Textures

Mike Bailey
mjb@cs.oregonstate.edu

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.

1. 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.

2. 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(a0, b0);
glNormal3f(nx0, ny0, nz0);
glVertex3f(x0, y0, z0);
glTexCoord2f(a1, b1);
glNormal3f(nx1, ny1, nz1);
glVertex3f(x1, y1, z1);
...```  

Disable texture mapping:

```glDisable(GL_TEXTURE_2D);```  

3. 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, this is:

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

Or, for a sphere,

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

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

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

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

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

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

Memory Types

<table>
<thead>
<tr>
<th>CPU Memory</th>
<th>GPU Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Host Visible GPU Memory</td>
<td>Device Local GPU Memory</td>
</tr>
<tr>
<td>memcpy()</td>
<td>veCUDAStorage()</td>
</tr>
<tr>
<td>Deluxe Sampling Hardware</td>
<td>RGBA to the Shader</td>
</tr>
</tbody>
</table>

NVIDIA A6000 Graphics:

- Memory Types:
  - Memory 0:
    - Memory 1: DeviceLocal
    - Memory 2: HostVisible HostCoherent
    - Memory 3: HostVisible HostCoherent HostCached
    - Memory 4: DeviceLocal HostVisible HostCoherent
    - Memory 5: DeviceLocal

Intel Integrated Graphics:

- Memory Types:
  - Memory 0:
    - Memory 1: DeviceLocal
    - Memory 2: DeviceLocal HostVisible HostCoherent
    - Memory 3: 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 left over 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 copies of the same texture so that the red texel gets included somehow in all resolution-level textures.

Average 4 pixels to make a new one

Total texture storage is ~ 2x what it was without 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 Texture:Pixel ratio and one more, and then blend the two RGBA's returned. This is known as VK_SAMPLER_ADDRESS_MODE_LINEAR.
```c
// transition the texture buffer layout:

VkMemoryAllocateInfo vmai;

if( Verbose )
    vkGetImageMemoryRequirements
    result =
		if (Verbose)
		vkGetImageSubresourceLayout
		result =
			if (Verbose)

VkImageSubresourceLayers visl;

vkCmdPipelineBarrier

vkAllocateMemory

vkCreateImage

ve3.depth = 1;
ve3.height = texHeight;
ve3.width = texWidth;
vo3.z = 0;
vo3.x = 0;
visl.mipLevel = 0;
visl.baseArrayLayer = 0;
visr.layerCount = 1;
visr.levelCount = 1;
visr.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;

vici.initialLayout = VK_IMAGE_LAYOUT_PREINITIALIZED;
vici.usage = VK_IMAGE_USAGE_TRANSFER_DST_BIT | VK_IMAGE_USAGE_SAMPLED_BIT;
vici.tiling = VK_IMAGE_TILING_OPTIMAL;
vici.samples = VK_SAMPLE_COUNT_1_BIT;
vici.arrayLayers = 1;
vici.mipLevels = 1;
vici.extent.height = texHeight;
vici.extent.width = texWidth;
vici.format = VK_FORMAT_R8G8B8A8_UNORM;

vici.sType = VK_STRUCTURE_TYPE_IMAGE_CREATE_INFO;

ffffffff( FpDebug, "Texture vmr.memoryTypeBits = 0x%08x\n", vmr.memoryTypeBits );

fprintf( FpDebug, "Texture vmr.alignment = %lld\n", vmr.alignment );

vici.pQueueFamilyIndices = (const uint32_t *)nullptr;

vkAllocateMemory

vkCreateImage(LogicalDevice

ve3.depth = 1;
ve3.height = texHeight;
ve3.width = texWidth;
vo3.z = 0;
vo3.x = 0;
visl.mipLevel = 0;
visl.baseArrayLayer = 0;
visr.layerCount = 1;
visr.levelCount = 1;
visr.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;

vici.initialLayout = VK_IMAGE_LAYOUT_PREINITIALIZED;
vici.usage = VK_IMAGE_USAGE_TRANSFER_DST_BIT | VK_IMAGE_USAGE_SAMPLED_BIT;
vici.tiling = VK_IMAGE_TILING_OPTIMAL;
vici.samples = VK_SAMPLE_COUNT_1_BIT;
vici.arrayLayers = 1;
vici.mipLevels = 1;
vici.extent.height = texHeight;
vici.extent.width = texWidth;
vici.format = VK_FORMAT_R8G8B8A8_UNORM;

vici.sType = VK_STRUCTURE_TYPE_IMAGE_CREATE_INFO;

ffffffff( FpDebug, "Image vmr.memoryTypeBits = 0x%08x\n", vmr.memoryTypeBits );

vici.pQueueFamilyIndices = (const uint32_t *)nullptr;

vkAllocateMemory

vkCreateImage

ve3.depth = 1;
ve3.height = texHeight;
ve3.width = texWidth;
vo3.z = 0;
vo3.x = 0;
visl.mipLevel = 0;
visl.baseArrayLayer = 0;
visr.layerCount = 1;
visr.levelCount = 1;
visr.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;

void * gpuMemory;

memcpy(gpuMemory, (void *)texture, (size_t)textureSize);

vsl = visl;

vimb.subresourceRange = visr;

vimb.srcAccessMask = 0;

vimb.dstQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;

vimb.srcQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;

vimb.newLayout = VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL;

vimb.oldLayout = VK_IMAGE_LAYOUT_PREINITIALIZED;

vimb.pNext = nullptr;

vimb.sType = VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER;

visr.layerCount = 1;

visr.baseArrayLayer = 0;

visr.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;

vkCmdPipelineBarrier

vkBeginCommandBuffer

vkCmdPipelineBarrier

vkEndCommandBuffer

vkQueueSubmit

vkQueueWaitFences

vkFreeMemory

vkDestroyImage

vkDestroyMemory

vkDeviceWaitIdle

vkDestroyImageView

vkDestroySurfaceKHR

vkDestroyDevice

vkDestroyInstance
```
// *******************************************************************************
// 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);
}

Remember to 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
The Actual
Image Data
8 bits Red
8 bits Green
8 bits Blue
8 bits Alpha

The 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.