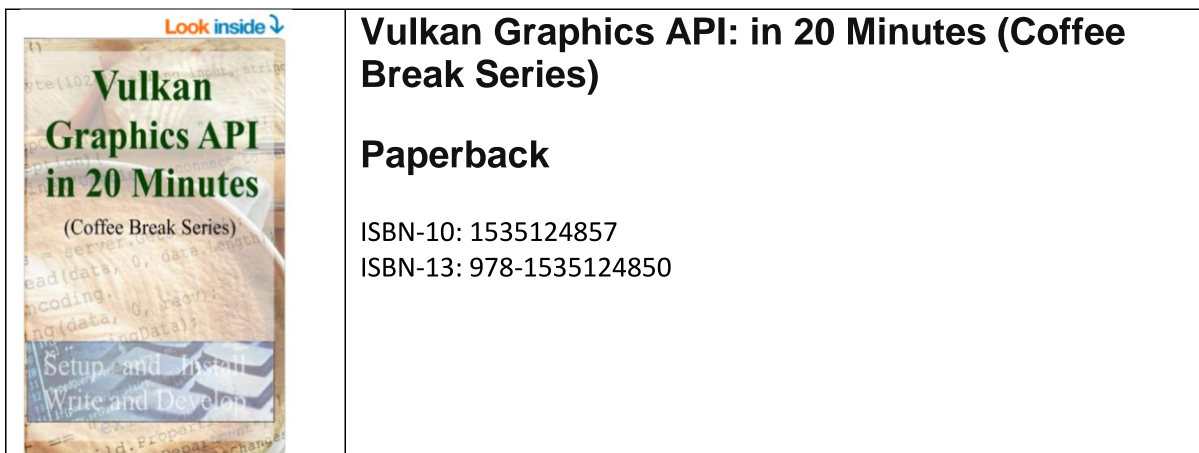
Vulkan API

Extension Chapter 02

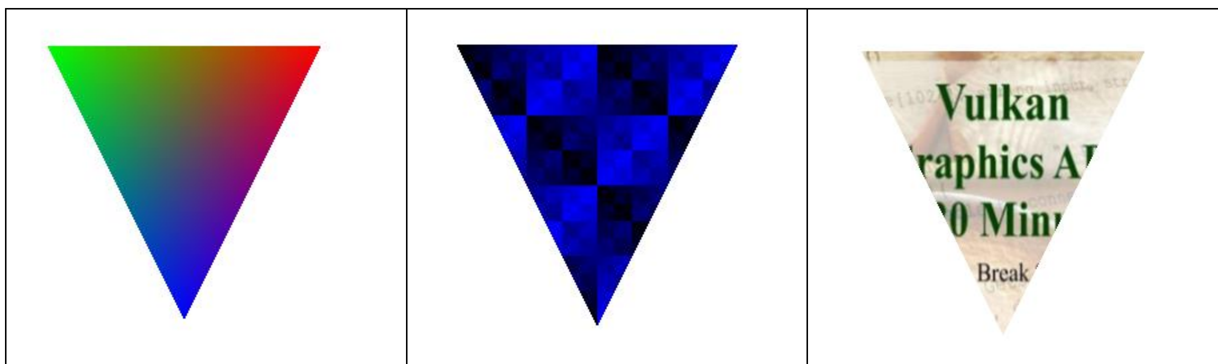
These few articles are tutorial extensions for the reader to work through to get familiar with the API. Prior to this tutorial, you should've setup a basic skeleton implementation (i.e., the renderings of a basic coloured triangle). You're now ready to start modifying the implementation to include additional features, for example, in this extension chapter, you'll add textured geometry. That is, you'll extend the basic triangle code to generate a pattern and map it onto the surface of the triangle. This will require you to update the vert/pixel shader, specify texture coordinates, generate/load an image, create a texture, and a texture sampler. This tutorial covers goes step by step through the necessary modifications you'll need to do, to add image mapping (texturing) onto your triangle – texturing is a crucial component (especially for complex shapes/worlds).



