POLITECNICO
MILANO 1863

DIPARTIMENTO DI ELETTRONICA
INFORMAZIONE E BIOINGEGNERIA

DEIB

2024

Dipartimento di Elettronica, Informazione e Bioingegneria

Computer Graphics

Milano, 2024

Computer Graphics

- Vulkan

Marco Gribaudo - DEIB

Introduction to Vulkan Applications

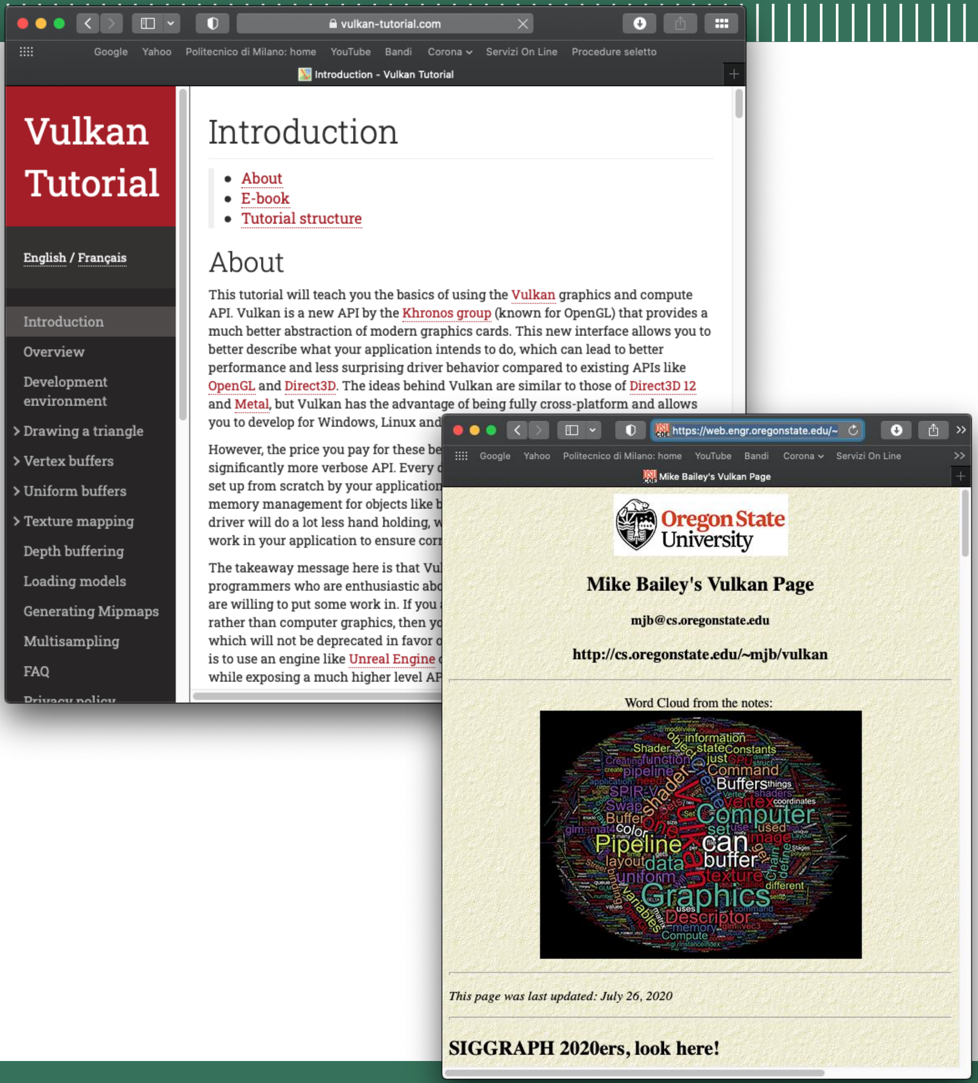
The main sources for learning Vulkan used in this course are the official *Vulkan Tutorial*:

https://vulkan-tutorial.com

And the 2020 SigGraph course:

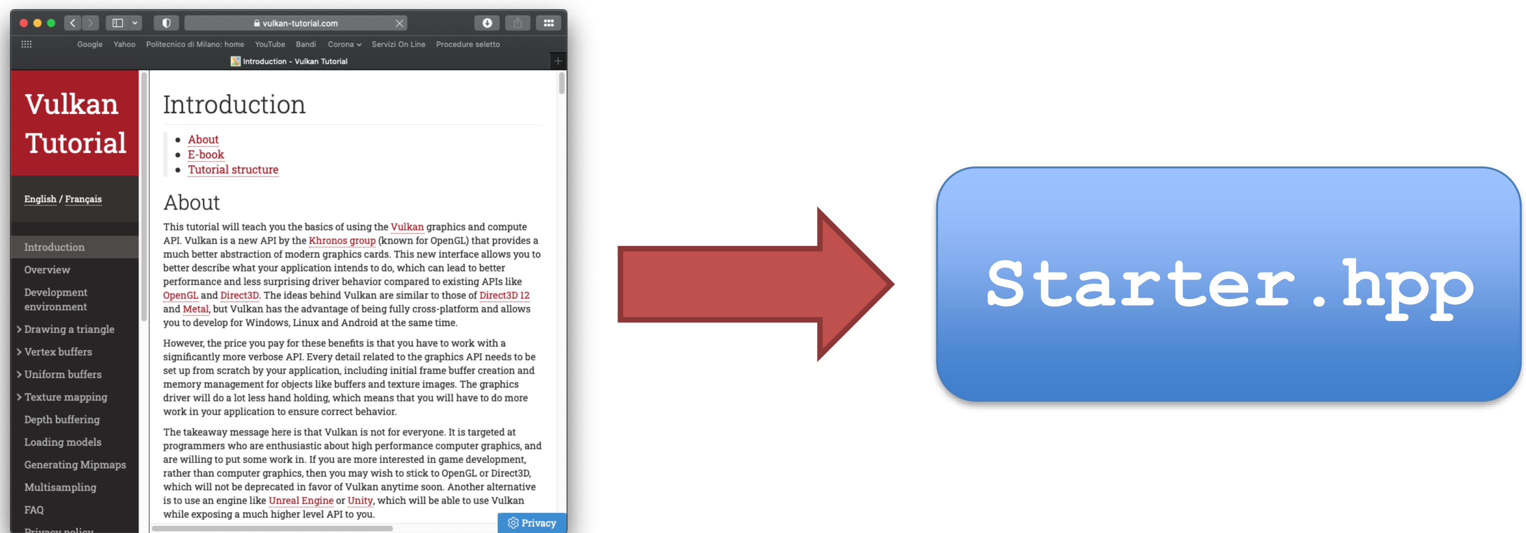
https://web.engr.oregonstate.edu/~mjb/vulkan/

Please have a look at them if you need further studying material.



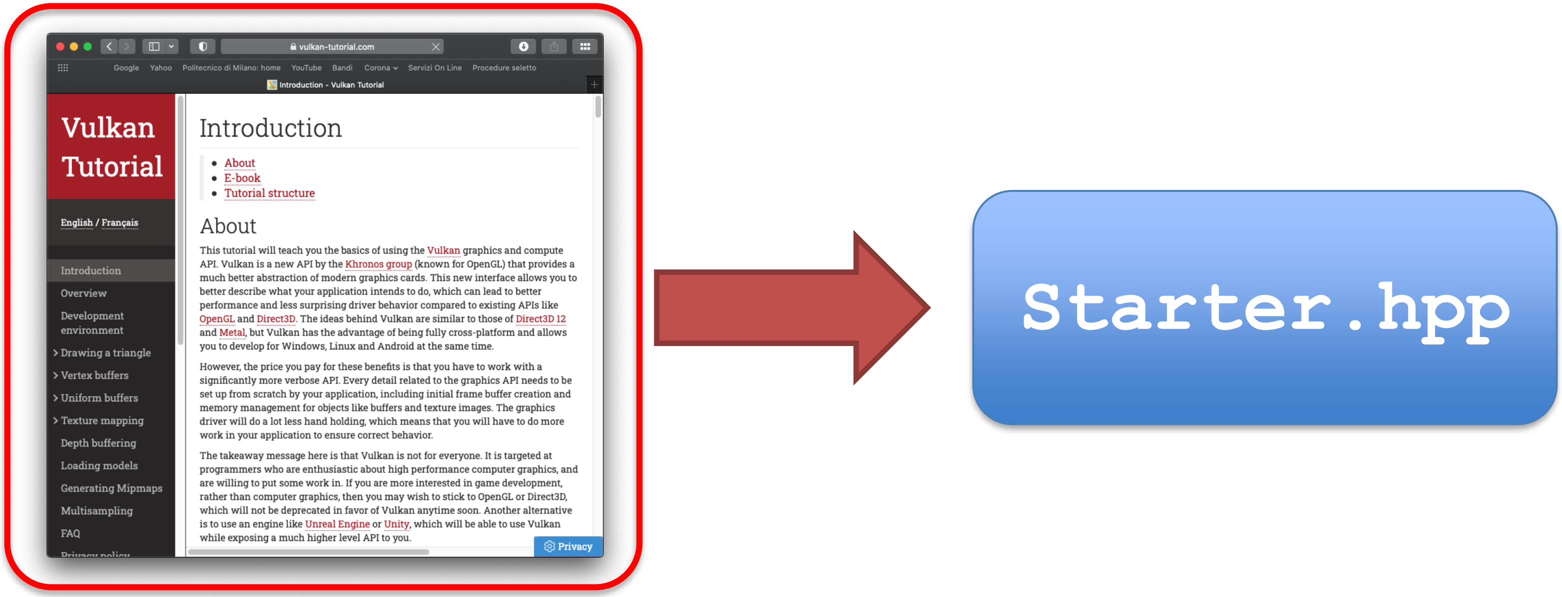
Introduction to Vulkan Applications

To simplify the interaction with Vulkan, I have developed over the years a library called "`Starter.hpp`", where I have wrapped the tutorial, and created a simpler to use interface. In this course, I will present in detail only the access to Vulkan using my library.



Introduction to Vulkan Applications

Students interested in learning Vulkan at a lower level, should follow the tutorial, and see how and where it is “inserted” inside our library.



Current computer architectures are characterized by:

- Several CPU cores
- One or more different GPUs
- Different memory types: CPU and GPUs memory
- Several concurrent applications or VMs needing to use the CPUs and the GPUs at the same time

Vulkan has been created to allow the users to exploit the available resources at their best.

This however has a big downside: an *enormous setup complexity*!

Vulkan supported systems

Vulkan can run on very different types of systems:

- Desktop computer (PC, Mac, Linux, …)
- Mobile (Smartphones, Tablets, VR HUDs, …)
- Console (Nintendo Switch, …)
- Embedded systems (Map display in a car, …)

Every system has its unique features: Vulkan aims at supporting them all!

Starter.hpp

A large number of steps are essential to exploit all the Vulkan features in an application. However, in most of the cases the user will relay on the same (solid) start-up sequence.

The file `modules/Starter.hpp` used in all the assignments, aims at defining a common initialization procedure, avoiding the user to explicitly repeat all the “normal startup steps” in her project.

During this brief presentation of Vulkan, we will rapidly explain where such steps occur in `modules/Starter.hpp`.

