Vulkan Tutorial

Alexander Overvoorde

September 2025

Contents

Introduction **5**
- À propos . . . 5
- E-book . . . 6
- Structure du tutoriel . . . 6

Vue d'ensemble **8**
- Origine de Vulkan . . . 8
- Le nécessaire pour afficher un triangle . . . 8
  - Étape 1 - Instance et sélection d'un physical device . . . 9
  - Étape 2 – Logical device et familles de queues (queue families) . . . 9
  - Étape 3 – Surface d'affichage (window surface) et swap chain . . . 9
  - Étape 4 - Image views et framebuffers . . . 9
  - Étape 5 - Render passes . . . 10
  - Étape 6 - Le pipeline graphique . . . 10
  - Étape 7 - Command pools et command buffers . . . 10
  - Étape 8 - Boucle principale . . . 10
  - Résumé . . . 11
- Concepts de l'API . . . 11
  - Conventions . . . 11
  - Validation layers . . . 12

Environnement de développement **13**
- Windows . . . 13
  - SDK Vulkan . . . 13
  - GLFW . . . 14
  - GLM . . . 15
  - Préparer Visual Studio . . . 15
- Linux . . . 21
  - Paquets Vulkan . . . 22
  - GLFW . . . 22
  - GLM . . . 23
  - Compilateur de shader . . . 23
  - Préparation d'un fichier makefile . . . 23
- MacOS . . . 25
  - Le SDK Vulkan . . . 25
  - GLFW . . . 26
  - GLM . . . 26
  - Préparation de Xcode . . . 27

Dessiner un triangle **31**
- Mise en place . . . 31
  - Code de base . . . 31

Instance . . . 34
Validation layers . . . 37
Physical devices et queue families . . . 44
Logical device et queues . . . 50
Présentation . . . 52
Window surface . . . 52
Swap chain . . . 55
Image views . . . 64
Pipeline graphique basique . . . 66
Introduction . . . 66
Modules shaders . . . 69
Fonctions fixées . . . 76
Render pass . . . 82
Conclusion . . . 85
Effectuer le rendu . . . 87
Framebuffers . . . 87
Command buffers . . . 89
Rendu et présentation . . . 93
Recréation de la swap chain . . . 104
Introduction . . . 104
Recréer la swap chain . . . 104
Swap chain non-optimales ou dépassées . . . 106
Explicitement gérer les redimensionnements . . . 107
Gestion de la minimisation de la fenêtre . . . 108

Vertex buffers **109**
Description des entrées des sommets . . . 109
Introduction . . . 109
Vertex shader . . . 109
Sommets . . . 110
Lier les descriptions . . . 110
Description des attributs . . . 111
Entrée des sommets dans la pipeline . . . 112
Création de vertex buffers . . . 112
Introduction . . . 112
Création d'un buffer . . . 112
Fonctionnalités nécessaires de la mémoire . . . 114
Allocation de mémoire . . . 115
Remplissage du vertex buffer . . . 116
Lier le vertex buffer . . . 117
Buffer intermédiaire . . . 118
Introduction . . . 118
Queue de transfert . . . 119
Abstraction de la création des buffers . . . 119
Utiliser un buffer intermédiaire . . . 120
Conclusion . . . 122
Index buffer . . . 123
Introduction . . . 123
Création d'un index buffer . . . 123
Utilisation d'un index buffer . . . 125

Uniform buffers **127**
Descriptor layout et buffer . . . 127
Introduction . . . 127

Vertex shader . . . 128
Organisation du set de descripteurs . . . 128
Uniform buffer . . . 130
Mise à jour des données uniformes . . . 131
Descriptor pool et sets . . . 133
Introduction . . . 133
Pool de descripteurs . . . 133
Set de descripteurs . . . 135
Utiliser des sets de descripteurs . . . 137
Alignement . . . 138
Plusieurs sets de descripteurs . . . 140

Texture mapping **141**
Images . . . 141
Introduction . . . 141
Librairie de chargement d'image . . . 142
Charger une image . . . 142
Buffer intermédiaire . . . 144
Texture d'image . . . 144
Transitions de l'organisation . . . 147
Copier un buffer dans une image . . . 150
Préparer la texture d'image . . . 151
Derniers champs de la barrière de transition . . . 151
Nettoyage . . . 152
Vue sur image et sampler . . . 153
Vue sur une image texture . . . 153
Samplers . . . 155
Capacité du device à supporter l'anistropie . . . 158
Sampler d'image combiné . . . 159
Introduction . . . 159
Modifier les descripteurs . . . 159
Coordonnées de texture . . . 161
Shaders . . . 162

Buffer de profondeur **167**
Introduction . . . 167
Géométrie en 3D . . . 167
Image de pronfondeur et views sur cette image . . . 170
Explicitement transitionner l'image de profondeur . . . 172
Render pass . . . 173
Framebuffer . . . 174
Supprimer les valeurs . . . 175
État de profondeur et de stencil . . . 175
Gestion des redimensionnements de la fenêtre . . . 176

Charger des modèles **178**
Introduction . . . 178
Une librairie . . . 178
Exemple de modèle . . . 179
Charger les vertices et les indices . . . 179
Déduplication des vertices . . . 182

Générer des mipmaps **184**
Introduction . . . 184
Création des images . . . 184

Génération des mipmaps 186
Support pour le filtrage linéaire 189
Sampler 190

Multisampling **194**
Introduction 194
Récupération du nombre maximal de samples 196
Mettre en place une cible de rendu 196
Ajouter de nouveaux attachements 198
Amélioration de la qualité 201
Conclusion 202

FAQ **203**

Introduction

> NOTICE: The English version of the tutorial has recently changed significantly (for the better) and these changes have not yet been applied to the French translation.

À propos

Ce tutoriel vous enseignera les bases de l'utilisation de l'API Vulkan qui expose les graphismes et le calcul sur cartes graphiques. Vulkan est une nouvelle API créée par le groupe Khronos (connu pour OpenGL). Elle fournit une bien meilleure abstraction des cartes graphiques modernes. Cette nouvelle interface vous permet de mieux décrire ce que votre application souhaite faire, ce qui peut mener à de meilleures performances et à des comportements moins variables comparés à des APIs existantes comme 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 cross-platform, et vous permet ainsi de développer pour Windows, Linux, Mac et Android en même temps.

Il y a cependant un contre-coup à ces avantages. L'API vous impose d'être explicite sur chaque détail. Vous ne pourrez rien laisser au hasard, et il n'y a aucune structure, aucun environnement créé pour vous par défaut. Il faudra le recréer à partir de rien. Le travail du driver graphique sera ainsi considérablement réduit, ce qui implique un plus grand travail de votre part pour assurer un comportement correct.

Le message véhiculé ici est que Vulkan n'est pas fait pour tout le monde. Cette API est conçue pour les programmeurs concernés par la programmation avec GPU de haute performance, et qui sont prêts à y travailler sérieusement. Si vous êtes intéressées dans le développement de jeux vidéo, et moins dans les graphismes eux-mêmes, vous devriez plutôt continuer d'utiliser OpenGL et DirectX, qui ne seront pas dépréciés en faveur de Vulkan avant un certain temps. Une autre alternative serait d'utiliser un moteur de jeu comme Unreal Engine ou Unity, qui pourront être capables d'utiliser Vulkan tout en exposant une API de bien plus haut niveau.

Cela étant dit, présentons quelques prérequis pour ce tutoriel:

- Une carte graphique et un driver compatibles avec Vulkan (NVIDIA, AMD, Intel)
- De l'expérience avec le C++ (familiarité avec RAII, listes d'initialisation, et autres fonctionnalités modernes)
- Un compilateur avec un support décent des fonctionnalités du C++17 (Visual Studio 2017+, GCC 7+ ou Clang 5+)
- Un minimum d'expérience dans le domaine de la programmation graphique

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 la projection de perspective, par exemple. Lisez ce livre pour une bonne introduction des concepts de rendu 3D. D'autres ressources pour le développement d'application graphiques sont : * Ray tracing en un week-end * Livre sur le Physical Based Rendering * Une application de Vulkan dans les moteurs graphiques open source Quake et de DOOM 3

Vous pouvez utiliser le C plutôt que le C++ si vous le souhaitez, mais vous devrez utiliser une autre bibliothèque d'algèbre linéaire et vous structurerez vous-même votre code. Nous utiliserons des possibilités du

C++ (RAII, classes) pour organiser la logique et la durée de vie des ressources. Il existe aussi une version alternative 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 l'API de base, nous allons utiliser l'API C originelle pour travailler avec Vulkan. Cependant, si vous utilisez le C++, vous pourrez préférer utiliser le binding Vulkan-Hpp plus récent, qui permet de s'éloigner de certains détails ennuyeux et d'éviter certains types d'erreurs.

E-book

Si vous préférez lire ce tutoriel en E-book, vous pouvez en télécharger une version EPUB ou PDF ici:

- EPUB
- PDF

Structure du tutoriel

Nous allons commencer par une approche générale du fonctionnement de Vulkan, et verrons d'abord rapidement le travail à effectuer pour afficher un premier triangle à l'écran. Le but de chaque petite étape aura ainsi plus de sens quand vous aurez compris leur rôle dans le fonctionnement global. 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, sur Linux Ubuntu avec GCC et sur MacOS.

Après cela, nous implémenterons tous les éléments nécessaires à un programme Vulkan pour afficher votre 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 l'API 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 certaine fonctionnalité de Vulkan. Ainsi le site peut vous être utile comme référence. Toutes les fonctions et les types Vulkan sont liés à leur spécification, vous pouvez donc cliquer dessus pour en apprendre plus. La spécification est par contre en Anglais. Vulkan est une API récente, il peut donc y avoir des lacunes dans la spécification elle-même. Vous êtes encouragés à transmettre vos retours dans ce repo Khronos.

Comme indiqué plus haut, Vulkan est une API assez prolixe, avec de nombreux paramètres, pensés pour vous fournir un maximum de contrôle sur le hardware graphique. Ainsi des opérations comme créer une texture prennent de nombreuses étapes qui doivent être répétées chaque fois. Nous créerons notre propre collection de fonctions d'aide tout le 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 bug et que voulez comparer. Tous les fichiers de code ont été testés sur des cartes graphiques de différents vendeurs pour pouvoir affirmer qu'ils fonctionnent. Chaque chapitre possède également une section pour écrire vos commentaires en relation avec le sujet discuté. Veuillez y indiquer votre plateforme, la version de votre driver, votre code source, le comportement attendu et celui obtenu pour nous simplifier la tâche de vous aider.

Ce tutoriel est destiné à être un effort de communauté. Vulkan est encore une API 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 créer une issue ou une pull request sur le repo GitHub. Vous pouvez *watch* le dépôt afin d'être notifié des dernières mises à jour du site.

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 API graphique auparavant, vous devez savoir qu'il y a nombre d'étapes avant d'afficher la première géométrie sur l'écran. Il y aura beaucoup plus de ces étapes préliminaires avec Vulkan, mais vous verrez que chacune d'entre elle est simple à comprendre et n'est pas redondante. Gardez aussi à l'esprit qu'une fois que vous savez afficher un triangle - certes peu intéressant -, afficher un modèle 3D parfaitement texturé ne nécessite pas tant de travail supplémentaire, et que chaque étape à partir de ce point est bien mieux récompensée visuellement.

Si vous rencontrez un problème en suivant ce tutoriel, vérifiez d'abord dans la FAQ que votre problème et sa solution n'y sont pas déjà listés. Si vous êtes toujours coincé après cela, demandez de l'aide dans la section des commentaires du chapitre le plus en lien avec votre problème.

Prêt à vous lancer dans le futur des API graphiques de haute performance? Allons-y!

Vue d'ensemble

Ce chapitre commencera par introduire Vulkan et les problèmes auxquels l'API s'adresse. Nous nous intéresserons ensuite aux éléments requis pour l'affichage d'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 l'API est communément utilisée.

Origine de Vulkan

Comme les APIs précédentes, Vulkan est conçue comme une abstraction des GPUs. Le problème avec la plupart de ces APIs est qu'elles furent créées à une époque où le hardware graphique était limité à des fonctionnalités prédéfinies tout juste configurables. Les développeurs devaient fournir les sommets dans un format standardisé, et étaient ainsi à la merci des constructeurs pour les options d'éclairage et les jeux d'ombre.

Au fur et à mesure que les cartes graphiques progressèrent, elles offrirent de plus en plus de fonctionnalités programmables. Il fallait alors intégrer toutes ces nouvelles fonctionnalités aux APIs existantes. Ceci résulta en une abstraction peu pratique et le driver devait deviner l'intention du développeur pour relier le programme aux architectures modernes. C'est pour cela que les drivers étaient mis à jour si souvent, et que certaines augmentaient soudainement les performances. À cause de la complexité de ces drivers, les développeurs devaient gérer les différences de comportement entre les fabricants, dont par exemple des tolérances plus ou moins importantes pour les shaders. Un exemple de fonctionnalité est le tiled rendering, pour laquelle une plus grande flexibilité mènerait à de meilleures performance. Ces APIs anciennes souffrent également d'une autre limitation : le support limité du multithreading, menant à des goulot d'étranglement du coté du CPU. Au-delà des nouveautés techniques, la dernière décennie a aussi été témoin de l'arrivée de matériel mobile. Ces GPUs portables ont des architectures différentes qui prennent en compte des contraintes spatiales ou énergétiques.

Vulkan résout ces problèmes en ayant été repensée à partir de rien pour des architectures modernes. Elle réduit le travail du driver en permettant (en fait en demandant) au développeur d'expliciter ses objectifs en passant par une API plus prolixe. Elle permet à plusieurs threads d'invoquer des commandes de manière asynchrone. Elle supprime les différences lors de la compilation des shaders en imposant un format en bytecode compilé par un compilateur officiel. Enfin, elle reconnaît les capacités des cartes graphiques modernes en unifiant le computing et les graphismes dans une seule et unique API.

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 afin d'y replacer tous les éléments.

Étape 1 - Instance et sélection d'un physical device

Une application commence par paramétrer l'API 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 `VkInstance`, vous pouvez demander l'accès à du hardware compatible avec Vulkan, et ainsi sélectionner un ou plusieurs «`VkPhysicalDevice`» pour y réaliser vos opérations. Vous pouvez traiter des informations telles que la taille de la VRAM ou des capacités de la carte graphique, et ainsi préférer par exemple du matériel dédié.

Étape 2 – Logical device et familles de queues (queue families)

Après avoir sélectionné le hardware qui vous convient, vous devez créer un `VkDevice` (logical device). Vous décrivez pour cela quelles `VkPhysicalDeviceFeatures` vous utiliserez, comme l'affichage multi-fenêtre ou des floats de 64 bits. Vous devrez également spécifier quelles `vkQueueFamilies` vous utiliserez. La plupart des opérations, comme les commandes d'affichage et les allocations de mémoire, sont exécutés de manière asynchrone en les envoyant à une `VkQueue`. Ces queues sont crées à partir d'une famille de queues, chacune de ces dernières supportant uniquement une certaine collection d'opérations. Il pourrait par exemple y avoir des familles différentes pour les graphismes, le calcul et les opérations mémoire. L'existence d'une famille peut aussi être un critère pour la sélection d'un physical device. En effet une queue capable de traiter les commandes graphiques et opérations mémoire permet d'augmenter encore un peu les performances. Il sera possible qu'un périphérique supportant Vulkan ne fournisse aucun graphisme, mais à ce jour toutes les opérations que nous allons utiliser devraient être disponibles.

Étape 3 – Surface d'affichage (window surface) et swap chain

À moins que vous ne soyez intéressé que par le rendu off-screen, vous devrez créer une fenêtre dans laquelle afficher les éléments. Les fenêtres peuvent être crées avec les APIs spécifiques aux différentes plateformes ou avec des librairies telles que GLFW et SDL. Nous utiliserons GLFW dans ce tutoriel, mais nous verrons tout cela dans le prochain chapitre.

Nous avons cependant encore deux composants à évoquer pour afficher quelque chose : une Surface (`VkSurfaceKHR`) et une Swap Chain (`VkSwapchainKHR`). Remarquez le suffixe «KHR», qui indique que ces fonctionnalités font partie d'une extension. L'API est elle-même totalement agnostique de la plateforme sur laquelle elle travaille, nous devons donc utiliser l'extension standard WSI (Window System Interface) pour interagir avec le gestionnaire de fenêtre. La Surface est une abstraction cross-platform de la fenêtre, et est généralement créée en fournissant une référence à une fenêtre spécifique à la plateforme, par exemple «HWND» sur Windows. Heureusement pour nous, la librairie GLFW possède une fonction permettant de gérer tous les détails spécifiques à la plateforme pour nous.

La swap chain est une collection de cibles sur lesquelles nous pouvons effectuer un rendu. Son but principal est d'assurer que l'image sur laquelle nous travaillons n'est pas celle utilisée par l'écran. Nous sommes ainsi sûrs 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 cible disponible. Une fois le traitement de la cible terminé, nous la rendrons à la swap chain qui l'utilisera en temps voulu pour l'affichage à l'écran. Le nombre de cibles et les conditions de leur affichage dépend du mode utilisé lors du paramétrage de la Swap Chain. Ceux-ci peuvent être le double buffering (synchronisation verticale) ou le 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 fenêtre, et ce en vous donnant la possibilité de créer une surface qui fait la taille de l'écran. Vous pouvez alors par exemple créer votre propre gestionnaire de fenêtre.

Étape 4 - Image views et framebuffers

Pour dessiner sur une image originaire de la swap chain, nous devons l'encapsuler dans une `VkImageView` et un `VkFramebuffer`. Une vue sur une image correspond à une certaine partie de l'image utilisée, et un framebuffer référence plusieurs vues pour les traiter comme des cible 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 framebuffers pour chacune d'entre elles, puis sélectionnerons celle qui nous convient au moment de l'affichage.

Étape 5 - Render passes

Avec Vulkan, une render pass décrit les types d'images utilisées lors du rendu, comment elles sont utilisées et comment leur contenu doit être traité. Pour notre affichage d'un triangle, nous dirons à Vulkan que nous utilisons une seule image pour la couleur et que nous voulons qu'elle soit préparée avant l'affichage en la remplissant d'une couleur opaque. Là où la passe décrit le type d'images utilisées, un framebuffer sert à lier les emplacements utilisés par la passe à une image complète.

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 les opérations réalisées par le depth buffer (gestion de la profondeur), et les étapes programmables à l'aide de `VkShaderModules`. Ces derniers sont créés à partir de byte code. Le driver doit également être informé des cibles du rendu utilisées dans le pipeline, ce que nous lui disons en référençant la render pass.

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 conformation des sommets, 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 la couleur de fond. Les états doivent aussi être anticipés : il n'y a par exemple pas de fonction de blending par défaut.

La bonne nouvelle est que grâce à cette anticipation, ce qui équivaut à peu près à une compilation versus une interprétation, il y a beaucoup plus d'optimisations possibles pour le driver et le temps d'exécution est plus prévisible, car les grandes étapes telles le changement de pipeline sont faites très explicites.

Étape 7 - Command pools et command buffers

Comme dit plus haut, nombre d'opérations comme le rendu doivent être transmise à une queue. Ces opérations doivent d'abord être enregistrées dans un `VkCommandBuffer` avant d'être envoyées. Ces command buffers sont alloués à partir d'une «`VkCommandPool`» spécifique à une queue family. Pour afficher notre simple triangle nous devrons enregistrer les opérations suivantes :

- Lancer la render pass
- Lier le pipeline graphique
- Afficher 3 sommets
- Terminer la passe

Du fait que l'image que nous avons extraite du framebuffer pour nous en servir comme cible dépend de l'image que la swap chain nous fournira, nous devons préparer un command buffer pour chaque image possible et choisir le bon au moment de l'affichage. Nous pourrions en créer un à chaque frame mais ce ne serait pas aussi efficace.

Étape 8 - Boucle principale

Maintenant que nous avons inscrit les commandes graphiques dans des command buffers, la boucle principale n'est plus qu'une question d'appels. Nous acquérons d'abord une image de la swap chain en utilisant `vkAcquireNextImageKHR`. Nous sélectionnons ensuite le command buffer approprié 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 objets de synchronisation tels que des sémaphores pour nous assurer que les opérations sont exécutées dans l'ordre voulu. L'exécution du command buffer d'affichage doit de plus attendre que l'acquisition de l'image soit terminée, sinon nous pourrions dessiner sur une image utilisée pour l'affichage. L'appel à `vkQueuePresentKHR` doit aussi attendre que l'affichage soit terminé.

Résumé

Ce 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 vertex Buffers, créer des Uniform Buffers et envoyer des textures, mais nous verrons cela dans des chapitres suivants. Nous allons commencer par les bases car Vulkan a suffisamment d'étapes ainsi. Notez que nous allons "tricher" en écrivant les coordonnées du triangle directement dans un shader, afin d'éviter l'utilisation d'un vertex buffer qui nécessite une certaine familiarité avec les Command Buffers.

En résumé nous devrons, pour afficher un triangle :

- Créer une `VkInstance`
- Sélectionner une carte graphique compatible (`VkPhysicalDevice`)
- Créer un `VkDevice` et une `VkQueue` pour l'affichage et la présentation
- Créer une fenêtre, une surface dans cette fenêtre et une swap chain
- Considérer les images de la swap chain comme des `VkImageViews` puis des `VkFramebuffers`
- Créer la render pass spécifiant les cibles d'affichage et leurs usages
- Créer des framebuffers pour ces passes
- Générer le pipeline graphique
- Allouer et enregistrer des Command Buffers contenant toutes les commandes pour toutes les images de la swap chain
- Dessiner sur les frames en acquérant une image, en soumettant la commande d'affichage correspondante et en retournant l'image à la swap chain

Cela fait beaucoup d'étapes, cependant le but de chacune d'entre elles sera explicitée 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.

Concepts de l'API

Ce chapitre va conclure en survolant la structure de l'API à un plus bas niveau.

Conventions

Toute les fonctions, les énumérations et les structures de Vulkan sont définies dans le header `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 énumération et les structures par 'Vk' et les macros par 'VK_'. L'API utilise massivement les structures pour la création d'objet plutôt que de passer des arguments à des fonctions. Par exemple la création d'objet 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;
```

```
10      return false;
11  }
```

De nombreuses structure imposent que l'on spécifie explicitement leur type dans le membre donnée «sType». Le membre donnée «pNext» peut pointer vers une extension et sera toujours `nullptr` dans ce tutoriel. Les fonctions qui créent ou détruisent les objets ont un paramètre appelé `VkAllocationCallbacks`, qui vous permettent 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 lesquels chaque fonction renvoie et ce qu'ils signifient.

Validation layers

Vulkan est pensé pour la performance et pour un travail minimal pour le driver. Il inclue donc très peu de gestion d'erreur et de système de débogage. Le driver crashera beaucoup plus souvent qu'il ne retournera de code d'erreur si vous faites quelque chose d'inattendu. Pire, il peut fonctionner sur votre carte graphique mais pas sur une autre.

Cependant, Vulkan vous permet d'effectuer des vérifications précises de chaque élément à l'aide d'une fonctionnalité nommée «validation layers». Ces layers consistent en du code s'insérant entre l'API et le driver, et permettent de lancer des analyses de mémoire et de relever les défauts. Vous pouvez les activer pendant le développement et les désactiver sans conséquence sur la performance. N'importe qui peut écrire ses validation layers, mais celui du SDK de LunarG est largement suffisant pour ce tutoriel. Vous aurez cependant à écrire vos propres fonctions de callback pour le traitement des erreurs émises par les layers.

Du fait que Vulkan soit si explicite pour chaque opération et grâce à l'extensivité des validations layers, trouver les causes de l'écran noir peut en fait être plus simple qu'avec OpenGL ou Direct3D!

Il reste une dernière étape avant de commencer à coder : mettre en place l'environnement de développement.

Environnement de développement

Dans ce chapitre nous allons paramétrer votre environnement de développement pour Vulkan et installer des librairies 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.

Windows

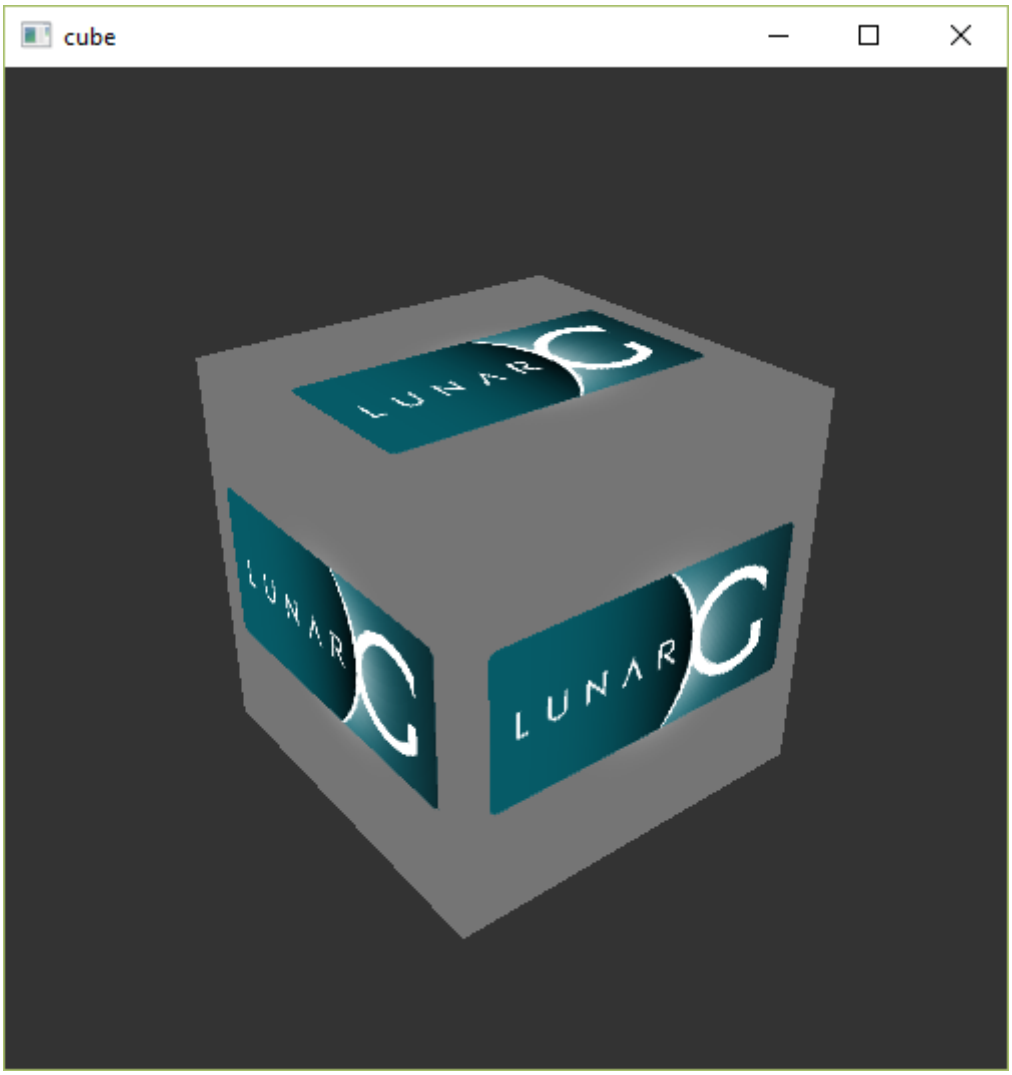
Si vous développez pour Windows, je partirai du principe que vous utilisez Visual Studio pour ce projet. Pour un support complet de C++17, il vous faut Visual Studio 2017 or 2019. Les étapes décrites ci-dessous ont été écrites pour VS 2017.

SDK Vulkan

Le composant central du développement d'applications Vulkan est le SDK. Il inclut les headers, les validation layers standards, des outils de débogage et un loader pour les fonctions Vulkan. Ce loader récupère les fonctions dans le driver 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.

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



Si vous recevez un message d'erreur assurez-vous que votre driver est à jour, 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 dans ce dossier qui vous seront utiles : "glslangValidator.exe" et "glslc.exe". Nous en aurons besoin pour la compilation des shaders. Ils transforment un code compréhensible facilement et semblable au C (le GLSL) en bytecode. Nous couvrirons cela dans le chapitre des modules shader. Le dossier "Bin" contient aussi les fichiers binaires du loader Vulkan et des validation layers. Le dossier "Lib" en contient les librairies.

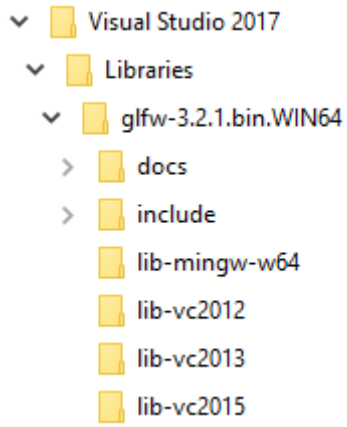
Enfin, le dossier "Include" contient les headers Vulkan. Vous pouvez parourir les autres fichiers, mais nous ne les utiliserons pas dans ce tutoriel.

GLFW

Comme dit précédemment, Vulkan ignore la plateforme sur laquelle il opère, et n'inclut pas d'outil de création de fenêtre où afficher les résultats de notre travail. Pour bien exploiter les possibilités cross-platform de Vulkan et éviter les horreurs de Win32, nous utiliserons la librairie GLFW pour créer une fenêtre et ce sur Windows, Linux ou MacOS. Il existe d'autres librairies 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

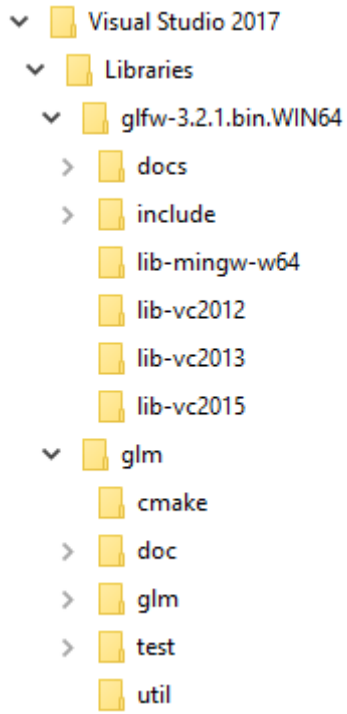
“Lib32” dans le SDK et non “Lib”. Après avoir téléchargé GLFW, extrayez l’archive à l’emplacement qui vous convient. J’ai choisi de créer un dossier “Librairies” dans le dossier de Visual Studio.



GLM

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

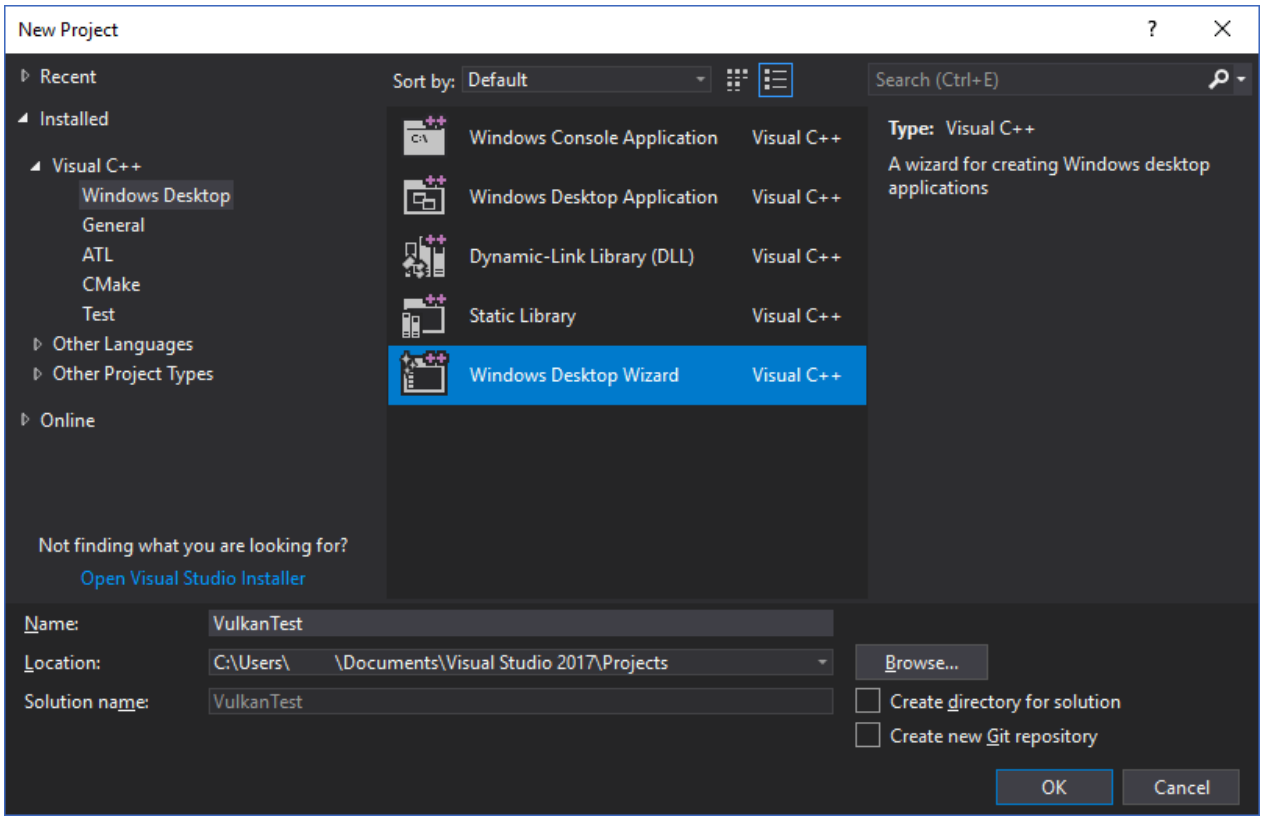
GLM est une librairie écrite exclusivement dans les headers, il suffit donc d’en télécharger la dernière version, la stocker où vous le souhaitez et l’inclure là où vous en aurez besoin. Vous devrez vous trouver avec quelque chose de semblable :



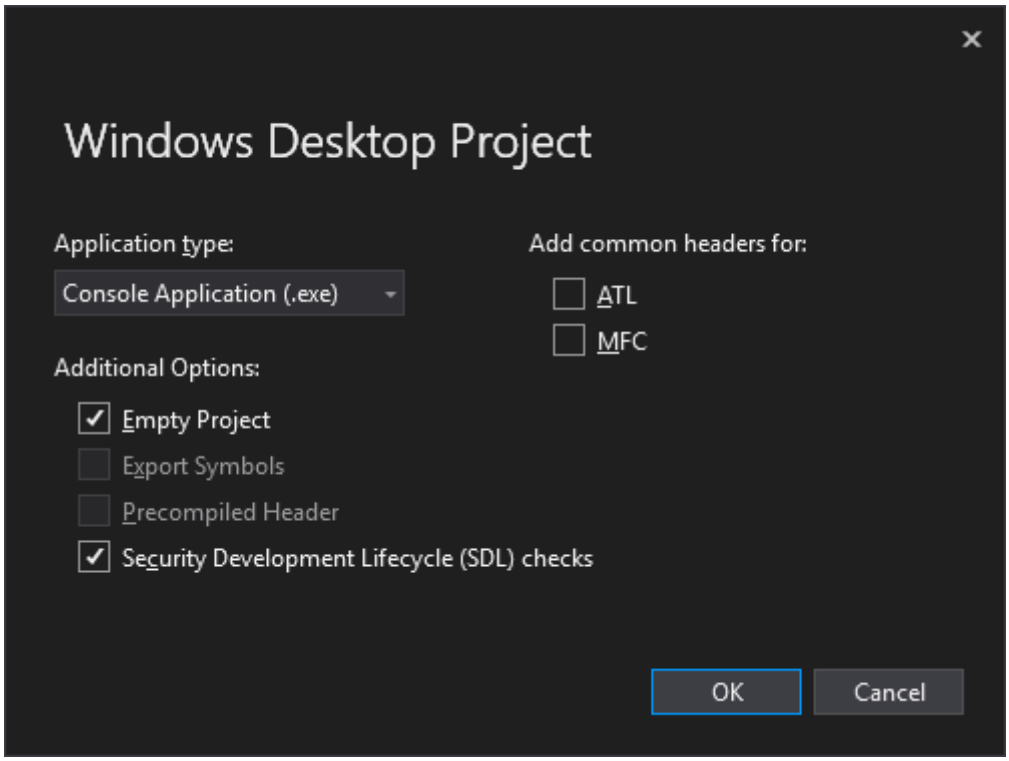
Préparer Visual Studio

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.

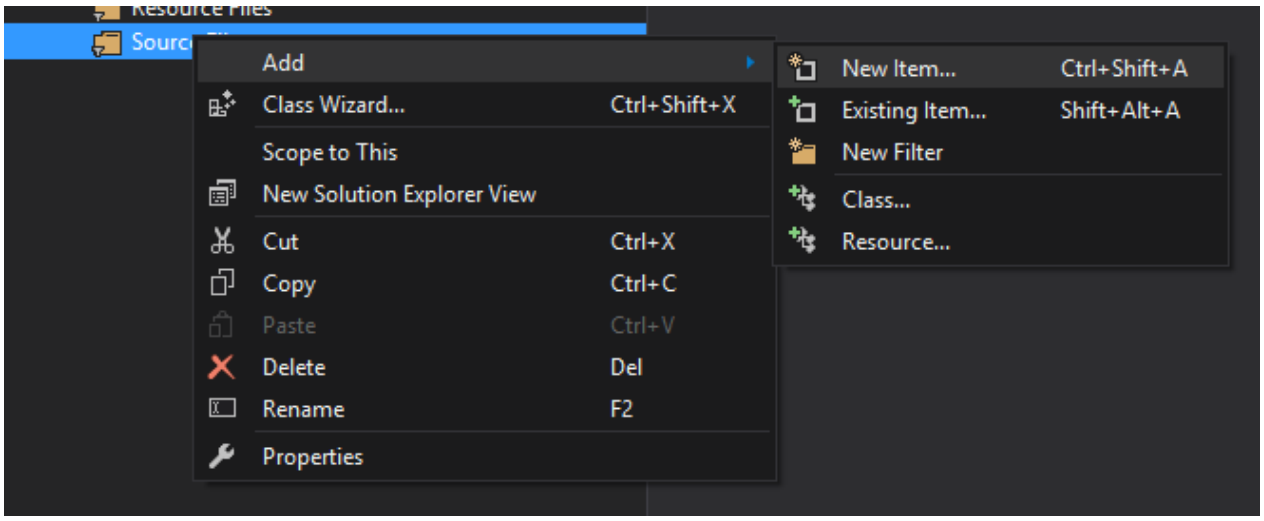
Lancez Visual Studio et créez un nouveau projet “Windows Desktop Wizard”, entrez un nom et appuyez sur OK.

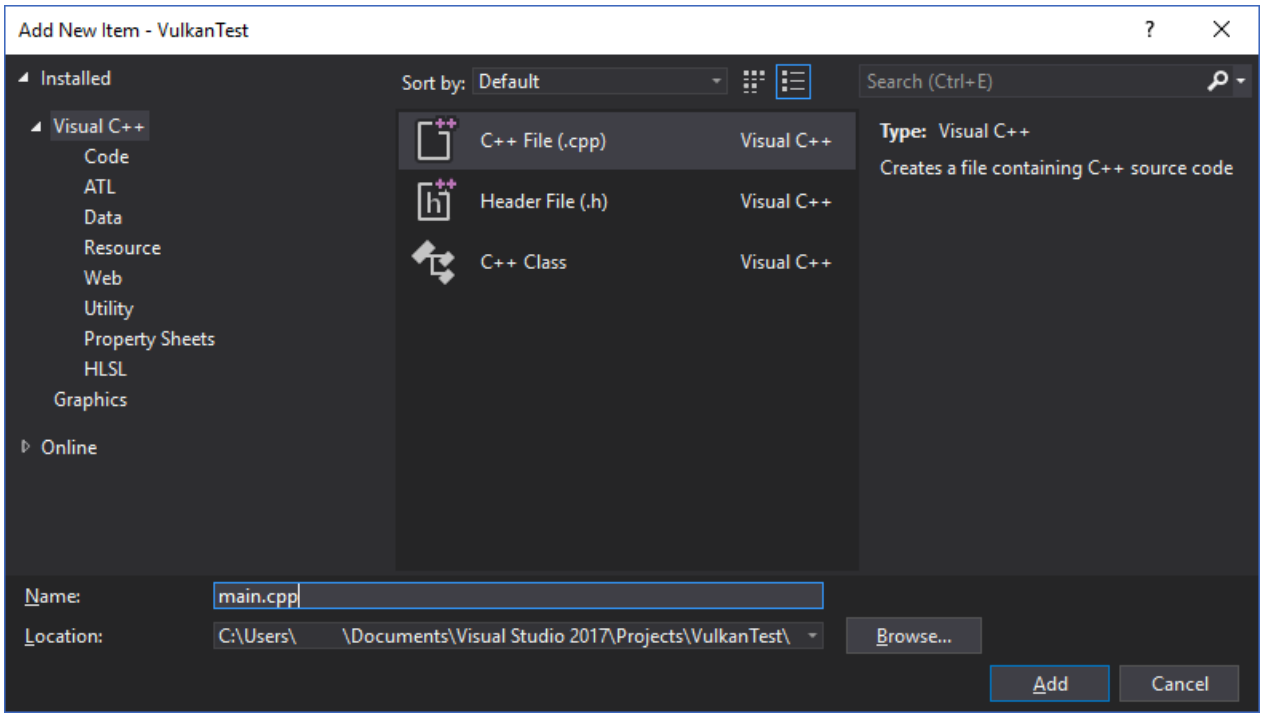


Assurez-vous que “Console Application (.exe)” est séléctionné pour le type d’application afin que nous ayons un endroit où afficher nos messages d’erreur, et cochez “Empty Project” afin que Visual Studio ne génère pas un code de base.



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.



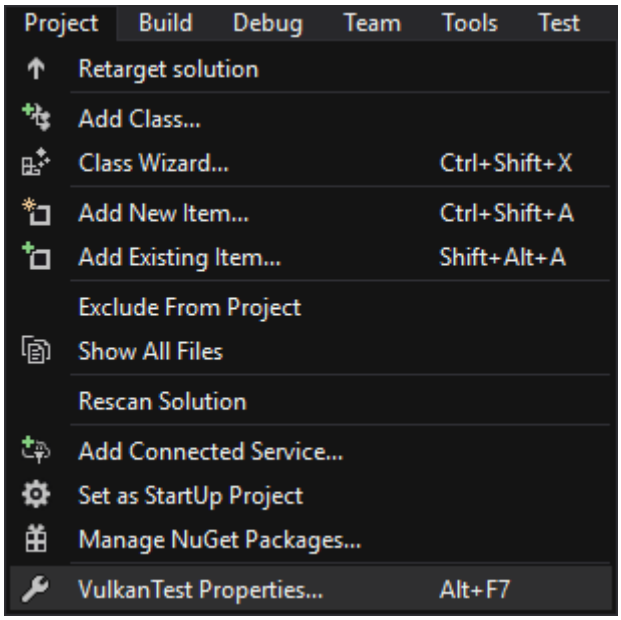


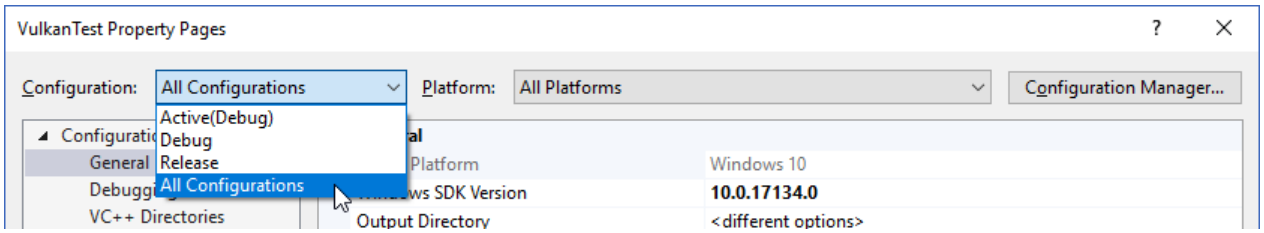
Ajoutez maintenant le code suivant à votre fichier. Ne cherchez pas à en comprendre les tenants et aboutissants, il sert juste à s'assurer que tout compile correctement et qu'une application Vulkan fonctionne. Nous recommencerons tout depuis le début dès le chapitre suivant.

```
#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();
    }
```

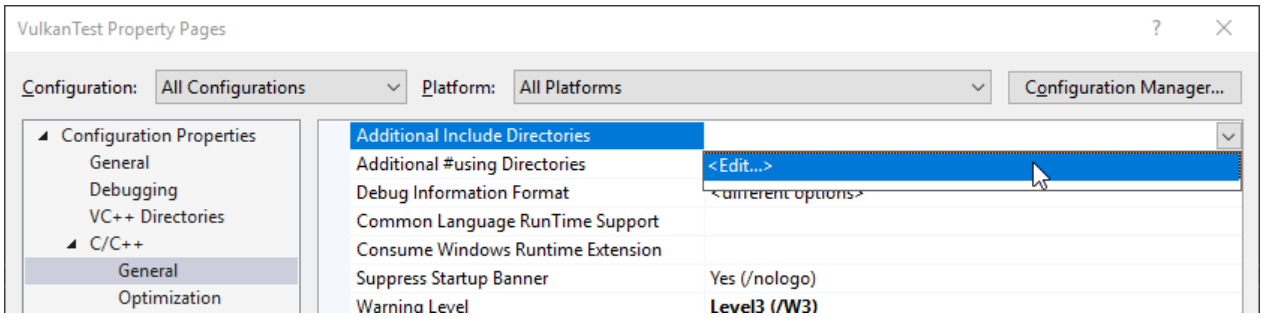
```
29
30      glfwDestroyWindow(window);
31
32      glfwTerminate();
33
34      return 0;
35  }
```

Configurons maintenant le projet afin de se débarrasser des erreurs. Ouvrez le dialogue des propriétés du projet et assurez-vous que “All Configurations” est sélectionné, car la plupart des paramètres s’appliquent autant à “Debug” qu’à “Release”.

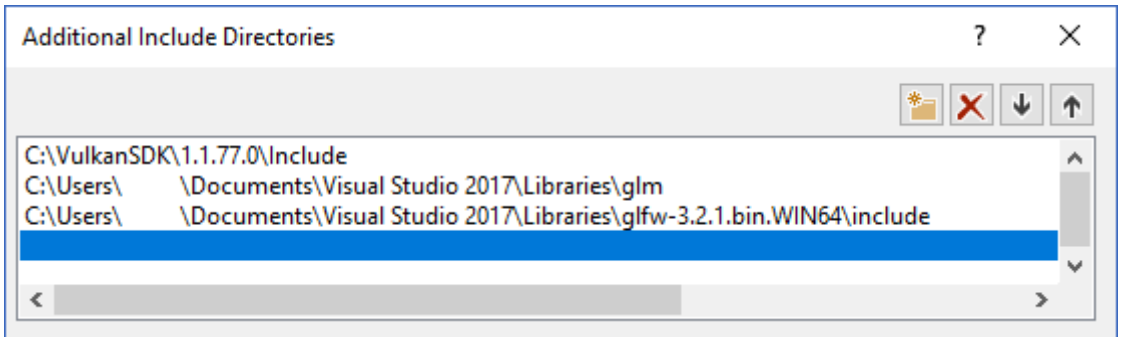




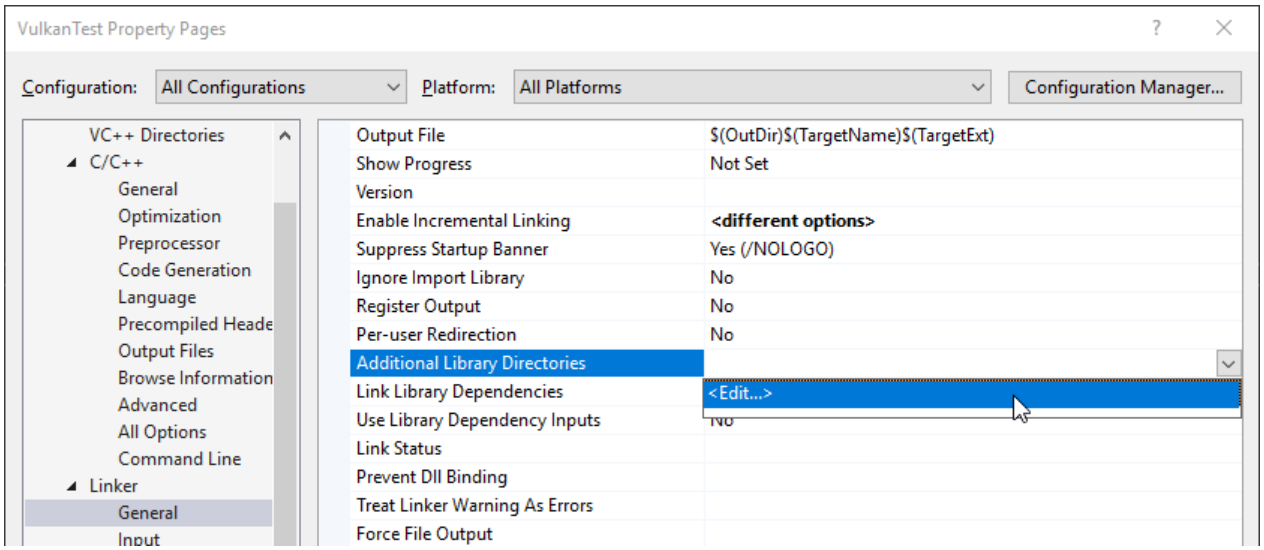
Allez à “C++” -> “General” -> “Additional Include Directories” et appuyez sur “<Edit…>” dans le menu déroulant.



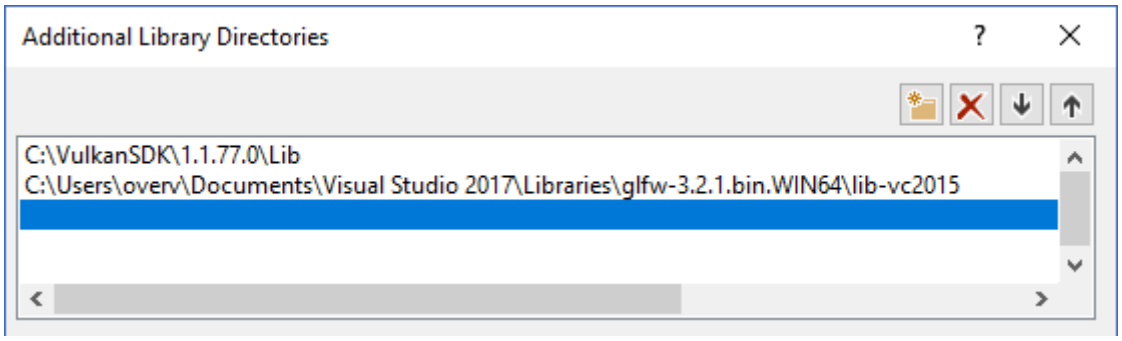
Ajoutez les dossiers pour les headers Vulkan, GLFW et GLM :



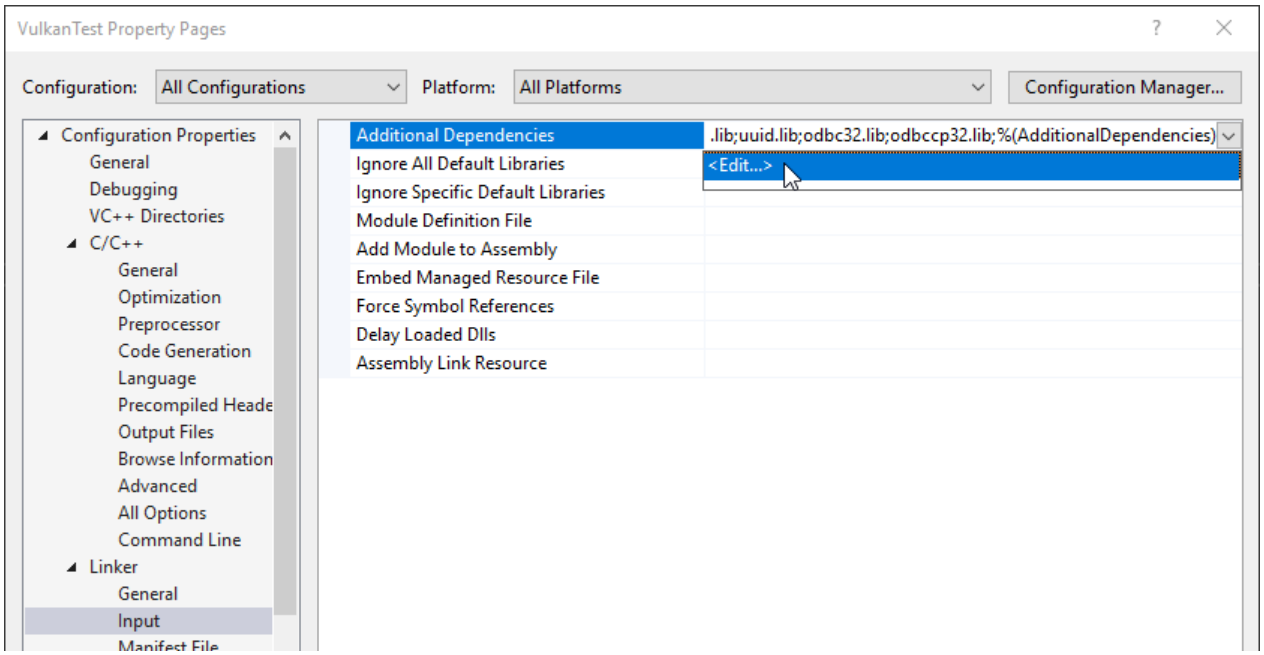
Ensuite, ouvrez l'éditeur pour les dossiers des librairies sous "Linker" -> "General" :



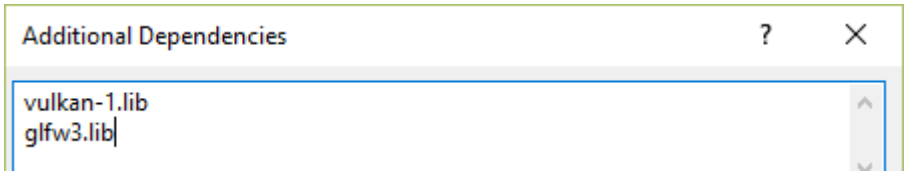
Et ajoutez les emplacements des fichiers objets pour Vulkan et GLFW :



Allez à "Linker" -> "Input" et appuyez sur "<Edit…>" dans le menu déroulant "Additional Dependencies" :

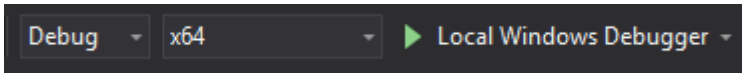


Entrez les noms des fichiers objets GLFW et Vulkan :

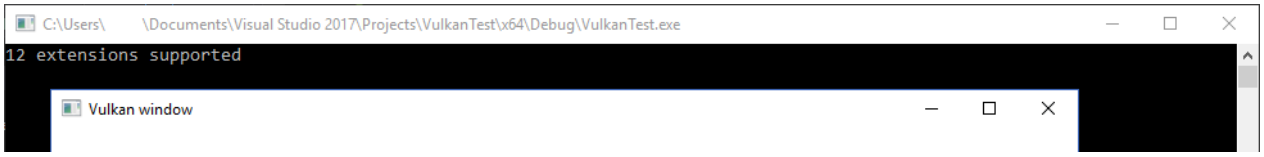


Vous pouvez enfin fermer le dialogue des propriétés. Si vous avez tout fait correctement vous ne devriez plus voir d'erreur dans votre code.

Assurez-vous finalement 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 prêts à vous lancer avec Vulkan!

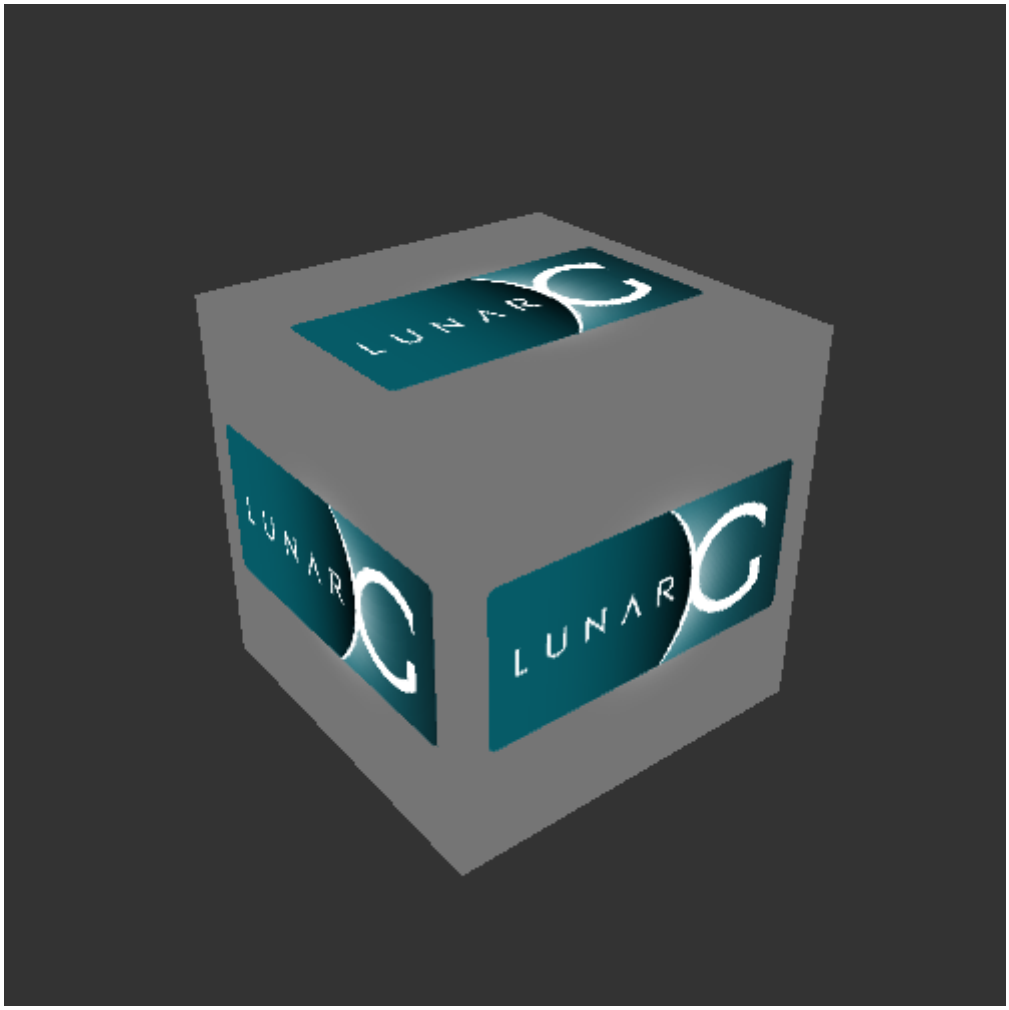
Linux

Ces instructions sont conçues pour les utilisateurs d'Ubuntu et Fedora, mais vous devriez pouvoir suivre ces instructions depuis une autre distribution si vous adaptez les commandes "apt" ou "dnf" à votre propre gestionnaire de packages. Il vous faut un compilateur qui supporte C++17 (GCC 7+ ou Clang 5+). Vous aurez également besoin de make.

Paquets Vulkan

Les composants les plus importants pour le développement d'applications Vulkan sous Linux sont le loader Vulkan, les validation layers et quelques utilitaires pour tester que votre machine est bien en état de faire fonctionner une application Vulkan: * `sudo apt install vulkan-tools` ou `sudo dnf install vulkan-tools`: Les utilitaires en ligne de commande, plus précisément `vulkaninfo` et `vkcube`. Lancez ceux-ci pour vérifier le bon fonctionnement de votre machine pour Vulkan. * `sudo apt install libvulkan-dev` ou `sudo dnf install vulkan-headers vulkan-loader-devel`: Installe le loader Vulkan. Il sert à aller chercher les fonctions auprès du driver de votre GPU au runtime, de la même façon que GLEW le fait pour OpenGL - si vous êtes familier avec ceci. * `sudo apt install vulkan-validationlayers-dev` ou `sudo dnf install mesa-vulkan-drivers vulkan-validation-layers-devel`: Installe les layers de validation standards. Ceux-ci sont cruciaux pour débugger vos applications Vulkan, et nous en reparlerons dans un prochain chapitre.

Si l'installation est un succès, vous devriez être prêt pour la partie Vulkan. N'oubliez pas de lancer `vkcube` et assurez-vous de voir la fenêtre suivante:



GLFW

Comme dit précédemment, Vulkan ignore la plateforme sur laquelle il opère, et n'inclut pas d'outil de création de fenêtre où afficher les résultats de notre travail. Pour bien exploiter les possibilités cross-platform de Vulkan, nous utiliserons la librairie GLFW pour créer une fenêtre sur Windows, Linux ou MacOS indifféremment. Il existe d'autres librairies telles que SDL, mais GLFW à l'avantage d'abstraire d'autres aspects spécifiques à

la plateforme requis par Vulkan.

Nous allons installer GLFW à l'aide de la commande suivante:

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

ou

