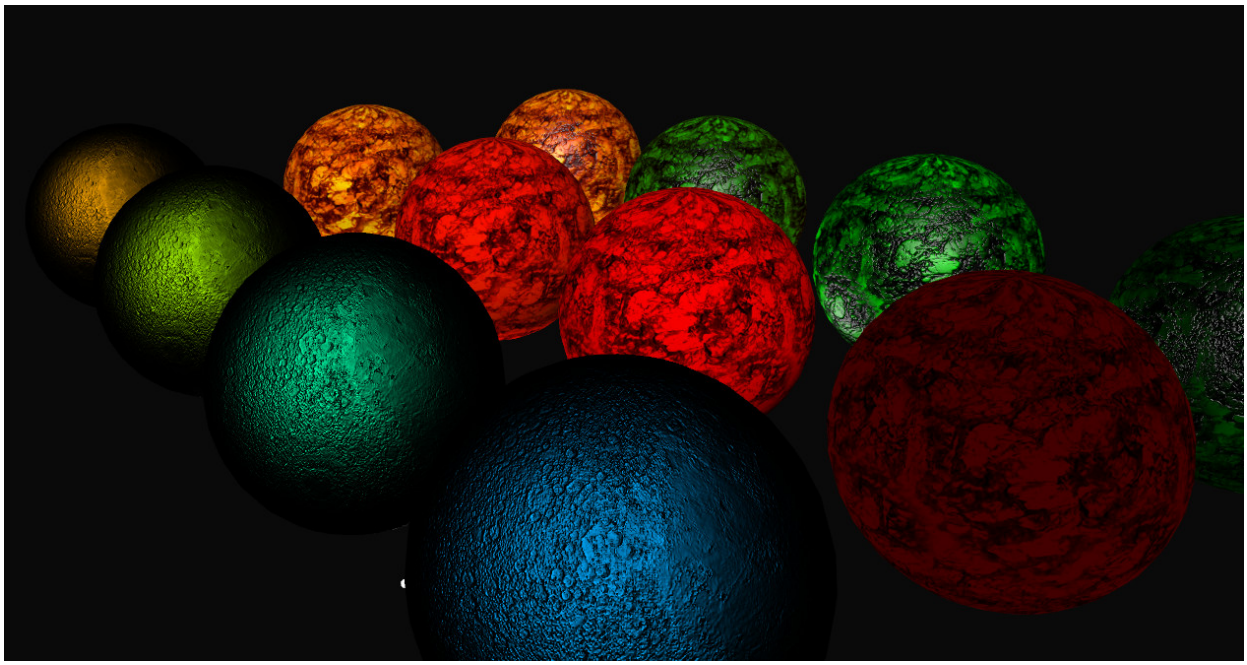
Lesson 8: Textures and Reflections

Lesson: Introduction

[start with http://mrdoob.github.com/three.js/examples/webgl_materials2.html]



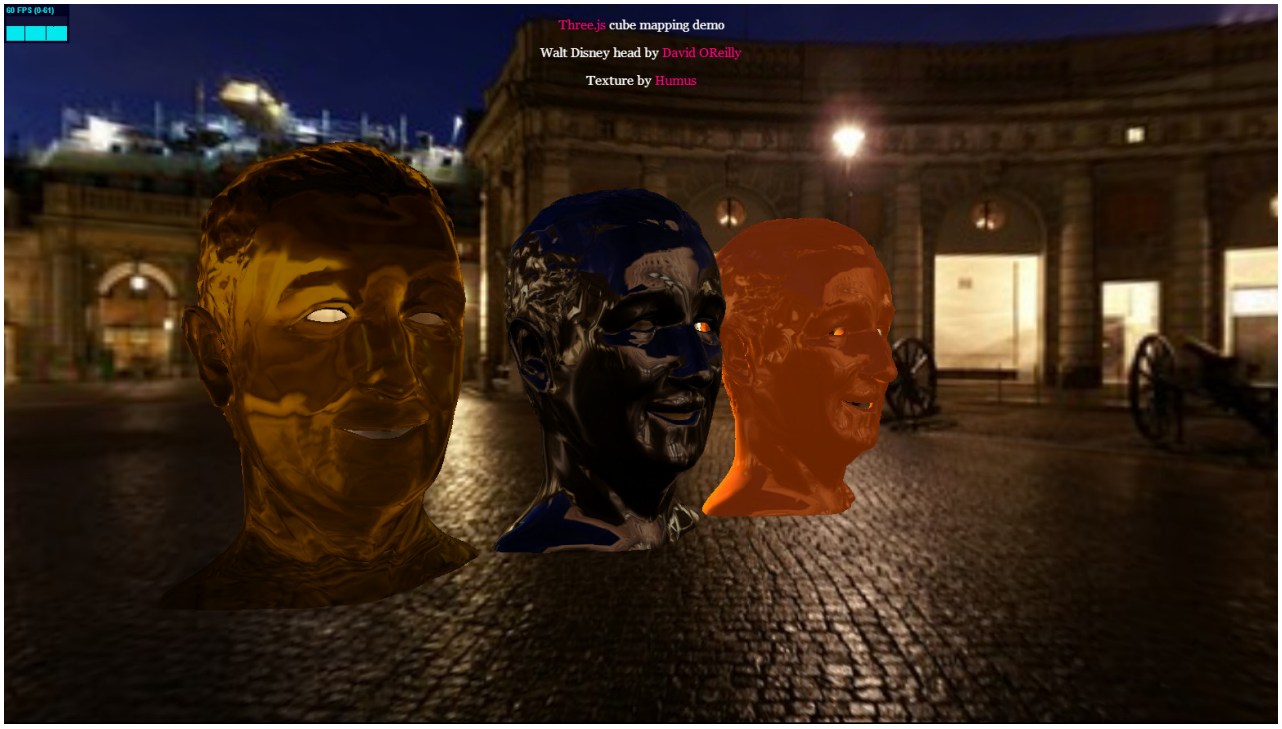
When the word “texture” is used in the real world, it is usually used to mean the feel of cloth or perhaps the roughness of a material. In computer graphics the word “texture” refers to any pattern or image applied to a surface to change its color, shininess, or just about any other part of its appearance.

In one sense, adding a texture to a surface is another form of modeling. You’re specifying the color or roughness or reflectivity of the object based on its surface location.

The most popular way to add a texture to a surface is to use images. Any 2D image can be applied to a surface. In these lessons we’ll delve into how these images are attached to objects,

and the various ways to control their display.

[Slow pan of http://mrdoob.github.com/three.js/examples/webgl_materials_cubemap.html]

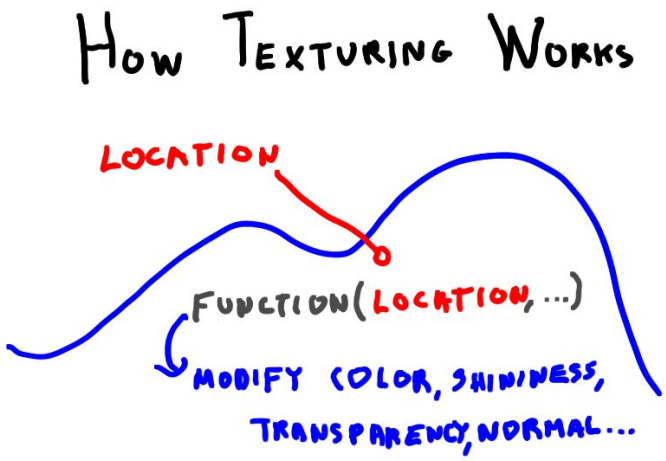


Another important use for textures is to act as surrounding environments. While it can be expensive to attempt to have objects reflect other objects in a scene, it turns out to be extremely fast to reflect textures representing a whole environment. There are some limitations, but surfaces can be made to look much more realistic and visually rich by this simple addition. Even an approximation of refraction can be done with this type of technique.

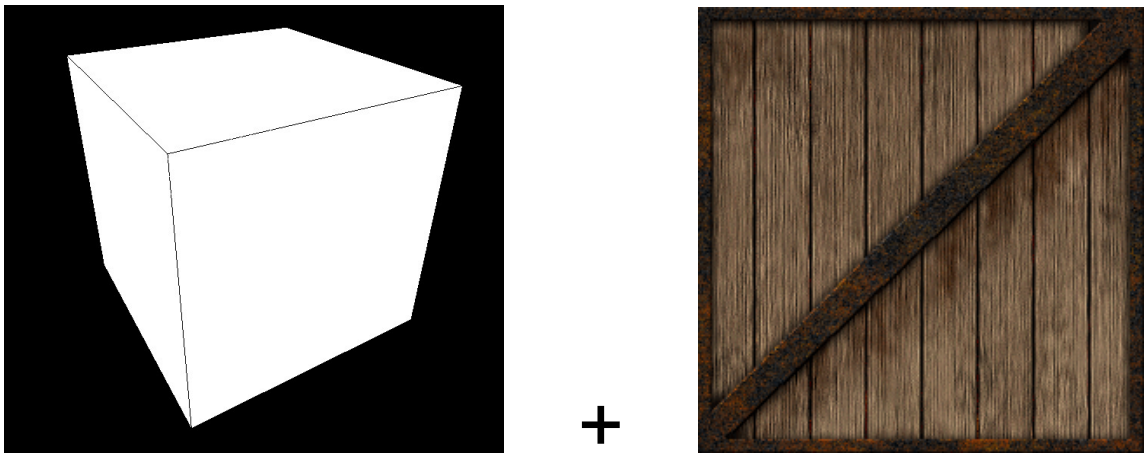
In this unit I'll provide you with the basics of how various forms of texturing works. I'll also be taking you on a tour of some of the major effects achievable with textures. I won't explain all of these in detail. The good news is that code for almost all the demos you'll see are a part of three.js. This gives you the ability to easily prod and poke at these various effects and see what makes them tick. I strongly encourage you to do so - I did this myself in creating some of these lessons and it was a lot of fun.

Lesson: How Texturing Works

[Given a location on a surface and possibly its normal
By using some function based on this location
Modify its color, shininess, transparency, normal, or even the surface height or shape]



At its simplest, texturing works as follows. You take some location on a surface. Given that location, use some function to change the surface attributes at that location. That's all there is to it. What the lessons ahead will cover is what sorts of functions are used, and some of popular ways to modify the surface.



Let's take a concrete example. Well, maybe not concrete - let's make it wood. In fact, let's make

it the most popular object in videogaming: the crate. I looked it up: one site claims there are 740-odd videogames with crates in them, *and counting*. [use James Bond villain voice, *and counting*]

You have some object, such as a box. You'd like to make it look like a crate. The most common way to apply a texture to a surface is to use an image and glue it on, sort of like wallpaper.

What you'd like to get out is this, the crate texture applied to the box.

```
var crateTxr = THREE.ImageUtils.loadTexture( 'textures/crate.gif' );
var material = new THREE.MeshBasicMaterial( { map: crateTxr } );
```

```
var crateTxr = THREE.ImageUtils.loadTexture( 'textures/crate.gif' );
var material = new THREE.MeshBasicMaterial( { map: crateTxr } );
```

In three.js this is amazingly simple. Here's the code snippet that does it. The first line loads the texture. The second line applies this texture to a material. Done. Maybe I should just wrap up this unit with that - nah, let's continue.

For this simple case everything "just worked". Three.js has a reasonable default for applying a square texture to square box sides.

That said, I should give you fair warning about loading textures. There are security concerns enforced with WebGL. The short version is that you can't normally load some texture from

another site in your program. Your program and the texture need to be in the same local space in order to run.

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

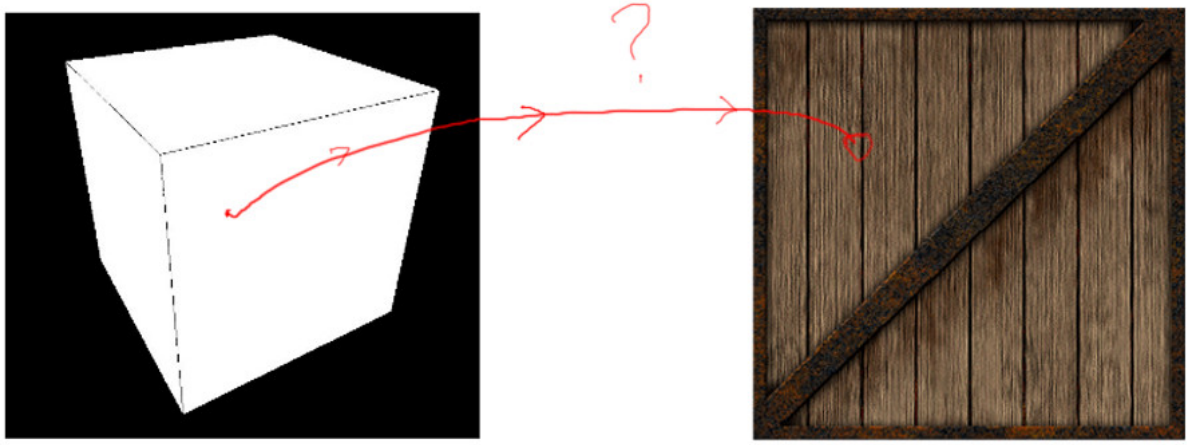
[Additional Course Materials:
For more about WebGL security issues with textures and how to handle them, see [this](https://hacks.mozilla.org/2011/11/using-cors-to-load-webgl-textures-from-cross-domain-images/) and
[this](http://blog.chromium.org/2011/07/using-cross-domain-images-in-webgl-and.html) article.
To run texture demos locally on your browser, you'll have to [set some permissions or set up a local servers](http://www.realtimerendering.com/blog/setting-up-a-windows-server-for-webgl/).

Here's the [list of videogames with crates](http://www.giantbomb.com/crate/93-31/). Another collection of crate screenshots is
[here](http://www.arminbwagner.com/crates_and_barrels/crates.html).

There are a number of places where you can download free textures, such as [Humus' site](http://www.humus.name/index.php?page=Textures), [Open Game Arts' site](http://opengameart.org/textures/all),
[TurboSquid](http://www.turbosquid.com/Search/Texture-Maps/free), and [Autodesk's AREA site](http://area.autodesk.com/downloads/textures).
]

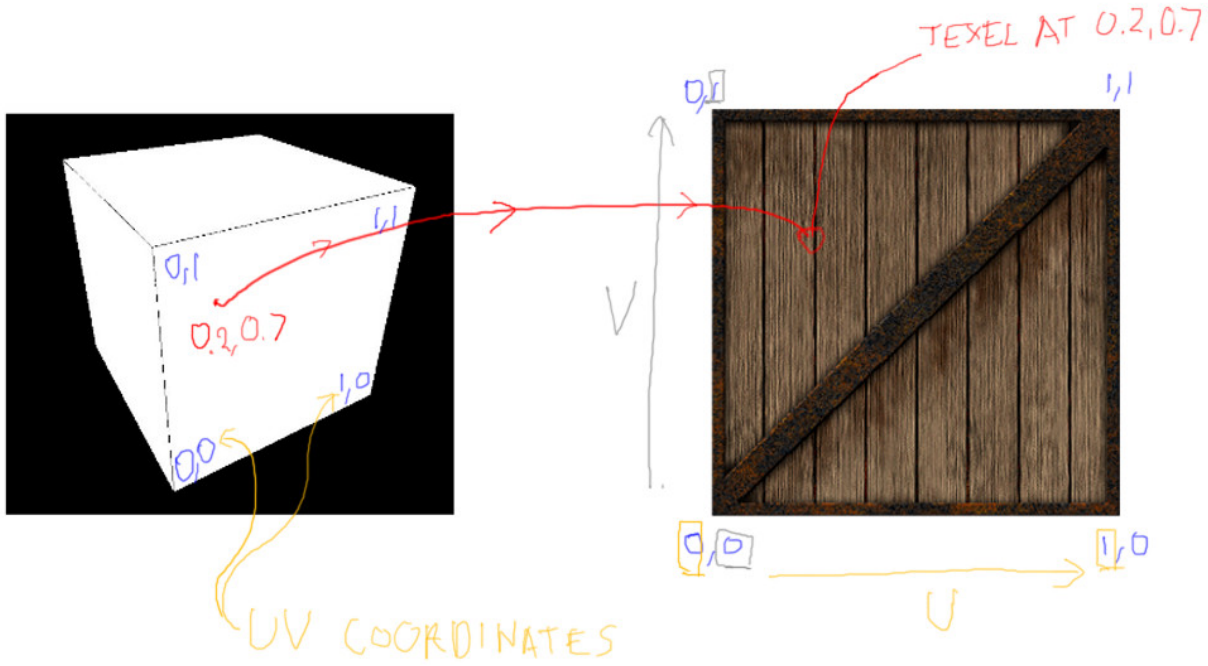
Lesson: Texture UVs

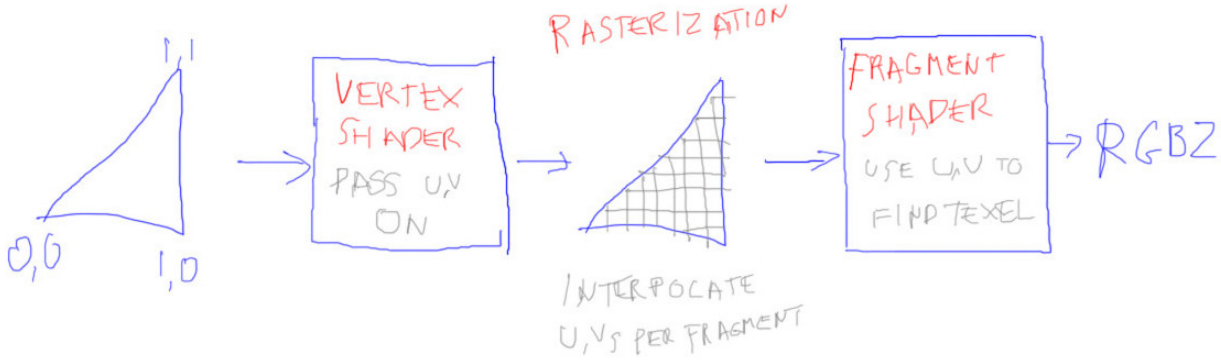
[In final version, put boxes a bit closer, so that there's more room for UV coordinates]



The question is, given a box and an image texture, how do the two get attached? Given a location on the box's surface, how do we find the corresponding location on the image? This is potentially a tricky problem.

Sau we just try to use the location in the world as our input. This won't work well, because our crate could be transformed with rotations or scalings. It might undergo an explosion that would change its shape. Any of these events would modify the world position on its surface, making it hard to get to the same pixel location on the image from frame to frame.





This location problem is most commonly solved by adding two values to every vertex. These are called UV coordinates. Just like any other attribute attached to a vertex, these values are interpolated across each triangle during rasterization. Every pixel fragment generated for the

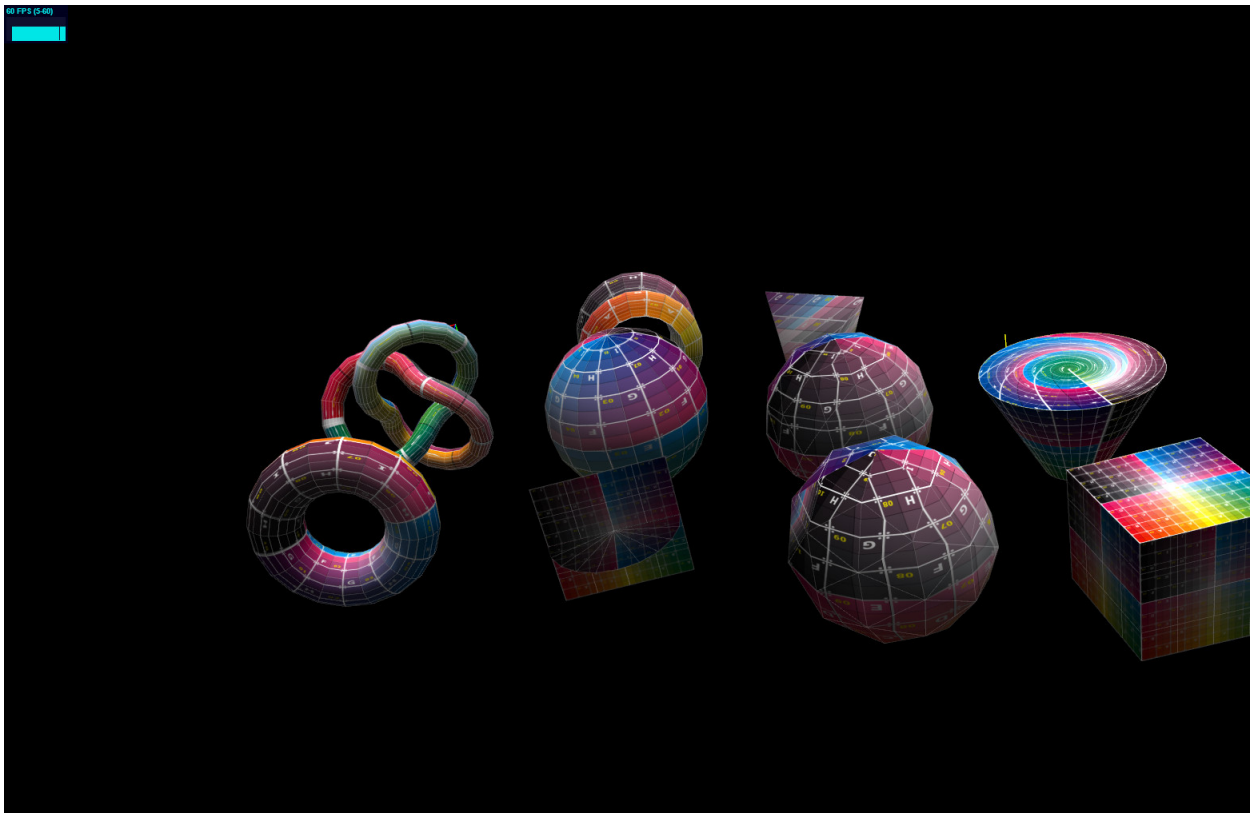
surface will have these values available. Within the fragment shader these two values are used to look up the corresponding pixel location in the texture. A pixel in a texture is often called a *texel*, to differentiate it from a pixel on the screen.

For example, on this face of the box we want to find the texel color to use for this pixel. At this pixel the U,V coordinate turns out to be U equal to 0.2, V equal to 0.7. Using these two values like coordinates on a map, we can look up the texel in the image and use this to modify the color of the surface.

[Additional Course Materials:
For a commented example of how to texture a box, see [Lee Stemkoski's code](http://stemkoski.github.com/Three.js/Textures.html).
]

Lesson: UVs in three.js

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



Many of three.js's geometry objects come along with their own UVs built in. Some of these are

displayed here. For planar objects the texture is put flat against the face. For round objects, the general idea is that the U value goes from 0 to 1 around the equator, and V goes from 0 to one along the axis. To be honest, the mapping on the bottom of the cylinder shown here is pretty, but doesn't make a lot of sense to me - maybe you'll be the one to fix it. Note that three.js truly is a work in progress, and anyone can submit bug fixes and improvements.

I should also point out the tetrahedron. Its mapping is a bit distorted. In fact, if you look at the code, no UV mapping is given at all, and three.js creates one using some rules. Sometimes there simply is no natural UV mapping for the object being made. It's then pretty much up to the programmer to decide how to proceed. What is happening with each of these objects is that the UVs are generated by the geometry class used to create the triangles.

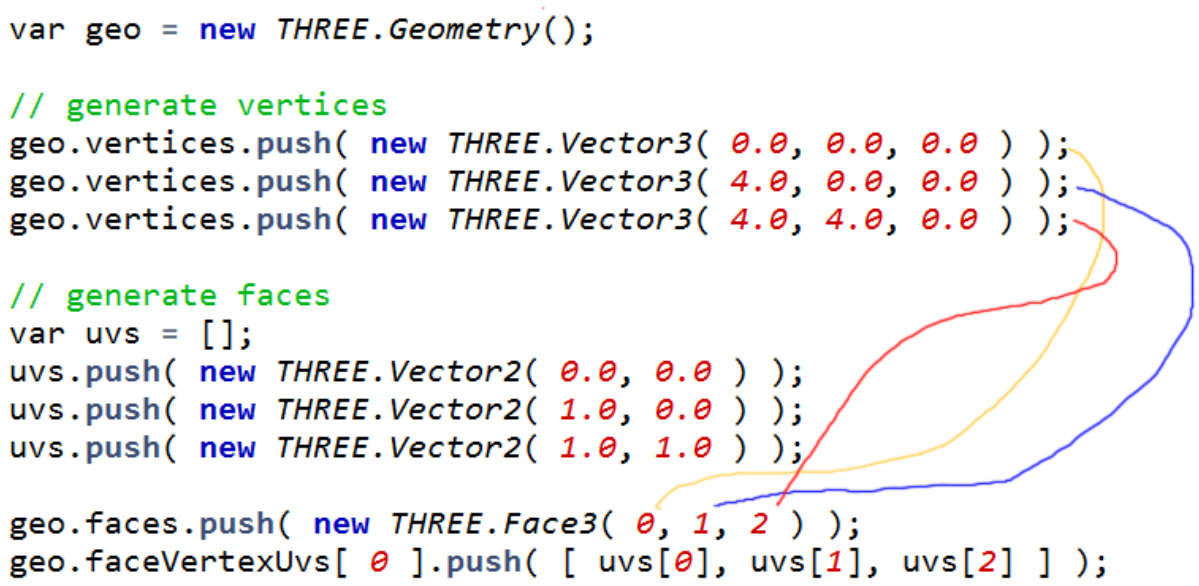
```
var geo = new THREE.Geometry();

// generate vertices
geo.vertices.push( new THREE.Vector3( 0.0, 0.0, 0.0 ) );
geo.vertices.push( new THREE.Vector3( 4.0, 0.0, 0.0 ) );
geo.vertices.push( new THREE.Vector3( 4.0, 4.0, 0.0 ) );

// generate faces
var uvs = [];
uvs.push( new THREE.Vector2( 0.0, 0.0 ) );
uvs.push( new THREE.Vector2( 1.0, 0.0 ) );
uvs.push( new THREE.Vector2( 1.0, 1.0 ) );

geo.faces.push( new THREE.Face3( 0, 1, 2 ) );
geo.faceVertexUvs[ 0 ].push( [ uvs[0], uvs[1], uvs[2] ] );
```

[start with code without the lines]

```
var geo = new THREE.Geometry();

// generate vertices
geo.vertices.push( new THREE.Vector3( 0.0, 0.0, 0.0 ) );
geo.vertices.push( new THREE.Vector3( 4.0, 0.0, 0.0 ) );
geo.vertices.push( new THREE.Vector3( 4.0, 4.0, 0.0 ) );

var uvs = [];
uvs.push( new THREE.Vector2( 0.0, 0.0 ) );
uvs.push( new THREE.Vector2( 1.0, 0.0 ) );
uvs.push( new THREE.Vector2( 1.0, 1.0 ) );

// generate faces
geo.faces.push( new THREE.Face3( 0, 1, 2 ) );
geo.faceVertexUvs[ 0 ].push( [ uvs[0], uvs[1], uvs[2] ] );
```

The form for making a triangle with UVs is this code. Here three vertices are defined and put inside the Geometry object. Three UVs are created; I put them in an array so that I can reuse any that I want.

[now draw the three lines - NOTE, the code image file itself has been fixed, with the comment moved to the correct location further down]



A face is defined as usual: the three numbers given are indices to the three vertex locations in the vertices array. The way the three UVs are attached to this face is a little non-standard. The

first element of the faceVertexUvs is accessed and the three UVs we want to use for this face are directly placed there.

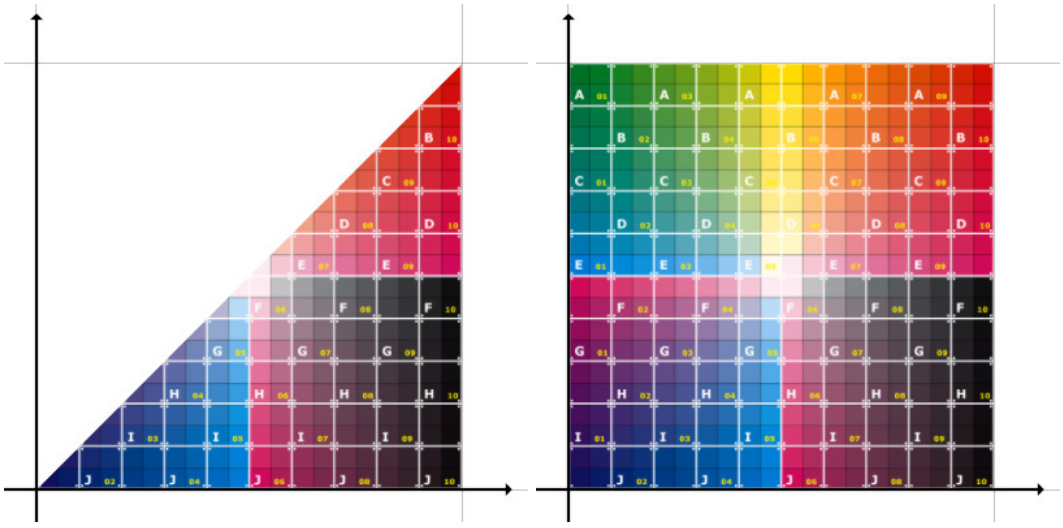
```
geo.faces.push( new THREE.Face3( 8, 2, 6 ) );
geo.faceVertexUvs[ 0 ].push( [ uvs[8], uvs[2], uvs[6] ] );
```

If you want to add more faces with UVs, note that you'll access the faceVertexUVs class the same way each time: you'll access array element 0 and pass in the three UVs desired.

[Additional Course Materials:
Try [the demo shown](http://mrdoob.github.com/three.js/examples/webgl_geometries.html) to see the latest mappings - bugs do get fixed, so it may well be different. If you want to submit fixes to three.js, you can learn about the process [here](https://github.com/mrdoob/three.js/wiki).

To look at the implicit mappings and other texture methods discussed in this unit, definitely try [the online editor](http://mrdoob.github.com/three.js/editor/). Load a directional light, load an object, then apply textures to your heart's content.
]

Exercise: Make a Textured Square



I'm going to start you off with a textured triangle. Your job is to make a textured square, by expanding the ***SquareGeometry*** class implementation.

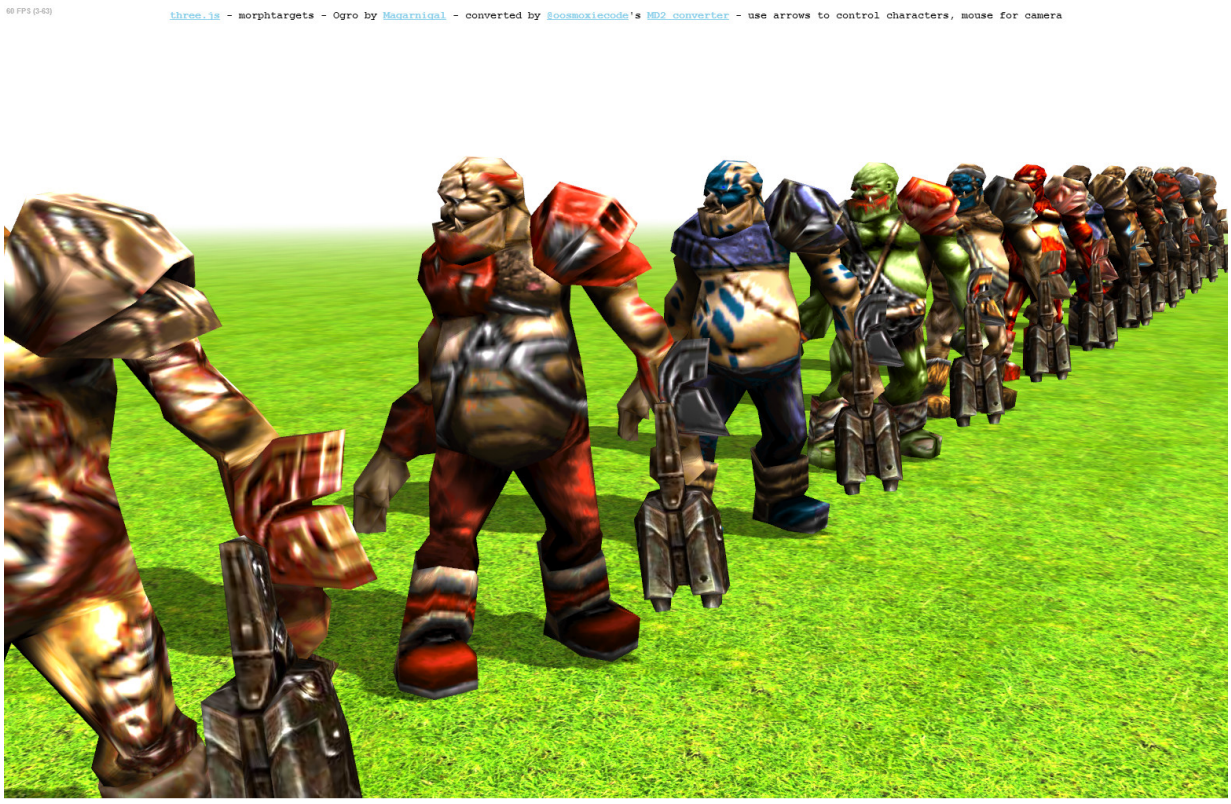
Answer

```
// generate vertices
geo.vertices.push( new THREE.Vector3( 0.0, 0.0, 0.0 ) );
geo.vertices.push( new THREE.Vector3( 1.0, 0.0, 0.0 ) );
geo.vertices.push( new THREE.Vector3( 1.0, 1.0, 0.0 ) );
geo.vertices.push( new THREE.Vector3( 0.0, 1.0, 0.0 ) );

var uvs = [];
uvs.push( new THREE.Vector2( 0.0, 0.0 ) );
uvs.push( new THREE.Vector2( 1.0, 0.0 ) );
uvs.push( new THREE.Vector2( 1.0, 1.0 ) );
uvs.push( new THREE.Vector2( 0.0, 1.0 ) );

// generate faces
geo.faces.push( new THREE.Face3( 0, 1, 2 ) );
geo.faceVertexUvs[ 0 ].push( [ uvs[0], uvs[1], uvs[2] ] );
geo.faces.push( new THREE.Face3( 0, 2, 3 ) );
geo.faceVertexUvs[ 0 ].push( [ uvs[0], uvs[2], uvs[3] ] );
```

The answer involves adding another vertex, adding another UV, and finally adding a face and associating UVs with it. Here is my code.

Lesson: Texture Mapping

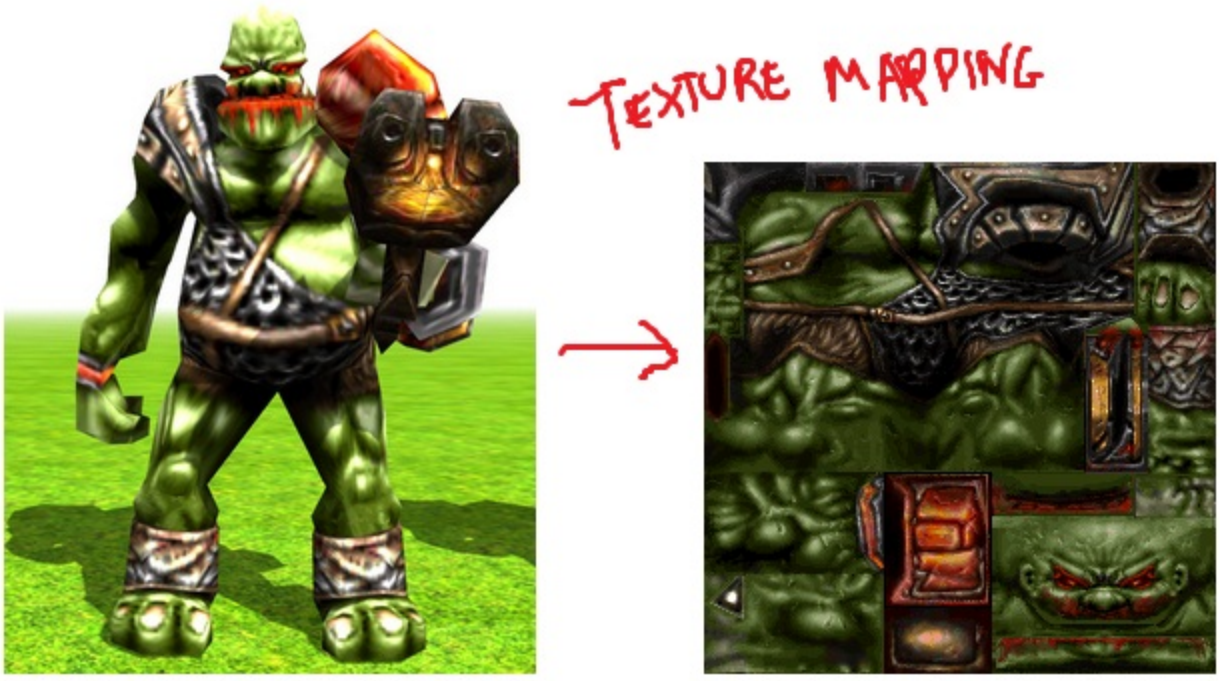
[Start with running
http://mrdoob.github.com/three.js/examples/webgl_morphtargets_md2_control.html]



Objects with UVs and textures can be imported into three.js. I won't be describing this process, as it does not have all that much to do with the principles of computer graphics. See the additional course materials for more information.

What's important to know is how the UVs on an object affects how the texture is mapped to the surface. The basic principle is that, by changing the UVs, we select a different part of the texture.

[Arrow should go in other direction!]



[add label to image on right: ***texture atlas*** ***mosaic***]

The way in which a model is associated with its texture is called ***texture mapping.*** The spheres and other objects we saw before had fairly natural projections of the texture onto their surfaces. For a more complex object such as this humanoid, an artist uses a modeling program to assign the parts of the texture to the model.

When a triangle mesh has a texture applied to it, the texture is used by the whole mesh. Because of this, a single texture is used to hold all the different images for the various parts of the mesh.

This kind of texture is called a texture atlas or a mosaic.

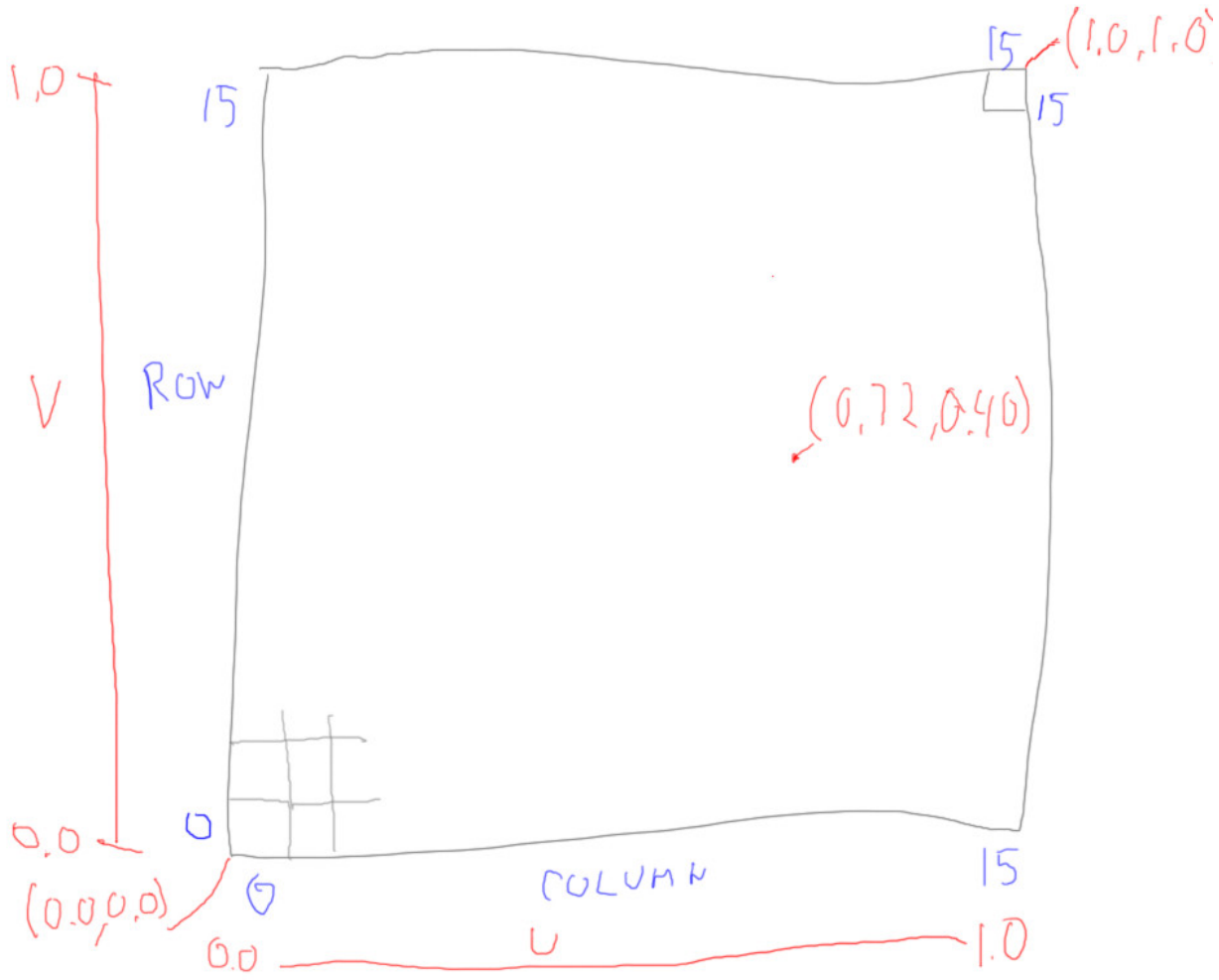
[Additional Course Material:
The character demo is located
[here](http://mrdoob.github.com/three.js/examples/webgl_morphtargets_md2_control.html).

It's worth noting that using mosaicing and putting a large number of objects into a single texture can help avoid changing textures, which can increase performance.
]

Question: Texel Coordinates

[recorded 3/29, part 1]

[drawing: 16x16, label 0,0 and 15,15, show U and V, draw 0.72 and 0.4]



I haven't talked about how to convert from U,V values to texels. That's a formula you can figure out.

We have a texture of size 16x16 applied to a square. The U axis goes along the columns, the V axis goes up the rows. 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, the upper right has 1.0,1.0.

