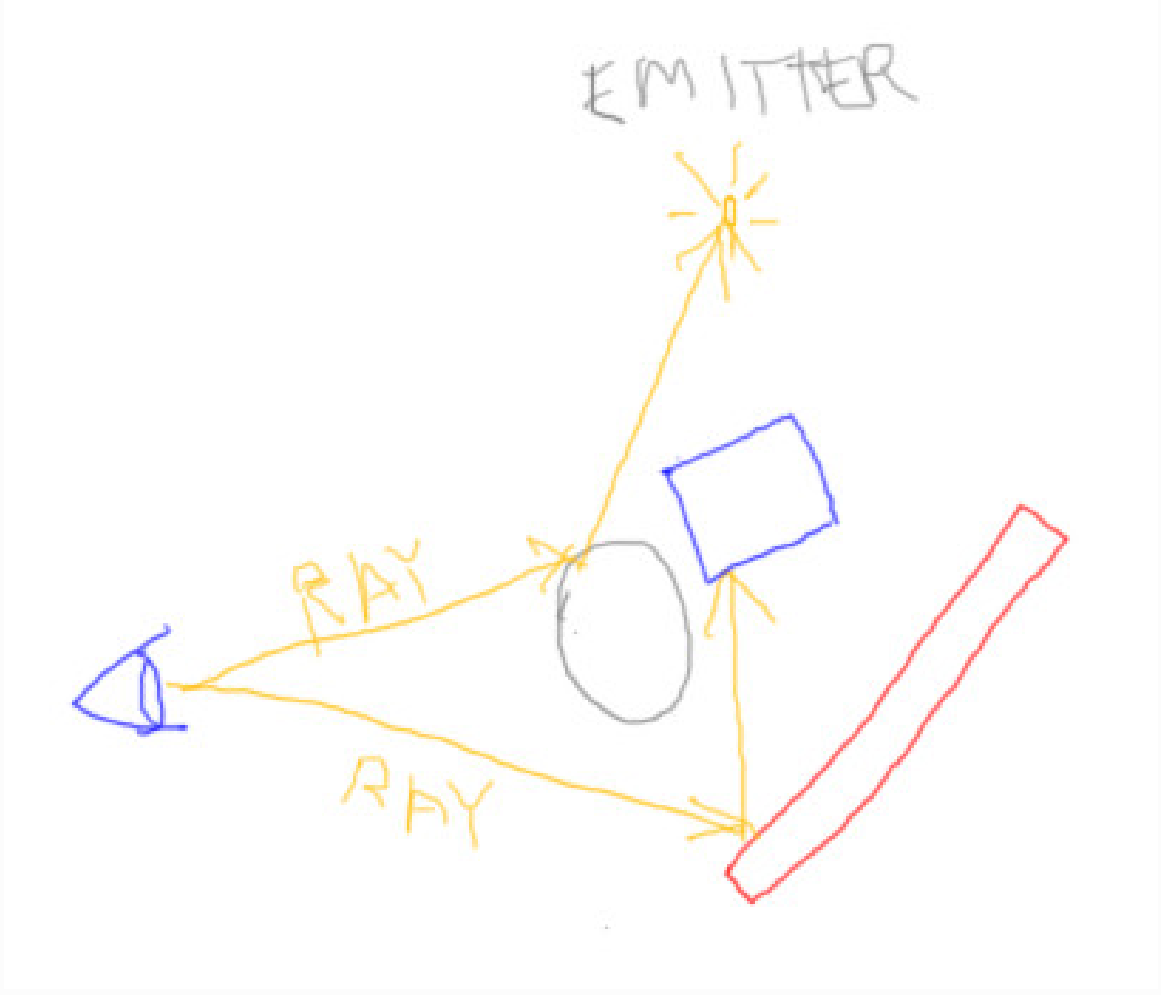
There are many more light paths that *can* be tracked. One rendering technique that can simulate these is called "ray tracing". We've seen how the GPU sends each object triangle in turn to the screen and rasterizes it. Ray tracing instead fires rays from the eye through each pixel. Objects are tested against each ray. The closest object found along a ray's path is the one that is used to compute the ray's color contribution.

You can think of each ray from the eye as rendering a single 1x1 pixel. One way to find the closest object for this pixel is to send every triangle down a pipeline and attempt to rasterize it, using the z-buffer to keep track of the closest object. This is perhaps the slowest way *ever* to perform ray tracing. In reality researchers and practitioners have spent a huge amount of time creating and tuning algorithms to rapidly find this closest object along the ray. Where rasterization gets some of its speed is from the fact that a single triangle usually covers a bunch

of pixels. Various intermediate results can then be shared when processing the triangle.

At its simplest, ray tracing can give the same result as rasterization: each ray from the eye finds the closest object along it. The effect of light on the surface is computed and the result is displayed. However, that's just the starting point for ray tracing.

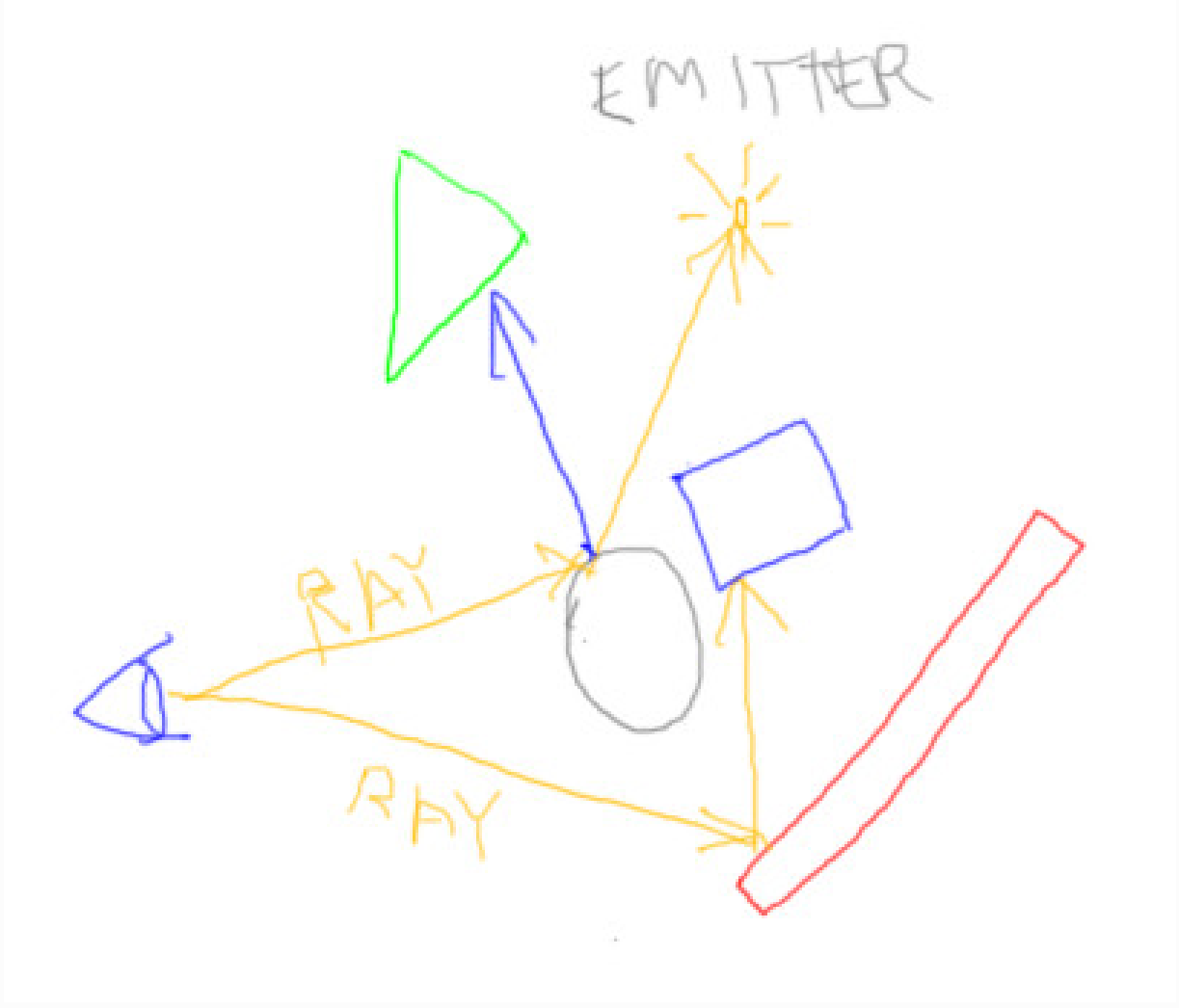
[blocker - NOTE: make this blocker so that one face faces the emitter and the eye, for later use.]



For the light to surface paths we've made one obvious simplification for basic rasterization: objects don't block other objects. In other words, no shadows are cast. In ray tracing, adding basic shadows is trivial: shoot a ray from the surface to the light. If there's something in the way, the light is blocked and can be ignored. Since all computations happen in world space, we can avoid many of the precision problems found in rasterization techniques such as shadow mapping.

