

Tutoriel Vulkan complet

Par Alexander Overvoorde  - Alexis Ronez (traducteur) 

Date de publication : 8 juin 2020

Dernière mise à jour : 10 octobre 2020

Vulkan est une bibliothèque bas niveau pour le rendu 3D et le calcul sur GPU. Ce tutoriel a pour objectif de vous aider à prendre en main la bibliothèque et de créer votre premier rendu avec celle-ci.

Commentez

I - Introduction.....	6
I-A - À propos.....	6
I-B - Structure du tutoriel.....	7
I-C - Communauté.....	7
I-C-1 - Contributeurs.....	8
II - Aperçu de Vulkan.....	8
II-A - Origine de Vulkan.....	8
II-B - Le nécessaire pour afficher un triangle.....	8
II-B-1 - Étape 1 - Instance et sélection d'un périphérique physique.....	9
II-B-2 - Étape 2 – Périphérique logique et familles de queues.....	9
II-B-3 - Étape 3 – Surface d'affichage et swap chain.....	9
II-B-4 - Étape 4 – Vue d'image et tampons d'image.....	9
II-B-5 - Étape 5 – Passes de rendu.....	10
II-B-6 - Étape 6 - Le pipeline graphique.....	10
II-B-7 - Étape 7 – Groupe de commandes et tampons de commandes.....	10
II-B-8 - Étape 8 - Boucle principale.....	10
II-B-9 - Résumé.....	11
II-C - Concepts de la bibliothèque.....	11
II-C-1 - Conventions.....	11
II-C-2 - Couches de validation.....	12
III - Environnement de développement.....	12
III-A - Windows.....	12
III-A-1 - SDK Vulkan.....	12
III-A-2 - GLFW.....	14
III-A-3 - GLM.....	14
III-A-4 - Préparer Visual Studio.....	15
III-B - Linux.....	20
III-B-1 - Paquets Vulkan.....	20
III-B-2 - GLFW.....	21
III-B-2-a - GLM.....	21
III-B-3 - Compilateur de shader.....	22
III-B-4 - Préparation d'un projet avec Makefile.....	22
III-C - MacOS.....	24
III-C-1 - SDK Vulkan.....	24
III-C-2 - GLFW.....	25
III-C-3 - GLM.....	25
III-C-4 - Préparation de Xcode.....	25
IV - Dessiner un triangle.....	29
IV-A - Mise en place.....	29
IV-A-1 - Code de base.....	29
IV-A-1-a - Structure générale.....	29
IV-A-1-b - Gestion des ressources.....	30
IV-A-1-c - Intégrer GLFW.....	30
IV-A-2 - Instance.....	32
IV-A-2-a - Création d'une instance Vulkan.....	32
IV-A-2-b - Vérification du support des extensions.....	33
IV-A-2-c - Libération des ressources.....	34
IV-A-3 - Couches de validation.....	34
IV-A-3-a - Qu'est-ce que les couches de validation ?.....	34
IV-A-3-b - Utiliser les couches de validation.....	35
IV-A-3-c - Gestion personnalisée des messages.....	37
IV-A-3-d - Déboguer la création et la destruction de l'instance.....	40
IV-A-3-e - Test.....	41
IV-A-3-f - Configuration.....	41
IV-A-4 - Périphériques physiques et famille de queues.....	42
IV-A-4-a - Sélection d'un périphérique physique.....	42
IV-A-4-b - Vérification des fonctionnalités de base.....	43
IV-A-4-c - Familles de queues (queue families).....	44

IV-A-5 - Périphérique logique et queues.....	47
IV-A-5-a - Introduction.....	47
IV-A-5-b - Spécifier les queues à créer.....	47
IV-A-5-c - Spécifier les fonctionnalités utilisées.....	48
IV-A-5-d - Créer le périphérique logique.....	48
IV-A-5-e - Récupérer des références aux queues.....	49
IV-B - Envoi du rendu à l'écran.....	49
IV-B-1 - Surface de fenêtre.....	49
IV-B-1-a - Création de la surface.....	50
IV-B-1-b - Vérification du support d'envoi du rendu à la fenêtre.....	51
IV-B-1-c - Création de la queue de présentation.....	52
IV-B-2 - Swap chain.....	53
IV-B-2-a - Vérification du support des swap chain.....	53
IV-B-2-b - Activation des extensions du périphérique.....	54
IV-B-2-c - Récupération des détails à propos du support de la « swap chain ».....	54
IV-B-2-d - Choix des bons paramètres pour la « swap chain ».....	55
IV-B-2-d-i - Format de la surface.....	56
IV-B-2-d-ii - Mode de présentation.....	57
IV-B-2-d-iii - La zone d'échange.....	57
IV-B-2-e - Création de la swap chain.....	58
IV-B-2-f - Récupération des images de la « swap chain ».....	61
IV-B-3 - Vues d'image.....	61
IV-C - Bases du pipeline graphique.....	63
IV-C-1 - Introduction.....	63
IV-C-2 - Module de shaders.....	66
IV-C-2-a - Le vertex shader.....	66
IV-C-2-b - Le fragment shader.....	68
IV-C-2-c - Une couleur pour chaque sommet.....	68
IV-C-2-d - Compilation des shaders.....	69
IV-C-2-d-i - Windows.....	70
IV-C-2-d-ii - Linux.....	70
IV-C-2-d-iii - Notes.....	70
IV-C-2-e - Charger un shader.....	71
IV-C-2-f - Créer des modules de shader.....	71
IV-C-2-g - Création des étapes programmables.....	72
IV-C-3 - Étapes fixes.....	73
IV-C-3-a - Les sommets en entrée.....	73
IV-C-3-b - Assembleur d'entrée.....	74
IV-C-3-c - Viewports et découpage.....	74
IV-C-3-d - Rastériseur.....	75
IV-C-3-e - Multiéchantillonnage.....	76
IV-C-3-f - Tests de profondeur et de pochoir.....	77
IV-C-3-g - Mélange de couleurs.....	77
IV-C-3-h - États dynamiques.....	78
IV-C-3-i - Agencement du pipeline.....	78
IV-C-3-j - Conclusion.....	79
IV-C-4 - Passes de rendu.....	79
IV-C-4-a - Mise en place.....	79
IV-C-4-b - Description de l'attache.....	80
IV-C-4-c - Sous-passes et références aux attaches.....	81
IV-C-4-d - Passe de rendu.....	82
IV-C-5 - Conclusion.....	82
IV-D - Rendu.....	84
IV-D-1 - Tampons d'images.....	84
IV-D-2 - Tampons de commandes.....	86
IV-D-2-a - Groupe de commandes.....	86
IV-D-2-b - Allocation des tampons de commandes.....	87
IV-D-2-c - Commencer l'enregistrement des commandes.....	88

IV-D-2-d - Commencer une passe de rendu.....	88
IV-D-2-e - Commandes de rendu basiques.....	89
IV-D-2-f - Finalisation.....	90
IV-D-3 - Rendu et présentation.....	90
IV-D-3-a - Mise en place.....	90
IV-D-3-b - Synchronisation.....	90
IV-D-3-c - Sémaffores.....	91
IV-D-3-d - Obtention d'une image provenant de la « swap chain ».....	92
IV-D-3-e - Envoi du tampon de commandes.....	92
IV-D-3-f - Dépendances des sous-passes.....	93
IV-D-3-g - Affichage.....	94
IV-D-3-h - Rendu en cours.....	96
IV-D-3-i - Conclusion.....	100
IV-E - Recréation de la « swap chain ».....	100
IV-E-1 - Introduction.....	100
IV-E-2 - Recréer la « swap chain ».....	100
IV-E-3 - Swap chain non optimale ou obsolète.....	102
IV-E-4 - Gestion explicite des redimensionnements.....	103
IV-E-5 - Gestion de la minimisation de la fenêtre.....	104
V - Tampons de sommets.....	105
V-A - Description des sommets en entrée.....	105
V-A-1 - Introduction.....	105
V-A-2 - Vertex shader.....	105
V-A-3 - Sommets.....	105
V-A-4 - Descriptions des liens.....	106
V-A-5 - Description des attributs.....	106
V-A-6 - Entrée des sommets dans le pipeline.....	108
V-B - Création du tampon de sommets.....	108
V-B-1 - Introduction.....	108
V-B-2 - Création du tampon.....	108
V-B-3 - Exigences concernant la mémoire.....	109
V-B-4 - Allocation de mémoire.....	111
V-B-5 - Remplissage du tampon de sommets.....	111
V-B-6 - Lier le tampon de sommets.....	112
V-C - Tampon intermédiaire.....	114
V-C-1 - Introduction.....	114
V-C-2 - Queue de transfert.....	114
V-C-3 - Abstraction de la création des tampons.....	115
V-C-4 - Utiliser un tampon intermédiaire.....	116
V-C-5 - Conclusion.....	118
V-D - Tampon d'indices.....	118
V-D-1 - Introduction.....	118
V-D-2 - Création d'un tampon d'indices.....	119
V-D-3 - Utilisation d'un tampon d'indices.....	120
VI - Tampons de variables uniformes.....	122
VI-A - Descripteur d'agencement et de tampon.....	122
VI-A-1 - Introduction.....	122
VI-A-2 - Vertex shader.....	123
VI-A-3 - Agencement de l'ensemble de descripteurs.....	123
VI-A-4 - Tampon de variables uniformes.....	125
VI-A-5 - Mise à jour des variables uniformes.....	126
VI-B - Groupe de descripteurs et ensembles.....	128
VI-B-1 - Introduction.....	128
VI-B-2 - Groupe de descripteurs.....	128
VI-B-3 - Ensemble de descripteurs.....	129
VI-B-4 - Utiliser des ensembles de descripteurs.....	131
VI-B-5 - Alignement.....	132
VI-B-6 - Plusieurs ensembles de descripteurs.....	134

VII - Application des textures.....	134
VII-A - Images.....	134
VII-A-1 - Introduction.....	134
VII-A-2 - Bibliothèque de chargement d'image.....	135
VII-A-2-a - Visual Studio.....	135
VII-A-2-b - Makefile.....	136
VII-A-3 - Chargement d'une image.....	136
VII-A-4 - Tampon intermédiaire.....	137
VII-A-5 - Texture d'image.....	138
VII-A-6 - Transitions de l'agencement.....	141
VII-A-7 - Copie d'un tampon dans une image.....	143
VII-A-8 - Préparer la texture.....	144
VII-A-9 - Masque de barrière de transition.....	144
VII-A-10 - Nettoyage.....	146
VII-B - Vue d'image et échantillonneur.....	146
VII-B-1 - Vue sur une texture.....	146
VII-B-2 - Échantillonneurs.....	148
VII-B-3 - Fonctionnalités de filtrage anisotrope du périphérique.....	151
VII-C - Association d'échantillonneur et d'image.....	152
VII-C-1 - Introduction.....	152
VII-C-2 - Modifier les descripteurs.....	152
VII-C-3 - Coordonnées de texture.....	154
VII-C-4 - Shaders.....	155
VIII - Tampon de profondeurs.....	159
VIII-A - Introduction.....	159
VIII-B - Géométrie en 3D.....	159
VIII-C - Image de profondeur et vue.....	162
VIII-C-1 - Transition explicite de l'image de profondeur.....	164
VIII-D - Passe de rendu.....	165
VIII-E - Tampon d'images.....	166
VIII-F - Valeurs de nettoyage des tampons.....	167
VIII-G - Configuration des tests profondeur et de pochoir.....	167
VIII-H - Gestion des redimensionnements de la fenêtre.....	168
IX - Chargement de modèles.....	169
IX-A - Introduction.....	169
IX-B - Une bibliothèque.....	169
IX-B-1 - Visual Studio.....	169
IX-B-2 - Makefile.....	170
IX-C - Modèle d'exemple.....	170
IX-D - Chargement des sommets et des indices.....	171
IX-E - Déduplication des sommets.....	173
X - Génération de mipmaps.....	175
X-A - Introduction.....	175
X-B - Création des images.....	175
X-C - Génération des mipmaps.....	177
X-D - Support du filtrage linéaire.....	180
X-E - Échantillonneur.....	181
XI - Multiéchantillonnage.....	183
XI-A - Introduction.....	183
XI-B - Récupération du nombre maximal d'échantillons.....	185
XI-C - Configurer une cible de rendu.....	186
XI-D - Ajout de nouvelles attaches.....	187
XI-E - Amélioration de la qualité.....	190
XI-F - Conclusion.....	191
XII - Résolution des problèmes.....	191

I - Introduction

I-A - À propos

Ce tutoriel vous enseignera les bases de l'utilisation de la bibliothèque **Vulkan** pour afficher des graphismes et pour réaliser des calculs sur GPU. Vulkan est une nouvelle bibliothèque créée par le **groupe Khronos** (notamment connu pour OpenGL). Cette bibliothèque offre une bien meilleure abstraction des cartes graphiques modernes. En effet, la nouvelle interface vous permet de mieux décrire ce que votre application souhaite faire. Cela peut permettre d'obtenir de meilleures performances ainsi qu'un comportement moins surprenant des pilotes graphiques comme cela pouvait être le cas avec **OpenGL** et **Direct3D**. Les concepts introduits par Vulkan sont similaires à ceux de **Direct3D 12** et **Metal**. Cependant Vulkan a l'avantage d'être complètement multiplateforme et permet donc de développer pour Windows, Linux, Mac et Android en même temps.

Il y a cependant un contrecoup à ces avantages : la bibliothèque est plus verbeuse. Tout ce qui concerne le rendu graphique doit être intégralement défini par votre application : cela inclut la création du tampon d'image initial, ou encore, la gestion de la mémoire pour les tampons ou les textures. Le pilote graphique ne vous tient plus par la main. Cela signifie qu'il y a plus de choses à faire dans l'application pour obtenir un comportement correct.

Le message véhiculé ici est que Vulkan n'est pas fait pour tout le monde. Cette bibliothèque est conçue pour les programmeurs intéressés par la programmation GPU haute performance et qui sont prêts à y travailler sérieusement. Si vous vous intéressez au développement de jeux vidéo et non particulièrement aux rendus graphiques, vous devriez plutôt continuer d'utiliser OpenGL et DirectX, qui ne seront pas prochainement dépréciés en faveur de Vulkan. Une autre option serait d'utiliser un moteur de jeu comme **Unreal Engine** ou **Unity**, qui sont capables d'utiliser Vulkan tout en exposant une bibliothèque de plus haut niveau.

Cela étant dit, présentons quelques prérequis pour ce tutoriel :

- une carte graphique et un pilote compatibles avec Vulkan (**NVIDIA, AMD, Intel**) ;
- une expérience avec le C++ (connaissance du RAII (Resource Acquisition Is Initialization), listes d'initialisation) ;
- un compilateur supportant les fonctionnalités du C++17 (Visual Studio 2017 ou supérieur, GCC 7 ou supérieur, Clang 5 ou supérieur) ;
- un minimum d'expérience dans le domaine de la programmation graphique 3D.

Ce tutoriel ne considérera pas comme acquis les concepts d'OpenGL et de Direct3D, mais il requiert que vous connaissiez les bases du rendu 3D. Il n'expliquera pas non plus les mathématiques derrière, par exemple, la projection de perspective. Lisez **ce livre** pour une bonne introduction des concepts de rendu 3D. Voici aussi d'autres ressources pour le développement d'applications graphiques :

- **le lancer de rayon en un week-end** ;
- **livre sur le rendu basé sur la physique** ;
- une utilisation de Vulkan dans les moteurs graphiques open source **Quake** et de **DOOM 3**.

Si vous le souhaitez, vous pouvez utiliser le C à la place du C++. Toutefois vous devrez utiliser une autre bibliothèque d'algèbre linéaire et vous devrez restructurer vous-même le code. Nous utiliserons des fonctionnalités du C++ (RAII, classes) pour organiser la logique et gérer la durée de vie des ressources. Il existe aussi une **autre version** de ce tutoriel pour les développeurs Rust.

Pour faciliter la tâche des développeurs utilisant d'autres langages de programmation, et pour acquérir de l'expérience avec la bibliothèque de base, nous allons utiliser l'API C originelle pour manipuler Vulkan. Cependant, si vous utilisez le C++, vous pourrez préférer utiliser **Vulkan-Hpp**, qui permet de s'éloigner de certains détails ennuyeux et d'éviter certains types d'erreurs.

I-B - Structure du tutoriel

Nous allons commencer par un aperçu du fonctionnement de Vulkan et le travail nécessaire pour obtenir notre premier triangle à l'écran. Le but de chaque petite étape et d'offrir une meilleure compréhension de leur rôle dans l'ensemble. Ensuite, nous préparerons l'environnement de développement, avec le **SDK Vulkan**, la **bibliothèque GLM** pour les opérations d'algèbre linéaire et **GLFW** pour la création d'une fenêtre. Ce tutoriel couvrira leur mise en place sur Windows avec Visual Studio et sur Linux Ubuntu avec GCC.

Après cela, nous implémenterons tous les éléments nécessaires à un programme Vulkan pour afficher le premier triangle. Chaque chapitre suivra approximativement la structure suivante :

- introduction d'un nouveau concept et de son utilité ;
- utilisation de tous les appels correspondants à la bibliothèque pour leur mise en place dans votre programme ;
- placement d'une partie de ces appels dans des fonctions pour une réutilisation future.

Bien que chaque chapitre soit écrit comme suite du précédent, il est également possible de lire chacun d'entre eux comme un article introduisant une fonctionnalité précise de Vulkan. Ainsi ce tutoriel peut être utilisé comme référence. Toutes les fonctions et les types provenant de Vulkan sont liés à leur documentation : vous pouvez donc cliquer dessus pour en apprendre plus. Vulkan est une bibliothèque récente, il peut donc y avoir des lacunes dans la spécification elle-même. Vous êtes encouragé à transmettre vos retours dans [ce dépôt de Khronos](#).

Comme indiqué plus haut, Vulkan est une bibliothèque assez prolixe, avec de nombreux paramètres, pensés pour vous fournir un maximum de contrôle sur le matériel graphique. Ainsi des opérations simples comme créer une texture se font en de nombreuses étapes devant être répétées à chaque fois. Nous créerons notre propre collection de fonctions d'aide tout au long du tutoriel.

Chaque chapitre se conclura avec un lien menant à la totalité du code écrit jusqu'à ce point. Vous pourrez vous y référer si vous avez un quelconque doute quant à la structure du code, ou si vous rencontrez un bogue et que vous voulez comparer. Tous les codes ont été testés sur les cartes graphiques des différents constructeurs et fonctionnent. Si vous avez un quelconque souci, vous pouvez utiliser [le forum](#). Veuillez y indiquer votre plateforme, la version de votre pilote, votre code source, le comportement attendu et celui obtenu pour nous aider à vous aider.

Après avoir accompli le rituel de l'affichage de votre premier triangle avec Vulkan, nous étendrons le programme pour y inclure les transformations linéaires, les textures et les modèles 3D.

Si vous avez déjà utilisé une bibliothèque graphique auparavant, vous devez savoir qu'il peut y avoir de nombreuses étapes avant l'affichage de la première géométrie à l'écran. Il y en a encore plus avec Vulkan, mais vous verrez que chacune d'entre elles est simple à comprendre et n'est pas redondante. Gardez aussi à l'esprit qu'une fois que vous savez afficher ce triangle – certes ennuyant – l'affichage d'un modèle 3D texturé ne nécessite pas beaucoup plus de travail. À partir de ce moment, chaque étape est plus satisfaisante.

Prêt à vous lancer dans le futur des bibliothèques graphiques de haute performance ? [Allons-y!](#)

I-C - Communauté

Ce tutoriel est destiné à être un effort de communauté. Vulkan est encore une bibliothèque très récente et les meilleures manières d'arriver à un résultat n'ont pas encore été déterminées. Si vous avez un quelconque retour sur le tutoriel et le site lui-même, n'hésitez alors pas à [en discuter ici](#). N'hésitez pas à [watch le dépôt officiel](#) afin d'être notifié des dernières mises à jour du site.

Si vous rencontrez un problème en suivant ce tutoriel, vérifiez d'abord dans la section des [problèmes courants](#) ne contient pas déjà une solution. Si vous êtes toujours coincé après cela, demandez de l'aide sur [le forum](#).

I-C-1 - Contributeurs

Le tutoriel original, écrit en anglais, a été réalisé par Alexander Overvoorde. La traduction française a été principalement réalisée par Alexis Ronez. La liste complète des contributeurs peut être trouvée sur [GitHub](#). La version hébergée sur Developpez.com a été revue par Alexandre Laurent ([LittleWhite](#)). De plus, cette version a reçu une relecture orthographique effectuée par [ClaudeLELOUP](#), [escartefiguer](#) et [-FloT-](#).

Un grand merci à Alexander Overvoorde et Alexis Ronez.

Le tutoriel est sous licence **CC BY-SA 4.0** et le code sous licence **CC0 1.0 Universal**.

II - Aperçu de Vulkan

Ce chapitre commencera par introduire Vulkan et la raison de sa création. Nous nous intéresserons ensuite aux éléments requis pour afficher un premier triangle. Cela vous donnera une vue d'ensemble pour mieux replacer les futurs chapitres dans leur contexte. Nous conclurons sur la structure de Vulkan et la manière dont la bibliothèque est communément utilisée.

II-A - Origine de Vulkan

Comme pour les bibliothèques précédentes, Vulkan est conçue comme une abstraction, multiplateforme, des **GPU**. Le problème avec la plupart de ces bibliothèques est qu'elles furent créées à une époque où le matériel graphique était limité à des fonctionnalités prédéfinies configurables. Les développeurs devaient fournir les données des sommets dans un format standardisé et étaient ainsi à la merci des constructeurs pour les options d'éclairage et d'ombrage.

Au fil des évolutions des cartes graphiques, elles offrent de plus en plus de fonctionnalités programmables. Ces fonctionnalités devaient être intégrées tant bien que mal aux bibliothèques existantes. Ceci résultait en une abstraction peu pratique où le pilote devait deviner l'intention du développeur et faire correspondre le programme aux architectures modernes. C'est pour cela qu'il existe des mises à jour de pilotes visant à améliorer les performances dans les jeux et même quelquefois, de manière significative. À cause de la complexité de ces pilotes, les développeurs doivent gérer les différences de comportement entre les fabricants dont notamment la syntaxe acceptée dans les **shaders**. En plus des nouvelles fonctionnalités, cette dernière décennie a vu un nombre grandissant d'appareils mobiles ayant de puissantes puces graphiques. Ces GPU mobiles ont des architectures différentes, pensées pour fonctionner avec des fortes contraintes d'énergie et d'espace. On peut citer le **rendu en tuiles**, qui offrirait de meilleures performances si les développeurs avaient un meilleur contrôle sur la fonctionnalité. Une autre limitation provenant de l'âge de ces bibliothèques est le manque de support du multithread créant un goulet d'étranglement sur le CPU.

Vulkan résout ces problèmes en ayant été conçu pour les architectures modernes. Elle réduit le travail du pilote en permettant au développeur d'expliquer ses objectifs grâce à une bibliothèque plus prolixe. Aussi elle permet à plusieurs threads de créer et d'envoyer des commandes en parallèle. Elle supprime les différences lors de la compilation des shaders en imposant un format de code intermédiaire (bytecode) interprété par un compilateur officiel. Enfin, la bibliothèque prend en compte le caractère générique des cartes graphiques et permet d'accéder aux fonctionnalités génériques et graphiques par le biais d'une unique API.

II-B - Le nécessaire pour afficher un triangle

Nous allons maintenant nous intéresser aux étapes nécessaires à l'affichage d'un triangle dans un programme Vulkan correctement conçu. Tous les concepts ici évoqués seront développés dans les prochains chapitres. Le but ici est simplement de vous donner une vue d'ensemble du processus afin que ce soit plus clair pour la suite.

II-B-1 - Étape 1 - Instance et sélection d'un périphérique physique

Une application commence par initialiser la bibliothèque à l'aide d'une `VkInstance`. Une instance est créée en décrivant votre application et les extensions que vous comptez utiliser. Après avoir créé votre instance, vous pouvez demander l'accès au matériel compatible avec Vulkan et ainsi sélectionner un ou plusieurs `VkPhysicalDevice` à utiliser. Vous pouvez récupérer des informations telles que la taille de la VRAM ou les fonctionnalités offertes par le périphérique sélectionné et ainsi choisir de travailler avec un matériel adapté pour votre application.

II-B-2 - Étape 2 – Périphérique logique et familles de queues

Après avoir sélectionné le matériel adéquat, vous devez créer un `VkDevice` (périphérique logique) au travers duquel vous allez définir les `VkPhysicalDeviceFeatures` que vous utiliserez. Par exemple, l'affichage multifenêtre ou le support des nombres flottants sur 64 bits. Vous devrez également spécifier quelles `vkQueueFamilies` (famille de queues) vous utiliserez. La plupart des opérations, comme les commandes d'affichage et les opérations mémoire, sont exécutées de manière asynchrone en les envoyant à une `VkQueue`. Ces queues sont créées à partir d'une famille de queues. Chaque famille supporte uniquement un sous-ensemble d'opérations. Il pourrait par exemple y avoir des familles différentes pour les graphismes, le calcul et les opérations mémoire. L'existence des familles peut aussi être utilisée comme critère pour la sélection d'un périphérique physique. Il est possible qu'un périphérique supportant Vulkan ne possède aucune fonctionnalité graphique. Toutefois, une carte graphique supportant Vulkan devrait nous offrir le support des opérations qui nous intéressent.

II-B-3 - Étape 3 – Surface d'affichage et swap chain

À moins que vous ne soyez intéressé que par le rendu hors écran, vous devrez créer une fenêtre pour y afficher votre rendu. Les fenêtres peuvent être créées avec les bibliothèques spécifiques aux différentes plateformes ou avec des bibliothèques telles que **GLFW** et **SDL**. Ce tutoriel repose sur GLFW, mais nous verrons cela dans le prochain chapitre.

Nous avons besoin de deux composants pour afficher quoi que ce soit : une surface (`VkSurfaceKHR`) et une « swap chain » (`VkSwapchainKHR`). Remarquez le suffixe « `KHR` », qui indique que ces fonctionnalités font partie d'une extension. La bibliothèque en elle-même est totalement agnostique de la plateforme, nous devons donc utiliser l'extension standard WSI (Window System Interface) pour interagir avec le gestionnaire de fenêtres. La surface est une abstraction multiplateforme de la fenêtre sur laquelle réaliser l'affichage. Elle est généralement créée à partir d'une référence à une fenêtre native, par exemple un `HWND` sur Windows. Heureusement, la bibliothèque GLFW possède une fonction permettant de gérer tous les détails spécifiques à la plateforme pour nous.

La « swap chain » est une collection de cibles de rendu. Son but principal est d'assurer que l'image sur laquelle nous travaillons n'est pas celle utilisée par l'écran. C'est important pour s'assurer que l'image affichée est complète. Chaque fois que nous voudrons afficher une image, nous devrons demander à la swap chain de nous fournir une image dans laquelle dessiner. Une fois le rendu effectué, l'image est rendue à la « swap chain » qui la donnera à l'écran au moment voulu. Le nombre de cibles et les conditions d'affichage de l'image finale dépendent du mode de présentation. Les modes les plus communs utilisent deux ou trois tampons (« double buffering » ou « triple buffering »). Nous détaillerons tout cela dans le [chapitre dédié à la « swap chain »](#).

Certaines plateformes permettent d'effectuer un rendu directement à l'écran sans passer par un gestionnaire de fenêtres grâce aux extensions `VK_KHR_display` et `VK_KHR_display_swapchain`. Celles-ci permettent de créer une surface représentant l'intégralité de l'écran et peuvent être utilisées pour implémenter votre propre gestionnaire de fenêtres.

II-B-4 - Étape 4 – Vue d'image et tampons d'image

Pour dessiner sur une image provenant de la « swap chain », nous devons l'encapsuler dans une `VkImageView` (vue d'image ou « image view ») et un `VkFramebuffer`. Une vue sur une image référence la partie d'une image à utiliser et un tampon d'image référence les vues qui seront utilisées pour les cibles de couleur, de profondeur ou de

stencil. Dans la mesure où il peut y avoir de nombreuses images dans la « swap chain », nous créerons en amont les vues et les tampons d'image pour chacune d'entre elles, puis sélectionnerons celles qui nous conviennent au moment de l'affichage.

II-B-5 - Étape 5 – Passes de rendu

La passe de rendu décrit le type des images utilisé lors des opérations de rendu, comment elles sont utilisées et comment leur contenu doit être traité. Pour l'affichage d'un triangle, nous indiquerons à Vulkan que nous utilisons une seule image pour la couleur et que nous voulons qu'elle soit remplie d'une couleur opaque avant l'affichage. Là où la passe de rendu décrit seulement le type des images, un tampon d'image (`VkFramebuffer`) lie les images appropriées à la passe.

II-B-6 - Étape 6 - Le pipeline graphique

Le pipeline graphique est configuré lors de la création d'un `VkPipeline`. Il décrit les éléments paramétrables de la carte graphique, comme la taille du viewport, les opérations réalisées sur le tampon de profondeur (depth buffer) et les étapes programmables à l'aide de `VkShaderModule`. Ces derniers sont créés à partir du code intermédiaire généré à partir des shaders. Le pilote doit également être informé des cibles du rendu utilisées dans le pipeline, ce que nous lui donnons en référençant la passe de rendu.

L'une des particularités les plus importantes de Vulkan est que la quasi-totalité de la configuration des étapes doit être réalisée à l'avance. Cela implique que si vous voulez changer un shader ou la disposition des données des sommets alors la totalité du pipeline doit être recréée. Vous aurez donc probablement de nombreux `VkPipeline` correspondant à toutes les combinaisons dont votre programme aura besoin. Seules quelques configurations basiques peuvent être changées de manière dynamique, comme le viewport ou la couleur de fond. Les états doivent aussi être définis explicitement : il n'y a par exemple pas de fonction de mélange (blending) par défaut.

La bonne nouvelle est que toute cette anticipation est comparable à une compilation en avance contrairement à une compilation juste à temps. Le pilote a plus d'opportunités d'optimisation et les performances à l'exécution sont prédictibles, car tous les changements d'état sont explicites.

II-B-7 - Étape 7 – Groupe de commandes et tampons de commandes

Comme dit plus haut, de nombreuses opérations telles que les opérations de rendu que nous souhaitons exécuter doivent être envoyées à une queue. Ces opérations doivent d'abord être enregistrées dans un tampon de commandes (`VkCommandBuffer`) avant de pouvoir être envoyées. Ces tampons de commandes sont alloués à partir d'une `VkCommandPool` associée à une famille de queues spécifique. Pour afficher un simple triangle nous devrons enregistrer un tampon de commandes avec les opérations suivantes :

- commencer la passe de rendu ;
- lier le pipeline graphique ;
- afficher trois sommets ;
- terminer la passe de rendu.

Sachant que l'image dans le tampon d'image dépend de l'image fournie par la swap chain, nous devons préparer un tampon de commandes pour chaque image possible et choisir la bonne à l'affichage. Nous pourrions enregistrer un tampon de commandes à chaque image, mais ce n'est pas aussi efficace.

II-B-8 - Étape 8 - Boucle principale

Maintenant que nous avons inscrit les commandes graphiques dans un tampon de commandes, la boucle principale est simple. D'abord, nous acquérons une image fournie par la « swap chain » en utilisant `vkAcquireNextImageKHR`. Nous sélectionnons ensuite le tampon de commandes adéquat pour cette image et le postons à la queue avec

vkQueueSubmit. Enfin, nous retournons l'image à la « swap chain » pour sa présentation à l'écran à l'aide de vkQueuePresentKHR.

Les opérations envoyées à la queue sont exécutées de manière asynchrone. Nous devons donc utiliser des mécanismes de synchronisation tels que des sémaphores pour nous assurer que les opérations sont exécutées dans l'ordre voulu. L'exécution du tampon de commandes pour l'affichage doit être configurée pour attendre la fin de l'acquisition de l'image, sinon nous pourrions dessiner sur une image en cours d'utilisation pour l'affichage. L'appel à vkQueuePresentKHR doit aussi attendre que le rendu soit terminé. Pour cela, nous utilisons un deuxième semaphore déclenché après la fin du rendu.

II-B-9 - Résumé

Ce rapide tour devrait vous donner une compréhension basique du travail que nous aurons à fournir pour afficher notre premier triangle. Un véritable programme contient plus d'étapes comme allouer des tampons de sommets, créer les tampons de variables uniformes et envoyer à la carte graphique les images pour les textures, mais nous verrons cela dans des chapitres suivants. Nous allons commencer simplement, car Vulkan a une courbe d'apprentissage abrupte. Notez que nous allons « tricher » en écrivant les coordonnées du triangle directement dans un shader au lieu d'utiliser un tampon de sommets. En effet, la gestion d'un tampon de sommets nécessite quelques familiarités avec les tampons de commandes.

En résumé, pour afficher un triangle, nous devons :

- créer une VkInstance ;
- sélectionner une carte graphique compatible (VkPhysicalDevice) ;
- créer un VkDevice et une VkQueue pour le rendu et la présentation ;
- créer une fenêtre, une surface associée à la fenêtre et une « swap chain » ;
- associer les images de la « swap chain » aux VkImageView ;
- créer la passe de rendu spécifiant les cibles de rendu et leur utilisation ;
- créer des tampons d'image pour ces passes ;
- générer le pipeline graphique ;
- allouer et enregistrer un tampon de commandes contenant les commandes de rendu pour toutes les images de la « swap chain » ;
- dessiner sur les tampons en acquérant une image, puis en soumettant le bon tampon de commandes et en renvoyant l'image à la « swap chain ».

Cela fait beaucoup d'étapes, cependant le but de chacune d'entre elles sera expliqué clairement et simplement dans les chapitres suivants. Si vous êtes confus quant à l'intérêt d'une étape dans le programme entier, référez-vous à ce premier chapitre.

II-C - Concepts de la bibliothèque

Ce chapitre va conclure en survolant la structure de la bibliothèque à un plus bas niveau.

II-C-1 - Conventions

Toutes les fonctions, les énumérations et les structures de Vulkan sont définies dans le fichier d'en-têtes *vulkan.h*, inclus dans le **SDK Vulkan** développé par LunarG. Nous verrons comment l'installer dans le prochain chapitre.

Les fonctions sont préfixées par 'vk', les types comme les énumérations et les structures par 'Vk' et les macros par 'VK_'. La bibliothèque utilise massivement les structures pour la création d'objets plutôt que de passer des arguments à des fonctions. Par exemple la création d'objets suit généralement le schéma suivant :

```
VkXXXCreateInfo createInfo = {};
createInfo.sType = VK_STRUCTURE_TYPE_XXX_CREATE_INFO;
createInfo.pNext = nullptr;
```

```
createInfo.foo = ...;
createInfo.bar = ...;

VkXXX object;
if (vkCreateXXX(&createInfo, nullptr, &object) != VK_SUCCESS) {
    std::cerr << "failed to create object" << std::endl;
    return false;
}
```

De nombreuses structures imposent que l'on spécifie explicitement leur type dans le membre sType. Le membre pNext peut pointer vers une extension de la structure et sera toujours nullptr dans ce tutoriel. Les fonctions qui créent ou détruisent les objets ont un paramètre appelé VkAllocationCallbacks, qui vous permet de spécifier un allocateur. Nous le mettrons également à nullptr.

La plupart des fonctions retournent un VkResult, qui peut être soit VK_SUCCESS soit un code d'erreur. La spécification décrit quelles erreurs sont retournées par chaque fonction et ce qu'elles signifient.

II-C-2 - Couches de validation

Vulkan est pensé pour la haute performance et pour un travail minimal pour le pilote. Par conséquent, il inclut, par défaut, très peu de gestion d'erreurs et de système de débogage. Le pilote crashera beaucoup plus souvent qu'il ne retournera de code d'erreur si vous faites quelque chose d'incorrect. Pire, il peut fonctionner sur votre carte graphique, mais pas sur une autre.

Cependant, Vulkan vous permet d'activer des vérifications précises à l'aide d'une fonctionnalité nommée couches de validation (validation layers). Ces couches consistent en du code s'insérant entre la bibliothèque et le pilote et permettent de lancer des analyses de mémoire et de relever les défauts. Vous pouvez les activer pendant le développement et les désactiver lors de la mise en production de votre code. Une fois désactivées, il n'y aura aucune conséquence sur la performance. N'importe qui peut écrire ses couches de validation, mais le SDK de LunarG fournit un ensemble de validations que nous allons utiliser dans ce tutoriel. Vous aurez cependant à écrire vos propres fonctions de callback pour récupérer les messages de débogage provenant des couches.

Du fait que Vulkan est explicite pour chaque opération et grâce à l'extensivité des couches de validation, il est plus facile de comprendre pourquoi l'écran est noir qu'avec OpenGL ou Direct3D !

Il reste une dernière étape avant de commencer à coder : mettre en place [l'environnement de développement](#).

III - Environnement de développement

Dans ce chapitre nous allons paramétrier l'environnement de développement pour Vulkan et installer quelques bibliothèques utiles. Tous les outils que nous allons utiliser, excepté le compilateur, seront compatibles Windows, Linux et MacOS. Cependant les étapes pour les installer diffèrent un peu, d'où les sections suivantes.

III-A - Windows

Si vous développez sur Windows, je partirai du principe que vous utilisez Visual Studio. Pour avoir un support complet du C++17, vous pouvez utiliser soit Visual Studio 2017, soit Visual Studio 2019. Les étapes ci-dessous ont été écrites avec Visual Studio 2017.

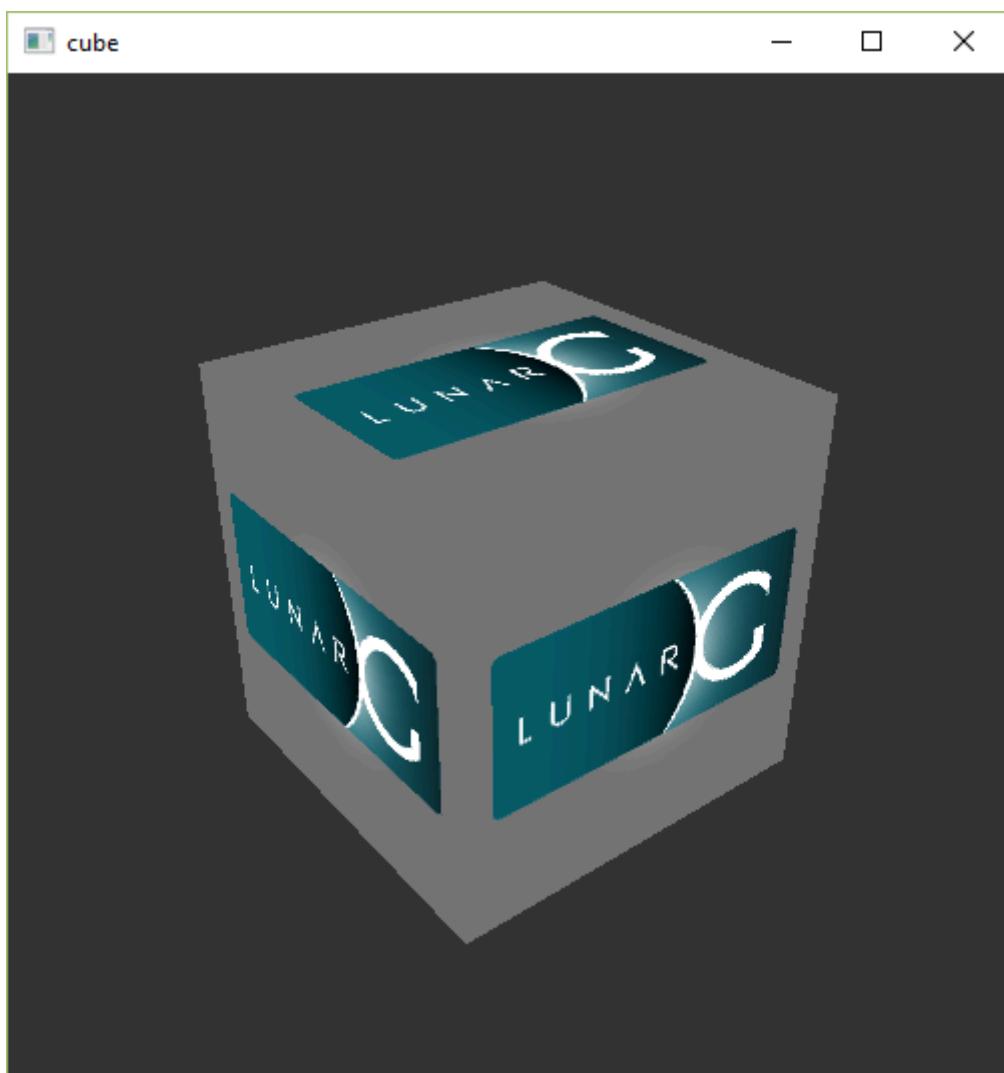
III-A-1 - SDK Vulkan

Le composant central du développement d'applications Vulkan est le SDK. Il inclut les fichiers d'en-têtes, les couches de validation standards, des outils de débogage et un chargeur pour les fonctions Vulkan. Ce chargeur récupère, à l'exécution, les fonctions exposées par le pilote, comme le ferait GLEW pour OpenGL.

Le SDK peut être téléchargé sur [le site de LunarG](#) en utilisant les boutons en bas de page. Vous n'avez pas besoin de compte, mais celui-ci vous donne accès à une documentation supplémentaire qui pourra vous être utile.



Réalisez l'installation et notez l'emplacement du SDK. La première chose que nous allons faire est de vérifier si votre carte graphique supporte Vulkan. Allez dans le dossier d'installation du SDK, ouvrez le dossier Bin et lancez vkcubes.exe. Vous devriez voir la fenêtre suivante :



Si vous recevez un message d'erreur, assurez-vous que votre pilote graphique est à jour, qu'il inclut Vulkan et que votre carte graphique est supportée. Référez-vous au [chapitre introductif](#) pour les liens vers les principaux constructeurs.

Il y a d'autres programmes utiles pour le développement dans ce dossier : glslangValidator.exe et gslc.exe. Nous en aurons besoin pour la compilation des shaders. Ils transforment un code facilement compréhensible et semblable au C (le **GLSL**) en code intermédiaire (bytecode). Nous couvrirons cela dans le chapitre des **modules shader**. Le

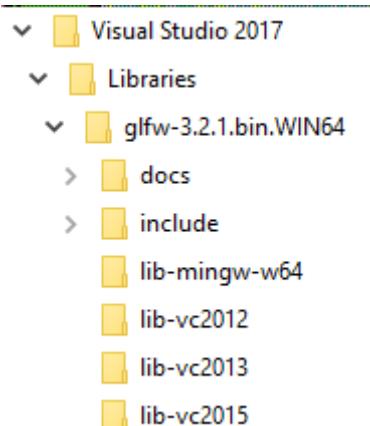
dossier Bin contient aussi les fichiers binaires du loader Vulkan et des couches de validation. Le dossier Lib contient les bibliothèques.

Enfin, le dossier Include contient les fichiers d'en-têtes de Vulkan. Vous pouvez parcourir les autres fichiers, mais nous ne les utiliserons pas dans ce tutoriel.

III-A-2 - GLFW

Comme dit précédemment, Vulkan ignore la plateforme sur laquelle il opère, et n'inclut pas d'outil de création de fenêtres où afficher notre rendu. Pour bien exploiter les possibilités multiplateformes de Vulkan et éviter les horreurs de Win32, nous utiliserons la **bibliothèque GLFW** pour créer une fenêtre, et ce, sur Windows, Linux ou MacOS. Il existe d'autres bibliothèques telles que **SDL**, mais GLFW a l'avantage d'abstraire d'autres aspects spécifiques à la plateforme requis par Vulkan.

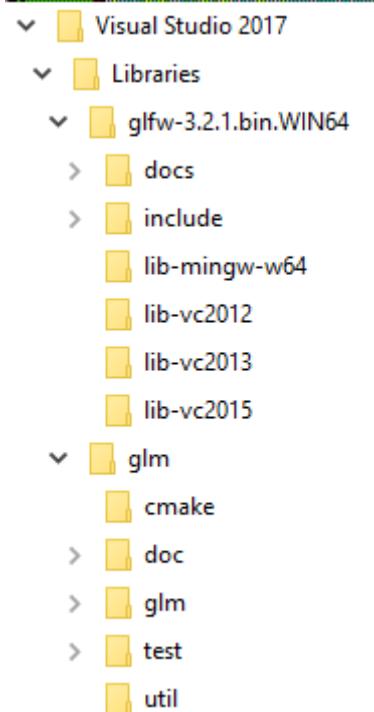
Vous pouvez trouver la dernière version de GLFW sur leur [site officiel](#). Nous utiliserons la version 64 bits, mais vous pouvez également utiliser la version 32 bits. Dans ce cas, assurez-vous de bien lier le dossier "Lib32" dans le SDK et non "Lib". Après avoir téléchargé GLFW, extrayez l'archive à l'emplacement qui vous convient. J'ai choisi de créer un dossier "Librairies" dans le dossier de Visual Studio (dans Mes Documents).



III-A-3 - GLM

Contrairement à DirectX 12, Vulkan n'intègre pas de bibliothèque pour l'algèbre linéaire. Nous devons donc en télécharger une. **GLM** est une bonne bibliothèque conçue pour être utilisée avec les bibliothèques graphiques et est souvent utilisée avec OpenGL.

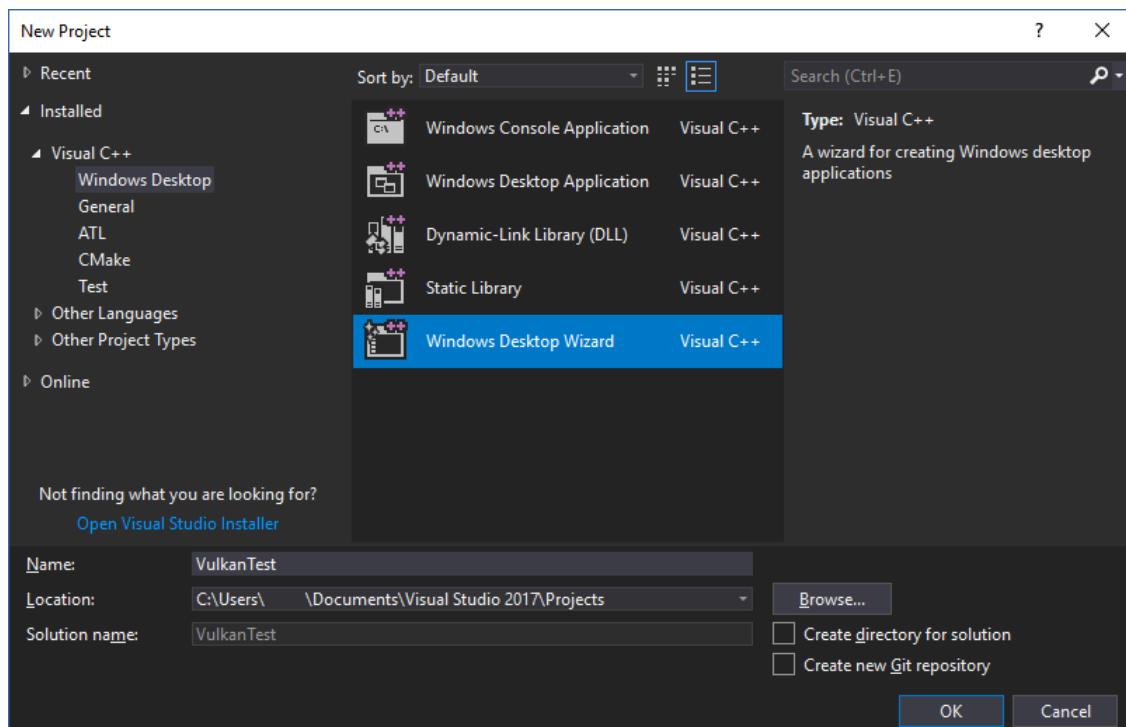
GLM est une bibliothèque écrite exclusivement dans les fichiers d'en-têtes. Il suffit donc d'en télécharger la [dernière version](#), la stocker où vous le souhaitez et l'inclure là où vous en aurez besoin. Vous devez maintenant obtenir une structure semblable :



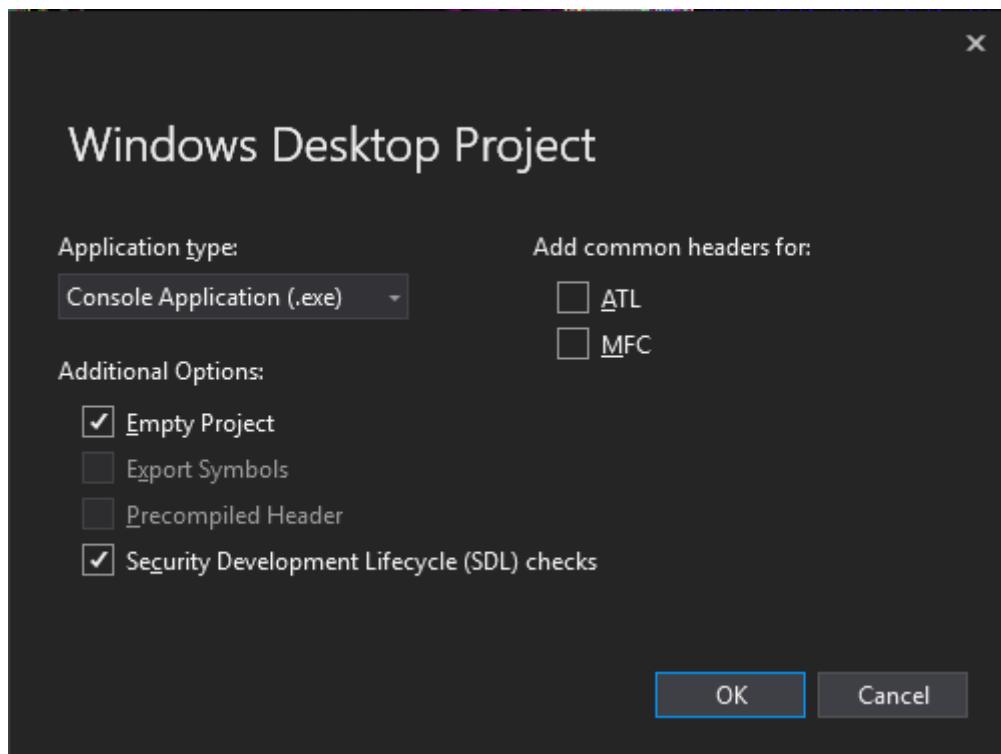
III-A-4 - Préparer Visual Studio

Maintenant que vous avez installé toutes les dépendances, nous pouvons préparer un projet Visual Studio pour Vulkan et écrire un peu de code pour vérifier que tout fonctionne.

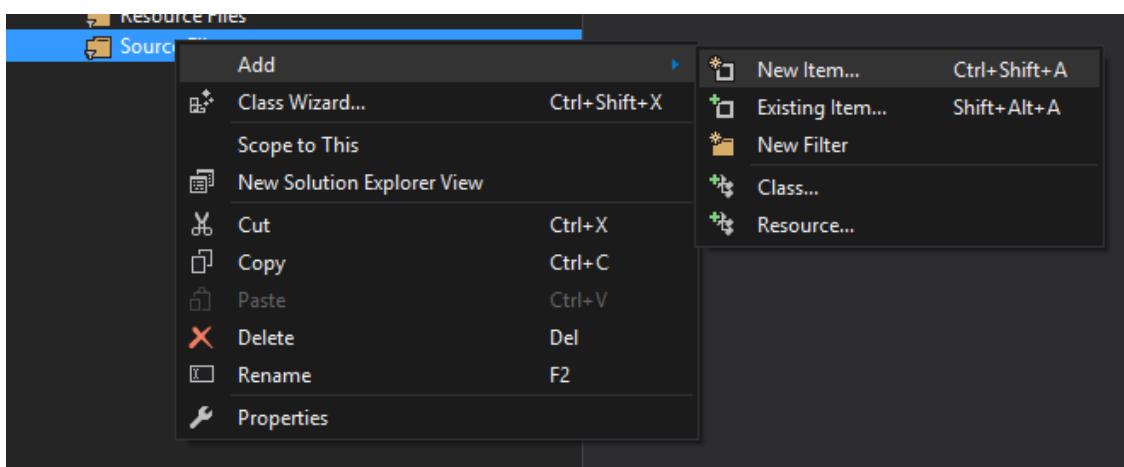
Lancez Visual Studio et créez un nouveau projet **Windows Desktop Wizard**, entrez un nom et appuyez sur OK.

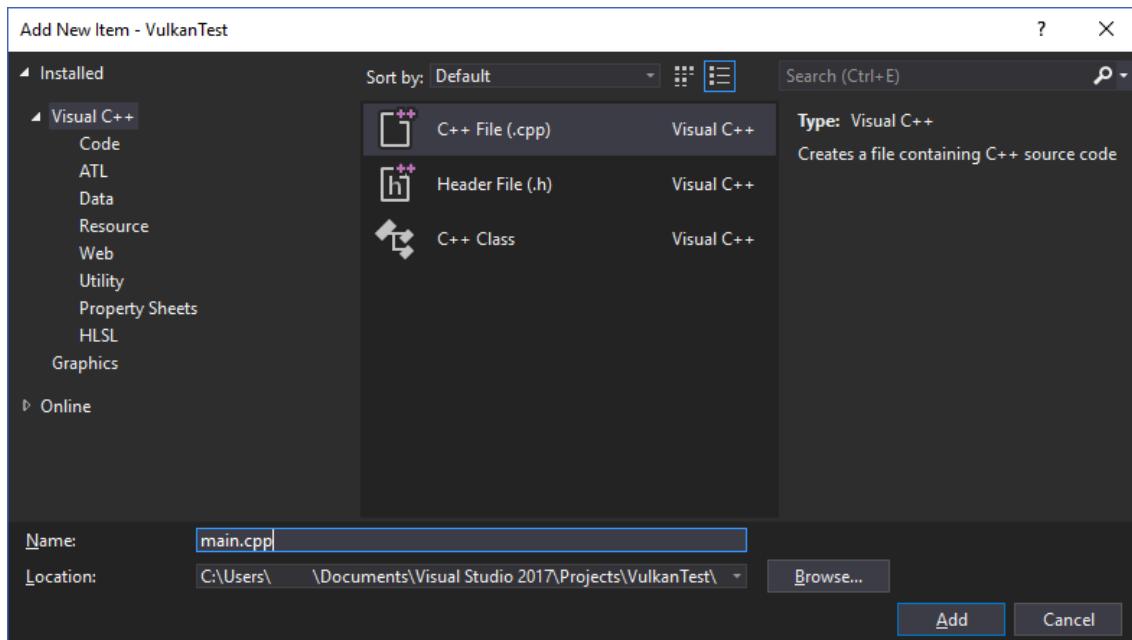


Assurez-vous que **Console Application (.exe)** est sélectionné comme type d'application afin que nous ayons un endroit où afficher nos messages d'erreurs et cochez **Empty Project** afin que Visual Studio ne génère pas un code de base.



Appuyez sur OK pour créer le projet et ajoutez un fichier source C++. Vous devriez déjà savoir faire ça, mais les étapes sont tout de même incluses ici.





Ajoutez maintenant le code suivant à votre fichier. Ne cherchez pas à le comprendre pour le moment, il sert juste à s'assurer que vous pouvez compiler et lancer une application Vulkan. Nous recommencerons tout depuis le début dès le chapitre suivant.

```
#define GLFW_INCLUDE_VULKAN
#include <GLFW/glfw3.h>

#define GLM_FORCE_RADIANS
#define GLM_FORCE_DEPTH_ZERO_TO_ONE
#include <glm/vec4.hpp>
#include <glm/mat4x4.hpp>

#include <iostream>

int main() {
    glfwInit();

    glfwWindowHint(GLFW_CLIENT_API, GLFW_NO_API);
    GLFWwindow* window = glfwCreateWindow(800, 600, "Vulkan window", nullptr, nullptr);

    uint32_t extensionCount = 0;
    vkEnumerateInstanceExtensionProperties(nullptr, &extensionCount, nullptr);

    std::cout << extensionCount << " extensions supported\n";

    glm::mat4 matrix;
    glm::vec4 vec;
    auto test = matrix * vec;

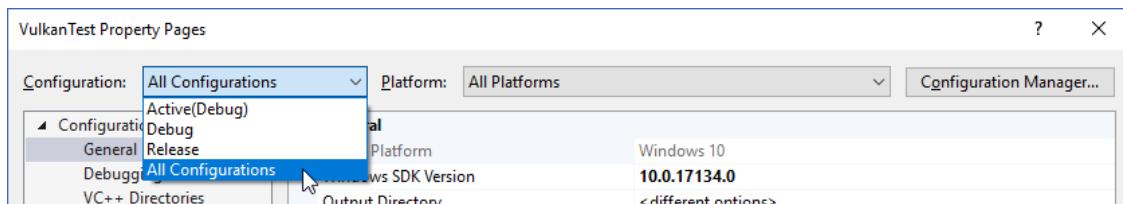
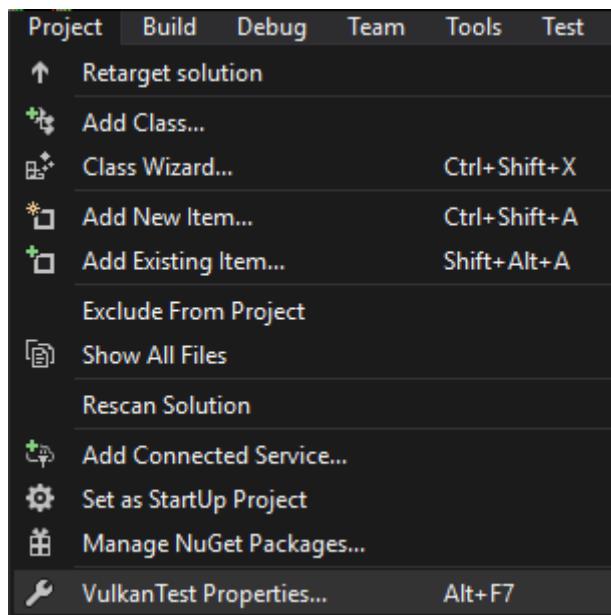
    while (!glfwWindowShouldClose(window)) {
        glfwPollEvents();
    }

    glfwDestroyWindow(window);

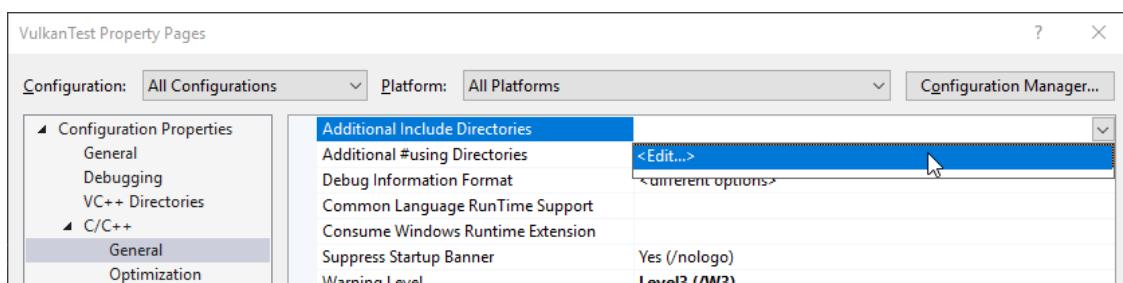
    glfwTerminate();

    return 0;
}
```

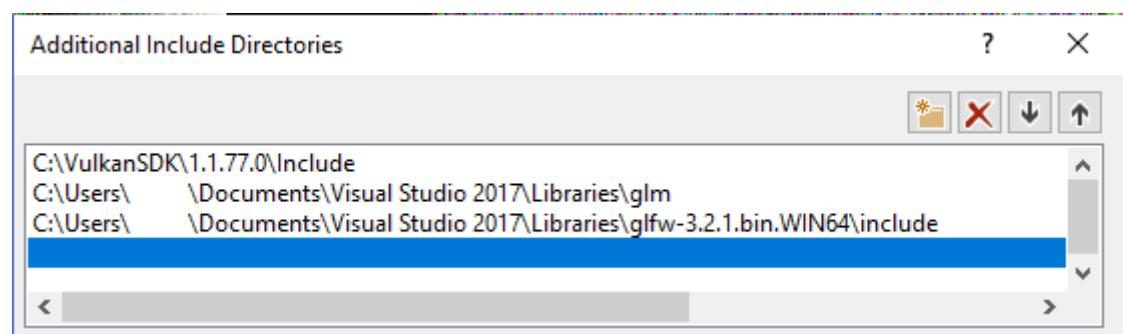
Configurons maintenant le projet afin de nous débarrasser des erreurs. Ouvrez les propriétés du projet et assurez-vous que « Toutes les configurations » (All Configurations) soit sélectionné, car la plupart des paramètres s'appliquent autant dans le mode Debug que dans le mode Release.



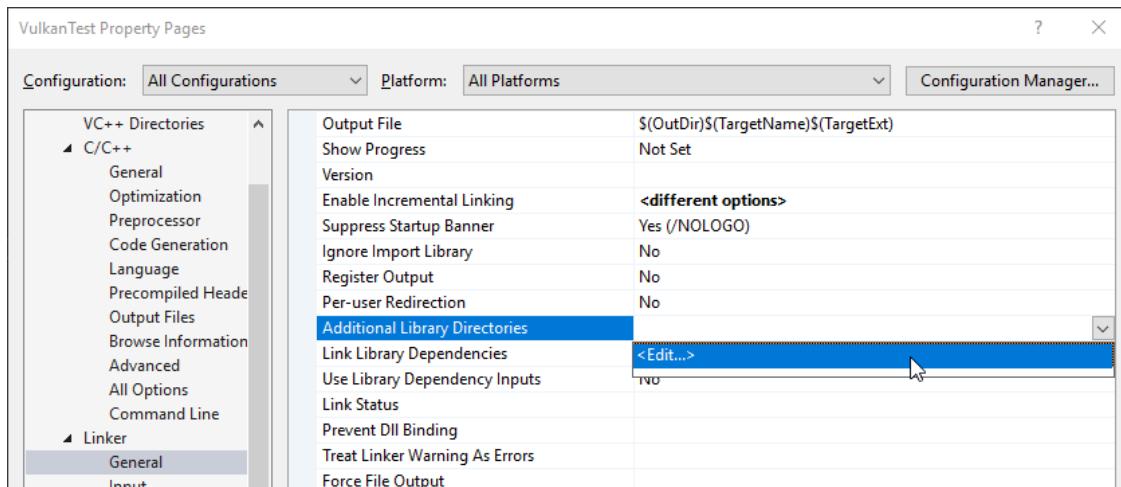
Allez à C++ → General → Autres répertoires Include et cliquez sur <Modifier...> dans le menu déroulant.



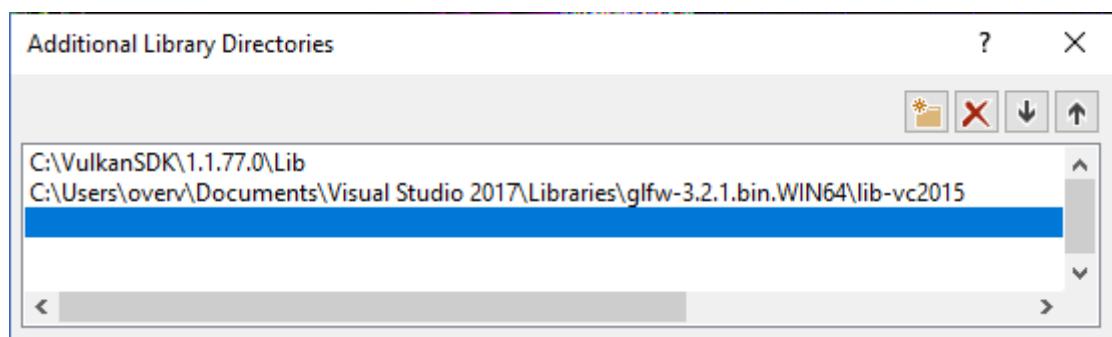
Ajoutez les dossiers contenant les fichiers d'en-têtes de Vulkan, GLFW et GLM :



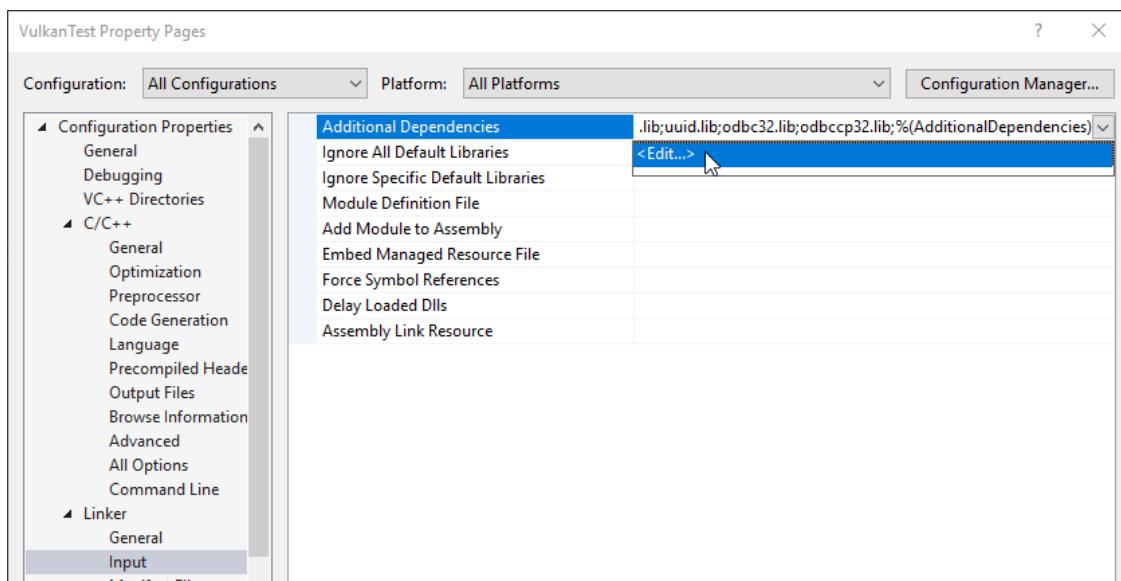
Ensuite, ouvrez l'éditeur pour les dossiers des bibliothèques sous Éditeur de liens → General :



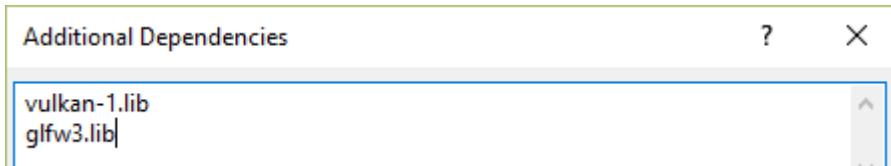
Et ajoutez les emplacements des fichiers objet pour Vulkan et GLFW :



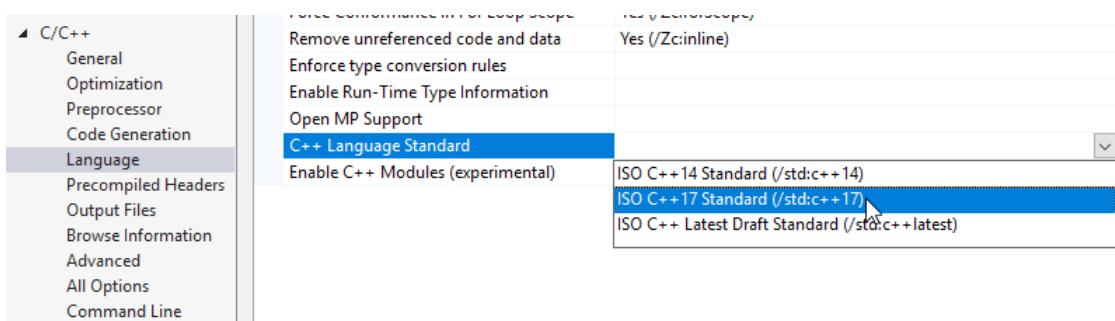
Allez à Linker → Entrée et cliquez sur <Modifier...> dans le menu déroulant de l'entrée Dépendances supplémentaires (Additional Dependencies) :



Entrez les noms des fichiers objet GLFW et Vulkan :

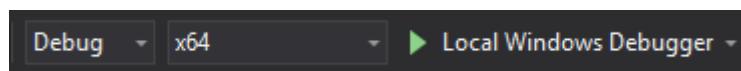


Finalement, activez le support des fonctionnalités du C++17 :

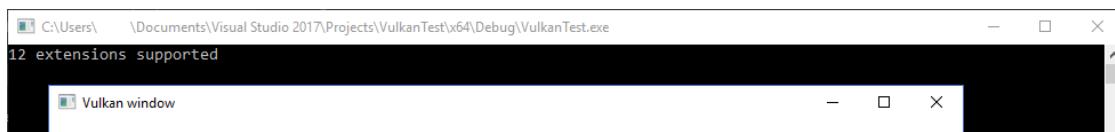


Vous pouvez enfin fermer la fenêtre des propriétés. Si vous avez tout fait correctement, vous ne devriez plus voir d'erreur dans votre code.

Finalement, assurez-vous que vous compilez effectivement en 64 bits :



Appuyez sur F5 pour compiler et lancer le projet. Vous devriez voir une fenêtre s'afficher comme cela :



Si le nombre d'extensions est nul, il y a un problème avec la configuration de Vulkan sur votre système. Sinon, vous êtes fin prêt à vous [lancer avec Vulkan](#) !

III-B - Linux

Ces instructions sont conçues pour les utilisateurs d'Ubuntu, mais vous devriez pouvoir suivre ces instructions depuis une autre distribution si vous adaptez les commandes apt à votre propre gestionnaire de paquets. Vous avez besoin d'un compilateur supportant le C++17 (GCC 7 ou supérieur ou Clang 5 ou supérieur). Vous avez aussi besoin de make.

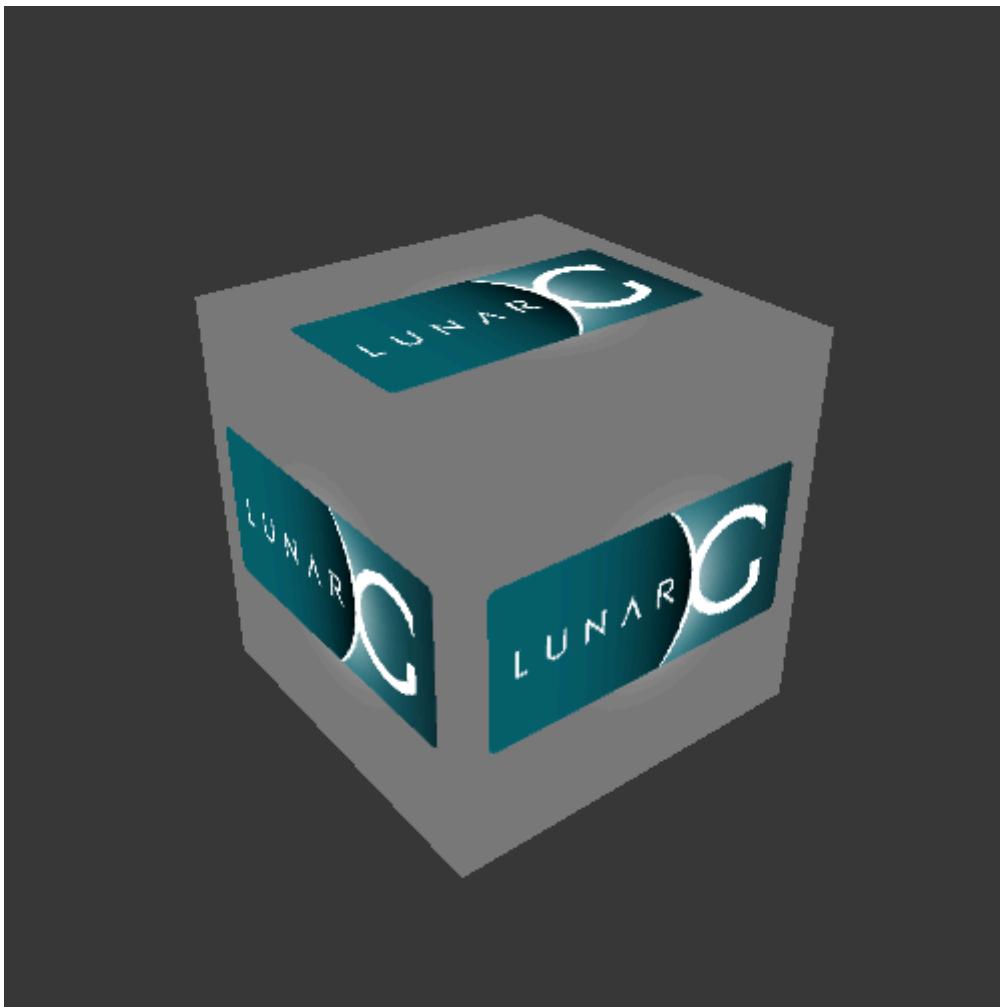
III-B-1 - Paquets Vulkan

Les composants les plus importants pour le développement d'applications Vulkan sous Linux sont le chargeur Vulkan, les couches de validations et quelques outils en ligne de commande pour tester si votre machine est capable d'exécuter une application Vulkan :

- `sudo apt install vulkan-tools` : installe les outils en ligne de commande, plus précisément `vulkaninfo` et `vkcube`. Lancez-les pour savoir si votre machine peut exécuter des applications Vulkan ;
- `sudo apt install libvulkan-dev` : installe le chargeur Vulkan. Il permet de récupérer les fonctions Vulkan auprès du pilote du GPU, de la même façon que peut le faire GLEW pour OpenGL ;

- sudo apt install vulkan-validationlayers-dev : installe les couches de validation standards. Celles-ci sont cruciales pour déboguer votre application Vulkan. Nous en reparlerons dans le prochain chapitre.

Si l'installation a réussi, vous devriez être prêt en ce qui concerne Vulkan. N'oubliez pas de lancer vkcube, qui devrait vous afficher la fenêtre suivante :



III-B-2 - GLFW

Comme dit précédemment, Vulkan ignore la plateforme sur laquelle il opère, et n'inclut pas d'outil de création de fenêtres où afficher notre rendu. Pour bien exploiter les possibilités multiplateformes de Vulkan, nous utiliserons la **bibliothèque GLFW** pour créer une fenêtre sur Windows, Linux ou MacOS. Il existe d'autres bibliothèques telles que **SDL**, mais GLFW a l'avantage d'abstraire d'autres aspects spécifiques à la plateforme requis par Vulkan.

Nous allons installer la bibliothèque GLFW grâce à la commande suivante :

```
sudo apt install libglfw3-dev
```

III-B-2-a - GLM

Contrairement à DirectX 12, Vulkan n'intègre pas de bibliothèque pour l'algèbre linéaire. Nous devons donc en télécharger une. **GLM** est une bonne bibliothèque conçue pour être utilisée avec les bibliothèques graphiques et est souvent utilisée avec OpenGL.

Cette bibliothèque contenue intégralement dans les fichiers d'en-têtes peut être installée depuis le paquet libglm-dev :

```
1. sudo apt install libglm-dev
```

III-B-3 - Compilateur de shader

Nous avons presque tout le nécessaire, sauf un programme pour compiler les shaders en GLSL, un langage compréhensible par les humains, vers du bytecode.

