Wide-Open Common API Guide

By: Mohammad Issawi

Disclaimer

Vulkan is owned by **The Khronos Group.**
Assimp is owned by its respectful owners.
GLFW is owned by its respectful owners.
I only created The API using the mentioned above projects

Table of Contents

1.0 Introduction

Wide-Open Common API is a C++ library that relies directly on Vulkan API, it contains a set of classes which contain essential elements for almost every Vulkan project (and especially for starters), Wide-Open Common API also contains the functions for most likely needed tasks for these elements,however,thus means more restriction over Vulkan API,which was made in the first place to be as abstracted as possible,moreover, Wide-Open Common API classes are mostly pure-virtual (or abstract as referred in other programming languages) we'll get more on that later,the main goal of the API is to accelerate setting up projects and make it easier to understand how to connect the elements when using pure Vulkan.
Note that if you didn't know about how to implement an abstract function, you can refer to other projects that uses the API, also you can always check the Vulkan spec on more details.

1.1 The API Concept

With extending and abstraction in mind,Wide-Open Common API (I'll refer to it from now on as "The API") relies mainly on the user to write his own classes extending the APIs' to meet his own needs, most classes in the API contain at least a pure-virtual function that needs overriding, overriding base classes is also possible but not recommended as you have to know what you're doing so you don't break the application.
Moreover,most classes are made with being used as singletons in mind, you'll see that they contain a protected empty constructor and an init function,however, when extending these classes you'll need to write the instance getter function yourself,while I think it's possible to avoid using singletons,but it's not recommend as this is the intended design.

2.0 The API Classes

2.1 The Window System Class

`Common::Window` is the class that handles the window-related tasks, unlike most API classes, `Common::Window` is a concrete class and doesn't not overriding by default, basically, it creates a plain window with no context,its underlying library is GLFW.

Functions:

- `void init(int height,int width)` : initializes a window with the given dimensions,using GLFW.
- `GLFWwindow* getWindow()` : returns a pointer to the window structure used,useful to pass to some GLFW functions.
- `int getHeight()` : returns the window's height.
- `int getWidth()` : returns the window's width.
- `void terminate()` : handles window system cleaning up.
- `Static Window& instance()` : returns the singleton's instance

2.2 Graphics System Classes

2.2.1 Renderer

`Common::Renderer` sets up the main components for the Vulkan API, it also contains the handles for many essential Vulkan structures,to be more specific, it creates a Vulkan instance, detects the physical device which to be used, creates the logical device, a surface which is GLFW's, a swap-chain, a command pool,a graphics queue,a presenting queue (sometimes they are the same,probably varies between GPUs), it also contains an Assimp importer object for importing meshes data.

Data members:

- `static Assimp::Importer importer` : This object is used to get data from mesh files, however,this is handled in the Mesh class so you don't need to use it directly, however it will be fine to use it to extend what data from a mesh is obtained, please refer to Assimp docs for more data if that's what you want to achieve.

Function:

- `virtual void createDescriptorPool()=0` : This function is meant to create a descriptor pool using `descriptorPool` data member,thus means when implementing it you MUST make sure `descriptorPool` is a valid pointer to a `VkDesecriptorPool` by the end of the function execution.
- `Virtual void allocateDescriptorSet(VkDescriptorSet* set)=0` : Since a descriptor set layout is an independent structure, this function has to be implemented to use a specific descriptor set layout to allocate a correspond descriptor set given the parameter.
- VkDeviceMemory allocateMemory(VkMemoryRequirements memReq,VkMemoryPropertyFlags properties) : this function allocates memory for the given parameters and returns a valid handle for binding, please refer to Vulkan spec for more on the parameters (seriously, it's not hard, look it up).
- `VkDevice getDevice()` : returns the handle to the logical device,MANY FUNCTIONS require the logical device as a parameter.
- `VkSurfaceFormatKHR getSwapchainFormat()` : This function returns the color space and the format of the swap-chain ,which are useful when creating an image for presenting into the screen.
- `VkExtent2D getExtent2D()` : returns the window/swap-chain/surface 2D dimensions in a VkExtent2D structure.