```
sudo dnf install glfw-devel
```

GLM

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

Cette librairie contenue intégralement dans les headers peut être installée depuis le package "libglm-dev" ou "glm-devel" :

```
sudo apt install libglm-dev
```

ou

```
sudo dnf install glm-devel
```

Compilateur de shader

Nous avons tout ce qu'il nous faut, excepté un programme qui compile le code GLSL lisible par un humain en bytecode.

Deux compilateurs de shader populaires sont `glslangValidator` de Khronos et `glslc` de Google. Ce dernier a l'avantage d'être proche de GCC et Clang à l'usage,. Pour cette raison, nous l'utiliserons: Ubuntu, téléchargez les exécutables non officiels et copiez `glslc` dans votre répertoire `/usr/local/bin`. Notez que vous aurez certainement besoin d'utiliser `sudo` en fonctions de vos permissions. Fedora, utilise `sudo dnf install glslc`. Pour tester, lancez `glslc` depuis le répertoire de votre choix et il devrait se plaindre qu'il n'a reçu aucun shader à compiler de votre part:

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

Nous couvrirons l'usage de `glslc` plus en détails dans le chapitre des modules shaders

Préparation d'un fichier 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.

Ajoutez maintenant le code suivant à votre fichier. Ne cherchez pas à en comprendre les tenants et aboutissants, il sert juste à s'assurer que tout compile correctement et qu'une application Vulkan fonctionne. 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>
```

```cpp

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 bases de makefile, dont les variables et les 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 `CFLAGS`, qui spécifiera les arguments pour la compilation :

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

Nous utiliserons du C++ moderne (`-std=c++17`) et compilerons avec le paramètre d'optimisation `-O2`. Vous pouvez le retirer pour compiler nos programmes plus rapidement, mais n'oubliez pas de le remettre pour compiler des exécutables prêts à être distribués.

Définissez de manière analogue `LDFLAGS` :

```
LDFLAGS = -lglfw -lvulkan -ldl -lpthread -lX11 -lXxf86vm -lXrandr -lXi
```

Le premier flag correspond à GLFW, `-lvulkan` correspond au loader dynamique des fonctions Vulkan. Le reste des options correspondent à des bibliothèques systèmes liés à la gestion des fenêtres et aux threads nécessaire pour le bon fonctionnement de GLFW.

Spécifier les commandes 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 le fichier fonctionne en le sauvegardant et en exécutant make depuis un terminal ouvert dans le dossier le contenant. 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
```

Lancer `make test` doit vous afficher le programme sans erreur, listant le nombre d'extensions disponible pour Vulkan. L'application devrait retourner le code de retour 0 (succès) quand vous fermez la fenêtre vide. Vous devriez désormais avoir un makefile ressemblant à ceci :

```
CFLAGS = -std=c++17 -O2
LDFLAGS = -lglfw -lvulkan -ldl -lpthread -lX11 -lXxf86vm -lXrandr -lXi

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 exemple pour vos futurs projets Vulkan. Faites-en une copie, changez le nom du projet pour quelque chose comme `HelloTriangle` et retirez tout le code contenu dans `main.cpp`.

Bravo, vous êtes fin prêts à vous lancer avec Vulkan!

MacOS

Ces instructions partent du principe que vous utilisez Xcode et le gestionnaire de packages Homebrew. Vous aurez besoin de MacOS 10.11 minimum, et votre ordinateur doit supporter l'API Metal.

Le SDK Vulkan

Le SDK est le composant le plus important pour programmer une application avec Vulkan. Il inclue headers, validations layers, outils de débogage et un loader dynamique pour les fonctions Vulkan. Le loader cherche les fonctions dans le driver pendant l'exécution, comme GLEW pour OpenGL, si cela vous parle.

Le SDK se télécharge sur le site de LunarG en utilisant les boutons en bas de page. Vous n'avez pas besoin de créer de compte, mais il permet d'accéder 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 à l'API Vulkan en appels au framework Metal d'Apple. Vous pouvez ainsi exploiter pleinement les possibilités de cet API automatiquement.

Une fois téléchargé, extrayez-en le contenu où vous le souhaitez. Dans le dossier extrait, il devrait y avoir un sous-dossier "Applications" comportant des exécutables lançant des démos du SDK. Lancez "vkcube" pour vérifier que vous obtenez ceci :

GLFW

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

Nous utiliserons le gestionnaire de package Homebrew pour installer GLFW. Le support 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 :

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

GLM

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

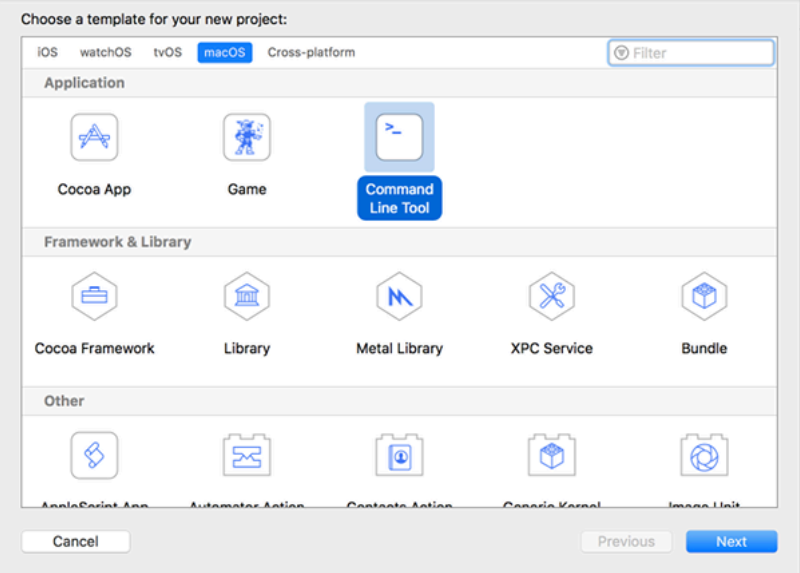
Cette librairie est intégralement codée dans les headers et se télécharge avec le package "glm" :

```
1 brew install glm
```

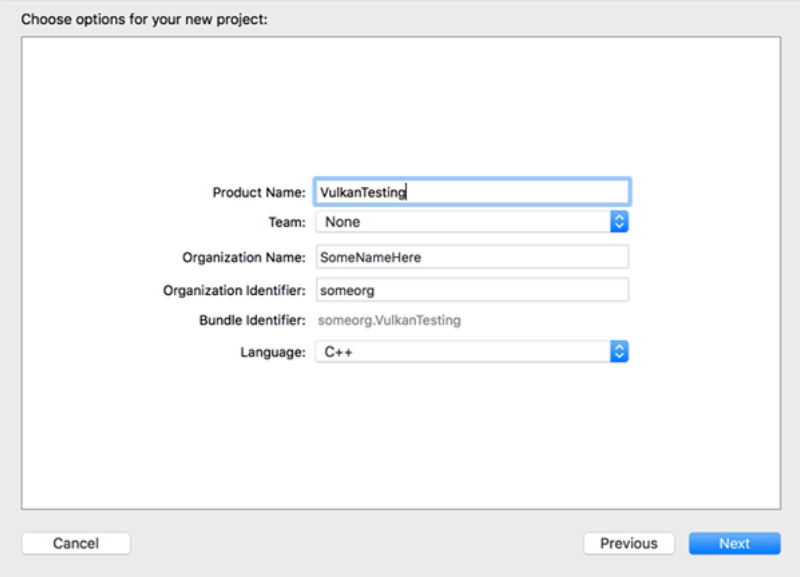
Préparation de Xcode

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 “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++” pour “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
```

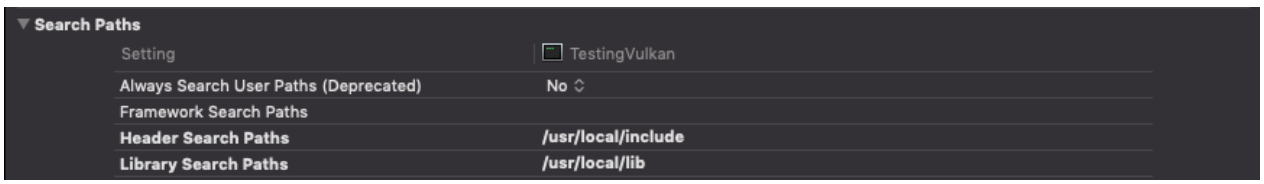
```cpp
#include <glm/vec4.hpp>
#include <glm/mat4x4.hpp>

#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 tout ce que le code fait, dans la mesure où il se contente d'appeler quelques fonctions de l'API pour s'assurer que tout fonctionne. Nous verrons toutes ces fonctions en détail plus tard.

Xcode devrait déjà vous afficher des erreurs comme le fait que des librairies soient introuvables. Nous allons maintenant les faire disparaître. Sélectionnez votre projet sur le menu *Project Navigator*. Ouvrez *Build Settings* puis :

- Trouvez le champ **Header Search Paths** et ajoutez "/usr/local/include" (c'est ici que Homebrew installe les headers) et "vulkansdk/macOS/include" pour le SDK.
- Trouvez le champ **Library Search Paths** et ajoutez "/usr/local/lib" (même raison pour les librairies) et "vulkansdk/macOS/lib".

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

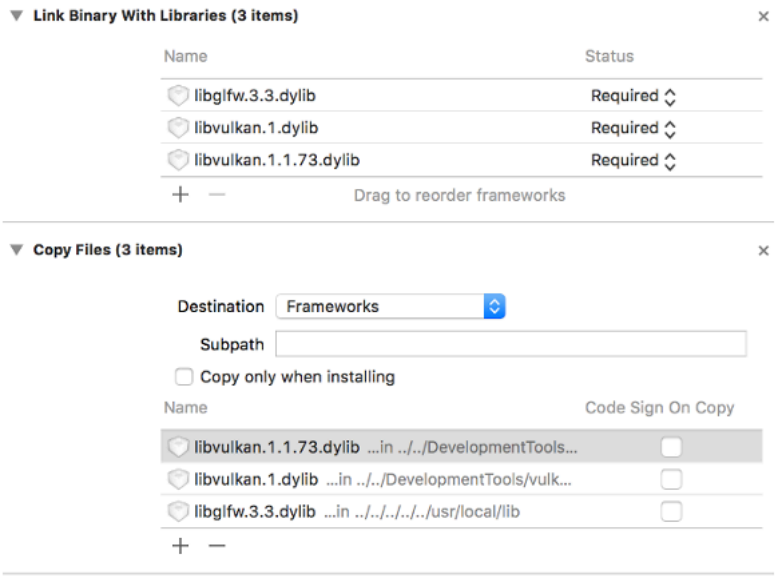


Maintenant, dans le menu *Build Phases*, ajoutez les frameworks "glfw3" et "vulkan" dans **Link Binary With Librairies**. Pour nous simplifier les choses, nous allons ajouter les librairies dynamiques directement dans le projet (référez-vous à la documentation de ces librairies si vous voulez les lier 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. Glissez ce fichier jusqu'à la barre des "Linked Frameworks and Librairies" dans Xcode.
- Pour Vulkan, rendez-vous dans "vulkansdk/macOS/lib" et répétez l'opération pour "libvulkan.1.dylib" et "libvulkan.1.x.xx .dylib" avec les x correspondant à la version du SDK que vous avez téléchargé.

Maintenant que vous avez ajouté ces librairies, remplissez le champ `Destination` avec "Frameworks" dans **Copy Files**, supprimez le sous-chemin et décochez "Copy only when installing". Cliquez sur le "+" et ajoutez-y les trois mêmes frameworks.

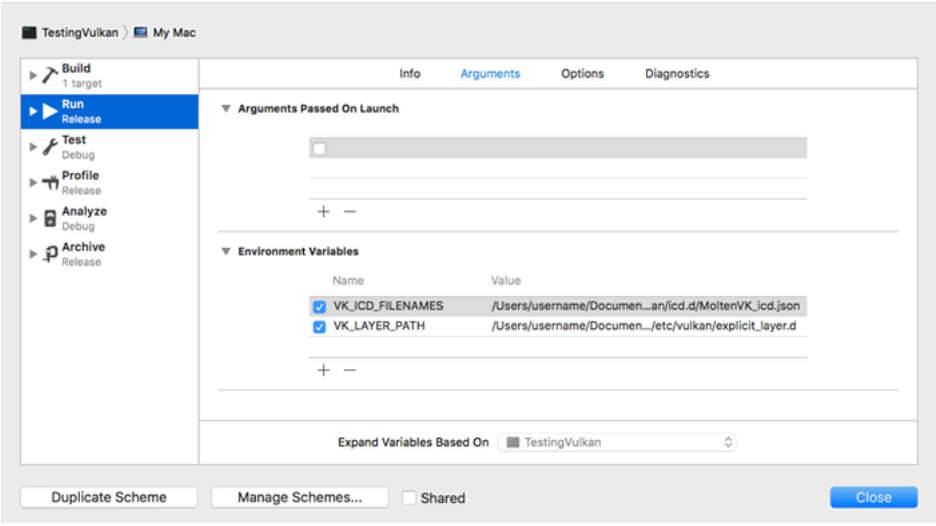
Votre configuration Xcode devrait ressembler à cela :



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 la liste `Arguments` ajoutez les deux variables suivantes :

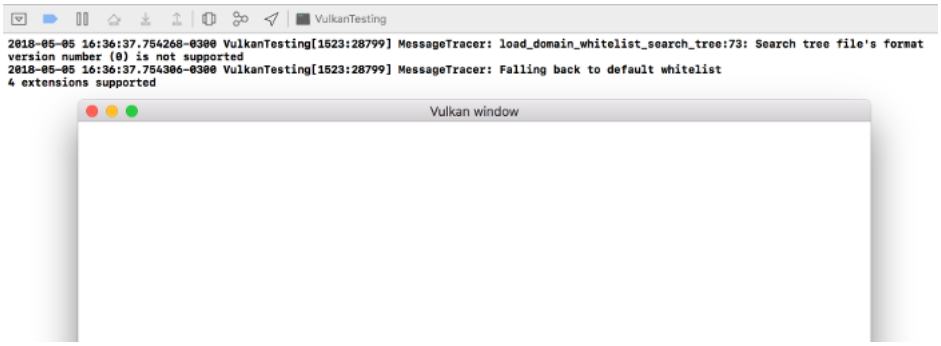
- VK_ICD_FILENAMES = `vulkansdk/macOS/share/vulkan/icd.d/MoltenVK_icd.json`
- VK_LAYER_PATH = `vulkansdk/macOS/share/vulkan/explicit_layer.d`

Vous avez normalement ceci :



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

avoir ceci :



Si vous obtenez `0 extensions supported`, il y a un problème avec la configuration de Vulkan sur votre système. Les autres données proviennent de librairies, et dépendent de votre configuration.

Vous êtes maintenant prêts à vous lancer avec Vulkan!.

Dessiner un triangle

Mise en place

Code de base

Structure générale

Dans le chapitre précédent nous avons créé un projet Vulkan avec une configuration solide et nous l'avons testé. Nous recommençons ici à partir du code suivant :

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

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

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

private:
    void initVulkan() {

    }

    void mainLoop() {

    }

    void cleanup() {

    }
};

int main() {
    HelloTriangleApplication app;

    try {
        app.run();
```

```
35      } catch (const std::exception& e) {
36          std::cerr << e.what() << std::endl;`
37          return EXIT_FAILURE;
38      }
39
40      return EXIT_SUCCESS;
41  }
```

Nous incluons d'abord le header Vulkan du SDK, qui fournit les fonctions, les structures et les énumérations. `<stdexcept>` et `<iostream>` nous permettront de reporter et de traiter les erreurs. Le header `<functional>` nous servira pour l'écriture d'une lambda dans la section sur la gestion des ressources. `<cstdlib>` nous fournit les macros `EXIT_FAILURE` et `EXIT_SUCCESS` (optionnelles).

Le programme est écrit à l'intérieur d'une classe, dans laquelle seront stockés les objets Vulkan. Nous avons également une fonction pour la création de chacun de ces objets. Une fois toute l'initialisation réalisée, nous entrons dans la boucle principale, qui attend que nous fermions la fenêtre pour quitter le programme, après avoir libéré grâce à la fonction cleanup toutes les ressources que nous avons allouées .

Si nous rencontrons une quelconque erreur lors de l'exécution nous lèverons une `std::runtime_error` comportant un message descriptif, qui sera affiché sur le terminal depuis la fonction `main`. Afin de s'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 introduiront une nouvelle fonction appelée dans `initVulkan` et un nouvel objet Vulkan qui sera justement créé par cette fonction. Il sera soit détruit dans `cleanup`, soit libéré automatiquement.

Gestion des ressources

De la même façon qu'une quelconque ressource explicitement allouée par `new` doit être explicitement libérée par `delete`, nous devrons explicitement détruire quasiment toutes les ressources Vulkan que nous allouerons. Il est possible d'exploiter des fonctionnalités du C++ pour s'acquitter automatiquement de cela. Ces possibilités sont localisées dans `<memory>` si vous désirez les utiliser. Cependant nous resterons explicites pour toutes les opérations dans ce tutoriel, car la puissance de Vulkan réside en particulier dans la clareté de l'expression de la volonté du programmeur. De plus, cela nous permettra de bien comprendre la durée de vie de chacun des objets.

Après avoir suivi ce tutoriel vous pourrez parfaitement implémenter une gestion automatique des ressources en spécialisant `std::shared_ptr` par exemple. L'utilisation du RAII à votre avantage est toujours recommandé en C++ pour de gros programmes Vulkan, mais il est quand même bon de commencer par connaître les détails de l'implémentation.

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 qu'il n'est plus utilisé où que ce soit, il faut le détruire en utilisant les fonctions `vkDestroyXXX` ou `vkFreeXXX`, respectivement. Les paramètres de ces fonctions varient sauf pour l'un d'entre eux : `pAllocator`. Ce paramètre optionnel vous permet de spécifier un callback sur un allocateur de mémoire. Nous n'utiliserons jamais ce paramètre et indiquerons donc toujours `nullptr`.

Intégrer GLFW

Vulkan marche très bien sans fenêtre si vous voulez l'utiliser pour du rendu sans écran (offscreen rendering en Anglais), mais c'est tout de même plus intéressant d'afficher quelque chose! Remplacez d'abord la ligne `#include <vulkan/vulkan.h>` par :

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

GLFW va alors automatiquement inclure ses propres définitions des fonctions Vulkan et vous fournir le header Vulkan. Ajoutez une fonction `initWindow` et appelez-la depuis `run` avant les autres appels. 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()`, ce qui initialise la librairie. 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);
```

Dans la mesure où redimensionner une fenêtre n'est pas chose aisée avec Vulkan, 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 ceci au-dessus de la définition de la classe `HelloTriangleApplication` :

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

et remplacé 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 s'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'évènements :

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

Ce code est relativement simple. GLFW récupère tous les évènements 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 requête pour la fermeture de la fenêtre récupérée, nous devons détruire toutes les ressources allouées et quitter GLFW. Voici notre première version de la fonction `cleanup` :

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

    glfwTerminate();
}
```

Si vous lancez l'application, vous devriez voir une fenêtre appelée "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++

Instance

Création d'une instance

La première chose à faire avec Vulkan est son initialisation au travers d'une *instance*. Cette instance relie l'application à l'API. Pour la créer vous devrez donner quelques informations au driver.

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

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

Ajoutez ensuite un membre donnée représentant cette instance :

```
private:
VkInstance instance;
```

Pour créer l'instance, nous allons d'abord remplir une première structure avec des informations sur notre application. Ces données sont optionnelles, mais elles peuvent fournir des informations utiles au driver pour optimiser ou diagnostiquer les erreurs lors de l'exécution, par exemple en reconnaissant le nom d'un moteur graphique. 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 membre `sType`. Cela permet d'indiquer la version exacte de la structure que nous voulons utiliser : il y aura dans le futur des extensions à celles-ci. Pour simplifier leur implémentation, les utiliser ne nécessitera que de changer le type `VK_STRUCTURE_TYPE_XXX` en `VK_STRUCTURE_TYPE_XXX_2` (ou plus de 2) et de fournir une structure complémentaire à l'aide du pointeur `pNext`. Nous n'utiliserons aucune extension, et donnerons donc toujours `nullptr` à `pNext`.

Avec Vulkan, nous rencontrerons souvent (TRÈS souvent) des structures à remplir pour passer les informations à Vulkan. Nous allons maintenant remplir le reste de la structure permettant la création de l'instance. Celle-ci n'est pas optionnelle. Elle permet d'informer le driver des extensions et des validation layers que nous utiliserons, et ceci de manière globale. Globale siginifie ici que ces données ne serons pas spécifiques à un périphérique. Nous verrons la signification de cela dans les chapitres suivants.

```
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 dont nous aurons besoin. Comme nous l'avons vu dans l'introduction, Vulkan ne connaît pas la plateforme sur laquelle il travaille, et nous aurons donc besoin d'extensions pour utiliser des interfaces avec le gestionnaire de fenêtre. GLFW possède une fonction très pratique qui nous donne la liste des extensions dont nous aurons besoin pour afficher nos résultats. Remplissez donc la structure de ces données :

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

glfwExtensions = glfwGetRequiredInstanceExtensions(&glfwExtensionCount);

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

Les deux derniers membres de la structure indiquent les validations layers 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 notre première instance. Nous pouvons enfin appeler `vkCreateInstance` :

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

Comme vous le reverrez, l'appel à une fonction pour la création d'un objet Vulkan a le prototype suivant :

- Pointeur sur une structure contenant l'information pour la création
- Pointeur sur une fonction d'allocation que nous laisserons toujours `nullptr`
- Pointeur sur une variable stockant une référence au nouvel objet

Si tout s'est bien passé, la référence à l'instance devrait être contenue dans le membre `VkInstance`. Quasiment toutes les fonctions Vulkan retournent une valeur de type VkResult, pouvant être soit `VK_SUCCESS` soit un code d'erreur. Afin de vérifier si la création de l'instance s'est bien déroulée nous pouvons placer l'appel dans un `if` :

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

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

Vérification du support des extensions

Si vous regardez la documentation pour `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 manque. Ce serait logique pour des fonctionnalités cruciales comme l'affichage, mais pas dans le cas d'extensions optionnelles.

La fonction `vkEnumerateInstanceExtensionProperties` permet de récupérer la totalité des extensions supportées par le système avant la création de l'instance. Elle demande un pointeur vers une variable stockant le nombre d'extensions supportées et un tableau où stocker des informations sur chacune des extensions. Elle possède également un paramètre optionnel permettant de filtrer les résultats pour une validation layer spécifique. Nous l'ignorerons pour le moment.

Pour allouer un tableau contenant les détails des extensions nous devons déjà connaître le nombre de ces extensions. Vous pouvez ne demander que cette information en laissant le premier paramètre `nullptr` :

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

Nous utiliserons souvent cette méthode. Allouez maintenant un tableau pour stocker les détails des extensions (incluez) :

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

Nous pouvons désormais accéder aux détails des extensions :

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

Chacune des structure `VkExtensionProperties` contient le nom et la version maximale supportée de l'extension. Nous pouvons les afficher à l'aide d'une boucle `for` toute simple (`\t` représente une tabulation) :

```
std::cout << "Extensions disponibles :\n";

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

Vous pouvez ajouter ce code dans la fonction `createInstance` si vous voulez indiquer des informations à propos du support Vulkan sur la machine. Petit challenge : programmez une fonction vérifiant si les extensions dont vous avez besoin (en particulier celles indiquées par GLFW) sont disponibles.

Libération des ressources

L'instance contenue dans `VkInstance` ne doit être détruite qu'à la fin du programme. Nous la détruirons dans la fonction `cleanup` grâce à la fonction `vkDestroyInstance` :

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

    glfwDestroyWindow(window);

    glfwTerminate();
}
```

Les paramètres de cette fonction sont évidents. Nous y retrouvons le paramètre pour un désallocateur que nous laissons `nullptr`. Toutes les ressources que nous allouerons à partir du prochain chapitre devront être libérées avant la libération de l'instance.

Avant d'avancer dans les notions plus complexes, créons un moyen de déboger notre programme avec les validations layers..

Code C++

Validation layers

Que sont les validation layers?

L'API Vulkan est conçue pour limiter au maximum le travail du driver. Par conséquent il n'y a aucun traitement d'erreur par défaut. Une erreur aussi simple que se tromper dans la valeur d'une énumération ou passer un pointeur nul comme argument non optionnel résultent en un crash. Dans la mesure où Vulkan nous demande d'être complètement explicite, il est facile d'utiliser une fonctionnalité optionnelle et d'oublier de mettre en place l'utilisation de l'extension à laquelle elle appartient, par exemple.

Cependant de telles vérifications peuvent être ajoutées à l'API. Vulkan possède un système élégant appelé validation layers. Ce sont des composants optionnels s'insérant dans les appels des fonctions Vulkan pour y ajouter des opérations. Voici un exemple d'opérations qu'elles réalisent :

- 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 toutes les informations sur les appels à l'aide de la sortie standard
- Suivre les appels Vulkan pour créer une analyse dynamique de l'exécution du programme

Voici ce à quoi une fonction de diagnostic pourrait ressembler :

```
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 validation layers peuvent être combinées à loisir pour fournir toutes les fonctionnalités de débogage nécessaires. Vous pouvez même activer les validations layers lors du développement et les désactiver lors du déploiement sans aucun problème, sans aucune répercussion sur les performances et même sur l'exécutable!

Vulkan ne fournit aucune validation layer, mais nous en avons dans le SDK de LunarG. Elles sont complètement open source, vous pouvez donc voir quelles erreurs elles suivent et contribuer à leur développement. Les utiliser est la meilleure manière d'éviter que votre application fonctionne grâce à un comportement spécifique à un driver.

Les validations layers ne sont utilisables que si elles sont installées sur la machine. Il faut le SDK installé et mis en place pour qu'elles fonctionnent.

Il a existé deux formes de validation layers : les layers spécifiques à l'instance et celles spécifiques au physical device (gpu). Elles ne vérifiaient ainsi respectivement que les appels aux fonctions d'ordre global et les appels aux fonctions spécifiques au GPU. Les layers spécifiques du GPU sont désormais dépréciées. Les autres portent désormais sur tous les appels. Cependant la spécification recommande encore que nous activions les validations layers au niveau du logical device, car cela est requis par certaines implémentations. Nous nous contenterons de spécifier les mêmes layers pour le logical device que pour le physical device, que nous verrons plus tard.

Utiliser les validation layers

Nous allons maintenant activer les validations layers fournies par le SDK de LunarG. Comme les extensions, nous devons indiquer leurs nom. Au lieu de devoir spécifier les noms de chacune d'entre elles, nous pouvons les activer à l'aide d'un nom générique : `VK_LAYER_KHRONOS_validation`.

Mais ajoutons d'abord deux variables spécifiant les layers à activer et si le programme doit en effet les activer. J'ai choisi d'effectuer ce choix selon si le programme est compilé en mode debug ou non. La macro `NDEBUG` fait partie du standard c++ et correspond au second cas.

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

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

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

Ajoutons une nouvelle fonction `checkValidationLayerSupport`, qui devra vérifier si les layers que nous voulons utiliser sont disponibles. Listez d'abord les validation layers disponibles à l'aide de la fonction `vkEnumerateInstanceLayerProperties`. Elle s'utilise de la même façon que `vkEnumerateInstanceExtensionProperties`.

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

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

    return false;
}
```

Vérifiez que toutes les layers de `validationLayers` sont présentes dans la liste des layers disponibles. 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 `createInstance` :

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

    ...
}
```

Lancez maintenant le programme en mode debug et assurez-vous qu'il fonctionne. Si vous obtenez une erreur, référez-vous à la FAQ.

Modifions enfin la structure `VkCreateInstanceInfo` pour inclure les noms des validation layers à utiliser lorsqu'elles sont activées :

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

Si l'appel à la fonction `checkValidationLayerSupport` est un succès, `vkCreateInstance` ne devrait jamais retourner `VK_ERROR_LAYER_NOT_PRESENT`, mais exécutez tout de même le programme pour être sûr que d'autres erreurs n'apparaissent pas.

Fonction de rappel des erreurs

Les validation layers affichent leur messages dans la console par défaut, mais on peut s'occuper de l'affichage nous-même en fournissant un callback explicite dans notre programme. Ceci nous permet également de choisir quels types de message 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 debug messenger avec un callback en utilisant l'extension `VK_EXT_debug_utils`.

Créons d'abord une fonction `getRequiredExtensions`. Elle nous fournira les extensions nécessaires selon que nous activons les validation layers 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. Remarquez 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 `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 dans la mesure où les validation layers devraient impliquer son support, mais sait-on jamais.

Intéressons-nous maintenant à la fonction de rappel. Ajoutez la fonction statique `debugCallback` à votre classe avec le prototype `PFN_vkDebugUtilsMessengerCallbackEXT`. `VKAPI_ATTR` et `VKAPI_CALL` assurent une compatibilité avec tous les compilateurs.

```
static VKAPI_ATTR VkBool32 VKAPI_CALL debugCallback(
    VkDebugUtilsMessageSeverityFlagBitsEXT messageSeverity,
    VkDebugUtilsMessageTypeFlagsEXT messageType,
    const VkDebugUtilsMessengerCallbackDataEXT* pCallbackData,
    void* pUserData) {

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

    return VK_FALSE;
}
```

Le premier paramètre indique la sévérité du message, et peut prendre les valeurs suivantes :

- `VK_DEBUG_UTILS_MESSAGE_SEVERITY_VERBOSE_BIT_EXT`: Message de suivi des appels
- `VK_DEBUG_UTILS_MESSAGE_SEVERITY_INFO_BIT_EXT`: Message d'information (allocation d'une ressource…)
- `VK_DEBUG_UTILS_MESSAGE_SEVERITY_WARNING_BIT_EXT`: Message relevant un comportement qui n'est pas un bug mais plutôt une imperfection involontaire
- `VK_DEBUG_UTILS_MESSAGE_SEVERITY_ERROR_BIT_EXT`: Message relevant un comportement invalide pouvant mener à un crash

Les valeurs de cette énumération on été conçues de telle sorte qu'il est possible de les comparer pour vérifier la sévérité d'un message, par exemple :

```
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 quelconque est survenu, sans lien avec les performances ou la spécification
- `VK_DEBUG_UTILS_MESSAGE_TYPE_VALIDATION_BIT_EXT` : Une violation de la spécification ou une potentielle erreur est survenue
- `VK_DEBUG_UTILS_MESSAGE_TYPE_PERFORMANCE_BIT_EXT` : Utilisation potentiellement 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 type C terminée par le caractère nul `\0`
- `pObjects`: Un tableau d'objets Vulkan liés au message
- `objectCount`: Le nombre d'objets dans le tableau précédent

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

La fonction de rappel 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 globalement utilisée que pour tester les validation layers elles-mêmes, nous retournerons donc invariablement `VK_FALSE`.

Il ne nous reste plus qu'à fournir notre fonction à Vulkan. Surprenamment, même le messager de débogage se gère à travers une référence de type `VkDebugUtilsMessengerEXT`, que nous devrons explicitement créer et détruire. Une telle fonction de rappel est appelée *messager*, et vous pouvez en posséder autant que vous le désirez. Ajoutez un membre donnée pour le messager sous 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 :

```
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 les niveaux de sévérité pour lesquels la fonction de rappel 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 bugs tout en éliminant la verbose.

De manière similaire, le champ `messageType` vous permet de filtrer les types de message 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.

Le champ `pfnUserCallback` indique le pointeur vers la fonction de rappel.

Vous pouvez optionnellement ajouter un pointeur sur une donnée de votre choix grâce au champ `pUserData`. Le pointeur fait partie des paramètres de la fonction de rappel.

Notez qu'il existe de nombreuses autres manières de configurer des messagers auprès des validation layers, mais nous avons ici 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

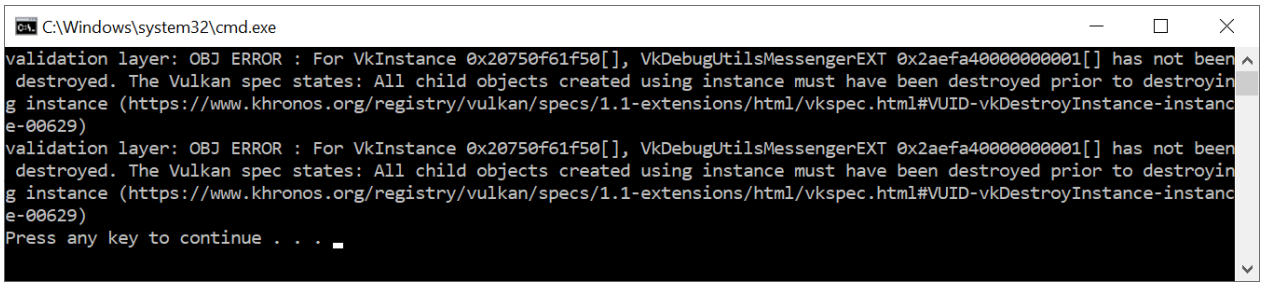
non incluse par GLFW. Nous devons donc gérer son activation nous-mêmes. Nous utiliserons la fonction `vkGetInstancePorcAddr` pous en récupérer un pointeur. Nous allons créer notre propre fonction - servant de proxy - pour abstraire cela. Je l'ai ajoutée au-dessus de la définition de la classe `HelloTriangleApplication`.

```
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 le messager 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 de rappel est spécifique de l'instance et des validation layers, nous devons donc passer l'instance en premier argument. Lancez le programme et vérifiez qu'il fonctionne. Vous devriez avoir le résultat suivant :



qui indique déjà un bug dans notre application! En effet l'objet `VkDebugUtilsMessengerEXT` doit être libéré explicitement à l'aide de la fonction `vkDestroyDebugUtilsMessagerEXT`. De même qu'avec `vkCreateDebugUtilsMessangerEXT` nous devons charger dynamiquement cette fonction. Notez qu'il est normal que le message s'affiche plusieurs fois; il y a plusieurs validation layers, et dans certains cas leurs domaines d'expertise se recoupent.

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

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

Nous pouvons maintenant l'appeler dans notre fonction `cleanup` :

```
void cleanup() {
```

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

    vkDestroyInstance(instance, nullptr);

    glfwDestroyWindow(window);

    glfwTerminate();
}
```

Si vous exécutez le programme maintenant, vous devriez constater que le message n'apparait plus. Si vous voulez voir quel fonction a lancé un appel au messager, vous pouvez insérer un point d'arrêt dans la fonction de rappel.

Déboguer la création et la destruction de l'instance

Même si nous avons mis en place un système de débogage très efficace, deux fonctions passent sous le radar. Comme il est nécessaire d'avoir une instance pour appeler `vkCreateDebugUtilsMessengerEXT`, la création de l'instance n'est pas couverte par le messager. Le même problème apparait avec la destruction de l'instance.

En lisant la documentation on voit qu'il existe un messager spécifiquement créé pour ces deux fonctions. Il suffit de passer un pointeur vers une instance de `VkDebugUtilsMessengerCreateInfoEXT` au membre `pNext` de `VkInstanceCreateInfo`. Plaçons le remplissage de la structure de création du messager dans une fonction :

```
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("failed to set up debug messenger!");
    }
}
```

Nous pouvons réutiliser cette fonction dans `createInstance` :

```
void createInstance() {
    ...

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

```
7
8       ...
9
10      VkDebugUtilsMessengerCreateInfoEXT debugCreateInfo{};
11      if (enableValidationLayers) {
12          createInfo.enabledLayerCount = static_cast<uint32_t>(validationLayers.size());
13          createInfo.ppEnabledLayerNames = validationLayers.data();
14          populateDebugMessengerCreateInfo(debugCreateInfo);
15          createInfo.pNext = (VkDebugUtilsMessengerCreateInfoEXT*) &debugCreateInfo;
16      } else {
17          createInfo.enabledLayerCount = 0;
18
19          createInfo.pNext = nullptr;
20      }
21
22      if (vkCreateInstance(&createInfo, nullptr, &instance) != VK_SUCCESS) {
23          throw std::runtime_error("failed to create instance!");
24      }
25  }
```

La variable `debugCreateInfo` est en-dehors du `if` pour qu'elle ne soit pas détruite avant l'appel à `vkCreateInstance`. La structure fournie à la création de l'instance à travers la structure `VkInstanceCreateInfo` mènera à la création d'un messager spécifique aux deux fonctions qui sera détruit automatiquement à la destruction de l'instance.

Configuration

Les validation layers peuvent être paramétrées de nombreuses autres manières que juste avec les informations que nous avons fournies dans la structure `VkDebugUtilsMessangerCreateInfoEXT`. Ouvrez le SDK Vulkan et rendez-vous dans le dossier `Config`. Vous y trouverez le fichier `vk_layer_settings.txt` qui vous expliquera comment configurer les validation layers.

Pour configurer les layers pour votre propre application, copiez le fichier dans les dossiers `Debug` et/ou `Release`, puis suivez les instructions pour obtenir le comportement que vous souhaitez. Cependant, pour le reste du tutoriel, je partirai du principe que vous les avez laissées avec leur comportement par défaut.

Tout au long du tutoriel je laisserai de petites erreurs intentionnelles pour vous montrer à quel point les validation layers sont pratiques, et à quel point vous devez comprendre tout ce que vous faites avec Vulkan. Il est maintenant temps de s'intéresser aux devices Vulkan dans le système.

Code C++

Physical devices et queue families

Sélection d'un physical device

La librairie étant initialisée à travers `VkInstance`, nous pouvons dès à présent chercher et sélectionner une carte graphique (physical device) dans le système qui supporte les fonctionnalitées dont nous aurons besoin. Nous pouvons en fait en sélectionner autant que nous voulons et travailler avec chacune d'entre elles, mais nous n'en utiliserons qu'une dans ce tutoriel pour des raisons de simplicité.

Ajoutez la fonction `pickPhysicalDevice` et appelez la depuis `initVulkan` :

```
1   void initVulkan() {
2       createInstance();
3       setupDebugMessenger();
4       pickPhysicalDevice();
5   }
```

```
6
7 void pickPhysicalDevice() {
8
9 }
```

Nous stockerons le physical device que nous aurons sélectionnée dans un nouveau membre donnée de la classe, et celui-ci sera du type `VkPhysicalDevice`. Cette référence sera implicitement détruit avec l'instance, nous n'avons donc rien à ajouter à la fonction `cleanup`.

```
1 VkPhysicalDevice physicalDevice = VK_NULL_HANDLE;
```

Lister les physical devices est un procédé très similaire à lister les extensions. Comme d'habitude, on commence par en lister le nombre.

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

Si aucun physical device ne supporte Vulkan, il est inutile de continuer l'exécution.

```
1 if (deviceCount == 0) {
2     throw std::runtime_error("aucune carte graphique ne supporte Vulkan!");
3 }
```

Nous pouvons ensuite allouer un tableau contenant toutes les références aux `VkPhysicalDevice`.

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

Nous devons maintenant évaluer chacun des gpus et vérifier qu'ils conviennent pour ce que nous voudrons en faire, car toutes les cartes graphiques n'ont pas été crées égales. Voici une nouvelle fonction qui fera le travail de sélection :

```
1 bool isDeviceSuitable(VkPhysicalDevice device) {
2     return true;
3 }
```

Nous allons dans cette fonction vérifier que le physical device respecte nos conditions.

```
1 for (const auto& device : devices) {
2     if (isDeviceSuitable(device)) {
3         physicalDevice = device;
4         break;
5     }
6 }
7
8 if (physicalDevice == VK_NULL_HANDLE) {
9     throw std::runtime_error("aucun GPU ne peut exécuter ce programme!");
10 }
```

La section suivante introduira les premières contraintes que devront remplir les physical devices. Au fur et à mesure que nous utiliserons de nouvelles fonctionnalités, nous les ajouterons dans cette fonction.

Vérification des fonctionnalités de base

Pour évaluer la compatibilité d'un physical device nous devons d'abord nous informer sur ses capacités. Des propriétés basiques comme le nom, le type et les versions de Vulkan supportées peuvent être obtenues en appelant `vkGetPhysicalDeviceProperties`.

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

Le support des fonctionnalités optionnelles telles que les textures compressées, les floats de 64 bits et le multi viewport rendering (pour la VR) s'obtiennent avec `vkGetPhysicalDeviceFeatures` :

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

De nombreux autres détails intéressants peuvent être requis, mais nous en remparlerons dans les prochains chapitres.

Voyons un premier exemple. Considérons que notre application a besoin d'une carte graphique dédiée supportant les geometry shaders. Notre fonction `isDeviceSuitable` ressemblerait alors 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 choisir le premier physical device nous convenant, nous pourrions attribuer un score à chacun d'entre eux et utiliser celui dont le score est le plus élevé. Vous pourriez ainsi préférer une carte graphique dédiée, mais utiliser un GPU intégré au CPU si le système n'en détecte aucune. Vous pourriez implémenter ce concept comme cela :

```
#include <map>

...

void pickPhysicalDevice() {
    ...

    // L'utilisation d'une map permet de les trier automatiquement de manière ascendante
    std::multimap<int, VkPhysicalDevice> candidates;

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

    // Voyons si la meilleure possède les fonctionnalités dont nous ne pouvons nous passer
    if (candidates.rbegin()->first > 0) {
        physicalDevice = candidates.rbegin()->second;
    } else {
        throw std::runtime_error("aucun GPU ne peut executer ce programme!");
    }
}

int rateDeviceSuitability(VkPhysicalDevice device) {
    ...

```

```
    int score = 0;

    // Les carte graphiques dédiées ont un énorme avantage en terme de performances
    if (deviceProperties.deviceType == VK_PHYSICAL_DEVICE_TYPE_DISCRETE_GPU) {
        score += 1000;
    }

    // La taille maximale des textures affecte leur qualité
    score += deviceProperties.limits.maxImageDimension2D;

    // L'application (fictive) 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, mais faites-le si vous voulez, à titre d'entrainement. Vous pourriez également vous contenter d'afficher les noms des cartes graphiques et laisser l'utilisateur choisir.

Nous ne faisons que commencer donc nous prendrons la première carte supportant Vulkan :

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

Nous discuterons de la première fonctionnalité qui nous sera nécessaire dans la section suivante.

Familles de queues (queue families)

Il a été évoqué que chaque opération avec Vulkan, de l'affichage jusqu'au chargement d'une texture, s'effectue en ajoutant une commande à une queue. Il existe différentes queues appartenant à différents types de *queue families*. De plus chaque queue family ne permet que certaines commandes. Il se peut par exemple qu'une queue ne traite que les commandes de calcul et qu'une autre ne supporte que les commandes d'allocation de mémoire.

Nous devons analyser quelles queue families existent sur le système et lesquelles correspondent aux commandes que nous souhaitons utiliser. Nous allons donc créer la fonction `findQueueFamilies` dans laquelle nous chercherons les commandes nous intéressant.

Nous allons chercher une queue qui supporte les commandes graphiques, la fonction pourrait ressembler à ça:

```
uint32_t findQueueFamilies(VkPhysicalDevice device) {
    // Code servant à trouver la famille de queue "graphique"
}
```

Mais dans un des prochains chapitres, nous allons avoir besoin d'une autre famille de queues, il est donc plus intéressant de s'y préparer dès maintenant en empactant plusieurs indices dans une structure:

```
struct QueueFamilyIndices {
    uint32_t graphicsFamily;
};

QueueFamilyIndices findQueueFamilies(VkPhysicalDevice device) {
    QueueFamilyIndices indices;
    // Code pour trouver les indices de familles à ajouter à la structure
```

```
    return indices
}
```

Que se passe-t-il si une famille n'est pas disponible ? On pourrait lancer une exception dans `findQueueFamilies`, mais cette fonction n'est pas vraiment le bon endroit pour prendre des decisions concernant le choix du bon Device. Par exemple, on pourrait *préférer* des Devices avec une queue de transfert dédiée, sans toutefois le requérir. Par conséquent nous avons besoin d'indiquer si une certaine famille de queues à été trouvé.

Ce n'est pas très pratique d'utiliser une valeur magique pour indiquer la non-existence d'une famille, comme n'importe quelle valeur de `uint32_t` peut théoriquement être une valeur valide d'index de famille, incluant `0`. 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; // faux

graphicsFamily = 0;

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

`std::optional` est un wrapper qui ne contient aucune valeur tant que vous ne lui en assignez pas une. Vous pouvez, quelque soit le moment, lui demander si il contient une valeur ou non en appelant sa fonction membre `has_value()`. On peut donc changer le code comme suit:

```
#include <optional>

...

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

QueueFamilyIndices findQueueFamilies(VkPhysicalDevice device) {
    QueueFamilyIndices indices;

    // Assigne l'index aux familles qui ont pu être trouvées

    return indices;
}
```

On peut maintenant commencer à implémenter `findQueueFamilies`:

```
QueueFamilyIndices findQueueFamily(VkPhysicalDevice) {
    QueueFamilyIndices indices;

    ...

    return indices;
}
```

Récupérer la liste des queue families 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 des informations sur la queue family, et en particulier le type d'opérations qu'elle supporte et le nombre de queues que l'on peut instancier à partir de cette famille. Nous devons trouver au moins une queue 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 pouvons maintenant utiliser cette fonction dans `isDeviceSuitable` pour s'assurer que le physical device peut recevoir les commandes que nous voulons lui envoyer :

```
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 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();
}
```

On peut également utiliser ceci pour sortir plus tôt de `findQueueFamilies`:

```
for (const auto& queueFamily : queueFamilies) {
    ...

    if (indices.isComplete()) {
        break;
    }
```

```
7
8       i++;
9   }
```

Bien, c'est tout ce dont nous aurons besoin pour choisir le bon physical device! La prochaine étape est de créer un logical device pour créer une interface avec la carte.

Code C++

Logical device et queues

Introduction

La sélection d'un physical device faite, nous devons générer un *logical device* pour servir d'interface. Le processus de sa création est similaire à celui de l'instance : nous devons décrire ce dont nous aurons besoin. Nous devons également spécifier les queues dont nous aurons besoin. Vous pouvez également créer plusieurs logical devices à partir d'un physical device si vous en avez besoin.

Commencez par ajouter un nouveau membre donnée pour stocker la référence au logical device.

```
1 VkDevice device;
```

Ajoutez ensuite une fonction `createLogicalDevice` et appelez-la depuis `initVulkan`.

```
1  void initVulkan() {
2      createInstance();
3      setupDebugMessenger();
4      pickPhysicalDevice();
5      createLogicalDevice();
6  }
7
8  void createLogicalDevice() {
9
10 }
```

Spécifier les queues à créer

La création d'un logical device requiert encore que nous remplissions des informations dans des structures. La première de ces structures s'appelle `VkDeviceQueueCreateInfo`. Elle indique le nombre de queues que nous désirons pour chaque queue family. Pour le moment nous n'avons besoin que d'une queue originaire d'une unique queue family : la première avec un support pour les graphismes.

```
1 QueueFamilyIndices indices = findQueueFamilies(physicalDevice);
2
3 VkDeviceQueueCreateInfo queueCreateInfo{};
4 queueCreateInfo.sType = VK_STRUCTURE_TYPE_DEVICE_QUEUE_CREATE_INFO;
5 queueCreateInfo.queueFamilyIndex = indices.graphicsFamily.value();
6 queueCreateInfo.queueCount = 1;
```

Actuellement les drivers ne vous permettent que de créer un petit nombre de queues pour chacune des familles, et vous n'avez en effet pas besoin de plus. Vous pouvez très bien créer les commandes (command buffers) depuis plusieurs threads et les soumettre à la queue d'un coup sur le thread principal, et ce sans perte de performance.

Vulkan permet d'assigner des niveaux de priorité aux queues à l'aide de floats compris entre `0.0` et `1.0`. Vous pouvez ainsi influencer l'exécution des command buffers. Il est nécessaire d'indiquer une priorité même lorsqu'une seule queue est présente :

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

Spécifier les fonctionnalités utilisées

Les prochaines informations à fournir sont les fonctionnalités du physical device que nous souhaitons utiliser. Ce sont celles dont nous avons vérifié la présence avec `vkGetPhysicalDeviceFeatures` dans le chapitre précédent. Nous n'avons besoin de rien de spécial pour l'instant, nous pouvons donc nous contenter de créer la structure et de tout laisser à `VK_FALSE`, valeur par défaut. Nous reviendrons sur cette structure quand nous ferons des choses plus intéressantes avec Vulkan.

```
VkPhysicalDeviceFeatures deviceFeatures{};
```

Créer le logical device

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

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

Référencez d'abord les structures sur la création des queues et sur les fonctionnalités utilisées :

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

createInfo.pEnabledFeatures = &deviceFeatures;
```

Le reste ressemble à la structure `VkInstanceCreateInfo`. Nous devons spécifier les extensions spécifiques de la carte graphique et les validation layers.

Un exemple d'extension spécifique au GPU est `VK_KHR_swapchain`. Celle-ci vous permet de présenter à l'écran les images sur lesquels votre programme a effectué un rendu. Il est en effet possible que certains GPU ne possèdent pas cette capacité, par exemple parce qu'ils ne supportent que les compute shaders. Nous reviendrons sur cette extension dans le chapitre dédié à la swap chain.

Comme dit dans le chapitre sur les validation layers, nous activerons les mêmes que celles que nous avons spécifiées lors de la création de l'instance. Nous n'avons pour l'instant besoin d'aucune validation layer en particulier. Notez que le standard ne fait plus la différence entre les extensions de l'instance et celles du device, au point que les paramètres `enabledLayerCount` et `ppEnabledLayerNames` seront probablement ignorés. Nous les remplissons quand même pour s'assurer de la bonne compatibilité 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;
}
```

C'est bon, nous pouvons maintenant instancier le logical device en appelant la fonction `vkCreateDevice`.

```
if (vkCreateDevice(physicalDevice, &createInfo, nullptr, &device) != VK_SUCCESS) {
    throw std::runtime_error("échec lors de la création d'un logical device!");
}
```

Les paramètres sont d'abord le physical device dont on souhaite extraire une interface, ensuite la structure contenant les informations, puis un pointeur optionnel pour l'allocation et enfin un pointeur sur la référence au logical device créé. Vérifions également si la création a été un succès ou non, comme lors de la création de l'instance.

Le logical device doit être explicitement détruit dans la fonction `cleanup` avant le physical device :

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

Les logical devices n'interagissent pas directement avec l'instance mais seulement avec le physical device, c'est pourquoi il n'y a pas de paramètre pour l'instance.

Récupérer des références aux queues

Les queue families sont automatiquement crées avec le logical device. Cependant nous n'avons aucune interface avec elles. Ajoutez un membre donnée pour stocker une référence à la queue family supportant les graphismes :

```
VkQueue graphicsQueue;
```

Les queues sont implicitement détruites avec le logical device, nous n'avons donc pas à nous en charger dans `cleanup`.

Nous pourrons ensuite récupérer des références à des queues avec la fonction `vkGetDeviceQueue`. Les paramètres en sont le logical device, la queue family, l'indice de la queue à récupérer et un pointeur où stocker la référence à la queue. Nous ne créons qu'une seule queue, nous écrirons donc `0` pour l'indice de la queue.

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

Avec le logical device et les queues nous allons maintenant pouvoir faire travailler la carte graphique! Dans le prochain chapitre nous mettrons en place les ressources nécessaires à la présentation des images à l'écran.

Code C++

Présentation

Window surface

Introduction

Vulkan ignore la plateforme sur laquelle il opère et ne peut donc pas directement établir d'interface avec le gestionnaire de fenêtres. Pour créer une interface permettant de présenter les rendus à l'écran, nous devons utiliser l'extension WSI (Window System Integration). Nous verrons dans ce chapitre l'extension `VK_KHR_surface`, l'une des extensions du WSI. Nous pourrons ainsi obtenir l'objet `VkSurfaceKHR`, qui est un type abstrait de surface sur lequel nous pourrons effectuer des rendus. Cette surface sera en lien avec la fenêtre que nous avons créée grâce à GLFW.

L'extension `VK_KHR_surface`, qui se charge au niveau de l'instance, a déjà été ajoutée, car elle fait partie des extensions retournées par la fonction `glfwGetRequiredInstanceExtensions`. Les autres fonctions WSI que nous verrons dans les prochains chapitres feront aussi partie des extensions retournées par cette fonction.

La surface de fenêtre doit être créée juste après l'instance car elle peut influencer le choix du physical device. Nous ne nous intéressons à ce sujet que maintenant car il fait partie du grand ensemble que nous abordons et qu'en parler plus tôt aurait été confus. Il est important de noter que cette surface est complètement optionnelle, et vous pouvez l'ignorer si vous voulez effectuer du rendu off-screen ou du calculus. Vulkan vous

offre ces possibilités sans vous demander de recourir à des astuces comme créer une fenêtre invisible, là où d'autres APIs le demandaient (cf OpenGL).

Création de la window surface

Commencez par ajouter un membre donnée `surface` sous le messager.

```
VkSurfaceKHR surface;
```

Bien que l'utilisation d'un objet `VkSurfaceKHR` soit indépendant de la plateforme, sa création ne l'est pas. Celle-ci requiert par exemple des références à `HWND` et à `HMODULE` sous Windows. C'est pourquoi il existe des extensions spécifiques à la plateforme, dont par exemple `VK_KHR_win32_surface` sous Windows, mais celles-ci sont aussi évaluées par GLFW et intégrées dans les extensions retournées par la fonction `glfwGetRequiredInstanceExtensions`.

Nous allons voir l'exemple de la création de la surface sous Windows, même si nous n'utiliserons pas cette méthode. Il est en effet contre-productif d'utiliser une librairie comme GLFW et un API comme Vulkan pour se retrouver à écrire du code spécifique à la plateforme. La fonction de GLFW `glfwCreateWindowSurface` permet de gérer les différences de plateforme. Cet exemple ne servira ainsi qu'à présenter le travail de bas niveau, dont la connaissance est toujours utile à une bonne utilisation de Vulkan.

Une window surface est un objet Vulkan comme un autre et nécessite donc de remplir une structure, ici `VkWin32SurfaceCreateInfoKHR`. 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 `HWND` de la fenêtre à l'aide de la fonction `glfwGetWin32Window`. La fonction `GetModuleHandle` fournit une référence au `HINSTANCE` du thread courant.

