ENSIIE STRASBOURG

Rapport de stage

Auteur :
Philippe
GAULTIER,

Élève ingénieur en
troisième année à l'ENSIIE
Strasbourg

Maître de stage :
Toto TITI,

tata tutu

Lausanne, Suisse, le 17 novembre 2015

The road is long and in the end
the journey is the destination

Unknown

Dans toute la suite du rapport, «GFP» désigne l'entreprise «Global Financial Products».

Table des matières

1 Introduction **4**

2 Présentation de EdgeLab **4**
2.1 Histoire . . . 4
2.2 Les équipes . . . 4
2.3 Clients . . . 4
2.4 Objectif du stage . . . 5

3 L'application **5**
3.1 Architecture . . . 5
3.2 Le projet front-end . . . 6
3.2.1 AngularJS . . . 6
3.2.2 Tests unitaires . . . 6
3.2.3 Tests d'intégration . . . 7
3.2.4 L'environnement . . . 8
3.3 Contraintes . . . 8
3.3.1 Skybot 3D . . . 8
3.3.2 Simulation . . . 9
3.4 Déroulement du stage . . . 11
3.5 Développement . . . 12
3.5.1 Bonnes pratiques . . . 12
3.5.2 Design Patterns . . . 12
3.5.3 Architecture . . . 15
3.6 Améliorations - Optimisations . . . 19
3.6.1 Structures de données - Data Locality . . . 19
3.6.2 Smart pointers («Pointeurs intelligents») . . . 20
3.6.3 Logs . . . 21
3.6.4 Problèmes restants - Fonctionnalités à améliorer . . . 21

4 Remerciements **21**

5 Conclusion **22**

6 Annexes **23**
6.1 Design Pattern détaillés . . . 23
6.1.1 Data Locality . . . 23
6.2 Outils utilisés . . . 24
6.2.1 Langages utilisés . . . 24
6.2.2 Bibliothèques utilisées . . . 24
6.2.3 Outils divers utilisés . . . 25
6.3 Captures d'écran . . . 26
6.4 Code source . . . 33
6.5 Glossaire . . . 44

6.6 Ressources 45

Table des figures

1 Diagramme de classe UML pour les objets graphiques 16
2 Diagramme de classe UML pour la scène 17
3 Hiérarchie des classes 18
4 Hiérarchie des fichiers 18
5 Visualisation de la planète Terre dans Skybot 3D en vue normale 27
6 Visualisation de la planète Terre dans Skybot 3D en vue Oculus 27
7 Visualisation du système solaire dans Skybot 3D en vue normale 28
8 Visualisation du système solaire dans Skybot 3D en vue Oculus 28
9 Simulation en vue normale 30
10 Simulation en vue normale 30
11 Simulation en vue normale 32
12 Simulation en vue normale 32

1 Introduction

La Suisse fait partie des leaders mondiaux du secteur financier, et dispose d'un statut particulier en Europe : elle fait partie de l'espace Schengen mais pas de l'Union Européenne, ainsi disposant d'un certain exotisme, comme une monnaie et une législation spécifique, tout en offrant certaines facilités pour les formalités administratives aux travailleurs européens, notamment français.

Lausanne, quatrième ville de Suisse, est particulièrement attrayante de part son emplacement, au bord du lac Léman, et son investissement dans l'éducation supérieure, particulièrement en mathématiques et en informatique, de rang mondial.

Pour toutes ces raisons, et afin de découvrir un milieu du travail différent de la France, j'ai décidé de rejoindre EdgeLab, startup du management du risque financier à Lausanne pour six mois en tant que développeur web.

2 Présentation de EdgeLab

2.1 Histoire

EdgeLab est fondée en ... en tant que filiale de l'entreprise de gestion de portefeuilles financiers Global Financial Products (GFP), suite à la demande de leurs clients d'un outil d'automatisation de gestion de portefeuilles et de calcul de risques pour ces derniers.

Ainsi naît EdgeLab, éditeur logiciel pour le management du risque financier qui se positionne dans le domaine du Business to Business (B2B). Les premiers membres sont des docteurs en mathématiques qui conçoivent le moteur de calcul de risque, puis viennent des développeurs web pour l'application web qui permet aux clients d'interagir avec ce moteur.

2.2 Les équipes

Aujourd'hui, EdgeLab est constitué d'une quinzaine de développeurs composant trois équipes à parts approximativement égales :

— Équipe Quantitative : elle travaille sur le moteur de calcul de risque
— Équipe Back-end : elle développe le back-end de l'application web
— Équipe Front-end : elle développe le front-end de l'application web

2.3 Clients

Les clients d'EdgeLab sont des entreprises du secteur bancaire qui gèrent des instruments financiers et qui sont intéressées par des mesures de risque dans leur processus de décision. De plus, le volume de données et les montants associés requièrent l'association de l'automatisation logicielle et de la supervision par des humains de niveaux d'expertise divers.

Cela inclut un large éventail d'entreprises allant des établissement bancaires aux gestionnaires de fortune, en passant par courtiers en bourse.

Actuellement, l'application est utilisée par des clients de différentes organisations.

2.4 Objectif du stage

Ce stage était centré sur le produit proposé par EdgeLab à ses clients, l'application web, et les objectif étaient multiples, dans plusieurs domaines :

— Développer de nouvelles fonctionnalités
— Améliorer la qualité du code
— Augmenter la couverture de tests
— Simplifier le processus de déploiement
— Améliorer les outils de développement
— Moderniser et simplifier l'interface

3 L'application

L'application web est un projet de grande taille pour les standards du web : environ un millier de fichiers et 15000 lignes de code source. C'est en effet une application client lourd qui suit les standards du web moderne et utilise les outils les plus récents : HTML5, CSS3, et la majorité de la logique en Javascript. Le serveur fournit en fait seulement une API, au même titre que l'API Facebook ou Twitter. Le rendu de pages se fait côté client, de même qu'une partie des calculs financiers, utilisant les capacités des navigateurs modernes au maximum, tout cela dans le but de rendre l'expérience utilisateur plus agréable, fluide et de minimiser les rechargements complets de page, ainsi que les échanges de données réseau, premier facteur de latence.

3.1 Architecture

L'architecture est assez simple et suit le modèle d'une entreprise comme Google. L'idée est d'utiliser l'outil le plus adapté à la tâche. Ainsi, le serveur web (back-end) est développé en Java 8 et utilise le framework réputé pour le développement web en entreprise, Spring.

Les calculs lourds sont délégués au moteur de calcul financier écrit en C++, focalisé sur la performance. Ces calculs sont soit effectués en temps réels pour les plus courts (durée inférieure à 30 secondes), soit lus dans une base de données qui est remplie la nuit, pour les calculs longs et prévisibles (plusieurs heures).

Mais sur quelles données financières s'appuie ce moteur de calcul, dans un contexte de volatilité des marchés ? Les données proviennent en fait de différents fournisseurs de données reconnus, par exemple Bloomberg, et sont actualisées chaque jour. Cependant, certaines données sont parfois contradictoires avec d'autres provenant d'un fournisseur différent. Rarement, le cas de données incomplètes, utilisant un format singulier, ou simplement erronées peut se poser. On comprend alors la nécessité de récupérer, valider, et agréger ces données avant de les utiliser. C'est la responsabilité d'un service à part, écrit en Java, qui va lui aussi chaque nuit effectuer ce long travail portant parfois sur des centaines de milliers de données. Il va rejeter les données suspectes puis va remplir la base de données avec les bonnes données, qui pourront alors être utilisées par le moteur de calcul, ou bien tout simplement par le back-end web qui les transmettra au front-end afin de les afficher à l'utilisateur.

À l'heure du trading haute fréquence, de l'ordre de la milliseconde, l'actualisation des données quotidienne peut ne pas sembler être assez. Au contraire, dans le contexte de la prévision et de l'analyse du risque à moyen ou long terme, et en tant qu'outil d'aide à la décision, cette fréquence est en fait suffisante. Si un événement perturbe les marchés, il faut simplement de

nouveau effectuer le calcul de risque pour les mois ou années à venir, et réévaluer ses décisions stratégiques.

3.2 Le projet front-end

3.2.1 AngularJS

L'application front-end utilise le framework open-source AngularJS, initialement développé par Google. Il permet d'organiser son project de manière cohérente, en séparant :

— Vues : l'interface avec laquelle l'utilisateur intéragit
— Contrôleurs : la logique des vues et la gestion des événements
— Services : les fonctionnalités génériques utilisées dans les contrôleurs
— Directives : les petits composants réutilisables (par exemple un datetimepicker sur un formulaire)
— Configuration

L'utilisation d'un framework bien connu est nécessaire, sinon indispensable, dans ce domaine assez peu structuré et évoluant très rapidement qu'est le développement front-end, ainsi que dans le contexte d'utilisation du très flexible et piégeux langage Javascript.

En sus, AngularJS automatise de nombreuses tâches indispensables à une expérience utilisateur riche : TODO

De plus, un avantage majeur d'AngularJS comparé aux autres frameworks, est le fait qu'il ait été conçu avec les tests en tête. Il fournit de nombreuses aides pour tests son code, allant même jusqu'à proposer une librairie de tests d'intégration, chose inédite dans l'écosystème Javascript.

Enfin, la communauté a travaillé dur pour procurer tout un assortiment de modules open sources pour répondre aux besoins courants du développement web.

Ma contribution pour ce sujet a consisté à appliquer les conventions préconisées par la communauté (John Papa's style guide). Cette uniformisation du nommage, de l'indentation, et des pratiques a nécessité un refactoring du projet, chose représentant un défi particulier pour le langage dynamique, faiblement typé et manquant de réelles capacités orienté objet qu'est Javascript.

3.2.2 Tests unitaires

Un code non testé doit être considéré invalide jusqu'à preuve du contraire. Partant de ce constat, nous avons considérablement développé les tests unitaires existant, couvrant un nombre réduit de cas, passant ainsi de 250 à un millier de tests. En plus d'écrire des tests pour le code existant, ce qui a permis de mettre en lumière certains bugs, ou simplement de considérer certains cas pas forcément pris en compte, l'écriture de tests pour le nouveau code a été institutionnalisée. Cela a eu comme résultat visible des dizaines de fichiers couverts à 100%, pour une moyenne de couverture du projet de 40%, expliquée en partie par une minorité de fichiers ne possédant aucun tests.

Enfin, cette décision a aussi porté ses fruits pour l'épineux sujet des régressions, qui consiste à une perte ou détérioration des fonctionnalités de l'application à la suite d'un changement. Sans tests, une régression passe souvent inaperçue jusqu'à ce qu'un utilisateur s'y heurte.

Avec une bonne couverture de tests, c'est de l'histoire ancienne : le changement provoque l'échec d'un ou plusieurs tests qui étaient valides auparavant. Ainsi la régression est signalée

instantanément au développeur, qui peut cerner son ampleur et la résoudre, tout cela avant qu'elle touche l'utilisateur final.

Cerise sur le gâteau, lorsqu'un bug est détecté, il suffit au développeur d'ajouter le test correspondant, qui est invalide avant la résolution, et valide après. On s'assure ainsi que le bug a réellement disparu, et une éventuelle réapparition sera automatiquement détectée.

3.2.3 Tests d'intégration

Les tests d'intégration, ou end-to-end, consistent à tester toute la chaîne logicielle, simulant les action d'un utilisateur, afin de vérifier que chaque composant s'imbrique sans problème avec les autres. Ils complètent les tests unitaires car disposent d'une granularité plus grande et deux composants peuvent être unitairement valides tout en s'intégrant de façon invalide, dans le cas par exemple de données échangées sous un format différent pour les deux parties.

Il est donc apparu comme indispensable de combler l'absence de tests d'intégration, surtout en disposant de la librairie Protractor mis à disposition par Angular dans cet objectif.

Pour le front-end, les tests d'intégration consistent à automatiser les interactions d'un éventuel utilisateur : mouvement de souris, clics, scroll, etc... C'est un travail titanesque à l'échelle d'une application complexe, disposant de multiples pages, avec de très nombreuses possibilités. Pourtant, partant du principe énoncé par Lao-Tseu, "Un voyage d'un millier de lieues commence par un premier pas", nous avons entamé ce gros chantier, et écrit de nombreux tests d'intégrations.

L'énorme avantage d'un test d'intégration de ce genre est qu'il permet d'écrire des scénarios. Par example, pour l'authentification d'un utilisateur, on peut écrire le scénario correspondant, en explorant plusieurs cas se produisant en réalité :

- J'entre une mauvaise combinaison d'email/mot de passe
- Je n'entre rien dans les champs, je clique directement sur "Login"
- J'entre la bonne combinaison d'email/mot de passe, puis je me déconnecte
- ...

Le code suivant illustre le premier cas :

```
it('should not change page if the login fails', function() {
    nameInput.sendKeys('toto@edgelab.ch');
    passwordInput.sendKeys('toto');

    loginButton.click();

    expect(isLoggedIn()).toBe(false);
  });
```

Listing 1 – Testing the login with Protractor

Ainsi, ils remplacent les tests manuels, fastidieux, et non-exhaustifs que le développeur est amené à faire à longueur de journée dans son navigateur.

Pour leur exécution, ces tests tournent dans un vrai navigateur, collant ainsi au plus près à la réalité. Encore mieux, au même titre que les tests unitaires, on peut les exécuter dans une multitude de navigateurs et de versions différentes d'un même navigateur, détectant ainsi une éventuelle régression dans une version spécifique d'un navigateur spécifique.

Dans notre cas, nous nous sommes limités à la dernière version de Chrome, Firefox, et Internet Explorer.

3.2.4 L'environnement

On juge un artisan à ses outils. C'est aussi le cas pour un développeur. Ses outils sont au moins aussi important que le code qu'il produit, car il passera probablement plus de temps à compiler, tester, débugguer et déployer son code qu'à l'écrire. En conséquence, ces outils conditionneront sa productivité et la qualité du code qu'il engendre.

