Rendering techniques part 2 and Screen Synchronization

Radiosity

Radiosity proposes a different simplification to the rendering equation. In particular it considers only material that have a constant BRDF.

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

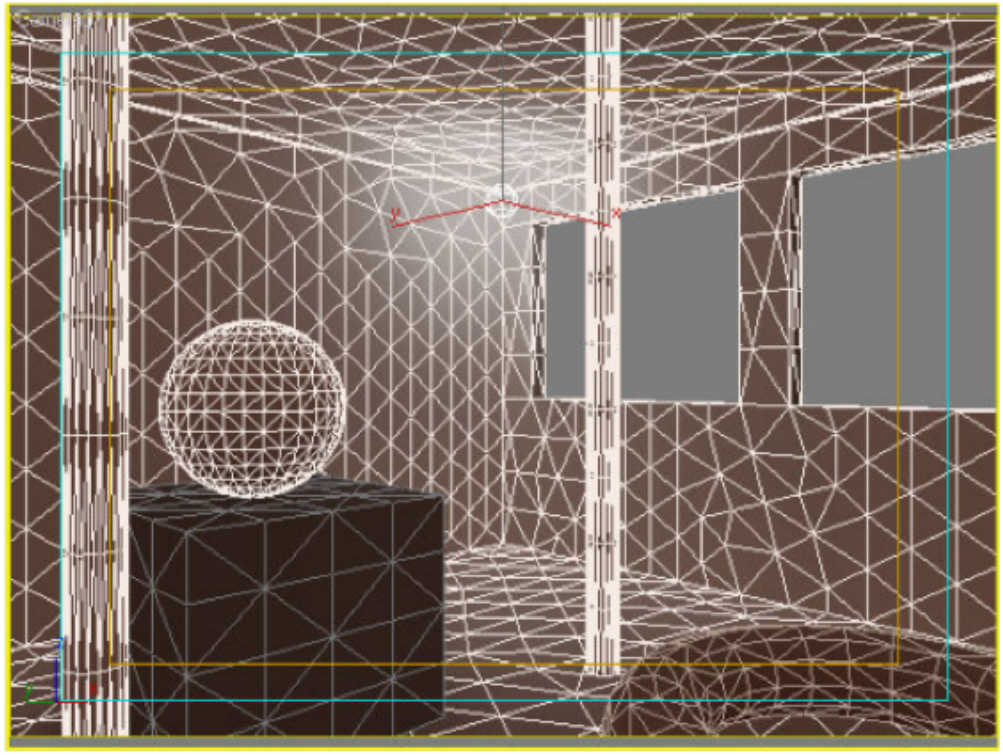
The unknowns of the rendering equations are thus reduced to one per point of the objects since in this way reflection does not depend on the direction from which it is seen. This unknown is called the *radiosity* of the object (that is the output counterparts of the irradiance).

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

$$L(x) = L_e(x) + \rho_x \int L(y) G(x, y) V(x, y) dy$$

Radiosity

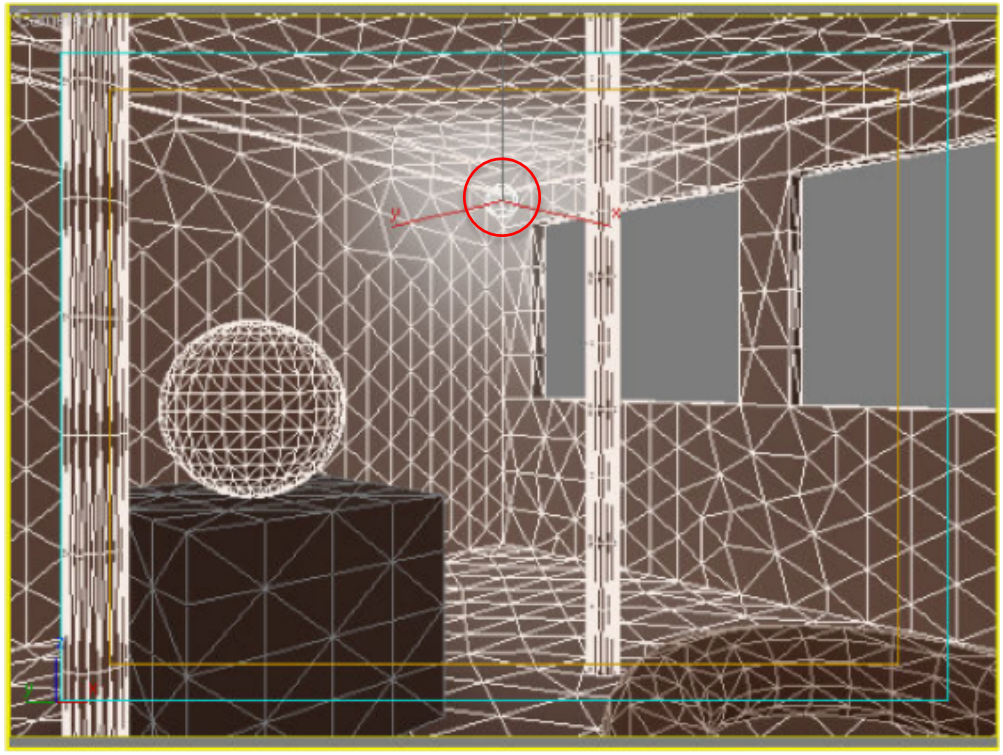
The surfaces of the objects are then split into patches, with one unknown per patch.



Radiosity

Light sources are implemented as patches that emits radiosity.

$$L(x) = L_e(x) + \rho_x \int L(y)G(x,y)V(x,y)dy$$



Radiosity

The rendering equation becomes a (large) system of linear equations that can be solved with an iterative technique.

$$L(x) = L_e(x) + \rho_x \int L(y)G(x,y)V(x,y)dy$$

$$L(x_i) = L_e(x_i) + \rho_{x_i} \sum_{y_j} L(y_j)G(x_i,y_j)V(x_i,y_j)$$

Radiosity

In matrix notation, vector L has one element per patch, and represents its *radiosity*.

Matrix *R* includes the visibility, the constant BRDF, and the geometric relations between any two patches of scene.

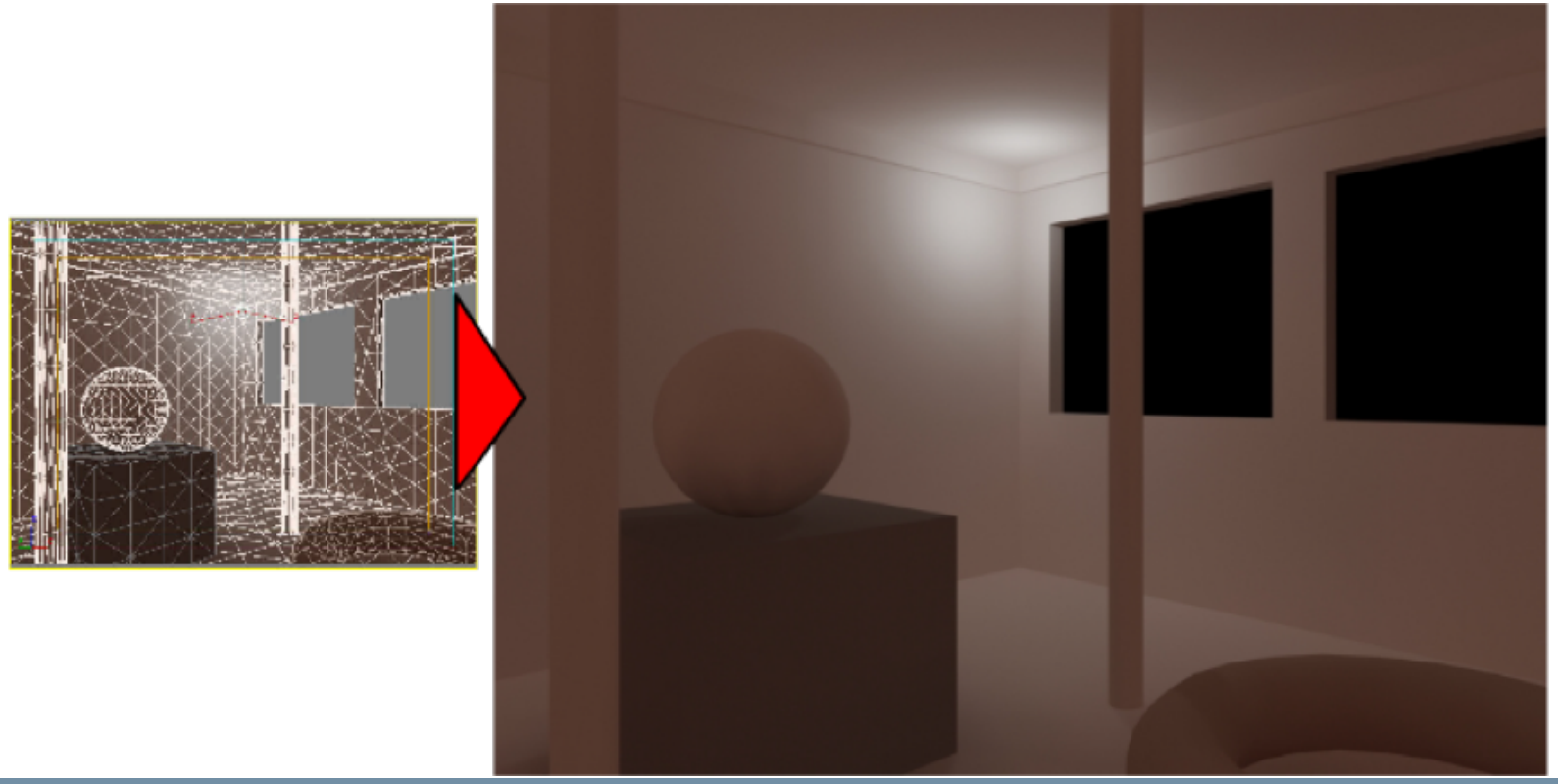
$$L(x_i) = L_e(x_i) + \rho_{x_i} \sum_{y_j} L(y_j) G(x_i, y_j) V(x_i, y_j)$$

$$L(x_i) = L_e(x_i) + \sum_{y_j} L(y_j) R(x_i, y_j) \qquad R(x_i, y_j) = \rho_{x_i} G(x_i, y_j) V(x_i, y_j)$$

$$L = L_e + R \cdot L$$

Radiosity

The solutions of the equations are then interpolated to produce a view of the scene.



