**if** **(**old\_image\_layout **==** VK\_IMAGE\_LAYOUT\_COLOR\_ATTACHMENT\_OPTIMAL**)** **{**

image\_memory\_barrier**.**srcAccessMask **=**

VK\_ACCESS\_COLOR\_ATTACHMENT\_WRITE\_BIT**;**

**}**

**if** **(**new\_image\_layout **==** VK\_IMAGE\_LAYOUT\_TRANSFER\_DST\_OPTIMAL**)** **{**

image\_memory\_barrier**.**dstAccessMask **=** VK\_ACCESS\_TRANSFER\_WRITE\_BIT**;**

**}**

**if** **(**new\_image\_layout **==** VK\_IMAGE\_LAYOUT\_TRANSFER\_SRC\_OPTIMAL**)** **{**

image\_memory\_barrier**.**dstAccessMask **=** VK\_ACCESS\_TRANSFER\_READ\_BIT**;**

**}**

**if** **(**old\_image\_layout **==** VK\_IMAGE\_LAYOUT\_TRANSFER\_DST\_OPTIMAL**)** **{**

image\_memory\_barrier**.**srcAccessMask **=** VK\_ACCESS\_TRANSFER\_WRITE\_BIT**;**

**}**

**if** **(**old\_image\_layout **==** VK\_IMAGE\_LAYOUT\_PREINITIALIZED**)** **{**

image\_memory\_barrier**.**srcAccessMask **=** VK\_ACCESS\_HOST\_WRITE\_BIT**;**

**}**

**if** **(**new\_image\_layout **==** VK\_IMAGE\_LAYOUT\_SHADER\_READ\_ONLY\_OPTIMAL**)** **{**

image\_memory\_barrier**.**srcAccessMask **=**

VK\_ACCESS\_HOST\_WRITE\_BIT **|** VK\_ACCESS\_TRANSFER\_WRITE\_BIT**;**

image\_memory\_barrier**.**dstAccessMask **=** VK\_ACCESS\_SHADER\_READ\_BIT**;**

**}**

**if** **(**new\_image\_layout **==** VK\_IMAGE\_LAYOUT\_COLOR\_ATTACHMENT\_OPTIMAL**)** **{**

image\_memory\_barrier**.**dstAccessMask **=**

VK\_ACCESS\_COLOR\_ATTACHMENT\_WRITE\_BIT**;**

**}**

**if** **(**new\_image\_layout **==** VK\_IMAGE\_LAYOUT\_DEPTH\_STENCIL\_ATTACHMENT\_OPTIMAL**)** **{**

image\_memory\_barrier**.**dstAccessMask **=**

VK\_ACCESS\_DEPTH\_STENCIL\_ATTACHMENT\_WRITE\_BIT**;**

**}**

1. #define GLFW\_INCLUDE\_VULKAN
2. #define GLFW\_EXPOSE\_NATIVE\_WIN32
3. #include <Windows.h>
4. #include <GLFW/glfw3.h>
5. #include <GLFW/glfw3native.h>
6. #include <iostream>
7. #include <vulkan/vulkan.h>
8. #include <vulkan/vulkan\_win32.h>
9. #include <vector>
10. #include <fstream>
11. #include <algorithm>
12. #include <set>
13. #include <numeric>
15. GLFWwindow\* window;
16. VkInstance instance;
17. std::vector<**const** **char**\*> validationLayerNames = { "VK\_LAYER\_LUNARG\_object\_tracker" };
18. **const** **float** width = 800;
19. **const** **float** height = 600;
20. VkSemaphore imageAvailableSemaphore;
21. VkSemaphore renderFinishedSemaphore;
22. VkQueue graphicsQueue;
23. VkQueue presentQueue;
24. VkDevice logicalDevice;
25. VkRenderPassCreateInfo renderPassInfo = {};
26. VkSwapchainKHR swapChain;
27. std::vector<VkCommandBuffer> commandBuffer;
29. **static** VKAPI\_ATTR VkBool32 VKAPI\_CALL debugCallback(
30. VkDebugUtilsMessageSeverityFlagBitsEXT messageSeverity,
31. VkDebugUtilsMessageTypeFlagsEXT messageType,
32. **const** VkDebugUtilsMessengerCallbackDataEXT\* pCallbackData,
33. **void**\* pUserData) {
35. std::cout << "validation layer: " << pCallbackData->pMessage << std::endl;
37. **return** VK\_FALSE;
38. }

41. **int** main()
42. {
43. // 1. Init Window
44. glfwInit();
45. glfwWindowHint(GLFW\_CLIENT\_API, GLFW\_NO\_API);
46. glfwWindowHint(GLFW\_RESIZABLE, GLFW\_FALSE);
47. window = glfwCreateWindow(width, height, "Vulkan window", nullptr, nullptr);
49. // 2. check validation layer support
50. uint32\_t layerCount;
51. vkEnumerateInstanceLayerProperties(&layerCount, nullptr);
52. std::vector<VkLayerProperties> availableLayers(layerCount);
53. vkEnumerateInstanceLayerProperties(&layerCount, availableLayers.data());
55. **for** (auto validationLayerName : validationLayerNames)
56. {
57. **bool** support = **false**;
58. **for** (auto supportedLayerName : availableLayers)
59. {
60. **if** (strcmp(supportedLayerName.layerName, validationLayerName) == 0)
61. {
62. support = **true**;
63. **break**;
64. }
65. }
66. **if** (support == **false**)
67. {
68. std::cout << validationLayerName << " is not supported . " << std::endl;
69. **throw** std::runtime\_error(" not all validation layer is supported . ");
70. }
71. }

