
# Introduction to Vulkan applications and The Rendering Equation

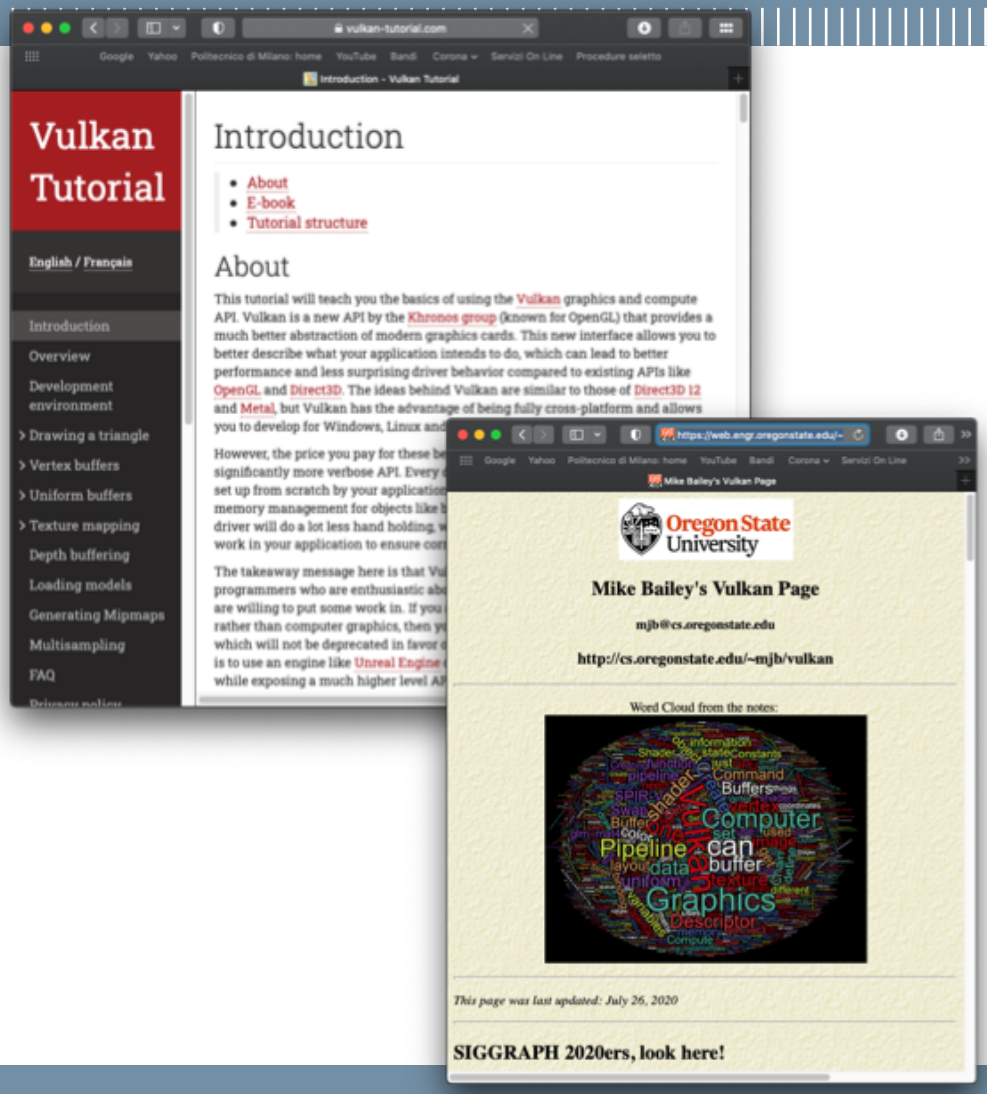
# Introduction to Vulkan Applications

In this course we will follow 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

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 GPUs at the same time

Vulkan has been created to allow the users exploiting 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 for allowing the application to interface with the user. Vulkan aims at supporting them all!

# Skeleton of a Vulkan application

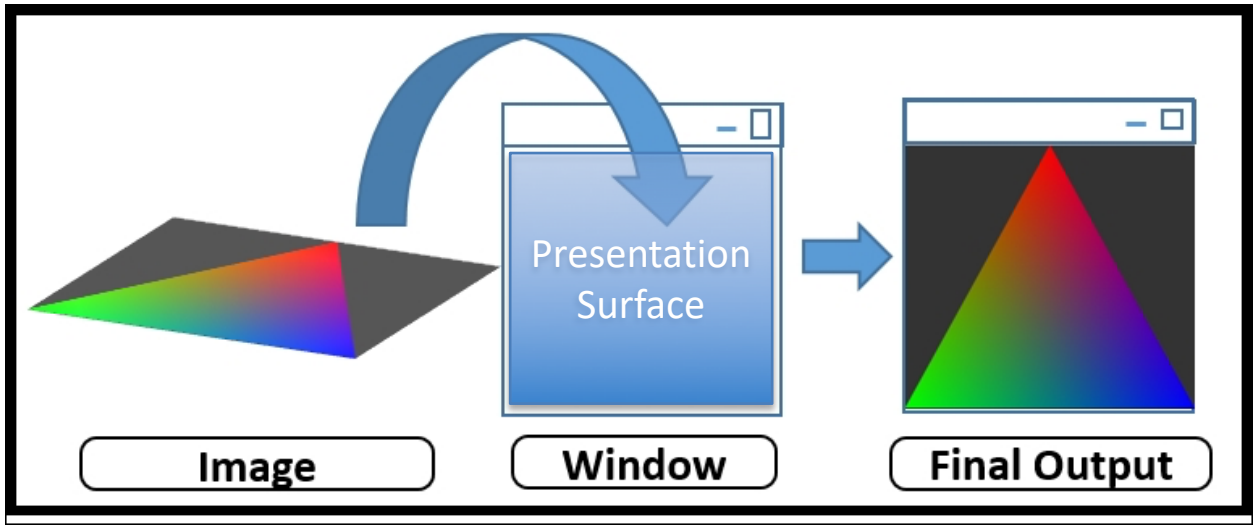
A typical Vulkan application has the following skeleton:

```
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*.

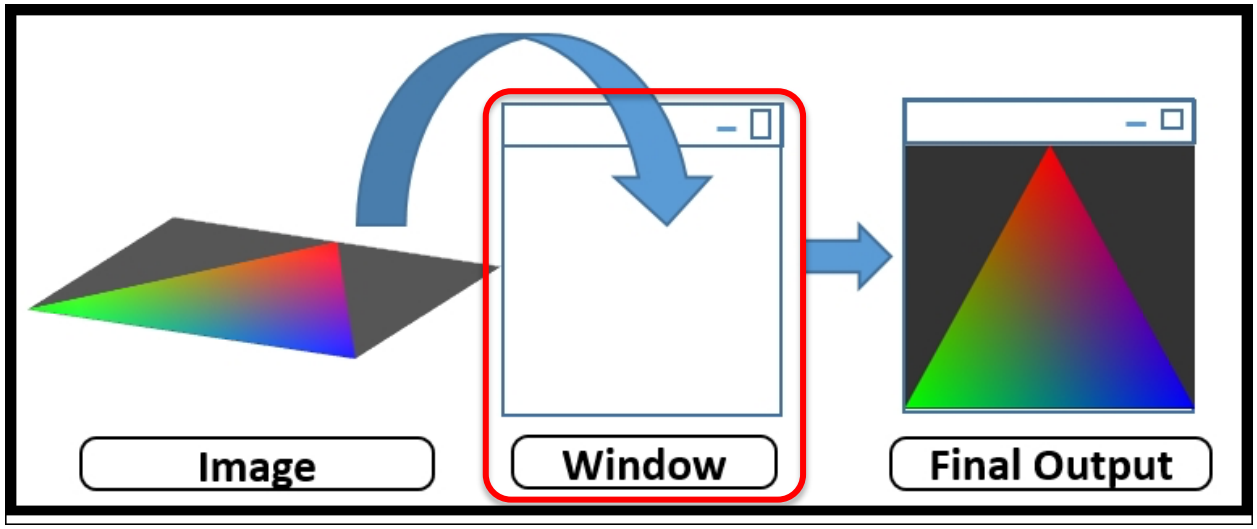
In order to work properly, a Vulkan application should acquire a proper presentation surface from the O.S. This will be system dependent, and we will return on this later.



# 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

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

```
391
392     void initWindow() {
393         glfwInit();
394
395         glfwWindowHint(GLFW_CLIENT_API, GLFW_NO_API);
396
397         window = glfwCreateWindow(WIDTH, HEIGHT, "Vulkan", nullptr, nullptr);
398
399
400     }
401
402
403
404
405
406
407
```

# 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 *OpenGL*, we must set the `GLFW_CLIENT_API` property to `GLFW_NO_API` to use Vulkan .

```
391
392     void initWindow() {
393         glfwInit();
394
395         glfwWindowHint(GLFW_CLIENT_API, GLFW_NO_API);
396
397         window = glfwCreateWindow(WIDTH, HEIGHT, "Vulkan", nullptr, nullptr);
398
399
400     }
401
402
403
404
405
406
407
```

# 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`.

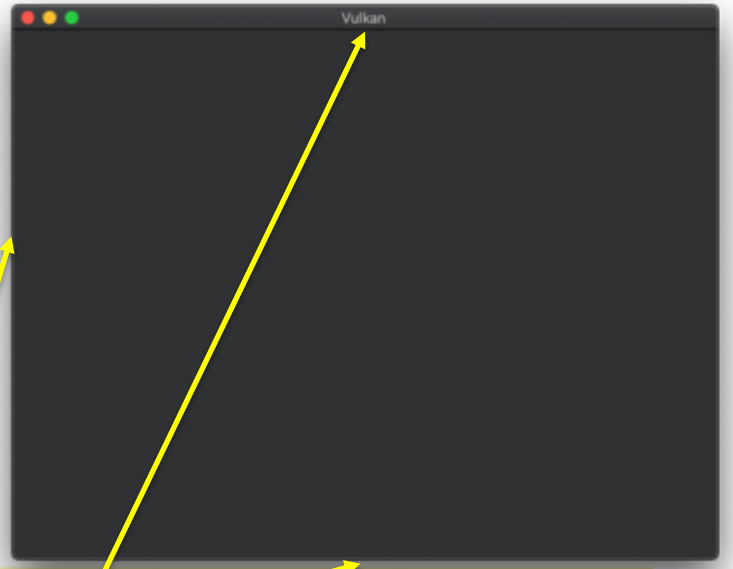
This hint only has an effect on platforms where screen coordinates and pixels always map 1:1 such as Windows and X11. On platforms like macOS the resolution of the framebuffer is changed independently of the window size.

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

# GLFW window creation

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

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.



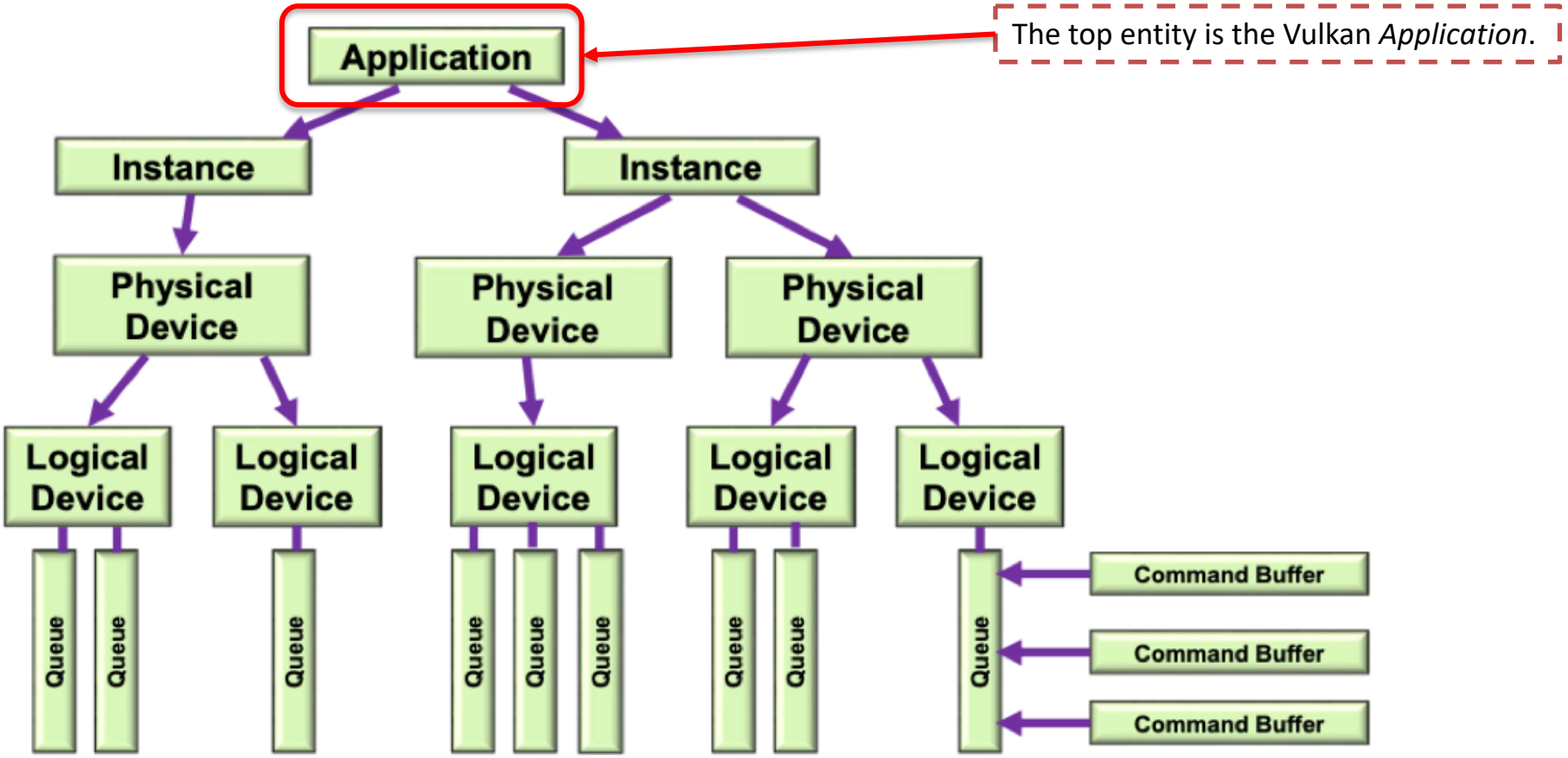

