Keeping your GPU fed without getting bitten

Tobias Hector
May 2017
Introduction

- You have delicious draw calls
  - Yummy!
Introduction

- You have delicious draw calls
  - Yummy!

- Your GPU wants to eat them
  - It’s really hungry
Introduction

• You have delicious draw calls
  - Yummy!

• Your GPU wants to eat them
  - It’s really hungry

• Keep it fed at all times
  - So it keeps making pixels
Introduction

- You have delicious draw calls
  - Yummy!

- Your GPU wants to eat them
  - It’s really hungry

- Keep it fed at all times
  - So it keeps making pixels

- Don’t want it biting your hand
  - Look at those teeth!
Keeping it fed

• GPU needs a constant supply of food
  - It doesn’t want to wait

• Certain foods are tough to digest
  - Provide multiple operations to hide stalls

• Draw calls provide a variety of nutrition
  - Vertex work, raster work, tessellation, vitamins A-K, etc.
Keeping it fed

<table>
<thead>
<tr>
<th></th>
<th>System</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>0 1</td>
</tr>
<tr>
<td>GPU</td>
<td>0 1</td>
</tr>
</tbody>
</table>
## Keeping it fed

<table>
<thead>
<tr>
<th></th>
<th>System</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td>GPU</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
</tr>
</tbody>
</table>
### Keeping it fed

<table>
<thead>
<tr>
<th></th>
<th>Vertex</th>
<th>Fragment</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPU</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Keeping it fed

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertex</td>
<td>GPU</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Fragment</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>
Not getting bitten

• GPU eating from lots of different plates
  - Don’t touch anything it’s using!

• It doesn’t want a mouthful of beef choc chip ice cream
  - Don’t change data whilst it’s accessing a resource

• Hey I’m eating that!
  - Don’t delete resources whilst the GPU is still using them
On to the serious bits...
Terminology

• Operation
  - An executable task

• Execution Dependency
  - Guarantee for one set of operations to wait on another set of operations

• Memory Space
  - RAM, caches or registers

• Write Propogation
  - Operation that copies a written value between memory spaces

• Memory Dependency
  - Execution dependency including write propagations
Vulkan Memory Spaces

Device Memory

Host Transfer

CPU

Pipeline Stages

Vertex Shader

Fragment Shader
Vulkan Memory Spaces

“VRAM”

Device Memory

“Caches”

Host Transfer

Pipeline Stages

Processor Registers

CPU

Vertex Shader

Fragment Shader
Vulkan Memory Spaces - Data Hazards

“VRAM”

Device Memory

“Caches”

Host “Visible”

Pipeline Stages

Processor Registers

Write 0

Vertex Shader

Fragment Shader
Vulkan Memory Spaces - Data Hazards

"VRAM"

Device Memory

"Caches"

Write 0

Pipeline Stages

Processor Registers

Write 0

Vertex Shader

Fragment Shader
Vulkan Memory Spaces - Data Hazards

- “VRAM”
- “Caches”
- Processor Registers

Device Memory
Pipeline Stages
Vertex Shader
Write 0
Write 0
Write 1
Vulkan Memory Spaces - Data Hazards

“VRAM”

“Caches”

Write 0

Device Memory

Write 1

Vertex Shader

Write 1

Processor Registers

Write 0

Write 0
Vulkan Memory Spaces - Data Hazards

"VRAM"

"Caches"
Write 0
Write 1

Processor Registers
Write 0
Vertex Shader
Write 1
Vulkan Memory Spaces - Data Hazards

“VRAM”

“Caches”

Processor Registers

????

????

????

????

????

????

????

????
Oops

• Let’s try that again...

