Vulkan

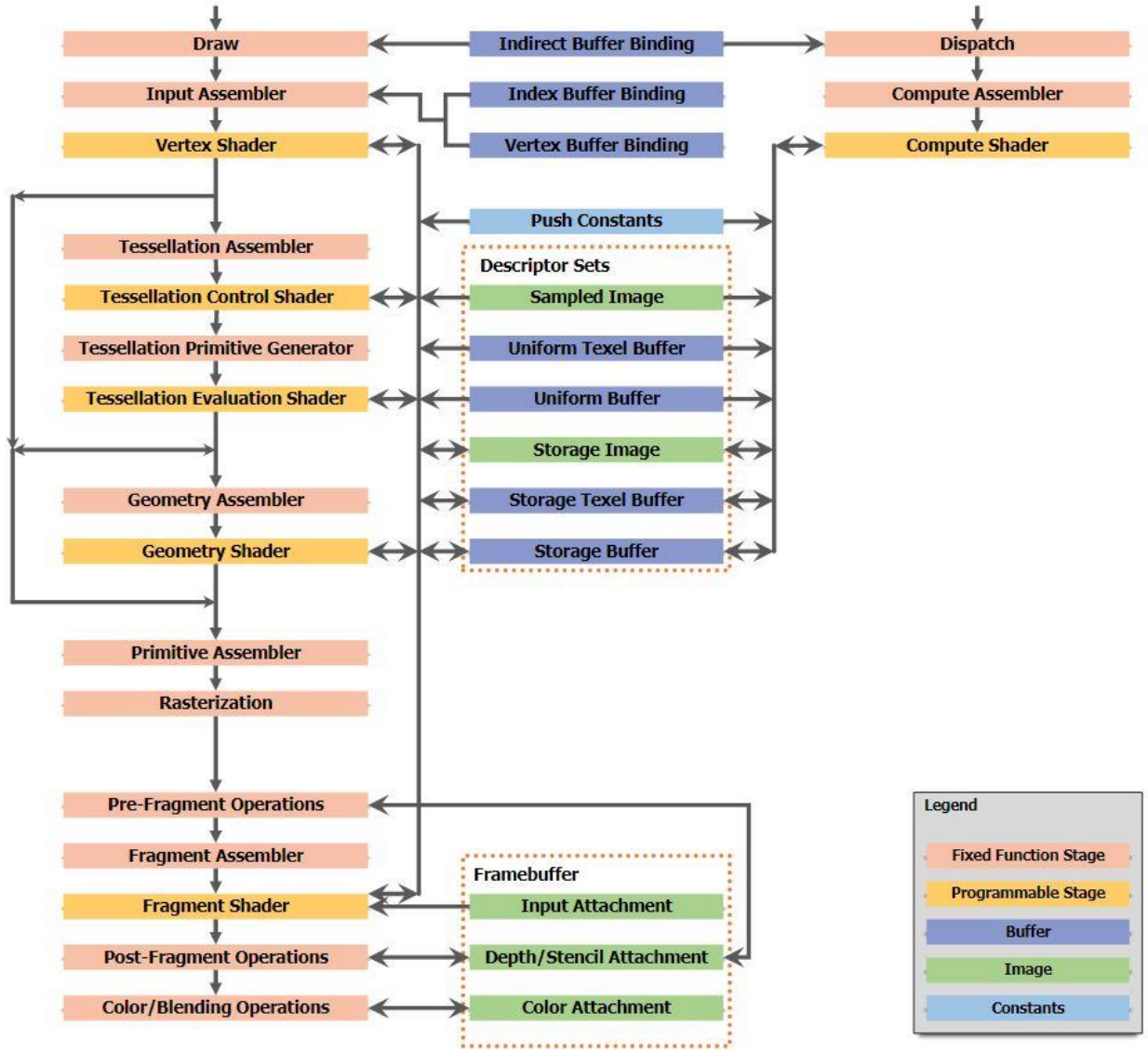
Malek.Isart@gmail.com

Objectifs pour ces séances

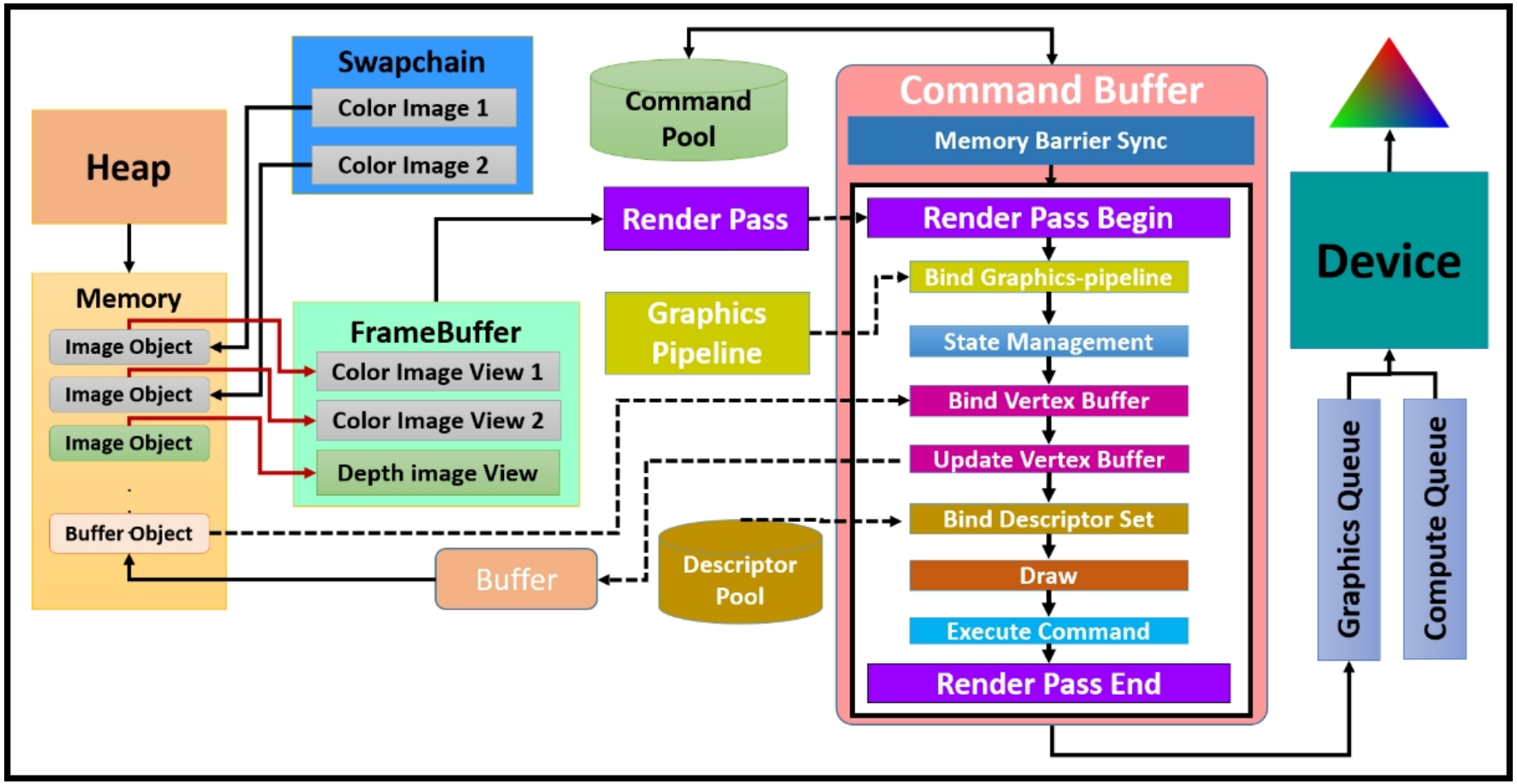
- Un modèle 3D texturé et illuminé
  - FBX ou autre, éventuellement normal mappé
- Au moins deux UBO (minimum matrices proj+view et world+normal)
  - Push Constants (équivalent de glUniformxx()) éventuellement
- Rendu en MSAA 4x
  - Dans une Render Target
- Une passe de Post Processing
  - Histoire de comprendre comment fonctionnent les "input attachment" de vulkan pour récupérer le contenu d'un rendu d'une passe précédente sous forme de texture (pour simplifier).

