```
/**************** 1. 枚举实例的层属性 ****************/
// 获取实例层的数量
uint32_t instanceLayerCount;

// 通过使用第二个参数为NULL的方式来返回层的数量
vkEnumerateInstanceLayerProperties(&instanceLayerCount, NULL);

VkLayerProperties *layerProperty = NULL;
vkEnumerateInstanceLayerProperties(&instanceLayerCount, layerProperty);

// 遍历每一个可用的实例层，获取功能扩展信息
foreach  layerProperty{
    VkExtensionProperties *instanceExtensions;
    res = vkEnumerateInstanceExtensionProperties(layer_name, &instanceExtensionCount,
instanceExtensions);
}

/**************** 2. 创建实例 ****************/
VkInstance instance;    // Vulkan实例对象
VkInstanceCreateInfo instanceInfo    = {};

// 设置实例中需要开启的层的名称
instanceInfo.ppEnabledLayerNames     = {"VK_LAYER_LUNARG_standard_validation",
                                         "VK_LAYER_LUNARG_object_tracker" };

// 设置实例中需要开启的功能扩展
instanceInfo.ppEnabledExtensionNames = {VK_KHR_SURFACE_EXTENSION_NAME,
                                         VK_KHR_WIN32_SURFACE_EXTENSION_NAME};

// 创建实例对象
vkCreateInstance(&instanceInfo, NULL, &instance);

/**************** 3. 枚举物理设备 ****************/
VkPhysicalDevice                gpu;        // 物理设备
uint32_t                        gpuCount;   // 物理设备数量
std::vector<VkPhysicalDevice>   gpuList;    // 物理设备列表
// 获取GPU的数量
vkEnumeratePhysicalDevices(instance, &gpuCount, NULL);

// 获取GPU的信息
vkEnumeratePhysicalDevices(instance, &gpuCount, gpuList);

/**************** 4. 创建逻辑设备 ****************/
// 获取队列和队列类型
vkGetPhysicalDeviceQueueFamilyProperties(gpu, &queueCount, queueProperties);

// 获取物理设备或者GPU的内存(显存)属性
vkGetPhysicalDeviceMemoryProperties(gpu, &memoryProperties);

// 获取物理设备或者GPU的属性
vkGetPhysicalDeviceProperties(gpu, &gpuProps);

// 从物理设备创建逻辑设备对象
VkDevice            device; // 逻辑设备
VkDeviceCreateInfo  deviceInfo = {};
vkCreateDevice(gpuList[0], &deviceInfo, NULL, &device);

/**************** 5. 呈现初始化 ****************/
```

```
//
// 创建空窗口
CreateWindowEx(...);        /*Windows平台*/
xcb_create_window(...);     /*Linux平台*/

// 查询WSI扩展函数
// vkCreateSwapchainKHR .....

// 创建抽象表面对象
VkWin32SurfaceCreateInfoKHR createInfo = {};
vkCreateWin32SurfaceKHR(instance, &createInfo, NULL, &surface);

// 在所有队列中选择一个支持呈现的队列
foreach Queue in All Queues{
    vkGetPhysicalDeviceSurfaceSupportKHR(gpu, queueIndex, surface, &isPresentaionSupported);
    // 保存该队列的索引
    if (isPresentaionSupported) {
        graphicsQueueFamilyIndex = Queue.index;
        break;
    }
}

// 获取兼容呈现的图形队列
vkGetDeviceQueue(device, graphicsQueueFamilyIndex, 0, &queue);

// 分配内从控件来记录绘制表面的格式总数
uint32_t formatCount;
vkGetPhysicalDeviceSurfaceFormatsKHR(gpu, surface, &formatCount, NULL);

VkSurfaceFormatKHR *surfaceFormats = allocate memory('formatCount' * VkSurfaceFormatKHR);

// 将表面格式保存到VkSurfaceFormatKHR对象中
vkGetPhysicalDeviceSurfaceFormatsKHR(gpu, surface, &formatCount, surfaceFormats);

/**************** 6. 创建交换链 ****************/

// 开始录入指令到指令缓存
vkBeginCommandBuffer(cmd, &cmdBufInfo);

// 获取表面能力的相关参数
vkGetPhysicalDeviceSurfaceCapabilitiesKHR(gpu, surface, &surfCapabilities);