74. // 3. Create Instance
75. // 3.1 fill the application info
76. VkApplicationInfo appInfo = {};
77. appInfo.sType = VK\_STRUCTURE\_TYPE\_APPLICATION\_INFO;
78. appInfo.pApplicationName = "Hello Triangle";
79. appInfo.applicationVersion = VK\_MAKE\_VERSION(1, 0, 0);
80. appInfo.pEngineName = "No Engine";
81. appInfo.engineVersion = VK\_MAKE\_VERSION(1, 0, 0);
82. appInfo.apiVersion = VK\_API\_VERSION\_1\_0;
83. appInfo.pNext = NULL;
85. // 3.2 get required extensions
86. uint32\_t glfwExtensionCount = 0;
87. **const** **char**\*\* glfwExtensions;
88. glfwExtensions = glfwGetRequiredInstanceExtensions(&glfwExtensionCount);
90. // 3.3 add validation layer required extension
91. std::vector<**const** **char**\*> required\_extensions(glfwExtensions, glfwExtensions + glfwExtensionCount);
92. required\_extensions.push\_back(VK\_EXT\_DEBUG\_UTILS\_EXTENSION\_NAME);
94. // 3.4 fill the instance create info and create
95. VkInstanceCreateInfo instanceCreateInfo = {};
96. instanceCreateInfo.sType = VK\_STRUCTURE\_TYPE\_INSTANCE\_CREATE\_INFO;
97. instanceCreateInfo.pApplicationInfo = &appInfo;
98. instanceCreateInfo.enabledExtensionCount = **static\_cast**<uint32\_t>(required\_extensions.size());
99. instanceCreateInfo.ppEnabledExtensionNames = required\_extensions.data();
100. instanceCreateInfo.enabledLayerCount = validationLayerNames.size();
101. instanceCreateInfo.ppEnabledLayerNames = validationLayerNames.data();
103. **if** (vkCreateInstance(&instanceCreateInfo, nullptr, &instance) != VK\_SUCCESS) {
104. **throw** std::runtime\_error("failed to create instance!");
105. }
107. // 3.5 set up debug messenger
108. VkDebugUtilsMessengerCreateInfoEXT messengerCreateInfo = {};
109. messengerCreateInfo.sType = VK\_STRUCTURE\_TYPE\_DEBUG\_UTILS\_MESSENGER\_CREATE\_INFO\_EXT;
110. messengerCreateInfo.messageSeverity = VK\_DEBUG\_UTILS\_MESSAGE\_SEVERITY\_VERBOSE\_BIT\_EXT | VK\_DEBUG\_UTILS\_MESSAGE\_SEVERITY\_WARNING\_BIT\_EXT | VK\_DEBUG\_UTILS\_MESSAGE\_SEVERITY\_ERROR\_BIT\_EXT;
111. messengerCreateInfo.messageType = VK\_DEBUG\_UTILS\_MESSAGE\_TYPE\_GENERAL\_BIT\_EXT | VK\_DEBUG\_UTILS\_MESSAGE\_TYPE\_VALIDATION\_BIT\_EXT | VK\_DEBUG\_UTILS\_MESSAGE\_TYPE\_PERFORMANCE\_BIT\_EXT;
112. messengerCreateInfo.pfnUserCallback = debugCallback;
113. messengerCreateInfo.pUserData = nullptr;
115. auto func = (PFN\_vkCreateDebugUtilsMessengerEXT)vkGetInstanceProcAddr(instance, "vkCreateDebugUtilsMessengerEXT");
116. **if** (func == NULL)
117. {
118. **throw** std::runtime\_error("get vkCreateDebugUtilsMessengerEXT fault.");
119. }
120. VkDebugUtilsMessengerEXT debugMessenger;
121. **if** (func(instance, &messengerCreateInfo, NULL, &debugMessenger) != VK\_SUCCESS)
122. {
123. **throw** std::runtime\_error("set debug messenger fault . ");
124. }
126. // 4. create physical device
128. // 4.1 enumerate all physical device
129. VkPhysicalDevice physicalDevice = VK\_NULL\_HANDLE;
130. uint32\_t deviceCount = 0;
131. vkEnumeratePhysicalDevices(instance, &deviceCount, NULL);
132. **if** (deviceCount == 0)
133. {
134. **throw** std::runtime\_error("failed to find physical device .");
135. }
136. std::vector<VkPhysicalDevice> devices(deviceCount);
137. vkEnumeratePhysicalDevices(instance, &deviceCount, devices.data());
139. // 4.2 choose a suitable physical device
140. **for** (**const** auto& device : devices)
141. {
142. VkPhysicalDeviceProperties deviceProperties;
143. VkPhysicalDeviceFeatures deviceFeatures;
144. vkGetPhysicalDeviceFeatures(device, &deviceFeatures);
145. vkGetPhysicalDeviceProperties(device, &deviceProperties);
147. **if** (deviceProperties.deviceType == VK\_PHYSICAL\_DEVICE\_TYPE\_DISCRETE\_GPU &&
148. deviceFeatures.geometryShader)
149. {
150. physicalDevice = device;
151. **break**;
152. }
153. }
155. **if** (physicalDevice == VK\_NULL\_HANDLE)
156. {
157. **throw** std::runtime\_error("no suitable device.");
158. }
160. // 5. create surface
161. VkSurfaceKHR surface;
162. VkWin32SurfaceCreateInfoKHR surfaceCreateInfo = {};
163. surfaceCreateInfo.sType = VK\_STRUCTURE\_TYPE\_WIN32\_SURFACE\_CREATE\_INFO\_KHR;
164. surfaceCreateInfo.hwnd = glfwGetWin32Window(window);
165. surfaceCreateInfo.hinstance = GetModuleHandle(nullptr);
167. **if** (vkCreateWin32SurfaceKHR(instance, &surfaceCreateInfo, NULL, &surface))
