Ensiie Strasbourg

Rapport de stage

Auteur :
Philippe
Gaultier,

élève ingénieur à l'Ensiie
Strasbourg

Maître de stage :
André schaafff,

Ingénieur de recherche au
CNRS

Strasbourg, le 4 août 2014

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

Table des figures

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 réputé 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 en astronomie.
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 tonnes, 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 (2014) 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.

Le CDS est l'un des fondateurs de l'IVOA («International Virtual Observatory Alliance») qui travaille à l'élaboration de standards d'interopérabilité pour les données astronomiques.

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 était centré autour de l'Oculus Rift et était double :

1. Intégration de l'Oculus Rift à une visualisation 3D du système solaire existante (Skybot 3D),
2. Développement d'un programme de simulation 3D d'objets célestes avec intégration de l'Oculus Rift (Simulation)

J'ai donc travaillé sur deux projets distincts mais néanmoins complémentaires.
Le but a donc été d'expérimenter en profondeur les possibilités de l'Oculus Rift, de développer les deux applications citées ci-dessus en tant que preuve de concept de l'utilisation de cette nouvelle technologie dans le domaine de l'éducation et de la simulation, et enfin d'obtenir des performances et une expérience utilisateur correctes. Ce fut donc un stage de recherche avec un rendu final concret.

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 visualisation 3D du système solaire où les échelles sont respectées, avec gestion du temps. 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 professionnel, 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 0.2.5 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 engendre 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 distorsion en coussinets si les «filtres», dont nous parleront plus tard, n'étaient pas appliqués au niveau logiciel au rendu graphique de l'application.

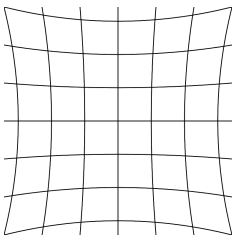

FIGURE 2 – Distorsion en coussinets («pincushion distorsion»)

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

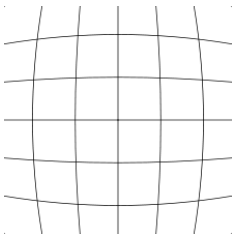

FIGURE 3 – Distorsion en barillets («barrel distorsion»)

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 et est bien connu dans le domaine de l'optique.

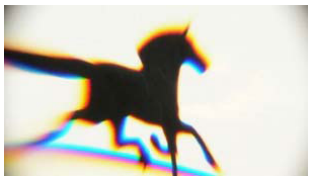

FIGURE 4 – Aberration chromatique («chromatic abberration»)

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 ressources;
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 ressources;
```

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

Algorithme 2 : Application de rendu graphique

