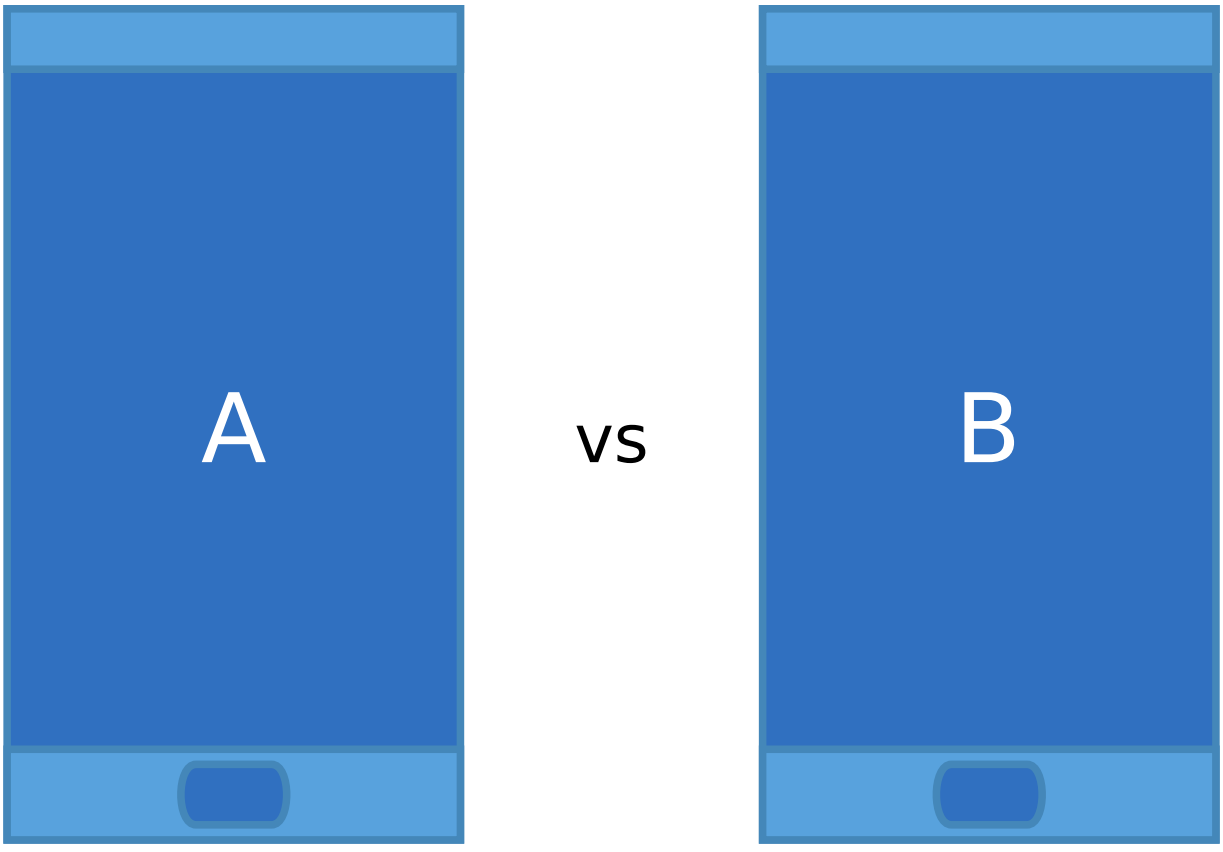
Vulkan Usage Recommendations

Introduction

This Vulkan usage guide assumes the reader is already familiar with the API, but wants to know how to use it effectively across a wide range of Galaxy devices. It also assumes the reader is familiar with the fundamentals of Tile Based Rendering (TBR) GPU architectures commonly found in mobile devices. If you are new to Vulkan, you can get started with our recommended SDKs that introduce the API concepts though code examples. You can also learn more about GPU architectures here.

Before reading this document, we recommend familiarizing yourself with our Game Asset Optimization Recommendations.

Understand your target



When developing high-performance applications, it's essential to understand the capabilities and performance characteristics of the APIs and hardware you are targeting.

For Vulkan, this includes:

- Maximum graphics API version
- Capabilities
  - e.g. Maximum texture resolution

Extensions

Like any performance focused API, there are situations in Vulkan where undefined behavior can occur if the API user doesn't account for it. The Khronos Vulkan Validation Layers and graphics API debugging tools are useful for identifying API misuse. It is also very important to test your application across a wide range of devices, chipsets and GPU architectures to identify bugs early in your development cycle.