168. {
169. **throw** std::runtime\_error(" failed to create window surface .");
170. }
172. // 6. prepare for creating device queue
173. // 6.1 find a queueFamily which can be a graphics queue and a present queue
174. uint32\_t queueFamilyCount = 0;
175. vkGetPhysicalDeviceQueueFamilyProperties(physicalDevice, &queueFamilyCount, NULL);
176. std::vector<VkQueueFamilyProperties> queueFamilies(queueFamilyCount);
177. vkGetPhysicalDeviceQueueFamilyProperties(physicalDevice, &queueFamilyCount, queueFamilies.data());
179. **int** i = 0;
180. **int** queue\_ind = -1;
182. **for** (**const** auto& queueFamily : queueFamilies) {
183. VkBool32 presentSupport = **false**;
184. vkGetPhysicalDeviceSurfaceSupportKHR(physicalDevice, i, surface, &presentSupport);
185. **if** (queueFamily.queueCount > 0 && queueFamily.queueFlags & VK\_QUEUE\_GRAPHICS\_BIT && presentSupport)
186. {
187. queue\_ind = i;
188. }
189. i++;
190. }
192. **if** (queue\_ind == -1) {
193. **throw** std::runtime\_error("No suitable queue .");
194. }
196. // 6.2 fill VkDeviceQueueCreateInfo
197. VkDeviceQueueCreateInfo queueCreateInfo = {};
198. queueCreateInfo.sType = VK\_STRUCTURE\_TYPE\_DEVICE\_QUEUE\_CREATE\_INFO;
199. queueCreateInfo.queueFamilyIndex = queue\_ind;
200. queueCreateInfo.queueCount = 1;
201. **float** queuePriority = 1.0f;
202. queueCreateInfo.pQueuePriorities = &queuePriority;
204. // 7. create logical device and use it to create some resources
206. // 7.1 check whether the physical device support the required extensions
207. **const** std::vector<**const** **char**\*> requireExtensions = {
208. VK\_KHR\_SWAPCHAIN\_EXTENSION\_NAME
209. };
211. uint32\_t availableExtensionCount;
212. vkEnumerateDeviceExtensionProperties(physicalDevice, NULL, &availableExtensionCount, NULL);
213. std::vector<VkExtensionProperties> availableExtensions(availableExtensionCount);
214. vkEnumerateDeviceExtensionProperties(physicalDevice, NULL, &availableExtensionCount, availableExtensions.data());
216. std::set<std::string> requireExtensionNames(requireExtensions.begin(), requireExtensions.end());
217. **for** (**const** auto& extension : availableExtensions)
218. {
219. requireExtensionNames.erase(extension.extensionName);
220. }
222. **if** (!requireExtensionNames.empty()) {
223. **throw** std::runtime\_error("extension not fulfill");
224. }
226. // 7.2 create logical device
227. VkPhysicalDeviceFeatures deviceFeatures = {};
228. VkDeviceCreateInfo deviceCreateInfo = {};
229. deviceCreateInfo.sType = VK\_STRUCTURE\_TYPE\_DEVICE\_CREATE\_INFO;
230. deviceCreateInfo.pQueueCreateInfos = &queueCreateInfo;
231. deviceCreateInfo.queueCreateInfoCount = 1;
232. deviceCreateInfo.pEnabledFeatures = &deviceFeatures;
233. deviceCreateInfo.enabledExtensionCount = requireExtensions.size();
234. deviceCreateInfo.ppEnabledExtensionNames = requireExtensions.data();
236. **if** (vkCreateDevice(physicalDevice, &deviceCreateInfo, NULL, &logicalDevice) != VK\_SUCCESS)
237. {
238. **throw** std::runtime\_error("failed to create logical device.");
239. }
241. // 7.3 retrieve device queue by logical device
242. vkGetDeviceQueue(logicalDevice, queue\_ind, 0, &graphicsQueue);
244. // 7.4 create swap chain by logical device
245. **struct** SwapChainSupportDetails {
246. VkSurfaceCapabilitiesKHR capabilities;
247. std::vector<VkSurfaceFormatKHR> formats;
248. std::vector<VkPresentModeKHR> presentModes;
249. };
251. SwapChainSupportDetails details;
252. vkGetPhysicalDeviceSurfaceCapabilitiesKHR(physicalDevice, surface, &details.capabilities);
254. // 7.4.1 choose the surface format and present mode
255. uint32\_t formatCount;
256. vkGetPhysicalDeviceSurfaceFormatsKHR(physicalDevice, surface, &formatCount, NULL);
257. **if** (formatCount != 0) {
258. details.formats.resize(formatCount);
259. vkGetPhysicalDeviceSurfaceFormatsKHR(physicalDevice, surface, &formatCount, details.formats.data());
260. }
262. uint32\_t presentModeCount;
263. vkGetPhysicalDeviceSurfacePresentModesKHR(physicalDevice, surface, &presentModeCount, NULL);
264. **if** (presentModeCount != 0) {
265. details.presentModes.resize(presentModeCount);
266. vkGetPhysicalDeviceSurfacePresentModesKHR(physicalDevice, surface, &presentModeCount, details.presentModes.data());
267. }