Il existe deux compilateurs de shader populaires : glslangValidator de Khronos et glslc de Google. Ce dernier s'utilise d'une façon similaire à GCC et Clang et c'est pourquoi nous allons l'utiliser. Téléchargez les **exécutables non officiels** et copiez le fichier glslc dans votre répertoire `/usr/local/bin`. Notez que vous devrez certainement utiliser `sudo` en fonction de vos permissions. Pour tester, tapez la commande `glslc` dans un terminal : celui-ci devrait vous retourner le message suivant, stipulant que vous ne lui avez pas spécifié de shader à compiler :

```
glslc: error: no input files
```

Nous couvrirons l'utilisation de `glslc` plus en détail dans le chapitre des **modules shaders**.

III-B-4 - Préparation d'un projet avec Makefile

Maintenant que vous avez installé toutes les dépendances, nous pouvons préparer un Makefile basique pour Vulkan et écrire un code très simple pour s'assurer que tout fonctionne correctement.

Créez un nouveau dossier là où cela vous plaît et nommez-le `VulkanTest`. Créez un fichier `main.cpp` et insérez-y le code suivant. Ne cherchez pas à le comprendre maintenant : il sert juste à s'assurer que vous pouvez compiler et exécuter une application Vulkan. Nous recommencerons tout depuis le début dès le chapitre suivant.

```
#define GLFW_INCLUDE_VULKAN
#include <GLFW/glfw3.h>

#define GLM_FORCE_RADIANS
#define GLM_FORCE_DEPTH_ZERO_TO_ONE
#include <glm/vec4.hpp>
#include <glm/mat4x4.hpp>

#include <iostream>

int main() {
    glfwInit();

    glfwWindowHint(GLFW_CLIENT_API, GLFW_NO_API);
    GLFWwindow* window = glfwCreateWindow(800, 600, "Vulkan window", nullptr, nullptr);

    uint32_t extensionCount = 0;
    vkEnumerateInstanceExtensionProperties(nullptr, &extensionCount, nullptr);

    std::cout << extensionCount << " extensions supported\n";

    glm::mat4 matrix;
    glm::vec4 vec;
    auto test = matrix * vec;

    while (!glfwWindowShouldClose(window)) {
        glfwPollEvents();
    }

    glfwDestroyWindow(window);

    glfwTerminate();

    return 0;
}
```

{}

Nous allons maintenant créer un Makefile pour compiler et lancer ce code. Créez un fichier Makefile. Je pars du principe que vous connaissez déjà les Makefiles, notamment le fonctionnement des variables et des règles. Sinon vous pouvez trouver des introductions claires sur internet, par exemple [ici](#).

Nous allons d'abord définir quelques variables pour simplifier le reste du fichier. Définissez la variable `CFLAGS` qui spécifiera les options de base du compilateur :

```
CFLAGS = -std=c++17 -O2
```

Nous utiliserons du C++ moderne (`-std=c++17`) et nous activons les optimisations avec `-O2`. Vous pouvez retirer cette option pour compiler les programmes plus rapidement, mais n'oubliez pas de la remettre pour compiler des exécutables pour la production.

Définissez de manière analogue la variable `LDFLAGS` :

```
LDFLAGS = -lglfw -lvulkan -ldl -lpthread -lx11 -lxxf86vm -lXrandr -lx1i
```

L'option `-lglfw` est pour la bibliothèque GLFW. L'option `-lvulkan` correspond au chargeur dynamique des fonctions Vulkan. Les autres options sont nécessaires pour GLFW et correspondent à des bibliothèques bas niveau pour la gestion des threads et des fenêtres.

La spécification de la règle pour la compilation de `VulkanTest` est désormais un jeu d'enfant. Assurez-vous que vous utilisez des tabulations et non des espaces pour l'indentation.

```
VulkanTest: main.cpp
g++ $(CFLAGS) -o VulkanTest main.cpp $(LDFLAGS)
```

Vérifiez que cette règle fonctionne en sauvegardant le fichier et en exécutant `make` depuis un terminal ouvert dans le dossier contenant les fichiers `main.cpp` et `Makefile`. Vous devriez avoir un exécutable appelé `VulkanTest`.

Nous allons ensuite définir deux règles, `test` et `clean`. La première exécutera le programme et le second supprimera l'exécutable :

```
.PHONY: test clean

test: VulkanTest
./VulkanTest

clean:
rm -f VulkanTest
```

La commande `make clean` provoque l'exécution du programme. Celui-ci affiche le nombre d'extensions Vulkan disponible ainsi qu'une fenêtre vide. L'application doit retourner le code de retour 0 (succès) quand vous fermez la fenêtre.

Vous devriez désormais avoir un Makefile ressemblant à ceci :

```
CFLAGS = -std=c++17 -O2
LDFLAGS = -lglfw -lvulkan -ldl -lpthread -lx11 -lxxf86vm -lXrandr -lx1i

VulkanTest: main.cpp
g++ $(CFLAGS) -o VulkanTest main.cpp $(LDFLAGS)

.PHONY: test clean

test: VulkanTest
./VulkanTest
```

```
clean:  
rm -f VulkanTest
```

Vous pouvez désormais utiliser ce dossier comme base pour vos projets Vulkan. Faites-en une copie, renommez-le en HelloTriangle et videz le fichier *main.cpp*.

Bravo, vous êtes fin prêt à vous [lancer avec Vulkan!](#)

III-C - MacOS

Ces instructions partent du principe que vous utilisez Xcode et le [gestionnaire de packages Homebrew](#). Vous aurez besoin de MacOS 10.11 au minimum et votre ordinateur doit supporter la [bibliothèque Metal](#).

III-C-1 - SDK Vulkan

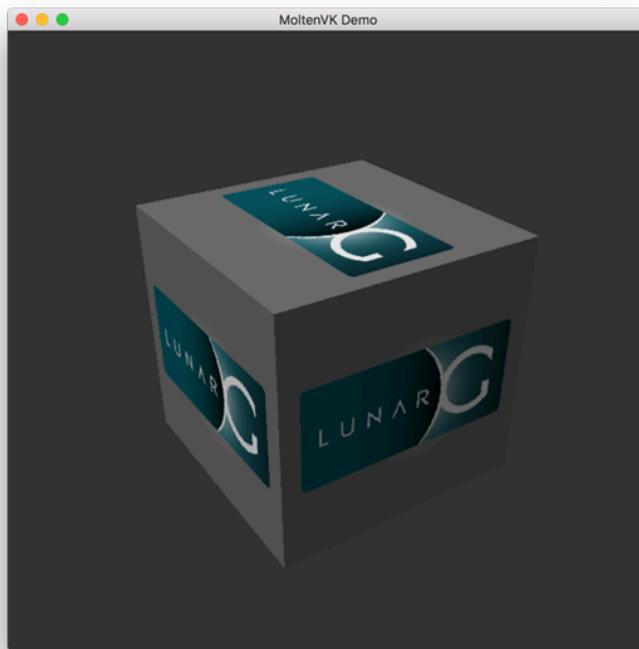
Le composant central du développement d'applications Vulkan est le SDK. Il inclut les fichiers d'en-têtes, les couches de validation standards, des outils de débogage et un chargeur pour les fonctions Vulkan. Ce chargeur récupère les fonctions exposées par le pilote à l'exécution, comme GLEW pour OpenGL, si cela vous parle.

Le SDK peut être téléchargé sur [le site de LunarG](#) en utilisant les boutons en bas de page. Vous n'avez pas besoin de compte, mais celui-ci vous donne accès à une documentation supplémentaire qui pourra vous être utile.



La version MacOS du SDK utilise [MoltenVK](#). Il n'y a pas de support natif pour Vulkan sur MacOS, donc nous avons besoin de MoltenVK pour transcrire les appels aux fonctions Vulkan en appels au framework d'Apple : Metal. Grâce à cela, vous pouvez tirer avantage des performances et du débogage du framework Metal.

Une fois téléchargé, extrayez-en le contenu où vous le souhaitez (gardez en tête l'emplacement, car vous devez pointer ce dossier dans les projets Xcode). Dans le dossier extrait, le sous-dossier Applications comporte quelques exécutables de démonstration du SDK. Lancez vkcube pour vérifier que vous obtenez ceci :



III-C-2 - GLFW

Comme dit précédemment, Vulkan ignore la plateforme sur laquelle il opère, et n'inclut pas d'outil de création de fenêtres où afficher notre rendu. Pour bien exploiter les possibilités multiplateformes de Vulkan, nous utiliserons la **bibliothèque GLFW** pour créer une fenêtre qui supportera Windows, Linux et MacOS. Il existe d'autres bibliothèques telles que **SDL**, mais GLFW à l'avantage d'abstraire d'autres aspects spécifiques à la plateforme requis par Vulkan.

Pour installer GLFW, nous utiliserons le gestionnaire de package Homebrew. Le support de Vulkan sur MacOS n'étant pas parfaitement disponible (à l'écriture du moins) sur la version 3.2.1, nous installerons le package glfw3 ainsi :

```
brew install glfw3 --HEAD
```

III-C-3 - GLM

Vulkan n'inclut aucune bibliothèque pour l'algèbre linéaire, nous devons donc en télécharger une. **GLM** est une bonne bibliothèque souvent utilisée avec les bibliothèques graphiques dont OpenGL.

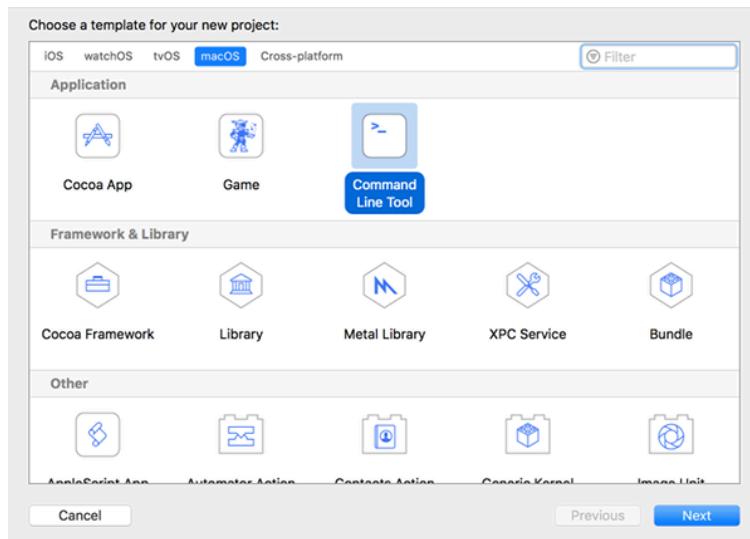
Cette bibliothèque est intégralement codée dans les fichiers d'en-têtes et se télécharge avec le paquet glm :

```
brew install glm
```

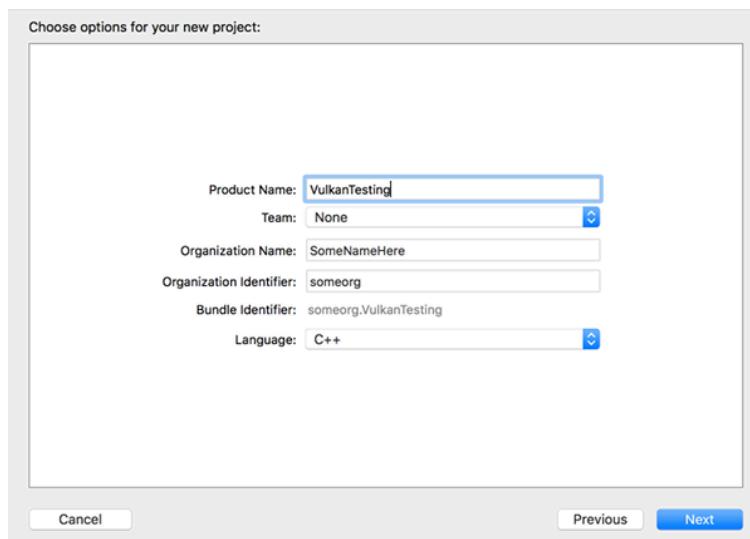
III-C-4 - Préparation de Xcode

Maintenant que nous avons toutes les dépendances, nous pouvons créer dans Xcode un projet Vulkan basique. La plupart des opérations seront de la « tuyauterie » pour lier les dépendances au projet. Notez que vous devrez remplacer toutes les mentions au dossier vulkansdk par le dossier où vous avez extrait le SDK Vulkan.

Lancez Xcode et créez un nouveau projet. Sur la fenêtre qui s'ouvre, sélectionnez Application → Command Line Tool.



Sélectionnez Next, inscrivez un nom de projet et choisissez C++ comme Language.



Appuyez sur Next et le projet devrait être créé. Copiez le code suivant à la place du code généré dans le fichier main.cpp :

```
#define GLFW_INCLUDE_VULKAN
#include <GLFW/glfw3.h>

#define GLM_FORCE_RADIANS
#define GLM_FORCE_DEPTH_ZERO_TO_ONE
#include <glm/vec4.hpp>
#include <glm/mat4x4.hpp>

#include <iostream>

int main() {
    glfwInit();

    glfwWindowHint(GLFW_CLIENT_API, GLFW_NO_API);
    GLFWwindow* window = glfwCreateWindow(800, 600, "Vulkan window", nullptr, nullptr);

    uint32_t extensionCount = 0;
    vkEnumerateInstanceExtensionProperties(nullptr, &extensionCount, nullptr);

    std::cout << extensionCount << " extensions supported\n";
}
```

```

glm::mat4 matrix;
glm::vec4 vec;
auto test = matrix * vec;

while (!glfwWindowShouldClose(window)) {
    glfwPollEvents();
}

glfwDestroyWindow(window);

glfwTerminate();

return 0;
}

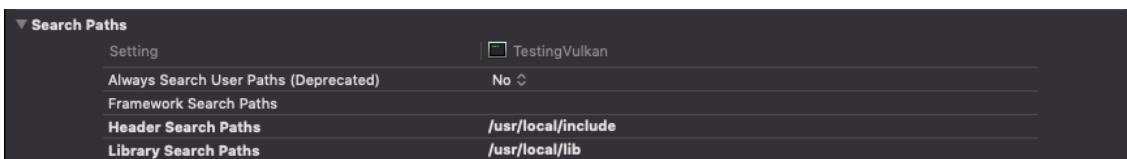
```

Gardez à l'esprit que vous n'avez pas à comprendre ce code pour le moment, sachant qu'il se contente d'appeler quelques fonctions de la bibliothèque pour s'assurer que tout fonctionne.

Xcode devrait déjà vous afficher des erreurs à propos de bibliothèques introuvables. Nous allons maintenant configurer le projet pour les faire disparaître. Sélectionnez votre projet sur le menu *Project Navigator*. Ouvrez l'onglet *Build Settings* puis :

- trouvez le champ **Header Search Paths** et ajoutez le chemin /usr/local/include (c'est ici que Homebrew installe les fichiers d'en-têtes) et le chemin vulkansdk/macOS/include pour le SDK ;
- trouvez le champ **Library Search Paths** et ajoutez le chemin /usr/local/lib (encore une fois, c'est le dossier où Homebrew installe les bibliothèques) et le chemin vulkansdk/macOS/lib.

Vous avez normalement (avec des différences évidentes selon l'endroit où vous avez placé votre SDK) :

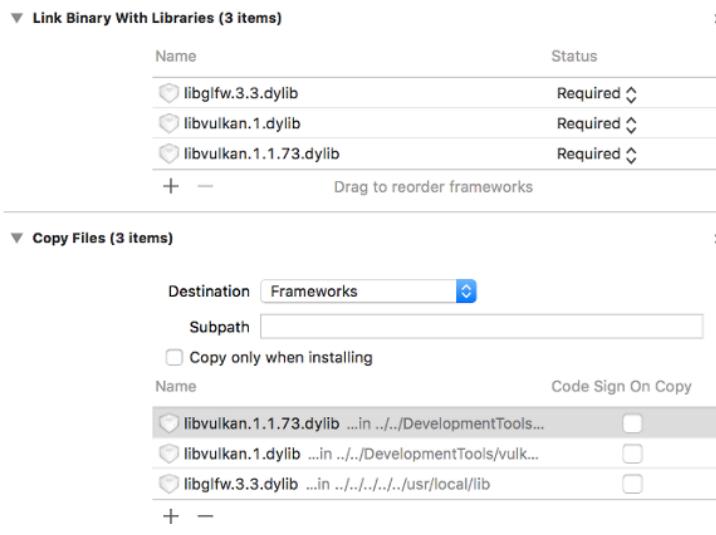


Maintenant, dans l'onglet *Build Phases*, sur l'entrée **Link Binary With Libraries** ajoutez les frameworks glfw3 et vulkan. Pour nous simplifier les choses, nous allons ajouter les bibliothèques dynamiques directement dans le projet (référez-vous à la documentation de ces bibliothèques si vous voulez les lier de manière statique).

- Pour GLFW ouvrez le dossier /usr/local/lib où vous trouverez un fichier avec un nom comme libglfw.3.x.dylib (où x est le numéro de la version. Il peut être différent chez vous). Glissez ce fichier jusqu'à l'onglet « Linked Frameworks and Librairies » de Xcode.
- Pour Vulkan, rendez-vous dans le dossier vulkansdk/macOS/lib et répétez l'opération pour libvulkan.1.dylib et libvulkan.1.x.xx.dylib (où les x correspondent à la version du SDK que vous avez téléchargé).

Maintenant que vous avez ajouté ces bibliothèques, remplissez le champ Destination avec « Frameworks » de l'onglet **Copy Files**, supprimez le sous-chemin et décochez « Copy only when installing ». Cliquez sur le « + » et ajoutez-y les trois frameworks.

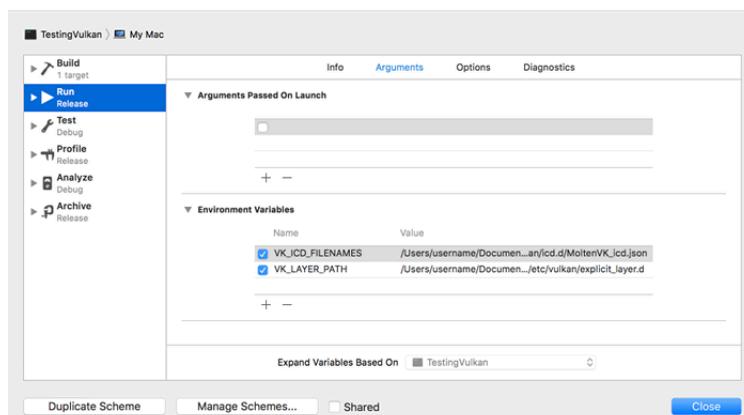
Votre configuration Xcode devrait ressembler à cela :



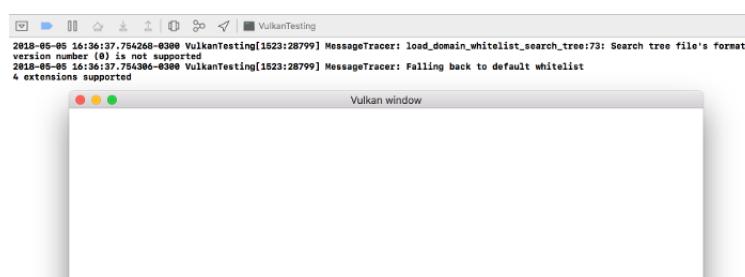
Il ne reste plus qu'à définir quelques variables d'environnement. Sur la barre d'outils de Xcode allez à Product → Scheme → Edit Scheme..., et dans l'onglet Arguments ajoutez les deux variables suivantes :

- `VK_ICD_FILERAMES = vulkansdk/macOS/etc/vulkan/icd.d/MoltenVK_icd.json` ;
- `VK_LAYER_PATH = vulkansdk/macOS/etc/vulkan/explicit_layer.d`.

Vous avez normalement ceci :



Vous êtes maintenant prêt ! Si vous lancez le projet (en pensant à bien choisir Debug ou Release) vous devrez avoir ceci :



Le nombre d'extensions doit être non nul. Les autres données proviennent de librairies et dépendent de votre configuration.

Vous êtes maintenant prêt à vous **lancer avec Vulkan!**

IV - Dessiner un triangle

IV-A - Mise en place

IV-A-1 - Code de base

IV-A-1-a - Structure générale

Dans le chapitre précédent, nous avons créé un projet Vulkan fonctionnel et nous l'avons testé à l'aide d'un code simple. Nous repartons de zéro, à partir du code suivant :

```
#include <vulkan/vulkan.h>

#include <iostream>
#include <stdexcept>
#include <cstdlib>

class HelloTriangleApplication {
public:
    void run() {
        initVulkan();
        mainLoop();
        cleanup();
    }

private:
    void initVulkan() {

    }

    void mainLoop() {

    }

    void cleanup() {

    };
};

int main() {
    HelloTriangleApplication app;

    try {
        app.run();
    } catch (const std::exception& e) {
        std::cerr << e.what() << std::endl;
        return EXIT_FAILURE;
    }

    return EXIT_SUCCESS;
}
```

Nous incluons d'abord le fichier d'en-tête Vulkan du SDK de LunarG, qui fournit les fonctions, les structures et les énumérations propres à la bibliothèque. Les fichiers d'en-tête `<stdexcept>` et `<iostream>` nous permettront de reporter et de traiter les erreurs. Le fichier d'en-tête `<cstdlib>` nous fournit les macros `EXIT_FAILURE` et `EXIT_SUCCESS`.

Le programme est écrit à l'intérieur d'une classe, dans laquelle seront stockés, comme membres privés, les objets Vulkan. La classe contiendra ainsi une fonction pour les initialiser, que nous appellerons `initVulkan`. Une fois l'initialisation réalisée, nous entrons dans la boucle principale, qui attend que nous fermions la fenêtre pour quitter le

programme. Une fois la fenêtre fermée et que le programme quitte la fonction `mainLoop()`, nous nous assurons de libérer les ressources que nous avons utilisées avec la fonction `cleanup()`.

Si nous rencontrons une quelconque erreur lors de l'exécution nous lèverons l'exception `std::runtime_error` avec un message descriptif, qui sera affiché sur le terminal depuis la fonction `main()`. Afin de nous assurer que nous récupérons bien toutes les erreurs, nous utilisons `std::exception` dans le `catch`. Nous verrons bientôt que la requête de certaines extensions peut mener à lever des exceptions.

À peu près tous les chapitres à partir de celui-ci présenteront une nouvelle fonction qui sera appelée dans `initVulkan()` ainsi qu'un ou plusieurs objets Vulkan, ajoutés comme membres privés de la classe et qui devront être détruits dans la fonction `cleanup()`.

IV-A-1-b - Gestion des ressources

De la même façon qu'une quelconque ressource explicitement allouée par `malloc` doit être explicitement libérée par `free`, nous devrons explicitement détruire toutes les ressources Vulkan que nous allouerons. En C++, il est possible d'automatiser cela grâce au RAII (Resource Acquisition Is Initialization) ou aux pointeurs intelligents disponibles dans le fichier d'en-tête `<memory>`. Toutefois, dans ce tutoriel, nous resterons explicites pour toutes les opérations d'allocation et de libération des objets Vulkan. Après tout, un des objectifs de Vulkan est de rendre toute opération explicite afin d'éviter des erreurs. En laissant explicite la durée de vie des objets, il devient plus facile d'apprendre le fonctionnement de la bibliothèque.

Après avoir suivi ce tutoriel, vous pourrez parfaitement implémenter une gestion automatique des ressources en écrivant des classes C++ pour lesquelles le constructeur alloue l'objet Vulkan et le destructeur le libère. Aussi, vous pouvez fournir un opérateur de destruction personnalisé aux instances de `std::unique_ptr` et de `std::shared_ptr`. L'utilisation du RAII est à privilégier pour les projets Vulkan plus imposants, mais dans un objectif d'apprentissage, il est meilleur de savoir ce qui se passe en coulisse.

Les objets Vulkan peuvent être créés de deux manières. Soit ils sont directement créés avec une fonction du type `vkCreateXXX`, soit ils sont alloués à l'aide d'un autre objet avec une fonction `vkAllocateXXX`. Après vous être assuré que l'objet n'est plus utilisé, il faut le détruire en utilisant les fonctions `vkDestroyXXX` ou `vkFreeXXX`, respectivement. Les paramètres de ces fonctions varient suivant le type d'objet. Par contre, elles ont un paramètre commun : `pAllocator`. Ce paramètre optionnel vous permet de spécifier une fonction de callback permettant au programmeur de gérer lui-même l'allocation mémoire. Nous n'utiliserons jamais ce paramètre et indiquerons donc toujours `nullptr`.

IV-A-1-c - Intégrer GLFW

Vulkan fonctionne très bien sans fenêtre, notamment pour une utilisation de rendu hors écran (`offscreen`), mais c'est tout de même plus intéressant d'afficher quelque chose ! Remplacez d'abord la ligne `#include <vulkan/vulkan.h>` par :

```
#define GLFW_INCLUDE_VULKAN
#include <GLFW/glfw3.h>
```

GLFW va alors inclure ses propres définitions et automatiquement charger le fichier d'en-tête de Vulkan. Ajoutez une fonction `initWindow()` et appelez-la depuis la fonction `run()` avant les autres fonctions. Nous utiliserons cette fonction pour initialiser GLFW et créer une fenêtre.

```
void run() {
    initWindow();
    initVulkan();
    mainLoop();
    cleanup();
}

private:
    void initWindow() {
```

{}

Le premier appel dans `initWindow()` doit être `glfwInit()`, pour initialiser la bibliothèque. Dans la mesure où GLFW a été créée pour fonctionner avec OpenGL, nous devons lui demander de ne pas créer de contexte OpenGL avec l'appel suivant :

```
glfwWindowHint(GLFW_CLIENT_API, GLFW_NO_API);
```

Le redimensionnement d'une fenêtre demande des précautions particulières, nous verrons cela plus tard et l'interdisons pour l'instant :

```
glfwWindowHint(GLFW_RESIZABLE, GLFW_FALSE);
```

Il ne nous reste plus qu'à créer la fenêtre. Ajoutez un membre privé `GLFWWindow* m_window` pour en stocker une référence, et initialisez-la ainsi :

```
window = glfwCreateWindow(800, 600, "Vulkan", nullptr, nullptr);
```

Les trois premiers paramètres indiquent respectivement la largeur, la hauteur et le titre de la fenêtre. Le quatrième vous permet optionnellement de spécifier un moniteur sur lequel ouvrir la fenêtre et le cinquième est spécifique à OpenGL.

Nous devrions plutôt utiliser des constantes pour la hauteur et la largeur dans la mesure où nous aurons besoin de ces valeurs dans le futur. J'ai donc ajouté au-dessus de la définition de la classe `HelloTriangleApplication` ces définitions :

```
const uint32_t WIDTH = 800;
const uint32_t HEIGHT = 600;
```

et remplacez la création de la fenêtre par :

```
window = glfwCreateWindow(WIDTH, HEIGHT, "Vulkan", nullptr, nullptr);
```

Vous avez maintenant une fonction `initWindow()` ressemblant à ceci :

```
void initWindow() {
    glfwInit();

    glfwWindowHint(GLFW_CLIENT_API, GLFW_NO_API);
    glfwWindowHint(GLFW_RESIZABLE, GLFW_FALSE);

    window = glfwCreateWindow(WIDTH, HEIGHT, "Vulkan", nullptr, nullptr);
}
```

Pour nous assurer que l'application tourne jusqu'à ce qu'une erreur ou un clic sur la croix ne l'interrompe, nous devons écrire une petite boucle de gestion d'événements :

```
void mainLoop() {
    while (!glfwWindowShouldClose(window)) {
        glfwPollEvents();
    }
}
```

Ce code est relativement simple. GLFW récupère tous les événements disponibles, puis vérifie qu'aucun d'entre eux ne correspond à une demande de fermeture de fenêtre. Ce sera aussi ici que nous appellerons la fonction qui affichera un triangle.

Une fois la fenêtre fermée, nous devons détruire toutes les ressources allouées et quitter GLFW. Voici la première version de la fonction `cleanup` :

```
void cleanup() {
    glfwDestroyWindow(window);

    glfwTerminate();
}
```

Si vous lancez l'application, vous devriez voir une fenêtre nommée « Vulkan » qui se ferme en cliquant sur la croix. Maintenant que nous avons une base pour notre application Vulkan, **créons notre premier objet Vulkan !**

Code C++

IV-A-2 - Instance

IV-A-2-a - Crédation d'une instance Vulkan

La première chose à faire est d'initialiser Vulkan à travers une *instance*. Cette instance permet de faire le lien entre l'application et la bibliothèque. Pour la créer, il est nécessaire de fournir quelques informations au pilote concernant l'application.

Créez une fonction `createInstance()` et appelez-la depuis la fonction `initVulkan()` :

```
void initVulkan() {
    createInstance();
}
```

Ajoutez ensuite un membre à la classe pour conserver cette instance :

```
private:
VkInstance instance;
```

Pour créer l'instance, nous devons d'abord remplir une structure avec des informations sur notre application. Techniquement, ces données sont optionnelles, mais elles peuvent fournir des informations utiles au pilote permettant d'optimiser des applications spécifiques (par exemple, lorsque l'application utilise un moteur très connu avec un comportement précis). Cette structure s'appelle `VkApplicationInfo` :

```
void createInstance() {
    VkApplicationInfo appInfo{};
    appInfo.sType = VK_STRUCTURE_TYPE_APPLICATION_INFO;
    appInfo.pApplicationName = "Hello Triangle";
    appInfo.applicationVersion = VK_MAKE_VERSION(1, 0, 0);
    appInfo.pEngineName = "No Engine";
    appInfo.engineVersion = VK_MAKE_VERSION(1, 0, 0);
    appInfo.apiVersion = VK_API_VERSION_1_0;
}
```

Comme mentionné précédemment, la plupart des structures Vulkan vous demandent d'expliciter leur propre type dans le champ `sType`. Aussi, cette structure fait partie de celles ayant un champ `pNext` pouvant pointer vers des informations supplémentaires. Ce champ est laissé à sa valeur par défaut : `nullptr`.

De nombreuses informations sont passées à Vulkan par le biais de structure et non pas de paramètre de fonctions. Nous devons remplir une autre structure pour compléter les informations nécessaires. Cette seconde structure est obligatoire et permet d'indiquer au pilote quelles sont les extensions globales et les couches de validation dont nous avons besoin. Le terme « *global* » indique qu'elles s'appliquent à l'intégralité du programme et non pas à un périphérique particulier. Ce concept sera plus clair dans les prochains chapitres.

```
VkInstanceCreateInfo createInfo{};
createInfo.sType = VK_STRUCTURE_TYPE_INSTANCE_CREATE_INFO;
createInfo.pApplicationInfo = &appInfo;
```

Les deux premiers paramètres sont simples. Les deux suivants spécifient les extensions globales dont nous avons besoin. Comme nous l'avons vu dans l'introduction, Vulkan n'a aucune connaissance de la plateforme sur laquelle il travaille et nous aurons donc besoin d'une extension pour s'interfacer avec le gestionnaire de fenêtres. GLFW possède une fonction très pratique qui nous donne la liste des extensions dont nous avons besoin pour cela :

```
uint32_t glfwExtensionCount = 0;
const char** glfwExtensions;

glfwExtensions = glfwGetRequiredInstanceExtensions(&glfwExtensionCount);

createInfo.enabledExtensionCount = glfwExtensionCount;
createInfo.pEnabledExtensionNames = glfwExtensions;
```

Les deux derniers membres de la structure indiquent les couches de validation à activer. Nous verrons cela dans le prochain chapitre, laissez ces champs vides pour le moment :

```
createInfo.enabledLayerCount = 0;
```

Nous avons maintenant indiqué tout ce dont Vulkan a besoin pour créer une instance. Nous pouvons enfin appeler **vkCreateInstance()** :

```
VkResult result = vkCreateInstance(&createInfo, nullptr, &instance);
```

Comme vous allez le constater, les fonctions de création d'un objet suivent un motif récurrent :

- un pointeur sur une structure contenant les informations de création de l'objet ;
- un pointeur sur une fonction d'allocation que nous laisserons toujours à `nullptr` ;
- un pointeur sur une variable pour stocker le nouvel objet.

Si tout s'est bien passé, l'instance devrait être stockée dans le membre **VkInstance** de la classe. Quasiment toutes les fonctions Vulkan retournent une valeur du type **VkResult**, pouvant être soit `VK_SUCCESS` soit un code d'erreur. Nul besoin de stocker le résultat, nous pouvons directement vérifier la valeur renournée ainsi :

```
if (vkCreateInstance(&createInfo, nullptr, &instance) != VK_SUCCESS) {
    throw std::runtime_error("failed to create instance!");
}
```

Lancez votre programme pour voir si l'instance s'est créée correctement.

IV-A-2-b - Vérification du support des extensions

Si vous regardez la documentation de la fonction **vkCreateInstance()** vous pourrez voir que l'un des messages d'erreur possible est `VK_ERROR_EXTENSION_NOT_PRESENT`. Nous pourrions juste interrompre le programme et afficher une erreur si une extension dont nous avons besoin est absente. Ce serait logique pour la plupart des extensions telles que celles pour le gestionnaire de fenêtres, mais quid du cas d'une extension optionnelle ?

La fonction **vkEnumerateInstanceExtensionProperties()** permet de récupérer la totalité des extensions supportées par le système avant la création de l'instance. Elle nécessite un pointeur vers une variable stockant le nombre d'extensions supportées et un tableau de **VkExtensionProperties** pour stocker les informations sur les extensions. Aussi, son premier paramètre est optionnel et permet de filtrer les extensions suivant une couche de validation spécifique. Nous l'ignorons pour le moment.

Pour allouer un tableau contenant les détails des extensions, nous devons déjà connaître le nombre de ces extensions. Vous pouvez récupérer celui-ci en définissant le dernier paramètre à `nullptr` :

```
uint32_t extensionCount = 0;
vkEnumerateInstanceExtensionProperties(nullptr, &extensionCount, nullptr);
```

Maintenant, il suffit d'allouer un tableau pour contenir les informations sur les extensions (il faut aussi ajouter #include <vector> pour avoir accès aux std::vector) :

```
std::vector<VkExtensionProperties> extensions(extensionCount);
```

Nous pouvons désormais récupérer les informations sur les extensions :

```
vkEnumerateInstanceExtensionProperties(nullptr, &extensionCount, extensions.data());
```

La structure **VkExtensionProperties** contient le nom et la version de l'extension. Nous pouvons les afficher à l'aide d'une simple boucle (\t représente une tabulation) :

```
std::cout << "available extensions:\n";
for (const auto& extension : extensions) {
    std::cout << '\t' << extension.extensionName << '\n';
}
```

Vous pouvez ajouter ce code dans la fonction `createInstance()` si vous voulez afficher les informations liées au support de Vulkan sur la machine.



Petit défi : programmez une fonction qui vérifie si les extensions rentrées par la fonction `glfwGetRequiredInstanceExtensions()` sont disponibles sur la machine.

IV-A-2-c - Libération des ressources

L'objet de type **VkInstance** ne doit être détruit qu'à la fin du programme. L'instance peut être libérée dans la fonction `cleanup()` grâce à la fonction **vkDestroyInstance()** :

```
void cleanup() {
    vkDestroyInstance(instance, nullptr);

    glfwDestroyWindow(window);

    glfwTerminate();
}
```

Les paramètres de cette fonction sont évidents. Comme préciser précédemment, les fonctions d'allocation et de désallocation possèdent un paramètre optionnel pour spécifier une fonction callback de gestion de la mémoire. Pour ignorer cela, nous passons `nullptr`. Toutes les autres ressources Vulkan que nous allons créer dans les chapitres suivants devront être libérées avant la destruction de l'instance Vulkan.

Avant de continuer sur des notions plus complexes, il est pratique de mettre en place des mécanismes de débogage grâce **aux couches de validation**.

Code C++

IV-A-3 - Couches de validation

IV-A-3-a - Qu'est-ce que les couches de validation ?

La bibliothèque Vulkan est conçue pour limiter au maximum le travail du pilote graphique. Par conséquent, par défaut, il n'y a aucune gestion des erreurs. Une erreur aussi simple que se tromper dans la valeur d'une énumération ou passer un pointeur nul comme argument non optionnel résulte en un crash ou un comportement indéfini. Dans la

mesure où Vulkan nous demande d'être complètement explicite sur ce que nous faisons, il est facile de faire des petites erreurs comme utiliser une nouvelle fonctionnalité du GPU et ne pas l'avoir demandée lors de la création du périphérique logique.

Cependant de telles vérifications peuvent être ajoutées à la bibliothèque. Vulkan possède un système élégant appelé couches de validation. Ce sont des composants optionnels s'insérant dans les appels des fonctions Vulkan pour y ajouter des opérations. Voici quelques exemples de ce que peut apporter une couche de validation :

- comparer les valeurs des paramètres à celles de la spécification pour détecter une mauvaise utilisation ;
- suivre la création et la destruction des objets pour repérer les fuites de mémoire ;
- vérifier la sécurité des threads en suivant l'origine des appels ;
- afficher tous les appels de fonctions et leurs paramètres sur la sortie standard ;
- tracer les appels Vulkan pour une analyse de performances ou pour les rejouer ultérieurement.

Voici une implémentation d'exemple d'une fonction fournie par une couche de validation à des fins de diagnostic :

```
VkResult vkCreateInstance(
    const VkInstanceCreateInfo* pCreateInfo,
    const VkAllocationCallbacks* pAllocator,
    VkInstance* instance) {

    if (pCreateInfo == nullptr || instance == nullptr) {
        log("Pointeur nul passé à un paramètre obligatoire !");
        return VK_ERROR_INITIALIZATION_FAILED;
    }

    return real_vkCreateInstance(pCreateInfo, pAllocator, instance);
}
```

Les couches de validations peuvent être combinées à loisir pour fournir toutes les fonctionnalités de débogage nécessaires. Vous pouvez activer les couches de validation lors du développement et complètement les désactiver lors du déploiement. En bref, le meilleur des deux mondes !

Vulkan ne fournit aucune couche de validation. Par contre, le SDK de LunarG fournit un ensemble de couches couvrant les erreurs courantes. Elles sont complètement **open source**, vous pouvez donc voir quelles erreurs sont détectées et contribuer à leur développement. Les utiliser est la meilleure manière de s'assurer que l'application fonctionne sur tous les pilotes et qu'elle ne repose pas sur un comportement indéfini.

Les couches de validation ne sont utilisables que si elles sont installées sur la machine. Par exemple, les couches de validation de LunarG ne sont disponibles que sur les PC ayant le SDK Vulkan installé.

Il a existé deux formes de couches de validation : les couches spécifiques à l'instance et celles spécifiques au périphérique. L'objectif était que les couches spécifiques à l'instance ne vérifient que les appels liés aux objets globaux de Vulkan (par exemple les instances) et que les couches spécifiques au périphérique ne vérifient que les appels liés au GPU. Les couches spécifiques au périphérique sont maintenant dépréciées. Les autres portent désormais sur tous les appels à Vulkan. Cependant la spécification recommande encore que nous activions les couches de validation au niveau du périphérique pour des raisons de compatibilité : certaines implémentations peuvent reposer sur ce comportement. Nous nous contenterons de spécifier les mêmes couches pour le périphérique logique que pour l'instance comme nous le verrons **plus tard**.

IV-A-3-b - Utiliser les couches de validation

Nous allons maintenant activer les couches de validation fournies par le SDK Vulkan. Comme pour les extensions, les couches de validation doivent être activées à partir de leur nom. Toutes les couches de validation standards sont rassemblées dans une couche fournie par le SDK sous le nom de `VK_LAYER_KHRONOS_validation`.

Premièrement, nous ajoutons au programme, deux variables de configuration pour indiquer les couches à activer et si elles doivent être activées ou non. La valeur de cette dernière repose sur la façon dont est compilé le programme.

La macro `NDEBUG` faisant partie du standard C++ indique que le programme n'est pas compilé pour le débogage (No Debug).

```
const uint32_t WIDTH = 800;
const uint32_t HEIGHT = 600;

const std::vector<const char*> validationLayers = {
    "VK_LAYER_KHRONOS_validation"
};

#ifndef NDEBUG
    constexpr bool enableValidationLayers = false;
#else
    constexpr bool enableValidationLayers = true;
#endif
```

Ajoutons une nouvelle fonction `checkValidationLayerSupport()`, qui devra vérifier si toutes les couches que nous voulons utiliser sont disponibles. D'abord, la fonction liste les couches de validation disponibles à l'aide de la fonction `vkEnumerateInstanceLayerProperties()`. Elle s'utilise de la même façon que `vkEnumerateInstanceExtensionProperties()`, que nous avons vue dans le chapitre précédent.

```
bool checkValidationLayerSupport() {
    uint32_t layerCount;
    vkEnumerateInstanceLayerProperties(&layerCount, nullptr);

    std::vector<VkLayerProperties> availableLayers(layerCount);
    vkEnumerateInstanceLayerProperties(&layerCount, availableLayers.data());

    return false;
}
```

Ensute, nous vérifions si toutes les couches indiquées par `validationLayers` sont présentes dans la liste des couches disponibles sur la machine. Vous aurez besoin de `<cstring>` pour la fonction `strcmp()`.

```
for (const char* layerName : validationLayers) {
    bool layerFound = false;

    for (const auto& layerProperties : availableLayers) {
        if (strcmp(layerName, layerProperties.layerName) == 0) {
            layerFound = true;
            break;
        }
    }

    if (!layerFound) {
        return false;
    }
}

return true;
```

Nous pouvons maintenant utiliser cette fonction dans la fonction `createInstance()` :

```
void createInstance() {
    if (enableValidationLayers && !checkValidationLayerSupport()) {
        throw std::runtime_error("les couches de validation sont activées, mais ne sont pas
disponibles !");
    }

    ...
}
```

Lancez maintenant le programme en mode debug et assurez-vous qu'il n'y a pas d'erreur. Si vous obtenez une erreur, [référez-vous à la FAQ](#).

Finalement, modifions la structure **VkCreateInstanceCreateInfo()** pour inclure les noms des couches de validations à utiliser, si elles doivent être activées :

```

if (enableValidationLayers) {
    createInfo.enabledLayerCount = static_cast<uint32_t>(validationLayers.size());
    createInfo.ppEnabledLayerNames = validationLayers.data();
} else {
    createInfo.enabledLayerCount = 0;
}
    
```

Si la vérification précédente s'est bien passée, la fonction **vkCreateInstance()** ne devrait jamais retourner **VK_ERROR_LAYER_NOT_PRESENT**, mais exécutez tout de même le programme pour en être sûr.

IV-A-3-c - Gestion personnalisée des messages

Par défaut, les couches de validation affichent leur message dans la console, mais on peut aussi s'occuper de l'affichage nous-mêmes en fournissant une fonction de callback explicite dans notre programme. Ceci nous permet également de choisir quels types de messages afficher, car tous ne sont pas des erreurs (fatales). Si vous ne voulez pas vous occuper de ça maintenant, vous pouvez sauter à la dernière section de ce chapitre.

Pour configurer un callback permettant de s'occuper des messages et des détails associés, nous devons mettre en place un messager de débogage (debug messenger) avec un callback en utilisant l'extension **VK_EXT_DEBUG_UTILS**.

Créons d'abord une fonction **getRequiredExtensions()**. Elle nous fournira une liste des extensions nécessaires selon que nous activons les couches de validation ou non :

```

std::vector<const char*> getRequiredExtensions() {
    uint32_t glfwExtensionCount = 0;
    const char** glfwExtensions;
    glfwExtensions = glfwGetRequiredInstanceExtensions(&glfwExtensionCount);

    std::vector<const char*> extensions(glfwExtensions, glfwExtensions + glfwExtensionCount);

    if (enableValidationLayers) {
        extensions.push_back(VK_EXT_DEBUG_UTILS_EXTENSION_NAME);
    }

    return extensions;
}
    
```

Les extensions spécifiées par GLFW seront toujours nécessaires, mais celle pour le débogage n'est ajoutée que conditionnellement. Notez l'utilisation de la macro **VK_EXT_DEBUG_UTILS_EXTENSION_NAME** au lieu du nom de l'extension pour éviter les erreurs de frappe.

Nous pouvons maintenant utiliser cette fonction dans la fonction **createInstance()** :

```

auto extensions = getRequiredExtensions();
createInfo.enabledExtensionCount = static_cast<uint32_t>(extensions.size());
createInfo.ppEnabledExtensionNames = extensions.data();
    
```

Exécutez le programme et assurez-vous que vous ne recevez pas l'erreur **VK_ERROR_EXTENSION_NOT_PRESENT**. Nous ne devrions pas avoir besoin de vérifier sa présence sachant qu'elle est liée à la présence des couches de validation.

Intéressons-nous maintenant à la fonction de callback. Ajoutez une nouvelle fonction statique nommée **debugCallback()** à votre classe ayant pour prototype **PFN_vkDebugUtilsMessengerCallbackEXT**. **VKAPI_ATTR** et **VKAPI_CALL** assurent que la signature est la bonne pour que Vulkan puisse l'appeler.

```

static VKAPI_ATTR VkBool32 VKAPI_CALL debugCallback(
    
```

```

VkDebugUtilsMessageSeverityFlagBitsEXT messageSeverity,
VkDebugUtilsMessageTypeFlagsEXT messageType,
const VkDebugUtilsMessengerCallbackDataEXT* pCallbackData,
void* pUserData) {

    std::cerr << "couche de validation : " << pCallbackData->pMessage << std::endl;

    return VK_FALSE;
}
    
```

Le premier paramètre indique la严重性 du message. Cela peut être :

- `VK_DEBUG_UTILS_MESSAGE_SEVERITY_VERBOSE_BIT_EXT` : message de diagnostic ;
- `VK_DEBUG_UTILS_MESSAGE_SEVERITY_INFO_BIT_EXT` : message d'information, telle que la création d'une ressource ;
- `VK_DEBUG_UTILS_MESSAGE_SEVERITY_WARNING_BIT_EXT` : message relevant un comportement qui n'est pas un bogue, mais qui pourrait en être un ;
- `VK_DEBUG_UTILS_MESSAGE_SEVERITY_ERROR_BIT_EXT` : message relevant un comportement invalide pouvant mener à un crash.

Les valeurs de cette énumération sont pensées de telle sorte qu'il est possible d'utiliser une comparaison pour savoir si un message est égal ou plus critique qu'un niveau de严重性 donné :

```

if (messageSeverity >= VK_DEBUG_UTILS_MESSAGE_SEVERITY_WARNING_BIT_EXT) {
    // Le message est suffisamment important pour être affiché
}
    
```

Le paramètre `messageType` peut prendre les valeurs suivantes :

- `VK_DEBUG_UTILS_MESSAGE_TYPE_GENERAL_BIT_EXT` : un événement sans rapport avec les performances ou la spécification ;
- `VK_DEBUG_UTILS_MESSAGE_TYPE_VALIDATION_BIT_EXT` : un événement provoquant une violation de la spécification et qui indique une erreur probable ;
- `VK_DEBUG_UTILS_MESSAGE_TYPE_PERFORMANCE_BIT_EXT` : une utilisation non optimale de Vulkan.

Le paramètre `pCallbackData` est une structure du type `VkDebugUtilsMessengerCallbackDataEXT` contenant les détails du message. Ses membres les plus importants sont :

- `pMessage` : le message sous la forme d'une chaîne de caractères terminée avec le caractère nul ;
- `pObjects` : un tableau d'objets Vulkan en rapport avec le message ;
- `objectCount` : le nombre d'objets dans le tableau.

Finalement, le paramètre `pUserData` est un pointeur sur une donnée quelconque que vous pouvez spécifier à la création du callback.

Le callback que nous programmons retourne un booléen déterminant si la fonction à l'origine de son appel doit être interrompue. Si elle retourne `VK_TRUE`, l'exécution de la fonction est interrompue et cette dernière retourne `VK_ERROR_VALIDATION_FAILED_EXT`. Cette fonctionnalité n'est normalement utilisée que pour tester les couches de validation elles-mêmes. Par conséquent, nous retournerons toujours `VK_FALSE`.

Il ne nous reste plus qu'à fournir notre fonction à Vulkan. Surprenamment, même le mécanisme de débogage se gère à travers une référence de type `VkDebugUtilsMessengerEXT`, que nous devrons explicitement créer et détruire. Le callback fait partie d'un messager de débogage et vous pouvez en posséder autant que vous le désirez. Ajoutez un membre à la classe pour le messager en dessous de l'instance :

```

VkDebugUtilsMessengerEXT callback;
    
```

Ajoutez ensuite une fonction `setupDebugMessenger()` et appelez-la dans `initVulkan()` après `createInstance()` :

```

void initVulkan() {
    createInstance();
    setupDebugMessenger();
}

void setupDebugMessenger() {
    if (!enableValidationLayers) return;

}

```

Nous devons maintenant remplir une structure avec des informations sur le messager et sa fonction de callback :

```

VkDebugUtilsMessengerCreateInfoEXT createInfo{};
createInfo.sType = VK_STRUCTURE_TYPE_DEBUG_UTILS_MESSENGER_CREATE_INFO_EXT;
createInfo.messageSeverity = VK_DEBUG_UTILS_MESSAGE_SEVERITY_VERBOSE_BIT_EXT | VK_DEBUG_UTILS_MESSAGE_SEVERITY_WARNING_BIT_EXT | VK_DEBUG_UTILS_MESSAGE_SEVERITY_ERROR_BIT_EXT;
createInfo.messageType = VK_DEBUG_UTILS_MESSAGE_TYPE_GENERAL_BIT_EXT | VK_DEBUG_UTILS_MESSAGE_TYPE_VALIDATION_BIT_EXT | VK_DEBUG_UTILS_MESSAGE_TYPE_PERFORMANCE_BIT_EXT;
createInfo.pfnUserCallback = debugCallback;
createInfo.pUserData = nullptr; // Optionnel

```

Le champ `messageSeverity` vous permet de filtrer pour quelle criticité de message votre fonction sera appelée. J'ai laissé tous les types sauf `VK_DEBUG_UTILS_MESSAGE_SEVERITY_INFO_BIT_EXT`, ce qui permet de recevoir toutes les informations à propos de possibles bogues tout en éliminant les informations génériques de débogage.

De manière similaire, le champ `messageType` vous permet de filtrer les types de messages pour lesquels la fonction de rappel sera appelée. J'y ai mis tous les types possibles. Vous pouvez très bien en désactiver s'ils ne vous servent à rien.

Finalement, le champ `pfnUserCallback` indique le pointeur vers la fonction de rappel. Optionnellement, vous pouvez passer un pointeur au champ `pUserData`, qui sera par la suite passé à la fonction de rappel au travers du paramètre `pUserData`. Par exemple, vous pouvez utiliser ce mécanisme pour passer un pointeur sur la classe `HelloTriangleApplication`.

Notez qu'il existe de nombreuses autres manières de configurer les messages des couches de validation et des fonctions de débogage. La configuration proposée ici offre une bonne base pour ce tutoriel. Référez-vous à la **spécification de l'extension** pour plus d'informations sur ces possibilités.

Cette structure doit maintenant être passée à la fonction `vkCreateDebugUtilsMessengerEXT` afin de créer l'objet `VkDebugUtilsMessengerEXT`. Malheureusement cette fonction fait partie d'une extension qui n'est pas chargée automatiquement. Nous devons donc trouver son adresse nous-mêmes grâce à la fonction `vkGetInstanceProcAddr()`. Nous allons créer notre propre fonction gérant ce genre de cas en fond. Je l'ai ajoutée au-dessus de la définition de la classe `HelloTriangleApplication`.

```

VkResult CreateDebugUtilsMessengerEXT(VkInstance instance, const
    VkDebugUtilsMessengerCreateInfoEXT* pCreateInfo, const VkAllocationCallbacks* pAllocator,
    VkDebugUtilsMessengerEXT* pCallback) {
    auto func = (PFN_vkCreateDebugUtilsMessengerEXT)
        vkGetInstanceProcAddr(instance, "vkCreateDebugUtilsMessengerEXT");
    if (func != nullptr) {
        return func(instance, pCreateInfo, pAllocator, pCallback);
    } else {
        return VK_ERROR_EXTENSION_NOT_PRESENT;
    }
}

```

La fonction `vkGetInstanceProcAddr()` retourne `nullptr` si la fonction n'a pas pu être chargée. Nous pouvons maintenant utiliser cette fonction pour créer l'objet provenant de l'extension s'il est disponible :

```

if (CreateDebugUtilsMessengerEXT(instance, &createInfo, nullptr, &callback) != VK_SUCCESS) {
    throw std::runtime_error("le messager n'a pas pu être créé!");
}

```

}

Le troisième paramètre est l'invariable allocateur optionnel que nous laissons `nullptr`. Les autres paramètres sont assez logiques. La fonction `callback` est spécifique à l'instance et aux couches de validation, nous devons donc passer l'instance en premier argument. Vous allez croiser ce mécanisme pour d'autres objets « enfants ».

L'objet `VkDebugUtilsMessengerEXT` doit être libéré grâce à la fonction `vkDestroyDebugUtilsMessengerEXT`. De même, la fonction `vkCreateDebugUtilsMessengerEXT` doit être chargée manuellement.

Créez une autre fonction proxy juste en dessous de `CreateDebugUtilsMessengerEXT` :

```
void DestroyDebugUtilsMessengerEXT(VkInstance instance, VkDebugUtilsMessengerEXT
    debugMessenger, const VkAllocationCallbacks* pAllocator) {
    auto func = (PFN_vkDestroyDebugUtilsMessengerEXT)
        vkGetInstanceProcAddr(instance, "vkDestroyDebugUtilsMessengerEXT");
    if (func != nullptr) {
        func(instance, debugMessenger, pAllocator);
    }
}
```

Assurez-vous que cette fonction est statique ou hors de la classe. Ensuite, nous pouvons l'appeler dans la fonction `cleanup()`.

```
void cleanup() {
    if (enableValidationLayers) {
        DestroyDebugUtilsMessengerEXT(instance, callback, nullptr);
    }

    vkDestroyInstance(instance, nullptr);

    glfwDestroyWindow(window);

    glfwTerminate();
}
```

IV-A-3-d - Déboguer la création et la destruction de l'instance

Même si nous avons des fonctionnalités de débogage grâce aux couches de validation, nous ne couvrons pas tous les cas. L'appel à la fonction `vkCreateDebugUtilsMessengerEXT`, nécessite une instance valide alors que `vkDestroyDebugUtilsMessengerEXT` doit être appelée avant que l'instance ne soit détruite. Par conséquent, il est impossible de déboguer les problèmes liés aux fonctions `vkCreateInstance()` et `vkDestroyInstance()`.

En regardant en détail [la documentation](#), vous allez trouver une méthode pour créer un messager de débogage spécifique à ces deux appels. Cela nécessite de passer simplement un pointeur à la structure `VkDebugUtilsMessengerCreateInfoEXT` dans le champ `pNext` de `VkInstanceCreateInfo`. Premièrement, déplacez le remplissage des informations du message dans une fonction à part :

```
void populateDebugMessengerCreateInfo(VkDebugUtilsMessengerCreateInfoEXT& createInfo) {
    createInfo = {};
    createInfo.sType = VK_STRUCTURE_TYPE_DEBUG_UTILS_MESSENGER_CREATE_INFO_EXT;
    createInfo.messageSeverity = VK_DEBUG_UTILS_MESSAGE_SEVERITY_VERBOSE_BIT_EXT |
        VK_DEBUG_UTILS_MESSAGE_SEVERITY_WARNING_BIT_EXT | VK_DEBUG_UTILS_MESSAGE_SEVERITY_ERROR_BIT_EXT;
    createInfo.messageType = VK_DEBUG_UTILS_MESSAGE_TYPE_GENERAL_BIT_EXT |
        VK_DEBUG_UTILS_MESSAGE_TYPE_VALIDATION_BIT_EXT |
        VK_DEBUG_UTILS_MESSAGE_TYPE_PERFORMANCE_BIT_EXT;
    createInfo.pfnUserCallback = debugCallback;
}

...
void setupDebugMessenger() {
    if (!enableValidationLayers) return;
    VkDebugUtilsMessengerCreateInfoEXT createInfo;
    populateDebugMessengerCreateInfo(createInfo);
```

```

if (CreateDebugUtilsMessengerEXT(instance, &createInfo, nullptr, &debugMessenger) != VK_SUCCESS) {
    throw std::runtime_error("Impossible de mettre en place le message de débogage !");
}
}

```

Maintenant, nous pouvons réutiliser cette fonction dans la fonction `createInstance()` :

```

void createInstance() {
    ...

    VkInstanceCreateInfo createInfo{};
    createInfo.sType = VK_STRUCTURE_TYPE_INSTANCE_CREATE_INFO;
    createInfo.pApplicationInfo = &appInfo;

    ...

    VkDebugUtilsMessengerCreateInfoEXT debugCreateInfo;
    if (enableValidationLayers) {
        createInfo.enabledLayerCount = static_cast<uint32_t>(validationLayers.size());
        createInfo.ppEnabledLayerNames = validationLayers.data();
        populateDebugMessengerCreateInfo(debugCreateInfo);
        createInfo.pNext = (VkDebugUtilsMessengerCreateInfoEXT*) &debugCreateInfo;
    } else {
        createInfo.enabledLayerCount = 0;
    }

    createInfo.pNext = nullptr;
}

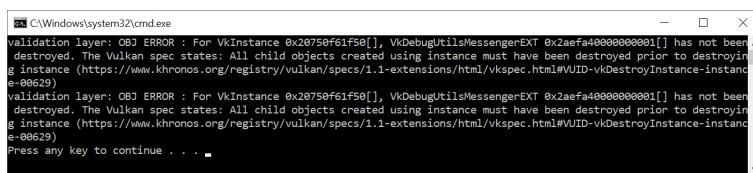
if (vkCreateInstance(&createInfo, nullptr, &instance) != VK_SUCCESS) {
    throw std::runtime_error("Échec de création de l'instance !");
}
}

```

La variable `debugCreateInfo` est en dehors du `if` pour qu'elle ne soit pas détruite avant l'appel à `vkCreateInstance()`. En créant le messager de débogage additionnel de cette façon, il sera appelé automatiquement durant les appels à `vkCreateInstance()` et `vkDestroyInstance()`. Il sera d'ailleurs détruit après ça.

IV-A-3-e - Test

Maintenant, plaçons intentionnellement une erreur pour voir les couches de validation en action. Temporairement, nous allons enlever l'appel à `DestroyDebugUtilsMessengerEXT` de la fonction `cleanup()` et exécuter notre programme. À la fermeture de celui-ci, vous devriez obtenir quelque chose de similaire :



 Si vous n'obtenez aucun message, vérifiez votre installation.

Si vous souhaitez voir quel appel a déclenché le message, vous pouvez ajouter un point d'arrêt dans la fonction de rappel et lire la liste d'appels.

IV-A-3-f - Configuration

Il y existe d'autres paramètres pour modifier le comportement des couches de validation que ceux que nous avons spécifiés dans la structure `VkDebugUtilsMessengerCreateInfoEXT`. Naviguez dans les dossiers du SDK Vulkan et

allez dans le dossier *Config*. Vous trouverez un fichier `vk_layer_settings.txt` qui explique comment configurer les couches.

Pour configurer les couches pour votre propre application, copiez le fichier dans les dossiers Debug et Release de votre projet et suivez les instructions pour définir le comportement voulu. Toutefois, dans la suite du tutoriel, je partirai du principe que vous les avez laissées avec leur comportement par défaut.

Tout au long du tutoriel je laisserai quelques petites erreurs intentionnelles pour vous montrer à quel point les couches de validation sont pratiques et à quel point il est important de comprendre tout ce que vous faites avec Vulkan. Il est maintenant temps de s'intéresser aux **périphériques Vulkan présent dans le système**.

Code C++

IV-A-4 - Périphériques physiques et famille de queues

IV-A-4-a - Sélection d'un périphérique physique

Après l'initialisation de Vulkan grâce à l'instance de type **VkInstance**, nous devons chercher et sélectionner une carte graphique présente sur le système qui supporte les fonctionnalités dont nous avons besoin. En fait, nous pouvons en sélectionner autant que nous voulons et les utiliser simultanément, mais nous n'utiliserons que la première carte convenant à notre besoin.

Nous devons ajouter une fonction `pickPhysicalDevice` et l'appeler dans la fonction `initVulkan()` :

```
void initVulkan() {
    createInstance();
    setupDebugMessenger();
    pickPhysicalDevice();
}

void pickPhysicalDevice() {
```

Nous stockerons la carte graphique sélectionnée dans une variable de type **VkPhysicalDevice** membre de la classe. Cet objet sera détruit implicitement lorsque la variable du type **VkInstance** sera détruite. Il n'y a donc pas à s'en soucier dans fonction `cleanup`.

```
VkPhysicalDevice physicalDevice = VK_NULL_HANDLE;
```

Obtenir la liste des cartes graphiques est similaire à l'obtention d'une liste des extensions. Cela débute par la récupération du nombre de périphériques présents :

```
uint32_t deviceCount = 0;
vkEnumeratePhysicalDevices(instance, &deviceCount, nullptr);
```

S'il n'y a aucun périphérique supportant Vulkan, il ne sert à rien de continuer :

```
if (deviceCount == 0) {
    throw std::runtime_error("Aucune carte graphique ne supporte Vulkan !");
}
```

Dans le cas contraire, nous pouvons allouer un tableau pour contenir les références aux **VkPhysicalDevice** :

```
std::vector<VkPhysicalDevice> devices(deviceCount);
vkEnumeratePhysicalDevices(instance, &deviceCount, devices.data());
```

Nous devons maintenant déterminer quel GPU correspond aux opérations que nous souhaitons effectuer. En effet, tous n'ont pas les mêmes capacités. Pour cela, nous créons une nouvelle fonction :

```
bool isDeviceSuitable(VkPhysicalDevice device) {
    return true;
}
```

Et nous allons vérifier si l'un des périphériques physiques correspond au besoin que nous spécifierons dans cette fonction.

```
for (const auto& device : devices) {
    if (isDeviceSuitable(device)) {
        physicalDevice = device;
        break;
    }
}

if (physicalDevice == VK_NULL_HANDLE) {
    throw std::runtime_error("Impossible de trouver un GPU adéquat !");
}
```

La section suivante introduira les premières contraintes à vérifier dans la fonction `isDeviceSuitable()`. Comme nous allons utiliser de plus en plus de fonctionnalités de Vulkan dans les prochains chapitres, nous allons ajouter de plus en plus de vérifications dans cette fonction.

IV-A-4-b - Vérification des fonctionnalités de base

Pour évaluer la compatibilité d'un périphérique, nous pouvons commencer par obtenir des informations dessus. Les propriétés telles que le nom, le type et la version supportée de Vulkan peuvent être obtenues grâce à la fonction `vkGetPhysicalDeviceProperties()`.

```
VkPhysicalDeviceProperties deviceProperties;
vkGetPhysicalDeviceProperties(device, &deviceProperties);
```

Le support des fonctionnalités optionnelles telles que la compression des textures, le support des nombres flottants sur 64 bits, le rendu sur plusieurs zones d'affichage (pour la réalité virtuelle) peuvent être obtenues grâce à la fonction `vkGetPhysicalDeviceFeatures()` :

```
VkPhysicalDeviceFeatures deviceFeatures;
vkGetPhysicalDeviceFeatures(device, &deviceFeatures);
```

Il est possible d'obtenir de nombreux autres détails sur les périphériques tels que la mémoire disponible ou les familles de queues.

Pour donner un exemple, nous allons considérer que notre application n'est utilisable que sur les cartes graphiques dédiées supportant les geometry shader. La fonction `isDeviceSuitable` ressemblera à cela :

```
bool isDeviceSuitable(VkPhysicalDevice device) {
    VkPhysicalDeviceProperties deviceProperties;
    VkPhysicalDeviceFeatures deviceFeatures;
    vkGetPhysicalDeviceProperties(device, &deviceProperties);
    vkGetPhysicalDeviceFeatures(device, &deviceFeatures);

    return deviceProperties.deviceType == VK_PHYSICAL_DEVICE_TYPE_DISCRETE_GPU &&
           deviceFeatures.geometryShader;
}
```

Au lieu de ne choisir que le premier périphérique adéquat, nous pourrions attribuer un score à chacun d'entre eux et utiliser celui dont le score est le plus élevé. Vous pourriez ainsi favoriser une carte graphique dédiée en lui donnant

un grand score, mais utiliser un GPU intégré au CPU si c'est le seul disponible. Cela pourrait être implémenté comme suit :

```
#include <map>

...
void pickPhysicalDevice() {
    ...

    // Utilise une map ordonnée pour trier automatiquement sur le score
    std::multimap<int, VkPhysicalDevice> candidates;

    for (const auto& device : devices) {
        int score = rateDeviceSuitability(device);
        candidates.insert(std::make_pair(score, device));
    }

    // Vérifie si le meilleur candidat correspond au besoin
    if (candidates.rbegin()->first > 0) {
        physicalDevice = candidates.rbegin()->second;
    } else {
        throw std::runtime_error("Impossible de trouver un GPU adéquat !");
    }
}

int rateDeviceSuitability(VkPhysicalDevice device) {
    ...

    int score = 0;

    // Les GPU dédiés ont un avantage significatif de par leur performance
    if (deviceProperties.deviceType == VK_PHYSICAL_DEVICE_TYPE_DISCRETE_GPU) {
        score += 1000;
    }

    // La taille maximale des textures impacte la qualité des graphismes
    score += deviceProperties.limits.maxImageDimension2D;

    // L'application ne peut fonctionner sans les geometry shaders
    if (!deviceFeatures.geometryShader) {
        return 0;
    }

    return score;
}
```

Vous n'avez pas besoin d'implémenter tout ça pour ce tutoriel. C'est uniquement pour donner une idée sur la façon de concevoir le processus de sélection. Vous pourriez également vous contenter d'afficher les noms des cartes graphiques et laisser l'utilisateur choisir.

Comme nous ne faisons que commencer, nous prendrons la première carte supportant Vulkan :

```
bool isDeviceSuitable(VkPhysicalDevice device) {
    return true;
}
```

Dans la section suivante, nous discuterons de la première réelle fonctionnalité à vérifier.

IV-A-4-c - Familles de queues (queue families)

Il a été évoqué que la majorité des opérations avec Vulkan, de l'affichage jusqu'au chargement d'une texture sur le GPU, s'effectuent en envoyant des commandes à une queue. Il existe différents types de queues provenant de différentes familles de queues. De plus chaque famille de queues ne permet qu'un sous-ensemble de commandes.

Par exemple, une famille de queues peut ne permettre que les commandes liées aux calculs et une autre ne permettre que les opérations liées aux transferts mémoire.

Nous devons vérifier les familles de queues supportées par le périphérique et lesquelles supportent les commandes que nous souhaitons utiliser. Pour cela, nous allons ajouter une nouvelle fonction `findQueueFamilies()` qui recherche toutes les queues dont nous avons besoin.

Dès à présent, nous allons chercher une queue qui supporte les commandes graphiques. La fonction pourrait ressembler à ça :

```
uint32_t findQueueFamilies(VkPhysicalDevice device) {
    // Logique pour trouver une famille de queues pour les opérations graphiques
}
```

Toutefois, dans l'un des prochains chapitres nous devrons chercher une autre queue. Préparons-nous à ce cas et empaquetons l'indice dans une structure :

```
struct QueueFamilyIndices {
    uint32_t graphicsFamily;
};

QueueFamilyIndices findQueueFamilies(VkPhysicalDevice device) {
    QueueFamilyIndices indices;
    // Logique pour trouver une famille de queues avec laquelle remplir la structure
    return indices;
}
```

Que se passe-t-il si une famille n'est pas disponible ? Nous pourrions lancer une exception dans la fonction `findQueueFamilies()`, mais cette fonction n'est pas vraiment le bon endroit pour prendre des décisions concernant le choix du bon périphérique. Par exemple, nous pourrions préférer des périphériques avec une queue de transfert dédiée, sans toutefois que ce soit une obligation. Par conséquent nous avons besoin d'indiquer si une certaine famille de queues a été trouvée.

Ce n'est pas très pratique d'utiliser une valeur magique pour indiquer la non-existence d'une famille alors que n'importe quelle valeur de `uint32_t` peut théoriquement être une valeur valide d'index de famille, 0 inclus. Heureusement, le C++17 introduit un type qui permet la distinction entre le cas où la valeur existe et celui où elle n'existe pas :

```
#include <optional>

...
std::optional<uint32_t> graphicsFamily;

std::cout << std::boolalpha << graphicsFamily.has_value() << std::endl; // false
graphicsFamily = 0;

std::cout << std::boolalpha << graphicsFamily.has_value() << std::endl; // true
```

`std::optional` est un wrapper qui ne contient aucune valeur tant que vous ne lui en assignez pas une. Vous pouvez, quel que soit le moment, lui demander s'il contient une valeur ou non en appelant la fonction `has_value()`. Nous pouvons donc changer le code comme suit :

```
#include <optional>

...
struct QueueFamilyIndices {
    std::optional<uint32_t> graphicsFamily;
};

QueueFamilyIndices findQueueFamilies(VkPhysicalDevice device) {
```

```

QueueFamilyIndices indices;
// Assigne l'index à la famille de queues qui a été trouvée
return indices;
}
    
```

Nous pouvons maintenant commencer à implémenter `findQueueFamilies` :

```

QueueFamilyIndices findQueueFamilies(VkPhysicalDevice device) {
    QueueFamilyIndices indices;

    ...
    return indices;
}
    
```

La récupération de la liste des familles de queues disponibles se fait de la même manière que d'habitude, avec la fonction `vkGetPhysicalDeviceQueueFamilyProperties()` :

```

uint32_t queueFamilyCount = 0;
vkGetPhysicalDeviceQueueFamilyProperties(device, &queueFamilyCount, nullptr);

std::vector<VkQueueFamilyProperties> queueFamilies(queueFamilyCount);
vkGetPhysicalDeviceQueueFamilyProperties(device, &queueFamilyCount, queueFamilies.data());
    
```

La structure `VkQueueFamilyProperties` contient les détails de la famille de queues, notamment le type des opérations supportées et le nombre de queues pouvant être créées pour cette famille . Nous devons trouver au moins une famille de queues supportant `VK_QUEUE_GRAPHICS_BIT` :

```

int i = 0;
for (const auto& queueFamily : queueFamilies) {
    if (queueFamily.queueFlags & VK_QUEUE_GRAPHICS_BIT) {
        indices.graphicsFamily = i;
    }

    i++;
}
    
```

Nous avons maintenant une fonction pratique pour trouver une famille de queues et nous pouvons l'utiliser comme vérification dans la fonction `isDeviceSuitable()` pour s'assurer que le périphérique peut recevoir les commandes que nous voulons utiliser :

```

bool isDeviceSuitable(VkPhysicalDevice device) {
    QueueFamilyIndices indices = findQueueFamilies(device);

    return indices.graphicsFamily.has_value();
}
    
```

Pour que ce soit plus pratique, nous allons aussi ajouter une fonction de vérification générique à la structure :

```

struct QueueFamilyIndices {
    std::optional<uint32_t> graphicsFamily;

    bool isComplete() {
        return graphicsFamily.has_value();
    }
};

...

bool isDeviceSuitable(VkPhysicalDevice device) {
    QueueFamilyIndices indices = findQueueFamilies(device);

    return indices.isComplete();
}
    
```

Nous pouvons maintenant l'utiliser pour sortir plus tôt de la fonction `findQueueFamilies` :

```
for (const auto& queueFamily : queueFamilies) {
    ...
    if (indices.isComplete()) {
        break;
    }
    i++;
}
```

Bien, c'est tout ce dont nous aurons besoin pour choisir le bon périphérique physique ! La prochaine étape est de créer un périphérique logique pour s'y interfaçer.

Code C++

IV-A-5 - Périphérique logique et queues

IV-A-5-a - Introduction

Après avoir sélectionné le périphérique physique, nous devons configurer un périphérique logique pour s'y interfaçer. Le processus de création est similaire à celui de l'instance : nous devons décrire les fonctionnalités dont nous avons besoin. De plus, nous devons spécifier les queues à créer maintenant que nous savons quelles familles de queues sont disponibles. Vous pouvez même créer plusieurs périphériques logiques pour un même périphérique physique dans le cas où vous avez des contraintes changeantes.

Commencez par ajouter un nouveau membre à la classe pour stocker la référence au périphérique logique :

```
VkDevice device;
```

Ensuite, ajoutez une fonction `createLogicalDevice()` qui sera appelée depuis `initVulkan()`.

```
void initVulkan() {
    createInstance();
    setupDebugMessenger();
    pickPhysicalDevice();
    createLogicalDevice();
}

void createLogicalDevice() {
}
```

IV-A-5-b - Spécifier les queues à créer

La création d'un périphérique logique nécessite de fournir plusieurs informations au travers de structures. La première de ces structures s'appelle **VkDeviceQueueCreateInfo**. Elle indique le nombre de queues que nous désirons pour chaque famille de queues. Pour le moment nous n'avons besoin que d'une queue de la famille des fonctionnalités graphiques.

```
QueueFamilyIndices indices = findQueueFamilies(physicalDevice);

VkDeviceQueueCreateInfo queueCreateInfo{};
queueCreateInfo.sType = VK_STRUCTURE_TYPE_DEVICE_QUEUE_CREATE_INFO;
queueCreateInfo.queueFamilyIndex = indices.graphicsFamily.value();
queueCreateInfo.queueCount = 1;
```

Les pilotes actuels ne permettent de créer qu'un petit nombre de queues pour chaque famille et vous n'avez pas réellement besoin de plus d'une. En vérité, vous pouvez créer les tampons de commandes (command buffers) sur plusieurs threads puis les envoyer en une fois au thread principal avec un seul appel peu coûteux en performance.

Vulkan permet d'assigner des priorités aux queues pour influencer l'ordonnancement l'exécution des tampons de commandes par le biais d'un nombre à virgule flottante entre 0.0 et 1.0. Même avec une seule queue, vous devez définir la priorité :

```
float queuePriority = 1.0f;
queueCreateInfo.pQueuePriorities = &queuePriority;
```

IV-A-5-c - Spécifier les fonctionnalités utilisées

Les prochaines informations à fournir sont les fonctionnalités du périphérique que nous allons utiliser. Ce sont celles dont nous avons vérifié la présence avec **vkGetPhysicalDeviceFeatures()** dans le chapitre précédent, tels que le support des geometry shaders. Pour le moment, nous n'avons besoin de rien de spécial. Nous pouvons donc nous contenter de créer la structure et de tout laisser à la valeur par défaut **VK_FALSE**. Nous reviendrons sur cette structure quand nous ferons des choses plus intéressantes avec Vulkan.

```
VkPhysicalDeviceFeatures deviceFeatures{};
```

IV-A-5-d - Créer le périphérique logique

Avec ces deux structures prêtes, nous pouvons enfin remplir la structure principale appelée **VkDeviceCreateInfo**.

```
VkDeviceCreateInfo createInfo{};
createInfo.sType = VK_STRUCTURE_TYPE_DEVICE_CREATE_INFO;
```

D'abord, ajoutez des pointeurs sur la création des queues et sur les fonctionnalités utilisées du périphérique :

```
createInfo.pQueueCreateInfos = &queueCreateInfo;
createInfo.queueCreateInfoCount = 1;

createInfo.pEnabledFeatures = &deviceFeatures;
```

Le reste devrait vous rappeler la structure **VkInstanceCreateInfo**. Nous devons spécifier les extensions et les couches de validation. La différence est que ces dernières sont maintenant spécifiques au périphérique.

Par exemple, **VK_KHR_swapchain** est une extension spécifique au GPU. Celle-ci vous permet d'envoyer le rendu généré par ce périphérique aux fenêtres. Il est possible de rencontrer des périphériques Vulkan sans cette fonctionnalité, notamment si le matériel ne supporte que les opérations de calcul. Nous reviendrons sur cette extension dans le chapitre dédié à la swap chain.