La surface peut maintenant être crée avec `vkCreateWin32SurfaceKHR`. Cette fonction prend en paramètre une instance, des détails sur la création de la surface, l'allocateur optionnel et la variable dans laquelle placer la référence. Bien que cette fonction fasse partie d'une extension, elle est si communément utilisée qu'elle est chargée par défaut par Vulkan. 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 la creation d'une window surface!");
}
```

Ce processus est similaire pour Linux, où la fonction `vkCreateXcbSurfaceKHR` requiert la fenêtre et une connexion à XCB comme paramètres pour X11.

La fonction `glfwCreateWindowSurface` implémente donc tout cela pour nous et utilise le code correspondant à la bonne plateforme. Nous devons maintenant l'intégrer à notre programme. Ajoutez la fonction `createSurface` et appelez-la dans `initVulkan` après la création de l'instance et du messager :

```
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 le tout très simple :

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

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

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

Détruisez bien la surface avant l'instance.

Demander le support pour la présentation

Bien que l'implémentation de Vulkan supporte le WSI, il est possible que d'autres éléments du système ne le supportent pas. Nous devons donc allonger `isDeviceSuitable` pour s'assurer que le logical device puisse présenter les rendus à la surface que nous avons créée. La présentation est spécifique aux queues families, ce qui signifie que nous devons en fait trouver une queue family supportant cette présentation.

Il est possible que les queue families supportant les commandes d'affichage et celles supportant les commandes de présentation ne soient pas les mêmes, nous devons donc considérer que ces deux queues sont différentes. En fait, les spécificités des queues families diffèrent majoritairement entre les vendeurs, et assez peu entre les modèles d'une même série. Nous devons donc étendre la structure `QueueFamilyIndices` :

```
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 queue family pouvant supporter les commandes de présentation. La fonction qui nous sera utile pour cela est `vkGetPhysicalDeviceSurfaceSupportKHR`. Elle possède quatre paramètres, le physical device, un indice de queue family, la surface et un booléen. Appelez-la depuis la même boucle que pour `VK_QUEUE_GRAPHICS_BIT` :

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

Vérifiez simplement la valeur du booléen et stockez la queue dans la structure si elle est intéressante :

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

Il est très probable que ces deux queue families soient en fait les mêmes, mais nous les traiterons comme si elles étaient différentes pour une meilleure compatibilité. Vous pouvez cependant ajouter un alorithme préférant des queues combinées pour améliorer légèrement les performances.

Création de la queue de présentation (presentation queue)

Il nous reste à modifier la création du logical device pour extraire de celui-ci la référence à une presentation queue `VkQueue`. Ajoutez un membre donnée pour cette référence :

```
VkQueue presentQueue;
```

Nous avons besoin de plusieurs structures `VkDeviceQueueCreateInfo`, une pour chaque queue family. Une manière de gérer ces structures est d'utiliser un `set` contenant tous les indices des queues et un `vector` pour les structures :

```
#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);
}
```

Puis modifiez `VkDeviceCreateInfo` pour qu'il pointe sur le contenu du vector :

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

Si les queues sont les mêmes, nous n'avons besoin de les indiquer qu'une seule fois, ce dont le set s'assure. Ajoutez enfin un appel pour récupérer les queue families :

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

Si les queues sont les mêmes, les variables contenant les références contiennent la même valeur. Dans le prochain chapitre nous nous intéresserons aux swap chain, et verrons comment elle permet de présenter les rendus à l'écran.

Code C++

Swap chain

Vulkan ne possède pas de concept comme le framebuffer par défaut, et nous devons donc créer une infrastructure qui contiendra les buffers sur lesquels nous effectuerons les rendus avant de les présenter à l'écran. Cette infrastructure s'appelle *swap chain* sur Vulkan et doit être créée explicitement. La swap chain est essentiellement une file d'attente d'images attendant 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 de la présentation dépendent du paramétrage de la swap chain. Cependant, l'intérêt principal de la swap chain est de synchroniser la présentation avec le rafraîchissement de l'écran.

Vérification du support de la swap chain

Toutes les cartes graphiques ne sont pas capables de présenter directement les images à l'écran, et ce pour différentes raisons. Ce pourrait être car elles sont destinées à être utilisées dans un serveur et n'ont aucune sortie vidéo. De plus, dans la mesure où la présentation est très dépendante du gestionnaire de fenêtres et de la surface, la swap chain ne fait pas partie de Vulkan "core". Il faudra donc utiliser des extensions, dont `VK_KHR_swapchain`.

Pour cela nous allons modifier `isDeviceSuitable` pour qu'elle vérifie 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 header Vulkan intègre la macro `VK_KHR_SWAPCHAIN_EXTENSION_NAME` qui permet d'éviter une faute de frappe. Toutes les extensions ont leur macro.

Déclarez d'abord une liste d'extensions nécessaires au physical device, comme nous l'avons fait pour les validation layers :

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

Créez ensuite une nouvelle fonction appelée `checkDeviceExtensionSupport` et appelez-la depuis `isDeviceSuitable` comme vérification supplémentaire :

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

    bool extensionsSupported = checkDeviceExtensionSupport(device);

    return indices.isComplete() && extensionsSupported;
}

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

Énumérez les extensions et vérifiez si toutes les extensions requises en font partie.

```
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 une collection de strings pour représenter les extensions requises en attente de confirmation. Nous pouvons ainsi facilement les éliminer en énumérant la séquence. Vous pouvez également utiliser des boucles imbriquées comme dans `checkValidationLayerSupport`, car la perte en performance n'est pas capitale dans cette phase de chargement. Lancez le code et vérifiez que votre carte graphique est capable de gérer une swap chain. Normalement la disponibilité de la queue de présentation implique que l'extension de la swap chain est supportée. Mais soyons tout de mêmes explicites pour cela aussi.

Activation des extensions du device

L'utilisation de la swap chain nécessite l'extension `VK_KHR_swapchain`. Son activation ne requiert qu'un léger changement à la structure de création du logical device :

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

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

Récupérer des détails à propos du support de la swap chain

Vérifier que la swap chain est disponible n'est pas suffisant. Nous devons vérifier qu'elle est compatible avec notre surface de fenêtre. La création de la swap chain nécessite un nombre important de paramètres, et nous devons récupérer encore d'autres détails pour pouvoir continuer.

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

- Possibilités basiques de la surface (nombre min/max d'images dans la swap chain, hauteur/largeur min/max des images)
- Format de la surface (format des pixels, palette de couleur)
- Mode de présentation disponibles

Nous utiliserons une structure comme celle dans `findQueueFamilies` pour contenir ces détails une fois qu'ils auront été récupérés. 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 basiques de la texture. 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 physical device et la surface de fenêtre soient passées en paramètres, car elle en extrait ces 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 de texture supportés. Comme c'est une liste de structure, cette acquisition suit le rituel des deux étapes :

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

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

Finalement, récupérer les modes de présentation supportés suit le même principe et utilise `vkGetPhysicalDeviceSurfacePres` :

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

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

Tous les détails sont dans des structures, donc étendons `isDeviceSuitable` une fois de plus et utilisons cette fonction pour vérifier que le support de la swap chain nous correspond. Nous ne demanderons que des choses très simples dans ce tutoriel.

```
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;
```

Choisir les bons paramètres pour la swap chain

Si la fonction `swapChainAdequate` retourne `true` le support de la swap chain est assuré. Il existe cependant encore plusieurs modes ayant chacun leur intérêt. Nous allons maintenant écrire quelques fonctions qui détermineront les bons paramètres pour obtenir la swap chain la plus efficace possible. Il y a trois types de paramètres à déterminer :

- Format de la surface (profondeur de la couleur)
- Modes de présentation (conditions de "l'échange" des images avec l'écran)
- Swap extent (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 nous restera de mieux.

Format de la surface La fonction utilisée pour déterminer ce paramètre commence ainsi. Nous lui passerons en argument le membre donnée `formats` de la structure `SwapChainSupportDetails`.

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

}
```

Chaque `VkSurfaceFormatKHR` contient les données `format` et `colorSpace`. Le `format` indique les canaux de couleur disponibles et les types qui contiennent les valeurs des gradients. Par exemple `VK_FORMAT_B8G8R8A8_SRGB` signifie que nous stockons les canaux de couleur R, G, B et A dans cet ordre et en entiers non signés de 8 bits. `colorSpace` permet de vérifier que le sRGB est supporté en utilisant le champ de bits `VK_COLOR_SPACE_SRGB_NONLINEAR_KHR`.

Pour l'espace de couleur nous utiliserons sRGB si possible, car il en résulte un rendu plus réaliste. Le format le plus commun est `VK_FORMAT_B8G8R8A8_SRGB`.

Itérons dans la liste et voyons si le meilleur 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 aussi nous pourrions trier les combinaisons disponibles, mais pour rester simple nous prendrons 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];
}
```

Mode de présentation Le mode de présentation est clairement le paramètre le plus important pour la swap chain, car il touche aux conditions d'affichage des 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, et l'écran récupère l'image en haut de la pile quand il est rafraîchi, alors que le programme insère ses nouvelles images 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 rafraichi s'appelle l'*intervalle de rafraîchissement vertical* (vertical blank).
- `VK_PRESENT_MODE_FIFO_RELAXED_KHR` : ce mode ne diffère du précédent que si l'application est en retard et que la queue est vide pendant le vertical blank. Au lieu d'attendre le prochain vertical blank, une image arrivant dans la file d'attente sera immédiatement transmise à l'écran.

- `VK_PRESENT_MODE_MAILBOX_KHR` : ce mode est une autre variation du second mode. Au lieu de bloquer l'application quand le file d'attente est pleine, les images présentes dans la queue sont simplement remplacées par de nouvelles. Ce mode peut être utilisé pour implémenter le triple buffering, qui vous permet d'éliminer le tearing tout en réduisant le temps de latence entre le rendu et l'affichage qu'une file d'attente implique.

Seul `VK_PRESENT_MODE_FIFO_KHR` est toujours disponible. Nous aurons donc encore à écrire une fonction pour réaliser un choix, car le mode que nous choisirons préférentiellement est `VK_PRESENT_MODE_MAILBOX_KHR` :

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

Je pense que le triple buffering est un très bon compromis. Vérifions si ce mode est disponible :

```
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;
}
```

Le swap extent Il ne nous reste plus qu'une propriété, pour laquelle nous allons créer encore une autre fonction :

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

}
```

Le swap extent donne la résolution des images dans la swap chain et correspond quasiment toujours 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 notre résolution à celle de la fenêtre fournie par le membre `currentExtent`. Cependant certains gestionnaires de fenêtres nous permettent de choisir une résolution différente, ce que nous pouvons détecter grâce aux membres `width` et `height` qui sont alors égaux à la plus grande valeur d'un `uint32_t`. Dans ce cas nous choisirons la résolution correspondant le mieux à la taille de la fenêtre, dans les bornes de `minImageExtent` et `maxImageExtent`.

```
#include <cstdint> // uint32_t
#include <limits> // std::numeric_limits
#include <algorithm> // std::clamp

...

VkExtent2D chooseSwapExtent(const VkSurfaceCapabilitiesKHR& capabilities) {
    if (capabilities.currentExtent.width != std::numeric_limits<uint32_t>::max()) {
        return capabilities.currentExtent;
    } else {
        VkExtent2D actualExtent = {WIDTH, HEIGHT};

        actualExtent.width = std::clamp(actualExtent.width,
            capabilities.minImageExtent.width, capabilities.maxImageExtent.width);
```

```
        actualExtent.height = std::clamp(actualExtent.height,
            capabilities.minImageExtent.height, capabilities.maxImageExtent.height);

        return actualExtent;
    }
}
```

La fonction `clamp` est utilisée ici pour limiter les valeurs `WIDTH` et `HEIGHT` entre le minimum et le maximum supportés par l'implémentation.

Création de la swap chain

Maintenant que nous avons toutes ces fonctions nous pouvons enfin acquérir toutes les informations nécessaires à la création d'une swap chain.

Créez une fonction `createSwapChain`. Elle commence par récupérer les résultats des fonctions précédentes. Appelez-la depuis `initVulkan` après la création du logical device.

```
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);
}
```

Il nous reste une dernière chose à faire : déterminer le nombre d'images dans la swap chain. L'implémentation décide d'un minimum nécessaire pour fonctionner :

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

Se contenter du minimum pose cependant un problème. Il est possible que le driver fasse attendre notre programme car il n'a pas fini certaines opérations, ce que nous ne voulons pas. Il est recommandé d'utiliser au moins une image de plus que ce 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 autre que la mémoire.

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

Comme la tradition le veut avec Vulkan, la création d'une swap chain nécessite de remplir une grande structure. Elle commence de manière familière :

```
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, les détails sur les images de la swap chain doivent être fournis :

```
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;
```

Le membre `imageArrayLayers` indique le nombre de couches que chaque image possède. Ce sera toujours `1` sauf si vous développez une application stéréoscopique 3D. Le champ de bits `imageUsage` spécifie le type d'opérations que nous appliquerons aux images de la swap chain. Dans ce tutoriel nous effectuerons un rendu directement sur les images, nous les utiliserons donc comme *color attachement*. Vous voudrez peut-être travailler sur une image séparée pour pouvoir appliquer des effets en post-processing. Dans ce cas vous devrez utiliser une valeur comme `VK_IMAGE_USAGE_TRANSFER_DST_BIT` à la place et utiliser une opération de transfert de mémoire pour placer le résultat final dans une 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
}
```

Nous devons ensuite indiquer comment les images de la swap chain seront utilisées dans le cas où plusieurs queues seront à l'origine d'opérations. Cela sera le cas si la queue des graphismes n'est pas la même que la queue de présentation. Nous devrons alors dessiner avec la graphics queue puis fournir l'image à la presentation queue. Il existe deux manières de gérer les images accédées par plusieurs queues :

- `VK_SHARING_MODE_EXCLUSIVE` : une image n'est accesible que par une queue à la fois et sa gestion doit être explicitement transférée à une autre queue pour pouvoir être utilisée. Cette option offre le maximum de performances.
- `VK_SHARING_MODE_CONCURRENT` : les images peuvent être simplement utilisées par différentes queue families.

Si nous avons deux 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 vous demande de spécifier à l'avance les queues qui partageront les images en utilisant les paramètres `queueFamilyIndexCount` et `pQueueFamilyIndices`. Si les graphics queue et presentation queue 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 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 `supportedTransforms` dans `capabilities`), comme par exemple 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. 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 activé avec `VK_TRUE` alors les couleurs des pixels masqués par d'autres fenêtres seront ignorées. Si vous n'avez pas un besoin particulier de lire ces informations, 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 mal adaptée pendant que votre application tourne, par exemple parce que la fenêtre a été redimensionnée. Dans ce cas la swap chain doit être intégralement recréée et une référence à l'ancienne swap chain doit être fournie. C'est un sujet compliqué que nous aborderons dans un chapitre futur. Pour le moment, considérons que nous ne devrons jamais créer qu'une swap chain.

Ajoutez un membre donnée pour stocker l'objet `VkSwapchainKHR` :

```
VkSwapchainKHR swapChain;
```

Créer la swap chain ne se résume plus qu'à appeler `vkCreateSwapchainKHR` :

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

Les paramètres sont le logical device, la structure contenant les informations, l'allocateur optionnel et la variable contenant la référence à la swap chain. Cet objet devra être explicitement détruit à l'aide de la fonction `vkDestroySwapchainKHR` avant de détruire le logical device :

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

Lancez maintenant l'application et contemplez la création de la swap chain! Si vous obtenez une erreur de violation d'accès dans `vkCreateSwapchainKHR` ou voyez un message comme `Failed to find 'vkGetInstanceProcAddress' in layer SteamOverlayVulkanLayer.ddl`, allez voir la FAQ à propos de la layer Steam.

Essayez de retirer la ligne `createInfo.imageExtent = extent;` avec les validation layers actives. Vous verrez que l'une d'entre elles verra 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 vkGetPhys
icalDeviceSurfaceCapabilitiesKHR().
```

Récupérer les images de la swap chain

La swap chain est enfin créée. Il nous faut maintenant récupérer les références aux `VkImage` dans la swap chain. Nous les utiliserons pour l'affichage et dans les chapitres suivants. Ajoutez un membre donnée pour les stocker :

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

Ces images ont été créées par l'implémentation avec la swap chain et elles seront automatiquement supprimées avec la destruction de la swap chain, nous n'aurons donc rien à rajouter dans la fonction `cleanup`.

Ajoutons le code nécessaire à la récupération des références à la fin de `createSwapChain`, juste après l'appel à `vkCreateSwapchainKHR`. Comme notre logique n'a au final informé Vulkan que d'un minimum pour le nombre d'images dans la swap chain, nous devons nous enquérir du nombre d'images avant de redimensionner le conteneur.

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

Une dernière chose : gardez dans des variables le format et le nombre d'images 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 travailler et qui peuvent être présentées pour être affichées. Dans le prochain chapitre nous verrons comment utiliser ces images comme des cibles de rendu, puis nous verrons le pipeline graphique et les commandes d'affichage!

Code C++

Image views

Quelque soit la `VkImage` que nous voulons utiliser, dont celles de la swap chain, nous devons en créer une `VkImageView` pour la manipuler. Cette image view correspond assez litéralement à une vue dans l'image. Elle décrit l'accès à l'image et les parties de l'image à accéder. Par exemple elle indique si elle doit être traitée comme une texture 2D pour la profondeur sans aucun niveau de mipmapping.

Dans ce chapitre nous écrirons une fonction `createImageViews` pour créer une image view basique pour chacune des images dans la swap chain, pour que nous puissions les utiliser comme cibles de couleur.

Ajoutez d'abord un membre donnée pour y stocker une image view :

```
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 image views que nous créerons :

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

}
```

Créez ensuite la boucle qui parcourra toutes les images de la swap chain.

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

}
```

Les paramètres pour la création d'image views se spécifient dans la structure `VkImageViewCreateInfo`. Les deux premiers paramètres sont assez simples :

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

Les champs `viewType` et `format` indiquent la manière dont les images doivent être interprétées. Le paramètre `viewType` permet de traiter les images comme des textures 1D, 2D, 3D ou cube map.

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

Le champ `components` vous permet d'altérer les canaux de couleur. Par exemple, vous pouvez envoyer tous les canaux au canal rouge pour obtenir une texture monochrome. Vous pouvez aussi donner les valeurs constantes `0` ou `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;
```

Le champ `subresourceRange` décrit l'utilisation de l'image et indique quelles parties de l'image devraient être accédées. 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 devrez alors créer une swap chain avec plusieurs couches. Vous pourriez alors créer plusieurs image views pour chaque image. Elles représenteront ce qui sera affiché pour l'œil gauche et pour l'œil droit.

Créer l'image view ne se résume plus qu'à appeler `vkCreateImageView` :

```cpp
if (vkCreateImageView(device, &createInfo, nullptr, &swapChainImageViews[i]) != VK_SUCCESS) {
    throw std::runtime_error("échec de la création d'une image view!");
}
```

À la différence des images, nous avons créé les image views explicitement et devons donc les détruire de la même manière, ce que nous faisons à l'aide d'une boucle :

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

    ...
}
```

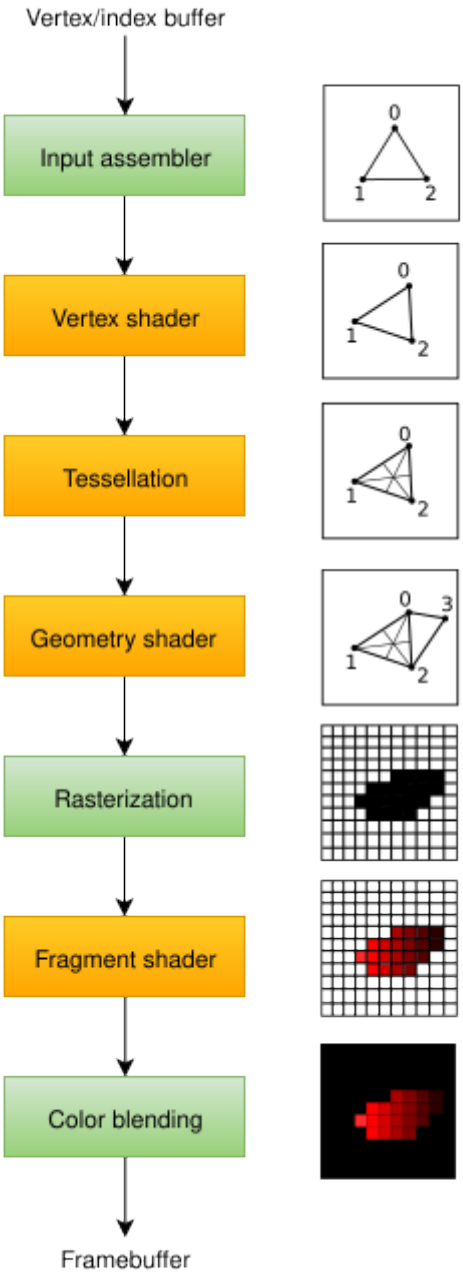
Une image view est suffisante pour commencer à utiliser une image comme une texture, mais pas pour que l'image soit utilisée comme cible d'affichage. Pour cela nous avons encore une étape, appelée framebuffer. Mais nous devons d'abord mettre en place le pipeline graphique.

Code C++

Pipeline graphique basique

Introduction

Dans les chapitres qui viennent nous allons configurer une pipeline graphique pour qu'elle affiche notre premier triangle. La pipeline graphique est l'ensemble des opérations qui prennent les vertices et les textures de vos éléments et les utilisent pour en faire des pixels sur les cibles d'affichage. Un résumé simplifié ressemble à ceci :



L'*input assembler* collecte les données des sommets à partir des buffers que vous avez mis en place, et peut aussi utiliser un *index buffer* pour répéter certains éléments sans avoir à stocker deux fois les mêmes données dans un buffer.

Le *vertex shader* est exécuté pour chaque sommet et leur applique en général des transformations pour que leurs coordonnées passent de l'espace du modèle (model space) à l'espace de l'écran (screen space). Il fournit ensuite des données à la suite de la pipeline.

Les *tesselation shaders* permettent de subdiviser la géométrie selon des règles paramétrables afin d'améliorer la qualité du rendu. Ce procédé est notamment utilisé pour que des surface comme les murs de briques ou les escaliers aient l'air moins plats lorsque l'on s'en approche.

Le *geometry shader* est invoqué pour chaque primitive (triangle, ligne, points…) et peut les détruire ou en créer de nouvelles, du même type ou non. Ce travail est similaire au tesselation shader tout en étant beaucoup plus flexible. Il n'est cependant pas beaucoup utilisé à cause de performances assez moyennes sur les cartes

graphiques (avec comme exception les GPUs intégrés d'Intel).

La *rasterization* transforme les primitives en *fragments*. Ce sont les pixels auxquels les primitives correspondent sur le framebuffer. Tout fragment en dehors de l'écran est abandonné. Les attributs sortant du vertex shader sont interpolés lorsqu'ils sont donnés aux étapes suivantes. Les fragments cachés par d'autres fragments sont aussi quasiment toujours éliminés grâce au test de profondeur (depth testing).

Le *fragment shader* est invoqué pour chaque fragment restant et détermine à quel(s) framebuffer(s) le fragment est envoyé, et quelles données y sont inscrites. Il réalise ce travail à l'aide des données interpolées émises par le vertex shader, ce qui inclut souvent des coordonnées de texture et des normales pour réaliser des calculs d'éclairage.

Le *color blending* applique des opérations pour mixer différents fragments correspondant à un même pixel sur le framebuffer. Les fragments peuvent remplacer les valeurs des autres, s'additionner ou se mélanger selon les paramètres de transparence (ou plus correctement de translucidité, en anglais translucency).

Les étapes écrites en vert sur le diagramme s'appellent *fixed-function stages* (étapes à fonction fixée). Il est possible de modifier des 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 pour y appliquer exactement ce que vous voulez. Cela vous permet par exemple d'utiliser les fragment shaders pour implémenter n'importe quoi, de l'utilisation de textures et d'éclairage jusqu'au *ray tracing*. Ces programmes tournent sur de nombreux coeurs simultanément pour y traiter de nombreuses données en parallèle.

Si vous avez utilisé d'anciens APIs comme OpenGL ou Direct3D, vous êtes habitués à pouvoir changer un quelconque paramètre de la pipeline à tout moment, avec des fonctions comme `glBlendFunc` ou `OMSSetBlendState`. Cela n'est plus possible avec Vulkan. La pipeline graphique y est quasiment fixée, et vous devrez en recréer une complètement si vous voulez changer de shader, y attacher différents framebuffers ou changer le color blending. Devoir créer une pipeline graphique pour chacune des combinaisons dont vous aurez besoin tout au long du programme représente un gros travail, mais permet au driver d'optimiser beaucoup mieux l'exécution des tâches car il sait à l'avance ce que la carte graphique aura à faire.

Certaines étapes programmables sont optionnelles selon ce que vous voulez faire. Par exemple la tesselation et le geometry shader peuvent être désactivés. Si vous n'êtes intéressé que par les valeurs de profondeur vous pouvez désactiver le fragment shader, ce qui est utile pour les shadow maps.

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

Créez la fonction `createGraphicsPipeline` et appelez-la depuis `initVulkan` après `createImageViews`. Nous travaillerons sur cette fonction dans les chapitres suivants.

```
1  void initVulkan() {
2      createInstance();
3      setupDebugMessenger();
4      createSurface();
5      pickPhysicalDevice();
6      createLogicalDevice();
7      createSwapChain();
8      createImageViews();
9      createGraphicsPipeline();
10 }
11
12 ...
13
14 void createGraphicsPipeline() {
```

```
15
16 }
```

Code C++

Modules shaders

À la différence d'anciens APIs, le code des shaders doit être fourni à Vulkan sous la forme de bytecode et non sous une forme facilement compréhensible par l'homme, comme GLSL ou HLSL. Ce format est appelé SPIR-V et est conçu pour fonctionner avec Vulkan et OpenCL (deux APIs de Khronos). Ce format peut servir à écrire du code éxécuté sur la carte graphique pour les graphismes et pour le calcul, mais nous nous concentrerons sur la pipeline graphique dans ce tutoriel.

L'avantage d'un tel format est que le compilateur spécifique de la carte graphique a beaucoup moins de travail d'interprétation. L'expérience a en effet montré qu'avec les syntaxes compréhensibles par l'homme, certains compilateurs étaient très laxistes par rapport à la spécification qui leur était fournie. Si vous écriviez du code complexe, il pouvait être accepté par l'un et pas par l'autre, ou pire s'éxécuter différemment. Avec le format de plus bas niveau qu'est SPIR-V, ces problèmes seront normalement éliminés.

Cela ne veut cependant pas dire que nous devrons écrire ces bytecodes à la main. Khronos fournit même un compilateur transformant GLSL en SPIR-V. Ce compilateur standard vérifiera que votre code correspond à la spécification. Vous pouvez également l'inclure comme une bibliothèque pour produire du SPIR-V au runtime, mais nous ne ferons pas cela dans ce tutoriel. Le compilateur est fourni avec le SDK et s'appelle `glslangValidator`, mais nous allons utiliser un autre compilateur nommé `glslc`, écrit par Google. L'avantage de ce dernier est qu'il utilise le même format d'options que GCC ou Clang, et inclu quelques fonctionnalités supplémentaires comme les *includes*. Les deux compilateurs sont fournis dans le SDK, vous n'avez donc rien de plus à télécharger.

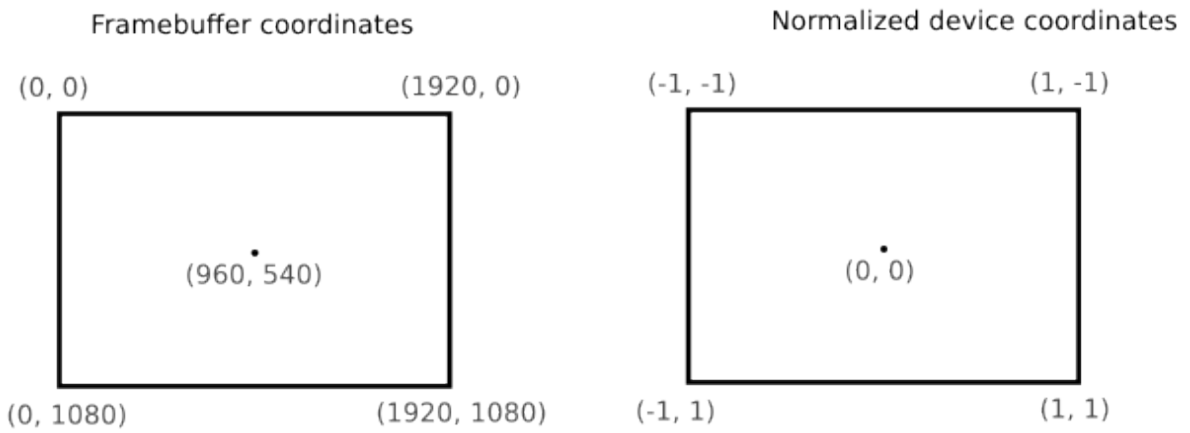
GLSL est un langage possédant une syntaxe proche du C. Les programmes y ont une fonction `main` invoquée pour chaque objet à traiter. Plutôt que d'utiliser des paramètres et des valeurs de retour, GLSL utilise des variables globales pour les entrées et sorties des invocations. Le langage possède des fonctionnalités avancées pour aider le travail avec les mathématiques nécessaires aux graphismes, avec par exemple des vecteurs, des matrices et des fonctions pour les traiter. On y trouve des fonctions pour réaliser des produits vectoriels ou des réflexions d'un vecteurs par rapport à un autre. Le type pour les vecteurs s'appelle `vec` et est suivi d'un nombre indiquant le nombre d'éléments, par exemple `vec3`. On peut accéder à ses données comme des membres avec par exemple `.y`, mais aussi créer de nouveaux vecteurs avec plusieurs indications, par exemple `vec3(1.0, 2.0, 3.0).xz` qui crée un `vec2` égal à `(1.0, 3.0)`. Leurs constructeurs peuvent aussi être des combinaisons de vecteurs et de valeurs. Par exemple il est possible de créer un `vec3` ainsi : `vec3(vec2(1.0, 2.0), 3.0)`.

Comme nous l'avons dit au chapitre précédent, nous devrons écrire un vertex shader et un fragment shader pour pouvoir afficher un triangle à l'écran. Les deux prochaines sections couvrirons ce travail, puis nous verrons comment créer des bytecodes SPIR-V avec ce code.

Le vertex shader

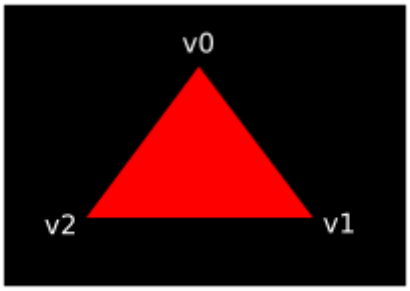
Le vertex shader traite chaque sommet envoyé depuis le programme C++. Il récupère des données telles la position, la normale, la couleur ou les coordonnées de texture. Ses sorties sont la position du somment dans l'espace de l'écran et les autres attributs qui doivent être fournies au reste de la pipeline, comme la couleur ou les coordonnées de texture. Ces valeurs seront interpolées lors de la rasterization afin de produire un dégradé continu. Ainsi les invocation du fragment shader recevrons des vecteurs dégradés entre deux sommets.

Une *clip coordinate* est un vecteur à quatre éléments émis par le vertex shader. Il est ensuite transformé en une *normalized screen coordinate* en divisant ses trois premiers composants par le quatrième. Ces coordonnées sont des coordonnées homogènes qui permettent d'accéder au frambuffer grâce à un repère de [-1, 1] par [-1, 1]. Il ressemble à cela :



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

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



Nous pouvons directement émettre ces coordonnées en mettant leur quatrième composant à 1 de telle sorte que la division ne change pas les valeurs.

Ces coordonnées devraient normalement être stockées dans un vertex buffer, mais sa création et son remplissage ne sont pas des opérations triviales. J'ai donc décidé de retarder ce sujet afin d'obtenir plus rapidement un résultat visible à l'écran. Nous ferons ainsi 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)
);

```

```
9  void main() {
10     gl_Position = vec4(positions[gl_VertexIndex], 0.0, 1.0);
11 }
```

La fonction `main` est invoquée pour chaque sommet. La variable prédéfinie `gl_VertexIndex` contient l'index du sommet à l'origine de l'invocation du `main`. Elle est en général utilisée comme index dans le vertex buffer, mais nous l'emploierons pour déterminer la coordonnée à émettre. Cette coordonnée est extraite d'un tableau prédéfini à trois entrées, et est combinée avec un `z` à 0.0 et un `w` à 1.0 pour faire de la division une identité. La variable prédéfinie `gl_Position` fonctionne comme sortie pour les coordonnées.

Le fragment shader

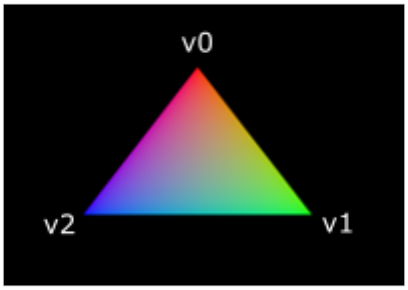
Le triangle formé par les positions émises par le vertex shader remplit un certain nombre de fragments. Le fragment shader est invoqué pour chacun d'entre eux et produit une couleur et une profondeur, qu'il envoie à un ou plusieurs framebuffer(s). Un fragment shader colorant tout en rouge est ainsi écrit :

```
1  #version 450
2
3  layout(location = 0) out vec4 outColor;
4
5  void main() {
6      outColor = vec4(1.0, 0.0, 0.0, 1.0);
7  }
```

Le `main` est appelé 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 doivent être incluses dans [0, 1]. Au contraire de `gl_Position`, il n'y a pas (plus exactement il n'y a plus) de variable prédéfinie dans laquelle entrer la valeur de la couleur. Vous devrez spécifier votre propre variable pour contenir la couleur du fragment, où `layout(location = 0)` indique l'index du framebuffer où la couleur sera écrite. Ici, la couleur rouge est écrite dans `outColor` liée au seul et unique premier framebuffer.

Une couleur pour chaque vertex

Afficher ce que vous voyez sur cette image ne serait pas plus intéressant qu'un triangle entièrement rouge?



Nous devons pour cela faire quelques petits changements aux deux shaders. Spécifions d'abord une couleur distincte pour chaque sommet. Ces couleurs seront inscrites dans le vertex shader de la même manière que les positions :

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

```
    vec3(0.0, 0.0, 1.0)
);
```

Nous devons maintenant passer ces couleurs au fragment shader afin qu'il puisse émettre des valeurs interpolées et dégradées au framebuffer. Ajoutez une variable de sortie pour la couleur dans le vertex shader et donnez lui une valeur dans le `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, dont la valeur sera l'interpolation correspondant à la position du fragment pour lequel le shader sera invoqué :

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

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

Les deux variables n'ont pas nécessairement le même nom, elles seront reliées selon l'index fourni dans la directive `location`. La fonction `main` doit être modifiée pour émettre une couleur possédant un canal alpha. Le résultat montré dans l'image précédente est dû à l'interpolation réalisée lors de la rasterization.

Compilation des shaders

Créez un dossier `shaders` à la racine de votre projet, puis enregistrez 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 correspondent à l'usage communément accepté.

Le contenu de `shader.vert` devrait être:

```
#version 450

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

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 bytecode SPIR-V à l'aide du programme `glslc`.

Windows

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

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

Corrigez le chemin vers `glslc.exe` pour que le .bat pointe effectivement là où le vôtre se trouve. Double-cliquez pour lancer ce script.

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
```

Corrigez le chemin menant au `glslc` pour qu'il pointe là où il est. Rendez le script exécutable avec la commande `chmod +x compile.sh` et lancez-le.

Fin des instructions spécifiques

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

Si votre shader contient une erreur de syntaxe le compilateur vous indiquera le problème et la ligne à laquelle il apparait. Essayez de retirer un point-virgule et voyez l'efficacité du debogueur. Essayez également de voir les arguments supportés. Il est possible de le forcer à émettre le bytecode sous un format compréhensible permettant de voir exactement ce que le shader fait et quelles optimisations le compilateur y a réalisées.

La compilation des shaders en ligne de commande est l'une des options les plus simples et les plus évidentes. C'est ce que nous utiliserons dans ce tutoriel. Sachez qu'il est également possible de compiler les shaders depuis votre code. Le SDK inclue la librairie libshaderc , qui permet de compiler le GLSL en SPIR-V depuis le programme C++.

Charger un shader

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

```cpp
#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(std::string {"échec de l'ouverture du fichier "} + filename
            + "!");
    }
}
```

La fonction `readFile` lira tous les octets du fichier qu'on lui indique et les retournera dans un `vector` de caractères servant ici d'octets. L'ouverture du fichier se fait avec deux paramètres particuliers : * `ate` : permet de commencer la lecture à la fin du fichier * `binary` : indique que le fichier doit être lu comme des octets et que ceux-ci ne doivent pas être formatés

Commencer la lecture à la fin permet d'utiliser la position du pointeur comme indicateur de la taille totale du fichier et nous pouvons ainsi allouer un stockage suffisant :

```cpp
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 d'un coup :

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

Nous pouvons enfin fermer le fichier et retourner les octets :

```cpp
file.close();

return buffer;
```

Appelons maintenant cette fonction depuis `createGraphicsPipeline` pour charger les bytecodes des deux shaders :

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

Assurez-vous que les shaders soient correctement chargés en affichant la taille des fichiers lus depuis votre programme puis en comparez ces valeurs à la taille des fichiers indiquées par l'OS. Notez que le code n'a pas besoin d'avoir un caractère nul en fin de chaîne car nous indiquerons à Vulkan sa taille exacte.

Créer des modules shader

Avant de passer ce code à la pipeline nous devons en faire un `VkShaderModule`. Créez pour cela une fonction `createShaderModule`.

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

}
```

Cette fonction prendra comme paramètre le buffer contenant le bytecode et créera un `VkShaderModule` avec ce code.

La création d'un module shader est très simple. Nous avons juste à indiquer un pointeur vers le buffer et la taille de ce buffer. Ces informations seront inscrites dans la structure `VkShaderModuleCreatInfo`. Le seul problème est que la taille doit être donnée en octets mais le pointeur sur le code est du type `uint32_t` et non du type `char`. Nous devrons donc utiliser `reinterpet_cast` sur notre pointeur. Cet opérateur de conversion nécessite que les données aient un alignement compatible avec `uint32_t`. Heuresement pour nous l'objet allocateur de la classe `std::vector` s'assure que les données satisfont le pire cas d'alignement.

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

Le `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 la création d'un module shader!");
}
```

Les paramètres sont les mêmes que pour la création des objets précédents : le logical device, le pointeur sur la structure avec les informations, le pointeur vers l'allocateur optionnnel et la référence à l'objet créé. Le buffer contenant le code peut être libéré immédiatement après l'appel. Retournez enfin le shader module créé :

```
return shaderModule;
```

Les modules shaders ne sont au fond qu'une fine couche autour du byte code chargé depuis les fichiers. Au moment de la création de la pipeline, les codes des shaders sont compilés et mis sur la carte. Nous pouvons donc détruire les modules dès que la pipeline est crée. Nous en ferons donc des variables locales à la fonction `createGraphicsPipeline` :

```
void createGraphicsPipeline() {
    auto vertShaderModule = createShaderModule(vertShaderCode);
    fragShaderModule = createShaderModule(fragShaderCode);

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

Ils doivent être libérés une fois que la pipeline est créée, juste avant que `createGraphicsPipeline` ne retourne. Ajoutez ceci à la fin de la fonction :

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

Le reste du code de ce chapitre sera ajouté entre les deux parties de la fonction présentés ci-dessus.

Création des étapes shader

Nous devons assigner une étape shader aux modules que nous avons crées. Nous allons utiliser une structure du type `VkPipelineShaderStageCreateInfo` pour cela.

Nous allons d'abord remplir cette structure pour le vertex shader, une fois de plus 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 le membre `sType`, consiste à dire à Vulkan à quelle étape le shader sera utilisé. Il existe une valeur d'énumération pour chacune des étapes possibles décrites dans le chapitre précédent.

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

Les deux membres suivants indiquent le module contenant le code et la fonction à invoquer en *entrypoint*. Il est donc possible de combiner plusieurs fragment shaders dans un seul module et de les différencier à l'aide de leurs points d'entrée. Nous nous contenterons du `main` standard.

Il existe un autre membre, celui-ci optionnel, appelé `pSpecializationInfo`, que nous n'utiliserons pas mais qu'il est intéressant d'évoquer. Il vous permet de donner des valeurs à des constantes présentes dans le code du shader. Vous pouvez ainsi configurer le comportement d'un shader lors de la création de la pipeline, ce qui est plus efficace que de le faire pendant l'affichage, car alors le compilateur (qui n'a toujours pas été invoqué!) peut éliminer des pants entiers de code sous un `if` vérifiant la valeur d'une constante ainsi configurée. Si vous n'avez aucune constante mettez ce paramètre à `nullptr`.

Modifier la structure pour qu'elle corresponde au fragment shader est très 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";
```

Intégrez ces deux valeurs dans un tableau que nous utiliserons plus tard et vous aurez fini ce chapitre!

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

C'est tout ce que nous dirons sur les étapes programmables de la pipeline. Dans le prochain chapitre nous verrons les étapes à fonction fixée.

Code C++ / Vertex shader / Fragment shader

Fonctions fixées

Les anciens APIs définissaient des configurations par défaut pour toutes les étapes à fonction fixée de la pipeline graphique. Avec Vulkan vous devez être explicite dans ce domaine également et devrez donc configurer la fonction de mélange par exemple. Dans ce chapitre nous remplirons toutes les structures nécessaires à la configuration des étapes à fonction fixée.

Entrée des sommets

La structure `VkPipelineVertexInputStateCreateInfo` décrit le format des sommets envoyés au vertex shader. Elle fait cela de deux manières :

- Liens (bindings) : espace entre les données et information sur ces données; sont-elles par sommet ou par instance? (voyez l'instanciation)
- Descriptions d'attributs : types d'attributs passés au vertex shader, de quels bindings les charger et avec quel décalage entre eux.

Dans la mesure où nous avons écrit les coordonnées directement dans le vertex shader, nous remplirons cette structure en indiquant qu'il n'y a aucune donnée à charger. Nous y reviendrons dans le chapitre sur les vertex buffers.

```
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`.

Input assembly

La structure `VkPipelineInputAssemblyStateCreateInfo` décrit la nature de la géométrie voulue quand les sommets sont reliés, et permet d'activer ou non la réévaluation des vertices. La première information est décrite dans le membre `topology` et peut prendre ces valeurs :

- `VK_PRIMITIVE_TOPOLOGY_POINT_LIST` : chaque sommet est un point
- `VK_PRIMITIVE_TOPOLOGY_LINE_LIST` : dessine une ligne liant deux sommet en n'utilisant ces derniers qu'une seule fois
- `VK_PRIMITIVE_TOPOLOGY_LINE_STRIP` : le dernier sommet de chaque ligne est utilisée comme premier sommet pour la ligne suivante
- `VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST` : dessine un triangle en utilisant trois sommets, sans en réutiliser pour le triangle suivant
- `VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP` : le deuxième et troisième sommets sont utilisées comme les deux premiers pour le triangle suivant

Les sommets sont normalement chargés séquentiellement depuis le vertex buffer. Avec un *element buffer* vous pouvez cependant choisir vous-même les indices à charger. Vous pouvez ainsi réaliser des optimisations, comme n'utiliser une combinaison de sommet qu'une seule fois au lieu de d'avoir les mêmes données plusieurs fois dans le buffer. Si vous mettez le membre `primitiveRestartEnable` à la valeur `VK_TRUE`, il devient alors possible d'interrompre les liaisons des vertices pour les modes `_STRIP` en utilisant l'index spécial `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;
```

Viewports et ciseaux

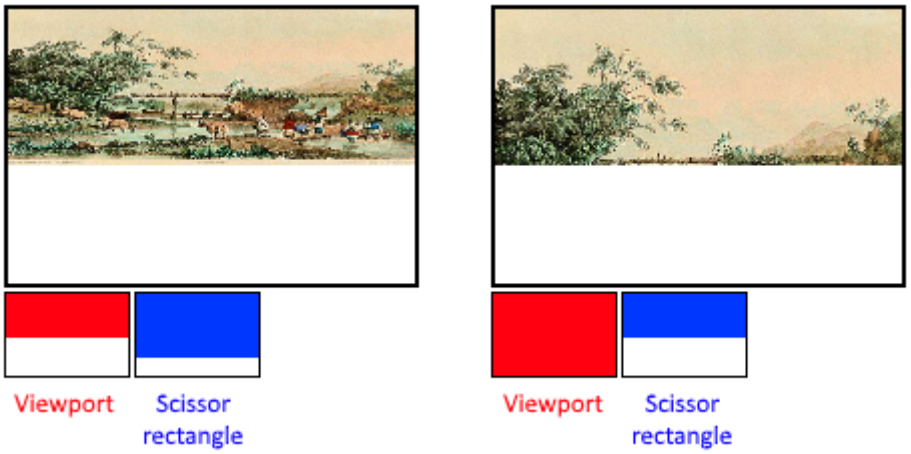
Un viewport décrit simplement la région d'un framebuffer sur laquelle le rendu sera effectué. Il couvrira dans la pratique quasiment toujours la totalité du framebuffer, et ce sera le cas dans ce tutoriel.

```
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 des macros `WIDTH` et `HEIGHT`. Les images de la swap chain seront plus tard les framebuffers sur lesquels la pipeline opérera, ce que nous devons prendre en compte en donnant les dimensions dynamiquement acquises.

Les valeurs `minDepth` et `maxDepth` indiquent l'étendue des valeurs de profondeur à utiliser pour le frambuffer. Ces valeurs doivent être dans `[0.0f, 1.0f]` mais `minDepth` peut être supérieure à `maxDepth`. Si vous ne faites rien de particulier contentez-vous des valeurs `0.0f` et `1.0f`.

Alors que les viewports définissent la transformation de l'image vers le framebuffer, les rectangles de ciseaux définissent la région de pixels qui sera conservée. Tout pixel en dehors des rectangles de ciseaux seront éliminés par le rasterizer. Ils fonctionnent plus comme un filtre que comme une transformation. Les différence sont illustrée ci-dessous. Notez que le rectangle de ciseau dessiné sous l'image de gauche n'est qu'une des possibilités : tout rectangle plus grand que le viewport aurait fonctionné.



Dans ce tutoriel nous voulons dessiner sur la totalité du framebuffer, et ce sans transformation. Nous définirons donc un rectangle de ciseaux couvrant tout le frambuffer :

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

Le viewport et le rectangle de ciseau se combinent en un *viewport state* à l'aide de la structure `VkPipelineViewportStateCreateInfo`. Il est possible sur certaines cartes graphiques d'utiliser plusieurs viewports et rectangles de ciseaux, c'est pourquoi la structure permet d'envoyer des tableaux de ces deux données. L'utilisation de cette possibilité nécessite de l'activer au préalable lors de la création du logical device.

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

Rasterizer

Le rasterizer récupère la géométrie définie par des sommets et calcule les fragments qu'elle recouvre. Ils sont ensuite traités par le fragment shaders. Il réalise également un test de profondeur, le face culling et le test de ciseau pour vérifier si le fragment doit effectivement être traité ou non. Il peut être configuré pour émettre des fragments remplissant tous les polygones ou bien ne remplissant que les cotés (wireframe rendering). 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 mis à `VK_TRUE`, les fragments au-delà des plans near et far ne pas supprimés mais affichés à cette distance. Cela est utile dans quelques situations telles que les shadow maps. Cela aussi doit être explicitement activé lors de la mise en place du logical device.

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

Si le membre `rasterizerDiscardEnable` est mis à `VK_TRUE`, aucune géométrie ne passe l'étape du rasterizer, ce qui désactive purement et simplement toute émission de donnée vers le frambuffer.

```
rasterizer.polygonMode = VK_POLYGON_MODE_FILL;
```

Le membre `polygonMode` définit la génération des fragments pour la géométrie. Les modes suivants sont disponibles :

- `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 sont dessinées comme des points

Tout autre mode que fill doit être activé lors de la mise en place du logical device.

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

Le membre `lineWidth` définit la largeur des lignes en terme de fragments. La taille maximale supportée dépend du GPU et pour toute valeur autre que `1.0f` l'extension `wideLines` doit être activée.

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

Le membre `cullMode` détermine quel type de face culling utiliser. Vous pouvez désactiver tout ce filtrage, n'éliminer que les faces de devant, que celles de derrière ou éliminer toutes les faces. Le membre `frontFace` indique l'ordre d'évaluation des vertices pour dire que la face est devant ou derrière, qui est le sens des aiguilles d'une montre ou le contraire.

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

Le rasterizer 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 le shadow mapping mais nous ne les utiliserons pas. Laissez `depthBiasEnabled` à la valeur `VK_FALSE`.

Multisampling

La structure `VkPipelineMultisampleCreateInfo` configure le multisampling, l'un des outils permettant de réaliser l'anti-aliasing. Le multisampling combine les résultats d'invocations du fragment shader sur des fragments de différents polygones qui résultent au même pixel. Cette superposition arrive plutôt sur les limites entre les géométries, et c'est aussi là que les problèmes visuels de hachage arrivent le plus. Dans la mesure où le fragment shader n'a pas besoin d'être invoqué plusieurs fois si seul un polygone correspond à un pixel, cette approche est beaucoup plus efficace que d'augmenter la résolution de la texture. Son utilisation nécessite son activation au niveau du 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.

Tests de profondeur et de pochoir

Si vous utilisez un buffer de profondeur (depth buffer) et/ou de pochoir (stencil buffer) vous devez configurer les tests de profondeur et de pochoir avec la structure `VkPipelineDepthStencilStateCreateInfo`. Nous n'avons aucun de ces buffers donc nous indiquerons `nullptr` à la place d'une structure. Nous y reviendrons au chapitre sur le depth buffering.

Color blending

La couleur donnée par un fragment shader doit être combinée avec la couleur déjà présente dans le framebuffer. Cette opération s'appelle color blending et il y a deux manières de la réaliser :

- Mélanger linéairement 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 color blending. La première, `VkPipelineColorBlendAttachmentState`, contient une configuration pour chaque framebuffer et la seconde, `VkPipelineColorBlendStateCreateInfo` contient les paramètres globaux pour ce color blending. Dans notre cas nous n'avons qu'un seul framebuffer :

```
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 de chaque framebuffer vous permet de configurer le color blending. L'opération sera effectuée à peu près comme ce pseudocode le montre :

```
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` vaut `VK_FALSE` la nouvelle couleur du fragment shader est inscrite dans le framebuffer sans modification et sans considération de la valeur déjà présente dans le framebuffer. Sinon les deux opérations de mélange sont exécutées pour former une nouvelle couleur. Un AND binaire lui est appliquée avec `colorWriteMask` pour déterminer les canaux devant passer.

L'utilisation la plus commune du mélange de couleurs utilise 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;
```

Avec cette méthode la valeur alpha correspond à une pondération pour la nouvelle valeur par rapport à l'ancienne. Les paramètres suivants permettent de faire exécuter ce calcul :

```
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 les énumérations `VkBlendFactor` et `VkBlendOp` dans la spécification.

La seconde structure doit posséder une référence aux structures spécifiques des framebuffers. Vous pouvez également y indiquer des constantes utilisables lors des opérations de mélange que nous venons de voir.

```
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 (la combinaison bit à bit) vous devez indiquer `VK_TRUE` au membre `logicOpEnable` et déterminer l'opération dans `logicOp`. Activer ce mode de mélange désactive automatiquement la première méthode aussi radicalement que si vous aviez indiqué `VK_FALSE` au membre `blendEnable` de la précédente structure pour chaque framebuffer. Le membre `colorWriteMask` sera également utilisé dans ce second 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 des invocations du fragment shader seront écrits directement dans le framebuffer.

États dynamiques

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

```
std::vector<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 = static_cast<uint32_t>(dynamicStates.size());
dynamicState.pDynamicStates = dynamicStates.data();
```

Les valeurs données lors de la configuration seront ignorées et vous devrez en fournir au moment du rendu. Nous y reviendrons plus tard. Cette structure peut être remplacée par `nullptr` si vous ne voulez pas utiliser de dynamisme sur ces états.

Pipeline layout

Les variables `uniform` dans les shaders sont des données globales similaires aux états dynamiques. Elles doivent être déterminées lors du rendu pour altérer les calculs des shaders sans avoir à les recréer. Elles sont très utilisées pour fournir les matrices de transformation au vertex shader et pour créer des samplers de texture dans les fragment shaders.

Ces variables doivent être configurées lors de la création de la pipeline en créant une variable de type `VkPipelineLayout`. Même si nous n'en utilisons pas dans nos shaders actuels nous devons en créer un vide.

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

```
VkPipelineLayout pipelineLayout;
```

Créons maintenant l'objet dans la fonction `createGraphicsPipline` :

```
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 de la création du pipeline layout!");
}
```

Cette structure informe également sur les *push constants*, une autre manière de passer des valeurs dynamiques au shaders que nous verrons dans un futur chapitre. Le pipeline layout sera utilisé pendant toute la durée du programme, nous devons donc le détruire dans la fonction `cleanup` :

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

Conclusion

Voila tout ce qu'il y a à savoir sur les étapes à fonction fixée! Leur configuration représente un gros travail mais nous sommes au courant de tout ce qui se passe dans la pipeline graphique, ce qui réduit les chances de comportement imprévu à cause d'un paramètre par défaut oublié.

Il reste cependant encore un objet à créer avant du finaliser la pipeline graphique. Cet objet s'appelle passe de rendu.

Code C++ / Vertex shader / Fragment shader

Render pass

Préparation

Avant de finaliser la création de la pipeline nous devons informer Vulkan des attachements des framebuffers utilisés lors du rendu. Nous devons indiquer combien chaque framebuffer aura de buffers de couleur et de profondeur, combien de samples il faudra utiliser avec chaque frambuffer et comment les utiliser tout au long des opérations de rendu. Toutes ces informations sont contenues dans un objet appelé *render pass*. Pour le configurer, créons la fonction `createRenderPass`. Appelez cette fonction depuis `initVulkan` avant `createGraphicsPipeline`.

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

...

void createRenderPass() {

}
```

Description de l'attachement

Dans notre cas nous aurons un seul attachement de couleur, et c'est une image de la swap chain.

```
void createRenderPass() {
    VkAttachmentDescription colorAttachment{};
    colorAttachment.format = swapChainImageFormat;
    colorAttachment.samples = VK_SAMPLE_COUNT_1_BIT;
}
```

Le `format` de l'attachement de couleur est le même que le format de l'image de la swap chain. Nous n'utilisons pas de multisampling pour le moment donc nous devons indiquer que nous n'utilisons qu'un seul sample.

```
colorAttachment.loadOp = VK_ATTACHMENT_LOAD_OP_CLEAR;
colorAttachment.storeOp = VK_ATTACHMENT_STORE_OP_STORE;
```

Les membres `loadOp` et `storeOp` définissent ce qui doit être fait avec les données de l'attachement respectivement avant et après le rendu. Pour `loadOp` nous avons les choix suivants :

- `VK_ATTACHMENT_LOAD_OP_LOAD` : conserve les données présentes dans l'attachement
- `VK_ATTACHMENT_LOAD_OP_CLEAR` : remplace le contenu par une constante
- `VK_ATTACHMENT_LOAD_OP_DONT_CARE` : ce qui existe n'est pas défini et ne nous intéresse pas

Dans notre cas nous utiliserons l'opération de remplacement pour obtenir un framebuffer noir avant d'afficher une nouvelle image. Il n'y a que deux possibilités pour le membre `storeOp` :

- `VK_ATTACHMENT_STORE_OP_STORE` : le rendu est gardé en mémoire et accessible plus tard
- `VK_ATTACHMENT_STORE_OP_DONT_CARE` : le contenu du framebuffer est indéfini dès la fin du rendu

Nous voulons voir le triangle à l'écran donc nous voulons l'opération de stockage.

```
colorAttachment.stencilLoadOp = VK_ATTACHMENT_LOAD_OP_DONT_CARE;
colorAttachment.stencilStoreOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
```

Les membres `loadOp` et `storeOp` s'appliquent aux données de couleur et de profondeur, et `stencilLoadOp` et `stencilStoreOp` s'appliquent aux données de stencil. Notre application n'utilisant pas de stencil buffer, nous pouvons indiquer que les données ne nous intéressent pas.

```
colorAttachment.initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
colorAttachment.finalLayout = VK_IMAGE_LAYOUT_PRESENT_SRC_KHR;
```

Les textures et les framebuffers dans Vulkan sont représentés par des objets de type `VkImage` possédant un certain format de pixels. Cependant l'organisation des pixels dans la mémoire peut changer selon ce que vous faites de cette image.

Les organisations les plus communes sont :

- `VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL` : images utilisées comme attachements de couleur
- `VK_IMAGE_LAYOUT_PRESENT_SRC_KHR` : images présentées à une swap chain
- `VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL` : image utilisées comme destination d'opérations de copie de mémoire

Nous discuterons plus précisément de ce sujet dans le chapitre sur les textures. Ce qui compte pour le moment est que les images doivent changer d'organisation mémoire selon les opérations qui leur sont appliquées au long de l'exécution de la pipeline.

Le membre `initialLayout` spécifie l'organisation de l'image avant le début du rendu. Le membre `finalLayout` fournit l'organisation vers laquelle l'image doit transitionner à la fin du rendu. La valeur `VK_IMAGE_LAYOUT_UNDEFINED` indique que le format précédent de l'image ne nous intéresse pas, ce qui peut faire perdre les données précédentes. Mais ce n'est pas un problème puisque nous effaçons de toute façon toutes les données avant le rendu. Puis, afin de rendre l'image compatible avec la swap chain, nous fournissons `VK_IMAGE_LAYOUT_PRESENT_SRC_KHR` pour `finalLayout`.