269. **if** (formatCount == 0 || presentModeCount == 0) {
270. **throw** std::runtime\_error(" no suitable format or present mode ");
271. }
273. **int** format\_ind = -1;
274. **for** (**const** auto& availableFormat : details.formats)
275. {
276. format\_ind++;
277. **if** (availableFormat.format == VK\_FORMAT\_R8G8B8A8\_UNORM
278. && availableFormat.colorSpace == VK\_COLOR\_SPACE\_SRGB\_NONLINEAR\_KHR)
279. {
280. **break**;
281. }
282. }
284. **int** present\_mode\_ind = -1;
285. **for** (**const** auto& availablePresentMode : details.presentModes) {
286. present\_mode\_ind++;
287. **if** (availablePresentMode == VK\_PRESENT\_MODE\_MAILBOX\_KHR) {
288. **break**;
289. }
290. }
292. // 7.4.2 choose the extent
293. VkExtent2D actualExtent;
294. actualExtent.width = std::fmax(details.capabilities.minImageExtent.width,
295. std::fmin(details.capabilities.maxImageExtent.width, width));
296. actualExtent.height = std::fmax(details.capabilities.minImageExtent.height,
297. std::fmin(details.capabilities.maxImageExtent.height, height));
298. uint32\_t imageCount = details.capabilities.minImageCount + 1;
300. **if** (details.capabilities.maxImageCount > 0 && imageCount > details.capabilities.maxImageCount) {
301. imageCount = details.capabilities.maxImageCount;
302. }
304. // 7.4.3 fill the VkSwapchainCreateInfoKHR and create .
305. VkSwapchainCreateInfoKHR swapChainCreateInfo = {};
306. swapChainCreateInfo.sType = VK\_STRUCTURE\_TYPE\_SWAPCHAIN\_CREATE\_INFO\_KHR;
307. swapChainCreateInfo.surface = surface;
308. swapChainCreateInfo.minImageCount = imageCount;
309. swapChainCreateInfo.imageFormat = details.formats[format\_ind].format;
310. swapChainCreateInfo.imageColorSpace = details.formats[format\_ind].colorSpace;
311. swapChainCreateInfo.imageExtent = actualExtent;
312. swapChainCreateInfo.imageArrayLayers = 1;
313. swapChainCreateInfo.imageUsage = VK\_IMAGE\_USAGE\_COLOR\_ATTACHMENT\_BIT;
314. swapChainCreateInfo.imageSharingMode = VK\_SHARING\_MODE\_EXCLUSIVE;
315. swapChainCreateInfo.preTransform = details.capabilities.currentTransform;
316. swapChainCreateInfo.compositeAlpha = VK\_COMPOSITE\_ALPHA\_OPAQUE\_BIT\_KHR;
317. swapChainCreateInfo.presentMode = details.presentModes[present\_mode\_ind];
318. swapChainCreateInfo.clipped = VK\_TRUE;
319. swapChainCreateInfo.oldSwapchain = VK\_NULL\_HANDLE;
321. **if** (vkCreateSwapchainKHR(logicalDevice, &swapChainCreateInfo, NULL, &swapChain) != VK\_SUCCESS) {
322. **throw** std::runtime\_error("swap chain create fault ");
323. }
325. //7.5 create swapchain ImageView
326. std::vector<VkImage> swapChainImages;
327. vkGetSwapchainImagesKHR(logicalDevice, swapChain, &imageCount, NULL);
328. swapChainImages.resize(imageCount);
329. vkGetSwapchainImagesKHR(logicalDevice, swapChain, &imageCount, swapChainImages.data());
331. std::vector<VkImageView> swapChainImageViews;
332. swapChainImageViews.resize(swapChainImages.size());
333. **for** (**size\_t** i = 0; i < swapChainImages.size(); i++) {
334. VkImageViewCreateInfo createInfo = {};
335. createInfo.sType = VK\_STRUCTURE\_TYPE\_IMAGE\_VIEW\_CREATE\_INFO;
336. createInfo.image = swapChainImages[i];
337. createInfo.viewType = VK\_IMAGE\_VIEW\_TYPE\_2D;
338. createInfo.format = swapChainCreateInfo.imageFormat;
339. createInfo.components.r = VK\_COMPONENT\_SWIZZLE\_IDENTITY;
340. createInfo.components.g = VK\_COMPONENT\_SWIZZLE\_IDENTITY;
341. createInfo.components.b = VK\_COMPONENT\_SWIZZLE\_IDENTITY;
342. createInfo.components.a = VK\_COMPONENT\_SWIZZLE\_IDENTITY;
343. createInfo.subresourceRange.aspectMask = VK\_IMAGE\_ASPECT\_COLOR\_BIT;
344. createInfo.subresourceRange.baseMipLevel = 0;
345. createInfo.subresourceRange.levelCount = 1;
346. createInfo.subresourceRange.baseArrayLayer = 0;
347. createInfo.subresourceRange.layerCount = 1;
349. **if** (vkCreateImageView(logicalDevice, &createInfo, NULL, &swapChainImageViews[i]) != VK\_SUCCESS) {
350. **throw** std::runtime\_error("failed to create image view.");
351. }
352. }
354. //7.6 create shader module
356. auto loadShaderByteCode = [](**const** std::string& fileName) {
357. std::ifstream loadFile(fileName, std::ios::ate | std::ios::binary);
358. **if** (!loadFile.is\_open())
359. {
360. **throw** std::runtime\_error("failed to open file!");
361. }