Skeleton of a Vulkan application

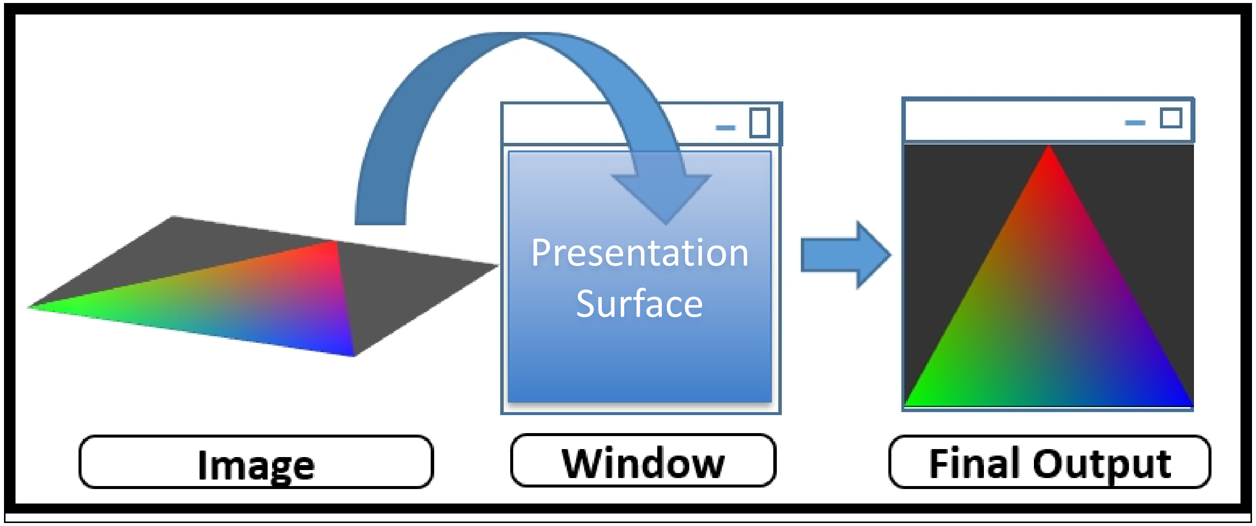
A typical Vulkan application has the following skeleton:

```
void run() {
    initWindow();       //create the O.S. window
    initVulkan();       //set up Vulkan resources
    initApp();          //loads and set up app. elements
    mainLoop();         //the update / render cycle of the app.
    cleanup();          //release all the resources
}
```

The Presentation Surface

The screen area where the host Operating System allows Vulkan to draw images is called the *Presentation Surface*.

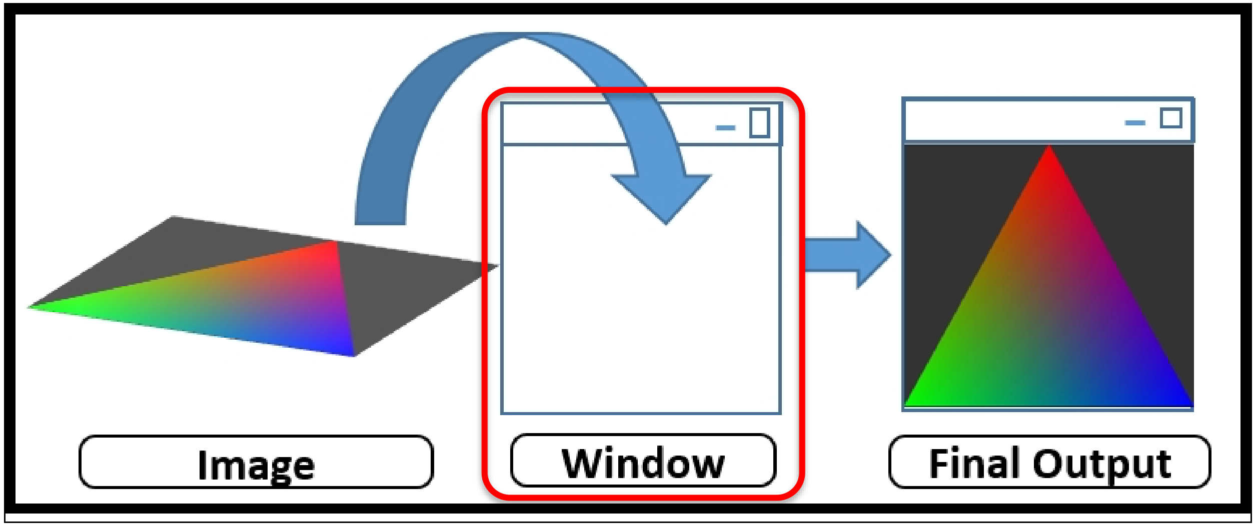
In order to work properly, a Vulkan application should acquire a proper presentation surface from the O.S. This step is system dependent, and we will return on this later.



The Application Window

In a desktop system, such as MS Windows, MacOS or Linux, the presentation surface will always be contained inside a Window.

In this course, we will only consider desktop applications.



GLFW window creation

The GLFW allows to open window in a host independent way. Before opening a window, GLFW should be initialized.

```
void initWindow() {
    glfwInit();

    glfwWindowHint(GLFW_CLIENT_API, GLFW_NO_API);

    window = glfwCreateWindow(WIDTH, HEIGHT, "Vulkan", nullptr, nullptr);


}
```

GLFW window creation

Several parameters can be used to define the characteristics of the window created. In GLFW this is done using the `glfwWindowHint(prop, val)` command, which assigns the value `val` to the considered property `prop`.

Since the default operating mode of GLFW is to supprot *OpenGL*, we must set the `GLFW_CLIENT_API` property to `GLFW_NO_API` to use Vulkan .

```
void initWindow() {
    glfwInit();

    glfwWindowHint(GLFW_CLIENT_API, GLFW_NO_API);

    window = glfwCreateWindow(WIDTH, HEIGHT, "Vulkan", nullptr, nullptr);


}
```

GLFW window creation

A large number of other options can be set: if interested, have a look at the GLFW documentation.

Window related hints

GLFW_RESIZABLE specifies whether the windowed mode window will be resizable *by the user*. The window will still be resizable using the **glfwSetWindowSize** function. Possible values are `GLFW_TRUE` and `GLFW_FALSE`. This hint is ignored for full screen and undecorated windows.

GLFW_VISIBLE specifies whether the windowed mode window will be initially visible. Possible values are `GLFW_TRUE` and `GLFW_FALSE`. This hint is ignored for full screen windows.

GLFW_DECORATED specifies whether the windowed mode window will have window decorations such as a border, a close widget, etc. An undecorated window will not be resizable by the user but will still allow the user to generate close events on some platforms. Possible values are `GLFW_TRUE` and `GLFW_FALSE`. This hint is ignored for full screen windows.

GLFW_FOCUSED specifies whether the windowed mode window will be given input focus when created. Possible values are `GLFW_TRUE` and `GLFW_FALSE`. This hint is ignored for full screen and initially hidden windows.

GLFW_AUTO_ICONIFY specifies whether the full screen window will automatically iconify and restore the previous video mode on input focus loss. Possible values are `GLFW_TRUE` and `GLFW_FALSE`. This hint is ignored for windowed mode windows.

GLFW_FLOATING specifies whether the windowed mode window will be floating above other regular windows, also called topmost or always-on-top. This is intended primarily for debugging purposes and cannot be used to implement proper full screen windows. Possible values are `GLFW_TRUE` and `GLFW_FALSE`. This hint is ignored for full screen windows.

GLFW_MAXIMIZED specifies whether the windowed mode window will be maximized when created. Possible values are `GLFW_TRUE` and `GLFW_FALSE`. This hint is ignored for full screen windows.

GLFW_CENTER_CURSOR specifies whether the cursor should be centered over newly created full screen windows. Possible values are `GLFW_TRUE` and `GLFW_FALSE`. This hint is ignored for windowed mode windows.

GLFW_TRANSPARENT_FRAMEBUFFER specifies whether the window framebuffer will be transparent. If enabled and supported by the system, the window framebuffer alpha channel will be used to combine the framebuffer with the background. This does not affect window decorations. Possible values are `GLFW_TRUE` and `GLFW_FALSE`.

GLFW_FOCUS_ON_SHOW specifies whether the window will be given input focus when **glfwShowWindow** is called. Possible values are `GLFW_TRUE` and `GLFW_FALSE`.

GLFW_SCALE_TO_MONITOR specified whether the window content area should be resized based on the **monitor content scale** of any monitor it is placed on. This includes the initial placement when the window is created. Possible values are `GLFW_TRUE` and `GLFW_FALSE`.

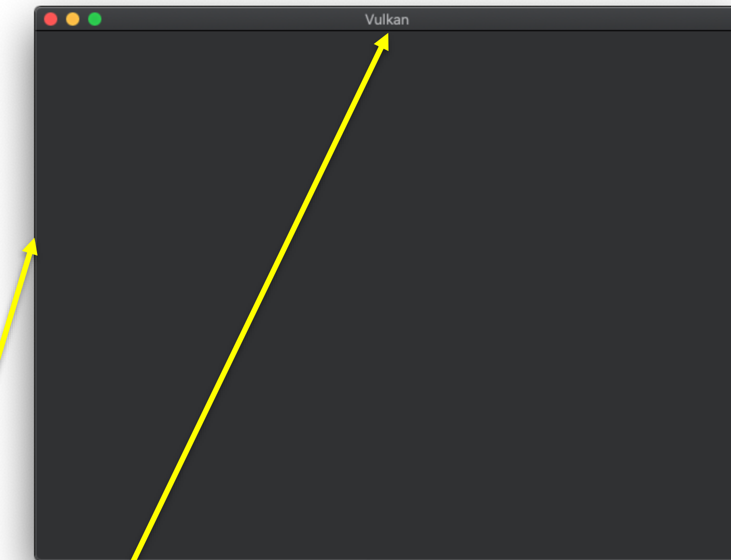
This hint only has an effect on platforms where screen coordinates and pixels always map 1:1 such as Windows and X11. On platforms like macOS the resolution of the framebuffer is changed independently of the window size.

Currently at
https://www.glfw.org/docs/3.3/window_guide.html#window_hints

GLFW window creation

Command `glfwCreateWindow(…)` creates the O.S. window, and return its identifier.

The procedure receives the horizontal and vertical size of the window (`WIDTH` and `HEIGHT`) in pixel, and the string to display in the title bar.




```
void initWindow() {
    glfwInit();

    glfwWindowHint(GLFW_CLIENT_API, GLFW_NO_API);

    window = glfwCreateWindow(WIDTH, HEIGHT, "Vulkan", nullptr, nullptr);

}
```

GLFW window creation

In our applications using `Starter.hpp`, this is done in a specific procedure callback procedure named `setWindowParameters()`.

```
// Here you set the main application parameters
void setWindowParameters() {
    // window size, titile and initial background
    windowWidth = 800;
    windowHeight = 600;
    windowTitle = "A09 - Smooth Mesh";
    windowResizable = GLFW_TRUE;
    initialBackgroundColor = {0.0f, 0.85f, 1.0f, 1.0f};

    // Descriptor pool sizes
    uniformBlocksInPool = 28 * 2 + 2;
    texturesInPool = 28 + 1;
    setsInPool = 28 + 1;

    Ar = 4.0f / 3.0f;
}
```

Example: `A09.cpp`

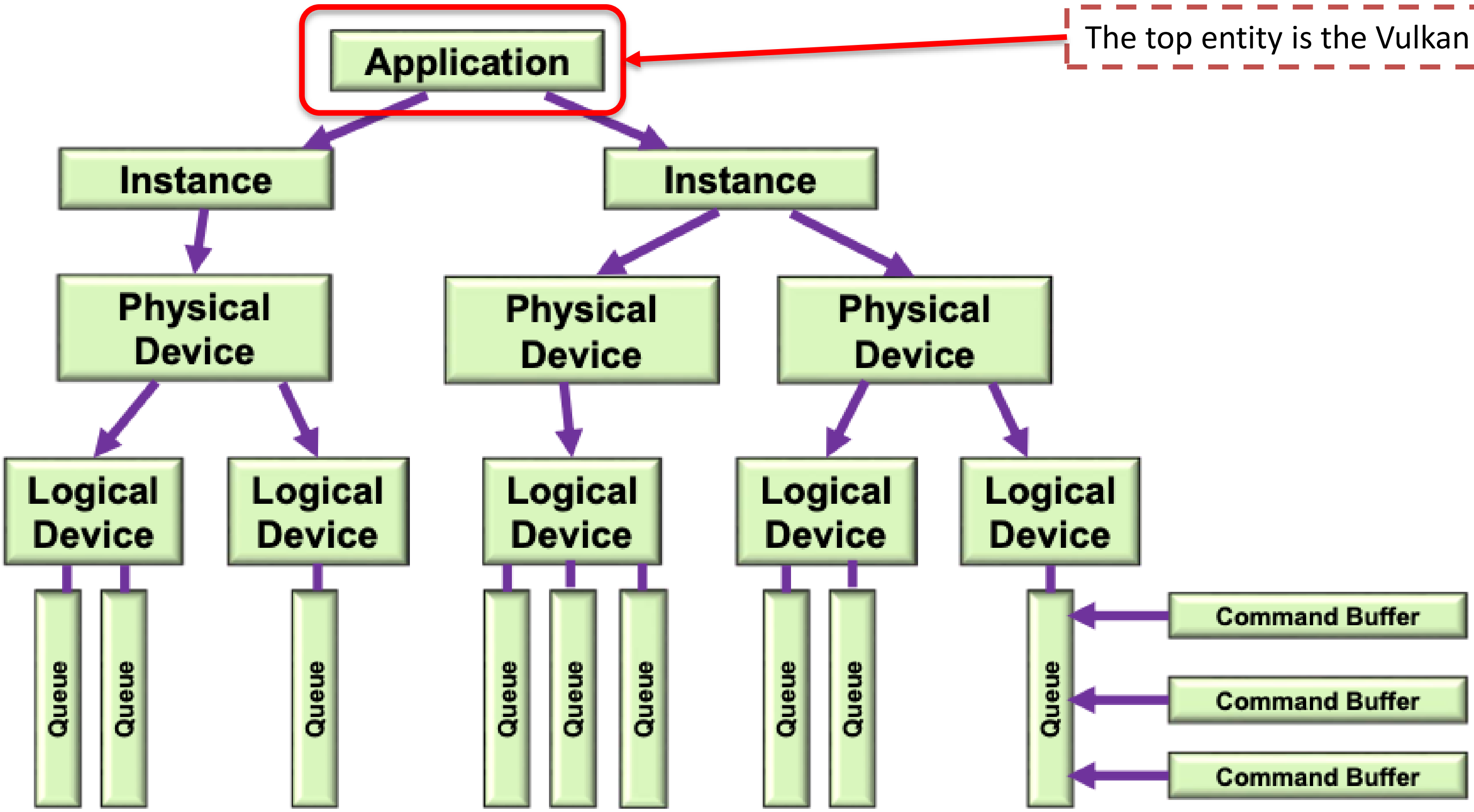
```
void initWindow() {
    glfwInit();

    glfwWindowHint(GLFW_CLIENT_API, GLFW_NO_API);

    window = glfwCreateWindow(WIDTH, HEIGHT, "Vulkan", nullptr, nullptr);


}
```