```
initialize the graphic ressources;
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 distorsion effects;
        render the objects in the scene;
    end
end
release the Oculus SDK;
release the graphic ressources;
```

Plus précisément, l'opération «apply the Oculus distorsion 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 :

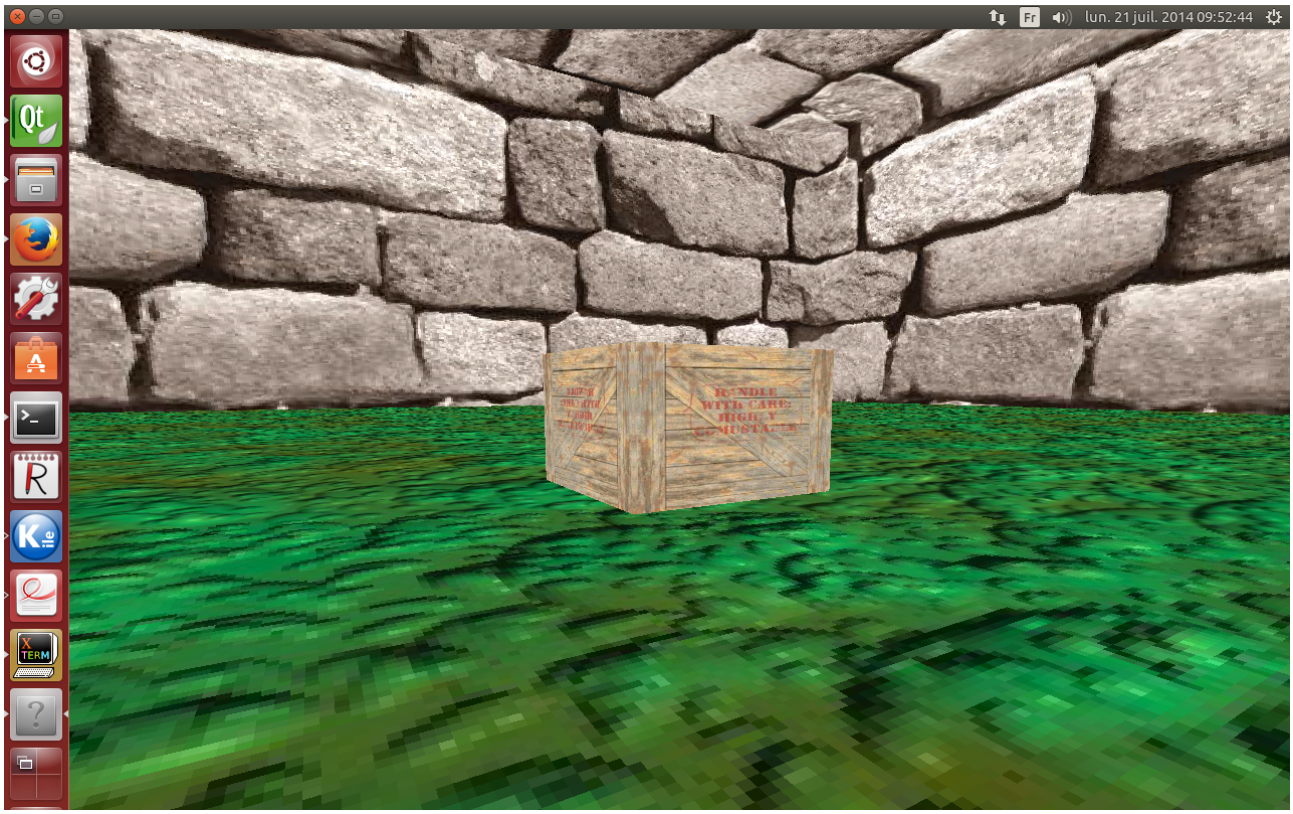

FIGURE 5 – Scène OpenGL simple avec le rendu normal

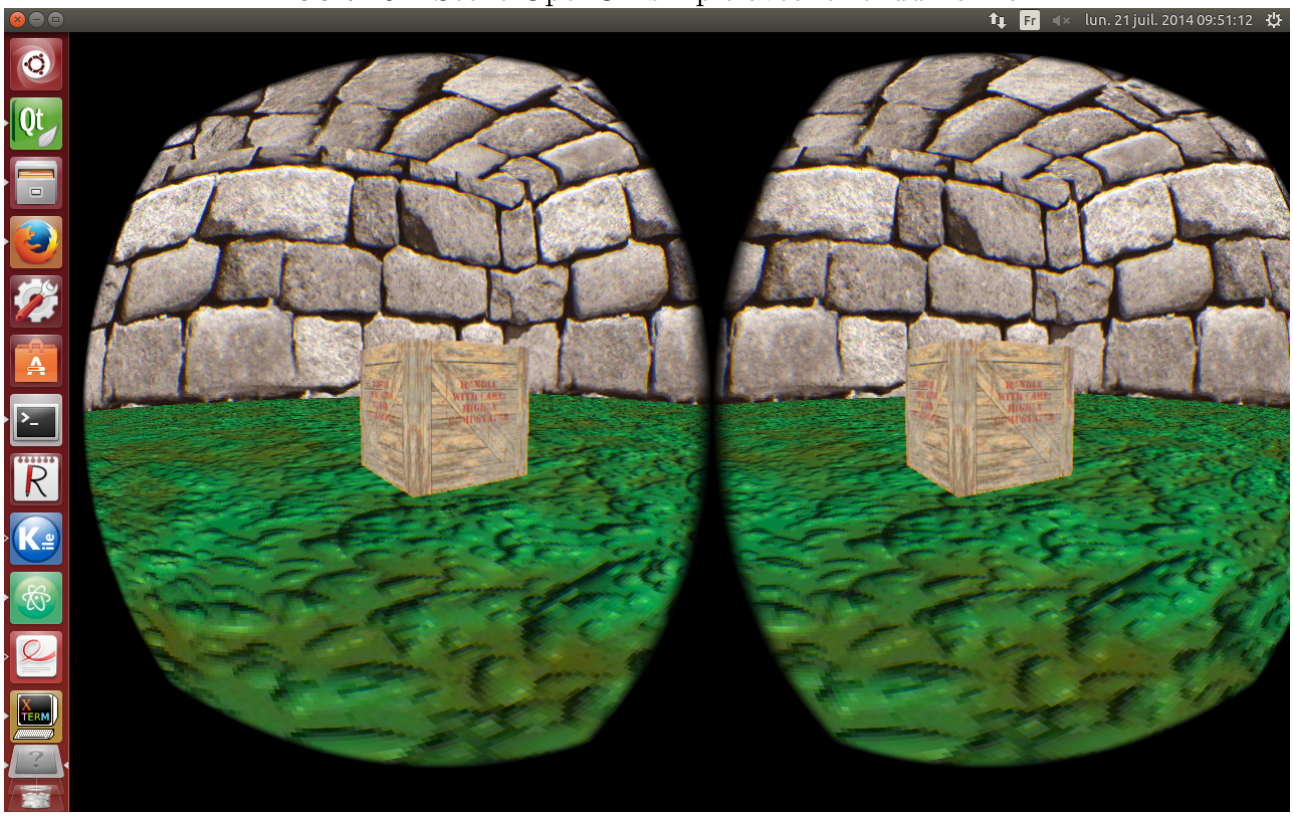

FIGURE 6 – Scène OpenGL simple avec le rendu Oculus

Les applications que j'ai développées durant 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, de taille conséquente (3840 fichiers, 22863 lignes de code). 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, 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, 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 exécutés 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 (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. 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 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.

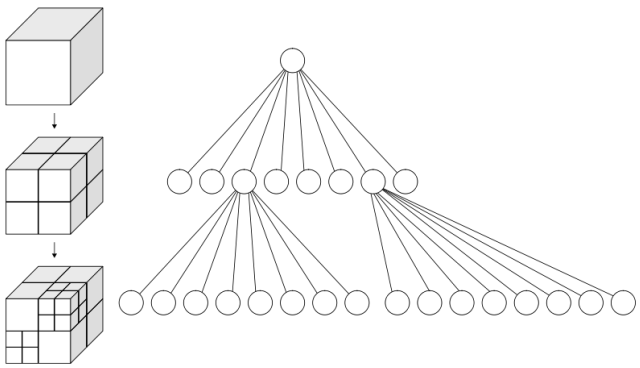

FIGURE 7 – Schématisation d'un octree

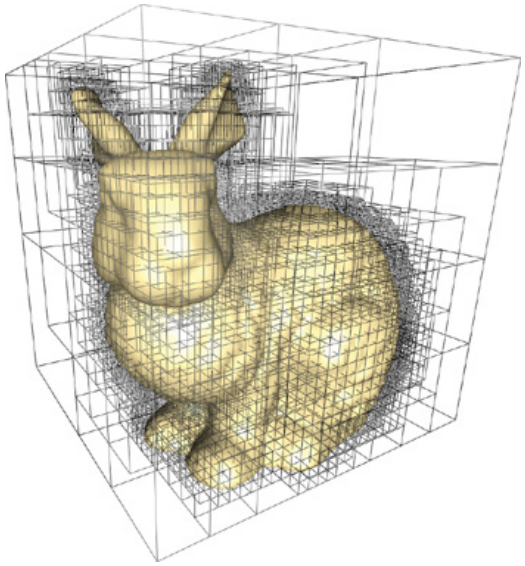

FIGURE 8 – Octree en action

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 (3.2.2) pour le rendu normal et là (3.2.2) 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

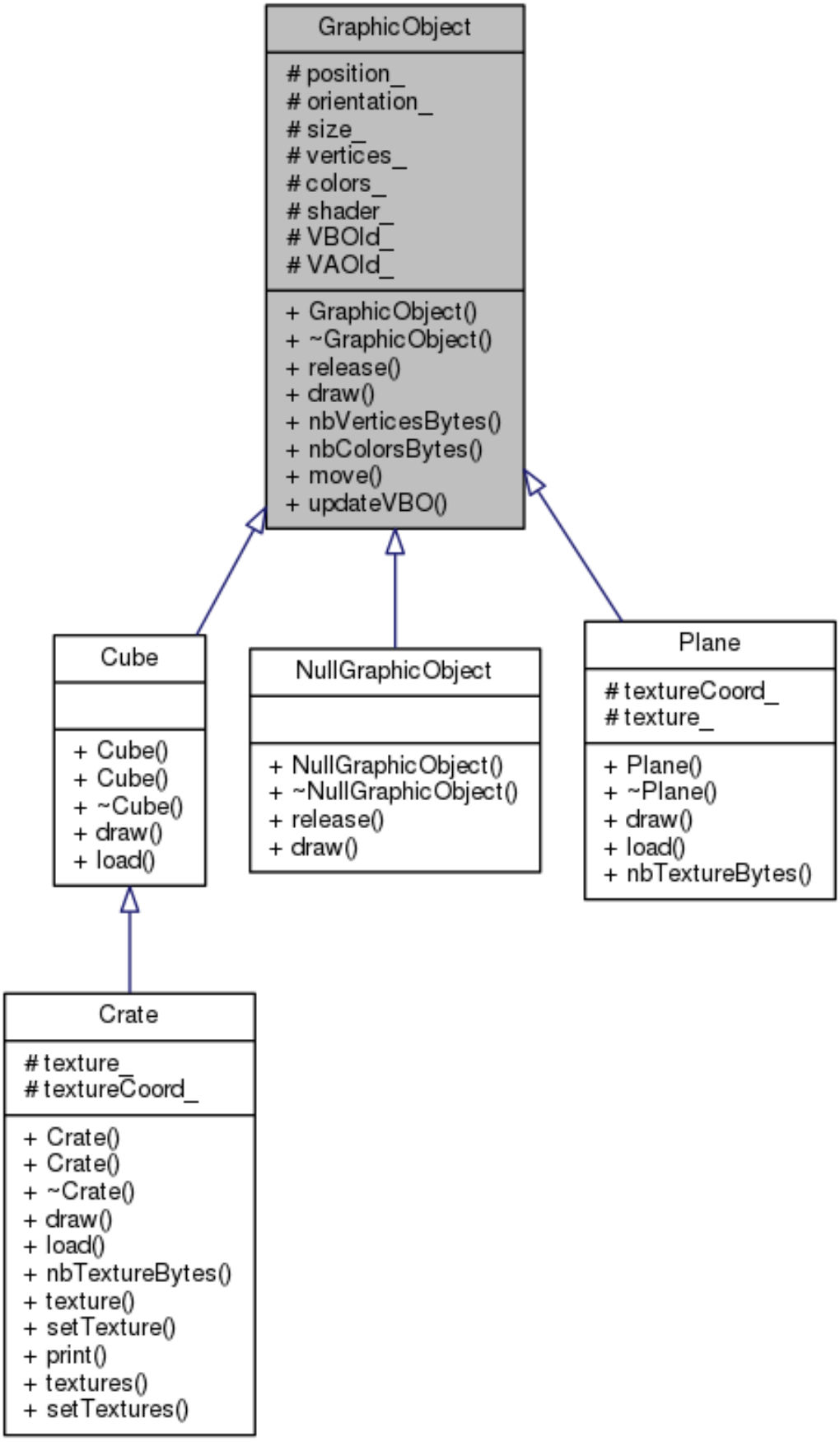

FIGURE 9 – Diagramme de classe UML pour les objets graphiques

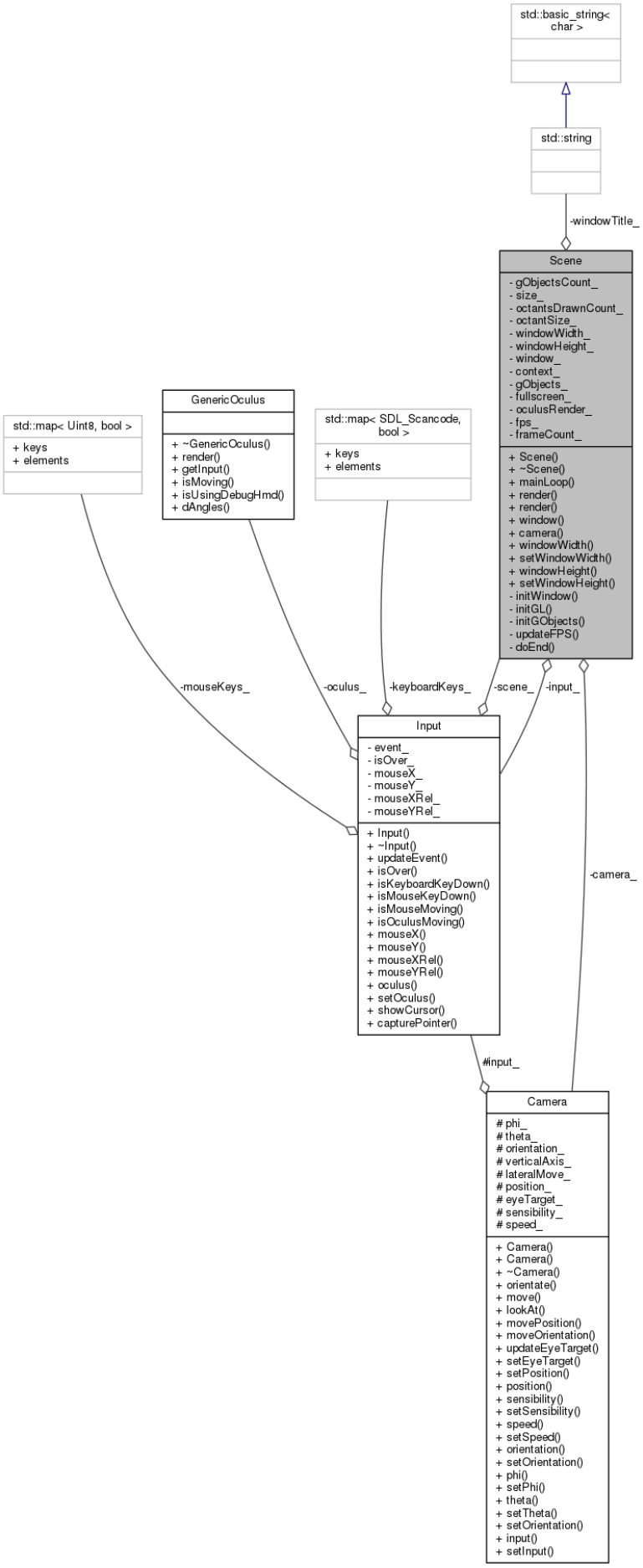

FIGURE 10 – Diagramme de classe UML pour la scène

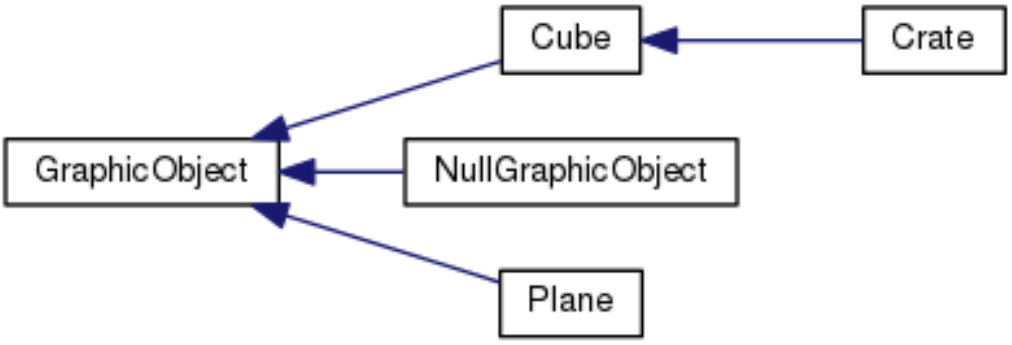

FIGURE 11 – Hiérarchie des classes

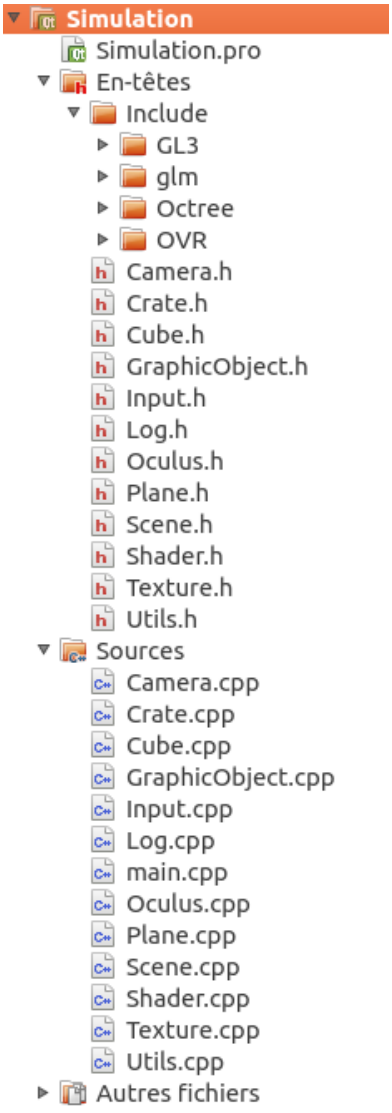

FIGURE 12 – Hiérarchie des fichiers

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

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

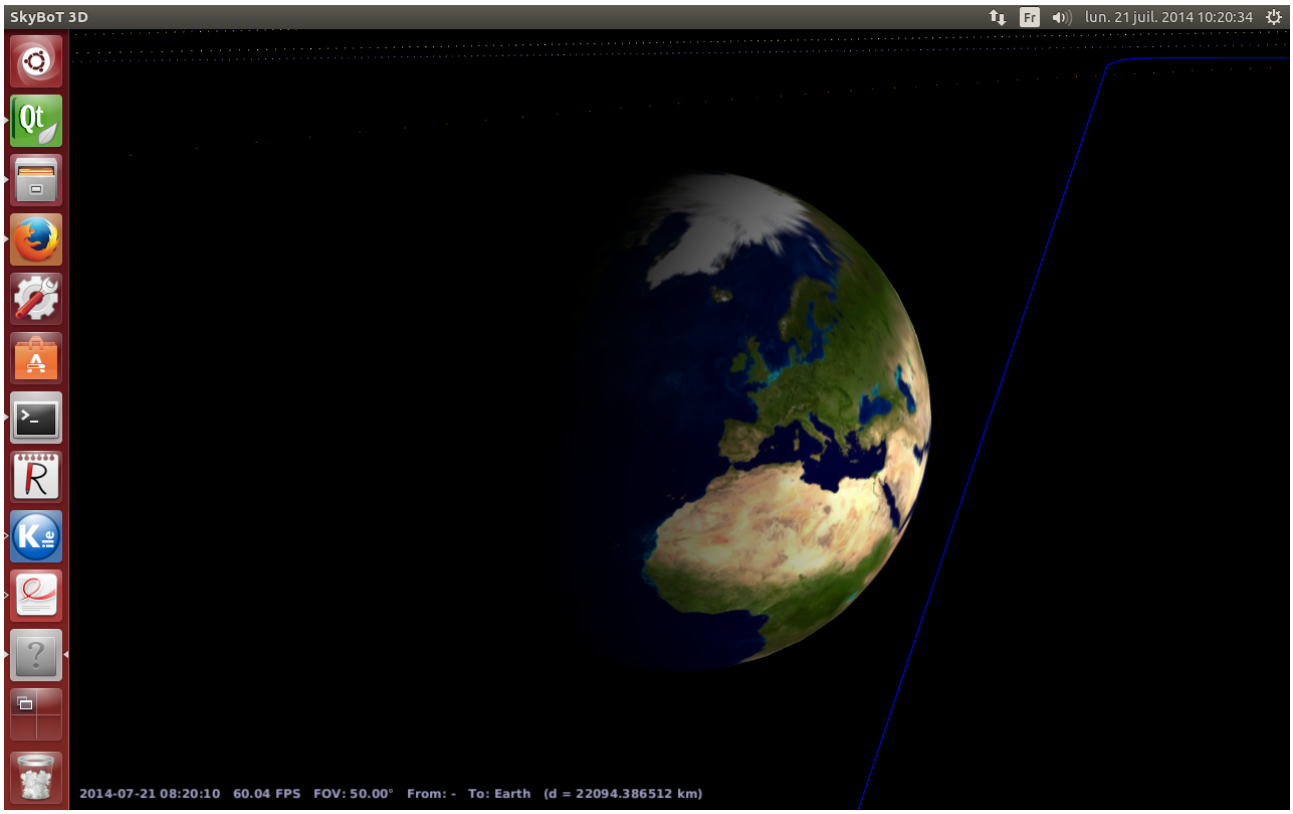

FIGURE 13 – Visualisation de la planète Terre dans Skybot 3D en vue normale

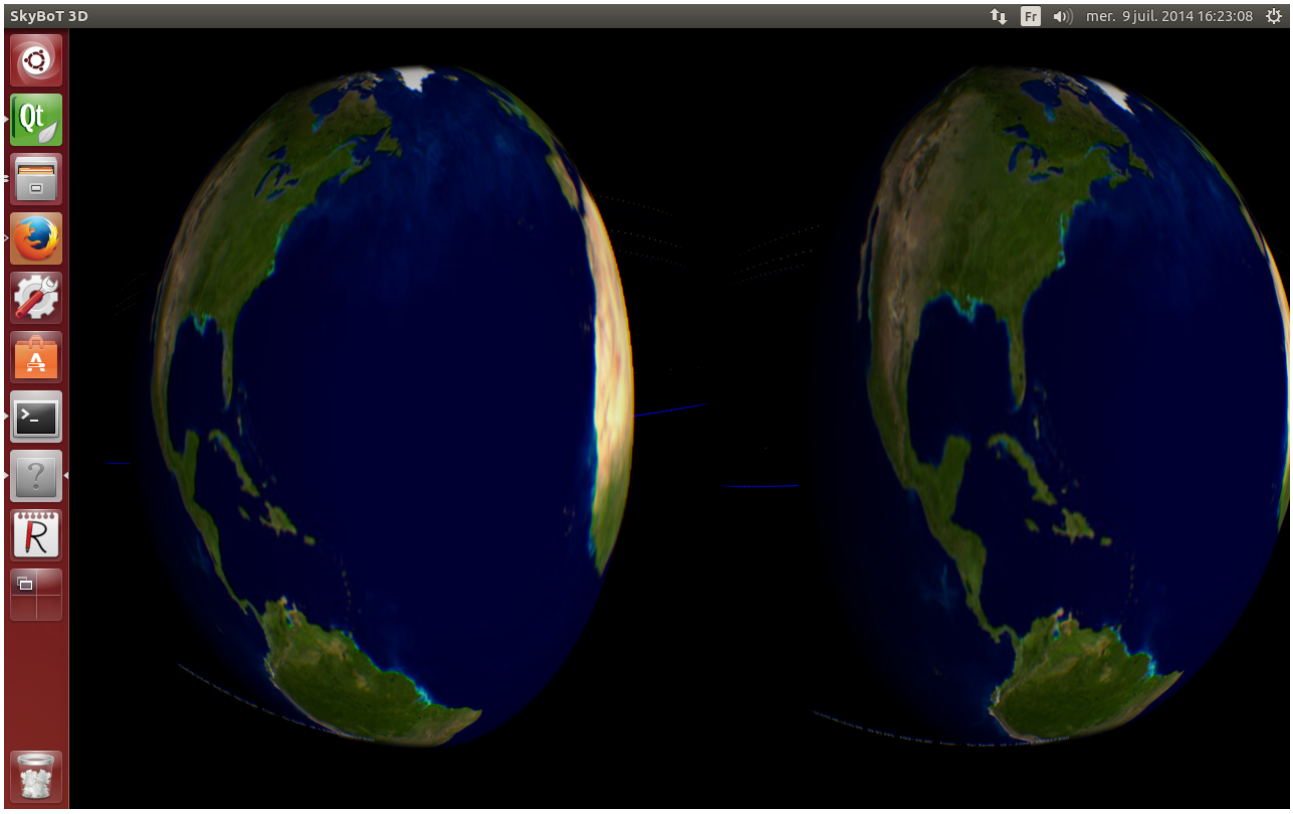

FIGURE 14 – Visualisation de la planète Terre dans Skybot 3D en vue Oculus

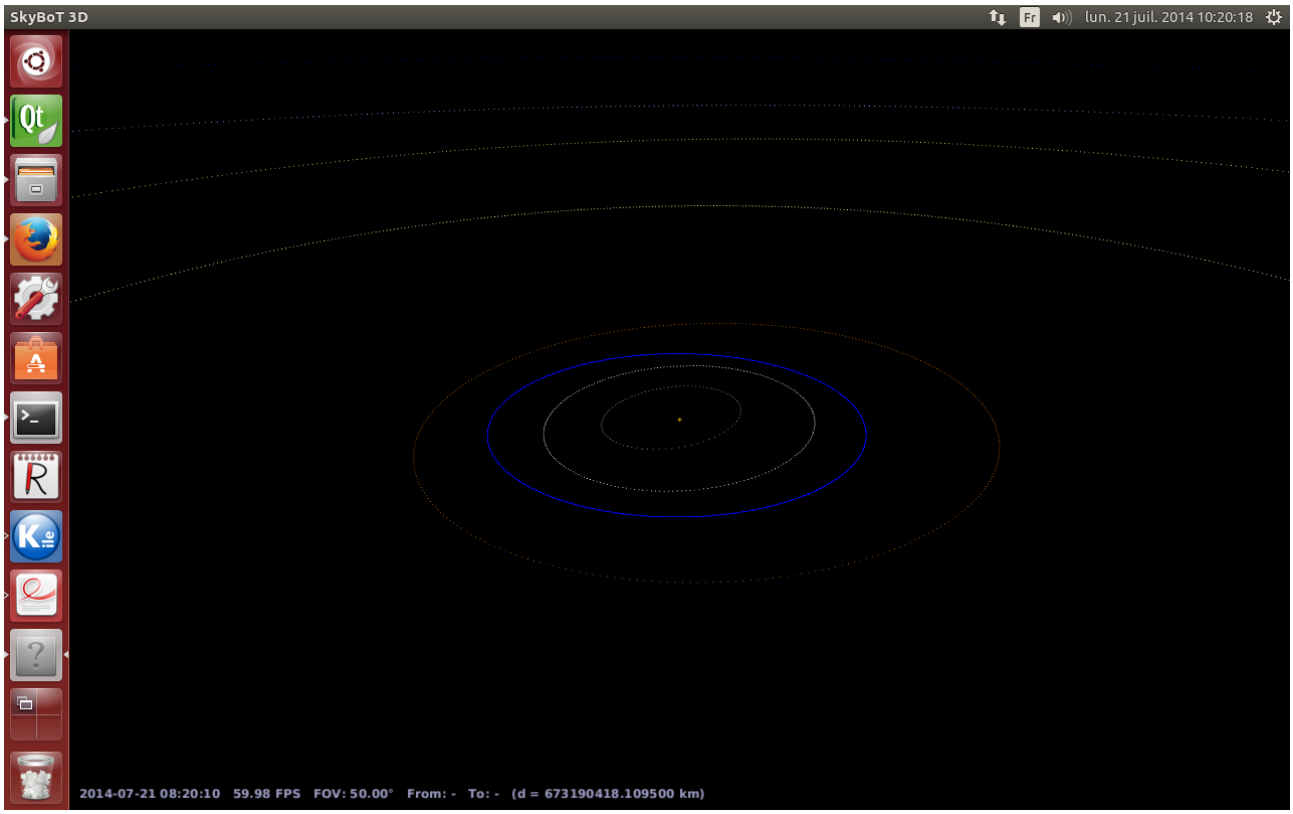

FIGURE 15 – Visualisation du système solaire dans Skybot 3D en vue normale

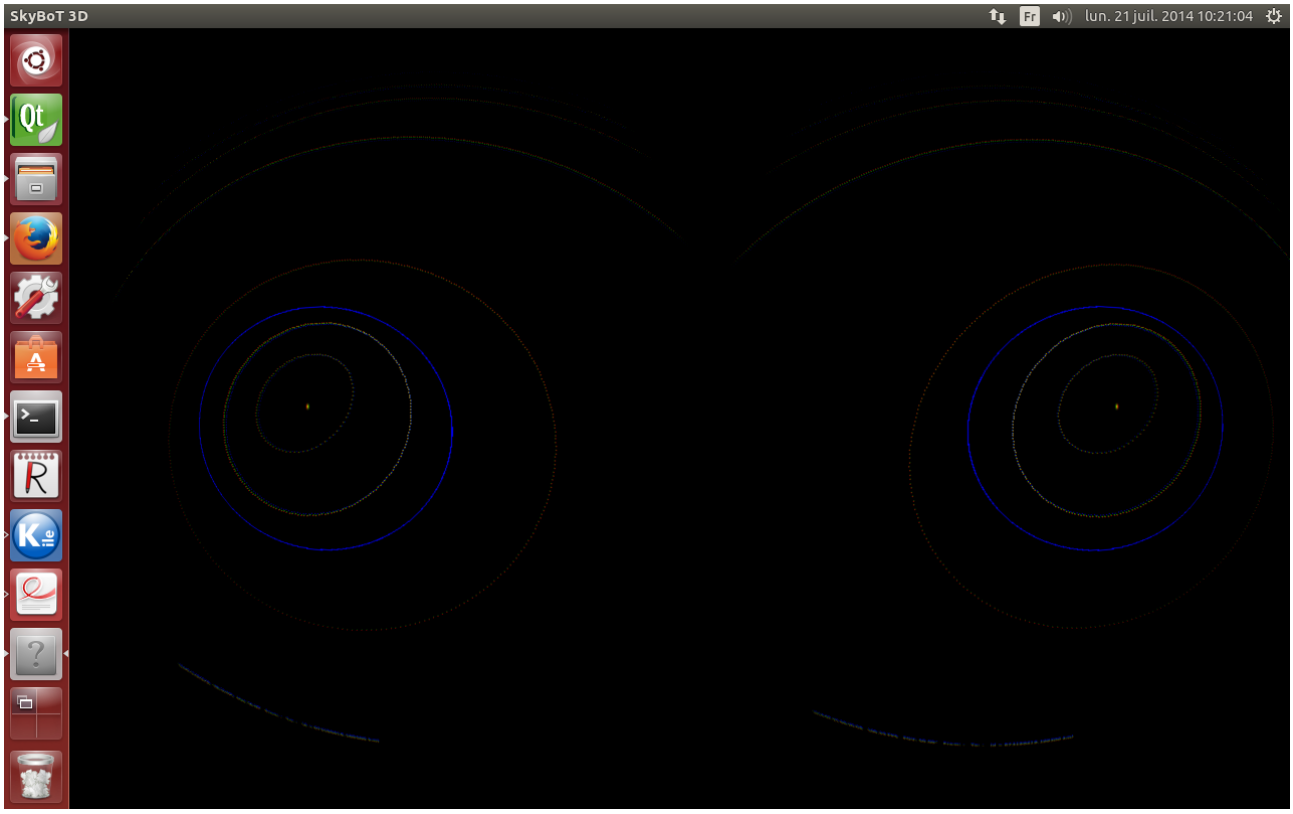

FIGURE 16 – Visualisation du système solaire dans Skybot 3D en vue Oculus