Procédez par étapes

- 1. Clear backbuffer
  - Sans puis avec une RenderPass
- 2. Premier triangle coloré en NDC
  - D'abord avec un simple shader (attribute less)
  - (vertex et index) buffers, 1 shader, 1 vertex input attribute et binding (input layout / VAO)
- 3. Rendu hors écran
  - puis en MSAA (sampleCount = 4 par ex)
- 4. Rendu 3D
  - Descriptor Sets pour les uniform buffers
- 5. Textures
  - Combined ou séparés (textures + 1 ou plusieurs samplers)


Draw
Indirect Buffer Binding
Dispatch
Input Assembler
Index Buffer Binding
Compute Assembler
Vertex Shader
Vertex Buffer Binding
Compute Shader
Push Constants
Tessellation Assembler
Descriptor Sets
Tessellation Control Shader
Sampled Image
Tessellation Primitive Generator
Uniform Texel Buffer
Tessellation Evaluation Shader
Uniform Buffer
Storage Image
Geometry Assembler
Storage Texel Buffer
Geometry Shader
Storage Buffer
Primitive Assembler
Rasterization
Pre-Fragment Operations
Fragment Assembler
Framebuffer
Fragment Shader
Input Attachment
Post-Fragment Operations
Depth/Stencil Attachment
Color/Blending Operations
Color Attachment
Legend
Fixed Function Stage
Programmable Stage
Buffer
Image
Constants

Les élements nécessaires pour le rendu



Terminologie

- **Instance** : une instanciation d'un processus Vulkan
  - On peut donc avoir plusieurs instances tournant en même temps
- **Physical Device** : accès à un GPU de la machine
  - On a souvent un GPU integré (IGP) et un ou plusieurs GPU internes ou externes
- **Device** : abstraction du GPU d'un point de vue logique (fonctionnel)
- **Surface** : abstraction de la surface de rendu native de l'OS
- **Presentation** : action consistant à envoyer une image à l'affichage
  - Vérifiez la capacité d'une Queue à effectuer la présentation !

- **Heap** : la mémoire du GPU est subdivisée en une ou plusieurs parties selon l'architecture et les capacités du système.

- **Memory Type** : chaque 'heap' peut être essentiellement de deux types, **'host'** (RAM) ou **'device'** (VRAM). Ils peuvent avoir une ou plusieurs des propriétés suivantes:
  - "**Device local**" signifie qu'il s'agit de la mémoire physique du GPU par défaut seulement accessible par le GPU
  - "**host visible**" indique que la mémoire est accessible par le CPU
  - "**Host coherent**" spécifie que les changements en mémoire par le CPU seront automatiquement visibles par le GPU et vice-versa (pas besoin de flush/invalider les caches).
  - "**Host cached**"

Usage de la mémoire

- Principalement allouer de la mémoire pour less buffers et images, mémoire qu'on manipule à l'aide de l'objet **VkDeviceMemory**.
  - Dans certains cas, par exemple lorsqu'une mémoire est "Device Local" mais pas "Host visible", il faut des créer des mémoires temporaires (staging) pour transférer des donner entre le CPU et le GPU

- Certaines ressources nécessitent la création d'un pool géré par Vulkan. On passe le Pool en paramètre des fonctions d'allocations dans ce cas:
  - **Command Pool**, pour allouer des commandes
  - **Descriptor Pool**, pour allouer des descriptor sets (interface ressource/shader)

Mémoire : fonctions de base

- Ne présupposez rien ; utilisez toujours les fonctions de requêtes - **vkGetBufferMemoryRequirements**() et **vkGetImageMemoryRequirements**()- avant d'allouer de la mémoire.
  - L'agencement mémoire, l'alignement mémoire peut changer même pour deux allocations de même type / dimensions ...

- Allocation et de-allocation : **vkAllocateMemory**() et **vkFreeMemory**()

- Binding : consiste à attribuer une allocation à une ressource
  - **vkBindBufferMemory**() et **vkBindImageMemory**()

- Mapping : **vkMapMemory**() et **vkUnmapMemory**()
  - Le mapping peut être persistant

Images

- **Image** : zone mémoire destinée à contenir des pixels
- **Image View** : permet à l'application d'interpréter une Image
  - Spécifier un Format : interprétation des pixels de l'image
  - Spécifier un Color Space : colorimétrie des pixels de l'image (linéaire / sRGB)
- **Image Sub Resource** : une image peut être découpée en plusieurs sous parties ("ranges"), permet de spécifier la part concernée
  - Exemple: atlas / sprite sheet / layer …
- **Image Layout** : mode d'agencement mémoire de l'image en fonction du GPU

