1 Texture Mapping

A texture is just an image, a bunch of pixels in a rectangular array. In fact, it is just like what we are trying to generate: a window full of colors. However, there are a lot of sources for images, cameras for example.

When an image is from a camera, we store it as an image file in a standard format like JPEG: mandrill.jpg. This format is compressed, meaning that we don't store three bytes per pixel. Instead we do something smarter.

Lossless compression does something smarter in a way that allows us to get back exactly the original pixel values. Lossy compression is even smarter: we get back something which humans can't tell from the original value.

For example, if you never read the textbook, then we could compress it by eliminating all the pages. As far as you were concerned, it would be exactly the same as before, only lighter to carry. Win win.

1.1 Textures in WebGL

The bytes in the image need to be loaded onto the graphics card. The process is similar to loading a buffer full of points, which we did before.

We need to tell OpenGL what filtering to use. You see, we won't be accessing pixels by integer location, but we will be using floating point. Worse yet, the x and y go from 0 to 1 no matter what the dimensions of the image are. So what does texture coordinates $(0.107, 0.203)$ mean in a 100 by 100 image? Does it mean pixel $(11, 20)$? But maybe we want something in between pixel $(10, 20)$ and $(11, 21)$ in shade?

This chooses the nearest $(11, 20)$:

```
gl.glTexParameterf(GL.GL_TEXTURE_2D, GL.GL_TEXTURE_MIN_FILTER, GL.GL_NEAREST);
gl.glTexParameterf(GL.GL_TEXTURE_2D, GL.GL_TEXTURE_MAG_FILTER, GL.GL_NEAREST);
```

MIN means for far away. MAG means for close up.

This does linear interpolation:

```
gl.glTexParameterf(GL.GL_TEXTURE_2D, GL.GL_TEXTURE_MIN_FILTER,
                   GL.GL_LINEAR_MIPMAP_LINEAR);
gl.glTexParameterf(GL.GL_TEXTURE_2D, GL.GL_TEXTURE_MAG_FILTER, GL.GL_LINEAR);
```

There is more to this picture (!), but let's do other things first and come back if we have time.

Another question is wrapping. If the texture coordates go outside the range 0 to 1, like $(-1.3, 2.4)$, what should happen? This means repeat the picture:

```
gl.glTexParameterf(GL.GL_TEXTURE_2D, GL.GL_TEXTURE_WRAP_S, GL.GL_REPEAT);
gl.glTexParameterf(GL.GL_TEXTURE_2D, GL.GL_TEXTURE_WRAP_T, GL.GL_REPEAT);
```

Anyway, we need to upload the actual bytes to the graphics card:

```
gl.glTexImage2D(GL.GL_TEXTURE_2D, 0, internalFormat, width, height, 0,
                format, type, pixelData);
```

pixelData is a buffer containing the bytes, just like we did for points.

Using the right incantation, we tell the fragement shader to bind the variable tex1 to the texture:

```
    gl.glActiveTexture(GL.GL_TEXTURE0);
    gl.glUniform1i(shader.getUniform(gl, "tex1"), 0);
    texture1.bind(gl);
```

In the fragment shader, we use the texture function to look up the color at some texture coordinates, which are floating point and range from 0.0 to 1.0 for the x and y of an texture.

```
precision mediump float;

uniform sampler2D tex1;
uniform sampler2D tex2;
attribute vec4 fTexCoords;
varying vec4 outColor;

void main()
{
  vec4 color1 = texture2D(tex1, fTexCoords.xy);
  vec4 color2 = texture2D(tex2, fTexCoords.xy);
  outColor = color1 * color2;
  //outColor = texture(tex2, vec2(0.3, 0.8));
}
```

What can we do with this power??

1.2 Pasting Textures to Faces

First, we can paint pictures on the "sides of buildings", so to speak. This is actually pretty popular.

For the sample program, we have a square spanning (-1,-1,0) to (1,1,0). We have to break it into two triangles as usual.

Unfortunately, a standard image file is indexed with (0,0) in the upper left, just like a window. OpenGL automatically scales the image so the opposite corner is (1,1), but that will make your typical image appear upside down unless we are careful. We might be able to take care of this using the magic of matrices.

Here is the mapping from point coordinates to texture coordinates:

- $(-1, -1, 0) \rightarrow (0, 1)$
- $(1, -1, 0) \rightarrow (1, 1)$
- $(-1, 1, 0) \rightarrow (0, 0)$
- $(1, 1, 0) \rightarrow (1, 0)$

So what we do is load the texture coordinates into a buffer, just like we did for color or normal vector. So we don't have to write more code, we can just set z to zero and use PV as usual.