Radiosity

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

```
01 Discretize the scene, and compute matrix R
02 Compute the solution of equation L = L_e + R · L
03 Render the scene using either scan-line or raytracing
04    Interpolate L to obtain the color for each pixel
```

Indeed the most time consuming steps are steps **01** and **02**.

However, once they have been computed, the scene can very quickly be rendered from any point of view.

Radiosity

In most of the case, the solution of the equation can be computed with a fixed-point iteration, starting from L = 0, and refining its value at every iteration.

```
01 Lold = Lnew = 0
02 Repeat
03     Lold = Lnew
04     Lnew = L_e + R · Lold
05 Until |Lnew − Lold| > threshold
```

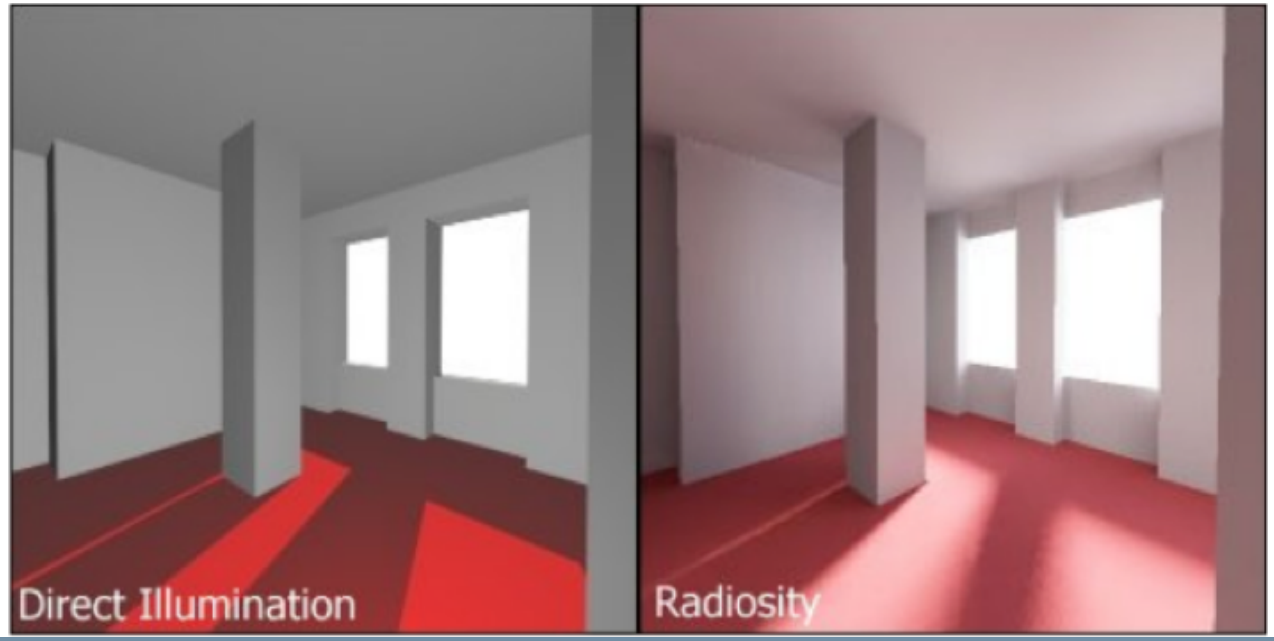
Even if the number of patches is big, the loop converges in a moderate number of iterations.

Radiosity

Radiosity is able to capture indirect illumination effects.

Shadows however are usually very poorly approximated due to the size of the patches.

Mirror reflections and refractions cannot be considered directly, since they greatly depend on the direction from which an object is seen.



Montecarlo techniques

Photorealistic results can only be achieved by approximating the solution of the complete rendering equation.

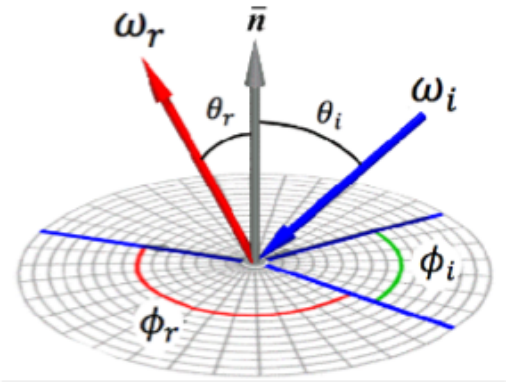
Due to its complexity, *Montecarlo* techniques are usually employed: the integral is computed by averaging several random points and directions chosen from the equations.

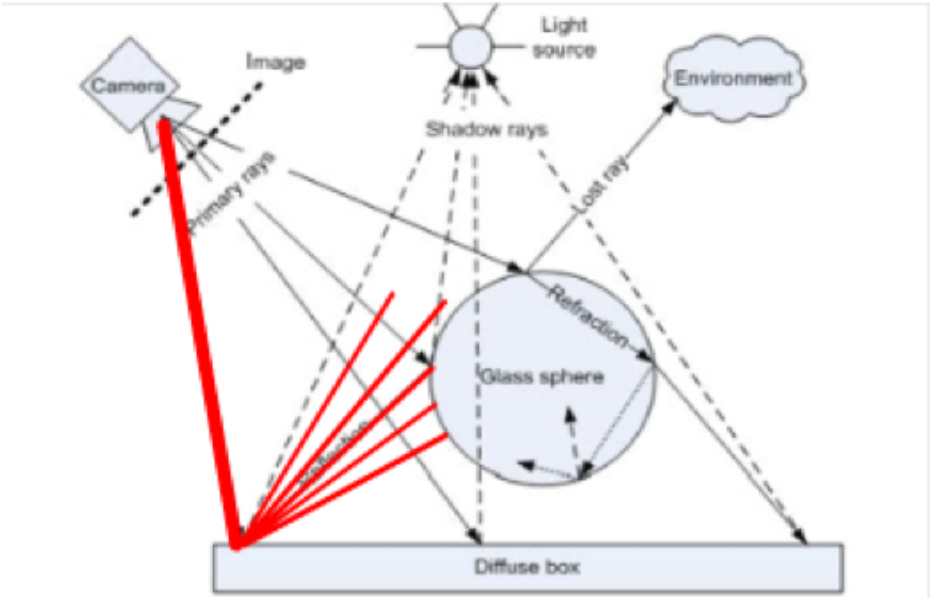
Many alternative approaches are possible: each advanced rendering engine exploits one of them.

Montecarlo techniques

Many techniques extend ray tracing: instead of sending a single ray in the mirror direction, a sampling of of the possible output directions is considered.

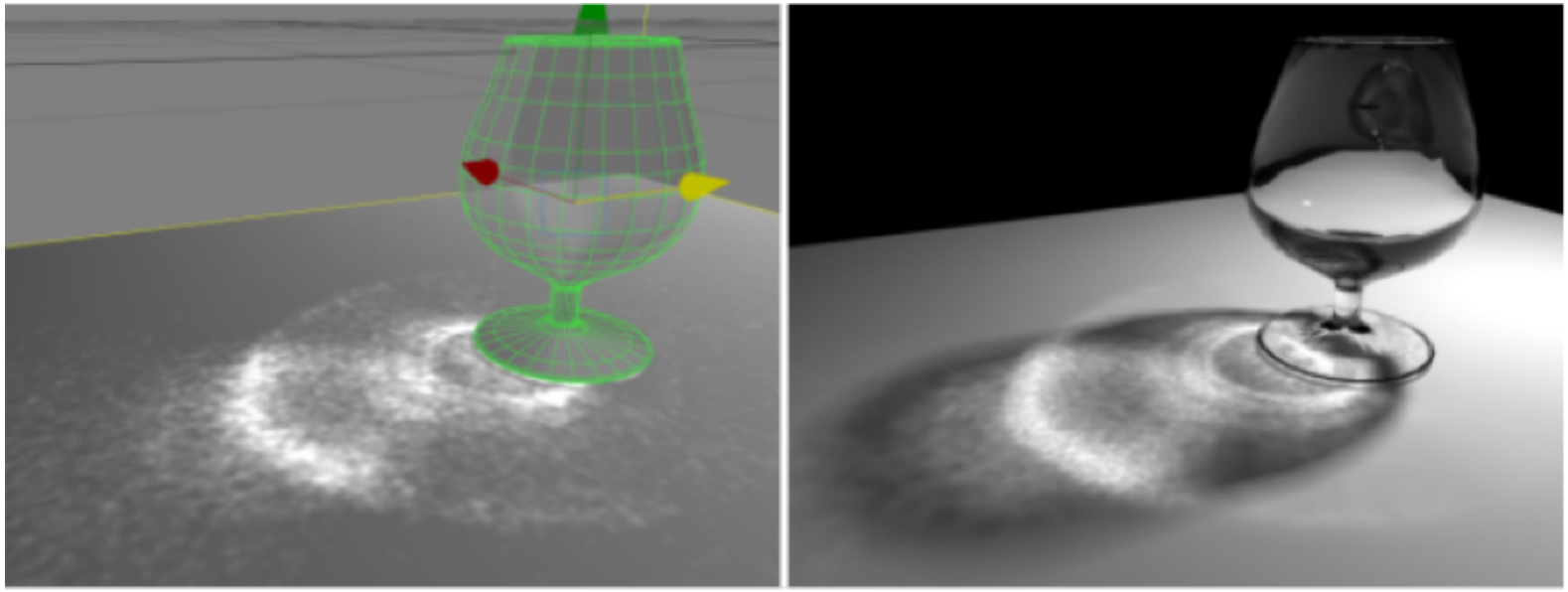
A ray is thus traced for each selected direction, and the radiance is computed using the BRDF of the considered material.





