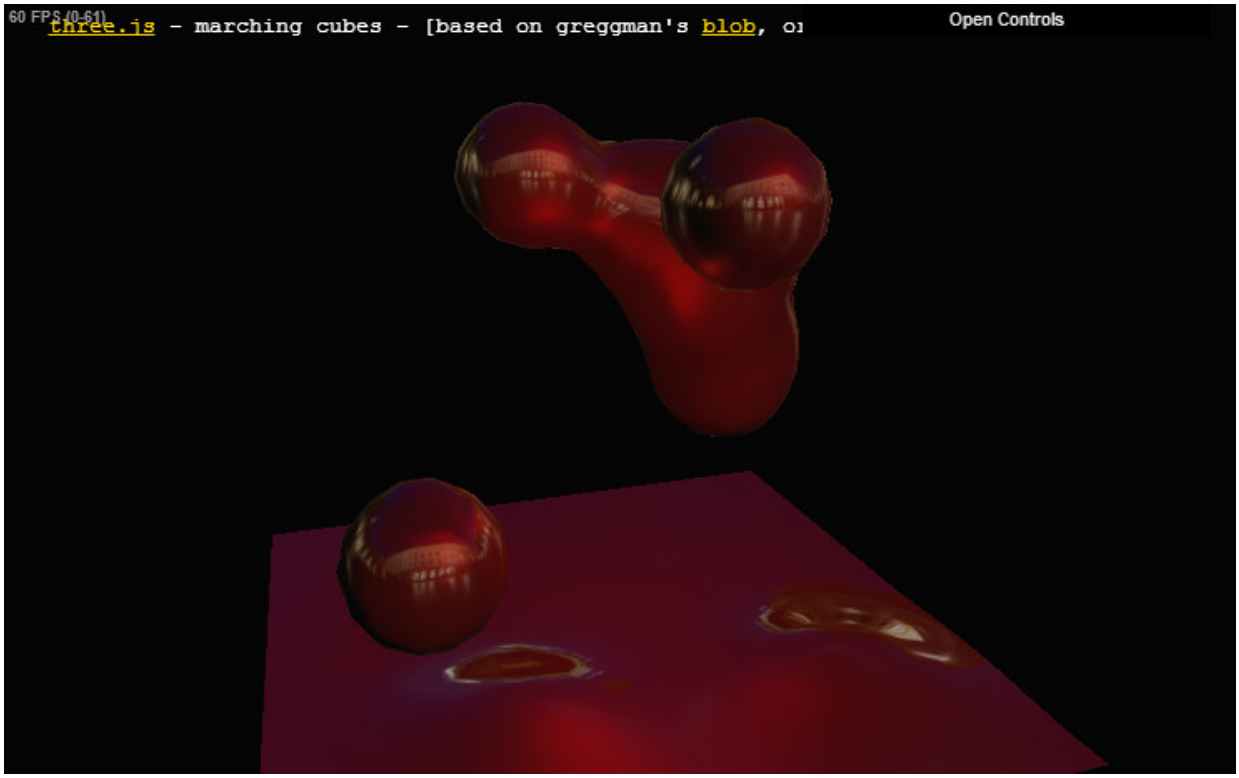
Lesson 3: Colors and Materials

Lesson: Overview

[Show this demo program with the voiceover, changing its materials as we go: http://mrdoob.github.com/three.js/examples/webgl_marching_cubes.html - flip through different materials]



In this unit I'm going to teach about colors and materials.

The application uses the GPU to compute a color for each pixel and displays it on the screen. We'll examine how this color is defined and what limitations there are in displaying them. In the second half of this unit we'll show how to create and control different types of materials..

We won't go quite as far as this demo does in its selection of materials. It has some interesting styles that we'll discuss more in the unit about shader programming. However, by the end of this unit you'll have a fair sense about how these images are formed. You should have a good start on color theory and on how materials are used. To get a flavor of the sorts of materials you can apply to a surface, give this demo a try now.

[Instructor Comments: You can try out the demo in the video here http://mrdoob.github.com/three.js/examples/webgl_marching_cubes.html]

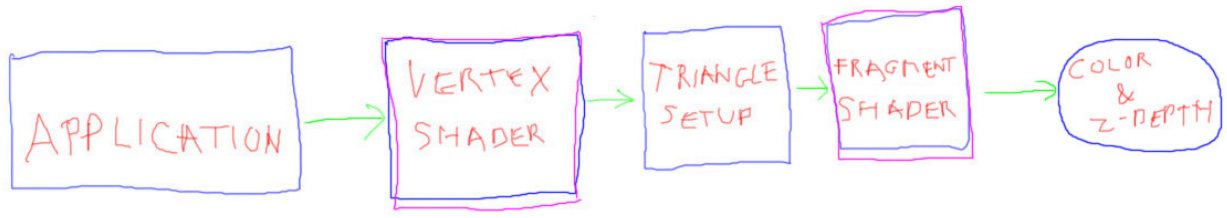
Lesson: The Programmable Pipeline

To get back to the rendering pipeline, our simplified view is this:
[**application -> transform to screen -> rasterize** [lots of room!] **-> color & z-depth**]

The application sends a triangle to the GPU. The GPU determines where the triangle's vertices are on the screen, including the z-depth. Each pixel inside the triangle is shaded. If the pixel passes the Z-buffer test, it is then saved in the image and displayed at the end of the frame.

[replace just first part of pipeline with vertex shader]

[Draw below this the idea of a triangle coming in.]



Modern GPUs have parts of the pipeline that are programmable. The "transform to screen" part of the pipeline is done by what is called a vertex shader. This programmable element - essentially a little computer - processes each vertex of a triangle. The vertex shader uses information provided by it to manipulate each vertex in some way. For example, the color of the triangle at this point could be computed. Or, the vertex's position itself could be modified, if for example you wanted to have an object inflate or explode. One operation the vertex shader always does is to output a location of the vertex on the screen.

[now replace second half of pipeline with the two elements - draw each operation, triangle setup

and fragment shader operation. Rasterize triangle, then show colors of triangle in grid.]

The second half of the pipeline we represent here by two stages: triangle setup and the fragment shader. Triangle setup uses the three screen locations generated by the vertex shader for an incoming triangle. This forms a triangle in screen space. Each pixel covered by a part of the triangle has what is called a fragment generated for it. This process is called “scan conversion”.

[PUT THIS IN A SEPARATE LAYER, so we can hide it next lesson.]
[**fragment shader - WegGL, OpenGL...**
pixel shader - DirectX
]

The fragments generated are sent to the fragment shader. Well, if you use Microsoft’s DirectX API, this is called the pixel shader.

The fragment shader is provided information by the triangle being processed. Similar to the vertex shader, the programmer can also feed in any other data desired. The fragment shader runs a program that typically outputs a color and a z-depth. The z-depth is then tested against the Z-buffer as usual. If the surface is visible, the color is saved for that pixel.

[DO NOT CLEAR THE SCREEN!]

Lesson: RGB Color

[Start with the previous lesson’s drawings, just change title.]

[**How do we (efficiently) calculate this color?**]

The shader pipeline is designed to compute the color at each pixel that the surface covers - that’s its ultimate purpose, after all, creation of an image. Everything done in the pipeline comes down to this: how do we efficiently calculate this color?

That’s what the last half of this unit is about: materials and how they work. Given a material and some incoming light, you want to compute a color coming off of that material.

[**red / green / blue - RGB. Channel**]

In interactive rendering systems a color is defined by three values, red, green, and blue, abbreviated RGB. Each color value is often called a channel. These values are usually in this

order in most systems, RGB.

[**BGR**]
That said, once in awhile you'll run into data in the order BGR, blue green red.

[Show a normal picture of some person, then a BGR-reversed image. e.g. http://commons.wikimedia.org/wiki/File:Half_a_strawberry.jpg and my BGR version.]

Without properly swapping the channels, you'll see some odd-looking images, such as this one on the right. The green stem looks fine, since the green channel is in the same place, but what was once red is now blue. OK, you've been warned.

Lesson: Color Definition

[**float 0.0 to 1.0**
integer 0 to 255]

We usually define colors in one of two ways: as floating point numbers, or as integers. We'll start with floating point, since that's generally the easier to use.

[**(0.0, 0.0, 0.0) RGB is black**
(1.0, 1.0, 1.0) RGB is white
(1.0, 0.0, 0.0) RGB is red
(0.0, 1.0, 0.0) RGB is green
(0.0, 0.0, 1.0) RGB is blue
]

For floating point, each color channel is defined as values from 0.0 to 1.0. (0,0,0) is black, (1,1,1)

is white. Setting just the red channel to 1.0 gives red, green to 1.0 gives green, and the same with blue.

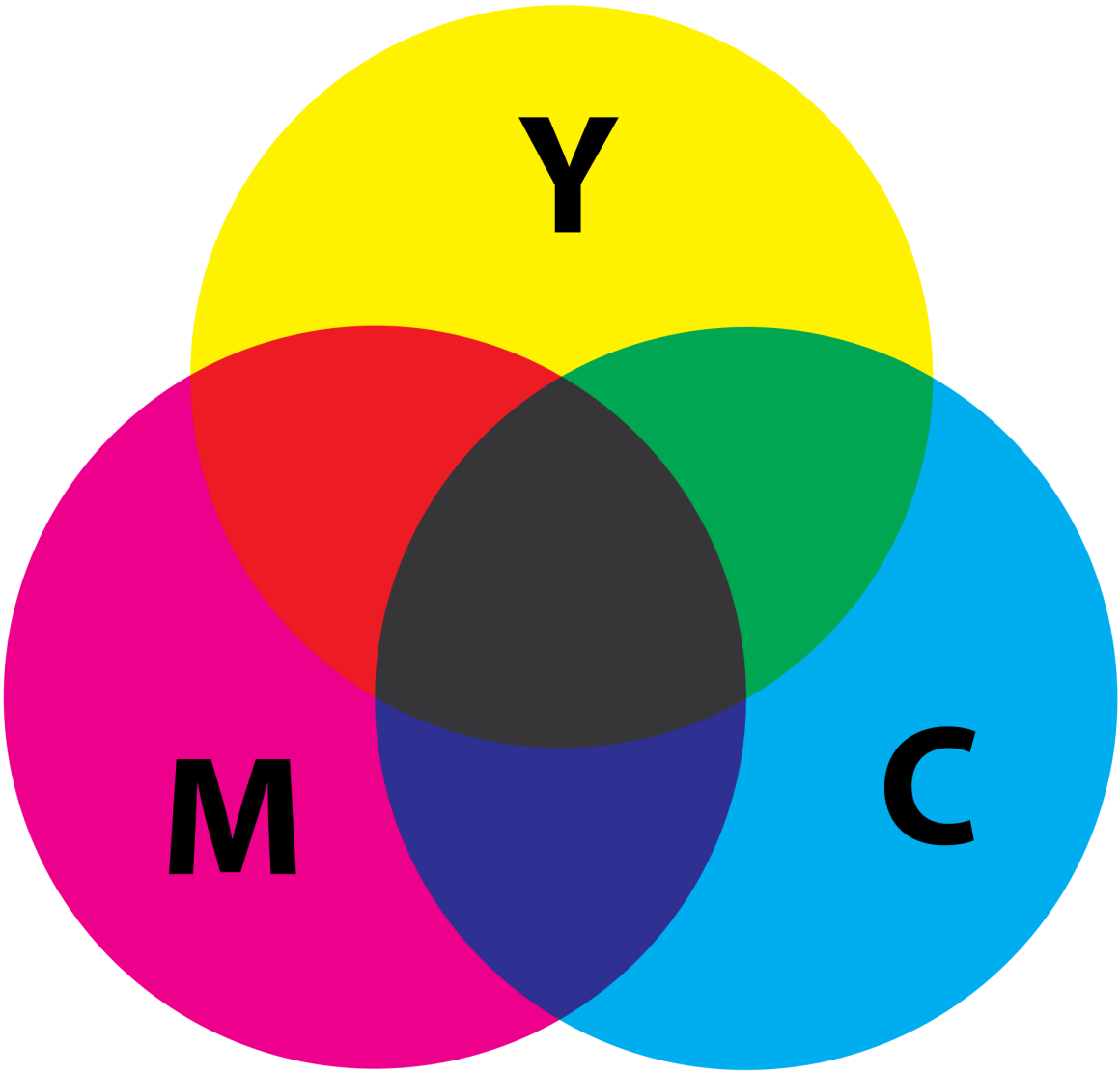
[**(1.0, 1.0, 0.0) RGB is …**]

Where things go off the tracks for some people is when we set a color as follows, RGB of 1, 1, and 0.

…

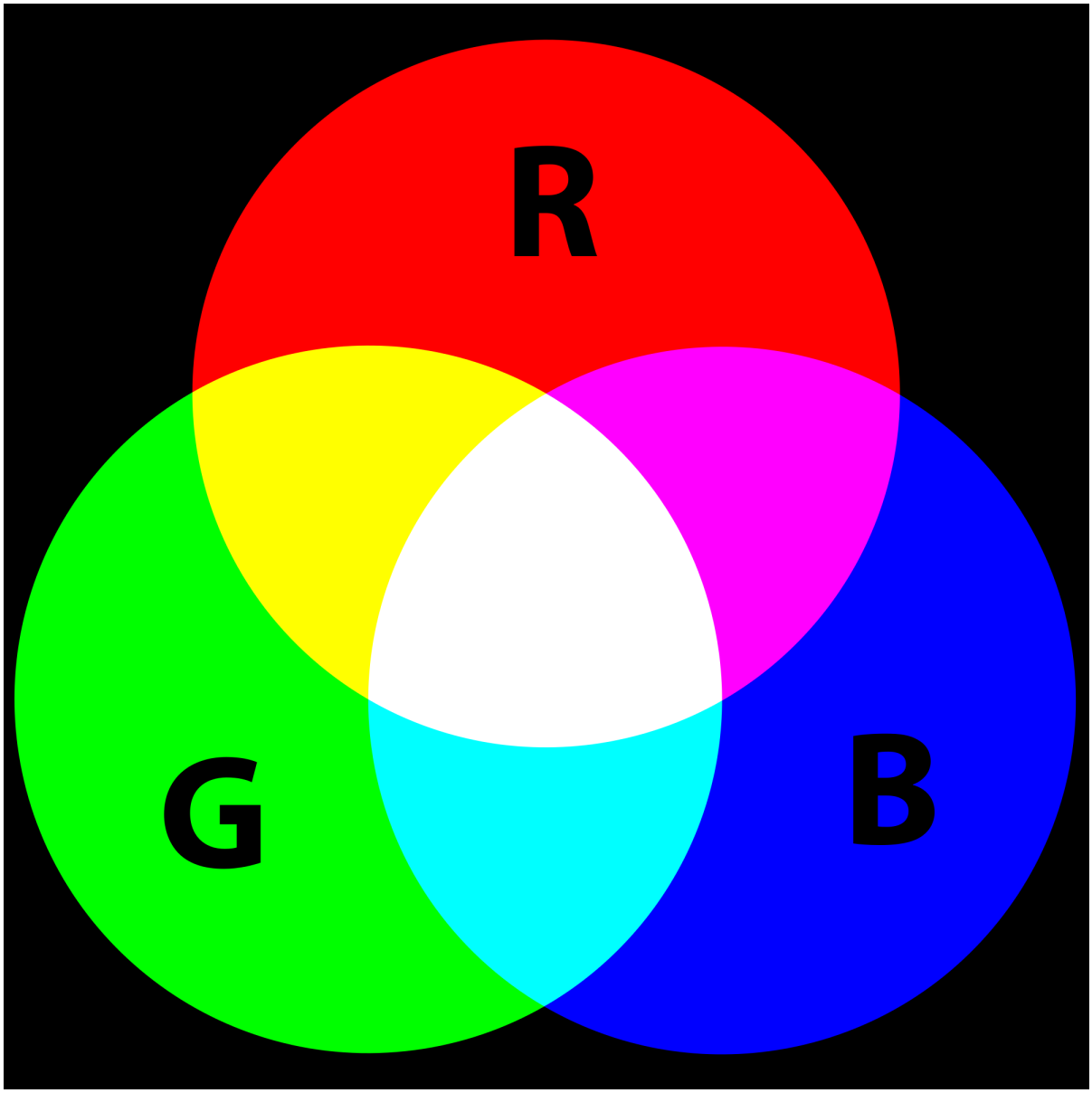
This says to make a color using the red and green channels at their maximum, and with no blue. If I take red paint and green paint and mix them together, I get at best a murky brown. This happens because paint colors are formed by absorption of various wavelengths of light.

[show phrase **subtractive color** and the three circles. Write **CMYK** in the upper right. Put both this and the RGB image on at the same time!]



The color model for mixing paints and inks is called *subtractive color*. The three primary colors used in printing are Cyan, Magenta, and Yellow, with black added as its own dark ink. This printing system is called CMYK, with “K” standing for “Key” and essentially meaning “black”. As different colors are mixed, the pigment becomes darker overall.

[show phrase **additive color** and below]



In computer graphics a monitor emits light of different wavelengths. Our eyes see light with wavelengths between 390 to 750 nanometers. Longer wavelength light is red, shorter is blue or violet, with the rainbow in between.

If we specify some red and some green light, then these contributions are added together, giving a brighter color overall. This color model is called **additive color**. See the additional course materials for more articles about this area. If you want a particular color, the additive color RGB values are what you'll need to determine.

To get familiar with this way of mixing colors, try out the demo that follows.

[Instructor comments: One good color mixing applet is here http://www.oberlin.edu/faculty/brichard/Apples/ColorMixingPage.html. It lets you vary the intensity of each color contribution and see the effect, and shows both additive and subtractive color mixing.

You can also play with additive colored lights in a WebGL application here http://voxelent.com/html/beginners-guide/1727_06/ch6_Wall_Final.html from the book “WebGL Beginner’s Guide” http://www.amazon.com/WebGL-Beginners-Guide-Diego-Cantor/dp/184969172X .

For more about wavelengths of light, the visible spectrum, and color in general, see Wikipedia’s articles here http://en.wikipedia.org/wiki/Visible_spectrum and here http://en.wikipedia.org/wiki/Color

Articles on Wikipedia also cover additive color http://en.wikipedia.org/wiki/Additive_color and subtractive color http://en.wikipedia.org/wiki/Subtractive_color, as well as CMYK http://en.wikipedia.org/wiki/CMYK]

Demo: RGB Color Demo

Lesson: Setting Colors

[var sphereMaterial = new THREE.MeshLambertMaterial( );

In three.js there are a number of ways to set a color. See the additional course materials for a link to the document page. We’ll cover four of them here.

We start by defining a material:

…

We’ll talk more about what exactly this material does in full later - for now, just know that you can set a color for it. If we don’t set any color at all, the default is white.

The MeshLambertMaterial object that you just created has several different parameters. These

can be set when you first create the object, or later on in a number of different ways. One of these parameters is the material's color.

[write **THREE.Color**]
This parameter is of type **THREE.Color**, and you can use all the methods associated with it.

One method is to set the R, G, and B channels separately.

[Write these out and put the color of the channel in its color, 1.0 0.0 0.0]

```
sphereMaterial.color.r = 1.0;
sphereMaterial.color.g = 0.0;
sphereMaterial.color.b = 0.0;
```

Here I'm setting the color to red. This directly sets the variables r, g, and b in the Color attribute to the desired values.

[display and write with colors

```
sphereMaterial.color.setRGB( 0.972, 0.749, 0.141 );
```

I can also use the setRGB method on the Color class, setting the color to green.

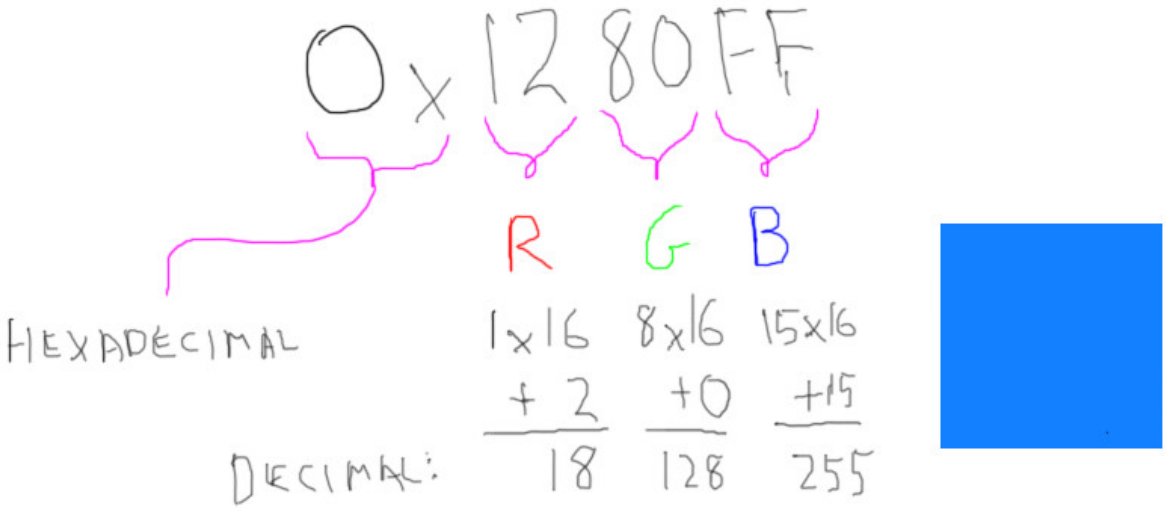
[display and write WITH COLORS

```
sphereMaterial.color.setHex( 0x1280FF );
```

A more compact way to set a color is to use hexadecimal numbers. Here I'm setting a bright blue.

The value "0x1280FF" is a single number holding the R, G, and B channels all together. Let's break it down:

[hex breakdown picture. DO ONE COLOR AT A TIME]



The “0x” prefix means that this is a hexadecimal number, base 16, where the letter A means 10, B means 11, on up to F meaning 15. If you don’t know about hexadecimal, see the additional course materials for a link to more information. The three channels, R, G, and B, are each encoded in order, treating each two digits as the channel’s value. For example, the red channel is One-Two in hexadecimal. This means 1 sixteen plus 2, to give 18 in decimal values.

[Draw green]
Green is computed similarly, 8 times 16 plus 0 gives a level of 128.

[Draw blue]
Blue is at the maximum, adding up to 255.

Each of these values can range from 0 to 255. The range 0-255 is used because most computer monitors have eight bits of precision for each color channel. Two to the eighth power is 256, so the range 0 to 255 is the entire range of possible color channel values.

So, once you have a color you like, in three.js you can conveniently just paste it into place. In fact, there’s a way of initializing the material to a color when you create it:

```
var sphereMaterial = new THREE.MeshLambertMaterial( { color: 0xF4F100 } );
```

When you put something in braces like this, you’re initializing that variable in the object to the given value.

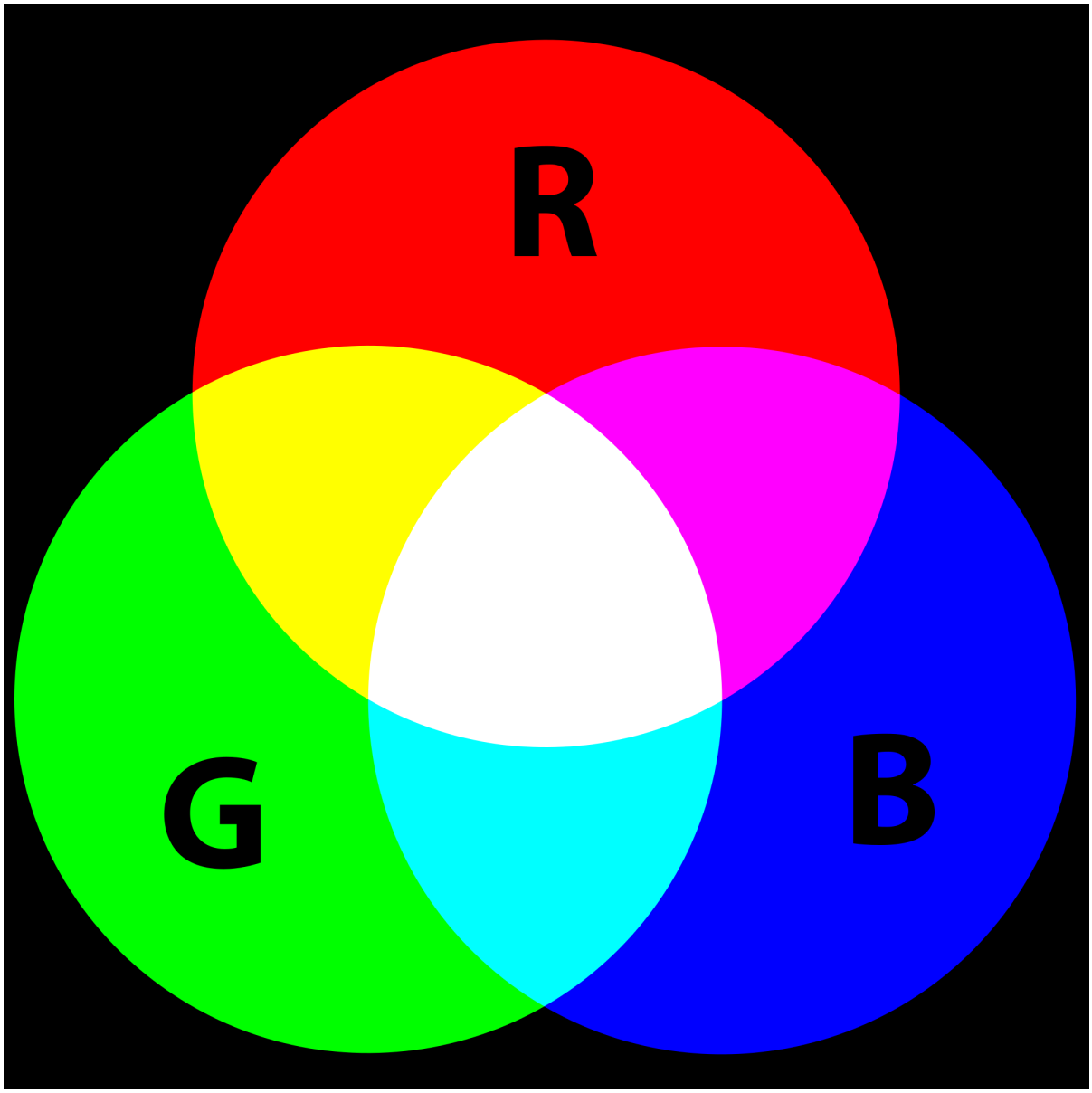
This particular hexadecimal value gives a yellow color. Notice that the hexadecimal letters such as the “f”s here can be upper or lowercase, it doesn’t matter.

[Instructor Comments: Look at the Color class documentation for three.js http://mrdoob.github.com/three.js/docs/53/#Reference/Core/Color for further help. If you don’t know about hexadecimal, the Wikipedia page is a good place to start. There are plenty of decimal to hexadecimal converters online. Here’s one http://www.statman.info/conversions/hexadecimal.html, for example.]

[I’ve been relying on Wikipedia being available. I expect the student will have the wherewithal to search on the subject if it’s not.]

Question: Magenta

[recorded 3/26]



Red is 1,0,0
Green is 0,1,0
Blue is 0,0,1

Using additive colors, what color is magenta? [highlight in image]

_____, _____, _____

Answer

Magenta is red and blue added together, so the answer is 1,0,1.

Question: Chartreuse

[recorded 3/26]

This question is not graded, I'm just interested in what you think of when you hear this color's name. Ready? 'Chartreuse'. What color does that word make you think of? Don't worry about your answer, I got this one wrong, too. I probably should have answered "I don't know", but I thought I *did* know.

( ) A deep purple
( ) A yellow-green color
( ) A red-orange color
( ) I don't know

Answer

The answer is "A yellow-green color". Chartreuse is named after a liqueur from France. I have a theory that women know this color more often than men, since it's a fabric color used in dresses. Me, I thought it was a red-orange color. Crayola thought it was a yellow color and revised it later.

[Additional Course Materials:
Here are the Wikipedia articles on [the color](http://en.wikipedia.org/wiki/Chartreuse_(color)) and [the liqueur](http://en.wikipedia.org/wiki/Chartreuse_(liqueur)).

See [Crayola's answer](http://en.wikipedia.org/wiki/List_of_Crayola_crayon_colors).
]

[Exercise: Drinking Bird Colors - removed]

[Cut this exercise. It's mostly tedium, I think, and we test color conversion elsewhere.]

[Put words in blue on screen]

Let's adjust our drinking bird model. I've made the model a little bit better, but messed up the colors. Here are the RGB values I'd like you to set:

hat 24,38,77

head 104,1,5

body 31,86,169

legs 173,167,155

feet 150,15,11

[we could get more involved here, e.g. some in floats, some in hex, but I think this is good enough. The last two are tricky, as they're in hex. The last one in particular is particularly tricky, in that the student has to recall that each channel has two hex digits.]

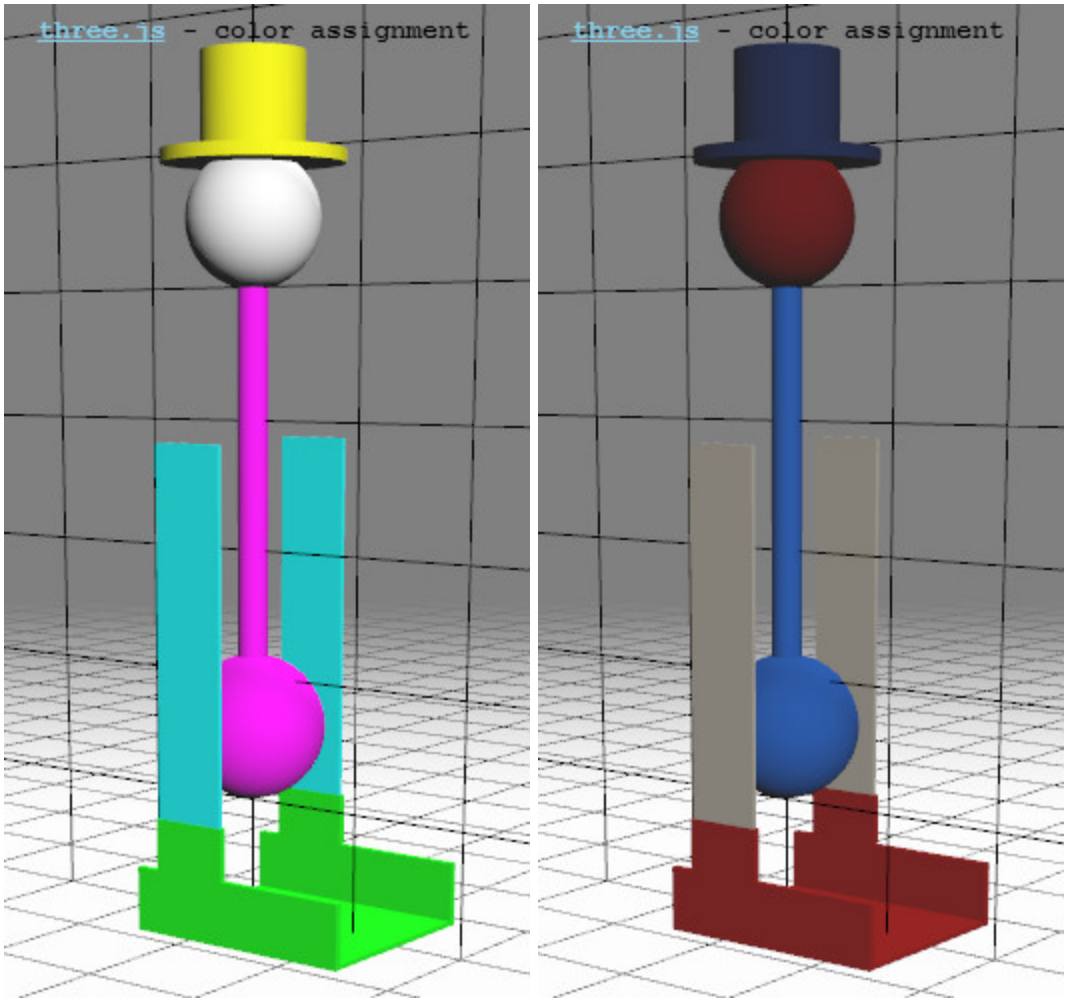
You'll see the materials in the code. Please use the format given for each to complete the exercise. About the only thing to be aware of is how to convert from integer RGBs to floating point RGBs. The right way is to divide the integer value by 255:

```
float fval = ival / 255.0;
```

I've seen people make the mistake of dividing by 256 - don't do that.

When you're done, the drinking bird should look something like this:

[need snapshot or actual video of solution for this one, once exercise is done.]



[The exercise is online at original-framework/unit3-drinkbird_col_exercise.js]

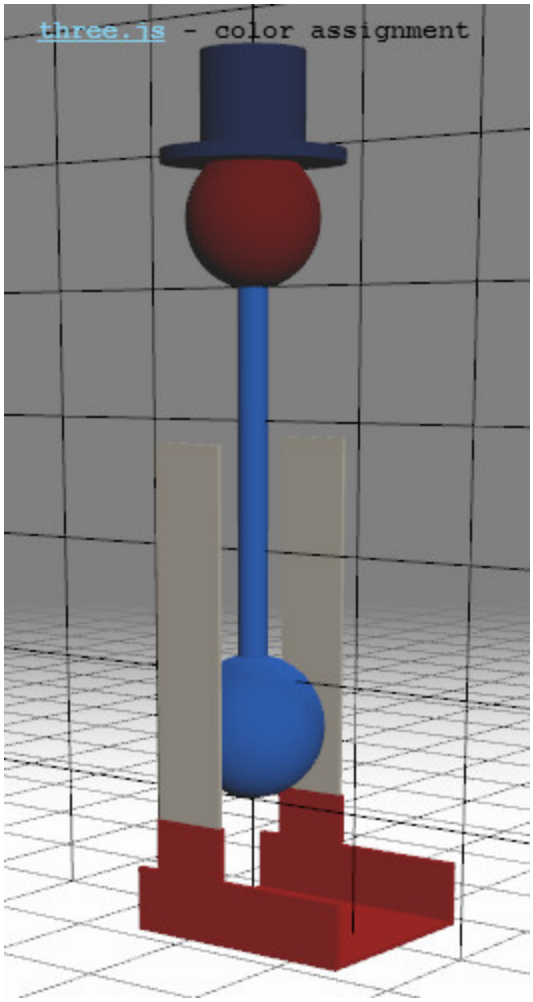
[Instructor Comments: feel free to have the program itself do the conversion work by assigning a formula to a color channel. Definitely use a hexadecimal converter such as this one http://www.statman.info/conversions/hexadecimal.html]

[Answer - removed]

Here's the final program as I coded it up. The code's pretty simple; it's just worth your time to deal with hexadecimal and the other options a bit so you know what they're about. As a reward, see the additional course materials for a color picker that gives you the color in both integer RGB triplets and as a hexadecimal string at the top. This hex encoding also frequently gets used when specifying colors in HTML pages and in Photoshop, so it's worth understanding this peculiar format.