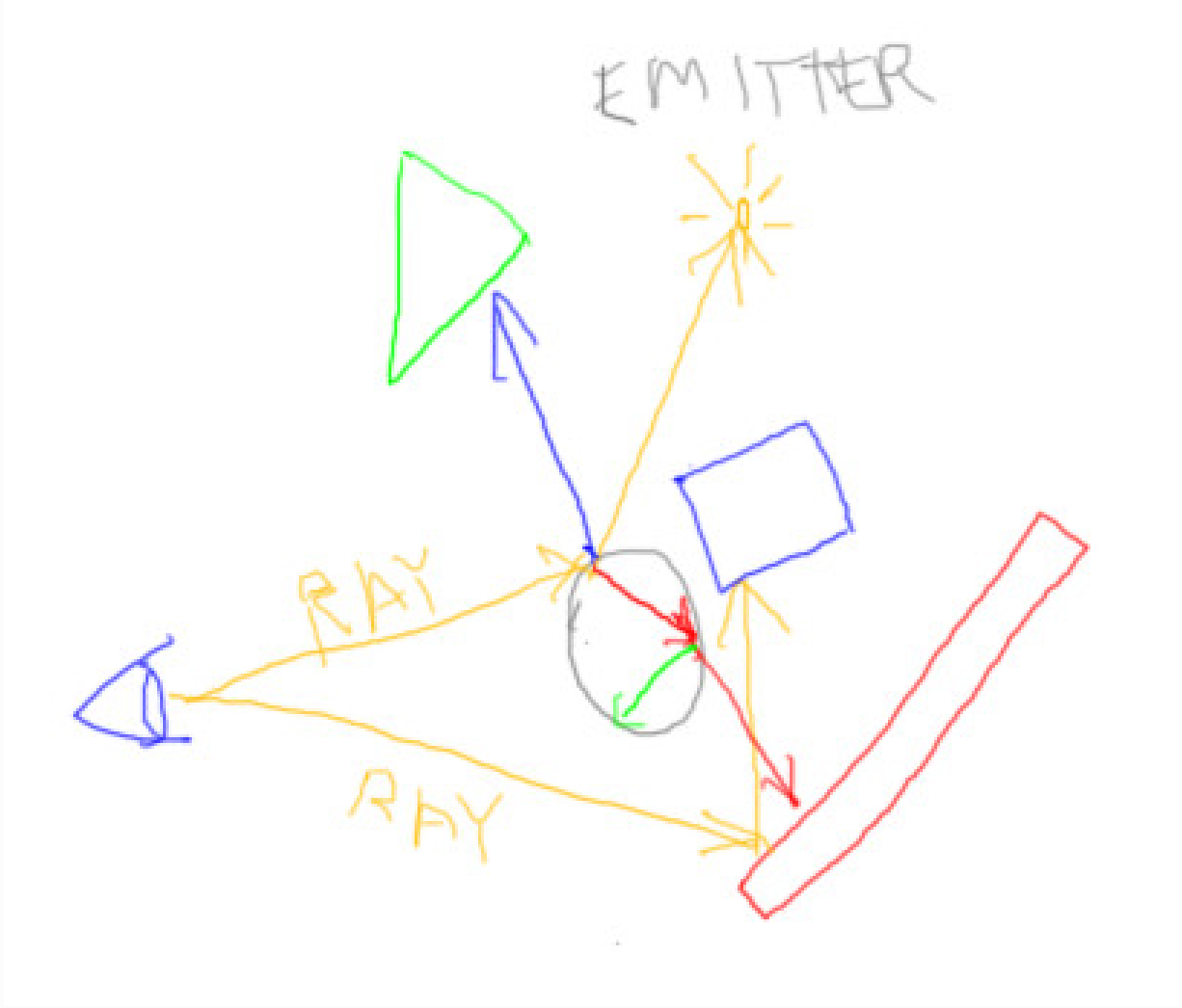
[reflectance - Note, make ray to green object be green, implying it picks up the green object's

color]



Classical ray tracing also offers us the ability to create true reflections and glass effects. For shiny surfaces, we can spawn a new ray in the reflection direction. Whatever this ray hits is what is reflected in the surface and so can be factored into the ray's final color, its ***radiance***.

[refraction]



Let's say our sphere is actually glass. Instead of simple filtering, we can also spawn a refraction ray and see where it hits. When it encounters the other side of the sphere and exits, we spawn yet another refraction ray and reflection ray. We can continue to spawn rays until we hit some limit.

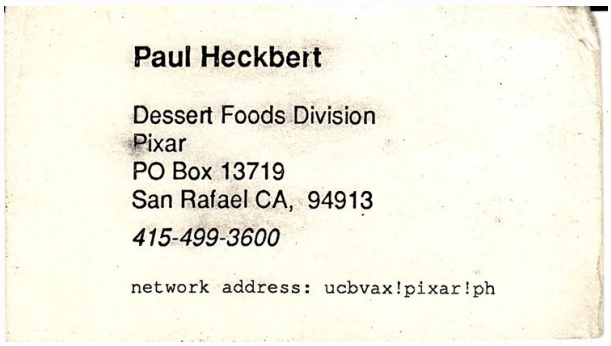
[Additional Course Materials:
There are interactive WebGL ray tracers out there, such as [this one](http://people.mozilla.com/~sicking/webgl/ray.html) and [this](http://frog.ivank.net/).

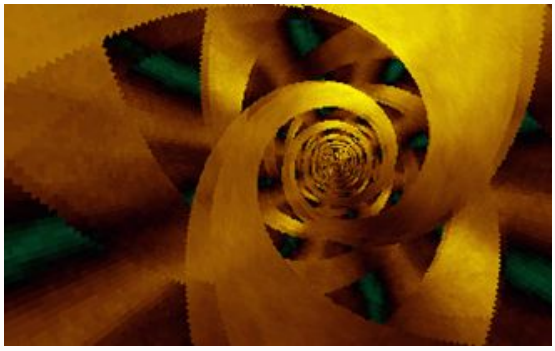
Currently my favorite book for learning about ray tracing is ["Ray Tracing from the Ground Up"](http://www.amazon.com/Ray-Tracing-Ground-Kevin-Suffern/dp/1568812728?tag=realtimerenderin).I gave a talk comparing the strengths and weaknesses of ray tracing and rasterization [here](http://erich.realtimerendering.com/RT08.pdf]; related links are at [my own site](http://erichaines.com). I should mention I edited an [informal journal about ray tracing](http://raytracingnews.org) for 20-odd years.

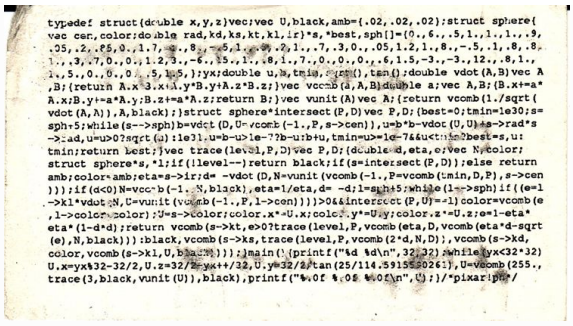
]

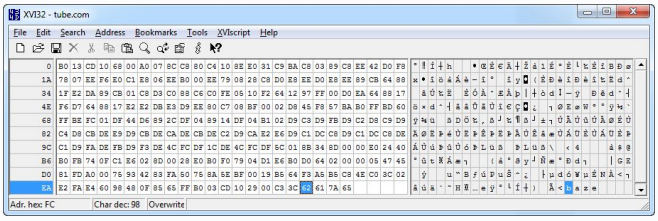
Lesson: Ray Tracing History and Future

[Heckbert's card and the BAZE ray tracer]









There are whole books written about this algorithm, though at its simplest it can fit on the back of a business card, such as this old one of Paul Heckbert's. There are even ray tracers with amazing animated effects that are written in 256 bytes, such as this demoscene program called "Tube". Well, actually, this program is only 252 bytes, the last four bytes spell out the author's name, Baze.

[show a nice ray trace,]

[http://en.wikipedia.org/wiki/File:Glasses_800_edit.png or
http://commons.wikimedia.org/wiki/File:Tpe10-bloom-f25-l7-1920.jpg]

This is as far as I'm going to go with ray tracing, but you now have a taste of its power. Shadows, sharp reflections, and refraction can all be added easily.

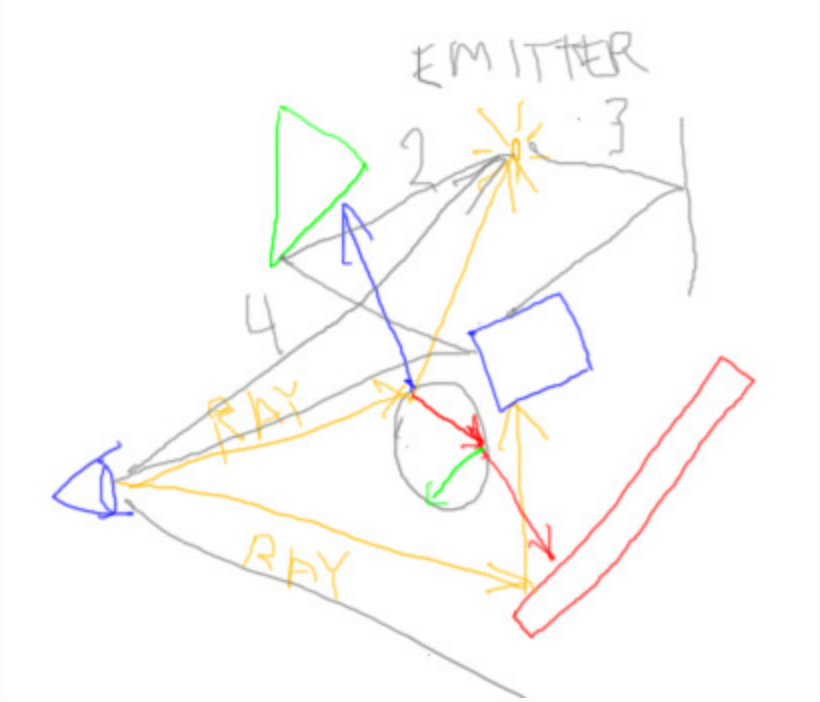
There's been much work to make ray tracers generate images at interactive rates for complex scenes. We're not quite there yet for some applications, and there are headaches with keeping a constant frame rate and reasonable quality, but we'll clearly see more progress in this area in the future.

[Additional Course Materials:
The 256 byte BAZE demo mentioned can be downloaded from [here](http://baze.au.com/) and seen in action [here](http://www.youtube.com/watch?v=3IrWzeTL5EE). This demo is a part of [the Demoscene](http://en.wikipedia.org/wiki/Demoscene) - as it turns out, the [author of three.js was also involved with and inspired by the demoscene](http://www.realtimerendering.com/blog/interview-with-three-js-creator/).
]

Question: What's Missing?

Classical ray tracing gives us a lot of paths from the eye into the environment. However, there are various paths ignored by this basic algorithm.

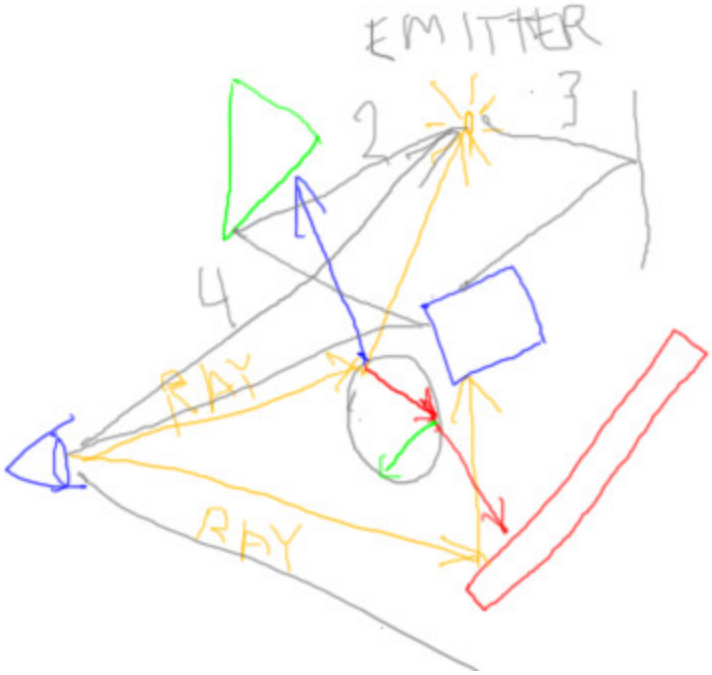
[draw paths in different colors - do it better than I do it here...]