What texel, column and row, is located at the U,V coordinate (0.72, 0.40)?

column = _____, row = _____

Answer

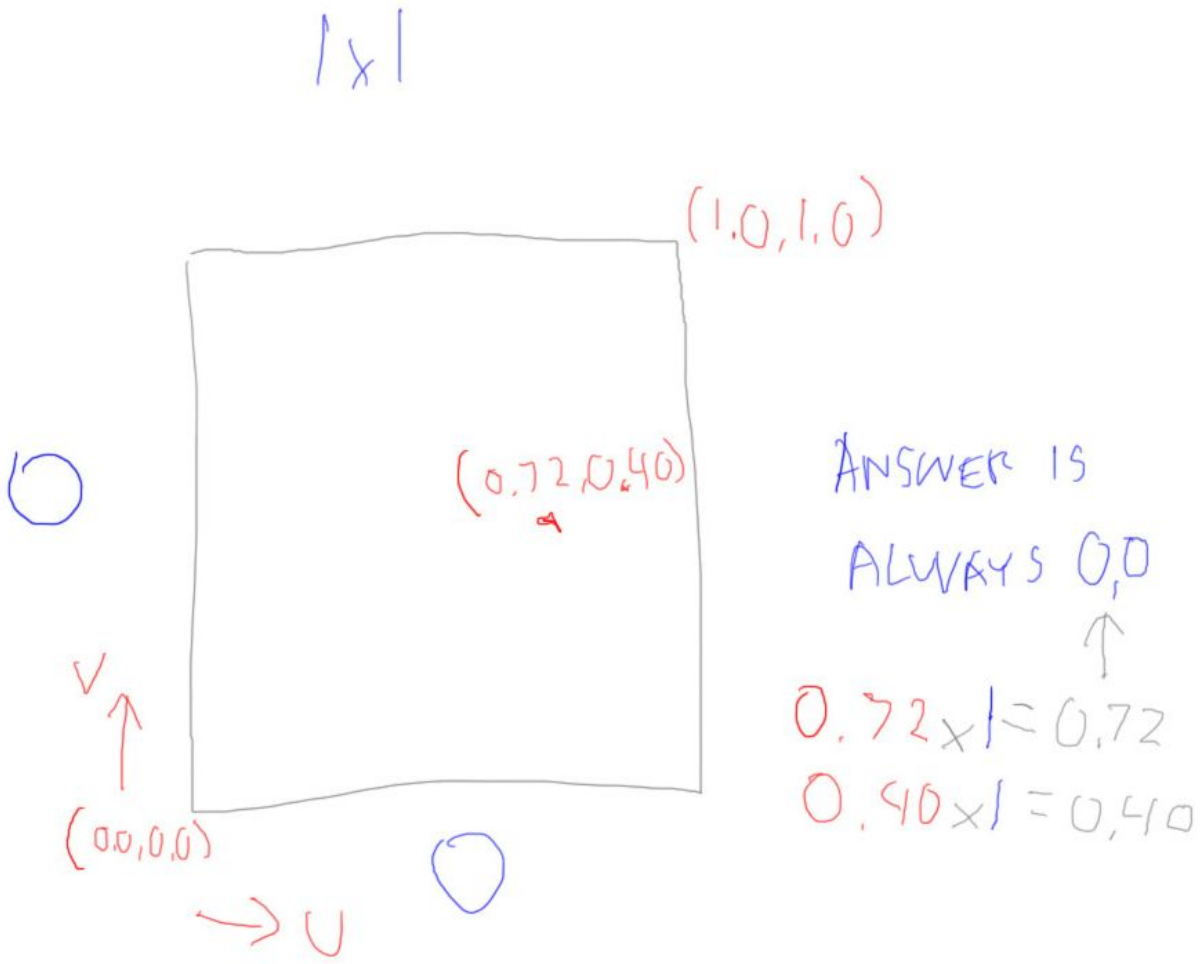
[recorded 3/29, part 1]

column = 0.72 * 16 = 11.52 = 11

row = 0.40 * 16 = 6.4 = 6

Converting is done by multiplying U and V by the number of texels in that direction, 16, then dropping the fraction, not rounding. It's a similar process to how we converted from Normalized Device Coordinates to Window Coordinates.

[show 1x1 pixel example, multiply by 1.]

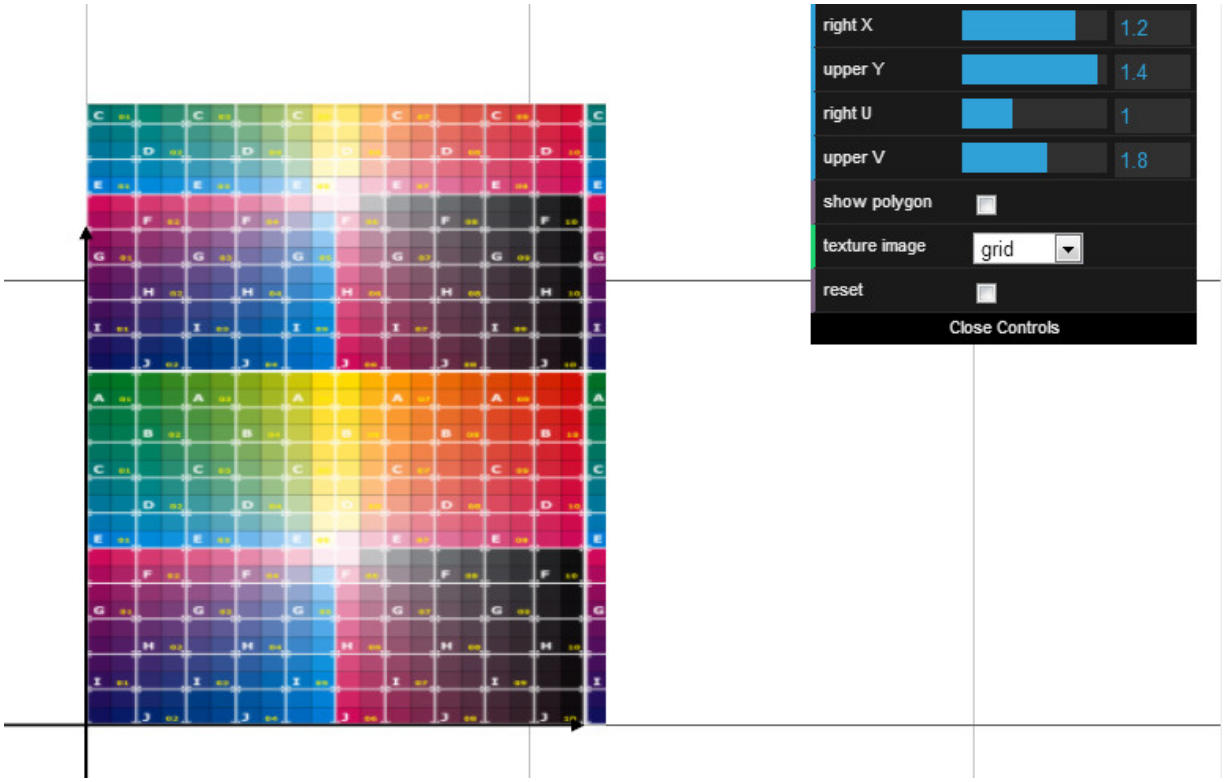


Here's one way to prove it to yourself that you drop the fraction instead of rounding. Try computing the texel location using UVs for a 1 by 1 texture, a texture with one texel. You know in advance the answer you always want for the texel coordinate is the integer 0. A texture coordinate of 0.72 will need to have its fraction dropped to get to this result, as will every other texture coordinate. If you rounded instead, you would get the wrong answer. If the formula doesn't work for a single pixel conversion, we can't expect it will work for larger textures.

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 when multiplied by the resolution of 16 by 16 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.

Lesson: Modifying UV Coordinates

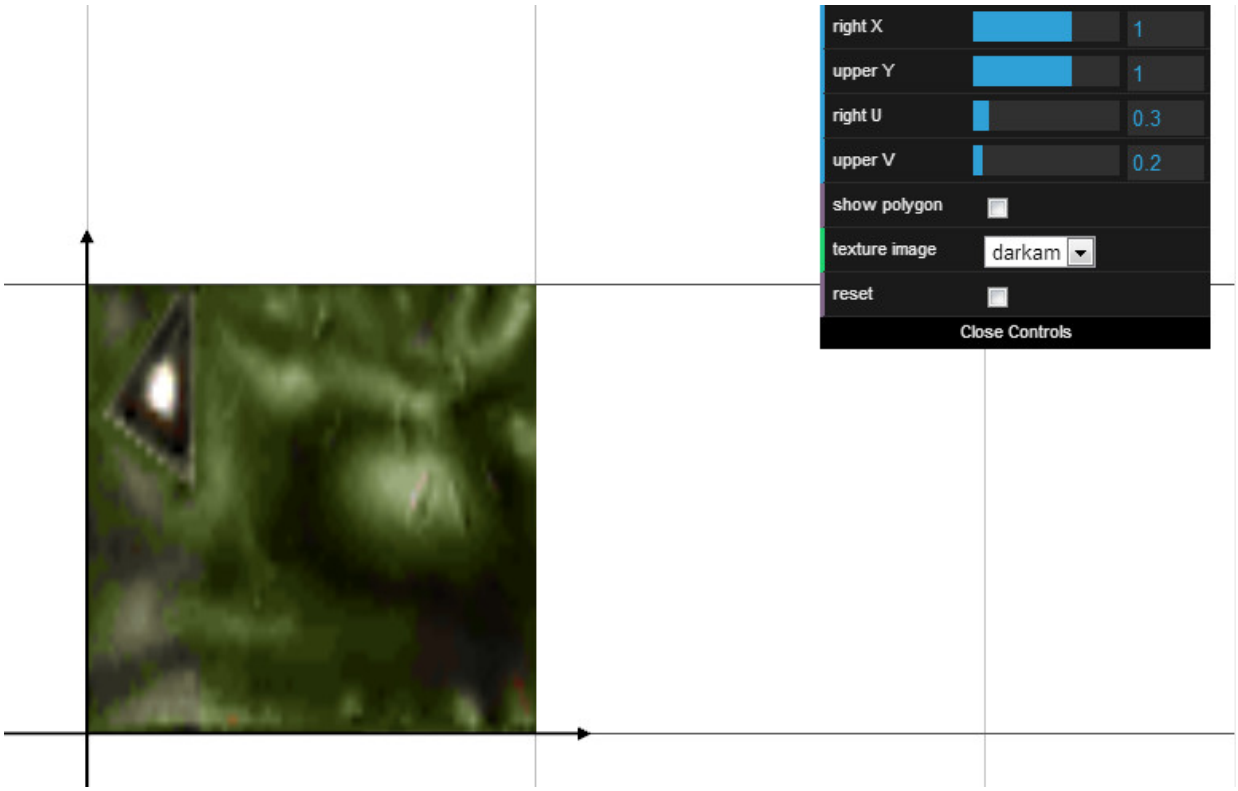
[run unit8-txr_adjust_rectangle.js]



Here's our square with a few slider controls. I can scale the polygon to be larger in either dimension. The texture stretches to fit.

What's more interesting is what happens when I change the U or V values. Here I'm changing the U value of the two points on the right edge. The value starts at 1. If I increase the value, this has the effect of putting more copies of the texture along that axis.

[go to V = 0.2, then U = 0.3]



Let's switch to the mosaic texture. If I decrease the value V, less of the texture is mapped to the square. I can do the same thing with U.

This is how mosaicing works. Each triangle is given some fractional U and V values that selects the piece of the texture needed.

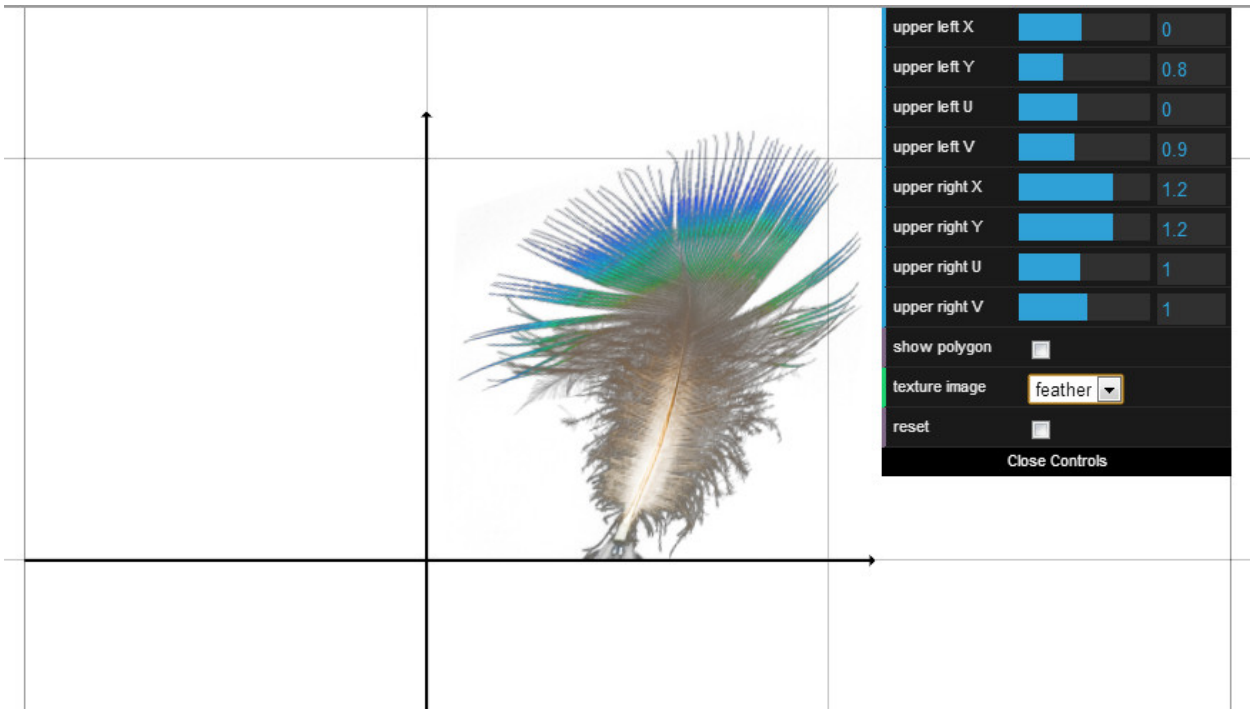
Give this demo a try and get a sense of how the values affect the rectangle and texture.

Demo: UV Coordinates Demo

[Student runs unit8-txr_adjust_rectangle.js]

Lesson: Texture Distortion

[Start with FEATHER with unit8-txr_adjust_upper.js]



Everything has been nice and rectangular so far. We've modified both the polygon and texture coordinates, but in such a way that these both always formed some rectangle. Now it's time to see what happens when this correspondence is broken.

In this new demo we can move the locations and change the positions and UVs of the two upper corners. Let's start by modifying the position of the upper right corner. You can see how this distorts the texture.

[REFRESH THE APP so we can continue safely from a known setting]

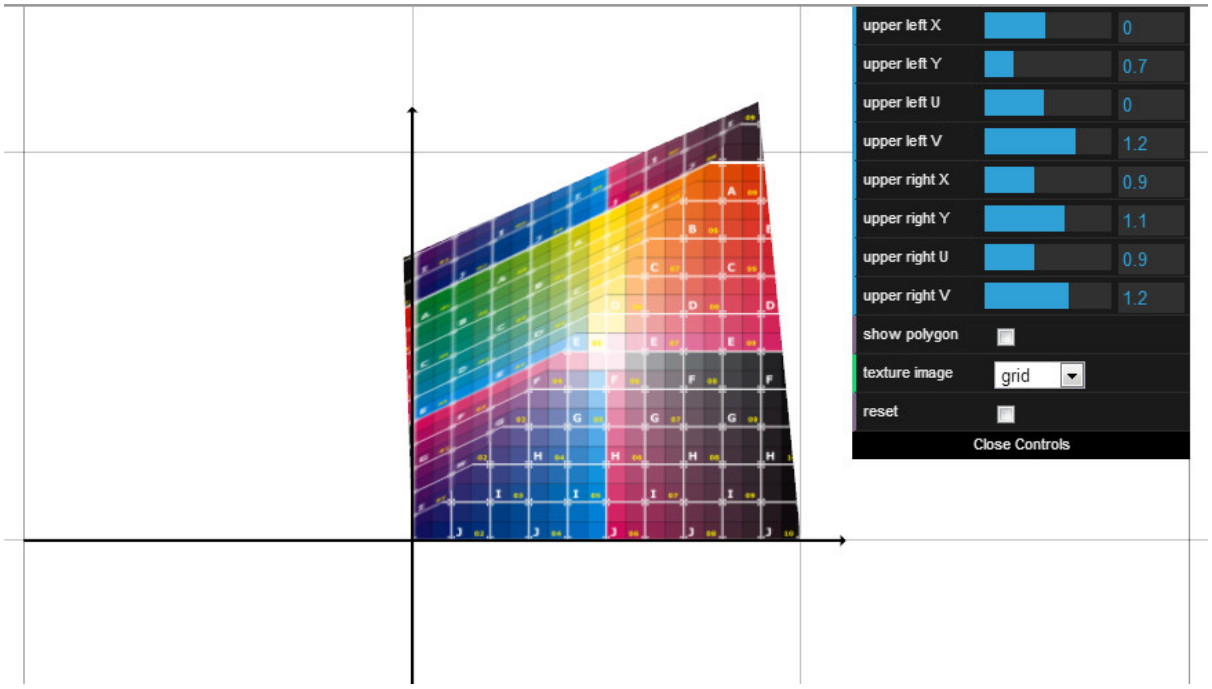
Let's reset the corner back to its starting position. Now let's change the UV coordinates. This also distorts the texture, but in an inverted way. For example, as we make V larger, the texture distorts in a manner similar to what we saw before. An increase of V, instead of a decrease in the Y coordinate, causes this distortion.

Try the demo yourself and see what various changes do. When you're done I'll ask a question about what you saw.

Demo: Texture Distortion Demo

[Run unit8-txr_adjust_upper.js]

Question: What Causes Texture Discontinuities?



When the texture is warped, there is a crease down the middle of our square, from upper right to lower left.

What is the root cause of this discontinuity?
( ) The original texture was stretched or distorted before the corner was moved.
( ) The three.js texture system is broken.
( ) The square is made of a fine mesh of little triangles.
( ) The square is made of two triangles.

Answer

The original texture is just fine, it's a square. Even if it were distorted in some funny way, why the upper left of the texture would be different than the lower right isn't clear.

The second answer is incorrect. Three.js may have some bugs in it, but this is not one of them.

This third answer is interesting. In fact, if the square *was* made out of a fine mesh of triangles, depending on how the triangles were created when the square was warped could make the distortion look more natural.

The fourth answer is indeed the reason.

[switch to view of demo unit8-txr_adjust_upper.js]

If you tried the “show polygon” option in the demo, you’d see the two triangles. Since the square is made of two triangles, moving the corner of the upper left triangle has no effect on the triangle in the lower right.

I highly recommend reading the article that is listed in the additional course materials. It discusses this and a number of other common rendering problems.

[Additional Course Materials:
You can read more about this problem (and many others) in this article, [“It’s Really Not a Rendering Bug, You
See...”](http://www.geekshavefeelings.com/x/wp-content/uploads/2010/03/Its-Really-Not-a-Rendering-Bug-You-see....pdf). This article also shows how meshing can help solve it.

There is one case I know [where triangles are not sufficient to properly render an object](http://tog.acm.org/resources/RTNews/html/rtnv10n2.html#art8), namely, the cone. [There are workarounds](http://tog.acm.org/resources/RTNews/html/rtnv10n3.html#art17), but they’re a bit unnatural.
]

Lesson: Wrap Modes

[D’oh, forgot to do this - Start with lesson saying “rap modes” and then add the “w”]

Modes: gansta rap, nerdcore hiphop, crunk

Erase and say

Modes: repeat, mirrored repeat, clamp to edge
]

There are many different modes of rap, such as gansta rap, crunk, and hyphy.

Um, I mean, there are three main ways within WebGL of repeating a texture. These are called wrap modes.

[Demo is unit8-txr_adjust_wrap.js]

Here's a demo showing them in action. The repeat mode is the one you're most likely to use. It simply repeats each texture one after another. If you want to have things look continuous, such as water, then the texture itself needs to be what is called "seamless", where its edges match up.

If your texture is not seamless, one cheap way to tile it across the plain is to set the wrap mode to be "mirror repeat". This concrete texture is not seamless, I just grabbed a piece of an image on Wikimedia Commons. With "mirror repeat" on, the texture flips on each repetition.

[switch to "R" texture]

This is a bit easier to see with the "R" texture. This wrap mode is not a great solution, it's often clear that the texture is being flipped in this way. Still, it's better than nothing.

The third wrap mode is called "clamp to edge". The drawback, or feature, of this mode is that the pixels on the edge are used to fill in the area where the texture does not appear. This is the default mode for wrapping, as it has some advantages for filtering along the edges when the texture is not going to be repeated.

[
var texture = new THREE.Texture();
texture.wrapS = texture.wrapT = THREE.RepeatWrapping;
texture.wrapS = texture.wrapT = THREE.MirroredRepeatWrapping;
texture.wrapS = texture.wrapT = THREE.ClampToEdgeWrapping;
]

```
var texture = new THREE.Texture();
texture.wrapS = texture.wrapT = THREE.RepeatWrapping;
texture.wrapS = texture.wrapT = THREE.MirroredRepeatWrapping;
texture.wrapS = texture.wrapT = THREE.ClampToEdgeWrapping;
```

This code shows how to set each of the three wrap modes in three.js. The S and T axes essentially mean the U and V axes. You can in fact mix and match these modes, repeating on one axis and clamping on the other.

[Additional Course Materials:

If you want to manipulate the wrap mode separately for each axis, try out [the demo here](http://voxelent.com/html/beginners-guide/1727_07/ch7_Texture_Wrapping.html) from the book "[WebGL Beginner's Guide](http://www.amazon.com/WebGL-Beginners-Guide-ebook/dp/B008CEMBPI?tag=realtimerenderin)". Another wrap mode example of using repeat is in [Lee Stemkoski's code](http://stemkoski.github.com/Three.js/Texture-Repeat.html).

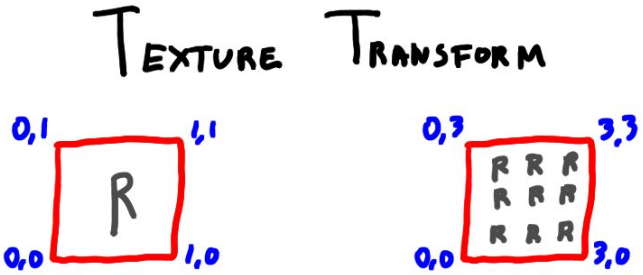
Note that wrap modes in WebGL [will only work for textures that are a power of two](http://www.khronos.org/webgl/wiki/WebGL_and_OpenGL_Differences).
]

Demo: Wrap Mode Demo

unit8-txr_adjust_wrap.js

[Additional Course Materials:
If you change the wrap mode, you need to set "texture.needsUpdate = true;". See the demo code for an example.
]

Lesson: Texture Transform



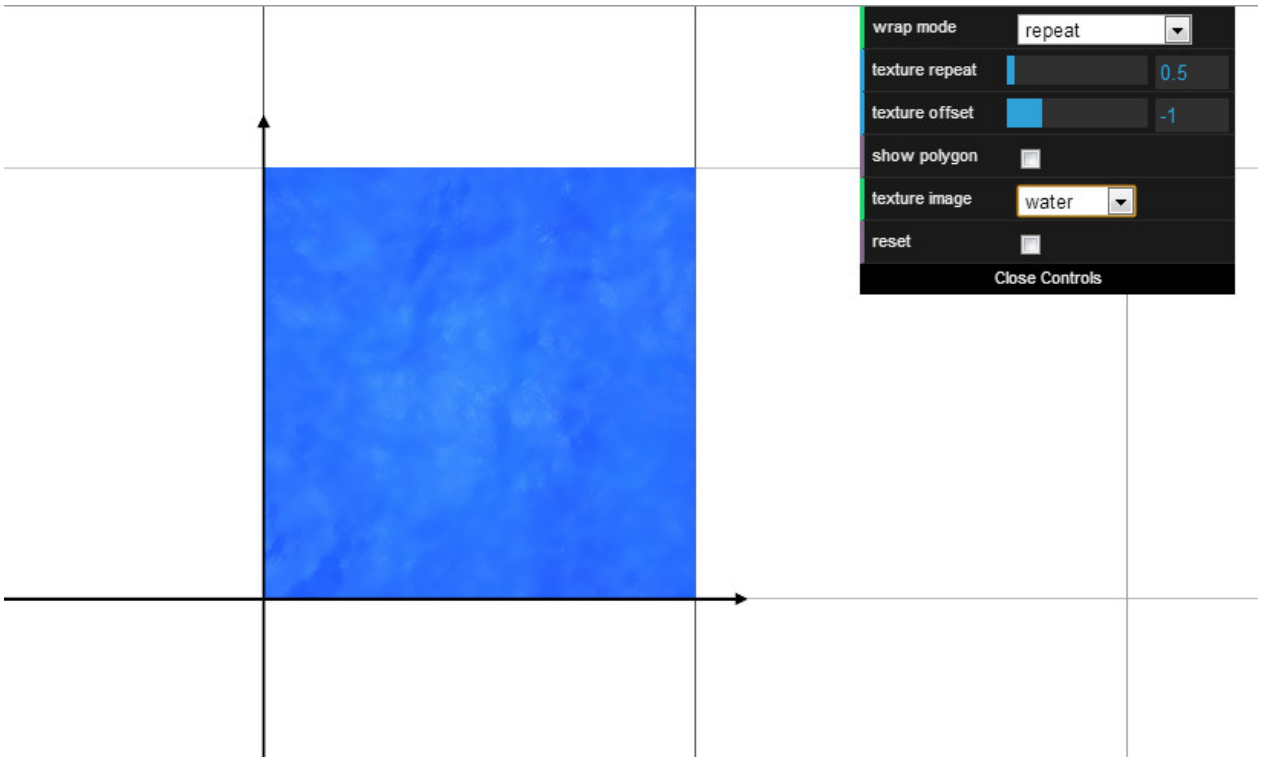

```
var texture = new THREE.Texture();
texture.repeat.set( 1, 1 );
texture.offset.set( 0, 0 );
```

```
var texture = new THREE.Texture();
texture.repeat = new THREE.Vector2( 1, 1 );
texture.offset = new THREE.Vector2( 0, 0 );
```

One way to change how many times a texture repeats is to modify the UV coordinates of the square. This is a bit intrusive, you have to modify the geometric mesh itself.

Three.js has an alternate solution. It allows you to set two parameters on a texture, called repeat and offset. These define a transform for the texture coordinates themselves. Repeat is essentially a scaling operation, saying how many times a texture should repeat across the surface. Offset is a translation, added to the UV coordinate after scaling.

[unit8-txr_adjust_wrap.js]



Here you can see the effect of changing the repetition. If you repeat a texture too much it becomes fairly obvious that it's being repeated. There are advanced solutions such as using Wang Tiles to break up this repetition. See the additional course materials for more on this topic.

Varying the offset shifts the texture. One use of this parameter is that you can animate a texture. For example, if you have a river, you can make the water flow by slowly changing one texture coordinate over time.

[Additional Course Materials:
If you want to know more about advanced color tiling, read about [Wang Tiles](http://procworld.blogspot.com/search/label/Wang%20Tiles).

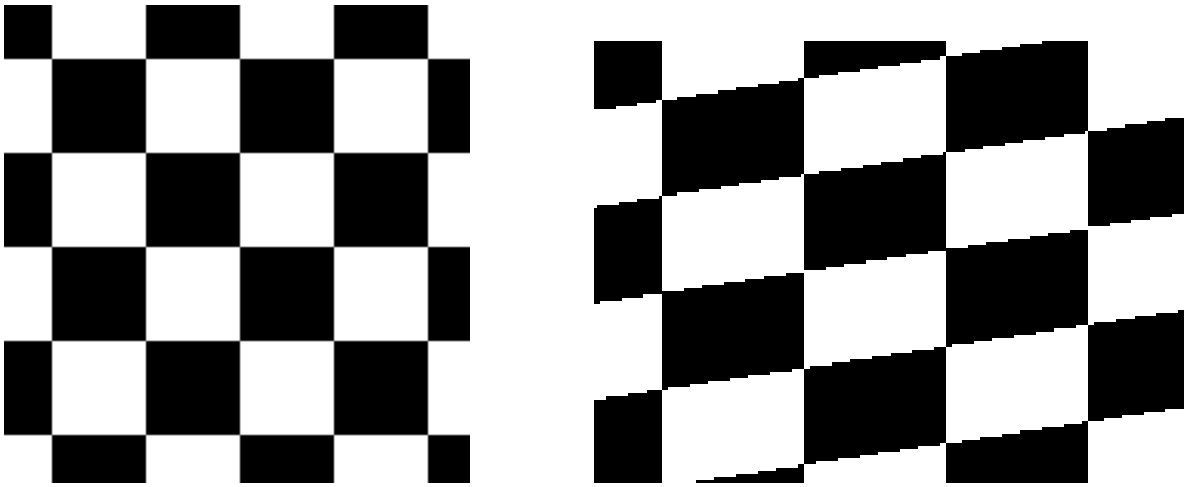
Minecraft now allows all textures to be animated, [leading to madness](http://www.youtube.com/watch?v=1tm8K3n_Tps).
]

Lesson: Texture Magnification

[show a pixel sampling the texture 2x2 checkerboard. note nearest neighbor. Excuse the resolution, I don't want to redraw]

[SAVE THIS IMAGE IN LAYERS! We'll reuse]

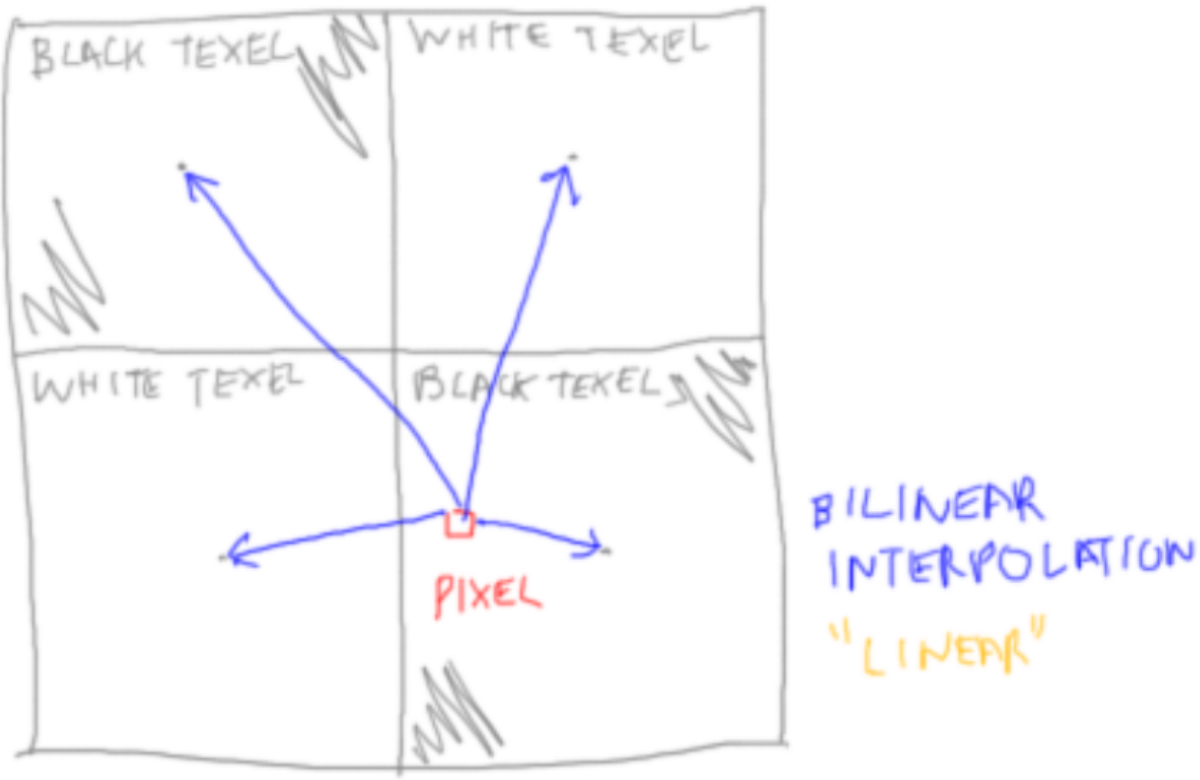
Say you have a very simple texture pattern, a checkerboard consisting of just 4 texels. What happens if you put this texture on a square? There are now many pixels all sampling the same texel. The obvious answer is that you want this texture to continue to look like a checkerboard, with a sharp edge between the squares.



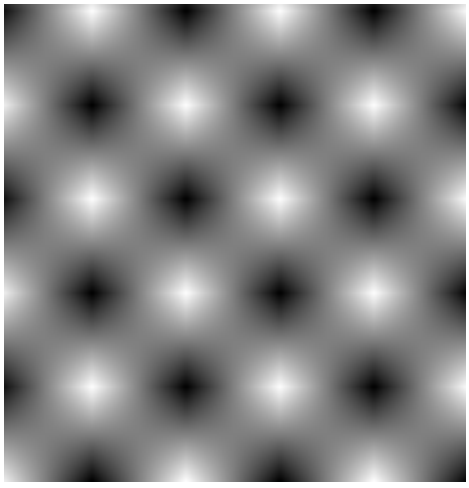
In other words, you want something like this. However, if you view this texture from a different angle, it doesn't look so great. This is a zoomed-in view of the texture tilted. You get this stair-stepping effect instead of nice smooth edges. The problem is that the texture will return either black or white at every pixel. You'd like 50 shades of gray or so, but there's only these two levels.

Sampling the texture in this way is called "***nearest neighbor***". Whatever texel center is closest to the pixel's center is the color the pixel gets.

[same drawing, but show bilinear]



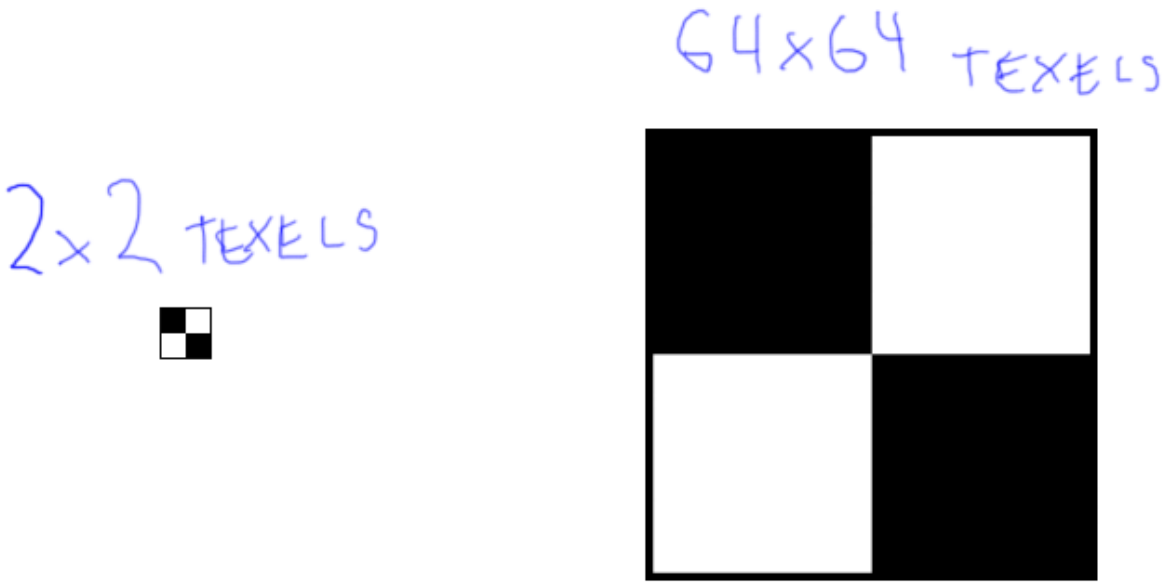
The other option is “bilinear interpolation”, or “linear” for short. What this mode does is take the four closest texel centers and interpolates among them. The closer the pixel is to a texel’s center, the more of that color you get.



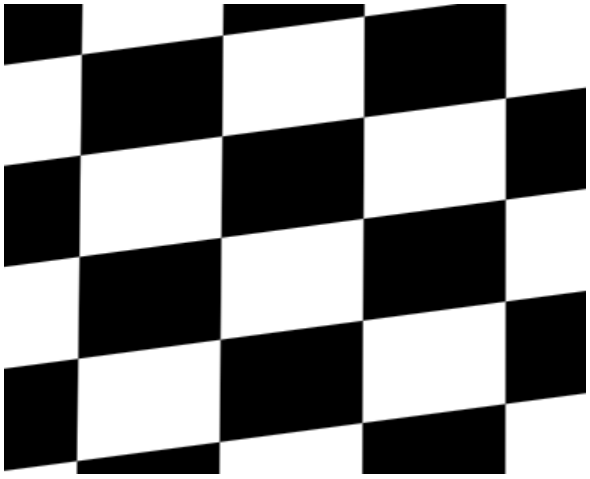
If you turn on bilinear interpolation, this is what you get. That’s definitely too much of a good thing. You’d like blurring along the edges of the squares, not across the whole texture.

The problem here is that one texel covers a huge number of pixels. Let's say we aim to change this ratio to be about 1 texel per pixel. In other words, say our original 2x2 checker pattern covers 100x100 pixels.

[import bigger_texture.png, erase letters, rewrite]



To bring the ratio of texels to pixels to be closer, let's make the 2x2 checker pattern be represented by an image texture of 64x64 texels, so that each checker square itself covers 32x32 texels. In other words, we'll just make the same pattern but with a larger texture. Each texel will now cover around one pixel or so per texel.



This turns out to look pretty good, the stair steps are gone. You'll try the demo for yourself in a

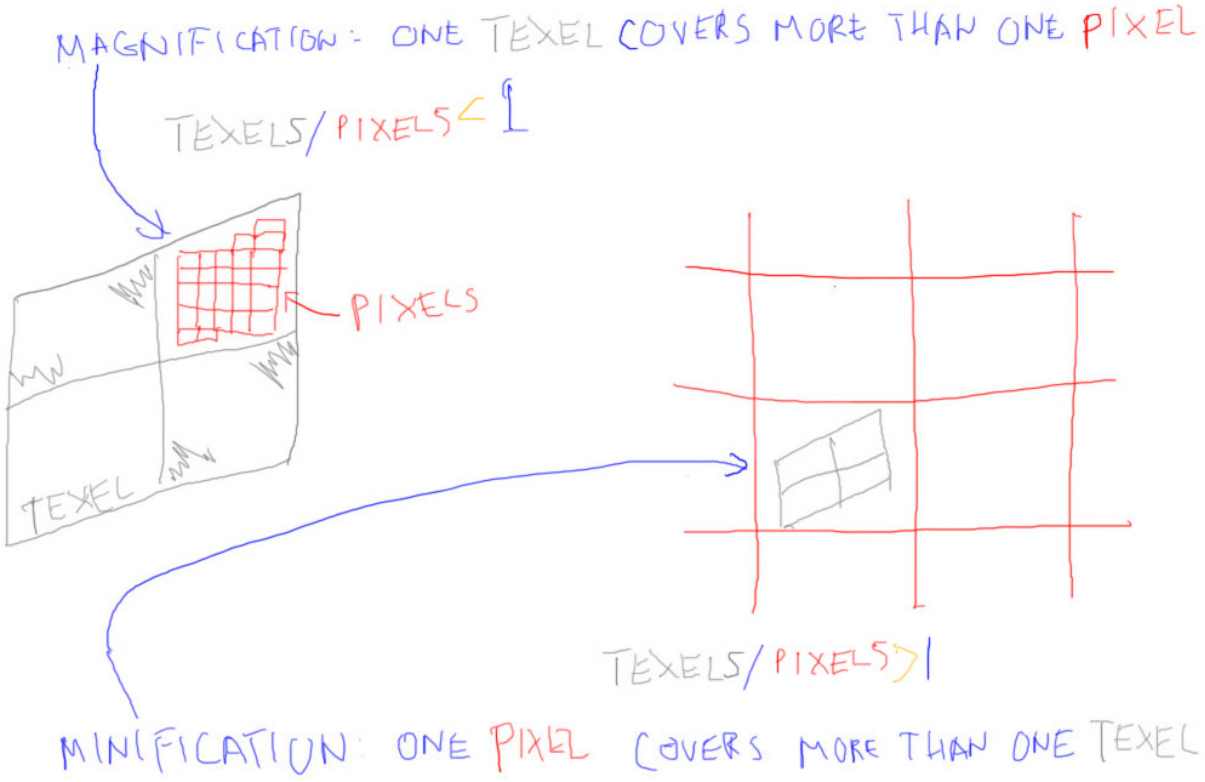
minute and see for yourself.

```
var texture = new THREE.Texture();
texture.magFilter = THREE.NearestFilter;
texture.magFilter = THREE.LinearFilter;
```

```
var texture = new THREE.Texture();
texture.magFilter = THREE.NearestFilter; // one "tap"
texture.magFilter = THREE.LinearFilter; // four "taps"
```

In three.js setting the filtering to be “nearest” or “linear” is straightforward, there’s not a lot to it. Using bilinear interpolation causes more “taps” to occur, that is, more texture samples to be retrieved, but it’s usually what you want. Generally, you should use linear, which is the default. At the same time, you want to avoid situations where a texel covers a lot of pixels.

[draw picture of checkers and pixels. KEEP THIS DIAGRAM AROUND! We reuse many times]



When we have a situation where a texel covers more than a pixel, this is called magnification.

When magnification is occurring, the texture's magFilter is used. Another way to say it is that if the number of texels divided by the number of pixels is less than 1, magnification is happening. You're magnifying the texture.

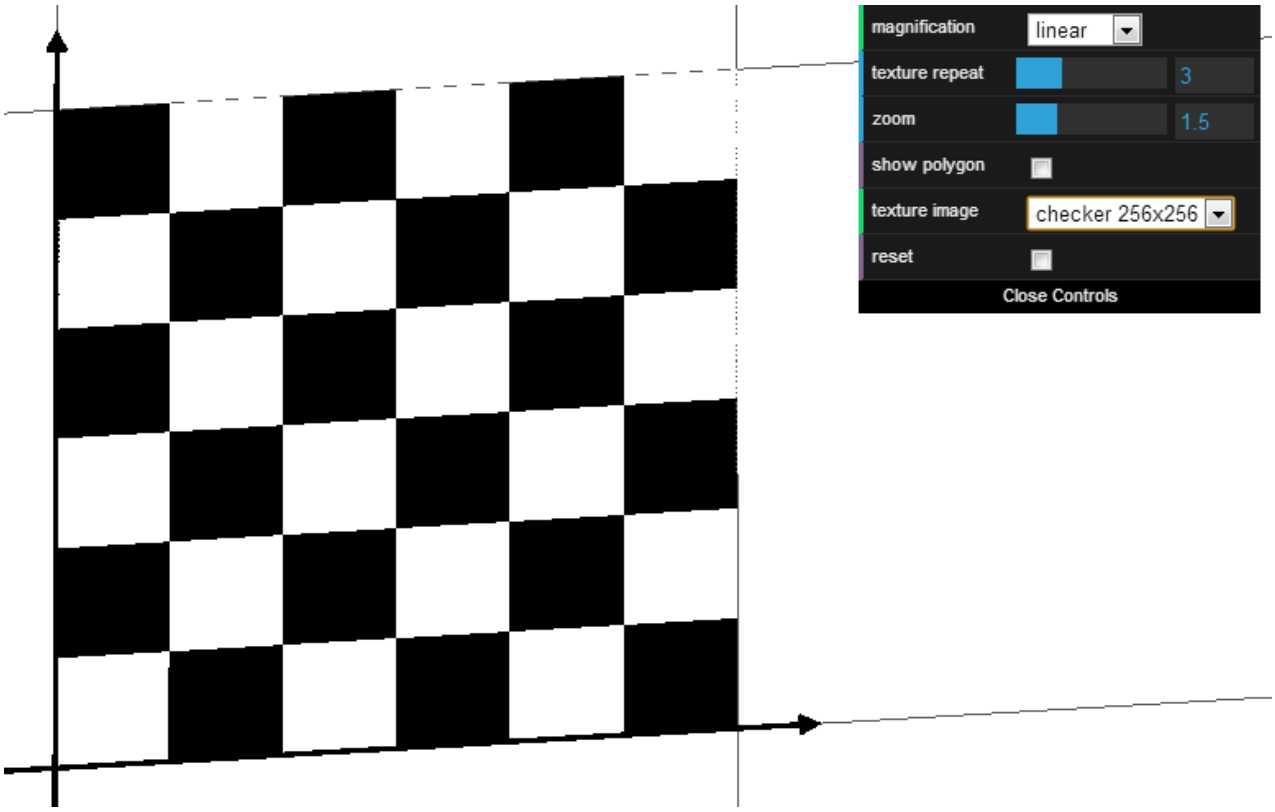
If a texel covers less than one pixel, that is, one pixel contains more than one texel, this situation is called minification. In this situation the magFilter is not used. We'll deal with minification and how to set the corresponding minFilter in the next lesson.