// 获取表面呈现模式
vkGetPhysicalDeviceSurfacePresentModesKHR(gpu, surface, &presentModeCount, NULL);
VkPresentModeKHR presentModes[presentModeCount];
vkGetPhysicalDeviceSurfacePresentModesKHR(gpu, surface, &presentModeCount, presentModes);

// 创建交换链
VkSwapchainCreateInfoKHR swapChainInfo = {};
fpCreateSwapchainKHR(device, &swapChainInfo, NULL, &swapChain);

// 创建所需交换链图像对应的图像视图
vkGetSwapchainImagesKHR(device, swapChain, &swapchainImageCount, NULL);
VkImage swapchainImages[swapchainImageCount];
vkGetSwapchainImagesKHR(device, swapChain, &swapchainImageCount, swapchainImages);

// 获取交换链中的图像
foreach swapchainImages{
    // 设置布局方式，与具体驱动方式的实现兼容
    SetImageLayout();
    // 插入流水线屏障
    VkImageMemoryBarrier imgMemoryBarrier = { ... };
    vkCmdPipelineBarrier(cmd,srcStages,destStages,0,0,NULL,0,NULL,1,&imgMemoryBarrier);
    SwapChainBuffer scBuffer = {...};
    // 创建图像视图
    vkCreateImageView(device, &colorImageView, NULL, &scBuffer.view);
    // 保存图像视图，稍后供应用程序使用
    buffers.push_back(scBuffer);
}

/**************** 7. 创建深度缓冲 ****************/

// 查询当前物理设备支持的深度格式
vkGetPhysicalDeviceFormatProperties(gpuList, depthFormat, &properties);

// 创建图像对象
vkCreateImage(device, &imageInfo, NULL, &imageObject);

// 获取图像资源所需的内存大小
vkGetImageMemoryRequirements(device, image, &memRequirements);

// 分配内存
vkAllocateMemory(device, &memAlloc, NULL, &memorys);

// 绑定内存
vkBindImageMemory(device, imageObject, mem, 0);

// 设置图像布局，可用于当前设备
SetImageLayout(. . .)

// 插入新的流水线屏障，确保刚才设置的图像布局在图像被真正的使用之前就已经被创建好了
vkCmdPipelineBarrier(cmd, srcStages, destStages, 0, 0, NULL, 0, NULL, 1, &imgPipelineBarrier);

// 创建图像视图
vkCreateImageView(device, &imgViewInfo, NULL, &view);


/**************** 8. 构建着色器模块 ****************/
#version 450
layout (location = 0) in vec4 pos;
layout (location = 1) in vec4 inColor;
layout (location = 0) out vec4 outColor;
out gl_PerVertex {
    vec4 gl_Position;
};
void main() {
    outColor = inColor;
    gl_Position = pos;
    gl_Position.y = -gl_Position.y;
    gl_Position.z = (gl_Position.z + gl_Position.w) / 2.0;
}

#version 450
layout (location = 0) in vec4 color;
layout (location = 0) out vec4 outColor;
void main() {
    outColor = color;
};

VkPipelineShaderStageCreateInfo vtxShdrStages = {....};
VkShaderModuleCreateInfo moduleCreateInfo = { ... };
```

```
moduleCreateInfo.pCode = spvVertexShaderData/*着色器编译好的SPIRV数据*/;
// 在逻辑设备上创建着色器模块
vkCreateShaderModule(device, &moduleCreateInfo, NULL, &vtxShdrStages.module);

/**************** 9. 创建描述符以及流水线布局 ****************/

// 描述符的定义类型与着色器中是对应的
VkDescriptorSetLayoutBinding layoutBindings[2];
layoutBindings[0].descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
layoutBindings[0].binding        = 0;
layoutBindings[0].stageFlags     = VK_SHADER_STAGE_VERTEX_BIT;
layoutBindings[1].descriptorType = VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER;
layoutBindings[1].binding        = 0;
layoutBindings[1].stageFlags     = VK_SHADER_STAGE_FRAGMENT_BIT;

