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

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.

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.

To draw a polygon, 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 in to draw each pixel, it uses that pixel's interpolated (s,t) to lookup a color in the texture image.

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:
```
gBegin( GL_TRIANGLES );
gTexCoord2f( s0, t0 );
gNormal3f( nx0, ny0, nz0 );
gVertex3f( x0, y0, z0 );
gTexCoord2f( s1, t1 );
gNormal3f( nx1, ny1, nz1 );
gVertex3f( x1, y1, z1 );
. . .
gEnd( );
```

(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:
```
gDisable( GL_TEXTURE_2D );
```

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

```
(X2, Y2, Z2, S2,T2)
(X1, Y1, Z1, S1,T1)
(X3, Y3, Z3, S3,T3)
(X4, Y4, Z4, S4,T4)
```

Interpolated (S,T) = (.78, .67)

(.78, .67) in S and T = (199.68, 171.52) in texels

```
[199.68, 171.52]
```

You specify an (s,t) pair at each vertex, along with the vertex coordinates. 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 in to draw each pixel, it uses that pixel's interpolated (s,t) to lookup a color in the texture image.
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:

\[
s = \frac{x - xmin}{xmax - xmin} \quad t = \frac{y - ymin}{ymax - ymin}
\]

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:

\[
s = \frac{\Theta - (-\pi)}{2\pi} \quad t = \frac{\Phi - (-\frac{\pi}{2})}{\pi}
\]

The OsuSphere code does it like this:

\[
s = \frac{\text{lng} + M\_\pi}{2*M\_\pi}; \quad t = \frac{\text{lat} + M\_\pi/2.}{M\_\pi};
\]

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

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

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

Be careful where \(s\) abruptly transitions from 1. back to 0.

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

Reading a Texture from a BMP File

unsigned char *BmpToTexture(char *name, int *width, int *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.
Define the texture wrapping parameters. This will control what happens when a texture coordinate is greater than 1.0 or less than 0.0:

```c
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 the 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.

---

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

```c
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.

---

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

```c
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.

---

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:

```c
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:
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." 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.

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:

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

Then, in Display():

```
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.

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

"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.)