ensiie
école nationale supérieure d'informatique pour l'industrie et l'entreprise

Ensiie Strasbourg

Rapport de stage

Auteur :
Philippe Gaultier,

élève ingénieur à l'Ensiie Strasbourg

Maître de stage :
André Schaaf,

enseignant-chercheur à l'Université de Strasbourg

Strasbourg, le 21 juillet 2014

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

---

Martin Golding

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

---

Unknown

Dans toute la suite du rapport, l'«Observatoire» désigne l'«Observatoire astronomique de Strasbourg» et l'«Unistra» ou l'«UDS» désignent l'«Université de Strasbourg».

Table des matières

1 Introduction

L'Observatoire est un établissement de recherche et d'enseignement centré sur l'astronomie, mais c'est aussi un centre de données astronomiques et un centre d'observation réputés mondialement. Il représente la continuité entre l'ancien et le nouveau car il dispose d'un riche patrimoine mais est aussi à la pointe de la recherche.
De plus l'Observatoire est le parfait exemple de l'informatique au service d'autres spécialités scientifiques, à la fois dans le domaine de l'expertise, mais aussi sous un aspect éducatif.
Pour toutes ces raisons, j'ai choisi d'effectuer mon stage de deuxième année au sein de l'Observatoire, au contact de technologies émergentes à savoir l'Oculus Rift et le rendu graphique 3D moderne.

2 Présentation de l'observatoire

2.1 Histoire

L'Observatoire a été fondé en 1881 sur l'initiative de l'empereur Guillaume II, l'Alsace étant allemande à cette époque.
Il est constitué de trois bâtiments : une Grande Coupole, un bâtiment des salles méridiennes avec deux coupoles, et un bâtiment à usage de bureau et de résidence.
La Grande Coupole en fer, de 9,2 mètres de diamètre et pesant 34 tonnes2, contient le Grand Réfracteur, une lunette de 48,7 cm d'ouverture et 7 m de focale, construite en 1877, la plus grande d'Europe au moment de son installation et aujourd'hui (2008) la troisième de France en taille.
Il dispose également d'un riche patrimoine d'instruments et d'ouvrage anciens.

2.2 Centre de données astronomiques de Strasbourg (CDS)

Le CDS est à la fois une équipe de recherche et un Service d'Observation. Les services de bases de données (SIMBAD, Vizier) et de visualisation (ALADIN) développés par le CDS sont utilisés par l'ensemble de la communauté astronomique mondiale.
Celui-ci est l'un des acteurs majeurs du développement de l'Observatoire Virtuel International en astronomie. Fin 2008, le CDS a été labellisé TGIR (Très Grande Infrastructure de Recherche) par le Ministère de l'Enseignement Supérieur et de la Recherche, reconfirmé comme Infrastructure de Recherche en 2012, ce qui le range au même niveau que des infrastructures internationales comme l'European Southern Observatory ou RENATER à l'échelon national.

2.3 Equipe de recherche Galaxies

L'équipe «Galaxies» étudie la formation et l'évolution des galaxies et de notre Galaxie au travers de leurs populations stellaires et de la dynamique des étoiles et de la matière noire.
Elle est impliquée dans la préparation de la mission satellitaire astrométrique Gaia de l'Agence Spatiale Européenne dont le lancement est prévu en 2012 et dans le grand relevé cinématique RAVE.

2.4 Equipe de recherche Hautes Énergies

L'équipe «Hautes Énergies» s'intéresse aux sources galactiques et extragalactiques émettrices en rayons X, objets compacts (étoiles à neutron, naines blanches, etc.) et noyaux actifs de galaxies.
Elle est impliquée dans le SSC-XMM, un consortium international de laboratoires sélectionné par l'ESA et labellisé par l'INSU comme Service d'Observation, qui est en charge de fournir des catalogues complets de sources X observées par le satellite XMM-Newton à la communauté internationale.

3 Mon stage

3.1 Objectif

L'objectif de ce stage a été centré autour de l'Oculus Rift et a été double :
— Intégration de l'Oculus Rift à une simulation 3D du système solaire existante,
— Développement d'un programme de visualisation 3D d'étoiles avec intégration de l'Oculus Rift
J'ai donc travaillé sur deux projets distincts mais néanmoins complémentaires.