Subpasses et références aux attachements

Une unique passe de rendu est composée de plusieurs subpasses. Les subpasses sont des opérations de rendu dépendant du contenu présent dans le framebuffer quand elles commencent. Elles peuvent consister en des opérations de post-processing exécutées l'une après l'autre. En regroupant toutes ces opérations en une seule passe, Vulkan peut alors réaliser des optimisations et conserver de la bande passante pour de potentiellement meilleures performances. Pour notre triangle nous nous contenterons d'une seule subpasse.

Chacune d'entre elle référence un ou plusieurs attachements décrits par les structures que nous avons vues précédemment. Ces références sont elles-mêmes des structures du type `VkAttachmentReference` et ressemblent à cela :

```
1 VkAttachmentReference colorAttachmentRef{};
2 colorAttachmentRef.attachment = 0;
3 colorAttachmentRef.layout = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL;
```

Le paramètre `attachment` spécifie l'attachement à référencer à l'aide d'un indice correspondant à la position de la structure dans le tableau de descriptions d'attachements. Notre tableau ne consistera qu'en une seule référence donc son indice est nécessairement `0`. Le membre `layout` donne l'organisation que l'attachement devrait avoir au début d'une subpasse utilsant cette référence. Vulkan changera automatiquement l'organisation de l'attachement quand la subpasse commence. Nous voulons que l'attachement soit un color buffer, et pour cela la meilleure performance sera obtenue avec `VK_IMAGE_LAYOUT_COLOR_OPTIMAL`, comme son nom le suggère.

La subpasse est décrite dans la structure `VkSubpassDescription` :

```
1 VkSubpassDescription subpass{};
2 subpass.pipelineBindPoint = VK_PIPELINE_BIND_POINT_GRAPHICS;
```

Vulkan supportera également des *compute subpasses* donc nous devons indiquer que celle que nous créons est destinée aux graphismes. Nous spécifions ensuite la référence à l'attachement de couleurs :

```
1 subpass.colorAttachmentCount = 1;
2 subpass.pColorAttachments = &colorAttachmentRef;
```

L'indice de cet attachement est indiqué dans le fragment shader avec le `location = 0` dans la directive `layout(location = 0)out vec4 outColor`.

Les types d'attachements suivants peuvent être indiqués dans une subpasse :

- `pInputAttachments` : attachements lus depuis un shader
- `pResolveAttachments` : attachements utilisés pour le multisampling d'attachements de couleurs
- `pDepthStencilAttachment` : attachements pour la profondeur et le stencil
- `pPreserveAttachments` : attachements qui ne sont pas utilisés par cette subpasse mais dont les données doivent être conservées

Passe de rendu

Maintenant que les attachements et une subpasse simple ont été décrits nous pouvons enfin créer la render pass. Créez une nouvelle variable du type `VkRenderPass` au-dessus de la variable `pipelineLayout` :

```
VkRenderPass renderPass;
VkPipelineLayout pipelineLayout;
```

L'objet représentant la render pass peut alors être créé en remplissant la structure `VkRenderPassCreateInfo` dans laquelle nous devons remplir un tableau d'attachements et de subpasses. Les objets `VkAttachmentReference` référencent les attachements 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 de la création de la render pass!");
}
```

Comme l'organisation de la pipeline, nous aurons à utiliser la référence à 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, mais nous allons enfin créer la pipeline graphique et l'utiliser dès le prochain chapitre!

Code C++ / Vertex shader / Fragment shader

Conclusion

Nous pouvons maintenant combiner toutes les structures et tous les objets des chapitres précédentes pour créer la pipeline graphique! Voici un petit récapitulatif des objets que nous avons :

- Étapes shader : les modules shader définissent le fonctionnement des étapes programmables de la pipeline graphique
- Étapes à fonction fixée : plusieurs structures paramètrent les étapes à fonction fixée comme l'assemblage des entrées, le rasterizer, le viewport et le mélange des couleurs
- Organisation de la pipeline : les uniformes et push constants utilisées par les shaders, auxquelles on attribue une valeur pendant l'exécution de la pipeline
- Render pass : les attachements référencés par la pipeline et leurs utilisations

Tout cela combiné définit le fonctionnement de la pipeline graphique. Nous pouvons maintenant remplir la structure `VkGraphicsPipelineCreateInfo` à la fin de la fonction `createGraphicsPipeline`, mais avant les appels à la fonction `vkDestroyShaderModule` pour ne pas invalider les shaders que la pipeline utilisera.

Commençons par référencer le tableau de `VkPipelineShaderStageCreateInfo`.

```
VkGraphicsPipelineCreateInfo pipelineInfo{};
pipelineInfo.sType = VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_CREATE_INFO;
pipelineInfo.stageCount = 2;
pipelineInfo.pStages = shaderStages;
```

Puis donnons toutes les structure décrivant les étapes à fonction fixée.

```
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
```

Après cela vient l'organisation de la pipeline, qui est une référence à un objet Vulkan plutôt qu'une structure.

```
pipelineInfo.layout = pipelineLayout;
```

Finalement nous devons fournir les références à la render pass et aux indices des subpasses. Il est aussi possible d'utiliser d'autres render passes avec cette pipeline mais elles doivent être compatibles avec `renderPass`. La signification de compatible est donnée ici, mais nous n'utiliserons pas cette possibilité dans ce tutoriel.

```
pipelineInfo.renderPass = renderPass;
pipelineInfo.subpass = 0;
```

Il nous reste en fait deux paramètres : `basePipelineHandle` et `basePipelineIndex`. Vulkan vous permet de créer une nouvelle pipeline en "héritant" d'une pipeline déjà existante. L'idée derrière cette fonctionnalité est qu'il est moins coûteux de créer une pipeline à partir d'une qui existe déjà, mais surtout que passer d'une pipeline à une autre est plus rapide si elles ont un même parent. Vous pouvez spécifier une pipeline de deux manières : soit en fournissant une référence soit en donnant l'indice de la pipeline à hériter. Nous n'utilisons pas cela donc nous indiquerons une référence nulle et un indice invalide. Ces valeurs ne sont de toute façon utilisées que si le champ `flags` de la structure `VkGraphicsPipelineCreateInfo` comporte `VK_PIPELINE_CREATE_DERIVATIVE_BIT`.

```
pipelineInfo.basePipelineHandle = VK_NULL_HANDLE; // Optionnel
pipelineInfo.basePipelineIndex = -1; // Optionnel
```

Préparons-nous pour l'étape finale en créant un membre donnée où stocker la référence à la `VkPipeline` :

```
VkPipeline graphicsPipeline;
```

Et créons enfin la pipeline graphique :

```
if (vkCreateGraphicsPipelines(device, VK_NULL_HANDLE, 1, &pipelineInfo, nullptr,
    &graphicsPipeline) != VK_SUCCESS) {
    throw std::runtime_error("échec de la création de la pipeline graphique!");
}
```

La fonction `vkCreateGraphicsPipelines` possède en fait plus de paramètres que les fonctions de création d'objet 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 que nous n'utilisons pas ici (mais que nous reverrons dans un chapitre qui lui sera dédié) sert à fournir un objet `VkPipelineCache` optionnel. Un tel objet peut être stocké et réutilisé entre plusieurs appels de la fonction et même entre plusieurs exécutions du programme si son contenu est correctement stocké dans un fichier. Cela permet de grandement accélérer la création des pipelines.

La pipeline graphique est nécessaire à toutes les opérations d'affichage, nous ne devrons donc la supprimer qu'à la fin du programme dans la fonction `cleanup` :

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

Exécutez votre programme pour vérifier que tout ce travail a enfin résulté dans la création d'une pipeline graphique. Nous sommes de plus en plus proches d'avoir un dessin à l'écran! Dans les prochains chapitres nous générerons les framebuffers à partir des images de la swap chain et préparerons les commandes d'affichage.

Code C++ / Vertex shader / Fragment shader

Effectuer le rendu

Framebuffers

Nous avons beaucoup parlé de framebuffers dans les chapitres précédents, et nous avons mis en place la render pass pour qu'elle en accepte un du même format que les images de la swap chain. Pourtant nous n'en avons encore créé aucun.

Les attachements de différents types spécifiés durant la render pass sont liés en les considérant dans des objets de type `VkFramebuffer`. Un tel objet référence toutes les `VkImageView` utilisées comme attachements par une passe. Dans notre cas nous n'en aurons qu'un : un attachement de couleur, qui servira de cible d'affichage uniquement. Cependant l'image utilisée dépendra de l'image fournie par la swap chain lors de la requête pour l'affichage. Nous devons donc créer un framebuffer pour chacune des images de la swap chain et utiliser le bon au moment de l'affichage.

Pour cela créez un autre `std::vector` qui contiendra des framebuffers :

```
std::vector<VkFramebuffer> swapChainFramebuffers;
```

Nous allons remplir ce `vector` depuis une nouvelle fonction `createFramebuffers` que nous appellerons depuis `initVulkan` juste après la création de la 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 framebuffers :

```
void createFramebuffers() {
    swapChainFramebuffers.resize(swapChainImageViews.size());
}
```

Nous allons maintenant itérer à travers toutes les images et créer un framebuffer à partir de chacune d'entre 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 de la création d'un framebuffer!");
    }
}
```

Comme vous le pouvez le voir la création d'un framebuffer est assez simple. Nous devons d'abord indiquer avec quelle `renderPass` le framebuffer doit être compatible. Sachez que si vous voulez utiliser un framebuffer avec plusieurs render passes, les render passes spécifiées doivent être compatibles entre elles. La compatibilité signifie ici approximativement qu'elles utilisent le même nombre d'attachements du même type. Ceci implique qu'il ne faut pas s'attendre à ce qu'une render pass puisse ignorer certains attachements d'un framebuffer qui en aurait trop.

Les paramètres `attachementCount` et `pAttachments` doivent donner la taille du tableau contenant les `VkImageViews` qui servent d'attachements.

Les paramètres `width` et `height` sont évidents. Le membre `layers` correspond au nombres de couches dans les images fournies comme attachements. Les images de la swap chain n'ont toujours qu'une seule couche donc nous indiquons `1`.

Nous devons détruire les framebuffers avant les image views et la render pass dans la fonction `cleanup` :

```
void cleanup() {
    for (auto framebuffer : swapChainFramebuffers) {
        vkDestroyFramebuffer(device, framebuffer, nullptr);
    }

    ...
}
```

Nous avons atteint le moment où tous les objets sont prêts pour l'affichage. Dans le prochain chapitre nous allons écrire les commandes d'affichage.

Code C++ / Vertex shader / Fragment shader

Command buffers

Les commandes Vulkan, comme les opérations d'affichage et de transfert mémoire, ne sont pas réalisées avec des appels de fonctions. Il faut pré-enregistrer toutes les opérations dans des *command buffers*. L'avantage est que vous pouvez préparer tout ce travail à l'avance et depuis plusieurs threads, puis vous contenter d'indiquer à Vulkan quel command buffer doit être exécuté. Cela réduit considérablement la bande passante entre le CPU et le GPU et améliore grandement les performances.

Command pools

Nous devons créer une *command pool* avant de pouvoir créer les command buffers. Les command pools gèrent la mémoire utilisée par les buffers, et c'est de fait les command pools qui nous instancient les command buffers. Ajoutez un nouveau membre donnée à la classe de type `VkCommandPool` :

```
VkCommandPool commandPool;
```

Créez ensuite la fonction `createCommandPool` et appelez-la depuis `initVulkan` après la création du framebuffer.

```
void initVulkan() {
    createInstance();
    setupDebugMessenger();
    createSurface();
    pickPhysicalDevice();
    createLogicalDevice();
    createSwapChain();
    createImageViews();
    createRenderPass();
    createGraphicsPipeline();
    createFramebuffers();
    createCommandPool();
}

...

void createCommandPool() {

}
```

La création d'une command pool 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; // Optionel
```

Les commands buffers sont exécutés depuis une queue, comme la queue des graphismes et de présentation que nous avons récupérées. Une command pool ne peut allouer des command buffers compatibles qu'avec une seule famille de queues. Nous allons enregistrer des commandes d'affichage, c'est pourquoi nous avons récupéré une queue de graphismes.

Il existe deux valeurs acceptées par `flags` pour les command pools :

- `VK_COMMAND_POOL_CREATE_TRANSIENT_BIT` : informe que les command buffers sont ré-enregistrés très souvent, ce qui peut inciter Vulkan (et donc le driver) à ne pas utiliser le même type d'allocation
- `VK_COMMAND_POOL_CREATE_RESET_COMMAND_BUFFER_BIT` : permet aux command buffers d'être ré-enregistrés individuellement, ce que les autres configurations ne permettent pas

Nous n'enregistrerons les command buffers qu'une seule fois au début du programme, nous n'aurons donc pas besoin de ces fonctionnalités.

```
if (vkCreateCommandPool(device, &poolInfo, nullptr, &commandPool) != VK_SUCCESS) {
    throw std::runtime_error("échec de la création d'une command pool!");
}
```

Terminez la création de la command pool à l'aide de la fonction `vkCreateComandPool`. Elle ne comprend pas de paramètre particulier. Les commandes seront utilisées tout au long du programme pour tout affichage, nous ne devons donc la détruire que dans la fonction `cleanup` :

```
void cleanup() {
    vkDestroyCommandPool(device, commandPool, nullptr);

    ...
}
```

Allocation des command buffers

Nous pouvons maintenant allouer des command buffers et enregistrer les commandes d'affichage. Dans la mesure où l'une des commandes consiste à lier un framebuffer nous devrons les enregistrer pour chacune des images de la swap chain. Créez pour cela une liste de `VkCommandBuffer` et stockez-la dans un membre donnée de la classe. Les command buffers sont libérés avec la destruction de leur command pool, nous n'avons donc pas à faire ce travail.

```
std::vector<VkCommandBuffer> commandBuffers;
```

Commençons maintenant à travailler sur notre fonction `createCommandBuffers` qui allouera et enregistrera les command buffers 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 command buffers sont alloués par la fonction `vkAllocateCommandBuffers` qui prend en paramètre une structure du type `VkCommandBufferAllocateInfo`. Cette structure spécifie la command pool et le nombre de buffers à allouer depuis celle-ci :

```
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 de l'allocation de command buffers!");
}
```

Les command buffers peuvent être *primaires* ou *secondaires*, ce que l'on indique avec le paramètre `level`. Il peut prendre les valeurs suivantes :

- `VK_COMMAND_BUFFER_LEVEL_PRIMARY` : peut être envoyé à une queue pour y être exécuté mais ne peut être appelé par d'autres command buffers
- `VK_COMMAND_BUFFER_LEVEL_SECONDARY` : ne peut pas être directement émis à une queue mais peut être appelé par un autre command buffer

Nous n'utiliserons pas la fonctionnalité de command buffer secondaire ici. Sachez que le mécanisme de command buffer secondaire est à la base de la génération rapie de commandes d'affichage depuis plusieurs threads.

Début de l'enregistrement des commandes

Nous commençons l'enregistrement des commandes en appelant `vkBeginCommandBuffer`. Cette fonction prend une petite structure du type `VkCommandBufferBeginInfo` en argument, permettant d'indiquer quelques détails sur l'utilisation du command buffer.

```
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; // Optionel

    if (vkBeginCommandBuffer(commandBuffers[i], &beginInfo) != VK_SUCCESS) {
        throw std::runtime_error("erreur au début de l'enregistrement d'un command buffer!");
    }
}
```

L'utilisation du command buffer s'indique avec le paramètre `flags`, qui peut prendre les valeurs suivantes :

- `VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT` : le command buffer sera ré-enregistré après son utilisation, donc invalidé une fois son exécution terminée
- `VK_COMMAND_BUFFER_USAGE_RENDER_PASS_CONTINUE_BIT` : ce command buffer secondaire sera intégralement exécuté dans une unique render pass
- `VK_COMMAND_BUFFER_USAGE_SIMULTANEOUS_USE_BIT` : le command buffer peut être ré-envoyé à la queue alors qu'il y est déjà et/ou est en cours d'exécution

Nous n'avons pas besoin de ces flags ici.

Le paramètre `pInheritanceInfo` n'a de sens que pour les command buffers secondaires. Il indique l'état à hériter de l'appel par le command buffer primaire.

Si un command buffer est déjà prêt un appel à `vkBeginCommandBuffer` le regénèrera implicitement. Il n'est pas possible d'enregistrer un command buffer en plusieurs fois.

Commencer une render pass

L'affichage commence par le lancement de la render pass réalisé par `vkCmdBeginRenderPass`. La passe est configurée à l'aide des paramètres remplis dans une structure de type `VkRenderPassBeginInfo`.

```
VkRenderPassBeginInfo renderPassInfo{};
renderPassInfo.sType = VK_STRUCTURE_TYPE_RENDER_PASS_BEGIN_INFO;
renderPassInfo.renderPass = renderPass;
renderPassInfo.framebuffer = swapChainFramebuffers[i];
```

Ces premiers paramètres sont la render pass elle-même et les attachements à lui fournir. Nous avons créé un framebuffer pour chacune des images de la swap chain qui spécifient ces images comme attachements de couleur.

```
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 chargements et stockages shaders se produiront. Les pixels hors de cette région auront une valeur non définie. Elle doit correspondre à la taille des attachements pour avoir une performance optimale.

```
VkClearValue 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 remplacer le contenu (fonctionnalité que nous avions activée avec `VK_ATTACHMENT_LOAD_CLEAR`). J'ai utilisé un noir complètement opaque.

```
vkCmdBeginRenderPass(commandBuffers[i], &renderPassInfo, VK_SUBPASS_CONTENTS_INLINE);
```

La render pass peut maintenant commencer. Toutes les fonctions enregistrables se reconnaisent à leur préfixe `vkCmd`. Comme elles retournent toutes `void` nous n'avons aucun moyen de gérer d'éventuelles erreurs avant d'avoir fini l'enregistrement.

Le premier paramètre de chaque commande est toujours le command buffer qui stockera l'appel. Le second paramètre donne des détails sur la render pass à l'aide de la structure que nous avons préparée. Le dernier paramètre informe sur la provenance des commandes pendant l'exécution de la passe. Il peut prendre ces valeurs :

- `VK_SUBPASS_CONTENTS_INLINE` : les commandes de la render pass seront inclues directement dans le command buffer (qui est donc primaire)
- `VK_SUBPASS_CONTENTS_SECONDARY_COMMAND_BUFFER` : les commandes de la render pass seront fournies par un ou plusieurs command buffers secondaires

Nous n'utiliserons pas de command buffer secondaire, nous devons donc fournir la première valeur à la fonction.

Commandes d'affichage basiques

Nous pouvons maintenant activer la pipeline graphique :

```
vkCmdBindPipeline(commandBuffers[i], VK_PIPELINE_BIND_POINT_GRAPHICS, graphicsPipeline);
```

Le second paramètre indique que la pipeline est bien une pipeline graphique et non de calcul. Nous avons fourni à Vulkan les opérations à exécuter avec la pipeline graphique et les attachements que le fragment shader devra utiliser. Il ne nous reste donc plus qu'à lui dire d'afficher un triangle :

```
vkCmdDraw(commandBuffers[i], 3, 1, 0, 0);
```

Le fonction `vkCmdDraw` est assez ridicule quand on sait tout ce qu'elle implique, mais sa simplicité est due à ce que tout a déjà été préparé en vue de ce moment tant attendu. Elle possède les paramètres suivants en plus du command buffer concerné :

- `vertexCount` : même si nous n'avons pas de vertex buffer, nous avons techniquement trois vertices à dessiner
- `instanceCount` : sert au rendu instancié (instanced rendering); indiquez `1` si vous ne l'utilisez pas
- `firstVertex` : utilisé comme décalage dans le vertex buffer et définit ainsi la valeur la plus basse pour `glVertexIndex`
- `firstInstance` : utilisé comme décalage pour l'instanced rendering et définit ainsi la valeur la plus basse pour `gl_InstanceIndex`

Finitions

La render pass peut ensuite être terminée :

```
vkCmdEndRenderPass(commandBuffers[i]);
```

Et nous avons fini l'enregistrement du command buffer :

```
if (vkEndCommandBuffer(commandBuffers[i]) != VK_SUCCESS) {
    throw std::runtime_error("échec de l'enregistrement d'un command buffer!");
}
```

Dans le prochain chapitre nous écrirons le code pour la boucle principale. Elle récupérera une image de la swap chain, exécutera le bon command buffer et retournera l'image complète à la swap chain.

Code C++ / Vertex shader / Fragment shader

Rendu et présentation

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 `mainLoop` :

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

...

void drawFrame() {

}
```

Synchronisation

Le fonction `drawFrame` réalisera les opérations suivantes :

- Acquérir une image depuis la swap chain
- Exécuter le command buffer correspondant au framebuffer dont l'attachement est l'image obtenue
- Retourner l'image à la swap chain pour présentation

Chacune de ces actions n'est réalisée qu'avec un appel de fonction. Cependant ce n'est pas aussi simple : les opérations sont par défaut exécutées de manière asynchrones. La fonction retourne aussitôt que les opérations sont lancées, et par conséquent l'ordre d'exécution est indéfini. Cela nous pose problème car chacune des opérations que nous voulons lancer dépendent des résultats de l'opération la précédant.

Il y a deux manières de synchroniser les évènements de la swap chain : les *fences* et les *sémaphores*. Ces deux objets permettent d'attendre qu'une opération se termine en relayant un signal émis par un processus généré par la fonction à l'origine du lancement de l'opération.

Ils ont cependant une différence : l'état d'une fence peut être accédé depuis le programme à l'aide de fonctions telles que `vkWaitForFences` alors que les sémaphores ne le permettent pas. Les fences sont généralement utilisées pour synchroniser votre programme avec les opérations alors que les sémaphores synchronisent les opérations entre elles. Nous voulons synchroniser les queues, les commandes d'affichage et la présentation, donc les sémaphores nous conviennent le mieux.

Sémaphores

Nous aurons besoin d'un premier sémaphore pour indiquer que l'acquisition de l'image s'est bien réalisée, puis d'un second pour prévenir de la fin du rendu et permettre à l'image d'être retournée dans la swap chain. Créez deux membres données pour stocker ces sémaphores :

```
VkSemaphore imageAvailableSemaphore;
VkSemaphore renderFinishedSemaphore;
```

Pour leur création nous allons avoir besoin d'une dernière fonction `create...` pour cette partie du tutoriel. Appelez-la `createSemaphores` :

```
void initVulkan() {
    createInstance();
    setupDebugMessenger();
    createSurface();
    pickPhysicalDevice();
    createLogicalDevice();
    createSwapChain();
    createImageViews();
    createRenderPass();
    createGraphicsPipeline();
    createFramebuffers();
    createCommandPool();
    createCommandBuffers();
    createSemaphores();
}

...

void createSemaphores() {

}
```

La création d'un sémaphore passe par le remplissage d'une structure de type `VkSemaphoreCreateInfo`. Cependant cette structure ne requiert pour l'instant rien d'autre que le membre `sType` :

```
void createSemaphores() {
    VkSemaphoreCreateInfo semaphoreInfo{};
    semaphoreInfo.sType = VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO;
}
```

De futures version de Vulkan ou des extensions pourront à terme donner un intérêt aux membre `flags` et `pNext`, comme pour d'autres structures. Créez les sémaphores comme suit :

```
if (vkCreateSemaphore(device, &semaphoreInfo, nullptr, &imageAvailableSemaphore) !=
    VK_SUCCESS ||
    vkCreateSemaphore(device, &semaphoreInfo, nullptr, &renderFinishedSemaphore) !=
        VK_SUCCESS) {

    throw std::runtime_error("échec de la création des sémaphores!");
}
```

Les sémaphores doivent être détruits à la fin du programme depuis la fonction `cleanup` :

```
void cleanup() {
    vkDestroySemaphore(device, renderFinishedSemaphore, nullptr);
    vkDestroySemaphore(device, imageAvailableSemaphore, nullptr);
```

Acquérir une image de la swap chain

La première opération à réaliser dans `drawFrame` est d'acquérir une image depuis la swap chain. La swap chain étant une extension nous allons encore devoir utiliser des fonction suffixées de `KHR` :

```
void drawFrame() {
    uint32_t imageIndex;
    vkAcquireNextImageKHR(device, swapChain, UINT64_MAX, imageAvailableSemaphore,
        VK_NULL_HANDLE, &imageIndex);
}
```

Les deux premiers paramètres de `vkAcquireNextImageKHR` sont le logical device et la swap chain depuis laquelle récupérer les images. Le troisième paramètre spécifie une durée maximale en nanosecondes avant d'abandonner l'attente si aucune image n'est disponible. Utiliser la plus grande valeur possible pour un `uint32_t` le désactive.

Les deux paramètres suivants sont les objets de synchronisation qui doivent être informés de la complétion de l'opération de récupération. Ce sera à partir du moment où le sémaphore que nous lui fournissons reçoit un signal que nous pouvons commencer à dessiner.

Le dernier paramètre permet de fournir à la fonction une variable dans laquelle elle stockera l'indice de l'image récupérée dans la liste des images de la swap chain. Cet indice correspond à la `VkImage` dans notre `vector swapChainImages`. Nous utiliserons cet indice pour invoquer le bon command buffer.

Envoi du command buffer

L'envoi à la queue et la synchronisation de celle-ci sont configurés à l'aide de paramètres dans la structure `VkSubmitInfo` que nous allons remplir.

```
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 (sans compter `sType`) fournissent le sémaphore indiquant si l'opération doit attendre et l'étape du rendu à laquelle s'arrêter. Nous voulons attendre juste avant l'écriture des couleurs sur l'image. Par contre nous laissons à l'implémentation la possibilité d'exécuter toutes les étapes précédentes

d'ici là. Notez que chaque étape indiquée dans `waitStages` correspond au sémaphore de même indice fourni dans `waitSemaphores`.

```
submitInfo.commandBufferCount = 1;
submitInfo.pCommandBuffers = &commandBuffers[imageIndex];
```

Les deux paramètres qui suivent indiquent les command buffers à exécuter. Nous devons ici fournir le command buffer qui utilise l'image de la swap chain que nous venons de récupérer comme attachement de couleur.

```
VkSemaphore signalSemaphores[] = {renderFinishedSemaphore};
submitInfo.signalSemaphoreCount = 1;
submitInfo.pSignalSemaphores = signalSemaphores;
```

Les paramètres `signalSemaphoreCount` et `pSignalSemaphores` indiquent les sémaphores auxquels indiquer que les command buffers ont terminé leur exécution. Dans notre cas nous utiliserons notre `renderFinishedSemaphore`.

```
if (vkQueueSubmit(graphicsQueue, 1, &submitInfo, VK_NULL_HANDLE) != VK_SUCCESS) {
    throw std::runtime_error("échec de l'envoi d'un command buffer!");
}
```

Nous pouvons maintenant envoyer notre command buffer à la queue des graphismes en utilisant `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 fence optionnelle. Celle-ci sera prévenue de la fin de l'exécution des command buffers. Nous n'en utilisons pas donc passerons `VK_NULL_HANDLE`.

Subpass dependencies

Les subpasses s'occupent automatiquement de la transition de l'organisation des images. Ces transitions sont contrôlées par des *subpass dependencies*. Elles indiquent la mémoire et l'exécution entre les subpasses. Nous n'avons certes qu'une seule subpasse pour le moment, mais les opérations avant et après cette subpasse comptent aussi comme des subpasses implicites.

Il existe deux dépendances préexistantes capables de gérer les transitions au début et à la fin de la render pass. Le problème est que cette première dépendance ne s'exécute pas au bon moment. Elle part du principe que la transition de l'organisation de l'image doit être réalisée au début de la pipeline, mais dans notre programme l'image n'est pas encore acquise à ce moment! Il existe deux manières de régler ce problème. Nous pourrions changer `waitStages` pour `imageAvailableSemaphore` à `VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT` pour être sûrs que la pipeline ne commence pas avant que l'image ne soit acquise, mais nous perdrions en performance car les shaders travaillant sur les vertices n'ont pas besoin de l'image. Il faudrait faire quelque chose de plus subtil. Nous allons donc plutôt faire attendre la render pass à l'étape `VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT` et faire la transition à ce moment. Cela nous donne de plus une bonne excuse pour s'intéresser au fonctionnement des subpass dependencies.

Celles-ci sont décrites dans une structure de type `VkSubpassDependency`. Créez en une dans la fonction `createRenderPass` :

```
VkSubpassDependency dependency{};
dependency.srcSubpass = VK_SUBPASS_EXTERNAL;
dependency.dstSubpass = 0;
```

Les deux premiers champs permettent de fournir l'indice de la subpasse d'origine et de la subpasse d'arrivée. La valeur particulière `VK_SUBPASS_EXTERNAL` réfère à la subpass implicite soit avant soit après la render pass, selon que cette valeur est indiquée dans respectivement `srcSubpass` ou `dstSubpass`. L'indice `0` correspond à notre seule et unique subpasse. La valeur fournie à `dstSubpass` doit toujours être supérieure à `srcSubpass` car sinon une boucle infinie peut apparaître (sauf si une des subpasse 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 doivent être considérées. Nous voulons attendre la fin de l'extraction de l'image avant d'y accéder, hors ceci est déjà configuré pour être synchronisé avec l'étape d'écriture sur l'attachement. C'est pourquoi nous n'avons qu'à attendre à cette étape.

```
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'attachement de couleur sont celles ayant trait à l'écriture. Ces paramètres permettent de faire attendre la transition jusqu'à ce qu'elle soit possible, ce qui correspond au moment où la passe accède à cet attachement puisqu'elle est elle-même configurée pour attendre ce moment.

```
renderPassInfo.dependencyCount = 1;
renderPassInfo.pDependencies = &dependency;
```

Nous fournissons enfin à la structure ayant trait à la render pass un tableau de configurations pour les subpass dependencies.

Présentation

La dernière étape pour l'affichage consiste à envoyer le résultat à la swap chain. La présentation est configurée avec une structure de type `VkPresentInfoKHR`, et nous ferons cela 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 les sémaphores devant signaler que la présentation peut se dérouler.

```
VkSwapchainKHR swapChains[] = {swapChain};
presentInfo.swapchainCount = 1;
presentInfo.pSwapchains = swapChains;
presentInfo.pImageIndices = &imageIndex;
```

Les deux paramètres suivants fournissent un tableau contenant notre unique swap chain qui présentera les images et l'indice de l'image pour celle-ci.

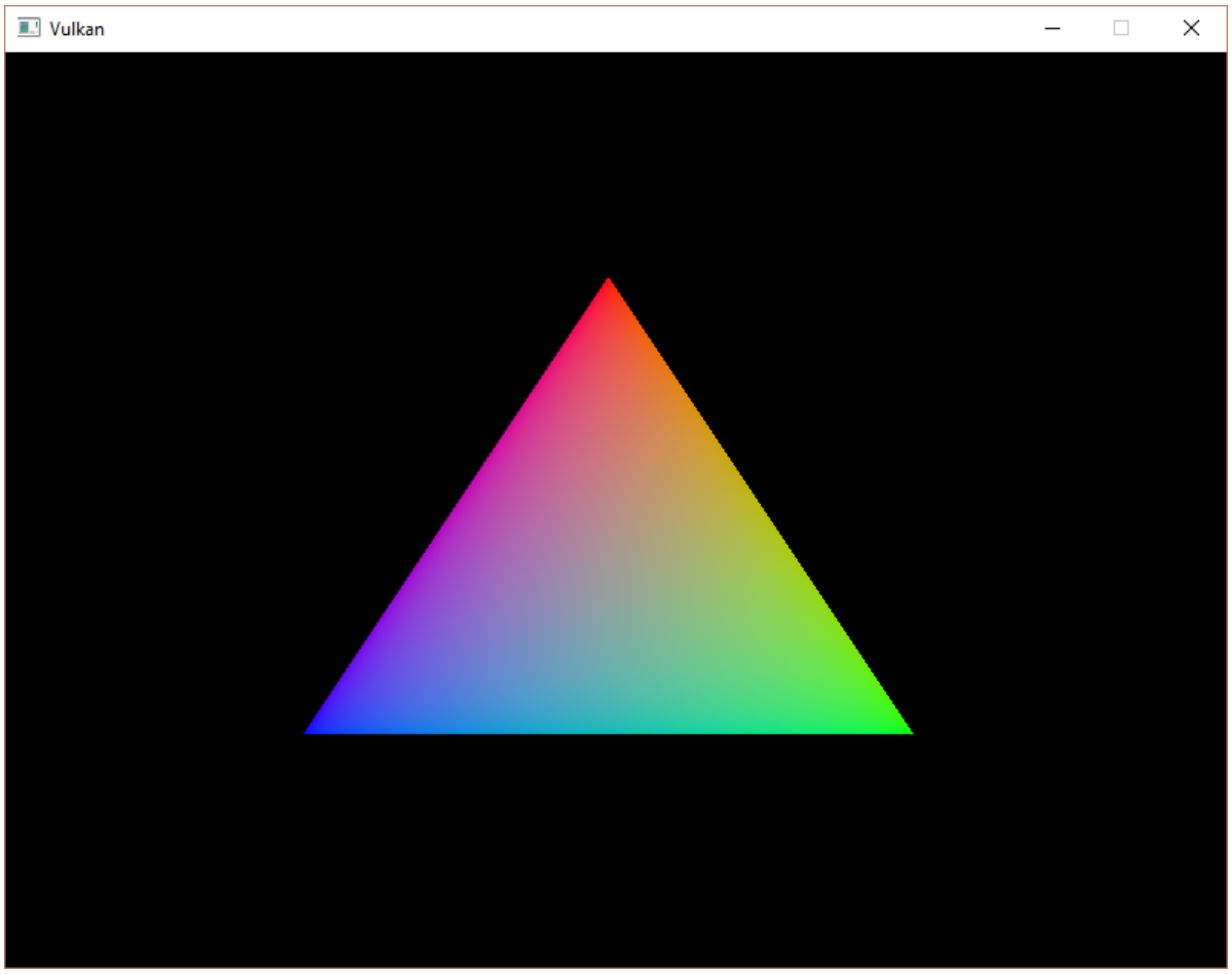
```
presentInfo.pResults = nullptr; // Optionnel
```

Ce dernier paramètre est optionnel. Il vous permet de fournir un tableau de `VkResult` que vous pourrez consulter pour vérifier que toutes les swap chain ont bien présenté leur image sans problème. Cela n'est pas nécessaire dans notre cas, car n'utilisant qu'une seule swap chain nous pouvons simplement regarder la valeur de retour de la fonction de présentation.

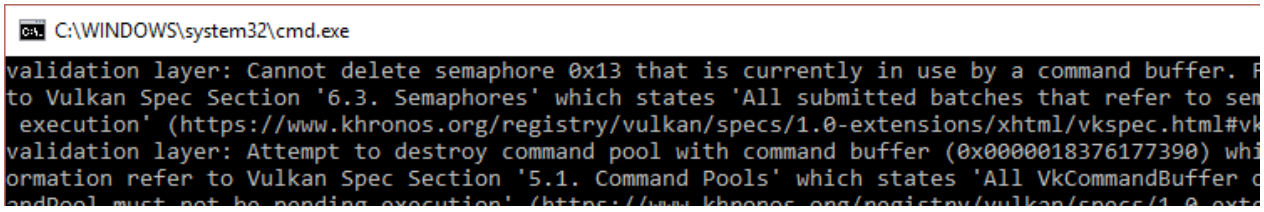
```
vkQueuePresentKHR(presentQueue, &presentInfo);
```

La fonction `vkQueuePresentKHR` émet la requête de présentation d'une image par 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, mais plutôt qu'il doit s'adapter à des changements.

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 essayez de quitter proprement le programme vous obtiendrez un crash et un message semblable à ceci :



N'oubliez pas que puisque les opérations dans `drawFrame` sont asynchrones il est quasiment certain que lorsque vous quittez le programme, celui-ci exécute encore des instructions et cela implique que vous essayez de libérer des ressources en train d'être utilisées. Ce qui est rarement une bonne idée, surtout avec du bas niveau comme Vulkan.

Pour régler ce problème nous devons attendre que le logical device finisse l'opération qu'il est en train de réaliser avant de quitter `mainLoop` et de détruire la fenêtre :

```
1 void mainLoop() {
2     while (!glfwWindowShouldClose(window)) {
3         glfwPollEvents();
```

```
        drawFrame();
    }

    vkDeviceWaitIdle(device);
}
```

Vous pouvez également attendre la fin d'une opération quelconque depuis une queue spécifique à l'aide de la fonction `vkQueueWaitIdle`. Ces fonction peuvent par ailleurs être utilisées pour réaliser une synchronisation très basique, mais très inefficace. Le programme devrait maintenant se terminer sans problème quand vous fermez la fenêtre.

Frames en vol

Si vous lancez l'application avec les validation layers maintenant, vous pouvez soit avoir des erreurs soit vous remarquerez que l'utilisation de la mémoire augmente, lentement mais sûrement. La raison est que l'application soumet rapidement du travail dans la fonction `drawframe`, mais que l'on ne vérifie pas si ces rendus sont effectivement terminés. Si le CPU envoie plus de commandes que le GPU ne peut en exécuter, ce qui est le cas car nous envoyons nos command buffers de manière totalement débridée, la queue de graphismes va progressivement se remplir de travail à effectuer. Pire encore, nous utilisons `imageAvailableSemaphore` et `renderFinishedSemaphore` ainsi que nos command buffers pour plusieurs frames en même temps.

Le plus simple est d'attendre que le logical device n'aie plus de travail à effectuer avant de lui en envoyer de nouveau, par exemple à l'aide de `vkQueueIdle` :

```
void drawFrame() {
    ...

    vkQueuePresentKHR(presentQueue, &presentInfo);

    vkQueueWaitIdle(presentQueue);
}
```

Cependant cette méthode n'est clairement pas optimale pour le GPU car la pipeline peut en général gérer plusieurs images à la fois grâce aux architectures massivement parallèles. Les étapes que l'image a déjà passées (par exemple le vertex shader quand elle en est au fragment shader) peuvent tout à fait être utilisées pour l'image suivante. Nous allons améliorer notre programme pour qu'il puisse supporter plusieurs images *en vol* (ou *in flight*) tout en limitant la quantité de commandes dans la queue.

Commencez par ajouter une constante en haut du programme qui définit le nombre de frames à traiter concurentiellement :

```
const int MAX_FRAMES_IN_FLIGHT = 2;
```

Chaque frame aura ses propres sémaphores :

```
std::vector<VkSemaphore> imageAvailableSemaphores;
std::vector<VkSemaphore> renderFinishedSemaphores;
```

La fonction `createSemaphores` doit être améliorée pour gérer la création de tout ceux-là :

```
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 d'une frame!");
        }
}
```

Ils doivent également être libérés à la fin du programme :

```
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 garder à portée de main l'indice de la frame en cours.

```
size_t currentFrame = 0;
```

La fonction `drawFrame` peut maintenant être modifiée pour utiliser les bons objets :

```
void drawFrame() {
    vkAcquireNextImageKHR(device, swapChain, UINT64_MAX,
        imageAvailableSemaphores[currentFrame], VK_NULL_HANDLE, &imageIndex);

    ...

    VkSemaphore waitSemaphores[] = {imageAvailableSemaphores[currentFrame]};

    ...

    VkSemaphore signalSemaphores[] = {renderFinishedSemaphores[currentFrame]};

    ...
}
```

Nous ne devons bien sûr pas oublier d'avancer à la frame suivante à chaque fois :

```
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 `MAX_FRAMES_IN_FLIGHT` est atteint.

Bien que nous ayons pas en place les objets facilitant le traitement de plusieurs frames simultanément, encore maintenant le GPU traite plus de `MAX_FRAMES_IN_FLIGHT` à la fois. Nous n'avons en effet qu'une synchronisation GPU-GPU mais pas de synchronisation CPU-GPU. Nous n'avons pas de moyen de savoir

que le travail sur telle ou telle frame est fini, ce qui a pour conséquence que nous pouvons nous retrouver à afficher une frame alors qu'elle est encore en traitement.

Pour la synchronisation CPU-GPU nous allons utiliser l'autre moyen fourni par Vulkan que nous avons déjà évoqué : les *fences*. Au lieu d'informer une certaine opération que tel signal devra être attendu avant de continuer, ce que les sémaphores permettent, les fences permettent au programme d'attendre l'exécution complète d'une opération. Nous allons créer une fence pour chaque frame :

```
std::vector<VkSemaphore> imageAvailableSemaphores;
std::vector<VkSemaphore> renderFinishedSemaphores;
std::vector<VkFence> inFlightFences;
size_t currentFrame = 0;
```

J'ai choisi de créer les fences avec les sémaphores et de renommer la fonction `createSemaphores` 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 la création des objets de synchronisation pour
                une frame!");
        }
    }
}
```

La création d'une `VkFence` est très similaire à la création d'un sémaphore. N'oubliez pas de libérer les fences :

```
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 voulons maintenant que `drawFrame` utilise les fences pour la synchronisation. L'appel à `vkQueueSubmit` inclut un paramètre optionnel qui permet de passer une fence. Celle-ci sera informée de la fin de l'exécution du command buffer. Nous pouvons interpréter ce signal comme la fin du rendu sur la frame.

```
void drawFrame() {
    ...

    if (vkQueueSubmit(graphicsQueue, 1, &submitInfo, inFlightFences[currentFrame]) !=
        VK_SUCCESS) {
        throw std::runtime_error("échec de l'envoi d'un command buffer!");
    }
    ...
}
```

La dernière chose qui nous reste à faire est de changer le début de `drawFrame` pour que la fonction attende le rendu de la frame précédente :

```
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 fences. Elle attend soit qu'une seule fence soit que toutes les fences déclarent être signalées avant de retourner. Le choix du mode d'attente se fait selon la valeur du quatrième paramètre. Avec `VK_TRUE` nous demandons d'attendre toutes les fences, même si cela ne fait bien sûr pas de différence vu que nous n'avons qu'une seule fence. Comme la fonction `vkAcquireNextImageKHR` cette fonction prend une durée en argument, que nous ignorons. Nous devons ensuite réinitialiser les fences manuellement à l'aide d'un appel à la fonction `vkResetFences`.

Si vous lancez le programme maintenant vous allez constater un comportement étrange. Plus rien ne se passe. Nous attendons qu'une fence soit signalée alors qu'elle n'a jamais été envoyée à aucune fonction. En effet les fences sont par défaut crées dans le mode non signalé. Comme nous appelons `vkWaitForFences` avant `vkQueueSubmit` notre première fence va créer une pause infinie. Pour empêcher cela nous devons initialiser les fences dans le mode signalé, et ce dès leur création :

```
void createSyncObjects() {
    ...

    VkFenceCreateInfo fenceInfo{};
    fenceInfo.sType = VK_STRUCTURE_TYPE_FENCE_CREATE_INFO;
    fenceInfo.flags = VK_FENCE_CREATE_SIGNALED_BIT;

    ...
}
```

La fuite de mémoire n'est plus, mais le programme ne fonctionne pas encore correctement. Si `MAX_FRAMES_IN_FLIGHT` est plus grand que le nombre d'images de la swapchain ou que `vkAcquireNextImageKHR` ne retourne pas les images dans l'ordre, alors il est possible que nous lancions le rendu dans une image qui est déjà *en vol*. Pour éviter ça, nous devons pour chaque image de la swapchain si une frame en vol est en train d'utiliser celle-ci. Cette correspondance permettra de suivre les images en vol par leur fences respective, de cette façon nous aurons immédiatement un objet de synchronisation à attendre avant qu'une nouvelle frame puisse utiliser cette image.

Tout d'abord, ajoutez une nouvelle liste nommée `imagesInFlight`:

```
std::vector<VkFence> inFlightFences;
std::vector<VkFence> imagesInFlight;
size_t currentFrame = 0;
```

Préparez-la dans `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 aucune frame n'utilise d'image, donc on peut explicitement l'initialiser à *pas de fence*. Maintenant, nous allons modifier `drawFrame` pour attendre la fin de n'importe quelle frame qui serait en train d'utiliser l'image qu'on nous assigné pour la nouvelle frame.

```
void drawFrame() {
    ...

    vkAcquireNextImageKHR(device, swapChain, UINT64_MAX,
        imageAvailableSemaphores[currentFrame], VK_NULL_HANDLE, &imageIndex);

    // Vérifier si une frame précédente est en train d'utiliser cette image (il y a une fence
        à attendre)
    if (imagesInFlight[imageIndex] != VK_NULL_HANDLE) {
        vkWaitForFences(device, 1, &imagesInFlight[imageIndex], VK_TRUE, UINT64_MAX);
    }
    // Marque l'image comme étant à nouveau utilisée par cette frame
    imagesInFlight[imageIndex] = inFlightFences[currentFrame];

    ...
}
```

Parce que nous avons maintenant plus d'appels à `vkWaitForFences`, les appels à `vkResetFences` doivent être **déplacés**. Le mieux reste de simplement l'appeler juste avant d'utiliser la fence:

```
void drawFrame() {
    ...

    vkResetFences(device, 1, &inFlightFences[currentFrame]);

    if (vkQueueSubmit(graphicsQueue, 1, &submitInfo, inFlightFences[currentFrame]) !=
        VK_SUCCESS) {
        throw std::runtime_error("échec de l'envoi d'un command buffer!");
    }

    ...
}
```

Nous avons implémenté tout ce qui est nécessaire à la synchronisation pour certifier qu'il n'y a pas plus de deux frames de travail dans la queue et que ces frames n'utilise pas accidentellement la même image. Notez qu'il est tout à fait normal pour d'autre parties du code, comme le nettoyage final, de se reposer sur des mécanismes de synchronisation plus durs comme `vkDeviceWaitIdle`. Vous devriez décider de la bonne approche à utiliser en vous basant sur vos besoins de performances.

Pour en apprendre plus sur la synchronisation rendez vous sur ces exemples complets par Khronos.

Conclusion

Un peu plus de 900 lignes plus tard nous avons enfin atteint le niveau où nous voyons des résultats à l'écran!! Créer un programme avec Vulkan est clairement un énorme travail, mais grâce au contrôle que cet API vous offre vous pouvez obtenir des performances énormes. Je ne peux que vous recommander de relire tout ce code et de vous assurer que vous visualisez bien tout les éléments mis en jeu. Nous allons maintenant construire sur ces acquis pour étendre les fonctionnalités de ce programme.

Dans le prochain chapitre nous allons voir une autre petite chose nécessaire à tout bon programme Vulkan.

Code C++ / Vertex shader / Fragment shader

Recréation de la swap chain

Introduction

Notre application nous permet maintenant d'afficher correctement un triangle, mais certains cas de figures ne sont pas encore correctement gérés. Il est possible que la surface d'affichage soit redimensionnée par l'utilisateur et que la swap chain ne soit plus parfaitement compatible. Nous devons faire en sorte d'être informés de tels changements pour pouvoir recréer la swap chain.

Recréer la swap chain

Créez la fonction `recreateSwapChain` qui appelle `createSwapChain` et toutes les fonctions de création d'objets dépendants 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 `vkDeviceIdle` car nous ne devons surtout pas toucher à des ressources en cours d'utilisation. La première chose à faire est bien sûr de recréer la swap chain. Les image views doivent être recrées également car elles dépendent des images de la swap chain. La render pass doit être recrée car elle dépend du format des images de la swap chain. Il est rare que le format des images de la swap chain soit altéré mais il n'est pas officiellement garanti qu'il reste le même, donc nous gérerons ce cas là. La pipeline dépend de la taille des images pour la configuration des rectangles de viewport et de ciseau, donc nous devons recréer la pipeline graphique. Il est possible d'éviter cela en faisant de la taille de ces rectangles des états dynamiques. Finalement, les framebuffers et les command buffers dépendent des images de la swap chain.

Pour être certains que les anciens objets sont bien détruits avant d'en créer de nouveaux, nous devrions créer une fonction dédiée à cela et que nous appellerons depuis `recreateSwapChain`. Créez donc `cleanupSwapChain` :

```
void cleanupSwapChain() {

}

void recreateSwapChain() {
    vkDeviceWaitIdle(device);

```

```
    cleanupSwapChain();

    createSwapChain();
    createImageViews();
    createRenderPass();
    createGraphicsPipeline();
    createFramebuffers();
    createCommandBuffers();
}
```

Nous allons déplacer le code de suppression depuis `cleanup` jusqu'à `cleanupSwapChain` :

```
void cleanupSwapChain() {
    for (auto framebuffer : swapChainFramebuffers) {
        vkDestroyFramebuffer(device, framebuffer, nullptr);
    }

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

    vkDestroySwapchainKHR(device, swapChain, nullptr);
}
```

Nous pouvons ensuite appeler cette nouvelle fonction depuis `cleanup` pour éviter la redondance de code :

```
void cleanup() {
    cleanupSwapChain();

    vkDestroyPipeline(device, graphicsPipeline, nullptr);
    vkDestroyPipelineLayout(device, pipelineLayout, nullptr);

    vkDestroyRenderPass(device, renderPass, nullptr);

    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) {
        DestroyDebugReportCallbackEXT(instance, callback, nullptr);
    }

    vkDestroySurfaceKHR(instance, surface, nullptr);
    vkDestroyInstance(instance, nullptr);

    glfwDestroyWindow(window);

    glfwTerminate();
}
```

Nous pourrions recréer la command pool à partir de rien mais ce serait du gâchis. J'ai préféré libérer les command buffers existants à l'aide de la fonction `vkFreeCommandBuffers`. Nous pouvons de cette manière réutiliser la même command pool mais changer les command buffers.

Pour bien gérer le redimensionnement de la fenêtre nous devons récupérer la taille actuelle du framebuffer qui lui est associé pour s'assurer que les images de la swap chain ont bien la nouvelle taille. Pour cela changez `chooseSwapExtent` afin que cette fonction prenne en compte la nouvelle taille réelle :

```
VkExtent2D chooseSwapExtent(const VkSurfaceCapabilitiesKHR& capabilities) {
    if (capabilities.currentExtent.width != std::numeric_limits<uint32_t>::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! Le problème cependant est que nous devons arrêter complètement l'affichage pendant la recréation alors que nous pourrions éviter que les frames en vol soient perdues. Pour cela vous devez passer l'ancienne swap chain en paramètre à `oldSwapChain` dans la structure `VkSwapchainCreateInfoKHR` et détruire cette ancienne swap chain dès que vous ne l'utilisez plus.

Swap chain non-optimales ou dépassées

Nous devons maintenant déterminer quand recréer la swap chain et donc quand appeler `recreateSwapChain`. Heureusement pour nous Vulkan nous indiquera quand la swap chain n'est plus adéquate au moment de la présentation. Les fonctions `vkAcquireNextImageKHR` et `vkQueuePresentKHR` peuvent pour cela retourner les valeurs suivantes :

- `VK_ERROR_OUT_OF_DATE_KHR` : la swap chain n'est plus compatible avec la surface de fenêtre et ne peut plus être utilisée pour l'affichage, ce qui arrive en général avec un redimensionnement de la fenêtre
- `VK_SUBOPTIMAL_KHR` : la swap chain peut toujours être utilisée pour présenter des images avec succès, mais les caractéristiques de la surface de fenêtre ne correspondent plus à celles de la swap chain

```
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 de la présentation d'une image à la swap chain!");
}
```

Si la swap chain se trouve être dépassée quand nous essayons d'acquérir une nouvelle image il ne nous est plus possible de présenter un quelconque résultat. Nous devons de ce fait aussitôt recréer la swap chain et tenter la présentation avec la frame suivante.

Vous pouvez aussi décider de recréer la swap chain si sa configuration n'est plus optimale, mais j'ai choisi de ne pas le faire ici car nous avons de toute façon déjà acquis l'image. Ainsi `VK_SUCCES` et `VK_SUBOPTIMAL_KHR`

sont considérés comme des indicateurs de 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("échec de la présentation d'une image!");
}

currentFrame = (currentFrame + 1) % MAX_FRAMES_IN_FLIGHT;
```

La fonction `vkQueuePresentKHR` retourne les mêmes valeurs avec la même signification. Dans ce cas nous recréons la swap chain si elle n'est plus optimale car nous voulons les meilleurs résultats possibles.

Explicitement gérer les redimensionnements

Bien que la plupart des drivers émettent automatiquement le code `VK_ERROR_OUT_OF_DATE_KHR` après qu'une fenêtre est redimensionnée, cela n'est pas garanti par le standard. Par conséquent nous devons explictement gérer ces cas de figure. Ajoutez une nouvelle variable qui indiquera que la fenêtre a été redimensionnée :

```
std::vector<VkFence> inFlightFences;
size_t currentFrame = 0;

bool framebufferResized = false;
```

La fonction `drawFrame` doit ensuite être modifiée 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 `vkQueuePresentKHR` pour que les sémaphores soient dans un état correct. Pour détecter les redimensionnements de la fenêtre nous n'avons qu'à mettre en place `glfwSetFrameBufferSizeCallback` qui nous informera d'un changement de la taille associée à la fenêtre :

```
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 car GLFW ne sait pas correctement appeler une fonction membre d'une classe avec `this`.

Nous récupérons une référence à la `GLFWwindow` dans la fonction de rappel que nous fournissons. De plus nous pouvons paramétrer un pointeur de notre choix qui sera accessible à toutes nos fonctions de rappel. Nous pouvons y mettre la classe elle-même.

```
window = glfwCreateWindow(WIDTH, HEIGHT, "Vulkan", nullptr, nullptr);
glfwSetWindowUserPointer(window, this);
glfwSetFramebufferSizeCallback(window, framebufferResizeCallback);
```

De cette manière nous pouvons changer la valeur de la variable servant d'indicateur des redimensionnements :

```
static void framebufferResizeCallback(GLFWwindow* window, int width, int height) {
    auto app = reinterpret_cast<HelloTriangleApplication*>(glfwGetWindowUserPointer(window));
    app->framebufferResized = true;
}
```

Lancez maintenant le programme et changez la taille de la fenêtre pour voir si tout se passe comme prévu.

Gestion de la minimisation de la fenêtre

Il existe un autre cas important où la swap chain peut devenir invalide : si la fenêtre est minimisée. Ce cas est particulier car il résulte en un framebuffer de taille `0`. Dans ce tutoriel nous mettrons en pause le programme jusqu'à ce que la fenêtre soit remise en avant-plan. À ce moment-là nous recréerons la swap chain.

```
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 `glfwGetFramebufferSize` prend en charge le cas où la taille est déjà correcte et `glfwWaitEvents` n'aurait rien à attendre.

Félicitations, vous avez codé un programme fonctionnel avec Vulkan! Dans le prochain chapitre nous allons supprimer les sommets du vertex shader et mettre en place un vertex buffer.

Code C++ / Vertex shader / Fragment shader

Vertex buffers

Description des entrées des sommets

Introduction

Dans les quatre prochains chapitres nous allons remplacer les sommets inscrits dans le vertex shader par un vertex buffer stocké dans la mémoire de la carte graphique. Nous commencerons par une manière simple de procéder en créant un buffer manipulable depuis le CPU et en y copiant des données avec `memcpy`. Puis nous verrons comment avantageusement utiliser un *staging buffer* pour accéder à de la mémoire de haute performance.

Vertex shader

Premièrement, changeons le vertex shader en retirant les coordonnées des sommets de son code. Elles seront maintenant stockés dans une variable. Elle sera liée au contenu du vertex buffer, ce qui est indiqué par le mot-clef `in`. Faisons de même avec la couleur.

```
#version 450

layout(location = 0) in vec2 inPosition;
layout(location = 1) in vec3 inColor;

layout(location = 0) out vec3 fragColor;

out gl_PerVertex {
    vec4 gl_Position;
};

void main() {
    gl_Position = vec4(inPosition, 0.0, 1.0);
    fragColor = inColor;
}
```

Les variables `inPosition` et `inColor` sont des *vertex attributes*. Ce sont des propriétés spécifiques du sommet à l'origine de l'invocation du shader. Ces données peuvent être de différentes natures, des couleurs aux coordonnées en passant par des coordonnées de texture. Recompilez ensuite le vertex shader.

Tout comme pour `fragColor`, les annotations de type `layout(location=x)` assignent un indice à l'entrée. Cet indice est utilisé depuis le code C++ pour les reconnaître. Il est important de savoir que certains types - comme les vecteurs de flottants de double précision (64 bits) - prennent deux emplacements. Voici un exemple d'une telle situation, où il est nécessaire de prévoir un écart entre deux entrés :

```
layout(location = 0) in dvec3 inPosition;
layout(location = 2) in vec3 inColor;
```

Vous pouvez trouver plus d'information sur les qualificateurs d'organisation sur le wiki.

Sommets

Nous déplaçons les données des sommets depuis le code du shader jusqu'au code C++. Commencez par inclure la librairie GLM, afin d'utiliser des vecteurs et des matrices. Nous allons utiliser ces types pour les vecteurs de position et de couleur.

```
#include <glm/glm.hpp>
```

Créez une nouvelle structure appelée `Vertex`. Elle possède deux attributs que nous utiliserons pour le vertex shader :

```
struct Vertex {
    glm::vec2 pos;
    glm::vec3 color;
};
```

GLM nous fournit des types très pratiques simulant les types utilisés par 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 utiliserons ensuite un tableau de structures pour représenter un ensemble de sommets. Nous utiliserons les mêmes couleurs et les mêmes positions qu'avant, mais elles seront combinées en un seul tableau d'objets.

Lier les descriptions

La prochaine étape consiste à indiquer à Vulkan comment passer ces données au shader une fois qu'elles sont stockées dans le GPU. Nous verrons plus tard comment les y stocker. Il y a deux types de structures que nous allons devoir utiliser.

Pour la première, appelée `VkVertexInputBindingDescription`, nous allons ajouter une fonction à `Vertex` qui renverra une instance de cette structure.