Vulkan initialization

The initialization of the Vulkan support is quite complex due to its large number of alternatives.

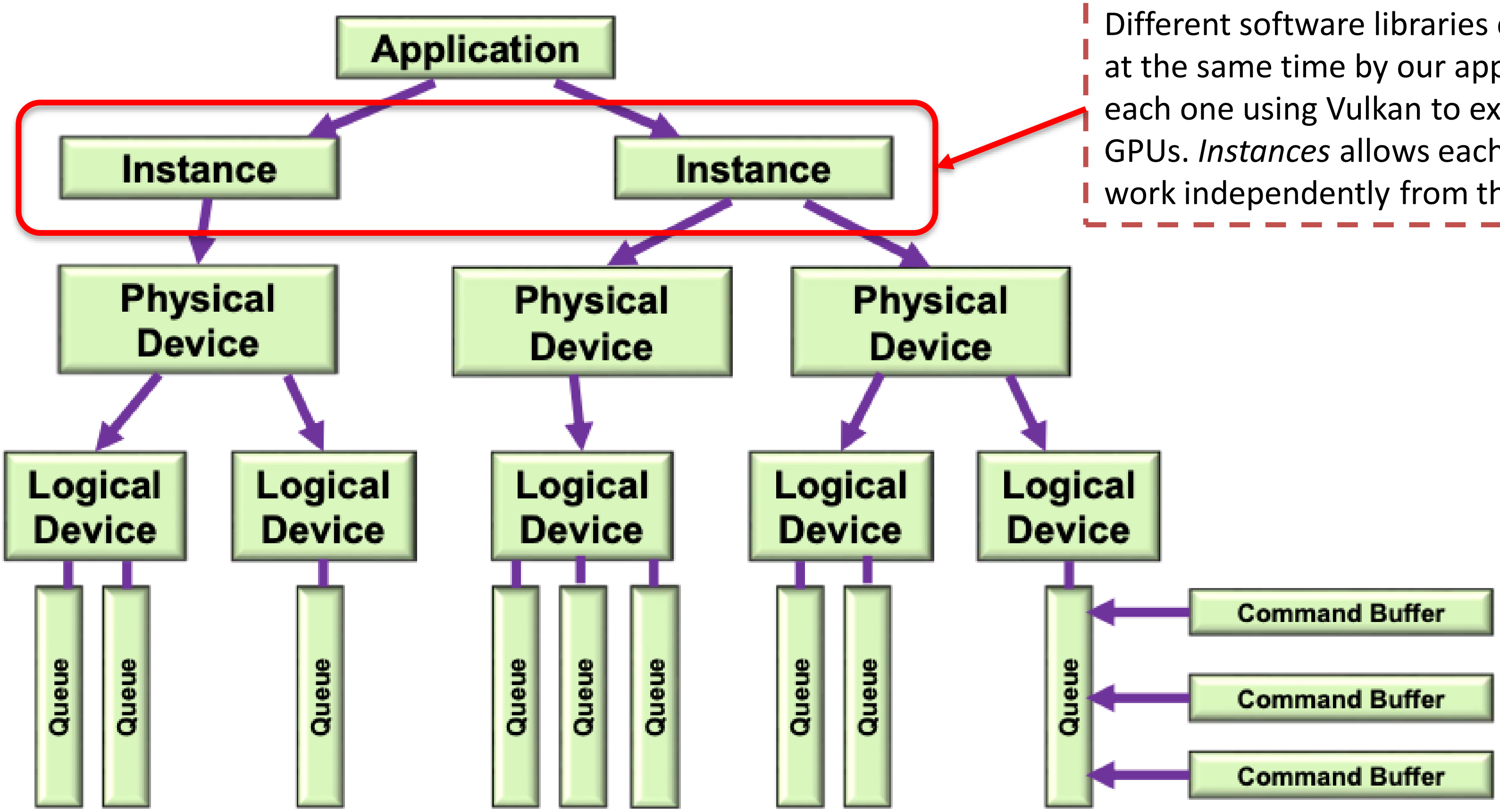
In order to understand it, we need to start from an high-level overview of an application.

A Vulkan architecture



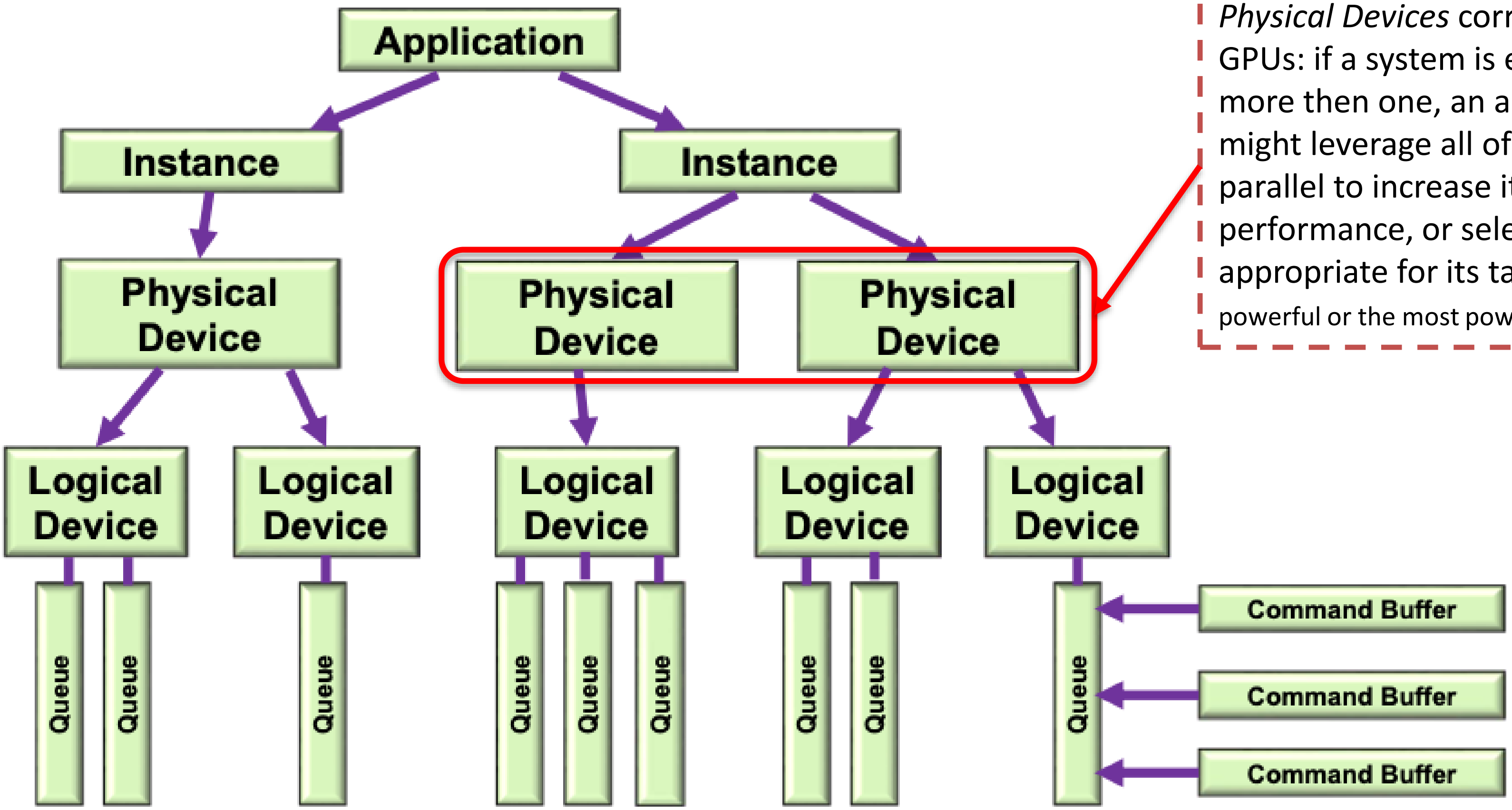
The top entity is the Vulkan *Application*.

A Vulkan architecture



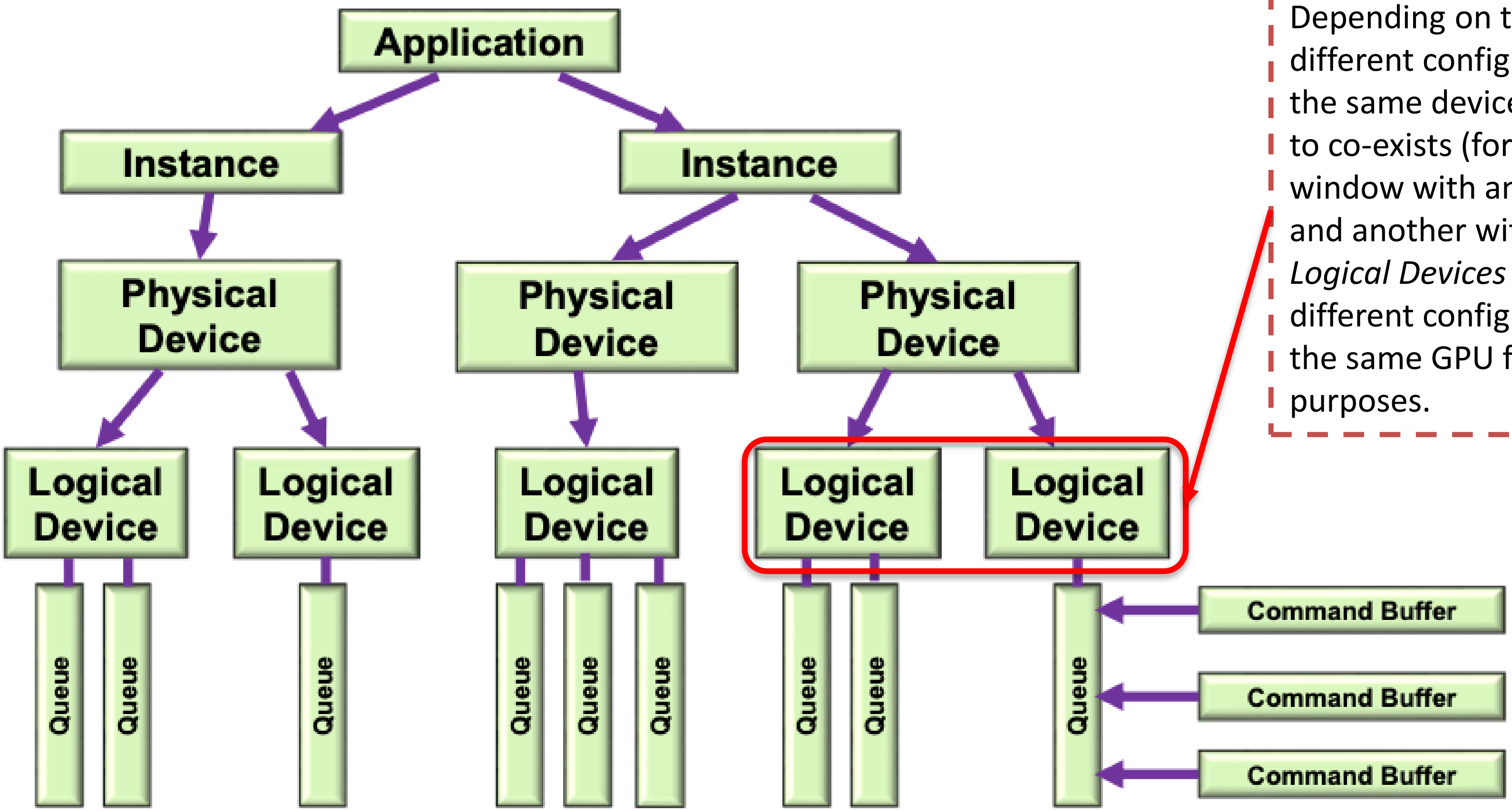
Different software libraries can be used at the same time by our application, each one using Vulkan to exploit the GPUs. *Instances* allows each library to work independently from the others.

