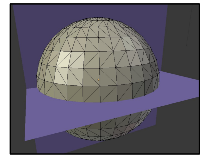
Texture Mapping

Oregon State University

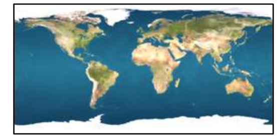
Mike Bailey

mjb@cs.oregonstate.edu

The Basic Idea: Wrap an Image Around a Piece of Geometry



+



=

In software, this is a very slow process. In hardware, this is very fast. The development of texture-mapping hardware was one of the most significant events in the history of computer graphics. This is really what finally enabled game development on a realistic scale.

The Basic Ideas

To prevent confusion, the texture image pixels are not called ***pixels***. A pixel is a dot in the final screen image. A dot in the texture image is called a ***texture element***, or ***texel***.

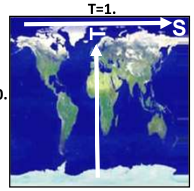
Similarly, to avoid terminology confusion, a texture image's width and height dimensions are not called *X* and *Y*. They are called **S** and **T**.

A texture image is not indexed by its actual resolution coordinates. Instead, it is indexed by a coordinate system that is resolution-independent. The left side is always **S=0**., the right side is **S=1**., the bottom is **T=0**., and the top is **T=1.**

Thus, you do not need to be aware of the texture's resolution when you are specifying coordinates that point into it. Think of S and T as a measure of what fraction of the way you are into the texture.

T=1.

S=0.



S=1.

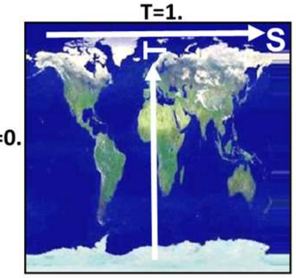
T=0.

The Basic Ideas

Texture mapping is a computer graphics operation in which a separate image, referred to as the **texture**, is stretched onto a piece of 3D geometry and follows it however it is transformed. This image is also known as a ***texture map***. This can be most any image.

T=1.

S=0.

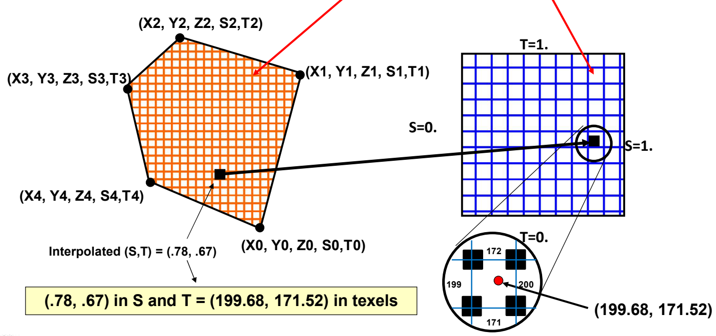


S=1.

T=0.

The Basic Ideas

The mapping between the geometry of the **3D object** and the S and T of the **texture image** works like this:



Oregon State University Computer Graphics

You specify an (s,t) pair at each vertex, along with the vertex coordinate. At the same time that OpenGL is interpolating the coordinates, colors, etc. inside the polygon, it is also interpolating the (s,t) coordinates. Then, when OpenGL goes to draw each pixel, it uses that pixel's interpolated (s,t) to lookup a color in the texture image.

5

Using a Texture: Assign an (s,t) to each vertex

Enable texture mapping:
glEnable(GL_TEXTURE_2D);

Draw your polygons, specifying **s** and **t** at each vertex:

```
glBegin( GL_TRIANGLES );
        glTexCoord2f( s0, t0 );
        glNormal3f( nx0, ny0, nz0 );
        glVertex3f( x0, y0, z0 );

        glTexCoord2f( s1, t1 );
        glNormal3f( nx1, ny1, nz1 );
        glVertex3f( x1, y1, z1 );

        . . .
glEnd( );
```

(If this geometry is static, i.e., will never change, ***it is a good idea to put this all into a display list***.)

