Rendering techniques and Command Buffers creation

Lights basics

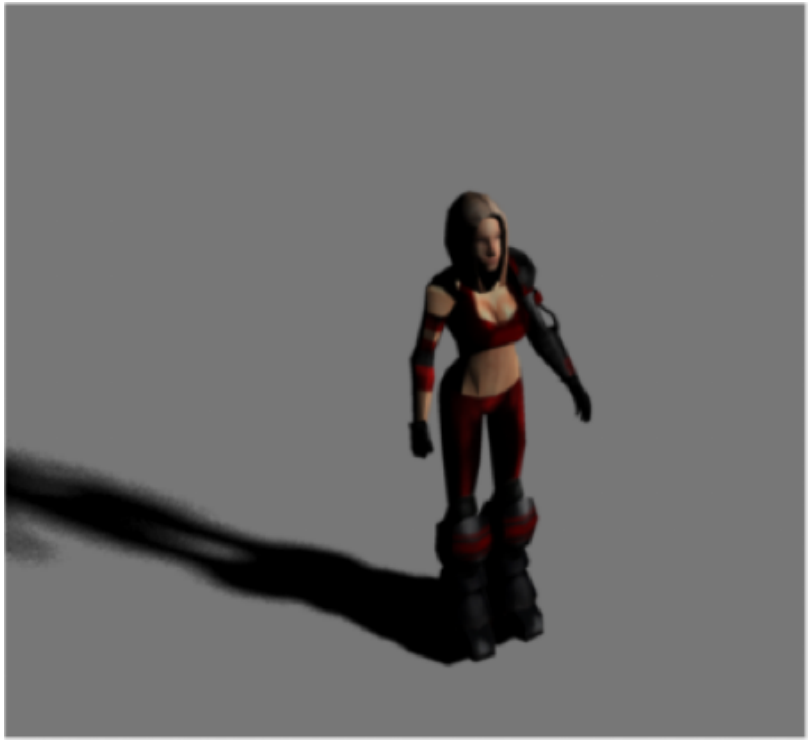
To characterize different approximations to the rendering equation, light sources can be divided into *direct* and *indirect*.

Direct sources represent lights coming from specific positions and directions (e.g. a lamp, the sun in an outdoor scene).

Indirect sources consider all the other types of illumination, mainly caused by light bounces and reflections among the surfaces.

Lights basics

With only *direct* sources, images become very dark and do not seem very realistic: if a point is not hit by any light, it appears black.



Lights basics

Projected shadows are created by the occlusion of direct light sources.

Lights basics

Indirect lighting adds realism, by making elements in sides not directly hit by light visible, but it requires a lot of computation, and it is not easy to implement in real time.

Rendering techniques

Several techniques have been introduced to approximate the rendering equation. We will briefly mention:

- *Scan-line rendering*
- *Ray casting*
- *Ray tracing*
- *Radiosity*
- *Montecarlo techniques*

} <- Next time

Scan-line rendering

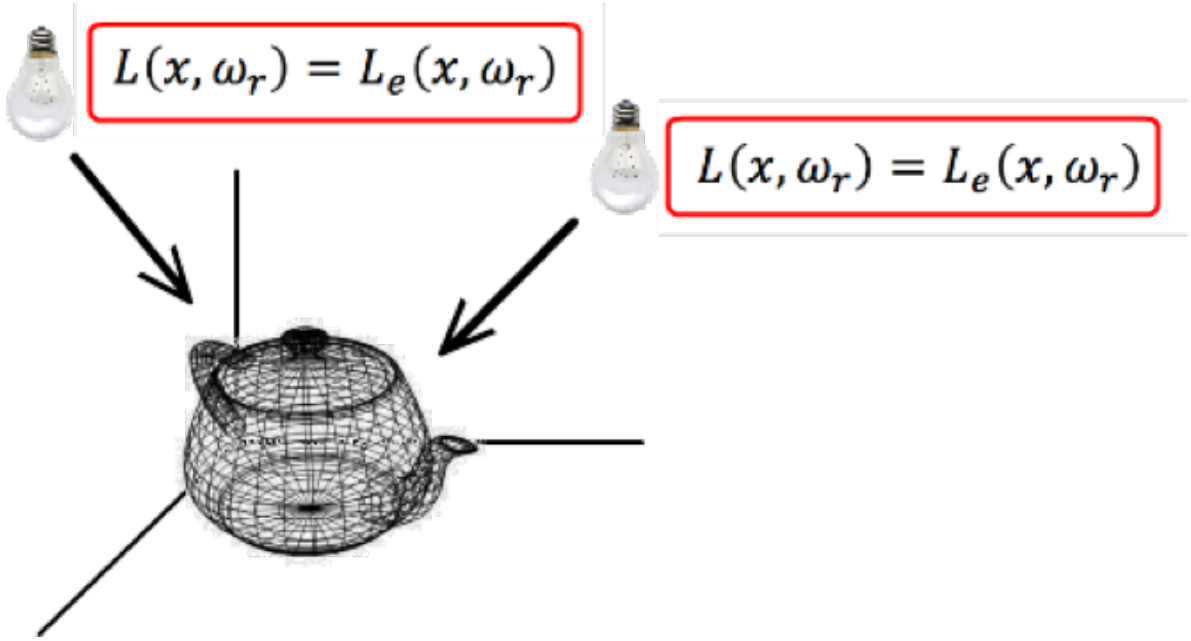
Scan-line rendering is the simplest approximation of the rendering equations.

It considers light sources and objects separately: the scene has a set of objects and a set of light sources.

No projected shadows or indirect lighting are produced.

Scan-line rendering

Lights are characterized by having only the emission term in the rendering equation: this however can vary in position and direction.



Scan-line rendering

Objects only reflect lights. They might emit some light, but they cannot illuminate other objects.

Inter-reflection between objects is not considered: the integral becomes a summation over all the light sources.

The geometric term is generally included in the BRDF:

$$L(x,\omega_r) = L_e(x,\omega_r) + \int L(y,\overrightarrow{yx}) f_r(x,\overrightarrow{yx},\omega_r) G(x,y) V(x,y) dy$$

With $f_{r'}(\ldots) = f_r(\ldots)G(\ldots)$

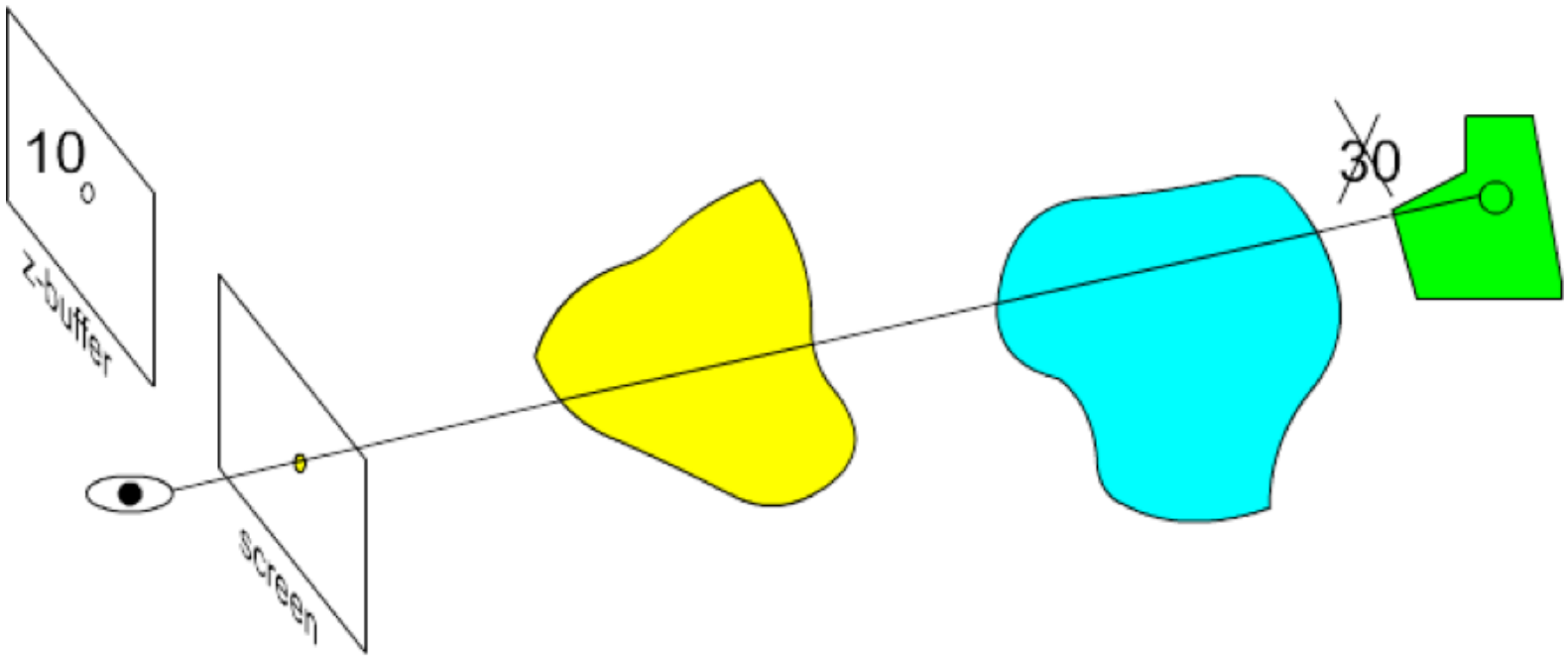
This is the emission term

$$L(x,\omega_r) = L_e(x,\omega_r) + \sum_l L_e\left(l,\overrightarrow{lx}\right) f_{r'}\left(x,\overrightarrow{lx},\omega_r\right)$$

The summation is over all the lights l in the scene

Scan-line rendering

Visibility is considered only with respect to the observer, by means of the z-buffer algorithm.