Les anciennes implémentations de Vulkan faisaient la distinction entre les couches de validation spécifiques à l'instance et celles spécifiques au périphérique. Ce n'est maintenant **plus le cas**. Cela signifie que les champs **enabledLayerCount** et **ppEnabledLayerNames** de la structure **VkDeviceCreateInfo** sont ignorés dans les implémentations à jour. Toutefois, cela reste une bonne idée de les définir afin d'être compatible avec les anciennes implémentations.

```
createInfo.enabledExtensionCount = 0;

if (enableValidationLayers) {
    createInfo.enabledLayerCount = static_cast<uint32_t>(validationLayers.size());
    createInfo.ppEnabledLayerNames = validationLayers.data();
} else {
    createInfo.enabledLayerCount = 0;
}
```

Pour le moment, nous n'avons besoin d'aucune extension spécifique au périphérique.

C'est bon, nous pouvons maintenant instancier le périphérique logique en appelant la fonction **vkCreateDevice()**.

```
if (vkCreateDevice(physicalDevice, &createInfo, nullptr, &device) != VK_SUCCESS) {
    throw std::runtime_error("Échec de création du périphérique logique !");
}
```

Le premier paramètre est le périphérique physique avec lequel s'interfacer. Ensuite vient la structure contenant les informations sur la queue et notre utilisation. Le troisième paramètre est un pointeur sur la fonction d'allocation personnalisée. Le dernier paramètre est un pointeur où stocker la référence au périphérique physique. Comme pour la fonction de création d'une instance, cette fonction retourne une erreur lors de l'activation d'une extension absente ou en indiquant l'utilisation de fonctionnalités non supportées.

Le périphérique doit être détruit dans la fonction **cleanup()** avec la fonction **vkDestroyDevice()** :

```
void cleanup() {
    vkDestroyDevice(device, nullptr);
    ...
}
```

Les périphériques logiques n'interagissent pas directement avec les instances. C'est pour cela que nous ne l'incluant pas comme paramètre.

IV-A-5-e - Récupérer des références aux queues

Les queues sont automatiquement créées avec le périphérique logique. Cependant nous n'avons aucune référence pour les utiliser. Ajoutez un membre à la classe pour stocker une référence à la queue :

```
VkQueue graphicsQueue;
```

Les queues sont implicitement détruites lors de la destruction du périphérique logique. Nous n'avons donc pas à nous en charger dans la fonction **cleanup()**.

Nous pouvons utiliser la fonction **vkGetDeviceQueue()** pour récupérer les références aux queues de chaque famille. Le premier paramètre est le périphérique logique, s'ensuit la famille de queues, l'index de queue et un pointeur sur la variable dans laquelle stocker la référence à la queue. Comme nous ne créons qu'une seule queue de cette famille, nous utilisons l'index 0.

```
vkGetDeviceQueue(device, indices.graphicsFamily.value(), 0, &graphicsQueue);
```

Grâce au périphérique logique et les queues, nous pouvons maintenant utiliser la carte graphique ! Dans les prochains chapitres, nous allons configurer les ressources pour envoyer le rendu au gestionnaire de fenêtres.

Code C++

IV-B - Envoi du rendu à l'écran

IV-B-1 - Surface de fenêtre

Vulkan n'a aucune connaissance des spécificités de la plateforme et ne peut donc pas s'interfacer avec le gestionnaire de fenêtres. Pour créer une connexion permettant d'envoyer (présenter) les rendus à l'écran, nous devons utiliser les extensions Window System Integration (WSI). Nous verrons dans ce chapitre la première d'entre elles : **VK_KHR_surface**. Celle-ci nous offre un objet **VkSurfaceKHR**, un type abstrait représentant une surface sur laquelle afficher les rendus. Dans notre programme, cette surface est fournie par la fenêtre que nous avons ouverte grâce à **GLFW**.

L'extension `VK_KHR_surface` est une extension au niveau de l'instance et nous l'avons déjà activée. En effet, elle est incluse dans la liste renvoyée par la fonction `glfwGetRequiredInstanceExtensions()`. Par ailleurs, cette liste inclut aussi quelques autres extensions WSI que nous allons utiliser dans les prochains chapitres.

La surface liée à la fenêtre doit être créée juste après l'instance, car elle peut influencer le choix du périphérique physique. Nous ne nous intéressons à ce sujet que maintenant, car les surfaces font partie d'un sujet plus grand : celui des cibles de rendu et de la présentation du rendu. Ajouter ce sujet à la configuration de base aurait complexifié le tutoriel. Il est important de noter que les surfaces de fenêtre sont complètement optionnelles dans Vulkan et vous pouvez vous contenter d'un rendu hors écran. Vulkan vous offre ces possibilités sans vous demander de recourir à des astuces comme créer une fenêtre invisible (chose nécessaire en OpenGL).

IV-B-1-a - Création de la surface

Commencez par ajouter un membre `surface` à la classe, sous le messager de débogage.

```
VkSurfaceKHR surface;
```

Bien que l'utilisation d'un objet `VkSurfaceKHR` soit indépendante de la plateforme, sa création ne l'est pas, car elle dépend du gestionnaire de fenêtres. Plus précisément, une telle fonction nécessite une référence à un `HWND` et à un `HMODULE` sous Windows. C'est pourquoi il existe des extensions spécifiques à la plateforme, dont `VK_KHR_win32_surface` sous Windows. Ces extensions sont automatiquement renvoyées par la fonction `glfwGetRequiredInstanceExtensions()`.

Je vais présenter comment utiliser cette extension spécifique à la plateforme pour créer une surface sur Windows, mais nous n'allons pas l'utiliser dans ce tutoriel. En effet, ce n'est pas logique d'utiliser une bibliothèque comme GLFW pour ensuite utiliser du code spécifique à la plateforme. GLFW offre une fonction `glfwCreateWindowSurface()` qui gère les différences entre les plateformes pour nous. Toutefois, cela reste une bonne chose de voir comment cela fonctionne avant de s'en servir.

La surface liée à la fenêtre est un objet Vulkan et nécessite de remplir la structure `VkWin32SurfaceCreateInfoKHR` pour la créer. Elle possède deux paramètres importants : `hwnd` et `hinstance`. Ce sont les références à la fenêtre et au processus courant.

```
VkWin32SurfaceCreateInfoKHR createInfo{};  
createInfo.sType = VK_STRUCTURE_TYPE_WIN32_SURFACE_CREATE_INFO_KHR;  
createInfo(hwnd = glfwGetWin32Window(window);  
createInfo.hinstance = GetModuleHandle(nullptr);
```

Nous pouvons extraire le `HWND` de la fenêtre à l'aide de la fonction `glfwGetWin32Window()`. La fonction `GetModuleHandle()` retourne la référence `HINSTANCE` du thread courant.

La surface peut maintenant être créée avec `vkCreateWin32SurfaceKHR()`. Cette fonction prend en paramètre une instance, les détails pour la création de la surface, l'allocateur optionnel et la variable dans laquelle stocker la surface. Bien que cette fonction fasse partie d'une extension WSI, elle est chargée par le chargeur standard Vulkan, car celle-ci est couramment utilisée. Nous n'avons ainsi pas à la charger à la main :

```
if (vkCreateWin32SurfaceKHR(instance, &createInfo, nullptr, &surface) != VK_SUCCESS) {  
    throw std::runtime_error("Échec de création de la surface !");  
}
```

Ce processus est similaire pour toutes les plateformes. Par exemple, sous Linux, vous devez utiliser la fonction `vkCreateXcbSurfaceKHR()`. Les paramètres spécifiques à X11 sont une référence à la fenêtre et une connexion à XCB.

La fonction `glfwCreateWindowSurface()` implémente donc tout cela pour nous et utilise le code adéquat pour chaque plateforme. Nous devons maintenant l'intégrer à notre programme. Ajoutez la fonction `createSurface()` et appelez-la dans la fonction `initVulkan()` juste après la création de l'instance et l'appel à la fonction `setupDebugMessenger()` :

```
void initVulkan() {
    createInstance();
    setupDebugMessenger();
    createSurface();
    pickPhysicalDevice();
    createLogicalDevice();
}

void createSurface() {
}
```

L'appel à la fonction fournie par GLFW ne prend que quelques paramètres au lieu d'une structure, ce qui rend la fonction `createSurface()` simple :

```
void createSurface() {
    if (glfwCreateWindowSurface(instance, window, nullptr, &surface) != VK_SUCCESS) {
        throw std::runtime_error("Échec lors de la création de la surface !");
    }
}
```

Les paramètres sont l'instance Vulkan (de type `VkInstance`), le pointeur sur la fenêtre GLFW, l'allocateur optionnel et un pointeur sur une variable de type `VkSurfaceKHR`. GLFW ne fournit aucune fonction pour détruire une surface, mais nous pouvons le faire nous-mêmes grâce à une fonction Vulkan :

```
void cleanup() {
    ...
    vkDestroySurfaceKHR(instance, surface, nullptr);
    vkDestroyInstance(instance, nullptr);
    ...
}
```

Détruez bien la surface avant l'instance.

IV-B-1-b - Vérification du support d'envoi du rendu à la fenêtre

Bien que l'implémentation de Vulkan puisse supporter l'intégration du gestionnaire de fenêtre, cela ne signifie pas que tous les périphériques du système le supportent. Nous devons donc modifier la fonction `isDeviceSuitable()` pour nous assurer que le périphérique est capable d'envoyer (présenter) des images à la surface que nous avons créée. Cet envoi est une fonctionnalité spécifique à certaines familles de queue. La problématique est donc de trouver une famille qui supporte cette fonctionnalité.

En réalité, il est possible que les familles de queue supportant les commandes d'affichage ne supportent pas l'envoi du rendu et inversement. Par conséquent, nous devons gérer le cas où la queue pour l'envoi du rendu ne serait pas la même que celle des commandes graphiques et modifier la structure `QueueFamilyIndices` en conséquence :

```
struct QueueFamilyIndices {
    std::optional<uint32_t> graphicsFamily;
    std::optional<uint32_t> presentFamily;

    bool isComplete() {
        return graphicsFamily.has_value() && presentFamily.has_value();
    }
};
```

Nous devons ensuite modifier la fonction `findQueueFamilies()` pour qu'elle cherche une famille de queues pouvant supporter les commandes de présentation. La fonction permettant de vérifier cela est `vkGetPhysicalDeviceSurfaceSupportKHR()`. Elle accepte comme paramètres : le périphérique physique, l'indice de la famille de queues, la surface et elle stocke le résultat dans le quatrième paramètre qui est un booléen. Appelez-la depuis la même boucle que pour `VK_QUEUE_GRAPHICS_BIT` :

```
VkBool32 presentSupport = false;
vkGetPhysicalDeviceSurfaceSupportKHR(device, i, surface, &presentSupport);
```

Il ne reste plus qu'à vérifier la valeur du booléen pour stocker la queue si elle correspond à notre besoin :

```
if (presentSupport) {
    indices.presentFamily = i;
}
```

Il est très probable que ces deux familles de queues soient en fait les mêmes, mais nous les traiterons comme si elles étaient différentes pour garder une approche uniforme. Vous pouvez cependant préférer un périphérique physique avec une famille de queues supportant le rendu et la présentation afin de légèrement améliorer les performances.

IV-B-1-c - Création de la queue de présentation

Il nous reste plus qu'à modifier la création du périphérique logique pour créer la queue de présentation et obtenir la référence de celle-ci (de type **VkQueue**). Ajoutez un membre à la classe pour cette référence :

```
VkQueue presentQueue;
```

Nous avons besoin de plusieurs structures **VkDeviceQueueCreateInfo**, une pour chaque famille de queues. Une manière de gérer ces structures est d'utiliser un ensemble contenant tous les indices des familles nécessaires pour obtenir les queues voulues, comme ci-dessous :

```
#include <set>

...
QueueFamilyIndices indices = findQueueFamilies(physicalDevice);

std::vector<VkDeviceQueueCreateInfo> queueCreateInfos;
std::set<uint32_t> uniqueQueueFamilies = {indices.graphicsFamily.value(),
    indices.presentFamily.value()};

float queuePriority = 1.0f;
for (uint32_t queueFamily : uniqueQueueFamilies) {
    VkDeviceQueueCreateInfo queueCreateInfo{};
    queueCreateInfo.sType = VK_STRUCTURE_TYPE_DEVICE_QUEUE_CREATE_INFO;
    queueCreateInfo.queueFamilyIndex = queueFamily;
    queueCreateInfo.queueCount = 1;
    queueCreateInfo.pQueuePriorities = &queuePriority;
    queueCreateInfos.push_back(queueCreateInfo);
}
```

Il faut ensuite modifier **VkDeviceCreateInfo** pour qu'il pointe sur le contenu du tableau :

```
createInfo.queueCreateInfoCount = static_cast<uint32_t>(queueCreateInfos.size());
createInfo.pQueueCreateInfos = queueCreateInfos.data();
```

Si les familles de queues sont les mêmes, nous n'avons besoin de les indiquer qu'une seule fois. Il faut enfin ajouter un appel pour récupérer la référence sur la queue :

```
vkGetDeviceQueue(device, indices.presentFamily.value(), 0, &presentQueue);
```

Si les queues sont les mêmes, les variables `VkQueue` auront la même valeur. Dans le prochain chapitre nous nous intéresserons aux swap chain et verrons comment elles permettent d'envoyer les rendus à l'écran.

Code C++

IV-B-2 - Swap chain

Vulkan ne possède pas de notion de tampon d'image par défaut. Il nous faut donc créer une infrastructure pour mettre en place ces tampons afin de les utiliser pour le rendu et de les afficher à l'écran. Sur Vulkan, une telle infrastructure s'appelle « swap chain » et doit être créée explicitement. Principalement, la « swap chain » est une liste d'images en attente d'être affichées. Notre application devra récupérer une des images de la file, dessiner dessus, puis la retourner à la file d'attente. Le fonctionnement de la file d'attente et les conditions d'envoi d'une image dépendent du paramétrage de la « swap chain ». Cependant, l'intérêt principal de la swap chain est de synchroniser l'envoi des images avec le rafraîchissement de l'écran.

IV-B-2-a - Vérification du support des swap chain

Toutes les cartes graphiques ne sont pas capables d'envoyer des images directement à l'écran, notamment dans les serveurs qui peuvent ne pas avoir de sortie vidéo. De plus, sachant que l'envoi d'image est très dépendant du gestionnaire de fenêtres et de la surface associée à la fenêtre, la « swap chain » ne fait pas partie du cœur de Vulkan. Vous devez vérifier le support de l'extension `VK_KHR_swapchain` et l'activer manuellement.

Pour cela, nous allons modifier la fonction `isDeviceSuitable()` pour vérifier si cette extension est supportée. Nous avons déjà vu comment lister les extensions supportées par un `VkPhysicalDevice`, donc cette modification devrait être assez simple. Notez que le fichier d'en-tête Vulkan fournit la macro `VK_KHR_SWAPCHAIN_EXTENSION_NAME` indiquant le nom de l'extension pour les « swap chain ». L'utiliser permet d'éviter les fautes de frappe.

Déclarez d'abord une liste d'extensions nécessaires au périphérique, comme nous l'avons fait pour la liste des couches de validation :

```
const std::vector<const char*> deviceExtensions = {
    VK_KHR_SWAPCHAIN_EXTENSION_NAME
};
```

Créez ensuite une nouvelle fonction nommée `checkDeviceExtensionSupport()` pour englober la nouvelle vérification et appelez-la depuis la fonction `isDeviceSuitable()` :

```
bool isDeviceSuitable(VkPhysicalDevice device) {
    QueueFamilyIndices indices = findQueueFamilies(device);

    bool extensionsSupported = checkDeviceExtensionSupport(device);

    return indices.isComplete() && extensionsSupported;
}

bool checkDeviceExtensionSupport(VkPhysicalDevice device) {
    return true;
}
```

Modifiez le code de la fonction pour lister les extensions et vérifiez la présence de celles qui nous sont utiles :

```
bool checkDeviceExtensionSupport(VkPhysicalDevice device) {
    uint32_t extensionCount;
    vkEnumerateDeviceExtensionProperties(device, nullptr, &extensionCount, nullptr);

    std::vector<VkExtensionProperties> availableExtensions(extensionCount);
    vkEnumerateDeviceExtensionProperties(device, nullptr, &extensionCount,
        availableExtensions.data());
```

```

std::set<std::string> requiredExtensions(deviceExtensions.begin(), deviceExtensions.end());

for (const auto& extension : availableExtensions) {
    requiredExtensions.erase(extension.extensionName);
}

return requiredExtensions.empty();
}
    
```

J'ai décidé d'utiliser un std::set contenant des std::string pour représenter les extensions requises en attente de confirmation. De cette façon, nous pouvons ainsi facilement les éliminer en énumérant la liste des extensions disponibles. Vous pouvez également utiliser des boucles imbriquées comme dans la fonction checkValidationLayerSupport(). La différence en termes de performances n'est pas significative. Lancez le code et vérifiez que votre carte graphique est capable de gérer une « swap chain ». Normalement, la présence d'une queue de présentation implique la présence de l'extension de la « swap chain ». Toutefois, il est préférable de prendre ces précautions, de plus l'extension doit être activée explicitement.

IV-B-2-b - Activation des extensions du périphérique

L'utilisation de la « swap chain » nécessite l'activation de l'extension VK_KHR_swapchain. Son activation ne requiert qu'un léger changement à la structure de création du périphérique logique :

```

createInfo.enabledExtensionCount = static_cast<uint32_t>(deviceExtensions.size());
createInfo.ppEnabledExtensionNames = deviceExtensions.data();
    
```

Supprimez bien l'ancienne ligne `createInfo.enabledExtensionCount = 0;`

IV-B-2-c - Récupération des détails à propos du support de la « swap chain »

La seule vérification de la disponibilité d'une « swap chain » ne suffit pas. En effet, celle-ci pourrait être incompatible avec la surface liée à la fenêtre. La création d'une « swap chain » nécessite plus de paramètres que pour la création d'une instance ou du périphérique. Par conséquent, nous devons récupérer plus d'informations avant de pouvoir continuer.

Il y a trois types de propriétés que nous devrons vérifier :

- les possibilités basiques de la surface (nombre minimum et maximum d'images dans la « swap chain », hauteur et largeur minimale et maximale des images) ;
- les formats de la surface (format des pixels, palette de couleur) ;
- les modes d'envoi disponibles.

Nous utiliserons une structure comme celle de la fonction `findQueueFamilies()` pour stocker ces informations. Les trois catégories mentionnées plus haut se présentent sous la forme de la structure et des listes de structures suivantes :

```

struct SwapChainSupportDetails {
    VkSurfaceCapabilitiesKHR capabilities;
    std::vector<VkSurfaceFormatKHR> formats;
    std::vector<VkPresentModeKHR> presentModes;
};
    
```

Créons maintenant une nouvelle fonction `querySwapChainSupport()` qui remplira cette structure :

```

SwapChainSupportDetails querySwapChainSupport(VkPhysicalDevice device) {
    SwapChainSupportDetails details;

    return details;
}
    
```

Cette section couvre la récupération des structures. Ce qu'elles signifient sera expliqué dans la section suivante.

Commençons par les capacités de base de la surface. Il suffit de demander ces informations et elles nous seront fournies sous la forme d'une structure du type `VkSurfaceCapabilitiesKHR`.

```
vkGetPhysicalDeviceSurfaceCapabilitiesKHR(device, surface, &details.capabilities);
```

Cette fonction requiert que le périphérique physique (`VkPhysicalDevice`) et la surface de fenêtre (`VkSurfaceKHR`) soient passés en paramètres pour en obtenir les capacités. Toutes les fonctions récupérant des capacités de la « swap chain » demanderont ces paramètres, car ils en sont les composants centraux.

La prochaine étape est de récupérer les formats supportés par la surface. Comme c'est une liste de structures, cette acquisition suit le rituel en deux étapes :

```
uint32_t formatCount;
vkGetPhysicalDeviceSurfaceFormatsKHR(device, surface, &formatCount, nullptr);

if (formatCount != 0) {
    details.formats.resize(formatCount);
    vkGetPhysicalDeviceSurfaceFormatsKHR(device, surface, &formatCount, details.formats.data());
}
```

Assurez-vous que le vecteur est redimensionné pour contenir tous les formats disponibles. Finalement, récupérez les modes de présentation supportés grâce à la fonction `vkGetPhysicalDeviceSurfacePresentModesKHR()` :

```
uint32_t presentModeCount;
vkGetPhysicalDeviceSurfacePresentModesKHR(device, surface, &presentModeCount, nullptr);

if (presentModeCount != 0) {
    details.presentModes.resize(presentModeCount);
    vkGetPhysicalDeviceSurfacePresentModesKHR(device, surface, &presentModeCount,
    details.presentModes.data());
}
```

Toutes les informations sont dans des structures. Il faut donc améliorer une nouvelle fois la fonction `isDeviceSuitable()` pour vérifier si le support de la « swap chain » correspond à notre besoin. Pour ce tutoriel, une « swap chain » est adéquate si elle a un format d'image et un mode de présentation correspondant à la surface que nous avons.

```
bool swapChainAdequate = false;
if (extensionsSupported) {
    SwapChainSupportDetails swapChainSupport = querySwapChainSupport(device);

    swapChainAdequate = !swapChainSupport.formats.empty() && !swapChainSupport.presentModes.empty();
}
```

Il est important de ne vérifier le support de la « swap chain » qu'après s'être assuré que l'extension est disponible. La dernière ligne de la fonction devient donc :

```
return indices.isComplete() && extensionsSupported && swapChainAdequate;
```

IV-B-2-d - Choix des bons paramètres pour la « swap chain »

Si les conditions validées par la fonction `swapChainAdequate()` sont remplies, alors le support de la swap chain est assuré. Il existe cependant plusieurs modes ayant chacun leur avantage. Nous allons maintenant écrire quelques fonctions qui détermineront les bons paramètres pour obtenir la meilleure « swap chain » possible. Il y a trois types de paramètres à déterminer :

- le format de la surface (profondeur de la couleur) ;
- le mode de présentation (les conditions d'échange des images vers l'écran) ;

- la zone d'échange (swap extent) (la résolution des images dans la « swap chain »).

Pour chacun de ces paramètres, nous aurons une valeur idéale que nous choisirons si elle est disponible, sinon, nous nous rabattrons sur ce qui il y aura de mieux.

IV-B-2-d-i - Format de la surface

La fonction utilisée pour déterminer ce paramètre commence ainsi. Nous lui passerons comme argument le membre formats de la structure SwapChainSupportDetails.

```
VkSurfaceFormatKHR chooseSwapSurfaceFormat(const std::vector<VkSurfaceFormatKHR>&
availableFormats) {
}
```

Chaque VkSurfaceFormatKHR contient une propriété format et une propriété colorSpace. Le format indique les canaux de couleur disponibles et les types. Par exemple VK_FORMAT_B8G8R8A8_SRGB signifie que nous stockons les canaux de couleur R, G, B et A dans cet ordre, dans des entiers non signés sur 8 bits. La propriété colorSpace permet de vérifier que l'espace de couleur sRGB est supporté ou non par le biais de l'indicateur VK_COLOR_SPACE_SRGB_NONLINEAR_KHR.

 L'indicateur VK_COLOR_SPACE_SRGB_NONLINEAR_KHR s'appelait VK_COLOSPACE_SRGB_NONLINEAR_KHR dans de précédentes versions de la spécification.

Nous utiliserons, si disponible, l'espace de couleur sRGB. Ce dernier donne des résultats **mieux perçus par l'œil humain**. Aussi, c'est quasiment l'espace de couleur standard pour les images, telles que les textures que nous allons utiliser plus tard. Par conséquent, nous devons aussi utiliser un format de couleurs SRGB. Le format VK_FORMAT_B8G8R8A8_SRGB est l'un des plus courants.

Itérons sur la liste et voyons si la meilleure combinaison est disponible :

```
for (const auto& availableFormat : availableFormats) {
    if (availableFormat.format == VK_FORMAT_B8G8R8A8_SRGB && availableFormat.colorSpace ==
VK_COLOR_SPACE_SRGB_NONLINEAR_KHR) {
        return availableFormat;
    }
}
```

Si cette approche échoue, nous pouvons trier les combinaisons disponibles suivant leur « qualité ». Mais, la plupart du temps, c'est suffisant de simplement utiliser le premier format disponible.

```
VkSurfaceFormatKHR chooseSwapSurfaceFormat(const std::vector<VkSurfaceFormatKHR>&
availableFormats) {
    for (const auto& availableFormat : availableFormats) {
        if (availableFormat.format == VK_FORMAT_B8G8R8A8_SRGB && availableFormat.colorSpace ==
VK_COLOR_SPACE_SRGB_NONLINEAR_KHR) {
            return availableFormat;
        }
    }

    return availableFormats[0];
}
```

IV-B-2-d-ii - Mode de présentation

Le mode de présentation est clairement le paramètre le plus important pour la « swap chain », car il indique les conditions pour afficher les images à l'écran. Il existe quatre modes avec Vulkan :

- `VK_PRESENT_MODE_IMMEDIATE_KHR` : les images émises par votre application sont directement envoyées à l'écran, ce qui peut produire des déchirures (tearing) ;
- `VK_PRESENT_MODE_FIFO_KHR` : la « swap chain » est une file d'attente. L'écran récupère l'image en haut de la pile quand il est rafraîchi, alors que le programme insère les images rendues à l'arrière. Si la queue est pleine, le programme doit attendre. Ce mode est très similaire à la synchronisation verticale utilisée par la plupart des jeux vidéo modernes. L'instant durant lequel l'écran est rafraîchi s'appelle l'intervalle de rafraîchissement vertical (vertical blank) ;
- `VK_PRESENT_MODE_FIFO_RELAXED_KHR` : ce mode ne diffère du précédent que lorsque l'application est en retard et que la queue est vide pendant l'intervalle de rafraîchissement vertical. Au lieu d'attendre le prochain rafraîchissement, une image arrivant dans la file d'attente sera immédiatement transmise à l'écran. Cela peut entraîner des déchirures.
- `VK_PRESENT_MODE_MAILBOX_KHR` : ce mode est une autre variation du second mode. Au lieu de bloquer l'application quand la file d'attente est pleine, les images présentes dans la queue sont remplacées par les nouvelles. Ce mode peut être utilisé pour implémenter le triple buffering, qui vous permet d'éliminer les déchirures sur les images tout en ayant moins de problèmes de latence qu'avec la synchronisation verticale standard reposant sur du double buffering.

Seul le mode `VK_PRESENT_MODE_FIFO_KHR` est toujours disponible. Nous aurons donc encore à écrire une fonction pour trouver le meilleur mode disponible :

```
VkPresentModeKHR chooseSwapPresentMode(const std::vector<VkPresentModeKHR>&
availablePresentModes) {
    return VK_PRESENT_MODE_FIFO_KHR;
}
```

Je pense que le triple buffering est un très bon compromis. Il permet d'éviter les déchirures tout en gardant une latence très basse en affichant les images qui sont à jour. Vérifions si ce mode est disponible dans la liste :

```
VkPresentModeKHR chooseSwapPresentMode(const std::vector<VkPresentModeKHR>&
availablePresentModes) {
    for (const auto& availablePresentMode : availablePresentModes) {
        if (availablePresentMode == VK_PRESENT_MODE_MAILBOX_KHR) {
            return availablePresentMode;
        }
    }
    return VK_PRESENT_MODE_FIFO_KHR;
}
```

IV-B-2-d-iii - La zone d'échange

Il ne nous reste plus qu'une propriété majeure, pour laquelle nous allons créer une dernière fonction :

```
VkExtent2D chooseSwapExtent(const VkSurfaceCapabilitiesKHR& capabilities) {
}
```

La zone d'échange (swap extent) correspond à la résolution des images dans la « swap chain ». Généralement, elle est égale à la résolution de la fenêtre que nous utilisons. L'étendue des résolutions disponibles est définie dans la structure `VkSurfaceCapabilitiesKHR`. Vulkan nous demande de faire correspondre la résolution de la fenêtre en indiquant la largeur et la hauteur dans la propriété `currentExtent`. Cependant, certains gestionnaires de fenêtres nous permettent de choisir une résolution différente, cela peut être précisé grâce à une valeur spéciale (la valeur maximale

pour un `uint32_t`) pour la largeur et la hauteur. Dans ce cas, nous choisirons la résolution correspondant le mieux à la taille de la fenêtre, comprise entre `minImageExtent` et `maxImageExtent`.

```
#include <cstdint> // Nécessaire pour UINT32_MAX

...

VkExtent2D chooseSwapExtent(const VkSurfaceCapabilitiesKHR& capabilities) {
    if (capabilities.currentExtent.width != UINT32_MAX) {
        return capabilities.currentExtent;
    } else {
        VkExtent2D actualExtent = {WIDTH, HEIGHT};

        actualExtent.width = std::max(capabilities.minImageExtent.width, std::min(capabilities.maxImageExtent.width,
        actualExtent.width));

        actualExtent.height = std::max(capabilities.minImageExtent.height, std::min(capabilities.maxImageExtent.height,
        actualExtent.height));

        return actualExtent;
    }
}
```

Les fonctions `std::min` et `std::max` sont utilisées pour limiter les valeurs `WIDTH` et `HEIGHT` entre le minimum et le maximum supportés par l'implémentation. Incluez le fichier d'en-tête `<algorithm>` pour les utiliser.

IV-B-2-e - Création de la swap chain

Maintenant que nous avons toutes ces fonctions nous aidant à faire un choix, nous pouvons enfin créer une « swap chain ».

Créez une fonction `createSwapChain()`. Elle commence par récupérer le résultat des fonctions précédentes. Appelez-la depuis `initVulkan()` après la création du périphérique logique.

```
void initVulkan() {
    createInstance();
    setupDebugMessenger();
    createSurface();
    pickPhysicalDevice();
    createLogicalDevice();
    createSwapChain();
}

void createSwapChain() {
    SwapChainSupportDetails swapChainSupport = querySwapChainSupport(physicalDevice);

    VkSurfaceFormatKHR surfaceFormat = chooseSwapSurfaceFormat(swapChainSupport.formats);
    VkPresentModeKHR presentMode = chooseSwapPresentMode(swapChainSupport.presentModes);
    VkExtent2D extent = chooseSwapExtent(swapChainSupport.capabilities);
}
```

En plus de ces propriétés, nous devons décider du nombre d'images que nous souhaitons avoir dans la « swap chain ». L'implémentation informe du nombre minimal pour fonctionner :

```
uint32_t imageCount = swapChainSupport.capabilities.minImageCount;
```

Toutefois, se contenter de la limite basse augmente le risque d'être dans l'attente du pilote lors de la récupération d'une nouvelle image pour effectuer le rendu. Il est donc recommandé de demander au moins une image de plus que le minimum :

```
uint32_t imageCount = swapChainSupport.capabilities.minImageCount + 1;
```

Il nous faut également prendre en compte le maximum d'images supportées par l'implémentation. La valeur 0 signifie qu'il n'y a pas de maximum.

```
if (swapChainSupport.capabilities.maxImageCount > 0 && imageCount >
    swapChainSupport.capabilities.maxImageCount) {
    imageCount = swapChainSupport.capabilities.maxImageCount;
}
```

Comme la tradition le veut avec la création des objets dans Vulkan, la création d'une « swap chain » nécessite de remplir une grande structure. Elle commence de manière habituelle :

```
VkSwapchainCreateInfoKHR createInfo{};
createInfo.sType = VK_STRUCTURE_TYPE_SWAPCHAIN_CREATE_INFO_KHR;
createInfo.surface = surface;
```

Après avoir indiqué la surface à laquelle la « swap chain » doit être liée, nous spécifions les détails sur les images de la « swap chain » :

```
createInfo.minImageCount = imageCount;
createInfo.imageFormat = surfaceFormat.format;
createInfo.imageColorSpace = surfaceFormat.colorSpace;
createInfo.imageExtent = extent;
createInfo.imageArrayLayers = 1;
createInfo.imageUsage = VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT;
```

La propriété `imageArrayLayers` indique le nombre de couches de chaque image. Ce sera toujours 1 sauf si vous développez une application stéréoscopique 3D. Le champ de bits `imageUsage` spécifie le type d'opérations que nous effectuerons avec les images de la « swap chain ». Dans ce tutoriel, nous effectuerons un rendu directement sur les images, c'est-à-dire qu'elles seront utilisées comme attaches de couleur. Il est aussi possible d'effectuer le rendu dans une autre image en vue de faire des effets de post traitement. Dans ce cas, vous devrez utiliser une valeur comme `VK_IMAGE_USAGE_TRANSFER_DST_BIT` et utiliser une opération mémoire pour transférer l'image rendue vers l'image de la « swap chain ».

```
QueueFamilyIndices indices = findQueueFamilies(physicalDevice);
uint32_t queueFamilyIndices[] = {indices.graphicsFamily.value(), indices.presentFamily.value()};

if (indices.graphicsFamily != indices.presentFamily) {
    createInfo.imageSharingMode = VK_SHARING_MODE_CONCURRENT;
    createInfo.queueFamilyIndexCount = 2;
    createInfo.pQueueFamilyIndices = queueFamilyIndices;
} else {
    createInfo.imageSharingMode = VK_SHARING_MODE_EXCLUSIVE;
    createInfo.queueFamilyIndexCount = 0; // Optionnel
    createInfo.pQueueFamilyIndices = nullptr; // Optionnel
}
```

Ensuite, nous devrons indiquer comment gérer les images de la « swap chain » utilisées par plusieurs familles de queues. Cela sera le cas dans notre application si la queue des graphismes n'est pas la même que la queue de présentation. Nous devrons alors dessiner avec la queue graphique puis fournir l'image à la queue de présentation. Il existe deux manières de gérer les images accédées par plusieurs queues :

- `VK_SHARING_MODE_EXCLUSIVE` : une image n'est accessible que par une queue à la fois et sa gestion doit être explicitement transférée à une autre queue pour pouvoir être utilisée dans une autre famille. Cette option offre de meilleures performances ;
- `VK_SHARING_MODE_CONCURRENT` : les images peuvent être simplement utilisées par différentes familles de queues.

Si nous avons deux familles de queues différentes, nous utiliserons le mode concurrent pour éviter d'ajouter un chapitre sur la possession des ressources, car cela nécessite des concepts que nous ne pourrons comprendre correctement que plus tard. Le mode concurrent nous demande de spécifier à l'avance les queues qui partageront les images en utilisant les paramètres `queueFamilyIndexCount` et `pQueueFamilyIndices`. Si les familles de queue

pour les graphismes et la présentation sont les mêmes, ce qui est le cas sur la plupart des cartes graphiques, nous devons rester sur le mode exclusif, car le mode concurrent requiert au moins deux familles de queues différentes.

```
createInfo.preTransform = swapChainSupport.capabilities.currentTransform;
```

Nous pouvons spécifier une transformation à appliquer aux images quand elles entrent dans la « swap chain » si cela est supporté (à vérifier avec la propriété `supportedTransforms` dans `capabilities`). Cela peut être une rotation de 90 degrés ou une symétrie verticale. Si vous ne voulez pas de transformation, spécifiez la transformation actuelle.

```
createInfo.compositeAlpha = VK_COMPOSITE_ALPHA_OPAQUE_BIT_KHR;
```

Le champ `compositeAlpha` indique si le canal alpha doit être utilisé pour mélanger les couleurs avec celles des autres fenêtres dans le gestionnaire de fenêtres. Vous voudrez quasiment tout le temps ignorer cela et indiquer `VK_COMPOSITE_ALPHA_OPAQUE_BIT_KHR` :

```
createInfo.presentMode = presentMode;
createInfo.clipped = VK_TRUE;
```

Le membre `presentMode` est assez simple. Si le membre `clipped` est à `VK_TRUE`, nous indiquons que les couleurs des pixels masqués, notamment par d'autres fenêtres, ne nous intéressent pas. Si vous n'avez pas un besoin particulier de lire ces informations et d'en obtenir des résultats stables, vous obtiendrez de meilleures performances en activant ce mode.

```
createInfo.oldSwapchain = VK_NULL_HANDLE;
```

Il nous reste un dernier champ, `oldSwapChain`. Il est possible avec Vulkan que la « swap chain » devienne invalide ou peu performante pendant l'exécution de votre application. Notamment, cela arrive lorsque la fenêtre est redimensionnée. Dans ce cas, la « swap chain » doit être intégralement recréée et vous devez fournir une référence pointant vers l'ancienne « swap chain ». C'est un sujet compliqué que nous aborderons [dans un futur chapitre](#). Pour le moment, nous gérons seulement la création de la « swap chain ».

Ajoutez un membre à la classe pour stocker l'objet `VkSwapchainKHR` :

```
VkSwapchainKHR swapChain;
```

Il ne reste plus qu'à appeler la fonction `vkCreateSwapchainKHR()` pour créer la « swap chain » :

```
if (vkCreateSwapchainKHR(device, &createInfo, nullptr, &swapChain) != VK_SUCCESS) {
    throw std::runtime_error("Échec de création de la swap chain !");
}
```

La fonction accepte comme paramètre le périphérique logique, les informations de création de la « swap chain », l'allocateur optionnel et la variable pour stocker la « swap chain » créée. Cet objet devra être explicitement détruit à l'aide de la fonction `vkDestroySwapchainKHR()` avant de détruire le périphérique logique :

```
void cleanup() {
    vkDestroySwapchainKHR(device, swapChain, nullptr);
    ...
}
```

Lancez l'application et contemplez la création de la « swap chain » ! Si vous obtenez une erreur de violation d'accès dans la fonction `vkCreateSwapchainKHR()` ou que vous obtenez un message similaire à `Failed to find 'vkGetInstanceProcAddress' in layer SteamOverlayVulkanLayer.dll`, allez voir [la FAQ à propos de la surcouche Steam](#).

Essayez de retirer la ligne `createInfo.imageExtent = extent;` tout en ayant les couches de validation actives. Vous verrez que l'une d'entre elles remontera l'erreur et un message vous sera envoyé :

```
validation layer: vkCreateSwapchainKHR() called with pCreateInfo->imageExtent = (0,0), which is not equal to the currentExtent = (800,600) returned by vkGetPhysicalDeviceSurfaceCapabilitiesKHR().
```

IV-B-2-f - Récupération des images de la « swap chain »

La « swap chain » est enfin créée. Il nous faut maintenant obtenir les images (**VkImage**) contenues dedans. Nous les utiliserons lors du rendu que nous verrons dans les chapitres suivants. Ajoutez un membre à la classe pour les stocker :

```
std::vector<VkImage> swapChainImages;
```

Ces images ont été créées par Vulkan lors de la création de la « swap chain » et elles seront automatiquement supprimées avec sa destruction. Nous n'aurons donc rien à rajouter dans la fonction `cleanup()`.

J'ajoute le code de récupération des images à la fin de la fonction `createSwapChain()`, juste après l'appel à `vkCreateSwapchainKHR()`. L'obtention des références est très similaire à ce que nous avons déjà fait lorsque nous avons récupéré un tableau d'objets provenant de Vulkan. Souvenez-vous que nous avons seulement spécifié un nombre minimal d'images dans la « swap chain ». L'implémentation peut en avoir créé plus. C'est pourquoi nous commençons par récupérer le nombre d'images avec la fonction `vkGetSwapchainImagesKHR()` avant de redimensionner le conteneur pour enfin obtenir les références.

```
vkGetSwapchainImagesKHR(device, swapChain, &imageCount, nullptr);
swapChainImages.resize(imageCount);
vkGetSwapchainImagesKHR(device, swapChain, &imageCount, swapChainImages.data());
```

Une dernière chose : conservez le format et la zone d'échange de la « swap chain ». Nous en aurons besoin dans de futurs chapitres.

```
VkSwapchainKHR swapChain;
std::vector<VkImage> swapChainImages;
VkFormat swapChainImageFormat;
VkExtent2D swapChainExtent;

...
swapChainImageFormat = surfaceFormat.format;
swapChainExtent = extent;
```

Nous avons maintenant un ensemble d'images sur lesquelles nous pouvons dessiner et qui peuvent être envoyées à la fenêtre. Dans le prochain chapitre, nous verrons comment utiliser ces images comme cibles de rendu, puis nous verrons le pipeline graphique et les commandes d'affichage !

Code C++

IV-B-3 - Vues d'image

Afin d'utiliser, dans le pipeline de rendu, n'importe quelle image (**VkImage**), notamment celles provenant de la « swap chain », nous devons créer une vue (**VkImageView**). Cette vue est littéralement une vue dans l'image. Elle décrit comment accéder à l'image et quelle partie accéder. Par exemple, elle indique si elle doit être traitée comme une texture de profondeur 2D sans aucun niveau de mipmapping.

Dans ce chapitre, nous écrirons une fonction `createImageViews()` pour créer une vue pour chaque image de « swap chain ». Ainsi, nous pourrons les utiliser comme cibles pour les couleurs plus tard.

Ajoutez d'abord un membre à la classe pour y stocker les vues :

```
std::vector<VkImageView> swapChainImageViews;
```

Créez la fonction `createImageViews()` et appelez-la juste après la création de la « swap chain ».

```
void initVulkan() {
    createInstance();
    setupDebugMessenger();
    createSurface();
    pickPhysicalDevice();
    createLogicalDevice();
    createSwapChain();
    createImageViews();
}

void createImageViews() {
}
```

Nous devons d'abord redimensionner la liste pour pouvoir y mettre toutes les vues que nous créerons :

```
void createImageViews() {
    swapChainImageViews.resize(swapChainImages.size());
}
```

Ensuite, insérez la boucle qui parcourra toutes les images de la « swap chain ».

```
for (size_t i = 0; i < swapChainImages.size(); i++) {
```

La structure `VkImageViewCreateInfo` permet d'indiquer les paramètres pour la création des vues. Les premiers paramètres sont assez simples :

```
VkImageViewCreateInfo createInfo{};
createInfo.sType = VK_STRUCTURE_TYPE_IMAGE_VIEW_CREATE_INFO;
createInfo.image = swapChainImages[i];
```

Les propriétés `viewType` et `format` indiquent la manière dont les données de l'image doivent être interprétées. Le paramètre `viewType` permet de traiter les images comme des textures 1D, 2D, 3D ou des cube maps.

```
createInfo.viewType = VK_IMAGE_VIEW_TYPE_2D;
createInfo.format = swapChainImageFormat;
```

La propriété `components` vous permet de réorganiser les canaux de couleur. Par exemple, vous pouvez envoyer tous les canaux au canal rouge pour obtenir une texture monochrome. Vous pouvez aussi utiliser des valeurs constantes entre 0 et 1 à un canal. Dans notre cas nous garderons les paramètres par défaut.

```
createInfo.components.r = VK_COMPONENT_SWIZZLE_IDENTITY;
createInfo.components.g = VK_COMPONENT_SWIZZLE_IDENTITY;
createInfo.components.b = VK_COMPONENT_SWIZZLE_IDENTITY;
createInfo.components.a = VK_COMPONENT_SWIZZLE_IDENTITY;
```

La propriété `subresourceRange` décrit le but de l'image et quelle partie doit être accédée. Notre image sera utilisée comme cible de couleur et n'aura ni mipmapping ni plusieurs couches.

```
createInfo.subresourceRange.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
createInfo.subresourceRange.baseMipLevel = 0;
createInfo.subresourceRange.levelCount = 1;
createInfo.subresourceRange.baseArrayLayer = 0;
createInfo.subresourceRange.layerCount = 1;
```

Si vous travailliez sur une application 3D stéréoscopique, vous devriez alors créer une « swap chain » avec plusieurs couches. Vous pourriez alors créer plusieurs vues pour chaque image et utiliser les couches pour accéder à ce qui sera affiché pour l'œil gauche et l'œil droit.

Il ne reste plus qu'à appeler la fonction **vkCreateImageView()** pour créer la vue :

```
if (vkCreateImageView(device, &CreateInfo, nullptr, &swapChainImageViews[i]) != VK_SUCCESS) {
    throw std::runtime_error("Echec de création des vues d'image !");
}
```

À la différence des images, nous avons créé les vues nous-mêmes et nous devons donc les détruire grâce à une boucle :

```
void cleanup() {
    for (auto imageView : swapChainImageViews) {
        vkDestroyImageView(device, imageView, nullptr);
    }

    ...
}
```

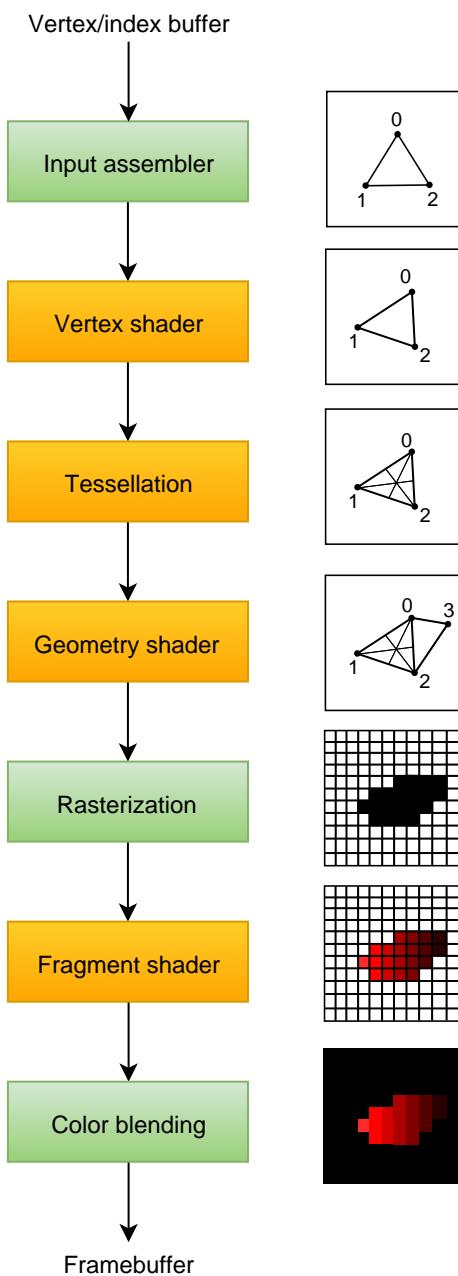
Une vue suffit pour commencer à utiliser une image comme une texture, mais pas pour que l'image soit utilisée comme cible de rendu. Pour cela, nous avons encore une étape, appelée tampon d'image. Mais d'abord, nous devons mettre en place le pipeline graphique.

Code C++

IV-C - Bases du pipeline graphique

IV-C-1 - Introduction

Dans les chapitres qui suivent, nous allons configurer un pipeline graphique pour dessiner notre premier triangle. Le pipeline graphique est l'ensemble des opérations qui, à partir des sommets et des textures de vos éléments, produisent les pixels qui sont écrits dans les cibles de rendu. En voici, un résumé simplifié :



L'assembleur des entrées (*input assembler*) collecte les données brutes des sommets à partir des tampons que vous avez configurés. Il est aussi possible d'utiliser un tampon d'indices pour répéter certains éléments sans avoir à les dupliquer dans les tampons.

Le *vertex shader* est exécuté pour chaque sommet. Généralement, il effectue des transformations pour que les coordonnées des sommets passent de l'espace modèle (model space) à l'espace écran (screen space). Il peut aussi transférer des données à la suite du pipeline.

Les *tesselation shaders* permettent de subdiviser la géométrie selon des règles paramétrables afin d'améliorer la qualité du modèle. Ce procédé est notamment utilisé pour augmenter le relief de certaines surfaces comme les murs de briques ou les escaliers.

Le *geometry shader* est exécuté pour chaque primitive (triangle, ligne, points...). Il peut en supprimer ou en créer à la volée. Ce travail est similaire à celui du tessellation shader tout en étant beaucoup plus flexible. Il n'est cependant pas beaucoup utilisé à cause de ses performances médiocres (sauf avec les GPU intégrés d'Intel).

L'étage de *rastérisation* (*ou matricialisation*) transforme les primitives en *fragments*. Pour cela, la carte graphique détermine les pixels du tampon d'image (framebuffer) qui sont à l'intérieur des primitives. Tout fragment en dehors de l'écran est abandonné. Les attributs sortant du vertex shader sont interpolés pour déterminer les valeurs propres aux fragments. Les fragments cachés par d'autres fragments peuvent être éliminés à cette étape grâce au test de profondeur (depth testing).

Le *fragment shader* est exécuté pour chaque fragment valide et détermine dans quel tampon d'image écrire. Le fragment shader détermine aussi quelle couleur et quelle valeur de profondeur écrire. Il réalise ce travail à l'aide des données interpolées émises par le vertex shader, notamment les coordonnées de texture et les normales permettant d'effectuer les calculs d'éclairage.

L'étape de mélange des couleurs (*color blending*) applique des opérations pour mixer différents fragments correspondant à un même pixel sur le tampon d'image. Les fragments peuvent écraser les valeurs précédentes, s'additionner ou se mélanger selon leur transparence.

Les étapes en vert sur le diagramme sont des étapes fixes. Il est possible d'en modifier les paramètres influençant les calculs, mais pas de modifier les calculs eux-mêmes.

Les étapes colorées en orange sont programmables, ce qui signifie que vous pouvez charger votre propre code dans la carte graphique et faire exactement ce que vous voulez. Par exemple, vous pouvez utiliser les fragment shaders pour implémenter n'importe quoi : utiliser des textures, mettre en place les effets de lumières ou encore, faire du lancer de rayon (*ray tracing*). Ces programmes sont exécutés en parallèle sur de nombreux cœurs pour traiter de nombreux modèles, sommets et fragments rapidement.

Si vous avez utilisé d'anciennes bibliothèques comme OpenGL ou Direct3D, vous êtes habitués à pouvoir changer les paramètres du pipeline à tout moment, avec des fonctions comme `glBlendFunc()` ou `OMSSetBlendState()`. Cela n'est plus possible avec Vulkan. Le pipeline graphique y est quasiment fixé et vous devrez en recréer un si vous voulez changer de shader, y attacher différents tampons d'image ou changer la fonction de mélange des couleurs. Le désavantage est qu'il est maintenant nécessaire de créer un grand nombre de pipelines afin de gérer toutes les combinaisons d'états que vous souhaitez utiliser. Par contre, comme le pilote connaît à l'avance tous les pipelines possibles, il peut donc effectuer de meilleures optimisations.

Certaines étapes programmables sont optionnelles suivant ce que vous voulez faire. Par exemple, la tessellation et le geometry shader peuvent être désactivés lorsque vous dessinez des géométries simples. Si vous n'êtes intéressé que par les valeurs de profondeur, vous pouvez désactiver le fragment shader, ce qui est utile pour créer les **textures d'ombre**.

Dans le prochain chapitre, nous allons d'abord créer les deux éléments nécessaires à l'affichage d'un triangle à l'écran : le vertex shader et le fragment shader. Les étapes fixes seront mises en place dans le chapitre d'après. La dernière préparation nécessaire à la mise en place du pipeline graphique Vulkan sera de fournir les tampons d'image en entrée et sortie.

Créez la fonction `createGraphicsPipeline()` et appelez-la depuis `initVulkan()` après l'appel à la fonction `createImageViews()`. Nous travaillerons sur cette fonction dans les chapitres suivants.

```

void initVulkan() {
    createInstance();
    setupDebugMessenger();
    createSurface();
    pickPhysicalDevice();
    createLogicalDevice();
    createSwapChain();
    createImageViews();
    createGraphicsPipeline();
}

...

void createGraphicsPipeline() {

```

{}

Code C++

IV-C-2 - Module de shaders

À la différence des bibliothèques précédentes, le code des shaders doit être fourni à Vulkan sous la forme d'un code intermédiaire (bytecode) et non sous une forme facilement compréhensible par l'homme, tel que le **GLSL** ou le **HLSL**. Ce code intermédiaire est appelé **SPIR-V**, il est conçu pour fonctionner avec Vulkan et OpenCL (deux bibliothèques de Khronos). Le code SPIR-V peut servir à écrire des shaders pour les graphiques ou pour du calcul. Évidemment, dans ce tutoriel, nous nous concentrerons sur les shaders à destination du pipeline graphique de Vulkan.

L'avantage d'utiliser un code intermédiaire est de réduire la complexité du code d'interprétation des shaders des pilotes graphiques. En effet, les constructeurs ne proposaient pas la même interprétation du langage GLSL. Par conséquent, il était possible d'écrire du code fonctionnant sur les cartes d'un constructeur donné, mais pour lequel vous obteniez des erreurs de syntaxe avec d'autres constructeurs. Pire, vous pouviez avoir des comportements différents. Avec un format de code intermédiaire comme le SPIR-V, ces problèmes tendent à être évités.

Cependant, cela ne veut pas dire que nous devrons écrire ce code intermédiaire à la main. Khronos a même fourni son propre compilateur, indépendant des constructeurs, permettant de transformer le GLSL en SPIR-V. Ce compilateur standard vérifie que votre code correspond à la spécification et produit un binaire SPIR-V à embarquer dans votre programme. Vous pouvez également l'inclure comme une bibliothèque pour produire du SPIR-V au cours de l'exécution du programme. Mais nous ne le ferons pas dans ce tutoriel. Le compilateur de Khronos s'appelle glslangValidator, mais nous allons utiliser glslc qui lui est développé par Google. L'avantage de ce dernier est qu'il utilise des paramètres semblables aux compilateurs GCC et Clang tout en ajoutant quelques fonctionnalités comme les inclusions. Les deux compilateurs sont présents dans le SDK, vous n'avez donc rien de plus à télécharger.

Le GLSL est un langage possédant une syntaxe proche du C. Les programmes doivent posséder une fonction `main()`, invoquée pour chaque objet à traiter. Plutôt que d'utiliser des paramètres et des valeurs de retour, le GLSL utilise des variables globales pour les entrées et les sorties. Le langage possède des fonctionnalités spécifiques à la programmation graphique, telles que des primitives pour les vecteurs et les matrices. De plus, le langage fournit les fonctions pour réaliser des produits en croix, des multiplications de matrices ainsi que des fonctionnalités de manipulation de vecteurs. Le vecteur est implémenté grâce aux types `vec`, suivi par un chiffre indiquant le nombre d'éléments compris dans celui-ci. Par exemple, pour une position 3D, nous utiliserons le type `vec3`. Il est possible d'accéder à chacun des éléments du vecteur comme on le ferait avec une structure. Par exemple, pour la coordonnée sur l'axe des X, on utilise `.x`. Il est aussi possible de créer un nouveau vecteur à partir de plusieurs composants d'un autre vecteur en une fois. Plus précisément, l'expression `vec3(1.0, 2.0, 3.0).xy` permet d'obtenir un `vec2`. Les constructeurs des vecteurs peuvent aussi bien accepter des vecteurs que des valeurs scalaires. Par exemple, vous pouvez construire un `vec3` avec la syntaxe `vec3(vec2(1.0, 2.0), 3.0)`.

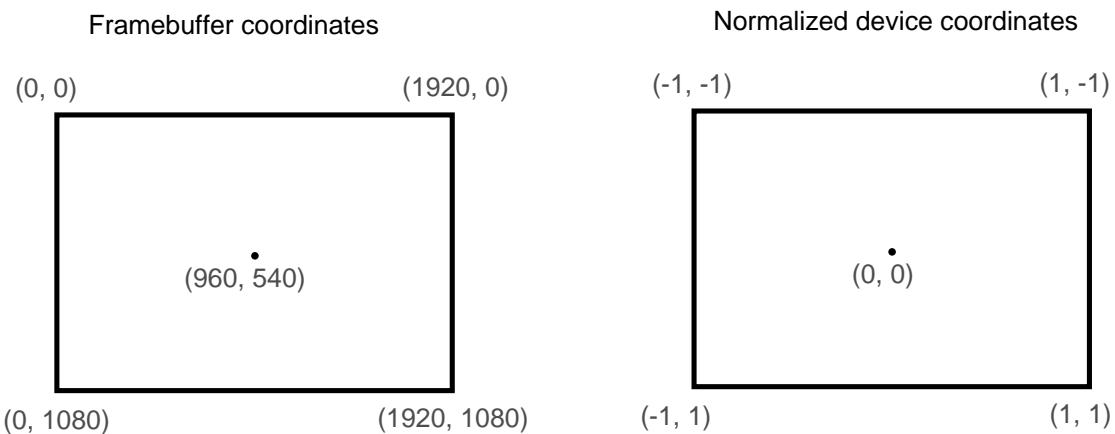
Comme nous l'avons dit au chapitre précédent, nous devons écrire un vertex shader et un fragment shader pour pouvoir afficher un triangle à l'écran. Les deux prochaines sections couvriront ce travail, puis nous verrons comment créer les binaires SPIR-V correspondants et les charger dans le programme.

IV-C-2-a - Le vertex shader

Le vertex shader traite chaque sommet fourni en entrée. Il récupère des attributs telles la position, la couleur, la normale ou les coordonnées de texture. En sortie, le vertex shader fournit une position en coordonnées dans l'espace de clipping et les attributs à envoyer au fragment shader telles que la couleur ou les coordonnées de texture. Ces valeurs seront interpolées lors de la rastérisation afin de produire un dégradé continu, puis elles seront passées au fragment shader.

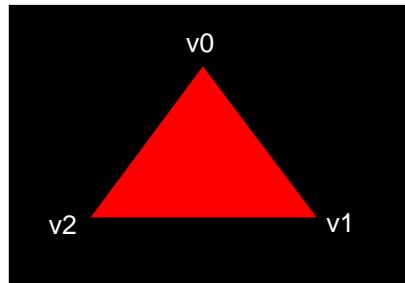
Une *coordonnée dans l'espace de clipping* est un vecteur à quatre éléments émis par le vertex shader. Il est ensuite transformé en une coordonnée normalisée grâce à la division de ses composants par le dernier d'entre eux. Les

coordonnées normalisées sont des **coordonnées homogènes** qui s'étalent sur le tampon d'image grâce à un système de coordonnées compris dans l'intervalle [-1, 1]. Il ressemble à cela :



Vous devriez déjà être familier avec ces notions si vous avez déjà manipulé des graphismes 3D. Si vous avez utilisé OpenGL avant, vous vous rendrez compte que l'axe Y est maintenant inversé et que l'axe Z va de 0 à 1, comme avec Direct3D.

Pour notre premier triangle, nous n'appliquerons aucune transformation, nous nous contenterons de spécifier directement les coordonnées normalisées des trois sommets pour créer la forme suivante :



Nous pouvons directement envoyer des coordonnées normalisées en tant que coordonnées dans l'espace de découpage en assignant le dernier composant à 1. Ainsi, la division transformant les coordonnées dans l'espace de découpage en coordonnées normalisées ne modifie pas les valeurs.

Normalement, ces coordonnées proviennent d'un tampon de sommets, mais la création et le remplissage de ce dernier ne sont pas des opérations triviales avec Vulkan. J'ai donc décidé de retarder ce sujet afin d'obtenir un résultat satisfaisant plus rapidement. Par conséquent, nous allons faire quelque chose de peu orthodoxe en attendant : inclure les coordonnées directement dans le vertex shader. Son code ressemble donc à ceci :

```
#version 450

vec2 positions[3] = vec2[] (
    vec2(0.0, -0.5),
    vec2(0.5, 0.5),
```

```

        vec2(-0.5, 0.5)
};

void main() {
    gl_Position = vec4(positions[gl_VertexIndex], 0.0, 1.0);
}

```

La fonction `main()` est appelée pour chaque sommet. La variable prédéfinie `gl_VertexIndex` contient l'indice du sommet en cours de traitement. Généralement, c'est l'indice du tampon de sommets, mais dans ce cas, nous l'utilisons comme indice pour nos données en dur. La position de chaque sommet est récupérée à partir du tableau constant du shader et complétée avec des valeurs factices pour produire les coordonnées dans l'espace de découpage. La variable `gl_Position` correspond à la sortie.

IV-C-2-b - Le fragment shader

Le triangle formé par les positions émises par le vertex shader colorie une zone de l'écran grâce aux nombreux fragments compris dedans. Le fragment shader est invoqué pour chacun d'entre eux et produit une couleur et une profondeur pour le tampon d'image (ou les tampons d'image). Un fragment shader dont le but est de renvoyer la couleur rouge pour l'intégralité de l'écran s'écrit ainsi :

```

#version 450
#extension GL_ARB_separate_shader_objects : enable

layout(location = 0) out vec4 outColor;

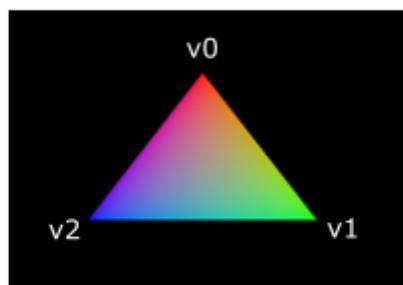
void main() {
    outColor = vec4(1.0, 0.0, 0.0, 1.0);
}

```

La fonction `main()` est appelée pour chaque fragment de la même manière que le vertex shader est appelé pour chaque sommet. Les couleurs sont des vecteurs de quatre composants : R, G, B et le canal alpha. Les valeurs sont comprises dans l'intervalle [0, 1]. Au contraire de `gl_Position`, il n'y a pas de variable prédéfinie dans laquelle écrire la valeur de la couleur. Vous devrez spécifier votre propre variable de sortie pour chaque tampon d'image. La syntaxe `layout(location = 0)` indique l'indice du tampon d'image où la couleur sera écrite. Ici, la couleur rouge est écrite dans la variable `outColor` liée au premier (et unique) tampon d'image ayant pour indice 0.

IV-C-2-c - Une couleur pour chaque sommet

Avoir un triangle complètement rouge n'est pas vraiment intéressant. N'aimez-vous pas avoir un triangle comme celui-ci ?



Nous devons pour cela faire quelques petits changements aux deux shaders. Spécifions d'abord une couleur distincte pour chaque sommet. Le vertex shader doit donc embarquer un tableau avec les couleurs, tout comme cela a été fait pour les positions :

```
vec3 colors[3] = vec3[] (
    vec3(1.0, 0.0, 0.0),
    vec3(0.0, 1.0, 0.0),
    vec3(0.0, 0.0, 1.0)
);
```

Nous devons maintenant envoyer ces couleurs au fragment shader afin qu'il puisse écrire les valeurs interpolées dans le tampon d'image. Ajoutez une variable de sortie au vertex shader pour la couleur et définissez sa valeur dans la fonction `main()` :

```
layout(location = 0) out vec3 fragColor;

void main() {
    gl_Position = vec4(positions[gl_VertexIndex], 0.0, 1.0);
    fragColor = colors[gl_VertexIndex];
}
```

Nous devons ensuite ajouter l'entrée correspondante dans le fragment shader :

```
layout(location = 0) in vec3 fragColor;

void main() {
    outColor = vec4(fragColor, 1.0);
}
```

La variable d'entrée n'a pas besoin d'avoir le même nom. Les sorties du vertex shader et les entrées du fragment shader sont liées par les indices spécifiés dans les directives `location`. La fonction `main()` doit être modifiée pour émettre la couleur provenant du vertex shader tout en lui ajoutant un canal alpha. Les valeurs de `fragColor` sont automatiquement interpolées pour obtenir les valeurs des fragments entre les trois sommets. Ainsi, et comme le montre l'image précédente, cela donne un dégradé uniforme.

IV-C-2-d - Compilation des shaders

Créez un dossier `shaders` à la racine de votre projet et enregistrez-y le vertex shader dans un fichier appelé `shader.vert` et le fragment shader dans un fichier appelé `shader.frag`. Les shaders en GLSL n'ont pas d'extension officielle, mais celles-ci sont habituellement utilisées pour les reconnaître.

Le contenu de `shader.vert` devrait être :

```
#version 450
#extension GL_ARB_separate_shader_objects : enable

out gl_PerVertex {
    vec4 gl_Position;
};

layout(location = 0) out vec3 fragColor;

vec2 positions[3] = vec2[] (
    vec2(0.0, -0.5),
    vec2(0.5, 0.5),
    vec2(-0.5, 0.5)
);

vec3 colors[3] = vec3[] (
    vec3(1.0, 0.0, 0.0),
    vec3(0.0, 1.0, 0.0),
    vec3(0.0, 0.0, 1.0)
```

```
);

void main() {
    gl_Position = vec4(positions[gl_VertexIndex], 0.0, 1.0);
    fragColor = colors[gl_VertexIndex];
}
```

Et `shader.frag` devrait contenir :

```
#version 450
#extension GL_ARB_separate_shader_objects : enable

layout(location = 0) in vec3 fragColor;

layout(location = 0) out vec4 outColor;

void main() {
    outColor = vec4(fragColor, 1.0);
}
```

Nous allons maintenant compiler ces shaders en code intermédiaire SPIR-V à l'aide du programme `glslc`.

IV-C-2-d-i - Windows

Créez un fichier `compile.bat` et copiez ceci dedans :

```
C:/VulkanSDK/x.x.x.x/Bin32/glslc.exe shader.vert -o vert.spv
C:/VulkanSDK/x.x.x.x/Bin32/glslc.exe shader.frag -o frag.spv
pause
```

Correz le chemin menant à `glslc.exe` avec le chemin où vous avez installé le SDK Vulkan. Double-cliquez dessus pour lancer ce script.

IV-C-2-d-ii - Linux

Créez un fichier `compile.sh` et copiez ceci dedans :

```
/home/user/VulkanSDK/x.x.x.x/x86_64/bin/glslc shader.vert -o vert.spv
/home/user/VulkanSDK/x.x.x.x/x86_64/bin/glslc shader.frag -o frag.spv
```

Correz le chemin menant à `glslc` pour correspondre au chemin de votre installation du SDK Vulkan. Rendez le script exécutable avec la commande `chmod +x compile.sh` et lancez-le.

IV-C-2-d-iii - Notes

Ces deux commandes demandent au compilateur de lire le code GLSL source contenu dans un fichier et d'écrire le code SPIR-V correspondant dans un fichier grâce à l'option `-o` (output).

Si votre shader contient une erreur de syntaxe, le compilateur vous indiquera la ligne et le problème, comme vous pouvez vous y attendre. Essayez de retirer un point-virgule et relancez le script. Aussi, essayez de lancer le compilateur sans arguments afin de voir la liste des options supportées. Par exemple, il est possible de produire un code intermédiaire lisible par un humain afin de comprendre exactement ce que fait le shader et les optimisations qui y ont été appliquées.

La compilation des shaders en ligne de commande est l'une des solutions les plus simples et c'est celle que nous utilisons dans ce tutoriel. Il est aussi possible de compiler les shaders dans votre code. Le SDK Vulkan inclut la bibliothèque **libshaderc** permettant de compiler le GLSL en SPIR-V directement depuis votre programme.

IV-C-2-e - Charger un shader

Maintenant que vous pouvez créer des shaders SPIR-V, il est temps de les charger dans le programme et de les intégrer au pipeline graphique. Nous allons d'abord écrire une fonction qui réalisera le chargement des données binaires à partir des fichiers.

```
#include <fstream>

...

static std::vector<char> readFile(const std::string& filename) {
    std::ifstream file(filename, std::ios::ate | std::ios::binary);

    if (!file.is_open()) {
        throw std::runtime_error("Échec d'ouverture du fichier !");
    }
}
```

La fonction `readFile()` lira tous les octets du fichier spécifié et les retournera dans un tableau d'octets géré par `std::vector`. Nous spécifions les deux options suivantes lors de l'ouverture du fichier :

- `ate`: permet de commencer la lecture à la fin du fichier ;
- `binary` : indique que le fichier lu est binaire (pour éviter les transformations spécifiques au texte).

En commençant la lecture par la fin du fichier, il est possible d'utiliser la position pour récupérer la taille du fichier pour allouer le tampon :

```
size_t fileSize = (size_t) file.tellg();
std::vector<char> buffer(fileSize);
```

Après cela, nous revenons au début du fichier et lisons tous les octets en une fois :

```
file.seekg(0);
file.read(buffer.data(), fileSize);
```

Enfin, nous pouvons fermer le fichier et renvoyer les octets :

```
file.close();

return buffer;
```

Appelons maintenant cette fonction depuis la fonction `createGraphicsPipeline()` pour charger le code intermédiaire des deux shaders :

```
void createGraphicsPipeline() {
    auto vertShaderCode = readFile("shaders/vert.spv");
    auto fragShaderCode = readFile("shaders/frag.spv");
}
```

Assurez-vous que les shaders sont correctement chargés en affichant la taille des fichiers lus depuis votre programme, puis en comparant cette valeur à la taille des fichiers indiquée par le système d'exploitation. Notez que le code n'a pas besoin d'utiliser un terminateur, car c'est du code binaire et que nous allons indiquer sa taille.

IV-C-2-f - Créer des modules de shader

Avant de passer ce code au pipeline, nous devons l'incorporer dans un objet **VkShaderModule**. Créez pour cela une fonction nommée `createShaderModule`.

```
VkShaderModule createShaderModule(const std::vector<char>& code) {
}
```

Cette fonction prendra comme paramètre le tampon contenant le code intermédiaire et créera un **VkShaderModule** à partir de celui-ci.

La création d'un module de shader est simple. Nous devons simplement indiquer un pointeur vers le tampon et la taille de celui-ci. Ces informations seront inscrites dans la structure **VkShaderModuleCreateInfo**. Le seul problème est que la taille du code intermédiaire doit être en octets, mais le pointeur sur le code intermédiaire est du type `uint32_t` et non du type `char`. Nous devons donc utiliser `reinterpret_cast` sur notre pointeur. Lors d'une telle transcription de pointeur, il faut s'assurer que les données sont compatibles avec l'alignement nécessaire pour `uint32_t`. Heureusement pour nous, l'allocateur par défaut d'un `std::vector` assure que les données stockées remplissent les conditions, même dans les pires cas.

```
VkShaderModuleCreateInfo createInfo{};
createInfo.sType = VK_STRUCTURE_TYPE_SHADER_MODULE_CREATE_INFO;
createInfo.codeSize = code.size();
createInfo.pCode = reinterpret_cast<const uint32_t*>(code.data());
```

L'objet **VkShaderModule** peut alors être créé en appelant la fonction **vkCreateShaderModule()** :

```
VkShaderModule shaderModule;
if (vkCreateShaderModule(device, &createInfo, nullptr, &shaderModule) != VK_SUCCESS) {
    throw std::runtime_error("Échec de création du module de shader !");
}
```

Les paramètres sont les mêmes que pour la création des objets précédents : le périphérique logique, le pointeur sur la structure contenant les informations, un pointeur optionnel vers l'allocateur et une référence pour stocker l'objet créé. Le tampon contenant le code peut être libéré immédiatement après la création du module. Enfin, renvoyez le module de shader créé :

```
return shaderModule;
```

Les modules de shader ne sont réellement qu'une fine couche autour du code intermédiaire chargé depuis les fichiers. La compilation et la liaison du code intermédiaire SPIR-V vers un code machine prêt à être exécuté par le GPU ne se font qu'au moment de la création du pipeline graphique. Cela signifie que vous pouvez détruire les modules de shader dès que la création du pipeline est finie. Pour cette raison, nous gardons les modules dans des variables locales dans la fonction `createGraphicsPipeline()` :

```
void createGraphicsPipeline() {
    auto vertShaderCode = readFile("shaders/vert.spv");
    auto fragShaderCode = readFile("shaders/frag.spv");

    VkShaderModule vertShaderModule = createShaderModule(vertShaderCode);
    VkShaderModule fragShaderModule = createShaderModule(fragShaderCode);
```

La libération doit être placée à la fin de la fonction grâce à deux appels à la fonction **vkDestroyShaderModule()**. Le reste du code de ce chapitre sera à ajouter avant ces deux lignes.

```
...
vkDestroyShaderModule(device, fragShaderModule, nullptr);
vkDestroyShaderModule(device, vertShaderModule, nullptr);
}
```

IV-C-2-g - Création des étapes programmables

Pour utiliser les shaders, nous devons les assigner à l'étape programmable du pipeline grâce à la structure **VkPipelineShaderStageCreateInfo**. Cette structure fait partie du processus de création du pipeline.

Nous allons d'abord remplir cette structure pour le vertex shader. Nous le faisons dans la fonction `createGraphicsPipeline()`.

```
VkPipelineShaderStageCreateInfo vertShaderStageInfo{};  
vertShaderStageInfo.sType = VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;  
vertShaderStageInfo.stage = VK_SHADER_STAGE_VERTEX_BIT;
```

La première étape, sans compter la propriété `sType`, consiste à dire à Vulkan dans quelle étape le shader sera utilisé. Il existe une énumération décrivant les étapes vues dans le chapitre précédent.

```
vertShaderStageInfo.module = vertShaderModule;  
vertShaderStageInfo.pName = "main";
```

Les deux propriétés suivantes indiquent le module contenant le code et la fonction à invoquer : le point d'entrée. Il est donc possible de combiner plusieurs fragment shaders dans un même module et les différencier à l'aide de leurs points d'entrée. Dans notre cas, nous nous contenterons de la fonction `main()` classique.

Il existe un autre membre, optionnel, appelé `pSpecializationInfo`, que nous n'utiliserons pas, mais qu'il est intéressant d'évoquer. Il vous permet de spécifier les valeurs des constantes du shader. Vous pouvez utiliser un seul module de shader et modifier son comportement à l'aide de constantes définies lors de la création du pipeline. C'est plus efficace que d'utiliser des variables définies lors du rendu, car le compilateur peut ainsi effectuer des optimisations, notamment supprimer les blocs d'un `if` inutilisés. Si vous n'avez pas de constante, alors vous pouvez définir ce membre à `nullptr`, valeur déjà définie par l'initialisation de la structure.

La modification de la structure pour la faire correspondre au fragment shader est simple :

```
VkPipelineShaderStageCreateInfo fragShaderStageInfo{};  
fragShaderStageInfo.sType = VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;  
fragShaderStageInfo.stage = VK_SHADER_STAGE_FRAGMENT_BIT;  
fragShaderStageInfo.module = fragShaderModule;  
fragShaderStageInfo.pName = "main";
```

Enfin, définissez un tableau contenant ces deux structures. Nous les utiliserons lors de la création du pipeline.

```
VkPipelineShaderStageCreateInfo shaderStages[] = {vertShaderStageInfo, fragShaderStageInfo};
```

Nous avons vu comment configurer les étapes programmables du pipeline. Dans le prochain chapitre, nous verrons les étapes fixes.

Code C++ / Vertex shader / Fragment shader

IV-C-3 - Étapes fixes

Les anciennes bibliothèques définissaient des configurations par défaut pour la plupart des étapes du pipeline graphique. Avec Vulkan, vous devez décrire l'intégralité du pipeline, du viewport aux fonctions de mélange de couleurs. Dans ce chapitre, nous remplirons toutes les structures nécessaires à la configuration des fonctionnalités fixes.

IV-C-3-a - Les sommets en entrée

La structure `VkPipelineVertexInputStateCreateInfo` décrit le format des données des sommets envoyés au vertex shader. Globalement, cela se fait en deux étapes :

- liens (bindings) : espacement entre les données et indication permettant de savoir si les données sont par sommet ou par instance (pour **l'instanciation**) ;

- descriptions des attributs : décrit le type des attributs passés au vertex shader, à partir de quel lien charger les données et à quelle position (offset).

