Notes on documentation

* Need to manage memory allocation and command buffers

Devices:

Queues:

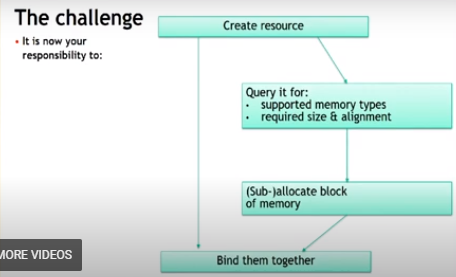
Command buffers:

Swap chains:

Creating window for rendering

* Vulkan does not include a native windowing system in its API
* Must use a separate library to create a window that Vulkan can interface with ex GLFW – this library also helps with handling the OS specific details to create a Vulkan surface

Memory management

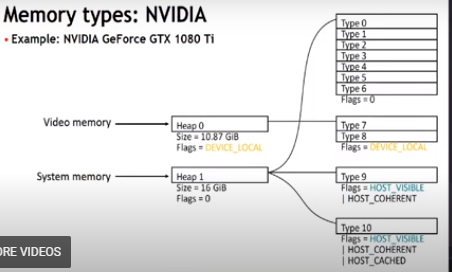


Memory types

Heaps: physical memory = VRAM, DRAM

Types: Describes the properties of the memory and which heap to use

*For example,* NVIDIA has:



Examples (flags that determine properties):

* DEVICE\_LOCAL:
  + Video memory. Fast access from GPU
  + No direct access from CPU. Mapping not possible
  + Good for resources written and read frequently by GPU
  + Good for resources uploaded once (immutable) or infrequently by CPU, read frequently by GPU, ex transfers vertex and index buffers
* HOST\_VISIBLE:
  + DRAM
  + System memory. Accessible to CPU – mapping possible.
  + Access from GPU possible but slow.
  + Across PCle bus, reads cached on GPU
  + Good for CPU-side (staging) copy of your resources – used as source of transfer.
  + Data written by CPU, read once by GPU (e.g. constant buffer) may work.
  + Large data read by GPU – place here as last resort.
  + Large data written and read by GPU – shouldn’t ever be here.
  + Hazard: Uncached. Writers may be write-combined.
* DEVICE\_LOCAL + HOST\_VISIBLE
  + Special pool of video memory.
  + Exposed on AMD only. 256 MiB
  + Fast access from GPU
  + Direct access by both CPU and GPU
    - Don’t need explicit transfers
    - Mapping possible
  + Good for resources updated frequently by CPU (dynamic), read by GPU.
  + Use as fallback if DEVICE\_LOCAL is small and oversubscribed.
  + Hazard: Driver will use this memory too.
  + Hazard: Uncached. Writers may be write-combined.
* HOST\_VISIBLE + HOST\_CACHED
  + System memory
  + CPU reads and writes cached (write-back). Fast reads from cpu
  + GPU access through PCle.
  + GPU reads snoop CPU cache.
  + Good for resources written by GPU, read by CPU.
    - Results of computations.
  + Direct access by both CPU and GPU.
    - No need to do explicit transfer.
  + Use for any resources read or accessed randomly on CPU.

*NOTE: APU (accelerate processing unit) – type of processor that combines a central processing unit (cpu) with a graphics processing unit (gpu). Designed for improving data transfer rates btwn two units. Most associated with AMD.*

Tips & Tricks for mem allocation:

*Vulkan allocates memory in memory blocks:*

* In old API’s the resource owns the memory.
  + Memory is allocated when creating the resource.
  + No control of where that memory is placed.
* In Vulkan the memory is allocated by itself.
  + Memory is allocated in blocks.
  + A single block can have many resources.
* Allows many new optimization techniques
  + Pool similar resources for tighter packing.
  + Keep object resources together to simplify streaming of that objects required resources.

*(Note:* ***a resource*** *refers to any data structure that can be used by the GPU, ex*

buffers *[linear chunks of memory],*

textures *[1d, 2d, 3d or cube maps],*

samplers *[state objects that describe how to access data from textures, including filtering, wrapping, and mipmapping params],*