Disable texture mapping:
glDisable(GL_TEXTURE_2D);

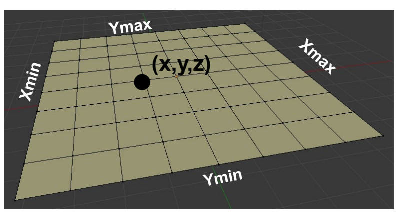
Oregon State University Computer Graphics

6

Using a Texture: How do you know what (s,t) to assign to each vertex?

glTexCoord2f(s0, t0);

The easiest way to figure out what s and t are at a particular vertex is to figure out what fraction across the object the vertex is living at. For a plane, this is pretty easy:



$$s = \frac{x - Xmin}{Xmax - Xmin} \qquad t = \frac{y - Ymin}{Ymax - Ymin}$$

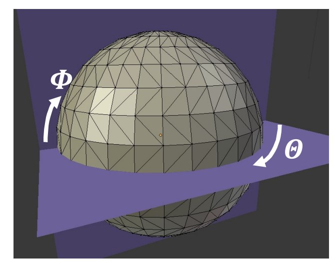
Oregon State University Computer Graphics

7

Using a Texture: How do you know what (s,t) to assign to each vertex?

glTexCoord2f(s0, t0);

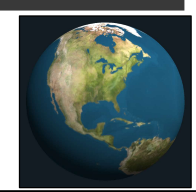
Or, for a sphere, you do the same thing you did for the plane, only the interpolated variables are angular (spherical) coordinates instead of linear coordinates



$$s = \frac{\Theta - (-\pi)}{2\pi} \qquad t = \frac{\Phi - (-\frac{\pi}{2})}{\pi}$$

The OsuSphere code does it like this:

```
s = ( lng + M_PI    ) / ( 2.*M_PI );
t  = ( lat + M_PI/2. ) / M_PI;
```



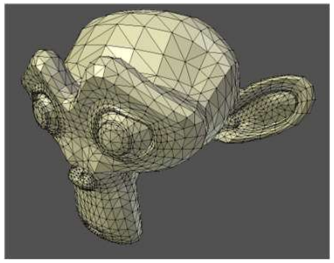
Oregon State University Computer Graphics

8

Using a Texture: How do you know what (s,t) to assign to each vertex? 9

glTexCoord2f(s0, t0);

Uh-oh. Now what? Here's where it gets tougher…,



$s = ?$ $t = ?$

Oregon State University Computer Graphics

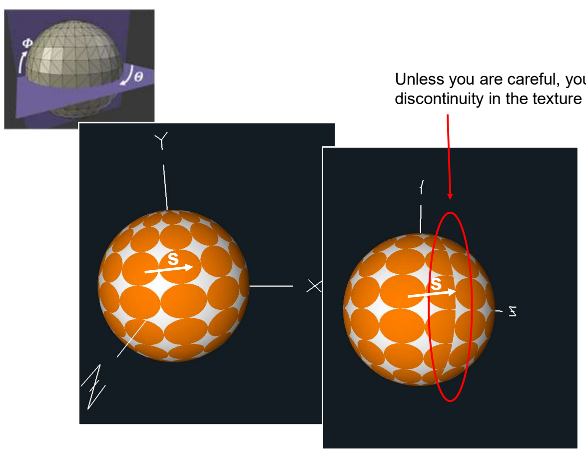
You really are at the mercy of whoever did the modeling and assigned the s,t coordinates… 10

Natural

Not-so natural

Oregon State University Computer Graphics

Be careful where *s* abruptly transitions from 1. back to 0. 11



Unless you are careful, you will see a discontinuity in the texture image

Oregon State University Computer Graphics

Reading a Texture from a BMP File 12

```
unsigned char *BmpToTexture( char *, int *, int * );
•••
int width, height;
unsigned char *texture = BmpToTexture( "filename.bmp", &width, &height );
```

This function is found in your sample code.