3.1.1 Skybot 3D

Skybot 3D est un logiciel développé en C par l'institut de mécanique céleste et de calcul des éphémérides (IMCCE), conjointement avec l'Observatoire de Paris et le CNRS. Son propos est de faire un rendu graphique réaliste en 3D à partir des données célestes de ces instituts. En pratique, c'est une simulation 3D du système solaire où les échelles sont respectées. Il est encore en développement à la date d'écriture de ce document et sa sortie est prévue pour fin 2014. Il fonctionne sur toutes les distributions Linux et utilise OpenGL pour le rendu graphique.

Mon travail a donc consisté en l'intégration du rendu Oculus dans cette application, tout en gardant le rendu existant.

3.1.2 Simulation

Ce projet a consisté en la représentation 3D de données provenant du Centre de Données de l'Observatoire, décrivant la taille, la position, l'âge et la densité de corps célestes. Ces données sont stockées dans des fichiers texte ou binaires, pouvant contenir plusieurs millions d'objets.

J'ai eu la liberté de choisir les outils, le langage et les bibliothèques externes utilisées dans ce programme, n'ayant pas de base de code préexistante.

3.2 L'Oculus Rift

3.2.1 Aperçu

L'Oculus Rift est un masque de réalité virtuelle, développé par Oculus VR, une entreprise basée en Californie et rachetée par Facebook en mars 2014 pour 2 milliards $. L'Oculus Rift a été initialement financé via une plateforme de financement collaboratif, Kickstarter, et a levé 91 millions $ à cette occasion.
Il permet une immersion réaliste dans une scène en trois dimensions, en donnant l'impression

d'y être physiquement présent, et crée ainsi une nouvelle expérience utilisateur.
De plus, son prix est relativement peu élevé (350 $, environ 300 €), ce qui le rend accessible au grand public.
Pour toutes ces raisons, l'Oculus Rift est adapté à un usage éducatif et professionel, dans des domaines aussi variés que la simulation scientifique, le divertissement, l'éducation, ...

La version grand public est prévue pour fin 2014 ou début 2015. J'ai pour ma part travaillé avec la première version du masque, le DK1, tandis que la deuxième version, le DK2 a été distribuée à partir d'août 2014.

FIGURE 1 – L'Oculus Rift (DK1)

3.2.2 Fonctionnement

Matériel

L'Oculus Rift est composé de :

— Un écran 60 Hz d'une résolution de $1280 * 800$
— Deux lentilles (une pour chaque oeil),
— Un gyroscope à 3 axes pour mesurer l'accélération angulaire,
— Un magnétomètre à 3 axes pour mesurer les champs magnétiques,
— Un accéloromètre à 3 axes pour mesurer l'accélération, y compris gravitationnelle
— Un port USB
— Un port HDMI

Il est à noter que la résolution de l'écran est à diviser par deux, chaque oeil voyant seulement une moitié de l'écran, la résolution effective est donc de $640 * 800$.

Logiciel

L'utilisation de l'Oculus Rift s'effectue au moyen de son SDK, qui permet :

— D'accéder aux différents capteurs,
— D'accéder aux propriétés du masque (distance inter-pupillaire, hauteur des yeux, ...),
— D'appliquer les «filtres» au rendu graphique afin d'avoir un rendu réaliste

Le SDK est écrit en C++ et possède une API en C. Pour mon stage, j'ai utilisé la version 2.5 puis la version 0.3.2.

Théorie

L'Oculus Rift exige que la scène soit rendue graphiquement en «split-screen stereo», c'est-à-dire avec l'écran divisé en deux verticalement, la partie gauche réservée à l'oeil gauche et la partie droite à l'oeil droit.

La distance inter-pupillaire est la distance entre les deux yeux. Elle varie d'un individu à l'autre mais elle est en moyenne de 65 mm. Cette distance est importante dans le procédé de rendu car ce dernier consiste à rendre graphiquement la scène deux fois, une fois pour chaque oeil, en translatant la caméra de la distance inter-pupillaire entre les deux rendus. C'est ce qui contribue à créer l'effet stéréoscopique, ce qui crée l'impression d'immersion.

Un autre aspect à prendre en compte est la présence des lentilles. Ces dernières agrandissent l'image pour fournir un champ de vision très large, pour améliorer l'immersion. Cependant ce procédé déforme l'image de façon significative, ce qui créerait une distortion en coussinets si les «filtres», dont nous parleront plus tard, n'étaient pas appliqués au niveau logiciel au rendu graphique de l'application.

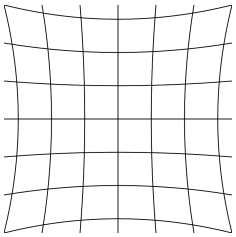

FIGURE 2 – Distortion en coussinets («pincushion distortion»)

Pour contrebalancer cette distortion, le programme doit, comme énoncé plus haut, appliquer un effet post-rendu. Il s'agit d'une distortion égale et opposée, appelée distortion en barillets.

