

Steps For Blue Screen Template

1. Get The Window Stub Program

2. STEPS FOR INSTANCE EXTENSIONS

1. find how many instance extensions are supported by the vulkan driver of this version and keep it in a local variable.
2. allocate and fill struct VkExtension properties array corresponding to above count.
3. fill and display a local string array of extension names obtained from VkExtension properties.
4. As not required here onwards free VkExtension Array.
5. find whether above extension names contain our required two extensions (VK_KHR_SURFACE_EXTENSION_NAME macro of VK_KHR_surface, VK_KHR_WIN32_SURFACE_EXTENSION_NAME macro of VK_KHR_win32_surface) accordingly set two global variable.
 - a. required extension count.
 - b. required extension name array.
6. As not needed hencefore free local string array.
7. Print whether our Vulkan driver support our required extensions found or not
8. Print only supported extension names.

3. STEPS FOR INSTANCE CREATION

(Do below 4 steps in initialize() and last step in uninitialize())

1. As Explain above fill and initialize required extension names and count global variables.
2. initialize VkApplicationInfo.
3. initialize struct VkInstanceCreateInfo by using information from step 1 and 2.
4. call vkCreateInstance() to get VkInstance in a global variable and do error checking.
5. destroy vkInstance in uninitialize() function.

4. STEPS FOR PRESENTATION SURFACE

1. Declare a global variable to hold presentation surface object.
2. Declare and memset platform specific (windows, Linux, android, etc) surface create info structure.
3. Initialize it particularly its hInstance and hwnd members.
4. Now Call vkCreateWin32SurfaceKHR() to create presentation surface object.

5. STEPS FOR PHYSICAL DEVICE

1. Declare 3 global variables for selected physical device, selected queue family index, for physical device's memory property(required letter).
2. call `vkEnumeratePhysicalDevices()` to get physical device count.
3. allocate `VkPhysicalDevice` array according to above count.
4. Call `vkEnumeratePhysicalDevices()` again to fill above array.
5. Start a loop Using Physical Device count and physical Device array (Note : Declare a boolean `bFound` Variable before this loop which will decide whether we found desired physical device or not)
Inside this loop:
 - a. declare a local variable to queue count.
 - b. call `vkGetPhysicalDeviceQueueFamilyProperties()` to queue count variable.
 - c. allocate `VkQueueFamilyProperty` array according to above count.
 - d. call `vkGetPhysicalDeviceQueueFamilyProperties()` to fill above array.
 - e. Declare `VkBool32` type array and allocate it using the same above queue count.
 - f. Start a nested loop and fill above `VkBool32` type array by calling `vkGetPhysicalDeviceSurfaceSupportKHR()`.
 - g. start another nested loop (not in nested above loop), check whether physical device in its array with its queue family has graphics bit or not, If yes then this is a selected physical device assign it to global variable. similarly, this index is selected queue family index assign it to global variable and set `bFound = true` and break from the second nested loop.
 - h. now we are back in main loop so free the queue family array and `VkBool32` array.
 - i. according to `bFound` variable break out from main loop.
 - j. free physical device array.
6. Do error checking according to the value of the `bFound`.
7. memset the global physical device property structure.
8. initialize above structure by using `vkGetPhysicalDeviceMemoryProperties()`.
9. Declare the a local structure variable `VkPhysicalDeviceFeatures`, memset it and initialize it by calling `vkGetPhysicalDeviceFeatures()`
10. By Using "tessellationShaderMember" of above structure check selected device tessellation Shader support.
11. By Using "geometryShaderMember" of above structure check selected device geometry shader support.
12. There is no need to free / destroy / uninitialized selected physical device. Because later we will create Vulkan logical device which we need to destroy and its destruction will automatically destroy the selected physical device.

6. STEPS FOR PRINT VULKAN INFO

1. Remove Local declarations of `physicalDeviceCount` and `physicalDeviceArray` do it globally from `getPhysicalDevice()`.
2. Accordingly remove `physicalDeviceArray` freeing block from `if (if(bFound == TRUE))` statement and we will later write freeing `physicalDeviceArray` block in `printVKInfo()`.
3. write `printVKInfo()` with following steps:
 - a. Start a loop using global `physicalDeviceCount` and inside it declare and `memset` `VkPhysicalDeviceProperties` struct variable.
 - b. initialize this struct variable by calling `vkGetPhysicalDeviceProperties()` Vulkan API.
 - c. Print vulkan API Version using "apiVersionMember" member of above struct this requires 3 vulkan macros.
 - d. Print device name by using "deviceName" member of above struct.
 - e. Use "deviceType" member of above struct in a switch case block and accordingly print device type.
 - f. Print hexadecimal Vendor id of device using "vendorID" member of above struct.
 - g. Print hexadecimal Device Id using "deviceID" member of above struct.

