THE MOST COMMON VULKAN MISTAKES

DOMINIK WITCZAK, AMD

TECH REVIEW: DANIEL RAKOS, AMD
DERRICK OWENS, AMD
WHO?

- Dominik Witczak
- MTS Software Development Engineer at AMD

- Regular contributor to the following standards:
  - OpenGL (4.x)
  - OpenGL ES (3.0 and beyond)
  - Vulkan

- After-hours demoscene activist:
  - Event organizer
  - Programmer

- Trivia:
  - Graduated from WMiI department back in 2010
WHAT?

Agenda for today:

• What is **Vulkan**?
• What is it and is it **not** about?
• Who is it for?

• **Problematic areas:**
  • Command queues
  • Descriptor sets
  • Images
  • Memory barriers
  • Memory management
  • Renderpasses
  • Synchronization
A simplified view of a typical OpenGL or <DX 10 app rendering pipeline:

- **CPU thread:**
  - (generate data)
  - (upload data to GPU)
  - (change state)
  - Dispatch/draw!
  - Swap buffers

- **GPU usage:**

Why?
A simplified view of a typical OpenGL or <DX 10 app rendering pipeline:

- **CPU thread:**
  - (generate data) → (upload data to GPU) → (change state) → Dispatch/draw! → Swap buffers

- **GPU usage:**

**Why?**

- **CPU->GPU** command submission is **time-consuming:**
  - Only submit and start executing GPU-side if:
    - All commands for a frame have been submitted.
    - Command buffer fills up
    - Upon app's explicit request.

- App can submit **different** commands **every frame:**
  - Cannot bake command buffers in advance!
VULKAN
THE BEGINNINGS

Typical **OpenGL** or **<DX 10** app rendering pipeline (more advanced apps):

- **CPU thread 1:** Generate data
- **CPU thread 2:** Generate data
- **CPU thread (N-1):** Generate data
- **CPU thread N:** Generate data
- **Rendering thread:** Wait → (upload data to GPU) → (change state) → Dispatch/draw! → Swap buffers

**GPU usage:**

Can we do any better?
Typical **OpenGL** or **<DX 10 app** rendering pipeline (more advanced apps):

- **CPU thread 1:** Generate data
- **CPU thread 2:** Generate data
- **CPU thread (N-1):** Generate data
- **CPU thread N:** Generate data
- **Rendering thread:** Wait → (upload data to GPU) → (change state) → Dispatch/draw! → Swap buffers

- **GPU usage:**

  - **Generate as much data on GPU as possible**
  - **Use multiple rendering threads to streamline data uploads**
  - **Batch draw calls by state configuration**
  - **Use compute shaders to calculate data in parallel on some HW**

**Render as many frames in advance as possible to reduce GPU bubbles**
These workarounds do not solve the biggest problem:

- GPUs are highly asynchronous constructs:
- Designed to perform many kinds of tasks in parallel:
  - Computations
  - DMA transfers
  - Rasterization
  - Other (eg. accelerated image data conversion)

But from API standpoint:
- GPU can only be requested to execute work chunks from one rendering thread!
- Apps cannot be trusted – CPU time spent on API call validation..
Do we really care?

- More and more **CPU-bound apps** are showing up on the market.
  - Driver thread(s) consuming CPU time
  - Increasing app complexity

- No easy way to address these in a cross-platform way.

- **Tilers** cannot leverage their full power on **OpenGL ES**.
- Only **vendor-specific** solutions exist (eg. Pixel Local Storage)

Not to mention use cases like:

- Multiple **GPU** support
- **VR**

This is where **Vulkan** comes into play...
Vulkan addresses all of the discussed issues:

- Exposes **GPU** as a set of **command queue families**.
- **Command buffers** can be **submitted to queues** from **multiple threads**.
- **Application** is responsible for:
  - submitting **work chunks** to the right **command queues**.
  - **synchronization** of **GPU jobs’ execution**.
- Exposes available **GPU memory** as a set of **memory heaps**.
- **Application** is responsible for **flushes / invalidation / management**.

- **Applications** are **required** to adapt to the running **GPU’s capabilities**.
- **Misbehave** and **hang the GPU**.
VULKAN

DO I NEED IT?