[put solution code

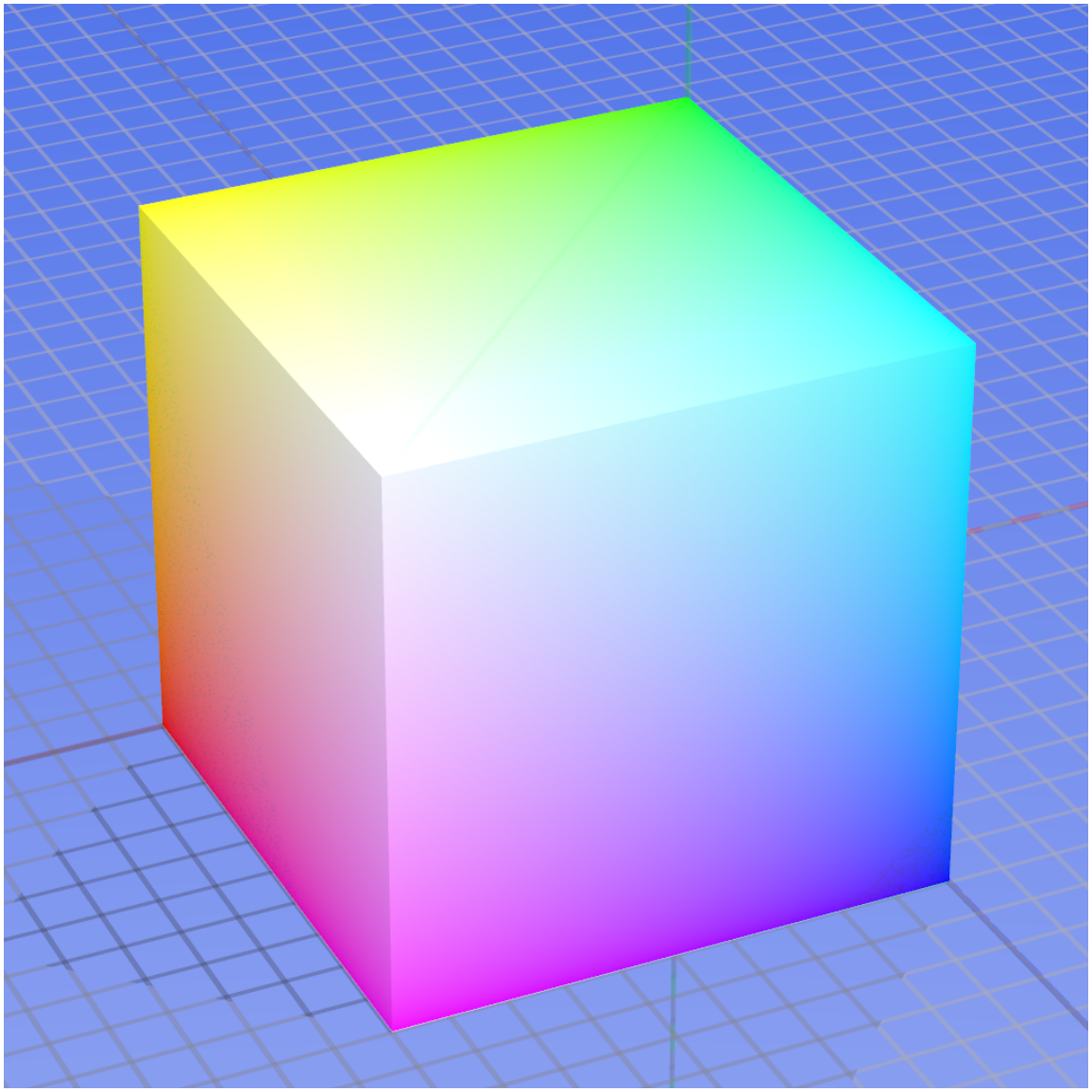
```
var headMaterial = new THREE.MeshLambertMaterial( );
headMaterial.color.r = 104/255;
headMaterial.color.g = 1/255;
headMaterial.color.b = 5/255;

var hatMaterial = new THREE.MeshLambertMaterial( );
hatMaterial.color.r = 24/255;
hatMaterial.color.g = 38/255;
hatMaterial.color.b = 77/255;

var bodyMaterial = new THREE.MeshLambertMaterial( );
bodyMaterial.color.setRGB( 31/255, 86/255, 169/255 );

var legMaterial = new THREE.MeshLambertMaterial( );
legMaterial.color.setHex( 0xAdA79b );

var footMaterial = new THREE.MeshLambertMaterial( { color: 0x960f0b } );
```



[Instructor Comments: See the color picker page http://www.colorpicker.com/ for one color picker. There are plenty more out there, just search on “web color picker” and you’ll find a bunch.]

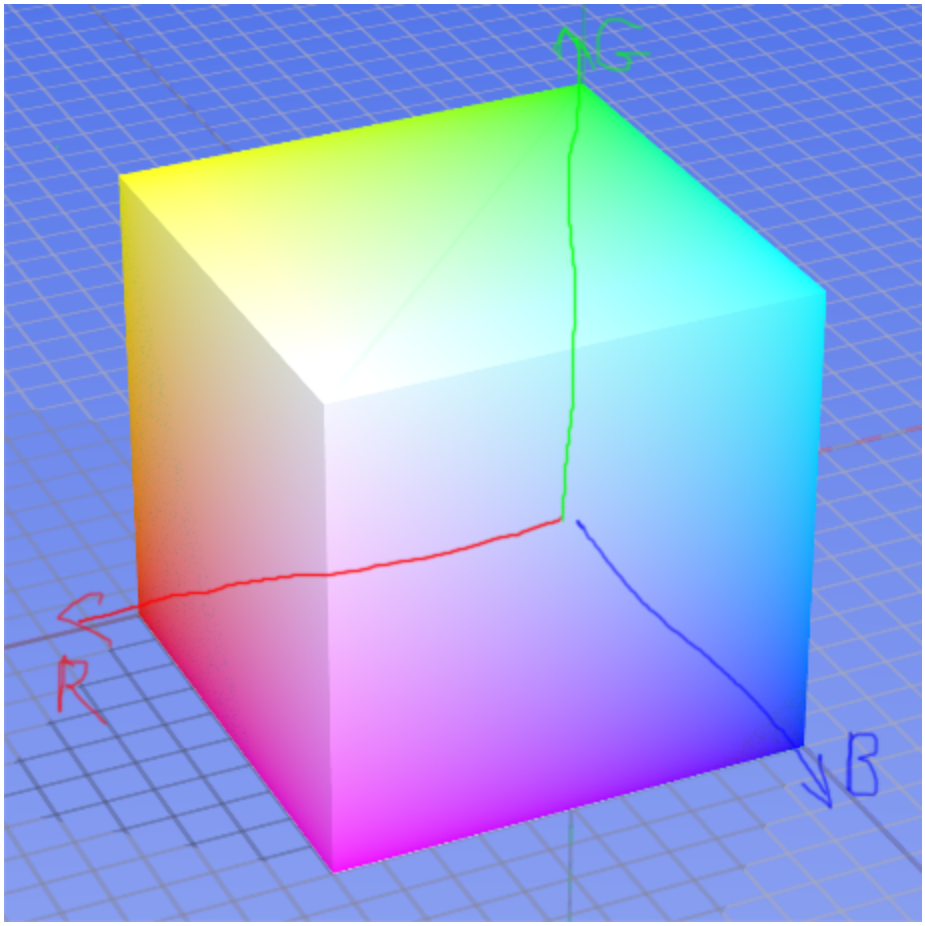
Lesson: the Color Cube

[color cube, http://upload.wikimedia.org/wikipedia/commons/a/af/RGB_color_solid_cube.png]



Here is a visualization of the RGB color space.

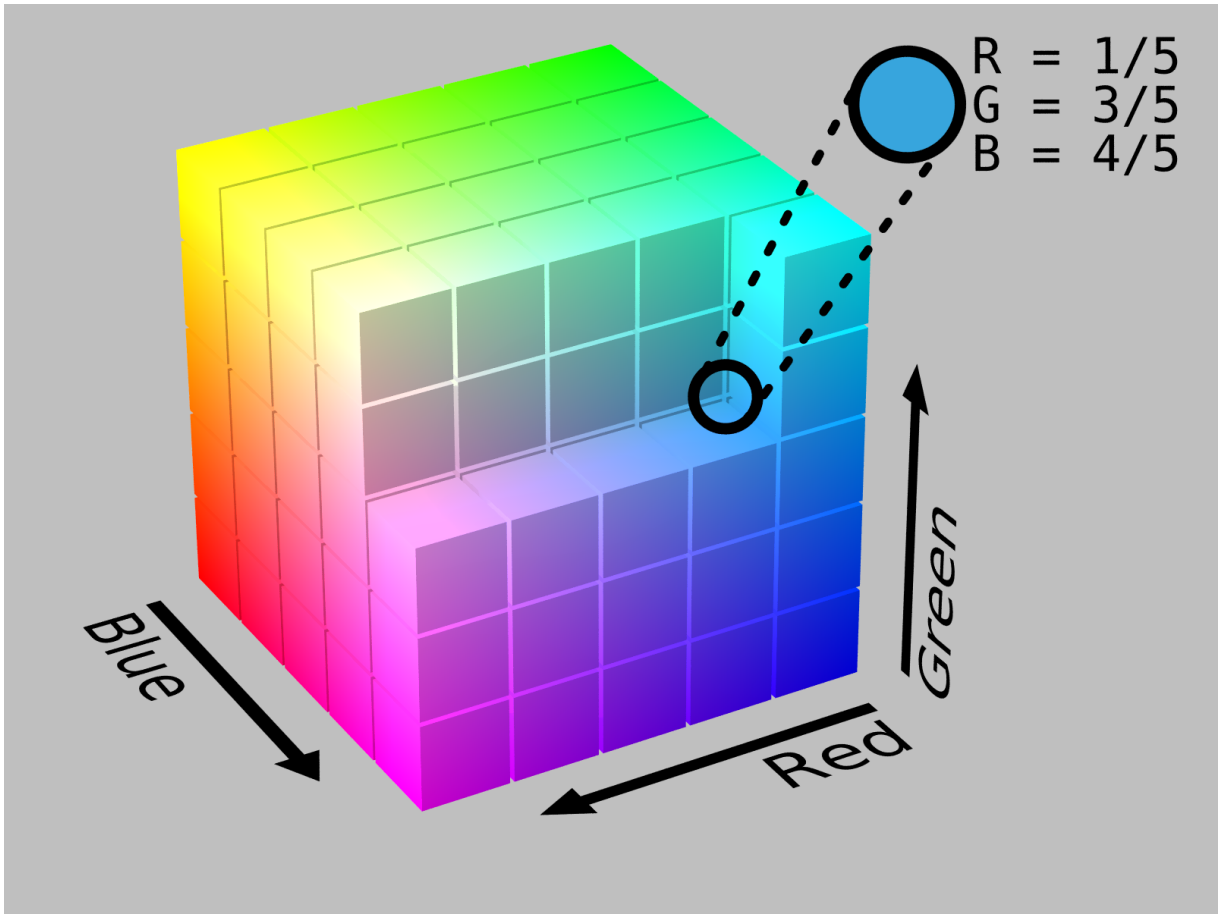
[draw RGB on the image itself.]



The axis to the left shows the red channel increasing, the vertical axis is green, and the axis to the right is the blue.

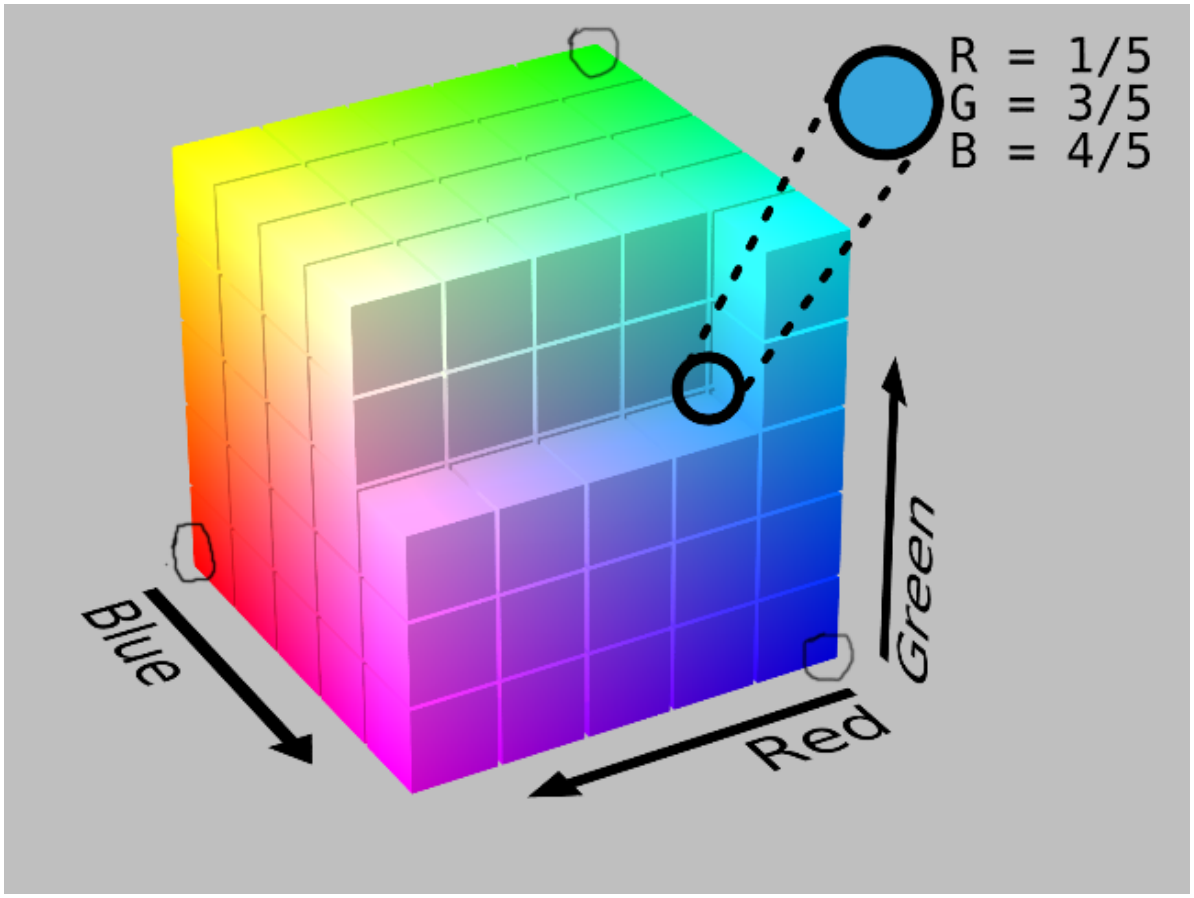
We can’t see the interior of the cube, showing intermediate value: all the values we can see in this particular view have one channel at a maximum.

[image http://upload.wikimedia.org/wikipedia/commons/8/83/RGB_Cube_Show_lowgamma_cutout_b.png]



Here's a view inside the cube, with the red and blue axes swapped - axis direction doesn't matter. This point inside the cube shows a sky blue formed by an RGB value of one-fifth red, three-fifths green, and four fifths blue.

[mark the corners in the figure above]



Another way to visualize the RGB color space is to take a slice through the cube at these three corners.

Each corner represents a single color channel at full intensity, with the other two channels off. For example, this vertex is where green is set to 1 and red and blue are 0. By slicing a plane through these three corners we'll be able to see where R + G + B = 1.0. This slice gives another view of how these colors blend. This is what the next exercise is about.

Exercise: Vertex Attributes

[recorded 3/26]

Up to this point a vertex has been defined by just a 3D coordinate. In fact, you can attach a large amount of data to each vertex. Data attached in this way is called an **attribute** in WebGL. To visualize a slice through the color cube, we'll attach a different color to each vertex.

[put code here]

To start, you'll add a triangle to the scene. To then add a color to each vertex, use the following kind of code after defining the triangle's face:

```
var color1 = new THREE.Color( 0xF08000 ); // orange
var color2 = new THREE.Color( 0x808000 ); // olive
var color3 = new THREE.Color( 0x0982FF ); // bright blue

geometry.faces[0].vertexColors = [ color1, color2, color3 ];
```

```
var color1 = new THREE.Color( 0xF08000 ); // orange
var color2 = new THREE.Color( 0x808000 ); // olive
var color3 = new THREE.Color( 0x0982FF ); // bright blue

geometry.faces[0].vertexColors = [ color1, color2, color3 ];
```

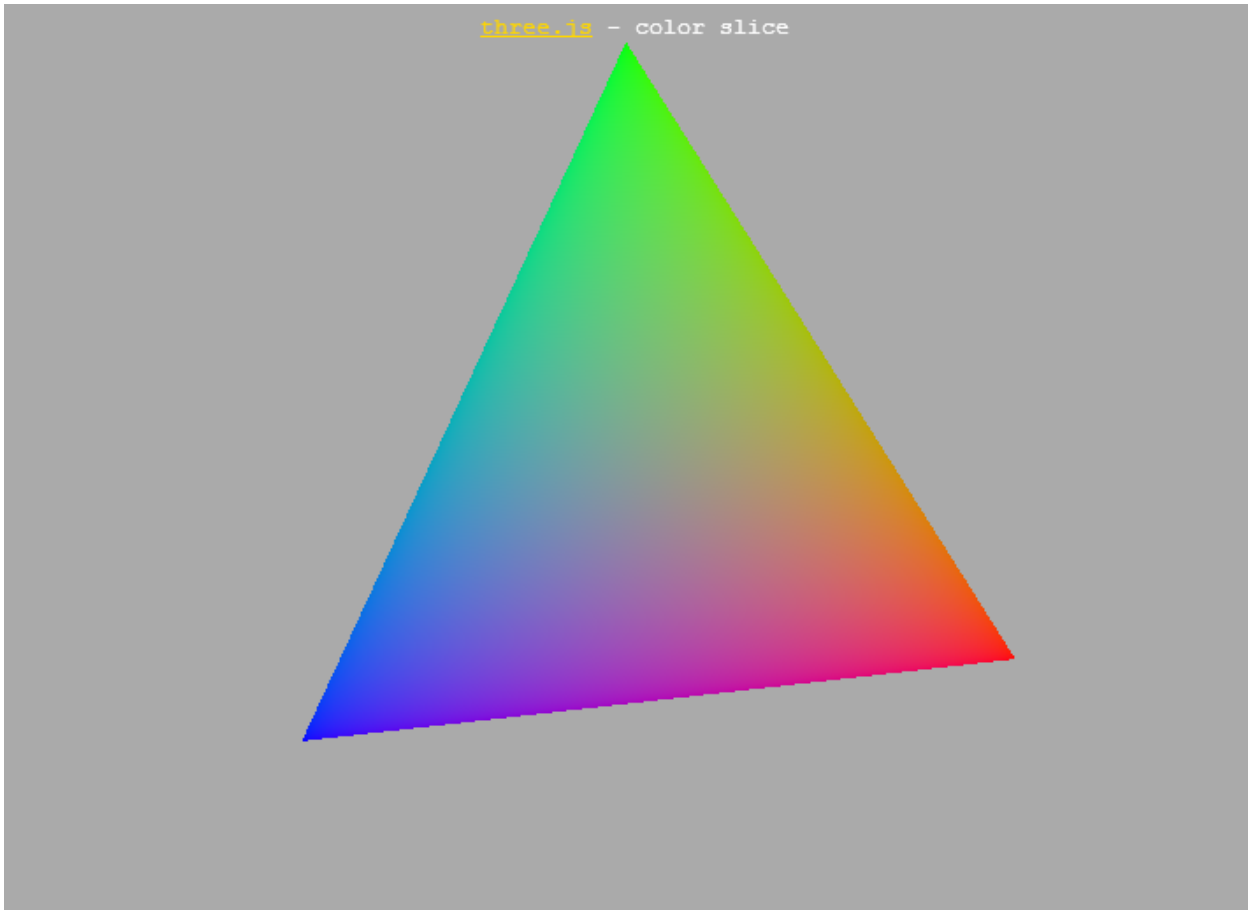
This last line adds the RGB color attribute by specifying a color at each vertex of the face.

Make a triangle with three vertices, at locations

Vertices
100,0,0 red
0,100,0 green
0,0,100 blue

Instead of these colors in the example code, add a full red color to the first vertex, full green to the second, and full blue to the third. The GPU will interpolate the colors across the surface, giving a blend of these colors in the triangle's interior.

[run program!]



[Exercise is unit3_color_triangle_exercise.html, solution is here: unit3_color_triangle_solution.html]

To “interpolate” means to find a value in between two or more other values. In this case, the original data are the three corner colors, red, green, and blue.Here the existing data are the three vertex colors, red, green, and blue. All the locations on the triangle are a blend of these three colors. If a fragment is close to a vertex, that vertex’s color contributes most to its final color. The influence of each corner is inverse of the distance from that corner: closer means the corner has more influence.

When you’re done with this exercise, the result should look like this.

Answer

```
geometry.vertices.push( new THREE.Vector3( 100, 0, 0 ) );
geometry.vertices.push( new THREE.Vector3( 0, 100, 0 ) );
geometry.vertices.push( new THREE.Vector3( 0, 0, 100 ) );
```

```
geometry.faces.push( new THREE.Face3( 0, 1, 2 ) );

// these actually multiply the color of the material, which is white by default
var color1 = new THREE.Color( 0xff0000 );
var color2 = new THREE.Color( 0x00ff00 );
var color3 = new THREE.Color( 0x0000ff );

geometry.faces[0].vertexColors = [ color1, color2, color3 ];
```

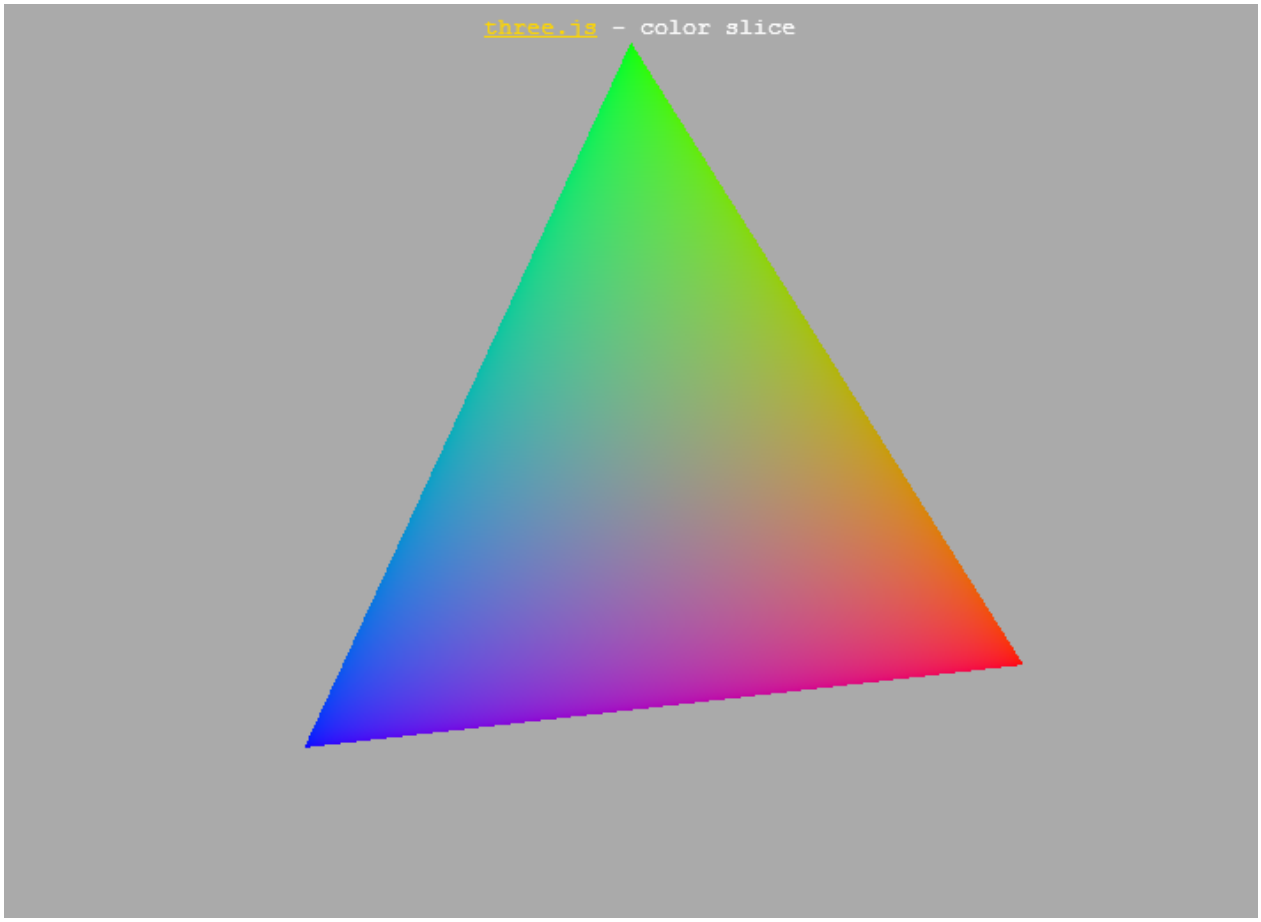
```
geometry.vertices.push( new THREE.Vector3( 100, 0, 0 ) );
geometry.vertices.push( new THREE.Vector3( 0, 100, 0 ) );
geometry.vertices.push( new THREE.Vector3( 0, 0, 100 ) );

geometry.faces.push( new THREE.Face3( 0, 1, 2 ) );

// these actually multiply the color of the material, which is white by default
var color1 = new THREE.Color( 0xff0000 );
var color2 = new THREE.Color( 0x00ff00 );
var color3 = new THREE.Color( 0x0000ff );

geometry.faces[0].vertexColors = [ color1, color2, color3 ];
```

The code itself is pretty straightforward. We set the vertex positions, then make the triangle face - nothing fancy there. We then create three colors red, green, and blue. Finally, an array of these three colors is assigned to the face itself.

One thing worth noting is that these vertex colors are actually multiplied by the material's color when displayed. Since the material color is by default a solid white, it doesn't have a noticeable effect in this particular case.

[TODO: need to move this code into github.]

[show solution again.]

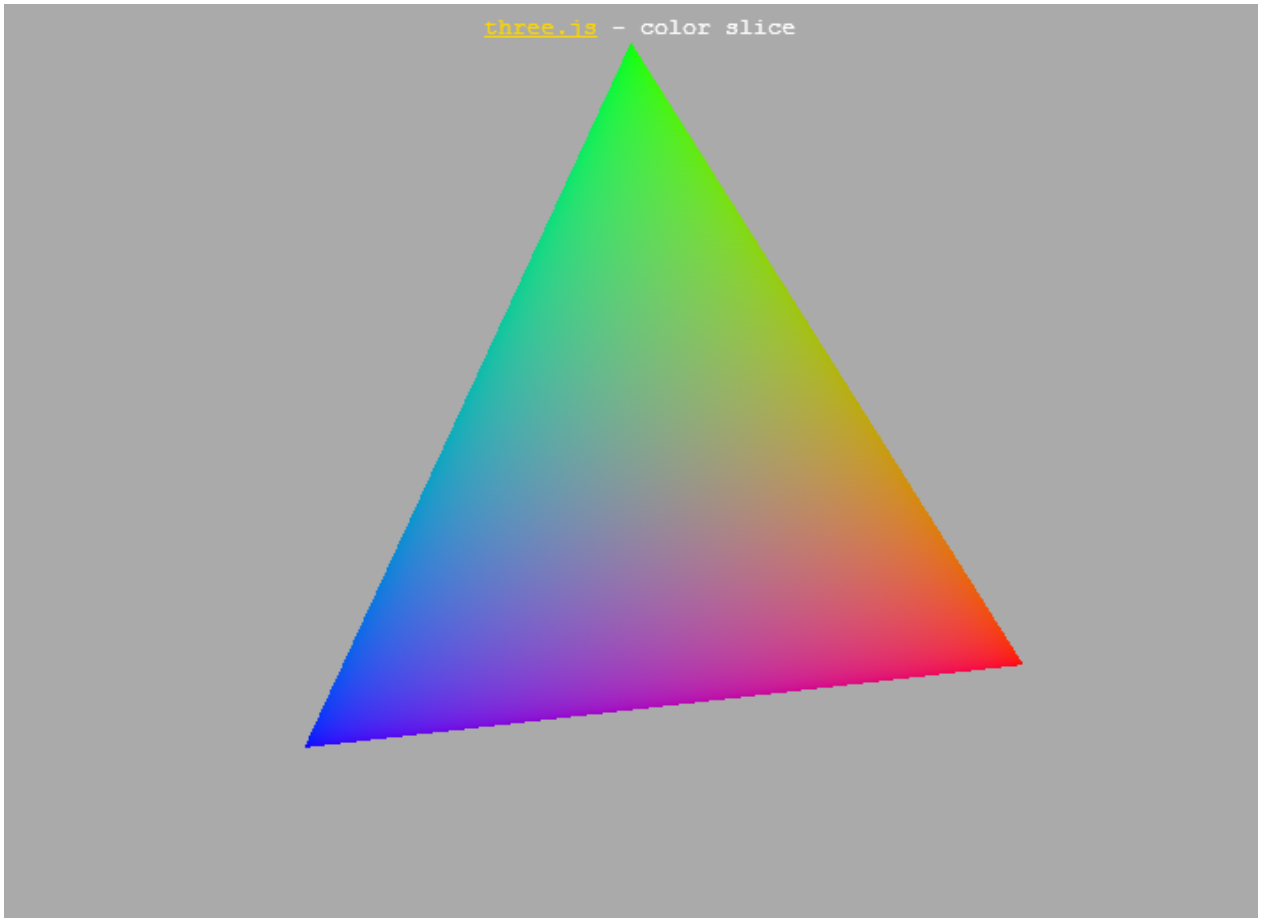


The final result is as we saw before.

You might want to try different colors at the corners, to see the results you can achieve.

[Additional Course Materials:
For another educational demo of vertex colors and how to set these per face of a cube, see [Lee Stemkoski's code](http://stemkoski.github.com/Three.js/Vertex-Colors.html).
]

Lesson: Color Interpolation



When we interpolate the red, green, and blue colors across a triangle, we get this result. It's important to understand how the triangle rasterization part of the pipeline actually interpolates across the triangle. Essentially, whatever is put at one vertex fades off across the triangle.

[green/black/black here]

For example, say we put green at the top corner and black at the two other corners. You can see in this case that the green is brightest at the corner and fades as we approach the other two corners. If you think of green as represented by 1 and black as zero, we linearly interpolate across the triangle and fade from 1 to 0.

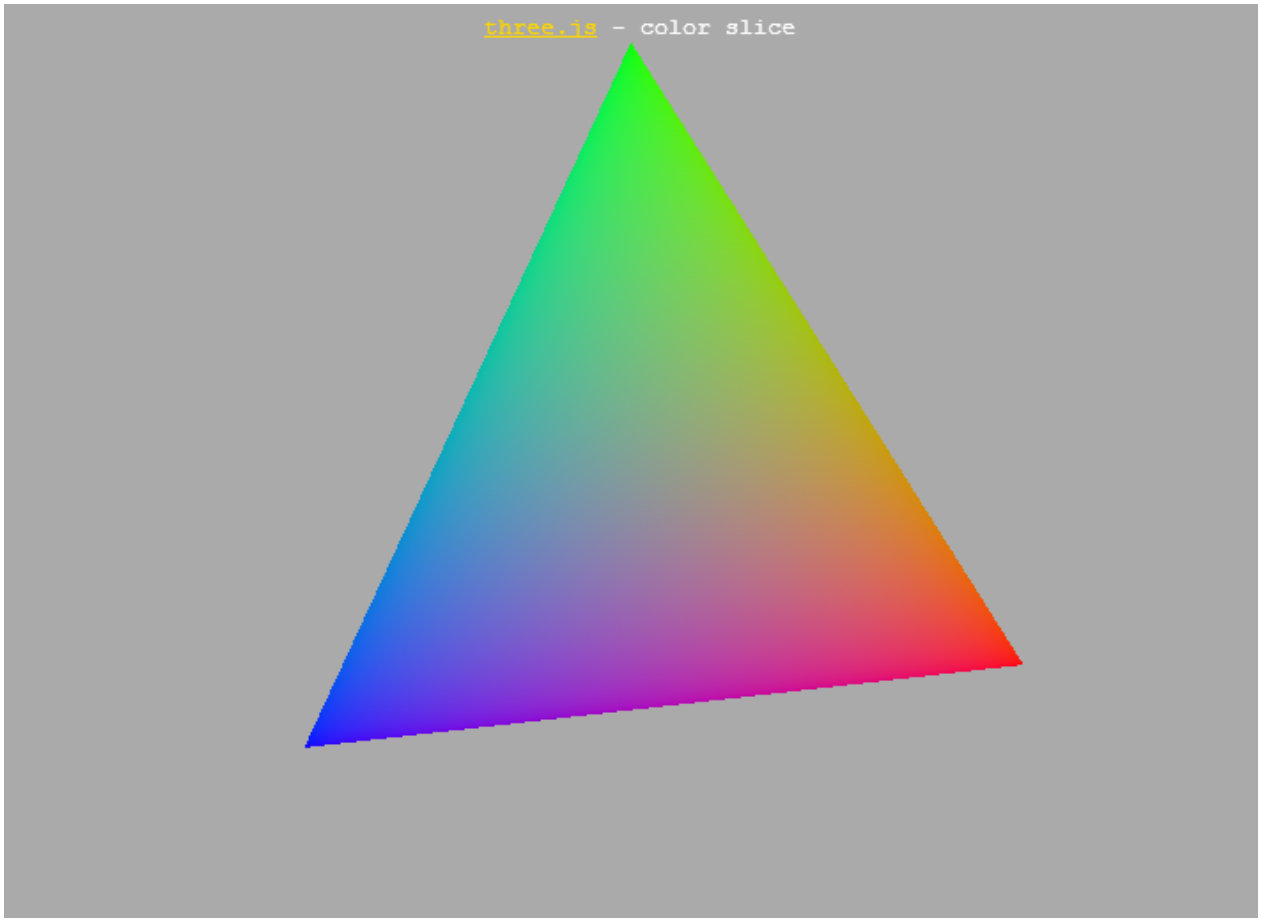
[blue corner]

This works for the other two corners in a similar fashion, with blue fading to black, as shown here. For comparison, here's the red component. Since light is additive, the GPU uses all three of these components to display the triangle. The GPU will interpolate *any* data put at its vertices in this way, a fact that we'll use later for material shading.

[Question: Triangle Center REMOVED]

What color is the center of the triangle, represented as three floating point numbers? By the “center”, I mean the point halfway between this corner and this edge, for all three corners.

[(draw three lines intersecting).]



The color is ____, ____, ____, in floating point representation.

