Utilisation et spécificités de l'API graphique Vulkan

Jean-Louis Boudrand - 22/08/2016

Abstract

Très intéressé par les différents aspects touchant à la programmation moteur, j'ai voulu dans le cadre de mon travail de recherche étudier la nouvelle génération d'API graphiques (DirectX 12, Vulkan, Metal…)
Je me suis en particulier intéressé à Vulkan, et aux différences par rapport à OpenGL dont Vulkan se présente comme étant le successeur.

Je vais présenter mon travail en 3 parties :
- Je vais commencer par présenter le fonctionnement global de Vulkan, les différents éléments de l'API et comment les utiliser pour effectuer des rendus. Je vais en particulier détailler les spécificités de cette nouvelle API.
- Je vais ensuite expliquer la conception de mon application de démonstration, et commenter celle-ci en expliquant quelques bonnes pratiques et optimisations possibles.
- Puis pour conclure je vais revenir sur les principales différences avec OpenGL et citer les avantages et inconvénients de Vulkan par rapport à celui-ci.

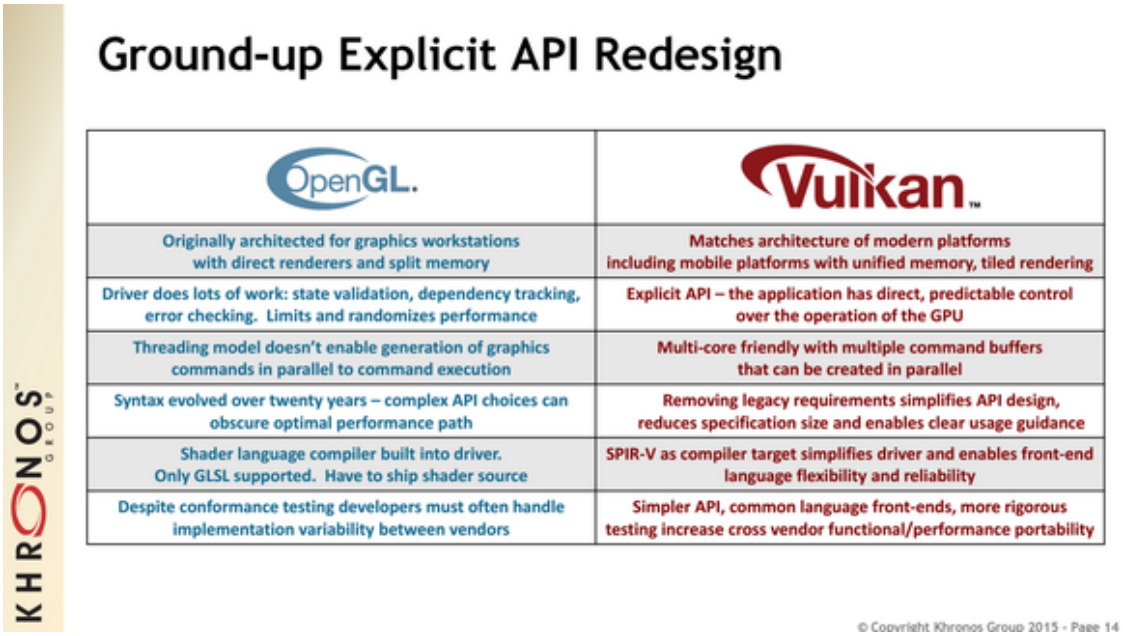
Introduction

Depuis les années 90, 2 API graphiques sont principalement utilisés pour développer des applications sur PC, DirectX créé par Microsoft en 1995, et OpenGL, développé en 1992 par Silicon Graphics puis actuellement maintenu par le Khronos Group (consortium regroupant des entreprises comme AMD, Nvidia, Apple…)
Depuis, ces API ont énormément évoluées au fil des versions, dans le but d'en améliorer les performances, et en apportant d'importantes nouveautés, comme par exemple les shaders il y a maintenant plus de 10 ans.

Au fil des versions, des fonctionnalités ont été ajoutées, supprimées, dépréciées, mais le fonctionnement ba base des API est resté globalement similaire.
C'est cet héritage d'une architecture vieille de plus de vingt ans et les défauts qui lui sont inhérents qui ont conduit le Khronos Group à repenser et à changer radicalement l'API OpenGL afin d'en créer une nouvelle : Vulkan

Vulkan s'appuie sur le projet d'API Mantle initié par AMD en 2013, puis abandonné et en partie repris par Vulkan, disponible depuis février 2016.

Cette nouvelle API a plusieurs objectifs :
- Donner un meilleur contrôle du GPU : L'API est plus bas niveau qu'OpenGL, ce qui permet de simplifier les drivers, optimiser l'utilisation CPU et permettre une gestion des ressources adaptée à l'application. Davantage de responsabilités sont laissées à l'utilisateur et moins au pilote.
- Diminuer les différences entre les pilotes grâce à cette API plus bas niveau qui laisse davantage de responsabilité à l'utilisateur et grâce à l'utilisation de shaders compilés.
- Être multi-platforme, Vulkan fonctionne aussi bien sur un PC, un Mac ou un smartphone.
- Faciliter le Multithreading avec une architecture basée sur des files de commandes remplissables depuis plusieurs threads.



L'API change aussi complètement sa philosophie en ne fonctionnant plus comme une machine à état sur laquelle on modifie les objets en les « bindant » pour pouvoir les modifier, mais en manipulant des « handles » passés aux différentes fonctions.

1. Utilisation

Je vais dans cette première partie évoquer rapidement les prérequis pour utiiser Vulkan et présenter les principaux éléments de l'API permettant d'effectuer des rendus.

Environnement

Vulkan étant une API récente, il faut pour pouvoir utiliser et développé des programmes basés sur Vulkan, une carte graphique récente et des pilotes à jour.
Toutes les cartes supportant OpenGL 4.3 peuvent supporter Vulkan.

Pour développer avec Vulkan, l'utilisation du Vulkan SDK de LunarG n'est théoriquement pas indispensable mais qui contient un certain nombre de choses utiles : loader, doc, validation layers, exemples…

La gestion de la fenêtre d'affiche et les outils mathématiques nécessitent d'utiliser d'autres bibliothèques en complément.

Je vais maintenant présenter le fonctionnement de l'API en présentant les différents types d'objets que l'on trouve dans l'API et leurs rôles :

Instance

L'instance (*VkInstance*) est le premier objet à créer, qui nous permettra ensuite de créer des « Device ».
La création de l'instance demande plusieurs types de paramètres :
- Des paramètres sur la version de l'API, l'application, le moteur, ce qui permet de faciliter l'optimisation des drivers pour une application ou un moteur spécifique.
- Le choix des extensions, qui comme pour OpenGL permettent d'intégrer des fonctionnalités supplémentaires.
- Le choix des « validation layers ».