- **Who NEEDS Vulkan?**
  - **CPU-bound applications:**
    - Vast majority of information required to compute / render – prebaked at loading time.
    - A frame can be rendered with just two commands!
    - **No driver-side validation** = more CPU time for stuff that really matters.
  - **GPU-bound applications:**
    - Improve **GPU utilization** by:
      - Submitting compute / graphics jobs to relevant queue families.
      - Performing VRAM -> VRAM & RAM <-> VRAM copy ops with transfer queues.
    - **No sudden performance drops** or spikes:
      - All GPU-side caches are flushed, according to app-specified information, at predictable times.
      - Driver no longer needs to do any guess-work.
**VULKAN**

**DO I NEED IT?**

- **Who ** **MAY** need **Vulkan?**
  - Existing **GL 4.x / <= DX 11** applications:
    - Moving to **Vulkan** may or may not bring performance benefits.
    - Likely to spend less CPU power.

- **Who does NOT need ** **Vulkan?**
  - Prototype applications requiring rapid development time:
    - **Validation layers** do not cover whole specification yet.
    - Many incorrect use cases are still not detected.
    - **Steep** learning curve.

  - **Simple applications** which are not **CPU- or GPU-bound**:
    - Unless for **learning purposes**, these are unlikely to benefit from **Vulkan**.
Our driver has been out for a few months now.

- **Top-level observations:**
  - Vulkan is **demanding** to use, both app-side and time-wise.
  - If an app works with GPU A, it **doesn’t have** to hold for GPU B.

- **Common pit-falls:**
  - Barriers
  - Correct data uploads
  - Image transitions
  - Renderpasses

- ISVs: generally **reluctant** to use validation layers.
  - **Please do.** This saves both you and us a lot of time
**VULKAN**

PROBLEMATIC AREAS: COMMAND QUEUES

- **CPU-side:**
  - No *rendering threads* in Vulkan
  - *Work chunks* submittable from multiple threads to GPU-side *command queue*.

- **GPU-side:**
  - *Command queues* are grouped by type(s) of *commands* they can execute.

- **Problem:**
  - Number of *command queues* – *hardware-dependent!*
  - Number of *queue families* – *hardware-dependent!*
VULKAN
PROBLEMATIC AREAS: COMMAND QUEUES

Why is this a problem?
- Efficient **GPU task distribution** is now **Vulkan** app’s responsibility.
- The solution must be able to **up-** and **down-scale**, depending on device caps.
- No **open-source solutions** available yet
- Only a single **compute+gfx queue family** guaranteed in **Vulkan 1.0**.
- Simple apps will likely rely **solely** on the presence of the **universal queue**.
- ..but wasn’t **Vulkan** written with **performance** in the 1st place?!

Solution:
- Test your rendering engine on various **Vulkan** implementations.
In **Vulkan**, **command buffers**:  
- ..hold **commands** to be executed **GPU-side**  
- ..are **reusable**, unless explicitly stated **otherwise** by the app.

**Problem:**  
- Apps often **re-record command buffers** every frame.

**Why is this a problem?**  
- Wastes a lot of **CPU time**.  
- **Not required** in many cases.
### Problem:
- Apps re-record **command buffers** every frame.

### Solution:
- Move all **parameters** that affect the **rendering logic** to **images / SBs / UBs**.
- Pre-bake all **command buffers** once per each **swapchain image**, if necessary.
- Use **indirect dispatch/draw commands** if they improve command buffer reusability.
Memory management is also Vulkan app’s responsibility:

- **Physical device** reports >= 1 memory heaps
- Each **memory heap**:
  - has **platform-specific size**.
  - **may**, but needs **not be device-local**.

