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 sometimes 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;

This is probably a bad idea in the long run.

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

```c
VkBuffer Buffer;
VkBufferCreateInfo vbci;
    vbci.sType = VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO;
    vbci.pNext = nullptr;
    vbci.flags = 0;
    vbci.size = 1024; // buffer size in bytes
    vbci.usage = 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;
    vbci.sharingMode = VK_SHARING_MODE_CONCURRENT;
    vbci.queueFamilyIndexCount = 0;
    vbci.pQueueFamilyIndices = (const uint32_t*) nullptr;
result = vkCreateBuffer (LogicalDevice, IN &vbci, PALLOCATOR,  OUT &Buffer);

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.memoryTypeIndex = FindMemoryThatIsHostVisible();
    vmai.allocationSize = vmr.size;
    vmai.memoryTypeIndex = FindMemoryThatIsHostVisible();
    vmai.alignment = 64;
    vmai.queueFamilyIndexCount = 0;
    vmai.pQueueFamilyIndices = (const uint32_t*) nullptr;
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);
```
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

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

Sidebar: The Vulkan Memory Allocator (VMA)

The Vulkan Memory Allocator is a set of functions to simplify your view of allocating buffer memory. I don't have experience using it (yet), so I'm not in a position to confidently comment on it. But, I am including its github link here and a little sample code in case you want to take a peak.

https://github.com/GPUOpen-LibrariesAndSDKs/VulkanMemoryAllocator
This repository includes a smattering of documentation.

#define VMA_IMPLEMENTATION
#include "vk_mem_alloc.h"
. . .
VkBufferCreateInfo vbci; . . .
VmaAllocationCreateInfo vaci; . . .
vaci.physicalDevice = PhysicalDevice;
vaci.device = LogicalDevice;
vaci.usage = VMA_MEMORY_USAGE_GPU_ONLY;
VmaAllocator var;
vmaCreateAllocator( IN &vaci, OUT &var ); . . .
. . .
VkBuffer Buffer; . . .
vmaCreateBuffer( IN var, IN &vbci, IN &vaci, OUT &Buffer, OUT &van, nullptr );
. . .
void *mappedDataAddr;
vmaMapMemory( IN var, IN van, OUT &mappedDataAddr );
memcpy( mappedDataAddr, &MyData, sizeof(MyData) );
vmaUnmapMemory( IN var, IN van );

Sidebar: The Vulkan Memory Allocator (VMA)

typedef struct MyBuffer
{
    VkDataBuffer buffer;
    VkDeviceMemory vdm;
    VkDeviceSize size;
} MyBuffer;
. . .
MyBuffer MyMatrixUniformBuffer;
I find it handy to encapsulate buffer information in a struct:

Something I’ve Found Useful

It's the usual object-oriented benefit – you can pass around just one data-item and everyone can access whatever information they need. It also makes it impossible to accidentally associate the wrong VkDeviceMemory and/or VkDeviceSize with the wrong data buffer.
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 a C struct used by the Sample Code to hold some uniform variables

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

Here's the associated GLSL shader code to access those uniform variables

```
layout ( std140, set = 0, binding = 0 ) uniform matBuf Matrices;
```

Filling those Uniform Variables

```
const double FOV = glm::radians(60.);      // field-of-view angle in radians
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 ) );
```

This code assumes that this line:

```
#define    GLM_FORCE_RADIANS
```

is listed before GLM is included!

Filling the Data Buffer

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

The Parade of Buffer Data

```
MyBuffer MyMatrixUniformBuffer;
```

```
The MyBuffer does not hold any actual data itself. It just information about what is in the data buffer
```

```
This C struct is holding the original data, written by the application.
```

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

```
uniform matBuf Matrices;
```

Creating and Filling the Data Buffer – the Details

```
Init05DataBuffer( OUT MyBuffer* pMyBuffer ) {
    ... 
    vbci.size = pMyBuffer->size = size;
    ... 
    result = vkCreateBuffer ( LogicalDevice, IN &vbci, PALLOCATOR, OUT &pMyBuffer->buffer );
    ... 
    pMyBuffer->vdm = vdm;
    ... 
    result = vkBindBufferMemory ( LogicalDevice, pMyBuffer->buffer, IN vdm, OFFSET_ZERO );
    return result;
}
```
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.