```
struct Vertex {
    glm::vec2 pos;
    glm::vec3 color;

    static VkVertexInputBindingDescription getBindingDescription() {
        VkVertexInputBindingDescription bindingDescription{};

        return bindingDescription;
    }
};
```

Un *vertex binding* décrit la lecture des données stockées en mémoire. Elle fournit le nombre d'octets entre les jeux de données et la manière de passer d'un ensemble de données (par exemple une coordonnée) au suivant. Elle permet à Vulkan de savoir comment extraire chaque jeu de données correspondant à une invocation du vertex shader du vertex buffer.

```
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'aurons besoin que d'un seul vertex binding. Le membre `binding` indique l'indice du vertex binding dans le tableau des bindings. Le paramètre `stride` fournit le nombre d'octets séparant les débuts de deux ensembles de données, c'est à dire l'écart entre les données devant être fournies à une invocation de vertex shader et celles devant être fournies à la suivante. Enfin `inputRate` peut prendre les valeurs suivantes :

- `VK_VERTEX_INPUT_RATE_VERTEX` : Passer au jeu de données suivante après chaque sommet
- `VK_VERTEX_INPUT_RATE_INSTANCE` : Passer au jeu de données suivantes après chaque instance

Nous n'utilisons pas d'*instanced rendering* donc nous utiliserons `VK_VERTEX_INPUT_RATE_VERTEX`.

Description des attributs

La seconde structure dont nous avons besoin est `VkVertexInputAttributeDescription`. Nous allons également en créer deux instances depuis une fonction membre de `Vertex` :

```
#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 de ces structures. Elles décrivent chacunes l'origine et la nature des données stockées dans une variable shader annotée du `location=x`, et la manière d'en déterminer les valeurs depuis les données extraites par le binding. Comme nous avons deux de ces variables, nous avons besoin de deux de ces structures. Voici ce qu'il faut remplir pour la position.

```
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 de la provenance des données du sommet qui mené à l'invocation du vertex shader, en lui fournissant le vertex binding qui les a extraites. Le paramètre `location` correspond à la valeur donnée à la directive `location` dans le code du vertex shader. Dans notre cas l'entrée `0` correspond à la position du sommet stockée dans un vecteur de floats de 32 bits.

Le paramètre `format` permet donc de décrire le type de donnée de l'attribut. Étonnement les formats doivent être indiqués avec des valeurs énumérées dont les noms semblent correspondre à des gradients de couleur :

- `float` : `VK_FORMAT_R32_SFLOAT`
- `vec2` : `VK_FORMAT_R32G32_SFLOAT`
- `vec3` : `VK_FORMAT_R32G32B32_SFLOAT`
- `vec4` : `VK_FORMAT_R32G32B32A32_SFLOAT`

Comme vous pouvez vous en douter il faudra utiliser le format dont le nombre de composants de couleurs correspond au nombre de données à transmettre. Il est autorisé d'utiliser plus de données que ce qui est prévu dans le shader, et ces données surnuméraires seront silencieusement ignorées. Si par contre il n'y a pas assez de valeurs les valeurs suivantes seront utilisées par défaut pour les valeurs manquantes : 0, 0 et 1 pour les deuxième, troisième et quatrième composantes. Il n'y a pas de valeur par défaut pour le premier membre car ce cas n'est pas autorisé. Les types (`SFLOAT`, `UINT` et `SINT`) et le nombre de bits doivent par contre correspondre parfaitement à ce qui est indiqué dans le shader. Voici quelques exemples :

- `ivec2` correspond à `VK_FORMAT_R32G32_SINT` et est un vecteur à deux composantes d'entiers signés de 32 bits

- `uvec4` correspond à `VK_FORMAT_R32G32B32A32_UINT` et est un vecteur à quatre composantes d'entiers non signés de 32 bits
- `double` correspond à `VK_FORMAT_R64_SFLOAT` et est un float à précision double (donc de 64 bits)

Le paramètre `format` définit implicitement la taille en octets des données. Mais le binding extrait dans notre cas deux données pour chaque sommet : la position et la couleur. Pour savoir quels octets doivent être mis dans la variable à laquelle la structure correspond, le paramètre `offset` permet d'indiquer de combien d'octets il faut se décaler dans les données extraites pour se trouver au début de la variable. Ce décalage est calculé automatiquement par la macro `offsetof`.

L'attribut de couleur est décrit de la même façon. Essayez de le remplir avant de regarder la solution ci-dessous.

```
attributeDescriptions[1].binding = 0;
attributeDescriptions[1].location = 1;
attributeDescriptions[1].format = VK_FORMAT_R32G32B32_SFLOAT;
attributeDescriptions[1].offset = offsetof(Vertex, color);
```

Entrée des sommets dans la pipeline

Nous devons maintenant mettre en place la réception par la pipeline graphique des données des sommets. Nous allons modifier une structure dans `createGraphicsPipeline`. Trouvez `vertexInputInfo` et ajoutez-y les références aux deux structures de description 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();
```

La pipeline peut maintenant accepter les données des vertices dans le format que nous utilisons et les fournir au vertex shader. Si vous lancez le programme vous verrez que les validation layers rapportent qu'aucun vertex buffer n'est mis en place. Nous allons donc créer un vertex buffer et y placer les données pour que le GPU puisse les utiliser.

Code C++ / Vertex shader / Fragment shader

Création de vertex buffers

Introduction

Les buffers sont pour Vulkan des emplacements mémoire qui peuvent permettre de stocker des données quelconques sur la carte graphique. Nous pouvons en particulier y placer les données représentant les sommets, et c'est ce que nous allons faire dans ce chapitre. Nous verrons plus tard d'autres utilisations répandues. Au contraire des autres objets que nous avons rencontré les buffers n'allouent pas eux-mêmes de mémoire. Il nous faudra gérer la mémoire à la main.

Création d'un buffer

Créez la fonction `createVertexBuffer` et appelez-la depuis `initVulkan` juste avant `createCommandBuffers`.

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

```
    createSurface();
    pickPhysicalDevice();
    createLogicalDevice();
    createSwapChain();
    createImageViews();
    createRenderPass();
    createGraphicsPipeline();
    createFramebuffers();
    createCommandPool();
    createVertexBuffer();
    createCommandBuffers();
    createSyncObjects();
}

...

void createVertexBuffer() {

}
```

Pour créer un buffer nous allons devoir remplir 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 buffer en octets. Nous pouvons utiliser `sizeof` pour déterminer la taille de notre tableau de valeur.

```
bufferInfo.usage = VK_BUFFER_USAGE_VERTEX_BUFFER_BIT;
```

Le deuxième champ, appelé `usage`, correspond à l'utilisation type du buffer. Nous pouvons indiquer plusieurs valeurs représentant les utilisations possibles. Dans notre cas nous ne mettons que la valeur qui correspond à un vertex buffer.

```
bufferInfo.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
```

De la même manière que les images de la swap chain, les buffers peuvent soit être gérés par une queue family, ou bien être partagés entre plusieurs queue families. Notre buffer ne sera utilisé que par la queue des graphismes, nous pouvons donc rester en mode exclusif.

Le paramètre `flags` permet de configurer le buffer tel qu'il puisse être constitué de plusieurs emplacements distincts dans la mémoire. Nous n'utiliserons pas cette fonctionnalité, laissez `flags` à `0`.

Nous pouvons maintenant créer le buffer en appelant `vkCreateBuffer`. Définissez un membre donnée pour stocker ce buffer :

```
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;
```

```
11
12      if (vkCreateBuffer(device, &bufferInfo, nullptr, &vertexBuffer) != VK_SUCCESS) {
13          throw std::runtime_error("echec de la creation d'un vertex buffer!");
14      }
15  }
```

Le buffer doit être disponible pour toutes les opérations de rendu, nous ne pouvons donc le détruire qu'à la fin du programme, et ce dans `cleanup` car il ne dépend pas de la swap chain.

```
1  void cleanup() {
2      cleanupSwapChain();
3
4      vkDestroyBuffer(device, vertexBuffer, nullptr);
5
6      ...
7  }
```

Fonctionnalités nécessaires de la mémoire

Le buffer a été créé mais il n'est lié à aucune forme de mémoire. La première étape de l'allocation de mémoire consiste à récupérer les fonctionnalités dont le buffer a besoin à l'aide de la fonction `vkGetBufferMemoryRequirements`.

```
1  VkMemoryRequirements memRequirements;
2  vkGetBufferMemoryRequirements(device, vertexBuffer, &memRequirements);
```

La structure que la fonction nous remplit possède trois membres :

- `size` : le nombre d'octets dont le buffer a besoin, ce qui peut différer de ce que nous avons écrit en préparant le buffer
- `alignment` : le décalage en octets entre le début de la mémoire allouée pour lui et le début des données du buffer, ce que le driver détermine avec les valeurs que nous avons fournies dans `usage` et `flags`
- `memoryTypeBits` : champs de bits combinant les types de mémoire qui conviennent au buffer

Les cartes graphiques offrent plusieurs types de mémoire. Ils diffèrent en performance et en opérations disponibles. Nous devons considérer ce dont le buffer a besoin en même temps que ce dont nous avons besoin pour sélectionner le meilleur type de mémoire possible. Créons une fonction `findMemoryType` pour y isoler cette logique.

```
1  uint32_t findMemoryType(uint32_t typeFilter, VkMemoryPropertyFlags properties) {
2
3  }
```

Nous allons commencer cette fonction en récupérant les différents types de mémoire que la carte graphique peut nous offrir.

```
1  VkPhysicalDeviceMemoryProperties memProperties;
2  vkGetPhysicalDeviceMemoryProperties(physicalDevice, &memProperties);
```

La structure `VkPhysicalDeviceMemoryProperties` comprend deux tableaux appelés `memoryHeaps` et `memoryTypes`. Une pile de mémoire (memory heap en anglais) correspond aux types physiques de mémoire. Par exemple la VRAM est une pile, de même que la RAM utilisée comme zone de swap si la VRAM est pleine en est une autre. Tous les autres types de mémoire stockés dans `memoryTypes` sont répartis dans ces piles. Nous n'allons pas utiliser la pile comme facteur de choix, mais vous pouvez imaginer l'impact sur la performance que cette distinction peut avoir.

Trouvons d'abord un type de mémoire correspondant au buffer :

```
for (uint32_t i = 0; i < memProperties.memoryTypeCount; i++) {
    if (typeFilter & (1 << i)) {
        return i;
    }
}

throw std::runtime_error("aucun type de memoire ne satisfait le buffer!");
```

Le paramètre `typeFilter` nous permettra d'indiquer les types de mémoire nécessaires au buffer lors de l'appel à la fonction. Ce champ de bit voit son n-ième bit mis à `1` si le n-ième type de mémoire disponible lui convient. Ainsi nous pouvons itérer sur les bits de `typeFilter` pour trouver les types de mémoire qui lui correspondent.

Cependant cette vérification ne nous est pas suffisante. Nous devons vérifier que la mémoire est accesible depuis le CPU afin de pouvoir y écrire les données des vertices. Nous devons pour cela vérifier que le champ de bits `properyFlags` comprend au moins `VK_MEMORY_PROPERTY_HOSY_VISIBLE_BIT`, de même que `VK_MEMORY_PROPERTY_HOSY_COHERENT_BIT`. Nous verrons pourquoi cette deuxième valeur est nécessaire quand nous lierons de la mémoire au buffer.

Nous placerons ces deux valeurs dans le paramètre `properties`. Nous pouvons changer la boucle pour qu'elle prenne en compte le champ de bits :

```
for (uint32_t i = 0; i < memProperties.memoryTypeCount; i++) {
    if ((typeFilter & (1 << i)) && (memProperties.memoryTypes[i].propertyFlags & properties)
        == properties) {
        return i;
    }
}
```

Le ET bit à bit fournit une valeur non nulle si et seulement si au moins l'une des propriétés est supportée. Nous ne pouvons nous satisfaire de cela, c'est pourquoi il est nécessaire de comparer le résultat au champ de bits complet. Si ce résultat nous convient, nous pouvons retourner l'indice de la mémoire et utiliser cet emplacement. Si aucune mémoire ne convient nous levons une exception.

Allocation de mémoire

Maintenant que nous pouvons déterminer un type de mémoire nous convenant, nous pouvons y allouer de la mémoire. Nous devons pour cela remplir 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 nous suffit d'indiquer une taille et un type, ce que nous avons déjà déterminé. Créez un membre donnée pour contenir la référence à l'espace mémoire et allouez-le à l'aide de `vkAllocateMemory`.

```
VkBuffer vertexBuffer;
VkDeviceMemory vertexBufferMemory;

...
if (vkAllocateMemory(device, &allocInfo, nullptr, &vertexBufferMemory) != VK_SUCCESS) {
    throw std::runtime_error("echec d'une allocation de memoire!");
}
```

Si l'allocation a réussi, nous pouvons associer cette mémoire au buffer 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 buffer. Nous avons alloué cette mémoire spécialement pour ce buffer, nous pouvons donc mettre `0`. Si vous décidez d'allouer un grand espace mémoire pour y mettre plusieurs buffers, sachez qu'il faut que ce nombre soit divisible par `memRequirements.alignement`. Notez que cette stratégie est la manière recommandée de gérer la mémoire des GPUs (voyez cet article).

Il est évident que cette allocation dynamique de mémoire nécessite que nous libérions l'emplacement nous-mêmes. Comme la mémoire est liée au buffer, et que le buffer sera nécessaire à toutes les opérations de rendu, nous ne devons la libérer qu'à la fin du programme.

```
void cleanup() {
    cleanupSwapChain();

    vkDestroyBuffer(device, vertexBuffer, nullptr);
    vkFreeMemory(device, vertexBufferMemory, nullptr);
```

Remplissage du vertex buffer

Il est maintenant temps de placer les données des vertices dans le buffer. Nous allons mapper la mémoire dans un emplacement accessible par le CPU à l'aide de la fonction `vkMapMemory`.

```
void* data;
vkMapMemory(device, vertexBufferMemory, 0, bufferInfo.size, 0, &data);
```

Cette fonction nous permet d'accéder à une région spécifique d'une ressource. Nous devons pour cela indiquer un décalage et une taille. Nous mettons ici respectivement `0` et `bufferInfo.size`. Il est également possible de fournir la valeur `VK_WHOLE_SIZE` pour mapper d'un coup toute la ressource. L'avant-dernier paramètre est un champ de bits pour l'instant non implémenté par Vulkan. Il est impératif de la laisser à `0`. Enfin, le dernier paramètre permet de fournir un pointeur vers la mémoire ainsi mappée.

```
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 `memcpy` pour copier les vertices dans la mémoire, puis démapper le buffer à l'aide de `vkUnmapMemory`. Malheureusement le driver peut décider de cacher les données avant de les copier dans le buffer. Il est aussi possible que les données soient copiées mais que ce changement ne soit pas visible immédiatement. Il y a deux manières de régler ce problème :

- Utiliser une pile de mémoire cohérente avec la RAM, ce qui est indiqué par `VK_MEMORY_PROPERTY_HOST_COHERENT_BIT`
- Appeler `vkFlushMappedMemoryRanges` après avoir copié les données, puis appeler `vkInvalidateMappedMemory` avant d'accéder à la mémoire

Nous utiliserons la première approche qui nous assure une cohérence permanente. Cette méthode est moins performante que le flushing explicite, mais nous verrons dès le prochain chapitre que cela n'a aucune importance car nous changerons complètement de stratégie.

Par ailleurs, notez que l'utilisation d'une mémoire cohérente ou le flushing de la mémoire ne garantissent que le fait que le driver soit au courant des modifications de la mémoire. La seule garantie est que le déplacement se finisse d'ici le prochain appel à `vkQueueSubmit`.

Remarquez également l'utilisation de `memcpy` qui indique la compatibilité bit-à-bit des structures avec la représentation sur la carte graphique.

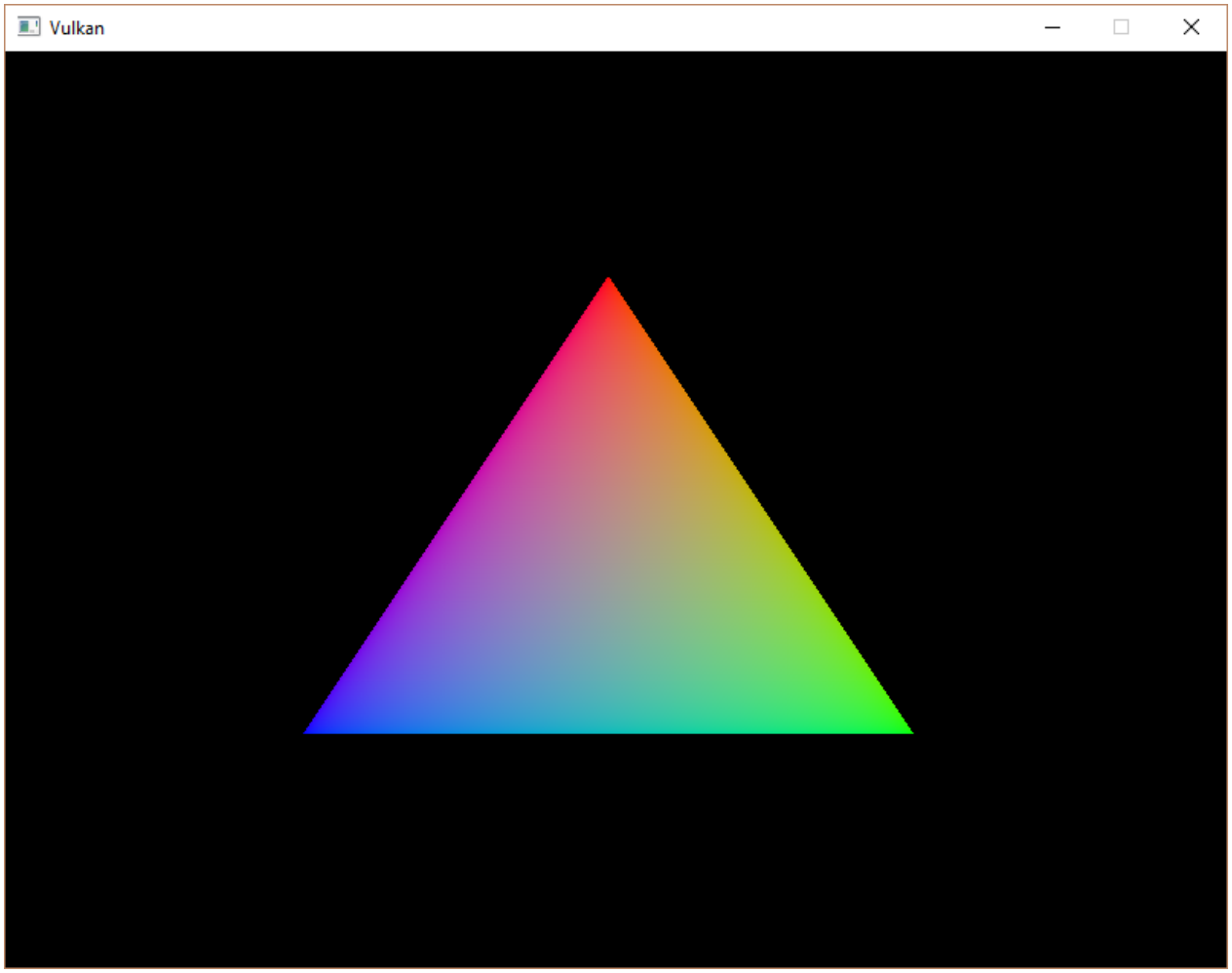
Lier le vertex buffer

Il ne nous reste qu'à lier le vertex buffer pour les 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` lie des vertex buffers aux bindings. Les deuxième et troisième paramètres indiquent l'indice du premier binding auquel le buffer correspond et le nombre de bindings qu'il contiendra. L'avant-dernier paramètre est le tableau de vertex buffers à lier, et le dernier est un tableau de décalages en octets entre le début d'un buffer et le début des données. Il est d'ailleurs préférable d'appeler `vkCmdDraw` avec la taille du tableau de vertices plutôt qu'avec un nombre écrit à la main.

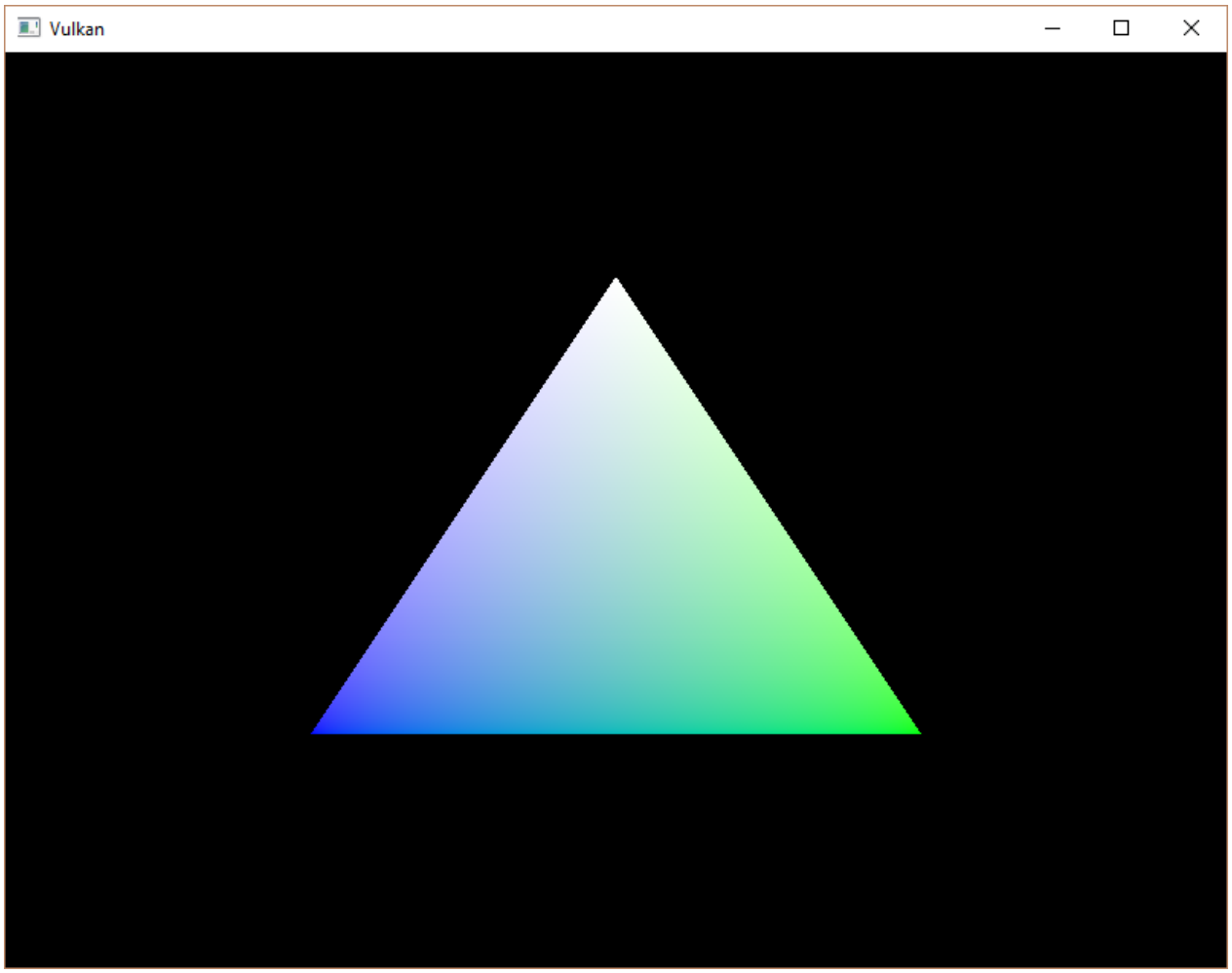
Lancez maintenant le programme; vous devriez voir le triangle habituel apparaître à l'écran.



Essayez de colorer le vertex du haut en blanc et relancez le programme :

```
const std::vector<Vertex> vertices = {
    {{0.0f, -0.5f}, {1.0f, 1.0f, 1.0f}},
```

```
3     {{0.5f, 0.5f}, {0.0f, 1.0f, 0.0f}},
4     {{-0.5f, 0.5f}, {0.0f, 0.0f, 1.0f}}
5 };
```



Dans le prochain chapitre nous verrons une autre manière de copier les données vers un buffer. Elle est plus performante mais nécessite plus de travail.

Code C++ / Vertex shader / Fragment shader

Buffer intermédiaire

Introduction

Nous avons maintenant un vertex buffer fonctionnel. Par contre il n'est pas dans la mémoire la plus optimale posible pour la carte graphique. Il serait préférable d'utiliser une mémoire `VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT`, mais de telles mémoires ne sont pas accessibles depuis le CPU. Dans ce chapitre nous allons créer deux vertex buffers. Le premier, un buffer intermédiaire (*staging buffer*), sera stocké dans de la mémoire accessible depuis le CPU, et nous y mettrons nos données. Le second sera directement dans la carte graphique, et nous y copierons les données des vertices depuis le buffer intermédiaire.

Queue de transfert

La commande de copie des buffers provient d'une queue family qui supporte les opérations de transfert, ce qui est indiqué par `VK_QUEUE_TRANFER_BIT`. Une bonne nouvelle : toute queue qui supporte les graphismes ou le calcul doit supporter les transferts. Par contre il n'est pas obligatoire pour ces queues de l'indiquer dans le champ de bit qui les décrit.

Si vous aimez la difficulté, vous pouvez préférer l'utilisation d'une queue spécifique aux opérations de transfert. Vous aurez alors ceci à changer :

- Modifier la structure `QueueFamilyIndices` et la fonction `findQueueFamilies` pour obtenir une queue family dont la description comprend `VK_QUEUE_TRANSFER_BIT` mais pas `VK_QUEUE_GRAPHICS_BIT`
- Modifier `createLogicalDevice` pour y récupérer une référence à une queue de transfert
- Créer une command pool pour les command buffers envoyés à la queue de transfert
- Changer la valeur de `sharingMode` pour les ressources qui le demandent à `VK_SHARING_MODE_CONCURRENT`, et indiquer à la fois la queue des graphismes et la queue ds transferts
- Émettre toutes les commandes de transfert telles `vkCmdCopyBuffer` - nous allons l'utiliser dans ce chapitre - à la queue de transfert au lieu de la queue des graphismes

Cela représente pas mal de travail, mais vous en apprendrez beaucoup sur la gestion des resources entre les queue families.

Abstraction de la création des buffers

Comme nous allons créer plusieurs buffers, il serait judicieux de placer la logique dans une fonction. Appelez-la `createBuffer` et déplacez-y le code suivant :

```
1  void createBuffer(VkDeviceSize size, VkBufferUsageFlags usage, VkMemoryPropertyFlags
       properties, VkBuffer& buffer, VkDeviceMemory& bufferMemory) {
2      VkBufferCreateInfo bufferInfo{};
3      bufferInfo.sType = VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO;
4      bufferInfo.size = size;
5      bufferInfo.usage = usage;
6      bufferInfo.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
7
8      if (vkCreateBuffer(device, &bufferInfo, nullptr, &buffer) != VK_SUCCESS) {
9          throw std::runtime_error("echec de la creation d'un buffer!");
10     }
11
12     VkMemoryRequirements memRequirements;
13     vkGetBufferMemoryRequirements(device, buffer, &memRequirements);
14
15     VkMemoryAllocateInfo allocInfo{};
16     allocInfo.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
17     allocInfo.allocationSize = memRequirements.size;
18     allocInfo.memoryTypeIndex = findMemoryType(memRequirements.memoryTypeBits, properties);
19
20     if (vkAllocateMemory(device, &allocInfo, nullptr, &bufferMemory) != VK_SUCCESS) {
21         throw std::runtime_error("echec de l'allocation de memoire!");
22     }
23
24     vkBindBufferMemory(device, buffer, bufferMemory, 0);
25 }
```

Cette fonction nécessite plusieurs paramètres, tels que la taille du buffer, les propriétés dont nous avons besoin et l'utilisation type du buffer. La fonction a deux résultats, elle fonctionne donc en modifiant la valeur des deux derniers paramètres, dans lesquels elle place les référernces aux objets créés.

Vous pouvez maintenant supprimer la création du buffer et l'allocation de la mémoire de `createVertexBuffer` et remplacer tout ça par un appel à votre 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 tout fonctionne toujours aussi bien.

Utiliser un buffer intermédiaire

Nous allons maintenant faire en sorte que `createVertexBuffer` utilise d'abord un buffer visible pour copier les données sur la carte graphique, puis qu'il utilise de la mémoire locale à la carte graphique pour le véritable buffer.

```
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 utilisons ainsi un nouveau `stagingBuffer` lié à la `stagingBufferMemory` pour transmettre les données à la carte graphique. Dans ce chapitre nous allons utiliser deux nouvelles valeurs pour les utilisations des buffers :

- `VK_BUFFER_USAGE_TRANSFER_SCR_BIT` : le buffer peut être utilisé comme source pour un transfert de mémoire
- `VK_BUFFER_USAGE_TRANSFER_DST_BIT` : le buffer peut être utilisé comme destination pour un transfert de mémoire

Le `vertexBuffer` est maintenant alloué à partir d'un type de mémoire local au device, ce qui implique en général que nous ne pouvons pas utiliser `vkMapMemory`. Nous pouvons cependant bien sûr y copier les données depuis le buffer intermédiaire. Nous pouvons indiquer que nous voulons transmettre des données entre ces buffers à l'aide des valeurs que nous avons vues juste au-dessus. Nous pouvons combiner ces informations avec par exemple `VK_BUFFER_USAGE_VERTEX_BUFFER_BIT`.

Nous allons maintenant écrire la fonction `copyBuffer`, qui servira à recopier le contenu du buffer intermédiaire dans le véritable buffer.

```
void copyBuffer(VkBuffer srcBuffer, VkBuffer dstBuffer, VkDeviceSize size) {

}
```

Les opérations de transfert de mémoire sont réalisées à travers un command buffer, comme pour l'affichage. Nous devons commencer par allouer des command buffers temporaires. Vous devriez d'ailleurs utiliser une autre command pool pour tous ces command buffer temporaires, afin de fournir à l'implémentation une occasion d'optimiser la gestion de la mémoire séparément des graphismes. Si vous le faites, utilisez `VK_COMMAND_POOL_CREATE_TRANSIENT_BIT` pendant la création de la command pool, car les commands buffers ne seront utilisés qu'une seule fois.

```
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 command buffer :

```
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 command buffer une fois seulement, et attendre que la copie soit terminée avant de sortir de la fonction. Il est alors préférable d'informer le driver de cela 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`. Elle prend les buffers de source et d'arrivée comme arguments, et un tableau des régions à copier. Ces régions sont décrites dans des structures de type `VkBufferCopy`, qui consistent en un décalage dans le buffer source, le nombre d'octets à copier et le décalage dans le buffer d'arrivée. Il n'est ici pas possible d'indiquer `VK_WHOLE_SIZE`.

```
vkEndCommandBuffer(commandBuffer);
```

Ce command buffer ne sert qu'à réaliser les copies des buffers, nous pouvons donc arrêter l'enregistrement dès maintenant. Exécutez le command buffer pour compléter le transfert :

```
VkSubmitInfo submitInfo{};
submitInfo.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;
submitInfo.commandBufferCount = 1;
submitInfo.pCommandBuffers = &commandBuffer;

```

```
6 vkQueueSubmit(graphicsQueue, 1, &submitInfo, VK_NULL_HANDLE);
7 vkQueueWaitIdle(graphicsQueue);
```

Au contraire des commandes d'affichage très complexes, il n'y a pas de synchronisation particulière à mettre en place. Nous voulons simplement nous assurer que le transfert se réalise immédiatement. Deux possibilités s'offrent alors à nous : utiliser une fence et l'attendre avec `vkWaitForFences`, ou simplement attendre avec `vkQueueWaitIdle` que la queue des transfert soit au repos. Les fences permettent de préparer de nombreux transferts pour qu'ils s'exécutent concurentiellement, et offrent au driver encore une manière d'optimiser le travail. L'autre méthode a l'avantage de la simplicité. Implémentez le système de fence si vous le désirez, mais cela vous obligera à modifier l'organisation de ce module.

```
1 vkFreeCommandBuffers(device, commandPool, 1, &commandBuffer);
```

N'oubliez pas de libérer le command buffer utilisé pour l'opération de transfert.

Nous pouvons maintenant appeler `copyBuffer` depuis la fonction `createVertexBuffer` pour que les sommets soient enfin stockées dans la mémoire locale.

```
1 createBuffer(bufferSize, VK_BUFFER_USAGE_TRANSFER_DST_BIT |
      VK_BUFFER_USAGE_VERTEX_BUFFER_BIT, VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT, vertexBuffer,
      vertexBufferMemory);
2
3 copyBuffer(stagingBuffer, vertexBuffer, bufferSize);
```

Maintenant que les données sont dans la carte graphique, nous n'avons plus besoin du buffer intermédiaire, et devons donc le détruire.

```
1     ...
2
3     copyBuffer(stagingBuffer, vertexBuffer, bufferSize);
4
5     vkDestroyBuffer(device, stagingBuffer, nullptr);
6     vkFreeMemory(device, stagingBufferMemory, nullptr);
7 }
```

Lancez votre programme pour vérifier que vous voyez toujours le même triangle. L'amélioration n'est peut-être pas flagrante, mais il est clair que la mémoire permet d'améliorer les performances, préparant ainsi le terrain pour le chargement de géométrie plus complexe.

Conclusion

Notez que dans une application réelle, vous ne devez pas allouer de la mémoire avec `vkAllocateMemory` pour chaque buffer. De toute façon le nombre d'appel à cette fonction est limité, par exemple à 4096, et ce même sur des cartes graphiques comme les GTX 1080. La bonne pratique consiste à allouer une grande zone de mémoire et d'utiliser un gestionnaire pour créer des décalages pour chacun des buffers. Il est même préférable d'utiliser un buffer pour plusieurs types de données (sommets et uniformes par exemple) et de séparer ces types grâce à des indices dans le buffer (voyez encore ce même article).

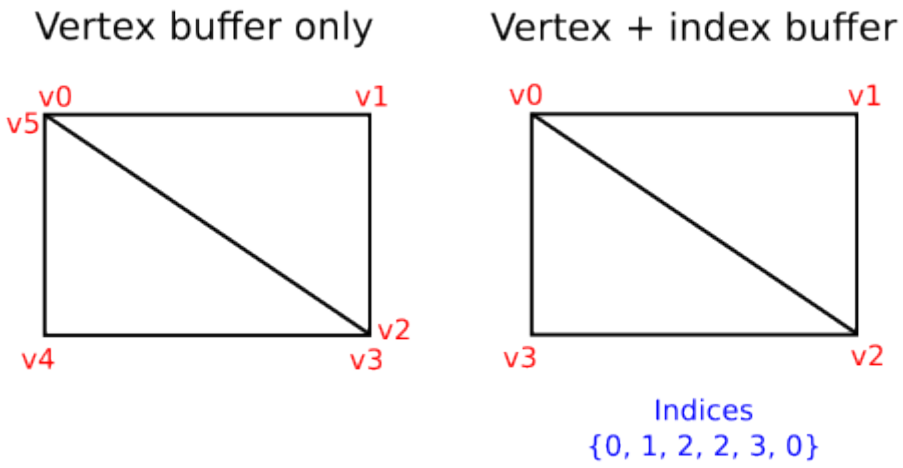
Vous pouvez implémenter votre propre solution, ou bien utiliser la librairie VulkanMemoryAllocator crée par GPUOpen. Pour ce tutoriel, ne vous inquiétez pas pour cela car nous n'atteindrons pas cette limite.

Code C++ / Vertex shader / Fragment shader

Index buffer

Introduction

Les modèles 3D que vous serez susceptibles d'utiliser dans des applications réelles partagerons le plus souvent des vertices communs à plusieurs triangles. Cela est d'ailleurs le cas avec un simple rectangle :



Un rectangle est composé de triangles, ce qui signifie que nous aurions besoin d'un vertex buffer avec 6 vertices. Mais nous dupliquerions alors des vertices, aboutissant à un gachis de mémoire. Dans des modèles plus complexes, les vertices sont en moyenne en contact avec 3 triangles, ce qui serait encore pire. La solution consiste à utiliser un index buffer.

Un index buffer est essentiellement un tableau de références vers le vertex buffer. Il vous permet de réordonner ou de dupliquer les données de ce buffer. L'image ci-dessus démontre l'utilité de cette méthode.

Création d'un index buffer

Dans ce chapitre, nous allons ajouter les données nécessaires à l'affichage d'un rectangle. Nous allons ainsi rajouter une coordonnée dans le vertex buffer et créer un index buffer. Voici les données des sommets au complet :

```
1 const std::vector<Vertex> vertices = {
2     {{-0.5f, -0.5f}, {1.0f, 0.0f, 0.0f}},
3     {{0.5f, -0.5f}, {0.0f, 1.0f, 0.0f}},
4     {{0.5f, 0.5f}, {0.0f, 0.0f, 1.0f}},
5     {{-0.5f, 0.5f}, {1.0f, 1.0f, 1.0f}}
6 };
```

Le coin en haut à 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. Les couleurs seront dégradées par l'interpolation du rasterizer. Nous allons maintenant créer le tableau `indices` pour représenter l'index buffer. Son contenu correspond à ce qui est présenté dans l'illustration.

```
1 const std::vector<uint16_t> indices = {
2     0, 1, 2, 2, 3, 0
3 };
```

Il est possible d'utiliser `uint16_t` ou `uint32_t` pour les valeurs de l'index buffer, en fonction du nombre d'éléments dans `vertices`. Nous pouvons nous contenter de `uint16_t` car nous n'utilisons pas plus de 65535 sommets différents.

Comme les données des sommets, nous devons placer les indices dans un `VkBuffer` pour que le GPU puisse y avoir accès. Créez deux membres donnée pour référencer les ressources du futur index buffer :

```
VkBuffer vertexBuffer;
VkDeviceMemory vertexBufferMemory;
VkBuffer indexBuffer;
VkDeviceMemory indexBufferMemory;
```

La fonction `createIndexBuffer` est quasiment identique à `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 : `bufferSize` correspond à la taille du tableau multiplié par `sizeof(uint16_t)`, et `VK_BUFFER_USAGE_VERTEX_BUFFER_BIT` est remplacé par `VK_BUFFER_USAGE_INDEX_BUFFER_BIT`. À part ça tout est identique : nous créons un buffer intermédiaire puis le copions dans le buffer final local au GPU.

L'index buffer doit être libéré à la fin du programme depuis `cleanup`.

```
void cleanup() {
    cleanupSwapChain();

    vkDestroyBuffer(device, indexBuffer, nullptr);
    vkFreeMemory(device, indexBufferMemory, nullptr);

```

```
    vkDestroyBuffer(device, vertexBuffer, nullptr);
    vkFreeMemory(device, vertexBufferMemory, nullptr);

    ...
}
```

Utilisation d'un index buffer

Pour utiliser l'index buffer lors des opérations de rendu nous devons modifier un petit peu `createCommandBuffers`. Tout d'abord il nous faut lier l'index buffer. La différence est qu'il n'est pas possible d'avoir plusieurs index buffers. De plus il n'est pas possible de subdiviser les sommets en leurs coordonnées, ce qui implique que la modification d'une seule coordonnée nécessite de créer un autre sommet le vertex buffer.

```
vkCmdBindVertexBuffers(commandBuffers[i], 0, 1, vertexBuffers, offsets);

vkCmdBindIndexBuffer(commandBuffers[i], indexBuffer, 0, VK_INDEX_TYPE_UINT16);
```

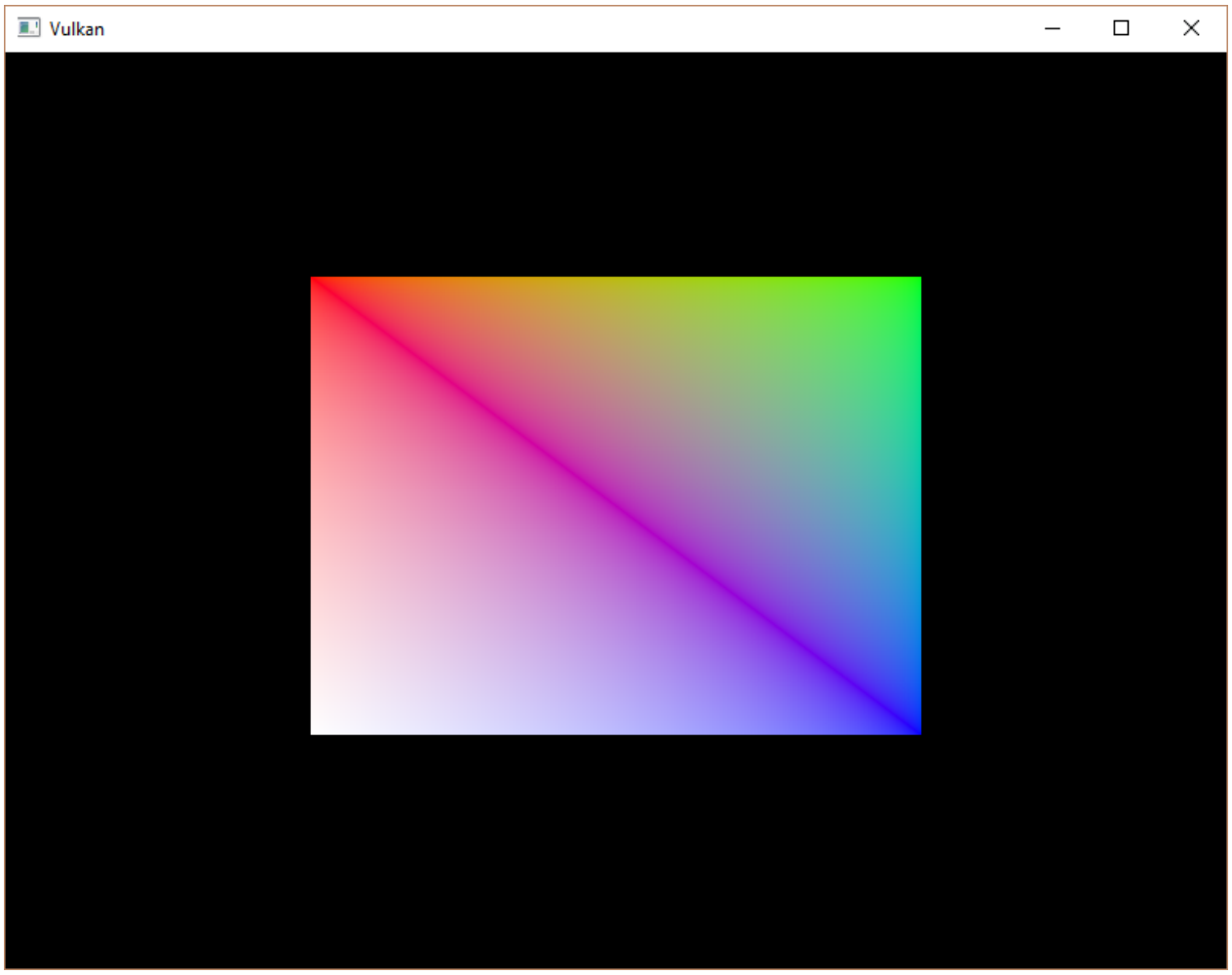
Un index buffer est lié par la fonction `vkCmdBindIndexBuffer`. Elle prend en paramètres le buffer, le décalage dans ce buffer et le type de donnée. Pour nous ce dernier sera `VK_INDEX_TYPE_UINT16`.

Simplement lier le vertex buffer ne change en fait rien. Il nous faut aussi mettre à jour les commandes d'affichage pour indiquer à Vulkan comment utiliser le buffer. Supprimez l'appel à `vkCmdDraw`, et remplacez-le par `vkCmdDrawIndexed` :

```
vkCmdDrawIndexed(commandBuffers[i], static_cast<uint32_t>(indices.size()), 1, 0, 0, 0);
```

Le deuxième paramètre indique le nombre d'indices. Le troisième est le nombre d'instances à invoquer (ici `1` car nous n'utilisons par cette technique). Le paramètre suivant est un décalage dans l'index buffer, sachant qu'ici il ne fonctionne pas en octets mais en indices. L'avant-dernier paramètre permet de fournir une valeur qui sera ajoutée à tous les indices juste avant de les faire correspondre aux vertices. Enfin, le dernier paramètre est un décalage pour le rendu instancié.

Lancez le programme et vous devriez avoir ceci :



Vous savez maintenant économiser la mémoire en réutilisant les vertices à l'aide d'un index buffer. Cela deviendra crucial pour les chapitres suivants dans lesquels vous allez apprendre à charger des modèles complexes.

Nous avons déjà évoqué le fait que le plus de buffers possibles devraient être stockés dans un seul emplacement mémoire. Il faudrait dans l'idéal allez encore plus loin : les développeurs des drivers recommandent également que vous placiez plusieurs buffers dans un seul et même `VkBuffer`, et que vous utilisiez des décalages pour les différencier dans les fonctions comme `vkCmdBindVertexBuffers`. Cela simplifie la mise des données dans des caches car elles sont regroupées en un bloc. Il devient même possible d'utiliser la même mémoire pour plusieurs ressources si elles ne sont pas utilisées en même temps et si elles sont proprement mises à jour. Cette pratique s'appelle d'ailleurs *aliasing*, et certaines fonctions Vulkan possèdent un paramètre qui permet au développeur d'indiquer s'il veut utiliser la technique.

Code C++ / Vertex shader / Fragment shader

Uniform buffers

Descriptor layout et buffer

Introduction

Nous pouvons maintenant passer des données à chaque groupe d'invocation de vertex shaders. Mais qu'en est-il des variables globales? Nous allons enfin passer à la 3D, et nous avons besoin d'une matrice model-view-projection. Nous pourrions la transmettre avec les vertices, mais cela serait un gachis de mémoire et, de plus, nous devrions mettre à jour le vertex buffer à chaque frame, alors qu'il est très bien rangé dans se mémoire à hautes performances.

La solution fournie par Vulkan consiste à utiliser des *descripteurs de ressource* (ou *resource descriptors*), qui font correspondre des données en mémoire à une variable shader. Un descripteur permet à des shaders d'accéder librement à des ressources telles que les buffers ou les *images*. Attention, Vulkan donne un sens particulier au terme image. Nous verrons cela bientôt. Nous allons pour l'instant créer un buffer qui contiendra les matrices de transformation. Nous ferons en sorte que le vertex shader puisse y accéder. Il y a trois parties à l'utilisation d'un descripteur de ressources :

- Spécifier l'organisation des descripteurs durant la création de la pipeline
- Allouer un set de descripteurs depuis une pool de descripteurs (encore un objet de gestion de mémoire)
- Lier le descripteur pour les opérations de rendu

L'*organisation du descripteur* (descriptor layout) indique le type de ressources qui seront accédées par la pipeline. Cela ressemble sur le principe à indiquer les attachements accédés. Un *set de descripteurs* (descriptor set) spécifie le buffer ou l'image qui sera lié à ce descripteur, de la même manière qu'un framebuffer doit indiquer les ressources qui le composent.

Il existe plusieurs types de descripteurs, mais dans ce chapitre nous ne verrons que les *uniform buffer objects* (UBO). Nous en verrons d'autres plus tard, et leur utilisation sera très similaire. Rentrons dans le vif du sujet et supposons maintenant que nous voulons que toutes les invocations du vertex shader que nous avons codé accèdent à la structure C suivante :

```
struct UniformBufferObject {
    glm::mat4 model;
    glm::mat4 view;
    glm::mat4 proj;
};
```

Nous devons la copier dans un `VkBuffer` pour pouvoir y accéder à l'aide d'un descripteur UBO depuis le vertex shader. De son côté le vertex shader y fait référence ainsi :

```
layout(binding = 0) uniform UniformBufferObject {
    mat4 model;
    mat4 view;
    mat4 proj;
} ubo;
```

```
6
7 void main() {
8     gl_Position = ubo.proj * ubo.view * ubo.model * vec4(inPosition, 0.0, 1.0);
9     fragColor = inColor;
10 }
```

Nous allons mettre à jour les matrices model, view et projection à chaque frame pour que le rectangle tourne sur lui-même et donne un effet 3D à la scène.

Vertex shader

Modifiez le vertex shader pour qu'il inclue l'UBO comme dans l'exemple ci-dessous. Je pars du principe que vous connaissez les transformations MVP. Si ce n'est pourtant pas le cas, vous pouvez vous rendre sur ce site déjà mentionné dans le premier chapitre.

```
1 #version 450
2
3 layout(binding = 0) uniform UniformBufferObject {
4     mat4 model;
5     mat4 view;
6     mat4 proj;
7 } ubo;
8
9 layout(location = 0) in vec2 inPosition;
10 layout(location = 1) in vec3 inColor;
11
12 layout(location = 0) out vec3 fragColor;
13
14 out gl_PerVertex {
15     vec4 gl_Position;
16 };
17
18 void main() {
19     gl_Position = ubo.proj * ubo.view * ubo.model * vec4(inPosition, 0.0, 1.0);
20     fragColor = inColor;
21 }
```

L'ordre des variables `in`, `out` et `uniform` n'a aucune importance. La directive `binding` est assez semblable à `location` ; elle permet de fournir l'indice du binding. Nous allons l'indiquer dans l'organisation du descripteur. Notez le changement dans la ligne calculant `gl_Position`, qui prend maintenant en compte la matrice MVP. La dernière composante du vecteur ne sera plus à `0`, car elle sert à diviser les autres coordonnées en fonction de leur distance à la caméra pour créer un effet de profondeur.

Organisation du set 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.

```
1 struct UniformBufferObject {
2     glm::mat4 model;
3     glm::mat4 view;
4     glm::mat4 proj;
5 };
```

Nous pouvons faire correspondre parfaitement la déclaration en C++ avec celle dans le shader grâce à GLM. De plus les matrices sont stockées d'une manière compatible bit à bit avec l'interprétation de ces données par les shaders. Nous pouvons ainsi utiliser `memcpy` sur une structure `UniformBufferObject` vers un `VkBuffer`.

Nous devons fournir des informations sur chacun des descripteurs utilisés par les shaders lors de la création de la pipeline, similairement aux entrées du vertex shader. Nous allons créer une fonction pour gérer toute cette information, et ainsi pour créer le set de descripteurs. Elle s'appelera `createDescriptorSetLayout` et sera appelée juste avant la finalisation de la création de la pipeline.

```
void initVulkan() {
    ...
    createDescriptorSetLayout();
    createGraphicsPipeline();
    ...
}

...

void createDescriptorSetLayout() {

}
```

Chaque `binding` doit être décrit à l'aide d'une structure de type `VkDescriptorSetLayoutBinding`.

```
void createDescriptorSetLayout() {
    VkDescriptorSetLayoutBinding uboLayoutBinding{};
    uboLayoutBinding.binding = 0;
    uboLayoutBinding.descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
    uboLayoutBinding.descriptorCount = 1;
}
```

Les deux premiers champs permettent de fournir la valeur indiquée dans le shader avec `binding` et le type de descripteur auquel il correspond. Il est possible que la variable côté shader soit un tableau d'UBO, et dans ce cas il faut indiquer le nombre d'éléments qu'il contient dans le membre `descriptorCount`. Cette possibilité pourrait être utilisée pour transmettre d'un coup toutes les transformations spécifiques aux différents éléments d'une structure hiérarchique. Nous n'utilisons pas cette possiblité et indiquons donc `1`.

```
uboLayoutBinding.stageFlags = VK_SHADER_STAGE_VERTEX_BIT;
```

Nous devons aussi informer Vulkan des étapes shaders qui accèderont à cette ressource. Le champ de bits `stageFlags` permet de combiner toutes les étapes shader concernées. Vous pouvez aussi fournir la valeur `VK_SHADER_STAGE_ALL_GRAPHICS`. Nous mettons uniquement `VK_SHADER_STAGE_VERTEX_BIT`.

```
uboLayoutBinding.pImmutableSamplers = nullptr; // Optionnel
```

Le champ `pImmutableSamplers` n'a de sens que pour les descripteurs liés aux samplers d'images. Nous nous attaquerons à ce sujet plus tard. Vous pouvez le mettre à `nullptr`.

Tous les liens des descripteurs sont ensuite combinés en un seul objet `VkDescriptorSetLayout`. Créez pour cela un nouveau membre donnée :

```
VkDescriptorSetLayout descriptorSetLayout;
VkPipelineLayout pipelineLayout;
```

Nous pouvons créer cet objet à l'aide de la fonction `vkCreateDescriptorSetLayout`. Cette fonction prend en argument une structure de type `VkDescriptorSetLayoutCreateInfo`. Elle contient un tableau contenant les structures qui décrivent les bindings :

```
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("echec de la creation d'un set de descripteurs!");
}
```

Nous devons fournir cette structure à Vulkan durant la création de la pipeline graphique. Ils sont transmis par la structure `VkPipelineLayoutCreateInfo`. Modifiez ainsi la création de cette structure :

```
VkPipelineLayoutCreateInfo pipelineLayoutInfo{};
pipelineLayoutInfo.sType = VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;
pipelineLayoutInfo.setLayoutCount = 1;
pipelineLayoutInfo.pSetLayouts = &descriptorSetLayout;
```

Vous vous demandez peut-être pourquoi il est possible de spécifier plusieurs set de descripteurs dans cette structure, dans la mesure où un seul inclut tous les `bindings` d'une pipeline. Nous y reviendrons dans le chapitre suivant, quand nous nous intéresserons aux pools de descripteurs.

L'objet que nous avons créé ne doit être détruit que lorsque le programme se termine.

```
void cleanup() {
    cleanupSwapChain();

    vkDestroyDescriptorSetLayout(device, descriptorSetLayout, nullptr);

    ...
}
```

Uniform buffer

Dans le prochain chapitre nous référencerons le buffer qui contient les données de l'UBO. Mais nous devons bien sûr d'abord créer ce buffer. Comme nous allons accéder et modifier les données du buffer à chaque frame, il est assez inutile d'utiliser un buffer intermédiaire. Ce serait même en fait contre-productif en terme de performances.

Comme des frames peuvent être "in flight" pendant que nous essayons de modifier le contenu du buffer, nous allons avoir besoin de plusieurs buffers. Nous pouvons soit en avoir un par frame, soit un par image de la swap chain. Comme nous avons un command buffer par image nous allons utiliser cette seconde méthode.

Pour cela créez les membres données `uniformBuffers` et `uniformBuffersMemory` :

```
VkBuffer indexBuffer;
VkDeviceMemory indexBufferMemory;

std::vector<VkBuffer> uniformBuffers;
std::vector<VkDeviceMemory> uniformBuffersMemory;
```

Créez ensuite une nouvelle fonction appelée `createUniformBuffers` et appelez-la après `createIndexBuffers`. Elle allouera les buffers :

```
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 buffer en appliquant à son contenu une transformation à chaque frame. Nous n'utiliserons donc pas `vkMapMemory` ici. Le buffer doit être détruit à la fin du programme. Mais comme il dépend du nombre d'images de la swap chain, et que ce nombre peut évoluer lors d'une reécration, nous devons le supprimer depuis `cleanupSwapChain` :

```
void cleanupSwapChain() {
    ...

    for (size_t i = 0; i < swapChainImages.size(); i++) {
        vkDestroyBuffer(device, uniformBuffers[i], nullptr);
        vkFreeMemory(device, uniformBuffersMemory[i], nullptr);
    }

    ...
}
```

Nous devons également le recréer depuis `recreateSwapChain` :

```
void recreateSwapChain() {
    ...
    createFramebuffers();
    createUniformBuffers();
    createCommandBuffers();
}
```

Mise à jour des données uniformes

Créez la fonction `updateUniformBuffer` et appelez-la dans `drawFrame`, juste après que nous avons déterminé l'image de la swap chain que nous devons acquérir :

```
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 générera une rotation à chaque frame pour que la géométrie tourne sur elle-même. Pour ces fonctionnalités mathématiques nous devons inclure deux en-têtes :

```
#define GLM_FORCE_RADIANS
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>

#include <chrono>
```

Le header `<glm/gtc/matrix_transform.hpp>` expose des fonctions comme `glm::rotate`, `glm:lookAt` ou `glm::perspective`, dont nous avons besoin pour implémenter la 3D. La macro `GLM_FORCE_RADIANS` permet d'éviter toute confusion sur la représentation des angles.

Pour que la rotation s'exécute à une vitesse indépendante du FPS, nous allons utiliser les fonctionnalités de mesure précise de la librairie standrarde C++. Incluez donc `<chrono>` :

```
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();
}
```

Nous commençons donc par écrire la logique de calcul du temps écoulé, mesuré en secondes et stocké dans un `float`.

Nous allons ensuite définir les matrices model, view et projection stockées dans l'UBO. La rotation sera implémentée comme une simple rotation autour de l'axe Z en fonction de 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)` la géométrie tournera 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));
```

Pour la matrice view, j'ai décidé de la générer de telle sorte que nous regardions le rectangle par dessus avec une inclinaison de 45 degrés. La fonction `glm::lookAt` prend en arguments la position de l'oeil, la cible du regard et l'axe servant de référence pour le haut.

```
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 de 45 degrés. Les autres paramètres de `glm::perspective` sont le ratio et les plans near et far. Il est important d'utiliser l'étendue actuelle de la swap chain pour calculer le ratio, afin d'utiliser les valeurs qui prennent en compte les redimensionnements de la fenêtre.

```
ubo.proj[1][1] *= -1;
```

GLM a été conçue pour OpenGL, qui utilise les coordonnées de clip et de l'axe Y à l'envers. La manière la plus simple de compenser cela consiste à changer le signe de l'axe Y dans la matrice de projection.

Maintenant que toutes les transformations sont définies nous pouvons copier les données dans le buffer uniform actuel. Nous utilisons la première technique que nous avons vue pour la copie de données dans un buffer.