Why do you need textured triangles?

To cut straight to the point, you can't capture the geometry and detail down to the individual micro-points. It would be too expensive. Instead, you'll use textures to add additional detail to a scene – also you'll use textures to put text/logos/menus on screen (instead of drawing the characters as millions of triangles – you'll use a texture that maps onto triangles).

As shown below, you'll start from a simple coloured triangle and end up with a textured triangle (the code below shows how to either generate the test image or load an image from a bmp file).



First thing you need to do, is add some variables to store your texture and the sampler:

```
// Extension (Textured Triangle)
VkSampler             g_textureSampler        = NULL;
VkImageView           g_textureView            = NULL;
```

Here is the code that you need to add to create the test image pattern (or loads a bitmap .bmp image).

```
// Extension (Textured Triangle)
{
       // create our texture sfampler:
    VkSamplerCreateInfo samplerCreateInfo = {};
    samplerCreateInfo.sType                 = VK_STRUCTURE_TYPE_SAMPLER_CREATE_INFO;
    samplerCreateInfo.magFilter             = VK_FILTER_LINEAR;
    samplerCreateInfo.minFilter             = VK_FILTER_LINEAR;
    samplerCreateInfo.mipmapMode            = VK_SAMPLER_MIPMAP_MODE_LINEAR;
    samplerCreateInfo.addressModeU          = VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE;
    samplerCreateInfo.addressModeV          = VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE;
    samplerCreateInfo.addressModeW          = VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE;
    samplerCreateInfo.mipLodBias           = 0;
    samplerCreateInfo.anisotropyEnable = VK_FALSE;
    samplerCreateInfo.minLod               = 0;
    samplerCreateInfo.maxLod               = 5;
    samplerCreateInfo.borderColor          = VK_BORDER_COLOR_FLOAT_TRANSPARENT_BLACK;
    samplerCreateInfo.unnormalizedCoordinates = VK_FALSE;


    VkResult result =
       vkCreateSampler( g_device,
                        &samplerCreateInfo,
                                    NULL,
                                    &g_textureSampler );
    DBG_ASSERT_VULKAN_MSG( result,
              "Failed to create sampler." );

    // image data
    struct loaded_image
    {
        int   width;
        int   height;
        void *data;
    };

    // either load from a file or generate an image
    // we show an example code snippet that loads a bmp image file
    #if 1
    // load our bmp using win32:
    HBITMAP bmpHandle =
    (HBITMAP) LoadImage(NULL,
                        ".\\testimage.bmp",
                        IMAGE_BITMAP,
                        0, 0,       // use real image size
                        LR_LOADFROMFILE );
```

```
   DBG_ASSERT_MSG( bmpHandle, "Failed to load bmp file." );

BITMAP bitmap;
GetObject( bmpHandle, sizeof(bitmap), (void*)&bitmap );

// Only support 32bit bmps for testing
DBG_ASSERT(bitmap.bmBitsPixel==32);
// check for padding
DBG_ASSERT(bitmap.bmWidthBytes/4 == bitmap.bmWidth );

uint32_t buffSize = bitmap.bmWidth * bitmap.bmHeight * 4;
DWORD* bmpBits    = new DWORD[ buffSize ];
uint32_t bitsRead = GetBitmapBits( bmpHandle, buffSize, bmpBits );
DBG_ASSERT_MSG( bitsRead == buffSize, "Could not read bitmap bits" );

loaded_image testImage;
testImage.width   = bitmap.bmWidth;
testImage.height  = bitmap.bmHeight;
testImage.data    = (void *) new float[ bitmap.bmWidth * bitmap.bmHeight * 3 ];
for (int i=0; i<testImage.width*testImage.height; ++i)
{
   ((float*)testImage.data)[i*3+0] = (((unsigned char*)bmpBits))[i*4+2] / 255.0f;
   ((float*)testImage.data)[i*3+1] = (((unsigned char*)bmpBits))[i*4+1] / 255.0f;
   ((float*)testImage.data)[i*3+2] = (((unsigned char*)bmpBits))[i*4+0] / 255.0f;
}
delete[] bmpBits;
#endif


// if we want to generate our texture/image - simple
// test patter (xor)
#if 0
loaded_image testImage;
testImage.width  = 512;
testImage.height = 512;
testImage.data   = (void *) new float[ testImage.width * testImage.height * 3 ];