Presentation

- La swap chain va nous permettre de manipuler une surface à partir de laquelle on va extraire des images (**VkImage**).
  - On va être au minimum en double-buffering, donc au moins 2 images

- Idealement il faut examiner les capacités de la swap chain et adapter le nombre d'images en fonction du mode de présentation (fifo, immediat, relaxed, etc...)
  - En FIFO, deux images suffisent, mais dans le même cas en Immediate ou relaxed on peut avoir un goulot d'étranglement car le système présente plus vite l'image à la fenêtre qu'il ne lui en fait pour rendre l'image disponible pour le rendu suivant.

- On crée ensuite une vue (**VkImageView**) pour chacune de ces images, qui est une description typée de l'image. Ces différentes vues vont pouvoir être utilisées comme render targets (attachment).

Attachments (render targets)

- **Attachment Description**, décrit
  - le format de l'image attendu par le GPU,
  - les actions à effectuer lors de l'utilisation de l'image et au moment de répercuter les changements
  - Les changements (transitions) au niveau de l'Image Layout à ces mêmes moments
  - Toujours attendu sous la forme d'un tableau de VkAttachmentDescription.

- **Attachment Reference**, indique quel Attachment Description est concerné (référencé par son index dans un tableau) ainsi que l'Image Layout attendu par le GPU (doit correspondre avec l'Image Layout défini dans l'Image Transition Layout)

- **Sub Pass Description**, permet de spécifier le pipeline pour lequel on va utiliser ces attachments (graphics, compute, transfer...) et les types d'attachments (color, depth, resolve)

Image Layout Transition

- Pour des raisons de performances, le hardware des GPUs n'utilise pas les images sous forme linéaire (bien que cela soit possible).
  - Le format classique est le stockage en zig-zag ou séquence de Morton (Optimal).
  - Les GPUs mobiles sont encore plus complexe avec leur fonctionnement en Tile.

- Le stockage optimum peut varier d'une étape à une autre du pipeline particulièrement sur les GPU tile-based.

- Vulkan nous impose de spécifier l'agencement des images (image layout) en nous donnant la possibilité d'informer le driver des changements de forme/usage/accès de l'image (image layout transition).

- Ex: lorsque l'on récupère une image de la swap chain elle est UNDEFINED. Mais au moment de la présentation elle doit être en PRESENT. Entre temps on souhaite l'utiliser comme color buffer et il est nécessaire de transitionner de UNDEFINED à COLOR_ATTACHMENT, et de même il faudra transitionner de COLOR_ATTACHMENT vers PRESENT après la soumission des commandes.

Vulkan Synchro

Execution asynchrone d'une commande

- Une commande envoyée au GPU via un Command Buffer puis une Queue peut lire et/ou écrire en mémoire.
- De plus, multiples Command Buffer peuvent être envoyé à une Queue.

- Cette commande peut être plus ou moins latente et une commande suivante peut commencer son exécution sans que l'exécution de la commande précédente ai pris fin.
  - cette dernière commande peut très bien lire ou écrire dans la même zone mémoire
  - Read/write hazard
  - C'est encore moins garanti lorsqu'il s'agit d'une queue différente.

- le résultat d'une commande doit d'abord passer par deux étapes avant que la prochaine commande dépendante s'exécute correctement :
  - Disponibilité (**availability**), en lecture et/ou écriture, les opérations précédentes sont terminées
  - Visibilité (**visibility**) des changements locaux à une échelle plus large (cohérence des caches)

Spécificités de Vulkan

- Il n'y a aucune garantie que des commandes de natures différentes s'exécutent dans l'ordre défini (ex: graphics puis compute).
  - Une barrière (Execution/Pipeline Barrier ici) permet de séquencer l'exécution.

- Une commande qui ne provoque que l'écriture ou la lecture de données de manière indépendante n'a pas besoin de synchronisation. Dans les autres cas il faut une barrière (Pipeline ou Memory) :
- Write-After-Read (WAR), nécessite une barrière d'exécution (Pipeline)
- Read-After-Write (RAW) et Write-After-Write (WAW) nécessite les deux types de barrière.
  - Une PipelineBarrier pour marquer la dépendance, et une MemoryBarrier pour s'assurer que les changements ont bien eu lieu (availability) dans les deux cas, et visible également (visibility) dans le cas RAW.

