Data Buffers

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 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:

typedef VkBuffer VkDataBuffer;

Vulkan: Creating a Data Buffer

Vulkan: Allocating Memory for a Buffer, Binding a Buffer to Memory, and Writing to the Buffer
Find Memory That Is Host Visible

```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

Find Memory That Is Device Local

```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

### 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

### 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:

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

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 the 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**

```glsl
layout (std140, set = 0, binding = 0) uniform matBuf;
{
    mat4 uModelMatrix;
    mat4 uViewMatrix;
    mat4 uProjectionMatrix;
    mat4 uNormalMatrix;
} Matrices;
```
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.uNormalMatrix = glm::inverseTransposed(glm::mat3(Matrices.uModelMatrix));
```

The Parade of Data

- **CPU:**
  - MyBuffer
  - MyMatrixUniformBuffer;

- **GPU:**
  - uniform matrix Matrices;

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

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

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);
```

The Descriptor Set for the Buffer

```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;
```

The Data Buffer

```cpp
Init05DataBuffer( sizeof(Matrices), &MyMatrixUniformBuffer );
Fill05DataBuffer( MyMatrixUniformBuffer, (void *) &Matrices );
```

Creating and Filling the Data Buffer – the Details

```cpp
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 );
    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
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;
}
```

Copy to GPU Memory via Memory Mapping

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