Pour toutes ces raisons, il nous est apparu vital d'améliorer la chaîne d'outils existants, qui bien qu'efficace, nous a semblé complexe à utiliser, à maintenir, et à améliorer. Les attributions de ces "méta-outils" de développement sont divers mais tous visent à faciliter le travail du développeur, avec un devis : tout ce qui peut être automatisé doit être automatisé. Voici les principales attributions de cet environnement :

— Installer les dépendances grâce à un gestionnaire de paquets
— Lancer le projet en mode debug
— Optimiser, minifier, concaténer et compresser le code pour le déploiement : le résultat est un "artefact"
— Déployer sur un serveur distant l'artefact
— Lancer l'analyse statique et vérifier les conventions de code
— Lancer les tests unitaires et/ou d'intégration
— Afficher une page d'aide expliquant les différentes actions possibles

Tout cela pour quatre environnements différents (développement, serveur de test, pré-production, production), avec dans l'idéal une seule commande pour chaque tâche. Après ce travail de refonte de cette chaîne d'outils, il est désormais possible de lancer l'une ou l'autre tâche avec une seule commande, dans l'environnement de notre choix, ce qui est un grand pas en avant non seulement pour la productivité, mais aussi pour la qualité du projet.

Les implications peuvent paraître minimes, mais elles sont en fait énormes. Il s'agit de bousculer les habitudes de travail des développeurs, en menant la vie dure aux mauvaises pratiques, et en encourageant et facilitant les bonnes.

Une autre conséquence est l'abaissement de la barrière d'entrée du projet : un nouveau développeur peut plus vite se focaliser sur le code en étant libéré des autres tâches "ingrates", qui sont automatisées, et en suivant tout de suite les bonnes conventions, son travail s'intégrera plus facilement avec le reste de l'équipe.

3.3 Contraintes

3.3.1 Skybot 3D

Existant

La contrainte principale pour le projet Skybot 3D était de travailler avec du code existant non documenté, de taille conséquente (3840 fichiers, 22863 lignes de code). En effet, j'avais accès à une version de développement non finalisée et non encore publiée. Cependant j'ai eu

de riches échanges avec les développeurs par mail et vidéoconférence, et j'ai eu la chance de les rencontrer à la fin du stage.

Langages

Une contrainte supplémentaire a été le conflit de langages : le programme existant est écrit en C, et le SDK Oculus en C++, exigeant en conséquence un compilateur C++. C et C++ sont des langages proches de par leur origine et leur histoire, C++ étant issu de C, ils sont donc en grande partie compatibles. La majorité du code n'a donc pas posé de problème, mais certains motifs ont dû être modifiés, notamment les conversions de types implicites, les arithmétiques de pointeurs et les pointeurs de fonctions ont du être réécrits de manière idiomatique en C++.

Versions d'OpenGL

Une contrainte additionnelle, et peut-être la plus importante, était l'utilisation de différentes fonctionnalités d'OpenGL, appartenant à des versions différentes. En effet, la totalité du rendu graphique dans l'application existante se fait avec le «fixed pipeline» d'OpenGL, c'est-à dire une suite d'opérations fixes de rendu. Cela consiste a faire un rendu graphique basé sur des appels à des fonctions OpenGL qui fournissent des fonctionnalités bien pratiques, comme des transformations matricielles, des lumières, ... Cependant ces appels, typiques d'OpenGL 1.x et 2.x, utilisent principalement le CPU et ont donc été dépréciés pour des raisons de performances dans OpenGL 3.x et 4.x, au profit d'une programmation «tout shader». Les shaders sont des programmes appliquant des effets sur chaque pixel de l'image et qui sont exécutés sur la carte graphique.

Le développeur doit donc maintenant tout faire manuellement, au profit des performances et de l'éventail de possibilités, mais au détriment de la simplicité.

Le SDK Oculus utilise les shaders pour appliquer les effets graphiques mentionnés plus tôt (distorsion, correction des aberrations, ...), et cela a créé quelques conflits au niveau du rendu graphique, avec le rendu existant n'utilisant pas ces shaders.

Échelles

Skybot 3D est une simulation du système solaire et en tant que telle manipule des distances très grandes. Cela n'est pas gênant sauf lorsque deux distances très grandes sont multipliées entre elles, par exemple dans un calcul matriciel. Cela peut provoquer un «overflow» («dépassement»), phénomène consistant en une variable ayant une valeur plus grande que ce que son type peut stocker en mémoire. Dans le meilleur des cas cela occasionne des erreurs d'arrondis causant des tremblements de la caméra ou des objets, des effets de «flashing» et «tearing» dans le rendu, et dans le pire des cas un crash du programme.

Dans mon cas, ce phénomène a provoqué un phénomène de «cross-eye» gênant pour l'utilisateur en mode Oculus (cf paragraphe 3.6.4).

3.3.2 Simulation

Portabilité

> I don't care if it works on your machine! We are not shipping your machine!
>
> Vidiu Platon

Pour ce projet, nous avons convenus dès le départ d'assurer la portabilité du programme, c'est-à-dire le fonctionnement multiplateforme.

Dans cette optique, j'ai choisi un langage fonctionnant sur n'importe quelle plateforme existante (dès lors qu'il existe un compilateur adéquat), le C++, et des bibliothèques multiplateformes, notamment pour le rendu graphique, pour permettre l'abstraction d'une API spécifique à une plateforme donnée, en fournissant une API générique. Un exemple est l'utilisation de la SDL, une bibliothèque de fenêtrage, ou d'OpenGL, une API de rendu graphique.

De plus, un soin particulier a été porté dans le développement à éviter l'introduction de code spécifique à une plateforme donnée. Une solution a été l'utilisation maximale de la libraire standard du langage.

Enfin j'ai utilisé un outil de compilation multi-plateforme, qmake, qui permet de compiler le code sur plusieurs plateformes différentes automatiquement.

Au final, j'ai uniquement travaillé sur Linux mais le code fonctionne selon toute probabilité sur Windows et OSX, avec des drivers à jour, sur des versions relativement récentes de ces systèmes d'exploitation.

Taille des données

Comme évoqué plus haut, j'ai travaillé sur des données avoisinant le million d'objets. Ces données proviennent de simulations à haute résolution de la réionisation de galaxies de la voie lactée. Cela a posé principalement un problème de performances. En effet, c'est en travaillant avec de tels nombres que l'on se rend compte de la disparité CPU (processeur) / GPU (carte graphique). En effet, malgré un processeur avec 16 GB de RAM, le fait de parcourir tous les objets pour les afficher (une fois par frame, $\mathcal{O}(n)$), prenait plus de 16 millisecondes, nombre critique dans le domaine du rendu graphique, puisqu'il correspond au temps de rendu maximal d'une frame si l'on veut un rendu à 60 FPS (frame per seconds), ce qui fournit une expérience correcte pour l'utilisateur : $1000ms/60 = 16.666ms$.

En sus, comme j'ai travaillé avec des +cubes de données (corps célestes dont les cooordonnées spatiales se trouvent toutes contenues dans un cube, typiquement de taille 64*64*64 ou 100*100*100), ce qui peut donner une boucle de rendu de complexité $\mathcal{O}(n^3)$, empirant alors le temps de rendu.

De plus, il faut garder à l'esprit que l'objectif final est d'avoir un rendu Oculus valide. Or le SDK Oculus fait un double rendu (un pour chaque oeil), en appliquant des transformations matricielles pour chacun des rendus. Il est donc primordial d'avoir des FPS corrects dans le rendu graphique normal.

Cependant, je me suis aperçu que la carte graphique ne rencontrait pas de problème de temps de rendu, gardant la plupart du temps un temps de rendu d'une frame inférieur à la milliseconde.

Dès lors, plusieurs solutions se sont présentées :

Travailler avec un seul objet graphique
: Cela consiste à avoir un seul objet dans le programme qui contient les coordonnées de tous les objets célestes. On «boucle» donc sur un seul objet et on envoie toute les

positions des objets célestes en une seule fois comme s'il n'y avait q'un objet et la carte graphique fait tout le travail. Cela fonctionne mais est peu flexible (comment faire pour sélectionner un seul objet céleste pour afficher des informations à son sujet ?) et on atteint les limites de la carte graphique pour un très grand nombre d'objets. Cependant le CPU a un minimum de travail.

Octree
Un Octree (cf figure **??**) est un arbre où chaque noeud (appelé «octant») compte jusqu'à 8 fils. Il correspond à partition d'un espace cubique, à la manière d'un quadtree en 2D, et permet de diviser notre scène en régions, contenant elles-mêmes des sous-régions et ainsi de suite. On peut alors décider d'afficher seulement les régions voisines de notre position sans afficher les régions que l'on ne peut pas voir ou qui sont trop lointaines. C'est la solution que j'ai choisie car c'est la plus flexible et celle qui offre le plus de possibilités. A noter cependant que cela impose une taille de cube d'une puissance de deux.

On optimise alors le rendu à la fois sur le CPU (moins d'objets parcourus dans la boucle de rendu à chaque frame) et sur le GPU (moins de données envoyées et rendues graphiquement à chaque frame).

Initialement, mes objets étaient tous stockés dans un tableau que je parcourais à chaque frame pour les afficher. Pour 1000 objets, j'avais une moyenne de 5 FPS. J'ai alors mis en place un Octree. J'ai eu alors des FPS variant entre 30 et 60 FPS, avec une moyenne de 55 FPS (le rendu est limité à 60 FPS maximum pour ne pas surcharger le CPU/GPU), quelque soit le nombre d'objets présents dans la scène, ce qui est acceptable. En effet, l'Octree permet d'afficher un nombre moyen constant d'objets quelque soit notre position. Avec un cube de taille 128*128*128 et 32768 objets, le temps de génération de l'Octree est d'environ 120s. J'affiche l'octant (et donc tous les objets célestes se trouvant dans cet octant) où la caméra se trouve et les 6 octants immédiatement voisins, avec une taille d'octant de 8. Pour résumer, on sacrifie le temps de démarrage du programme au profit des performances à l'exécution.

3.4 Déroulement du stage

> Premature optimization is the source of all evil
>
> Donald Knuth

Mon stage s'est principalement déroulé en trois temps : formation, développement, optimisation.

La première semaine a été consacrée à la découverte de l'Oculus Rift, l'essai de démos, et l'installation des outils de développement.

Les deuxième et troisième semaines ont consisté à se former à l'Oculus SDK et OpenGL, et à tester la faisabilité de l'intégration de l'Oculus Rift à un moteur de rendu 3D existant, Irrlicht. Malheureusement cela s'est révélé plus complexe que prévu et la décision a été prise d'utiliser OpenGL sans framework, vu le temps imparti. Cette période a aussi servi à se (re)former au C++. C'est un langage que j'avais déjà utilisé sur d'anciens projets mais les nouveautés introduites par C++11 m'étaient inconnues.

Les quatrième et cinquième semaines ont été centrées sur le développement de programmes minimalistes mettant en place OpenGL et l'intégration de l'Oculus Rift, et la découverte du code source de Skybot 3D.

Les sixième, septième et huitième semaines ont été axées sur le développement de mes deux projets.

Les neuvième et dixième semaines ont porté sur la rédaction de ce rapport et sur les possibles optimisations des deux projets, ainsi que la refactorisation du code et la documentation complète.

Je me suis toutefois formé tout au long de mon stage à OpenGL, notamment à propos des différences entre les versions 1.x/2.x et 3.x/4.x, où la philosophie du «tout shader» a été introduite, et la compatibilité entre ces versions.

3.5 Développement

3.5.1 Bonnes pratiques

> Always code as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live
>
> — Martin Golding

> Programs must be written for people to read, and only incidentally for machines to execute
>
> — Arold Abelson

Comme mentionné plus tôt, j'ai tout au long de mon stage versionné mon code avec git et Github, ce qui a permis de le partager facilement avec l'équipe Skybot 3D et André SCHAAFF.

De plus j'ai régulièrement utilisé des outils d'analyse statique et dynamique pour jauger de la qualité de mon code et l'améliorer.

En sus, j'ai activé les options de compilation aidant dans cette tâche : «-Wall», «-Wextra», «-Werror», ...

J'ai également entièrement documenté mon code avec l'outil dédié Doxygen, et gardé une constance dans le nommage des variables, l'indentation du code, ...

Enfin, j'ai utilisé deux compilateurs différents, clang et gcc, en comparant leurs performances, notamment avec les options d'optimisations «-O1», «-O2», «-O3», «-Ofast». Pour finir, j'ai utilisé le déboggeur gdb et l'analyseur de mémoire Valgrind pour vérifier que mon application était exempte de fuites mémoires (mis à part celles provenant du SDK Oculus pour lesquelles je ne peux rien faire).

3.5.2 Design Patterns

> Controlling complexity is the essence of computer programming
>
> — Brian Kernighan

Les design patterns sont des motifs de programmation, des «façons de faire» que l'on retrouve régulièrement pour résoudre des problèmes bien connus. Ce ne sont pas des «tours de magie» ou des «gadgets à la mode», mais des solutions pratiques à mettre en place quand le besoin s'en fait sentir, ou pour améliorer le code.

Pendant ce stage, j'ai utilisé plusieurs design patterns. Voici lesquels et pourquoi :

NullObject pattern

Ce design pattern sert à gérer élégamment la situation où l'on a un ensemble d'objets à gérer dont certains sont nuls, sans savoir lesquels à l'avance, ou sans vouloir vérifier à chaque utilisation. Plutôt que d'écrire des dizaine de conditions pour vérifier si l'on peut bien effectuer telle ou telle action afin d'éviter une segmentation fault, la variable sur laquelle nous sommes en train de travailler étant nulle (c'est-à-dire ayant pour valeur le pointeur «NULL» ou «nullptr» en C++11), ce pattern propose de travailler avec une classe générique. De cette dernière hérite(nt) la ou les classe(s) dont nous nous servons en pratique dans notre programme, et une classe «NullObject» dont le corps des méthodes sont vides (ou adaptées selon la situation). Ainsi au lieu de travailler avec des pointeurs nuls, on travaille avec de vrais objets, inoffensifs lorsque l'on intéragit avec eux, grâce au polymorphisme. Ce faisant on peut aussi élégamment activer/désactiver des services.

