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.

Enable texture mapping:

```glEnable(GL_TEXTURE_2D);```

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

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

```glBegin(GL_POLYGON);
gTexCoord2f(s0, t0);
gNormal3f(nx0, ny0, nz0);
Vertex3f(x0, y0, z0);
gTexCoord2f(s1, t1);
gNormal3f(nx1, ny1, nz1);
Vertex3f(x1, y1, z1);
. . .
gEnd();```

Disable texture mapping:

```glDisable(GL_TEXTURE_2D);```
Triangles in an Array of Structures

struct vertex
{
    glm::vec3 position;
    glm::vec3 normal;
    glm::vec3 color;
    glm::vec2 texCoord;
};

struct vertex VertexData[  ] =
{
    // triangle 0-2-3:
    // vertex #0:
    {
        {-1., -1., -1.},
        { 0.,  0., -1.},
        { 0.,  0.,  0.},
        { 1., 0. }
    },
    // vertex #2:
    {
        {-1.,  1., -1.},
        { 0.,  0., -1.},
        { 0.,  1.,  0.},
        { 1., 1. }
    },
    // vertex #3:
    {
        { 1.,  1., -1.},
        { 0.,  0., -1.},
        { 1.,  1.,  0.},
        { 0., 1. }
    },
    0 1 3
};

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,

\[ s = \frac{x - X_{\text{min}}}{X_{\text{max}} - X_{\text{min}}} \quad t = \frac{y - Y_{\text{min}}}{Y_{\text{max}} - Y_{\text{min}}} \]

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

Or, for a sphere,

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

\[ s = \frac{\text{lng} + \pi}{2\pi}, \quad t = \frac{\text{lat} + \pi/2}{\pi} \]

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

NVIDIA A6000 Graphics:
- 6 Memory Types:
  - Memory 0: DeviceLocal
  - Memory 1: HostVisible HostCoherent
  - Memory 2: HostVisible HostCoherent HostCached
  - Memory 3: DeviceLocal HostVisible HostCoherent
  - Memory 4: DeviceLocal HostVisible HostCoherent HostCached
  - Memory 5: DeviceLocal

Intel Integrated Graphics:
- 3 Memory Types:
  - Memory 0: DeviceLocal
  - Memory 1: DeviceLocal HostVisible HostCoherent
  - Memory 2: DeviceLocal HostVisible HostCoherent HostCached
I find it handy to encapsulate texture information in a struct, just like I do with buffer information:

```c
// holds all the information about a data buffer so it can be encapsulated in one variable:
typedef struct MyBuffer {
    VkDataBuffer buffer;
    VkDeviceMemory vdm;
    VkDeviceSize size;
} MyBuffer;
```

```c
// holds all the information about a texture so it can be encapsulated in one variable:
typedef struct MyTexture {
    uint32_t                        width;
    uint32_t                        height;
    unsigned char *            pixels;
    VkImage texImage;
    VkImageView texImageView;
    VkSampler texSampler;
    VkDeviceMemory vdm;
} MyTexture;
```

```
MyTexture MyPuppyTexture;
```

```c
VkSamplerCreateInfo vsci;
vsci.magFilter = VK_FILTER_LINEAR;
vsci.minFilter = VK_FILTER_LINEAR;
vsci.mipmapMode = VK_SAMPLER_MIPMAP_MODE_LINEAR;
vsci.addressModeU = VK_SAMPLER_ADDRESS_MODE_REPEAT;
vsci.addressModeV = VK_SAMPLER_ADDRESS_MODE_REPEAT;
vsci.addressModeW = VK_SAMPLER_ADDRESS_MODE_REPEAT;
```