Scan-line rendering

Since term *V()* of the rendering equation is not considered for lights, scan-line rendering does not generate projected shadows.

Neither it does include light emitted by other objects in the scene, and thus it does not produces reflection, refraction or indirect illumination.

However it can easily consider different types of BDRF functions to describe the materials.

It is the main technique used for real-time rendering, and it will be investigated in the following lessons.

$$L(x,\omega_r) = L_e(x,\omega_r) + \sum_l L_e\left(l,\overrightarrow{lx}\right) f_r,\left(x,\overrightarrow{lx},\omega_r\right)$$

The BRDF is used to reproduce materials

Scan-line rendering

The pseudo-code of a scan-line rendering algorithm is the following:

```
01 for each object A in the scene do
02   for each visible (passes back-face culling and clipping) triangle t of A do
03     for each pixel x of t on screen do
04       if pixel x is visible (passes Z-buffer test) then
05         Set the pixel color C = L_e(x, ω_r) (emission and ambient light)
06         for each light l in the scene do
07           Set C = C + L(l,lx)*fr(x,lx, ω_r)
                              (contribution of light l to x)
08         end
09       end if
10     end
11   end
12 end
```

Scan-line rendering

When performing real-time rendering with Vulkan, the user implements the parts in red, either in the *Vertex* or in the *Fragment Shader*, while Vulkan does the parts in green.

```
01 for each object A in the scene do
02   for each visible (passes back-face culling and clipping) triangle t of A do
03     for each pixel x of t on screen do
04       if pixel x is visible (passes Z-buffer test) then
05         Set the pixel color C = L_e(x, ω_r)
06         for each light l in the scene do
07           Set C = C + L(l,lx)*fr(x,lx, ω_r)
08         end
09       end if
10     end
11   end
12 end
```

Ray casting

Ray casting is an extension of the scan-line rendering that computes the visibility function for all the point/light couples in the scene.

$$L(x,\omega_r) = L_e(x,\omega_r) + \int L(y,\overrightarrow{yx}) f_r(x,\overrightarrow{yx},\omega_r) G(x,y) V(x,y) dy$$

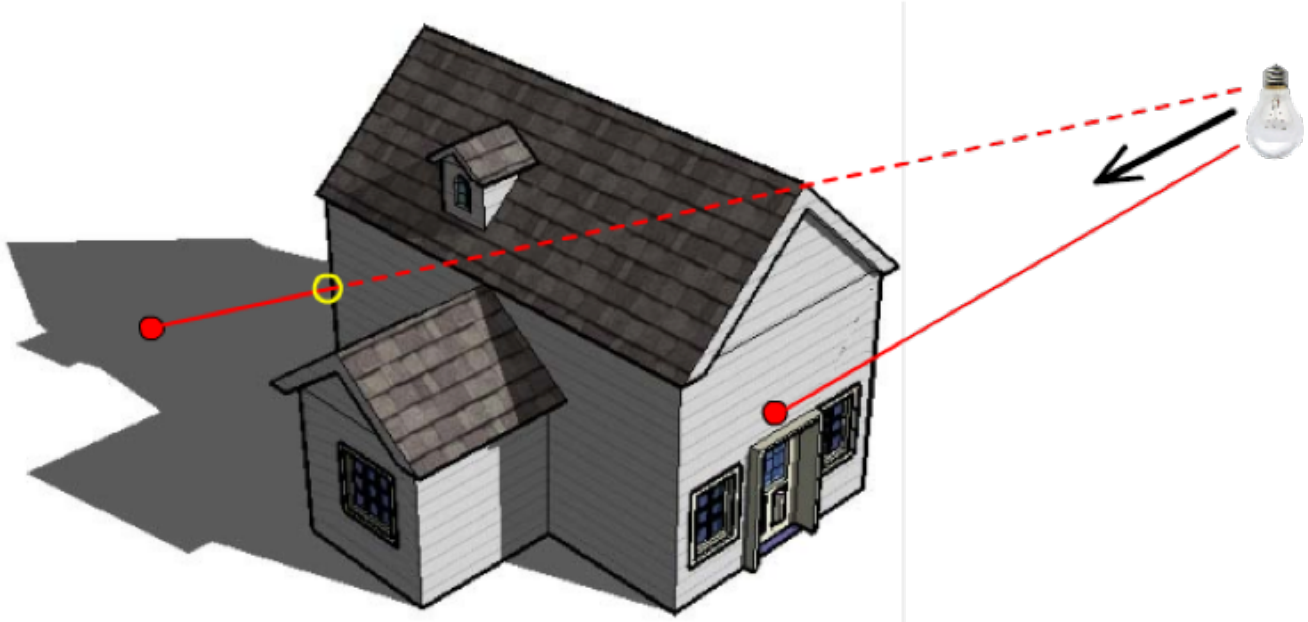
$$L(x,\omega_r) = L_e(x,\omega_r) + \sum_l L(l,\overrightarrow{lx}) f_r,(x,\overrightarrow{lx},\omega_r) V(x,l)$$

$L(x,\omega_r) = L_e(x,\omega_r) + \sum_l L_e(l,\overrightarrow{lx}) f_r,(x,\overrightarrow{lx},\omega_r)$ ← For comparison, this was the scan-line rendering equation

Ray casting

Ray casting allows including projected shadows.

The visibility function is computed by casting a ray that connects the considered points with each light source: if the ray intersects an object, the light is occluded and its effect is not considered in the rendering equation.



Ray casting

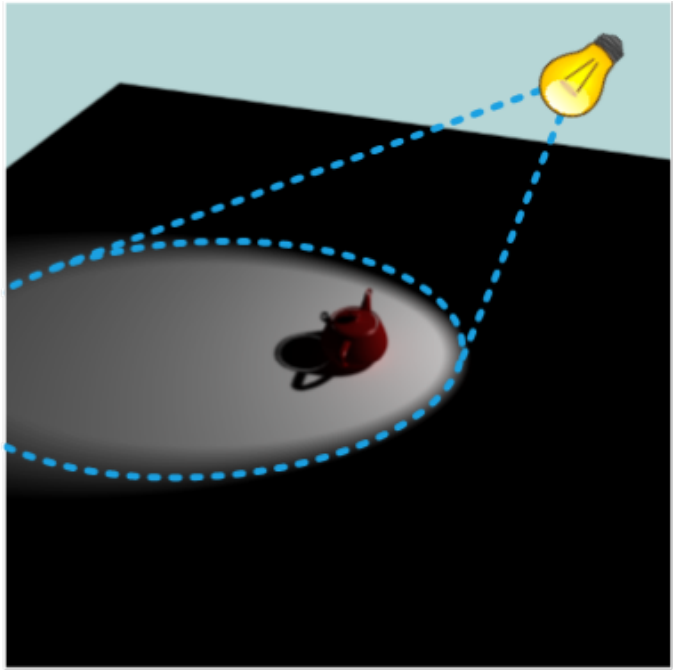
The pseudo-code of a ray casting rendering algorithm is the following:

```
01 for each object A in the scene do
02   for each visible (pass back-face culling and clipping) triangle t of A do
03     for each pixel x of t on screen do
04       if pixel x is visible (pass Z-buffer test) then
05         Set the pixel color C = 0
06         for each light l in the scene do
07           if light l is not occluded (ray-casting) then
08             Set C = C + L(l,lx)*fr(x,lx, ω_r )
09           end if
10         end
11       end if
12     end
13   end
14 end
```

Ray casting

One of the typical techniques to test if a light is occluded in real-time is using *Shadow Maps*:

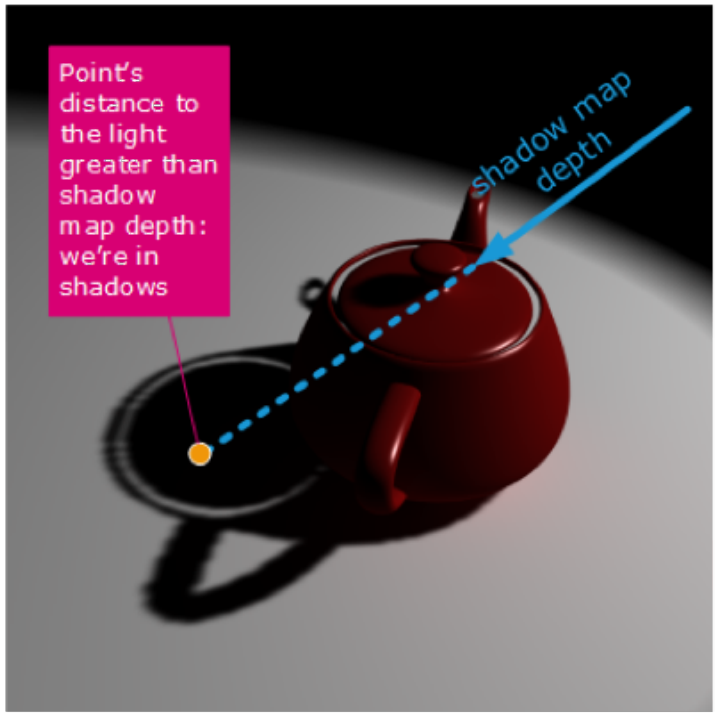
- an image rendered from the position of the light source, where each pixel represent the distance of the point from the light.



Ray casting

The information stored in the image is used to determine if a pixel is hit by the light or not.

The shader computes the distance from the light source, and compares it with the one in the map: if it greater, then the light is not considered.



Ray tracing

Ray tracing considers for each pixel also the light emitted by other objects in two specific directions: the *mirror reflection* and the *refraction* (for transparent objects).

$$L(x,\omega_r) = L_e(x,\omega_r) + \int L(y,\overrightarrow{yx}) f_r(x,\overrightarrow{yx},\omega_r) G(x,y) V(x,y) dy$$

$$L(x,\omega_r) = L_e(x,\omega_r) + \sum_l L\left(l,\overrightarrow{lx}\right) f_r,\left(x,\overrightarrow{lx},\omega_r\right) V(x,l) +$$

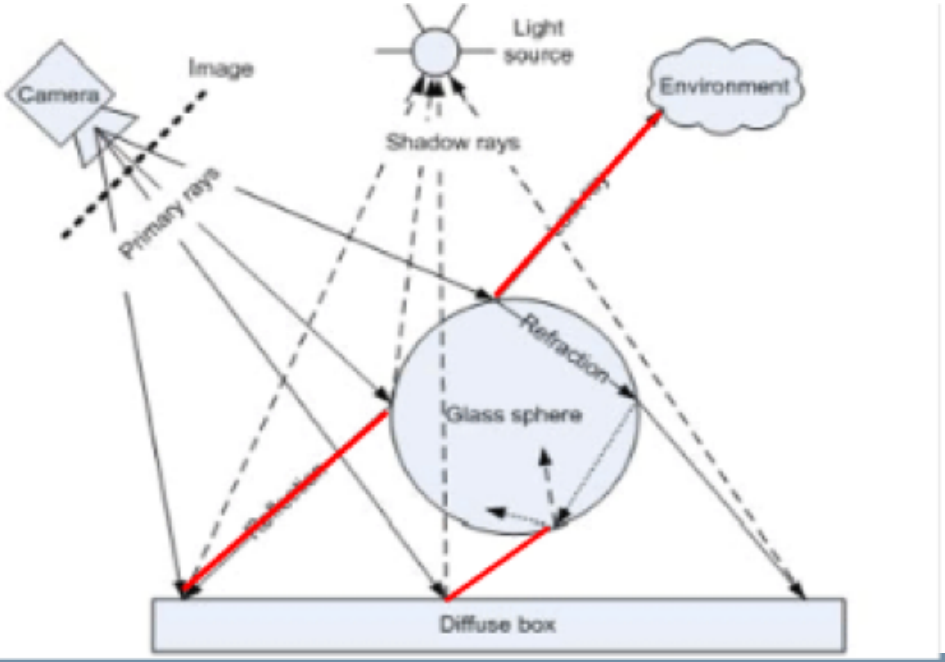
Reflection $L(r,\overrightarrow{rx}) f_r,(x,\overrightarrow{rx},\omega_r) V(x,r) +$

Refraction $L\left(t,\overrightarrow{tx'}\right) f_t,\left(x',\overrightarrow{tx'},\omega_r\right) V(x,t)$

Ray tracing

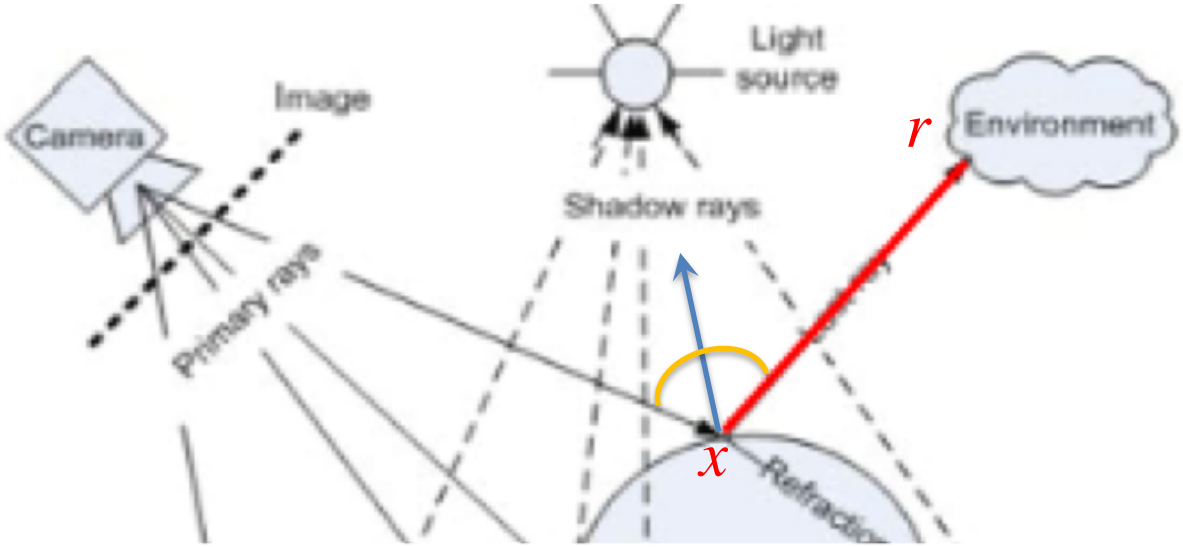
For reflection, this direction corresponds to the mirror direction with respect to the normal vector of the surface in the considered point.

This allows the reproduction of realistic perfect reflections.



Ray tracing

In particular, for each point x, the algorithm looks for the objects in direction $\overrightarrow{rx}$, the one that has the same outgoing angle as the incoming one.



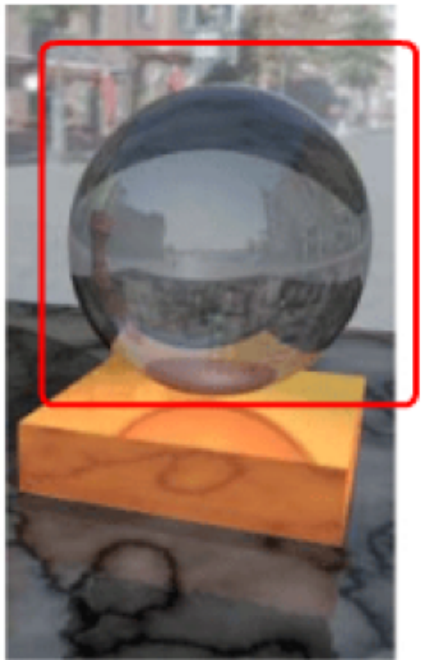
$$L(x,\omega_r) = L_e(x,\omega_r) + \sum_l L\left(l,\overrightarrow{lx}\right) f_{r'}\left(x,\overrightarrow{lx},\omega_r\right) V(x,l) +$$

