Documentation and learning material

Little Vulkan Engine "Lve"

January 2022

Fievez Thibault

Contents

1 Introduction

Vulkan is an open source cross platform modern graphic API used in various industry.Vulkan targets high-performance real-time 3D graphics applications. It is the successor of the famous OpenGL as both belongs to the non-profit Khronos Group. The particularities of Vulkan is that it is known have a steep learning curve and to be more complex than OpenGL as more tasks have to be taken care of by the programmer allowing for better optimisation.

This paper serves as general documentation for the little Vulkan engine I created, and it mainly focus on the "setup" of Vulkan to at least display a simple triangle, should anyone would like to learn or study this code. It is not meant to be neither a full blown or step by step tutorial. **The code I produced is by itself very much commented** (some may say a bit too much), this document is just a complement of information,it is especially useful as a complement for part of the code less commented. It is a helper document regrouping useful concepts , diagrams, code snipset, comments and implementations. It is not meant to be "stand-alone".

This code has been made through the help of various sources,but first and foremost it follows the structure of this amazing Youtube series made by Brendan Galea [1], alongside of following the official Vulkan tutorial [2].
Last but not least, I extensively used the resources and information showcased on GpuOpen [3], an AMD driven platform that provides free learnings as well as samples and technologies using Vulkan and other API. In other words this last website is a gold mine of information, and will appear several times within this paper.

More than anything, it is more than useful to cross information sources, and I recommend having the "Understanding Vulkan® Objects" article [4] written by Adam Sawicki on GpuOpen opened at all time alongside reading this paper. A really useful object relation/diagram taken from this article is showcased on figure 1.

[1] https://www.youtube.com/c/BrendanGalea/videos
[2] https://vulkan-tutorial.com/
[3] https://gpuopen.com/
[4] https://gpuopen.com/learn/understanding-vulkan-objects/

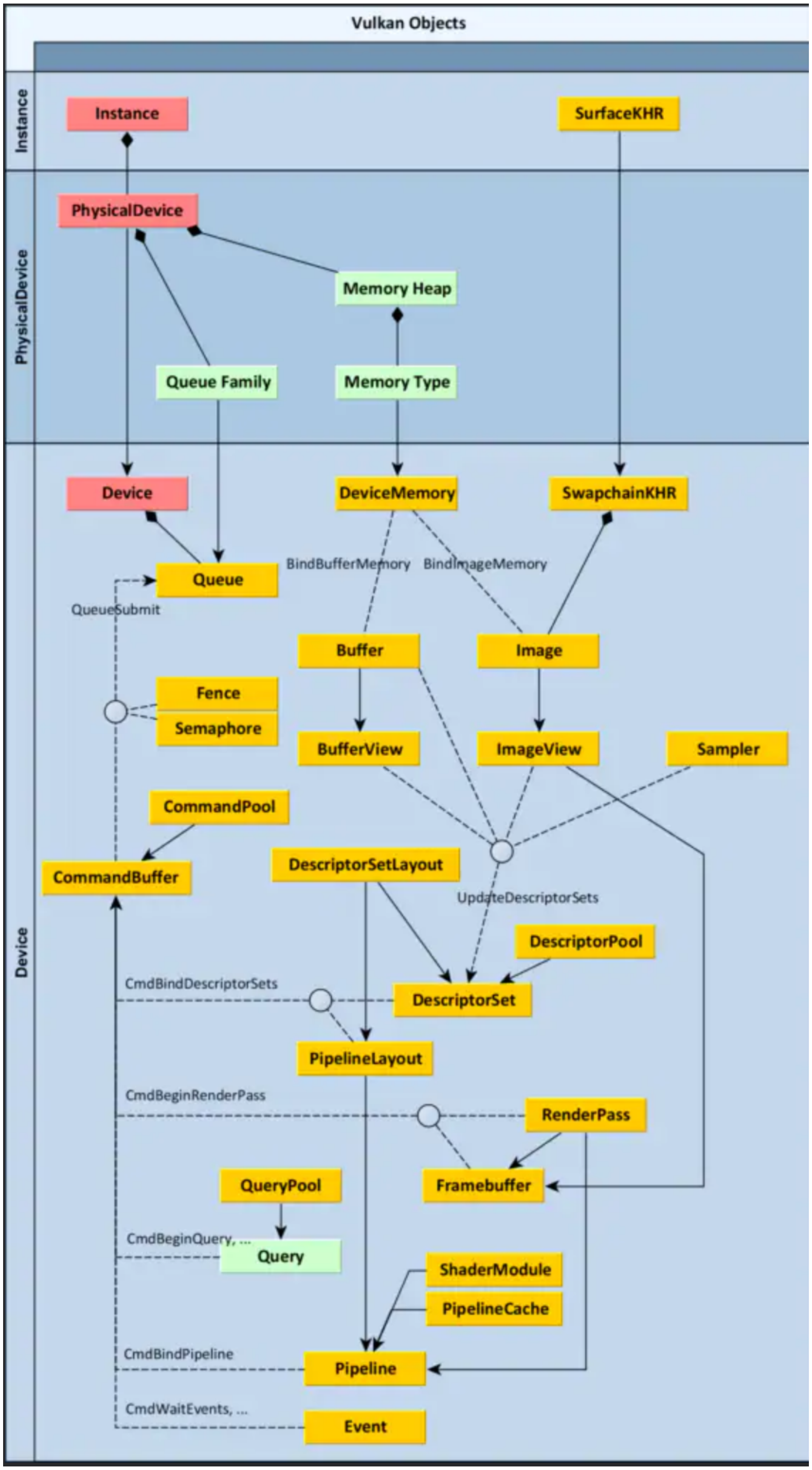


Figure 1: Vulkan object relation diagram. It is very handy to keep this diagram for reading through the code or this document. [3]

2 Code Overview

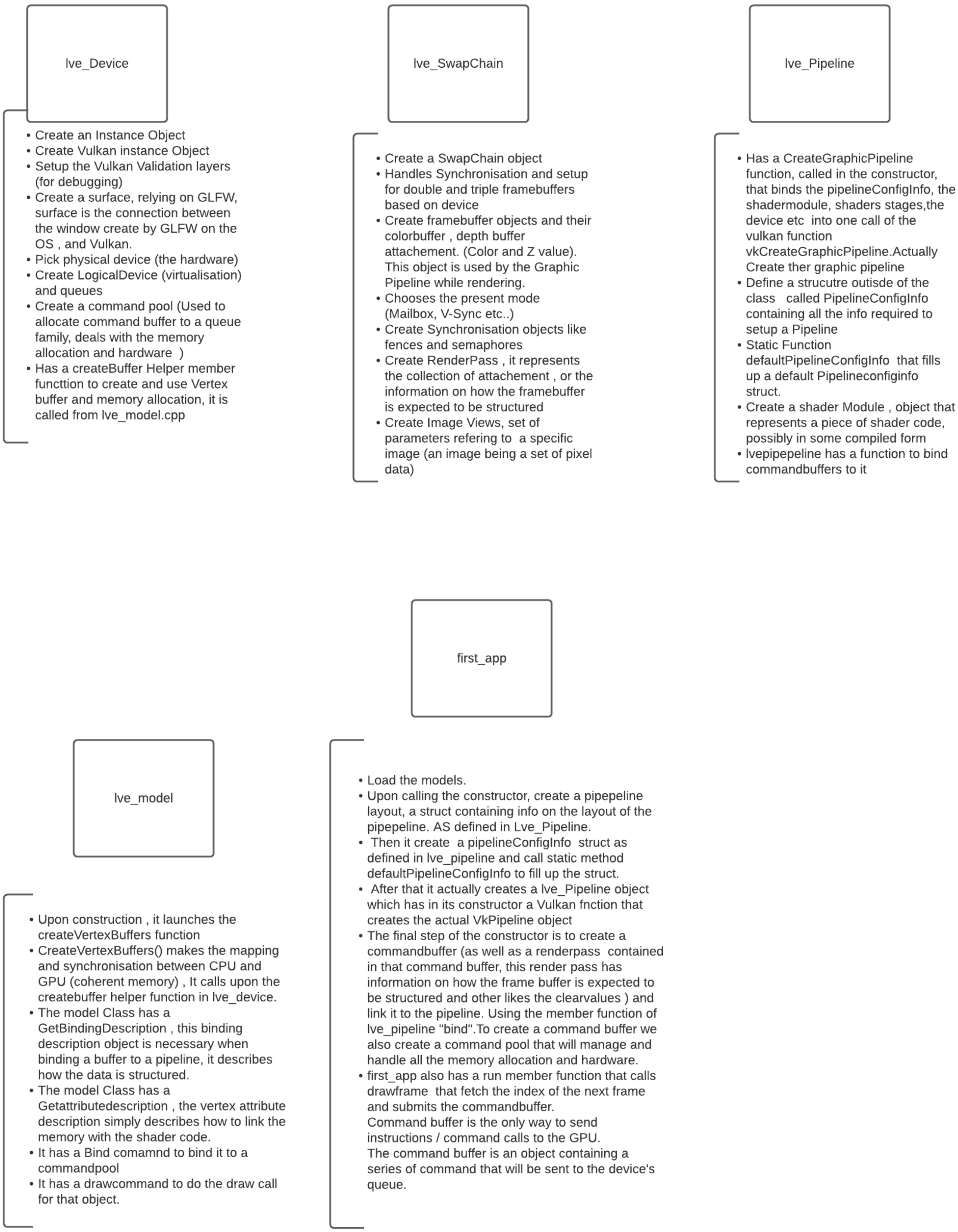
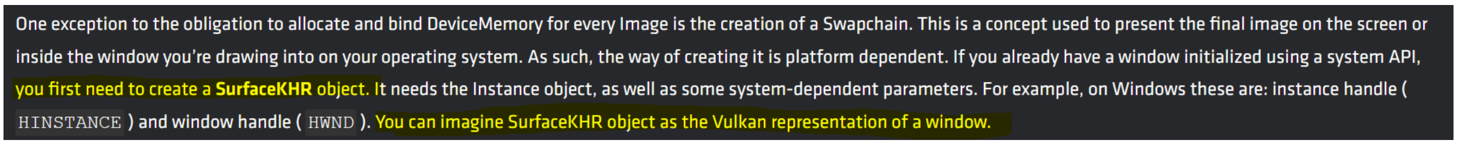


