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 Basic Idea

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

\[(X_0, Y_0, S_0, T_0)\]
\[(X_2, Y_2, S_2, T_2)\]
\[(X_3, Y_3, S_3, T_3)\]
\[(X_4, Y_4, S_4, T_4)\]
\[(X_0, Y_1, S_1, T_1)\]

Interpolated \((S, T) = (0.78, 0.67)\)

\((199.68, 171.52)\) in texels

Enable texture mapping:

```gl
glEnable( GL_TEXTURE_2D );
```

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

```gl
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 );

```

Enable texture mapping:

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

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

struct vertex VertexData[4] = {
    // 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,

\[
\begin{align*}
    s &= \frac{x - \text{Xmin}}{\text{Xmax} - \text{Xmin}} \\
    t &= \frac{y - \text{Ymin}}{\text{Ymax} - \text{Ymin}}
\end{align*}
\]
Using a Texture: How do you know what (s, t) to assign to each vertex?

Or, for a sphere,

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

Distorted Texture:

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

CPU Memory

GPU Memory

Host Visible GPU Memory

Device Local GPU Memory

memcpy() vkCmdCopyImage()

Texture Sampling Hardware

RGBA to the Shader

NVIDIA A6000 Graphics:

6 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:

3 Memory Types:
Memory 0: DeviceLocal
Memory 1: DeviceLocal HostVisible HostCoherent
Memory 2: DeviceLocal HostVisible HostCoherent HostCached
Something I’ve Found Useful

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;

// 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;
```

Texture Sampling Parameters

```c
// OpenGL
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 );
```

```c
// Vulkan
result = vkCreateSampler( LogicalDevice, IN &vsci, PALLOCATOR, OUT &MyPuppyTexture->texSampler);
```
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 Texture:Pixel 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”

mjb – December 29, 2022
VkResult Init07TextureSampler( MyTexture * pMyTexture )
{
VkResult result;
VkSamplerCreateInfo vsci;
    vsci.sType = VK_STRUCTURE_TYPE_SAMPLER_CREATE_INFO;
    vsci.pNext = nullptr;
    vsci.flags = 0;
    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;
    #ifdef CHOICES
        VK_SAMPLER_ADDRESS_MODE_REPEAT
        VK_SAMPLER_ADDRESS_MODE_MIRRORED_REPEAT
        VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE
        VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_BORDER
        VK_SAMPLER_ADDRESS_MODE_MIRROR_CLAMP_TO_EDGE
    #endif
    vsci.mipLodBias = 0.;
    vsci.anisotropyEnable = VK_FALSE;
    vsci.maxAnisotropy = 1.;
    vsci.compareEnable = VK_FALSE;
    vsci.compareOp = VK_COMPARE_OP_NEVER;
    #ifdef CHOICES
        VK_COMPARE_OP_NEVER
        VK_COMPARE_OP_LESS
        VK_COMPARE_OP_EQUAL
        VK_COMPARE_OP_LESS_OR_EQUAL
        VK_COMPARE_OP_GREATER
        VK_COMPARE_OP_NOT_EQUAL
        VK_COMPARE_OP_GREATER_OR_EQUAL
        VK_COMPARE_OP_ALWAYS
    #endif
    vsci.minLod = 0.;
    vsci.maxLod = 0.;
    vsci.borderColor = VK_BORDER_COLOR_FLOAT_OPAQUE_BLACK;
    #ifdef CHOICES
        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
    vsci.unnormalizedCoordinates = VK_FALSE;
        // VK_TRUE means we are using raw texels as the index
        // VK_FALSE means we are using the usual 0. - 1.
    result = vkCreateSampler( LogicalDevice, IN &vsci, PALLOCATOR, OUT &MyPuppyTexture->texSampler );
}

VkResult Init07TextureBuffer(INOUT 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
VkImage stagingImage;
VkImage textureImage;
    // *******************************************************************************
        // this first {...} is to create the staging image:
    // *******************************************************************************
    {
        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.usage = VK_IMAGE_USAGE_TRANSFER_SRC_BIT;
            vici.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
    #ifdef CHOICES
        VK_IMAGE_TILING_OPTIMAL
        VK_IMAGE_TILING_LINEAR
    #endif
    result = vkCreateImage( LogicalDevice, IN &vici, PALLOCATOR, OUT &stagingImage );
}
```c
#ifdef CHOICES
VK_IMAGE_LAYOUT_UNDEFINED
VK_IMAGE_LAYOUT_PREINITIALIZED
#endif

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

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\n", vmr.size);
    fprintf(FpDebug, "Image vmr.alignment = %lld\n", vmr.alignment);
    fprintf(FpDebug, "Image vmr.memoryTypeBits = 0x%08x\n", vmr.memoryTypeBits);
    fflush(FpDebug);
}

VkMemoryAllocateInfo vmai;
vmai.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
vmai.pNext = nullptr;
vmai.allocationSize = vmr.size;
vmai.memoryTypeIndex = FindMemoryThatIsHostVisible(); // because we want to mmap it