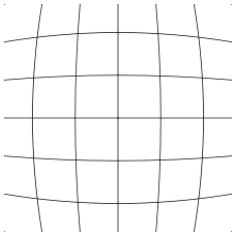

FIGURE 3 – Distortion en barillets («barrel distortion»)

De plus, le programme doit corriger les aberrations chromatiques, qui consistent en un effet d'arc en ciel aux contours des objets. Cet effet est causé par les lentilles.

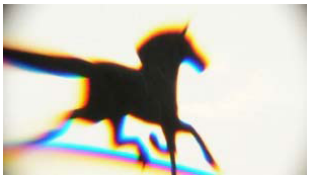

FIGURE 4 – Aberration chromatique («chromatic abberration»)

Pratique

Pour le développeur, ces éléments théoriques sont gérés de manière interne par le SDK Oculus.

Pour une application qui fait un rendu graphique, la programme est typiquement de la forme :

Algorithme 1 : Application de rendu graphique

```
initialize the graphic resources;
fill the scene with graphic objects;
while the application is running do
    process the user input;
    update the objects in the scene;
    render the objects in the scene;
end
release the graphic resources;
```

Un programme qui fait un rendu Oculus exclusivement aura pour sa part la forme suivante :

Algorithme 2 : Application de rendu graphique

```
initialize the graphic resources;
initialize the Oculus SDK;
fill the scene with graphic objects;
while the application is running do
    process the Oculus input;
    update the objects in the scene;
    for each eye do
        translate the camera by the inter-pupillary distance;
        apply the Oculus distortion effects;
        render the objects in the scene;
    end
end
release the Oculus SDK;
release the graphic resources;
```

Plus précisément, l'opération «apply the Oculus distortion effects» se fait de façon graphique au moyen de shaders, qui sont des programmes qui appliquent des transformations à chaque pixel de l'image.

Nous avons alors le rendu suivant, pour une scène simple composée d'un cube texturé, d'un plan et d'une skybox rudimentaire, avec le même point de vue :