At this point I want you to study a demo, as I'll be asking you about it. Try out various resolution textures and see how they work with nearest neighbor and linear filtering. Try the higher resolution checker patterns and try tilting the view to see what happens, then play with the zoom.

Demo

[unit8-txr_magnify.js]

Lesson: Smooth to Sharp



[Run this demo. Note that I do *NOT* have to sync my talking with the demo, just that the demo should be long enough to finish.

DIRECTIONS: Show tilted unit8-txr_magnify.js,, increasing res: 2x2 to 8x8 to 64x64 to 128x128, until jaggies appear, then zoom in until they disappear. Slide zoom a little between each level.]

Say you tilt the view a little and set the linear filter. As you increase the resolution of the texture, the checker edges get less and less blurry. At some point the edges become sharp. If you now use the mouse wheel to move in towards the texture, at a certain distance the edges suddenly become smooth again. This may not show up on the video, and the texture size it happens for you may vary, but you should see the edges suddenly get sharp. Go back to the demo and try it yourself.

Question: Why Smooth to Sharp?

Why do the edges jump from smooth to sharp? What is causing the edges to become sharp again? More than one answer is correct, so check all that apply.

[ ] Minification is occurring.

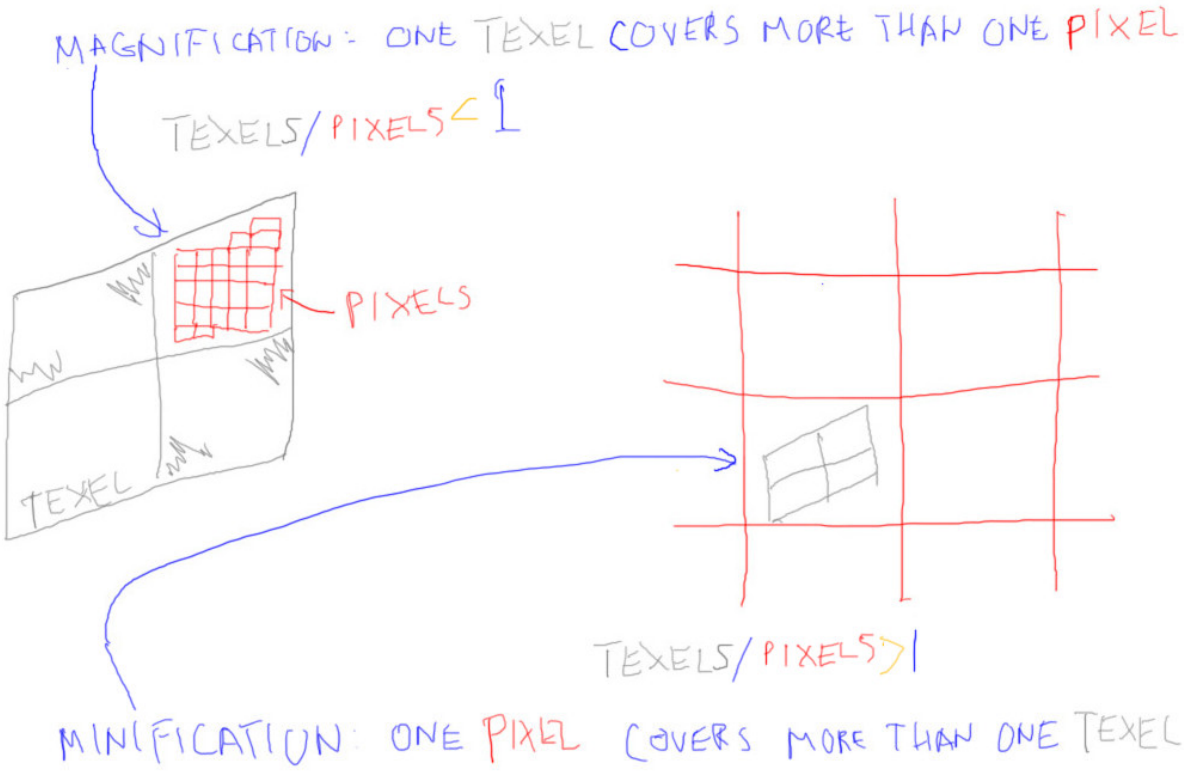
[ ] Linear interpolation is no longer being used.
[ ] On average, each pixel covers more than one texel.
[ ] The magFilter no longer applies.

Answer

The answer is that all of these are correct. When a pixel begins to include more than one texel, minification is occurring. Under these conditions the magFilter no longer applies, so its Linear filter is not being used.

[
If we don't like all these answers being correct - I do! - we could put
[ ] The GPU cannot handle this high contrast data and fails.
as the wrong one]

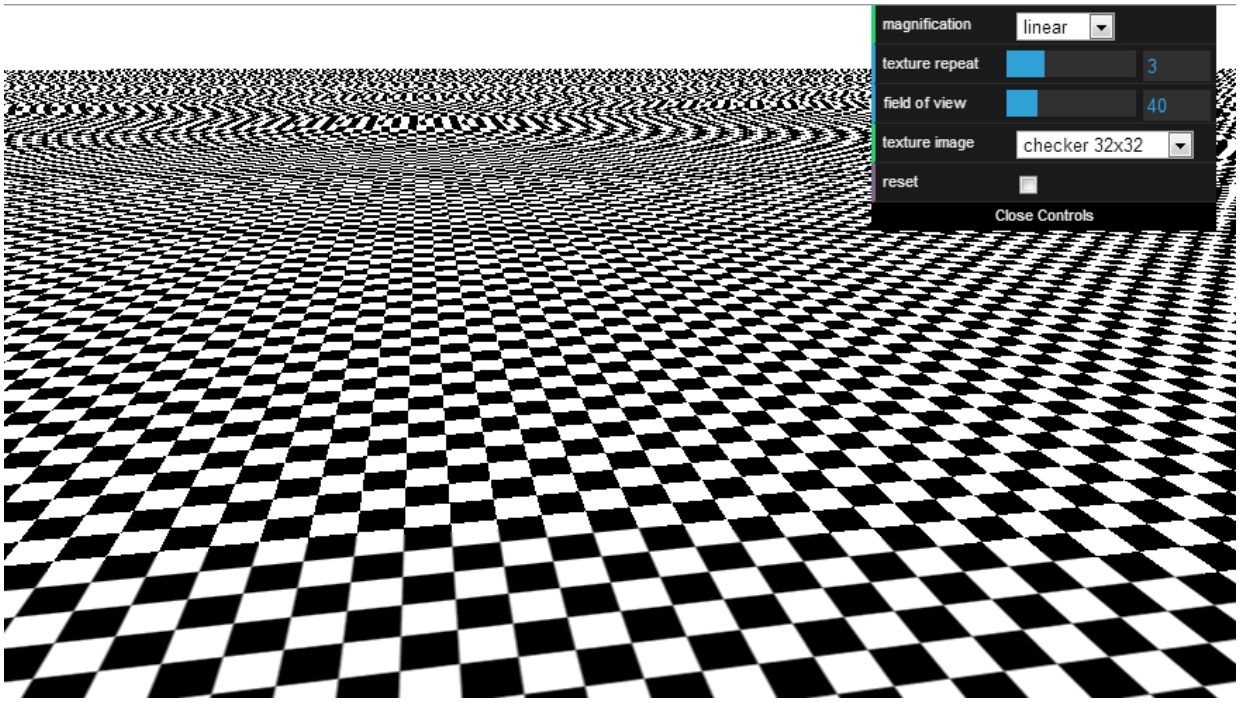
Lesson: Minification



In general we'd like the ratio between texels and pixels to be around 1 to 1. If the texels cover too many pixels, we'll get blurring, as we saw in the previous lesson. This isn't so bad for real textures, so erring a bit on the side of smaller textures is not the worst crime you can commit.

If a pixel covers too many texels, we get minification, a different problem. For a high resolution checker texture it simply caused the edges to become sharp. However, the problem is much worse than this under normal viewing conditions.

[unit8-txr_minify.js]



Here's the checkerboard again. You can see horrible noise and Moire patterns in the distance. Changing the texture resolution does not help.

When a pixel covers more than one texel, minification occurs. This is what's happening in the distance. There is a separate filter on the texture for this situation. The reason it's separate is that this problem has a different solution.

[turn on bilinear for minification]

Currently this minification filter is set to be "nearest neighbor". As we saw for magnification, this is usually a bad choice. Let's turn on bilinear interpolation for the minification filter, too. This helps a little bit towards the middle of the image, but not much. The problem is that each pixel in the distance covers a few texels. As we get closer and closer to the horizon, the number of texels per pixel increases. When the fragment shader looks up what is in the texture, it gets whatever texel happens to be at the center of the pixel. At the horizon this selection is almost random, so we get noise as we vary between white and black texels. Even with bilinear interpolation on it's still just noise.

With magnification we found that too many pixels covering one texel caused blurring. Getting this ratio closer to 1 to 1 made things just right. Here we have too many texels in one pixel.

One solution is to go in the opposite direction that we went for magnification. Instead of making our checkerboard have a larger and larger resolution, we'd like it if the distant pixels used a lower resolution texture. This would bring the ratio of texels to pixels about back to 1 to 1.

[go to 16x16 8x8 4x4 2x2, then jump back to 32x32]

I can show this by lowering the resolution of the texture. I'll say it again: the video may not show much difference, as these videos are heavily compressed so that they're quicker to download. Your best bet is to try this demo out for yourself. It follows this lesson.

Anyway, changing the resolution helps a bit towards the middle of the image, since these pixels have more and more texels as we move to the horizon. Blurriness is much better than noisiness, especially when moving the view around.

Things still look terrible on the horizon, because every pixel in that area covers so many texels that even our 2x2 texel image fails. If we go to a 1x1 checkerboard pattern, that might help a bit. What in the world is a 1x1 checkerboard? Well, it's just a single checker square, a single texel. Since I can't put both black and white in this texel, I make this texel gray.

[go to 1x1]

Choosing this texture is like the nuclear option. The horizon looks better than noise, but now *everything* is gray. Basically, at some point near the horizon there are just too many texels to pixels, so to avoid noise we go to gray. Try the demo yourself at this point and see how changing the texel to pixel ratio affects the image.

Demo: Minification Demo

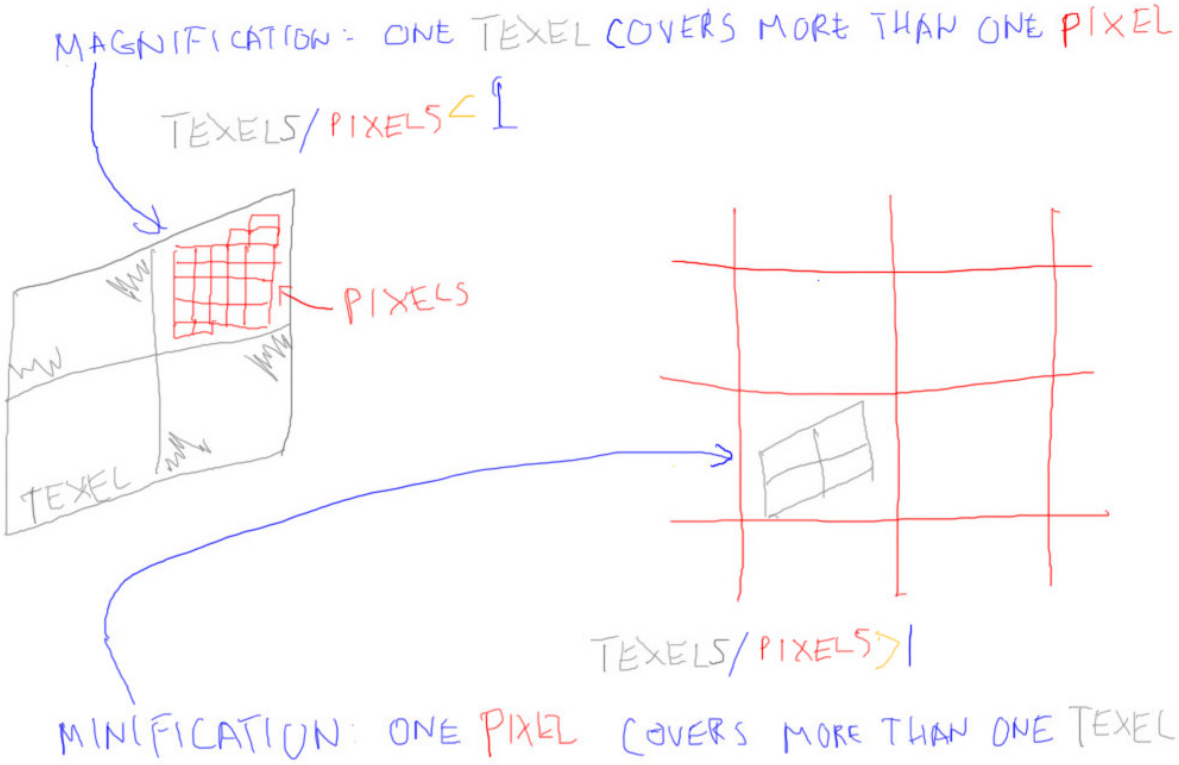
[Use the ***minify*** demo unit8-txr_minify.js (the other two demos expose more options, don't want them to see these yet).]

[Additional Course Materials:
Try different texture resolutions to see where minification takes place. Change the minification to "nearest" to see a small improvement, but more will need to be done... in the next lesson.
]

Lesson: Mipmapping

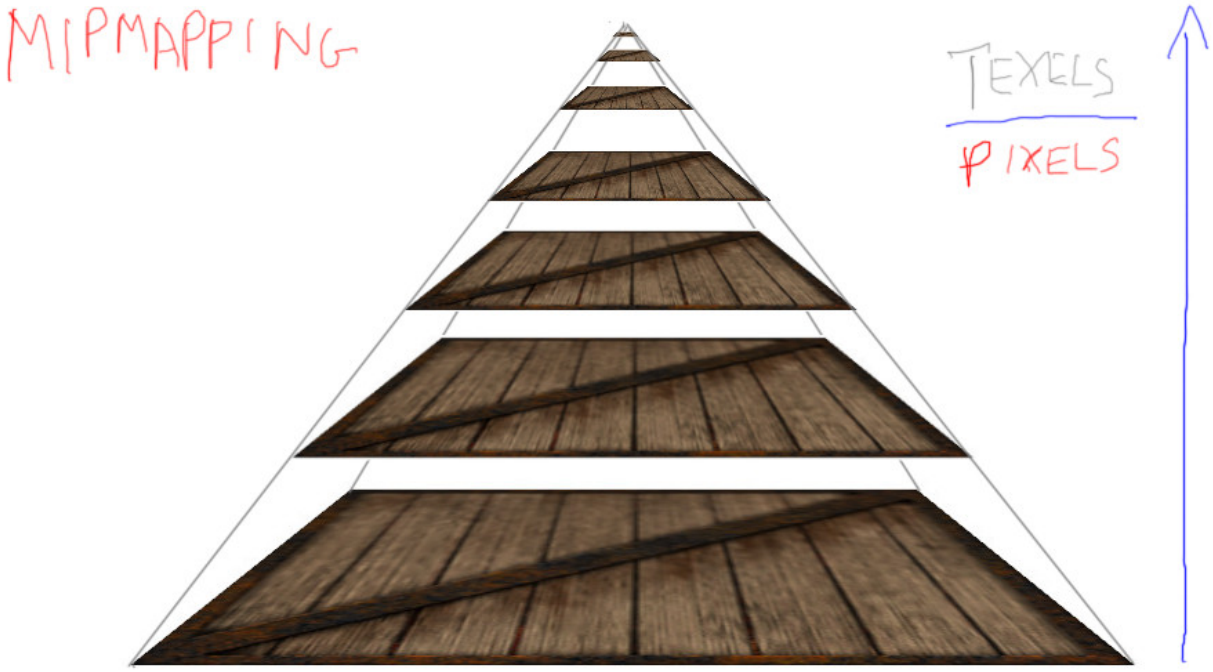
[**Note addition of ratio equations!**]

[ADD WORD "***MIPMAPPING***" to upper right]

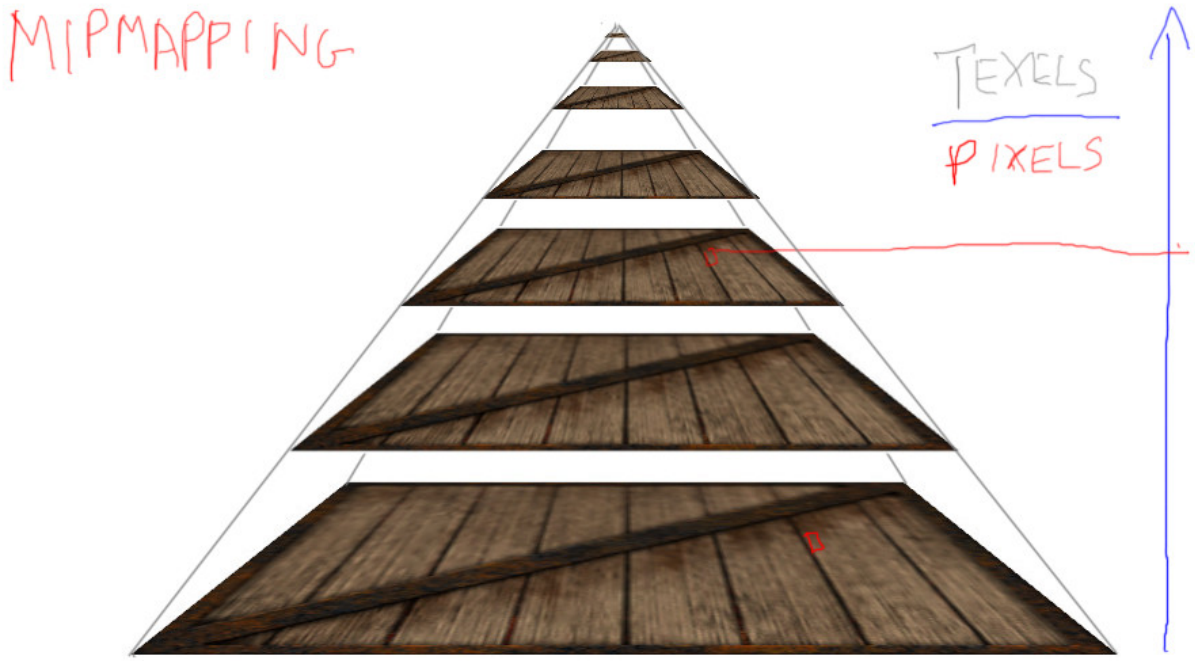


With magnification we found that if the texel covered too many pixels, we could just make our original checker texture larger and larger, so that about 1 texel covered one pixel. With minification, we'd like each pixel to cover no more than one texel.

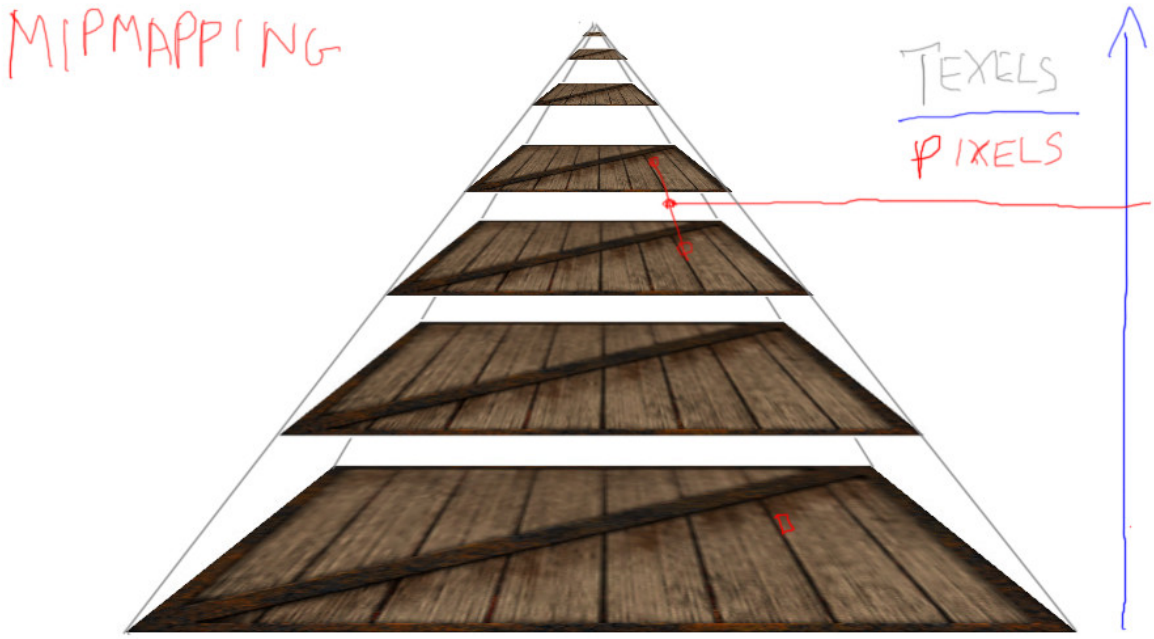
The GPU has built into it special functionality that implements an algorithm called ***mipmapping***. With mipmapping the GPU computes approximately what the texel to pixel ratio is. If this ratio is less than 1, magnification happens and the magFilter takes effect. If this ratio is greater than one, minification is happening. This is where mipmapping comes in.



What the GPU does is determines the ratio of texels to pixels. If this ratio gets much larger than 1 to 1, a lower-resolution version of the texture is used instead. As this ratio gets higher and higher, a lower and lower resolution of the texture gets used. These textures can be thought of as forming a pyramid.

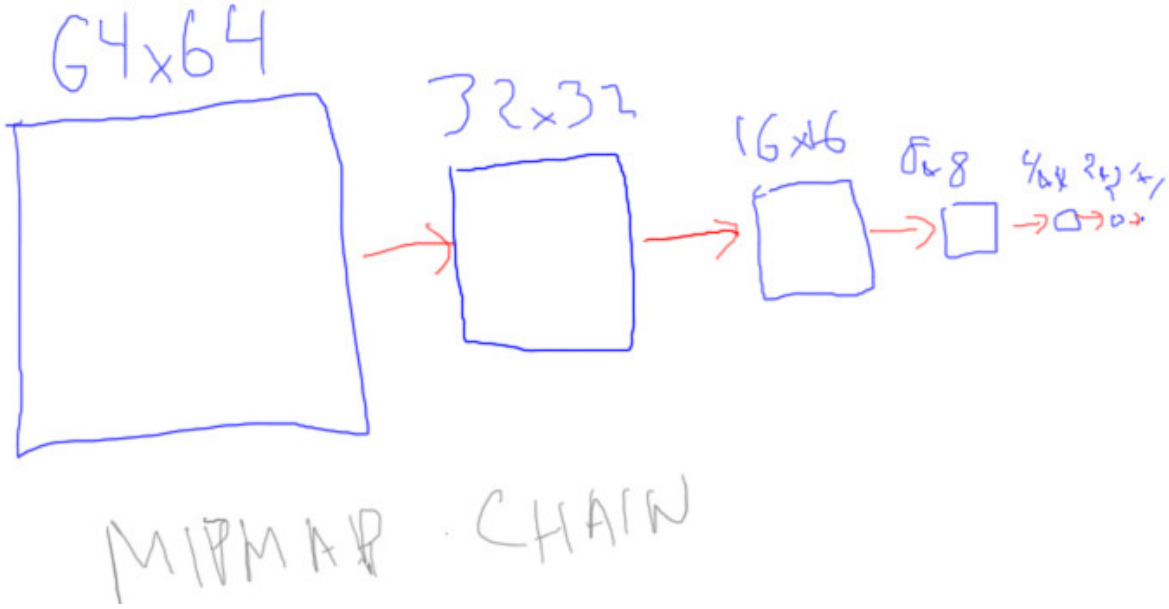


For example, say a pixel needs the texture color. Instead of taking whatever texel happens to cover the center of the pixel down at the bottom level, the GPU determines the texel to pixel ratio for this pixel, to see how high up the pyramid to go. The texture at this level of the pyramid is then sampled. Since the ratio of texels to pixels is about 1 to 1 for this lower resolution texture, the rendering looks much better, without the noise we saw before.

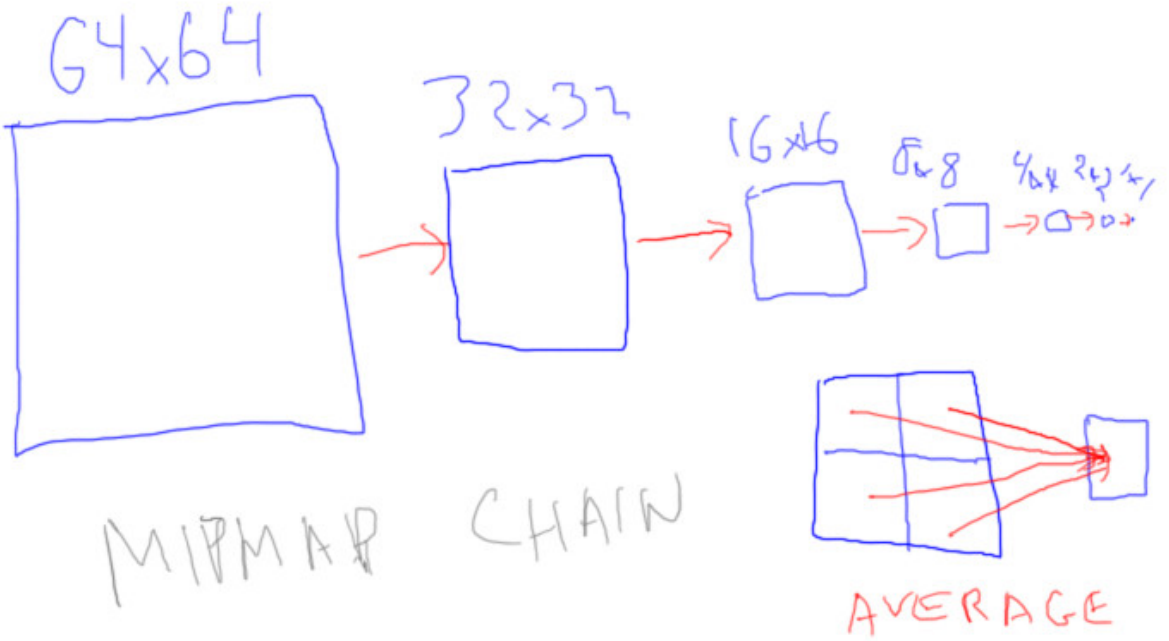


In fact, for even better quality rendering, we can set the GPU to linearly interpolate between levels. This is called "***trilinear filtering***". Say our ratio is in between two pyramid levels. We then sample the texture above and below where our sample is in the pyramid and interpolate between these two samples.

[mipmap chain]



My example pyramid was not drawn to scale. Within the GPU the pyramid is created and stored in powers of two. That is, if you start with a texture that is 64x64, the pyramid is formed with this texture, a 32x32 texture, a 16x16 texture, 8x8, 4x4, 2x2, and 1x1 at the end. This set of textures is called the mipmap pyramid or the mipmap chain.



Each smaller texture is derived from the larger one. One technique is to simply average the four texels of the larger texture to make the smaller, which can be done in advance or by the GPU

itself. There are some subtleties, such as gamma correction, that can change how these levels are formed. That's for a later lesson.

[say it louder!]

This is an important point, so pay attention: you want to make your textures to be powers of 2 in size, both rows and columns. You will not be able to use mipmapping if you don't. GPUs generally expect powers of two, so don't disappoint them unless you know what you're doing and have a very good reason for doing so.

Give the new version of the demo a try. You can now pick mipmapping as a filtering option for minification.

[Additional Course Material:
You can see the effects of different filtering and also which mipmap level is used (by its color) in [this
demo](http://mrdoob.github.com/three.js/examples/webgl_materials_texture_manualmipmap.html). Also try [this
demo](http://mrdoob.github.com/three.js/examples/webgl_materials_texture_filters.html) to see the effect of various sampling and filtering settings.
]

Demo: Mipmapping Demo

[unit8-txr_mipmap.js]

[Additional Course Materials:
Change the minification to "mipmap" to see the improvement.
]

Question: Mipmap Size Increase

[***mip: "multum in parvo" - much in a small space***]

The word "mip" in the name mipmap is an abbreviation for the Latin phrase "multum in parvo", meaning "much in a small space". This is an excellent description, as all the extra textures generated in the mipmap pyramid don't take up much space, but prove themselves quite useful.

Given a 32x32x1 channel texture, memory taken up is 1024 bytes. How much memory is used in total by a mipmap version of this texture?

_____ bytes

Answer

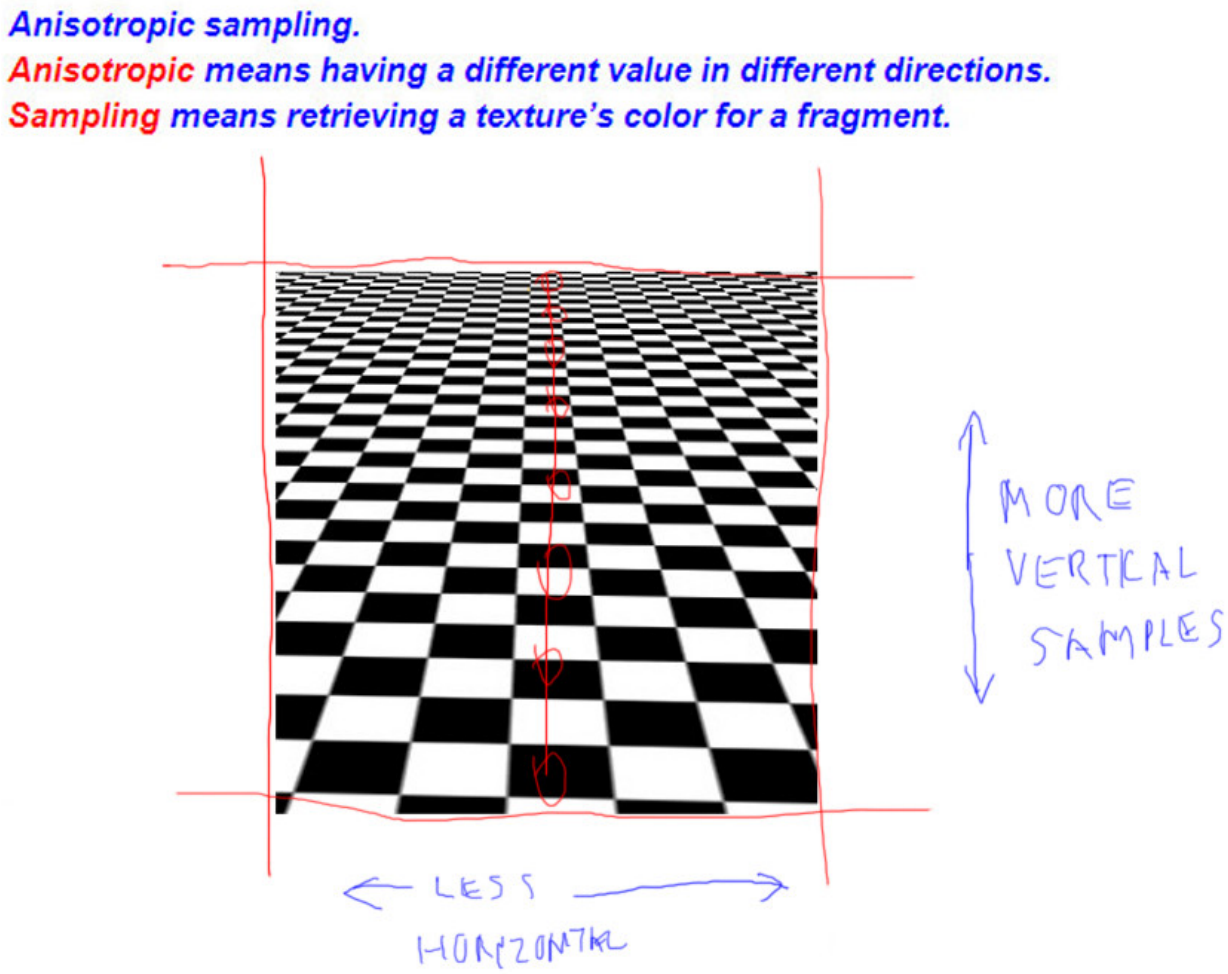
32x32 + 16x16 + 8x8 + 4x4 + 2x2 + 1x1 = 1365 bytes. This is about 33% additional space needed for the mipmap. This approximately 33% figure is true for any mipmap, and it is never more than this percentage.

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

Lesson: Anisotropy

[***Anisotropic sampling.***
Anisotropic means having a different value in different directions.
Sampling means retrieving a texture's color for a fragment.
]

[Draw the sample axis with a thick line, make sure the samples are evenly spaced and noticeable]



Mipmapping looks much better, but goes to gray at the horizon. Modern GPUs can do even better than this, by using anisotropic sampling. Anisotropic means having a property that has a different value when measured in different directions. Sampling means retrieving some piece of information meant to represent the whole. For example, a fragment shader will take a sample from a texture's mipmap pyramid to represent the texture's color at that pixel.

Where the word anisotropic comes into play is when we get to these pixels on the horizon. Think about looking along one column of the checkerboard. At the horizon a pixel may cover a large number of texels vertically, but fewer horizontally. That is, the number of texels differs in different directions. What anisotropic sampling does is take extra samples along the axis where texels are more frequent and blends these together. These extra samples generally give a better result, not blurring out as quickly as the mipmap does.

Lesson: Sampling and Filtering

[LEAVE ROOM ABOVE THIS CODE FOR WORDS “SAMPLING” AND “FILTERING”]

```
var texture = new THREE.Texture();
texture.magFilter = THREE.NearestFilter;
texture.magFilter = THREE.LinearFilter;      // the norm

texture.minFilter = THREE.NearestFilter;
texture.minFilter = THREE.LinearFilter;
texture.minFilter = THREE.LinearMipMapLinearFilter;       // the norm

texture.anisotropy = 1;
texture.anisotropy = renderer.getMaxAnisotropy();
```

```
var texture = new THREE.Texture();
texture.magFilter = THREE.NearestFilter;
texture.magFilter = THREE.LinearFilter; // the norm

texture.minFilter = THREE.NearestFilter;
texture.minFilter = THREE.LinearFilter;
texture.minFilter = THREE.LinearMipMapLinearFilter; // the norm

texture.anisotropy = 1;
texture.anisotropy = renderer.getMaxAnisotropy();
```

Here’s a summary of the various three.js settings for magnification, minification, and anisotropy.

The minimal settings are listed first, the higher quality further down.For mipmapping I list the common choice, where you linearly interpolate between mipmap levels and also bilinearly interpolate among texture samples on a texture level. There are other options, for back when mipmapping was an expensive process.

Quality does cost more GPU processing time, so you have to decide what settings you can justify. The norm nowadays is to have linear magnification and the best quality mipmap filtering on.

I should note that anisotropic sampling does not exist on older GPUs, though is an extension in WebGL. Each vendor has its own secret sauce as to how they implement this algorithm, so

rendering quality can vary. The maximum anisotropic sampling rate can also vary from GPU to GPU. Calling the getMaxAnisotropy function gives the maximum rate possible on the GPU running the program. This is usually a power of two, usually no higher than 16. You can set the value higher if you want; three.js will use the highest level available on the card. More samples cost more processing power, so again you need to decide what rate is reasonable.

[***SAMPLING and FILTERING***]

There are yet still better sampling methods possible, anisotropic filtering is not the ultimate answer. I went into perhaps excessive detail on this aspect of the GPU, but for a good reason. This whole process of sampling and filtering comes up again and again throughout all of computer graphics. If you take too many samples, you're wasting time and possibly causing other problems; too few and you can get artifacts. Where the samples are taken, and how the samples are combined, or filtered, also can make a huge difference.

[perk up, important conclusion]

Many processes can be thought of in these terms: did I tessellate the surface enough to capture its essence? Should my computation happen in the vertex shader or the fragment shader? Should my shadow buffer have a higher resolution, or should I use a different algorithm? When I think of computer graphics and rendering as a whole, I think of it primarily in terms of sampling and filtering.

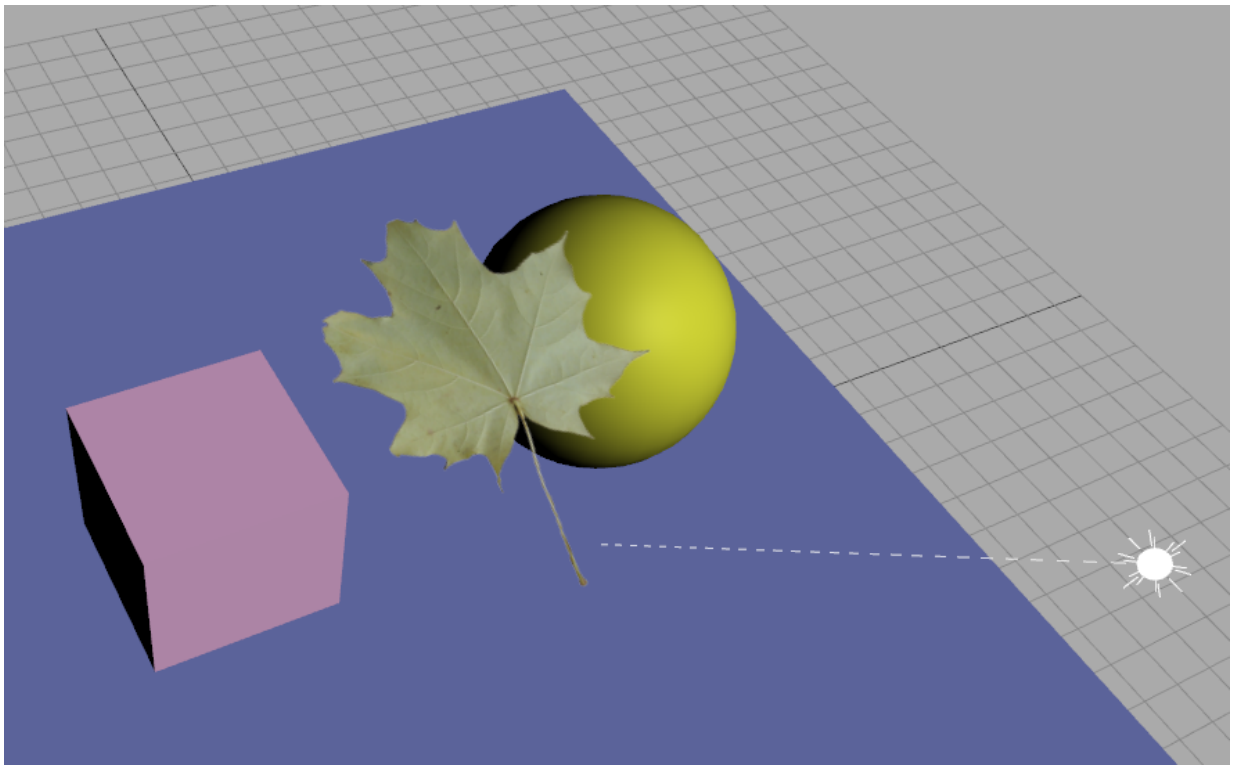
Give the demo a try. It will show you the valid levels for your GPU - if you only see the value "1", it means your GPU does not support anisotropic filtering.

Demo: Anisotropy Demo

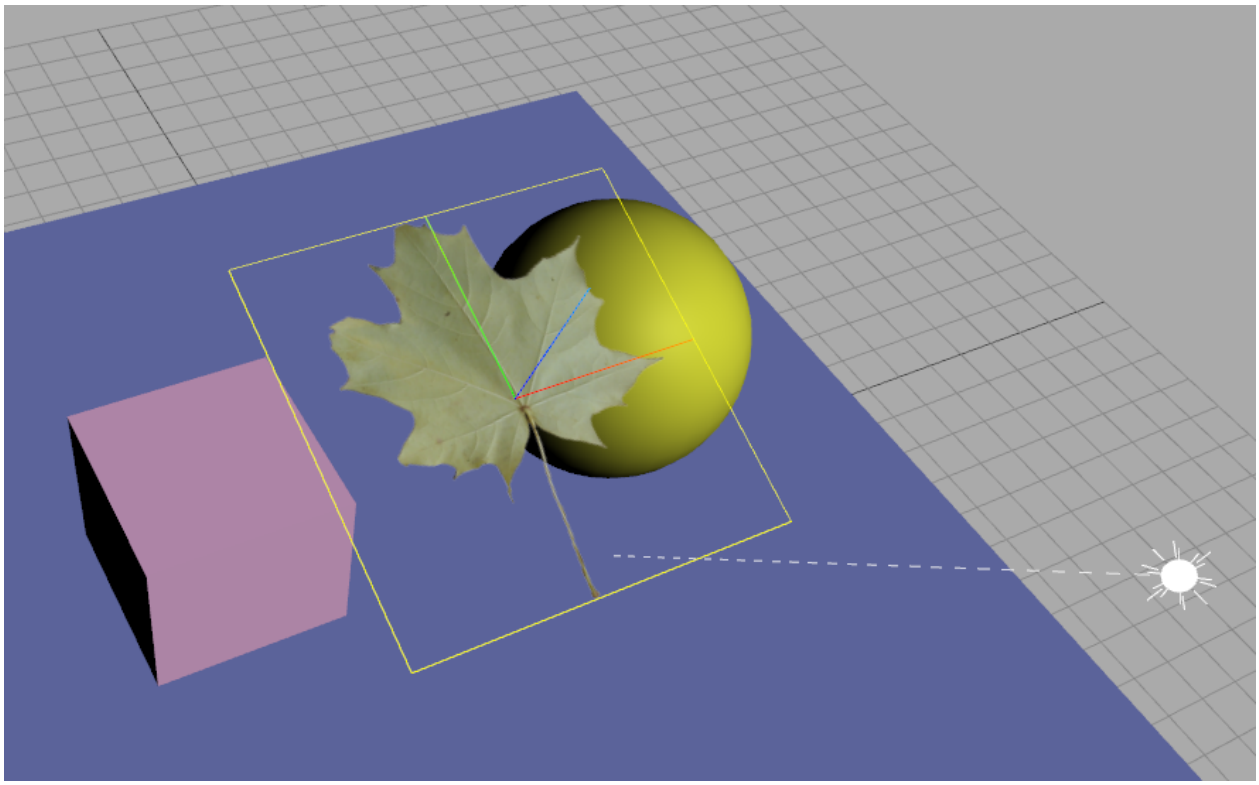
[Student runs unit8-txr_anisotropy.js]

[Additional Course Materials:
You can and should try out [this additional anisotropy demo](http://mrdoob.github.com/three.js/examples/webgl_materials_texture_anisotropy.html).
Depending on your machine's GPU, the anisotropy demo may not work for you.
]

Lesson: Transparency Mapping



Just as we can change the color of an object using a texture, we can also change its transparency. Here's a simple scene with a leaf added to it by putting a leaf texture on a square.