Figure 2: Classes Overview

3 Lve-Window

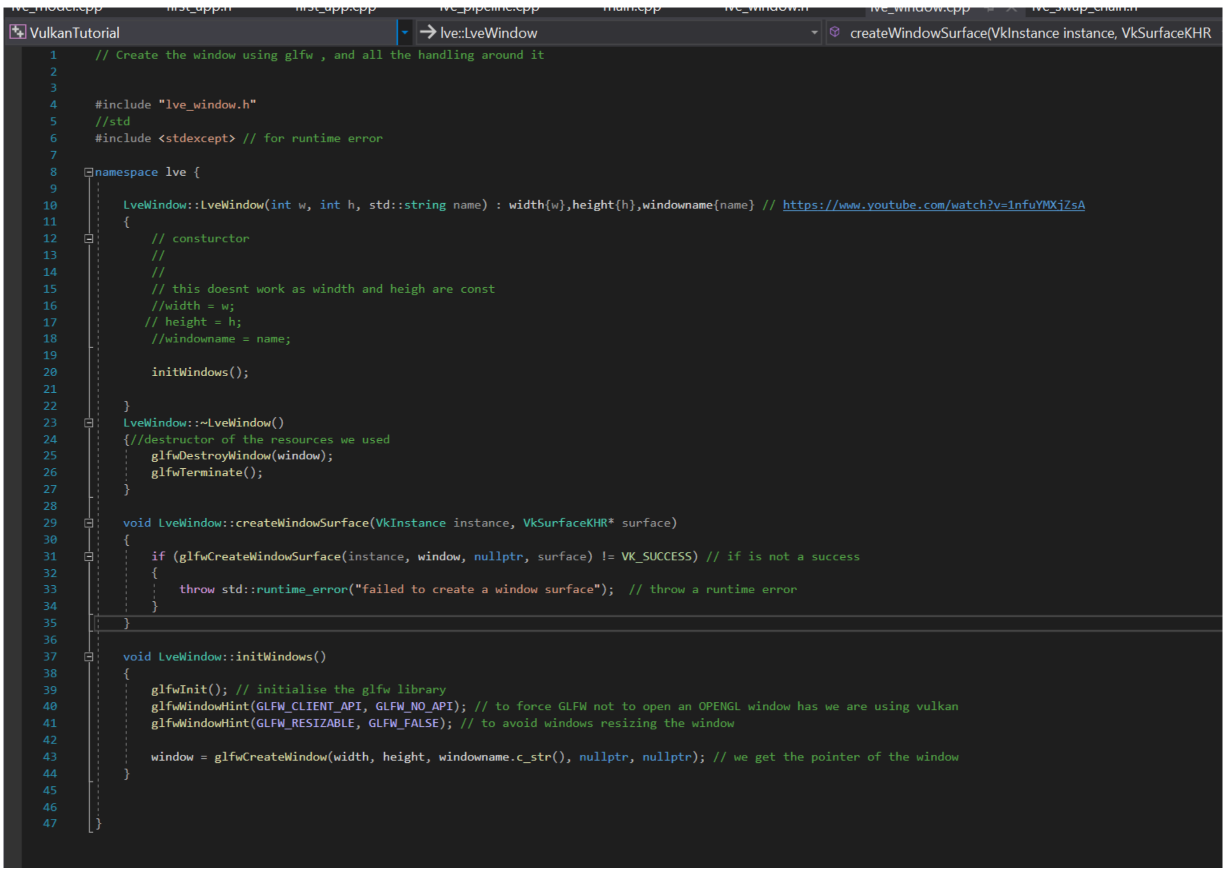
This class , called lveWindow, simply has a task of creating a window using GLFW. GLFW is originally an open source API for OpengGl that is meant to provide a simple way for creating windows, contexts and surfaces, receiving input and events. It also works with Vulkan and is used here to create windows.

In the world of Vulkan , the equivalent object representing a window is called a Surface, as seen on figure 3.



One exception to the obligation to allocate and bind DeviceMemory for every Image is the creation of a Swapchain. This is a concept used to present the final image on the screen or inside the window you're drawing into on your operating system. As such, the way of creating it is platform dependent. If you already have a window initialized using a system API, you first need to create a **SurfaceKHR** object. It needs the Instance object, as well as some system-dependent parameters. For example, on Windows these are: instance handle (`HINSTANCE`) and window handle (`HWND`). You can imagine SurfaceKHR object as the Vulkan representation of a window.

Figure 3: The Vulkan equivalent of a window to display is called a Surface. [3]




```cpp
// Create the window using glfw , and all the handling around it


#include "lve_window.h"
//std
#include <stdexcept> // for runtime error

namespace lve {

    LveWindow::LveWindow(int w, int h, std::string name) : width{w},height{h},windowname{name} // https://www.youtube.com/watch?v=1nfuYMXjZsA
    {
        // consturctor
        //
        //
        // this doesnt work as windth and heigh are const
        //width = w;
       // height = h;
        //windowname = name;

        initWindows();

    }
    LveWindow::~LveWindow()
    {//destructor of the resources we used
        glfwDestroyWindow(window);
        glfwTerminate();
    }

    void LveWindow::createWindowSurface(VkInstance instance, VkSurfaceKHR* surface)
    {
        if (glfwCreateWindowSurface(instance, window, nullptr, surface) != VK_SUCCESS) // if is not a success
        {
            throw std::runtime_error("failed to create a window surface");  // throw a runtime error
        }
    }

    void LveWindow::initWindows()
    {
        glfwInit(); // initialise the glfw library
        glfwWindowHint(GLFW_CLIENT_API, GLFW_NO_API); // to force GLFW not to open an OPENGL window has we are using vulkan
        glfwWindowHint(GLFW_RESIZABLE, GLFW_FALSE); // to avoid windows resizing the window

        window = glfwCreateWindow(width, height, windowname.c_str(), nullptr, nullptr); // we get the pointer of the window
    }


}
```

Figure 4: lve-window.cpp deals with the creating of a window thanks to the GLFW library.

4 Lve-Device

The main role of that class is as follow, when constructed it automatically do the following:

- Creating a device object.
- Creating a Vulkan Instance object and Vulkan device object.
- Setting up Validation layers , by default the Vulkan API doesn't have a validation layer in order to improve performance, it has to be enabled manually.
- Creating a surface , relying on GLFW and the lve-window class of section 1.
- Pick a physical device (the GPU hardware , there could be many GPU available in one computer).
- Create a Logical Device (virtualisation).
- Create a command pool (used to allocate command buffers to a queue family, also deals with the memory allocation and the hardware).

The header on figure5 can give a rough idea of the different tasks achieved through LveDevice. It is especially useful to have a look at the constructor on figure 6.

```cpp
#pragma once

#include "lve_window.h"

// std lib headers
#include <string>
#include <vector>

namespace lve {

struct SwapChainSupportDetails {
  VkSurfaceCapabilitiesKHR capabilities;
  std::vector<VkSurfaceFormatKHR> formats;
  std::vector<VkPresentModeKHR> presentModes;
};

struct QueueFamilyIndices {
  uint32_t graphicsFamily;
  uint32_t presentFamily;
  bool graphicsFamilyHasValue = false;
  bool presentFamilyHasValue = false;
  bool isComplete() { return graphicsFamilyHasValue && presentFamilyHasValue; }
};

class LveDevice {
 public:
#ifdef NDEBUG
  const bool enableValidationLayers = false;
#else
  const bool enableValidationLayers = true;
#endif

  LveDevice(LveWindow &window);
  ~LveDevice();

  // Not copyable or movable
  LveDevice(const LveDevice &) = delete;
  void operator=(const LveDevice &) = delete;
  LveDevice(LveDevice &&) = delete;
  LveDevice &operator=(LveDevice &&) = delete;

  VkCommandPool getCommandPool() { return commandPool; }
  VkDevice device() { return device_; }
  VkSurfaceKHR surface() { return surface_; }
  VkQueue graphicsQueue() { return graphicsQueue_; }
  VkQueue presentQueue() { return presentQueue_; }

  SwapChainSupportDetails getSwapChainSupport() { return querySwapChainSupport(physicalDevice); }
  uint32_t findMemoryType(uint32_t typeFilter, VkMemoryPropertyFlags properties);
  QueueFamilyIndices findPhysicalQueueFamilies() { return findQueueFamilies(physicalDevice); }
  VkFormat findSupportedFormat(
      const std::vector<VkFormat> &candidates, VkImageTiling tiling, VkFormatFeatureFlags features);

  // Buffer Helper Functions
  void createBuffer(
      VkDeviceSize size,
      VkBufferUsageFlags usage,
      VkMemoryPropertyFlags properties,
      VkBuffer &buffer,
      VkDeviceMemory &bufferMemory);
  VkCommandBuffer beginSingleTimeCommands();
  void endSingleTimeCommands(VkCommandBuffer commandBuffer);
  void copyBuffer(VkBuffer srcBuffer, VkBuffer dstBuffer, VkDeviceSize size);
  void copyBufferToImage(
      VkBuffer buffer, VkImage image, uint32_t width, uint32_t height, uint32_t layerCount);

  void createImageWithInfo(
      const VkImageCreateInfo &imageInfo,
      VkMemoryPropertyFlags properties,
      VkImage &image,
      VkDeviceMemory &imageMemory);

  VkPhysicalDeviceProperties properties;

 private:
  void createInstance();
  void setupDebugMessenger();
  void createSurface();
  void pickPhysicalDevice();
  void createLogicalDevice();
  void createCommandPool();

  // helper functions
  bool isDeviceSuitable(VkPhysicalDevice device);
  std::vector<const char *> getRequiredExtensions();
  bool checkValidationLayerSupport();
  QueueFamilyIndices findQueueFamilies(VkPhysicalDevice device);
  void populateDebugMessengerCreateInfo(VkDebugUtilsMessengerCreateInfoEXT &createInfo);
  void hasGflwRequiredInstanceExtensions();
  bool checkDeviceExtensionSupport(VkPhysicalDevice device);
  SwapChainSupportDetails querySwapChainSupport(VkPhysicalDevice device);

  VkInstance instance;
  VkDebugUtilsMessengerEXT debugMessenger;
  VkPhysicalDevice physicalDevice = VK_NULL_HANDLE;
  LveWindow &window;
  VkCommandPool commandPool;

  VkDevice device_;
  VkSurfaceKHR surface_;
  VkQueue graphicsQueue_;
  VkQueue presentQueue_;

  const std::vector<const char *> validationLayers = {"VK_LAYER_KHRONOS_validation"};
  const std::vector<const char *> deviceExtensions = {VK_KHR_SWAPCHAIN_EXTENSION_NAME};
};

}  // namespace lve
```

Figure 5: lve-device.h Overview of the device class.

```cpp
// class member functions
LveDevice::LveDevice(LveWindow &window) : window{window} {   // CONSTRUCTOR
  createInstance(); // CREATE A VULKAN INSTANCE , initanlise the vulkan library and is the connectio0n between or application and vulkan
  setupDebugMessenger(); // Validation layers , by default the vulkan API has very smal validation layer and error handling ,
  //small error wont get caught and ends up in crash , for debugging this shold be enabled , and disabled for realease
  createSurface(); // create surface relying on GLFW , connection between the window i created and vulkan ability to display result
  pickPhysicalDevice(); // pick the physical device , is the graphic hardware
  createLogicalDevice(); // create logical device , what features of our physical device we want to use
  createCommandPool(); // convenient to create a command pool for later use , help for command buffer allocation later on
}

LveDevice::~LveDevice() {
  vkDestroyCommandPool(device_, commandPool, nullptr);
  vkDestroyDevice(device_, nullptr);

  if (enableValidationLayers) {
    DestroyDebugUtilsMessengerEXT(instance, debugMessenger, nullptr);
  }

  vkDestroySurfaceKHR(instance, surface_, nullptr);
  vkDestroyInstance(instance, nullptr);
}
```

Figure 6: The constructor of LveDevice.cpp. It follows the same flow as the bullet points above.