Le concept de « validation layer » est une nouveauté introduite par Vulkan.
Un validation layer est un layer entre le programme et Vulkan, qui intercepte nos appels de fonctions, pour effectuer des vérifications et traitements avant de les transmettre à Vulkan.
Ils permettent par exemple de détecter des fuites de mémoire ou qu'un paramètre d'une fonction est bien valide.
Ce mécanisme rend la gestion des erreurs complètement facultative et personnalisable (il est possible d'écrire ses propres layers).

En effet, là où OpenGL gérait les erreurs et permettait d'accéder à celle-ci avec *glError,* Vulkan réduit la gestion des erreurs au minimum et ces dernières provoquent des crashs ou des comportements indéfinis.
Toute la gestion des erreurs se fait maintenant via les validation layers qui sont facilement activables ou désactivables.
Cela permet d'avoir accès à la gestion des erreurs en phase de développement et de désactiver celle-ci en release pour gagner en performance sur le CPU.
Un callback est rattaché aux validation layers, pour par exemple afficher les messages envoyés par les layers dans la console ou dans un fichier.

Le Vulkan SDK est fourni avec un plusieurs validation layers permettant d'identifier la plupart des erreurs courantes.

Logical device

Une fois l'instance créée on peut créer un « device » (*VkDevice*) en fournissant différents paramètres :

- Le choix du « physical device » (*VkPhysicalDevice*).
Il est possible de lister les différentes cartes graphiques présentes et de choisir laquelle utiliser selon différents critères : disponibilité d'une feature particulière, modèle de la carte, quantité de VRAM…
- Les « devices features » permettent d'activer ou non des fonctionnalités du device, par exemple l'utilisation des geometry shaders.
- Les extensions spécifiques au device.
- Les validation layers spécifiques au device.
- La configuration des queues (*VkQueue* ou files en français) utilisées.

Les queues sont les files de commandes dans lesquelles nous allons envoyer les différentes instructions que la carte graphique devra exécuter.
Il s'agit là encore d'une nouveauté de la dernière génération d'API graphique : les commandes (draw, binding de textures ou de buffer…), ne sont plus exécutés directement, mais enregistrées dans des buffers de commandes qui sont envoyés à ces files de commandes. Les commandes d'une même file seront ensuite exécutées dans l'ordre (FIFO), mais de façon asynchrone, après que l'on ait envoyé les instructions dans la file.
Cette nouvelle logique, plus proche du fonctionnement du GPU, en plus d'être plus performante est aussi intéressante vis-à-vis des possibilités de multithreading : plusieurs files peuvent exister en parallèles et les files peuvent être remplies depuis plusieurs threads.

Il existe différents types de files adaptés aux différents types d'opérations effectués par le device. (graphiques, transferts de donnés, calculs, présentation…)

Cette étape consiste donc à définir combien de files de chaque famille (une famille supporte un ou plusieurs types d'opérations) on souhaite créer et utiliser.
Les familles de files disponibles dépendent du physical device choisit.

Une fois le device créé on peut récupérer des handles sur les queues pour pouvoir les utiliser.
Typiquement, on va créer et récupérer au moins une queue pour les opérations graphiques et de présentation (envoi des images à la « swap chain » pour affichage).
Les opérations de transferts (transférer une image entre différentes zones de la VRAM par exemple) sont toujours supportées par les files supportant les opérations graphiques.

Swap chain

La swap chain est l'objet permettant de gérer les différentes images utilisées pour effectuer nos rendus et les afficher à l'écran.

La swap chain est une extension de l'API, car contrairement à OpenGL, Vulkan peut-être utilisé sans faire de rendu (Vulkan peut-être utilisé en tant qu'API de calculs par exemple).
Il n'y a donc pas de buffer de rendu créé automatiquement à la création de l'instance, il faut les créer

avec la swap chain.

La création de la swap chain nécessite de créer une « surface », qui est un handle sur la fenêtre sur laquelle on souhaite effectuer nos rendus.
La création de cette surface se fait avec des extensions spécifiques aux différents OS, mais cela se fait assez facilement avec une librairie multiplateforme comme GLFW qui dispose d'une fonction pour créer la surface correspondante à la fenêtre.

La configuration de la swap chain consiste à renseigner des paramètres comme la taille des images, le format des pixels (RGBA par exemple) mais aussi combien d'images on souhaite utiliser pour celle-ci, son mode de présentation ou encore un « pre-transform » (pour tourner ou retourner les images par exemple).
Ces paramètres, et en particulier le mode de rendu permettent de mettre en place un double ou triple buffering.

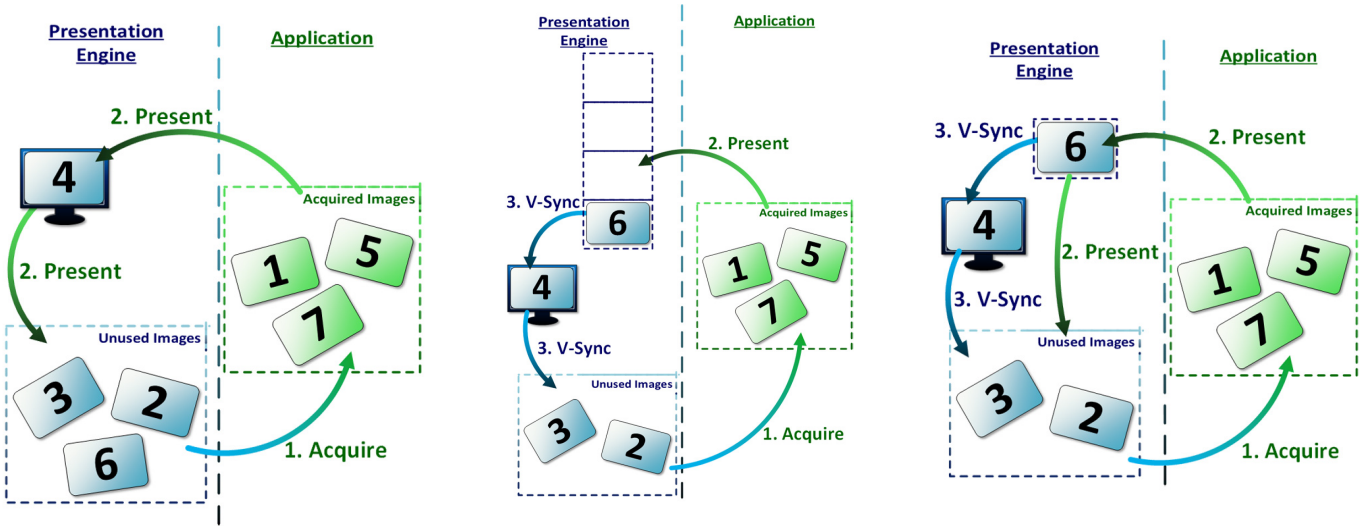


Illustration des différents modes de rendu de la swap chain.
De gauche à droite le mode immédiat, où l'image envoyée est directement affichée, le mode FIFO, où les images envoyées sont stockées dans une pile FIFO puis affichées suivant la V-Sync, puis le mode « mailbox » où seul l'image envoyée la plus récente est gardée dans le buffer avant d'être affichée.