Consider GPU architecture differences.
Check capabilities and extensions at run-time. Implement fall-backs.
Test your application on lots of devices.

Asset optimization

Please see our Game Asset Optimization Recommendations.

Shader Precision

SPIR-V supports precision qualifiers (OpDecorate RelaxedPrecision). Precision hints enable developers to tell compilers where reduced precision can be used to improve the performance of ALU operations and, in turn, reduce the power consumption of the GPU.

It's valid for compilers to promote the requested precision of a variable, for example to use 32-bit floating point precision when RelaxedPrecision is specified by the developer. Compilers tend to do this when the instructions introduced for precision conversion introduce more overhead than running the calculations at full precision.

Beware of compilers promoting precision. A shader that runs perfectly on device A (promoted precision) may have artefacts on device B (honors precision qualifier)
Beware of rendering errors that are hidden on some devices by reduced precision

> Recommendations
>
> Use reduced precision to improve performance and reduce power consumption.
> Beware of compilers promoting precision. Test on lots of devices to catch shader precision artefacts early.
> Beware of rendering errors that are hidden on some devices by reduced precision.

Pipeline management

Creating pipelines at draw time can introduce performance stutters. We recommend creating pipelines as early as possible in application execution. If you are unable to re-architect your rendering engine to create pipelines before draw time, we recommend creating pipelines once and adding them to a map so they can be looked up from hashed state by subsequent draws.

Pipeline caches enable the driver to reuse state from cached pipelines when new pipelines are created. This can significantly improve performance by reusing baked state instead of repeating costly operations, such as shader compilation. We recommend using a single pipeline cache to ensure the driver can reuse state from all previously created pipelines. We also recommend writing the pipeline cache to a file so it can be reused by future application runs.

Pipeline derivatives let applications express "child" pipelines as incremental state changes from a similar "parent"; on some architectures, this can reduce the cost of switching between similar states. Many mobile GPUs gain performance primarily through pipeline caches, so pipeline derivatives often provide no benefit to portable mobile applications.

> Recommendations
>
> Create pipelines early in application execution. Avoid pipeline creation at draw time.
> Use a single pipeline cache for all pipeline creation.
> Write the pipeline cache to a file between application runs.
> Avoid pipeline derivatives.

Descriptor set management

Descriptor sets define resource bindings for a draw. Ideally, descriptor sets should be generated at build time and cached for run-time execution. When this is not possible, descriptor sets should be created as early as possible in application execution (application load or level load).

Like most resources in Vulkan, the API user is responsible for synchronizing descriptor set updates to ensure changes aren't made on the host while there are pending device reads. We recommend using a pool of descriptors per swap index to simplify resource synchronization and facilitate descriptor sharing between draws with the same bindings. If you are unable to re-architect your rendering engine and need to update descriptor sets at draw time, descriptor set and buffer management strategies should be considered very carefully to avoid modification of in-flight descriptors. As discussed in Buffer Management, designing an engine to cope with the spec minimum `VkPhysicalDeviceLimits::maxUniformBufferRange` value is important, as this limit can be easily hit when sharing buffers between descriptors.

If uniform or storage buffers offsets need to be changed at a high frequency (e.g. per-draw), we recommend binding the buffers with as dynamic with `VK_DESCRIPTOR_TYPE_*_BUFFER_DYNAMIC` and setting the offset with `pDynamicOffsets` when `vkCmdBindDescriptorSets` is called. This allows the offsets to be changed before draw execution without modifying the descriptor set.

> Recommendations
>
> Build descriptor sets as early as possible (ideally at asset build time).
> Do not bind resources that are not referenced by draws.
> If buffer offsets need to be changed at a high frequency (e.g. between draws), use `pDynamicOffsets` when the descriptor is bound.
> Avoid high-frequency descriptor set modification. If dynamic updates are required, ensure reads and writes are synchronized.

Buffer management

As discussed in Game Asset Optimizations: Interleaved vertex attributes, position attributes should be stored in a separate buffer to all other attributes. This enables modern mobile GPUs to execute vertex shading more efficiently. Unless updated at different frequencies, all other attributes should be interleaved in a single buffer.

Uniform buffers should be allocated as early as possible in application execution and per-frame allocations should be avoided (reusing previous allocations is much faster). Fences should be used to ensure in-flight renders finish accessing buffer ranges that will be modified by the current frame's API calls.