- `VkExtent3D getExtent3D()` : same as getExtent2D but with a third dimension set to 1 (some functions require a 3D extent so it's faster to call this function…)
- `uint32_t getGraphicsQueueFamilyIndex()` : returns the index to graphics queue family index.
- `VkImageView* getSwapchainImageViews()` : returns a pointer to an array of VkImageView, note that `swapchainImagesCount` is a protected data member that contains the number of swap-chain images.
- `VkCommandPool getCmdPool()` : returns the handle to the command pool used by the renderer.
- `VkQueue getGraphicsQueue()` : returns the graphics queue handle,which is required when submitting a command buffer.
- `VkSwapchainKHR getSwapchain()` : returns the handle to the swap-chain.
- `void init()` : initializes the renderer class as mentioned above in the class description.
- `void allocateDescriptorSet(VkDescriptorSet* set,VkDescriptorSetLayout layout)` : allocates a descriptor set using the corresponding descriptor set layout.
- `void terminate()` : cleans up the singleton resources, after calling this function, don't get any of it's data members as they might be deleted and would break the application, it's recommended to terminate the renderer as the last Graphics System singleton since most Vulkan resources rely on the logical device, we'll get more on that later.

2.2.2 Render Pass

`Common::RenderPass` handles everything related to frame buffers and sub passes, also it's the best class to write a command buffer commands (my recommendation is to add a function in the derived class that contains the drawing and binding commands and call it when ready,as used in Blinn-Phong With Shadows).
Please note that before initializing a render pass, it's pure-virtual functions MUST BE IMPLEMENTED CORRECTLY, otherwise it'll fail.

Frame buffer structure

`Common::RenderPass::Framebuffer` holds data for a frame buffer that can be used in a render pass and it MUST be filled when implementing `setupFramebuffers`

Framebuffer data members

- `VkFramebuffer framebuffer` : Vulkan handle to a framebuffer which should be sat up when implementing `setupFramebuffers`
- `VkImageView* imageViews` : a pointer to an array of image views for the frame buffer images, swap-chain image excluded.
- `VkImage* images` : a pointer to an array of images which are used in the framebuffer, swap-chain image excluded.
- `VkDeviceMemory* imagesMemory` : a pointer to an array of device memory which MUST be bound to the images during implementing.
- `VkImageView* swapchainImage` : a pointer to the swap chain image view if it's intended to be used in the frame buffer, in case you don't want it in the framebuffer process, set it to nullptr (it is its default value anyway).
- `uint32_t imagesCount` : the number of images used in the framebuffer minus the swap-chain image (even when it's used), also it is the same as `imageViews,images,imagesMemory` length.

Render Pass Functions

- `virtual void setupAttachments()=0` : this function MUST set `attachmentsCount,attachmentRefsCount` and allocate memory for `attachments,attachmentsRefs` arrays,and also set up each attachment description and attachment reference as needed.

- `Virtual void setupSubPasses()=0` : this function MUST set `subPassesCount,dependenciesCount` and allocate memory for `subPasses,dependencies` arrays, and also set up each sub pass and sub pass dependency as needed.
- `Virtual void setupFramebuffers()=0` : this function MUST set `framebuffersCount` and allocate memory for `framebuffers,` and also set up each frame buffer as needed.
- `Void init(Renderer* renderer)` : takes up a Renderer pointer which will be used by the render pass, initializes the render pass according the pure-virtual-implemented functions.
- `Void terminate()` : cleans up memory resources and destroy associated data.
- `VkCommandBuffer getCmdBuffer()` : returns a handle to the render pass command buffer.
- `VkRenderPass getRenderPass()` : returns a handle to the Vulkan structure render pass.
- `VkFramebuffer getFramebuffer(uint32_t index)` : returns the corresponding Vulkan structure frame buffer from `framebuffer` array.

