Data Buffers

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 so far as to extend some of Vulkan’s own terminology:

typedef VkBuffer VkDataBuffer;

This is probably a bad idea in the long run.
Creating a Vulkan Data Buffer

```cpp
VkBuffer Buffer;    // or "VkDataBuffer Buffer"
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;
result = vkCreateBuffer ( LogicalDevice, IN &vbci, PALLOCATOR, OUT &Buffer );
```

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

```cpp
VkMemoryRequirements
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

```cpp
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

```cpp
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

6 Memory Types:
Memory 0: DeviceLocal
Memory 1: HostVisible HostCoherent
Memory 2: HostVisible HostCoherent HostCached
Memory 4: DeviceLocal HostVisible HostCoherent
Memory 5: DeviceLocal

4 Memory Heaps:
Heap 0: size = 0x0d600000 DeviceLocal
Heap 1: size = 0x02000000 DeviceLocal
Heap 2: size = 0x002000000 DeviceLocal
Heap 3: size = 0x002000000 DeviceLocal

These are the numbers for the Nvidia A6000 cards

Memory-Mapped Copying to GPU Memory, Example I

```c
void *mappedDataAddr;
vkMapMemory( LogicalDevice, myBuffer.vdm, 0, VK_WHOLE_SIZE, 0, OUT (void *)&mappedDataAddr );
memcpy( mappedDataAddr, &VertexData, sizeof(VertexData) );
vkUnmapMemory( LogicalDevice, myBuffer.vdm );
```

Memory-Mapped Copying to GPU Memory, Example II

```c
struct vertex *vp;
vkMapMemory( LogicalDevice, IN myBuffer.vdm, 0, VK_WHOLE_SIZE, 0, OUT (void *)&vp );
for( int i = 0; i < numTrianglesInObjFile; i++ ) // number of triangles
    for( int j = 0; j < 3; j++ ) // 3 vertices per triangle
        { vp->position = glm::vec3( ... );
          vp->normal = glm::vec3( ... );
          vp->color = glm::vec3( ... );
          vp->texCoord = glm::vec2( ... );
          vp++;
        }
vkUnmapMemory( LogicalDevice, myBuffer.vdm );
```

Sidebar: The Vulkan Memory Allocator (VMA)

The Vulkan Memory Allocator is a set of functions to simplify your view of allocating buffer memory. I am including its github link here and a little sample code in case you want to take a peek.

[https://github.com/GPUOpen-LibrariesAndSDKs/VulkanMemoryAllocator](https://github.com/GPUOpen-LibrariesAndSDKs/VulkanMemoryAllocator)

This repository also includes a smattering of documentation.

See our class VMA noteset for more VMA details
Sidebar: The Vulkan Memory Allocator (VMA)

#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;

VmaAllocation van;

vmaCreateBuffer( IN var, IN &vbci, IN &vaci, OUT &Buffer, OUT &van, nullptr );

void *mappedDataAddr;

vmaMapMemory( var, van, OUT &mappedDataAddr );

memcpy( mappedDataAddr, &VertexData, sizeof(VertexData) );

vmaUnmapMemory( var, van );

See our class VMA noteset for more VMA details

 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.

InitO5DataBuffer( VkDeviceSize size, VkBufferUsageFlags usage, OUT MyBuffer * pMyBuffer )
{
  vbci.size = pMyBuffer->size = size;
  result = vkCreateBuffer ( LogicalDevice, IN &vbci, PALLOCATOR, OUT &pMyBuffer->buffer );
  pMyBuffer->vdm = vdm;
}

It's the usual object-oriented benefit – you can pass around just one data-item and everyone can access whatever information they need.

Something I've Found Useful

typedef struct MyBuffer {
  VkDataBuffer buffer;
  VkDeviceMemory vdm;
  VkDeviceSize size; // in bytes
} MyBuffer;

MyBuffer;

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.

Here are C/C++ structs used by the Sample Code to hold some uniform variables

The uNormal is set to:

 glm::inverseTranspose( uView * uSceneOrient * uModel )

In the vertex shader, each object vertex gets transformed by:

uProjection* uView * uSceneOrient * uModel

In the vertex shader, each surface normal vector gets transformed by the uNormal.
Filling those Uniform Variables

const float EYEDIST = 3.0f;
const double FOV = glm::radians(60); // field of view in radians
glm::vec3 eye(0., 0., EYEDIST);
glm::vec3 look(0., 0., 0.);
glm::vec3 up(0., 1., 0.);
Scene.uProjection = glm::perspective(FOV, (double)Width/(double)Height, 0.1, 1000);
Scene.uView = glm::lookAt(eye, look, up);
Scene.uSceneOrient = glm::mat4(1.);
Object.uModelOrient = glm::mat4(1.); // identity
Object.uNormal = glm::inverseTranspose(Scene.uView * Scene.uSceneOrient * Object.uModel);

This code assumes that this line:
#define GLM_FORCE_RADIANS
is listed before GLM is #included!

Filling the Data Buffer

struct MyBuffer
{ 
    VkDeviceSize size;
    VkBuffer buffer;
}

Init05UniformBuffer( sizeof(Object), OUT &MyObjectUniformBuffer );
Fill05DataBuffer( MyObjectUniformBuffer, IN (void*) &Object );

Creating and Filling the Data Buffer – the Details

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, 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.