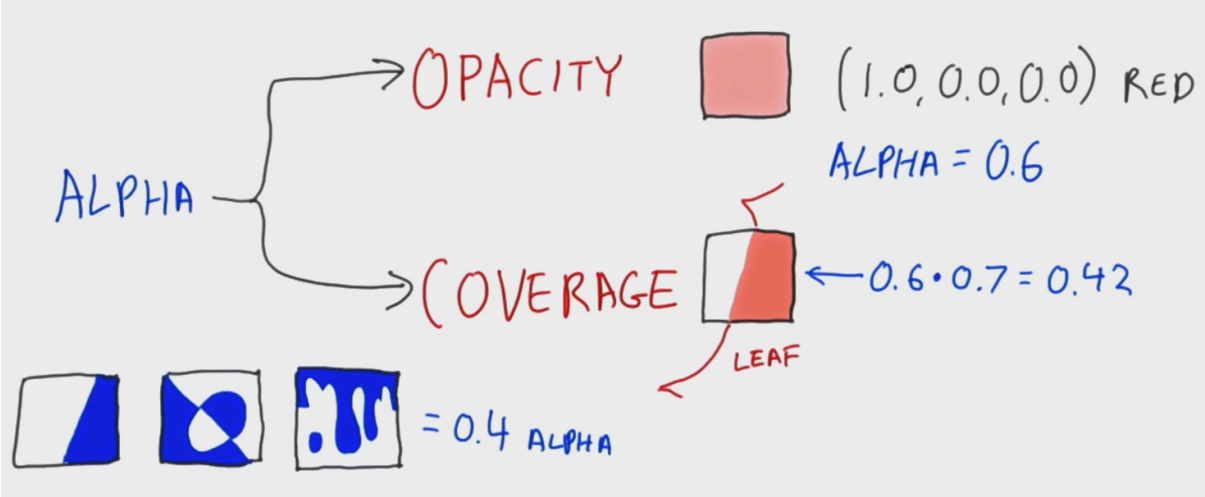


Here's the same scene with the square selected, so you can see the underlying geometry. The two key things I needed to do was to mark the square as transparent, and to make a texture with an alpha channel.

Recall that a transparent object needs to be given a transparent material. This ensures that it is rendered after all opaque objects, and blending is turned on so that any transparent pixels will blend properly with the objects behind. In this case most of the pixels are either opaque or fully transparent, but the idea is the same. By turning on blending, the leaf's RGB values are properly blended using its alpha values.

Actually making the texture with an alpha channel I leave to you. Packages such as GIMP, paint-dot-net, or Photoshop can create an alpha channel for an image. You'll typically want to use the PNG format to store the results, since PNG supports alpha. Here's an example of an object with its alpha texture. White areas are where the plant is, black is transparent, and gray alphas mean the pixels are partially covered by some leaf.

Lesson: The Meaning of Alpha



When using transparency for materials, we set the color's alpha value to specify an object's *opacity*. You might have a red material and the alpha is used to make it semitransparent. When used for texturing, we often think of the alpha as the amount of *coverage*.

For example, to create a cut-out leaf, you could specify for each texel whether the leaf is visible or not, an alpha of 1.0 or 0.0. However, you can get a better result by specifying an alpha between 1.0 and 0.0 for texels on the edge of the leaf. Each edge texel's alpha represents how much the leaf covers that particular texel.

What is interesting here is that a single alpha value is used for both opacity and coverage. [draw to make filled area semitransparent] It can even mean both at the same time: you could have a cutout figure that is also semitransparent. Whatever the interpretation, we use the same *over* blending equation when putting one object atop another.

[draw two alpha coverages and different options of the second on top the first]
Using alpha for coverage is a simplification. For example, when we have just an alpha value, we don't know the true area covered by the object. Combining a few coverages together is something of a best guess when we use blending. This is usually a fine assumption under normal circumstances, where only a few pixels have multiple overlapping edges.

Lesson: Premultiplied Alpha

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

$C_p \leftarrow \alpha_s C_s$ *premultiplied*

*C = Cp + (1 - αs) * Cd* *one less multiply*

One idea you'll run across with transparent textures is that of premultiplied alpha. PNG files never use premultiplied alpha, other formats may. Texture data can be stored however you want in memory, either unmultiplied or premultiplied

Here's the blending equation normally used for transparency. The source, that is, the transparent object is blended with the destination, the object behind it, using the source's alpha value.

Premultiplied alpha means just that. The term alpha-source times color-source is going to be the same every time this texture data is accessed. We can premultiply the original color by the alpha and store this new color in memory. This equation saves a multiplication every single time this texel is accessed. Say we access this texture 60 times a second and a million texels are accessed. That saves 60 million multiplies a second, which can start to add up.

[Additional Course Materials:
There are a number of benefits to premultiplied alpha, some of which are discussed in [this blog post by Tom Forsyth](http://home.comcast.net/~tom_forsyth/blog.wiki.html#%5B%5BPremultiplied%20alpha%5D%5D).

A number of free packages let you work with texture alpha, including [GIMP](http://www.gimp.org/) and [paint.net](http://www.getpaint.net/). Careful where you click on the paint.net page, it's a minefield.
]

Question: Valid Unmultiplying

[recorded 3/29, part 1]

[show RGBA premultiplied get multiplied by 0.5 - note to apply these Alphas only to RGB values, not A itself.]

Here's an RGBA texel value that is not pre-multiplied, let's call it unmultiplied: ***(0.3, 0.7, 1.0, 0.5).*** If we premultiply the RGB value by the alpha value, we get ***(0.15, 0.35, 0.5, 0.5)***. If we undo this multiplication, we get the unmultiplied RGBA values back. All unmultiplied RGBA values can be converted to premultiplied RGBA values and then back again to unmultiplied. Precision could be lost in converting back and forth like this, but otherwise it's always valid when starting with an unmultiplied value.

However, you can start with a premultiplied RGBA value that cannot be properly converted and stored as an unmultiplied RGBA, and so could not have come from unmultiplied values.

Which premultiplied texels below come from valid unmultiplied numbers?

Valid means that all four values of the unmultiplied RGBA are between 0 and 1, inclusive. The order of values below is ***RGBA***.

[ ] (0.5, 0.0, 0.0, 0.2)
[ ] (0.0, 0.5, 0.5, 0.5)
[ ] (0.2, 0.2, 0.2, 0.0)
[ ] (0.0, 0.0, 0.0, 0.0)

My advice is to think about what the premultiplied RGBA value is representing, if you have any concerns about whether it can be converted back to an unmultiplied value.

Answer

[recorded 3/29, part 1]

[ ] (0.5, 0.0, 0.0, 0.2)
[X] (0.0, 0.5, 0.5, 0.5)
[ ] (0.2, 0.2, 0.2, 0.0)
[X] (0.0, 0.0, 0.0, 0.0)

This first premultiplied texel would turn into (2.5, 0.0, 0.0, 0.2). The 2.5 value is clearly out of range of a value that can be stored in a texel, in the range 0.0 to 1.0, so this texel cannot be properly represented when unmultiplied.

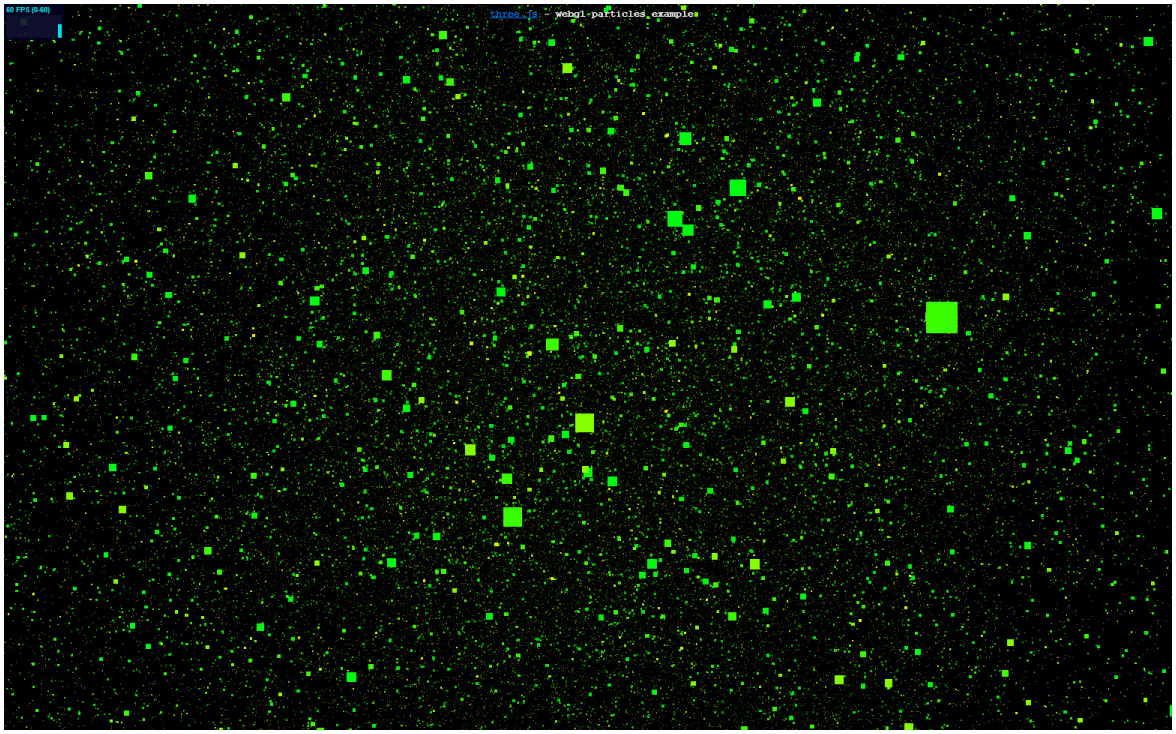
The second texel would convert to (0.0, 1.0, 1.0, 0.5). This is a valid unmultiplied RGBA value.

To convert the third texel the alpha value would have to be used to divide the RGB values. Dividing a non-zero number by 0 is illegal. So this premultiplied RGBA value cannot be converted and stored in unmultiplied form. It is not a value that could have come from any unmultiplied RGBA value, because the alpha (which doesn't change in conversion) would multiply any RGB value and so set it to 0.

The last RGBA looks like it might be invalid to convert, since you would need to divide by zero to convert from premultiplied to unmultiplied form. However, this RGBA is entirely valid and is the same in premultiplied form. It represents an entirely transparent pixel. If you started with an unmultiplied RGB of any value and an alpha of 0, the premultiplied form would be the same. Note that the premultiplied form would lose the RGB that the unmultiplied form had in this case.

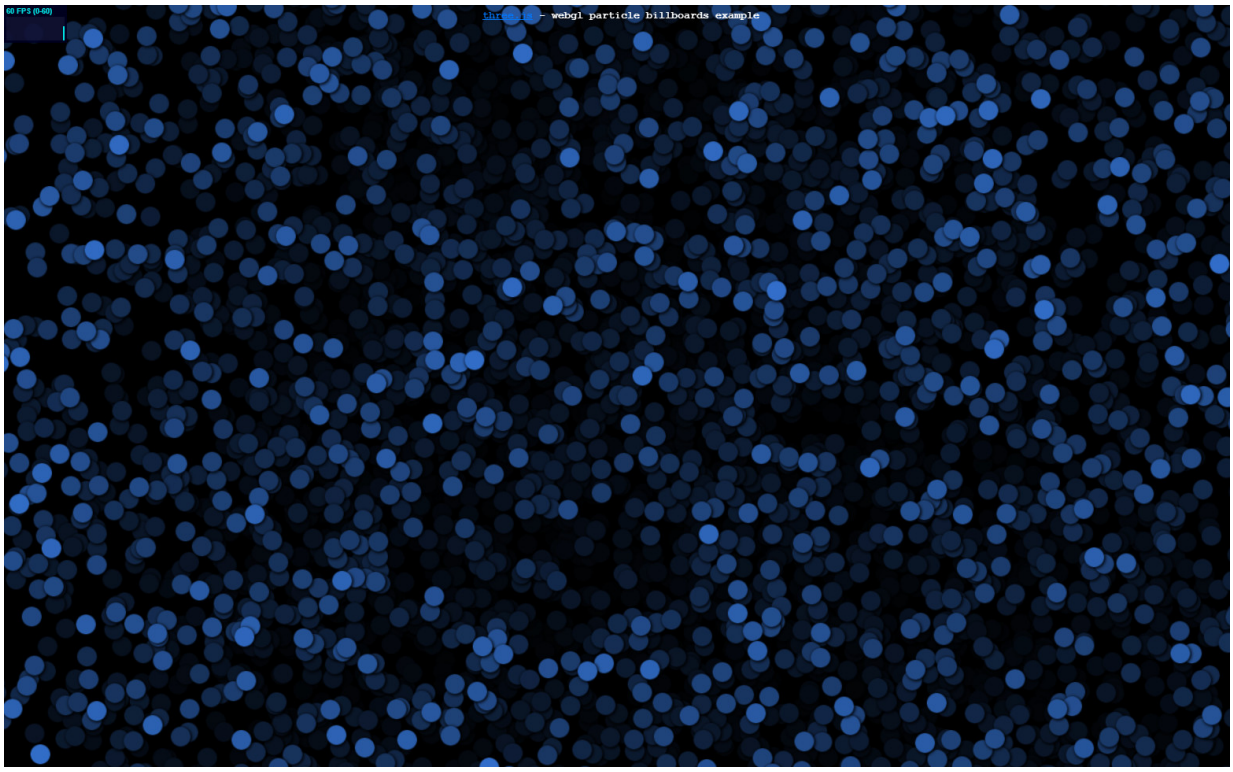
Lesson: Particles and Billboards

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



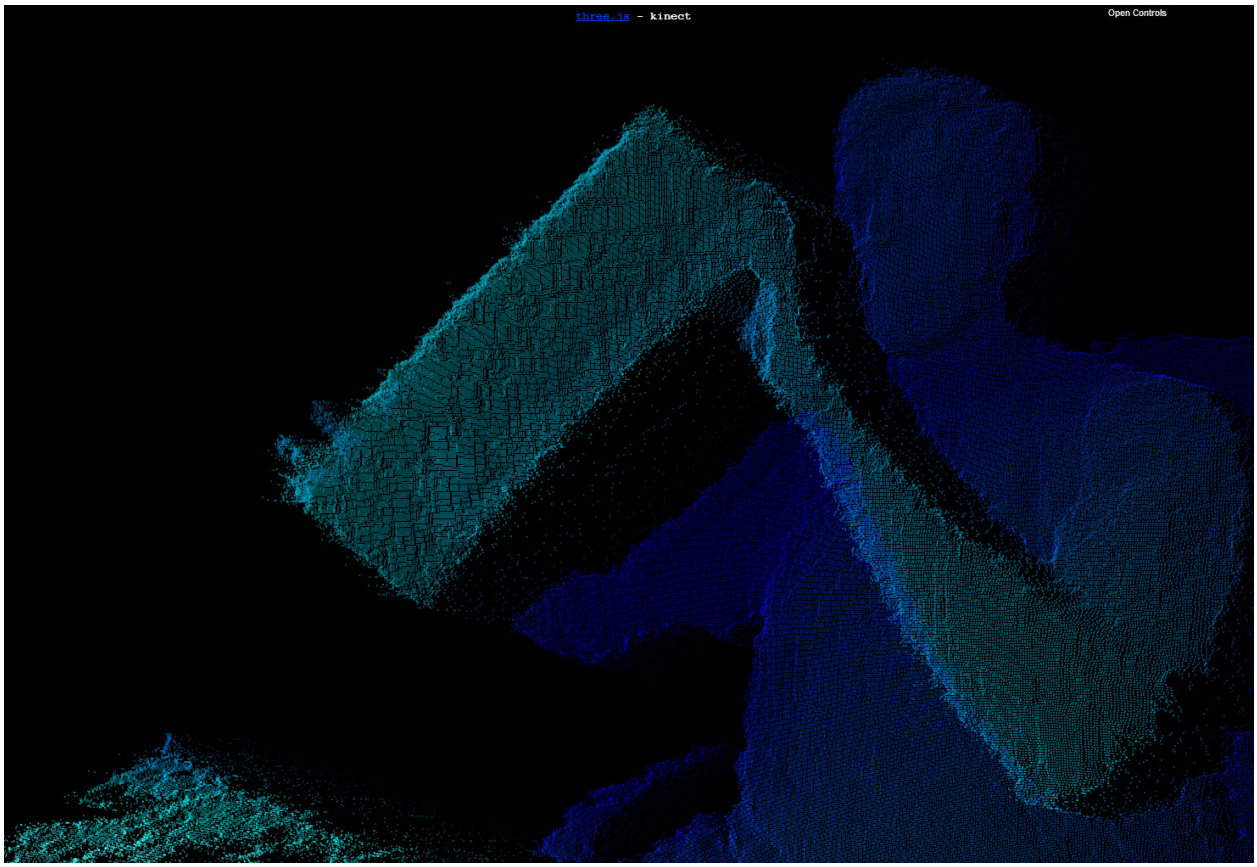
One cool thing you can do with a square that has a transparency map is to create a bunch of these and have them move around. When a large number of objects like this move around, we call them “particles”. This demo has 100,000 particles. One particularly useful way to display particles is as if they’re 2D objects flat on the screen. The objects are actually in a 3D world, however. As the camera moves, these objects are reoriented so that they always face forward. In this example, we see squares moving about, with each square always facing the viewer.

[http://mrdoob.github.com/three.js/examples/webgl_particles_billboards.html - circle cutouts, good next step]



You can then apply a texture with an alpha channel to get an interesting look. Here are a bunch of dots moving around.

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



An advantage of particles is that they can directly represent 3D scanned data. For example, here some movement data captured using a Kinect is displayed with particles. Some researchers and developers are actively looking at how to use particles as basic building blocks instead of triangles.

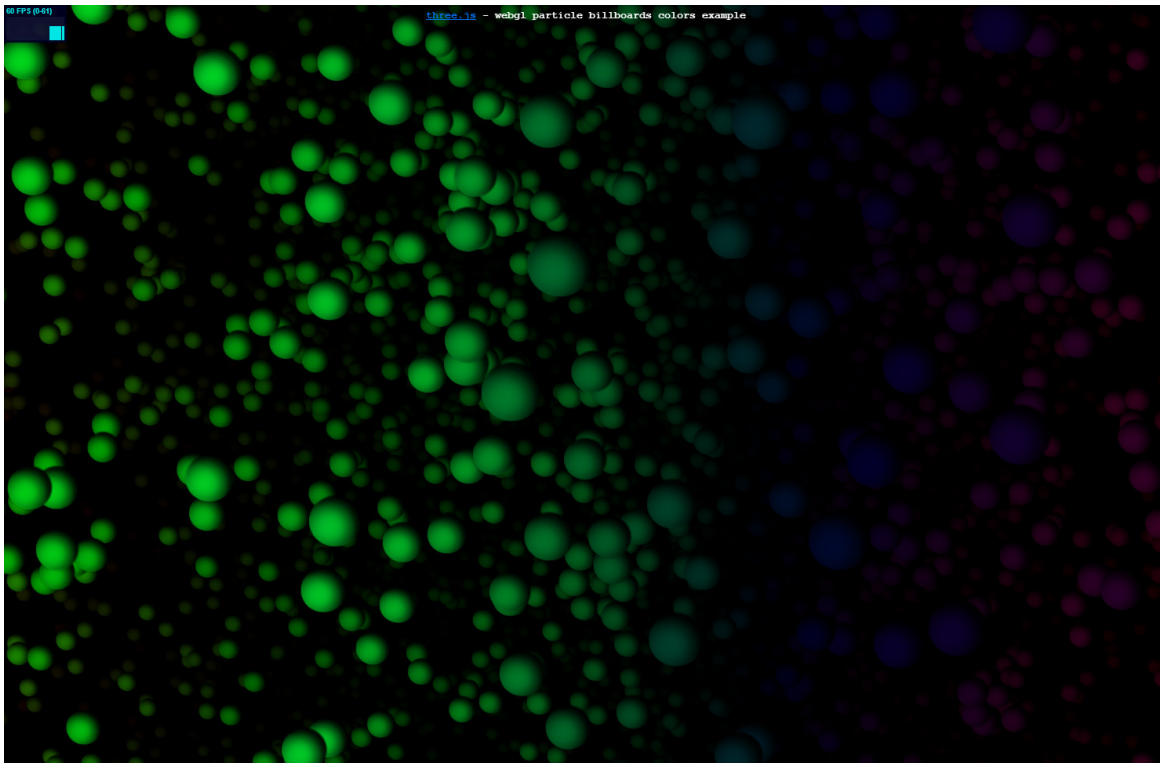
[asked http://potree.org/demo/skatepark_v1.0/skatepark_v1.0.html mschuetz gave permission to show skate park potree.org demo here]

Here's an example from Markus Schütz on his potree.org site that shows a realistic view made entirely of particles. This data was captured by a laser scan system by Faro Technologies. One advantage of particle systems is that data can be captured by a scanner and then directly viewed. No processing is needed to reconstruct any triangles or meshes.

This program is running in the browser, with more particles flowing in and filling in the gaps the longer you view it. That's one advantage of particles: you can stream them in as you wish, bit by bit. With triangles you tend to send in either some simplified mesh or the whole object, causing objects to pop into existence.

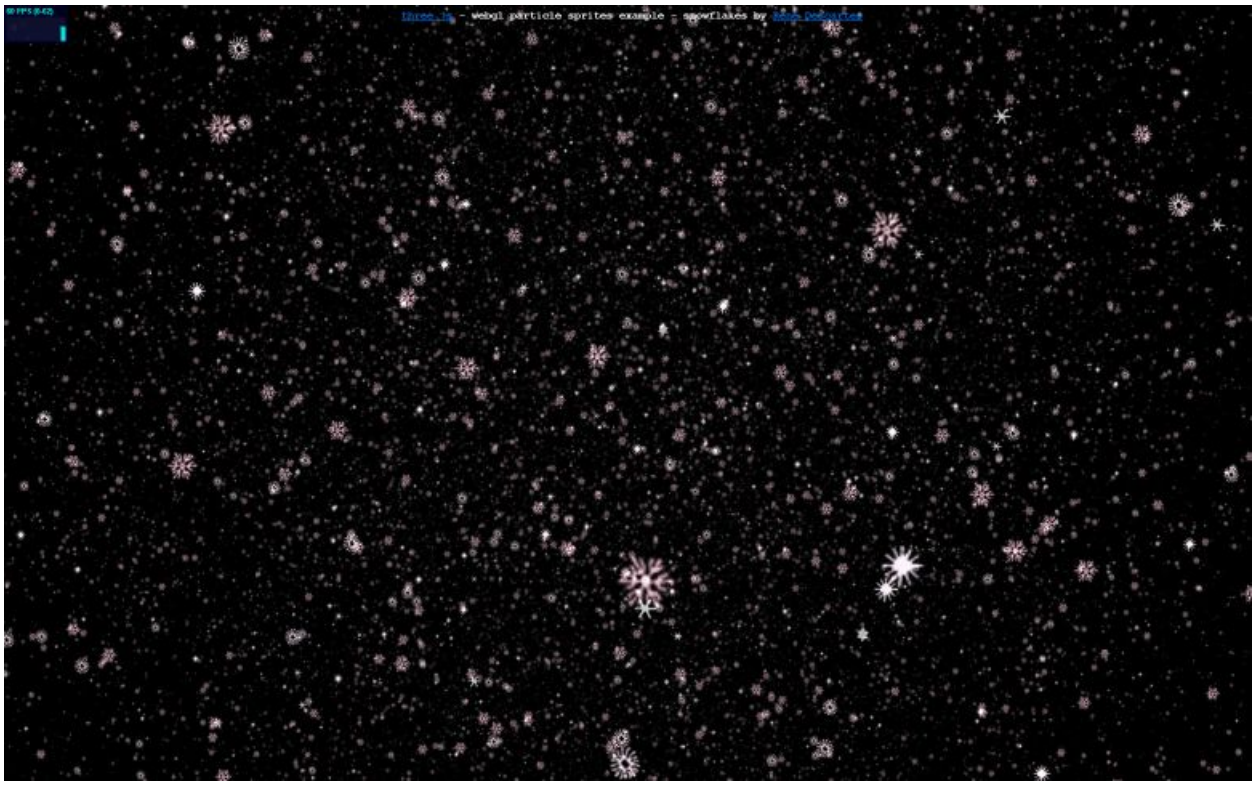
[leftover, video clip 2676-yacht-keystone.mp4 - don't use, skatepark is better]

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



The particles themselves can be made to look more realistic. By using a sphere texture instead of a circle, these particles are more convincing as 3D objects.

[http://mrdoob.github.com/three.js/examples/webgl_particles_sprites.html - snow, good next]



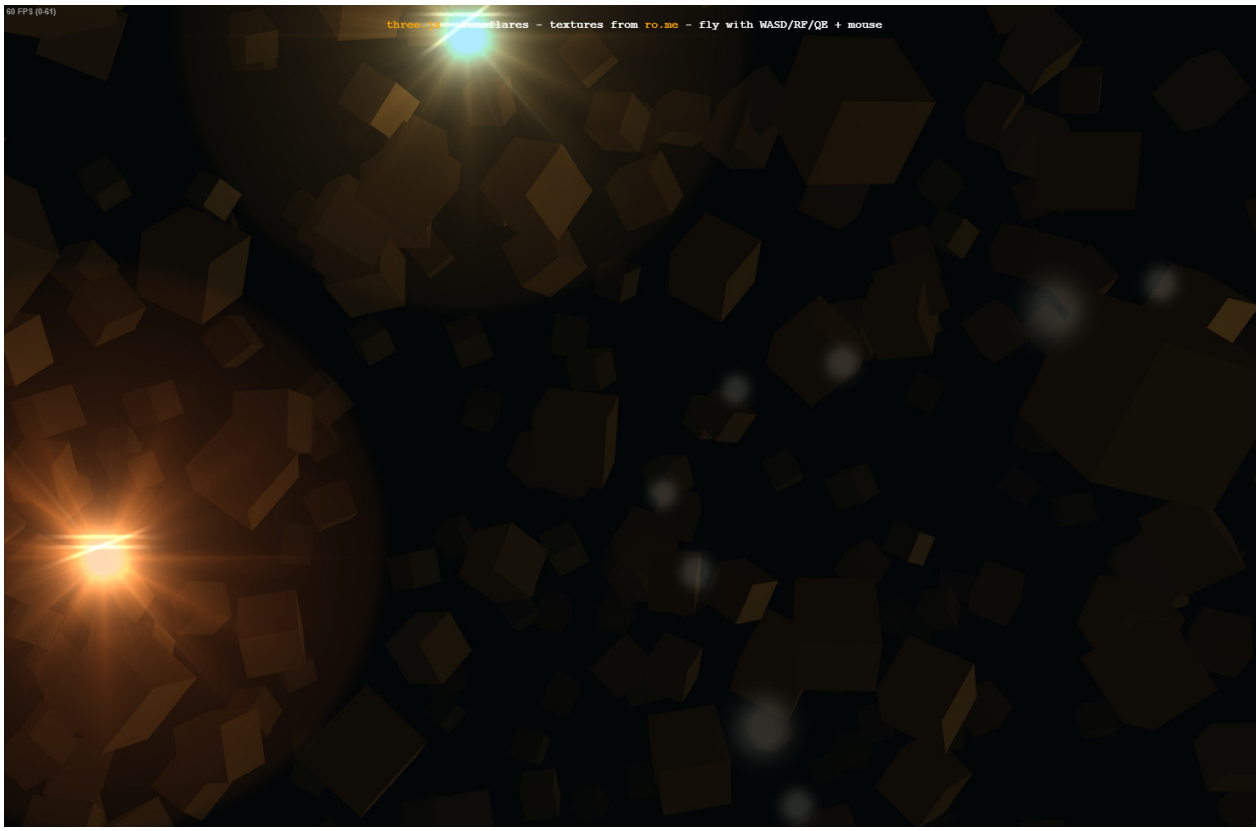
Weather is a common use for particle systems, for making rain drops or snowflakes.

[http://mrdoob.com/lab/javascript/webgl/clouds/]

Particles usually face the viewer. This idea of having objects always face forward is called billboarding. It allows us to make 2D textures seem more like 3D objects, as we've seen.

This demo shows a popular use for transparent textures on billboards: simulating clouds. This demo is by the main author of three.js, in fact. Larger billboards of cloud shapes are combined and overlapped to give the illusion of puffiness.

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



I have to conclude with one of the cliches of computer graphics, the lens flare. It's so easy to add this effect that it's been in many games, though there probably have been more games with crates. The sparkling patterns that simulate interaction of light with the camera's parts are all transparent textures whose positions are computed on the fly.

[Additional Course Materials:

Almost all of the demos shown are a part of the three.js distribution. Here is the list:

[Simple squares](http://mrdoob.github.com/three.js/examples/webgl_particles_random.html)
[Circle cutouts](http://mrdoob.github.com/three.js/examples/webgl_particles_billboards.html)
[Kinect](http://mrdoob.github.com/three.js/examples/webgl_kinect.html)
[Point cloud skatepark](http://potree.org/demo/skatepark_v1.0/skatepark_v1.0.html) from Markus Schütz on (his potree.org site)[http://potree.org/].
[Spheres](http://mrdoob.github.com/three.js/examples/webgl_particles_billboards_colors.html)
[Snowflakes](http://mrdoob.github.com/three.js/examples/webgl_particles_sprites.html)
[Clouds](http://mrdoob.com/lab/javascript/webgl/clouds/)
[Lensflare](http://mrdoob.github.com/three.js/examples/webgl_lensflares.html)

For more examples of using particles to represent real scenes, try [the potree.org demos](http://potree.org/wp/demo/) - the longer you wait, the more particles get loaded.

For commented code that shows how to create and use billboards, see [Lee Stemkoski's code](http://stemkoski.github.com/Three.js/Sprites.html).

Lens flare is described [here](http://en.wikipedia.org/wiki/Lens_flare), and some ways to simulate it on the GPU are [here](http://www.john-chapman.net/content.php?id=18).
]

Lesson: Making Particles

[recorded 3/29, part 1]

[Add the sprite on top, as shown]

```
var disk = THREE.ImageUtils.loadTexture( "textures/disc.png" );
var material = new THREE.ParticleBasicMaterial(
        { size: 35, sizeAttenuation: false, map: disk, transparent: true } );
material.color.setHSL( 0.9, 0.2, 0.6 );

var particles = new THREE.ParticleSystem( geometry, material );
particles.sortParticles = true;
scene.add( particles );
```

```
var disk = THREE.ImageUtils.loadTexture( "textures/disc.png" );
var material = new THREE.ParticleBasicMaterial(
    { size: 35, sizeAttenuation: false, map: disk, transparent: true } );
material.color.setHSL( 0.9, 0.2, 0.6 );

var particles = new THREE.ParticleSystem( geometry, material );
particles.sortParticles = true;
scene.add( particles );
```

Within three.js you can create a set of particles by using the ParticleSystem object. The code here shows one way to make some particles.

For the material, we load a disc texture that is transparent around the edges. Using this texture in ParticleBasicMaterial says to make a billboard type of object that points towards the camera. In

other words, we want to see this sprite texture always facing towards us. Setting sizeAttenuation to false means that we want each particle to have a constant size on the screen. Setting the size to 35 means we want the particles to be 35 pixels wide.

The next bit is to make the particle system object itself, by using the set of points and the particle material. By setting sortParticles to true, this means we want the particles in the system to be sorted with respect to the camera's view. This is important, as otherwise blending will not occur properly. When you have objects with transparency, you need to draw them in a back to front order so that they combine correctly.

```
var geometry = new THREE.Geometry();
for ( var i = 0; i < 8000; i ++ ) {
        var vertex = new THREE.Vector3();
        // accept the point only if it's in the sphere
        do {
                vertex.x = 2000 * Math.random() - 1000;
                vertex.y = 2000 * Math.random() - 1000;
                vertex.z = 2000 * Math.random() - 1000;
        } while ( vertex.length() > 1000 );
        geometry.vertices.push( vertex );
}
```


```
var geometry = new THREE.Geometry();
for ( var i = 0; i < 8000; i ++ ) {
    var vertex = new THREE.Vector3();
    // accept the point only if it's in the sphere
    do {
        vertex.x = 2000 * Math.random() - 1000;
        vertex.y = 2000 * Math.random() - 1000;
        vertex.z = 2000 * Math.random() - 1000;
    } while ( vertex.length() > 1000 );
    geometry.vertices.push( vertex );
}
```

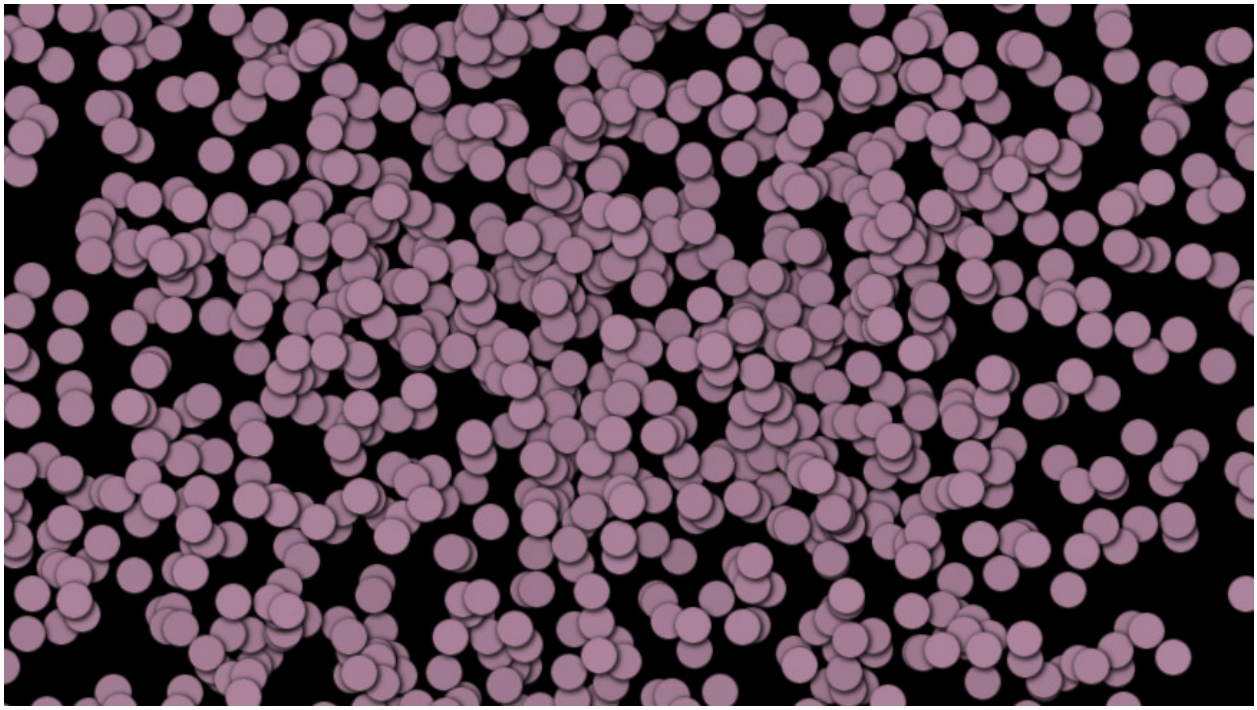

The geometry object you pass in is a list of vertices. In this example eight thousand particles are created and added to the scene. I randomly generate these vertices within a box going from -1000 to 1000 in X, Y, and Z, and check that each is inside a sphere of radius 1000. If not, I ask for another random point. There are more efficient ways to generate random points in a sphere,

but this is quick to remember and code.

Try the demo out. If you dolly out far enough you'll see the sphere formed.

Demo: Particle Sphere

[unit8-particle_demo.js]



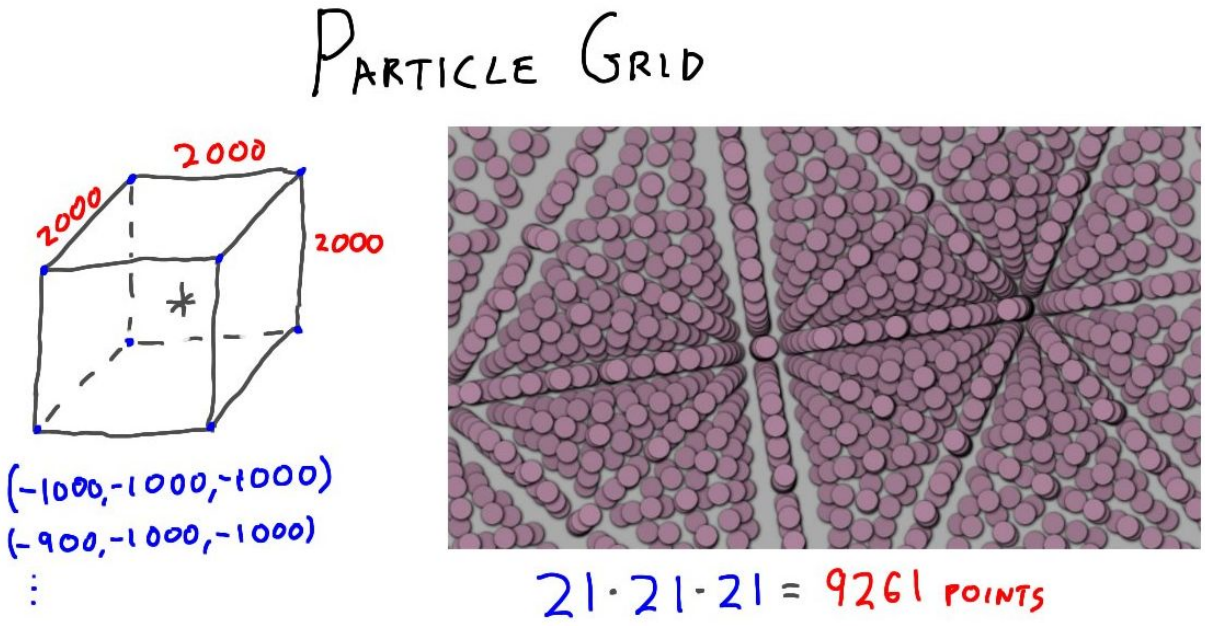
Exercise: Particle Grid

[recorded 3/29, part 1]
I want you to change the particle demo a little. Instead of 8000 particles placed at random, I want you to create 9261 particles in a highly-structured grid. Rewrite the point generation part of the code to place a particle at every point in a ***2000x2000x2000 unit cube*** centered around the origin, spaced 100 units apart. In other words, you'll want a point at ***XYZ -1000,-1000,-1000, XYZ -900,-1000,-1000*** and so on throughout the cube, up to point ***XYZ 1000,1000,1000. There are 21 points you'll place along any given line. This should add up to 9261 points, which is 21*21*21.***

When you're done, you should see this on the screen. I've turned the background color to gray

so that it's easier to see the particles themselves.

[really do just put up just the still starting image, and not the program itself, so student can truly compare.]



[exercise at unit8-particle_exercise.js]

Answer

[recorded 3/29, part 1]
[solution at unit8-particle_solution.js]

```
for ( var x = -1000; x <= 1000; x+= 100 ) {
        for ( var y = -1000; y <= 1000; y+= 100 ) {
                for ( var z = -1000; z <= 1000; z+= 100 ) {

                        var vertex = new THREE.Vector3(x,y,z);
                        geometry.vertices.push( vertex );
                }
        }
}
```

```
for ( var x = -1000; x <= 1000; x+= 100 ) {
    for ( var y = -1000; y <= 1000; y+= 100 ) {
        for ( var z = -1000; z <= 1000; z+= 100 ) {

            var vertex = new THREE.Vector3(x,y,z);
            geometry.vertices.push( vertex );
        }
    }
}
```

My solution is to use three loops, one for each axis, directly setting the x,y, and z values to each of the grid intervals. I then create a vertex with each point and add it to the list.

I encourage you to modify various parameters in this particle system and see the effects. If you set sizeAttenuation to false, the particles will have a size of 35 units in the world itself, instead of 35 pixels. If you turn off sortParticles, you'll see rendering errors as the particles are drawn in the wrong order from various view directions. Try removing the sprite texture entirely to see how that looks.