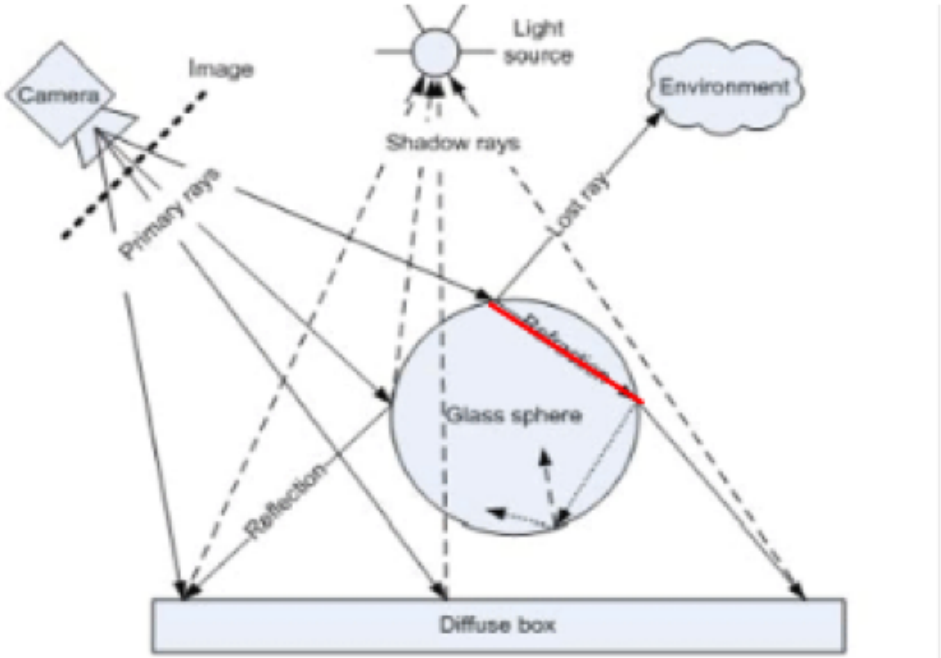
Reflection $L(r,\overrightarrow{rx}) f_{r'}(x,\overrightarrow{rx},\omega_r) V(x,r) +$

$$L\left(t,\overrightarrow{tx'}\right) f_{t'}\left(x',\overrightarrow{tx'},\omega_r\right) V(x,t)$$

Ray tracing

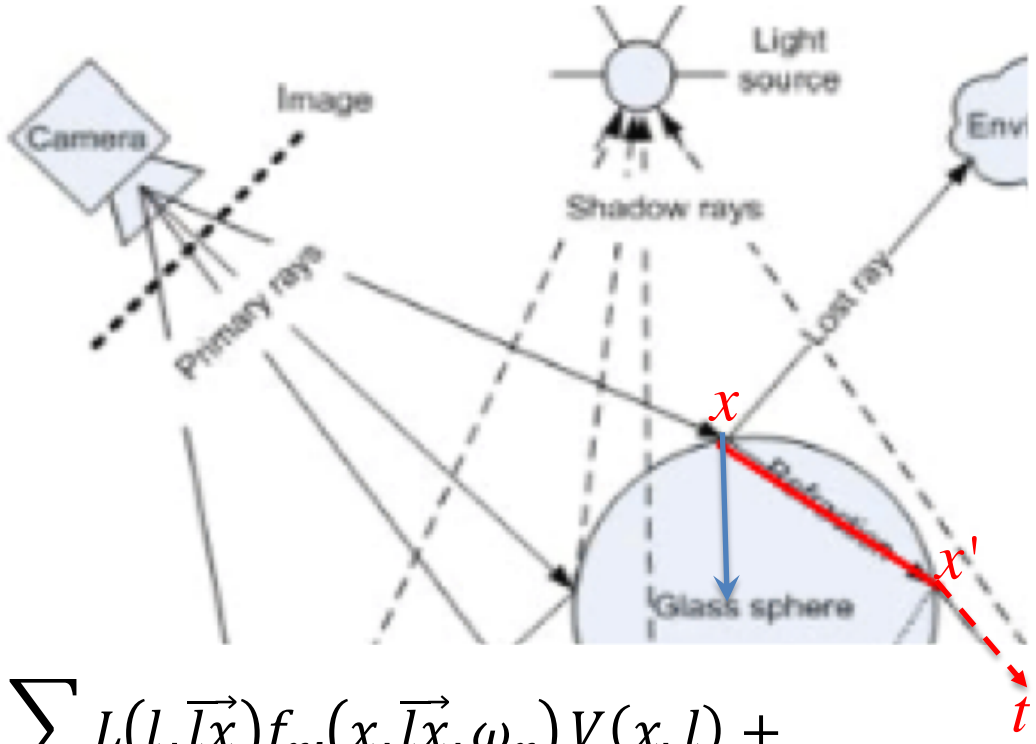
For refraction, the physical properties of the objects are emulated by considering the *index of refraction* for the material to determine the angle at which casting the refraction ray.





Ray tracing

In this case, for each point x, the algorithm searches for the objects in direction $\overrightarrow{tx}$, calculating point x' using the different refraction indices of the solids at the two sides of the surface.



$$L(x,\omega_r) = L_e(x,\omega_r) + \sum_l L\left(l,\overrightarrow{lx}\right) f_{r'}\left(x,\overrightarrow{lx},\omega_r\right) V(x,l) +$$

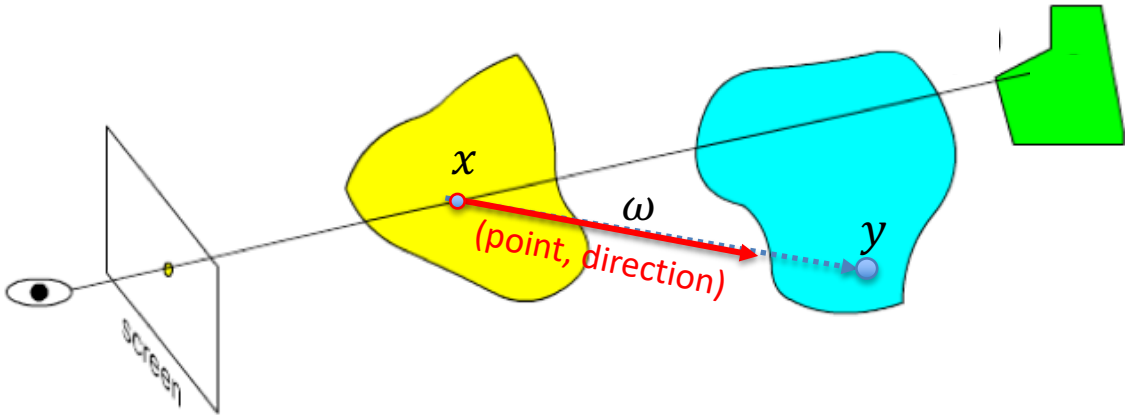
$$L(r,\overrightarrow{rx}) f_{r'}(x,\overrightarrow{rx},\omega_r) V(x,r) +$$

Refraction $\quad L\left(t,\overrightarrow{tx'}\right) f_{t'}\left(x',\overrightarrow{tx'},\omega_r\right) V(x,t)$

Ray tracing

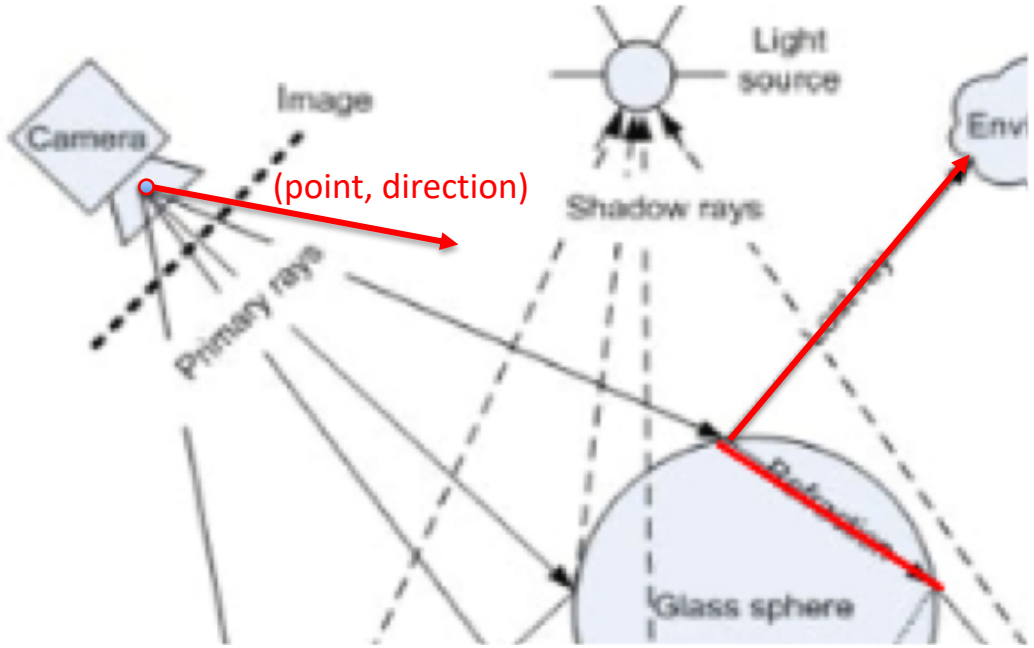
The algorithm relies on a ray-casting procedure that computes the colors seen from a given (point-in-space *x*, direction ω) couple.

The procedure searches the closest object to point *y* in the given direction ω, and applies the approximated rendering equation to compute $L(y, \omega)$.