```
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 le plus efficace pour transmettre des données fréquemment mises à jour. Une meilleure pratique consiste à utiliser les *push constants*, que nous aborderons peut-être dans un futur chapitre.

Dans un avenir plus proche nous allons lier les sets de descripteurs au `VkBuffer` contenant les données des matrices, afin que le vertex shader puisse y avoir accès.

Code C++ / Vertex shader / Fragment shader

Descriptor pool et sets

Introduction

L'objet `VkDescriptorSetLayout` que nous avons créé dans le chapitre précédent décrit les descripteurs que nous devons lier pour les opérations de rendu. Dans ce chapitre nous allons créer les véritables sets de descripteurs, un pour chaque `VkBuffer`, afin que nous puissions chacun les lier au descripteur de l'UBO côté shader.

Pool de descripteurs

Les sets de descripteurs ne peuvent pas être crées directement. Il faut les allouer depuis une pool, comme les command buffers. Nous allons créer la fonction `createDescriptorPool` pour générer une pool de descripteurs.

```
void initVulkan() {
    ...
    createUniformBuffer();
    createDescriptorPool();
    ...
}

...

void createDescriptorPool() {
```

```
11
12 }
```

Nous devons d'abord indiquer les types de descripteurs et combien sont compris dans les sets. Nous utilisons pour cela une structure du type `VkDescriptorPoolSize` :

```
1 VkDescriptorPoolSize poolSize{};
2 poolSize.type = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
3 poolSize.descriptorCount = static_cast<uint32_t>(swapChainImages.size());
```

Nous allons allouer un descripteur par frame. Cette structure doit maintenant être référencée dans la structure principale `VkDescriptorPoolCreateInfo`.

```
1 VkDescriptorPoolCreateInfo poolInfo{};
2 poolInfo.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_POOL_CREATE_INFO;
3 poolInfo.poolSizeCount = 1;
4 poolInfo.pPoolSizes = &poolSize;
```

Nous devons aussi spécifier le nombre maximum de sets de descripteurs que nous sommes susceptibles d'allouer.

```
1 poolInfo.maxSets = static_cast<uint32_t>(swapChainImages.size());
```

La structure possède un membre optionnel également présent pour les command pools. Il permet d'indiquer que les sets peuvent être libérés indépendemment les uns des autres avec la valeur `VK_DESCRIPTOR_POOL_CREATE_FREE_DESCRIPTOR_SET_BIT`. Comme nous n'allons pas toucher aux descripteurs pendant que le programme s'exécute, nous n'avons pas besoin de l'utiliser. Indiquez `0` pour ce champ.

```
1 VkDescriptorPool descriptorPool;
2
3 ...
4
5 if (vkCreateDescriptorPool(device, &poolInfo, nullptr, &descriptorPool) != VK_SUCCESS) {
6     throw std::runtime_error("echec de la creation de la pool de descripteurs!");
7 }
```

Créez un nouveau membre donnée pour référencer la pool, puis appelez `vkCreateDescriptorPool`. La pool doit être recrée avec la swap chain..

```
1  void cleanupSwapChain() {
2      ...
3      for (size_t i = 0; i < swapChainImages.size(); i++) {
4          vkDestroyBuffer(device, uniformBuffers[i], nullptr);
5          vkFreeMemory(device, uniformBuffersMemory[i], nullptr);
6      }
7
8      vkDestroyDescriptorPool(device, descriptorPool, nullptr);
9
10     ...
11 }
```

Et recréée dans `recreateSwapChain` :

```
1 void recreateSwapChain() {
2     ...
3     createUniformBuffers();
4     createDescriptorPool();
```

```
    createCommandBuffers();
}
```

Set de descripteurs

Nous pouvons maintenant allouer les sets de descripteurs. Créez pour cela la fonction `createDescriptorSets` :

```
void initVulkan() {
    ...
    createDescriptorPool();
    createDescriptorSets();
    ...
}

...

void createDescriptorSets() {

}
```

L'allocation de cette ressource passe par la création d'une structure de type `VkDescriptorSetAllocateInfo`. Vous devez bien sûr y indiquer la pool d'où les allouer, de même que le nombre de sets à créer et l'organisation qu'ils doivent suivre.

```
std::vector<VkDescriptorSetLayout> layouts(swapChainImages.size(), descriptorSetLayout);
VkDescriptorSetAllocateInfo allocInfo{};
allocInfo.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_ALLOCATE_INFO;
allocInfo.descriptorPool = descriptorPool;
allocInfo.descriptorSetCount = static_cast<uint32_t>(swapChainImages.size());
allocInfo.pSetLayouts = layouts.data();
```

Dans notre cas nous allons créer autant de sets qu'il y a d'images dans la swap chain. Ils auront tous la même organisation. Malheureusement nous devons copier la structure plusieurs fois car la fonction que nous allons utiliser prend en argument un tableau, dont le contenu doit correspondre indice à indice aux objets à créer.

Ajoutez un membre donnée pour garder une référence aux sets, et allouez-les avec `vkAllocateDescriptorSets` :

```
VkDescriptorPool descriptorPool;
std::vector<VkDescriptorSet> descriptorSets;

...

descriptorSets.resize(swapChainImages.size());
if (vkAllocateDescriptorSets(device, &allocInfo, descriptorSets.data()) != VK_SUCCESS) {
    throw std::runtime_error("echec de l'allocation d'un set de descripteurs!");
}
```

Il n'est pas nécessaire de détruire les sets de descripteurs explicitement, car leur libération est induite par la destruction de la pool. L'appel à `vkAllocateDescriptorSets` alloue donc tous les sets, chacun possédant un unique descripteur d'UBO.

```
void cleanup() {
    ...
    vkDestroyDescriptorPool(device, descriptorPool, nullptr);
```

```
4
5       vkDestroyDescriptorSetLayout(device, descriptorSetLayout, nullptr);
6       ...
7   }
```

Nous avons créé les sets mais nous n'avons pas paramétré les descripteurs. Nous allons maintenant créer une boucle pour rectifier ce problème :

```
1   for (size_t i = 0; i < swapChainImages.size(); i++) {
2
3   }
```

Les descripteurs référant à un buffer doivent être configurés avec une structure de type `VkDescriptorBufferInfo`. Elle indique le buffer contenant les données, et où les données y sont stockées.

```
1   for (size_t i = 0; i < swapChainImages.size(); i++) {
2       VkDescriptorBufferInfo bufferInfo{};
3       bufferInfo.buffer = uniformBuffers[i];
4       bufferInfo.offset = 0;
5       bufferInfo.range = sizeof(UniformBufferObject);
6   }
```

Nous allons utiliser tout le buffer, il est donc aussi possible d'indiquer `VK_WHOLE_SIZE`. La configuration des descripteurs est maintenant mise à jour avec la fonction `vkUpdateDescriptorSets`. Elle prend un tableau de `VkWriteDescriptorSet` en paramètre.

```
1   VkWriteDescriptorSet descriptorWrite{};
2   descriptorWrite.sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
3   descriptorWrite.dstSet = descriptorSets[i];
4   descriptorWrite.dstBinding = 0;
5   descriptorWrite.dstArrayElement = 0;
```

Les deux premiers champs spécifient le set à mettre à jour et l'indice du binding auquel il correspond. Nous avons donné à notre unique descripteur l'indice `0`. Souvenez-vous que les descripteurs peuvent être des tableaux ; nous devons donc aussi indiquer le premier élément du tableau que nous voulons modifier. Nous n'en n'avons qu'un.

```
1   descriptorWrite.descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
2   descriptorWrite.descriptorCount = 1;
```

Nous devons encore indiquer le type du descripteur. Il est possible de mettre à jour plusieurs descripteurs d'un même type en même temps. La fonction commence à `dstArrayElement` et s'étend sur `descriptorCount` descripteurs.

```
1   descriptorWrite.pBufferInfo = &bufferInfo;
2   descriptorWrite.pImageInfo = nullptr; // Optionnel
3   descriptorWrite.pTexelBufferView = nullptr; // Optionnel
```

Le dernier champ que nous allons utiliser est `pBufferInfo`. Il permet de fournir `descriptorCount` structures qui configureront les descripteurs. Les autres champs correspondent aux structures qui peuvent configurer des descripteurs d'autres types. Ainsi il y aura `pImageInfo` pour les descripteurs liés aux images, et `pTexelBufferInfo` pour les descripteurs liés aux buffer views.

```
1   vkUpdateDescriptorSets(device, 1, &descriptorWrite, 0, nullptr);
```

Les mises à jour sont appliquées quand nous appelons `vkUpdateDescriptorSets`. La fonction accepte deux tableaux, un de `VkWriteDesciptorSets` et un de `VkCopyDescriptorSet`. Le second permet de copier des descripteurs.

Utiliser des sets de descripteurs

Nous devons maintenant étendre `createCommandBuffers` pour qu'elle lie les sets de descripteurs aux descripteurs des shaders avec la commande `vkCmdBindDescriptorSets`. Il faut invoquer cette commande dans l'enregistrement des command buffers avant l'appel à `vkCmdDrawIndexed`.

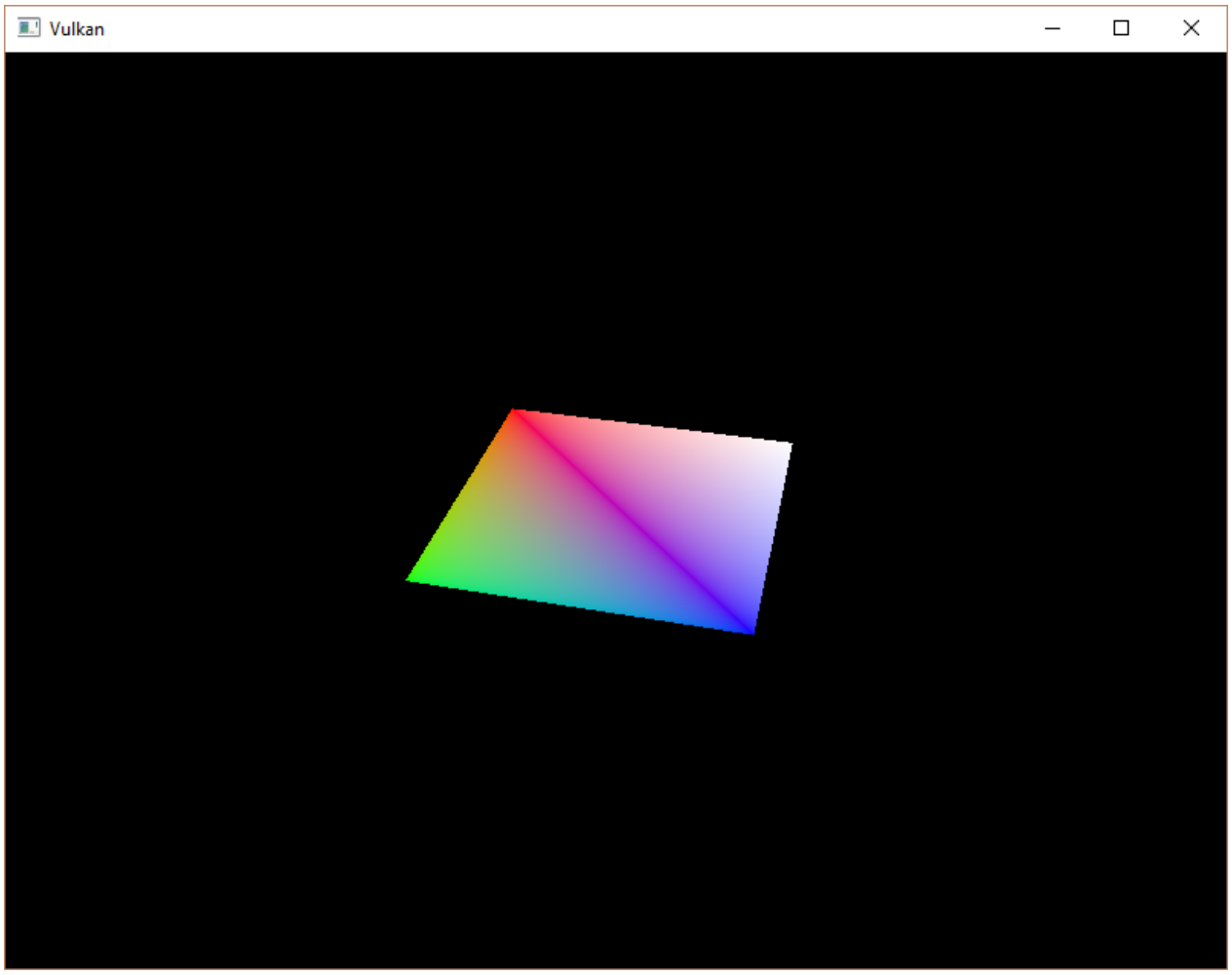
```
vkCmdBindDescriptorSets(commandBuffers[i], VK_PIPELINE_BIND_POINT_GRAPHICS, pipelineLayout,
    0, 1, &descriptorSets[i], 0, nullptr);
vkCmdDrawIndexed(commandBuffers[i], static_cast<uint32_t>(indices.size()), 1, 0, 0, 0);
```

Au contraire des buffers de vertices et d'indices, les sets de descripteurs ne sont pas spécifiques aux pipelines graphiques. Nous devons donc spécifier que nous travaillons sur une pipeline graphique et non pas une pipeline de calcul. Le troisième paramètre correspond à l'organisation des descripteurs. Viennent ensuite l'indice du premier descripteur, la quantité à évaluer et bien sûr le set d'où ils proviennent. Nous y reviendrons. Les deux derniers paramètres sont des décalages utilisés pour les descripteurs dynamiques. Nous y reviendrons aussi dans un futur chapitre.

Si vous lanciez le programme vous verrez que rien ne s'affiche. Le problème est que l'inversion de la coordonnée Y dans la matrice induit l'évaluation des vertices dans le sens inverse des aiguilles d'une montre (*counter-clockwise* en anglais), alors que nous voudrions le contraire. En effet, les systèmes actuels utilisent ce sens de rotation pour détermnier la face de devant. La face de derrière est ensuite simplement ignorée. C'est pourquoi notre géométrie n'est pas rendue. C'est le *backface culling*. Changez le champ `frontface` de la structure `VkPipelineRasterizationStateCreateInfo` dans la fonction `createGraphicsPipeline` de la manière suivante :

```
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` inclut d'office les redimensionnements d'écran, il n'est donc pas nécessaire de recréer les descripteurs dans `recreateSwapChain`.

Alignement

Jusqu'à présent nous avons évité la question de la compatibilité des types côté C++ avec la définition des types pour les variables uniformes. Il semble évident d'utiliser des types au même nom des deux côtés :

```
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 la modification suivante :

```
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! La raison réside dans cette question de l'alignement.

Vulkan s'attend à un certain alignement des données en mémoire pour chaque type. Par exemple :

- Les scalaires doivent être alignés sur leur nombre d'octets N (float de 32 bits donne un alognement de 4 octets)
- Un `vec2` doit être aligné sur 2N (8 octets)
- Les `vec3` et `vec4` doivent être alignés sur 4N (16 octets)
- Une structure imbriquée doit être alignée sur la somme des alignements de ses membres arrondie sur le multiple de 16 octets au-dessus
- Une `mat4` doit avoir le même alignement qu'un `vec4`

Les alignemenents imposés peuvent être trouvés dans la spécification

Notre shader original et ses trois `mat4` était bien aligné. `model` a un décalage de 0, `view` de 64 et `proj` de 128, ce qui sont des multiples de 16.

La nouvelle structure commence avec un membre de 8 octets, ce qui décale tout ce qui suit. Les décalages sont augmentés de 8 et ne sont alors plus multiples de 16. Nous pouvons fixer ce problème avec le mot-clef `alignas` :

```
struct UniformBufferObject {
    glm::vec2 foo;
    alignas(16) glm::mat4 model;
    glm::mat4 view;
    glm::mat4 proj;
};
```

Si vous recompilez et relancez, le programme devrait fonctionner à nouveau.

Heureusement pour nous, GLM inclue un moyen qui nous permet de plus penser à ce souci d'alignement :

```
#define GLM_FORCE_RADIANS
#define GLM_FORCE_DEFAULT_ALIGNED_GENTYPES
#include <glm/glm.hpp>
```

La ligne `#define GLM_FORCE_DEFAULT_ALIGNED_GENTYPES` force GLM a s'assurer de l'alignement des types qu'elle expose. La limite de cette méthode s'atteint en utilisant des structures imbriquées. Prenons l'exemple suivant :

```
struct Foo {
    glm::vec2 v;
};
struct UniformBufferObject {
    Foo f1;
    Foo f2;
};
```

Et côté shader mettons :

```
struct Foo {
    vec2 v;
};
layout(binding = 0) uniform UniformBufferObject {
    Foo f1;
    Foo f2;
} ubo;
```

Nous nous retrouvons avec un décalage de 8 pour `f2` alors qu'il lui faudrait un décalage de 16. Il faut dans ce cas de figure utiliser `alignas` :

```
struct UniformBufferObject {
    Foo f1;
    alignas(16) Foo f2;
};
```

Pour cette raison il est préférable de toujours être explicite à propos de l'alignement de données que l'on envoie aux shaders. Vous ne serez pas supris par des problèmes d'alignement imprévus.

```
struct UniformBufferObject {
    alignas(16) glm::mat4 model;
    alignas(16) glm::mat4 view;
    alignas(16) glm::mat4 proj;
};
```

Recompilez le shader avant de continuer la lecture.

Plusieurs sets de descripteurs

Comme on a pu le voir dans les en-têtes de certaines fonctions, il est possible de lier plusieurs sets de descripteurs en même temps. Vous devez fournir une organisation pour chacun des sets pendant la mise en place de l'organisation de la pipeline. Les shaders peuvent alors accéder aux descripteurs de la manière suivante :

```
layout(set = 0, binding = 0) uniform UniformBufferObject { ... }
```

Vous pouvez utiliser cette possibilité pour placer dans différents sets les descripteurs dépendant d'objets et les descripteurs partagés. De cette manière vous éviter de relier constemment une partie des descripteurs, ce qui peut être plus performant.

Code C++ / Vertex shader / Fragment shader

Texture mapping

Images

Introduction

Jusqu'à présent nous avons écrit les couleurs dans les données de chaque sommet, pratique peu efficace. Nous allons maintenant implémenter l'échantillonnage (sampling) des textures, afin que le rendu soit plus intéressant. Nous pourrons ensuite passer à l'affichage de modèles 3D dans de futurs chapitres.

L'ajout d'une texture comprend les étapes suivantes :

- Créer un objet *image* stocké sur la mémoire de la carte graphique
- La remplir avec les pixels extraits d'un fichier image
- Créer un sampler
- Ajouter un descripteur pour l'échantillonnage de l'image

Nous avons déjà travaillé avec des images, mais nous n'en avons jamais créé. Celles que nous avons manipulées avaient été automatiquement crées par la swap chain. Créer une image et la remplir de pixels ressemble à la création d'un vertex buffer. Nous allons donc commencer par créer une ressource intermédiaire pour y faire transiter les données que nous voulons retrouver dans l'image. Bien qu'il soit possible d'utiliser une image comme intermédiaire, il est aussi autorisé de créer un `VkBuffer` comme intermédiaire vers l'image, et cette méthode est plus rapide sur certaines plateformes. Nous allons donc d'abord créer un buffer et y mettre les données relatives aux pixels. Pour l'image nous devrons nous enquérir des spécificités de la mémoire, allouer la mémoire nécessaire et y copier les pixels. Cette procédure est très similaire à la création de buffers.

La grande différence - il en fallait une tout de même - réside dans l'organisation des données à l'intérieur même des pixels. Leur organisation affecte la manière dont les données brutes de la mémoire sont interprétées. De plus, stocker les pixels ligne par ligne n'est pas forcément ce qui se fait de plus efficace, et cela est dû à la manière dont les cartes graphiques fonctionnent. Nous devrons donc faire en sorte que les images soient organisées de la meilleure manière possible. Nous avons déjà croisé certaines organisation lors de la création de la passe de rendu :

- `VK_IMAGE_LAYOUT_PRESENT_SCR_KHR` : optimal pour la présentation
- `VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL` : optimal pour être l'attachement cible du fragment shader donc en tant que cible de rendu
- `VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL` : optimal pour être la source d'un transfert comme `vkCmdCopyImageToBuffer`
- `VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL` : optimal pour être la cible d'un transfert comme `vkCmdCopyBufferToImage`
- `VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL` : optimal pour être échantillonné depuis un shader

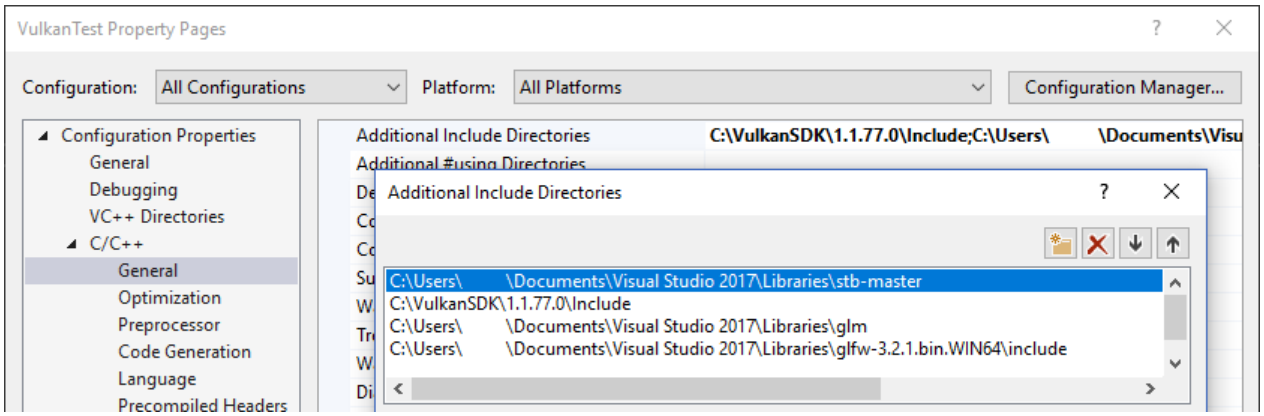
La plus commune des méthode spour réaliser une transition entre différentes organisations est la *barrière pipeline*. Celles-ci sont principalement utilisées pour synchroniser l'accès à une ressource, mais peuvent aussi permettre la transition d'un état à un autre. Dans ce chapitre nous allons utiliser cette seconde possibilité. Les barrières peuvent enfin être utilisées pour changer la queue family qui possède une ressource.

Librairie de chargement d'image

De nombreuses librairies de chargement d'images existent ; vous pouvez même écrire la vôtre pour des formats simples comme BMP ou PPM. Nous allons utiliser stb_image, de la collection stb. Elle possède l'avantage d'être écrite en un seul fichier. Téléchargez donc `stb_image.h` et placez-la ou vous voulez, par exemple dans le dossier où sont stockés GLFW et GLM.

Visual Studio

Ajoutez le dossier comprenant `stb_image.h` dans `Additional Include Directories`.



Makefile

Ajoutez le dossier comprenant `stb_image.h` aux chemins parcourus par 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)
```

Charger une image

Incluez la librairie de cette manière :

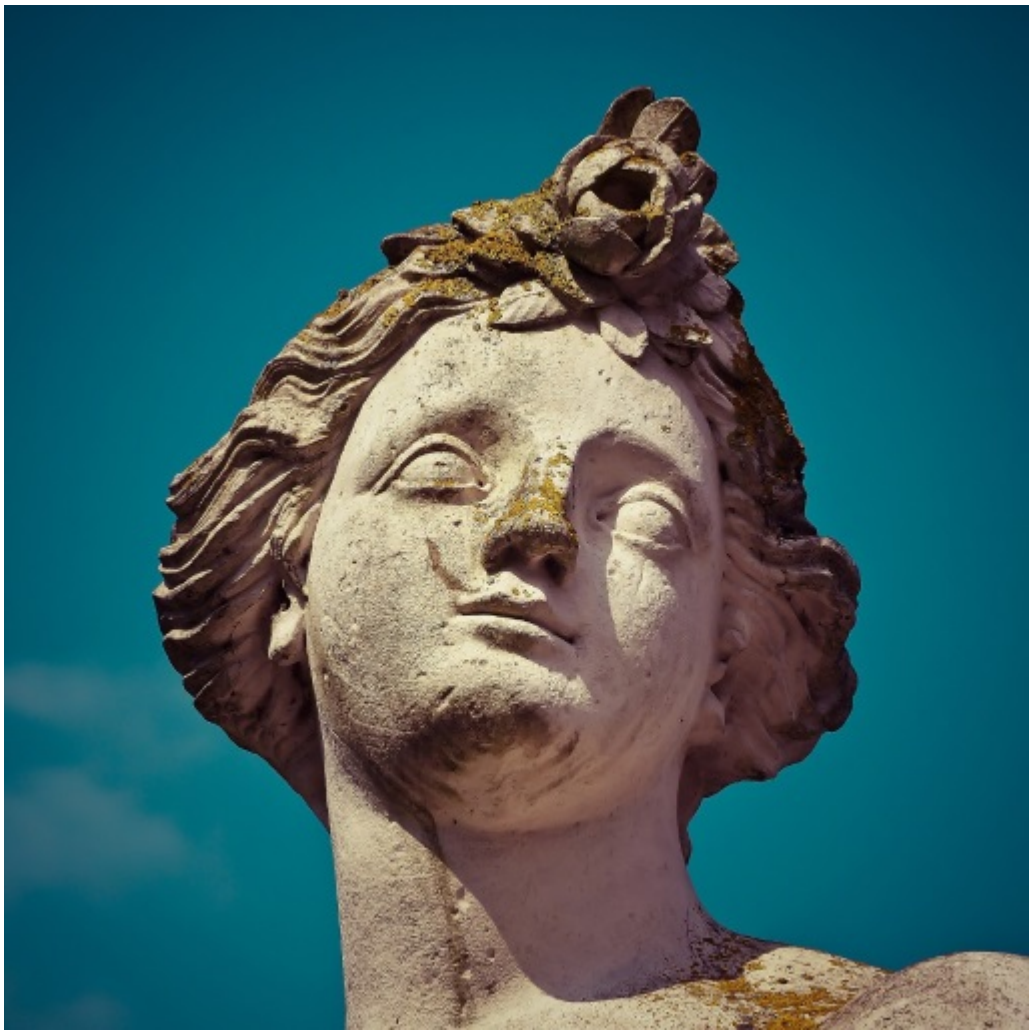
```
#define STB_IMAGE_IMPLEMENTATION
#include <stb_image.h>
```

Le header simple ne fournit que les prototypes des fonctions. Nous devons demander les implémentations avec la define `STB_IMAGE_IMPLEMENTATION` pour ne pas avoir d'erreurs à l'édition des liens.

```
void initVulkan() {
    ...
    createCommandPool();
    createTextureImage();
    createVertexBuffer();
    ...
}

...

void createTextureImage() {

}
```

Créez la fonction `createTextureImage`, depuis laquelle nous chargerons une image et la placerons dans un objet Vulkan représentant une image. Nous allons avoir besoin de command buffers, il faut donc appeler cette fonction après `createCommandPool`.

Créez un dossier `textures` au même endroit que `shaders` pour y placer les textures. Nous allons y mettre un fichier appelé `texture.jpg` pour l'utiliser dans notre programme. J'ai choisi d'utiliser cette image de license CC0 redimensionnée à 512x512, mais vous pouvez bien sûr en utiliser une autre. La librairie supporte des formats tels que JPEG, PNG, BMP ou GIF.



Le chargement d'une image est très facile avec cette librairie :

```
1 void createTextureImage() {
2     int texWidth, texHeight, texChannels;
3     stbi_uc* pixels = stbi_load("textures/texture.jpg", &texWidth, &texHeight, &texChannels,
          STBI_rgb_alpha);
4     VkDeviceSize imageSize = texWidth * texHeight * 4;
5
6     if (!pixels) {
7         throw std::runtime_error("échec du chargement d'une image!");
8     }
9 }
```

La fonction `stbi_load` prend en argument le chemin de l'image et les différentes canaux à charger. L'argument

`STBI_rgb_alpha` force la fonction à créer un canal alpha même si l'image originale n'en possède pas. Cela simplifie le travail en homogénéisant les situations. Les trois arguments transmis en addresse servent de résultats pour stocker des informations sur l'image. Les pixels sont retournés sous forme du pointeur `stbi_uc *pixels`. Ils sont organisés ligne par ligne et ont chacun 4 octets, ce qui représente `texWidth * texHeight * 4` octets au total pour l'image.

Buffer intermédiaire

Nous allons maintenant créer un buffer en mémoire accessible pour que nous puissions utiliser `vkMapMemory` et y placer les pixels. Ajoutez les variables suivantes à la fonction pour contenir ce buffer temporaire :

```
VkBuffer stagingBuffer;
VkDeviceMemory stagingBufferMemory;
```

Le buffer doit être en mémoire visible pour que nous puissions le mapper, et il doit être utilisable comme source d'un transfert vers une image, d'où l'appel suivant :

```
createBuffer(imageSize, VK_BUFFER_USAGE_TRANSFER_SRC_BIT, VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT
    | VK_MEMORY_PROPERTY_HOST_COHERENT_BIT, stagingBuffer, stagingBufferMemory);
```

Nous pouvons placer tel quels les pixels que nous avons récupérés dans le buffer :

```
void* data;
vkMapMemory(device, stagingBufferMemory, 0, imageSize, 0, &data);
    memcpy(data, pixels, static_cast<size_t>(imageSize));
vkUnmapMemory(device, stagingBufferMemory);
```

Il ne faut surtout pas oublier de libérer le tableau de pixels après cette opération :

```
stbi_image_free(pixels);
```

Texture d'image

Bien qu'il nous soit possible de paramétrer le shader afin qu'il utilise le buffer comme source de pixels, il est bien plus efficace d'utiliser un objet image. Ils rendent plus pratique, mais surtout plus rapide, l'accès aux données de l'image en nous permettant d'utiliser des coordonnées 2D. Les pixels sont appelés texels dans le contexte du shading, et nous utiliserons ce terme à partir de maintenant. Ajoutez les membres données suivants :

```
VkImage textureImage;
VkDeviceMemory textureImageMemory;
```

Les paramètres pour la création d'une image sont indiqués 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 contenu dans `imageType` indique à Vulkan le repère dans lesquels les texels sont placés. Il est possible de créer des repères 1D, 2D et 3D. Les images 1D peuvent être utilisés comme des tableaux ou des gradients. Les images 2D sont majoritairement utilisés comme textures. Certaines techniques les utilisent pour stocker autre chose que des couleur, par exemple des vecteurs. Les images 3D peuvent être utilisées

pour stocker des voxels par exemple. Le champ `extent` indique la taille de l'image, en terme de texels par axe. Comme notre texture fonctionne comme un plan dans un espace en 3D, nous devons indiquer `1` au champ `depth`. Finalement, notre texture n'est pas un tableau, et nous verrons le mipmapping plus tard.

```
imageInfo.format = VK_FORMAT_R8G8B8A8_SRGB;
```

Vulkan supporte de nombreux formats, mais nous devons utiliser le même format que les données présentes dans le buffer.

```
imageInfo.tiling = VK_IMAGE_TILING_OPTIMAL;
```

Le champ `tiling` peut prendre deux valeurs :

- `VK_IMAGE_TILING_LINEAR` : les texels sont organisés ligne par ligne
- `VK_IMAGE_TILING_OPTIMAL` : les texels sont organisés de la manière la plus optimale pour l'implémentation

Le mode mis dans `tiling` ne peut pas être changé, au contraire de l'organisation de l'image. Par conséquent, si vous voulez pouvoir directement accéder aux texels, comme il faut qu'il soient organisés d'une manière logique, il vous faut indiquer `VK_IMAGE_TILING_LINEAR`. Comme nous utilisons un buffer intermédiaire et non une image intermédiaire, nous pouvons utiliser le mode le plus efficace.

```
imageInfo.initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
```

Idem, il n'existe que deux valeurs pour `initialLayout` :

- `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 situations où il est nécessaire de préserver les texels pendant la première transition. L'une d'elle consiste à utiliser l'image comme ressource intermédiaire en combinaison avec `VK_IMAGE_TILING_LINEAR`. Il faudrait dans ce cas la faire transitionner vers un état source de transfert, sans perte de données. Cependant nous utilisons un buffer comme ressource intermédiaire, et l'image transitionne d'abord vers cible de transfert. À ce moment-là elle n'a pas de donnée intéressante.

```
imageInfo.usage = VK_IMAGE_USAGE_TRANSFER_DST_BIT | VK_IMAGE_USAGE_SAMPLED_BIT;
```

Le champ de bits `usage` fonctionne de la même manière que pour la création des buffers. L'image sera destination d'un transfert, et sera utilisée par les shaders, d'où les deux indications ci-dessus.

```
imageInfo.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
```

L'image ne sera utilisée que par une famille de queues : celle des graphismes (qui rappelons-le supporte implicitement les transferts). Si vous avez choisi d'utiliser une queue spécifique vous devrez mettre `VK_SHARING_MODE_CONCURENT`.

```
imageInfo.samples = VK_SAMPLE_COUNT_1_BIT;
imageInfo.flags = 0; // Optionnel
```

Le membre `sample` se réfère au multisampling. Il n'a de sens que pour les images utilisées comme attachements d'un framebuffer, nous devons donc mettre `1`, traduit par `VK_SAMPLE_COUNT_1_BIT`. Finalement, certaines informations se réfèrent aux *images étendues*. Ces image étendues sont des images dont seule une partie est stockée dans la mémoire. Voici une exemple d'utilisation : si vous utilisiez une image 3D pour représenter un terrain à l'aide de voxels, vous pourriez utiliser cette fonctionnalité pour éviter d'utiliser de la mémoire qui au final ne contiendrait que de l'air. Nous ne verrons pas cette fonctionnalité dans ce tutoriel, donnez à `flags` la valeur `0`.

```
if (vkCreateImage(device, &imageInfo, nullptr, &textureImage) != VK_SUCCESS) {
    throw std::runtime_error("echec de la creation d'une image!");
}
```

L'image est créée par la fonction `vkCreateImage`, qui ne possède pas d'argument particulièrement intéressant. Il est possible que le format `VK_FORMAT_R8G8B8A8_SRGB` ne soit pas supporté par la carte graphique, mais c'est tellement peu probable que nous ne verrons pas comment y remédier. En effet utiliser un autre format demanderait de réaliser plusieurs conversions compliquées. Nous reviendrons sur ces conversions dans le chapitre sur le buffer de profondeur.

```
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("echec de l'allocation de la mémoire pour l'image!");
}

vkBindImageMemory(device, textureImage, textureImageMemory, 0);
```

L'allocation de la mémoire nécessaire à une image fonctionne également de la même façon que pour un buffer. Seuls les noms de deux fonctions changent : `vkGetBufferMemoryRequirements` devient `vkGetImageMemoryRequirements` et `vkBindBufferMemory` devient `vkBindImageMemory`.

Cette fonction est déjà assez grande ainsi, et comme nous aurons besoin d'autres images dans de futurs chapitres, il est judicieux de déplacer la logique de leur création dans une fonction, comme nous l'avons fait pour les buffers. Voici donc la fonction `createImage` :

```
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("echec de la creation d'une 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("echec de l'allocation de la memoire d'une image!");
    }

    vkBindImageMemory(device, image, imageMemory, 0);
}
```

La largeur, la hauteur, le mode de tiling, l'usage et les propriétés de la mémoire sont des paramètres car ils varierons toujours entre les différentes images que nous créerons dans ce tutoriel.

La fonction `createTextureImage` peut maintenant être réduite à ceci :

```
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 l'image!");
    }

    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);
}
```

Transitions de l'organisation

La fonction que nous allons écrire inclut l'enregistrement et l'exécution de command buffers. Il est donc également judicieux de placer cette logique dans une autre fonction :

```
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 `copyBuffer`. Vous pouvez maintenant réduire `copyBuffer` à :

```
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 de simples buffers nous pourrions nous contenter d'écrire une fonction qui enregistre l'appel à `vkCmdCopyBufferToImage`. Mais comme cette fonction utilse une image comme cible nous devons changer l'organisation de l'image avant l'appel. Créez une nouvelle fonction pour gérer de manière générique 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 pour mémoire d'image*. Une telle barrière de pipeline est en général utilisée pour synchroniser l'accès à une ressource, mais nous avons déjà évoqué ce sujet. Il existe au passage un équivalent pour les buffers : une barrière pour mémoire de buffer.

```
VkImageMemoryBarrier barrier{};
barrier.sType = VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER;
barrier.oldLayout = oldLayout;
barrier.newLayout = newLayout;
```

Les deux premiers champs indiquent la transition à réaliser. Il est possible d'utiliser `VK_IMAGE_LAYOUT_UNDEFINED` pour `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 sont utilisés pour transmettre la possession d'une queue à une autre. Il faut leur indiquer les indices des familles de queues correspondantes. Comme nous ne les utilisons pas, nous devons les mettre à `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, puis la partie de l'image concernées par les changements. Comme notre image n'est pas un tableau, et que nous n'avons pas mis en place de mipmapping, les paramètres sont tous mis au minimum.

```
barrier.srcAccessMask = 0; // TODO
barrier.dstAccessMask = 0; // TODO
```

Comme les barrières sont avant tout des objets de synchronisation, nous devons indiquer les opérations utilisant la ressource avant et après l'exécution de cette barrière. Pour pouvoir remplir les champs ci-dessus nous devons déterminer ces opérations, ce que nous ferons plus tard.

```
vkCmdPipelineBarrier(
    commandBuffer,
    0 /* TODO */, 0 /* TODO */,
    0,
    0, nullptr,
    0, nullptr,
    1, &barrier
);
```

Tous les types de barrière sont mis en place à l'aide de la même fonction. Le paramètre qui suit le command buffer indique une étape de la pipeline. Durant celle-ci seront réalisées les opération devant précéder la barrière. Le paramètre d'après indique également une étape de la pipeline. Cette fois les opérations exécutées durant cette étape attendront la barrière. Les étapes que vous pouvez fournir comme avant- et après-barrière dépendent de l'utilisation des ressources qui y sont utilisées. Les valeurs autorisées sont listées dans ce tableau. Par exemple, si vous voulez lire des données présentes dans un UBO après une barrière qui s'applique au buffer, vous devrez indiquer `VK_ACCESS_UNIFORM_READ_BIT` comme usage, et si le premier shader à utiliser l'uniform est le fragment shader il vous faudra indiquer `VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT` comme étape. Dans ce cas de figure, spécifier une autre étape qu'une étape shader n'aurait aucun sens, et les validation layers vous le feraient remarquer.

Le paramètre sur la troisième ligne peut être soit `0` soit `VK_DEPENDENCY_BY_REGION_BIT`. Dans ce second cas la barrière devient une condition spécifique d'une région de la ressource. Cela signifie entre autres que l'implémentation peut lire une région aussitôt que le transfert y est terminé, sans considération pour les

autres régions. Cela permet d'augmenter encore les performances en permettant d'utiliser les optimisations des architectures actuelles.

Les trois dernières paires de paramètres sont des tableaux de barrières pour chacun des trois types existants : barrière mémorielle, barrière de buffer et barrière d'image.

Copier un buffer dans une image

Avant de compléter `vkCreateTextureImage` nous allons écrire une dernière fonction appelée `copyBufferToImage` :

```
void copyBufferToImage(VkBuffer buffer, VkImage image, uint32_t width, uint32_t height) {
    VkCommandBuffer commandBuffer = beginSingleTimeCommands();

    endSingleTimeCommands(commandBuffer);
}
```

Comme avec les recopies de buffers, nous devons indiquer les parties du buffer à copier et les parties de l'image où écrire. Ces données doivent être placées dans 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. `bufferOffset` indique l'octet à partir duquel les données des pixels commencent dans le buffer. L'organisation des pixels doit être indiquée dans les champs `bufferRowLenght` et `bufferImageHeight`. Il pourrait en effet avoir un espace entre les lignes de l'image. Comme notre image est en un seul bloc, nous devons mettre ces paramètres à `0`. Enfin, les membres `imageSubResource`, `imageOffset` et `imageExtent` indiquent les parties de l'image qui receveront les données.

Les copies buffer vers 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. Normalement l'image doit être dans l'organisation optimale pour la réception de données. Nous avons paramétré la copie pour qu'un seul command buffer soit à l'origine de la copie successive de tous les pixels. Nous aurions aussi pu créer un tableau de `VkBufferImageCopy` pour que le command buffer soit à l'origine de plusieurs copies simultanées.

Préparer la texture d'image

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 y avions fait consistait à créer l'image texture. Notre prochaine étape est donc d'y placer les pixels en les copiant depuis le buffer intermédiaire. Il y a deux étapes pour cela :

- Transitionner l'organisation de l'image vers `VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL`
- Exécuter le buffer de copie

C'est simple à réaliser avec les 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 une organisation `VK_LAYOUT_UNDEFINED`, car le contenu initial ne nous intéresse pas.

Pour ensuite pouvoir échantillonner la texture depuis le fragment shader nous devons réaliser une dernière transition, qui la préparera à être accédée depuis un shader :

```
transitionImageLayout(textureImage, VK_FORMAT_R8G8B8A8_SRGB,
    VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL, VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL);
```

Derniers champs de la barrière de transition

Si vous lanciez le programme vous verrez que les validation layers vous indiquent que les champs d'accès et d'étapes shader sont invalides. C'est normal, nous ne les avons pas remplis.

Nous sommes pour le moment interessés par deux transitions :

- Non défini → cible d'un transfert : écritures par transfert qui n'ont pas besoin d'être synchronisées
- Cible d'un transfert → lecture par un shader : la lecture par le shader doit attendre la fin du transfert

Ces règles sont indiquées en utilisant les valeurs suivantes pour l'accès et les étapes shader :

```
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 d'orgisation 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, l'écriture dans l'image doit se réaliser à l'étape pipeline de transfert. Mais cette opération d'écriture ne dépend d'aucune autre opération. Nous pouvons donc fournir une condition d'accès nulle et `VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT` comme opération pré-barrière. Cette valeur correspond au début de la pipeline, mais ne représente pas vraiment une étape. Elle désigne plutôt le moment où la pipeline se prépare, et donc sert communément aux transferts. Voyez la documentation pour de plus amples informations sur les pseudo-étapes.

L'image sera écrite puis lue dans la même passe, c'est pourquoi nous devons indiquer que le fragment shader aura accès à la mémoire de l'image.

Quand nous aurons besoin de plus de transitions, nous compléterons la fonction de transition pour qu'elle les prenne en compte. L'application devrait maintenant tourner sans problème, bien qu'il n'y aie aucune différence visible.

Un point intéressant est que l'émission du command buffer génère implicitement une synchronisation de type `VK_ACCESS_HOST_WRITE_BIT`. Comme la fonction `transitionImageLayout` exécute un command buffer ne comprenant qu'une seule commande, il est possbile d'utiliser cette synchronisation. Cela signifie que vous pourriez alors mettre `srcAccessMask` à `0` dans le cas d'une transition vers `VK_ACCESS_HOST_WRITE_BIT`. C'est à vous de voir si vous voulez être explicites à ce sujet. Personnellement je n'aime pas du tout faire dépendre mon application sur des opérations cachées, que je trouve dangereusement proche d'OpenGL.

Autre chose intéressante à savoir, il existe une organisation qui supporte toutes les opérations. Elle s'appelle `VK_IMAGE_LAYOUT_GENERAL`. Le problème est qu'elle est évidemment moins optimisée. Elle est cependant utile dans certains cas, comme quand une image doit être utilisée comme cible et comme source, ou pour pouvoir lire l'image juste après qu'elle aie quittée l'organisation préinitialisée.

Enfin, il important de noter que les fonctions que nous avons mises en place exécutent les commandes de manière synchronisées et attendent que la queue soit en pause. Pour de véritables applications il est bien sûr recommandé de combiner toutes ces opérations dans un seul command buffer pour qu'elles soient exécutées de manière asynchrones. Les commandes de transitions et de copie pourraient grandement bénéficier d'une telle pratique. Essayez par exemple de créer une fonction `setupCommandBuffer`, puis d'enregistrer les commandes nécessaires depuis les fonctions actuelles. Appelez ensuite une autre fonction nommée par exemple `flushSetupCommands` qui exécutera le command buffer. Avant d'implémenter ceci attendez que nous ayons fait fonctionner l'échantillonage.

Nettoyage

Complétez la fonction `createImageTexture` en libérant le buffer intermédiaire et en libérant la mémoire :

```
    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);
}
```

L'image texture est utilisée jusqu'à la fin du programme, nous devons donc la libérer dans `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 de quoi y accéder depuis la pipeline. Nous y travaillerons dans le prochain chapitre.

C++ code / Vertex shader / Fragment shader

Vue sur image et sampler

Dans ce chapitre nous allons créer deux nouvelles ressources dont nous aurons besoin pour pouvoir échantillonner une image depuis la pipeline graphique. Nous avons déjà vu la première en travaillant avec la swap chain, mais la seconde est nouvelle, et est liée à la manière dont le shader accédera aux texels de l'image.

Vue sur une image texture

Nous avons vu précédemment que les images ne peuvent être accédées qu'à travers une vue. Nous aurons donc besoin de créer une vue sur notre nouvelle image texture.

Ajoutez un membre donnée pour stocker la référence à la vue de type `VkImageView`. Ajoutez ensuite la fonction `createTextureImageView` qui créera cette vue.

```
VkImageView textureImageView;

...

void initVulkan() {
    ...
    createTextureImage();
    createTextureImageView();
    createVertexBuffer();
    ...
}

...

void createTextureImageView() {

}
```

Le code de cette fonction peut être basé sur `createImageViews`. Les deux seuls changements sont dans `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.components = VK_COMPONENT_SWIZZLE_IDENTITY;
viewInfo.subresourceRange.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
viewInfo.subresourceRange.baseMipLevel = 0;
```

```
viewInfo.subresourceRange.levelCount = 1;
viewInfo.subresourceRange.baseArrayLayer = 0;
viewInfo.subresourceRange.layerCount = 1;
```

Appellons `vkCreateImageView` pour finaliser la création de la vue :

```
if (vkCreateImageView(device, &viewInfo, nullptr, &textureImageView) != VK_SUCCESS) {
    throw std::runtime_error("échec de la création d'une vue sur l'image texture!");
}
```

Comme la logique est similaire à celle de `createImageViews`, nous ferions bien de la déplacer dans une fonction. Créez donc `createImageView` :

```
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 la creation de la vue sur une image!");
    }

    return imageView;
}
```

Et ainsi `createTextureImageView` peut être réduite à :

```
void createTextureImageView() {
    textureImageView = createImageView(textureImage, VK_FORMAT_R8G8B8A8_SRGB);
}
```

Et de même `createImageView` se résume à :

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

    for (uint32_t i = 0; i < swapChainImages.size(); i++) {
        swapChainImageViews[i] = createImageView(swapChainImages[i], swapChainImageFormat);
    }
}
```

Préparons dès maintenant la libération de la vue sur l'image à la fin du programme, juste avant la destruction de l'image elle-même.

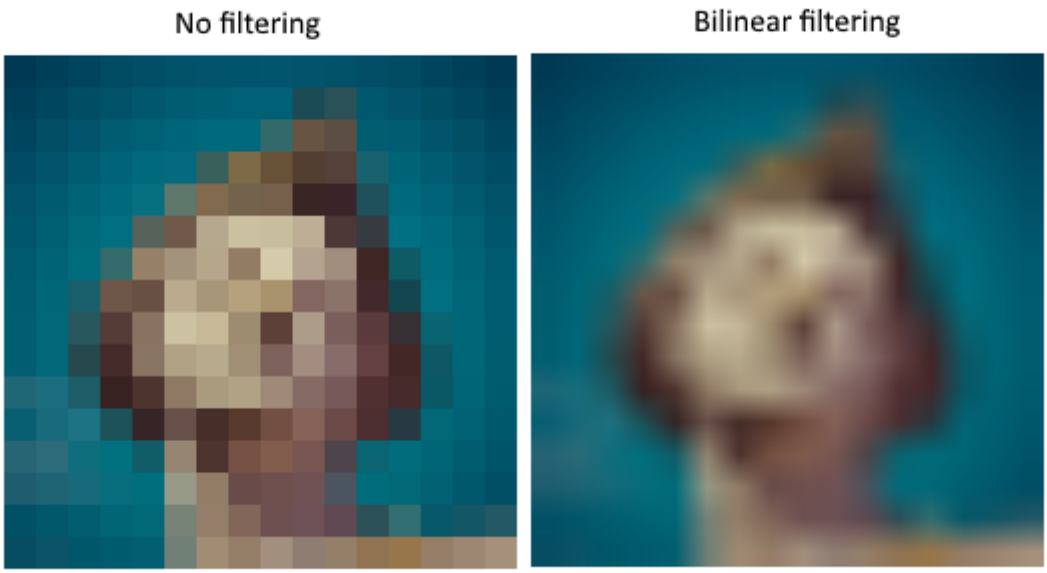
```
void cleanup() {
    cleanupSwapChain();

    vkDestroyImageView(device, textureImageView, nullptr);

```

```
6     vkDestroyImage(device, textureImage, nullptr);
7     vkFreeMemory(device, textureImageMemory, nullptr);
```

Samplers

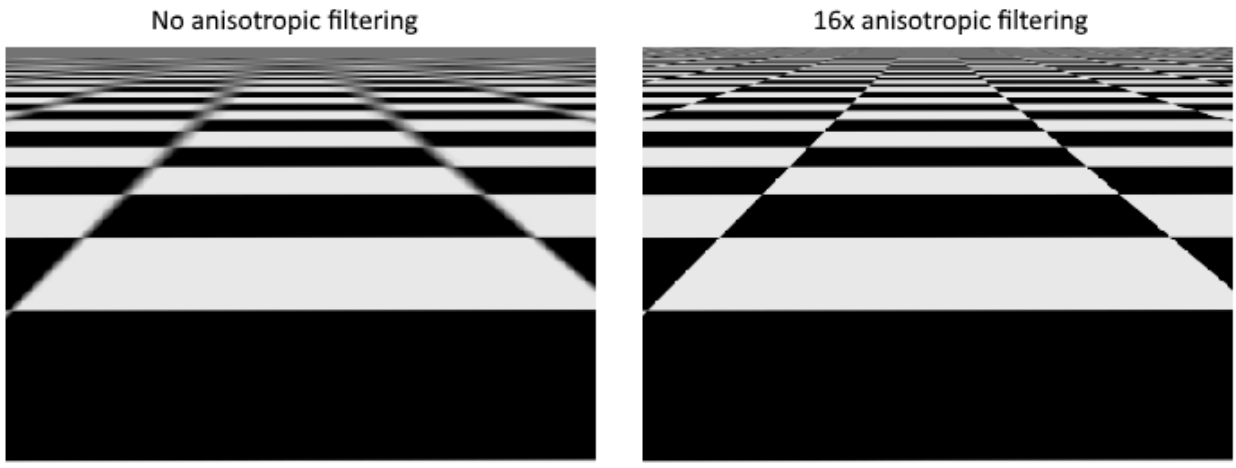
Il est possible pour les shaders de directement lire les texels de l'image. Ce n'est cependant pas la technique communément utilisée. Les textures sont généralement accédées à travers un sampler (ou échantillonneur) qui filtrera et/ou transformera les données afin de calculer la couleur la plus désirable pour le pixel.

Ces filtres sont utiles pour résoudre des problèmes tels que l'oversampling. Imaginez une texture que l'on veut mettre sur de la géométrie possédant plus de fragments que la texture n'a de texels. Si le sampler se contentait de prendre le pixel le plus proche, une pixellisation apparaît :



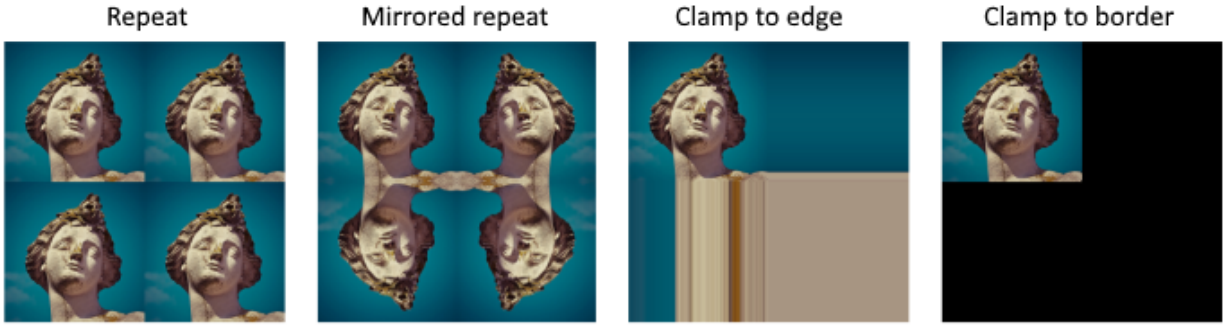
En combinant les 4 texels les plus proches il est possible d'obtenir un rendu lisse comme présenté sur l'image de droite. Bien sûr il est possible que votre application cherche plutôt à obtenir le premier résultat (Minecraft), mais la seconde option est en général préférée. Un sampler applique alors automatiquement ce type d'opérations.

L'undersampling est le problème inverse. Cela crée des artefacts particulièrement visibles dans le cas de textures répétées vues à un angle aigu :



Comme vous pouvez le voir sur l'image de droite, la texture devient d'autant plus floue que l'angle de vision se réduit. La solution à ce problème peut aussi être réalisée par le sampler et s'appelle anisotropic filtering. Elle est par contre plus gourmande en ressources.

Au delà de ces filtres le sampler peut aussi s'occuper de transformations. Il évalue ce qui doit se passer quand le fragment shader essaie d'accéder à une partie de l'image qui dépasse sa propre taille. Il se base sur le *addressing mode* fourni lors de sa configuration. L'image suivante présente les différentes possiblités :



Nous allons maintenant créer la fonction `createTextureSampler` pour mettre en place un sampler simple. Nous l'utiliserons pour lire les couleurs de la texture.

```
1  void initVulkan() {
2      ...
3      createTextureImage();
4      createTextureImageView();
5      createTextureSampler();
6      ...
7  }
8
9  ...
10
11 void createTextureSampler() {
12
13 }
```

Les samplers se configurent avec une structure de type `VkSamplerCreateInfo`. Elle 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 membres `magFilter` et `minFilter` indiquent comment interpoler les texels respectivement magnifiés et minifiés, ce qui correspond respectivement aux problèmes évoqués plus haut. Nous avons choisi `VK_FILTER_LINEAR`, qui indiquent l'utilisation des méthodes pour régler les problèmes vus plus haut.

```
samplerInfo.addressModeU = VK_SAMPLER_ADDRESS_MODE_REPEAT;
samplerInfo.addressModeV = VK_SAMPLER_ADDRESS_MODE_REPEAT;
samplerInfo.addressModeW = VK_SAMPLER_ADDRESS_MODE_REPEAT;
```

Le addressing mode peut être configuré pour chaque axe. Les axes disponibles sont indiqués ci-dessus ; notez l'utilisation de U, V et W au lieu de X, Y et Z. C'est une convention dans le contexte des textures. Voilà les différents modes possibles :

- `VK_SAMPLER_ADDRESS_MODE_REPEAT` : répète le texture
- `VK_SAMPLER_ADDRESS_MODE_MIRRORED_REPEAT` : répète en inversant les coordonnées pour réaliser un effet miroir
- `VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE` : prend la couleur du pixel de bordure le plus proche
- `VK_SAMPLER_ADDRESS_MODE_MIRROR_CLAMP_TO_EDGE` : prend la couleur de l'opposé du plus proche côté de l'image
- `VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_BORDER` : utilise une couleur fixée

Le mode que nous utilisons n'est pas très important car nous ne dépasserons pas les coordonnées dans ce tutoriel. Cependant le mode de répétition est le plus commun car il est infiniment plus efficace que d'envoyer plusieurs fois le même carré à la pipeline, pour dessiner un pavage au sol par exemple.

```
samplerInfo.anisotropyEnable = VK_TRUE;
samplerInfo.maxAnisotropy = 16.0f;
```

Ces deux membres paramètrent l'utilisation de l'anistropic filtering. Il n'y a pas vraiment de raison de ne pas l'utiliser, sauf si vous manquez de performances. Le champ `maxAnistropy` est le nombre maximal de texels utilisés pour calculer la couleur finale. Une plus petite valeur permet d'augmenter les performances, mais résulte évidemment en une qualité réduite. Il n'existe à ce jour aucune carte graphique pouvant utiliser plus de 16 texels car la qualité ne change quasiment plus.

```
samplerInfo.borderColor = VK_BORDER_COLOR_INT_OPAQUE_BLACK;
```

Le paramètre `borderColor` indique la couleur utilisée pour le sampling qui dépasse les coordonnées, si tel est le mode choisi. Il est possible d'indiquer du noir, du blanc ou du transparent, mais vous ne pouvez pas indiquer une couleur quelconque.

```
samplerInfo.unnormalizedCoordinates = VK_FALSE;
```

Le champ `unnomalizedCoordinates` indique le système de coordonnées que vous voulez utiliser pour accéder aux texels de l'image. Avec `VK_TRUE`, vous pouvez utiliser des coordonnées dans `[0, texWidth)` et `[0, texHeight)`. Sinon, les valeurs sont accédées avec des coordonnées dans `[0, 1)`. Dans la plupart des cas les coordonnées sont utilisées normalisées car cela permet d'utiliser un même shader pour des textures de résolution différentes.

