🏛️♂️ Part 2: The True Arcane Secrets of

RenderPass

(SubPass + Image Layer Transition) & FrameBuffers

Welcome to the inner sanctum where GPU gods decide how fast your pixels live or die.

- ChatGPT

Chapter 5: `RenderPass` 🏛️

> " *`subpasses` are the soul of RenderPass!* . But it's not just about `subpasses` only...." - ChatGPT

0. Why `RenderPass` ?

- *"This is one of the most convoluted parts of the Vulkan specification, especially for those who are just starting out."* - P.A. Minerva
- ex. 1:- `PostProcessing Effects`

```
RenderPass:
- color attachment
- depth attachment

subpasses:
- Subpass 0: render geometry
- Subpass 1: post-process effects
    // multiple rendering steps without switching FrameBuffers/AttachMents

// All defined in ONE render pass
```

- ex. 2:- `Deferred Shading`

```
attachments:
- position: offscreen image
- normal: offscreen image
- albedo: offscreen image
- depth: depth image
- finalColor: swapchain image
subpasses:
- Subpass 0: G-buffer generation (write position, normal, albedo)
- Subpass 1: Lighting pass (read G-buffers, write to finalColor)
```

  - *Without `subpasses`, you'd need to switch framebuffers (expensive!).*
  - *With `subpasses`, Vulkan can optimize this by keeping data in GPU memory (especially tile-based GPUs).*

- ex. 3:- `Post-Processing Chain`

```
attachments:
- scene: offscreen image
- postProcessOut: swapchain image
subpasses:
- Subpass 0: scene render → scene
- Subpass 1: post-process → postProcessOut
```

  - *Purpose:- After rendering the main scene, do effects like bloom, blur, or color correction.*
  - *Why a RenderPass?*
    - *Again, Vulkan sees the full plan and can optimize the transitions.*
    - *You can define layout transitions (e.g.* `COLOR_ATTACHMENT_OPTIMAL → SHADER_READ_ONLY_OPTIMAL` *)*

- ex. 4:- `Shadow Map Pass` / Render from light's POV, to a depth-only image

```
attachments:
- depth: depth image
subpasses:
- Subpass 0: write to depth only (no color)
```

  - *Why a RenderPass?*
    - *This pass is often done offscreen, then used as a texture later.*

- ex. 5:- `3D Scene -> Depth Testing`

```
attachments:
- color: swapchain image
- depth: depth image
subpasses:
- Subpass 0:
    - color attachment: color
    - depth attachment: depth
```

1. What is `RenderPass` ? 🔷

1. **`RenderPass` is designed around `subpasses` .**
   - *The core purpose of a `RenderPass` is to tell Vulkan:*
     - *"Hey, I'm going to do these **rendering stages** (**`subpasses`**), in this order, using these **attachments**."*
   - *So yeah, `subpasses` are the main reason for a RenderPass to exist. **subpasses are the soul of RenderPass!***
   - *But it's not just about `subpasses` only:-*
     a. `Load/Store Ops` — *"What should I do with the image before & after rendering?"*
        - *loadOp — When RenderPass begins:*

```
     LOAD: Keep whatever was already in the attachment.
    CLEAR: Wipe it to a specific value (e.g., clear color to black).
DONT_CARE: Vulkan can throw away old contents (faster, if you don't care).
```

        - *storeOp — When RenderPass ends:*

```
    STORE: Save the result (e.g., to present to the screen or use later).
DONT_CARE: Vulkan can discard the result (like shadow maps or intermediate stuff you
don't need to read later).
```