- Dans une barrière on spécifie les étapes pour lesquelles on souhaite une synchro (src stages) et les étapes pour lesquelles le pipeline est bloqué (dst stages) tant que les étapes sources ne sont pas terminées.
- Les commandes concernées par la synchro sont celles définies avant et après la barrière.
  - Il est parfois utile d'utiliser un masque no-op (pas de synchro), on utilisera TOP_OF_PIPE pour un no-op en source et BOTTOM_OF_PIPE pour un no-op en destination.

Synchro implicite

- L'ordre des primitives (ex: index buffer) est garanti selon l'ordre de soumission (car vertex cache)
- Ordre de soumission des commandes garanti un **ordre correct** dans l'**enregistrement** et la **soumission** à une queue.
  - Ne signifie **PAS** qu'il existe implicitement une **dépendance d'exécution ou d'accès mémoire** entre deux soumissions successives à une queue.
  - Le Command Buffer N+1 ne se terminera pas avant le Command Buffer N
  - … mais peut être executé en parallèle !

- Les commandes d'action ou de synchronisation dans un Command Buffer s'exécutent dans le pipeline en suivant l'ordre de soumission
- Les commandes d'état établissent l'état courant au moment de l'exécution d'une commande d'action dans l'ordre de soumission.

- L'ordre des transitions est également garanti dans le cadre d'une Image Barrier.

Fonctions de synchronisation

- Il est recommandé de commencer par des synchronisations globales avant de tenter des synchronisations locales (à travers des vkCmdXXX), commencez simple.

- **vkQueueWaitIdle**() équivalent d'un glFinish(), bloque tant que queue active.
- **vkDeviceWaitIdle**() idem mais plus global, sur l'ensemble des queues

- **VkMemoryBarrier**, **VkBufferMemoryBarrier**, **VkImageMemoryBarrier** : paramètres qui agissent sur les accès mémoires de l'ensemble des objets existant au moment de l'exécution d'une commande associant une ou plusieurs barrières.
  - Certains objets comme les **Render Pass** impliquent des points de synchronisation à préférer aux commandes explicites de synchros (ex: **VkSubpassDependency**).

- **vkCmdPipelineBarrier**() défini une dépendance entre les commandes soumises dans la même Queue ou entre les commandes d'une même Sub pass.

Les primitives de synchronisation

- **Fences**, pour communiquer une complétion (GPU -> CPU par ex) avec **vkWaitForFences**() par exemple (vérifier l'exécution d'un Command Buffer, l'acquisition d'une image de la swap chain...)
  - Toujours créer une fence en état signalé avec **VK_FENCE_CREATE_SIGNALED_BIT**

- **Semaphores**, afin de contrôler les accès aux ressources par différents Command Buffer, différentes queues, différents systèmes (Windowing...)

- **Events**, permet un contrôle plus fin. On insère un event dans un Command Buffer. L'exécution de l'event ne bloque pas l'exécution mais passe en état signalé.
  - **vkCmdWaitEvents**() attend le signalement d'un "event" défini à un moment précis d'une command queue.

Retour à la swap chain

- Acquire -> Render -> Present => (next) => Acquire -> Render -> Present => (next) => ...

- Dans le schéma le plus simple, pour chaque buffer (au moins 2) on a autant d'images présentables que de Command Buffer. Il nous faut:

- 1 fence par Command Buffer afin de s'assurer que les commandes ont bien été exécutées avant de pouvoir le réutiliser / reset
  - On insert la fence dans un dernier VkSubmitInfo / vkQueueSubmit()
  - Une fence est un signal, il faut penser à vkResetFences()
- 1 semaphore par image, afin qu'au moment de l'acquisition d'une (next) image on soit sûr que l'image n'est plus utilisée par le WSI
  - À ce moment vkAcquireNextImageKHR() peut nous informer d'un changement (resize) ou d'un warning/erreur
- 1 semaphore par image, pour gérer le passage à la présentation
  - VkPresentInfoKHR / vkQueuePresentKHR()