Les images suivantes 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.

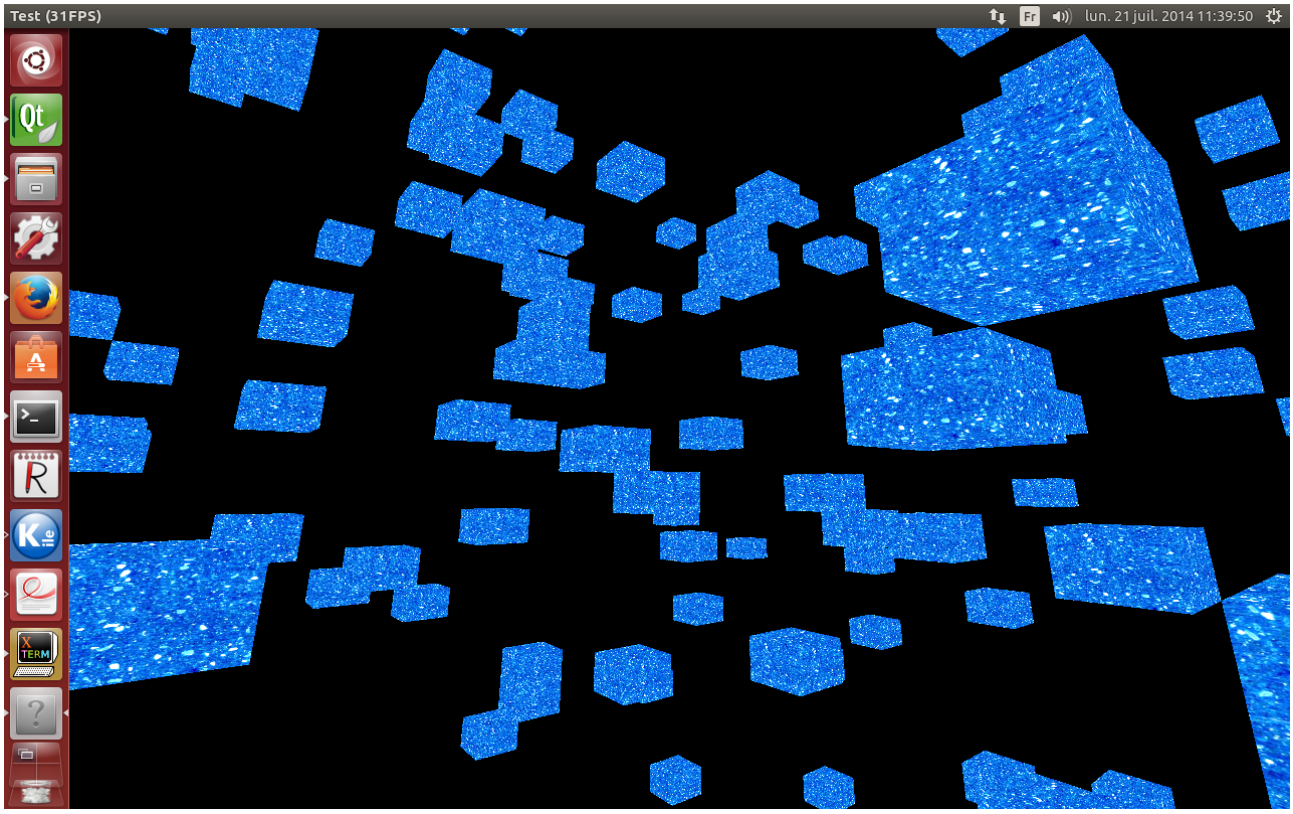

FIGURE 17 – Simulation en vue normale

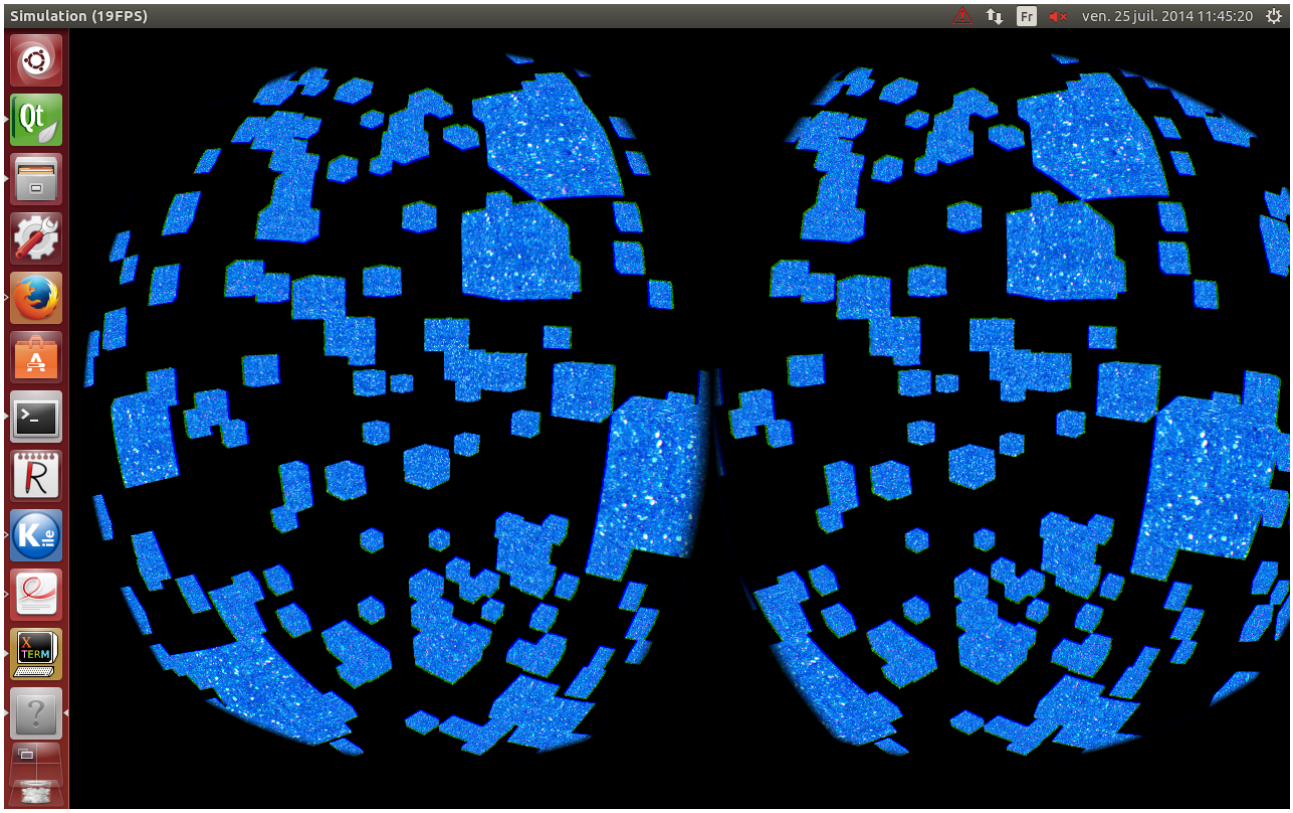

FIGURE 18 – Simulation en vue normale

Les images suivantes 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.

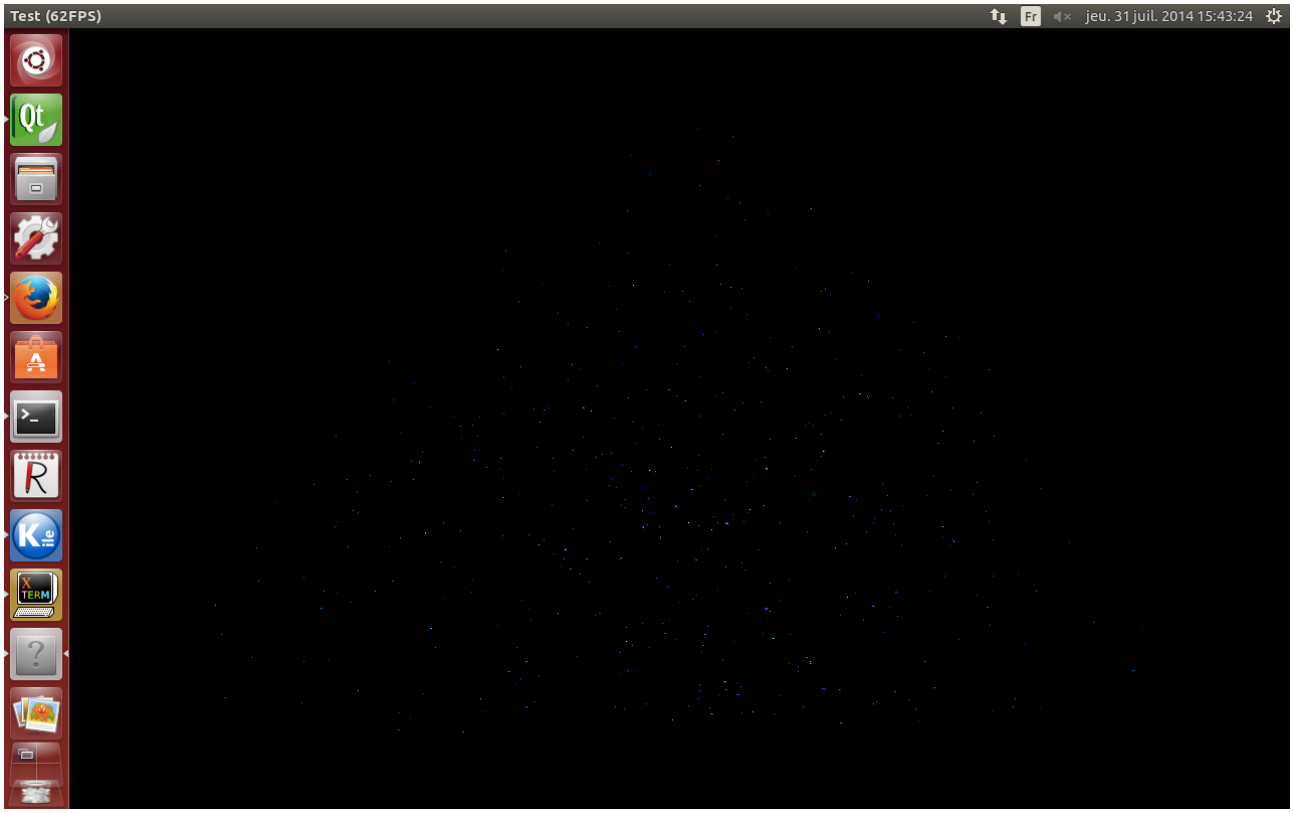

FIGURE 19 – Simulation en vue normale

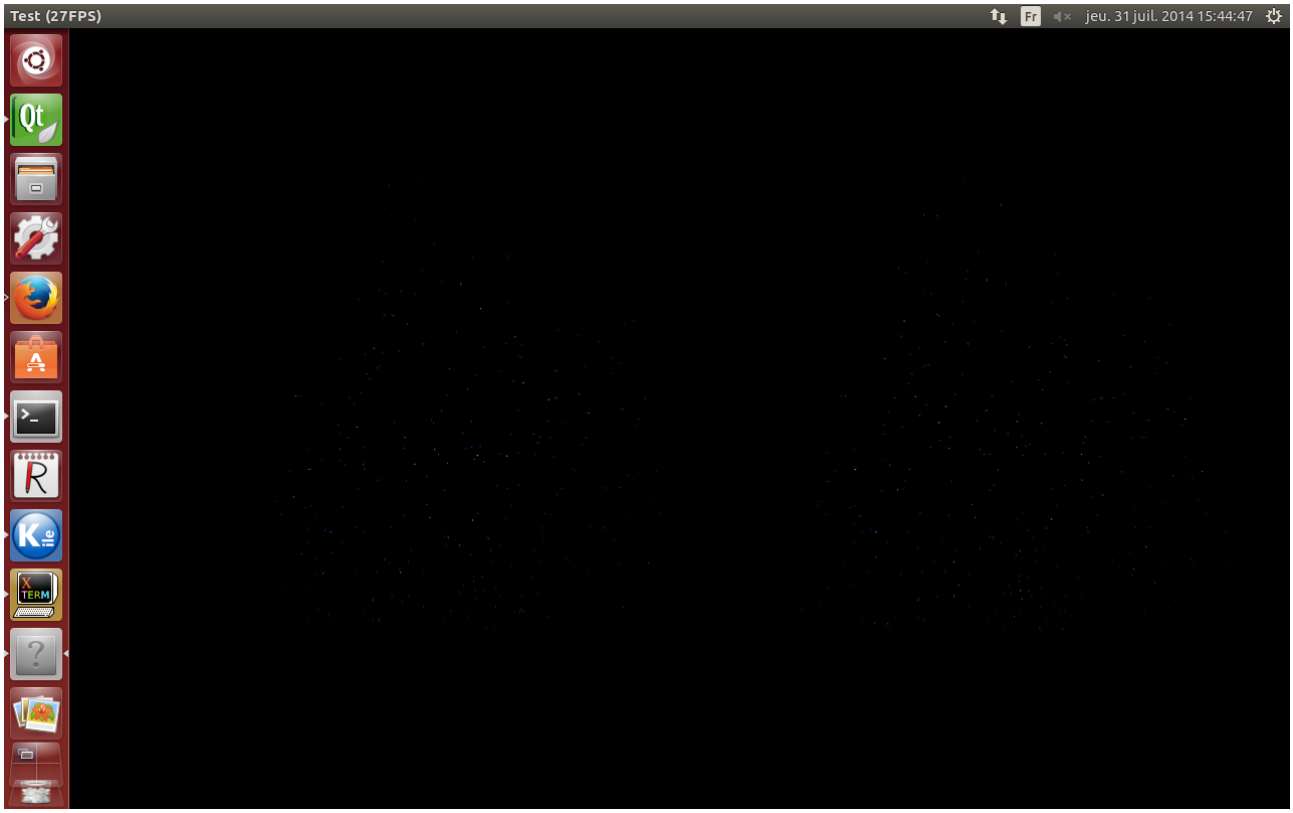

FIGURE 20 – Simulation en vue normale

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();
```

```
48
49      glm::vec3 dAngles() const;
50  };
51
52
53  template<class T>
54  /**
55   * @brief The Oculus templated class
56   * @details It is a singleton to avoid initializing/releasing the Oculus SDK
        multiple times.
57   * The template argument is the type of the OpeGL scene we render.
58   */
59  class Oculus: public GenericOculus
60  {
61  public:
62      /**
63       * @brief Constructor
64       * @details Initializes the Oculus SDK, creates a debug Oculus Rift if none
        is connected, and starts the sensors.
65       * @param scene The OpenGL scene that contains the objects render
66       */
67      Oculus(T & scene):
68          scene_ {scene},
69          textureId_ {0},
70          FBOId_ {0},
71          depthBufferId_ {0},
72          hmd_ {0},
73          windowSize_ {0, 0},
74          textureSizeLeft_ {0, 0},
75          textureSizeRight_ {0, 0},
76          textureSize_ {0, 0},
77          angles_ {0, 0, 0},
78          dAngles_ {0, 0, 0},
79          distortionCaps_ {0},
80          usingDebugHmd_ {false},
81          multisampleEnabled_ {false}
82      {
83          //Oculus is a singleton and cannot be instanciated twice
84          assert(!alreadyCreated);
85          logger->debug(logger->get() << "Oculus constructor" );
86
87          ovr_Initialize();
88
89          hmd_ = ovrHmd_Create(0);
90
91          if(!hmd_)
92          {
93              hmd_ = ovrHmd_CreateDebug(ovrHmd_DK1);
94              usingDebugHmd_ = true;
95
96              //Cannot create the debug hmd
97              assert(hmd_);
98
99              logger->debug(logger->get() << "Using the debug hmd");
100         }
101
102         ovrHmd_GetDesc(hmd_, &hmdDesc_);
```

```
103
104         computeSizes();
105
106         distortionCaps_ = ovrDistortionCap_Chromatic | ovrDistortionCap_TimeWarp
    ;
107
