Lesson 9: Shader Programming

Lesson: Overview

[recorded in early March or so, 01_l]

[This is where I roll in a bunch of real materials - there's more to life than diffuse, specular]

[put three balls on.]

There are a few canned materials that three.js provides. You can have a diffuse surface, a glossy surface, and a reflective surface. You can make objects transparent [put lens]. You can even apply textures [rotate ball and show texture].

In this unit we'll start by digging a bit deeper into the graphics pipeline. One important element of the modern GPU is that significant parts of its pipeline can be programmed to do whatever we want.

[start putting different materials on]

This is particularly compelling for displaying materials. You can use a wide variety of equations to simulate how a surface looks.

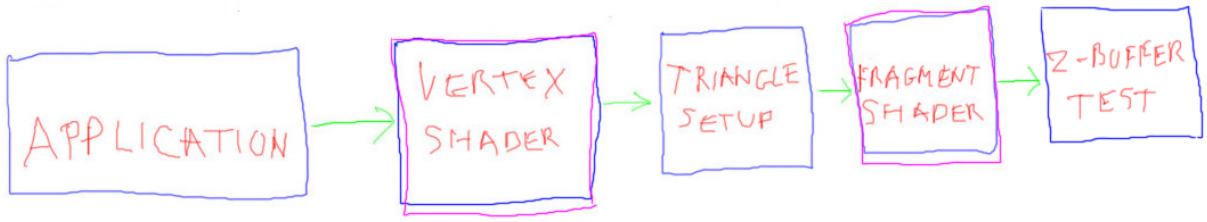
With this ability to fully control the illumination model, you can create a wider and wider range of substances as you learn more techniques. You can create objects that look refractive like glass, or semitransparent and hollow. [glass ball, hold in front of stuff and then put down. Green plastic ball.] You can make objects have unusual optical properties [anisotropic].

Once you get into illumination models and shader programming, the world takes on a slightly different look. I know myself that when I look at some unusual mineral or beautiful wood grain

[pour things on, more and more]

Lesson: Programmable Shaders

[show pipeline with vertex shader and fragment shader]



Here's our basic programmable pipeline, with a vertex shader and fragment shader. Once upon a time both of these stages were handled by what is called the ***fixed-function pipeline***. Transistors in the GPU were specifically dedicated to the transform and rasterization processes. The programmer basically sets a bunch of switches and values that control the flow of data. However, there's no real programmability involved. The **Nintendo Wii** is about the last to used this type of architecture.

In 2002 GPUs started to appear that included vertex and fragment shaders. A *shader* is truly a programmable piece of the pipeline. In other words, you actually send little, or sometimes large, shader programs to each of these units. These programs are written in a C-like language, if you know what C or C++ looks like. Up to this point we've been having three.js do the programming for us. When we create a material, three.js actually creates two small programs, one each for the vertex and fragment shaders. When the object with that material is to be displayed, these shaders are loaded with their programs. The triangles are then sent down the pipeline and the programs are executed.

Lesson: Computer Chip Design Challenges

This decision to make parts of the pipeline programmable is in fact one of the challenges of computer chip design. The designers have to decide how many transistors are dedicated to memory caches and registers, how many to instruction processing, and how many to dedicated algorithm logic.

[list out:
*** memory**

*** instruction processing**
*** algorithm processing** - later add phrase **fixed-function processing**
]

CPUs spend their transistors on memory caches and instruction processing. GPUs used to spend theirs almost entirely on algorithm logic, which is also called fixed-function processing. [gesture at top pipeline, if it's still on the screen...]. What's great about fixed-function logic is that it can be extremely fast, while using less transistors and less power. The drawback of fixed-function logic is that you're locked in.

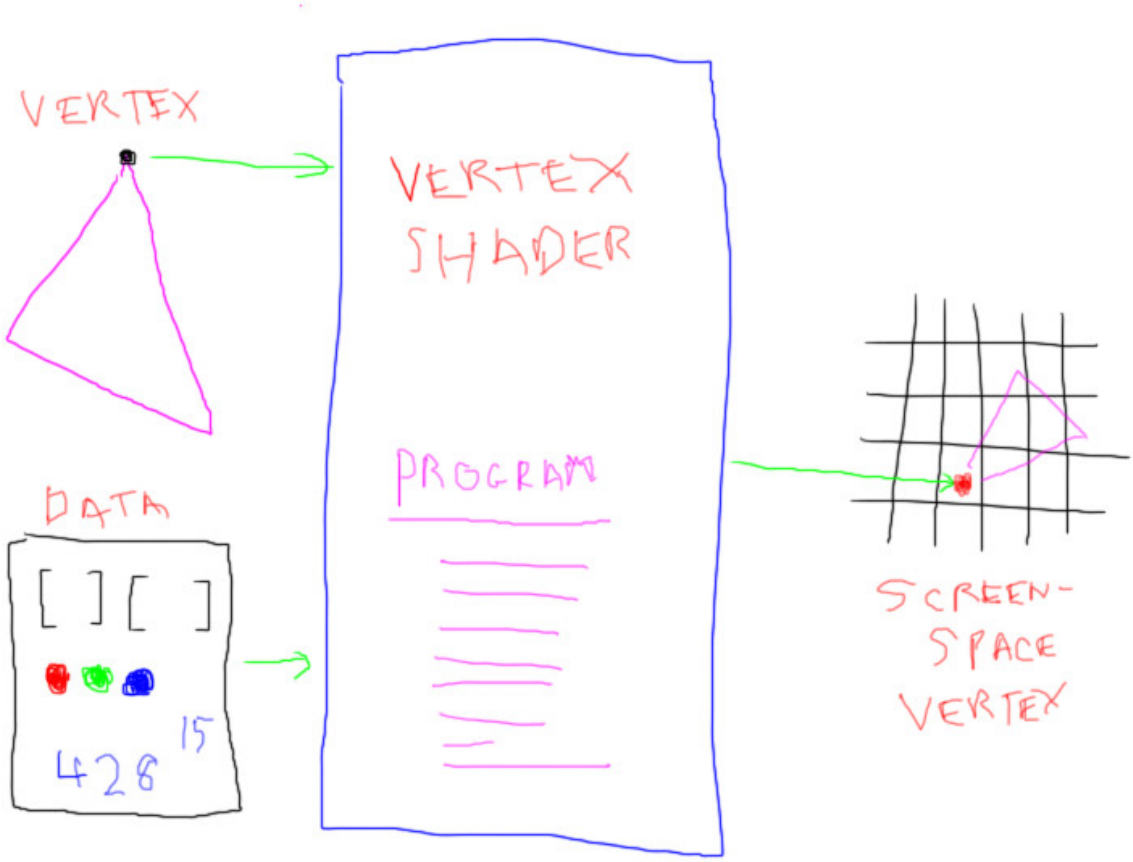
The trend over the past decade or so has been to add to the GPU more and more instruction processing, in other words, more general programmability. In our pipeline the vertex and fragment shaders are programmable. Only those tasks that are commonly performed and have a large bang for the buck have survived as fixed-function hardware in the pipeline. For example, triangle setup and rasterization, as well as z-depth testing, are used all the time and do not normally need programmability.

[Additional Course Materials:
Here is [the OpenGL pipeline](http://openglinsights.com/pipeline.html) in great detail; from the book ["OpenGL Insights']](http://www.amazon.com/OpenGL-Insights-Patrick-Cozzi/dp/1439893764?tag=realtimerenderin).
]

Lesson: Vertex and Fragment Shader

The vertex shader performs the transform of the vertex position to the screen. Its inputs are a vertex from the triangle, along with whatever data the programmer wants to provide. For example, the color of the material could be passed in. The output of a vertex shader is a vertex with a transformed position and possibly other information.

[vertex shader inputs and output diagram. Compare to fragment shader diagram, put reusable stuff in a separate layer.]



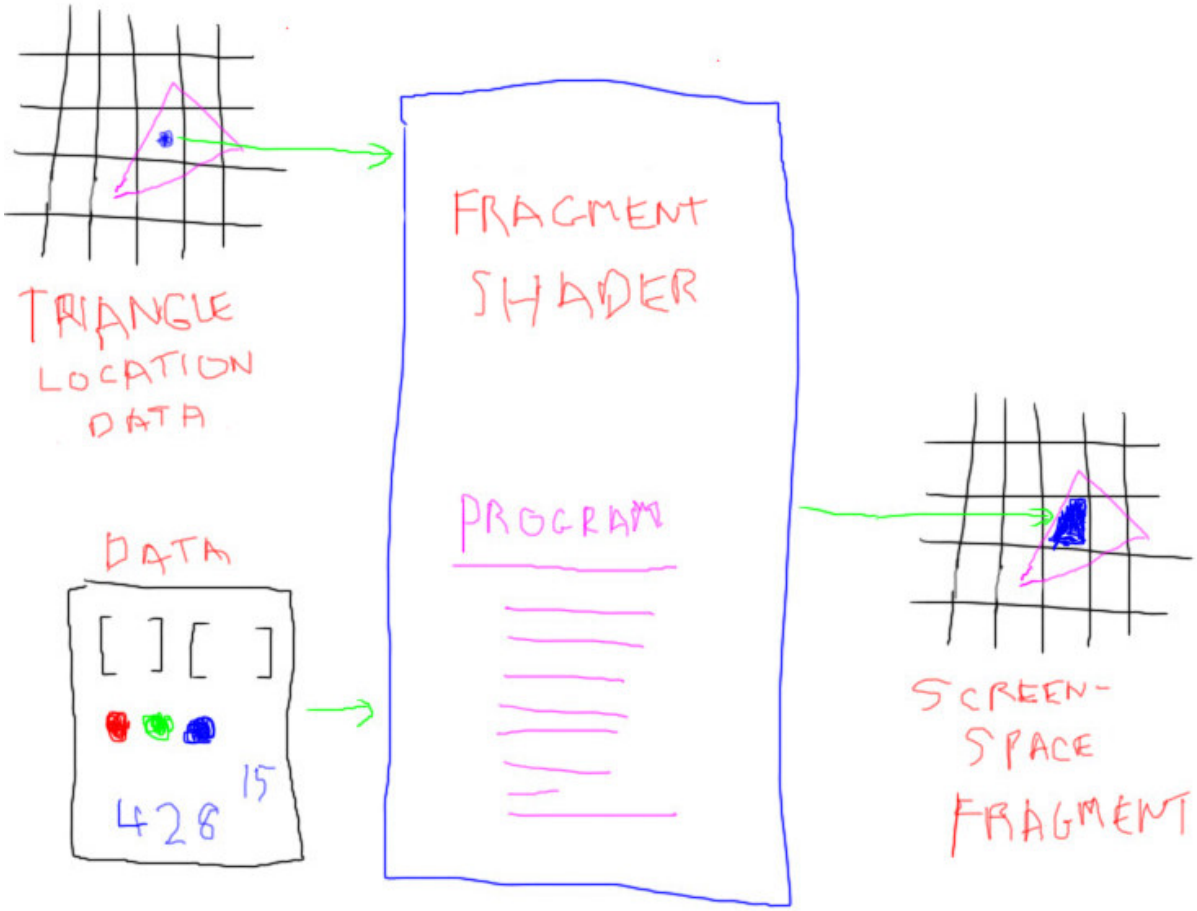
The transformed triangle is then rasterized.

Triangle setup sends the data at each pixel inside the triangle to another programmable unit, the fragment shader. If you use Microsoft's DirectX API, this is called the pixel shader.

[**fragment shader - WegGL, OpenGL...**
pixel shader - DirectX
]

The fragment shader is handed various information from the triangle being processed. Similar to the vertex shader, the programmer can feed in any other data needed to process the triangle's data at each pixel. The fragment shader program is run. The output, not surprisingly called a fragment, is typically a color and a z-depth. By the way, the reason we call it a fragment is that it represents the piece of the triangle covering the pixel.

[fragment shader diagram - we can reuse the vertex shader diagram mostly.]



]

At this point this fragment, color plus z-depth, is compared to the stored depth in the Z-buffer. If the fragment is closer than the z-depth previously stored, the triangle is visible at this pixel and its color values are saved. The z-buffer test is again a fixed-function bit of hardware.

Notice how the fragment shader is pretty similar to the vertex shader in the way it functions. In fact, modern GPUs use what is called a unified shader in the hardware itself. These shader processors are assigned on the fly by the GPU itself to execute vertex shader or fragment shader programs, depending on where the bottleneck is found.

[end recording in early March]

Problem: Fragment Shader Bottleneck

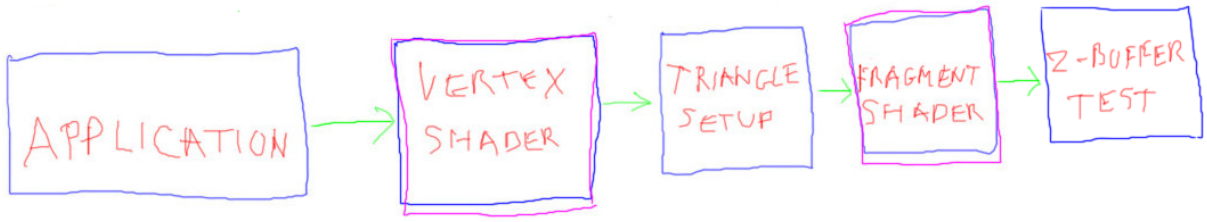
[recorded 4/7]

[I have this graphic already drawn, reuse it, I assume it's stored in Unit 8 - redo because answer

is missing from original version.]

Say you're given this pipeline

[application -> vertex shader -> primitive assembly -> fragment shader -> z-buffer test]



You're told that the fragment shader is turning out to be a huge bottleneck in this extremely primitive GPU design made by Acme GPUs and Bubble Gum Incorporated. Their GPU has a single vertex shader and a single fragment shader processor. This is pretty pitiful, and is in fact what differentiates various GPUs in the same class: some will be more powerful simply because they have more shaders on the chip. Modern GPUs process hundreds if not thousands of vertices and pixels simultaneously with these cores, which is why they are called massively parallel.

Acme knows that they need to add more cores to be competitive, so they're making a much larger chip. They know that it will be able to have a few hundred separate vertex shaders and separate fragment shaders, plus room for other features.

You've been hired to design their next chip, which will have many more transistors available. Your team brainstorms a few architectural ideas and you're asked to evaluate these.

Worth considering

[ ] Reverse the order, performing the fragment shader before the vertex shader.
[ ] Test the z-depth just *before* running the fragment shader and abort if this fails.
[ ] Make it so all shader cores can be used as vertex shaders or fragment shaders.
[ ] Divide each triangle into 4 smaller triangles and render these.

Personally, I think half of these ideas are worth exploring further.

[Additional Course Materials:
Interestingly, you can visit [this web page](http://analyticalgraphicsinc.github.io/webglreport/), which will report on the capabilities of your machine. Within WebGL there are also [extensions](http://www.khronos.org/registry/webgl/extensions/), proposed new capabilities not supported by all (or sometimes, even any) machines.

]

Answer

They say in brainstorming sessions, “there are no bad ideas”. Fair enough, but at some point you have to cull out the ones that sound counterproductive or incoherent. The first idea here is in that category, since it makes no sense at all. The input to a fragment shader is a point inside a triangle, typically produced by triangle setup. The application doesn’t feed surface points to the GPU, it provides triangles. Even if it fed such points to the GPU, these points would not normally be properly placed on the screen. That’s something the vertex shader does, transform the triangle to screen coordinates. So, forget idea number one.

This second idea’s a good one. If we know early on that the fragment is not visible, then we don’t have to use the fragment shader at all. GPUs take advantage of this and so perform the z-depth test early, so saving on executing the fragment shader’s program. This type of speedup is called **Early-Z**. After the fragment is computed, it is then put into the Z-buffer and color image as usual.

However, there’s a little subtle catch with this idea. If the fragment shader itself actually changes the z-depth passed in, then you *cannot* safely check the z-depth before running the shader program. That said, about 99 point 44 one-hundredths of all fragment shaders never touch the z-depth. In fact, in WebGL it’s not currently possible, though this has been proposed as an extension.

This third idea, making each core be usable as either a vertex or fragment shader, is worthwhile. This in fact is how modern GPUs are architected. Vertex and fragment shaders are close enough together in functionality that a single core can be used as either. This is called a “***unified shader***”. An advantage for the fragment shader bottleneck problem is that such cores can be allocated on the fly. If fragments are piling up, cores are assigned to them, if vertices are queued up, cores are moved to this area.

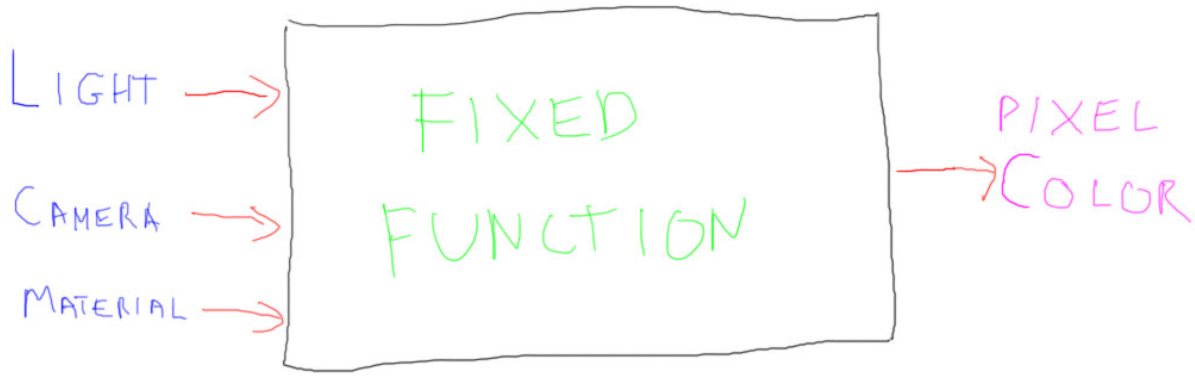
The fourth solution doesn’t change anything as far as the fragment shader goes. The same number of pixels will be covered and generate fragments, so that the fragment shader still has to work as hard. Worse yet, adding more triangles means more strain on other parts of the pipeline, such as the vertex shader. So this idea causes more overall work, not less.

Lesson: Shader Architecture

The Blinn-Phong reflection model looks pretty good. It’s been used for well more than 30 years because it’s fairly flexible and quick to compute. The three.js library provides this material with its **MeshPhongMaterial**.

On fixed function graphics hardware, where you could not program the shaders, this reflection model was the one locked into transistors and could not be changed. Back then you could simulate other reflection models by various tricks and setting a large number of switches on the chip, but for the most part you were stuck with what was burnt into the chip.

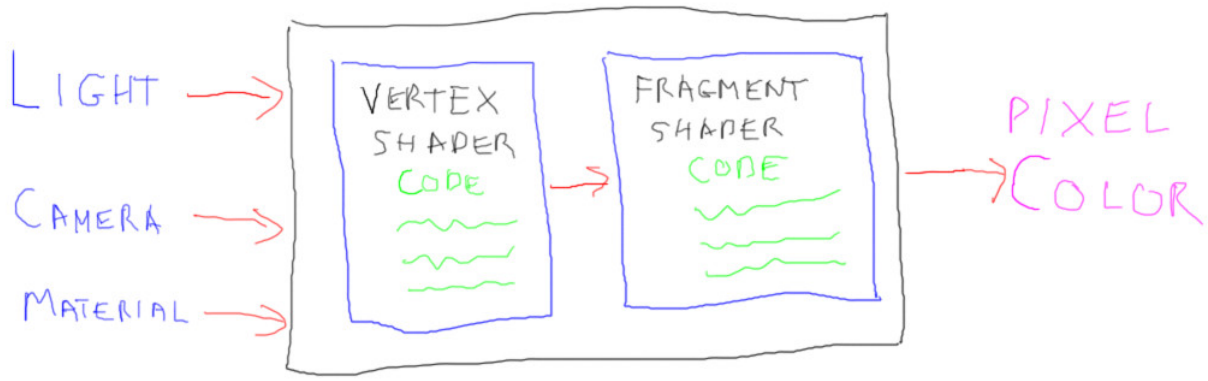
[draw light, eye, material as inputs to large black box, color out other end. Do green label last! Easier to erase. MAKE BOX BIG - we'll need all the room below.]



Once vertex and fragment shaders became available, all this changed. You could now program whatever functions you wanted.

Lesson: Shader Inputs

[Redo image below entirely! Show VS and FS but with uniform, varying and attribute inputs/outputs.]



[Note: image above needs an entire redo. Show highlit words below.]

When you select a material, what the three.js library does under the hood is to create efficient vertex and fragment shader programs for you. However, you can create and modify your own materials by using three.js's **ShaderMaterial**.

Vertex and fragment shaders have similar structures. You first define the inputs to the shader. There are two kinds of declarations in three.js: **uniform** and **varying**. In WebGL there are also "***attributes***" for the vertex shader, which are the values such as the vertex position, normal, and UV texture coordinate data. These attributes are built in for three.js so do not have to be declared. They have the names:

position, normal, uv [note for three.js only]

Uniform data is whatever is constant for the triangle being shaded. For example, values such as the positions of the light sources and the color and shininess of the material are uniforms.

A value is labeled "varying" if it is computed or set by the vertex shader as an output and passed to the fragment shader as an input. The vertex shader outputs these values, and then the rasterizer interpolates these across the triangle's surface. In other words, these values can and will vary per fragment. You are in total control of what gets interpolated. The more data per vertex that you want interpolated, the more time and resources it takes - nothing comes for free.

The fragment shader also takes uniform data as inputs. The fragment shader then, as a minimum, outputs a fragment color, with the name **gl_FragColor**. I say "as a minimum", because in fact in many graphics APIs the fragment shader can output to a number of images at one time. This is called ***MRT***, for ***multiple render targets*** - "targets" is another name for the output images. Unfortunately, WebGL does not support this - yet.

[final text:
ShaderMaterial
Vertex Shader
Inputs
uniforms - light positions, material color, shininess, etc.
attributes - data stored in the vertex; built into three.js
Output
at least gl_Position
varying - any data to be passed to fragment shader

Fragment Shader
Inputs

uniforms - light positions, material color & shininess
varying - data output by the vertex shader
Output
at least gl_FragColor
]

[Additional Course Materials:
Look [here](https://github.com/mrdoob/three.js/wiki/Uniforms-types) for a list of uniform types.

In three.js you can use custom attributes with ShaderMaterial. It's a bit awkward but it works, see examples
[here](http://mrdoob.github.io/three.js/examples/webgl_custom_attributes_particles.html) and
[here](http://mrdoob.github.io/three.js/examples/webgl_custom_attributes.html).
]

Question: Legal Inputs and Outputs

I've run through the basics of what the vertex and fragment shaders use for inputs and outputs. Say you have a triangle with a shading normal stored at each vertex.

Which of the following are true?

[ ] The vertex shader can output the average position of the three vertices.
[ ] The fragment shader can use the interpolated shading normal.
[ ] The vertex shader can output a color for the fragment shader to use.
[ ] The fragment shader can use the position of the third vertex in a triangle.

Check the box if the statement is true.

Answer

The vertex shader knows only about a single vertex at a time, it normally does not know about the whole triangle. So this first statement is false - the vertex shader is given just one vertex at a time so has only its position.

The second statement is true: the vertex shader can output the shading normal, which is then interpolated across the triangle's surface. The fragment shader will probably want to normalize the shading normal before using it.

The third statement is also true: even though the vertex itself does not have a color stored in it, the vertex shader can compute and output anything it wants. Even though we call the outputs

things like “position”, “normal”, “color”, and so on, as far as the GPU is concerned it’s all just data to interpolate.

The fourth statement is false, for the same reason the first is false. Each vertex is processed separately by the vertex shader. So, data from a specific vertex cannot be passed to the fragment shader in this way.

[***geometry shader*** - yes, it’s in an answer, but not vital]

You could work around this limitation by doing expensive things like storing the data for all three vertices at every vertex and have the vertex shader sort it out. A better solution is to use a ***geometry shader***, if available. Currently it is not supported by WebGL (how many times have I said this?), but I expect this will change in the future. When available, the geometry shader is optional, and is applied after the vertex shader. It allows you to access the data for the triangle as a whole. You can in fact use the geometry shader to create new points, lines, and triangles.

Lesson: GLSL ES

The heart of a shader is the actual shader program it runs. This program is passed to the GPU as a string of characters. I’ll repeat that, since it’s a bit unusual: instead of sending compiled code, or binary data, or some other processed information to the driver, you actually send it a string of characters.

WebGL shaders are written in ***GLSL ES***, the ***OpenGL Shading Language for Embedded Systems***. No one calls it GLSL ES, everyone simply says GLSL. If you use ***DirectX***, you’d use ***HLSL***, their ***High Level Shading Language***. In either case, the character string is what gets sent to the graphics driver. The graphics driver compiles these into assembly instructions that are sent to the GPU.

```
vertexShader: [

    "uniform vec3 uMaterialColor;",
    "uniform vec3 uDirLight;",

    "varying vec3 vColor;",

    "void main() {",
        // Transform the vertex from model space to screen space
        "gl_Position = projectionMatrix * modelViewMatrix * vec4( position, 1.0 );",
        "vec3 light = normalize( uDirLight );",
```

```
                    // Compute a diffuse color using Lambertian reflection, N * L
                    "float diffuse = max( dot( normal, light ), 0.0);",

                    "vColor = uMaterialColor * diffuse;",
                "}"
            ].join("\n"),
```

```
vertexShader: [
    "uniform vec3 uMaterialColor;",
    "uniform vec3 uDirLight;",

    "varying vec3 vColor;",

    "void main() {",
        // Transform the vertex from model space to clip coordinates
        "gl_Position = projectionMatrix * modelViewMatrix * vec4( position, 1.0 );",
        "vec3 light = normalize( uDirLight );",

        // Compute a diffuse color using Lambertian reflection, N * L
        "float diffuse = max( dot( normal, light ), 0.0);",

        "vColor = uMaterialColor * diffuse;",
    "}"
].join("\n"),
```

Here's how to create a shader string in JavaScript. The first line is how JavaScript says it's going to define an array. What we're doing here is defining a string for each line of our program.

