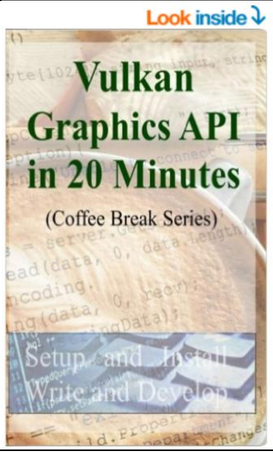
Vulkan API

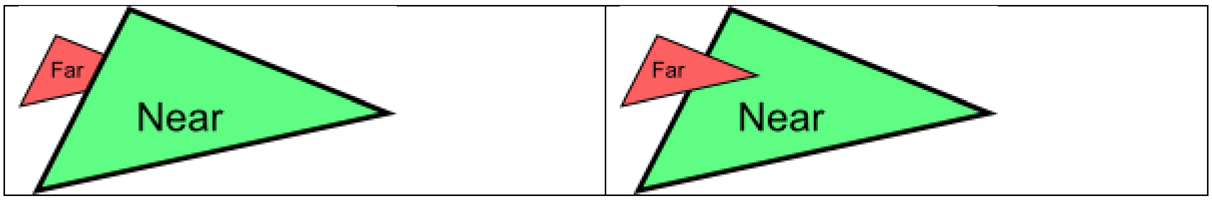
Extension Chapter 01

These few articles are tutorial exercises for the reader to work through to get familiar with the API. You've setup a basic solution. Rendered a coloured triangle. You're now ready to go through the code and start modifying the features, customizing and adding code to perform more complex operations. This will help you get to know the API inside-out. This extension chapter focuses on the steps necessary to add a depth buffer to the scene – a crucial component (especially for complex shapes).

 Look inside Vulkan Graphics API in 20 Minutes (Coffee Break Series) Setup and Install Write and Develop	**Vulkan Graphics API: in 20 Minutes (Coffee Break Series)** **Paperback** ISBN-10: 1535124857 ISBN-13: 978-1535124850

Why do we need the depth buffer?

To understand what's happens, here's an illustration to show the problems of drawing a scene with multiple triangles in the wrong order (without a depth buffer) – drawing a "far" triangle and a "near" triangle :



Currently, final listing demo only has a single triangle (i.e., 3 vertices). To help show the depth buffer working, you need to modify your code to draw a `cube', here is the code for an array of triangles that make up a cube:

```
// Multiple triangles (cube) – a cube has 6 faces with 2 triangles each, so this makes
6*2=12 triangles, and 12*3 vertices
static
const float      g_verticesForCube[] =
{
    -1.0f,-1.0f,-1.0f,  // triangle 1 : begin
    -1.0f,-1.0f, 1.0f,
    -1.0f, 1.0f, 1.0f,  // triangle 1 : end
```

```
     1.0f, 1.0f,-1.0f,  // triangle 2 : begin
    -1.0f,-1.0f,-1.0f,
    -1.0f, 1.0f,-1.0f,  // triangle 2 : end
     1.0f,-1.0f, 1.0f,
    -1.0f,-1.0f,-1.0f,
     1.0f,-1.0f,-1.0f,
     1.0f, 1.0f,-1.0f,
     1.0f,-1.0f,-1.0f,
    -1.0f,-1.0f,-1.0f,
    -1.0f,-1.0f,-1.0f,
    -1.0f, 1.0f, 1.0f,
    -1.0f, 1.0f,-1.0f,
     1.0f,-1.0f, 1.0f,
    -1.0f,-1.0f, 1.0f,
    -1.0f,-1.0f,-1.0f,
    -1.0f, 1.0f, 1.0f,
    -1.0f,-1.0f, 1.0f,
     1.0f,-1.0f, 1.0f,
     1.0f, 1.0f, 1.0f,
     1.0f,-1.0f,-1.0f,
     1.0f, 1.0f,-1.0f,
     1.0f,-1.0f,-1.0f,
     1.0f, 1.0f, 1.0f,
     1.0f,-1.0f, 1.0f,
     1.0f, 1.0f, 1.0f,
     1.0f, 1.0f,-1.0f,
    -1.0f, 1.0f,-1.0f,
     1.0f, 1.0f, 1.0f,
    -1.0f, 1.0f,-1.0f,
    -1.0f, 1.0f, 1.0f,
     1.0f, 1.0f, 1.0f,
    -1.0f, 1.0f, 1.0f,
     1.0f,-1.0f, 1.0f
};

const int      g_numTriangles = 12;
```

We also need to store our depth buffer image and its associated view handle:

```
// Extension A (Depth Buffer)
VkImage          g_depthImage             = NULL;
VkImageView      g_depthImageView        = NULL;
```