A Vulkan architecture



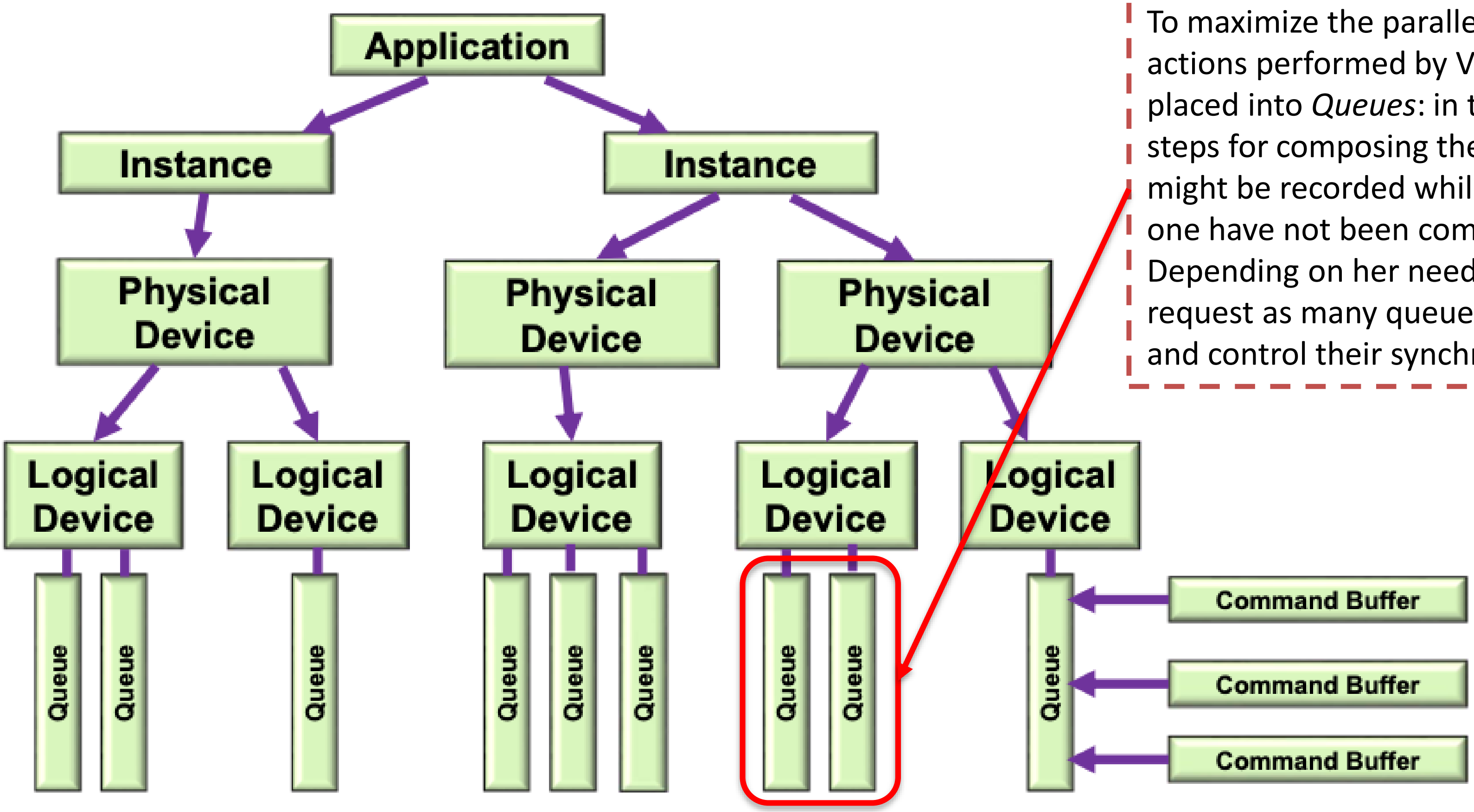
Physical Devices corresponds to GPUs: if a system is equipped with more then one, an application might leverage all of them in parallel to increase its performance, or select the most appropriate for its task (i.e. the most powerful or the most power efficient).

A Vulkan architecture



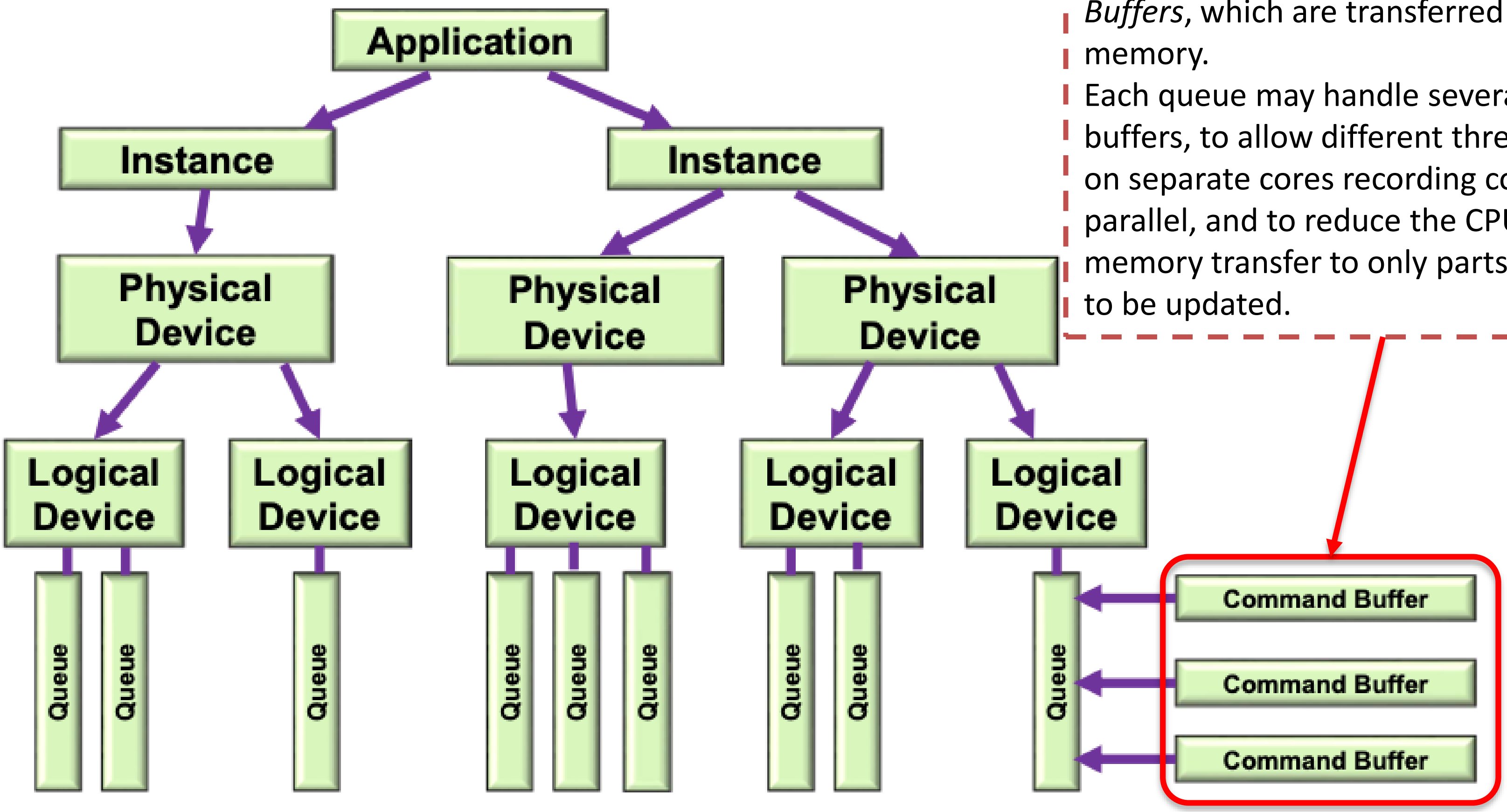
Depending on the needs, different configurations of the same device might need to co-exists (for example, a window with antialiasing, and another without). *Logical Devices* allow to use different configurations of the same GPU for different purposes.

A Vulkan architecture



To maximize the parallelization, all actions performed by Vulkan are placed into *Queues*: in this way, the steps for composing the next image might be recorded while the previous one have not been completed yet. Depending on her needs, the user can request as many queues as necessary, and control their synchronization.

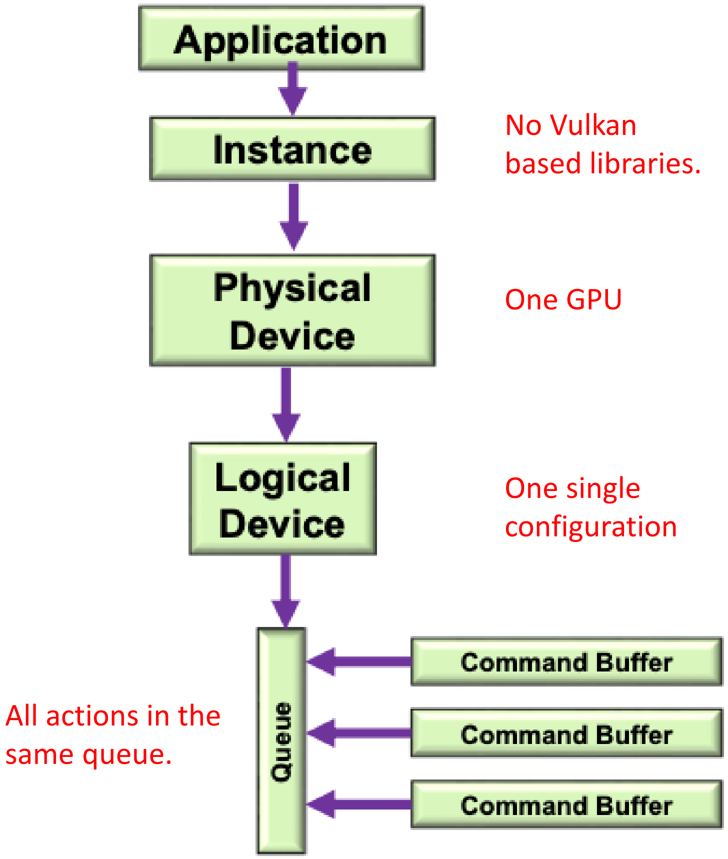
A Vulkan architecture



Vulkan operations are stored in *Command Buffers*, which are transferred into GPU memory.
Each queue may handle several command buffers, to allow different threads running on separate cores recording commands in parallel, and to reduce the CPU to GPU memory transfer to only parts that need to be updated.