Une fois la swap chain créée, une boucle de rendu typique consistera à :
- Demander une image à la swap chain. (*vkAcquireNextImageKHR*)
- Effectuer le rendu sur cette image en envoyant des commandes à la file graphique.
- Renvoyer l'image à la swap chain pour affichage en utilisant la file de présentation. (*vkQueuePresentKHR*)

Il y a un cependant un facteur important à prendre en compte dans cette boucle : l'acquisition d'une image de la swap chain ainsi que l'exécution des commandes envoyées aux files ne se font pas immédiatement.
Il faut donc veiller à ce que les commandes d'affichage ne s'exécute qu'une fois l'image acquise et que l'image est renvoyée pour affichage seulement une fois le rendu effectué.

Pour ça Vulkan utilise 2 primitives permettant de gérer ce genre de synchronisations : les sémaphores (*VkSemaphore*) et les fences (*VkFence*).
On pourra donc résoudre ce problème en utilisant 2 sémaphores.

Graphics pipeline

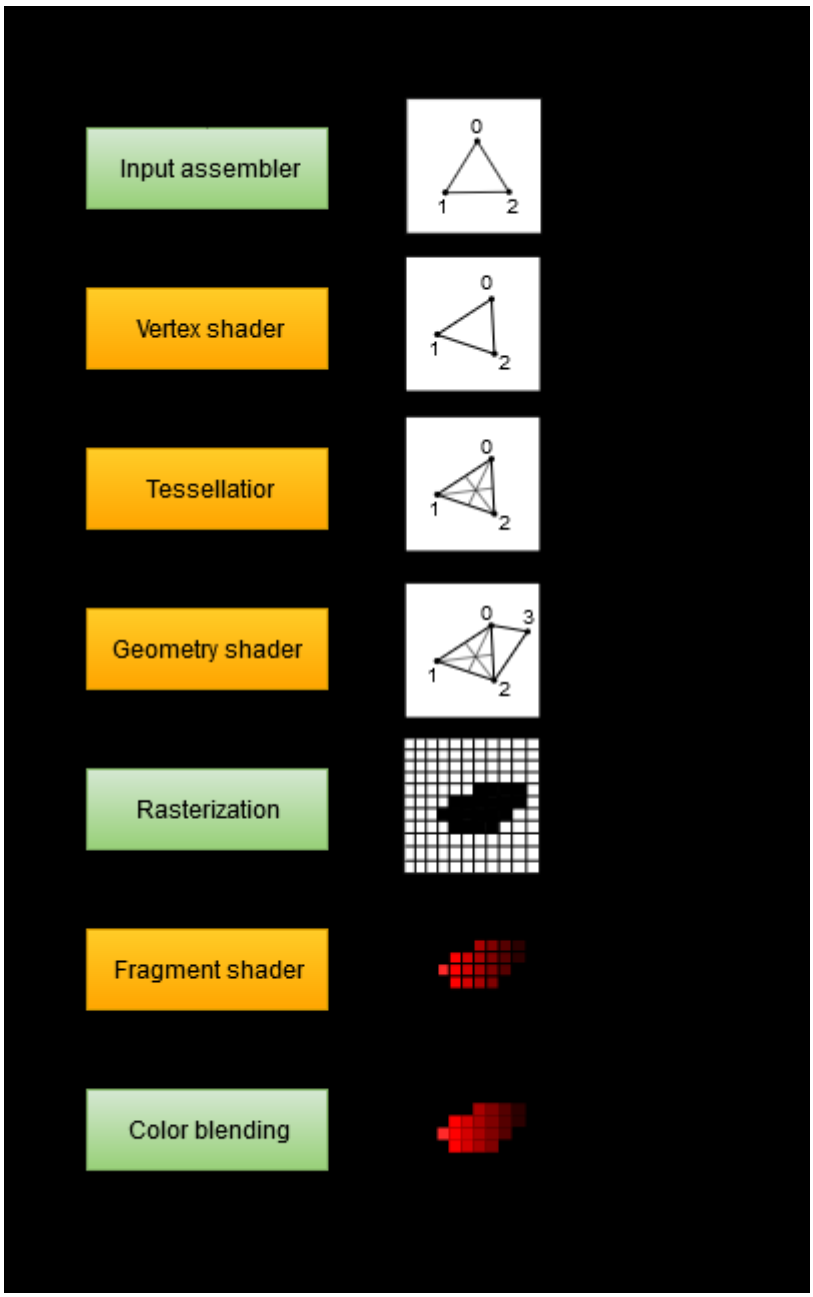
Le pipeline graphique est assez similaire à celui que l'on trouve dans OpenGL, il passe par les mêmes étapes classiques tel que le vertex shader, la rastérisation ou encore le fragment shader.

Dans Vulkan, cependant, le pipeline graphique constitue un objet à part entière, qui est construit à l'avance et qui est bindé au moment du rendu,

Les parties fixes du pipeline sont configurables via des structures qui seront passés en paramètres à la création du pipeline, tandis que la partie programmable consiste à charger les shaders en mémoire et à indiquer pour quel parti du pipeline on utilise chacun d'eux.
On trouve dans la configuration du pipeline fixe des éléments tel que le type de primitives à dessiner, l'utilisation du back/frontface culling, ou l'utilisation du depth buffer.

La configuration du pipeline est fixe, ce qui signifie que si l'on souhaite modifier un paramètre il faudra en recréer un nouveau.
Cependant quelques paramètres qualifiés de « dynamiques » du pipeline sont prévus pour être modifiable au moment du rendu (la taille du viewport par exemple)

Le fait de construire le pipeline en amont permet de réutiliser les pipelines créés et de gagner en performance.



Le pipeline graphique de Vulkan, en vert la partie « fixe » du pipeline et en jaune la partie programmable via les shaders

Les shaders sont un autre élément caractéristique de Vulkan.
Ils ne sont plus écrits en GLSL mais en SPIR-V, un langage compilé, ce qui permet :
- De ne plus avoir à compiler les shaders au runtime
- De ne plus avoir de différence d'interprétation du GLSL entre les compilateurs des différents constructeurs de carte graphiques (ce qui pouvait arriver en utilisant certaines particularités du langage)

Pour l'écriture de ces shaders, il existe un compilateur pour compiler les shaders GLSL en SPIR-V, développé par Khronos, ce qui ne complique pas la conception de ces derniers.
Il existe de même une librairie pour compiler le GLSL en SPIR-V directement au runtime.