Dans mon cas, j'ai travaillé avec la classe «GraphicObject», dont héritent les classes «Cube», «Plane», ..., mais aussi «NullGraphicObject», dont la méthode d'affichage ne faisait rien. Ainsi dans ma boucle de rendu, je pouvais afficher tous mes objets de la scène, sans avoir peur que certaines parties de l'Octree soient vides et que cela occasionne un plantage.

De la même façon, j'ai utilisé ce pattern pour gérer les deux modes de rendu. J'ai une classe «GenericOculus», dont héritent les classes «NullOculus», dont les méthodes sont vides, et «Oculus», où est implémenté le rendu Oculus. Dans le mode normal, j'utilise la classe «NullOculus» qui ne fait rien, tandis qu'en mode Oculus j'utilise la classe «Oculus».

Singleton pattern

Design pattern bien connu et parfois décrié, il permet de limiter le nombre d'instances d'un objet, typiquement à 1.

Je l'ai utilisé pour la classe Oculus, afin d'éviter de faire l'initialisation (et la libération) du SDK Oculus plusieurs fois.

Game Loop pattern

Ce pattern décrit la boucle de rendu typique d'une application de rendu graphique.

Je l'ai utilisé pour l'application de simulation, comme décrit ici (**??**) pour le rendu normal et là (**??**) pour le rendu Oculus.

Flyweight Pattern

Ce pattern sert à améliorer les performances d'une application où sont présents de nombreux objets identiques, mis à part la valeur de certaines propriétés, et qui utilisent des ressources similaires. Au lieu que chaque objet possède une instance de la ressource, générant une perte de performances inutile, ce pattern propose de mettre en commun tout ce qui peut l'être. Chaque objet aura alors un pointeur vers la ressource partagée. Ce partage est transparent pour l'application qui continue à utiliser les objets comme

bon lui semble, en utilisant les informations particulières de chaque objet.

Dans ma situation, la simulation concernait des milliers voire des millions d'objets célestes. Dans les cas que j'ai rencontrés, il s'agissait toujours d'un seul type d'objet céleste, par exemple une étoile. Si l'on choisit de représenter une étoile par un modèle 3D, une sphère par exemple, et une texture plaquée sur cette sphère, et que l'on charge la ressource «texture» à partir d'un fichier, il est inconcevable de lire ce fichier des millions de fois par seconde (à chaque rendu), ou même des millions de fois au démarrage du programme (initialisation des textures des objets célestes). On choisit alors de charger cette ressource une fois pour toute au démarrage. On lit une et une seule fois le fichier. Chaque étoile possède un pointeur vers cette texture partagée. La seule différence entre les étoiles sont les informations contextuelles (position, densité, âge, ...).

On peut alors faire le rendu sans problème de performances : au final, on fait une seule lecture de fichier, et l'on a une seule texture en mémoire, au lieu de milliers/millions auparavant. L'initialisation prend ainsi moins de temps et la libération mémoire également à la fin du programme.

En pratique, l'application de ce design pattern a permis de transformer un démarrage poussif (dizaines de secondes) avec 100 objets célestes, en un démarrage en environ 3s pour 1000 objets.

Factory pattern

Ce pattern permet typiquement de créer dynamiquement des instances d'objets différents, dont le type n'est connu qu'à l'exécution (par exemple conséquence d'un choix utilisateur). Cependant il peut aussi être utilisé pour encapsuler la construction d'un objet derrière une interface qui fait des vérifications diverses à cette occasion.

Dans mon cas, je l'ai utilisé pour implémenter le groupe de textures partagées. En effet lorsqu'un nouvel objet graphique texturé est créé, il fait une requête auprès de la fabrique de textures pour obtenir sa texture. La fabrique vérifie si la texture existe déjà en mémoire. Si oui elle renvoie un pointeur vers cette texture, sinon elle créé la texture correspondante, l'ajoute au groupe de textures partagées, et finalement renvoie un pointeur vers cette texture nouvellement créée.

L'avantage d'avoir une fabrique de textures est que cette dernière peut être utilisée par plusieurs classes différentes : dans mon cas la classe «Crate» («caisse», qui représente un cube texturé) et «Plane», («plan», qui représente un surface plane texturée) utilisent le groupe de textures partagées, et donc font appel à la fabrique. Ainsi cette logique n'est pas inhérente à une classe et peut être utilisée partout dans le programme. De plus si deux classes différentes utilisent la même texture, une seule sera créée, et non deux.

RAII

«Resource Acquisition Is Initialization», l'acquisition de ressources c'est leur initialisation. Ce pattern permet une gestion efficace et simple de la mémoire et est inhérent au C++.

Il repose sur l'assurance fournie par le langage que le destructeur d'un objet sera appelé lorsque l'on sort du bloc où il a été défini. Ainsi, lorsque l'on acquiert une ressource, par exemple ouvrir un fichier, et que cette ressource a besoin d'être libérée plus tard, par exemple fermer ce fichier, on peut automatiser ces deux actions en les plaçant à un endroit adéquat du code.

En pratique, on fait l'acquisition dans le constructeur de l'objet utilisant/représentant

la ressource (d'où le nom), et la libération dans le destructeur. Pour l'utilisateur de la classe ces opérations sont transparentes et il est assuré qu'elles seront exécutées automatiquement.

Pour ma part, j'ai utilisé ce pattern pour l'initialisation et la libération des ressources graphiques et du SDK Oculus.

Data Locality

Ce design pattern est un moyen d'optimiser les performances d'un programme en tirant parti du cache du CPU.

La fragmentation est l'utilisation inefficace de l'espace mémoire. C'est un phénomène qui se produit lorsque sont effectuées de multiples allocations et désallocations de mémoire, engendrant alors une baisse de la capacité mémoire et/ou des performances.

Dans mon cas, j'ai donc transformé mon tableau de pointeurs sur objets célestes en un tableau d'objets célestes. Cela est rendu possible par le fait que je ne dispose que d'un seul type d'objet céleste dans ma simulation. De plus l'usage de pointeurs dans mon programme est majoritaire dans la partie génération et rendu d'objets célestes de part leur nombre. Cependant il est à noter que dans le cas d'un nombre très grand d'objets (de l'ordre du million), il est possible de dépasser la taille de la pile, créant alors un «stack overflow».

J'ai ensuite mesuré les temps d'exécution dans les deux implémentations, exposés au paragraphe 3.6.1 et en ait tiré des conclusions suprenantes.

Voir le paragraphe 6.1.1 pour plus de détails.

3.5.3 Architecture

> Perfection [in design] is achieved, not when there is nothing more to add, but when there is nothing left to take away
>
> Antoine de Saint-Exupéry