108         eyeFov_[0] = hmdDesc_.DefaultEyeFov[0];
109         eyeFov_[1] = hmdDesc_.DefaultEyeFov[1];
110
111         setOpenGLState();
112         initFBO();
113         initTexture();
114         initDepthBuffer();
115
116         computeSizes();
117         setCfg();
118         setEyeTexture();
119         ovrBool configurationRes = ovrHmd_ConfigureRendering(hmd_, &cfg_.Config,
     distortionCaps_, eyeFov_, eyeRenderDesc_);
120         //Cannot configure OVR rendering
121         assert(configurationRes);
122
123         ovrHmd_StartSensor(hmd_, ovrSensorCap_Orientation |
    ovrSensorCap_YawCorrection | ovrSensorCap_Position, ovrSensorCap_Orientation)
    ;
124
125         Oculus::alreadyCreated = true;
126     }
127
128     /**
129      * @brief Destructor
130      * @details Releases the Oculus SDK and the OpenGL resources required for
    the Oculus rendering
131      */
132     ~Oculus()
133     {
134         logger->debug(logger->get() << "Oculus destructor");
135         glDeleteFramebuffers(1, &FBOId_);
136         glDeleteTextures(1, &textureId_);
137         glDeleteRenderbuffers(1, &depthBufferId_);
138
139         ovrHmd_Destroy(hmd_);
140
141         ovr_Shutdown();
142
143         Oculus::alreadyCreated = false;
144     }
145
146     /**
147      * @brief Renders the OpenGL scene with the Oculus effects
148      */
149     void render()
150     {
151         glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0);
152         glBindBuffer(GL_ARRAY_BUFFER, 0);
153         glUseProgram(0);
154
```

```
155         frameTiming_ = ovrHmd_BeginFrame(hmd_, 0);
156
157         // Bind the FBO...
158         glBindFramebuffer(GL_FRAMEBUFFER, FBOId_);
159         // Clear...
160         glClearColor(0, 0, 0, 1);
161         glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
162
163         getInput();
164
165         ovrPosef eyeRenderPose[2];
166
167         for (int eyeIndex = 0; eyeIndex < ovrEye_Count; eyeIndex++)
168         {
169             ovrEyeType eye = hmdDesc_.EyeRenderOrder[eyeIndex];
170             eyeRenderPose[eye] = ovrHmd_BeginEyeRender(hmd_, eye);
171
172             glViewport(eyeTexture_[eye].OGL.Header.RenderViewport.Pos.x,
173                        eyeTexture_[eye].OGL.Header.RenderViewport.Pos.y,
174                        eyeTexture_[eye].OGL.Header.RenderViewport.Size.w,