// generate a pretty test pattern
int xx = 0;
for (int y=0; y<testImage.height; ++y)
{
   for (int x=0; x<testImage.width; ++x)
   {
      int cc = (x^y)    &0xFF;
      int r  = (cc>>16)&0xFF;
      int g  = (cc>>8) &0xFF;
      int b  = (cc>>0) &0xFF ;

      // Our beautiful XOR pattern!!!
      float *pixel = ((float *) testImage.data) + ( x * testImage.height * 3 ) + (y * 3);
      pixel[0] = r/255.0f; // red
      pixel[1] = g/255.0f; // greed
      pixel[2] = b/255.0f; // blue
   }// End inner loop
}// End outer loop
#endif


VkImageCreateInfo textureCreateInfo = {};
textureCreateInfo.sType         = VK_STRUCTURE_TYPE_IMAGE_CREATE_INFO;
textureCreateInfo.imageType     = VK_IMAGE_TYPE_2D;
```

```
textureCreateInfo.format        = VK_FORMAT_R32G32B32_SFLOAT;
textureCreateInfo.extent.width  = testImage.width;
textureCreateInfo.extent.height = testImage.height;
textureCreateInfo.extent.depth  = 1;
textureCreateInfo.mipLevels     = 1;
textureCreateInfo.arrayLayers   = 1;
textureCreateInfo.samples       = VK_SAMPLE_COUNT_1_BIT;
textureCreateInfo.tiling        = VK_IMAGE_TILING_LINEAR;
textureCreateInfo.usage         = VK_IMAGE_USAGE_SAMPLED_BIT;
textureCreateInfo.sharingMode   = VK_SHARING_MODE_EXCLUSIVE;
textureCreateInfo.initialLayout = VK_IMAGE_LAYOUT_PREINITIALIZED;

VkImage textureImage;
result =
   vkCreateImage( g_device,
                  &textureCreateInfo,
                  NULL,
                  &textureImage );
DBG_ASSERT_VULKAN_MSG( result,
          "Failed to create texture image." );

// allocate and bind image memory:
VkMemoryRequirements textureMemoryRequirements = {};
vkGetImageMemoryRequirements( g_device,
                              textureImage,
                              &textureMemoryRequirements );

VkMemoryAllocateInfo textureImageAllocateInfo = {};
textureImageAllocateInfo.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
textureImageAllocateInfo.allocationSize = textureMemoryRequirements.size;

// read the device memory properties
VkPhysicalDeviceMemoryProperties memoryProperties;
vkGetPhysicalDeviceMemoryProperties( g_physicalDevice, &memoryProperties );

uint32_t textureMemoryTypeBits = textureMemoryRequirements.memoryTypeBits;
VkMemoryPropertyFlags tDesiredMemoryFlags = VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT;
for( uint32_t i = 0; i < 32; ++i )
   {
    VkMemoryType memoryType = memoryProperties.memoryTypes[i];
    if( textureMemoryTypeBits & 1 )
          {
        if( ( memoryType.propertyFlags & tDesiredMemoryFlags ) == tDesiredMemoryFlags )
                 {
            textureImageAllocateInfo.memoryTypeIndex = i;
            break;
        }
    }
    textureMemoryTypeBits = textureMemoryTypeBits >> 1;
}

VkDeviceMemory textureImageMemory = { 0 };
result = vkAllocateMemory( g_device,
                           &textureImageAllocateInfo,
                           NULL,
                           &textureImageMemory   );
DBG_ASSERT_VULKAN_MSG( result,
          "Failed to allocate device memory." );

result = vkBindImageMemory( g_device, textureImage, textureImageMemory, 0 );
DBG_ASSERT_VULKAN_MSG( result,
          "Failed to bind image memory." );
```

```
    // copy our image content:
    void *imageMapped;
    result = vkMapMemory( g_device, textureImageMemory, 0, VK_WHOLE_SIZE, 0, &imageMapped );
    DBG_ASSERT_VULKAN_MSG( result,
              "Failed to map uniform buffer memory." );

    memcpy( imageMapped,
            testImage.data,
            sizeof(float) * testImage.width * testImage.height * 3 );

    VkMappedMemoryRange memoryRange = {};
    memoryRange.sType  = VK_STRUCTURE_TYPE_MAPPED_MEMORY_RANGE;
    memoryRange.memory = textureImageMemory;
    memoryRange.offset = 0;
    memoryRange.size   = VK_WHOLE_SIZE;
    vkFlushMappedMemoryRanges( g_device, 1, &memoryRange );

    vkUnmapMemory( g_device, textureImageMemory );


    delete[] testImage.data;
    testImage.data = NULL;


    // before using the texture image must change
    // to VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL layout