[Answer REMOVED]

[show original]

Since this point is halfway between the full color and no color at all, for all three channels, the

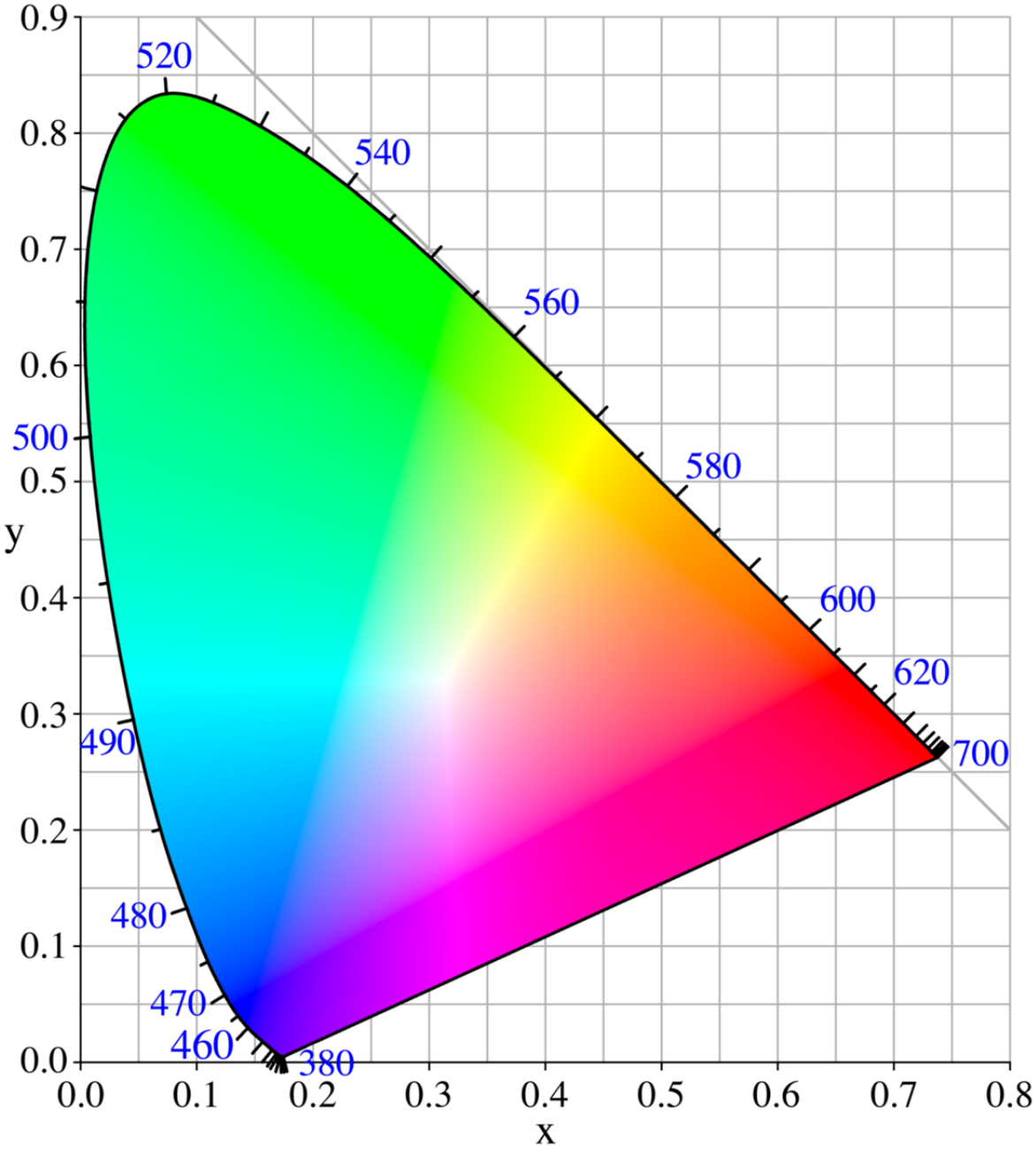
color is (0.5,0.5,0.5). [Turns out this is wrong! The three lines actually intersect at ⅔ / ⅓ across the lines themselves. Too tricky to explain correctly, plus gamma correction comes into play.]

Lesson: The Color Gamut

The color slice in the previous exercise gives a sense of the limits of what colors can be displayed on your monitor.

Look at this diagram:

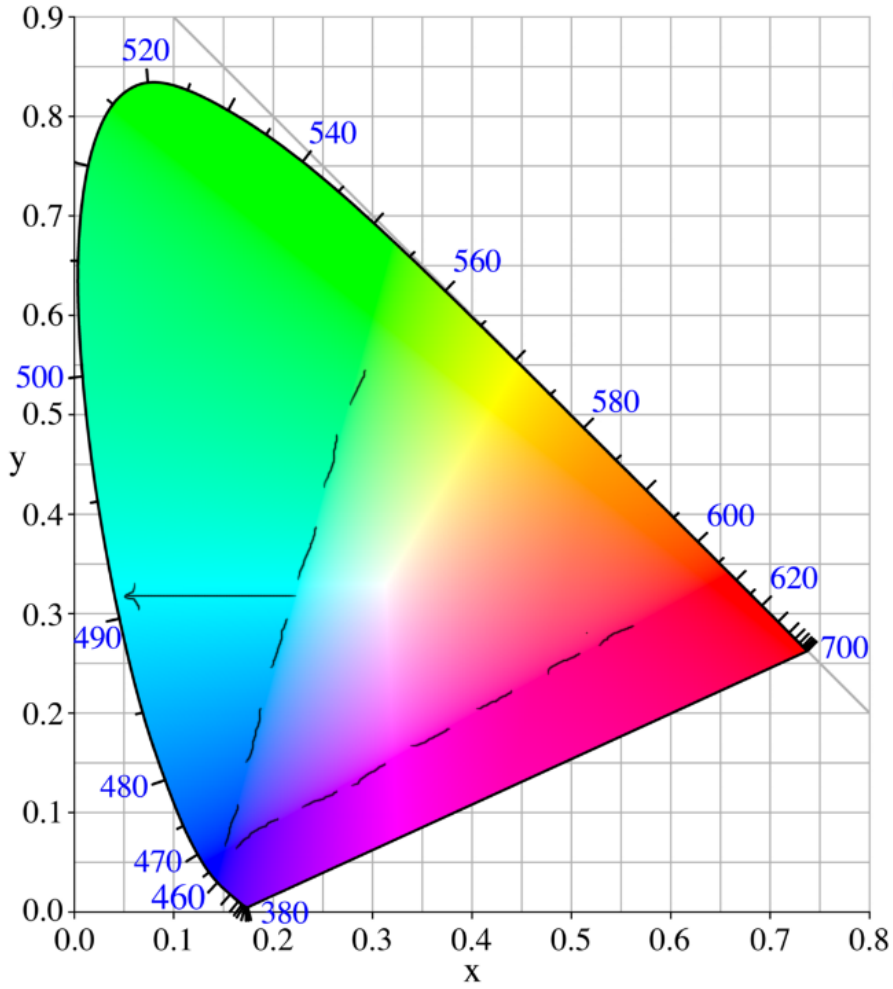
[http://upload.wikimedia.org/wikipedia/commons/b/b0/CIExy1931.png from http://en.wikipedia.org/wiki/CIE_1931_color_space]



We won't get heavily into this diagram, which shows what is called the **CIE XYZ color space**, [put that text to right of diagram]
but I want to point out a few key bits. Along this curved edge are the pure spectral colors. The values here are the wavelengths of light that produce various colors. You can in fact see the colors of the rainbow as you move along it: red, orange, yellow, green, blue, and so on.

You might also notice a funny artifact along here and here. I find these lines will show up more strongly if I look at my computer screen from a different angle.

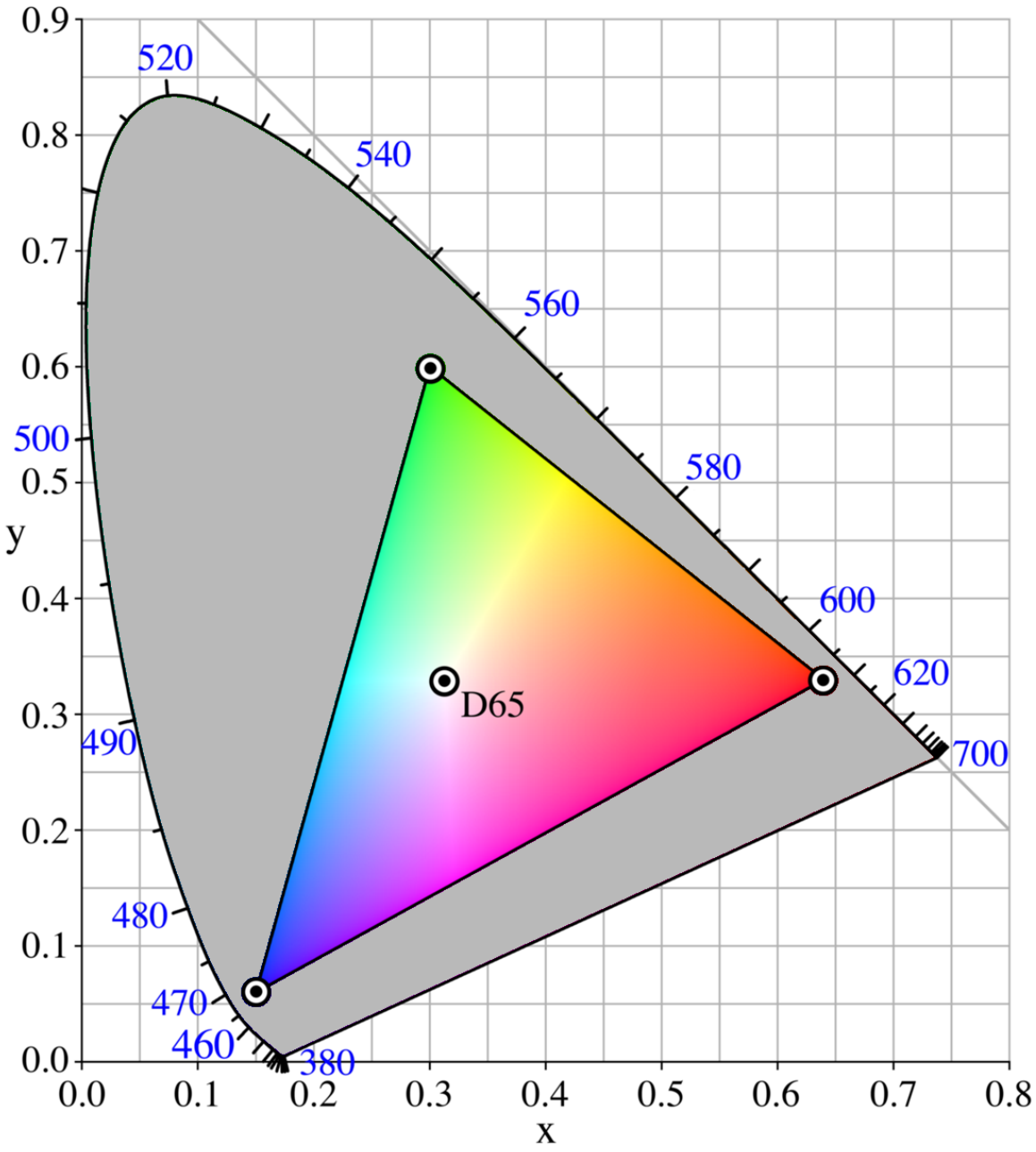
[point to the color Mach bands visible, emanating from near the lower left curve corner. - dashed lines are where I'll move the pen on the image but not actually draw. The arrow I'll actually draw.]



It turns out that the pure spectral colors on this outer rim are not something a monitor can display. The pixels in a monitor each have a red, green, and blue component, and none of these components is a pure spectral color but rather a mix of differing wavelengths of light. What this means is that our monitors have limits on the colors they display.

What is happening outside of these dashed lines is that the illustrator is just faking it by picking the best color possible and repeating it on out to the edge of the curve. What the diagram of displayable colors looks like is this:

[http://en.wikipedia.org/wiki/File:CIExy1931_sRGB_gamut_D65.png from http://en.wikipedia.org/wiki/RGB_color_model]



This triangle is called the device's "color gamut". The vertices are the same RGB vertices we used to form our own color triangle, and is a slice out of the color cube.

This whole area reminds me of seeing advertisements for TVs on my television. The announcer would talk about how their new TV's image was brighter, sharper, and with richer colors. On the screen they'd be showing some bright, crisp video. But of course I was seeing this video with my old TV, so there wasn't much point in them showing anything in the advertisement at all.

The key takeaway in all this is that the RGB color space is directly connected to the monitor

itself, and that monitors have limits to what colors they can display. The idea of a color gamut is also important in printing. For example, for high-end printing, custom inks can be used to increase the extents of the gamut.

[Instructor's Comment: For further information on this subject, the CIE XYZ article http://en.wikipedia.org/wiki/CIE_1931 and Gamut article http://en.wikipedia.org/wiki/Color_gamut should be of interest.

[Something to think about and try: what if different colors are put at the corners of the triangle? For example, white (1,1,1) or cyan (0,1,1) or some intermediate grays (e.g. 0.7,0.7,0.7).

[A recent cookbook was printed with high-quality inks to increase the color gamut http://modernistcuisine.com/2010/12/modernist-quality/ .

[In preparing this lesson I learned that Isaac Newton, who first named the rainbow colors, started with five colors and added orange and indigo later. What he called blue we actually call cyan, and his indigo is our blue. Read more in this Wikipedia article http://en.wikipedia.org/wiki/Rainbow#Number_of_colours_in_spectrum_or_rainbow

[this next one is a long aside - it could be a separate lesson but is not critical for interactive rendering and we blithely ignore it 99% of the time. However, I think it's worth mentioning, so hope the notes makes sense as a place to put it. It's a huge misconception that two materials give off the same spectrum of wavelengths if they're the same color.]

[I should mention that the idea of color is entirely in our heads. It turns out that different combinations of various wavelengths of light are interpreted by the cones in our eyes as having the same color. This makes a lot of sense if you think about it: there are three kinds of cones in our eyes: short, medium, and long. They take in a bunch of different wavelengths and come out with three stimulation values for the three different cone types for each area of the eye. So different distributions of wavelengths, of which there are a massive number of combinations, get filtered down into three values by the cones. Different spectral distributions can give the same color, even though the actual light wavelengths involved are not the same. Where things get interesting is that materials seen under one kind of light will look the same, while under another type of light will look different. See this article http://en.wikipedia.org/wiki/Metamerism_(color) for more on this fascinating topic. The takeaway is that RGB is not sufficient to perfectly represent light or material colors.

]

Lesson: A Simple Lighting Model

[Show lights diagram, one light and place in full shadow facing away is for ambient. Show photons bouncing around, hitting floor and hitting red sphere underside. Show eye. Leave room to the right for the four terms. Have a shiny mirror on the wall.]

We now know a lot about color, but nothing yet about materials. Say we have a red ball. How do we make the GPU compute how the lights in a scene and the ball's material interact, so that the ball looks real?

[Add these terms below to figure]

When performing interactive rendering, the standard way to think about a material's appearance is as a few different components added together:

Emissive +
Ambient +
Diffuse +
Specular

The idea of the emissive term is that it's for coloring glowing objects.. For example, a light bulb has its own color, and other light sources don't really affect it. In reality, the emissive term is simply a way to add in some constant color, regardless of the lighting conditions.

This second component, ambient, is a fudge factor, something added in so objects look better, especially in areas that are not directly lit. You will see it in various systems controlled in various ways. When all is said and done the ambient term helps compute a constant color value that is added to a fragment's final color. The emissive and ambient terms are kept separate as they fulfill different functions and often have different controls. For example, three.js allows you to set an ambient light that affects all materials with an ambient term.

Remember that we make a simplification with lights in a scene: we assume photons come only from lights and don't bounce around. In reality photons would reflect from the floor to the bottom of the sphere to our eye. However, with our assumption, the lower part of the sphere doesn't get any light and so would look black. The ambient term gives us at least some color in these types of areas, instead of being entirely black.

The diffuse and specular components are computed based on the lights in the scene, while

emissive and ambient are essentially independent of these. The diffuse material term can be thought of as a flat, matte finish, and specular can be thought of as the shininess of an object. In terms of computation, the specular term is also affected by the viewer's location, while the diffuse term is determined by only each light's location.

[text below put here]

Let's sum this up:

Surface color =
Emissive +
Ambient +
For each light:
Diffuse(light) + Specular(light, viewer)

[Draw math below]

The mathematical expression for this sort of thing looks more elaborate, but is really just compact:

$$C = E + A + \Sigma (D(L) + S(L,V))$$

If you haven't seen it before, Σ is the summation sign. It means to sum up all lights' diffuse and specular contributions.

Putting math in any lecture is often considered the kiss of death. The good news is that you can mostly ignore this equation here. It's just another way of showing all the previous text. I've put this math notation here mostly because you'll run across the summation term a lot when looking at material reflection models. Now I'm sure you know what it means.

[probably draw lines from each word to each symbol, to seal the deal.]

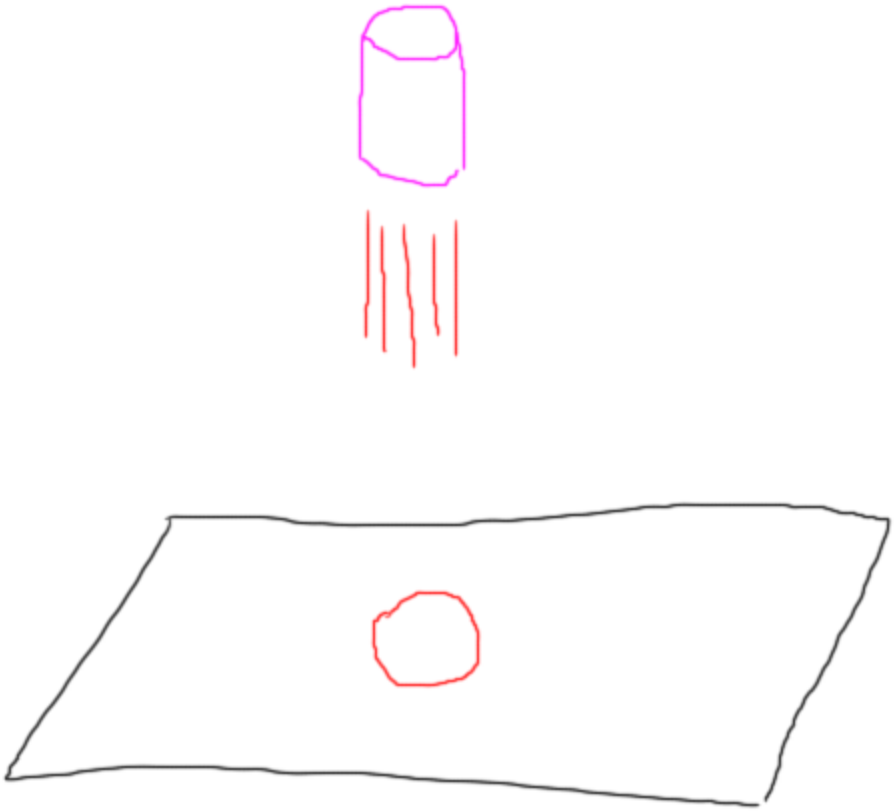
The emissive and ambient terms are pretty simple, so our focus for the rest of the unit will be looking at the diffuse and specular terms in depth.

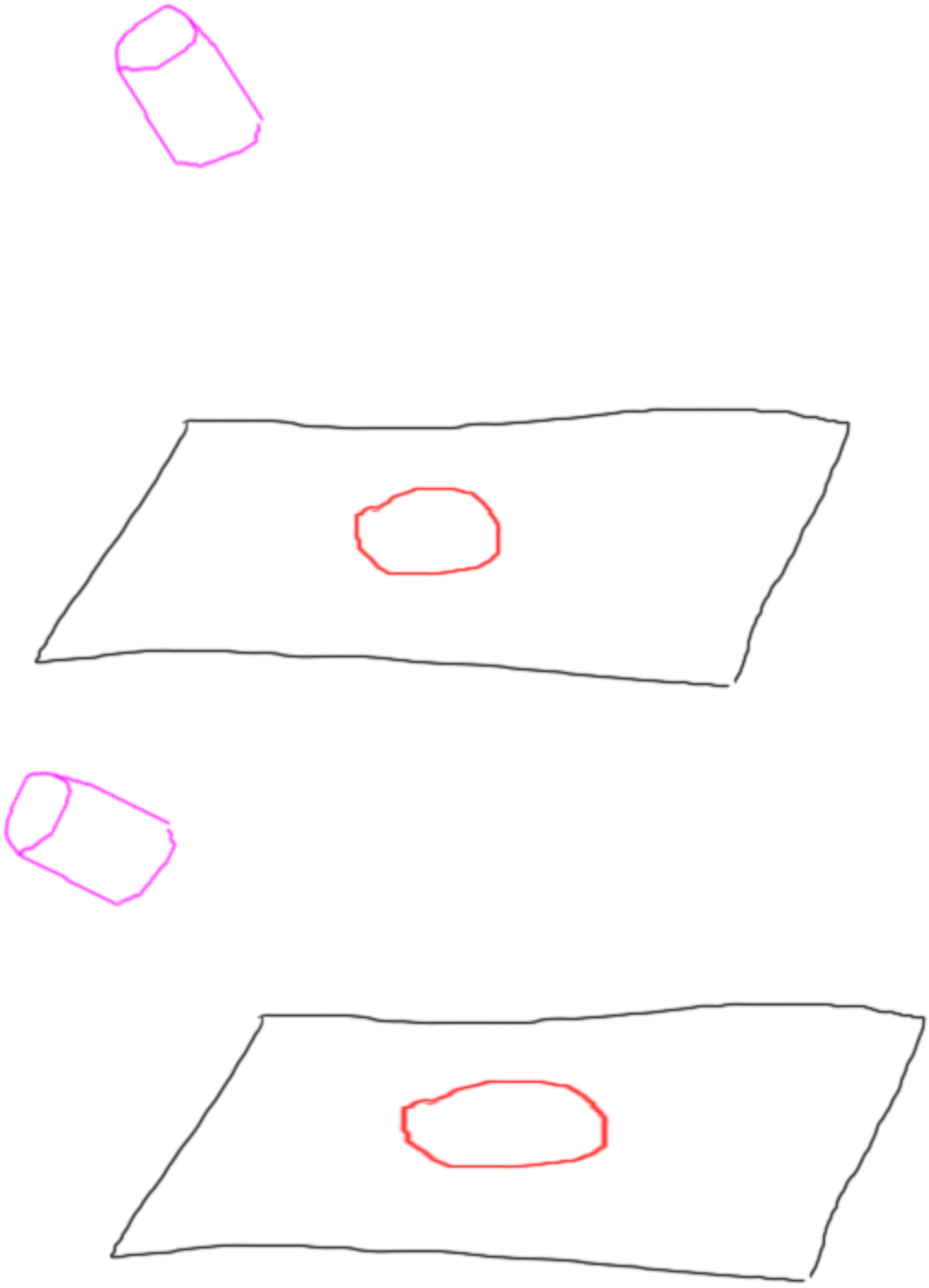
Lesson: Light on a Diffuse Plane

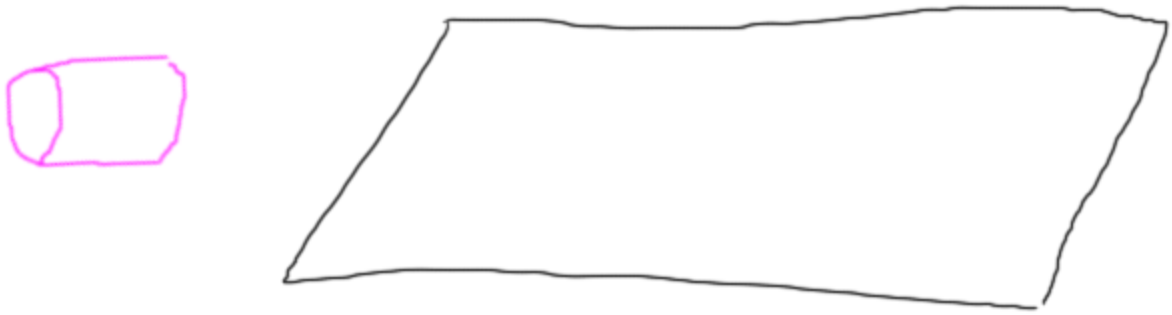
The diffuse term depends on the surface itself and the incoming light's direction.

[Gundega demo?]

[Question: Are rays of light worth showing in all views? Easy enough, but seemed like clutter]



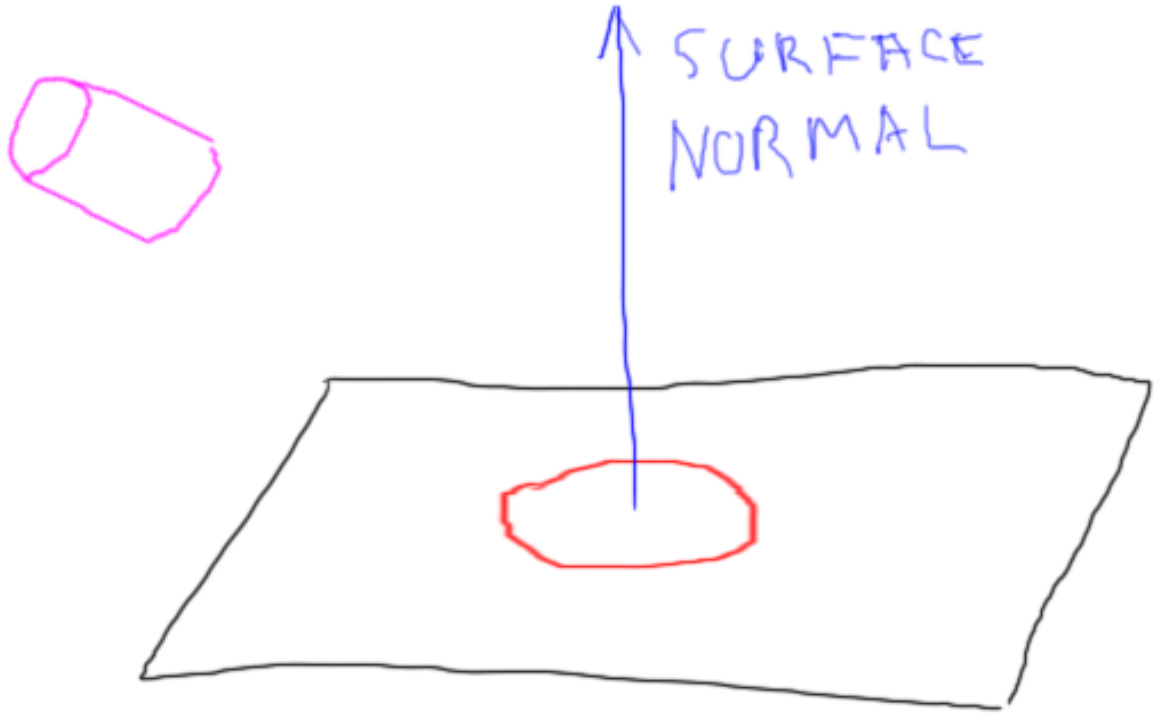




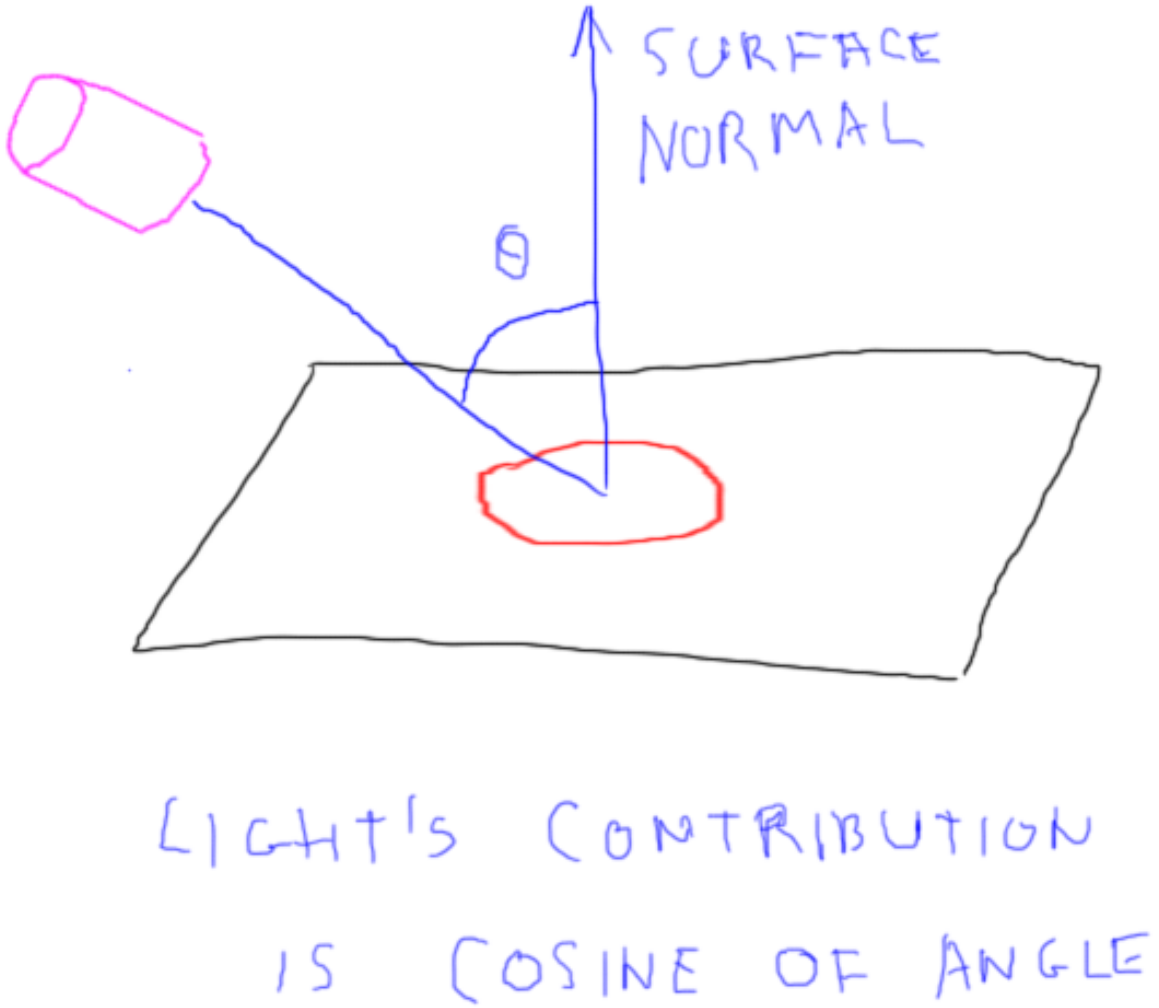
Think about a spotlight over a piece of paper on a desk. As the angle of the lamp changes with respect to the paper, the number of photons coming from the light are spread over a wider area. This results in the paper becoming dimmer. If the lamp moves to a horizontal position, no light (at least in theory) reaches the paper. Certainly if it moves under the desk, the light has no contribution.

[draw image below]

The way we compute this contribution is by using a surface's normal.
[draw normal, and label it]



The normal is a vector pointing in the direction the surface faces. It turns out that as the angle between the surface and the light increases
[show angle, vector towards light]



the light's contribution to the surface goes down by the cosine of this angle.

[draw on a new page of paper the cosine graph - label axes!]

Here's the cosine graph

So at an angle of 0 degrees, the cosine of this angle is 1; at 60 degrees, the cosine is one-half, at 90 degrees the cosine is 0, no contribution.
[show by using pen itself as a light vector, or better yet make a vector out of wire or something.]

Beyond this angle the cosine goes negative. Since we don't allow negative light (though we could!), any negative value is clamped to zero. This dropoff is called Lambert's cosine Law and dates back to the year 1760. The LambertMaterial in three.js is named after him.

[put this picture somewhere, from http://en.wikipedia.org/wiki/Johann_Heinrich_Lambert - put

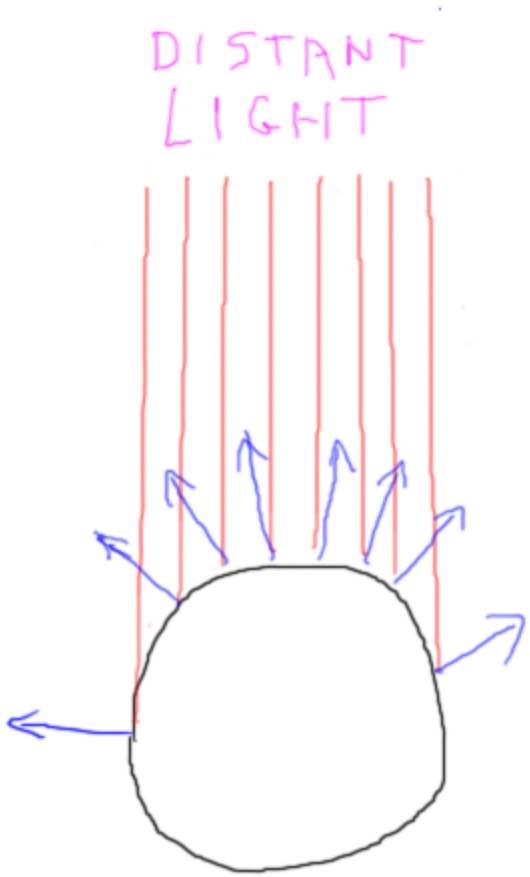
LambertMaterial]

Johann Heinrich Lambert was an extremely productive Swiss mathematician and scientist. He's most well-know for proving that PI is an irrational number.