• This time, with a memory dependency!
Vulkan Memory Spaces - Data Hazards

“VRAM”

“Caches”

Processor Registers

Device Memory

Pipeline Stages

Vertex Shader

Fragment Shader

Write 0

Write 0
Vulkan Memory Spaces - Data Hazards

"VRAM"

Write 0

"Caches"

Write 0
Write 0

Processor Registers

Write 0
Vertex Shader
Write 0
Vulkan Memory Spaces - Data Hazards

“VRAM”
- Write 0

“Caches”
- Write 0
- Write 0

Processor Registers
- Write 0
- Vertex Shader
- Write 1
Vulkan Memory Spaces - Data Hazards

"VRAM"
- Write 0

"Caches"
- Write 0
- Write 1

Processor Registers
- Write 0
- Vertex Shader
- Write 1
Vulkan Memory Spaces - Data Hazards

“VRAM”  Write 1

“Caches”
- Write 1
- Write 1

Processor Registers
- Write 1
- Vertex Shader
- Write 1
How do I do that?

- Synchronization primitives!
Synchronization Types

• 3 types of explicit synchronization in Vulkan

• Pipeline Barriers, Events and Subpass Dependencies
  - Within a queue
  - Explicit memory dependencies

• Semaphores
  - Between Queues

• Fences
  - Whole queue operations to CPU

OpenGL has just two, very coarse synchronization primitives: memory barriers and fences. They are loosely similar to the equivalently named concepts in Vulkan.
Pipeline Barriers

- Precise set of pipeline stages
- Memory Barriers to execute
- Single point in time

Executing a pipeline barrier is similar to a glMemoryBarrier call, though with much more control.
Events

- Events
  - Same info as Pipeline Barriers
  - ...but operate over a range

```c
void vkCmdSetEvent(
    VkCommandBuffer commandBuffer,
    VkEvent event,
    VkPipelineStageFlags stageMask);

void vkCmdResetEvent(
    VkCommandBuffer commandBuffer,
    VkEvent event,
    VkPipelineStageFlags stageMask);

void vkCmdWaitEvents(
    VkCommandBuffer commandBuffer,
    uint32_t eventCount,
    const VkEvent* pEvents,
    VkPipelineStageFlags srcStageMask,
    VkPipelineStageFlags dstStageMask,
    uint32_t memoryBarrierCount,
    const VkMemoryBarrier* pMemoryBarriers,
    uint32_t bufferMemoryBarrierCount,
    const VkBufferMemoryBarrier* pBufferMemoryBarriers,
    uint32_t imageMemoryBarrierCount,
    const VkImageMemoryBarrier* pImageMemoryBarriers);
```
Events

• Events
  - Same info as Pipeline Barriers
  - ...but operate over a range

• CPU interaction
  - No explicit CPU wait

```
VkResult vkSetEvent(
    VkDevice device,
    VkEvent event);

VkResult vkResetEvent(
    VkDevice device,
    VkEvent event);

VkResult vkGetEventStatus(
    VkDevice device,
    VkEvent event);
```
Events

- Events
  - Same info as Pipeline Barriers
  - ...but operate over a range

- CPU interaction
  - No explicit CPU wait

- Warning!
  - May timeout
  - Set events soon after submission
  - Could you just defer submission?
Pipeline Barriers vs Events

• Use pipeline barriers for point synchronization
  - Dependant operation immediately precedes operation that depends on it
  - May be more optimal than set/wait event pair

• Use events if other work possible between two operations
  - Set immediately after the dependant operation
  - Wait immediately before the operation that depends on it
  - Allows more overlap of work

• Use events for CPU/GPU synchronization
  - Memory accesses between processors
  - Late latching of data to reduce latency
Memory Barriers

• Defines write propagations
  - Between “visible” and device memory

• Three types...

OpenGL’s memory barriers imply execution dependencies, which Vulkan memory barriers do not - execution dependencies are provided by a pipeline barrier, event or subpass dependency.
Global Memory Barriers

- **Global Memory Barriers**
  - All memory used by access types
  - Flushes/invalidates whole caches

- **Use when many resources transition**
  - Cheaper than one-by-one
  - Don’t transition unnecessarily!