A uniform buffer should not contain data that is updated at different frequencies. For example, if a draw depends on data that is static and data that is set per-frame (e.g. transformation matrices), two buffers should be used. Uniform data that is common to multiple draws should be stored in a single buffer.

To avoid redundant transfers, uniform buffer memory should be allocated with `VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT` set. This flag enables `vkMapMemory` to be used for efficient modifications from the host (one copy fewer compared with using a staging buffer). Frequent Map/Unmap calls should be avoided. A buffer can be mapped persistently by setting `VkMemoryPropertyFlagBits::VK_MEMORY_PROPERTY_HOST_COHERENT_BIT`, if available. Beware: persistent mapping will make it much more difficult for API capture tools, to track buffer modifications. For this reason, we would recommend implementing a non-persistently mapped fallback path for platforms with dedicated GPU memory and to simplify debugging.

Buffers should be aligned to `VkPhysicalDeviceLimits::min*Alignment` limits.The maximum size of buffer allocations and descriptor set bindings can be queried with:

Limit	Description
`VkPhysicalDeviceLimits::maxUniformBufferRange`	Maximum uniform buffer memory range
`VkPhysicalDeviceLimits::maxDescriptorSetUniformBuffers`	Maximum number of uniform buffers that can be bound to a descriptor set
`VkPhysicalDeviceLimits::maxDescriptorSetUniformBuffersDynamic`	Maximum number of dynamic uniform buffers that can be bound to a descriptor set
`VkPhysicalDeviceLimits::maxPerStageDescriptorUniformBuffers`	Maximum number of uniform buffers that can be accessed by a single shader stage

As discussed in Descriptor set management, dynamic buffers and dynamic offsets should be used to minimize descriptor set updates.

For shader inputs, uniform buffers should always be preferred over storage buffers.

> Recommendations
>
> Store attribute data in two buffers - one for vertex positions, one for all other attributes.

Align offsets for uniform buffers to `VkPhysicalDeviceLimits::minUniformBufferOffsetAlignment`.
On devices with unified memory, use `VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT` and prefer persistent buffer mappings to flushes and frequent map/unmap calls.
For shader inputs, uniform buffers should always be preferred over storage buffers.

Shader inputs

Vulkan provides a variety of mechanisms to set shader constants including; uniform buffers, specialization constants and push constants.

Specialization constants are static values that can be set at pipeline creation time (when SPIR-V binaries are compiled).

Push constants provide a mechanism to set shader constant input data via Vulkan commands rather than buffer objects. Push constant storage space is limited. The maximum number of bytes available for push constant storage can be set can be queried with `VkPhysicalDeviceLimits::maxPushConstantsSize`. When available specialized offset mechanisms, such as dynamic uniform buffer offsets, should be preferred over push constants. If a rendering engine cannot guarantee that all draws will consume less than the spec minimum `maxPushConstantsSize` bytes for push constant data (128 bytes), a uniform buffer based fallback should be implemented. This will ensure the rendering engine works on Vulkan implementations that only support the smallest number of push constant bytes. In the dynamic uniform buffer offset path, offsets can be specified per-draw by calling `vkCmdBindDescriptorSets`.

The list below outlines our recommendations for setting constant values in shaders:

Static branching: If you can define execution paths at SPIR-V generation time, generate dedicated SPIR-V for the execution path (for example, by #define in GLSL). Otherwise, use Specialization Constants at SPIR-V pipeline creation time
Pipeline creation time constants: If you can set a constant value at pipeline creation time, use Specialization Constants
Frequent uniform updates: If uniform data updates apply to multiple draws following a `vkCmdBindDescriptorSets` call, use use a dynamic uniform buffer and dynamic offsets when `vkCmdBindDescriptorSets` is called to avoid necessary descriptor set modification (see Descriptor set management for more information). If uniform data is updated at a higher frequency, for example per-draw, consider push constants instead of dynamic uniform buffer offsets

Recommendations

Create dedicated SPIR-V for generation-time static branching (e.g. #define in shader source).
Use specialization constants for SPIR-V compilation-time static branching.
Use specialization constants to define SPIR-V compilation-time constant values.
If uniform data updates apply to multiple draws following a *vkCmdBindDescriptorSets* call, use a dynamically offset uniform buffer.
Consider push constants for uniform data updated at a per-draw frequency.
Only rely on push constants for high frequency data if your engine uses <=1 28 bytes. If this can't be guaranteed, implement a dynamic uniform buffer fallback path.