What paths are ignored?

Paths from the …
[do each in a different color and make that path on the diagram]
[ ] eye to the background.
[ ] light to an object to a diffuse object.
[ ] light to a mirror to an object.
[ ] eye to a light itself.

Answer



The path from the eye to the background is pretty trivial and easily handled. An empty scene is about the easiest thing to render, so this path is not a problem.

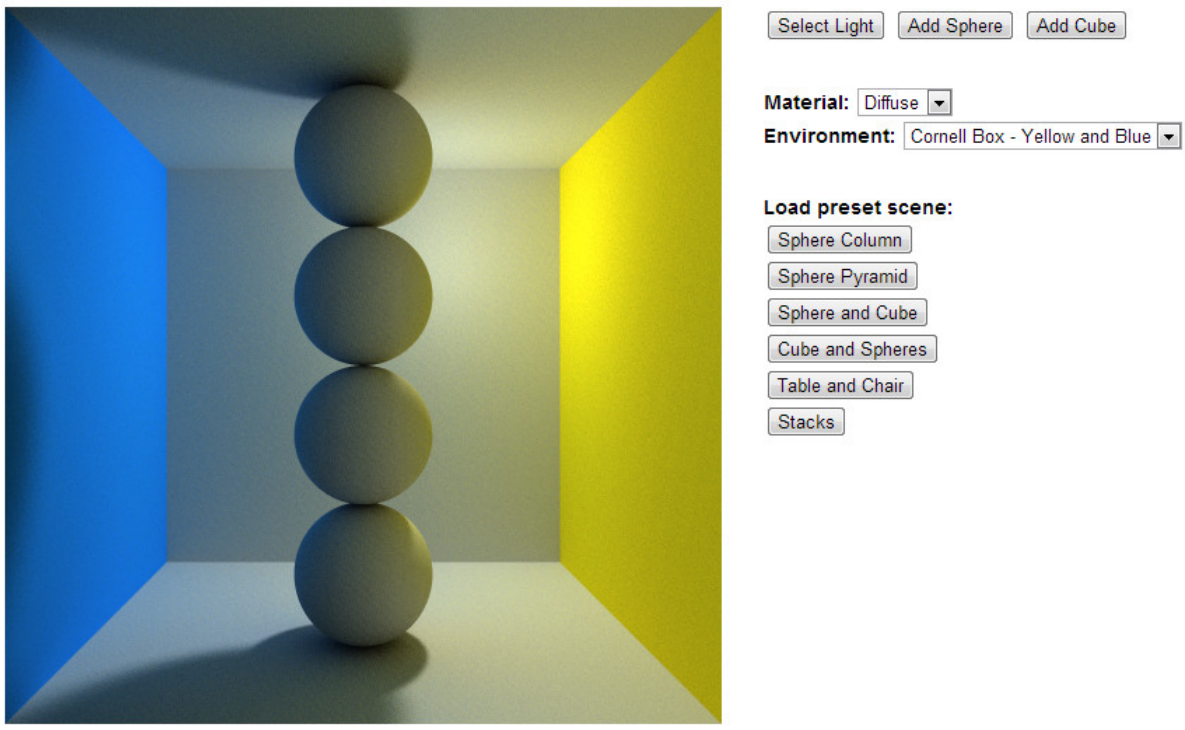
The path from a light to any object to a diffuse object is not well-handled by ray tracing from the eye. Objects can be reflective, but when the light's contribution to a visible surface is computed, only paths directly from the light to the surface are checked.

Imagine you're viewing the world from a point on a surface. If you think about it, just about every surrounding visible object is reflecting light at the surface, not just the light sources in the scene. Determining which paths could possibly contribute more light is difficult. Any paths from the light source are ignored with classical ray tracing, as there are a huge number of them.

The fourth case is easy to handle: even rasterization typically handles lights by giving them a fixed solid color, and classical ray tracing can treat them in the same way.

Lesson: Path Tracing

[Show http://madebyevan.com/webgl-path-tracing/ with Evan Wallace's permission.]



[think of what to show in demo]

One solution to tracking the various paths that light can take to reach the eye is to shoot a lot more rays per pixel and bounce them around in hopes of finding light sources. This is called path tracing. It can be very noisy to start with, but can give correct results with enough time. This is a demo by Evan Wallace in WebGL, one I recommend you try yourself. Notice the “color bleeding” effect of light bouncing off of the walls onto the spheres.

[switch to glossy]

You can also make objects have glossy reflections instead of sharp ones.

This demo uses what is called “progressive rendering”, shooting more and more rays for each pixel and blending in the results. The longer you wait, the better the image gets.

You can even use path tracing to render scenes from Minecraft. Here's one I made using the free Chunky path tracer, giving a beam of light effect on a scene. A basic, pure path tracer is pretty straightforward to write: you generally just shoot more rays in sensible directions and sum up the light contributions found. Where all the time gets burned is that you're shooting tens of thousands of rays per pixel or more. This scene took about 16 hours with 12 CPU threads, and it was still a bit noisy at that point.

I should mention there are many other algorithms, such as photon mapping and bi-directional path tracing, that work to get the best of both worlds. The general idea is to send light out from emitters using rays, depositing radiance wherever the rays reach. The scene is then ray traced from the camera's view and the emitted light is gathered.

Algorithms such as path tracing can give unparalleled realism, given enough time. Set up properly, a rendering can be a true simulation of how light percolates through a scene, not just an artistic approximation.

I used to work on a global illumination system back in the 80's and 90's. Even back then we knew that path tracing could, given enough computer cycles, get the right answer. The running joke was that we were mostly just biding our time, coming up with optimizations for other algorithms while waiting the 50 years needed for computers to get powerful enough to shoot 10,000 rays per pixel to get the right answer. Nowadays I think you probably need more like 100,000 or a million rays per pixel. But, by my watch, I just need to wait another quarter century for rendering to be over. Assuming people don't put more objects in their scenes.

[Additional Course Materials:
Try out the [WebGL path tracer](http://madebyevan.com/webgl-path-tracing/) by Evan Wallace.

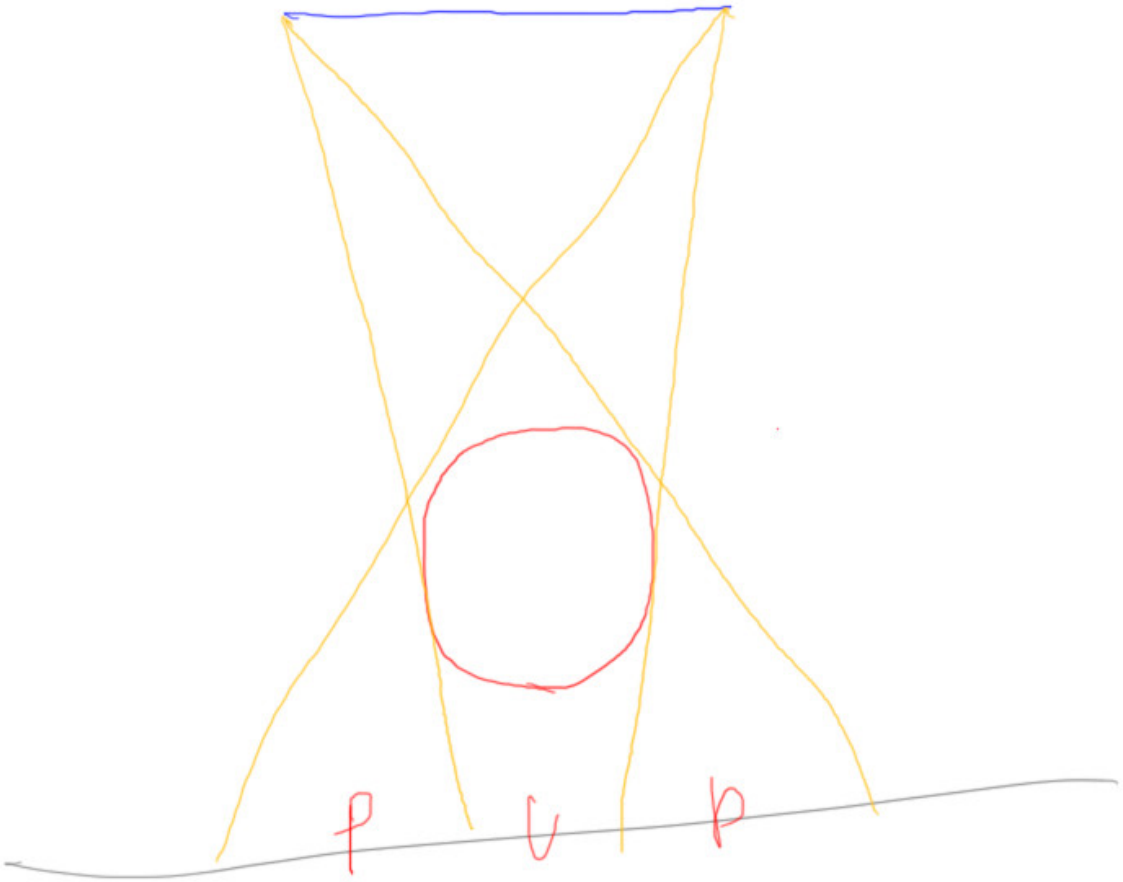
Wikipedia has articles on [path tracing](http://en.wikipedia.org/wiki/Path_tracing) and [photon mapping](http://en.wikipedia.org/wiki/Photon_mapping), thouse these tend to be a little too technical for my tastes.