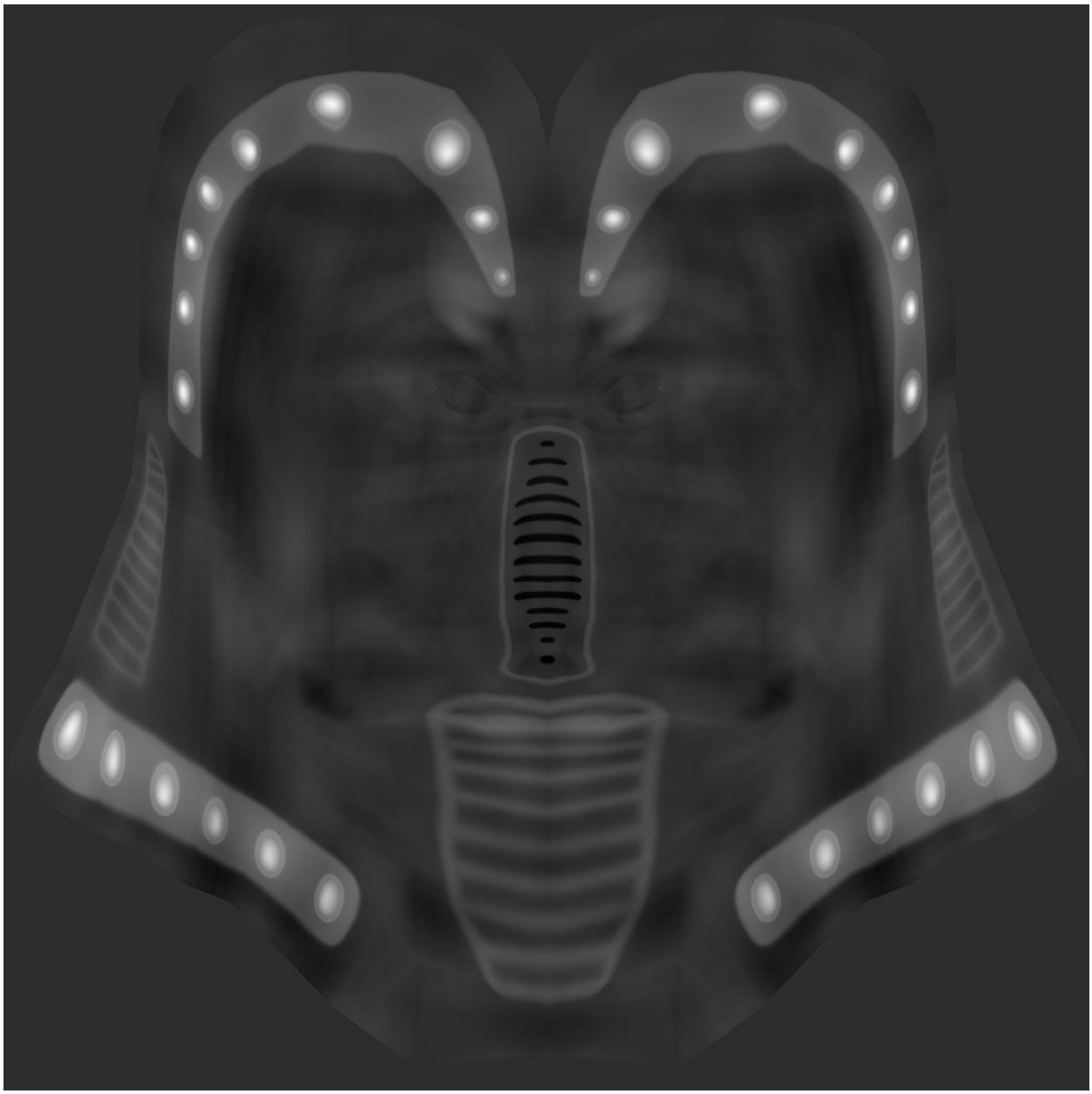
[Cut Lesson: Displacement Mapping]

[Run my modified unit6_displacement_map.html, or just show this still?]

There are plenty of different parameters that are used in creating a material. Most are straightforward to modify. You change the color with a color image texture. You change the shininess by making an image where the white areas are shiny, black areas are dull.

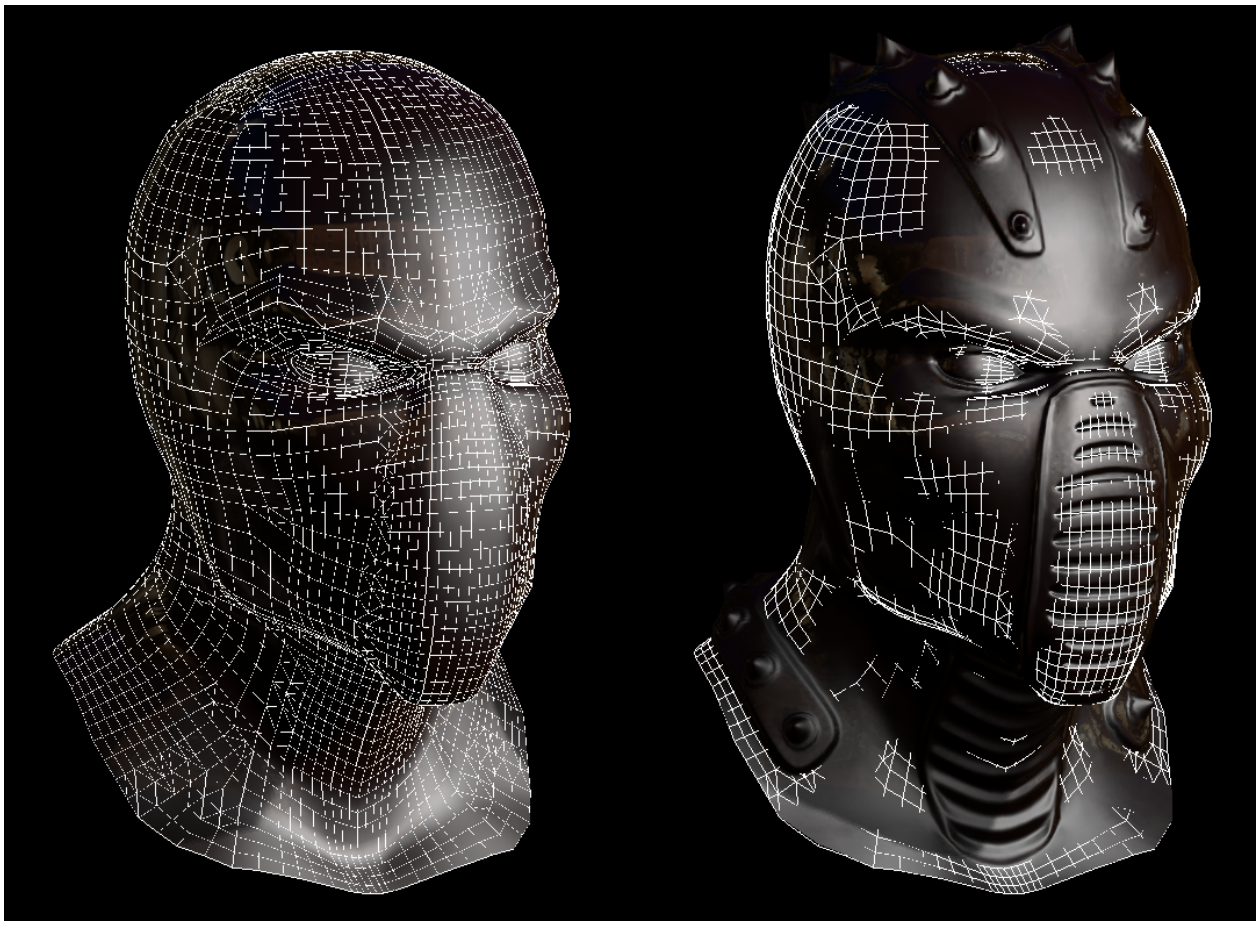
Another element we can change with a texture is the mesh itself. The idea is to have a rough model to start with. An artist then adds detail by raising and lowering the heights of the various points on the mesh. Instead of actually modifying the mesh directly, these modifications are stored in a texture.

Here’s an example in action. On the left is the mesh. In the middle this mesh is rendered without displacement mapping. On the right the mesh is modified by displacements.



Here is the displacement texture itself. White areas are made higher, darker areas made lower. For example, the white areas are where spikes were added to the head.

[show my unit6_displacement_map_reveal.html program in action]



Here we can see the displacements that have occurred due to this texture.

I should mention that newer GPUs have added to the pipeline three new stages, called the Hull Shader, Tessellator, and Domain Shader in DirectX 11, that can perform things such as on-the-fly tessellation. Such capabilities are also in OpenGL, but not currently in the WebGL specification, which is aimed to support older GPUs.

[Additional Course Materials:
You can see displacement shaders, particle systems, and much else in action on DirectX 11-capable GPUs linked from [this page](http://forum.unity3d.com/threads/166715-Museum-of-the-Microstar-A-DX11-contest-entry-by-RUST-LTD-and-co).

The tessellation pipeline in DirectX 11 is explained [here](http://msdn.microsoft.com/en-us/library/windows/hardware/ff569022(v=vs.85).aspx).]

[Move Question: Where Does Displacement Happen?]

A question for you: ***where does the displacement of the surface happen?*** That is, which part of the pipeline deforms the surface itself?

( ) The vertex shader
( ) During rasterization
( ) The fragment shader
( ) The Z-Buffer

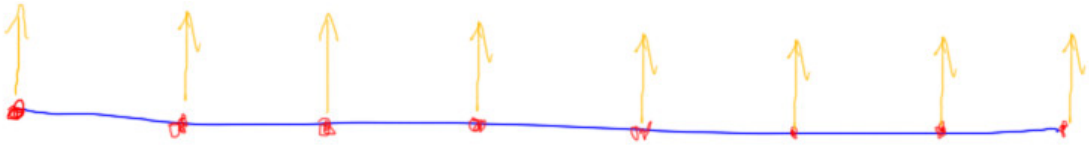
[Additional Course Materials:
The original version of this program can be found here http://mrdoob.github.com/three.js/examples/webgl_materials_normalmap.html . Note that this program may not run on older computers, as it requires Shader Model 3.0 level of GPU support.]

Answer

The answer is that this displacement happens in the vertex shader. This shader has control of what happens to each incoming vertex. The vertex shader reads the displacement map texture at the vertex's location and uses it to displace the vertex. The other parts of the pipeline take these displaced triangles and process fragments created by them. Each fragment has a pixel location and cannot change to another pixel location.

[Cut Lesson: Normal Mapping]

[use layers! copy and paste yellow normals]

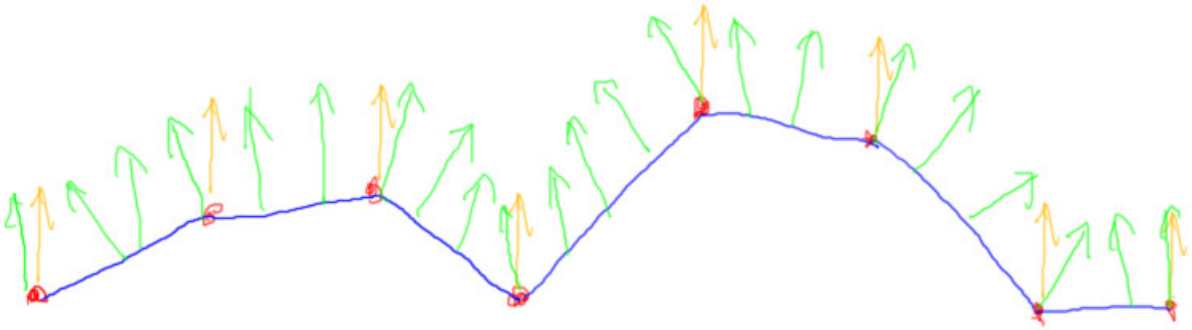


normal map:

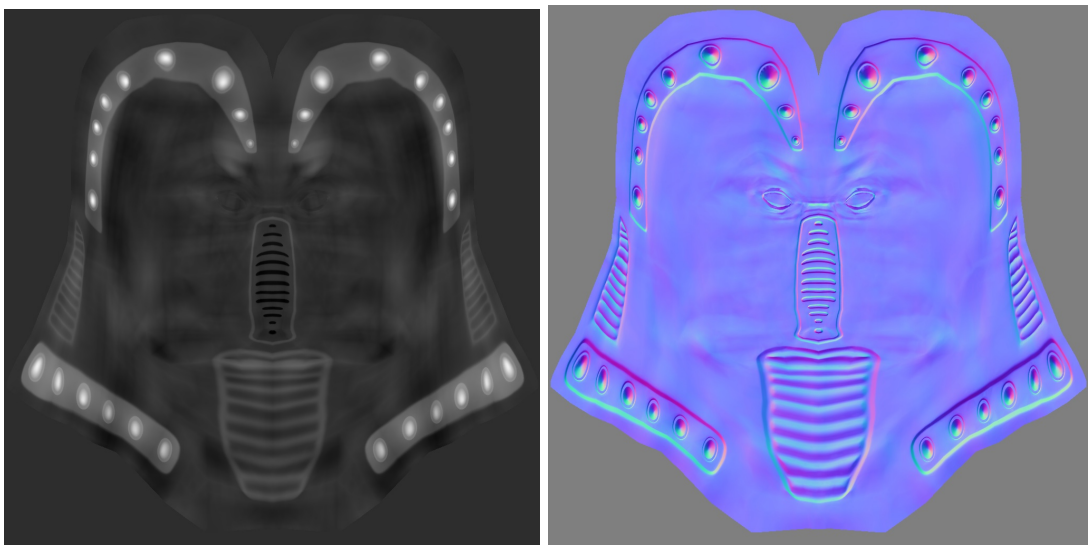
Displacement mapping uses a texture to change the location of the vertices themselves. However, that's only half of the story. Imagine you have a flat mesh, and each vertex has a normal. The normals all point straight up. You then displace the mesh, moving the vertices to different locations. This changes the geometry, but does nothing to modify the shading normals.

The way around this is to use what is called a ***normal map***. Instead of storing color or heights, this map stores what the normal should be at various points on the surface. Now we can change both the heights and the normals.

In theory we could change the normal at each vertex and pass these down to be interpolated during rasterization. However, we can achieve a much higher quality result than that. Unlike the displacement map, which has to work on the underlying vertices, the normal map can work on surface locations. The normal map does not change the position of any vertex, just what normal is to be used at that point on the surface.

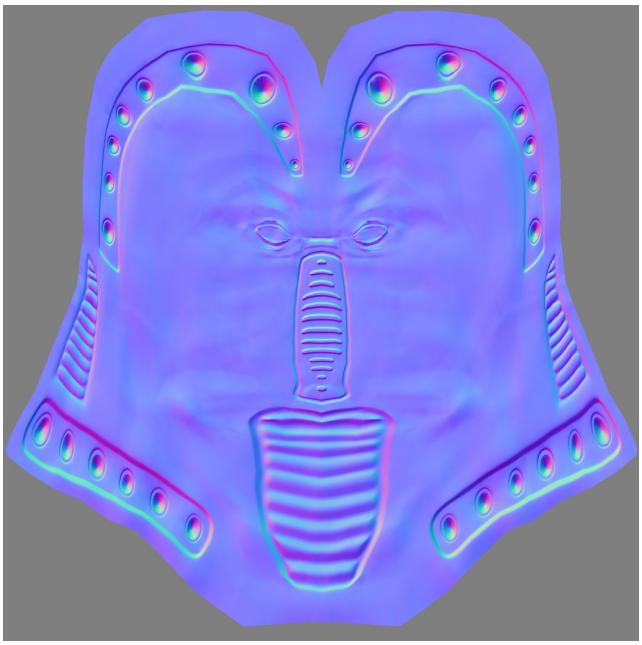


This means that, just like a color texture, we can look up a normal at each pixel. The fragment shader does this work, using the normal found in the normal map to modify the original shading normal.



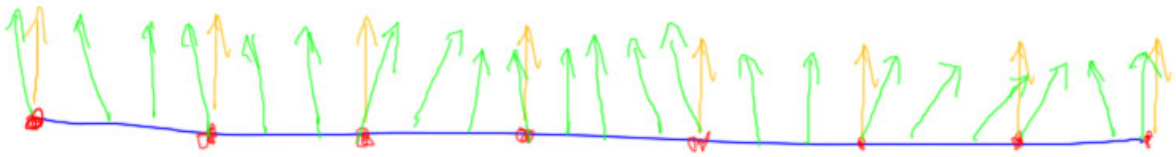
Here's the displacement map for the model, and here's what the normal map looks like. The normal map is in fact derived from the displacement map, and there are tools to create this type of texture. Each normal in the normal map is derived from the differences in the nearby heights in the displacement map.

You can always tell a normal map by its distinctive light blue. The normal map stores an XYZ direction vector at each texel. When we view this XYZ value as RGB colors, we get this type of look. Specifically, the red channel is X, with positive to the right, the green channel is Y, with positive up, and blue is the Z axis. This is called "***tangent space***". The range of values you can store in a color channel is 0 to 255, and normalized normals can range from -1 to 1. A color value of 0 maps to a coordinate of -1, a value of 255 maps to 1. Zero maps to about 128, so we convert by subtracting 128 and dividing by the same, more or less. A normal of (0,0,1) in XYZ is represented by 128,128,255, which gives this light blue. The more the color diverges from this blue, the more the shading normal points away from straight up.



Notice that this map is blue all over. However, the displacements and normals vary all over the place in world space. The way the displacement and normal maps are applied is with respect to the shading normals at the vertices. Each mesh shading normal forms the Z axis, with the UV axes forming X and Y. You can think of the normal map as how much to perturb the shading normal.

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



What if we got rid of the displacement map entirely and just used the normal map? In other words, we don’t vary the positions of the vertices but just vary the shader normals per pixel.

[add these pictures below]

It turns out that this looks pretty good. Here, from left to right, is our original mesh, the mesh with just a normal map, and with a normal map and displacement map. You can see that having only normal maps loses some details around the silhouette edge, but the rest of the model looks just about the same.

[add ***bump mapping*** underneath middle one]

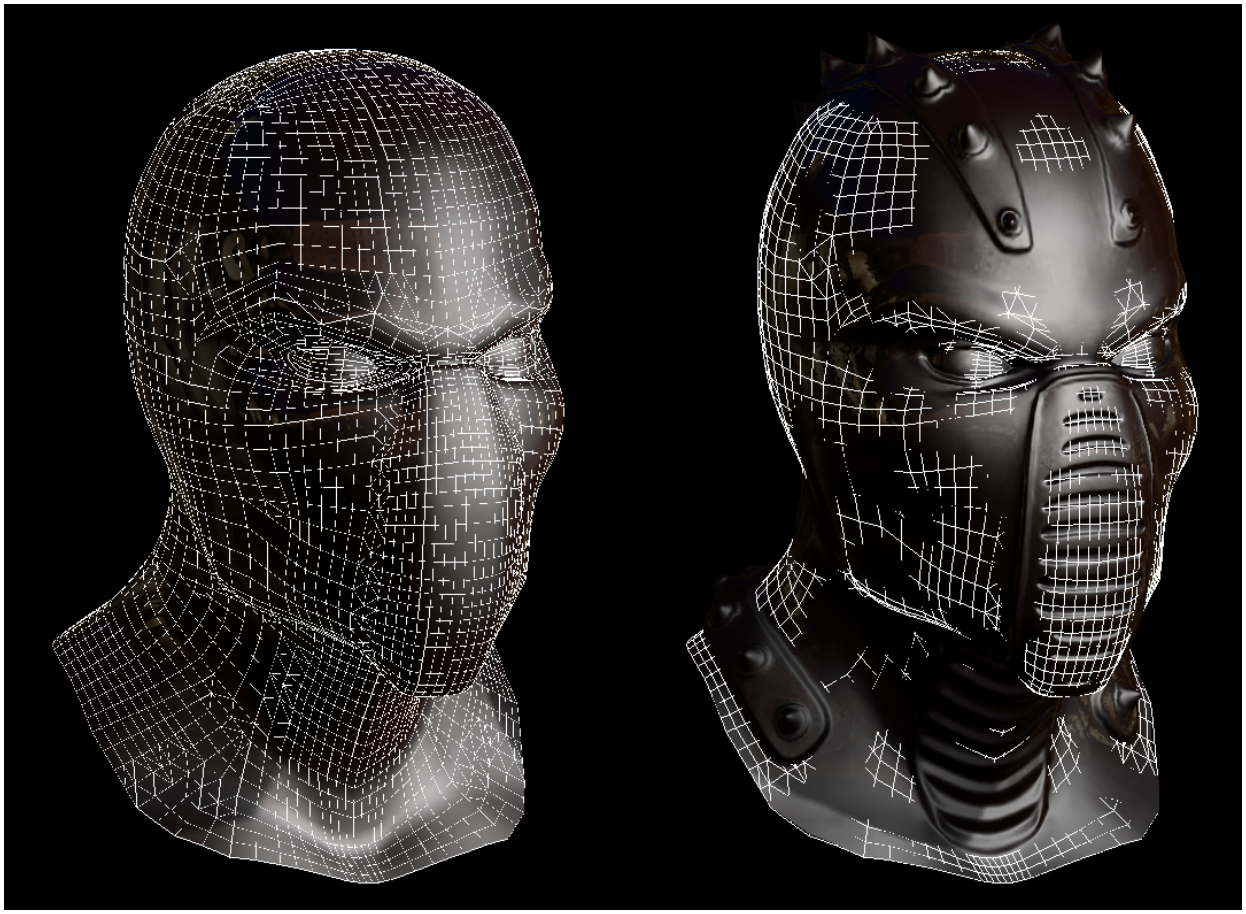
This idea of changing just the shading normals and not performing displacement was invented in 1978 by Jim Blinn, the same person who created the Blinn-Phong lighting model. The original term used for it was **bump mapping,** and back then the heightfield was sampled directly to get the shading normal. Normal maps were invented in the 1990s to make the algorithm work efficiently on the GPU.

When I learned about this technique in 1984 it took me awhile to get my head around it. It's a pretty non-physical concept: we've just made a flat surface, but somehow the angle of the surface points in different directions at different places.

I introduced displacement mapping first as a way to talk about making objects bumpy. The main advantage of displacement maps is that it's often easier to modify an image on the fly. So, for example, over time the horns could grow out of the head by simply changing the displacement map image.

[Additional Course Materials: One free displacement map to normal map converter is here http://forums.getpaint.net/index.php?/topic/18918-height-field-to-normal-map-textures-semi-community-tutorial/ . You can find free normal maps here http://opengameart.org/textures/5654]

[CUT Question: Map Baking]



Since displacement mapping happens at the vertex level, we could in fact bake the displacement map's effect directly into the triangle mesh itself. If the mesh is not changing, it's usually more efficient to do so. By baking, we don't need to evaluate the displacement map each frame. The normal map is all that's needed to get the head on the right.

In theory we could do the same thing for the normal map, simply bake in into the mesh. The tricky bit is that we'd have to make a much denser mesh, so that every texel on our normal map had a corresponding vertex where its normal could be stored.

What is the main problem with a dense mesh for this model?

( ) The fragment shader will not work without a shading normal.
( ) Making a lot more vertices can slow down the system.
( ) Triangle setup can introduce precision errors.

( ) Surface normals can become reversed if applied in this fashion.

Answer

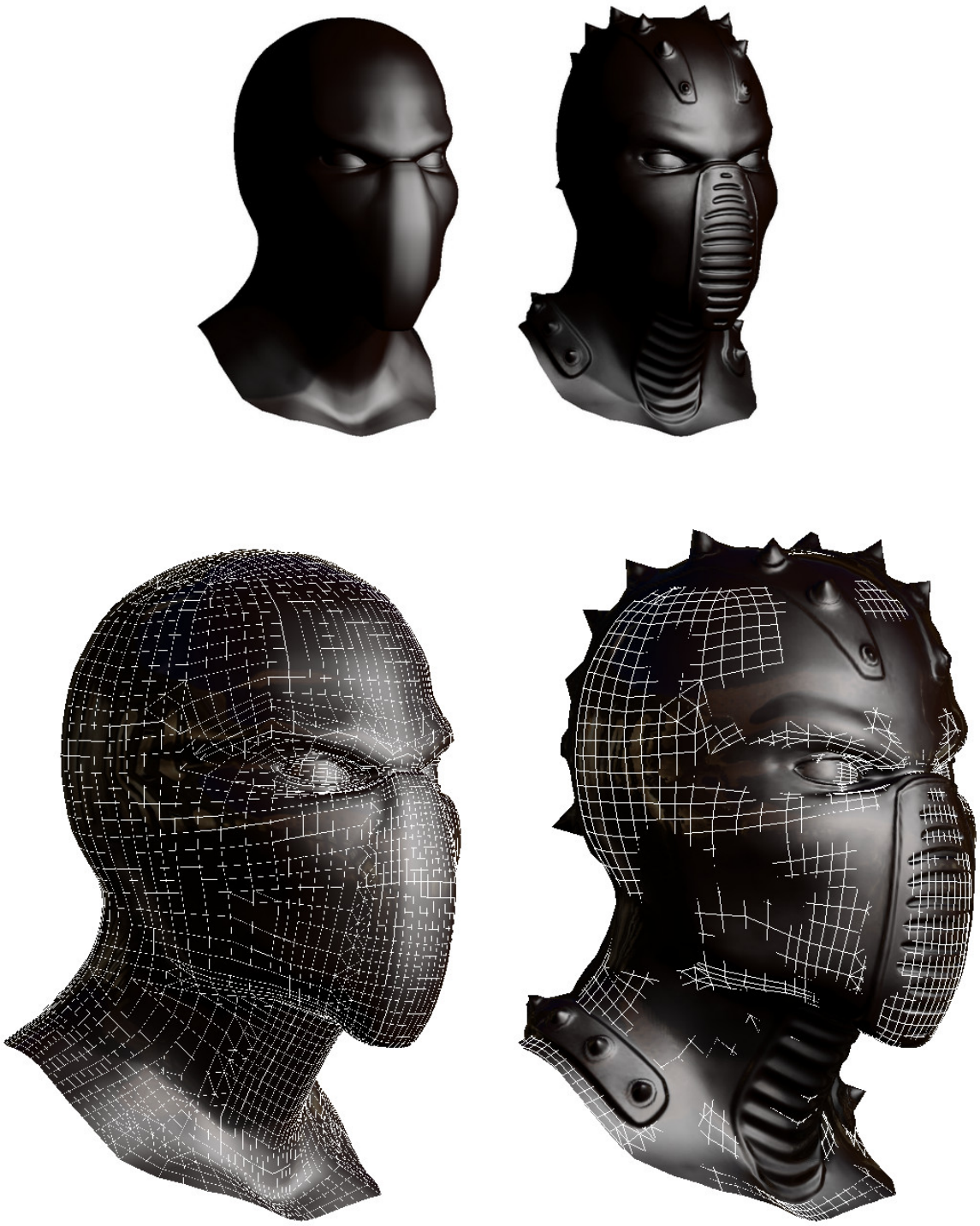
[#1, #4, #3] If the fragment shader doesn't work, or if surface normals get reversed, that's a bug that should get fixed. It's in fact true that triangle setup can introduce precision errors, but that has little to do with making more triangles.

[# 2] The correct answer is that throwing a large number of vertices at the GPU is decidedly inefficient. A huge mesh will also take up a considerably larger amount of memory, larger than the amount saved by getting rid of the normal map. Some triangles will be so small that they won't even cover a pixel center and so will be a waste to process. These are just some of the reasons that normal maps are used.

That said, large meshes *can* be used by the artist while creating the object. A program can then turn the normals into a normal map and then simplify the mesh to have fewer triangles in the final model.

Shortened Lesson: Displacement and Normal Mapping

[recorded 3/29, part 2]
[YOU CAN TOGGLE THE MESH WITH "W" KEY
unit6_displacement_map_reveal.html - note: not in demos
]

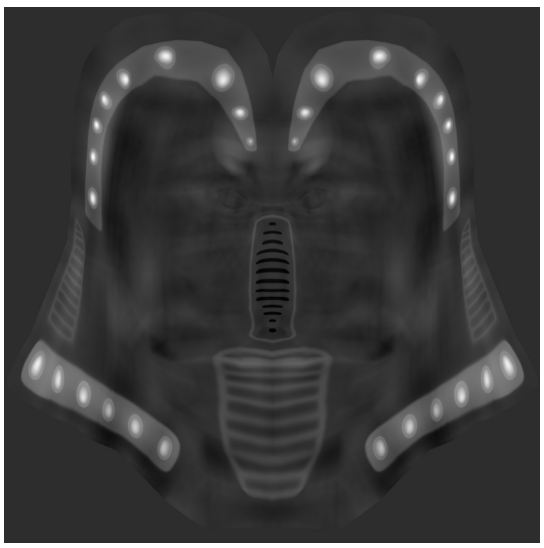


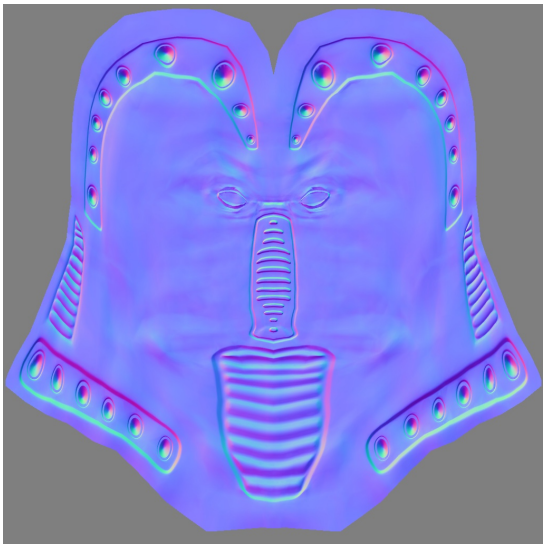
There are two other texture mapping methods that can add considerable detail to a model. The first is displacement mapping. Here a texture, called a heightfield, is used to change the height of the surface itself at each vertex. In other words, each vertex has a U,V value. This value is used to retrieve the height value from the texture. The height value is then used to displace the vertex, in other words move it upward or downward along its normal.

However, simply displacing the vertices does not change the shading normals of the surrounding surface. You would see the bumpiness in the silhouette of the object, but otherwise the shading would look fairly similar to the original.

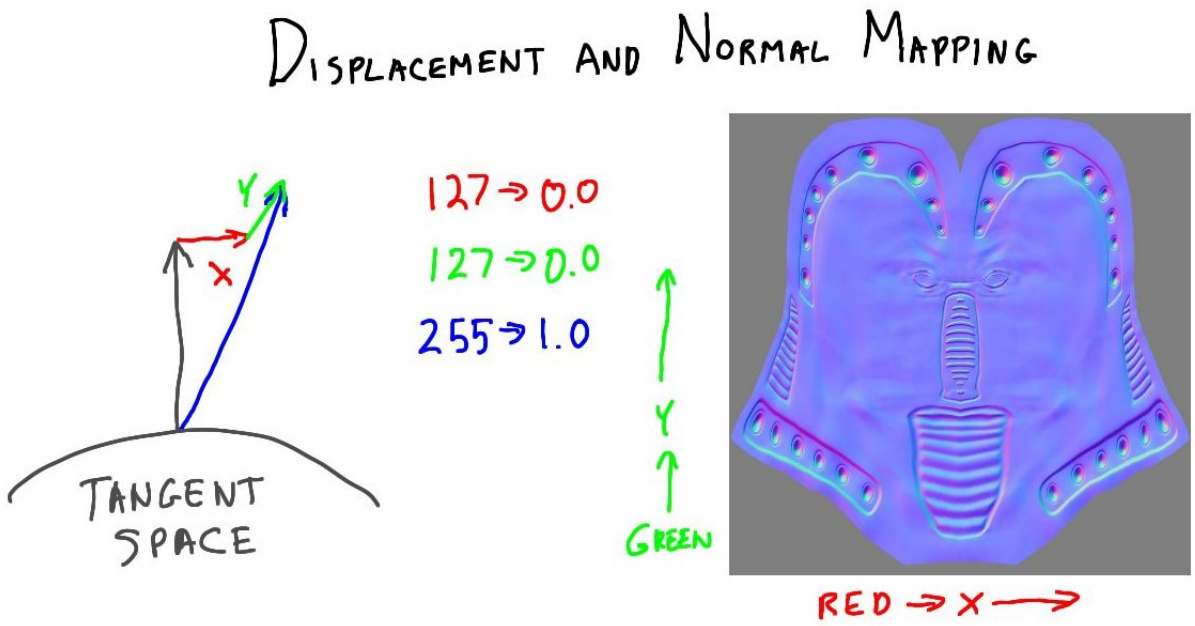
To change the shading normals themselves, another texture is applied to the surface. This texture is called a “normal map”, and is considerably different than other textures we’ve encountered up to this point.

[leave room for red and green color axes on normal map, and words “tangent space”]





Here’s the displacement map for the model, and here’s what the normal map looks like. The gray level of this heightfield shows the amount to displace the vertex, with brighter pixels pushed farther above the original surface. The normal map is in fact derived from this displacement map, and there are tools to create this type of texture. Each normal in the normal map is found from the differences in the nearby heights in the displacement map.



You can always tell a normal map by its distinctive light blue. The normal map stores an XYZ direction vector at each texel. When we view this XYZ value as an RGB color, we get this type of look. Specifically, the red channel is X, with positive to the right, the green channel is Y, with positive up, and blue is the Z axis. This XYZ space is relative to the surface of the model and is called "***tangent space***". If the normal is not to be changed at all, the normal stored is 0,0,1, straight up along the Z axis.

A value of 127 in this texture means zero, a value of 255 means 1. The light blue then comes from an X and Y value of 0, which gives a red and green channel of 127; the Z value is 1, giving a blue channel value of 255. So an unchanged normal has an RGB of 127,127,255, a light blue.

The basic idea is that you're using this normal value to change the existing shading normal. The normal in the texture map takes the place of the shading normal, and relative to the shading normal's original direction.

[unit6_normal_vs_displacement.html - note: not in demos]

Here's the interesting part: once you have the normal map, you don't have to apply the displacement map. They're separate processes: the displacement map moves the vertices, the normal map modifies the shading normals. Here we have the unmodified head, the head with just normal mapping, and the head with both. While the model with both mappings is the most convincing, using just the normal map gives you much of what you want but with many less triangles. In fact, artists often first create highly detailed geometric models. A program can then bake the normals into a normal map. The mesh can then be simplified, saving both processing time and memory while retaining much of the detail of the original.

Having a surface that is smooth but has differing shading normals is not something we usually see in the real world - in reality bumps are what cause the surface to change its orientation. However, it's fine in computer graphics to have one without the other. We've already been using the idea that the shading normal can be different than the geometric surface normal. Here we're extending the idea, varying the shading normal per texel instead of per vertex.

See the additional course materials for more information about this process.

[Additional Course Materials:
My original lesson plan spent more time on this area. See the [Lesson Scripts](https://www.udacity.com/wiki/cs291#course-notes) if you want to see the initial, considerably longer lessons.

The original version of the heads program can be found [here](http://mrdoob.github.com/three.js/examples/webgl_materials_normalmap.html). Note that this program may not run on older computers, as it requires Shader Model 3.0 level of GPU support.

Evan Wallace's lovely [water demo](http://madebyevan.com/webgl-water/) in WebGL shows

displacement mapping in action.
]

Where Does Displacement Happen?

[recorded 3/29, part 2]

A question for you: ***where does the displacement of the surface happen?*** That is, which part of the pipeline deforms and shifts the surface itself?

( ) The vertex shader
( ) During rasterization
( ) The fragment shader
( ) The Z-Buffer

There's only one part of the pipeline that can do this sort of operation.

Answer

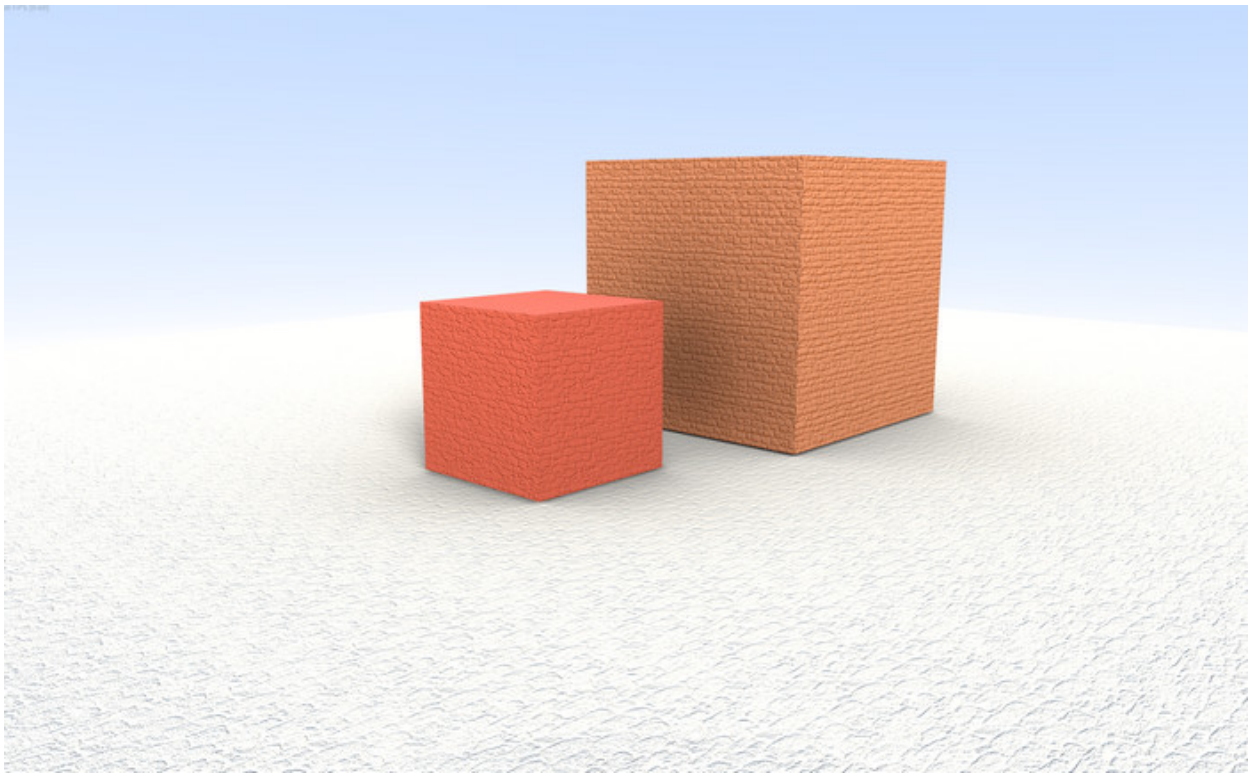
[recorded 3/29, part 2]

The answer is that this displacement happens in the vertex shader. This shader has control of what happens to each incoming vertex. The vertex shader reads the displacement map texture at the vertex's location and uses it to displace the vertex. The other parts of the pipeline take these displaced triangles and process fragments created by them. Each fragment has a pixel location and cannot change to another pixel location. The fragment shader can apply the *normal* map, but this operation does not change the location of the vertex.

Lesson: Light Mapping

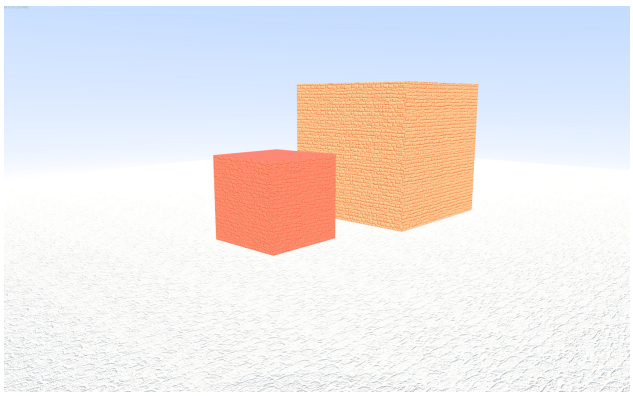
[recorded 3/29, part 2]

[from http://mrdoob.github.com/three.js/examples/webgl_materials_lightmap.html, but just use the still image!]

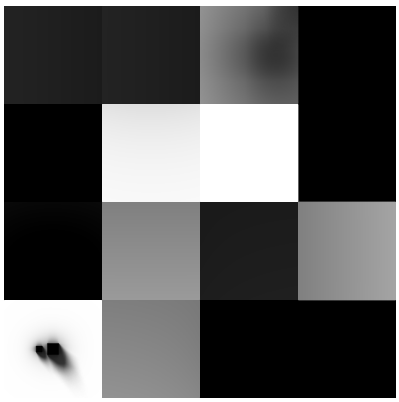


Light mapping is a way of lighting objects to make them look highly realistic, or for that matter, highly stylized. For static objects in particular, those that aren't moving, the idea is to precompute shadows, reflected light, and any other lighting effects desired. You can see in this scene that the smaller cube has a subtle, realistic shadow effect on the larger one, for example. This type of effect is difficult to compute at interactive rates.

[NOTE MULTIPLY SIGN - should be in middle in real thing.]



X

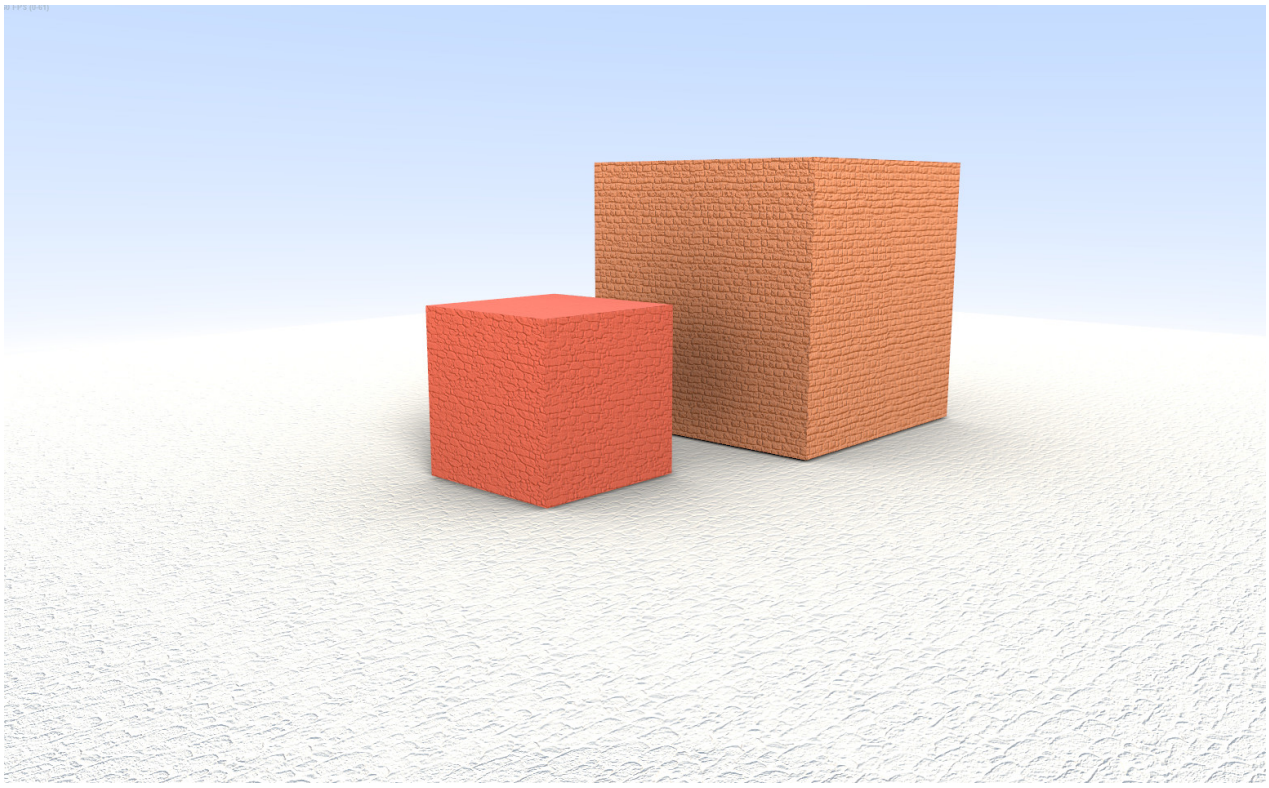


[***light map***]

This is how the process works. First, here's the scene without any lighting applied. Every

surface is assigned an additional texture, called a ***light map***. This extra texture will be used to capture the light's effect. Some offline process is done to the illumination in the scene. As much time as desired can then be spent determining how much light reaches each pixel on each surface. The results are stored in the light map. The light map is a mosaic texture, holding many different surfaces' illumination calculations. For example, the lower left corner of this light map is for the ground plane. You can see the two shadows of the cubes laying on it.

When the objects are displayed, the fragment shader takes the object's color texture and multiplies it by its correspond light map texture. This gives the surface an illuminated look.



Here's the result. Notice that it's the same or darker that the original image without illumination. This is why light maps are sometimes half-jokingly called "dark maps", since they lessen the overall light.