Beware: No implicit uniform type conversions

Unlike OpenGL ES, Vulkan will not perform implicit uniform type conversions. Developers are responsible for ensuring the contents of a buffer binding match the shader uniforms they are bound to.

> Recommendations
>
> Ensure uniform buffer data types match shader uniform variables.

View frustum culling

The cheapest draw the driver and GPU will ever process is the draw that is never submitted. To avoid redundant driver and GPU processing, a common rendering engine optimization is to submit a draw to the graphics API only if it falls within, or intersects, the bounds of the view frustum. View frustum culling is usually cheap to execute on the CPU and should always be considered when rendering complex 3D scenes.

> Recommendations
>
> Always try to use view frustum culling to avoid redundant driver and GPU processing.

Command buffer building

Vulkan's is designed to allow command buffers to be built across multiple threads, enabling this costly task to be done across multiple CPU cores. Additionally, secondary command buffers can be created, making it easier to break the work down into smaller chunks. Secondary command buffers must be committed to a primary command buffer after they have been built. However, in some implementations, the GPU requires all commands in a render pass to belong to a single contiguous block of memory - in which case, the Vulkan drivers for these GPUs need to memcpy() the secondary command buffers to a primary command buffer before the commands are executed. Because of this overhead, we recommend preferring primary command buffers to secondary command buffers. If you would like to multi-thread your render across multiple CPU cores, we would recommend prioritizing building your primary command buffers in parallel before considering secondary command buffers.

If you decide to use secondary command buffers, you should carefully consider your partitioning scheme. When the scene is built in chunks, it will be harder to for your engine to optimize draw call submission order and minimize state changes. If a secondary command buffer building path is implemented you should decide at run-time if the path is required, or if primary command buffer builds would be faster.

> Recommendations
>
> Prefer submitting commands to primary command buffers than using secondary command buffers.
> Avoid secondary command buffers on GPU-limited devices.
> Consider building primary command buffers in parallel before considering secondary command buffers.
> If using secondary command buffers, consider your partitioning scheme carefully.

Instanced draws

All Vulkan CmdDraw* functions accept an `instanceCount` parameter. Per-instance data can be provided by binding a buffer with `VkVertexInputBindingDescription::inputRate` set to VK_VERTEX_INPUT_RATE_INSTANCE.

> Recommendations
>
> Always use a single draw and per-instance data to render instanced objects.

Clearing framebuffer attachments

In Vulkan, there three mechanisms to clear framebuffer attachments:

Renderpass load operation (`VK_ATTACHMENT_LOAD_OP_CLEAR`)
`vkCmdClearAttachments`
`vkCmdClearColorImage` and `vkCmdClearDepthStencilImage`

To ensure the operations are performed efficiently, it's important to ensure the correct mechanism is being used for a given scenario.

> Recommendations
>
> When clearing attachments at the start of a render pass, use `VK_ATTACHMENT_LOAD_OP_CLEAR.`
> When clearing attachments within a sub-pass, use `vkCmdClearAttachments.`
> `vkCmdClearColorImage` and `vkCmdClearDepthStencilImage` can be used to clear outside of a render pass. These functions are the least efficient mechanism on tile-based GPU architectures.

Efficient render pass upscaling

A common bottleneck in high-fidelity 3D games is fragment shading execution time. To reduce the per-frame fragment shading cost, the game scene can be rendered at a reduced resolution then up scaled before rendering the user interface at the device's native resolution.

Render pass A (reduced resolution)

Render game scene

Render pass B (native resolution)

Upscale game scene image
Render the UI

An upscale can be performed in two ways:

1. `vkCmdBlitImage`

a. Copy regions of a source image into a destination image, potentially performing format conversion, arbitrary scaling and filtering

b. Depending on the implementation, this operation may be performed by dedicated blitting hardware, by the GPU or by the CPU

2. Render a full-screen quad

a. Start render pass B with a full-screen draw call that samples the image

Although `vkCmdBlitImage` may seem like the best option, it tends to be less efficient than rendering a full-screen quad on mobile GPUs. The reason for this is that it requires an explicit copy from one `VkImage` to another `VkImage`. On implementations that use the GPU for the blit operation, this may be implemented as an additional render pass between A and B - consuming memory bandwidth and GPU cycles that could have been spent elsewhere. The full-screen quad approach, on the other hand, only requires an image layout transition of one `VkImage`. Depending on the type of transition, an implementation may be able to perform this "for free".