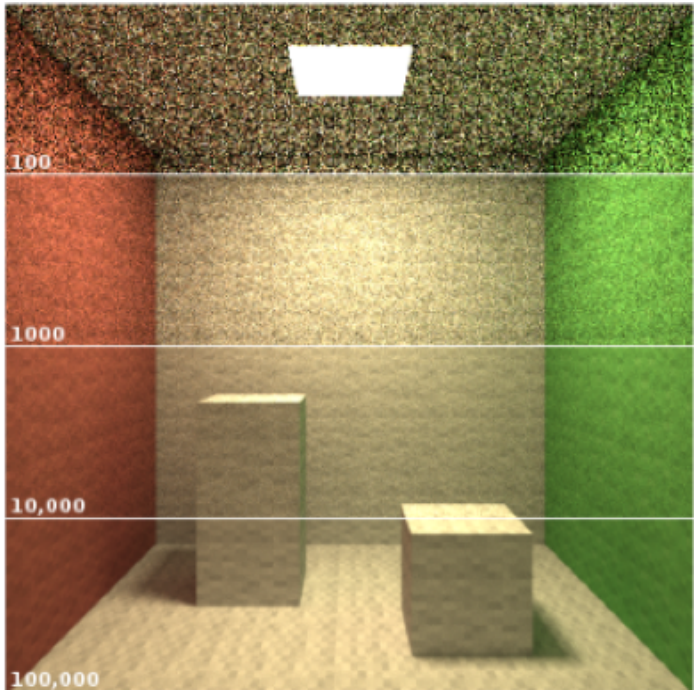
Montecarlo techniques

Photon mapping instead emulates the movements of photons in the scene, considering bounces, focalizations and other advanced phenomenon such as caustics.



Montecarlo techniques

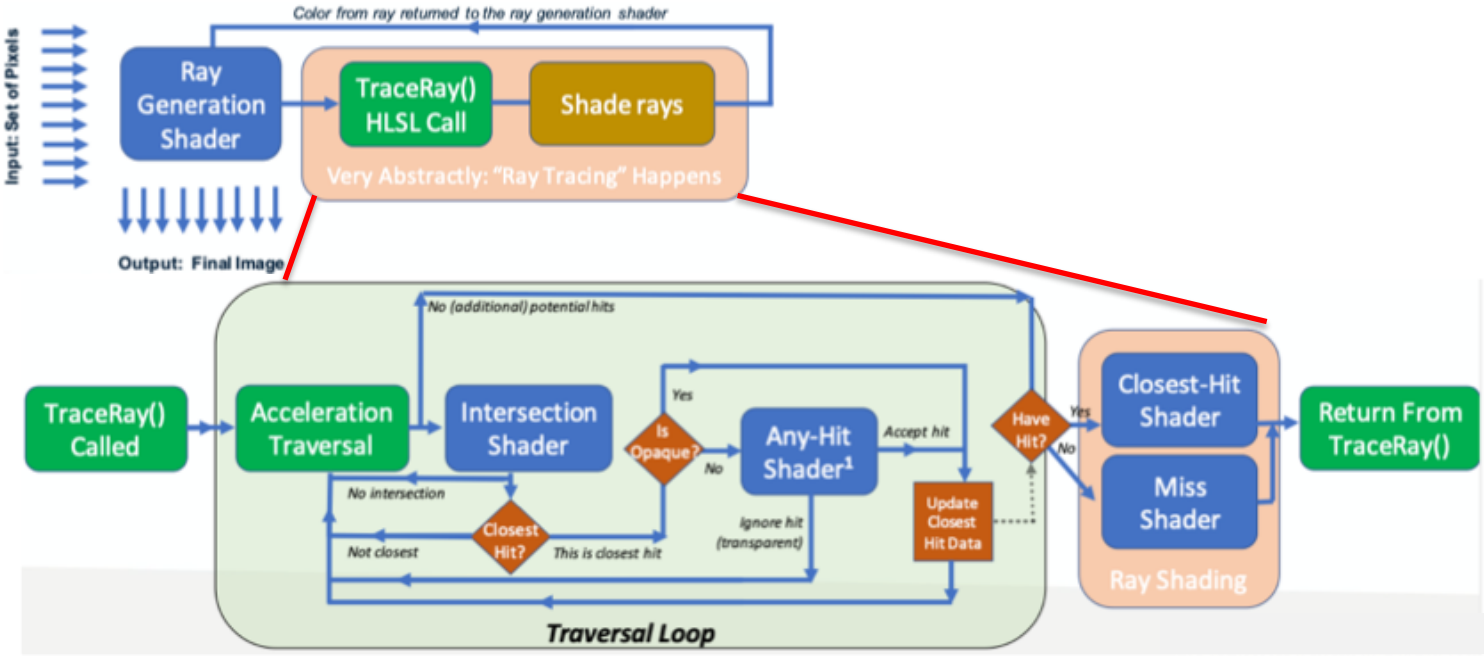
Due to the randomness that drives the techniques, *Montecarlo* based rendering algorithms tend to produce noisy images, whose effect can only be reduced by increasing the number of rays (and consequently, the rendering time).



Raytracing support in modern graphics card

Ray-tracing pipelines, allows sending several rays per pixel, aiming at performing rendering with Montecarlo techniques in real time.

Although being very accurate, they can be efficiently supported only in the top level hardware: for this reason they still cannot be considered a commodity technology.



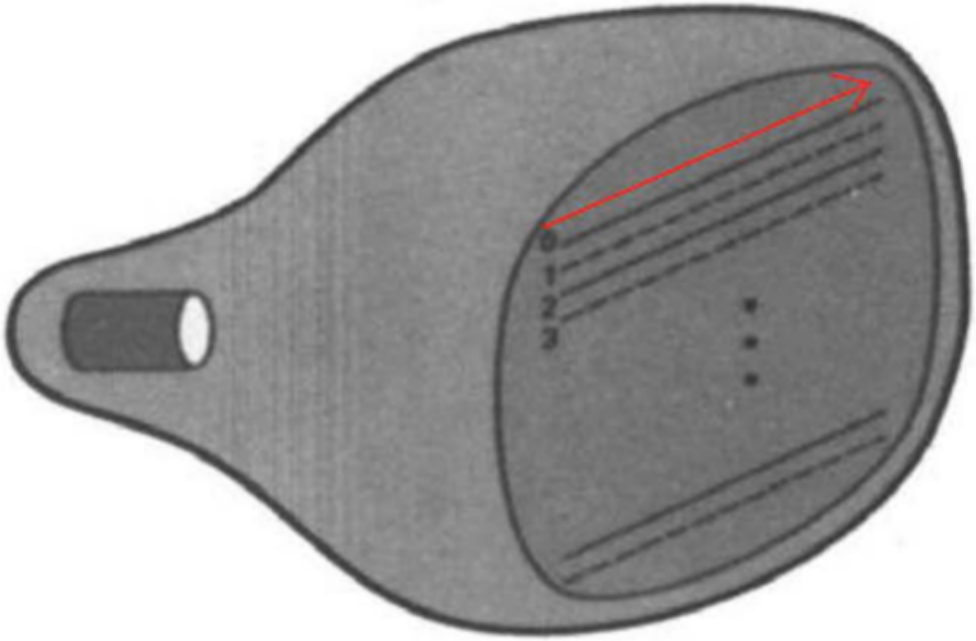
Screen Synchronization

The CPU, the GPU and the screen visualization run at different and independent speeds.

If the program is not written properly, this may cause flickering in animations.

Screen Synchronization

Monitors and display devices compose the image by updating pixels in a predefined order (usually, scanning horizontally left to right, and top to bottom).



Screen Synchronization

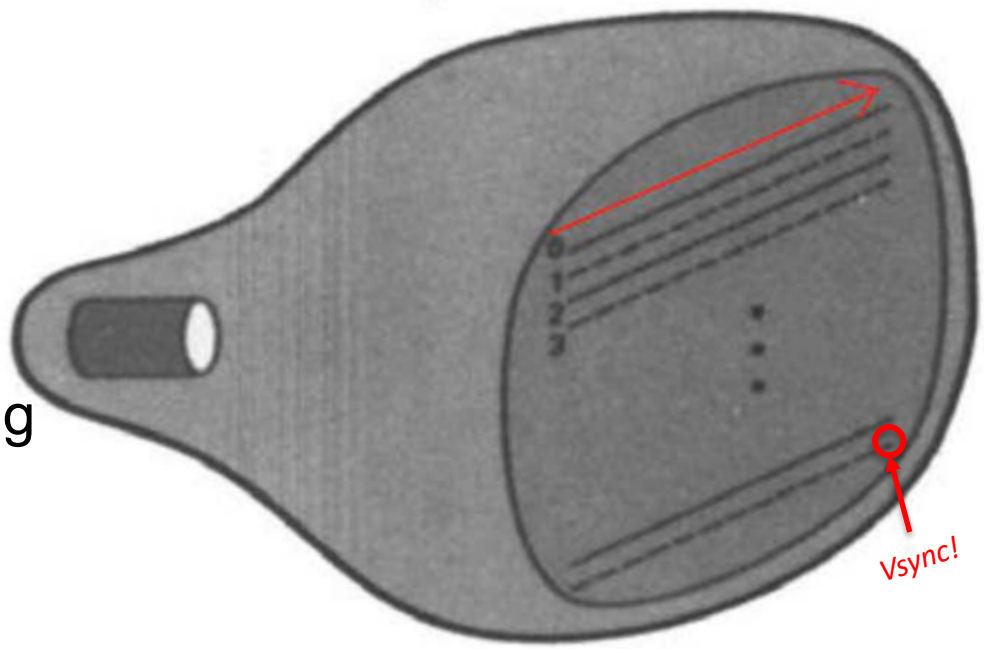
Flickering may result in a "break" into the animation **(tearing)**.

This happens because the graphic adapter reads the video memory when the program has not finished yet composing the image.

Screen Synchronization

Every graphic adapter sends an interrupt to the processor called *Vsync* whenever it finishes tracing the screen.

