# Vulkan Bits and Pieces: Synchronization in a Frame

# Vulkan Bits and Pieces: Synchronization in a Frame

Mon 15 May 2017

One of the things that you must implement when using Vulkan is basic structure of a rendering frame, which consists of filling a command buffer, submitting it to a command queue, acquiring image from swap chain and finally presenting it (order not preserved here). Things happen in parallel, as GPU works asynchronously to the CPU, so you must explicitly synchronize all these things. In this post I'd like to present basic structure of the rendering frame, with all necessary synchronization.

There are actually two objects that we must take care of simultaneously. First is **command buffer**. One time it is being filled on the CPU by using `vkBeginCommandBuffer`, `vkEndCommandBuffer` and everything in between (like starting render passes and posting all the `vkCmd*` commands). Another time (after being submitted to a queue using `vkQueueSubmit`) it waits for execution or it is being executed on the GPU. These things cannot happen at the same time, so we need synchronization. Because it's a GPU-to-CPU synchronization, we use fence for that. It's better to have multiple command buffers and all these synchronization objects, stored in array and used in a round-robin fashion (with help of `index` variable) because this way one can be filled while the other one is consumed at the same time.

- When submitting command buffer to the queue near the end of a frame using `vkQueueSubmit`, we specify a fence that will be signaled when the command buffer finishes executing.
- At the beginning of next frame, we wait for fence of the command buffer we want to use this time (better to take different one) to become signaled, using `vkWaitForFences`.
- Only then we can call `vkBeginCommandBuffer`, all rendering commands and finally `vkEndCommandBuffer`.
- We also need to manually reset the fence of our command buffer using `vkResetFences` anywhere between `vkWaitForFences` and `vkQueueSubmit`.

Second object is an **image** acquired from the swapchain. One time it is being rendered to during command buffer execution. Another time it is presented on the screen. Again, we need to synchronize it so these states happen in a sequence. Because it's a GPU-to-GPU synchronization, we use semaphores for that. There are multiple images as well because swapchain consists of several of them (not necessarily used in round-robin fashion, we need to obtain index of a new image using `vkAcquireNextImageKHR` function, so there is separate `imageIndex`).

- When acquiring fresh image from swapchain (or actually just its index) using `vkAcquireNextImageKHR`, it's better to do it as late as possible in a frame (not at the beginning of any rendering, but right before we are going to do final postprocessing pass and render to that image), because this function may block.
- Despite that, after the function returns, swapchain image at returned index is still not ready. We specify a semaphore as argument to this function that will become signaled when the image is really eligible for use. This is the first semaphore, called `g_ImageAvailableSemaphores` below.
- We specify the same semaphore from `g_ImageAvailableSemaphores` among `pWaitSemaphores` when calling `vkQueueSubmit`, so this function waits for the semaphore to become signaled before submitting command buffer that actually needs this image.
- As another argument to `vkQueueSubmit` (or rather `VkSubmitInfo` structure) called `pSignalSemaphores` we specify second semaphore, which I named `g_RenderFinishedSemaphores`. This one will become signaled after submitted command buffer finishes execution.
- That second semaphore from `g_RenderFinishedSemaphores` is in turn passed as `pWaitSemaphores` member of `VkPresentInfoKHR` structure when calling `vkQueuePresentKHR`, so that the image is presented only when rendering to it finishes.

Having following objects already initialized:

```
VkDevice g_Device;
VkQueue g_GraphicsQueue;
VkSwapchainKHR g_Swapchain;
VkImageView g_SwapchainImageViews[MAX_SWAPCHAIN_IMAGES];

const uint32_t COUNT = 2;
uint32_t g_NextIndex = 0;
VkCommandBuffer g_CmdBuf[COUNT];
VkFence g_CmdBufExecutedFences[COUNT]; // Create with VK_FENCE_CREATE_SIGNALED_BIT.
VkSemaphore g_ImageAvailableSemaphores[COUNT];
VkSemaphore g_RenderFinishedSemaphores[COUNT];
```

Structure of a frame may look like this:

```
uint32_t index = (g_NextIndex++) % COUNT;
VkCommandBuffer cmdBuf = g_CmdBuf[index];
```

```
vkWaitForFences(g_Device, 1, &g_CmdBufExecutedFences[index], VK_TRUE, UINT64_MAX);

VkCommandBufferBeginInfo cmdBufBeginInfo = { VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO };
cmdBufBeginInfo.flags = VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT;
vkBeginCommandBuffer(cmdBuf, &cmdBufBeginInfo);

// Post some rendering commands to cmdBuf.

uint32_t imageIndex;
vkAcquireNextImageKHR(g_Device, g_Swapchain, UINT64_MAX, g_ImageAvailableSemaphores[index], VK_NULL_HANDLE,
 &imageIndex);

// Post those rendering commands that render to final backbuffer:
// g_SwapchainImageViews[imageIndex]

vkEndCommandBuffer(cmdBuf);

vkResetFences(g_Device, 1, &g_CmdBufExecutedFences[index]);

VkPipelineStageFlags submitWaitStage = VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT;
VkSubmitInfo submitInfo = { VK_STRUCTURE_TYPE_SUBMIT_INFO };
submitInfo.waitSemaphoreCount = 1;
submitInfo.pWaitSemaphores = &g_ImageAvailableSemaphores[index];
submitInfo.pWaitDstStageMask = &submitWaitStage;
submitInfo.commandBufferCount = 1;
submitInfo.pCommandBuffers = &cmdBuf;
submitInfo.signalSemaphoreCount = 1;
submitInfo.pSignalSemaphores = &g_RenderFinishedSemaphores[index];
vkQueueSubmit(g_GraphicsQueue, 1, &submitInfo, g_CmdBufExecutedFences[index]);

VkPresentInfoKHR presentInfo = { VK_STRUCTURE_TYPE_PRESENT_INFO_KHR };
presentInfo.waitSemaphoreCount = 1;
presentInfo.pWaitSemaphores = &g_RenderFinishedSemaphores[index];
presentInfo.swapChainCount = 1;
presentInfo.pSwapchains = &g_Swapchain;
presentInfo.pImageIndices = &imageIndex;
vkQueuePresentKHR(g_GraphicsQueue, &presentInfo);
```

At least that’s what I currently believe is the correct and efficient way. If you think I’m wrong, please leave a comment or write me an e-mail.

That’s just one aspect of using Vulkan. There are still more things to do before you can even render your first triangle, like transitioning swapchain image layout between `VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL` and `VK_IMAGE_LAYOUT_PRESENT_SRC_KHR`.

2 Comments | #vulkan #graphics Share

Comments