> Recommendations
>
> Only consider game scene upscaling when fragment shader limited. Beware of the upscale's bandwidth cost.
> Prefer full-screen quad upscaling to `vkCmdBlitImage.`

Subpasses

Mobile devices have limited memory bandwidth. Additionally, memory bandwidth data transfers are power intensive to use so it's best to use it as little as possible.

In 3D graphics rendering, a framebuffer may need more than one attachment. In many cases, only some attachment data needs to be preserved - all other attachment data is temporary. For example, a color buffer may be required for the rendered image and a depth buffer may be needed to ensure primitives are rendered in the intended order. In this scenario, the depth data doesn't need to be preserved so writing it from GPU memory to system memory wastes bandwidth. Additionally, the color and depth buffer contents from frame N-1 may not be required for frame N. As uploading this data would redundantly use memory bandwidth, we want to tell the driver those operations aren't required.

Attachment load op

Each attachment's VkAttachmentLoadOp property defines how the attachment should be initialized at the start of a subpass.

`VK_ATTACHMENT_LOAD_OP_DONT_CARE`

This is the most efficient option. It should be used when the attachment doesn't need to be initialized, e.g. in renders where every pixel will be colored by a sky box or draw

`VK_ATTACHMENT_LOAD_OP_CLEAR`

This operation is very efficient on tilers. As discussed in the Clearing framebuffer attachments section, it is the most efficient way to clear an attachment at the start of a render pass

`VK_ATTACHMENT_LOAD_OP_LOAD`

This is the most costly operation. On TBRs, on-tile memory is initialized by loading the attachment's preserved data from system memory

Attachment store op

Each attachment's VkAttachmentStoreOp property defines how the attachment should be stored at the end of a subpass.

`VK_ATTACHMENT_STORE_OP_DONT_CARE`

This is the most efficient option. It should be used when the attachment doesn't need to be preserved, e.g. depth and stencil attachment data only used to render this pass

`VK_ATTACHMENT_STORE_OP_STORE`

This is the most costly operation. On TBRs, on-tile memory is preserved by storing to an attachment image in system memory

If an attachment image is never loaded or stored, the `VkImage` should be created with `VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT` and bound to memory with the `VK_MEMORY_PROPERTY_LAZILY_ALLOCATED_BIT` property. This enables the driver to treat the image as transient, such that backing memory may be allocated lazily.

Advanced subpass usage

Advanced subpass topics, such as multi-pass rendering for deferred lighting systems, is discussed in our Introduction to Vulkan Render Passes article.

> Recommendations
>
> Use `VK_ATTACHMENT_LOAD_OP_DONT_CARE` by default, `VK_ATTACHMENT_LOAD_OP_CLEAR` if the attachment needs to be cleared and `VK_ATTACHMENT_LOAD_OP_LOAD` if previous attachment data needs to be preserved.
> Use `VK_ATTACHMENT_STORE_OP_DONT_CARE` for all attachments that do not need to be preserved.
> If an attachment image is never loaded or stored, allocate it with `VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT` and back it with memory allocated with the `VK_MEMORY_PROPERTY_LAZILY_ALLOCATED_BIT` property.

Synchronization

Synchronization in Vulkan is complex and a common source of bugs in the games we have supported. The following chapters outline our synchronization recommendations.

Terminology

Before attempting to understand synchronization primitives, it's important to understand the Vulkan terminology used throughout these spec chapters:

Synchronization scope: A collection of operations. In most synchronization commands, synchronization scopes are defined by source and destination pipeline stage masks

Execution dependency: Most synchronization commands define an execution dependency. All operations defined in the first synchronization scope must execute before all operations defined in the second synchronization scope

Availability operation: Specifies a memory operation that must complete before a subsequent memory access. For example, an image memory barrier may specify that a given image `VkImage`'s color attachment writes must complete before a subsequent access

Visibility operation: Specifies a memory operation that must occur once a given availability operation has occurred. For example, an image memory barrier may specify that a shader will read an attachment once the specified availability operations have completed

Resource transition: Some `VkImage` resources may need to transition from one layout to another between availability and visibility operations. In the case of image memory barriers, resource transitions are defined by `oldLayout` and `newLayout`

Memory dependency: Defines a set of availability operations that must complete before a set of visibility operations. A memory dependency may also define a resource transition

Semaphores

Semaphores can be used to control resource access across multiple queues. The most common semaphore use case is to synchronize graphics and presentation queues.