A link to the Chunky renderer and some sample renderings can be found [here](http://www.realtimerendering.com/erich/minecraft/public/chunky/jpegs/). If you play Minecraft, try it yourself on your own world!
]

[end recording session 4 on 3/26]

Lesson: Umbra and Penumbra

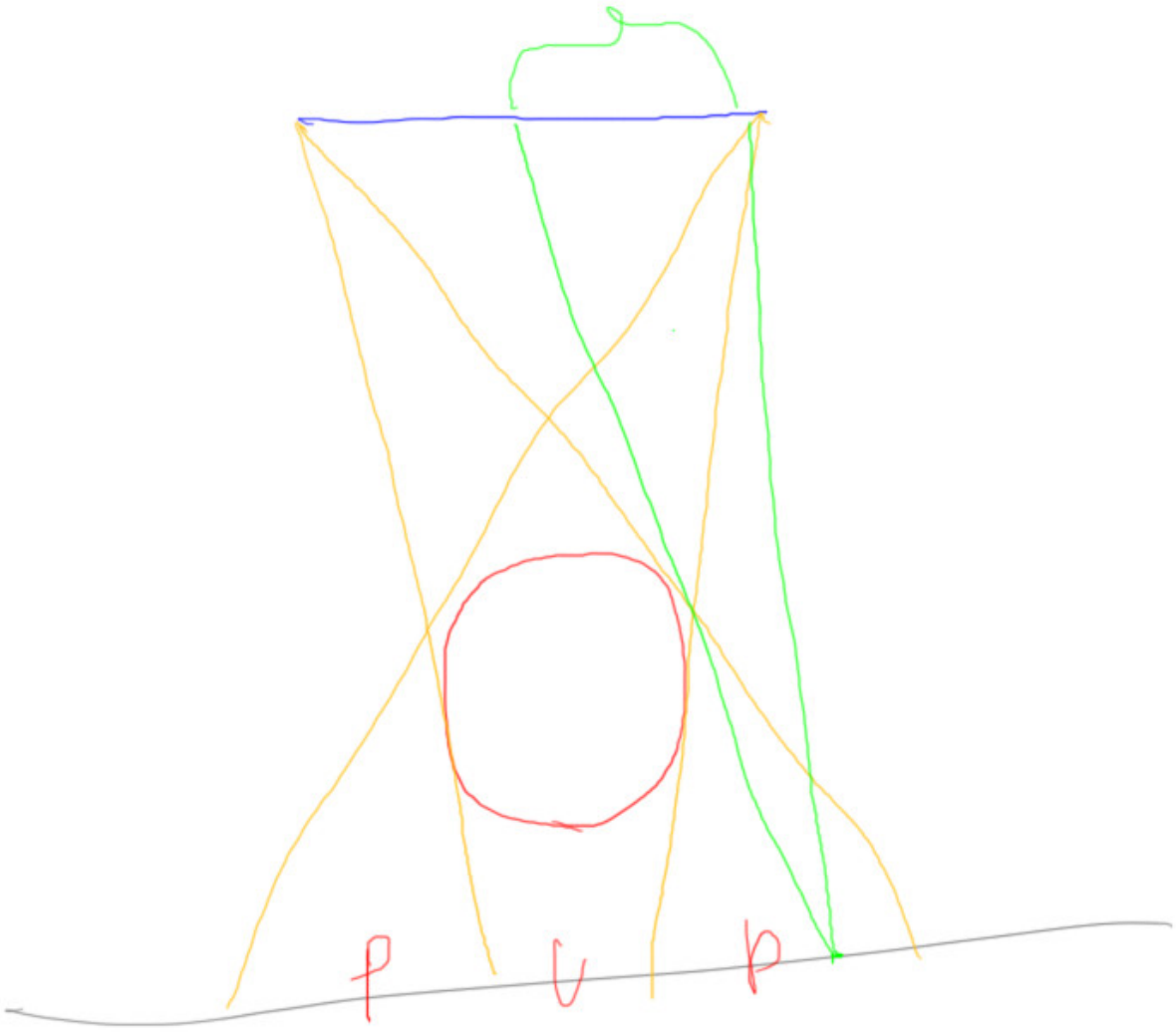
[recorded 3/27]
[umbra and penumbra - Label the top blue thing as the emitter!]



One feature of shadows that is not captured well by basic shadow mapping is how lights truly cast shadows. All lights have an area, but shadow mapping assumes the light comes from an infinitesimal point. Unless the real light itself has an extremely small filament and no diffusing bulb surface, treating it as a point of light is a weak approximation at best.

The area fully in shadow, where no part of the light is visible, is called the umbra. Real lights cast shadows with areas that are neither fully in shadow nor fully in light. This area of the shadow is called the penumbra.

[show “view” from penumbra]



An interesting way to look at the problem is from the point of view of a location in the penumbra. However much of the area of the light is visible correlates to how much light the location receives. For example, the green point on the ground plane here can “see” about half of the emitter above, so will receive about half the emitter’s light. You have to worry about the angle of the surface normal to the light, the light’s distribution of energy, and so on, but the major factor is how much of the light’s surface is visible to each point.

[draw some place twice as far away = less light]

[Leave this off, I think - reviewers? Interesting, but not really critical: *This leads to an interested related fact. We all learned in school that the effect of a light falls off with the square of the distance. If you get twice as far away from a light, it sheds one fourth the illumination on you than it previously did.*

It's actually a bit more nuanced than that. The effect of a light on a location really is a function of how much area that light takes up compared to everything else. Imagine looking at a light through a screen door. You count the number of squares covered by the light. Move the light to be twice as far away and the number of pixels drops to about a quarter of this number. When the light has a large enough area, that is, when it's close enough to the viewer, the square of the distance law falls apart and the area of a sphere surrounding the viewer is more directly correlated.]

Interview: David Larsson

[Headshot introduction, recorded 4/2]

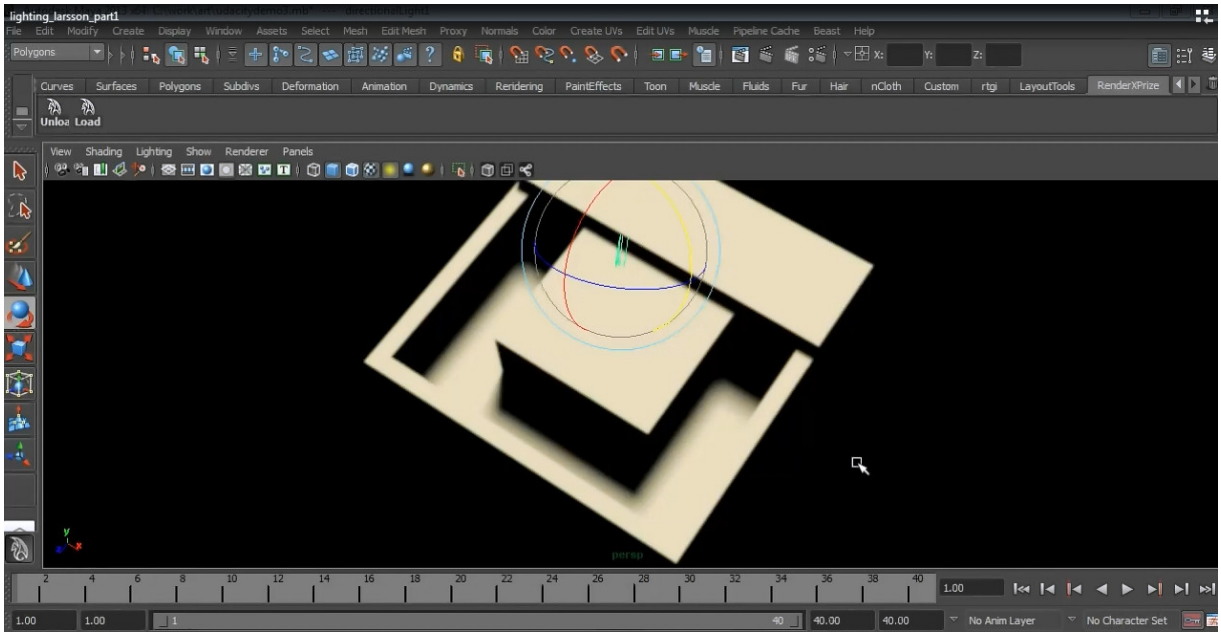
"We cover three.js's built-in lighting capabilities in this unit. However, more elaborate forms of computing illumination are certainly possible. To give you a glimpse at some of these higher-end lighting techniques, I asked David Larsson to create a few lessons using the lighting system he helped develop."

[now include video of David Larsson intro: Unit6_DavidLarssonInterview directory, sd-David_01_background.mp4. Cut initial 0:08 off of this video, start with "Many of the light demos you'll see in this unit..."
]

[Then include video of David Larsson interview sd-David_05_beastingames.mp4 - I'd just make this all one clip, since this last video is short.]

Lesson: Shadow Summary

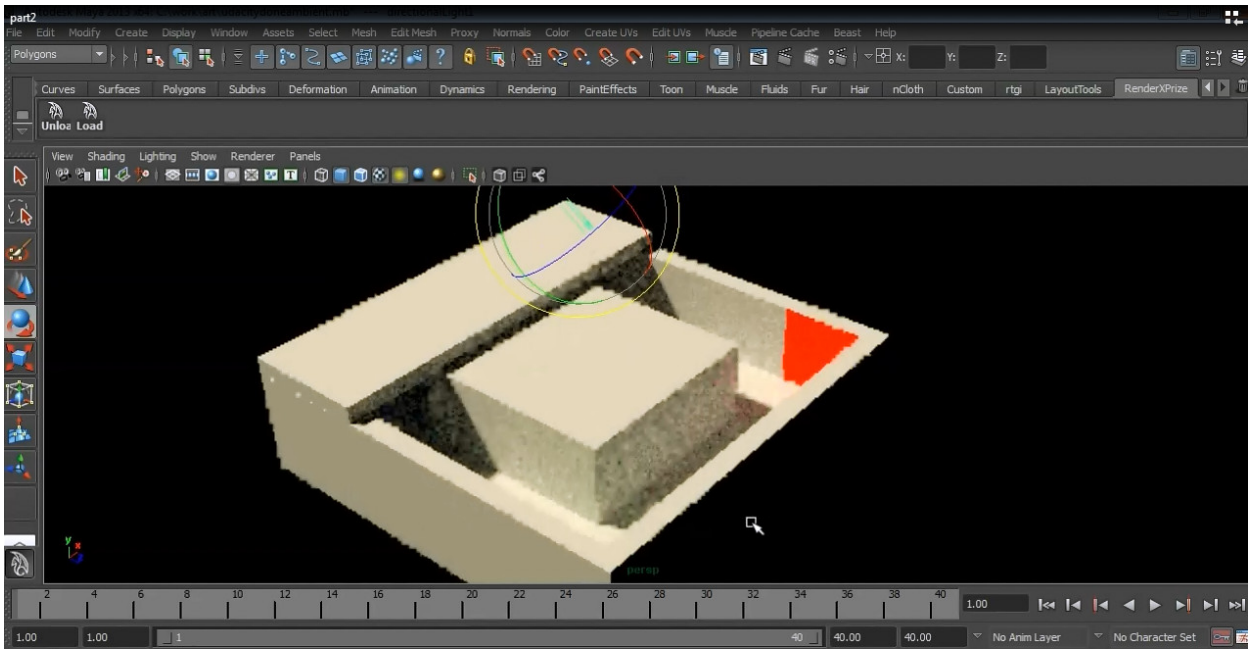
[Video, Unit6_UmbraAndPenumbraPost, lighting_larsson_part1.mp4 - Video with sound.]



