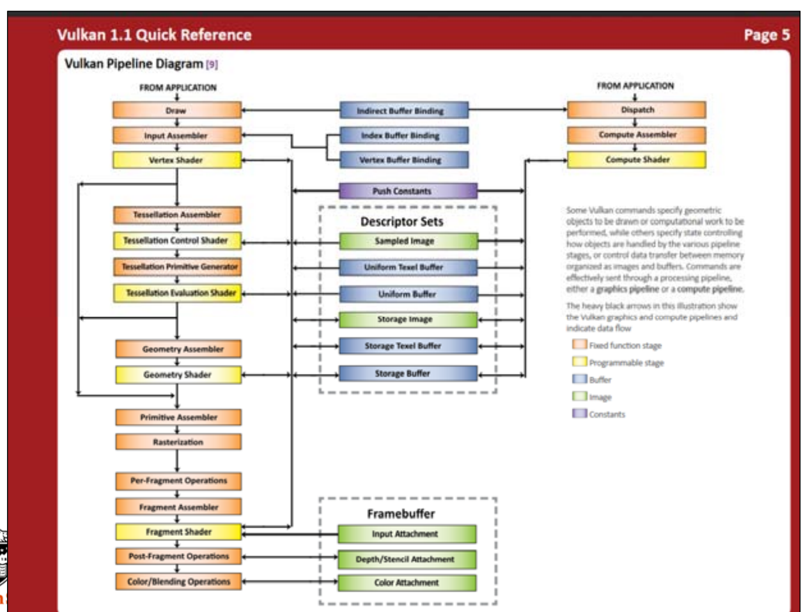
Data Buffers

Oregon State University
Mike Bailey
mjb@cs.oregonstate.edu

From the Quick Reference Card



Terminology Issues

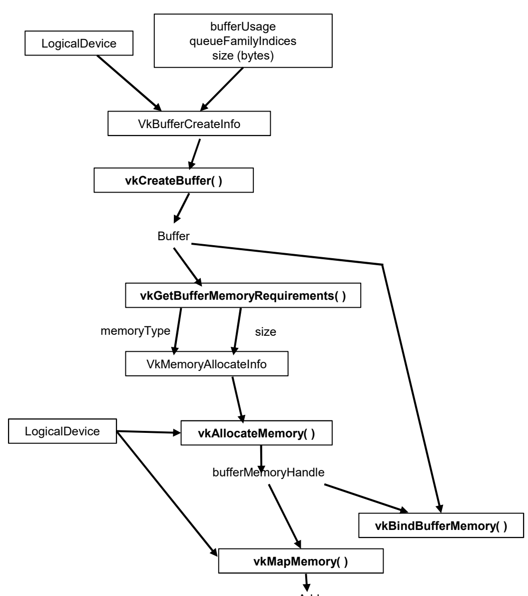
A **Data Buffer** is just a group of contiguous bytes in GPU memory. They have no inherent meaning. The data that is stored there is whatever you want it to be. (This is sometimes called a “Binary Large Object”, or “BLOB”.)

It is up to you to be sure that the writer and the reader pf the Data Buffer are interpreting the bytes in the same way!

Vulkan calls these things “Buffers”. But, Vulkan calls other things “Buffers”, too, such as Texture Buffers and Command Buffers. So, I have taken to calling these things “Data Buffers” and have even gone to far as to override some of Vulkan’s own terminology:

```
typedef VkBuffer        VkDataBuffer;
```


Vulkan: Buffers



Vulkan: Creating a Data Buffer

5

```
VkBufferCreateInfo  vbci;
        vbci.sType = VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO;
        vbci.pNext = nullptr;
        vbci.flags = 0;
        vbci.size = << buffer size in bytes >>
        vbci.usage = <<or'ed bits of: >>
                VK_USAGE_TRANSFER_SRC_BIT
                VK_USAGE_TRANSFER_DST_BIT
                VK_USAGE_UNIFORM_TEXEL_BUFFER_BIT
                VK_USAGE_STORAGE_TEXEL_BUFFER_BIT
                VK_USAGE_UNIFORM_BUFFER_BIT
                VK_USAGE_STORAGE_BUFFER_BIT
                VK_USAGE_INDEX_BUFFER_BIT
                VK_USAGE_VERTEX_BUFFER_BIT
                VK_USAGE_INDIRECT_BUFFER_BIT
        vbci.sharingMode = << one of: >>
                VK_SHARING_MODE_EXCLUSIVE
                VK_SHARING_MODE_CONCURRENT
        vbci.queueFamilyIndexCount = 0;
        vbci.pQueueFamilyIndices = (const iont32_t) nullptr;

VkBuffer  Buffer;

result = vkCreateBuffer ( LogicalDevice, IN &vbci, PALLOCATOR,  OUT &Buffer );
```