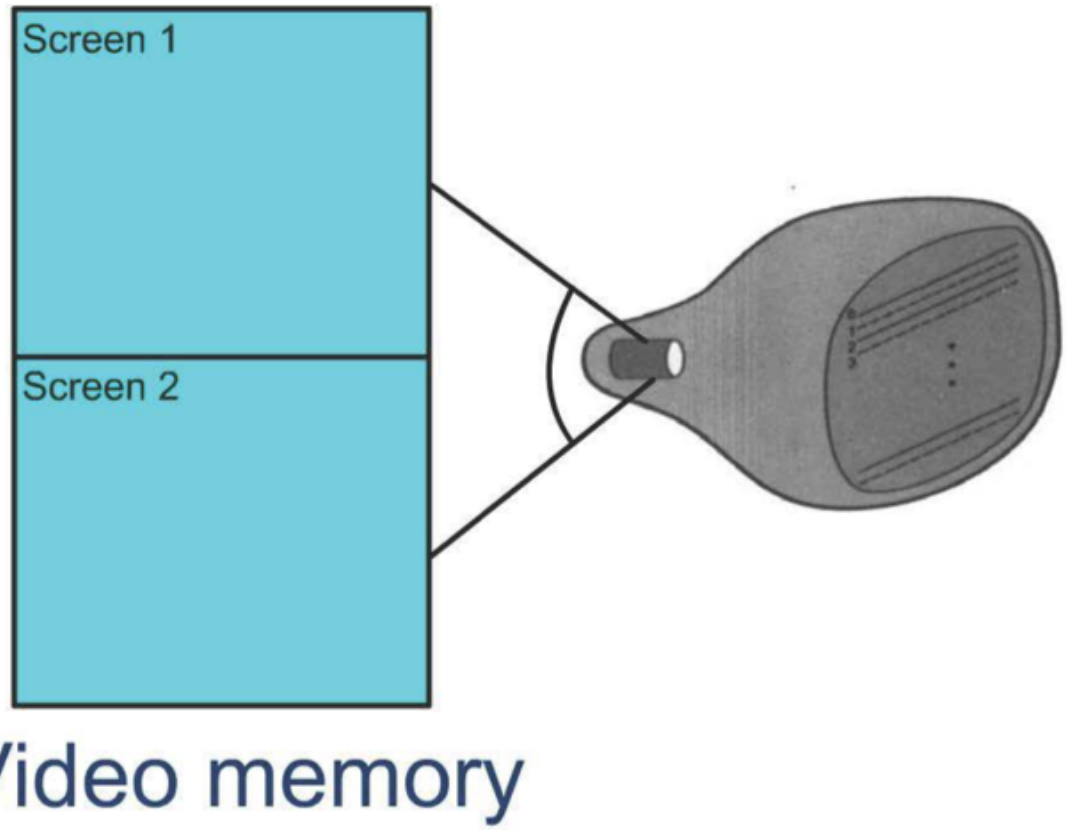
The application can intercept this interrupt, and start updating the background only when the *Vsync* signal is received.



Double Buffering

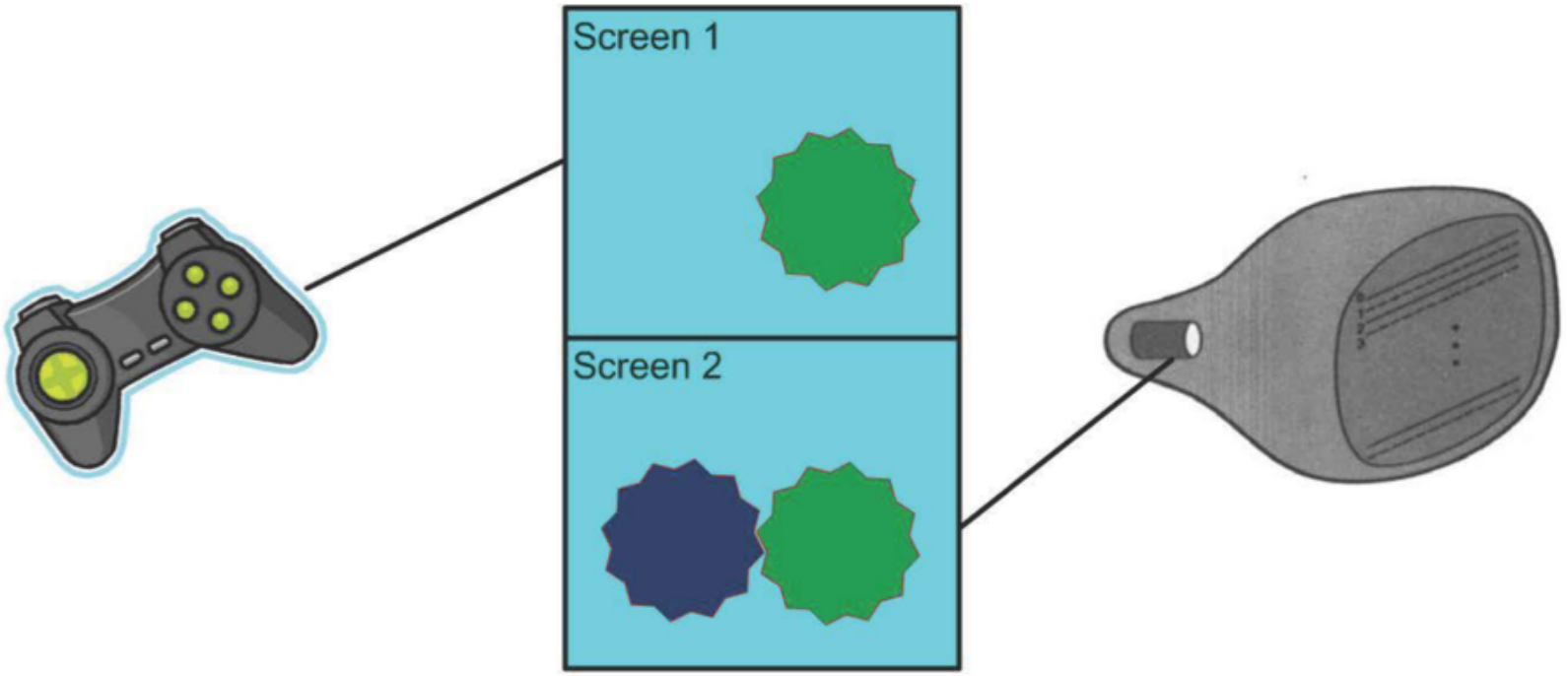
With *Double Buffering*, the video memory has two frame buffers: the *front buffer* and the *back buffer*.

While the video adapter is showing the content of one buffer, the application can compose the image in the other.



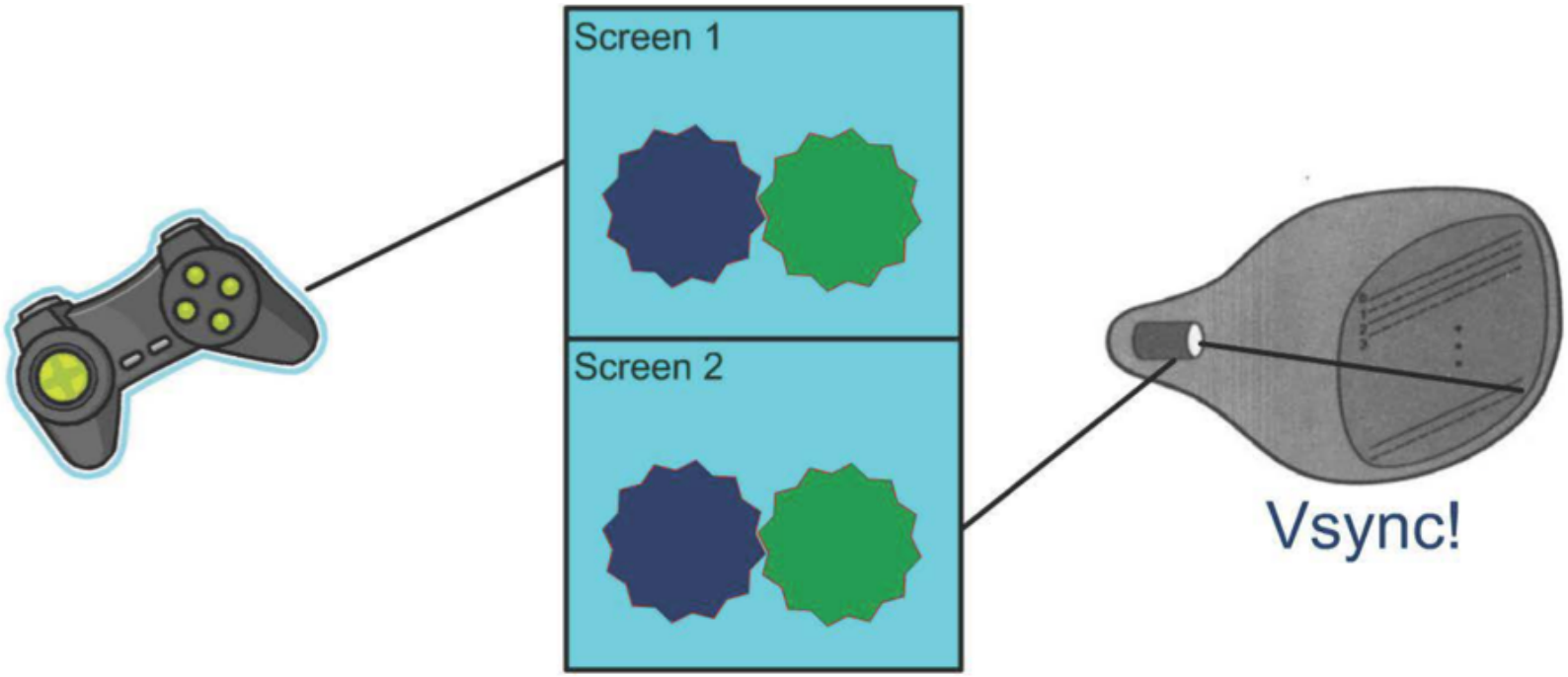
Double Buffering

In the beginning the application works on one screen (the *back buffer)*, while the video adapter is showing the other one (the *front buffer).*