[Let David talk]

Lesson: Indirect Lighting

[Video, Unit6_IndirectLighting, part2.mp4 - play whole video & sound as lesson]



Question: Which are Global?

Check all of the following that require global illumination.

[ ] Light through stained glass hitting the floor.
[ ] A Lambertian material, such as Spectralon.
[ ] Patterns of light at the bottom of a swimming pool.
[ ] A surface evaluated using the Blinn-Phong Illumination model.

Ultimately, all materials can be globally illuminated. What I'm looking for here are those situations where global illumination techniques are required.

Oh, and what's Spectralon? See the additional course materials if you want to know about this material (though it's not all that important).

[Additional Course Materials:
I'd never heard of [Spectralon](http://en.wikipedia.org/wiki/Spectralon) before, it's a highly diffuse material; I found it in looking for examples of Lambertian materials.
]

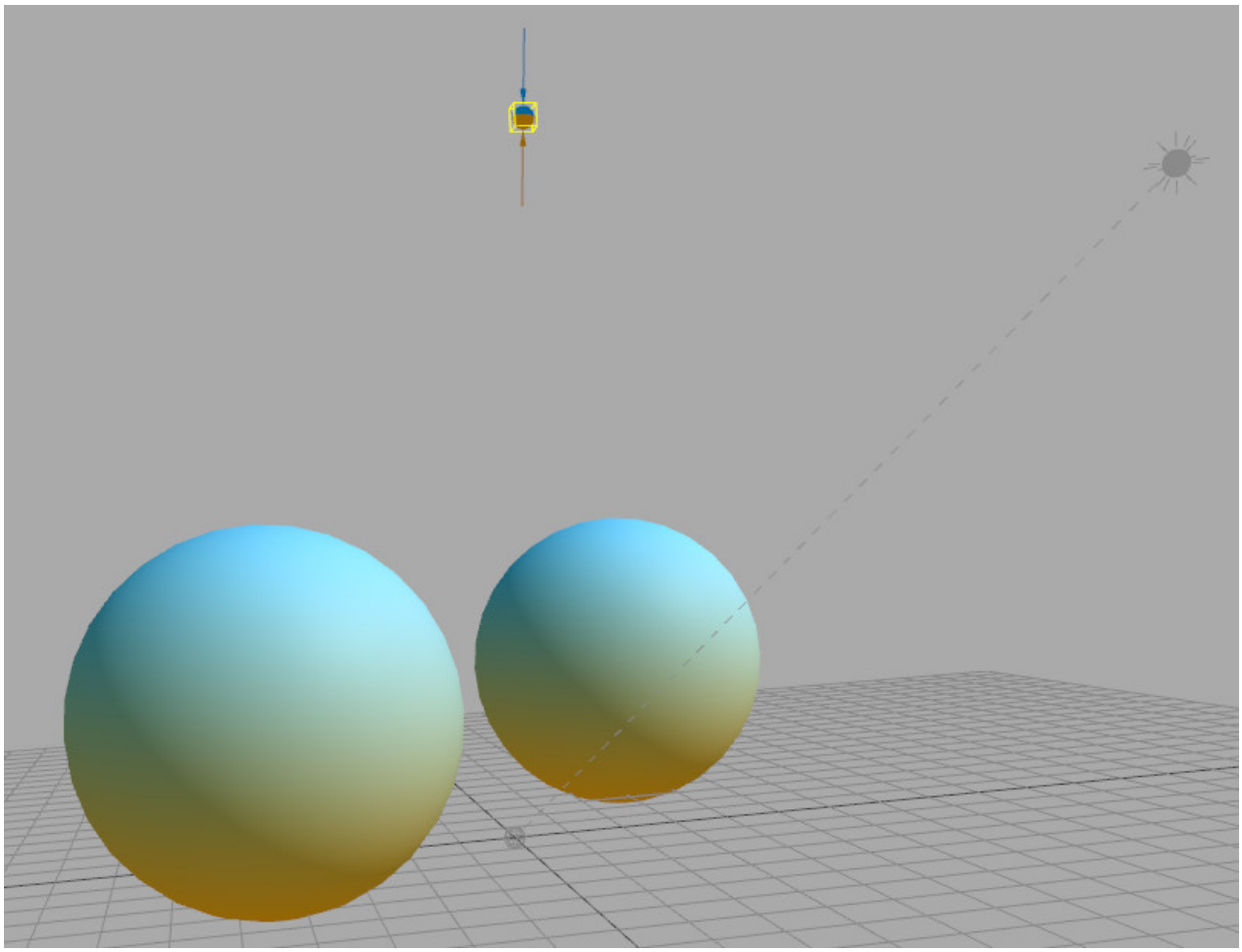
Answer

Global illumination is when one object affects another's appearance. Light traveling through stained glass and hitting the floor is a case of global illumination, as the glass affects the color of the light hitting the floor. Similarly, the bottom of a pool has caustics, places where the light gathers, due to the water rippling. Even an object casting a shadow on another object is a form of global illumination, in that one object affects the other.

The Lambertian, in other words, diffuse, and the Blinn-Phong equation are local illumination models. They are computed using just the information from the object itself, not using other surrounding objects.

Lesson: Hemisphere Lights

[Use editor and explain as we go http://mrdoob.github.com/three.js/editor/]



One light source in three.js that attempts to simulate the surrounding bounced light in a scene is

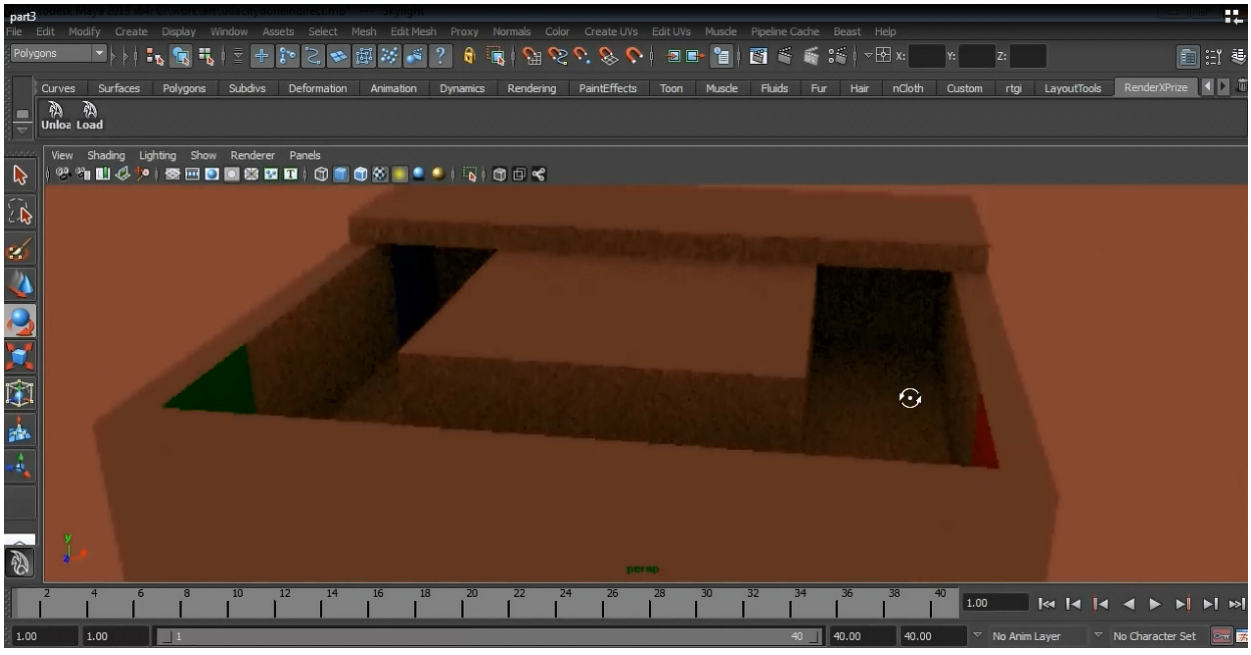
the hemisphere light. Instead of this light having a particular position, it is considered to fully surround the scene, with light coming from every direction. This is also how you can think of ambient light, but with a difference.

With a hemisphere light you assign a light color to the top and a different color to the bottom. The surface's normal determines the color of the light it receives from this light source. If the normal points upward, the light's color is the top color; if the surface normal points straight down, the bottom color. Any direction in between gives a blend of these two colors. The idea is that you assign the top color to be something like the sky, the bottom something like the surround terrain. It's basically a more elaborate form of ambient lighting.

When we talk about textures, we'll see a more involved version of this scheme, called a diffuse map.