- **Memory heaps** – **not directly** accessible to apps.
- Instead, the driver exposes an **array** of HW-specific „**memory types**“:

```c
typedef struct VkMemoryType {
    VkMemoryPropertyFlags propertyFlags;
    uint32_t heapIndex;
} VkMemoryType;
```

- When alloc’ing **GPU memory**, Vulkan app specifies **memory type index**.
VULKAN
PROBLEMATIC AREAS: MEMORY MANAGEMENT

- **What’s the hard part?**
  - **Vulkan**<->**app contract** is very thin.
  - The following is **guaranteed**:
    - At least one **memory type** is host-visible & host-coherent.
    - At least one **memory type** is device-local.
  - **Buffer** & **image memory** alloc’s must come from driver-specific **memory types**
  - The types **MAY** vary, depending on:
    - Object properties
    - Object type
  - But the best is yet to come..
VULKAN
PROBLEMATIC AREAS: MEMORY MANAGEMENT

- **What’s the hardest part?**
  - ISVs tend to **ignore** the `maxMemoryAllocationCount` limit:
    - The **min max** for the **simultaneous live allocations limit** is **4096**.
      - Very **easy** to reach in **complex applications**.
      - The **usual** value reported by **desktop Vulkan implementations**.

- **Solution:**
  - **Pre-allocate & manage** available **GPU memory app-side**.
  - **Avoid** small memory allocations, **sub-allocate them from larger ones**.
Majority of shaders access external data.

In Vulkan:
- These are exposed via descriptors.
- Descriptors cannot be created directly.
- Instead, they are retrieved from a descriptor pool instantiated by the app:

```c
typedef struct VkDescriptorPoolCreateInfo {
    VkStructureType       sType;
    const void*            pNext;
    VkDescriptorPoolCreateFlags  flags;
    uint32_t               maxSets;
    uint32_t               poolSizeCount;
    const VkDescriptorPoolSize* pPoolSizes;
} VkDescriptorPoolCreateInfo;

typedef struct VkDescriptorPoolSize {
    VkDescriptorType  type;
    uint32_t           descriptorCount;
} VkDescriptorPoolSize;
```
Problem:
- `<maxSets>` does not work as ISVs seem to expect.

Frequently seen misunderstanding:
- „I can allocate `<maxSets> * {poolSizeCount * pPoolSizes} descriptors”
- „No? Your driver sucks, that’s what I can do with vendor X’s driver!”

Correct understanding:
- Up to `N` of preallocated descriptors can be distributed to up to `<maxSets> DSES`.
Descriptors are then grouped into Descriptor Sets for later usage.

- Descriptor type <-> binding relations is defined by a DS layout.
- Actual buffers / images for GPU consumption are bound in command buffers.

A DS layout is created with:

```c
typedef struct VkDescriptorSetLayoutCreateInfo {
    VkStructureType sType;
    const void* pNext;
    VkDescriptorSetLayoutCreateFlags flags;
    uint32_t bindingCount;
    const VkDescriptorSetLayoutBinding* pBindings;
} VkDescriptorSetLayoutCreateInfo;
```

```c
typedef struct VkDescriptorSetLayoutBinding {
    uint32_t binding;
    VkDescriptorType descriptorType;
    uint32_t descriptorCount;
    VkShaderStageFlags stageFlags;
    const VkSampler* pImmutableSamplers;
} VkDescriptorSetLayoutBinding;
```
VULKAN
PROBLEMATIC AREAS: SPARSE DESCRIPTOR BINDINGS

**Problem:**

- How should a DS layout look for the following descriptor set:
  - Binding 0: Storage buffer
  - Binding 2: Storage image

- Do I need to include a VkDescriptorSetLayoutBinding item for binding 1 or not?

```c
typedef struct VkDescriptorSetLayoutCreateInfo {
    VkStructureType sType;
    const void* pNext;
    VkDescriptorSetLayoutCreateFlags flags;
    uint32_t bindingCount;
    const VkDescriptorSetLayoutBinding* pBindings;
} VkDescriptorSetLayoutCreateInfo;
```

```c
typedef struct VkDescriptorSetLayoutBinding {
    uint32_t binding;
    VkDescriptorType descriptorType;
    uint32_t descriptorCount;
    VkShaderStageFlags stageFlags;
    const VkSampler* pImmutableSamplers;
} VkDescriptorSetLayoutBinding;
```
VULKAN
PROBLEMATIC AREAS: SPARSE DESCRIPTOR BINDINGS

- **Problem:**
  - How should a *DS layout* look for the following *descriptor set*:
    - Binding 0: Storage buffer
    - Binding 2: Storage image
  - Do I need to include a *VkDescriptorSetLayoutBinding* item for *binding 1* or not?