You're now ready to add the code to create and setup your depth buffer (e.g., allocate memory, type, and associated views). You need to add this whole block of code. Later code updates will highlight the lines that you need to change to integrate the depth buffer with the working implementation (e.g., framebuffer and pipeline).

```
       // Extension A (Depth Buffer)
       {
       // create a depth image:
    VkImageCreateInfo imageCreateInfo = {};
    imageCreateInfo.sType       = VK_STRUCTURE_TYPE_IMAGE_CREATE_INFO;
    imageCreateInfo.imageType   = VK_IMAGE_TYPE_2D;
    imageCreateInfo.format      = VK_FORMAT_D16_UNORM;
       VkExtent3D f = { g_width, g_height, 1 };
    imageCreateInfo.extent      = f; // { context.width, context.height, 1 };
    imageCreateInfo.mipLevels   = 1;
    imageCreateInfo.arrayLayers = 1;
    imageCreateInfo.samples     = VK_SAMPLE_COUNT_1_BIT;
    imageCreateInfo.tiling      = VK_IMAGE_TILING_OPTIMAL;
    imageCreateInfo.usage       = VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT;
    imageCreateInfo.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
    imageCreateInfo.queueFamilyIndexCount = 0;
    imageCreateInfo.pQueueFamilyIndices = NULL;
    imageCreateInfo.initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
    VkResult result =
       vkCreateImage( g_device, &imageCreateInfo, NULL, &g_depthImage );
    DBG_ASSERT_VULKAN_MSG( result,
              "Failed to create depth image." );

    VkMemoryRequirements memoryRequirements = {};
    vkGetImageMemoryRequirements( g_device, g_depthImage, &memoryRequirements );

       // Allocate memory for our depth buffer
           VkMemoryAllocateInfo imageAllocateInfo = {};
    imageAllocateInfo.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
    imageAllocateInfo.allocationSize = memoryRequirements.size;

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

    uint32_t memoryTypeBits = memoryRequirements.memoryTypeBits;
    VkMemoryPropertyFlags desiredMemoryFlags = VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT;
    for( uint32_t i = 0; i < VK_MAX_MEMORY_TYPES; ++i )
       {
       VkMemoryType memoryType = memoryProperties.memoryTypes[i];
        if( memoryTypeBits & 1 )
       {
       if( ( memoryType.propertyFlags & desiredMemoryFlags ) == desiredMemoryFlags )
        {
            imageAllocateInfo.memoryTypeIndex = i;
            break;
        }
       }
        memoryTypeBits = memoryTypeBits >> 1;
    }

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

    result = vkBindImageMemory( g_device, g_depthImage, imageMemory, 0 );
    DBG_ASSERT_VULKAN_MSG( result, "Failed to bind image memory." );

    // create the depth image view:
    VkImageAspectFlags aspectMask             = VK_IMAGE_ASPECT_DEPTH_BIT;
    VkImageViewCreateInfo imageViewCreateInfo = {};
    imageViewCreateInfo.sType                 =
VK_STRUCTURE_TYPE_IMAGE_VIEW_CREATE_INFO;
    imageViewCreateInfo.image                 = g_depthImage;
    imageViewCreateInfo.viewType              = VK_IMAGE_VIEW_TYPE_2D;
    imageViewCreateInfo.format                = imageCreateInfo.format;
       VkComponentMapping g                      = { VK_COMPONENT_SWIZZLE_IDENTITY,
VK_COMPONENT_SWIZZLE_IDENTITY, VK_COMPONENT_SWIZZLE_IDENTITY,
VK_COMPONENT_SWIZZLE_IDENTITY };
    imageViewCreateInfo.components            = g;
    imageViewCreateInfo.subresourceRange.aspectMask     = aspectMask;
    imageViewCreateInfo.subresourceRange.baseMipLevel   = 0;
    imageViewCreateInfo.subresourceRange.levelCount     = 1;
    imageViewCreateInfo.subresourceRange.baseArrayLayer = 0;
    imageViewCreateInfo.subresourceRange.layerCount = 1;
    result =
         vkCreateImageView( g_device,
                                 &imageViewCreateInfo,
                                         NULL,
                                         &g_depthImageView );
    DBG_ASSERT_VULKAN_MSG( result,
              "Failed to create image view." );
       }
```