4.1 Creating an Instance

The createInstance member function called at the construction of Lve-device is showcased on figure 7.

```
void LveDevice::createInstance() {
  if (enableValidationLayers && !checkValidationLayerSupport()) {
    throw std::runtime_error("validation layers requested, but not available!");
  }

  VkApplicationInfo appInfo = {};
  appInfo.sType = VK_STRUCTURE_TYPE_APPLICATION_INFO;
  appInfo.pApplicationName = "LittleVulkanEngine App";
  appInfo.applicationVersion = VK_MAKE_VERSION(1, 0, 0);
  appInfo.pEngineName = "No Engine";
  appInfo.engineVersion = VK_MAKE_VERSION(1, 0, 0);
  appInfo.apiVersion = VK_API_VERSION_1_0;

  VkInstanceCreateInfo createInfo = {};
  createInfo.sType = VK_STRUCTURE_TYPE_INSTANCE_CREATE_INFO;
  createInfo.pApplicationInfo = &appInfo;

  auto extensions = getRequiredExtensions();
  createInfo.enabledExtensionCount = static_cast<uint32_t>(extensions.size());
  createInfo.ppEnabledExtensionNames = extensions.data();

  VkDebugUtilsMessengerCreateInfoEXT debugCreateInfo;
  if (enableValidationLayers) {
    createInfo.enabledLayerCount = static_cast<uint32_t>(validationLayers.size());
    createInfo.ppEnabledLayerNames = validationLayers.data();

    populateDebugMessengerCreateInfo(debugCreateInfo);
    createInfo.pNext = (VkDebugUtilsMessengerCreateInfoEXT *)&debugCreateInfo;
  } else {
    createInfo.enabledLayerCount = 0;
    createInfo.pNext = nullptr;
  }

  if (vkCreateInstance(&createInfo, nullptr, &instance) != VK_SUCCESS) {
    throw std::runtime_error("failed to create instance!");
  }

  hasGflwRequiredInstanceExtensions();
}
```

Figure 7: Creating a Vulkan Instance object.Check for validation layers availability.

To have a better understanding of what is an Vulkan instance object, it's one of the first Vulkan object to be created on figure 1. See the definition of an instance on Vulkan on figure 8.

Instance is the first object you create. It represents the connection from your application to the Vulkan runtime and therefore only should exist once in your application. It also stores all application specific state required to use Vulkan. Therefore you must specify all layers (like the Validation Layer) and all extensions you want to enable when creating an Instance.

Figure 8: The instance Object. [3]

When the instance is created, required Extensions for the good functioning of Vulkan are also fetched as seen on figure 9

```cpp
std::vector<const char *> LveDevice::getRequiredExtensions() {
  uint32_t glfwExtensionCount = 0;
  const char **glfwExtensions;
  glfwExtensions = glfwGetRequiredInstanceExtensions(&glfwExtensionCount);

  std::vector<const char *> extensions(glfwExtensions, glfwExtensions + glfwExtensionCount);

  if (enableValidationLayers) {
    extensions.push_back(VK_EXT_DEBUG_UTILS_EXTENSION_NAME);
  }

  return extensions;
}

void LveDevice::hasGflwRequiredInstanceExtensions() {
  uint32_t extensionCount = 0;
  vkEnumerateInstanceExtensionProperties(nullptr, &extensionCount, nullptr);
  std::vector<VkExtensionProperties> extensions(extensionCount);
  vkEnumerateInstanceExtensionProperties(nullptr, &extensionCount, extensions.data());

  std::cout << "available extensions:" << std::endl;               // THIS IS WHERE WE PRINT THE EXTENSION WE NEED AND THOPSE WE HAVE
  std::unordered_set<std::string> available;
  for (const auto &extension : extensions) {
    std::cout << "\t" << extension.extensionName << std::endl;
    available.insert(extension.extensionName);
  }

  std::cout << "required extensions:" << std::endl;
  auto requiredExtensions = getRequiredExtensions();
  for (const auto &required : requiredExtensions) {
    std::cout << "\t" << required << std::endl;
    if (available.find(required) == available.end()) {
      throw std::runtime_error("Missing required glfw extension");
    }
  }
}

bool LveDevice::checkDeviceExtensionSupport(VkPhysicalDevice device) {
  uint32_t extensionCount;
  vkEnumerateDeviceExtensionProperties(device, nullptr, &extensionCount, nullptr);

  std::vector<VkExtensionProperties> availableExtensions(extensionCount);
  vkEnumerateDeviceExtensionProperties(
      device,
      nullptr,
      &extensionCount,
      availableExtensions.data());

  std::set<std::string> requiredExtensions(deviceExtensions.begin(), deviceExtensions.end());

  for (const auto &extension : availableExtensions) {
    requiredExtensions.erase(extension.extensionName);
  }

  return requiredExtensions.empty();
}
```

Figure 9: Extension check at Instance creation.

4.2 Setup Debug Messenger and Validation layer

By default Vulkan doesn't run validation layers, in order to maximise performance, those must be enabled manually for while working on a debug build and disabled for finished products.

```cpp
void LveDevice::setupDebugMessenger() {
  if (!enableValidationLayers) return;
  VkDebugUtilsMessengerCreateInfoEXT createInfo;
  populateDebugMessengerCreateInfo(createInfo);
  if (CreateDebugUtilsMessengerEXT(instance, &createInfo, nullptr, &debugMessenger) != VK_SUCCESS) {
    throw std::runtime_error("failed to set up debug messenger!");
  }
}
```

Figure 10: Validation Layer Setup.

This function also sets up the debug messenger to communicate with the programmer.

4.3 Creating a Surface

This section is really simple, as the surface creation processes has been done on section 3, Lve-window.

```cpp
void LveDevice::createSurface() { window.createWindowSurface(instance, &surface_); }
```

Figure 11: Create a surface, simple call to the window object of section 3

4.4 Pick Physical Device

Here again, it is quite self-explanatory. The programs looks for Vulkan suitable devices.

```cpp
void LveDevice::pickPhysicalDevice() {
  uint32_t deviceCount = 0;
  vkEnumeratePhysicalDevices(instance, &deviceCount, nullptr);
  if (deviceCount == 0) {
    throw std::runtime_error("failed to find GPUs with Vulkan support!");
  }
  std::cout << "Device count: " << deviceCount << std::endl;
  std::vector<VkPhysicalDevice> devices(deviceCount);
  vkEnumeratePhysicalDevices(instance, &deviceCount, devices.data());

  for (const auto &device : devices) {
    if (isDeviceSuitable(device)) {
      physicalDevice = device;
      break;
    }
  }

  if (physicalDevice == VK_NULL_HANDLE) {
    throw std::runtime_error("failed to find a suitable GPU!");
  }

  vkGetPhysicalDeviceProperties(physicalDevice, &properties);
  std::cout << "physical device: " << properties.deviceName << std::endl;
}
```

Figure 12: Pick a physical device.

PhysicalDevice represents a specific Vulkan-compatible device, like a graphics card. You enumerate these from "Instance" and you can then query them for their `vendorID`, `deviceID`, and supported features, as well as other properties and limits.

PhysicalDevice can enumerate all available types of **Queue Families**. The graphics queue is the main one, but you may also have additional ones that support only Compute or Transfer.

PhysicalDevice can also enumerate Memory Heaps and Memory Types inside them. A **Memory Heap** represents a specific pool of RAM. It may abstract your system RAM on the motherboard or a certain memory space in video RAM on a dedicated graphics card, or any other host- or device-specific memory the implementation wants to expose. You must specify the **Memory Type** when allocating memory. It holds specific requirements for the memory blob like visible to the host, coherent (between CPU and GPU) and cached. There may be an arbitrary combination of these, depending on the device driver.

Figure 13: What is a physical device in Vulkan. [3]

4.5 Create Logical device

This next function creates a logical device (virtualisation layer) and setup its queues.

```cpp

void LveDevice::createLogicalDevice() {
  QueueFamilyIndices indices = findQueueFamilies(physicalDevice);

  std::vector<VkDeviceQueueCreateInfo> queueCreateInfos;
  std::set<uint32_t> uniqueQueueFamilies = {indices.graphicsFamily, indices.presentFamily};

  float queuePriority = 1.0f;
  for (uint32_t queueFamily : uniqueQueueFamilies) {
    VkDeviceQueueCreateInfo queueCreateInfo = {};
    queueCreateInfo.sType = VK_STRUCTURE_TYPE_DEVICE_QUEUE_CREATE_INFO;
    queueCreateInfo.queueFamilyIndex = queueFamily;
    queueCreateInfo.queueCount = 1;
    queueCreateInfo.pQueuePriorities = &queuePriority;
    queueCreateInfos.push_back(queueCreateInfo);
  }

  VkPhysicalDeviceFeatures deviceFeatures = {};
  deviceFeatures.samplerAnisotropy = VK_TRUE;

  VkDeviceCreateInfo createInfo = {};
  createInfo.sType = VK_STRUCTURE_TYPE_DEVICE_CREATE_INFO;

  createInfo.queueCreateInfoCount = static_cast<uint32_t>(queueCreateInfos.size());
  createInfo.pQueueCreateInfos = queueCreateInfos.data();

  createInfo.pEnabledFeatures = &deviceFeatures;
  createInfo.enabledExtensionCount = static_cast<uint32_t>(deviceExtensions.size());
  createInfo.ppEnabledExtensionNames = deviceExtensions.data();

  // might not really be necessary anymore because device specific validation layers
  // have been deprecated
  if (enableValidationLayers) {
    createInfo.enabledLayerCount = static_cast<uint32_t>(validationLayers.size());
    createInfo.ppEnabledLayerNames = validationLayers.data();
  } else {
    createInfo.enabledLayerCount = 0;
  }

  if (vkCreateDevice(physicalDevice, &createInfo, nullptr, &device_) != VK_SUCCESS) {
    throw std::runtime_error("failed to create logical device!");
  }

  vkGetDeviceQueue(device_, indices.graphicsFamily, 0, &graphicsQueue_);
  vkGetDeviceQueue(device_, indices.presentFamily, 0, &presentQueue_);
}

```

Figure 14: Create a logical device

Device can be thought of as a logical device, or opened device. It is the main object that represents an initialized Vulkan device that is ready to create all other objects. This is a similar concept to the Device object in DirectX®. During device creation, you need to specify which features you want to enable, and some of them are fundamental like anisotropic texture filtering. You also must state all queues that will be in use, their number and their Queue Families.

Queue is an object representing a queue of commands to be executed on the device. All the actual work to be done by the GPU is requested by filling CommandBuffers and submitting them to Queues, using the function `vkQueueSubmit`. If you have multiple queues like the main graphics queue and a compute queue, you can submit different CommandBuffers to each of them. This way you can enable asynchronous compute, which can lead to a substantial speed up if done right.

Figure 15: Explanation on Vulkan Device and Queues. [3]

4.6 Create command pool

Create a command pool, it is used to allocate command buffer to a queue family, deals with the memory allocation and hardware.

CommandPool is a simple object that is used to allocate CommandBuffers. It's connected to a specific Queue Family.

Figure 16: Command Pool. [3]

```cpp
void LveDevice::createCommandPool() {
  QueueFamilyIndices queueFamilyIndices = findPhysicalQueueFamilies();

  VkCommandPoolCreateInfo poolInfo = {};
  poolInfo.sType = VK_STRUCTURE_TYPE_COMMAND_POOL_CREATE_INFO;
  poolInfo.queueFamilyIndex = queueFamilyIndices.graphicsFamily;
  poolInfo.flags =
      VK_COMMAND_POOL_CREATE_TRANSIENT_BIT | VK_COMMAND_POOL_CREATE_RESET_COMMAND_BUFFER_BIT;

  if (vkCreateCommandPool(device_, &poolInfo, nullptr, &commandPool) != VK_SUCCESS) {
    throw std::runtime_error("failed to create command pool!");
  }
}
```

Figure 17: Command pool initialisation.

