Render Loop

Initialization
aka boilerplate

Start

Quit request? — yes → End → Cleanup

no

Pump msg
`glfw*Events()`

Have swapchain? ← no ← Surface resized? — yes → Recreate `VkSwapchainKHR`

yes → Render

no

yes

Swapchain Recreation

Start

Stop everything
`vk*WaitIdle()`

Get new caps
`vkGetPhysicalDeviceSurface`
`CapabilitiesKHR()`

Destroy stuff
dependent on
swapchain

- `VkFramebuffer` depends on swapchain images..
- `VkCommandBuffer` depends on that framebuffer.
- `VkPipeline` may depend on resolution (if non-dynamic viewport)
- etc

Window minimized
(or something)?

no

Create new swapchain
`vkCreateSwapchainKHR()`

Recreate swapchain
dependent stuff

End

yes

VkQueue
VkCommandBuffer
VkFences
VkSemaphores
etc.

If this is an interactive app, then there is not much point in keeping more than two frames in Queue. Also, simple app such as this could hold arbitrarily high number of frames in flight, which could interfere with cleanup in the Driver and Layers.

Start

Limit in-flight frames count
vkWaitForFences()

Get swapchain image
vkAcquireNextImageKHR()

Swapchain valid — no → Recreate swapchain

yes

Submit frame rendering command buffer
vkQueueSubmit()

Present frame
vkQueuePresentKHR()

Swapchain valid — no → Recreate swapchain

Swapchain valid — yes → End

Begin recording
vkBeginCommandBuffer()

VkRenderPass
VkFramebuffer → Begin render pass instance
vkCmdBeginRenderPass()

VkPipeline → Bind pipeline
vkCmdBindPipeline()

VkBuffer → Bind vertex buffer
vkCmdBindVertexBuffers()

Draw!
vkCmdDraw()

End render pass instance
vkCmdEndRenderPass()

End recording
vkEndCommandBuffer()

Synchronization

Frame begining

`vkAcquireNextImageKHR()` gives you signal-pending `VkSemaphore`

`vkQueueSubmit()` takes a `VkSemaphore` to wait for

`vkQueueSubmit()` waits on it in `pWaitDstStageMask` stage

This forms a dependency chain. It does not matter so much which stage is used as long as it is equal across the chain. As a matter of style I suggest to use the first "real" stage.

That is `VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT` for a color attachent.

The external subpass dependency should wait on that stage

The external subpass dependency should have `dstStage` appropriate to load operation

Load operation `loadOp`

BTW layout transition happens here

Draw

The external subpass dependency should not block anything

The external subpass dependency should have `srcStage` appropriate to store operation

Store operation `storeOp`

`vkQueueSubmit()` signals a semaphore

That means `dstStage` of `VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT`

`vkQueuePresent()` takes a signal-pending `VkSemaphore`

Freame end

Multiframe Synchronization

Basic Vulkan App

Frame 0 (mod 2)

vkAcquireNextImage · vkQueueSubmit · vkQueuePresent

consumed by (frame 2)

retired by

WAIT F[0] · RESET F[0] · SIGNAL $S_a[0]$ · WAIT $S_a[0]$ · SIGNAL S_r · SIGNAL F[0] · WAIT S_r

consumed by · consumed by

Frame 1 (mod 2)