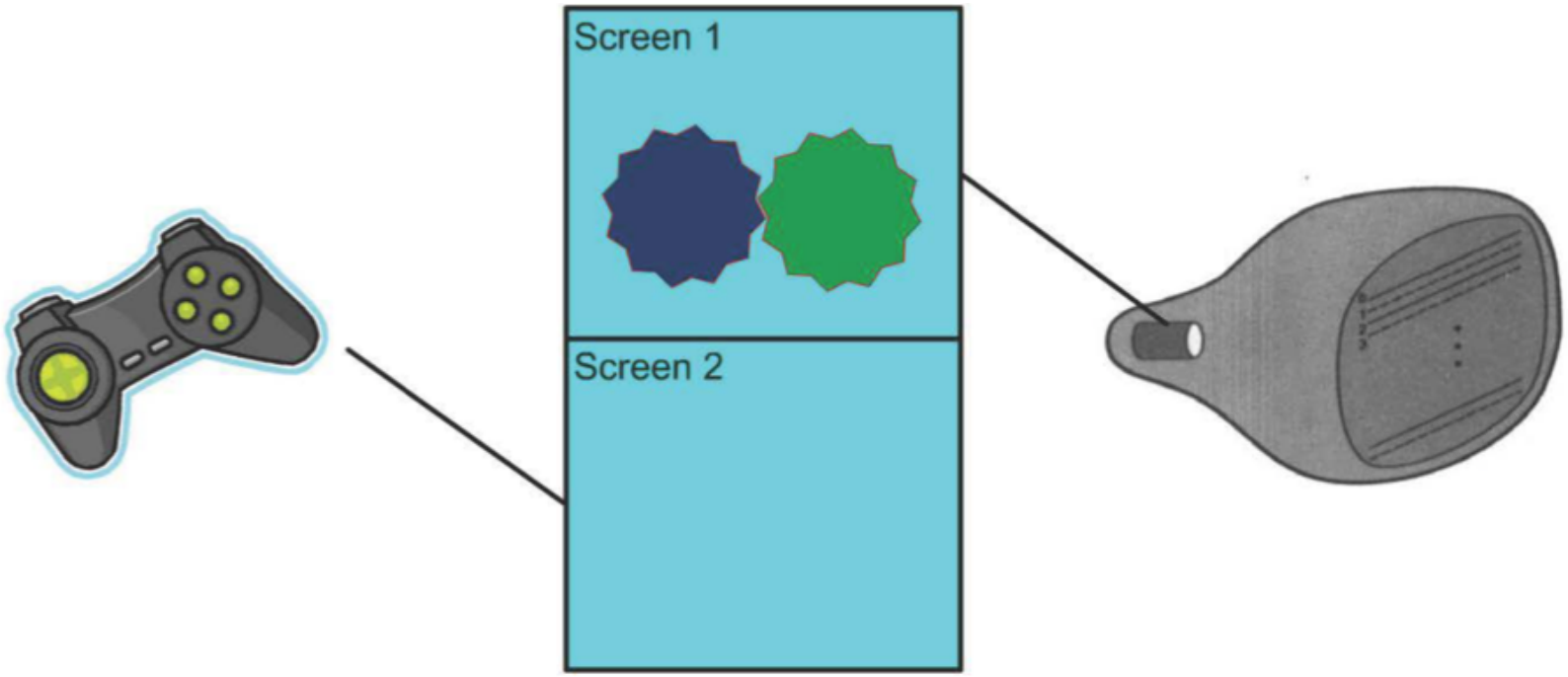
Double Buffering

As soon as the new frame has been composed, the application waits for the Vsync signal from the monitor.



Double Buffering

It then swaps the front with the back buffer, and starts composing the new frame while the monitor is showing the one just finished.



Triple Buffering

Double Buffering, although being very effective, has several drawbacks.

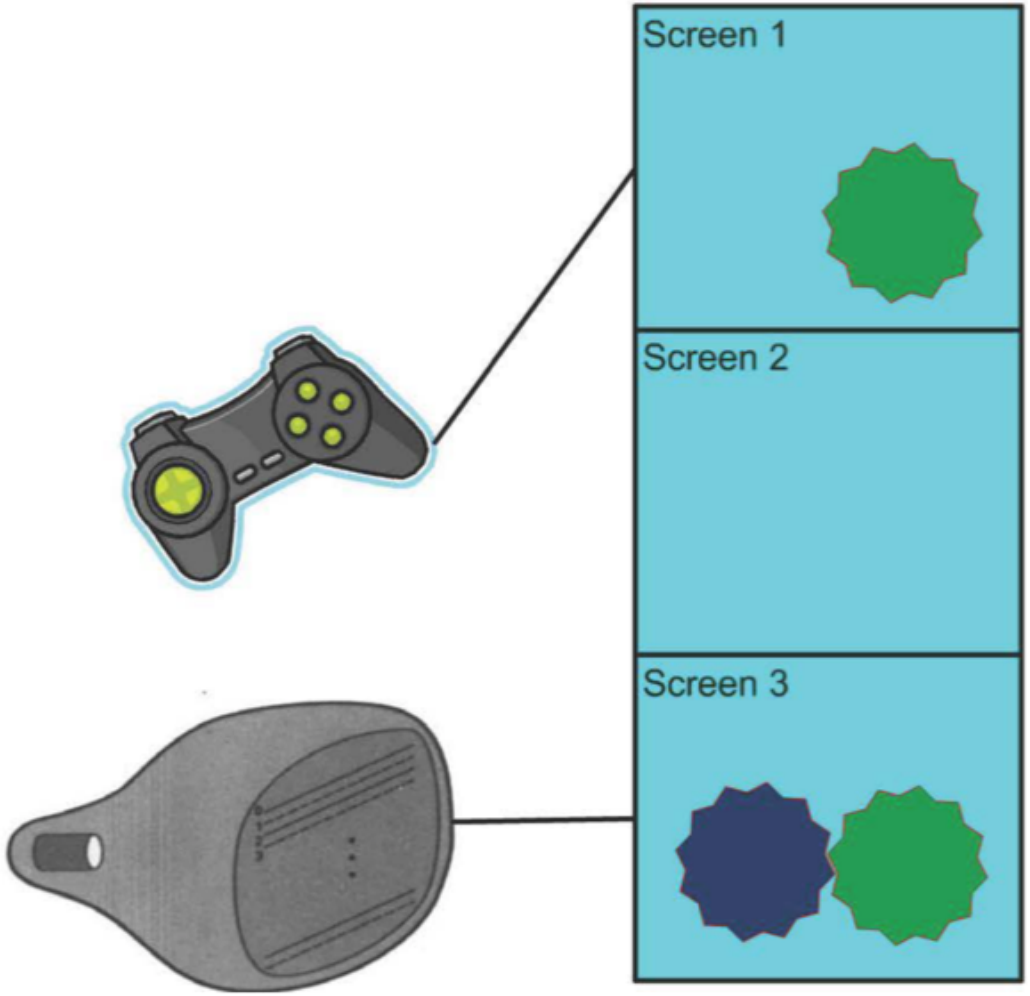
The application must stop composing the image as soon as a screen is completed, and wait for the *Vsync* interrupt to continue.

This limits the frame rate to the one of the monitor, and creates locks in the application, reducing the utilization of both the CPU and the GPU.

Triple Buffering solves this issue by allowing complete independence between the application and the presentation.

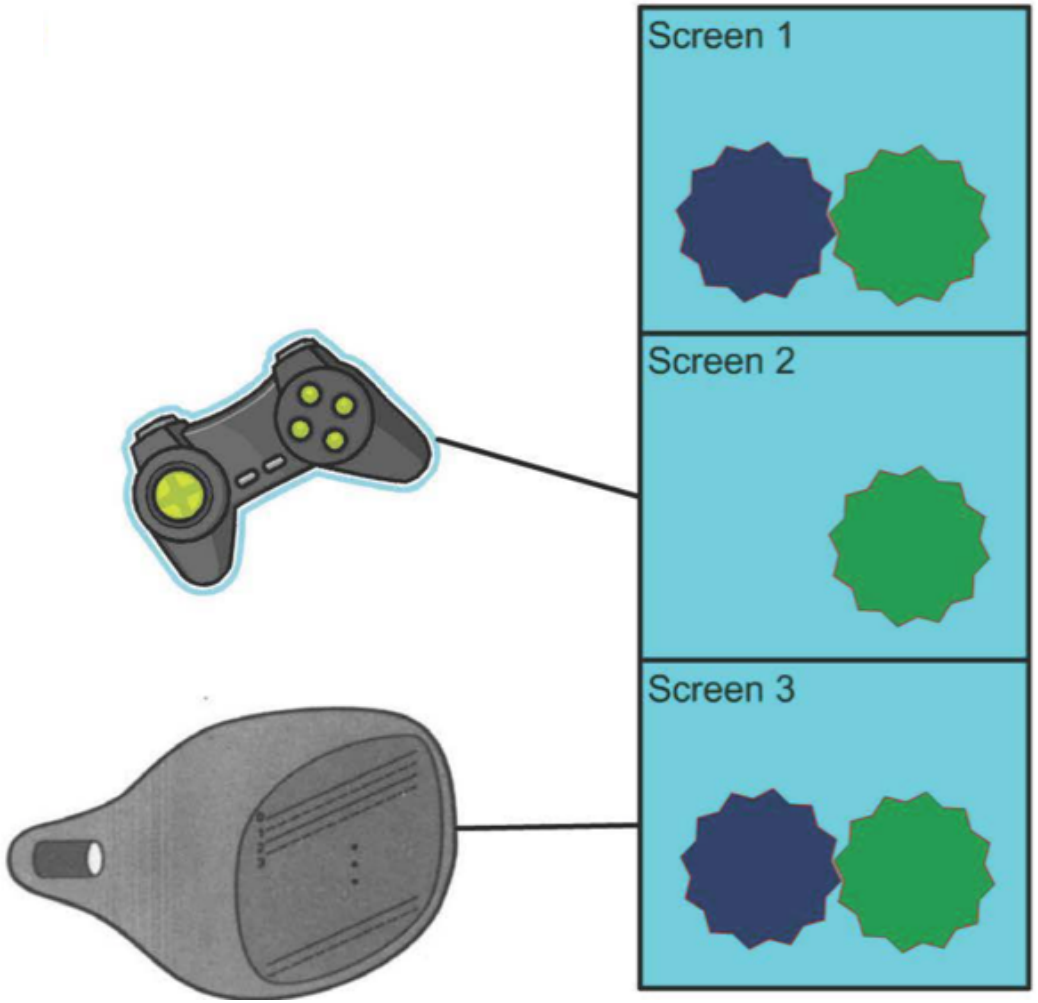
Triple Buffering

Initially, the application works on one frame buffer, while the system is displaying another.