362. **size\_t** fileSize = (**size\_t**)loadFile.tellg();
363. std::vector<**char**> buffer(fileSize);
364. loadFile.seekg(0);
365. loadFile.read(buffer.data(), fileSize);
366. loadFile.close();
367. **return** buffer;
368. };
370. auto vertShaderCode = loadShaderByteCode("vert.spv");
371. auto fragShaderCode = loadShaderByteCode("frag.spv");
373. auto createShaderModule = [](**const** std::vector<**char**>& code) {
374. VkShaderModuleCreateInfo createInfo = {};
375. createInfo.sType = VK\_STRUCTURE\_TYPE\_SHADER\_MODULE\_CREATE\_INFO;
376. createInfo.codeSize = code.size();
377. createInfo.pCode = **reinterpret\_cast**<**const** uint32\_t\*>(code.data());
379. VkShaderModule shaderModule;
380. **if** (vkCreateShaderModule(logicalDevice, &createInfo, NULL, &shaderModule) != VK\_SUCCESS)
381. {
382. **throw** std::runtime\_error("failed to create shader module .");
383. }
385. **return** shaderModule;
386. };
388. VkShaderModule vertShaderModule = createShaderModule(vertShaderCode);
389. VkShaderModule fragShaderModule = createShaderModule(fragShaderCode);
391. // 7.7 create pipeline
392. // 7.7.1 prepare shader stage
393. VkPipelineShaderStageCreateInfo vertShaderStageInfo = {};
394. vertShaderStageInfo.sType = VK\_STRUCTURE\_TYPE\_PIPELINE\_SHADER\_STAGE\_CREATE\_INFO;
395. vertShaderStageInfo.stage = VK\_SHADER\_STAGE\_VERTEX\_BIT;
396. vertShaderStageInfo.module = vertShaderModule;
397. vertShaderStageInfo.pName = "main";
399. VkPipelineShaderStageCreateInfo fragShaderStageInfo = {};
400. fragShaderStageInfo.sType = VK\_STRUCTURE\_TYPE\_PIPELINE\_SHADER\_STAGE\_CREATE\_INFO;
401. fragShaderStageInfo.stage = VK\_SHADER\_STAGE\_FRAGMENT\_BIT;
402. fragShaderStageInfo.module = fragShaderModule;
403. fragShaderStageInfo.pName = "main";
405. VkPipelineShaderStageCreateInfo shaderStages[] = { vertShaderStageInfo , fragShaderStageInfo };
407. //7.7.2 prepare vertex input state
408. VkPipelineVertexInputStateCreateInfo vertexInputInfo = {};
409. vertexInputInfo.sType = VK\_STRUCTURE\_TYPE\_PIPELINE\_VERTEX\_INPUT\_STATE\_CREATE\_INFO;
410. vertexInputInfo.vertexBindingDescriptionCount = 0;
411. vertexInputInfo.pVertexBindingDescriptions = NULL;
412. vertexInputInfo.vertexAttributeDescriptionCount = 0;
413. vertexInputInfo.pVertexAttributeDescriptions = NULL;
415. //7.7.3 prepare input assembly state
416. VkPipelineInputAssemblyStateCreateInfo inputAssembly = {};
417. inputAssembly.sType = VK\_STRUCTURE\_TYPE\_PIPELINE\_INPUT\_ASSEMBLY\_STATE\_CREATE\_INFO;
418. inputAssembly.topology = VK\_PRIMITIVE\_TOPOLOGY\_TRIANGLE\_LIST;
419. inputAssembly.primitiveRestartEnable = VK\_FALSE;
421. //7.7.4 prepare viewport and scissor state
422. VkViewport viewport = {};
423. viewport.x = 0.0f;
424. viewport.y = 0.0f;
425. viewport.width = width;
426. viewport.height = height;
427. viewport.minDepth = 0.0f;
428. viewport.maxDepth = 1.0f;
430. VkRect2D scissor = {};
431. scissor.offset = { 0 , 0 };
432. scissor.extent = swapChainCreateInfo.imageExtent;
434. VkPipelineViewportStateCreateInfo viewportState = {};
435. viewportState.sType = VK\_STRUCTURE\_TYPE\_PIPELINE\_VIEWPORT\_STATE\_CREATE\_INFO;
436. viewportState.viewportCount = 1;
437. viewportState.pViewports = &viewport;
438. viewportState.scissorCount = 1;
439. viewportState.pScissors = &scissor;
441. //7.7.5 prepare rasterization state
442. VkPipelineRasterizationStateCreateInfo rasterizer = {};
443. rasterizer.sType = VK\_STRUCTURE\_TYPE\_PIPELINE\_RASTERIZATION\_STATE\_CREATE\_INFO;
444. rasterizer.depthClampEnable = VK\_FALSE;
445. rasterizer.rasterizerDiscardEnable = VK\_FALSE;
446. rasterizer.polygonMode = VK\_POLYGON\_MODE\_FILL;
447. rasterizer.lineWidth = 1.0f;
448. rasterizer.cullMode = VK\_CULL\_MODE\_BACK\_BIT;
449. rasterizer.frontFace = VK\_FRONT\_FACE\_CLOCKWISE;
450. rasterizer.depthBiasEnable = VK\_FALSE;
451. rasterizer.depthBiasConstantFactor = 0.0f;
452. rasterizer.depthBiasClamp = 0.0f;
453. rasterizer.depthBiasSlopeFactor = 0.0f;
455. //7.7.6 prepare multisample state
456. VkPipelineMultisampleStateCreateInfo multisampling = {};
457. multisampling.sType = VK\_STRUCTURE\_TYPE\_PIPELINE\_MULTISAMPLE\_STATE\_CREATE\_INFO;
458. multisampling.sampleShadingEnable = VK\_FALSE;
459. multisampling.rasterizationSamples = VK\_SAMPLE\_COUNT\_1\_BIT;