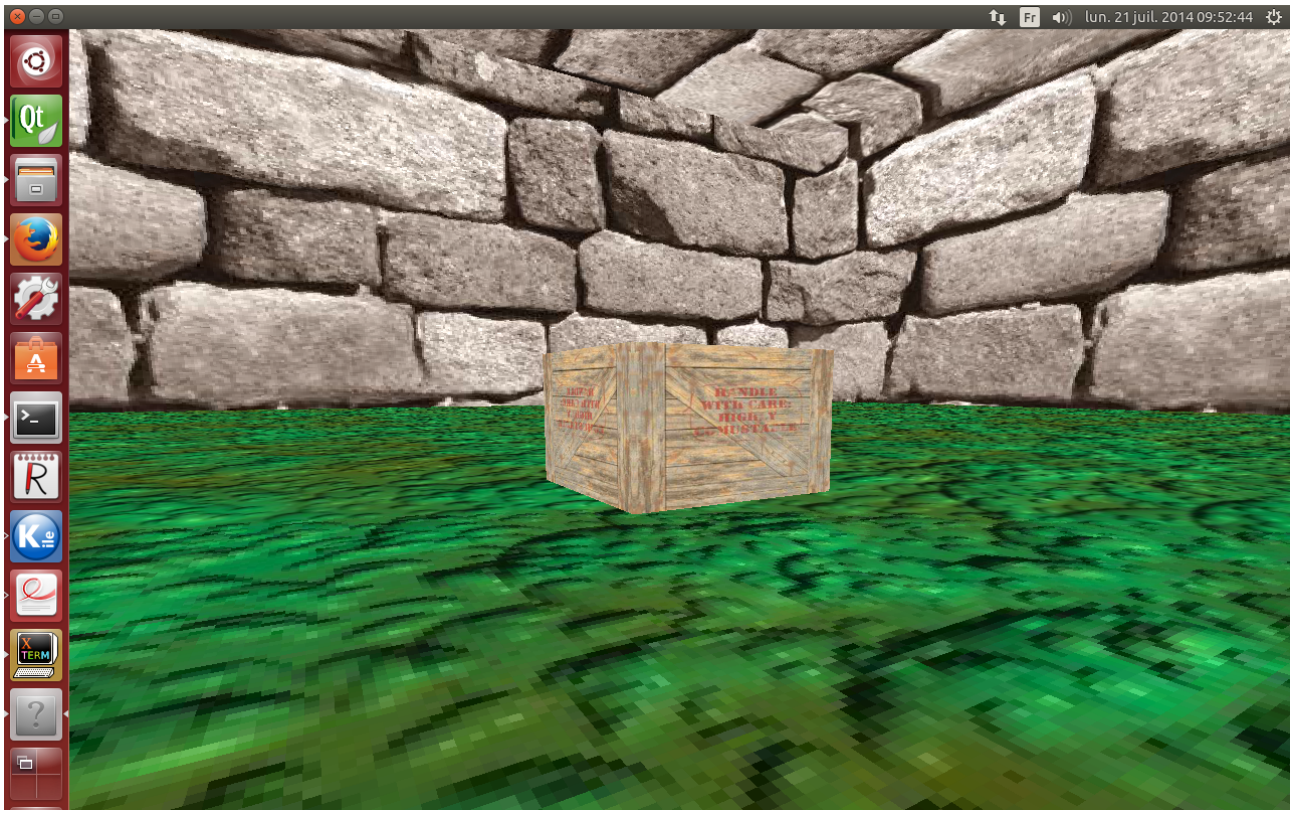

FIGURE 5 – Scène OpenGL simple avec le rendu normal

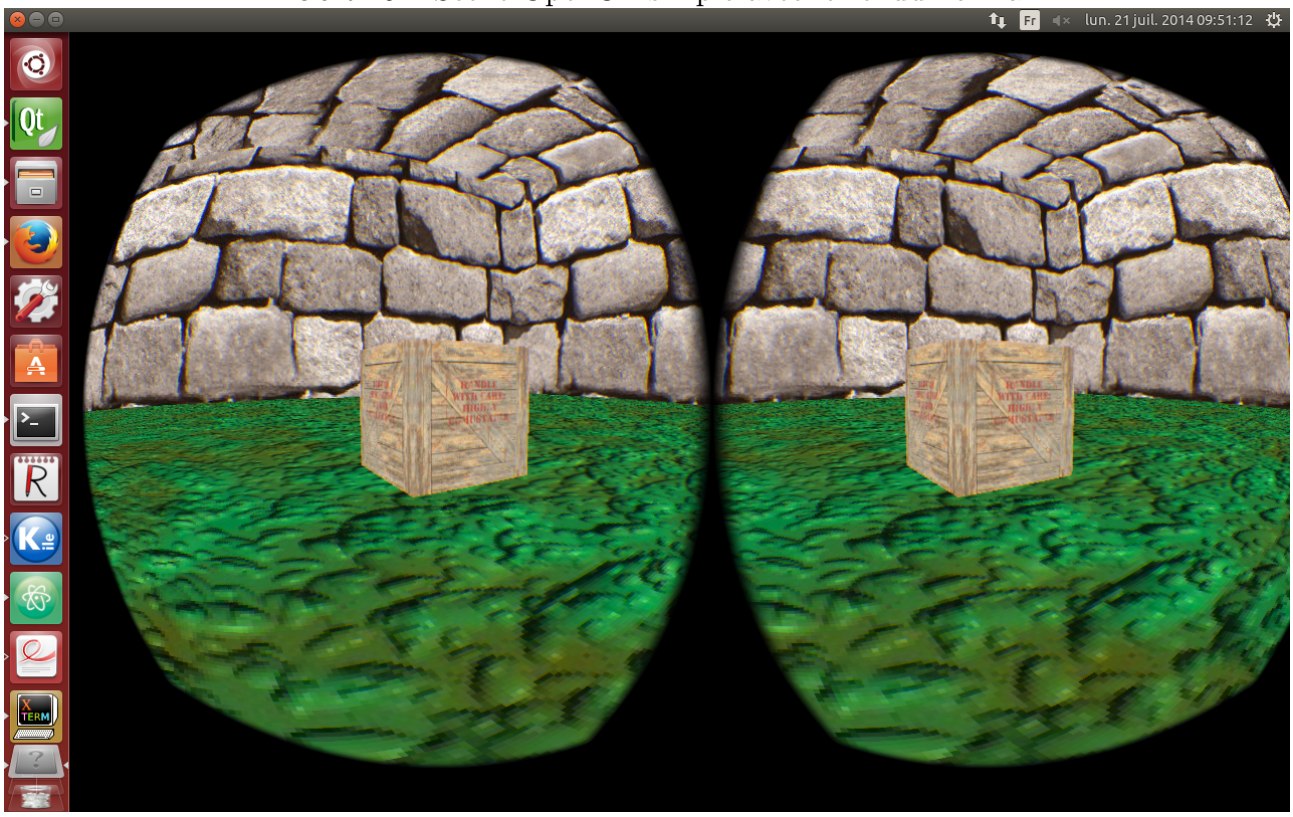

FIGURE 6 – Scène OpenGL simple avec le rendu Oculus

Les applications que j'ai développées au sein de mon stage peuvent fournir le rendu normal et le rendu Oculus, selon l'option spécifiée.