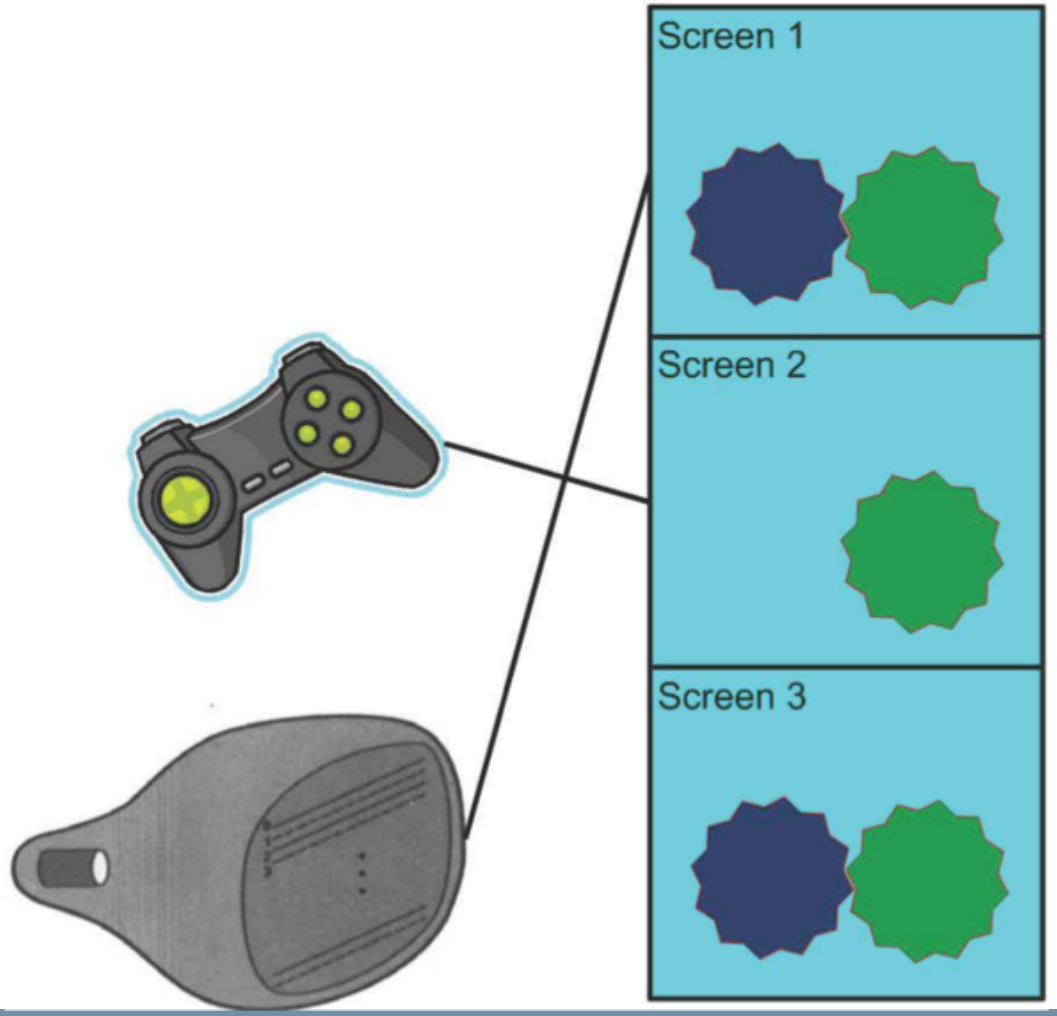
Triple Buffering

As soon as the application has finished composing the screen, it starts working on the next one in the buffer currently not used by the presentation.



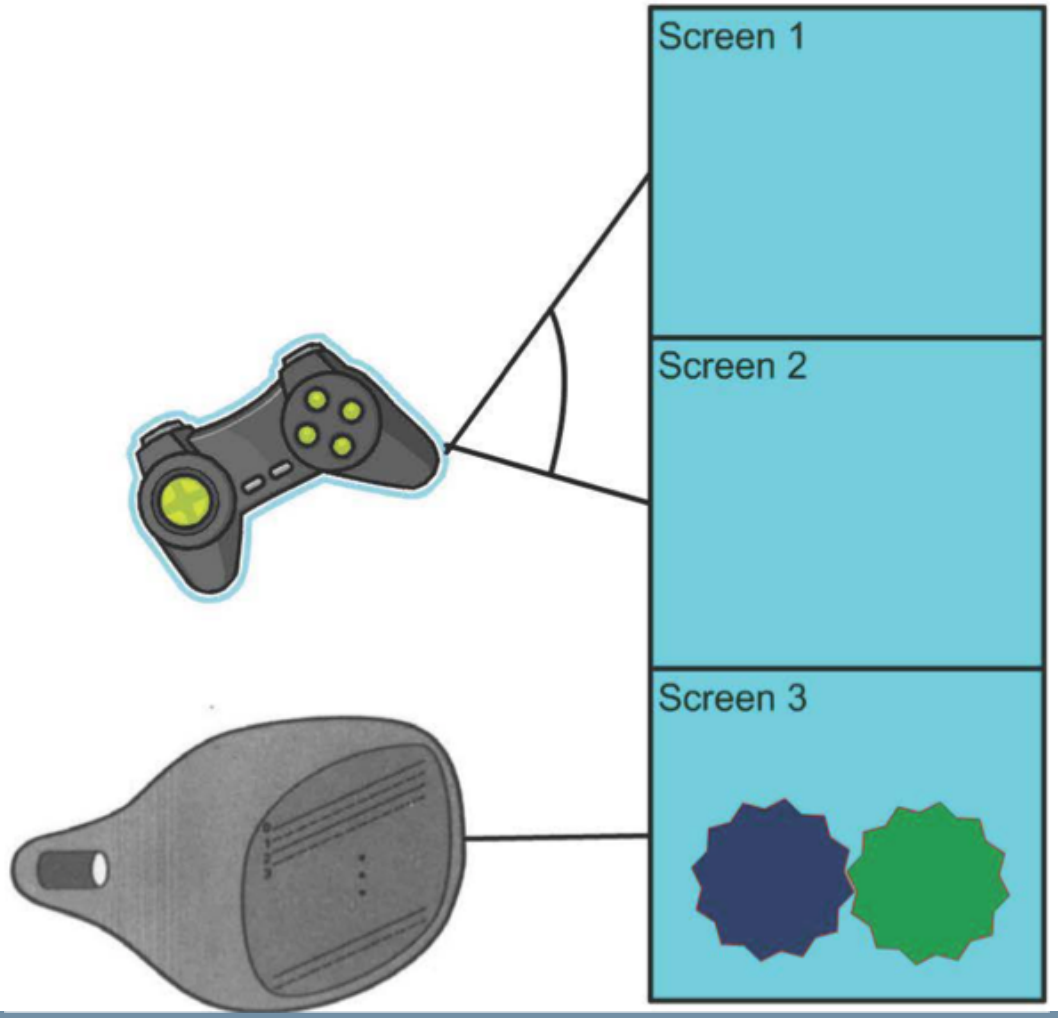
Triple Buffering

At the next *Vsync* signal, the presentation shows the last frame fully composed (the one where the application is currently not working).



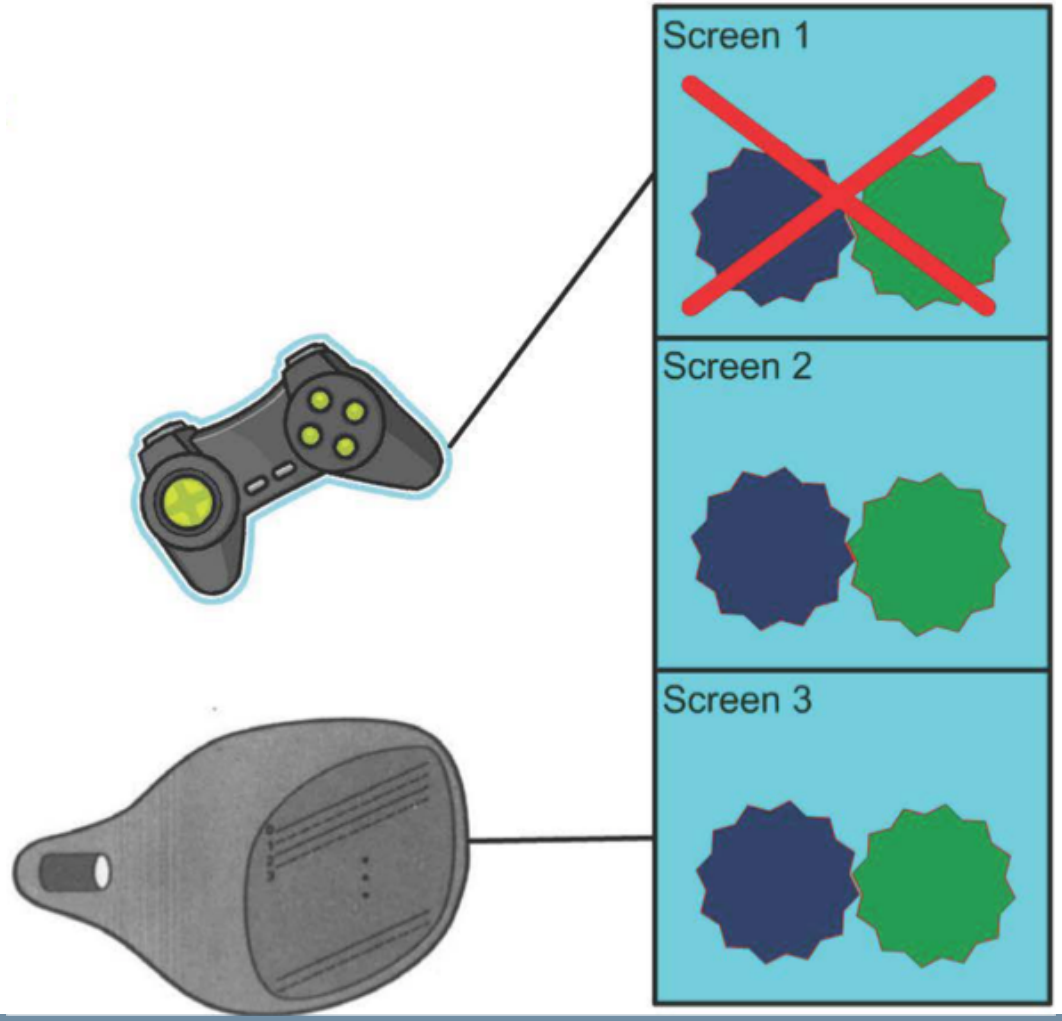
Triple Buffering

The application can switch as many times as needed between the two back buffers not currently shown, while waiting for the *vSync.*



Triple Buffering

If a frame is completed before the *vSync* is received, the screen previously generated is *skipped* (it is never displayed).