```
391
392     void initWindow() {
393         glfwInit();
394
395         glfwWindowHint(GLFW_CLIENT_API, GLFW_NO_API);
396
397         window = glfwCreateWindow(WIDTH, HEIGHT, "Vulkan", nullptr, nullptr);
398
399
400     }
```

# Vulkan initialization

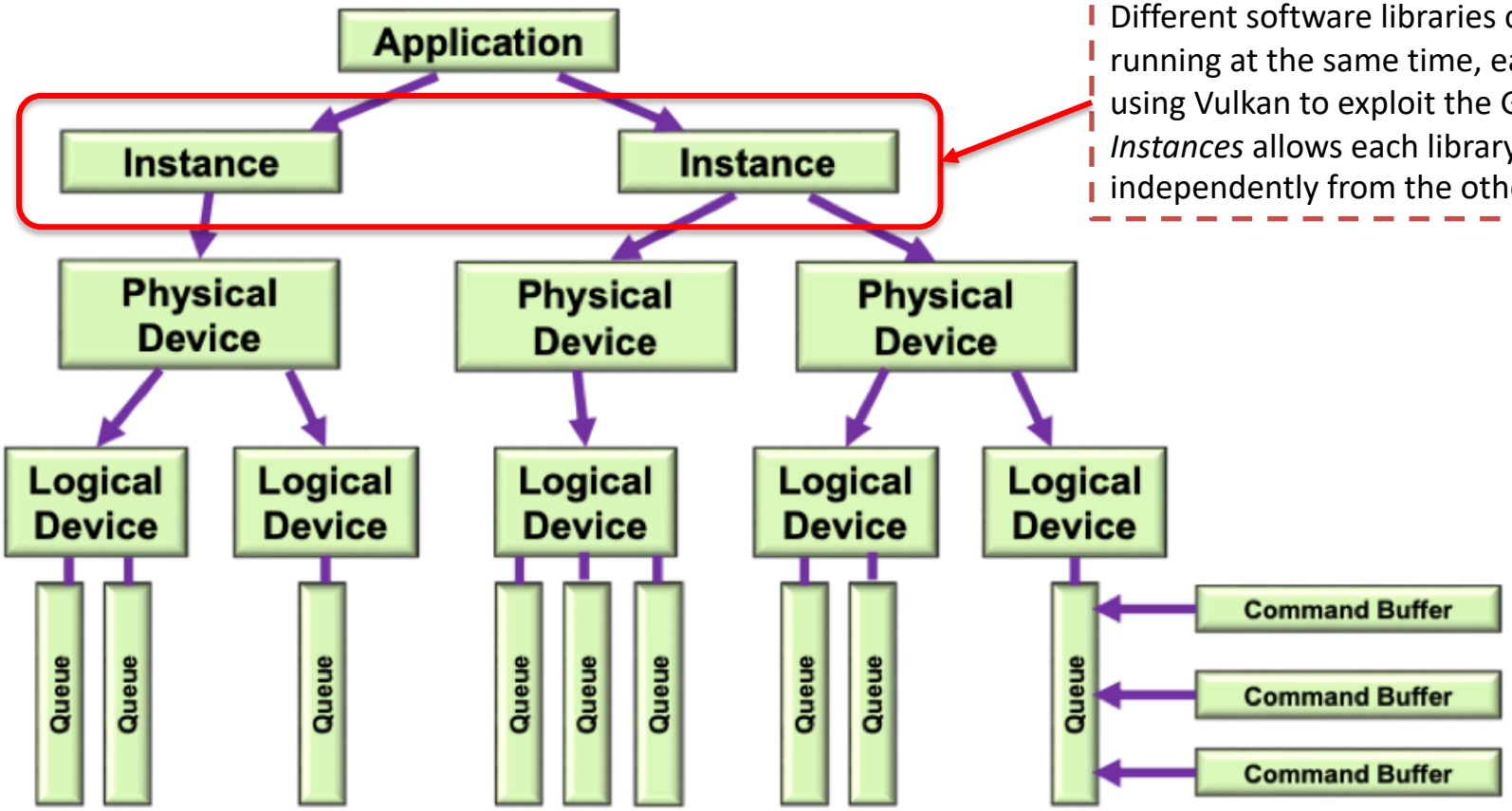
The initialization of the Vulkan support is quite complex due to its large number of alternatives.

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

# A Vulkan architecture

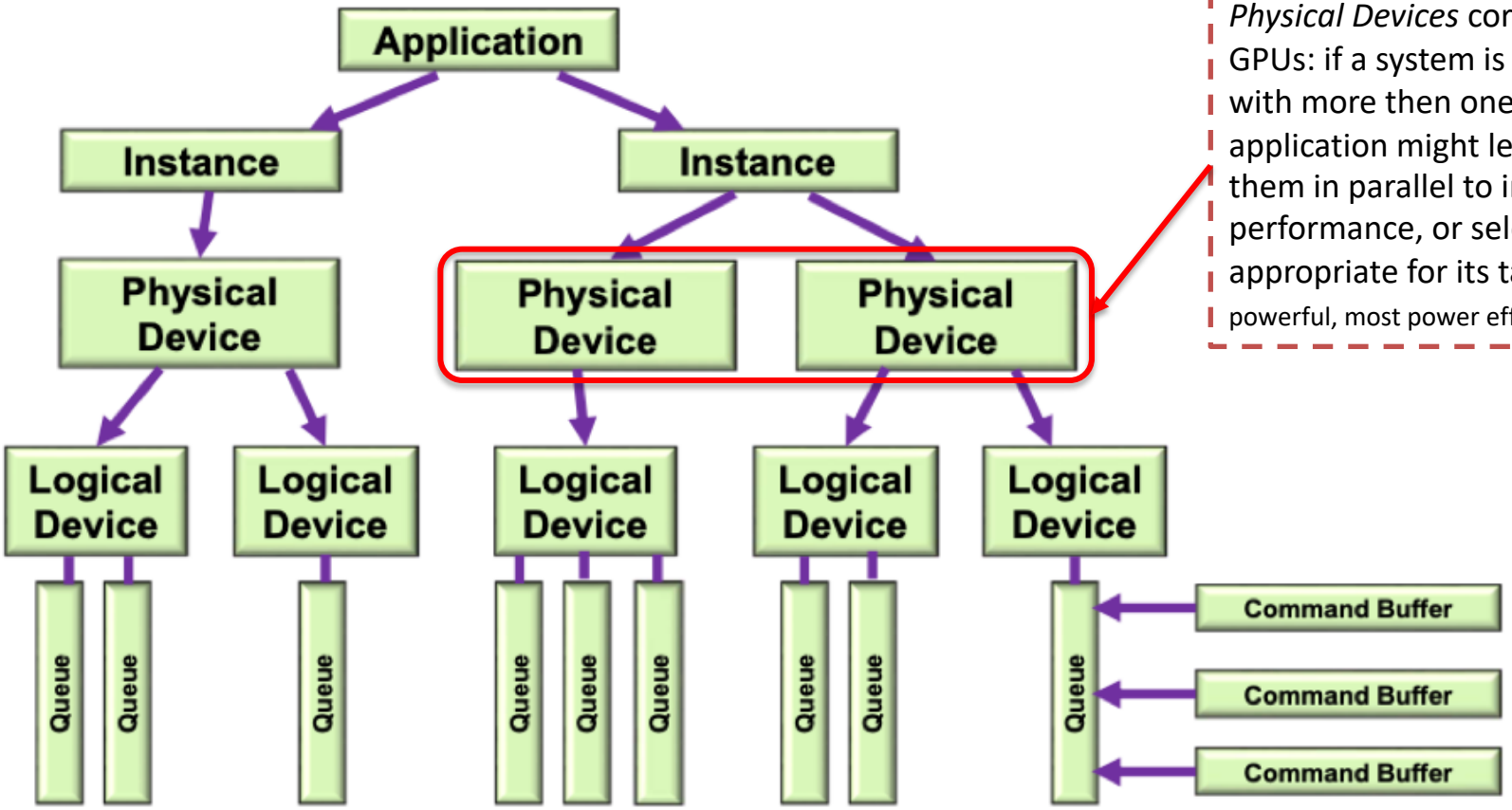


# A Vulkan architecture



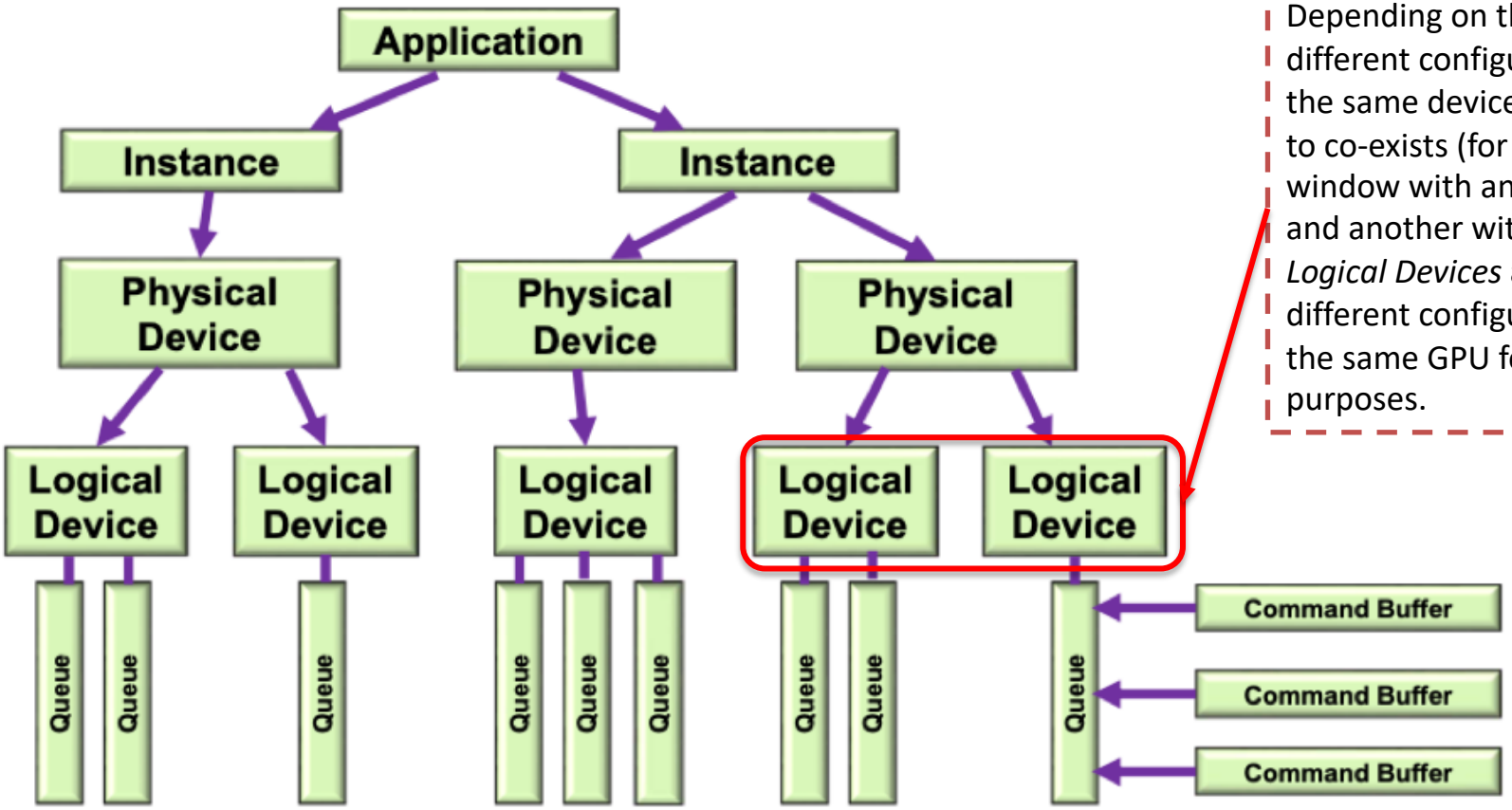

Different software libraries can be running at the same time, each one using Vulkan to exploit the GPUs. *Instances* allows each library to work independently from the others.

# A Vulkan architecture



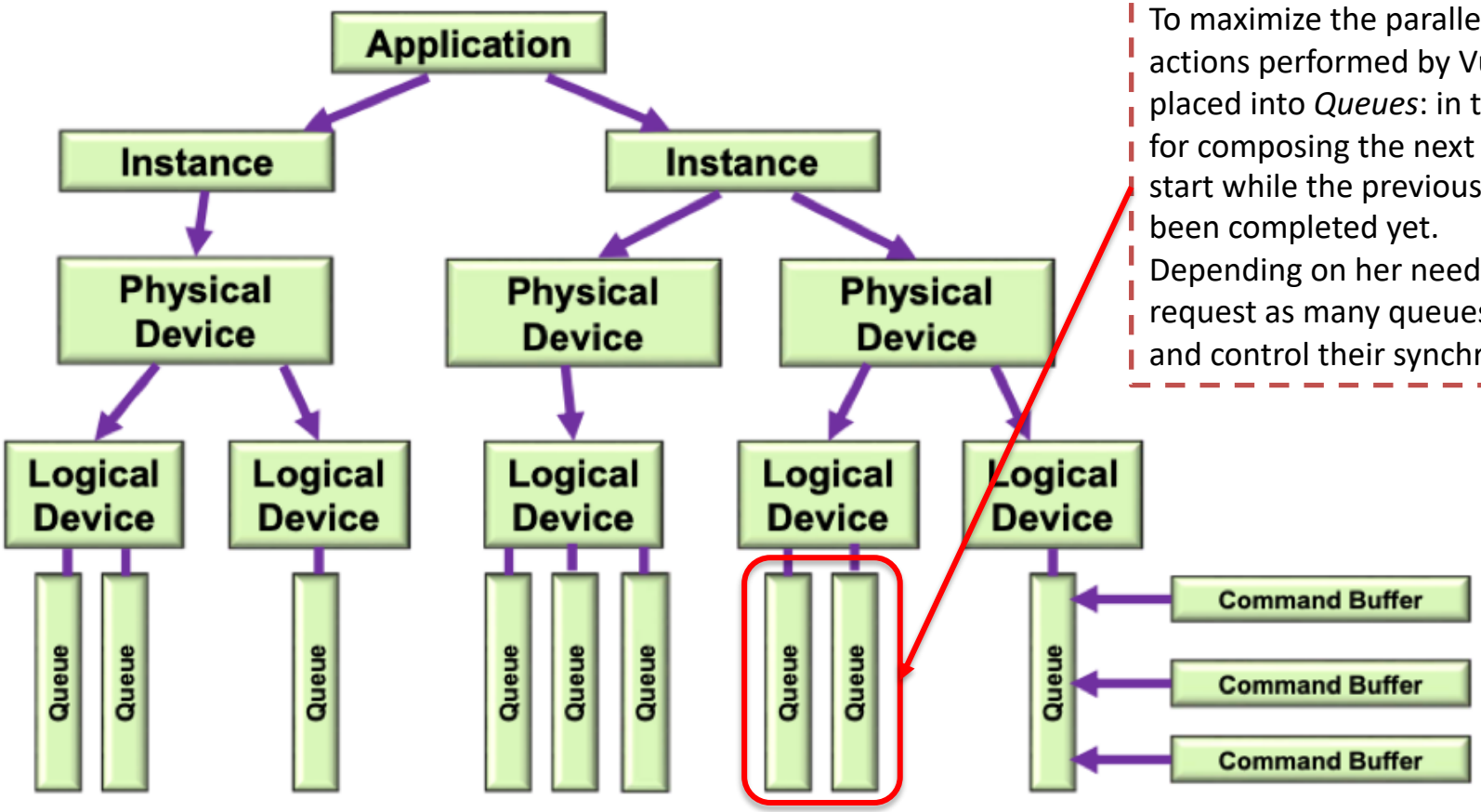

*Physical Devices* corresponds to GPUs: if a system is equipped with more then one, an application might leverage all of them in parallel to increase its performance, or select the most appropriate for its task (i.e. most powerful, most power efficient).

# A Vulkan architecture



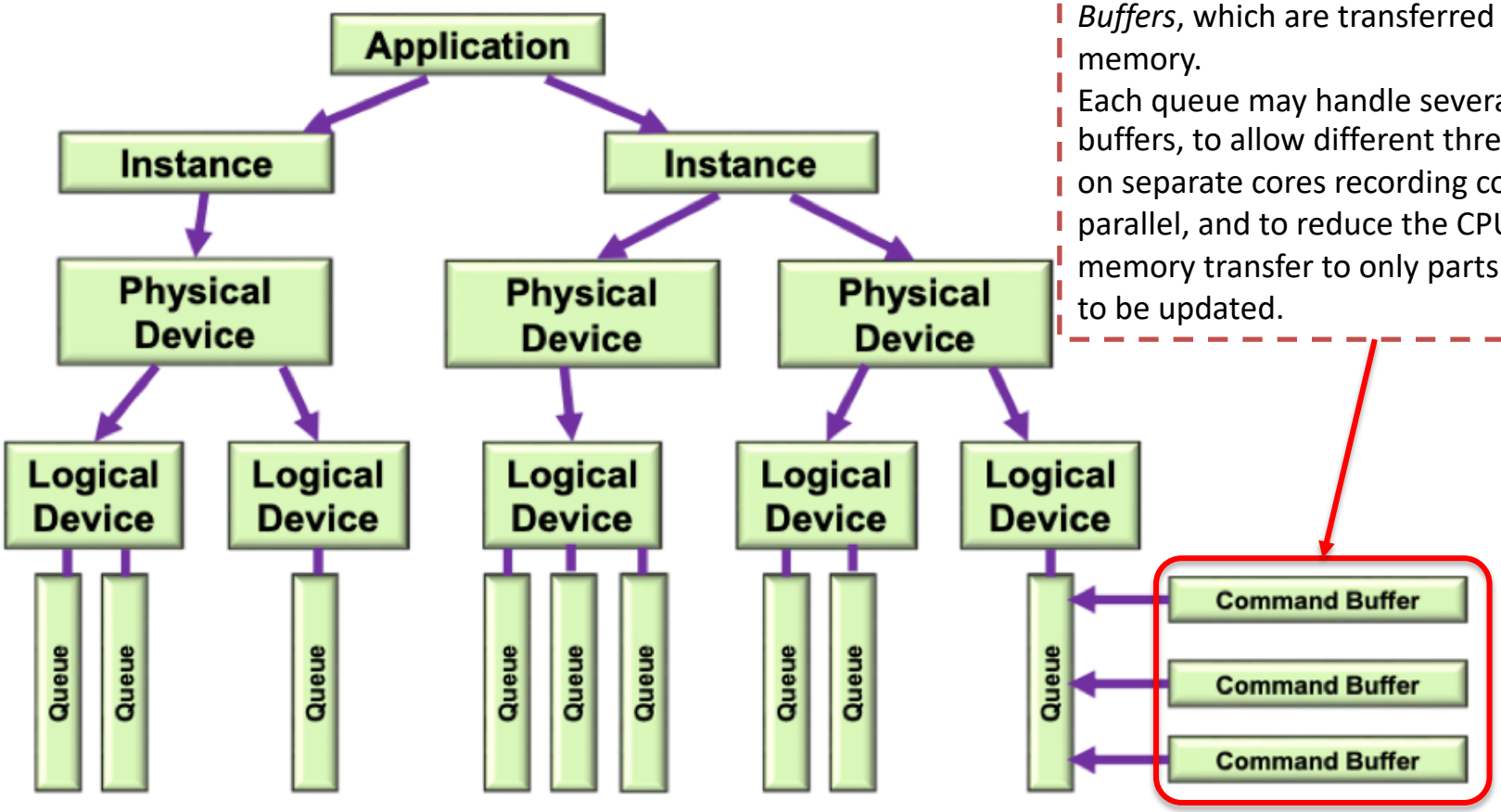

Depending on the needs, different configurations of the same device might need to co-exists (for example, a window with antialiasing, and another without). *Logical Devices* allow to use different configurations of the same GPU for different purposes.

# A Vulkan architecture




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

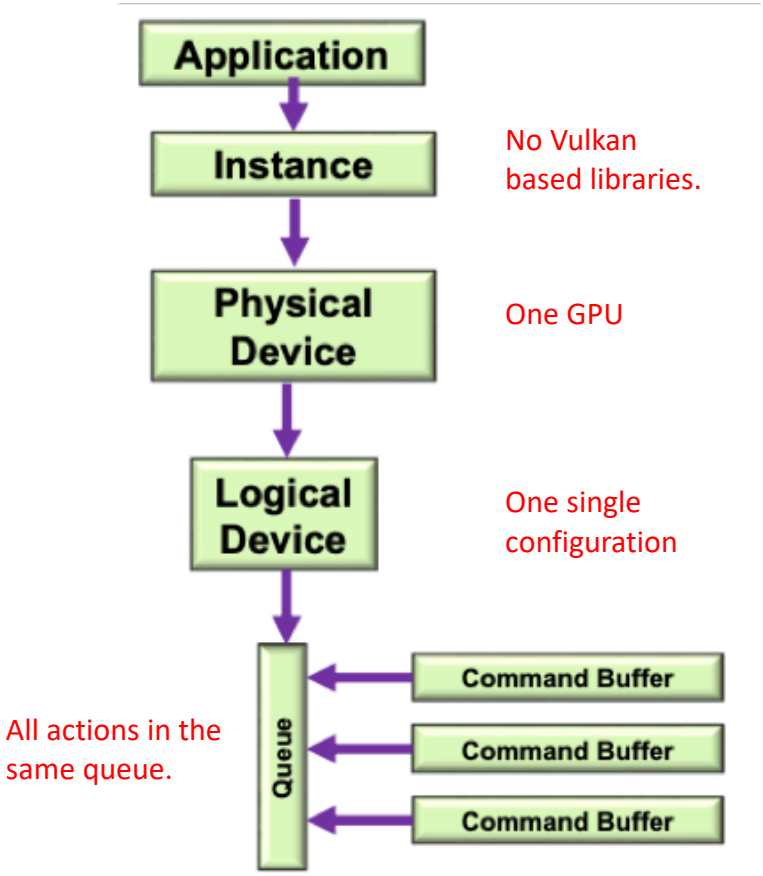
# A Vulkan architecture




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.

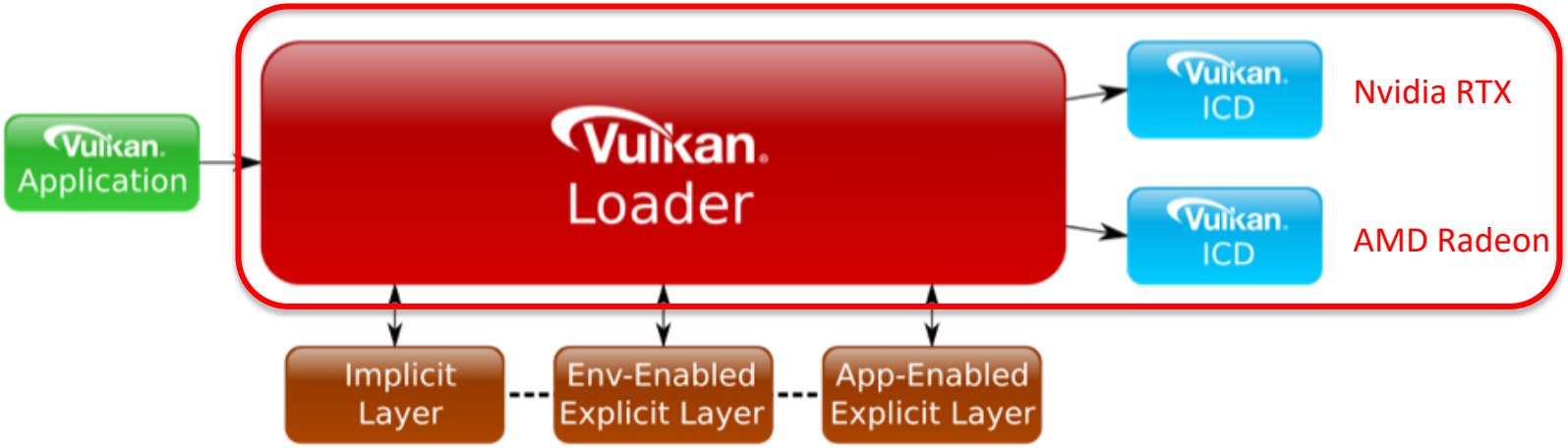
# A typical Vulkan Application

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



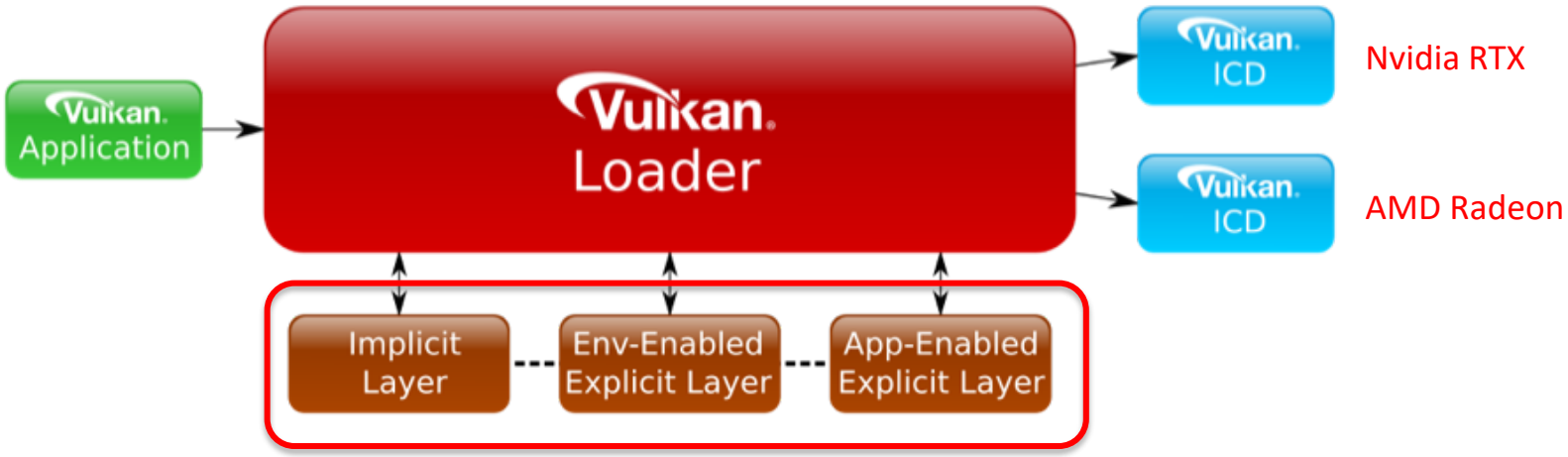
# The Extensions mechanism

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, or to access special hardware features.



# The Extensions mechanism

Instance extensions can be enumerated using the `vkEnumerateInstanceExtensionProperties` command.

Depending on its parameters (null pointer in the last parameter), it can either count the available extensions, or enumerate them.

Usually the procedure is called twice: first to count the extensions, then a buffer of the suitable size is allocated, and with the second call it is filled with the retrieved extensions names.

