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. This can be most any image. 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... The X and Y dimensions did not need to be the same power of two, just a power of two. So, a 128x512 image would have been OK; a 129x511 image might not have.

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 );
gVertex3f( x0, y0, z0 );
gTexCoord2f( s1, t1 );
gNormal3f( nx1, ny1, nz1 );
gVertex3f( x1, y1, z1 );
...```
```gEnd();```

Disable texture mapping:
```gDisable( GL_TEXTURE_2D );```
Triangles in an Array of Structures

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

```
struct vertex VertexData[3] =
{
    // 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. }
};
```

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,

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

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

Or, for a sphere,

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

From the Sphere code:

```c
s = ( lng + M_PI ) / ( 2.*M_PI );
t = ( lat + M_PI/2. ) / M_PI;
```
You really are at the mercy of whoever did the modeling...

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

VkDescriptorSetLayoutBinding TexSamplerSet[1];
TexSamplerSet[1].binding            = 0;
TexSamplerSet[1].descriptorType = VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER;
TexSamplerSet[1].stageFlags = VK_SHADER_STAGE_FRAGMENT_BIT;
TexSamplerSet[1].pImmutableSamplers = (VkSampler *)nullptr;
TexSamplerSet[1].descriptorCount = 1;

VkDescriptorImageInfo vdii0;
vdii0.sampler   = MyPuppyTexture.texSampler;
vdii0.imageView = MyPuppyTexture.texImageView;
vdii0.imageLayout = VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL;

VkWriteDescriptorSet vwds3;
vwds3.sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
vwds3.pNext = nullptr;
vwds3.dstSet = DescriptorSets[3];
vwds3.dstBinding = 0;vwds3.dstArrayElement = 0;
vwds3.descriptorCount = 1;
vwds3.descriptorType = VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER;
vwds3.pImageInfo = &vdii0;
vwds3.pTexelBufferView = (VkBufferView *)nullptr;

You create your texture here

CPU Memory

Host

Visible

GPU Memory (the “Staging Buffer”)

Device

Local

GPU Memory

Texture Sampling Hardware

RGBA to the Shader
**Memory Types**

**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
- Memory 9: HostVisible HostCoherent
- 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

---

**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.

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.

---

**Texture Sampling Parameters**

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

```
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, &vsci, PALLOCATOR, pTextureSampler);
```

---

**Texture Mip*-mapping**

- Total texture storage is ~ 2x what it was without mip-mapping
- 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**.

---

*Latin: multim in parvo, “many things in a small place”*
VkResult Init07TextureSampler(MyTexture * pMyTexture)
{
    Init07TextureBuffer(INOUT MyTexture * pMyTexture);

    VkResult result;  
    VkSamplerCreateInfo vsci;  
    uint32_t texWidth = pMyTexture->width;  
    uint32_t texHeight = pMyTexture->height;  
    unsigned char *texture = pMyTexture->pixels;  
    VkDeviceSize textureSize = texWidth * texHeight * 4;  
    // rgba, 1 byte each  
    vsci.sType = VK_STRUCTURE_TYPE_SAMPLER_CREATE_INFO;  
    vsci.pNext = nullptr;  
    vsci.flags = 0;  
    vsci.addressModeU = VK_SAMPLER_ADDRESS_MODE_REPEAT;  
    vsci.addressModeV = VK_SAMPLER_ADDRESS_MODE_REPEAT;  
    vsci.addressModeW = VK_SAMPLER_ADDRESS_MODE_REPEAT;  
    vsci.addressMode = VK_SAMPLER_ADDRESS_MODE_REPEAT;  
    vsci.mipLodBias = 0.;  
    vsci.anisotropyEnable = VK_FALSE;  
    vsci.compareEnable = VK_FALSE;  
    vsci.compareOp = VK_COMPARE_OP_NEVER;  
    vsci.format = VK_FORMAT_R8G8B8A8_UNORM;  
    vsci.extent.width = texWidth;  
    vsci.extent.height = texHeight;  
    vsci.extent.depth = 1;  
    vsci.usage = VK_IMAGE_USAGE_TRANSFER_SRC_BIT;  
    vsci.queueFamilyIndexCount = 0;  
    vsci.pQueueFamilyIndices = (const uint32_t *)nullptr;  
    result = vkCreateSampler(LogicalDevice, IN &vsci, PALLOCATOR, OUT &pMyTexture->texSampler);