- **User defines prior access**
  - Driver not tracking for you

```c
typedef struct VkMemoryBarrier {
    VkStructureType sType;
    const void* pNext;
    VkAccessFlags srcAccessMask;
    VkAccessFlags dstAccessMask;
} VkMemoryBarrier;
```
Buffer Barriers

- **Buffer Barriers**
  - A single buffer range
  - Defines access types
  - Defines queue ownership

- **Buffer Range**
  - Offset and size within a buffer

- **Queue Ownership**
  - Defines which queue families are *allowed* to access a write

```c
typedef struct VkBufferMemoryBarrier {
    VkStructureType sType;
    const void* pNext;
    VkAccessFlags srcAccessMask;
    VkAccessFlags dstAccessMask;
    uint32_t srcQueueFamilyIndex;
    uint32_t dstQueueFamilyIndex;
    VkBuffer buffer;
    VkDeviceSize offset;
    VkDeviceSize size;
} VkBufferMemoryBarrier;
```
Image Barriers

- Image Barriers
  - A single image subresource range
  - Defines access types
  - Defines queue ownership
  - Defines image layout

- Image subresource range
  - Specific levels/layers of an image

- Image layouts
  - Additional access information for images
  - Enables GPU image compression
  - Use GENERAL rather than frequent switching

typedef struct VkImageMemoryBarrier {
    VkStructureType sType;
    const void* pNext;
    VkAccessFlags srcAccessMask;
    VkAccessFlags dstAccessMask;
    VkImageLayout oldLayout;
    VkImageLayout newLayout;
    uint32_t srcQueueFamilyIndex;
    uint32_t dstQueueFamilyIndex;
    VkImage image;
    VkImageSubresourceRange subresourceRange;
} VkImageMemoryBarrier;
Example - Texture Upload

// Read image from file, flush to ‘host visible’ memory space
fread(mappedBufferMemory, 1, imageDataSize, imageFile);
vkFlushMappedMemoryRanges(..., {mappedBufferMemory, ...});
// Transition the buffer from host write to transfer read
bBarrier.srcAccessMask = VK_ACCESS_HOST_WRITE_BIT; // Buffer being written to
bBarrier.dstAccessMask = VK_ACCESS_TRANSFER_READ_BIT; // Buffer will be read
// Transition the image to transfer destination
iBarrier.srcAccessMask = 0; // No prior access
iBarrier.dstAccessMask = VK_ACCESS_TRANSFER_WRITE_BIT; // Get image prepared to be transferred to
iBarrier.oldLayout = VK_IMAGE_LAYOUT_UNDEFINED; // No prior access
iBarrier.newLayout = VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL; // Get image prepared to be transferred to
// Pipeline barrier for pre-transfer memory dependency - buffer write was on the host
vkCmdPipelineBarrier(commandBuffer, VK_PIPELINE_STAGE_HOST_BIT, VK_PIPELINE_STAGE_TRANSFER_BIT, &bBarrier, &iBarrier);
// Copy from buffer to image
vkCmdCopyBufferToImage(commandBuffer, srcBuffer, image, VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL, 1, &copy);
// Transition the image from transfer destination to shader read
iBarrier.srcAccessMask = VK_ACCESS_TRANSFER_WRITE_BIT; // Image was just written to
iBarrier.dstAccessMask = VK_ACCESS_SHADER_READ_BIT; // Get image prepared to be read by a shader
iBarrier.oldLayout = VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL; // Image was just written to
iBarrier.newLayout = VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL; // Get image prepared to be read by a shader
// Pipeline barrier for post-transfer memory dependency - fragment shader will read
vkCmdPipelineBarrier(commandBuffer, VK_PIPELINE_STAGE_TRANSFER_BIT, VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT, &iBarrier);
Usually though...