Synchronisation des image layout

- On l'a dit, il est impératif qu'une image soit dans un "layout" VK_IMAGE_LAYOUT_PRESENT_SRC_KHR avant un vkQueuePresentKHR
- Cependant, l'image est initialement en VK_IMAGE_LAYOUT_UNDEFINED, ce qui indique qu'elle peut contenir n'importe quoi (aucune garantie de préservation n'a été définie).
- Il faut alors transitionner l'Image Layout cependant on ne peut pas le faire à n'importe quel moment.
- On doit au moins attendre que toutes les écritures (voire lectures) aient été terminés afin de transitionner correctement

vkCmdPipelineBarrier

- Introduit une barrière d'exécution garantissant l'ordre d'exécution des commandes.
  - Induit une dépendance entre différents stages du pipeline

- Attend que tout le travail soumis jusqu'à l'insertion de la barrière ait été traité par le pipeline.
  - On peut filtrer les stages concernés par avec le paramètre **srcStageMask**
- De même, pour les commandes qui suivent la barrière, on peut indiquer quelles parties du pipeline sont concernées avec **dstStageMask**.

VkMemoryBarrier

- Barrière sur l'accès aux ressources quelles qu'elles soient.

- Barrier.**srcAccessMask** : les types d'opération qui doivent se terminer avant les stages sources de la PipelineBarrier, généralement des écritures, 0 = rien
  - Garantie que la ressource a finie d'être modifiée à ce moment-là (availability, ou flush)
- Barrier.**dstAccessMask** : les types d'opération qui doivent être correctement visible par les stages destination de la PipelineBarrier. 0 = rien.
  - Garantie la visibilité (visibility, invalidate cache) de la ressource après la barrière.

- Cela permet d'obtenir une cohérence au niveau de la mémoire (CPU<->GPU ou Thread<->Thread ou Stage<->Stage)

VkImageMemoryBarrier

- Passée en paramètre de **vkCmdPipelineBarrier**()

- Même propriétés que l'on vient de voir mais pour les VkImage

- Barrier.**oldLayout** : le layout actuel de l'image
- Barrier.**newLayout** : le layout vers lequel on souhaite transitionner.

- Contrairement à VkMemoryBarrier, il n'est pas nécessaire que la mémoire de l'image soit visible (pas besoin que les autres étapes attendent que les caches soient cohérents) mais seulement disponible (available) c’est-à-dire que la ressource a bel et bien été modifiée.

Render Pass

- Une Render Pass Instance nous permet d'agir sur un framebuffer.
  - Plus spécifiquement, un framebuffer object nécessite une Render Pass Instance !
- On peut spécifier les attachments (render targets) concernées pour chaque subpass potentielle mais également spécifier les dépendances entre les subpass ce qui permet de réduire un petit peu les problématiques de configuration de barrières.
  - On peut par exemple utiliser une subpass pour résoudre une passe de rendu multi-échantillonnée (MSAA) vers une target standard, ou bien dessiner dans un depth buffer seulement, depth buffer que l'on réutilisera pour les rendus suivants en lecture seule.
- Il est important de bien spécifier au début et à la fin (mais également à chaque subpass) comment on va lire et éventuellement écrire dans les render targets
  - Par exemple, spécifier si au début on doit clear une image, ou bien si on va lire et ou écrire l'image, est-ce que l'on va préserver le résultat d'une subpass précédente etc..

Render Pass suite

- RenderPassBeginInfo a un rôle important car il permet de customiser légèrement la Render Pass.
- Permet de mettre en relation RenderPass et Framebuffer
- Mais également de spécifier les clear colors des différents attachments.

Retour sur la Swap Chain 2, cas d'une RenderPass

- Pour que le rendu dans une RenderPass soit correctement utilisable par la Swap Chain il faudrait spécifier une barrière mémoire avec les propriétés:

```
// nous sommes sur que l'image ne sera plus modifiee jusqu’à un prochain acquire
PresentBarrier.srcStageMask = VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT;
PresentBarrier.dstStageMask = VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT;
// on souhaite transitionner l'image layout en présentable
presentBarrier.oldLayout = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL;
presentBarrier.newLayout = VK_IMAGE_LAYOUT_PRESENT_SRC_KHR;
// on indique attendre la fin des ecritures et on quitte le pipeline
PresentBarrier.srcAccessMask = VK_COLOR_ATTACHMENT_WRITE_BIT;
PresentBarrier.dstAccessMask = 0;
```