    VkImageCreateInfo vici;  
    vsci.imageType = VK_IMAGE_TYPE_2D;  
    vsci.format = VK_FORMAT_R8G8B8A8_UNORM;  
    vsci.extent.width = texWidth;  
    vsci.extent.height = texHeight;  
    vsci.extent.depth = 1;  
    vsci.usage = VK_IMAGE_USAGE_TRANSFER_SRC_BIT;  
    vsci.sharingMode = VK_SHARING_MODE_EXCLUSIVE;  
    result = vkCreateImage(LogicalDevice, IN stagingImage, OUT &vici);

    VkMemoryRequirements vmr;  
    vkGetImageMemoryRequirements(LogicalDevice, stagingImage, OUT &vmr);

    // find memory that is host visible: because we want to mmap it
    VkDeviceMemory vdm;  
    FindMemoryThatIsHostVisible(&vdm);

    if (vsl.rowPitch == 4 * texWidth)
    {
        fprintf(FpDebug, "Image vmr.size = %lld
", vmr.size);
        fprintf(FpDebug, "Image vmr.alignment = %lld
", vmr.alignment);
        fprintf(FpDebug, "Image vmr.memoryTypeBits = 0x%08x
", vmr.memoryTypeBits);
        memcpy(gpuMemory, (void *)texture, (size_t)textureSize);
    } else{
        unsigned char *gpuBytes = (unsigned char *)gpuMemory;  
        for (unsigned int y = 0; y < texHeight; y++)
        {
            if (vsl.rowPitch == 4 * texWidth)
            {
                memcpy(&gpuBytes[y * vsl.rowPitch], &texture[4 * y * texWidth], (size_t)(4*texWidth) );
            }
            else  
            {
                memcpy(&gpuBytes[y * vsl.rowPitch], &texture[4 * y * texWidth], (size_t)(4*texWidth) );
            }
        }
    }

    VkImageSubresource vis;  
    vkGetImageSubresourceLayout(LogicalDevice, stagingImage, IN &vis, OUT &vsl);
    if (Verbose)
    {
        fprintf(FpDebug, "Subresource Layout:
"  
        fprintf(FpDebug, "	offset = %lld
", vsl.offset);
        fprintf(FpDebug, "	size = %lld
", vsl.size);
        fprintf(FpDebug, "	rowPitch = %lld
", vsl.rowPitch);
        fprintf(FpDebug, "	arrayPitch = %lld
", vsl.arrayPitch);
        fprintf(FpDebug, "	depthPitch = %lld
", vsl.depthPitch);
        fflush(FpDebug);
    }

    VkImageCreateInfo vici;  
    vici.imageType = VK_IMAGE_TYPE_2D;  
    vici.format = VK_FORMAT_R8G8B8A8_UNORM;  
    vici.extent.width = texWidth;  
    vici.extent.height = texHeight;  
    vici.extent.depth = 1;  
    vici.usage = VK_IMAGE_USAGE_TRANSFER_SRC_BIT;  
    vici.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
    result = vkCreateImage(LogicalDevice, IN stagingImage, OUT &vici);
    vmr = vkGetImageMemoryRequirements(LogicalDevice, stagingImage, OUT &vmr);
    vici.usage = VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT;
    vici.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
    // find memory that is host visible: because we want to mmap it
    VkDeviceMemory vdm;  
    FindMemoryThatIsHostVisible(&vdm);
    result = vkCreateImage(LogicalDevice, IN stagingImage, OUT &vici);  
    vmr = vkGetImageMemoryRequirements(LogicalDevice, stagingImage, OUT &vmr);
    vici.usage = VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT;  
    vici.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
    // find memory that is host visible: because we want to mmap it
    VkDeviceMemory vdm;  
    FindMemoryThatIsHostVisible(&vdm);
    result = vkCreateImage(LogicalDevice, IN stagingImage, OUT &vici);  
    vmr = vkGetImageMemoryRequirements(LogicalDevice, stagingImage, OUT &vmr);  
    vici.usage = VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT;  
    vici.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
    // find memory that is host visible: because we want to mmap it
    VkDeviceMemory vdm;  
    FindMemoryThatIsHostVisible(&vdm);
    result = vkCreateImage(LogicalDevice, IN stagingImage, OUT &vici);  
    vmr = vkGetImageMemoryRequirements(LogicalDevice, stagingImage, OUT &vmr);  
    vici.usage = VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT;  
    vici.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
    // find memory that is host visible: because we want to mmap it
    VkDeviceMemory vdm;  
    FindMemoryThatIsHostVisible(&vdm);
    result = vkCreateImage(LogicalDevice, IN stagingImage, OUT &vici);  
    vmr = vkGetImageMemoryRequirements(LogicalDevice, stagingImage, OUT &vmr);  
    vici.usage = VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT;  
    vici.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
    // find memory that is host visible: because we want to mmap it
    VkDeviceMemory vdm;  
    FindMemoryThatIsHostVisible(&vdm);
    result = vkCreateImage(LogicalDevice, IN stagingImage, OUT &vici);  
    vmr = vkGetImageMemoryRequirements(LogicalDevice, stagingImage, OUT &vmr);  
    vici.usage = VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT;  
    vici.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
// *******************************************************************************
// copy pixels from the staging image to the texture:
// this second {...} is to create the actual texture image:
// *******************************************************************************