Ray tracing

The algorithm starts considering each point on the projection plane (each pixel of the generated image), in the direction of the projection ray, and applies the ray-casting procedure to it.



Ray tracing

For considering the reflection and refraction part of each pixel, the procedure is called recursively with different points and direction rays.

$$L(x,\omega_r) = L_e(x,\omega_r) + \sum_l L\left(l,\overrightarrow{lx}\right) f_r,\left(x,\overrightarrow{lx},\omega_r\right) V(x,l) +$$

$$L(r,\overrightarrow{rx}) f_r,(x,\overrightarrow{rx},\omega_r) V(x,r) +$$

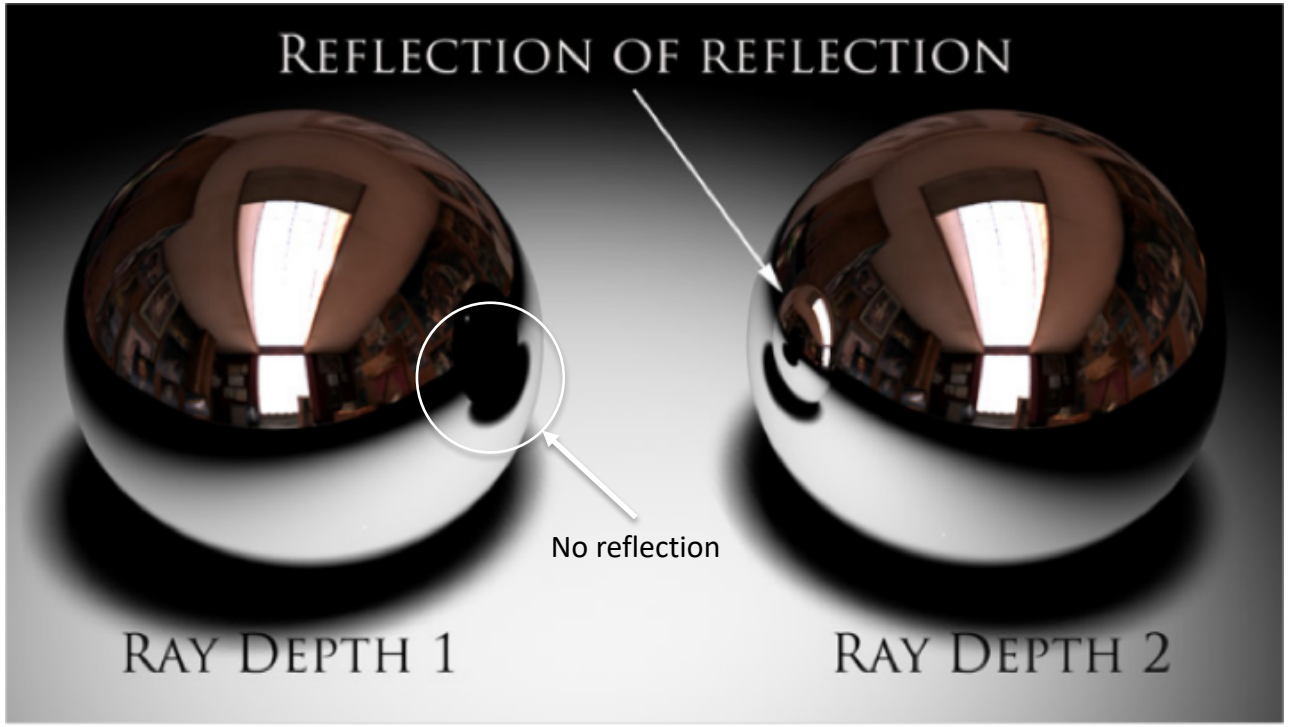
$$L\left(t,\overrightarrow{tx'}\right) f_t,\left(x',\overrightarrow{tx'},\omega_r\right) V(x,t)$$

Recursion to compute reflection:
r is the point seen in the reflection.

Recursion to compute refraction:
t is the point seen through the surface by the refraction
x' is the point on the other side refracted to x

Ray tracing

The recursion is repeated up to a given number of bounces, called the *ray depth*.



Ray tracing

The pseudo-code of a ray tracing rendering algorithm is the following:

```
01 for each pixel x on screen do
02   Compute color casting a ray from x according the projection
03 end
```

The ray-casting procedure, is the hearth of the technique:

```
01 Determine the point q of the closest object w.r.t. the ray
02 Set the pixel color C = 0
03 for each light l in the scene do
04   if light l is not occluded (ray-casting) then
05     Set C = C + contribution of light l to q
06   end if
07 end
08 Set C = C + reflection contribution (recursion)
09 Set C = C + refraction contribution (recursion)
```

Ray tracing

In a Ray Tracing pipeline, the *RayGen* Shader, and the *Closest Hit Shader* perform the tasks highlighted before.

```
01 for each pixel x on screen do
02   Compute color casting a ray from x according the projection
03 end
```

Ray Generation Shader

```
01 Determine the point q of the closest object w.r.t. the ray
02 Set the pixel color C = 0
03 for each light l in the scene do
04   if light l is not occluded (ray-casting) then
05     Set C = C + contribution of light l to q
06   end if
07 end
08 Set C = C + reflection contribution (recursion)
09 Set C = C + refraction contribution (recursion)
```

Closest hit Shader

Ray tracing

The number of traced rays potentially doubles at every step: this can significantly increase the rendering times. Moreover, it requires computation of the closest intersection using the *acceleration structure* instead of the Z-buffer (which is more computation intensive).

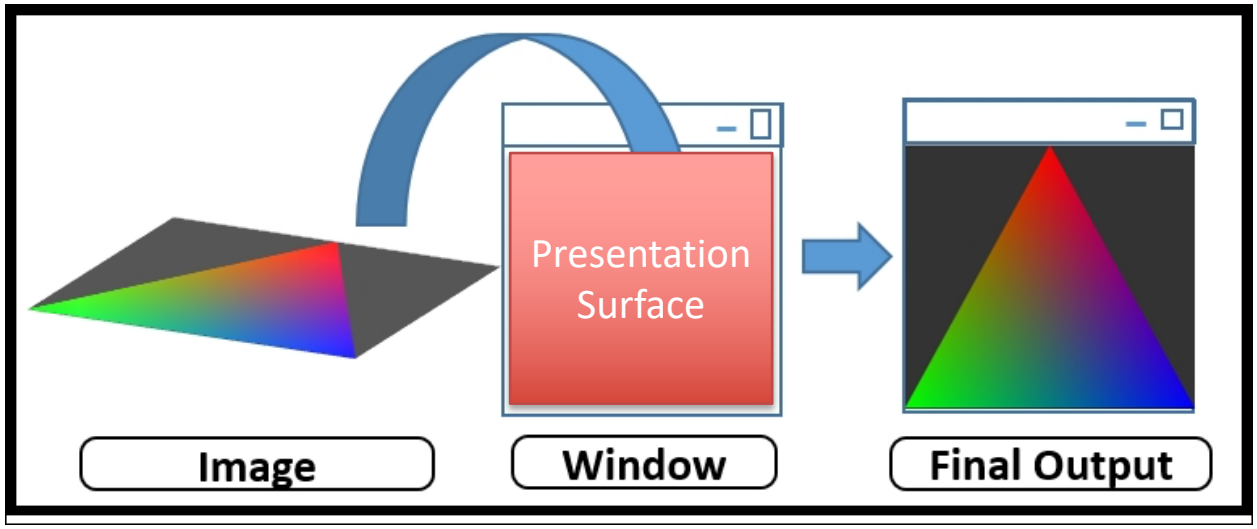
Ray tracing allows including mirror reflection and transparency with refraction: this can be used to realistically reproduce glass, fluid, shiny metals and many other objects.

However, ray tracing is not able to simulate indirect lighting or consider glossy reflections, limiting the level of achievable realism.

Presentation surface

Last time we have seen how to create the application Window in a system independent way using GLFW.