4.7 Buffer Helper function

The device object also has a helper function that is used later in the Model.cpp class to help create a memory allocation (and Vk memory object) for the vertex buffer.

```cpp
void LveDevice::createBuffer( // Helper function used in Model.cpp , it's a out function , you pass buffer and bufferMemory in and they will be changed
    // When we will integrate a memory allocator we will need to rewrtie this function
    VkDeviceSize size,
    VkBufferUsageFlags usage,
    VkMemoryPropertyFlags properties,
    VkBuffer &buffer,
    VkDeviceMemory &bufferMemory) {
  VkBufferCreateInfo bufferInfo{};
  bufferInfo.sType = VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO;
  bufferInfo.size = size;
  bufferInfo.usage = usage;
  bufferInfo.sharingMode = VK_SHARING_MODE_EXCLUSIVE;

  if (vkCreateBuffer(device_, &bufferInfo, nullptr, &buffer) != VK_SUCCESS) { // Create/ fill in the  the vkbuffer object
    throw std::runtime_error("failed to create vertex buffer!");
  }

  VkMemoryRequirements memRequirements;  // Create memory of proper size
  vkGetBufferMemoryRequirements(device_, buffer, &memRequirements);

  VkMemoryAllocateInfo allocInfo{}; // with all its appropriate properties
  allocInfo.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
  allocInfo.allocationSize = memRequirements.size;
  allocInfo.memoryTypeIndex = findMemoryType(memRequirements.memoryTypeBits, properties);

  if (vkAllocateMemory(device_, &allocInfo, nullptr, &bufferMemory) != VK_SUCCESS) { // Create/fill in  the Vkmemory object
    throw std::runtime_error("failed to allocate vertex buffer memory!");
  }

  vkBindBufferMemory(device_, buffer, bufferMemory, 0);// Bind the buffer to the memory we just allocated
}
```

Figure 18: Buffer memory allocation helper function.

5 Lve-Swapchain

From the surface created in section 4, we can now create a SwapChain. See the definition onn figure 20.

The main role of that class is as follow :

- Creating a SwapChain object (from a device object).
- Handles Synchronisation and setup for double or triple frame buffering based on device capabilities.
- Creating Framebuffer objects and their attachement (Color and Z value). This object is used by the graphic pipeline while rendering.
- Choose the present mode [5].
- Create Synchronisation objects like fences and semaphores.
- Create RenderPass , it represents the collection of attachments, in other words it contains the information on how the frame buffer is structured.
- Create ImageViews , a set of parameters referring to a specific image (an image being a set of pixel data).

[5] For more information on those check the official documentation `https://www.khronos.org/registry/vulkan/specs/1.3-extensions/man/html/VkPresentModeKHR.html`

```cpp
#pragma once

#include "lve_device.h"

// vulkan headers
#include <vulkan/vulkan.h>

// std lib headers
#include <string>
#include <vector>

namespace lve {

    //This deals handles the synchronisation and setup for double and triple framebuffers based on device  , also create framebufffer objects and their colorbuffer and depth attachment for our
//graphic pipeline to use while rendering

// Will come back to it for advacned subject like offscreen frame buffer for shadows rednindering


class LveSwapChain {
 public:
  static constexpr int MAX_FRAMES_IN_FLIGHT = 2; // WE CAN AT MOST HAVE TWO COMMAND BUFFER IN THE PENDING STATE

  LveSwapChain(LveDevice &deviceRef, VkExtent2D windowExtent);
  ~LveSwapChain();

  LveSwapChain(const LveSwapChain &) = delete;
  void operator=(const LveSwapChain &) = delete;

  VkFramebuffer getFrameBuffer(int index) { return swapChainFramebuffers[index]; }
  VkRenderPass getRenderPass() { return renderPass; }
  VkImageView getImageView(int index) { return swapChainImageViews[index]; }
  size_t imageCount() { return swapChainImages.size(); }
  VkFormat getSwapChainImageFormat() { return swapChainImageFormat; }
  VkExtent2D getSwapChainExtent() { return swapChainExtent; }
  uint32_t width() { return swapChainExtent.width; }
  uint32_t height() { return swapChainExtent.height; }

  float extentAspectRatio() {
    return static_cast<float>(swapChainExtent.width) / static_cast<float>(swapChainExtent.height);
  }
  VkFormat findDepthFormat();

  VkResult acquireNextImage(uint32_t *imageIndex);
  VkResult submitCommandBuffers(const VkCommandBuffer *buffers, uint32_t *imageIndex);

 private:
  void createSwapChain();
  void createImageViews();
  void createDepthResources();
  void createRenderPass();
  void createFramebuffers();
  void createSyncObjects();

  // Helper functions
  VkSurfaceFormatKHR chooseSwapSurfaceFormat(
      const std::vector<VkSurfaceFormatKHR> &availableFormats);
  VkPresentModeKHR chooseSwapPresentMode(
      const std::vector<VkPresentModeKHR> &availablePresentModes);
  VkExtent2D chooseSwapExtent(const VkSurfaceCapabilitiesKHR &capabilities);

  VkFormat swapChainImageFormat;
  VkExtent2D swapChainExtent;

  std::vector<VkFramebuffer> swapChainFramebuffers; // that will be the swapchain we create
  VkRenderPass renderPass;

  std::vector<VkImage> depthImages; // this will be the image depth
  std::vector<VkDeviceMemory> depthImageMemorys;
  std::vector<VkImageView> depthImageViews;
  std::vector<VkImage> swapChainImages; // color buffer
  std::vector<VkImageView> swapChainImageViews;

  LveDevice &device;
  VkExtent2D windowExtent;

  VkSwapchainKHR swapChain;

  std::vector<VkSemaphore> imageAvailableSemaphores;
  std::vector<VkSemaphore> renderFinishedSemaphores;
  std::vector<VkFence> inFlightFences;
  std::vector<VkFence> imagesInFlight;
  size_t currentFrame = 0;
};

}  // namespace lve
```

Figure 19: SwapChain header file.

One exception to the obligation to allocate and bind DeviceMemory for every Image is the creation of a Swapchain. This is a concept used to present the final image on the screen or inside the window you're drawing into on your operating system. As such, the way of creating it is platform dependent. If you already have a window initialized using a system API, you first need to create a **SurfaceKHR** object. It needs the Instance object, as well as some system-dependent parameters. For example, on Windows these are: instance handle (`HINSTANCE`) and window handle (`HWND`). You can imagine SurfaceKHR object as the Vulkan representation of a window.

From it you can create **SwapchainKHR.** This object requires a Device. It represents a set of images that can be presented on the Surface, e.g. using double- or triple-buffering. From the swapchain you can query it for the Images it contains. These images already have their backing memory allocated by the system.

Figure 20: SwapChain definition. [3]

```
LveSwapChain::LveSwapChain(LveDevice &deviceRef, VkExtent2D extent)
    : device{deviceRef}, windowExtent{extent} {
  createSwapChain();
  createImageViews();
  createRenderPass();
  createDepthResources();
  createFramebuffers();
  createSyncObjects();
}
```

Figure 21: Lve-SwapChain Constructor. It follows the same steps as described above.

5.1 Creating the SwapChain Object

When creating the Vulkan SwapChain object, a few present mode can be chosen (FIFO, V-Sync etc...).[6]

[6] For more information on those check the official documentation https://www.khronos.org/registry/vulkan/specs/1.3-extensions/man/html/VkPresentModeKHR.html

```cpp
void LveSwapChain::createSwapChain() {
  SwapChainSupportDetails swapChainSupport = device.getSwapChainSupport();

  VkSurfaceFormatKHR surfaceFormat = chooseSwapSurfaceFormat(swapChainSupport.formats);
  VkPresentModeKHR presentMode = chooseSwapPresentMode(swapChainSupport.presentModes); // depending on our graphic device it is settupt to choose the mailbox present mode per default
  //and Vsync as fallback
  //  present mopde define how our swap chain handle  synchroniation  with the display
  // by default it will be vsyn " FIFO" see explaination  https://www.youtube.com/watch?v=IUYH74MqxOA https://vulkan-tutorial.com/Drawing_a_triangle/Presentation/Swap_chain
  VkExtent2D extent = chooseSwapExtent(swapChainSupport.capabilities);

  uint32_t imageCount = swapChainSupport.capabilities.minImageCount + 1;
  if (swapChainSupport.capabilities.maxImageCount > 0 &&
      imageCount > swapChainSupport.capabilities.maxImageCount) {
    imageCount = swapChainSupport.capabilities.maxImageCount;
  }

  VkSwapchainCreateInfoKHR createInfo = {};
  createInfo.sType = VK_STRUCTURE_TYPE_SWAPCHAIN_CREATE_INFO_KHR;
  createInfo.surface = device.surface();

  createInfo.minImageCount = imageCount;
  createInfo.imageFormat = surfaceFormat.format;
  createInfo.imageColorSpace = surfaceFormat.colorSpace;
  createInfo.imageExtent = extent;
  createInfo.imageArrayLayers = 1;
  createInfo.imageUsage = VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT;

  QueueFamilyIndices indices = device.findPhysicalQueueFamilies();
  uint32_t queueFamilyIndices[] = {indices.graphicsFamily, indices.presentFamily};

  if (indices.graphicsFamily != indices.presentFamily) {
    createInfo.imageSharingMode = VK_SHARING_MODE_CONCURRENT;
    createInfo.queueFamilyIndexCount = 2;
    createInfo.pQueueFamilyIndices = queueFamilyIndices;
  } else {
    createInfo.imageSharingMode = VK_SHARING_MODE_EXCLUSIVE;
    createInfo.queueFamilyIndexCount = 0;      // Optional
    createInfo.pQueueFamilyIndices = nullptr;  // Optional
  }

  createInfo.preTransform = swapChainSupport.capabilities.currentTransform;
  createInfo.compositeAlpha = VK_COMPOSITE_ALPHA_OPAQUE_BIT_KHR;

  createInfo.presentMode = presentMode;
  createInfo.clipped = VK_TRUE;

  createInfo.oldSwapchain = VK_NULL_HANDLE;

  if (vkCreateSwapchainKHR(device.device(), &createInfo, nullptr, &swapChain) != VK_SUCCESS) {
    throw std::runtime_error("failed to create swap chain!");
  }

  // we only specified a minimum number of images in the swap chain, so the implementation is
  // allowed to create a swap chain with more. That's why we'll first query the final number of
  // images with vkGetSwapchainImagesKHR, then resize the container and finally call it again to
  // retrieve the handles.
  vkGetSwapchainImagesKHR(device.device(), swapChain, &imageCount, nullptr);
  swapChainImages.resize(imageCount);
  vkGetSwapchainImagesKHR(device.device(), swapChain, &imageCount, swapChainImages.data());

  swapChainImageFormat = surfaceFormat.format;
  swapChainExtent = extent;
}
```

Figure 22: createSwapChain() function. Creates the Vulkan SwapChain object.

```cpp
VkPresentModeKHR LveSwapChain::chooseSwapPresentMode(
    const std::vector<VkPresentModeKHR> &availablePresentModes) {
  for (const auto &availablePresentMode : availablePresentModes) {
    if (availablePresentMode == VK_PRESENT_MODE_MAILBOX_KHR) { // depending on our graphic device it is settupt to choose the mailbox present mode per default and Vsync as fallback
      std::cout << "Present mode: Mailbox" << std::endl; //  present mopde define how our swap chain handle  synchroniation  with the display
      return availablePresentMode; // by default it will be vsyn " FIFO" see explaination  https://www.youtube.com/watch?v=IUYH74MqxOA https://vulkan-tutorial.com/Drawing_a_triangle/P
    }
  }
   // to Alwasy use V-sync comment out those lines
  // for (const auto &availablePresentMode : availablePresentModes) {
  //   if (availablePresentMode == VK_PRESENT_MODE_IMMEDIATE_KHR) {
  //     std::cout << "Present mode: Immediate" << std::endl;
  //     return availablePresentMode;
  //   }
  // }

  std::cout << "Present mode: V-Sync" << std::endl;
  return VK_PRESENT_MODE_FIFO_KHR;
}
```

Figure 23: PresentMode selection. By Default , the "mailbox" option will be selected as it is most likely to work with most devices.

