The Basic Idea

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. At one time, some graphics hardware required the image’s pixel dimensions to be a power of two. This restriction has been lifted on most (all?) graphics cards, but just to be safe…

Also, to prevent confusion, the texture 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’s width and height dimensions are not called X and Y. They are called S and T. A texture map is not generally 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.

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

In OpenGL terms: assigning an (s,t) to each vertex

Enable texture mapping:
```
glEnable( GL_TEXTURE_2D );
```

Draw your polygons, specifying s and t at each vertex:
```
BEGIN( GL_POLYGON );
gTexCoord2f( s0, t0 );
Normal3f( nx0, ny0, nz0 );
Vertex3f( x0, y0, z0 );
```
```
gTexCoord2f( s1, t1 );
Normal3f( nx1, ny1, nz1 );
Vertex3f( x1, y1, z1 );
```
```
. . .
```
```
END( );
```

In OpenGL terms: assigning an (s,t) to each vertex

```
BEGIN( GL_POLYGON );
gTexCoord2f( s0, t0 );
Norm3f( nx0, ny0, nz0 );
gVertex3f( x0, y0, z0 );
```
```
gTexCoord2f( s1, t1 );
Norm3f( nx1, ny1, nz1 );
gVertex3f( x1, y1, z1 );
```
```
. . .
```
```
END( );
```

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,

```
s = (x - Xmin) / (Xmax - Xmin)
t = (y - Ymin) / (Ymax - Ymin)
```

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

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

Or, for a sphere,

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

From the Sphere code:

\[ s = \left(\text{lng} + \pi\right) / (2.*\pi) \]
\[ t = \left(\text{lat} + \pi/2\right) / \pi; \]

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

\[ s = ? \quad t = ? \]

You really are at the mercy of whoever did the modeling...

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

You create your texture here

Memory Types

<table>
<thead>
<tr>
<th>CPU Memory</th>
<th>GPU Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Host Visible</td>
<td>Device Local</td>
</tr>
<tr>
<td>GPU Memory (the &quot;Staging Buffer&quot;)</td>
<td>GPU Memory</td>
</tr>
<tr>
<td>Texture</td>
<td>Texture Sampling</td>
</tr>
<tr>
<td>RGBA to</td>
<td>Hardware</td>
</tr>
<tr>
<td>the Shader</td>
<td>RGBA in</td>
</tr>
<tr>
<td></td>
<td>the Shader</td>
</tr>
</tbody>
</table>
Textures' Undersampling Artifacts

As an object gets farther away and covers a smaller and smaller part of the screen, the texels : pixels ratio used in the coverage becomes larger and larger. This means that there are pieces of the texture leftover in between the pixels that are being drawn into, so that some of the texture image is not being taken into account in the final image. This means that the texture is being undersampled and could end up producing artifacts in the rendered image.

Textures Mip*-mapping

Texture sampling is performed against mip-maps, which are lower-resolution resolutions of the texture. As an object gets farther away, more mip-maps are used, reducing the texels : pixels ratio and thus allowing more of the texture to be included in the rendered image.

Texture Mip*-mapping Parameters

Different APIs and libraries may have different parameters for mip-mapping. For example, in Vulkan, the `VkSamplerCreateInfo` structure contains parameters such as `maxLod`, `compareOp`, `compareEnable`, `maxAnisotropy`, `anisotropyEnable`, `mipLodBias`, and others.

Memory Types

Different memory types are available for different purposes, such as device local, host visible, host coherent, and host cached. These memory types are used to optimize the performance and memory usage of applications.
Note that, at this point, the CPU buffer and the GPU Staging Buffer are no longer needed, and can be destroyed.

Reading in a Texture from a BMP File

```c
typedef struct MyTexture
{
    uint32_t                        width;
    uint32_t                        height;
    VkImage texImage;
    VkImageView texImageView;
    VkSampler texSampler;
    VkDeviceMemory vdm;
} MyTexture;

MyTexture MyPuppyTexture;
```

This function can be found in the `sample.cpp` file. The BMP file needs to be created by something that writes uncompressed 24-bit color BMP files, or was converted to the uncompressed BMP format by a tool such as ImageMagick's `convert`, Adobe Photoshop, or GNU's GIMP.

Anisotropic Texture Filtering