In theory you could bake the lighting into each color texture itself instead of using a separate light map texture. While possible, this can quickly get expensive. One major advantage of light maps is that they can have a much lower resolution than the texture on the surface itself, and can be stored in a single color channel instead of three. Rather than storing six high-resolution textures per cube in this scene, each with the lighting baked in, just one texture and six small light maps are stored for each.

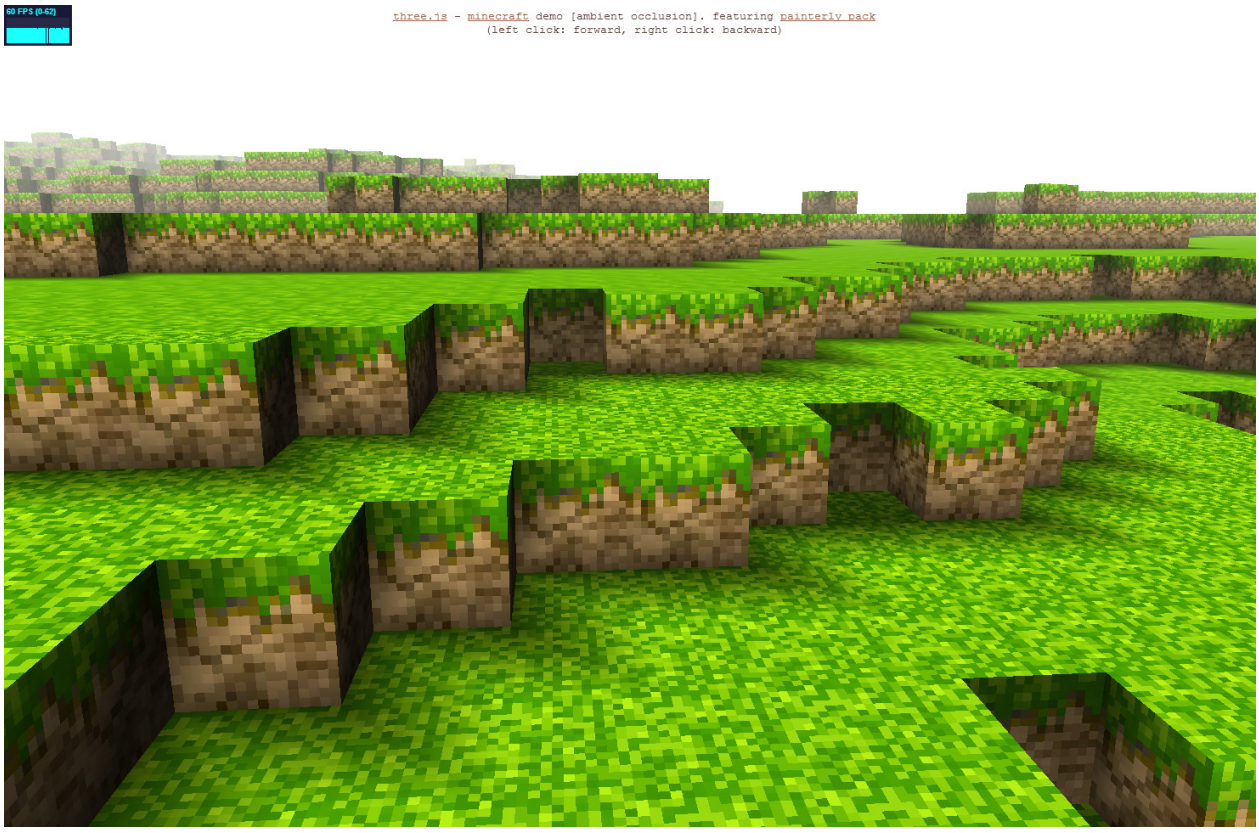
Lesson: Ambient Occlusion

[recorded 3/29, part 3 - like screenflick 17]
[add terms ***global illumination (GI)*** and ***ambient occlusion (AO)***]
[add ***screen-space ambient occlusion (SSAO)***]

I haven't said much about the process by which light is applied to the surfaces. One class of methods is to use some form of global illumination algorithm, such as path tracing.

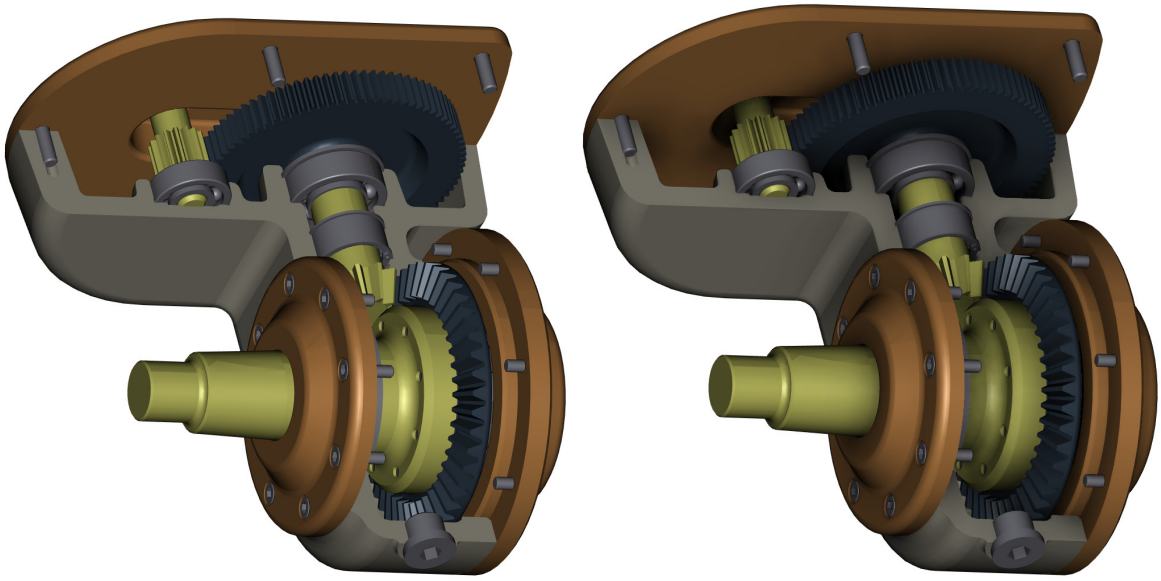
Another technique is called "ambient occlusion", which is more concerned with the geometry in the scene. Instead of percolating light through the world, ambient occlusion techniques look for crevices, valleys, and other areas where light is less likely to penetrate.

[http://mrdoob.github.com/three.js/examples/webgl_geometry_minecraft_ao.html - Run demo and slowly walk around, look at crevices]



Here's a simple example of ambient occlusion. This particular form stores at each vertex how much that vertex is in a corner or crevice. These values are then interpolated across each cube face and blended in.

[EffectsSandbox_SSAO.wmv - video editor, please use the first part of this video and the last part, basically 50/50 for this piece of spoken text. Remove the part where the menu is shown in the middle of this video.]



One last thing I should mention: there is a class of algorithms for interactive rendering called screen-space ambient occlusion. These techniques work for the current camera view, which is where the name “screen-space” comes from. They examine the Z-buffer depths stored nearby each surface and darken any nooks and crannies found. While just an approximation, this effect can look quite convincing.

[Additional Course Materials:
The demo that created these images is [here](http://mrdoob.github.com/three.js/examples/webgl_materials_lightmap.html). Ambient occlusion has some basic coverage on [Wikipedia](http://en.wikipedia.org/wiki/Ambient_occlusion), as does [screen-space ambient occlusion](http://en.wikipedia.org/wiki/Screen_space_ambient_occlusion). To see ambient occlusion and screen-space ambient occlusion in use, along with many other effects, see [the Agni’s Philosophy page](http://www.4gamer.net/games/032/G003263/20121201006/) - I recommend clicking the “translate” button.
]

Lesson: Time to Explore

[recorded 3/29, part 3]
[show some utterly cool demo here, e.g.
http://alteredqualia.com/three/examples/webgl_city.html - asked Altered Qualia for permission, and granted]

At this point I'm going to set you free. The lessons for the rest of this unit will be showing a number of demos related to more advanced effects. Most of them have code samples you can look at and see what makes them tick. Rather than me describing the five things you need to do to enable textures and hook them up, just look at the code and documentation. Three.js makes many of these effects surprisingly easy to use out of the box.

Be assured that there will still be some exercises in the problem set to get you applying some basic texturing to surfaces. However, the effects in the lessons ahead are such that I'd feel a bit guilty making you connect some bits of code together just to see the syntax. This is where the rollercoaster really starts to go wild and the best way for you to learn is to rip into the code yourself and see how it works.

[Additional Course Materials:
The demo shown can be found [here](http://alteredqualia.com/three/examples/webgl_city.html) - warning, music will play.
]

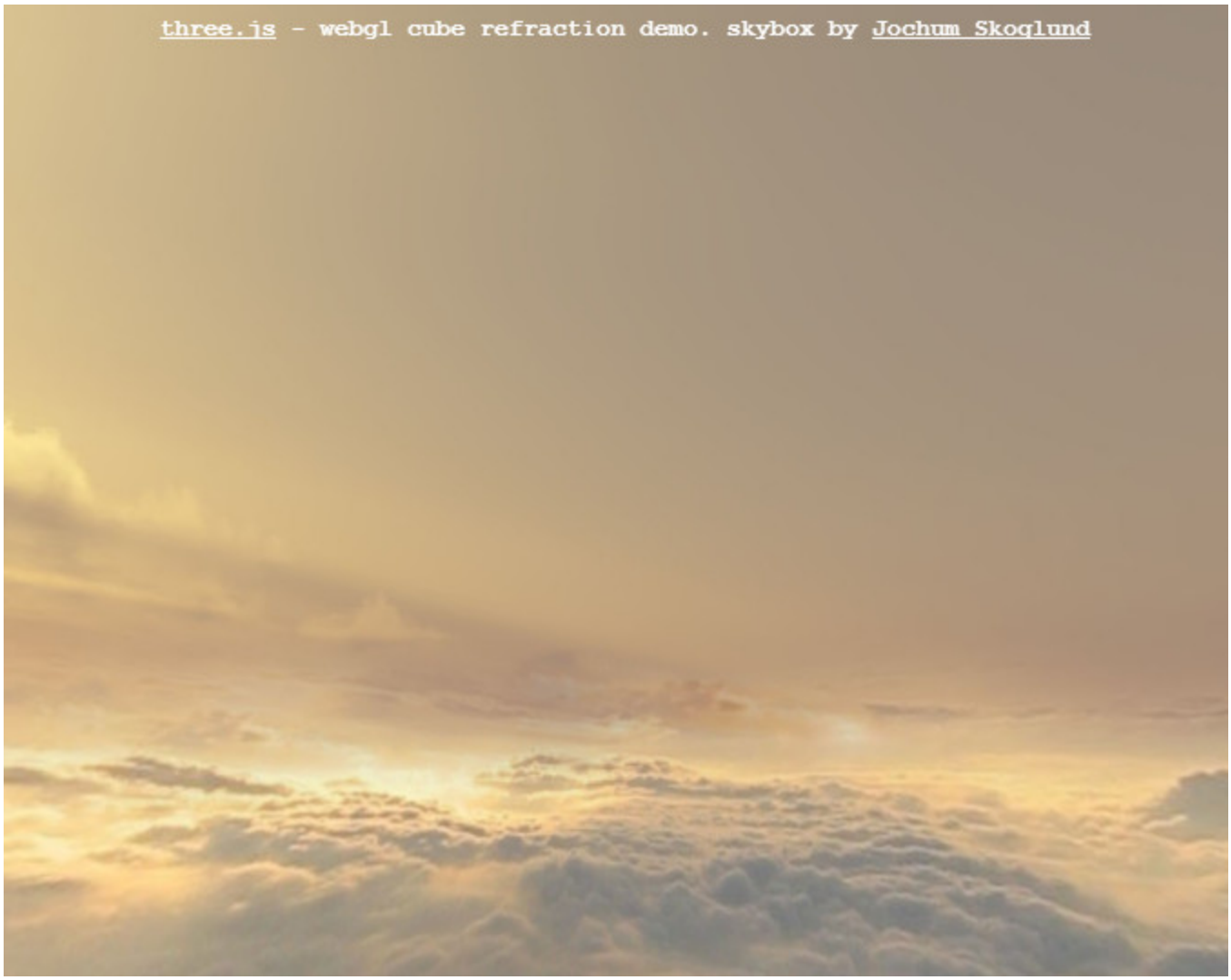
Lesson: Skybox

[recorded on 3/30, to end of unit (not including PS)]

[START WITH CUBE BOX OFF, TOGGLE WITH W - basically, run it a long time, then toggle it on and run for awhile unit6_webgl_materials_cubemap_skybox.html - note: not in demos

[This is just the following demo with the spheres turned off, and with Wire toggleable:

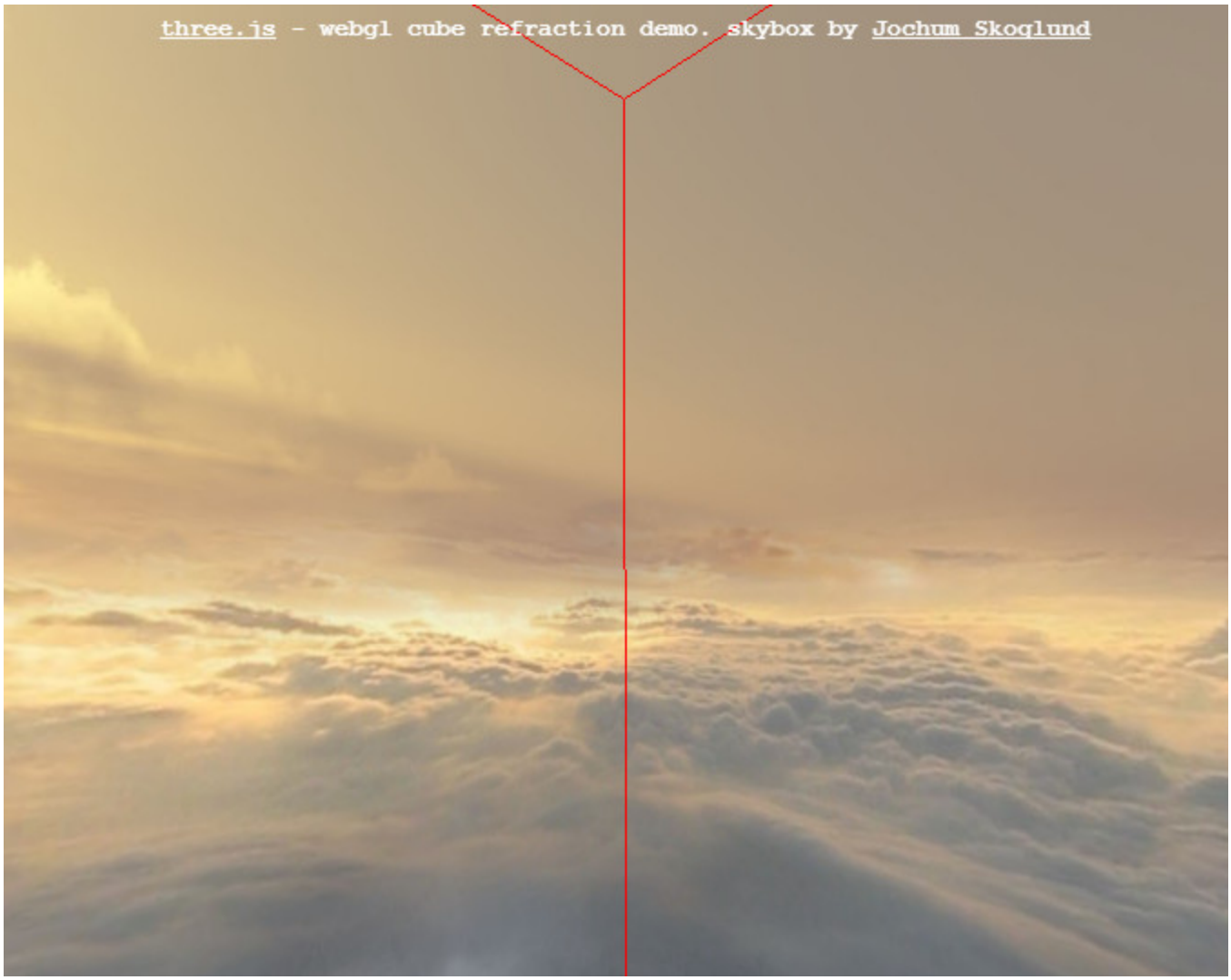
http://mrdoob.github.com/three.js/examples/webgl_materials_cubemap_balls_refraction.html]



Instead of applying a texture to a surface, we can apply a texture to represent the world itself. In the distance are clouds, in all directions. The way this illusion is done is with a skybox. The idea is to put the viewer in the center of a box and put the environment on the walls, floor, and ceiling.

Doing this reminds me a bit of the movie “The Truman Show”, where Truman’s world is highly limited and he runs his ship into - well, I don’t want to spoil anything, if you haven’t seen it. The key is that you have to be careful to not let the viewer get too close to the sky so that he doesn’t figure out it’s just an illusion.

[hit the W key to toggle]



If I turn on the wireframe for the cube, the trick is revealed. You can see the walls of the box. Without the wireframe the illusion is seamless. If the viewer moves around, the center of the box moves with them. In this way there is no distortion, as the view of the walls never changes.

[py
pz px nz nx
ny

DRAW A LITTLE CUBE IN THE LOWER RIGHT CORNER, label some faces CORRECTLY

+y
+z +x -z -x
-y
]

cube map

+y

+z ***+x*** ***-z*** ***-x***

-y

This type of texture is called a ***cube map***. It's made of 6 images that form a cube. The viewer is always placed at the center of the cube.

The cube shape is not required, it's possible to have other geometry, such as a hemispherical dome around the scene. Whatever the shape, our eyes can't see the z-buffer, so can't tell that the sky is only, say, 100 meters away.

[speak slowly, the video is longish, but doesn't need to run its whole length.Video at Showcase video Unit7_Skybox ShowcaseEnvironments.wmv
]

Skyboxes are typically used for starfields or, well, the sky, for times when you're not near the ground. Alternately, you can simply make the surrounding skybox a part of the scene, such as shown here. Real geometry has the advantage that it can receive shadows from the object. The main rule is that you want to make sure that nothing pokes through the walls of the skybox itself.

Oh, and you definitely want to set the "far" distance of your camera so that it encompasses the skybox itself, otherwise it will be clipped out!

[Additional Course Materials:
There is [a commented tutorial by Lee Stemkoski on skyboxes](http://stemkoski.github.com/Three.js/Skybox.html). You can also find [skybox code with the three.js distribution](http://mrdoob.github.com/three.js/examples/webgl_materials_cubemap_balls_refraction.html).

See [Humus' site](http://www.humus.name/index.php?page=Textures) for a large variety of freely-reusable reflection cube maps and other textures.

For a nice implementation of a surrounding environment, see [this demo](http://oos.moxiecode.com/js_webgl/water_noise/).
]

[Leftover: how to make a skybox. Easy enough to look up, so not an exercise.

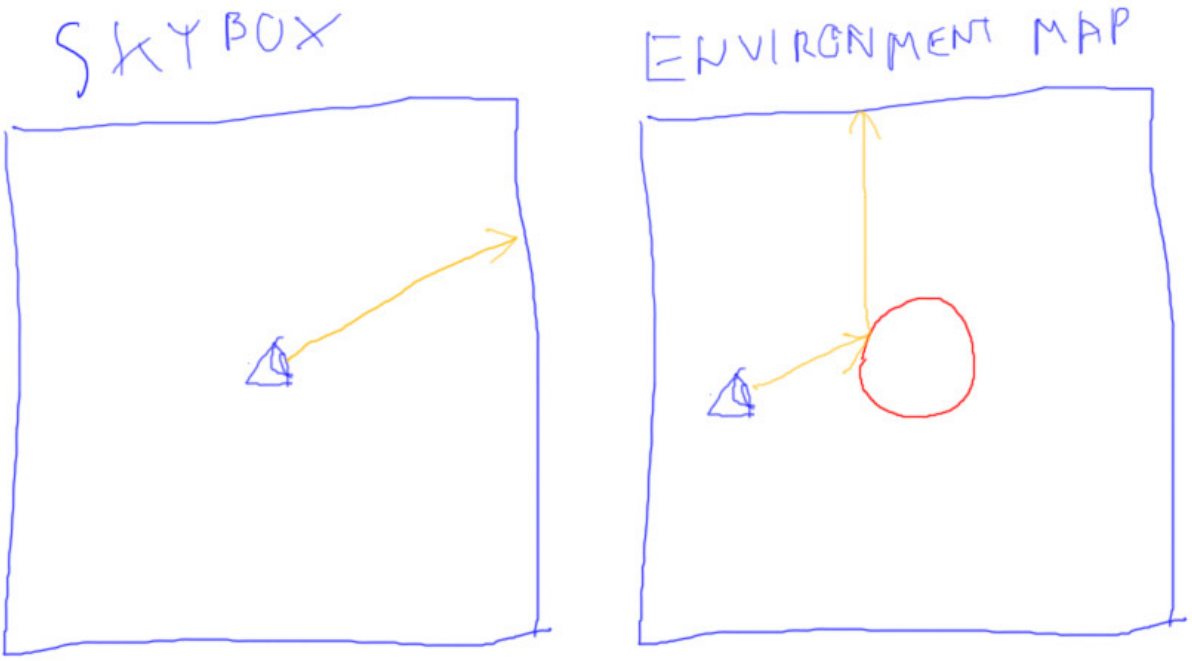
```
        // make sure the camera's "far" value is large enough so that it will render the skyBox!
        var skyBoxGeometry = new THREE.CubeGeometry( 10000, 10000, 10000 );
        var skyBoxMaterial = new THREE.MeshBasicMaterial( { color: 0x9999ff } );
        var skyBox = new THREE.Mesh( skyBoxGeometry, skyBoxMaterial );
        skyBox.flipSided = true; // render faces from inside of the cube, instead of from outside
(default).
        scene.add(skyBox);
]
```

Lesson: Reflection Mapping

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

There are many variations of using textures to modify the surface's parameters in the lighting equation. Another way to vary the surface while calculating illumination is to apply textures as *light sources*. We've seen how a skybox can be made to surround a scene. An extremely clever technique is to have surfaces reflect a skybox.

[drawing of skybox around an eye, and around a reflective sphere viewed by an eye. NOTE: add green normal to surface for reflection. PUT EVERYTHING of a given color in a separate layer. We'll use later for diffuse map.]



This technique is called ***reflection mapping***, or sometimes ***environment mapping***. The basic idea is to make one object surrounding the world something that other objects can reflect. This is called the ***environment map***. The illumination process is a bit like ray tracing, where a ray is reflected off a shiny surface and picks up the color of any reflected objects. The difference is that there's only one simple object to reflect, typically a cube map.

Unlike a skybox, the environment map is used only when a shiny surface needs a reflected color. You can certainly also put a skybox in a scene using the same cube map, this is commonly done in order to produce a convincing effect. However, the environment map is a separate object that is a part of a shiny object's material description. That said, skyboxes and reflections maps work using the same idea. You can think of a skybox as a physical object, which is what we usually do, or you can think of it as a texture function: what is the color of the skybox in the direction the eye is looking? An environment map is exactly the same, only for reflected rays: what is the color of the environment map in the reflection direction?

How it works is this: we render a fragment on a reflective surface, such as a sphere. We compute the usual elements as desired, such as diffuse, ambient, and so on. For the environment map we also need the direction to the eye and the shading normal at the fragment. These two vectors are used to compute the reflection direction, which is then used to find the texel on our environment map. This texel color can then be blended into the final color of the fragment. The color from the reflection map makes the surface appear mirrorlike.

The very simplest cube map to reflect is one that is “at infinity”. Remember how a directional light came from a certain direction but was considered infinitely far away? Directional lights are inexpensive to use because their direction vector never changes for any surface. Similarly, if you define an environment map as being “at infinity”, you’re saying that “no matter where I am in my virtual world, when I look a certain direction I want to always get the same color back from my environment map.”

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

To sum up, the reflection direction is computed for a reflective surface and this direction is then used to access the environment map. This algorithm has limitations somewhat the reverse of the skybox. The reflected objects are always seen as being infinitely far away. For example, say you have a reflective sphere in a room represented by an environment map and a skybox. If you move the sphere from the center of the room to near a wall, the reflection of the wall will not look closer.

That said, reflection mapping is an inexpensive way to add convincing visual detail to a model. Our eyes are usually not very good at reversing the process of reflection. We see something reflected and our brain interprets the object we’re viewing as being shiny. We don’t waste brain power trying to figure out if the reflections are perfectly correct, since in the real world they always are.

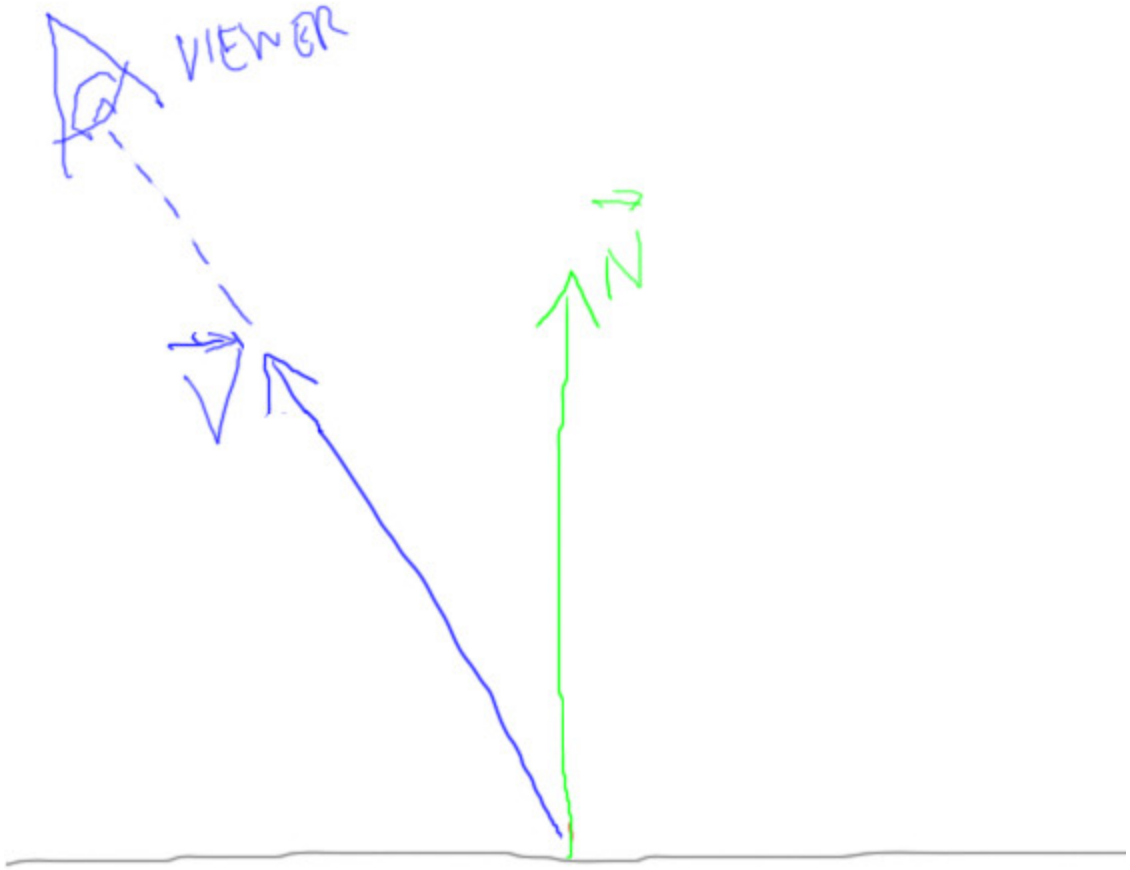
[Additional Course Material:
For commented code on how to set up reflection mapping, see [Lee Stemkoski's demo](http://stemkoski.github.com/Three.js/Reflection.html).

See [Humus' site](http://www.humus.name/index.php?page=Textures) for a large variety of freely-reusable reflection cube maps and other textures.

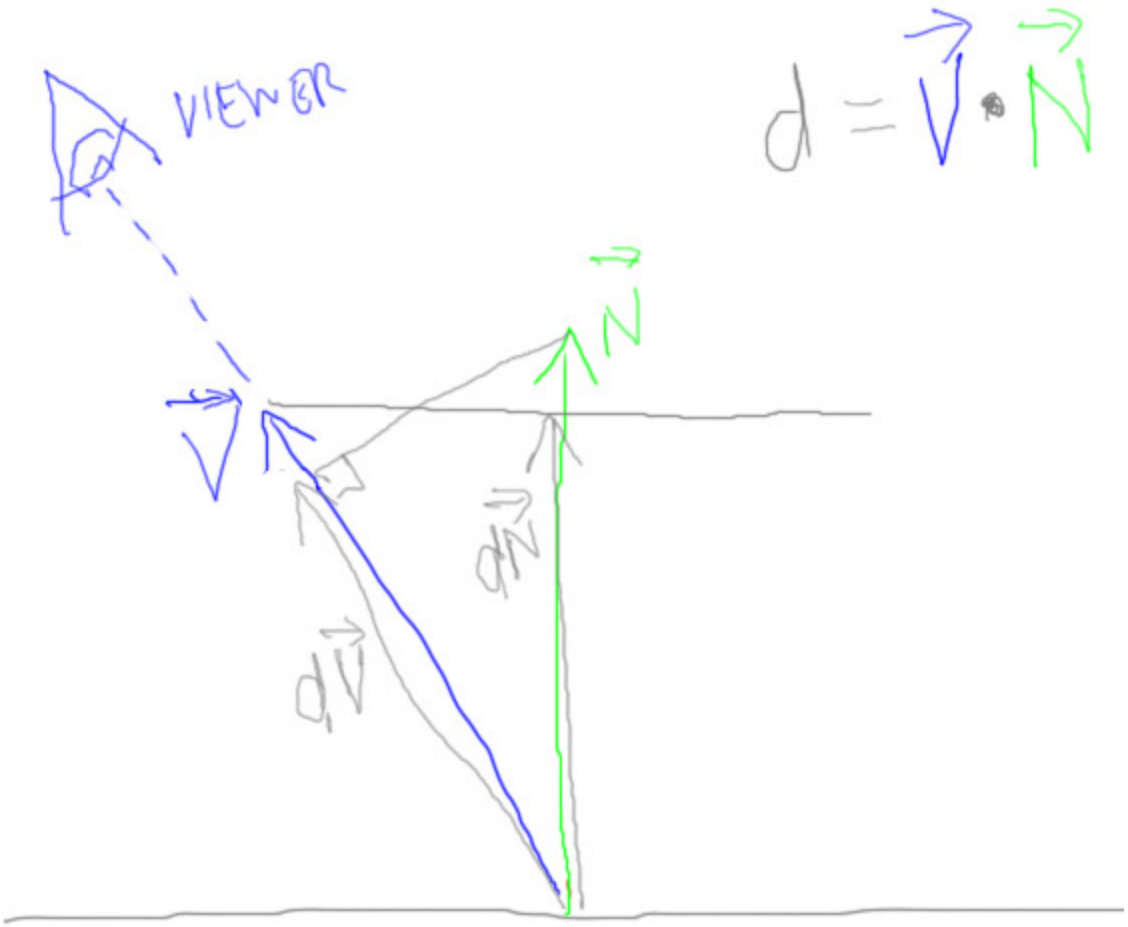
Demos shown in the video were done with three.js, [here](http://mrdoob.github.com/three.js/examples/webgl_materials_cars.html) and [here](http://mrdoob.github.com/three.js/examples/webgl_loader_ctm_materials.html). Here's [yet another car demo](http://jeromeetienne.github.com/tquery/plugins/car/examples/). It's fun in that you can drive the car around with the arrow keys. One more nicely-done car is [here](http://carvisualizer.plus360degrees.com/threejs/).
]

Exercise: Reflection Equation

[Careful figure; make sure to ***draw vector arrows over values***. List N, dN, V, dV as possibles. BUG: Draw R vector last.]



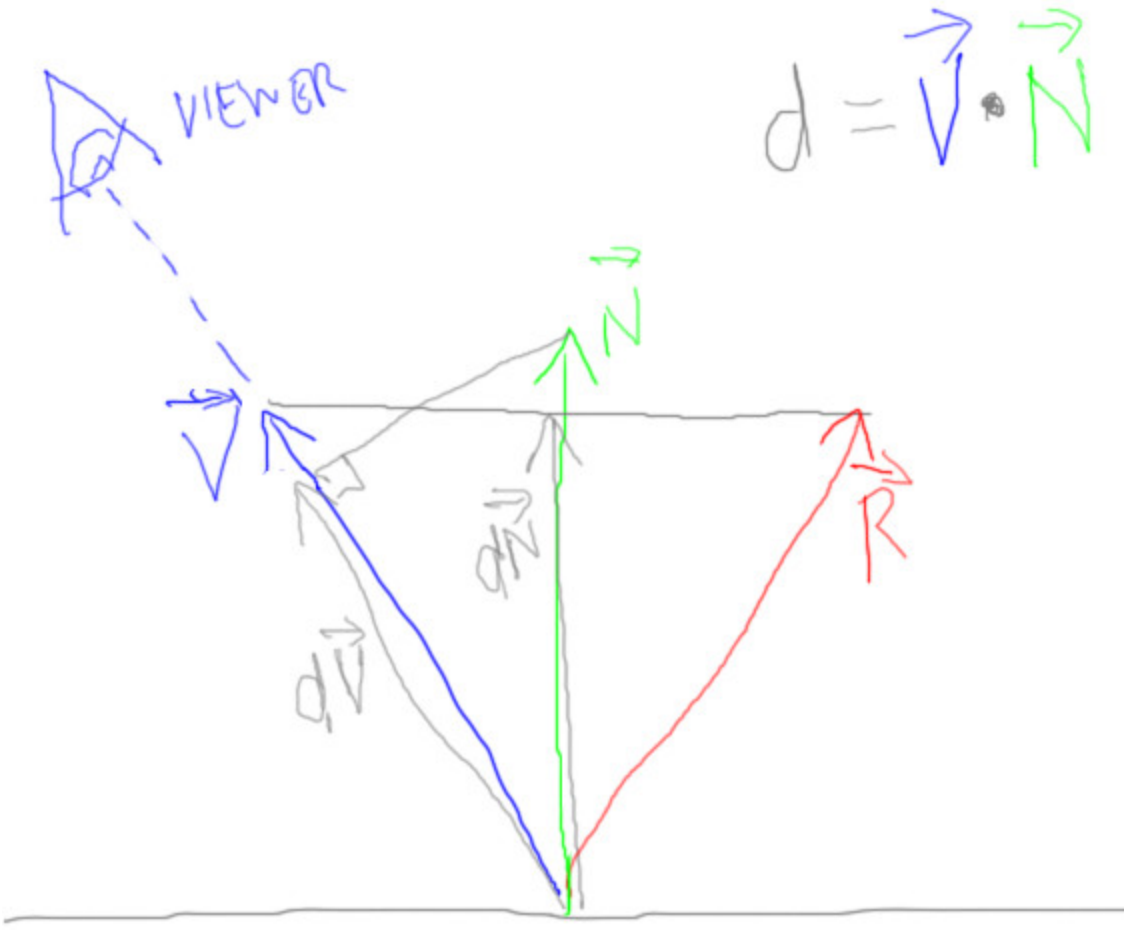
Say you're given the following four vectors for a reflective surface: a normalized vector towards the view, V, and the normalized mirror's surface normal N.



The dot product between the view vector V and the normal N is computed and stored as “d”. You’re given two other vectors: dV, which is the V vector multiplied by this dot product, and dN, the normal multiplied by this dot product.

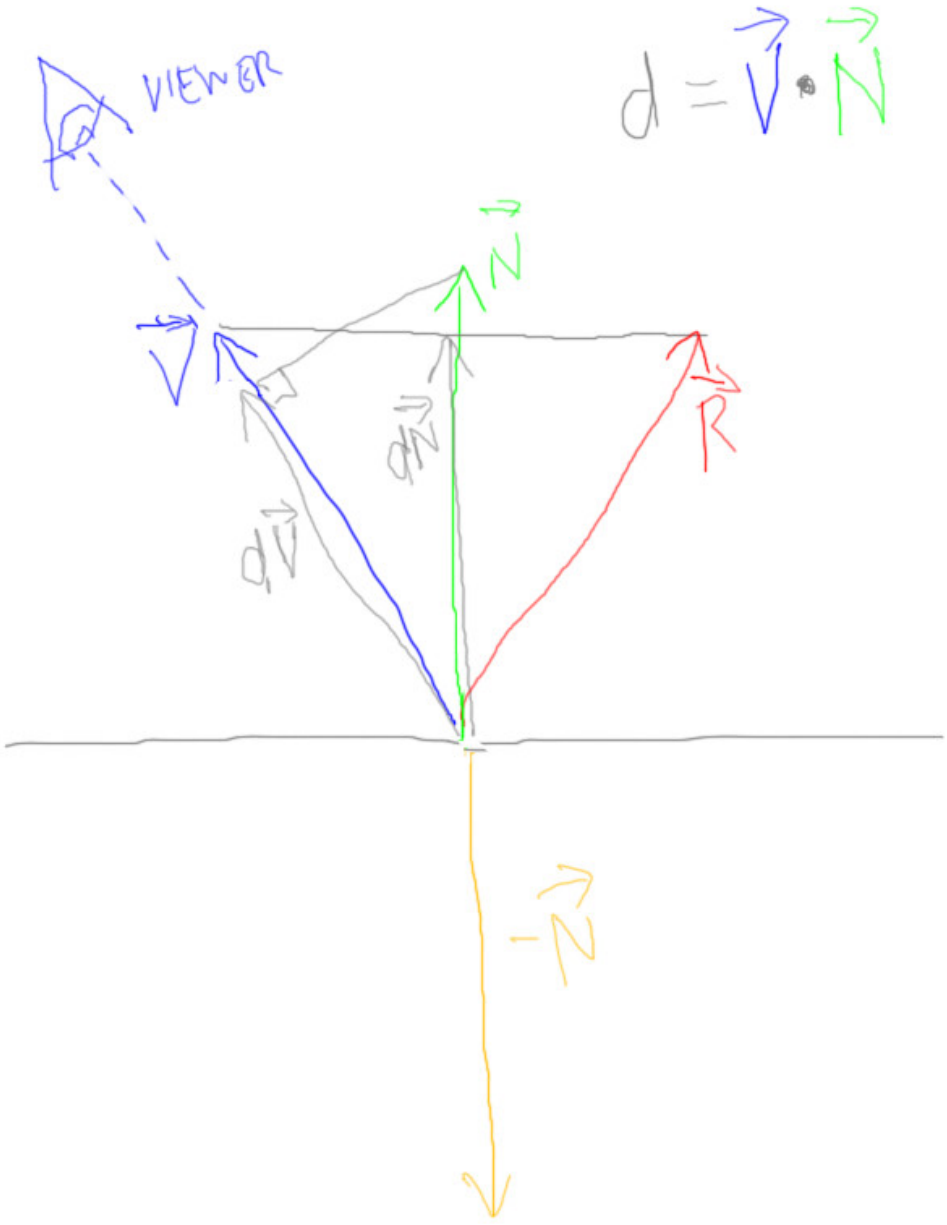
You can see these two vectors in the diagram. For example, dV is shown as the projection of N onto the V vector; dN is shown as the projection of V onto the N vector.

[Now draw R vector]



The question to you is, given these four vectors, how do we compute the normalized reflection direction vector R? You can use any and all of the four vectors in your answer, which should consist of four integers. These integers can be positive, negative, or zero.

[show negative N vector, -1, 0, 0, 0 answer in yellow]



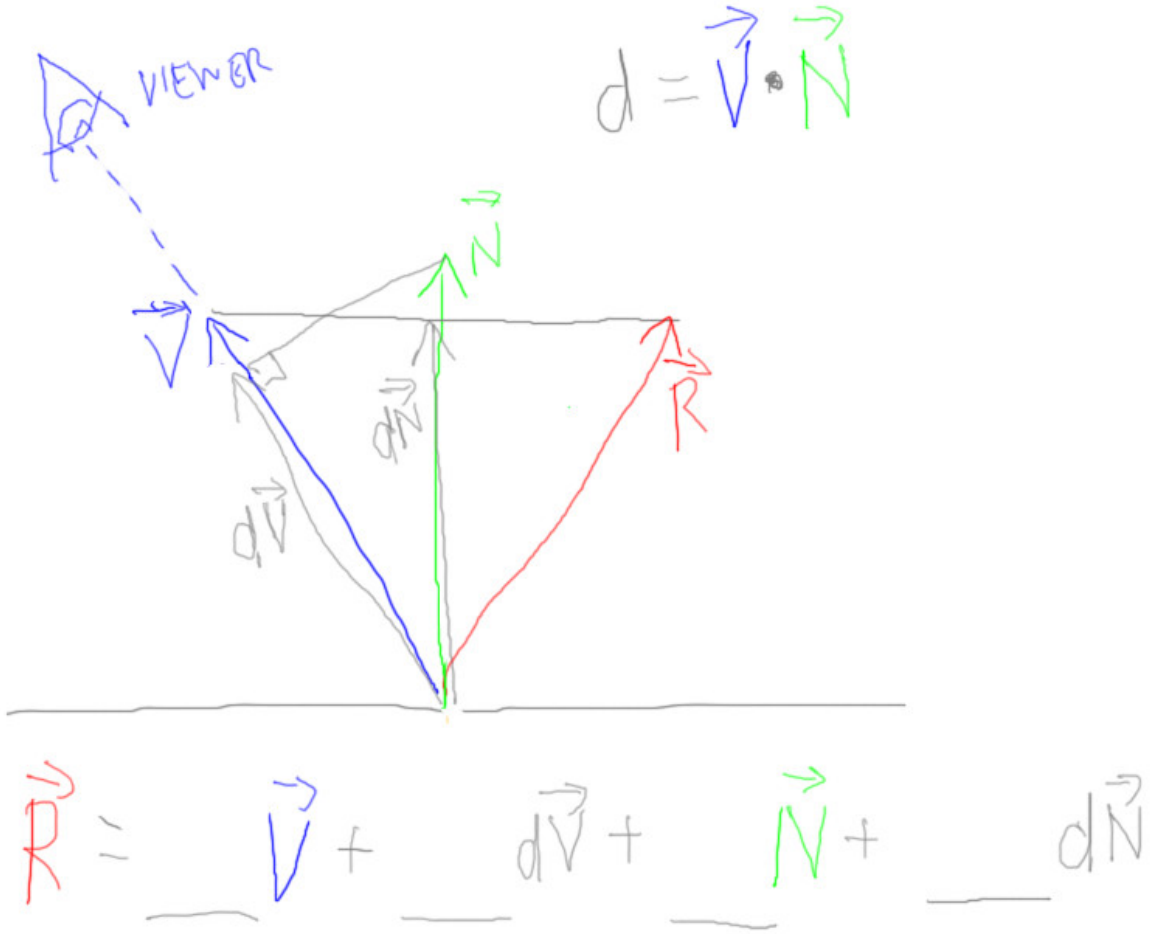
For example, if you think a single copy of the dN vector negated is the same as the R vector, you would put 0, -1, 0, 0.

