3

Contents

[Build an endless 2D canvas application 3](#_Toc504950250)

[Application Overview 4](#_Toc504950251)

[Configure 5](#_Toc504950252)

[Retrieve Image filename from disk 5](#_Toc504950253)

[Setup 6](#_Toc504950254)

[Buffers in Vulkan 6](#_Toc504950255)

[Texture Sampler 9](#_Toc504950256)

[Descriptors 10](#_Toc504950257)

[2D geometry in Vulkan 12](#_Toc504950258)

[Images 19](#_Toc504950259)

[Graphics Pipeline 30](#_Toc504950260)

[Shaders 32](#_Toc504950261)

[Building Command Buffers 34](#_Toc504950262)

[Render 35](#_Toc504950263)

[Update 36](#_Toc504950264)

[Close 37](#_Toc504950265)

[Application Output 38](#_Toc504950266)

[Summary 39](#_Toc504950267)

3

# Build an endless 2D canvas application

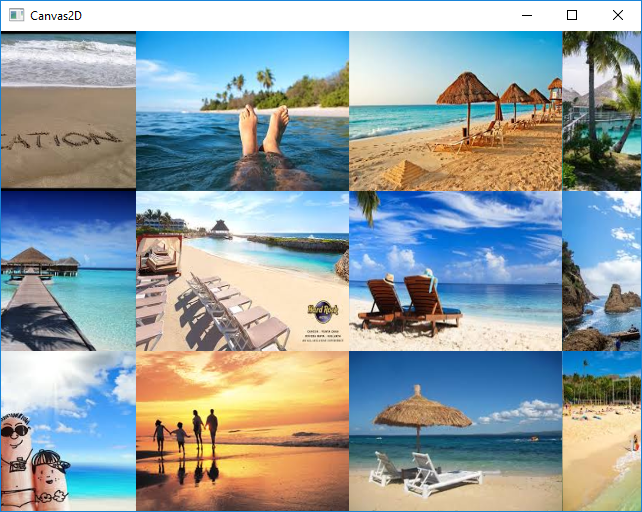
In the last chapter we looked at setting up the Vulkan development environment and developed a simple app using VulkanApp base class. This chapter will explain the steps to create an end-to-end graphics application that will let you to view your own collection of pictures (jpg images) in a configurable 2D scrolling tiles. We will be using the VulkanApp class to handle majority of the Vulkan specific tasks for this application. We will also be adding additional methods and member variables that will be handy for this application as well as several other applications that we will be building using VulkanApp class in this book.

In this chapter we will cover the following technical topics and by the end of the chapter you should be able to understand the steps required to build a 2D graphics application using Vulkan.

* Steps to create a 2D geometry in Vulkan
* Steps to create textures in Vulkan
* Steps to send application specific data to shader
* Vertex operations in vertex shader
* Sampling texture in the fragment shader
* Summary

## Application Overview

The following screenshot shows the output of application we are about to build in this chapter. The source code of this chapter is under “Canvas2D” in the git repository. We will skip the application setup as it is very similar to the previous chapter and jump right in to the additional changes we did from Chapter 3 to build this app. As shown below, the application loads a series of images and render them as tiles using Vulkan API. These tiles scroll smoothly horizontally back and forth. The number of tiles in each row and column can be changed in the app.



This example uses the following 2 classes:

**Canvas2D**: A class derived from VulkanApp to manage the entire application.

**Quad**: A class to manage Vulkan objects for each Quad primitive (tile) in the scrolling canvas. Canvas2D class keeps an array of Quad objects to render each tile on the screen.

The application also use the following 3rd party module:

* **Stb libraries**: These are set of single file header libraries in C/C++. In this chapter we will be using the stb\_image.h and stb\_image\_resize.h file. You need to download these 2 files from https://github.com/nothings/stb and place it in 3rdparty\stb\ folder. Todo: We may git rid of this using Qt

Now let’s look how to build this application by looking at what we need to do at Configure(), Setup(), Update(), Render(), Close() methods in Canvas2DApp class.

## Configure

In the configure method in Canvas2DApp class we set the application name and the application windows dimension as shown below. The GetImageFilenames() helper function retrieves image file names in the executable folder and store it in a m\_imageFiles.

|  |
| --- |
| bool Canvas2DApp::Configure()  {  SetApplicationName("Canvas2D");  SetWindowDimension(800, 600);  m\_numImageFiles = GetImageFilenames();  return (m\_numImageFiles ? true : false);  } |

### Retrieve Image filename from disk

The implementation for GetImageFilenames() is shown below. If there are no image files found in the executable folder then this methods returns 0 which will automatically quit the application.

|  |
| --- |
| // Get the list of jpg image files in the current folder and  // store them in m\_imageFiles.  // Returns the count of jpg files  int Canvas2DApp::GetImageFilenames()  {  QDirIterator qDirIt(".", QStringList() << "\*.jpg",  QDir::Files, QDirIterator::Subdirectories);  while (qDirIt.hasNext())  {  QString qText = qDirIt.next();  std::string filename = qText.toUtf8().constData();  m\_imageFiles.push\_back(filename);  }  return ((int)m\_imageFiles.size());  } |

## Setup

The following setup method in Canvas2DApp creates several new Vulkan objects compared to the previous chapter. We will learn how the host application can pass data to the shader using Uniform buffers, how to create a 2D mesh (ie., Quad) using vertex buffer and how to load and apply texture to the 2D mesh.

|  |
| --- |
| bool Canvas2DApp::Setup()  {  bool result = false;  CreateUniformBuffer();  CreateTextureSampler();  CreateDescriptorLayout();  CreateDescriptorPool();  CreateImageTiles();  CreateGraphicsPipeline();  result = BuildCommandBuffers();  return (result);  } |

### Buffers in Vulkan

Buffer is a vulkan object which represents a linear arrays of data that can be used at graphics or compute pipeline. The details about the buffer are typically specified using a descriptor sets or via certain commands. There are several usages for buffer object in Vulkan. In this chapter we will be creating the following buffers:

* **Uniform Buffer**: Uniform buffers are used to pass data from the host application to the shader in the graphics or compute pipeline. The shader can only read from this type of buffer. Application typically use this type of buffer to keep both the host application running in the CPU & shaders executed in the GPU in sync with application specific information, such as 3D camera position, game status, etc.
* **Transfer Buffer**: Transfer buffers are used to transfer data from one buffer to another buffer. Typical usage is to transfer image content from a staging buffer from CPU to a GPU buffer.
* **Vertex Buffer**: Vertex buffers are used to hold vertex data. The data is accessed by the vertex shader in the graphics pipeline.