On peut imaginer qu'à l'avenir il existe à l'avenir d'autres compilateurs pour générer des shaders SPIR-V.

Une autre caractéristique intéressante de SPIR-V est sa compatibilité avec la bibliothèque OpenCL et donc la possibilité d'utiliser ce langage pour le calcul.

Buffers et images

L'affichage d'un objet nécessite la création d'un vertex buffer, et potentiellement d'un index buffer. Il faut donc les créer et y transférer les données.
La création de buffer nécessite davantage de travail qu'avec OpenGL : il faut gérer soit même la mémoire en l'allouant, puis en liant cette mémoire au buffer créé.
En effet avec Vulkan, la gestion de la mémoire est confiée à l'utilisateur et interviendra pour tous les types de buffer que l'on voudra utiliser : vertex buffer, images, uniforms…

Il s'agit d'un enjeu important dans l'utilisation de l'API que je détaillerais davantage dans la deuxième partie de ce document.

La mémoire peut s'allouer depuis différentes « memory heap » aux caractéristiques différentes : certaines mémoires peuvent être écrites et visibles depuis le CPU, d'autre plus rapides uniquement depuis le GPU.
Une technique couramment utilisée est donc l'utilisation de « staging buffer » qui consiste à écrire les données dans des buffers temporaires puis à les copier vers des buffers plus rapides accessibles uniquement au GPU.

Une fois les vertex et index buffer alloués, ils sont bindés au moment du rendu.
Le format du vertex buffer - son nombre de composantes (position, normal, UV, couleur...) et le format de celle ci - est précisé au moment de la création du pipeline graphique.

Les « uniform buffer », qui sont utilisés dans Vulkan pour passer des données aux shaders (matrice world, paramètres de matériaux, ...) utilisent aussi les buffers : il faut leur allouer un buffer et y copier nos données.
Ils remplacent le *glUniform* d'OpenGL et permettent de définir précisément nos besoins mémoires et d'y regrouper nos uniforms dans une même allocation.

La création d'images (*VkImage*) est assez proche de la création des buffers : Il faut créer l'image, allouer la mémoire nécessaire (dont la quantité est calculée avec la largeur, hauteur et nombre de composantes par pixel de l'image), puis lier cette mémoire à notre image.
Pour pouvoir manipuler l'image, on peut ensuite créer une image view (*VkImageView*).

Descriptor

Les descriptors (*VkDescripor*) sont des objets auquels on lie les différentes ressources (buffers, images) qui seront utilisées par les shaders.
Chacune des entrés du descriptor contient des informations sur la ressource (buffer correspondant, taille et type de la ressource...) et le « binding » (id qui permet de faire le lien avec la ressource correspondante dans le shader).
Le descriptor est bindé au moment du rendu ; il permet d'indiquer au shader les ressources utilisables.

Il faut avant de créer le descriptor, créer un layout de notre descriptor (*VkDescriptorSetLayout*) qui est renseigné dans le pipeline graphique.

Il permet d’indiquer quel binding est utilisé à quel phase du pipeline (vertex, geometry, fragment...)

Les descriptors sont alloués à partir d’une pool de descriptors (*VkDescritorPool*) qu’il faut préalablement créer.

RenderPass

La render pass est l’objet qui décrit le déroulement d’une passe de rendu.
Lors du moment du rendu, il faut alors commencer la passe de rendu (*vkCmdBeginRenderPass*) puis une fois le rendu est effectué mettre fin à la passe de rendu (*vkCmdEndRenderPass*)

La passe de rendu décrit plusieurs choses :
- Les « attachements » de la passe de rendu, c’est-à-dire les types d’image va utiliser (image sur laquelle effectuer le rendu, depth buffer, stencil buffer…)
Pour chacun de ces attachements on décrit le format de l’image (RGBA 32 bits, depth 32 bits…) ainsi que les actions à effectuer aux différentes phases du rendu (clear, garder l’image, don’t care...)
- Les « subpass » qui composent la passe de rendu, avec les layouts des différents attachements pour celle-ci.

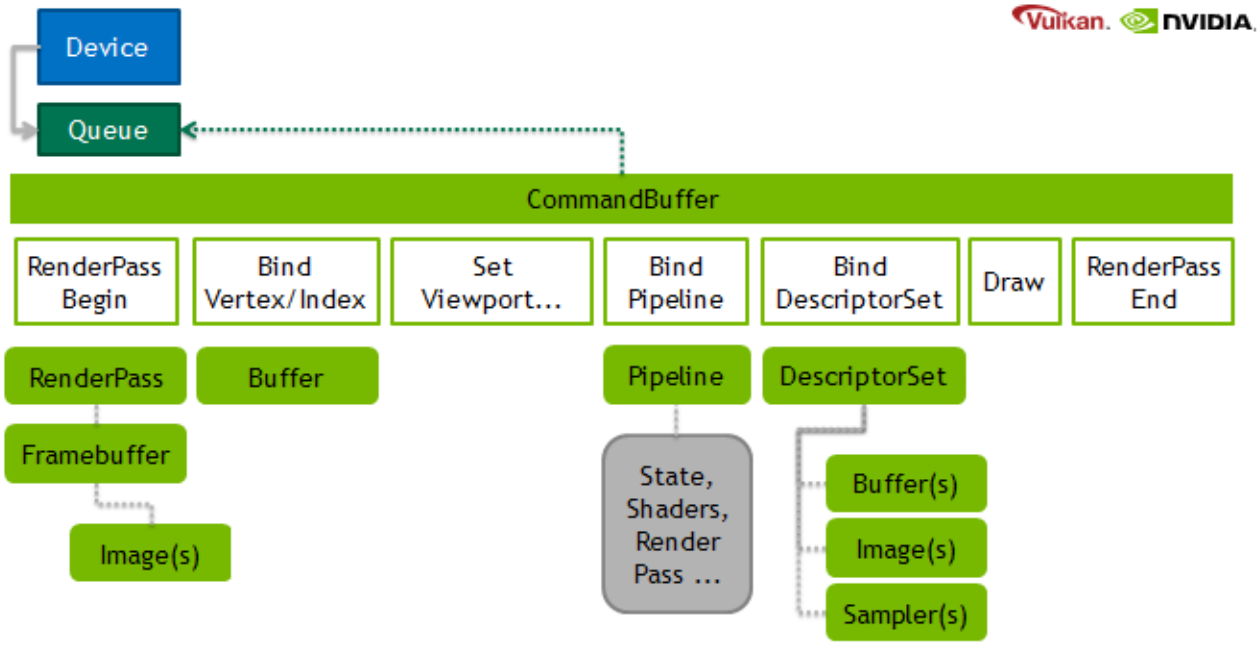
En effet avec Vulkan les images peuvent être agencées selon différent layouts (*VkImageLayout*) pour être utilisées de façon optimale en fonction de l’utilisation qu’on veut en faire.
Il y a des layouts adaptés au transfert de l’image, aux color attachements, au depth attachements…
La transition entre ces layout se fera automatiquement entre les différentes subpass.

