ISART DIGITAL

VULKAN

Malek Bengougam

m.bengougam@isartdigital.com

ISART DIGITAL

PHILOSOPHIE DE L'API

PHILOSOPHIE DE L'API

Les principes de base

Changement de paradigme

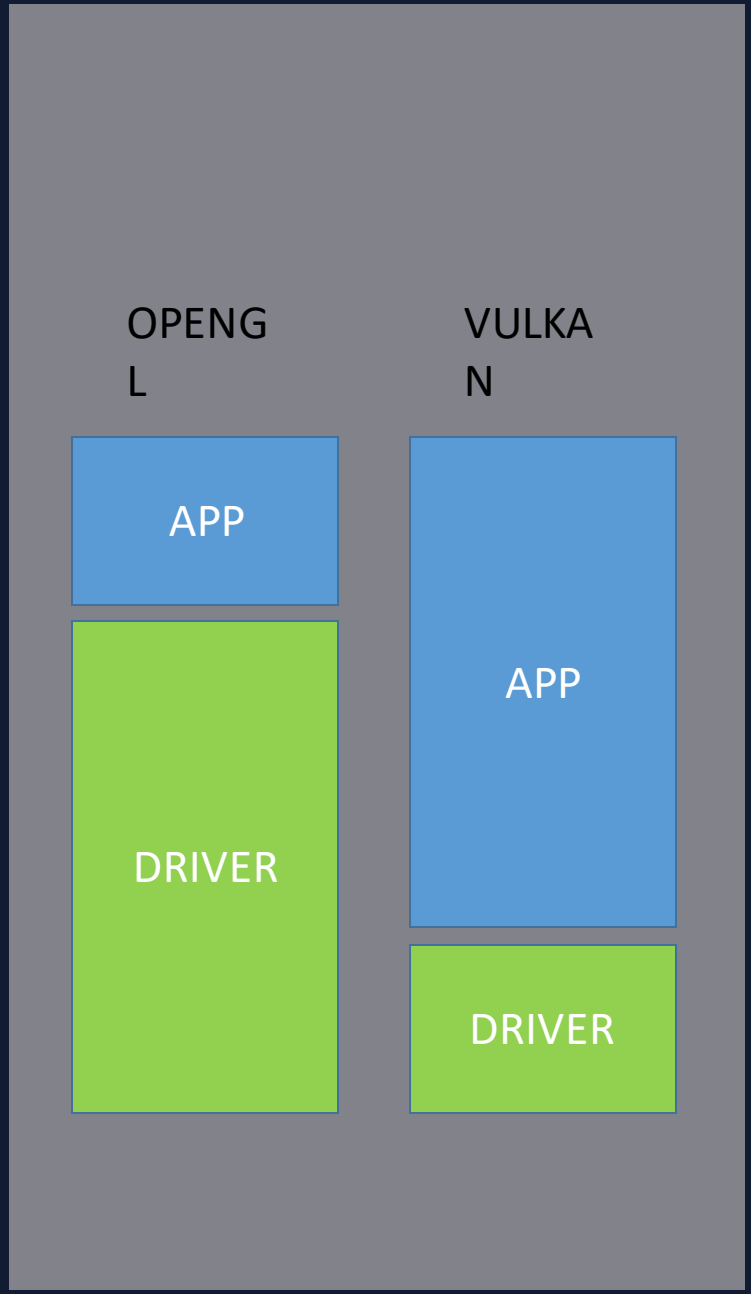
Re-design quasi obligatoire du Renderer pour espérer profiter du gain apporté par Vulkan.

Un portage direct n'aura qu'une influence mineure, voire négative si l'on si prend mal !

Les drivers OpenGL/Direct3D11 masquent une bonne partie des problématiques liées aux performances, cela implique beaucoup d'efforts pour faire mieux mais c'est souvent payant.