Triple Buffering

Frame skipping allows having frame rates higher than the one of the display, by automatically discarding frames not used.

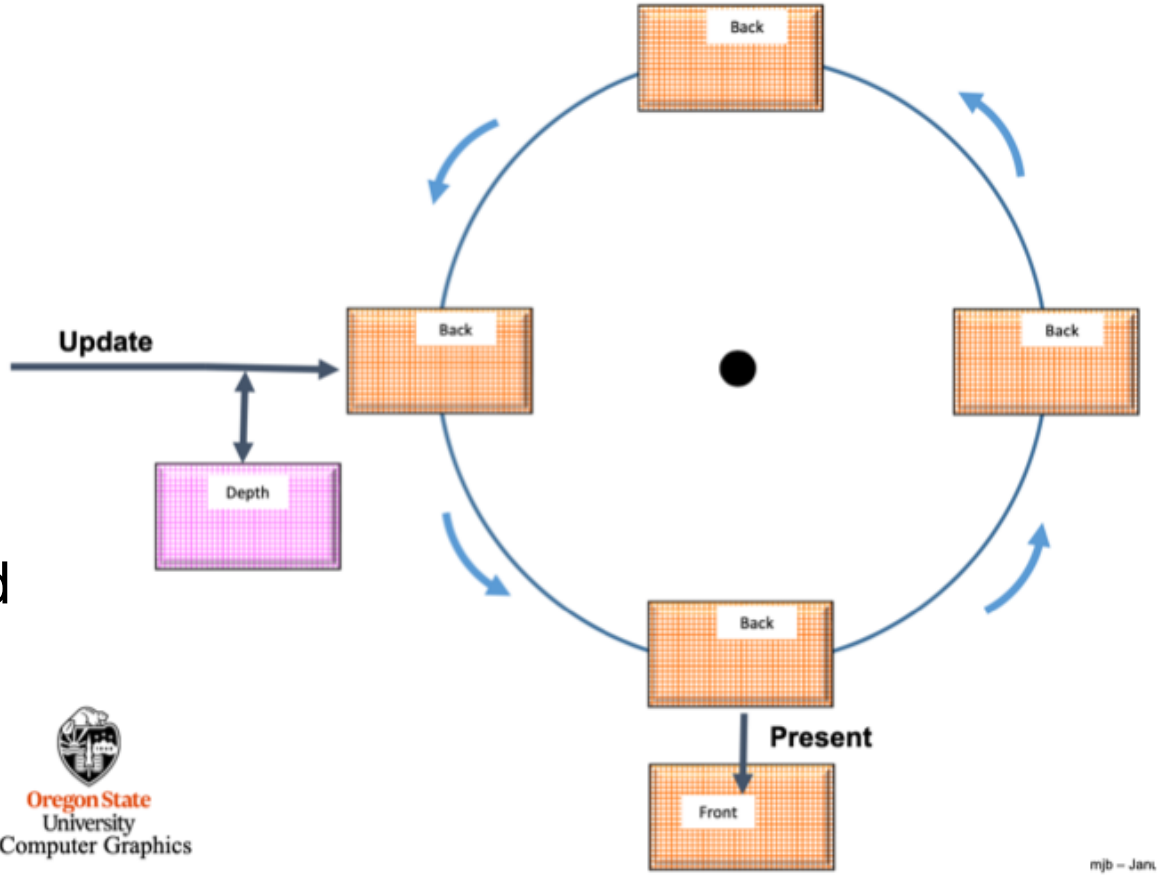
Swapping at the Vsync, allows for smooth animations and prevents the tearing effect.

The drawbacks of Triple Buffering are memory requirements (at least three frame buffers are needed), and unwise resource utilization, since both the CPU and the GPU might be wasting time (and energy) on a lot of frames that will be discarded.

The Vulkan Swap Chain

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

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



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

The Vulkan Swap Chain

To allow Vulkan being as independent as possible from the context, *Swap Chain* support must be enabled by adding the `VK_KHR_SWAPCHAIN_EXTENSION_NAME` device extension to the *Logical Device* creation procedure.

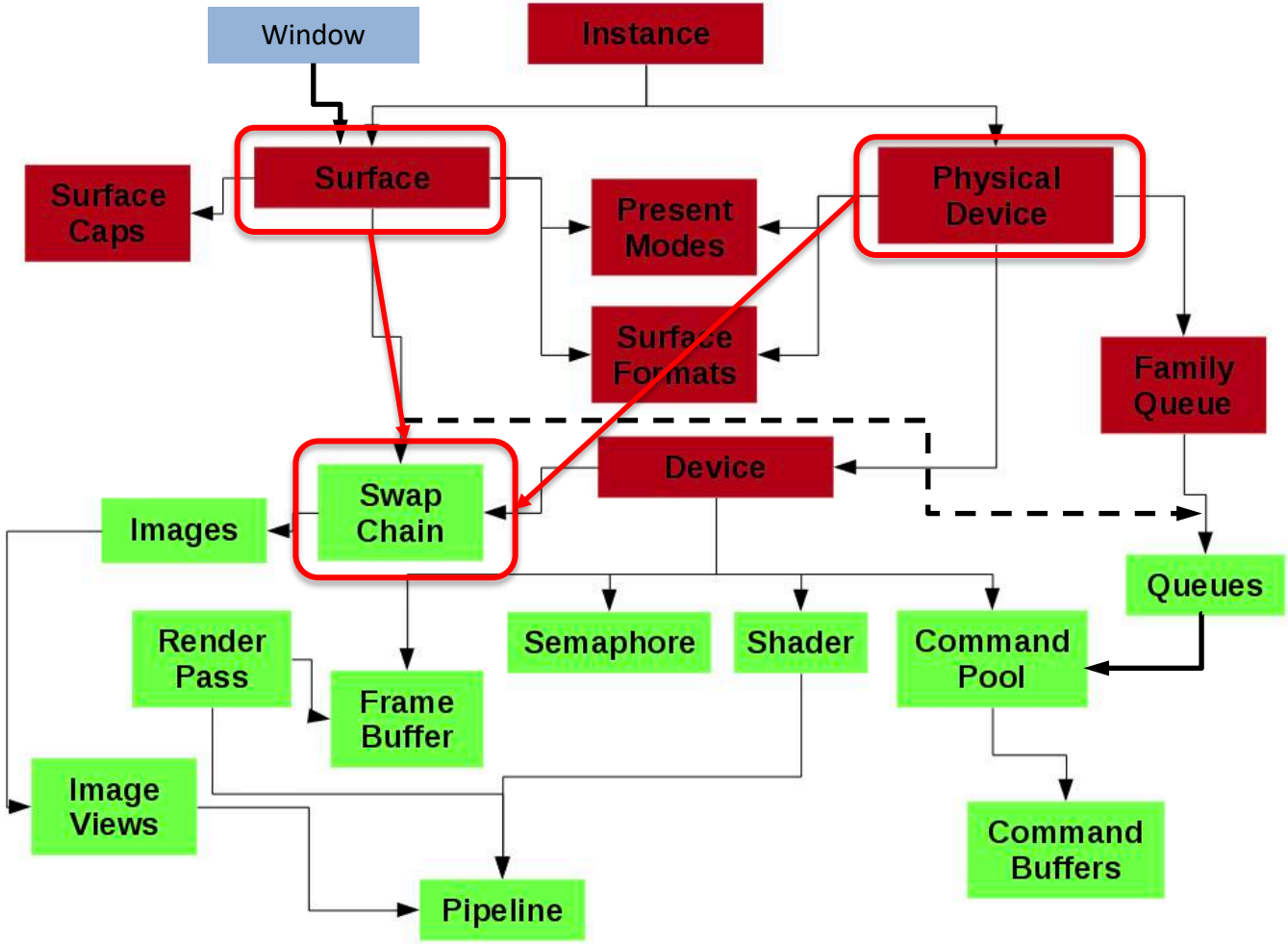
```
const std::vector<const char*> deviceExtensions = {
    VK_KHR_SWAPCHAIN_EXTENSION_NAME
};

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 =
    static_cast<uint32_t>(deviceExtensions.size());
createInfo.ppEnabledExtensionNames = deviceExtensions.data();
createInfo.enabledLayerCount = 0;

result = vkCreateDevice(physicalDevice, &createInfo,
                                nullptr, &device);
...
```

Swap Chain properties

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



Swap Chain properties

Each swap chain is characterized by:

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

Swap Chain capabilities

Swap chain capabilities include the size of the framebuffer, and the minimum and maximum number of buffers supported. They are contained in the `VkSurfaceCapabilitiesKHR` structure and queried with the `vkGetPhysicalDeviceSurfaceCapabilitiesKHR()` command.

```
VkSurfaceCapabilitiesKHR SCcapabilities;
vkGetPhysicalDeviceSurfaceCapabilitiesKHR(physicalDevice, surface,
                                          &SCcapabilities);
```

Swap Chain capabilities

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

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

Number of buffers

Screen size

Layers (for example left and right eye)