Creating a buffer requires the following 4 steps:

1. Create a handle to the Vulkan buffer object using vkCreateBuffer().
2. Obtain the memory requirement for the buffer object using vkGetBufferMemoryRequirements().
3. Allocate the device memory as specified by the application for this buffer object using vkAllocateMemory().
4. Bind the memory object to the buffer object using vkBindBufferMemory()

The following helper method in VulkanApp creates a buffer object and device memory object based on buffer size, buffer usage and required memory preference.

|  |
| --- |
| bool VulkanApp::CreateBuffer(  VkDeviceSize size, VkBufferUsageFlags usage,  VkMemoryPropertyFlags properties, VkBuffer& buffer,  VkDeviceMemory& bufferMemory)  {  bool result = true;  VkBufferCreateInfo bufferInfo = {};  bufferInfo.sType = VK\_STRUCTURE\_TYPE\_BUFFER\_CREATE\_INFO;  bufferInfo.size = size;  bufferInfo.usage = usage;  bufferInfo.sharingMode = VK\_SHARING\_MODE\_EXCLUSIVE;  if (vkCreateBuffer(m\_hDevice, &bufferInfo,  nullptr, &buffer) == VK\_SUCCESS)  {  VkMemoryRequirements memRequirements = {};  vkGetBufferMemoryRequirements(m\_hDevice, buffer,  &memRequirements);  VkMemoryAllocateInfo allocInfo = {};  allocInfo.sType = VK\_STRUCTURE\_TYPE\_MEMORY\_ALLOCATE\_INFO;  allocInfo.allocationSize = memRequirements.size;  allocInfo.memoryTypeIndex =  GetMemoryTypeIndex(properties, memRequirements.memoryTypeBits);  if (vkAllocateMemory(m\_hDevice, &allocInfo,  nullptr, &bufferMemory) == VK\_SUCCESS)  {  vkBindBufferMemory(m\_hDevice, buffer, bufferMemory, 0);  }  else  {  result = false;  LogError("Failed to allocate buffer memory!");  }  }  else  {  result = false;  LogError("Failed to create buffer!");  }  return (result);  } |

The GetMemoryTypeIndex() helper method scans through the available memory properties supported by the physical device and retrieve the memory type index based on the given memory properties and memory requirements of the buffer object.

|  |
| --- |
| uint32\_t VulkanApp::GetMemoryTypeIndex(  VkMemoryPropertyFlags properties, uint32\_t memoryTypeBits)  {  uint32\_t memoryTypeIndex = -1;  for (uint32\_t i = 0;  i < m\_physicalDeviceInfo.memProp.memoryTypeCount &&  (memoryTypeIndex == -1); i++)  {  if ((memoryTypeBits & (1 << i)) &&  (m\_physicalDeviceInfo.memProp.memoryTypes[i].propertyFlags &  properties) == properties)  {  memoryTypeIndex = i;  }  }  if (memoryTypeIndex == -1)  {  LogError("Unable to find suitable memory type!");  }  return (memoryTypeIndex);  } |

#### Uniform Buffer

In this chapter we will be using uniform buffer to update the x-offset to scroll the Canvas back and forth horizontally using the vertex shader. In order to do this we need to create a data structure on the host application that needs to be sent to the vertex shader. Create the following struct to manage the Uniform buffer content. We will be creating a similar layout in the vertex shader to access the offset values on each shader invocation.

|  |
| --- |
| struct UniformBufferObject  {  glm::vec2 offset; // Scrolling offset  }; |

Now let’s look at the method to create a Uniform buffer for our application to pass application specific data to the shader. We will be using the CreateBuffer() helper method to create the buffer object and allocate memory to hold the UniformBufferObject data structure.

The following members in Canvas2D.h holds the handle to the uniform buffer object and the associated memory object.

|  |
| --- |
| VkBuffer m\_uniformBuffer;  VkDeviceMemory m\_uniformBufferMemory; |

The following code shows the initialization for the required parameters for CreateBuffer() to create a uniform buffer.

|  |
| --- |
| void Canvas2DApp::CreateUniformBuffer()  {  // memory size to allocate  VkDeviceSize size = sizeof(UniformBufferObject);    // Usage flag set to Uniform buffer  VkBufferUsageFlags usageFlags = VK\_BUFFER\_USAGE\_UNIFORM\_BUFFER\_BIT;    // Set the property flag to indicate the memory is visible  // on host side and the content is coherent between CPU & GPU  VkMemoryPropertyFlags propertyFlags =  VK\_MEMORY\_PROPERTY\_HOST\_VISIBLE\_BIT |  VK\_MEMORY\_PROPERTY\_HOST\_COHERENT\_BIT;    CreateBuffer(size, usageFlags, propertyFlags,  m\_uniformBuffer, m\_uniformBufferMemory);  } |

We will look at how the Uniform buffer is sent to the GPU using a descriptor set. But before that we need to create a texture sampler which is necessary to inform the fragment shader how to sample the texture images for each quad.

### Texture Sampler

The following member in Canvas2D.h holds the handle to the uniform buffer object and the associated memory object.

|  |
| --- |
| VkSampler m\_textureSampler; |

To create a texture sampler using the vkCreateSampler() method we need to set the sampler info with the appropriate sampler details. The following code shows the values that we will be using for this application.

|  |
| --- |
| void Canvas2DApp::CreateTextureSampler()  {  VkSamplerCreateInfo samplerInfo = {};  samplerInfo.sType = VK\_STRUCTURE\_TYPE\_SAMPLER\_CREATE\_INFO;  samplerInfo.magFilter = VK\_FILTER\_LINEAR;  samplerInfo.minFilter = VK\_FILTER\_LINEAR;  samplerInfo.addressModeU = VK\_SAMPLER\_ADDRESS\_MODE\_REPEAT;  samplerInfo.addressModeV = VK\_SAMPLER\_ADDRESS\_MODE\_REPEAT;  samplerInfo.addressModeW = VK\_SAMPLER\_ADDRESS\_MODE\_REPEAT;  samplerInfo.anisotropyEnable = VK\_TRUE;  samplerInfo.maxAnisotropy = 16;  samplerInfo.borderColor = VK\_BORDER\_COLOR\_INT\_OPAQUE\_BLACK;  samplerInfo.unnormalizedCoordinates = VK\_FALSE;  samplerInfo.compareEnable = VK\_FALSE;  samplerInfo.compareOp = VK\_COMPARE\_OP\_ALWAYS;  samplerInfo.mipmapMode = VK\_SAMPLER\_MIPMAP\_MODE\_LINEAR;  if (vkCreateSampler(m\_hDevice, &samplerInfo, nullptr, &m\_textureSampler) != VK\_SUCCESS)  {  LogError("vkCreateSampler() failed!");  }  } |

### Descriptors

Descriptors represent binding information for buffers, samplers or images etc for each of the shader stages in the graphics pipeline or compute pipeline. Descriptors are typically grouped together into descriptor set objects, an opaque object that contains storage for a set of descriptors.

##### Descriptor Set Layout

A descriptor set layout represents a group of descriptor bindings. In our example we need to setup a descriptor layout to bind the uniform buffer to the vertex shader stage and a descriptor layout to bind the texture sampler in the fragment shader stage.