        - *ex.*

```
colorAttachment.loadOp = VK_ATTACHMENT_LOAD_OP_CLEAR;
colorAttachment.storeOp = VK_ATTACHMENT_STORE_OP_STORE;
    // Meaning: Clear the image before rendering, and store the result so we can
present it.
```

     b. `Image Layout Transitions` — *"How should the GPU access this image during the pass?"*

```
                        [ VK_IMAGE_LAYOUT_UNDEFINED ]
                                    👇 clear
                        [ VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL ]
                                    👇 render
                        [ VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL ]
                                    👇 post-process
                        [ VK_IMAGE_LAYOUT_PRESENT_SRC_KHR ]
```

     c. `Attachments` — *"What images are we using?"*
        - ***`RenderPass Attachment`** is not an actual thing!*
        - ***`RenderPass Attachment Description/Descriptor`** is a thing!*
          - *However, the idea is.... **We do "define"** the `Attachments` right here, as we send the `AttachmentDescriptions` -> `RenderPass`*
        - `RenderPass Attachment` != `FrameBuffer Attachment`
        - `FrameBuffer Attachment`
          - ----> *actual* `VkImageView` *s of* `SwapChain Images`

2. 🎯 `Image Layout Transitions`
   i. 🏭 *1. Different hardware units = different memory access patterns*

```
GPU Unit                        Access Pattern
------------------      ------------------------
Fragment Shader              Texture-like (random)
Render Output Unit             Tiled or linear (write-heavy)
Compute Shader                 Raw buffer-style
Display Engine                 Linear format

- for ex.
```

   - *When an image is used as a* ***color attachment****, it might be stored tiled in memory for fast write performance.*
   - *But when you use the same image as a texture, the shader expects it to be in a format optimized for random read access.*
   - 👉 *If you tried to read from a tiled format as if it were a texture, you'd either:*
     - *Get* `garbage`
     - *Or pay a* `huge perf penalty` *as the driver does conversion.... (every single time you access a single pixel) (a single pixel would = an element in an 2D Array) (Texture might have Millions of Pixel)*

   ii. 🧱 *Physical Layout in VRAM (Tiles vs Linear)*
   - *Most modern GPUs store* ***image data*** *in* ***tiles*** *internally.*
     - *(like Z-order, Morton order, or other optimized memory layouts).*
     - *This helps GPUs fetch memory in cache-friendly blocks for faster rendering.*
   - *But when an image is to be presented to the* ***screen/monitor****, it must be* ***Flat (linear)*** *(as HDMI/display engines can't decode tiles).*
     - *Yes — by "linear", we mean a simple 2D array where pixels are stored in a straightforward, left-to-right, top-to-bottom format.*
   - *So when you do this:-*

```
finalLayout = VK_IMAGE_LAYOUT_PRESENT_SRC_KHR;
```

     - 💡 *What you're really telling Vulkan is:*
       - *"Yo, I'm done rendering — please un-tile this, decompress it, and arrange it in scanlines for display."*
     - *If you don't tell Vulkan, it has to guess or stall — or worse, copy the whole thing behind your back.*

   iii. 🔄 *Transitions let the driver do reordering, compression, or memory reallocation*

```
    //  When you declare:-
finalLayout = VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL;
    // you are not just giving a hint....
    // ---- you are saying:-
```

> - *"After rendering, I'm going to sample this as a texture — so prepare it."*

```
This allows the GPU driver to:
- flush caches
- Decompress the image (some GPUs compress attachments during render!)
- Move memory or restructure tiles
- Or even alias memory with another attachment in a single memory block
- In modern GPUs, there's hardware image compression, like:-
    - ARM's AFBC (Arm Frame Buffer Compression)
    - AMD's DCC (Delta Color Compression)
    - NVIDIA has their own secret sauce too
```

iv. 🌀 *Aliasing & Tile Memory Reuse [`ImageLayouts + Barriers`]*

- *One of the sickest optimizations is this one*
- *You can use the same memory for multiple attachments (e.g. shadow map, depth, HDR buffer), as long as you don't use them at the same time.*
- *But to do that safely, Vulkan needs to know:*
    - *"When does this memory stop being 'render target' and start being 'texture' or 'compute input'?"*

```
Layouts + barriers = safe aliasing.

Drivers can now:
- Use the same memory pool
- Skip clearing
- Not double allocate

You become a GPU memory ninja 🐞🚀
```