```
samplerInfo.compareEnable = VK_FALSE;
samplerInfo.compareOp = VK_COMPARE_OP_ALWAYS;
```

Si une fonction de comparaison est activée, les texels seront comparés à une valeur. Le résultat de la comparaison est ensuite utilisé pour une opération de filtrage. Cette fonctionnalité est principalement utilisée pour réaliser un percentage-closer filtering sur les shadow maps. 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 filtre.

Nous avons maintenant paramétré toutes les fonctionnalités du sampler. Ajoutez un membre donnée pour stocker la référence à ce sampler, puis créez-le avec `vkCreateSampler` :

```
VkImageView textureImageView;
VkSampler textureSampler;

...

void createTextureSampler() {
    ...

    if (vkCreateSampler(device, &samplerInfo, nullptr, &textureSampler) != VK_SUCCESS) {
        throw std::runtime_error("échec de la creation d'un sampler!");
    }
}
```

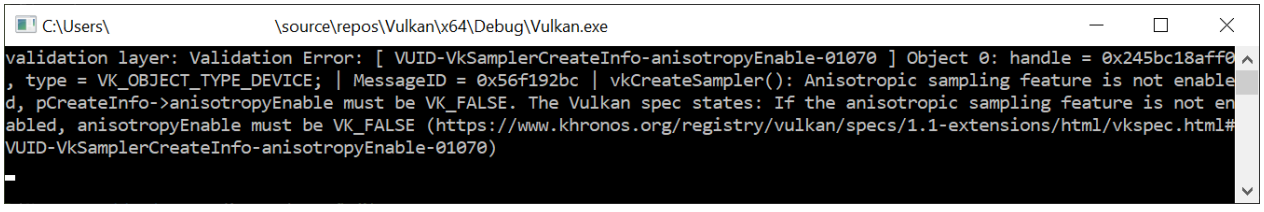
Remarquez que le sampler n'est pas lié à une quelconque `VkImage`. Il ne constitue qu'un objet distinct qui représente une interface avec les images. Il peut être appliqué à n'importe quelle image 1D, 2D ou 3D. Cela diffère d'anciens APIs, qui combinaient la texture et son filtrage.

Préparons la destruction du sampler à la fin du programme :

```
void cleanup() {
    cleanupSwapChain();

    vkDestroySampler(device, textureSampler, nullptr);
    vkDestroyImageView(device, textureImageView, nullptr);

    ...
}
```

Capacité du device à supporter l'anistropie

Si vous lancez le programme, vous verrez que les validation layers vous envoient un message comme celui-ci :



En effet, l'anistropic filtering est une fonctionnalité du device qui doit être activée. Nous devons donc mettre à jour la fonction `createLogicalDevice` :

```
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 devrions aussi adapter `isDeviceSuitable` pour en être sûr.

```
bool isDeviceSuitable(VkPhysicalDevice device) {
    ...

    VkPhysicalDeviceFeatures supportedFeatures;
    vkGetPhysicalDeviceFeatures(device, &supportedFeatures);

    return indices.isComplete() && extensionsSupported && swapChainAdequate &&
        supportedFeatures.samplerAnisotropy;
}
```

La structure `VkPhysicalDeviceFeatures` permet d'indiquer les capacités supportées quand elle est utilisée avec la fonction `VkPhysicalDeviceFeatures`, plutôt que de fournir ce dont nous avons besoin.

Au lieu de simplement obliger le client à posséder une carte graphique supportant l'anistropic filtering, nous pourrions conditionnellement activer ou pas l'anistropic filtering :

```
samplerInfo.anisotropyEnable = VK_FALSE;
samplerInfo.maxAnisotropy = 1.0f;
```

Dans le prochain chapitre nous exposerons l'image et le sampler au fragment shader pour qu'il puisse utiliser la texture sur le carré.

C++ code / Vertex shader / Fragment shader

Sampler d'image combiné

Introduction

Nous avons déjà évoqué les descripteurs dans la partie sur les buffers d'uniformes. Dans ce chapitre nous en verrons un nouveau type : les *samplers d'image combinés* (*combined image sampler*). Ceux-ci permettent aux shaders d'accéder au contenu d'images, à travers un sampler.

Nous allons d'abord modifier l'organisation des descripteurs, la pool de descripteurs et le set de descripteurs pour qu'ils incluent le sampler d'image combiné. Ensuite nous ajouterons des coordonnées de texture à la structure `Vertex` et modifierons le vertex shader et le fragment shader pour qu'il utilisent les couleurs de la texture.

Modifier les descripteurs

Trouvez la fonction `createDescriptorSetLayout` et créez une instance de `VkDescriptorSetLayoutBinding`. Cette structure correspond aux descripteurs d'image combinés. Nous n'avons quasiment que l'indice du binding à y mettre :

```
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 également de bien indiquer le fragment shader dans le champ `stageFlags`. Ce sera à cette étape que la couleur sera extraite de la texture. Il est également possible d'utiliser le sampler pour échantilloner une texture dans le vertex shader. Cela permet par exemple de déformer dynamiquement une grille de vertices pour réaliser une heightmap à partir d'une texture de vecteurs.

Si vous lancez l'application, vous verrez que la pool de descripteurs ne peut pas allouer de set avec l'organisation que nous avons préparée, car elle ne comprend aucun descripteur de sampler d'image combiné. Il nous faut donc modifier la fonction `createDescriptorPool` pour qu'elle inclue une structure `VkDesciptorPoolSize` qui corresponde à ce type de 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 le sampler aux descripteurs du set 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'image combiné doivent être fournies dans une structure de type `VkDescriptorImageInfo`. Cela est similaire à la création d'un descripteur pour buffer. Les objets que nous avons créés dans les chapitres précédents s'assemblent enfin!

```
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 des informations sur l'image, comme pour les buffers. Cette fois nous allons utiliser le tableau `pImageInfo` plutôt que `pBufferInfo`. Les descripteurs sont maintenant prêts à l'emploi.

Coordonnées de texture

Il manque encore un élément au mapping de textures. Ce sont les coordonnées spécifiques aux sommets. Ce sont elles qui déterminent les coordonnées de la texture à lier à 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 qu'elle comprenne un `vec2`, qui servira à contenir 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 vers le fragment shader afin que l'interpolation les transforment en un gradient.

```
const std::vector<Vertex> vertices = {
    {{-0.5f, -0.5f}, {1.0f, 0.0f, 0.0f}, {1.0f, 0.0f}},
    {{0.5f, -0.5f}, {0.0f, 1.0f, 0.0f}, {0.0f, 0.0f}},
    {{0.5f, 0.5f}, {0.0f, 0.0f, 1.0f}, {0.0f, 1.0f}},
    {{-0.5f, 0.5f}, {1.0f, 1.0f, 1.0f}, {1.0f, 1.0f}}
};
```

Dans ce tutoriel nous nous contenterons de mettre une texture sur le carré en utilisant des coordonnées normalisées. Nous mettrons le `0, 0` en haut à gauche et le `1, 1` en bas à droite. Essayez de mettre des valeurs sous `0` ou au-delà de `1` pour voir l'addressing mode en action. Vous pourrez également changer le mode dans la création du sampler pour voir comment ils se comportent.

Shaders

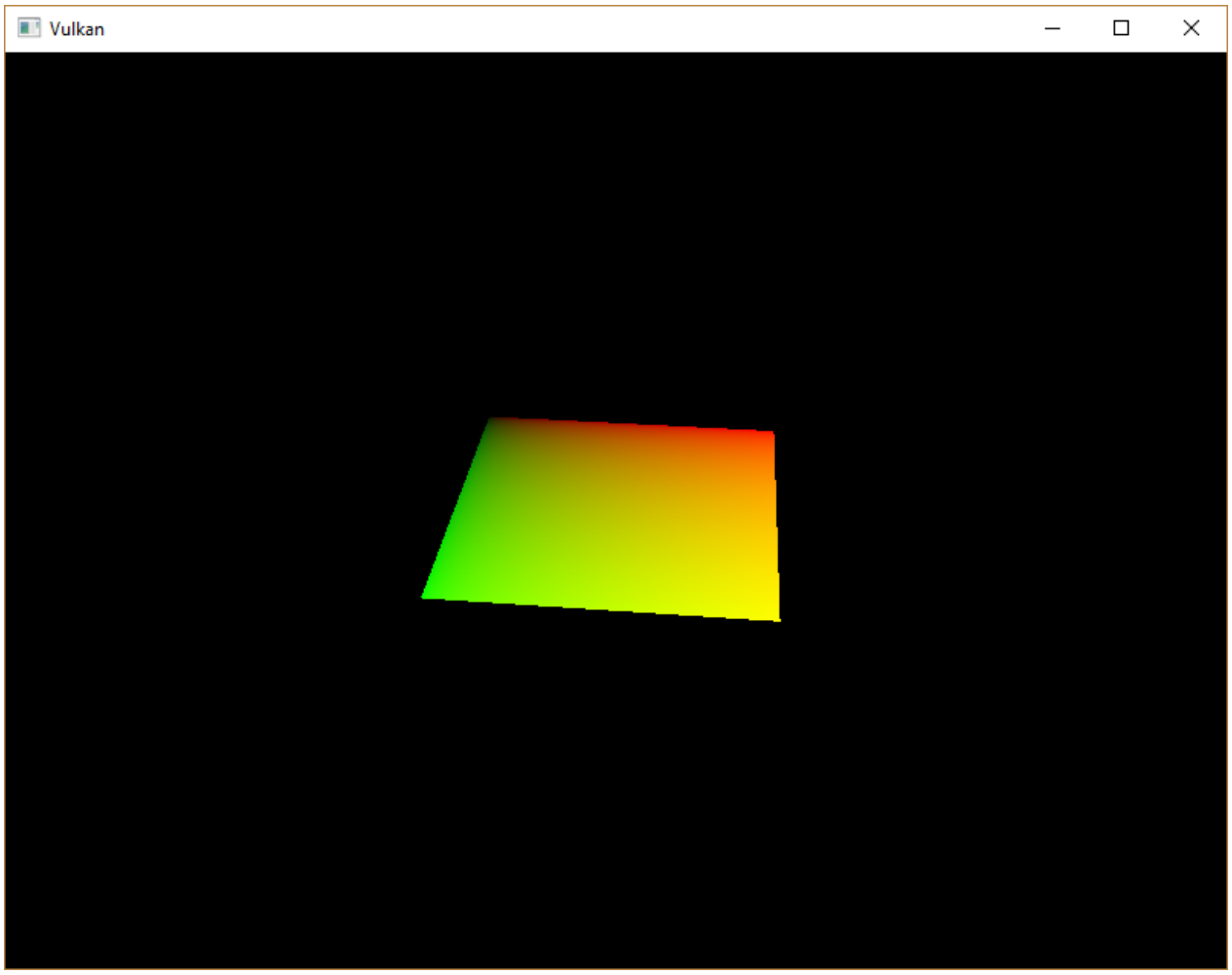
La dernière étape consiste à modifier les shaders pour qu'ils utilisent la texture et non les couleurs. Commençons par le vertex shader :

```
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 spécifiques aux vertices, les valeurs `fragTexCoord` seront interpolées dans le carré par le rasterizer pour créer un gradient lisse. Le résultat de l'interpolation peut être visualisé en utilisant les coordonnées comme couleurs :

```
#version 450

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 l'horizontale et le rouge la verticale. Les coins noirs et jaunes confirment la normalisation des valeurs de `0, 0` à `1, 1`. Utiliser les couleurs pour visualiser les valeurs et déboguer est similaire à utiliser `printf`. C'est peu pratique mais il n'y a pas vraiment d'autre option.

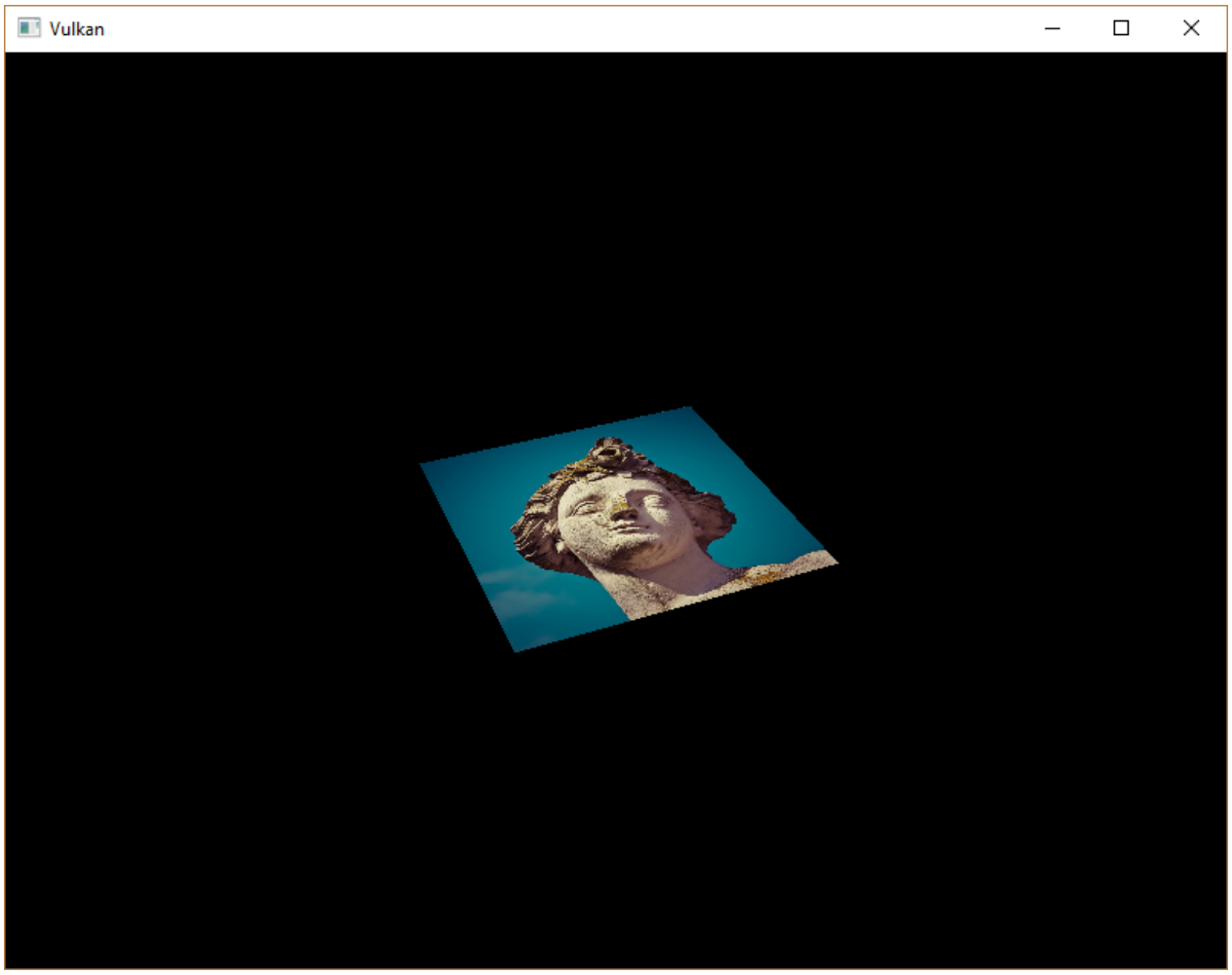
Un descripteur de sampler d'image combiné est représenté dans les shaders par un objet de type `sampler` placé dans une variable uniforme. Créez donc une variable `texSampler` :

```
layout(binding = 1) uniform sampler2D texSampler;
```

Il existe des équivalents 1D et 3D pour de telles textures.

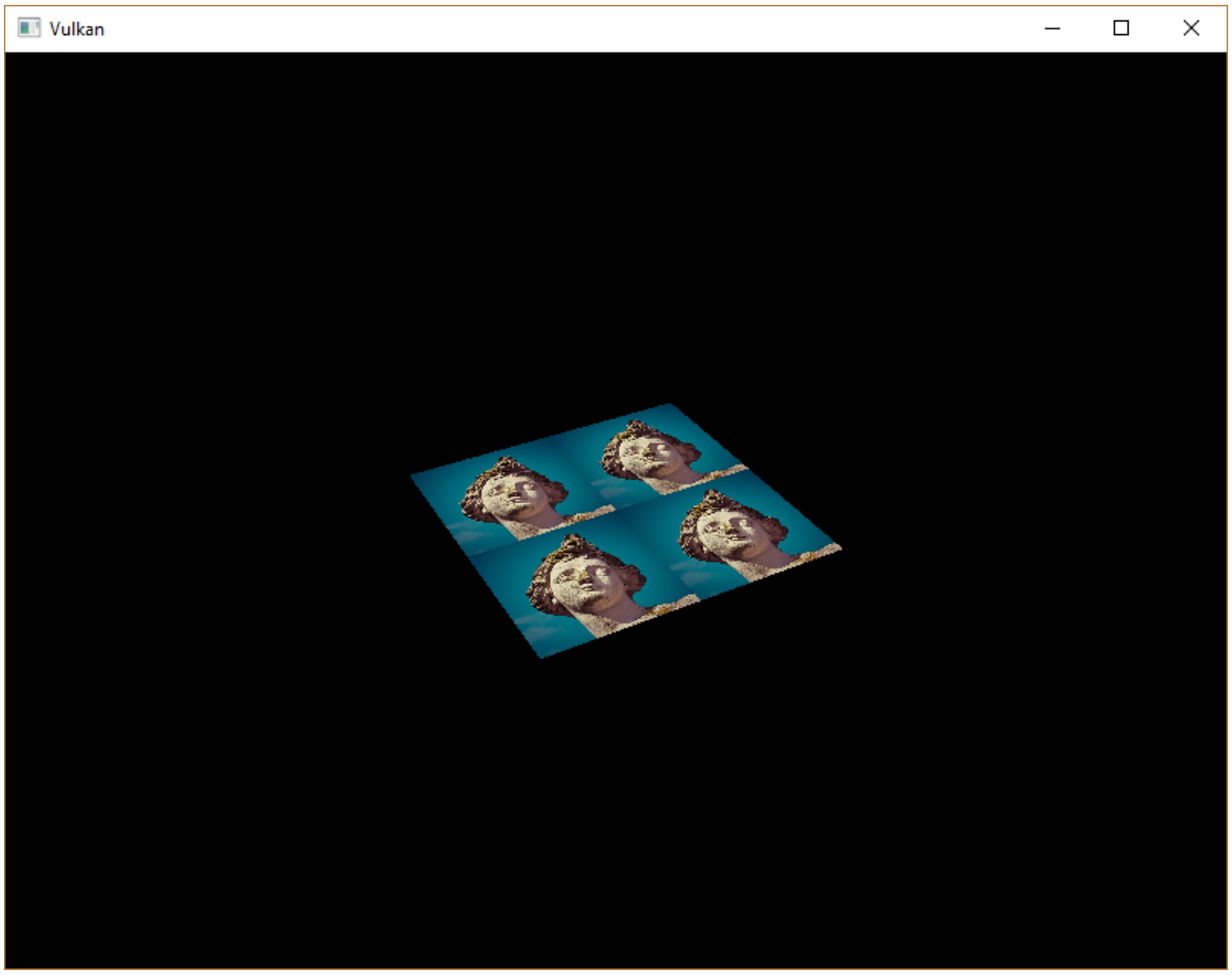
```
void main() {
    outColor = texture(texSampler, fragTexCoord);
}
```

Les textures sont échantillonées à l'aide de la fonction `texture`. Elle prend en argument un objet `sampler` et des coordonnées. Le sampler exécute les transformations et le filtrage en arrière-plan. Vous devriez voir la texture sur le carré maintenant!



Expérimentez avec l'addressing mode en fournissant des valeurs dépassant `1`, et vous verrez la répétition de texture à l'oeuvre :

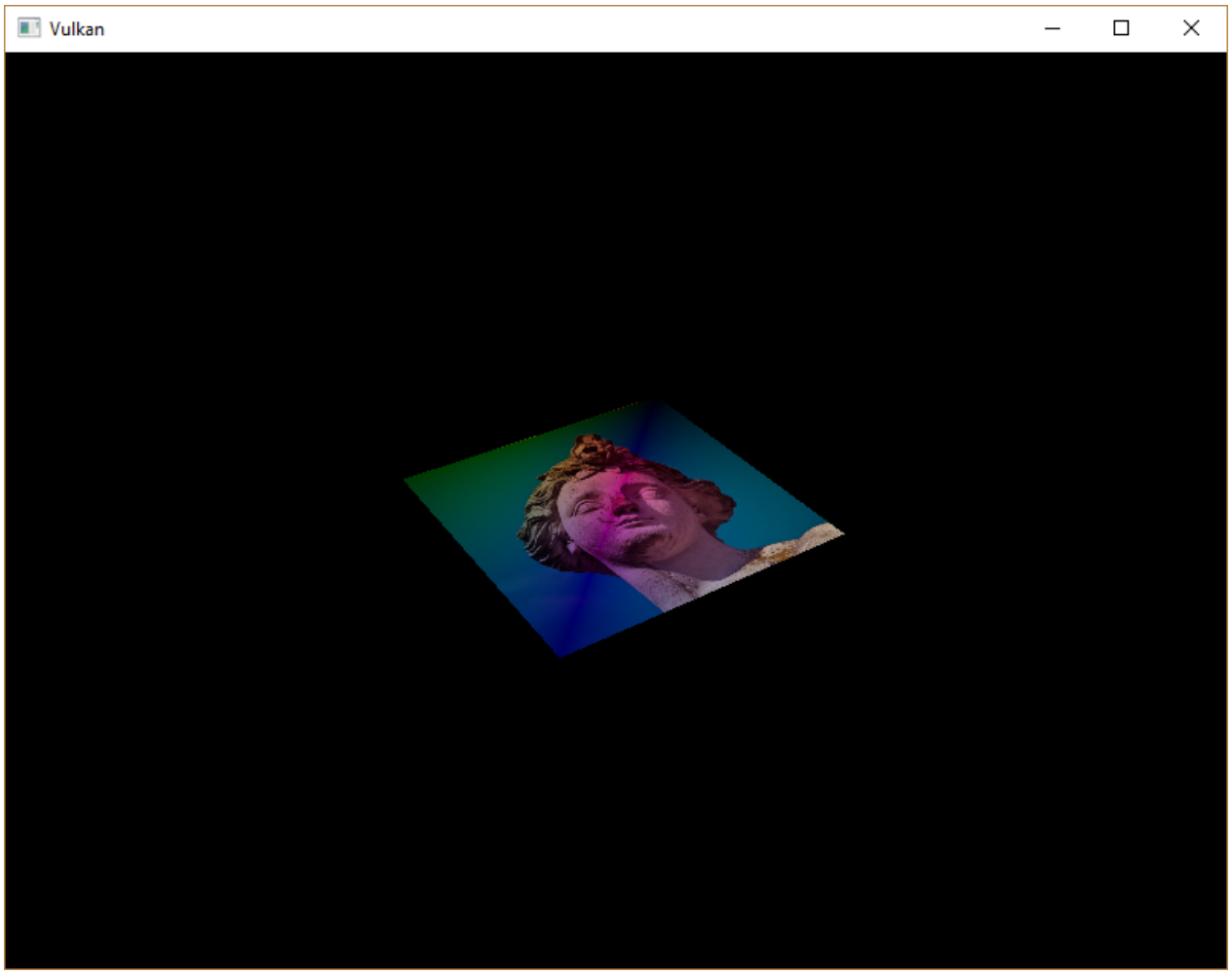
```
void main() {
    outColor = texture(texSampler, fragTexCoord * 2.0);
}
```



Vous pouvez aussi combiner les couleurs avec celles écrites à la main :

```
void main() {
    outColor = vec4(fragColor * texture(texSampler, fragTexCoord).rgb, 1.0);
}
```

J'ai séparé l'alpha du reste pour ne pas altérer la transparence.



Nous pouvons désormais utiliser des textures dans notre programme! Cette technique est extrêmement puissante et permet beaucoup plus que juste afficher des couleurs. Vous pouvez même utiliser les images de la swap chain comme textures et y appliquer des effets post-processing.

Code C++ / Vertex shader / Fragment shader

Buffer de profondeur

Introduction

Jusqu'à présent nous avons projeté notre géométrie en 3D, mais elle n'est toujours définie qu'en 2D. Nous allons ajouter l'axe Z dans ce chapitre pour permettre l'utilisation de modèles 3D. Nous placerons un carré au-dessus ce celui que nous avons déjà, et nous verrons ce qui se passe si la géométrie n'est pas organisée par profondeur.

Géométrie en 3D

Mettez à jour la structure `Vertex` pour que les coordonnées soient des vecteurs à 3 dimensions. Il faut également changer le champ `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);

        ...
    }
};
```

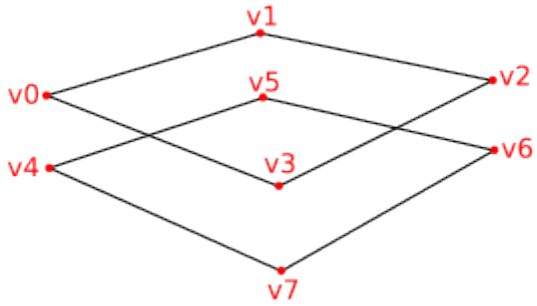
Mettez également à jour l'entrée du vertex shader qui correspond aux coordonnées. Recompilez le shader.

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

...

void main() {
    gl_Position = ubo.proj * ubo.view * ubo.model * vec4(inPosition, 1.0);
    fragColor = inColor;
    fragTexCoord = inTexCoord;
}
```

Enfin, il nous faut ajouter la profondeur là où nous créons les instances de `Vertex`.

```
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}}
};
```

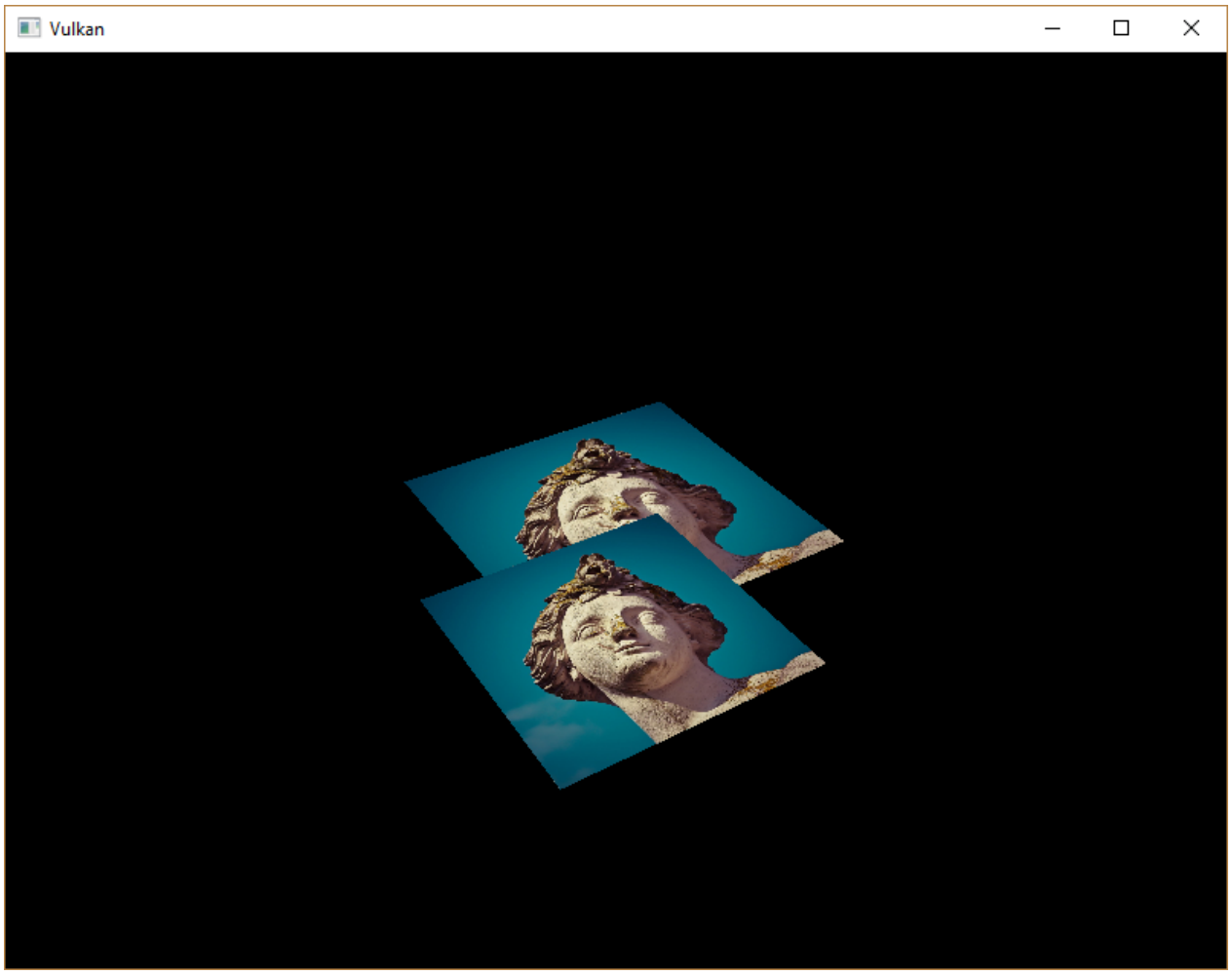
Si vous lancez l'application vous verrez exactement le même résultat. Il est maintenant temps d'ajouter de la géométrie pour rendre la scène plus intéressante, et pour montrer le problème évoqué plus haut. Dupliquez les vertices afin qu'un second carré soit rendu au-dessus de celui que nous avons maintenant :



Nous allons utiliser `-0.5f` comme coordonnée Z.

```
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}}
};

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 verrez que le carré d'en-dessous est rendu au-dessus de l'autre :



Ce problème est simplement dû au fait que le carré d'en-dessous est placé après dans le tableau des vertices. Il y a deux manières de régler ce problème :

- Trier tous les appels en fonction de la profondeur
- Utiliser un buffer de profondeur

La première approche est communément utilisée pour l'affichage d'objets transparents, car la transparence non ordonnée est un problème difficile à résoudre. Cependant, pour la géométrie sans transparence, le buffer de profondeur est un très bonne solution. Il consiste en un attachement supplémentaire au framebuffer, qui stocke les profondeurs. La profondeur de chaque fragment produit par le rasterizer est comparée à la valeur déjà présente dans le buffer. Si le fragment est plus distant que celui déjà traité, il est simplement éliminé. Il est possible de manipuler cette valeur de la même manière que la couleur.

```
1 #define GLM_FORCE_RADIANS
2 #define GLM_FORCE_DEPTH_ZERO_TO_ONE
3 #include <glm/glm.hpp>
4 #include <glm/gtc/matrix_transform.hpp>
```

La matrice de perspective générée par GLM utilise par défaut la profondeur OpenGL comprise en -1 et 1. Nous pouvons configurer GLM avec `GLM_FORCE_DEPTH_ZERO_TO_ONE` pour qu'elle utilise des valeurs correspondant à Vulkan.

Image de pronfondeur et views sur cette image

L'attachement de profondeur est une image. La différence est que celle-ci n'est pas créée par la swap chain. Nous n'avons besoin que d'un seul attachement de profondeur, car les opérations sont séquentielles. L'attachement aura encore besoin des trois mêmes ressources : une image, de la mémoire et une image view.

```
VkImage depthImage;
VkDeviceMemory depthImageMemory;
VkImageView depthImageView;
```

Créez une nouvelle fonction `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'attachement de couleur, définie par l'étendue de la swap chain. Elle doit aussi être configurée comme image de profondeur, avoir un tiling optimal et une mémoire placée sur la carte graphique. Une question persiste : quelle est l'organisation optimale pour une image de profondeur? Le format contient un composant de profondeur, indiqué par `_Dxx_` dans les valeurs de type `VK_FORMAT`.

Au contraire de l'image de texture, nous n'avons pas besoin de déterminer le format requis car nous n'accéderons pas à cette texture nous-mêmes. Nous n'avons besoin que d'une précision suffisante, en général un minimum de 24 bits. Il y a plusieurs formats qui satisfont cette nécéssité :

- `VK_FORMAT_D32_SFLOAT` : float signé de 32 bits pour la profondeur
- `VK_FORMAT_D32_SFLOAT_S8_UINT` : float signé de 32 bits pour la profondeur et int non signé de 8 bits pour le stencil
- `VK_FORMAT_D24_UNORM_S8_UINT` : float signé de 24 bits pour la profondeur et int non signé de 8 bits pour le stencil

Le composant de stencil est utilisé pour le test de stencil. C'est un test additionnel qui peut être combiné avec le test de profondeur. Nous y reviendrons dans un futur chapitre.

Nous pourrions nous contenter d'utiliser `VK_FORMAT_D32_SFLOAT` car son support est pratiquement assuré, mais il est préférable d'utiliser une fonction pour déterminer le meilleur format localement supporté. Créez pour cela la fonction `findSupportedFormat`. Elle vérifiera que les formats en argument sont supportés et choisira le meilleur en se basant sur leur ordre dans le vecteurs des formats acceptables fourni en argument :

```
VkFormat findSupportedFormat(const std::vector<VkFormat>& candidates, VkImageTiling tiling,
    VkFormatFeatureFlags features) {

}
```

Leur support dépend du mode de tiling et de l'usage, nous devons donc les transmettre en argument. Le support des formats peut ensuite être demandé à l'aide de la fonction `vkGetPhysicalDeviceFormatProperties` :

```
for (VkFormat format : candidates) {
    VkFormatProperties props;
    vkGetPhysicalDeviceFormatProperties(physicalDevice, format, &props);
}
```

La structure `VkFormatProperties` contient trois champs :

- `linearTilingFeatures` : utilisations supportées avec le tiling linéaire
- `optimalTilingFeatures` : utilisations supportées avec le tiling optimal
- `bufferFeatures` : utilisations supportées avec les buffers

Seuls les deux premiers cas nous intéressent ici, et celui que nous vérifierons dépendra du 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 désirée, nous pouvons 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("aucun des formats demandés n'est supporté!");
}
```

Nous allons utiliser cette fonction depuis une autre fonction `findDepthFormat`. Elle sélectionnera un format avec un composant de profondeur qui supporte d'être un attachement 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
    );
}
```

Utilisez bien `VK_FORMAT_FEATURE_` au lieu de `VK_IMAGE_USAGE_`. Tous les candidats contiennent la profondeur, mais certains ont le stencil en plus. Ainsi il est important de voir que dans ce cas, la profondeur n'est qu'une *capacité* et non un *usage* exclusif. Autre point, nous devons prendre cela en compte pour les transitions d'organisation. Ajoutez une fonction pour determiner si le format contient un composant de stencil ou non :

```
bool hasStencilComponent(VkFormat format) {
    return format == VK_FORMAT_D32_SFLOAT_S8_UINT || format == VK_FORMAT_D24_UNORM_S8_UINT;
}
```

Appelez cette fonction depuis `createDepthResources` pour déterminer le format de profondeur :

```
VkFormat depthFormat = findDepthFormat();
```

Nous avons maintenant toutes les informations nécessaires pour invoquer `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 cette fonction part du principe que la `subresource` est toujours `VK_IMAGE_ASPECT_COLOR_BIT`, il nous faut donc en faire un paramètre.

```
VkImageView createImageView(VkImage image, VkFormat format, VkImageAspectFlags aspectFlags) {
    ...
    viewInfo.subresourceRange.aspectMask = aspectFlags;
    ...
}
```

Changez également les appels à cette fonction pour prendre en compte ce changement :

```
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 tout pour la création de l'image de profondeur. Nous n'avons pas besoin d'y envoyer de données ou quoi que ce soit de ce genre, car nous allons l'initialiser au début de la render pass tout comme l'attachement de couleur.

Explicitement transitionner l'image de profondeur

Nous n'avons pas besoin de faire explicitement la transition du layout de l'image vers un attachement de profondeur parce qu'on s'en occupe directement dans la render pass. En revanche, pour l'exhaustivité je vais quand même vous décrire le processus dans cette section. Vous pouvez sauter cette étape si vous le souhaitez.

Faites un appel à `transitionImageLayout` à la fin de `createDepthResources` comme ceci:

```
transitionImageLayout(depthImage, depthFormat, VK_IMAGE_LAYOUT_UNDEFINED,
    VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL);
```

L'organisation indéfinie peut être utilisée comme organisation intiale, dans la mesure où aucun contenu d'origine n'a d'importance. Nous devons faire évaluer la logique de `transitionImageLayout` pour qu'elle puisse utiliser la bonne subresource.

```
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 stencil, nous devons nous en occuper dans les transitions de l'image de profondeur.

Ajoutez enfin le bon accès et les bonnes étapes 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'organisation non supportée!");
}
```

Le buffer de profondeur sera lu avant d'écrire un fragment, et écrit après qu'un fragment valide soit traité. La lecture se passe en `VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT` et l'écriture en `VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT`. Vous devriez choisir la première des étapes correspondant à l'opération correspondante, afin que tout soit prêt pour l'utilisation de l'attachement de profondeur.

Render pass

Nous allons modifier `createRenderPass` pour inclure l'attachement de profondeur. Spécifiez d'abord un `VkAttachementDescription` :

```
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. Pour cette fois nous ne garderons pas les données de profondeur, car nous n'en avons plus besoin après le rendu. Encore une fois le hardware pourra réaliser des optimisations. Et de même nous n'avons pas besoin des valeurs du rendu précédent pour le début du rendu de la frame, nous pouvons donc mettre `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'attachement dans notre seule et unique subpasse :

```
VkSubpassDescription subpass{};
subpass.pipelineBindPoint = VK_PIPELINE_BIND_POINT_GRAPHICS;
subpass.colorAttachmentCount = 1;
subpass.pColorAttachments = &colorAttachmentRef;
subpass.pDepthStencilAttachment = &depthAttachmentRef;
```

Les subpasses ne peuvent utiliser qu'un seul attachement de profondeur (et de stencil). Réaliser le test de profondeur sur plusieurs buffers n'a de toute façon pas beaucoup de sens.

```
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;
```

Changez enfin la structure `VkRenderPassCreateInfo` pour qu'elle se réfère aux deux attachements.

Framebuffer

L'étape suivante va consister à modifier la création du framebuffer pour lier notre image de profondeur à l'attachement de profondeur. Trouvez `createFramebuffers` et indiquez la view sur l'image de profondeur comme second attachement :

```
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'attachement de couleur doit différer pour chaque image de la swap chain, mais l'attachement de profondeur peut être le même pour toutes, car il n'est utilisé que par la subpasse, et la synchronisation que nous avons mise en place ne permet pas l'exécution de plusieurs subpasses en même temps.

Nous devons également déplacer l'appel à `createFramebuffers` pour que la fonction ne soit appelée qu'après la création de l'image de profondeur :

```
void initVulkan() {
    ...
    createDepthResources();
    createFramebuffers();
    ...
}
```

Supprimer les valeurs

Comme nous avons plusieurs attachements avec `VK_ATTACHMENT_LOAD_OP_CLEAR`, nous devons spécifier plusieurs valeurs de suppression. Allez à `createCommandBuffers` et créez un tableau de `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, `0.0` correspond au plan near et `1.0` au plan far. La valeur initiale doit donc être `1.0`, afin que tout fragment puisse s'y afficher. Notez que l'ordre des `clearValues` correspond à l'ordre des attachements auquelles les couleurs correspondent.

État de profondeur et de stencil

L'attachement de profondeur est prêt à être utilisé, mais le test de profondeur n'a pas encore été activé. Il est configuré à l'aide d'une 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` permet d'activer la comparaison de la profondeur des fragments. Le champ `depthWriteEnable` indique si la nouvelle profondeur des fragments qui passent le test doivent être écrite dans le tampon de profondeur.

```
depthStencil.depthCompareOp = VK_COMPARE_OP_LESS;
```

Le champ `depthCompareOp` permet de fournir le test de comparaison utilisé pour conserver ou éliminer les fragments. Nous gardons le `<` car il correspond le mieux à la convention employée par Vulkan.

```
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 optionnels d'encadrement de profondeur. Ils permettent de ne garder que des fragments dont la profondeur est comprise entre deux valeurs fournies ici. Nous n'utiliserons pas cette fonctionnalité.

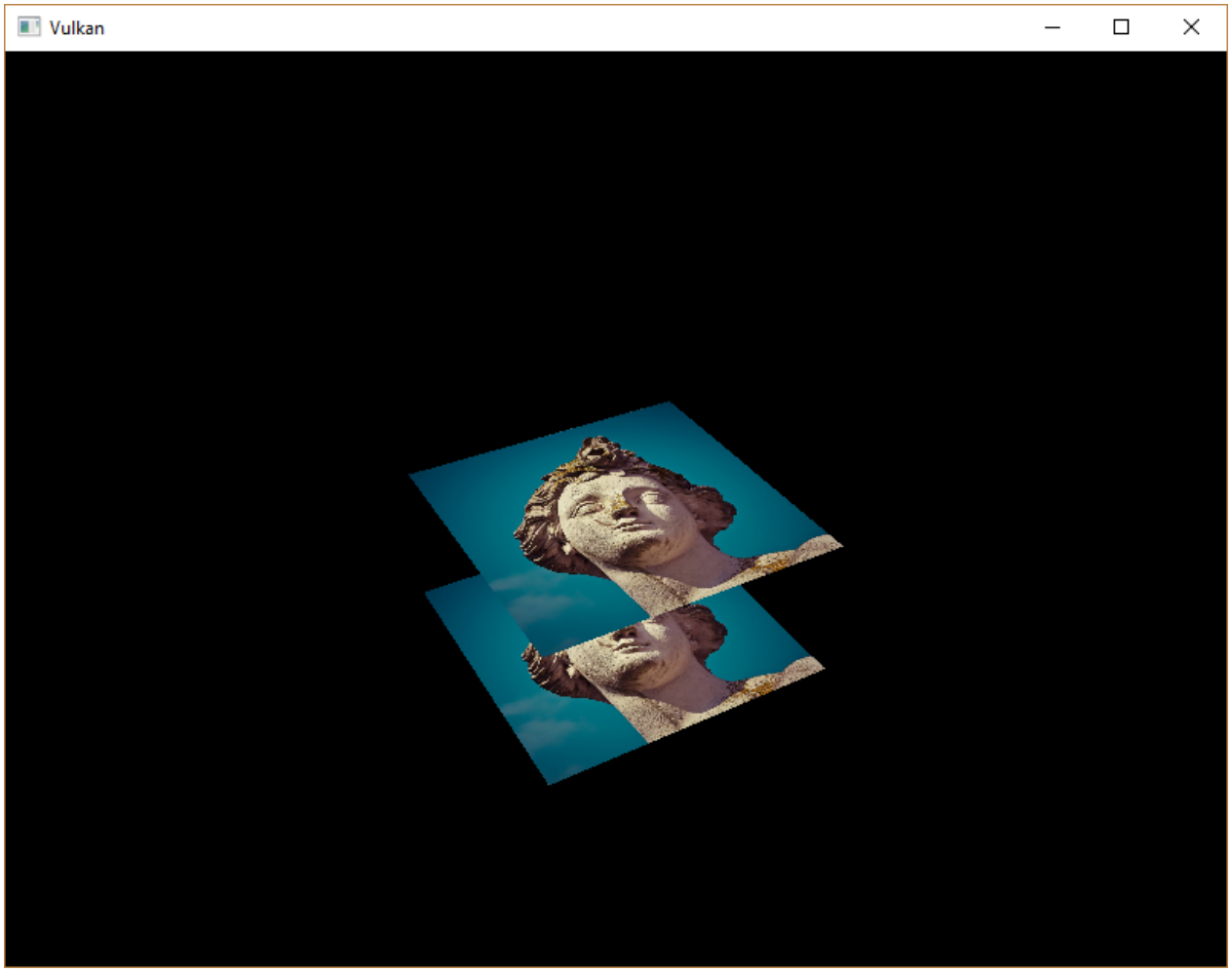
```
depthStencil.stencilTestEnable = VK_FALSE;
depthStencil.front = {}; // Optionnel
depthStencil.back = {}; // Optionnel
```

Les trois derniers champs configurent les opérations du buffer de stencil, 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 un composant pour le stencil.

```
pipelineInfo.pDepthStencilState = &depthStencil;
```

Mettez à jour la création d'une instance de `VkGraphicsPipelineCreateInfo` pour référencer l'état de profondeur et de stencil que nous venons de créer. Un tel état doit être spécifié si la passe contient au moins l'une de ces fonctionnalités.

Si vous lancez le programme, vous verrez que la géométrie est maintenant correctement rendue :



Gestion des redimensionnements de la fenêtre

La résolution du buffer de profondeur doit changer avec la fenêtre quand elle redimensionnée, pour pouvoir correspondre à la taille de l'attachement. Étendez `recreateSwapChain` pour régénérer les ressources :

```
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);

    ...
}
```

Votre application est maintenant capable de rendre correctement de la géométrie 3D! Nous allons utiliser cette fonctionnalité pour afficher un modèle dans le prohain chapitre.

Code C++ / Vertex shader / Fragment shader

Charger des modèles

Introduction

Votre programme peut maintenant réaliser des rendus 3D, mais la géométrie que nous utilisons n'est pas très intéressante. Nous allons maintenant étendre notre programme pour charger les sommets depuis des fichiers. Votre carte graphique aura enfin un peu de travail sérieux à faire.

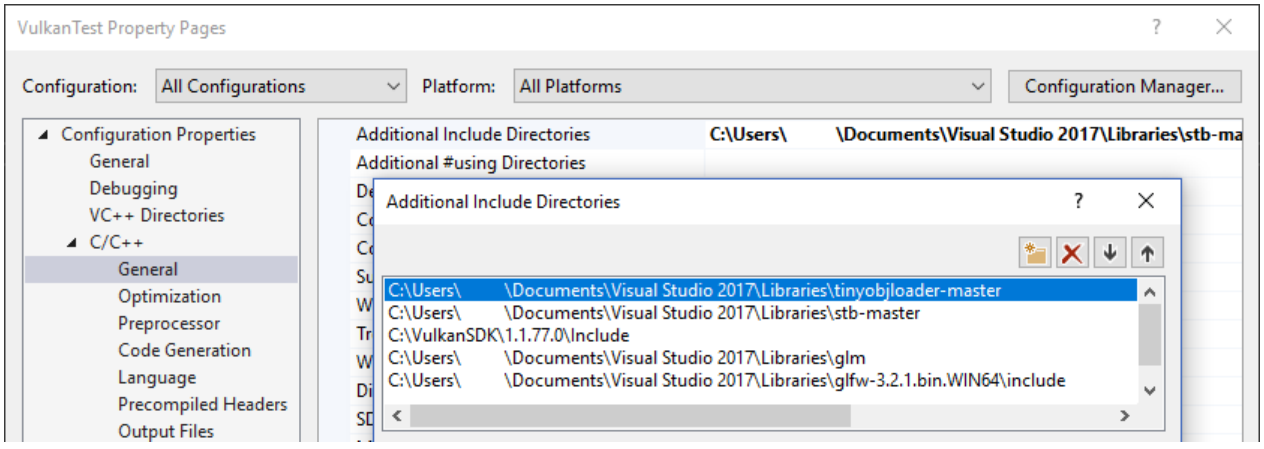
Beaucoup de tutoriels sur les APIs graphiques font implémenter par le lecteur un système pour charger les modèle OBJ. Le problème est que ce type de fichier est limité. Nous *allons* charger des modèles en OBJ, mais nous nous concentrerons plus sur l'intégration des sommets dans le programme, plutôt que sur les aspects spécifiques de ce format de fichier.

Une librairie

Nous utiliserons la librairie tinyobjloader pour charger les vertices et les faces depuis un fichier OBJ. Elle est facile à utiliser et à intégrer, car elle est contenue dans un seul fichier. Téléchargez-la depuis le lien GitHub, elle est contenue dans le fichier `tiny_obj_loader.h`.

Visual Studio

Ajoutez dans `Additional Include Directories` le dossier dans lequel est contenu `tiny_obj_loader.h`.



Makefile

Ajoutez le dossier contenant `tiny_obj_loader.h` aux dossiers d'inclusions de GCC :

```
1 VULKAN_SDK_PATH = /home/user/VulkanSDK/x.x.x.x/x86_64
2 STB_INCLUDE_PATH = /home/user/libraries/stb
3 TINYOBJ_INCLUDE_PATH = /home/user/libraries/tinyobjloader
4
5 ...
```

```

CFLAGS = -std=c++17 -I$(VULKAN_SDK_PATH)/include -I$(STB_INCLUDE_PATH)
    -I$(TINYOBJ_INCLUDE_PATH)
```

Exemple de modèle

Nous n'allons pas utiliser de lumières pour l'instant. Il est donc préférable de charger un modèle qui comprend les ombres pour que nous ayons un rendu plus intéressant. Vous pouvez trouver de tels modèles sur Sketchfab.

Pour ce tutoriel j'ai choisi d'utiliser le Viking room créé par nigelgoh (CC BY 4.0). J'en ai changé la taille et l'orientation pour l'utiliser comme remplacement de notre géométrie actuelle :

- viking_room.obj
- viking_room.png

Il possède un demi-million de triangles, ce qui fera un bon test pour notre application. Vous pouvez utiliser un autre modèle si vous le désirez, mais assurez-vous qu'il ne comprend qu'un seul matériau et que ses dimensions sont d'approximativement 1.5 x 1.5 x 1.5. Si il est plus grand vous devrez changer la matrice view. Mettez le modèle dans un dossier appelé `models`, et placez l'image dans le dossier `textures`.

Ajoutez deux variables de configuration pour la localisation 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";
```

Changez la fonction `createTextureImage` pour qu'elle utilise cette seconde constante pour charger la texture.

```
stbi_uc* pixels = stbi_load(TEXTURE_PATH.c_str(), &texWidth, &texHeight, &texChannels,
    STBI_rgb_alpha);
```

Charger les vertices et les indices

Nous allons maintenant charger les vertices et les indices depuis le fichier OBJ. Supprimez donc les tableaux `vertices` et `indices`, et remplacez-les par des vecteurs dynamiques :

```
std::vector<Vertex> vertices;
std::vector<uint32_t> indices;
VkBuffer vertexBuffer;
VkDeviceMemory vertexBufferMemory;
```

Il faut aussi que le type des indices soit maintenant un `uint32_t` car nous allons avoir plus que 65535 sommets. Changez également le paramètre de type dans l'appel à `vkCmdBindIndexBuffer`.

```
vkCmdBindIndexBuffer(commandBuffers[i], indexBuffer, 0, VK_INDEX_TYPE_UINT32);
```

La librairie que nous utilisons s'inclue de la même manière que les librairies STB. Il faut définir la macro `TINYOBJLOADER_IMLEMENTATION` pour que le fichier comprenne les définitions des fonctions.

```
#define TINYOBJLOADER_IMPLEMENTATION
#include <tiny_obj_loader.h>
```

Nous allons ensuite écrire la fonction `loadModel` pour remplir le tableau de vertices et d'indices depuis le fichier OBJ. Nous devons l'appeler avant que les buffers de vertices et d'indices soient créés.

```
void initVulkan() {
    ...
    loadModel();
    createVertexBuffer();
    createIndexBuffer();
    ...
}

...

void loadModel() {

}
```

Un modèle se charge dans la librairie 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 err;

    if (!tinyobj::LoadObj(&attrib, &shapes, &materials, &err, MODEL_PATH.c_str())) {
        throw std::runtime_error(err);
    }
}
```

Dans un fichier OBJ on trouve des positions, des normales, des coordonnées de textures et des faces. Ces dernières sont une collection de vertices, avec chaque vertex lié à une position, une normale et/ou un coordonnée de texture à l'aide d'un indice. Il est ainsi possible de réutiliser les attributs de manière indépendante.

Le conteneur `attrib` contient les positions, les normales et les coordonnées de texture dans les vecteurs `attrib.vertices`, `attrib.normals` et `attrib.texcoords`. Le conteneur `shapes` contient tous les objets et leurs faces. Ces dernières se réfèrent donc aux données stockées dans `attrib`. Les modèles peuvent aussi définir un matériau et une texture par face, mais nous ignorerons ces attributs pour le moment.

La chaîne de caractères `err` contient les erreurs et les messages générés pendant le chargement du fichier. Le chargement des fichiers ne rate réellement que quand `LoadObj` retourne `false`. Les faces peuvent être constitués d'un nombre quelconque de vertices, alors que notre application ne peut dessiner que des triangles. Heureusement, la fonction possède la capacité - activée par défaut - de trianguliser les faces.

Nous allons combiner toutes les faces du fichier en un seul modèle. Commençons par itérer sur ces faces.