Framebuffer

Le framebuffer regroupe l’ensemble des attachements qui vont être utilisés lors d’une render pass.
La render pass décrit en quelque sorte le « layout » de ces attachements, leur format, utilisations… et le framebuffer est un ensemble d’attachements qui seront passés en paramètre à la passe de rendu.
En 3D, on utilise typiquement 2 attachements : le color buffer, sur lequelle est effectué le rendu (que l’on récupérera depuis la swap chain) et le depth buffer.
Il est alors utile de créer autant de framebuffers que d’images dans la swap chain : chaque framebuffer à alors son image de la swap chain associé.
Le depth buffer peut lui être le même quelque soit l’image de la swap chain : son contenu n’a pas besoin d’être sauvegardé à la fin du rendu (contrairement au color buffer qui doit être conservé pour l’affichage).
On peut alors créer le depth buffer nous même (de la même manière qu’on crée une image dans Vulkan) et l’utiliser en paramètre de nos différents framebuffers.

Command buffer

Le principe de command buffer (*VkCommandBuffer*) est encore une spécificité de la dernière génération d'API graphique.
Le command buffer stocke une série de commandes pour la carte graphique, que l'on a au préalable enregistré, puis il est envoyé à une « queue » (file) de command buffers où ceux-ci seront exécutés les un après les autres.



La première étape est de créer une pool de command buffer où l'on indique le type de queue auquel sera envoyé le buffer (graphique, présentation...) ainsi que le nombre de buffers dans la pool.
Cette pool nous permet d'instancier des command buffers.

L'enregistrement des command buffers se fait à l'aide des fonctions *vkBeginCommandBuffer* et *vkEndCommandBuffer.*
Entre ces 2 appels il suffit d'appeler les commandes que l'on souhaite y intégrer pour les enregistrer dans le buffer.

Typiquement, un commande buffer pour afficher un objet simple serait :
- *vkCmdBeginRenderPass* : Début de la passe de rendu
- *vkCmdBindPipeline* : On bind le pipeline graphique
- *vkCmdBindVertexBuffers* : On bind le vertex buffer
- *vkCmdBindIndexBuffer* : On bind l'index buffer
- *vkCmdBindDescriptorSets* : On bind le descriptor (nécessaire pour les autres ressources du shader, textures, matrices...)
- *vkCmdDrawIndexed* : On fait le rendu la scène
- *vkCmdEndRenderPass* : Fin de la passe de rendu

Chacune de ces commandes dispose bien sur de plusieurs paramètres.
On peut ensuite exécuter cette série de commandes avec la fonction *vkQueueSubmit*.

Les command buffers ont l'avantage d'être rapides et réutilisable, ce qui permet de réduire le coût CPU lors de leurs utilisations.

Un command buffer peut aussi contenir des commandes exécutant d'autres command buffers.
Il est ainsi possible de décomposer le rendu en différents « sous-command buffers » pour augmenter leurs réutilisations.
On parle de command buffers primaires et secondaires.

2. Construction d’une application d’exemple

Structure et fonctionnement

Après m’être initié au fonctionnement basique de Vulkan, j’ai voulu développer un petit programme d’exemple avec Vulkan.
J’ai en plus de Vulkan utilisé les bibliothèques GLFW (gestion de la fenêtre), GLM (math), STB (chargement d’images) et Assimp (chargement de modèles 3D).

Le code produit dans le cadre de ce travail est disponible sur bitbucket :
https://bitbucket.org/jlouis/vulkandemo

J’ai construit mon application autour de plusieurs classes principales :

- Le Device, qui gère la fenêtre et crée le VkVideoDriver et le SceneManager.

- Le VkVideoDriver, qui gère toutes les opérations de rendu avec Vulkan.
Il commence par initialiser tous les objets Vulkan, puis il permet de dessiner la scène à chaque frame.
Le principal intérêt de la séparation Device/VkVideoDriver et d’encapsuler le code Vulkan dans VkVideoDriver et de pouvoir ainsi facilement ajouter des drivers pour d’autres API à l’avenir.

- Le SceneManager, qui gère les nodes à afficher dans la scène.

- Le Mesh, qui contient la liste de vertices et d’indices du mesh en question.
Les modèles 3D sont chargés avec la méthode AssimpLoader::loadMeshes qui permet de charger facilement de nombreux formats de fichiers avec la bibliothèque Assimp.
Le mesh contient aussi 2 « BufferHandle » qui est une classe simple encapsulant un buffer (*VkBuffer*) ou une image (*VkImage*) ainsi que la mémoire du buffer associé (*VkDeviceMemory*).
On a donc :
- Le vertex buffer
- L’indices buffer
Ces handles sur les buffers ne sont pas initialisés par le loader Assimp mais par le VkVideoDriver quand celui-ci alloue les ressources correspondantes.
Le mesh à aussi un booléen « dirty » qui permet d’indiquer à VkVideoDriver si les données du mesh doivent être (re-)copier dans les buffers (par exemple lorsqu’on change la position d’un vertex).
Les buffers sont copiés en mémoire « device local » depuis des staging buffers.

- Le SceneNode, qui représente un objet de la scène.
Le node à une position, une rotation ainsi qu’un scale et peut avoir un Mesh* associé à afficher
Le node a aussi un BufferHandle correspondant au buffer uniform contenant sa matrice « World ».
Ce buffer est donc mis à jour à chaque frame en fonction de la position, de la rotation et du scale du node.
Le buffer est « device local » pour de bonne performance, et un « staging buffer » unique et commun à tous les nodes est créé dans le VkVideoDriver lors de son initialisation pour les opérations de mise à jour et de copie du buffer.

- La Texture, qui permet de charger et stocker des images avec la bibliothèque STB.
La texture stocke les données de l’image, mais possède aussi un BufferHandle ainsi qu’une ImageView correspondants aux données de l’image dans Vulkan.