5.2 Create ImageViews

According to GpuOpen, [3] ImageViews "is a set of parameters referring to a specific image. There you can interpret pixels as having some other (compatible) format, swizzle any components, and limit the view to a specific range of MIP levels or array layers."

```
void LveSwapChain::createImageViews() {
  swapChainImageViews.resize(swapChainImages.size());
  for (size_t i = 0; i < swapChainImages.size(); i++) {
    VkImageViewCreateInfo viewInfo{};
    viewInfo.sType = VK_STRUCTURE_TYPE_IMAGE_VIEW_CREATE_INFO;
    viewInfo.image = swapChainImages[i];
    viewInfo.viewType = VK_IMAGE_VIEW_TYPE_2D;
    viewInfo.format = swapChainImageFormat;
    viewInfo.subresourceRange.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
    viewInfo.subresourceRange.baseMipLevel = 0;
    viewInfo.subresourceRange.levelCount = 1;
    viewInfo.subresourceRange.baseArrayLayer = 0;
    viewInfo.subresourceRange.layerCount = 1;

    if (vkCreateImageView(device.device(), &viewInfo, nullptr, &swapChainImageViews[i]) !=
        VK_SUCCESS) {
      throw std::runtime_error("failed to create texture image view!");
    }
  }
}
```

Figure 24: Creation of ImageViews.

5.3 Create a render pass

Here the render pass is being initialised.

In other graphics APIs you can take the immediate mode approach and just render whatever comes next on your list. This is not possible in Vulkan. Instead, you need to plan the rendering of your frame in advance and organize it into passes and subpasses. Subpasses are not separate objects, so we won't talk about them here, but they're an important part of the rendering system in Vulkan. Fortunately, you don't need to know all the details when preparing your workload. For example, you can specify the number of triangles to render on submission. The crucial part when defining a RenderPass in Vulkan is the number and formats of attachments that will be used in that pass.

Figure 25: Why a render pass. [3]

```cpp
void LveSwapChain::createRenderPass() {
  VkAttachmentDescription depthAttachment{};
  depthAttachment.format = findDepthFormat();
  depthAttachment.samples = VK_SAMPLE_COUNT_1_BIT;
  depthAttachment.loadOp = VK_ATTACHMENT_LOAD_OP_CLEAR;
  depthAttachment.storeOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
  depthAttachment.stencilLoadOp = VK_ATTACHMENT_LOAD_OP_DONT_CARE;
  depthAttachment.stencilStoreOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
  depthAttachment.initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
  depthAttachment.finalLayout = VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL;

  VkAttachmentReference depthAttachmentRef{};
  depthAttachmentRef.attachment = 1;
  depthAttachmentRef.layout = VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL;

  VkAttachmentDescription colorAttachment = {};
  colorAttachment.format = getSwapChainImageFormat();
  colorAttachment.samples = VK_SAMPLE_COUNT_1_BIT;
  colorAttachment.loadOp = VK_ATTACHMENT_LOAD_OP_CLEAR;
  colorAttachment.storeOp = VK_ATTACHMENT_STORE_OP_STORE;
  colorAttachment.stencilStoreOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
  colorAttachment.stencilLoadOp = VK_ATTACHMENT_LOAD_OP_DONT_CARE;
  colorAttachment.initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
  colorAttachment.finalLayout = VK_IMAGE_LAYOUT_PRESENT_SRC_KHR;

  VkAttachmentReference colorAttachmentRef = {};
  colorAttachmentRef.attachment = 0;
  colorAttachmentRef.layout = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL;

  VkSubpassDescription subpass = {};
  subpass.pipelineBindPoint = VK_PIPELINE_BIND_POINT_GRAPHICS;
  subpass.colorAttachmentCount = 1;
  subpass.pColorAttachments = &colorAttachmentRef;
  subpass.pDepthStencilAttachment = &depthAttachmentRef;

  VkSubpassDependency dependency = {};
  dependency.srcSubpass = VK_SUBPASS_EXTERNAL;
  dependency.srcAccessMask = 0;
  dependency.srcStageMask =
      VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT | VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT;
  dependency.dstSubpass = 0;
  dependency.dstStageMask =
      VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT | VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT;
  dependency.dstAccessMask =
      VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT | VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT;

  std::array<VkAttachmentDescription, 2> attachments = {colorAttachment, depthAttachment};
  VkRenderPassCreateInfo renderPassInfo = {};
  renderPassInfo.sType = VK_STRUCTURE_TYPE_RENDER_PASS_CREATE_INFO;
  renderPassInfo.attachmentCount = static_cast<uint32_t>(attachments.size());
  renderPassInfo.pAttachments = attachments.data();
  renderPassInfo.subpassCount = 1;
  renderPassInfo.pSubpasses = &subpass;
  renderPassInfo.dependencyCount = 1;
  renderPassInfo.pDependencies = &dependency;

  if (vkCreateRenderPass(device.device(), &renderPassInfo, nullptr, &renderPass) != VK_SUCCESS) {
    throw std::runtime_error("failed to create render pass!");
  }
}
```

Figure 26: Creation of renderpass.

5.4 Create depth resources

In this code, the depth attribute of the image is being defined. Figure 27.

```
void LveSwapChain::createDepthResources() {
  VkFormat depthFormat = findDepthFormat();
  VkExtent2D swapChainExtent = getSwapChainExtent();

  depthImages.resize(imageCount());
  depthImageMemorys.resize(imageCount());
  depthImageViews.resize(imageCount());

  for (int i = 0; i < depthImages.size(); i++) {
    VkImageCreateInfo imageInfo{};
    imageInfo.sType = VK_STRUCTURE_TYPE_IMAGE_CREATE_INFO;
    imageInfo.imageType = VK_IMAGE_TYPE_2D;
    imageInfo.extent.width = swapChainExtent.width;
    imageInfo.extent.height = swapChainExtent.height;
    imageInfo.extent.depth = 1;
    imageInfo.mipLevels = 1;
    imageInfo.arrayLayers = 1;
    imageInfo.format = depthFormat;
    imageInfo.tiling = VK_IMAGE_TILING_OPTIMAL;
    imageInfo.initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
    imageInfo.usage = VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT;
    imageInfo.samples = VK_SAMPLE_COUNT_1_BIT;
    imageInfo.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
    imageInfo.flags = 0;

    device.createImageWithInfo(
        imageInfo,
        VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT,
        depthImages[i],
        depthImageMemorys[i]);

    VkImageViewCreateInfo viewInfo{};
    viewInfo.sType = VK_STRUCTURE_TYPE_IMAGE_VIEW_CREATE_INFO;
    viewInfo.image = depthImages[i];
    viewInfo.viewType = VK_IMAGE_VIEW_TYPE_2D;
    viewInfo.format = depthFormat;
    viewInfo.subresourceRange.aspectMask = VK_IMAGE_ASPECT_DEPTH_BIT;
    viewInfo.subresourceRange.baseMipLevel = 0;
    viewInfo.subresourceRange.levelCount = 1;
    viewInfo.subresourceRange.baseArrayLayer = 0;
    viewInfo.subresourceRange.layerCount = 1;

    if (vkCreateImageView(device.device(), &viewInfo, nullptr, &depthImageViews[i]) != VK_SUCCESS) {
      throw std::runtime_error("failed to create texture image view!");
    }
  }
}
```

Figure 27: Creation of depth ressources.

5.5 Create Frame Buffer

Here again we can get a clearer explanation on what is the frame buffer from GpuOpen [3] on figure 28.

Framebuffer (not to be confused with SwapchainKHR) represents a link to actual Images that can be used as attachments (render targets). You create a Framebuffer object by specifying the RenderPass and a set of ImageViews. Of course, their number and formats must match the specification of the RenderPass. Framebuffer is another layer on top of Images and basically groups these ImageViews together to be bound as attachments during rendering of a specific RenderPass. Whenever you begin rendering of a RenderPass, you call the function `vkCmdBeginRenderPass` and you also pass the Framebuffer to it.

Figure 28: Definition of Frame buffer. [3]

```cpp
void LveSwapChain::createFramebuffers() {
  swapChainFramebuffers.resize(imageCount());
  for (size_t i = 0; i < imageCount(); i++) {
    std::array<VkImageView, 2> attachments = {swapChainImageViews[i], depthImageViews[i]};

    VkExtent2D swapChainExtent = getSwapChainExtent();
    VkFramebufferCreateInfo framebufferInfo = {};
    framebufferInfo.sType = VK_STRUCTURE_TYPE_FRAMEBUFFER_CREATE_INFO;
    framebufferInfo.renderPass = renderPass;
    framebufferInfo.attachmentCount = static_cast<uint32_t>(attachments.size());
    framebufferInfo.pAttachments = attachments.data();
    framebufferInfo.width = swapChainExtent.width;
    framebufferInfo.height = swapChainExtent.height;
    framebufferInfo.layers = 1;

    if (vkCreateFramebuffer(
            device.device(),
            &framebufferInfo,
            nullptr,
            &swapChainFramebuffers[i]) != VK_SUCCESS) {
      throw std::runtime_error("failed to create framebuffer!");
    }
  }
}
```

Figure 29: Creation of frame buffer.

5.6 Create Sync Objects

The sync object definition is provided on figure 31.

A **Semaphore** is created without configuration parameters. It can be used to control resource access across multiple queues. It can be signaled or waited on as part of command buffer submission, also with a call to `vkQueueSubmit` , and it can be signaled on one queue (e.g. compute) and waited on other (e.g. graphics).

An **Event** is also created without parameters. It can be waited on or signaled on the GPU as a separate command submitted to CommandBuffer, using the functions `vkCmdSetEvent` , `vkCmdResetEvent` , and `vkCmdWaitEvents` . It can also be set, reset and waited upon (via polling calls to `vkGetEventStatus` from one or more CPU threads. `vkCmdPipelineBarrier` can also be used for a similar purpose if synchronization occurs at a single point on the GPU, or subpass dependencies can be used within a render pass.