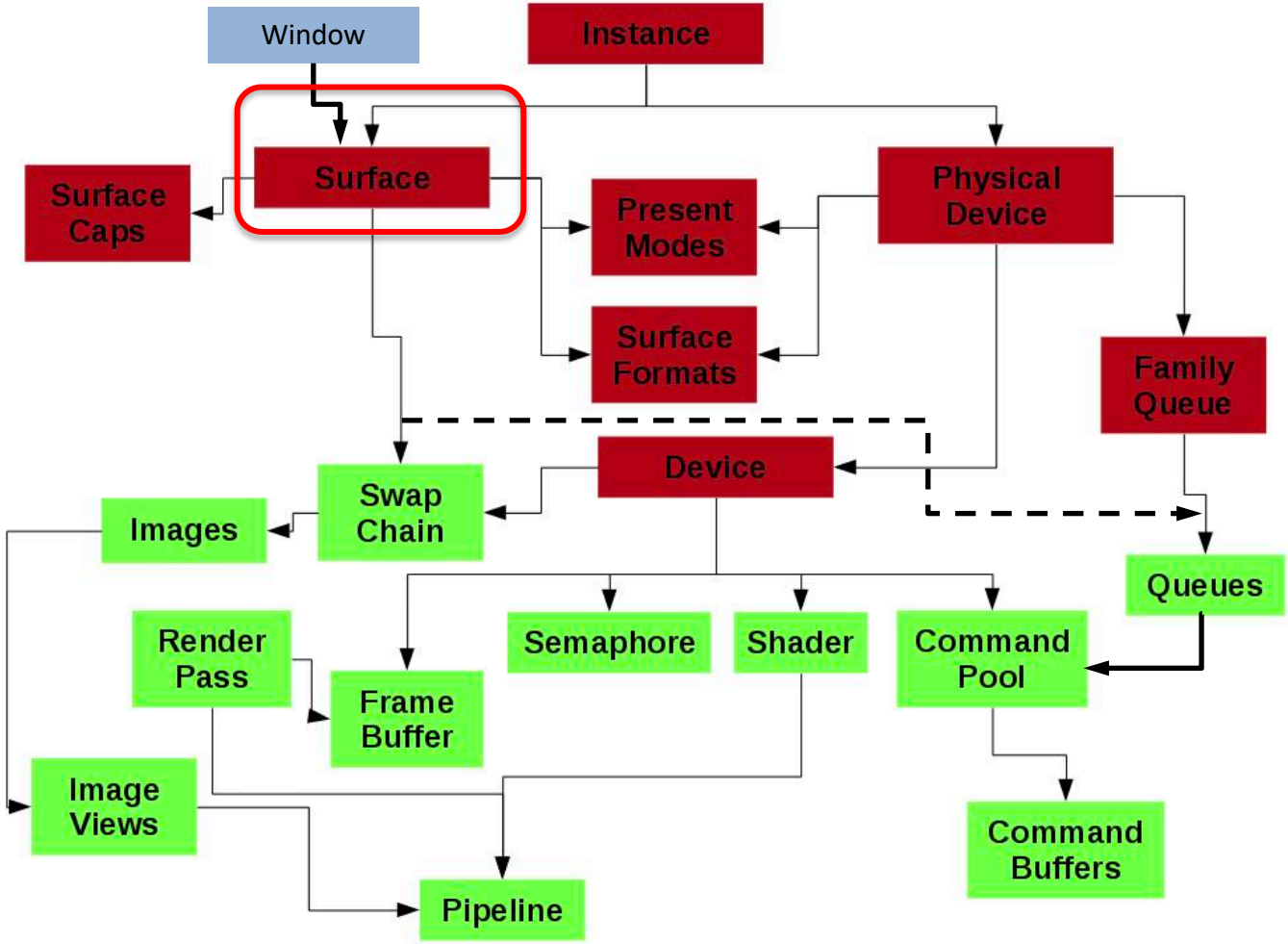
The next step is creating the *Presentation Surface*.



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);
```

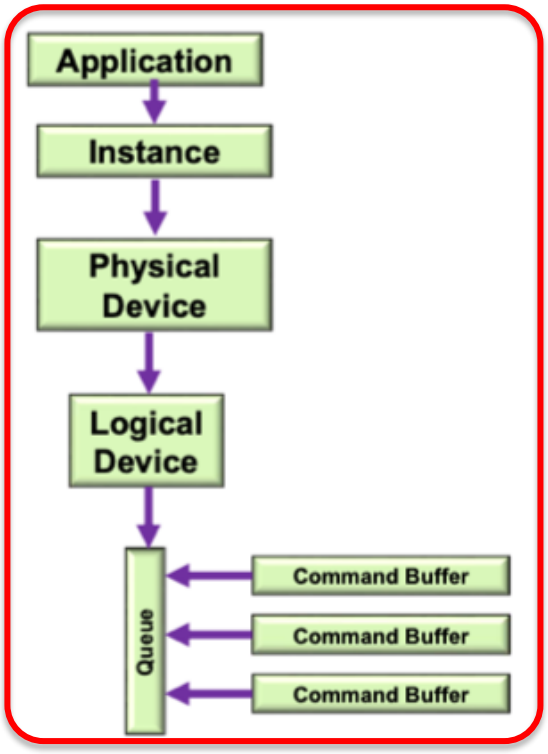
Presentation surface

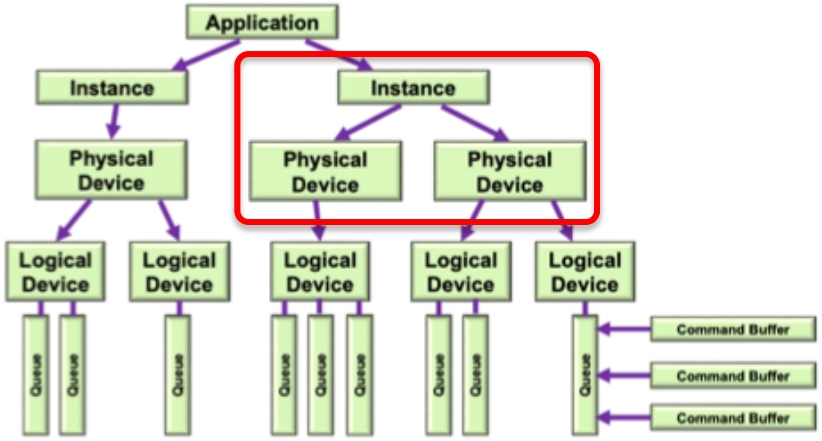
Presentation surface needs to be released at the end of the application, before destroying the instance and the window, using the `vkDestroySurfaceKHR()` command.

```
// 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.



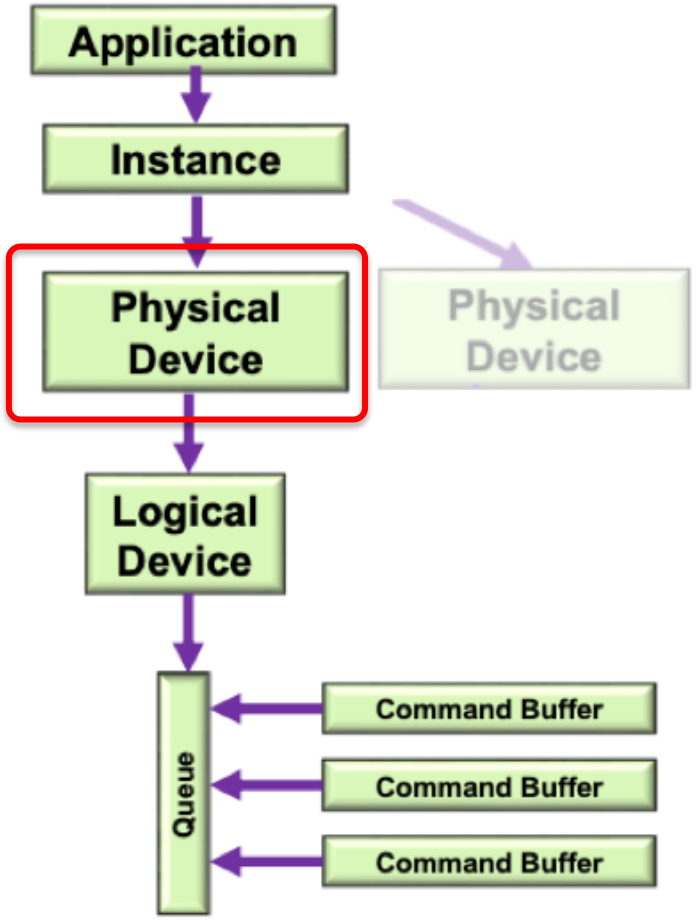


Vulkan Physical Devices

We need a procedure to select the most appropriate one, in a system independent way!

This can achieved:

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



Vulkan Physical Devices

Following the same two-calls pattern presented last time to find the available extensions, Physical devices can be enumerated with the `vkEnumeratePhysicalDevices()` command, which returns a set of `VkPhysicalDevices`.

```
// Enumerate devices
uint32_t deviceCount = 0;
result = vkEnumeratePhysicalDevices(instance, &deviceCount, nullptr);
if(result != VK_SUCCESS || deviceCount <= 0) {
     throw std::runtime_error("failed to find GPUs with Vulkan Support!");
}

std::vector<VkPhysicalDevice> devices(deviceCount);
result = vkEnumeratePhysicalDevices(instance, &deviceCount, devices.data());
if(result != VK_SUCCESS) {
     throw std::runtime_error("Could not enumerate devices");
}

for (const auto& physicalDevice : devices) {
     …
}
```

Vulkan Physical Devices

The application can the cycle through each device, to determine whether is suitable for graphics, and assign it a score to pick up the best one available.

```
// Enumerate devices
uint32_t deviceCount = 0;
result = vkEnumeratePhysicalDevices(instance, &deviceCount, nullptr);
if(result != VK_SUCCESS || deviceCount <= 0) {
     throw std::runtime_error("failed to find GPUs with Vulkan Support!");
}

std::vector<VkPhysicalDevice> devices(deviceCount);
result = vkEnumeratePhysicalDevices(instance, &deviceCount, devices.data());
if(result != VK_SUCCESS) {
     throw std::runtime_error("Could not enumerate devices");
}

for (const auto& physicalDevice : devices) {
     …
}
```

Vulkan Physical Devices

Each device is characterized by:

- *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.