    {
        VkCommandBufferBeginInfo beginInfo = {};
        beginInfo.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO;
        beginInfo.flags = VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT;

        vkBeginCommandBuffer( g_drawCmdBuffer, &beginInfo );

        VkImageMemoryBarrier layoutTransitionBarrier = {};
        layoutTransitionBarrier.sType                = VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER;
        layoutTransitionBarrier.srcAccessMask        = VK_ACCESS_HOST_WRITE_BIT;
        layoutTransitionBarrier.dstAccessMask        = VK_ACCESS_SHADER_READ_BIT;
        layoutTransitionBarrier.oldLayout            = VK_IMAGE_LAYOUT_PREINITIALIZED;
        layoutTransitionBarrier.newLayout            =
VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL;
        layoutTransitionBarrier.srcQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
        layoutTransitionBarrier.dstQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
        layoutTransitionBarrier.image                = textureImage;
        VkImageSubresourceRange resourceRange        = { VK_IMAGE_ASPECT_COLOR_BIT, 0, 1, 0, 1
};
        layoutTransitionBarrier.subresourceRange     = resourceRange;

        vkCmdPipelineBarrier(   g_drawCmdBuffer,
                                VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT,
                                VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT,
                                0,
                                0, NULL,
                                0, NULL,
                                1, &layoutTransitionBarrier );

        vkEndCommandBuffer( g_drawCmdBuffer );

        VkPipelineStageFlags waitStageMash[] = { VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT
};
        VkSubmitInfo submitInfo         = {};
        submitInfo.sType                = VK_STRUCTURE_TYPE_SUBMIT_INFO;
```

```
        submitInfo.waitSemaphoreCount   = 0;
        submitInfo.pWaitSemaphores      = NULL;
        submitInfo.pWaitDstStageMask    = waitStageMash;
        submitInfo.commandBufferCount   = 1;
        submitInfo.pCommandBuffers      = &g_drawCmdBuffer;
        submitInfo.signalSemaphoreCount = 0;
        submitInfo.pSignalSemaphores    = NULL;

        VkFence submitFence;
        VkFenceCreateInfo fenceCreateInfo = {};
        fenceCreateInfo.sType = VK_STRUCTURE_TYPE_FENCE_CREATE_INFO;
        vkCreateFence( g_device, &fenceCreateInfo, NULL, &submitFence );

        result =
        vkQueueSubmit( g_presentQueue, 1, &submitInfo, submitFence );

        vkWaitForFences( g_device, 1, &submitFence, VK_TRUE, UINT64_MAX );
        vkResetFences( g_device, 1, &submitFence );
        vkResetCommandBuffer( g_drawCmdBuffer, 0 );
    }

      // create and map the texture view
    VkImageViewCreateInfo textureImageViewCreateInfo = {};
    textureImageViewCreateInfo.sType    = VK_STRUCTURE_TYPE_IMAGE_VIEW_CREATE_INFO;
    textureImageViewCreateInfo.image    = textureImage;
    textureImageViewCreateInfo.viewType = VK_IMAGE_VIEW_TYPE_2D;
    textureImageViewCreateInfo.format   = VK_FORMAT_R32G32B32_SFLOAT;
    VkComponentMapping j = { VK_COMPONENT_SWIZZLE_R,
                             VK_COMPONENT_SWIZZLE_G,
                             VK_COMPONENT_SWIZZLE_B,
                             VK_COMPONENT_SWIZZLE_A };
    textureImageViewCreateInfo.components                      = j;
    textureImageViewCreateInfo.subresourceRange.aspectMask     = VK_IMAGE_ASPECT_COLOR_BIT;
    textureImageViewCreateInfo.subresourceRange.baseMipLevel   = 0;
    textureImageViewCreateInfo.subresourceRange.levelCount     = 1;
    textureImageViewCreateInfo.subresourceRange.baseArrayLayer = 0;
    textureImageViewCreateInfo.subresourceRange.layerCount     = 1;


    result =
      vkCreateImageView( g_device,
                         &textureImageViewCreateInfo,
                         NULL,
                         &g_textureView );

    DBG_ASSERT_VULKAN_MSG( result,
              "Failed to create image view." );

      }
```