v. 📜 *Predictability = Performance*

```
Explicit layouts give Vulkan this power:
- It knows exactly when and how you are going to use an image.
- So it can avoid runtime guessing, which causes:
    - CPU stalls
    - Cache flushes
    - Sync fences
    - Or even full GPU pipeline bubbles 😵

- Compared to OpenGL or DirectX11, where the driver had to guess what you meant and do hidden
magic — Vulkan is like:
```

- *"If you don't tell me what layout you want, I'll trip and fall flat on my face 🤦"*

vi. *You can skip transitions altogether if you do it right*

- *This is the reward -> If your RenderPass is smart — using* `VK_ATTACHMENT_LOAD_OP_DONT_CARE` *and reusing image layouts cleverly — you can avoid layout transitions entirely.*
  - *This is massive for tile-based GPUs (like on mobile phones):*
  - *No layout transition = no VRAM flush*
  - *Everything happens on-chip, like magic*

vii. *Analogy: Baking Cookies*

```
Let's say you're:
- Baking cookies (rendering)
- Then you plate them for display (presenting)
- Later you want to show them off or decorate them (sample in shaders)
```

- *Here's the deal:*

```
Vulkan Image Layout                          Cookie Stage
-----------------------      ---------------------------------
UNDEFINED                              Empty tray, nothing on it yet
COLOR_ATTACHMENT_OPTIMAL               You're baking the cookies
SHADER_READ_ONLY_OPTIMAL               You've finished baking and wanna decorate (like sampling
in a post-process shader)
PRESENT_SRC_KHR                           You're plating the cookies to serve  (sending to the
 screen)
```

- *But... here's the twist:*
  - *You can't decorate cookies while they're still baking in the oven.*
  - *And you definitely can't serve someone cookies that are still stuck in a 200°C tray.*
- *So Vulkan says:*
  - *"Please transition between layouts, so I know what stage your cookie is in — and I'll move it to the right place, with oven mitts, spatulas, etc."*

viii. *Why does this matter?*

- *If you don't do the transitions:*

```
You may try to grab a cookie off a 200°C tray and get burned ( invalid reads)
The cookies may not be fully baked ( undefined writes)
Or worse: you show your customer an empty plate because Vulkan never moved them to the
PRESENT_SRC_KHR plate
```

ix. *What makes Vulkan powerful?*

```
You get to say:
1. "Bake in tray A"
2. "Decorate using buffer B"
3. "Present from plate C"

But you must tell Vulkan when to move cookies from one surface to another.
Layouts = telling Vulkan exactly thaaat!
```

x. 🩹 *Subpass Optimization (Tile-Based GPUs)*

```
On tile-based GPUs (like PowerVR or Mali):
- Entire framebuffers live on-chip, in tiles
- You can run all subpasses without touching VRAM!

But it only works if Vulkan knows:
- The image will stay in the same layout
- No unnecessary STORE or layout transitions

By carefully using:
    layout = VK_IMAGE_LAYOUT_ATTACHMENT_OPTIMAL;
    loadOp = DONT_CARE;
    storeOp = DONT_CARE;

You unlock 🥷 zero-cost rendering.
```

- *That's why subpasses and layouts are so closely linked — **no layout change → no memory movement.***

3. 🏁 TL;DR: `Image Layout Transitions` Aren't Just Bureaucracy

```
🎯 They are literal instructions to the driver:
    - "Where this image lives"
    - "How it's structured"
    - "What GPU unit will touch it next"
    - "Whether you need to prepare, flush, decompress, or alias it"

🎯 And by explicitly telling the GPU, you:
    - Avoid expensive guesses
    - Skip hidden memory ops
    - Unlock mobile-level optimizations
    - Prevent subtle bugs and undefined behavior
```