460. multisampling.minSampleShading = 1.0f;
461. multisampling.pSampleMask = nullptr;
462. multisampling.alphaToCoverageEnable = VK\_FALSE;
463. multisampling.alphaToOneEnable = VK\_FALSE;
465. //7.7.7 prepare color blend state
466. VkPipelineColorBlendAttachmentState colorBlendAttachment = {};
467. colorBlendAttachment.colorWriteMask =
468. VK\_COLOR\_COMPONENT\_R\_BIT | VK\_COLOR\_COMPONENT\_G\_BIT |
469. VK\_COLOR\_COMPONENT\_B\_BIT | VK\_COLOR\_COMPONENT\_A\_BIT;
470. colorBlendAttachment.blendEnable = VK\_FALSE;
471. colorBlendAttachment.srcColorBlendFactor = VK\_BLEND\_FACTOR\_ONE;
472. colorBlendAttachment.dstColorBlendFactor = VK\_BLEND\_FACTOR\_ZERO;
473. colorBlendAttachment.colorBlendOp = VK\_BLEND\_OP\_ADD;
474. colorBlendAttachment.srcAlphaBlendFactor = VK\_BLEND\_FACTOR\_ONE;
475. colorBlendAttachment.dstAlphaBlendFactor = VK\_BLEND\_FACTOR\_ZERO;
476. colorBlendAttachment.alphaBlendOp = VK\_BLEND\_OP\_ADD;
478. VkPipelineColorBlendStateCreateInfo colorBlending = {};
479. colorBlending.sType = VK\_STRUCTURE\_TYPE\_PIPELINE\_COLOR\_BLEND\_STATE\_CREATE\_INFO;
480. colorBlending.logicOpEnable = VK\_FALSE;
481. colorBlending.logicOp = VK\_LOGIC\_OP\_COPY;
482. colorBlending.attachmentCount = 1;
483. colorBlending.pAttachments = &colorBlendAttachment;
484. colorBlending.blendConstants[0] = 0.0f;
485. colorBlending.blendConstants[1] = 0.0f;
486. colorBlending.blendConstants[2] = 0.0f;
487. colorBlending.blendConstants[3] = 0.0f;
489. //7.7.8 create render pass
490. VkAttachmentDescription colorAttachment = {};
491. colorAttachment.format = swapChainCreateInfo.imageFormat;
492. colorAttachment.samples = VK\_SAMPLE\_COUNT\_1\_BIT;
493. colorAttachment.loadOp = VK\_ATTACHMENT\_LOAD\_OP\_CLEAR;
494. colorAttachment.storeOp = VK\_ATTACHMENT\_STORE\_OP\_STORE;
495. colorAttachment.stencilLoadOp = VK\_ATTACHMENT\_LOAD\_OP\_DONT\_CARE;
496. colorAttachment.stencilStoreOp = VK\_ATTACHMENT\_STORE\_OP\_DONT\_CARE;
497. colorAttachment.initialLayout = VK\_IMAGE\_LAYOUT\_UNDEFINED;
498. colorAttachment.finalLayout = VK\_IMAGE\_LAYOUT\_PRESENT\_SRC\_KHR;
500. VkAttachmentReference colorAttachmentRef = {};
501. colorAttachmentRef.attachment = 0;
502. colorAttachmentRef.layout = VK\_IMAGE\_LAYOUT\_COLOR\_ATTACHMENT\_OPTIMAL;
504. VkSubpassDescription subpass = {};
505. subpass.pipelineBindPoint = VK\_PIPELINE\_BIND\_POINT\_GRAPHICS;
506. subpass.colorAttachmentCount = 1;
507. subpass.pColorAttachments = &colorAttachmentRef;
508. VkRenderPass renderPass;
509. VkRenderPassCreateInfo renderPassInfo = {};
511. renderPassInfo.sType = VK\_STRUCTURE\_TYPE\_RENDER\_PASS\_CREATE\_INFO;
512. renderPassInfo.attachmentCount = 1;
513. renderPassInfo.pAttachments = &colorAttachment;
514. renderPassInfo.subpassCount = 1;
515. renderPassInfo.pSubpasses = &subpass;
516. renderPassInfo.dependencyCount = 0;
517. renderPassInfo.pDependencies = NULL;
519. **if** (vkCreateRenderPass(logicalDevice, &renderPassInfo, NULL, &renderPass) != VK\_SUCCESS) {
520. **throw** std::runtime\_error("failed to create render pass!");
521. }
523. //7.7.9 create pipeline layout
524. VkPipelineLayout pipelineLayout;
525. VkPipelineLayoutCreateInfo pipelineLayoutInfo = {};
526. pipelineLayoutInfo.sType = VK\_STRUCTURE\_TYPE\_PIPELINE\_LAYOUT\_CREATE\_INFO;
527. pipelineLayoutInfo.setLayoutCount = 0;
528. pipelineLayoutInfo.pushConstantRangeCount = 0;
530. **if** (vkCreatePipelineLayout(logicalDevice, &pipelineLayoutInfo, NULL, &pipelineLayout) != VK\_SUCCESS)
531. {
532. **throw** std::runtime\_error("failed to create graphics pipeline . ");
533. }
535. // 7.7.10 merge all the state and create graphics pipeline
537. VkGraphicsPipelineCreateInfo pipelineInfo = {};
538. pipelineInfo.sType = VK\_STRUCTURE\_TYPE\_GRAPHICS\_PIPELINE\_CREATE\_INFO;
539. pipelineInfo.stageCount = 2;
540. pipelineInfo.pStages = shaderStages;
541. pipelineInfo.pVertexInputState = &vertexInputInfo;
542. pipelineInfo.pInputAssemblyState = &inputAssembly;
543. pipelineInfo.pViewportState = &viewportState;
544. pipelineInfo.pRasterizationState = &rasterizer;
545. pipelineInfo.pMultisampleState = &multisampling;
546. pipelineInfo.pDepthStencilState = NULL;
547. pipelineInfo.pColorBlendState = &colorBlending;
548. pipelineInfo.pDynamicState = NULL;