Figure 30: Definition of semaphores and fences in vulkan. [3]

```cpp
void LveSwapChain::createSyncObjects() {
  imageAvailableSemaphores.resize(MAX_FRAMES_IN_FLIGHT);
  renderFinishedSemaphores.resize(MAX_FRAMES_IN_FLIGHT);
  inFlightFences.resize(MAX_FRAMES_IN_FLIGHT);
  imagesInFlight.resize(imageCount(), VK_NULL_HANDLE);

  VkSemaphoreCreateInfo semaphoreInfo = {};
  semaphoreInfo.sType = VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO;

  VkFenceCreateInfo fenceInfo = {};
  fenceInfo.sType = VK_STRUCTURE_TYPE_FENCE_CREATE_INFO;
  fenceInfo.flags = VK_FENCE_CREATE_SIGNALED_BIT;

  for (size_t i = 0; i < MAX_FRAMES_IN_FLIGHT; i++) {
    if (vkCreateSemaphore(device.device(), &semaphoreInfo, nullptr, &imageAvailableSemaphores[i]) !=
            VK_SUCCESS ||
        vkCreateSemaphore(device.device(), &semaphoreInfo, nullptr, &renderFinishedSemaphores[i]) !=
            VK_SUCCESS ||
        vkCreateFence(device.device(), &fenceInfo, nullptr, &inFlightFences[i]) != VK_SUCCESS) {
      throw std::runtime_error("failed to create synchronization objects for a frame!");
    }
  }
}
```

Figure 31: Implementation of semaphores and fences

6 Lve-Pipeline

The main role of the Lve-Pipeline class is as follow:

- Define a structure outisde of the class, called PipelineConfigInfo containing all the info required to setup a Pipeline
- Has a static function defaultPipelineConfigInfo that fills up a default PipelineConfigInfo stuct passed as argument.
- Create a shader Module, object that represents a pice of shader code, possibly in some compiled form.
- Lve-Pipeline has a function to bind commandbuffers to it.
- When the constructor is called, it calls upon CreateGraphicPipelinefunction that take as argument the PipelineConfigInfo , shaderModule , shaders stages, device etc... and bind them together into one call of the Vulkan function vkCreateGraphicPipeline. This Actually create the Vulkan graphic pipeline object.

You can see on figure 32 , the header file for Lve-pipeline.

```cpp
#pragma once

#include <string>
#include <vector>
#include "lve_device.h"

namespace lve {


        // lets create a structur that will hold our data used to configure our pipeline, it's a helper structure
    struct PipelineConfigInfo
    { // we will explicitely set the different value of each stage of our pipeline and how it will work
        // this will be initialised in the  defaultPipelineConfigInfo function !
        //REALLY INTERESTING AND IMPORANT TO UNDERSTAND THOSE
        PipelineConfigInfo(const PipelineConfigInfo&) = delete;  // delete copy constructor
        PipelineConfigInfo& operator=(const PipelineConfigInfo&) = delete;
        PipelineConfigInfo() = default;// default constructor
        VkPipelineViewportStateCreateInfo viewportInfo;
        VkViewport viewport;
        VkRect2D scissor;
        //VkPipelineViewportStateCreateInfo viewportInfo; Fixing mistake
        VkPipelineInputAssemblyStateCreateInfo inputAssemblyInfo;
        VkPipelineRasterizationStateCreateInfo rasterizationInfo;
        VkPipelineMultisampleStateCreateInfo multisampleInfo;
        VkPipelineColorBlendAttachmentState colorBlendAttachment;
        VkPipelineColorBlendStateCreateInfo colorBlendInfo;
        VkPipelineDepthStencilStateCreateInfo depthStencilInfo;
        VkPipelineLayout pipelineLayout = nullptr; // we dont provide any default fir this members and we will set them outside of defaultPipelineConfigInfo
        VkRenderPass renderPass = nullptr;// we dont provide any default fir this members and we will set them outside of defaultPipelineConfigInfo
        uint32_t subpass = 0;// we dont provide any default fir this members and we will set them outside of defaultPipelineConfigInfo



    };
    // The reason it's out of pipeline class is because we want our application layer to be easily able to reach and configure a pipeline compeltely as well as to sahre
    //the configuration between mutliple pilenine



    class LvePipeline {
    public:
        LvePipeline(LveDevice& device,const std::string& vertFilepath, const std::string& fragFilepath , const PipelineConfigInfo& configInfo);//constructor

        ~LvePipeline();// destructor



        LvePipeline(const LvePipeline&) = delete; // delete the default copy constructor https://www.youtube.com/watch?v=BvR1Pgzzr38&t=590s
        // we want to avoid  duplicating the pointer to our vulkan object by mistake
        void operator=(const LvePipeline&) = delete;



        void bind(VkCommandBuffer commandBuffer);// bind graphic pipeline to a commandbuffer

        static void defaultPipelineConfigInfo(PipelineConfigInfo& configInfo, uint32_t width, uint32_t height);// static function to create a default  PipelineConficInfo structure

    private:
        static std::vector<char> readFile(const std::string& filepath); // returns a string of char , takes in a ref of a string
        void createGraphicPipeline(const std::string& vertFilepath, const std::string& fragFilepath, const PipelineConfigInfo& configInfo);

        void createShaderModule(const std::vector<char>& code, VkShaderModule* shaderModule); //takes as input the shader code as a vector of carachter , shaderModule is a pointer to a pointer !

        LveDevice& lveDevice; // private member variable storinng the device reference,  could be memory unsafe if  our device is realesed from memory before the pipeline ,
        //we could try to dereference a dangling pointer, would crash the program.
        //This is a reference type member varaible , it should implicitely outlive the class it lives on it .
        VkPipeline graphicsPipeline; // handle to vulkan pipeline object , it's a pointer !
        VkShaderModule vertShaderModule; // shader module variable for vert shader
        VkShaderModule fragShaderModule;// shader module variable for frag shader
    };

}
```

Figure 32: Lve-Pipeline.h

To see the definition of the Vulkan pipeline object, check figure 33.

Pipeline is the big one, as it composes most of the objects listed before. It represents the configuration of the whole pipeline and has a lot of parameters. One of them is PipelineLayout – it defines the layout of descriptors and push constants. There are two types of Pipelines – ComputePipeline and GraphicsPipeline. ComputePipeline is the simpler one, because all it supports is compute-only programs (sometimes called compute shaders). GraphicsPipeline is much more complex, because it encompasses all the parameters like vertex, fragment, geometry, compute and tessellation where applicable, plus things like vertex attributes, primitive topology, backface culling, and blending mode, to name just a few. All those parameters that used to be separate settings in much older graphics APIs (DirectX 9, OpenGL), were later grouped into a smaller number of state objects as the APIs progressed (DirectX 10 and 11) and must now be baked into a single big, immutable object with today's modern APIs like Vulkan. For each different set of parameters needed during rendering you must create a new Pipeline. You can then set it as the current active Pipeline in a CommandBuffer by calling the function `vkCmdBindPipeline`.

Figure 33: Definition of the Vulkan pipeline object. [3]

6.1 Config Info Structure

That configInfoStruct is visible in the pipeline header on figure 32. This struct is use to hold the parameters of the soon to be created pipeline. Notice how the struct is defined outside of the Pipeline class.

6.2 Static Function defaultPipelineConfigInfo

This static function takes as input a configInfoStruct of section 6.1 and fills it up with default values. This function is extremely long and extensively commented within the code, as such I decided not to showcase it here. Nevertheless it is very much worth taking the time to analyze is as it contains many parameters regarding rasterisation, anti aliasing, tessellation etc...

6.3 Create a shaderModule object

Create a shader Module, object that represents a piece of shader code, in SPV compiled format.

```
void LvePipeline::createShaderModule(const std::vector<char>& code, VkShaderModule* shaderModule) { // CREATE A SHADER MODULE
    VkShaderModuleCreateInfo createInfo{}; //we create  local Vk shader module
    //common pattern instead of calling a function with  a bunch of paramter , we conmfigure a strcut and call a function  with a pointer to the struct
    createInfo.sType = VK_STRUCTURE_TYPE_SHADER_MODULE_CREATE_INFO; // the type of that structure is a VK structure shader module create info
    createInfo.codeSize = code.size();// the code size is the size of the array
    createInfo.pCode = reinterpret_cast<const uint32_t*>(code.data()); // a uint32 and charcther are not the same size so we gotta be carful with it
    //,We need to cast the code from being char to being uint32 as vulkan expect it to be uint32
    // becasue our data is stored in a vector , this cast will work , however C style characther array that wouldnt be a valid cast .


    if (vkCreateShaderModule(lveDevice.device(), &createInfo, nullptr, shaderModule) != VK_SUCCESS) // LveDevice.device() is a getter to the device handle  that our device classs encapsulate
    { // pointer to createinfo , nullpointer is function callbacks
        throw std::runtime_error("failed to create shader module");//runtime error
    }

}
```

Figure 34: ShaderModule creation

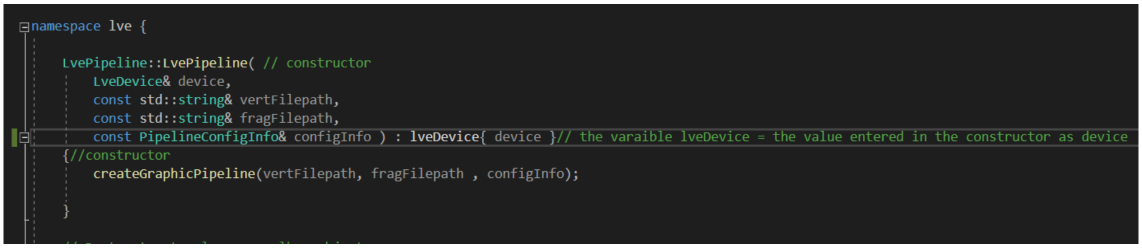
6.4 Bind Function

This function is important as it allows to bind a commandBuffer to the pipeline.

6.5 Object Creation and Constructor call

The Lve-Pipeline constructor takes as input argument the device, the vertex and fragment shader, and the configInfo struct described in section6.1.That config info struct should have been initialised and given default values before being passed here. From there the constructor calls upon the createGraphicPipeline function and bind them together into one call of the Vulkan function vkCreateGraphicPipeline. This Actually create the Vulkan graphic pipeline object.

This createGraphicPipeline function,as it takes as argument the file path to the vertex and fragment shader, calls upon the createShaderModule function of section 6.3.



```cpp
namespace lve {

    LvePipeline::LvePipeline( // constructor
        LveDevice& device,
        const std::string& vertFilepath,
        const std::string& fragFilepath,
        const PipelineConfigInfo& configInfo ) : lveDevice{ device }// the varaible lveDevice = the value entered in the constructor as device
    {//constructor
        createGraphicPipeline(vertFilepath, fragFilepath , configInfo);

    }
```

Figure 35: The constructor for Lve-Pipeline. Notice the arguments

Here Again, the function **createGraphicPipeline** is really long and extensively commented, it would require to talk a few pages and I therefore chose not showcase it this document. Although it is very important to read it carefully and understand it. Notice on figure 32 and 35 the function definition.

7 First-app

This class, is the "orchestrator " of the code, it is the one that will call most of the other constructor.The only function above is the main function that is simply used to run First-app (for now).