```
// query the number of instance extensions
uint32_t extensionCount = 0;
vkEnumerateInstanceExtensionProperties(nullptr, &extensionCount, nullptr);
// query the name of the instance extensions
std::vector<VkExtensionProperties> extensions(extensionCount);
vkEnumerateInstanceExtensionProperties(nullptr, &extensionCount,
                                       extensions.data());
// print the available extensions
std::cout << "Available global extensions:\n";
for (const auto& extension : extensions) {
    std::cout << '\t' << extension.extensionName << '\n';
}
```

# The Extensions mechanism

Here you can find 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

# Resource creation procedure

Most of the resource creation procedures follow the same pattern: they require a pointer to a data structure containing the parameters (first parameter), and return another pointer to a structure containing the result (last parameter).

```
VkInstance instance;

VkApplicationInfo appInfo{};
appInfo.sType = VK_STRUCTURE_TYPE_APPLICATION_INFO;
appInfo.pApplicationName = "Assignment 12";
appInfo.applicationVersion = VK_MAKE_VERSION(1, 0, 0);
appInfo.pEngineName = "No Engine";
appInfo.engineVersion = VK_MAKE_VERSION(1, 0, 0);
appInfo.apiVersion = VK_API_VERSION_1_0;

VkInstanceCreateInfo createInfo{};
createInfo.sType = VK_STRUCTURE_TYPE_INSTANCE_CREATE_INFO;
createInfo.pApplicationInfo = &appInfo;
createInfo.enabledExtensionCount = glfwExtensionCount;
createInfo.ppEnabledExtensionNames = glfwExtensions;
createInfo.enabledLayerCount = 0;

VkResult result = vkCreateInstance(&createInfo, nullptr, &instance);

if(result != VK_SUCCESS) {
      throw std::runtime_error("failed to create instance!");
}
```

# Resource creation procedure

They return a `VkResult` data, which corresponds to `VK_SUCCESS` if the command could be accomplished correctly.

```
VkInstance instance;

	VkApplicationInfo appInfo{};
	appInfo.sType = VK_STRUCTURE_TYPE_APPLICATION_INFO;
	appInfo.pApplicationName = "Assignment 12";
	appInfo.applicationVersion = VK_MAKE_VERSION(1, 0, 0);
	appInfo.pEngineName = "No Engine";
	appInfo.engineVersion = VK_MAKE_VERSION(1, 0, 0);
	appInfo.apiVersion = VK_API_VERSION_1_0;

	VkInstanceCreateInfo createInfo{};
	createInfo.sType = VK_STRUCTURE_TYPE_INSTANCE_CREATE_INFO;
	createInfo.pApplicationInfo = &appInfo;
	createInfo.enabledExtensionCount = glfwExtensionCount;
	createInfo.ppEnabledExtensionNames = glfwExtensions;
	createInfo.enabledLayerCount = 0;

	VkResult result = vkCreateInstance(&createInfo, nullptr, &instance);

	if(result != VK_SUCCESS) {
		throw std::runtime_error("failed to create instance!");
	}
```

# Data structures

All Vulkan data structures have a field called `sType`, where a corresponding `VK_STRUCTURE_TYPE_...` *constant* identifying the object type must be assigned.

```
VkInstance instance;

      VkApplicationInfo appInfo{};
      appInfo.sType = VK_STRUCTURE_TYPE_APPLICATION_INFO;
      appInfo.pApplicationName = "Assignment 12";
      appInfo.applicationVersion = VK_MAKE_VERSION(1, 0, 0);
      appInfo.pEngineName = "No Engine";
      appInfo.engineVersion = VK_MAKE_VERSION(1, 0, 0);
      appInfo.apiVersion = VK_API_VERSION_1_0;

      VkInstanceCreateInfo createInfo{};
      createInfo.sType = VK_STRUCTURE_TYPE_INSTANCE_CREATE_INFO;
      createInfo.pApplicationInfo = &appInfo;
      createInfo.enabledExtensionCount = glfwExtensionCount;
      createInfo.ppEnabledExtensionNames = glfwExtensions;
      createInfo.enabledLayerCount = 0;

      VkResult result = vkCreateInstance(&createInfo, nullptr, &instance);

      if(result != VK_SUCCESS) {
            throw std::runtime_error("failed to create instance!");
      }
```

# Data structures

Sometimes structures are nested to better organize the parameters.