vkQueueSubmit · vkQueuePresent

retired by (frame1) · consumed by

SIGNAL S_r · WAIT S_r

retired by (frame2)
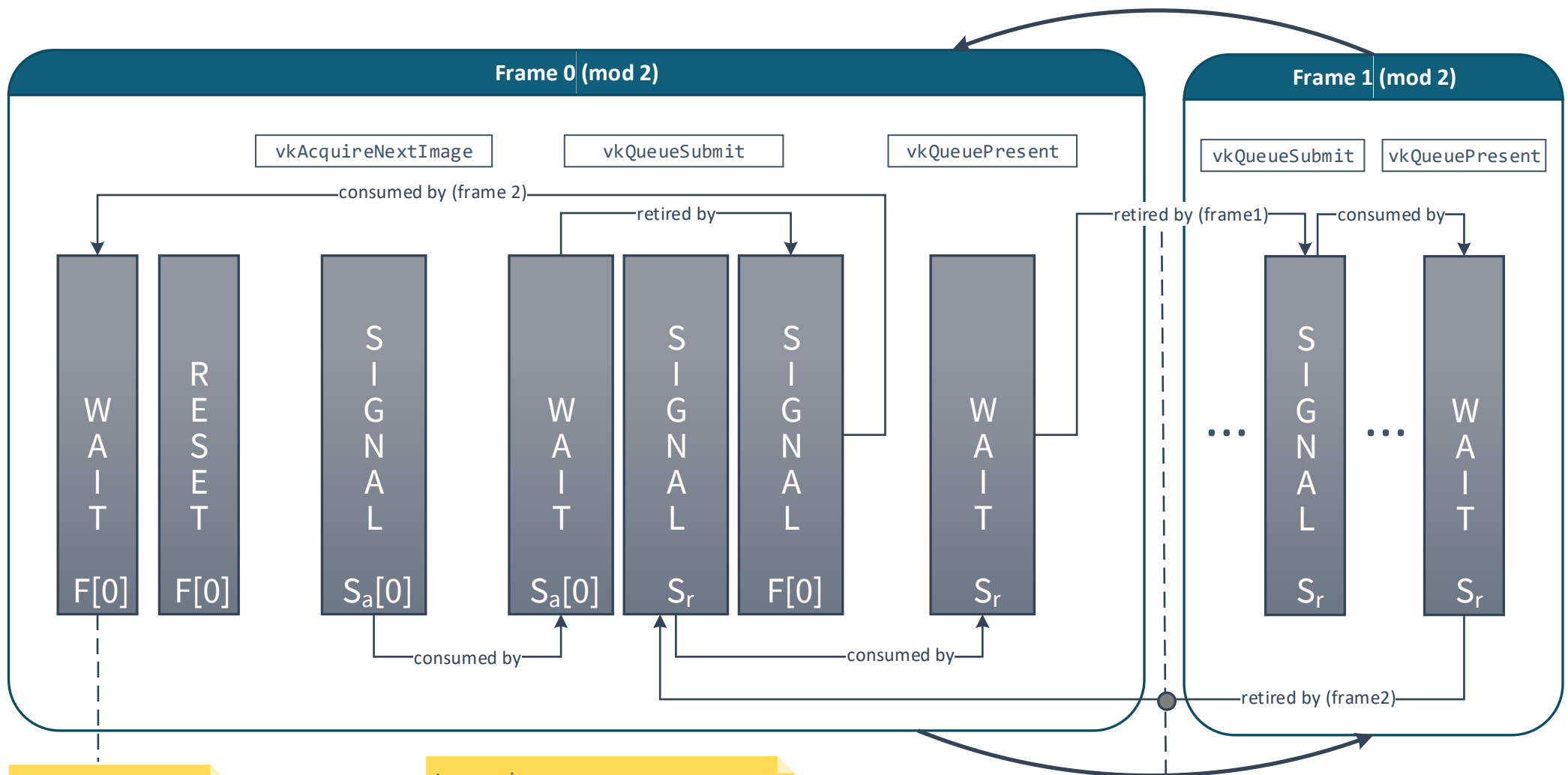
I know I can reuse S_a and S_r here; they are no longer in any pending state.

Legend:
F is `submissionFences`
S_a is `imageReadySs`
S_r is `renderDoneS`

`vkQueueSubmit` synchronization scope with Signal Operation Order guarantees that semaphore of any previous present (S_r) will be unsignaled before its signal op

Begin

You basically just simply create all
the objects you need
`vkCreate*()`

Really, it is just basically filling the creation structs
and in 90 % you can perfectly understand it from
spec or reference without resorting to tutorials or
other middle-men..
`Vk*CreateInfo`

End

Mappable resource creation (VkBuffer\VkImage)

Begin

Useful for the vertex buffer creation

Create resource
`vkCreateImage\Buffer()`

Get resource requirements
`vkGet*MemoryRequirements()`

This is ineficient for complex app. You are instead supposed to suballocate, or use Vulkan Memory Allocator

Create memory
`vkAllocateMemory()`
`VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT`
`VK_MEMORY_PROPERTY_HOST_COHERENT_BIT`

Bind memory to the resource
`vkBind*Memory()`

Map the memory
`vkMapMemory()`

Write your data into the pointer
It will be visible\synchronized with
the next `vkQueueSubmit()`

Copy it to
`VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT`
resource for efficiency

End