VkCommandBufferBeginInfo vcbbi;
{
    vcbbi.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO;
    vcbbi.pNext = nullptr;
    vcbbi.flags = VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT;
    vcbbi.pInheritanceInfo = (VkCommandBufferInheritanceInfo *)nullptr;
}

VkImageCreateInfo vici;
{
    vici.sType = VK_STRUCTURE_TYPE_IMAGE_CREATE_INFO;
    vici.pNext = nullptr;
    vici.flags = 0;
    vici.imageType = VK_IMAGE_TYPE_2D;
    vici.format = VK_FORMAT_R8G8B8A8_UNORM;
    vici.extent.width = texWidth;
    vici.extent.height = texHeight;
    vici.extent.depth = 1;
    vici.mipLevels = 1;
    vici.arrayLayers = 1;
    vici.samples = VK_SAMPLE_COUNT_1_BIT;
    vici.tiling = VK_IMAGE_TILING_OPTIMAL;
    vici.usage = VK_IMAGE_USAGE_TRANSFER_DST_BIT | VK_IMAGE_USAGE_SAMPLED_BIT;
    vici.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
    vici.queueFamilyIndexCount = 0;
    vici.pQueueFamilyIndices = (const uint32_t *)nullptr;
}

result = vkCreateImage(LogicalDevice, IN &vici, PALLOCATOR, OUT &textureImage); // allocated, but not filled

VkImageMemoryBarrier vimb;
{
    vimb.sType = VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER;
    vimb.pNext = nullptr;
    vimb.oldLayout = VK_IMAGE_LAYOUT_PREINITIALIZED;
    vimb.newLayout = VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL;
    vimb.srcQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
    vimb.dstQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
    vimb.image = textureImage;
    vimb.srcAccessMask = 0;
    vimb.dstAccessMask = 0;
    vimb.subresourceRange = visr;
}

vkCmdPipelineBarrier(TextureCommandBuffer, VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT, VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT, 0, 0, (VkMemoryBarrier *)nullptr, 0, (VkBufferMemoryBarrier *)nullptr, 1, IN &vimb);

// ************************************************************ *******************
// transition the texture buffer layout:// *******************************************************************************
// because we want to sample from it

VkImageSubresourceLayers visl;
{
    visl.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
    visl.mipLevel = 0;
    visl.baseArrayLayer = 0;
    visl.layerCount = 1;
}

VkOffset3D vo3;
{
    vo3.x = 0;
    vo3.y = 0;
    vo3.z = 0;
}

VkExtent3D ve3;
{
    ve3.width = texWidth;
    ve3.height = texHeight;
    ve3.depth = 1;
}

vkCmdCopyImage(TextureCommandBuffer, stagingImage, VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL, textureImage, VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL, 1, IN &vic);

// now do the final image transfer:

// ************************************************************ *******************
// transition the staging buffer layout:// *******************************************************************************

VkImageCopy vic;
{
    vic.srcSubresource = visl;
    vic.srcOffset = vo3;
    vic.dstSubresource = visl;
    vic.dstOffset = vo3;
    vic.extent = ve3;
}

VkImageMemoryBarrier vimb;
{
    vimb.sType = VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER;
    vimb.pNext = nullptr;
    vimb.oldLayout = VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL;
    vimb.newLayout = VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL;
    vimb.srcQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
    vimb.dstQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
    vimb.image = stagingImage;
    vimb.srcAccessMask = 0;
    vimb.dstAccessMask = VK_ACCESS_SHADER_READ_BIT;
    vimb.subresourceRange = visr;
}

vkCmdPipelineBarrier(TextureCommandBuffer, VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT, VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT, 0, 0, (VkMemoryBarrier *)nullptr, 0, (VkBufferMemoryBarrier *)nullptr, 1, IN &vimb);