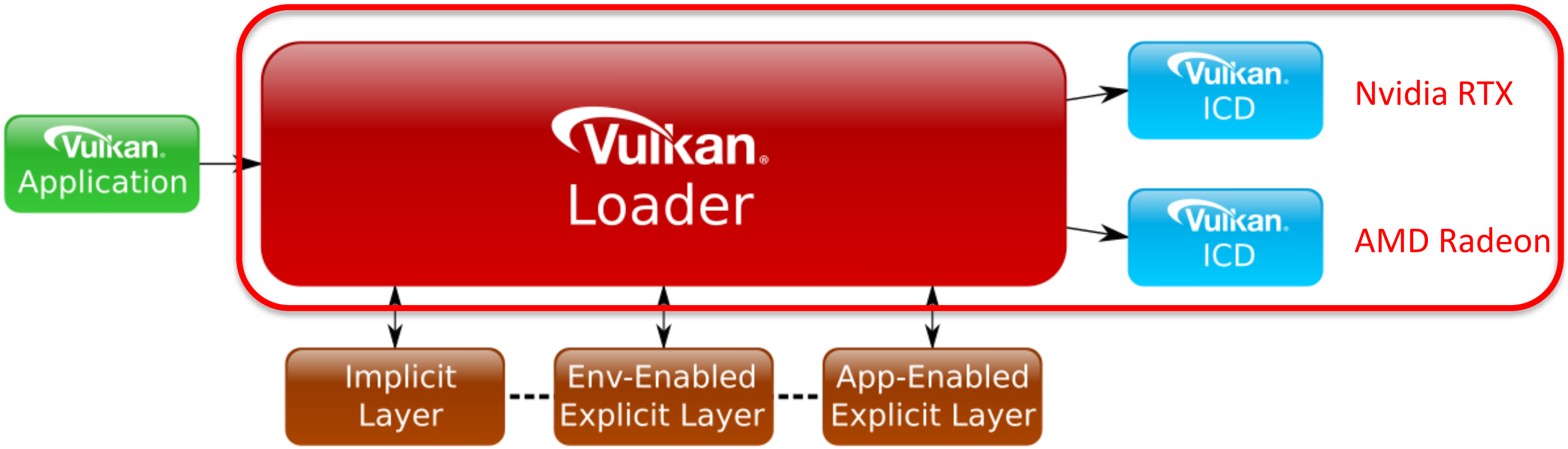
A typical Vulkan Application

Most Vulkan Applications, however, will use only a single library instance, they will run on a single GPU, using just one configuration, and they will perform all commands in a single queue.



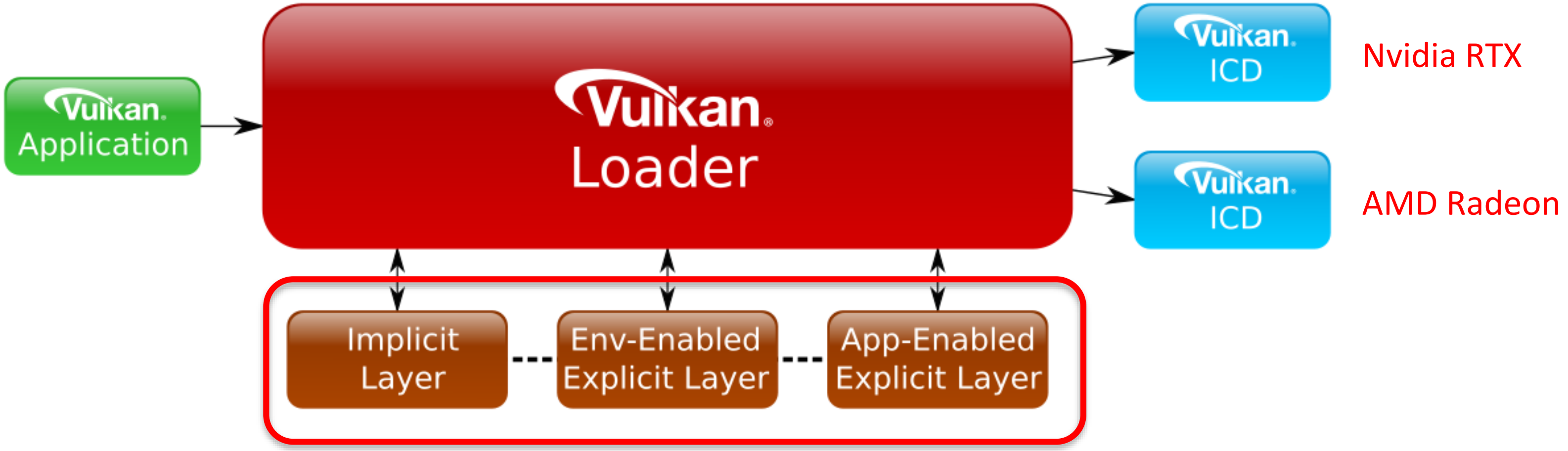
The Extensions mechanism

The way in which Vulkan is structured, is very complex. There is a fixed component, namely the *Vulkan Loader*, plus a set of GPU drivers called *Installable Client Devices (ICD)*.



The Extensions mechanism

A specific Vulkan deployment can add a set of *Extension Layers*, that can be used to expose *O.S. specific* or *Device specific* functions. These functions allow Vulkan to work in a given environment, and to access special hardware features.



The Extensions mechanism

Here you can find for example a list of global extensions on MacOS and Windows:

```
Available global extensions:
	VK_KHR_device_group_creation
	VK_KHR_external_fence_capabilities
	VK_KHR_external_memory_capabilities
	VK_KHR_external_semaphore_capabilities
	VK_KHR_get_physical_device_properties2
	VK_KHR_get_surface_capabilities2
	VK_KHR_surface
	VK_EXT_debug_report
	VK_EXT_debug_utils
	VK_EXT_metal_surface
	VK_EXT_swapchain_colorspace
	VK_MVK_macos_surface

Extensions required by GLFW:
	VK_KHR_surface
	VK_MVK_macos_surface
```

MacOS 10.15

```
Available global extensions:
	VK_KHR_device_group_creation
	VK_KHR_external_fence_capabilities
	VK_KHR_external_memory_capabilities
	VK_KHR_external_semaphore_capabilities
	VK_KHR_get_physical_device_properties2
	VK_KHR_get_surface_capabilities2
	VK_KHR_surface
	VK_KHR_win32_surface
	VK_EXT_debug_report
	VK_EXT_debug_utils
	VK_EXT_swapchain_colorspace
	VK_KHR_display
	VK_KHR_get_display_properties2
	VK_KHR_surface_protected_capabilities
	VK_NV_external_memory_capabilities

Extensions required by GLFW:
	VK_KHR_surface
	VK_KHR_win32_surface
```

Windows 11

Instance creation

In order to create an instance, the following information should be specified:

- List of requested extensions
- Name, and other features of the application

To allow an application to be platform independent, the interface library GLFW has a command called `glfwGetRequiredInstanceExtensions()` to retrieve the extensions required by the specific architecture the program is running on.

The minimal main loop

The minimal main loop, just waits for the user to close the window with the `glfwWindowShouldClose(…)` and the repeatedly calls the `glfwPollEvents()` command to check if there has been some input from the user:

```
void mainLoop() {
    while (!glfwWindowShouldClose(window)) {
        glfwPollEvents();
    }
}
```

In our applications based on `Starter.hpp`, we can trigger the closure of a window with the following call:

```
// Standard procedure to quit when the ESC key is pressed
if(glfwGetKey(window, GLFW_KEY_ESCAPE)) {
    glfwSetWindowShouldClose(window, GL_TRUE);
}
```

Resources release

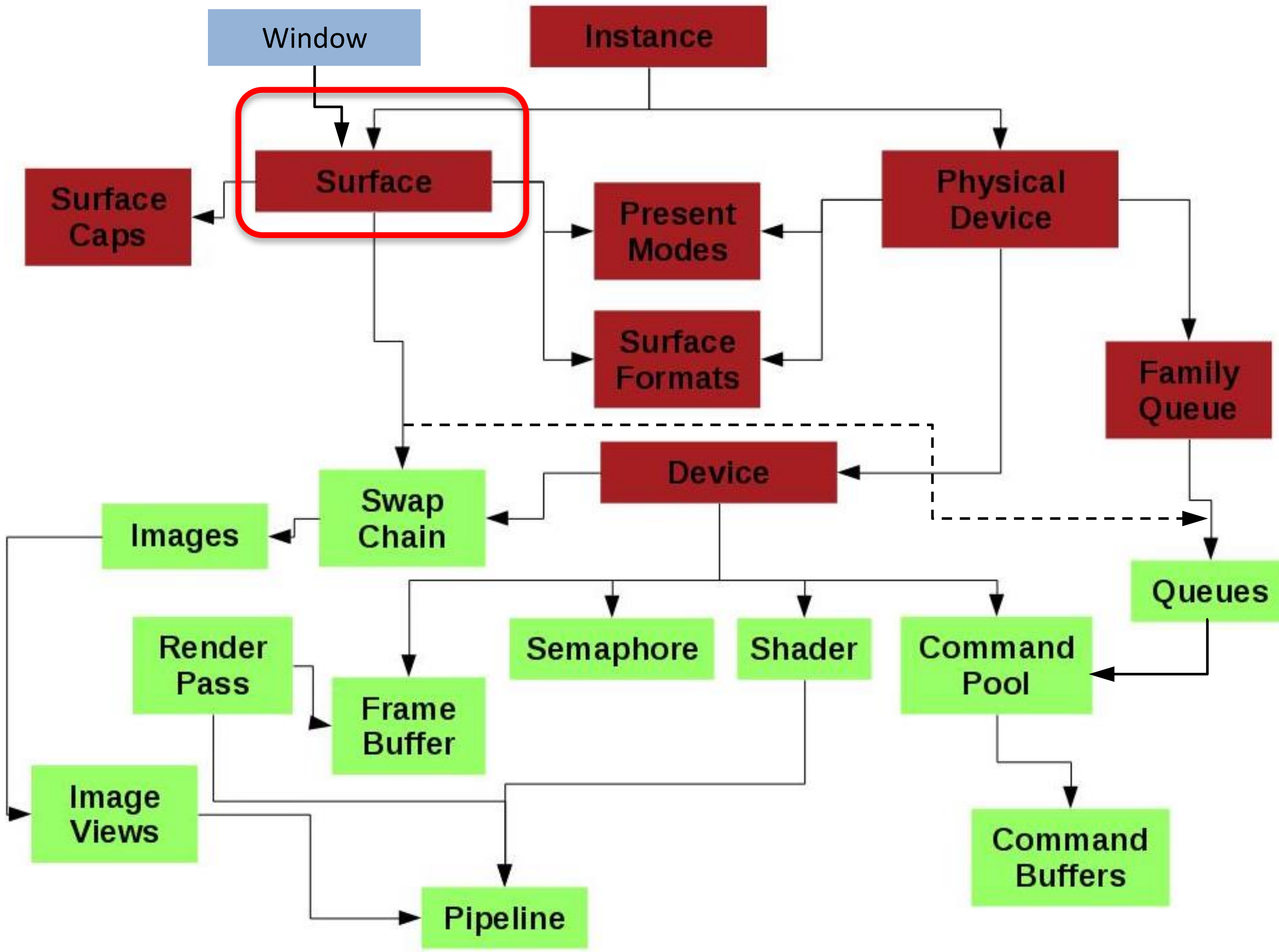
Instance should be released with `vkDestroyInstance(…)` and the O.S. window closed with `glfwDestroyWindow(…)`. The GLFW library, requires also a call to `glfwTerminate()` for freeing all its remaining resources,

```
 void cleanup() {
    vkDestroyInstance(instance, nullptr);

    glfwDestroyWindow(window);

    glfwTerminate();
}
```

Presentation surface

The presentation surface requires both the Window and to the Vulkan instance.

For this reason, it can be created only after both steps have been accomplished.