Here's a hint: this answer is not correct.

[Erase bad answer]

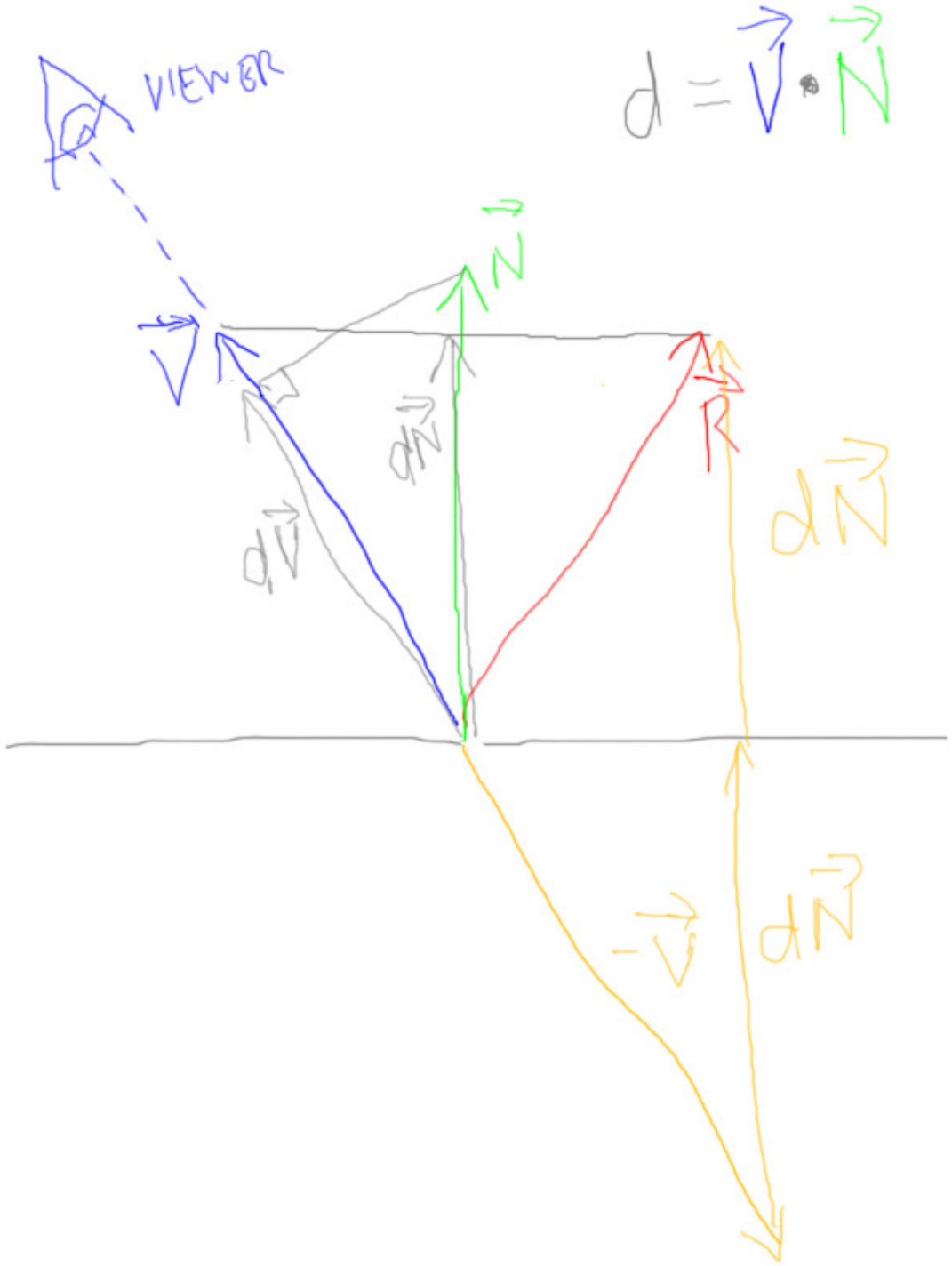
What values form the right answer?

[make sure to show arrows over vectors!]
[FIX: ***It is better if we can put question itself to the right of the drawing!*** Leaves room to show work. ALSO, make order N, dN, V, dV]



Answer

[draw on figure; answer is ***-1, 0 (or blank), 0 (or blank), 2***]
[CORRECT, with new order: draw on figure; answer is 2dN - V: ***0 (or blank), 2, -1, 0 (or blank)***]



Looking at the figure, I can see I'm going to need to go to the right somehow. The V vector looks like a good place to start, negated. I put a -1 for this V vector.

That got me going to the right, now I need to go upwards. I could add the vector N twice, but this

would take me too high; after adding N just once I can see that I'm above the mirror's surface, so another N will take me beyond my goal of R. The vector dN looks like a winner, as this is essentially just the Y component of the V vector. Adding it to a negated V vector is the same as simply moving to the right. Adding dN twice gets us up to the R vector, so 2 dN vectors are needed.

[make sure to show arrows over vectors!]

The final formula, then, is ***R = 2*dN - V***, or to expand out, ***R = 2 * (V DOT N) * N - V***

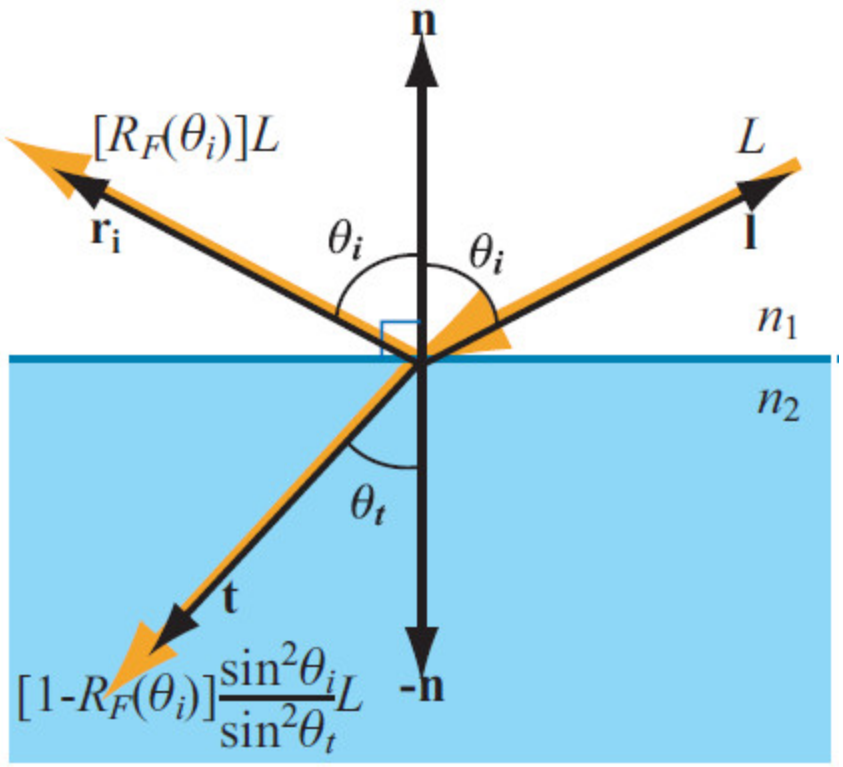
Lesson: Refraction Mapping

[http://mrdoob.github.com/three.js/examples/webgl_materials_cubemap_refraction.html - angels, look from above to see colors]

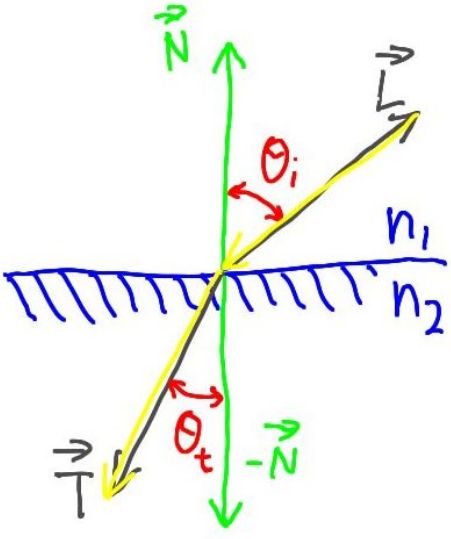
Just as you can create a reflection vector to perform reflection mapping, you can also create a refraction vector and use that instead to access the environment map. This can be combined with the color of the object to get different variations of a material that looks a fair bit like glass.

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

[don't draw reflection ray. ***Draw vectors with arrows over them***. note n1 and n2 are index of refraction(s)]

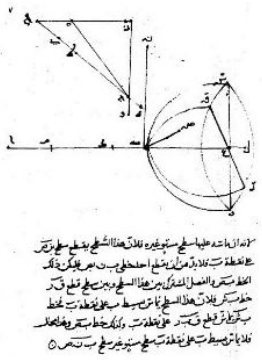


Refraction Mapping



Snell's Law: Ibn Sahl:

$$n_1 \cdot \sin(\theta_i) = n_2 \cdot \sin(\theta_t)$$



Snell's law is what is normally used for computing the refraction direction. This physical law is:

n1 * sin (theta-i) = n2 * sin (theta-t)

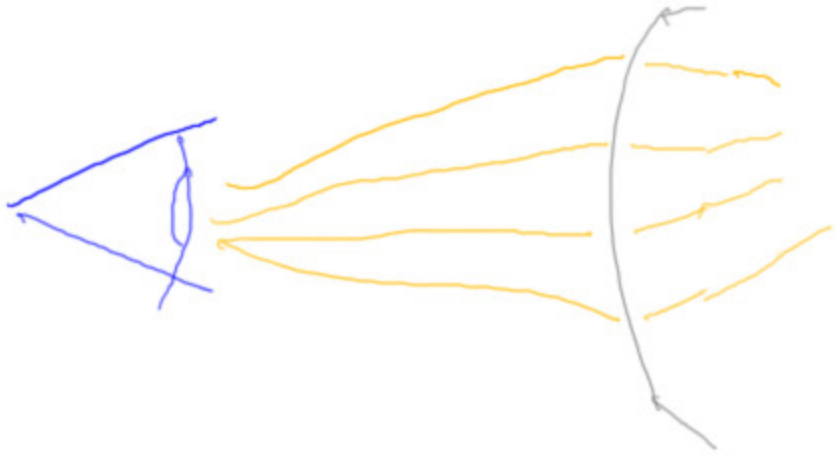
In the diagram, we have a ray of light traveling through, say, the air and hitting a transparent object, say made of glass.

In this equation, n1 and n2 are the index of refraction of the air and glass, respectively. The index of refraction is a physical value for a material that is essentially how much slower light travels through it compared to light in a vacuum. As examples,

air's index of refraction is ~1.0
water is 1.33
glass is 1.52 or so, depending
diamond is 2.42

When the index of refraction of the second material is greater than the first, the effect is that the sine of the angle must be smaller, to compensate. This means the angle from the normal becomes smaller, to compensate. In other words, light bends towards the normal when going from air to glass. The effect is reversed in the other direction: going from glass to air causes the light to bend at a greater angle.

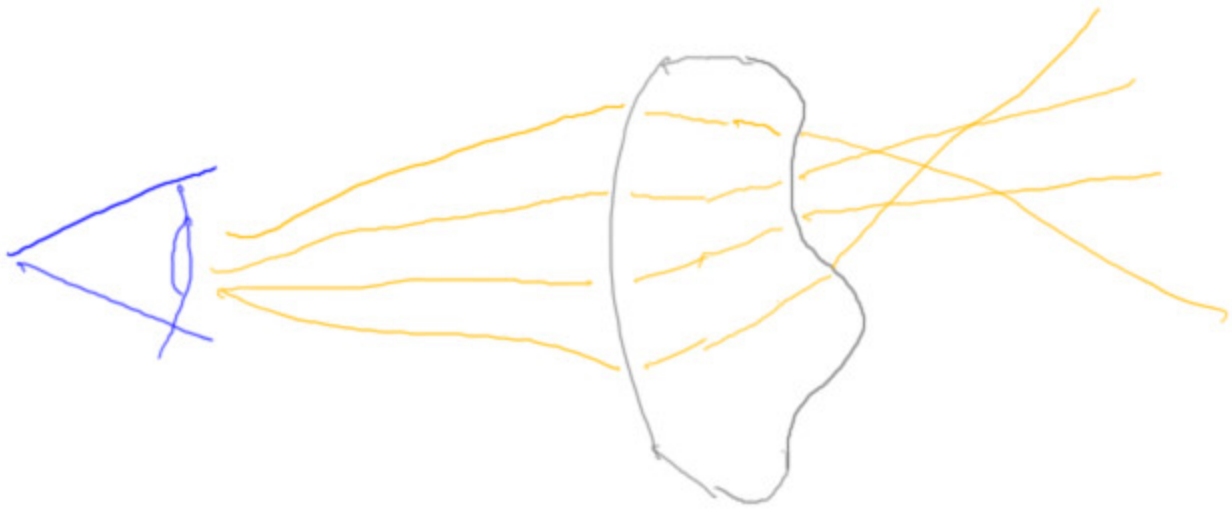
[draw the front part of a lens and show rays bending when they enter]



This is what refraction mapping does, it changes the direction of the light passing through a transparent object.

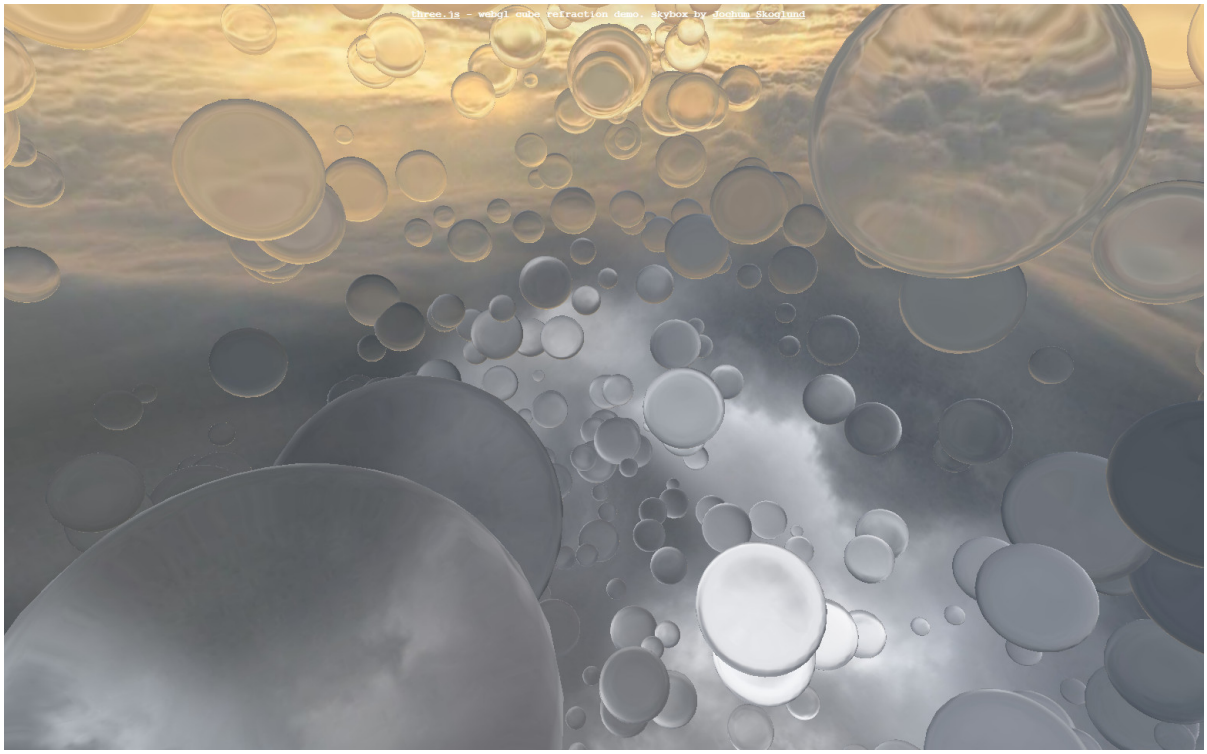
[show rest of lens bending. NOTE: Draw the backside so that it's at more of an angle to the rays

- the one below is actually most dead-on, perpendicular, which wouldn't change a thing! Just make it a wobbly line or something.]



However, real objects have at least two refraction events when light passes through them, both when entering and exiting the surface. Refraction mapping just does the one refraction.

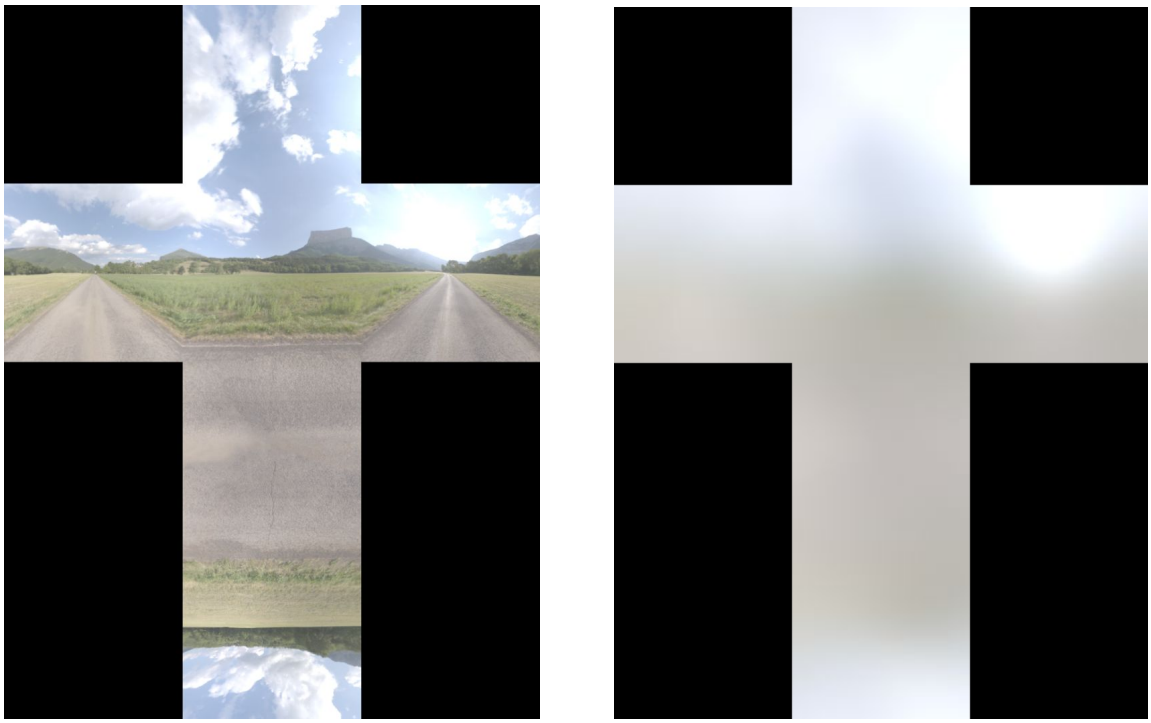
[http://mrdoob.github.com/three.js/examples/webgl_materials_cubemap_balls_refraction.html]



Refraction mapping is not quite as convincing as reflection mapping, in my opinion. Part of the reason is that the distortions produced are not as complex as the real thing. More importantly, all that you can ever see through these transparent objects is the environment map, nothing else. Notice with this spheres demo, lovely as it is, that you can't see any spheres through any other spheres, at least not without a lot of extra work.

[
Additional Course Materials:
Three.js provides a number of refraction mapping demos:
[spheres](http://mrdoob.github.com/three.js/examples/webgl_materials_cubemap_balls_refraction.html),
[angels](http://mrdoob.github.com/three.js/examples/webgl_materials_cubemap_refraction.htm), and [heads](http://mrdoob.github.com/three.js/examples/webgl_materials_cubemap.html).
]

Lesson: Glossy Reflection



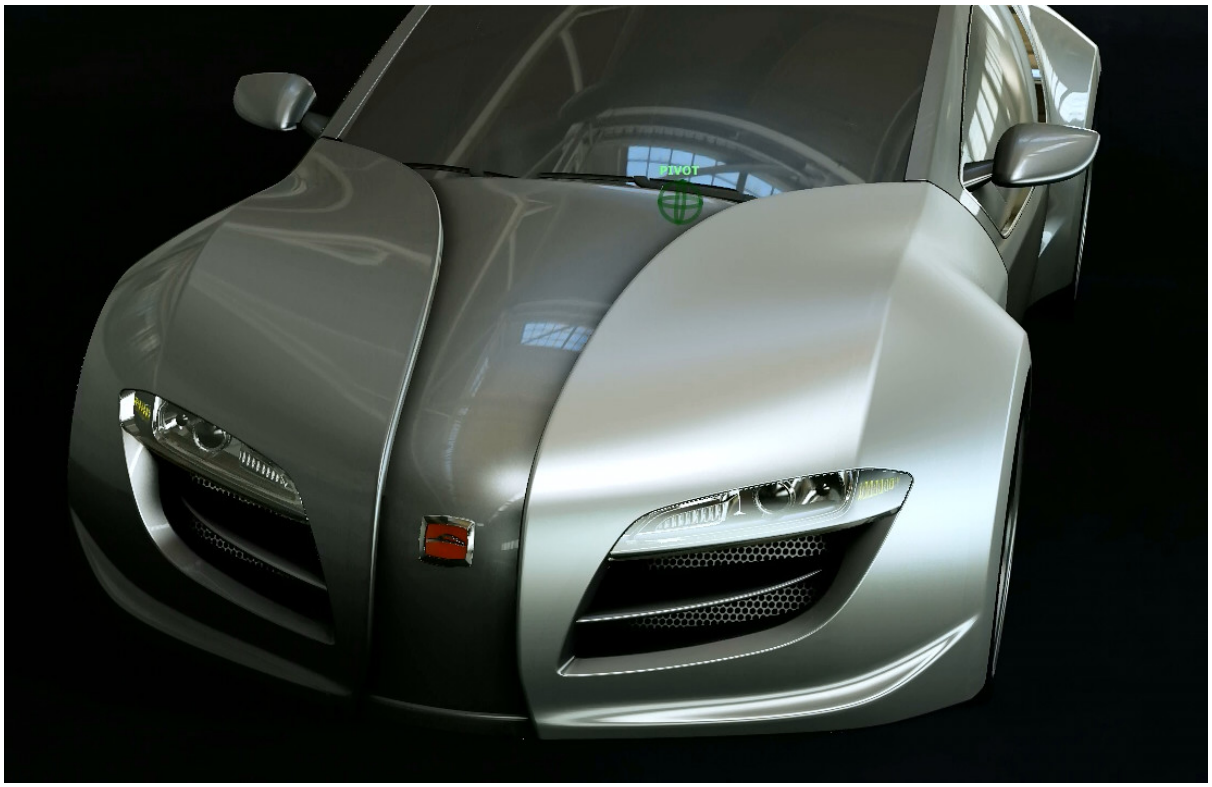
With reflection mapping we get sharp reflections. This is good as far as it goes, but many reflective surfaces are "glossy": They are shiny, but they are not mirror-like. Reflections are a bit blurred. One way to simulate this effect is to blur the environment cube map. This blurring is

usually does in advance and the texture stored. Some cleverness is needed to properly filter from face to face of the cube map, but utilities exist that perform this function for you.

[Play videos of various glossy cubemaps in action:

[What I want is to start each clip about 10 seconds in, after the selection and change has occurred. Basically, these 6 clips can fill the time I talk.

DarkVeryGlossyShowcase.wmv - run this whole clip, showing the part change
GrayShinyShowcase.wmv - from 0:09 on
GlossyShowcase.wmv - from 0:10 on
WhiteEnamelShowcase.wmv from 0:09 on
WhiteVeryGlossShowcase.wmv from 0:15 to 0:25 (after that there is a jump)
ChromeShowcase.wmv - from 0:07 on
]



[Really, I'm talking on and on here a bit just so the videos can run.]

Think about the Blinn-Phong specular reflection model for a moment. Blinn-Phong and other reflection models work by taking the light's direction and deciding at each point on a surface how much that point on the surface is likely to be a perfect mirror. Whatever portion is a mirror reflects this light. In reality, much of the world most of the time is sending photons to any given

point in space from many different directions. Look around, and everything you can see is sending light your way. It doesn't matter if the light is from an emitter or a surface (except that emitters are usually brighter). What an environment map does is treats the whole surrounding world as a source of light.

While blurring an environment map is not entirely physically correct, it is extremely convincing. There is less of a problem of a reflection not matching up with what is in the environment, since you can't see a clear reflection on a burnished surface. Another advantage is that these blurry cube maps can be considerably smaller in size and still look good. For the two cube maps I showed, the original had faces that were 512 by 512 texels in size, the blurred version only 128 by 128. A lower resolution is all that is needed since the colors change more slowly.

If you want a wider range of materials, you can blur the original reflection map by different amounts and get different gloss textures. You can also mix and match between sharp and blurred reflections on the same surface, giving an impression of a multicoat material. Combining these ideas with different reflectivities, illumination models, and surface colors can give a wide range of effects.

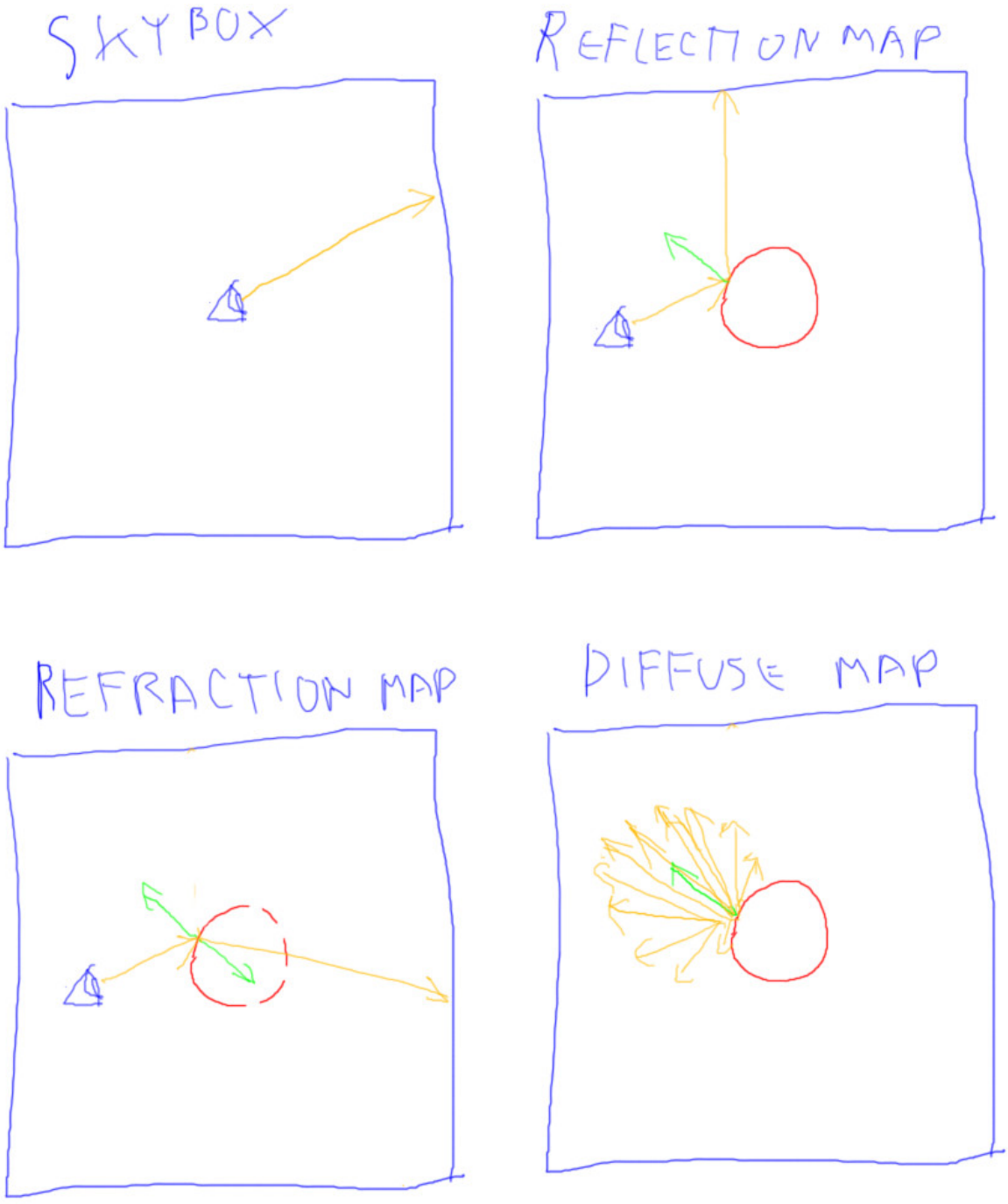
[Additional Course Materials:
A [free utility to blur cube maps](http://developer.amd.com/resources/archive/archived-tools/gpu-tools-archive/cubemapgen/) is provided by AMD. The site has a good runthrough of the effects achievable.

The car videos were made using [Showcase](http://www.autodesk.com/products/showcase/overview), which is a pretty fun program to toy with for styles and materials. Beyond the free trial, as a Udacity student you can [register](http://students.autodesk.com/?nd=register&tagent=EDU-FY12_Launch_EC_Banner-JG-5-19-2011) for [free use of Autodesk software](http://students.autodesk.com/?nd=download_center). When you register, note that Udacity is considered to be in Sunnyvale, CA for purposes of registration.
]

Lesson: Diffuse Environment Mapping

[show skybox, reflection, glossy reflection, refraction. Each of these should be a separate drawing, sharing the same box, and put one in place of the other.. Now add diffuse, no eye needed. NOTE: easiest to draw a lot of yellow samplers and then erase into a sinusoidal thing, then add arrowheads.]

[LABEL AS ***DIFFUSE ENVIRONMENT MAP***.]



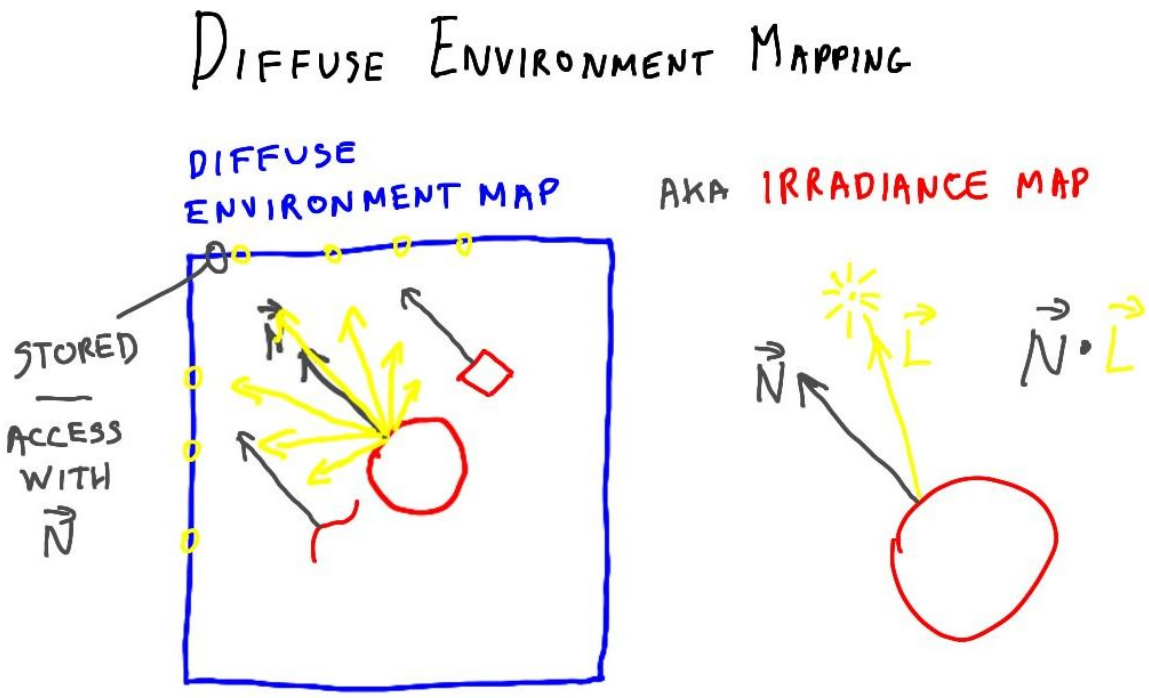
We've used cube maps for skyboxes and environment maps for sharp and glossy reflections and for refraction. It turns out we can even use this same mechanism for diffuse lighting. For these previous techniques we needed the direction to the eye and the shading normal in order to compute a reflection or refraction ray. For the diffuse component we just need the shading normal. Remember, the diffuse component doesn't depend on the eye's direction.

We used the dot product between the normal and light source vectors when we computed the

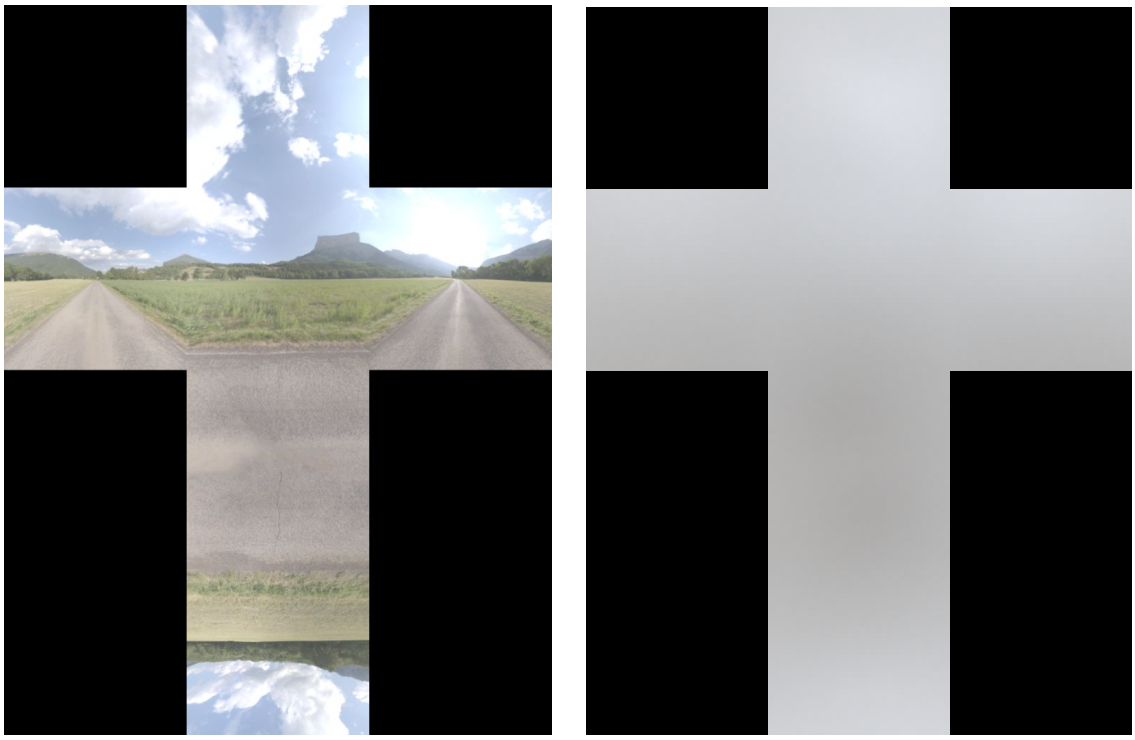
effect of a single light on a surface using Lambert's Law. Now think big: for a particular point on a surface, consider every texel in our environment map as a source of light, emitted or reflected. Take the dot product of the direction vector to each texel from the surface and compute the dot product with the surface normal. Add these all up. You now have the contribution of the entire environment map on a single location.

You might think this is a bit expensive to do for every pixel we want to illuminate, and you'd be right - doing this properly is a good task for path tracing. However, there are a few things working in our favor. First, once you've computed the sum of the contributions of all the texels for one point on one surface, you've actually computed it for *all* points with the same normal direction. Remember that the environment map is infinitely far away, so what's true for one surface normal is true for them all!

[LABEL AS ***DIFFUSE ENVIRONMENT MAP***. Show storing the direction's data on cube map, and add "***irradiance map***" term.]



The other element working in our favor is that a cube map can be used to store the results of our computations. After we compute the sum of all the contributions for a given normal, we then simply store this value directly in that location of a cube map texture. This texture is called the diffuse environment map or irradiance map. Now when we want the sum total lighting for the surrounding environment for a given normal, we just look it up in the cube map.

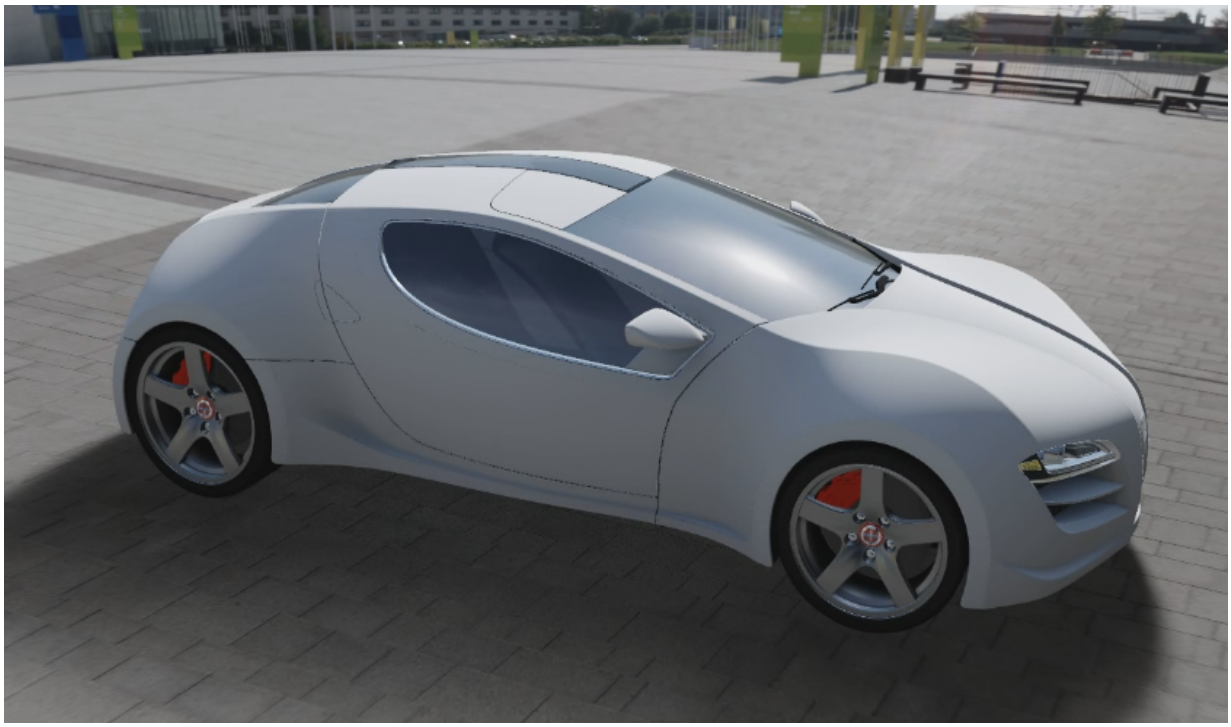


One other advantage we have is that the diffuse map can be quite low resolution. While this original sharp environment map texture had cube faces that were 512 by 512 texels in size, and our gloss map 128 by 128, this diffuse map is usable at only 32 by 32 texels in size.

Nonetheless, it can be expensive to compute the diffuse map, so what we do is the same as for the glossy map: we compute the texture in advance with some image processing software. This gives us the texture we need to light the scene. During rendering we just use the shading normal to find the overall contribution of the environment's lighting to a surface fragment.

[Dutkay videos, Unit7_DiffuseMaps, show directional 1_Technicon_Directional.mp4, then 2_Technicon_IBL.mp4 diffuse only, then 4_Technicon_Directional+IBL+Env.mp4 final result. Video editor: please start each clip about 8-9 seconds in on each clip, showing the car from the front. Basically, I want the three clips to tie to the three paragraphs below, then just let the last clip loop..]

Lesson: Diffuse Mapping Example



Here's a car painted with diffuse paint - something you'd probably never see in the real world - with just a directional light on, representing the sun. Notice how the area in shadow is entirely dark. Filling this in with a solid ambient light wouldn't look much better.

Here the car is shown with just a diffuse map that captures the surrounding environment. The surface has a much more realistic look, even without the directional light. A diffuse map captures the effect of all the incoming distant light.

The final result, showing the surrounding environment and shadows, gives a highly realistic effect. This demo runs at interactive rates.

Looking at demos like this one gets me excited to add another ten lessons on various topics that have to do with texturing. For example, I'm feeling a bit guilty that I haven't talked about high dynamic range textures, HDR for short. The quick version about HDR textures is that they're used to capture a wide range of lighting levels, basically more bits per channel. This wide range is particularly useful for environment maps, as the light sources in a scene can in reality be magnitudes brighter than other objects. These textures allow a variety of materials to use them in a more physically believable fashion, and also avoid banding artifacts due to precision limits. For example, some objects such as plastics mostly pick up just bright lights as reflections and little else. HDR textures allow us to use the same environment map for these materials as well as highly reflective metallic surfaces.

[asked Natasha for a light probe image to show here - nothing great came of it.]

I've barely scratched the surface of what's possible. To give an example, there are even more compact representations than diffuse maps for storing the environment's effect on Lambertian materials. A representation called spherical harmonics can be used to store this information in an array of 9 colors.

One limitation of using a single diffuse map is that the lighting is the same everywhere in a world. An advanced technique is to create what are called light probes. Each light probe stores the illumination in all directions for a single point in space. As an object moves through space, the light probes nearest to it are combined and used to compute its diffuse lighting. This approach involves a considerable amount of precomputation, but gives realistic, changing lighting at interactive rates.

[Additional Course Materials:
There's more about diffuse maps, aka irradiance maps, [at this site](http://codeflow.org/entries/2011/apr/18/advanced-webgl-part-3-irradiance-environment-map/), along with a demo.