```
for (const auto& shape : shapes) {

}
```

Grâce à la triangularisation nous sommes sûrs que les faces n'ont que trois vertices. Nous pouvons donc simplement les copier vers le vecteur des vertices finales :

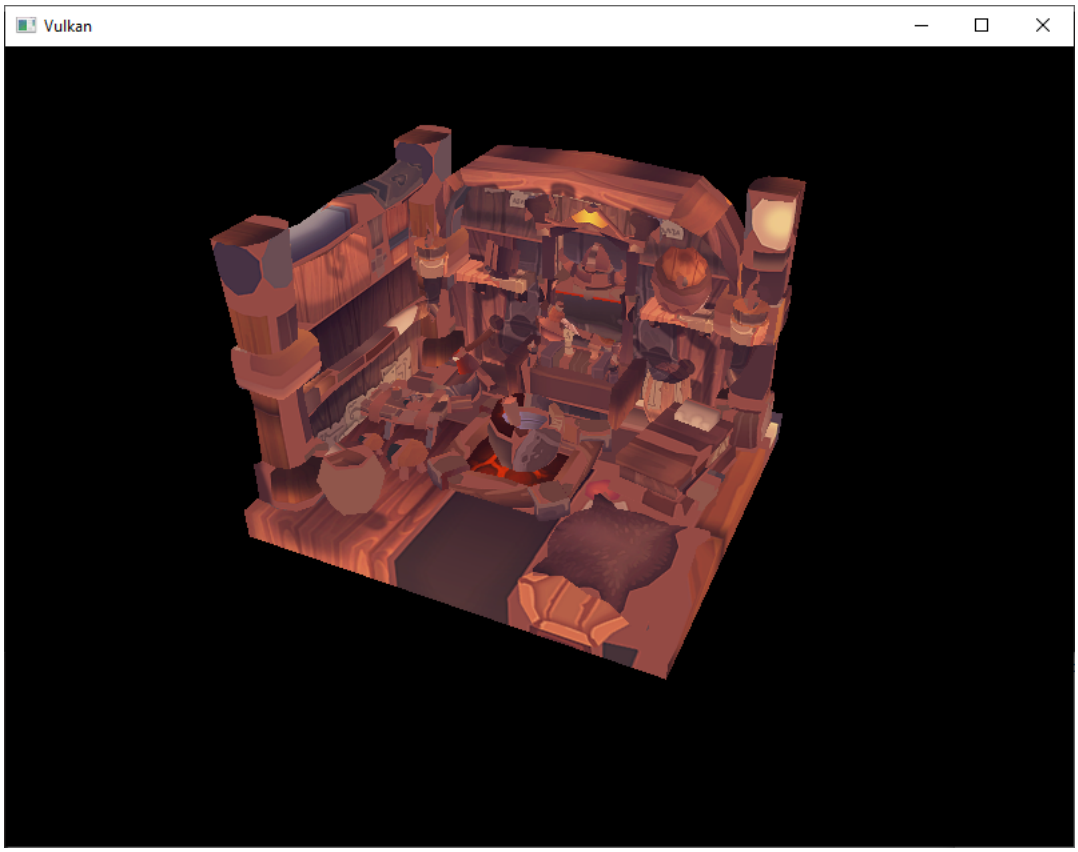
```
for (const auto& shape : shapes) {
    for (const auto& index : shape.mesh.indices) {
        Vertex vertex{};

        vertices.push_back(vertex);
        indices.push_back(indices.size());
    }
}
```

Pour faire simple nous allons partir du principe que les sommets sont uniques. La variable `index` est du type `tinyobj::index_t`, et contient `vertex_index`, `normal_index` et `texcoord_index`. Nous devons traiter ces données pour les relier aux données contenues 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` est constitués de floats et non de vecteurs à trois composants comme `glm::vec3`. Il faut donc multiplier les indices par 3. De même on trouve deux coordonnées de texture par entrée. Les décalages `0`, `1` et `2` permettent ensuite d'accéder aux composant X, Y et Z, ou aux U et V dans le cas des textures.

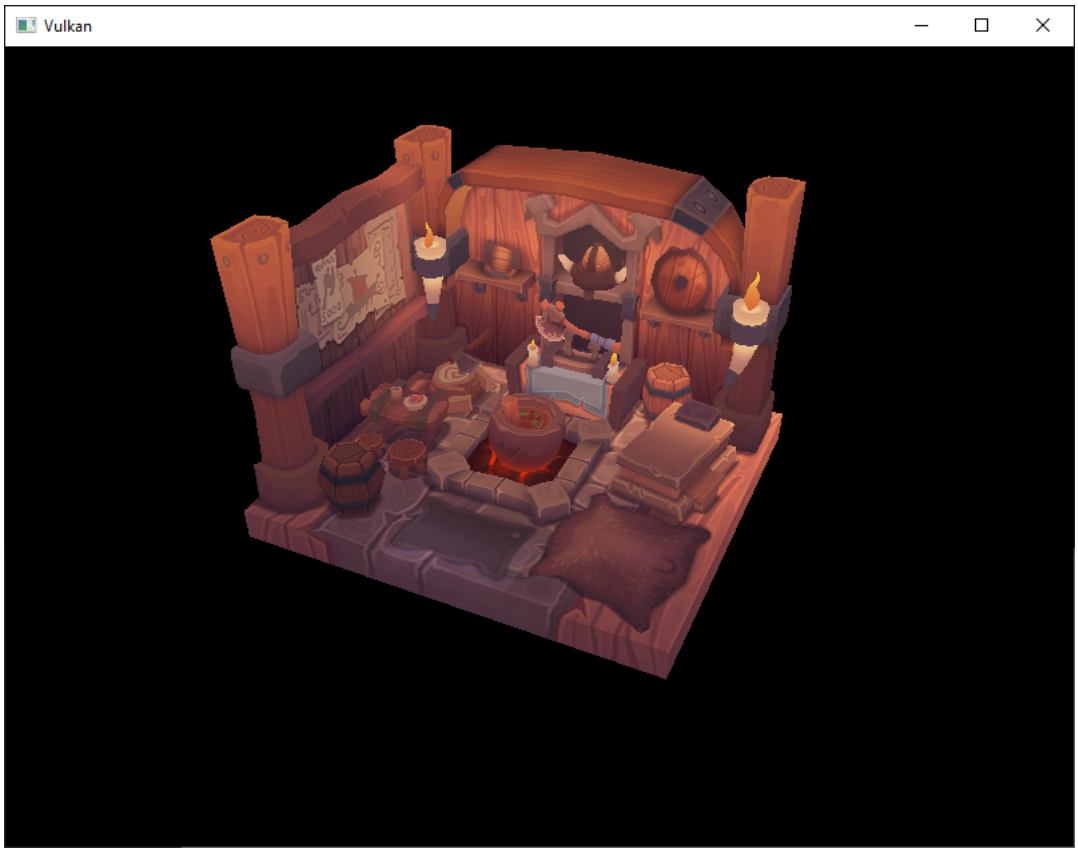
Lancez le programme avec les optimisation activées (`Release` avec Visual Studio ou avec l'argument `-O3` pour GCC). Vous pourriez le faire sans mais le chargement du modèle sera très long. Vous devriez voir ceci :



La géométrie est correcte! Par contre les textures sont quelque peu… étranges. En effet le format OBJ part d'en bas à gauche pour les coordonnées de texture, alors que Vulkan part d'en haut à gauche. Il suffit de changer cela pendant le chargement du modèle :

```
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 être correct :



Déduplication des vertices

Pour le moment nous n'utilisons pas l'index buffer, et le vecteur `vertices` contient beaucoup de vertices dupliquées. Nous ne devrions les inclure qu'une seule fois dans ce conteneur et utiliser leurs indices pour s'y référer. Une manière simple de procéder consiste à utiliser une `unoredered_map` pour suivre les vertices multiples et leurs indices.

```
#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 vertex du fichier, nous devons vérifier si nous avons déjà manipulé un vertex possédant les mêmes attributs. Si il est nouveau, nous le stockerons dans `vertices` et placerons son indice dans `uniqueVertices` et dans `indices`. Si nous avons déjà un tel vertex nous regarderons son indice depuis `uniqueVertices` et copierons cette valeur dans `indices`.

Pour l'instant le programme ne peut pas compiler, car nous devons implémenter une fonction de hachage et l'opérateur d'égalité pour utiliser la structure `Vertex` comme clé dans une table de hachage. L'opérateur est simple à surcharger :

```
bool operator==(const Vertex& other) const {
    return pos == other.pos && color == other.color && texCoord == other.texCoord;
}
```

Nous devons définir une spécialisation du patron de classe `std::hash<T>` pour la fonction de hachage. Le hachage est un sujet compliqué, mais cppreference.com recommande l'approche suivante pour combiner correctement les champs d'une 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 hashage des type GLM sont activés avec la définition et l'inclusion suivantes :

```
#define GLM_ENABLE_EXPERIMENTAL
#include <glm/gtx/hash.hpp>
```

Le dossier `glm/gtx/` contient les extensions expérimentales de GLM. L'API peut changer dans le futur, mais la librairie a toujours été très stable.

Vous devriez pouvoir compiler et lancer le programme maintenant. Si vous regardez la taille de `vertices` vous verrez qu'elle est passée d'un million et demi vertices à seulement 265645! Les vertices sont utilisés pour six triangles en moyenne, ce qui représente une optimisation conséquente.

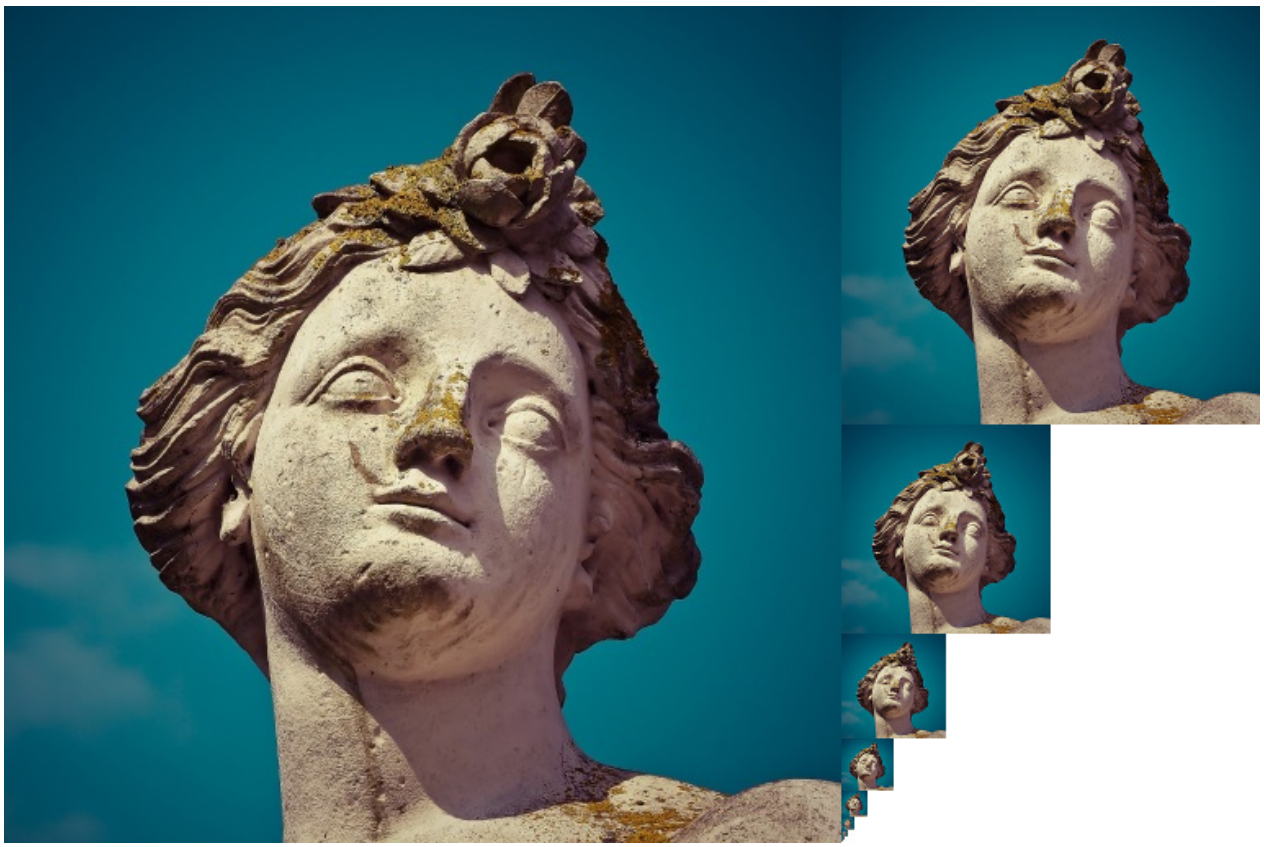
Code C++ / Vertex shader / Fragment shader

Générer des mipmaps

Introduction

Notre programme peut maintenant charger et afficher des modèles 3D. Dans ce chapitre nous allons ajouter une nouvelle fonctionnalité : celle de générer et d'utiliser des mipmaps. Elles sont utilisées dans tous les applications 3D. Vulkan laisse au programmeur un control quasiment total sur leur génération.

Les mipmaps sont des versions de qualité réduite précalculées d'une texture. Chacune de ces versions est deux fois moins haute et large que l'originale. Les objets plus distants de la caméra peuvent utiliser ces versions pour le sampling de la texture. Le rendu est alors plus rapide et plus lisse. Voici un exemple de mipmaps :



Création des images

Avec Vulkan, chaque niveau de mipmap est stocké dans les différents *niveaux de mipmap* de l'image originale. Le niveau 0 correspond à l'image originale. Les images suivantes sont souvent appelées *mip chain*.

Le nombre de niveaux de mipmap doit être fourni lors de la création de l'image. Jusqu'à présent nous avons indiqué la valeur `1`. Nous devons ainsi calculer le nombre de mipmaps à générer à partir de la taille de l'image. Créez un membre donnée pour contenir cette valeur :

```
...
uint32_t mipLevels;
VkImage textureImage;
...
```

La valeur pour `mipLevels` peut être déterminée une fois que nous avons chargé la texture dans `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;
```

La troisième ligne ci-dessus calcule le nombre de niveaux de mipmaps. La fonction `max` chosit la plus grande des dimensions, bien que dans la pratique les textures seront toujours carrées. Ensuite, `log2` donne le nombre de fois que les dimensions peuvent être divisées par deux. La fonction `floor` gère le cas où la dimension n'est pas un multiple de deux (ce qui est déconseillé). `1` est finalement rajouté pour que l'image originale soit aussi comptée.

Pour utiliser cette valeur nous devons changer les fonctions `createImage`, `createImageView` et `transitionImageLayout`. Nous devrons y indiquer le nombre de mipmaps. Ajoutez donc cette donnée en paramètre à toutes 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;
    ...
```

Il nous faut aussi mettre à jour les appels.

```
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);
```

Génération des mipmaps

Notre texture a plusieurs niveaux de mipmaps, mais le buffer intermédiaire ne peut pas gérer cela. Les niveaux autres que 0 sont indéfinis. Pour les remplir nous devons générer les mipmaps à partir du seul niveau que nous avons. Nous allons faire cela du côté de la carte graphique. Nous allons pour cela utiliser la commande `vkCmdBlitImage`. Elle effectue une copie, une mise à l'échelle et un filtrage. Nous allons l'appeler une fois par niveau.

Cette commande est considérée comme une opération de transfert. Nous devons donc indiquer que la mémoire de l'image sera utilisée à la fois comme source et comme destination de la commande. Ajoutez `VK_IMAGE_USAGE_TRANSFER_SRC_BIT` à 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'organisation de l'image sur laquelle elle opère. Nous pourrions transitionner l'image vers `VK_IMAGE_LAYOUT_GENERAL`, mais les opérations prendraient beaucoup de temps. En fait il est possible de transitionner les niveaux de mipmaps indépendemment les uns des autres. Nous pouvons donc mettre l'image initiale à `VK_IMAGE_LAYOUT_TRANSFER_SCR_OPTIMAL` et la chaîne de mipmaps à `VK_IMAGE_LAYOUT_DST_OPTIMAL`. Nous pourrons réaliser les transitions à la fin de chaque opération.

La fonction `transitionImageLayout` ne peut réaliser une transition d'organisation que sur l'image entière. Nous allons donc devoir écrire quelque commandes liées aux barrières de pipeline. Supprimez la transition vers `VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL` dans `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));
//transitionné vers VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL lors de la generation des mipmaps
...
```

Tous les niveaux de l'image seront ainsi en `VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL`. Chaque niveau sera ensuite transitionné vers `VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL` après l'exécution de la commande.

Nous allons maintenant écrire la fonction qui génèrera les mipmaps.

```
void generateMipmaps(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 pour cela nous réutiliserons cette structure `VkImageMemoryBarrier`. Les champs remplis ci-dessus seront valides pour tous les niveaux, et nous allons changer les champs manquant au fur et à mesure de la génération des mipmaps.

```
int32_t mipWidth = texWidth;
int32_t mipHeight = texHeight;

for (uint32_t i = 1; i < mipLevels; i++) {

}
```

Cette boucle va enregistrer toutes les commandes `VkCmdBlitImage`. Remarquez 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 transitionnons le `i-1`ième niveau vers `VK_IMAGE_LAYOUT_TRANSFER_SCR_OPTIMAL`. Cette transition attendra que le niveau de mipmap soit prêt, que ce soit par copie depuis le buffer pour l'image originale, ou bien par `vkCmdBlitImage`. La commande de génération de la mipmap suivante attendra donc la fin de la précédente.

```
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. Le niveau de mipmap source est `i-1` et le niveau destination est `i`. Les deux éléments du tableau `scrOffsets` déterminent en 3D la région source, et `dstOffsets` la région cible. Les coordonnées X et Y sont à chaque fois divisées par deux pour réduire la taille des mipmaps. La coordonnée Z doit être mise à la profondeur de l'image, c'est à dire 1.

```
vkCmdBlitImage(commandBuffer,
    image, VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL,
    image, VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL,
    1, &blit,
    VK_FILTER_LINEAR);
```

Nous enregistrons maintenant les commandes. Remarquez que `textureImage` est utilisé à la fois comme source et comme cible, car la commande s'applique à plusieurs niveaux de l'image. Le niveau de mipmap source vient d'être transitionné vers `VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL`, et le niveau cible est resté en destination depuis sa création.

Attention au cas où vous utilisez une queue de transfert dédiée (comme suggéré dans Vertex buffers) : la fonction `vkCmdBlitImage` doit être envoyée dans une queue graphique.

Le dernier paramètre permet de fournir un `VkFilter`. Nous voulons le même filtre que pour le sampler, nous pouvons donc mettre `VK_FILTER_LINEAR`.

```
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);
```

Ensuite, la boucle transtionne le `i-1`ième niveau de mipmap vers l'organisation optimale pour la lecture par shader. La transition attendra la fin de la commande, de même que les opérations de sampling.

```
    ...
    if (mipWidth > 1) mipWidth /= 2;
    if (mipHeight > 1) mipHeight /= 2;
}
```

Les tailles de la mipmap sont ensuite divisées par deux. Nous vérifions quand même que ces dimensions sont bien supérieures à 1, ce qui peut arriver dans le cas d'une image qui n'est pas carrée.

```
    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 command buffer, nous devons ajouter une dernière barrière. Elle transitionne le dernier niveau de mipmap vers `VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL`. Ce cas n'avait pas été géré par la boucle, car elle n'a jamais servie de source à une copie.

Appelez finalement cette fonction depuis `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));
//transions vers VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL pendant la génération des mipmaps
...
generateMipmaps(textureImage, texWidth, texHeight, mipLevels);
```

Les mipmaps de notre image sont maintenant complètement remplies.

Support pour le filtrage linéaire

La fonction `vkCmdBlitImage` est extrêmement pratique. Malheureusement il n'est pas garanti qu'elle soit disponible. Elle nécessite que le format de l'image texture supporte ce type de filtrage, ce que nous pouvons vérifier avec la fonction `vkGetPhysicalDeviceFormatProperties`. Nous allons vérifier sa disponibilité dans `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) {

    ...
}
```

Utilisez `vkGetPhysicalDeviceFormatProperties` dans `generateMipmaps` pour récupérer les propriétés liés au format :

```
void generateMipmaps(VkImage image, VkFormat imageFormat, int32_t texWidth, int32_t
    texHeight, uint32_t mipLevels) {

    // Vérifions si l'image supporte le filtrage linéaire
    VkFormatProperties formatProperties;
    vkGetPhysicalDeviceFormatProperties(physicalDevice, imageFormat, &formatProperties);

    ...
```

La structure `VkFormatProperties` possède les trois champs `linearTilingFeatures`, `optimalTilingFeature` et `bufferFeaetures`. Ils décrivent chacun l'utilisation possible d'images de ce format dans certains contextes. Nous avons créé l'image avec le format optimal, les informations qui nous concernent sont donc dans `optimalTilingFeatures`. Le support pour le filtrage linéaire est ensuite indiqué par `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 l'image texture ne supporte pas le filtrage
        lineaire!");
}
```

Il y a deux alternatives si le format ne permet pas l'utilisation de `vkCmdBlitImage`. Vous pouvez créer une fonction pour essayer de trouver un format supportant la commande, ou vous pouvez utiliser une librairie pour générer les mipmaps comme stb_image_resize. Chaque niveau de mipmap peut ensuite être chargé de la même manière que vous avez chargé l'image.

Souvenez-vous qu'il est rare de générer les mipmaps pendant l'exécution. Elles sont généralement prégénérées et stockées dans le fichier avec l'image de base. Le chargement de mipmaps prégénérées est laissé comme exercice au lecteur.

Sampler

Un objet `VkImage` contient les données de l'image et un objet `VkSampler` contrôle la lecture des données pendant le rendu. Vulkan nous permet de spécifier les valeurs `minLod`, `maxLod`, `mipLodBias` et `mipmapMode`, où "Lod" signifie *level of detail* (*niveau de détail*). Pendant l'échantillonnage d'une texture, le sampler sélectionne le niveau de mipmap à utiliser suivant ce pseudo-code :

```
lod = getLodLevelFromScreenSize(); //plus petit quand l'objet est proche, peut être negatif
lod = clamp(lod + mipLodBias, minLod, maxLod);

level = clamp(floor(lod), 0, texture.mipLevels - 1);  //limité par le nombre de niveaux de
    mipmaps dans le texture

if (mipmapMode == VK_SAMPLER_MIPMAP_MODE_NEAREST) {
    color = sample(level);
} else {
    color = blend(sample(level), sample(level + 1));
}
```

Si `samplerInfo.mipmapMode` est `VK_SAMPLER_MIPMAP_MODE_NEAREST`, la variable `lod` correspond au niveau de mipmap à échantillonner. Sinon, si il vaut `VK_SAMPLER_MIPMAP_MODE_LINEAR`, deux niveaux de mipmaps sont samplés, puis interpolés linéairement.

L'opération d'échantillonnage est aussi affectée par `lod` :

```
if (lod <= 0) {
    color = readTexture(uv, magFilter);
} else {
    color = readTexture(uv, minFilter);
}
```

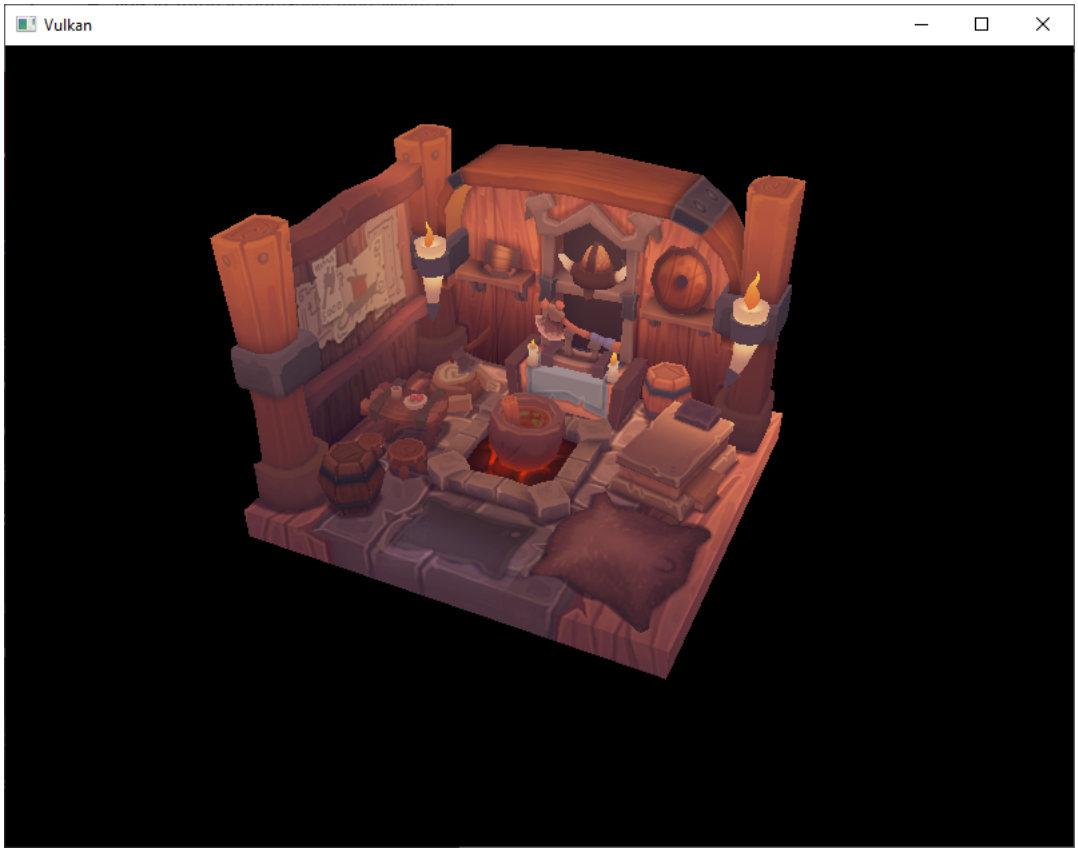
Si l'objet est proche de la caméra, `magFilter` est utilisé comme filtre. Si l'objet est plus distant, `minFilter` sera utilisé. Normalement `lod` est positif, est devient nul au niveau de la caméra. `mipLodBias` permet de forcer Vulkan à utiliser un `lod` plus petit et donc un noveau de mipmap plus élevé.

Pour voir les résultats de ce chapitre, nous devons choisir les valeurs pour `textureSampler`. Nous avons déjà fourni `minFilter` et `magFilter`. Il nous reste les valeurs `minLod`, `maxLod`, `mipLodBias` et `mipmapMode`.

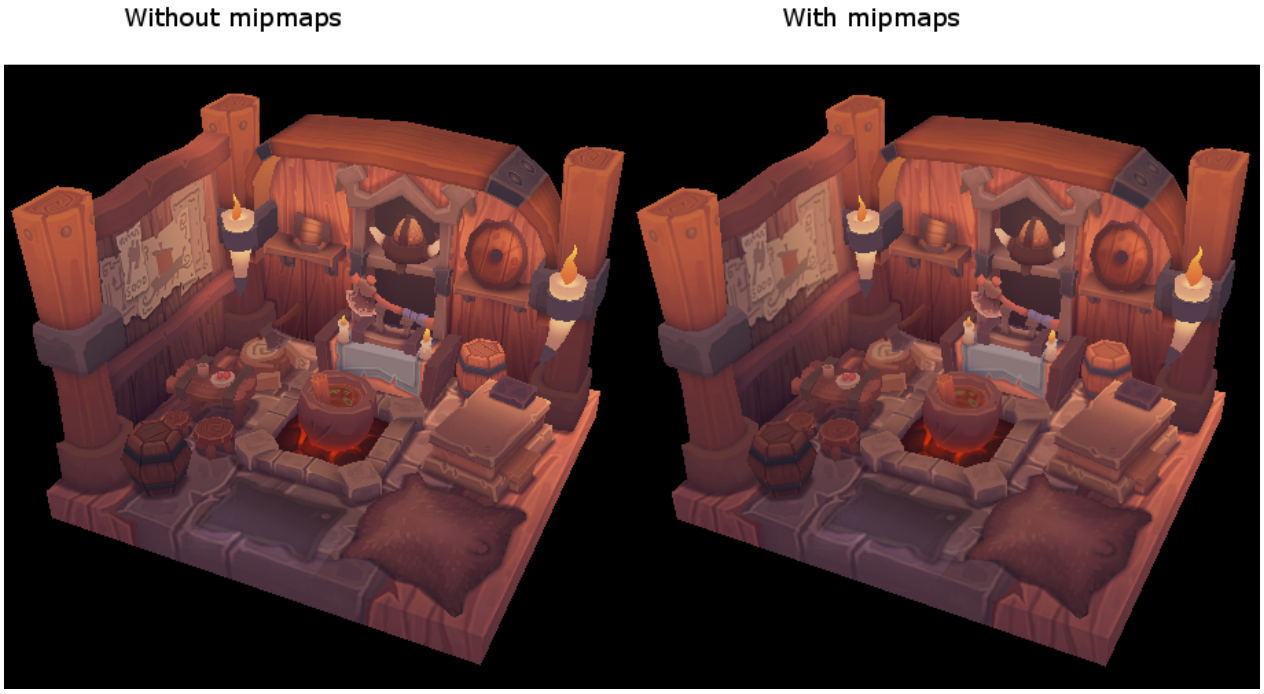
```
void createTextureSampler() {
    ...
    samplerInfo.mipmapMode = VK_SAMPLER_MIPMAP_MODE_LINEAR;
    samplerInfo.minLod = 0.0f;
    samplerInfo.maxLod = VK_LOD_CLAMP_NONE;
    samplerInfo.mipLodBias = 0.0f; // Optionnel
    ...
}
```

Pour utiliser la totalité des niveaux de mipmaps, nous mettons `minLod` à `0.0f` et `maxLod` à `VK_LOD_CLAMP_NONE`. Cette constante est égale à `1000.0f`, ce qui veut dire que la totalité des niveaux de mipmaps disponible dans la texture sera échantillonée. Nous n'avons aucune raison d'altérer `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. En comparant précisement on peut voir quelques différences.

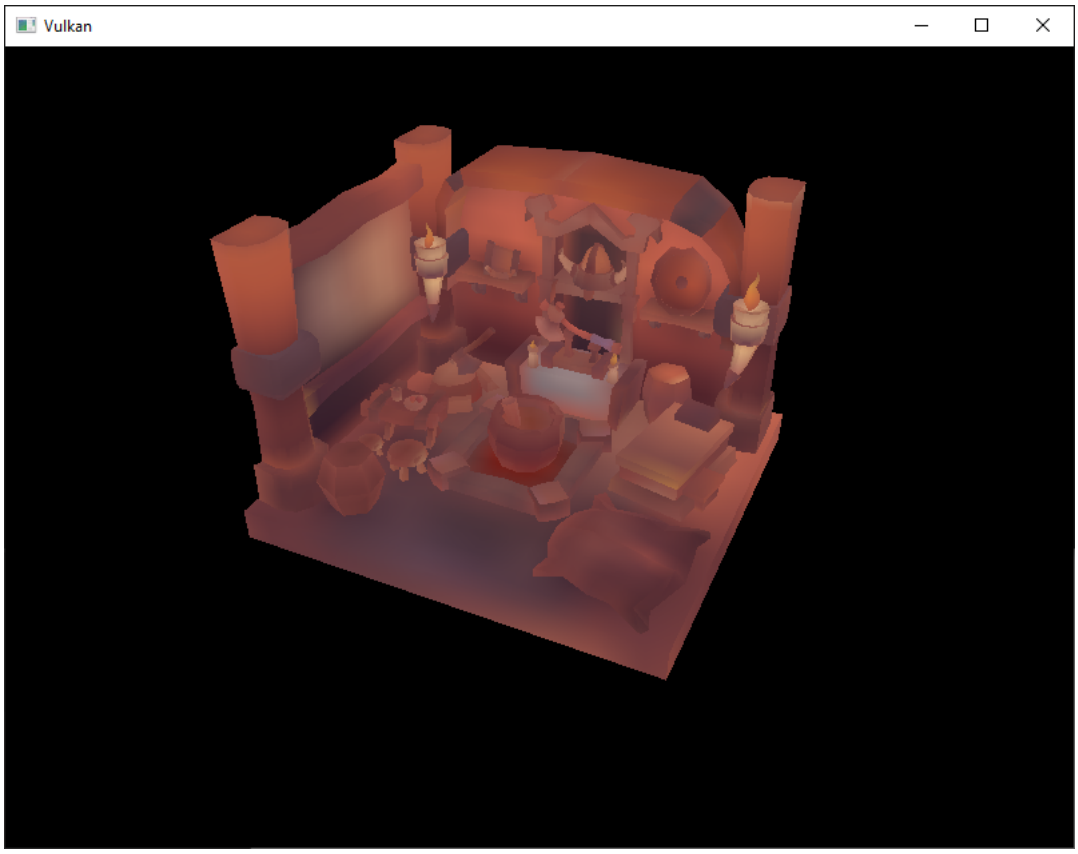


La différence la plus évidente est l'écriture sur le paneau, plus lisse avec les mipmaps.

Vous pouvez modifier les paramètres du sampler pour voir l'impact sur le rendu. Par exemple vous pouvez empêcher le sampler d'utiliser le plus haut nivau de mipmap en ne lui indiquant pas le niveau le plus bas :

```
1 samplerInfo.minLod = static_cast<float>(mipLevels / 2);
```

Ce paramètre produira ce rendu :

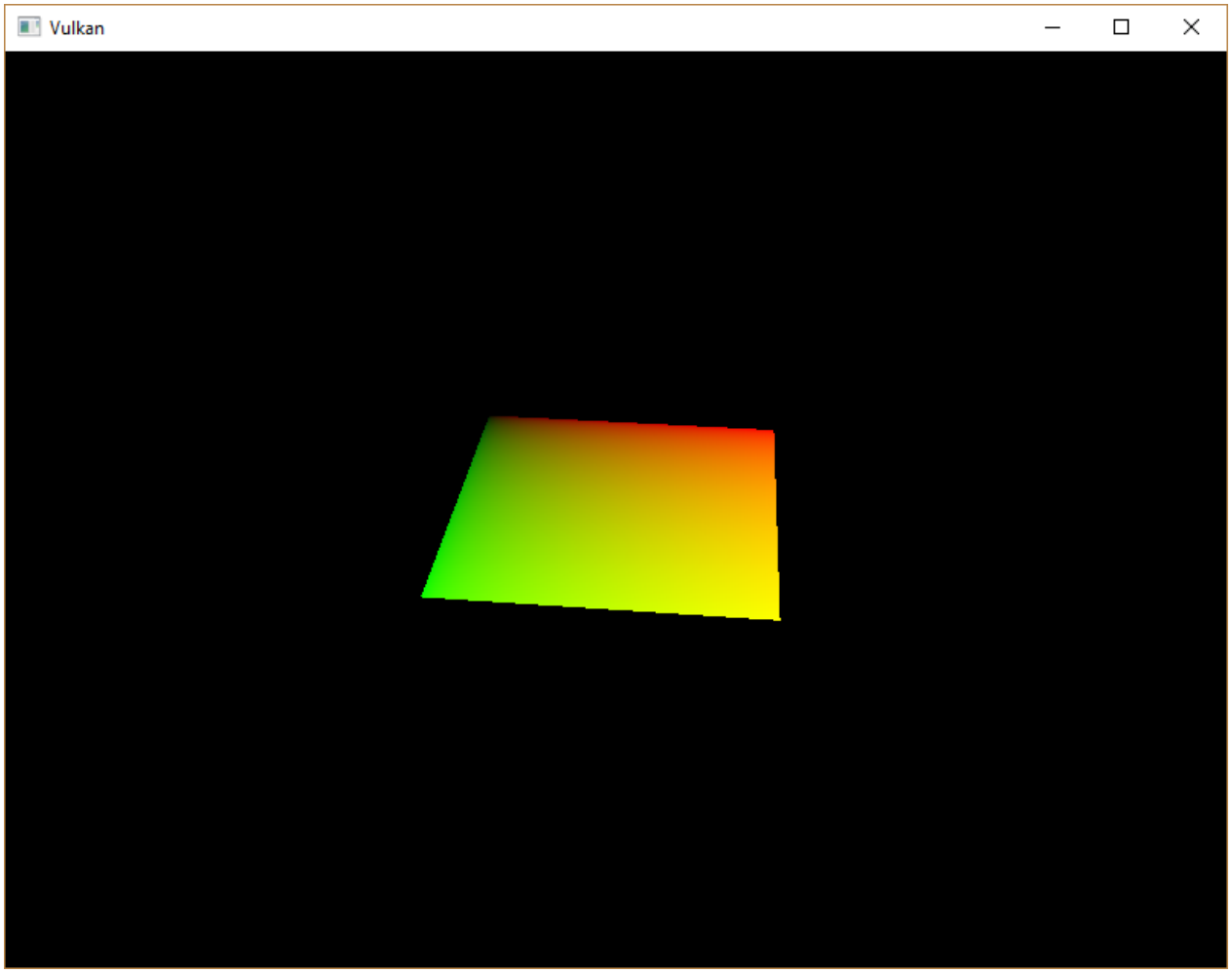


Code C++ / Vertex shader / Fragment shader

Multisampling

Introduction

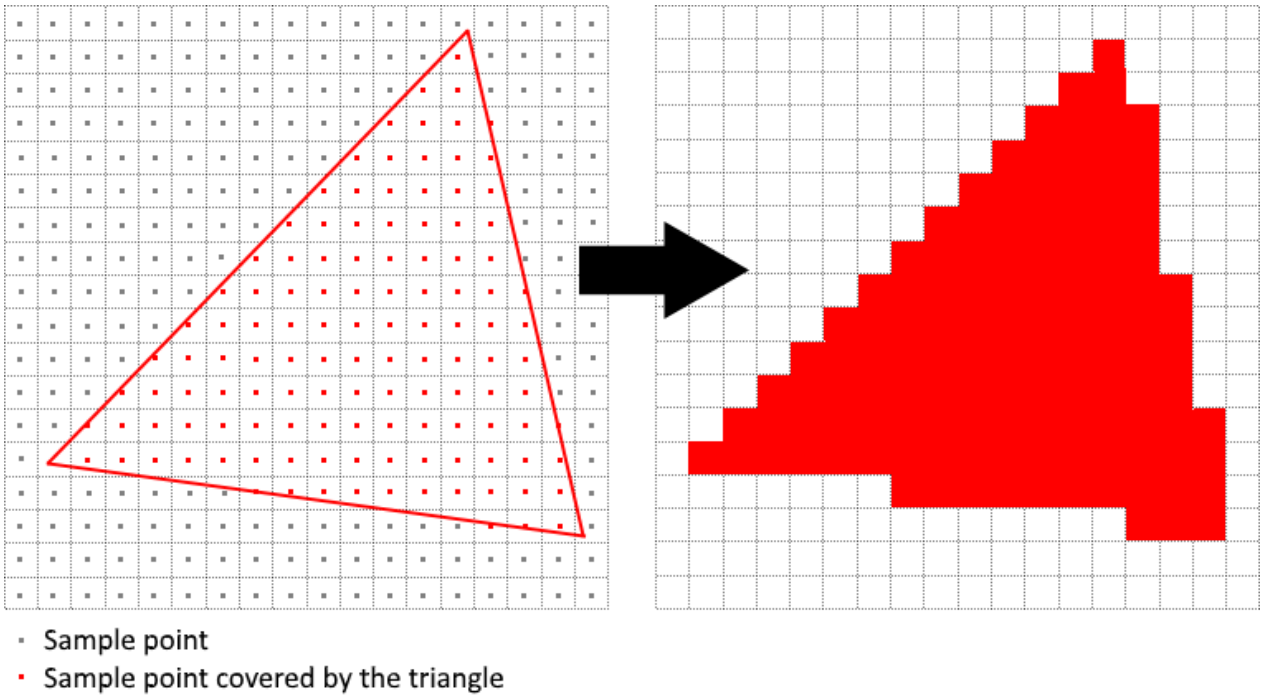
Notre programme peut maintenant générer plusieurs niveaux de détails pour les textures qu'il utilise. Ces images sont plus lisses quand vues de loin. Cependant on peut voir des motifs en dent de scie si on regarde les textures de plus près. Ceci est particulièrement visible sur le rendu de carrés :



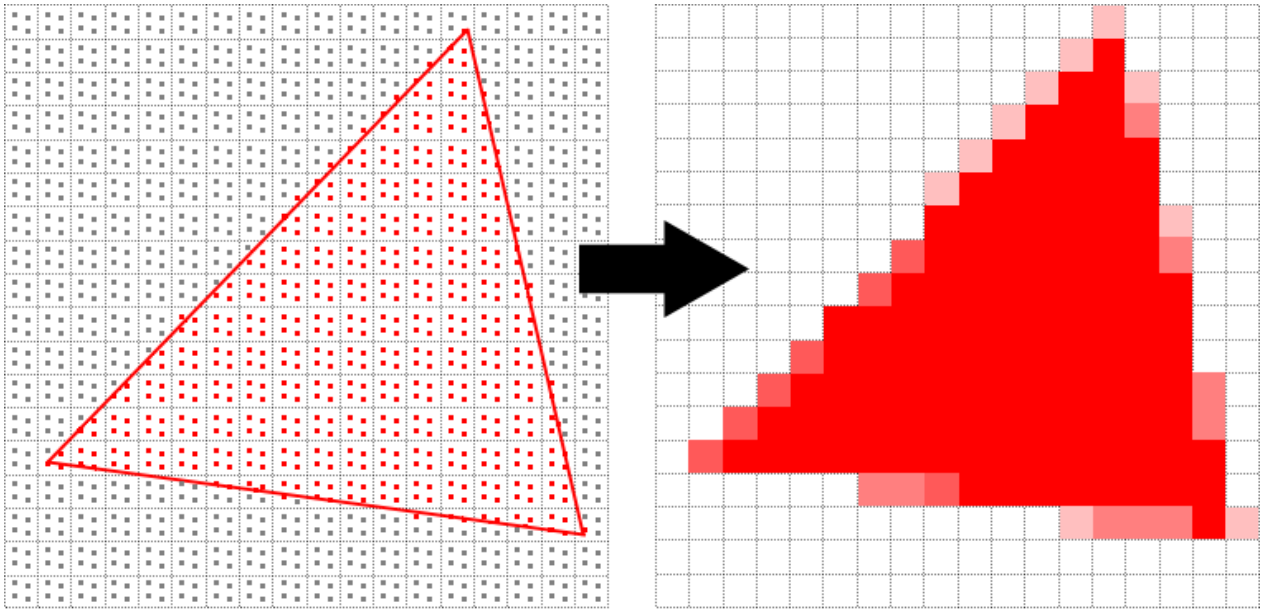
Cet effet indésirable s'appelle "aliasing". Il est dû au manque de pixels pour afficher tous les détails de la géométrie. Il sera toujours visible, par contre nous pouvons utiliser des techniques pour le réduire considérablement. Nous allons ici implémenter le multisample anti-aliasing, terme condensé en MSAA.

Dans un rendu standard, la couleur d'un pixel est déterminée à partir d'un unique sample, en général le

centre du pixel. Si une ligne passe partiellement par un pixel sans en toucher le centre, sa contribution à la couleur sera nulle. Nous voudrions plutôt qu'il y contribue partiellement.



Le MSAA consiste à utiliser plusieurs points dans un pixel pour déterminer la couleur d'un pixel. Comme on peut s'y attendre, plus de points offrent un meilleur résultat, mais consomment plus de ressources.



Nous allons utiliser le maximum de points possible. Si votre application nécessite plus de performances, il vous suffira de réduire ce nombre.

Récupération du nombre maximal de samples

Commençons par déterminer le nombre maximal de samples que la carte graphique supporte. Les GPUs modernes supportent au moins 8 points, mais il peut tout de même différer entre modèles. Nous allons stocker ce nombre dans un membre donnée :

```
...
VkSampleCountFlagBits msaaSamples = VK_SAMPLE_COUNT_1_BIT;
...
```

Par défaut nous n'utilisons qu'un point, ce qui correspond à ne pas utiliser de multisampling. Le nombre maximal est inscrit dans la structure de type `VkPhysicalDeviceProperties` associée au GPU. Comme nous utilisons un buffer de profondeur, nous devons prendre en compte le nombre de samples pour la couleur et pour la profondeur. Le plus haut taux de samples supporté par les deux (&) sera celui que nous utiliserons. Créez une fonction dans laquelle les informations seront récupérées :

```
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 maintenant utiliser cette fonction pour donner une valeur à `msaaSamples` pendant 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;
        }
    }
    ...
}
```

Mettre en place une cible de rendu

Le MSAA consiste à écrire chaque pixel dans un buffer indépendant de l'affichage, dont le contenu est ensuite rendu en le résolvant à un framebuffer standard. Cette étape est nécessaire car le premier buffer est une image particulière : elle doit supporter plus d'un échantillon par pixel. Il ne peut pas être utilisé comme framebuffer dans la swap chain. Nous allons donc devoir changer notre rendu. Nous n'aurons besoin que d'une cible de rendu, car seule une opération de rendu n'est autorisée à s'exécuter à un instant donné. Créez les membres données suivants :

```
...
VkImage colorImage;
VkDeviceMemory colorImageMemory;
VkImageView colorImageView;
...
```

Cette image doit supporter le nombre de samples déterminé auparavant, nous devons donc le lui fournir durant sa création. Ajoutez un paramètre `numSamples` à la fonction `createImage` :

```
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;
    ...
```

Mettez à jour tous les appels avec `VK_SAMPLE_COUNT_1_BIT`. Nous changerons cette valeur pour la nouvelle image.

```
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 buffer de couleur à plusieurs samples. Créez la fonction `createColorResources`, et passez `msaaSamples` à `createImage` depuis cette fonction. Nous n'utilisons également qu'un niveau de mipmap, ce qui est nécessaire pour conformer à la spécification de Vulkan. Mais de toute façon cette image n'a pas besoin de mipmaps.

```
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 mettons cette fonction juste avant `createDepthResource`.

```
void initVulkan() {
    ...
    createColorResources();
    createDepthResources();
    ...
}
```

Nous avons maintenant un buffer de couleurs qui utilise le multisampling. Occupons-nous maintenant de la profondeur. Modifiez `createDepthResources` et changez le nombre de samples utilisé :

```
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 quelques ressources, nous devons les libérer :

```
void cleanupSwapChain() {
    vkDestroyImageView(device, colorImageView, nullptr);
    vkDestroyImage(device, colorImage, nullptr);
    vkFreeMemory(device, colorImageMemory, nullptr);
    ...
}
```

Mettez également à jour `recreateSwapChain` pour prendre en charge les recréations de l'image couleur.

```
void recreateSwapChain() {
    ...
    createGraphicsPipeline();
    createColorResources();
    createDepthResources();
    ...
}
```

Nous avons fini le paramétrage initial du MSAA. Nous devons maintenant utiliser ces ressources dans la pipeline, le framebuffer et la render pass!

Ajouter de nouveaux attachements

Gérons d'abord la render pass. Modifiez `createRenderPass` et changez-y la création des attachements de couleur et de profondeur.

```
void createRenderPass() {
    ...
    colorAttachment.samples = msaaSamples;
    colorAttachment.finalLayout = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL;
    ...
    depthAttachment.samples = msaaSamples;
    ...
```

Nous avons changé l'organisation finale à `VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL`, car les images qui utilisent le multisampling ne peuvent être présentées directement. Nous devons la convertir en une image plus classique. Nous n'aurons pas à convertir le buffer de profondeur, dans la mesure où il ne sera jamais présenté. Nous avons donc besoin d'un nouvel attachement pour la couleur, dans lequel les pixels seront résolus.

```
    ...
    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 render pass doit maintenant être configurée pour résoudre l'attachement multisamplé en un attachement simple. Créez une nouvelle référence au futur attachement qui contiendra le buffer de pixels résolus :

```
    ...
    VkAttachmentReference colorAttachmentResolveRef{};
    colorAttachmentResolveRef.attachment = 2;
    colorAttachmentResolveRef.layout = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL;
    ...
```

Ajoutez la référence à l'attachement dans le membre `pResolveAttachments` de la structure de création de la subpasse. La subpasse n'a besoin que de cela pour déterminer l'opération de résolution du multisampling :

```
    ...
    subpass.pResolveAttachments = &colorAttachmentResolveRef;
    ...
```

Fournissez ensuite l'attachement de couleur à la structure de création de la render pass.

```
    ...
    std::array<VkAttachmentDescription, 3> attachments = {colorAttachment, depthAttachment,
        colorAttachmentResolve};
    ...
```

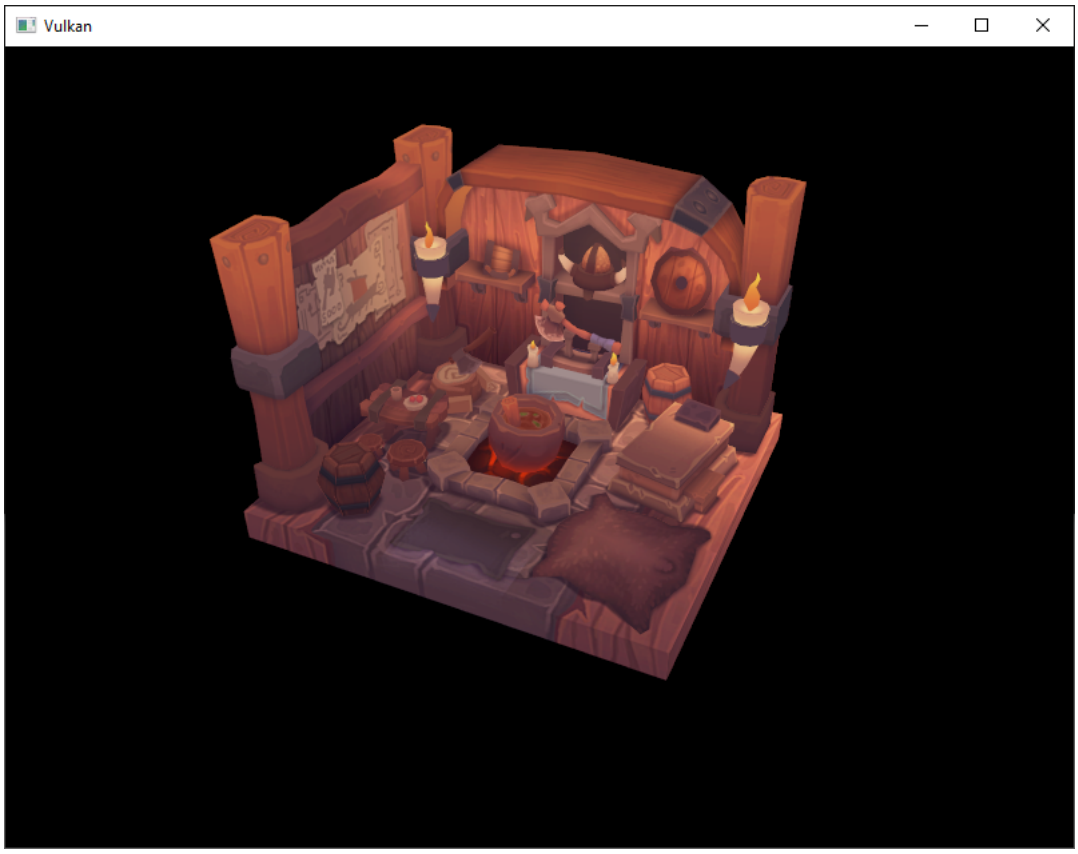
Modifiez ensuite `createFramebuffer` afin de d'ajouter une image view de couleur à la liste :

```
void createFrameBuffers() {
        ...
        std::array<VkImageView, 3> attachments = {
            colorImageView,
            depthImageView,
            swapChainImageViews[i]
        };
        ...
}
```

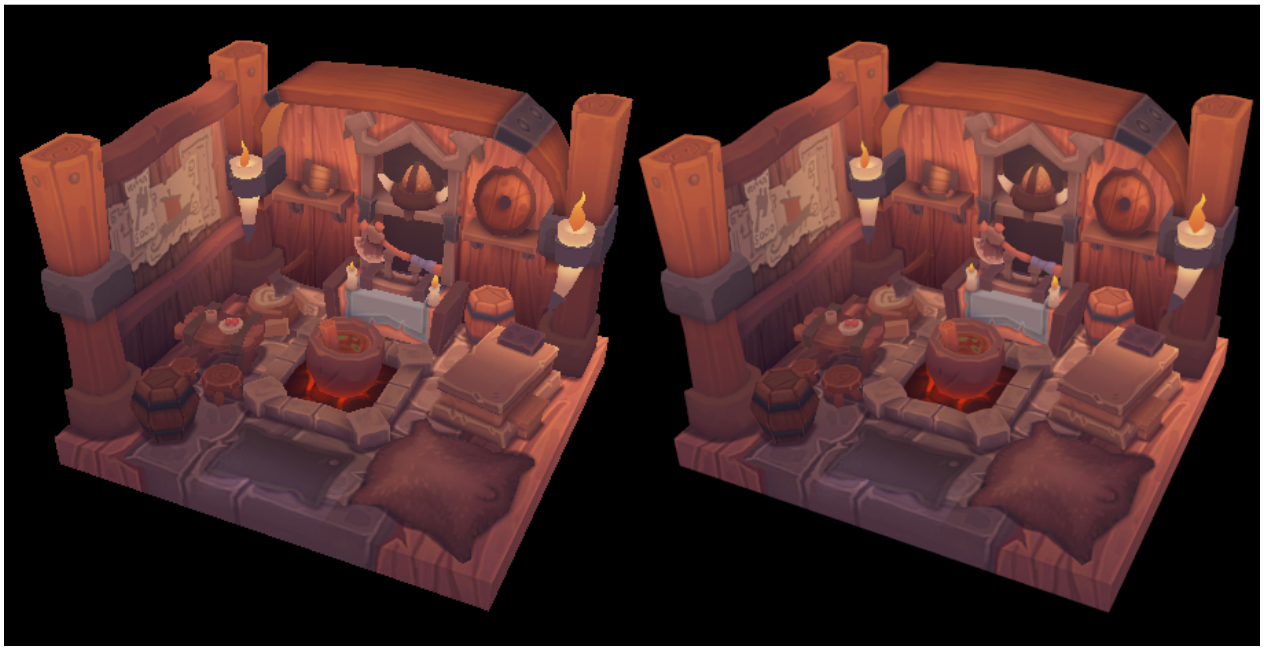
Il ne reste plus qu'à informer la pipeline du nombre de samples à utiliser pour les opérations de rendu.

```
void createGraphicsPipeline() {
    ...
    multisampling.rasterizationSamples = msaaSamples;
    ...
}
```

Lancez votre programme et vous devriez voir ceci :



Comme pour le mipmapping, la différence n'est pas forcément visible immédiatement. En y regardant de plus près, vous pouvez normalement voir que, par exemple, les bords sont beaucoup plus lisses qu'avant.



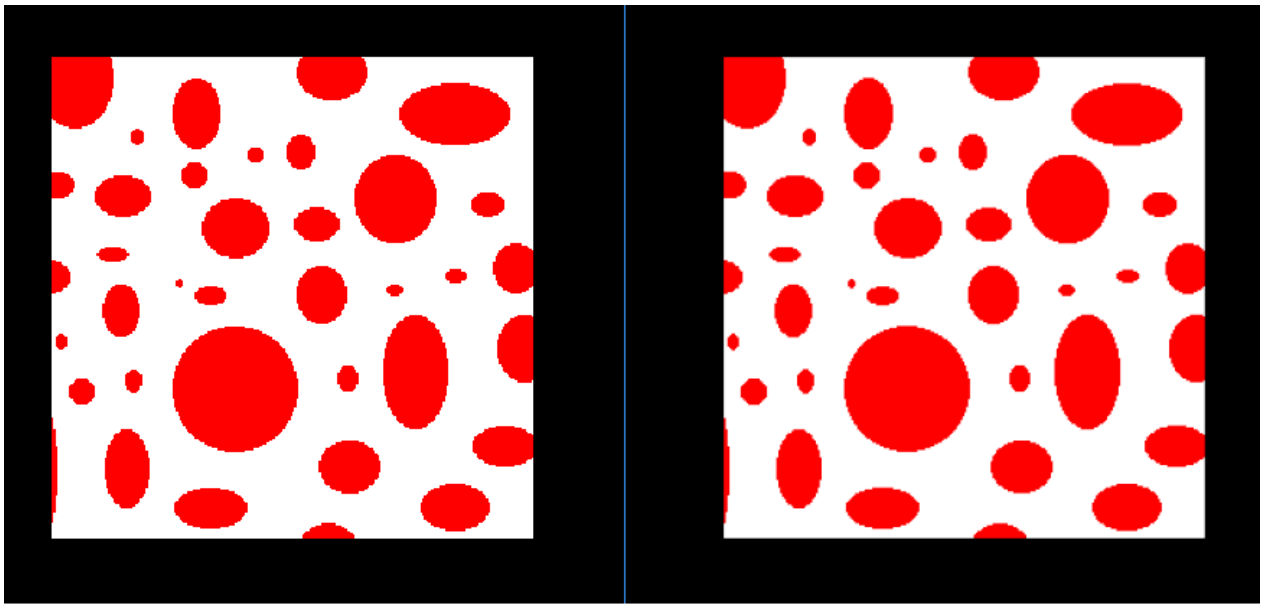
La différence est encore plus visible en zoomant sur un bord :

Amélioration de la qualité

Notre implémentation du MSAA est limitée, et ces limitations impactent la qualité. Il existe un autre problème d'aliasing dû aux shaders qui n'est pas résolu par le MSAA. En effet cette technique ne permet que de lisser les bords de la géométrie, mais pas les lignes contenus dans les textures. Ces bords internes sont particulièrement visibles dans le cas de couleurs qui contrastent beaucoup. Pour résoudre ce problème nous pouvons activer le sample shading, qui améliore encore la qualité de l'image au prix de performances encore réduites.

```
1
2  void createLogicalDevice() {
3      ...
4      deviceFeatures.sampleRateShading = VK_TRUE; // Activation du sample shading pour le device
5      ...
6  }
7
8  void createGraphicsPipeline() {
9      ...
10     multisampling.sampleShadingEnable = VK_TRUE; // Activation du sample shading dans la
           pipeline
11     multisampling.minSampleShading = .2f; // Fraction minimale pour le sample shading; plus
           proche de 1 lisse d'autant plus
12     ...
13 }
```

Dans notre tutoriel nous désactiverons le sample shading, mais dans certain cas son activation permet une nette amélioration de la qualité du rendu :



Sample Shading disabled

Sample Shading enabled

Conclusion

Il nous a fallu beaucoup de travail pour en arriver là, mais vous avez maintenant une bonne connaissances des bases de Vulkan. Ces connaissances vous permettent maintenant d'explorer d'autres fonctionnalités, comme :

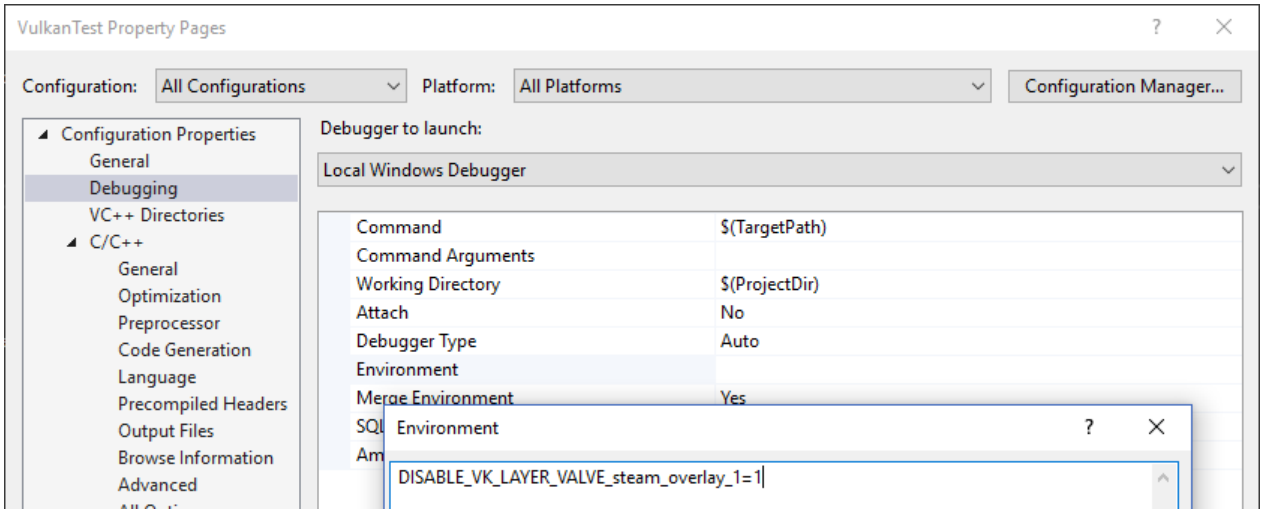
- Push constants
- Instanced rendering
- Uniforms dynamiques
- Descripteurs d'images et de samplers séparés
- Pipeline caching
- Génération des command buffers depuis plusieurs threads
- Multiples subpasses
- Compute shaders

Le programme actuel peut être grandement étendu, par exemple en ajoutant l'éclairage Blinn-Phong, des effets en post-processing et du shadow mapping. Vous devriez pouvoir apprendre ces techniques depuis des tutoriels conçus pour d'autres APIs, car la plupart des concepts sont applicables à Vulkan.

Code C++ / Vertex shader / Fragment shader

FAQ

Cette page liste quelques problèmes que vous pourriez rencontrer lors du développement d'une application Vulkan.

- **J'obtiens un erreur de violation d'accès dans les validations layers** : assurez-vous que MSI Afterburner / RivaTuner Statistics Server ne tournent pas, car ils possèdent des problèmes de compatibilité avec Vulkan.
- **Je ne vois aucun message provenant des validation layers / les validation layers ne sont pas disponibles** : assurez-vous d'abord que les validation layers peuvent écrire leurs message en laissant le terminal ouvert après l'exécution. Avec Visual Studio, lancez le programme avec Ctrl-F5. Sous Linux, lancez le programme depuis un terminal. S'il n'y a toujours pas de message, revoyez l'installation du SDK en suivant les instructions de cette page (section "Verify the Installation"). Assurez-vous également que le SDK est au moins de la version 1.1.106.0 pour le support de `VK_LAYER_KHRONOS_validation`.
- **vkCreateSwapchainKHR induit une erreur dans SteamOverlayVulkanLayer64.dll** : Il semble qu'il y ait un problème de compatibilité avec la version beta du client Steam. Il y a quelques moyens de régler le conflit :
  - Désinstaller Steam
  - Mettre la variable d'environnement `DISABLE_VK_LAYER_VALVE_steam_overlay_1` à `1`
  - Supprimer la layer de Steam dans le répertoire sous `HKEY_LOCAL_MACHINE\SOFTWARE\Khronos\Vulkan\ImplicitLa`

Exemple pour la variable :