```
VkInstance instance;

	VkApplicationInfo appInfo{};
	appInfo.sType = VK_STRUCTURE_TYPE_APPLICATION_INFO;
	appInfo.pApplicationName = "Assignment 12";
	appInfo.applicationVersion = VK_MAKE_VERSION(1, 0, 0);
	appInfo.pEngineName = "No Engine";
	appInfo.engineVersion = VK_MAKE_VERSION(1, 0, 0);
	appInfo.apiVersion = VK_API_VERSION_1_0;

	VkInstanceCreateInfo createInfo{};
	createInfo.sType = VK_STRUCTURE_TYPE_INSTANCE_CREATE_INFO;
	createInfo.pApplicationInfo = &appInfo;
	createInfo.enabledExtensionCount = glfwExtensionCount;
	createInfo.ppEnabledExtensionNames = glfwExtensions;
	createInfo.enabledLayerCount = 0;

	VkResult result = vkCreateInstance(&createInfo, nullptr, &instance);

	if(result != VK_SUCCESS) {
		throw std::runtime_error("failed to create instance!");
	}
```

# Instance Creation data structures

The *application definition* specifies a few parameters, such as its name, its version, and the required Vulkan version. The “engine” field are reserved for major game engine developer to inform the drivers and allow then for possible custom optimiations.

```
VkInstance instance;

	VkApplicationInfo appInfo{};
	appInfo.sType = VK_STRUCTURE_TYPE_APPLICATION_INFO;
	appInfo.pApplicationName = "Assignment 12";
	appInfo.applicationVersion = VK_MAKE_VERSION(1, 0, 0);
	appInfo.pEngineName = "No Engine";
	appInfo.engineVersion = VK_MAKE_VERSION(1, 0, 0);
	appInfo.apiVersion = VK_API_VERSION_1_0;

	VkInstanceCreateInfo createInfo{};
	createInfo.sType = VK_STRUCTURE_TYPE_INSTANCE_CREATE_INFO;
	createInfo.pApplicationInfo = &appInfo;
	createInfo.enabledExtensionCount = glfwExtensionCount;
	createInfo.ppEnabledExtensionNames = glfwExtensions;
	createInfo.enabledLayerCount = 0;

	VkResult result = vkCreateInstance(&createInfo, nullptr, &instance);

	if(result != VK_SUCCESS) {
		throw std::runtime_error("failed to create instance!");
	}
```

# Instance Creation data structures

The instance parameters specify the application data previously defined, and the extensions required.

```
VkInstance instance;

	VkApplicationInfo appInfo{};
	appInfo.sType = VK_STRUCTURE_TYPE_APPLICATION_INFO;
	appInfo.pApplicationName = "Assignment 12";
	appInfo.applicationVersion = VK_MAKE_VERSION(1, 0, 0);
	appInfo.pEngineName = "No Engine";
	appInfo.engineVersion = VK_MAKE_VERSION(1, 0, 0);
	appInfo.apiVersion = VK_API_VERSION_1_0;

	VkInstanceCreateInfo createInfo{};
	createInfo.sType = VK_STRUCTURE_TYPE_INSTANCE_CREATE_INFO;
	createInfo.pApplicationInfo = &appInfo;
	createInfo.enabledExtensionCount = glfwExtensionCount;
	createInfo.ppEnabledExtensionNames = glfwExtensions;
	createInfo.enabledLayerCount = 0;

	VkResult result = vkCreateInstance(&createInfo, nullptr, &instance);

	if(result != VK_SUCCESS) {
		throw std::runtime_error("failed to create instance!");
	}
```

# Instance Creation data structures

Each O.S. needs a minimal different set of extensions to allow Vulkan interfacing with the corresponding windowing framework.

```
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 data structures

GLFW has the `glfwGetRequiredInstanceExtensions()` function that returns the required extensions to allow the application working in the considered host O.S.

```
VkInstance instance;
…

uint32_t glfwExtensionCount = 0;
const char** glfwExtensions;
glfwExtensions = glfwGetRequiredInstanceExtensions(&glfwExtensionCount);

      VkInstanceCreateInfo createInfo{};
      createInfo.sType = VK_STRUCTURE_TYPE_INSTANCE_CREATE_INFO;
      createInfo.pApplicationInfo = &appInfo;
      createInfo.enabledExtensionCount = glfwExtensionCount;
      createInfo.ppEnabledExtensionNames = glfwExtensions;
      createInfo.enabledLayerCount = 0;

      VkResult result = vkCreateInstance(&createInfo, nullptr, &instance);

      if(result != VK_SUCCESS) {
            throw std::runtime_error("failed to create instance!");
      }
```

# The minimal main loop

The minimal main loop, just waits for the user to close the window with the `glfwWindowShouldClose(…)` and the `glfwPollEvents()` commands, and then leaves the loop:

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

# Resources release

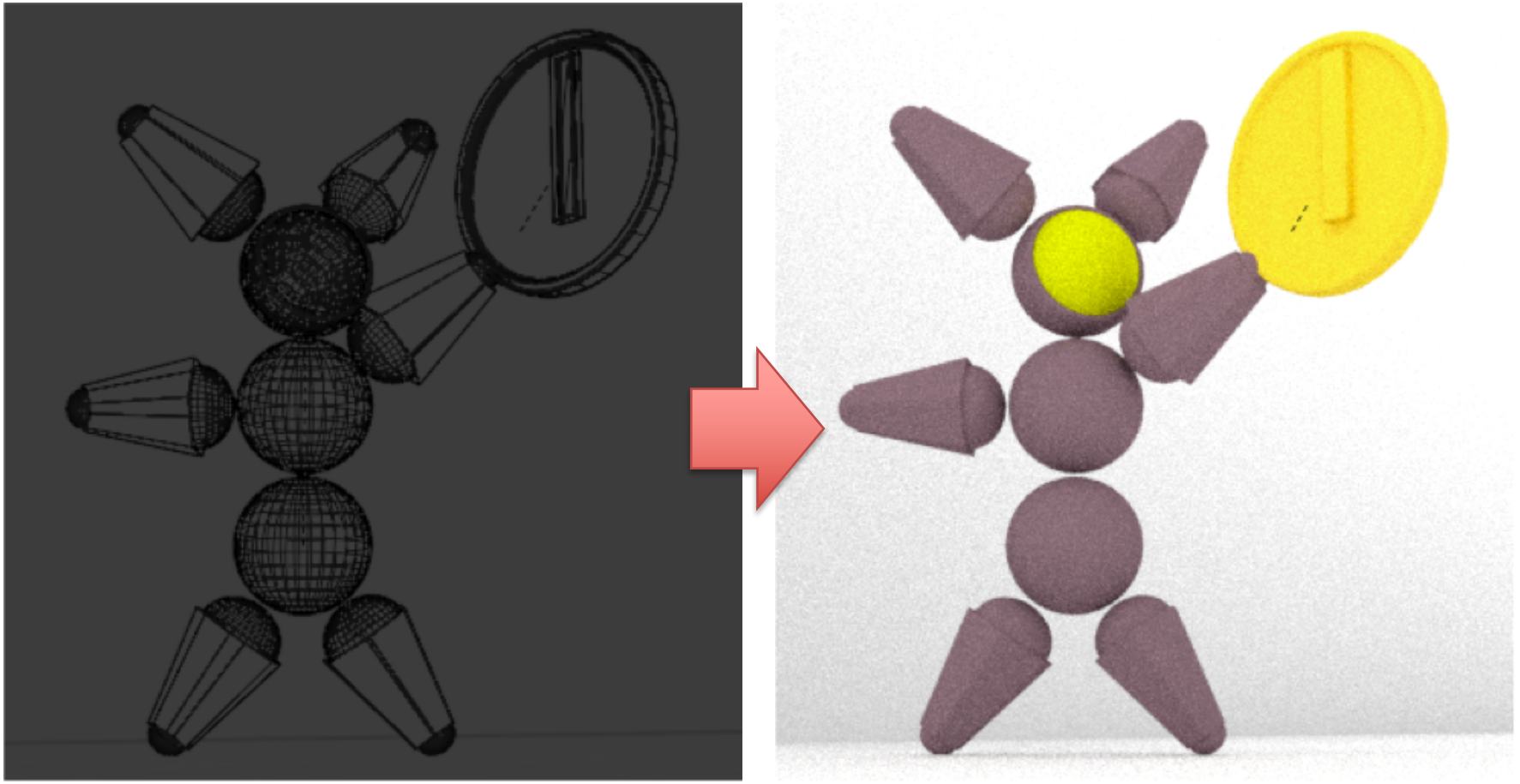
Instance should be released with `vkDestroyInstance(…)` and the O.S. window closed with `glfwDestroyWindow(…)`. The GLFW library, requires also a call to `glfwTerminate()` for freeing all its remaining resources,

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

    glfwDestroyWindow(window);

