Textures

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.

The mapping between the geometry of the 3D object and the S and T of the texture image works like this:

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:

```glBegin( GL_POLYGON );
glTexCoord2f( s0, t0 );
glNormal3f( nx0, ny0, nz0 );
glVertex3f( x0, y0, z0 );
glTexCoord2f( s1, t1 );
glNormal3f( nx1, ny1, nz1 );
glVertex3f( x1, y1, z1 );
. . .
glEnd( );
```

Disable texture mapping:

```glDisable( 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,

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

Or, for a sphere,

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

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 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 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
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 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: multîm in parvo, "many things in a small place"
#ifdef CHOICES
VkResult VK_IMAGE_LAYOUT_UNDEFINED
#endif
{
    Init07TextureBuffer(INOUT MyTexture * pMyTexture)
    
    VkResult result;
    vici.pQueueFamilyIndices = (const uint32_t *)nullptr;
    
    uint32_t texWidth = pMyTexture->width;
    unsigned char *texture = pMyTexture->pixels;
    VkDeviceSize textureSize = texWidth * pMyTexture->height * 4;  // rgba, 1 byte each
    
    VkImage stagingImage;
    VkImageCreateInfo vici;
    vici.sType = VK_STRUCTURE_TYPE_IMAGE_CREATE_INFO;
    vici.pNext = nullptr;
    
    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_LINEAR;
    vici.usage = VK_IMAGE_USAGE_TRANSFER_SRC_BIT;
    vici.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
    
    // this first {...} is to create the staging image:
    VkImageCreateInfo vici;
    vici.sType = VK_STRUCTURE_TYPE_IMAGE_CREATE_INFO;
    vici.pNext = nullptr;
    
    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_LINEAR;
    vici.usage = VK_IMAGE_USAGE_TRANSFER_SRC_BIT;
    vici.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
    
    VkImage stagingImage;
    VkImageCreateInfo vici;
    vici.sType = VK_STRUCTURE_TYPE_IMAGE_CREATE_INFO;
    vici.pNext = nullptr;
    
    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_LINEAR;
    vici.usage = VK_IMAGE_USAGE_TRANSFER_SRC_BIT |
    #ifdef CHOICES
    VK_IMAGE_USAGE_TRANSFER_DST_BIT | VK_IMAGE_USAGE_SAMPLED_BIT
    #endif
    
    VkDeviceMemory vdm;
    VkMemoryAllocateInfo vmai;
    vmai.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
    vmai.pNext = nullptr;
    vmai.allocationSize = vmr.size;
    vmai.memoryTypeIndex = FindMemoryThatIsHostVisible();
    result = vkAllocateMemory(LogicalDevice, IN &vmai, PALLOCATOR, OUT &vdm);
    
    // this second {...} is to create the actual texture image:
    VkImageCreateInfo vici;
    vici.sType = VK_STRUCTURE_TYPE_IMAGE_CREATE_INFO;
    vici.pNext = nullptr;
    
    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;
    
    VkImage textureImage;
    VkImageCreateInfo vici;
    vici.sType = VK_STRUCTURE_TYPE_IMAGE_CREATE_INFO;
    vici.pNext = nullptr;
    
    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_LINEAR;
    vici.usage = VK_IMAGE_USAGE_TRANSFER_DST_BIT | VK_IMAGE_USAGE_SAMPLED_BIT;
    vici.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
    
    VkImage stagingImage;
    VkImageCreateInfo vici;
    vici.sType = VK_STRUCTURE_TYPE_IMAGE_CREATE_INFO;
    vici.pNext = nullptr;
    
    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_LINEAR;
    vici.usage = VK_IMAGE_USAGE_TRANSFER_SRC_BIT;
    vici.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
    
    VkImage stagingImage;
    VkImageCreateInfo vici;
    vici.sType = VK_STRUCTURE_TYPE_IMAGE_CREATE_INFO;
    vici.pNext = nullptr;
    
    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_LINEAR;
    vici.usage = VK_IMAGE_USAGE_TRANSFER_SRC_BIT;
    vici.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
    
    VkImage stagingImage;
    VkImageCreateInfo vici;
    vici.sType = VK_STRUCTURE_TYPE_IMAGE_CREATE_INFO;
    vici.pNext = nullptr;
    
    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_LINEAR;
    vici.usage = VK_IMAGE_USAGE_TRANSFER_SRC_BIT;
    vici.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
// copy pixels from the staging image to the texture:

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;

result = vkBeginCommandBuffer(TextureCommandBuffer, IN &vcbbi);

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

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

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 = ... = stagingImage;
vimb.srcAccessMask = VK_ACCESS_HOST_WRITE_BIT;
vimb.dstAccessMask = 0;
vimb.subresourceRange = visr;

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

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

// transition the texture buffer layout:// *******************************************************************************

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

vimb.oldLayout = VK_IMAGE_LAYOUT_PREINITIALIZED;

vimb.newLayout = VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL;

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

// now do the final image transfer:

VkImageSubresourceLayers visl;
visl.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
visl.baseArrayLayer = 0;
visl.mipLevel = 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;

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

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

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

// transition the texture buffer layout a second time:// *******************************************************************************

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

vimb.oldLayout = VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL;

vimb.newLayout = VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL;

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 = ... = 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:
// (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, &vivci, allocator, &pMyTexture->texImageView);
return result;

Note that, at this point, the Staging Buffer is 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 ImageMagick’s convert, Adobe Photoshop, or GNU’s GIMP.