shaders *[compiled code resource that runs on gpu],*

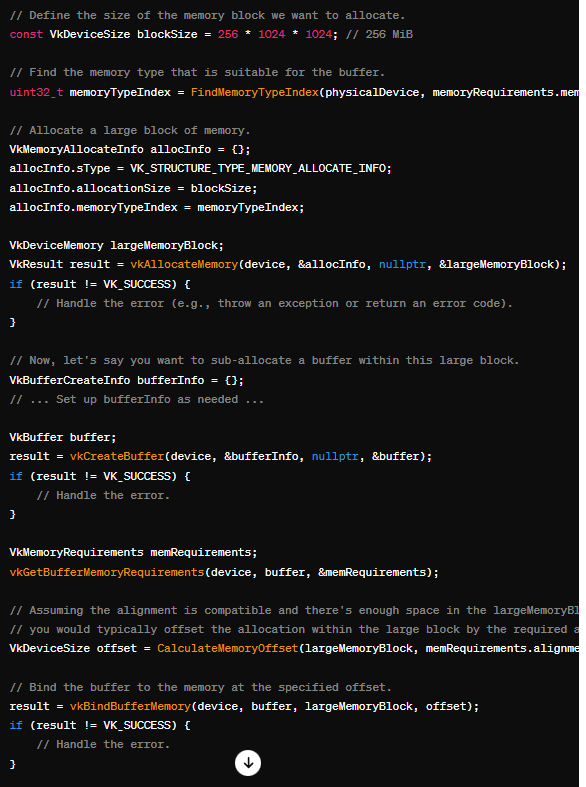
framebuffers *[collections of attachments that the gpu can render to, ex color, depth, stencil],*

renderpasses *[define a sequence of operations that happen during rendering, specifying how the attachments of a framebuffer are used through out the rendering process],*

descriptor *sets [sets of descriptors that provide the shaders with access to buffer and image resources. Essentially pointers to resources with some additional info about how the resource will be accessed.],* pipeline states *[define the fixed-function stages of the graphics pipeline, such as input assembly, rasterization, etc.], command buffers [sequences of commands that you record and submit to the gpu. They orchestrate the rendering process by using the other resources mentioned above.])*

*IMPORTANT:*

**Suballocation**

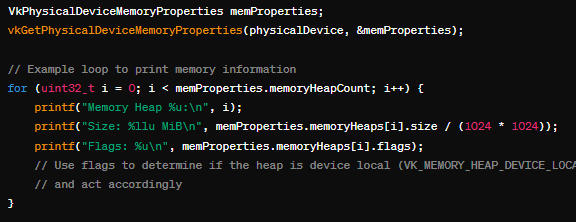
* Don’t allocate a separate memory block for each resource
  + Small limit on maximum number of allocations (ex, 4096).
* Allocate bigger blocks and sub-allocate ranges for your resources.
  + 256 MiB is good default block size.
  + For Heaps <= 1 GiB use smaller blocks. (e.g. heap size / 8).
  + This means you’ll manually manage the offsets within the block to handle multiple resources.­­
* Allocations are slow
  + Prefer not to allocate of free memory blocks during gameplay to avoid hitching (brief pauses/ stutters in the rendering frame).
  + If you need to, you can do it on background thread.

Ex:

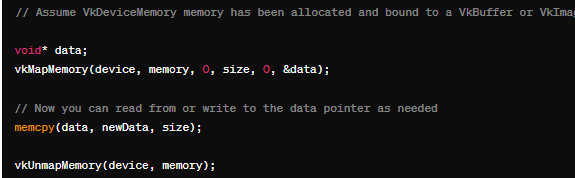
*Issues*

**Over-commitment**

Exceeding the maximum amount of physical video memory:

* Depends on driver:
  + Allocation may fail (VK\_ERROR\_OUT\_OF\_DEVICE\_MEMORY)
  + Allocation may succeed (VK\_SUCCESS).
    - Some blocks are silently migrated to system memory.
* Blocks may be migrated to system memory anyway.
  + Other applications can use video memory
  + Using blocks migrated to system memory on gpu degrades performance.
* Don’t use 100% of the heap
  + Leave room for implicit resources
  + (ex, 20% of DEVICE\_LOCAL, 33% of DEVICE\_LOCAL + HOST\_VISIBLE).
  + Vulkan allows you to query the gpu for its physical properties including the amount of available memory.
    - vkGetPhysicalDeviceMemoryProperties returns vkPhysicalDeviceMemoryProperties structure
    - gpu memory is typically divided into several heaps and each heap may contain one or more memory types. Heaps represent large pools of memory.
    - Some memory management algorithms can help to defragment memory or ensure that allocations are as contiguous as possible.
    - Ways to query mem with vulkan example: 

**MAPPING**

* Having entire memory block persistently mapped is generally ok.
  + Don’t need to unmap before using on gpu.
* Keeping many large memory blocks mapped may impact stability or performance f debugging tools.
* Mapping process:
  + 1. Allocate a block of mem from a specific type on the gpu
  + 2. Bind the resource to a portion of that memory. This step associates the resource but doesn’t allow you to access the memory from the cpu yet.
  + 3. **Map** the memory to the address space of the application using vkMapMemory. This provides you with a cpu-visible pointer to the memory.
  + 4. With the pointer from the mapping, read from or write to the memory.
  + 5. If you’re done with the memory, you unmap it using vkUnmapMemory – removes the cpu access to that memory.
  + Need to avoid race conditions and other with synchronization
  + Example of mapping: 