    glfwTerminate();
}
```

# Rendering



# Rendering

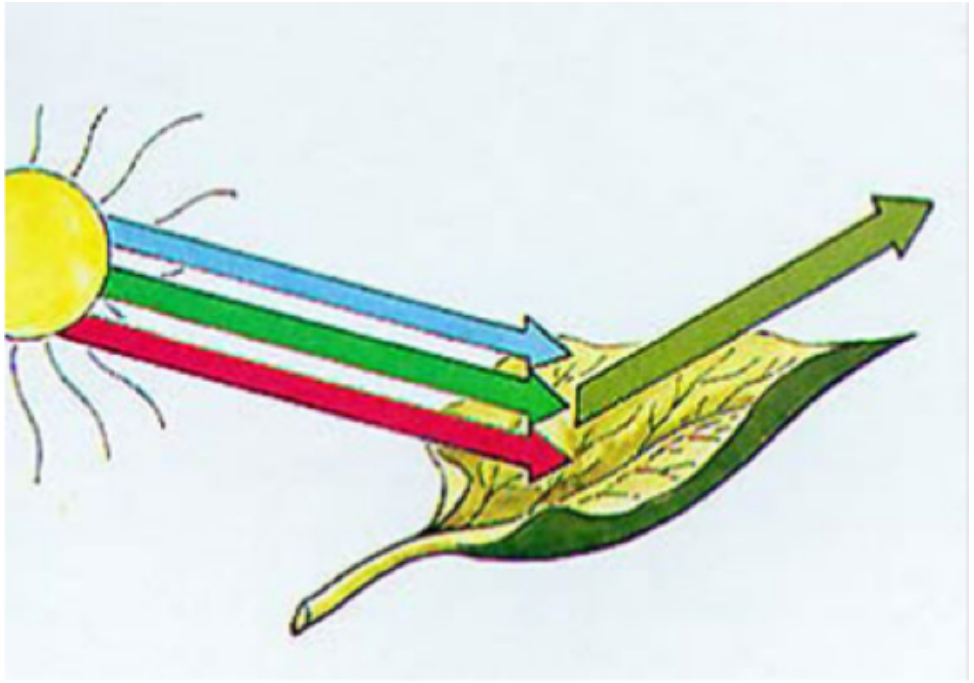
To obtain realistic images with filled 3D primitives, light reflection should be correctly emulated.

*Rendering* reproduces the effects of the illumination by defining light sources and surface properties.

*Lights sources* are elements of the scene from which illumination starts.

# Rendering

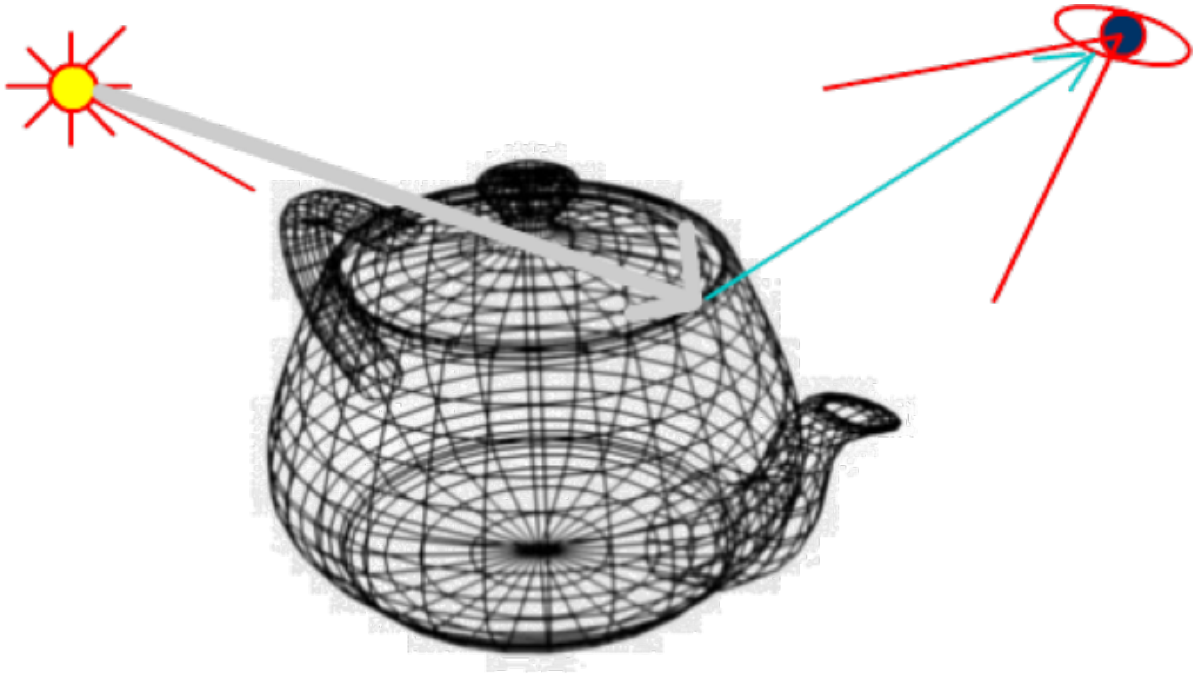
As seen in the previous lessons, light sources emits different frequencies of the spectrum, and objects reflect part of them.



# Rendering

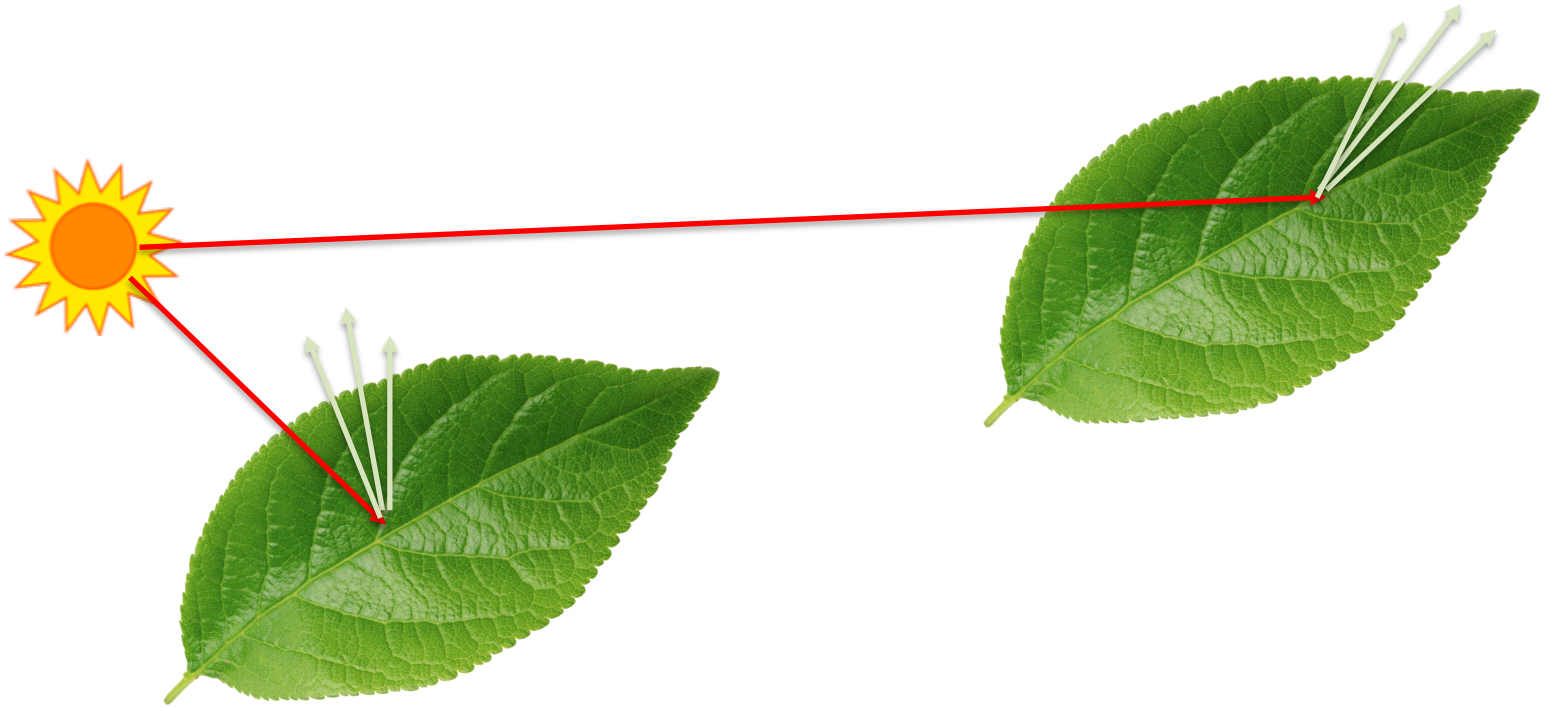
The photons emitted by the light sources bounce on the objects, and some of them reach the viewpoint (the camera).

Rendering computes the quantity and the color of such photons.



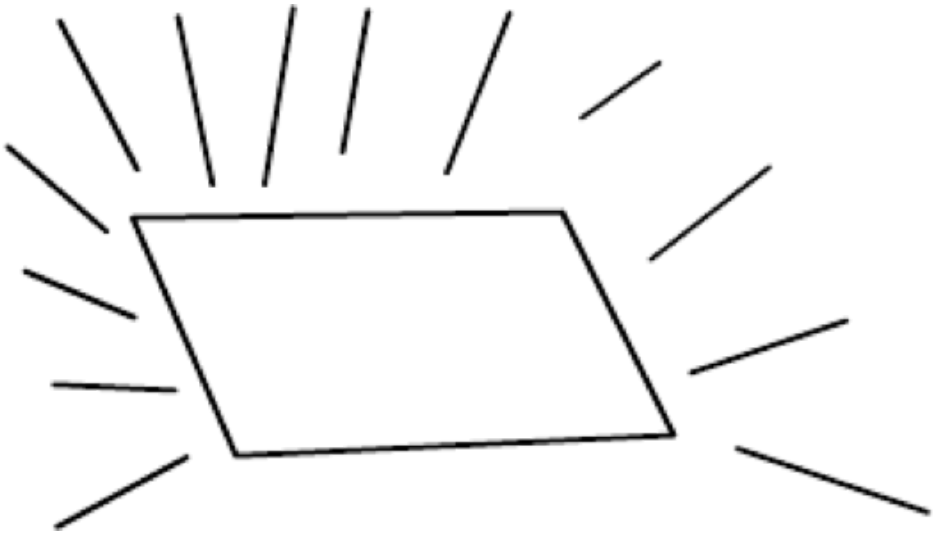
# Rendering

The quantity of light reflected depends on the input direction, and can bounce in many different output directions.



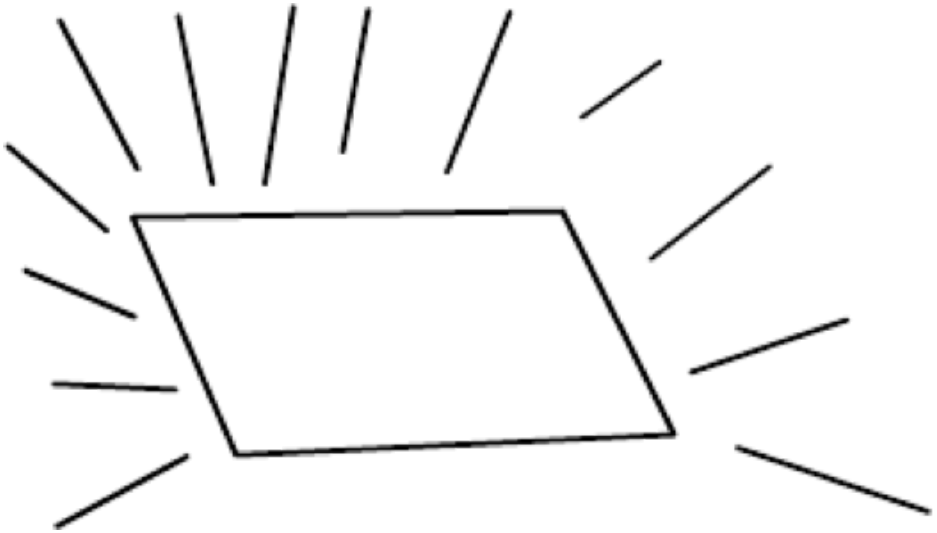
# Radiance

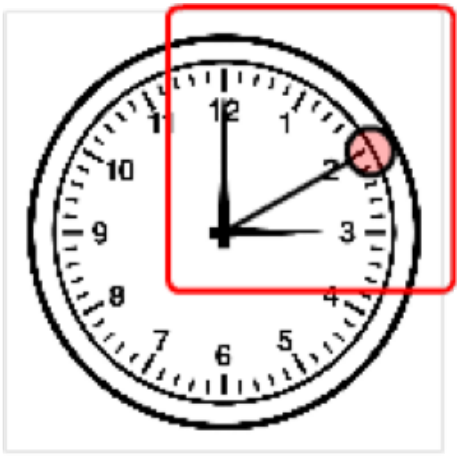
The *energy* (expressed in *Joules* - *J*) measures the total light emitted by a surface in all the directions during a time interval.



# Radiance

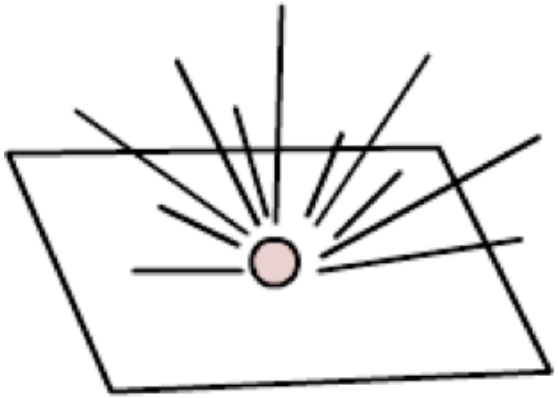
The *power* (measured in *Watts* - $W = J \cdot s^{-1}$) is the instantaneous light energy (emitted by a surface in all the directions in a given time instant).

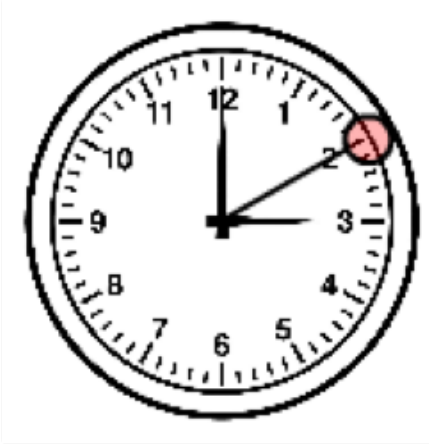




# Radiance

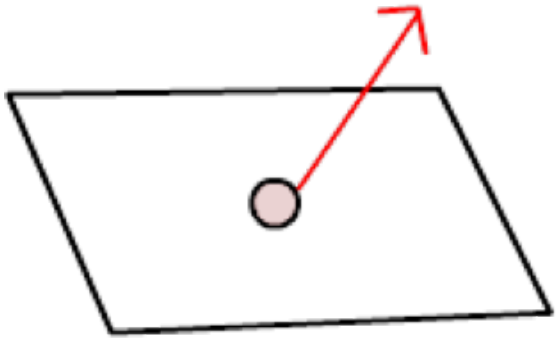
The *irradiance* is the fraction of power emitted by a point of a surface (in a given time instant). It is measured in $W / m^2$. In the following, it will be denoted by letter $E$.

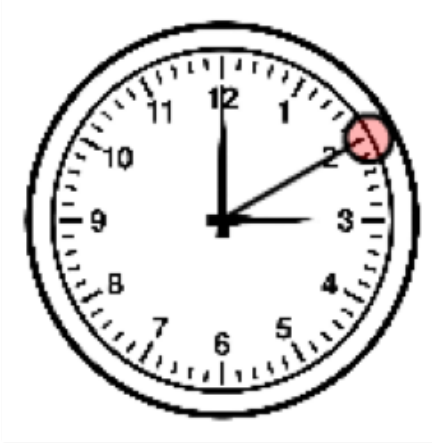




# Radiance

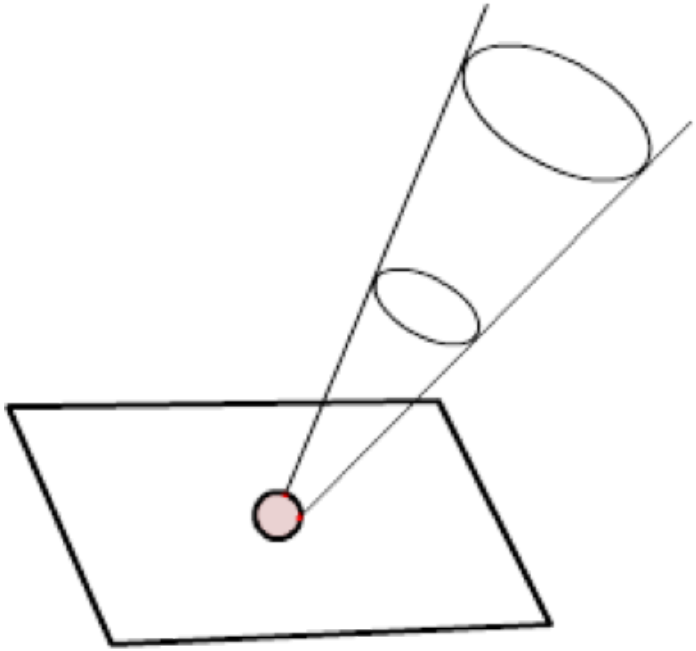
*Radiance* measures the energy emitted in a given time instant from a point of a surface in a given direction. It is measured in $W / (m^2 \cdot sr)$ (where *sr* are *steradians*, the unit of measure for the solid-angle). In the following, it will be denoted by letter *L*.

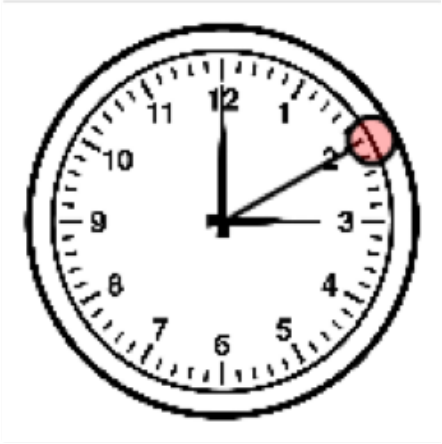




# Radiance

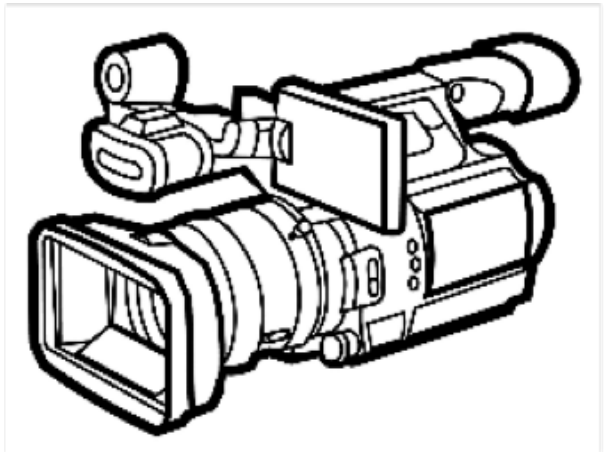
The use of solid angles allows the radiance measure to be independent from the distance from the considered point.





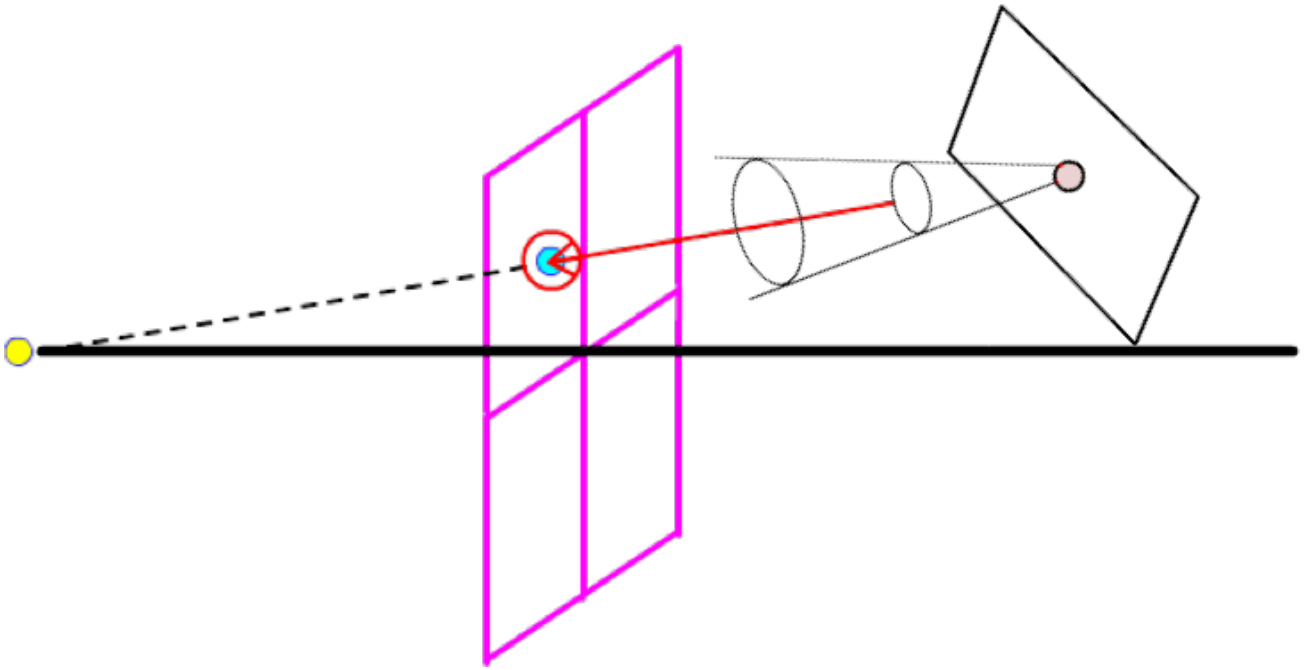
# Radiance

Readings of most light sensors (including the human eyes) are proportional to the *radiance*.



# Radiance

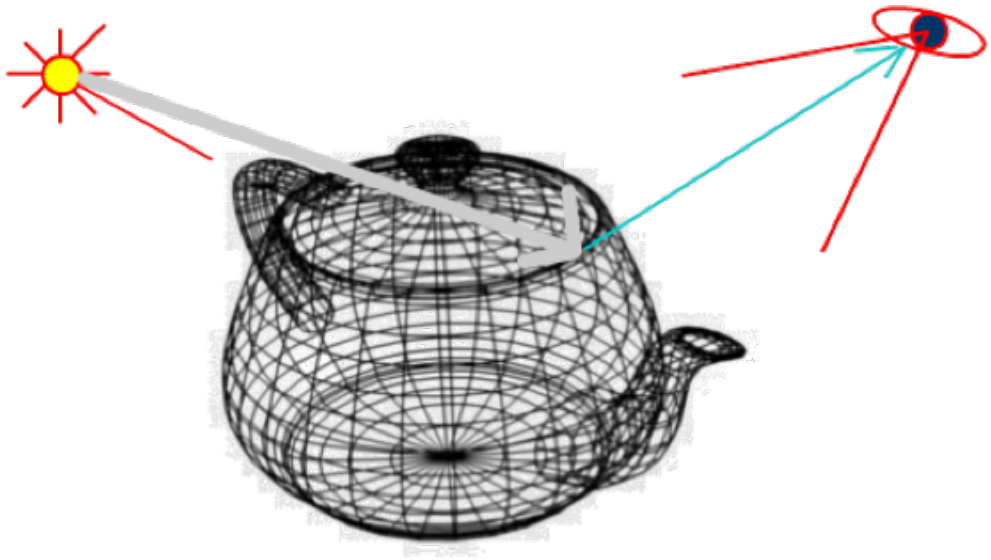
Rendering determines the *radiance* received in *each point* of the *projection plane* (i.e each pixel of the screen) according to the direction of the corresponding *projection ray*.



# The Bidirectional Reflectance Distribution Function

Light emitted by sources is characterized by its radiance.

The material that composes the surface of an object determines the direction and the intensity of reflected light.

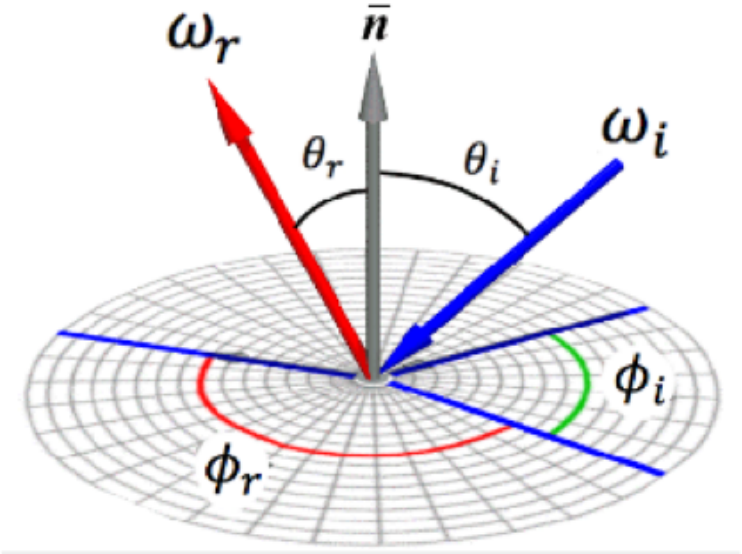


# The Bidirectional Reflectance Distribution Function

In practice, the direction of the bounces depends on the microscopic structure of the surface: it can be very different from material to material.