* NOTE * For the sake of completeness we can repeat step 5 a to h from `getPhysicalDevice()` but now instead of assigning selected queue and selected device print whether this device supports graphic bit, compute bit, transfer bit using if-else if-else if block, Similarly we also can repeat Device features from `getPhysicalDevice()` function and can print all around 50+ device features including supporting tessellation shader and geometry shader.

- h. Free physical device array here which we removed from if block (`if(bFound == TRUE)`) of `getPhysicalDevice()`.

7. STEPS FOR FILLING DEVICE EXTENSIONS

1. using the similar steps of **instance extensions** we create device extensions, change API names, and variable names accordingly. (`VK_KHR_SWAPCHAIN_EXTENSION_NAME`)

8. CREATING VULKAN LOGICAL DEVICE

1. Create A User Defined function "createVulkanDevice()".
2. Call Previously created fill device extension name function in it.
3. Declare and initialize `VkDeviceCreateInfo` structure, use previously obtained device extension count and device extension array to initialize this structure.
4. Now call `vkCreateDevice()` vulkan API to actually create the vulkan logical device and do error checking.
5. Destroy this device when done, Before Destroying the device ensure that all operation on that device are finished. Till then wait on that device using "`vkDeviceWaitIdle()`".

9. DEVICE QUEUE

1. call `vkGetDeviceQueue()` using newly created `vkDevice`, Selected family index, 0th queue in that selected queue family
NOTE : when we create `vulkanDevice` it creates `deviceQueue` automatically and when we uninitialized the `vulkanDevice` it will uninitialized the `deviceQueue`.

10. GET SURFACE FORMAT AND COLOR SPACE

1. call `vkGetPhysicalDeviceSurfaceFormatKHR()` first to retire count of supported format.
2. Declare and allocate array of `VkSurfaceFormat` Structure corresponding to above count, this structure has two members first `VkFormat` and second `VkColorSpaceKHR`.
3. call the same above function again but now to fill above array.
4. According to the contains of above filled array decide the surface color format and surface color space.
5. Free the above array.

11. PRESENT MODE

1. call `vkGetPhysicalDeviceSurfacePresentModesKHR()` first to retire count of supported format.
2. Declare and allocate array of `VkPresentModeKHR` Enum corresponding to above count.
3. call above function again to full above array.
4. According to the contains of above filled array decide the present mode.
5. free the above array

12. STEPS FOR SWAPCHAIN

1. Get Physical device surface supported color format and physical device surface supported color space using previous step No 10.
2. Get Physical device surface capabilities by using vulkan API `vkGetPhysicalDeviceSurfaceCapabilitiesKHR()` and accordingly initialize `VkSurfaceCapabilitiesKHR` structure.
3. By using `minImageCount` and `maxImageCount` members of above structure beside desired image count for swapchain.
4. By using `currentExtent.width` and `currentExtent.height` members of above structure and comparing them with current width and height of window beside image width and image height of swap chain.
5. Decide How we are going to use the swapchain images means whether we are going to store image data and use it letter(deferred Rendering) or we are going to use it as color attachment.
6. Swapchain is capable to storing transformed image before presentation which is called as pre transformed. While creating swapchain we can decide whether to pre transformed or not the swapchain images (pre transformed also include flipping the image).
7. Get present mode for swapchain images using above step 11.
8. According to above data declare, memset and initialize `VkSwapchainCreateInfo` structure.
9. At the end call `vkCreateSwapchainKHR()` vulkan API to create the swapchain.
10. when done destroy it in `uninitialize()` by using `vkDestroySwapchain()` vulkan API.

13. STEPS FOR SWAPCHAIN IMAGES AND IMAGE VIEW

1. Get Swapchain image count in a global variable using `VkGetSwapchainImagesKHR()`.
2. Declare a global `VkImage` type array and allocate it to the swapchain image count using `malloc()`.
3. Now call the same function again from step 1, and fill this array.
4. Declare another global array of type `VkImageView` and allocate it to the size of swapchain image count.
5. Declare and initialize `VkImageViewCreateInfo` structure except its ".image" member.
6. Now Start a Loop for swapchain image count and inside this loop initialize above ".image" member to the swapchain image array index we obtain above, and then call `vkCreateImageView()` API to fill above `VkImageView` Array.
7. In `uninitialize()`, keeping the destructor logic aside, first destroy swapchain images from the swapchain images array in a loop using `vkDestroyImage()`.
8. In `uninitialize()`, now actually free the image array using `free()`.
9. In `uninitialize()`, destroy image view from image view array in a loop by using `vkDestroyImageViews()`.
10. In `uninitialize()`, now actually free the image view array using `free()`.