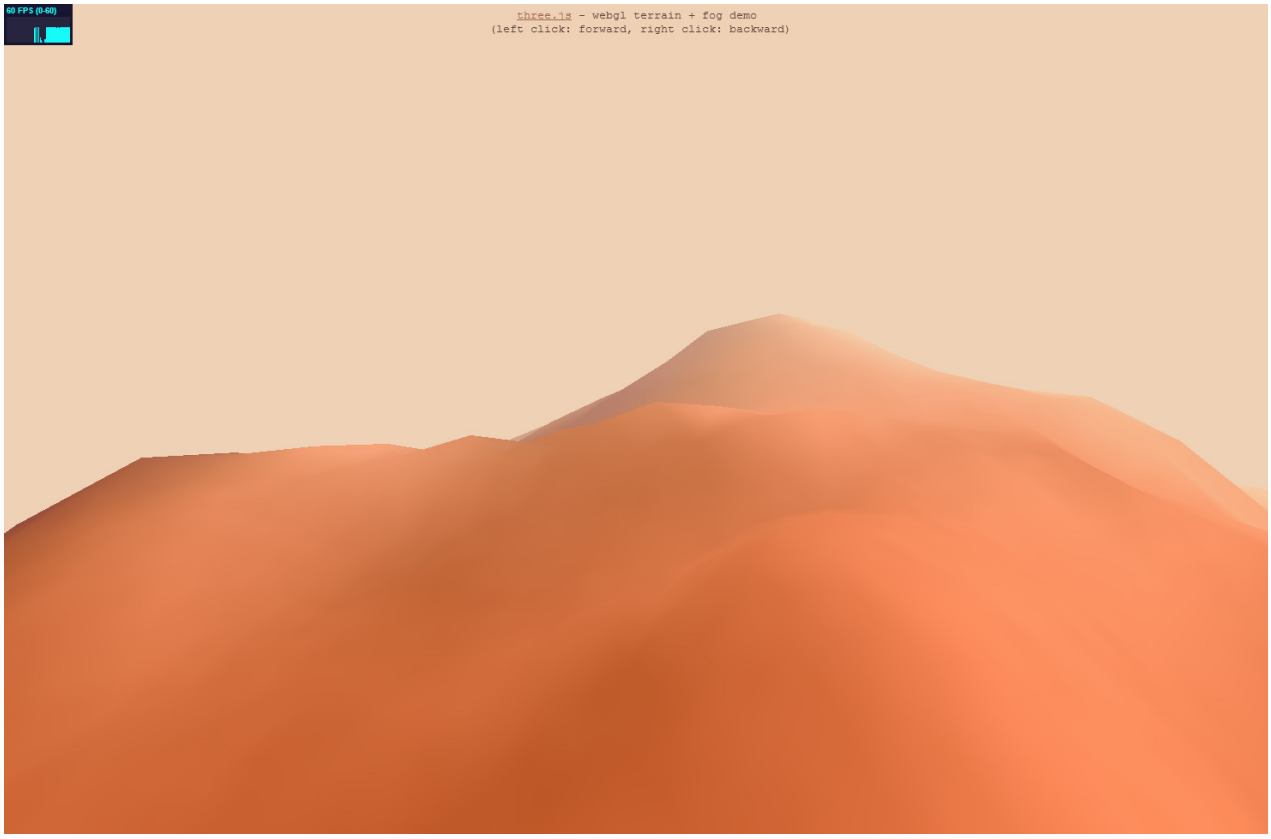
Lesson: Skylighting

[video David Larsson: Unit6_SkyLight, part3.mp4]



Lesson: Fog

[Really good start:
http://mrdoob.github.com/three.js/examples/webgl_geometry_terrain_fog.html]

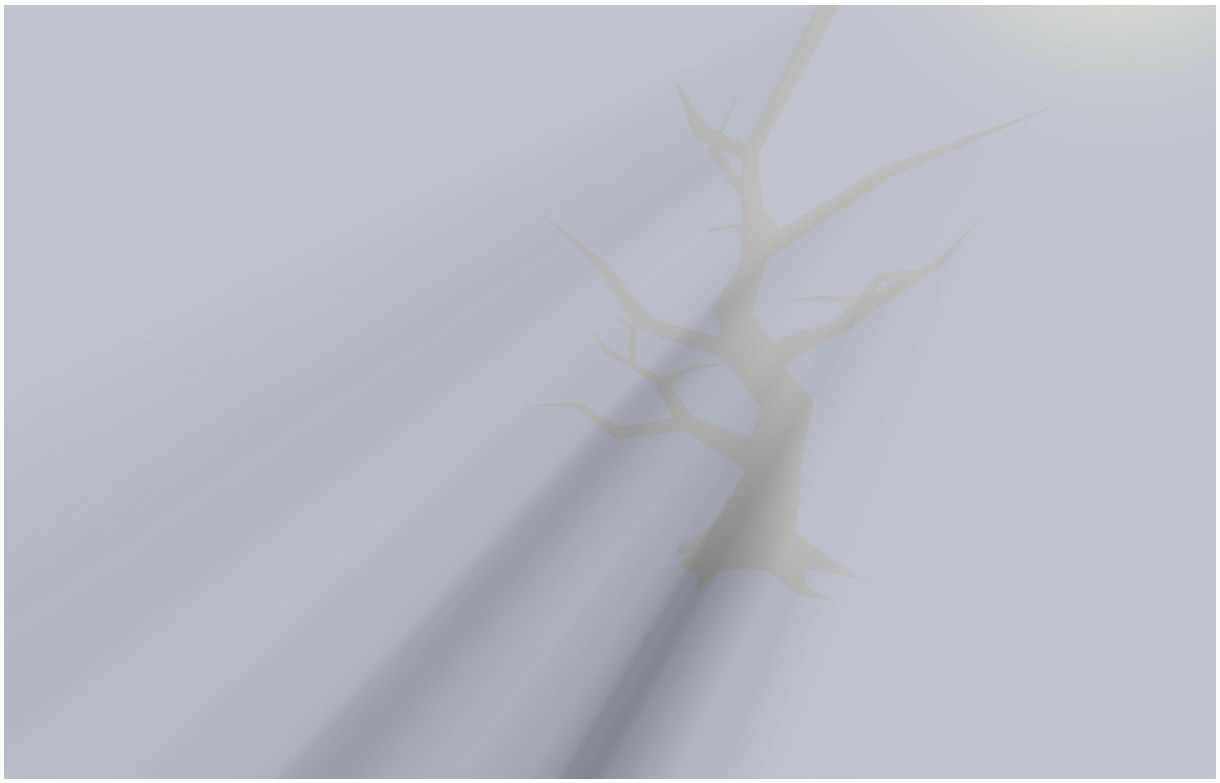


One phenomenon that's somewhere between a light and a camera effect is fog. In the real world, the atmosphere can absorb and reflect light in many different ways. Three.js has two simple forms of fog. One is called linear fog, and we've been using that in a number of demos. A minimum distance is specified for where the fog starts, and a maximum for where the scene is entirely the fog color. Anything in between these two distances get the fog color blended in.

The other type of fog is exponential. This is a bit more like real life, in that you specify a density of particles. This value affects how the fog increases, somewhat realistically.

Both kinds of fog are useful for limiting how far we see into the distance. There's only so much content a program can load, and without other clever tricks the user will see the limits of the world.

[http://mrdoob.github.com/three.js/examples/webgl_postprocessing_godrays.html]



There are plenty of other types of fog and atmospheric effects possible. This demo shows what are called “godrays”, a beams of light type of effect. See the additional course materials for more about this particular effect, which uses post-processing image-based shaders.

To finish up, try out the fog demo that follows. You can choose from among the different types of fog and change the related parameters and fog color.

[Additional Course Materials:
Try [the demo](http://mrdoob.github.com/three.js/examples/webgl_geometry_terrain_fog.html) at the start of the lesson yourself.

Three.js has two types of fog built in,
[Fog](http://mrdoob.github.com/three.js/docs/56/#Reference/Scenes/Fog), which falls off linearly, and
[FogExp2](http://mrdoob.github.com/three.js/docs/56/#Reference/Scenes/FogExp2), which falls off exponentially.

For the godrays demo, see [the comments at the top of the shader code](https://github.com/mrdoob/three.js/blob/master/examples/js/ShaderGodRays.js).
]

Demo: three.js Fog

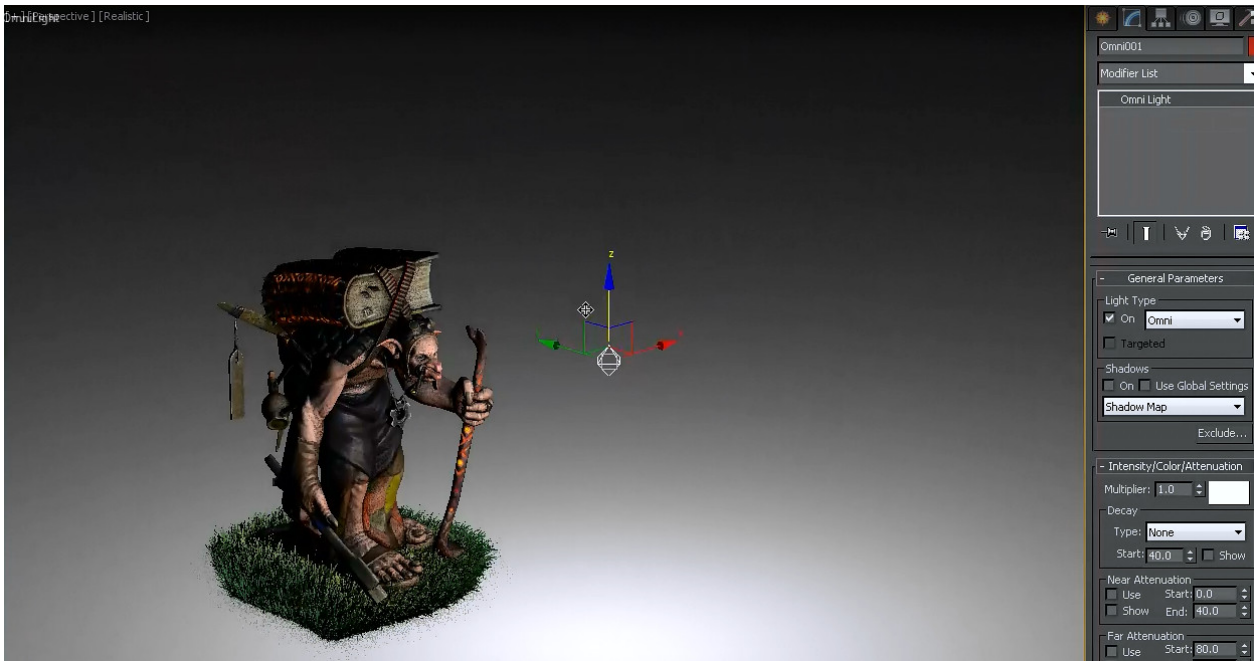
[Demo at unit6-fog_demo.js]

Problem Set: Lighting

[begin recording 4/3 part 1]

Problem 6.1: Omni Light

[Video Unit6_PS_OmniLight, OmniLight.mp4]



There are many different types of lights you'll encounter in various applications. Programs for lighting design will include all sorts of data, such as how particular fixtures affect the distribution of light from an emitter.

The other thing you may run into is different terminology. Here a light source called an omni light is being moved around a scene and modified.