# The Bidirectional Reflectance Distribution Function

The surface properties of one object can be encoded in a function called the *Bidirectional Reflectance Distribution Function* (*BRDF*).

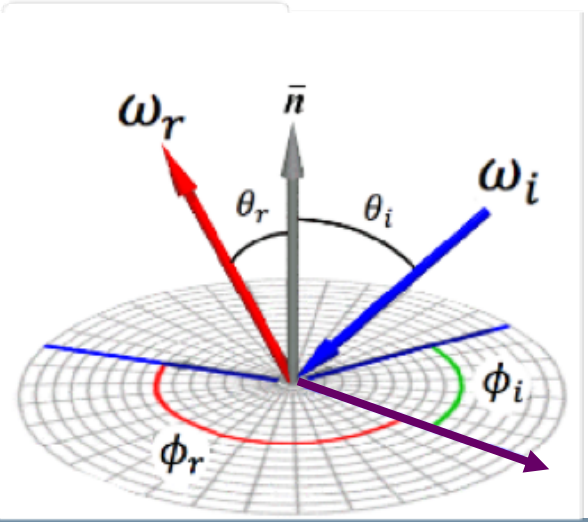


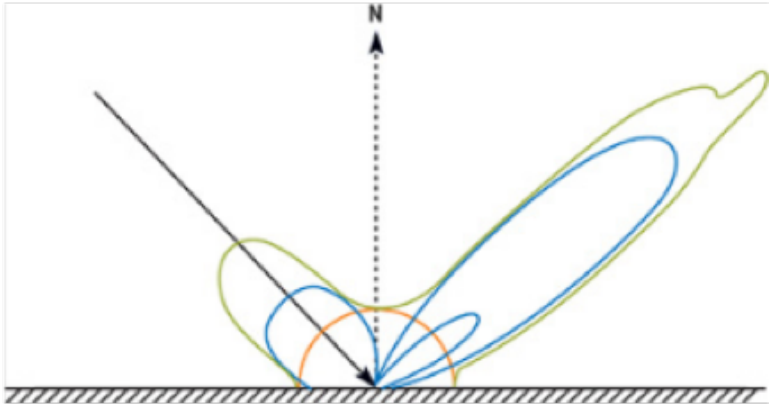
# The Bidirectional Reflectance Distribution Function

The function inputs are the incoming $\omega_i$ and outgoing $\omega_r$ directions. They are both vectors.

The function tells how much *irradiance* from the incoming angle, is reflected to an outgoing angle. It is measured in $sr^{-1}$.

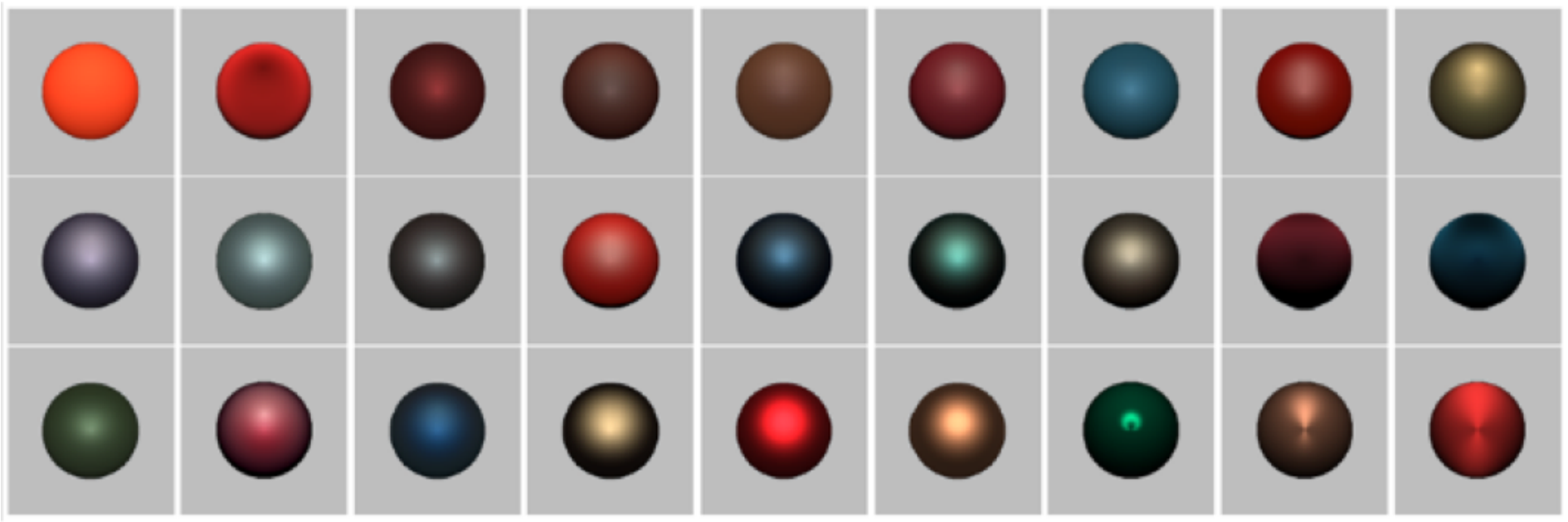
$$f_r(\theta_i, \phi_i, \theta_r, \phi_r) = f_r(\omega_i, \omega_r)$$



# The Bidirectional Reflectance Distribution Function

This allows considering different ways in which the incoming radiance can be reflected at the different angles, depending on each material.



# The Bidirectional Reflectance Distribution Function

A good BRDF should satisfy two main properties: *Reciprocity* and *Energy conservation*.

*Reciprocity* means that if the ingoing and outgoing directions are swapped, the value of the function does not change (for this reason this is known as *bidirectional*).

$$f_r(\omega_i, \omega_r) = f_r(\omega_r, \omega_i)$$

*Energy conservation* means that the BRDF cannot increase the total irradiance that leaves a point on a surface.

$$\int f_r(\omega_i, \omega_r) \cos\theta_r \, d\omega_r \leq 1$$

The integral can be less than one if the point *absorbs* energy.

# The Bidirectional Reflectance Distribution Function

Several approximations to BRDF functions have been proposed in the literature: some of them can provide good results during rendering, even if they do not satisfy the two previous properties.

In the following lessons, we will present several common BRDF functions and how can they be implemented using Shaders.

Databases that provides measured BRDF exist: see for example the *MERL database at*

http://www.merl.com/brdf/

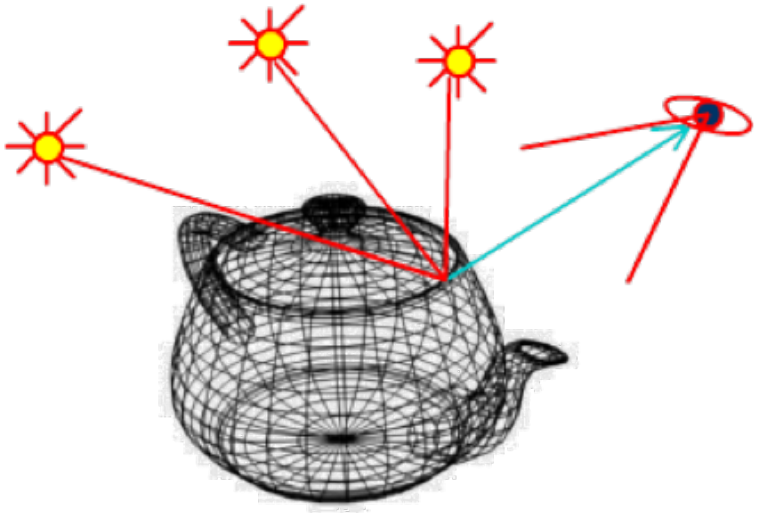
# The rendering equation

The BRDF allows relating together the irradiance in all the directions for all the points of the objects composing a scene. This relation is called the *rendering equation*:

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

# The rendering equation

The equation determines the radiance of each point *x* of any object in any direction $\omega$ of the space.



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

# The rendering equation

Objects can emit light: term $L_e()$ accounts for the light that the object at x emits in direction $\omega$.


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

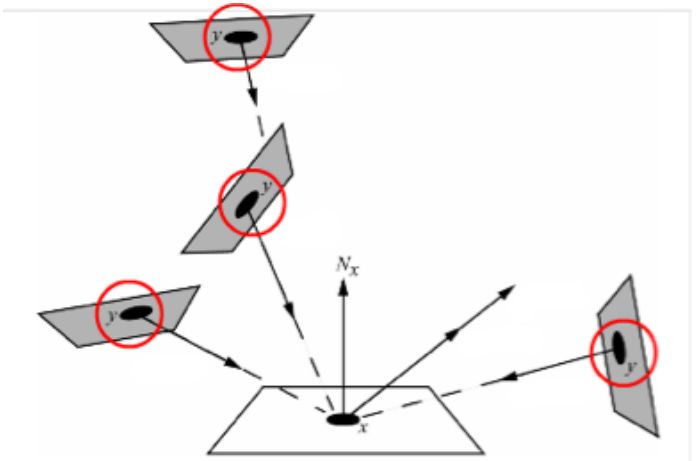
# The rendering equation

This parameter mainly characterizes light sources (i.e. a bulb or a neon).


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

# The rendering equation

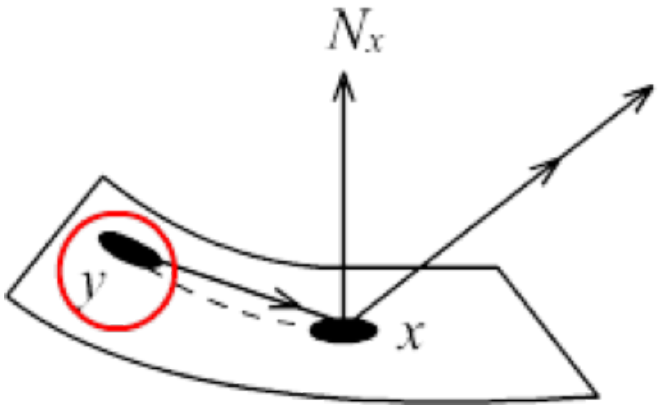
The integral accounts for the light that hits the considered point *x* from all the points *y* of the surfaces of all the objects and lights in the scene.



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

# The rendering equation

The integral also includes other points of the same object to allow the computation of effects such as *self-shadowing* or *self-reflection*.