3.3 Contraintes

3.3.1 Skybot 3D

Existant

La contrainte principale pour le projet Skybot 3D a été de travailler avec du code existant non documenté. En effet, j'ai eu 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.

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, a été 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 éxecutés sur la carte graphique.

Le développeur doit donc maintenant tout faire manuellement mais cela 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 (distortion, correction des aberrations, ...), et cela a créé quelques conflits au niveau du rendu graphique, avec le rendu existant n'utilisant pas ces shaders.

Échelles

3.3.2 Simulation

Portabilité

Pour ce projet, il a été convenu 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'abstration 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 l'évitement de l'introduction de code spécifique à une plateforme donnée, par exemple en utilisant de façon maximale la libraire standard du langage.

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. 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 cubes de données (corps celéstes 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 est un arbre où chaque noeud (appelé «octant») compte jusqu'à 8 fils. Il

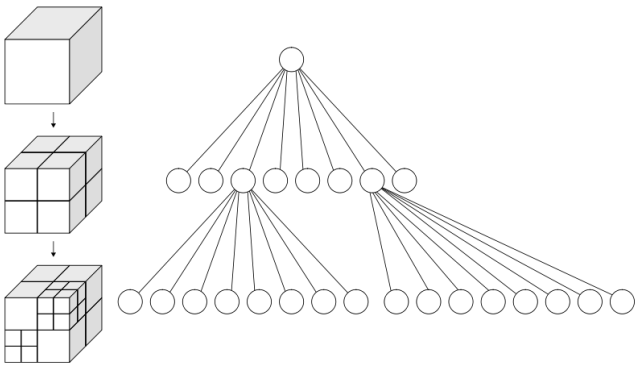

FIGURE 7 – Schématisation d'un octree

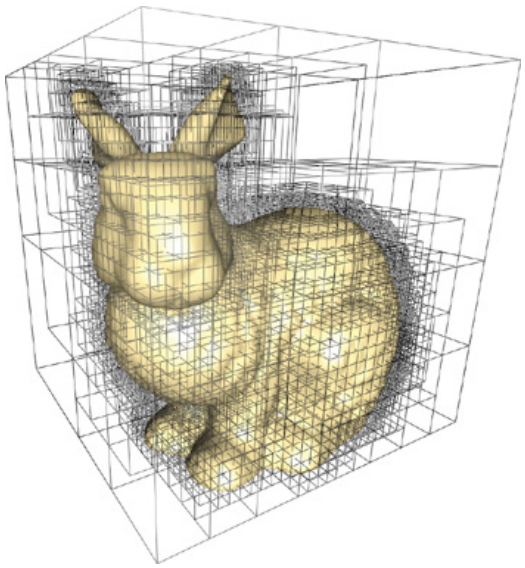

FIGURE 8 – Octree en action

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 Outils utilisés

3.4.1 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
— Mot-clé «auto»

3.4.2 Bibliothèques utilisés

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.

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.

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.

glm

«OpenGL Mathematics» est une bibliothèque de fonctions mathématiques pour OpenGL. Je l'ai utilisée pour le projet de simulation. Licence MIT.

GLEW

«OpenGL Extension Wrangler Library» est une bibliothèque multiplateforme de chargement des fonctionnalités d'OpenGL. Licence BSD modifiée.

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.

3.4.3 Programmes utilisés

QtCreator
: IDE C++ open source avec débuggeur et outils d'analyses intégrés. Licence LGPL.

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

GCC
: «GNU Compiler Collection», compilateur C/C++. Licence GNU GPL 3+.

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

Clang Static Analyzer
: Outil d'analyse statique (à la compilation) de code et qui fait partie du projet Clang.

Git
: Logiciel de gestion de version. Licence GNU GPL v2.

Github
: Site web de stockage en ligne de projets open source via git.

GDB
: «GNU Project Debugger», le déboggeur standard sur Linux. Licence GNU GPL.