Mais ces drivers font également un travail complexe même lorsque le programmeur fait les choses correctement :

- Détections d'erreurs : coût non négligeable au runtime même pour une implémentation « parfaite »
- Validation : le plus gros morceaux. Le driver s'assure que tout colle parfaitement (ex: VAO avec attributs des shaders, format des textures, usage des buffers etc...) … et patch les commandes si il trouve des problèmes !
- Emplacement des données : la plupart des usages sont des requêtes (« hint ») appliquées à une heuristique interne, aucune garantie que le driver les suivent
- Temporisation : afin de pouvoir soumettre les commandes rapidement (et éventuellement les patcher) puis redonner la main au programmeur, le driver stocke plusieurs frames (et copies de buffers !) avant de les envoyer au GPU => LAG. Le temps de rendu est donc difficile prédictible.



ISART DIGITAL

PHILOSOPHIE DE L'API

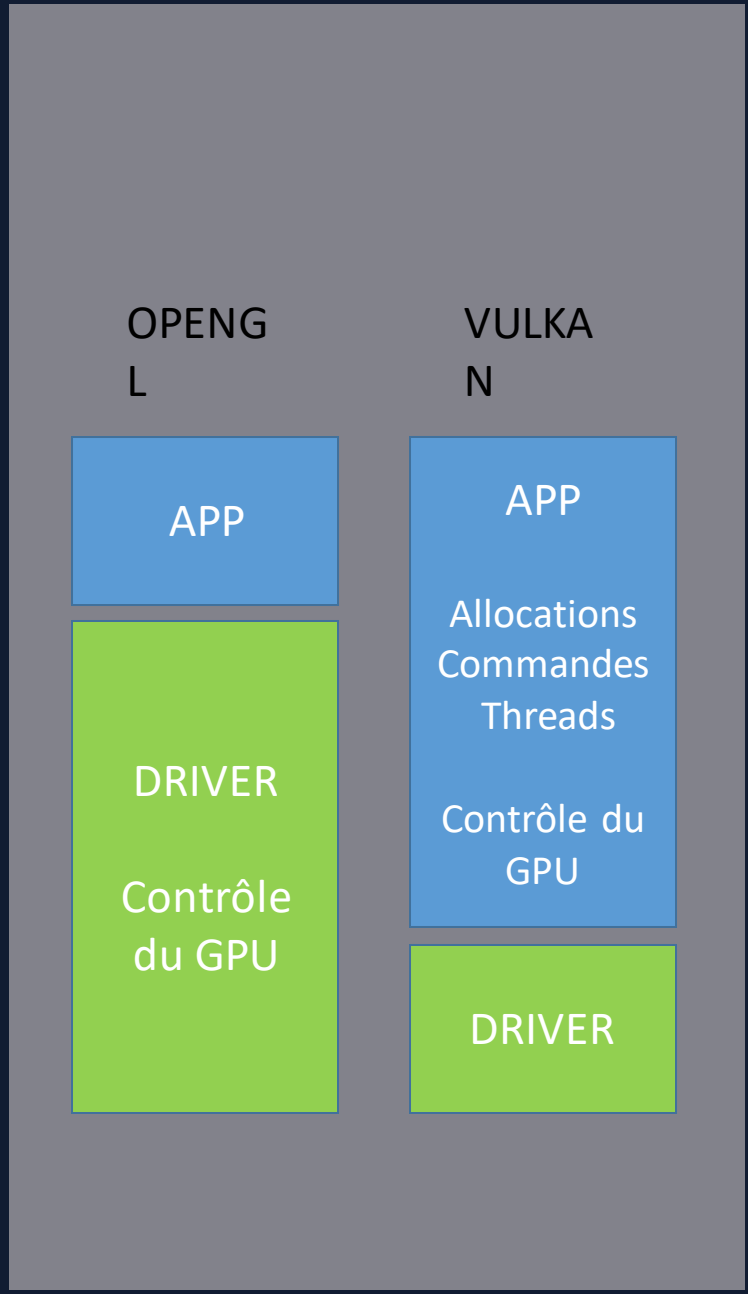
Les principes de base

Une API explicite

Une réponse aux besoins des développeurs d'avoir une API similaire à celles des consoles de jeu. Cela se traduit par une implication fortement réduite du driver et un contrôle quasi-total du programmeur sur le GPU. …induit une complexité accrue par rapport à OpenGL ou Direct3D11 -> plus d'efforts de la part du programmeur.