4. 📜 `RenderPass Attachments Desc.`
   - `RenderPass Attachment` *is not an actual thing!*
   - `RenderPass Attachment Description/Descriptor` *is a thing!*
     - *However, the idea is.... We do "define" the* `Attachments` *right here, as we send the* `AttachmentDescriptions` -> `RenderPass`
   - `RenderPass Attachment Description/Descriptors` *are not actual images — they're a template for what the RenderPass expects!*
     - *& The FrameBuffers must delivery* `RenderPass` *exactly with that*
   - `RenderPass Attachment` *!=* `FrameBuffer Attachment`

RenderPass Attachments	*Framebuffers*
Define what is needed	*Provide which resources to use*
Abstract (format, usage, layout)	*Concrete (image views)*
Reusable across Framebuffers	*Swapchain-dependent (often 1:1)*

```
Think of it like a Socket & Plug
    - `RenderPass`  = The RenderPass defines the socket (shape, voltage).
    - `Framebuffer` = The Framebuffer provides the plug (actual wires) that fits the
socket.
```

5. 🎁 `FrameBuffer Attachment`
   - *Actual* `VkImageView`

```
Image Views (VkImageView):
    Handles to specific images (e.g., swapchain images, depth textures).
Compatibility:
    Must match the RenderPass's attachment definitions (format, sample count, size).
Swapchain Link:
    Typically, one Framebuffer per swapchain image.
```

6. 🛒 `FrameBuffers` [👐🍎🥚🍾]
   - *Binds concrete* `ImageViews` *(e.g.,* `SwapChain Images` *,* `Depth Textures` *) to the* `attachments` *defined in the* `RenderPass` *.*
   - *Must match the* `RenderPass's Attachment Descriptions` *(format, size, sample count).*
   - *Is* `SwapChain` *-dependent (e.g., each* `SwapChainImage` *typically has its own* `Framebuffer` *).*
   - *Analogy*

```
- `RenderPass`  = A recipe requiring "2 eggs and 1 cup of flour" (attachments).
- `Framebuffer` = The actual eggs and flour (image views) you use to bake a cake (render a
 frame).
```

7. 🎁 `Attachments`

```
RenderPass Attachments (Blueprint)   →       Framebuffer (Instance)
┌--------------------------------┐        ┌-----------------------------┐
| Attachment 0: Color (SRGB)     |        | Image View 0: Swapchain IMG |
| Attachment 1: Depth (D32_SF)   |        | Image View 1: Depth Texture |
└--------------------------------┘        └-----------------------------┘
```

- *Attachments are simply images (or buffers) where Vulkan stores or reads data during a RenderPass.*
- *Attachments are the actual framebuffer images (swapchain images, depth buffers, offscreen render targets, etc.)*

i. 🖌 *Color Attachments = where the pretty pixels (RGBA) are painted and stored. This is like your paint palette!* 🎨
ii. 🏔 *Depth Attachments = the landscapes that prevent objects from clipping or showing up out of order. Imagine topography maps for depth!* 🗺
iii. 🖍 *Stencil Attachments = the guides that show where we can paint, like drawing a "map" where only certain areas can be modified.* 🗺

- *What's inside?*
  - *A framebuffer that stores things like RGBA values (Red, Green, Blue, Alpha/Transparency).*
  - *For example,*
    - *Color Attachment 0 might hold the albedo or the final color of an object, while*
    - *Color Attachment 1 could store the lighting information or additional passes like ambient occlusion.*

```
Each attachment you declare includes:
- Format (VK_FORMAT_B8G8R8A8_SRGB, etc.)
- Sample count (for MSAA)
- Load/store ops
- Layouts (see above)
```

- *Then, each subpass tells Vulkan:*

> - *"From all the attachments I've declared, I'm gonna use these ones in this subpass."*