VkDeviceMemory vdm;
result = vkAllocateMemory(LogicalDevice, IN &vmai, PALLOCATOR, OUT &vdm);
pMyTexture->vdm = 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, IN &vis, OUT &vsl);

if (Verbose)
{
    fprintf(FpDebug, "Subresource Layout:\n");
    fprintf(FpDebug, "	offset = %lld\n", vsl.offset);
    fprintf(FpDebug, "	size = %lld\n", vsl.size);
    fprintf(FpDebug, "	rowPitch = %lld\n", vsl.rowPitch);
    fprintf(FpDebug, "	arrayPitch = %lld\n", vsl.arrayPitch);
    fprintf(FpDebug, "	depthPitch = %lld\n", vsl.depthPitch);
    fflush(FpDebug);
}

void * gpuMemory;
vkMapMemory(LogicalDevice, vdm, 0, VK_WHOLE_SIZE, 0, OUT &gpuMemory);
// 0 and 0 = offset and memory map flags

if (vsl.rowPitch == 4 * texWidth)
{
    memcpy(gpuMemory, (void *)texture, (size_t)textureSize);
}
else
{
    unsigned char *gpuBytes = (unsigned char *)gpuMemory;
    for (unsigned int y = 0; y < texHeight; y++)
    {
        memcpy(&gpuBytes[y * vsl.rowPitch], &texture[4 * y * texWidth], (size_t)(4*texWidth));
    }
}

vkUnmapMemory(LogicalDevice, vdm);

```
// this second {...} is to create the actual texture image:
{
    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;
    // because we are transferring into it and will eventually sample from it
    vici.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
    vici.initialLayout = VK_IMAGE_LAYOUT_PREINITIALIZED;
    vici.queueFamilyIndexCount = 0;
    vici.pQueueFamilyIndices = (const uint32_t *)nullptr;

    result = vkCreateImage(LogicalDevice, IN &vici, PALLOCATOR, OUT &textureImage); // allocated, but not filled
    VkMemoryRequirements vmr;
    vkGetImageMemoryRequirements(LogicalDevice, IN textureImage, OUT &vmr);
    if(Verbose)
    {
        fprintf(FpDebug, "Texture vmr.size = %lld\n", vmr.size);
        fprintf(FpDebug, "Texture vmr.alignment = %lld\n", vmr.alignment);
        fprintf(FpDebug, "Texture vmr.memoryTypeBits = 0x%08x\n", vmr.memoryTypeBits);
        fflush(FpDebug);
    }
    VkMemoryAllocateInfo vmai;
    vmai.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
    vmai.pNext = nullptr;
    vmai.allocationSize = vmr.size;
    vmai.memoryTypeIndex = FindMemoryThatIsDeviceLocal();  // because we want to sample from it
    VkDeviceMemory vdm;
    result = vkAllocateMemory(LogicalDevice, IN &vmai, PALLOCATOR, OUT &vdm);
    result = vkBindImageMemory(LogicalDevice, IN textureImage, IN vdm, 0);        // 0 = offset
}
// ******************************************************************************

// 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);

    VkImageSubresourceRange vsrcra;
    vsrcra.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
    vsrcra.baseMipLevel = 0;
    vsrcra.levelCount = 1;
    vsrcra.baseArrayLayer = 0;
    vsrcra.layerCount = 1;
    VkImageMemoryBarrier vimb;
    vimb.sType = VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER;
    vimb.pNext = nullptr;
    vimb.srcQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
    vimb.dstQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
    vimb.image = stagingImage;
    vimb.srcAccessMask = VK_ACCESS_HOST_WRITE_BIT;
    vimb.dstAccessMask = 0;
    vimb.subresourceRange = vsrcra;
    vkCmdPipelineBarrier(TextureCommandBuffer, VK_PIPELINE_STAGE_HOST_BIT, VK_PIPELINE_STAGE_HOST_BIT, 0, 0, (VkMemoryBarrier *)nullptr, 0, (VkBufferMemoryBarrier *)nullptr, 1, IN &vimb);

    VkImageSubresourceRange vtrgra;
    vtrgra.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
    vtrgra.baseMipLevel = 0;
    vtrgra.levelCount = 1;
    vtrgra.baseArrayLayer = 0;
    vtrgra.layerCount = 1;
    VkImageMemoryBarrier vimb;
    vimb.sType = VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER;
    vimb.pNext = nullptr;
    vimb.srcQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
    vimb.dstQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
    vimb.image = textureImage;
    vimb.srcAccessMask = VK_ACCESS_HOST_WRITE_BIT;
    vimb.dstAccessMask = 0;
    vimb.subresourceRange = vtrgра;
    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:

// *******************************************************************************
// transition the texture 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 = 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 = VK_ACCESS_TRANSFER_WRITE_BIT;
    vimb.subresourceRange = visr;

    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:

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, &vimb);

// 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;
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, PALLOCATOR, &pMyTexture->texImageView);
return result;

Note that, at this point, the Staging Buffer is no longer needed, and can be destroyed.
typedef struct MyTexture
{
  uint32_t                        width;
  uint32_t                        height;
  VkImage                        texImage;
  VkImageView                    texImageView;
  VkSampler                      texSampler;
  VkDeviceMemory                 vdm;
} MyTexture;

MyTexture MyPuppyTexture;

result = Init06TextureBufferAndFillFromBmpFile ("puppy1.bmp", &MyPuppyTexture);
Init06TextureSampler( &MyPuppyTexture.texSampler );

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.

Oregon State University
Computer Graphics