[----- new page ----]

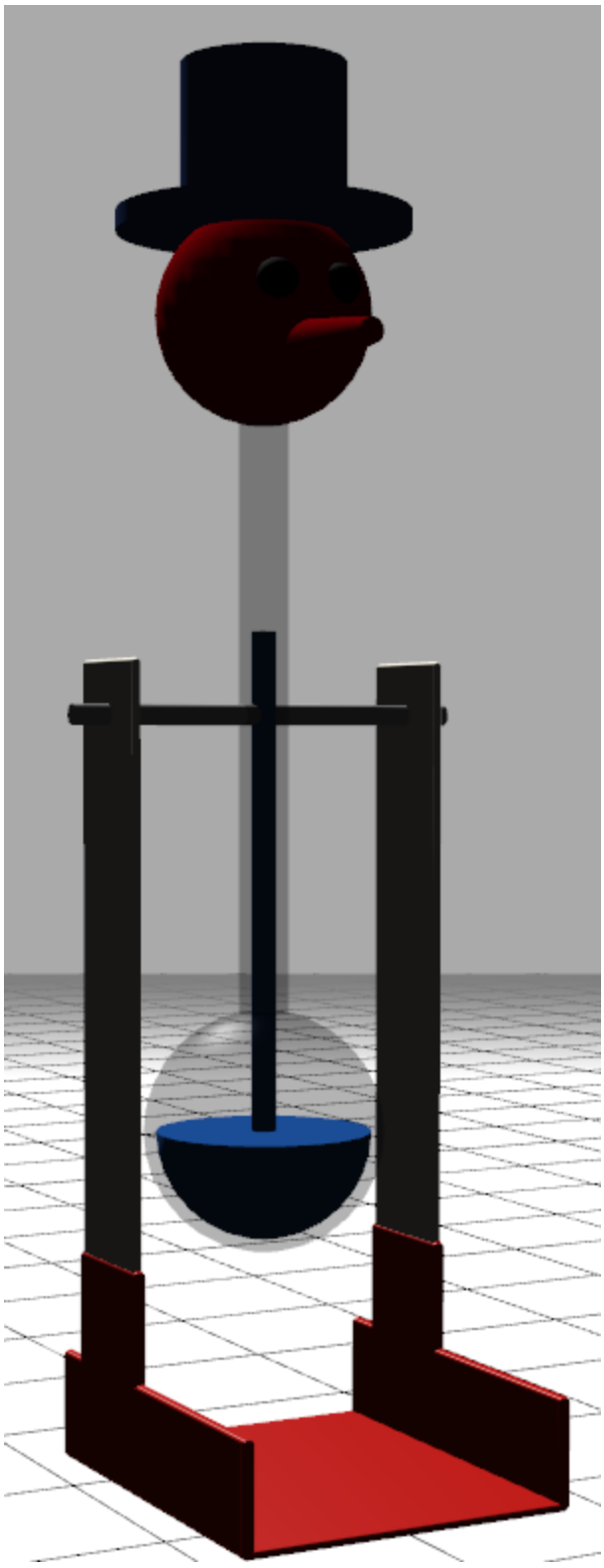
What's the three.js name for this sort of light source?

( ) directional light
( ) point light
( ) spot light
( ) hemisphere light

Answer

The light's location moves around in the scene, and the light emits in all directions. This describes a point light.

Problem 6.2: Swivel Light Control



In this exercise you start with a directional light pointing at the back of the bird. It doesn't look very

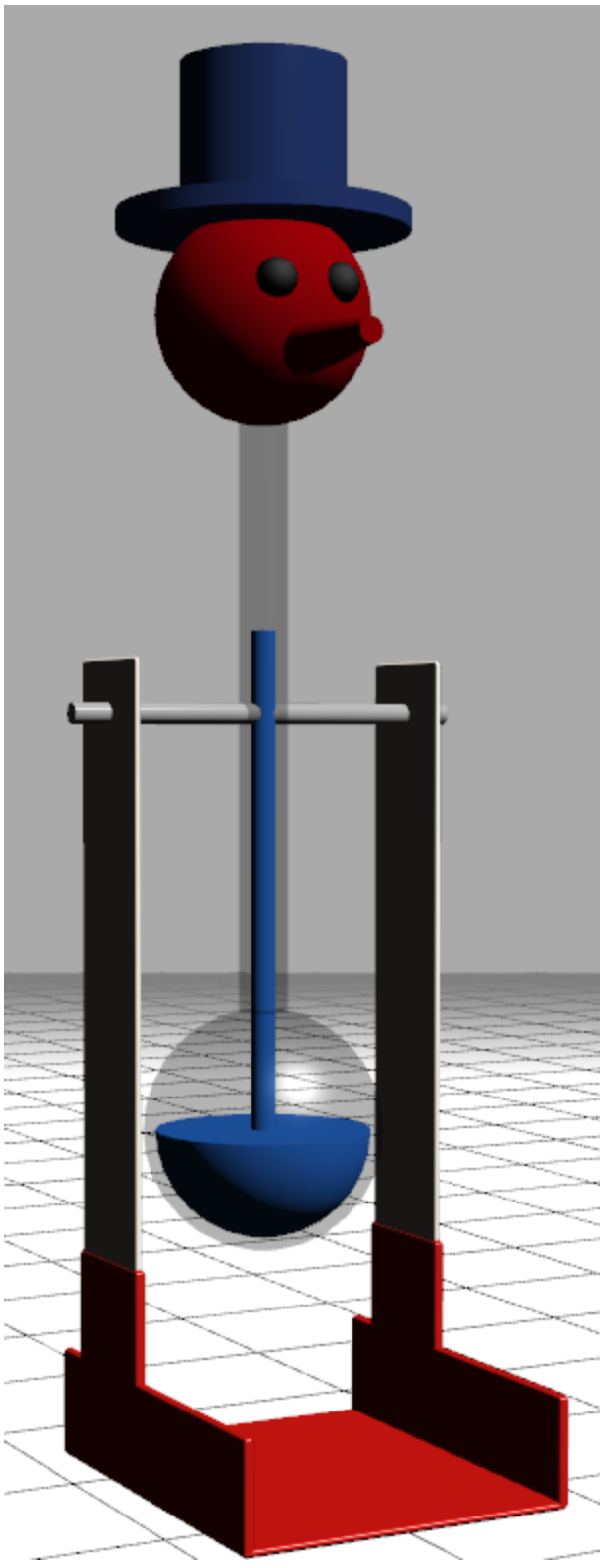
good. I want you to hook up a slider to the light and ***vary its X and Z direction by the cosine and sine of the angle.***

Save your solution!

[show unit6-ps_swivel_light_solution.js]

When you've got the code running, the program should look like this as you vary the angle. Make sure to save your solution, as you'll also use it in the next exercise.

Answer



The code needed is put in the render method.

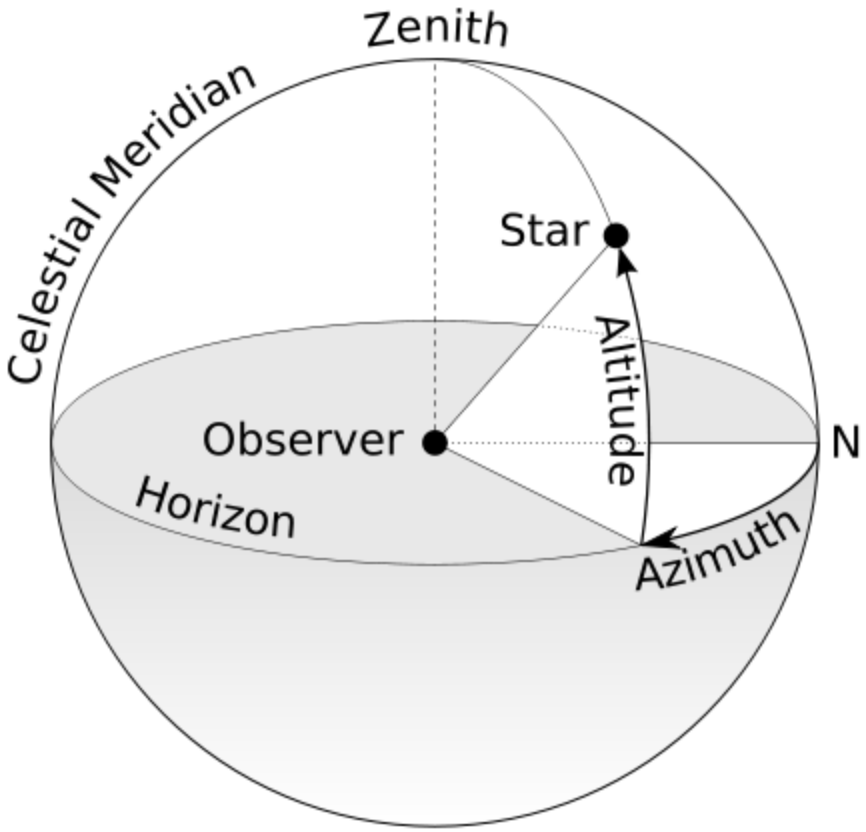
light.position.x = Math.cos(effectController.angle * Math.PI/180.0);
light.position.z = Math.sin(effectController.angle * Math.PI/180.0);

```
light.position.x = Math.cos(
    effectController.angle * Math.PI/180.0 );
light.position.z = Math.sin(
    effectController.angle * Math.PI/180.0 );
```

Notice that the light position set will not be normalized - the Y position is always 1 - but this doesn't hurt anything.