The following member in Canvas2D.h holds the handle to the descriptor set layout.

|  |
| --- |
| VkDescriptorSetLayout m\_descSetLayout; |

The following method creates a descriptor layout object using vkCreateDescriptorSetLayout() method. Uniform buffer is set to bind to slot 0 in uniform buffer namespace in vertex shader. Texture sampler is set to bind to slot 1 in image sampler namespace in fragment shader.

|  |
| --- |
| void Canvas2DApp::CreateDescriptorLayout()  {  // Setup binding for uniform buffer for Vertex shader  // and sampler for the fragment shader  VkDescriptorSetLayoutBinding bindings[2] = {};  // Initialize the layout binding for Uniform buffer  bindings[0].binding = 0;  bindings[0].descriptorCount = 1;  bindings[0].descriptorType = VK\_DESCRIPTOR\_TYPE\_UNIFORM\_BUFFER;  bindings[0].pImmutableSamplers = nullptr;  bindings[0].stageFlags = VK\_SHADER\_STAGE\_VERTEX\_BIT;  // Initialize the layout binding for the sampler  VkDescriptorSetLayoutBinding samplerLayoutBinding = {};  bindings[1].binding = 1;  bindings[1].descriptorCount = 1;  bindings[1].descriptorType =  VK\_DESCRIPTOR\_TYPE\_COMBINED\_IMAGE\_SAMPLER;  bindings[1].pImmutableSamplers = nullptr;  bindings[1].stageFlags = VK\_SHADER\_STAGE\_FRAGMENT\_BIT;  // Create the descriptor set layout  VkDescriptorSetLayoutCreateInfo layoutInfo = {};  layoutInfo.sType =  VK\_STRUCTURE\_TYPE\_DESCRIPTOR\_SET\_LAYOUT\_CREATE\_INFO;  layoutInfo.bindingCount = 2;  layoutInfo.pBindings = bindings;  if (vkCreateDescriptorSetLayout(m\_hDevice,  &layoutInfo,  nullptr,  &m\_descSetLayout) != VK\_SUCCESS)  {  LogError("vkCreateDescriptorSetLayout() failed!");  }  } |

##### Descriptor Pool

A descriptor pool is used to maintain a pool of descriptors from which descriptor sets are allocated. In our example we need to create a descriptor set to render each quad with its own descriptors.

The following member in Canvas2D.h holds the handle to the descriptor pool.

|  |
| --- |
| VkDescriptorPool m\_descriptorPool; |

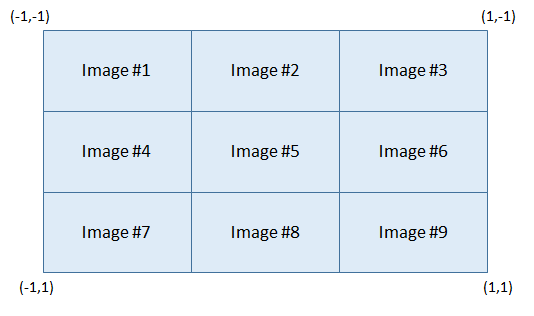
Allocate the descriptor pool to hold a descriptor for each image as shown below. The m\_numImageFiles contains the count of number of images found in the executable folder. So we need to create a pool of size m\_numImageFiles.

|  |
| --- |
| void Canvas2DApp::CreateDescriptorPool()  {  VkDescriptorPoolSize poolSizes[2] = {};  poolSizes[0].type = VK\_DESCRIPTOR\_TYPE\_UNIFORM\_BUFFER;  poolSizes[0].descriptorCount = m\_numImageFiles;    poolSizes[1].type = VK\_DESCRIPTOR\_TYPE\_COMBINED\_IMAGE\_SAMPLER;  poolSizes[1].descriptorCount = m\_numImageFiles;  VkDescriptorPoolCreateInfo poolInfo = {};  poolInfo.sType = VK\_STRUCTURE\_TYPE\_DESCRIPTOR\_POOL\_CREATE\_INFO;  poolInfo.poolSizeCount = 2;  poolInfo.pPoolSizes = poolSizes;  poolInfo.maxSets = m\_numImageFiles;  if (vkCreateDescriptorPool(m\_hDevice, &poolInfo,  nullptr, &m\_descriptorPool) != VK\_SUCCESS)  {  LogError("vkCreateDescriptorPool() failed!");  }  } |

### 2D geometry in Vulkan

In the previous chapter we looked at rendering a single triangle without sending any vertex information to the GPU. We used the vertex id and hard coded the vertex position in the vertex shader. In this section we will look at creating a 2D geometry to render a QUAD using Vulkan.

In this example we render a set of images as a matrix of tiles as shown below. Hence if we have to show 9 images as a 3x3 tile then each image takes about 1/3 of the width of the display window and 1/3rd of the height of the display window. We will define the tile’s offset and size in normalized device coordinates.



The application must manage the location and size of each tile, the image to render and the relative delta from the origin of the application window to scroll the tile horizontally. The following method initialize a Quad object for each tile. By default the application shows 3x3 visible tiles but the same code works for any dimension of tiles. Once we calculate the total number of columns and rows to create we pass the tile’s position, dimension and the image filename to the Init() method in the Quad class. The Init() method creates several internal Vulkan objects to render the Quad. The SetDescriptorSet() method creates the descriptor set for the tile using the tile’s image.

The following members are added in Canvas2D.h

|  |
| --- |
| // A Quad object for each image file  vector<Quad> m\_quad;  // Total number of columns needed for the canvas  float m\_numTotalCols;  // Number of visible columns to show in the window  float m\_numVisibleCols;  // Number of visible rows to show in the window  float m\_numVisibleRows; |

The following method initialize each Quad object:

|  |
| --- |
| // Create an instance of Quad object for each image  // file found in the executable folder  void Canvas2DApp::CreateImageTiles()  {  // Num of visible tiles per column  m\_numVisibleCols = 3;    // Num of visible tiles per row  m\_numVisibleRows = 3;  // Total colums to create  m\_numTotalCols = m\_numImageFiles / m\_numVisibleRows;  m\_quad.resize(m\_numImageFiles);  float w = 2.0f / m\_numVisibleCols;  float h = 2.0f / m\_numVisibleRows;  uint32\_t idx = 0;  for (float i = 0; i < m\_numTotalCols; i++)  {  for (float j = 0; j < m\_numVisibleRows; j++)  {  if (idx < m\_numImageFiles)  {  // Initialize a quad object for the  // given image file  m\_quad[idx].Init(  this, i, j, w, h, m\_imageFiles[idx]);    // Create and set the descriptor set  m\_quad[idx].SetDescriptorSet(  CreateDescriptorSet(  m\_quad[idx].GetImageView()));  idx++;  }  }  } } |

#### Quad class

We can encapsulate many of the Vulkan objects required to manage the QUAD into a C++ class. The following class definition shows the list of methods and members in Quad class.

|  |
| --- |
| class Quad  {  public:  ...  void Init(…);  void SetDescriptorSet(…);  ...  private:  // Store app specific objects  VkBuffer m\_vertexBuffer;  VkDeviceMemory m\_vertexBufferMemory;  VkDevice m\_hDevice;  VulkanApp\* m\_pApp;  VkVertexInputBindingDescription m\_vertexBindDesc;  VkVertexInputAttributeDescription m\_vertexAttrDesc[3];  VkImage m\_textureImage;  VkDeviceMemory m\_textureImageMemory;  VkImageView m\_textureImageView;  VkDescriptorSet m\_descriptorSet;  bool CreateVertexBuffer(…);  void CreateTextureImage(…);  bool TransitionImageLayout(…);  void CopyBufferToImage(…);  VkImageView CreateImageView(…);  }; |

#### Initializing a Quad object

The Quad class manages a list of Vulkan objects to render the Quad in the GPU. The following bullets list these Vulkan objects and its purpose.