**Transfer**

*Another way of getting data to your queues.*

* Designed for efficient transfer via PCLe
  + Heavily underutilized queue.
  + Use it in parallel with 3D rendering, even asynchronously to rendering frames.
    - Good for texture streaming
  + Fastest way to copy across the PCle bus.
  + Does not use any compute or graphics hardware.
  + Do your transfers long before the data is needed on graphics queue.
* Transfer queue can be used for defragmenting.
  + Copy resource to a new address asynchronously (to graphics hardware) to rendering.
  + When done update next frames descriptor.
* Some hardware supports more than one queue
* Gpu to gpu transfers are much faster on graphics queue
* Always respect queue granularly for copies otherwise you may have corruption, crash or even deadlock.

**Aliasing**

* Bind same address to multiple resources
* Saves memory
* As long as you don’t use them simultaneously
* After the memory was used by different resource, treat your resource as unintialized

**Tips on resource creation**

* Don’t use images with TILING\_LINEAR – use TILING\_OPTIMAL
  + Copy images from/to buffers
* Minimize number of USAGE\_BITS used during resource creation.
* Avoid VK\_SHARING\_MODE\_CONCURRENT on render and depth target textures.
  + It disables optimizations
  + Prefer VK\_SHARING\_MODE\_EXCLUSIVE and do explicit queue family ownership transfer barriers.
* Use VK\_KHR\_image\_format\_list for mutable formats.

**Libraries**

AMD created Vulkan Memory Allocator

* MIT licence
* Single header integration
* Main idea is to pick the correct memory type and pick the right subblock

**DRAWING A TRIANGLE**

**Step 1**: Instance and physical device selection

* Must set up vulkan api through a vkInstance.
* After creating an instance, can query for Vulkan supported hardware and select one or more vkPhysicalDevice(s).
* Can query for properties like VRAM size and device capabilities to select desired devices

**Step 2:** Logical device and queue families

* **Queue**: an ordered sequence of commands that the gpu executes. When work is submitted to gpu, the commands form a queue to be processed
* After selecting the right hardware, need to create a vkDevice (logical device) – here I’ll describe more specifically which VkPhysicalDeviceFeatures I’ll use, like multi viewport rendering and 64 bit floats.
* Also need to specify which queue families I’ll use. Most operations performed with Vulkan, like draw commands and memory operations, are asynchronously executed by submitting them to a VkQueue.
* Queues are allocated from queue families, where each queue family supports a specific set of operations in its queues.

**Step 3:** Window surface and swap chain

* Windows can be created with native platform APIs or libraries like GLFW and SDL.
* Two more components are needed to render a window
  + 1. Window surface: vkSurfaceKHR
  + 2. Swap Chain: vkSwapchainKHR. (KHR means part of vulkan extension).
    - Swap chain is a collection of render targets. Its main purpose is to ensure that the image (2d collection of pixels) we’re currently rendering to is different from the one that is currently on screen. This is important to make sure only complete images are shown.
* Glfw library has a built-in function to deal with the platform specific details of this.

**Step 4:** Image views and framebuffers

* To draw an image acquired from the swap chain, must wrap it into a VkImageView and VKFrameBuffer.
* Image view: provides info how the pipeline should interpret the image
* Frame buffer: A logical data structure acting as a collection of pointers to image views.

**Step 5:** Render passes

* Render passes in vulkan describe the type of images that are used during rendering operations, how they will be used and how their contents should be treated. Ex solid color, single image, etc.

**Step 6:** Graphics pipeline

* Set up by creating VkPipeline object. It describes the configurable state of the graphics card, like viewport size and depth buffer and the programmable state using VkShaderModule objects.
* If you want to tweak things like change shader or slightly change vertex layout, then need to entirely recreate the graphics pipeline.

**Step 7:** Command pools and command buffers

* Command buffers: objects that record and store the sequence of operations you want the gpu to execute.
* Command pools: a container that holds and manages the memory from which command buffers are allocated. Each pool is associated with a specific pool family

**Step 8:** Main loop

* First acquire an image from swap chain with vkAcquireNextImageKHR.
* Then select the appropriate command buffer for that image and execute it with vkQueueSubmit.
* Return the image to the swap chain for presentation to the screen with vkQueuePresentKHR
* *Operations submitted to queues are asynchronous. Use semaphores to ensure correct order of execution.*

**EXTRA NOTES**

*Functions have a lower case vk prefix*

*Enumerations and structs have a Vk prefix and enumeration values have a VK\_ prefix*