Note: BmpToTexture should be called *once*, and must be used at the end of *InitGraphics( )*.

Do not call BmpToTexture from the Display( ) function.
Do not call BmpToTexture from the Display( ) function.
Do not call BmpToTexture from the Display( ) function.
Do not call BmpToTexture from the Display( ) function.

Oregon State University Computer Graphics

Texture Wrapping

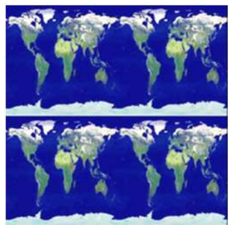
Define the texture wrapping parameters. This will control what happens when a texture coordinate is greater than 1.0 or less than 0.0:

glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, *wrap*);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, *wrap*);

where *wrap* is:

GL_REPEAT specifies that this pattern will repeat (i.e., wrap-around) if transformed texture coordinates less than 0.0 or greater than 1.0 are encountered.

GL_CLAMP specifies that the pattern will “stick” to the value at 0.0 or 1.0.



Oregon State University Computer Graphics

Texture Filtering

Define the texture filter parameters. This will control what happens when a texture is scaled up or down.

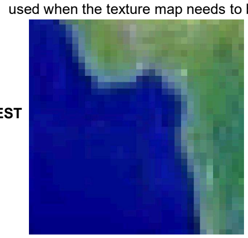
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, *filter*);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, *filter*);

where *filter* is:

GL_NEAREST specifies that point sampling is to be used when the texture map needs to be magnified or minified.

GL_LINEAR specifies that bilinear interpolation among the four nearest neighbors is to be used when the texture map needs to be magnified or minified.

GL_NEAREST



GL_LINEAR

Oregon State University Computer Graphics

Texture Environment

This tells OpenGL what to do with the texel colors when it gets them:

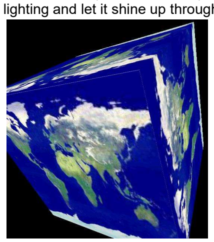
glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, mode);

There are several *modes* that can be used. Two of the most useful are:

GL_REPLACE specifies that the 3-component texture will be applied as an opaque image on top of the polygon, replacing the polygon’s specified color.

GL_MODULATE specifies that the 3-component texture will be applied as piece of colored plastic on top of the polygon. The polygon’s specified color “shines” through the plastic texture. This is very useful for applying lighting to textures: paint the polygon white with lighting and let it shine up through a texture.

GL_REPLACE



GL_MODULATE

Oregon State University Computer Graphics

Setting up the Texture in InitGraphics( )


```
int width, height;
unsigned char *texture = BmpToTexture( "filename.bmp", &width, &height );
int level=0, ncomps=3, border=0;
glPixelStorei( GL_UNPACK_ALIGNMENT, 1 );
glTexImage2D( GL_TEXTURE_2D, level, ncomps, width, height, border, GL_RGB, GL_UNSIGNED_BYTE, texture );
```

where:

level is used with mip-mapping. Use 0

ncomps number of components in this texture: 3 if using RGB, 4 if using RGBA. Use 3

width width of this texture map, in texels.

height height of this texture map, in texels.

border width of the texture border, in texels. Use 0

texture the name of an array of unsigned characters holding the texel colors.

This function physically **transfers** the array of texels from the CPU to the GPU and makes it the currently-active texture. You can get away with specifying this ahead of time only if you are using a ***single texture***. If you are using multiple textures, you must make each texture current in **Display( )** right before you need it. See the upcoming section about ***binding*** texture objects.