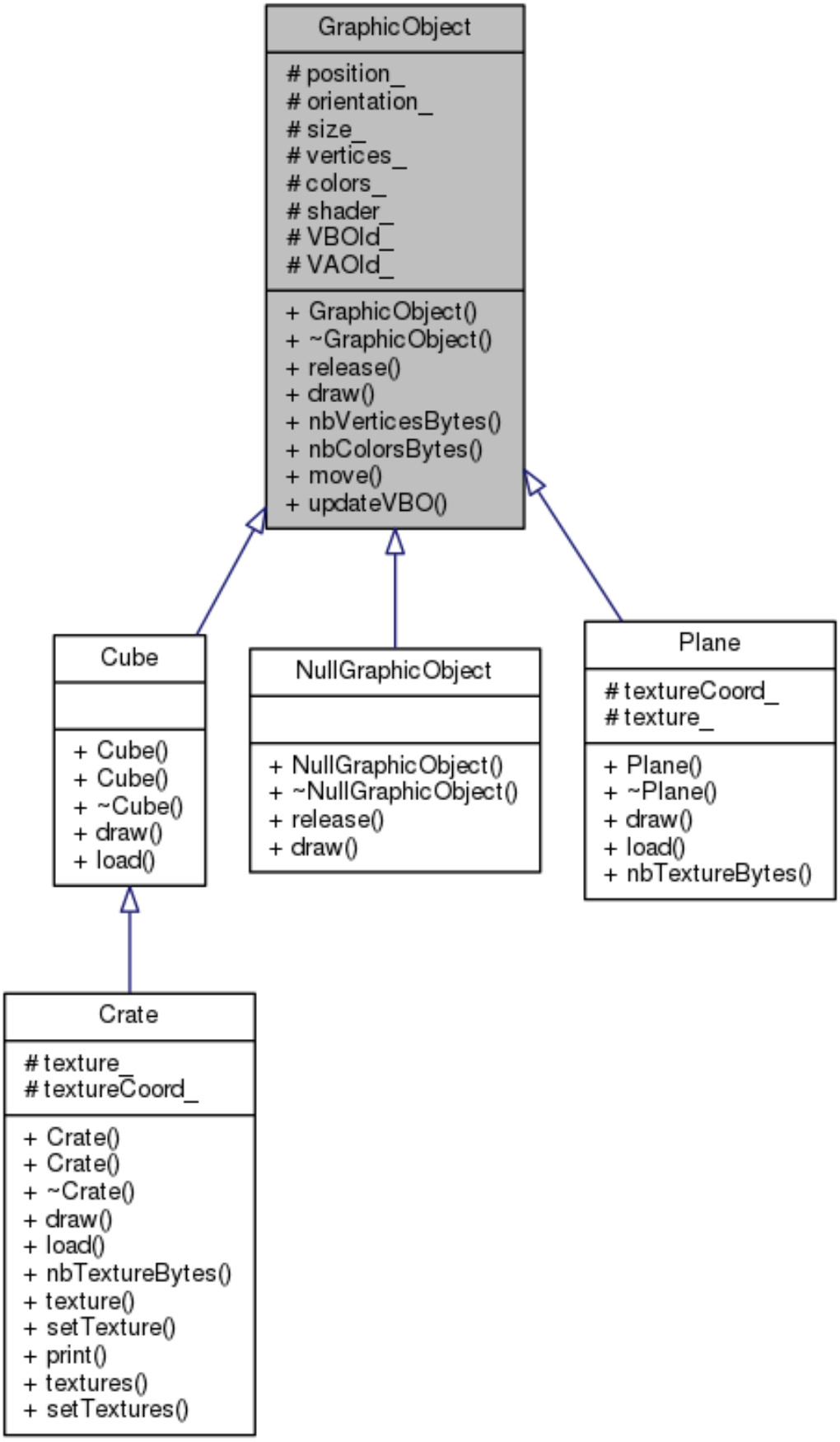

FIGURE 1 – Diagramme de classe UML pour les objets graphiques

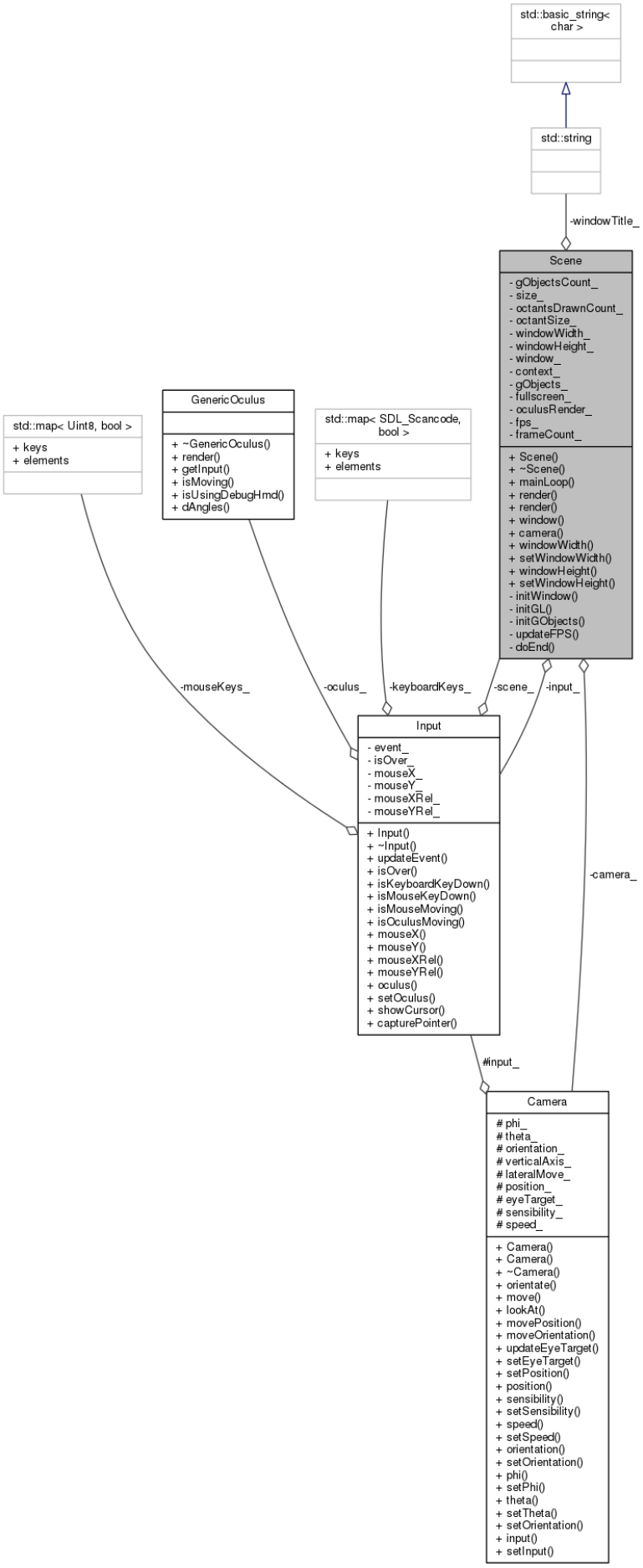

FIGURE 2 – Diagramme de classe UML pour la scène

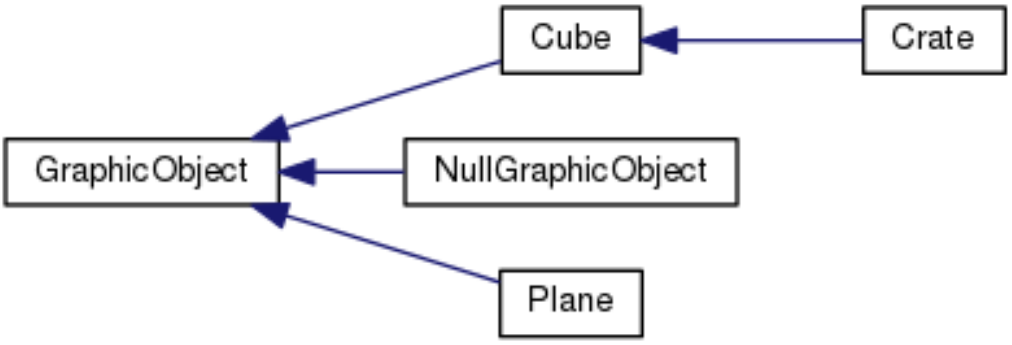

FIGURE 3 – Hiérarchie des classes

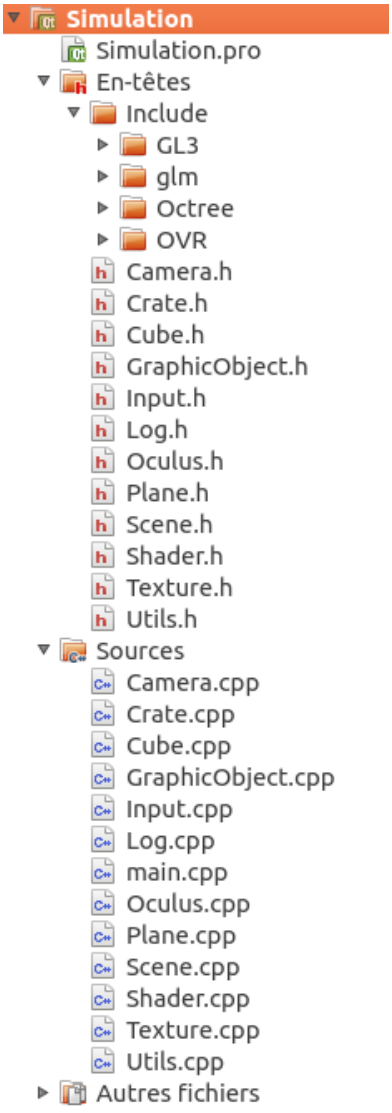

FIGURE 4 – Hiérarchie des fichiers

3.6 Améliorations - Optimisations

3.6.1 Structures de données - Data Locality

Stockage	Méthode	Nombre d'objets	Génération	FPS moyens	Remarques
Tableau	Polymorphisme	1024	3.7 s	37.2	array
Tableau	Polymorphisme	10000	38.2 s	9.5	array
Tableau	Polymorphisme	1024	3.6 s	35.9	vector, ajout avec push_back
Tableau	Polymorphisme	10000	36.4 s	9.6	vector, ajout avec push_back
Tableau	Data locality	1024	11 s	35.6	vector, ajout avec emplace_back
Tableau	Data locality	1024	11 s	34.6	vector, ajout avec push_back
Tableau	Data locality	1024	7.3 s	34.2	vector, ajout avec emplace_back, réservation de la taille requise à l'avance
Tableau	Data locality	1024	7.2 s	33.5	vector, ajout avec push_back, réservation de la taille requise à l'avance
Tableau	Data locality	10000	74.4 s	9.2	vector, ajout avec emplace_back, réservation de la taille requise à l'avance
Tableau	Data locality	10000	71 s	9.2	vector, ajout avec push_back, réservation de la taille requise à l'avance
Octree	Polymorphisme	1024	3.6 s	62.5	
Octree	Polymorphisme	32768	120 s	62.5	

La structure «vector» est un tableau dynamique de la librairie standard. Sa méthode «reserve» permet de réserver une certaine taille à l'avance. Cela permet d'éviter, lors de l'ajout d'un nouvel objet, que la zone mémoire occupée par le «vector» se révèle trop petite, et qu'il faille en réallouer une nouvelle, en y déplaçant tous les éléments existants du «vector». On voit que l'usage de cette méthode fait gagner quelques secondes pour la génération de 1024 objets (on passe de 11s à 7s). La structure «array» est un tableau statique de la librairie standard.

On ne remarque cependant pas de différence majeure de performances entre ces deux structures de données une fois que la taille nécessaire en mémoire est réservée initialement.

Un autre point à noter est la croissance linéaire du temps de génération en fonction du nombre d'objets, ce qui est prévisible étant donné que les structures de données array et vector sont de complexité linéaire pour le parcours et l'ajout.

Cependant l'évolution du nombre de FPS en fonction du nombre d'objets n'est pas linéaire : cela est probablement dû au fonctionnement interne d'OpenGL.

Les résultats de la comparaison entre usage du polymorphisme et usage du pattern Data Locality sont surprenants : en effet, l'application de ce design pattern a ralenti les performances de l'application, en tout cas dans la partie génération, le rendu graphique restant inchangé avec des FPS constants, au lieu de les améliorer.

Cela s'explique par le fait que ce pattern améliore les performances d'applications ralenties par les «cache misses». Lors de l'analyse de l'application par un outil d'analyse de cache, Valgrind, il est apparu que seulement 0 à 2% des accès au cache créaient des «cache misses». Le problème de performances n'était donc pas dû à cela.

Pourquoi alors les performances ne sont-elles pas restées constantes ? En fait, au lieu de stocker dans notre structure de données un pointeur sur objet, qui prend typiquement 4 octets en mémoire, on stocke l'ensemble de l'objets, c'est à dire dans mon cas 136 octets. En C++, le passage d'un argument à une fonction se fait par défaut par valeur, c'est à dire qu'il y a une copie de l'objet. On copie donc 136 octets (au lieu de 4 auparavant) à chaque fois que l'on ajoute un objet dans la structure de données, et ce autant de fois qu'il y a d'objets. C'est ce qui ralentit la génération.

L'application du pattern Data Locality s'est donc révélé infructueux dans mon cas. Ce pattern reste cependant valide dans le cas d'applications souffrant de problèmes de «caches misses» et de fragmentation.

3.6.2 Smart pointers («Pointeurs intelligents»)

> You can either have software quality or you can have pointer arithmetic, but you cannot have both at the same time
>
> Bertrand Meyer

Les smart pointers sont des objets de la librairies standard encapsulant les pointeurs. En se basant sur la RAII (3.5.2, paragraphe «Design Pattern»), ils permettent de s'assurer que les ressources mémoires allouées manuellement seront bien libérées. De plus ils fournissent des fonctionnalités bien pratiques, comme le compte de pointeurs pointant sur la variable encapsulée, sorte de compte de références, pratique pour implémenter un groupe de ressources partagées : lorsque plus personne dans le programme n'utilise ces ressources, on les supprime, ou plutôt elles sont supprimées automatiquement.

C'est ce qui s'approche en C++ d'un garbage collector, à la différence que la norme du langage assure à quel endroit du code le destructeur de l'objet sera appelé et que le développeur doit mettre en place manuellement cette gestion «automatisée», ou plutôt «aidée», de la mémoire.

L'utilisation de ces pointeurs intelligents est considérée dans le monde du développement C++ comme une bonne pratique et est encouragée. De plus il n'y a peu ou pas de baisse de performances par rapport à l'utilisation de pointeurs nus («raw pointers», pointeurs traditionnels «à la C»).

Cette optimisation ne vise donc pas les performances mais la maintenabilité, la sécurité et la stabilité du programme. Au final, pour générer 1024 objets, la différence entre pointeurs nus et pointeurs intelligents est de 0.06 s, et il n'y a aucune fuite mémoire. De plus le code est plus court et plus simple.

3.6.3 Logs

> There are two ways to write error-free programs ; only the third one works
>
> Alan J. Perlis

Les logs sont l'affichage de messages de contrôle au cours du programme. Ils permettent de vérifier son exécution et de détecter des bugs. Après avoir utilisé des logs rudimentaires, je me suis intéressé aux bibliothèques de logs existantes, notamment Boost.Log .

Cependant, ces bibliothèques se sont révelées bien trop riches et complexes pour la taille de ce programme. J'ai donc développé une classe simple de logs, qui offrent les fonctionnalités simples suivantes :

— Quatre niveaux de logs différents (trace, debug, info, error),
— Logs dans la console (activable/désactivable),
— Logs dans des fichiers «.log» et «.err» selon le niveau du log (activable/désactivable),
— Filtre des logs selon le niveau,
— Syntaxe simple

Cette amélioration n'est pas directement liée aux performances, mais vise à améliorer encore une fois la maintenabilité et la stabilité du code. Cependant, les performances de l'application bénéficient du fait que les logs soient désactivables en partie ou complètement, au choix.

3.6.4 Problèmes restants - Fonctionnalités à améliorer

Skybot 3D

Un problème handicapant restant dans Skybot 3D est la vue Oculus. En effet cette dernière est fonctionnelle, mais souffre du phénomène de «cross-eye» («yeux croisés»). Cet effet est dû à une inversion du rendu entre les deux yeux : le rendu de l'oeil gauche se retrouve sur la moitié droite sur l'écran et vice-versa. En conséquence, lorsque l'Oculus est équipé, les deux images provenant de chaque oeil ne se superposent pas parfaitement pour le cerveau comme cela devrait être le cas : une impression de strabisme (fait de loucher) se produit.

Malgré de nombreux efforts et de multiples échanges avec l'équipe Skybot 3D, aucune solution n'a émergé de cette communication. Ce phénomène est donc toujours en place et est probablement dû à une application erronnée des matrices de tranformation.

Simulation

Une fonctionnalité que je n'ai pas eu le temps d'implémenter pour la simulation est l'utilisation du joystick. En effet la bibliothèque de fenêtrage utilisée, SDL, est capable de détecter un joystick, et ce dernier facilité l'immersion et le déplacement dans la scène avec l'Oculus Rift masquant le clavier.

4 Remerciements

Plusieurs personnes m'ont apporté une aide significative sur ce projet et je tiens à les remercier chaleureusement ici :

— André SCHAAFF, mon maître de stage,

— L'équipe Skybot 3D : Jérôme Berthier et Jonathan Normand,
— Brad Davis, auteur du livre «Oculus Rift in Action», pour l'aide qu'il m'a apporté sur les forums d'Oculus Rift,
— Nicolas Deparis et Romain Houpin, stagiaires à l'Observatoire sur des sujets de rendu graphique 3D,
— Sébastien Derriere, astronome adjoint au CDS,
— Pierre Ocvirk, astronome adjoint au CDS, qui nous a fourni les données de simulation et nous a apporté son expertise dans ce domaine

5 Conclusion

Ce stage de 10 semaines m'a apporté énormément. L'intérêt du stage était à la hauteur des défis rencontrés. J'ai eu la chance d'expérimenter le travail collaboratif en participant à un gros projet existant, mais aussi de vivre un projet individuel avec une grande liberté, avec un sujet unique en son genre. J'ai également pu approfondir mes connaissances sur des sujets génériques et réutilisables tels que le génie logiciel, l'organisation d'un projet et la collaboration. Je suis reconnaissant à l'Observatoire et à mon maître de stage de m'avoir fait confiance et donné cette chance.

Au final, je finis ce stage satisfait de ce qui a été accompli et intéressé dans le future par les domaines abordés, tel que l'imagerie et le rendu 3D.

6 Annexes

6.1 Design Pattern détaillés

6.1.1 Data Locality

Tous les CPU modernes disposent d'une certaine quantité de mémoire vive, la RAM, qui représentent plusieurs Gigaoctets. Pour accélérer l'accès à cette mémoire, le CPU dispose d'un cache (plusieurs en réalité) de taille réduite (sur ma machine je dispose d'un Intel Core i5 cadencé à 3.2 GHz avec 4 coeurs et 6144 kB de cache), qui est en fait une zone mémoire à accès très rapide où sont stockées les informations dont l'accès est fréquent dans la RAM. On accélère ainsi les opérations les plus régulièrement effectuées : c'est la mise en cache («caching»).

Cependant, le polymorphisme impose en C++ l'usage de pointeurs (cas le plus fréquent) ou de références (cas particuliers. Mais les pointeurs, par définition, pointent vers différentes zones mémoires. Ainsi, lorsque l'on veut afficher tous les objets de la scène, on parcoure la structure de données qui stocke les pointeurs sur tous ces objets et on les affiche un par un.

Le problème est que contrairement au parcours d'un tableau de valeurs (non pointeurs), qui sont stockées contigument dans la mémoire, et sont donc particulièrement adaptées à la mise en cache, le fait de parcourir un tableau de pointeurs va silloner toute la RAM, et donc ne va pas du tout se prêter à la mise en cache. Le CPU va soit ne pas mettre ces variables (et les zones mémoires voisines) dans le cache, soit le faire et en conséquence changer la majeure partie du cache à chaque fois, ce qui n'est pas du tout optimal. On passe donc à côté d'un beau gain de performance : c'est le «cache miss».

Plus généralement, cette situation se produit dès lors qu'il y a un usage fréquent de pointeurs dans un programme, et surtout le déréférencement de ces derniers.

Un autre point à considérer est la différence entre l'utilisation , dans une structure de donnée, du polymorphisme (plusieurs objets différents coexistent, héritant d'un objet commun) et l'utilisation d'un seul type d'objets. Dans le second cas, une méthode appelé par un objet est déterminée à la compilation (on sait quel est le type de l'objet puisqu'il n'y en a qu'un) tandis que dans le premier cas, elle est déterminée à l'exécution, ce qui peut engendrer une baisse de performances supplémentaires.

Pour résumer, le polymorphisme apporte généricité et flexibilité, mais peut engendrer une baisse de performances, alors que l'usage d'un seul type d'objets est plus rigide, mais peut améliorer les performances.