That's one way to get shader programs into three.js, but it's not usually the most convenient. There are all these quote marks and commas, which are visually confusing and a pain to add and maintain. That said, one advantage of this approach is that you can glue together strings on the fly to make shaders. This is in fact exactly what three.js does. When you ask for three lights and a shiny material, it creates a custom shader for exactly those elements.

If you have a specific, fixed shader in mind, an easier way is to make your program a separate text resource. I won't work through the mechanics here - this isn't really a web programming class - but see the code in this course's Github repository for how we do it. Basically, you create a separate text file or script element that has your program's text in it.

[Additional Course Materials:
Here are some GLSL (the OpenGL Shading Language) resources: [reference site](http://www.opengl.org/documentation/glsl/), [shading language description](http://www.opengl.org/wiki/OpenGL_Shading_Language), [manual pages](http://www.opengl.org/sdk/docs/manglsl/), [data types](http://www.lighthouse3d.com/tutorials/glsl-core-tutorial/glsl-core-tutorial-data-types/), and

[reference guide](http://www.cs.cmu.edu/afs/cs/academic/class/15462-f10/www/lec_slides/glslref.pdf). The official specification is [here](http://www.khronos.org/registry/gles/specs/2.0/GLSL_ES_Specification_1.0.17.pdf).
]

Lesson: Vertex Shader Example

[code again, no quotes]

```
uniform vec3 uMaterialColor;
uniform vec3 uDirLight;

varying vec3 vColor;

void main() {
	// Transform the vertex from model space to clip coordinates
	gl_Position = projectionMatrix * modelViewMatrix * vec4( position, 1.0 );
	vec3 light = normalize( uDirLight );

	// Compute a diffuse color using Lambertian reflection, N * L
	float diffuse = max( dot( normal, light ), 0.0);

	vColor = uMaterialColor * diffuse;
}
```

```
uniform vec3 uMaterialColor;
uniform vec3 uDirLight;

varying vec3 vColor;

void main() {
    // Transform the vertex from model space to clip coordinates
    gl_Position = projectionMatrix * modelViewMatrix * vec4( position, 1.0 );
    vec3 light = normalize( uDirLight );

    // Compute a diffuse color using Lambertian reflection, N * L
    float diffuse = max( dot( normal, light ), 0.0);

    vColor = uMaterialColor * diffuse;
}
```

Here's our example vertex shader without all the quotes and commas. With syntax highlighting on, it's considerably more readable and easier to edit.

The GLSL language is sort of like the C language, which is kind of like JavaScript anyway, but simpler. Because vectors are so important in graphics, the language also has vector support built in.

This vertex shader computes the diffuse, Lambertian lighting component at each vertex.

At the top you can see the material color is defined as a uniform, as something the vertex shader uses as an input. Similarly, the light's direction is passed in as a uniform. "vec3" means a 3 element vector: RGB for the color, XYZ for the light's direction.

Next is a varying color, vColor, which is an output color. We'll use this in the fragment shader.

The vertex shader is required to always output gl_Position, which is a predefined variable. This vector holds the clipping coordinates of the vertex location on the screen.

This next line normalizes the direction to the light. This operation is done here more for convenience of the designer, so that the light direction does not have to be passed in normalized. You could get rid of this line if you required that the light direction must be normalized before being passed to the vertex shader.

The diffuse reflection model is computed on this next line. We take the dot product of the vertex normal and the light. We use a maximum function here to make sure the result is not negative. A negative number happens when the surface is pointing away from the light. We don't allow the light's contribution to be negative - that would be like the light is sucking energy from the surface.

The final line of the program itself computes the output color, called vColor, by multiplying the color by the diffuse contribution. The shader language is a lot like C or JavaScript, but has some nice built-in features. Here we're multiplying a vector, the material color, times a single value. The language knows to multiply each value of the color vector by the diffuse value.

[Additional Course Materials:
See [this posting](https://github.com/mrdoob/three.js/issues/1188#issuecomment-3666286) for what matrices are built-in for shaders in three.js. You can also [look at code](https://github.com/mrdoob/three.js/blob/master/src/renderers/WebGLRenderer.js#L6469) and see the list.
]

Lesson: Fragment Shader Example

The position “gl_Position” and the color “vColor” were generated by the vertex shader. These are passed to the rasterizer, which interpolates these across the triangle and creates fragments. Each fragment is then passed through the fragment shader. You’ll be happy to hear this shader is very simple.

```
varying vec3 vColor;

void main() {
        gl_FragColor = vec4(vColor, 1.0);
}
```


```
varying vec3 vColor;

void main() {
    gl_FragColor = vec4(vColor, 1.0);
}
```


All this shader does is take in the interpolated color and copy it over to gl_FragColor, the built-in output variable for what color appears on the screen. The fourth coordinate is the **alpha value**, which says how solid the surface is.

Notice how the language elegantly understands vectors. vColor is a vector with 3 coordinates. We construct gl_FragColor, a four-element vector, by using vColor and appending a number. GLSL knows what this means. In general the language is very aware of common vector and scalar operations. Here’s an example:

```
vec3 uSpecularColor;
float specular;
...
gl_FragColor.rgb += specular * uSpecularColor;
```

Say we have an input specular color and a specular amount. We’ve computed a value for the “specular” variable, how much it should affect the fragment’s color. In this line of code we multiply this variable, a single floating point number, by a vector with three elements. We add this to the fragment color by specifying which components we want changed. Here the R, G, and B components are modified. The component names for the four elements are

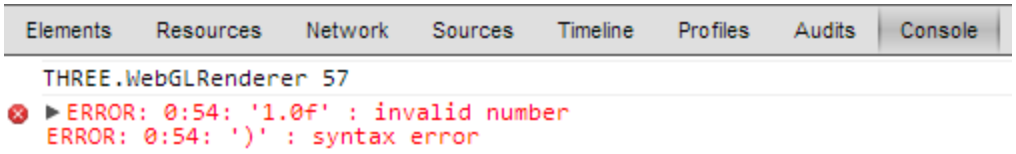
x,y,z,w typically for points and vectors
r,g,b,a typically for colors

s,t,p,q typically for texture coordinates

I won't go into all the features of the language, see the additional course materials for full references. One thing worth mentioning is that there are all sorts of built-in functions. Some you'll probably be familiar with, such as ***abs(), sin(), pow(), sqrt().*** Others are more specific to graphics, such as ***normalize(), dot(), cross(), and reflect().***

```
gl_FragColor = vec4( uAmbientLightColor * uMaterialColor, 1.0f );
```

```
gl_FragColor = vec4( uAmbientLightColor * uMaterialColor,
```



For debugging, the browser's debug console can often give useful errors. For example, for this line of code I've put a representation for a floating point number that's perfectly valid in C but is not a part of the GLSL language.

```
float diffuse = max( dot( normal, lVector ), 0 );
```

Sometimes the errors are a bit cryptic. In this case the "0" should be "0.0" - GLSL is very picky about having a floating point number have a decimal point.

In either case, you'll get a blank screen and not much warning. I sometimes find myself adding a line or two at a time and seeing if everything stays on the screen; if not, I comment out lines until my scene reappears. I call this "binary search debugging".

[Additional Course Materials:
This [nice in-browser GLSL tutorial](http://pixelshaders.com/sample/) lets you change lines and immediately see the result. Larger editor view [here](http://pixelshaders.com/editor/).

If you use the Sublime Text 2 editor, you may want to get [the GLSL syntax

add-on](https://github.com/euler0/sublime-glsl). There are a few, this was the best one I found.

There is also a [GLSL shader validator](https://github.com/WebGLTools/GL-Shader-Validator) I just noticed; I haven't tried it yet.
]

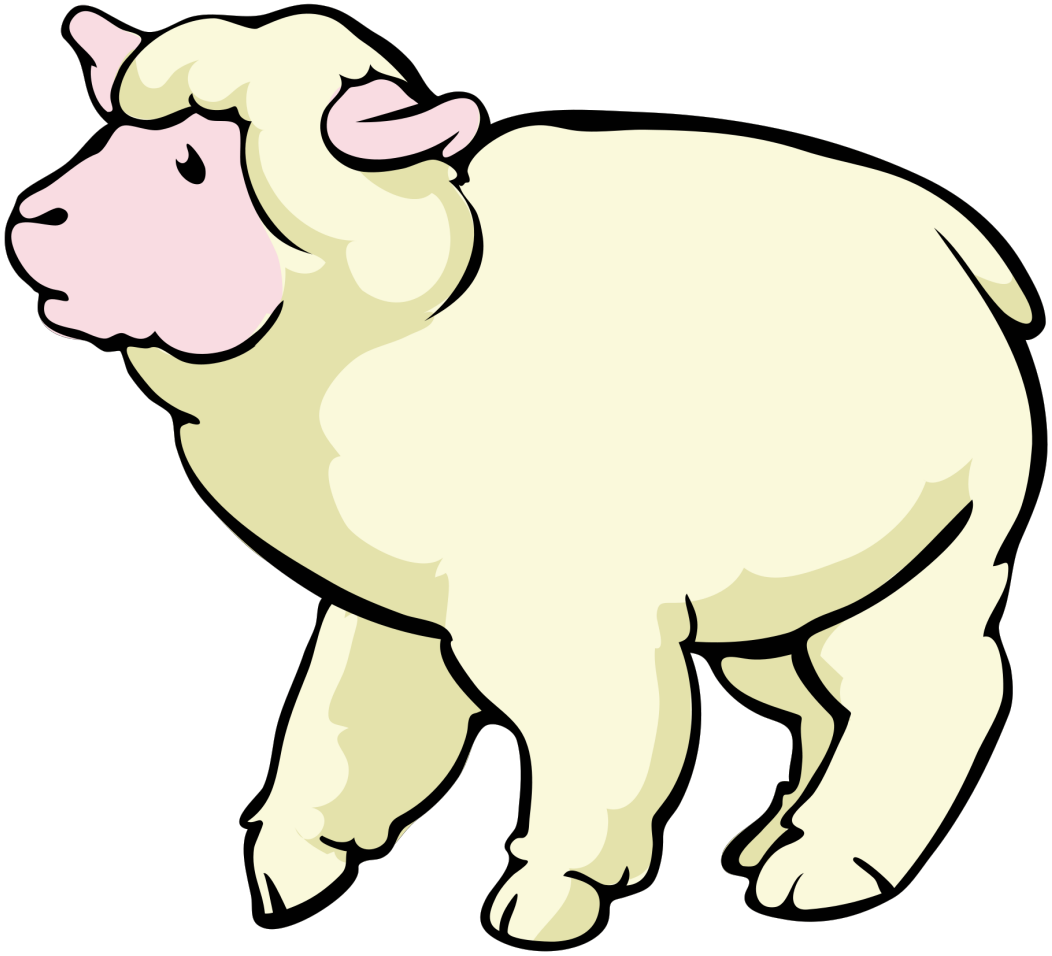
[end recorded 4/7 (more below)]

Lesson: Cartoon Shading

[recorded 4/6]

One common way of giving a cartoon-like appearance to a surface is to give it two or more shades of the same color. For example, in this drawing the sheep's wool has just two tones, light and dark.

[from http://commons.wikimedia.org/wiki/File:Sheep_cartoon_04.svg]



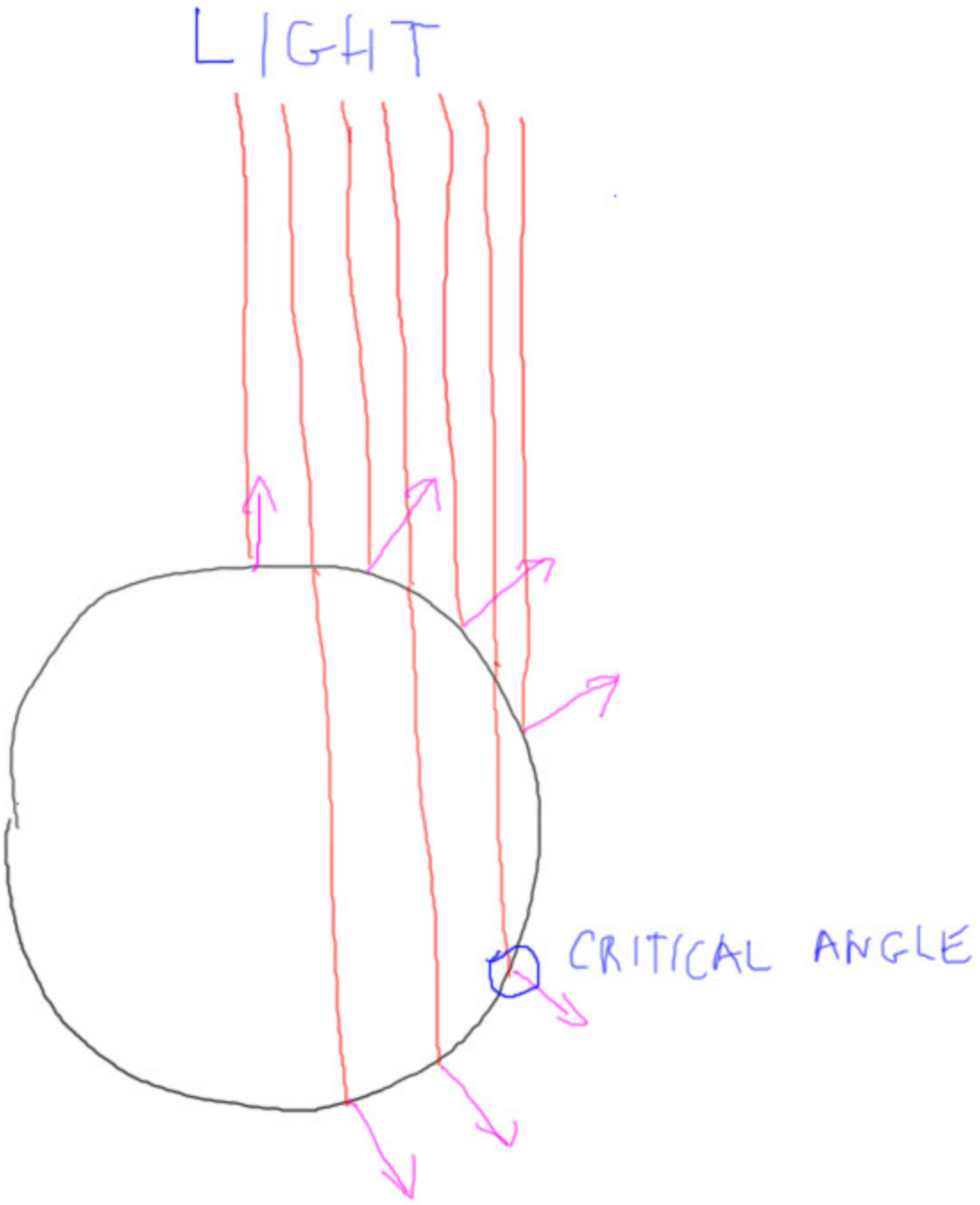
You're going to do the same sort of thing with the diffuse shader we just described. The idea is to modify the diffuse component. In the diffuse shader code, we computed this value by taking the dot product of the normal and the light:

```
float diffuse = max( dot( normal, light ), 0.0);
```

float diffuse = max(dot(normal, light), 0.0);

Instead of using the diffuse component for the shade, we really want to identify two zones, light and dark. If the normal is pointing some fair bit towards the light, consider the area lit, otherwise it's dark.

[Replace sheep with this figure, keep line of code on. ADD a ray that is at 90 degrees, at the light's limit. Sphere below light, showing the points where we go from light to dark.]



Take this sphere as an example. Normally we clamp the diffuse term at 0 and not let it go negative. Above zero and the light has a greater and greater contribution as the dot product increases. This is realistic, but we can do what we want in the shader. One idea to get a two-tone effect is to define a critical angle, essentially a dot product value. Below this value we'll make the surface uniformly dark, otherwise it's fully lit.

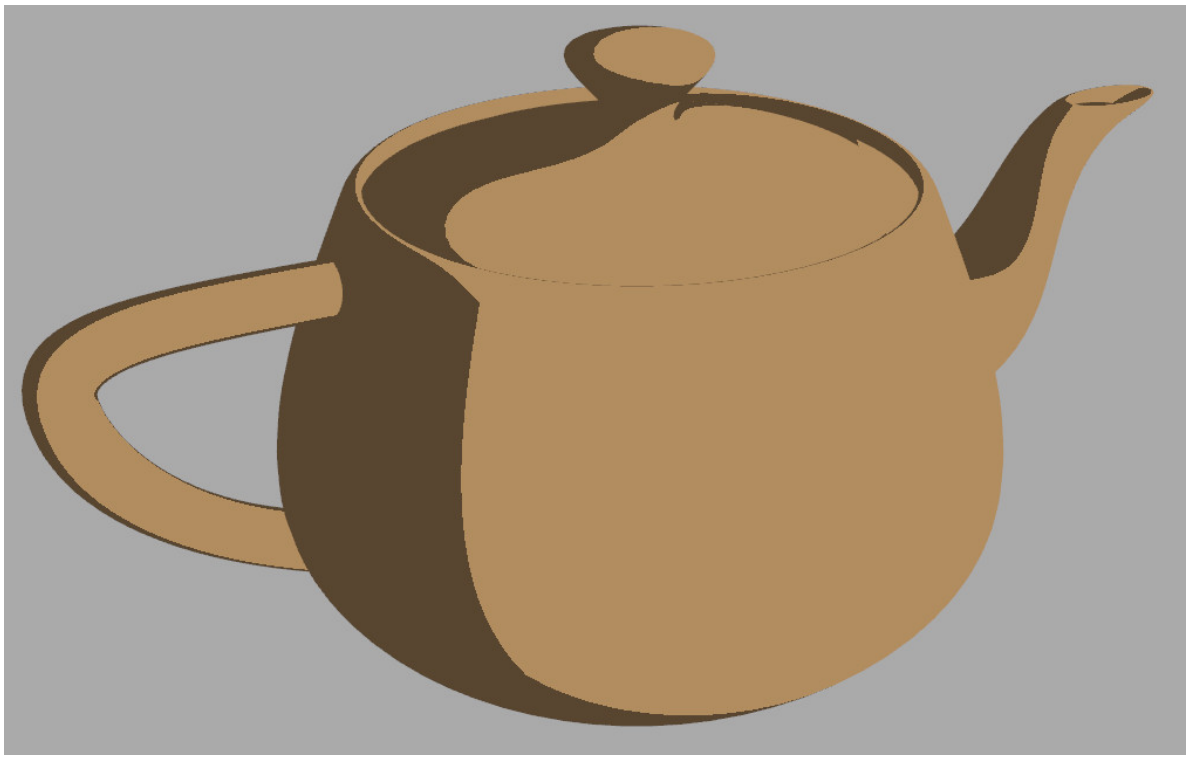
Exercise: Two-Tone Shading

For this exercise you'll start with a fragment shader that performs diffuse lighting per pixel. I've hooked up a uniform called uBorder to the user interface. Change the program to instead compute the diffuse component as follows:

if the dot product is greater than uBorder,
then diffuse is 1.0
else diffuse is 0.5

Make sure to use floating point numbers, and don't leave off the "point 0" for 1.0.

[show and slide "border" UI toon_solution.html]



When you get this new code in place, it should look like this. Once you have the solution, try the "border" slider and see the effect. If nothing happens for negative values of "border", you might want to think about the effect of the "max()" function.

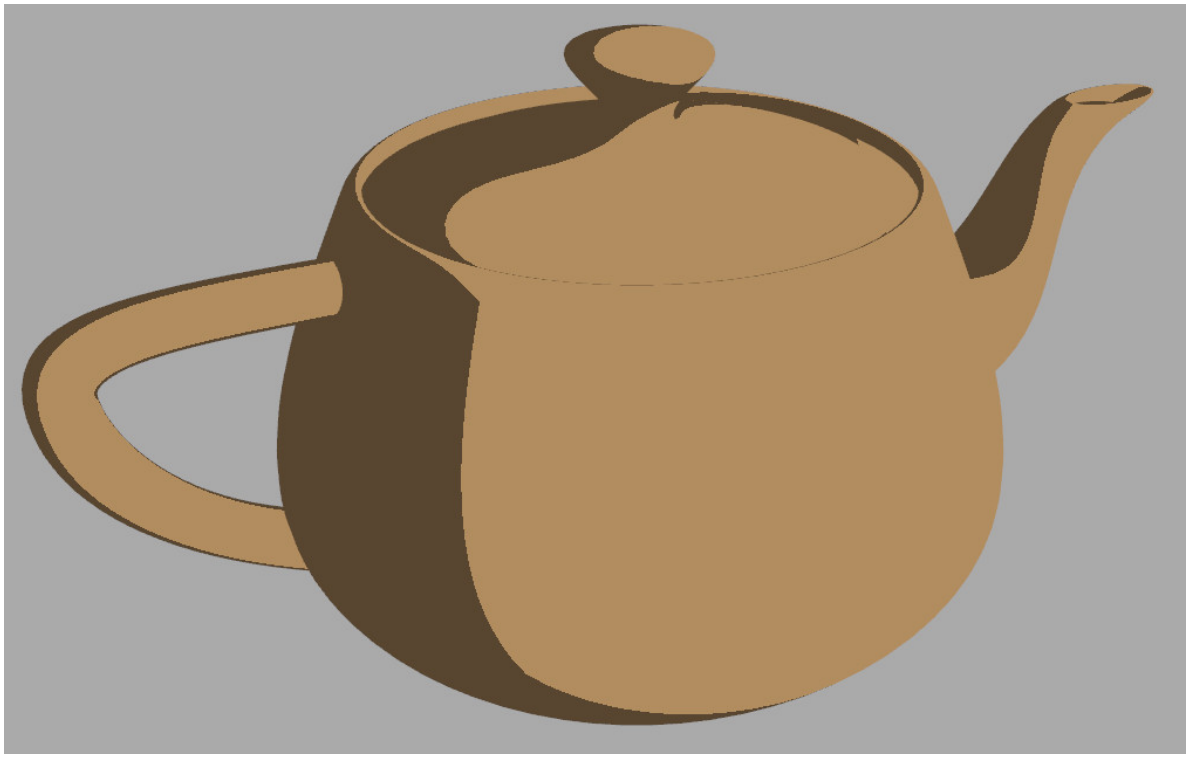
[exercise is here: toon_exercise.html]

[Instructor Comments:]

Answer

[solution is here: toon_solution.html]

[Is there room to add this picture? put code to right? Note there are 2 pieces of code... AHA, put code first, image after if space.]



```
//was: float diffuse = max( dot( normal, lVector ), 0.0);
float diffuse = dot( normal, lVector );
if ( diffuse > uBorder ) { diffuse = 1.0; }
else { diffuse = 0.5; }
```


```
//was: float diffuse = max( dot( normal, lVector ), 0.0);
float diffuse = dot( normal, lVector );
if ( diffuse > uBorder ) { diffuse = 1.0; }
else { diffuse = 0.5; }
```


Here's one way to code the answer.

We still compute the diffuse component, but no longer take a maximum value, since we're going to compare it to uBorder. The comparison here then changes the diffuse value so it gives a cartoony effect.

```
float diffuse = dot( normal, lVector );
diffuse = ( diffuse > uBorder ) ? 1.0 : 0.5;
```

A more compact solution is this one, using a conditional operator instead of a full-blown if/then/else statement.

---------- new page ----------

[put code and new 3 tone image]

This rule is entirely arbitrary, of course. You could even try more elaborate rules. For example, this code:

```
float diffuse = dot( normal, lVector );
if ( diffuse > 0.6 ) { diffuse = 1.0; }
else if ( diffuse > -0.2 ) { diffuse = 0.7; }
else { diffuse = 0.3; }
```

gives three levels of illumination.

[Show this image as part of new page: three_tone.jpg - NOTE: KEEP IMAGE ON SCREEN FOR NEXT LESSON]

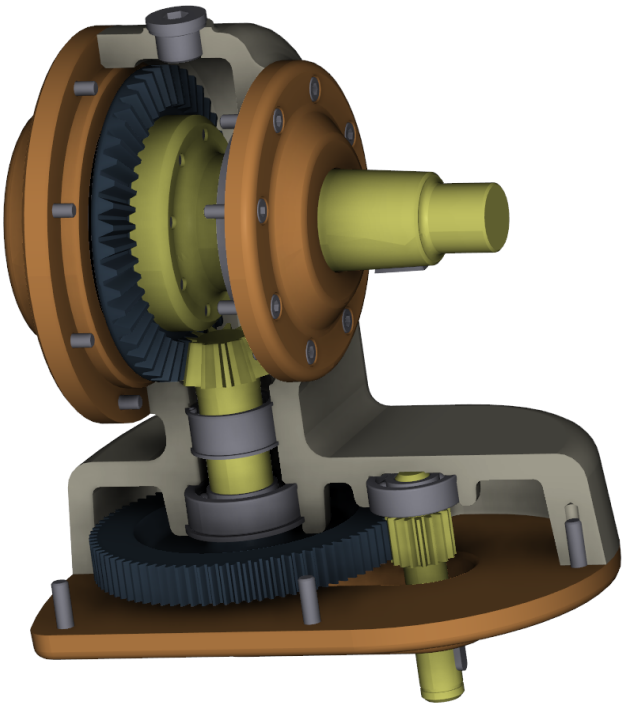
It gives an image with a different feel, a bit more depth and a little less cartoony. This type of effect was almost impossible with the old fixed function pipeline. With shaders it's easy to experiment with different illumination models.