Avantages de la RenderPass

Retour sur la Swap Chain 3, cas Cmd Clear

- Dans notre premier programme nous avons utilisé vkCmdClearColorImage(), sans RenderPass donc, mais cette commande induit un transfert - type memset(), cela donne :

```
// nous sommes sur que l'image ne sera plus modifiee jusqu'à un prochain acquire
PresentBarrier.srcStageMask = VK_PIPELINE_STAGE_TRANSFER_BIT;
PresentBarrier.dstStageMask = VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT;
// on souhaite transitionner l'image layout en présentable
presentBarrier.oldLayout = VK_IMAGE_LAYOUT_TRANSFER_DST;
presentBarrier.newLayout = VK_IMAGE_LAYOUT_PRESENT_SRC_KHR;
// on indique attendre la fin des ecritures et on quitte le pipeline
PresentBarrier.srcAccessMask = VK_COLOR_ATTACHMENT_WRITE_BIT;
PresentBarrier.dstAccessMask = 0;
```

Mais aussi avant vkCmdClearColorImage()

- En effet, au moment de vkAcquireNextImageKHR() l'image est en layout "undefined'. Il faut donc transitionner l'image layout en Transfert :

```
// on ne fait rien avec l'image tant que l'on n'en est pas encore au stage "transfer"
PresentBarrier.srcStageMask = VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT;
PresentBarrier.dstStageMask = VK_PIPELINE_STAGE_TRANSFER_BIT;
// on souhaite transitionner l'image layout en "transfert"
presentBarrier.oldLayout = VK_IMAGE_LAYOUT_UNDEFINED;
presentBarrier.newLayout = VK_IMAGE_LAYOUT_TRANSFER_DST;
// on indique que les opérations de type transfert (clear ici) doivent attendre
PresentBarrier.srcAccessMask = 0;
PresentBarrier.dstAccessMask = VK_ACCESS_TRANSFER_WRITE_BIT;
```

En pratique

Pipeline

- Le Pipeline object (**VkPipeline**) connecte les étapes (stages) avec les différentes ressources, shaders, render passes, framebuffers. Il permet de définir les états de rendu.
  - A noter que la configuration d'un pipeline est statique (immuable) mise à part quelques étapes dynamiques (essentiellement axées autour du viewport/clipping)

- De plus c'est un objet assez coûteux à créer, et il en faut plusieurs si on a besoin de configuration d'étapes différentes. C'est pourquoi Vulkan propose le Pipeline Cache Object, un mécanisme pour sauvegarder un Pipeline object pour éviter d'avoir à le recréer complètement

- Pipeline layout : permet de spécifier les descriptors sets utilisés par ce pipeline et par quelle(s) partie(s) du pipeline ils se verront utilisables.

Utilisation des shaders

- On va utiliser le GLSL dans une variante adaptée à Vulkan.

- Deux outils pour compiler le glsl en SPIR-V: glslangvalidator (LunarG) et glslc (google, ligne de commande type gcc)
  - **glslangvalidator.exe** -V shader.vert -o shader.vert.spv
    - L'extension détermine le type de shader (.vert, .tesc, .tese, .geom, .frag, .comp) ou explicitement avec -S
  - **glslc.exe** shader.vert -o shader.vert.spv
    - Idem mais explicitement avec –fshader-stage=<stage>

- **Shader Module** : le bytecode SPIR-V d'un shader stocké par le GPU et utilisable dans un **VkPipeline** au moment de la création de ce dernier, via un tableau de structures **VkPipelineShaderStageCreateInfo** qui décrit les shaders utilisés (et à quelles étapes). Ce qui signifie qu'il faut un VkPipeline par combinaison de shaders.

- Voir https://vulkan-tutorial.com/Drawing_a_triangle/Graphics_pipeline_basics/Shader_modules

Input Layout / VAO

- **VkVertexInputBindingDescription**, permet de spécifier, pour un vertex buffer rattaché (bind) au binding point (un nombre) défini ici, le stride et l'input rate (ce dernier pour l'instancing, doit-on lire des données à chaque exécution d'un vertex, ou une fois par instance, toutes les N instances...)

- **VkVertexInputLayoutDescription**, sert à indiquer, pour un binding point particulier, quel est le "location" correspondant dans le vertex shader, le format des données ainsi qu'un éventuel offset.

