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. 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.

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

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

Draw your polygons, specifying s and t at each vertex:
```
gBegin(GL_POLYGON);
  glTexCoord2f( s0, t0 );
  glNormal3f( nx0, ny0, nz0 );
  glVertex3f( x0, y0, z0 );
  glTexCoord2f( s1, t1 );
  glNormal3f( nx1, ny1, nz1 );
  glVertex3f( x1, y1, z1 );
...```

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

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

You specify an (s,t) pair at each vertex, along with the vertex coordinate. At the same time that the rasterizer is interpolating the coordinates, colors, etc. inside the polygon, it is also interpolating the (s,t) coordinates. Then, when it goes to draw each pixel, it uses that pixel's interpolated (s,t) to look-up a color in the texture image.
Triangles in an Array of Structures

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

struct vertex VertexData[3] =
{
    // triangle 0-2-3:
    // vertex #0:
    // (x,y,z) (0,0,-1) (0,0,0) (1,0)
    { -1., -1., -1. },
    {  0.,  0., -1. },
    {  0.,  0.,  0. },
    {  1.,  0. },

    // vertex #2:
    // (x,y,z) (0,0,-1) (0,1,0) (1,1)
    { -1.,  1., -1. },
    {  0.,  0., -1. },
    {  0.,  1.,  0. },
    {  1.,  1. },

    // vertex #3:
    // (x,y,z) (0,0,-1) (1,1,0) (0,1)
    {  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,

\[
\begin{align*}
    s &= \frac{x - X_{\text{min}}}{X_{\text{max}} - X_{\text{min}}} \\
    t &= \frac{y - Y_{\text{min}}}{Y_{\text{max}} - Y_{\text{min}}}
\end{align*}
\]

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

Or, for a sphere,

\[
\begin{align*}
    s &= \frac{\Phi - (-\pi)}{\pi} \\
    t &= \frac{\theta - (-\pi/2)}{\pi/2}
\end{align*}
\]

From the Sphere code:

```c
    s = ( lng + M_PI ) / ( 2.*M_PI );
    t = ( lat + M_PI/2. ) / M_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.

```
VkDescriptorSetLayoutBinding TexSamplerSet[1];
TexSamplerSet[1].binding = 0;
TexSamplerSet[1].descriptorType = VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER;
// uniform sampler2D uSampler
// vec4 rgba = texture( uSampler, vST );
TexSamplerSet[1].descriptorCount = 1;
TexSamplerSet[1].stageFlags = VK_SHADER_STAGE_FRAGMENT_BIT;
TexSamplerSet[1].pImmutableSamplers = ( VkSampler * )nullptr;

VkDescriptorImageInfo vdi0;
vdi0.sampler = MyPuppyTexture.texSampler;
vdi0.imageView = MyPuppyTexture.texImageView;
vdi0.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 = &vdi0;
vwds3.pTexelBufferView = ( VkBufferView * )nullptr;
```

Memory Types:

CPU Memory | GPU Memory
---|---
Host Visible | GPU Memory
Local GPU Memory | Device Local GPU Memory

memcpy

Texture RGBA Data Values

Combined Image Sampler

Texture Sampler

Texture Sampler

RGBA to the Shader

Texture Sampling Hardware

memcpy

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

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

OpenGL

```
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 );
```

Vulkan

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 InitTextureSampler(MyTexture * pMyTexture)
{
  VkResult result;

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

  VkSamplerCreateInfo vsci;
  vsci.sType = VK_STRUCTURE_TYPE_SAMPLER_CREATE_INFO;
  vsci.pNext = nullptr;

  VkImage stagingImage;

  vsci.addressModeU = VK_SAMPLER_ADDRESS_MODE_REPEAT;
  vsci.addressModeV = VK_SAMPLER_ADDRESS_MODE_REPEAT;
  vsci.addressModeW = VK_SAMPLER_ADDRESS_MODE_REPEAT;

  VkImageCreateInfo vici;
  vici.sType = VK_STRUCTURE_TYPE_IMAGE_CREATE_INFO;
  vici.pNext = nullptr;
  vici.flags = 0;

  vsci.mipLodBias = 0.0;
  vsci.anisotropyEnable = VK_FALSE;
  vsci.compareEnable = VK_FALSE;
  vsci.compareOp = VK_COMPARE_OP_NEVER;

  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;

  // ************************************************************ *******************
  // this first {...} is to create the staging image:
  // VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_BORDER
  // VK_SAMPLER_ADDRESS_MODE_MIRROR_CLAMP_TO_EDGE
  // VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE

  vici.sharingMode = VK_SHARING_MODE_EXCLUSIVE;

  vsci.minLod = 0.0;
  vsci.borderColor = VK_BORDER_COLOR_FLOAT_OPAQUE_BLACK;

  #ifdef CHOICES
  vici.pQueueFamilyIndices = (const uint32_t *)nullptr;
  result = vkCreateSampler(LogicalDevice, IN &vsci, PALLOCATOR, OUT &pMyTexture->texSampler);
  // VK_BORDER_COLOR_FLOAT_TRANSPARENT_BLACK
  // VK_BORDER_COLOR_INT_TRANSPARENT_BLACK
  // VK_BORDER_COLOR_FLOAT_OPAQUE_BLACK
  // VK_BORDER_COLOR_INT_OPAQUE_BLACK
  // VK_BORDER_COLOR_FLOAT_OPAQUE_WHITE
  // VK_BORDER_COLOR_INT_OPAQUE_WHITE
  #endif

  // VK_TRUE means we are use raw texels as the index
  // VK_FALSE means we are using the usual 0. - 1.