// 设置布局绑定，创建描述符集
VkDescriptorSetLayoutCreateInfo descriptorLayout = {};
descriptorLayout.pBindings                        = layoutBindings;
VkDescriptorSetLayout descLayout[2];
vkCreateDescriptorSetLayout(device, &descriptorLayout, NULL, descLayout.data());

// 现在我们可以使用描述符来创建一个流水线布局了
VkPipelineLayoutCreateInfo pipelineLayoutCI = { ... };
pipelineLayoutCI.pSetLayouts                = descLayout.data();
vkCreatePipelineLayout(device, &pipelineLayoutCI, NULL, &pipelineLayout);


/**************** 10. 渲染通道 ****************/

// 定义两个附件，分别对盈利颜色和深度缓冲
VkAttachmentDescription attachments[2];
attachments[0].format = colorImageformat;
attachments[0].loadOp = clear ? VK_ATTACHMENT_LOAD_OP_CLEAR
                            : VK_ATTACHMENT_LOAD_OP_DONT_CARE;
attachments[1].format = depthImageformat;
attachments[1].loadOp = VK_ATTACHMENT_LOAD_OP_CLEAR;

VkAttachmentReference colorReference, depthReference = {...};

// 使用颜色图像和深度图像来构建子通道
VkSubpassDescription subpass     = {};
subpass.pColorAttachments        = &colorReference;
subpass.pDepthStencilAttachment = &depthReference;

// 定义渲染通道控制的结构体
VkRenderPassCreateInfo rpInfo   = { &attachments,&subpass ...};

VkRenderPass renderPass; // Create Render Pass object
vkCreateRenderPass(device, &rpInfo, NULL, &renderPass);

/**************** 11. 创建帧缓存 ****************/
VkImageView attachments[2];                      // [0]表示颜色, [1] 深度缓存
attachments[1] = Depth.view;

VkFramebufferCreateInfo fbInfo = {};
fbInfo.renderPass               = renderPass;   // 渲染缓存对象
fbInfo.pAttachments             = attachments;  // 图像视图附件
fbInfo.width                    = width;        // 帧缓存高度
fbInfo.height                   = height;       // 帧缓存宽度

// 为交换链中的每个图像的帧缓存对象分配内存，每个图像只是有一个帧缓存
```

```
VkFramebuffer framebuffers[交换链中的图像数量];

foreach(drawing buffer in swapchain) {
    attachments[0] = currentSwapChainDrawImage.view;
    vkCreateFramebuffer(device, &fbInfo, NULL, &framebuffers[i]);
}

/****************  12. 生成几何体，在GPU内存中保存顶点 ****************/

static const VertexWithColor triangleData[] = {
    /*{  x ,     y,    z,    w,    r,    g,    b,   a },*/
    { 0.0f,  1.0f, 0.0f, 1.0f, 1.0f, 0.0f, 0.0f, 1.0 },
    { 1.0f, -1.0f, 0.0f, 1.0f, 0.0f, 0.0f, 1.0f, 1.0 },
    { -1.0f, -1.0f, 0.0f, 1.0f, 0.0f, 1.0f, 0.0f, 1.0 },
};

VkBuffer                buffer;
VkMemoryRequirements    mem_requirement;
VkDeviceMemory          deviceMemory;

// 创建缓存对象，查询所需的内存空间，分配内存
VkBufferCreateInfo buffer_info = { ... };
vkCreateBuffer(device, &buffer_info, NULL, &buffer);

vkGetBufferMemoryRequirements(device, buffer, &mem_requirement);

VkMemoryAllocateInfo alloc_info = { ... };

vkAllocateMemory(device, &alloc_info, NULL, &(deviceMemory));

// 通过内存映射的方式，将三角形集合数据复制到GPU内存中
uint8_t *pData;
vkMapMemory(device, deviceMemory, 0, mem_requirement.size, 0, &pData);
memcpy(pData, triangleData, dataSize); /**** 复制数据 ****/
vkUnmapMemory(device, deviceMemory);

// 绑定分配后的内存
vkBindBufferMemory(device, buffer, deviceMemory, 0);