Tous ces outils sont open source et multiplateformes.

3.5 Déroulement du stage

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.

La deuxième et troisième semaine 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.

La quatrième et cinquième semaine 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.

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

La neuvième et dixième semaine ont porté sur la rédaction de ce rapport et sur les possibles optimisations des deux projets.

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.

3.6 Développement

3.6.1 Bonnes pratiques

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 SCHAAF.

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 commenté de manière modérée mon code, 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.

3.6.2 Design Patterns

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. 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, 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.

Dans mon cas, j'ai travaillé avec la classe «GraphicObject», dont héritent les classes «Cube», «Plane», ..., mais aussi «NullGraphicObject». 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.

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 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 (3.2.2) et là (3.2.2).

3.6.3 Architecture

4 Remerciements

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

— André SCHAAF, 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

5 Conclusion

6 Annexes

6.1 Captures d'écran

6.2 Code source

```
#ifndef OCULUS_H
#define OCULUS_H

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

#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"

template<class T>
class Oculus
{

public:
    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);

        ovr_Initialize();

        hmd_ = ovrHmd_Create(0);
```

```
57          if (!hmd_)
            {
59              hmd_ = ovrHmd_CreateDebug(ovrHmd_DK1);
                usingDebugHmd_ = true;
61
                //Cannot create the debug hmd
63              assert(hmd_);
65
                LOG("Using the debug Hmd", std::cout);
67          }

69          ovrHmd_GetDesc(hmd_, &hmdDesc_);

71          computeSizes();

73          distortionCaps_ = ovrDistortionCap_Chromatic | ovrDistortionCap_TimeWarp
        ;

75          eyeFov_[0] = hmdDesc_.DefaultEyeFov[0];
            eyeFov_[1] = hmdDesc_.DefaultEyeFov[1];
77
            setOpenGLState();
79          initFBO();
            initTexture();
81          initDepthBuffer();

83          computeSizes();
            setCfg();
85          setEyeTexture();
            ovrBool configurationRes = ovrHmd_ConfigureRendering(hmd_, &cfg_.Config,
         distortionCaps_, eyeFov_, eyeRenderDesc_);
87          //Cannot configure OVR rendering
            assert(configurationRes);
89
            ovrHmd_StartSensor(hmd_, ovrSensorCap_Orientation |
        ovrSensorCap_YawCorrection | ovrSensorCap_Position, ovrSensorCap_Orientation)
        ;
91
            Oculus::alreadyCreated = true;
93      }
        ~Oculus()
95      {
            glDeleteFramebuffers(1, &FBOId_);
97          glDeleteTextures(1, &textureId_);
            glDeleteRenderbuffers(1, &depthBufferId_);
99
            ovrHmd_Destroy(hmd_);
101
            ovr_Shutdown();
103
            Oculus::alreadyCreated = false;
105     }

107     void render()
        {
```

```
109         glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0);
            glBindBuffer(GL_ARRAY_BUFFER, 0);
111         glUseProgram(0);

113         frameTiming_ = ovrHmd_BeginFrame(hmd_, 0);

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

121         getInput();

123         ovrPosef eyeRenderPose[2];

125         for (int eyeIndex = 0; eyeIndex < ovrEye_Count; eyeIndex++)
            {
127             ovrEyeType eye = hmdDesc_.EyeRenderOrder[eyeIndex];
                eyeRenderPose[eye] = ovrHmd_BeginEyeRender(hmd_, eye);
129
                glViewport(eyeTexture_[eye].OGL.Header.RenderViewport.Pos.x,
131                            eyeTexture_[eye].OGL.Header.RenderViewport.Pos.y,
                            eyeTexture_[eye].OGL.Header.RenderViewport.Size.w,
133                         eyeTexture_[eye].OGL.Header.RenderViewport.Size.h
                            );
135
                // Get Projection and ModelView matrici from the device...
137             OVR::Matrix4f MV = OVR::Matrix4f::Translation(eyeRenderDesc_[eye].
    ViewAdjust)
                        * OVR::Matrix4f(OVR::Quatf(eyeRenderPose[eye].Orientation).
    Inverted());
139
                OVR::Matrix4f Proj = OVR::Matrix4f(ovrMatrix4f_Projection(
    eyeRenderDesc_[eye].Fov, 0.01f, 10000.0f, true));
141
                glm::mat4 glmMV = Utils::ovr2glmMat(MV.Transposed());
143
                glm::mat4 glmProj = Utils::ovr2glmMat(Proj.Transposed());