- Vertex Input State (**VkPipelineVertexInputStateCreateInfo**) affecte un tableau de Vertex Input Binding et un tableau de Vertex Input Layout –tous deux optionnels- au pipeline.

- Vertex Assembly State (**VkPipelineVertexAssemblyStateCreateInfo**), défini une topologie (point, line, triangle...). Ce qui signifie que pour dessiner des points et des triangles il faut créer au moins un pipeline pour chaque topologie.

Fenêtrage et rasterization

- **VkPipelineViewportStateCreateInfo** permet de définir les viewports et scissors à utiliser durant le rendu. Il s'agit de valeurs fixes pour le pipeline mais on peut utiliser l'équivalent en Dynamic State pour une simple update.

- **VkPipelineRasterizerStateCreateInfo** contient les états de la rasterization, culling, taille des lignes, bias, etc... certains états sont reconfigurables dynamiquement.

- **VkPipelineMultisampleStateCreateInfo**, sert à définir le nombre d'échantillons par pixel.

- **VkDynamicStateCreateInfo**, il existe une petite dizaine d'états (**VkDynamicState**) qui peuvent être reconfiguré durant une RenderPass sans avoir besoin de recréer un VkPipeline. Néanmoins cela peut parfois être très coûteux (sensible au nombre de dynamic).

Tests & output merger

- Depth Stencil
- Color Blending

Rendu Hors Ecran

Objectif : plus rien qui s'affiche

- Plutôt que de dessiner directement dans l'image de la swapchain on va dessiner dans une VkImage que l'on aura créé préalablement

- 1 VkImage
- 1 VkImageView

- Dans notre Framebuffer, en lieu et place de l'image view de la swapchain on va utiliser l'image view que l'on vient de créer.

- Et voilà (n'oubliez pas de les détruire en quittant le programme)...
- Enfin presque, cette fois-ci le "finalLayout" de l'attachment ne peut pas être PRESENT_SRC, on peut utiliser GENERAL ou mieux COLOR_ATTACHMENT_OPTIMAL

Comment récupérer le rendu hors écran ?

- Il faut d'abord s'assurer que le résultat de la RenderPass est bien stocké (availability). "storeOp" doit être STORE_OP_STORE_BIT.

- Plusieurs options nous sont offertes :

- Lancer une commande de copie/transfert (vkCmdCopy/Blit)
  - Implique de transitionner l'image layout en TRANSFER puis en PRESENT…mouais

- Faire une seconde RenderPass avec un shader qui applique la texture sur un quad ou un triangle fullscreen
  - Implique de…faire une seconde render pass….

Input Attachment

- Définie via `pInputAttachments` de VkSubpassDescription, que l'on va ajouter à une seconde Sub Pass
  - **VkCmdNextSubPass**() déclenche le changement de subpass vers la suivante

- Necessite un second pipeline car on va utiliser un autre shader
  - Similaire au fait d'executer une seconde Render Pass mais plus optimal

- Le GLSL Vulkan offre un type nouveau, subpassInput, pour accéder à un input attachment

```
layout (input_attachment_index = 0, set = 0, binding = 0) uniform subpassInput inputColor;
```

- Pour sampler l'input attachment on utilise la fonction subpassLoad()

```
vec3 color = subpassLoad(inputColor).rgb;
```

- https://www.saschawillems.de/blog/2018/07/19/vulkan-input-attachments-and-sub-passes/

Descriptor Set

- Un Descriptor Set permet de décrire les ressources utilisables par un shader pour un "set" donné.
  - Dans un shader on peut spécifier l'identifiant du set auquel on souhaite accéder

```
layout (set = 0, binding = 0) uniform …;
```

- Au niveau du pilote, un descriptor est une méta-donnée spécifiant le type et la composition de la ressource (buffer, image, image + sampler, sampler...)

- Du point de vue du programmeur un descriptor se caractérise plutôt par son "layout" soit une description des ressources (type, taille, offset, …) que, pour des raisons d'optimisation, on passe en groupe, donc un Descriptor Set.

Descriptor Set

- VkDescriptorSetLayout

- VkDescriptorSet alloué depuis un VkDescriptorPool et rattaché à un VkDescriptorSetLayout. Le nombre de descripteur par pool est spécifiable via un tableau de VkDescriptorPoolSize.

- VkDescriptorSetLayout est rattaché à un VkPipelineLayout