Problem 6.3: Swivel and Tilt Light Control

[from http://en.wikipedia.org/wiki/File:Azimuth-Altitude_schematic.svg]



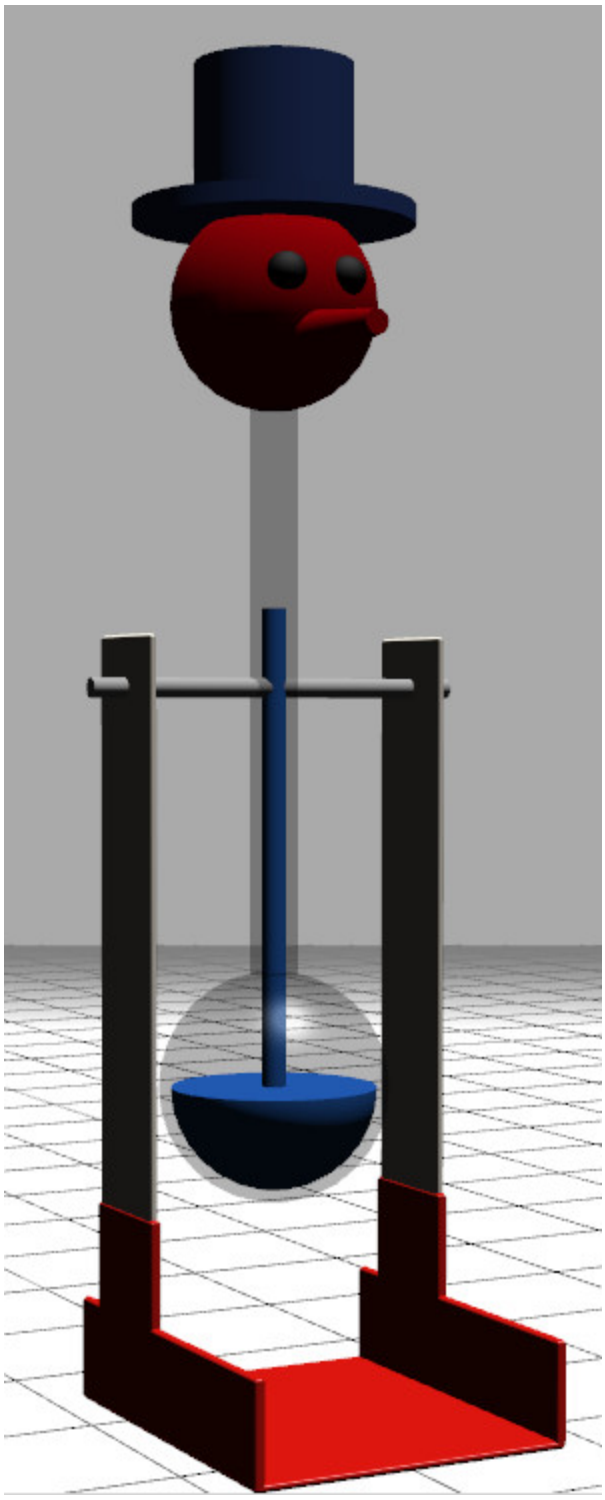
Add another slider to also vary the altitude angle of the light. To get it right, you'll have to take into account that the X and Z components need to be scaled down as the Y component increases,

so that the direction *is* normalized.

In other words, ***the light's direction X^2 + Y^2 + Z^2 = 1.***

[show running unit6-ps_azalt_light_solution.js]

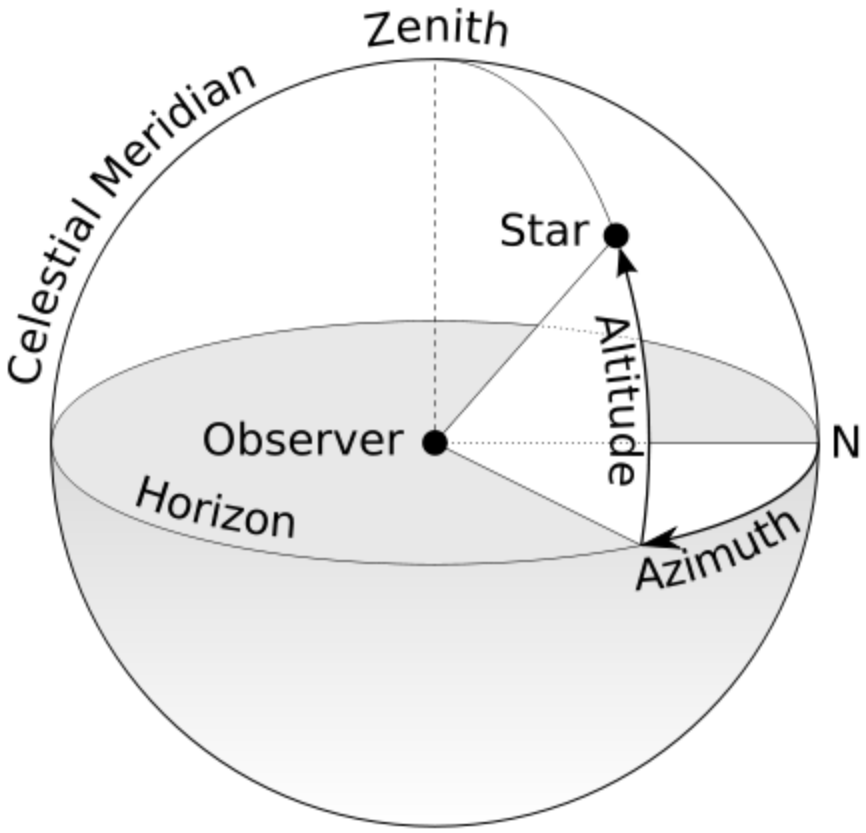
When you're done, the light should now operate with altitude and azimuth angles. Altitude is similar to latitude on the earth, azimuth is similar to the longitude.



[Additional Course Materials:
The altitude-azimuth diagram is from
[Wikipedia](http://en.wikipedia.org/wiki/File:Azimuth-Altitude_schematic.svg).
]

Answer

[show image again http://en.wikipedia.org/wiki/File:Azimuth-Altitude_schematic.svg]



[Note: this is corrected from what's in the original video, which had an error (no square root) for light.position.y]

```
// altitude
light.position.y = Math.sin( effectController.altitude * Math.PI/180.0 );

// azimuth
var length = Math.sqrt(1 - light.position.y*light.position.y);

light.position.x = length * Math.cos( effectController.azimuth * Math.PI/180.0 );
light.position.z = length * Math.sin( effectController.azimuth * Math.PI/180.0 );
```

```
// altitude
light.position.y = Math.sin( effectController.altitude * Math.PI/180.0 );

// azimuth
var length = Math.sqrt(1 - light.position.y*light.position.y);

light.position.x = length * Math.cos( effectController.azimuth * Math.PI/180.0 );
light.position.z = length * Math.sin( effectController.azimuth * Math.PI/180.0 );
```

The code first computes the altitude. It then uses this position to figure out how much length is left to share among the other two coordinates. Their positions are computed and multiplied by this length. Since cos^2 + sin^2 = 1, we know that X and Z will use up the remaining length.

Problem 6.4: Light Characteristics

In three.js, what attributes does each type of light have?

Color	*Intensity*	*Direction*	
[ ]	*[ ]*	*[ ]*	***Ambient***
[ ]	*[ ]*	*[ ]*	***Directional***
[ ]	*[ ]*	*[ ]*	***Positional***
[ ]	*[ ]*	*[ ]*	***Spot***

[Additional Course Materials:
Three.js documentation is [here](http://mrdoob.github.com/three.js/docs).
]

Answer

All lights have a color.

The ambient light doesn't have an intensity, as this is a special light meant to fill in areas in shadow and so should not be cranked up. All other lights have an intensity parameter.

Directional and spotlights have a direction, positional does not. Both of these types of lights also cast shadows in three.js.

[end recording 4/3 part 1 (which continues in Camera below)]

Problem Set 6-5: The Disappearing Spotlight

Remember the spotlight and shadow exercise? If you look at the code, you'll notice a MeshPhongMaterial is used for the ground plane. This material is something we normally need for specular highlights. However, the ground plane is not shiny, so say we change this material to MeshLambertMaterial. The spotlight disappears, but the shadow remains! What's the reason for this?

( ) The spotlight is evaluated per vertex, not per pixel.
( ) Shadows are baked onto the surface, the spotlight is not.
( ) The spotlight is actually defining a specular highlight.
( ) A spotlight can be thought of as a negative shadow.

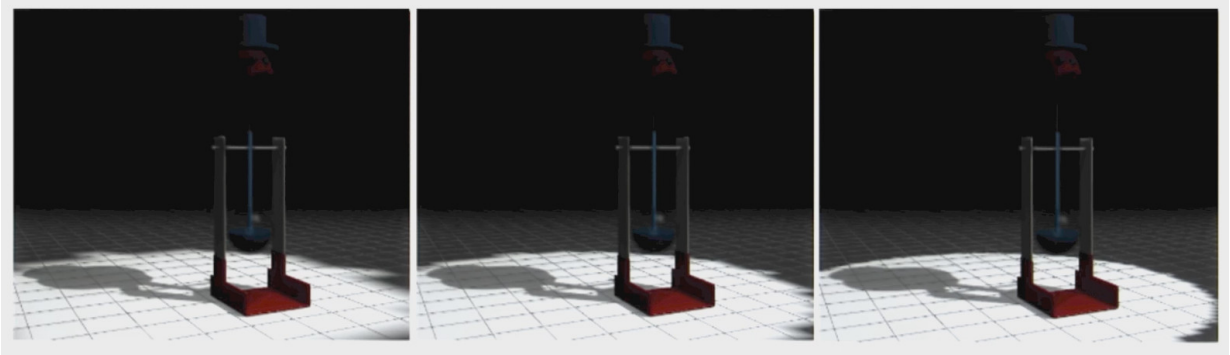
Answer

Even though the spotlight has disappeared, the way three.js's shadow system works in this case is that it dims down all contributions, including ambient.

The reason the spotlight itself disappears is that the Lambert material is normally evaluated per vertex for illumination. The ground plane has just four corners, all of which are outside of the spotlight's area of influence, so just the ambient light is applied.

The shadow in the scene isn't baked on, it's in fact evaluated per pixel, but that's a separate part of the illumination process in this case. A spotlight is not a specular highlight, it affects diffuse and shiny objects alike. Finally, you can think of the area outside of a spotlight as being in shadow, but that doesn't explain why the Lambert material affects it.

[show partial image]



If you were to use the Lambert material and tessellate the ground plane into a grid of squares, you'd start to see the effect of the spotlight, as you can see in these images. Going from left to right the ground plane is tessellated more and more, catching the spotlight's effect better and better. However, evaluating the spotlight's effect per pixel instead of per vertex means many fewer triangles needed to be generated and processed.

[Cut: Problem 6.6: Over the Shoulder Lighting

Exercise:
Make a headlight that is “over the left shoulder”. Use a spotlight with shadow. Goal is to rotate light’s position so that it is always say 20 degrees left, 10 degrees up from where the viewer is. I’ll have to experiment a bit, as we want to make it a little challenging but not too hard. The trick is gimbal lock-ishness when we get near the pole. Maybe I have a special control that doesn’t let me go up or down too much, but pulls back at an angle.

Alternately, get the lookat matrix from the camera. Use this to then position the light! This could well be a second question in the Camera section!

Answer

....]