Oregon State University Computer Graphics

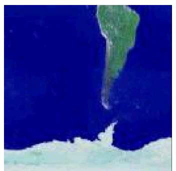
Texture Transformation

In addition to the Projection and ModelView matrices, OpenGL maintains a transformation for texture map coordinates **S** and **T** as well. You use all the same transformation functions you are used to: **glRotatef( ), glScalef( ), glTranslatef( )**, but you must first specify the **Matrix Mode**:

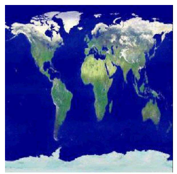
glMatrixMode(GL_TEXTURE);

The only trick to this is to remember that you are transforming the *texture coordinates*, not the *texture image*. Transforming the texture image forward is the same as transforming the texture coordinates backwards:

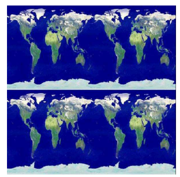
Scale = 0.5



Scale = 1.



Scale = 2.

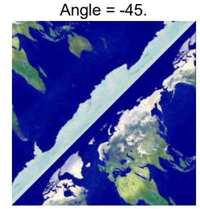


Oregon State University Computer Graphics

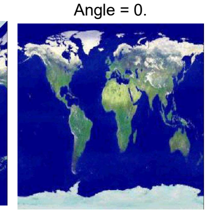
Texture Transformation

The only trick to this is to remember that you are transforming the texture coordinates, not the texture image. Transforming the texture image forward is the same as transforming the texture coordinates backwards:

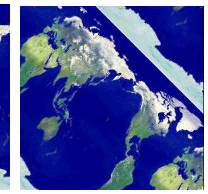
Angle = -45.



Angle = 0.



Angle = 45.



Oregon State University Computer Graphics

Texture Objects

The OpenGL **glTexImage2D** function doesn't just use that texture, it ***downloads*** all those bytes from the CPU to the GPU, *every time that call is made*! After the download, this texture becomes the "current texture image".

```
glTexImage2D( GL_TEXTURE_2D, level, ncomps, width, height, border, GL_RGB, GL_UNSIGNED_BYTE, texture );
```

If your scene has only one texture, this is easy to manage. Just do it once and forget about it.

But, if you have several textures, all to be used at different times on different objects, it will be important to maximize the efficiency of how you create, store, and manage those textures. In this case you should ***bind* texture objects**.

Texture objects leave your textures on the graphics card and then re-uses them, which is always going to be faster than re-loading them. Re-binding a texture object is basically "throwing a switch" in the GPU.

Oregon State University Computer Graphics

Texture Objects

Create a texture object by generating a texture name and then binding the texture object to the texture data and texture properties. The first time you execute **glBindTexture( ),** you fill the texture object. Subsequent times you do this, you are making that texture object current. So, create *global* Texture IDs like this:

```
int		Tex0, Tex1;			// global variables
. . .
Then, at the end of InitGraphics( ) you add:

int width0, height0, width1, height1;
unsigned char * textureArray0 = BmpToTexture( "image0.bmp", &width0, &height0 );
unsigned char * textureArray1 = BmpToTexture( "image1.bmp", &width1, &height1 );
glPixelStorei( GL_UNPACK_ALIGNMENT, 1 );

glGenTextures( 1, &Tex0 );			// assign binding "handles" to texture objects
glGenTextures( 1, &Tex1 );
. . .
glBindTexture( GL_TEXTURE_2D, Tex0 );  // make Tex0 the current texture and store its parameters

glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT );
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,  GL_REPEAT );
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR );
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,   GL_LINEAR);
glTexImage2D( GL_TEXTURE_2D, 0, 3, width0, height0, 0, GL_RGB, GL_UNSIGNED_BYTE, textureArray0 );
```

Oregon State University Computer Graphics

Texture Objects


```
glBindTexture( GL_TEXTURE_2D, Tex1 );   // make Tex1 the current texture and store its parameters

glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT );
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT );
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST );
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexImage2D( GL_TEXTURE_2D, 0, 3, width1, height1, 0, GL_RGB,GL_UNSIGNED_BYTE, textureArray1 );
```

Then, in **Display( ):**

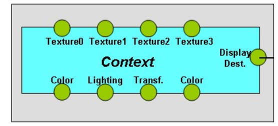
```
glEnable( GL_TEXTURE_2D );

glBindTexture( GL_TEXTURE_2D, Tex0 );
glTexEnvf( GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE );
glCallList( DL0 );


glBindTexture( GL_TEXTURE_2D, Tex1 );
glTexEnvf( GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE );
glCallList( DL1 );

glDisable( GL_TEXTURE_2D );
```


What Does "Binding" Really Mean?