2.2.3 Pipeline

`Common::Pipeline` contains A LOT of pure-virtual functions associated to create a pipeline without restricting, be careful when implementing these functions to avoid allocating a structure on stack memory (especially attributes and bindings description structures).

Functions

- `virtual void create*State()=0` : These set of functions all share the same goal; fill the `*StateCreateInfo` structure,and nothing other than filling,we don't create any actual objects here just yet,however, how we tailor our structures will be used in the pipeline on initializing.
- `Virtual void createShaderModules()=0` : This function MUST set `shaderModulesCount`,and allocate an array for the pointer `shaderModules`, and fill the shader modules structures too.
- `Virtual void createShaderStages()=0` : This function MUST set `shaderStagesCount,` and allocate an array for the pointer `shaderStageCreateInfo` , and fill each structure properly.
- `Virtual void createLayout()=0` : This function MUST create the Vulkan pipeline layout structure based on the descriptor set layout data member `dsl` in the Pipeline class.
- `VkShaderModule createShaderModule(const char* path)` : creates a shader module out of the SPIR-V shader given its path, this function is recommended to be used when implementing `createShaderModules` (Sorry about the names similarity).
- Void init(Renderer* renderer,DescriptorSetLayout* dsl,VkRenderPass renderPass,uint32_t subPassIndex) : This function creates the Vulkan pipeline object within the given parameters, the pipeline layout will match up with the given descriptor set layout, and the sub pass you want to use the pipeline with within that render pass.
- `VkPipeline getPipeline()` : returns the Vulkan handle to the pipeline,quite useful for binding pipeline calls.
- `VkPipelineLayout getLayout()` : returns the Vulkan handle to the pipeline layout.
- `Void terminate()` : cleans up the singleton used resources.

2.2.4 Descriptor Set Layout

This singleton will be used to serve as the descriptor set layout.

Functions

- `virtual void init(Renderer* renderer)=0` : This function MUST be implemented to create a descriptor set layout in the Vulkan handle `descriptorSetLayout`.
- `VkDescriptorSetLayout getDSL()` : This function returns the Vulkan handle to the descriptor set layout created in init function.
- `Void terminate()` : cleans up the class resources,obviously, the descriptor set layout.

2.2.5 Mesh

`Common::Mesh<V,U>` is a template class for the sake of variety on what you might include of a vertex and uniform data,so before you derive this class you must create two classes (One that marks what data each vertex would holds, and the other marks what a uniform buffer would holds), it's also fine to define more class members if you want to use more than a single descriptor set layout (another uniform buffer) with the same mesh other than creating another mesh with a different uniform buffer object.

Functions

- `virtual V* readVertices(aiMesh* mesh)=0` : This function MUST use an Assimp mesh structure to read the defined vertices data into the mesh object, it returns a pointer to an array of vertices of type V.
- virtual void applyUBO()=0 : This function MUST update the descriptor set this mesh uses which is specified on creation,in other words ,it tells the descriptor set what unfirommm buffers and images if present to use, it will be called on creation too.
- `Mesh(const char* path,U uniformBufferObject,Renderer* renderer)` : initializes the mesh object with the data in the path (it must be an Assimp supported format,such as FBX,GLTF…) it also fills the uniform buffer with the data from the corresponding parameter,Note that it (by default) uses the descriptor set layout used in `Renderer::allocateDescriptorSet` , but more than one descriptor set layout can be used, as explained further.
- `VkDescriptorSet getDescriptorSet(int index=0)` : returns the descriptor set corresponding to the given index.
- Void setDescriptorSetsCount(int index) : sets the number of descriptor sets that to be used in the mesh (default is 1),NOTE: if you want to use multiple descriptor sets you first must increase the sets count by 1, allocate the new descriptor set using Renderer::allocateDescriptorSet , after that you can obtain it using getDescriptorSet, you'll also need to write a function that applies the data described in that new descriptor set, in other words, you'll have to manually handle a new applyUBO function.
- `VkBuffer* getVertexBuffer()` : returns a pointer to the Vulkan handle of vertices buffer,useful when writing draw commands.
- `VkBuffer getIndexBuffer()` : returns the handle to Vulkan buffer that holds the data of the mesh indices, also useful when writing draw commands
- `uint32_t getIndicesCount()` : returns the number of indices that are store in the indices buffer,also useful when writing draw commands
- `void cleanup()` : while meshes are not singletons, cleaning them up manually is required since they use internal Vulkan resources that might be deleted before the scope of the mesh,that's why cleaning up them is REQUIRED before cleaning up other resources,I'll get more on that later.