Now you need to modify your existing code so it integrates the texture with the geometry. As previously your vertex will have been a position and a colour. Now you'll include texture coordinates.

```
      // Chapter 11
      // Our handle to our created vertex buffer and
      // its data
      // Vertex info
      struct vertex
```

```
{
  float x, y, z, w; // position
  float r, g, b;    // color
  float u, v;       // texture
};
```

Also when you create your texture, don't forget to include some texture coordinates:

```
vertex *triangle = (vertex *) mapped;
vertex v1 =
{  -1.0f, -1.0f, 0.0f, 1.0f,   // position
       0.0f,  1.0f, 0.0f,         // color
       0.0f,  0.0f };             // texture coords
vertex v2 =
{   1.0f, -1.0f, 0.0f, 1.0f,
       1.0f,  0.0f, 0.0f,
       1.0f,  0.0f };
vertex v3 =
{   0.0f,  1.0f, 0.0f, 1.0f,
       0.0f,  0.0f, 1.0f,
       0.5f,  1.0f };
```

You'll need to update your shader descriptor/binding information as you now have an additional texture/sampler/coordinates:

```
// Chapter 13
{
// Define and create our descriptors:
VkDescriptorSetLayoutBinding bindings[2];

// uniform buffer for our matrices:
bindings[0].binding            = 0;
bindings[0].descriptorType     = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
bindings[0].descriptorCount    = 1;
bindings[0].stageFlags         = VK_SHADER_STAGE_VERTEX_BIT;
bindings[0].pImmutableSamplers = NULL;

bindings[1].binding            = 1;
bindings[1].descriptorType     = VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER;
bindings[1].descriptorCount    = 1;
bindings[1].stageFlags         = VK_SHADER_STAGE_FRAGMENT_BIT;
bindings[1].pImmutableSamplers = NULL;

VkDescriptorSetLayoutCreateInfo setLayoutCreateInfo = {};
setLayoutCreateInfo.sType        = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO;
setLayoutCreateInfo.bindingCount = 2;
setLayoutCreateInfo.pBindings    = bindings;


VkResult result =
vkCreateDescriptorSetLayout( g_device,
                             &setLayoutCreateInfo,
                             NULL,
                             &g_setLayout );
DBG_ASSERT_VULKAN_MSG( result,
  "Failed to create DescriptorSetLayout." );
```

```
// decriptor pool creation:
VkDescriptorPoolSize uniformBufferPoolSize[2];
uniformBufferPoolSize[0].type            = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
uniformBufferPoolSize[0].descriptorCount = 1;
uniformBufferPoolSize[1].type            = VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER;
uniformBufferPoolSize[1].descriptorCount = 1;



VkDescriptorPoolCreateInfo poolCreateInfo = {};
poolCreateInfo.sType         = VK_STRUCTURE_TYPE_DESCRIPTOR_POOL_CREATE_INFO;
poolCreateInfo.maxSets       = 1;
poolCreateInfo.poolSizeCount = 2;
poolCreateInfo.pPoolSizes    = uniformBufferPoolSize;


VkDescriptorPool descriptorPool;
result = vkCreateDescriptorPool( g_device, &poolCreateInfo, NULL, &descriptorPool );
DBG_ASSERT_VULKAN_MSG( result,
  "Failed to create descriptor pool." );

// allocate our descriptor from the pool:
VkDescriptorSetAllocateInfo dsai = {};
dsai.sType              = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_ALLOCATE_INFO;
dsai.descriptorPool     = descriptorPool;
dsai.descriptorSetCount = 1;