Lesson: Light on a Diffuse Sphere

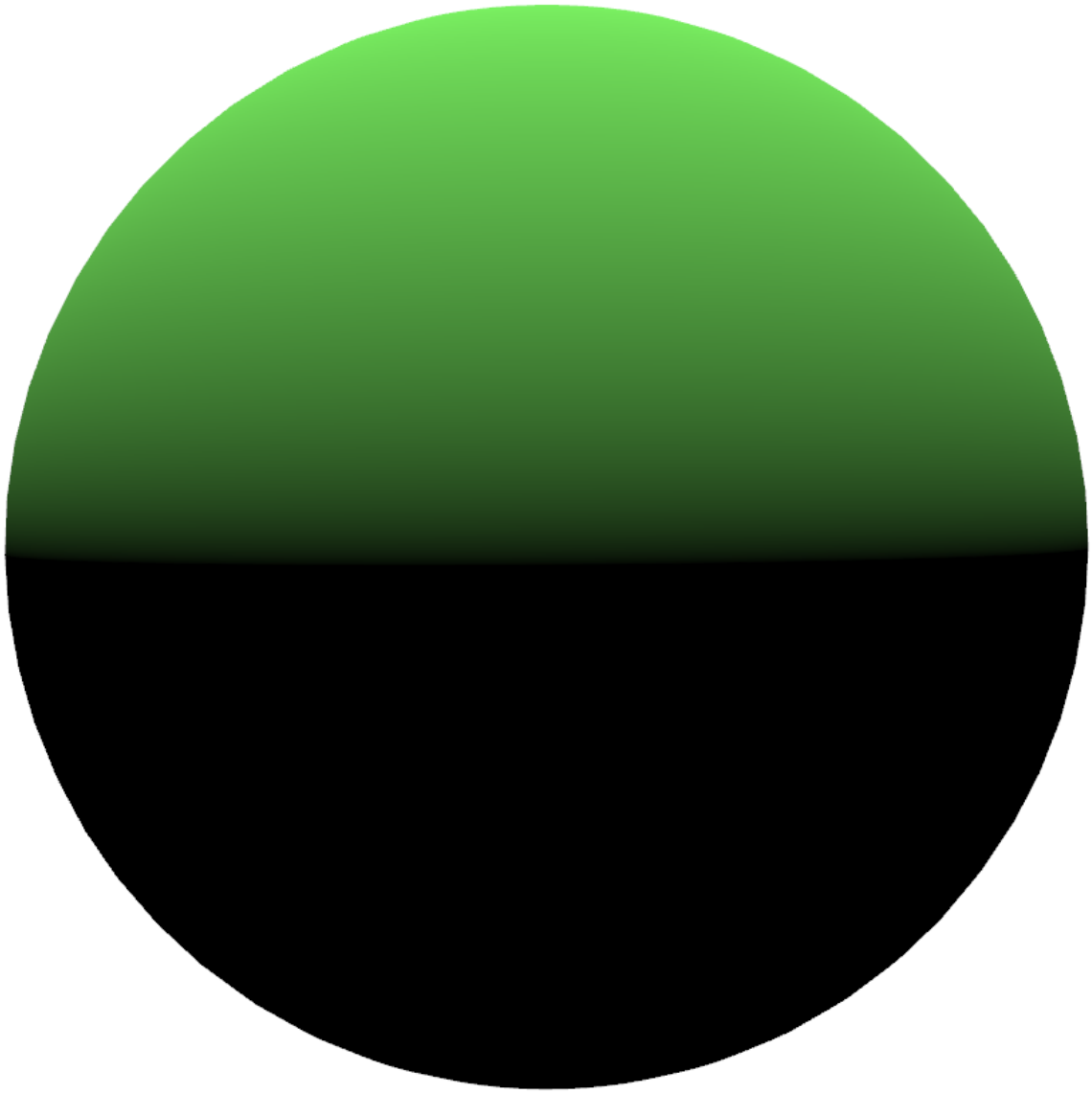
Let's flip this around, keeping the light fixed and showing the effect on a sphere. In fact, throughout this unit let's assume each light is extremely far away, like the sun, so has a direction instead of a position. That way, we don't have to think about the light's direction changing as the surface changes position.

[light and sphere]



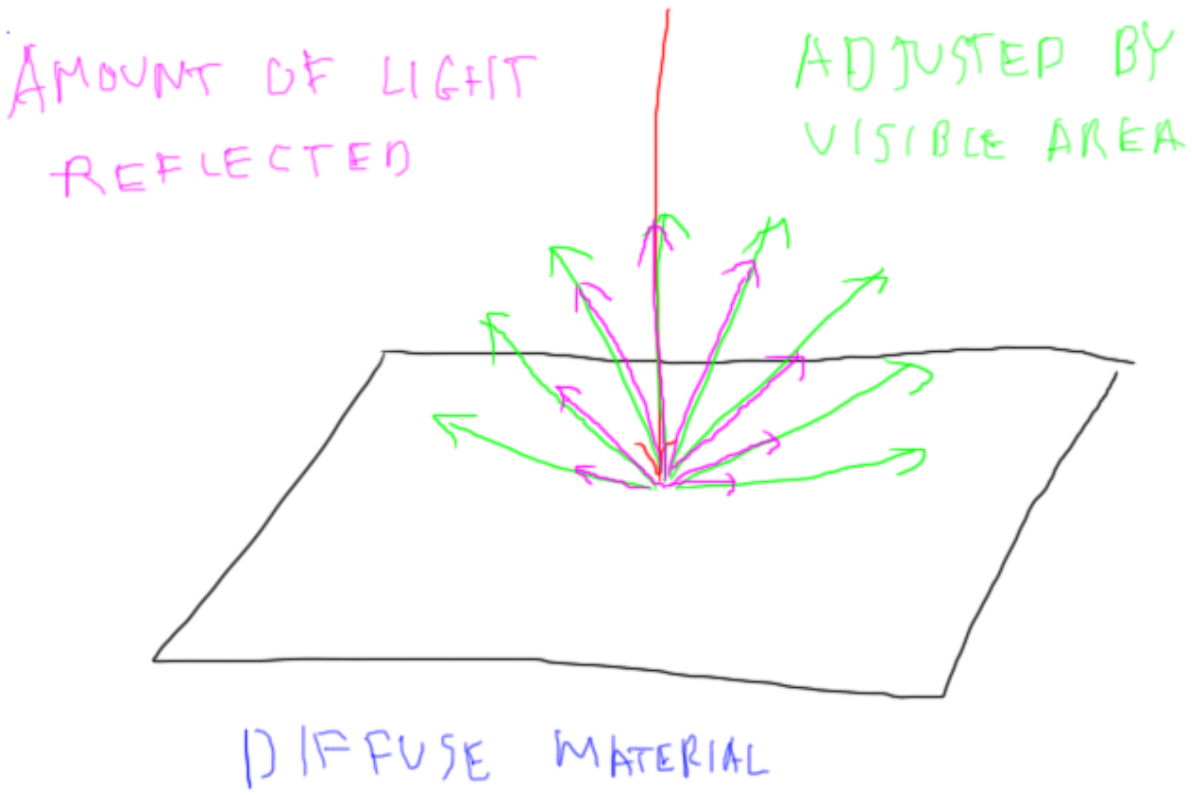
Here the sphere's normal changes with respect to the light. The light's effect is at a maximum at the top of the sphere, goes to zero at the equator, and stays that way since the bottom of the sphere faces away from the light.

[diffuse sphere here]



This is what this equation looks like when used in our reflection model.

[cosine distribution]



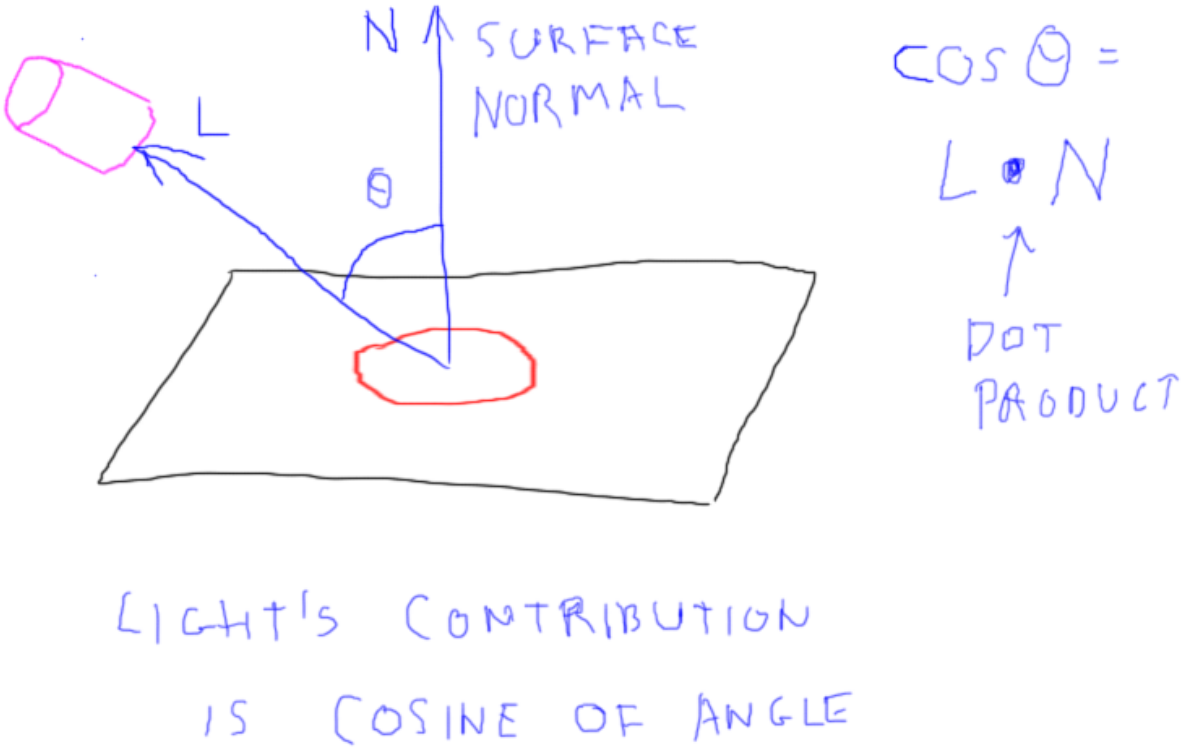
A diffuse material is defined as one that scatters light in all directions above the plane according to this cosine law. That is, the amount of light from a single point on the surface drops off with the cosine of the angle of the surface compared to the *eye*. So, you might think less light would come when the eye looks at the surface from a shallow angle.

However, a balancing factor is that, as the angle of the surface to the viewer increases, the area of the surface visible through a pixel proportionally increases. That is, you see more of the surface, even though less light per unit area is reflected. These offsetting effects give the net result we want: the viewer's location doesn't matter at all, as the shade of the surface depends only on the angle of the light itself. Not needing the eye direction makes the diffuse term faster to compute, which is always important in interactive rendering.

Lesson: Normalizing a Vector

To shade a diffuse surface, we need the cosine of the angle between the direction to the light source and the surface's normal.

[Show the two vectors for a plane & light again]



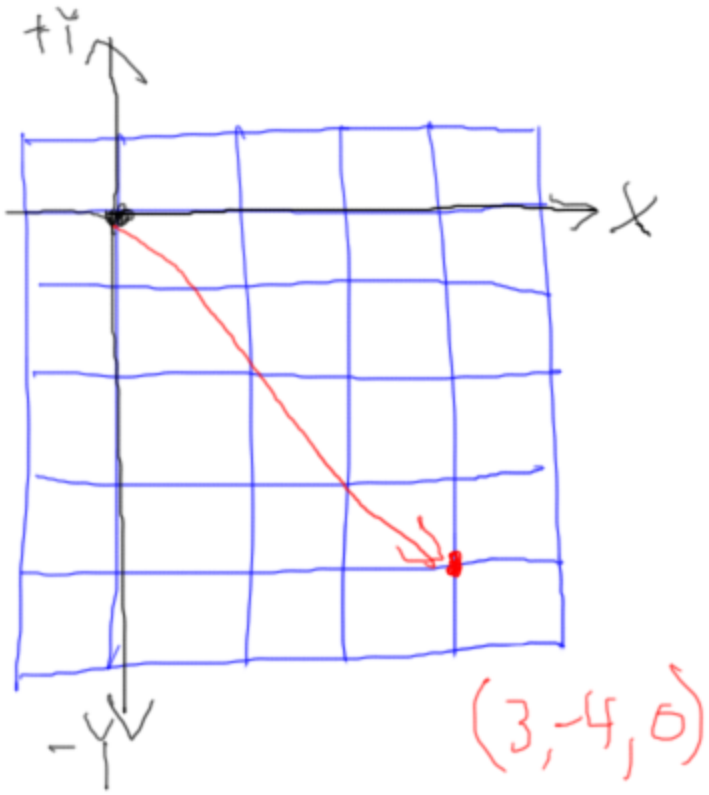
We can perform a vector operation called the **dot product** to directly compute this cosine. First we must **normalize** the surface normal and the vector to the light. Normalizing means rescaling a vector so it has a length of 1. Normalized vectors are the norm - ha, funny - in reflection models. As we'll see, the dot product of two normalized vectors gives a value between -1 and 1, which will prove useful in computing the effect of lighting.

[put text below to here

For example, say I have the vector (3,-4,0)

(3,-4,0)

[graph below needs to have Z labeled as coming out of paper, not Y. Draw graph on left side, we'll use the right.]



To normalize it, I find the length of the vector. This is simply the Pythagorean Theorem:

[draw line by line. See final figure below.]

Length = $\sqrt{3^2 + (-4)^2 + 0^2}$ **=** $\sqrt{9 + 16 + 0}$ **= 5**

Take each component of the vector, square it:

[draw next line.

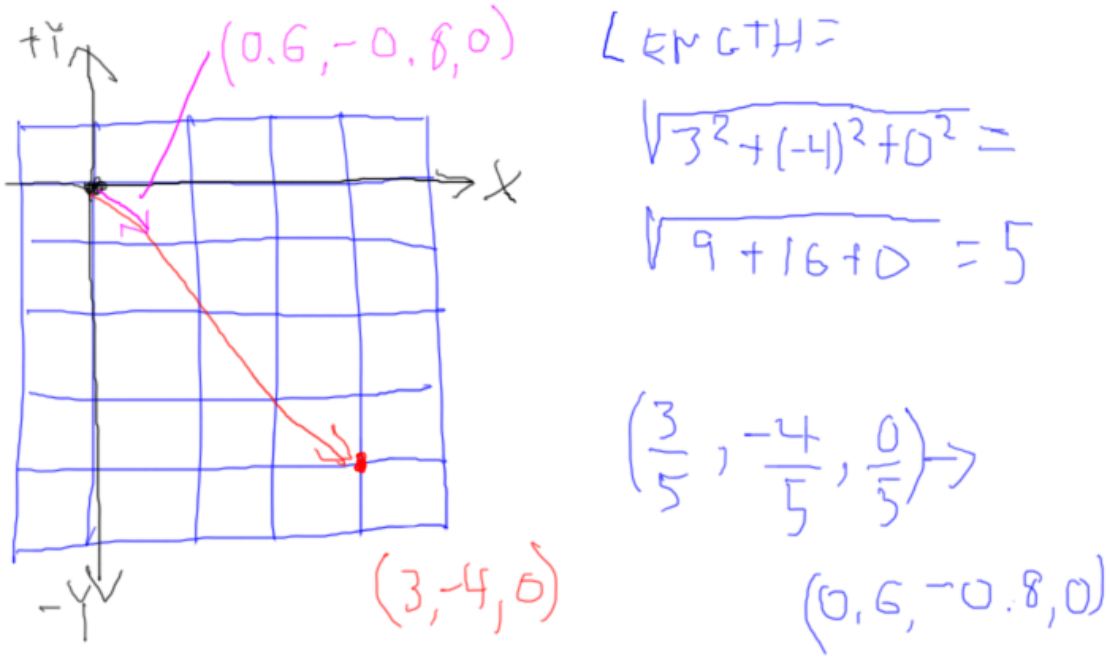
3 squared is 9, 4 squared is 16, and 0 squared is 0.

[draw next line]

Take the square root and you get the length of the vector, 5. By dividing the vector by its length, you get a normalized vector. So (3,-4,0) normalized is

[shown on figure lower down]

(3/5, -4/5, 0/5) = (0.6, -0.8, 0)



Looking at this vector, it goes the same direction, but only travels one unit.

Note that normalizing a vector that's already normalized leaves it unchanged. Try normalizing this vector again and you'll find the length is 1. Dividing the vector by a length of 1 does nothing.

It's very handy to normalize vectors in this way. Usually we store the surface normals as normalized vectors. We'll see exactly why in the next lesson. For now, a quick quiz.

Quiz: Vector Length

What is length of this vector?

(6, -15, 10)

Answer

The answer is the square root of the sum of the squares:

$$\sqrt{6 * 6 + (-15) * (-15) + 10 * 10} = \sqrt{361} = 19$$

[Instructor Comments, if any: There are plenty of these Pythagorean quadruples, see http://en.wikipedia.org/wiki/Pythagorean_quadruple]

Lesson: The Dot Product

Given two normalized vectors, the cosine of the angle between them is computed by the dot product operation:

A * B = Ax * Bx + Ay * By + Az * Bz

For example, if we have these two normalized vectors:

[Draw approximately what these might look like]

(0.80, -0.36, 0.48)
(0.00, -0.60, 0.80)

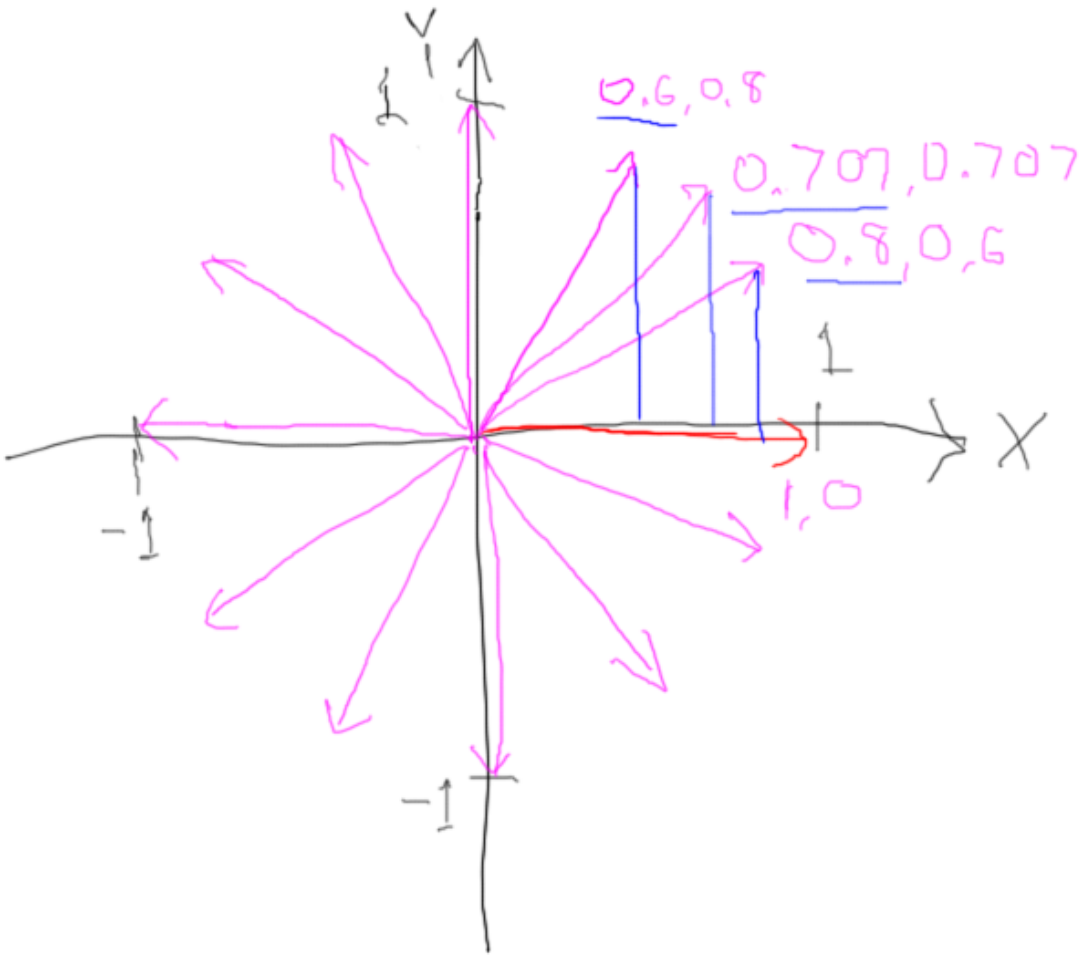
the dot product is then:

0.80 * 0.00 + (-0.36) * (-0.60) + 0.48 * 0.80 = 0.00 + 0.216 + 0.384 = 0.6

Out of curiosity, if we take the arccosine of this value, 0.6, we get 53.13 degrees. However, we don't care about the angle, we truly want just the cosine, as this is then how much we'll scale the light's contribution to the surface.

The dot product is a little bit mysterious, and is ***so*** fundamental to a large part of computer graphics, that it's worth building up some understand of what it means. Why does this operation have any connection with cosine?

[Draw 1,0,0, vector, then add new vectors as I talk]



Let's start with a normalized vector (1,0,0), pointing along the X axis. If I draw a normalized vector here, at (0.8,0.6,0), the dot product is simply the X component, 0.8.

With the vector (0.707,0.707,1), a 45 degree angle, the dot product is again the X component, 0.707.

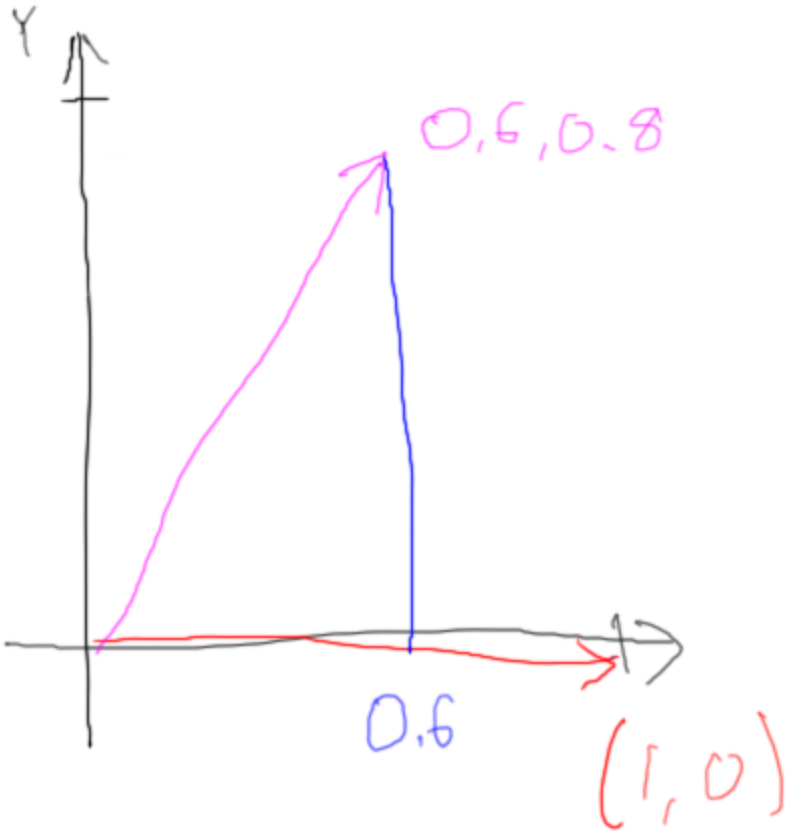
With the vector (0.6,0.8,0), the X component is 0.6. It's always the X component.

So continuing on with (0,1,0), the value is 0. Two vectors perpendicular to each other always have a dot product of 0. If you work your way around with more vectors, you will form a circle. As you may recall from geometry, the cosine is often defined as the distance in X of a point on this circle. So as this vector angle increases, the cosine of this angle is computed by the dot product. I've known this fact for 30 years, but it's still slightly magical to me that taking a dot product of two normalized vectors - which just multiplies and adds them together - gives you a trigonometric

value.

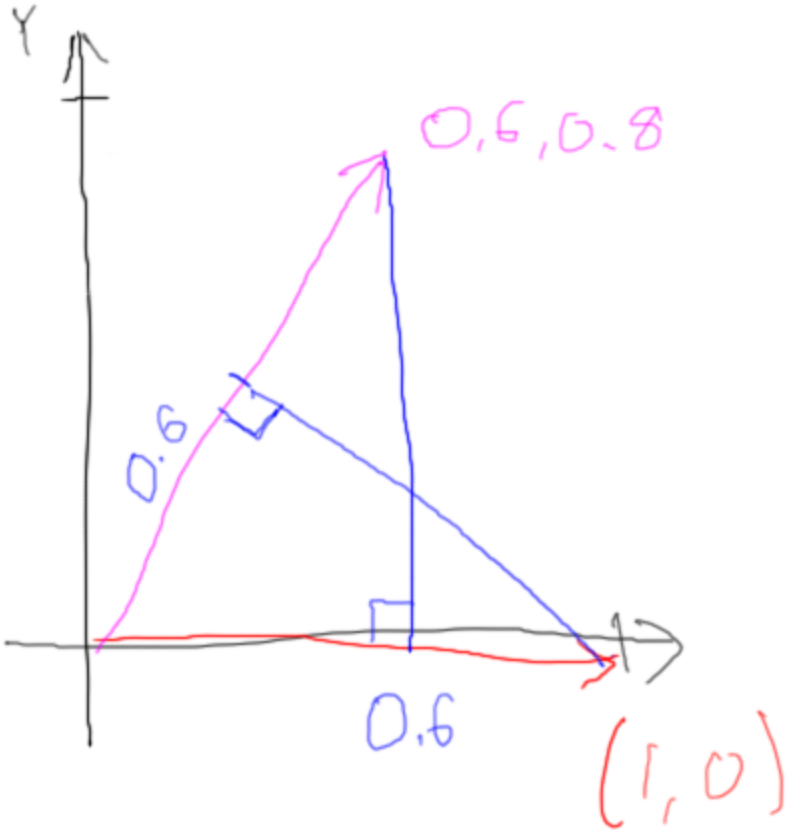
Geometrically, what is happening is that one normalized vector is being projected on to the other. That is, the dot product shows how far one vector extends with respect to some given vector. So for (0.6,0.8,0), the vector extends 0.6 along our initial vector.

[Show how 0.6 vector projects onto 1.0 vector - put the extra 0's here for Z]



Another way to say this is that the one vector projects onto the other vector to give this distance.

This works both ways: our initial vector can be projected onto the second vector and you get the same length of 0.6. In other words, the dot product operation is commutative: A dot B gives the same answer as B dot A.
[Add how 1.0 vector projects onto 0.6 vector - again, add Z = 0 to these!]



That's it in a nutshell, the dot product projects one vector onto another vector. If both vectors are normalized, the dot product gives the cosine between them. We'll see in the next unit, on transforms, what happens when one or both vectors are not normalized. For computing the shade of a material, however, we have the tool we need, so let's use it!

Exercise: Diffuse Material

[record - we need the code in a figure, and the exercise]

In this exercise you'll set up a diffuse material. The sphere in this scene starts with a solid color, unaffected by the lighting. To give this object a diffuse material, the first step is to change the material from three.js's basic material to Lambert:

MeshBasicMaterial → MeshLambertMaterial

The LambertMaterial has a few separate color components. The equation that the Lambert material uses is:

[this should be hand-written for sure]

C = ambient + color * Σ(N * Li)

This second term is the diffuse contribution. It is the color of the material times the sum of the lights' contributions, which is the surface normal dotted with each light's direction.

In the exercise you'll see code like this:

[need code bits]

```
material = new THREE.MeshBasicMaterial( { color: 0x80fc66, shading:
THREE.FlatShading } );
```

This code sets up the solid color. By switching this material to the LambertMaterial, the color set here is for the diffuse component.

You could also explicitly set this color with the following code, as usual:

[need code bits for all the following]

```
material.color.setRGB( … );
```

If you want the value of one channel of the color, you can access it like this, for example:

```
var newRed = material.color.r * 0.7;
```

This first term of the equation is the material's ambient color setting. The ambient color is called "ambient":

```
material.ambient.setRGB( … );
```

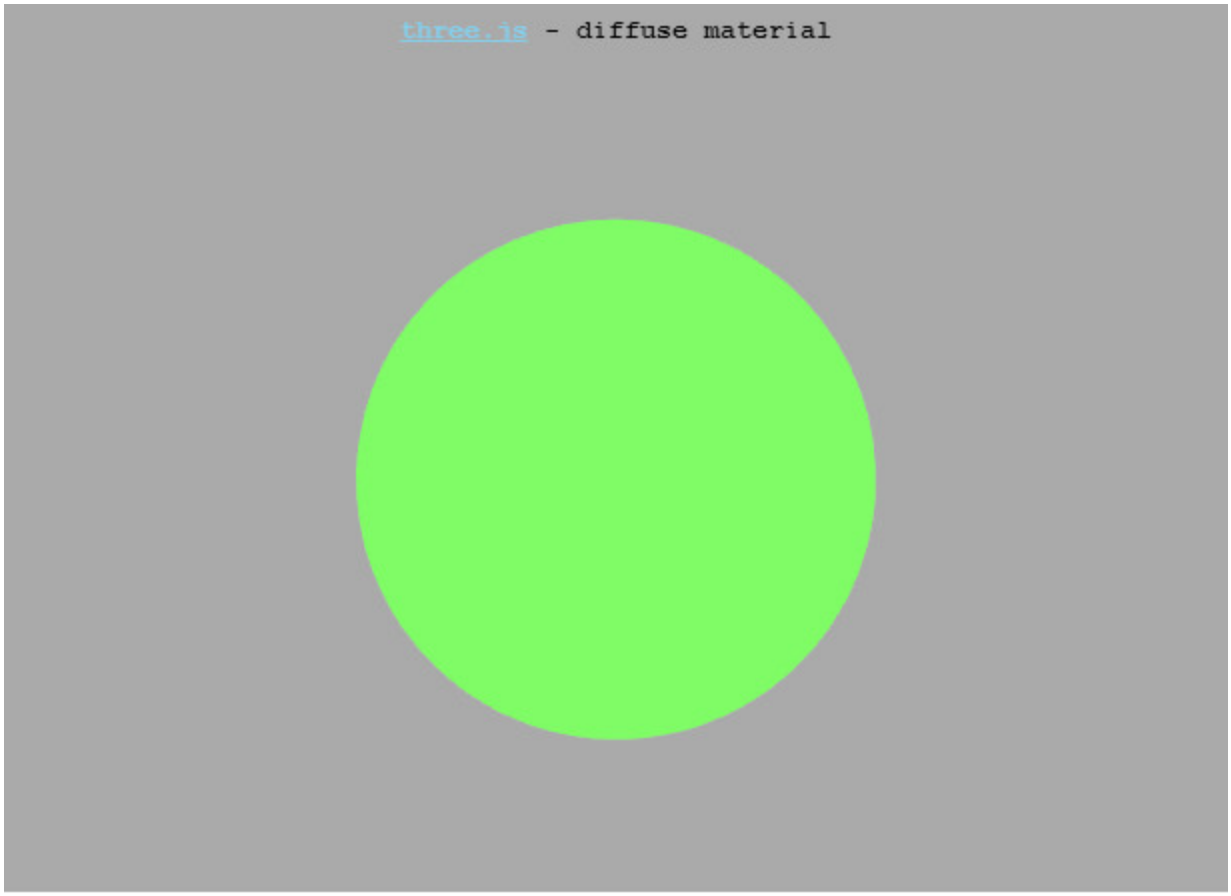
The ambient color is multiplied by the ambient light's color, which I've set to white. Your task is to set this ambient color to be the material color times **0.4**.

When you're done, you should see something like this:

[shaded sphere here]

[Student exercise is at http://www.realtimerendering.com/udacity/?load=unit3/diffuse-material.js]

[Student starts with this on screen]



Answer: Diffuse Material

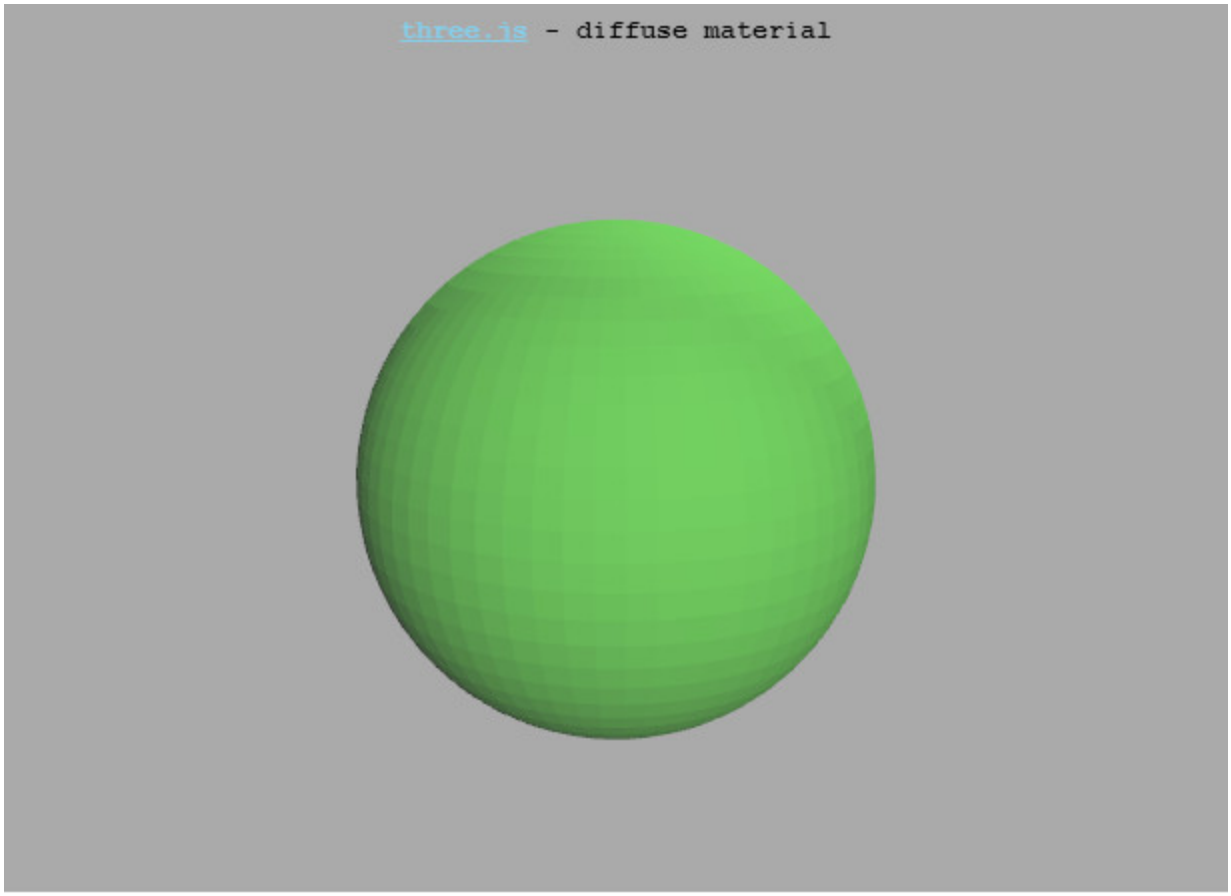
[need code]

Notice that when you rotate around the finished scene, the effect of lighting on the objects don't change as the view changes.

The other thing you'll notice is that it looks pretty bad! The sphere is not smooth - we see the facets, the underlying triangles. This is truly how a teapot made out of triangles would look in the real world. How do we make it look good in our virtual world?

[Solution at http://www.realtimerendering.com/udacity/?load=unit3/smooth-lambert.js]

[I've actually changed this exercise a bit, but the image below is close; it's representative.]