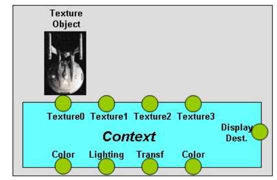
The OpenGL Rendering Context contains all the characteristic information necessary to produce an image from geometry. This includes the current transformations, colors, lighting, textures, where to send the display, etc.

The OpenGL term "binding" refers to "attaching" or "docking" (a metaphor which I find to be more visually pleasing) an OpenGL object to the Context. You can then assign characteristics, and they will "flow" through the Context into the object.

Before Binding



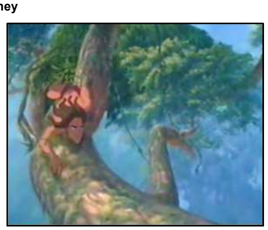
After Binding



Some Great Uses for Texture Mapping you have seen in the Movies

Disney

Disney



Disney

Yes, I know, I know, these are older examples, but I especially like them because, at the time, the CG (and the textures) became part of the story-telling for the first time.

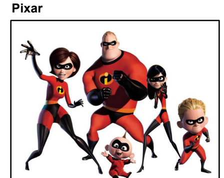
Some Great Uses for Texture Mapping you have seen in the Movies More Recently

Disney

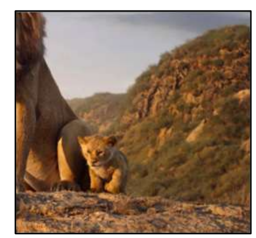
Pixar

Disney

Pixar



Disney

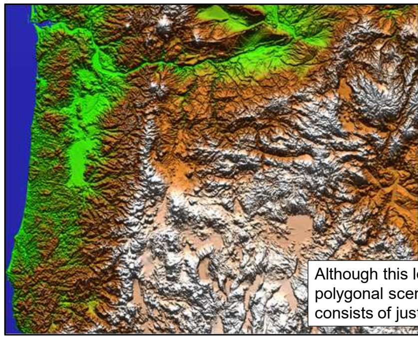


Disney

Bonus Topic: Procedural Texture Mapping

You can also create a texture from data on-the-fly. In this case, the fragment shader takes a grid of heights and uses cross-products to produce surface normal vectors for lighting.

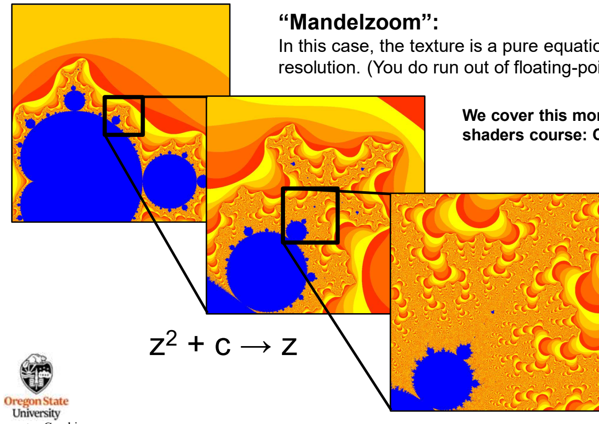
While this is "procedural", the amount of height data is finite, so you can still run out of resolution



We cover this more in the shaders course: CS 457/557

Although this looks like an incredible amount of polygonal scene detail, the geometry for this scene consists of just a ***single* quadrilateral**

Bonus Topic: Procedural Texture Mapping



$$z^2 + c \longrightarrow z$$

"Mandelzoom":

In this case, the texture is a pure equation, so you never run out of resolution. (You do run out of floating-point precision, however.)

We cover this more in the shaders course: CS 457/557