A **Data Buffer** is just a group of contiguous bytes in GPU memory. They have not 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;
```

### Terminology Issues

#### Vulkan: Buffers

- `%vkCreateBuffer( )%`<br>`VkBufferCreateInfo`<br>`bufferUsage`<br>`queueFamilyIndices`<br>`size (bytes)`
- `%vkGetBufferMemoryRequirements( )%`<br>`Buffer`<br>`VkMemoryAllocateInfo`<br>`size = memoryType`<br>`vkAllocateMemory( )%`<br>`LogicalDevice`<br>`vkBindBufferMemory( )%`<br>`bufferMemoryHandle`<br>`vkMapMemory( )%`<br>`gpuAddress`<br>`vkUnmapMemory( )%`<br>`LogicalDevice`

---

```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
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 );
```

---

```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;
```

### Vulkan: Creating a Data Buffer

```c
result = FindMemoryThatIsHostVisible( )
```

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

```c
result = vkBindBufferMemory( LogicalDevice, Buffer, IN vdm, 0 );
```

---

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

---

```c<< do the memory copy >>
```

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

Something I’ve Found Useful

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

Initializing a Data Buffer

```c
Here’s the C struct to hold some uniform variables

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

Here’s the shader code to access those uniform variables

```c
layout (std140, set = 0, binding = 0) uniform matBuf
{
    mat4 uModelMatrix;
    mat4 uViewMatrix;
    mat4 uProjectionMatrix;
    mat4 uNormalMatrix;
};
```
Filling those Uniform Variables

```cpp
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 Parade of Data

```cpp
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.

The Descriptor Set for the Buffer

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

```cpp
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;
```

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

Creating and Filling the Data Buffer

```cpp
void 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

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

The Parade of Data

```cpp
MyBuffer MyMatrixUniformBuffer;
```

This MyBuffer that will represent the collection of data buffer information is not holding any actual data. This is used by Vulkan

Creating and Filling the Data Buffer

```cpp
void Init05UniformBuffer( sizeof(Matrices), OUT MyMatrixUniformBuffer ){
    Fill05DataBuffer( MyMatrixUniformBuffer, (void *) &Matrices );
}

void Fill05DataBuffer( IN MyMatrixUniformBuffer, 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 MyMatrixUniformBuffer.vdm, 0, VK_WHOLE_SIZE, 0, &pGpuMemory );
    // 0 and 0 are offset and flags
    memcpy( pGpuMemory, data, (size_t)MyMatrixUniformBuffer.size );
    vkUnmapMemory( LogicalDevice, IN MyMatrixUniformBuffer.vdm );
    return VK_SUCCESS;
}
```