Ces handles sont nuls au chargement de la textures, puis sont initialisés par le VkVideoDriver une fois que celui-ci à allouer les ressources Vulkan nécessaire avec une méthode createTextureImageView (Texture* tex)

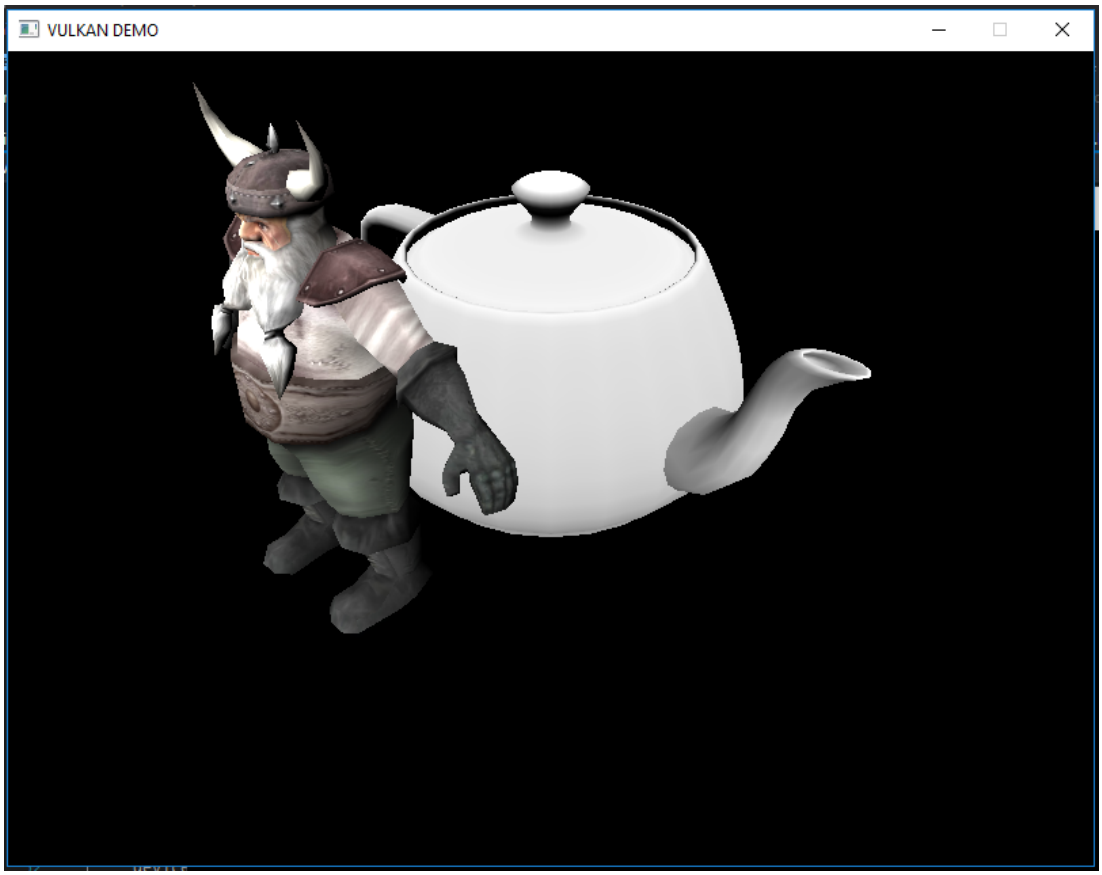
La phase d'initialisation crée les objets Vulkan nécessaires au rendu présenté dans la première partie de ce document : instance, device, swap chain et sémaphores, render pass, framebuffers et pipeline graphique.
Dans ce programme tous les objets utiliseront le même pipeline graphique, c'est-à-dire qu'ils utiliseront les même shaders.
Les shaders utilisés sont des shaders assez classiques affichant le modèle texturé avec une diffuse map et incluant un éclairage ambiant + point light.

Dans une application réelle utilisant plusieurs shaders différents, il faudrait utiliser au moins un pipeline par shader utilisé.

La main loop se décompose en différentes opérations :
- Acquisition d'une image de la swap chain

- Calcul des matrices « View » et « Projection » et mise à jour d'un uniform buffer contenant ces 2 informations.
En effet, ces 2 informations étant commune à tout les nodes, elles sont copiés dans un unique buffer à part qui est utilisé par tous les nodes pour économiser de la mémoire.
- Mise à jour des buffers des mesh de la scène si necessaire
- Mise à jour des matrices world des nodes dans les buffer Vulkan correspondants
- Enregistrement d'un commande buffer, début de passe de rendu, binding du pipeline graphique
- Pour chaque node visible :
→ Effectuer les opérations de rendus nécessaires du mesh : binding du vertex buffer, de l'index buffer et du descriptor du node
- Fin de la passe de rendu
- Arrêter l'enregistrement du command buffer et l'envoyer à la file graphique. Le command buffer est exécuté seulement une fois l'image de swap chain récupérée (géré avec un sémaphore)

- Envoi de l'image rendue à la file de présentation pour affichage (synchronisé avec un sémaphore pour s'assurer que le rendu est bien fini avant ça)



Optimisations possibles

J’ai créé mon application le plus simplement possible par manque de temps et en étant peu confronté aux problèmes de performances sur ma simple scène de démonstration.
Cependant plusieurs améliorations sont envisageables et nécessaires pour exploiter au mieux Vulkan et permettre de meilleur performances pour des applications de plus grande ampleur.

Gestion des command buffers

Le programme utilise actuellement un unique command buffer qui est ré-enregistré à chaque frame.
Le ré-enregistrement à chaque frame permet d’ajouter de nouveaux objets à la scène après l’initialisation du programme, mais cela est coûteux : l’utilisation de command buffer en soit permet déjà de gagner en performance, plus rapide que des appels successif de fonction en OpenGL, mais les réutiliser peut encore davantage améliorer les performances.

On peut imaginer à place de décomposer le rendu en plusieurs command buffers, par exemple un par node.
Il suffirait alors de construire un nouveau command buffer à la création d’un node, et les commandes buffers des autres nodes pourrait eux être réutilisé.
Les command buffers des nodes pourrait alors être exécuté à la suite en étant rarement

reconstruit.
Les commandes nécessaires pour commencer et finir la scène (render pass, binding du pipeline…) seraient dans 2 command buffers à part exécutés au début et à la fin du rendu.

On pourrait aussi envisager de regrouper les appels à ces buffers dans un buffer principale qu'il faudrait reconstruire seulement à l'ajout ou la suppression d'un node.