Example: graphics queue ⇔ presentation queue synchronization

Code Block 1 Per-frame

Show Lines

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

VkPipelineStageFlags waitDstStageMask = VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT

// Submit command buffers
submitInfo.waitSemaphoreCount = 1;
submitInfo.pWaitSemaphores = &acquireSemaphore;
submitInfo.pWaitDstStageMask = &waitDstStageMask;
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);

```

See this page for more information.

Recommendations

Always use semaphores to synchronize the graphics and presentation queues.

Fences

Fences can be used to communicate from a queue to the host. The most common fence use case is to signal when a graphics render has completed so that resources can be reused for a subsequent frame. For optimal performance, we recommend a 1:1 mapping between the number of presentable images and resources

We recommend to avoid calling `vkWaitForFences` in your frame loop, as this stalls execution and results in decreased performance (1-3 fps drop observed in games we have profiled). Instead, we advise to call `vkGetFenceStatus` to determine which presentable image is available.

Example: graphics queue ⇒ host synchronization

Code Block 2 Initialization

Show Lines

```
/* Create a fence for each swapchain index.
Default all to signalled so that they are considered "available" in our later test
*/
fenceInfo.flags = VK_FENCE_CREATE_SIGNALED_BIT;
VkFence fences[SWAPCHAIN_IMAGE_COUNT];
for(int i=0; i < SWAPCHAIN_IMAGE_COUNT; i++) {
    vkCreateFence(device, fenceInfo, NULL, fences);
}
```

Code Block 3 Per-frame rendering loop content

Show Lines

```
vkAcquireNextImageKHR(device, swapchain, UINT64_MAX, acquireSemaphore, VK_NULL_HANDLE
/* Some per-frame operations don’t need to write to device
resources immediately. If a fence hasn’t been signalled yet,
we can poll for the fence status and process small jobs while
we’re waiting. Job execution time should be small (<1ms) to
ensure the fence status is queried regularly. Note that the
first time through the loop, before any rendering, our fences
are signalled so that we don't block
*/
VkResult fenceStatus = VK_NOT_READY;
while(!smallJobQueue.empty() && fenceStatus != VK_SUCCESS){
     // Pop a job from the queue and execute it
     // …
     fenceStatus = vkGetFenceStatus(device, fences[nextImageIndex]);
}
// If we’ve run out of jobs and the fence hasn’t been signalled, wait
vkWaitForFences(mDevice, 1, &fences[nextImageIndex], VK_TRUE, UINT64_MAX);
// Set the fence state to unsignalled
vkResetFences(mDevice, 1, &fences[nextImageIndex]);

// Submit work to the queue. Set the VkFence that should be triggered on completion
vkQueueSubmit(graphicsQueue, 1, &submitInfo, fences[imageIndex]);