// ************************************************************ *******************
// transition the final image buffer layout:// *******************************************************************************

VkImageCopy vic;
{
    vic.srcSubresource = visl;
    vic.srcOffset = vo3;
    vic.dstSubresource = visl;
    vic.dstOffset = vo3;
    vic.extent = ve3;
}

VkImageMemoryBarrier vimb;
{
    vimb.sType = VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER;
    vimb.pNext = nullptr;
    vimb.oldLayout = VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL;
    vimb.newLayout = VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL;
    vimb.srcQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
    vimb.dstQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
    vimb.image = finalImage;
    vimb.srcAccessMask = 0;
    vimb.dstAccessMask = VK_ACCESS_SHADER_READ_BIT;
    vimb.subresourceRange = visr;
}

vkCmdPipelineBarrier(TextureCommandBuffer, VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT, VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT, 0, 0, (VkMemoryBarrier *)nullptr, 0, (VkBufferMemoryBarrier *)nullptr, 1, IN &vimb);

// now do the final image transfer:

VkImageCopy vic;
{
    vic.srcSubresource = visl;
    vic.srcOffset = vo3;
    vic.dstSubresource = visl;
    vic.dstOffset = vo3;
    vic.extent = ve3;
}

VkImageMemoryBarrier vimb;
{
    vimb.sType = VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER;
    vimb.pNext = nullptr;
    vimb.oldLayout = VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL;
    vimb.newLayout = VK_IMAGE_LAYOUT_PRESENT_SRC_KHR;
    vimb.srcQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
    vimb.dstQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
    vimb.image = finalImage;
    vimb.srcAccessMask = 0;
    vimb.dstAccessMask = VK_ACCESS_PRESENT_BIT;
    vimb.subresourceRange = visr;
}

vkCmdPipelineBarrier(TextureCommandBuffer, VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT, VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT, 0, 0, (VkMemoryBarrier *)nullptr, 0, (VkBufferMemoryBarrier *)nullptr, 1, IN &vimb);

// now do the final image transfer:
// transition the texture buffer layout a second time:

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

VkImageMemoryBarrier vimb;
vimb.sType = VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER;
vimb.pNext = nullptr;
vimb.oldLayout = VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL;

vimb.newLayout = VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL;

vimb.srcQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
vimb.dstQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;

vimb.image = textureImage;

vimb.srcAccessMask = 0;

vimb.dstAccessMask = VK_ACCESS_SHADER_READ_BIT;

vimb.subresourceRange = visr;

vkCmdPipelineBarrier(TextureCommandBuffer,
VK_PIPELINE_STAGE_TRANSFER_BIT, VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT, 0,

0, (VkMemoryBarrier *)nullptr,
0, (VkBufferMemoryBarrier *)nullptr,
1, IN &vimb);

result = vkEndCommandBuffer(TextureCommandBuffer);

VkSubmitInfo vsi;
vsi.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;
vsi.pNext = nullptr;

vsi.commandBufferCount = 1;
vsi.pCommandBuffers = &TextureCommandBuffer;

vsi.waitSemaphoreCount = 0;
vsi.pWaitSemaphores = (VkSemaphore *)nullptr;

vsi.signalSemaphoreCount = 0;

vsi.pSignalSemaphores = (VkSemaphore *)nullptr;

vsi.pWaitDstStageMask = (VkPipelineStageFlags *)nullptr;

result = vkQueueSubmit(Queue, 1, IN &vsi, VK_NULL_HANDLE);

result = vkQueueWaitIdle(Queue);

// create an image view for the texture image:

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

VkImageMemoryBarrier vimb;
vimb.sType = VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER;

vimb.pNext = nullptr;

vimb.oldLayout = VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL;

vimb.newLayout = VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL;

vimb.srcQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;

vimb.dstQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;

vimb.image = textureImage;

vimb.srcAccessMask = 0;

vimb.dstAccessMask = VK_ACCESS_SHADER_READ_BIT;

vimb.subresourceRange = visr;

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

return result;

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

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 Image Magick’s convert, Adobe Photoshop, or GNU’s GIMP.

Anisotropic Texture Filtering

https://en.wikipedia.org/wiki/Anisotropic_filtering