1. **Vertex Buffer**: A handle to the vertex buffer that represent the quad
2. **Device memory for vertex buffer**: A handle to the device memory to store the 4 vertex of the Quad
3. **Vertex Input Binding Description**: A structure to describe the Quad vertex binding information, such as binding slot number, vertex stride etc
4. **Vertex Input Attribute Description**: A structure to describe the attributes of each elements in the vertex, such as type, offset, binding slot, format etc.
5. **Texture image**: A handle to the texture image that fills the quad
6. **Device memory for Texture image**: A handle to the device memory to store the image content
7. **Texture Image view**: A handle to the texture image view for the Quad’s texture. This is used in the descriptor set.
8. **Descriptor set**: A descriptor set to bind to the graphics pipeline to render the Quad.

The following Init() method initialize the vulkan objects required to render the quad.

* x,y parameters specifies the column and row number
* w,h parameters specifies the width and height in normalized device coordinates
* textureFilename specifies the full path of the texture filename (.jpg)

|  |
| --- |
| void Quad::Init(VulkanApp\* pApp, float x, float y,  float w, float h, string textureFilename)  {  m\_pApp = pApp;  m\_hDevice = pApp->GetDevice();  // Calculate vertex positions in normalized device coordinates  float xf = (float)x\*w;  float yf = (float)y\*h;  float x1 = -1.0f + xf;  float y1 = -1.0f + yf;  float x2 = x1 + w;  float y2 = y1 + h;  // Initialize vertex positions for the quad in  // normalized device coordinates  const std::vector<Vertex> vertices =  {  { { x1, y1 },{ 1.0f, 1.0f, 1.0f },{ 0.0f,0.0f } }, // Top left  { { x2, y1 },{ 0.0f, 0.0f, 1.0f },{ 1.0f,0.0f } }, // Top right  { { x1, y2 },{ 0.0f, 1.0f, 0.0f },{ 0.0f,1.0f } }, // Bottom left  { { x2, y2 },{ 1.0f, 0.0f, 0.0f },{ 1.0f,1.0f } } // Bottom right  };    // Initialize vertex binding description  m\_vertexBindDesc.binding = 0;  m\_vertexBindDesc.stride = sizeof(Vertex);  m\_vertexBindDesc.inputRate = VK\_VERTEX\_INPUT\_RATE\_VERTEX;  // Initialize Vertex attributes  // Vertex position  m\_vertexAttrDesc[0].binding = 0;  m\_vertexAttrDesc[0].location = 0;  m\_vertexAttrDesc[0].format = VK\_FORMAT\_R32G32\_SFLOAT;  m\_vertexAttrDesc[0].offset = offsetof(Vertex, position);  // Vertex color  m\_vertexAttrDesc[1].binding = 0;  m\_vertexAttrDesc[1].location = 1;  m\_vertexAttrDesc[1].format = VK\_FORMAT\_R32G32B32\_SFLOAT;  m\_vertexAttrDesc[1].offset = offsetof(Vertex, color);  // Texture coordinate  m\_vertexAttrDesc[2].binding = 0;  m\_vertexAttrDesc[2].location = 2;  m\_vertexAttrDesc[2].format = VK\_FORMAT\_R32G32\_SFLOAT;  m\_vertexAttrDesc[2].offset = offsetof(Vertex, texCoord);  CreateVertexBuffer(vertices);  CreateTextureImage(textureFilename);  m\_textureImageView = CreateImageView(  m\_textureImage, VK\_FORMAT\_R8G8B8A8\_UNORM);  } |

The method first calculates the quad coordinate in the normalized device coordinate and initialize the 4 vertex of the quad. The Vertex data structure holds the per vertex position, color and texture coordinate information.

|  |
| --- |
| struct Vertex  {  glm::vec2 position;  glm::vec3 color;  glm::vec2 texCoord;  }; |

The vertex binding description is initialized with the binding slot number 0 and the stride is set to the size of the vertex data structure. The vertex attribute description for each of the elements in the vertex data structure is initialized with the appropriate binding slot number, location index, format and offset. The vertex shader can refer these elements using the corresponding binding slot and location index.

#### Vertex Buffer

Once the 4 vertices are defined we create the vertex buffer using CreateVertexBuffer() method as shown below. Note that we are using our helper method CreateBuffer() here to create the handle to the vertex buffer object. The buffer size is set to the size of the 4 Vertex data structure, the usage is set to vertex buffer. The object is set to host visible so we can update the vertex buffer from the host side (Though this is not an optimal method we will address a performant way to handle this in future chapters). The helper method returns the handle to the vertex buffer and the handle to the vertex buffer memory. The UpdateMemory() helper method copies the vertices to the vertex buffer memory object.

|  |
| --- |
| bool Quad::CreateVertexBuffer(const std::vector<Vertex> vertices)  {  bool result = true;  // Calculate the buffer size to allocate  VkDeviceSize bufferSize = sizeof(vertices[0]) \* vertices.size();  result = m\_pApp->CreateBuffer(bufferSize,  VK\_BUFFER\_USAGE\_VERTEX\_BUFFER\_BIT,  (VK\_MEMORY\_PROPERTY\_HOST\_VISIBLE\_BIT |  VK\_MEMORY\_PROPERTY\_HOST\_COHERENT\_BIT),  m\_vertexBuffer,  m\_vertexBufferMemory);    if (result == true)  {  // Update memory with vertex data  m\_pApp->UpdateMemory(m\_vertexBufferMemory, 0,  bufferSize, 0, vertices.data());  }  return (result);  } |

Here is the implementation of the UpdateMemory() helper method in VulkanApp. The method uses vkMapMemory() to returns a memory pointer to copy the 4 vertices to the vertex buffer memory object. Once we copy the content we must un-map using the vkUnmapMemory() API.

|  |
| --- |
| void VulkanApp::UpdateMemory(VkDeviceMemory deviceMem,  VkDeviceSize offset, VkDeviceSize size,  VkMemoryMapFlags flags, const void\* pData)  {  void\* pHostMem = nullptr;  vkMapMemory(m\_hDevice, deviceMem, offset, size, flags, &pHostMem);  memcpy(pHostMem, pData, (size\_t)size);  vkUnmapMemory(m\_hDevice, deviceMem);  } |

### Images

Earlier we looked at creating Vulkan buffers to hold linear data such as vertex buffer. In this section we will look at Images, a 2 dimensional or 3 dimensional arrays of data that can be bound to the graphics or compute pipeline. Example: To bind a texture at the fragment shader to fill a 3D mesh.

The next step in the Quad initialization step is to load the jpg file associated with the Quad and create a Vulkan image handle and copy the loaded jpg pixel content in to it.