175                        eyeTexture_[eye].OGL.Header.RenderViewport.Size.h
176                        );
177
178             // Get Projection and ModelView matrici from the device...
179             OVR::Matrix4f MV = OVR::Matrix4f::Translation(eyeRenderDesc_[eye].
    ViewAdjust)
180                     * OVR::Matrix4f(OVR::Quatf(eyeRenderPose[eye].Orientation).
    Inverted());
181
182             OVR::Matrix4f Proj = OVR::Matrix4f(ovrMatrix4f_Projection(
    eyeRenderDesc_[eye].Fov, 0.01f, 10000.0f, true));
183
184             glm::mat4 glmMV = Utils::ovr2glmMat(MV.Transposed());
185
186             glm::mat4 glmProj = Utils::ovr2glmMat(Proj.Transposed());
187
188             scene_.render(glmMV, glmProj);
189             Utils::GLGetError();
190
191             ovrHmd_EndEyeRender(hmd_, eye, eyeRenderPose[eye], &eyeTexture_[eye
    ].Texture);
192         }
193
194         ovrHmd_EndFrame(hmd_);
195     }
196
197     /**
198      * @brief Tells if we are using a debug Oculus Rift
199      * @return true if no Oculus Rift is connected and we had to create a debug
    one, else false
200      */
201     bool isUsingDebugHmd()
202     {
203         return usingDebugHmd_;
204     }
205
206     /**
```

```
207         * @brief Tells if the Oculus Rift the moving
208         * @details It compares the current angular position with the previous
    angular position
209         * @return true if the Oculus Rift if moving, else false
210         */
211        using GenericOculus::isMoving;
212        bool isMoving() const
213        {
214            bool res = false;
215            for(int i=0; i < 3; i++)
216            {
217                res = res && Utils::isEqual(angles_[i], dAngles_[i]);
218            }
219            return !res;
220        }
221
222        glm::vec3 angles() const
223        {
224            return angles_;
225        }
226
227        void setAngles(const glm::vec3 &angles)
228        {
229            angles_ = angles;
230        }
231
232        /**
233         * @brief Retrieves the values from the Oculus Rift sensors
234         * @details It gets the current angular position from the sensors and the
    prediction tool, and stores the old angular position.
235         * @warning The angles from the sensors are in radians and OpenGL expects
    angles in degrees, hence the required conversion