- **Solution:**
  - The app is *inefficient*, dummy bindings *negatively* affect *performance*.
  - But if you really need them: *yes*, the *binding* is *needed*.
  - Make sure to set ::*descriptorCount* to 0 for each *unused binding*. 
VULKAN
PROBLEMATIC AREAS: IMAGES

- In Vulkan, **texture**:
  - **state** is stored in **Image Objects**
  - **data** is stored in **Memory Objects**, bound to an **Image Object**

- **Image Objects** are created by specifying **properties** of the **image** data:
  - The usual bits and bobs such as:
    - **Type** (1D, 2D or 3D)
    - **Base mipmap size**
    - **Number of mipmaps**
  - **Tiling type**
  - **Usage flags**
  - Other miscellanea..
VULKAN

PROBLEMATIC AREAS: IMAGES

- In Vulkan, texture:
  - state is stored in Image Objects
  - data is stored in Memory Objects, bound to an Image Object

- Image Objects are created by specifying properties of the image data:
  - The usual bits and bobs such as:
    - Type (1D, 2D or 3D)
    - Base mipmap size
    - Number of mipmaps
    - Tiling type
    - Usage flags
  - Other miscellanea..
Vulkan requires **up-front image usage** declaration at **creation time.**

- Usage is a **bit combination** of one or more flags below:

```c
typedef enum VkImageUsageFlagBits {
    VK_IMAGE_USAGE_TRANSFER_SRC_BIT = 0x00000001,
    VK_IMAGE_USAGE_TRANSFER_DST_BIT = 0x00000002,
    VK_IMAGE_USAGE_SAMPLED_BIT = 0x00000004,
    VK_IMAGE_USAGE_STORAGE_BIT = 0x00000008,
    VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT = 0x00000010,
    VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT = 0x00000020,
    VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT = 0x00000040,
    VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT = 0x00000080,
} VkImageUsageFlagBits;
```

- A driver **may not** provide **format support** for certain **image usages**

- When it does, **usage setting restricts**:
  - supported **memory types**
  - maximum **image resolution**, **number of samples**, etc.
VULKAN

PROBLEMATIC AREAS: IMAGE USAGE FLAGS

- **Common problem**: App specifies incorrect *image usage*.

- **Example**:
  - Consider an *image* created with `VK_IMAGE_USAGE_TRANSFER_DST_BIT` usage.
  - The *image* must not be used as a *color attachment*.
  - App does not care.

- **Outcome**:
  - Undefined behavior

- **Solution**:
  - This class of problems can be easily detected when *validation* is enabled.
VULKAN
PROBLEMATIC AREAS: IMAGE TILING

- **Tiling** setting determines *image data layout* used by the GPU:
  - **Linear**: row-major image row arrangement, each row *potentially padded*
  - **Optimal**: platform-specific data arrangement, optimized for *speed*.

- **Properties of linearly-tiled images**:
  - Support a *subset of functionality* provided for *optimally-tiled images*
  - Less performant

- Why bother with *linear images* then?
  - Crucial if you need to *read back* image data rendered by *GPU*. 
**VULKAN**

**PROBLEMATIC AREAS: IMAGE TILING**

- **Common problem:** ISVs copy data **directly** to **optimally-tiled images.**
- **Typical scenario:**
  - **Image A** is created with `VK_IMAGE_TILING_OPTIMAL` tiling setting.
  - Application calls `vkGetImageSubresourceLayout()` for **image A**:

```
void vkGetImageSubresourceLayout(
    VkDevice device,  
    VkImage image, 
    const VkImageSubresource* pSubresource, 
    VkSubresourceLayout* pLayout);
```

- Application tries to upload data using the „reported” characteristics.

```
typedef struct VkImageSubresource {
    VkImageAspectFlags aspectMask;
    uint32_t mipLevel;
    uint32_t arrayLayer;
} VkImageSubresource;
```

```
typedef struct VkSubresourceLayout { 
    VkDeviceSize offset;
    VkDeviceSize size;
    VkDeviceSize rowPitch;
    VkDeviceSize arrayPitch;
    VkDeviceSize depthPitch;
} VkSubresourceLayout;
```
**Solution:**

- Use a **staging buffer** to copy data to **optimally-tiled images**:
  1. Create a **buffer object** and bind a **memory region** to it.
  2. Fill it with data.
  3. Transition the image to **GENERAL** or **TRANSFER_DST_OPTIMAL** layout.
  4. Schedule a **copy op** by calling `vkCmdCopyBufferToImage()`.
  5. Submit the **command buffer**, wait till it finishes executing.
  6. Release the **temporary buffer object**.