Gestion de la mémoire

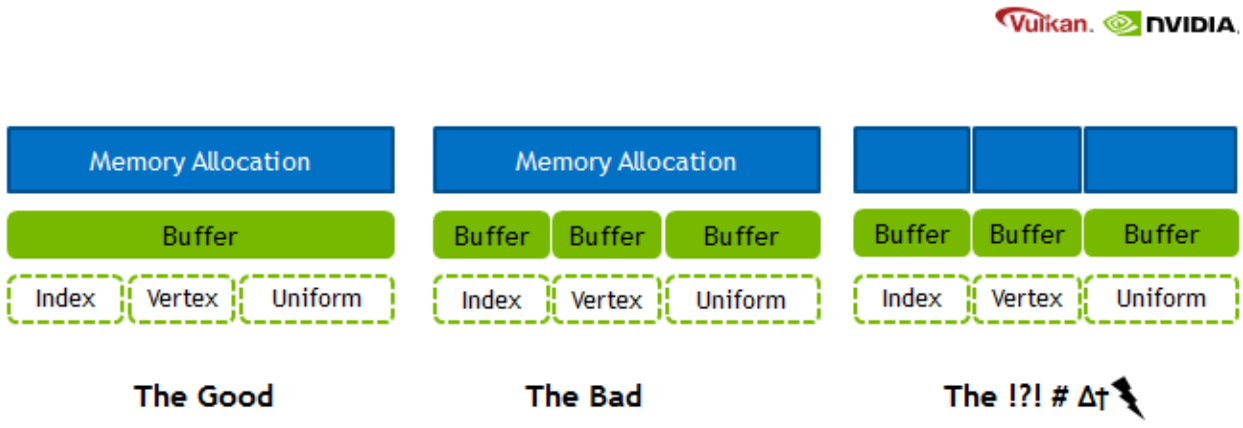
Avec Vulkan, la gestion de la mémoire étant laissée à l'utilisateur, celle-ci devient un sujet important et bien faite elle peut améliorer les performances des programmes.

Une première recommandation est l'utilisation de grand buffers regroupant différentes ressources ensemble.

Actuellement l'affichage d'un mesh nécessite l'allocation de 3 zones mémoire et de 3 buffers différents : le vertex buffer, l'indices buffer et l'uniform buffer.
En plus de ça le node utilise éventuellement une texture, ce qui représente une allocation supplémentaire.
Or, faire des allocations mémoire est coûteux, en particulier pour de petits buffers comme le buffer uniform qui contient peu d'informations.

Il est donc conseillé de regrouper les différents objets ensembles pour de meilleur performance.
Vulkan permet ensuite d'utiliser des offsets pour ainsi pouvoir utiliser des « sous-buffer ».
Cela réduit le nombre d'allocations et permet des optimisations de cache pour les ressources utilisant un même buffer.

Le nombre d'allocations mémoire possible est d'ailleurs limité (2048 sur les cartes haut de gamme), donc l'utilisation de cette technique sur des scènes comportant de nombreux objets devient indispensable.



Dans le cas de mon programme, je pourrais par exemple envisager de regrouper les ressources par mesh (vertex + index + uniform), ou encore regrouper tous les uniforms ensemble dans un même buffer pour éviter l'utilisation de nombreux petits buffers.

L’utilisation de staging buffer pour copier les données vers des zones mémoires plus performantes fait aussi partie des bonnes pratiques. C’est actuellement la technique utilisée dans mon programme de démonstration.

Il est aussi recommandé de réutiliser des zones mémoires déjà allouées plutôt que d’en désallouer et d’en ré-allouer de nouvelles.
Par exemple pour les opérations de copie de buffers (copie d’un staging buffer vers un buffer plus performant), j’alloue et désalloue actuellement les staging buffers à chaque fois, là où l’utilisation d’un unique et large staging buffer pourrait suffire pour la plupart des opérations de copie.

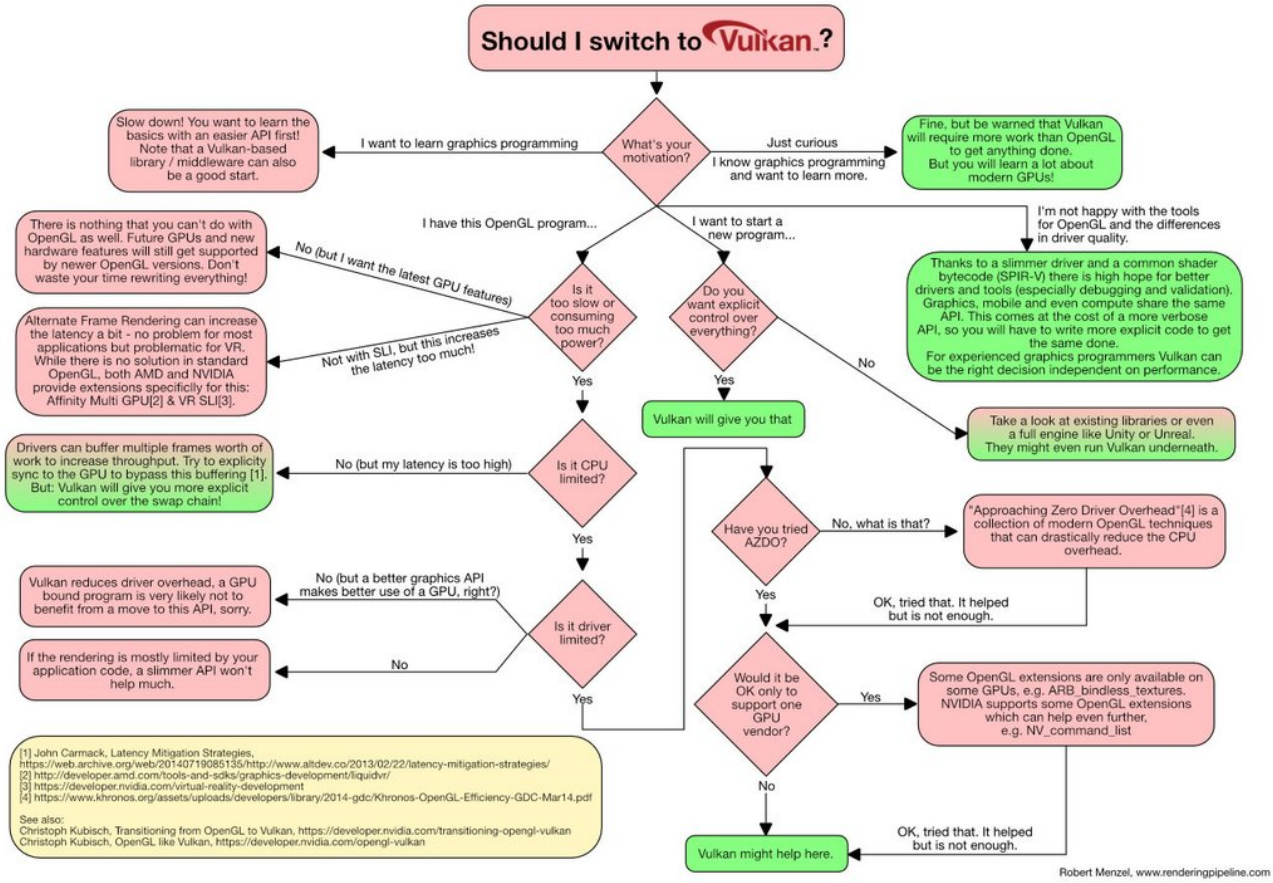
Utilisation du « push constant »

La commande « *vkCmdPushConstants* » permet d’envoyer des données au shader via un buffer uniform rapide de 256 octets au maximum.
Cela pourrait être utilisé pour transmettre les données des uniform buffers aux shaders et ainsi réduire le nombre d’allocations mémoires nécessaires dans le programme.