/****************  13.顶点绑定 ****************/
VkVertexInputBindingDescription viBinding;
viBinding.binding = 0;
viBinding.inputRate = VK_VERTEX_INPUT_RATE_VERTEX;
viBinding.stride = sizeof(triangleData) /*数据间隔*/;

VkVertexInputAttributeDescription viAttribs[2];
viAttribs[0].binding = 0;
viAttribs[0].location = 0;
viAttribs[0].format = VK_FORMAT_R32G32B32A32_SFLOAT;
viAttribs[0].offset = 0;
viAttribs[1].binding = 0;
viAttribs[1].location = 1;
viAttribs[1].format = VK_FORMAT_R32G32B32A32_SFLOAT;
viAttribs[1].offset = 16;

/****************  14. 定义状态 ****************/
// Vertex Input state
VkPipelineVertexInputStateCreateInfo vertexInputStateInfo   = { ... };
vertexInputStateInfo.vertexBindingDescriptionCount          = 1;
vertexInputStateInfo.pVertexBindingDescriptions             = &viBinding;
vertexInputStateInfo.vertexAttributeDescriptionCount        = 2;
vertexInputStateInfo.pVertexAttributeDescriptions           = viAttribs;

// 动态状态
VkPipelineDynamicStateCreateInfo dynamicState               = { ... };
// 输入装配状态控制的结构体
VkPipelineInputAssemblyStateCreateInfo inputAssemblyInfo    = { ... };
// 光栅化状态控制的结构体
VkPipelineRasterizationStateCreateInfo rasterStateInfo      = { ... };
// 颜色混合附件状态控制的结构体
VkPipelineColorBlendAttachmentState colorBlendSI            = { ... };
// 颜色混合状态控制的结构体
VkPipelineColorBlendStateCreateInfo colorBlendStateInfo     = { ... };
// 视口状态控制的结构体
VkPipelineViewportStateCreateInfo viewportStateInfo         = { ... };
// 深度/模板状态控制的结构体
VkPipelineDepthStencilStateCreateInfo depthStencilStateInfo = { ... };
// 多重采样状态控制的结构体
VkPipelineMultisampleStateCreateInfo   multiSampleStateInfo = { ... };

/****************  15. 创建图形流水线 ****************/
VkPipelineCache pipelineCache;
VkPipelineCacheCreateInfo pipelineCacheInfo;
vkCreatePipelineCache(device, &pipelineCacheInfo, NULL, &pipelineCache);

// 定义图形流水线的控制参数结构体
VkGraphicsPipelineCreateInfo pipelineInfo;
pipelineInfo.layout                 = pipelineLayout;
pipelineInfo.pVertexInputState      = &vertexInputStateInfo;
pipelineInfo.pInputAssemblyState    = &inputAssemblyInfo;
pipelineInfo.pRasterizationState    = &rasterStateInfo;
pipelineInfo.pColorBlendState       = &colorBlendStateInfo;
pipelineInfo.pMultisampleState      = &multiSampleStateInfo;
pipelineInfo.pDynamicState          = &dynamicState;
pipelineInfo.pViewportState         = &viewportStateInfo;
pipelineInfo.pDepthStencilState     = &depthStencilStateInfo;
pipelineInfo.pStages                = shaderStages;
pipelineInfo.stageCount             = 2;
pipelineInfo.renderPass             = renderPass;

// 创建图形流水线
vkCreateGraphicsPipelines(device, pipelineCache, 1, &pipelineInfo, NULL, &pipeline);

/****************  16. 获取渲染图像 ****************/

// 定义图像获取操作的同步信号量。只有当图像可用时，才会执行渲染操作。
VkSemaphore imageAcquiredSemaphore ;
VkSemaphoreCreateInfo imageAcquiredSemaphore CreateInfo = { ... };
imageAcquiredSemaphore CreateInfo.sType = VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO;
vkCreateSemaphore(device, &imageAcquiredSemaphoreCreateInfo,