The main tasks for this class are the following :

- The first function call when calling the constructor is to load the models.It is there for testing purposes.
- Upon calling the constructor, first-app calls createpipelineLayout() a function that creates a pipelineLayout, a Vulkan struct called VkPipelineLayoutCreateInfo containing info on the layout of the pipeline. This could be well be done within the Pipeline class but for ease of access it has been taken out in the first-app.
- Next, createPipeline() is called , creating a pipelineConfigInfo struct from section 6.1, this is possible as this struct is defined outside of the Lve-Pipeline class, initialising it thanks to the static function of Lve-Pipeline described on section 6.2. Within this function the constructor of Lve-Pipeline is called, thus a Vulkan pipeline is created. See the pipeline section.
- The final step of the constructor is to create a commandbuffer (as well as a renderpass contained in that command buffer, this render pass has information on how the frame buffer is expected to be structured and other likes the clearvalues) and link it to the pipeline. Using the member function of lve-pipeline "bind".To create a command buffer we also create a command pool that will manage and handle all the memory allocation and hardware.
- First-app also has a run member function that calls drawframe that fetch the index of the next frame and submits the commandbuffer.Command buffer is the only way to send instructions / command calls to the GPU.The command buffer is an object containing a series of command that will be sent to the device's queue.

```cpp
FirstApp::FirstApp()
{
    loadModels();
    createPipelineLayout();
    createPipeline();
    createCommandBuffers();
}


FirstApp::~FirstApp()
{
    vkDestroyPipelineLayout(lveDevice.device(), pipelineLayout, nullptr);
}


```

Figure 36: The constructor of First-app.

```cpp
#include "lve_window.h"
#include "lve_pipeline.h"
#include "lve_device.h"
#include "lve_swap_chain.h"

// include for memory
//STD
#include <memory> // to access smart pointers
#include <vector>
namespace lve {

    class FirstApp {
    public:
        static constexpr int WIDTH = 800;
        static constexpr int HEIGHT = 600;
        FirstApp(); // constructor
        ~FirstApp(); // destructor
        //We will delete the copy construcor and copy destroctor from our window class , bc we use a pointer to the glfw window
        FirstApp(const FirstApp&) = delete; //https://www.youtube.com/watch?v=BvR1Pgzzr38
        FirstApp& operator = (const FirstApp&) = delete;
        // see shallow and deep copies ,

        void run() ;

    private:

        void loadModels();
        void createPipelineLayout();
        void createPipeline();
        void createCommandBuffers(); // in vulkan , no directly command with function calls , we use a command buffer that stores instrcution and goes in a device queue to be executed , this allow for sequences of command to be recorded
        void drawFrame();

        LveWindow lveWindow{ WIDTH, HEIGHT,"Hello Vulkan!" }; //https://www.youtube.com/watch?v=1nfuYMXjZsA   , https://www.youtube.com/watch?v=FXhALMsHwEY&list=PLlrATfBNZ98dudnM48yfGUldqGD0S4FFb&index=25
        // not using a pointer or any dynamic memory allocation
        //// here he put {} instead of () , this is a way to initainalise value with the constructor ! calling a csontrcutor from a header file
        // is this due to C++ 17 ?  https://en.cppreference.com/w/cpp/language/constructor
        //
        LveDevice lveDevice{ lveWindow };
        LveSwapChain lveSwapChain{ lveDevice , lveWindow.getExtent() }; // HERE ORDER MATTERS ! varaibles intialised top to bottom

        std::unique_ptr<LvePipeline> lvePipeline; // Smart pointer  https://www.youtube.com/watch?v=UOB7-B2MfwA , automamte the new , delete thing to allocate memory on the stack.
        // Unique poitner is a scoped poitner , when we going out of scope this pointer is deleted , you cant copy a unique pointer , stack allocator object
        VkPipelineLayout pipelineLayout;
        std::vector<VkCommandBuffer> commandBuffers;
        std::unique_ptr<LveModel> lveModel; // creating a smart pointer , allocate on the heap  , memory gets freed automatically by the smart pointer,
    };
```

Figure 37: Header file of first-app

7.1 LoadModels

This function simply loads 3 vertices for testing purposes.

7.2 Create Pipeline Layout

Upon calling the constructor, first-app calls createpipelineLayout() a function that creates a pipelineLayout, a Vulkan struct called VkPipelineLayoutCreateInfo containing info on the layout of the pipeline. This could be well be done within the Pipeline class but for ease of access it has been taken out in the first-a

```cpp
void FirstApp::createPipelineLayout()
{ // Empty layout
    VkPipelineLayoutCreateInfo pipelineLayoutInfo{};
    pipelineLayoutInfo.sType = VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;
    pipelineLayoutInfo.setLayoutCount = 0;
    pipelineLayoutInfo.pSetLayouts = nullptr; // Pipelinesetlayout is used to pass data other than vertex data to our vertex and fragemetn shaders, like texture and uniforms buffer objects
    pipelineLayoutInfo.pushConstantRangeCount = 0;
    pipelineLayoutInfo.pPushConstantRanges = nullptr; // a way to very efficeiently send a small data to our shader program ( see tutorial 8)
    if (vkCreatePipelineLayout(lveDevice.device(), &pipelineLayoutInfo, nullptr, &pipelineLayout) != VK_SUCCESS) {
        throw std::runtime_error("failed to create pipeline layout!");
    }
}
```

Figure 38: createPipelineLayout() function, it is used to create a pipeline layout struct that contains info on the layout of the soon to be initialised pipeline.

7.3 Create Pipeline

```
void FirstApp::createPipeline()
{
    PipelineConfigInfo pipelineConfig{};
    LvePipeline::defaultPipelineConfigInfo(
        pipelineConfig,
        lveSwapChain.width(),
        lveSwapChain.height()); // important to get the width and heigh of the swapchain as it doesnt necessary match the window  , for exemple apple retina display of apple , the window in screen coordinate is smaller than actual resolution

    pipelineConfig.renderPass = lveSwapChain.getRenderPass();// default render pass creatd from the swapchain code , render pass describe structure and format of our frame buffer object and its attachement , exemple attachement 0 is color buffer
    pipelineConfig.pipelineLayout = pipelineLayout;
    lvePipeline = std::make_unique<LvePipeline>( // Smartpointer  CALL THE CONSTRUCTOR OF lve_pipeline , the constructor launches a function that create the Graphic pipeline.
        lveDevice,
        "shaders/simple_shader.vert.spv",
        "shaders/simple_shader.frag.spv",
        pipelineConfig
        );
}
```

Figure 39: when createPipeline() is called , it is creating a pipelineConfigInfo struct from section 6.1, this is possible as this struct is defined outside of the Lve-Pipeline class, initialising it thanks to the static function of Lve-Pipeline described on section 6.2. Within this function the constructor of Lve-Pipeline is called, thus a Vulkan pipeline is created. See the pipeline section.

7.4 Create Command Buffer

This function is used create a commandbuffer (as well as a renderpass contained in that command buffer, this render pass has information on how the frame buffer is expected to be structured and other likes the clearvalues) and link it to the pipeline. Using the member function of lve-pipeline "bind".To create a command buffer we also create a command pool that will manage and handle all the memory allocation and hardware.

This part of the code is pretty lengthy, and once again really well commented, i decided once again not to include it here as it would require several pages and screenshots.

7.5 Run function

This function is pretty self explanatory, once called by main, it will call drawframe and display the result of our code.

```
void FirstApp::run()
{

    while (!lveWindow.shouldClose()) // see if the windows wants to closeor not
    {
        // while we do not want to close
        glfwPollEvents();  // check if event like  key stroke  user click to dismiss the window
        drawFrame();
    }

    vkDeviceWaitIdle(lveDevice.device()); // with this the CPU will block up until the GPU is done , so that when
    //we close the window , it's not in the middle of GPU calculation

}
```

Figure 40: The run function, pretty straight forward and self explanatory.

```cpp
void FirstApp::drawFrame()
{
    uint32_t imageIndex;
    auto result = lveSwapChain.acquireNextImage(&imageIndex); // fetch the index of the frame that is rendered next and automatically handles the CPU /GPU synchronisation  for double or triple buffering
    if (result != VK_SUCCESS && result != VK_SUBOPTIMAL_KHR)
    {
        throw std::runtime_error("failed to acquire swap chain image  !"); // in the futur we will need to handle this situation as this happens when a window is resized
    }

    result = lveSwapChain.submitCommandBuffers(&commandBuffers[imageIndex], &imageIndex); // pointer at the commandbuffer at image index and pointer of image index , this fucntion submits the command buffer to the device graphic queue
    if (result != VK_SUCCESS )
    {
        throw std::runtime_error("failed to acquire swap chain image  !"); // in the futur we will need to handle this situation as this happens when a window is resized
    }

}


}
```

Figure 41: The draw frame function, it fetches the index of the next frame and submits the command-buffer.Command buffers are the only way to send instructions / command calls to the GPU.The command buffer is an object containing a series of command that will be sent to the device's queue.

8 Lve-model

The goal of Lve-model is to create and link a vertex buffer, so that vertices would not need to be hard coded within the buffer.

- Upon construction, it launches the createVertexBuffers() CreateVertexBuffers() makes the mapping and synchronisation between CPU and GPU (coherent memory).It calls upon the createbuffer helper function in lve-device.
- The model class has a GetBindingDescription() , this binding description object is necessary when binding a buffer to a pipeline, it describes how the data is structured.
- The model Class has a Getattributedescription() , the vertex attribute description simply describes how to link the memory with the shader code.
- It has a Bind command to bind it to a commandpool
- It has a draw command to do the draw call for that object.

```cpp
#include <vector>

namespace lve
{
    class LveModel
    {
        //The purpose of this class it to take vertex data created by and read from a file from the CPU
        //and then allocate the memory on the GPU and copy the data over there for efficicent reading

    public:

        struct Vertex
        {
            glm::vec2 posistion;
            static std::vector<VkVertexInputBindingDescription> getBindingDescriptions();
            static std::vector<VkVertexInputAttributeDescription> getAttributeDescriptions();
        };

        LveModel(LveDevice& device, const std::vector<Vertex>& vertices); // constructor
        ~LveModel(); // destructor


        //We will delete the copy construcor and copy destructor from our LveModelclass , as it deals with memory of vulkan , if we
        //make copies it might corrupt the memory
        LveModel(const LveModel&) = delete; //https://www.youtube.com/watch?v=BvR1Pgzzr38
        LveModel& operator = (const LveModel&) = delete;
        // see shallow and deep copies ,

        void bind(VkCommandBuffer commandBuffer);
        void draw(VkCommandBuffer commandBuffer);


    private:

        void createVertexBuffers(const std::vector<Vertex>& vertices);


        LveDevice& lveDevice;
        VkBuffer vertexBuffer; // The buffer
        VkDeviceMemory vertexBufferMemory; // The assgigned memory of the buffer , 2 separate objects !
        // The memory is not automatically asssigned , so us , the programmer is in controll of memory management
        uint32_t vertexCount;

    }; // namespace lve


}
```

Figure 42: Header of Lve-Model, notice the vertex structure containing the the vertices.

```cpp
// constructor
LveModel::LveModel(LveDevice& device, const std::vector<Vertex>& vertices) : lveDevice{device}
{
    createVertexBuffers(vertices);
}


// destructor
LveModel::~LveModel()
{
    vkDestroyBuffer(lveDevice.device(), vertexBuffer, nullptr);
    vkFreeMemory(lveDevice.device(), vertexBufferMemory, nullptr);
    //allocation of memory
    // The reason why the allcoation of buffer and memory is separated is taht allcoating memory takes time
    // Also there is a hard limit on the total number of active allocation at once depending on the GPU
    //The maximum number of valid memory allocations that can exist simultaneously within a VkDevice may be restricted by implementation- or platform-dependent limits.
    // https://www.khronos.org/registry/vulkan/specs/1.2-extensions/man/html/vkAllocateMemory.html
    // Beter then to allcoate bigger chunks of memory and assign pieces for different models
    //Best practice is to use the Vulkan Memory Allocator libray ! VMA https://gpuopen.com/vulkan-memory-allocator/
}   // How to make a memory allocator by ourselve http://kylehalladay.com/blog/tutorial/2017/12/13/Custom-Allocators-Vulkan.html
```