236         * @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.
237         */
238        void getInput()
239        {
240            glm::vec3 oldAngles = angles_;
241
242            sensorState_ = ovrHmd_GetSensorState(hmd_, frameTiming_.
    ScanoutMidpointSeconds);
243
244            if(sensorState_.StatusFlags & (ovrStatus_OrientationTracked |
    ovrStatus_PositionTracked))
245            {
246                ovrPosef pose = sensorState_.Predicted.Pose;
247                OVR::Quatf quat = pose.Orientation;
248
249                quat.GetEulerAngles<OVR::Axis_Y, OVR::Axis_X, OVR::Axis_Z>(&angles_.
    x, &angles_.y, &angles_.z);
250
251                dAngles_ = angles_ - oldAngles;
252
253                logger->debug(logger->get() << "Angles: "
254                              << OVR::RadToDegree(angles_[0]) << ", "
255                              << OVR::RadToDegree(angles_[1]) << ", "
256                              << OVR::RadToDegree(angles_[1]) << " degrees");
```

```
257
258             logger->debug(logger->get() << "Angles: "
259                     << angles_[0] << ", "
260                     << angles_[1] << ", "
261                     << angles_[1] << " rad");
262
263             logger->debug(logger->get() << "DAngles: "
264                     << OVR::RadToDegree(dAngles_[0]) << ", "
265                     << OVR::RadToDegree(dAngles_[1]) << ", "
266                     << OVR::RadToDegree(dAngles_[1]) << " degrees");
267         }
268         else
269         {
270             logger->debug(logger->get() << "No input data (using debug hmd)");
271         }
272     }
273
274 protected:
275     /**
276      * @brief Creates the OpenGL texture required for the Oculus rendering
277      * @details The Oculus rendering makes under the hood a double (for each eye
    ) render to texture of the scene and then
278      * displays this texture to the screen, hence the big size of the texture.
279      */
280     void initTexture()
281     {
282         // The texture we're going to render to...
283         glGenTextures(1, &textureId_);
284         // "Bind" the newly created texture : all future texture functions will
    modify this texture...
285         glBindTexture(GL_TEXTURE_2D, textureId_);
286         // Give an empty image to OpenGL (the last "0")