- Many pipeline barrier operations doable via a render pass
  - At least for drawing operations

- Render passes are a dependency graph
  - Allows driver to plan ahead of time how to execute
  - Typically much more efficient than manual barriers
Subpass Dependencies

• **Subpass dependencies**
  - Similar info to Pipeline Barriers
  - Explicitly between two subpasses

• **Memory barriers**
  - Implicit for attachments
  - Explicit for other resources

• **Framebuffer-local dependencies**
  - Same fragment/sample location
  - Cheap for most implementations
  - Use region dependency flag:
    - `VK_DEPENDENCY_BY_REGION_BIT`

```c
typedef struct VkSubpassDependency {
    uint32_t srcSubpass;
    uint32_t dstSubpass;
    VkPipelineStageFlags srcStageMask;
    VkPipelineStageFlags dstStageMask;
    VkAccessFlags srcAccessMask;
    VkAccessFlags dstAccessMask;
    VkDependencyFlags dependencyFlags;
} VkSubpassDependency;
```
Subpass Self-Dependencies

• Subpass self-dependencies
  - Subpasses can wait on themselves
  - A pipeline barrier in the subpass

• Forward progress only
  - Can’t wait on later stages
  - Must wait on earlier or same stage

• Only framebuffer-local for fragments
  - Must use flag:
    - VK_DEPENDENCY_BY_REGION_BIT

typedef struct VkSubpassDependency {
  uint32_t srcSubpass;
  uint32_t dstSubpass;
  VkPipelineStageFlags srcStageMask;
  VkPipelineStageFlags dstStageMask;
  VkAccessFlags srcAccessMask;
  VkAccessFlags dstAccessMask;
  VkDependencyFlags dependencyFlags;
} VkSubpassDependency;

void vkCmdPipelineBarrier(
  VkCommandBuffer commandBuffer,     commandBuffer,
  VkPipelineStageFlags srcStageMask,  srcStageMask,
  VkPipelineStageFlags dstStageMask,  dstStageMask,
  VkDependencyFlags dependencyFlags,  dependencyFlags,
  uint32_t memoryBarrierCount,        memoryBarrierCount,
  const VkMemoryBarrier* pMemoryBarriers,    pMemoryBarriers,
  uint32_t bufferMemoryBarrierCount,    bufferMemoryBarrierCount,
  const VkBufferMemoryBarrier* pBufferMemoryBarriers, pBufferMemoryBarriers,
  uint32_t imageMemoryBarrierCount,    imageMemoryBarrierCount,
  const VkImageMemoryBarrier* pImageMemoryBarriers);
Subpass External Dependencies

- **Subpass external dependencies**
  - Wait on ‘external’ operations
  - `vkCmdWaitEvent` in the subpass
  - Events set outside the render pass

- **Very useful for common dependencies**
  - Use to move between PRESENT_SRC and COLOR_ATTACHMENT_OUTPUT
  - Avoids need for pipeline barriers

```c
typedef struct VkSubpassDependency {
    uint32_t srcSubpass;
    uint32_t dstSubpass;
    VkPipelineStageFlags srcStageMask;
    VkPipelineStageFlags dstStageMask;
    VkAccessFlags srcAccessMask;
    VkAccessFlags dstAccessMask;
    VkDependencyFlags dependencyFlags;
} VkSubpassDependency;

void vkCmdWaitEvents(
    VkCommandBuffer commandBuffer,
    uint32_t eventCount,
    const VkEvent* pEvents,
    VkPipelineStageFlags srcStageMask,
    VkPipelineStageFlags dstStageMask,
    uint32_t memoryBarrierCount,
    const VkMemoryBarrier* pMemoryBarriers,
    uint32_t bufferMemoryBarrierCount,
    const VkBufferMemoryBarrier* pBufferMemoryBarriers,
    uint32_t imageMemoryBarrierCount,
    const VkImageMemoryBarrier* pImageMemoryBarriers);
```
Example - Acquire, Render, Present