// Present images to the display
// ...
vkQueuePresentKHR(presentQueue, &presentInfo);

```

Recommendations

> Always use fences to synchronize the graphics queue with the host.
> Keep references to dynamic resources in a circular buffer, and use fences to determine when each resource can be reused.
> Avoid calling `vkWaitForFences` in a frame loop, and use `vkGetFenceStatus` instead.

Barriers

Vulkan's barriers enable API users to insert dependencies between commands in the same queue, or between commands in the same subpass. Execution dependencies are defined by pipeline stage synchronization scopes. In addition to execution dependencies, `vkCmdPipelineBarrier` calls can accept three types of memory access barriers - global, buffer and image. Memory barriers enable API users to ensure write operations during (or before) the first synchronization scope complete before read operations in the second synchronization scope. As the name suggests, global memory barriers are used for synchronizing all memory accesses rather than specifying a particular resource. For more fine grained synchronization, buffer memory and image memory barriers can be used.

<table>
<tr><th>Host</th><th>Transfer</th><th>Compute</th><th>Graphics</th></tr>
<tr><td colspan="4">TOP_OF_PIPE_BIT</td></tr>
<tr><td>HOST_BIT</td><td></td><td rowspan="2"></td><td rowspan="2"></td></tr>
<tr><td rowspan="14"></td><td>TRANSFER_BIT</td></tr>
<tr><td rowspan="13"></td><td>COMPUTE_SHADER_BIT</td></tr>
<tr><td>DRAW_INDIRECT_BIT</td><td>DRAW_INDIRECT_BIT</td></tr>
<tr><td rowspan="12"></td><td>VERTEX_INPUT_BIT</td></tr>
<tr><td>VERTEX_SHADER_BIT</td></tr>
<tr><td>TESSELLATION_CONTROL_SHADER_BIT</td></tr>
<tr><td>TESSELLATION_EVALUATION_SHADER_BIT</td></tr>
<tr><td>GEOMETRY_SHADER_BIT</td></tr>
<tr><td>EARLY_FRAGMENT_TESTS_BIT</td></tr>
<tr><td>FRAGMENT_SHADER_BIT</td></tr>
<tr><td>LATE_FRAGMENT_TESTS_BIT</td></tr>
<tr><td>COLOR_ATTACHMENT_OUTPUT_BIT</td></tr>
<tr><td colspan="4">BOTTOM_OF_PIPE_BIT</td></tr>
</table>

The four columns in the table above show Vulkan's pipelines. `TOP_OF_PIPE_BIT and BOTTOM_OF_PIPE_BIT` are common to all pipelines. Respectively, they mark generic pipeline stages for the first command that begins execution and the last that completes execution.

To avoid pipeline bubbles, it's important for API users to consider the execution dependencies of barriers very carefully. This is especially true of barrier calls made within a subpass on tile-based GPU architectures. For example, if a barrier set within a subpass that has `BOTTOM_OF_PIPE_BIT` in it's first synchronization scope and `TOP_OF_PIPE_BIT` in it's second scope, all GPU commands prior to the barrier will be flushed and commands after the barrier will have to wait for the flush to complete before they can begin executing.

To avoid bubbles, the first synchronization scope of a barrier should be set as early in the pipeline as possible and the second synchronization scope should be set to the latest possible pipeline stages. Additionally, the first and second synchronization scopes should be as narrow as possible. Setting `TOP_OF_PIPE_BIT` in `srcStageMask` will never block the barrier, and behaves as if the first synchronization scope is empty. Similarly, `BOTTOM_OF_PIPE_BIT` in `dstStageMask` will mean an empty second synchronization scope. There are cases where this behavior is desirable - when other synchronization (e.g. semaphores) already enforce the required dependencies - but these options should be used carefully.

Global memory barriers should be preferred to buffer memory barriers, unless fine-grained buffer synchronization is required - for example synchronizing writes to a specific range of a buffer. Image and buffer memory barriers that depend on the same synchronization scopes should be batched in a single `vkCmdPipelineBarrier` call.

> Recommendations
>
> Set your source dependency as early in the pipeline as possible and your destination as late as possible.
> Beware of pipeline bubbles that can be introduced by setting barriers within a subpass on tile-based architectures.
> Prefer global memory barriers to buffer memory barriers.
> Batch image and buffer memory barriers into a single `vkCmdPipelineBarrier` call if they have the same scopes.
> Consult the Khronos Group's Synchronization Examples for use case specific recommendations.

Events

Events provide a fine-grained synchronization mechanism for:

- Host to graphics queue synchronization
- Command dependencies within a render pass

`vkCmdWaitEvents` takes very similar arguments to `vkCmdPipelineBarrier`. The additional parameters are `eventCount` and `pEvents`. The synchronization scope defined by `pEvents` and `srcStageMask` must finish executing before the commands after `vkCmdWaitEvents` and `dstStageMask` can execute.

Host to graphics queue event synchronization may be useful when resource writes need to occur after a command dependent on those resources has been written to a command buffer. For example, to reduce latency between user input and GPU execution, a VR compositor may write a matrix representing a head orientation delta since the scene was rendered to a uniform buffer before time warp composition is performed. `vkCmdWaitEvents` blocks execution until this event has been signalled. However, note that a GPU submission which takes too long may be killed by the system (on the assumption that it may simply have crashed), so extreme caution is needed with this approach.

> Recommendations
>
> Prefer barriers to events.

Wait idle