Un dernier point à envisager est la différence de stockage des variables entre l'utilisation de pointeurs sur variables et l'utilisation de variables. Dans le premier cas, les variables sont allouées (et libérées) manuellement et sont placées sur le tas («heap»), alors que dans le second cas elles sont allouées automatiquement sur la pile («stack»). Ces deux zones mémoires sont présentes dans chaque programme et sont réservées par le système d'exploitation pour le programme à son lancement et sont fort différentes :

Stack :
— Accès rapide
— Limitée en taille
— L'espace est géré automatiquement et efficacement par le CPU (pas de fragmentation)

Heap :
— Accès plus lent

— Pas de limite de taille
— Pas de gestion automatique de l'espace (fragmentation)

Ces allocations/désallocations ne sont pas non plus gratuites en termes de temps d'exécution, puisque elles font appel à un certain nombre de fonctions.

6.2 Outils utilisés

6.2.1 Langages utilisés

Les deux projets sur lesquels j'ai travaillé ont été développés en C++, même si la quasi-totalité du projet Skybot 3D est écrite en C, adapté pour pouvoir compiler en C++. Sur le deuxième projet, j'ai de plus mis en oeuvre des fonctionnalités nouvelles de C++, standardisées par la norme C++11 datant de 2011. On peut citer :

— Listes d'initialisation
— Pointeur null «nullptr»
— Boucle for sur une plage de valeurs
— Référence sur une rvalue
— Outils de mesure du temps à haute précision
— Mot-clé «auto»
— Smart pointers (pointeur intelligents)

6.2.2 Bibliothèques utilisées

GLEW
«OpenGL Extension Wrangler Library» est une bibliothèque multiplateforme de chargement des fonctionnalités d'OpenGL. Licence BSD modifiée.
http://glew.sourceforge.net/

GLM
«OpenGL Mathematics» est une bibliothèque de fonctions mathématiques pour OpenGL. Je l'ai utilisée pour le projet de simulation. Licence MIT.
http://glm.g-truc.net/0.9.5/index.html

GLUT
«OpenGL Utility Toolkit» est l'équivalent de la SDL en plus restreint. Je m'en suis servi pour le projet Skybot 3D via son implémentation open source «freeglut». Licence X-Consortium.
http://freeglut.sourceforge.net/ et http://www.opengl.org/ressources/libraries/glut/

Octree
Une bibliothèque C++ implémentant les Octree. Licence GPL.
http://nomis80.org/code/octree.html

Oculus SDK
Le SDK Oculus est l'interface de programmation permettant d'accéder à l'Oculus Rift et est fourni par Oculus VR (fabriquant de l'Oculus Rift). Je me suis servi des versions 0.2.5 et 0.3.2. Licence particulière mais analogue à une licence MIT, à ceci près qu'elle restreint les utilisations pouvant mettre en danger la vie ou la santé d'individus.

OpenGL
«Open Graphics Library» est une API multiplateforme pour effectuer des rendus graphiques 2D et 3D. L'implémentation est laissée à la charge des constructeurs de cartes graphiques et est fournie par les drivers. Pour ma part j'ai travaillé avec la carte graphique AMD Radeon HD 8570 disposant d'1Gb de RAM dédiée et le driver ATI Fire GL datant de mai 2014, implémentant OpenGL 4.4 (dernière version d'OpenGL). Je l'ai utilisée pour les deux projets. Pas de licence.
http://www.opengl.org/

SDL
«Simple DirectMedia Layer» est une bibliothèque multiplateforme donnant accès au système de fenêtrage, au clavier, à la souris et à un éventuel joystick. Je l'ai utilisée pour le projet de simulation. Licence zlib.
https://www.libsdl.org/

6.2.3 Outils divers utilisés

Clang
Compilateur/interface de compilation C/C++ open source développé par Google. Licence de l'Université de l'Illinois / licence NCSA open source.
http://clang.llvm.org/

Clang Static Analyzer
Outil d'analyse statique (à la compilation) de code et qui fait partie du projet Clang.
http://clang-analyzer.llvm.org/scan-build.html

CMake
Outil multiplateforme 'aide à la compilation et de génération de Makefiles. Licence BSD.
http://www.cmake.org/

Doxygen
Outil de documentation de projets informatiques et de code sources. Licence GPL
http://www.stack.nl/~dimitri/doxygen/

GCC
«GNU Compiler Collection», compilateur C/C++. Licence GNU GPL 3+.
https://gcc.gnu.org/ item [GDB]
«GNU Project Debugger», le déboggeur standard sur Linux. Licence GNU GPL.
http://www.gnu.org/software/gdb/

Git
Logiciel de gestion de version. Licence GNU GPL v2.
http://git-scm.com/

Github
Site web de stockage en ligne de projets open source via git.
https://github.com/

Qmake
Outil d'aide à la compilation multi-plateforme, semblable à Cmake. Licence LGPL.
http://qt-project.org/doc/qt-4.8/qmake-manual.html

QtCreator
IDE C++ open source avec débuggeur et outils d'analyses intégrés. Licence LGPL.
http://qt-project.org/wiki/category:tools::qtcreator

Valgrind
Outil d'analyse dynamique (à l'exécution) de programme, pour notamment traquer les fuites de mémoire. Licence GPL v2.
http://valgrind.org/

Tous ces outils sont open source et multiplateformes.