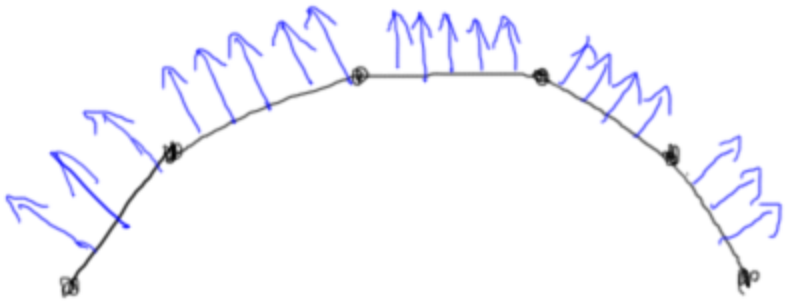


Lesson: Shading Normals

[**Shading normal vs. Geometric Normal**]

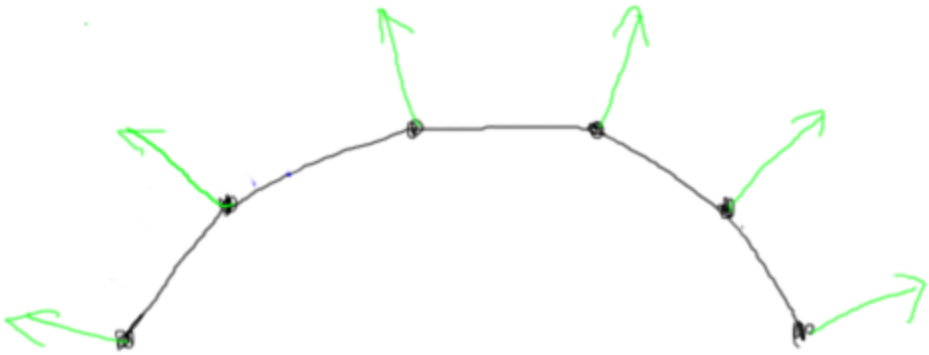
To make a tessellated surface made of triangles look smooth, we introduce the idea of a shading normal. For our tessellated sphere, the geometric normals for the triangles look like this:

[straight normals on sphere silhouette - NOTE TO SELF, save sphere outline without normals, as I'll reuse this in the next drawing.]



The facets making up the sphere are just an approximation. What we can also store at each vertex is the shading normal, which is what the surface's normal is at that point:

[shading normals on sphere]



These shading normals are created by the application in some way. Sometimes you can get the normal from the underlying surface. A sphere is a good example, as the normal at a point is always the direction from the sphere's center. However, a shading normal is entirely made up; you can set it however you want. In fact, there are different ways of averaging the geometric normals of the triangles that share a vertex. None of these ways is perfect, each has its own strengths and weaknesses.

This is different than a geometric normal. If you're given a triangle, you can always get its geometric normal - that's a property that is derived from the triangle's points.

[Additional course material: point at Max and the other guys' methods for getting shading normals.]

Exercise: Enable Smooth Shading

The first step we did to use a diffuse material was to change the MeshBasicMaterial to MeshLambertMaterial.

MeshBasicMaterial → MeshLambertMaterial

The next step is to simply not use flat shading. This is simple in three.js. Making this change is often more involved in other systems, and ultimately depends on how the geometry is structured and the shaders are written.

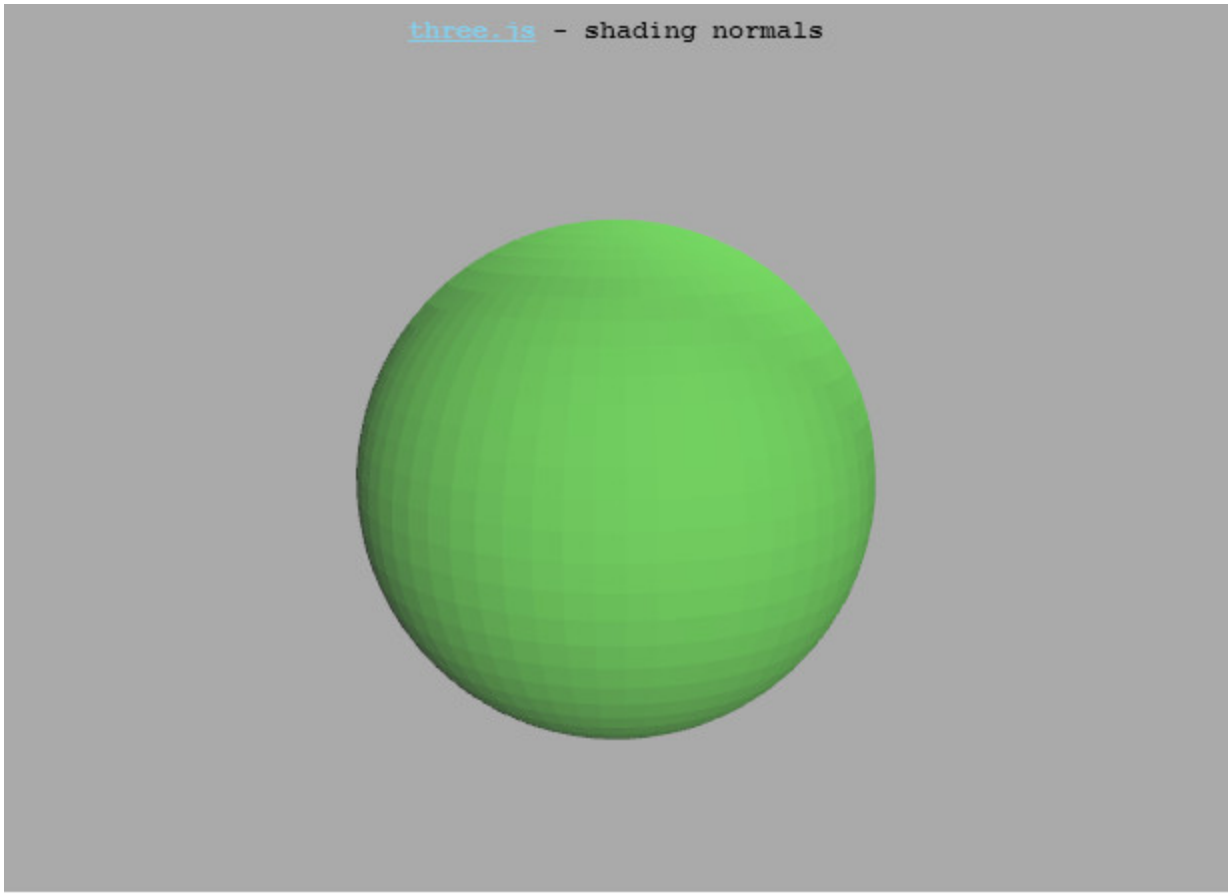
Here's how the material was declared:

material = new THREE.MeshLambertMaterial({ color: 0x80fc66, shading: THREE.FlatShading });

Setting the shading to FlatShading has the effect of using the geometric normals. Simply removing this setting will give you a smooth sphere. Go do that now.

[exercise code at http://www.realtimerendering.com/udacity/?load=unit3/smooth-lambert.js]

[what student starts with, more or less]



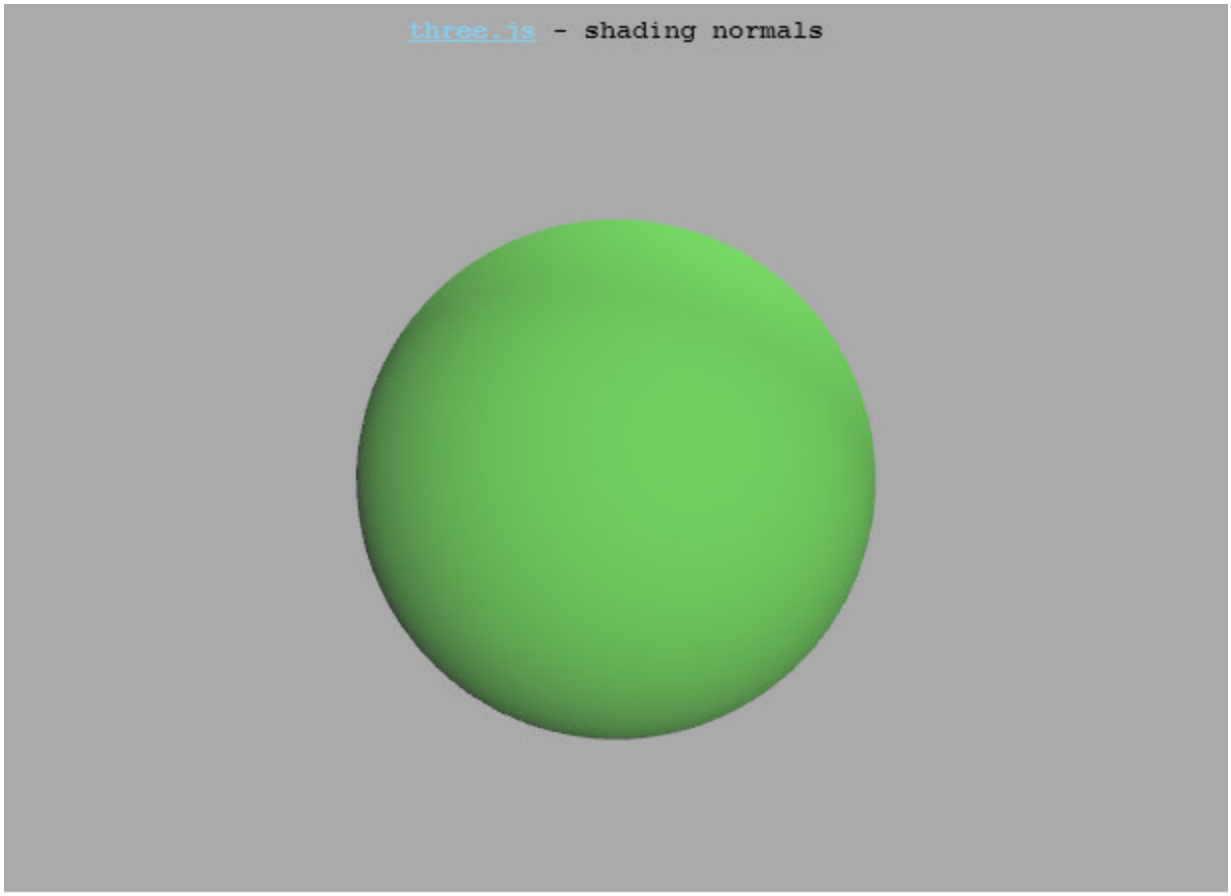
Answer

The resulting object now uses a smooth shading normal stored at each vertex to compute the effect of lighting and interpolate the result.

This is the usual way to use the this Lambertian, diffuse material. However, now you know what's going on behind the scene, with the application supplying tessellated geometry and smooth shading normals.

[solution at: *redacted*]

[what student ends with]



Lesson: Ka, Kd, and HSL

[recorded 3/26]

[put text below]

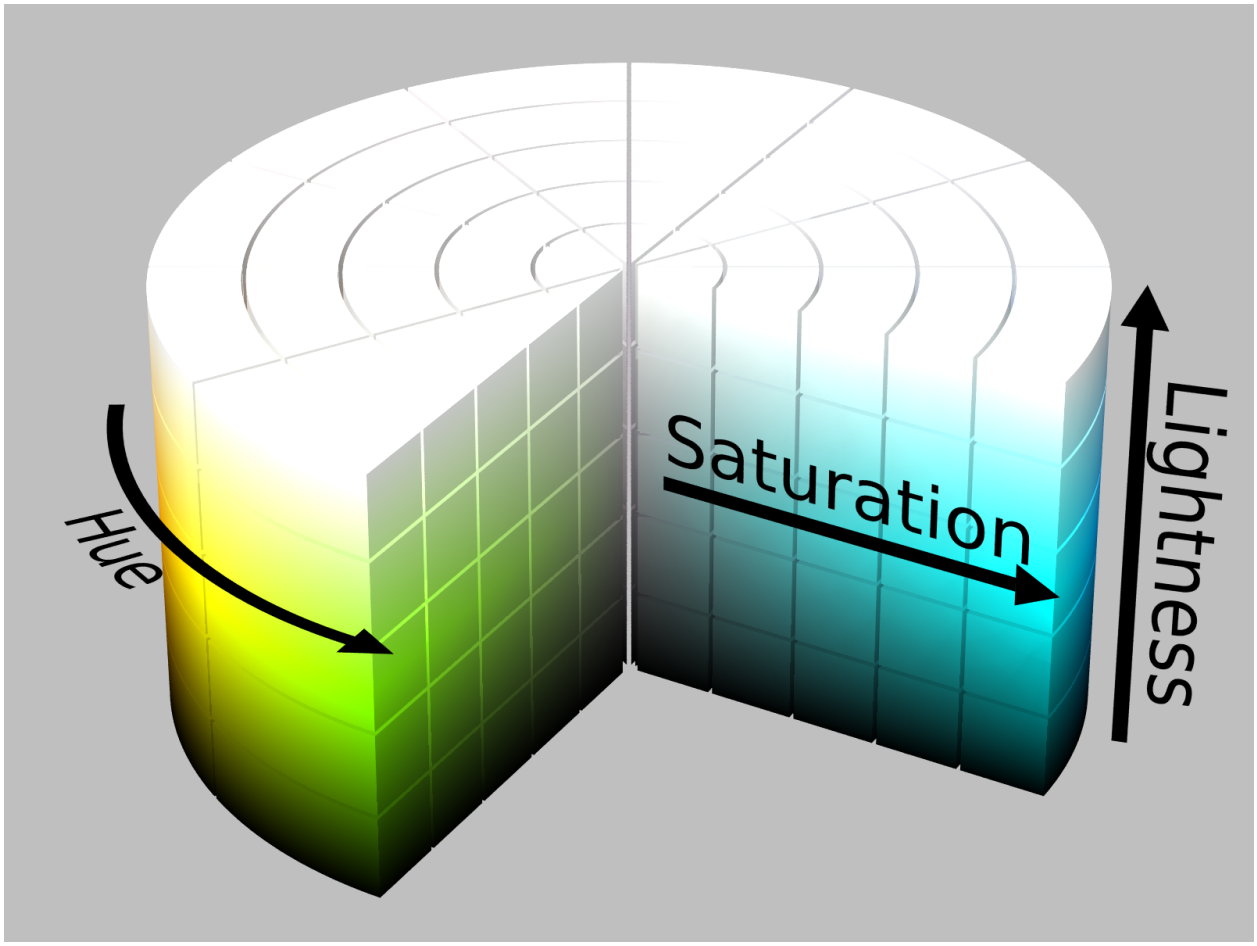
Now that you understand the components of the diffuse reflection model, give them a try. In the demo that follows we use the terms Ka and Kd in the reflection equation. These are simply numbers used to scale the effect of the ambient and diffuse contributions up and down. Ka is the grayscale level of the ambient light, and Kd is a handy way to modify the diffuse contribution. To sum up the equation we use for this demo, it's:

fragment color = Ka * material + Kd * material * (N*L)

This is actually a bit simplified from what three.js allows. In three.js you can set an entirely different ambient color, though to be honest this is a peculiar thing to do.

Also in this demo you'll see a different color space used: instead of RGB, it's Hue, Saturation,

and Lightness, "***HSL***" for short. "Hue" means the color, essentially. "Saturation" is how much the hue is used. "Lightness" is a scale factor, letting you easily make the color itself brighter or darker. Here's a visualization of this color space:



The three.js library lets you use HSL to set colors, for example this code:

```
var color = new Color();

color.setHSL( 0.117, 0.937, 0.557 );
```

```
var color = new Color();
color.setHSL( 0.117, 0.937, 0.557 );
```

gives a mustard color.

So, go try out the demo. Vary the ambient and diffuse contributions, change the material color,

move the view around, and see the effect. It's worth your while to catch your breath and play around a bit, so that you truly understand this reflection model.

[Instructor Comments: The demo program to try out is exercises/unit3_lambert_demo.html - TODO add to demos]

For more on HSL and a related space called HSV, such as the conversion to and from RGB, see this page http://en.wikipedia.org/wiki/HSL_and_HSV]

Demo: Lambert Demo TODO

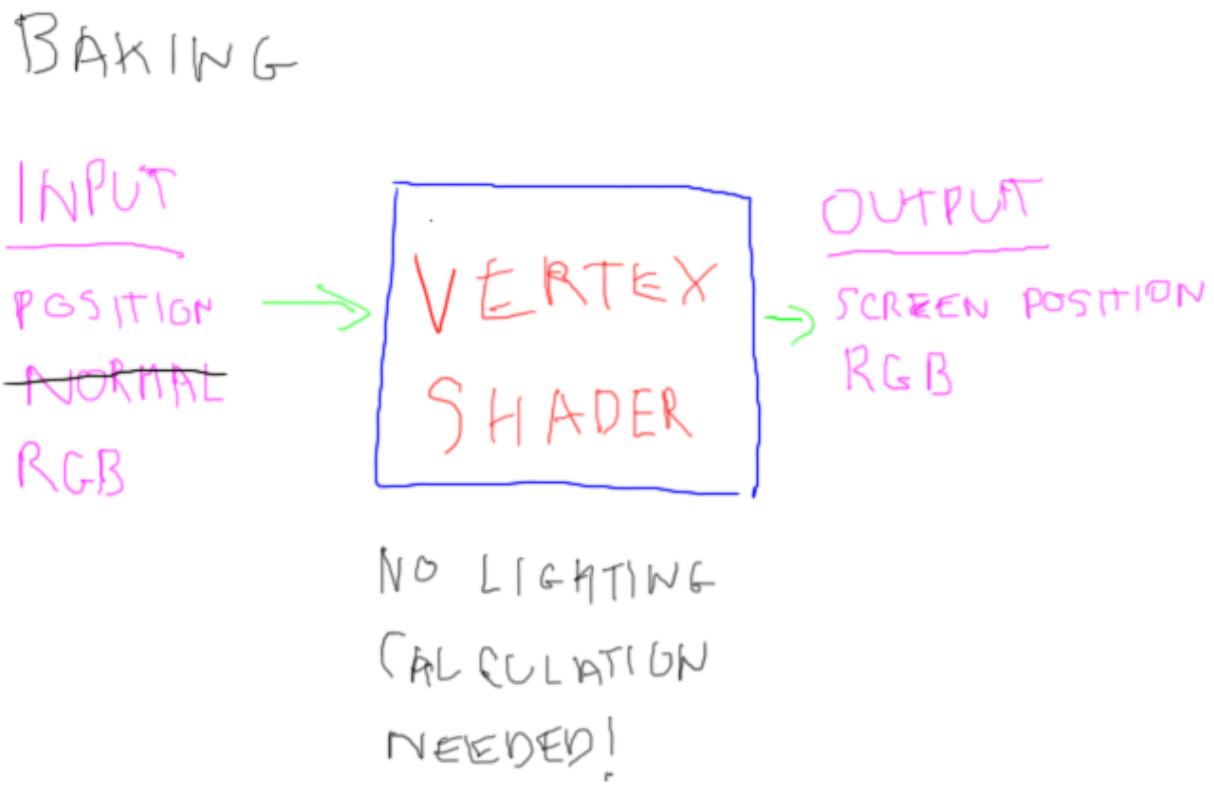
[TODO add to demos. Basically, smooth shading solution exercises/unit3_lambert_demo.html with HSL control]

Quiz: Baking

One method of saving on GPU computations is to bake the lighting into the surfaces themselves. Say I have a scene with a bunch of diffuse objects, each with its own material color and with surface normals stored at the vertices. If I could compute the shade of the object once and store this color instead of the normal, I could save on lighting calculations each frame. As before, assume every light is extremely far away, but that it can change direction, just as the sun's angle to the ground changes direction over time.

[show **XYZ Normal -> XYZ RGB**]

[In figure below, I added lights as an input - no longer needed]



Instead of storing the surface normal, under what exact conditions could I safely compute the lighting once and store an RGB color at each vertex?

[ ] If the object orientation does not change.
[Narration:] by “object orientation” I mean whether the object is rotating.
[ ] If the eye position and lights’ directions do not change from frame to frame.
[ ] If the lights’ directions and object orientation do not change.
[ ] If the eye position and object orientation do not change.

Answer

There are three elements that can change in the scene: the lights’ directions, the eye’s position, and the object’s orientation.

In this first answer, the object does not move. This means the object’s normals are the same each frame, but lights can still move. If the lights move, the shade of the object will change, so we can rule out this answer.

This second answer locks down the eye position and lights' directions. Now the light cannot move, but the object could rotate in space. If it does, the direction of its normals compared to each light will change, which changes the diffuse calculation. So, this answer does not work.

This third answer locks down the lights' directions and the object's orientation. The eye position is not locked down, so can move and change the direction it is looking. However, the two elements that affect the diffuse lighting equation are the lights' directions and the object's normals. Eye direction is unimportant, as a diffuse material has the same shade from any direction it's viewed. So, this answer is correct.

The fourth answer keeps the object's orientation fixed, but, like the first answer, the lights can change direction, so the surface's shading can change. So, this answer is incorrect.

Lesson: Specular Material

There are quite a few materials that are nearly diffuse reflectors, such as rough wood, newspaper, and concrete. However, a considerable number of surfaces are shiny or glossy. We call these specular materials. Examples include polished metals, plastics, polished wood, glass, ceramics, and enamel paint. These materials look different when you view them from different angles. So, we need to take into account the direction from the surface to the eye.

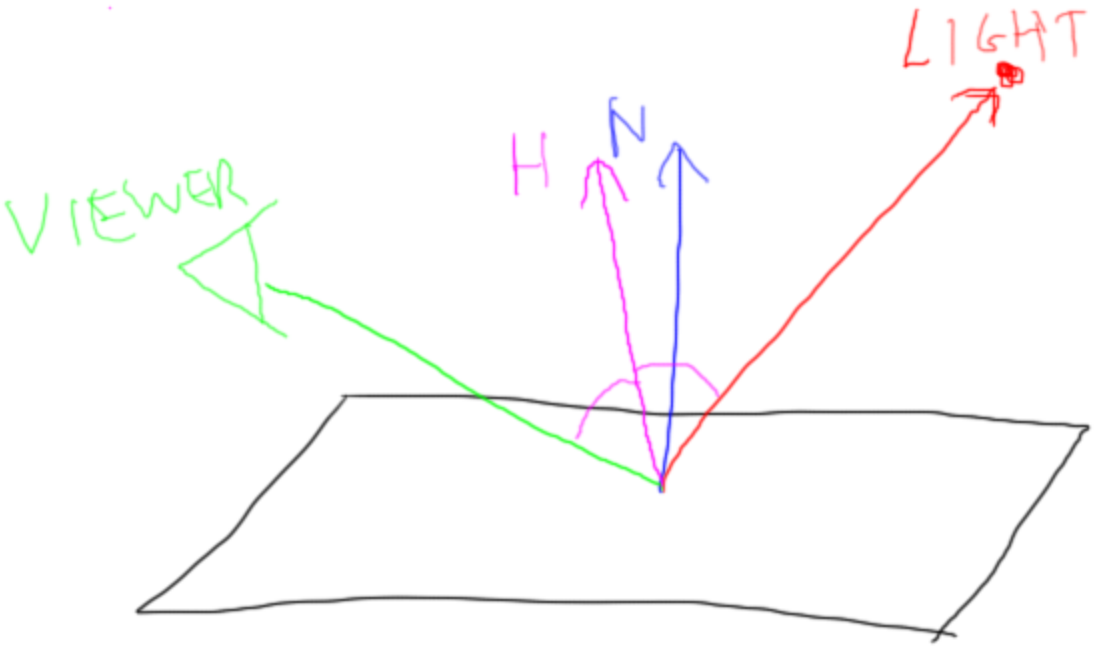
One standard way of simulating specular materials is called the Blinn-Phong reflection model, named after its inventors. The full model has a number of terms in it, for self-shadowing and for a shininess factor called the Fresnel coefficient, but the simplest and most common form is this:

Blinn-Phong reflection model

Specular = $max(N * H, 0)^S$

N is the surface normal, same as with the diffuse material. H is called the half-angle vector. Say you're given a light source direction and a viewer direction. How would you point a mirror so that the light reflected directly towards the viewer? The answer is the half-angle vector, which is the vector halfway between these two directions.

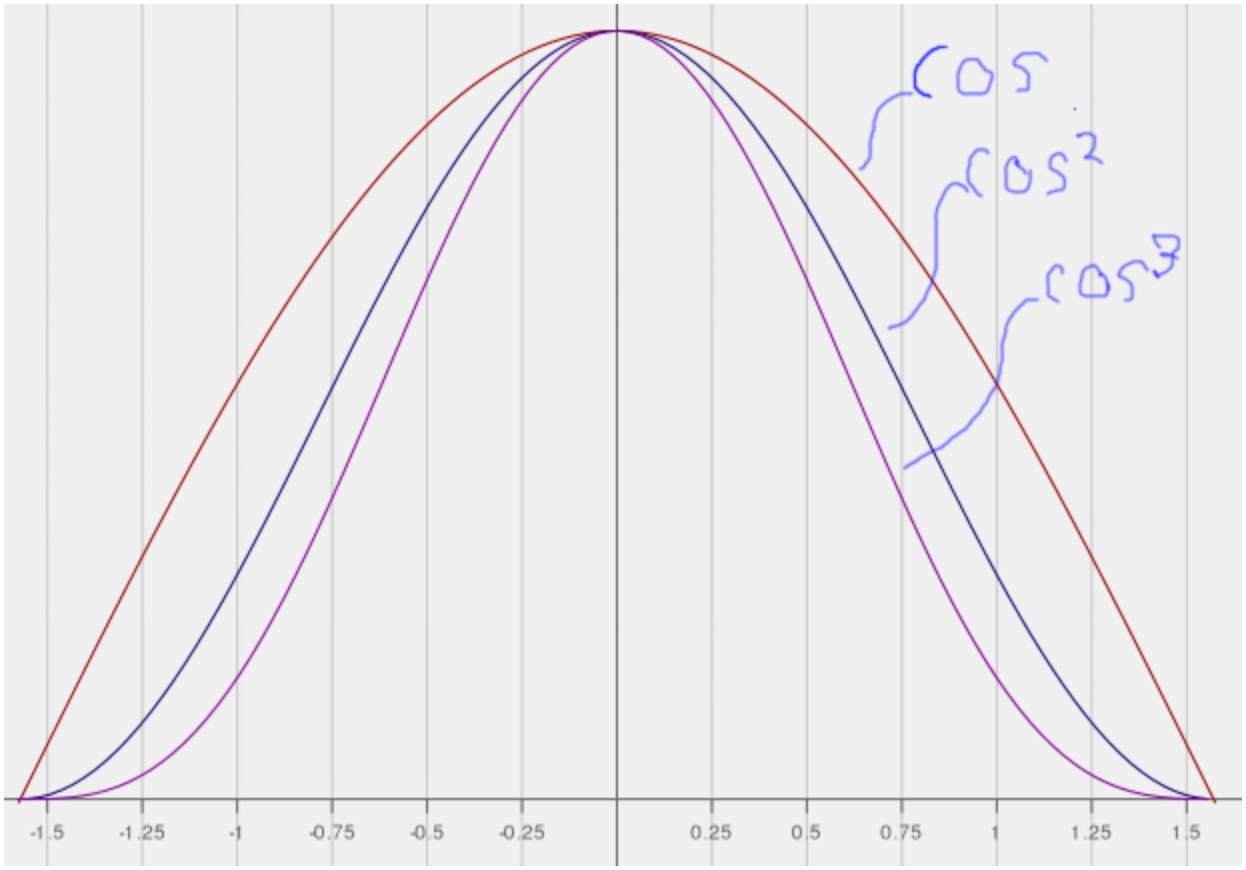
[draw half-angle - the formula above is still on the page]



If the surface normal and the half angle are identical, then the surface is perfectly aligned to reflect the light to the eye, so N * H is 1 and all light is reflected. As the normal and the half-angle diverge in direction, N * H, the cosine of the angle between them, becomes smaller. Once the angle between these vectors is 90 degrees, the contribution goes to 0. The Maximum function here [point to equation] limits the inputs so that this value is never negative.

The S factor is the shininess or specular power, and has a range of 1 to infinity, though anything above 100 is not much different. When you raise a fraction to a power, the result is smaller, and smaller still the higher the power. For example, 0.5 squared is 0.25, cubed is 0.125, and so on. By raising this term to a higher power, the object appears shinier. We can see this in a graph of N * H vs. the specular intensity:

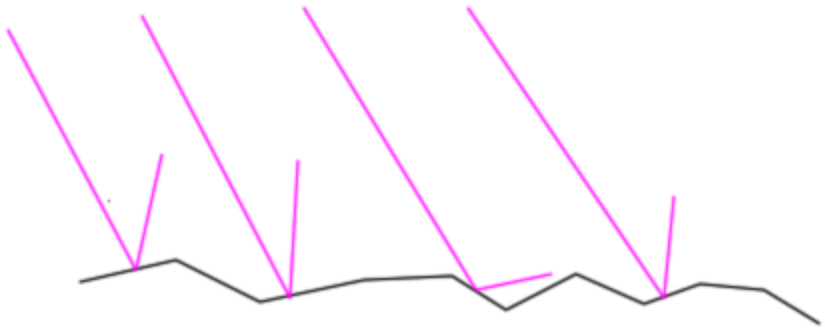
[graph, made with http://www.meta-calculator.com/online/ - not a great program, some mysteries with it, like I can't use degrees properly.]



What the half-angle represents is the distribution of microfacets on a surface. A microfacet is a way of thinking how a material reflections light. For example, a fairly smooth surface may look like this

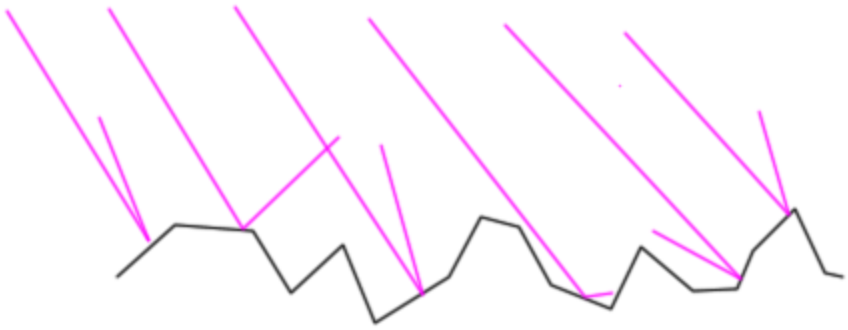
[microfacets smooth]

[Gundega: I tried drawing this in freehand, it looked terrible, so better to use the polyline tool here. In general I assume I'm to use the freehand mode, right? Any rules on this?]



Light coming in from one direction will bounce off the surface mostly in the reflection direction. A rougher surface, with a lower shininess, has a distribution of facets more like this

[microfacets rough]



and the light will still generally go in the reflection direction but with a wider dispersal.

At this point it's best for you to try out the specular power function and see how it responds. In the example program that follows, you control the ambient, diffuse, and specular contributions. Try playing with the shininess and other controls to see their effects.

[Instructor Comments: The demo program to try out is here: http://www.realtimerendering.com/udacity/?load=demo/unit3-tessellation-demo.js

Just about everything in real life turns out to be shiny to some extent, see this article http://filmicgames.com/archives/547 . Truly diffuse materials are difficult to come by; the material Spectralon http://en.wikipedia.org/wiki/Spectralon is made to act as much like a perfect diffuse as possible.

One model I won't cover in these lectures is the original Phong reflection model. It instead relies on the dot product of the light's reflection vector and the view direction. You can read about it in this article http://en.wikipedia.org/wiki/Phong_reflection_model, and how the Blinn-Phong model compares in this article http://en.wikipedia.org/wiki/Blinn%E2%80%93Phong_reflection_model.
]

Lesson: Gouraud Shading

[show image - leave room for terms below

You should have noticed in the demonstration program that the rendering quality was pretty bad, that you could see strong signs of the underlying grid. The problem is that we're computing the effect of specular shading at each vertex and interpolating across each triangle. This same problem can be seen with diffuse lighting, but is particularly obvious with specular highlights because the color changes more rapidly.

When we compute lighting at vertices and interpolate like this, it is called **Gouraud Shading**. Unfortunately, our eyes are very good at picking out places where the rate of shading changes. Our visual system will even perceived creases, which are called **Mach Bands**.

To avoid this problem we'd like to compute the lighting more frequently. One way to do this is to increase the number of triangles making up the surface.

[Instructor Comments: Mach Bands are described further here
http://en.wikipedia.org/wiki/Mach_bands]

Question: Increasing Tessellation Introduction

[Saved screen, but with word “Introduction” (?)]

There are pros and cons to increasing tessellation. Run the following demo and see how changing the tessellation changes the sphere. You can also toggle whether smooth shading is done. Afterwards think about the following questions:

[Leave some room for the figure as I’ll be adding in the answer!]

Tessellation
Rate
Low High
[ ] [ ] The amount of memory needed to store the object is lower
[ ] [ ] The lighting is evaluated more frequently
[ ] [ ] For pixels inside the sphere, the average number of polygons covering a pixel is lower
[ ] [ ] The silhouette of the sphere is smoother

[Instructor Comments: if you want to see the effect of the tessellation rate, try this demo program: http://www.realtimerendering.com/udacity/?load=demo/unit3-tessellation-demo.js . You can use the up and down arrow keys to change the tessellation rate, and ‘F’ to switch between flat and smooth shading.]

Demo: Sphere Tessellation

[Gundega’s demo]

Question: Increasing Tessellation

[Same screen as before, without the word “Introduction”. We do *NOT* want to change the title on this one, we want to use the screen I saved before, as then I don’t have to refilm the answer.]

Check the box for which object has a given advantage; check neither box if neither has an advantage.

Tessellation
Rate

Low High

[ ] [ ] The amount of memory needed to store the object is lower

[ ] [ ] The lighting is evaluated more frequently

[ ] [ ] For pixels inside the sphere, the average number of polygons covering a pixel is lower

[ ] [ ] The silhouette of the sphere is smoother

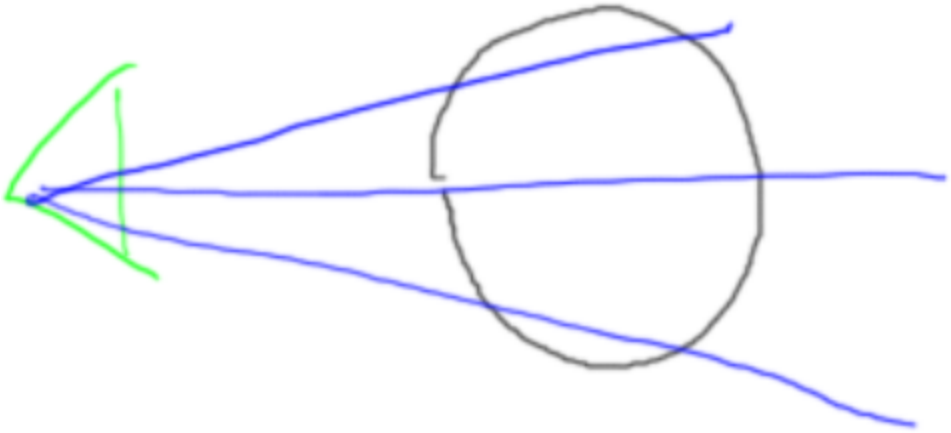
Answer

The less tessellation an object has, the fewer triangles are needed to represent it, so the amount of memory is lower to store it. So the first answer is “low”.

With more vertices for the same object, the lighting is indeed evaluated more frequently for the highly tessellated sphere.

For any given pixel inside either sphere, only two triangles will cover any pixel, one on the front side and one on the back.