145
                scene_->render(glmMV, glmProj);
147             Utils::GLGetError();

149             ovrHmd_EndEyeRender(hmd_, eye, eyeRenderPose[eye], &eyeTexture_[eye
    ].Texture);
            }
151
            ovrHmd_EndFrame(hmd_);
153


155     }

157     bool isUsingDebugHmd()
        {
159         return usingDebugHmd_;
        }
161
```

```
    bool isMoving() const
163 {
        bool res = false;
165     for(int i=0; i < 3; i++)
        {
167         res = res && Utils::isEqual(angles_[i], dAngles_[i]);
        }
169     return !res;

171 }

173 glm::vec3 dAngles() const
    {
175     return dAngles_;
    }
177
    void setDAngles(const glm::vec3 &dAngles)
179 {
        dAngles_ = dAngles;
181 }

183 glm::vec3 angles() const
    {
185     return angles_;
    }
187 void setAngles(const glm::vec3 &angles)
    {
189     angles_ = angles;
    }
191 void getInput()
    {
193     glm::vec3 oldAngles = angles_;

195     sensorState_ = ovrHmd_GetSensorState(hmd_, frameTiming_.
    ScanoutMidpointSeconds);

197     if(sensorState_.StatusFlags & (ovrStatus_OrientationTracked |
    ovrStatus_PositionTracked))
        {
199         ovrPosef pose = sensorState_.Predicted.Pose;
            OVR::Quatf quat = pose.Orientation;
201
            quat.GetEulerAngles<OVR::Axis_Y, OVR::Axis_X, OVR::Axis_Z>(&angles_.
    x, &angles_.y, &angles_.z);
203
            dAngles_ = angles_ - oldAngles;
205
            LOG("Angles: " +
207             std::to_string(OVR::RadToDegree(angles_[0])) + ", " +
                std::to_string(OVR::RadToDegree(angles_[1])) + ", " +
209             std::to_string(OVR::RadToDegree(angles_[2])) + " degrees"
                , std::cout);
211         LOG("Angles: " +
                std::to_string(angles_[0]) + ", " +
213             std::to_string(angles_[1]) + ", " +
                std::to_string(angles_[2]) + " rad"
215             , std::cout);
```

```
217             LOG("dAngles_: " +
                        std::to_string(OVR::RadToDegree(dAngles_[0])) + ", " +
219                     std::to_string(OVR::RadToDegree(dAngles_[1])) + ", " +
                        std::to_string(OVR::RadToDegree(dAngles_[2])) + " degrees"
221                     , std::cout);
        }
223     else
        {
225         LOG("No input data (using debug Hmd)", std::cout);
        }
227

    }
229

protected:
231     void initTexture()
    {
233         // The texture we're going to render to...
            glGenTextures(1, &textureId_);
235         // "Bind" the newly created texture : all future texture functions will
    modify this texture...
            glBindTexture(GL_TEXTURE_2D, textureId_);
237         // 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);
239         // Linear filtering...
            glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
241         glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);

243         Utils::GLGetError();

245     }
    void initFBO()
247     {
            // We will do some offscreen rendering, setup FBO...
249         assert(textureSize_.w != 0);
            assert(textureSize_.h != 0);
251

            glGenFramebuffers(1, &FBOId_);
253         Utils::GLGetError();
            //Cannot create the FBO
255         assert(FBOId_ != 0);

257         glBindFramebuffer(GL_FRAMEBUFFER, FBOId_);
            Utils::GLGetError();
259     }
    void initDepthBuffer()
261     {
            glGenRenderbuffers(1, &depthBufferId_);
263         Utils::GLGetError();
            //Cannot create the depth buffer
265         assert(depthBufferId_ != 0);

267         glBindRenderbuffer(GL_RENDERBUFFER, depthBufferId_);
            Utils::GLGetError();
269
```

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

273         glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT,
    GL_RENDERBUFFER, depthBufferId_);
            Utils::GLGetError();
275
            // Set the texture as our colour attachment #0...
277         glFramebufferTexture(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, textureId_,
    0);
            Utils::GLGetError();
279
            // Set the list of draw buffers...
281         GLenum GLDrawBuffers[1] = { GL_COLOR_ATTACHMENT0 };

283         glDrawBuffers(1, GLDrawBuffers); // "1" is the size of DrawBuffers
            Utils::GLGetError();
285
            // Check if everything is OK...
287         GLenum check = glCheckFramebufferStatus(GL_DRAW_FRAMEBUFFER);

289         //There is a problem with the FBO
            assert(check == GL_FRAMEBUFFER_COMPLETE);
291
            // Unbind...
293         glBindRenderbuffer(GL_RENDERBUFFER, 0);
            glBindTexture(GL_TEXTURE_2D, 0);
295         glBindFramebuffer(GL_FRAMEBUFFER, 0);
            Utils::GLGetError();
297
    }
299 void setOpenGLState()
    {
301         // Some state...