Figure 43: Constructor and destructor of the lveModel object. Notice the comment , allocating memory takes time and there is a hard limit on the total number of active allocation at once depending on the GPU, it is then better to allocate bigger chunks of memory and assign pieces for different models. A good way to do that is to use the VMA , Vulkan Memory Allocator from AMD. [3] It is also possible to make it from scratch. [4]

8.1 Create Vertex Buffers

CreateVertexBuffers() makes the mapping and synchronisation between CPU and GPU (coherent memory).It calls upon the createbuffer helper function in lve-device.

```cpp
void LveModel::createVertexBuffers(const std::vector<Vertex>& vertices)
{
    vertexCount = static_cast<uint32_t>(vertices.size()); // cast the number of vertices
    //let's assert that we have minimum of 3 vertices to have minimum a triangle , use cassert
    assert(vertexCount >= 3 && "Vertex count must be at least 3");
    VkDeviceSize bufferSize = sizeof(vertices[0]) * vertexCount; // sizeof gives us the byte number for one vertex , then mutliply by the number of vertices we have in total and we get the buffer size
    //Next call the create buffer function , it's a helper function in the lveDeviceclass
    // it's a out function , you pass buffer and bufferMemory in and they will be changed

    lveDevice.createBuffer(
        bufferSize,
        VK_BUFFER_USAGE_VERTEX_BUFFER_BIT,// this tells our device that this buffer will be to hold vertex data
        VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT | VK_MEMORY_PROPERTY_HOST_COHERENT_BIT, // | is the bitwise or operator,combines the bit flags together, the host is the CPU , device is GPU ,
        //the host visible bit tels vulkan that we awant the allocated memory to be accessible from the CPU, the Coherant part is to keep the memory of GPU and CPU coherant, aka the same!
        // If this coherant property would be absent we would need to call VK flush to propagate any change from the device to the CPU
        vertexBuffer,
        vertexBufferMemory
    );
    void* data; // local varaible void pointer data
    vkMapMemory(lveDevice.device(), vertexBufferMemory, 0, bufferSize, 0, &data);// 0 is offset ,  other 0 is for not providing any vk flags, and then data
    // this function will set data to point to the memory location of the begining of the vertex buffer memory in the GPU ! So data on the Cpu sides holds that adress
    // //this function actually creates a mapping between the CPU and GPU memory
    //Then use memcopy function
    memcpy(data, vertices.data(), static_cast<size_t>(bufferSize)); // takes the vertices data and copy it in the host mapped memory region, becasue we have the host coherant bit VK_MEMORY_PROPERTY_HOST_COHERENT_BIT , the host memory will automatically
    //be flushed into the GPU to update its memory, if this VK_MEMORY_PROPERTY_HOST_COHERENT_BIT bit  wasn't there we would need to manually call flush function
    vkUnmapMemory(lveDevice.device(), vertexBufferMemory);


}
```

Figure 44: Create the mapping between the CPU memory and GPU. Care for the comments on this particular function. As we set the coherent bit, we automate the CPU and GPU memory to be coherent, at each CPU memory update it will update the GPU. Otherwise a manual Vulkan "flush" would be required.

8.2 Get Binding description

When binding a buffer to the pipeline, it is necessary to pass pass a vertex binding description struct at pipeline creation. This struct describe how the data in the buffer is structured (interleaved or not etc...)[7]. See figure 45.

[7]For more information on data structure within the buffer , interleaving or not, check this link : `https://anteru.net/blog/2016/storing-vertex-data-to-interleave-or-not-to-interleave/`

```cpp
std::vector<VkVertexInputBindingDescription> LveModel::Vertex::getBindingDescriptions()
{//See hand notes
    std::vector<VkVertexInputBindingDescription> bindingDescriptions(1);// local variable , vector of VkVertexInputBinding Description of size1
    bindingDescriptions[0].binding = 0;
    bindingDescriptions[0].stride = sizeof(Vertex); // Sizeof gives the memory spqce in bytes
    bindingDescriptions[0].inputRate = VK_VERTEX_INPUT_RATE_VERTEX; // other option is for instanced data
    return bindingDescriptions; // this binding descriptions corresponds to a single vertex buffer that will ocupy the first binding at index
    //the stride is the size of the vertex in bytes

}

```

Figure 45: Binding description structure

8.3 Get Attribute description

```cpp
std::vector<VkVertexInputAttributeDescription> LveModel::Vertex::getAttributeDescriptions()
{
    std::vector<VkVertexInputAttributeDescription> attributeDescriptions(1);
    attributeDescriptions[0].binding = 0; // This is  for the binding 0
    attributeDescriptions[0].location = 0; // that correspond to the location specific in the vertex shaders => layout(location=0) in vec2 position;
    attributeDescriptions[0].format = VK_FORMAT_R32G32_SFLOAT; // specify the data type , there are 2 component and each of them are 32 bit signed floats
    attributeDescriptions[0].offset = 0;
    return attributeDescriptions;
}

```

Figure 46: This struct describes how to link the memory to the shader code.

8.4 Command buffer binding and Draw

```cpp
void LveModel::bind(VkCommandBuffer commandBuffer)
{
    VkBuffer buffers[] = { vertexBuffer }; // create a local variable that is a list
    VkDeviceSize offsets[] = { 0 };
    vkCmdBindVertexBuffers(commandBuffer, 0, 1, buffers, offsets); // This will record to our command buffer to bind one vertrex buffer starting at binding 0 with offset of 0
    // when we will eventually add mutliple biding we can easily do so by by using the variable offset and buffers !
}


void LveModel::draw(VkCommandBuffer commandBuffer)
{
    vkCmdDraw(commandBuffer, vertexCount, 1, 0, 0);
}

```

Figure 47: Binding to the commandpool and draw call.

9 Current Output

Depending on the vertices created in First-app for testing purposes, we can now display some fun triangles. There is still much to do !

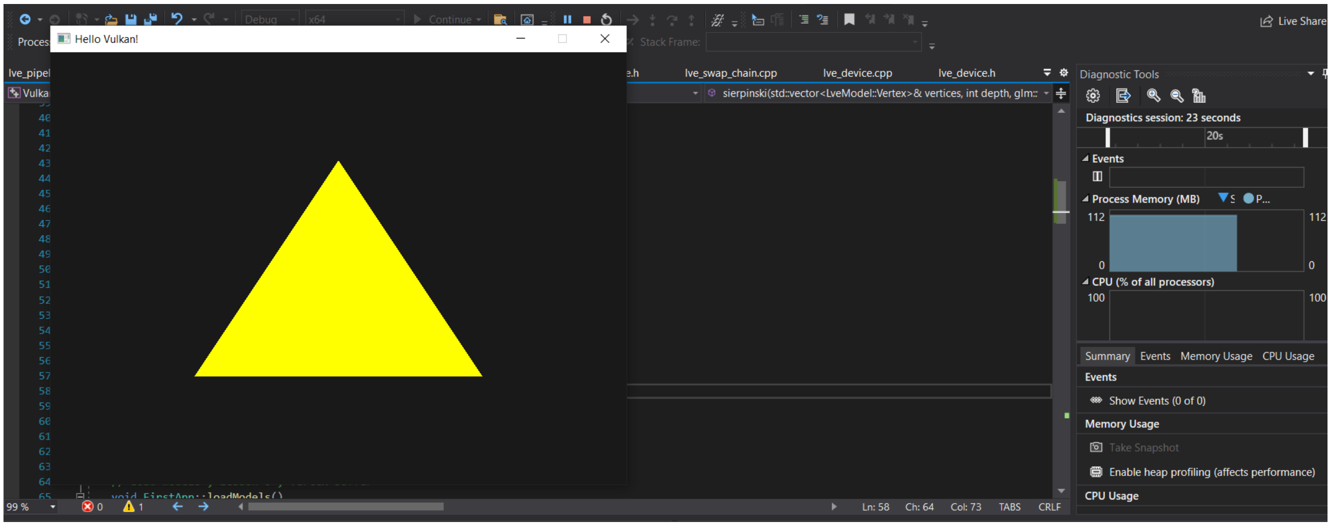


Figure 48: Simple Output with 3 vertices.

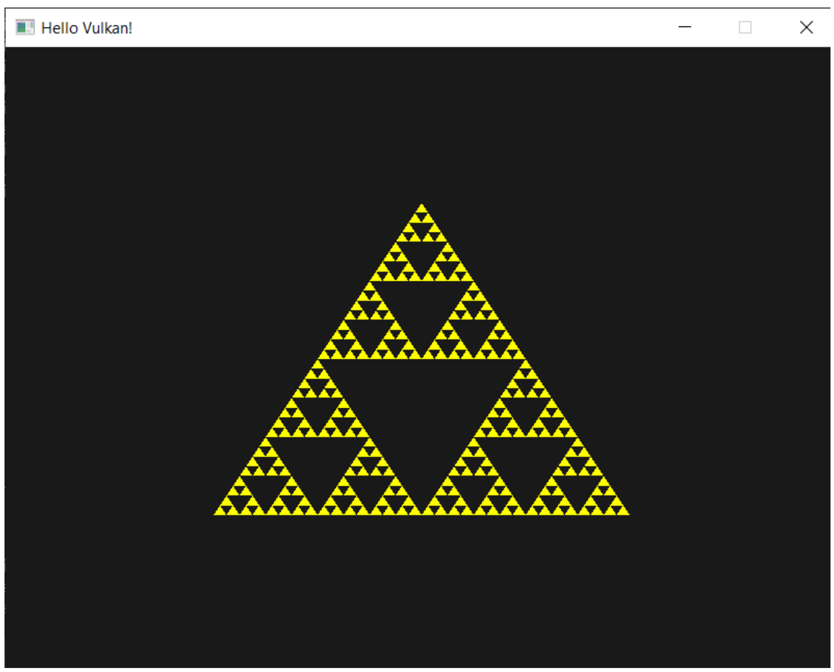


Figure 49: Sierpinski triangle solution. [1]

References

[1] Brendan Galea tutorial series on making a Vulkan Engine, visited on january 2022.
`https://www.youtube.com/c/BrendanGalea/videos`

[2] The official Vulkan Tutorial , visited on january 2022.
`https://vulkan-tutorial.com/`

[3] Understanding Vulkan Object , Adma Sawicki, visited on january 2022.
`https://gpuopen.com/learn/understanding-vulkan-objects/`

[4] Kyle Halladay, make a custom memory allocator in Vulkan.visited on january 2022.
`http://kylehalladay.com/blog/tutorial/2017/12/13/Custom-Allocators-Vulkan.html`

[5] Vulkan Memory Allocator , GpuOpen .visited on january 2022.
`https://gpuopen.com/vulkan-memory-allocator/`

[6] Storing vertex data: To interleave or not to interleave? visited on january 2022. `https://anteru.net/blog/2016/storing-vertex-data-to-interleave-or-not-to-interleave/`