[viewer and sphere]

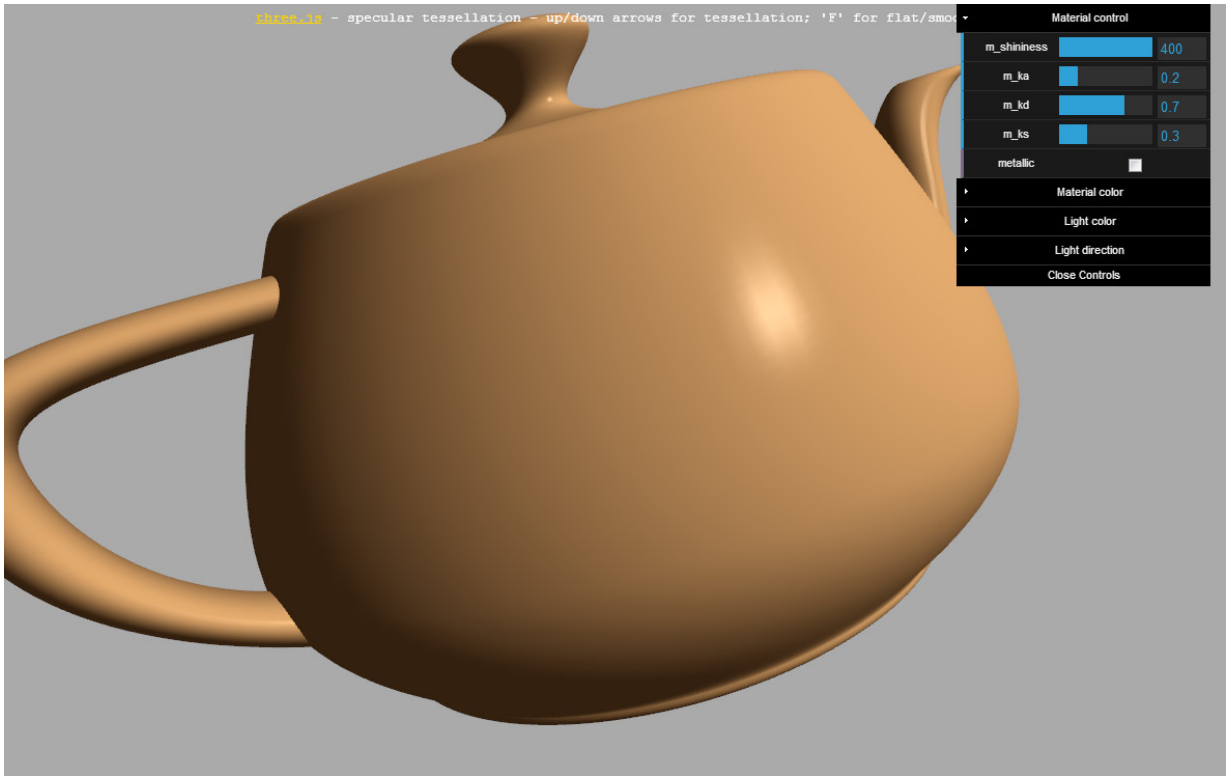


If backface culling is on, only one triangle will cover any pixel, one of those that is visible. So there is no change regardless of tessellation rate.
The silhouette edge of the object is smoother for the highly tessellated sphere, so “high” is the answer for the fourth question.

Lesson: Phong Shading

In the last demo you can see that increasing tessellation can make lighting computations look better.

[This should be a movie clip. Show higher resolution - still not great]



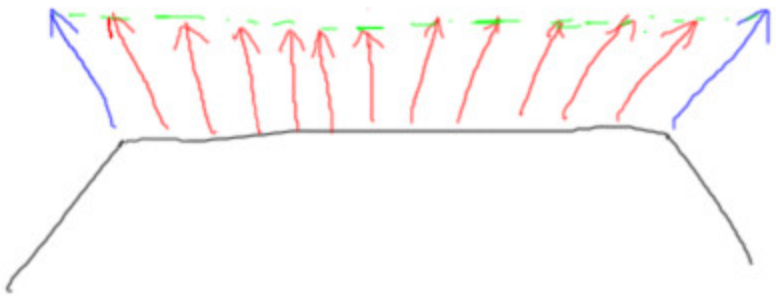
However, if you turn up the shininess, you can still pick out problems. Also, along the edges of the sphere the density of triangles is much higher than in the middle. If we keep increasing the sphere's tessellation, we'll be making a huge number of triangles along the fringe which cost us computation time without adding much to the final image.

What we'd ideally like is to evaluate the reflection model each pixel. This is an entirely fine solution! Instead of interpolating the RGB values that result from computing the effect of lighting, we interpolate anything we need to evaluate in the reflection model. In this case, all we need to interpolate is the shading normal.

[vertex shader new output]

What this means is that the vertex shader outputs the vertex normal along with its position. When interpolation occurs, all additional attributes, such as the normal, are interpolated across the surface of the triangle. This interpolated normal is input to the fragment shader, which then evaluates the reflection model and computes the RGB to be displayed.

[show interpolation of normals]

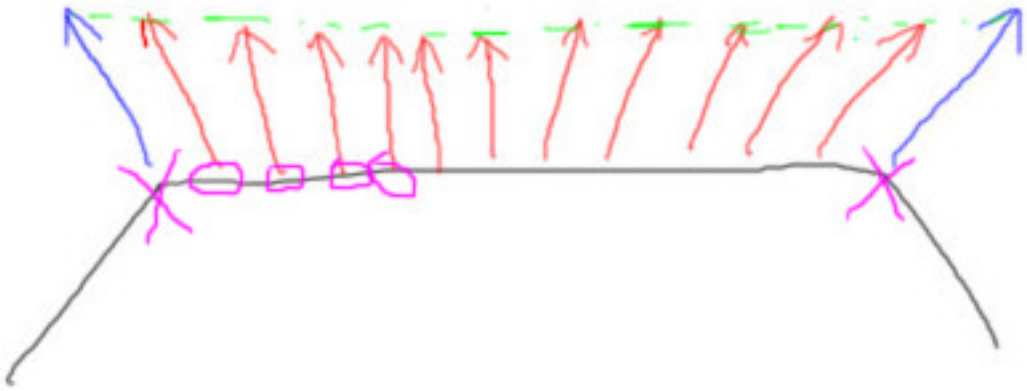


I should mention one important point with normal interpolation. The interpolated normals will not be normalized. As you can see in this drawing, the interpolated normals are always a bit shorter when interpolation occurs. So each normal should be normalized before using it in the lighting computation.

When we interpolate normals and evaluate the reflection model at each pixel, this is called **per-pixel shading** or **Phong Shading**. Confusingly, Phong invented two cool things. One was the Phong reflection model, which gives us a specular highlight, a shiny spot. The other is Phong shading, which is where we interpolate data across the triangle and compute the color per pixel. This is compared with Gouraud shading, where the color is computed at each vertex and interpolated.

[draw on figure above: point at bottom of each normal for phong shading, per pixel. X the

vertices when talking about Gouraud.]



Try running the new demo and see how Phong Shading vastly improves image quality. There is a cost, of course, in that the lighting calculations need to be performed much more frequently.

Demo: Phong vs. Gouraud

[Teapot demo from unit 1.]

Try the tessellation control, toggling Phong shading on and off.

Question: Material Calculations

Say it takes roughly the same amount of time to compute the lighting at a vertex as it does per pixel. You're trying to get some idea of how much more expensive it is to draw objects using material evaluation using fragment shading instead of vertex shading.

Say your object and application has the following characteristics:

[Draw a mesh of triangles, showing a vertex shared by 5 triangles. Show triangle mesh extending outwards.

- ***Each vertex in the mesh is shared by an average of 5 triangles.*** That is, any vertex where we compute the material's color can be shared among 5 triangles.

[Put 60 inside a triangle

- ***On average, each triangle covers 60 pixels.*** Don't worry about visibility, assume the pixel shader is evaluated for all of these.

What is the ratio of fragment vs. vertex shader computations: ____ to 1

Answer

The number of triangles doesn't matter, since the number of vertices lit and the number of pixels lit vary directly with the number of triangles. But, it's a bit confusing to think about how 5 triangles share vertices, at least for me.

Since our object can have any number of vertices, give it 100. 100 triangles times 3 vertices per triangle [write **100 triangles * 3 vertices/triangle = 300 vertices**] gives 300 vertices. We know that each vertex's material calculation will be shared by 5 triangles, so that gives 300 vertices ***/ 5 triangles per material calculation = 60 material calculations***.

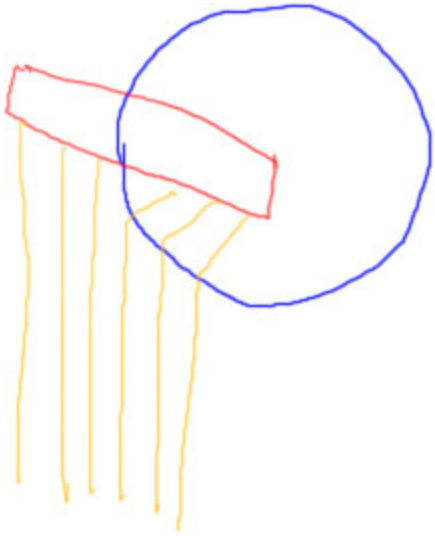
For the fragment shader there is no sharing, so ***100 triangles times 60 material calculations per triangle*** means there are ***6000 material calculations***.

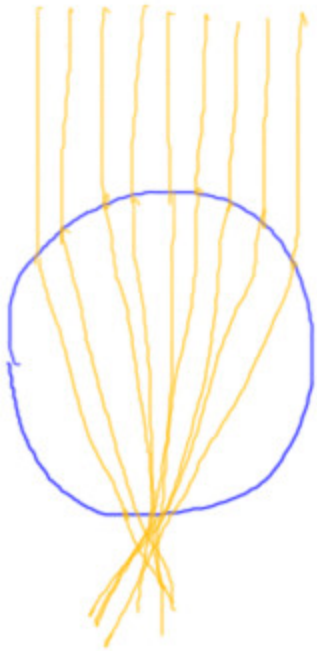
6000 material calculations / 60 material calculations = 100, so the answer is 100 to 1.

As you can see, it can be considerably more costly to evaluate per pixel than per vertex. Note that this doesn't mean that the program itself runs 100 times slower. There are mitigating factors, and other parts of the pipeline can be the bottleneck, which changes this ratio in practice.

Lesson: Introduction to Transparency

[***Refraction: direction of light changes***]





[***Refraction*** ***Caustics***]

Up to this point we've been working with opaque materials, where light hits the surface and bounces off. For a number of reasons, transparent objects are more challenging. First, light refracts and so changes its direction when it hits a transparent surface. For example, here light comes from the surface of the pencil, travels through water and hits the glass. The glass causes the light to go a different direction, such that different light rays reach our eyes and cause the pencil to look bent.

This change in direction can also affect the distribution of light on a surface. This crystal ball acts similar to a lens, focusing the light and creating various bright spots below. In computer graphics

we call these bright spots ***caustics***.

The color and intensity of the light can be filtered by the transparent object. The farther a ray of light travels through an object, the more light that can be absorbed. Even thin filters can change the color of the light.

All of these effects can be simulated, or at least approximated in a convincing way, at interactive rates. Refraction, where the light changes direction, is expensive to do correctly. I'll be discussing it further in later lessons. It's also possible to compute the amount of light absorbed due to traveling through an object, though this can be costly. In the following lessons I'm going to focus on absorption of light by thin transparent surfaces, such as these eyeglass filters.

To simplify further still, I'll be talking about only the effect of such filters on objects visible through them. In other words, I won't be talking about how light passes through the filter, reaches another surface, and then traveling to the eye. [cross out path we're ignoring]

[I have contacted the author, Aleksandar Rodic, and asked for permission to show the jellyfish demo in the lesson itself. He's granted it! Run http://chrysaora.com/]

This all sounds like a lot of simplification, but you can still get wonderful effects from just this form of filtering alone. For example, this fantastic demo make by Aleksandar Rodic shows how transparency can be used to great effect. There are also other cool effects going on here, such as the “god rays” of light. It's one of my favorite WebGL programs.

Talking with Aleksandar, he first encountered the transparency sorting problem when he created this demo. He had to do a bit of hacking to rapidly sort and display the objects by depth. Look in the additional course materials for more about this demo and try it out yourself.

[Additional Course Material: The jellyfish demo is at http://chrysaora.com/. Another good use of transparency is the Zygote Body demo http://www.zygotebody.com/. There is a vertical slider on the left side of the screen - zoom in and move it up and down slowly.

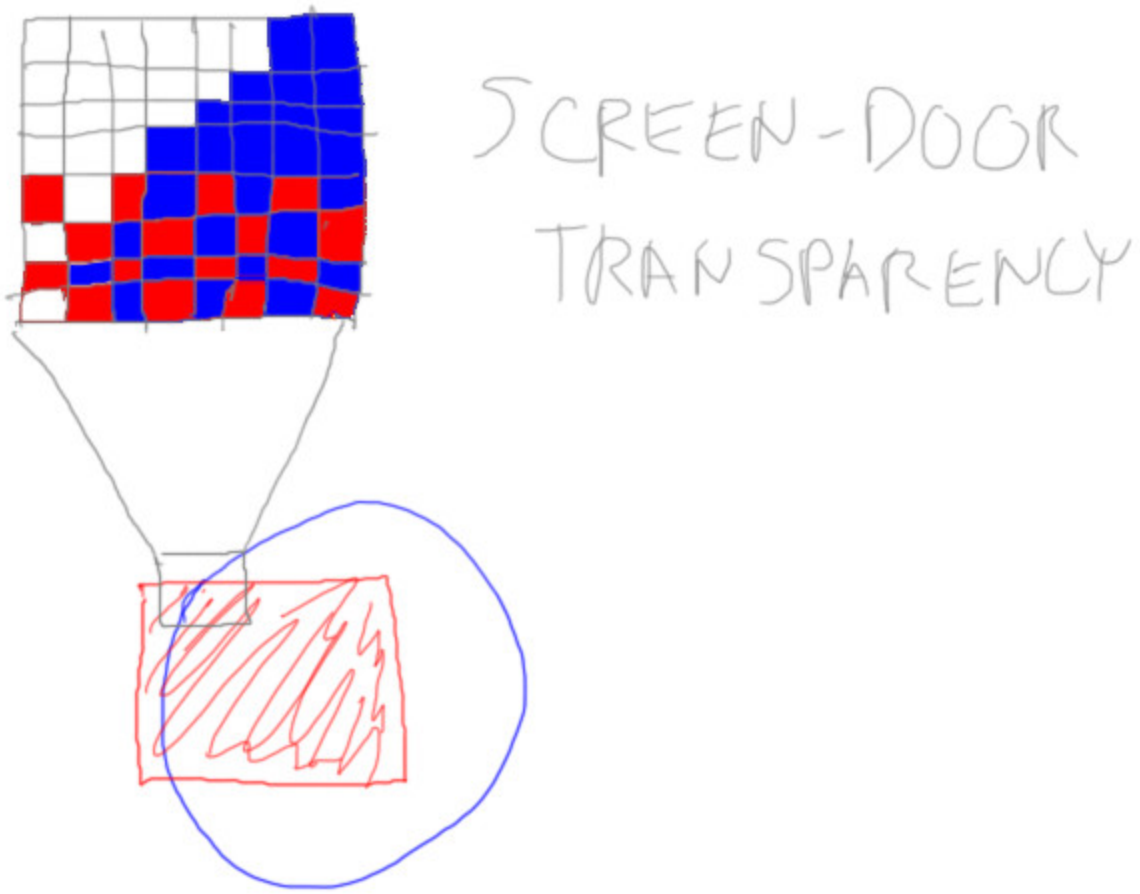
From Rodic, folded into last lesson:

There is another version of the demo that is way more optimized:
http://aleksandarrodic.com/p/jellyfish/
I removed the particles and shadow rays that caused a lot of overdraw. Also, Skinning weights are pre-computed and stored in vertex attributes.

The transparency was one of the biggest challenges I had with the demo. Actually, that was the first time I got introduced with transparency sorting problem. What I did was a dirty hack. I pre-sorted all triangles along Z axis, then I twisted the skeleton so each joint looks at the camera facing Z :) Here is a quick video showing the modeling process.
https://vimeo.com/20234863

The purpose of the whole demo was a video installation in Hong Kong
https://vimeo.com/20875622
The challenge there was to sync the simulation across 4 computers and make it look seamless. I did that by running first stage of the sim on node.js and streaming that to browsers. Here is a demo of that
https://vimeo.com/20599736
]

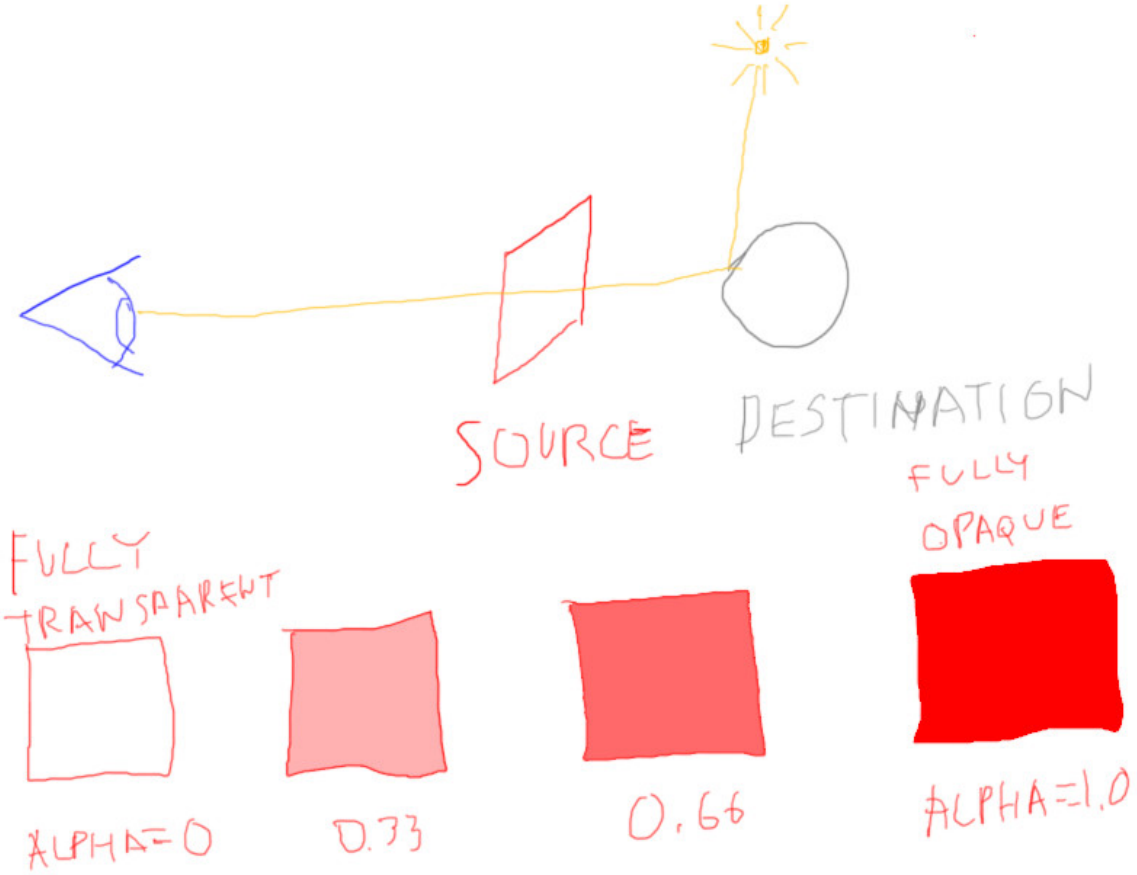
Lesson: Blending



One simple way to think of transparency is that you want to see a bit of both objects. If there's a

red filter in front of a blue object, you could color in every other pixel with red, leaving the rest blue. Pixels are tiny, so the eye will blend the two colors together to get a magenta where the objects overlap. This is a very limited form of transparency, however: it supports only two objects, and only a 50/50 mix looks good.

[**FIX**: Make the destination object blue, so I can use blue in the equations]



What would be more general is to have the transparent filter's color affect the color behind it. In other words, we want to blend the two colors. In blending in computer graphics, the original color is called the "destination", and the color of the transparent object we're looking through is called the "source color".

We'd also like to control how much can be seen through this filter. In computer graphics this control value is called "alpha". Here it represents our filter's opacity. When alpha is 1, the filter is fully opaque and does not let any light through. As this value decreases, more of the destination color becomes visible. When alpha is 0, our filter is entirely transparent and has no effect.

[Note: use alpha, not "A"]

C* = *αs Cs* + *(1 - αs) * Cd ***over*** *operator*

A typical way to blend these colors is by using this equation. Here the color of the source, of our filter, is multiplied by the source's alpha. This gives how much the source will affect the final color. The larger alpha is, the more that the source color affects the final color. To counterbalance this, this same alpha is subtracted from one, and this value determines how much the destination color influences the result. This type of blend is call an "over operator", as it gives the effect of putting one color over another.

[replace alpha with 0, then erase terms from original equation]

$$C = \alpha_s C_s + (1 - \alpha_s) * C_d$$

$$C = C_d \quad \text{when } \alpha_s = 0$$

The alpha of the source, our transparent filter, can be any value from 0 to 1. If the value is 0, this equation becomes simply "C is equal to the destination color", since the source filter is entirely transparent.

[show original equation again, replace alpha with 1, then erase terms from eqn]

$$C = \alpha_s C_s + (1 - \alpha_s) * C_d$$

$$C = C_s \quad \text{when } \alpha_s = 1$$

If the value of alpha is 1, the source is fully opaque and hides whatever it's being placed over. The equation simplifies as expected, with the final color being the same as the source color.

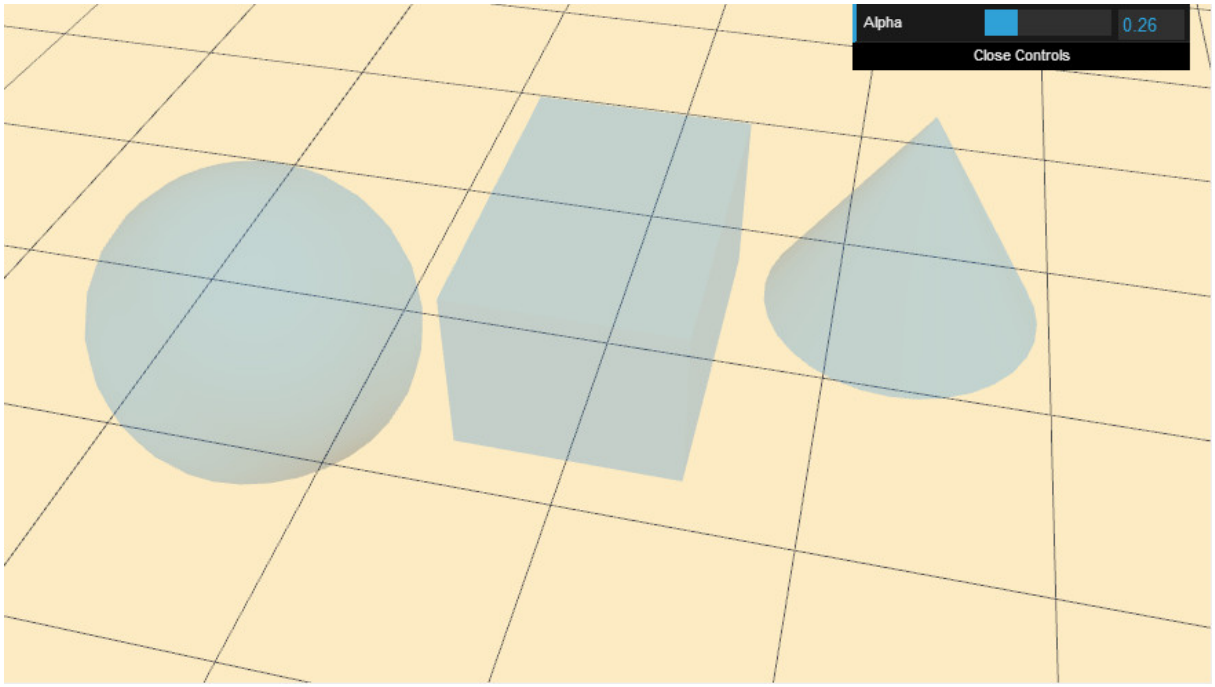
[restore eqn again]

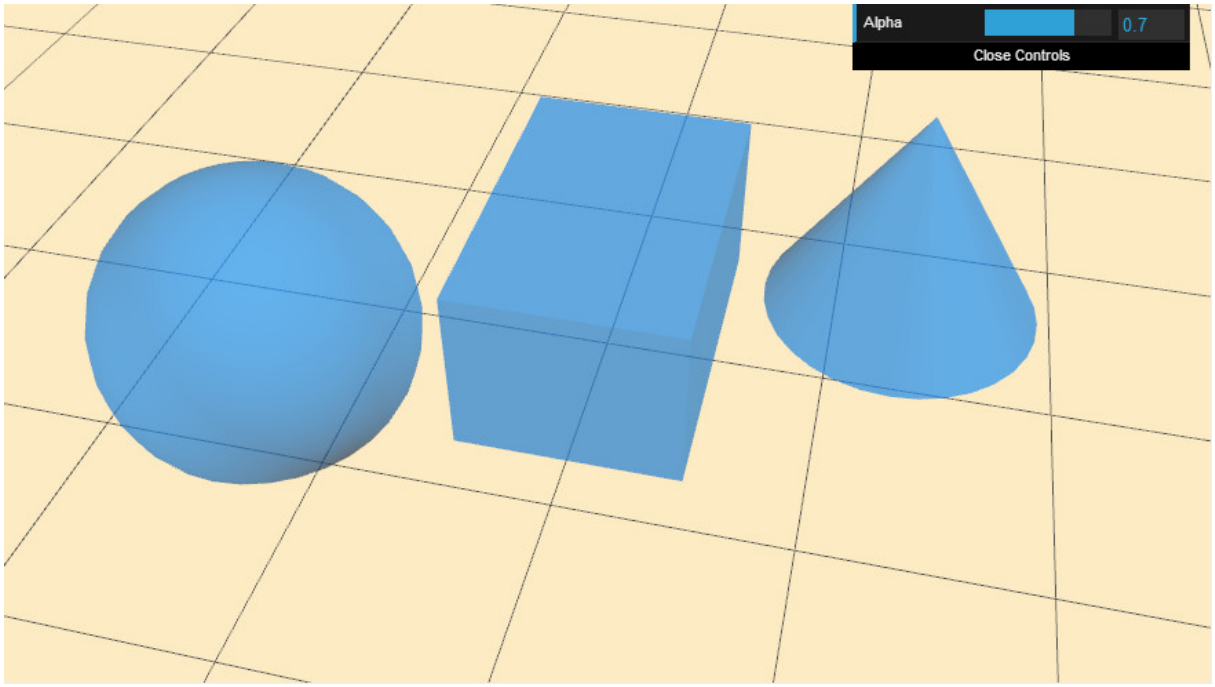
$$C = \alpha_s C_s + (1 - \alpha_s) * C_d$$

linear interpolation

The alpha value blends between these two colors. Another way to say it is that the equation performs linear interpolation. As one color's influence increases, the other color's effect drops off in proportion. This idea of having a control value, in this case alpha, that varies between 0 and 1, is commonly used in many areas of computer graphics. Expect to see this sort of equation elsewhere as you explore the field.

[demo at http://www.realtimerendering.com/udacity/?load=demo/unit3-blending.js]





The “over” operator is useful for transparency. There are many other blend modes available in WebGL. For example, one blend mode is called “add”, as it adds the source and destination colors together. In three.js the blend mode used is specified with the material’s blending parameter.

Here we see the “over” equation in action. By varying the alpha we fade the effect of the

transparent object on and off. Try it yourself.

[Additional course materials: A blending demo that shows how the various different blend modes interact are here: http://voxelent.com/html/beginners-guide/1727_06/ch6_Blending.html from the book “WebGL Beginner’s Guide” http://www.amazon.com/WebGL-Beginners-Guide-Diego-Cantor/dp/184969172X. Here is a demo in three.js that examines many of the possible combinations http://mrdoob.github.com/three.js/examples/webgl_materials_blending_custom.html - it’s a bit cryptic, but you can modify things with the controls on the left.]

Demo

[http://www.realtimerendering.com/udacity/?load=demo/unit3-blending.js]

Question: The Over Operator

You have a reddish filter, a color of 0.9, 0.2, 0.1 red green blue, and a yellowish background, a color of 0.9, 0.9, 0.4. You set the filter to have an alpha of 0.7, fairly opaque. What is the resulting color when you use the over operator?

Filter color 0.9, 0.2, 0.1
Background color 0.9, 0.9, 0.4
Filter alpha is 0.7

Resulting color is _____, _____, _____

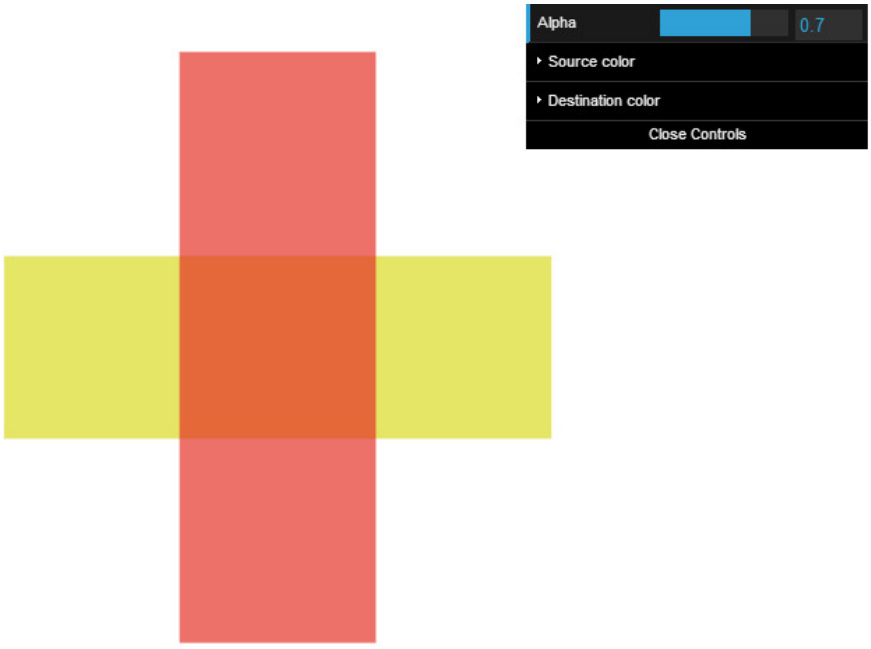
Answer

C = 0.7 * (0.9, 0.2, 0.1) + (1 - 0.7) * (0.9, 0.9, 0.4)
C = (0.63, 0.14, 0.07) + (0.27, 0.27, 0.12)
C = (0.90, 0.41, 0.19)

Substituting the filter color into the over operator’s source, the alpha of 0.7 for the source alpha, and the background color into the destination color, we get this equation. Evaluating it gives a reddish-orange color.

The demo that follows shows the result of this particular exercise. Trying varying the alpha and the colors to see the effect.

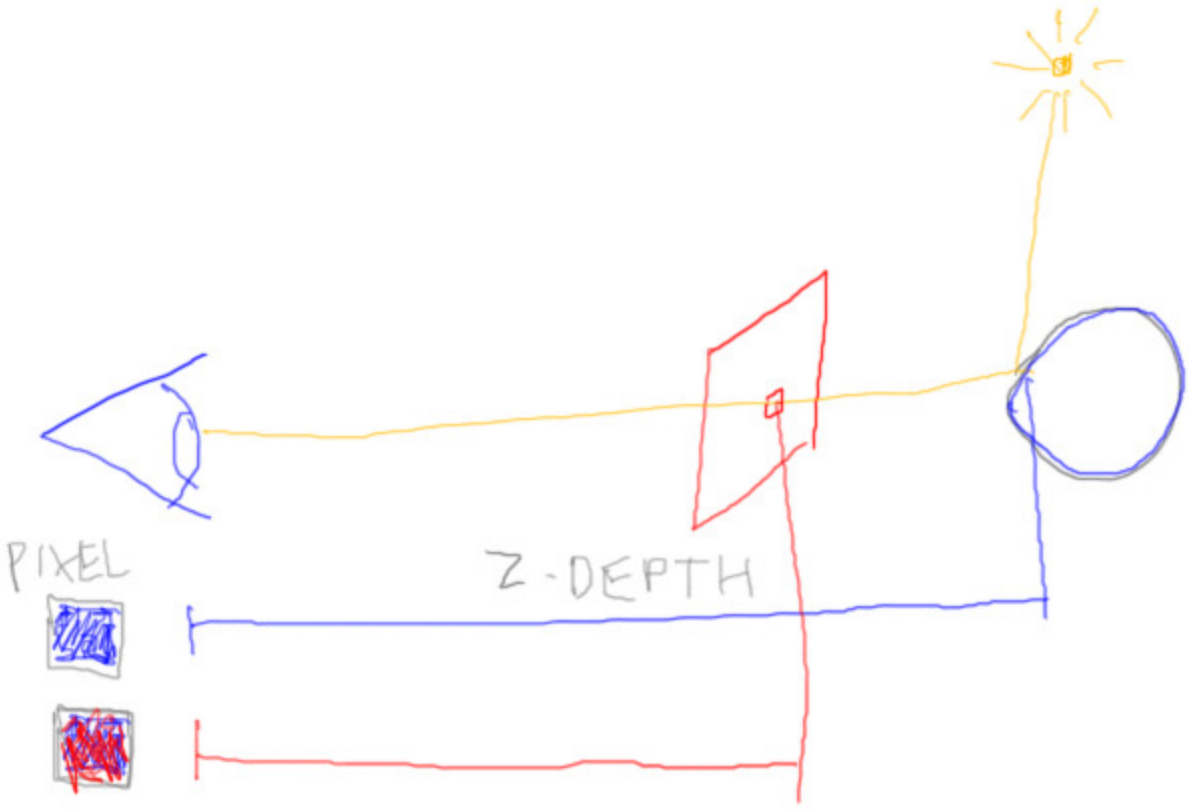
[show two swatches, red over yellow and the result, 0xE53319 over 0xE5E566]



Demo: The Over Operator

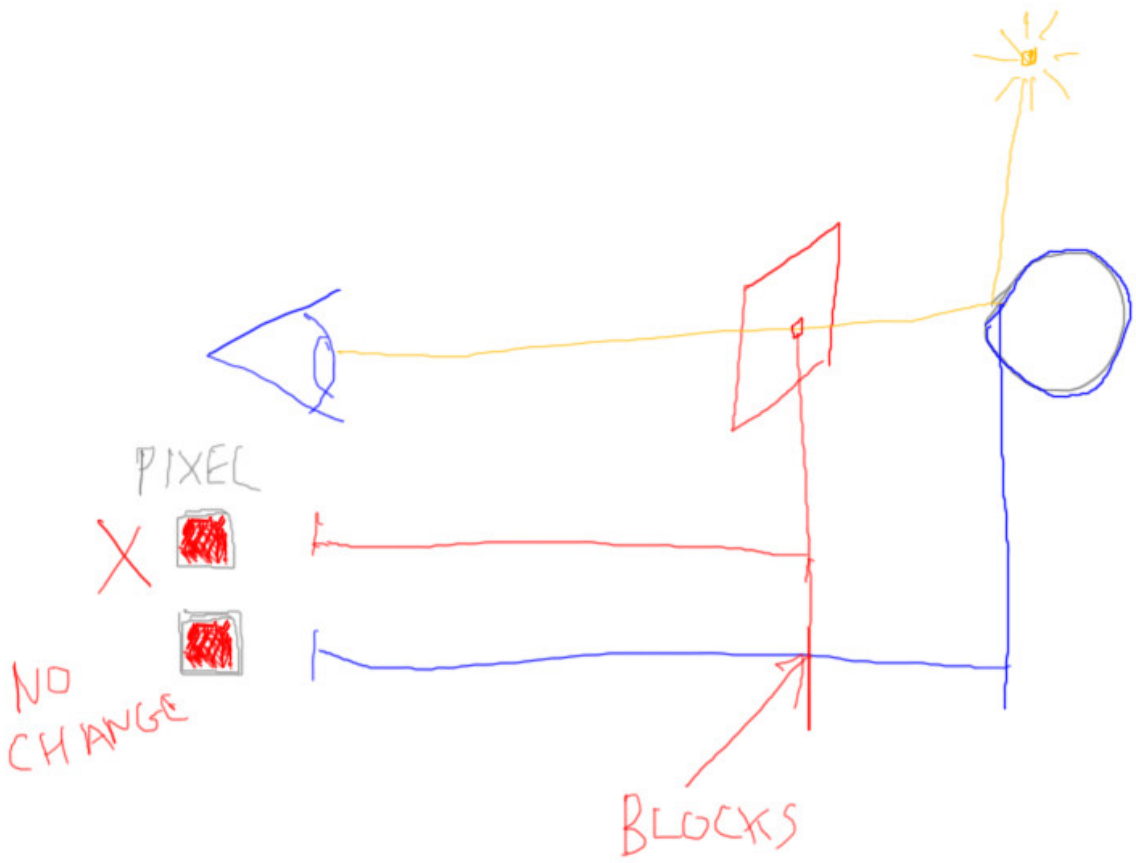
[The demo to vary alpha and the colors is here
http://www.realtimerendering.com/udacity/?load=demo/unit3-over_operator.js]

Lesson: The Z-Buffer and Transparency



The over operator shows how we can get a transparent effect by blending two objects. However, there's one little catch: how does the Z-buffer work when using transparent objects?