287         glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, textureSize_.w, textureSize_.h,
    0, GL_RGBA, GL_UNSIGNED_BYTE, 0);
288         // Linear filtering...
289         glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
290         glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
291
292         Utils::GLGetError();
293     }
294
295     /**
296      * @brief Creates the Frame Buffer Object needed for the Oculus rendering
297      * @details The Oculus rendering uses this FBO to send the texture to the
    graphic card
298      */
299     void initFBO()
300     {
301         // We will do some offscreen rendering, setup FBO...
302         assert(textureSize_.w != 0);
303         assert(textureSize_.h != 0);
304
305         glGenFramebuffers(1, &FBOId_);
306         Utils::GLGetError();
307         //Cannot create the FBO
308         assert(FBOId_ != 0);
309
```

```
310         glBindFramebuffer(GL_FRAMEBUFFER, FBOId_);
311         Utils::GLGetError();
312     }
313 
314     /**
315      * @brief Creates the depth buffer needed for the Oculus rendering
316      */
317     void initDepthBuffer()
318     {
319         glGenRenderbuffers(1, &depthBufferId_);
320         Utils::GLGetError();
321         //Cannot create the depth buffer
322         assert(depthBufferId_ != 0);
323 
324         glBindRenderbuffer(GL_RENDERBUFFER, depthBufferId_);
325         Utils::GLGetError();
326 
327         glRenderbufferStorage(GL_RENDERBUFFER, GL_DEPTH_COMPONENT, textureSize_.
    w, textureSize_.h);
328         Utils::GLGetError();
329 
330         glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT,
    GL_RENDERBUFFER, depthBufferId_);
331         Utils::GLGetError();
332 
333         // Set the texture as our colour attachment #0...
334         glFramebufferTexture(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, textureId_,
    0);
335         Utils::GLGetError();
336 
337         // Set the list of draw buffers...
338         GLenum GLDrawBuffers[1] = { GL_COLOR_ATTACHMENT0 };
339 
340         glDrawBuffers(1, GLDrawBuffers); // "1" is the size of DrawBuffers
341         Utils::GLGetError();
342 
343         // Check if everything is OK...
344         GLenum check = glCheckFramebufferStatus(GL_DRAW_FRAMEBUFFER);
345 
346         //There is a problem with the FBO
347         assert(check == GL_FRAMEBUFFER_COMPLETE);
348 