You'll update the framebuffer (Chapter 9), I've highlighted the specific lines below to show the key lines of code you need to change to get the depth buffer working with the framebuffer (i.e., the lines highlighted in red).

```
       // Chapter 9
       // Frame buffer
       {
       // define our attachment points

       // 0 - colour screen buffer
       VkAttachmentDescription pass[2] = { };
       pass[0].format          = VK_FORMAT_B8G8R8A8_UNORM;
       pass[0].samples         = VK_SAMPLE_COUNT_1_BIT;
       pass[0].loadOp          = VK_ATTACHMENT_LOAD_OP_CLEAR;
       pass[0].storeOp         = VK_ATTACHMENT_STORE_OP_STORE;
       pass[0].stencilLoadOp  = VK_ATTACHMENT_LOAD_OP_DONT_CARE;
       pass[0].stencilStoreOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
       pass[0].initialLayout  = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL;
       pass[0].finalLayout    = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL;

    VkAttachmentReference car = {};
       car.attachment = 0;
       car.layout     = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL;

       // 1 - depth buffer
       pass[1].format          = VK_FORMAT_D16_UNORM;
    pass[1].samples         = VK_SAMPLE_COUNT_1_BIT;
    pass[1].loadOp          = VK_ATTACHMENT_LOAD_OP_CLEAR;
    pass[1].storeOp         = VK_ATTACHMENT_STORE_OP_DONT_CARE;
    pass[1].stencilLoadOp  = VK_ATTACHMENT_LOAD_OP_DONT_CARE;
    pass[1].stencilStoreOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
    pass[1].initialLayout  = VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL;
    pass[1].finalLayout    = VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL;

       VkAttachmentReference dar = {};
    dar.attachment = 1;
    dar.layout     = VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL;


       // create the one main subpass of our renderpass:
       VkSubpassDescription subpass    = {};
       subpass.pipelineBindPoint       = VK_PIPELINE_BIND_POINT_GRAPHICS;
       subpass.colorAttachmentCount    = 1;
```

```
        subpass.pColorAttachments       = &car;
        subpass.pDepthStencilAttachment = &dar;

        // create our main renderpass:
        VkRenderPassCreateInfo rpci = {};
        rpci.sType            = VK_STRUCTURE_TYPE_RENDER_PASS_CREATE_INFO;
        rpci.attachmentCount = 2; // colour and depth
        rpci.pAttachments     = pass;
        rpci.subpassCount     = 1;
        rpci.pSubpasses       = &subpass;

        VkResult result =
        vkCreateRenderPass( g_device, &rpci, NULL, &g_renderPass );

        DBG_ASSERT_VULKAN_MSG( result,
          "Failed to create renderpass" );


        // create our frame buffers:
        VkImageView frameBufferAttachments[2] = {0};

        frameBufferAttachments[1] = g_depthImageView;

        VkFramebufferCreateInfo fbci = {};
        fbci.sType = VK_STRUCTURE_TYPE_FRAMEBUFFER_CREATE_INFO;
        fbci.renderPass       = g_renderPass;
        // must be equal to the attachment count on render pass
        fbci.attachmentCount = 2;
        fbci.pAttachments     = frameBufferAttachments;
        fbci.width            = g_width;
        fbci.height           = g_height;
        fbci.layers           = 1;

        // create a framebuffer per swap chain imageView:
        g_frameBuffers = new VkFramebuffer[ 2 ];
        for( uint32_t i = 0; i < 2; ++i )
        {
               frameBufferAttachments[0] = g_presentImageViews[ i ];
               result =
               vkCreateFramebuffer( g_device, &fbci, NULL, &g_frameBuffers[i] );
               DBG_ASSERT_VULKAN_MSG( result,
                 "Failed to create framebuffer.");
        }// End for i

        }
```

Next, you'll update the vertex buffer. You're are now drawing more than one triangle (a cube).

```
        const int numberOfVertices = sizeof( g_verticesForCube ) / (sizeof( float ) *
3);

        const int numberOfTrianges = numberOfVertices / 3;
        DBG_ASSERT(numberOfTrianges == g_numTriangles );

        // create our vertex buffer:
        VkBufferCreateInfo vertexInputBufferInfo = {};
        vertexInputBufferInfo.sType  = VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO;
        // size in bytes of our data
        // a single triangle requires 3 vertices
        vertexInputBufferInfo.size                  = sizeof(vertex) *
numberOfVertices;
        vertexInputBufferInfo.usage                 =
VK_BUFFER_USAGE_VERTEX_BUFFER_BIT;
        vertexInputBufferInfo.sharingMode           = VK_SHARING_MODE_EXCLUSIVE;
        vertexInputBufferInfo.queueFamilyIndexCount = 0;
        vertexInputBufferInfo.pQueueFamilyIndices   = NULL;
```

