From the Quick Reference Card
A Vulkan **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 of 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:

```c
typedef VkBuffer VkDataBuffer;
```

**Vulkan: Buffers**

- `vkCreateBuffer( )`
  - bufferUsage
  - queueFamilyIndices
  - size (bytes)
- `vkGetBufferMemoryRequirements( )`
  - memoryType
  - size
- `vkAllocateMemory( )`
  - `LogicalDevice`
  - `VkMemoryAllocateInfo`
  - `bufferMemoryHandle`
- `vkBindBufferMemory( )`
  - `bufferMemoryHandle`
- `vkMapMemory( )`
  - `gpuAddress`
Vulkan: Creating a Data Buffer

```c
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 int32_t) nullptr;

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

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

```c
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 );
```
Finding the Right Type of Memory

```c
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;
}
```

Finding the Right Type of Memory

```c
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;
}
```
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.

```c
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 a C struct to hold some uniform variables

```c
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

```c
layout( std40, set = 0, binding = 0 ) uniform matBuf
{
  mat4 uModelMatrix;
  mat4 uViewMatrix;
  mat4 uProjectionMatrix;
  mat4 uNormalMatrix;
} Matrices;
```
Filling those Uniform Variables

```c
uint32_t Height, Width;
const double FOV = glm::radians(60.); // field-of-view angle
glm::vec3 eye(0.,0.,EYEDIST);
glm::vec3 look(0.,0.,0.);
glm::vec3 up(0.,1.,0.);
Matrices.uModelMatrix = glm::mat4(1); // 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.; // account for Vulkan's LH screen coordinate system
Matrices.uNormalMatrix = glm::inverseTranspose(glm::mat3(Matrices.uModelMatrix));
```

The Parade of Data

MyBuffer MyMatrixUniformBuffer;

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

```c
struct matBuf Matrices;
```

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

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

```c
uniform matBuf Matrices;
```
The Descriptor Set for the Buffer

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.

```c
VkDescriptorBufferInfo
    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);
```

Filling the Data Buffer

```c
def struct MyBuffer
{
    VkDeviceSize size;
    void *buffer;
    VkDeviceMemory vdm;
    MyBuffer;
    ...
    MyBuffer MyMatrixUniformBuffer;
}

Init05UniformBuffer( sizeof(Matrices),     &MyMatrixUniformBuffer );
Fill05DataBuffer( MyMatrixUniformBuffer,     (void *)&Matrices );
```

```c
// Matrices

matrices.unModelMatrix = glm::mat4( 1.0f ); // identity
matrices.unViewMatrix = glm::lookAt( eye, look, up );
matrices.unProjectionMatrix = glm::perspective( glm::radians(45.0f), (double)Width/(double)Height, 0.1f, 1000.0f );
matrices.unModelView = glm::mat4();
matrices.unNormalMat = glm::transpose( glm::mat4X( Matrices.unModelView, Matrices.unModelView ) );
```
Creating and Filling the Data Buffer – the Details

```c
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;
}
```

Creating and Filling the Data Buffer – the Details

```c
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.