Here are the pseudo steps to create the following method uses:

1. Load the jpg file and retrieve the pixel content.
2. Create a staging buffer object that can be accessible on the host (CPU) side.
3. Copy the pixel content to the staging buffer object
4. Create an image object that is only accessible on the GPU side
5. Setup a command buffer to copy the staging buffer to the image object.
6. Destroy the staging buffer object and its associated device memory object

The following code shows how to load the JPG file and retrieve the pixel content. We use the STB image library to load the JPG file and retrieve the pixel information. The stb\_image.h found in <https://github.com/nothings/stb> must be copied to the source\3rdparty\stb folder and include the stb\_image.h in Quad.cpp

|  |
| --- |
| #define STB\_IMAGE\_IMPLEMENTATION  #include <stb\_image.h>  void Quad::CreateTextureImage(string filename)  {  // 1: Load the jpg file and retrieve the pixel content.  int texWidth, texHeight, texChannels;  stbi\_uc\* pPixels = stbi\_load(filename.c\_str(), &texWidth,  &texHeight, &texChannels, STBI\_rgb\_alpha);  …  …  } |

The 2nd step is to create a staging buffer using our CreateBuffer() helper method. The size of the buffer is set to the width\*height\*4, where each pixel takes 4 bytes (1 byte to represent each channel of RGBA).

|  |
| --- |
| void Quad::CreateTextureImage(string filename)  {  // 1: Load the jpg file and retrieve the pixel content.  …  …  if (pPixels)  {  // 2: Create a staging buffer object that can be  // accessible on the host (CPU) side.  VkBuffer stagingBuffer;  VkDeviceMemory stagingBufferMemory;    VkDeviceSize imageSize = texWidth \* texHeight \* 4;  m\_pApp->CreateBuffer(imageSize,  VK\_BUFFER\_USAGE\_TRANSFER\_SRC\_BIT,  (VK\_MEMORY\_PROPERTY\_HOST\_VISIBLE\_BIT |  VK\_MEMORY\_PROPERTY\_HOST\_COHERENT\_BIT),  stagingBuffer,  stagingBufferMemory);  …  } |

The 3rd step is to copy the pixel content to the staging buffer using out helper method UpdateMemory() in VulkanApp class. Once the buffer is updated we free up the memory allocated by the STB library for the pixel content.

|  |
| --- |
| void Quad::CreateTextureImage(string filename)  {    …  …    // 3: Copy the pixel content to the staging buffer object  m\_pApp->UpdateMemory(stagingBufferMemory, 0,  imageSize, 0, pPixels);  stbi\_image\_free(pPixels);  } |

The 4th step is to create an image object that is only accessible only in the GPU. To do this we need to create a new helper method similar to CreateBuffer(). The following CreateImage() helper method resembles a lot like the CreateBuffer() method except it includes details to setup the 2D or 3D properties of the image.

|  |
| --- |
| bool VulkanApp::CreateImage(uint32\_t width, uint32\_t height,  VkFormat format, VkImageTiling tiling,  VkImageUsageFlags usage, VkMemoryPropertyFlags properties,  VkImage& image, VkDeviceMemory& imageMemory)  {  bool result = true;  VkImageCreateInfo imageInfo = {};  imageInfo.sType = VK\_STRUCTURE\_TYPE\_IMAGE\_CREATE\_INFO;  imageInfo.imageType = VK\_IMAGE\_TYPE\_2D;  imageInfo.extent.width = width;  imageInfo.extent.height = height;  imageInfo.extent.depth = 1;  imageInfo.mipLevels = 1;  imageInfo.arrayLayers = 1;  imageInfo.format = format;  imageInfo.tiling = tiling;  imageInfo.initialLayout = VK\_IMAGE\_LAYOUT\_UNDEFINED;  imageInfo.usage = usage;  imageInfo.samples = VK\_SAMPLE\_COUNT\_1\_BIT;  imageInfo.sharingMode = VK\_SHARING\_MODE\_EXCLUSIVE;  if (vkCreateImage(m\_hDevice, &imageInfo, nullptr, &image)  == VK\_SUCCESS)  {  VkMemoryRequirements memRequirements;  vkGetImageMemoryRequirements(m\_hDevice, image, &memRequirements);  VkMemoryAllocateInfo allocInfo = {};  allocInfo.sType = VK\_STRUCTURE\_TYPE\_MEMORY\_ALLOCATE\_INFO;  allocInfo.allocationSize = memRequirements.size;  allocInfo.memoryTypeIndex = GetMemoryTypeIndex(  properties, memRequirements.memoryTypeBits);  if (vkAllocateMemory(m\_hDevice, &allocInfo,  nullptr, &imageMemory) != VK\_SUCCESS)  {  LogError("vkAllocateMemory() failed!");  }  vkBindImageMemory(m\_hDevice, image, imageMemory, 0);  }  else  {  LogError("vkCreateImage() failed!");  }  return (result);  } |

Now call the helper method to create the texture image object and the device memory object. The format of the image is set to 4 byte un-normalized RGBA format. The tiling mode is set to ‘optimal’ mode for optimal memory access in the GPU. The usage flag VK\_IMAGE\_USAGE\_TRANSFER\_DST\_BIT indicate that the image can be used as a destination of a transfer command to copy data from other resource and the usage flag VK\_IMAGE\_USAGE\_SAMPLED\_BIT indicate that the image can be sampled by a shader. VK\_MEMORY\_PROPERTY\_DEVICE\_LOCAL\_BIT indicate that the memory allocated can reside on a memory that is most efficient for GPU.

The following method returns the image object handle (m\_textureImage) and memory object handle (m\_textureImageMemory)

|  |
| --- |
| void Quad::CreateTextureImage(string filename)  {    …  …  // 4: Create an image object that is only accessible side  // on the GPU  bool result = m\_pApp->CreateImage(texWidth, texHeight,  VK\_FORMAT\_R8G8B8A8\_UNORM,  VK\_IMAGE\_TILING\_OPTIMAL,  (VK\_IMAGE\_USAGE\_TRANSFER\_DST\_BIT |  VK\_IMAGE\_USAGE\_SAMPLED\_BIT),  VK\_MEMORY\_PROPERTY\_DEVICE\_LOCAL\_BIT,  m\_textureImage, m\_textureImageMemory);  } |

The 5th step requires setting up a command buffer to copy the staging buffer to the image object. This is a multi-step operation:

1. Set the layout of the image object suitable to use as a destination in the copy command. This can be done using vkCmdPipelineBarrier() API.
2. Call vkCmdCopyBufferToImage() to copy the staging buffer to the image object.
3. Set the layout of the image object suitable to use in the shader.

|  |
| --- |
| void Quad::CreateTextureImage(string filename)  {    …  …    // 5: Setup a command buffer to copy the staging  // buffer to the image object.  TransitionImageLayout(  m\_textureImage,  VK\_IMAGE\_LAYOUT\_UNDEFINED,  VK\_IMAGE\_LAYOUT\_TRANSFER\_DST\_OPTIMAL);  CopyBufferToImage(stagingBuffer, m\_textureImage,  static\_cast<uint32\_t>(texWidth),  static\_cast<uint32\_t>(texHeight));  TransitionImageLayout(  m\_textureImage,  VK\_IMAGE\_LAYOUT\_TRANSFER\_DST\_OPTIMAL,  VK\_IMAGE\_LAYOUT\_SHADER\_READ\_ONLY\_OPTIMAL);  } |

The TransitionImageLayout() method is a helper method in VulkanApp class whose main purpose is to set the pipeline barrier to a queue such that memory dependency are maintained between commands that were submitted before it and those submitted after it.