Wait idle is a very heavy-weight form of synchronization. `vkQueueWaitIdle` waits for all queue operations to complete and is functionally equivalent to waiting on a fence. `vkDeviceWaitIdle` waits for all device operations to complete. The wait idle functions guarantee there is no overlap, and should only be used for rendering engine tear down.

> Recommendations
>
> Only use WaitIdle functions for rendering engine tear down.

Swapchains

When creating a swapchain, we recommend that the VK_PRESENT_MODE_FIFO_KHR presentation mode used and `minImageCount` is set to 3.

While using the VK_PRESENT_MODE_MAILBOX_KHR presentation mode can potentially make your frame rate more stable; however, this is done by throwing away entire rendered frames, working the GPU more heavily than needed (and thereby using up more power). We strongly recommend profiling and optimizing using FIFO, and only using MAILBOX when minimal latency is absolutely required.

The main consideration in deciding the number of images in a swapchain is balancing memory use with smoothness. Android supports creation of a swapchain with just 2 images. This reduces the memory required, but introduces bubbles in the rendering pipeline if a frame is not rendered in time for a v-sync. It is possible to request a swapchain with more than 3 images, but the benefits of this versus additional memory consumption should be considered carefully.

Semaphores should be used to synchronize the graphics and presentation queues. Fences should be used to synchronize the graphics queue with the host. For more information, please refer to the following examples:

Example: graphics queue ⇔ presentation queue synchronization
Example: graphics queue ⇒ host synchronization

> Recommendations
>
> Use the VK_PRESENT_MODE_FIFO_KHR presentation mode.
> Set your swapchain's `minImageCount` to 3.
> Use semaphores to synchronize the graphics and presentation queues.
> Use fences to synchronize the graphics queue with the host.

Minimizing overdraw

Fragments are rasterized in the order that primitives are submitted to the graphics queue. If multiple primitives overlap, all the fragments may be rendered, even if the resulting fragments are occluded by others in the final image. Rendering of fragments whose values will be overwritten by later fragments is known as overdraw.

Some architectures have optimizations to reduce the overhead of shading fragments that will later be obscured. Nevertheless, to get the best portable performance, it is recommended to use early depth/stencil testing and to submit opaque draw calls in depth order (typically front to back, depending on depth test mode), which allows the GPU to determine whether a primitive is visible prior to shading. Transparent primitives should be rendered after opaque primitives in order to preserve blending behaviour.

> Recommendations
>
> Sort opaque draws from front to back, then render transparent primitives. Ordering doesn't have to be perfect to give a good overdraw reduction. Find an algorithm that gives a good balance between CPU sorting overhead and overdraw reduction.

Avoid redundant API calls

Although redundant Vulkan calls, such as repeatedly setting state without rendering, are unlikely to cause bottlenecks, they still have a cost.

> Recommendations
>
> Avoid redundant API calls.

Robust Buffer Access

Vulkan drivers assume the API is being used correctly by the calling application. This assumption enables the driver to avoid costly run-time validation checks. Applications can request stronger robustness guarantees by enabling the `robustBufferAccess` feature. The primary purpose of robust buffer access is to provide out of bounds buffer checks. The Vulkan specification guarantees that `robustBufferAccess` will be an available feature on the physical device, but enabling it can incur significant performance penalties on some architectures.

> Recommendations
>
> Use `robustBufferAccess` during development to catch bugs. Disable it in release builds.

Validation Layers

The Khronos Group's Vulkan validation layers should used regularly during development to identify API misuse. For the best results, we recommend using the latest available validation layers. The source code of the validation layers is hosted on GitHub and is straightforward to build. Each layer release is tagged as sdk-*.

You should not rely on a sub-set of available layers. To catch all potential issues, we recommend regularly using all layers provided by Khronos.

As a rendering engine may use different Vulkan features on different devices (e.g. texture formats), we recommend running the validation layers with multiple Galaxy devices. All validation layer messages should be addressed before a game is released.

Error codes are not message specific. They categorize a type of issue and may be reused by more than one message. We do not recommend parsing the layer output, as error codes and message text may vary between validation layer releases.

If the meaning of a message reported by the layer isn't clear, we recommend searching the validation source code to better understand the cause.

If you identify any false positives or a scenario where invalid API usage isn't being caught, please report an issue against the layers on GitHub .

If you would like to know more about Khronos' validation layers, we would recommend reading LunarG's Vulkan Validation Layers Deep Dive slides.

> Recommendations
>
> Use the Khronos validation layers regularly during development.
> Always use the latest layer release.
> Use the layers on multiple devices to catch device-specific messages.
> Beware: Error codes categorize an issue - they are not a unique identifier for a given message.
> If a message is unclear, refer to the layer source code.