                    NULL, &imageAcquiredSemaphore );

// 获取交换链中下一帧中可用的图像的索引:
vkAcquireNextImageKHR(device, swapChain, UINT64_MAX,
                        imageAcquiredSemaphore , NULL, &currentSwapChainImageIndex);

/****************  17. 准备渲染通道的控制结构体 ****************/

// 定义清屏颜色以及深度/模板值
const VkClearValue clearValues[2] = {
    [0] = { .color.float32 = { 0.2f, 0.2f, 0.2f, 0.2f } },
    [1] = { .depthStencil = { 1.0f, 0 } },
};

// 帧缓存的渲染通道执行数据结构
VkRenderPassBeginInfo renderPassBegin;
renderPassBegin.sType       = VK_STRUCTURE_TYPE_RENDER_PASS_BEGIN_INFO;
renderPassBegin.pNext       = NULL;
renderPassBegin.renderPass  = renderPass;
renderPassBegin.framebuffer = framebuffers[currentSwapchainImageIndex];
renderPassBegin.renderArea.offset.x         = 0;
renderPassBegin.renderArea.offset.y         = 0;
renderPassBegin.renderArea.extent.width     = width;
renderPassBegin.renderArea.extent.height    = height;
renderPassBegin.clearValueCount             = 2;
renderPassBegin.pClearValues                = clearValues;


/****************  18. 渲染通道执行 ****************/
/**** 开始渲染通道 ****/
vkCmdBeginRenderPass(cmd, &renderPassBegin, VK_SUBPASS_CONTENTS_INLINE);

vkCmdBindPipeline(cmd, VK_PIPELINE_BIND_POINT_GRAPHICS, pipeline); // 绑定流水线
const VkDeviceSize offsets[1] = { 0 };
vkCmdBindVertexBuffers(cmd, 0, 1, &buffer, offsets);     // 绑定三角形缓存数据
vkCmdSetViewport(cmd, 0, NUM_VIEWPORTS, &viewport); // 视口 = {0, 0, 500, 500, 0 ,1}
vkCmdSetScissor(cmd, 0, NUM_SCISSORS, &scissor);     // 裁剪器  = {0, 0, 500, 500}
vkCmdDraw(cmd, 3, 1, 0, 0);          //绘制指令 - 3 顶点, 1 实例, 初始索引为0

/**** 结束渲染通道 ****/
vkCmdEndRenderPass(cmd);

// 设置交换链图像布局
setImageLayout()
vkCmdPipelineBarrier(cmd, ....);

/**** 结束指令缓存的录入 ****/
vkEndCommandBuffer(cmd);

/****************  19. 提交到队列 ****************/

VkFenceCreateInfo fenceInfo = { ... }; VkFence drawFence;
// 创建围栏对象，用于等待指令缓存执行完成
vkCreateFence(device, &fenceInfo, NULL, &drawFence);

// 填充命令缓存提交结构体
VkSubmitInfo submitInfo[1]          = { ... };
submitInfo[0].pNext                 = NULL;
submitInfo[0].sType                 = VK_STRUCTURE_TYPE_SUBMIT_INFO;
submitInfo[0].pWaitSemaphores       = &imageAcquiredSemaphore ;
submitInfo[0].commandBufferCount    = 1;
submitInfo[0].pCommandBuffers       = &cmd;

// 将指令缓存提交到队列中执行
vkQueueSubmit(queue, 1, submitInfo, NULL);

/****************  20. 在窗口呈现绘制结果 ****************/

// 定义交换链图像呈现数据结构
VkPresentInfoKHR present = { ... };
present.sType           = VK_STRUCTURE_TYPE_PRESENT_INFO_KHR;
present.pNext           = NULL;
present.swapchainCount  = 1;
present.pSwapchains      = &swapChain;
present.pImageIndices    = &swapChainObjCurrent_buffer;

// 检查是不是之前发送的围栏已经完成
do {
    res = vkWaitForFences(device, 1, &drawFence, VK_TRUE, FENCE_TIMEOUT);
} while (res == VK_TIMEOUT);

// 将当前的交换链图像和换到呈现队列，准备呈现到输出设备
vkQueuePresentKHR(queue, &present);

// 删除同步对象
vkDestroySemaphore(device, imageAcquiredSemaphore , NULL);
vkDestroyFence(device, drawFence, NULL);
```