Pour le moment et comme nous avons écrit en dur les données des sommets dans le vertex shader, nous allons remplir cette structure pour indiquer qu'il n'y a pas de données de sommets. Nous reviendrons sur cette structure dans le chapitre sur les tampons de sommets.

```
VkPipelineVertexInputStateCreateInfo vertexInputInfo{};
vertexInputInfo.sType = VK_STRUCTURE_TYPE_PIPELINE_VERTEX_INPUT_STATE_CREATE_INFO;
vertexInputInfo.vertexBindingDescriptionCount = 0;
vertexInputInfo.pVertexBindingDescriptions = nullptr; // Optionnel
vertexInputInfo.vertexAttributeDescriptionCount = 0;
vertexInputInfo.pVertexAttributeDescriptions = nullptr; // Optionnel
```

Les membres `pVertexBindingDescriptions` et `pVertexAttributeDescriptions` pointent vers un tableau de structures décrivant les détails du chargement des données des sommets. Ajoutez cette structure à la fonction `createGraphicsPipeline()` juste après le tableau `shaderStages`.

IV-C-3-b - Assembleur d'entrée

La structure `VkPipelineInputAssemblyStateCreateInfo` décrit deux choses : quel type de géométrie dessiner et si la réévaluation des sommets doit être activée. La première information est décrite dans le membre `topology` et peut prendre ces valeurs :

- `VK_PRIMITIVE_TOPOLOGY_POINT_LIST` : chaque sommet correspond à un point ;
- `VK_PRIMITIVE_TOPOLOGY_LINE_LIST` : dessine une ligne en utilisant les sommets deux par deux ;
- `VK_PRIMITIVE_TOPOLOGY_LINE_STRIP` : le dernier sommet de chaque ligne est utilisé comme premier sommet pour la ligne suivante ;
- `VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST` : dessine un triangle en utilisant les sommets trois par trois ;
- `VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP` : les deuxième et troisième sommets sont utilisés comme les deux premiers pour le triangle suivant.

Normalement, les sommets sont chargés en utilisant les indices séquentiellement depuis le tampon de sommets. En utilisant un tampon d'éléments, vous pouvez spécifier vous-même les indices. Vous pouvez ainsi réaliser des optimisations, notamment en réutilisant les sommets plusieurs fois. Si vous mettez le membre `primitiveRestartEnable` à la valeur `VK_TRUE`, il devient alors possible d'interrompre les lignes ou triangles lors de l'utilisation d'un mode `_STRIP` grâce aux valeurs `0xFFFF` ou `0xFFFFFFFF`.

Nous n'afficherons que des triangles dans ce tutoriel, nous nous contenterons donc de remplir la structure de cette manière :

```
VkPipelineInputAssemblyStateCreateInfo inputAssembly{};
inputAssembly.sType = VK_STRUCTURE_TYPE_PIPELINE_INPUT_ASSEMBLY_STATE_CREATE_INFO;
inputAssembly.topology = VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST;
inputAssembly.primitiveRestartEnable = VK_FALSE;
```

IV-C-3-c - Viewports et découpage

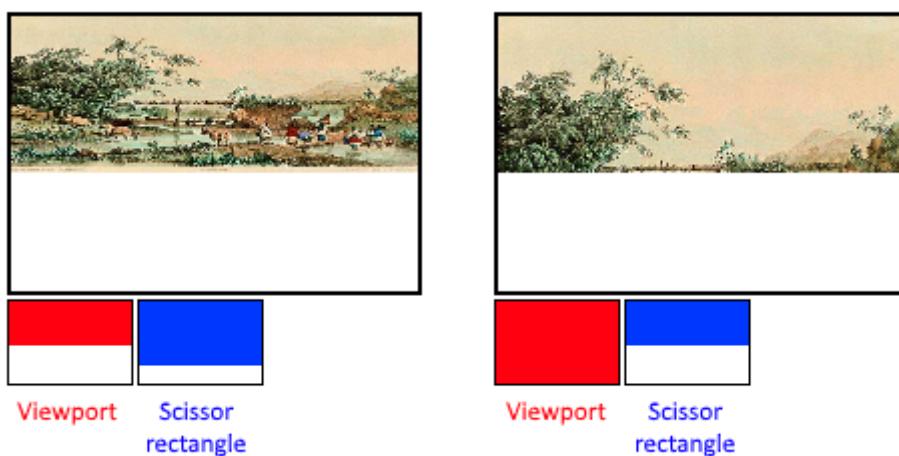
Un viewport décrit la région d'un tampon d'image sur laquelle le rendu sera effectué. Généralement, il démarre de `(0, 0)` et a une taille correspondant à la largeur et la hauteur de l'image. Cela est le cas dans ce tutoriel.

```
VkViewport viewport{};
viewport.x = 0.0f;
viewport.y = 0.0f;
viewport.width = (float) swapChainExtent.width;
viewport.height = (float) swapChainExtent.height;
viewport.minDepth = 0.0f;
viewport.maxDepth = 1.0f;
```

N'oubliez pas que la taille des images de la « swap chain » peut différer de celle des macros `WIDTH` et `HEIGHT`. Les images de la « swap chain » seront utilisées comme tampon d'image plus tard, nous devons donc utiliser leur taille.

Les valeurs `minDepth` et `maxDepth` indiquent les valeurs minimales et maximales pour la profondeur dans le tampon d'image. Ces valeurs doivent être comprises dans l'intervalle `[0.0f, 1.0f]`, mais `minDepth` peut être supérieure à `maxDepth`. Si vous ne faites rien de particulier, contentez-vous des valeurs classiques `0.0f` et `1.0f`.

Alors que les viewports définissent la transformation de l'image vers le tampon d'image, les rectangles de découpage (scissors) définissent la région de pixels qui sera conservée. Tout pixel en dehors de ces rectangles sera éliminé par le rastériseur. Leur fonctionnement ressemble plus à un filtre qu'à une transformation. La différence est montrée ci-dessous. Notez que le rectangle de découpage pour l'image de gauche est l'une des possibilités permettant d'obtenir une telle image. L'image sera la même tant que le rectangle est plus grand que le viewport.



Dans ce tutoriel, nous voulons dessiner sur la totalité du tampon d'image. Nous utilisons donc un rectangle de découpage couvrant l'intégralité du tampon d'image :

```
VkRect2D scissor{};
scissor.offset = {0, 0};
scissor.extent = swapChainExtent;
```

Le viewport et le rectangle de découpage se combinent en un état de `viewport` à l'aide de la structure `VkPipelineViewportStateCreateInfo`. Sur certaines cartes graphiques, il est possible d'utiliser plusieurs viewports et rectangles de découpage. C'est pourquoi la structure accepte des tableaux pour ces deux données. L'utilisation de cette possibilité nécessite de l'activer lors de la création du périphérique logique.

```
VkPipelineViewportStateCreateInfo viewportState{};
viewportState.sType = VK_STRUCTURE_TYPE_PIPELINE_VIEWPORT_STATE_CREATE_INFO;
viewportState.viewportCount = 1;
viewportState.pViewports = &viewport;
viewportState.scissorCount = 1;
viewportState.pScissors = &scissor;
```

IV-C-3-d - Rastériseur

Le rastériseur récupère la géométrie formée par des sommets provenant du vertex shader et les transforme en fragments qui seront traités par le fragment shader. Il réalise également le **test de profondeur**, la **suppression des faces** et le test de découpage et peut être configuré pour retourner des fragments couvrant l'intégralité de la géométrie ou juste les bords (rendu en fil de fer). Tout cela se configure dans la structure `VkPipelineRasterizationStateCreateInfo`.

```
VkPipelineRasterizationStateCreateInfo rasterizer{};
```

```
rasterizer.sType = VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_STATE_CREATE_INFO;
rasterizer.depthClampEnable = VK_FALSE;
```

Si le membre `depthClampEnable` est `VK_TRUE`, les fragments au-delà des plans proche et lointain sont fixés aux valeurs des plans et non plus supprimés. Cela peut être pratique lors du rendu des textures d'ombrage (shadow maps). Cette fonctionnalité nécessite d'activer une fonctionnalité du GPU.

```
rasterizer.rasterizerDiscardEnable = VK_FALSE;
```

Si le membre `rasterizerDiscardEnable` est `VK_TRUE`, aucune géométrie ne passe l'étape du rastériseur. En clair, cela désactive tout rendu dans le tampon d'image.

```
rasterizer.polygonMode = VK_POLYGON_MODE_FILL
```

Le propriété `polygonMode` définit comment les fragments sont générés à partir de la géométrie. Vulkan vous donne accès aux modes suivants :

- `VK_POLYGON_MODE_FILL` : remplit les polygones de fragments ;
- `VK_POLYGON_MODE_LINE` : les côtés des polygones sont dessinés comme des lignes ;
- `VK_POLYGON_MODE_POINT` : les sommets des polygones sont dessinés comme des points.

Tout autre mode que `FILL` nécessite d'activer une fonctionnalité GPU.

```
rasterizer.lineWidth = 1.0f;
```

La propriété `lineWidth` définit l'épaisseur des lignes en termes de fragments. La taille maximale supportée dépend du GPU et toute autre valeur que `1.0f` nécessite l'activation de la fonctionnalité GPU `wideLines`.

```
rasterizer.cullMode = VK_CULL_MODE_BACK_BIT;
rasterizer.frontFace = VK_FRONT_FACE_CLOCKWISE;
```

La propriété `cullMode` détermine quelles faces seront supprimées lors de l'étape de suppression des faces. Vous pouvez désactiver toute suppression, n'éliminer que les faces avant, que celles de derrière ou éliminer toutes les faces. La propriété `frontFace` indique l'ordre d'évaluation des sommets permettant de dire si la face est avant ou arrière : dans le sens des aiguilles d'une montre ou l'opposé.

```
rasterizer.depthBiasEnable = VK_FALSE;
rasterizer.depthBiasConstantFactor = 0.0f; // Optionnel
rasterizer.depthBiasClamp = 0.0f; // Optionnel
rasterizer.depthBiasSlopeFactor = 0.0f; // Optionnel
```

Le rastériseur peut altérer la profondeur en y ajoutant une valeur constante ou en la modifiant selon l'inclinaison du fragment. Ces possibilités sont parfois exploitées pour la génération des textures d'ombrage, mais nous ne les utiliserons pas. Laissez `depthBiasEnabled` à la valeur `VK_FALSE`.

IV-C-3-e - Multiéchantillonnage

La structure **`VkPipelineMultisampleCreateInfo`** permet la configuration du multiéchantillonnage (multisampling), une des méthodes pour effectuer de l'anticrénelage (**anti-aliasing**). Cette méthode combine les résultats du fragment shader de plusieurs polygones dessinant le même pixel. Cela se produit notamment pour les bordures, endroit où le crénage est le plus présent. Comme il n'est pas utile d'exécuter plusieurs fois le fragment shader lorsqu'il n'y a qu'un polygone associé à un pixel, c'est bien moins coûteux que d'effectuer un rendu de plus haute résolution et de le redimensionner. Activer le multiéchantillonnage nécessite l'activation d'une fonctionnalité GPU.

```
VkPipelineMultisampleStateCreateInfo multisampling{};
multisampling.sType = VK_STRUCTURE_TYPE_PIPELINE_MULTISAMPLE_STATE_CREATE_INFO;
multisampling.sampleShadingEnable = VK_FALSE;
```

```

multisampling.rasterizationSamples = VK_SAMPLE_COUNT_1_BIT;
multisampling.minSampleShading = 1.0f; // Optionnel
multisampling.pSampleMask = nullptr; // Optionnel
multisampling.alphaToCoverageEnable = VK_FALSE; // Optionnel
multisampling.alphaToOneEnable = VK_FALSE; // Optionnel
    
```

Nous reverrons le multisampling plus tard, pour l'instant laissez-le désactivé.

IV-C-3-f - Tests de profondeur et de pochoir

Si vous utilisez un tampon de profondeur (depth buffer) et/ou de pochoir (stencil buffer) vous devez configurer les tests de profondeur et de pochoir avec la structure **VkPipelineDepthStencilCreateInfo**. Nous n'avons aucun de ces tampons, donc nous indiquerons `nullptr` à la place du pointeur vers une telle structure. Nous y reviendrons au chapitre sur le tampon de profondeur.

IV-C-3-g - Mélange de couleurs

La couleur renvoyée par un fragment shader doit être combinée avec la couleur déjà présente dans le tampon d'image. Cette opération s'appelle mélange de couleurs (color blending) et peut être réalisée de deux façons :

- mélanger l'ancienne et la nouvelle couleur pour créer la couleur finale ;
- combiner l'ancienne et la nouvelle couleur à l'aide d'une opération bit à bit.

Il y a deux types de structures pour configurer le mélange de couleurs. La première, **VkPipelineColorBlendAttachmentState**, contient une configuration pour chaque tampon d'image et la seconde, **VkPipelineColorBlendStateCreateInfo**, contient les paramètres globaux pour le mélange de couleurs. Dans notre cas, nous n'avons qu'un tampon d'image :

```

VkPipelineColorBlendAttachmentState colorBlendAttachment{};
colorBlendAttachment.colorWriteMask = VK_COLOR_COMPONENT_R_BIT | VK_COLOR_COMPONENT_G_BIT |
    VK_COLOR_COMPONENT_B_BIT | VK_COLOR_COMPONENT_A_BIT;
colorBlendAttachment.blendEnable = VK_FALSE;
colorBlendAttachment.srcColorBlendFactor = VK_BLEND_FACTOR_ONE; // Optionnel
colorBlendAttachment.dstColorBlendFactor = VK_BLEND_FACTOR_ZERO; // Optionnel
colorBlendAttachment.colorBlendOp = VK_BLEND_OP_ADD; // Optionnel
colorBlendAttachment.srcAlphaBlendFactor = VK_BLEND_FACTOR_ONE; // Optionnel
colorBlendAttachment.dstAlphaBlendFactor = VK_BLEND_FACTOR_ZERO; // Optionnel
colorBlendAttachment.alphaBlendOp = VK_BLEND_OP_ADD; // Optionnel
    
```

Cette structure spécifique à chaque tampon d'image vous permet de configurer la première méthode pour effectuer un mélange de couleurs. L'opération effectuée ressemblera à ce pseudocode :

```

if (blendEnable) {
    finalColor.rgb = (srcColorBlendFactor * newColor.rgb) <colorBlendOp> (dstColorBlendFactor *
    oldColor.rgb);
    finalColor.a = (srcAlphaBlendFactor * newColor.a) <alphaBlendOp> (dstAlphaBlendFactor *
    oldColor.a);
} else {
    finalColor = newColor;
}

finalColor = finalColor & colorWriteMask;
    
```

Si `blendEnable` est `VK_FALSE` la nouvelle couleur du fragment shader est écrite telle qu'elle. Sinon, les deux opérations de mélange sont exécutées pour calculer la nouvelle couleur. Le résultat est combiné avec un ET binaire et la propriété `colorWriteMask` détermine quel canal est conservé.

L'utilisation la plus commune pour mélanger les couleurs est d'utiliser le canal alpha pour déterminer l'opacité du matériau et donc le mélange lui-même. La couleur finale devrait alors être calculée ainsi :

```
finalColor.rgb = newAlpha * newColor + (1 - newAlpha) * oldColor;
finalColor.a = newAlpha.a;
```

Cela peut être réalisé avec les paramètres suivants :

```
colorBlendAttachment.blendEnable = VK_TRUE;
colorBlendAttachment.srcColorBlendFactor = VK_BLEND_FACTOR_SRC_ALPHA;
colorBlendAttachment.dstColorBlendFactor = VK_BLEND_FACTOR_ONE_MINUS_SRC_ALPHA;
colorBlendAttachment.colorBlendOp = VK_BLEND_OP_ADD;
colorBlendAttachment.srcAlphaBlendFactor = VK_BLEND_FACTOR_ONE;
colorBlendAttachment.dstAlphaBlendFactor = VK_BLEND_FACTOR_ZERO;
colorBlendAttachment.alphaBlendOp = VK_BLEND_OP_ADD;
```

Vous pouvez trouver toutes les opérations possibles dans la spécification aux sections `VkBlendFactor` et `VkBlendOp`.

La seconde structure possède un tableau de structures pour tous les tampons d'image et vous permet de définir des valeurs constantes permettant d'altérer le calcul du mélange de couleurs vu précédemment.

```
VkPipelineColorBlendStateCreateInfo colorBlending{};
colorBlending.sType = VK_STRUCTURE_TYPE_PIPELINE_COLOR_BLEND_STATE_CREATE_INFO;
colorBlending.logicOpEnable = VK_FALSE;
colorBlending.logicOp = VK_LOGIC_OP_COPY; // Optionnel
colorBlending.attachmentCount = 1;
colorBlending.pAttachments = &colorBlendAttachment;
colorBlending.blendConstants[0] = 0.0f; // Optionnel
colorBlending.blendConstants[1] = 0.0f; // Optionnel
colorBlending.blendConstants[2] = 0.0f; // Optionnel
colorBlending.blendConstants[3] = 0.0f; // Optionnel
```

Si vous voulez utiliser la seconde méthode de mélange de couleurs (la combinaison bit à bit), vous devez indiquer `VK_TRUE` au membre `logicOpEnable`. L'opération est spécifiée dans la propriété `logicOp`. En activant ce mode, la première méthode sera désactivée et vous devez mettre `VK_FALSE` dans tous les champs du tampon d'image attaché. La propriété `colorWriteMask` sera également utilisée dans ce mode pour déterminer les canaux affectés. Il est aussi possible de désactiver les deux modes comme nous l'avons fait ici. Dans ce cas, les résultats du fragment shader seront directement écrits dans le tampon d'image.

IV-C-3-h - États dynamiques

Un petit nombre d'états que nous avons spécifiés dans les structures précédentes peuvent être modifiés sans devoir recréer le pipeline. On y trouve la taille du viewport, la largeur des lignes et les constantes du mélange de couleurs. Pour cela, vous devrez remplir la structure `VkPipelineDynamicStateCreateInfo` comme suit :

```
VkDynamicState dynamicStates[] = {
    VK_DYNAMIC_STATE_VIEWPORT,
    VK_DYNAMIC_STATE_LINE_WIDTH
};

VkPipelineDynamicStateCreateInfo dynamicState{};
dynamicState.sType = VK_STRUCTURE_TYPE_PIPELINE_DYNAMIC_STATE_CREATE_INFO;
dynamicState.dynamicStateCount = 2;
dynamicState.pDynamicStates = dynamicStates;
```

Les valeurs données lors de la configuration seront ignorées et vous devrez fournir les données nécessaires au moment du rendu. Nous y reviendrons plus tard. Cette structure peut être remplacée par `nullptr` si vous ne voulez pas utiliser d'état dynamique.

IV-C-3-i - Agencement du pipeline

Vous pouvez utiliser des variables uniformes dans les shaders : ce sont des données globales similaires aux états dynamiques que vous pouvez modifier lors du rendu pour modifier le comportement des shaders sans avoir à les

recréer. Elles sont généralement utilisées pour passer les matrices de transformation au vertex shader ou pour créer les échantillonneurs de texture dans les fragment shader.

Les variables uniformes doivent être spécifiées lors de la création du pipeline en créant un objet **VkPipelineLayout**. Même si nous n'en utilisons pas dans nos shaders actuels, nous devons créer un agencement de pipeline vide.

Créez un membre pour stocker la structure, car nous en aurons besoin plus tard.

```
VkPipelineLayout pipelineLayout;
```

Créons maintenant l'objet dans la fonction `createGraphicsPipeline()` :

```
VkPipelineLayoutCreateInfo pipelineLayoutInfo{};
pipelineLayoutInfo.sType = VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;
pipelineLayoutInfo.setLayoutCount = 0; // Optionnel
pipelineLayoutInfo.pSetLayouts = nullptr; // Optionnel
pipelineLayoutInfo.pushConstantRangeCount = 0; // Optionnel
pipelineLayoutInfo.pPushConstantRanges = nullptr; // Optionnel

if (vkCreatePipelineLayout(device, &pipelineLayoutInfo, nullptr, &pipelineLayout) != VK_SUCCESS) {
    throw std::runtime_error("Échec lors de la création de l'agencement du pipeline !");
}
```

Cette structure indique aussi les constantes poussées (push constants), une autre manière de passer des valeurs dynamiques aux shaders que nous verrons dans un futur chapitre. L'agencement du pipeline sera utilisé pendant toute la durée du programme, nous devons donc le détruire dans la fonction `cleanup()` :

```
void cleanup() {
    vkDestroyPipelineLayout(device, pipelineLayout, nullptr);
    ...
}
```

IV-C-3-j - Conclusion

Voilà tout ce qu'il y a à savoir sur les étapes fixes ! Leur configuration représente un gros travail, mais en contrepartie, nous connaissons l'intégralité de ce qui se passe dans le pipeline graphique ! Ainsi, les risques d'un comportement inattendu lié à une valeur par défaut d'une étape du pipeline sont diminués.

Il reste cependant encore un objet à créer avant de pouvoir créer le pipeline graphique : la **passe de rendu**.

Code C++ / Vertex shader / Fragment shader

IV-C-4 - Passes de rendu

IV-C-4-a - Mise en place

Avant de finaliser la création du pipeline, nous devons indiquer à Vulkan les attaches au tampon d'image utilisées lors du rendu. Nous devons indiquer combien de tampons de couleurs et de profondeur il y aura et combien d'échantillons nous allons utiliser pour chacun d'eux, ainsi que la façon dont le contenu sera géré au travers des opérations de rendu. Toutes ces informations sont contenues dans un objet appelé passe de rendu (render pass) que nous allons créer dans une fonction `createRenderPass()`. Appelez cette fonction depuis `initVulkan()` avant `createGraphicsPipeline()`.

```
void initVulkan() {
    createInstance();
    setupDebugMessenger();
    createSurface();
```

```

        pickPhysicalDevice();
        createLogicalDevice();
        createSwapChain();
        createImageViews();
        createRenderPass();
        createGraphicsPipeline();
    }

    ...

void createRenderPass() {
}

```

IV-C-4-b - Description de l'attache

Dans notre cas, nous n'aurons qu'une seule attache pour la couleur correspondant à une image provenant de la « swap chain ».

```

void createRenderPass() {
    VkAttachmentDescription colorAttachment{};
    colorAttachment.format = swapChainImageFormat;
    colorAttachment.samples = VK_SAMPLE_COUNT_1_BIT;
}

```

Le format de l'attache de couleur doit correspondre au format des images de la « swap chain ». Pour le moment, nous n'utilisons pas de multiéchantillonnage, donc, nous devons indiquer que nous n'utilisons qu'un seul échantillon.

```

colorAttachment.loadOp = VK_ATTACHMENT_LOAD_OP_CLEAR;
colorAttachment.storeOp = VK_ATTACHMENT_STORE_OP_STORE;

```

Les propriétés `loadOp` et `storeOp` définissent respectivement ce qui doit être fait avec les données de l'attache avant et après le rendu. Pour `loadOp`, nous avons les choix suivants :

- `VK_ATTACHMENT_LOAD_OP_LOAD` : conserve les données présentes dans l'attache ;
- `VK_ATTACHMENT_LOAD_OP_CLEAR` : redéfinit, au commencement, les valeurs à une constante ;
- `VK_ATTACHMENT_LOAD_OP_DONT_CARE` : le contenu existant est non défini et nous n'en avons rien à faire.

Dans notre cas, nous utiliserons l'opération de remplacement pour obtenir un tampon d'image noir avant d'afficher une nouvelle image. Quant à `storeOp`, il existe deux possibilités :

- `VK_ATTACHMENT_STORE_OP_STORE` : le rendu est stocké en mémoire et peut être lu par la suite ;
- `VK_ATTACHMENT_STORE_OP_DONT_CARE` : le contenu du tampon d'image est indéfini après l'opération de rendu.

Comme nous voulons voir le triangle à l'écran, nous allons utiliser l'opération de stockage.

```

colorAttachment.stencilLoadOp = VK_ATTACHMENT_LOAD_OP_DONT_CARE;
colorAttachment.stencilStoreOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;

```

Les propriétés `loadOp` et `storeOp` s'appliquent aux données de couleur et de profondeur. De même, il existe les propriétés `stencilLoadOp` et `stencilStoreOp` pour les données de pochoir. Notre application n'utilise pas de tampon de pochoir, nous indiquons que les opérations de chargement et de sauvegarde ne nous sont pas utiles.

```

colorAttachment.initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
colorAttachment.finalLayout = VK_IMAGE_LAYOUT_PRESENT_SRC_KHR;

```

Les textures et les tampons d'image dans Vulkan sont représentés par des objets de type **VkImage** comprenant un format de pixels donné. Cependant, l'agencement des pixels dans la mémoire peut changer selon ce que vous faites de cette image.

Les agencements les plus communs sont :

- **VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL** : images utilisées comme attache de couleur ;
- **VK_IMAGE_LAYOUT_PRESENT_SRC_KHR** : images qui seront envoyées à la « swap chain » ;
- **VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL** : images utilisées comme destination d'une opération de copie en mémoire.

Nous discuterons plus précisément de ce sujet dans le chapitre sur les textures. Pour le moment, il faut juste retenir que les images doivent avoir un agencement adéquat pour les opérations dans lesquelles elles seront utilisées.

La propriété `initialLayout` spécifie l'agencement de l'image avant le début de la passe de rendu. La propriété `finalLayout` indique l'agencement que l'image doit avoir à la fin de la passe de rendu. La propriété `initialLayout` peut avoir pour valeur **VK_IMAGE_LAYOUT_UNDEFINED** permettant de spécifier que l'agencement précédent ne nous intéresse pas. En contrepartie, il n'y a aucune garantie que le contenu de l'image soit conservé. Mais ce n'est pas un problème puisque, de toute façon, nous effaçons toutes les données avant le rendu. Après le rendu, nous souhaitons envoyer l'image à la « swap chain », nous utilisons donc **VK_IMAGE_LAYOUT_PRESENT_SRC_KHR** pour la propriété `finalLayout`.

IV-C-4-c - Sous-passes et références aux attaches

Une passe de rendu est composée de plusieurs sous-passes (subpasses). Les sous-passes sont des opérations de rendu qui dépendent du contenu présent dans le tampon d'image fourni par les passes précédentes. Un exemple concret est l'implémentation d'une série d'effets de post-traitement qui se suivent. En regroupant toutes ces opérations en une seule passe, Vulkan peut alors réorganiser les sous-passes, réduire l'utilisation de la bande passante et ainsi être plus performant. Pour notre premier triangle, nous nous contenterons d'une seule sous-passe.

Chaque sous-passe référence une ou plusieurs attaches que nous avons définies au travers des structures vues dans la section précédente. Ces références sont représentées par le type **VkAttachmentReference** et ressemblent à cela :

```
VkAttachmentReference colorAttachmentRef{};

colorAttachmentRef.attachment = 0;
colorAttachmentRef.layout = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL;
```

Le paramètre `attachment` spécifie l'attache à référencer à l'aide d'un indice correspondant à sa position dans le tableau de descriptions des attaches. Notre tableau ne consistera qu'en une seule référence de type **VkAttachmentDescription**, donc son indice est 0. La propriété `layout` indique l'agencement que doit avoir l'attache pendant l'exécution de la sous-passe l'utilisant. Vulkan changera automatiquement l'agencement de l'attache au début de la sous-passe. Nous souhaitons que l'attache soit un tampon de couleur avec l'agencement **VK_IMAGE_LAYOUT_COLOR_OPTIMAL**. Comme son nom le suggère, cela nous donnera de meilleures performances.

La sous-passe est décrite dans la structure **VkSubpassDescription** :

```
VkSubpassDescription subpass{};

subpass.pipelineBindPoint = VK_PIPELINE_BIND_POINT_GRAPHICS;
```

Dans le futur, Vulkan pourra supporter des sous-passes de calcul. Par conséquent, nous devons indiquer que notre sous-passe est une sous-passe graphique. Ensuite, nous spécifions la référence à l'attache de couleur :

```
subpass.colorAttachmentCount = 1;
subpass.pColorAttachments = &colorAttachmentRef;
```

L'indice de cette attache est directement mentionné dans le fragment shader avec la directive `layout(location = 0) out vec4 outColor;`

Les types d'attaches suivants peuvent être référencés dans une sous passe :

- `pInputAttachments` : attaches lues depuis un shader ;
- `pResolveAttachments` : attaches utilisées pour le multiéchantillonnage des attaches de couleur ;
- `pDepthStencilAttachment` : attaches pour la profondeur et le pochoir ;
- `pPreserveAttachments` : attaches qui ne sont pas utilisées par cette sous-passe, mais dont les données doivent être conservées.

IV-C-4-d - Passe de rendu

Maintenant que l'attache et qu'une sous-passe la référençant ont été mises en place, nous pouvons créer la passe de rendu. Créez une nouvelle variable du type `VkRenderPass` au-dessus de la variable `pipelineLayout` :

```
VkRenderPass renderPass;  
VkPipelineLayout pipelineLayout;
```

L'objet représentant la passe de rendu peut être créé en remplissant la structure `VkRenderPassCreateInfo` avec un tableau d'attaches et un tableau de sous passes. Les objets `VkAttachmentReference` réfèrent les attaches en utilisant les indices de ce tableau.

```
VkRenderPassCreateInfo renderPassInfo{};  
renderPassInfo.sType = VK_STRUCTURE_TYPE_RENDER_PASS_CREATE_INFO;  
renderPassInfo.attachmentCount = 1;  
renderPassInfo.pAttachments = &colorAttachment;  
renderPassInfo.subpassCount = 1;  
renderPassInfo.pSubpasses = &subpass;  
  
if (vkCreateRenderPass(device, &renderPassInfo, nullptr, &renderPass) != VK_SUCCESS) {  
    throw std::runtime_error("Échec lors de la création de la passe de rendu!");  
}
```

Comme pour l'agencement du pipeline, nous aurons à utiliser la passe de rendu tout au long du programme. Nous devons donc la détruire dans la fonction `cleanup()` :

```
void cleanup() {  
    vkDestroyPipelineLayout(device, pipelineLayout, nullptr);  
    vkDestroyRenderPass(device, renderPass, nullptr);  
    ...  
}
```

Nous avons eu beaucoup de travail. Dans le prochain chapitre, nous allons assembler le tout et créer le pipeline graphique !

Code C++ / Vertex shader / Fragment shader

IV-C-5 - Conclusion

Nous pouvons maintenant combiner toutes les structures et tous les objets des chapitres précédents pour créer le pipeline graphique ! Voici un petit récapitulatif des objets que nous avons :

- étapes programmables : les modules de shader qui définissent le fonctionnement des étapes programmables du pipeline graphique ;
- étapes fixes : plusieurs structures qui paramètrent les étapes fixes comme l'assemblage des entrées, le rastérisateur, le viewport et le mélange des couleurs ;

- l'agencement du pipeline : les valeurs uniformes et les constantes poussées utilisées par les shaders et pouvant être modifiées lors du rendu ;
- la passe de rendu : les attaches référencées par le pipeline et leur utilisation.

Tous ces éléments rassemblés permettent de définir le fonctionnement du pipeline graphique. Nous pouvons donc maintenant remplir la structure **VkGraphicsPipelineCreateInfo** à la fin de la fonction `createGraphicsPipeline()`, toutefois, nous devons remplir cette structure avant l'appel à la fonction **vkDestroyShaderModule()**, car cette dernière détruit les shaders dont le pipeline a besoin.

```
VkGraphicsPipelineCreateInfo pipelineInfo{};
pipelineInfo.sType = VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_CREATE_INFO;
pipelineInfo.stageCount = 2;
pipelineInfo.pStages = shaderStages;
```

Commençons par référencer le tableau de **VkPipelineShaderStageCreateInfo**.

```
pipelineInfo.pVertexInputState = &vertexInputInfo;
pipelineInfo.pInputAssemblyState = &inputAssembly;
pipelineInfo.pViewportState = &viewportState;
pipelineInfo.pRasterizationState = &rasterizer;
pipelineInfo.pMultisampleState = &multisampling;
pipelineInfo.pDepthStencilState = nullptr; // Optionnel
pipelineInfo.pColorBlendState = &colorBlending;
pipelineInfo.pDynamicState = nullptr; // Optionnel
```

Puis spécifions les structures décrivant les étapes fixes.

```
pipelineInfo.layout = pipelineLayout;
```

Après cela vient l'agencement du pipeline. Ici, il faut passer la structure par copie plutôt que par un pointeur.

```
pipelineInfo.renderPass = renderPass;
pipelineInfo.subpass = 0;
```

Finalement, nous avons une référence à la passe de rendu, ainsi que l'indice de la sous-passe dans laquelle le pipeline sera utilisé. Il est aussi possible d'utiliser d'autres passes de rendu avec le pipeline de cette instance, mais elles doivent toutes être *compatibles* avec `renderPass`. Cette notion de compatibilité est décrite [ici](#), mais nous n'utiliserons pas cette possibilité dans ce tutoriel.

```
pipelineInfo.basePipelineHandle = VK_NULL_HANDLE; // Optionnel
pipelineInfo.basePipelineIndex = -1; // Optionnel
```

En réalité, il nous reste deux paramètres : `basePipelineHandle` et `basePipelineIndex`. Vulkan vous permet de créer un nouveau pipeline à partir d'un pipeline existant. L'idée derrière cette fonctionnalité est qu'il est moins coûteux de créer un pipeline à partir d'un autre partageant la plupart des fonctionnalités et le changement d'un pipeline à un autre peut être réalisé plus rapidement lorsqu'ils ont le même parent. Vous pouvez spécifier un pipeline de deux manières : soit en fournissant une référence à un pipeline existant avec `basePipelineHandle`, soit en donnant l'indice d'un pipeline qui va être créé avec `basePipelineIndex`. Pour le moment, nous n'avons qu'un seul pipeline. Nous passons donc un pointeur nul et un index invalide. Ces valeurs ne sont utilisées que si le champ `VK_PIPELINE_CREATE_DERIVATIVE_BIT` est spécifié dans la propriété `flags` de la structure **VkGraphicsPipelineCreateInfo**.

Préparons l'étape finale en créant une variable membre où stocker l'objet de type **VkPipeline** :

```
VkPipeline graphicsPipeline;
```

Finalement, créons le pipeline graphique :

```
if (vkCreateGraphicsPipelines(device,
    VK_NULL_HANDLE, 1, &pipelineInfo, nullptr, &graphicsPipeline) != VK_SUCCESS) {
```

```
        throw std::runtime_error("Échec pour créer le pipeline graphique !");
}
```

La fonction **vkCreateGraphicsPipelines()** possède plus de paramètres que les fonctions de création d'objets que nous avons pu voir jusqu'à présent. Elle peut en effet accepter plusieurs structures **VkGraphicsPipelineCreateInfo** et créer plusieurs **VkPipeline** en un seul appel.

Le second paramètre, pour lequel nous passons **VK_NULL_HANDLE**, référence un objet **VkPipelineCache** optionnel. Un cache de pipelines peut être utilisé pour stocker et réutiliser des données utiles lors des différents appels à **VkCreateGraphicsPipelines()** et même entre plusieurs exécutions du programme si le cache est sauvegardé dans un fichier. Cela permet de grandement accélérer la création. Nous verrons ce cas dans un chapitre dédié.

Le pipeline graphique est nécessaire lors des opérations de dessin, nous devons donc le détruire à la fin du programme :

```
void cleanup() {
    vkDestroyPipeline(device, graphicsPipeline, nullptr);
    vkDestroyPipelineLayout(device, pipelineLayout, nullptr);
    ...
}
```

Exécutez votre programme pour vérifier que tout ce travail a permis la création d'un pipeline graphique sans erreur ! Nous sommes de plus en plus proches d'obtenir quelque chose à l'écran ! Dans les prochains chapitres, nous configurerons les tampons d'image à partir des images de la « swap chain » et préparerons les commandes de dessin.

Code C++ / Vertex shader / Fragment shader

IV-D - Rendu

IV-D-1 - Tampons d'images

Nous avons beaucoup parlé de tampons d'images dans les chapitres précédents et nous avons mis en place une passe de rendu qui attend un tampon d'images avec le même format que les images provenant de la « swap chain ». Toutefois, nous ne l'avons toujours pas créé.

Les attaches spécifiées durant la création de la passe de rendu sont liées en les associant à un objet de type **VkFramebuffer**. Cet objet référence toutes les vues (**VkImageView**) correspondant aux attaches. Par contre, l'image que nous devons utiliser comme attache dépend de l'image retournée par la « swap chain » lorsque nous en récupérons une pour l'affichage. Cela signifie que nous devons créer un tampon d'images pour toutes les images de la « swap chain » et utiliser le tampon qui correspond à l'image reçue au moment du rendu.

Pour cela, créons un autre `std::vector` membre de la classe et qui contiendra les tampons d'image :

```
std::vector<VkFramebuffer> swapChainFramebuffers;
```

Les objets qui rempliront ce tableau seront créés dans une nouvelle fonction nommée `createFramebuffers()` que nous appellerons depuis la fonction `initVulkan()` juste après la création du pipeline graphique :

```
void initVulkan() {
    createInstance();
    setupDebugMessenger();
    createSurface();
    pickPhysicalDevice();
    createLogicalDevice();
    createSwapChain();
    createImageViews();
    createRenderPass();
    createGraphicsPipeline();
```

```

        createFramebuffers();
    }

    ...

void createFramebuffers() {
}

```

Commencez par redimensionner le conteneur afin qu'il puisse stocker tous les tampons d'image :

```

void createFramebuffers() {
    swapChainFramebuffers.resize(swapChainImageViews.size());
}

```

Nous allons maintenant itérer sur toutes les vues d'images et créer un tampon d'images à partir de chacune d'elles :

```

for (size_t i = 0; i < swapChainImageViews.size(); i++) {
    VkImageView attachments[] = {
        swapChainImageViews[i]
    };

    VkFramebufferCreateInfo framebufferInfo{};
    framebufferInfo.sType = VK_STRUCTURE_TYPE_FRAMEBUFFER_CREATE_INFO;
    framebufferInfo.renderPass = renderPass;
    framebufferInfo.attachmentCount = 1;
    framebufferInfo.pAttachments = attachments;
    framebufferInfo.width = swapChainExtent.width;
    framebufferInfo.height = swapChainExtent.height;
    framebufferInfo.layers = 1;

    if (vkCreateFramebuffer(device, &framebufferInfo, nullptr, &swapChainFramebuffers[i]) != VK_SUCCESS) {
        throw std::runtime_error("Échec lors de la création du tampon d'image !");
    }
}

```

Comme vous le pouvez le voir, la création d'un tampon d'images est assez simple. Nous devons d'abord indiquer avec quelle passe de rendu le tampon d'images doit être compatible. Vous ne pouvez utiliser qu'un tampon d'images qui est **compatible** avec la passe de rendu, c'est-à-dire que le tampon et la passe de rendu doivent globalement avoir le même nombre d'attaches et qu'elles soient du même type.

Les paramètres `attachmentCount` et `pAttachments` spécifient les objets de type **VkImageView** qui doivent être liés aux descriptions d'attaches que nous avons spécifiées dans le tableau `pAttachment` de la passe de rendu.

Les paramètres `width` et `height` sont évidents. La propriété `layers` correspond au nombre de couches dans les tableaux d'image. Les images de notre « swap chain » sont uniques, donc nous spécifions `1`.

Nous devons détruire les tampons d'images avant les vues d'images et avant la passe de rendu, mais seulement après la fin du rendu :

```

void cleanup() {
    for (auto framebuffer : swapChainFramebuffers) {
        vkDestroyFramebuffer(device, framebuffer, nullptr);
    }

    ...
}

```

Nous avons atteint le moment où tous les objets sont prêts pour le rendu. Dans le prochain chapitre, nous allons écrire les premières commandes de rendu.

Code C++ / Vertex shader / Fragment shader

IV-D-2 - Tampons de commandes

Dans Vulkan, les commandes comme les opérations de dessin ou de transfert mémoire ne sont pas réalisées avec des appels de fonctions. Il faut enregistrer toutes les opérations dans des tampons de commandes. L'avantage est que vous pouvez faire ce travail en amont, et ce, depuis plusieurs threads. Ensuite, il ne reste plus qu'à dire à Vulkan d'exécuter les commandes dans la boucle principale.

IV-D-2-a - Groupe de commandes

Nous devons créer un groupe de commandes (command pool) avant de pouvoir créer les tampons de commandes. Les groupes gèrent la mémoire nécessaire pour stocker les tampons. Les tampons de commandes sont alloués à partir des groupes. Ajoutez un nouveau membre à la classe pour stocker une variable de type **VkCommandPool** :

```
VkCommandPool commandPool;
```

Créez ensuite la fonction nommée `createCommandPool` et appelez-la depuis la fonction `initVulkan()` après la création des tampons d'images.

```
void initVulkan() {
    createInstance();
    setupDebugMessenger();
    createSurface();
    pickPhysicalDevice();
    createLogicalDevice();
    createSwapChain();
    createImageViews();
    createRenderPass();
    createGraphicsPipeline();
    createFramebuffers();
    createCommandPool();
}

...
void createCommandPool() {
```

La création d'un groupe de commandes ne nécessite que deux paramètres :

```
QueueFamilyIndices queueFamilyIndices = findQueueFamilies(physicalDevice);

VkCommandPoolCreateInfo poolInfo{};
poolInfo.sType = VK_STRUCTURE_TYPE_COMMAND_POOL_CREATE_INFO;
poolInfo.queueFamilyIndex = queueFamilyIndices.graphicsFamily.value();
poolInfo.flags = 0; // Optionnel
```

Les tampons de commandes sont exécutés en les envoyant à une queue, telle que celles que nous avons récupérées pour les graphiques et l'affichage. Chaque groupe de commandes ne peut allouer que des tampons de commandes qui seront envoyés à un type unique de queue. Nous allons enregistrer des commandes pour le dessin, c'est pourquoi nous avons choisi la famille de queue pour les graphiques.

Il existe deux valeurs acceptées par la propriété `flags` des groupes de commandes :

- `VK_COMMAND_POOL_CREATE_TRANSIENT_BIT` : informe que les tampons de commandes sont très souvent réenregistrés avec de nouvelles commandes (le comportement de l'allocation de la mémoire peut être différent) ;
- `VK_COMMAND_POOL_CREATE_RESET_COMMAND_BUFFER_BIT` : permet aux tampons de commandes d'être réenregistrés individuellement. Sans cet indicateur, ils doivent tous être réinitialisés ensemble.

Nous n'enregistrerons les tampons de commandes qu'une seule fois, au début du programme, et les exécuterons énormément de fois dans la boucle principale. Nous n'avons donc pas besoin de ces indications.

```
if (vkCreateCommandPool(device, &poolInfo, nullptr, &commandPool) != VK_SUCCESS) {
    throw std::runtime_error("Échec de création du groupe de commandes !");
}
```

La création du groupe de commandes s'effectue avec la fonction **vkCreateComandPool()**. Elle ne possède aucun paramètre particulier. Les commandes seront utilisées tout au long du programme pour dessiner des choses à l'écran. Donc, le groupe ne doit être détruit qu'à la fin, dans la fonction cleanup() :

```
void cleanup() {
    vkDestroyCommandPool(device, commandPool, nullptr);

    ...
}
```

IV-D-2-b - Allocation des tampons de commandes

Nous pouvons maintenant allouer des tampons de commandes et y enregistrer des commandes de rendu. Dans la mesure où l'une des commandes consiste à lier le bon tampon d'image (**VkFramebuffer**), nous devons enregistrer un tampon pour chaque image de la « swap chain ». Pour cela, créez une liste d'objets de type **VkCommandBuffer** et stockez-la dans une variable membre de la classe. Les tampons de commande sont libérés automatiquement lorsque leur groupe de commandes est détruit. Nous n'avons donc pas besoin de le faire explicitement.

```
std::vector<VkCommandBuffer> commandBuffers;
```

Commençons maintenant à travailler sur une nouvelle fonction `createCommandBuffers()` qui allouera et enregistrera les tampons de commandes pour chacune des images de la « swap chain ».

```
void initVulkan() {
    createInstance();
    setupDebugMessenger();
    createSurface();
    pickPhysicalDevice();
    createLogicalDevice();
    createSwapChain();
    createImageViews();
    createRenderPass();
    createGraphicsPipeline();
    createFramebuffers();
    createCommandPool();
    createCommandBuffers();
}

...

void createCommandBuffers() {
    commandBuffers.resize(swapChainFramebuffers.size());
}
```

Les tampons de commandes sont alloués en appelant la fonction **vkAllocateCommandBuffers()** qui prend en paramètre une structure du type **VkCommandBufferAllocateInfo**. Cette structure spécifie le groupe de commandes et le nombre de tampons à allouer :

```
VkCommandBufferAllocateInfo allocInfo{};
allocInfo.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_ALLOCATE_INFO;
allocInfo.commandPool = commandPool;
allocInfo.level = VK_COMMAND_BUFFER_LEVEL_PRIMARY;
allocInfo.commandBufferCount = (uint32_t) commandBuffers.size();

if (vkAllocateCommandBuffers(device, &allocInfo, commandBuffers.data()) != VK_SUCCESS) {
```

```
        throw std::runtime_error("Échec d' allocation des tampons de commandes !");
}
```

Le paramètre level indique si les tampons de commandes alloués sont primaires ou secondaires :

- **VK_COMMAND_BUFFER_LEVEL_PRIMARY** : peut être envoyé à une queue pour y être exécuté, mais ne peut pas être appelé à partir d'autres tampons de commandes ;
- **VK_COMMAND_BUFFER_LEVEL_SECONDARY** : ne peut pas être directement envoyé à une queue, mais peut être appelé à partir d'un tampon de commandes primaire.

Nous n'utiliserons pas de tampon de commandes secondaire, mais vous pouvez voir que ce mécanisme est utile pour réutiliser des opérations communes dans des tampons de commandes primaires.

IV-D-2-c - Commencer l'enregistrement des commandes

Nous commençons l'enregistrement des commandes dans le tampon de commandes en appelant la fonction **vkBeginCommandBuffer()**. Cette fonction prend en paramètre une structure du type **VkCommandBufferBeginInfo**. Elle spécifie quelques détails sur l'utilisation du tampon de commandes.

```
for (size_t i = 0; i < commandBuffers.size(); i++) {
    VkCommandBufferBeginInfo beginInfo{};
    beginInfo.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO;
    beginInfo.flags = 0; // Optionnel
    beginInfo.pInheritanceInfo = nullptr; // Optionnel

    if (vkBeginCommandBuffer(commandBuffers[i], &beginInfo) != VK_SUCCESS) {
        throw std::runtime_error("Échec de lancement de l'enregistrement du tampon de commandes !");
    }
}
```

Le paramètre flags indique notre utilisation du tampon de commandes. Voici une liste des valeurs disponibles :

- **VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT** : le tampon de commandes sera réenregistré juste après son utilisation ;
- **VK_COMMAND_BUFFER_USAGE_RENDER_PASS_CONTINUE_BIT** : le tampon de commandes secondaire sera intégralement exécuté dans une unique passe de rendu ;
- **VK_COMMAND_BUFFER_USAGE_SIMULTANEOUS_USE_BIT** : le tampon de commandes peut être réenvoyé alors qu'il est déjà en cours d'exécution.

Nous n'avons besoin d'aucune de ces valeurs ici.

Le paramètre pInheritanceInfo n'a de sens que pour les tampons de commandes secondaires. Il indique de quel état hériter lors de l'appel par un tampon de commandes primaire.

Si un tampon de commandes a déjà été enregistré, alors un appel à la fonction **vkBeginCommandBuffer()** le réinitialisera. Il n'est pas possible d'ajouter des commandes à un tampon une fois son enregistrement fini.

IV-D-2-d - Commencer une passe de rendu

Le rendu démarre par le lancement de la passe de rendu grâce à la fonction **vkCmdBeginRenderPass()**. La passe est configurée grâce aux paramètres de la structure de type **VkRenderPassBeginInfo**.

```
VkRenderPassBeginInfo renderPassInfo{};
renderPassInfo.sType = VK_STRUCTURE_TYPE_RENDER_PASS_BEGIN_INFO;
renderPassInfo.renderPass = renderPass;
renderPassInfo.framebuffer = swapChainFramebuffers[i];
```

Les premiers paramètres indiquent la passe de rendu et les attaches à lier. Nous avons créé un tampon d'images pour chaque image de la « swap chain » spécifiant l'attache pour les couleurs.

```
renderPassInfo.renderArea.offset = {0, 0};
renderPassInfo.renderArea.extent = swapChainExtent;
```

Les deux paramètres qui suivent définissent la taille de la zone de rendu. Cette zone de rendu définit où les shaders vont charger et écrire. Les pixels hors de cette région auront une valeur non définie. Cette zone doit correspondre à la taille des attaches pour obtenir les meilleures performances.

```
VkClearColorValue clearColor = {0.0f, 0.0f, 0.0f, 1.0f};
renderPassInfo.clearValueCount = 1;
renderPassInfo.pClearValues = &clearColor;
```

Les deux derniers paramètres définissent les valeurs à utiliser pour réinitialiser le tampon lors de l'opération `VK_ATTACHMENT_LOAD_CLEAR` que nous avons mise en place lors du chargement de l'attache des couleurs. J'ai utilisé un noir complètement opaque.

```
vkCmdBeginRenderPass(commandBuffers[i], &renderPassInfo, VK_SUBPASS_CONTENTS_INLINE);
```

La passe de rendu peut maintenant commencer. Toutes les fonctions enregistrant des commandes ont le préfixe `vkCmd`. Comme elles retournent toutes `void`, nous n'avons aucun moyen de détecter une erreur tant que l'enregistrement n'est pas fini.

Le premier paramètre de chaque commande est toujours le tampon de commandes qui stockera l'appel. Le second paramètre donne des détails sur la passe de rendu que nous venons de fournir. Le dernier paramètre contrôle comment les commandes de rendu comprises dans la passe de rendu seront fournies. Nous avons deux possibilités :

- `VK_SUBPASS_CONTENTS_INLINE` : les commandes de la passe de rendu seront directement embarquées dans le tampon de commandes et aucun tampon secondaire ne sera exécuté ;
- `VK_SUBPASS_CONTENTS_SECONDARY_COMMAND_BUFFER` : les commandes de la passe de rendu seront exécutées à partir de tampons de commandes secondaires.

Nous n'utiliserons pas de tampon de commandes secondaire, nous devons donc fournir la première valeur.

IV-D-2-e - Commandes de rendu basiques

Nous pouvons maintenant lier le pipeline graphique :

```
vkCmdBindPipeline(commandBuffers[i], VK_PIPELINE_BIND_POINT_GRAPHICS, graphicsPipeline);
```

Le deuxième paramètre indique si le pipeline passé est un pipeline graphique ou de calcul. Nous avons maintenant fourni quelles opérations à exécuter dans le pipeline graphique et quelle attache utiliser dans le fragment shader. Il ne reste plus qu'à indiquer de dessiner le triangle :

```
vkCmdDraw(commandBuffers[i], 3, 1, 0, 0);
```

La fonction `vkCmdDraw()` est assez ridicule quand on sait tout ce que nous avons fait pour en arriver là. En plus du tampon de commandes, elle possède les paramètres suivants :

- `vertexCount` : même si nous n'avons pas de tampon de sommets, nous avons techniquement trois sommets à dessiner ;
- `instanceCount` : utilisé lors d'un rendu avec instanciation. Indiquez `1` si vous ne l'utilisez pas ;
- `firstVertex` : utilisé comme décalage dans le tampon de sommets et définit ainsi la valeur la plus basse pour `glVertexIndex` ;

- `firstInstance` : utilisé comme décalage pour l'instanciation et définit ainsi la valeur la plus basse pour `gl_InstanceIndex`.

IV-D-2-f - Finalisation

La passe de rendu peut ensuite être terminée :

```
vkCmdEndRenderPass(commandBuffers[i]);
```

Et nous avons fini l'enregistrement du tampon de commandes :

```
if (vkEndCommandBuffer(commandBuffers[i]) != VK_SUCCESS) {
    throw std::runtime_error("Erreur d'enregistrement du tampon de commandes !");
}
```

Dans le prochain chapitre, nous écrirons le code pour la boucle principale. Elle récupérera une image provenant de la « swap chain », exécutera le bon tampon de commandes et renverra l'image complète à la « swap chain ».

Code C++ / Vertex shader / Fragment shader

IV-D-3 - Rendu et présentation

IV-D-3-a - Mise en place

Nous en sommes au chapitre où tout s'assemble. Nous allons écrire une fonction `drawFrame()` qui sera appelée depuis la boucle principale et affichera les triangles à l'écran. Créez la fonction et appelez-la depuis la fonction `mainLoop()` :

```
void mainLoop() {
    while (!glfwWindowShouldClose(window)) {
        glfwPollEvents();
        drawFrame();
    }
}

...

void drawFrame() {
```

IV-D-3-b - Synchronisation

La fonction `drawFrame()` réalisera les opérations suivantes :

- récupérer une image depuis la « swap chain » ;
- exécuter le tampon de commandes avec l'image obtenue comme attache du tampon d'images ;
- retourner l'image à la « swap chain » pour l'afficher.

Chacune de ces actions n'est réalisée qu'avec un appel de fonction. Cependant, elles sont asynchrones. Les appels de fonction finiront avant que les opérations ne soient réellement terminées. De plus, l'ordre d'exécution des opérations n'est pas déterministe. C'est dommage, d'autant plus que chacune de ces opérations dépend de la précédente.

Il y a deux manières d'ordonnancer les événements de la « swap chain » : utiliser les barrières (fences) et les sémaphores. Ces deux types d'objets peuvent être utilisés pour coordonner les opérations. En effet, chaque opération

sera en attente du déclenchement de la barrière ou du sémaphore. Le déclenchement se produira après la fin de l'opération précédente.

Toutefois, il existe une différence entre les barrières et les sémaphores : les barrières sont les seules dont l'état peut être récupéré par le programme grâce à la fonction **vkWaitForFences()**. Les barrières sont principalement utilisées pour ordonner les actions de l'application elle-même entre les opérations de rendu alors que les sémaphores sont utilisés pour ordonner les opérations à l'intérieur ou entre les queues de commandes. Nous souhaitons agencer les opérations de rendu d'une queue et de l'affichage, nous utiliserons donc des sémaphores qui correspondent mieux à ce besoin.

IV-D-3-c - Sémaphores

Nous avons besoin d'un sémaphore pour indiquer qu'une image est prête pour le rendu et d'un deuxième sémaphore pour signaler que le rendu est terminé et que l'affichage peut avoir lieu. Créez donc deux membres de la classe pour stocker ces objets :

```
VkSemaphore imageAvailableSemaphore;
VkSemaphore renderFinishedSemaphore;
```

Pour créer les sémaphores, nous allons ajouter une dernière fonction nommée `createSemaphores()` dans la fonction `initVulkan()` :

```
void initVulkan() {
    createInstance();
    setupDebugMessenger();
    createSurface();
    pickPhysicalDevice();
    createLogicalDevice();
    createSwapChain();
    createImageViews();
    createRenderPass();
    createGraphicsPipeline();
    createFramebuffers();
    createCommandPool();
    createCommandBuffers();
    createSemaphores();
}

...

void createSemaphores() {
```

La création d'un sémaphore nécessite de renseigner la structure **VkSemaphoreCreateInfo**. Toutefois, dans la version actuelle de la bibliothèque, la structure ne contient que la propriété `sType` :

```
void createSemaphores() {
    VkSemaphoreCreateInfo semaphoreInfo{};
    semaphoreInfo.sType = VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO;
}
```

Les prochaines versions de Vulkan (ou des extensions) pourraient ajouter des fonctionnalités au travers des paramètres `flags` et `pNext`, comme c'est le cas avec d'autres structures. La création d'un sémaphore suit le processus habituel :

```
if (vkCreateSemaphore(device, &semaphoreInfo, nullptr, &imageAvailableSemaphore) != VK_SUCCESS ||
    vkCreateSemaphore(device, &semaphoreInfo, nullptr, &renderFinishedSemaphore) != VK_SUCCESS) {

    throw std::runtime_error("Échec de création des sémaphores !");
}
```

Les sémaphores doivent être détruits à la fin du programme, lorsque toutes les commandes sont terminées et qu'il n'y a donc plus besoin de mécanisme de synchronisation :

```
void cleanup() {
    vkDestroySemaphore(device, renderFinishedSemaphore, nullptr);
    vkDestroySemaphore(device, imageAvailableSemaphore, nullptr);
```

IV-D-3-d - Obtention d'une image provenant de la « swap chain »

Comme indiqué ci-dessus, la première chose à faire dans la fonction `drawFrame()` est d'obtenir une image provenant de la « swap chain ». Comme la « swap chain » est une fonctionnalité provenant d'une extension, nous devons utiliser une fonction nommée `vk*KHR` :

```
void drawFrame() {
    uint32_t imageIndex;
    vkAcquireNextImageKHR(device, swapChain, UINT64_MAX, imageAvailableSemaphore,
    VK_NULL_HANDLE, &imageIndex);
```

Les deux premiers paramètres de la fonction `vkAcquireNextImageKHR()` sont le périphérique logique et la « swap chain » depuis laquelle récupérer une image. Le troisième paramètre spécifie une durée maximale en nanosecondes avant d'abandonner l'attente si aucune image n'est disponible. Vous pouvez désactiver cette expiration en utilisant la plus grande valeur pour un entier non signé sur 64 bits.

Les deux paramètres suivants indiquent les objets de synchronisation qui doivent être utilisés pour signaler lorsque le moteur d'affichage a terminé d'utiliser l'image. C'est à ce moment-là que nous pouvons dessiner dedans. Il est possible de donner en paramètre un sémaphore, une barrière ou les deux. Nous allons utiliser notre sémaphore `imageAvailableSemaphore` pour cela.

Le dernier paramètre permet de récupérer une variable dans laquelle est placé l'indice de l'image de la « swap chain » qui vient d'être mise à disposition. L'indice correspond à un objet de type `VkImage` présent dans notre tableau `swapChainImages`. Nous allons aussi utiliser cet indice pour choisir le bon tampon de commandes.

IV-D-3-e - Envoi du tampon de commandes

L'envoi de la queue et la synchronisation se configurent au travers des paramètres de la structure `VkSubmitInfo`.

```
VkSubmitInfo submitInfo{};
submitInfo.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;

VkSemaphore waitSemaphores[] = {imageAvailableSemaphore};
VkPipelineStageFlags waitStages[] = {VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT};
submitInfo.waitSemaphoreCount = 1;
submitInfo.pWaitSemaphores = waitSemaphores;
submitInfo.pWaitDstStageMask = waitStages;
```

Les trois premiers paramètres indiquent le sémaphore à attendre avant de commencer l'exécution et à quelle étape du pipeline attendre. Nous souhaitons attendre que l'image soit disponible avant d'écrire les couleurs dans l'image. En théorie, cela signifie que l'implémentation peut déjà exécuter notre vertex shader et d'autres étapes même si l'image n'est pas encore disponible. Chaque entrée dans le tableau `waitStages` correspond au sémaphore dans `pWaitSemaphores` de même indice.

```
submitInfo.commandBufferCount = 1;
submitInfo.pCommandBuffers = &commandBuffers[imageIndex];
```

Les deux paramètres qui suivent indiquent les tampons de commandes à exécuter. Nous devons ici fournir le tampon de commandes qui utilise l'image de la « swap chain » que nous venons de récupérer comme attache de couleurs.

```
VkSemaphore signalSemaphores[] = {renderFinishedSemaphore};
submitInfo.signalSemaphoreCount = 1;
submitInfo.pSignalSemaphores = signalSemaphores;
```

Les paramètres `signalSemaphoreCount` et `pSignalSemaphores` indiquent quels sémaphores déclencher lorsque l'exécution du tampon de commandes est terminée. Dans notre cas, nous utilisons le sémaphore `renderFinishedSemaphore`.

```
if (vkQueueSubmit(graphicsQueue, 1, &submitInfo, VK_NULL_HANDLE) != VK_SUCCESS) {
    throw std::runtime_error("Échec d'envoi du tampon de commandes pour le rendu !");
}
```

Nous pouvons maintenant envoyer notre tampon de commandes à la queue graphique grâce à la fonction `vkQueueSubmit()`. Cette fonction prend en argument un tableau de structures de type `VkSubmitInfo` pour une question d'efficacité. Le dernier paramètre permet de fournir une barrière optionnelle déclenchée lorsque le tampon de commandes s'est exécuté. N'en ayant pas l'usage, nous passons `VK_NULL_HANDLE`.

IV-D-3-f - Dépendances des sous-passes

Les sous-passes de la passe de rendu gèrent automatiquement les transitions de l'agencement de l'image. Ces transitions sont contrôlées par les dépendances de sous-passes. Elles permettent de décrire les dépendances liées à la mémoire ou à l'exécution des sous-passes. Nous n'avons qu'une seule sous-passe pour le moment, mais les opérations juste avant et après cette sous-passe comptent comme sous-passes implicites.

Il existe deux dépendances embarquées dans Vulkan capables de gérer les transitions au début et à la fin de la passe de rendu. Toutefois, cette première dépendance ne s'exécute pas au bon moment. Elle part du principe que la transition démarre au début du pipeline, mais nous n'avons pas encore l'image à ce moment-là ! Il existe deux méthodes pour gérer ce cas. Nous pouvons modifier `waitStages` pour `imageAvailableSemaphore` à `VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT` pour être sûrs que la passe de rendu ne commence pas tant que l'image n'est pas disponible. Sinon, nous pouvons faire en sorte que la passe de rendu attende l'étape `VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT`. J'ai décidé d'implémenter cette deuxième option, car c'est une bonne occasion de détailler les dépendances de sous-passe et leur fonctionnement.

Les dépendances de sous-passe sont spécifiées à l'aide d'une structure du type `VkSubpassDependency`. Allez à la fonction `createRenderPass()` pour en ajouter une :

```
VkSubpassDependency dependency{};
dependency.srcSubpass = VK_SUBPASS_EXTERNAL;
dependency.dstSubpass = 0;
```

Les deux premiers champs permettent de fournir l'indice de la dépendance et la sous-passe de laquelle dépendre. La valeur `VK_SUBPASS_EXTERNAL` référence à la sous-passe implicite avant ou après la passe de rendu selon que vous utilisez `srcSubpass` ou `dstSubpass`. L'indice 0 correspond à notre seule et unique sous-passe. La valeur fournie à `dstSubpass` doit toujours être supérieure à `srcSubpass` pour éviter les boucles infinies (sauf si une des sous-passes est `VK_SUBPASS_EXTERNAL`).

```
dependency.srcStageMask = VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT;
dependency.srcAccessMask = 0;
```

Les deux paramètres suivants indiquent les opérations à attendre et les étapes durant lesquelles les opérations à attendre se produisent. Nous voulons attendre que la « swap chain » finisse de lire l'image avant d'y accéder. Nous pouvons attendre que l'étape d'écriture de l'attache de couleurs se termine. Cela revient au même.

```
dependency.dstStageMask = VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT;
dependency.dstAccessMask = VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT;
```

Nous indiquons ici que les opérations qui doivent attendre pendant l'étape liée à l'attache de couleurs sont celles ayant trait à l'écriture de l'attache de couleurs. Ces paramètres permettent d'effectuer la transition uniquement lorsque nécessaire (et permis) : lorsque nous souhaitons écrire les couleurs.

```
renderPassInfo.dependencyCount = 1;  
renderPassInfo.pDependencies = &dependency;
```

La structure **VkRenderPassCreateInfo** possède deux champs pour spécifier un tableau de dépendances.

IV-D-3-g - Affichage

La dernière étape consiste à envoyer le résultat à la « swap chain » afin que celle-ci l'affiche à l'écran. L'affichage se configure grâce à une structure de type **VkPresentInfoKHR**, que nous gérons à la fin de la fonction **drawFrame()**.

```
VkPresentInfoKHR presentInfo{};  
presentInfo.sType = VK_STRUCTURE_TYPE_PRESENT_INFO_KHR;  
  
presentInfo.waitSemaphoreCount = 1;  
presentInfo.pWaitSemaphores = signalSemaphores;
```

Les deux premiers paramètres permettent d'indiquer quels sémaphores attendre avant que l'affichage ne puisse être effectué.

```
VkSwapchainKHR swapChains[] = {swapChain};  
presentInfo.swapchainCount = 1;  
presentInfo.pSwapchains = swapChains;  
presentInfo.pImageIndices = &imageIndex;
```

Les deux paramètres suivants spécifient un tableau de « swap chains » auxquelles envoyer les images et l'indice de l'image dans chaque « swap chain ». Il n'y en aura presque toujours qu'une.

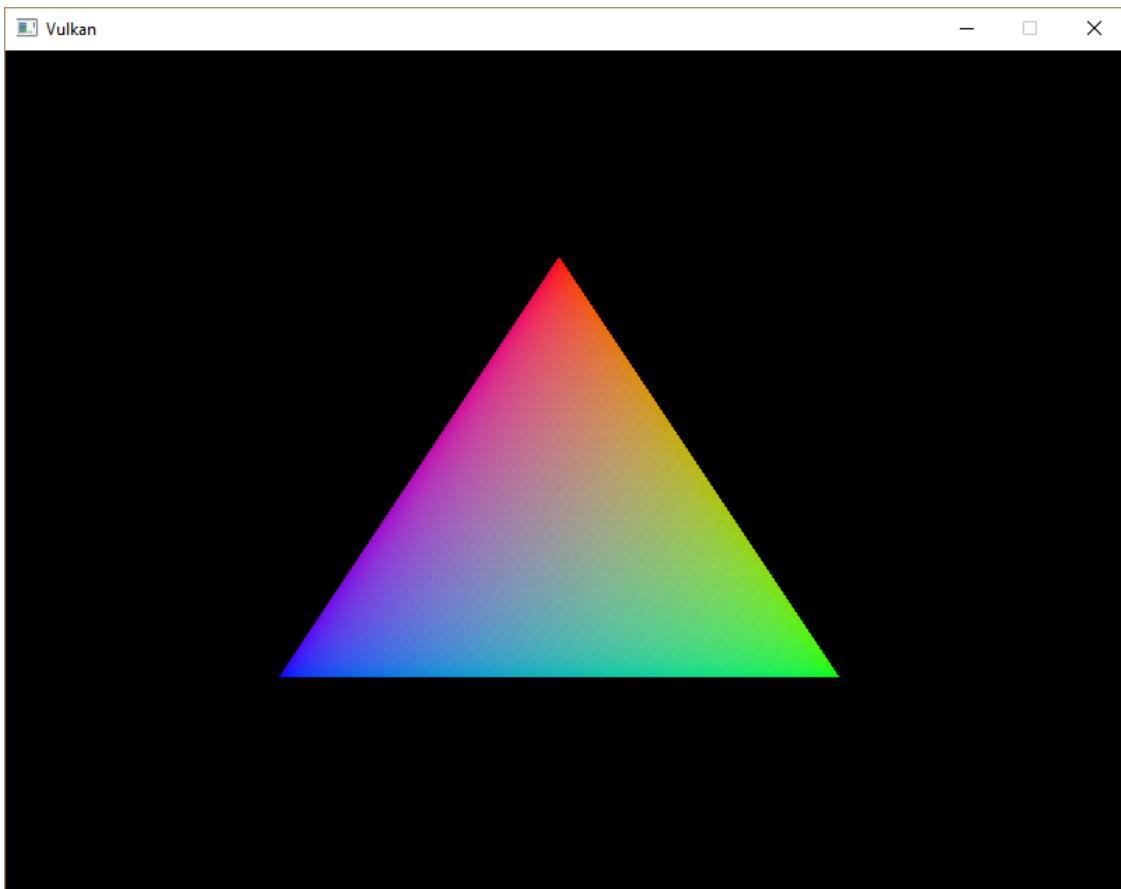
```
presentInfo.pResults = nullptr; // Optionnel
```

Ce dernier paramètre, nommé **pResults**, est optionnel. Il vous permet de fournir un tableau de **VkResult** que vous pourrez consulter pour vérifier que toutes les « swap chain » ont bien affiché leur image sans encombre. Cela n'est pas nécessaire dans le cas où vous n'utilisez qu'une « swap chain », car vous pouvez utiliser la valeur de retour de la fonction de présentation.

```
vkQueuePresentKHR(presentQueue, &presentInfo);
```

La fonction **vkQueuePresentKHR()** envoie une requête de présentation d'une image à la « swap chain ». Nous ajouterons la gestion des erreurs pour **vkAcquireNextImageKHR()** et **vkQueuePresentKHR()** dans le prochain chapitre, car une erreur à ces étapes n'implique pas forcément que le programme doit se terminer, contrairement aux fonctions vues jusqu'à présent.

Si vous avez fait tout ça correctement, vous devriez avoir quelque chose comme cela à l'écran quand vous lancez votre programme :



Enfin ! Malheureusement, si vous avez les couches de validation actives, vous verrez que le programme « crashe » dès que vous essayez de le fermer. Les messages qui s'affichent dans le terminal nous en indiquent la raison :

```
C:\WINDOWS\system32\cmd.exe
validation layer: Cannot delete semaphore 0x13 that is currently in use by a command buffer. Refer to Vulkan Spec Section '6.3. Semaphores' which states 'All submitted batches that refer to semaphores for execution' (https://www.khronos.org/registry/vulkan/specs/1.0-extensions/xhtml/vkspec.html#vkCreateSemaphore)
validation layer: Attempt to destroy command pool with command buffer (0x0000018376177390) which refers to Vulkan Spec Section '5.1. Command Pools' which states 'All VkCommandBuffer objects in a command pool must not be pending execution' (https://www.khronos.org/registry/vulkan/specs/1.0-extensions/xhtml/vkspec.html#vkDestroyCommandPool)
```

N'oubliez pas que les opérations dans la fonction `drawFrame()` sont asynchrones. Il est donc probable que, lorsque vous quittez la boucle dans `mainLoop()`, celle-ci effectue toujours des opérations de rendu ou d'affichage. Libérer des ressources dans de telles conditions n'est pas une bonne idée.

Pour régler ce problème, nous devons attendre que le périphérique logique termine ses opérations avant de quitter la fonction `mainLoop()` :

```
void mainLoop() {
    while (!glfwWindowShouldClose(window)) {
        glfwPollEvents();
        drawFrame();

        vkDeviceWaitIdle(device);
    }
}
```

Vous pouvez également attendre la fin des opérations d'une queue de commandes grâce à la fonction **`vkQueueWaitIdle()`**. Cette fonction peut aussi être utilisée pour mettre en place une synchronisation rudimentaire. Le programme devrait maintenant se terminer sans problème quand vous fermez la fenêtre.

IV-D-3-h - Rendu en cours

Si vous lancez l'application avec les couches de validations activées, vous aurez peut-être des erreurs ou une consommation mémoire qui augmente au fil du temps. La raison est que l'application soumet rapidement du travail dans la fonction `drawframe()`, mais ne vérifie pas si ces opérations se sont effectivement terminées. Si le CPU envoie plus de commandes que le GPU ne peut en exécuter, alors la queue se remplira avec du travail. Pire encore, nous réutilisons les sémaphores `imageAvailableSemaphore` et `renderFinishedSemaphore` ainsi que le tampon de commandes pour plusieurs rendus à la fois.

Pour corriger cela, le plus simple est d'attendre que le travail soit terminé avant d'en envoyer de nouveau. C'est possible avec la fonction `vkQueueWaitIdle()` :

```
void drawFrame() {
    ...

    vkQueuePresentKHR(presentQueue, &presentInfo);

    vkQueueWaitIdle(presentQueue);
}
```

Cependant, avec cette méthode, nous ne profitons pas de toute la puissance du GPU : le pipeline graphique n'est utilisé que pour un rendu à la fois. Les étapes ayant déjà été effectuées pour le rendu en cours peuvent déjà être réutilisées pour le prochain rendu. Nous allons modifier notre application pour permettre d'obtenir plusieurs rendus en parallèle tout en limitant l'effet d'entassement dans la queue.

Commencez par ajouter une constante en haut du programme qui définit le nombre de rendus pouvant être réalisés en parallèle :

```
const int MAX_FRAMES_IN_FLIGHT = 2;
```

Chaque rendu aura ses sémaphores dédiés :

```
std::vector<VkSemaphore> imageAvailableSemaphores;
std::vector<VkSemaphore> renderFinishedSemaphores;
```

La fonction `createSemaphores()` doit être améliorée pour gérer la création des nouveaux sémaphores :

```
void createSemaphores() {
    imageAvailableSemaphores.resize(MAX_FRAMES_IN_FLIGHT);
    renderFinishedSemaphores.resize(MAX_FRAMES_IN_FLIGHT);

    VkSemaphoreCreateInfo semaphoreInfo{};
    semaphoreInfo.sType = VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO;

    for (size_t i = 0; i < MAX_FRAMES_IN_FLIGHT; i++) {
        if (vkCreateSemaphore(device, &semaphoreInfo, nullptr, &imageAvailableSemaphores[i]) != VK_SUCCESS ||
            vkCreateSemaphore(device, &semaphoreInfo, nullptr, &renderFinishedSemaphores[i]) != VK_SUCCESS) {
            throw std::runtime_error("Échec de la création des sémaphores !");
        }
    }
}
```

Évidemment, nous devons les libérer dans la fonction `cleanup()` :

```
void cleanup() {
    for (size_t i = 0; i < MAX_FRAMES_IN_FLIGHT; i++) {
        vkDestroySemaphore(device, renderFinishedSemaphores[i], nullptr);
        vkDestroySemaphore(device, imageAvailableSemaphores[i], nullptr);
    }
}
```

```
    ...
}
```

Pour utiliser la bonne paire de sémaphores à chaque fois, nous devons mémoriser le rendu qui est en cours de traitement :

```
size_t currentFrame = 0;
```

La fonction `drawFrame()` peut maintenant être modifiée pour utiliser les objets adéquats :

```
void drawFrame() {
    vkAcquireNextImageKHR(device, swapChain, UINT64_MAX, imageAvailableSemaphores[currentFrame],
    VK_NULL_HANDLE, &imageIndex);

    ...

    VkSemaphore waitSemaphores[] = {imageAvailableSemaphores[currentFrame]};

    ...

    VkSemaphore signalSemaphores[] = {renderFinishedSemaphores[currentFrame]};

    ...
}
```

Nous ne devons pas oublier de faire avancer l'indice du rendu :

```
void drawFrame() {
    ...

    currentFrame = (currentFrame + 1) % MAX_FRAMES_IN_FLIGHT;
}
```

En utilisant l'opérateur de modulo %, nous pouvons nous assurer que l'indice boucle à chaque fois que nous avons traité `MAX_FRAMES_IN_FLIGHT` rendus.

Bien que nous ayons mis en place les objets facilitant le traitement de plusieurs images en parallèle, nous n'empêchons pas de réaliser plus de `MAX_FRAMES_IN_FLIGHT` rendus à la fois. Pour le moment, nous n'avons qu'un mécanisme de synchronisation GPU-GPU, mais aucun mécanisme pour synchroniser le CPU avec le GPU et ainsi connaître l'avancement du travail. Nous pouvons toujours utiliser les objets du rendu 0 alors que celui-ci est en cours de traitement.