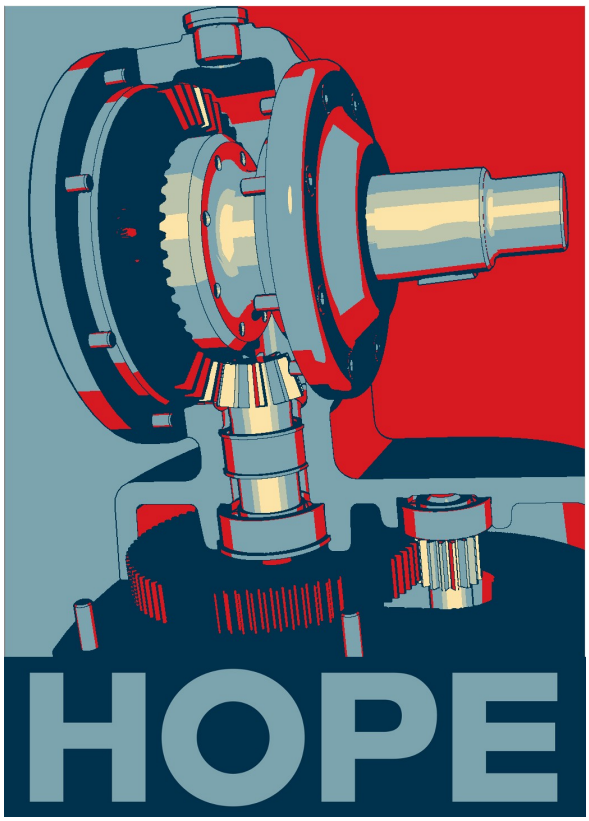
Lesson: Non-Photorealistic Rendering

[keep image in place, add terms to left]

This kind of reflection model is called **toon shading** or **cel shading,** or sometimes **posterization**. “Toon” is short for cartoon. “Cel” is short for celluloid, the material for the transparent sheets that are used when animation is drawn by hand.

[hope.jpg]





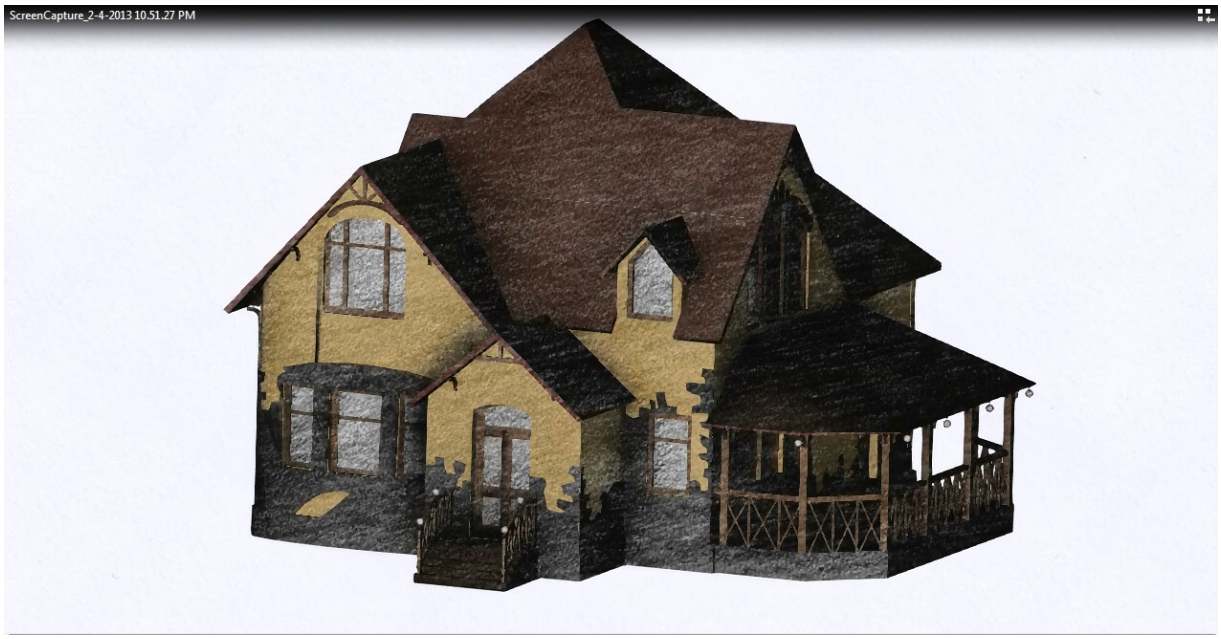
[put “**non-photorealistic rendering**” or “**NPR**” to right of image]

There are all sorts of variations for this type of rendering. For example, you can change colors based on shading level and get more elaborate effects.

In this rendering, a separate image processing shader is used to detect the edges of the object and outline them.

This type of illumination model is one example of what is called “**non-photorealistic rendering**” or “**NPR**” for short. “Photorealistic” means “looks like a photo”.

[video: Unit8_NPR, NPRsandboxHouse.wmv - just let it run]



If you think about it, there's a nearly infinite number of ways to draw things that do not look realistic. The running joke is that we've been doing non-photorealistic rendering on computers for decades, since realism is hard to achieve.

The way I see it is that Non-photorealistic rendering has an intent behind it. Just as we assign different fonts to text to convey a different impression, NPR is a choice we can make about 3D rendering.

Scott McCloud, in his book "Understanding Comics", talks about "amplification through simplification". The idea is that by displaying only the relevant pieces of information you strengthen your message. For example, an automobile owner manual usually has simplified drawings instead of photos.

Non-photorealistic techniques often try to convey information in a more sketchy or artistic way. For example, you might render an object so that it looks like it's hand-drawn or painted. Beyond adding visual interest, using a style like this can give a different impression. For example, a sketchy look can let the viewer know that the result they're looking at is just in an early stage of design.

[Instructor Comments:
The 3 tone solution is
[here](http://www.realtimerendering.com/erich/udacity/exercises/unit3_toon_solution3.html) - the border variable is not hooked up. Feel free to grab this code and modify it.

You might want to try [this

demo](http://mrdoob.github.com/three.js/examples/webgl_marching_cubes.html) again. For the first few materials you should have a pretty good sense of how these work, and can pick out what sorts of artifacts you might see. To see a few other NPR techniques in action, try out the various materials from “toon1” on down.

A nice runthrough of NPR for architects, with lots of examples, is shown in [this promotion video for “Impression”](http://download.autodesk.com/us/impression/2009overview/index.html).

NPR has its own conference, search [Kesen’s site](http://kesen.realtimerendering.com/) for “NPAR”. Older links to NPR resources can be found [here](http://www.realtimerendering.com/#npr).

I highly recommend the book [“Understanding Comics”](http://www.amazon.com/Understanding-Comics-The-Invisible-Art/dp/006097625X?tag=realtimerenderin) for a thoughtful and engaging look at how comics and visual simplification works.
]

Question: NPR per Material

The cel shader we developed can fail in various ways. Which of these cases causes the most problems?

() Texture mapping
() Reflection mapping
() Shadows
() Transparency

Answer

Transparency is the main point of failure. The other methods change the illumination on a single surface,. The final value is then mapped in some fashion to a color. With transparency you’re first mapping the opaque material to a color, then you’re blending this color with whatever transparent filter is covering a pixel, then mapping *that* result to some new color. It’s mixing results from shaded pixels with solid colors, which is unlikely to work well.

We’ll see in a later lesson that image processing is often a better way to perform cel shading, as this maps from some final shaded color to a solid value just once.

Lesson: Vertex Shader Programming

```
varying vec3 vNormal;
varying vec3 vViewPosition;

uniform float uSphereRadius2;        // squared

void main() {

        gl_Position = projectionMatrix * modelViewMatrix * vec4( position, 1.0 );
        vNormal = normalize( normalMatrix * normal );
        vec4 mvPosition = modelViewMatrix * vec4( position, 1.0 );
        vViewPosition = -mvPosition.xyz;
}
```

```
varying vec3 vNormal;
varying vec3 vViewPosition;

uniform float uSphereRadius2;   // squared

void main() {

    gl_Position = projectionMatrix * modelViewMatrix * vec4( position, 1.0 );
    vNormal = normalize( normalMatrix * normal );
    vec4 mvPosition = modelViewMatrix * vec4( position, 1.0 );
    vViewPosition = -mvPosition.xyz;
}
```

I find myself usually modifying the *fragment* shader in a program, since that's where all the per-pixel processing is happening. Let's back up a bit and show how various parameters were produced by the *vertex* shader.

[underline these in code itself]

The position and normal of the vertex are passed in with these names, ***position*** and ***normal***.

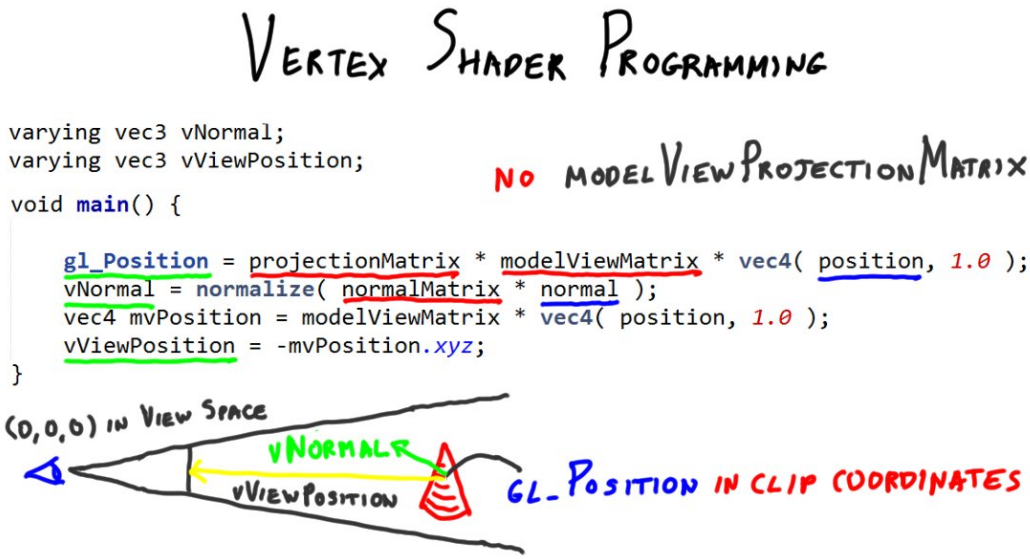
[underline these in code itself, different color]

A few built-in matrices are used, namely ***projectionMatrix***, ***modelViewMatrix***, and ***normalMatrix***. In three.js these are always available to the shader if desired; in WebGL itself

you need to do a little more work. There's currently not a ***modelViewProjectionMatrix [draw on screen]*** of these two matrices multiplied together - maybe you'll be the person to add it to three.js, as it's commonly used and would be more efficient to use here.

What we get out of this shader are a few vectors.

[draw object in a frustum side view, give a point, label gl_Position.]



First, the gl_Position is set, which is the location in clip coordinates. This vector must always be set by the vertex shader, at a minimum. One of the features of the vertex shader is that you *can* change the shape of the object; you can't really change it in the fragment shader.

[draw normal on object and label]

The normal in model-view space is computed here, using the normal transform matrix.

[draw vector from object to camera]

Finally, a vector from the location in model-view space towards the viewer is computed. First the position in model-view space is computed. Negating this vector gives the direction toward the viewer from the surface, instead of from viewer to object. Remember that the camera is at the origin in view space.

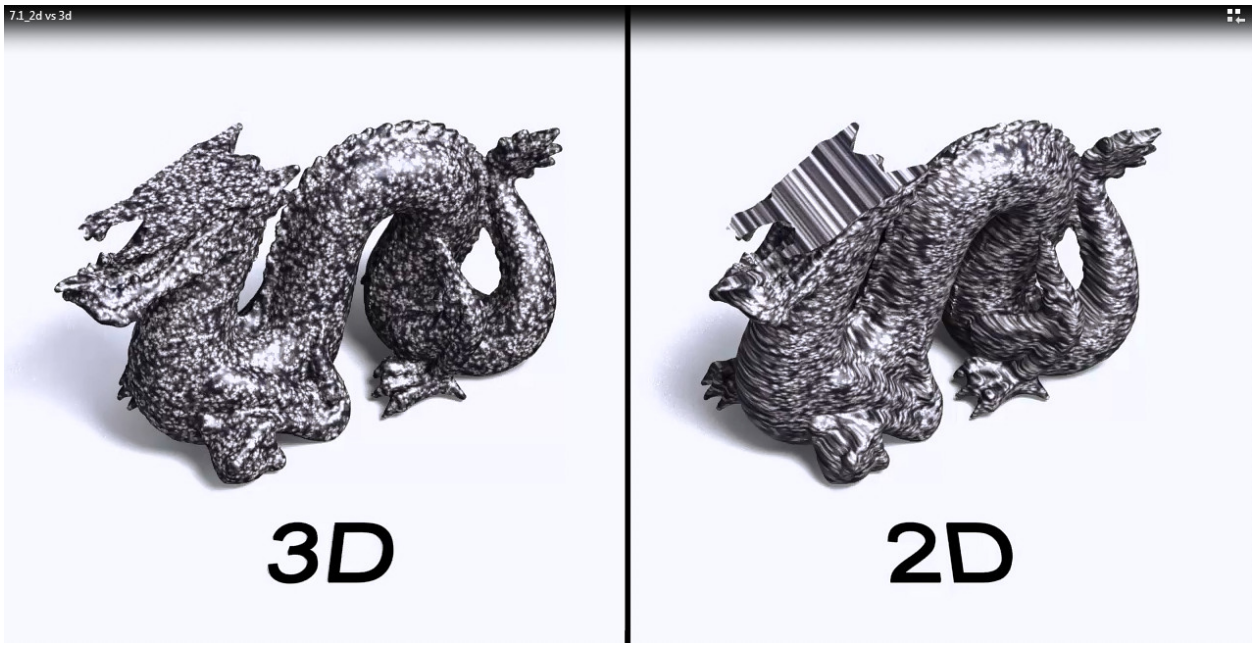
We don't really need the temporary mvPosition vector, we could have combined these last two lines of code. This example is here to show how to compute it.

[OPTIONAL, space dependent and flow dependent: draw triangle, show normal and position as arrow and dot for each vertex. Note interpolation.]

To sum up, the vertex shader took as inputs the model-space position and normal. It transformed them to create a point in clip coordinates for the rasterizer. It also transformed the normal and position. The resulting transformed vertices are then interpolated across the triangle during rasterization and sent to the fragment shader for each fragment produced.

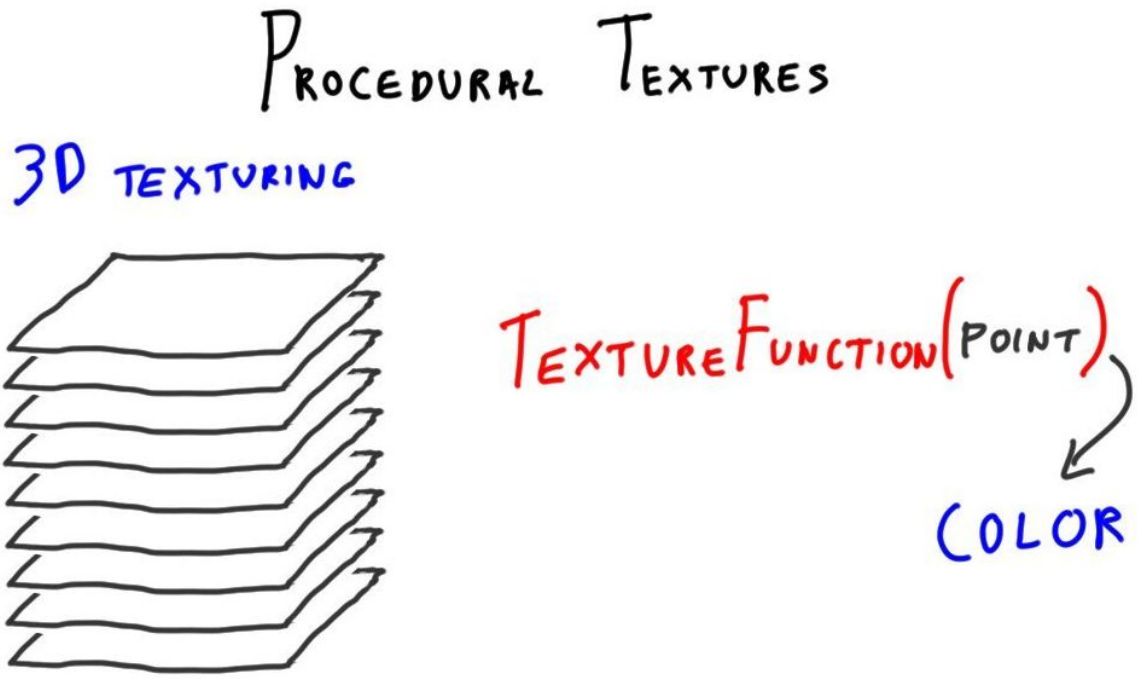
Lesson: Procedural Textures

[video Unit8_ProceduralTexturing, 7.1_2d vs 3d.mp4 - 3D vs. 2D - just repeat video until I'm done talking]



We've shown how an image texture can be applied to a model. However, these sorts of mappings are only skin deep. If we carve away any part of the object, the fact that a 2D texture is getting used is revealed. What we'd like is something that's more like a real material. The solution is to create a 3D texture that fills space.

[***3D texturing***:
draw stack of images]

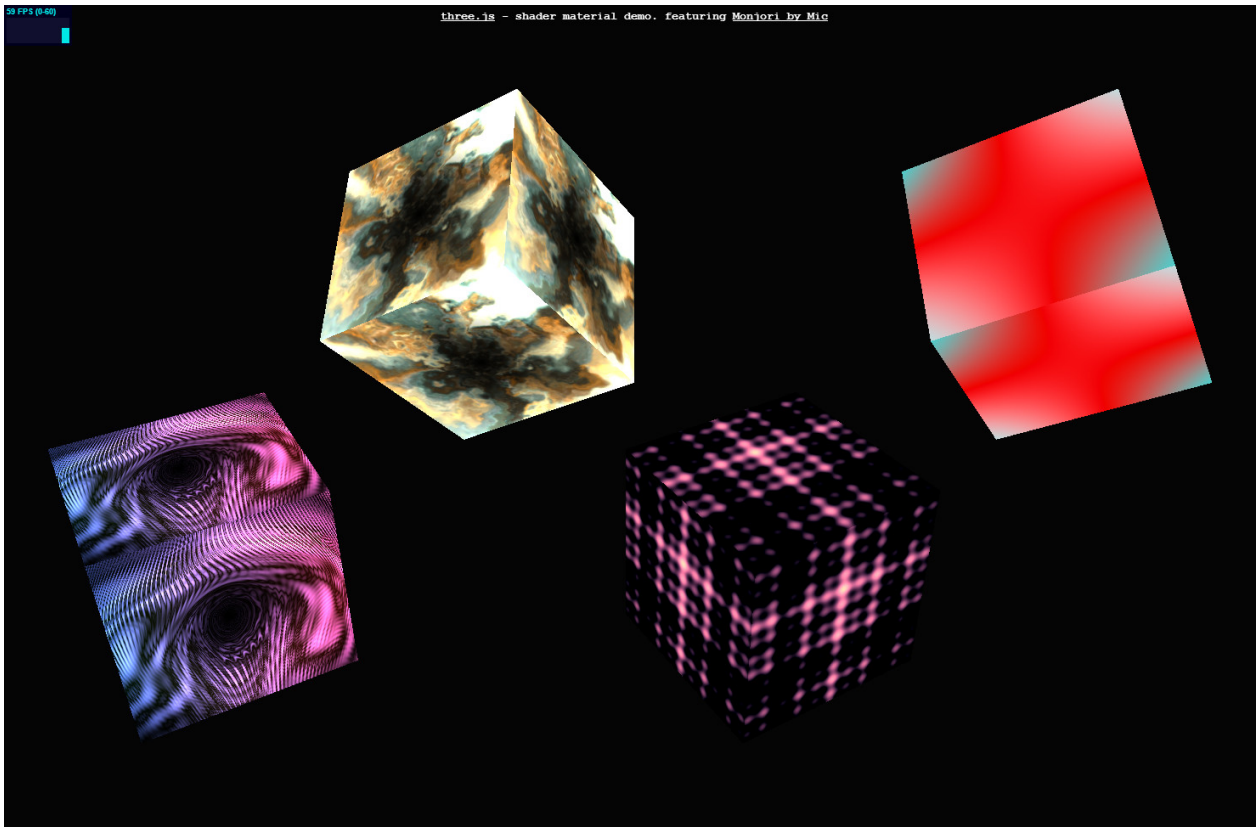


One way to define this texture is to actually create something like a stack of images. Unfortunately, 3D textures are not yet available in WebGL. My guess is, wait a year.

[***texture(point) --> color***]

Another method is to make a function that takes in a location on a surface and gives back a color. If you think about it, these are equivalent in intent. Using images also takes in a 3D point and gives back a color. Each approach has its strengths and weaknesses: functions have unlimited resolution compared to textures, but you also have to be careful to filter their results.

[net http://mrdoob.github.com/three.js/examples/webgl_shader2.html]



While I'm at it, I should mention you can shade points on surfaces however you want using a fragment shader program. Here's an example of animated procedural textures in use. These examples are 2D functions, they don't fill space, but show how elaborate effects can be computed on the fly in the fragment shader.

[Additional Course Materials:
Here's [another beautiful example program](http://mrdoob.github.com/three.js/examples/webgl_shader_lava.html), done by overlaying textures.
]

[end recording 4/6]

Exercise: 3D Procedural Texturing

[recorded 4/8]

Let's take the teapot shader and vary its color with location. Here's what the function should do.

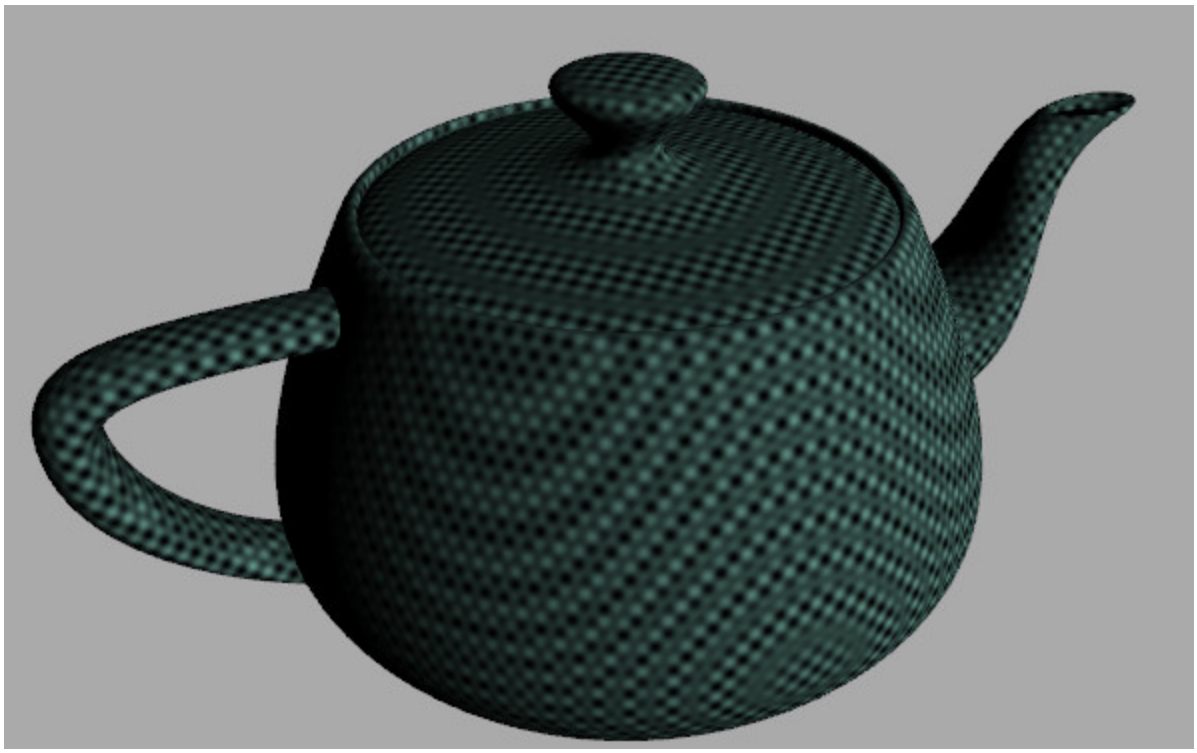
multiply diffuse by:

1/2 + 1/2 * sin(uScale * x) * sin(uScale * y) * sin(uScale * z)

Use vModelPosition to give the x, y, and z values.

The function here makes sine waves through space in three directions. The one-halfs in the equation make the result be in the range 0 to 1.

[run and set slider all the way down, then up
procedural_texture_solution.html
]



When you're done, the result will look like this. There is a slider to change the overall scale of the function.