- *in Code:*

```
attachments[0] = colorAttachment;   // swapchain image
attachments[1] = depthAttachment;   // depth image

subpass.colorAttachment = &attachments[0];
subpass.depthAttachment = &attachments[1];
```

```
So even if your RenderPass only has one subpass, the Vulkan driver still wants to know:
- How many attachments
- What to do with them (clear/store?)
- What layouts they go into and come out as
```

8. `FrameBuffers` v/s `Attachments` :- The Last Fight, (If Above stuffs got you confused):-

i. Quick Comparison Table

Aspect	*Attachments (RenderPass)* ❌	*Framebuffers* 🖼️
Purpose	*Define what resources are needed (format, usage, layout transitions)* 📝	*Specify which actual images (image views) to use for those resources* 🖍️
Concrete/ Abstract	*Abstract (blueprint)* 📋	*Concrete (instance)* 🏗️
Lifetime	*Long-lived (reused across frames)* ♻️	*Short-lived (often recreated with swapchain)* ↻
Dependencies	*Independent of images/swapchain* ⛔🖼️	*Tied to swapchain images or specific textures* 🔗
Example	*"Need a color attachment (SRGB) and depth attachment (D32_SFLOAT)"* 🎨➕🎯	*"Use this swapchain image and that depth texture"* 🖼️🖼️

ii. Lifecycle Flowchart

```
RenderPass Creation 🏗️
│
├── Define Attachments 📄 (format, load/store ops, layouts)
│   └── "I need a color slot (B8G8R8A8_SRGB) and depth slot (D32_SFLOAT)" 🎨🎯
│
└── Define Subpasses 🔗 (how attachments are used in rendering steps)

↓

Framebuffer Creation 🛠️
│
├── Bind Image Views to Attachment Slots 🔌
│   └── "Slot 0 = Swapchain Image View 🖼️, Slot 1 = Depth Texture View 📦"
│
└── Validate Compatibility ☑️ (size, format, sample count)
```

iii. Use-Case Scenarios

Scenario	*Attachments (RenderPass)* ⚙️	*Framebuffers* 🖼️
Swapchain Rendering	*Define color/depth formats and layouts.* 🎨🔄🎯	*Bind swapchain images + depth texture.* 🖼️🔗📦
Deferred Rendering	*Define G-Buffer attachments (Albedo, Normal, Position).* 📦📦📦	*Bind actual G-Buffer image views.* 🖼️🖼️🖼️🖼️
Post-Processing	*Define input (e.g., HDR color) + output (e.g., SRGB).* ✨➡️🎨	*Bind input texture + swapchain image.* ⚙️🖼️

iv. *Key Interactions*

```
RenderPass Begin Command 🚦
│
├── Uses RenderPass Attachments 🧩 (format, load/store rules)
│
└── Uses Framebuffer 🖼️ (actual images to write to)

↓

GPU Renders 🎮
│
├── Reads/Writes to Framebuffer's Image Views 📊
│
└── Follows Attachment Rules (clearing, layout transitions) 🔄
```

v. *Emoji Analogy Time!* 🤯

- *Attachments = Recipe Ingredients List 📜 (e.g., "2 eggs 🥚🥚, 1 cup flour 🍚").*
- *Framebuffers = Actual Ingredients 🛒 (e.g., "This egg 🥚 from the fridge, that flour 🍚 from the pantry").*
- *Rendering = Baking the Cake 🎂 (combine them using the recipe steps!).*

9. Next Chapter will be on 🛒 `FrameBuffers` !!!! 🔷

Everything above is written with help from chatGPT

Everything below is not!