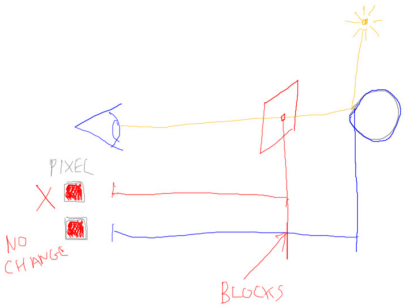
Think about what the z-buffer is doing. It stores the depth of the object that is closest to the eye. Say we draw the blue, far object first, then draw and blend in the red transparent object. This works fine: the blue object's color is taken and blended with the red fragment that is drawn.



However, if we reverse the order, we start to get into problems. We get the red fragment and want to blend it. But the blue object hasn't been put in the z-buffer yet! Even if we decide to try to wait for the blue object, we get into trouble. When the blue fragment renders, it is considered hidden by the red filter, since it's farther away from the eye.

Question: Solving Transparent Z-Buffering, part 1

[Shrink, to give room]



Here are some solutions to the problem just outlined, that drawing the red transparent object first causes the blue object to not be seen by the camera and blended in. Which of these solutions will solve this particular case? More than one answer can be correct.

[ ] Draw all objects sorted by size, smallest to largest
[ ] Draw all objects sorted back to front order
[ ] Draw all objects as if they're all transparent, blending each one
[ ] Draw the fully opaque objects first, transparent later

Answer

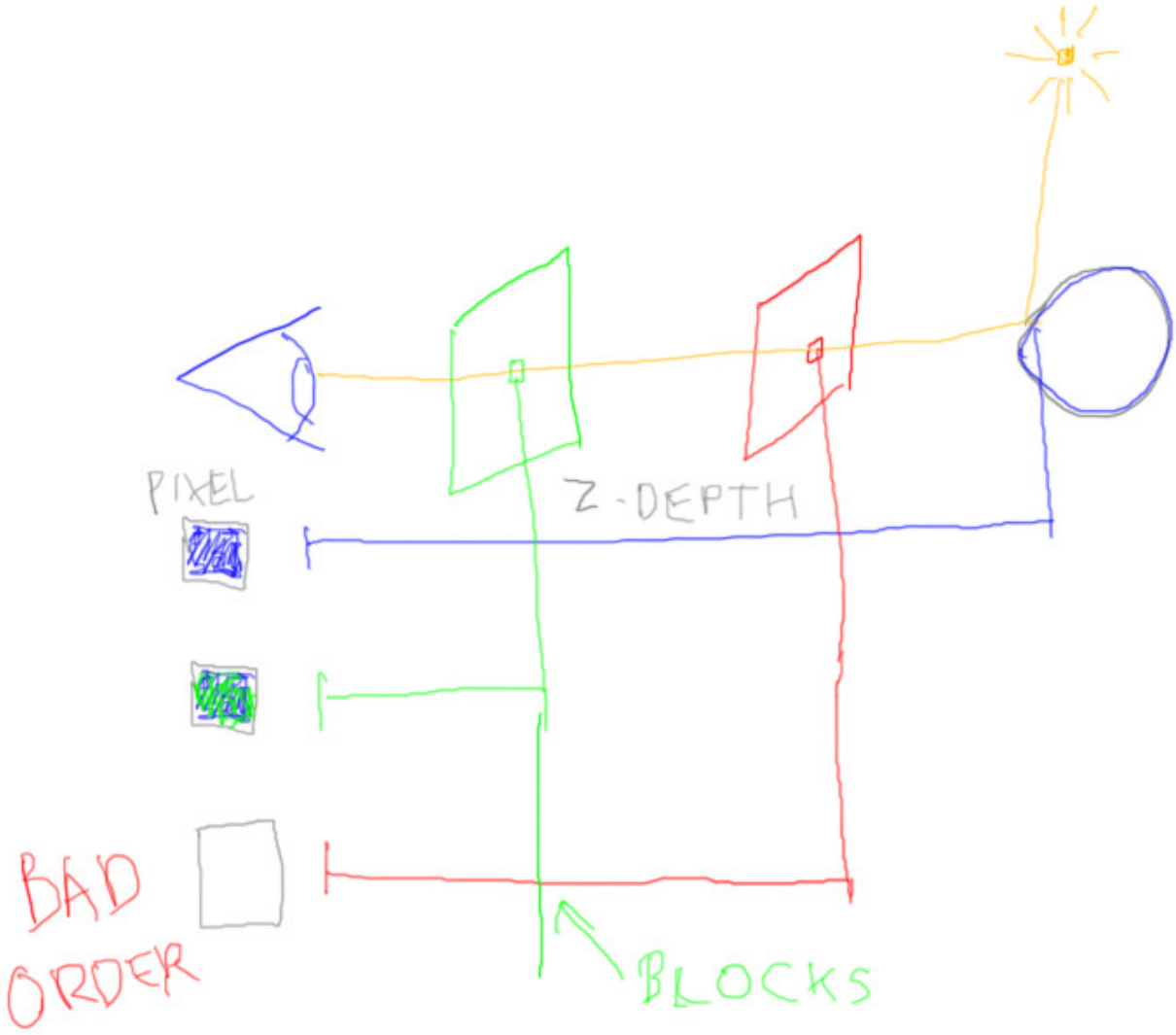
This first answer doesn't really help. Size doesn't matter, it's transparency that is causing the problem.

This second answer works. The blue object is farther from the eye than the red one, so will be drawn first. However, for more complicated scenes it's a bit of overkill and inefficient to boot. The sorting order of a set of opaque objects doesn't affect the final result on the screen - that's the Z-buffer's job. Drawing opaque objects in a back to front order is not efficient, as fragments will be processed by the fragment shader, then covered up by the next closest object.

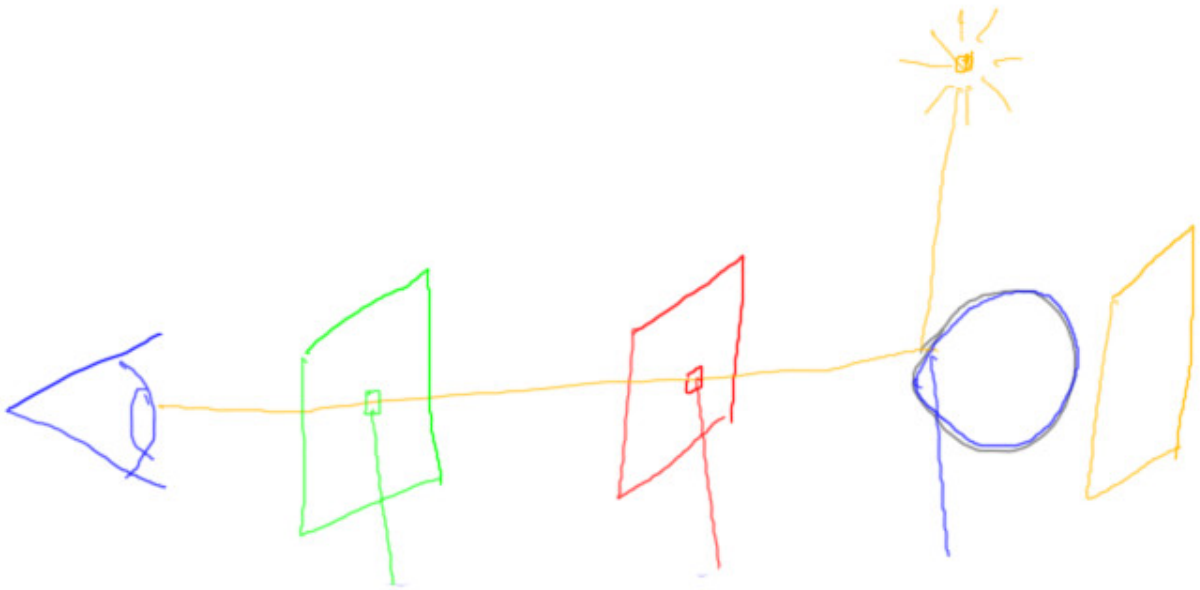
This third answer is not particularly meaningful and won't solve anything. The blue object essentially already has an alpha. It's equal to 1, meaning fully opaque. Even with blending on, an opaque object will still act like an opaque object. This solution does not fix the problem that the blue object is not being rendered early enough.

The fourth answer works for this case, where there is only one transparent object. First all opaque objects are drawn. It is then guaranteed that the transparent object will be able to blend with these opaque objects as needed.

Question: Solving Transparent Z-Buffering, part 2



Say we now add a transparent green object to our scene. To avoid our original rendering problem, the rule is that all opaque objects are drawn first, so the blue object is rendered first. If we now render the green object and the red object, in that order, the green object will block the red one. We want the blue object to be affected by both the green and red filters, but the red filter is rendered too late and has no effect.



Just to make life exciting, say we add yet another filter, a yellow one behind the blue object. This filter is not visible for the pixel shown, but whatever solution we choose must make sure this object is taken into account properly.

Which is the best solution, the one that will give us the result of red blended with blue, and then green blended over that?

Draw all the transparent objects...

() ...sorted by their alpha values, largest to smallest.
() ...using the Painter's algorithm, with z-buffer testing off.
() ...sorted in back to front order.
() ...based on intensity, with brightest objects drawn first.

Answer

Ideally we want to draw the blue object first, then the red, then the green. The yellow filter is behind the blue object and so should have no effect.

For this first answer, since we aren't given the alpha values of the green and red filters, we don't know how this affects our draw order. The red or the green object may draw first. We can also rule out this last answer for the same reason. We don't know the intensity, or even how intensity is defined, so can't use this as a way to solve the problem.

This second answer works well for the green and red filter, but falls apart on the yellow filter.

With the z-buffer off, the yellow filter would be drawn on top of the blue object, filtering it. This is clearly incorrect.

This leaves the third answer. This answer is similar to the second answer, in that the Painter's Algorithm also sorts objects from back to front. However, this time, with the z-buffer on, the yellow filter is properly hidden by the blue object. The red filter is rendered next and blended, then the green is added in on top.

Lesson: Transparency and Three.js

Transparency

- ***Render all opaque objects first, with the Z-Buffer on.***
- ***Turn on blending.***
- ***Render transparent objects sorted back to front.***

We've derived one system of performing transparency that mostly works. To sum up, all the opaque objects are rendered first, with z-buffering on. Blending is then turned on for the transparent objects that follow. Blending takes extra time for the GPU to compute, so it is turned on only when needed.

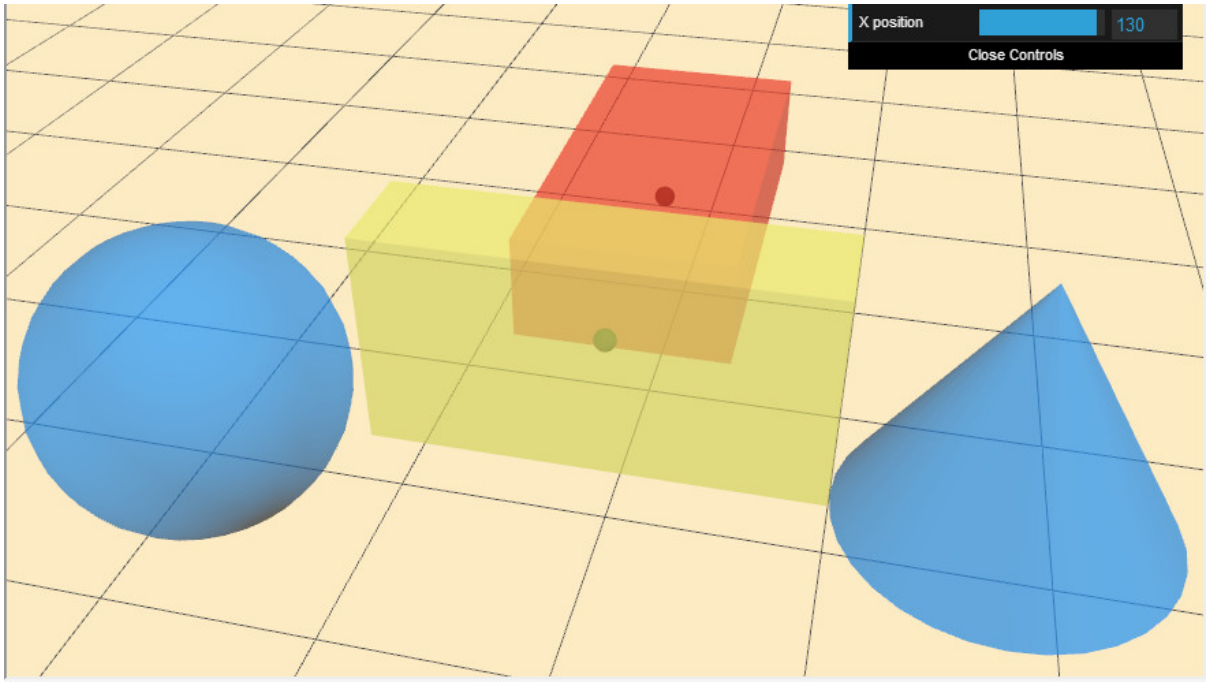
Finally, the transparent objects are sorted by their distance along the view and are rendered in back to front order. If the camera or objects are moving, this sorting has to be done every frame.

```
var movingBoxMaterial = new THREE.MeshLambertMaterial(
    { color: 0xE53319, opacity: 0.7, transparent: true } );
```

This algorithm is in fact what three.js implements. You make an object transparent by setting its material. There are two parameters to set. The opacity is the alpha value used for blending. You must also set the "transparent" parameter to be true.

This transparency scheme usually works fairly well in practice, but there are still quite a few problem cases. One problem is interpenetration.

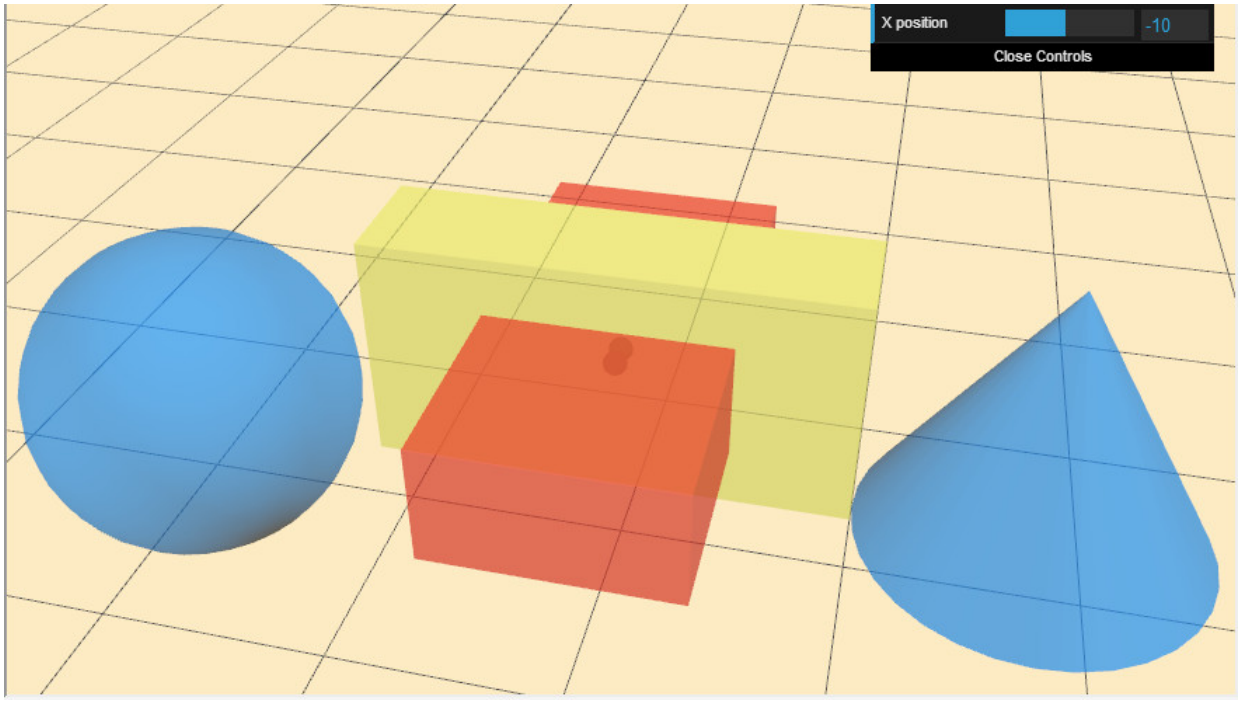
[http://www.realtimerendering.com/udacity/?load=demo/unit3-transparency.js]



Here's a demo showing showing one transparent block moving through the other. Give this demo a try, move the camera around and use the slider to position the block. The dot in the center of each object is what is used to measure its distance along the direction the camera is viewing. Once you're done with the demo, you'll then answer a question as to what's going wrong.

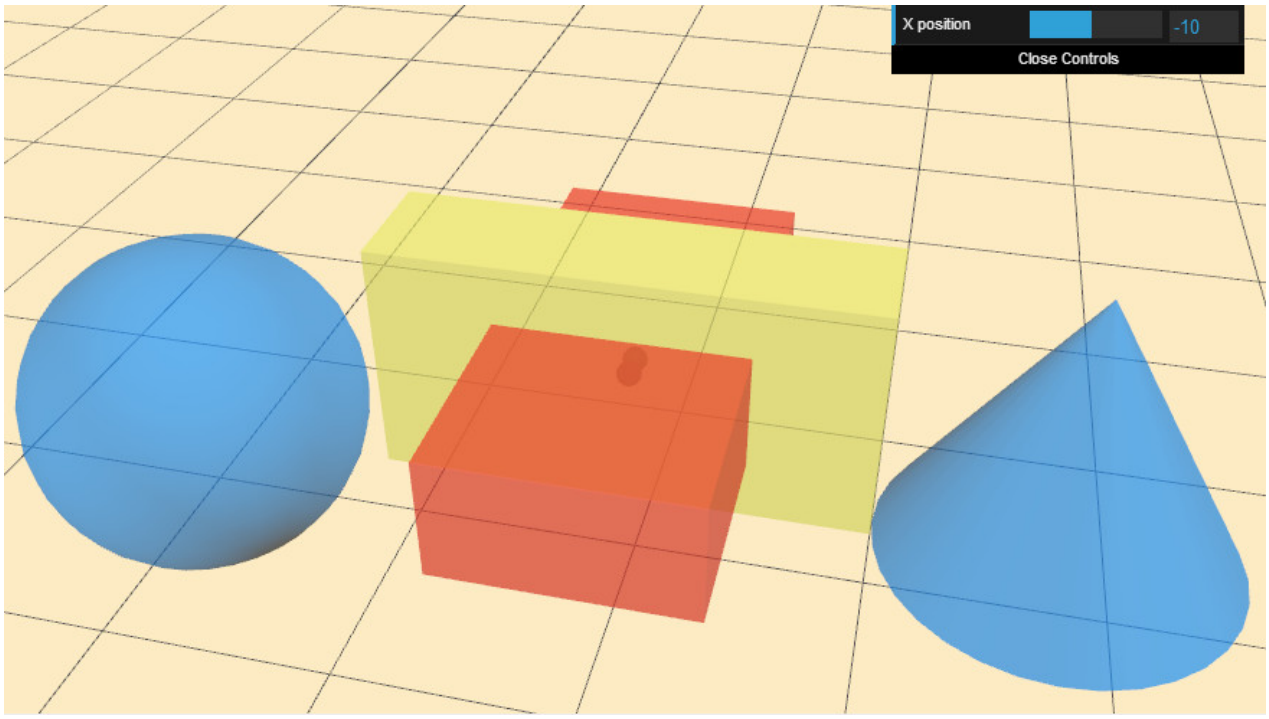
[Additional Course Materials: See Lee Stemkoski's tutorial code http://stemkoski.github.com/Three.js/Translucence.html for more examples of how to set transparent materials.]

Demo



[Find at: http://www.realtimerendering.com/udacity/?load=demo/unit3-transparency.js]

Question: What Went Wrong?

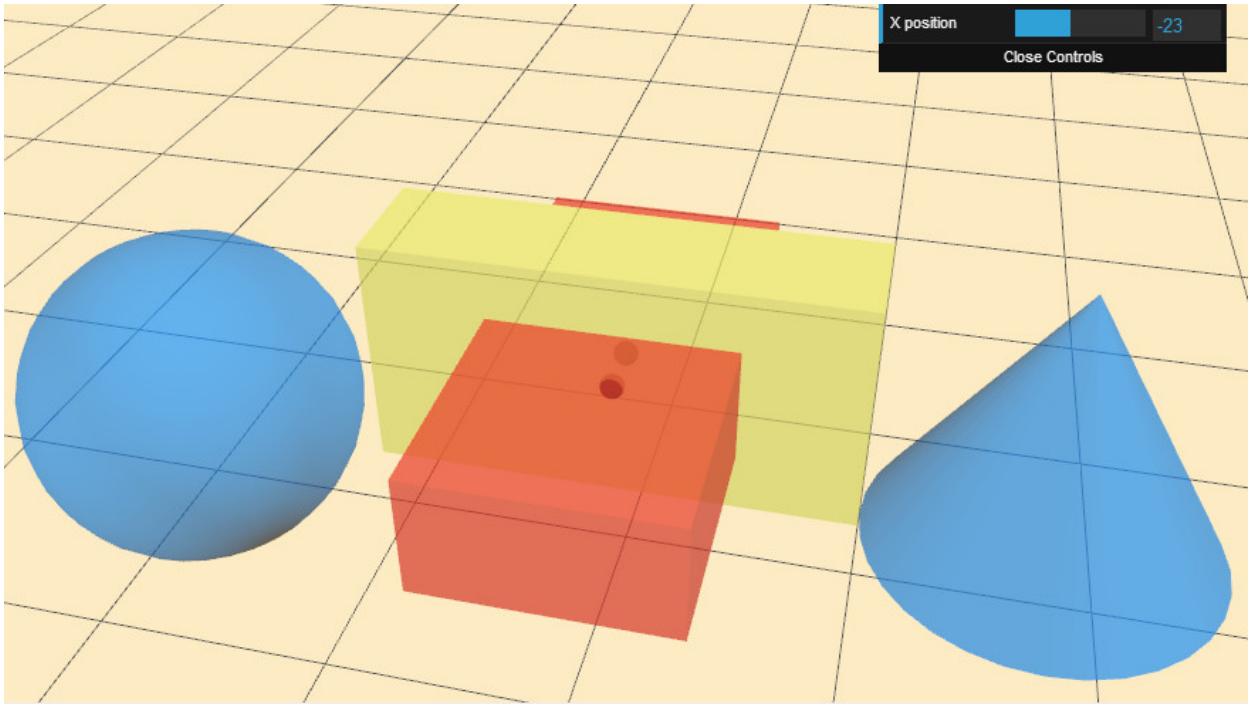


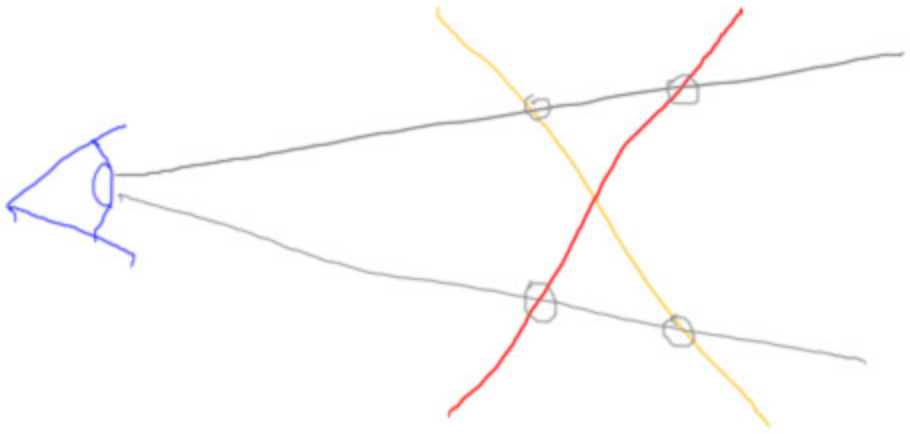
Looking at the demo, you should have noticed that when you pull the red box to be close to you,

suddenly you can't see it through the yellow object. You can see the ground plane grid through the yellow object, but the red object has disappeared.

Which of these is true for the picture? More than one answer can be correct.
[ ] The red object is rendered after the yellow object.
[ ] Blending has been turned off for the yellow object.
[ ] The yellow object was not drawn to the z-buffer.
[ ] The red object is closer to the eye than the yellow object.

Answer





Say you look at just two polygons from the side, a red one and a yellow one. Because they interpenetrate, the draw order will always be wrong for one pixel or another. If the yellow object is drawn first, this upper pixel will have the wrong order of yellow, then red. If the red object is drawn first, this lower pixel then has the wrong order of red, then yellow.

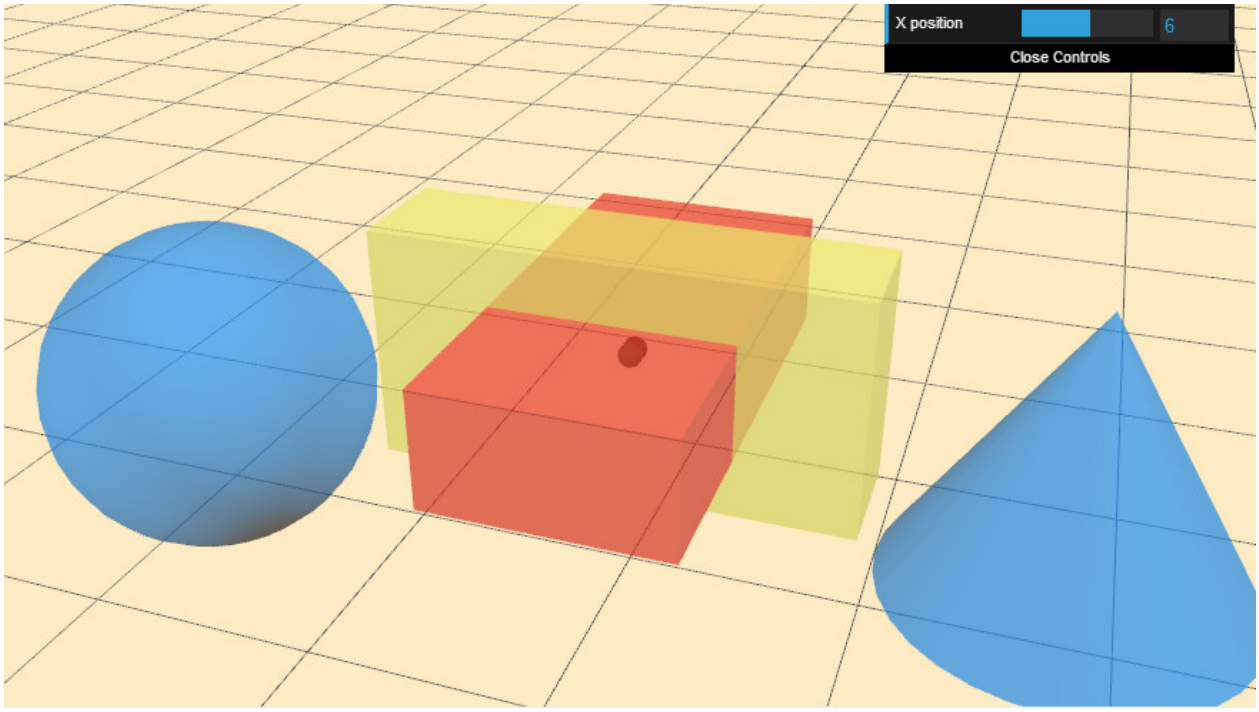
For the first answer, with the yellow object being rendered first, there's no red object for it to blend with. This answer is correct.

Blending has not been turned off for the yellow object, since you can see the grid lines through it. This answer is wrong.

The yellow object blocks part of the red object, at the top, so it must have been drawn first and written to the z-buffer. This answer is also wrong.

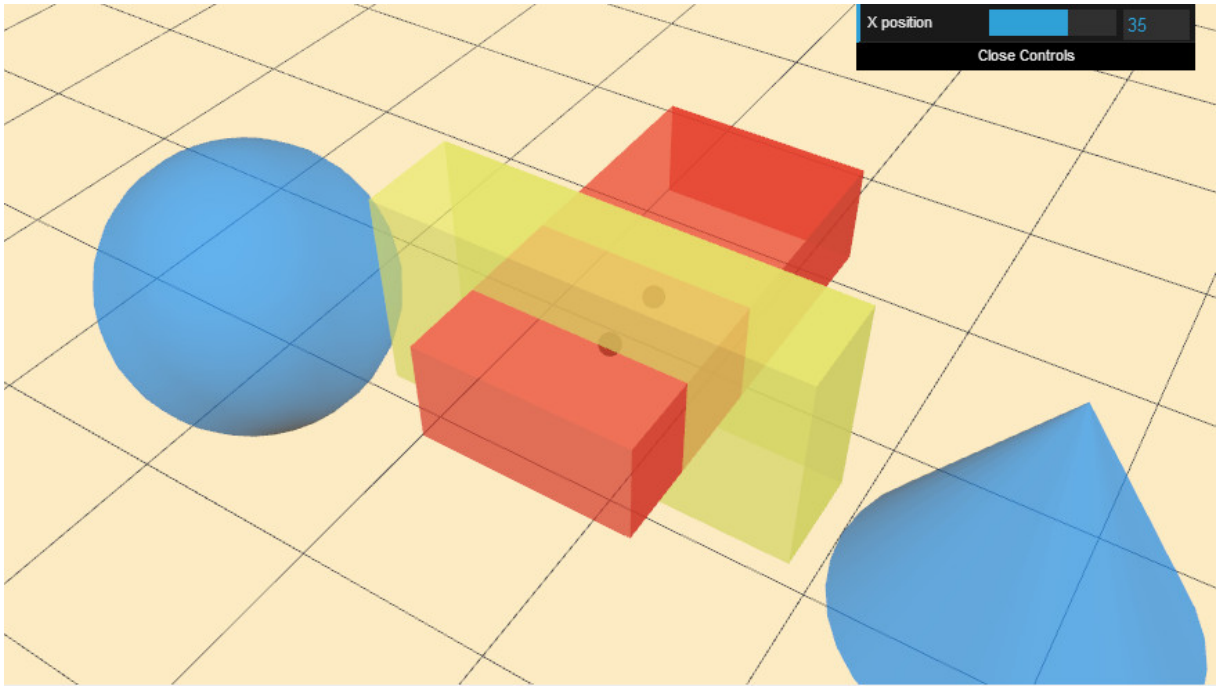
The fourth answer is also correct. The transparency algorithm states that the more distant object, the yellow one, is drawn first. Another way to think about it is that you can have red over yellow, or yellow over red, but not both at the same time.

Lesson: Advanced Transparency Methods



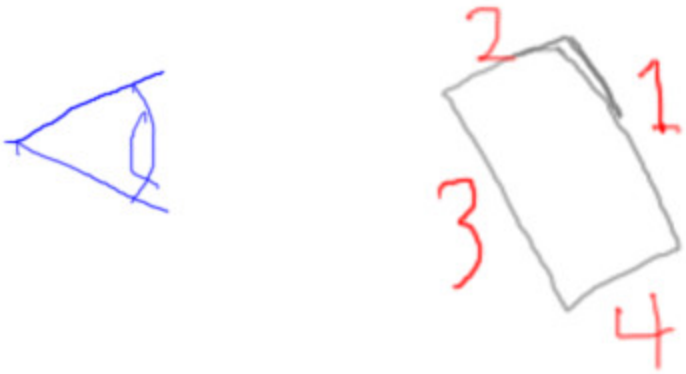
There are even more problems with simply sorting objects and drawing them roughly back to front. For example, notice that backfaces are not shown for any transparent objects.