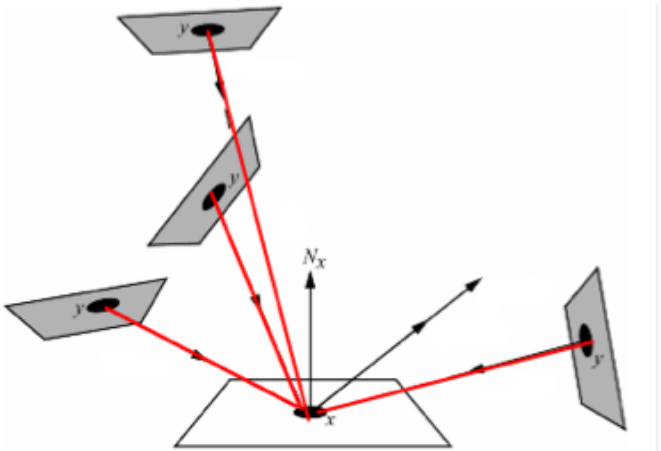



$$L(x, \omega_r) = L_e(x, \omega_r) + \int L(y, \cancel{yx}) f_r(x, \cancel{yx}, \omega_r) G(x, y) V(x, y) dy$$

# The rendering equation

For each object, the equation considers the radiance emitted toward point *x*.

Here $\overrightarrow{yx}$ represents the direction of the line that connects point *y* to point *x*.

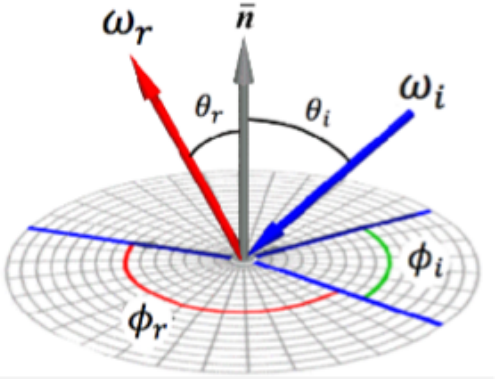



$$L(x, \omega_r) = L_e(x, \omega_r) + \int L(y, \overrightarrow{yx}) f_r(x, \overrightarrow{yx}, \omega_r) G(x, y) V(x, y) dy$$

# The rendering equation

$f_r()$ is the *BRDF* of the material of the object at point *x*.

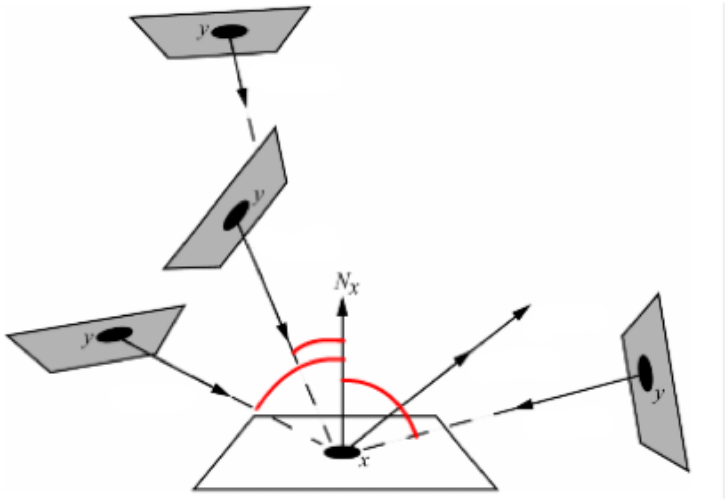
Since the materials of objects might change over their surface, the BRDF depends also on position of point *x*.




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

# The rendering equation

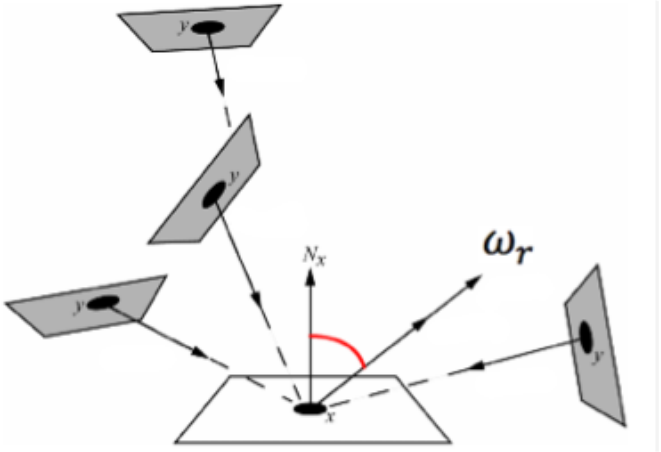
The input angle corresponds to the direction of the segment that connects *y* to *x*.




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

# The rendering equation

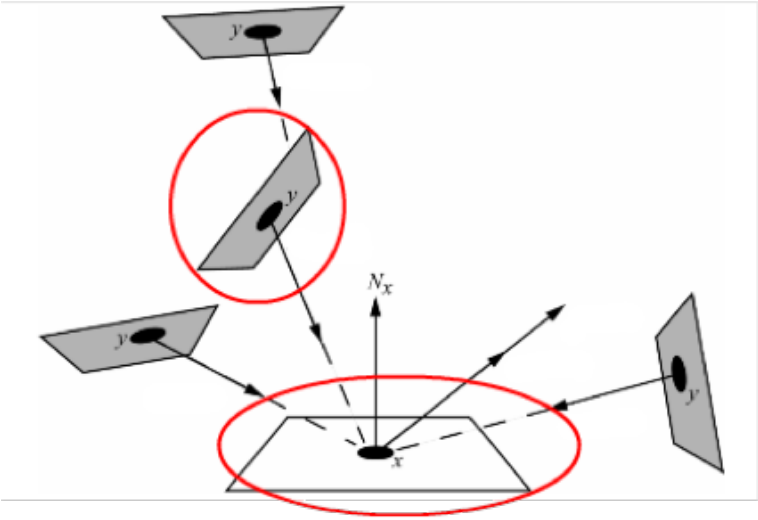
The output angle corresponds to the direction from which the output radiance is being computed.
It is parameter $\omega_r$ on the left hand side of the equation.




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

# The rendering equation

Factor *G(x,y)* encodes the geometric relation between points *x* and *y*.



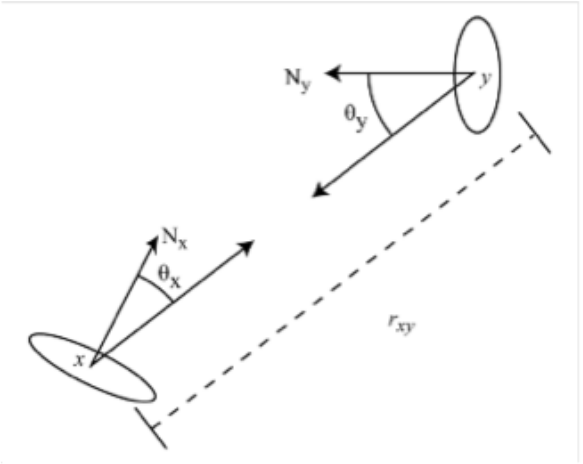

$$L(x, \omega_r) = L_e(x, \omega_r) + \int L(y, \cancel{yx}) f_r(x, \cancel{yx}, \omega_r) G(x, y) V(x, y) dy$$

# The rendering equation

It considers both the relative orientation and the distance of the two points, and it is defined in the following way.

The two *cos()* terms accounts for the angle relative to the respective normal vectors, and $r_{xy}^2$ represents the squared distance of the two points.

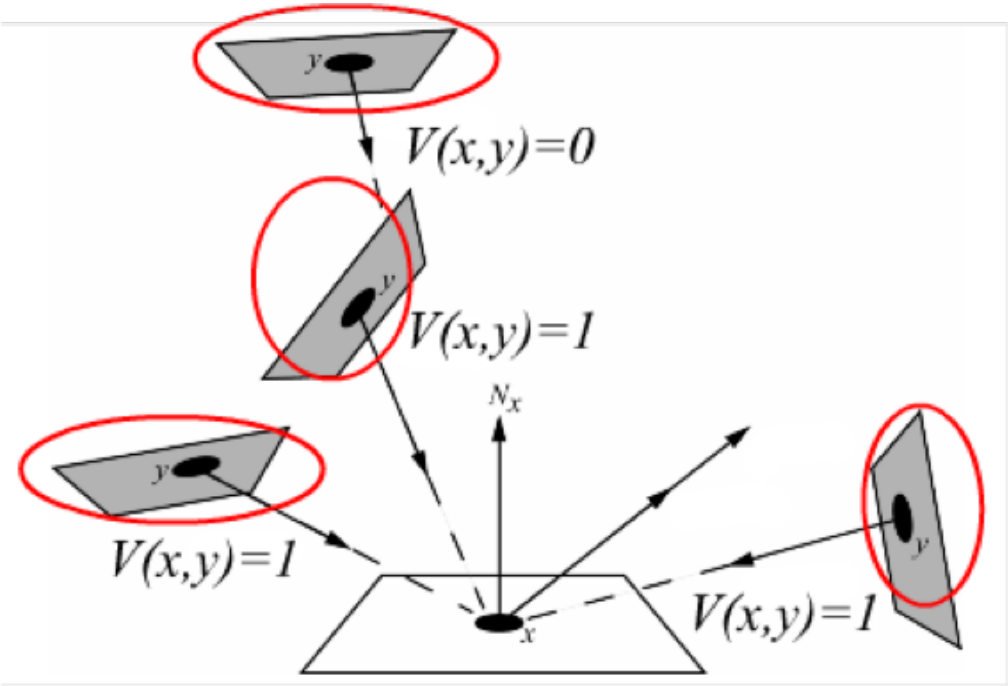
$$G(x,y) = \frac{\cos\theta_x \cos\theta_y}{r_{xy}^2}$$




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

# The rendering equation

Finally term *V(x,y)* considers the visibility between points x and y: *V(x,y) = 1* if the two points can see each other, and *V(x,y) = 0* if point y is hidden by some other object in between.




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

# The rendering equation

Term V(x,y) allows for the computation of shadows, and makes sure that in each input direction at most a single object is considered.

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

# The rendering equation

Term *L(x, ω)* is the unknown of the equation.

Since it appears on both sides of the equation, the rendering equation is an *integral equation of the second kind*.

$$\varphi(x) = f(x) + \lambda \int_a^b K(x,t)\,\varphi(t)\,dt.$$

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

# Considering colors

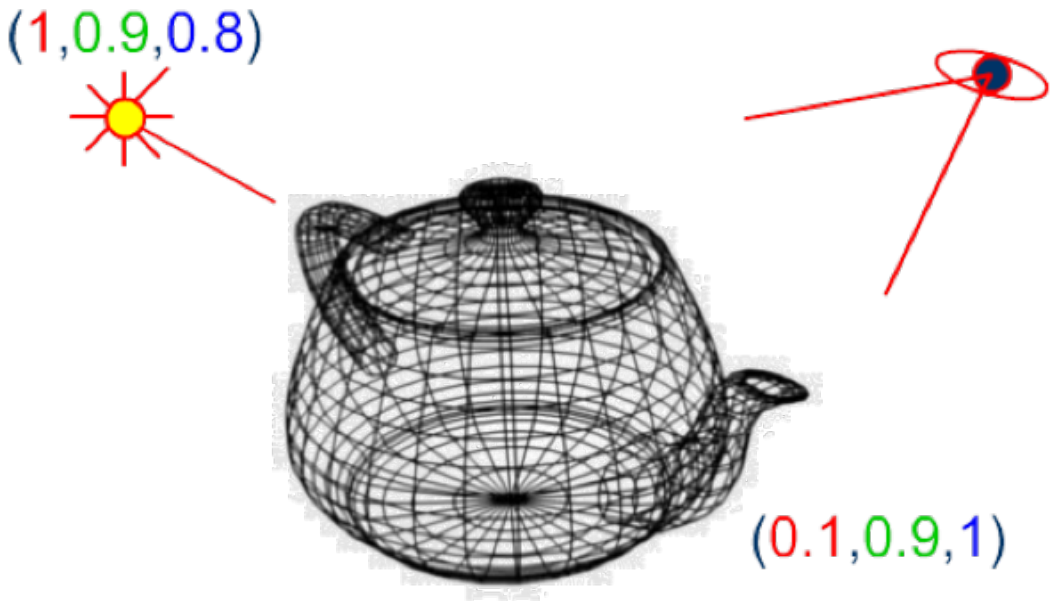
The rendering equation is repeated for every wavelength λ of the light: usually this means that the equation is repeated for the three different RGB channels.

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

# Considering colors

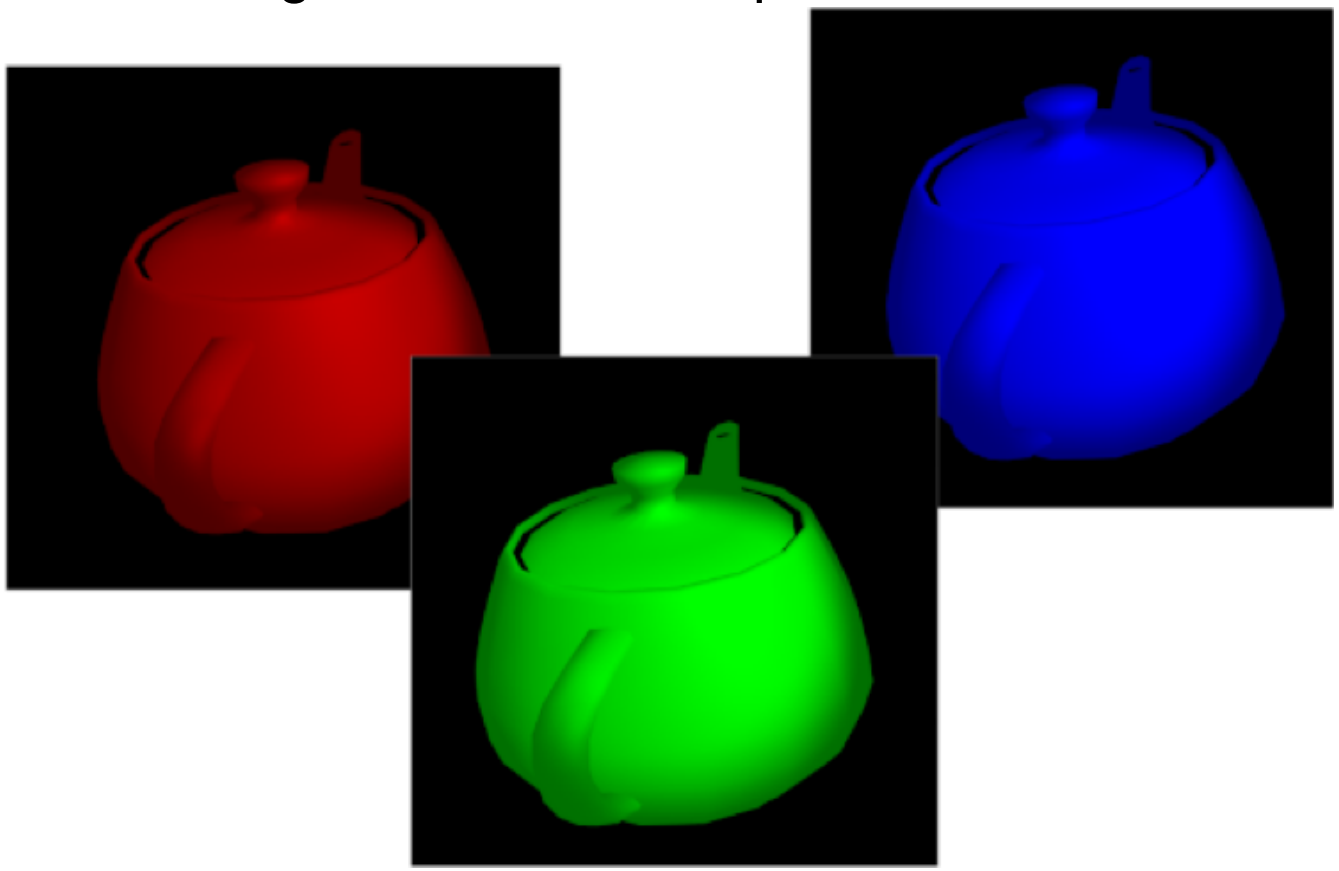