[Additional Course Materials:
The classic book ["Texturing and Modeling: A Procedural Approach"](http://www.amazon.com/Texturing-Modeling-Third-Procedural-Approach/dp/1558608486?tag=realtimerenderin) is still quite relevant, despite its age. If you delve into procedural textures at all, this is the first book to get.
]

[Idea for these functions came from Steve Worley. Extra notes from Steve, I'm leaving these in because he's has decades of experience:

Then you can make things more complex by starting to sum multiple sine waves at different scales in a for loop to build a poor man's Perlin-noise like random function. (which would segue nicely to Perlin noise itself as a future project.)

Sine waves also allow filtering (also a segue to a future project.)

Anyway, the nice part of sine waves is just that they start simple, they are well behaved, you can incrementally build more complex appearances from them, and they lead onto "real" texturing.

Also: you can take derivatives more easily if you want to get into bump mapping. Bump mapping always looks cooler than simple color maps!

Also: Don't forget time varying effects. sin(t) can be added to make things pulse and throb and vary. Again smoothly, using the same ideas, etc.

Now the other suggestion is that related but more dramatic and useful effect that's of similar construction is not a procedural texture, but a procedural space distortion. Add a procedural offset or varying bias to a COORDINATE that's then used for a texture map lookup or another procedural texture. Think of an ocean surface bitmap that gets slowly repetitively perturbed... that looks a LOT more like water than a static image!

Or you can twist coordinates for fun (stuff like convert to polar coordinates, compute thetaPrime=theta+1/(R+1) then back to rectangular coords.)

It's a big advantage to apply these effects to an image map since there's lots of detail in those to show off, so you get more "bang for your buck."

Also, in practice, think of all the tricks the games people play with such distortions. Smoke maps on partially transparent impostor polygons is a typical example. Procedurally modify the smoke coordinates and you get free billowing and such. People make pretty convincing fire from this trick alone (and a lot of tuning and experience, but it's still just blobs mapped on transparent polys, with the "procedural" element distorting them minorly, another procedural fading from yellow glow to red fire to grey smoke to black haze, with opacity also fading over time so the smoke can go away (and impostor recycled).

And these also lead to vertex effects too, to make your whole teapot shudder. or make A head distort.. ALWAYS fun for the kids. (Best, you may be surprised, is a high quality head with a TINY almost unnoticeable distortion. Our eyes are so good at feeling something's wrong, but we don't know what!)

Anyway, just some ideas off the top of my head.
]

Answer

```
float diffuse = max( dot( normal, lVector ), 0.0);

// Student: use the vModelPosition as an input to a function
diffuse *= ( 0.5 +
        0.5 * sin( uScale * vModelPosition.x ) *
                sin( uScale * vModelPosition.y ) *
                sin( uScale * vModelPosition.z ) );

gl_FragColor = vec4( uKd * uMaterialColor * uDirLightColor * diffuse, 1.0 );
```

```
float diffuse = max( dot( normal, lVector ), 0.0);

// Student: use the vModelPosition as an input to a function
diffuse *= ( 0.5 +
    0.5 * sin( uScale * vModelPosition.x ) *
          sin( uScale * vModelPosition.y ) *
          sin( uScale * vModelPosition.z ) );

gl_FragColor = vec4( uKd * uMaterialColor * uDirLightColor * diffuse, 1.0 );
```

Here's one solution. It directly multiplies the diffuse component by the function. I highly recommend trying out any other functions you want here. The main thing is to aim for functions that give numbers back in the 0 to 1 range. The numbers can be outside this range - smoke won't come out of your computer. All that happens is that the value gets clamped to 0 or 1.

Exercise: Debugging Shaders

```
gl_FragColor = vec4( uKd * uMaterialColor * uDirLightColor * diffuse, 1.0 );
```

```
gl_FragColor = vec4( uKd * uMaterialColor * uDirLightColor * diffuse, 1.0 );
```

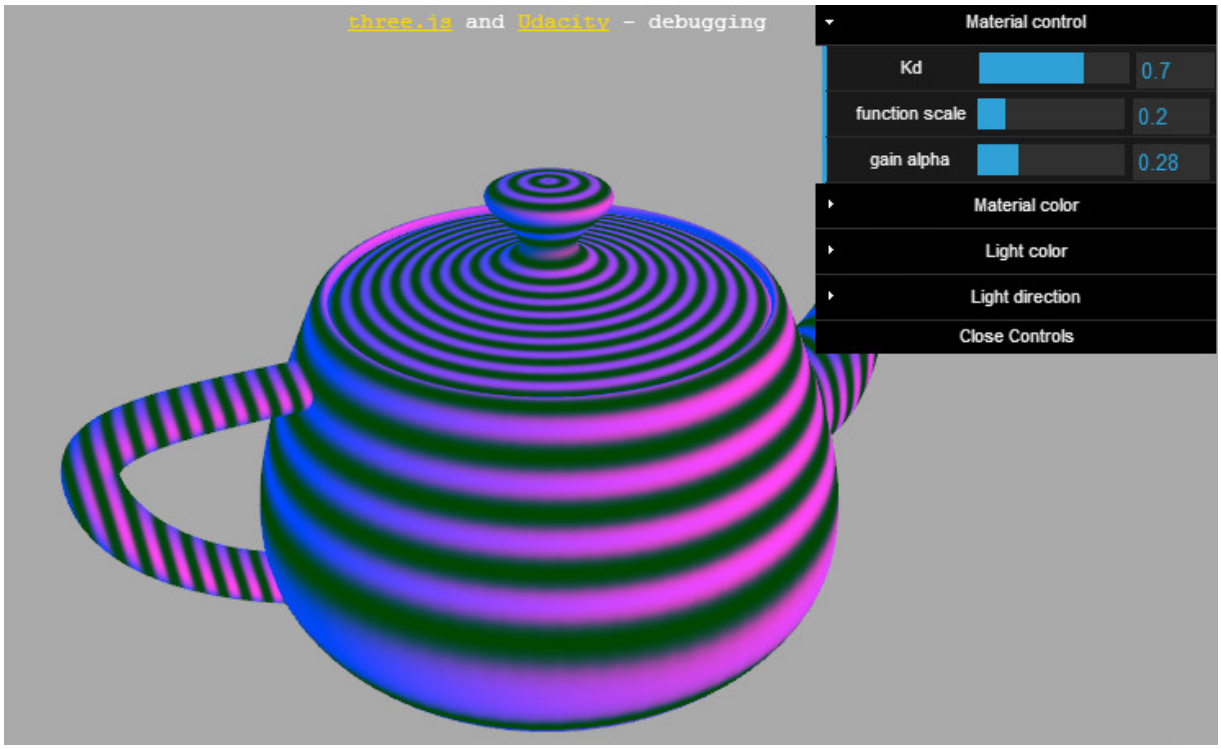
In this exercise you'll start with a different procedural texturer. It works fine, but I want you to know about an important debugging technique. Since you can't put a breakpoint inside a shader, it's hard to know what's going on. The answer is to send the various shader computations to the screen itself.

red channel: diffuse
green channel: uGainAlpha
blue channel: attenuation

In other words, instead of putting your final result into gl_FragColor, put these other intermediate values. In fact, you have three whole color channels to use and view on the screen.
Your task is to send the following program values to the screen [read off the values above].

All three of these values is in the proper range of 0 to 1, so need no adjustment.

[show this, rotate for 3 seconds then do gain alpha slider: debugging_solution.html]



Here's what things will look like when you're done. Under "material control" in the user interface is a slider called "gain alpha". Sliding it will change the green channel to match the value selected.

One bit to notice in the code is that there is a subroutine called computeBias. It's fine to use subroutines inside shaders, and they have a C-language type of interface.

[Additional Course Materials:
There's a bit on using [Chrome's Canvas inspection for debugging WebGL itself](http://learningthreejs.com/blog/2013/04/05/debugging-with-chromes-canvas-inspection/).

For image examination and capture I recommend the (not free, but worth it) [FastStone Capture](http://www.faststone.org/FSCaptureDetail.htm).
]

Answer

```
gl_FragColor = vec4( diffuse, uGainAlpha, attenuation, 1.0 );
```

```
gl_FragColor = vec4( diffuse, uGainAlpha, attenuation, 1.0 );
```

You replace the last line of the fragment shader with this line of code. The order of values is red, green, and blue.

```
gl_FragColor.r = diffuse;
gl_FragColor.g = uGainAlpha;
gl_FragColor.b = attenuation;
gl_FragColor.a = 1.0;
```

```
gl_FragColor.r = diffuse;
gl_FragColor.g = uGainAlpha;
gl_FragColor.b = attenuation;
gl_FragColor.a = 1.0;
```

An alternate solution is to set each of these individually. Either solution is fine - you could even use X, Y, Z, and W for the coordinate names, the shading language interpreter doesn't care.

I wanted to show this colorful, multivariable way of debugging, and it's sometimes handy for seeing correlations between parameters. In practice, though, it's usually clearer to copy just a single value to all three color channels.

Lesson: Fresnel Reflectance

[note: ***Augustin-Jean Fresnel,*** ***French engineer and physicist,*** ***1788–1827***]

One physical phenomenon that is sometimes factored into the illumination model is described by the Fresnel equations. These equations were discovered by the French engineer and physicist ***Augustin-Jean Fresnel***. He received little recognition of his work in optics while he was alive, much of it was published after his death in 1827. Happily, what evidently gave him the most pleasure was the discovery and experimental confirmation of a scientific theory.

The Fresnel equations themselves have to do with the amount of reflection and refraction off a surface.

[show a plane and the eye low to it, shallow angle, highly reflective; show transmission too]

The basic relationship is this: the more on-edge you look at a surface, the more reflective it is. If the surface is refractive, in other words transparent, the amount of light transmitted will drop considerably as you approach this shallow angle.

[draw a lake? Or at least an observer, and looking & seeing a fish vs. on angle. Make surface

blue, person gray, fish red]

My favorite example is a lake or the ocean. The next time you're in shallow water, notice that you can look straight down into it and see the bottom. As you look out toward the horizon, the angle of your eye to the surface's normal approaches 90 degrees and reflection will dominate.

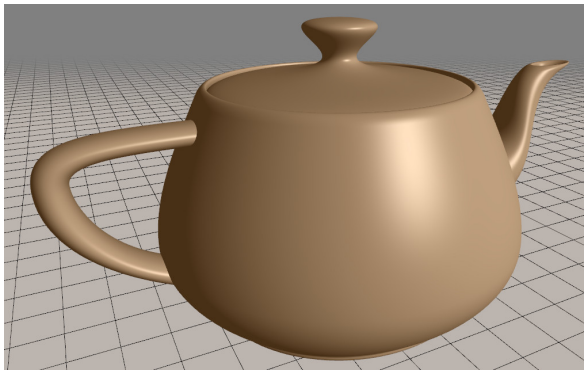
For materials that aren't transparent this effect still takes place. The Fresnel effect is particularly noticeable for what are called "dielectrics", which is a fancy way of saying "insulators", in other words materials such as glass, plastic, and clay. For example, glass is about 25 times as reflective at a shallow angle than directly head on.

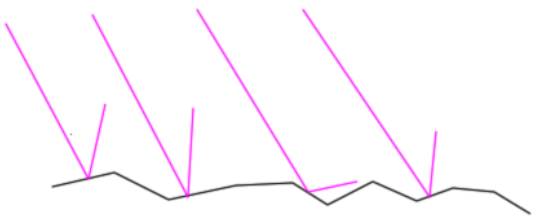
---- new page ----

[Blank screen except for title. Take a piece of paper and a SMALL book, show it edge on]

An experiment you can do right now to prove this effect to yourself: take a sheet of normal paper - nothing very shiny nor very rough - and put it on a flat surface, such as a book. Hold the paper so that you're looking at it nearly edge-on, and look towards a lit area, such as a window. At nearly 90 degrees to the surface normal you'll see the paper become quite reflective! In fact, the Fresnel effect should also lessen the diffuse contribution - as the object becomes more reflective in real life, there are less photons left to go off in other directions.

[Pull in reflective teapot image; draw flat surface below and normal. Pull in microfacets image on to right, and a specular teapot. Compose whole slide first - may need a new page?]





There is one implementation detail I want to mention: you should use the normal that represents the mirror orientation. For example, for a mirror reflective surface, this is simply the shading normal; for most other materials, it's not.

With the Blinn-Phong illumination equation, the idea is that the material is made of microfacets.

[- add an eye location for one of the microfacets and extend the reflection ray to the eye, and add the half-angle]

Only a few facets are angled just right to reflect light towards the eye. These facets have normal vectors pointing exactly halfway between the light and eye vectors. Recall that this special direction is the "half-angle vector". Happily this is a vector we compute anyway for specular highlighting, but it's important to use this direction and not the shading normal.

[cut: For interactive rendering it's common to approximate the full Fresnel equation with the ***Schlick approximation***.]

Try the demo to get a sense of the effect. Notice how a nearly edge-on angle causes the reflectivity to increase.

[Additional Course Materials:
See the Wikipedia article on the [Fresnel equations](http://en.wikipedia.org/wiki/Fresnel_equation) for more background. A more in-depth article](https://seblagarde.wordpress.com/2013/04/29/memo-on-fresnel-equations/) by Sébastien Lagarde is worth the effort. [Our book](http://www.amazon.com/Real-Time-Rendering-Tomas-MOller/dp/1568814240?tag=realtimerenderin) also covers this topic.

The [Schlick approximation](http://en.wikipedia.org/wiki/Schlick%27s_approximation) to the Fresnel term is often used in interactive rendering, since it's quick to compute.

One problem you'll see in the Fresnel demo that follows is that at some angles the specular highlight simply disappears. Read more about this problem in [this article](http://www.geekshavefeelings.com/x/wp-content/uploads/2010/03/Its-Really-Not-a-Rendering-Bug-You-see....pdf), which I've pointed at many times so far.
]

Demo: Fresnel Demo

[fresnel_demo.html]

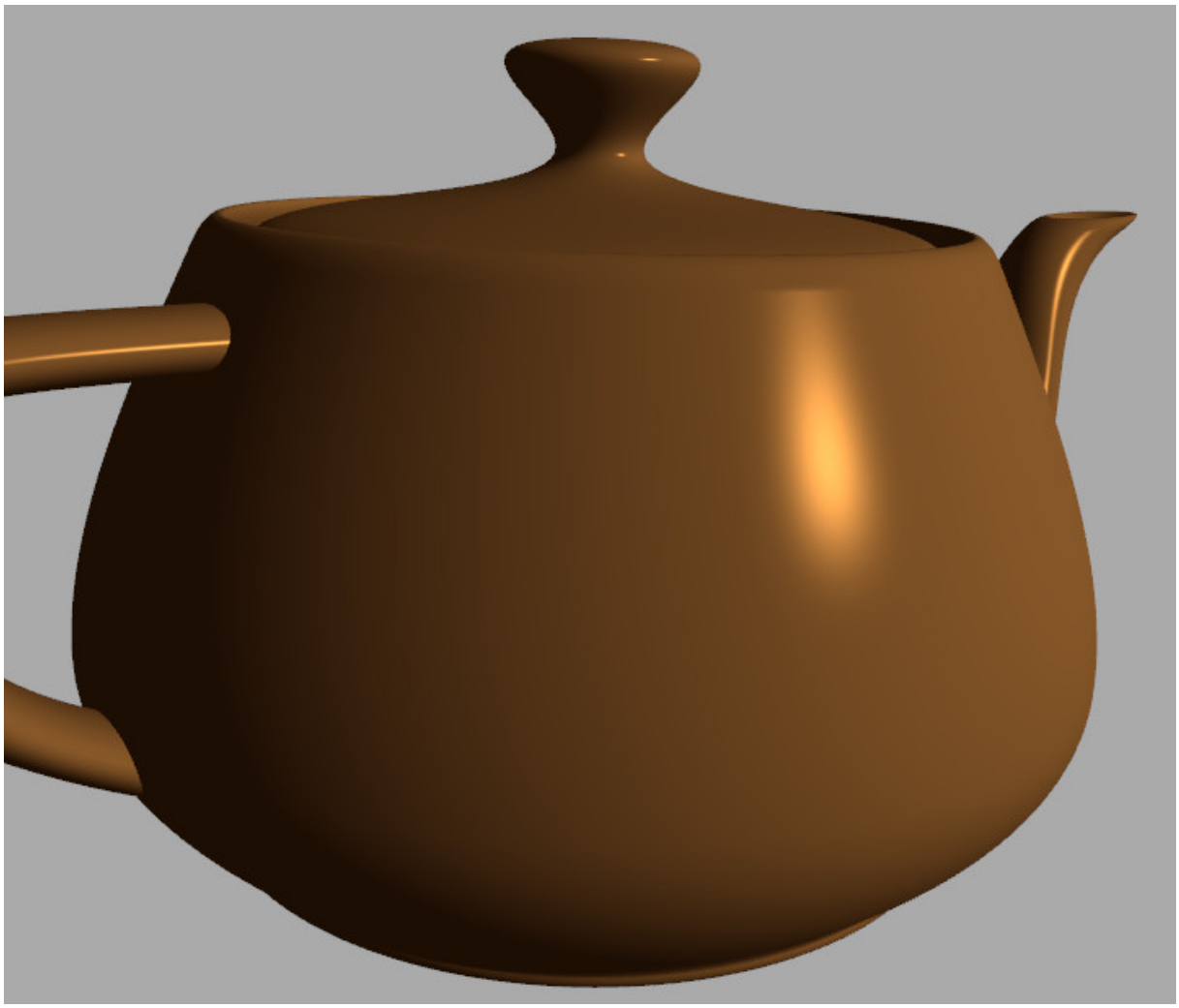
Lesson: Energy-Balanced Materials

[teapot demo.]

The Blinn-Phong reflection model has been around for more than 30 years. It used to be hard-coded into older GPUs from the early 2000's. It's easy to evaluate and somewhat intuitive to control.

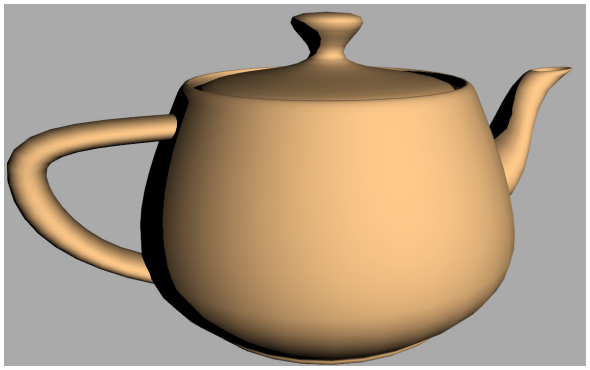
One technique worth mentioning with this reflection model is that you can give the specular highlight a different color than the diffuse component. For example, if the specular component is given a white color, the object looks more like a shiny plastic. If both the specular and diffuse components are multiplied by the same color, the material looks more metallic.

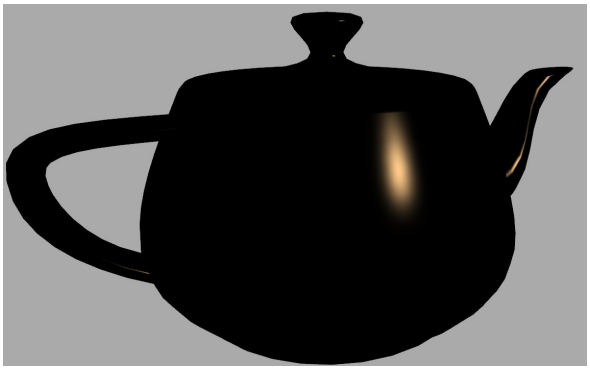
[Show demo unit3_teapot_demo.html , turning metallic on and off. Good settings are Ka 0.2, Kd 0.45, Ks 0.85]



However, this classic reflection model is not energy balanced. You'll notice as I change the shininess that the material looks smoother, but overall the amount of light reflected becomes less.

[shininess with specular-only on - show effect of increasing shininess.]



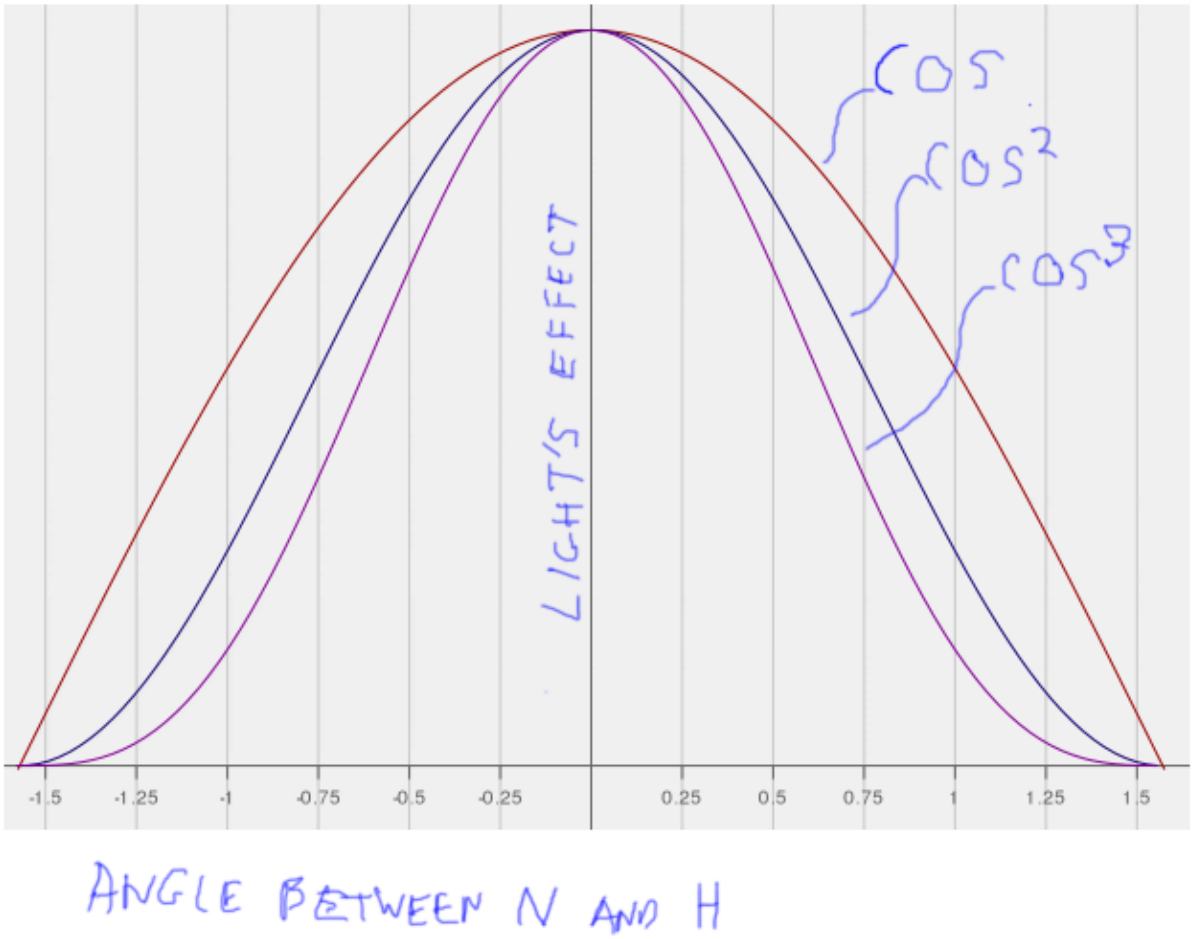


If you think about it, this makes a lot of sense. Here's the Blinn-Phong equation again:

Blinn-Phong reflection model
Specular = max(N * H, 0) ^S

Let's plot how shininess affects things. The graph of the angle between N and H vs. the specular intensity is like this:

[graph]



Clearly the area under the graph for cosine squared is smaller than for cosine, so the amount of energy coming from the surface will be less as you increase the shininess. Cosine cubed has even less overall energy. As the shininess goes up, the area under the curve goes down.

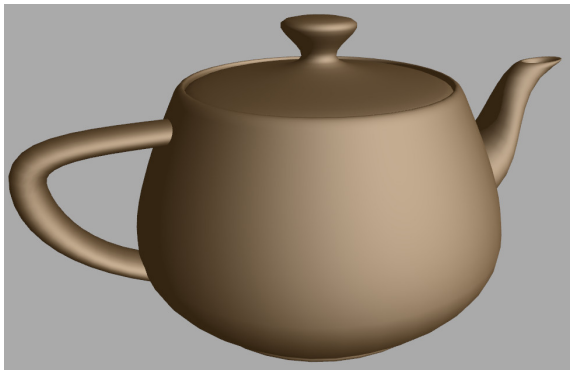
Two changes give a better result, one that is both more plausible and easier to control. One idea is to attenuate the specular term by the Lambertian falloff. In other words, just like diffuse, make the specular term drop off using the angle of the light to the surface, N*L. The other idea is to

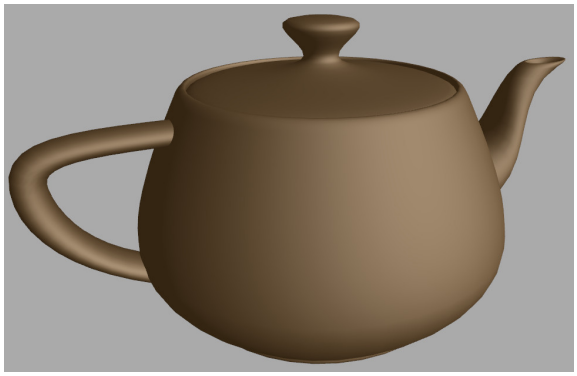
make these narrower curves be higher, giving them roughly the same volume in 3D.
[Don’t forget the vector arrows!]