// Subpass dependency to express that an attachment has just been acquired
acquiredDependency.srcSubpass = VK_SUBPASS_EXTERNAL;
acquiredDependency.dstSubpass = 0; // First subpass it’s used in
acquiredDependency.srcStageMask = VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT; // Semaphore/Submit guarantees this is sufficient
acquiredDependency.dstStageMask = VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT; // Latest possible stage
acquiredDependency.srcAccessMask = 0; // Previous semaphore will guarantee it is in device memory
acquiredDependency.dstAccessMask = VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT; // Access type

// Subpass dependency to express that an attachment will be presented
presentDependency.srcSubpass = 4; // Last subpass it’s used in
presentDependency.dstSubpass = VK_SUBPASS_EXTERNAL;
presentDependency.srcStageMask = VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT; // Earliest possible stage
presentDependency.dstStageMask = VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT; // Semaphore/Submit guarantees this is sufficient
presentDependency.srcAccessMask = VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT; // Access type
presentDependency.dstAccessMask = 0; // Previous semaphore will guarantee it is in device memory

// Acquire image, render to it, then present it (render loop!)
vkAcquireImageKHR(...);
vkQueueSubmit(...); // Includes execution of render pass
vkQueuePresentKHR(...);
Semaphores

- **Semaphores**
  - Used to synchronize queues
  - Not necessary for single-queue

- **Fairly coarse**
  - Per submission batch
    - E.g. a set of command buffers
  - Multiple per submit command

- **Some implicit memory dependencies**
  - Writes propagated between all device “visible” memory spaces
    - Not guaranteed visible to host

```c
typedef struct VkSubmitInfo {
    VkStructureType sType;
    const void* pNext;
    uint32_t waitSemaphoreCount;
    const VkSemaphore* pWaitSemaphores;
    const VkPipelineStageFlags* pWaitDstStageMask;
    uint32_t commandBufferCount;
    const VkCommandBuffer* pCommandBuffers;
    uint32_t signalSemaphoreCount;
    const VkSemaphore* pSignalSemaphores;
} VkSubmitInfo;
```
Example - Acquire, Render, Present 2

// Acquire an image. Pass in a semaphore to be signalled
vkAcquireNextImageKHR(device, swapchain, UINT64_MAX, acquireSemaphore, VK_NULL_HANDLE, &imageIndex);

// Submit command buffers
submitInfo.waitSemaphoreCount = 1;
submitInfo.pWaitSemaphores = &acquireSemaphore;
submitInfo.commandBufferCount = 1;
submitInfo.pCommandBuffers = &commandBuffer;
submitInfo.signalSemaphoreCount = 1;
submitInfo.pSignalSemaphores = &graphicsSemaphore;

vkQueueSubmit(graphicsQueue, 1, &submitInfo, fence);

// Present images to the display
presentInfo.waitSemaphoreCount = 1;
presentInfo.pWaitSemaphores = &graphicsSemaphore;
presentInfo.swapchainCount = 1;
presentInfo.pSwapchains = &swapchain;
presentInfo.pImageIndices = &imageIndex;

vkQueuePresentKHR(presentQueue, &presentInfo);
Fences

• Fences
  - Used to synchronize queue to CPU

• Very coarse grain
  - Per queue submit command

• Implicit memory dependency
  - Writes propagated between all device “visible” memory spaces
    - Not guaranteed visible to host

GL’s fences are like a combination of a semaphore and a fence in Vulkan - they can synchronize GPU and CPU in multiple ways at a coarse granularity.

VkResult vkQueueSubmit(
    VkQueue queue,
    uint32_t submitCount,
    const VkSubmitInfo* pSubmits,
    VkFence fence);

VkResult vkResetFences(
    VkDevice device,
    uint32_t fenceCount,
    const VkFence* pFences);

VkResult vkGetFenceStatus(
    VkDevice device,
    VkFence fence);

VkResult vkWaitForFences(
    VkDevice device,
    uint32_t fenceCount,
    const VkFence* pFences,
    VkBool32 waitAll,
    uint64_t timeout);
Other important synchronization...