2. `vkCreateRenderPass()`

- https://vkdoc.net/man/vkCreateRenderPass
- **REY_DOCs**
  - *Copy Paste* `amVK_SwapChain.hh` *Current Implementation & Change it as needed*
    - *Trust me, this is the most fun way of doing this, xP*

3. `VkRenderPassCreateInfo()`

- https://vkdoc.net/man/VkRenderPassCreateInfo
  - `.flags` *-> Only Option:- used for Qualcom Extension*
  - `.pAttachments` *-> this->SubChapter4*
  - `.pSubpasses` *-> this->SubChapter5*
  - `.pDependencies` *-> this->SubChapter6*

4. `ImageViews`

1. `vkGetSwapchainImagesKHR()`
   - *https://vkdoc.net/man/vkGetSwapchainImagesKHR*
   - *Implement Exactly like Chapter2.5* 😉
     - `vkGetPhysicalDeviceQueueFamilyProperties()`
   - **REY_DOCs**

```
class amVK_SwapChain  {
    ...
  public:
    amVK_Device *D = nullptr;
    VkSwapchainKHR SC = nullptr;
    REY_Array<VkImage>    amVK_1D_SC_IMGs;
    REY_Array<amVK_Image> amVK_1D_SC_IMGs_amVK_WRAP;
    bool called_GetSwapchainImagesKHR = false;

  public:
    ...
```

2. `vkCreateImageView()`
   - *https://vkdoc.net/man/vkCreateImageView*
   - **REY_DOCs**

```
void CreateSwapChainImageViews(void) {
    REY_Array_LOOP(amVK_1D_SC_IMGs_amVK_WRAP, i) {
        amVK_1D_SC_IMGs_amVK_WRAP[i].createImageView();
    }
}
```

     - `amVK_Image.hh` *:- 4.guide.chapter5.3.2.Image.hh*

3. `VkImageViewCreateInfo`
   - *https://vkdoc.net/man/VkImageViewCreateInfo*
   - **REY_DOCs**

```
void amVK_SwapChain::CreateSwapChainImageViews(void) {
    REY_Array_LOOP(amVK_1D_SC_IMGs_amVK_WRAP, i) {
            // ViewCI.image
            // ViewCI.format
                // should be set inside amVK_SwapChain::GetSwapchainImagesKHR()
        amVK_1D_SC_IMGs_amVK_WRAP[i].ViewCI.viewType = VK_IMAGE_VIEW_TYPE_2D;

        amVK_1D_SC_IMGs_amVK_WRAP[i].ViewCI.components = { // Equivalent to:
            VK_COMPONENT_SWIZZLE_R,         // VK_COMPONENT_SWIZZLE_IDENTITY
            VK_COMPONENT_SWIZZLE_G,         // VK_COMPONENT_SWIZZLE_IDENTITY
            VK_COMPONENT_SWIZZLE_B,         // VK_COMPONENT_SWIZZLE_IDENTITY
            VK_COMPONENT_SWIZZLE_A          // VK_COMPONENT_SWIZZLE_IDENTITY
        };

        amVK_1D_SC_IMGs_amVK_WRAP[i].ViewCI.subresourceRange.aspectMask =
VK_IMAGE_ASPECT_COLOR_BIT;
        amVK_1D_SC_IMGs_amVK_WRAP[i].ViewCI.subresourceRange.baseMipLevel = 0;
        amVK_1D_SC_IMGs_amVK_WRAP[i].ViewCI.subresourceRange.levelCount = 1;
        amVK_1D_SC_IMGs_amVK_WRAP[i].ViewCI.subresourceRange.baseArrayLayer = 0;
        amVK_1D_SC_IMGs_amVK_WRAP[i].ViewCI.subresourceRange.layerCount = 1;

        amVK_1D_SC_IMGs_amVK_WRAP[i].createImageView();
    }
}
```

5. `VkAttachmentDescription`

- https://vkdoc.net/man/VkAttachmentDescription

6. `VkSubpassDescription`

- https://vkdoc.net/man/VkSubpassDescription

7. `VkSubpassDependency`

- https://vkdoc.net/man/VkSubpassDependency

8. All the last 3 together ---> Code

```
class amVK_RenderPass  {
  public:
    REY_ArrayDYN<VkAttachmentDescription> attachments;
    REY_ArrayDYN<VkSubpassDescription> subpasses;
    REY_ArrayDYN<VkSubpassDependency> dependencies;

    void set_attachments_subpasses_dependencies(void);
}
```