Specular = (N * L) * max(N * H, 0) ^S * (S + 2) / 8

This idea is captured in this last term. As shininess increases, this last term increases. When combined with the Lambertian term this new equation gives a reasonably balanced result.

Original model
highlight power 3 ***100***

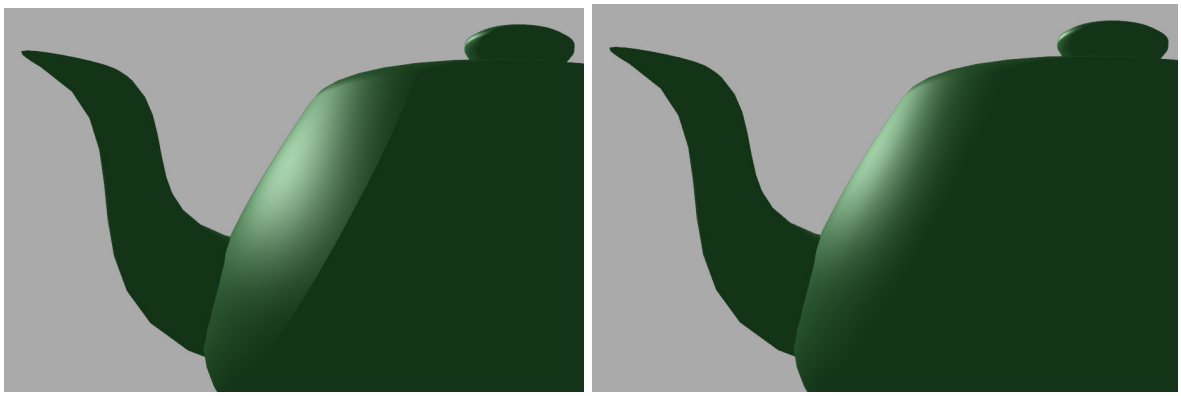





Here's the original Blinn-Phong equation. You can see than a shininess of 3 is overall brighter than a shininess of 100. By “**energy balanced**” I mean that changing the shininess does not noticeably change the amount of energy reflected from the surface. You can see the effect by running the demo that follows. By the way, this demo puts all its shaders inside the javascript program itself, if you want to look at an example of how that’s done.

----- new page ---

[***specular falloff; shininess 16***]



Using the Lambertian N dot L term also eliminates a serious problem with specular falloff. You may have noticed it yourself with the basic Blinn-Phong model. Here's a view of the model with a low shininess and the light coming from behind it. The diffuse term drops off smoothly but the specular suddenly drops to zero, giving a pretty bad result. By using the Lambertian dropoff, the specular term now fades properly.

[Additional Course Materials:
[This slide set](http://www.thetenthplanet.de/archives/3684) gives a great runthough of why physically-based materials are worth using.
]

[end of recording part 4]

Demo: Energy Balanced Blinn-Phong

[demo: Try the demo program here energy_demo.html

You can toggle between the traditional and energy-balanced versions. The program starts with the traditional model. Try varying the shininess to see the difference.
]

Lesson: Physically-Based Materials

[show surface giving off more light than it receives - bad! Make second one that's good]

Further restrictions can be built into a reflection model, such as maintaining **energy**

conservation. This means that there is not more energy reflected from the surface than shines on it. For example, the diffuse term should drop off where the specular term is stronger. This sort of reflection model is called "**physically based**", since it has some relationship to how light truly works.

Within film production and game creation there is a trend toward physically-based systems. One advantage is that materials are more understandable by artists. For example, with our energy-balanced Blinn-Phong material we can adjust the shininess without worrying as much that we'll also have to adjust the overall intensity.

[draw changing lighting conditions: sun, moon.]

One element in a physically-based system is getting the materials right, but that's just the start. For example, the time of day might change from daytime to twilight, so the overall illumination in the scene is considerably less. Instead of simply making everything dark, some rendering systems try to adapt.

[pull in zoom lens picture]

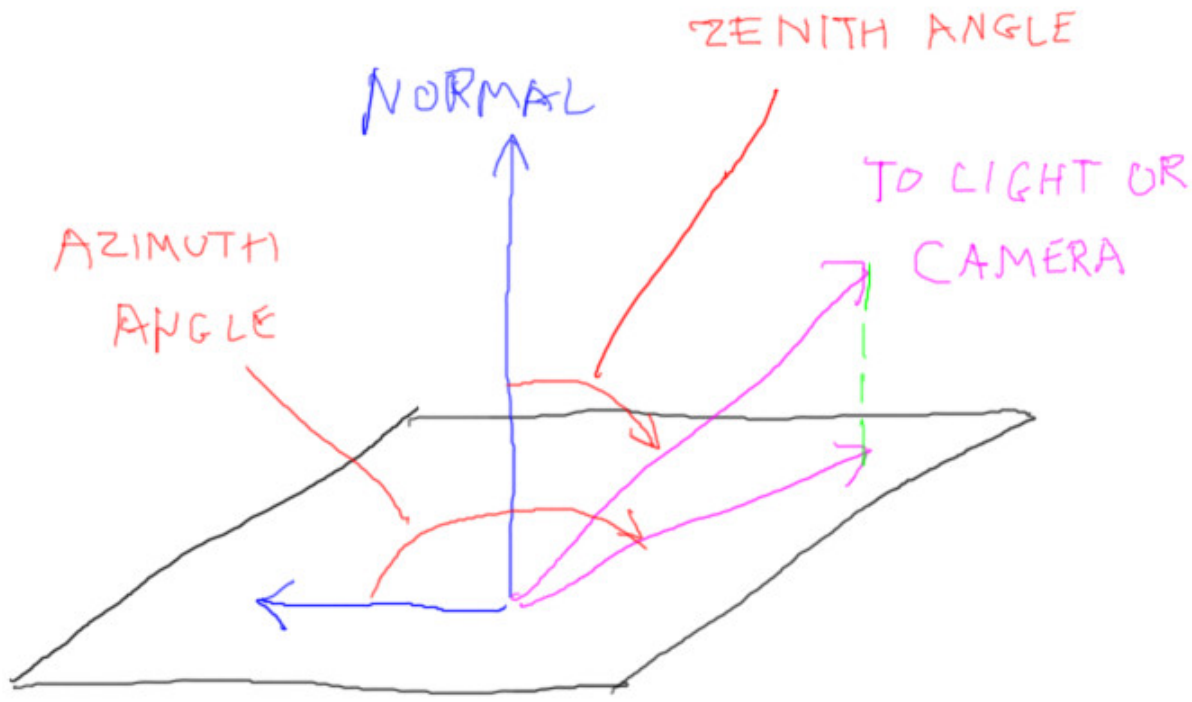
This is similar to what a photographer does by adjusting the exposure. One way is to compute a high precision image and automatically change the brightness and contrast to make the displayed image look good. This sort of analysis and display is called ***tone mapping***, and is fairly commonly used in games and other applications now.

[Instructor Comments:
For a shading language (admittedly, more for off-line high-quality rendering) that fully embraces

physically-based rendering, see OSL, the [Open Shading Language](https://github.com/imageworks/OpenShadingLanguage).
]

Lesson: BRDF

[draw in layers and save!!! We'll add separate camera angle azimuth later. Say "eye" instead of "camera"]



[put all terms below]

I want to give you a taste of how materials can be represented in a more general way. You'll also learn a great term to impress your friends and confound your enemies.

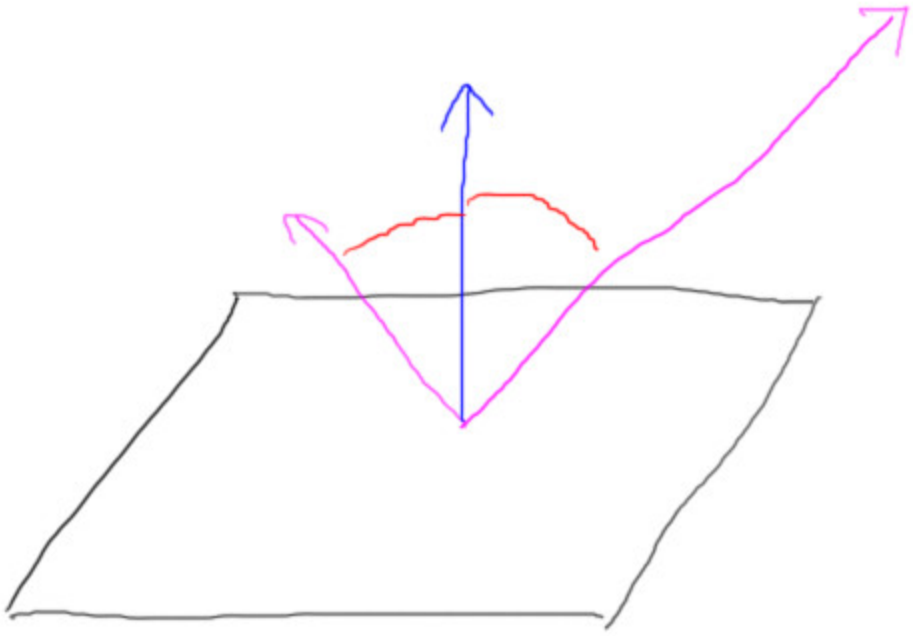
Think about a surface and how it reflects light. The two variables we use are the light's incoming direction and the amount of light reflecting toward the eye. So at its simplest, a material can be represented by this function:

intensity = material(light direction, eye direction)

This function is called the

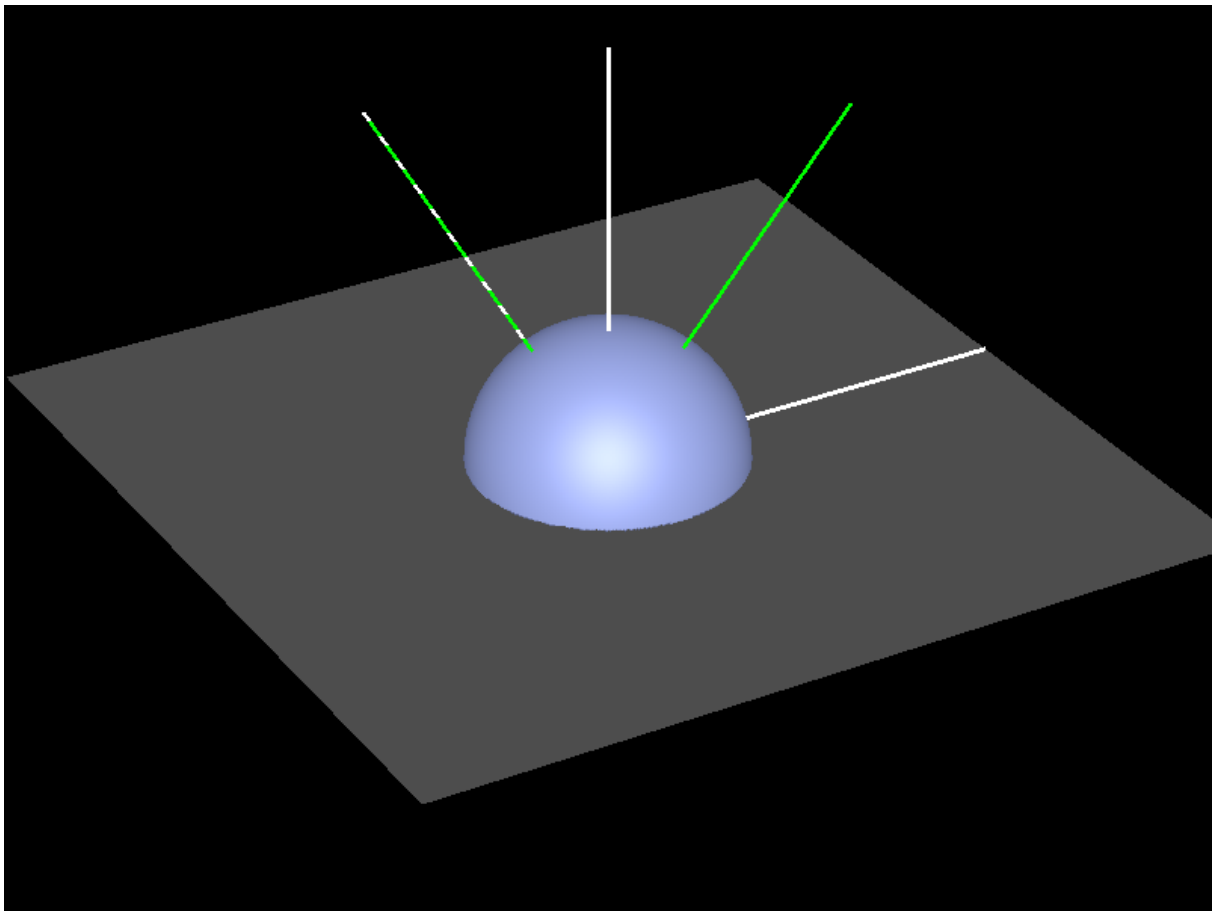
BRDF, which stands for "**Bidirectional Reflectance Distribution Function**". Let's look at that phrase. First, it's a function: the inputs are the light and eye directions. The function depends on two directions, so it's "Bidirectional". These directions are normally given with respect to the surface itself. That is, each vector is often given as two numbers, the altitude angle and the azimuth. The altitude is the angle away from the normal, and the azimuth is the angle of the vector when projected onto the plane.

[drawing of mirror reflection]



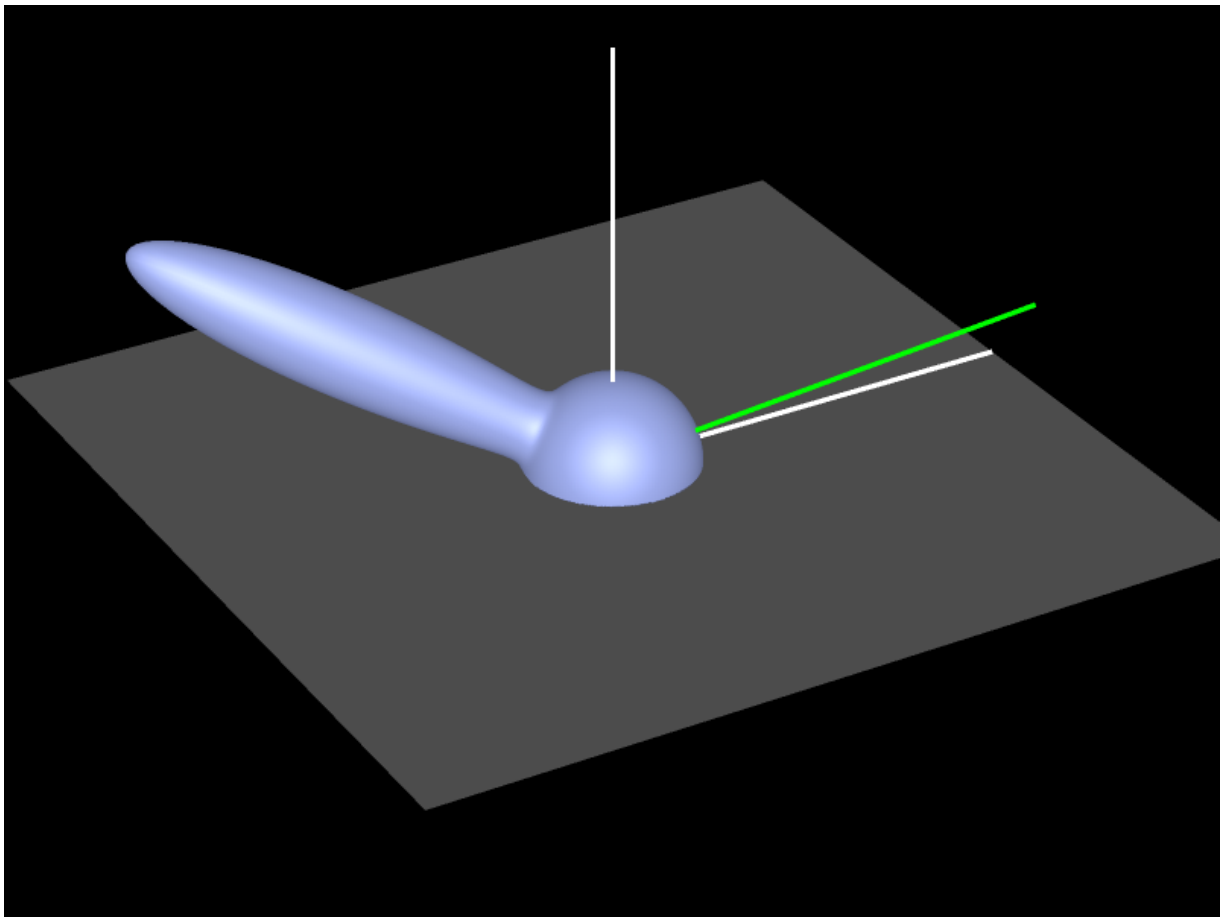
The phrase "reflectance distribution" means how the light spreads. One simple example is a perfect mirror. The reflectance distribution in this case is that when the eye's direction is exactly equal to the light's reflection direction, all light is reflected towards it. Every other eye direction gets no light.

[Prusinkiewicz program figure from my files]



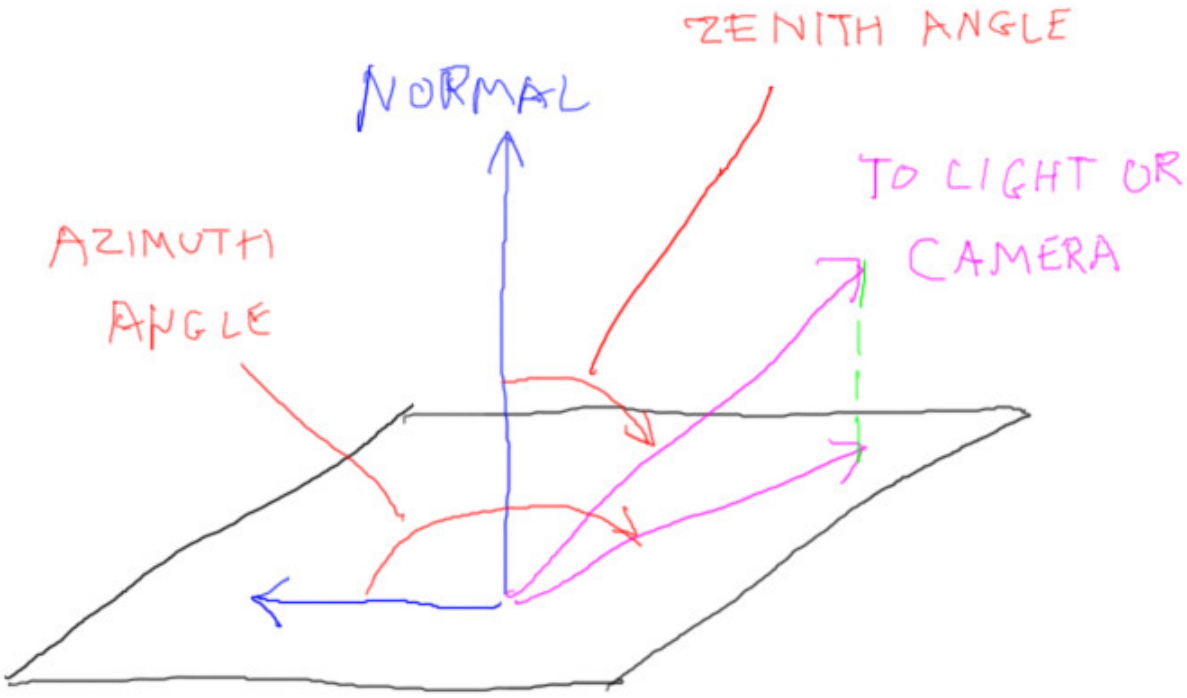
Another basic distribution is diffuse reflection. For some given incoming light direction, the direction to the eye does not matter. That's the definition of diffuse reflection. Since this value is constant, diffuse is then represented by the surface of a hemisphere.

[Prusinkiewicz figures from my files]



Specular highlights are represented by lobes. This distribution represents a glossy surface, where light is reflected in a general direction. The light's direction determines where most of the light's energy is reflected. If the lobe gets wider, the specular reflection spreads out.

[add eye angle azimuth and show azimuth difference]



If you think about it, normally you just need the angle between the incoming light azimuth direction and the eye azimuth light direction. For example, if you put a sheet of paper on a table top and rotate it, both the light azimuth and eye azimuth angle change with respect to the paper, but the angle between the two remains the same. Most materials act this way.

[Additional Course Materials:
Naty Hoffman has taught extensively on this subject at SIGGRAPH.
[Here](http://renderwonk.com/publications/s2010-shading-course/hoffman/s2010_physically_based_shading_hoffman_a.pdf) is an early introduction. [This slideset](http://blog.selfshadow.com/publications/s2012-shading-course/hoffman/s2012_pbs_physics_math_slides.pdf) and [background notes](http://blog.selfshadow.com/publications/s2012-shading-course/hoffman/s2012_pbs_physics_math_notes.pdf) bring things up to date.
]

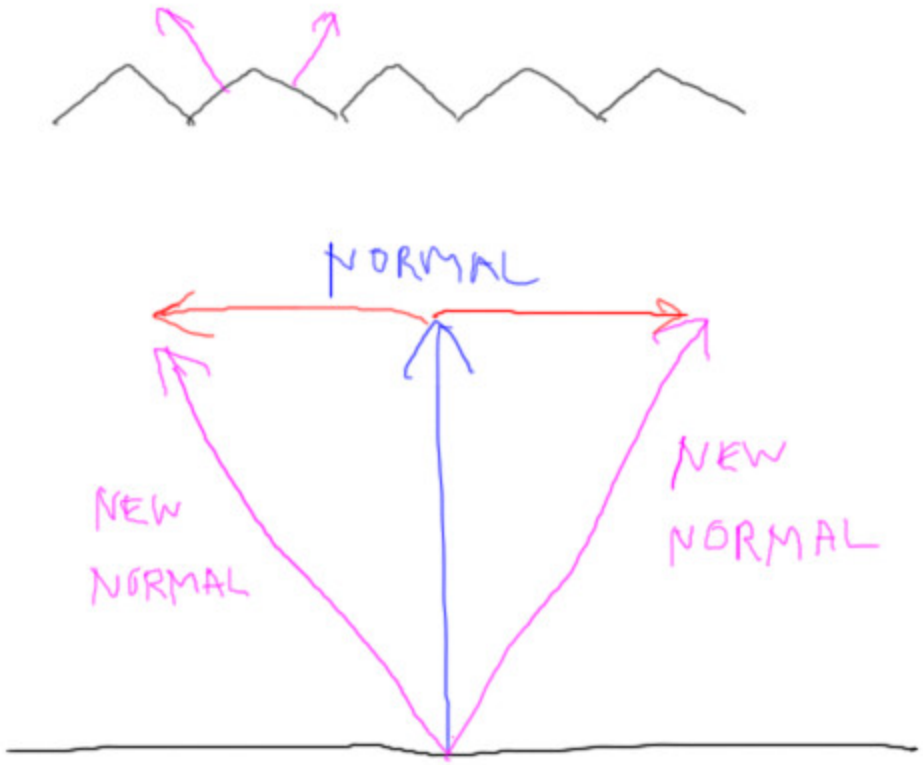
Exercise: Anisotropic Material

Some materials have a directionality built into them. Velvet and brushed aluminum are good

examples of anisotropic materials: as you rotate these the light reflects in a different fashion. These materials are called **anisotropic**, versus **isotropic**, which is how most materials are.

A simple way of making a material anisotropic is to give the object a pair of normals instead of a single normal. You start off with our energy-balanced Blinn-Phong reflection model for a plane. What I want you to do is change the single normal to be two normals, as if the material has grooves in it.

[show a drawing of how I offset the normal, and what that means]



If you look closely at the fragment shader you'll see I cheat a bit. I take the fragment's location and use its X and Z position to make these new normals. I can do this because I know the original normal is mostly pointing up along the Y axis.

Your job is to take these two normals and use them instead of the original normal. Basically, you need to apply the reflection model twice, once to each normal. To keep things about the same intensity, make both of these contributions be each just half as strong.