Device Properties

Device Properties can be queried with the `vkGetPhyiscalDeviceProperties()` command, which fills a `VkPhysicalDeviceProperties` structure.

```
// Show device properties
VkPhysicalDeviceProperties deviceProperties;
vkGetPhysicalDeviceProperties(physicalDevice, &deviceProperties);

std::cout <<  "\tAPI version: 0x" << std::hex << deviceProperties.apiVersion << "\n";
std::cout << "\tDriver version: 0x" << std::hex << deviceProperties.driverVersion << "\n";
std::cout << "\tVendor ID: 0x" << std::hex << deviceProperties.vendorID << "\n";
std::cout << "\tDevice ID: 0x" << std::hex << deviceProperties.deviceID << "\n";
std::cout << "\tPhysical Device Type: " << deviceProperties.deviceType << "\t";
if(deviceProperties.deviceType == VK_PHYSICAL_DEVICE_TYPE_DISCRETE_GPU )
     std::cout << " (Discrete GPU)\n";
if(deviceProperties.deviceType == VK_PHYSICAL_DEVICE_TYPE_INTEGRATED_GPU )
     std::cout << " (Integrated GPU)\n";
if(deviceProperties.deviceType == VK_PHYSICAL_DEVICE_TYPE_VIRTUAL_GPU )
     std::cout << " (Virtual GPU)\n";
if(deviceProperties.deviceType == VK_PHYSICAL_DEVICE_TYPE_CPU )
     std::cout <<  " (CPU)\n";
std::cout << "\tDevice Name: " << deviceProperties.deviceName << "\n";
```

Device Features

Similarly, Device Features are queried with the `vkGetPhyiscalDeviceFeatures()` command, which fills a `VkPhysicalDeviceFeatures` structure.

```
VkPhysicalDeviceFeatures deviceFeatures;
vkGetPhysicalDeviceFeatures(physicalDevice, &deviceFeatures);

std::cout << "\n\tPhysical Device Features:\n";
std::cout << "\t\tgeometryShader = " << deviceFeatures.geometryShader << "\n";
std::cout << "\t\ttessellationShader = " << deviceFeatures.tessellationShader << "\n";
std::cout << "\t\tmultiDrawIndirect = " << deviceFeatures.multiDrawIndirect << "\n";
std::cout << "\t\twideLines = " << deviceFeatures.wideLines << "\n";
std::cout << "\t\tlargePoints = " << deviceFeatures.largePoints << "\n";
std::cout << "\t\tmultiViewport = " << deviceFeatures.multiViewport << "\n";
std::cout << "\t\tocclusionQueryPrecise = " << deviceFeatures.occlusionQueryPrecise << "\n";
std::cout << "\t\tpipelineStatisticsQuery = " << deviceFeatures.pipelineStatisticsQuery << "\n";
std::cout << "\t\tshaderFloat64 = " << deviceFeatures.shaderFloat64 << "\n";
std::cout << "\t\tshaderInt64 = " << deviceFeatures.shaderInt64 << "\n";
std::cout << "\t\tshaderInt16 = " << deviceFeatures.shaderInt16 << "\n";
```

Available Properties and Features

In the previous slides, only the main features and properties have been shows.

The *Properties* structure, 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

GPU Memory

Memory types and heaps can be inspected with the `vkGetPhyiscalDeviceMemoryProperties()` command, which fills a `VkPhysicalDeviceMemoryPropreties` structure.

```
VkPhysicalDeviceMemoryProperties vpdmp;
vkGetPhysicalDeviceMemoryProperties(physicalDevice, &vpdmp);

std::cout << "\n\tMemory Types: " << vpdmp.memoryTypeCount << "\n";
for(unsigned int i = 0; i < vpdmp.memoryTypeCount; i++) {
     VkMemoryType vmt = vpdmp.memoryTypes[i];
     …
}

std::cout << "\n\tMemory Heaps: " << vpdmp.memoryHeapCount << "\n";
for(unsigned int i = 0; i < vpdmp.memoryHeapCount; i++ ) {
     VkMemoryHeap vmh = vpdmp.memoryHeaps[i];
     …
}
```

GPU Memory

The command returns two arrays of `VkMemoryType` and `VkMemoryHeap` properties, containing the available classes of memory and their size.

```
VkPhysicalDeviceMemoryProperties vpdmp;
vkGetPhysicalDeviceMemoryProperties(physicalDevice, &vpdmp);

std::cout << "\n\tMemory Types: " << vpdmp.memoryTypeCount << "\n";
for(unsigned int i = 0; i < vpdmp.memoryTypeCount; i++) {
     VkMemoryType vmt = vpdmp.memoryTypes[i];
     …
}

std::cout << "\n\tMemory Heaps: " << vpdmp.memoryHeapCount << "\n";
for(unsigned int i = 0; i < vpdmp.memoryHeapCount; i++ ) {
     VkMemoryHeap vmh = vpdmp.memoryHeaps[i];
     …
}
```

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";
}
```

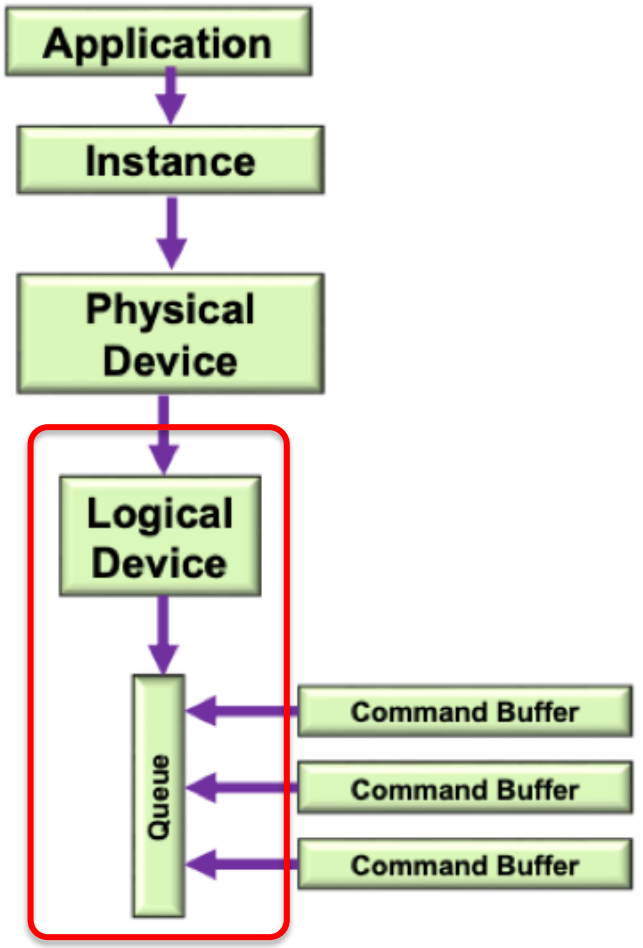
https://www.khronos.org/registry/vulkan/specs/1.3-extensions/man/html/VkMemoryHeapFlagBits.html

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 two-calls patterns to the function `vkGetPhysicalDeviceQueueFamilyProperties()` into an array of `VkQueueFamilyProperties` structures.

```
// 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

The array contains one element per supported Queue family, each one showing how many queues it can create, and the type of operations it supports.

```
std::optional<uint32_t> aQueueWithGraphicsCapability;
std::optional<uint32_t> aQueueWithPresentationCapability;
for(unsigned int i = 0; i < queueFamCount; i++ ) {
     std::cout << "\t\QqueueCount = " << vqueues[i].queueCount << "; ";
     if((queues[i].queueFlags & VK_QUEUE_GRAPHICS_BIT) != 0) {
          std::cout << " Graphics";
          aQueueWithGraphicsCapability = i;
     }
     if((queues[i].queueFlags & VK_QUEUE_COMPUTE_BIT) != 0)
          std::cout << " Compute";
     if((queues[i].queueFlags & VK_QUEUE_TRANSFER_BIT) != 0)
          std::cout << " Transfer";
     VkBool32 presentSupport = false;
     vkGetPhysicalDeviceSurfaceSupportKHR(physicalDevice, i, surface, &presentSupport);
     if(presentSupport) {
          std::cout << " Presentation";
          aQueueWithPresentationCapability = i;
     }
```

Queue families

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

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

We will not consider the Sparse Memory, and we will focus on the Graphics operation.