- `amVK_RenderPass.hh` [Full Implementation]:- 4.guide.chapter5.8.RenderPass.hh

```
amVK_RenderPass *RP = new amVK_RenderPass(D);
    RP->attachments.push_back({
        .format = SC->CI.imageFormat,                       // Use the color format selected by the
swapchain
        .samples = VK_SAMPLE_COUNT_1_BIT,                   // We don't use multi sampling in this
example
        .loadOp = VK_ATTACHMENT_LOAD_OP_CLEAR,              // Clear this attachment at the start of
the render pass
        .storeOp = VK_ATTACHMENT_STORE_OP_STORE,
            // Keep its contents after the render pass is finished (for displaying it)
        .stencilLoadOp = VK_ATTACHMENT_LOAD_OP_DONT_CARE,
            // Similar to loadOp, but for stenciling (we don't use stencil here)
        .stencilStoreOp = VK_ATTACHMENT_STORE_OP_DONT_CARE,
            // Similar to storeOp, but for stenciling (we don't use stencil here)
        .initialLayout = VK_IMAGE_LAYOUT_UNDEFINED,
            // Layout at render pass start. Initial doesn't matter, so we use undefined
        .finalLayout = VK_IMAGE_LAYOUT_PRESENT_SRC_KHR,
            // Layout to which the attachment is transitioned when the render pass is finished
            // As we want to present the color attachment, we transition to PRESENT_KHR
    });

    VkAttachmentReference colorReference = {
        .attachment = 0,
        .layout = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL
    };
```

```
    RP->subpasses.push_back({
        .pipelineBindPoint = VK_PIPELINE_BIND_POINT_GRAPHICS,

        .inputAttachmentCount = 0,
            // Input attachments can be used to sample from contents of a previous subpass
        .pInputAttachments = nullptr,          // (Input attachments not used by this example)
        .colorAttachmentCount = 1,             // Subpass uses one color attachment
        .pColorAttachments = &colorReference,  // Reference to the color attachment in slot 0

        .pResolveAttachments = nullptr,
            // Resolve attachments are resolved at the end of a sub pass and can be used for e.g.
multi sampling
        .pDepthStencilAttachment = nullptr,    // (Depth attachments not used by this sample)
        .preserveAttachmentCount = 0,
            // Preserved attachments can be used to loop (and preserve) attachments through subpasses
        .pPreserveAttachments = nullptr        // (Preserve attachments not used by this example)
    });

    RP->dependencies.push_back({
        // Setup dependency and add implicit layout transition from final to initial layout for the
color attachment.
        // (The actual usage layout is preserved through the layout specified in the attachment
reference).
        .srcSubpass = VK_SUBPASS_EXTERNAL,
        .dstSubpass = 0,
        .srcStageMask = VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT,
        .dstStageMask = VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT,
        .srcAccessMask = VK_ACCESS_NONE,
        .dstAccessMask = VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT | VK_ACCESS_COLOR_ATTACHMENT_READ_BIT,
    });

    RP->set_attachments_subpasses_dependencies();
    RP->createRenderPass();

    ------------------------------------ Made with help from P.A.Minerva Vulkan Guide
-------------------------------------
    https://paminerva.github.io/docs/LearnVulkan/01.A-Hello-Window#416---creating-a-render-pass
```

- `main.cpp` [Full Implementation]:- 4.guide.chapter5.8.main.cpp

9. By This time, VkSurfaceKHR deserves it's very own class `amVK_Surface`

- `amVK_Surface.hh` [Full Implementation]:- 4.guide.chapter5.9.Surface.hh