dsai.pSetLayouts        = &g_setLayout;

result =
vkAllocateDescriptorSets( g_device, &dsai, &g_descriptorSet );
DBG_ASSERT_VULKAN_MSG( result,
  "Failed to allocate descriptor sets." );

// When a set is allocated all values are undefined
// and all descriptors are uninitialized. We must
// init all statically used bindings:
VkDescriptorBufferInfo dbi = {};
dbi.buffer = g_buffer;
dbi.offset = 0;
dbi.range  = VK_WHOLE_SIZE;

VkDescriptorImageInfo imageInfo = {};
imageInfo.imageLayout = VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL;
imageInfo.imageView   = g_textureView;
imageInfo.sampler     = g_textureSampler;

VkWriteDescriptorSet wd[2];
wd[0].sType            = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
wd[0].dstSet           = g_descriptorSet;
wd[0].dstBinding       = 0;
wd[0].dstArrayElement  = 0;
wd[0].descriptorCount  = 1;
wd[0].descriptorType   = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
wd[0].pImageInfo       = NULL;
wd[0].pBufferInfo      = &dbi;
wd[0].pTexelBufferView = NULL;

wd[1].sType            = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
wd[1].dstSet           = g_descriptorSet;
wd[1].dstBinding       = 1;
wd[1].dstArrayElement  = 0;
wd[1].descriptorCount  = 1;
```

```
        wd[1].descriptorType   = VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER;
        wd[1].pImageInfo       = &imageInfo;
        wd[1].pBufferInfo      = NULL;
        wd[1].pTexelBufferView = NULL;

        vkUpdateDescriptorSets( g_device, 2, wd, 0, NULL );
        }
```

The shader attributes (what data you're passing to your shaders). You're now passing texture coordinates as well as the positions and colours.

```
        VkVertexInputAttributeDescription vertexAttributeDescritpion[3];

        // position:
        vertexAttributeDescritpion[0].location = 0;
        vertexAttributeDescritpion[0].binding  = 0;
        vertexAttributeDescritpion[0].format   = VK_FORMAT_R32G32B32A32_SFLOAT;
        vertexAttributeDescritpion[0].offset   = 0;

        // colors:
        vertexAttributeDescritpion[1].location = 1;
        vertexAttributeDescritpion[1].binding  = 0;
        vertexAttributeDescritpion[1].format   = VK_FORMAT_R32G32B32_SFLOAT;
        vertexAttributeDescritpion[1].offset   = 4 * sizeof(float);

        // texture coords:
        vertexAttributeDescritpion[2].location = 2;
        vertexAttributeDescritpion[2].binding  = 0;
        vertexAttributeDescritpion[2].format   = VK_FORMAT_R32G32_SFLOAT;
        vertexAttributeDescritpion[2].offset   = 7 * sizeof(float);

        VkPipelineVertexInputStateCreateInfo vertexInputStateCreateInfo = {};
        vertexInputStateCreateInfo.sType =
VK_STRUCTURE_TYPE_PIPELINE_VERTEX_INPUT_STATE_CREATE_INFO;
        vertexInputStateCreateInfo.vertexBindingDescriptionCount   = 1;
        // attribute indexing is a function of the vertex index
        vertexInputStateCreateInfo.pVertexBindingDescriptions      = &vertexBindingDescription;
        vertexInputStateCreateInfo.vertexAttributeDescriptionCount = 3;
        vertexInputStateCreateInfo.pVertexAttributeDescriptions    =
vertexAttributeDescritpion;
```

Finally, you need to update your shaders (vertex and pixel shader to include the texture coordinates and sampler).