6.3 Captures d'écran

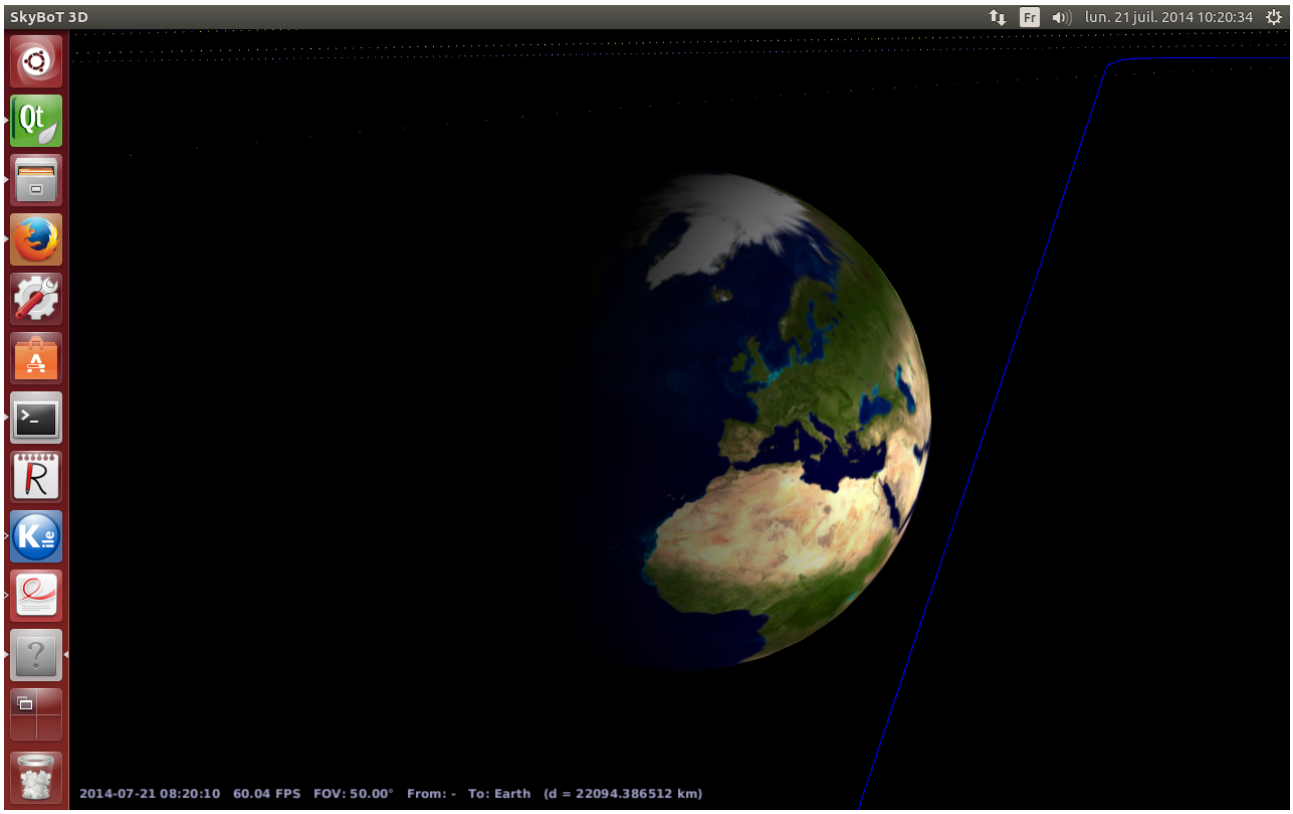

FIGURE 5 – Visualisation de la planète Terre dans Skybot 3D en vue normale

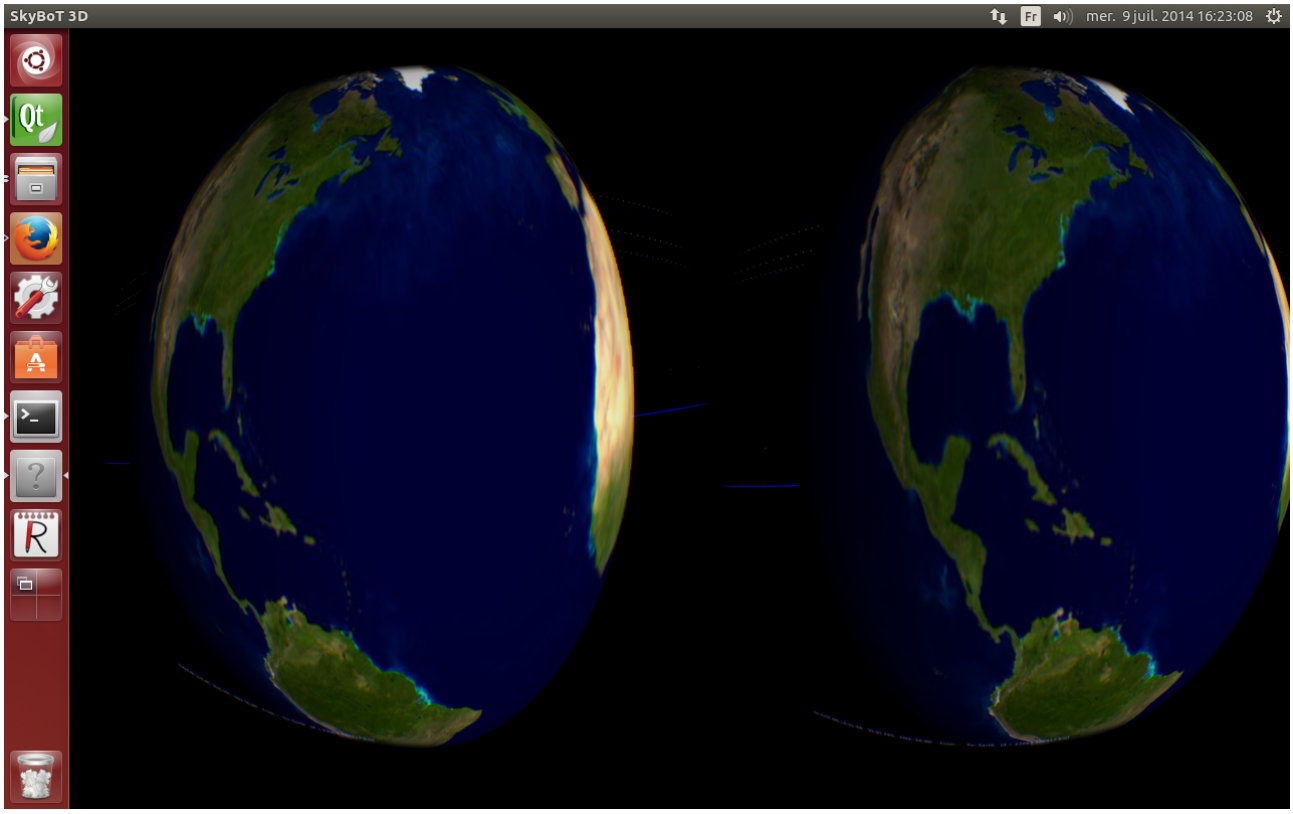

FIGURE 6 – Visualisation de la planète Terre dans Skybot 3D en vue Oculus

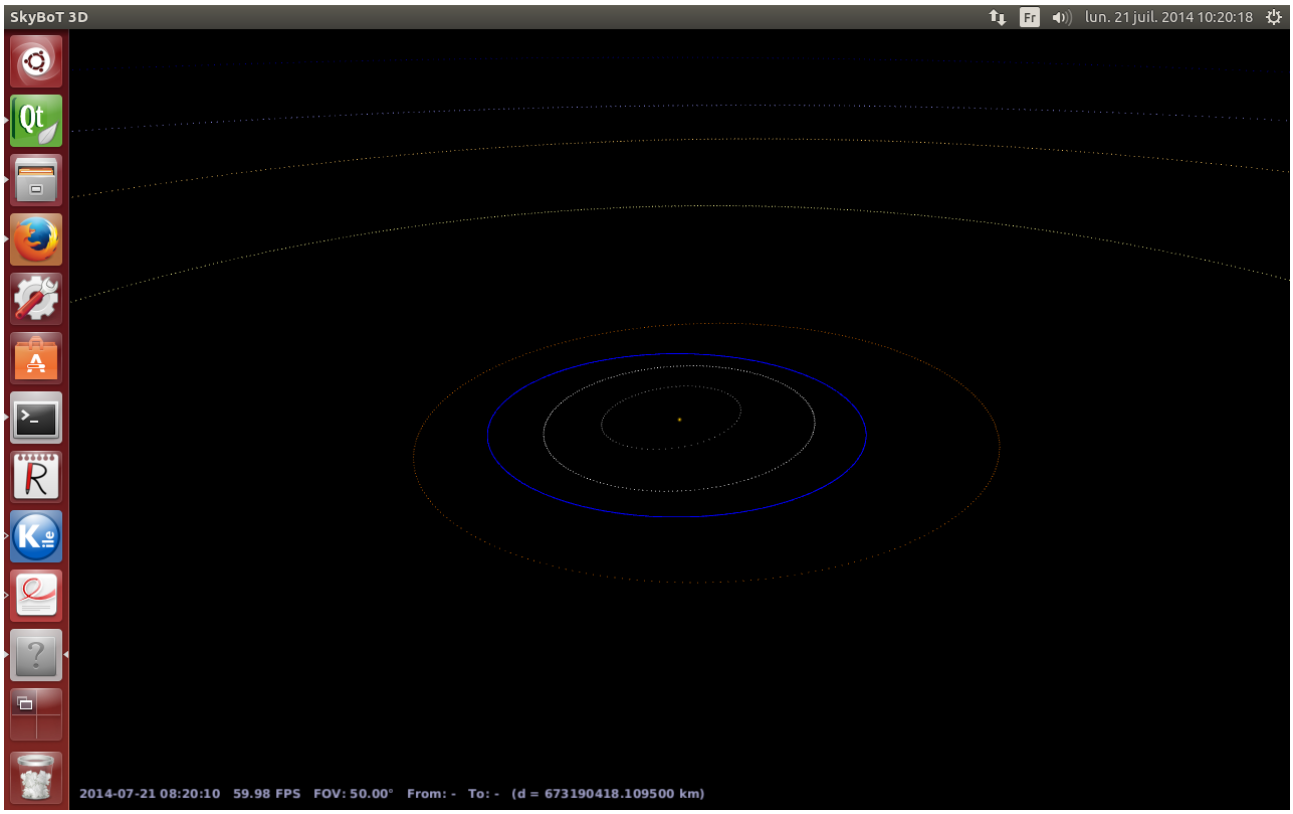

FIGURE 7 – Visualisation du système solaire dans Skybot 3D en vue normale

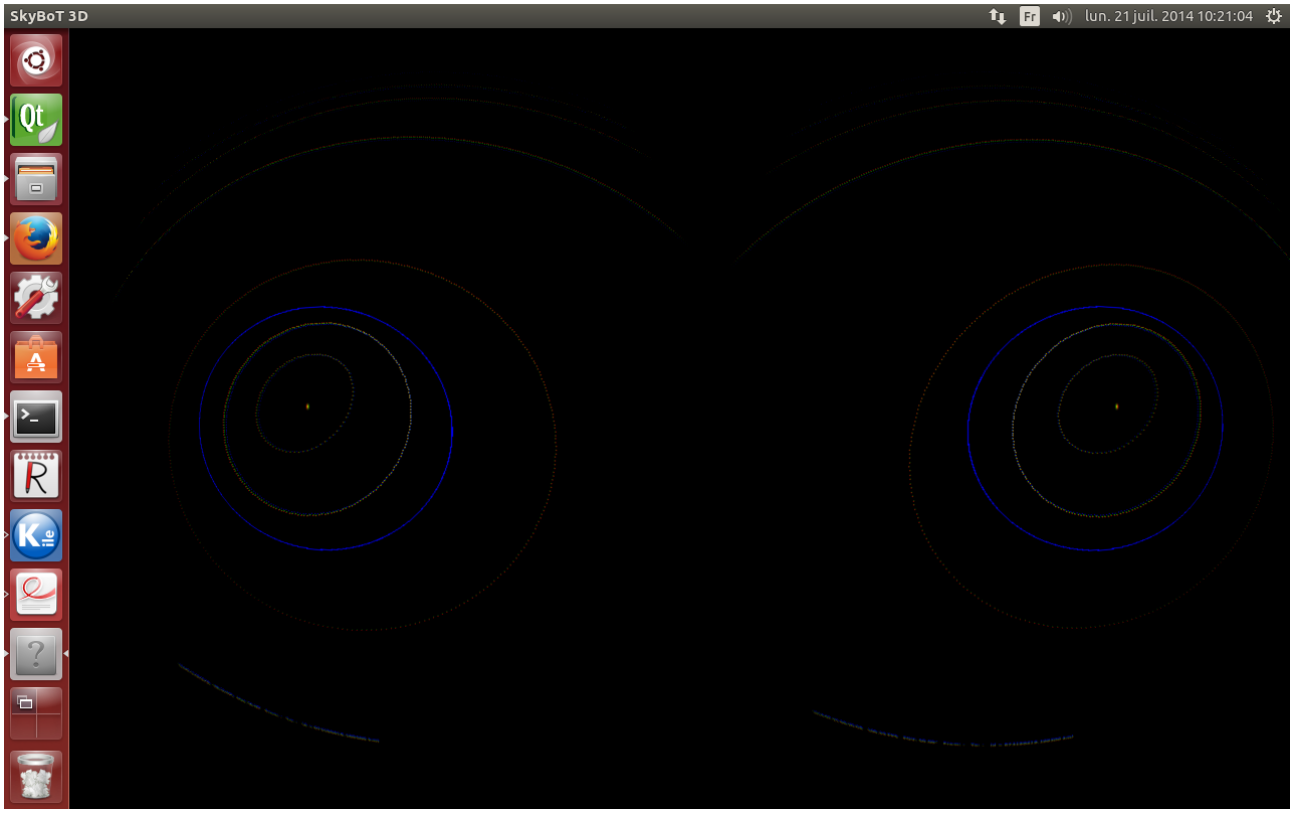

FIGURE 8 – Visualisation du système solaire dans Skybot 3D en vue Oculus

Les images suivantes (figures 9 et 10) présentent l'application de simulation d'un cube de données d'objets célestes, organisés en un Octree de taille 128*128*128, divisé en octants de taille 8.

Les objets céleste sont modélisés par des cubes texturés de taille 1.

Seul l'octant actuel et les octants directement voisins sont visibles pour des raisons de performances.

La caméra est dirigeable par les mouvements de la tête en mode Oculus.

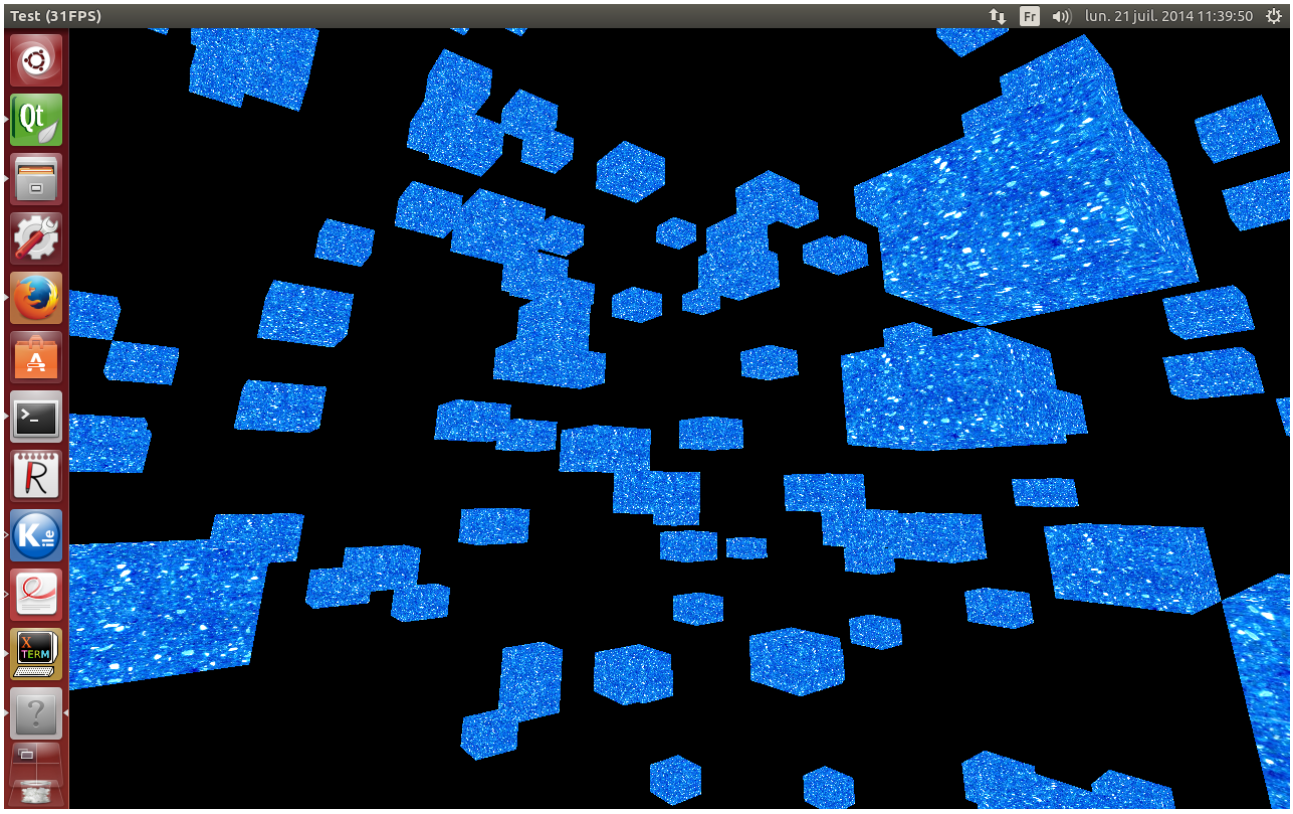

FIGURE 9 – Simulation en vue normale

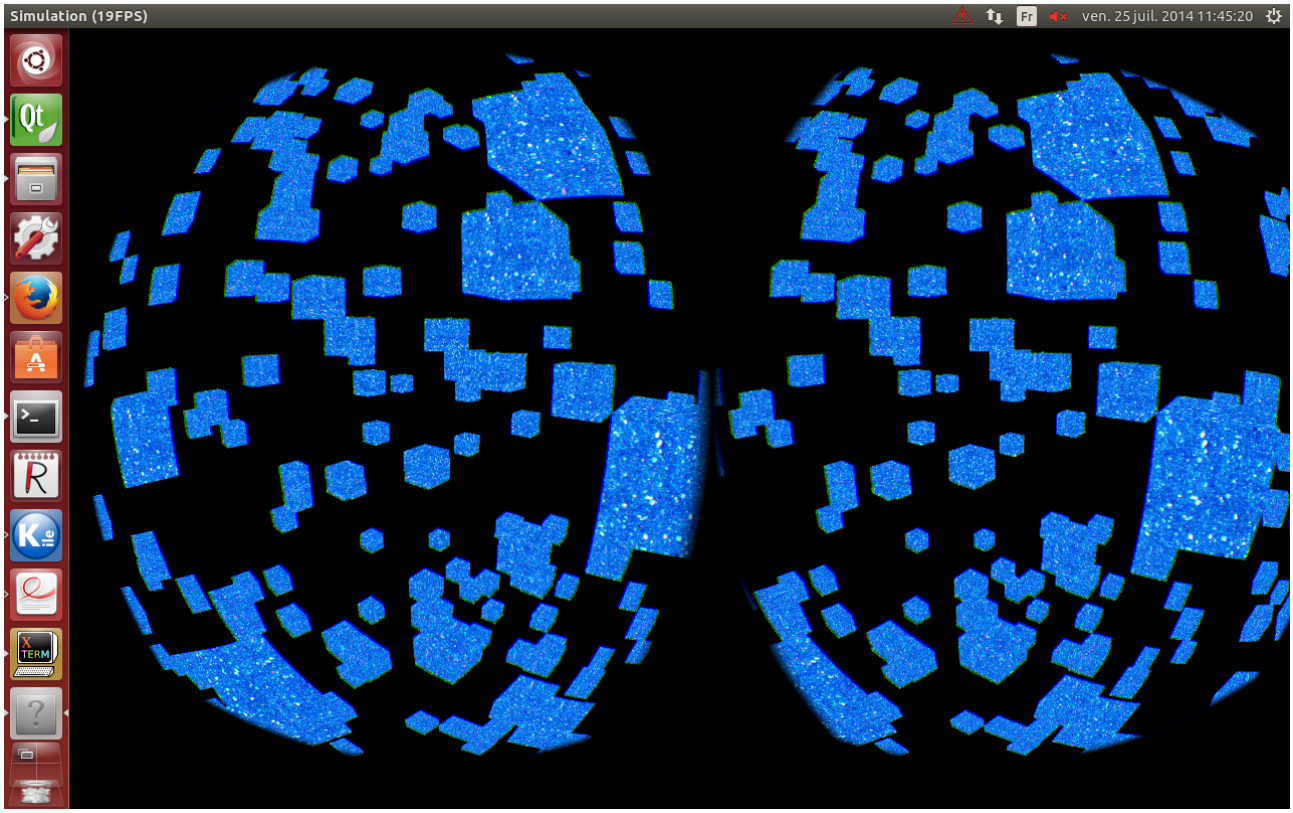

FIGURE 10 – Simulation en vue normale

Les images suivantes (figures 11 et 12) présentent l'application de simulation d'un cube de données d'objets célestes, organisés en un tableau simple.

Les objets céleste sont modélisés par des cubes texturés de taille 1.