[put this next to the first picture, leaving room at the bottom to draw]

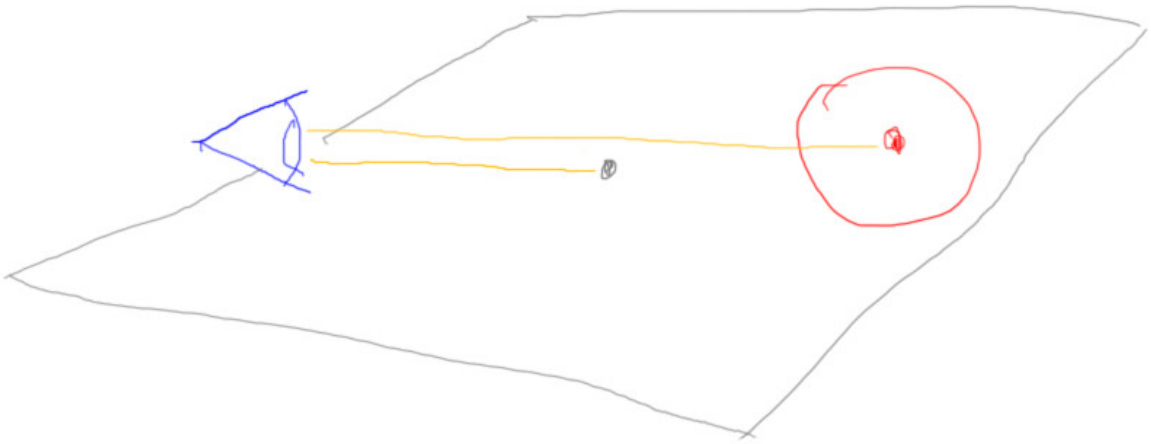


If you turn these on, you'll start to get strange rendering artifacts. Some backfaces will be drawn

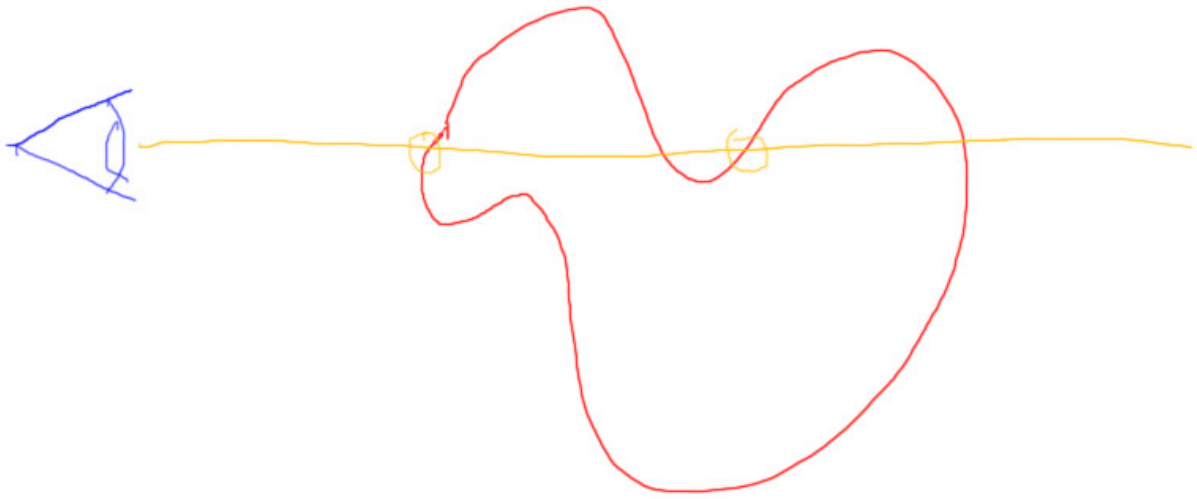
before frontfaces blend with them, some won't, giving a patchy look.



For example, here's a view of a block from the side, along with an order that the faces are drawn. Since face 1 is drawn first, face 2 blends with it. Face 3 is drawn, filling the z-buffer and also blends with face 1. Face 4 is behind face 3, so is invisible when rendered. Of the two backfaces, in this case only face 1 had an effect on the image generated.



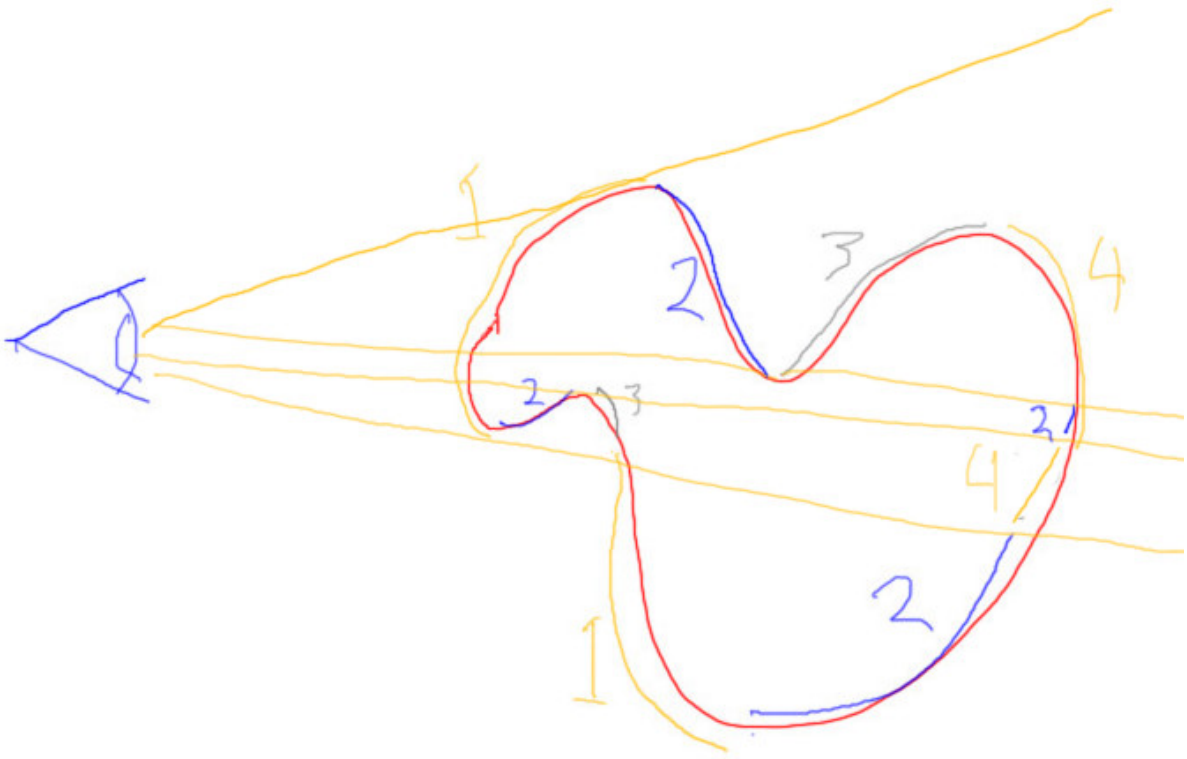
Even with backface culling on and a good sort order, you can have problems with different-sized objects. Say a ground plane and a red object resting on it are both transparent. The middle of the ground plane is closer to the camera, so by sort order it will be drawn *after* the red object. This will make the ground plane to appear to be above the red object.



Here's one more problem. Complex objects can have two or more surfaces overlapping a single pixel. Again it depends on draw order as to what appears on the screen. If the fragment on the left is drawn first, then the fragment on the right will improperly blend with it at this pixel. There's even research about ordering the triangles in the mesh itself to avoid these problems, but this type of technique can be costly to apply.

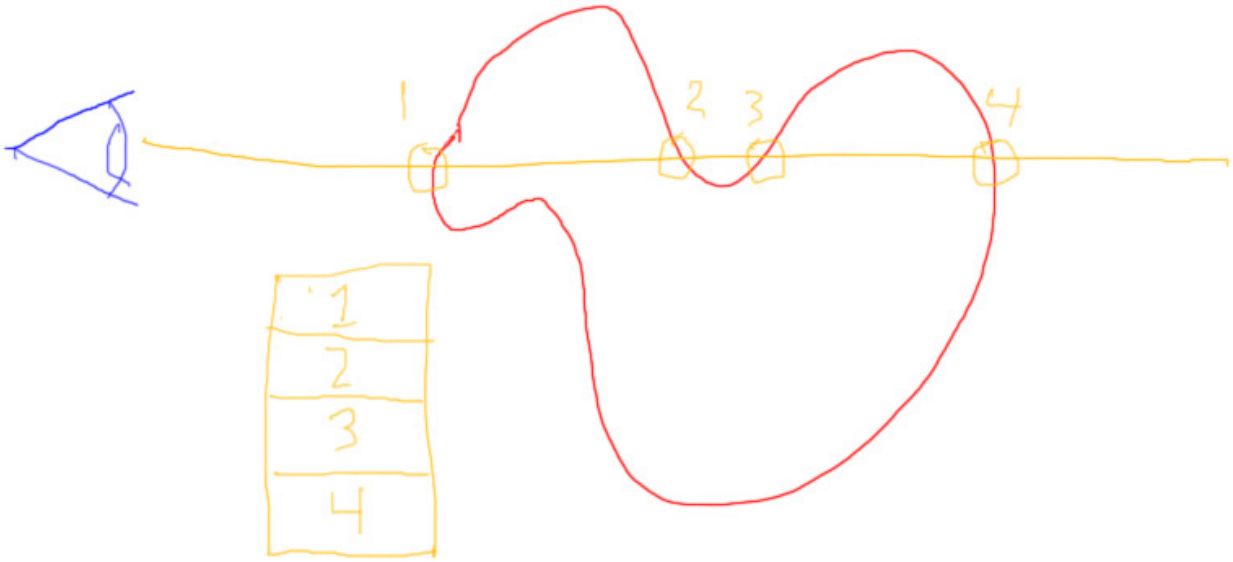
At this point you might have given up all hope for transparency always working right. The Z-buffer was meant to store only one object at a time, and unless you can guarantee the order of objects covering the pixels, there's little you can do. There are some further tricks you can do by only reading and not writing to the Z-buffer, but there's no perfect solution.

[[add phrase ***depth peeling*** to image - carefully draw the critical rays hitting the silhouette points first, so that it becomes easier to draw the layers.]



If you desperately want to get the right answer, there is one technique that works fairly well, called ***depth-peeling***. The idea is to peel away each transparent layer until all layers are processed, by storing an additional minimum z-depth for each pixel. So, for example, the first layer is all the transparent surfaces closest to the camera. The second layer is all the second closest surfaces, and so on. That's conceptually the key idea; see the additional course materials for more about this algorithm. The drawback is that each peel operation needs to have all the transparent objects rendered again. Many passes may be needed before all layers are found and processed.

[[Add ***A-Buffer*** to figure]



Ultimately what would solve the problem is storing a list of all the transparent fragments in each pixel, along with their depths. This approach is known as the ***A-Buffer***. Once we have all this information, we can then combine all these fragments in the right order.

[add words ***tile-based, stochastic transparency***]

This type of storage is possible in newer GPUs for the desktop, but can use a considerable amount of memory. What is interesting is that mobile devices usually use a different architecture, called ***tile-based***, that can more easily keep such fragment lists around.

Yet another approach is called ***stochastic transparency***, which uses screen-door transparency within the pixel itself along with some randomness to get a reasonable average result. This brings us full circle back to the original transparency algorithm I presented. There's no simple answer on the GPU.

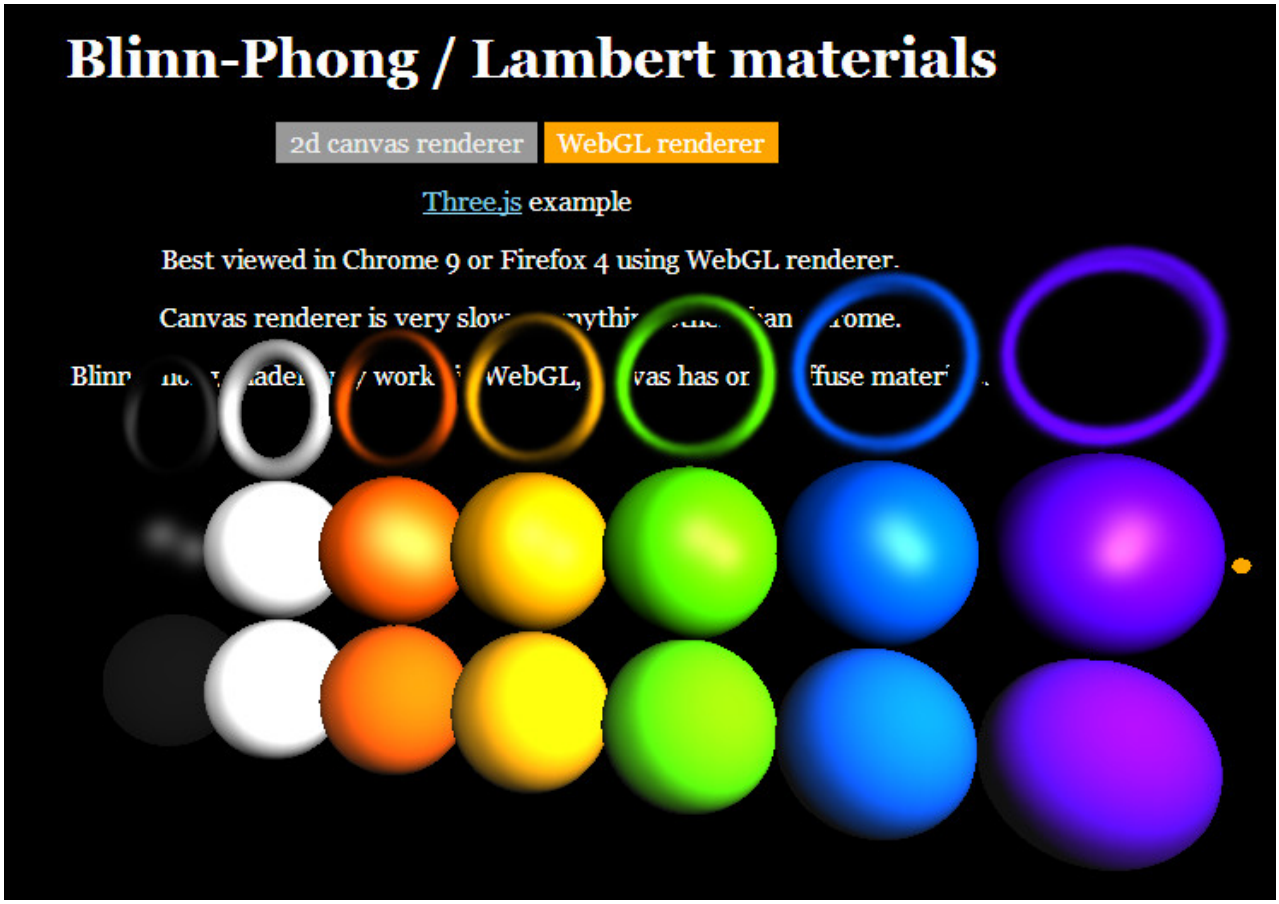
In the long-term this whole transparency mess may become a thing of the past as architectures evolve. For example, ray tracing is an algorithm that solves transparency issues directly. Be aware that currently transparency is not a solved problem on most GPUs and our simple solutions can have some serious artifacts. In the meantime, mellow out and enjoy the jellyfish!

[Additional Course Materials: You can read more about depth peeling here http://developer.download.nvidia.com/SDK/10/opengl/src/dual_depth_peeling/doc/DualDepthPeeling.pdf . This stochastic transparency research article discusses this technique, and also provides an overview of previous solutions: http://www.nvidia.com/object/nvidia_research_pub_016.html. Ordering triangles in a mesh to render in proper draw order regardless of view is discussed in this paper: http://dl.acm.org/citation.cfm?doid=2366145.2366179. Don't forget the jellyfish, one of my favorite WebGL programs http://chrysaora.com/]

Conclusion

[recorded 2/26]

[Show http://mrdoob.github.com/three.js/examples/webgl_materials_shaders.html - remove the text in the background as possible. Or zoom in a lot and cut off top?
]



“Now you know how to give 3D objects a variety of interesting materials and to make them look good. The next step is to learn how to precisely position and change the shapes of the objects themselves. With these two basics in place, materials and object transformation, you’ll be able to really control your virtual world.”

Problem Set

Problem 3.0

[***This problem should actually be swapped with the "Drinking Bird Colors" exercise. I'm putting it here so I can print it out, and as a reminder to swap.***
]

Let's try three color conversions. Given the color on the left, write the color on the right. Give only

Hexadecimal 0x0524D0 to integer _____, _____, _____
Integer 127, 200, 255 to float ____, ____, ____
(round to 3 decimals places)
Float 0.4, 0.0, 0.11 to Hexadecimal _____
(round to integer)

Answer

05 is easy, that's ***5*** in integer.
24 is 2 times 16, which is 32, plus 4, which is then ***36*** total.
The hexadecimal number D0 is 208, as D is decimal 13, and 13*16 is ***208***.

To convert to floating point, we divide by 255. Not much divides evenly by this value, so we round off. 127 converts to 0.49803 etc., which we round to ***0.498***. 200 divided by 255 is ***0.784***, 255 divided by 255 is ***1.0*** exactly.

0.4 times 255 is 102, in hex this is 6 times 16 plus 6, so 66. [write ***0x66***]
0.0 times 255 is zero, so in hex this is just two more zeroes. [write ***00***]
0.11 times 255 is 28.05, which we round to 28, which is 1C in hexadecimal [write ***1C***]

Problem 3.1: Drinking Bird Materials

Let's change some materials on the drinking bird to look a bit more realistic. Change the follow materials to use the

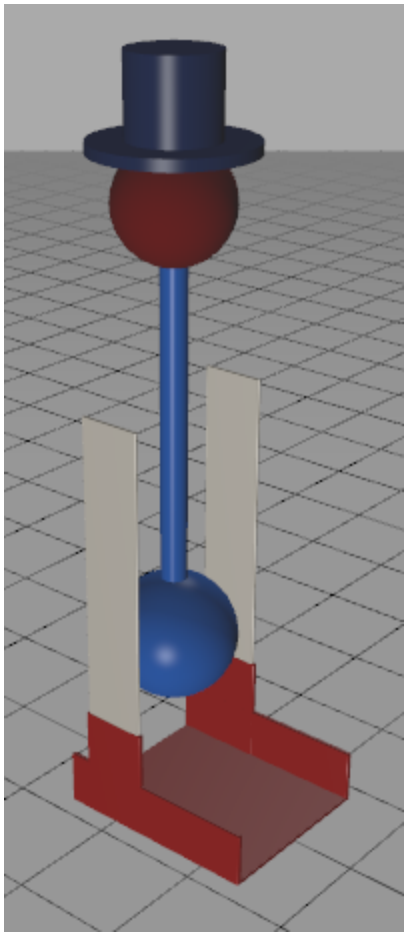
[leave room to right for solution image, further down]

MeshPhongMaterial:

hatMaterial, bodyMaterial: set the specular color to 0.5,0.5,0.5, shininess to 100
legMaterial: set the specular color to 0.5,0.5,0.5, shininess to 4
footMaterial: set the specular color to 0.5,0.5,0.5, shininess to 30

See the additional course materials for the link to the three.js documentation.

When you're done, the result should definitely look shinier:



[Instructor Comments: you can find the three.js documentation here http://mrdoob.github.com/three.js/docs/54/. You can find the MeshPhongMaterial in the left column.]

Answer 3.1

[solution is here: *redacted*]

This exercise is mostly here to get you to try out the MeshPhongMaterial. The code involves setting a few parameters on each material. Here's one typical solution for the hatMaterial changes:

```
var hatMaterial = new THREE.MeshPhongMaterial( { shininess: 100 } );
hatMaterial.color.r = 24/255;
hatMaterial.color.g = 38/255;
hatMaterial.color.b = 77/255;
hatMaterial.specular.setRGB( 0.5, 0.5, 0.5 );
```

Here the shininess gets passed in as an initializer, and the specular color is set separately.

Problem 3.2: Improved Bird

I modified the drinking bird's geometry itself a bit from the original model. See the additional course materials for a link if you forget what the model looked like when you first made it.

Which part has slightly different geometry?

( ) hat
( ) head
( ) body bulb
( ) legs and feet

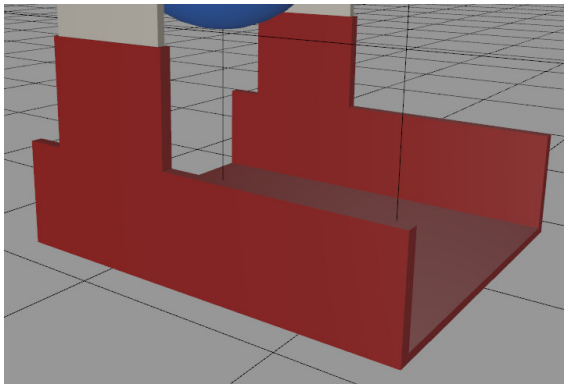
Once you figure out what part has changed, think about why I modified it. I won't test you on this - sadly, the computer can't currently grade essay questions - but there's an interesting reason.

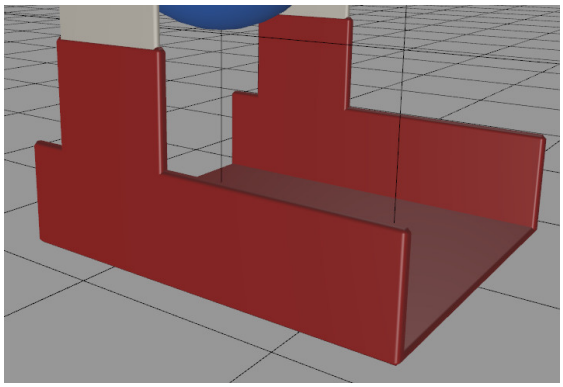
Answer 3.2

I changed the feet and legs of the model to have rounded edges. So, instead of a block with perfectly sharp edges, there's a small bevel. This detail looks a bit more realistic, since most plastic parts are not razor sharp. More importantly, a bevel lets the model pick up specular highlights along its edge.

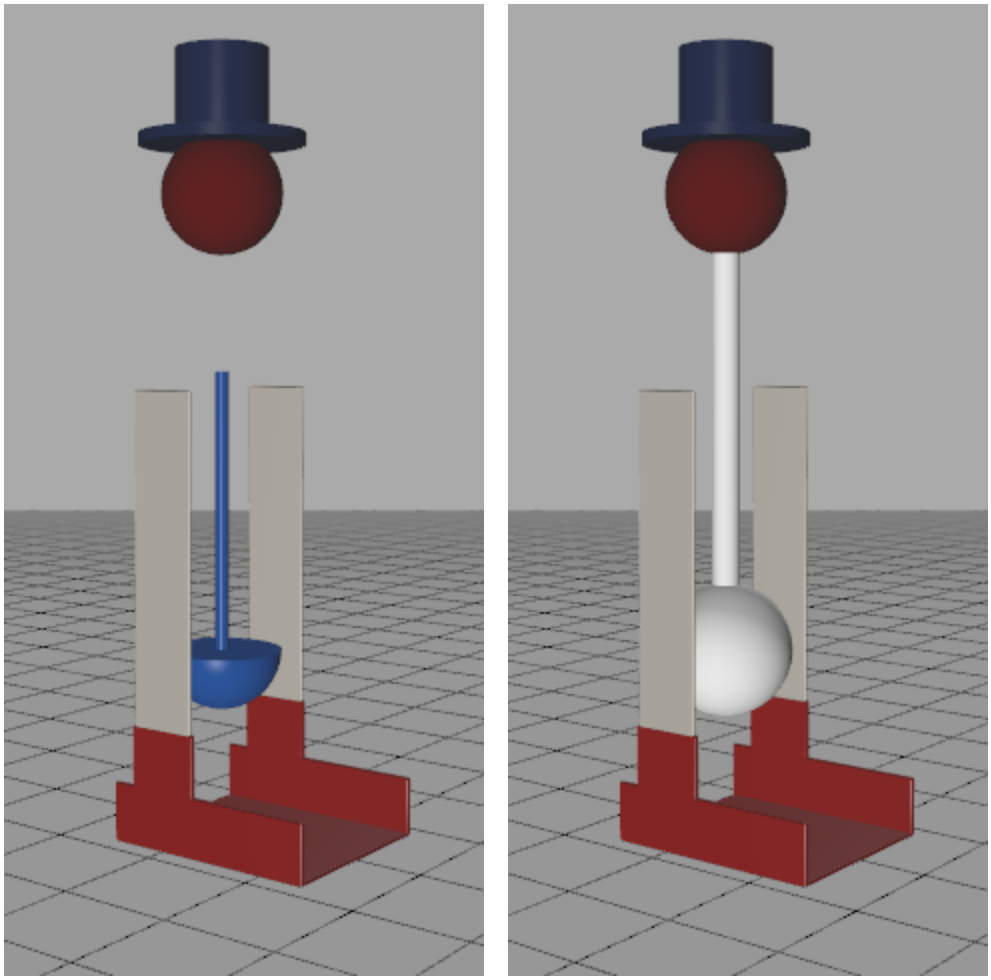
Here’s sharp edges versus smooth. It’s worth keeping in mind that specular highlights don’t add all that much for flat surfaces, which is why I used the teapot for most material demos in this unit.

[sharp edge version unit3_db_material_solution_no_bevel.html]





Problem 3.3: Glass Body Bird



Let's improve the drinking bird look a bit better still by making its body out of glass. I'll do the geometry improvements, you do the glass material.

I've now split the body into two pieces: an exterior shell, and an interior blue fluid. On the left is the blue fluid without the shell, and on the right is how things look with the shell. Clearly the shell needs to be made, well, clear, in order to see the fluid inside. Your job is to make this material transparent.

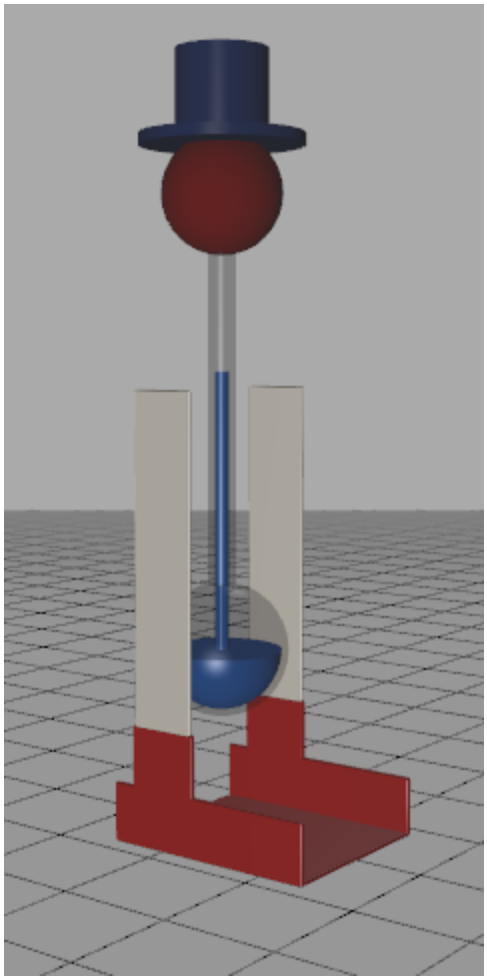
The fluid's material is called "***bodyMaterial***" in the code, the glass body uses "***glassMaterial***". The currrent glassMaterial's shininess is fine, but otherwise needs fixing.

I want you to set the glass body so that it's

30% opaque
Regular material color fully black
Specular color fully white

Definitely look up the documentation on this material to learn how to set the specular color.

Answer



The solution is this piece of code:

var glassMaterial = new THREE.MeshPhongMaterial({ color: 0x0, specular: 0xFFFFFF, shininess: 100, opacity: 0.3, transparent: true });

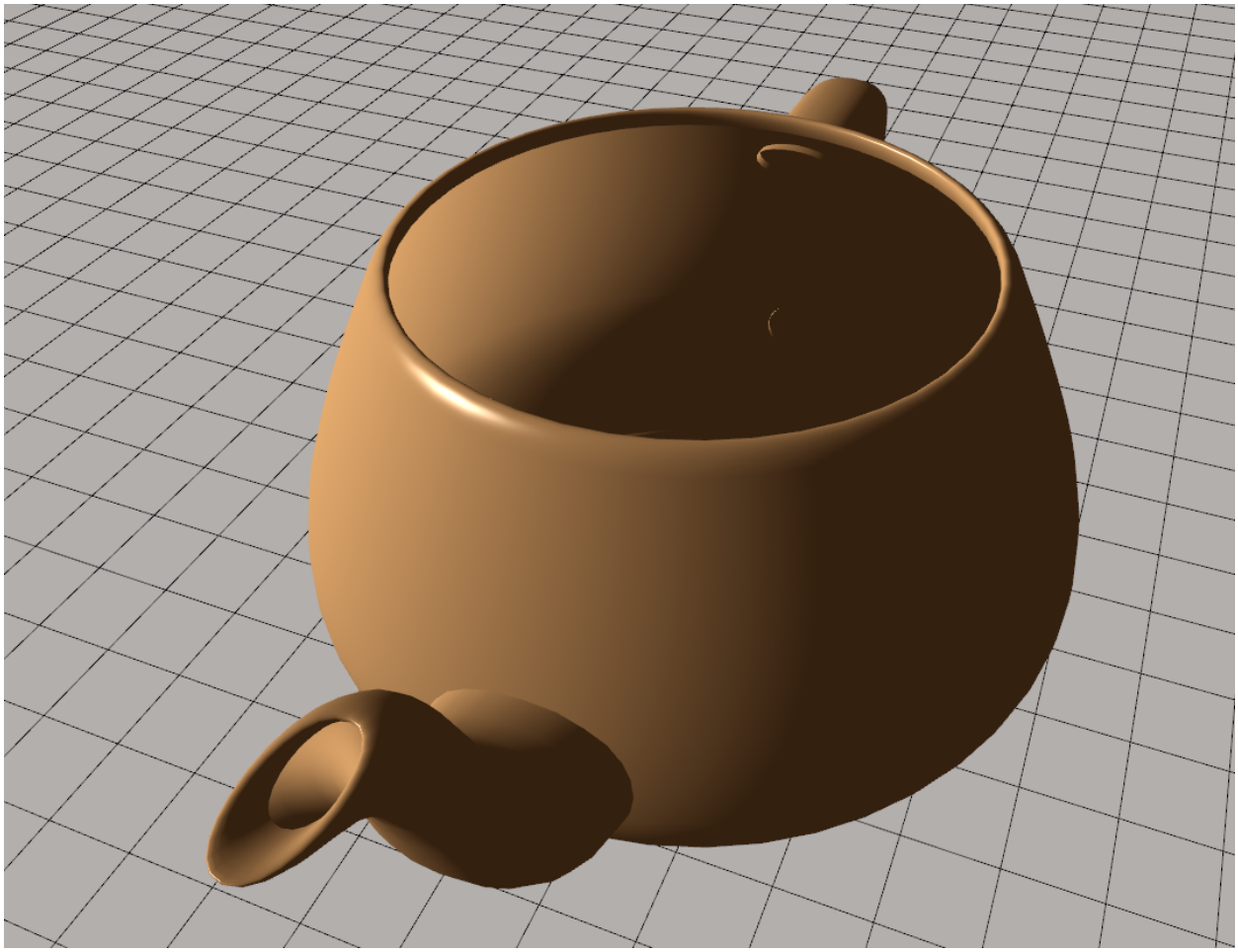
```
var glassMaterial = new THREE.MeshPhongMaterial(
    { color: 0x0, specular: 0xFFFFFF, shininess: 100,
      opacity: 0.3, transparent: true } );
```

Opacity is set to 0.3, which is 30%, and the transparent flag is turned on. The color variable is set to black, the specular to full white.

Problem 3.4: Paper Lantern Shading

Here's a picture of the teapot with the lid off.

[backface inside]



You can see the inside of the teapot. What's odd here is that the outside and inside surfaces have the same lighting. This is because the same normals are used to calculate both the inside and outside surfaces. That is, all the normals point outwards, regardless of whether we're looking at the inside or outside. This is certainly not how the real world works!

[show a light inside a "cup" with normals pointing outwards - inside of cup is all dark]

http://commons.wikimedia.org/wiki/File:Pontocho_paper_lantern_by_James_Disley_in_Kyoto.jpg
]

DRAW HERE a schematic of the lantern, showing the normals pointing outwards.

I call this the “paper lantern” effect, where the surface is sort of like a paper lantern, looking the same on both sides. However, if you think about it, the shading here is really not like anything in the real world. For example, if we put a light inside this object, most of the surfaces would be dark. This is because their normals all point away from the light.

Say we want to fix this problem and have the surface be double sided.

Which of the follow fixes will work?

[] If the normal is facing away from the light, don’t render diffuse and specular
[] Render the model twice: once with backface culling on, once with frontface culling on and flipping each shade normal direction.
[] Make and also render a second set of triangles in the same locations, but with their normals reversed. Do not cull.
[] Put a second light directly opposite the first light, pointing in the opposite direction.

[Instructor Comments: The demo for the backface teapot can be found here http://www.realtimerendering.com/udacity/?load=demo/unit1-teapot-demo.js - remove the lid]

Answer

The first answer doesn't solve anything. It's just saying to do what we already pretty much do: shade the surface only if it points towards the light.

The second answer works. You can use a slightly different vertex shader to render the object's geometry a second time, reversing the normal before using it.

The third answer doesn't work because culling is off. If you render two meshes in exactly the same location, you will get z-fighting. This answer as stated is incorrect. If culling was turned on and the data set properly, this solution would work. While this solution doubles the size of the geometry you need to store, it does have the advantage of not requiring any special treatment. It's just another object to render in a normal way.

The fourth answer will make all surfaces get some light. This fix will have the effect of properly making the dark inside of the object be lit. However, there are also areas inside that are lit that shouldn't be, and these will continue to be lit. Similarly, the outside of the object will now be lit on both sides, which gives too much light. So, this answer is wrong.

Problem 3.5: Glass Cube

Say we use depth peeling or other advanced transparency technique. We're getting great pictures of transparent objects. Front faces and back faces are both rendered, and in the right order. Life is good.

However, we then see another rendering problem when we put a blue glass cube on a wood table. What is this major problem that gives horrible artifacts that change from frame to frame as the view changes?

[What is the major problem with a clear glass cube on the table?]

( ) The bottom of the cube has z-fighting with the top of the table.
( ) Absorption of the light is not properly accounted for by the GPU.
( ) The bottom of the cube is black, due to total internal reflection.
( ) The sides of the cube do not refract the light, causing black areas to occur.

Answer

The problem is that you get z-fighting with the bottom of the cube and the top of the table. The other three answers may or may not be true, but they would not cause serious artifacts that would change with the view.

[Additional Course Materials: You can read more about this transparency 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, which is where I got the idea for this problem.]

[Need two more problems here]

Problem 3.6: Shading and Illumination Costs

We have ***Phong*** compared to ***Gouraud***, and we have ***Blinn-Phong*** compared to ***Lambertian***.

[put these four terms in a grid:

Phong Gouraud
B-P
L

]

Say we have a scene and try each of these four combinations. Assume that shading costs in execution time far outweigh the illumination costs. In other words, if a fragment shader is used to evaluate illumination, it will be considerably more expensive than any vertex-based shading approach.

Assign each combination a number, rating them from least to most expensive in cost. In other words, “1” means the smallest amount of time needed, “4” means the largest.

Answer

[Phong Gouraud
B-P 4 2

L 3 1

]

To answer this, you have to know what all the words mean. Shading refers to per vertex vs. per pixel. with Gouraud being per vertex and Phong being per pixel. We’re given that per pixel will be considerably more costly than per vertex, so this side of the grid will be more expensive.

Illumination refers to the Lambertian diffuse model vs. the Blinn-Phong specular model. Since the Blinn-Phong model builds on the Lambertian model, adding shininess, it must be more costly. So this top part of the grid will be more expensive.

These two facts establish that Gouraud and Lambert is the quickest, and Phong shading with Blinn-Phong illumination is the most expensive. The tie break between these other two combinations is that per-pixel shading is given to be much more costly than vertex shading. This tips the balance so that Gouraud shading and Blinn-Phong illumination is faster than Phong shading of Lambertian illumination.

[7th problem? HSL Visualization: make a volume showing HSL]