The following method creates a one time command buffer and setup the parameters to execute a pipeline barrier command to transition from an old image layout to the new image layout. Once the command buffer is generated it is submitted to the queue and wait for the queue to before idle.

|  |
| --- |
| bool Quad::TransitionImageLayout(VkImage image,  VkImageLayout oldLayout, VkImageLayout newLayout)  {  bool result = true;  VkCommandBuffer commandBuffer =  m\_pApp->BeginOneTimeCommandBuffer();  VkImageMemoryBarrier barrier = {};  barrier.sType = VK\_STRUCTURE\_TYPE\_IMAGE\_MEMORY\_BARRIER;  barrier.oldLayout = oldLayout;  barrier.newLayout = newLayout;  barrier.srcQueueFamilyIndex = VK\_QUEUE\_FAMILY\_IGNORED;  barrier.dstQueueFamilyIndex = VK\_QUEUE\_FAMILY\_IGNORED;  barrier.image = image;  barrier.subresourceRange.aspectMask = VK\_IMAGE\_ASPECT\_COLOR\_BIT;  barrier.subresourceRange.baseMipLevel = 0;  barrier.subresourceRange.levelCount = 1;  barrier.subresourceRange.baseArrayLayer = 0;  barrier.subresourceRange.layerCount = 1;  VkPipelineStageFlags sourceStage;  VkPipelineStageFlags destinationStage;  if (oldLayout == VK\_IMAGE\_LAYOUT\_UNDEFINED &&  newLayout == VK\_IMAGE\_LAYOUT\_TRANSFER\_DST\_OPTIMAL)  {  barrier.srcAccessMask = 0;  barrier.dstAccessMask = VK\_ACCESS\_TRANSFER\_WRITE\_BIT;  sourceStage = VK\_PIPELINE\_STAGE\_TOP\_OF\_PIPE\_BIT;  destinationStage = VK\_PIPELINE\_STAGE\_TRANSFER\_BIT;  }  else if (oldLayout == VK\_IMAGE\_LAYOUT\_TRANSFER\_DST\_OPTIMAL &&  newLayout == VK\_IMAGE\_LAYOUT\_SHADER\_READ\_ONLY\_OPTIMAL)  {  barrier.srcAccessMask = VK\_ACCESS\_TRANSFER\_WRITE\_BIT;  barrier.dstAccessMask = VK\_ACCESS\_SHADER\_READ\_BIT;  sourceStage = VK\_PIPELINE\_STAGE\_TRANSFER\_BIT;  destinationStage = VK\_PIPELINE\_STAGE\_FRAGMENT\_SHADER\_BIT;  }  else  {  result = false;  m\_pApp->LogError("Unsupported layout transition!");  }    if (result)  {  vkCmdPipelineBarrier(commandBuffer,  sourceStage, destinationStage,  0, 0, nullptr, 0, nullptr, 1, &barrier);  }  m\_pApp->EndOneTimeCommandBuffer(commandBuffer);  return (result);  } |

The BeginOneTimeCommandBuffer() helper method in VulkanAPI returns a handle to the command buffer to submit one time commands to the GPU, such as copy operation. The command buffer is allocated from the command pool which VulkanApp already creates by default.

|  |
| --- |
| VkCommandBuffer VulkanApp::BeginOneTimeCommandBuffer()  {  VkCommandBufferAllocateInfo allocInfo = {};  allocInfo.sType = VK\_STRUCTURE\_TYPE\_COMMAND\_BUFFER\_ALLOCATE\_INFO;  allocInfo.level = VK\_COMMAND\_BUFFER\_LEVEL\_PRIMARY;  allocInfo.commandPool = m\_hCommandPool;  allocInfo.commandBufferCount = 1;  // Allocate a single command buffer from the command pool  VkCommandBuffer commandBuffer;  vkAllocateCommandBuffers(m\_hDevice, &allocInfo, &commandBuffer);  // Begin the command buffer  VkCommandBufferBeginInfo beginInfo = {};  beginInfo.sType = VK\_STRUCTURE\_TYPE\_COMMAND\_BUFFER\_BEGIN\_INFO;  beginInfo.flags = VK\_COMMAND\_BUFFER\_USAGE\_ONE\_TIME\_SUBMIT\_BIT;  vkBeginCommandBuffer(commandBuffer, &beginInfo);  return commandBuffer;  } |

The EndOneTimeCommandBuffer() code ends the command buffer and submit it to the graphics queue. Note that both graphics queue and compute queue supports transfer operations by default, hence we can use the existing graphics queue to submit the copy operation.

|  |
| --- |
| void VulkanApp::EndOneTimeCommandBuffer(VkCommandBuffer commandBuffer)  {  // End the command buffer  vkEndCommandBuffer(commandBuffer);  // Submit to the graphics queue  VkSubmitInfo submitInfo = {};  submitInfo.sType = VK\_STRUCTURE\_TYPE\_SUBMIT\_INFO;  submitInfo.commandBufferCount = 1;  submitInfo.pCommandBuffers = &commandBuffer;  vkQueueSubmit(m\_hGraphicsQueue, 1, &submitInfo, VK\_NULL\_HANDLE);  // Wait for the graphics queue to idle  vkQueueWaitIdle(m\_hGraphicsQueue);  // Free the allocated one time commmand buffer  vkFreeCommandBuffers(m\_hDevice, m\_hCommandPool, 1, &commandBuffer);  } |

CopyBufferToImage() call in step 5 sets up the one time command buffer to submit the vkCopyBufferToImage() to the GPU to transfer the staging buffer content to the image object.

|  |
| --- |
| void Quad::CopyBufferToImage(VkBuffer buffer, VkImage image,  uint32\_t width, uint32\_t height)  {  // Begin one time command buffer  VkCommandBuffer commandBuffer = m\_pApp->BeginOneTimeCommandBuffer();  // Copy buffer to image  VkBufferImageCopy region = {};  region.bufferOffset = 0;  region.bufferRowLength = 0;  region.bufferImageHeight = 0;  region.imageSubresource.aspectMask = VK\_IMAGE\_ASPECT\_COLOR\_BIT;  region.imageSubresource.mipLevel = 0;  region.imageSubresource.baseArrayLayer = 0;  region.imageSubresource.layerCount = 1;  region.imageOffset = { 0, 0, 0 };  region.imageExtent = { width, height, 1};  vkCmdCopyBufferToImage(commandBuffer, buffer, image,  VK\_IMAGE\_LAYOUT\_TRANSFER\_DST\_OPTIMAL, 1, &region);  // End one time command buffer  m\_pApp->EndOneTimeCommandBuffer(commandBuffer);  } |

In step 6, last step in the texture image creation process we destroy the staging buffer and the device allocation after executing the copy call.

|  |
| --- |
| void Quad::CreateTextureImage(string filename)  {    …  …  // 6: Destroy the staging buffer object  // and its associated device memory object  vkDestroyBuffer(m\_hDevice, stagingBuffer, nullptr);  vkFreeMemory(m\_hDevice, stagingBufferMemory, nullptr);  } |

The next step in Quad initialization routine is to create an image view for the newly created image object to bind the texture to the shader pipeline. This can be done using the following method in Quad class.

Add the following member in the Quad class.

|  |
| --- |
| VkImageView m\_textureImageView; |

The vkCreateImageView() method returns a handle to the VkImageView object. The viewInfo data structure defines the image object to use to create the image view, the type of view (ie., 2D), format of the view etc. The output of this method is assigned to m\_textureImageView.

|  |
| --- |
| VkImageView Quad::CreateImageView(VkImage image, VkFormat format)  {  VkImageViewCreateInfo viewInfo = {};  viewInfo.sType = VK\_STRUCTURE\_TYPE\_IMAGE\_VIEW\_CREATE\_INFO;  viewInfo.image = image;  viewInfo.viewType = VK\_IMAGE\_VIEW\_TYPE\_2D;  viewInfo.format = format;  viewInfo.subresourceRange.aspectMask = VK\_IMAGE\_ASPECT\_COLOR\_BIT;  viewInfo.subresourceRange.baseMipLevel = 0;  viewInfo.subresourceRange.levelCount = 1;  viewInfo.subresourceRange.baseArrayLayer = 0;  viewInfo.subresourceRange.layerCount = 1;  VkImageView imageView = 0;  if (vkCreateImageView(m\_hDevice, &viewInfo, nullptr, &imageView) != VK\_SUCCESS)  {  m\_pApp->LogError("vkCreateImageView() failed!");  }  return imageView;  } |

This completes the Quad::Init() implementation. Once the Init() method is called for each Quad object, we need to create a descriptor set for the quad object using the uniform buffer and the quad’s image view.