303         glDisable(GL_TEXTURE_2D);
            glEnable(GL_DEPTH_TEST);
305         if(multisampleEnabled_)
            {
307             glEnable(GL_MULTISAMPLE);
            }
309
    }
311
    void setCfg()
313 {
            cfg_.OGL.Header.API = ovrRenderAPI_OpenGL;
315         cfg_.OGL.Header.Multisample = multisampleEnabled_;
            cfg_.OGL.Header.RTSize.w = windowSize_.w;
317         cfg_.OGL.Header.RTSize.h = windowSize_.h;

319         SDL_SysWMinfo info;
            SDL_VERSION(&info.version);
321         SDL_bool infoRes = SDL_GetWindowWMInfo(scene_->getWindow(), &info);
            //Cannot retrieve SDL window info
323         assert(infoRes == SDL_TRUE);
```

```
325 #if defined(OVR_OS_WIN32)
            cfg.OGL.Window = info.info.win.window;
327 #elif defined (OVR_OS_MAC)
            cfg.OGL.Window = info.info.cocoa.window
329 #elif defined(OVR_OS_LINUX)
            cfg_.OGL.Win = info.info.x11.window;
331         cfg_.OGL.Disp = info.info.x11.display;
    #endif
333     }
        void setEyeTexture()
335     {
            eyeTexture_[0].OGL.Header.API = ovrRenderAPI_OpenGL;
337         eyeTexture_[0].OGL.Header.TextureSize.w = textureSize_.w;
            eyeTexture_[0].OGL.Header.TextureSize.h = textureSize_.h;
339         eyeTexture_[0].OGL.Header.RenderViewport.Pos.x = 0;
            eyeTexture_[0].OGL.Header.RenderViewport.Pos.y = 0;
341         eyeTexture_[0].OGL.Header.RenderViewport.Size.h = textureSize_.h;
            eyeTexture_[0].OGL.Header.RenderViewport.Size.w = textureSize_.w/2;
343         eyeTexture_[0].OGL.TexId = textureId_;

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

349     }
        void computeSizes()
351     {
            windowSize_.w = scene_->windowWidth();  //hmdDesc_.Resolution.w;
353         windowSize_.h = scene_->windowHeight(); //hmdDesc_.Resolution.h;

355         LOG("FoV: " + std::to_string(Utils::radToDegree(2 * atan(hmdDesc_.
    DefaultEyeFov[0].UpTan))), std::cout );

357         textureSizeLeft_ = ovrHmd_GetFovTextureSize(hmd_, ovrEye_Left, hmdDesc_.
    DefaultEyeFov[0], 1.0f);
            textureSizeRight_ = ovrHmd_GetFovTextureSize(hmd_, ovrEye_Right,
    hmdDesc_.DefaultEyeFov[1], 1.0f);
359         textureSize_.w = textureSizeLeft_.w + textureSizeRight_.w;
            textureSize_.h = (textureSizeLeft_.h > textureSizeRight_.h ?
    textureSizeLeft_.h : textureSizeRight_.h);
361
        }
363
        static bool alreadyCreated;
365
        T* scene_;
367
        //GL
369     GLuint textureId_;
        GLuint FBOId_;
371     GLuint depthBufferId_;

373     //OVR
        ovrHmd hmd_;
375     ovrHmdDesc hmdDesc_;
```

```
    ovrEyeRenderDesc eyeRenderDesc_[2];
377 ovrGLTexture eyeTexture_[2];
    ovrFovPort eyeFov_[2];
379 ovrGLConfig cfg_;
    ovrSizei windowSize_;
381 ovrSizei textureSizeLeft_;
    ovrSizei textureSizeRight_;
383 ovrSizei textureSize_;
    ovrFrameTiming frameTiming_;
385 ovrSensorState sensorState_;
    glm::vec3 angles_;
387 glm::vec3 dAngles_;
    int distortionCaps_;
389 bool usingDebugHmd_;
    bool multisampleEnabled_;
391
};
393
template<class T>
395 bool Oculus<T>::alreadyCreated = false;


397


399 #endif
```

Oculus.h

6.3 Bibliographie

6.4 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.