Presentation support, since it is connected with the surface, must be determined in a different way.

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

Queue families

In particular, it should be determined with the `vkGetPhisicalDeviceSupportKHR()` command.

```
std::optional<uint32_t> aQueueWithGraphicsCapability;
std::optional<uint32_t> aQueueWithPresentationCapability;
for(unsigned int i = 0; i < queueFamCount; i++ ) {
     std::cout << "\t\QqueueCount = " << vqueues[i].queueCount << "; ";
     if((queues[i].queueFlags & VK_QUEUE_GRAPHICS_BIT) != 0) {
          std::cout << " Graphics";
          aQueueWithGraphicsCapability = i;
     }
     if((queues[i].queueFlags & VK_QUEUE_COMPUTE_BIT) != 0)
          std::cout << " Compute";
     if((queues[i].queueFlags & VK_QUEUE_TRANSFER_BIT) != 0)
          std::cout << " Transfer";
     VkBool32 presentSupport = false;
     vkGetPhysicalDeviceSurfaceSupportKHR(physicalDevice, i, surface, &presentSupport);
     if(presentSupport) {
          std::cout << " Presentation";
          aQueueWithPresentationCapability = i;
     }
```

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.

Queue families

Queue Families are identified by their index in the corresponding data structure. The application must remember at least one index for a *Graphic* and a *Presentation* queue (which might be different or identical)

```
std::optional<uint32_t> aQueueWithGraphicsCapability;
std::optional<uint32_t> aQueueWithPresentationCapability;
for(unsigned int i = 0; i < queueFamCount; i++ ) {
     std::cout << "\t\QqueueCount = " << vqueues[i].queueCount << "; ";
     if((queues[i].queueFlags & VK_QUEUE_GRAPHICS_BIT) != 0) {
          std::cout << " Graphics";
          aQueueWithGraphicsCapability = i;
     }
     if((queues[i].queueFlags & VK_QUEUE_COMPUTE_BIT) != 0)
          std::cout << " Compute";
     if((queues[i].queueFlags & VK_QUEUE_TRANSFER_BIT) != 0)
          std::cout << " Transfer";
     VkBool32 presentSupport = false;
     vkGetPhysicalDeviceSurfaceSupportKHR(physicalDevice, i, surface, &presentSupport);
     if(presentSupport) {
          std::cout << " Presentation";
          aQueueWithPresentationCapability = i;
     }
}
```

Queue families

To make sure that at least one queue Family per type exist, the `std::optional` template might be used. In this way, the application can easily check if no value has been found for a given family.

```
std::optional<uint32_t> aQueueWithGraphicsCapability;
std::optional<uint32_t> aQueueWithPresentationCapability;
for(unsigned int i = 0; i < queueFamCount; i++ ) {
     std::cout << "\t\QqueueCount = " << vqueues[i].queueCount << "; ";
     if((queues[i].queueFlags & VK_QUEUE_GRAPHICS_BIT) != 0) {
          std::cout << " Graphics";
          aQueueWithGraphicsCapability = i;
     }
     if((queues[i].queueFlags & VK_QUEUE_COMPUTE_BIT) != 0)
          std::cout << " Compute";
     if((queues[i].queueFlags & VK_QUEUE_TRANSFER_BIT) != 0)
          std::cout << " Transfer";
     VkBool32 presentSupport = false;
     vkGetPhysicalDeviceSurfaceSupportKHR(physicalDevice, i, surface, &presentSupport);
     if(presentSupport) {
          std::cout << " Presentation";
          aQueueWithPresentationCapability = i;
     }
}
```

Logical device creation

When the Physical Device has been selected, we can use it to create a Logical device. In the selection process, we must at least check if there exists queues supporting graphics and presentation, exploiting the `has_value()` method of the `std::optional` objects.

```
if(aQueu}WithGraphicsCapability.has_value() &&
   aQueueWithPresentationCapability.has_value()) {
    std::cout << "\tGraphic queue family selected: " <<
                                aQueueWithGraphicsCapability.value() << "\n";
    std::cout << "\tPresentation queue family selected: " <<
                                aQueueWithPresentationCapability.value() << "\n";
     …
{
```

Logical device creation

Logical devices are created together with their queues in using the `vkCreateDevice()` command, starting from a Physical Device. Following the standard Vulkan conventions, all parameters are inserted in a `VkDeviceCreateInfo` structure.

```
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

Device features are enabled using an object with the same format as the one of used for querying the available features.

Extensions and *Debug Layers* (we will return on this later), can be enabled as well. Here none has been used.

```
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

The queues that must be created, are specified in the `pQueueCreateInfo` and `pQueueCreateInfoCount` fields.

As outlined before, these can be either one single or two different queues for graphics and presentation.

```
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

A C++ standard template object `set` can be used to filter the unique type of queues used in the application, to make it support both cases.

```
std::vector<VkDeviceQueueCreateInfo> queueCreateInfos;
std::set<uint32_t> uniqueQueueFamilies =
          {aQueueWithGraphicsCapability.value(),
           aQueueWithPresentationCapability.value()};

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);
}
```

Logical device creation

Set elements can then be used to fill the array of `VkDeviceQueueCreateInfo` objects, one per different requested queue family.

```
std::vector<VkDeviceQueueCreateInfo> queueCreateInfos;
std::set<uint32_t> uniqueQueueFamilies =
          {aQueueWithGraphicsCapability.value(),
           aQueueWithPresentationCapability.value()};

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);
}
```

Logical device creation

For each requested queue, both the family index and the count (here always one) is inserted in the structure.

```
std::vector<VkDeviceQueueCreateInfo> queueCreateInfos;
std::set<uint32_t> uniqueQueueFamilies =
          {aQueueWithGraphicsCapability.value(),
           aQueueWithPresentationCapability.value()};

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);
}
```

Logical device creation

To help Vulkan schedule events, each queue can also have a floating point priority, a value between 0 and 1. Since more queues from the same family might be created with a single command, priorities must be passed as a pointer to an array (here not necessary because we have a single element).

```
std::vector<VkDeviceQueueCreateInfo> queueCreateInfos;
std::set<uint32_t> uniqueQueueFamilies =
          {aQueueWithGraphicsCapability.value(),
           aQueueWithPresentationCapability.value()};

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);
}
```

Logical device creation

On success, the `VkDevice` structure containing the Device Handle passed to the command is filled.

Following the Vulkan notation, Logical Devices will be simply referred to as *Devices*.

```
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.

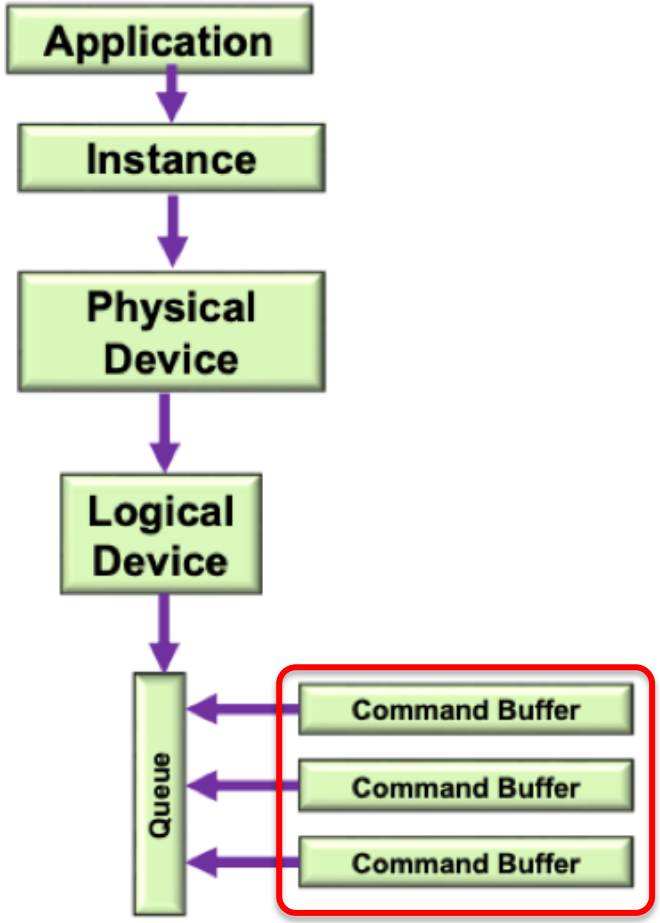
```
vkDestroyDevice(device, nullptr);

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

Command buffer creation

Once queues has 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. On success, the handle to the command pool fills the `VkCommandPool` argument.

```
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

Two types of command buffers are available: *primary* and *secondary*. The purpose is to allow the creation of subroutine that can be called from different command buffers. In this course we will only consider primary command buffers.

```
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!");
}
```