The following method does this by allocating a descriptor set from the descriptor pool. The newly allocated descriptor set is updated with the details of the uniform buffer and the image view.

|  |
| --- |
| VkDescriptorSet Canvas2DApp::CreateDescriptorSet(VkImageView imageView)  {  bool result = true;  // Allocate descriptor set  VkDescriptorSetAllocateInfo allocInfo = {};  allocInfo.sType = VK\_STRUCTURE\_TYPE\_DESCRIPTOR\_SET\_ALLOCATE\_INFO;  allocInfo.descriptorPool = m\_descriptorPool;  allocInfo.descriptorSetCount = 1;  allocInfo.pSetLayouts = &m\_descSetLayout;  VkDescriptorSet descSet = 0;  if (vkAllocateDescriptorSets(m\_hDevice, &allocInfo, &descSet)  == VK\_SUCCESS)  {  // Setup buffer info for uniform buffer object  VkDescriptorBufferInfo bufferInfo = {};  bufferInfo.buffer = m\_uniformBuffer;  bufferInfo.offset = 0;  bufferInfo.range = sizeof(UniformBufferObject);  // Setup image info for the texture  VkDescriptorImageInfo imageInfo = {};  imageInfo.imageLayout =  VK\_IMAGE\_LAYOUT\_SHADER\_READ\_ONLY\_OPTIMAL;  imageInfo.imageView = imageView;  imageInfo.sampler = m\_textureSampler;  VkWriteDescriptorSet descriptorWrites[2] = {};  // Setup descriptor write for buffer info  descriptorWrites[0].sType =  VK\_STRUCTURE\_TYPE\_WRITE\_DESCRIPTOR\_SET;  descriptorWrites[0].dstSet = descSet;  descriptorWrites[0].dstBinding = 0;  descriptorWrites[0].dstArrayElement = 0;  descriptorWrites[0].descriptorType =  VK\_DESCRIPTOR\_TYPE\_UNIFORM\_BUFFER;  descriptorWrites[0].descriptorCount = 1;  descriptorWrites[0].pBufferInfo = &bufferInfo;  // Setup descriptor write for image info  descriptorWrites[1].sType =  VK\_STRUCTURE\_TYPE\_WRITE\_DESCRIPTOR\_SET;  descriptorWrites[1].dstSet = descSet;  descriptorWrites[1].dstBinding = 1;  descriptorWrites[1].dstArrayElement = 0;  descriptorWrites[1].descriptorType =  VK\_DESCRIPTOR\_TYPE\_COMBINED\_IMAGE\_SAMPLER;  descriptorWrites[1].descriptorCount = 1;  descriptorWrites[1].pImageInfo = &imageInfo;  // Update the descriptor set  vkUpdateDescriptorSets(  m\_hDevice, 2, descriptorWrites, 0, nullptr);  }  else  {  LogError("vkAllocateDescriptorSets() failed!");  }  // Return the newly created descriptor set  return (descSet);  } |

The newly created descriptor set is stored in the Quad object as shown below.

|  |
| --- |
| void Canvas2DApp::CreateImageTiles()  {  …  for (float i = 0; i < m\_numTotalCols; i++)  {  for (float j = 0; j < m\_numVisibleRows; j++)  {  if (idx < m\_numImageFiles)  {  …  …  …    // Create and set the descriptor set  m\_quad[idx].SetDescriptorSet(  CreateDescriptorSet(  m\_quad[idx].GetImageView()));  idx++;  }  }  }  } |

### Graphics Pipeline

The graphics pipeline setup for this chapter looks very similar to the last chapter with some key changes highlighted below:

Key changes include:

1. New vertex and fragment shader to render the QUAD. We will review them shortly.
2. Setup the pipeline vertex input state using the QUAD’s vertex input attribute and binding information
3. Pipeline layout using the descriptor layout information

|  |
| --- |
| bool Canvas2DApp::CreateGraphicsPipeline()  {  bool result = true;  // Compile the vertex shader  VkShaderModule vertShader =  CreateShader(m\_hDevice,"./source/shaders/QuadVert.spv");  ...  ...    // Compile the fragment shader  VkShaderModule fragShader =  CreateShader(m\_hDevice,"./source/shaders/QuadFrag.spv");  ...  ...  // Setup the vertex input  VkPipelineVertexInputStateCreateInfo vertexInputInfo = {};  vertexInputInfo.sType =  VK\_STRUCTURE\_TYPE\_PIPELINE\_VERTEX\_INPUT\_STATE\_CREATE\_INFO;  vertexInputInfo.vertexBindingDescriptionCount = 1;  vertexInputInfo.vertexAttributeDescriptionCount =  m\_quad[0].GetVertexInputAttributeDescCount();  vertexInputInfo.pVertexBindingDescriptions =  m\_quad[0].GetVertexInputBindingDesc();  vertexInputInfo.pVertexAttributeDescriptions =  m\_quad[0].GetVertexInputAttributeDesc();  ...  ...  // Create pipeline layout  VkPipelineLayoutCreateInfo pipelineLayoutInfo = {};  pipelineLayoutInfo.sType =  VK\_STRUCTURE\_TYPE\_PIPELINE\_LAYOUT\_CREATE\_INFO;  pipelineLayoutInfo.setLayoutCount = 1;  pipelineLayoutInfo.pushConstantRangeCount = 0;  pipelineLayoutInfo.pSetLayouts = &m\_descSetLayout;  VkResult vkResult = vkCreatePipelineLayout(  m\_hDevice, &pipelineLayoutInfo, nullptr,  &m\_canvas2DPipelineLayout);  if (vkResult != VK\_SUCCESS)  {  LogError("vkCreatePipelineLayout() failed!");  result = false;  }  else  {  // Create graphics pipeline  VkGraphicsPipelineCreateInfo pipelineInfo = {};  …  pipelineInfo.stageCount = 2;  pipelineInfo.pStages = shaderStages;  pipelineInfo.pVertexInputState = &vertexInputInfo;  …  …  …  …  …  vkResult = vkCreateGraphicsPipelines(  m\_hDevice, VK\_NULL\_HANDLE, 1,  &pipelineInfo, nullptr, &m\_canvas2DGraphicsPipeline);  if (vkResult != VK\_SUCCESS)  {  LogError("vkCreateGraphicsPipelines() failed!");  result = false;  }  }  // Cleanup  vkDestroyShaderModule(m\_hDevice, fragShader, nullptr);  vkDestroyShaderModule(m\_hDevice, vertShader, nullptr);  return (result);  } |

### Shaders

The vertex shader to render the quad is shown below with key changes highlighted in red.