Je vais pour la suite de mon travail essayer d’implémenter ses différentes optimisations et évaluer les gains de performances apportés.

3. Conclusion

Pour conclure, en modifiant radicalement le fonctionnement des API de rendus, Vulkan permet une programmation plus bas niveau et d'important gain en terme d'utilisation du CPU.
Pour autant Vulkan est-il amené à remplacer OpenGL pour toutes les nouvelles applications à venir ?

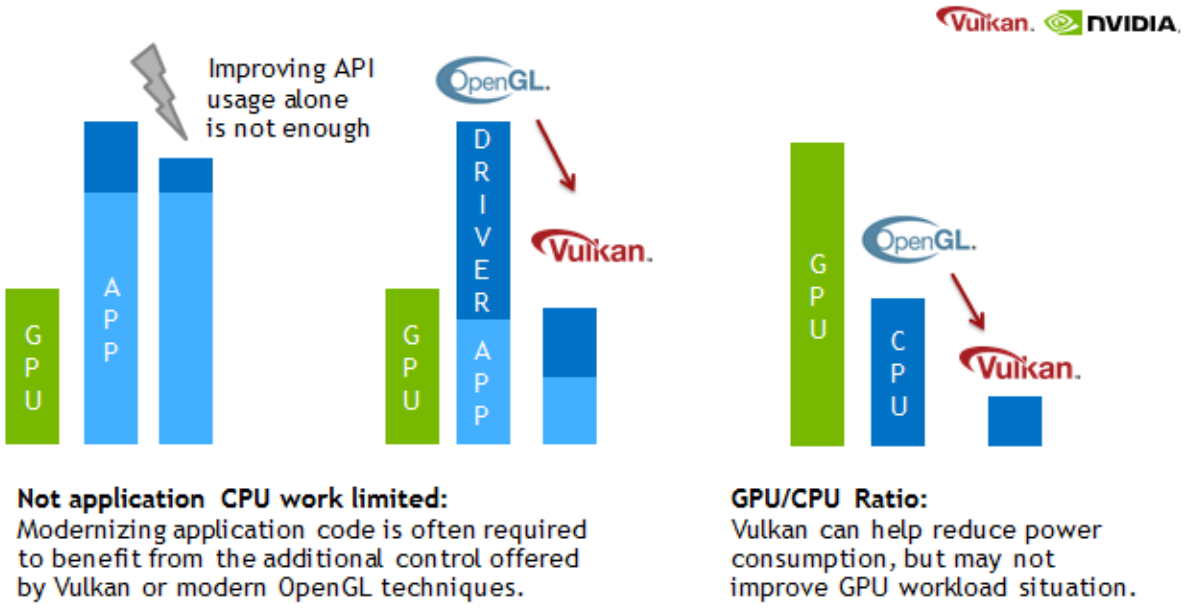


Vulkan permet des gains de performances, mais à un coût : La création de programme avec Vulkan est complexe et demande plus de travail, pour un gain de performance qui ne sera pas toujours significatif en fonction des cas.
La question se pose en particulier pour les applications existantes, ou cela implique de repenser le fonctionnement de l'API dans l'application pour être efficace.
Adapter tel qu'elle une application OpenGL en remplaçant les appels à l'API par des appels à Vulkan est possible, mais cela n'apportera pas ou que peu de gain de performances.

Il est aussi intéressant de noter qu'OpenGL 4.5 et en particulier ses extensions permettent d'obtenir sur certains points une conception proche de celle de Vulkan (On parle d'AZDO : Approaching Zero Driver Overhead) :
- L'extension *ARB_direct_state_access* d'OpenGL permet le « Direct State Access » (DSA) qui se rapproche de l'utilisation d'handles avec Vulkan en évitant de devoir binder les objets pour les modifier.
- L'extension *ARB_multi_draw_indirect* permet de dessiner un objet en utilisant une sous-partie d'un buffer. On peut donc comme les bonnes pratiques le suggère allouer de grand buffers constitués de plusieurs sous-buffer.
- Les extensions *ARB_buffer_storage* (buffers immuables), *ARB_texture_storage* (textures immuables) et *ARB_texture_view* (stockage de plusieurs textures dans différent layers d'une texture) permettent de faire des allocations mémoires plus efficaces.

- L’extension *NV_command_list* de Nvidia permet d’utiliser un système proche des command buffers de Vulkan avec OpenGL
- …

Vulkan permet des économies CPU avec un driver plus proche de la carte graphique, mais pas forcément des gains GPU : il est donc important d’identifier si une application peut être « CPU limited » ou non pour évaluer les gains de performances.



Il reste cependant après ça les autres avantages de Vulkan : une API multi-plateforme, multithreading friendly, adaptée au calcul, des shaders compilés…

En conclusion, Vulkan comporte de nombreux avantages, mais l’API est en contrepartie beaucoup plus complexe à utiliser qu’OpenGL.
OpenGL reste une alternative intéressante qui permet d’accéder au même fonctionnalités des cartes graphiques que Vulkan.
La principale différence résidera dans le choix d’une API plus simple d’accès et permettant de développer des applications plus rapidement, ou une API davantage « low overhead » permettant un coût CPU minimum au prix de davantage de travail.

Enfin, d’un point de vue personnel, ce travail a été très intéressant et m’a permis d’avoir une compréhension plus fine du fonctionnement des cartes graphiques et des techniques d’optimisations des nouvelles API.
Vulkan est un sujet vaste et complexe, dont j’ai pu avoir un petit aperçu pendant grâce à ce travail réalisé sur une courte période.
Il me reste cependant de nombreux points à explorer, tel que les subpass, le post processing, l’instancing..., mais cette première découverte de l’API a été très motivante et je vais continuer à travailler sur le sujet.

Références

https://vulkan-tutorial.com/
http://vulkan.developpez.com/
https://developer.nvidia.com/engaging-voyage-vulkan
https://developer.nvidia.com/vulkan-memory-management
https://developer.nvidia.com/vulkan-shader-resource-binding
https://software.intel.com/en-us/articles/api-without-secrets-introduction-to-vulkan-part-1
https://www.khronos.org/registry/vulkan/specs/1.0/man/html/
https://renderdoc.org/vulkan-in-30-minutes.html
https://developer.nvidia.com/transitioning-opengl-vulkan