On peut résumer les principes de cette API ainsi :

- Pas de machine à états globale (partiellement déjà le cas pour Direct3D11)
- Contrôle total sur les ressources et la mémoire
- Séparation des ressources et de leur usage
- Données réutilisables et groupables
- Commandes de rendu/calcul explicites
- Exploitation du parallélisme du CPU et du GPU
- Comportement et performances prédictibles
- Adapté aux multiples configurations et extensible
- Validation, debugging etc.. optionnelles



ISART DIGITAL

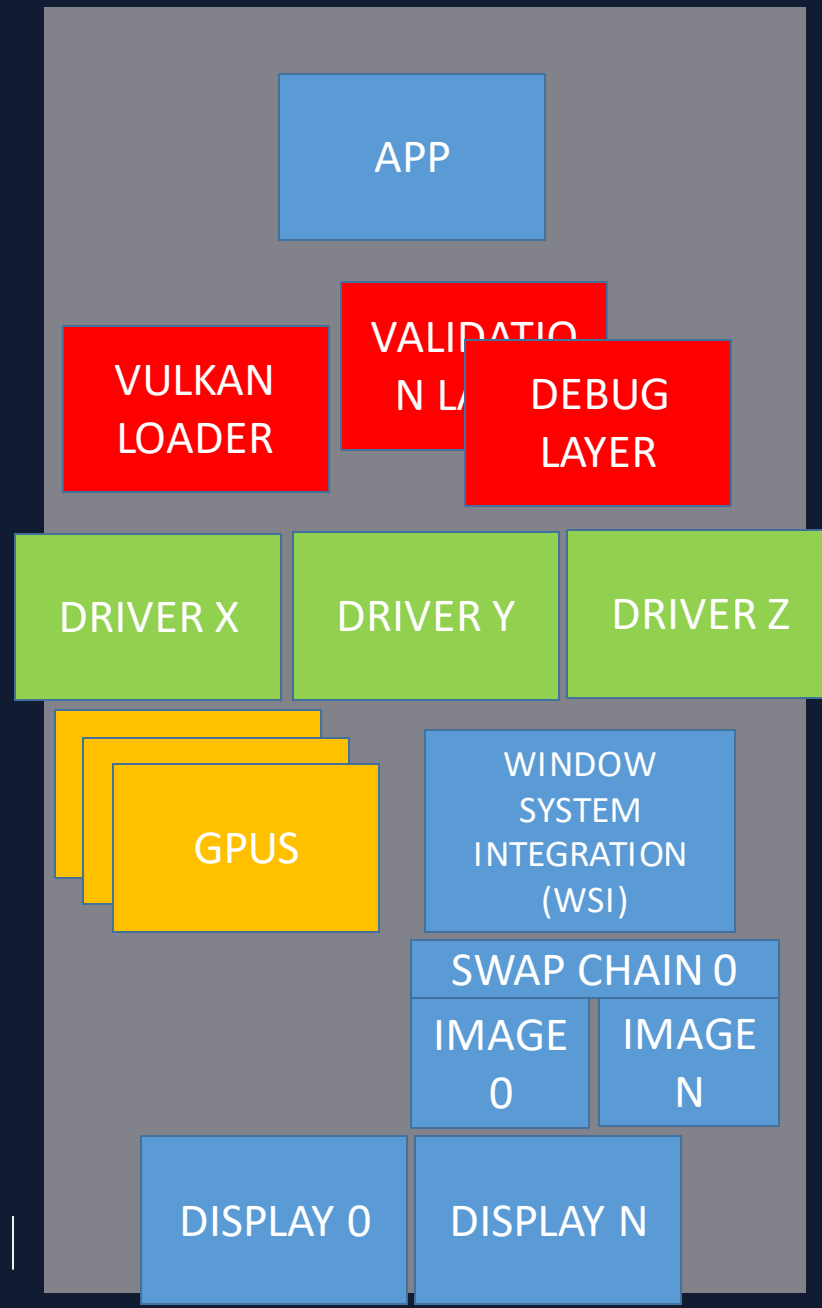
PHILOSOPHIE DE L'API

Les principes de base

Modèle de programmation de Vulkan

A environnement complexe, initialisation complexe.

- Initialisation du contexte (instance) et du hardware
- Spécification des layers optionnels
- Initialisation du système de fenêtrage et swap-chain(s)
- Préparation des ressources
- Préparation des pipelines (shaders, states, descriptors …)
- do
- Préparation des commandes (batches)
- Soumission des commandes (render passes)
- Présentation (display)
- while



ISART DIGITAL

PHILOSOPHIE DE L'API

Les principes de base

Nomenclatures

- Notation hongroise minimaliste ('p' pour les pointeurs essentiellement)