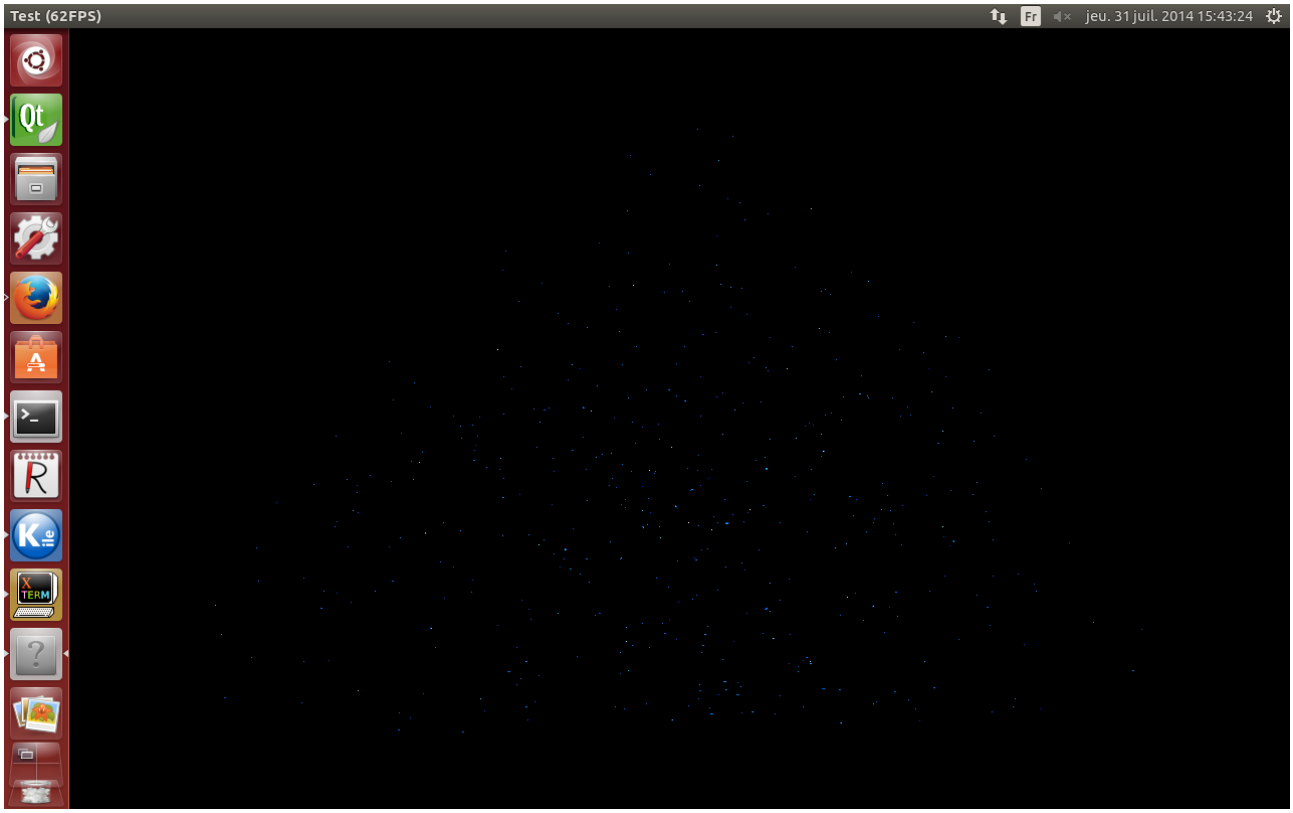

FIGURE 11 – Simulation en vue normale

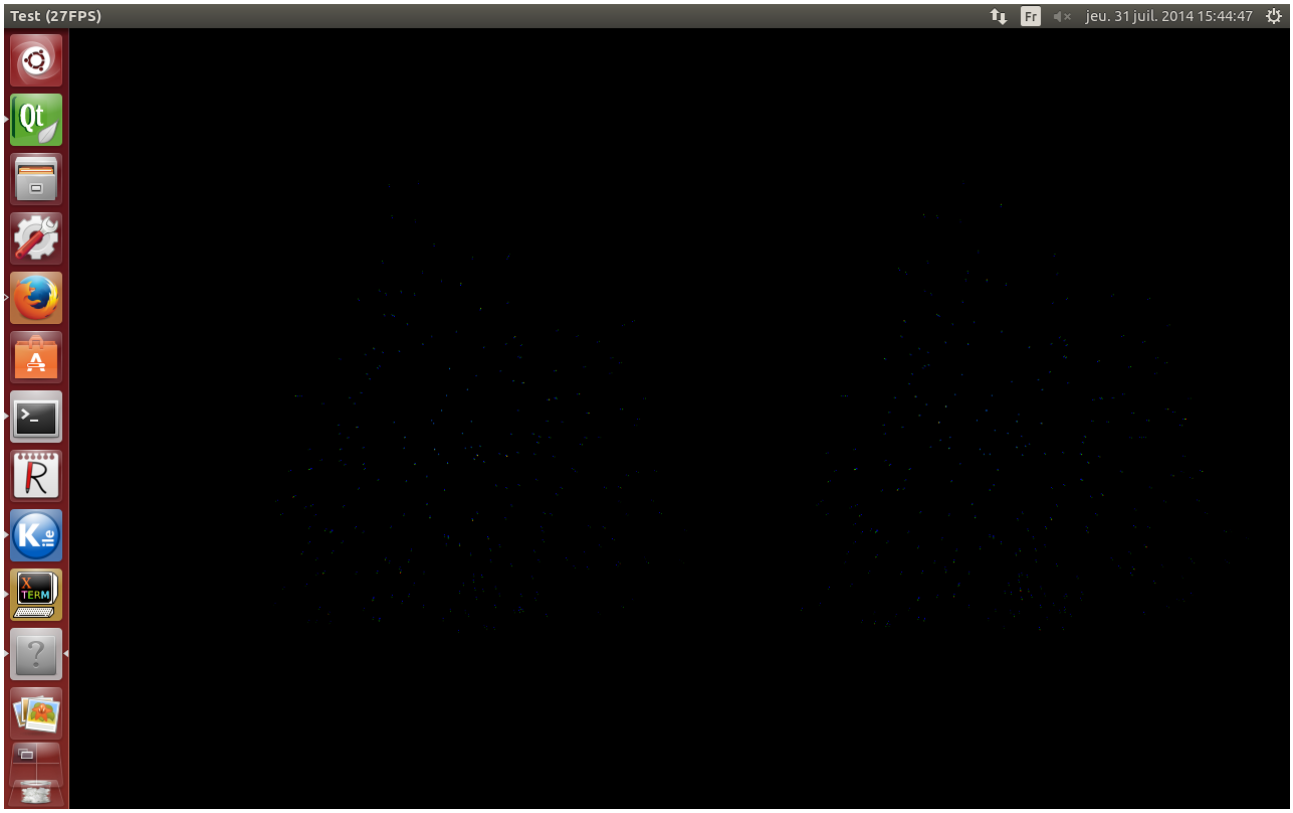

FIGURE 12 – Simulation en vue normale

6.4 Code source

Talk is cheap. Show me the code

Linus Torvalds

«Oculus.h» est un fichier d'en-tête décrivant la classe Oculus. Il prend comme argument de template la scene OpenGL générique que l'on veut afficher en mode Oculus. La seule contrainte est que cette scène fournisse la méthode «render».

```
#ifndef OCULUS_H
#define OCULUS_H

/** @file
 * @brief All Oculus related features live in here
 * @author Philippe Gaultier
 * @version 1.0
 * @date 24/07/14
 */

#include <GL/glew.h>

#include "Include/OVR/LibOVR/Include/OVR.h"
#include "Include/OVR/LibOVR/Src/OVR_CAPI.h"
#include "Include/OVR/LibOVR/Src/OVR_CAPI_GL.h"
#include "Include/OVR/LibOVR/Src/Kernel/OVR_Math.h"
#include "SDL2/SDL.h"
#define GL3_PROTOTYPES 1
#include "Include/GL3/gl3.h"
#include "Include/glm/glm.hpp"
#include "SDL2/SDL_syswm.h"
#include "Utils.h"
#include "Log.h"

#include <iostream>
//To ignore the asserts uncomment this line:
//#define NDEBUG
#include <cassert>
#include <cmath>
#include <limits>
#include <string>
#include <memory>

/**
 * @brief The GenericOculus class
 */
class GenericOculus
{
public:
    virtual ~GenericOculus() {}
    virtual void render() = 0;

    virtual void getInput() {}

    virtual bool isMoving();

    virtual bool isUsingDebugHmd();
```

```
    glm::vec3 dAngles() const;
};


template<class T>
/**
 * @brief The Oculus templated class
 * @details It is a singleton to avoid initializing/releasing the Oculus SDK
    multiple times.
 * The template argument is the type of the OpeGL scene we render.
 */
class Oculus: public GenericOculus
{
public:
    /**
     * @brief Constructor
     * @details Initializes the Oculus SDK, creates a debug Oculus Rift if
   none is connected, and starts the sensors.
     * @param scene The OpenGL scene that contains the objects render
     */
    Oculus(T & scene):
        scene_ {scene},
        textureId_ {0},
        FBOId_ {0},
        depthBufferId_ {0},
        hmd_ {0},
        windowSize_ {0, 0},
        textureSizeLeft_ {0, 0},
        textureSizeRight_ {0, 0},
        textureSize_ {0, 0},
        angles_ {0, 0, 0},
        dAngles_ {0, 0, 0},
        distortionCaps_ {0},
        usingDebugHmd_ {false},
        multisampleEnabled_ {false}
    {
        //Oculus is a singleton and cannot be instanciated twice
        assert(!alreadyCreated);
        logger->debug(logger->get() << "Oculus constructor" );

        ovr_Initialize();

        hmd_ = ovrHmd_Create(0);

        if(!hmd_)
        {
            hmd_ = ovrHmd_CreateDebug(ovrHmd_DK1);
            usingDebugHmd_ = true;

            //Cannot create the debug hmd
            assert(hmd_);

            logger->debug(logger->get() << "Using the debug hmd");
        }

        ovrHmd_GetDesc(hmd_, &hmdDesc_);
```

```
        computeSizes();

        distortionCaps_ = ovrDistortionCap_Chromatic |
ovrDistortionCap_TimeWarp;

        eyeFov_[0] = hmdDesc_.DefaultEyeFov[0];
        eyeFov_[1] = hmdDesc_.DefaultEyeFov[1];

        setOpenGLState();
        initFBO();
        initTexture();
        initDepthBuffer();

        computeSizes();
        setCfg();
        setEyeTexture();
        ovrBool configurationRes = ovrHmd_ConfigureRendering(hmd_, &cfg_.
Config, distortionCaps_, eyeFov_, eyeRenderDesc_);
        //Cannot configure OVR rendering
        assert(configurationRes);

        ovrHmd_StartSensor(hmd_, ovrSensorCap_Orientation |
ovrSensorCap_YawCorrection | ovrSensorCap_Position,
ovrSensorCap_Orientation);

        Oculus::alreadyCreated = true;
    }

    /**
     * @brief Destructor
     * @details Releases the Oculus SDK and the OpenGL resources required
for the Oculus rendering
     */
    ~Oculus()
    {
        logger->debug(logger->get() << "Oculus destructor");
        glDeleteFramebuffers(1, &FBOId_);
        glDeleteTextures(1, &textureId_);
        glDeleteRenderbuffers(1, &depthBufferId_);

        ovrHmd_Destroy(hmd_);

        ovr_Shutdown();

        Oculus::alreadyCreated = false;
    }

    /**
     * @brief Renders the OpenGL scene with the Oculus effects
     */
    void render()
    {
        glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0);
        glBindBuffer(GL_ARRAY_BUFFER, 0);
        glUseProgram(0);
```

```
        frameTiming_ = ovrHmd_BeginFrame(hmd_, 0);

        // Bind the FBO...
        glBindFramebuffer(GL_FRAMEBUFFER, FBOId_);
        // Clear...
        glClearColor(0, 0, 0, 1);
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

        getInput();

        ovrPosef eyeRenderPose[2];

        for (int eyeIndex = 0; eyeIndex < ovrEye_Count; eyeIndex++)
        {
            ovrEyeType eye = hmdDesc_.EyeRenderOrder[eyeIndex];
            eyeRenderPose[eye] = ovrHmd_BeginEyeRender(hmd_, eye);

            glViewport(eyeTexture_[eye].OGL.Header.RenderViewport.Pos.x,
                       eyeTexture_[eye].OGL.Header.RenderViewport.Pos.y,
                       eyeTexture_[eye].OGL.Header.RenderViewport.Size.w,
                       eyeTexture_[eye].OGL.Header.RenderViewport.Size.h
                       );

            // Get Projection and ModelView matrici from the device...
            OVR::Matrix4f MV = OVR::Matrix4f::Translation(eyeRenderDesc_[eye
].ViewAdjust)
                    * OVR::Matrix4f(OVR::Quatf(eyeRenderPose[eye].
Orientation).Inverted());

            OVR::Matrix4f Proj = OVR::Matrix4f(ovrMatrix4f_Projection(
eyeRenderDesc_[eye].Fov, 0.01f, 10000.0f, true));

            glm::mat4 glmMV = Utils::ovr2glmMat(MV.Transposed());

            glm::mat4 glmProj = Utils::ovr2glmMat(Proj.Transposed());

            scene_.render(glmMV, glmProj);
            Utils::GLGetError();

            ovrHmd_EndEyeRender(hmd_, eye, eyeRenderPose[eye], &eyeTexture_[
eye].Texture);
        }

        ovrHmd_EndFrame(hmd_);
    }

    /**
     * @brief Tells if we are using a debug Oculus Rift
     * @return true if no Oculus Rift is connected and we had to create a
debug one, else false
     */
    bool isUsingDebugHmd()
    {
        return usingDebugHmd_;
    }

    /**
```

```
 * @brief Tells if the Oculus Rift the moving
 * @details It compares the current angular position with the previous
angular position
 * @return true if the Oculus Rift if moving, else false
 */
using GenericOculus::isMoving;
bool isMoving() const
{
    bool res = false;
    for(int i=0; i < 3; i++)
    {
        res = res && Utils::isEqual(angles_[i], dAngles_[i]);
    }
    return !res;
}

glm::vec3 angles() const
{
    return angles_;
}

void setAngles(const glm::vec3 &angles)
{
    angles_ = angles;
}

/**
 * @brief Retrieves the values from the Oculus Rift sensors
 * @details It gets the current angular position from the sensors and
the prediction tool, and stores the old angular position.
 * @warning The angles from the sensors are in radians and OpenGL
expects angles in degrees, hence the required conversion
 * @warning If no Oculus Rift is connected and we had to create a debug
one, there are no values to be retrieved: We use the mouse position.
 */
void getInput()
{
    glm::vec3 oldAngles = angles_;

    sensorState_ = ovrHmd_GetSensorState(hmd_, frameTiming_.
ScanoutMidpointSeconds);

    if(sensorState_.StatusFlags & (ovrStatus_OrientationTracked |
ovrStatus_PositionTracked))
    {
        ovrPosef pose = sensorState_.Predicted.Pose;
        OVR::Quatf quat = pose.Orientation;

        quat.GetEulerAngles<OVR::Axis_Y, OVR::Axis_X, OVR::Axis_Z>(&
angles_.x, &angles_.y, &angles_.z);

        dAngles_ = angles_ - oldAngles;

        logger->debug(logger->get() << "Angles: "
                    << OVR::RadToDegree(angles_[0]) << ", "
                    << OVR::RadToDegree(angles_[1]) << ", "
                    << OVR::RadToDegree(angles_[1]) << " degrees");
```

```
            logger->debug(logger->get() << "Angles: "
                        << angles_[0] << ", "
                        << angles_[1] << ", "
                        << angles_[1] << " rad");

            logger->debug(logger->get() << "DAngles: "
                        << OVR::RadToDegree(dAngles_[0]) << ", "
                        << OVR::RadToDegree(dAngles_[1]) << ", "
                        << OVR::RadToDegree(dAngles_[1]) << " degrees");
        }
        else
        {
            logger->debug(logger->get() << "No input data (using debug hmd)"
    );
        }
    }

protected:
    /**
     * @brief Creates the OpenGL texture required for the Oculus rendering
     * @details The Oculus rendering makes under the hood a double (for each
    eye) render to texture of the scene and then
     * displays this texture to the screen, hence the big size of the
    texture.
     */
    void initTexture()
    {
        // The texture we're going to render to...
        glGenTextures(1, &textureId_);
        // "Bind" the newly created texture : all future texture functions
    will modify this texture...
        glBindTexture(GL_TEXTURE_2D, textureId_);
        // Give an empty image to OpenGL (the last "0")
        glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, textureSize_.w, textureSize_
    .h, 0, GL_RGBA, GL_UNSIGNED_BYTE, 0);
        // Linear filtering...
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);

        Utils::GLGetError();
    }

    /**
     * @brief Creates the Frame Buffer Object needed for the Oculus
    rendering
     * @details The Oculus rendering uses this FBO to send the texture to
    the graphic card
     */
    void initFBO()
    {
        // We will do some offscreen rendering, setup FBO...
        assert(textureSize_.w != 0);
        assert(textureSize_.h != 0);

        glGenFramebuffers(1, &FBOId_);
        Utils::GLGetError();
```

```
        //Cannot create the FBO
        assert(FBOId_ != 0);

        glBindFramebuffer(GL_FRAMEBUFFER, FBOId_);
        Utils::GLGetError();
    }

    /**
     * @brief Creates the depth buffer needed for the Oculus rendering
     */
    void initDepthBuffer()
    {
        glGenRenderbuffers(1, &depthBufferId_);
        Utils::GLGetError();
        //Cannot create the depth buffer