```
result = vkCreateSampler( LogicalDevice, IN &vsci, PALLOCATOR, OUT &MyPuppyTexture->texSampler);
```

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-resolution textures by averaging 4 pixels to make a new one. This reduces the total texture storage by about 2x. The rendering system is simply sampling from two levels in the mip-map and blending them. This is known as VK_SAMPLER_MIPMAP_MODE_LINEAR.
#ifdef CHOICES

VK_BORDER_COLOR_INT_OPAQUE_WHITE
VK_BORDER_COLOR_INT_OPAQUE_BLACK
VK_BORDER_COLOR_FLOAT_OPAQUE_BLACK
VK_BORDER_COLOR_INT_TRANSPARENT_BLACK
VK_BORDER_COLOR_FLOAT_TRANSPARENT_BLACK

VK_COMPARE_OP_ALWAYS
VK_COMPARE_OP_GREATER_OR_EQUAL
VK_COMPARE_OP_NOT_EQUAL
VK_COMPARE_OP_GREATER
VK_COMPARE_OP_EQUAL
VK_COMPARE_OP_LESS

#ifdef CHOICES

VK_SAMPLER_ADDRESS_MODE_MIRROR_CLAMP_TO_EDGE
VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE
VK_SAMPLER_ADDRESS_MODE_MIRRORED_REPEAT
VK_SAMPLER_ADDRESS_MODE_REPEAT

Init07TextureSampler( MyTexture * pMyTexture )

VkResult

vsci.unnormalizedCoordinates = VK_FALSE; // VK_TRUE means we are use raw texels as the index
vsci.borderColor = VK_BORDER_COLOR_FLOAT_OPAQUE_BLACK;

vsci.maxLod = 0.;
vsci.minLod = 0.;

vsci.compareEnable = VK_FALSE;

vsci.maxAnisotropy = 1.;

vsci.anisotropyEnable = VK_FALSE;

vsci.addressModeW = VK_SAMPLER_ADDRESS_MODE_REPEAT;

vsci.addressModeV = VK_SAMPLER_ADDRESS_MODE_REPEAT;

vsci.addressModeU = VK_SAMPLER_ADDRESS_MODE_REPEAT;

vsci.mipmapMode = VK_SAMPLER_MIPMAP_MODE_LINEAR;

vsci.minFilter = VK_FILTER_LINEAR;

vsci.magFilter = VK_FILTER_LINEAR;

vsci.pNext = nullptr;

Init07TextureBuffer( INOUT MyTexture * pMyTexture)

VkResult

// this first {...} is to create the staging image:

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

// this first {...} is to create the staging image:

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

// this first {...} is to create the staging image:

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

VkImage textureImage;

VkDeviceSize textureSize = texWidth * texHeight * 4; // rgba, 1 byte each
unsigned char *texture = pMyTexture->pixels;
uint32_t texHeight = pMyTexture->height;
uint32_t texWidth = pMyTexture->width;

vkCreateImage(LogicalDevice, IN &vici, PALLOCATOR, OUT &textureImage); // allocated, but not filled
vkUnmapMemory(LogicalDevice, IN stagingImage, OUT &vdm);
vkAllocateMemory(LogicalDevice, IN stagingImage, IN vdm, 0); // 0 = offset

FindMemoryThatIsHostVisible();

vkMapMemory(LogicalDevice, vdm, 0, VK_WHOLE_SIZE, 0, OUT &gpuMemory);

vkBindImageMemory(LogicalDevice, stagingImage, textureImage, 0);

vkGetImageSubresourceLayout(LogicalDevice, textureImage, OUT &vsl);

if (Verbose)
    vkGetImageSubresourceLayout(LogicalDevice, textureImage, OUT &vsl);

vis.arrayLayer = 0;
vis.mipLevel = 0;
vis.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;

vkGetImageMemoryRequirements(LogicalDevice, textureImage, OUT &vmr);

vcai.pQueueFamilyIndices = (const uint32_t *)nullptr;
vcai.queueFamilyIndexCount = 0;

vcai.pImageCreateInfo = &vici;

vkCreateImage(LogicalDevice, IN &vcai, PALLOCATOR, OUT &textureImage);

if (Verbose)
    vkCreateImage(LogicalDevice, IN &vcai, PALLOCATOR, OUT &textureImage);

if (Verbose)
    vkCreateImage(LogicalDevice, IN &vcai, PALLOCATOR, OUT &textureImage);

if (Verbose)
    vkCreateImage(LogicalDevice, IN &vcai, PALLOCATOR, OUT &textureImage);

memcpy(gpuBytes, texture, (size_t)textureSize);

if (vsl.rowPitch == 4 * texWidth)
    memcpy(gpuMemory, texture, (size_t)textureSize);

else
    memcpy(&gpuBytes[y * vsl.rowPitch], &texture[4 * y * texWidth], (size_t)(4*texWidth) );
// this second {...} is to create the actual texture image:
// *******************************************************************************
result = vkCreateImage
    ( LogicalDevice, IN &vmai, PALLOCATOR, OUT &vdm);
// transition the staging buffer layout:
// *******************************************************************************
result = vkCmdPipelineBarrier
    ( TextureCommandBuffer, IN &vcbbi);
// copy pixels from the staging image to the texture:
// *******************************************************************************
result = vkCmdCopyImage
    ( TextureCommandBuffer, IN &vic);
// transition the texture image:
// *******************************************************************************
result = vkBindImageMemory
    ( LogicalDevice, IN &vmai, PALLOCATOR, OUT &vdm);
result = vkAllocateMemory
    ( LogicalDevice, IN textureImage, IN vdm, 0);        // 0 = offset
FindMemoryThatIsDeviceLocal
    ( LogicalDevice, IN textureImage, OUT &vmr);
result = vkCreateImage
    ( LogicalDevice, IN &vici, PALLOCATOR, OUT &ve3);
result = vkAllocateMemory
    ( LogicalDevice, IN ve3, OUT &vo3);
result = vkBindImageMemory
    ( LogicalDevice, IN &vmai, PALLOCATOR, OUT &vdm);
Note that, at this point, the Staging Buffer is no longer needed, and can be destroyed.

Reading in a Texture from a BMP File

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.