Pour mettre en place une synchronisation CPU-GPU, Vulkan offre un autre type de primitive de synchronisation : les barrières (fences). Les barrières sont similaires aux sémaphores dans l'aspect où ceux-ci peuvent être déclenchés et attendre un tel déclenchement. Toutefois, dans ce cas, c'est notre code qui doit attendre. Nous allons d'abord créer une barrière pour chaque rendu :

```
std::vector<VkSemaphore> imageAvailableSemaphores;
std::vector<VkSemaphore> renderFinishedSemaphores;
std::vector<VkFence> inFlightFences;
size_t currentFrame = 0;
```

J'ai choisi de créer les barrières avec les sémaphores dans la fonction `createSemaphores()` et donc de la renommer en `createSyncObjects()` :

```
void createSyncObjects() {
    imageAvailableSemaphores.resize(MAX_FRAMES_IN_FLIGHT);
    renderFinishedSemaphores.resize(MAX_FRAMES_IN_FLIGHT);
    inFlightFences.resize(MAX_FRAMES_IN_FLIGHT);

    VkSemaphoreCreateInfo semaphoreInfo{};

    ...
}
```

```

semaphoreInfo.sType = VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO;

VkFenceCreateInfo fenceInfo{};
fenceInfo.sType = VK_STRUCTURE_TYPE_FENCE_CREATE_INFO;

for (size_t i = 0; i < MAX_FRAMES_IN_FLIGHT; i++) {
    if (vkCreateSemaphore(device, &semaphoreInfo, nullptr, &imageAvailableSemaphores[i]) != VK_SUCCESS ||
        vkCreateSemaphore(device, &semaphoreInfo, nullptr, &renderFinishedSemaphores[i]) != VK_SUCCESS ||
        vkCreateFence(device, &fenceInfo, nullptr, &inFlightFences[i]) != VK_SUCCESS) {
        throw std::runtime_error("Échec de création des objets de synchronisation !");
    }
}
}
}

```

La création d'un objet de type **VkFence** est très similaire à la création d'un sémaphore. N'oubliez pas de libérer les barrières :

```

void cleanup() {
    for (size_t i = 0; i < MAX_FRAMES_IN_FLIGHT; i++) {
        vkDestroySemaphore(device, renderFinishedSemaphores[i], nullptr);
        vkDestroySemaphore(device, imageAvailableSemaphores[i], nullptr);
        vkDestroyFence(device, inFlightFences[i], nullptr);
    }

    ...
}

```

Nous allons maintenant modifier la fonction `drawFrame()` pour utiliser les barrières. L'appel à la fonction `vkQueueSubmit()` inclut un paramètre optionnel qui permet de passer la barrière à déclencher lorsque le tampon de commandes a été exécuté. Nous pouvons utiliser ce signal pour déterminer qu'un rendu est terminé.

```

void drawFrame() {
    ...

    if (vkQueueSubmit(graphicsQueue, 1, &submitInfo, inFlightFences[currentFrame]) != VK_SUCCESS) {
        throw std::runtime_error("Échec d'envoi du tampon de commandes !");
    }
    ...
}

```

La dernière chose qui nous reste à modifier se trouve au début de la fonction `drawFrame()` afin d'attendre que le rendu soit terminé :

```

void drawFrame() {
    vkWaitForFences(device, 1, &inFlightFences[currentFrame], VK_TRUE, UINT64_MAX);
    vkResetFences(device, 1, &inFlightFences[currentFrame]);

    ...
}

```

La fonction **vkWaitForFences()** prend en argument un tableau de barrières et attend le déclenchement d'une ou de toutes les barrières avant de rendre la main. Le paramètre `VK_TRUE` permet d'indiquer que nous devons attendre toutes les barrières, mais dans le cas où il n'y en a qu'une, cela importe peu. Tout comme avec la fonction `vkAcquireNextImageKHR()`, cette fonction prend aussi un temps avant expiration. Contrairement aux sémaphores, nous devons manuellement remettre la barrière à un état non déclenché. Cette réinitialisation se fait avec la fonction **vkResetFences()**.

Si vous lancez le programme maintenant vous allez constater un comportement étrange. L'application ne semble plus rien afficher. En activant les couches de validation, vous aurez le message suivant :

```
C:\WINDOWS\system32\cmd.exe
validation layer: Object: 0x14 (Type = 7) | vkWaitForFences called for fence 0x14 which has not been submitted on a Queue or during acquire next image.
```

Cela signifie que nous attendons pour une barrière qui n'a pas été envoyée à Vulkan. Le problème est que, par défaut, les barrières sont créées dans un état non déclenché. Cela signifie que la fonction **vkWaitForFences()** attendra pour toujours si nous n'avons pas envoyé la barrière à Vulkan auparavant. Pour corriger cela, nous allons changer la création de la barrière afin de l'initialiser avec un état déclenché comme si nous avions effectué un premier rendu qui se serait terminé :

```
void createSyncObjects() {
    ...

    VkFenceCreateInfo fenceInfo{};
    fenceInfo.sType = VK_STRUCTURE_TYPE_FENCE_CREATE_INFO;
    fenceInfo.flags = VK_FENCE_CREATE_SIGNALED_BIT;

    ...
}
```

La fuite de mémoire n'existe plus, mais le programme ne fonctionne toujours pas correctement. Si le nombre `MAX_FRAMES_IN_FLIGHT` est plus grand que le nombre d'images de la « swap chain » ou que la fonction `vkAcquireNextImageKHR()` ne retourne pas les images dans l'ordre, il devient possible que nous affichions une image déjà en cours de traitement. Pour l'éviter, nous devons enregistrer quelles sont les images en cours de traitement. Cette correspondance permettra de lier les images en cours d'utilisation à leur barrière. Ainsi nous aurons un objet pour effectuer une attente afin d'empêcher qu'un nouveau rendu utilise une image en cours de traitement.

Tout d'abord, ajoutez une nouvelle liste nommée `imagesInFlight` pour garder trace des images en cours d'utilisation :

```
std::vector<VkFence> inFlightFences;
std::vector<VkFence> imagesInFlight;
size_t currentFrame = 0;
```

Initialisez la liste dans la fonction `createSyncObjects()` :

```
void createSyncObjects() {
    imageAvailableSemaphores.resize(MAX_FRAMES_IN_FLIGHT);
    renderFinishedSemaphores.resize(MAX_FRAMES_IN_FLIGHT);
    inFlightFences.resize(MAX_FRAMES_IN_FLIGHT);
    imagesInFlight.resize(swapChainImages.size(), VK_NULL_HANDLE);

    ...
}
```

Initialement, aucun rendu n'utilise d'image, donc on peut explicitement initialiser la liste à une valeur indiquant l'absence de barrière. Maintenant, nous allons modifier la fonction `drawFrame()` pour attendre la fin de n'importe quel rendu qui serait en train d'utiliser l'image que nous avons reçue pour le prochain rendu.

```
void drawFrame() {
    ...

    vkAcquireNextImageKHR(device, swapChain, UINT64_MAX, imageAvailableSemaphores[currentFrame],
    VK_NULL_HANDLE, &imageIndex);

    // Vérifie si le rendu précédent utilise cette image (c'est-à-dire il y a une barrière à attendre)
    if (imagesInFlight[imageIndex] != VK_NULL_HANDLE) {
        vkWaitForFences(device, 1, &imagesInFlight[imageIndex], VK_TRUE, UINT64_MAX);
    }
    // Marque l'image comme étant maintenant utilisée par ce rendu
    imagesInFlight[imageIndex] = inFlightFences[currentFrame];

    ...
}
```

{}

Parce que nous avons maintenant plus d'appels à la fonction **vkWaitForFences()**, les appels à la fonction **vkResetFences()** doivent être **déplacés**. Le mieux reste de simplement l'appeler juste avant d'utiliser la barrière :

```
void drawFrame() {
    ...
    vkResetFences(device, 1, &inFlightFences[currentFrame]);
    if (vkQueueSubmit(graphicsQueue, 1, &submitInfo, inFlightFences[currentFrame]) != VK_SUCCESS) {
        throw std::runtime_error("Échec d'envoi du tampon de commandes du rendu !");
    }
    ...
}
```

Nous avons implémenté toutes les synchronisations nécessaires pour garantir qu'il y a un maximum de deux rendus en cours de traitement et que ces rendus n'utilisent pas la même image. Notez qu'il est correct pour les autres morceaux de code, tels que le nettoyage final, de reposer sur une synchronisation plus basique comme **vkDeviceWaitIdle()**. L'approche à adopter dépend de votre besoin de performances.

Pour en apprendre plus sur la synchronisation, rendez-vous sur [cet article](#) de Khronos.

IV-D-3-i - Conclusion

Un peu plus de 900 lignes plus tard nous avons enfin atteint le moment où nous avons des résultats à l'écran ! Le démarrage avec Vulkan demande vraiment beaucoup de travail, mais ce qu'il faut retenir c'est que Vulkan vous donne énormément de contrôle à travers sa verbosité. Je vous recommande de prendre un peu de temps pour relire le code et vous construire une vue d'esprit du but de chaque objet Vulkan et comment ils interagissent. Nous allons reposer sur cette connaissance pour étendre les fonctionnalités du programme.

Dans le prochain chapitre, nous allons voir une autre petite chose nécessaire à tout bon programme Vulkan.

Code C++ / Vertex shader / Fragment shader

IV-E - Recréation de la « swap chain »

IV-E-1 - Introduction

Notre application affiche enfin un triangle, mais certains cas ne sont pas gérés correctement. Il est possible que la surface associée à la fenêtre soit modifiée d'une certaine façon, rendant la « swap chain » incompatible. Ce cas survient notamment lorsqu'on redimensionne la fenêtre. Nous allons donc mettre en place un code s'exécutant lors d'un redimensionnement.

IV-E-2 - Recréer la « swap chain »

Créez la fonction `recreateSwapChain()` qui appelle la fonction `createSwapChain()` et toutes les fonctions créant un objet dépendant de la « swap chain » ou de la taille de la fenêtre.

```
void recreateSwapChain() {
    vkDeviceWaitIdle(device);

    createSwapChain();
    createImageViews();
    createRenderPass();
```

```

        createGraphicsPipeline();
        createFramebuffers();
        createCommandBuffers();
    }
}

```

Nous appelons d'abord la fonction **vkDeviceWaitIdle()**, car, comme vu dans le chapitre précédent, nous ne devons pas modifier des ressources en cours d'utilisation. Évidemment, nous allons commencer par recréer la « swap chain ». Les vues d'image doivent être recréées, car elles reposent sur les images de la « swap chain ». La passe de rendu est aussi impactée, car elle dépend du format des images de la « swap chain ». Il est rare que le format des images de la « swap chain » soit modifié lors d'un redimensionnement de la fenêtre, mais ce cas doit quand même être géré. La taille du viewport et du rectangle de découpage est spécifiée durant la création du pipeline graphique faisant que nous devons aussi reconstruire celui-ci. Sachez qu'il est possible d'éviter ce travail en utilisant les états dynamiques pour le viewport et le rectangle de découpage. Finalement, les tampons d'images et les tampons de commandes dépendent aussi directement des images de la « swap chain ».

Pour être certains que les anciens objets sont bien détruits avant d'en recréer, nous devons déplacer certains morceaux du code de nettoyage dans une nouvelle fonction. Nous nommons celle-ci `cleanupSwapChain()` et nous l'appelons depuis la fonction `recreateSwapChain()`.

```

void cleanupSwapChain() {
}

void recreateSwapChain() {
    vkDeviceWaitIdle(device);

    cleanupSwapChain();

    createSwapChain();
    createImageViews();
    createRenderPass();
    createGraphicsPipeline();
    createFramebuffers();
    createCommandBuffers();
}
}

```

Nous allons déplacer tout le code de libération des objets qui vont être recréés avec la « swap chain », de la fonction `cleanup()` vers la fonction `cleanupSwapChain()` :

```

void cleanupSwapChain() {
    for (size_t i = 0; i < swapChainFramebuffers.size(); i++) {
        vkDestroyFramebuffer(device, swapChainFramebuffers[i], nullptr);
    }

    vkFreeCommandBuffers(device, commandPool, static_cast<uint32_t>(commandBuffers.size()), commandBuffers.data());

    vkDestroyPipeline(device, graphicsPipeline, nullptr);
    vkDestroyPipelineLayout(device, pipelineLayout, nullptr);
    vkDestroyRenderPass(device, renderPass, nullptr);

    for (size_t i = 0; i < swapChainImageViews.size(); i++) {
        vkDestroyImageView(device, swapChainImageViews[i], nullptr);
    }

    vkDestroySwapchainKHR(device, swapChain, nullptr);
}

void cleanup() {
    cleanupSwapChain();

    for (size_t i = 0; i < MAX_FRAMES_IN_FLIGHT; i++) {
        vkDestroySemaphore(device, renderFinishedSemaphores[i], nullptr);
        vkDestroySemaphore(device, imageAvailableSemaphores[i], nullptr);
        vkDestroyFence(device, inFlightFences[i], nullptr);
    }
}

```

```

vkDestroyCommandPool(device, commandPool, nullptr);

vkDestroyDevice(device, nullptr);

if (enableValidationLayers) {
    DestroyDebugUtilsMessengerEXT(instance, debugMessenger, nullptr);
}

vkDestroySurfaceKHR(instance, surface, nullptr);
vkDestroyInstance(instance, nullptr);

glfwDestroyWindow(window);

glfwTerminate();
}

```

Ce serait du gâchis de recréer le groupe de commandes à partir de zéro. J'ai préféré libérer les tampons de commandes existants à l'aide de la fonction `vkFreeCommandBuffers()`. De cette manière, nous pouvons réutiliser le groupe de commandes pour allouer de nouveaux tampons de commandes.

Pour gérer correctement le redimensionnement de la fenêtre, nous devons aussi récupérer la taille actuelle du tampon d'images afin de nous assurer que les images de la « swap chain » ont bien la nouvelle taille. Pour cela, modifiez la fonction `chooseSwapExtent()` afin que cette fonction prenne en compte la taille actuelle :

```

VkExtent2D chooseSwapExtent(const VkSurfaceCapabilitiesKHR& capabilities) {
    if (capabilities.currentExtent.width != UINT32_MAX) {
        return capabilities.currentExtent;
    } else {
        int width, height;
        glfwGetFramebufferSize(window, &width, &height);

        VkExtent2D actualExtent = {
            static_cast<uint32_t>(width),
            static_cast<uint32_t>(height)
        };

        ...
    }
}

```

C'est tout ce que nous avons à faire pour recréer la « swap chain » ! Cependant, le problème de cette approche est que nous devons arrêter le rendu avant de créer la nouvelle « swap chain ». Il est possible de créer une nouvelle « swap chain » tout en continuant l'exécution de commandes de rendu sur une image provenant de l'ancienne « swap chain ». Pour cela, vous devez passer l'ancienne « swap chain » en paramètre dans le champ `oldSwapChain` de la structure `VkSwapchainCreateInfoKHR` et détruire cette ancienne « swap chain » dès que vous ne l'utilisez plus.

IV-E-3 - Swap chain non optimale ou obsolète

Nous devons maintenant déterminer quand recréer la « swap chain » et donc quand appeler la fonction `recreateSwapChain()`. Heureusement, Vulkan nous indiquera que la « swap chain » n'est plus appropriée au moment de la présentation. Les fonctions `vkAcquireNextImageKHR()` et `vkQueuePresentKHR()` peuvent, dans ce cas, retourner les valeurs suivantes :

- `VK_ERROR_OUT_OF_DATE_KHR` : la « swap chain » n'est plus compatible avec la surface et ne peut plus être utilisée pour le rendu. Cela se produit après un redimensionnement de la fenêtre ;
- `VK_SUBOPTIMAL_KHR` : la « swap chain » peut toujours être utilisée pour présenter des images à la surface, mais les propriétés de la surface ne correspondent plus exactement.

```

VkResult result = vkAcquireNextImageKHR(device, swapChain, UINT64_MAX,
                                         imageAvailableSemaphores[currentFrame], VK_NULL_HANDLE, &imageIndex);

if (result == VK_ERROR_OUT_OF_DATE_KHR) {

```

```

        recreateSwapChain();
        return;
    } else if (result != VK_SUCCESS && result != VK_SUBOPTIMAL_KHR) {
        throw std::runtime_error("Échec lors de la récupération de l'image de la « swap chain » !");
    }
}

```

Si la « swap chain » se retrouve obsolète lorsque nous essayons d'obtenir une nouvelle image, il devient aussi impossible d'en afficher. Nous devons alors immédiatement recréer une « swap chain » et réessayer au prochain appel à la fonction drawFrame().

Vous pouvez aussi décider de recréer la « swap chain » si elle n'est plus optimale, mais j'ai décidé de continuer l'affichage, car nous avons déjà obtenu une image. Ainsi, les deux cas `VK_SUCCESS` et `VK_SUBOPTIMAL_KHR` sont considérés comme des valeurs indiquant un succès.

```

result = vkQueuePresentKHR(presentQueue, &presentInfo);

if (result == VK_ERROR_OUT_OF_DATE_KHR || result == VK_SUBOPTIMAL_KHR) {
    recreateSwapChain();
} else if (result != VK_SUCCESS) {
    throw std::runtime_error("Impossible de présenter une image de la swap chain !");
}

currentFrame = (currentFrame + 1) % MAX_FRAMES_IN_FLIGHT;

```

La fonction `vkQueuePresentKHR()` utilise les mêmes valeurs pour les mêmes raisons. Dans ce cas, nous recréons la « swap chain » aussi dans le cas où elle n'est plus optimale, car nous souhaitons les meilleurs résultats possible.

IV-E-4 - Gestion explicite des redimensionnements

Bien que la plupart des pilotes envoient automatiquement le code `VK_ERROR_OUT_OF_DATE_KHR` après qu'une fenêtre a été redimensionnée, cela n'est en aucun cas garanti. C'est pourquoi nous allons ajouter un code supplémentaire pour gérer explicitement les redimensionnements. Premièrement, ajoutez une nouvelle variable membre qui indique qu'un redimensionnement a eu lieu :

```

std::vector<VkFence> inFlightFences;
size_t currentFrame = 0;

bool framebufferResized = false;

```

Modifiez ensuite la fonction `drawFrame()` pour prendre en compte cette nouvelle variable :

```

if (result == VK_ERROR_OUT_OF_DATE_KHR || result == VK_SUBOPTIMAL_KHR || framebufferResized) {
    framebufferResized = false;
    recreateSwapChain();
} else if (result != VK_SUCCESS) {
    ...
}

```

Il est important de faire cela après la fonction `vkQueuePresentKHR()` pour s'assurer que les sémaphores sont dans un état consistant. Autrement, un sémaforo déclenché ne pourra plus jamais être utilisé pour effectuer une attente. Afin de détecter les redimensionnements, nous pouvons utiliser la fonction `glfwSetFrameBufferSizeCallback()` fournie par le framework GLFW pour mettre en place une fonction callback :

```

void initWindow() {
    glfwInit();

    glfwWindowHint(GLFW_CLIENT_API, GLFW_NO_API);

    window = glfwCreateWindow(WIDTH, HEIGHT, "Vulkan", nullptr, nullptr);
    glfwSetFrameBufferSizeCallback(window, framebufferResizeCallback);
}

```

```
static void framebufferResizeCallback(GLFWwindow* window, int width, int height) {  
}
```

Nous devons utiliser une fonction statique comme callback, car GLFW ne sait pas correctement appeler une fonction membre d'une classe avec le bon pointeur `this` pointant sur notre instance de la classe `HelloTriangleApplication`.

Toutefois, nous obtenons une référence du type `GLFWwindow` dans la fonction callback que nous fournissons. De plus, nous pouvons stocker le pointeur de notre choix dans une instance GLFW grâce à la fonction `glfwSetWindowUserPointer()` :

```
window = glfwCreateWindow(WIDTH, HEIGHT, "Vulkan", nullptr, nullptr);  
glfwSetWindowUserPointer(window, this);  
glfwSetFramebufferSizeCallback(window, framebufferResizeCallback);
```

La valeur stockée peut être récupérée dans la fonction callback grâce à la fonction `glfwGetWindowUserPointer()` afin de changer la valeur de notre variable indiquant un redimensionnement :

```
static void framebufferResizeCallback(GLFWwindow* window, int width, int height) {  
    auto app = reinterpret_cast<HelloTriangleApplication*>(glfwGetWindowUserPointer(window));  
    app->framebufferResized = true;  
}
```

Lancez maintenant le programme et redimensionnez la fenêtre pour voir si le tampon d'image est correctement redimensionné avec la fenêtre.

IV-E-5 - Gestion de la minimisation de la fenêtre

Il existe un autre cas important où la « swap chain » peut devenir obsolète : si la fenêtre est minimisée. Ce cas est particulier, car la taille du tampon d'image devient nulle. Dans ce tutoriel, nous allons gérer cela en mettant en pause la fenêtre jusqu'à ce que la fenêtre soit à nouveau remise au premier plan. Nous modifions la fonction `recreateSwapChain()` pour ce nouveau cas :

```
void recreateSwapChain() {  
    int width = 0, height = 0;  
    glfwGetFramebufferSize(window, &width, &height);  
    while (width == 0 || height == 0) {  
        glfwGetFramebufferSize(window, &width, &height);  
        glfwWaitEvents();  
    }  
  
    vkDeviceWaitIdle(device);  
  
    ...  
}
```

L'appel initial à la fonction `glfwGetFramebufferSize()` prend en charge le cas où la taille est déjà correcte et `glfwWaitEvents()` n'aurait rien à attendre.

Félicitations, vous avez créé votre premier programme fonctionnel avec Vulkan ! Dans le prochain chapitre, nous allons supprimer les sommets en dur du vertex shader et mettre en place un tampon de sommets.

Code C++ / Vertex shader / Fragment shader

V - Tampons de sommets

V-A - Description des sommets en entrée

V-A-1 - Introduction

Dans les prochains chapitres nous allons remplacer les sommets inscrits dans le vertex shader par un tampon de sommets stocké dans la mémoire de la carte graphique. Nous commencerons par la manière la plus simple pour créer un tampon qui soit accessible par le CPU. Nous utiliserons la fonction `memcpy()` pour y copier les sommets. Ensuite, nous verrons comment utiliser un tampon intermédiaire pour placer les données dans une mémoire haute performance.

V-A-2 - Vertex shader

Premièrement, supprimons les données de sommets du vertex shader. Le vertex shader recevra les données d'un tampon de sommets. Ces données entrantes sont désignées par le mot clef `in`.

```
#version 450
#extension GL_ARB_separate_shader_objects : enable

layout(location = 0) in vec2 inPosition;
layout(location = 1) in vec3 inColor;

layout(location = 0) out vec3 fragColor;

void main() {
    gl_Position = vec4(inPosition, 0.0, 1.0);
    fragColor = inColor;
}
```

Les variables `inPosition` et `inColor` sont des attributs de sommet (vertex attributes). Ce sont des propriétés spécifiques à chaque sommet et provenant du tampon de sommets. Finalement, le fonctionnement est similaire aux deux tableaux que nous venons de supprimer. N'oubliez pas de recompiler le vertex shader !

Tout comme pour la variable `fragColor`, les annotations `layout(location=x)` assignent un indice aux entrées du vertex shader que nous pourrons référencer plus tard. Il est important de savoir que certains types, tels les vecteurs 64 bits `dvec3`, utilisent plusieurs emplacements. Cela signifie que l'indice de la variable en entrée suivante ne doit pas utiliser l'emplacement qui suit :

```
layout(location = 0) in dvec3 inPosition;
layout(location = 2) in vec3 inColor;
```

Vous pouvez trouver plus d'informations sur les mots clefs liés à l'agencement des données dans [le wiki d'OpenGL](#).

V-A-3 - Sommets

Nous déplaçons les données des sommets depuis le code du shader jusqu'au code C++. Commencez par inclure la bibliothèque GLM pour pouvoir utiliser les vecteurs et les matrices. Nous allons utiliser ces types pour définir les vecteurs de position et de couleur.

```
#include <glm/glm.hpp>
```

Créez une nouvelle structure `Vertex`. Elle possède deux champs que nous utiliserons dans le vertex shader :

```
struct Vertex {
```

```
    glm::vec2 pos;
    glm::vec3 color;
};
```

La bibliothèque GLM fournit des types C++ correspondant aux types fournis par le GLSL.

```
const std::vector<Vertex> vertices = {
    {{0.0f, -0.5f}, {1.0f, 0.0f, 0.0f}},
    {{0.5f, 0.5f}, {0.0f, 1.0f, 0.0f}},
    {{-0.5f, 0.5f}, {0.0f, 0.0f, 1.0f}}
};
```

Nous utilisons la structure `Vertex` pour spécifier un tableau contenant les données des sommets. Nous reprenons exactement les mêmes positions et couleurs qu'auparavant, mais elles sont regroupées dans un unique tableau de sommets. Nous intercalons (interleaving) les couleurs parmi les positions.

V-A-4 - Descriptions des liens

La prochaine étape consiste à indiquer à Vulkan comment envoyer le format des données au vertex shader, une fois qu'elles sont stockées dans le GPU. Il existe deux types de structure pour contenir cette information.

La première est la structure `VkVertexInputBindingDescription`. Nous ajoutons une fonction membre à la structure `Vertex` pour la renseigner.

```
struct Vertex {
    glm::vec2 pos;
    glm::vec3 color;

    static VkVertexInputBindingDescription getBindingDescription() {
        VkVertexInputBindingDescription bindingDescription{};

        return bindingDescription;
    }
};
```

Une liaison pour les sommets (vertex binding) décrit comment lire les données en mémoire pour les envoyer comme sommets. Par conséquent, il faut indiquer le nombre d'octets entre les données et s'il faut passer à la donnée suivante après chaque sommet ou après chaque instance.

```
VkVertexInputBindingDescription bindingDescription{};
bindingDescription.binding = 0;
bindingDescription.stride = sizeof(Vertex);
bindingDescription.inputRate = VK_VERTEX_INPUT_RATE_VERTEX;
```

Nos données sont compactées en un seul tableau, nous n'avons donc besoin que d'un seul lien. La propriété `binding` indique l'indice du lien dans le tableau des liaisons. Le paramètre `stride` fournit le nombre d'octets séparant le début des données de chaque sommet. Finalement, la propriété `inputRate` peut prendre les valeurs suivantes :

- `VK_VERTEX_INPUT_RATE_VERTEX` : passer au jeu de données suivant après chaque sommet ;
- `VK_VERTEX_INPUT_RATE_INSTANCE` : passer au jeu de données suivant après chaque instance.

Nous ne faisons pas de rendu instancié donc nous utilisons `VK_VERTEX_INPUT_RATE_VERTEX`.

V-A-5 - Description des attributs

La seconde structure de type `VkVertexInputAttributeDescription` décrit comment gérer les sommets entrants. Nous allons ajouter une autre fonction à la structure `Vertex` pour remplir ces structures :

```
#include <array>

...

static std::array<VkVertexInputAttributeDescription, 2> getAttributeDescriptions() {
    std::array<VkVertexInputAttributeDescription, 2> attributeDescriptions{};

    return attributeDescriptions;
}
```

Comme le prototype le laisse entendre, nous allons avoir besoin de deux instances de cette structure. Une description d'attribut permet de définir comment extraire un attribut de sommets à partir des données de sommets provenant d'une description de lien. Nous avons deux attributs : la position et la couleur, donc deux descriptions.

```
attributeDescriptions[0].binding = 0;
attributeDescriptions[0].location = 0;
attributeDescriptions[0].format = VK_FORMAT_R32G32_SFLOAT;
attributeDescriptions[0].offset = offsetof(Vertex, pos);
```

Le paramètre `binding` informe Vulkan du lien de provenance des données de sommets. Le paramètre `location` correspond à la valeur donnée à la directive `location` parmi les données entrantes dans le code du vertex shader. Dans notre cas, l'entrée 0 correspond à la position du sommet stockée dans une structure de deux nombres flottants sur 32 bits.

Le paramètre `format` permet de décrire le type des données de l'attribut. Étonnamment, les formats doivent être indiqués avec la même énumération que celle pour les formats de couleurs. Voici les valeurs les plus couramment utilisées :

- float : `VK_FORMAT_R32_SFLOAT` ;
- vec2 : `VK_FORMAT_R32G32_SFLOAT` ;
- vec3 : `VK_FORMAT_R32G32B32_SFLOAT` ;
- vec4 : `VK_FORMAT_R32G32B32A32_SFLOAT`.

Comme vous pouvez le voir, vous devez utiliser le format dont le nombre de composants de couleurs correspond au nombre de composants de la donnée reçue par le shader. Il est autorisé d'utiliser plus de données que ce qui est attendu par le shader : ces données superflues seront silencieusement ignorées. Si le nombre de canaux est inférieur au nombre de composants attendu par le shader, les valeurs vert, bleu et alpha prendront les valeurs par défauts (0, 0, 1). Le type de couleurs (SFLOAT, UINT, SINT) et la taille doivent correspondre au type spécifié en entrée dans le shader. Voici quelques exemples :

- ivec2 correspond à `VK_FORMAT_R32G32_SINT` : un vecteur de deux entiers signés sur 32 bits ;
- uvec4 correspond à `VK_FORMAT_R32G32B32A32_UINT` : un vecteur de quatre entiers non signés sur 32 bits ;
- double correspond à `VK_FORMAT_R64_SFLOAT` : un nombre flottant en double précision sur 64 bits.

Le paramètre `format` détermine implicitement la taille en octets des données de l'attribut et le paramètre `offset` spécifie le décalage en octets dans les données de sommets pour lire les données de l'attribut. Le lien charge une instance `Vertex` à la fois et la position (`pos`) est à l'octet 0 (pas de décalage). C'est automatiquement calculé par la macro `offsetof`.

```
attributeDescriptions[1].binding = 0;
attributeDescriptions[1].location = 1;
attributeDescriptions[1].format = VK_FORMAT_R32G32B32_SFLOAT;
attributeDescriptions[1].offset = offsetof(Vertex, color);
```

L'attribut de couleur est décrit de la même façon.

V-A-6 - Entrée des sommets dans le pipeline

Nous devons configurer le pipeline graphique pour accepter les données de sommets dans ce format grâce aux structures dans la fonction `createGraphicsPipeline()`. Trouvez la structure `vertexInputInfo` et modifiez-la pour pointer sur les deux descriptions que nous venons de créer :

```
auto bindingDescription = Vertex::getBindingDescription();
auto attributeDescriptions = Vertex::getAttributeDescriptions();

vertexInputInfo.vertexBindingDescriptionCount = 1;
vertexInputInfo.vertexAttributeDescriptionCount = static_cast<uint32_t>(attributeDescriptions.size());
vertexInputInfo.pVertexBindingDescriptions = &bindingDescription;
vertexInputInfo.pVertexAttributeDescriptions = attributeDescriptions.data();
```

Le pipeline peut maintenant accepter les données des sommets dans le format du tableau vertices et les passer au vertex shader. Si vous lancez le programme avec les couches de validation actives, vous verrez des messages liés à l'absence de tampon de sommets associé au lien. La prochaine étape est de créer ce tampon de sommets et de copier nos données dans celui-ci afin de les rendre accessibles par le GPU.

Code C++ / Vertex shader / Fragment shader

V-B - Crédit du tampon de sommets

V-B-1 - Introduction

Dans Vulkan, les tampons sont des emplacements mémoire utilisés pour stocker des données pouvant être lues par la carte graphique. Les tampons peuvent stocker des données de sommets comme nous allons le faire dans ce chapitre, mais peuvent aussi être utilisés pour de nombreuses autres choses comme nous allons le voir dans les prochains chapitres. Contrairement aux objets Vulkan que nous avons vus jusqu'à présent, les tampons n'allouent pas automatiquement la mémoire dont ils ont besoin. Comme nous l'avons vu, la bibliothèque Vulkan donne énormément de contrôle aux développeurs et la gestion de la mémoire fait partie des choses déléguées aux développeurs.

V-B-2 - Crédit du tampon

Ajoutez une fonction nommée `createVertexBuffer()` et appelez-la depuis la fonction `initVulkan()` juste avant l'appel à la fonction `createCommandBuffers()`.

```
void initVulkan() {
    createInstance();
    setupDebugMessenger();
    createSurface();
    pickPhysicalDevice();
    createLogicalDevice();
    createSwapChain();
    createImageViews();
    createRenderPass();
    createGraphicsPipeline();
    createFramebuffers();
    createCommandPool();
    createVertexBuffer();
    createCommandBuffers();
    createSyncObjects();
}

...

void createVertexBuffer() {
```

La création d'un tampon se configure au travers d'une structure de type **VkBufferCreateInfo**.

```
VkBufferCreateInfo bufferInfo{};
bufferInfo.sType = VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO;
bufferInfo.size = sizeof(vertices[0]) * vertices.size();
```

Le premier champ de cette structure s'appelle `size`. Il spécifie la taille du tampon en octets. Nous pouvons utiliser l'opérateur `sizeof` pour déterminer la taille de notre tableau de données.

```
bufferInfo.usage = VK_BUFFER_USAGE_VERTEX_BUFFER_BIT;
```

Le deuxième champ, appelé `usage`, indique comment nous allons utiliser les données du tampon. Il est possible de définir plusieurs usages grâce à un OR binaire. Notre cas d'utilisation est celui d'un tampon de sommets. Nous verrons d'autres utilisations dans de prochains chapitres.

```
bufferInfo.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
```

De la même manière que les images de la « swap chain », les tampons peuvent appartenir à une famille de queues spécifique ou être partagés entre plusieurs familles. Le tampon sera uniquement utilisé par la queue graphique. Nous pouvons donc utiliser un accès exclusif à une queue.

Le paramètre `flags` permet de configurer la mémoire distincte (sparse) du tampon, chose ne nous intéressant pas pour le moment. Nous allons laisser la valeur par défaut de 0.

Nous pouvons maintenant créer le tampon en appelant la fonction **vkCreateBuffer()**. Définissez une variable membre pour stocker ce tampon :

```
VkBuffer vertexBuffer;

...

void createVertexBuffer() {
    VkBufferCreateInfo bufferInfo{};
    bufferInfo.sType = VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO;
    bufferInfo.size = sizeof(vertices[0]) * vertices.size();
    bufferInfo.usage = VK_BUFFER_USAGE_VERTEX_BUFFER_BIT;
    bufferInfo.sharingMode = VK_SHARING_MODE_EXCLUSIVE;

    if (vkCreateBuffer(device, &bufferInfo, nullptr, &vertexBuffer) != VK_SUCCESS) {
        throw std::runtime_error("Échec lors de la création du tampon de sommets !");
    }
}
```

Le tampon doit être utilisable dans les opérations de rendu, nous ne pouvons donc le détruire qu'à la fin du programme. Comme il ne dépend pas de la « swap chain », nous le libérons dans la fonction `cleanup()`.

```
void cleanup() {
    cleanupSwapChain();

    vkDestroyBuffer(device, vertexBuffer, nullptr);

    ...
}
```

V-B-3 - Exigences concernant la mémoire

Le tampon a été créé, mais il n'est associé à aucune région de la mémoire. Afin de pouvoir allouer de la mémoire pour le tampon, nous devons récupérer ses exigences concernant la mémoire avec la fonction **vkGetBufferMemoryRequirements()**.

```
VkMemoryRequirements memRequirements;
vkGetBufferMemoryRequirements(device, vertexBuffer, &memRequirements);
```

La structure **VkMemoryRequirements** remplie par la fonction possède trois propriétés :

- **size** : le nombre d'octets dont le tampon a besoin. Cela peut être différent de **bufferInfo.size** ;
- **alignment** : le décalage en octets où le tampon débute, par rapport au début de la zone mémoire allouée. L'alignement dépend de **bufferInfo.usage** et **bufferInfo.flags** ;
- **memoryTypeBits** : un champ de bits précisant les types de mémoire convenant au tampon.

Les cartes graphiques offrent plusieurs types de mémoire à partir desquels faire l'allocation. Chaque type de mémoire offre des caractéristiques différentes en termes de performance et peut ne permettre qu'un ensemble limité d'opérations. Nous devons trouver le bon type de mémoire à utiliser suivant les exigences de notre tampon ainsi que nos propres exigences. Créons une nouvelle fonction **findMemoryType()** pour cela.

```
uint32_t findMemoryType(uint32_t typeFilter, VkMemoryPropertyFlags properties) {
}
```

D'abord, nous récupérons les types de mémoire disponibles avec la fonction **vkGetPhysicalDeviceMemoryProperties()**.

```
VkPhysicalDeviceMemoryProperties memProperties;
vkGetPhysicalDeviceMemoryProperties(physicalDevice, &memProperties);
```

La structure **VkPhysicalDeviceMemoryProperties** contient deux tableaux appelés **memoryTypes** et **memoryHeaps**. Les zones mémoire (memory heap) indiquent des zones distinctes telles que la mémoire vidéo (VRAM) ou l'espace du fichier d'échange en RAM lorsque la mémoire vidéo est remplie. Vous pouvez ainsi choisir la mémoire à utiliser pour faire votre allocation. Les différents types de mémoire existent parmi ces zones. Actuellement, notre seule préoccupation est le type de mémoire et non de quelle zone elle provient. Mais vous pouvez deviner que cela a un impact sur les performances.

Trouvons d'abord un type de mémoire adéquat pour le tampon :

```
for (uint32_t i = 0; i < memProperties.memoryTypeCount; i++) {
    if (typeFilter & (1 << i)) {
        return i;
    }
}

throw std::runtime_error("Aucun type de mémoire adéquat !");
```

Le paramètre **typeFilter** sera utilisé comme champ de bits pour indiquer les types de mémoire adéquats. Cela signifie que nous pouvons trouver un index correspondant à un type de mémoire adéquat avec une simple boucle qui vérifie si le bit est à 1.

Cependant, nous ne sommes pas seulement intéressés par un type adéquat pour le tampon de sommets. Nous devons aussi pouvoir écrire nos données de sommets dans cette mémoire. Le tableau **memoryTypes** contient des structures de type **VkMemoryType** spécifiant la zone et les propriétés de chaque type de mémoire. Ces propriétés renseignent sur des fonctionnalités spécifiques à la mémoire, telles que la possibilité de faire correspondre la région de mémoire à une zone mémoire sur le CPU. Cette propriété est indiquée par **VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT**, mais nous devons aussi utiliser **VK_MEMORY_PROPERTY_HOST_COHERENT_BIT**. Nous verrons l'utilité de cette deuxième valeur lorsque nous accéderons à la mémoire.

Nous modifions la boucle pour vérifier le support de ces propriétés :

```
for (uint32_t i = 0; i < memProperties.memoryTypeCount; i++) {
```

```

        if ((typeFilter & (1 << i)) && (memProperties.memoryTypes[i].propertyFlags & properties) ==
properties) {
            return i;
        }
    }
}
    
```

Nous ne pouvons pas simplement vérifier si le résultat du ET bit à bit est non nul. Nous devons comparer son résultat avec les propriétés voulues afin de nous assurer que toutes les propriétés voulues sont supportées. S'il existe un type mémoire adéquat pour le tampon qui possède toutes les propriétés que nous souhaitons, nous retournons son index, sinon nous envoyons une exception.

V-B-4 - Allocation de mémoire

Maintenant que nous sommes capables de déterminer le bon type de mémoire, nous pouvons allouer de la mémoire en remplissant la structure **VkMemoryAllocateInfo**.

```

VkMemoryAllocateInfo allocInfo{};
allocInfo.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
allocInfo.allocationSize = memRequirements.size;
allocInfo.memoryTypeIndex = findMemoryType(memRequirements.memoryTypeBits,
    VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT | VK_MEMORY_PROPERTY_HOST_COHERENT_BIT);
    
```

Pour allouer de la mémoire, il suffit de spécifier la taille et le type. Ces deux données proviennent des exigences mémoire du tampon de sommets et des fonctionnalités que nous souhaitons. Créez une variable membre pour stocker la référence vers la mémoire et allouez-la avec la fonction **vkAllocateMemory()**.

```

VkBuffer vertexBuffer;
VkDeviceMemory vertexBufferMemory;

...

if (vkAllocateMemory(device, &allocInfo, nullptr, &vertexBufferMemory) != VK_SUCCESS) {
    throw std::runtime_error("Échec lors de l'allocation de la mémoire pour le tampon de
sommets !");
}
    
```

Si l'allocation a réussi, nous pouvons associer cette mémoire au tampon avec la fonction **vkBindBufferMemory()** :

```
vkBindBufferMemory(device, vertexBuffer, vertexBufferMemory, 0);
```

Les trois premiers paramètres sont évidents. Le quatrième indique le décalage entre le début de la mémoire et le début du tampon. Comme nous avons alloué la mémoire spécifiquement pour ce tampon, le décalage est 0. Si le décalage n'est pas zéro, il doit alors être divisible par `memRequirements.alignement`.

Évidemment et tout comme pour une allocation dynamique de mémoire en C++, la mémoire doit être libérée. La mémoire liée à un tampon doit être libérée une fois que le tampon n'est plus utilisé, c'est-à-dire après la destruction du tampon :

```

void cleanup() {
    cleanupSwapChain();

    vkDestroyBuffer(device, vertexBuffer, nullptr);
    vkFreeMemory(device, vertexBufferMemory, nullptr);
}
    
```

V-B-5 - Remplissage du tampon de sommets

Nous pouvons maintenant copier les données de sommets dans le tampon. Cela se fait en faisant **correspondre la mémoire du tampon** à un emplacement mémoire accessible par le CPU grâce à la fonction **vkMapMemory()**.

```
void* data;
vkMapMemory(device, vertexBufferMemory, 0, bufferInfo.size, 0, &data);
```

Cette fonction nous permet d'accéder à une région d'une ressource mémoire spécifiée par un décalage et une taille. Le décalage et la taille sont 0 et `bufferInfo.size` respectivement. Il est aussi possible de spécifier la valeur `VK_WHOLE_SIZE` pour accéder à l'intégralité de la mémoire du tampon. L'avant-dernier paramètre permet de spécifier des indicateurs, mais, dans la version actuelle de Vulkan, il n'en existe pas. Nous devons donc passer 0. Le dernier paramètre indique le pointeur à remplir pour indiquer où la zone mémoire sera accessible.

```
void* data;
vkMapMemory(device, vertexBufferMemory, 0, bufferInfo.size, 0, &data);
memcpy(data, vertices.data(), (size_t) bufferInfo.size);
vkUnmapMemory(device, vertexBufferMemory);
```

Vous pouvez maintenant utiliser la fonction `memcpy()` pour copier les sommets dans la mémoire, puis supprimer la correspondance avec la fonction **`vkUnmapMemory()`**. Malheureusement, le pilote peut ne pas copier les données dans la mémoire du tampon immédiatement, notamment à cause des mécanismes de cache. Il est aussi possible que les écritures dans le tampon ne soient pas visibles dans la mémoire correspondante. Il existe deux façons de pallier ce problème :

- utiliser une zone de mémoire cohérente avec l'hôte. Cette mémoire est spécifiée par la propriété `VK_MEMORY_PROPERTY_HOST_COHERENT_BIT` ;
- appeler la fonction **`vkFlushMappedMemoryRanges()`** après avoir écrit dans la mémoire correspondante et appeler la fonction **`vkInvalidateMappedMemory()`** avant une lecture dans la mémoire correspondante.

Nous utiliserons la première approche qui nous assure que la mémoire correspondante est toujours cohérente avec le contenu de la mémoire allouée. Gardez à l'esprit que cela peut être moins efficace que d'utiliser une opération explicite, mais nous allons voir pourquoi ce n'est pas important dans le prochain chapitre.

L'utilisation d'une mémoire cohérente ou d'une opération de mise à jour des données signifie que le pilote sera informé de nos écritures dans le tampon, mais cela ne signifie pas que les données seront visibles par le GPU. Le transfert de données vers le GPU est une opération se produisant en arrière-plan. La spécification offre la seule garantie que le déplacement est effectif au prochain appel à **`vkQueueSubmit()`**.

V-B-6 - Lier le tampon de sommets

Il ne nous reste qu'à lier le tampon de sommets lors des opérations de rendu. Nous allons pour cela compléter la fonction `createCommandBuffers()`.

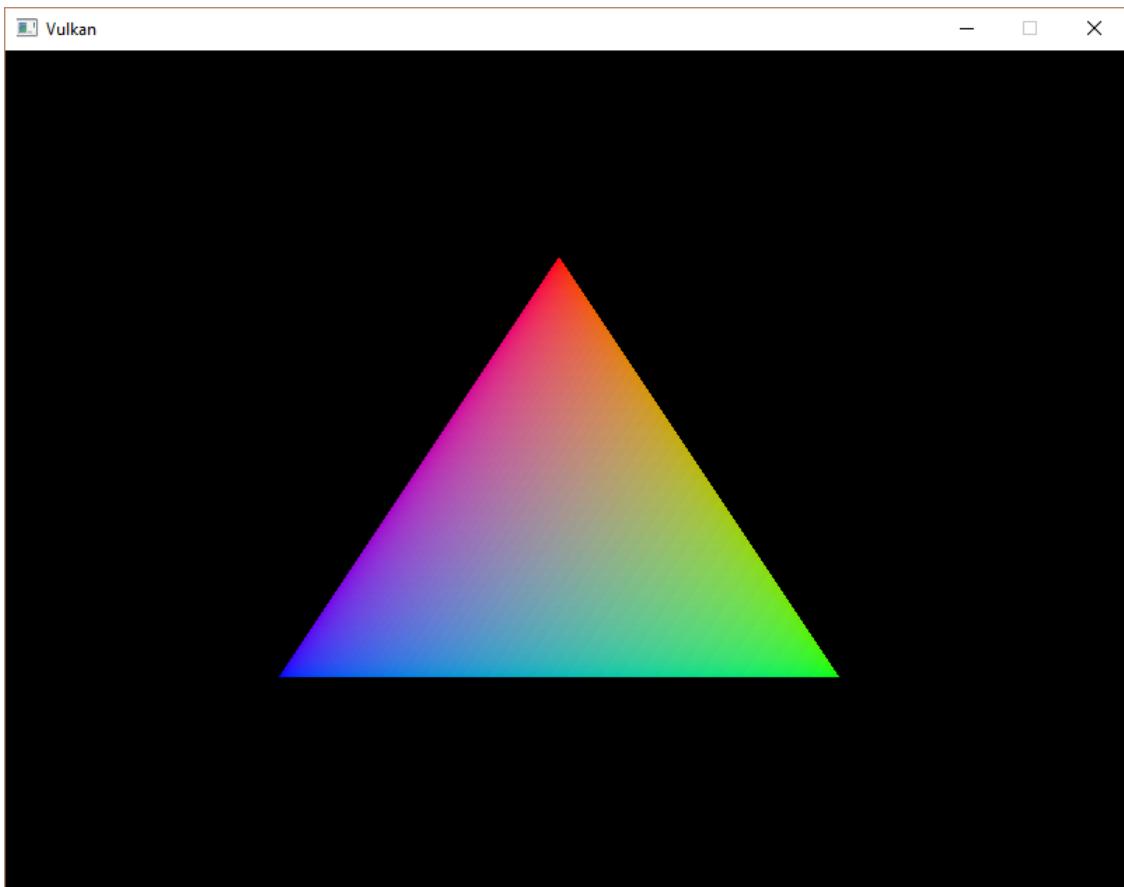
```
vkCmdBindPipeline(commandBuffers[i], VK_PIPELINE_BIND_POINT_GRAPHICS, graphicsPipeline);

VkBuffer vertexBuffers[] = {vertexBuffer};
VkDeviceSize offsets[] = {0};
vkCmdBindVertexBuffers(commandBuffers[i], 0, 1, vertexBuffers, offsets);

vkCmdDraw(commandBuffers[i], static_cast<uint32_t>(vertices.size()), 1, 0, 0);
```

La fonction **`vkCmdBindVertexBuffers()`** associe des tampons de sommets aux liens, comme celui que nous avons mis en place dans le chapitre précédent. Les deuxième et troisième paramètres indiquent le décalage et le nombre de liaisons que nous allons associer. Les deux derniers paramètres correspondent à un tableau de tampons de sommets à lier et au décalage à partir duquel démarrer la lecture des données. Vous devez modifier l'appel à la fonction **`vkCmdDraw()`** pour passer le nombre de sommets du tampon et non plus le nombre 3 en dur.

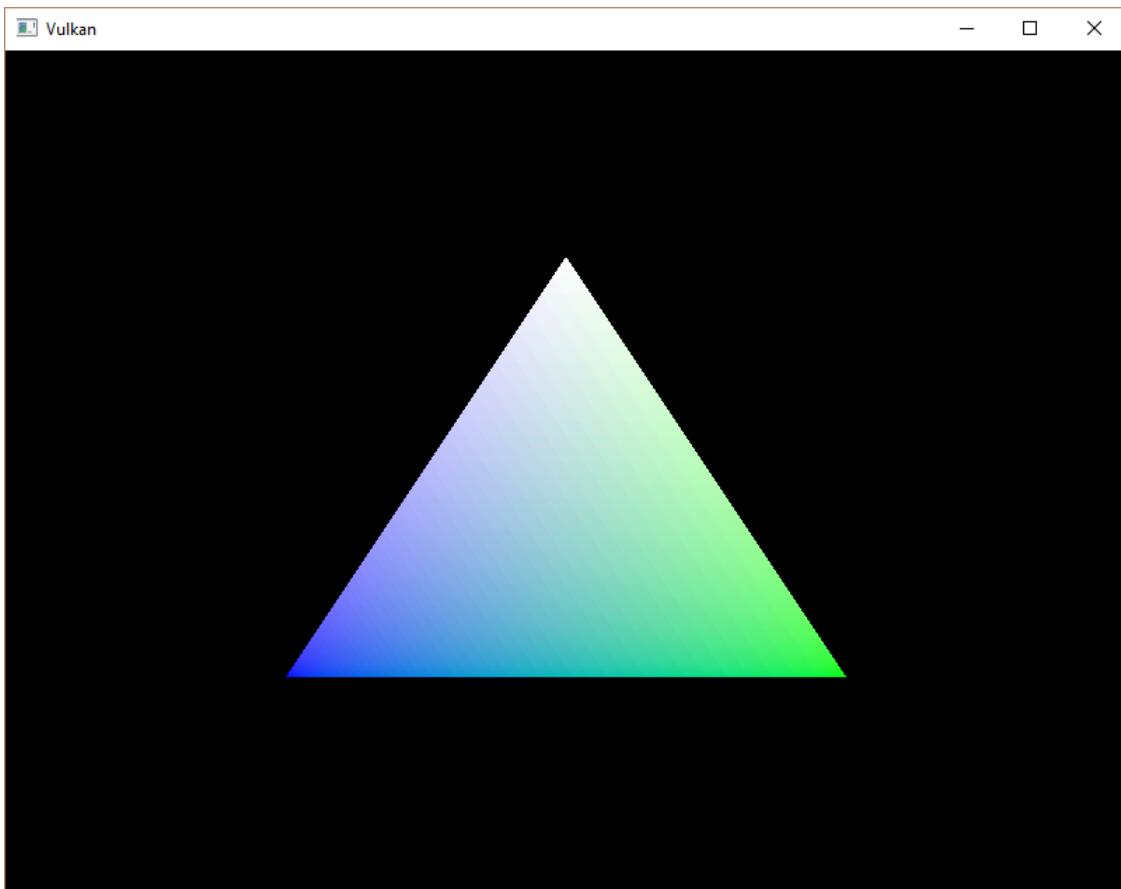
Lancez maintenant le programme. Vous devriez voir le triangle habituel apparaître à l'écran.



Essayez de modifier la couleur du sommet en haut en modifiant le tableau `vertices` :

```
const std::vector<Vertex> vertices = {
    {{0.0f, -0.5f}, {1.0f, 1.0f, 1.0f}},
    {{0.5f, 0.5f}, {0.0f, 1.0f, 0.0f}},
    {{-0.5f, 0.5f}, {0.0f, 0.0f, 1.0f}}
};
```

Relancez le programme et vous devriez obtenir ceci :



Dans le prochain chapitre, nous verrons une autre manière de copier les données de sommets vers le tampon. Elle est plus performante, mais nécessite plus de travail.

Code C++ / Vertex shader / Fragment shader

V-C - Tampon intermédiaire

V-C-1 - Introduction

Le tampon de sommets que nous avons mis en place fonctionne correctement, mais le type de mémoire qui nous permet d'avoir un accès depuis le CPU n'est pas le type le plus optimal pour une lecture à partir de la carte graphique. La mémoire la plus efficace possède la propriété `VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT`, mais n'est habituellement pas accessible par le CPU pour les cartes graphiques dédiées. Dans ce chapitre, nous allons créer deux tampons de sommets. Le premier, un tampon intermédiaire (*staging buffer*), utilisera de la mémoire accessible par le CPU pour envoyer les données du tableau de sommets vers le tampon. Le second sera dans la mémoire locale au périphérique graphique. Nous allons donc utiliser une commande de copie de tampon pour déplacer les données du tampon intermédiaire vers le vrai tampon de sommets.

V-C-2 - Queue de transfert

La commande pour copier les tampons nécessite une famille de queues supportant les opérations de transfert. Une telle queue est indiquée par le bit `VK_QUEUE_TRANSFER_BIT`. La bonne nouvelle est qu'une famille de queues estampillée `VK_QUEUE_GRAPHICS_BIT` ou `VK_QUEUE_COMPUTE_BIT` supporte implicitement les propriétés indiquées par le bit `VK_QUEUE_TRANSFER_BIT`. Par conséquent, le bit spécifique au transfert peut ne pas être présent dans la propriété `queueFlags` pour les queues graphiques ou de calcul.

Si vous aimez la difficulté, vous pouvez toujours utiliser une famille de queues dédiée pour les opérations de transfert. Vous devrez alors faire les modifications suivantes :

- modifier la structure QueueFamilyIndices et la fonction findQueueFamilies() pour trouver une famille de queues ayant le bit VK_QUEUE_TRANSFER_BIT, mais pas le bit VK_QUEUE_GRAPHICS_BIT ;
- modifier la fonction createLogicalDevice() pour récupérer une référence à une queue de transfert ;
- créer un nouveau groupe de commandes pour les tampons de commandes envoyés à la famille de queues de transfert ;
- changer la valeur de la propriété sharingMode des ressources pour être VK_SHARING_MODE_CONCURRENT et indiquer à la fois la queue des graphismes et la queue des transferts ;
- envoyer toutes les commandes de transfert telles que **vkCmdCopyBuffer()** (que nous allons utiliser dans ce chapitre) à la queue de transfert et non pas à la queue des graphismes.

Cela représente pas mal de travail, mais vous en apprendrez beaucoup sur le partage des ressources entre les familles de queues.

V-C-3 - Abstraction de la création des tampons

Comme nous allons créer plusieurs tampons, il est judicieux de placer la création des tampons dans une fonction. Appelez-la `createBuffer` et déplacez le code provenant de la fonction `createVertexBuffer()` (mis à part le code rendant le tampon visible au CPU) :

```
void createBuffer(VkDeviceSize size, VkBufferUsageFlags usage, VkMemoryPropertyFlags properties,
    VkBuffer& buffer, VkDeviceMemory& bufferMemory) {
    VkBufferCreateInfo bufferInfo{};
    bufferInfo.sType = VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO;
    bufferInfo.size = size;
    bufferInfo.usage = usage;
    bufferInfo.sharingMode = VK_SHARING_MODE_EXCLUSIVE;

    if (vkCreateBuffer(device, &bufferInfo, nullptr, &buffer) != VK_SUCCESS) {
        Échec lors de la création du tampon !"throw std::runtime_error();
    }

    VkMemoryRequirements memRequirements;
    vkGetBufferMemoryRequirements(device, buffer, &memRequirements);

    VkMemoryAllocateInfo allocInfo{};
    allocInfo.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
    allocInfo.allocationSize = memRequirements.size;
    allocInfo.memoryTypeIndex = findMemoryType(memRequirements.memoryTypeBits, properties);

    if (vkAllocateMemory(device, &allocInfo, nullptr, &bufferMemory) != VK_SUCCESS) {
        throw std::runtime_error("Échec lors de l'allocation de la mémoire pour le tampon !");
    }

    vkBindBufferMemory(device, buffer, bufferMemory, 0);
}
```

Cette fonction nécessite plusieurs paramètres : la taille du tampon, les propriétés de la mémoire et l'utilisation prévue pour ce tampon. S'ajoutent à cela deux paramètres permettant à la fonction de renvoyer les références vers les ressources créées.

Vous pouvez maintenant supprimer le code de création du tampon et d'allocation de la mémoire de la fonction `createVertexBuffer()` et y mettre à la place l'appel à la nouvelle fonction :

```
void createVertexBuffer() {
    VkDeviceSize bufferSize = sizeof(vertices[0]) * vertices.size();
    createBuffer(bufferSize, VK_BUFFER_USAGE_VERTEX_BUFFER_BIT,
    VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT | VK_MEMORY_PROPERTY_HOST_COHERENT_BIT, vertexBuffer,
    vertexBufferMemory);
```

```

void* data;
vkMapMemory(device, vertexBufferMemory, 0, bufferSize, 0, &data);
    memcpy(data, vertices.data(), (size_t) bufferSize);
vkUnmapMemory(device, vertexBufferMemory);
}
    
```

Lancez votre programme et assurez-vous que le tampon de sommets fonctionne toujours aussi bien.

V-C-4 - Utiliser un tampon intermédiaire

Nous allons maintenant faire en sorte que la fonction `createVertexBuffer()` n'utilise plus qu'un tampon visible par l'hôte comme tampon temporaire. Nous allons exploiter un tampon local au périphérique comme tampon de sommets.

```

void createVertexBuffer() {
    VkDeviceSize bufferSize = sizeof(vertices[0]) * vertices.size();

    VkBuffer stagingBuffer;
    VkDeviceMemory stagingBufferMemory;
    createBuffer(bufferSize, VK_BUFFER_USAGE_TRANSFER_SRC_BIT,
    VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT | VK_MEMORY_PROPERTY_HOST_COHERENT_BIT, stagingBuffer,
    stagingBufferMemory);

    void* data;
    vkMapMemory(device, stagingBufferMemory, 0, bufferSize, 0, &data);
        memcpy(data, vertices.data(), (size_t) bufferSize);
    vkUnmapMemory(device, stagingBufferMemory);

    createBuffer(bufferSize, VK_BUFFER_USAGE_TRANSFER_DST_BIT |
    VK_BUFFER_USAGE_VERTEX_BUFFER_BIT, VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT, vertexBuffer,
    vertexBufferMemory);
}
    
```

Nous ajoutons un nouveau `stagingBuffer` avec un `stagingBufferMemory` pour transmettre les données. Dans ce chapitre, nous allons nous servir de deux nouvelles valeurs pour indiquer notre utilisation des tampons :

- `VK_BUFFER_USAGE_TRANSFER_SRC_BIT` : indiquant que le tampon peut être employé comme source dans une opération de transfert mémoire ;
- `VK_BUFFER_USAGE_TRANSFER_DST_BIT` : indiquant que le tampon peut être employé comme destination dans une opération de transfert de mémoire.

Le `vertexBuffer` est maintenant alloué à partir d'un type de mémoire local au périphérique, ce qui, en général, signifie que nous ne pouvons pas utiliser la fonction `vkMapMemory()`. Toutefois, nous pouvons copier les données du tampon intermédiaire (`stagingBuffer`) vers le tampon de sommets (`vertexBuffer`). Nous devons spécifier notre volonté en indiquant que le tampon intermédiaire sera la source d'un transfert et le tampon de sommets une destination lorsque nous définissons l'utilisation des tampons.

Nous allons maintenant écrire une fonction nommée `copyBuffer()` pour copier le contenu d'un tampon dans un autre.

```

void copyBuffer(VkBuffer srcBuffer, VkBuffer dstBuffer, VkDeviceSize size) {
}
    
```

Les opérations de transfert mémoire sont réalisées à travers un tampon de commandes, tout comme pour les commandes de rendu. Par conséquent, nous devons allouer un tampon de commandes temporaire. Vous pouvez envisager de créer un tampon de commandes dédié pour ce genre de tampon ayant une courte durée de vie, car l'implémentation pourrait optimiser l'allocation mémoire. Vous devrez utiliser `VK_COMMAND_POOL_CREATE_TRANSIENT_BIT` pendant la création du groupe de commandes.

```

void copyBuffer(VkBuffer srcBuffer, VkBuffer dstBuffer, VkDeviceSize size) {
    VkCommandBufferAllocateInfo allocInfo{};

}
    
```

```

allocInfo.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_ALLOCATE_INFO;
allocInfo.level = VK_COMMAND_BUFFER_LEVEL_PRIMARY;
allocInfo.commandPool = commandPool;
allocInfo.commandBufferCount = 1;

VkCommandBuffer commandBuffer;
vkAllocateCommandBuffers(device, &allocInfo, &commandBuffer);
}

```

Enregistrez ensuite le tampon de commandes :

```

VkCommandBufferBeginInfo beginInfo{};
beginInfo.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO;
beginInfo.flags = VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT;

vkBeginCommandBuffer(commandBuffer, &beginInfo);

```

Nous allons utiliser le tampon de commandes une seule fois et attendre que la copie soit terminée avant de sortir de la fonction. Il est conseillé d'informer le pilote de notre intention à l'aide de `VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT`.

```

VkBufferCopy copyRegion{};
copyRegion.srcOffset = 0; // Optionnel
copyRegion.dstOffset = 0; // Optionnel
copyRegion.size = size;
vkCmdCopyBuffer(commandBuffer, srcBuffer, dstBuffer, 1, &copyRegion);

```

La copie est réalisée à l'aide de la commande `vkCmdCopyBuffer()`. Ses paramètres sont le tampon source et le tampon de destination et un tableau des zones mémoire à copier. Ces régions sont définies grâce aux structures de type `VkBufferCopy`. Elles spécifient un décalage dans le tampon source, un décalage dans le tampon de destination et une taille. Par contre, il n'est pas possible d'utiliser la valeur `VK_WHOLE_SIZE` dans ce cas.

```
vkEndCommandBuffer(commandBuffer);
```

Ce tampon de commandes ne contient que la commande de copie. Nous pouvons donc arrêter l'enregistrement juste après la copie. Exécutez le tampon de commandes pour effectuer le transfert :

```

VkSubmitInfo submitInfo{};
submitInfo.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;
submitInfo.commandBufferCount = 1;
submitInfo.pCommandBuffers = &commandBuffer;

vkQueueSubmit(graphicsQueue, 1, &submitInfo, VK_NULL_HANDLE);
vkQueueWaitIdle(graphicsQueue);

```

Contrairement aux commandes de rendu, il n'y a pas d'événement à attendre. Nous souhaitons juste exécuter le transfert des données immédiatement. Encore une fois, il y a deux façons d'attendre la fin du transfert. Nous pouvons utiliser une barrière et attendre avec la fonction `vkWaitForFences()`, ou simplement attendre que la queue de transfert devienne inactive avec la fonction `vkQueueWaitIdle()`. Une barrière permettrait d'effectuer plusieurs transferts en parallèle et d'attendre qu'ils soient tous terminés. Le pilote pourrait alors optimiser le transfert.

```
vkFreeCommandBuffers(device, commandPool, 1, &commandBuffer);
```

N'oubliez pas de libérer le tampon de commandes utilisé pour l'opération de transfert.

Nous pouvons maintenant appeler la fonction `copyBuffer()` depuis la fonction `createVertexBuffer()` pour que les sommets soient stockés dans le tampon local au périphérique.

```

createBuffer(bufferSize, VK_BUFFER_USAGE_TRANSFER_DST_BIT | VK_BUFFER_USAGE_VERTEX_BUFFER_BIT,
VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT, vertexBuffer, vertexBufferMemory);

```

```
copyBuffer(stagingBuffer, vertexBuffer, bufferSize);
```

Après avoir copié les données du tampon intermédiaire dans le tampon du périphérique, nous devons effectuer un peu de nettoyage :

```
...  
  
copyBuffer(stagingBuffer, vertexBuffer, bufferSize);  
  
vkDestroyBuffer(device, stagingBuffer, nullptr);  
vkFreeMemory(device, stagingBufferMemory, nullptr);  
}
```

Lancez votre programme pour vérifier que vous voyez toujours le même triangle. L'amélioration n'est peut-être pas flagrante, mais les données des sommets sont maintenant chargées à partir d'une mémoire haute performance. Cela aura un intérêt lorsque nous afficherons des géométries plus complexes.

V-C-5 - Conclusion

Notez que dans une application réelle, vous n'êtes pas censé appeler la fonction **vkAllocateMemory()** pour chaque tampon. Le nombre d'allocations effectuées est limité par le périphérique physique à `maxMemoryAllocationCount`. Sa valeur peut être plutôt basse, et ce, même sur un périphérique haut de gamme. Par exemple, sur une NVIDIA GTX 1080, la valeur est de 4096. La bonne façon pour allouer une zone mémoire pour pouvoir y stocker un grand nombre d'objets est d'utiliser un allocateur personnalisé qui répartira les allocations unitaires des différents objets dans la zone mémoire grâce au paramètre spécifiant un décalage (`offset`).

Vous pouvez implémenter votre propre allocateur, ou bien utiliser la bibliothèque **VulkanMemoryAllocator** créée par l'initiative GPUOpen. Toutefois, dans ce tutoriel, nous pouvons nous contenter d'une allocation mémoire pour chaque ressource, car nous n'atteindrons pas cette limite.

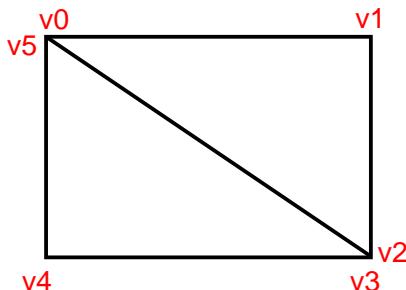
Code C++ / Vertex shader / Fragment shader

V-D - Tampon d'indices

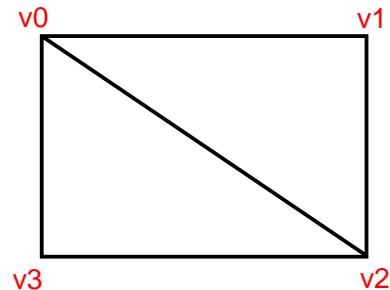
V-D-1 - Introduction

Les modèles 3D que vous allez afficher dans une application réelle vont certainement partager plusieurs sommets entre plusieurs triangles. Cela se produit même dans des cas simples, tels que l'affichage d'un rectangle :

Vertex buffer only



Vertex + index buffer



Indices
 $\{0, 1, 2, 2, 3, 0\}$

L'affichage d'un rectangle se réalise grâce à deux triangles. Nous avons donc besoin d'un tampon de sommet avec six sommets. Le problème est que les données de deux sommets sont dupliquées et nous obtenons 50 % de redondance. Cela devient pire avec des modèles plus complexes, où de nombreux sommets sont réutilisés dans trois triangles ou plus. La solution à ce problème est d'utiliser un tampon d'indices.

Un tampon d'indices est essentiellement un tableau de pointeurs vers le tampon de sommets. Il vous permet de réordonner les données de sommets et de réutiliser les données dans plusieurs triangles. Le schéma ci-dessus montre à quoi ressemble le tampon d'indices pour le rectangle si nous avions un tampon de sommets contenant quatre sommets uniques. Les trois premiers indices définissent le triangle haut droit et les trois derniers indices définissent les sommets du triangle bas gauche.

V-D-2 - Création d'un tampon d'indices

Dans ce chapitre, nous allons modifier les données pour afficher un rectangle comme celui du schéma ci-dessus. Voici les nouvelles données définissant les quatre coins :

```
const std::vector<Vertex> vertices = {
    {{-0.5f, -0.5f}, {1.0f, 0.0f, 0.0f}},
    {{0.5f, -0.5f}, {0.0f, 1.0f, 0.0f}},
    {{0.5f, 0.5f}, {0.0f, 0.0f, 1.0f}},
    {{-0.5f, 0.5f}, {1.0f, 1.0f, 1.0f}}
};
```

Le coin en haut gauche est rouge, celui en haut à droite est vert, celui en bas à droite est bleu et celui en bas à gauche est blanc. Nous allons ajouter un nouveau tableau nommé `indices` pour représenter le contenu du tampon d'indices. Il contient les indices comme spécifié dans le schéma et permet de dessiner le triangle haut droit et le triangle bas gauche

```
const std::vector<uint16_t> indices = {
    0, 1, 2, 2, 3, 0
};
```

Il est possible d'utiliser les types `uint16_t` ou `uint32_t` pour les valeurs du tampon d'indices suivant le nombre d'entrées dans le tableau `vertices`. Nous pouvons nous contenter du type `uint16_t` pour le moment, car nous allons utiliser moins de 65 535 sommets différents.

Comme pour les données des sommets, les indices doivent être envoyés au GPU à travers un objet du type `VkBuffer`. Définissez deux nouveaux membres pour stocker les ressources du tampon d'indices :

```
VkBuffer vertexBuffer;
VkDeviceMemory vertexBufferMemory;
VkBuffer indexBuffer;
VkDeviceMemory indexBufferMemory;
```

La fonction `createIndexBuffer()` que nous ajoutons est quasiment identique à la fonction `createVertexBuffer()` :

```
void initVulkan() {
    ...
    createVertexBuffer();
    createIndexBuffer();
    ...
}

void createIndexBuffer() {
    VkDeviceSize bufferSize = sizeof(indices[0]) * indices.size();

    VkBuffer stagingBuffer;
    VkDeviceMemory stagingBufferMemory;
    createBuffer(bufferSize, VK_BUFFER_USAGE_TRANSFER_SRC_BIT,
    VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT | VK_MEMORY_PROPERTY_HOST_COHERENT_BIT, stagingBuffer,
    stagingBufferMemory);

    void* data;
    vkMapMemory(device, stagingBufferMemory, 0, bufferSize, 0, &data);
    memcpy(data, indices.data(), (size_t) bufferSize);
    vkUnmapMemory(device, stagingBufferMemory);

    createBuffer(bufferSize, VK_BUFFER_USAGE_TRANSFER_DST_BIT | VK_BUFFER_USAGE_INDEX_BUFFER_BIT,
    VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT, indexBuffer, indexBufferMemory);

    copyBuffer(stagingBuffer, indexBuffer, bufferSize);

    vkDestroyBuffer(device, stagingBuffer, nullptr);
    vkFreeMemory(device, stagingBufferMemory, nullptr);
}
```

Il n'y a que deux différences notables : la valeur de `bufferSize` correspond à la taille du tableau multipliée par la taille en mémoire du type des données utilisées (`sizeof(uint16_t)`, ou `sizeof(uint32_t)`). L'usage que nous avons du tampon `indexBuffer` doit être `VK_BUFFER_USAGE_INDEX_BUFFER_BIT` au lieu de `VK_BUFFER_USAGE_VERTEX_BUFFER_BIT`. À part ça, le processus est le même. Nous créons un tampon intermédiaire pour y copier le contenu du tableau `indices` puis nous copions le contenu du tampon intermédiaire dans le tampon du périphérique.

Le tampon d'indices doit être libéré à la fin du programme, tout comme pour le tampon de sommets.

```
void cleanup() {
    cleanupSwapChain();

    vkDestroyBuffer(device, indexBuffer, nullptr);
    vkFreeMemory(device, indexBufferMemory, nullptr);

    vkDestroyBuffer(device, vertexBuffer, nullptr);
    vkFreeMemory(device, vertexBufferMemory, nullptr);

    ...
}
```

V-D-3 - Utilisation d'un tampon d'indices

Pour utiliser le tampon d'indices lors des opérations de rendu nous devons effectuer deux modifications à la fonction `createCommandBuffers()`. Nous devons d'abord lier le tampon d'indices, tout comme nous l'avons fait pour le tampon de sommets. La différence est que nous pouvons seulement n'avoir qu'un tampon d'indices. Malheureusement, il n'est pas possible d'utiliser un tampon d'indices différent pour chaque attribut de sommet : nous devons donc avoir de la duplication de sommets si un attribut varie.

```
vkCmdBindVertexBuffers(commandBuffers[i], 0, 1, vertexBuffers, offsets);  
vkCmdBindIndexBuffer(commandBuffers[i], indexBuffer, 0, VK_INDEX_TYPE_UINT16);
```

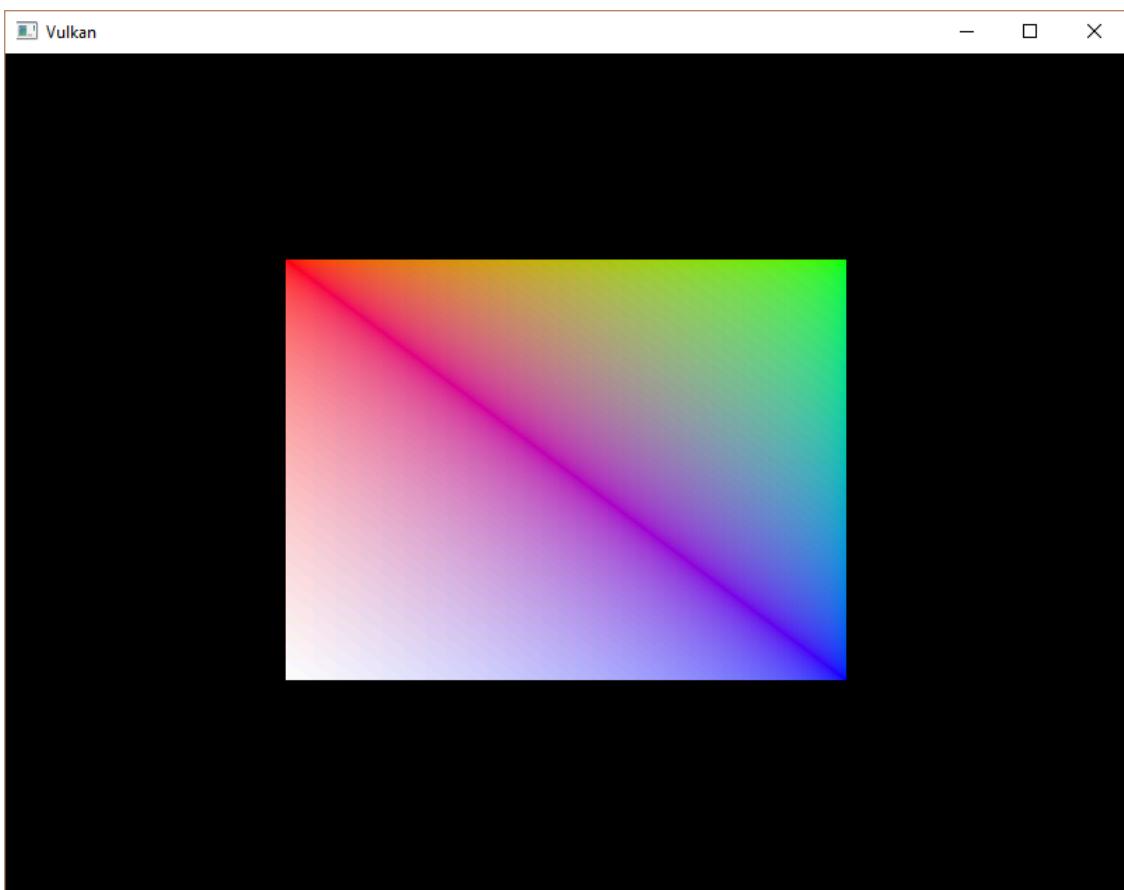
Un tampon d'indices se lie grâce à la fonction **vkCmdBindIndexBuffer()**. La fonction prend en paramètres le tampon d'indices, un décalage et le type des données des indices. Comme indiqué précédemment, les types possibles sont `VK_INDEX_TYPE_UINT16` et `VK_INDEX_TYPE_UINT32`.

La seule liaison du tampon d'indices ne change rien. Nous devons aussi modifier la commande de rendu pour dire à Vulkan d'utiliser le tampon d'indices. Enlevez la fonction **vkCmdDraw()** et remplacez-la par la fonction **vkCmdDrawIndexed()** :

```
vkCmdDrawIndexed(commandBuffers[i], static_cast<uint32_t>(indices.size()), 1, 0, 0, 0);
```

Cette fonction est similaire à la fonction **vkCmdDraw()**. Les deux premiers paramètres indiquent le nombre d'indices et le nombre d'instances. Nous n'utilisons pas l'instanciation, donc nous spécifions une unique instance. Le nombre d'indices représente le nombre de sommets qui seront passés au tampon de sommets. Le paramètre suivant indique un décalage dans le tampon d'indices. L'avant-dernier paramètre indique un nombre à ajouter aux indices du tampon d'indices lors du rendu. Le dernier paramètre indique un décalage pour l'instanciation, que nous n'utilisons pas.