There are ways to extend to irradiance map idea to vary with location, see [this seminal paper](http://www.sci.utah.edu/~bigler/images/msthesis/The%20irradiance%20volume.pdf) from Gene Greger et al., and [this recent work in "Far Cry 3"](http://fileadmin.cs.lth.se/cs/Education/EDAN35/lectures/L10b-Nikolay_DRTV.pdf) for just one application of it. Better yet, [there's a demo in three.js](http://codeflow.org/entries/2012/aug/25/webgl-deferred-irradiance-volumes/) along with a full explanation.
]

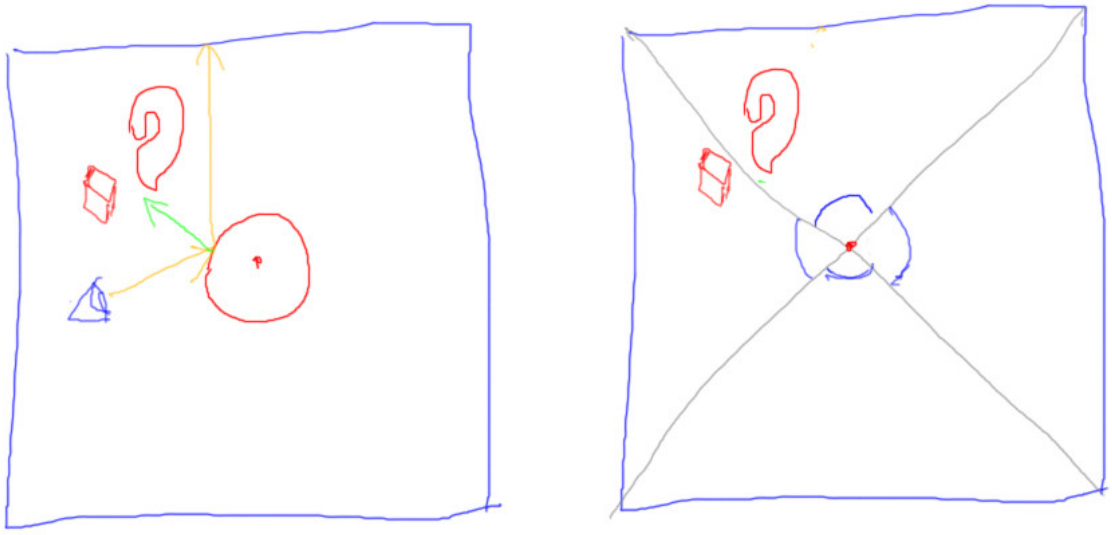
Lesson: On the Fly Cube Maps

[http://mrdoob.github.com/three.js/examples/webgl_materials_cubemap_dynamic2.html - note how no one can figure out if knot or cube textures are bad]

Here’s a very nice demo showing an important technique for cube maps. Up to this point the techniques described have used cube maps created in advance. In fact, the GPU is plenty powerful enough to make cube maps on the fly.

Look at this demo carefully. What’s great about it is that the shiny knot figure and block are clearly reflected in the sphere, changing position every frame.

[show how it’s done]



The way this is done is to make a cube map from the viewpoint of the sphere's position. The sphere's center and each face of the cube map forms a view frustum. You render the scene six times, one for each cube face, and each rendering is then used for the new cube face.

The sphere itself is not rendered into the cube map. After the cube map is rendered, it is immediately applied to all the objects in the scene. The cube map includes the knot and cube in it, so you see these reflected in the sphere. The exact reflection is not precisely right, but close enough that we accept it as correct. For efficiency, this same map is used on the knot and block, but the reflections are so distorted that we don't notice that the environment map is incorrect, that it includes two objects and not the third.

[go back to demo]

Notice how this demo is recursive in nature: the objects reflected are also reflective. Each object when rendered is using the previous frame's environment map on it, so they look reflective. I recommend toying with this demo and changing objects and materials to see how it changes what you see.

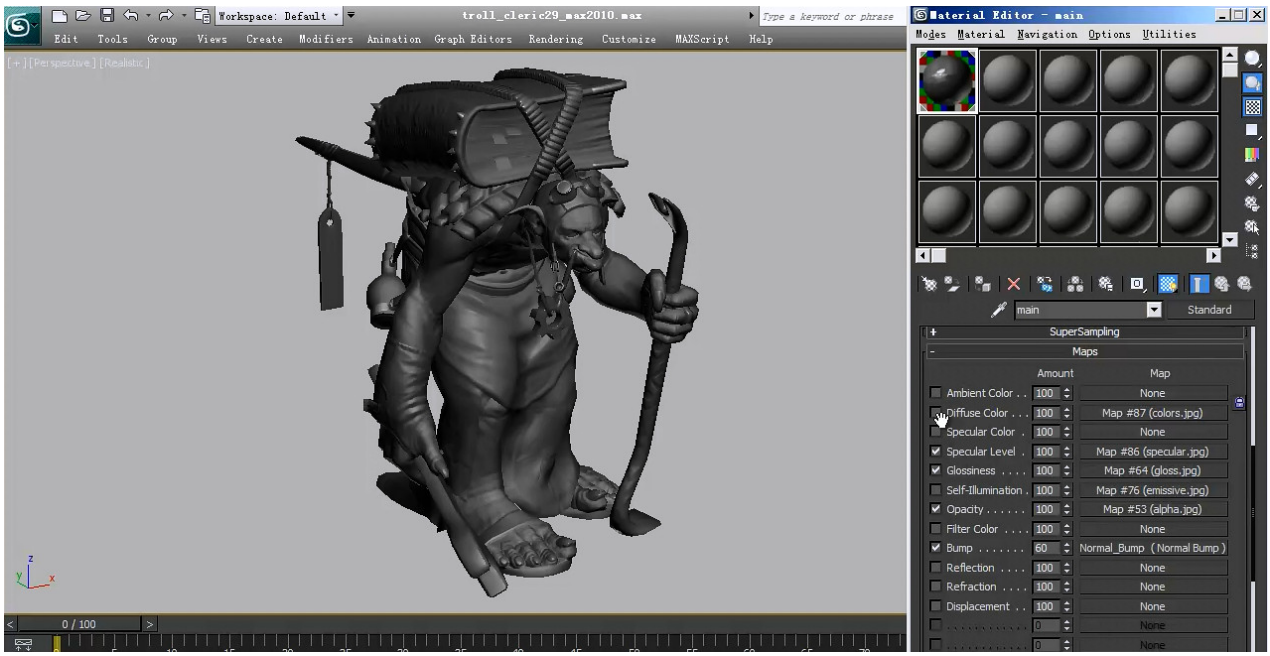
This technique of generating cube maps on the fly is popular in racing games. The car is removed from the scene and the rest of the environment is rendered to a cube map, which is then applied to the car.

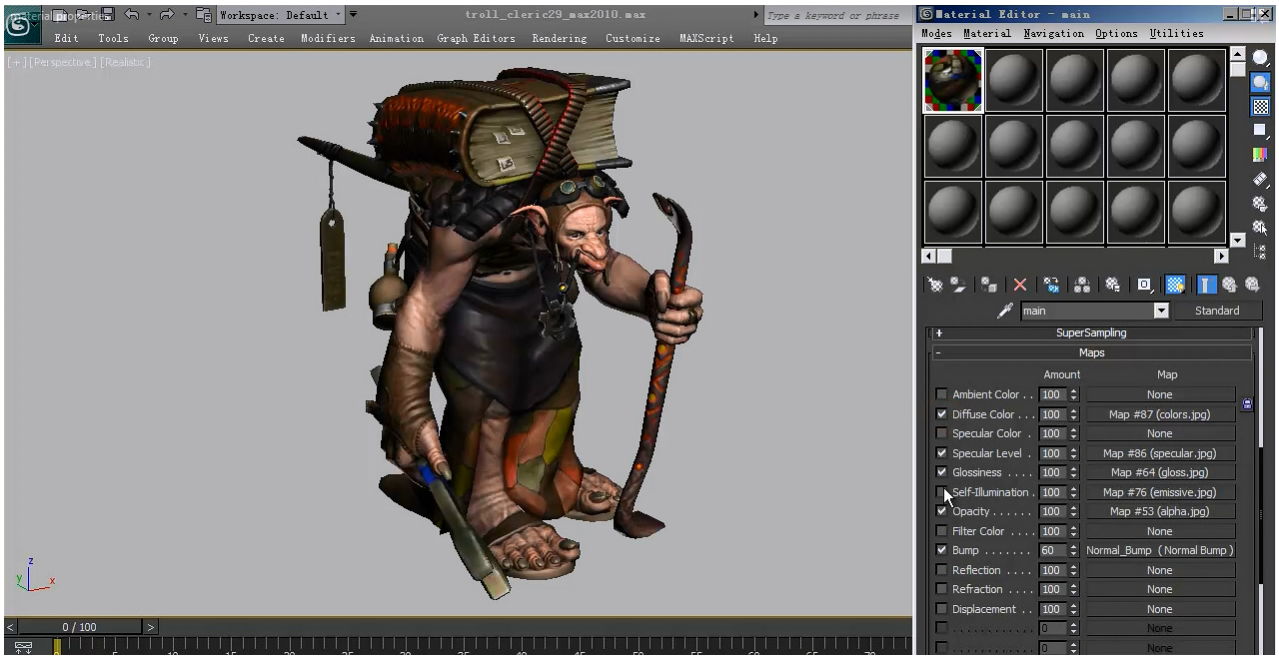
[Additional Course Materials:
The demo shown is
[here](http://mrdoob.github.com/three.js/examples/webgl_materials_cubemap_dynamic2.html).

To see how to make a recursive video screen, see Lee Stemkoski's [commented demo code](http://stemkoski.github.com/Three.js/Camera-Texture.html).
]

Lesson: Conclusion

[Start with short video clip Unit7_Conclusion, material properties.mp4, 0:25 to 0:29, toggling texture on and off]





We’ve seen how adding just simple color textures can vastly improve the look of a model.

[http://mrdoob.github.com/three.js/examples/misc_controls_fly.html - this has the most annoying UI ever. Hit W to zoom in, then Don’t touch anything.

Better is http://alteredqualia.com/glearth/
]

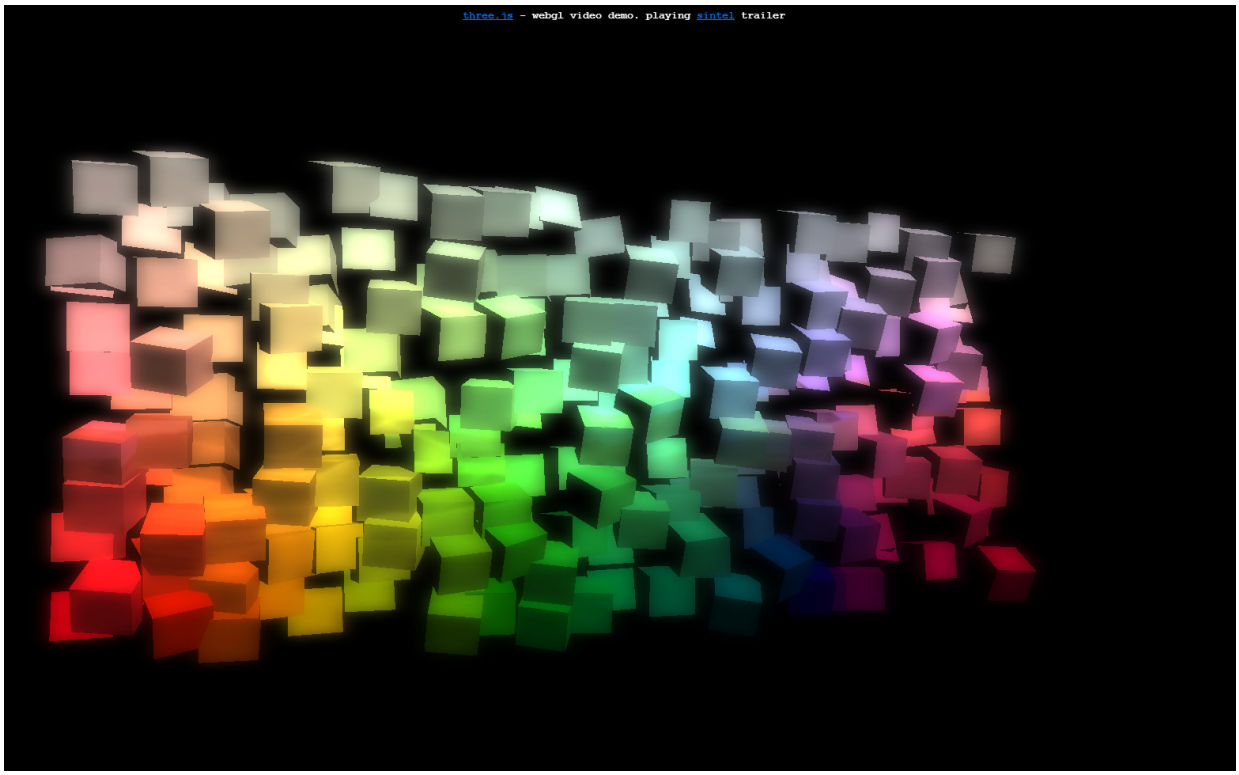
GPUs are very much optimized to access textures, having considerable numbers of transistors dedicated to just this task. This in turn has lead to a huge number of uses for textures.

The idea of applying two or more textures has any number of other uses. For example, a surface can be made visually rich by giving it textures that add scratches, decals, planetary cloud cover, or any number of other phenomena. Multiple textures can be combined by addition, multiplication, or other operations and all affect the same attribute.

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

Mirror reflection can be done by rendering from a different angle and then using the reflected image as a texture used at each pixel. This is a whole huge subcategory, where a scene is rendered and then its resulting image is post processed in various ways. We'll talk a bit more about these when we get into shader programming.

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



Because three.js is a web component, it can interact in interesting ways with other data. For example, you can take a video and use it as a texture. This texture can then be placed on multiple moving screens, as shown in this demo.

Now that you know the basics of texture mapping, you're equipped to go ahead and dig into other people's demos and see what makes them tick. Better yet, you can now try out some of your own effects. Texturing is a powerful tool, and I look forward to seeing what you do with it.

[Additional Course Materials:
The [video display demo shown](http://mrdoob.github.com/three.js/examples/webgl_materials_video.html) is distributed with three.js. A demo by Lee Stemkoski has commented code to [display a video on a surface](http://stemkoski.github.com/Three.js/Video.html). You can also see how to capture webcam data in [this](http://stemkoski.github.com/Three.js/Webcam-Test.html) and [this](http://stemkoski.github.com/Three.js/Webcam-Texture.html) demo of his, and how to warp camera views [here](http://stemkoski.github.io/Three.js/Many-Cameras.html).
]

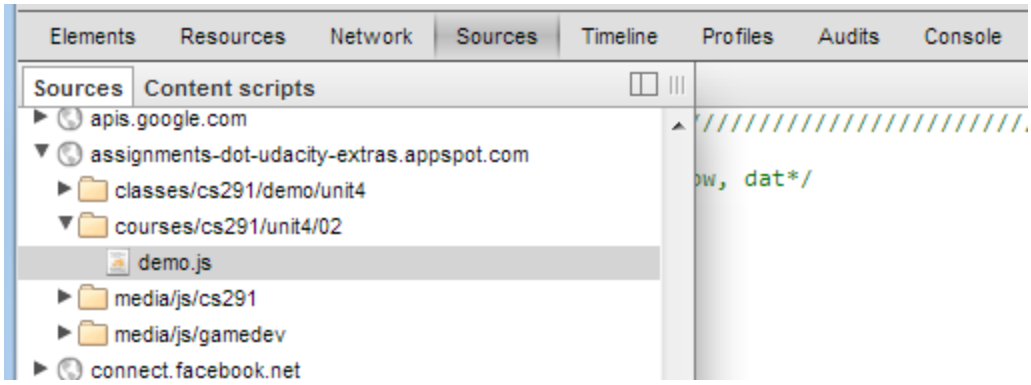
[end recording 3/30]

Problem Set

[begin: recorded 4/3]

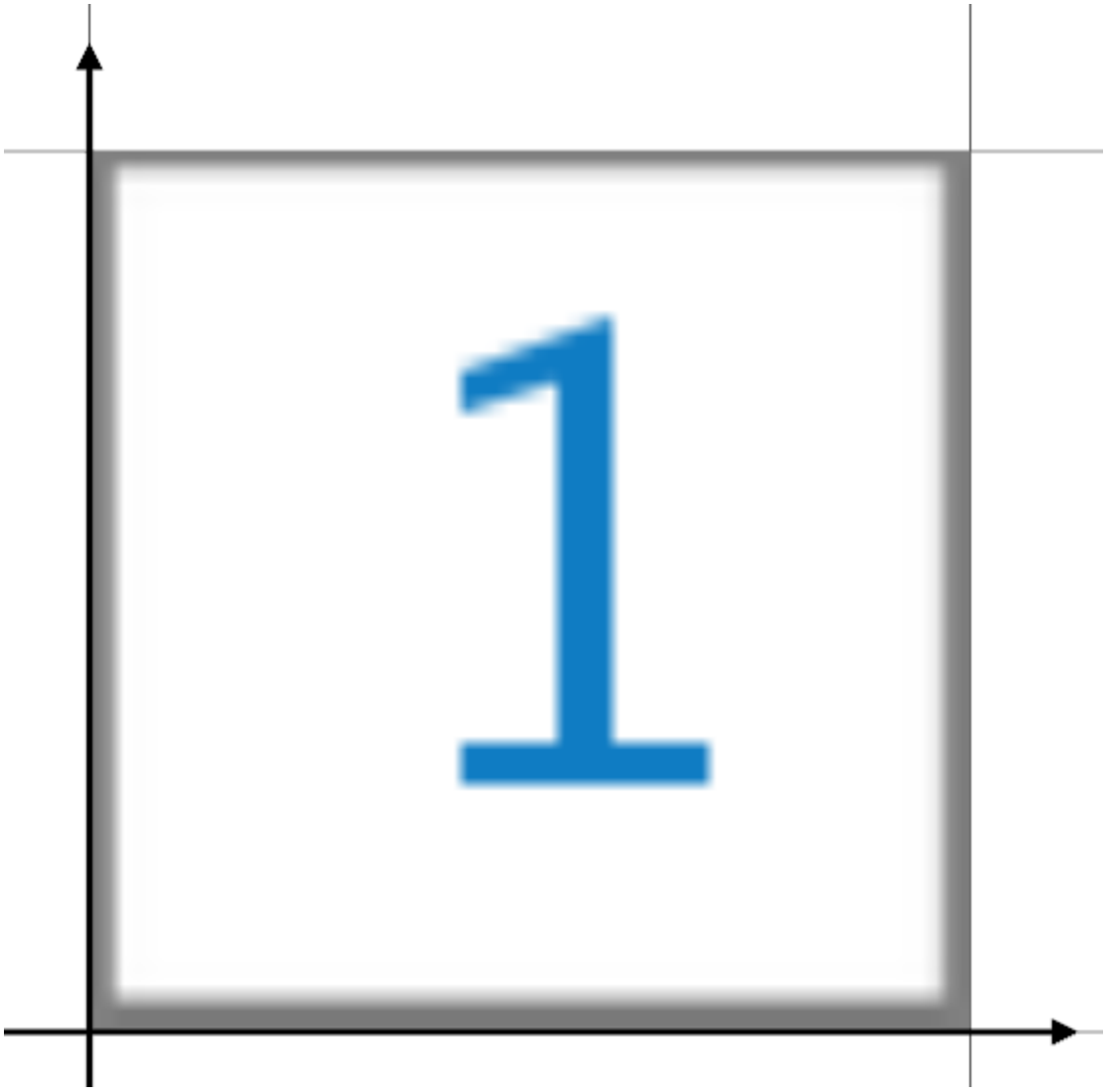
Problem: Pick a Letter

Modify the existing code so that you display just the numeral 1 from the grid. You should pick out a piece of the texture that is a quarter of the width and height of the square.



By the way, here's a tip: if you want to look at demo code, you can download it from github, or can examine it right in your browser if you dig a little bit with the browser's debugger. For example, here's where I found the Euler angle demo in a previous unit, using Chrome's debugger.

[SHOW THIS ACTUAL IMAGE CAPTURE, not program]



To get back to the exercise, when you're done the view on the square should look like this.

[exercise is unit8-ps_letter_exercise.js]

Answer

[unit8-ps_letter_solution.js]

```
var uvs = [];
uvs.push( new THREE.Vector2( 0.75, 0.25 ) );
```

```
uvs.push( new THREE.Vector2( 1.0, 0.25 ) );
uvs.push( new THREE.Vector2( 1.0, 0.5 ) );
uvs.push( new THREE.Vector2( 0.75, 0.5 ) );
```

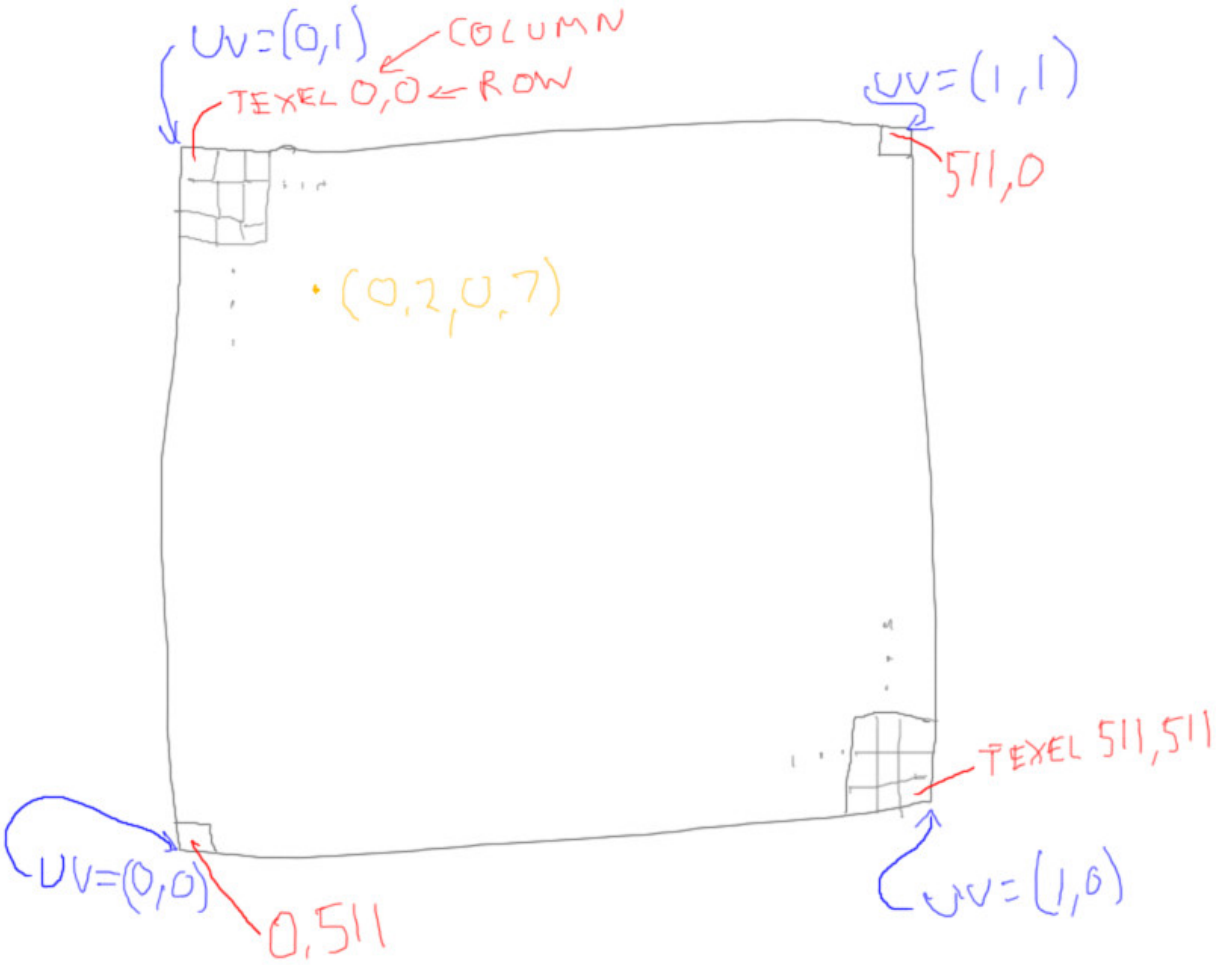
```
var uvs = [];
uvs.push( new THREE.Vector2( 0.75, 0.25 ) );
uvs.push( new THREE.Vector2( 1.0, 0.25 ) );
uvs.push( new THREE.Vector2( 1.0, 0.5 ) );
uvs.push( new THREE.Vector2( 0.75, 0.5 ) );
```

[draw lines from coordinates to points on square for numeral “1”.]

The main thing here is to realize you need to change the UVs to select the numeral. Here’s the code that I modified in the SquareGeometry method.

Problem: Y-Flipped Texture Coordinates

[pixel image and show 0,0 in upper left; 512 columns, 512 rows. Leave room to right to show answer.]



What texel corresponds to U,V coordinates 0.2,0.7?

column _____, row _____

You will sometimes run into textures being flipped top to bottom, with the Y axis reversed. In three.js you can correct this by setting the flipY parameter on your texture:

```
texture.flipY = true;
```

```
texture.flipY = true;
```

The texture coordinates are a bit trickier when the texture is flipped. Say our image is 512 x 512 pixels in size, with the upper left corner of this image being 0,0. This is different than how UVs normally get defined, using the lower left corner as 0.0,0.0.

Also recall that each of the UV coordinates point at the outer corner of the image, not the texel center.

The question to you:

What texel corresponds to U,V coordinates 0.2,0.7?

What I want here are two integers, not floating point numbers. Each texel has a location in an array, I want this location.

Answer

U: 0.2 * 512 = 102.4, drop the fraction to get 102.

V: 0.7 * 512 = 358.4, drop the fraction to get 358.
lower-left origin row 0 would be upper-left origin 511 [draw numbers on figure]
lower-left origin row 511 would be upper-left origin 0
So lower-left origin row R would be upper-left origin row (511-R)
511-358 = 153

The answer for which column is relatively straightforward: U goes from 0 to 1 and there are 512 columns, starting at 0, so I multiply the U of 0.2 by 512 and get 102.4. I drop the fraction to get the texel column. As you will recall from the earlier exercise, we drop the fraction.

For the row, if I were to make the lower left corner to be texel 0,0, then we would simply multiply again and be done. Row 358 would be the answer.

However, counting from the upper left corner is what we want. Think about the relationship: the lower corner's row 0 is the upper corner's 511, and vice versa. To convert from one to the other we subtract the row from 511. For example, row number 1 at the bottom is indeed row number 510 when counting from the top.

Subtracting 358 from 511 gives us row 153.

Problem: Grassy Plain

[show this unit8-ps_db_ground_solution.js]

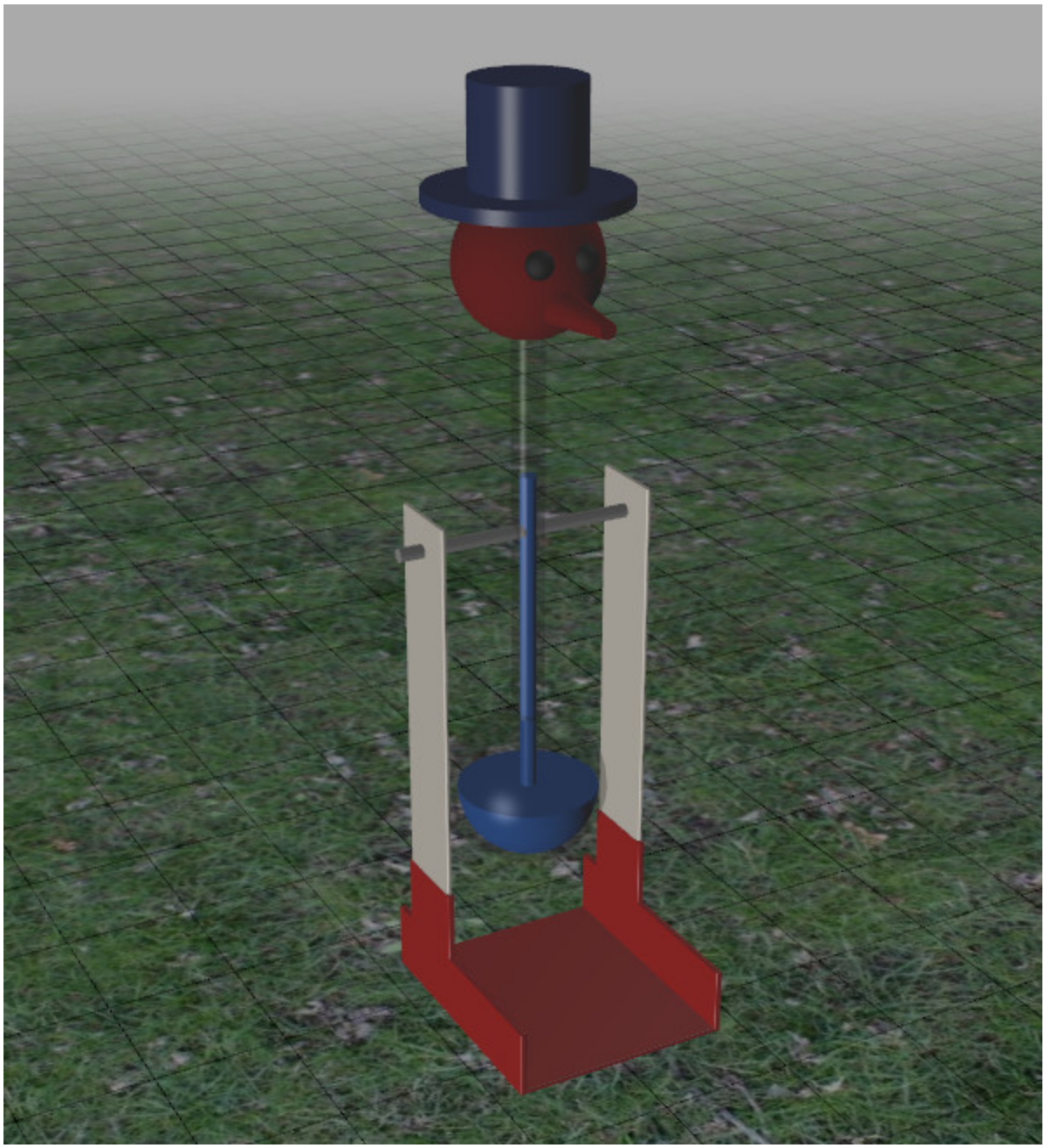
The drinking bird could use a more pleasant outdoor environment. Your job is to add a grass texture to the ground plain beneath the bird. Just apply a simple color texture. Since the plain is

large, please repeat the texture 10 times in each direction.

See the exercise's code for more details. When you're done, the view should look like this.

One hint: don't forget about setting the wrap mode. I didn't do this at first and it was extremely confusing as to why the texture seemed to disappear.

[Additional Course Materials:
The grass texture is a lower-resolution version from [Humus' site](http://www.humus.name/index.php?page=Textures), which has many freely-reusable reflection cube maps and other textures.
]



Answer

```
var texture = THREE.ImageUtils.loadTexture( 'textures/grass512x512.jpg' );
texture.wrapS = texture.wrapT = THREE.RepeatWrapping;
texture.repeat.set( 10, 10 );
var solidGround = new THREE.Mesh(
        new THREE.PlaneGeometry( 10000, 10000, 100, 100 ),
        new THREE.MeshLambertMaterial( { map: texture } ) );
```

```
var texture = THREE.ImageUtils.loadTexture(
    'textures/grass512x512.jpg' );
texture.wrapS = texture.wrapT = THREE.RepeatWrapping;
texture.repeat.set( 10, 10 );
var solidGround = new THREE.Mesh(
    new THREE.PlaneGeometry( 10000, 10000, 100, 100 ),
    new THREE.MeshLambertMaterial( { map: texture } ) );
```

The solution has a few steps: first the texture is loaded, then its wrap mode is set to repeat and the repeat factor itself set to 10 for each axis. The material's map is then set to the texture to display the grass.

Feel free to remove the gridlines, of course. If you do, you'll see that the bird does not feel very attached to the surface, since there's no shadow.

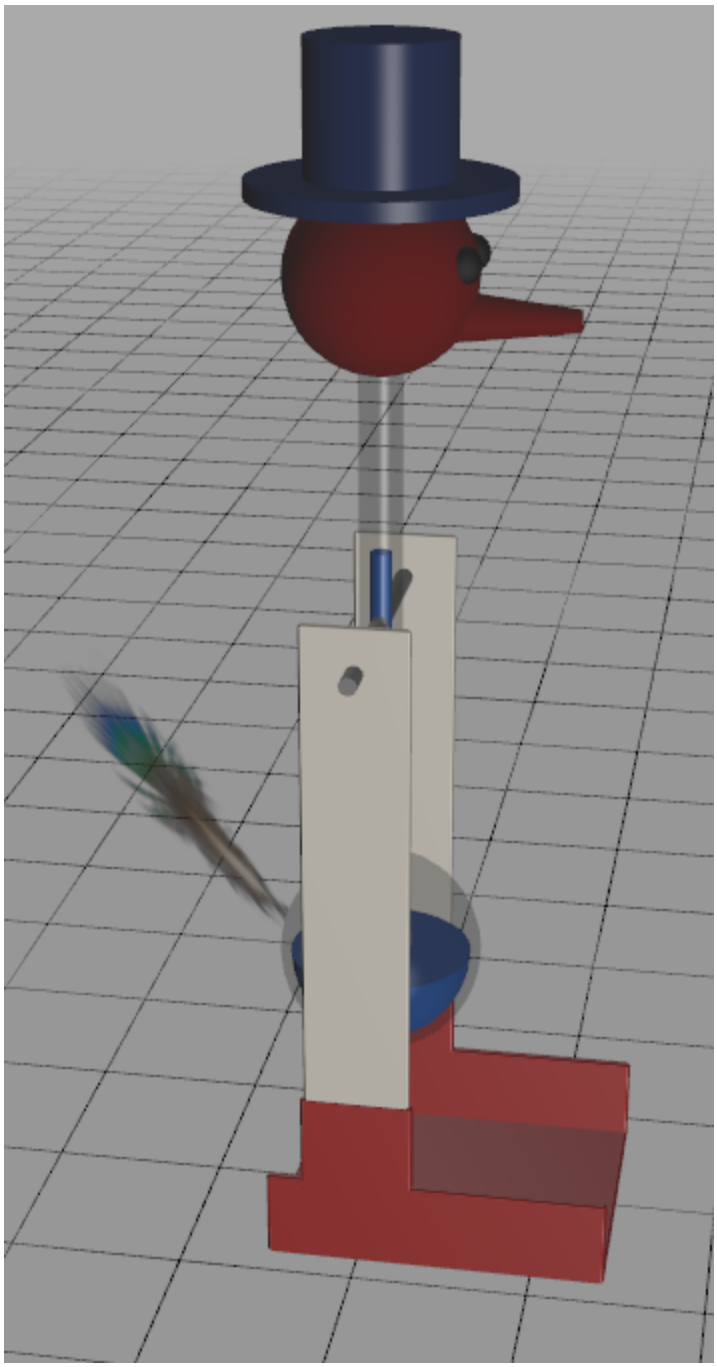
Problem: Drinking Bird Tail

[run the whole time unit8-ps_db_tail_solution.js]

The drinking bird most definitely needs a tail, and at long last we're going to give him a feather.

I've set up the tail polygon. This was a bit tricky, in fact: to avoid adding an Object3D(), I added some code to change the order the Euler angles are evaluated.

Your task is to add the semi-transparent feather texture to this tail. This is how it should look when you're done.



Answer

```
var tailTexture = THREE.ImageUtils.loadTexture( 'textures/feather.png' );
var tail = new THREE.Mesh(
        new THREE.PlaneGeometry( 100, 100, 1, 1 ),
```

```
        new THREE.MeshLambertMaterial(
            { map: tailTexture, side: THREE.DoubleSide, transparent: true } ) );
```

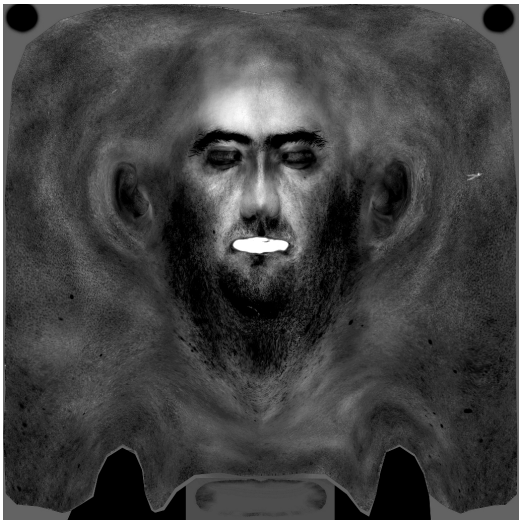
```
var tailTexture = THREE.ImageUtils.loadTexture(
    'textures/feather.png' );
var tail = new THREE.Mesh(
    new THREE.PlaneGeometry( 100, 100, 1, 1 ),
    new THREE.MeshLambertMaterial(
        { map: tailTexture, side: THREE.DoubleSide,
            transparent: true } ) );
```

A few bits of code are needed. First the tail texture is loaded. The material itself has its map texture set, and the transparent flag must be set to true in order to use the texture's alpha.

Problem: Specular Mapping

[http://mrdoob.github.com/three.js/examples/webgl_materials_normalmap2.html]

Just about anything can be changed with texture mapping. Beyond color and alpha, here's an example of changing the shininess for different locations on the body. The face is fairly shiny, the back of the head and shoulders less so.

Here are the color and specular textures for the face. In three.js the specular map uses the red channel of the texture as the strength of the specular contribution. A level of 255 means the specular component is scaled by one, in other words full strength, dropping off as this value falls.


Your task is to apply a texture as a specular map. See the exercise's code for the path to the texture. Look at the online documentation for how to set the specular map for the MeshPhongMaterial. On the left is the teapot without the specular map. Once you're done you'll see a teapot like the one on the right. As you change the view you'll see that just the specular highlights are textured.

[Additional Course Materials:
To play with this feature and other texture mapping methods discussed in this unit, try [the three.js online editor](http://mrdoob.github.com/three.js/editor/). Load a directional light, load an object, then apply textures to your heart's content.

As far as searching through the code base goes, I found the free program [Agent Ransack](http://www.mythicsoft.com/page.aspx?type=agentransack&page=home) invaluable on Windows for looking for various classes and their use. If you know of an even better program on Windows, or a similar program for the Mac or Linux, please post it on the forums..
]

Answer

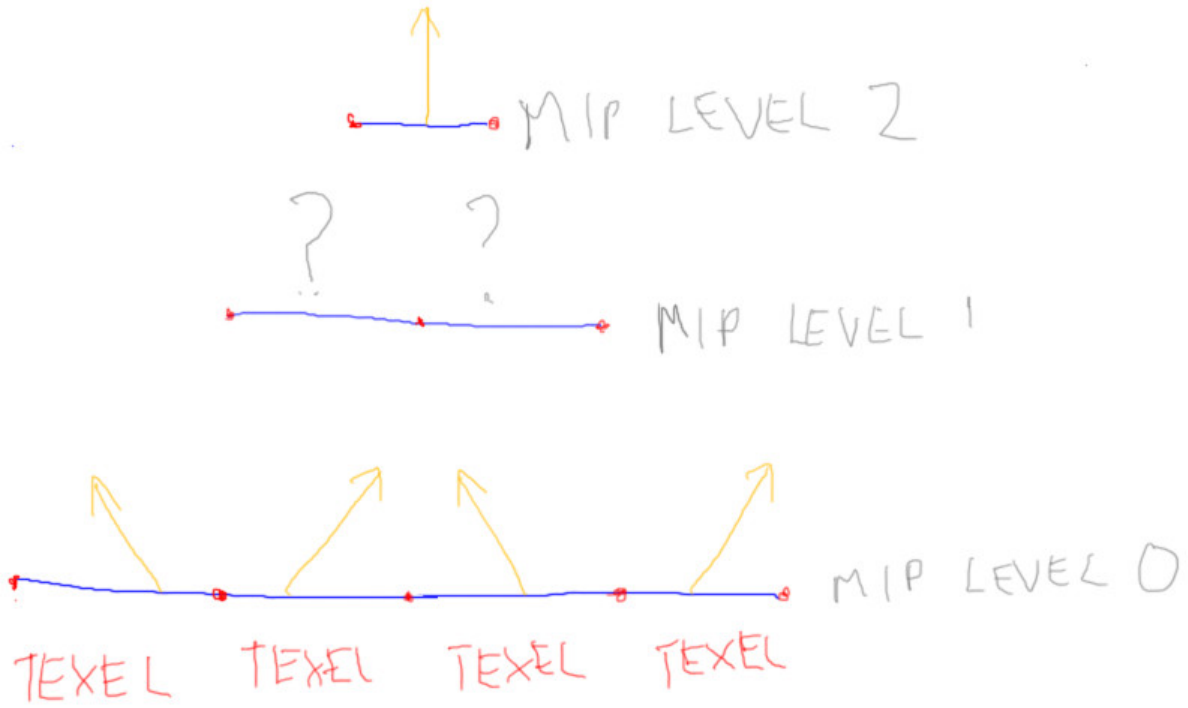
```
// MATERIALS
var texture = THREE.ImageUtils.loadTexture( 'textures/water.jpg' );
var material = new THREE.MeshPhongMaterial( { shininess: 50 } );
material.specularMap = texture;
material.color.setHSL( 0.09, 0.46, 0.2 );
material.ambient.copy( material.color );
material.specular.setHSL( 0.09, 0.46, 1.0 );
```

```
// MATERIALS
var texture = THREE.ImageUtils.loadTexture(
    'textures/water.jpg' );
var material = new THREE.MeshPhongMaterial(
    { shininess: 50 } );
material.specularMap = texture;
material.color.setHSL( 0.09, 0.46, 0.2 );
material.ambient.copy( material.color );
material.specular.setHSL( 0.09, 0.46, 1.0 );
```

The two lines of code needed are one to create the texture, the other to set the material's specularMap to this texture. You might want to try changing "specularMap" to just "map", which sets the diffuse map instead. The effect is considerably different.

Problem: Bump Map Filtering

[be careful to draw this precisely as possible]



For color textures we saw that mipmapping improved the quality of the images produced. Noisy patterns were blurred out. Normal maps can suffer the same sorts of problems, with noisy sparkles twinkling on and off as we happen to catch a normal that reflects light directly to our eye.

Say we use mipmapping on the normal map here to avoid this problem. These bumps describe two ridges. We average neighboring shading normals to get the next level up. The topmost mipmap will be a single normal that's the average of all the normals at the bottom level. So if the object is far enough away and this topmost level is used, it will not look bumpy at all.

What happens when the surface is far enough away that mip level 1 is used?

( ) The ridges disappear.
( ) The ridges become twice as high.
( ) The ridges become half as high.
( ) The ridges reverse their directions.

Answer

The answer is that mip level 1 will also have no real bumps left. The two normals in each case will cancel out and the surface will look flat. This is a problem with normal maps, that they disappear when filtering is done. There are approaches that have been developed to help

improve this situation, but mipmapping itself is often not done on normal maps because of this problem.

Problem: Reflection Mapping

You start with the shiny teapot on the left, surrounded by a skybox. Assign this skybox cube map in the code to be the environment map for the teapot. See the documentation about the teapot's material for help. When you assign the map, you'll get the result on the right.

Answer

```
var teapotMaterial = new THREE.MeshPhongMaterial(
        { color: 0x770000, specular:0xffaaaa,
        envMap: textureCube } );
```

```
var teapotMaterial = new THREE.MeshPhongMaterial(
    { color: 0x770000, specular:0xffaaaa,
    envMap: textureCube } );
```

Since the cube map is already loaded, all that's needed is to set the envMap parameter to this textureCube.

[Additional Course Materials:
By default the reflection map is multiplied by the computed surface. There are other ways to mix the two results using the "combine" parameter, such as mixing and addition. See [this demo's code](http://mrdoob.github.com/three.js/examples/webgl_materials_cars.html), for example.
]
[end: recorded 4/3