        assert(depthBufferId_ != 0);

        glBindRenderbuffer(GL_RENDERBUFFER, depthBufferId_);
        Utils::GLGetError();

        glRenderbufferStorage(GL_RENDERBUFFER, GL_DEPTH_COMPONENT,
textureSize_.w, textureSize_.h);
        Utils::GLGetError();

        glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT,
GL_RENDERBUFFER, depthBufferId_);
        Utils::GLGetError();

        // Set the texture as our colour attachment #0...
        glFramebufferTexture(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,
textureId_, 0);
        Utils::GLGetError();

        // Set the list of draw buffers...
        GLenum GLDrawBuffers[1] = { GL_COLOR_ATTACHMENT0 };

        glDrawBuffers(1, GLDrawBuffers); // "1" is the size of DrawBuffers
        Utils::GLGetError();

        // Check if everything is OK...
        GLenum check = glCheckFramebufferStatus(GL_DRAW_FRAMEBUFFER);

        //There is a problem with the FBO
        assert(check == GL_FRAMEBUFFER_COMPLETE);

        // Unbind...
        glBindRenderbuffer(GL_RENDERBUFFER, 0);
        glBindTexture(GL_TEXTURE_2D, 0);
        glBindFramebuffer(GL_FRAMEBUFFER, 0);
        Utils::GLGetError();

    }

    /**
     * @brief Sets some OpenGL states to adequate values for the Oculus
rendering
     * @warning The multisample value does not seem the be taken into
```

```
account by the Oculus SDK as of yet
  *  and the Oculus rendering seems unchanged
  */
 void setOpenGLState()
 {
     glDisable(GL_TEXTURE_2D);
     glEnable(GL_DEPTH_TEST);
     if(multisampleEnabled_)
     {
         glEnable(GL_MULTISAMPLE);
     }
 }

 /**
  * @brief Sets the Oculus SDK configuration to adequate values for the
Oculus rendering
  * @warning The Windows and OSX modes have not been tested but should
work just fine
  */
 void setCfg()
 {
     cfg_.OGL.Header.API = ovrRenderAPI_OpenGL;
     cfg_.OGL.Header.Multisample = multisampleEnabled_;
     cfg_.OGL.Header.RTSize.w = windowSize_.w;
     cfg_.OGL.Header.RTSize.h = windowSize_.h;

     SDL_SysWMinfo info;
     SDL_VERSION(&info.version);
     SDL_bool infoRes = SDL_GetWindowWMInfo(scene_.window(), &info);
     //Cannot retrieve SDL window info
     assert(infoRes == SDL_TRUE);

     #if defined(OVR_OS_WIN32)
         cfg.OGL.Window = info.info.win.window;
     #elif defined (OVR_OS_MAC)
         cfg.OGL.Window = info.info.cocoa.window
     #elif defined(OVR_OS_LINUX)
         cfg_.OGL.Win = info.info.x11.window;
         cfg_.OGL.Disp = info.info.x11.display;
     #endif
 }

 /**
  * @brief Sets the Oculus SDK texture configuration to adequate values
for the Oculus rendering
  */
 void setEyeTexture()
 {
     eyeTexture_[0].OGL.Header.API = ovrRenderAPI_OpenGL;
     eyeTexture_[0].OGL.Header.TextureSize.w = textureSize_.w;
     eyeTexture_[0].OGL.Header.TextureSize.h = textureSize_.h;
     eyeTexture_[0].OGL.Header.RenderViewport.Pos.x = 0;
     eyeTexture_[0].OGL.Header.RenderViewport.Pos.y = 0;
     eyeTexture_[0].OGL.Header.RenderViewport.Size.h = textureSize_.h;
     eyeTexture_[0].OGL.Header.RenderViewport.Size.w = textureSize_.w/2;
     eyeTexture_[0].OGL.TexId = textureId_;
```

```
        // Right eye the same, except for the x-position in the texture...
        eyeTexture_[1] = eyeTexture_[0];
        eyeTexture_[1].OGL.Header.RenderViewport.Pos.x = (textureSize_.w +
1) / 2;

    }

    /**
     * @brief Computes the texture size
     * @details This computation depends on the window dimensions. The
optimal dimensions are 1280*800, which is the Oculus
     * resolution
     * @warning Other resolutions and window resizing have not been tested
but should work just fine
     */
    void computeSizes()
    {
        windowSize_.w = scene_.windowWidth();
        windowSize_.h = scene_.windowHeight();

        logger->debug(logger->get() << "Fov: " << Utils::radToDegree(2 *
atan(hmdDesc_.DefaultEyeFov[0].UpTan)));

        textureSizeLeft_ = ovrHmd_GetFovTextureSize(hmd_, ovrEye_Left,
hmdDesc_.DefaultEyeFov[0], 1.0f);
        textureSizeRight_ = ovrHmd_GetFovTextureSize(hmd_, ovrEye_Right,
hmdDesc_.DefaultEyeFov[1], 1.0f);
        textureSize_.w = textureSizeLeft_.w + textureSizeRight_.w;
        textureSize_.h = (textureSizeLeft_.h > textureSizeRight_.h ?
textureSizeLeft_.h : textureSizeRight_.h);

    }

    /**
     * @brief Boolean that shows whether or not an instance has already been
    created
     * @details Part of the Singleton Pattern
     */
    static bool alreadyCreated;

    /**
     * @brief The generic OpenGL scene
     * @details Oculus is a templated class and its only argument is the
type of \a scene. The only requirement is that scene
     * has a method \a render, wich takes as argument the modelview matrix
and the projection matrix.
     */
    T & scene_;

    //GL
    /**
     * @brief The id of the OpenGL texture used in the Oculus rendering
     */
    GLuint textureId_;

    /**
     * @brief The id of the OpenGL Frame Buffer Object used in the Oculus
```

```
rendering
  */
 GLuint FBOId_;

 /**
  * @brief The id of the OpenGL depth buffer used in the Oculus rendering
  */
 GLuint depthBufferId_;

 //OVR
 /**
  * @brief The Oculus Rift
  * @details If no Oculus Rift is connected, a debug one is created. The
last does not have proper sensors.
  */
 ovrHmd hmd_;

 /**
  * @brief The description of the Oculus Rift
  * @details Contains lots of values like inter-pupillary distance,
resolution, etc
  */
 ovrHmdDesc hmdDesc_;

 /**
  * @brief The description of each eye
  * @details Contains lots of values like dimensions, wether it is the
left or right eye, etc.
  */
 ovrEyeRenderDesc eyeRenderDesc_[2];

 /**
  * @brief The texture of each eye
  */
 ovrGLTexture eyeTexture_[2];

 /**
  * @brief The Field of View of each eye
  */
 ovrFovPort eyeFov_[2];

 /**
  * @brief The configuration for the OpenGL Oculus rendering
  */
 ovrGLConfig cfg_;

 /**
  * @brief The dimensions of the window
  */
 ovrSizei windowSize_;

 /**
  * @brief The dimensions of the texture that the left eye can see
  */
 ovrSizei textureSizeLeft_;

 /**
```

```
     * @brief The dimensions of the texture that the right eye can see
     */
    ovrSizei textureSizeRight_;

    /**
     * @brief The dimensions of the texture overall
     */
    ovrSizei textureSize_;

    /**
     * @brief Time variable used by the sensor and the predication tool
     */
    ovrFrameTiming frameTiming_;

    /**
     * @brief The Oculus Rift sensors
     */
    ovrSensorState sensorState_;

    /**
     * @brief The Oculus Rift angular position
     */
    glm::vec3 angles_;

    /**
     * @brief The Oculus Rift angular position variation
     */
    glm::vec3 dAngles_;

    /**
     * @brief Flag used for the Oculus rendering configuration
     */
    int distortionCaps_;

    /**
     * @brief Boolean indicating if we are using a debug Oculus Rift
     */
    bool usingDebugHmd_;

    /**
     * @brief Boolean indicating if the Oculus rendering is multisampled
     * @warning The Oculus SDK does not seem to take this variable into
   account as of yet
     */
    bool multisampleEnabled_;
};

template<class T>
bool Oculus<T>::alreadyCreated = false;

/**
 * @brief The NullOculus class
 * @details Part of the Null object pattern
 */
class NullOculus: public GenericOculus
{
public:
```

```
    NullOculus();

    ~NullOculus();

    void render() {}
};

/**
 * @brief nullOculus
 * @details Implements the null object pattern
 */
extern std::unique_ptr<NullOculus> nullOculus;

#endif
```

Oculus.h

6.5 Glossaire

API
«Application Programming Interface», interface de programmation. Ensemble normalisé de classes et de fonctions qui sert de façade par laquelle un logiciel offre des services à d'autres logiciels.

C
Langage de programmation impératif, procédural utilisé dans des applications ayant un besoin critique de performances.

C++
Langage de programmation mutliplateforme, multiparadigme, générique, compilé, largement utilisé dans les domaines scientifiques, industriels, de l'entreprise, de l'image, ...

CPU
«Central Processing Unit», processeur. Composant de l'ordinateur qui exécute les instructions machine des programmes informatiques.

FPS
«Frame Per Seconds», mesure de la fluidité du rendu graphique en images par seconde.

Frame
Image rendue graphiquement par un programme, typiquement 60 fois par seconde.

Framework
Ensemble cohérent de composants logiciels structurels.

GPU
«Graphics Processing Unit», processeur graphique. Circuit intégré présent sur une carte graphique et assurant les fonctions de calcul de l'affichage.

Oculus Rift
Masque de réalité virtuelle fournissant une expérience d'immersion inédite.

Oculus VR
Entreprise de réalité virtuelle fabriquant l'Oculus Rift.

SDK
«Software Develoment Kit», kit de développement. Ensemble d'outils permettant aux développeurs de créer des applications de type défini.

Shader
Programme informatique, utilisé en image de synthèse, pour paramétrer une partie du processus de rendu réalisé par une carte graphique ou un moteur de rendu logiciel. Ils peuvent permettre de décrire l'absorption et la diffusion de la lumière, la texture à utiliser, les réflexions et réfractions, l'ombrage, le déplacement de primitives et des effets post-traitement.

6.6 Ressources

Site web du créateur du langage C++
http://www.stroustrup.com/

Conventions de code C++
http://www.stroustrup.com/JSF-AV-rules.pdf

Site d'Oculus pour les développeurs
https://developer.oculusvr.com/

Wiki officiel d'OpenGL
http://www.opengl.org/wiki/Main_Page

Site web sur les design patterns
http://gameprogrammingpatterns.com/

Blog sur le développement Oculus Rift
http://rifty-business.blogspot.fr

Tutoriel C++
http://cpp.developpez.com/faq/cpp/

Tutoriel OpenGL 3.x
http://tomdalling.com/blog/category/modern-opengl/

Site web de Skybot 3D
http://vo.imcce.fr/webservices/skybot3d

Site web de l'Observatoire
http://astro.unistra.fr/

Site web traitant du problème d'échelle et des grands nombres dans OpenGL
http://www.floatingorigin.com/

Autre tutoriel OpenGL 3.x
http://open.gl/

Oculus SDK
https://developer.oculusvr.com/?action=dl

Page Github du projet de Simulation
https://github.com/gaultier/Simulation_Stage_2014