I should point out that GLSL has the idea of a loop built in:

```
for ( int i = 0; i < 2; i++) {
```

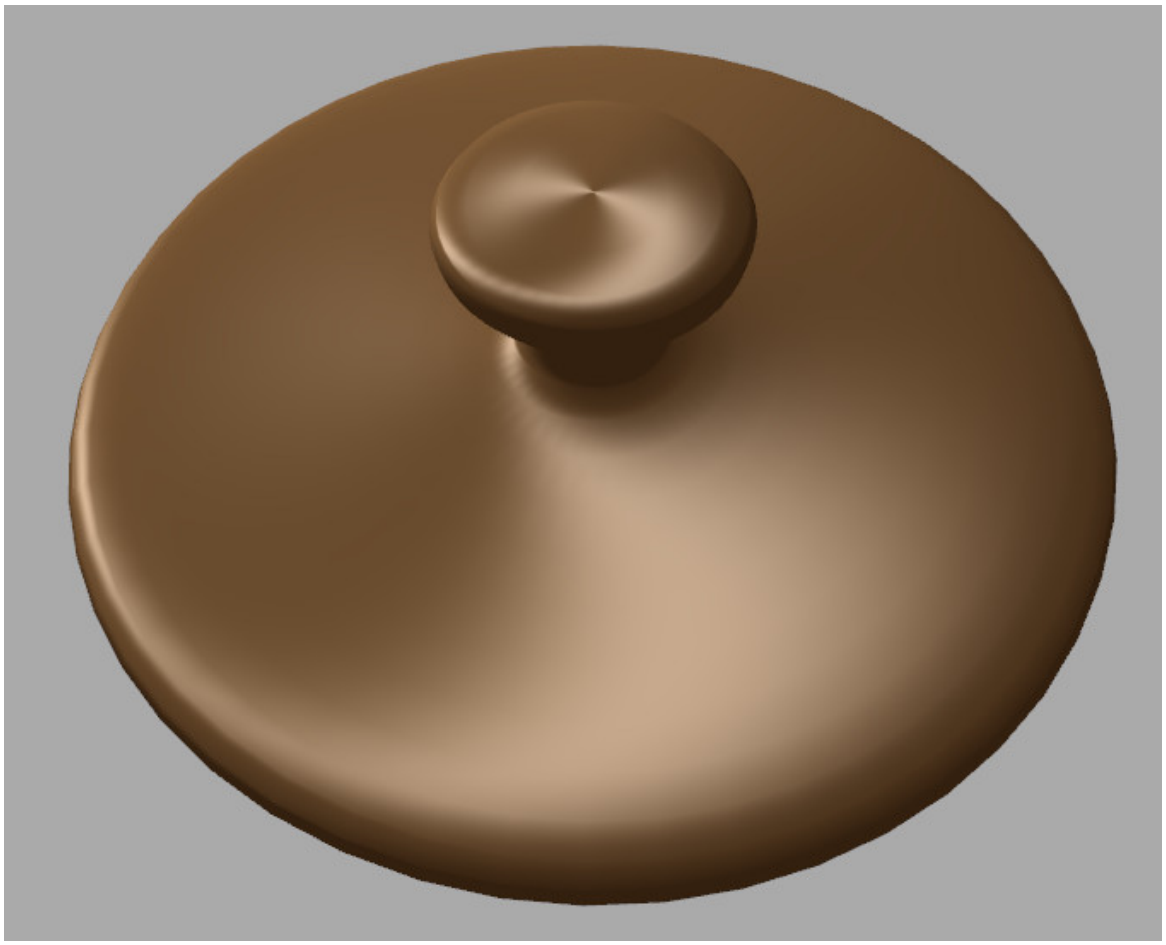
... do things ...
}

```
for ( int i = 0; i < 2; i++) {
    ... do things ...
}
```

This should prove handy for this exercise.

[show answer video: Vary groove parameter to show effect, and especially start by going to 0 to show “no grooves”

anisotropy_solution.html
]



Here's the result, which - even though it's a hack - looks pretty nice, I don't mind saying. Remember, the major rule of graphics is, "it just has to look right". One thing to keep in mind: the "groove" parameter, when set to 0, should give a result exactly the same as the original reflection model. If it doesn't, you'll need to figure out what to fix.

[exercise is at
anisotropy_exercise.html]

[Additional Course Material:
While the code here is something of a hack, there are real implementations of anisotropy, where you define a direction on the surface for how the material is oriented. See [this article](http://content.gpwiki.org/index.php/D3DBook:(Lighting)_Ward) for one classic model.
]

Answer

[Solution idea is here: anisotropy_solution.html]

```
for ( int i = 0; i < 2; i++) {
        vec3 offset = (i==0) ? vWorldPosition : -vWorldPosition;
        offset.y = 0.0;
        vec3 jiggledNormal = normalize( normal + uGroove * normalize( offset ) );
        float diffuse = max( dot( jiggledNormal, lVector ), 0.0);

        // scale diffuse contribution down by half, since there are two normals
        gl_FragColor.rgb += 0.5 * uKd * uMaterialColor * uDirLightColor * diffuse;
        ...
        float pointDotNormalHalf = max( dot( jiggledNormal, pointHalfVector ), 0.0 );
        ...
        // scale specular contribution down by half, since there are two normals
        gl_FragColor.rgb += 0.5 * uDirLightColor * uSpecularColor * specular;
}
```

```
for ( int i = 0; i < 2; i++) {
    vec3 offset = (i==0) ? vWorldPosition : -vWorldPosition;
    offset.y = 0.0;
    vec3 jiggledNormal = normalize( normal + uGroove * normalize( offset ) );
    float diffuse = max( dot( jiggledNormal, lVector ), 0.0);

    // scale diffuse contribution down by half, since there are two normals
    gl_FragColor.rgb += 0.5 * uKd * uMaterialColor * uDirLightColor * diffuse;
    ...
    float pointDotNormalHalf = max( dot( jiggledNormal, pointHalfVector ), 0.0 );
    ...
    // scale specular contribution down by half, since there are two normals
    gl_FragColor.rgb += 0.5 * uDirLightColor * uSpecularColor * specular;
}
```

The brute force answer is to copy and paste the reflection equation and substitute in the normals, making sure to divide each contribution by two. I'm hoping you used a loop.

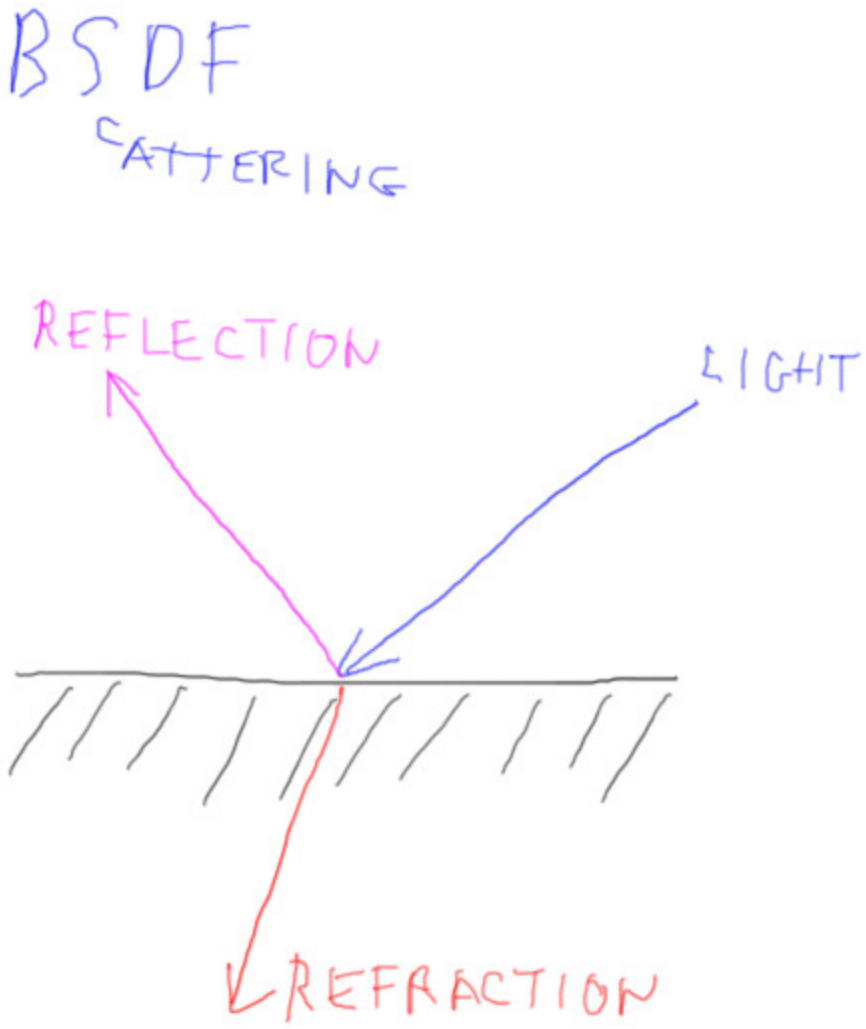
The overall coding change is to take the two jiggled normals and use them in the shading equation. Each iteration is multiplied by a half so that the total contribution is the same.

Lesson: BSDF and BSSRDF

I simply made up the anisotropic function out of my head. However, there's considerable research on how to capture BRDFs from materials and how to make functions to compactly

represent them. BRDFs are just the start. There's also the **BSDF**, the Beet Sugar Development Foundation. We're more interested in the Bidirectional Scattering Distribution Function. This type of function captures both how light reflects from and transmits through the material.

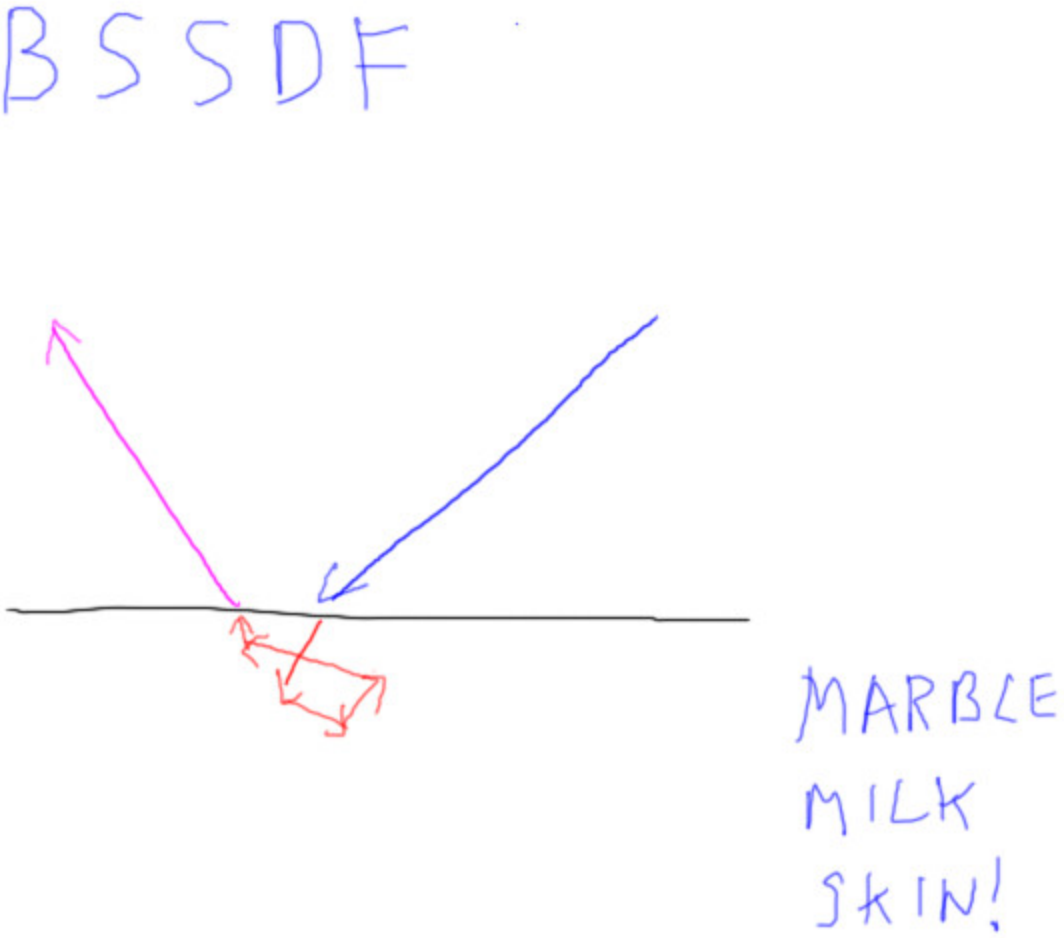
[draw reflection and transmission,]



[BSSRDF]

There's also the **BSSRDF**, which stands for "Bidirectional surface scattering reflectance distribution function" - say *that* one three times fast. This function is important for materials like marble and milk. For these materials in particular the light enters at one location on the surface, bounces around inside the material and comes out somewhere nearby.

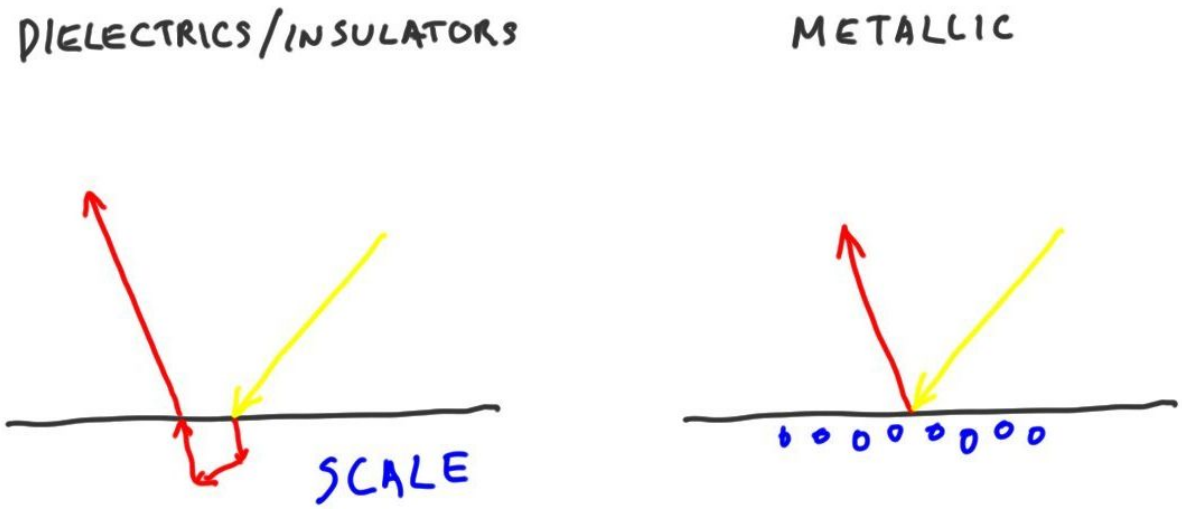
[draw example - NOTE - MISLABELED below , should be BSS***R***DF]



[Run skin demo http://mrdoob.github.com/three.js/examples/webgl_materials_skin.html or http://mrdoob.github.com/three.js/examples/webgl_materials_bumpmap_skin.html (which is newest, and with shine. See what looks better on video).]

One other extremely important material that has this sort of scattering is skin. Getting skin to look good for interactive rendering can be quite involved, but the results are more convincing than using some simple reflection model. See the additional course materials for more information. That said, the key factor here is ***scale***. The effect of sub-surface scattering lessens as the viewer's distance from the object increases. Close up, a photon might exit at a location that is a fair number of pixels away from where it entered the surface. Farther away, there may be no change in pixel location.

[draw "dielectrics / insulators / non-metallic" and show a little scattering. Show metals as direct reflectors. Make room to show eye location close and far, add as separate layer.]

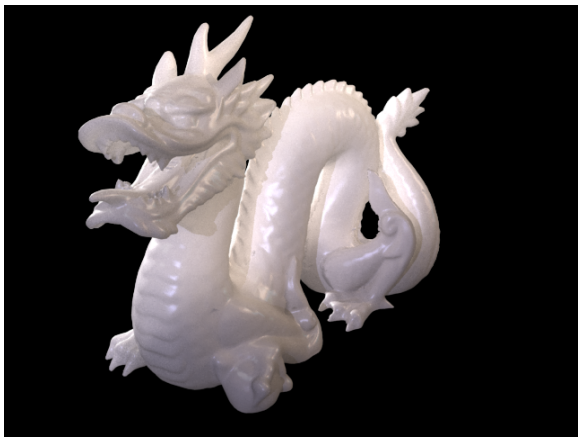


In fact, the diffuse component for all non-metallic materials comes from sub-surface scattering (it's just that in many cases this scattering is over an imperceptibly small distance). Metals themselves are essentially all specular. Let me say that again, because all this time we've been living a lie: metallic objects have no Lambertian diffuse term. Well, not a lie, I just like being dramatic. Really, diffuse is simply an approximation of which we should be aware. Using it is fine; even high-quality applications do so. It's quick to compute and looks plausible. In reality metals can indeed be given a roughened surface to give them a glossy or diffuse look, so a diffuse term is fine. However, on an atomic level metallic objects have a free floating soup of electrons on the surface which absorbs and re-emits all incoming photons. If your surface represents a shiny metal, you probably don't want a diffuse term.

Insulators have a diffuse term because the photons undergo subsurface scattering. Most of the time the entry and exit points are so close together it doesn't matter, but the direction of exit certainly does.

Materials such as that in an unglazed clay pot, concrete, and even the Moon itself are rough enough that the Lambertian reflection model doesn't capture them fully. This again turns out to be a matter of scale, having to do with the relationship of surface roughness with sub-surface scattering.

[show three marble/wax/jade things and label them]



Admittedly, trying to capture all of these effects leads to a lot of work and possibly inefficient shaders. These sub-surface scattering renderings are from 3D Studio Max and rendered offline, not at interactive rates. The main thing is to realize we don't have to stick with illumination models from the 1970's because of inertia or ignorance.

Using reflection models based on how the real-world works has a number of advantages. First and foremost, if everything is properly modeled, your virtual world acts like the real world. Change lighting conditions and you don't have to tweak material settings to look good. For design software this assurance can mean you can trust what you see on the screen to have some relationship to what you manufacture.

[http://alteredqualia.com/three/examples/webgl_lights_deferred_pointlights.html perhaps?]

Physically-based rendering is also a great help to virtual world content creators, such as game and film makers. It's a time-saver to have predictable illumination models, as the artist does not have to learn obscure sliders that have no real-world counterparts. Knowing that materials won't show some glitch from a certain angle, and knowing that lighting can be changed without destroying the sense of realism, is vastly reassuring. Rather than limit creativity, a well-designed

system makes for a more productive and unrestricted environment.

[Instructor Comments:
Now you can [handle the truth about diffuse vs. specular](http://photorealizer.blogspot.jp/2012/05/diffuse-and-specular-reflection.html).

The Hideo Kojima team has [a great video about their latest physically-based-rendering game engine](http://www.gamespot.com/metal-gear-solid-v-the-phantom-pain/videos/hideo-kojima-gdc-2013-panel-6406092/) for Metal Gear Solid. Also, compare [night](http://www.youtube.com/watch?v=ltH1eWxZutE) and [day](http://www.youtube.com/watch?v=kTmMOPh-3Is&t=2m20s).

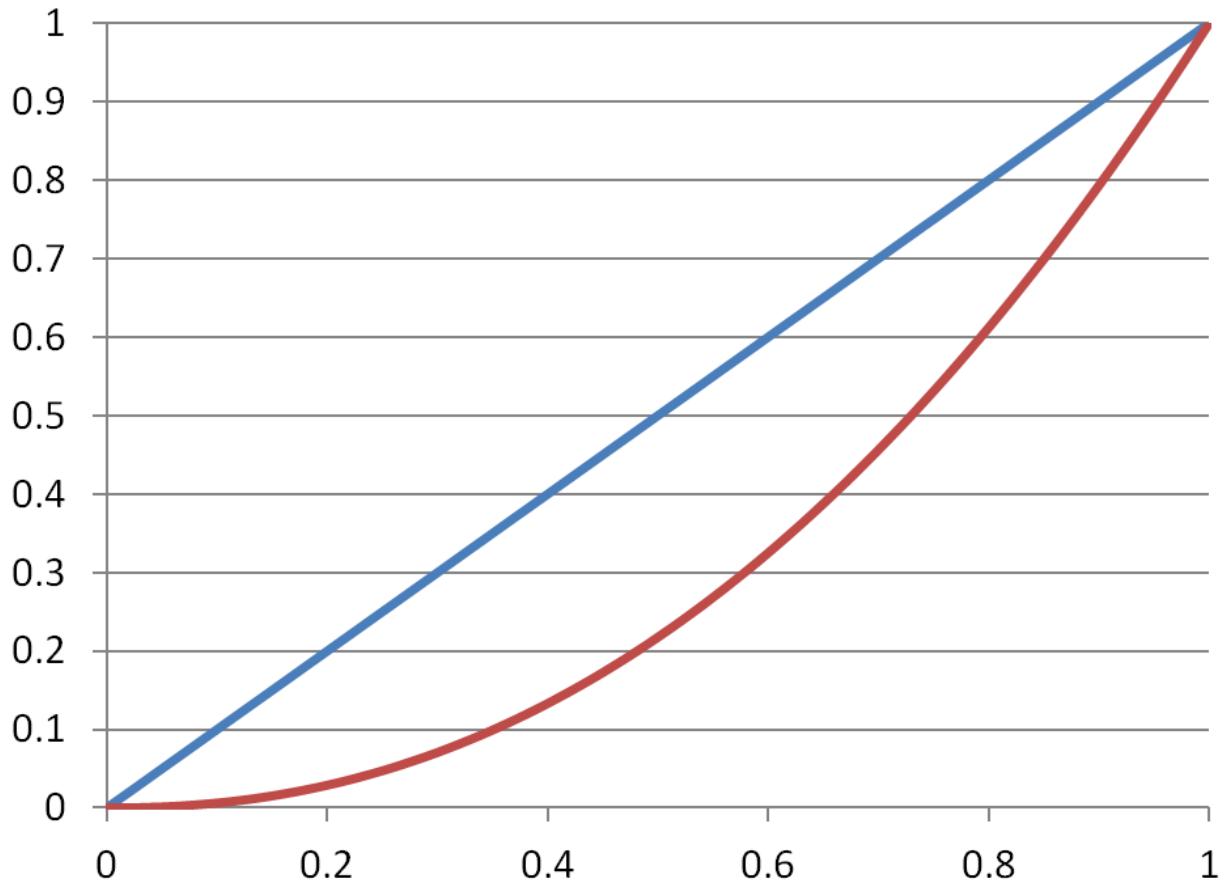
Skin demos can be seen [here](http://mrdoob.github.com/three.js/examples/webgl_materials_bumpmap_skin.html), which uses a simple wrap-around lighting model and has a more shiny and sharp look, and [here](http://mrdoob.github.com/three.js/examples/webgl_materials_skin.html), with a softer, more realistic feel, and that uses a more complex multipass texturing approach. There are a few related articles online from the [GPU Gems books](http://www.realtimerendering.com/index.html#books) about skin in particular: [this](http://http.developer.nvidia.com/GPUGems/gpugems_ch03.html) and [this](http://http.developer.nvidia.com/GPUGems/gpugems_ch16.html) from “GPU Gems”, and [this newer article](http://http.developer.nvidia.com/GPUGems3/gpugems3_ch14.html) in “GPU Gems 3”.
]

Monitor Response and Perception

[1.0, 0.6, 0.2]
[0.5, 0.3, 0.1]

Say you want to make some pixel have half the intensity as some given pixel. Logically, you'd set the pixel's channel values to be half of the original's. This is certainly a reasonable idea, and in fact I recommend it, once your system is set up properly.

The only problem is that displays do not actually respond to channel settings in this way. A monitor has a power-law response to an input signal.
[show graph]



Labels: vertical is output signal, horizontal is input signal

If you send a signal of say 0.5, then send a signal of 1.0, the intensity of the pixel doesn't double, it in fact increases by more than 4 times.

[leave off: TMI
1.0, 0.325, 0.029 is measured intensity]
[0.218, 0.071, 0.006 is measured intensity]
x4.59 x4.59 x4.59

This non-linear response of monitors sounds like something that should just get fixed, but it's in fact a feature. It turns out that our eyes have a similar response curve. A monitor has a limited number of levels of color it can display. To get good coverage of levels that the eye can detect, it's a fine thing to space these levels out in this non-linear way.

However, we usually think we're calculating colors in a linear space - if I compute a color that is half as intense, I'd expect to see that result.

Many applications ignore this mismatch between how to calculate color and how to display it, and it's less noticeable than you might think. If everything you do is in this space, then you often don't feel anything is wrong, it's what you're used to seeing. In fact, I'll warn you that many of the shader program examples in this unit blithely ignore this problem, since the focus is on keeping the programs simple.

However, there are a few important places where computing in the monitor's space can cause problems:

[* Lights do not add correctly. {Draw two circles overlapping}] If you have two spotlights overlapping, this area of overlap will not be twice the intensity as the individual spotlit areas. If you want to be more physically-based and be able to trust that your virtual world has some connection with the real world, you need to be able to add lights together sensibly.
[* The hue of colors can shift.] As you vary the intensity, the channels shift in a non-linear fashion and so shift the perceived hue.
[* Antialiasing won't look as good. {show half-covered pixel.}] Say you have a white polygon on a black background. If a pixel is half-covered by a polygon, it should be half the intensity as one fully covered by the polygon. If no correction is made for the monitor's non-linear response to inputs, the half-covered pixel will be dimmer. This error will cause even highly sampled pixels to be wrong and gives what's called "braiding" or "roping" along the edges. See the additional course materials for a comparison.
[* Mipmapped textures will appear dimmer in the distance. { Put checkerboard 0,1,1,0, will blend to 0.5 but is perceived as 0.5^2.2 }]. The simplest case is the mipmap for a 2x2 checkerboard texture. The 1x1 average mipmap is a gray of level 0.5, but as I've said maybe 5 times now, this gray will appear darker on the screen. This can cause textures with high contrast to look dimmer as these lower resolution mipmaps are accessed.

The solution to these problems is called "***gamma correction***".

[Additional Course Materials:
The [gamma correction article](http://en.wikipedia.org/wiki/Gamma_correction) on Wikipedia is pretty good.
To see what your device's gamma is, see [Cornell's gamma page](http://www.graphics.cornell.edu/~westin/gamma/gamma.html) - it's usually 2.2. If you try this on a laptop, make sure to look at the screen face on. Many laptop displays vary considerably with view angle.