In the vertex shader, we copy the texture coordinates and pass them to the frament shader:

```
precision mediump float;

attribute vec4 position;
attribute vec4 texCoords;
varying vec4 fTexCoords;

void main()
{
   gl_Position = position;
   fTexCoords = texCoords;
}
```

As above, the fragment shader simply looks up the color in the texture and uses it as the output color. In the example, it multiplies two colors.

What if we wanted to paint the picture on the sphere?

We could use the x and y of the sphere to generate the texture coordinates. Since range from -1 to 1, we can use GL_REPEAT.

So let's just use the x and y as the texture coordinates....

What else can we do?

1.3 Environment Map: Mirror

Let's make the sphere look like a MIRROR that is reflecting a picture. This is the idea of an ENVIRONMENT MAP.

So we have p, n, l, and v. We will only use l to do a ordinary specular reflection with a large alpha.

For environment mapping, r should be the reflection of the eye direction:

$$r = 2(n \cdot v)n - v;$$

We need to figure out if and where the ray $p + sr$ intersects a background picture.

How do we tell the fragment shader where we have hung the picture???

The picture is floating in the world. Its upper left corner is at o, which we turn into a translation matrix T. Its u_x, u_y, and u_z directions are loaded into R. u_y points down. There is scale matrix S which scales x by w and a y by h. (When we "hang" this picture, we not only choose its position and orientation but also its size.)

Before we hang the picture, it's a square with corners are $(0, 0, 0)$, $(1, 0, 0)$, $(1, 1, 0)$ and $(0, 1, 0)$. A picture coordinate $t = (x, y, z)$ maps to

$$T \cdot R \cdot S \cdot t$$

So picture2world is $T \cdot R \cdot S$. world2picture is the inverse–easily calculated.

p, n, and v are in model coordinates. So we pass the matrix

$$\text{model2picture} = \text{world2picture} \cdot \text{model2world}$$

to the fragment shader. (We don't have model2world, but you know how to calculate it.)

The ray $p + sr$ maps to $p' + sr'$ where

- $p' = \text{model2picture} \cdot p$
- $r' = \text{model2picture} \cdot r$

In picture coordinates, the picture is in the plane $z = 0$. So we must pick s so that $z = 0$.

$$p'_z + sr'_z = 0 \qquad \rightarrow \qquad s = -p'_z/r'_z$$

If $s < 0$ then it's the *opposite* of the ray that hits the picture, so forget it. Otherwise, the x and y coordinates of $p' + sr'$ are the picture coordinates we want. If they are in the range 0 to 1, then the ray hits the picture. So we look it up and display it.

1.4 Refraction

We can even make a diamond! Snell's law tells us a different r that goes *into* the object.

Under Snell's law, if v makes angle θ_1 with n, then r makes angle θ_2 with $-n$, where θ_1 and θ_2 are related by the velocities of light in the substances. So if v_1 is the velocity of light in air and v_2 is the velocity in diamond (which is 2.4 times slower), then

$$\sin(\theta_2)/\sin(\theta_1) = v_2/v_1 = 1/2.4.$$

You can show this using a little calculus and the principle that light follows the path that takes the least amount of time. Light travels farther in air because it can move quicker in air.

First, let's project v onto the surface. If u is the projection of v, we know

$$u = v - sn$$

for some s because the projection is straight downward in the direction of $-n$. On the other hand, if u lies in the surface, then it is perpendicular to n so,

$$u \cdot n = 0$$

Putting these together,

$$(v - sn) \cdot n = 0 \qquad \to \qquad v \cdot n - sn \cdot n = 0, \qquad \to \qquad s = v \cdot n$$

So

$$u = v - (v \cdot n)n$$

This should not be surprising. If you consider the triangle with angle θ, v is the hypothenuse, sn is the adjacent side (hence has length $\cos\theta$), and a vector parallel to u is the opposite side. This tells us that $|u| = \sin\theta$.

Let θ', v' and u' and $-s'n$ be the "refracted evil twins" of θ, v, u, and sn. We know that $|u'| = \sin\theta'$, so

$$\sin\theta' = \frac{1}{2.4}\sin\theta, \qquad \to \qquad |u'| = \frac{1}{2.4}|u|. \qquad \to \qquad u' = -\frac{1}{2.4}u.$$

We also know that

$$s' = \cos\theta' = \sqrt{1 - \sin^2\theta'} = \sqrt{1 - |u'|^2}.$$

Don't use pow to calculate $|u'|^2$!. Finally,

$$v' = u' + -s'n.$$