- Implicit... ish?
Queue Submit

- **Queue Submission**
  - Used to push operations to a device

- **Guaranteed forward progress**
  - Operations will execute once pushed

- **Implicit memory dependency**
  - Writes propagated from ‘host visible’ to all ‘device visible’ memory spaces

```c
VkResult vkQueueSubmit(
    VkQueue queue,
    uint32_t submitCount,
    const VkSubmitInfo* pSubmits,
    VkFence fence);
```
Wait Idle

- Ensures execution completes
  - VERY heavy-weight

- vkQueueWaitIdle
  - Wait for queue operations to finish
  - Equivalent to waiting on a fence

- vkDeviceWaitIdle
  - Waits for device operations to finish
  - Includes vkQueueWaitIdle for queues
Wait Idle

- Ensures execution completes
  - VERY heavy-weight

vkQueueWaitIdle
- Wait for queue operations to finish
- Equivalent to waiting on a fence

vkDeviceWaitIdle
- Waits for device operations to finish
- Includes vkQueueWaitIdle for queues

- Warning!
  - Only use for tear-down
  - Will guarantee no overlap
Programmer Guidelines

• Specify EXACTLY the right amount of synchronization
  - Too much and you risk starving your GPU
  - Miss any and your GPU will bite you

• Use the validation layers to help!
  - Won’t catch everything, improving over time

• Pay particular attention to the pipeline stages
  - Fiddly but become intuitive as you use them

• Consider Image Layouts
  - If your GPU can save bandwidth it will

• Prefer render passes
  - Driver able to plan workloads efficiently

• Pay attention to implicit dependencies
  - Submit and Semaphores guarantee a lot - don’t add more!

• Different behaviour depending on implementation
  - Test/Tune on every platform you can find!
Keep your GPU fed without getting bitten!

Questions?
BACKUP SLIDES
Example - Compute to Draw Indirect

// Add a subpass dependency to express the wait on an external event
externalDependency.srcSubpass = VK_SUBPASS_EXTERNAL;
externalDependency.srcStageMask = VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT;
externalDependency.dstStageMask = VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT;
externalDependency.srcAccessMask = VK_ACCESS_SHADER_WRITE_BIT;
externalDependency.dstAccessMask = VK_ACCESS_INDIRECT_COMMAND_READ_BIT;

// Dispatch a compute shader that generates indirect command structures
vkCmdDispatch(...);

// Set an event that can be later waited on (same source stage).
vkCmdSetEvent(commandBuffer, event, VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT);

vkCmdBeginRenderPass(...);

// Transition the buffer from shader write to indirect command - details match external dependency!
bufferBarrier.srcAccessMask = VK_ACCESS_SHADER_WRITE_BIT;
bufferBarrier.dstAccessMask = VK_ACCESS_INDIRECT_COMMAND_READ_BIT;
bufferBarrier.buffer = indirectBuffer;
vkCmdWaitEvent(commandBuffer, event, VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT, VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT, &bufferBarrier);

vkCmdDrawIndirect(commandBuffer, indirectBuffer, ...);
Example - Multi-buffering

// Have enough resources and fences to have one per in-flight-frame, usually the swapchain image count
VkBuffer buffers[swapchainImageCount];
VkFence fence[swapchainImageCount];

// Can use the index from the presentation engine - 1:1 mapping between swapchain images and resources
vkAcquireNextImageKHR(device, swapchain, UINT64_MAX, semaphore, VK_NULL_HANDLE, &nextIndex);

// Make absolutely sure that the work has completed
vkWaitForFences(device, 1, &fence[nextIndex], true, UINT64_MAX);

// Reset the fences we waited on, so they can be re-used
vkResetFences(device, 1, &fence[nextIndex]);

// Change the data in your per-frame resources (with appropriate events/barriers!)
...

// Submit any work to the queue, with those fences being re-used for the next time around
vkQueueSubmit(graphicsQueue, 1, &sSubmitInfo, fence[nextIndex]);