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

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

  if (Verbose)
  {
    fprintf(FpDebug, "Image vmr.size = %lld
", vmr.size);
    fprintf(FpDebug, "Image vmr.alignment = %lld
", vmr.alignment);
    if (vsl.rowPitch == 4 * texWidth)
    {
      fprintf(FpDebug, "Image vmr.memoryTypeBits = 0x%08x
", vmr.memoryTypeBits);
      fflush(FpDebug);
    }
    else {
      void * gpuMemory;
      VkDeviceMemory vdm;
      result = vkAllocateMemory(LogicalDevice, IN &vdm, PALLOCATOR, OUT &pMyTexture->vdm);
      result = vkBindImageMemory(LogicalDevice, IN stagingImage, IN vdm, 0);  // 0 = offset
      // we have now created the staging image -- fill it with the pixel data:
      VkImageSubresource vis;
      vis.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
      vis.mipLevel = 0;
      vis.arrayLayer = 0;

      VkSubresourceLayout vsl;
      vkGetImageSubresourceLayout(LogicalDevice, stagingImage, vis, &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);
      }
    }
    else {
      unsigned char * gpuBytes = (unsigned char *)gpuMemory;
      for (unsigned int y = 0; y < texHeight; y++)
      {
        VkMemoryAllocateInfo vmai;
        vmai.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
        vmai.pNext = nullptr;
        memcpy(&gpuBytes[y * vsl.rowPitch], &texture[4 * y * texWidth], (size_t)(4*texWidth) );
      }
    }
  } else {
    void * gpuMemory;
    VkDeviceMemory vdm;
    result = vkAllocateMemory(LogicalDevice, IN &vdm, PALLOCATOR, OUT &pMyTexture->vdm);
    result = vkBindImageMemory(LogicalDevice, IN stagingImage, IN vdm, 0);  // 0 = offset
    // we have now created the staging image -- fill it with the pixel data:
    VkImageSubresource vis;
    vis.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
    vis.mipLevel = 0;
    vis.arrayLayer = 0;

    VkSubresourceLayout vsl;
    vkGetImageSubresourceLayout(LogicalDevice, stagingImage, vis, &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);
    }
  }
  // FindMemoryThatIsHostVisible();   // because we want to mmap it
}

if (Verbose)
{
  fprintf(FpDebug, "vkGetImageMemoryRequirements(LogicalDevice, stagingImage, &vmr);
");
  fprintf(FpDebug, "if (Verbose)
");
  fprintf(FpDebug, "{fprintf(FpDebug, "Image vmr.size = %lld
", vmr.size);
");
  fprintf(FpDebug, "Image vmr.alignment = %lld
", vmr.alignment);
  if (vsl.rowPitch == 4 * texWidth)
  {
    fprintf(FpDebug, "Image vmr.memoryTypeBits = 0x%08x
", vmr.memoryTypeBits);
    fflush(FpDebug);
  }
  else {
    void * gpuMemory;
    VkDeviceMemory vdm;
    result = vkAllocateMemory(LogicalDevice, IN &vdm, PALLOCATOR, OUT &pMyTexture->vdm);
    result = vkBindImageMemory(LogicalDevice, IN stagingImage, IN vdm, 0);  // 0 = offset
    // we have now created the staging image -- fill it with the pixel data:
    VkImageSubresource vis;
    vis.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
    vis.mipLevel = 0;
    vis.arrayLayer = 0;

    VkSubresourceLayout vsl;
    vkGetImageSubresourceLayout(LogicalDevice, stagingImage, vis, &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);
    }
  } else {
    unsigned char * gpuBytes = (unsigned char *)gpuMemory;
    for (unsigned int y = 0; y < texHeight; y++)
    {
      VkMemoryAllocateInfo vmai;
      vmai.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
      vmai.pNext = nullptr;
      memcpy(&gpuBytes[y * vsl.rowPitch], &texture[4 * y * texWidth], (size_t)(4*texWidth) );
    }
  }
}
// ******************************************************************************
// copy pixels from the staging image to the texture:
// this second {...} is to create the actual texture image:
// ******************************************************************************
VkCommandBufferBeginInfo vcbbi = {VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO, nullptr, VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT, nullptr};
VkImageCreateInfo vici = {VK_STRUCTURE_TYPE_IMAGE_CREATE_INFO, nullptr, 0, VK_IMAGE_TYPE_2D, VK_FORMAT_R8G8B8A8_UNORM, texWidth, texHeight, 1, 1, 1, VK_SAMPLE_COUNT_1_BIT, VK_IMAGE_TILING_OPTIMAL, VK_IMAGE_USAGE_TRANSFER_DST_BIT | VK_IMAGE_USAGE_SAMPLED_BIT, VK_SHARING_MODE_EXCLUSIVE, &visr, 0, nullptr};
result = vkCreateImage(LogicalDevice, &vici, PALLOCATOR, &textureImage); // allocated, but not filled

VkImageMemoryBarrier vimb = {VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER, nullptr, VK_IMAGE_LAYOUT_PREINITIALIZED, VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL, 0, (VkMemoryBarrier *)nullptr, 1, IN &vimb};

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

VkImageCopy vic = {0, 0, 0, texWidth, texHeight, 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:
VkImageSubresourceLayers visl = {0, 0, texWidth, texHeight, 1};

// transitions the staging buffer layout:// ******************************************************************************
// because we are transferring into it and will eventual sample from it

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

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

```c
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);
```