Doesn't actually allocate memory – just creates a **VkBuffer** data structure

Oregon State University
Computer Graphics

mjb – September 17, 2018

Vulkan: Allocating Memory for a Buffer, Binding a Buffer to Memory, and Writing to the Buffer

6

```
VkMemoryRequirements                vmr;
result = vkGetBufferMemoryRequirements( LogicalDevice, Buffer, OUT &vmr );


VkMemoryAllocateInfo                vmai;
        vmai.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
        vmai.pNext = nullptr;
        vmai.flags = 0;
        vmai.allocationSize = vmr.size;
        vmai.memoryTypeIndex = FindMemoryThatIsHostVisible( );

. . .

VkDeviceMemory                      vdm;
result = vkAllocateMemory( LogicalDevice, IN &vmai, PALLOCATOR,  OUT &vdm );

result = vkBindBufferMemory( LogicalDevice, Buffer, IN vdm, 0 );        // 0 is the offset

. . .

result = vkMapMemory( LogicalDevice, IN vdm, 0, VK_WHOLE_SIZE, 0, &ptr );

        << do the memory copy >>

result = vkUnmapMemory( LogicalDevice, IN vdm );
```

Oregon State University
Computer Graphics

mjb – September 17, 2018

Finding the Right Type of Memory

7

```
int
FindMemoryThatIsHostVisible( )
{
	VkPhysicalDeviceMemoryProperties	vpdmp;
	vkGetPhysicalDeviceMemoryProperties( PhysicalDevice, OUT &vpdmp );
	for( unsigned int i = 0; i < vpdmp.memoryTypeCount; i++ )
	{
		VkMemoryType vmt = vpdmp.memoryTypes[ i ];
		if( ( vmt.propertyFlags & VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT ) != 0 )
		{
			return i;
		}
	}
	return -1;
}
```

Oregon State University Computer Graphics

mjb – September 17, 2018

Finding the Right Type of Memory

8

```
int
FindMemoryThatIsDeviceLocal( )
{
	VkPhysicalDeviceMemoryProperties	vpdmp;
	vkGetPhysicalDeviceMemoryProperties( PhysicalDevice, OUT &vpdmp );
	for( unsigned int i = 0; i < vpdmp.memoryTypeCount; i++ )
	{
		VkMemoryType vmt = vpdmp.memoryTypes[ i ];
		if( ( vmt.propertyFlags & VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT ) != 0 )
		{
			return i;
		}
	}
	return -1;
}
```

Oregon State University Computer Graphics

mjb – September 17, 2018

Finding the Right Type of Memory


```
VkPhysicalDeviceMemoryProperties                    vpdmp;
vkGetPhysicalDeviceMemoryProperties( PhysicalDevice, OUT &vpdmp );
```

```
11 Memory Types:
Memory  0:
Memory  1:
Memory  2:
Memory  3:
Memory  4:
Memory  5:
Memory  6:
Memory  7:  DeviceLocal
Memory  8:  DeviceLocal
Memory  9:  HostVisible HostCoherent
Memory 10:  HostVisible HostCoherent HostCached

2 Memory Heaps:
Heap 0:  size = 0xb7c00000 DeviceLocal
Heap 1:  size = 0xfac00000
```


Something I've Found Useful

I find it handy to encapsulate buffer information in a struct:

```
typedef struct MyBuffer
{
        VkDataBuffer          buffer;
        VkDeviceMemory     vdm;
        VkDeviceSize          size;
} MyBuffer;

. . .

MyBuffer                        MyMatrixUniformBuffer;
```

It's the usual object-oriented benefit – you can pass around just one data-item and everyone can access whatever information they need.