Lancez le programme et vous devriez obtenir ceci :



Vous savez maintenant économiser la mémoire en réutilisant les sommets à l'aide d'un tampon d'indices. Cela deviendra crucial dans les chapitres suivants dans lesquels vous allez apprendre à charger des modèles 3D complexes.

Nous avons déjà évoqué le fait que vous devriez allouer plusieurs ressources, telles que les tampons, avec une seule opération d'allocation mémoire. En réalité, vous devez aller encore plus loin. Les **développeurs de pilotes**

recommandent que vous stockiez plusieurs tampons, tels que le tampon de sommets et le tampon d'indices dans un unique **VkBuffer** et d'utiliser des décalages dans les commandes telles que **vkCmdBindVertexBuffers()**. L'avantage est que vos données regroupées permettent une meilleure utilisation des caches. Il est même possible de réutiliser le même morceau de mémoire pour plusieurs ressources si elles ne sont pas utilisées dans les mêmes opérations de rendu et que les données sont mises à jour. Cela s'appelle de l'aliasing et certaines fonctions Vulkan possèdent des options spécifiques pour implémenter ce mécanisme.

Code C++ / Vertex shader / Fragment shader

VI - Tampons de variables uniformes

VI-A - Descripteur d'agencement et de tampon

VI-A-1 - Introduction

Nous pouvons maintenant passer des attributs différents pour chaque sommet au vertex shader, mais qu'en est-il des variables globales ? À partir de ce chapitre, nous allons effectuer un rendu 3D : cela nécessite une matrice de modèle-vue-projection. Nous pouvons l'inclure comme données de sommet, mais c'est un énorme gâchis de mémoire. De plus, il faudrait mettre à jour le tampon de sommets à chaque fois que nous voulons modifier la transformation : c'est-à-dire à chaque image !

La solution fournie par Vulkan consiste à utiliser des *descripteurs de ressource* (resource descriptors). Un descripteur est un moyen pour les shaders d'obtenir un accès libre aux ressources telles que des tampons ou des images. Nous allons configurer un tampon qui contiendra les matrices de transformation. Le vertex shader pourra y accéder grâce à un descripteur. Leur mise en place se fait en trois parties :

- spécifier l'agencement du descripteur (descriptor layout) durant la création du pipeline ;
- allouer un ensemble de descripteurs (descriptor set) depuis un groupe de descripteurs (descriptor pool) ;
- lier l'ensemble de descripteurs durant les opérations de rendu.

L'agencement du descripteur indique le type de ressources auquel le pipeline pourra accéder. Cela fonctionne de manière similaire à ce que nous avons fait pour les attaches auxquelles la passe de rendu doit accéder. L'ensemble de descripteurs indique le tampon ou les images qui seront liées aux descripteurs. Cela fonctionne comme le tampon d'images qui spécifie les vues d'images à lier aux attaches de la passe de rendu. L'ensemble de descripteurs est ensuite lié aux commandes de rendu, tout comme les tampons de sommets et le tampon d'images.

Il existe plusieurs types de descripteurs, mais dans ce chapitre, nous travaillerons avec les tampons de variables uniformes (*uniform buffer objects* (UBO)). Nous verrons les autres types de descripteurs plus tard, mais le processus de mise en place est le même. Partons du principe que les données que nous voulons envoyer au vertex shader sont stockées dans une structure C comme suit :

```
struct UniformBufferObject {
    glm::mat4 model;
    glm::mat4 view;
    glm::mat4 proj;
};
```

Nous devons copier les données dans un objet de type **VkBuffer** et y accéder par le biais d'un descripteur d'objet de tampon de variables uniformes dans le vertex shader :

```
layout(binding = 0) uniform UniformBufferObject {
    mat4 model;
    mat4 view;
    mat4 proj;
} ubo;
```

```
void main() {
    gl_Position = ubo.proj * ubo.view * ubo.model * vec4(inPosition, 0.0, 1.0);
    fragColor = inColor;
}
```

Nous allons mettre à jour les matrices de modèle, vue et projection à chaque image afin de faire tourner le rectangle dans une scène 3D.

VI-A-2 - Vertex shader

Modifiez le vertex shader pour inclure les variables uniformes comme décrit plus haut. Je pars du principe que vous connaissez les transformations de modèle, vue et projection. Si ce n'est pas le cas, vous pouvez lire [ce tutoriel](#).

```
#version 450
#extension GL_ARB_separate_shader_objects : enable

layout(binding = 0) uniform UniformBufferObject {
    mat4 model;
    mat4 view;
    mat4 proj;
} ubo;

layout(location = 0) in vec2 inPosition;
layout(location = 1) in vec3 inColor;

layout(location = 0) out vec3 fragColor;

void main() {
    gl_Position = ubo.proj * ubo.view * ubo.model * vec4(inPosition, 0.0, 1.0);
    fragColor = inColor;
}
```

Notez que l'ordre des variables `uniform`, `in` et `out` n'a aucune importance. La directive `binding` est semblable à la directive `location` pour les attributs. Nous référençons ce binding dans l'agencement du descripteur. La ligne concernant la variable `gl_Position` a été modifiée pour utiliser les transformations dans le calcul permettant d'obtenir la position finale dans l'espace de coordonnées de découpage. Contrairement aux triangles 2D, le dernier composant de la coordonnée peut ne pas être 1. Par conséquent, une division aura bien lieu lors du passage aux coordonnées normalisées pour l'écran. Cette division de perspective permet de faire que les objets les plus proches sont plus gros que les objets au loin.

VI-A-3 - Agencement de l'ensemble de descripteurs

La prochaine étape consiste à définir l'UBO côté C++. Nous devons aussi informer Vulkan que nous voulons l'utiliser dans le vertex shader.

```
struct UniformBufferObject {
    glm::mat4 model;
    glm::mat4 view;
    glm::mat4 proj;
};
```

Nous pouvons faire correspondre parfaitement la déclaration C++ avec celle du shader grâce aux types fournis par GLM. Les données dans les matrices sont binairement compatibles avec ce qui est attendu par les shaders. Ainsi, nous pouvons utiliser la fonction `memcpy()` pour copier l'objet `UniformBufferObject` dans un `VkBuffer`.

Nous devons fournir des informations sur chacun des descripteurs utilisés par les shaders lors de la création du pipeline, tout comme nous le faisons pour chaque attribut de sommet. Évidemment, nous devons aussi spécifier leur indice pour correspondre à la directive `binding`. Nous allons mettre en place une fonction nommée `createDescriptorSetLayout()` ayant ce rôle. La fonction doit être appelée avant la création du pipeline.

```

void initVulkan() {
    ...
    createDescriptorSetLayout();
    createGraphicsPipeline();
    ...
}

...
void createDescriptorSetLayout() {
}

```

Chaque lien (binding) doit être décrit grâce à la structure de type `VkDescriptorSetLayoutBinding`.

```

void createDescriptorSetLayout() {
    VkDescriptorSetLayoutBinding uboLayoutBinding{};
    uboLayoutBinding.binding = 0;
    uboLayoutBinding.descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
    uboLayoutBinding.descriptorCount = 1;
}

```

Les deux premières propriétés indiquent le binding spécifié dans le shader et le type du descripteur, c'est-à-dire un tampon de variables uniformes. Il est possible que la variable du shader soit un tableau d'UBO. Pour ce cas, la propriété `descriptorCount` indique le nombre d'éléments dans le tableau. Cette possibilité pourrait être utilisée pour transmettre la transformation à appliquer à chaque os d'un squelette pour effectuer une animation. Notre transformation modèle, vue, projection ne consiste qu'en un objet UBO. Par conséquent, la valeur de la propriété `descriptorCount` est 1.

```
uboLayoutBinding.stageFlags = VK_SHADER_STAGE_VERTEX_BIT;
```

Nous devons aussi indiquer à Vulkan dans quelles étapes programmables les descripteurs seront référencés. Le champ de bits `stageFlags` peut être une combinaison des valeurs `VkShaderStageFlagBits` ou la valeur `VK_SHADER_STAGE_ALL_GRAPHICS`. Nous utilisons ce descripteur uniquement dans le vertex shader.

```
uboLayoutBinding.pImmutableSamplers = nullptr; // Optionnel
```

La propriété `pImmutableSamplers` n'est utile que pour les descripteurs en rapport avec l'échantillonnage des images que nous verrons plus tard. Nous laissons donc la valeur par défaut.

Tous les liens de descripteurs sont ensuite combinés en un seul objet de type `VkDescriptorSetLayout` correspondant à l'agencement des descripteurs. Créez pour cela une nouvelle variable membre nommée `pipelineLayout` :

```
VkDescriptorSetLayout descriptorsetLayout;
VkPipelineLayout pipelineLayout;
```

Nous pouvons créer cet objet grâce à la fonction `vkCreateDescriptorSetLayout()`. Cette fonction prend en argument une structure de type `VkDescriptorSetLayoutCreateInfo` contenant un tableau avec les liens :

```

VkDescriptorSetLayoutCreateInfo layoutInfo{};
layoutInfo.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO;
layoutInfo.bindingCount = 1;
layoutInfo.pBindings = &uboLayoutBinding;

if (vkCreateDescriptorSetLayout(device, &layoutInfo, nullptr, &descriptorsetLayout) != VK_SUCCESS) {
    throw std::runtime_error("Échec lors de la création de l'agencement des descripteurs !");
}

```

Nous devons fournir cette structure à Vulkan durant la création du pipeline graphique afin que les shaders puissent utiliser les descripteurs. Les agencements des descripteurs sont spécifiés dans l'agencement du pipeline graphique. Modifiez la structure **VkPipelineLayoutCreateInfo** pour référencer le nouvel objet :

```
VkPipelineLayoutCreateInfo pipelineLayoutInfo = {};
pipelineLayoutInfo.sType = VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;
pipelineLayoutInfo.setLayoutCount = 1;
pipelineLayoutInfo.pSetLayouts = &descriptorsetLayout;
```

Vous pouvez vous demander pourquoi il est possible de spécifier plusieurs agencements de descripteurs alors qu'un unique objet inclut toutes les liaisons. Nous allons revenir sur ce point plus tard, quand nous allons détailler les groupes de descripteurs et les ensembles de descripteurs.

L'objet doit persister tant que nous créons des pipelines graphiques, autrement dit, jusqu'à la fin du programme :

```
void cleanup() {
    cleanupSwapChain();

    vkDestroyDescriptorSetLayout(device, descriptorsetLayout, nullptr);

    ...
}
```

VI-A-4 - Tampon de variables uniformes

Dans le prochain chapitre, nous allons spécifier le tampon contenant les données UBO pour le shader. Toutefois, nous devons d'abord créer un tampon. À chaque image, nous allons copier des données différentes dans le tampon, il est donc contre-productif d'utiliser un tampon intermédiaire. Cela ajouterait de la complexité et dégraderait les performances.

Nous avons besoin de plusieurs tampons, car nous pouvons traiter plusieurs rendus en parallèle et nous ne souhaitons pas mettre à jour un tampon qui est toujours en cours de lecture pour le rendu précédent. Nous pouvons soit en avoir un par rendu, soit un par image de la « swap chain ». Comme nous devons référencer un tampon de variables uniformes à partir du tampon de commandes qui lui-même est distinct pour chaque image de la « swap chain », il est donc logique d'avoir un tampon de variables uniformes pour chaque image.

Pour cela, créez les variables membres `uniformBuffers` et `uniformBuffersMemory` :

```
VkBuffer indexBuffer;
VkDeviceMemory indexBufferMemory;

std::vector<VkBuffer> uniformBuffers;
std::vector<VkDeviceMemory> uniformBuffersMemory;
```

Par ailleurs, créez une nouvelle fonction nommée `createUniformBuffers()` et appelez-la après la fonction `createIndexBuffers()`. Son but est d'allouer les tampons :

```
void initVulkan() {
    ...
    createVertexBuffer();
    createIndexBuffer();
    createUniformBuffers();
    ...

    void createUniformBuffers() {
        VkDeviceSize bufferSize = sizeof(UniformBufferObject);

        uniformBuffers.resize(swapChainImages.size());
```

```

uniformBuffersMemory.resize(swapChainImages.size());

for (size_t i = 0; i < swapChainImages.size(); i++) {
    createBuffer(bufferSize, VK_BUFFER_USAGE_UNIFORM_BUFFER_BIT,
    VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT | VK_MEMORY_PROPERTY_HOST_COHERENT_BIT, uniformBuffers[i],
    uniformBuffersMemory[i]);
}
}

```

Nous allons créer une autre fonction qui mettra à jour le tampon avec la nouvelle transformation, et ce, à chaque rendu. C'est pourquoi il n'y a pas d'appel à la fonction **vkMapMemory()** ici. Les données uniformes seront utilisées dans tous les rendus. Par conséquent, le tampon contenant les variables uniformes doit être détruit après l'arrêt du rendu. Comme le tampon dépend du nombre d'images fournies par la « swap chain », nous devons le modifier après la reconstruction de la « swap chain ». Nous allons donc faire le nettoyage dans la fonction `cleanupSwapChain()` :

```

void cleanupSwapChain() {
    ...

    for (size_t i = 0; i < swapChainImages.size(); i++) {
        vkDestroyBuffer(device, uniformBuffers[i], nullptr);
        vkFreeMemory(device, uniformBuffersMemory[i], nullptr);
    }
}

```

Le tampon doit donc être recréé dans la fonction `recreateSwapChain()` :

```

void recreateSwapChain() {
    ...

    createFramebuffers();
    createUniformBuffers();
    createCommandBuffers();
}

```

VI-A-5 - Mise à jour des variables uniformes

Créez une nouvelle fonction nommée `updateUniformBuffer()` et appelez-la dans la fonction `drawFrame()`, juste après avoir obtenu une image de la « swap chain » :

```

void drawFrame() {
    ...

    uint32_t imageIndex;
    VkResult result = vkAcquireNextImageKHR(device, swapChain, UINT64_MAX,
    imageAvailableSemaphores[currentFrame], VK_NULL_HANDLE, &imageIndex);

    ...

    updateUniformBuffer(imageIndex);

    VkSubmitInfo submitInfo{};
    submitInfo.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;

    ...
}

...

void updateUniformBuffer(uint32_t currentImage) {
}

```

Cette fonction génère une transformation à chaque rendu permettant de tourner le modèle 3D. Nous devons inclure deux nouveaux fichiers d'en-têtes pour implémenter cette fonctionnalité :

```
#define GLM_FORCE_RADIANS
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>

#include <chrono>
```

Le fichier d'en-tête `<glm/gtc/matrix_transform.hpp>` fournit des fonctions permettant de générer des matrices de transformations. Nous avons besoin des fonctions `glm::rotate()` pour la matrice du modèle, `glm::lookAt()` pour la matrice de vue et `glm::perspective()` pour la matrice de projection. La macro `GLM_FORCE_RADIANS` assure l'utilisation des radians pour les angles, notamment ceux passés en paramètre à la fonction `glm::rotate()`.

La bibliothèque standard chrono fournit des fonctions liées à la mesure du temps. Nous allons l'utiliser pour implémenter une rotation fluide de 90 degrés par seconde, et ce, quel que soit le nombre d'images par seconde :

```
void updateUniformBuffer(uint32_t currentImage) {
    static auto startTime = std::chrono::high_resolution_clock::now();

    auto currentTime = std::chrono::high_resolution_clock::now();
    float time = std::chrono::duration<float>(std::chrono::seconds::period)(currentTime - startTime).count();
}
```

La fonction `updateUniformBuffer()` commence par la logique pour calculer le temps en secondes depuis le début du rendu.

Ensuite, nous définissons les matrices de modèle, vue et projection que nous stockons dans le tampon de variables uniformes. La matrice du modèle représente une simple rotation sur l'axe Z suivant la variable `time` :

```
UniformBufferObject ubo{};
ubo.model = glm::rotate(glm::mat4(1.0f),
    time * glm::radians(90.0f), glm::vec3(0.0f, 0.0f, 1.0f));
```

La fonction `glm::rotate()` accepte en argument une matrice déjà existante, un angle de rotation et un axe de rotation. Le constructeur `glm::mat4(1.0)` crée une matrice identité. Avec la multiplication `time * glm::radians(90.0f)` l'objet tournera avec une vitesse de 90 degrés par seconde.

```
ubo.view = glm::lookAt(glm::vec3(2.0f, 2.0f, 2.0f), glm::vec3(0.0f, 0.0f, 0.0f), glm::vec3(0.0f, 0.0f, 1.0f));
```

J'ai décidé de placer la vue 45° au-dessus de l'objet. La fonction `glm::lookAt` prend en arguments la position de l'oeil, la direction du regard et l'axe servant de référence pour le haut.

```
1. ubo.proj = glm::perspective(glm::radians(45.0f), swapChainExtent.width / (float)
    swapChainExtent.height, 0.1f, 10.0f);
```

J'ai opté pour un champ de vision vertical de 45 degrés. Les autres paramètres de la fonction `glm::perspective()` sont le ratio et les plans proche et lointain. Il est important d'utiliser la zone d'échange de la « swap chain » en cours pour calculer le ratio, afin d'utiliser des valeurs à jour suite aux redimensionnements de la fenêtre.

```
ubo.proj[1][1] *= -1;
```

La bibliothèque GLM a été initialement conçue pour OpenGL, qui utilise des coordonnées de découpage inversé pour l'axe Y. La manière la plus simple de compenser cela consiste à changer le signe de l'axe Y dans la matrice de projection. Si vous ne le faites pas, l'image sera retournée.

Maintenant que nous avons toutes les transformations, nous pouvons copier les données dans le tampon de variables uniformes. Pour ce faire, nous faisons comme pour les tampons de sommets, mais sans tampon intermédiaire :

```
void* data;
vkMapMemory(device, uniformBuffersMemory[currentImage], 0, sizeof(ubo), 0, &data);
```

```
    memcpy(data, &ubo, sizeof(ubo));
vkUnmapMemory(device, uniformBuffersMemory[currentImage]);
```

Utiliser un UBO de cette manière n'est pas ce qu'il y a de plus efficace pour transmettre au shader des données fréquemment mises à jour. Le mieux est d'utiliser des constantes poussées (push constant) que nous aborderons dans un futur chapitre.

Dans le chapitre suivant, nous allons mettre en place les ensembles de descripteurs qui vont lier les objets de type **VkBuffer** aux descripteurs de tampon de variables uniformes afin que le shader puisse accéder aux transformations.

Code C++ / Vertex shader / Fragment shader

VI-B - Groupe de descripteurs et ensembles

VI-B-1 - Introduction

L'agencement de descripteurs du chapitre précédent décrit le type des descripteurs que nous pouvons lier. Dans ce chapitre, nous allons créer un ensemble de descripteurs pour chaque ressource de type **VkBuffer** à lier au descripteur de tampon de variables uniformes.

VI-B-2 - Groupe de descripteurs

Les ensembles de descripteurs ne peuvent pas être créés directement. Il faut les allouer depuis un groupe (pool), comme pour les tampons de commandes. Nous allons mettre en place une fonction nommée `createDescriptorPool()` pour configurer un groupe de descripteurs.

```
void initVulkan() {
    ...
    createUniformBuffers();
    createDescriptorPool();
    ...
}

void createDescriptorPool() {
}
```

Nous devons d'abord indiquer les types des descripteurs contenus dans le groupe ainsi que leur nombre. Pour cela, nous utilisons une structure du type **VkDescriptorPoolSize** :

```
VkDescriptorPoolSize poolSize{};
poolSize.type = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
poolSize.descriptorCount = static_cast<uint32_t>(swapChainImages.size());
```

Nous allons allouer un descripteur pour chaque image. Cette structure est référencée dans la structure principale **VkDescriptorPoolCreateInfo**.

```
VkDescriptorPoolCreateInfo poolInfo{};
poolInfo.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_POOL_CREATE_INFO;
poolInfo.poolSizeCount = 1;
poolInfo.pPoolSizes = &poolSize;
```

En plus du nombre maximal de descripteurs, nous devons aussi spécifier le nombre maximum d'ensembles de descripteurs pouvant être alloués :

```
poolInfo.maxSets = static_cast<uint32_t>(swapChainImages.size());
```

La structure possède un indicateur optionnel, similaire à celui des groupes de commandes, pour indiquer si des ensembles individuels de descripteurs peuvent être libérés ou non : `VK_DESCRIPTOR_POOL_CREATE_FREE_DESCRIPTOR_SET_BIT`. Nous n'allons pas toucher l'ensemble après sa création. Cet indicateur est donc inutile et nous laissons la valeur par défaut.

```
VkDescriptorPool descriptorPool;
...
if (vkCreateDescriptorPool(device, &poolInfo, nullptr, &descriptorPool) != VK_SUCCESS) {
    throw std::runtime_error("Échec de création du groupe de descripteurs !");
}
```

Ajoutez une nouvelle variable membre pour stocker la référence au groupe de descripteurs et appelez la fonction `vkCreateDescriptorPool()` pour créer le groupe. Le groupe doit être détruit lorsque la « swap chain » est reconstruite, car le groupe dépend du nombre d'images :

```
void cleanupSwapChain() {
    ...
    for (size_t i = 0; i < swapChainImages.size(); i++) {
        vkDestroyBuffer(device, uniformBuffers[i], nullptr);
        vkFreeMemory(device, uniformBuffersMemory[i], nullptr);
    }
    vkDestroyDescriptorPool(device, descriptorPool, nullptr);
}
```

Le groupe doit être recréé dans la fonction `recreateSwapChain()` :

```
void recreateSwapChain() {
    ...
    createUniformBuffers();
    createDescriptorPool();
    createCommandBuffers();
}
```

VI-B-3 - Ensemble de descripteurs

Nous pouvons maintenant allouer les ensembles de descripteurs. Ajoutez une fonction nommée `createDescriptorSets()` :

```
void initVulkan() {
    ...
    createDescriptorPool();
    createDescriptorSets();
    ...
}

void recreateSwapChain() {
    ...
    createDescriptorPool();
    createDescriptorSets();
    ...
}

void createDescriptorSets() {
```

L'allocation de cette ressource est définie par la structure de type **VkDescriptorSetAllocateInfo**. Vous devez indiquer le groupe de descripteurs à partir duquel faire l'allocation, le nombre d'ensembles à créer et l'agencement de ceux-ci :

```
std::vector<VkDescriptorSetLayout> layouts(swapChainImages.size(), descriptorsetLayout);
VkDescriptorSetAllocateInfo allocInfo{};
allocInfo.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_ALLOCATE_INFO;
allocInfo.descriptorPool = descriptorPool;
allocInfo.descriptorCount = static_cast<uint32_t>(swapChainImages.size());
allocInfo.pSetLayouts = layouts.data();
```

Dans notre cas, nous créons autant d'ensembles de descripteurs qu'il y a d'images dans la « swap chain ». Ils auront tous le même agencement. Malheureusement, nous devons copier l'agencement plusieurs fois, car la fonction suivante prend un tableau ayant la même taille que le nombre d'ensembles à créer.

Ajoutez une variable membre pour stocker les références des ensembles et allouez-les avec la fonction **vkAllocateDescriptorSets()** :

```
VkDescriptorPool descriptorPool;
std::vector<VkDescriptorSet> descriptorSets;
...

descriptorSets.resize(swapChainImages.size());
if (vkAllocateDescriptorSets(device, &allocInfo, descriptorSets.data()) != VK_SUCCESS) {
    throw std::runtime_error("Échec lors de l'allocation des ensembles de descripteurs !");
}
```

Vous n'avez pas besoin de libérer les ensembles de descripteurs manuellement. Ils seront libérés lors de la destruction du groupe de descripteurs. L'appel à la fonction **vkAllocateDescriptorSets()** alloue les ensembles de descripteurs, chacun possédant un descripteur de tampon de variable uniforme.

Les ensembles de descripteurs sont alloués, mais les descripteurs à l'intérieur doivent être configurés. Nous allons ajouter une boucle pour définir la configuration de chaque descripteur :

```
for (size_t i = 0; i < swapChainImages.size(); i++) {
```

Les descripteurs référant à un tampon, comme c'est le cas pour notre descripteur de tampon de variables uniformes, sont configurés avec une structure de type **VkDescriptorBufferInfo**. La structure indique le tampon et la région dans ce tampon contenant les données pour le descripteur.

```
for (size_t i = 0; i < swapChainImages.size(); i++) {
    VkDescriptorBufferInfo bufferInfo{};
    bufferInfo.buffer = uniformBuffers[i];
    bufferInfo.offset = 0;
    bufferInfo.range = sizeof(UniformBufferObject);
}
```

Si vous écrasez l'intégralité du tampon, comme nous le faisons ici, il est aussi possible d'utiliser la valeur **VK_WHOLE_SIZE** pour la propriété **range**. La configuration des descripteurs se met à jour avec la fonction **vkUpdateDescriptorSets()**. Elle prend, en paramètre, un tableau contenant des instances de la structure **VkWriteDescriptorSet**.

```
VkWriteDescriptorSet descriptorWrite{};
descriptorWrite.sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
descriptorWrite.dstSet = descriptorSets[i];
descriptorWrite.dstBinding = 0;
descriptorWrite.dstArrayElement = 0;
```

Les deux premières propriétés spécifient l'ensemble de descripteurs à mettre à jour et l'indice du lien auquel il correspond. Nous spécifions 0, car nous n'avons qu'une unique liaison de tampon de variables uniformes. Souvenez-vous que les descripteurs peuvent être des tableaux ; nous devons donc aussi indiquer l'indice du tableau à partir duquel nous voulons effectuer des modifications. Nous n'utilisons pas de tableau, donc nous spécifions 0.

```
descriptorWrite.descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
descriptorWrite.descriptorCount = 1;
```

Encore une fois, nous devons indiquer le type du descripteur. Il est possible de mettre à jour plusieurs descripteurs à la fois grâce à un tableau. La mise à jour commence à partir de l'index `dstArrayElement`. La propriété `descriptCount` indique le nombre d'éléments dans le tableau que nous voulons mettre à jour.

```
descriptorWrite.pBufferInfo = &bufferInfo;
descriptorWrite.pImageInfo = nullptr; // Optionnel
descriptorWrite.pTexelBufferView = nullptr; // Optionnel
```

Le dernier champ référence un tableau de `descriptorCount` éléments représentant les configurations des descripteurs. La propriété à utiliser parmi les trois dépend du type du descripteur. Le champ `pBufferInfo` s'utilise avec les descripteurs référençant un tampon de données, `pImageInfo` s'utilise avec des descripteurs référençant des données images et `pTexelBufferView` s'utilise avec des descripteurs référençant des vues de tampons. Notre descripteur repose sur les tampons, donc nous utilisons le champ `pBufferInfo`.

```
vkUpdateDescriptorSets(device, 1, &descriptorWrite, 0, nullptr);
```

Les mises à jour sont appliquées avec la fonction `vkUpdateDescriptorSets()`. La fonction accepte deux tableaux, un tableau contenant des instances du type `VkWriteDescriptorSets` et un second contenant des instances de type `VkCopyDescriptorSet`. Comme son nom l'indique, le deuxième permet de copier des descripteurs.

VI-B-4 - Utiliser des ensembles de descripteurs

Nous devons maintenant modifier la fonction `createCommandBuffers()` pour lier le bon ensemble de descripteurs suivant l'image de la « swap chain » aux descripteurs présents dans le shader grâce à la fonction `vkCmdBindDescriptorSets()`. Cela doit être fait avant l'appel à la fonction `vkCmdDrawIndexed()`.

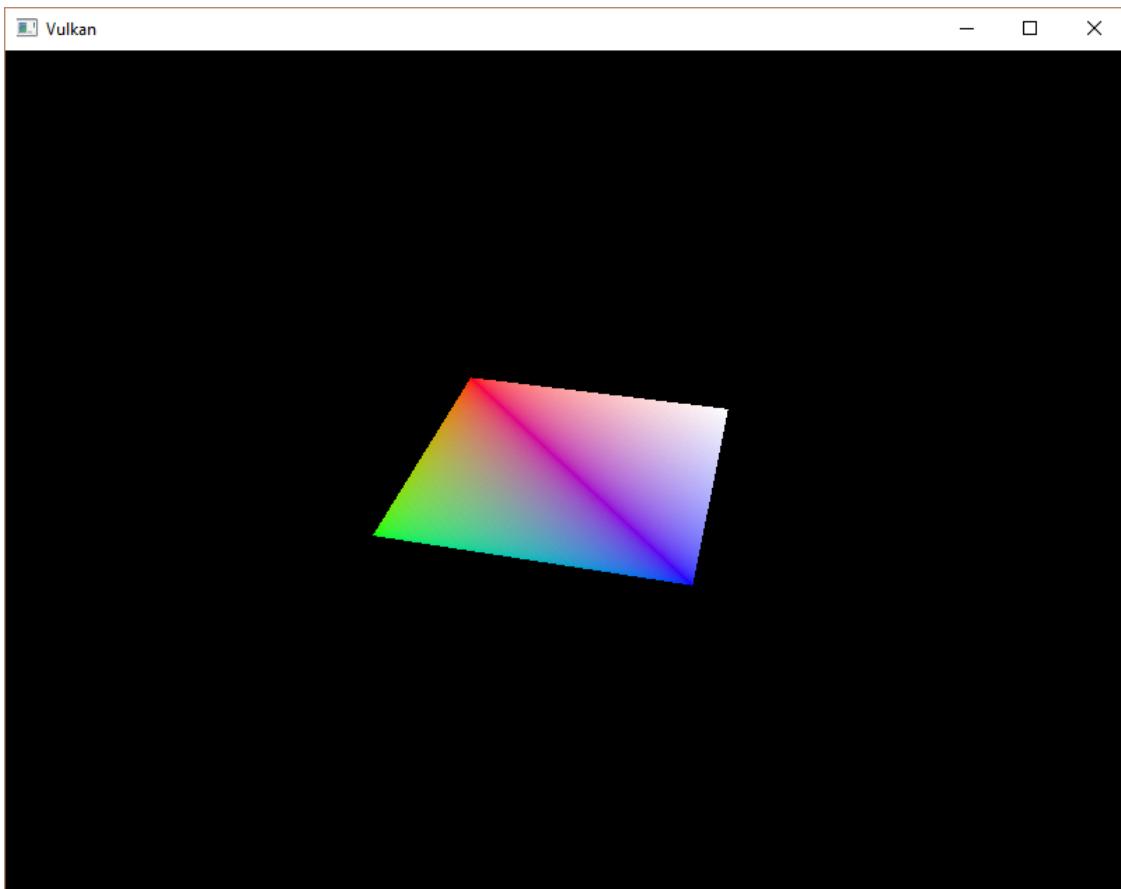
```
vkCmdBindDescriptorSets(commandBuffers[i], VK_PIPELINE_BIND_POINT_GRAPHICS,
    pipelineLayout, 0, 1, &descriptorSets[i], 0, nullptr);
vkCmdDrawIndexed(commandBuffers[i], static_cast<uint32_t>(indices.size()), 1, 0, 0, 0);
```

Contrairement aux tampons de sommets ou d'indices, les ensembles de descripteurs ne sont pas spécifiques aux pipelines graphiques. Par conséquent, nous devons indiquer à quel pipeline lier nos ensembles de descripteurs. Le paramètre suivant correspond à l'agencement sur lequel les descripteurs reposent. Les trois paramètres suivants sont l'index du premier ensemble de descripteurs, le nombre d'ensembles à lier et le tableau des ensembles à lier. Nous allons revenir sur ce point dans un instant. Les deux derniers paramètres permettent de spécifier un tableau de décalage à utiliser pour les descripteurs dynamiques. Nous y reviendrons aussi dans un futur chapitre.

Si vous lancez le programme, vous verrez que rien ne s'affiche. Le problème est que l'inversion de la coordonnée Y dans la matrice de projection fait que les sommets sont dessinés dans le sens inverse des aiguilles d'une montre. Par conséquent, l'étape de suppression des faces, configurée pour supprimer les faces arrière (backface culling), supprime nos triangles. Allez dans la fonction `createGraphicsPipeline()` et modifiez le paramètre `frontFace` de la structure `VkPipelineRasterizationStateCreateInfo` pour corriger cela :

```
rasterizer.cullMode = VK_CULL_MODE_BACK_BIT;
rasterizer.frontFace = VK_FRONT_FACE_COUNTER_CLOCKWISE;
```

Maintenant, vous devriez voir ceci en lançant votre programme :



Le rectangle est maintenant un carré, car la matrice de projection corrige son aspect. La fonction `updateUniformBuffer()` gère le redimensionnement de la fenêtre, il n'est donc pas nécessaire de recréer les descripteurs dans la fonction `recreateSwapChain()`.

VI-B-5 - Alignement

Jusqu'à présent, nous avons ignoré si les données des structures C++ correspondent avec la définition des variables uniformes du shader. Cela semble évident d'utiliser les mêmes types dans les deux déclarations :

```
struct UniformBufferObject {
    glm::mat4 model;
    glm::mat4 view;
    glm::mat4 proj;
};

layout(binding = 0) uniform UniformBufferObject {
    mat4 model;
    mat4 view;
    mat4 proj;
} ubo;
```

Pourtant, ce n'est pas aussi simple. Essayez par exemple de modifier la structure et le shader comme suit :

```
struct UniformBufferObject {
    glm::vec2 foo;
    glm::mat4 model;
    glm::mat4 view;
    glm::mat4 proj;
};

layout(binding = 0) uniform UniformBufferObject {
    vec2 foo;
```

```
mat4 model;
mat4 view;
mat4 proj;
} ubo;
```

Recompilez les shaders et relancez le programme. Le carré coloré a disparu ! Cela se produit, car nous avons ignoré les alignements mémoire des structures.

Vulkan attend que vos structures soient alignées d'une certaine façon :

- les nombres scalaires doivent être alignés sur N (soit quatre octets pour les nombres flottant sur 32 bits) ;
- un `vec2` doit être aligné sur $2N$ (huit octets) ;
- un `vec3` ou `vec4` doit être aligné sur $4N$ (16 octets) ;
- une structure imbriquée doit être alignée suivant la somme des alignements de ses membres arrondie sur le multiple de 16 supérieur ;
- une matrice `mat4` doit avoir le même alignement qu'un `vec4`.

Vous pouvez trouver la liste complète des alignements dans [la spécification](#).

Notre premier shader spécifiait trois champs de type `mat4` et correspondait donc aux règles d'alignements sus-citées. Chaque variable de type `mat4` s'aligne sur 4 octets, soit $4 * 4 * 4 = 64$ octets. La matrice modèle se situe à l'octet 0, la matrice de vue à l'octet 64 et la matrice de projection à l'octet 128. Toutes les matrices ont un décalage multiple de 16 faisant que tout va bien.

La nouvelle structure débute avec une variable de type `vec2` stockée sur 8 octets et provoquant donc un décalage de toutes les autres variables. La matrice modèle se situe à l'octet 8, la matrice vue à l'octet 72 et la matrice de projection à l'octet 136. Aucun des décalages n'est multiple de 16 dans ce cas. Pour corriger ce problème, vous pouvez utiliser le mot clef **alignas** introduit avec le C++11 :

```
struct UniformBufferObject {
    glm::vec2 foo;
    alignas(16) glm::mat4 model;
    glm::mat4 view;
    glm::mat4 proj;
};
```

Si vous recompilez et relancez le programme vous devriez constater que le shader reçoit correctement les matrices.

Heureusement, il existe une méthode pour ne pas avoir à penser à ces problématiques d'alignement la plupart du temps. Nous pouvons définir la macro `GLM_FORCE_DEFAULT_ALIGNED_GENTYPES` avant d'inclure le fichier d'en-tête de GLM :

```
#define GLM_FORCE_RADIANS
#define GLM_FORCE_DEFAULT_ALIGNED_GENTYPES
#include <glm/glm.hpp>
```

Ainsi, nous forçons la bibliothèque GLM à utiliser une version des types `vec2` et `mat4` ayant le même alignement mémoire que Vulkan. Si vous ajoutez cette macro, vous pouvez supprimer le mot clef `alignas` de la structure et votre programme fonctionnera toujours.

Malheureusement, cette méthode n'est pas suffisante si vous utilisez des structures imbriquées. Prenons l'exemple suivant :

```
struct Foo {
    glm::vec2 v;
};

struct UniformBufferObject {
    Foo f1;
```

```
    Foo f2;
};
```

Et pour le shader, utilisons :

```
struct Foo {
    vec2 v;
};

layout(binding = 0) uniform UniformBufferObject {
    Foo f1;
    Foo f2;
} ubo;
```

Dans ce cas, la variable f2 se situe à l'octet 8 alors qu'elle devrait être à l'octet 16, car c'est une structure imbriquée. Dans ce cas, vous devez indiquer vous-même l'alignement :

```
struct UniformBufferObject {
    Foo f1;
    alignas(16) Foo f2;
};
```

À cause de ce genre de problème, il est préférable de toujours expliciter l'alignement. De cette manière, vous ne serez pas pris au dépourvu par des comportements étranges liés à des erreurs d'alignement.

```
struct UniformBufferObject {
    alignas(16) glm::mat4 model;
    alignas(16) glm::mat4 view;
    alignas(16) glm::mat4 proj;
};
```

N'oubliez pas de recompiler le shader avec avoir supprimé le champ `foo`.

VI-B-6 - Plusieurs ensembles de descripteurs

Comme nous avons pu le voir, certaines fonctions permettent de lier plusieurs ensembles de descripteurs à la fois. Vous devez spécifier un agencement de descripteur pour chaque ensemble lors de la création de l'agencement du pipeline. Les shaders peuvent référencez un ensemble spécifique de cette façon :

```
layout(set = 0, binding = 0) uniform UniformBufferObject { ... }
```

Vous pouvez utiliser cette fonctionnalité pour utiliser des descripteurs variant par objet et avoir des descripteurs référencés par plusieurs ensembles. Dans ce cas, vous évitez de relier la plupart de vos descripteurs lors des appels de rendu, ce qui peut être plus performant.

Code C++ / Vertex shader / Fragment shader

VII - Application des textures

VII-A - Images

VII-A-1 - Introduction

Jusqu'à présent, les couleurs de la géométrie sont déterminées grâce aux données envoyées pour chaque sommet. Le résultat est plutôt limité. Dans ce chapitre, nous allons appliquer une texture afin de rendre la géométrie plus intéressante. Nous aurons alors le nécessaire pour charger et dessiner des modèles 3D.

L'ajout d'une texture comprend les étapes suivantes :

- créer un objet image/texture stocké sur la mémoire de la carte graphique ;
- remplir l'objet avec les pixels provenant d'un fichier image ;
- créer un échantillonneur d'image (sampler) ;
- ajouter un descripteur associé à l'échantillonneur afin d'accéder aux pixels de la texture depuis le shader.

Nous avons déjà travaillé avec des images, mais ces dernières étaient créées par l'extension de la « swap chain ». La création d'une image et l'envoi des données dans cette image ressemblent à ce que nous avons déjà fait pour le tampon de sommets. Nous allons donc commencer par créer une ressource intermédiaire à laquelle nous allons envoyer les données des pixels, puis copier cette ressource dans l'objet image final utilisé pour le rendu. Bien qu'il soit possible de créer une image intermédiaire pour cela, Vulkan permet de copier les pixels depuis un objet **VkBuffer** vers l'image. De plus, cette méthode est **plus rapide sur certaines plateformes**. Nous allons donc d'abord créer un tampon et le remplir des valeurs des pixels, puis nous allons créer une image pour y copier les pixels. La création d'une image n'est pas très différente de la création d'un tampon. Cela nécessite de récupérer les exigences de la mémoire, d'allouer la mémoire du périphérique et de lier la ressource. Nous avons déjà vu tout cela.

Toutefois, le fonctionnement d'une image induit une différence. Les images peuvent avoir des agencements différents impactant l'organisation des pixels en mémoire. Le fonctionnement des cartes graphiques fait que le simple stockage des pixels ligne par ligne ne permet pas toujours d'obtenir les meilleures performances. Nous devons nous assurer que l'agencement est optimal pour les opérations que nous souhaitons effectuer. Nous avons déjà rencontré certains de ces agencements lorsque nous avons configuré la passe de rendu :

- `VK_IMAGE_LAYOUT_PRESENT_SCR_KHR` : optimal pour la présentation ;
- `VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL` : optimal pour une attache pour l'écriture des couleurs par le fragment shader ;
- `VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL` : optimal pour être la source d'un transfert comme avec la fonction `vkCmdCopyImageToBuffer()` ;
- `VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL` : optimal pour être la cible d'un transfert comme avec la fonction `vkCmdCopyBufferToImage()` ;
- `VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL` : optimal pour être échantillonné depuis un shader.

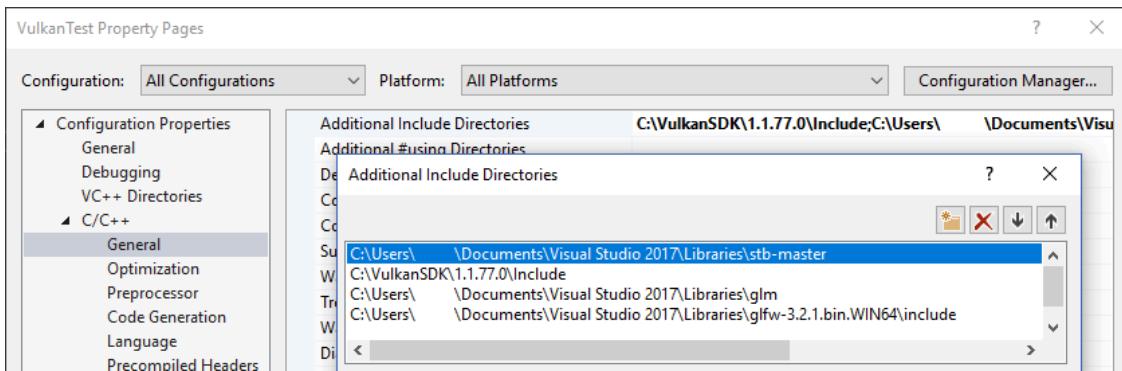
La méthode la plus utilisée pour réaliser un changement d'agencement d'une image est d'utiliser une barrière de pipeline (pipeline barrier). Les barrières de pipeline sont principalement utilisées pour synchroniser l'accès à une ressource : par exemple, pour s'assurer qu'une image a été écrite avant de la lire. Les barrières peuvent aussi être utilisées pour effectuer une transition d'agencement. Dans ce chapitre, nous verrons comment utiliser une barrière pour cela. Les barrières peuvent également être employées pour changer le propriétaire d'une famille de queues lorsque vous avez utilisé l'option `VK_SHARING_MODE_EXCLUSIVE`.

VII-A-2 - Bibliothèque de chargement d'image

De nombreuses bibliothèques permettent le chargement d'une image. Vous pouvez même écrire le code pour lire des formats simples comme le BMP ou PPM. Dans ce tutoriel, nous utilisons la bibliothèque `stb_image` provenant de la **collection stb**. Elle possède l'avantage que tout son code est contenu dans un seul fichier. Téléchargez le fichier `stb_image.h` et placez-le dans un emplacement adéquat, par exemple dans le dossier où sont stockées les bibliothèques GLFW et GLM.

VII-A-2-a - Visual Studio

Ajoutez le dossier comprenant `stb_image.h` dans « Autres répertoires Include » (Additional Include Directories).



VII-A-2-b - Makefile

Ajoutez le dossier comprenant `stb_image.h` aux chemins d'inclusion de GCC :

```
VULKAN_SDK_PATH = /home/user/VulkanSDK/x.x.x.x/x86_64
STB_INCLUDE_PATH = /home/user/libraries/stb

...
CFLAGS = -std=c++17 -I$(VULKAN_SDK_PATH)/include -I$(STB_INCLUDE_PATH)
```

VII-A-3 - Chargement d'une image

Incluez la bibliothèque de cette manière :

```
#define STB_IMAGE_IMPLEMENTATION
#include <stb_image.h>
```

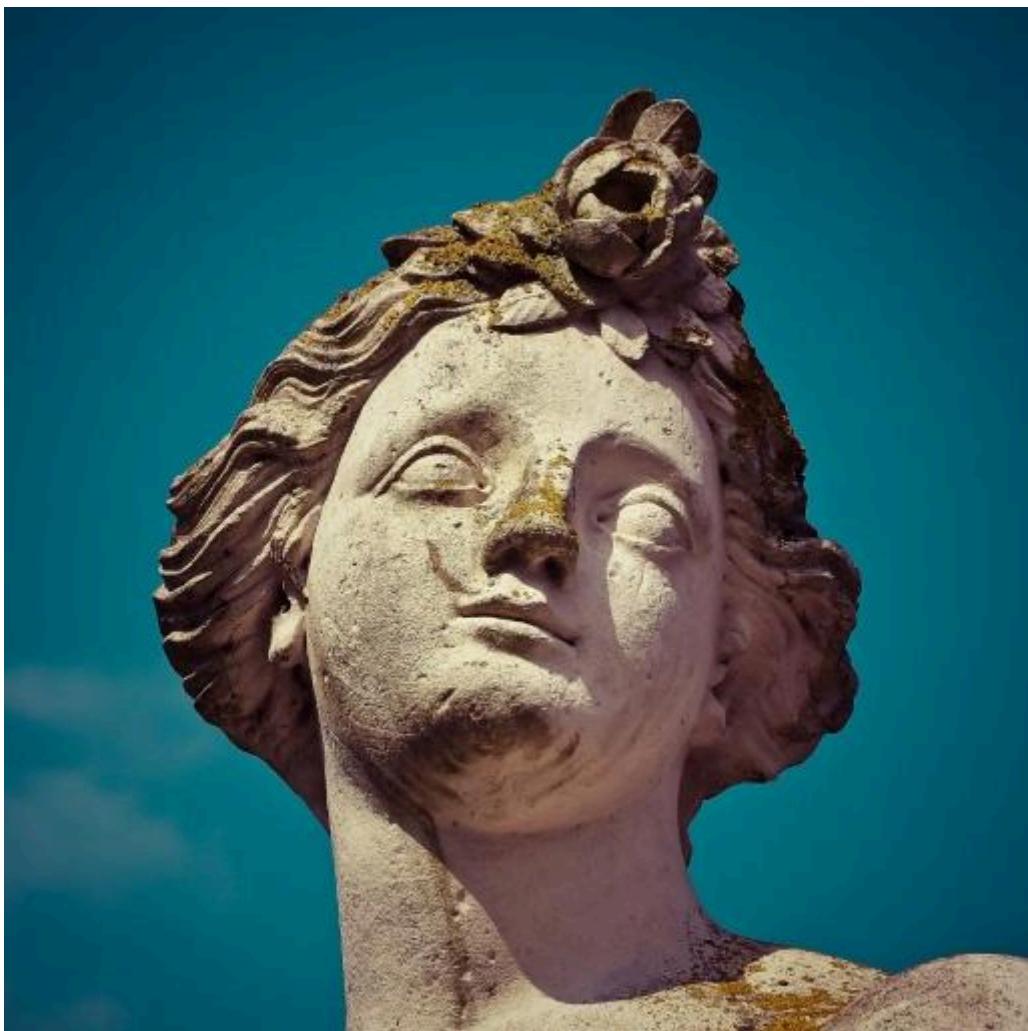
De base, le fichier d'en-tête ne contient que les prototypes des fonctions. Pour aussi avoir le code des fonctions, vous devez ajouter la macro `STB_IMAGE_IMPLEMENTATION`. Sans cela, vous obtiendrez des erreurs lors de l'édition des liens.

```
void initVulkan() {
    ...
    createCommandPool();
    createTextureImage();
    createVertexBuffer();
    ...
}

...
void createTextureImage() {
```

Créez une nouvelle fonction nommée `createTextureImage`, dans laquelle nous chargerons une image et l'environs dans un objet image de Vulkan. Nous allons utiliser des tampons de commandes, donc la fonction doit être appelée après la fonction `createCommandPool()`.

Créez un nouveau dossier nommé `textures` au même endroit que le dossier `shaders` pour y placer les textures. Nous chargerons une image appelée `texture.jpg` qui sera placée dans le nouveau dossier. J'ai choisi d'utiliser **cette image sous licence CC0** redimensionnée à une taille de 512 x 512. Vous pouvez utiliser l'image que vous voulez. La bibliothèque supporte des formats tels que JPEG, PNG, BMP ou GIF.



Il est très facile de charger une image avec la bibliothèque `stb_image` :

```
void createTextureImage() {
    int texWidth, texHeight, texChannels;
    stbi_uc* pixels = stbi_load("textures/texture.jpg", &texWidth, &texHeight, &texChannels,
STBI_rgb_alpha);
    VkDeviceSize imageSize = texWidth * texHeight * 4;

    if (!pixels) {
        throw std::runtime_error("Échec lors du chargement de la texture !");
    }
}
```

La fonction `stbi_load()` prend en argument le chemin de l'image et le nombre de canaux à charger. L'argument `STBI_rgb_alpha` force la présence d'un canal alpha, même si l'image d'origine n'en a pas. Cela simplifie le travail en homogénéisant les situations. Les trois paramètres au milieu permettent de récupérer la largeur, la hauteur et le nombre de canaux réellement présents dans l'image. Le pointeur retourné pointe sur un tableau des pixels de l'image. Les pixels sont agencés ligne par ligne avec quatre octets par pixel (grâce à `STBI_rgb_alpha`). Il y a donc `texWidth * texHeight * 4` pixels.

VII-A-4 - Tampon intermédiaire

Nous allons maintenant créer un tampon accessible par le CPU afin d'utiliser la fonction `vkMapMemory()` et y copier les pixels. Ajoutez les variables pour le tampon intermédiaire dans la fonction `createTextureImage()` :

```
VkBuffer stagingBuffer;
VkDeviceMemory stagingBufferMemory;
```

Le tampon doit être accessible par le CPU et il doit être utilisable comme source de transfert afin de copier les données vers l'image :

```
createBuffer(imageSize, VK_BUFFER_USAGE_TRANSFER_SRC_BIT, VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT |  
VK_MEMORY_PROPERTY_HOST_COHERENT_BIT, stagingBuffer, stagingBufferMemory);
```

Nous pouvons directement copier les pixels provenant de l'image dans le tampon :

```
void* data;  
vkMapMemory(device, stagingBufferMemory, 0, imageSize, 0, &data);  
memcpy(data, pixels, static_cast<size_t>(imageSize));  
vkUnmapMemory(device, stagingBufferMemory);
```

N'oubliez pas de libérer la mémoire du tableau de pixels :

```
stbi_image_free(pixels);
```

VII-A-5 - Texture d'image

Bien qu'il soit possible de paramétriser le shader afin qu'il utilise le tampon comme source pour les pixels, il est préférable d'utiliser l'objet Vulkan dédié à cette utilisation. Les images accélèrent et facilitent la récupération des pixels en utilisant un système de coordonnées 2D. Les pixels d'une image se nomment texels et nous utiliserons ce terme à partir de maintenant. Ajoutez les variables membres suivantes :

```
VkImage textureImage;
VkDeviceMemory textureImageMemory;
```

Les paramètres pour la création d'une image sont spécifiés dans une structure de type **VkImageCreateInfo** :

```
VkImageCreateInfo imageInfo{};
imageInfo.sType = VK_STRUCTURE_TYPE_IMAGE_CREATE_INFO;
imageInfo.imageType = VK_IMAGE_TYPE_2D;
imageInfo.extent.width = static_cast<uint32_t>(texWidth);
imageInfo.extent.height = static_cast<uint32_t>(texHeight);
imageInfo.extent.depth = 1;
imageInfo.mipLevels = 1;
imageInfo.arrayLayers = 1;
```

Le type d'image défini par le champ `imageType` indique à Vulkan quel système de coordonnées utiliser pour accéder aux texels. Il est possible de créer des images 1D, 2D et 3D. Les images 1D peuvent être utilisées comme tableaux pour stocker des données ou pour les dégradés. Les images 2D sont majoritairement utilisées comme textures et les images 3D peuvent être utilisées pour stocker des volumes en voxel. Le champ `extent` indique la taille de l'image, c'est-à-dire combien il y a de texels sur chaque axe. C'est pourquoi nous indiquons 1 et non 0 pour le champ `depth`. Notre texture n'est pas un tableau et nous n'utilisons pas le mipmapping pour le moment.

```
imageInfo.format = VK_FORMAT_R8G8B8A8_SRGB;
```

Vulkan supporte de nombreux formats, mais nous devons utiliser le même format que les données présentes dans le tampon. Sans quoi, l'opération de copie échouera.

```
imageInfo.tiling = VK_IMAGE_TILING_OPTIMAL;
```

Le champ `tiling` peut prendre deux valeurs :

- `VK_IMAGE_TILING_LINEAR` : les texels sont organisés ligne par ligne comme pour notre tableau de pixels ;

- **VK_IMAGE_TILING_OPTIMAL** : l'organisation des texels est déterminée par l'implémentation afin d'optimiser les accès ;

Contrairement à l'agencement de l'image, le mode défini par le champ tiling ne peut pas être changé par la suite. Si vous voulez directement accéder aux texels à partir de la mémoire de l'image, vous devez utiliser **VK_IMAGE_TILING_LINEAR**. Comme nous utilisons un tampon intermédiaire et non une image intermédiaire, nous pouvons utiliser le mode le plus efficace.

```
imageInfo.initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
```

Le champ `initialLayout` peut prendre deux valeurs :

- **VK_IMAGE_LAYOUT_UNDEFINED** : inutilisable par le GPU, son contenu sera éliminé à la première transition ;
- **VK_IMAGE_LAYOUT_PREINITIALIZED** : inutilisable par le GPU, mais la première transition conservera les texels.

Il n'existe que quelques rares situations où il est nécessaire de conserver les texels pendant la première transition. L'une d'elles consiste à utiliser l'image comme ressource intermédiaire en combinaison avec l'agencement **VK_IMAGE_TILING_LINEAR**. Dans ce cas, vous voudriez envoyer les données des texels dans l'image intermédiaire puis effectuer une transition de l'image pour qu'elle devienne source d'un transfert, et ce, sans perdre les données contenues. Dans notre cas, nous allons faire une transition pour définir l'image comme destination d'un transfert, puis y copier les texels provenant de l'objet tampon. Par conséquent, nous n'avons pas besoin de cette propriété et nous pouvons utiliser **VK_IMAGE_LAYOUT_UNDEFINED** sans crainte.

```
imageInfo.usage = VK_IMAGE_USAGE_TRANSFER_DST_BIT | VK_IMAGE_USAGE_SAMPLED_BIT;
```

Le champ `usage` est identique à celui utilisé pour la création d'un tampon. L'image devra être utilisée comme destination d'une opération de copie. Nous souhaitons aussi pouvoir accéder à l'image dans le shader afin de colorier notre modèle, donc nous devons utiliser **VK_IMAGE_USAGE_SAMPLED_BIT**.

```
imageInfo.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
```

L'image sera uniquement utilisée dans une famille de queues : la famille pour les opérations graphiques et implicitement pour les opérations de transfert.

```
imageInfo.samples = VK_SAMPLE_COUNT_1_BIT;
imageInfo.flags = 0; // Optionnel
```

La propriété `sample` est liée au multiéchantillonnage. Il n'a d'utilité que pour les images qui seront utilisées comme attache. Nous pouvons donc laisser un échantillon. Il y a aussi des indicateurs optionnels pour les images liées aux images distinctes. Ces images ont la particularité d'être partiellement contenues en mémoire. Si vous utilisez une image 3D pour représenter un terrain en voxels, vous pouvez éviter d'allouer de la mémoire pour de grandes zones ne contenant que de l'air. Nous n'utilisons pas cette fonctionnalité dans ce tutoriel, laissez donc la valeur par défaut (0).

```
if (vkCreateImage(device, &imageInfo, nullptr, &textureImage) != VK_SUCCESS) {
    throw std::runtime_error("Échec lors de la création de l'image !");
}
```

L'image est créée par la fonction **vkCreateImage()**, qui ne possède pas de paramètres particuliers. Il est possible que le format **VK_FORMAT_R8G8B8A8_SRGB** ne soit pas supporté par la carte graphique. Vous devez trouver une liste d'autres possibilités et choisir la meilleure supportée. Toutefois, comme le format est communément supporté, nous allons passer cette étape. L'utilisation d'autres formats demande de pénibles conversions. Nous reviendrons sur ce point dans le chapitre sur le tampon de profondeur, dans lequel nous allons implémenter un tel mécanisme.

```
VkMemoryRequirements memRequirements;
vkGetImageMemoryRequirements(device, textureImage, &memRequirements);
```

```

VkMemoryAllocateInfo allocInfo{};
allocInfo.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
allocInfo.allocationSize = memRequirements.size;
allocInfo.memoryTypeIndex = findMemoryType(memRequirements.memoryTypeBits,
    VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT);

if (vkAllocateMemory(device, &allocInfo, nullptr, &textureImageMemory) != VK_SUCCESS) {
    throw std::runtime_error("Échec lors de l'allocation de la mémoire pour l'image !");
}

vkBindImageMemory(device, textureImage, textureImageMemory, 0);

```

L'allocation de la mémoire pour l'image fonctionne exactement comme une allocation pour un tampon. Il suffit d'utiliser la fonction **vkGetImageMemoryRequirements()** à la place de **vkGetBufferMemoryRequirements()** et **vkBindImageMemory()** à la place de **vkBindBufferMemory()**.

Cette fonction est déjà longue et nous allons devoir créer plus d'images dans les prochains chapitres. Nous devons donc abstraire la création d'une image dans une fonction dédiée nommée `createImage`. Créez la fonction et déplacez le code lié à la création de l'objet image et de l'allocation mémoire :

```

void createImage(uint32_t width, uint32_t height, VkFormat format, VkImageTiling tiling,
    VkImageUsageFlags usage, VkMemoryPropertyFlags properties, VkImage& image, VkDeviceMemory&
    imageMemory) {
    VkImageCreateInfo imageInfo{};
    imageInfo.sType = VK_STRUCTURE_TYPE_IMAGE_CREATE_INFO;
    imageInfo.imageType = VK_IMAGE_TYPE_2D;
    imageInfo.extent.width = width;
    imageInfo.extent.height = height;
    imageInfo.extent.depth = 1;
    imageInfo.mipLevels = 1;
    imageInfo.arrayLayers = 1;
    imageInfo.format = format;
    imageInfo.tiling = tiling;
    imageInfo.initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
    imageInfo.usage = usage;
    imageInfo.samples = VK_SAMPLE_COUNT_1_BIT;
    imageInfo.sharingMode = VK_SHARING_MODE_EXCLUSIVE;

    if (vkCreateImage(device, &imageInfo, nullptr, &image) != VK_SUCCESS) {
        throw std::runtime_error("Échec de création de l'image !");
    }

    VkMemoryRequirements memRequirements;
    vkGetImageMemoryRequirements(device, image, &memRequirements);

    VkMemoryAllocateInfo allocInfo{};
    allocInfo.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
    allocInfo.allocationSize = memRequirements.size;
    allocInfo.memoryTypeIndex = findMemoryType(memRequirements.memoryTypeBits, properties);

    if (vkAllocateMemory(device, &allocInfo, nullptr, &imageMemory) != VK_SUCCESS) {
        throw std::runtime_error("Échec lors de l'allocation de la mémoire pour l'image !");
    }

    vkBindImageMemory(device, image, imageMemory, 0);
}

```

La largeur, la hauteur, le tiling, l'utilisation et les propriétés de la mémoire deviennent des paramètres, car ils varient suivant les images que nous allons créer dans ce tutoriel.

La fonction `createTextureImage()` peut maintenant être simplifiée :

```

void createTextureImage() {
    int texWidth, texHeight, texChannels;
    stbi_uc* pixels = stbi_load("textures/texture.jpg", &texWidth, &texHeight, &texChannels,
        STBI_rgb_alpha);

```

```

VkDeviceSize imageSize = texWidth * texHeight * 4;

if (!pixels) {
    throw std::runtime_error("Échec du chargement de la texture !");
}

VkBuffer stagingBuffer;
VkDeviceMemory stagingBufferMemory;
createBuffer(imageSize, VK_BUFFER_USAGE_TRANSFER_SRC_BIT,
VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT | VK_MEMORY_PROPERTY_HOST_COHERENT_BIT, stagingBuffer,
stagingBufferMemory);

void* data;
vkMapMemory(device, stagingBufferMemory, 0, imageSize, 0, &data);
memcpy(data, pixels, static_cast<size_t>(imageSize));
vkUnmapMemory(device, stagingBufferMemory);

stbi_image_free(pixels);

createImage(texWidth, texHeight, VK_FORMAT_R8G8B8A8_SRGB, VK_IMAGE_TILING_OPTIMAL,
VK_IMAGE_USAGE_TRANSFER_DST_BIT | VK_IMAGE_USAGE_SAMPLED_BIT,
VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT, textureImage, textureImageMemory);
}

```

VII-A-6 - Transitions de l'agencement

La fonction que nous allons écrire inclut l'enregistrement et l'exécution du tampon de commandes. C'est donc l'occasion de déplacer cette logique dans d'autres fonctions :

```

VkCommandBuffer beginSingleTimeCommands() {
    VkCommandBufferAllocateInfo allocInfo{};
    allocInfo.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_ALLOCATE_INFO;
    allocInfo.level = VK_COMMAND_BUFFER_LEVEL_PRIMARY;
    allocInfo.commandPool = commandPool;
    allocInfo.commandBufferCount = 1;

    VkCommandBuffer commandBuffer;
    vkAllocateCommandBuffers(device, &allocInfo, &commandBuffer);

    VkCommandBufferBeginInfo beginInfo{};
    beginInfo.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO;
    beginInfo.flags = VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT;

    vkBeginCommandBuffer(commandBuffer, &beginInfo);

    return commandBuffer;
}

void endSingleTimeCommands(VkCommandBuffer commandBuffer) {
    vkEndCommandBuffer(commandBuffer);

    VkSubmitInfo submitInfo{};
    submitInfo.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;
    submitInfo.commandBufferCount = 1;
    submitInfo.pCommandBuffers = &commandBuffer;

    vkQueueSubmit(graphicsQueue, 1, &submitInfo, VK_NULL_HANDLE);
    vkQueueWaitIdle(graphicsQueue);

    vkFreeCommandBuffers(device, commandPool, 1, &commandBuffer);
}

```

Le code de ces fonctions est basé sur celui de la fonction `copyBuffer()`. Vous pouvez maintenant simplifier la fonction `copyBuffer()` :

```

void copyBuffer(VkBuffer srcBuffer, VkBuffer dstBuffer, VkDeviceSize size) {
    VkCommandBuffer commandBuffer = beginSingleTimeCommands();

```

```

VkBufferCopy copyRegion{};
copyRegion.size = size;
vkCmdCopyBuffer(commandBuffer, srcBuffer, dstBuffer, 1, &copyRegion);

endSingleTimeCommands(commandBuffer);
}
    
```

Si nous utilisions toujours des tampons, nous aurions pu écrire une fonction pour enregistrer et exécuter la fonction **vkCmdCopyBufferToImage()**. Toutefois, la fonction **vkCmdCopyBufferToImage()** nécessite d'avoir une image agencée correctement. Créez une nouvelle fonction pour gérer les transitions :

```

void transitionImageLayout(VkImage image, VkFormat format, VkImageLayout oldLayout, VkImageLayout newLayout) {
    VkCommandBuffer commandBuffer = beginSingleTimeCommands();

    endSingleTimeCommands(commandBuffer);
}
    
```

L'une des manières de réaliser une transition consiste à utiliser une barrière mémoire d'image. Une telle barrière de pipeline est en général utilisée pour synchroniser l'accès aux ressources, notamment pour s'assurer que l'écriture d'un tampon se termine avant que nous puissions le lire. Mais il est aussi possible de l'utiliser pour les transitions d'agencement d'images ou pour changer le propriétaire d'une famille de queues lorsque **VK_SHARING_MODE_EXCLUSIVE** a été utilisé. Il existe un équivalent pour les tampons : une barrière mémoire de tampon.

```

VkImageMemoryBarrier barrier{};
barrier.sType = VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER;
barrier.oldLayout = oldLayout;
barrier.newLayout = newLayout;
    
```

Les deux premières propriétés indiquent la transition à réaliser. Il est possible d'utiliser **VK_IMAGE_LAYOUT_UNDEFINED** pour le champ **oldLayout** si le contenu de l'image ne vous intéresse pas.

```

barrier.srcQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
barrier.dstQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
    
```

Ces deux paramètres permettent d'indiquer l'indice des familles de queues lors du transfert de propriété d'une famille. Si vous ne faites pas un tel transfert, il faut utiliser la valeur **VK_QUEUE_FAMILY_IGNORED**.

```

barrier.image = image;
barrier.subresourceRange.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
barrier.subresourceRange.baseMipLevel = 0;
barrier.subresourceRange.levelCount = 1;
barrier.subresourceRange.baseArrayLayer = 0;
barrier.subresourceRange.layerCount = 1;
    
```

Les paramètres **image** et **subresourceRange** servent à indiquer l'image et quelle partie de l'image doit être affectée par le changement. Notre image n'est pas un tableau et n'a pas de niveau de mipmaping, donc il n'y a qu'un niveau et qu'une couche.

```

barrier.srcAccessMask = 0; // TODO
barrier.dstAccessMask = 0; // TODO
    
```

Comme les barrières sont avant tout des objets de synchronisation, nous devons indiquer quels types d'opérations en relation avec la ressource doivent être réalisés avant la barrière et quelles opérations doivent attendre la barrière. Nous devons faire cela même si nous utilisons la fonction **vkQueueWaitIdle()**. Les bonnes valeurs dépendent de l'ancien et du nouvel agencement, donc nous reviendrons sur ces propriétés une fois que nous aurons déterminé les transitions à utiliser.

```

vkCmdPipelineBarrier(
    commandBuffer,
    
```

```

0 /* TODO */, 0 /* TODO */,
0,
0, nullptr,
0, nullptr,
1, &barrier
);

```

Tous les types de barrières de pipeline sont envoyés par la même fonction. Le premier paramètre après le tampon de commandes indique dans quelle étape du rendu sont les opérations devant survenir avant la barrière. Le deuxième paramètre indique dans quelle étape sont les opérations qui vont attendre la barrière. Les étapes du pipeline que vous pouvez spécifier avant et après la barrière dépendent de comment vous utilisez la ressource avant et après la barrière. Les valeurs permises sont listées [dans ce tableau](#) provenant de la spécification. Par exemple, si vous allez lire une variable uniforme après la barrière, vous devez indiquer `VK_ACCESS_UNIFORM_READ_BIT` comme usage ainsi que le premier shader à utiliser la variable uniforme. Cela peut être le fragment shader. Dans un tel cas, l'étape à spécifier est `VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT`. Dans ce cas de figure, spécifier une autre étape qu'une étape programmable n'aurait aucun sens, et les couches de validation vous avertissent si vous spécifiez une étape du pipeline qui ne correspond pas à l'usage indiqué.

Le troisième paramètre peut être soit 0 soit `VK_DEPENDENCY_BY_REGION_BIT`. Ce dernier transforme la barrière en une condition par région. Cela signifie que l'implémentation peut déjà commencer la lecture pour les parties de la ressource qui ont déjà été écrites.

Les trois dernières paires de paramètres sont des tableaux de barrières de pipeline pour chacun des trois types existants : barrière mémoire, barrière de tampon et barrière d'image (celle que nous utilisons ici). Notez que nous n'avons pas utilisé le paramètre `VkFormat` pour le moment, mais nous allons l'utiliser pour des transitions spécifiques dans le chapitre du tampon de profondeur.

VII-A-7 - Copie d'un tampon dans une image

Avant de revenir à la fonction `createTextureImage()`, nous allons écrire une nouvelle fonction nommée `copyBufferToImage` :

```

void copyBufferToImage(VkBuffer buffer, VkImage image, uint32_t width, uint32_t height) {
    VkCommandBuffer commandBuffer = beginSingleTimeCommands();
    endSingleTimeCommands(commandBuffer);
}

```

Comme pour les copies de tampons, nous devons spécifier quelles parties du tampon seront copiées dans quelles parties de l'image. Cela se spécifie grâce à une structure de type `VkBufferImageCopy` :

```

VkBufferImageCopy region{};
region.bufferOffset = 0;
region.bufferRowLength = 0;
region.bufferImageHeight = 0;

region.imageSubresource.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
region.imageSubresource.mipLevel = 0;
region.imageSubresource.baseArrayLayer = 0;
region.imageSubresource.layerCount = 1;

region.imageOffset = {0, 0, 0};
region.imageExtent = {
    width,
    height,
    1
};

```

La plupart de ces champs sont évidents. La propriété `bufferOffset` indique l'octet à partir duquel les données des pixels commencent dans le tampon. Les propriétés `bufferRowLength` et `bufferImageHeight` indiquent l'agencement des pixels en mémoire. Par exemple, vous pourriez avoir des octets de remplissage entre les lignes d'une image.

En spécifiant 0 pour les deux champs, nous indiquons que les pixels se suivent. Les propriétés `imageSubResource`, `imageOffset` et `imageExtent` indiquent quelle partie de l'image recevra les données.

Les opérations de copie d'un tampon vers une image sont envoyées à la queue avec la fonction `vkCmdCopyBufferToImage()`.

```
vkCmdCopyBufferToImage(
    commandBuffer,
    buffer,
    image,
    VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL,
    1,
    &region
);
```

Le quatrième paramètre indique l'organisation de l'image au moment de la copie. Je suppose que l'image a déjà été réorganisée pour avoir un agencement optimal pour effectuer une copie des pixels. Pour le moment, nous copions tous les pixels de l'image en une opération, mais par la suite, il est possible de spécifier un tableau de `VkBufferImageCopy` pour effectuer plusieurs copies différentes du tampon vers l'image.

VII-A-8 - Préparer la texture

Nous avons maintenant tous les outils nécessaires pour compléter la mise en place de la texture d'image. Nous pouvons retourner à la fonction `createTextureImage()`. La dernière chose que nous avons ajoutée dans cette fonction est la création de la texture. La prochaine étape est de copier le tampon intermédiaire vers la texture. Cela s'effectue en deux étapes :

- passer la texture à l'agencement `VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL` ;
- exécuter la copie du tampon vers l'image.

Ces opérations sont simples grâce aux fonctions que nous venons de créer :

```
transitionImageLayout(textureImage, VK_FORMAT_R8G8B8A8_SRGB, VK_IMAGE_LAYOUT_UNDEFINED,
    VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL);
copyBufferToImage(stagingBuffer,
    textureImage, static_cast<uint32_t>(texWidth), static_cast<uint32_t>(texHeight));
```

Nous avons créé l'image avec un agencement `VK_LAYOUT_UNDEFINED`. Nous devons donc spécifier cette valeur comme ancien agencement de `textureImage`. Souvenez-vous que vous devez faire cela, car nous ne sommes pas intéressés par le contenu avant la copie.

Pour pouvoir échantillonner la texture depuis le fragment shader, nous devons réaliser une dernière transition afin de la préparer pour que le shader y accède :

```
transitionImageLayout(textureImage, VK_FORMAT_R8G8B8A8_SRGB,
    VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL, VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL);
```

VII-A-9 - Masque de barrière de transition

Si vous lancez le programme avec les couches de validation, vous verriez des messages à propos de masque d'accès et des étapes du pipeline invalides dans la fonction `transitionImageLayout()`. Nous devons les définir selon les agencements dans la transition.

Nous devons gérer deux transitions :

- indéfini → destination d'un transfert : les écritures du transfert n'ont pas besoin d'attendre quoi que ce soit ;

- destination d'un transfert → lecture par un shader : la lecture par le shader doit attendre la fin du transfert.
Plus précisément, ce sont les lectures effectuées depuis le fragment shader qui doivent attendre pour la fin du transfert, car c'est dans celui-ci que nous utilisons la texture.

Ces règles sont indiquées en utilisant les valeurs suivantes pour le masque d'accès et les étapes du pipeline :

```
VkPipelineStageFlags sourceStage;
VkPipelineStageFlags destinationStage;

if (oldLayout == VK_IMAGE_LAYOUT_UNDEFINED && newLayout ==
VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL) {
    barrier.srcAccessMask = 0;
    barrier.dstAccessMask = VK_ACCESS_TRANSFER_WRITE_BIT;

    sourceStage = VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT;
    destinationStage = VK_PIPELINE_STAGE_TRANSFER_BIT;
} else if (oldLayout == VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL && newLayout ==
VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL) {
    barrier.srcAccessMask = VK_ACCESS_TRANSFER_WRITE_BIT;
    barrier.dstAccessMask = VK_ACCESS_SHADER_READ_BIT;

    sourceStage = VK_PIPELINE_STAGE_TRANSFER_BIT;
    destinationStage = VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT;
} else {
    throw std::invalid_argument("Transition de l'agencement non supportée !");
}

vkCmdPipelineBarrier(
    commandBuffer,
    sourceStage, destinationStage,
    0,
    0, nullptr,
    0, nullptr,
    1, &barrier
);
```

Comme vous avez pu le voir dans le tableau mentionné plus haut, les écritures dans l'image doivent se réaliser à l'étape de transfert du pipeline. Comme l'écriture ne dépend d'aucune autre opération, nous pouvons spécifier un masque d'accès vide et la première étape du pipeline : `VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT` comme opérations avant la barrière. La valeur `VK_PIPELINE_STAGE_TRANSFER_BIT` ne référence pas une étape existante dans les pipelines graphiques ou de calcul. C'est plutôt une pseudo-étape où les transferts sont effectués. Lisez [la documentation](#) pour de plus amples informations et d'autres exemples de pseudo-étapes.

L'image sera écrite dans la même étape de pipeline puis lue par le fragment shader. C'est pourquoi nous spécifions un accès en lecture pour le shader dans l'étape du pipeline liée au fragment shader.

Nous étendrons cette fonction dans le futur pour prendre en compte plus de transitions. L'application doit maintenant fonctionner sans problème, bien qu'il n'y ait aucune différence visible.

Il est à noter que la soumission du tampon de commandes génère une synchronisation implicite de type `VK_ACCESS_HOST_WRITE_BIT`. Comme la fonction `transitionImageLayout()` exécute un tampon de commandes avec une seule commande, vous pouvez utiliser cette synchronisation implicite et définir `srcAccessMask` à 0 si vous avez besoin d'une dépendance à `VK_ACCESS_HOST_WRITE_BIT` dans la transition d'agencement. C'est à vous de voir si vous voulez être explicite ou non. Personnellement, je n'aime pas reposer sur ce type d'opérations « cachées » rappelant OpenGL.

Il existe un type d'agencement d'image supportant toutes les opérations : `VK_IMAGE_LAYOUT_GENERAL`. Le problème est qu'il est évidemment moins optimisé. Il est cependant utile dans certains cas, notamment lorsqu'une image est à la fois une entrée et une sortie ou pour lire une image après qu'elle a quitté l'agencement avec initialisation.

Toutes les fonctions envoyant des commandes ont été configurées pour s'exécuter de manière synchrone et attendre que la queue n'ait plus de travail. Dans les applications réelles, il est recommandé d'assembler ces opérations dans un

unique tampon de commandes et de les exécuter de manière asynchrone afin d'obtenir un meilleur débit, notamment pour les transitions et la copie effectuées dans la fonction `createTextureImage()`. Essayez de le faire en créant une fonction nommée `setupCommandBuffer` qui sera utilisée pour enregistrer les commandes des fonctions utilitaires et ajoutez une fonction `flushSetupCommands()` pour exécuter les commandes enregistrées jusqu'alors. Une telle transformation du programme doit être faite après avoir finalisé l'implémentation de l'application des textures afin de vérifier que tout fonctionne.