2.3 Misc.

2.3.1 Log Class

`Common::Log` is a simple singleton that is used for debugging across the API,it simply writes the log to “debug.txt” file, it includes the type of the message,a timestamp,and the message content.

Functions

- `static Log& instance()` : returns the singleton instance.
- `Void log(const char* msg)` : writes the message as a LOG message to the log file.
- `Void log(int numbers[],int count)` : writes the numbers array as a LOG message to the log file.
- `Void error(const char* msg,bool isFatal=true)` : writes the message as an ERR message to the log file and throws an exception if `isFatal` is true,otherwise it’ll continue the execution with the message logged.
- `Void close()` : closes the log file.

2.3.2 Helper Functions

These functions live in `Common` name-space and are included when including `Helpers.h` header

- `VkPhysicalDevice getPhysicalDevice(VkInstance instance)` : returns a handle to the first discrete GPU found on the hardware,it is used internally in the API so you don’t need to worry about it,you can,however, change it if you are having a problem in detecting the GPU.
- `VkCommandPool* createCommandPool(VkDevice& device,uint32_t queueIndex)` : creates a command pool for the corresponding queue family index,again, it is used internally and it’s fine if you don’t use it or don’t know what it is used for.
- `Char* readBinFile(const char* path,uint32_t size)` : reads a binary file and return the pointer to the data in bytes,it also takes a `size` parameter which WILL have the number of read bytes after the execution.

2.3.3 Definitions

This file basically holds some `#define` statements to make some shortcuts when coding, especially when using singletons, note that `ALLOCATOR` is currently a `nullptr` since there is no allocator structure .

3.0 Recommended Application Model

This section will describe a good order to initialize your singletons since some of them relies on the other.
Please note that I used abstract classes names in the following example,in your code, they should be the derived classes (instance function must be implemented for all singletons in order for the application to work, if the base class constructor is protected).

```
Window::instance().init(480,640);
Renderer::instance().init();
RenderPass::instance().init(&Renderer::instance());
//other Render Passes could be initialized here too
DescriptorSetLayout::instance().init(&Renderer::instance());
//other DSLs could be initialized here too
Pipeline::instance().init(&Renderer::instance(),&DescriptorSetLayo
ut::instance(),RenderPass::instance().getRenderPass(),0);
//other pipelines could be initialized here too
UniformBufferObject ubo;
//ubo could be filled here
Mesh mesh("x.gltf",ubo);
//other meshes could be defined here
//recording and rendering loop could be written here
mesh.cleanup();
//other meshes could be cleaned up here
Pipeline::instance().terminate();
//other pipelines could be terminated here
DescriptorSetLayout::instance().terminate();
//other DSLs could be terminated here
RenderPass::instance().terminate();
//other render passes could be terminated here
Renderer::instance().terminate();
Window::instance().terminate();
```

4.0 A Final Word

I hope you find this API useful in your journey of learning graphics programming, I know it might be frustrating, but I hope you get the most of it, I might add new examples in the future,mostly based on the Common API,so if you're interested , stay tuned !
If you feel like supporting, my Patreon profile is available in the project GitHub page.
Made with love and fat kitties.