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.

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.

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

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

Draw your polygons, specifying a and b at each vertex:
```
 glBegin(GL_POLYGON);

glTexCoord2f(a, b);
glNormal3f(nx, ny, nz);
glVertex3f(x, y, z);

. . .

. . .

. . .

 glEnd();
```

Disable texture mapping:
```
 glDisable(GL_TEXTURE_2D);
```

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,

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

\[
t = \frac{y - Y_{\text{min}}}{Y_{\text{max}} - Y_{\text{min}}}
\]
Using a Texture: How do you know what (s,t) to assign to each vertex?

Or, for a sphere,

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

$$t = \frac{\phi - (-\frac{\pi}{2})}{\pi}$$

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...

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

Memory Types

CPU Memory

GPU Memory

Device Local GPU Memory

Host Visible GPU Memory

memcpy( ) vkCmdCopyImage( )

NVIDIA Discrete Graphics:

11 Memory Types:

Memory 0:

Memory 1:

Memory 2:

Memory 3:

Memory 4:

Memory 5:

Memory 6:

Memory 7: DeviceLocal

Memory 8: DeviceLocal HostVisible HostCoherent HostCached

Memory 9: HostVisible HostCoherent HostCached

Memory 10: HostVisible HostCoherent HostCached

Intel Integrated Graphics:

3 Memory Types:

Memory 0: DeviceLocal

Memory 1: DeviceLocal HostVisible HostCoherent

Memory 2: DeviceLocal HostVisible HostCoherent HostCached
Graphics hardware determines which level to use based on the texels : pixels ratio.

In addition to just picking one mip-map level, the rendering system can sample from two of them, one less that the T:P ratio and one more, and then blend the two RGBAs returned. This is known as VK_SAMPLER_MIPMAP_MODE_LINEAR.

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.

Consider a texture that consists of one red texel and all the rest white. It is easy to imagine an object rendered with that texture as ending up all white, with the red texel having never been included in the final image. The solution is to create lower-resolutions of the same texture so that the red texel gets included somehow in all resolution-level textures.
result = vkCreateImage( LogicalDevice, IN &vici, PALLOCATOR, OUT &textureImage); // allocated, but not filled

// transition the staging buffer layout:

// *******************************************************************************

// transition the texture buffer layout:

// *******************************************************************************
// create an image view for the texture image
// (an "image view" is used to indirectly access an image)

VkImageSubresourceRange visr;
visr.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
visr.baseMipLevel = 0;
visr.levelCount = 1;
visr.baseArrayLayer = 0;
visr.layerCount = 1;

VkImageViewCreateInfo vivci;
vivci.sType = VK_STRUCTURE_TYPE_IMAGE_VIEW_CREATE_INFO;
vivci.pNext = nullptr;
vivci.flags = 0;
vivci.image = textureImage;
vivci.viewType = VK_IMAGE_VIEW_TYPE_2D;
vivci.format = VK_FORMAT_R8G8B8A8_UNORM;
vivci.components.r = VK_COMPONENT_SWIZZLE_R;
vivci.components.g = VK_COMPONENT_SWIZZLE_G;
vivci.components.b = VK_COMPONENT_SWIZZLE_B;
vivci.components.a = VK_COMPONENT_SWIZZLE_A;
vivci.subresourceRange = visr;

result = vkCreateImageView(LogicalDevice, IN &vivci, PALLOCATOR, OUT &pMyTexture->texImageView);
return result;

Note that, at this point, the Staging Buffer is no longer needed, and can be destroyed.

Image View

Reading in a Texture from a BMP File

typedef struct MyTexture
{
    uint32_t width, 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.