Some Vulkan commands specify geometric objects to be drawn or computational work to be performed, while others specify state controlling how objects are handled by the various pipeline stages, or control data transfer between memory organized as images and buffers. Commands are effectively sent through a processing pipeline, either a graphics pipeline or a compute pipeline.

The heavy black arrows in this illustration show the Vulkan graphics and compute pipelines and indicate data flow.
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:

```c
typedef VkBuffer VkDataBuffer;
```
Vulkan: Buffers

- **vkCreateBuffer( )**
  - LogicalDevice
  - VkBufferCreateInfo
    - bufferUsage
    - queueFamilyIndices
    - size (bytes)

- **vkGetBufferMemoryRequirements( )**
  - Buffer
  - VkMemoryAllocateInfo
    - memoryType
    - size

- **vkAllocateMemory( )**
  - LogicalDevice
  - VkMemoryAllocateInfo
    - bufferMemoryHandle

- **vkBindBufferMemory( )**
  - logicalDevice
  - bufferMemoryHandle

- **vkMapMemory( )**
  - logicalDevice
  - bufferMemoryHandle
  - gpgpuAddress
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 iont32_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

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

<table>
<thead>
<tr>
<th>Memory Type</th>
<th>Heap Size</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory 0</td>
<td></td>
<td>DeviceLocal</td>
</tr>
<tr>
<td>Memory 1</td>
<td></td>
<td>DeviceLocal</td>
</tr>
<tr>
<td>Memory 2</td>
<td></td>
<td>HostVisible HostCoherent</td>
</tr>
<tr>
<td>Memory 3</td>
<td></td>
<td>HostVisible HostCoherent</td>
</tr>
<tr>
<td>Memory 4</td>
<td></td>
<td>HostCached</td>
</tr>
<tr>
<td>Memory 5</td>
<td></td>
<td>DeviceLocal</td>
</tr>
<tr>
<td>Memory 6</td>
<td></td>
<td>HostVisible HostCoherent</td>
</tr>
<tr>
<td>Memory 7</td>
<td></td>
<td>HostCached</td>
</tr>
<tr>
<td>Memory 8</td>
<td></td>
<td>HostVisible HostCoherent</td>
</tr>
<tr>
<td>Memory 9</td>
<td></td>
<td>HostCached</td>
</tr>
<tr>
<td>Memory 10</td>
<td></td>
<td>HostVisible HostCoherent HostCached</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Memory Heaps</th>
<th>SIZE</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heap 0:</td>
<td>0xb7c00000</td>
<td>DeviceLocal</td>
</tr>
<tr>
<td>Heap 1:</td>
<td>0xfac00000</td>
<td></td>
</tr>
</tbody>
</table>
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;

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 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

```c
layout( std140, set = 0, binding = 0 ) uniform matBuf
{
    mat4 uModelMatrix;
    mat4 uViewMatrix;
    mat4 uProjectionMatrix;
    mat4 uNormalMatrix;
} Matrices;
```
```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 ) );
```
This C struct is holding the actual data. It is writeable by the application.

```c
struct matBuf
{
    glm::vec3 eye(0.0f, 0.0f, 0.0f);
    glm::vec3 look(0.0f, 0.0f, 0.0f);
    glm::vec3 up(0.1f, 0.0f, 0.0f);
    matBuf.
        Matrices.uModelMatrix = glm::mat4(1.0f); // identity
        Matrices.uViewMatrix = glm::lookAt(eye, look, up);
        Matrices.uProjectionMatrix = glm::perspective(FOV, (double)Width/(double)Height, 0.1f, 1000.0f);
        Matrices.uNormalMatrix = glm::inverseTranspose(glm::mat3(Matrices.uModelMatrix));
};
```

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
MyBuffer MyMatrixUniformBuffer;

VkResult
init05BufferData( VkDeviceSize size, VkBufferUsageFlags usage, OUT MyBuffer * pMyBuffer )
{

    ... vbc.size = pMyBuffer->size = size;
    ...
    result = vkCreateBuffer( LogicalDevice, IN &vbc, PALLOCATOR, OUT &pMyBuffer->buffer );
    ...
    pMyBuffer->vdm = vdm;
}
```

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

```c
uniform matBuf Matrices;

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

There is one more step in here—Descriptor Sets.

Here’s a quick preview…
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.

```cpp
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 );
```
Filling the Data Buffer

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

... MyBuffer MyMatrixUniformBuffer;

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

```c
glm::vec3 eye(0.0,0.0,EYEDIST);
glm::vec3 look(0.0,0.0);
glm::vec3 up(0.1,0.0);

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] = -1.0f;

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

```
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.