Copy the new vertices from your static array you declared at the start of the document into the Vulkan buffer:

```
        result =
        vkMapMemory( g_device, vertexBufferMemory, 0,
          VK_WHOLE_SIZE, 0, &mapped );
        DBG_ASSERT_VULKAN_MSG( result,
          "Failed to mapp buffer memory." );

        vertex *trianglevertices = (vertex *) mapped;

        for (int i=0; i<numberOfVertices; ++i)
        {
                // position
                trianglevertices[i].x = g_verticesForCube[i*3 + 0];
                trianglevertices[i].y = g_verticesForCube[i*3 + 1];
                trianglevertices[i].z = g_verticesForCube[i*3 + 2];
                trianglevertices[i].w = 1;

                // colour
                trianglevertices[i].r = 1;
                // small changing value for the green
                trianglevertices[i].g = (float)(i%10) * 0.1f);
                trianglevertices[i].b = 0;
        }// End for i


        vkUnmapMemory( g_device, vertexBufferMemory );
```

You now need to tell your pipeline that you have a depthbuffer setup and it should be enabled:

```
        // add depth buffer options to the pipeline
        VkPipelineDepthStencilStateCreateInfo depthStencil = {};
        depthStencil.sType                     =
VK_STRUCTURE_TYPE_PIPELINE_DEPTH_STENCIL_STATE_CREATE_INFO;
        depthStencil.depthTestEnable           = VK_TRUE;
        depthStencil.depthWriteEnable          = VK_TRUE;
        depthStencil.depthCompareOp            = VK_COMPARE_OP_LESS;
```

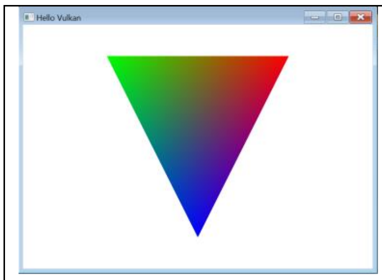
```
       depthStencil.depthBoundsTestEnable = VK_FALSE;
       depthStencil.minDepthBounds        = 0.0f; // Optional
       depthStencil.maxDepthBounds        = 1.0f; // Optional
       depthStencil.stencilTestEnable     = VK_FALSE;


       // and finally, pipeline config and creation:
       VkGraphicsPipelineCreateInfo pipelineCreateInfo = {};
       pipelineCreateInfo.sType = VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_CREATE_INFO;
       pipelineCreateInfo.stageCount          = 2;
       pipelineCreateInfo.pStages             = shaderStageCreateInfo;
       pipelineCreateInfo.pVertexInputState   = &vertexInputStateCreateInfo;
       pipelineCreateInfo.pInputAssemblyState = &inputAssemblyStateCreateInfo;
       pipelineCreateInfo.pTessellationState  = NULL;
       pipelineCreateInfo.pViewportState      = &viewportState;
       pipelineCreateInfo.pRasterizationState = &rasterizationState;
       pipelineCreateInfo.pMultisampleState   = &multisampleState;
       pipelineCreateInfo.pDepthStencilState  = &depthStencil;
       pipelineCreateInfo.pColorBlendState    = &colorBlendState;
       pipelineCreateInfo.pDynamicState       = &dynamicStateCreateInfo;
       pipelineCreateInfo.layout              = g_pipelineLayout;
       pipelineCreateInfo.renderPass          = g_renderPass;
       pipelineCreateInfo.subpass             = 0;
       pipelineCreateInfo.basePipelineHandle  = NULL;
       pipelineCreateInfo.basePipelineIndex   = 0;
```

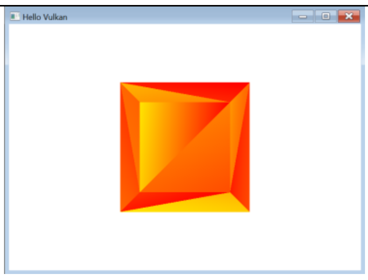
Finally, you need to modify your draw call. As you're drawing a cube:

```
vkCmdDraw( g_drawCmdBuffer,
                 // vertex count
                 g_numTriangles * 3,
                 // instance count
                 g_numTriangles,
                 // first index
                 0,
                 // first instance
                 0 );
```

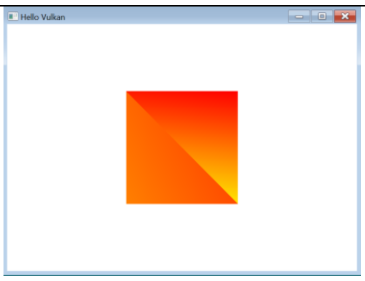
You've changed the code to draw a cube – if you enable/disable the depth buffer you can see why it's so important).



(a)



(b)



(c)

(a) is the original triangle demo

(b) draw a cube (without the depth buffer)

(c) draw the cube with the added depth buffer