* We define the layout for the uniform buffer object as shown below. The binding slot is set to 0 as indicated in the descriptor.
* The layout for each of the input and output are defined as shown below.
* Finally we add the offset to the vertex position, which transforms the entire quad by offset.x and offset.y

|  |
| --- |
| // Filename: Quad.vert  #version 450  #extension GL\_ARB\_separate\_shader\_objects : enable  layout (binding = 0) uniform UniformBufferObject  {  vec2 offset;  } ubo;  layout(location = 0) in vec2 inPosition;  layout(location = 1) in vec3 inColor;  layout(location = 2) in vec2 inTexCoord;  layout(location = 0) out vec3 fragColor;  layout(location = 1) out vec2 fragTexCoord;  out gl\_PerVertex  {  vec4 gl\_Position;  };  void main()  {  vec2 pos = inPosition;  pos += ubo.offset;  gl\_Position = vec4(pos, 0.0, 1.0);  fragColor = inColor;  fragTexCoord = inTexCoord;  } |

The fragment shader is shown below with key highlights in red.

* The texture sampler layout is defined as described in the descriptor set
* The additional texture coordinate value outputted from vertex shader is defined as shader input as shown below.
* The output color is set to the sampled texture color at the given texture coordinate from the texture.

|  |
| --- |
| // Filename: Quad.frag  #version 450  #extension GL\_ARB\_separate\_shader\_objects : enable  layout(binding = 1) uniform sampler2D texSampler;  layout(location = 0) in vec3 fragColor;  layout(location = 1) in vec2 fragTexCoord;  layout(location = 0) out vec4 outColor;  void main()  {  // Set texture color as fragment output  outColor = texture(texSampler, fragTexCoord);  } |

### Building Command Buffers

The steps to build the command buffer follows the same flow as in chapter 2, where Step 1 to 3 are identically same as chapter 2. In step 4, we need to bind the vertex buffer and descriptor set for each of the QUAD before issuing a draw command. Note that the vertex buffer for each quad contains the QUAD’s offset and size of quad defined in the normalized device coordinate. Additionally the descriptor set binds the information about the uniform buffer accessible at the vertex shader and Quad’s texture accessible in the fragment shader.

|  |
| --- |
| bool Canvas2DApp::BuildCommandBuffers()  {  bool result = true;  // The following code records the commands to draw  // a quad in a blue background  …  …  …  // For each command buffers in the command buffer list  for (…)  {    // Step 1: Begin command buffer  …  // Step 2: Begin render pass  …  // Step 3: Bind graphics pipeline  …  // Step 4: Draw all textured quads  uint32\_t c = 0;  for (int x = 0; x < m\_numTotalCols; x++)  {  for (int y = 0; y < m\_numVisibleRows; y++)  {  if (c < m\_numImageFiles)  {  vkCmdBindDescriptorSets(  m\_hCommandBufferList[i],  VK\_PIPELINE\_BIND\_POINT\_GRAPHICS,  m\_canvas2DPipelineLayout, 0, 1,  m\_quad[c].GetDescriptorSet(), 0, nullptr);  VkBuffer vertexBuffers[] = {  m\_quad[c].GetVertexBuffer() };  VkDeviceSize offsets[] = { 0 };    vkCmdBindVertexBuffers(  m\_hCommandBufferList[i], 0, 1,  vertexBuffers, offsets);  vkCmdDraw(m\_hCommandBufferList[i], 4, 1, 0, 0);  c++;  }  }  }  // Step 5: End the Render pass  …  // Step 6: End the Command buffer  …  }  return (result);  } |

## Render

Once the command buffer is built, we just need to submit them on each frame render to render the output. All we need to do in the Canvas2DApp::Render() method is to call the VulkanApp:Render() method to submit the command buffer to the graphics queue.

|  |
| --- |
| bool Canvas2DApp::Render()  {  return VulkanApp::Render();  } |

## Update

The Update() method is called before the Render() call, hence we can update the offset value in the uniform buffer to scroll the tiles horizontally.

|  |
| --- |
| bool Canvas2DApp::Update()  {  // Update the uniform buffer with the new x offset  UpdateUniformBuffer();  return true;  } |

The following method sets the updated x offset in a host data structure on every frame update and using the UpdateMemory() helper method it update the uniform buffer memory object.

|  |
| --- |
| void Canvas2DApp::UpdateUniformBuffer()  {  UniformBufferObject ubo = {};  // Set the x offset to the scroll delta  ubo.offset.x = (float)(m\_scrollDelta);  ubo.offset.y = 0;  // Update the scroll delta with a fraction  m\_scrollDelta -= m\_tmpScrollDelta / 5000.0f;  // When the scroll reached the end then reverse  // the scrolling direction  float windowWidth = (float)m\_windowDim.width;  float col = m\_numTotalCols\*(windowWidth / m\_numVisibleCols);  col = (col - windowWidth) / windowWidth \* 2.0f;  if ((m\_scrollDelta <= -col) || (m\_scrollDelta > 0))  {  m\_tmpScrollDelta = -m\_tmpScrollDelta;  }    // Update the uniform buffer memory object with the new offset value  UpdateMemory(m\_uniformBufferMemory, 0, sizeof(ubo), 0, &ubo);  } |

## Close

In the application’s Close() method we need to clean up all Vulkan objects we created for each Quad as well as destroy the texture sampler and descriptor layout object managed in the Canvas2DApp class.

|  |
| --- |
| void Canvas2DApp::Close()  {  // Clean up Vulkan objects in each instance of the quad  for (uint32\_t i = 0; i < m\_numImageFiles; i++)  {  m\_quad[i].Cleanup();  }  // Destroy the texture sampler  vkDestroySampler(m\_hDevice, m\_textureSampler, nullptr);  // Destroy the descriptor set layout  vkDestroyDescriptorSetLayout(m\_hDevice, m\_descSetLayout, nullptr);  } |

The list of vulkan objects to destroy in Quad object are shown below.

|  |
| --- |
| void Quad::Cleanup()  {  if (m\_vertexBuffer != 0)  {  vkDestroyBuffer(m\_hDevice, m\_vertexBuffer, nullptr);  m\_vertexBuffer = 0;  }  if (m\_vertexBufferMemory != 0)  {  vkFreeMemory(m\_hDevice, m\_vertexBufferMemory, nullptr);  m\_vertexBufferMemory = 0;  }  if (m\_textureImageView)  vkDestroyImageView(m\_hDevice, m\_textureImageView, nullptr);  if (m\_textureImage)  vkDestroyImage(m\_hDevice, m\_textureImage, nullptr);  if (m\_textureImageMemory)  vkFreeMemory(m\_hDevice, m\_textureImageMemory, nullptr);  } |

## Application Output

The following screenshot shows the application output at 3x3, 2x2 and 5x5 scrolling tile setup:

You can set the tiling configuration in Canvas2DApp::CreateImageTiles() method by setting the m\_numVisibleCols & m\_numVisibleRows member variables.

|  |  |
| --- | --- |
| 2x2 scrolling tiles | 3x3 scrolling tiles |
| 4x4 scrolling tiles | 5x5 scrolling tiles |

## Summary

In this chapter we transformed the Chapter 2 example that renders a single triangle into an application to preview images as scrolling tiles. We learned how to use vertex buffer to create primitives (quad) to draw, how to fill a primitive with a texture and how to send data from the CPU to GPU using uniform buffers. In the next chapter we will learn about building an application to preview the media content in a 3D terrain.