- Remember: **buffer -> image copy ops** will **not** work for **MS images**.
- To upload data there, you’ll need to use an actual **dispatch/draw** call.
GPUs may (de-)compress or rearrange data on-the-fly
- Less bandwidth pressure => better performance
- DX <=11 and OpenGL: transparent, heuristics-driven process.
- Vulkan: happens at image layout transition time.
- Example: DCC (see http://gpuopen.com/dcc-overview/)

Hardware-level optimizations:
- Differ between HW architectures & HW generations.
- Generally vendor-specific
**VULKAN**

**PROBLEMATIC AREAS: IMAGE LAYOUT TRANSITIONS**

- **In Vulkan:**
  - **Images must be moved** to the **right layout** before **usage**.
  - This can be requested by:
    - injecting **image barriers** into **command buffers**
    - correct **renderpass** & **subpass** configuration
  - Get it wrong and **visual corruption may occur**:

```c
typedef enum VkImageLayout {
    VK_IMAGE_LAYOUT_UNDEFINED = 0,
    VK_IMAGE_LAYOUT_GENERAL = 1,
    VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL = 2,
    VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL = 3,
    VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL = 4,
    VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL = 5,
    VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL = 6,
    VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL = 7,
    VK_IMAGE_LAYOUT_PREINITIALIZED = 8,
} VkImageLayout;
```
VULKAN

PROBLEMATIC AREAS: IMAGE LAYOUT TRANSITIONS

Image A is created (UNDEFINED layout)

Command buffer 1

Image A (UNDEFINED -> COLOR_ATTACHMENT_OPTIMAL layout)

Command buffer 2

Render to Image A

Image A (COLOR_ATTACHMENT_OPTIMAL -> SHADER_READ_ONLY_OPTIMAL layout)

Dispatch call (fetches texels from Image A)

Image A (SHADER_READ_ONLY_OPTIMAL -> COLOR_ATTACHMENT_OPTIMAL layout)
VULKAN
PROBLEMATIC AREAS: IMAGE LAYOUT TRANSITIONS

- **Common problems:**
  1. Image is **transitioned** into an **invalid layout**.

![Diagram showing image layout transitions](image)
Common problems:

2. **Old layout** defined in an **image barrier** is **incorrect**.

VULKAN
PROBLEMATIC AREAS: IMAGE LAYOUT TRANSITIONS

- **Image A**
  - \( \text{COLOR\ ATTACHMENT\ OPTIMAL} \rightarrow \text{SHADER\ READ\ ONLY\ OPTIMAL} \)
  - Dispatch call (fetches texels from Image A)
  - \( \text{TRANSFER\ DST\ OPTIMAL} \rightarrow \text{COLOR\ ATTACHMENT\ OPTIMAL} \)
**VULKAN**

PROBLEMATIC AREAS: IMAGE LAYOUT TRANSITIONS

- **Common problems:**
  3. „Hey AMD, my app works on vendor Y’s driver, your driver sucks!”

  – Some vendors ignore **image barriers**. We **do not**.
  – Whose driver is **wrong** then? 😊

- **Solution:**
  – **Validation layers** are constantly improving – use them!
  – Test your software on various **Vulkan implementations**.
Common problems:

4. ISVs misunderstand how renderpasses transition image subresources.

- Renderpasses are a novel, complex concept in Vulkan.
- Introduced to let the driver „travel in time” and know in advance:
  - what color/DS attachments will be rasterized to or accessed (When? How?)
  - which image subresource ranges need to be synchronized (When? How?)
  - what layouts image subresources should be transitioned to, and when.