In particular, lights have associated RGB values that accounts for the photons emitted for each of the three main frequencies.

Objects are characterized by a different BRDF for each of the three primary colors.



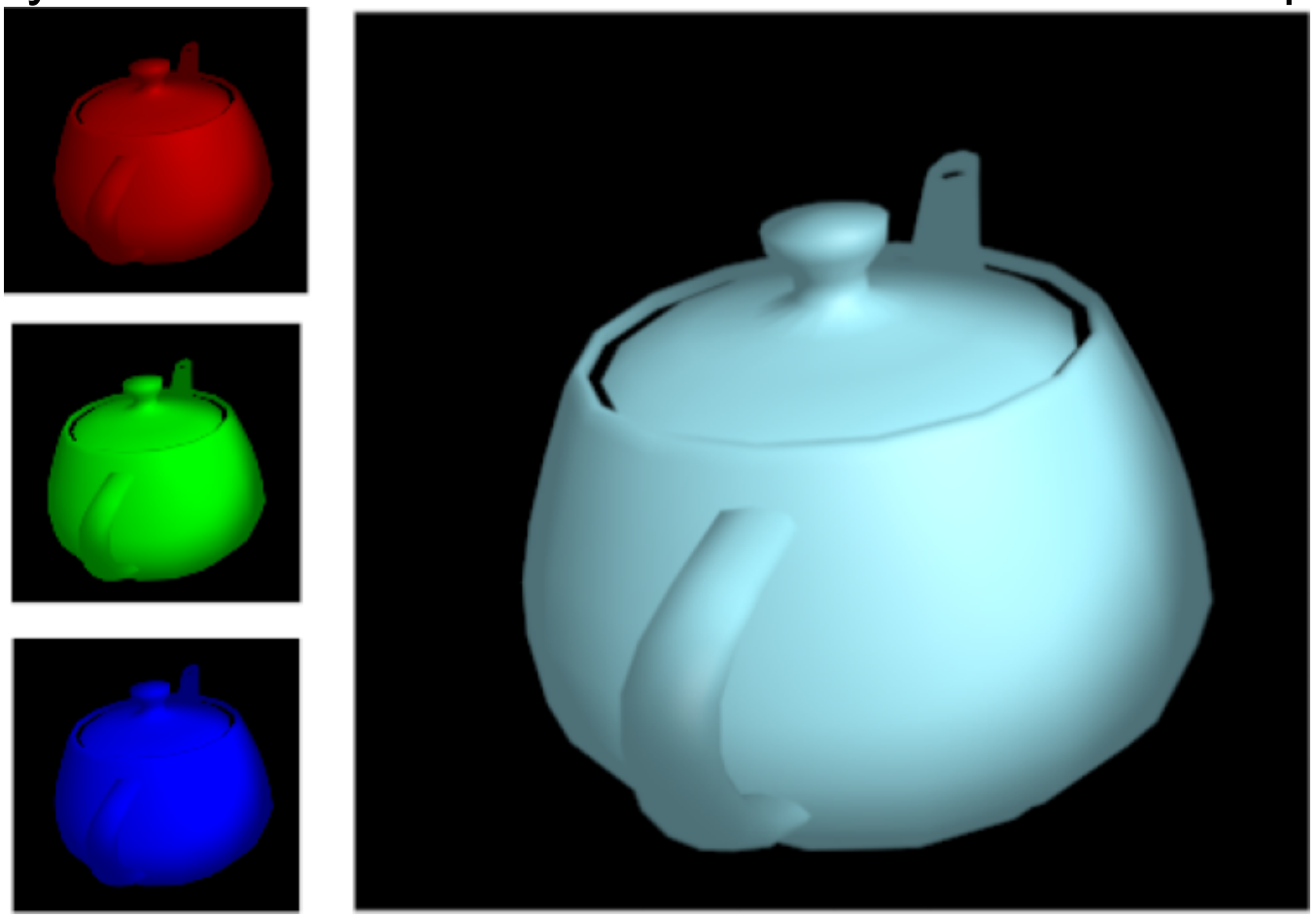
# Considering colors

Three images are produced independently, each considering either the red, green or blue components alone.



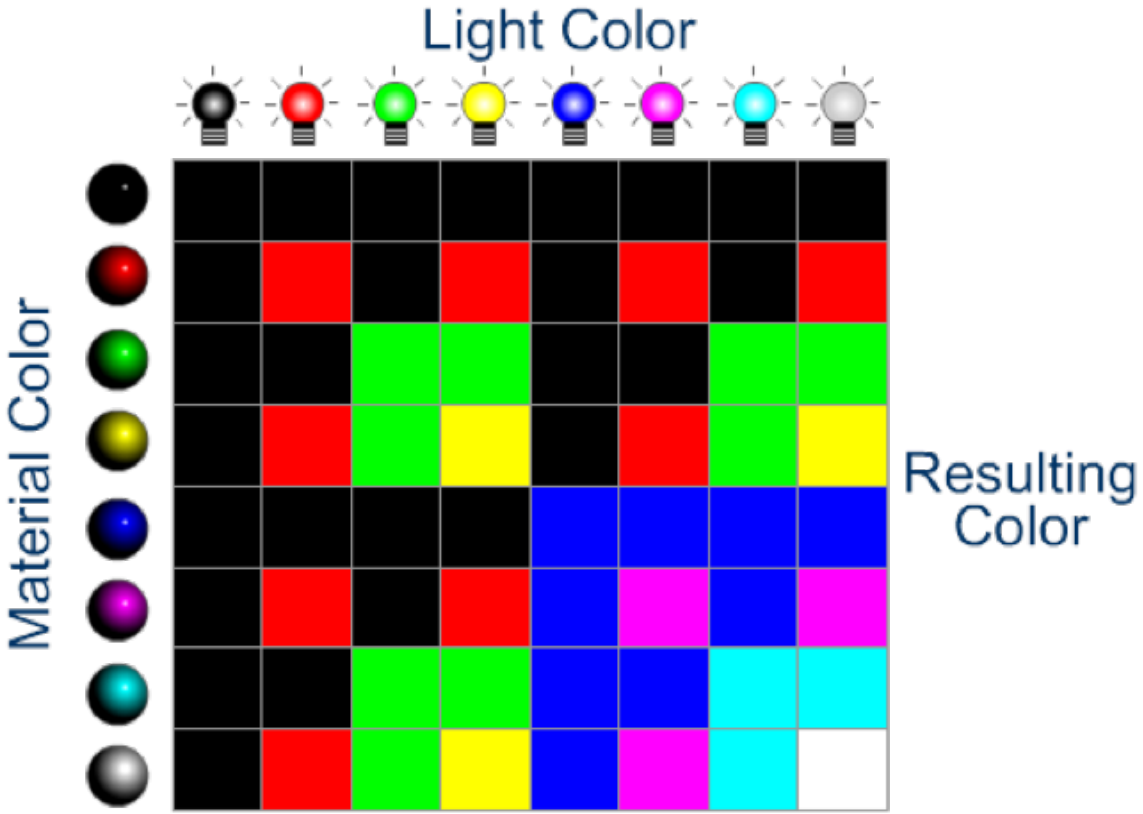
# Considering colors

Finally the three colors are assembled to create the output image.



# Considering colors

Due to the separation of the color components, combinations of light and material colors can lead to unexpected results.



# Extensions to the rendering equation

The proposed rendering equation is capable of accurately computing several effects like reflections, shadows, matte and glossy objects.

However, it cannot simulate other effects such as *participating media* (i.e. rendering of gases and fumes) or even transparent objects like glass or water.

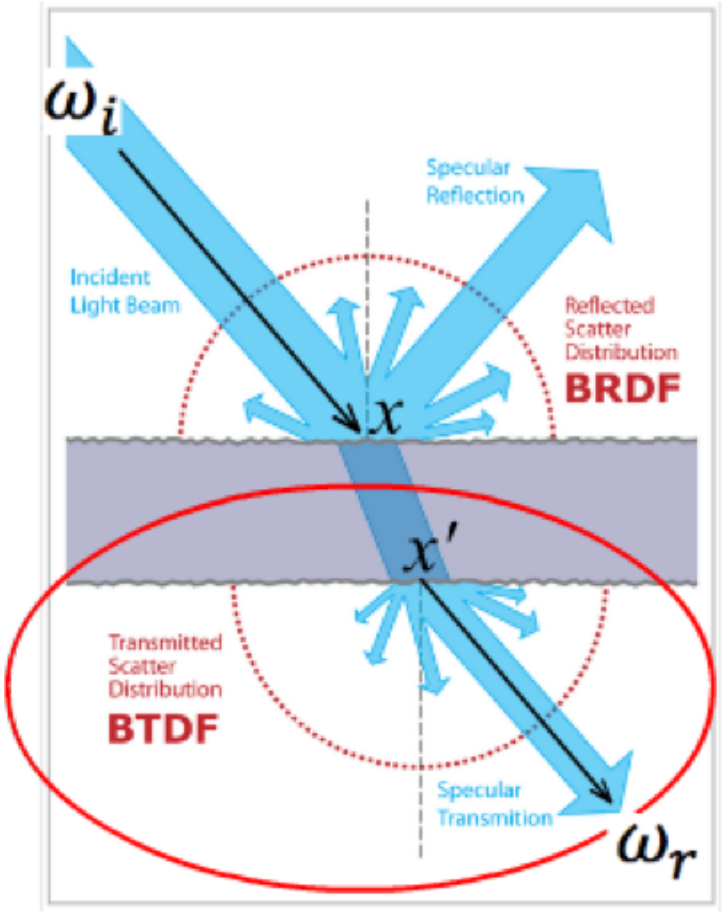
The BDRF and the rendering equations have been extended to account for these other important materials.

# Extensions to the rendering equation

The first extension is to consider transmitted lights: this is done by defining the *BTDF*: *Bidirectional Transmittance Distribution Function*.

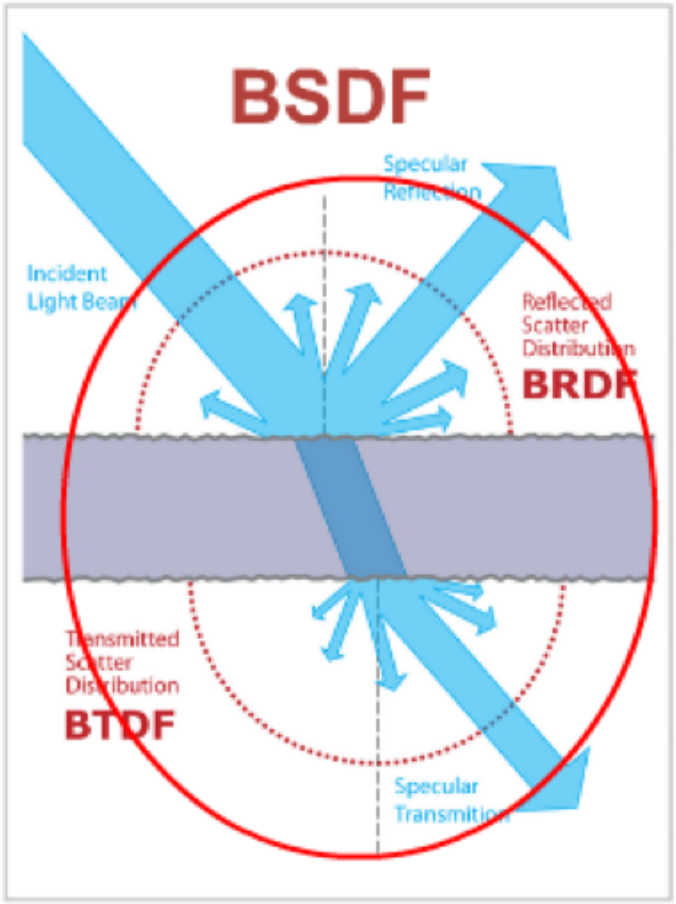
It has a similar definition to the *BRDF*, but it is used in the opposite direction.

$$f_t(\theta_i, \phi_i, \theta_r, \phi_r) = f_t(\omega_i, \omega_r)$$



# Extensions to the rendering equation

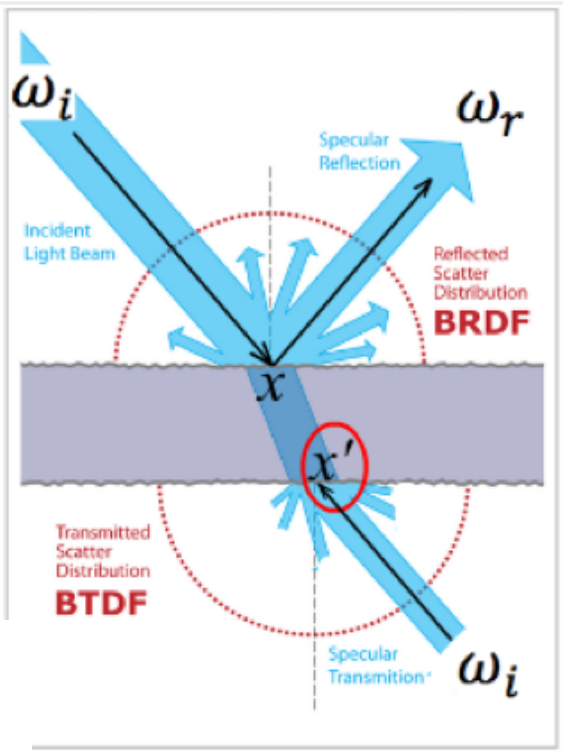
Since usually the angles for the *BRDF* and *BTDF* do not overlap, they are included in a single function called *BSDF*: *Bidirectional Scattering Distribution Function*.



# Extensions to the rendering equation

The rendering equations should be updated to consider both the BRDF and the BTDF.

Here $x'$ denotes the point on the other side of the surface (supposing it is unique) from which light is directed toward point $x$.




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

# Extensions to the rendering equation

In several important materials, light can bounce inside the object and exit from another point. This phenomenon is called *sub-surface scattering*.

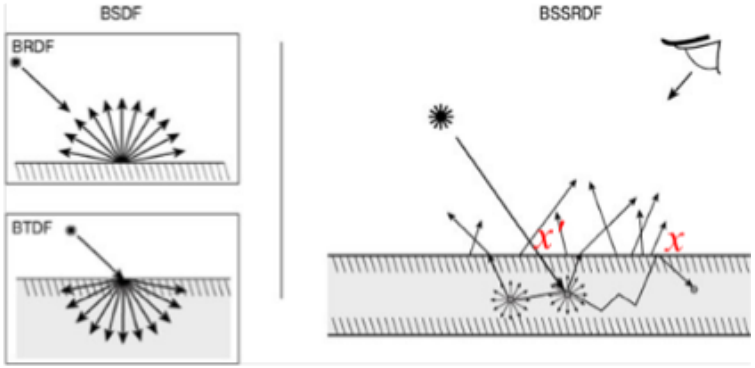
Examples of such materials are human skin and marble.

# Extensions to the rendering equation

A more complex function, called *BSSRDF* (*Bidirectional surface reflectance distribution function*) must be used.

The function has an extra parameter $x'$ that considers the point on the surface from which lights enters at angle $\omega_i$.

The rendering equation now integrates over all the points of an object to compute the quantity of lights that exits from a give position.




$$L(x, \omega_r) = L_e(x, \omega_r) + \iint L\left(y, \overrightarrow{yx'}\right) f_{ss}\left(x, \overrightarrow{yx'}, x', \omega_r\right) G(x', y) V(x', y) dx' dy$$

# Extensions to the rendering equation

The rendering equations are very hard to solve, and generally require complex discretization processes.

In the following, we will see very simple approximations to the rendering equation that are capable of providing good results with a reasonable complexity.