- Les types et objets Vulkan sont préfixés par 'Vk' , ex: VkInstance, VkBuffer
- Les fonctions sont préfixées par 'vk' , ou 'vkCmd' pour les commandes ex: vkBindImageMemory(), vkCmdDraw()
- Les enums et macros sont en majuscules, chaque mot séparé par un '_', ex : VK_NULL_HANDLE, VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT
- Les bitfields sont suffixés par 'Flags'
- Parmi les types on trouve des structures servant à initialiser ou créer un objet (info). On retrouve un schéma récurrent :

```
// Exemple 1 : Création
VkXXXCreateInfo createInfo = {};
createInfo.sType =
VK_STRUCTURE_TYPE_XXX_CREATE_INFO;
createInfo.pNext = nullptr; // ext. futures
createInfo.xxx =...;

VkXXX object;
VkResult res = vkCreateXXX(&createInfo,
nullptr, &object);
if (res != VK_SUCCESS) {}

// Exemple 2 : Passage de paramètres
VkXXXBeginInfo beginInfo = {};
beginInfo.sType =
VK_STRUCTURE_TYPE_XXX_BEGIN_INFO;
beginInfo.xxx =...;

VkXXX object;
res = VkBeginXXX(object, &beginInfo);
```

ISART DIGITAL

PHILOSOPHIE DE L'API

Les principes de base

Les fonctions de gestion

C'est le Logical Device (abstraction du GPU) qui permet de gérer les handles des objets

On retrouve une convention de code communes aux grandes étapes:

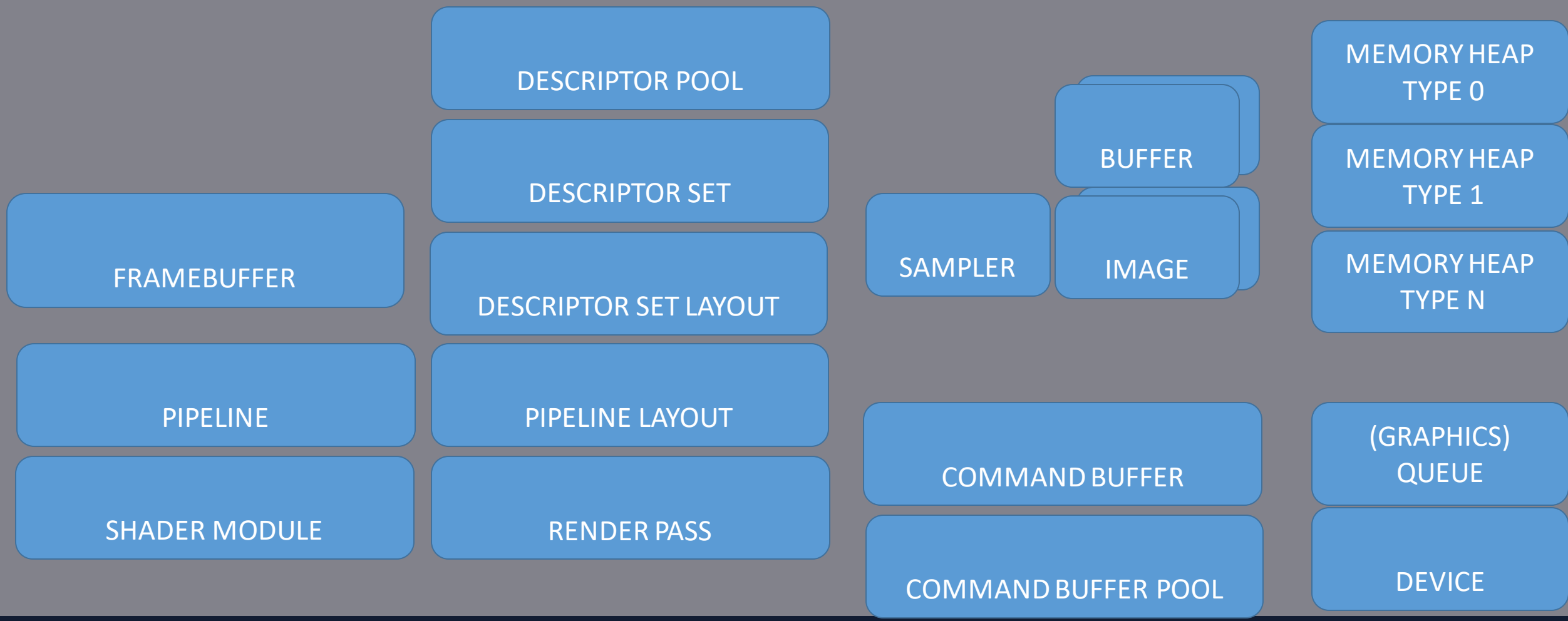
- Creation : vkCreateXXX(…,&info, allocator = nullptr, &xxx handle)
- Destruction vkDestroyXXX(…, xxx handle, allocator = nullptr)
- Allocation : vkAllocateXXXs(…, &info, &xxx[0]) // notez le 's' à la fin
- Bind : vkBindXXXs() // pour attribuer la mémoire aux ressources
- Libération : vkFreeXXXs(…, pool, …, count, &xxx[0])
- vkBeginXXX() / vkEndXXX() // command buffers, render passes…

```
VkCommandPoolCreateInfo info = ...;
vkCreateCommandPool(Device, &info, 0,
CommandPool);
vkDestroyCommandPool(Device, CommandPool, 0);

VkSemaphoreCreateInfo info = ...;
vkCreateSemaphore(Device, &info, 0, &Sema);
vkDestroySemaphore(Device, Sema, 0);

VkCommandBufferAllocateInfo info = ...;
vkAllocateCommandBuffers(Device, &info
/*contient CommandPool et BufferCount ...*/,
CommandBuffers);
vkFreeCommandBuffers(Device, CommandPool,
BufferCount, CommandBuffers);
```

ISART DIGITAL


DESCRIPTOR POOL
DESCRIPTOR SET
FRAMEBUFFER
DESCRIPTOR SET LAYOUT
BUFFER
SAMPLER
IMAGE
MEMORY HEAP TYPE 0
MEMORY HEAP TYPE 1
MEMORY HEAP TYPE N
PIPELINE
PIPELINE LAYOUT
SHADER MODULE
RENDER PASS
COMMAND BUFFER
COMMAND BUFFER POOL
(GRAPHICS) QUEUE
DEVICE

ISART DIGITAL