549. pipelineInfo.renderPass = renderPass;
550. pipelineInfo.subpass = 0;
551. pipelineInfo.basePipelineHandle = VK\_NULL\_HANDLE;
552. pipelineInfo.layout = pipelineLayout;
554. VkPipeline graphicsPipeline;
555. **if** (vkCreateGraphicsPipelines(logicalDevice, VK\_NULL\_HANDLE, 1, &pipelineInfo, NULL, &graphicsPipeline)
556. != VK\_SUCCESS)
557. {
558. **throw** std::runtime\_error("failed to create graphics pipeline");
559. }
561. vkDestroyShaderModule(logicalDevice, fragShaderModule, nullptr);
562. vkDestroyShaderModule(logicalDevice, vertShaderModule, nullptr);
564. //7.7.11 create frame buffer
565. std::vector<VkFramebuffer> swapChainFrameBuffer;
566. swapChainFrameBuffer.resize(swapChainImageViews.size());
567. **for** (**size\_t** i = 0; i < swapChainImageViews.size(); i++)
568. {
569. VkImageView attachments[] = {
570. swapChainImageViews[i]
571. };
573. VkFramebufferCreateInfo frameBufferInfo = {};
574. frameBufferInfo.sType = VK\_STRUCTURE\_TYPE\_FRAMEBUFFER\_CREATE\_INFO;
575. frameBufferInfo.renderPass = renderPass;
576. frameBufferInfo.width = width;
577. frameBufferInfo.height = height;
578. frameBufferInfo.layers = 1;
579. frameBufferInfo.pAttachments = attachments;
580. frameBufferInfo.attachmentCount = 1;
582. **if** (vkCreateFramebuffer(logicalDevice, &frameBufferInfo, NULL, &swapChainFrameBuffer[i]) != VK\_SUCCESS)
583. {
584. **throw** std::runtime\_error("create frame buffer fault . ");
585. }
586. }
588. //7.7.12 create command buffer
589. VkCommandPool commandPool;
590. VkCommandPoolCreateInfo poolCreateInfo = {};
591. poolCreateInfo.sType = VK\_STRUCTURE\_TYPE\_COMMAND\_POOL\_CREATE\_INFO;
592. poolCreateInfo.queueFamilyIndex = queue\_ind;
593. poolCreateInfo.flags = 0;
595. **if** (vkCreateCommandPool(logicalDevice, &poolCreateInfo, NULL, &commandPool) != VK\_SUCCESS)
596. {
597. **throw** std::runtime\_error("create command pool fault . ");
598. }
600. commandBuffer.resize(swapChainFrameBuffer.size());
602. VkCommandBufferAllocateInfo allocInfo = {};
603. allocInfo.sType = VK\_STRUCTURE\_TYPE\_COMMAND\_BUFFER\_ALLOCATE\_INFO;
604. allocInfo.commandPool = commandPool;
605. allocInfo.level = VK\_COMMAND\_BUFFER\_LEVEL\_PRIMARY;
606. allocInfo.commandBufferCount = (uint32\_t)commandBuffer.size();
608. **if** (vkAllocateCommandBuffers(logicalDevice, &allocInfo, commandBuffer.data()) != VK\_SUCCESS)
609. {
610. **throw** std::runtime\_error("alloc command buffer fault . ");
611. }
613. **for** (**size\_t** i = 0; i < commandBuffer.size(); i++)
614. {
615. VkCommandBufferBeginInfo beginInfo = {};
616. beginInfo.sType = VK\_STRUCTURE\_TYPE\_COMMAND\_BUFFER\_BEGIN\_INFO;
617. beginInfo.flags = 0;
619. **if** (vkBeginCommandBuffer(commandBuffer[i], &beginInfo) != VK\_SUCCESS) {
620. **throw** std::runtime\_error("failed to begin recording command buffer . ");
621. }
623. VkRenderPassBeginInfo renderPassBeginInfo = {};
624. renderPassBeginInfo.sType = VK\_STRUCTURE\_TYPE\_RENDER\_PASS\_BEGIN\_INFO;
625. renderPassBeginInfo.renderPass = renderPass;
626. renderPassBeginInfo.framebuffer = swapChainFrameBuffer[i];
627. renderPassBeginInfo.renderArea.offset = { 0 , 0 };
628. renderPassBeginInfo.renderArea.extent = swapChainCreateInfo.imageExtent;
630. VkClearValue clearColor = { 0.0f , 0.0f , 0.0f , 1.0f };
631. renderPassBeginInfo.clearValueCount = 1;
632. renderPassBeginInfo.pClearValues = &clearColor;
633. vkCmdBeginRenderPass(commandBuffer[i], &renderPassBeginInfo, VK\_SUBPASS\_CONTENTS\_INLINE);
634. vkCmdBindPipeline(commandBuffer[i], VK\_PIPELINE\_BIND\_POINT\_GRAPHICS, graphicsPipeline);
635. vkCmdDraw(commandBuffer[i], 3, 1, 0, 0);
636. vkCmdEndRenderPass(commandBuffer[i]);
637. **if** (vkEndCommandBuffer(commandBuffer[i]) != VK\_SUCCESS)
638. {
639. **throw** std::runtime\_error("failed to record command buffer . ");
640. }
642. }
644. // 7.7.13 create semaphore
645. VkSemaphoreCreateInfo semaphoreInfo = {};
646. semaphoreInfo.sType = VK\_STRUCTURE\_TYPE\_SEMAPHORE\_CREATE\_INFO;
648. **if** (vkCreateSemaphore(logicalDevice, &semaphoreInfo, NULL, &imageAvailableSemaphore) != VK\_SUCCESS ||
649. vkCreateSemaphore(logicalDevice, &semaphoreInfo, NULL, &renderFinishedSemaphore) != VK\_SUCCESS)
650. {
651. **throw** std::runtime\_error("failed to create semaphore . ");
652. }

