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.

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);
 glVertex3f(x0, y0, z0);
 glTexCoord2f(s1, t1);
 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:
gDisable(GL_TEXTURE_2D);
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:

\[ s = \frac{x - X_{\text{min}}}{X_{\text{max}} - X_{\text{min}}} \]

\[ t = \frac{y - Y_{\text{min}}}{Y_{\text{max}} - Y_{\text{min}}} \]

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:

\[ s = \frac{\theta - (-\pi)}{2\pi} \]

\[ t = \frac{\phi - (-\pi/2)}{\pi} \]

The Sphere code does it like this:

\[ s = \frac{\text{lng} + \pi}{2\pi} \]

\[ t = \frac{\text{lat} + \pi/2}{\pi} \]

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

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

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

Be careful where a abruptly transitions from 1. back to 0.

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

Reading in a Texture from a BMP File

unsigned char *BmpToTexture( char *, int *, int * );

unsigned char *Texture;

... Texture = BmpToTexture( "filename.bmp", &width, &height );

This function is found in your sample code.

Note: this function should be called once, and called from InitGraphics() .

Do not call it from the Display() function. Do not call it from the Display() function. Do not call it 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:

```
#define of GL_TEXTURE_2D
#define of GL_TEXTURE_WRAP_S
#define of GL_TEXTURE_WRAP_T

where
```

**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.

---

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

```
#define of GL_TEXTURE_MAG_FILTER
#define of GL_TEXTURE_MIN_FILTER

where
```

**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:

```
#define of GL_TEXTURE_ENV
#define of GL_TEXTURE_ENV_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 routines you are used to: 

```
#define of GL_TEXTURE_ENV
#define of GL_TEXTURE_ENV_MODE

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 only trick to this is to remember that you are transforming the texture coordinates, not the texture image.
The OpenGL `glTexImage2D` function doesn’t just use that texture, it downloads it from the CPU to the GPU, every time that call is made! After the download, this texture becomes the “current texture image”.

```c
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.

```c
GLuint Tex0, Tex1; // global variables
...
Then, at the end of `InitGraphics()` you add:
```
```c
int width0, height0, width1, height1;
unsigned char * TextureArray0 = BmpToTexture( "file0.bmp", &width0, &height0 );
unsigned char * TextureArray1 = BmpToTexture( "file1.bmp", &width1, &height1 );
```

```c
glPixelStorei( GL_UNPACK_ALIGNMENT, 1 );
```

```c
glGenTextures( 1, &Tex0 ); // assign binding "handles"
glGenTextures( 1, &Tex1 );
```

```c
glBindTexture( GL_TEXTURE_2D, Tex0 ); // make the Tex0 texture current and set its parameters
```

```c
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP );
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,  GL_CLAMP );
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR );
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,   GL_LINEAR);
```

```c
glTexImage2D( GL_TEXTURE_2D, 0, 3, width0, height0, 0, GL_RGB, GL_UNSIGNED_BYTE, TextureArray0 );
```

```c
Then, in `Display()`:  
```c
glEnable( GL_TEXTURE_2D );
```

```c
glBindTexture( GL_TEXTURE_2D, Tex0 );
```

```c
glTexEnvf( GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE );
```

```c
glCallList( List0 );
```

```c
glBindTexture( GL_TEXTURE_2D, Tex1 );
```

```c
glTexEnvf( GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE );
```

```c
glCallList( List1 );
```

```c
glDisable( GL_TEXTURE_2D );
```

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.

### What Does “Binding” Really Mean?

Before Binding

After Binding

---

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

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

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

"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.