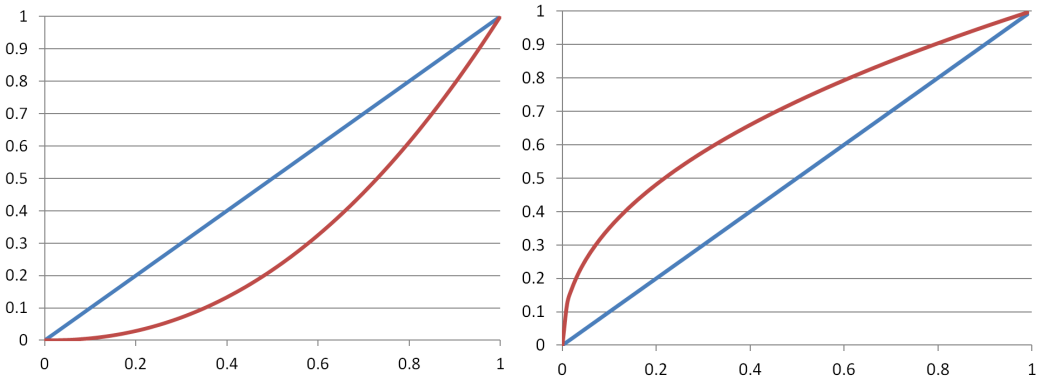
For an excellent comparison of gamma-corrected edges, see [this properly corrected image](http://www.realtimerendering.com/gamma22.png) vs. [this uncorrected one](http://www.realtimerendering.com/gamma10.png).

[Here's](http://filmicgames.com/Images/GammaIntro/800/page0039.jpg) a compare of lighting of a single light and how the falloff is poor without gamma correction. Note the sharp edge for the

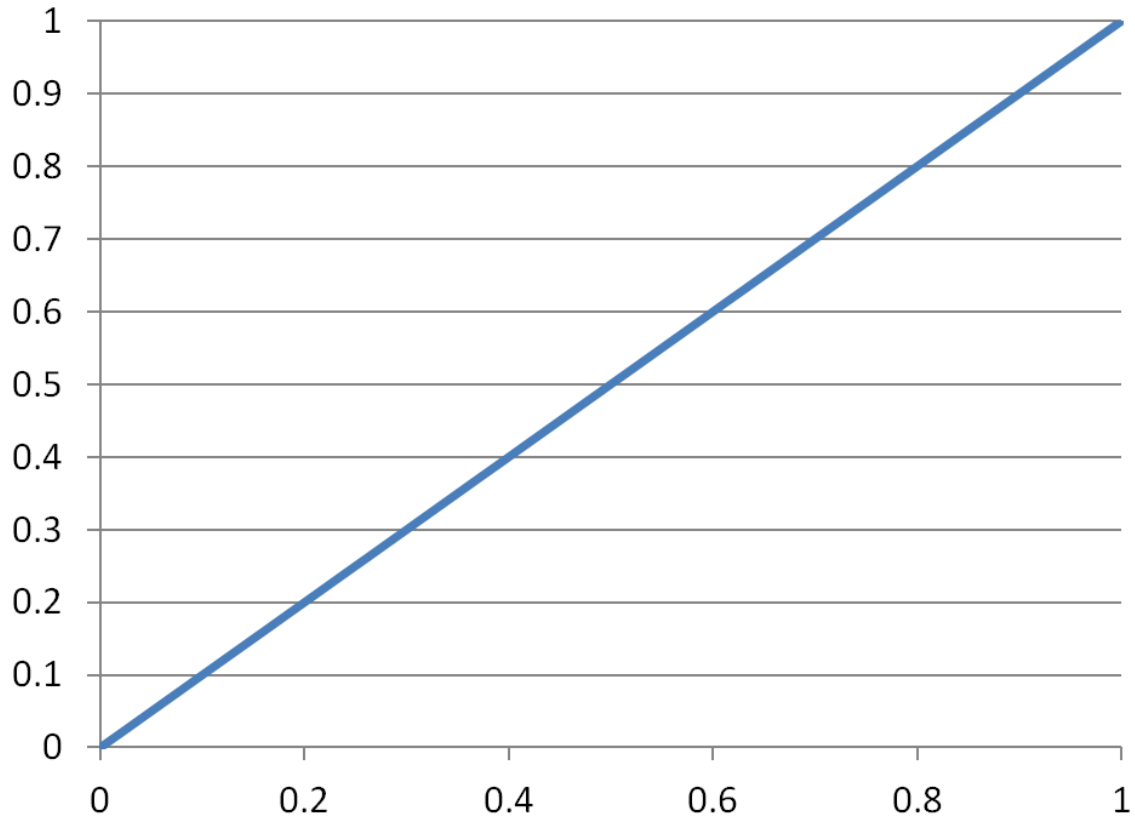
lighting falloff in the upper image.
]

Gamma Correction

[show monitor response curve, then show correction curve, for linear]



[linear:]



We have two separate wishes:

**** Compute our renderings in a linear space.***
**** Compensate for the power curve of monitors.***

To resolve these differences, we perform “gamma correction”. At its simplest, the equation is just this:
[say this: ***display channel value = computed channel value ^ (1/2.2)]***

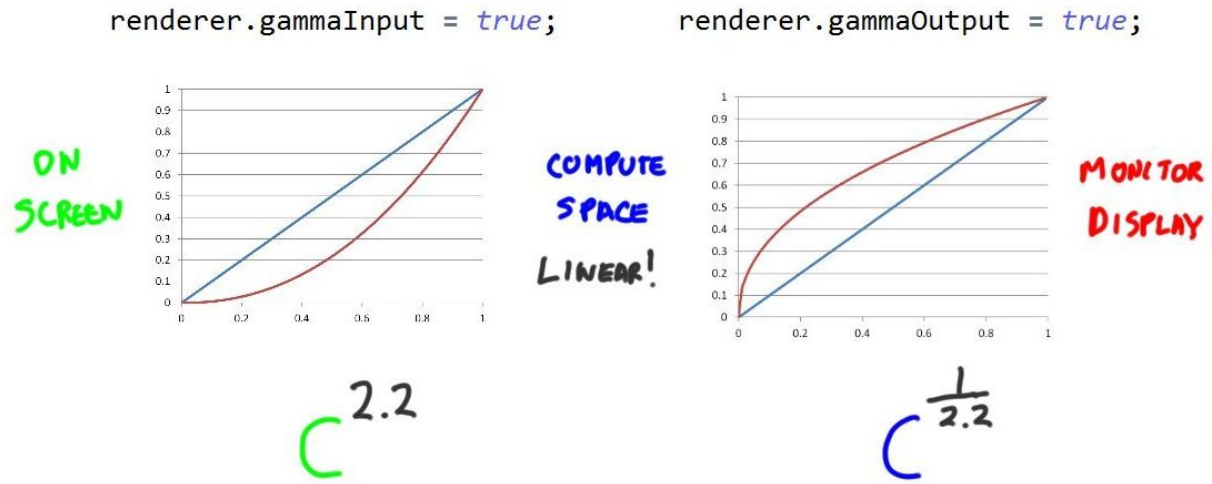
This value of 2.2 is called the gamma correction factor. Older Mac displays used 1.8 or so, but now 2.2 is the norm for all monitors.

[***computed color (1.0, 0.6, 0.2) -> (1.0, 0.793, 0.481) color to display***]
For example, say we have a value of (1.0, 0.6, 0.2). We raise 1 to this power [point: of (1/2.2)] and it doesn't change, it stays 1.0. 0.6 goes to the value 0.793, 0.2 raised to a power goes to 0.481. Notice how each channel changes by a different factor: 1.0 doesn’t change at all, 0.2 more than doubles. This is why color shifting can occur if you don’t gamma correct.

Gamma correction is something that is applied to the whole image just before display. If you try to do it yourself, you have to be aware of precision problems with your input data: if you have only 8 bits per channel coming in, you won't be getting 8 bits of good data coming out.
The good news is that three.js has gamma correction built in, all you have to do is ask for it.

One last detail is that when you look at a color swatch or a texture on the screen, you're seeing the monitor's power-curve version of the data. If you want to use that data properly when computing lighting effects, you actually have to raise each channel value to the power of 2.2 before using it for anything, either making mipmaps or making images themselves. Three.js uses an on-the-fly approximation, squaring the texture’s color when it’s sampled.

[Draw the three curves again? Show monitor vs. compute space, and show G ^ 2.2 and L ^ (1 / 2.2)]



To sum up, whatever we see on the screen is in this non-linear gamma space. To use it in lighting computations we want to make the data linear, so we raise each channel's value to the 2.2 power. We perform our lighting calculations, then at the end we go back to the monitor's space by raising to 1 / 2.2.

```
renderer.gammaInput = true;
renderer.gammaOutput = true;
```


```
renderer.gammaInput = true;
renderer.gammaOutput = true;
```


The first parameter, gammaInput, says to take the inputs such as color swatches and texture maps and linearize them. The second parameter says on output gamma correction should be performed.

Some GPUs and APIs have built-in capabilities for dealing with gamma properly, but currently these aren't accessible through WebGL and three.js. Internally in three.js there is some approximation of gamma, so a value might be squared instead of raised to the 2.2 power to bring it into the linear computation space.

[Additional Course Materials:
Three.js does not properly correct texture colors as they're read in, but rather on sampling the uncorrected mipmap levels. This leads to mipmaps still getting darker as higher mip levels are used.

A quick visual rundown of the effect of gamma correction, and much else having to do with physically-based materials, is in this [worthwhile slide set](http://www.thetenthplanet.de/archives/3684). My favorite article on gamma correction is from the free and wonderful book [GPU Gems 3](http://http.developer.nvidia.com/GPUGems3/gpugems3_ch24.html). See [our online books page](http://www.realtimerendering.com/index.html#books) for more free books. Other good articles are [here](http://filmicgames.com/archives/299) and [here](http://filmicgames.com/archives/category/gamma), among many others, such as [Wikipedia's coverage](http://en.wikipedia.org/wiki/Gamma_correction).

There's a bit more to monitor response and gamma correction, see [this article on sRGB](http://en.wikipedia.org/wiki/Srgb) for what the industry aims for. See [our

blog](http://www.realtimerendering.com/blog/?s=srgb) for way more on sRGB and related monitor and color issues. WebGL has a proposed (but unimplemented) sRGB extension](http://www.khronos.org/registry/webgl/extensions/EXT_sRGB/), more [here](http://www.khronos.org/registry/gles/extensions/EXT/EXT_sRGB.txt).
]

[end recorded 4/8]

Demo: Gamma Correction Demo

Use two spotlights to show additive problems and color shifting.

[Additional Course Materials:
This demo shows the effect of gamma correction on output. Light colors are not adjusted on input (renderer.gammaInput = false;). Note how the overlapping area is considerably brighter when gamma correction is turned off.
]

Lesson: Texturing and Post-Processing

[recorde 4/9]

[add to previous lesson: try the demo out and see the difference for yourself.]

I haven't talked about texture access in fragment shaders. The short answer is: "you can access textures in fragment shaders". The theory of sampling and filtering is the same as you've already learned in previous lessons. Most of the rest is just the syntax of setting up the texture access. The key bit is that in the shader you use the "***texture2D***" function to access a texture.

```
vec4 texelColor = texture2D( map, vUv );
```

vec4 texelColor = texture2D(map, vUv);

This gives back a color from the texture, which you then use as you wish. Search the three.js codebase for this keyword and you'll find more than forty shaders that use texture access. The main file for material shading is ***WebGLShaders.js***, and is worth your time if you want to become better with shaders.

The other function to look for is "***textureCube***", for cube maps, which takes a 3D direction as an input.

```
vec4 cubeColor = textureCube( tCube,
        vec3( -vReflect.x, vReflect.yz ) );
```

```
vec4 cubeColor = textureCube( tCube,
    vec3( -vReflect.x, vReflect.yz ) );
```

Many of the rest of the shaders perform ***image processing***, where the pixels of the image are used to form another image. Let's take a concrete example: you want to convert a color image to grayscale. The formula is:

luminance = 0.212671 * R + 0.715160 * G + 0.072169 * B for linearized colors

Luminance is intensity, the grayscale. There's also another formula for luma:

luma = 0.3 * R + 0.59 * G + 0.11 * B when not gamma correcting

It's surprising to me that, in both formulas, how important green is and how little blue matters.

Whichever formula you use, you might think you could simply apply it as the final step in a fragment shader. However, transparency again is a problem: you want to apply this formula as a ***post-process.*** That is, you want to convert to grayscale after the whole scene is rendered. But by then it's too late, the image has been sent to the screen.

---- new page ----

Well, in fact you can send the image off screen, and have it become a texture. This is called, simply, an ***offscreen texture***, ***pixel buffer,*** or ***render target***. This texture is not a typical powers-of-2 texture, it's not normally 512x512 or anything that you'd use to make a mipmap chain.

[draw texture, show quad getting drawn and shader reading texture and output new, same-sized texture, then chain another shader and texture produced, ellipsis. Multiple passes, but we'll show pingponging later)]

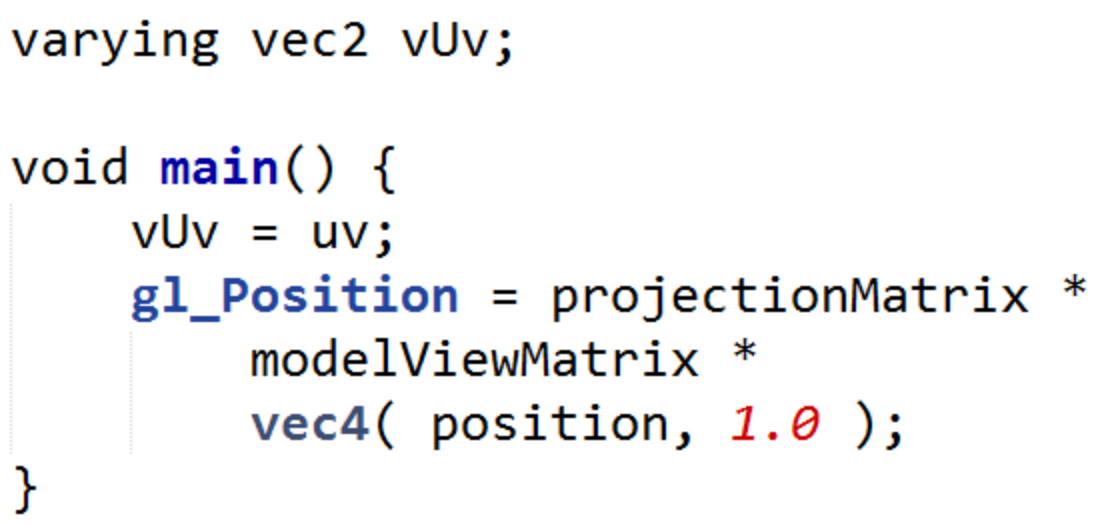
Our goal is to perform image processing on this so-called texture. We want to read a pixel and convert it to gray. You'd think there would be a mode that could just take one texel and spit out a new pixel, but the GPU is optimized for drawing triangles and accessing textures.

The way we use this texture is by drawing a single rectangle with U,V coordinates that exactly

fills the screen. You'll hear this called a "***screen-filling quad***". The UV values then are used in the fragment shader to precisely grab the single texel that corresponds to our final output pixel on the screen, one for one. This sounds inefficient, and in a certain sense it is, but this process is fast enough that often a considerable number of post-processing passes can be done each frame. In three.js the ***EffectComposer*** class lets you chain different passes together with just a few lines of code.

For our grayscale post-process the vertex shader is along these lines:

```
varying vec2 vUv;

void main() {
	vUv = uv;
	gl_Position = projectionMatrix * modelViewMatrix * vec4( position, 1.0 );
}
```



This is almost as simple as a vertex shader can get. It copies over the UV value and projects the screen-filling quadrilateral to clip coordinates.

The whole point of rendering the rectangle is to force the fragment shader to be evaluated at every pixel.

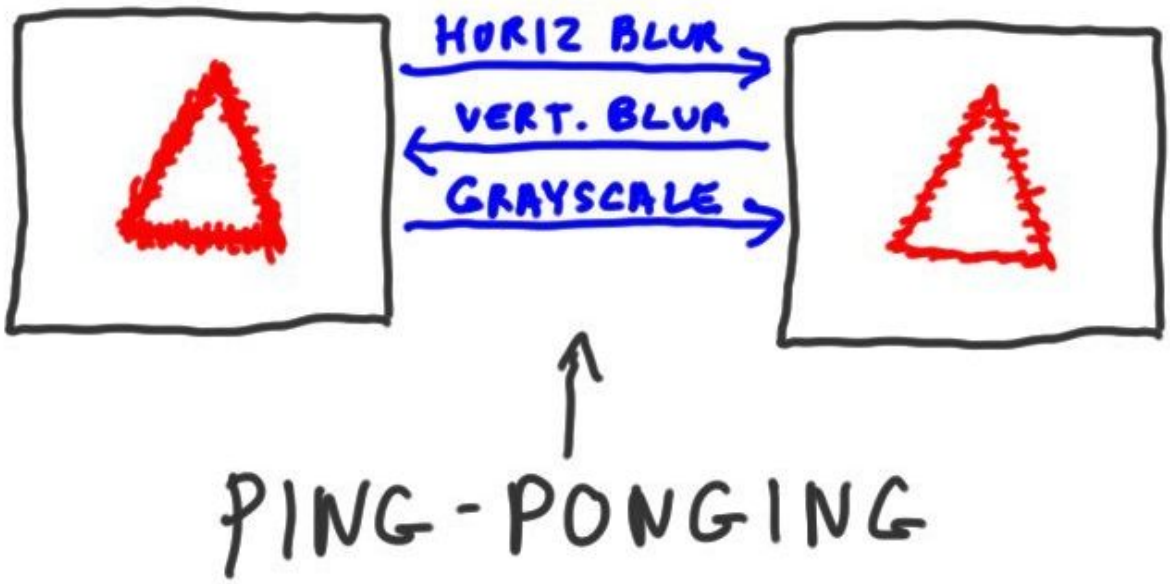
```
uniform sampler2D tDiffuse;
varying vec2 vUv;

void main() {
	vec4 cTextureScreen = texture2D( tDiffuse, vUv );
```

```
        // luma, for non-gamma-corrected computations
        vec3 lumaColor = vec3(
                cTextureScreen.r * 0.3 +
                cTextureScreen.g * 0.59 +
                cTextureScreen.b * 0.11 )

        gl_FragColor.rgb = vec4( lumaColor, 1.0 );
    }
```

```
uniform sampler2D tDiffuse;
varying vec2 vUv;

void main() {
    vec4 cTextureScreen = texture2D( tDiffuse, vUv );

    // luma, for non-gamma-corrected computations
    vec3 lumaColor = vec3(
        cTextureScreen.r * 0.3 +
        cTextureScreen.g * 0.59 +
        cTextureScreen.b * 0.11 )

    gl_FragColor.rgb = vec4( lumaColor, 1.0 );
}
```

The fragment shader code is also quite simple: the texture is accessed by the screen location, essentially, so each texel will be associated with its corresponding output pixel. We use the color of this texel to compute the grayscale color, in this case using the luma equation. This color is then saved to the fragment's output color and we're done.

[draw two textures (images), label arrow from one to another "horizontal blur" and back as "vertical blur" and leave room for another arrow, "luma"]



PING-PONGING

But, we don't have to stop there. Multiple post-process passes can be done, and it's often necessary or even more efficient. For example, a blur can be done in two passes, a horizontal and a vertical blur. Doing so results in fewer texture lookups than a single pass. Multiple passes can be quite memory efficient, as the input texture for one pass can become the output texture for the next, and vice versa. This process is called ***ping-ponging***, as the flow of data goes back and forth. Here the horizontal blur uses the left image as input, the right for output. The vertical blur uses the right as input, left as output.

[add luma]

Say we now want to convert to grayscale, we'd add another pass and reverse the direction again.

[show demo http://www.airtightinteractive.com/demos/js/shaders/preview/ - zoom in a bit to remove lower left corner.

About 5 seconds on each operation:

toggle luminosity to start, then off

brightness/contrast on/off

dot screen on/off

kaleido on/off

mirror ON ONLY

vignette ADD

dotScreen ADD

]

Here's an example of grayscale in action. The critical idea here is that we can do all sorts of things in this fragment shader. We can sample nearby texels and blend them together to blur the image, detect edges, or a hundred other operations. We can chain together each post-process to feed the next.

This demo by Felix Turner shows some of the many effects you can achieve. Three.js in fact has lots of post-processing operations in its library, and it's easy to add your own - I highly recommend looking at the Additional Course Materials for links to Felix's tutorial and other worthwhile resources.

[Additional Course Material:
A much more in-depth tutorial is
[here](http://www.airtightinteractive.com/2013/02/intro-to-pixel-shaders-in-three-js/), along with
[the demo](http://www.airtightinteractive.com/demos/js/shaders/preview/) we used.
Lee Stemkoski has related commented tutorial programs
[here](http://stemkoski.github.com/Three.js/Shader-Simple.html),
[here](http://stemkoski.github.com/Three.js/Shader-Explorer.html), and render to texture
[here](http://stemkoski.github.io/Three.js/Camera-Texture.html).

Another good image manipulation demo in WebGL is
[here](http://evanw.github.com/webgl-filter/). Image processing can do great things, like
[this](http://29a.ch/sandbox/2012/fluidwebgl/).

For a peek at what graphics hardware can do to perform image processing, try out this demo, http://voxelent.com/html/beginners-guide/1727_10/ch10_PostProcessing.html, from the book
The ["ShaderX^2: Tips & Tricks" book](http://tog.acm.org/resources/shaderx/) is quite old, but free to download. It has some excellent, comprehensive articles on image processing on the GPU.

The ["WebGL Beginner's
Guide"](http://www.amazon.com/WebGL-Beginners-Guide-Diego-Cantor/dp/184969172X?tag=realtimerenderin) also has [a post-processing
demo](http://voxelent.com/html/beginners-guide/1727_10/ch10_PostProcessing.html).

The difference between luma and luminance is discussed
[here](http://www.poynton.com/notes/colour_and_gamma/ColorFAQ.html).
]

Question/Exercise: Gamma Banding

In theory you could take an output image and use gamma correction on it as a post-process. I mentioned that there's a problem with trying to gamma correct if you have just 8 bits per channel. Let's put a number on it. Your job is to figure out that number.

Say we walk through all possible channel levels, 0 through 255, and gamma correct each value.

[table - also leave room for answer, and "console.log()" line of code.]
[***input 0.0 to 1.0 raise to 1/2.2 0.0 to 255.0 round***
0 0/255 0.0 0.0 0
1 1/255 0.0805597 20.542 21
2 2/255 0.1103951 28.1507 28

]

There are a few steps to convert a channel. First convert to a floating point number by dividing by 255. Then use the gamma correction value to raise it to a power. Convert back to the range 0 through 255 by multiplying by 255, then round the fraction to get the channel number.

For example, if the stored value is 0, this converts to zero on output - no surprises there. If our computed channel level is 1, our post-processor's output is level 21. Level 2 jumps to 28. At the lower values we have to boost the output considerably to get the equivalent gamma corrected value for display. Out of the first 29 output levels, 0 through 28, we're using only three. Already there are 26 levels out of 255 we'll never use.

On the low end some levels are never used. On the high end you can have the opposite, that a number of different input levels map to a single output level. The question to you is: how many unique output levels are there? I'm expecting an integer value, and it's clearly less than 256.

_____ unique levels

Use whatever means you like to figure out this answer, any programming language you like - this is an actual, real-live programming assignment! I personally wrote a little JavaScript program and used

console.log("Different levels output:" + count);

```
console.log( "Different levels output:" + count );
```

to show me in the debug console the answer it computed.

Answer

The answer is 184. You lose more than a quarter of all output levels when you gamma correct an 8 bit image. Doing so can lead to what's called ***banding***, especially with lower values, where you can see the different levels due to the gaps.

```
var count = 0;
var prev = -1;
for (var i = 0; i < 256; ++i) {
        var infloat = Math.pow( i/255, 1/2.2 );
        var found = Math.round( infloat*255 );
        console.log( "For " + i + " gamma corrected: "
                + infloat + ", channel " + found );
        // if this next value doesn't match previous,
        // note it as a channel that is used.
        if (found !== prev) { ++count; }
        prev = found;
} console.log( "Different levels output:" + count);
```

```
var count = 0;
var prev = -1;
for (var i = 0; i < 256; ++i) {
    var infloat = Math.pow( i/255, 1/2.2 );
    var found = Math.round( infloat*255 );
    console.log( "For " + i + " gamma corrected: "
        + infloat + ", channel " + found );
    // if this next value doesn't match previous,
    // note it as a channel that is used.
    if (found !== prev) { ++count; }
    prev = found;
} console.log( "Different levels output:" + count);
```

Here's a solution, using the observation that each channel computed will be either a new one not encountered before, or equal to the previous output channel computed.

This is Patrick Cozzi's solution, which was better than mine. Patrick, Mauricio Vives, and later on, Altered Qualia (aka Branislav) did massive amounts of review on this course's materials. Thanks so much, guys. This course is at least twice as good as it would have been without their help.

[end recording 4/9]

Lesson: Conclusion

[recorded 4/2]

Headshot:

"You've now learned about the most powerful feature of GPUs - their programmability. GPUs truly are astounding things, and I guess their evolution might be considered inevitable. More than a quarter of our brains are dedicated to processing what comes from our eyes - we're a very visual species. If we were all bloodhounds, we'd probably have NPUs instead - nasal processing units. [point to nose.]