655. **while** (!glfwWindowShouldClose(window)) {
656. glfwPollEvents();
657. uint32\_t imageIndex;
659. uint64\_t limit = (std::numeric\_limits<uint64\_t>::max)();
660. vkAcquireNextImageKHR(logicalDevice, swapChain, limit, imageAvailableSemaphore, VK\_NULL\_HANDLE, &imageIndex);
662. VkSubmitInfo submitInfo = {};
663. submitInfo.sType = VK\_STRUCTURE\_TYPE\_SUBMIT\_INFO;
665. VkSemaphore waitSemaphores[] = { imageAvailableSemaphore };
666. VkPipelineStageFlags waitStages[] = { VK\_PIPELINE\_STAGE\_COLOR\_ATTACHMENT\_OUTPUT\_BIT };
667. submitInfo.waitSemaphoreCount = 1;
668. submitInfo.pWaitSemaphores = waitSemaphores;
669. submitInfo.pWaitDstStageMask = waitStages;
670. submitInfo.commandBufferCount = 1;
671. submitInfo.pCommandBuffers = &commandBuffer[imageIndex];
673. VkSemaphore signalSemaphores[] = { renderFinishedSemaphore };
674. submitInfo.signalSemaphoreCount = 1;
675. submitInfo.pSignalSemaphores = signalSemaphores;
677. **if** (vkQueueSubmit(graphicsQueue, 1, &submitInfo, VK\_NULL\_HANDLE) != VK\_SUCCESS) {
678. **throw** std::runtime\_error("failed to submit draw command buffer . ");
679. }
681. VkPresentInfoKHR presentInfo = {};
682. presentInfo.sType = VK\_STRUCTURE\_TYPE\_PRESENT\_INFO\_KHR;
683. presentInfo.waitSemaphoreCount = 1;
684. presentInfo.pWaitSemaphores = signalSemaphores;
686. VkSwapchainKHR swapChains[] = { swapChain };
687. presentInfo.swapchainCount = 1;
688. presentInfo.pSwapchains = swapChains;
689. presentInfo.pImageIndices = &imageIndex;
690. presentInfo.pResults = NULL;
691. vkQueuePresentKHR(graphicsQueue, &presentInfo);
692. }
694. //clean up
695. **for** (auto framebuffer : swapChainFrameBuffer) {
696. vkDestroyFramebuffer(logicalDevice, framebuffer, nullptr);
697. }
698. //  vkDestroySemaphore(logicalDevice, renderFinishedSemaphore, nullptr);
699. vkDestroySemaphore(logicalDevice, imageAvailableSemaphore, nullptr);
700. vkDestroyCommandPool(logicalDevice, commandPool, nullptr);
702. vkDestroyPipeline(logicalDevice, graphicsPipeline, nullptr);
703. vkDestroyPipelineLayout(logicalDevice, pipelineLayout, nullptr);
704. vkDestroyRenderPass(logicalDevice, renderPass, nullptr);
706. **for** (auto imageView : swapChainImageViews) {
707. vkDestroyImageView(logicalDevice, imageView, nullptr);
708. }
710. vkDestroySwapchainKHR(logicalDevice, swapChain, nullptr);
711. vkDestroyDevice(logicalDevice, nullptr);
713. auto destroyFunc = (PFN\_vkDestroyDebugUtilsMessengerEXT)vkGetInstanceProcAddr(instance, "vkDestroyDebugUtilsMessengerEXT");
714. **if** (destroyFunc != nullptr) {
715. destroyFunc(instance, debugMessenger, NULL);
716. }
717. **else** {
718. **throw** std::runtime\_error(" not find function PFN\_vkDestroyDebugUtilsMessengerEXT . ");
719. }
720. vkDestroySurfaceKHR(instance, surface, nullptr);
721. vkDestroyInstance(instance, nullptr);
723. glfwDestroyWindow(window);
724. glfwTerminate();
726. **return** 0;
727. }