Initializing a Data Buffer

It's the usual object-oriented benefit – you can pass around just one data-item and everyone can access whatever information they need.

```
VkResult
Init05DataBuffer( VkDeviceSize size, VkBufferUsageFlags usage, OUT MyBuffer * pMyBuffer )
{
. . .
        vbci.size = pMyBuffer->size = size;
. . .
        result = vkCreateBuffer ( LogicalDevice, IN &vbci, PALLOCATOR,  OUT &pMyBuffer->buffer );
. . .
        pMyBuffer->vdm = vdm;
. . .
}
```


Here's the C struct to hold some uniform variables


```
struct matBuf
{
        glm::mat4 uModelMatrix;
        glm::mat4 uViewMatrix;
        glm::mat4 uProjectionMatrix;
        glm::mat3 uNormalMatrix;
} Matrices;
```

Here's the shader code to access those uniform variables

```
layout( std140, set = 0, binding = 0 ) uniform matBuf
{
        mat4 uModelMatrix;
        mat4 uViewMatrix;
        mat4 uProjectionMatrix;
        mat4 uNormalMatrix;
} Matrices;
```

Filling those Uniform Variables

13

```
glm::vec3  eye(0.,0.,EYEDIST);
glm::vec3  look(0.,0.,0.);
glm::vec3  up(0.,1.,0.);

Matrices.uModelMatrix      = glm::mat4( );             // identity

Matrices.uViewMatrix       = glm::lookAt( eye, look, up );

Matrices.uProjectionMatrix = glm::perspective( FOV, (double)Width/(double)Height, 0.1, 1000. );
Matrices.uProjectionMatrix[1][1] *= -1.;

Matrices.uNormalMatrix = glm::inverseTranspose(  glm::mat3( Matrices.uModelMatrix )  );
```

Oregon State University Computer Graphics

mjb – September 17, 2018

The Parade of Data

14

CPU: ***MyBuffer   MyMatrixUniformBuffer;***

The MyBuffer does not hold any actual data itself. It just represents a container of data buffer information that will be used by Vulkan

```
VkResult
Init05DataBuffer( VkDeviceSize size, VkBufferUsageFlags usage, OUT MyBuffer * pMyBuffer )
{
. . .
      vbci.size = pMyBuffer->size = size;
. . .
      result = vkCreateBuffer ( LogicalDevice, IN &vbci, PALLOCATOR,  OUT &pMyBuffer->buffer );
. . .
      pMyBuffer->vdm = vdm;
. . .
}
```

This C struct is holding the actual data. It is writeable by the application.

Memory-mapped copy

CPU:

struct matBuf   Matrices;

```
glm::vec3  eye(0.,0.,EYEDIST);
glm::vec3  look(0.,0.,0.);
glm::vec3  up(0.,1.,0.);

Matrices.uModelMatrix      = glm::mat4( );             // identity

Matrices.uViewMatrix       = glm::lookAt( eye, look, up );

Matrices.uProjectionMatrix = glm::perspective( FOV, (double)Width/(double)Height, 0.1, 1000. );
Matrices.uProjectionMatrix[1][1] *= -1.;

Matrices.uNormalMatrix = glm::inverseTranspose(  glm::mat3( Matrices.uModelMatrix )  );
```

The Data Buffer in GPU memory is holding the actual data. It is readable by the shaders

uniform matBuf   Matrices;

```
layout( std140, set = 0, binding = 0 ) uniform matBuf
{
      mat4  uModelMatrix;
      mat4  uViewMatrix;
      mat4  uProjectionMatrix;
      mat4  uNormalMatrix;
} Matrices;
```

GPU:

There is one more step in here– **Descriptor Sets**. Here's a quick preview…

Oregon State University Computer Graphics

mjb – September 17, 2018

The Descriptor Set for the Buffer

15

We will come to **Descriptor Sets** later, but for now think of them as the link between the BLOB of uniform variables in GPU memory and the block of variable names in your shader programs.

```
VkDescriptorBufferInfo                    vdbi0;
            vdbi0.buffer = MyMatrixUniformBuffer.buffer;
            vdbi0.offset = 0;       // bytes
            vdbi0.range = sizeof(Matrices);
```

```
VkWriteDescriptorSet                      vwds0;
            // ds 0:
            vwds0.sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
            vwds0.pNext = nullptr;
            vwds0.dstSet = DescriptorSets[0];
            vwds0.dstBinding = 0;
            vwds0.dstArrayElement = 0;
            vwds0.descriptorCount = 1;
            vwds0.descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
            vwds0.pBufferInfo = &vdbi0;
            vwds0.pImageInfo = (VkDescriptorImageInfo *)nullptr;
```

```
vkUpdateDescriptorSets( LogicalDevice, 1, IN &vwds0, IN 0, (VkCopyDescriptorSet *)nullptr );
```

Oregon State
University
Computer Graphics

mjb – September 17, 2018

Filling the Data Buffer

16

```
typedef struct MyBuffer
{
       VkDataBuffer          buffer;
       VkDeviceMemory        vdm;
       VkDeviceSize          size;
} MyBuffer;

. . .

MyBuffer                   MyMatrixUniformBuffer;
```

```
Init05UniformBuffer( sizeof(Matrices),          &MyMatrixUniformBuffer );

Fill05DataBuffer( MyMatrixUniformBuffer,     (void *) &Matrices );
```

```
glm::vec3  eye(0.,0.,EYEDIST);
glm::vec3  look(0.,0.,0.);
glm::vec3  up(0.,1.,0.);

Matrices.uModelMatrix      = glm::mat4( );            // identity

Matrices.uViewMatrix       = glm::lookAt( eye, look, up );

Matrices.uProjectionMatrix = glm::perspective( FOV, (double)Width/(double)Height, 0.1, 1000. );
Matrices.uProjectionMatrix[1][1]  *= -1.;

Matrices.uNormalMatrix = glm::inverseTranspose(  glm::mat3( Matrices.uModelMatrix )  );
```

Oregon State
University
Computer Graphics

mjb – September 17, 2018

Creating and Filling the Data Buffer – the Details

17

```
VkResult
Init05DataBuffer( VkDeviceSize size, VkBufferUsageFlags usage, OUT MyBuffer * pMyBuffer )
{
	VkResult result = VK_SUCCESS;
	VkBufferCreateInfo  vbci;
		vbci.sType = VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO;
		vbci.pNext = nullptr;
		vbci.flags = 0;
		vbci.size = pMyBuffer->size = size;
		vbci.usage = usage;
		vbci.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
		vbci.queueFamilyIndexCount = 0;
		vbci.pQueueFamilyIndices = (const uint32_t *)nullptr;
	result = vkCreateBuffer ( LogicalDevice, IN &vbci, PALLOCATOR,  OUT &pMyBuffer->buffer );

	VkMemoryRequirements			vmr;
	vkGetBufferMemoryRequirements( LogicalDevice, IN pMyBuffer->buffer, OUT &vmr );		// fills vmr

	VkMemoryAllocateInfo			vmai;
		vmai.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
		vmai.pNext = nullptr;
		vmai.allocationSize = vmr.size;
		vmai.memoryTypeIndex = FindMemoryThatIsHostVisible( );

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

	result = vkBindBufferMemory( LogicalDevice, pMyBuffer->buffer, IN vdm, 0 );		// 0 is the offset
	return result;
}
```

Computer Graphics

Copy to GPU Memory via Memory Mapping

18

```
VkResult
Fill05DataBuffer( IN MyBuffer myBuffer, IN void * data )
{
	// the size of the data had better match the size that was used to Init the buffer!

	void * pGpuMemory;
	vkMapMemory( LogicalDevice, IN myBuffer.vdm, 0, VK_WHOLE_SIZE, 0, OUT &pGpuMemory );
							// 0 and 0 are offset and flags
	memcpy( pGpuMemory, data, (size_t)myBuffer.size );
	vkUnmapMemory( LogicalDevice, IN myBuffer.vdm );
	return VK_SUCCESS;
}
```

Remember – to Vulkan and GPU memory, these are ***just bits***. It is up to *you* to handle their meaning correctly.

Oregon State University
Computer Graphics