"Me, I was amazed when 3D graphics accelerators caught on in the latter half of the 1990's. Who in the world would pay a few hundred dollars to accelerate perhaps one or

two games? Happily, I was wrong.

“I asked a few others what they’ve found surprising over the past decade.”

[**video** sd-01_thequestion1.mp4 follows. It’s in]

I like the “3 people 1 question” interviews of sd-01_thequestion1.mp4 and sd-02_question2.mp4, past and future. I can imagine the first being a part of Unit 8 , and the second being a part of Unit 9 (like maybe a final thing after the final lesson?).

[Additional Course Materials:
For some idea where the state of the art is today, see [these talks](http://advances.realtimerendering.com/) from the “Advances in Real-Time Rendering” course from SIGGRAPH. Better yet, [attend SIGGRAPH](http://siggraph.org)! Volunteers are welcome.

An excellent source of information about the internals and optimizing for the GPU is the [Beyond Programmable Shading course](http://bps12.idav.ucdavis.edu/) at SIGGRAPH. Despite the name, it has much material directly relevant to shader programming.
]

Problem Set 9

[recorded 4/6]

Problem 9.1: Vertex or Fragment Shader?

For the following tasks, which type of shader is normally used, vertex or fragment? Note that you can check both, or neither. The tasks are.

Vertex / Fragment Shader
[ ] [ ] Blurring an image
[ ] [ ] Changing an object’s shape
[ ] [ ] Evaluating an illumination model
[ ] [ ] Performing Gouraud interpolation

No trickiness here - you always need to have both shaders. Also, there’s sometimes a way to do

some of these operations using a shader, since they're programmable. Just check the ones that are the norm and can do the bulk of the work.

Answer

[ ] [X] Blurring an image
[X] [ ] Changing an object's shape
[X] [X] Evaluating an illumination model
[ ] [ ] Performing Gouraud interpolation

Blurring is an image processing effect, so most of the work is done by the fragment shader.

The vertex shader is the only one that can move vertex positions, so is the one that can change an object's shape.

Evaluating an illumination model can be done at the vertex or at the pixel level, so both shaders can do this work.

Gouraud interpolation itself is done in the rasterization stage, so neither shader does anything along these lines.

Problem 9.2: Make a Moving Flashlight

[show original flashlight exercise itself, and vary radius, and vary angle of view to show size increase.
flashlight_exercise.html
]

Here's a flashlight effect. It's pretty simple to code, we'll look at that in a second. However, the effect is not very convincing, as the flashlight can't currently move around.

```
// flashlight: is XY point in camera space inside radius?
if ( length( vViewPosition.xy ) > uFlashRadius ) {
        return;
}
```

```
// flashlight: is XY point in camera space inside radius?
if ( length( vViewPosition.xy ) > uFlashRadius ) {
    return;
}
```

This is the only addition to the fragment shader. The vViewPosition is the location of the surface in view space (actually, this is the position negated, but for our purposes it doesn't matter).

Since view space is looking down the -Z axis and X and Y are zero at the center of the screen, the flashlight always shines at the middle of the screen. We can return early if the surface is outside the given flashlight radius, so that only the ambient contribution lights the object.

I've added the uniform ***uFlashOffset***, a ***two element vector*** that gives a center point for the flashlight. There are sliders hooked up to it, but the vector is not used in the fragment shader itself - that's your task. If the X,Y view position and this location are close enough together, the flashlight should illuminate the surface.

[show solution, varying X and Y position
flashlight_solution.html
]

When you're done, the sliders should move the flashlight around. Look in the additional course materials for some GLSL documentation links. You can solve this problem with a one-line change by using the proper GLSL built-in function.

[Additional Course Materials:
Here are some GLSL (the OpenGL Shading Language) resources: [reference site](http://www.opengl.org/documentation/glsl/), [shading language description](http://www.opengl.org/wiki/OpenGL_Shading_Language), [manual pages](http://www.opengl.org/sdk/docs/manglsl/), [data types](http://www.lighthouse3d.com/tutorials/glsl-core-tutorial/glsl-core-tutorial-data-types/), and [reference guide](http://www.cs.cmu.edu/afs/cs/academic/class/15462-f10/www/lec_slides/glslref.pdf).
]

Answer

```
// flashlight: is XY point in camera space close enough to uFlashOffset?
if ( distance( vViewPosition.xy, uFlashOffset ) > uFlashRadius ) {
        return;
}
```

```
// flashlight: is XY point in camera space close enough to uFlashOffset?
if ( distance( vViewPosition.xy, uFlashOffset ) > uFlashRadius ) {
    return;
}
```

Instead of checking the length of vViewPosition from the center of the view camera's screen, take the distance from vViewPosition to the uFlashOffset position. The "distance" function is built in, and works for any type of vector, 2, 3, or 4 elements.

Problem 9.3 Model Deformation

[Show original code:]

```
varying vec3 vNormal;
varying vec3 vViewPosition;

uniform float uSphereRadius2;        // squared

void main() {

    gl_Position = projectionMatrix * modelViewMatrix * vec4( position, 1.0 );
    vNormal = normalize( normalMatrix * normal );
    vec4 mvPosition = modelViewMatrix * vec4( position, 1.0 );
    vViewPosition = -mvPosition.xyz;
}
```

```
varying vec3 vNormal;
varying vec3 vViewPosition;

uniform float uSphereRadius2;   // squared

void main() {

    gl_Position = projectionMatrix * modelViewMatrix * vec4( position, 1.0 );
    vNormal = normalize( normalMatrix * normal );
    vec4 mvPosition = modelViewMatrix * vec4( position, 1.0 );
    vViewPosition = -mvPosition.xyz;
}
```

In this exercise we start with a square tessellated into a grid of smaller squares. This grid allows us to deform the surface.

Deformation happens in the vertex shader.

[underline ***position*** in code]

What I want you to do is modify the incoming vertex position. Since this value is passed in, you can't just change it. You'll need to copy it over to a temporary variable and modify that. You'll also need to use this temporary variable in the rest of the shader instead of the position

[NOTE THAT THERE ARE TWO SEPARATE SQUARE ROOTS HERE]

zpos = square root (uSphereRadius2 - xpos^2 - ypos^2) - sqrt(uSphereRadius2)

How I want you to modify the position is by using this formula. The first half of the function finds a point on the sphere, the second half translates the whole object down by a constant amount so that it stays in view.

The uSphereRadius2 is a value passed in by the main program, it's the square of the radius set with a slider.

someVector.y = 5.0;

Xpos, ypos, and zpos are the position coordinates. To access an individual value of a vector, use x,y, and z.

[just show partial solution image, NOT the running code]

When you're done you should see something on the screen that looks like part of a sphere. It's a bit bland, and we'll talk about this more later.

Definitely save your work, since we'll build on it.

Answer

```
void main() {

        // make a local variable we can modify
        vec3 newPosition = position;
        // set the position to be on a sphere
        newPosition.z = sqrt(uSphereRadius2 -
                newPosition.x * newPosition.x - newPosition.y * newPosition.y);
        // Offset the surface so the center stays in view. Could be done in step above.
        newPosition.z -= sqrt(uSphereRadius2);
        gl_Position = projectionMatrix * modelViewMatrix * vec4( newPosition, 1.0 );
        vNormal = normalize( normalMatrix * normal );
        vec4 mvPosition = modelViewMatrix * vec4( newPosition, 1.0 );
        vViewPosition = -mvPosition.xyz;
}
```

```
void main() {

    // make a local variable we can modify
    vec3 newPosition = position;
    // set the position to be on a sphere
    newPosition.z = sqrt(uSphereRadius2 -
        newPosition.x * newPosition.x - newPosition.y * newPosition.y);
    // Offset the surface so the center stays in view. Could be done in step above.
    newPosition.z -= sqrt(uSphereRadius2);
    gl_Position = projectionMatrix * modelViewMatrix * vec4( newPosition, 1.0 );
    vNormal = normalize( normalMatrix * normal );
    vec4 mvPosition = modelViewMatrix * vec4( newPosition, 1.0 );
    vViewPosition = -mvPosition.xyz;
}
```

Here's one solution. First the incoming position is copied, so I can modify its values. I then set the Z position with the formula given. I do this over two lines and explain what each step does.

The rest of the code remains the same, other than changing every use of the "position" vector with the "newPosition" variable.

Problem 9.4: What Went Wrong?

[show partial solution in action while talking over it
vertex_partial_solution.html
]

In the previous exercise the vertex shader was modified to change the surface locations so that a part of a sphere was formed. However, the illumination looks pretty bad, it doesn't look very curved. As we rotate around, there are lighting computations going on, the shading changes, but something's not right.

[MAKE SURE IT'S MULTIPLE CHOICE, not boxes]

Why is the illumination incorrect?

( ) The vertex shader has reversed the loops, causing the normals to reverse.
( ) The fragment shader is wrong and must be modified to properly shade the surface.
( ) The vertex shader modified the positions, but not the shading normals.
( ) The geometric normals have been warped by the vertex shader.

Answer

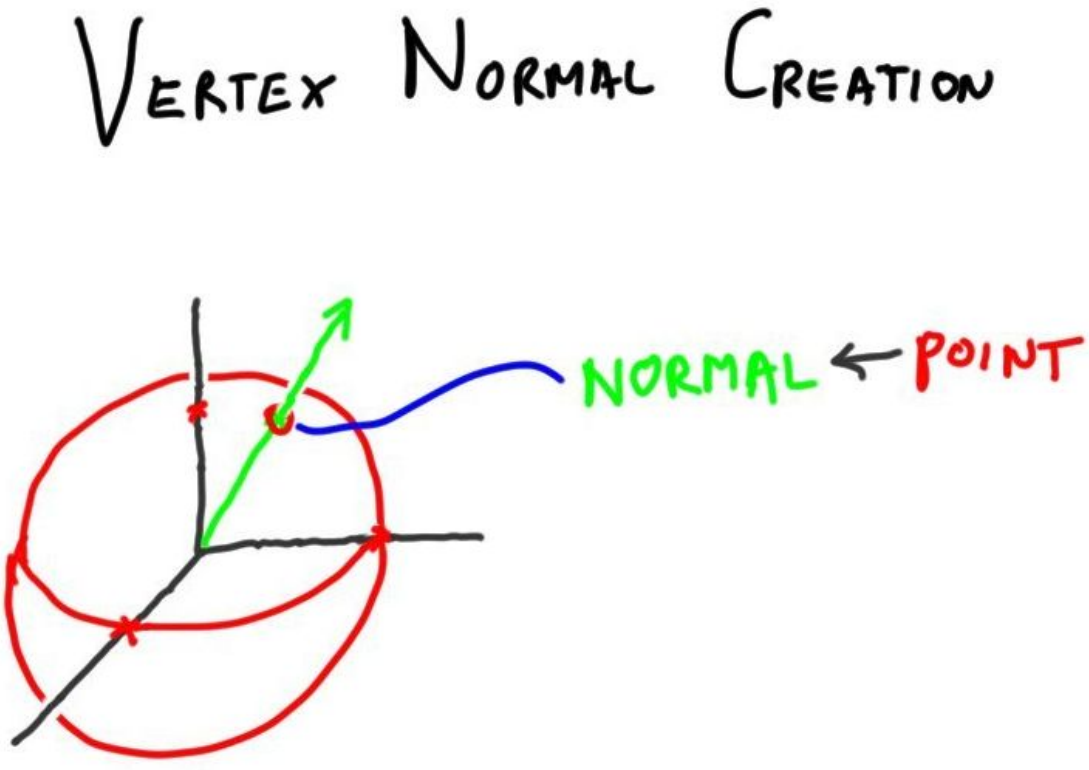
[The vertex shader modified the positions, but not the shading normals.]

The problem is that the vertex shader changed the positions but did not modify the corresponding shading normals. All the shading normals continue to point straight up.

In theory the fragment shader could also be modified, but it doesn't *have* to be; it's not wrong. It *could* be used to the position to compute this normal. It's much more straightforward to modify the shading normal in the vertex shader. This, in fact, is the next exercise.

Problem 9.5: Vertex Normal Creation

[draw sphere centered at origin - show normal and point are pointing same direction]



In addition to computing a new position to form a piece of a sphere, I want you to also set the normal. For a sphere centered at the origin, this is straightforward: notice that the vector to the point and the normal both go in the same direction.

Take the solution from the previous vertex shading problem and now change the shading normal to point in the correct direction. You'll have to use a temporary normal, just as you had to use a temporary position before. I should note you don't have to normalize this shading normal, as the fragment shader will do that later.

[Additional Course Material:
Lee Stemkoski has [a nice tutorial program](http://stemkoski.github.com/Three.js/Graphulus-Function.html) allowing you to put in a function and have the mesh generated for it. This is not vertex-shader-based, but is worth a look.

Worth a second look, Evan Wallace's [water ripples demo](http://madebyevan.com/webgl-water/) in WebGL shows normals generated on the fly in a great way.
]

Answer

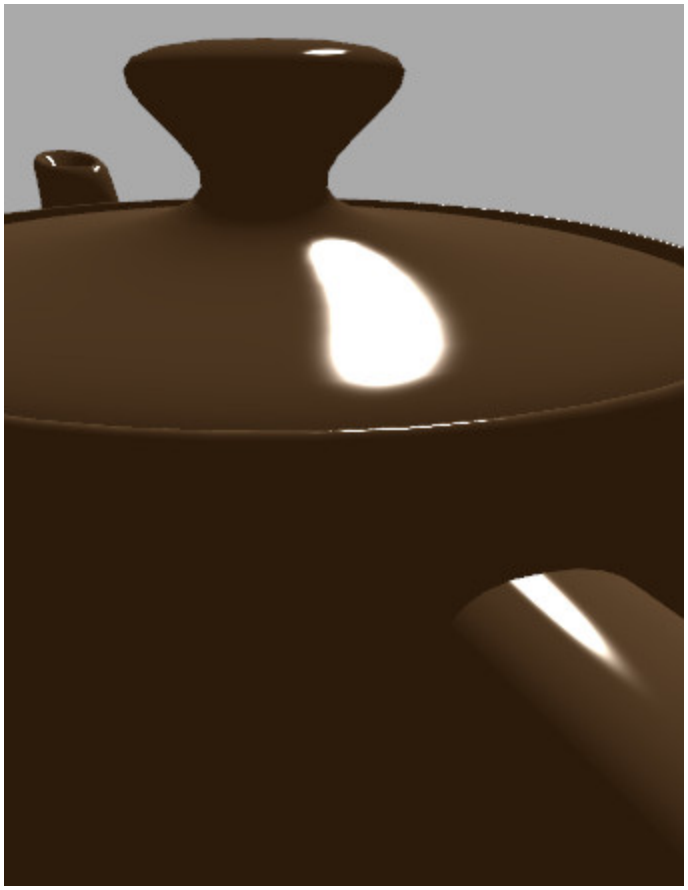
```
void main() {
	// make a local variable we can modify
	vec3 newPosition = position;
	// set the position to be on a sphere
	newPosition.z = sqrt(uSphereRadius2 -
		newPosition.x * newPosition.x - newPosition.y * newPosition.y);
	// set the normal to be identical (results from it will be normalized later)
	vec3 newNormal = newPosition;
	// Offset the surface so the center stays in view; do after the normal is set.
	newPosition.z -= sqrt(uSphereRadius2);
	gl_Position = projectionMatrix * modelViewMatrix * vec4( newPosition, 1.0 );
	vNormal = normalize( normalMatrix * newNormal );
	vec4 mvPosition = modelViewMatrix * vec4( newPosition, 1.0 );
	vViewPosition = -mvPosition.xyz;
}
```

```
void main() {
    // make a local variable we can modify
    vec3 newPosition = position;
    // set the position to be on a sphere
    newPosition.z = sqrt(uSphereRadius2 -
        newPosition.x * newPosition.x - newPosition.y * newPosition.y);
    // set the normal to be identical (results from it will be normalized later)
    vec3 newNormal = newPosition;
    // Offset the surface so the center stays in view; do after the normal is set.
    newPosition.z -= sqrt(uSphereRadius2);
    gl_Position = projectionMatrix * modelViewMatrix * vec4( newPosition, 1.0 );
    vNormal = normalize( normalMatrix * newNormal );
    vec4 mvPosition = modelViewMatrix * vec4( newPosition, 1.0 );
    vViewPosition = -mvPosition.xyz;
}
```

Here's the whole vertex shader. There are two changes to the previous solution. A temporary normal is copied from the new position, before this position is translated away from the origin. This new normal is then used to compute vNormal, the model-view shading normal.

Problem 9.6 Sharp Specular

[exercise_narrow.jpg]



One thing about Blinn-Phong specular highlighting is that the dropoff is always gradual around the fringe. Even if you turn the highlight power up to 1000 you'll still see some dropoff fringe. You also can't control the width of the highlight.

[put threshold, etc. to right of image
Check if specular < uDropoff - if so, set it to 0, else 1]

Your task is to ***threshold*** the basic Blinn-Phong specular term. What this means is that, after computing the specular contribution, test it against the variable uDropoff. If specular is less than

uDropoff, set it to 0, else set the specular to 1.

[show uDropoff sharp_spec_solution.html
]

When you're done, the solution should look like this. Note that there's a slider for "dropoff" that allows you to change how wide the highlight is.]

[Additional Course Materials:
[Thresholding](http://en.wikipedia.org/wiki/Thresholding_(image_processing)) is described on Wikipedia.

See the [Beyond Programmable Shading course notes](http://bps12.idav.ucdavis.edu/) for more about optimizing code for GPUs.
]

Answer

```
specular *= (8.0 + shininess)/(8.0*3.14159);
```

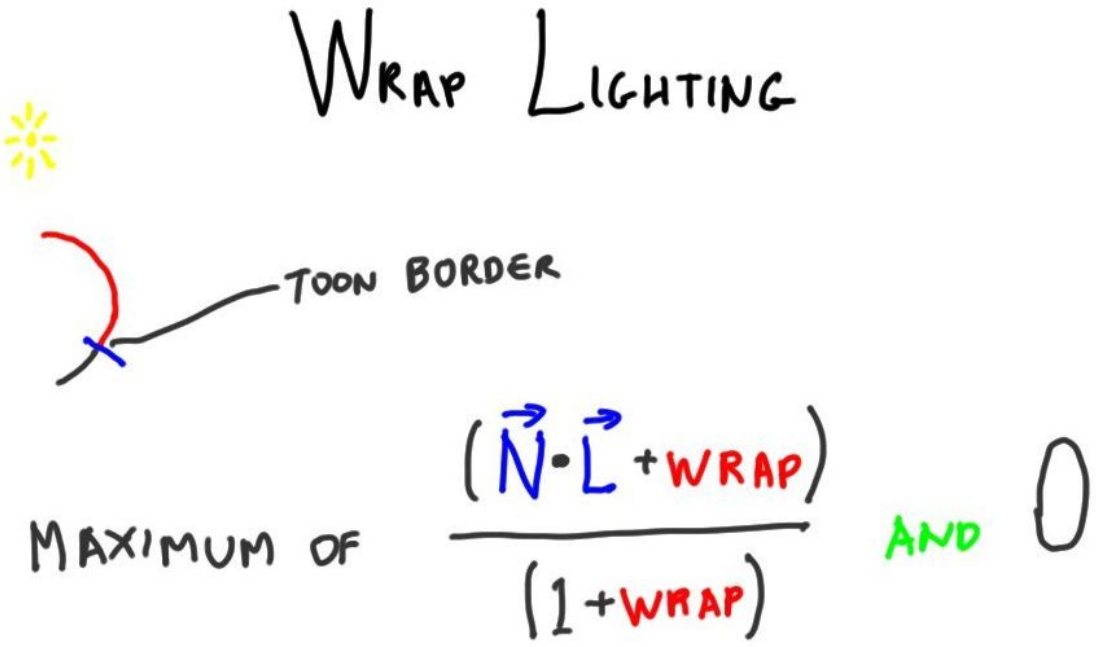
```
specular = ( specular < uDropoff ) ? 0.0 : 1.0;

gl_FragColor.rgb += uDirLightColor * uSpecularColor * uKs * specular;
```

```
specular *= (8.0 + shininess)/(8.0*3.14159);

specular = ( specular < uDropoff ) ? 0.0 : 1.0;

gl_FragColor.rgb += uDirLightColor * uSpecularColor * uKs * specular;
```

For the solution I added a line of thresholding code, using a ternary “if” statement.

One problem you might notice with this algorithm is that the highlight’s actually *too* sharp now. If you look closely at the result, it’s quite aliased, since values are either on or off. This sharpness is also true for our toon shader, but it’s even more obvious here because the contrast between light and dark is so great. There are ways of modifying the function so that it is smoother at the transition point, though because it’s view-dependent it’s hard to get this exactly right. Alternately, antialiasing schemes such as FXAA can clean up problems like this. I leave it to you to experiment!

Problem 9.7 Wrap Lighting



One cheap technique to give a material a slightly different look is to use ***wrap lighting***. Remember with toon rendering how we shifted where the diffuse term went from light to dark? We can do the same shift with the diffuse term, but not simply threshold the value.

Your task is to change the diffuse term to be this wrap lighting term.

[MAKE SURE TO put vector arrows over N*L]

> ***maximum of (N*L + wrap) / (1 + wrap) and 0***

[demo here, slide wrap slider
/wrap_lighting_solution.html
]

When you're done, the wrap slider will subtly alter the look of the surface. This equation is from an article on skin rendering by Simon Green. This lighting change is not a huge effect for teapots, but can help give a more realistic look to some materials.

[Additional Course Materials:
The article that talks about wrap lighting is [Real-Time Approximations to Subsurface Scattering](http://http.developer.nvidia.com/GPUGems/gpugems_ch16.html).
]

Answer

float diffuse = max((dot(normal, lVector) + uWrap) / (1.0+uWrap), 0.0);

```
float diffuse = max(
    (dot( normal, lVector ) + uWrap) / (1.0+uWrap),
    0.0 );
```

The standard diffuse term is changed to this line of code.

The hardest thing about this exercise for me was remembering to put “dot 0” after the “1” in the equation.

[end recording 4/6]

--- LEFTOVERS ----------

Problem 9.X Posterization

Post processing to give posterization/cel-shading effect, more reliable.

[could have Lesson: Depth Coloring - instead of setting the normal, set the color using the depth. This is pretty trivial, though, and 3D texturing does more elaborate stuff.]

Exercise:

Add a second light to a shader

Exercise:

Rim lighting for head?

Other possibles, Drinking Bird:

* Give other surfaces different settings

* Felt BRDF for bird's head?

* Radial BRDF for brim of hat?

Problem 9.X More Cel Shading

Three levels of cel shading, have them add two sliders to control.

Problem 9.X: Phong Dropoff

Note the problem of the phong highlight rapidly dropping to zero at the silhouette. Can I make a quiz question out of this? Really, mostly covered by the new energy-balanced lesson.

http://www.geekshavefeelings.com/x/wp-content/uploads/2010/03/Its-Really-Not-a-Rendering-Bug-You-see....pdf

---

Problem 9.X: Better Anisotropy

Use surface normal to generate two new normals, one to left and one to right. Do this in the vertex shader.

Problem 9.X: Fill Light

[drinking bird with specular no matter where you move]

We introduced the idea of a headlight some time ago, to take the place of the boring solid ambient light. However, the headlight idea can be too much of a good thing. The light *always* puts a specular highlight on any shiny surface. A sphere, such as the drinking bird's body, will have a shiny dot in the middle of it. This can be distracting and misleading.

[http://en.wikipedia.org/wiki/File:SVLightWithFlexReflect.jpg]

What we'd prefer is something more like a photographer's fill light. This is now possible since we can control how a light affects a scene using shaders. As a start, we could simply not compute the specular component of the material when this fill light is used.

[Additional Course Materials: Look on Wikipedia for more about fill lights http://en.wikipedia.org/wiki/Fill_light .]

Exercise: Remove the Specular

Shader with two lights, first is the headlight. Have student remove specular from first light.

[Question: Texel to UV - probably delete]

[pixel image and show 0,0 in lower left; 16 columns, 16 rows. Leave room to right to show answer.]

[drawing: 16x16, label 0,0 and 15,15 and 10,15 (on upper edge)]

Usually when we're texturing surfaces, the computer needs to go from U,V coordinates to texels. When you're trying to figure out a misalignment or other bug, you may need to go from texels to U,V coordinates.

When dealing with images as textures, you sometimes need to go

We have a texture of size 16x16. The lower left corner texel has a location of 0,0, the upper right is 15,15, with the X axis coordinate listed first. The lower left corner of this texel has U,V coordinates of 0.0,0.0. This means the U,V coordinates of the center of the lower left corner are going to be ***0.5/16,0.5/16***, in other word ***0.03125,0.03125***, because the center of this texel is 0.5,0.5 in texel coordinates.

What are the U,V coordinates of the center of texel 10,15?

U = _____, V = _____

[Answer]

U: 10.5/16 = 0.65625

V: 15.5/16 = 0.96875

The main element needed to solve this problem is the fact that the center of the texel means adding 0.5,0.5 to the texel's coordinates. Once this is done, you just divide by the number of texels to get the U or V value.

Converting in the other direction is done by multiplying U and V by the number of texels in that direction, 16. It's interesting to note that the upper right corner of the upper right texel, which has a U,V of 1.0,1.0, would then have texel coordinates of 16.0,16.0, a texel that doesn't exist. In practical terms this doesn't matter, as the value 15.999999 repeating is the same value as 16.0, so there is a valid texel and a valid texel location to sample.