Vertex shader file (with the modifications highlighted):

```
// Vertex Shader
// shader version
#version 420

// input matrix transforms (3D) passed in
// from our program
```

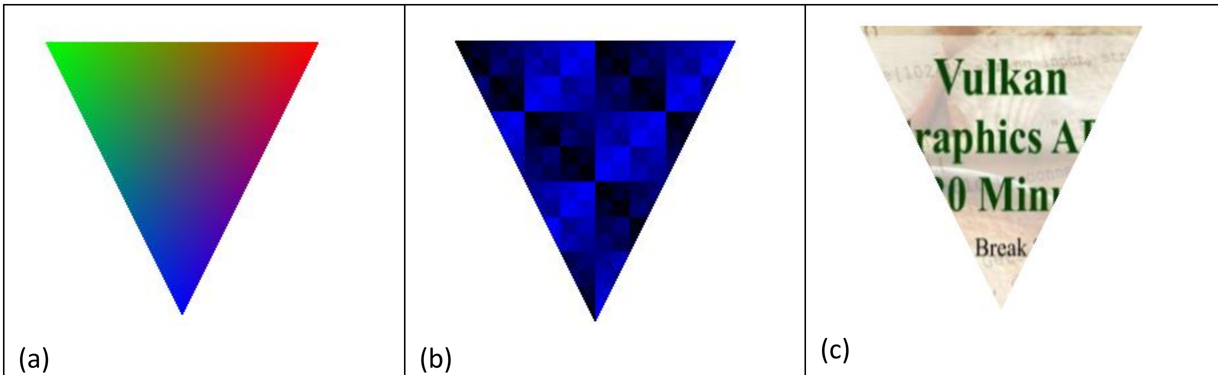
```
layout ( std140, set = 0, binding = 0 ) uniform buffer
{
  mat4 projection_matrix;
  mat4 view_matrix;
  mat4 model_matrix;
} UBO;

// input a position and a colour
layout( location = 0 ) in vec4 pos;
layout( location = 1 ) in vec3 color;
layout( location = 2 ) in vec2 texCoords;

// output a color
layout( location = 0 ) out vec4 outColor;
layout( location = 1 ) out vec2 outTexCoords;
```

Pixel shader file with the modifications highlighted:

```
// shader entry point
void main()
{
  // combine the matrices
  mat4 modelView = UBO.model_matrix * UBO.view_matrix;

  // transform position by matrices
  gl_Position = pos * ( modelView * UBO.projection_matrix );

  // pass input colour to the next stage
  outColor = vec4(color, 1.0);
  // Or a fixed test color (RED)
  // OUT.vColor = vec4( 1.0, 0.0, 0.0, 1 );

  // pass input texture coordinates onto the fragment shader
  outTexCoords = texCoords;

}// End main(..)


// Fragement Shader or sometimes called the
// `pixel shader'
// shader version
#version 420

layout(binding = 1) uniform sampler2D texSampler;

// input (from our vertex shader above)
layout ( location = 0 ) in vec4 inColor;
layout ( location = 1 ) in vec2 inTexCoords;


// final screen output color
layout ( location = 0 ) out vec4 uFragColor;

// shader entry point
void main()
{
  // pass input color along without any
  // modifications (e.g., Phong lighting
  // calculations could be done here)
```

```
  // uFragColor = inColor;

  uFragColor = texture(texSampler, inTexCoords);

}// End main(..)
```

If you did everything correctly, you should have a texture on your triangle.



(a) Original triangle with only colours

(b) Textured triangle using the test pattern (xor image)

(c)Textured triangle using the image loaded from the 32bit test image file