Presentation surface

GLFW can create the presentation surface with command `glfwCreateWindowSurface()`, and returns a handle to the considered surface in the `VkSurfaceKHR` object whose pointer is passed as the last argument of the function.

```
// create the Surface [requires the Vulkan Instance and
// the window to be already created]
VkSurfaceKHR surface;
if (glfwCreateWindowSurface(instance, window, nullptr, &surface) != VK_SUCCESS) {
     throw std::runtime_error("failed to create window surface!");
}

…

vkDestroySurfaceKHR(instance, surface, nullptr);
vkDestroyInstance(instance, nullptr);
glfwDestroyWindow(window);
```

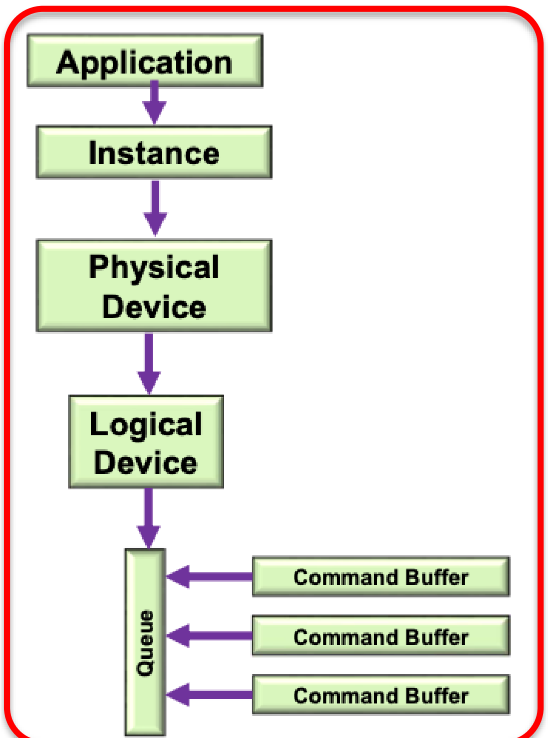
Presentation surface

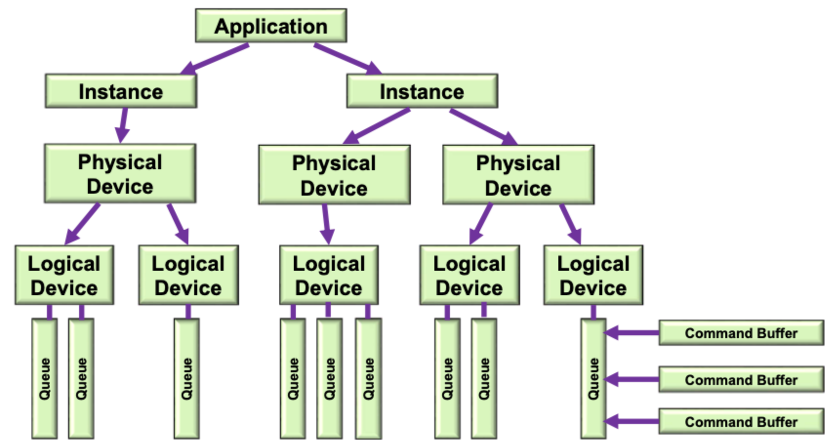
Presentation surface needs to be released at the end of the application, before destroying the instance and the window, using the `vkDestroySurfaceKHR()` command.

```
// create the Surface [requires the Vulkan Instance and
// the window to be already created]
VkSurfaceKHR surface;
if (glfwCreateWindowSurface(instance, window, nullptr, &surface) != VK_SUCCESS) {
     throw std::runtime_error("failed to create window surface!");
}

…

vkDestroySurfaceKHR(instance, surface, nullptr);
vkDestroyInstance(instance, nullptr);
glfwDestroyWindow(window);
```

Vulkan Physical Devices, Logical Devices and Queues

Even if in this course we will consider only applications exploiting a single GPU, the considered system might have more than one available.



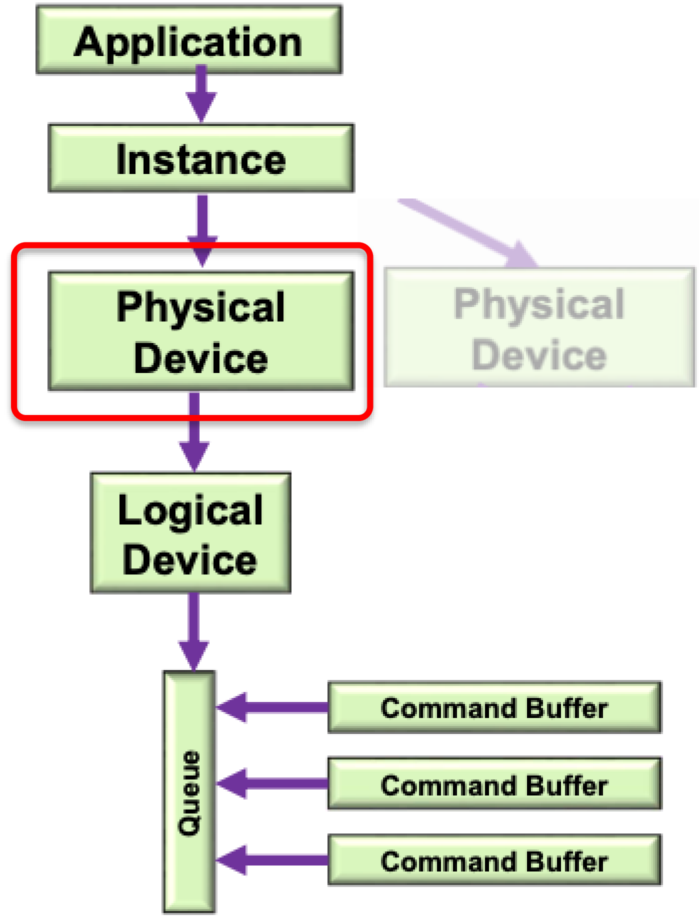


Vulkan Physical Devices

`Starter.hpp` follows a standard procedure to select the most appropriate one, in a system independent way!

This is achieved by:

- Enumerating the devices
- Checking their features
- Ranking them according to the requirements
- Selecting the one with the highest rank



Vulkan Physical Devices

Each device is characterized by sets of:

- *Properties*: i.e. manufacturer, whether it is integrated in the CPU or separate, drivers id, etc…
- *Features*: support for specific types of shaders, data types, graphics commands, etc…
- *Memory Types*: shared, GPU specific, etc…
- *Memory Heaps*: how much memory is available
- *Supported Queue Families*: which type of operations it can perform.

Available Properties and Features

Each *Property* has a field called `limits`, where the maximum sizes of the supported objects is shown.

The *Features* structure has a lot more fields, each one containing a feature that can be selected during logical device creation.

If interested, see the official documentation for a complete list.

Properties: https://www.khronos.org/registry/vulkan/specs/1.3-extensions/man/html/VkPhysicalDeviceProperties.html
Limits: https://www.khronos.org/registry/vulkan/specs/1.3-extensions/man/html/VkPhysicalDeviceLimits.html
Features: https://www.khronos.org/registry/vulkan/specs/1.3-extensions/man/html/VkPhysicalDeviceFeatures.html

Memory Types

Memory types describes whether the corresponding type is CPU visible, GPU only and how it can be interfaced from Vulkan.

```
for(unsigned int i = 0; i < vpdmp.memoryTypeCount; i++) {
     VkMemoryType vmt = vpdmp.memoryTypes[i];

     if((vmt.propertyFlags & VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT) != 0 )
          std::cout << " DeviceLocal";
     if((vmt.propertyFlags & VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT) != 0 )
          std::cout << " HostVisible";
     if((vmt.propertyFlags & VK_MEMORY_PROPERTY_HOST_COHERENT_BIT) != 0 )
          std::cout << " HostCoherent";
     if((vmt.propertyFlags & VK_MEMORY_PROPERTY_HOST_CACHED_BIT) != 0 )
          std::cout << " HostCached";
     if((vmt.propertyFlags & VK_MEMORY_PROPERTY_LAZILY_ALLOCATED_BIT) != 0 )
          std::cout << " LazilyAllocated";
     std::cout << "\n";
}
```

https://www.khronos.org/registry/vulkan/specs/1.3-extensions/man/html/VkMemoryPropertyFlagBits.html

Memory Heaps

Memory Heaps define the quantity of available memory, and whether they are local for the GPU.

```
for(unsigned int i = 0; i < vpdmp.memoryHeapCount; i++ ) {
     VkMemoryHeap vmh = vpdmp.memoryHeaps[i];

     std::cout << " size = " << std::hex << (unsigned long int)vmh.size;
     if((vmh.flags & VK_MEMORY_HEAP_DEVICE_LOCAL_BIT) != 0)
          std::cout << " DeviceLocal";
     std::cout << "\n";
}
```

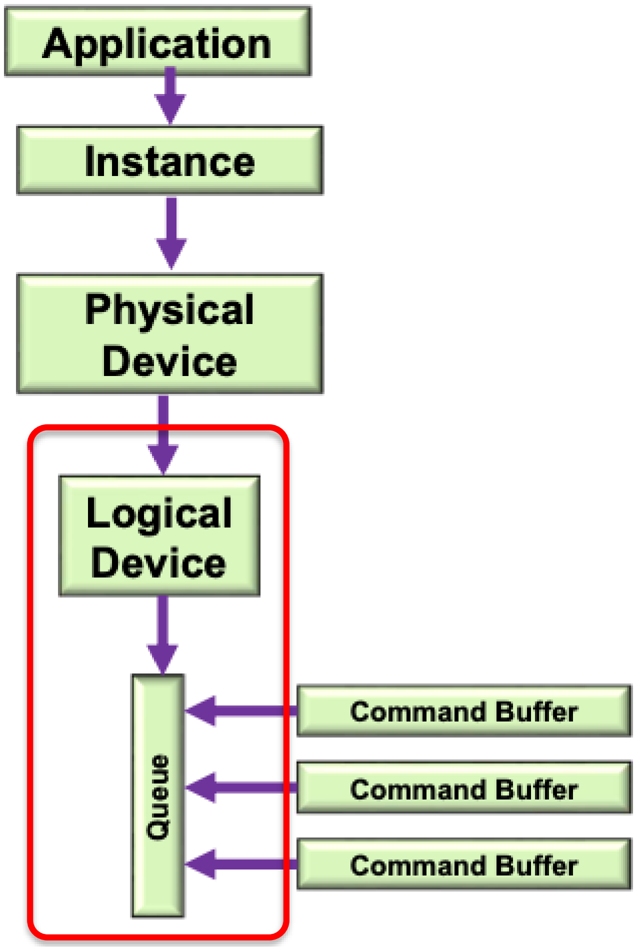
https://www.khronos.org/registry/vulkan/specs/1.3-extensions/man/html/VkMemoryHeapFlagBits.html

Logical Devices and Queues

As we have seen, from a *Physical Device*, *Logical Devices* are created, each one containing one or more *Queues*.

Each Physical Device can support different types of Queues.

The selection of Physical Device might be determined also by the Queues its Logical Devices can use.



Queue families

Queues are grouped into *Families*, each one supporting different type of operations they can execute.

Families supported by a Physical Device can be enumerated with the function `vkGetPhysicalDeviceQueueFamilyProperties()`.

```
// Queues
uint32_t queueFamCount = -1;
vkGetPhysicalDeviceQueueFamilyProperties(physicalDevice, &queueFamCount, nullptr);
std::cout << "\n\tQueue Families found: " << queueFamCount << "\n";
std::vector<VkQueueFamilyProperties> queues(queueFamCount);
vkGetPhysicalDeviceQueueFamilyProperties(physicalDevice, &queueFamCount, queues.data());
```

Queue families

There are several types of operations that a Queue can perform.

- Graphics
- Compute
- Transfer
- Sparse Memory Management
- Presentation

https://www.khronos.org/registry/vulkan/specs/1.3-extensions/man/html/VkQueueFlagBits.html

Queue families

A Computer Graphics Application, requires at least a Graphic and a Presentation queue.

Depending on the system, these might be separate or supported by one specific queue family.

A logic capable of supporting both options is then required: this has been implemented in `Starter.hpp` following directly *the Vulkan tutorial.*

Logical device creation

Logical devices are created together with their queues in using the `vkCreateDevice()` command, starting from a Physical Device.

```
VkDevice device;
VkPhysicalDeviceFeatures deviceFeatures{};
VkDeviceCreateInfo createInfo{};
createInfo.sType = VK_STRUCTURE_TYPE_DEVICE_CREATE_INFO;
createInfo.pQueueCreateInfos = queueCreateInfos.data();
createInfo.queueCreateInfoCount = static_cast<uint32_t>(queueCreateInfos.size());
createInfo.pEnabledFeatures = &deviceFeatures;
createInfo.enabledExtensionCount = 0;
createInfo.ppEnabledExtensionNames = nullptr;
createInfo.enabledLayerCount = 0;

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

Logical device creation

During the creation, Device features are enabled, togheter with *Extensions* and *Debug Layers* (we will return on this later).