VII-A-10 - Nettoyage

Complétez la fonction `createlImageTexture()` en libérant le tampon intermédiaire et la mémoire allouée pour celui-ci :

```
transitionImageLayout(textureImage, VK_FORMAT_R8G8B8A8_SRGB,
VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL, VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL);

vkDestroyBuffer(device, stagingBuffer, nullptr);
vkFreeMemory(device, stagingBufferMemory, nullptr);
}
```

La texture est utilisée jusqu'à la fin du programme, nous devons donc la libérer dans la fonction `cleanup()` :

```
void cleanup() {
    cleanupSwapChain();

    vkDestroyImage(device, textureImage, nullptr);
    vkFreeMemory(device, textureImageMemory, nullptr);

    ...
}
```

L'image contient maintenant la texture, mais nous n'avons toujours pas mis en place une méthode pour y accéder depuis le pipeline graphique. C'est l'objectif du prochain chapitre.

C++ code / Vertex shader / Fragment shader

VII-B - Vue d'image et échantillonneur

Dans ce chapitre, nous allons créer deux nouvelles ressources nécessaires pour pouvoir échantillonner une image depuis le pipeline graphique. Nous avons déjà travaillé avec la première ressource, notamment en conjonction avec les images de la « swap chain ». La seconde ressource est nouvelle. Elle est liée à la manière selon laquelle le shader accédera aux texels de l'image.

VII-B-1 - Vue sur une texture

Comme nous l'avons vu précédemment avec les images de la « swap chain » et le tampon d'images, les images ne peuvent être accédées qu'au travers de vues. Nous aurons donc besoin de créer une vue pour la texture.

Ajoutez une variable membre pour stocker la référence à l'objet de type `VkImageView`. Ajoutez aussi une fonction nommée `createTextureImageView()` dans laquelle nous créerons la vue :

```
VkImageView textureImageView;

...
void initVulkan() {
    ...
    createTextureImage();
    createTextureImageView();
    createVertexBuffer();
```

```

    ...
}

...
void createTextureImageView() {
}

```

Le code de cette fonction peut être basé sur la fonction `createImageViews()`. Les deux seuls changements sont dans les propriétés `format` et `image` :

```

VkImageViewCreateInfo viewInfo{};
viewInfo.sType = VK_STRUCTURE_TYPE_IMAGE_VIEW_CREATE_INFO;
viewInfo.image = textureImage;
viewInfo.viewType = VK_IMAGE_VIEW_TYPE_2D;
viewInfo.format = VK_FORMAT_R8G8B8A8_SRGB;
viewInfo.subresourceRange.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
viewInfo.subresourceRange.baseMipLevel = 0;
viewInfo.subresourceRange.levelCount = 1;
viewInfo.subresourceRange.baseArrayLayer = 0;
viewInfo.subresourceRange.layerCount = 1;

```

Je n'ai pas initialisé explicitement la propriété `viewInfo.components`, la valeur de `VK_COMPONENT_SWIZZLE_IDENTITY` est 0. Finissez la création de la vue en appelant la fonction `vkCreateImageView()` :

```

if (vkCreateImageView(device, &viewInfo, nullptr, &textureImageView) != VK_SUCCESS) {
    throw std::runtime_error("Échec de création de la vue pour la texture !");
}

```

Comme la logique est similaire à celle de la fonction `createImageViews()`, vous pourriez vouloir abstraire la création d'une vue dans une nouvelle fonction `createView()` :

```

VkImageView createImageView(VkImage image, VkFormat format) {
    VkImageViewCreateInfo viewInfo{};
    viewInfo.sType = VK_STRUCTURE_TYPE_IMAGE_VIEW_CREATE_INFO;
    viewInfo.image = image;
    viewInfo.viewType = VK_IMAGE_VIEW_TYPE_2D;
    viewInfo.format = format;
    viewInfo.subresourceRange.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
    viewInfo.subresourceRange.baseMipLevel = 0;
    viewInfo.subresourceRange.levelCount = 1;
    viewInfo.subresourceRange.baseArrayLayer = 0;
    viewInfo.subresourceRange.layerCount = 1;

    VkImageView imageView;
    if (vkCreateImageView(device, &viewInfo, nullptr, &imageView) != VK_SUCCESS) {
        throw std::runtime_error("Échec de création de la vue !");
    }

    return imageView;
}

```

La fonction `createTextureImageView()` peut maintenant être simplifiée :

```

void createTextureImageView() {
    textureImageView = createImageView(textureImage, VK_FORMAT_R8G8B8A8_SRGB);
}

```

Ainsi que la fonction `createImageView()` :

```

void createImageViews() {
    swapChainImageViews.resize(swapChainImages.size());
}

```

```

for (uint32_t i = 0; i < swapChainImages.size(); i++) {
    swapChainImageViews[i] = createImageView(swapChainImages[i], swapChainImageFormat);
}
    
```

Assurons-nous de détruire la vue à la fin du programme, juste avant la destruction de l'image elle-même :

```

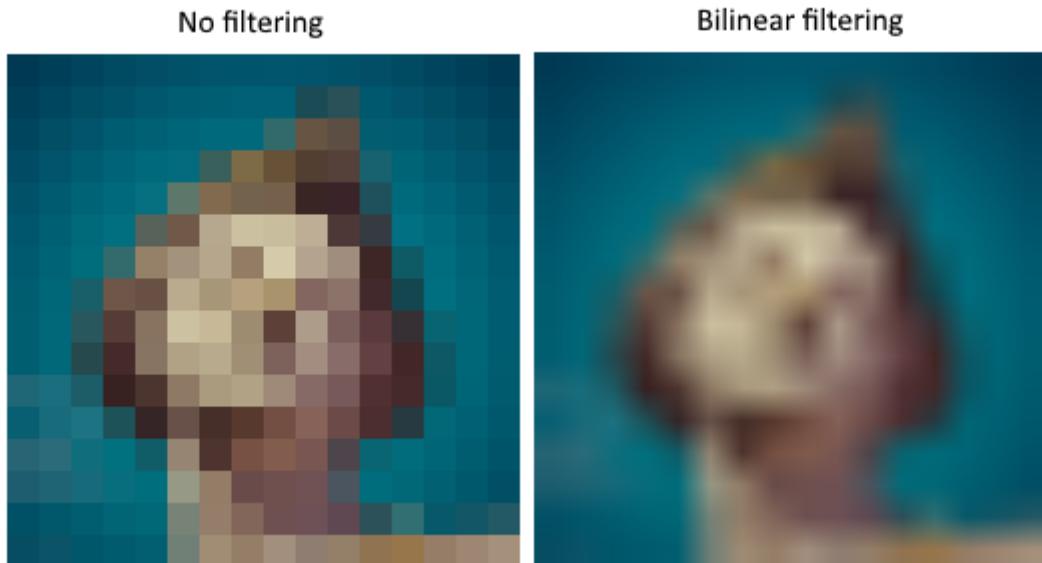
void cleanup() {
    cleanupSwapChain();

    vkDestroyImageView(device, textureImageView, nullptr);
    vkDestroyImage(device, textureImage, nullptr);
    vkFreeMemory(device, textureImageMemory, nullptr);
}
    
```

VII-B-2 - Échantillonneurs

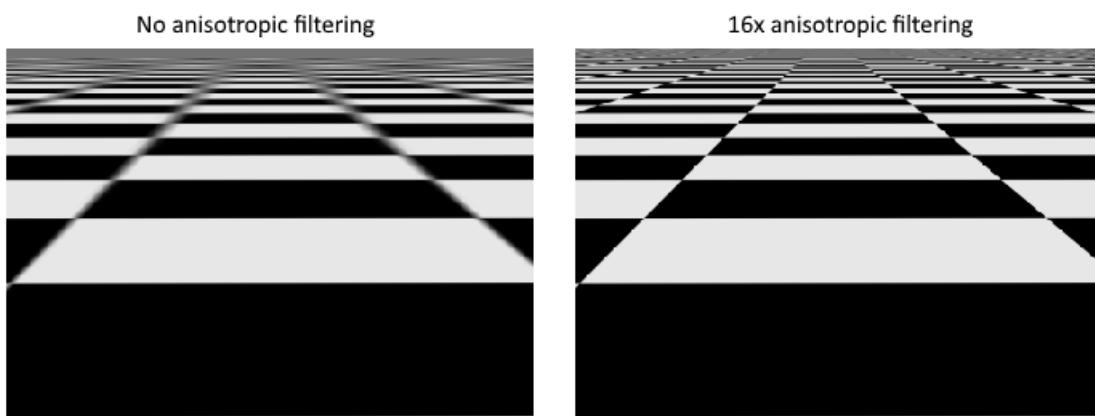
Les shaders peuvent lire les texels directement à partir des images. Toutefois, ce n'est pas la technique habituelle pour les textures. Les textures sont généralement accédées à travers un échantillonneur (sampler) qui filtrera et/ou transformera les données afin de calculer la couleur finale à retourner au shader.

Ces filtres sont utiles pour résoudre des problèmes tels que le suréchantillonnage. Imaginez une texture appliquée sur une géométrie possédant plus de fragments que la texture n'a de texels. Si l'échantillonneur se contentait de prendre le pixel le plus proche, une pixellisation apparaîtrait :



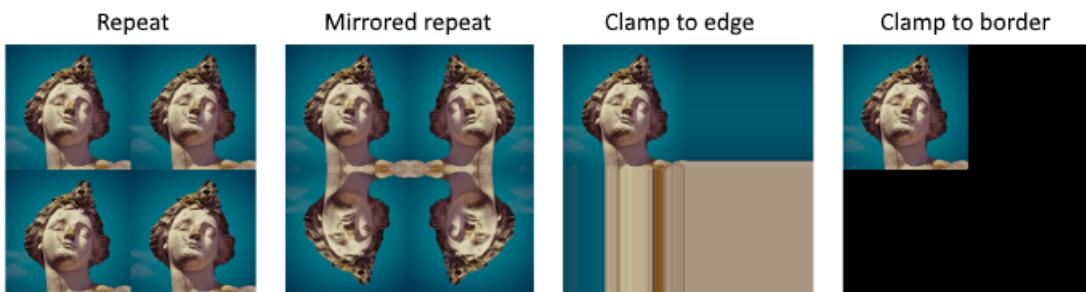
En combinant les quatre texels les plus proches grâce à une interpolation linéaire, il est possible d'obtenir un rendu lisse comme présenté sur l'image de droite. Bien sûr, il est possible que votre application cherche plutôt à obtenir le premier résultat (comme Minecraft), mais la seconde option est plus couramment utilisée. Un échantillonneur applique alors automatiquement ce type d'opérations pour vous lors de la lecture d'une couleur à partir de la texture.

Le sous-échantillonnage est le problème inverse : vous avez plus de texels que de fragments. Cela crée des artefacts particulièrement visibles dans le cas de textures ayant des motifs répétés (comme un damier) vues à un angle aigu :



Comme vous pouvez le voir sur l'image de gauche, la texture devient floue au loin. La solution à ce problème est le **filtrage anisotrope**, qui peut aussi être automatiquement appliqué par l'échantillonneur.

Au-delà de ces filtres, l'échantillonneur peut aussi s'occuper de transformations. Il détermine ce qui se passe lorsque vous essayez de lire des texels en dehors des limites de l'image. L'image ci-dessous montre quelques exemples de configurations :



Nous allons maintenant créer la fonction nommée `createTextureSampler` pour mettre en place un échantillonneur simple. Nous l'utiliserons pour lire les couleurs de la texture à partir du shader.

```
void initVulkan() {
    ...
    createTextureImage();
    createTextureImageView();
    createTextureSampler();
    ...
}

...

void createTextureSampler() {
}
```

Les échantillonneurs se configurent par le biais d'une structure de type **VkSamplerCreateInfo**. Ce type de structure permet d'indiquer les filtres et les transformations à appliquer.

```
VkSamplerCreateInfo samplerInfo{};
samplerInfo.sType = VK_STRUCTURE_TYPE_SAMPLER_CREATE_INFO;
samplerInfo.magFilter = VK_FILTER_LINEAR;
samplerInfo.minFilter = VK_FILTER_LINEAR;
```

Les champs `magFilter` et `minFilter` indiquent comment interpoler les texels dans les cas respectivement d'un grossissement et d'une réduction. Cela permet de gérer les cas vus plus haut. Nous avons le choix entre

VK_FILTER_NEAREST et VK_FILTER_LINEAR dont le rendu correspond aux images gauche et droite du premier exemple.

```
samplerInfo.addressModeU = VK_SAMPLER_ADDRESS_MODE_REPEAT;
samplerInfo.addressModeV = VK_SAMPLER_ADDRESS_MODE_REPEAT;
samplerInfo.addressModeW = VK_SAMPLER_ADDRESS_MODE_REPEAT;
```

Le mode d'adressage peut être configuré pour chaque axe grâce aux champs addressMode. Les valeurs possibles sont listées ci-dessous. Notez que les axes sont nommés U, V et W à la place de X, Y et Z. C'est la convention pour les coordonnées de texture.

- VK_SAMPLER_ADDRESS_MODE_REPEAT : répète la texture lors d'accès hors des dimensions de l'image ;
- VK_SAMPLER_ADDRESS_MODE_MIRRORRED_REPEAT : similaire à la répétition, mais inverse les coordonnées pour faire une image miroir lors d'un accès hors des dimensions de l'image ;
- VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE : prend la couleur du pixel de la bordure la plus proche ;
- VK_SAMPLER_ADDRESS_MODE_MIRROR_CLAMP_TO_EDGE : similaire au mode précédent, mais en utilisant la bordure opposée à la bordure la plus proche ;
- VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_BORDER : retourne une couleur déterminée lors d'un accès hors des dimensions de l'image.

Le mode d'adressage que nous utilisons n'est pas très important, car nous ne dépasserons pas les dimensions de l'image dans ce tutoriel. Cependant, le mode de répétition est le plus commun, car il permet de pavier les sols et les murs.

```
samplerInfo.anisotropyEnable = VK_TRUE;
samplerInfo.maxAnisotropy = 16.0f;
```

Ces deux propriétés paramètrent l'utilisation du filtrage anisotrope. Il n'y a pas vraiment de raison de ne pas l'utiliser, sauf si vous manquez de performances. Le champ maxAnisotropy est le nombre maximal de texels pouvant être utilisés pour calculer la couleur finale. Une valeur plus faible donne de meilleures performances, mais une qualité moindre. Il n'existe à ce jour aucune carte graphique pouvant utiliser plus de 16 texels, car au-delà, l'amélioration est négligeable.

```
samplerInfo.borderColor = VK_BORDER_COLOR_INT_OPAQUE_BLACK;
```

Le paramètre borderColor indique la couleur à retourner lors d'un accès hors des dimensions de l'image pour le mode d'adressage VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_BORDER. Il est possible d'indiquer du noir, du blanc ou du transparent par le biais d'un nombre à virgule flottante ou d'un nombre entier. Vous ne pouvez pas indiquer une couleur arbitraire.

```
samplerInfo.unnormalizedCoordinates = VK_FALSE;
```

Le champ unnormalizedCoordinates indique le système de coordonnées que vous souhaitez utiliser pour accéder aux texels dans une image. Avec la valeur VK_TRUE, vous pouvez utiliser des coordonnées comprises entre [0, texWidth] et [0, texHeight]. Sinon, les valeurs sont accessibles avec des coordonnées comprises dans l'intervalle [0, 1] pour tous les axes. La majorité des applications utilise des coordonnées normalisées. Ainsi, il est possible d'utiliser des textures avec des résolutions différentes tout en conservant les mêmes coordonnées.

```
samplerInfo.compareEnable = VK_FALSE;
samplerInfo.compareOp = VK_COMPARE_OP_ALWAYS;
```

Si une fonction de comparaison est activée, les texels seront d'abord comparés à une valeur. Le résultat de la comparaison est ensuite utilisé dans les opérations de filtrage. Cette fonctionnalité est principalement utilisée pour réaliser un **filtrage au pourcentage le plus proche** sur les textures d'ombrages. Nous verrons cela dans un futur chapitre.

```
samplerInfo.mipmapMode = VK_SAMPLER_MIPMAP_MODE_LINEAR;
samplerInfo.mipLodBias = 0.0f;
samplerInfo.minLod = 0.0f;
samplerInfo.maxLod = 0.0f;
```

Tous ces champs sont liés au mipmapping. Nous y reviendrons dans un [prochain chapitre](#), mais pour faire simple, c'est encore un autre type de filtrage.

L'échantillonneur est maintenant complètement configuré. Ajoutez une variable membre pour stocker la référence à cet échantillonneur, puis créez-le avec la fonction `vkCreateSampler()` :

```
VkImageView textureImageView;
VkSampler textureSampler;

...

void createTextureSampler() {
    ...

    if (vkCreateSampler(device, &samplerInfo, nullptr, &textureSampler) != VK_SUCCESS) {
        throw std::runtime_error("Échec lors de la création de l'échantillonneur !");
    }
}
```

Remarquez que l'échantillonneur ne référence aucun objet de type `VkImage`. Il constitue un objet distinct fourni par une interface pour extraire les couleurs d'une texture. Il peut être utilisé avec n'importe quelle image 1D, 2D ou 3D. Cela diffère des anciennes bibliothèques qui combinaient la texture et son filtrage en un seul état.

Détruisons l'échantillonneur à la fin du programme, là où nous n'accéderons plus à l'image :

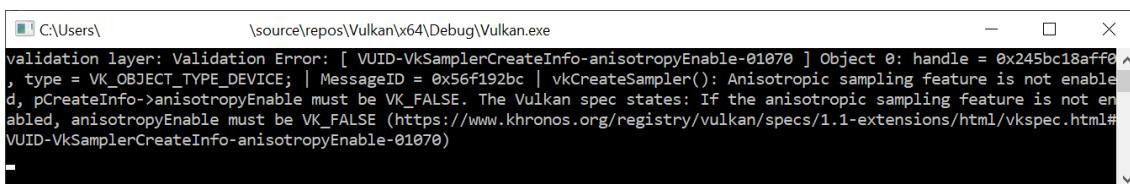
```
void cleanup() {
    cleanupSwapChain();

    vkDestroySampler(device, textureSampler, nullptr);
    vkDestroyImageView(device, textureImageView, nullptr);

    ...
}
```

VII-B-3 - Fonctionnalités de filtrage anisotrope du périphérique

Si vous lancez le programme, vous verrez que les couches de validation vous envoient un message similaire à celui-ci :



En effet, le filtrage anisotrope est une fonctionnalité optionnelle du périphérique. Nous devons donc mettre à jour la fonction `createLogicalDevice()` pour obtenir l'accès à cette fonctionnalité :

```
VkPhysicalDeviceFeatures deviceFeatures{};
deviceFeatures.samplerAnisotropy = VK_TRUE;
```

Et bien qu'il soit très peu probable qu'une carte graphique moderne ne supporte pas cette fonctionnalité, nous devons aussi mettre à jour la fonction `isDeviceSuitable()` pour vérifier la présence de la fonctionnalité :

```

bool isDeviceSuitable(VkPhysicalDevice device) {
    ...

    VkPhysicalDeviceFeatures supportedFeatures;
    vkGetPhysicalDeviceFeatures(device, &supportedFeatures);

    return indices.isComplete() && extensionsSupported && swapChainAdequate &&
    supportedFeatures.samplerAnisotropy;
}
    
```

La fonction **vkGetPhysicalDeviceFeatures()** réutilise la structure **VkPhysicalDeviceFeatures** pour indiquer quelles sont les fonctionnalités supportées plutôt que de les récupérer en utilisant des valeurs booléennes.

Au lieu d'obliger le client à posséder une carte graphique supportant le filtrage anisotope, il est aussi possible de ne pas l'utiliser :

```

samplerInfo.anisotropyEnable = VK_FALSE;
samplerInfo.maxAnisotropy = 1;
    
```

Dans le prochain chapitre, nous exposerons l'image et l'échantillonneur au fragment shader pour dessiner la texture sur le carré.

C++ code / Vertex shader / Fragment shader

VII-C - Association d'échantillonneur et d'image

VII-C-1 - Introduction

Nous avons vu, pour la première fois, les descripteurs dans le tutoriel sur les tampons de variables uniformes. Dans ce chapitre, nous verrons un nouveau type de descripteurs : l'échantillonneur combiné d'image (combined image sampler). Ce descripteur permet aux shaders d'accéder à une image à l'aide d'un échantillonneur comme celui que nous avons créé dans le chapitre précédent.

Nous allons d'abord modifier l'agencement du descripteur, le groupe de descripteurs et l'ensemble de descripteurs pour inclure un descripteur d'échantillonneur et d'image associés. Ensuite, nous ajouterons des coordonnées de texture à la structure Vertex et modifierons le fragment shader pour qu'il lise les couleurs à partir de la texture et non plus à partir des couleurs de sommets interpolés.

VII-C-2 - Modifier les descripteurs

Trouvez la fonction `createDescriptorSetLayout()` et ajoutez une instance à la structure **VkDescriptorSetLayoutBinding** pour le descripteur d'échantillonneur combiné d'image. Nous allons placer la fonction dans la liaison après le tampon uniforme :

```

VkDescriptorSetLayoutBinding samplerLayoutBinding{};
samplerLayoutBinding.binding = 1;
samplerLayoutBinding.descriptorCount = 1;
samplerLayoutBinding.descriptorType = VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER;
samplerLayoutBinding.pImmutableSamplers = nullptr;
samplerLayoutBinding.stageFlags = VK_SHADER_STAGE_FRAGMENT_BIT;

std::array<VkDescriptorSetLayoutBinding, 2> bindings = {uboLayoutBinding, samplerLayoutBinding};
VkDescriptorSetLayoutCreateInfo layoutInfo{};
layoutInfo.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO;
layoutInfo.bindingCount = static_cast<uint32_t>(bindings.size());
layoutInfo.pBindings = bindings.data();
    
```

Assurez-vous de définir le champ `stageFlags` pour indiquer notre intention d'utiliser un descripteur d'échantillonneur combiné d'image dans le fragment shader. C'est ici que la couleur du fragment sera déterminée. Il est possible d'échantillonner une texture dans le vertex shader, notamment pour déformer dynamiquement une grille de sommets dans un **champ de hauteur**.

Si vous lancez l'application avec les couches de validation, vous verrez des messages indiquant que le groupe de descripteur ne peut pas allouer d'ensemble pour cet agencement. En effet, elle ne comprend aucun descripteur d'échantillonneur combiné d'image. Allez à la fonction `createDescriptorPool()` pour inclure une structure `VkDescriptorPoolSize` pour ce descripteur :

```
std::array<VkDescriptorPoolSize, 2> poolSizes{};
poolSizes[0].type = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
poolSizes[0].descriptorCount = static_cast<uint32_t>(swapChainImages.size());
poolSizes[1].type = VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER;
poolSizes[1].descriptorCount = static_cast<uint32_t>(swapChainImages.size());

VkDescriptorPoolCreateInfo poolInfo{};
poolInfo.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_POOL_CREATE_INFO;
poolInfo.poolSizeCount = static_cast<uint32_t>(poolSizes.size());
poolInfo.pPoolSizes = poolSizes.data();
poolInfo.maxSets = static_cast<uint32_t>(swapChainImages.size());
```

La dernière étape consiste à lier l'image et l'échantillonneur aux descripteurs de l'ensemble de descripteurs. Allez à la fonction `createDescriptorSets()`.

```
for (size_t i = 0; i < swapChainImages.size(); i++) {
    VkDescriptorBufferInfo bufferInfo{};
    bufferInfo.buffer = uniformBuffers[i];
    bufferInfo.offset = 0;
    bufferInfo.range = sizeof(UniformBufferObject);

    VkDescriptorImageInfo imageInfo{};
    imageInfo.imageLayout = VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL;
    imageInfo.imageView = textureImageView;
    imageInfo.sampler = textureSampler;

    ...
}
```

Les ressources nécessaires à la structure paramétrant un descripteur d'échantillonneur combiné d'image doivent être fournies dans une structure de type `VkDescriptorImageInfo`. C'est similaire à la structure de type `VkDescriptorBufferInfo` utilisée pour les tampons. C'est à cet endroit que les objets du chapitre précédents s'assemblent.

```
std::array<VkWriteDescriptorSet, 2> descriptorWrites{};

descriptorWrites[0].sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
descriptorWrites[0].dstSet = descriptorSets[i];
descriptorWrites[0].dstBinding = 0;
descriptorWrites[0].dstArrayElement = 0;
descriptorWrites[0].descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
descriptorWrites[0].descriptorCount = 1;
descriptorWrites[0].pBufferInfo = &bufferInfo;

descriptorWrites[1].sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
descriptorWrites[1].dstSet = descriptorSets[i];
descriptorWrites[1].dstBinding = 1;
descriptorWrites[1].dstArrayElement = 0;
descriptorWrites[1].descriptorType = VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER;
descriptorWrites[1].descriptorCount = 1;
descriptorWrites[1].pImageInfo = &imageInfo;

vkUpdateDescriptorSets(device, static_cast<uint32_t>(descriptorWrites.size()),
    descriptorWrites.data(), 0, nullptr);
```

Les descripteurs doivent être mis à jour avec les informations sur l'image, comme pour le tampon. Cette fois, nous allons utiliser le tableau `pImageInfo` au lieu du tableau `pBufferInfo`. Les descripteurs sont maintenant prêts à être utilisés dans les shaders !

VII-C-3 - Coordonnées de texture

Il manque encore un élément important pour appliquer une texture : les coordonnées de texture de chaque sommet. Les coordonnées déterminent comment appliquer l'image sur la géométrie.

```
struct Vertex {
    glm::vec2 pos;
    glm::vec3 color;
    glm::vec2 texCoord;

    static VkVertexInputBindingDescription getBindingDescription() {
        VkVertexInputBindingDescription bindingDescription{};
        bindingDescription.binding = 0;
        bindingDescription.stride = sizeof(Vertex);
        bindingDescription.inputRate = VK_VERTEX_INPUT_RATE_VERTEX;

        return bindingDescription;
    }

    static std::array<VkVertexInputAttributeDescription, 3> getAttributeDescriptions() {
        std::array<VkVertexInputAttributeDescription, 3> attributeDescriptions{};

        attributeDescriptions[0].binding = 0;
        attributeDescriptions[0].location = 0;
        attributeDescriptions[0].format = VK_FORMAT_R32G32_SFLOAT;
        attributeDescriptions[0].offset = offsetof(Vertex, pos);

        attributeDescriptions[1].binding = 0;
        attributeDescriptions[1].location = 1;
        attributeDescriptions[1].format = VK_FORMAT_R32G32B32_SFLOAT;
        attributeDescriptions[1].offset = offsetof(Vertex, color);

        attributeDescriptions[2].binding = 0;
        attributeDescriptions[2].location = 2;
        attributeDescriptions[2].format = VK_FORMAT_R32G32_SFLOAT;
        attributeDescriptions[2].offset = offsetof(Vertex, texCoord);

        return attributeDescriptions;
    }
};
```

Modifiez la structure `Vertex` pour ajouter une variable de type `vec2`, pour stocker les coordonnées de texture. Ajoutez également un `VkVertexInputAttributeDescription` afin que ces coordonnées puissent être accédées en entrée du vertex shader. Il est nécessaire de les passer du vertex shader au fragment shader afin que les coordonnées soient interpolées sur la surface du carré.

```
const std::vector<Vertex> vertices = {
    {{-0.5f, -0.5f}, {1.0f, 0.0f, 0.0f}, {0.0f, 0.0f}},
    {{0.5f, -0.5f}, {0.0f, 1.0f, 0.0f}, {1.0f, 0.0f}},
    {{0.5f, 0.5f}, {0.0f, 0.0f, 1.0f}, {1.0f, 1.0f}},
    {{-0.5f, 0.5f}, {1.0f, 1.0f, 1.0f}, {0.0f, 1.0f}}
};
```

Dans ce tutoriel, je vais simplement remplir le carré avec la texture en utilisant la coordonnée (0, 0) pour le coin haut gauche et la coordonnée (1, 1) pour le coin bas droit. Essayez des coordonnées différentes ! Essayez aussi des coordonnées inférieures à 0 et supérieures à 1 pour constater les implications des modes d'adressage.

VII-C-4 - Shaders

La dernière étape consiste à modifier les shaders pour récupérer les couleurs de la texture. Nous devons d'abord modifier le vertex shader pour copier les coordonnées de texture en sortie à destination du fragment shader :

```
layout(location = 0) in vec2 inPosition;
layout(location = 1) in vec3 inColor;
layout(location = 2) in vec2 inTexCoord;

layout(location = 0) out vec3 fragColor;
layout(location = 1) out vec2 fragTexCoord;

void main() {
    gl_Position = ubo.proj * ubo.view * ubo.model * vec4(inPosition, 0.0, 1.0);
    fragColor = inColor;
    fragTexCoord = inTexCoord;
}
```

Comme pour les couleurs définies par sommet, les valeurs de fragTexCoord seront interpolées sur la surface du carré par le rastériseur. Nous pouvons visualiser cela en écrivant un fragment shader qui retourne comme couleurs les coordonnées de texture :

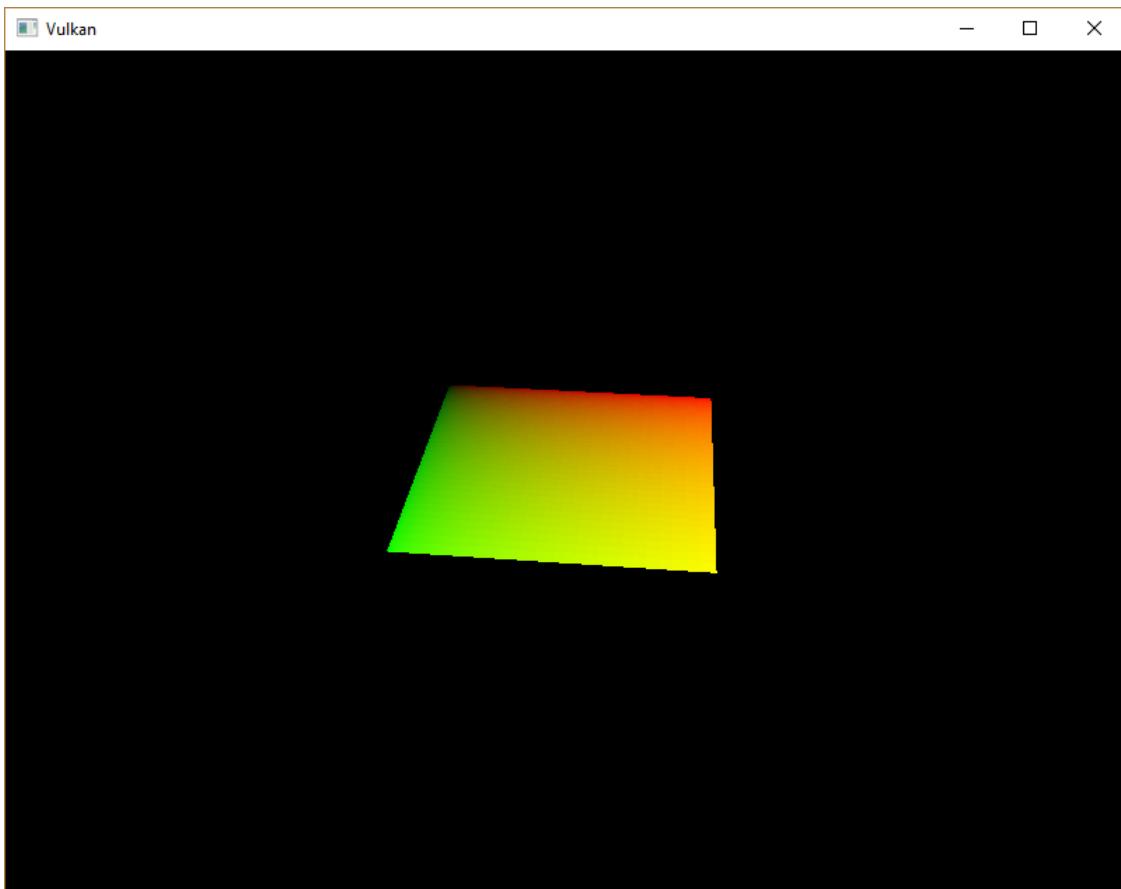
```
#version 450
#extension GL_ARB_separate_shader_objects : enable

layout(location = 0) in vec3 fragColor;
layout(location = 1) in vec2 fragTexCoord;

layout(location = 0) out vec4 outColor;

void main() {
    outColor = vec4(fragTexCoord, 0.0, 1.0);
}
```

Vous devriez avoir un résultat similaire à l'image suivante. N'oubliez pas de recompiler les shader !



Le vert représente les coordonnées horizontales et le rouge les coordonnées verticales. Le coin noir et le coin jaune confirment que les coordonnées sont interpolées correctement de (0, 0) à (1, 1) sur la surface du carré. L'utilisation des couleurs pour visualiser les valeurs est l'équivalent du débogage à l'aide de `printf` pour les shaders. Il n'y a pas vraiment d'autre option !

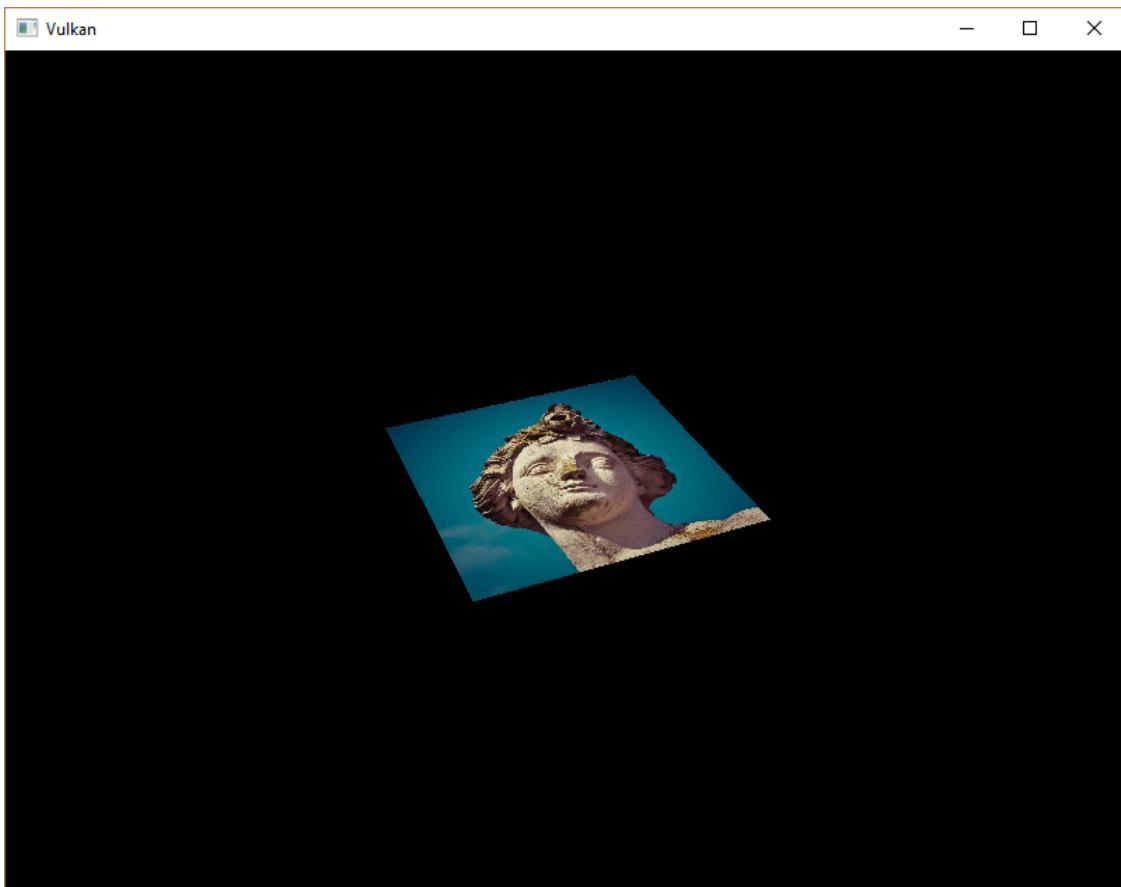
Un descripteur d'échantillonneur combiné d'image correspond à une variable uniforme de type échantillonneur (sampler) en GLSL. Ajoutez-y une référence dans le fragment shader :

```
layout(binding = 1) uniform sampler2D texSampler;
```

Il existe des échantillonneurs 1D (`sampler1D`) et 3D (`sampler3D`) pour les autres types d'images. Aussi, assurez-vous d'utiliser la bonne liaison.

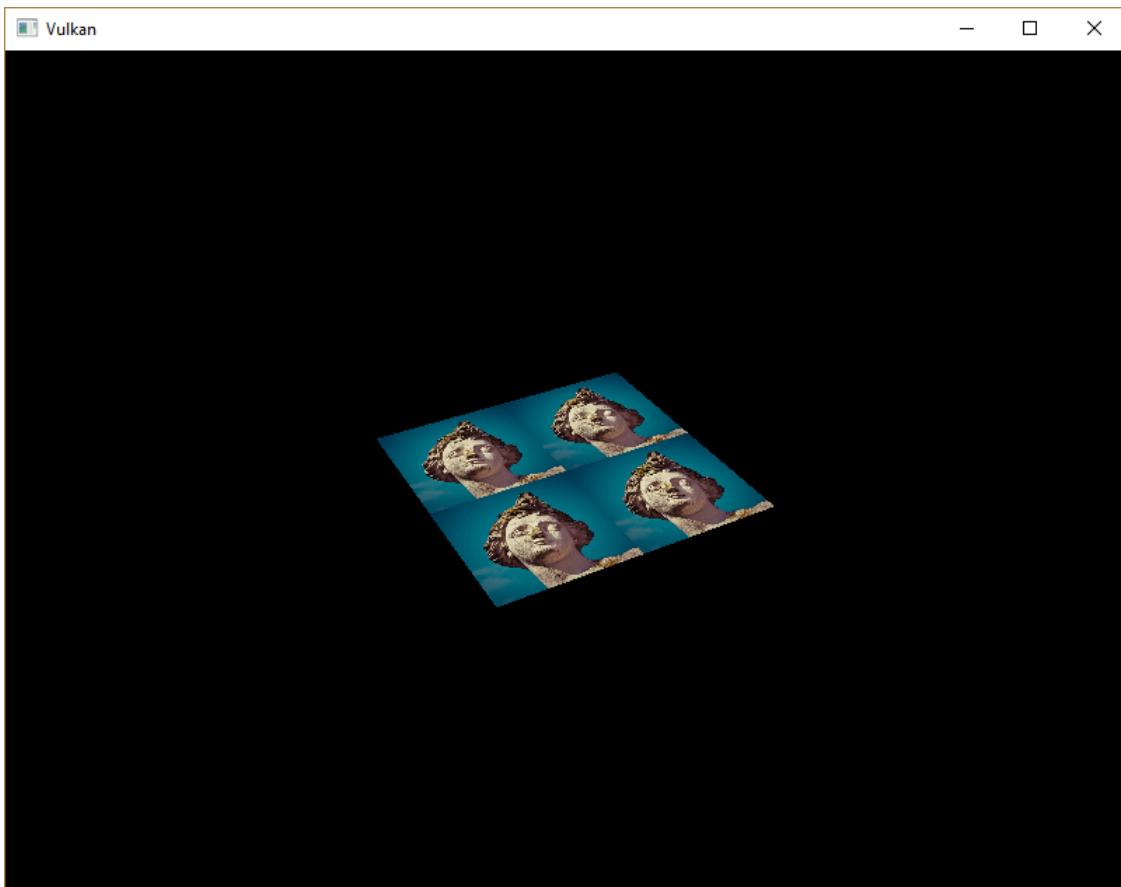
```
void main() {
    outColor = texture(texSampler, fragTexCoord);
}
```

Les textures sont échantillonnées à l'aide de la fonction `texture()` fournie par le GLSL. La fonction `texture()` prend en argument un objet `sampler` et des coordonnées. L'échantillonneur prend en charge automatiquement le filtrage et les transformations. Vous devriez maintenant voir la texture sur le carré :



Expérez avec les modes d'adressage en modifiant les valeurs des coordonnées de texture. Par exemple, le fragment shader suivant produit l'image ci-dessous avec le mode d'adressage `VK_SAMPLER_ADDRESS_MODE_REPEAT` ;

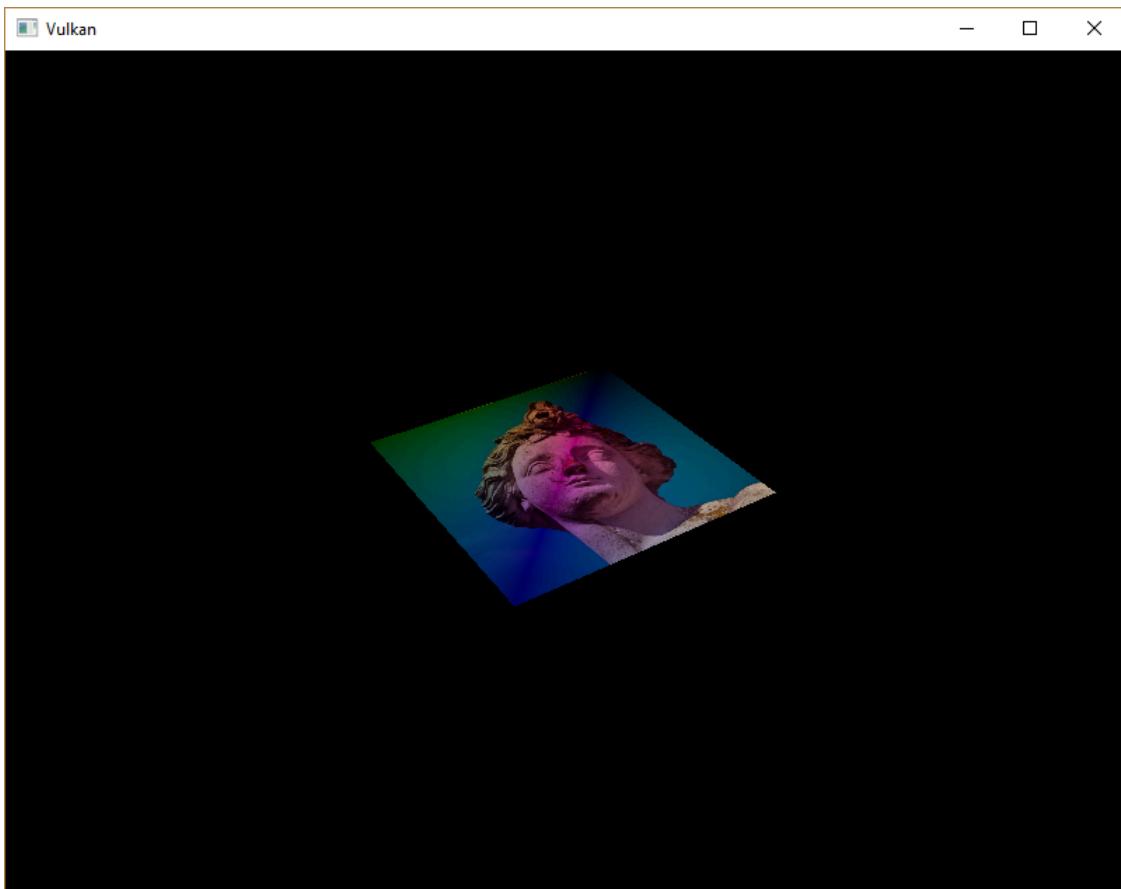
```
void main() {
    outColor = texture(texSampler, fragTexCoord * 2.0);
}
```



Vous pouvez aussi combiner les couleurs de la texture avec celles des sommets :

```
void main() {
    outColor = vec4(fragColor * texture(texSampler, fragTexCoord).rgb, 1.0);
}
```

J'ai séparé l'alpha du reste pour ne pas altérer le canal alpha.



Nous savons désormais comment accéder aux images à partir de nos shaders ! C'est une technique très puissante, lorsqu'associée à des images provenant du rendu d'un tampon d'image. Vous pouvez utiliser les images comme entrées pour implémenter des effets sympas comme des effets de post-traitement ou pour afficher le retour d'une caméra dans votre monde 3D.

Code C++ / Vertex shader / Fragment shader

VIII - Tampon de profondeurs

VIII-A - Introduction

Le modèle que nous affichons est certes projeté dans un monde 3D, mais il est complètement plat. Dans ce chapitre, nous allons ajouter une coordonnée Z aux positions afin de pouvoir afficher des modèles 3D. Nous allons utiliser cette troisième coordonnée pour placer un carré au-dessus du carré déjà existant et voir apparaître un problème lorsque les modèles ne sont pas triés selon leur profondeur.

VIII-B - Géométrie en 3D

Mettez à jour la structure `Vertex` pour utiliser des vecteurs 3D pour la position. Modifiez aussi le format dans la structure `VkVertexInputAttributeDescription` correspondant aux coordonnées :

```
struct Vertex {
    glm::vec3 pos;
    glm::vec3 color;
    glm::vec2 texCoord;

    ...
}
```

```

static std::array<VkVertexInputAttributeDescription, 3> getAttributeDescriptions() {
    std::array<VkVertexInputAttributeDescription, 3> attributeDescriptions{};

    attributeDescriptions[0].binding = 0;
    attributeDescriptions[0].location = 0;
    attributeDescriptions[0].format = VK_FORMAT_R32G32B32_SFLOAT;
    attributeDescriptions[0].offset = offsetof(Vertex, pos);

    ...
}

};

```

Modifiez le vertex shader pour accepter comme entrée des coordonnées 3D. N'oubliez pas de le recompiler après modification !

```

layout(location = 0) in vec3 inPosition;

...

void main() {
    gl_Position = ubo.proj * ubo.view * ubo.model * vec4(inPosition, 1.0);
    fragColor = inColor;
    fragTexCoord = inTexCoord;
}

```

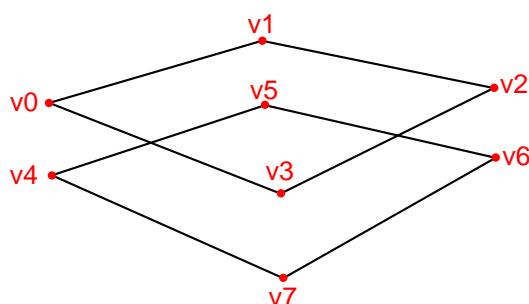
Finalement, ajoutez la profondeur dans les données des sommets :

```

const std::vector<Vertex> vertices = {
    {{-0.5f, -0.5f, 0.0f}, {1.0f, 0.0f, 0.0f}, {0.0f, 0.0f}},
    {{0.5f, -0.5f, 0.0f}, {0.0f, 1.0f, 0.0f}, {1.0f, 0.0f}},
    {{0.5f, 0.5f, 0.0f}, {0.0f, 0.0f, 1.0f}, {1.0f, 1.0f}},
    {{-0.5f, 0.5f, 0.0f}, {1.0f, 1.0f, 1.0f}, {0.0f, 1.0f}}
};

```

Si vous lancez l'application, vous verrez exactement le même résultat qu'auparavant. Il est temps d'ajouter une autre géométrie pour rendre la scène plus intéressante et surtout, pour montrer le problème que nous résolvons dans ce chapitre. Dupliquez les sommets pour définir la position d'un second carré qui sera juste au-dessus du carré actuel :



Utilisez la valeur `-0.5f` comme profondeur du second carré :

```

const std::vector<Vertex> vertices = {
    {{-0.5f, -0.5f, 0.0f}, {1.0f, 0.0f, 0.0f}, {0.0f, 0.0f}},
    {{0.5f, -0.5f, 0.0f}, {0.0f, 1.0f, 0.0f}, {1.0f, 0.0f}},
    {{0.5f, 0.5f, 0.0f}, {0.0f, 0.0f, 1.0f}, {1.0f, 1.0f}},
    {{-0.5f, 0.5f, 0.0f}, {1.0f, 1.0f, 1.0f}, {0.0f, 1.0f}},
    {{-0.5f, -0.5f, -0.5f}, {1.0f, 0.0f, 0.0f}, {0.0f, 0.0f}},
    {{0.5f, -0.5f, -0.5f}, {0.0f, 1.0f, 0.0f}, {1.0f, 0.0f}},
    {{0.5f, 0.5f, -0.5f}, {0.0f, 0.0f, 1.0f}, {1.0f, 1.0f}},
    {{-0.5f, 0.5f, -0.5f}, {1.0f, 1.0f, 1.0f}, {0.0f, 1.0f}}
};

```

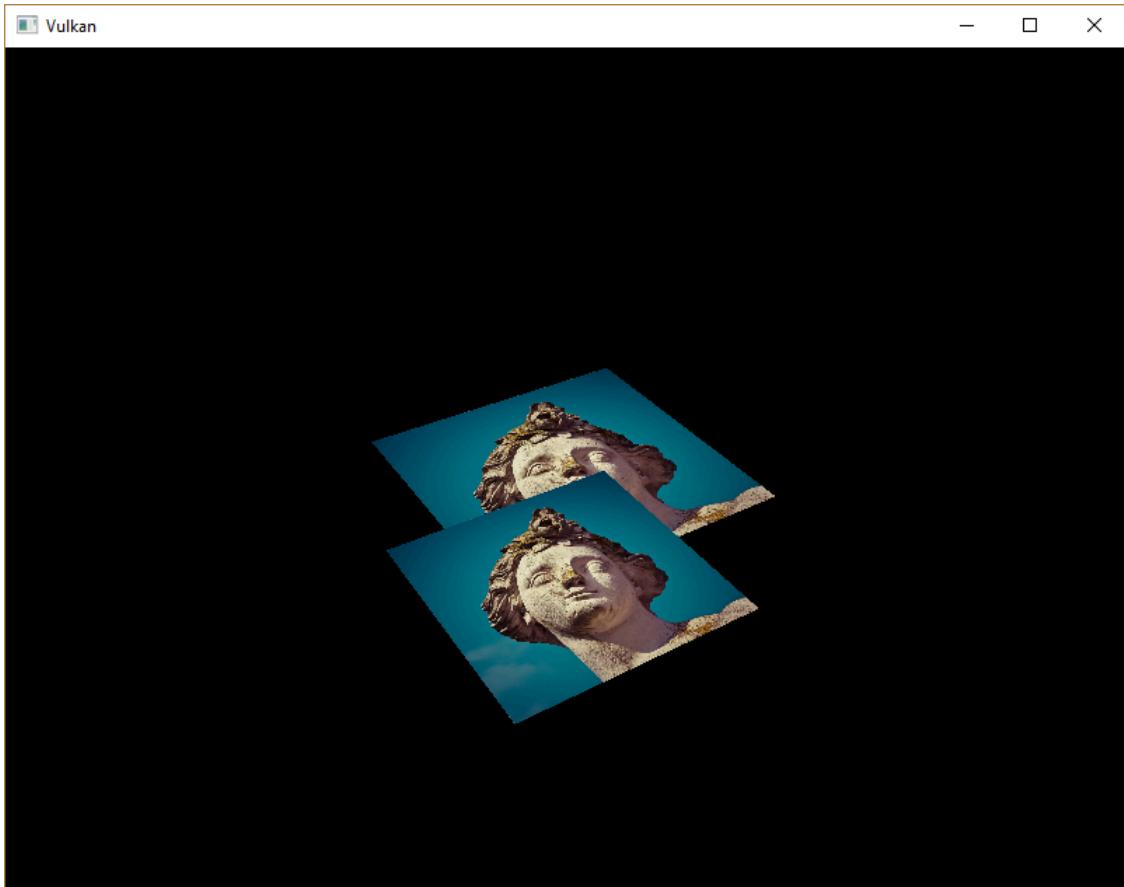
```

        {{0.5f, -0.5f, -0.5f}, {0.0f, 1.0f, 0.0f}, {1.0f, 0.0f}},
        {{0.5f, 0.5f, -0.5f}, {0.0f, 0.0f, 1.0f}, {1.0f, 1.0f}},
        {{-0.5f, 0.5f, -0.5f}, {1.0f, 1.0f, 1.0f}, {0.0f, 1.0f}}
    };

const std::vector<uint16_t> indices = {
    0, 1, 2, 2, 3, 0,
    4, 5, 6, 6, 7, 4
};

```

Si vous lancez le programme maintenant, vous obtiendrez quelque chose rappelant une illustration de **Maurits Cornelis Escher** :



Le problème apparaît, car les fragments du carré inférieur sont écrits par-dessus les valeurs du carré supérieur. Cet ordre dépend de l'ordre de rendu des carrés. Il y a deux manières de régler ce problème :

- trier nos géométries en fonction de la profondeur ;
- utiliser un tampon de profondeur.

La première approche est communément utilisée pour l'affichage d'objets transparents, car la transparence indépendante de l'ordre d'affichage est un problème difficile à résoudre. Toutefois, la problématique du tri des géométries est réglée en utilisant un tampon de profondeur (depth buffer). Un tampon de profondeur est une attache supplémentaire conservant la profondeur de chaque position, à l'instar d'une attache de couleurs qui sauvegarde la couleur pour chaque position. Chaque fois que le rastériseur produit un fragment, sa profondeur est lue pour savoir si le nouveau fragment se situe devant ou derrière le fragment déjà en place. S'il est plus loin, le fragment est ignoré. Si le fragment est plus près et qu'il réussit donc le test de profondeur, sa valeur est alors inscrite dans le tampon de profondeur. Il est possible de manipuler cette valeur dans le fragment shader, tout comme vous pouvez manipuler la couleur.

```
#define GLM_FORCE_RADIANS
```

```
#define GLM_FORCE_DEPTH_ZERO_TO_ONE
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
```

La matrice de projection générée par GLM utilise par défaut l'échelle de profondeur d'OpenGL comprise en -1.0 et 1.0. Nous devons configurer GLM avec GLM_FORCE_DEPTH_ZERO_TO_ONE pour qu'elle utilise les valeurs correspondantes à Vulkan (soit, sur l'échelle 0.0 à 1.0).

VIII-C - Image de profondeur et vue

L'attache pour la profondeur repose sur une image, tout comme l'attache des couleurs. La différence est que celle-ci n'est pas créée automatiquement par la « swap chain ». Nous n'avons besoin que d'une seule image de profondeur, car il n'y a qu'une seule opération de rendu à la fois. L'image de profondeur nécessite le classique triplet de ressources : image, mémoire et vue d'image.

```
VkImage depthImage;
VkDeviceMemory depthImageMemory;
VkImageView depthImageView;
```

Créez une nouvelle fonction nommée `createDepthResources` pour mettre en place ces ressources :

```
void initVulkan() {
    ...
    createCommandPool();
    createDepthResources();
    createTextureImage();
    ...
}

...

void createDepthResources() {
}
```

La création d'une image de profondeur est assez simple. Elle doit avoir la même résolution que l'attache des couleurs, définie par la zone d'échange de la « swap chain ». Son utilisation doit correspondre à une attache de profondeur, elle doit avoir un tiling optimal et une mémoire provenant du périphérique local. La seule question est : quel est le bon format pour une image de profondeur ? Le format doit contenir des profondeurs, indiquées par `_D??_` dans les valeurs `VK_FORMAT_`.

Contrairement à la texture, nous n'avons pas nécessairement besoin d'un format précis, car nous n'accédons pas aux texels directement. L'image de profondeur doit avoir une précision suffisante : les applications classiques utilisent 24 bits de précision. Plusieurs formats correspondent à ce besoin :

- `VK_FORMAT_D32_SFLOAT` : des nombres flottants sur 32 bits pour la profondeur ;
- `VK_FORMAT_D32_SFLOAT_S8_UINT` : des nombres flottants signés sur 32 bits pour la profondeur et un entier non signé sur 8 bits pour le tampon de pochoir ;
- `VK_FORMAT_D24_UNORM_S8_UINT` : des nombres flottants signés sur 24 bits pour la profondeur et des entiers non signés sur 8 bits pour le tampon de pochoir.

La partie pour le tampon de pochoir est utilisée dans le **test de pochoir** (stencil). C'est un test additionnel qui peut être combiné avec le test de profondeur. Nous y reviendrons dans un prochain chapitre.

Nous pourrions nous contenter d'utiliser le format `VK_FORMAT_D32_SFLOAT`, car son support est pratiquement assuré (consultez la base de données des matériels), mais c'est toujours un plus d'écrire une application flexible. Dans ce but, créez une fonction nommée `findSupportedFormat`. Elle recevra une liste des formats candidats, du plus intéressant au moins intéressant et vérifiera quel est le premier format supporté par le matériel :

```
VkFormat findSupportedFormat(const std::vector<VkFormat>& candidates, VkImageTiling tiling,
    VkFormatFeatureFlags features) {
}
```

Le support de tel ou tel format dépend du mode de tiling et de l'utilisation. C'est pourquoi nous les passons en paramètres à la fonction. Le support d'un format donné peut être obtenu grâce à la fonction **vkGetPhysicalDeviceFormatProperties()** :

```
for (VkFormat format : candidates) {
    VkFormatProperties props;
    vkGetPhysicalDeviceFormatProperties(physicalDevice, format, &props);
}
```

La structure **VkFormatProperties** contient trois champs :

- `linearTilingFeatures` : cas d'utilisations supportés avec le tiling linéaire ;
- `optimalTilingFeatures` : cas d'utilisations supportés avec le tiling optimal ;
- `bufferFeatures` : cas d'utilisations supportés avec les tampons.

Seuls les deux premiers cas nous intéressent ici. Nous vérifierons l'un ou l'autre suivant le mode de tiling fourni en paramètre :

```
if (tiling == VK_IMAGE_TILING_LINEAR && (props.linearTilingFeatures & features) == features) {
    return format;
} else if (tiling == VK_IMAGE_TILING_OPTIMAL && (props.optimalTilingFeatures & features) == features) {
    return format;
}
```

Si aucun des candidats ne supporte l'utilisation souhaitée, nous pouvons soit retourner une valeur particulière, soit lever une exception :

```
VkFormat findSupportedFormat(const std::vector<VkFormat>& candidates, VkImageTiling tiling,
    VkFormatFeatureFlags features) {
    for (VkFormat format : candidates) {
        VkFormatProperties props;
        vkGetPhysicalDeviceFormatProperties(physicalDevice, format, &props);

        if (tiling == VK_IMAGE_TILING_LINEAR && (props.linearTilingFeatures & features) == features) {
            return format;
        } else if (tiling == VK_IMAGE_TILING_OPTIMAL && (props.optimalTilingFeatures & features) == features) {
            return format;
        }
    }

    throw std::runtime_error("Échec pour trouver un format supporté !");
}
```

Nous allons utiliser cette fonction depuis la fonction `findDepthFormat()` pour sélectionner un format avec une composante pour la profondeur qui supporte une utilisation dans une attache de profondeur :

```
VkFormat findDepthFormat() {
    return findSupportedFormat(
        {VK_FORMAT_D32_SFLOAT, VK_FORMAT_D32_SFLOAT_S8_UINT, VK_FORMAT_D24_UNORM_S8_UINT},
        VK_IMAGE_TILING_OPTIMAL,
        VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT
    );
}
```

Assurez-vous d'utiliser un indicateur `VK_FORMAT_FEATURE` et non `VK_IMAGE_USAGE`. Tous les candidats contiennent une composante pour la profondeur, mais les deux derniers contiennent aussi une composante pour le pochoir. Pour le moment, nous ne l'utilisons pas, mais nous devons prendre cet aspect en compte lors des transitions d'agencement des images ayant de tels formats. Ajoutez une simple fonction qui nous indique si le format choisi pour la profondeur contient une composante de pochoir :

```
bool hasStencilComponent(VkFormat format) {
    return format == VK_FORMAT_D32_SFLOAT_S8_UINT || format == VK_FORMAT_D24_UNORM_S8_UINT;
}
```

Appelez cette fonction depuis la fonction `createDepthResources()` pour déterminer le format de profondeur :

```
VkFormat depthFormat = findDepthFormat();
```

Nous avons maintenant toutes les informations nécessaires pour appeler nos fonctions `createImage()` et `createImageView()` :

```
createImage(swapChainExtent.width, swapChainExtent.height, depthFormat, VK_IMAGE_TILING_OPTIMAL,
VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT, VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT, depthImage,
depthImageMemory);
depthImageView = createImageView(depthImage, depthFormat);
```

Cependant, la fonction `createImageView()` suppose que la sous-ressource (subresource) est toujours `VK_IMAGE_ASPECT_COLOR_BIT`. Ce champ devient donc un paramètre :

```
VkImageView createImageView(VkImage image, VkFormat format, VkImageAspectFlags aspectFlags) {
    ...
    viewInfo.subresourceRange.aspectMask = aspectFlags;
    ...
}
```

Modifiez tous les appels à cette fonction pour passer la bonne valeur au nouveau paramètre :

```
swapChainImageViews[i] = createImageView(swapChainImages[i], swapChainImageFormat,
VK_IMAGE_ASPECT_COLOR_BIT);
...
depthImageView = createImageView(depthImage, depthFormat, VK_IMAGE_ASPECT_DEPTH_BIT);
...
textureImageView = createImageView(textureImage, VK_FORMAT_R8G8B8A8_SRGB,
VK_IMAGE_ASPECT_COLOR_BIT);
```

Voilà, pour la création de l'image de profondeur. Nous n'avons pas besoin d'y envoyer de données ou d'y copier une image, car nous allons l'initialiser au début de la passe de rendu, comme nous le faisons pour l'attache des couleurs.

VIII-C-1 - Transition explicite de l'image de profondeur

Nous n'avons pas besoin d'effectuer une transition de l'agencement de l'image vers une attache de profondeur, car nous allons le faire dans la passe de rendu. Toutefois et pour des fins de complétude, je vais décrire le processus dans cette section. Vous pouvez la passer, si vous le souhaitez.

Appelez la fonction `transitionImageLayout()` à la fin de la fonction `createDepthResources()` :

```
transitionImageLayout(depthImage, depthFormat, VK_IMAGE_LAYOUT_UNDEFINED,
VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL);
```

L'agencement indéfini peut être utilisé comme agencement initial, car il n'y a pas de contenu important dans l'image de profondeur. Nous devons mettre à jour la logique de la fonction `transitionImageLayout()` pour utiliser le bon aspect de sous-ressource ;

```

if (newLayout == VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL) {
    barrier.subresourceRange.aspectMask = VK_IMAGE_ASPECT_DEPTH_BIT;

    if (hasStencilComponent(format)) {
        barrier.subresourceRange.aspectMask |= VK_IMAGE_ASPECT_STENCIL_BIT;
    }
} else {
    barrier.subresourceRange.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
}

```

Même si nous n'utilisons pas le composant de pochoir, nous devons l'inclure dans les transitions d'agencement de l'image de profondeur.

Enfin, ajoutez les bons masques d'accès et les étapes du pipeline :

```

if (oldLayout == VK_IMAGE_LAYOUT_UNDEFINED && newLayout ==
VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL) {
    barrier.srcAccessMask = 0;
    barrier.dstAccessMask = VK_ACCESS_TRANSFER_WRITE_BIT;

    sourceStage = VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT;
    destinationStage = VK_PIPELINE_STAGE_TRANSFER_BIT;
} else if (oldLayout == VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL && newLayout ==
VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL) {
    barrier.srcAccessMask = VK_ACCESS_TRANSFER_WRITE_BIT;
    barrier.dstAccessMask = VK_ACCESS_SHADER_READ_BIT;

    sourceStage = VK_PIPELINE_STAGE_TRANSFER_BIT;
    destinationStage = VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT;
} else if (oldLayout == VK_IMAGE_LAYOUT_UNDEFINED && newLayout ==
VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL) {
    barrier.srcAccessMask = 0;
    barrier.dstAccessMask = VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_READ_BIT |
VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT;

    sourceStage = VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT;
    destinationStage = VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT;
} else {
    throw std::invalid_argument("Transition d'agencement non supportée !");
}

```

Le tampon de profondeur sera lu lors du test de profondeur pour savoir si un fragment est visible et sera écrit lors du rendu d'un nouveau fragment. La lecture se produit à l'étape `VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT` et l'écriture à l'étape `VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT`. Vous devez choisir la première des étapes correspondant à l'opération indiquée, afin que tout soit prêt lors de l'utilisation de l'attache de profondeur.

VIII-D - Passe de rendu

Nous allons modifier la fonction `createRenderPass()` pour inclure une attache de profondeur. Définissez d'abord la description grâce à la structure **VkAttachmentDescription** :

```

VkAttachmentDescription depthAttachment{};
depthAttachment.format = findDepthFormat();
depthAttachment.samples = VK_SAMPLE_COUNT_1_BIT;
depthAttachment.loadOp = VK_ATTACHMENT_LOAD_OP_CLEAR;
depthAttachment.storeOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
depthAttachment.stencilLoadOp = VK_ATTACHMENT_LOAD_OP_DONT_CARE;
depthAttachment.stencilStoreOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
depthAttachment.initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
depthAttachment.finalLayout = VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL;

```

Le format doit être celui de l'image de profondeur. Cette fois, nous ne garderons pas les données de profondeur (`storeOp`), car nous n'en avons plus besoin après le rendu. Cela pourrait permettre au matériel d'effectuer des

optimisations. Tout comme pour le tampon de couleurs, nous n'avons pas besoin des valeurs du rendu précédent. Nous pouvons donc utiliser `VK_IMAGE_LAYOUT_UNDEFINED` comme valeur pour `initialLayout`.

```
VkAttachmentReference depthAttachmentRef{};

depthAttachmentRef.attachment = 1;
depthAttachmentRef.layout = VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL;
```

Ajoutez une référence à l'attache dans notre première (et unique) sous-passe :

```
VkSubpassDescription subpass{};

subpass.pipelineBindPoint = VK_PIPELINE_BIND_POINT_GRAPHICS;
subpass.colorAttachmentCount = 1;
subpass.pColorAttachments = &colorAttachmentRef;
subpass.pDepthStencilAttachment = &depthAttachmentRef;
```

Contrairement aux attaches de couleurs, une sous-passe ne peut avoir qu'une attache de profondeur (et de pochoir). En effet, aucun intérêt de réaliser plusieurs tests de profondeur sur plusieurs tampons.

```
std::array<VkAttachmentDescription, 2> attachments = {colorAttachment, depthAttachment};

VkRenderPassCreateInfo renderPassInfo{};
renderPassInfo.sType = VK_STRUCTURE_TYPE_RENDER_PASS_CREATE_INFO;
renderPassInfo.attachmentCount = static_cast<uint32_t>(attachments.size());
renderPassInfo.pAttachments = attachments.data();
renderPassInfo.subpassCount = 1;
renderPassInfo.pSubpasses = &subpass;
renderPassInfo.dependencyCount = 1;
renderPassInfo.pDependencies = &dependency;
```

Finalement, mettez à jour la structure `VkRenderPassCreateInfo` pour référencer ces deux attaches.

VIII-E - Tampon d'images

L'étape suivante va consister à modifier la création du tampon d'images pour lier notre image de profondeur à l'attache de profondeur. Dans la fonction `createFramebuffers()`, indiquez la vue de l'image de profondeur comme deuxième attache :

```
std::array<VkImageView, 2> attachments = {
    swapChainImageViews[i],
    depthImageView
};

VkFramebufferCreateInfo framebufferInfo{};
framebufferInfo.sType = VK_STRUCTURE_TYPE_FRAMEBUFFER_CREATE_INFO;
framebufferInfo.renderPass = renderPass;
framebufferInfo.attachmentCount = static_cast<uint32_t>(attachments.size());
framebufferInfo.pAttachments = attachments.data();
framebufferInfo.width = swapChainExtent.width;
framebufferInfo.height = swapChainExtent.height;
framebufferInfo.layers = 1;
```

L'attache de couleurs change pour chaque image de la « swap chain », mais la même attache de profondeur peut être utilisée pour toutes les images, car une seule sous-passe s'exécute à la fois grâce à nos sémaphores.

Nous devons également déplacer l'appel à la fonction `createFramebuffers()` pour que la fonction soit appelée après la création de l'image de profondeur :

```
void initVulkan() {
    ...
    createDepthResources();
    createFramebuffers();
    ...
}
```

VIII-F - Valeurs de nettoyage des tampons

Comme nous avons plusieurs attaches avec `VK_ATTACHMENT_LOAD_OP_CLEAR`, nous devons spécifier plusieurs valeurs de nettoyage. Allez à la fonction `createCommandBuffers()` et créez un tableau de la structure `VkClearValue` :

```
std::array<VkClearValue, 2> clearValues{};  
clearValues[0].color = {0.0f, 0.0f, 0.0f, 1.0f};  
clearValues[1].depthStencil = {1.0f, 0};  
  
renderPassInfo.clearValueCount = static_cast<uint32_t>(clearValues.size());  
renderPassInfo.pClearValues = clearValues.data();
```

Avec Vulkan, l'échelle des valeurs des profondeurs dans le tampon est de 0.0 à 1.0, où 1.0 indique le plan lointain et 0.0, le plan proche. La valeur initiale de chaque point dans le tampon doit être au plus loin possible, donc 1.0.

Notez que l'ordre des valeurs de `clearValues` doit correspondre à l'ordre de vos attaches.

VIII-G - Configuration des tests profondeur et de pochoir

L'attache de profondeur est prête à être utilisée, mais le test de profondeur n'a pas encore été activé dans le pipeline. Il se configure grâce à la structure de type `VkPipelineDepthStencilStateCreateInfo` :

```
VkPipelineDepthStencilStateCreateInfo depthStencil{};  
depthStencil.sType = VK_STRUCTURE_TYPE_PIPELINE_DEPTH_STENCIL_STATE_CREATE_INFO;  
depthStencil.depthTestEnable = VK_TRUE;  
depthStencil.depthWriteEnable = VK_TRUE;
```

Le champ `depthTestEnable` indique si la profondeur des nouveaux fragments doit être comparée avec la profondeur présente dans le tampon afin de savoir si les fragments vont être ignorés. Le champ `depthWriteEnable` indique si la nouvelle profondeur des fragments ayant réussi le test doit être écrite dans le tampon de profondeur. C'est utile pour l'affichage d'objets transparents. En effet, les fragments d'un objet transparent doivent passer le test, mais ne doivent pas empêcher les objets derrière d'être dessinés.

```
depthStencil.depthCompareOp = VK_COMPARE_OP_LESS;
```

Le champ `depthCompareOp` fournit le test de comparaison utilisé pour conserver ou éliminer les fragments. Comme nous sommes dans un cas classique où une petite valeur de profondeur indique un objet proche, la profondeur des nouveaux fragments doit être inférieure.

```
depthStencil.depthBoundsTestEnable = VK_FALSE;  
depthStencil.minDepthBounds = 0.0f; // Optionnel  
depthStencil.maxDepthBounds = 1.0f; // Optionnel
```

Les champs `depthBoundsTestEnable`, `minDepthBounds` et `maxDepthBounds` sont utilisés pour des tests de profondeur encadrés. Ils permettent de ne garder que des fragments dont la profondeur est comprise entre deux valeurs fournies ici. Nous n'utiliserons pas cette fonctionnalité.

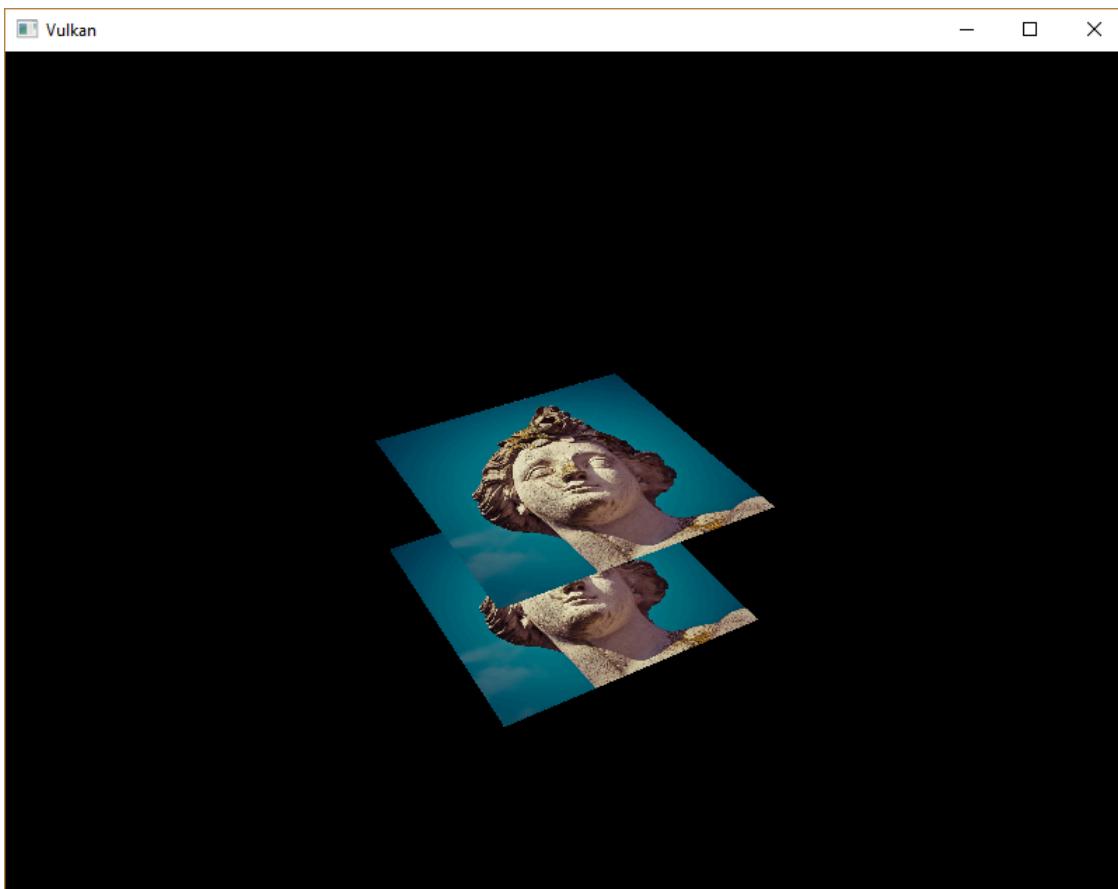
```
depthStencil.stencilTestEnable = VK_FALSE;  
depthStencil.front{}; // Optionnel  
depthStencil.back{}; // Optionnel
```

Les trois derniers champs configurent les opérations liées au tampon de pochoir, que nous n'utiliserons pas non plus dans ce tutoriel. Si vous voulez l'utiliser, vous devrez vous assurer que le format sélectionné pour la profondeur contient aussi une composante pour le pochoir.

```
1. pipelineInfo.pDepthStencilState = &depthStencil;
```

Mettez à jour la structure **VkGraphicsPipelineCreateInfo** pour référencer l'état des tests de profondeur et de pochoir que nous venons de créer. Un tel état doit toujours être spécifié si la passe de rendu contient au moins une attache de profondeur.

Si vous lancez le programme, vous verrez que les géométries apparaissent correctement :



VIII-H - Gestion des redimensionnements de la fenêtre

La résolution du tampon de profondeur doit changer avec la taille de la fenêtre quand elle est redimensionnée, afin de correspondre à la taille de l'attache des couleurs. Étendez la fonction `recreateSwapChain()` pour régénérer les ressources liées à la profondeur :

```
void recreateSwapChain() {
    int width = 0, height = 0;
    while (width == 0 || height == 0) {
        glfwGetFramebufferSize(window, &width, &height);
        glfwWaitEvents();
    }

    vkDeviceWaitIdle(device);

    cleanupSwapChain();

    createSwapChain();
    createImageViews();
    createRenderPass();
    createGraphicsPipeline();
    createDepthResources();
    createFramebuffers();
    createUniformBuffers();
    createDescriptorPool();
    createDescriptorSets();
```

```
    createCommandBuffers();  
}
```

La libération des ressources doit avoir lieu dans la fonction de libération de la « swap chain » :

```
void cleanupSwapChain() {  
    vkDestroyImageView(device, depthImageView, nullptr);  
    vkDestroyImage(device, depthImage, nullptr);  
    vkFreeMemory(device, depthImageMemory, nullptr);  
  
    ...  
}
```

Félicitations, votre application est maintenant capable d'afficher correctement n'importe quelle géométrie 3D. Nous allons essayer dans le prochain chapitre en affichant un modèle texturé !

