The Swap Chain

How We Think of OpenGL Framebuffers

Vulkan Thinks of it as a Ring Buffer

Vulkan Thinks of it as a Ring Buffer

What is a Swap Chain?

Vulkan does not use the idea of a “back buffer”. So, we need a place to render into before moving an image into place for viewing. This is called the Swap Chain.

In essence, the Swap Chain manages one or more image objects that form a sequence of images that can be drawn into and then given to the Surface to be presented to the user for viewing.

Swap Chains are arranged as a ring buffer

Swap Chains are tightly coupled to the window system.

After creating the Swap Chain in the first place, the process for using the Swap Chain is:

1. Ask the Swap Chain for an image
2. Render into it via the Command Buffer and a Queue
3. Return the image to the Swap Chain for presentation
4. Present the image to the viewer (copy to “front buffer”)
What is a Swap Chain?

Because it has the word "chain" in it, let's try to visualize the Swap Chain as a physical chain.

A bicycle chain isn't far off. A bicycle chain goes around and around, each section of the chain taking its turn on the gear teeth, off the gear teeth, on, off, on, off, etc.

Because the Swap Chain is actually a ring buffer, the images in a Swap Chain go around and around too, each image taking its turn being drawn into, being presented, drawn into, being presented etc.

In the same way that bicycle chain links are "re-used", Swap Chain images get re-used too.

We Need to Find Out What our Display Capabilities Are

VulkanDebug.txt output:

We Need to Find Out What our Display Capabilities Are
Creating a Swap Chain

```cpp
uint32_t imageCount; // # of display buffers – 2?
result = vkGetSwapchainImagesKHR( LogicalDevice, IN SwapChain, OUT &imageCount, (VkImage *)nullptr );
PresentImages = new VkImage[ imageCount ];
result = vkGetSwapchainImagesKHR( LogicalDevice, SwapChain, OUT &imageCount, PresentImages );
```

```
PresentImages = new VkImage[ imageCount ];
for( unsigned int i = 0; i < imageCount; i++ )
{
    VkImageViewCreateInfo vivci;
    vivci.sType = VK_STRUCTURE_TYPE_IMAGE_VIEW_CREATE_INFO;
    vivci.pNext = nullptr;
    vivci.flags = 0;
    vivci.viewType = VK_IMAGE_VIEW_TYPE_2D;
    vivci.format = VK_FORMAT_B8G8R8A8_UNORM;
    vivci.components.r = VK_COMPONENT_SWIZZLE_R;
    vivci.components.g = VK_COMPONENT_SWIZZLE_G;
    vivci.components.b = VK_COMPONENT_SWIZZLE_B;
    vivci.components.a = VK_COMPONENT_SWIZZLE_A;
    vivci.subresourceRange.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
    vivci.subresourceRange.baseMipLevel = 0;
    vivci.subresourceRange.levelCount = 1;
    vivci.subresourceRange.baseArrayLayer = 0;
    vivci.subresourceRange.layerCount = 1;
    vivci.image = PresentImages[ i ];
    result = vkCreateImageView( LogicalDevice, IN &vivci, PALLOCATOR, OUT &PresentImageViews[ i ] );
}
```

## Rendering into the Swap Chain, I

```cpp
VkSemaphore imageReadySemaphore;
result = vkCreateSemaphore( LogicalDevice, IN &vsci, PALLOCATOR, OUT &imageReadySemaphore );
```

```
uint32_t nextImageIndex;
uint64_t timeout = UINT64_MAX;
result = vkAcquireNextImageKHR( LogicalDevice, IN SwapChain, IN timeout, IN imageReadySemaphore, IN VK_NULL_HANDLE, OUT &nextImageIndex );
```

```
result = vkBeginCommandBuffer( CommandBuffers[ nextImageIndex ], IN &vcbbi );
```

```
vkCmdBeginRenderPass( CommandBuffers[ nextImageIndex ], IN &vrpbi, IN VK_SUBPASS_CONTENTS_INLINE );
vkCmdBindPipeline( CommandBuffers[ nextImageIndex ], VK_PIPELINE_BIND_POINT_GRAPHICS, GraphicsPipeline );
```

```
vkCmdEndRenderPass( CommandBuffers[ nextImageIndex ] );
vkEndCommandBuffer( CommandBuffers[ nextImageIndex ] );
```
VkFenceCreateInfo vfci;
vfci.sType = VK_STRUCTURE_TYPE_FENCE_CREATE_INFO;
vfci.pNext = nullptr;
vfci.flags = 0;

VkFence renderFence;
vkCreateFence(LogicalDevice, &vfci, PALLOCATOR, OUT &renderFence);

VkQueue presentQueue;
vkGetDeviceQueue(LogicalDevice, FindQueueFamilyThatDoesGraphics(), 0, OUT &presentQueue);

 VkSubmitInfo vsi;
 vsi.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;
vsi.pNext = nullptr;
vsi.waitSemaphoreCount = 1;
vsi.pWaitSemaphores = &imageReadySemaphore;
vsi.pWaitDstStageMask = &waitAtBottom;
vsi.commandBufferCount = 1;
vsi.pCommandBuffers = &CommandBuffers[nextImageIndex];
vsi.signalSemaphoreCount = 0;
vsi.pSignalSemaphores = &SemaphoreRenderFinished;

result = vkQueueSubmit(presentQueue, 1, IN &vsi, IN renderFence); // 1 = submitCount

result = vkWaitForFences(LogicalDevice, 1, IN &renderFence, VK_TRUE, UINT64_MAX);

VkPresentInfoKHR vpi;
 vpi.sType = VK_STRUCTURE_TYPE_PRESENT_INFO_KHR;
vpi.pNext = nullptr;
vpi.waitSemaphoreCount = 0;
vpi.pWaitSemaphores = (VkSemaphore *)nullptr;
vpi.swapchainCount = 1;
vpi.pSwapchains = &SwapChain;
vpi.pImageIndices = &nextImageIndex;
vpi.pResults = (VkResult *)nullptr;

result = vkQueuePresentKHR(presentQueue, IN &vpi);