```
VkDevice device;
VkPhysicalDeviceFeatures deviceFeatures{};
VkDeviceCreateInfo createInfo{};
createInfo.sType = VK_STRUCTURE_TYPE_DEVICE_CREATE_INFO;
createInfo.pQueueCreateInfos = queueCreateInfos.data();
createInfo.queueCreateInfoCount = static_cast<uint32_t>(queueCreateInfos.size());
createInfo.pEnabledFeatures = &deviceFeatures;
createInfo.enabledExtensionCount = 0;
createInfo.ppEnabledExtensionNames = nullptr;
createInfo.enabledLayerCount = 0;

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

Queue retrieval

Once device has been successfully created, queue handles must be retrieved using the `vkGetDeviceQueue()` command.

Again, since more queues per family can be created, the command requires both the family index, and the queue id (here 0).

```
VkQueue graphicsQueue;
VkQueue presentQueue;

vkGetDeviceQueue(device, aQueueWithGraphicsCapability.value(), 0, &graphicsQueue);
vkGetDeviceQueue(device, aQueueWithPresentationCapability.value(), 0, &presentQueue);
```

Device release

At the end of the application, Logical Devices must be released.

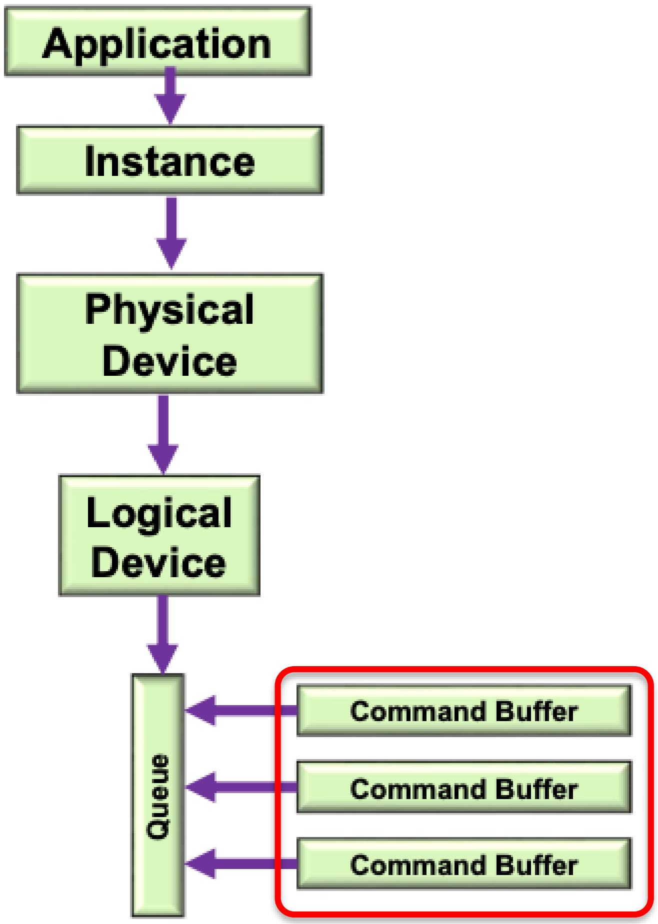
```
vkDestroyDevice(device, nullptr);

vkDestroySurfaceKHR(instance, surface, nullptr);
vkDestroyInstance(instance, nullptr);
glfwDestroyWindow(window);
```

Command buffer creation

Once queues have been retrieved, *Command Buffers* using them can be created.

Since the use of several command buffers is common, they are allocated from larger groups called *Command Pools*.

Each *Command Pool* is strictly connected to the *Queue* families it uses.



Command Pools

Command Pools are created with the `vkCreateCommandPool()` function. The only parameter that needs to be defined in the creation structure is the *Queue* family on which its commands will be executed using the `queueFamilyIndex` field.

```
VkCommandPool commandPool;

VkCommandPoolCreateInfo poolInfo{};
poolInfo.sType = VK_STRUCTURE_TYPE_COMMAND_POOL_CREATE_INFO;
poolInfo.queueFamilyIndex = aQueueWithGraphicsCapability.value();
poolInfo.flags = 0; // Optional

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

Currently, we are only interested in the queue for graphics creation.

Command Pools

Command Pools must be released when no longer necessary with the `vkDestroyCommandPool()` function.

```
vkDestroyCommandPool(device, commandPool, nullptr);
vkDestroyDevice(device, nullptr);

vkDestroySurfaceKHR(instance, surface, nullptr);
vkDestroyInstance(instance, nullptr);
glfwDestroyWindow(window);
```

Command Buffers

Command Buffers are created from the pools with the `vkAllocateCommandBuffers()` function, and their handle is returned in a `VkCommandBuffer` object.

The corresponding Pool handle is passed in the `commandPool` field of the creation structure.

```
VkCommandBuffer commandBuffer;

VkCommandBufferAllocateInfo allocInfo{};
allocInfo.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_ALLOCATE_INFO;
allocInfo.commandPool = commandPool;
allocInfo.level = VK_COMMAND_BUFFER_LEVEL_PRIMARY;
allocInfo.commandBufferCount = 1;

result = vkAllocateCommandBuffers(device, &allocInfo, &commandBuffer);
if (result != VK_SUCCESS) {
     throw std::runtime_error("failed to allocate command buffer!");
}
```

Command Buffers

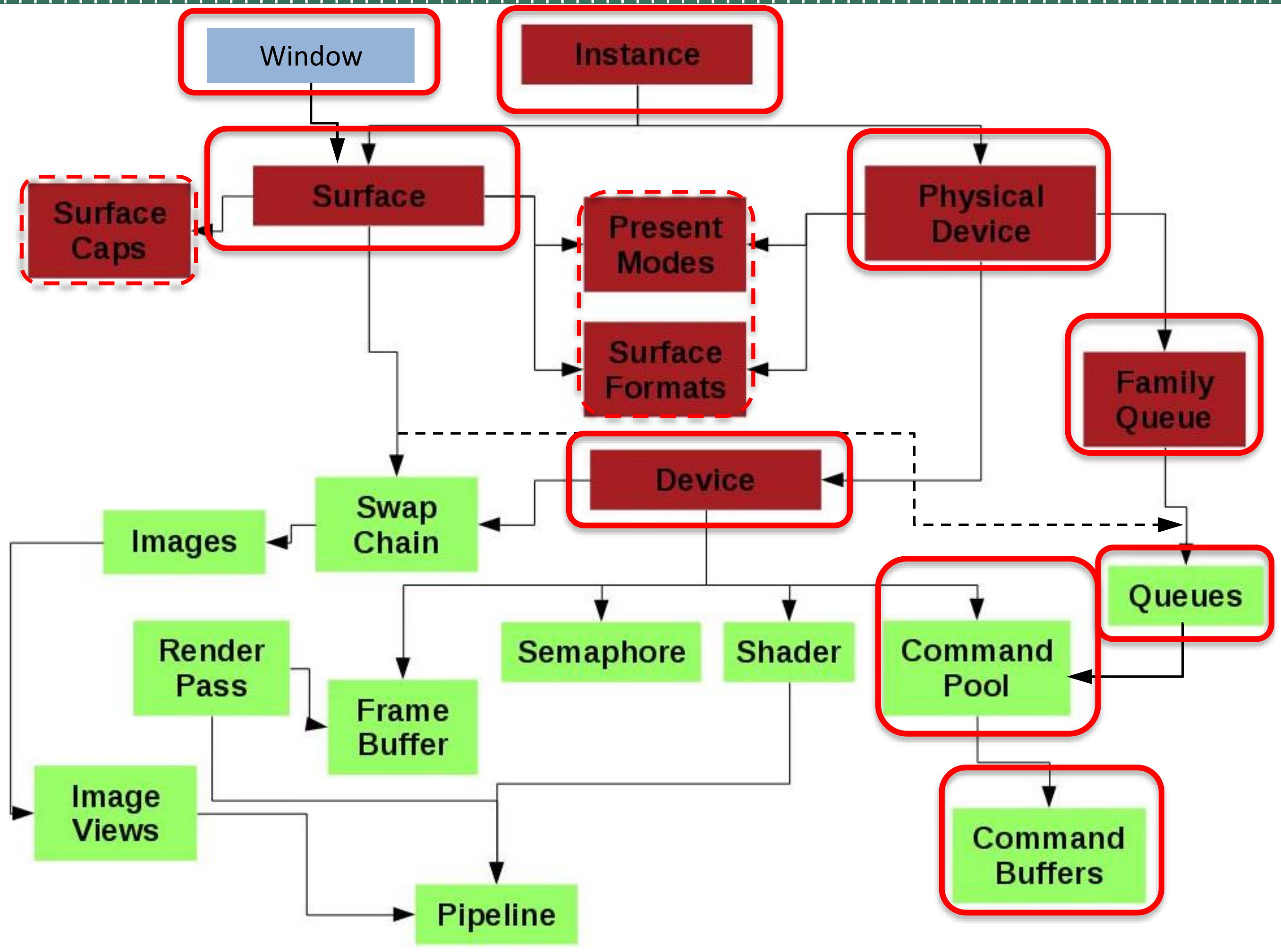
Several command buffers could be created in the same call: their number is specified in the `commandBufferCount` field (if more than one buffer is required, the return value must be an array of sufficient size).

Command Buffers are automatically destroyed when the corresponding Pool is released, so no explicit action is required.

```
VkCommandBuffer commandBuffer;

VkCommandBufferAllocateInfo allocInfo{};
allocInfo.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_ALLOCATE_INFO;
allocInfo.commandPool = commandPool;
allocInfo.level = VK_COMMAND_BUFFER_LEVEL_PRIMARY;
allocInfo.commandBufferCount = 1;