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

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

Swap Chain formats

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

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

Swap Chain formats

Swap chain formats are queried with the two-step procedure, and returned as an array of `VkSurfaceFormatKHR` objects by the `vkGetPhysicalDeviceSurfaceFormatsKHR` command.

```
uint32_t formatCount;
vkGetPhysicalDeviceSurfaceFormatsKHR(physicalDevice, surface,
        &formatCount, nullptr);

std::vector<VkSurfaceFormatKHR> SCformats;
if (formatCount != 0) {
    SCformats.resize(formatCount);
    vkGetPhysicalDeviceSurfaceFormatsKHR(physicalDevice, surface,
            &formatCount, SCformats.data());
    std::cout << "\t Supported Formats: " << formatCount << "\n";
    for(int i; i < formatCount; i++) {
          std::cout << "\t\tFormat: " << SCformats[i].format <<
                       ", Color Space:" << SCformats[i].colorSpace <<"\n";
    }
}
```

Swap Chain formats

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

```
uint32_t formatCount;
vkGetPhysicalDeviceSurfaceFormatsKHR(physicalDevice, surface,
        &formatCount, nullptr);

std::vector<VkSurfaceFormatKHR> SCformats;
if (formatCount != 0) {
    SCformats.resize(formatCount);
    vkGetPhysicalDeviceSurfaceFormatsKHR(physicalDevice, surface,
            &formatCount, SCformats.data());
    std::cout << "\t Supported Formats: " << formatCount << "\n";
    for(int i; i < formatCount; i++) {
        std::cout << "\t\tFormat: " << SCformats[i].format <<
                     ", Color Space:" << SCformats[i].colorSpace <<"\n";
    }
}
```

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

Swap Chain Presentation Modes

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

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

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

Swap Chain Presentation Modes

Four main presentation modes are supported:

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

Single Buffer

Triple Buffer

Double Buffer

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

Swap Chain Creation

Swap chains are created by filling the `VkSwapchainKHR` structure, and calling the `vkCreateSwapchainKHR` command.

```
VkSwapchainKHR swapChain;

VkSwapchainCreateInfoKHR SCcreateInfo{};
SCcreateInfo.sType =
                VK_STRUCTURE_TYPE_SWAPCHAIN_CREATE_INFO_KHR;
SCcreateInfo.surface = surface;
SCcreateInfo.minImageCount = imageCount;
SCcreateInfo.imageFormat = surfaceFormat.format;
SCcreateInfo.imageColorSpace = surfaceFormat.colorSpace;
SCcreateInfo.imageExtent = extent;
SCcreateInfo.imageArrayLayers = 1;
SCcreateInfo.imageUsage = VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT;

... -> DEFINED LATER!

SCcreateInfo.preTransform = SCcapabilities.currentTransform;
SCcreateInfo.compositeAlpha =
                            VK_COMPOSITE_ALPHA_OPAQUE_BIT_KHR;
SCcreateInfo.presentMode = presentMode;
SCcreateInfo.clipped = VK_TRUE;
SCcreateInfo.oldSwapchain = VK_NULL_HANDLE;

VkResult result = vkCreateSwapchainKHR(device, &SCcreateInfo,
                                       nullptr, &swapChain);
if (result != VK_SUCCESS) {
     throw std::runtime_error("failed to create swap chain!");
```

Swap Chain Creation

Creation requires selecting the number of images in the buffers, their format, the color space, the extent, and the presentation mode.

These parameters are device dependent, and must be decided analyzing the swap chain properties previously described.

```
VkSwapchainKHR swapChain;

VkSwapchainCreateInfoKHR SCcreateInfo{};
SCcreateInfo.sType =
                VK_STRUCTURE_TYPE_SWAPCHAIN_CREATE_INFO_KHR;
SCcreateInfo.surface = surface;
SCcreateInfo.minImageCount = imageCount;
SCcreateInfo.imageFormat = surfaceFormat.format;
SCcreateInfo.imageColorSpace = surfaceFormat.colorSpace;
SCcreateInfo.imageExtent = extent;
SCcreateInfo.imageArrayLayers = 1;
SCcreateInfo.imageUsage = VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT;

... -> DEFINED LATER!

SCcreateInfo.preTransform = SCcapabilities.currentTransform;
SCcreateInfo.compositeAlpha =
                            VK_COMPOSITE_ALPHA_OPAQUE_BIT_KHR;
SCcreateInfo.presentMode = presentMode;
SCcreateInfo.clipped = VK_TRUE;
SCcreateInfo.oldSwapchain = VK_NULL_HANDLE;

VkResult result = vkCreateSwapchainKHR(device, &SCcreateInfo,
                                       nullptr, &swapChain);
if (result != VK_SUCCESS) {
     throw std::runtime_error("failed to create swap chain!");
```

Swap Chain Creation

The Vulkan tutorial suggest a few procedure for selecting these parameter in a meaningful way: in this course, we will not see them in detail, and we will simply exploit them!

```
VkSurfaceFormatKHR surfaceFormat =
                chooseSwapSurfaceFormat(SCformats);
VkPresentModeKHR presentMode =
                chooseSwapPresentMode(SCpresentModes);
VkExtent2D extent = chooseSwapExtent(SCcapabilities, window);

uint32_t imageCount = SCcapabilities.minImageCount + 1;

if (SCcapabilities.maxImageCount > 0 &&
          imageCount > SCcapabilities.maxImageCount) {
     imageCount = SCcapabilities.maxImageCount;
}

// chooseSwapSurfaceFormat()
// chooseSwapPresentMode()
// chooseSwapExtent()
// Can all be copied and pasted from either the Vulkan tutorial
// or Exanple 09
```

Swap Chain Creation

Other parameters specify that swap chain images will be used for rendering (`imageUsage`), what should be done with the alpha channel if present (`compositeAlpha`), the rotation (`preTransform`, which can be taken from the current surface parameter), and the number of layers used (`imageArrayLayers`).

```
VkSwapchainKHR swapChain;

VkSwapchainCreateInfoKHR SCcreateInfo{};
SCcreateInfo.sType =
                VK_STRUCTURE_TYPE_SWAPCHAIN_CREATE_INFO_KHR;
SCcreateInfo.surface = surface;
SCcreateInfo.minImageCount = imageCount;
SCcreateInfo.imageFormat = surfaceFormat.format;
SCcreateInfo.imageColorSpace = surfaceFormat.colorSpace;
SCcreateInfo.imageExtent = extent;
SCcreateInfo.imageArrayLayers = 1;
SCcreateInfo.imageUsage = VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT;

... -> DEFINED IN THE NEXT SLIDE!

SCcreateInfo.preTransform = SCcapabilities.currentTransform;
SCcreateInfo.compositeAlpha =
                           VK_COMPOSITE_ALPHA_OPAQUE_BIT_KHR;
SCcreateInfo.presentMode = presentMode;
SCcreateInfo.clipped = VK_TRUE;
SCcreateInfo.oldSwapchain = VK_NULL_HANDLE;

VkResult result = vkCreateSwapchainKHR(device, &SCcreateInfo,
                                       nullptr, &swapChain);
if (result != VK_SUCCESS) {
     throw std::runtime_error("failed to create swap chain!");
```

Swap Chain Creation

The Swap Chain needs also the pointers to the presentation and graphic queues, if they are different.

For this reason a special procedure should be implemented for supporting this case.

```
VkSwapchainKHR swapChain;

…
SCcreateInfo.imageArrayLayers = 1;
SCcreateInfo.imageUsage = VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT;

uint32_t queueFamilyIndices[] =
          {aQueueWithGraphicsCapability.value(),
           aQueueWithPresentationCapability.value()};
if (aQueueWithPresentationCapability !=
                aQueueWithPresentationCapability) {
     SCcreateInfo.imageSharingMode =
                          VK_SHARING_MODE_CONCURRENT;
     SCcreateInfo.queueFamilyIndexCount = 2;
     SCcreateInfo.pQueueFamilyIndices = queueFamilyIndices;
} else {
     SCcreateInfo.imageSharingMode =
                          VK_SHARING_MODE_EXCLUSIVE;
     SCcreateInfo.queueFamilyIndexCount = 0;
     SCcreateInfo.pQueueFamilyIndices = nullptr;
}

SCcreateInfo.preTransform = SCcapabilities.currentTransform;
SCcreateInfo.compositeAlpha =
                          VK_COMPOSITE_ALPHA_OPAQUE_BIT_KHR;
…
```

Swap Chain release

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

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

Swap Chain Images retrieval

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

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

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

Image Views

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

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

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

Swap chain Image Views retrieval

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

```
std::vector<VkImageView> swapChainImageViews;
swapChainImageViews.resize(swapChainImages.size());

for (size_t i = 0; i < swapChainImages.size(); i++) {
     VkImageViewCreateInfo viewInfo{};
     viewInfo.sType = VK_STRUCTURE_TYPE_IMAGE_VIEW_CREATE_INFO;
     viewInfo.image = swapChainImages[i];
     viewInfo.viewType = VK_IMAGE_VIEW_TYPE_2D;
     viewInfo.format = surfaceFormat.format;
     viewInfo.subresourceRange.aspectMask =
                                        VK_IMAGE_ASPECT_COLOR_BIT;
     viewInfo.subresourceRange.baseMipLevel = 0;
     viewInfo.subresourceRange.levelCount = 1;
     viewInfo.subresourceRange.baseArrayLayer = 0;
     viewInfo.subresourceRange.layerCount = 1;

     VkImageView imageView;

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

Swap chain Image Views retrieval

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

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

```
std::vector<VkImageView> swapChainImageViews;
swapChainImageViews.resize(swapChainImages.size());

for (size_t i = 0; i < swapChainImages.size(); i++) {
     VkImageViewCreateInfo viewInfo{};
     viewInfo.sType = VK_STRUCTURE_TYPE_IMAGE_VIEW_CREATE_INFO;
     viewInfo.image = swapChainImages[i];
     viewInfo.viewType = VK_IMAGE_VIEW_TYPE_2D;
     viewInfo.format = surfaceFormat.format;
     viewInfo.subresourceRange.aspectMask =
                                        VK_IMAGE_ASPECT_COLOR_BIT;
     viewInfo.subresourceRange.baseMipLevel = 0;
     viewInfo.subresourceRange.levelCount = 1;
     viewInfo.subresourceRange.baseArrayLayer = 0;
     viewInfo.subresourceRange.layerCount = 1;

     VkImageView imageView;

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

Swap chain Image Views release

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

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