Code C++ / Vertex shader / Fragment shader

IX - Chargement de modèles

IX-A - Introduction

Votre programme peut maintenant afficher des modèles 3D texturés, mais l'objet défini par les sommets des tableaux **vertices et indices** n'est pas ce qu'il y a de plus intéressant. Dans ce chapitre, nous allons modifier le programme pour charger les sommets et les indices depuis un fichier afin de réellement exploiter la carte graphique.

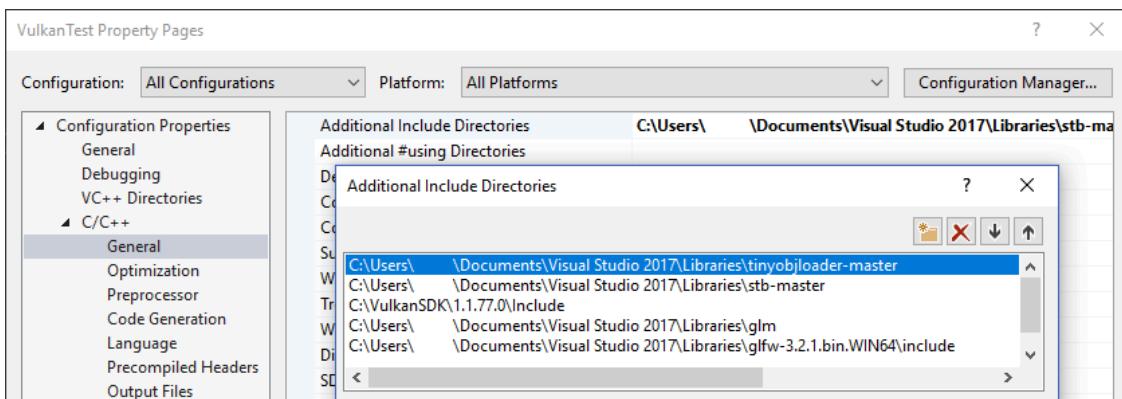
Beaucoup de tutoriels sur les bibliothèques graphiques décrivent l'implémentation pour charger les modèles au format OBJ. Le problème est que n'importe quelle application 3D intéressante nécessite des fonctionnalités non supportées par ce format, notamment les animations de squelette. Nous allons, nous aussi, charger un modèle au format OBJ, mais nous nous concentrerons sur l'intégration des données du modèle dans le programme plutôt que sur les détails liés au chargement du fichier.

IX-B - Une bibliothèque

Nous utiliserons la bibliothèque **tinyobjloader** pour charger les sommets et les faces depuis un fichier OBJ. C'est rapide et facile à intégrer, car, comme pour la bibliothèque stb_image, il n'y a qu'un fichier à ajouter au projet. Suivez le lien ci-dessus et téléchargez le fichier **tiny_obj_loader.h**. Placez-le dans votre dossier dédié aux bibliothèques. Assurez-vous d'utiliser la version du fichier provenant de la branche master, car la dernière version publiée n'est pas à jour.

IX-B-1 - Visual Studio

Ajoutez dans la section « Autres répertoires Include » (Additional Include Directories) le dossier dans lequel est contenu **tiny_obj_loader.h**.



IX-B-2 - Makefile

Ajoutez le dossier contenant `tiny_obj_loader.h` aux dossiers d'inclusions de GCC :

```
VULKAN_SDK_PATH = /home/user/VulkanSDK/x.x.x.x/x86_64
STB_INCLUDE_PATH = /home/user/libraries/stb
TINYOBJ_INCLUDE_PATH = /home/user/libraries/tinyobjloader

...
CFLAGS = -std=c++17 -I$(VULKAN_SDK_PATH)/include -I$(STB_INCLUDE_PATH) -I$(TINYOBJ_INCLUDE_PATH)
```

IX-C - Modèle d'exemple

Dans ce chapitre, nous n'activons pas l'éclairage. Il est donc préférable de charger un modèle intégrant les effets de lumière dans la texture. Il est facile de trouver de tels modèles en explorant la section des scans 3D de [Sketchfab](#). Vous pouvez y trouver de nombreux modèles au format OBJ ayant une licence permissive.

Pour ce tutoriel, j'ai choisi d'utiliser la [maison viking](#) créée par [nigelgoh \(CC BY 4.0\)](#). J'ai modifié la taille et l'orientation pour l'utiliser comme remplacement de notre objet actuel :

- [modèle](#)
- [texture](#)

N'hésitez pas à utiliser votre propre modèle. Simplement, assurez-vous qu'il ne comprend qu'un seul matériau et que ses dimensions sont de 1.5 x 1. x 1.5 unités. S'il est plus grand, vous devrez changer la matrice de vue. Placez le modèle dans un dossier appelé `models` et l'image dans le dossier `textures`.

Ajoutez deux variables de configuration pour indiquer l'emplacement du modèle et de la texture :

```
const uint32_t WIDTH = 800;
const uint32_t HEIGHT = 600;

const std::string MODEL_PATH = "models/viking_room.obj";
const std::string TEXTURE_PATH = "textures/viking_room.png";
```

Modifiez la fonction `createTextureImage()` pour charger la texture du modèle :

```
stbi_uc* pixels = stbi_load(TEXTURE_PATH.c_str(), &texWidth, &texHeight, &texChannels,
STBI_rgb_alpha);
```

IX-D - Chargement des sommets et des indices

Nous allons charger les sommets et les indices depuis le fichier du modèle. Remplacez les tableaux `vertices` et `indices` par des vecteurs dynamiques membres de notre classe :

```
std::vector<Vertex> vertices;
std::vector<uint32_t> indices;
VkBuffer vertexBuffer;
VkDeviceMemory vertexBufferMemory;
```

Il faut aussi changer le type des indices afin d'utiliser les `uint32_t`, car nous allons dépasser les 65 535 sommets. Changez également le paramètre de type dans l'appel à la fonction `vkCmdBindIndexBuffer()` :

```
vkCmdBindIndexBuffer(commandBuffers[i], indexBuffer, 0, VK_INDEX_TYPE_UINT32);
```

La bibliothèque `tinyobjloader` s'inclut de la même façon que toutes les bibliothèques STB. Assurez-vous de définir la macro `TINYOBJLOADER_IMPLEMENTATION` afin d'avoir la définition des fonctions (et ainsi, éviter des erreurs lors de l'édition des liens) :

```
#define TINYOBJLOADER_IMPLEMENTATION
#include <tiny_obj_loader.h>
```

Nous allons ensuite écrire une fonction nommée `loadModel()` pour remplir le tableau de sommets et d'indices avec les données du modèle. Nous devons l'appeler avant la création des tampons de sommets et d'indices :

```
void initVulkan() {
    ...
    loadModel();
    createVertexBuffer();
    createIndexBuffer();
    ...
}

...

void loadModel() {
```

Le chargement du modèle s'effectue avec la fonction `tinyobj::LoadObj()` :

```
void loadModel() {
    tinyobj::attrib_t attrib;
    std::vector<tinyobj::shape_t> shapes;
    std::vector<tinyobj::material_t> materials;
    std::string warn, err;

    if (!tinyobj::LoadObj(&attrib, &shapes, &materials, &warn, &err, MODEL_PATH.c_str())) {
        throw std::runtime_error(warn + err);
    }
}
```

Un fichier OBJ contient des positions, des normales, des coordonnées de textures et des faces. Ces dernières consistent en un nombre arbitraire de sommets, dont la position, la normale et/ou la coordonnée de texture sont référencées par un index. Chaque sommet est constitué d'une position, une normale et/ou une coordonnée de texture. Ainsi, il est possible de réutiliser des attributs spécifiques et non l'intégralité d'un sommet.

Le conteneur `attrib` contient les positions, les normales et les coordonnées de texture dans les propriétés `attrib.vertices`, `attrib.normals` et `attrib.texcoords`. Le conteneur `shapes` contient tous les objets séparément et leurs faces. Ces dernières contiennent un tableau de sommets, où chaque sommet contient les indices pour la position,

pour la normale et pour la coordonnée de texture. Les modèles OBJ peuvent aussi définir un matériel et une texture par face, mais nous ignorons cette particularité.

La chaîne de caractères `err` contient les erreurs et la chaîne `warn` contient les messages d'avertissemens liés au chargement du fichier. Notamment, ces variables peuvent indiquer l'absence de la définition d'un matériel. Le chargement a échoué si la fonction `LoadObj()` retourne `false`. Comme indiqué précédemment, les faces dans les fichiers OBJ peuvent contenir un nombre arbitraire de sommets alors que l'application n'est capable que de dessiner des triangles. Heureusement, la fonction `LoadObj()` possède un paramètre optionnel pour déterminer les triangles à partir des faces. Cette option est activée par défaut.

Nous allons combiner toutes les faces du fichier en un seul modèle. Il n'y a donc qu'à parcourir les formes :

```
for (const auto& shape : shapes) {
}
```

Grâce à la triangularisation, nous sommes sûrs que les faces n'ont que trois sommets. Nous pouvons donc simplement parcourir les sommets et les copier directement dans notre tableau `vertices` :

```
for (const auto& shape : shapes) {
    for (const auto& index : shape.mesh.indices) {
        Vertex vertex = {};

        vertices.push_back(vertex);
        indices.push_back(indices.size());
    }
}
```

Pour faire simple, nous allons partir du principe que tous les sommets sont uniques. Ainsi, nous pouvons définir les indices par un simple auto-incrémentation. La variable `index` est du type `tinyobj::index_t`, et contient les propriétés `vertex_index`, `normal_index` et `texcoord_index`. Nous devons utiliser ces indices pour trouver les attributs du sommet à utiliser se trouvant dans les tableaux `attrib` :

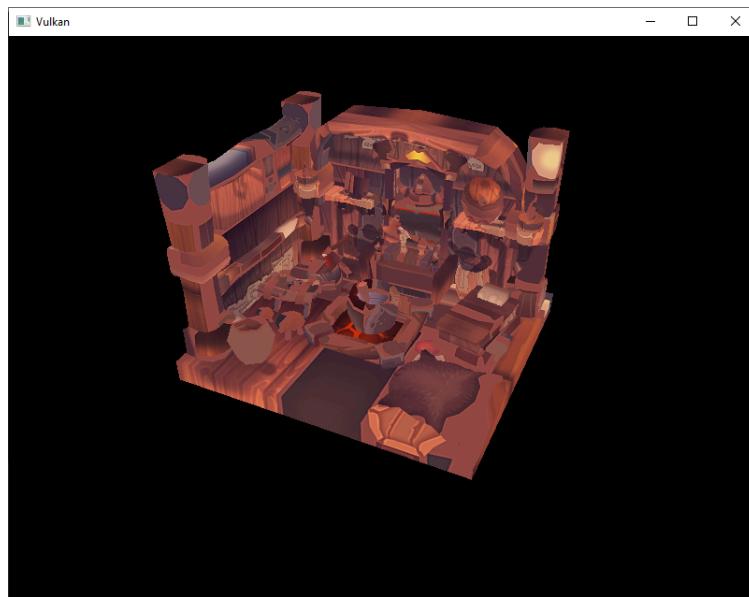
```
vertex.pos = {
    attrib.vertices[3 * index.vertex_index + 0],
    attrib.vertices[3 * index.vertex_index + 1],
    attrib.vertices[3 * index.vertex_index + 2]
};

vertex.texCoord = {
    attrib.texcoords[2 * index.texcoord_index + 0],
    attrib.texcoords[2 * index.texcoord_index + 1]
};

vertex.color = {1.0f, 1.0f, 1.0f};
```

Le tableau `attrib.vertices` contient des valeurs flottantes et non pas un type similaire à `glm::vec3`. Il faut donc multiplier les indices par 3. De même, il y a deux composants par élément pour les coordonnées de texture. Les décalages 0, 1 et 2 permettent d'accéder aux composants X, Y et Z, ou aux composants U et V dans le cas des textures.

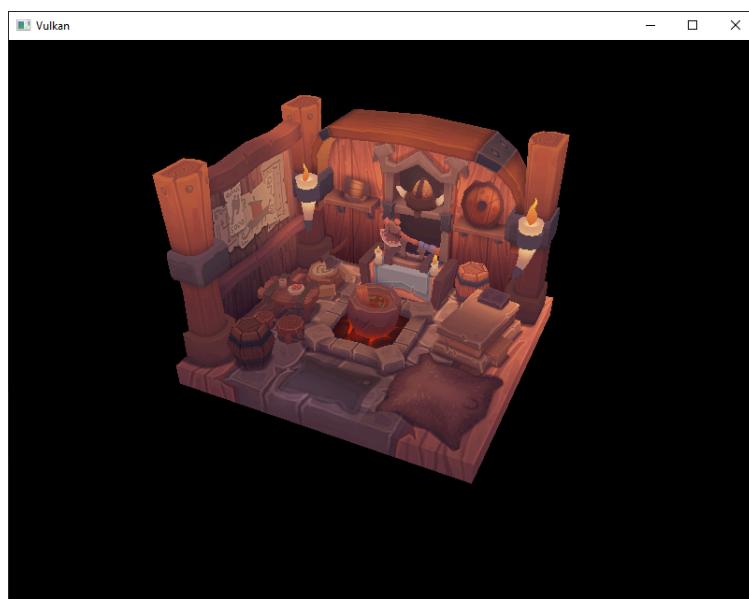
Lancez le programme en activant les optimisations (compilation `Release` avec Visual Studio ou avec l'option `-O3` pour GCC). Vous êtes obligé de faire ainsi, sans quoi le chargement du modèle sera très lent. Vous devriez obtenir le résultat suivant :



Génial, la géométrie semble correcte ! Par contre, que se passe-t-il avec les textures ? Le format OBJ contient des coordonnées de texture où la coordonnée 0 est placée en bas de l'image. Dans notre cas, nous avons envoyé à Vulkan l'image où la coordonnée 0 indique le haut de l'image. Il suffit d'inverser la composante verticale des coordonnées de texture pour régler le problème :

```
vertex.texCoord = {
    attrib.texcoords[2 * index.texcoord_index + 0],
    1.0f - attrib.texcoords[2 * index.texcoord_index + 1]
};
```

Vous pouvez lancer à nouveau le programme. Le rendu devrait maintenant être correct :



Notre long travail commence enfin à porter ses fruits !

IX-E - Déduplication des sommets

Malheureusement, nous ne profitons pas du tampon d'indices. Le tableau vertices contient énormément de sommets dupliqués, car beaucoup d'entre eux sont utilisés dans plusieurs triangles. Nous ne devrions inclure que des sommets

uniques et utiliser le tampon d'indice pour les réutiliser lorsque possible. Une approche simple est d'implémenter une map ou une `unordered_map` pour garder une trace des sommets uniques et de leur indice :

```
#include <unordered_map>

...

std::unordered_map<Vertex, uint32_t> uniqueVertices{};

for (const auto& shape : shapes) {
    for (const auto& index : shape.mesh.indices) {
        Vertex vertex{};

        ...

        if (uniqueVertices.count(vertex) == 0) {
            uniqueVertices[vertex] = static_cast<uint32_t>(vertices.size());
            vertices.push_back(vertex);
        }

        indices.push_back(uniqueVertices[vertex]);
    }
}
```

Chaque fois que l'on extrait un sommet du fichier OBJ, nous devons vérifier si nous avons déjà rencontré un sommet possédant exactement la même position et la même coordonnée de texture. Si ce n'est pas le cas, nous l'ajoutons dans `vertices` et nous stockons son index dans `uniqueVertices`. Ensuite, nous ajoutons l'indice au nouveau tableau `indices`. Si le sommet est connu, il suffit de récupérer son indice à partir de `uniqueVertices` et de le stocker dans `indices`.

Pour l'instant, le programme ne peut pas compiler, car nous utilisons notre structure `Vertex` comme clé de la table de hachage. Dans un tel cas, nous devons implémenter deux fonctions : un test d'égalité et une fonction de hachage. Le test d'égalité est facile à implémenter en surchargeant l'opérateur `==` de la structure :

```
bool operator==(const Vertex& other) const {
    return pos == other.pos && color == other.color && texCoord == other.texCoord;
}
```

Nous devons spécialiser le template `std::hash<T>` pour obtenir un hachage pour la structure `Vertex`. L'écriture d'une fonction de hachage est compliquée, mais [cppreference.com recommande](#) l'approche suivante pour obtenir une fonction de bonne qualité : combiner le hachage des champs de la structure.

```
namespace std {
    template<> struct hash<Vertex> {
        size_t operator()(Vertex const& vertex) const {
            return ((hash<glm::vec3>()(vertex.pos) ^
                    (hash<glm::vec3>()(vertex.color) << 1) >> 1) ^
                    (hash<glm::vec2>()(vertex.texCoord) << 1);
        }
    };
}
```

Ce code doit être placé hors de la définition de `Vertex`. Les fonctions de hachage des types provenant de GLM s'activent avec la définition et l'inclusion suivantes :

```
#define GLM_ENABLE_EXPERIMENTAL
#include <glm/gtx/hash.hpp>
```

Les fonctions de hachage sont définies dans le dossier `gtx` et proviennent donc d'une extension expérimentale de GLM. C'est pourquoi vous devez définir `GLM_ENABLE_EXPERIMENTAL` pour les utiliser. Cela signifie que la bibliothèque peut être modifiée dans les prochaines versions de GLM, mais en pratique, la bibliothèque est très stable.

Vous devriez maintenant pouvoir compiler et lancer le programme. Si vous vérifiez la taille de vertices, vous verrez qu'elle est passée de 1 500 000 à 265 645 éléments ! Cela signifie que les sommets sont réutilisés dans six triangles différents en moyenne. Cela permet d'économiser beaucoup de mémoire GPU.

Code C++ / Vertex shader / Fragment shader

X - Génération de mipmaps

X-A - Introduction

Notre programme peut maintenant charger et afficher des modèles 3D. Dans ce chapitre, nous allons ajouter une fonctionnalité : la génération des mipmaps. Les mipmaps sont largement utilisées dans les jeux et les logiciels de rendu et Vulkan nous donne un contrôle complet sur leur création.

Les mipmaps sont des réductions précalculées de vos images. Chaque nouvelle image a une taille correspondant à la moitié de la taille de l'image précédente. Les mipmaps sont utilisées comme technique de niveau de détail (Level of Detail (LOD)) : les objets au loin utiliseront des textures avec des images mipmaps de plus petite taille. En utilisant des images réduites, la vitesse de rendu sera améliorée et les artefacts tels que le **moiré** seront réduits. Voici un exemple de mipmaps :



X-B - Création des images

Avec Vulkan, chaque niveau de mipmap est stocké dans les différents *niveaux de mipmap* de la ressource de type **VkImage**. Le niveau 0 correspond à l'image originale et les niveaux suivants sont appelés chaînes de mip (mip chain).

Le nombre de niveaux de mipmap doit être indiqué lors de la création de la ressource de type **VkImage**. Jusqu'à présent, nous définissons cette valeur à 1. Nous devons maintenant calculer le nombre de niveaux à partir des dimensions de l'image. D'abord, ajoutez une variable membre pour stocker ce nombre :

```
...
```

```
uint32_t mipLevels;
VkImage textureImage;
...
```

La valeur de la variable `mipLevels` ne peut être déterminée qu'une fois la texture chargée par la fonction `createTextureImage()` :

```
int texWidth, texHeight, texChannels;
stbi_uc* pixels = stbi_load(TEXTURE_PATH.c_str(), &texWidth, &texHeight, &texChannels,
    STBI_rgb_alpha);
...
mipLevels = static_cast<uint32_t>(std::floor(std::log2(std::max(texWidth, texHeight)))) + 1;
```

Ce code permet de calculer le nombre de niveaux dans notre chaîne de mip. La fonction `max()` récupère la plus grande dimension de l'image. La fonction `log2()` détermine combien de fois cette dimension peut être divisée par 2. La fonction `floor()` gère le cas où la dimension utilisée n'est pas un multiple de deux. On ajoute 1 afin de prendre en compte le niveau de l'image d'origine.

Pour utiliser cette valeur, nous devons modifier les fonctions `createImage()`, `createImageView()` et `transitionImageLayout()` afin d'y spécifier le nombre de niveaux de mip. Ajoutez un paramètre `mipLevels` à ces fonctions :

```
void createImage(uint32_t width, uint32_t height, uint32_t mipLevels, VkFormat format,
    VkImageTiling tiling, VkImageUsageFlags usage, VkMemoryPropertyFlags properties, VkImage& image,
    VkDeviceMemory& imageMemory) {
    ...
    imageInfo.mipLevels = mipLevels;
    ...
}
```

```
VkImageView createImageView(VkImage image, VkFormat format, VkImageAspectFlags aspectFlags,
    uint32_t mipLevels) {
    ...
    viewInfo.subresourceRange.levelCount = mipLevels;
    ...
}
```

```
void transitionImageLayout(VkImage image, VkFormat format, VkImageLayout oldLayout, VkImageLayout
    newLayout, uint32_t mipLevels) {
    ...
    barrier.subresourceRange.levelCount = mipLevels;
    ...
}
```

Nous devons aussi mettre à jour ces appels pour utiliser les bonnes valeurs :

```
createImage(swapChainExtent.width, swapChainExtent.height, 1, depthFormat,
    VK_IMAGE_TILING_OPTIMAL, VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT,
    VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT, depthImage, depthImageMemory);
...
createImage(texWidth, texHeight, mipLevels, VK_FORMAT_R8G8B8A8_SRGB,
    VK_IMAGE_TILING_OPTIMAL, VK_IMAGE_USAGE_TRANSFER_DST_BIT | VK_IMAGE_USAGE_SAMPLED_BIT,
    VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT, textureImage, textureImageMemory);
```

```
swapChainImageViews[i] = createImageView(swapChainImages[i], swapChainImageFormat,
    VK_IMAGE_ASPECT_COLOR_BIT, 1);
...
depthImageView = createImageView(depthImage, depthFormat, VK_IMAGE_ASPECT_DEPTH_BIT, 1);
...
textureImageView = createImageView(textureImage, VK_FORMAT_R8G8B8A8_SRGB,
    VK_IMAGE_ASPECT_COLOR_BIT, mipLevels);
```

```
transitionImageLayout(depthImage, depthFormat, VK_IMAGE_LAYOUT_UNDEFINED,
    VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL, 1);
...
```

```
transitionImageLayout(textureImage, VK_FORMAT_R8G8B8A8_SRGB, VK_IMAGE_LAYOUT_UNDEFINED,
VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL, mipLevels);
```

X-C - Génération des mipsmaps

Notre texture a maintenant plusieurs niveaux de mipsmaps, mais le tampon intermédiaire ne peut être utilisé que pour remplir le niveau 0. Les données des autres niveaux ne sont toujours pas définies. Pour les remplir, nous devons générer les données des mipsmaps à partir du seul niveau que nous avons. Nous allons utiliser la commande **vkCmdBlitImage**. Cette commande effectue une copie tout en permettant le redimensionnement et des opérations de filtrages. Nous pouvons l'appeler plusieurs fois pour copier les données de chaque niveau dans notre texture.

La commande **vkCmdBlitImage** est considérée comme une opération de transfert. Nous devons donc indiquer à Vulkan que nous souhaitons utiliser la texture aussi bien comme source que comme destination d'un transfert. Ajoutez le bit **VK_IMAGE_USAGE_TRANSFER_SRC_BIT** aux utilisations souhaitées lors de la création de l'image.

```
...
createImage(texWidth, texHeight, mipLevels, VK_FORMAT_R8G8B8A8_SRGB, VK_IMAGE_TILING_OPTIMAL,
VK_IMAGE_USAGE_TRANSFER_SRC_BIT | VK_IMAGE_USAGE_TRANSFER_DST_BIT | VK_IMAGE_USAGE_SAMPLED_BIT,
VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT, textureImage, textureImageMemory);
...
```

Comme pour les autres opérations sur les images, la commande **vkCmdBlitImage** dépend de l'agencement de l'image sur laquelle elle opère. Nous pourrions effectuer une transition de l'intégralité de l'image vers **VK_IMAGE_LAYOUT_GENERAL**, mais cela serait certainement lent. Pour obtenir des performances optimales, l'image source doit être dans l'agencement **VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL** et l'image de destination doit être avec l'agencement **VK_IMAGE_LAYOUT_DST_OPTIMAL**. Vulkan nous permet de faire une transition de chaque niveau de mip indépendamment. Chaque copie ne travaillera que sur deux niveaux de mip à la fois, donc nous pouvons effectuer une transition entre chaque opération pour toujours avoir l'agencement optimal.

La fonction `transitionImageLayout()` n'effectue des transitions d'agencement que sur l'intégralité de l'image. Nous devons donc ajouter des barrières de pipeline. Supprimez la transition existante vers **VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL** de la fonction `createTextureImage()` :

```
...
transitionImageLayout(textureImage, VK_FORMAT_R8G8B8A8_SRGB, VK_IMAGE_LAYOUT_UNDEFINED,
VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL, mipLevels);
copyBufferToImage(stagingBuffer,
textureImage, static_cast<uint32_t>(texWidth), static_cast<uint32_t>(texHeight));
//transitioned to VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL while generating mipsmaps
...
```

Une fois cela fait, tous les niveaux de la texture sont dans l'agencement **VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL**. Chaque niveau sera ensuite transitionné vers **VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL** après que la commande de copie a fini de lire dans l'image.

Nous allons maintenant écrire la fonction pour générer les mipsmaps :

```
void generateMipsmaps(VkImage image, int32_t texWidth, int32_t texHeight, uint32_t mipLevels) {
VkCommandBuffer commandBuffer = beginSingleTimeCommands();

VkImageMemoryBarrier barrier{};
barrier.sType = VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER;
barrier.image = image;
barrier.srcQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
barrier.dstQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
barrier.subresourceRange.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
barrier.subresourceRange.baseArrayLayer = 0;
barrier.subresourceRange.layerCount = 1;
barrier.subresourceRange.levelCount = 1;
```

```

        endSingleTimeCommands(commandBuffer);
    }
}

```

Nous allons réaliser plusieurs transitions et donc réutiliser la structure **VkImageMemoryBarrier**. Les champs remplis ci-dessus seront valides pour toutes les barrières. Les champs subresourceRange.mipLevel, oldLayout, newLayout, srcAccessMask et dstAccessMask seront modifiés à chaque transition

```

int32_t mipWidth = texWidth;
int32_t mipHeight = texHeight;

for (uint32_t i = 1; i < mipLevels; i++) {
}

```

Cette boucle enregistre les commandes **VkCmdBlitImage**. Notez que la boucle commence à 1, et pas à 0.

```

barrier.subresourceRange.baseMipLevel = i - 1;
barrier.oldLayout = VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL;
barrier.newLayout = VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL;
barrier.srcAccessMask = VK_ACCESS_TRANSFER_WRITE_BIT;
barrier.dstAccessMask = VK_ACCESS_TRANSFER_READ_BIT;

vkCmdPipelineBarrier(commandBuffer,
    VK_PIPELINE_STAGE_TRANSFER_BIT, VK_PIPELINE_STAGE_TRANSFER_BIT, 0,
    0, nullptr,
    0, nullptr,
    1, &barrier);

```

Tout d'abord, nous effectuons une transition du niveau `i - 1` vers l'agencement `VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL`. Cette transition attendra le remplissage du niveau `i - 1`, que ce soit par la commande précédente ou par la fonction **vkCmdCopyBufferToImage()**. La commande de copie actuelle attendra l'exécution de la transition.

```

VkImageBlit blit{};
blit.srcOffsets[0] = { 0, 0, 0 };
blit.srcOffsets[1] = { mipWidth, mipHeight, 1 };
blit.srcSubresource.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
blit.srcSubresource.mipLevel = i - 1;
blit.srcSubresource.baseArrayLayer = 0;
blit.srcSubresource.layerCount = 1;
blit.dstOffsets[0] = { 0, 0, 0 };
blit.dstOffsets[1] = { mipWidth > 1 ? mipWidth / 2 : 1, mipHeight > 1 ? mipHeight / 2 : 1, 1 };
blit.dstSubresource.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
blit.dstSubresource.mipLevel = i;
blit.dstSubresource.baseArrayLayer = 0;
blit.dstSubresource.layerCount = 1;

```

Nous devons maintenant indiquer les régions concernées par la commande de copie. Le niveau de mip source est `i - 1` et le niveau de destination est `i`. Les deux éléments du tableau `srcOffsets` déterminent la région 3D à partir de laquelle les données seront copiées. Le tableau `dstOffsets`, définit la région recevant les données de la copie. Les dimensions X et Y de `dstOffsets[1]` sont divisées par 2, car chaque niveau de mip est deux fois moins grand que le niveau précédemment. La dimension Z de `srcOffsets[1]` et `dstOffsets[1]` doit être 1, car notre image 2D a une profondeur de 1.

```

vkCmdBlitImage(commandBuffer,
    image, VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL,
    image, VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL,
    1, &blit,
    VK_FILTER_LINEAR);

```

Nous enregistrons la commande de copie. Notez que la variable `textureImage` est utilisée aussi bien comme source que comme destination. C'est que nous voulons copier différents niveaux de mip de la même image. Le niveau de mip source vient de passer à l'agencement `VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL` et le niveau

destination est encore avec l'agencement `VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL` obtenu grâce à la fonction `createTextureImage()`.

⚠️ Lorsque vous utilisez une queue dédiée au transfert (comme suggéré dans ce chapitre), la commande `vkCmdBlitImage` doit être enregistrée dans une queue graphique.

Le dernier paramètre permet de fournir un filtre de type `VkFilter` à utiliser lors de la copie. Nous avons accès aux mêmes options de filtrage qu'avec la structure `VkSampler`. Nous utilisons la valeur `VK_FILTER_LINEAR`, pour activer l'interpolation linéaire.

```
barrier.oldLayout = VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL;
barrier.newLayout = VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL;
barrier.srcAccessMask = VK_ACCESS_TRANSFER_READ_BIT;
barrier.dstAccessMask = VK_ACCESS_SHADER_READ_BIT;

vkCmdPipelineBarrier(commandBuffer,
    VK_PIPELINE_STAGE_TRANSFER_BIT, VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT, 0,
    0, nullptr,
    0, nullptr,
    1, &barrier);
```

Ce code attend que le niveau $i - 1$ passe à l'agencement `VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL`. Cette transition attend que la copie finisse. Toutes les opérations d'échantillonnage vont attendre que la transition se termine.

```
...
if (mipWidth > 1) mipWidth /= 2;
if (mipHeight > 1) mipHeight /= 2;
}
```

À la fin de la boucle, nous divisons les dimensions du niveau de mip actuel par 2. Nous vérifions les dimensions avant d'effectuer la division pour s'assurer que la dimension ne soit jamais 0. Ainsi, nous gérons le cas où les images ne sont pas carrées pour lesquelles une des deux dimensions serait 1 avant l'autre. Lorsque cela se produit, la dimension est toujours 1 pour les niveaux restants.

```
barrier.subresourceRange.baseMipLevel = mipLevels - 1;
barrier.oldLayout = VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL;
barrier.newLayout = VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL;
barrier.srcAccessMask = VK_ACCESS_TRANSFER_WRITE_BIT;
barrier.dstAccessMask = VK_ACCESS_SHADER_READ_BIT;

vkCmdPipelineBarrier(commandBuffer,
    VK_PIPELINE_STAGE_TRANSFER_BIT, VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT, 0,
    0, nullptr,
    0, nullptr,
    1, &barrier);

endSingleTimeCommands(commandBuffer);
}
```

Avant de terminer avec le tampon de commandes, nous devons insérer une dernière barrière. Cette barrière permet d'effectuer la transition du dernier niveau de mip de l'agencement `VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL` vers `VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL`. Ce cas n'est pas pris en compte par la boucle, car nous ne copions jamais le dernier niveau de mip.

Appelez la fonction `generateMipmaps()` dans la fonction `createTextureImage()` :

```
transitionImageLayout(textureImage, VK_FORMAT_R8G8B8A8_SRGB, VK_IMAGE_LAYOUT_UNDEFINED,
    VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL, mipLevels);
```

```

        copyBufferToImage(stagingBuffer,
        textureImage, static_cast<uint32_t>(texWidth), static_cast<uint32_t>(texHeight));
//transition vers l'agencement VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL lors de la génération des
mipmaps
...
generateMipmaps(textureImage, texWidth, texHeight, mipLevels);
    
```

Les mipmaps de notre image sont maintenant complètement remplies.

X-D - Support du filtrage linéaire

Il est très pratique d'utiliser une fonction telle que **vkCmdBlitImage()** pour générer tous les niveaux de mip, mais il n'est malheureusement pas garanti qu'elle soit supportée sur toutes les plateformes. La fonction repose sur le support du filtrage linéaire pour le format d'image utilisé. Le support de cette fonctionnalité peut être vérifié avec la fonction **vkGetPhysicalDeviceFormatProperties()**. Nous effectuons cette vérification dans la fonction `generateMipmaps()`.

Ajoutez d'abord un paramètre qui indique le format de l'image :

```

void createTextureImage() {
    ...

    generateMipmaps(textureImage, VK_FORMAT_R8G8B8A8_SRGB, texWidth, texHeight, mipLevels);
}

void generateMipmaps(VkImage image, VkFormat imageFormat, int32_t texWidth, int32_t texHeight,
                     uint32_t mipLevels) {
    ...
}
    
```

Dans la fonction `generateMipmaps()`, utilisez la fonction **vkGetPhysicalDeviceFormatProperties()** pour récupérer les propriétés du format de la texture :

```

void generateMipmaps(VkImage image, VkFormat imageFormat, int32_t texWidth, int32_t texHeight,
                     uint32_t mipLevels) {

    // Vérifie si le format supporte le filtrage linéaire
    VkFormatProperties formatProperties;
    vkGetPhysicalDeviceFormatProperties(physicalDevice, imageFormat, &formatProperties);

    ...
}
    
```

La structure **VkFormatProperties** possède les trois champs : `linearTilingFeatures`, `optimalTilingFeature` et `bufferFeaetures`. Chacun décrit l'utilisation possible du format suivant la façon dont l'image est utilisée. Nous créons nos textures avec le format de tiling optimal, donc nous devons utiliser le champ `optimalTilingFeatures`. Le support du filtrage linéaire est indiqué par le bit `VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT`.

```

if (!(formatProperties.optimalTilingFeatures &
      VK_FORMAT_FEATURE_SAMPLED_IMAGE_FILTER_LINEAR_BIT)) {
    throw std::runtime_error("Le format de la texture ne supporte pas le filtrage linéaire !");
}
    
```

Si le format de texture ne supporte pas le filtrage linéaire, vous avez deux possibilités :

- implémenter une fonction cherchant un format avec le support du filtrage linéaire dans les opérations de copie ;
- implémenter la génération des mipmaps de manière logicielle, notamment grâce à une bibliothèque telle que **stb_image_resize**. Chaque image peut être chargée dans une image, de la même façon que vous avez chargé l'image originale.

Il est à noter que ce n'est pas une bonne pratique de générer les mipmaps au cours de l'exécution. Habituellement, elles sont pré-générées et stockées dans le fichier de la texture avec le niveau de base afin d'améliorer les vitesses de chargement. L'implémentation logicielle du redimensionnement et le chargement des différents niveaux à partir d'un fichier sont laissés comme exercice pour le lecteur.

X-E - Échantillonneur

Une ressource de type **VkImage** contient les données de mipmap. Une ressource de type **VkSampler** contrôle comment les données sont lues lors du rendu. Vulkan nous fournit les propriétés suivantes : minLod, maxLod, mipLodBias et mipmapMode (où « Lod » signifie « Level of Detail » (niveau de détail)). Pendant l'échantillonnage d'une texture, l'échantillonneur sélectionne le niveau de mip suivant le pseudo-code suivant :

```
lod = getLodLevelFromScreenSize(); // plus petit lorsque l'objet est proche, peut être négatif
lod = clamp(lod + mipLodBias, minLod, maxLod);

level = clamp(floor(lod), 0, texture.mipLevels - 1); // limité par rapport au nombre de niveaux de mip dans la texture

if (mipmapMode == VK_SAMPLER_MIPMAP_MODE_NEAREST) {
    color = sample(level);
} else {
    color = blend(sample(level), sample(level + 1));
}
```

Si la valeur de `samplerInfo.mipmapMode` est `VK_SAMPLER_MIPMAP_MODE_NEAREST`, la variable `lod` sélectionne le niveau de mip à partir duquel échantillonner. Si le mode est `VK_SAMPLER_MIPMAP_MODE_LINEAR`, `lod` sélectionne deux niveaux de mip pour effectuer l'échantillonnage. Le résultat correspond au mélange linéaire des couleurs des deux niveaux.

L'opération d'échantillonnage est aussi affectée par `lod` :

```
if (lod <= 0) {
    color = readTexture(uv, magFilter);
} else {
    color = readTexture(uv, minFilter);
}
```

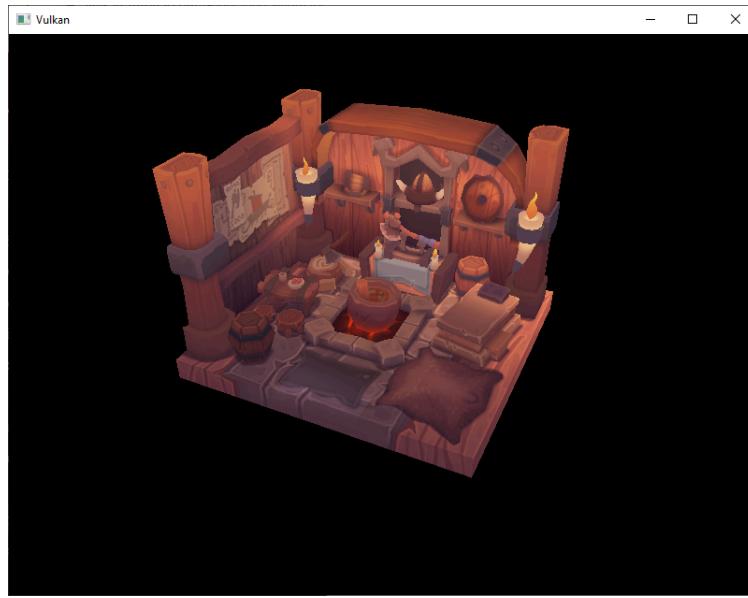
Si l'objet est proche de la caméra, `magFilter` est utilisé comme filtre. Si l'objet est distant, `minFilter` sera utilisé. Normalement, `lod` est un nombre positif et devient nul lorsque l'objet est proche de la caméra. `mipLodBias` permet de forcer Vulkan à utiliser un `lod` particulier et donc, un niveau de mip plus petit que ce qu'il aurait utilisé habituellement.

Pour voir les bénéfices du mipmapping, nous devons définir de nouvelles valeurs pour notre échantillonneur `textureSampler`. Nous avons déjà fourni les valeurs de `minFilter` et `magFilter` pour utiliser le filtrage linéaire `VK_FILTER_LINEAR`. Il nous reste à choisir les valeurs de `minLod`, `maxLod`, `mipLodBias` et `mipmapMode`.

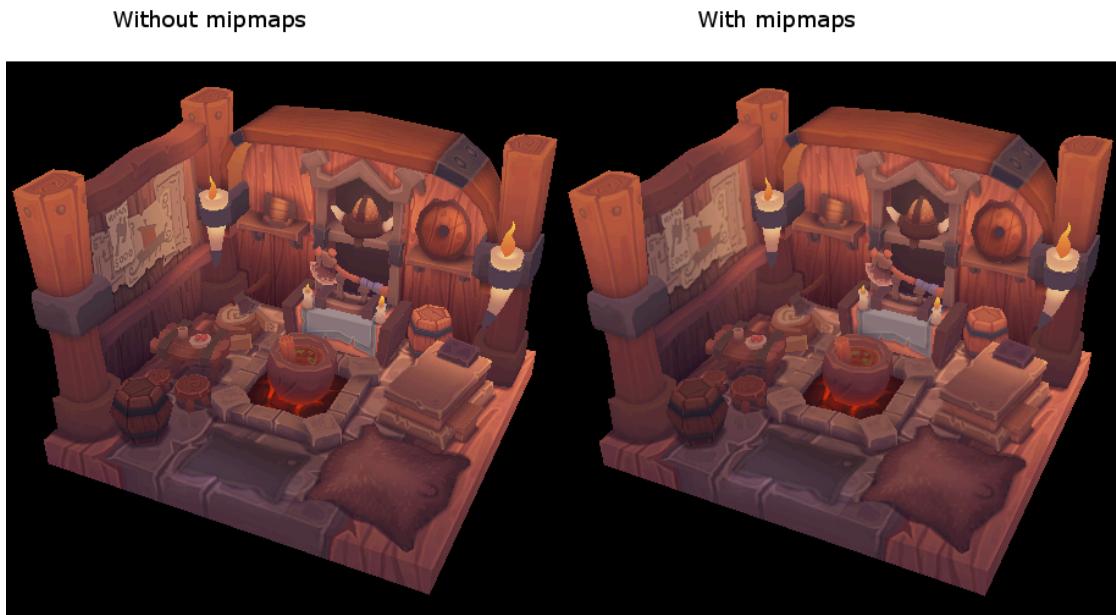
```
void createTextureSampler() {
    ...
    samplerInfo.mipmapMode = VK_SAMPLER_MIPMAP_MODE_LINEAR;
    samplerInfo.minLod = 0.0f; // Optionnel
    samplerInfo.maxLod = static_cast<float>(mipLevels);
    samplerInfo.mipLodBias = 0.0f; // Optionnel
    ...
}
```

Pour utiliser la totalité des niveaux de mipmaps, nous mettons `minLod` à `0.0f` et `maxLod` au nombre de niveaux de mip. Nous n'avons aucune raison de changer la valeur de `lod` avec `mipLodBias`, alors nous pouvons le mettre à `0.0f`.

Lancez votre programme et vous devriez voir ceci :



Notre scène est si simple qu'il n'y a pas de différence majeure. Il y a quelques différences subtiles si vous regardez avec attention :

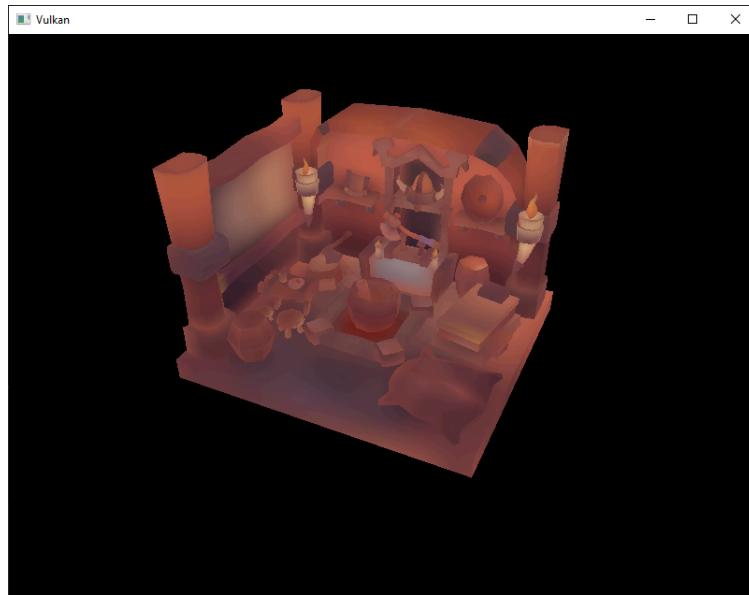


La différence la plus évidente est le texte sur les papiers. Avec les mipmaps, les écritures sont plus lisses. Sans les mipmaps, les écritures ont des bordures dures et des trous, à cause du moiré.

Vous pouvez jouer avec les paramètres de l'échantillonneur et voir leur effet sur le mipmapping. Par exemple, en changeant la valeur de `minLod`, vous pouvez forcer l'échantillonneur à ne pas utiliser les niveaux de mip les plus bas :

```
samplerInfo.minLod = static_cast<float>(mipLevels / 2);
```

Ce qui donne :



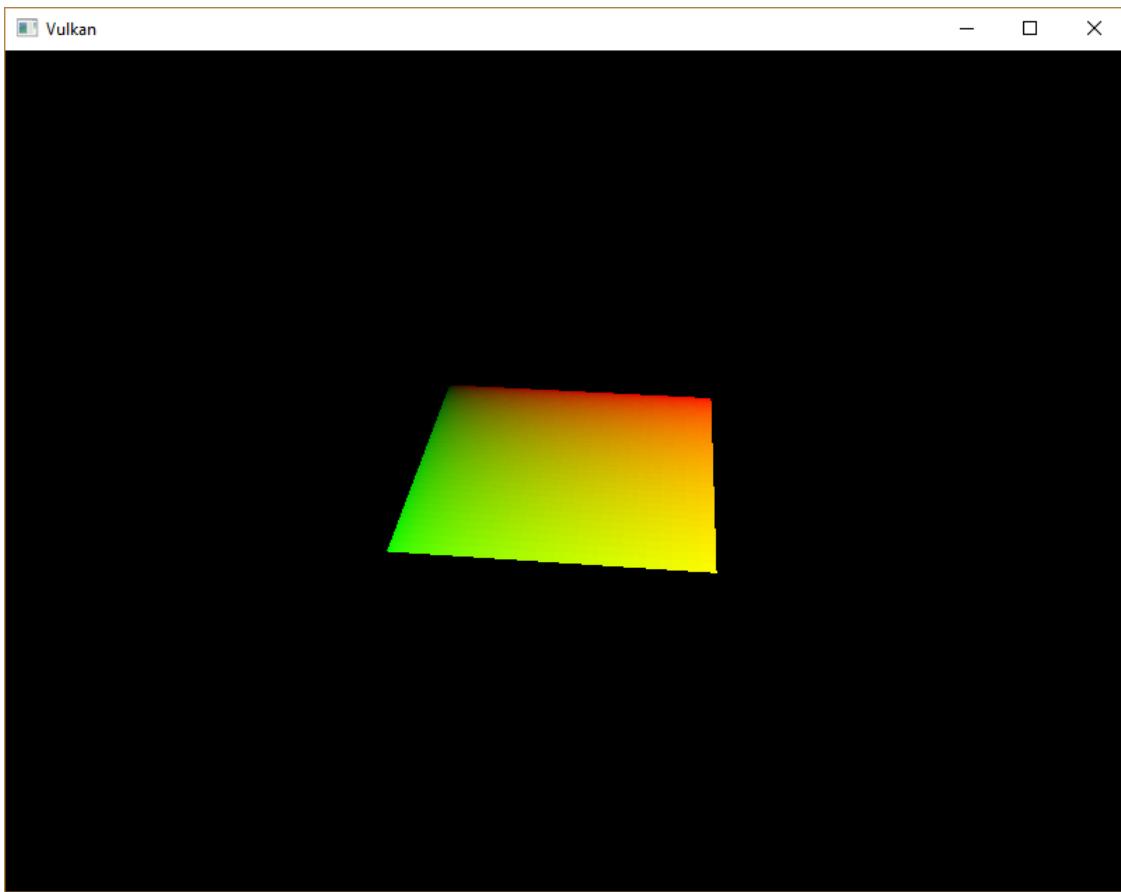
C'est ce que donneront les niveaux de mip les plus hauts qui seront utilisés lorsque les objets sont loin de la caméra.

Code C++ / Vertex shader / Fragment shader

XI - Multiéchantillonnage

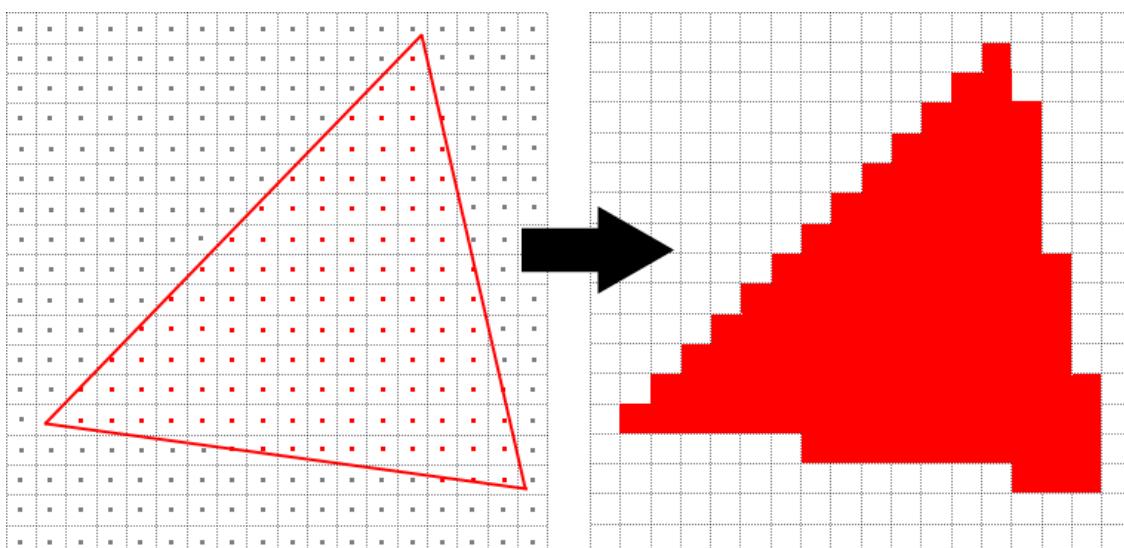
XI-A - Introduction

Notre programme peut maintenant générer plusieurs niveaux de détails pour les textures et ainsi supprimer quelques artefacts lors du rendu d'objets lointains. L'image est plus nette, mais si vous faites attention, vous remarquerez des motifs en dents de scie sur les bordures des objets dessinés. C'est d'autant plus vrai dans nos premiers programmes où nous affichions un carré :



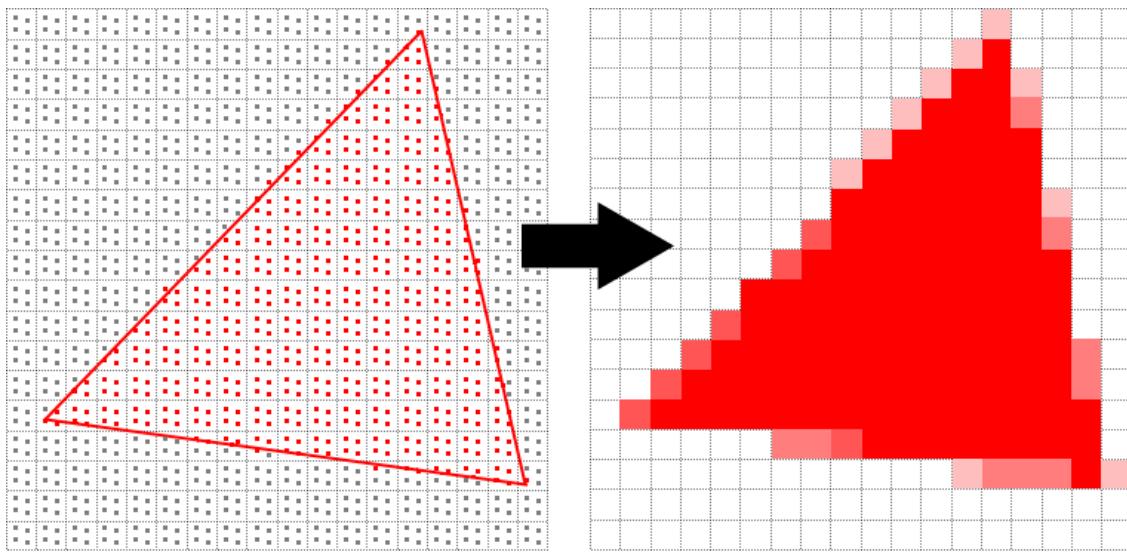
Cet effet indésirable s'appelle crénelage (aliasing). Il est la conséquence du nombre limité de pixels lors du rendu. Comme il n'existe aucun écran ayant une résolution illimitée, il sera toujours visible. Il existe plusieurs méthodes pour corriger cela et ce chapitre se concentrera sur l'une des plus populaires : l'anticrénelage par multiéchantillonnage (**multisample anti-aliasing (MSAA)**).

Dans un rendu classique, la couleur d'un pixel est déterminée à partir d'un unique échantillon, en général le centre du pixel. Si une ligne passe à travers certains pixels sans couvrir le point utilisé par l'échantillon, alors le pixel restera blanc, provoquant cet effet d'escalier.



- Sample point
- Sample point covered by the triangle

Le MSAA consiste à utiliser plusieurs points pour échantillonner un pixel et ainsi déterminer sa couleur. Comme on peut s'y attendre, plus on utilise de points, meilleur est le résultat, mais cela consomme aussi plus de ressources.



Using 4 samples per pixel (MSAAx4)

Pour notre implémentation, nous allons utiliser le maximum de points possible. Selon votre application, cela peut ne pas être la meilleure approche et il peut être judicieux d'utiliser moins d'échantillons afin d'obtenir de meilleures performances.

XI-B - Récupération du nombre maximal d'échantillons

Commençons par déterminer le nombre maximal d'échantillons supporté par la carte graphique. Les cartes graphiques modernes supportent au moins huit échantillons, mais ce nombre n'est pas une norme. Nous allons stocker ce nombre dans une variable membre :

```
...
VkSampleCountFlagBits msaaSamples = VK_SAMPLE_COUNT_1_BIT;
...
```

Par défaut nous n'utilisons qu'un échantillon par pixel, ce qui correspond à ne pas utiliser de multiéchantillonnage. Dans un tel cas, l'image finale ne sera pas impactée. Le nombre d'échantillons maximum supporté par la carte peut être obtenu à partir de la structure de type **VkPhysicalDeviceProperties** associée au périphérique choisi. Nous utilisons aussi un tampon de profondeur. Nous devons donc prendre en compte le nombre d'échantillons pour la couleur et pour la profondeur. Le plus haut nombre d'échantillons supporté par les deux (&) sera le maximum supporté. Ajoutez une fonction pour récupérer cette information :

```
VkSampleCountFlagBits getMaxUsableSampleCount() {
    VkPhysicalDeviceProperties physicalDeviceProperties;
    vkGetPhysicalDeviceProperties(physicalDevice, &physicalDeviceProperties);

    VkSampleCountFlags counts = physicalDeviceProperties.limits.framebufferColorSampleCounts &
    physicalDeviceProperties.limits.framebufferDepthSampleCounts;
    if (counts & VK_SAMPLE_COUNT_64_BIT) { return VK_SAMPLE_COUNT_64_BIT; }
    if (counts & VK_SAMPLE_COUNT_32_BIT) { return VK_SAMPLE_COUNT_32_BIT; }
    if (counts & VK_SAMPLE_COUNT_16_BIT) { return VK_SAMPLE_COUNT_16_BIT; }
    if (counts & VK_SAMPLE_COUNT_8_BIT) { return VK_SAMPLE_COUNT_8_BIT; }
    if (counts & VK_SAMPLE_COUNT_4_BIT) { return VK_SAMPLE_COUNT_4_BIT; }
    if (counts & VK_SAMPLE_COUNT_2_BIT) { return VK_SAMPLE_COUNT_2_BIT; }

    return VK_SAMPLE_COUNT_1_BIT;
}
```

Nous allons utiliser cette fonction pour définir la variable `msaaSamples` pendant le processus de la sélection du GPU. Nous devons modifier la fonction `pickPhysicalDevice()` :

```
void pickPhysicalDevice() {
    ...
    for (const auto& device : devices) {
        if (isDeviceSuitable(device)) {
            physicalDevice = device;
            msaaSamples = getMaxUsableSampleCount();
            break;
        }
    }
    ...
}
```

XI-C - Configurer une cible de rendu

Avec le MSAA, chaque pixel est échantillonné à partir d'un tampon hors écran qui est ensuite affiché à l'écran. Ce nouveau tampon est légèrement différent des images sur lesquelles nous avons dessiné jusqu'à présent : il a la possibilité de stocker plus d'un échantillon par pixel. Une fois le tampon multiéchantillonné créé, il doit être utilisé pour déterminer le tampon d'image classique (qui stocke uniquement un échantillon par pixel). C'est pourquoi nous devons créer une cible de rendu supplémentaire et modifier le processus de rendu. Nous n'avons besoin que d'une cible de rendu, car seule une opération de rendu s'effectue à la fois, tout comme pour le tampon de profondeur. Ajoutez les variables membres suivantes :

```
...
VkImage colorImage;
VkDeviceMemory colorImageMemory;
VkImageView colorImageView;
...
```

Cette nouvelle image doit pouvoir stocker le nombre d'échantillons voulus par pixel. Nous devons donc passer ce nombre à la structure **VkImageCreateInfo** lors de sa création. Modifiez la fonction `createImage()` pour ajouter le paramètre `numSamples` :

```
void createImage(uint32_t width, uint32_t height, uint32_t mipLevels, VkSampleCountFlagBits
    numSamples, VkFormat format, VkImageTiling tiling, VkImageUsageFlags usage,
    VkMemoryPropertyFlags properties, VkImage& image, VkDeviceMemory& imageMemory) {
    ...
    imageInfo.samples = numSamples;
    ...
}
```

Pour le moment, mettez à jour tous les appels à cette fonction avec `VK_SAMPLE_COUNT_1_BIT`. Nous utiliserons la valeur adéquate par la suite.

```
createImage(swapChainExtent.width, swapChainExtent.height, 1, VK_SAMPLE_COUNT_1_BIT,
    depthFormat, VK_IMAGE_TILING_OPTIMAL, VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT,
    VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT, depthImage, depthImageMemory);
...
createImage(texWidth, texHeight, mipLevels, VK_SAMPLE_COUNT_1_BIT, VK_FORMAT_R8G8B8A8_SRGB,
    VK_IMAGE_TILING_OPTIMAL, VK_IMAGE_USAGE_TRANSFER_SRC_BIT | VK_IMAGE_USAGE_TRANSFER_DST_BIT |
    VK_IMAGE_USAGE_SAMPLED_BIT, VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT, textureImage,
    textureImageMemory);
```

Nous allons maintenant créer un tampon de couleurs multiéchantillonné. Ajoutez une fonction nommée `createColorResources` et notez que nous passons la variable `msaaSamples` à la fonction `createImage()`. Nous n'utilisons qu'un seul niveau de mipmap, ce qui est forcé par la spécification de Vulkan pour les images multiéchantillonnées. De plus, ce tampon de couleurs n'a pas besoin de mipmap, car elle n'est pas utilisée comme texture.

```
void createColorResources() {
    VkFormat colorFormat = swapChainImageFormat;
```

```

        createImage(swapChainExtent.width, swapChainExtent.height, 1, msaaSamples,
colorFormat, VK_IMAGE_TILING_OPTIMAL, VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT |
VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT, VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT, colorImage,
colorImageMemory);
        colorImageView = createImageView(colorImage, colorFormat, VK_IMAGE_ASPECT_COLOR_BIT, 1);
}
    
```

Pour une question de cohérence, nous appelons cette fonction juste avant la fonction `createDepthResource()` :

```

void initVulkan() {
    ...
    createColorResources();
    createDepthResources();
    ...
}
    
```

Nous avons maintenant un tampon de couleurs multiéchantillonné. Occupons-nous de la profondeur. Modifiez la fonction `createDepthResources()` et mettez à jour le nombre d'échantillons utilisés :

```

void createDepthResources() {
    ...
    createImage(swapChainExtent.width, swapChainExtent.height, 1, msaaSamples,
depthFormat, VK_IMAGE_TILING_OPTIMAL, VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT,
VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT, depthImage, depthImageMemory);
    ...
}
    
```

Comme nous avons créé de nouvelles ressources, nous devons les libérer :

```

void cleanupSwapChain() {
    vkDestroyImageView(device, colorImageView, nullptr);
    vkDestroyImage(device, colorImage, nullptr);
    vkFreeMemory(device, colorImageMemory, nullptr);
    ...
}
    
```

Mettez également à jour la fonction `recreateSwapChain()` afin de reconstruire une nouvelle image pour les couleurs avec la bonne résolution lorsque la fenêtre est redimensionnée :

```

void recreateSwapChain() {
    ...
    createGraphicsPipeline();
    createColorResources();
    createDepthResources();
    ...
}
    
```

Nous avons fini le paramétrage initial du MSAA. Nous devons maintenant utiliser ces nouvelles ressources dans le pipeline, le tampon d'images et la passe de rendu !

XI-D - Ajout de nouvelles attaches

Commençons par la passe de rendu. Modifiez la fonction `createRenderPass()` et mettez à jour les attaches de couleur et de profondeur dans la structure de création de la passe de rendu :

```

void createRenderPass() {
    ...
    colorAttachment.samples = msaaSamples;
    colorAttachment.finalLayout = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL;
    ...
    depthAttachment.samples = msaaSamples;
    ...
}
    
```

Remarquez que nous avons changé l'agencement final (`finalLayout`) de `VK_IMAGE_LAYOUT_PRESENT_SRC_KHR` pour `VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL`. En effet, les images multiéchantillonnées ne peuvent être directement présentées. Nous devons les convertir en une image plus classique. Cette contrainte ne s'applique pas au tampon de profondeur, car nous ne l'affichons dans aucun cas. Par conséquent, nous devons ajouter une unique attache pour les couleurs que nous appelons attache de conversion :

```
...
VkAttachmentDescription colorAttachmentResolve{};
colorAttachmentResolve.format = swapChainImageFormat;
colorAttachmentResolve.samples = VK_SAMPLE_COUNT_1_BIT;
colorAttachmentResolve.loadOp = VK_ATTACHMENT_LOAD_OP_DONT_CARE;
colorAttachmentResolve.storeOp = VK_ATTACHMENT_STORE_OP_STORE;
colorAttachmentResolve.stencilLoadOp = VK_ATTACHMENT_LOAD_OP_DONT_CARE;
colorAttachmentResolve.stencilStoreOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
colorAttachmentResolve.initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
colorAttachmentResolve.finalLayout = VK_IMAGE_LAYOUT_PRESENT_SRC_KHR;
...
```

La passe de rendu doit maintenant être configurée pour convertir l'image multiéchantillonnée en attache classique. Créez une nouvelle référence d'attache pointant sur le tampon de couleurs qui nous servira de destination :

```
...
VkAttachmentReference colorAttachmentResolveRef{};
colorAttachmentResolveRef.attachment = 2;
colorAttachmentResolveRef.layout = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL;
...
```

Modifiez la variable membre de la structure de sous-passe `pResolveAttachments` pour indiquer la nouvelle référence d'attache. C'est suffisant pour que la passe de rendu effectue une conversion, ce qui nous permet d'afficher l'image à l'écran :

```
...
subpass.pResolveAttachments = &colorAttachmentResolveRef;
...
```

Fournissez ensuite la nouvelle attache de couleurs à la structure de création de la passe de rendu.

```
...
std::array<VkAttachmentDescription, 3> attachments = {colorAttachment, depthAttachment,
colorAttachmentResolve};
...
```

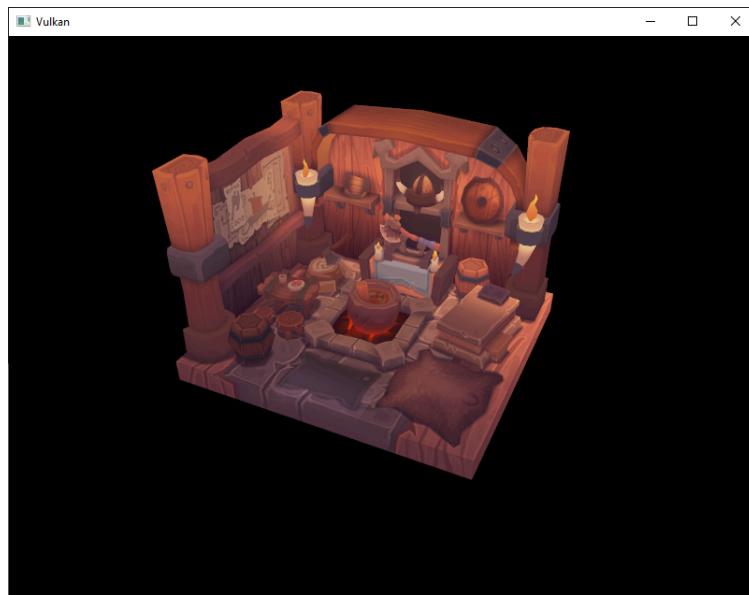
Modifiez ensuite la fonction `createFramebuffer()` et ajoutez la nouvelle vue d'image à la liste :

```
void createFrameBuffers() {
    ...
    std::array<VkImageView, 3> attachments = {
        colorImageView,
        depthImageView,
        swapChainImageViews[i]
    };
    ...
}
```

Enfin, il ne reste plus qu'à indiquer (dans la fonction `createGraphicsPipeline()`) au nouveau pipeline d'utiliser plusieurs échantillons :

```
void createGraphicsPipeline() {
    ...
    multisampling.rasterizationSamples = msaaSamples;
    ...
}
```

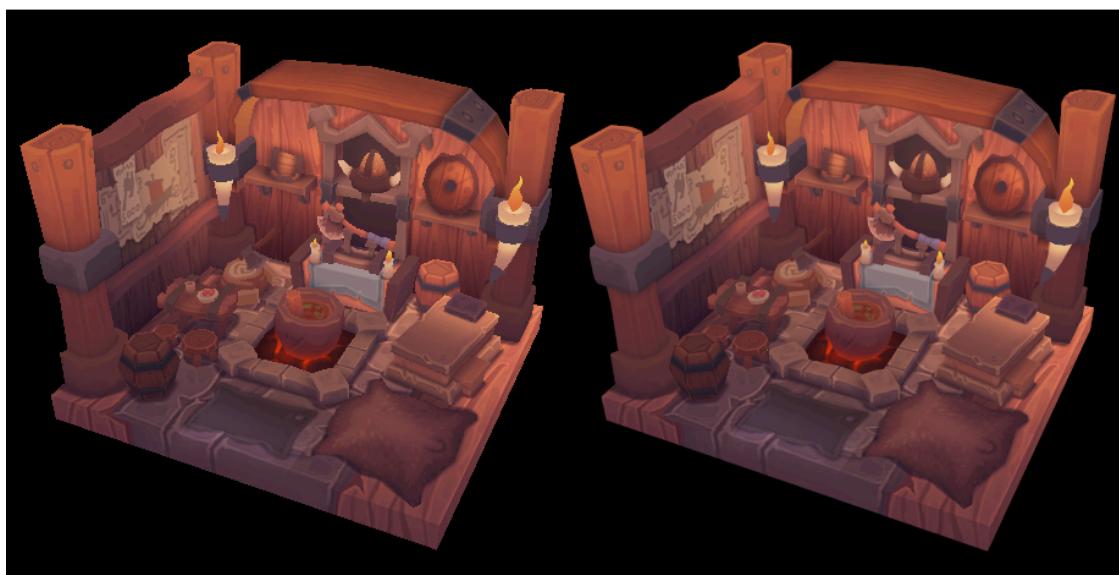
Lancez votre programme et vous devriez voir ceci :



Comme pour le mipmapping, la différence n'est pas immédiatement visible. En regardant de plus près, vous remarquerez que les bordures sont moins crénelées et que l'image est plus lisse qu'avant.

Without multisampling

With multisampling (MSAAx8)



La différence est encore plus visible en zoomant sur un bord :

Without multisampling



With multisampling (MSAAx8)



XI-E - Amélioration de la qualité

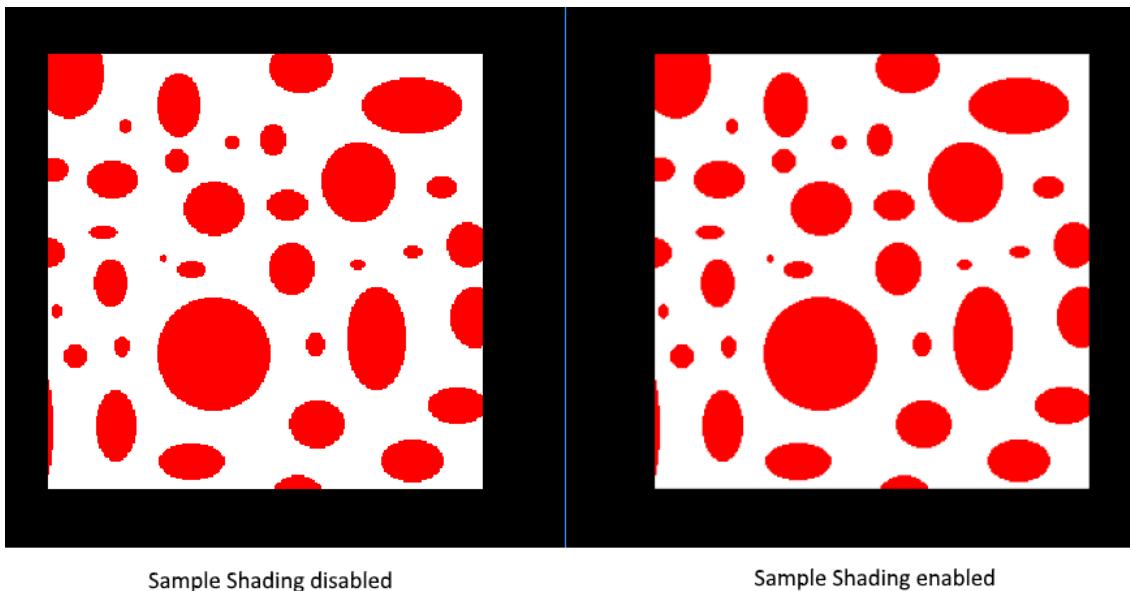
Notre implémentation du MSAA est limitée et cela peut impacter la qualité de l'image affichée dans des scènes plus détaillées. Par exemple, nous ne corrigons pas les problèmes potentiels liés au crênelage causé par les shaders : c'est-à-dire que le MSAA n'adoucit que les bordures de la géométrie, mais pas son remplissage. Cela est marquant, lorsque vous affichez un polygone lisse alors que la texture le remplissant sera crénelée, car elle contient des hauts contrastes. Une façon de résoudre ce problème est d'activer l'échantillonnage des fragments (**sample shading**), qui améliore encore la qualité de l'image au prix de performances encore réduites.

```

void createLogicalDevice() {
    ...
    deviceFeatures.sampleRateShading = VK_TRUE; // active la fonctionnalité d'échantillonnage des
    fragments pour ce périphérique
    ...
}

void createGraphicsPipeline() {
    ...
    multisampling.sampleShadingEnable = VK_TRUE; // active l'échantillonnage des fragments dans
    le pipeline
    multisampling.minSampleShading = .2f; // fraction minimale pour l'échantillonnage des
    fragments ; plus la valeur est proche de 1, plus le résultat est doux
    ...
}
    
```

Nous n'activons pas l'échantillonnage des fragments dans notre application, mais dans certains cas son activation permet une nette amélioration de la qualité du rendu :



Sample Shading disabled

Sample Shading enabled

XI-F - Conclusion

Il nous a fallu beaucoup de travail pour en arriver là, mais vous avez maintenant une bonne connaissance des bases de Vulkan. Ces connaissances vous permettent maintenant d'explorer d'autres fonctionnalités, comme :

- les constantes poussées (push constants) ;
- le rendu instancié ;
- les variables uniformes dynamiques ;
- les descripteurs d'images et d'échantillonneurs séparés ;
- la mise en cache de pipeline ;
- la génération des tampons de commandes depuis plusieurs threads ;
- les sous-passes multiples ;
- les shaders de calcul (compute shaders).

Le programme actuel peut être grandement étendu, par exemple en ajoutant l'éclairage Blinn-Phong, des effets de post-traitement et l'application des ombres. Vous devriez pouvoir apprendre ces techniques depuis des tutoriels conçus pour d'autres bibliothèques, car ces algorithmes fonctionnent de la même façon, quelle que soit la bibliothèque.

Code C++ / Vertex shader / Fragment shader

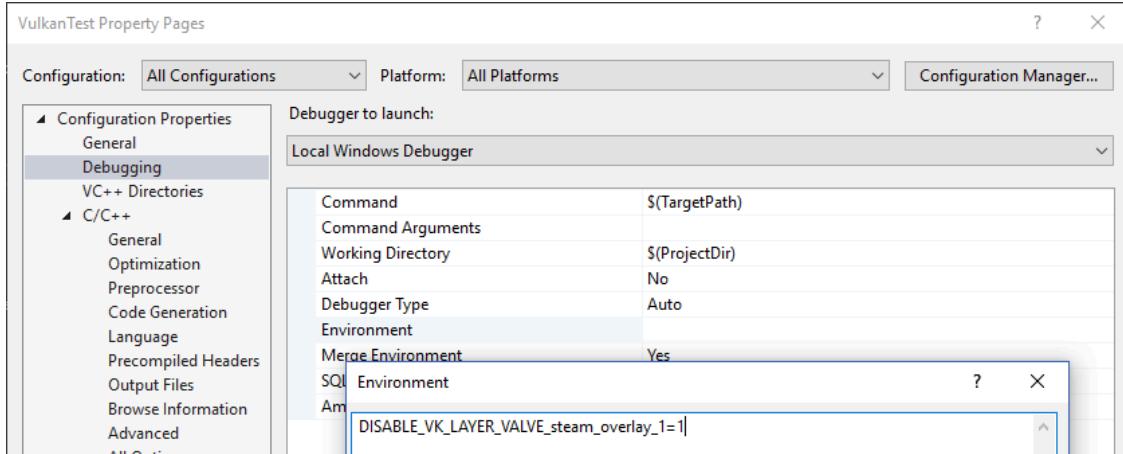
XII - Résolution des problèmes

Cette page liste les problèmes les plus courants que vous pourriez rencontrer lors du développement d'une application Vulkan.

- **J'obtiens une erreur de violation d'accès dans les couches de validation** : assurez-vous que les logiciels MSI Afterburner/RivaTuner Statistics Server sont arrêtés. Ils possèdent des problèmes de compatibilité avec Vulkan.
- **Je ne vois aucun message provenant des couches de validation. Les couches de validation ne sont pas disponibles** : assurez-vous que les couches de validation peuvent écrire leurs messages en laissant le terminal ouvert après l'exécution. Avec Visual Studio, lancez le programme avec Ctrl-F5 et non pas F5. Sous Linux, lancez le programme depuis un terminal. S'il n'y a toujours pas de message et que vous êtes certain que les couches de validation sont actives alors vous devez vérifier que le SDK Vulkan est correctement installé [comme indiqué ici](#), (section « Verify the Installation »). Assurez-vous également que le SDK est au moins de la version 1.1.106.0 qui supporte la couche VK_LAYER_KHRONOS_validation.

- **vkCreateSwapchainKHR provoque une erreur dans SteamOverlayVulkanLayer64.dll** : il semble qu'il y ait un problème de compatibilité avec la version bêta du client Steam. Voici quelques méthodes pour contourner le problème :

- désactiver le programme bêta de Steam ;
- définissez la variable d'environnement `DISABLE_VK_LAYER_VALVE_steam_overlay_1` à 1 ;
-



supprimez l'entrée dans la base de registre `HKEY_LOCAL_MACHINE\SOFTWARE\Khronos\Vulkan\ImplicitLayers` de la couche Vulkan de la surcouche Steam.