result = vkAllocateCommandBuffers(device, &allocInfo, &commandBuffer);
if (result != VK_SUCCESS) {
     throw std::runtime_error("failed to allocate command buffer!");
}
```

Initialization dependencies

In this picture, we highlight the dependencies between the objects that have been considered up to now.

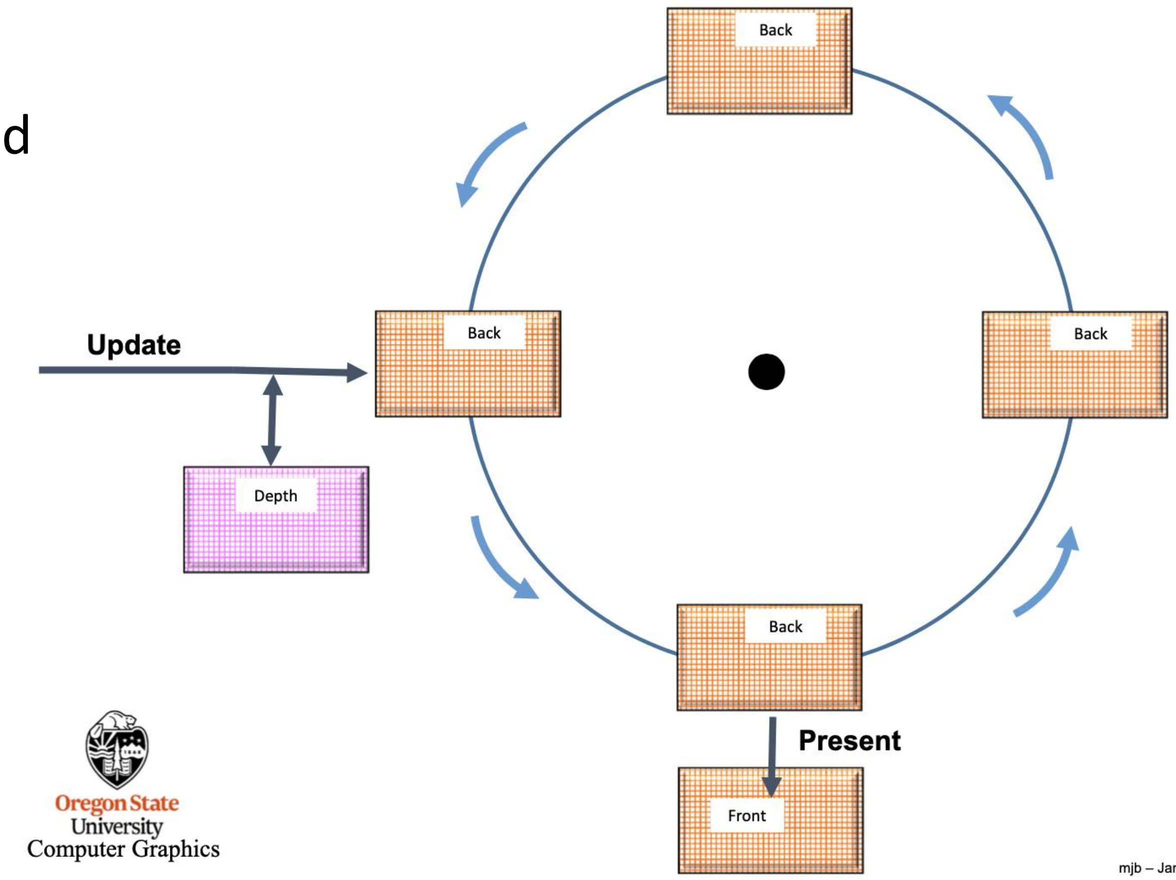


Dashed boxes represent ancillary data structures, required to define the parameters for the main blocks, but not much important in a Vulkan overview.

The Vulkan Swap Chain

In Vulkan, Screen Synchronization is handled with a generic circular queue, called the *Swap Chain*.

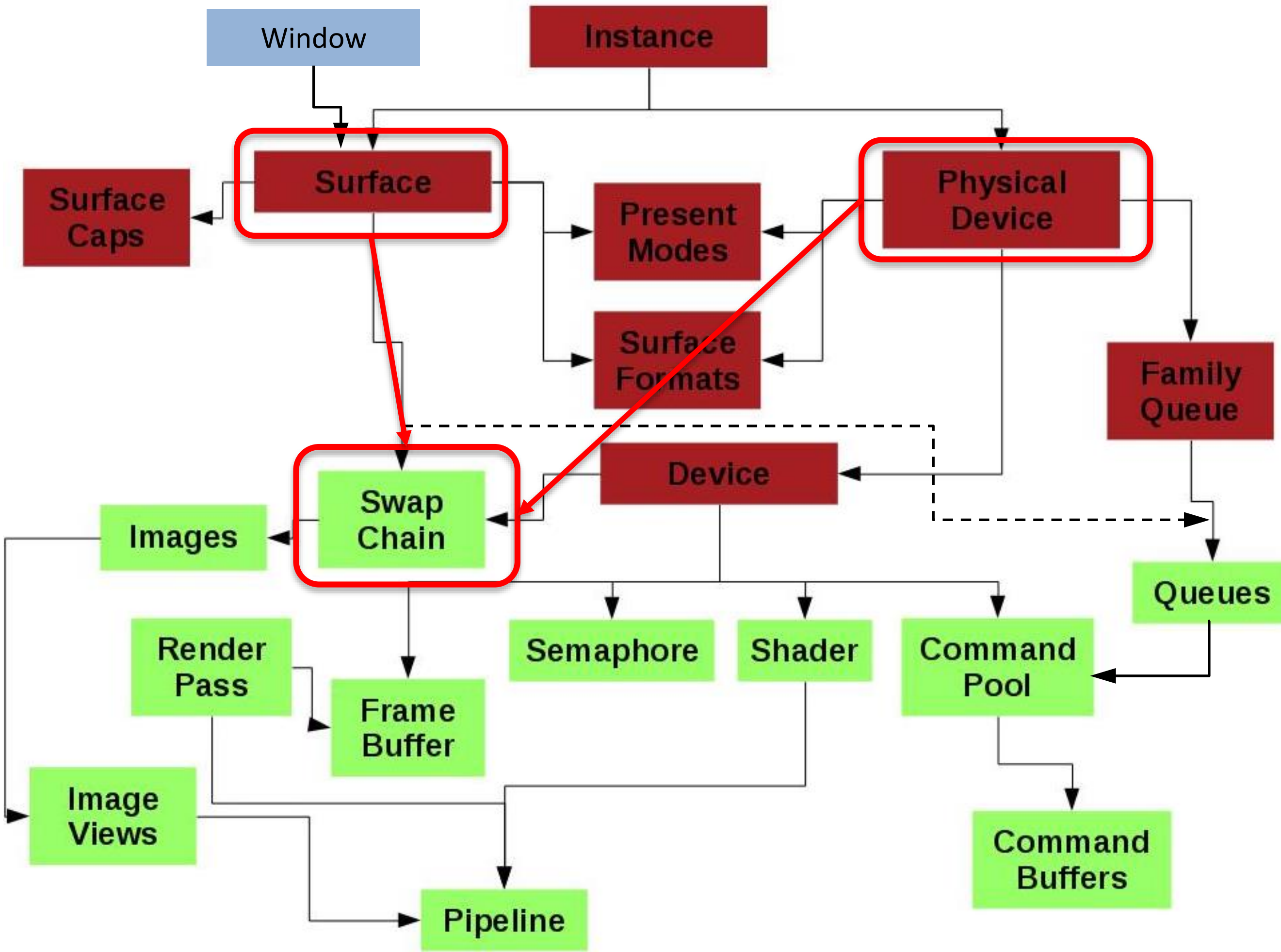
It can handle Single, Double, Triple buffer and potentially even longer presentation queues.



From: https://web.engr.oregonstate.edu/~mjb/vulkan/

Swap Chain properties

Swap Chain properties depends on the Surface / Physical Device combination.



Swap Chain properties

Each swap chain is characterized by:

- A set of capabilities
- Several supported formats
- Several presentation modes

Swap Chain capabilities

Swap Chain Capabilities account for basic information such as number of buffers supported, and graphical extents.

```
// Provided by VK_KHR_surface
typedef struct VkSurfaceCapabilitiesKHR {
    uint32_t                         minImageCount;
    uint32_t                         maxImageCount;
    VkExtent2D                       currentExtent;
    VkExtent2D                       minImageExtent;
    VkExtent2D                       maxImageExtent;
    uint32_t                         maxImageArrayLayers;
    VkSurfaceTransformFlagsKHR       supportedTransforms;
    VkSurfaceTransformFlagBitsKHR    currentTransform;
    VkCompositeAlphaFlagsKHR         supportedCompositeAlpha;
    VkImageUsageFlags                supportedUsageFlags;
} VkSurfaceCapabilitiesKHR;
```

Number of buffers

Screen size

Layers (for example left and right eye)

Screen rotation and mirroring
(for hand held devices and projectors)

https://www.khronos.org/registry/vulkan/specs/1.3-extensions/man/html/VkSurfaceCapabilitiesKHR.html

Swap Chain formats

Even if colors are encoded using the RGB system, several alternative formats, with different color spaces and resolution exist.

Each graphic adapter can support a variety of them (i.e. 8bpc, 10bpc, 16bpc), each one defining a different tradeoff between memory, performance and quality.

Swap Chain Formats are characterized by the number of bits and components (defined in an enumeration), and the corresponding color profile.

Swap Chain Presentation Modes

Presentation Modes are the equivalent of synchronization algorithms in Vulkan terminology. They are returned as an array of `VkPresentModeKHR` enumerations, with the `vkGetPhysicalDeviceSurfacePresentModesKHR` command.

```
uint32_t presentModeCount;
vkGetPhysicalDeviceSurfacePresentModesKHR(physicalDevice, surface,
     &presentModeCount, nullptr);

std::vector<VkPresentModeKHR> SCpresentModes;
if (presentModeCount != 0) {
     SCpresentModes.resize(presentModeCount);
     vkGetPhysicalDeviceSurfacePresentModesKHR(physicalDevice, surface,
               &presentModeCount, SCpresentModes.data());
     std::cout << "\t Supported Modes: " << presentModeCount << "\n";
     for(int i; i < presentModeCount; i++) {
          switch(SCpresentModes[i]) {
            case VK_PRESENT_MODE_IMMEDIATE_KHR:
               std::cout << "\t\tVK_PRESENT_MODE_IMMEDIATE_KHR\n";
               break;
            ...
```

Swap Chain Presentation Modes

Presentation Modes are the equivalent of synchronization algorithms in Vulkan terminology. Four main presentation modes are supported:

```
// Provided by VK_KHR_surface
typedef enum VkPresentModeKHR {
    VK_PRESENT_MODE_IMMEDIATE_KHR = 0,
    VK_PRESENT_MODE_MAILBOX_KHR = 1,
    VK_PRESENT_MODE_FIFO_KHR = 2,
    VK_PRESENT_MODE_FIFO_RELAXED_KHR = 3,
  // Provided by VK_KHR_shared_presentable_image
    VK_PRESENT_MODE_SHARED_DEMAND_REFRESH_KHR = 1000111000,
  // Provided by VK_KHR_shared_presentable_image
    VK_PRESENT_MODE_SHARED_CONTINUOUS_REFRESH_KHR = 1000111001,
} VkPresentModeKHR;
```

Single Buffer

Triple Buffer

Double Buffer

https://www.khronos.org/registry/vulkan/specs/1.3-extensions/man/html/VkPresentModeKHR.html

Swap Chain release

When no longer needed, swap chains can be released with the `vkDestroySwapchainKHR()` command.

```
vkDestroySwapchainKHR(device, swapChain, nullptr);
vkDestroyDevice(device, nullptr);
std::cout << "\tResources released correctly\n\n";
```

Swap Chain Images retrieval

Each buffer of the swap chain, is considered by Vulkan as a generic image which must be retrieved after creation. Images are identified by `VkImage` objects, and the ones corresponding to the swap chain are retrieved with the `vkGetSwapchainImagesKHR` command, using the two calls procedure.

```
std::vector<VkImage> swapChainImages;

vkGetSwapchainImagesKHR(device, swapChain, &imageCount, nullptr);
swapChainImages.resize(imageCount);
vkGetSwapchainImagesKHR(device, swapChain, &imageCount,
                        swapChainImages.data());
```

Image Views

Images can be of very different formats, and might be used for a lot of different purposes.

A lot of extra information is required to tell how images are structured and how their pixel can be accessed.

Image Views, are the way in which Vulkan associate to each image, the description on how it can be used and accessed, and they are necessary to support them.

Swap chain Image Views retrieval

After retrieving the swap chain images, their Views (contained in `VkImageView` objects) must be created using the `vkCreateImageView()` command.

```
std::vector<VkImageView> swapChainImageViews;
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 = surfaceFormat.format;
    viewInfo.subresourceRange.aspectMask =
                                VK_IMAGE_ASPECT_COLOR_BIT;
    viewInfo.subresourceRange.baseMipLevel = 0;
    viewInfo.subresourceRange.levelCount = 1;
    viewInfo.subresourceRange.baseArrayLayer = 0;
    viewInfo.subresourceRange.layerCount = 1;

    VkImageView imageView;

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

Swap chain Image Views retrieval

The most important information, is the corresponding image, contained in the `image` field.

On the other values, we will returns in the following.

```
std::vector<VkImageView> swapChainImageViews;
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 = surfaceFormat.format;
	viewInfo.subresourceRange.aspectMask =
	                              VK_IMAGE_ASPECT_COLOR_BIT;
	viewInfo.subresourceRange.baseMipLevel = 0;
	viewInfo.subresourceRange.levelCount = 1;
	viewInfo.subresourceRange.baseArrayLayer = 0;
	viewInfo.subresourceRange.layerCount = 1;

	VkImageView imageView;

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

Swap chain Image Views release

Swap chain images are destroyed with the `VkSwapchainKHR` object. Image views, however, must be explicitly destroyed with the `vkDestroyImageView()` command.

```
for (size_t i = 0; i < swapChainImageViews.size(); i++){
        vkDestroyImageView(device, swapChainImageViews[i], nullptr);
}
vkDestroySwapchainKHR(device, swapChain, nullptr);
vkDestroyDevice(device, nullptr);
```

Marco Gribaudo

Associate Professor

CONTACTS

Tel. +39 02 2399 3568
marco.gribaudo@polimi.it
https://www.deib.polimi.it/eng/home-page

(Remember to use the phone, since mails might require a lot of time to be answered. Microsoft Teams messages might also be faster than regular mails)