- That’s a lot of info to get wrong, especially when described manually 🙃
Common problems:

4. ISVs misunderstand how **renderpasses** transition **image subresources**.

- **Renderpass**: each "rendering pass" is described by **user-specified subpass**
**Common problems:**

4. ISVs misunderstand how **renderpasses** transition **image subresources**.

Renderpass: **execution order** is **deduced** from user-specified **dependencies**.
Common problems:

4. ISVs misunderstand how renderpasses transition image subresources.

VULKAN
PROBLEMATIC AREAS: IMAGE LAYOUT TRANSITIONS & RENDERPASSES

Render subpass → Image
Blurred subpass
Sum subpass

Specified at renderpass create time: (VkAttachmentDescription)
- ::initialLayout (layout of the image when renderpass begins)
- ::finalLayout (layout to transition to when renderpass ends)

Renderpass: image transitions are deduced from subpass AND renderpass configuration
### Common problems:

4. ISVs misunderstand how **renderpasses** transition **image subresources**.

**VULKAN PROBLEMATIC AREAS: IMAGE LAYOUT TRANSITIONS & RENDERPASSES**

```c
typedef struct VkAttachmentDescription {
    VkAttachmentDescriptionFlags flags;
    VkFormat format;
    VkSampleCountFlagBits samples;
    VkAttachmentLoadOp loadOp;
    VkAttachmentStoreOp storeOp;
    VkAttachmentLoadOp stencilLoadOp;
    VkAttachmentStoreOp stencilStoreOp;
    VkImageLayout initialLayout;
    VkImageLayout finalLayout;
} VkAttachmentDescription;
```

**Render subpass** → **Image**

- Specified at **renderpass** create time: `(VkAttachmentDescription)`
  - `::initialLayout` (layout of the image when **renderpass** begins)
  - `::finalLayout` (layout to **transition to** when **renderpass** ends)

**Renderpass: image transitions** are deduced from **subpass** AND **renderpass** configuration
**VULKAN**

**PROBLEMATIC AREAS: IMAGE LAYOUT TRANSITIONS & RENDERPASSES**

- **Common problems:**
  4. ISVs misunderstand how **renderpasses** transition **image subresources**.

```c
typedef struct VkSubpassDescription {
    VkSubpassDescriptionFlags flags;
    VkPipelineBindPoint pipelineBindPoint;
    uint32_t inputAttachmentCount;
    const VkAttachmentReference* pInputAttachments;
    const VkAttachmentReference* colorAttachmentCount;
    const VkAttachmentReference* pColorAttachments;
    const VkAttachmentReference* resolveAttachmentCount;
    const VkAttachmentReference* pResolveAttachments;
    const VkAttachmentReference* preserveAttachmentCount;
    const VkAttachmentReference* pPreserveAttachments;
} VkSubpassDescription;
```

- **Blur subpass**
  - Specified for each **subpass**: (VkAttachmentReference)
  - `::layout (layout to transition to when subpass starts)`

```c
typedef struct VkAttachmentReference {
    uint32_t attachment;
    VkImageLayout layout;
} VkAttachmentReference;
```

**Renderpass: image transitions** are deduced from **subpass AND renderpass configuration**
VULKAN

PROBLEMATIC AREAS: GPU-SIDE SYNCHRONIZATION

- Uber-general Vulkan’s GPU-side command execution rules:

  1. **Command queues** run independently of each other.
  2. When submitted to *queue A*, command buffers execute in the specified order.
  3. Unless order is enforced by **barriers / renderpass / sync primitives**:
      1. Submitted commands may be executed in parallel.
      2. Submitted commands may be executed out-of-order.

- The following **sync objects** are available:
  - **Events** (intra-queue synchronization)
  - **Semaphores** (inter-queue synchronization)
  - **Fence** (blocks CPU thread until submitted job chunk<es> finish<es> running)
VULKAN
PROBLEMATIC AREAS: GPU-SIDE SYNCHRONIZATION

- **Problem:**
  - ISVs sometimes create *sync objects* every frame.

- **Solution:**
  - Avoid at all cost!
  - Remember that:
    1. *Events* can be reset CPU- and GPU-side
    2. *Fences* can be reset CPU-side
    3. *Semaphores* automatically reset after being successfully waited upon.

  - If more feasible, bake per-*swapchain* image set of *sync objects* in advance.
ANY QUESTIONS?
THANK YOU