14. STEPS FOR COMMAND POOL

1. declare and initialize `VkCommandPoolCreateInfo` struct.
2. call `vkCreateCommandPool()` to create the commandPool.
3. destroy command pool using `vkDestroyCommandPool()`.

15. STEPS FOR COMMAND BUFFERS

1. Declare and initialize struct `vkCommandBufferAllocateInfo`
2. NOTE: the number of command buffer are conventionally equal to the number of swapchain images.
3. Declare a command buffer array globally and allocate it to size of swapchain image count.
4. In a loop allocate each array buffer in above array by using `vkAllocateCommandBuffer()`, At a time of allocation of buffer will be empty letter we will fill it for computation or rendering.
5. In `uninitialize()` free each command buffer in a loop of size `swapchainImageCount`.
6. Free the actual command buffer array.

16. STEPS FOR COMMAND BUFFERS

1. declare and initialize `VkAttachmentDescription` array (number of array elements depends upon number of attachments) although we have only one attachment i.e. color attachment we will consider it as array.
2. declare and initialize `VkAttachmentReference` structure which will have information about the attachment we describe above.
3. Declare and initialize `VkSubpassDescription` structure and keep information about above `VkAttachmentReference` structure.
4. Declare and initialize `VkRenderPassCreateInfo` structure and refer `VkAttachmentDescription` and `VkSubpassDescription` into it.
Remember: here also we need to specify interdependency of subpasses if needed and also attachment information in the form of image views which will used by framebuffer later to create the actual `RenderPass`.
5. in `uninitialize` destroy the renderpass by using `vkDestroyRenderPass()`.

17. STEPS FOR COMMAND BUFFERS

1. Declare an array of `VkImageView` equal to number of attachments means in our example array of one attachment.
2. Declare and initialize `VkFramebufferCreateInfo` structure.
3. Allocate the framebuffer array by `malloc` equal to the size of swapchain image count.
4. start a loop for swapchain image count and call `vkCreateFramebuffer()` to create FrameBuffers.
5. In `uninitialize()` destroy framebuffer in a loop for swapchain image count.

18. STEPS FOR FENCES AND SEMAPHORE

1. Globally declare an array of (pointer type) fences of type `VkFence`, additionally declare two semaphore objects of type `VkSemaphore`.
2. In `createSemaphore()` UDF(User Defined Function) declare, `memset` and initialize `VkSemaphoreCreateInfo` structure.
3. now call `vkCreateSemaphore()` 2 times to create our two semaphore objects.

REMEMBER: - both will use same `VkSemaphoreCreateInfo` structure.

- By default, semaphore type is binary semaphore.

4. In `createFences()` UDF declare, `memset` and initialize `VkFenceCreateInfo` structure.
5. in this UDF function allocate our global fence array to the size of swapchain image count using `malloc()`.
6. Now in a loop call `vkCreateFence()` to initialize our global fences array.
7. In `uninitialize()` first in a loop with swapchain image count as counter destroy fence array objects using `vkDestroyFence()` and then actually free the allocated fences array by using `free()`.
8. destroy both global semaphore objects with 2 separate calls to `vkDestroySemaphore()`.

19. STEPS FOR BUILD COMMAND BUFFERS

1. start a loop with `swapchainImageCount` as counter.
2. beside the loop call `vkResetCommandBuffer()` to reset the content of `commandBuffer`.
3. then declare, `memset` and initialize `VkCommandBufferBeginInfo` structure.
4. Now call `vkBeginCommandBuffer()` API to record vulkan drawing related commands do error checking.
5. Declare, `memset` and initialize struct array of `VkClearValue` type internally it is union. Our array will be of 1 element this number depends upon the number of attachments in the `frameBuffer`. As we have only one attachment i.e. color attachment hence our array is of 1 element. when our array becomes size of 2 then the color member is meaningless because is union. to do this initialize `VkClearColorValue` struct to do this declare globally `VkClearColorValue` structure variable and `memset` and initialize it in `inititalize()`.

REMEMBER: We are going to clear `.color` member of `VkClearValue` structure by `VkClearColorValue` structure because in step 16-RenderPass we specified `.loadOp` member of `VkAttachmentDescription` structure to `VK_ATTACHMENT_LOAD_OP_CLEAR`.

6. Then declare `memset` and initialize `VkRenderPassBeginInfo` structure.
7. Begin renderPass by `vkCmdBeginRenderPass()`.
8. REMEMBER : The code written inside "beginRenderPass" and endRenderPass itself is the code of subpass if no subpass is explicitly created. In other word if there is no subpass declared there is always atleast on subpass.
9. End renderPass by calling `vkCmdEndRenderPass()`.
10. End the recording of `commandBuffer` by calling `vkEndCommandBuffer()` and do error checking.
11. close the loop.