349         // Unbind...
350         glBindRenderbuffer(GL_RENDERBUFFER, 0);
351         glBindTexture(GL_TEXTURE_2D, 0);
352         glBindFramebuffer(GL_FRAMEBUFFER, 0);
353         Utils::GLGetError();
354 
355     }
356 
357     /**
358      * @brief Sets some OpenGL states to adequate values for the Oculus
    rendering
359      * @warning The multisample value does not seem the be taken into account by
     the Oculus SDK as of yet
360      *  and the Oculus rendering seems unchanged
361      */
```

```
362     void setOpenGLState()
363     {
364         glDisable(GL_TEXTURE_2D);
365         glEnable(GL_DEPTH_TEST);
366         if(multisampleEnabled_)
367         {
368             glEnable(GL_MULTISAMPLE);
369         }
370     }
371
372     /**
373      * @brief Sets the Oculus SDK configuration to adequate values for the
        Oculus rendering
374      * @warning The Windows and OSX modes have not been tested but should work
        just fine
375      */
376     void setCfg()
377     {
378         cfg_.OGL.Header.API = ovrRenderAPI_OpenGL;
379         cfg_.OGL.Header.Multisample = multisampleEnabled_;
380         cfg_.OGL.Header.RTSize.w = windowSize_.w;
381         cfg_.OGL.Header.RTSize.h = windowSize_.h;
382
383         SDL_SysWMinfo info;
384         SDL_VERSION(&info.version);
385         SDL_bool infoRes = SDL_GetWindowWMInfo(scene_.window(), &info);
386         //Cannot retrieve SDL window info
387         assert(infoRes == SDL_TRUE);
388
389         #if defined(OVR_OS_WIN32)
390             cfg.OGL.Window = info.info.win.window;
391         #elif defined (OVR_OS_MAC)
392             cfg.OGL.Window = info.info.cocoa.window
393         #elif defined(OVR_OS_LINUX)
394             cfg_.OGL.Win = info.info.x11.window;
395             cfg_.OGL.Disp = info.info.x11.display;
396         #endif
397     }
398
399     /**
400      * @brief Sets the Oculus SDK texture configuration to adequate values for
        the Oculus rendering
401      */
402     void setEyeTexture()
403     {
404         eyeTexture_[0].OGL.Header.API = ovrRenderAPI_OpenGL;
405         eyeTexture_[0].OGL.Header.TextureSize.w = textureSize_.w;
406         eyeTexture_[0].OGL.Header.TextureSize.h = textureSize_.h;
407         eyeTexture_[0].OGL.Header.RenderViewport.Pos.x = 0;
408         eyeTexture_[0].OGL.Header.RenderViewport.Pos.y = 0;
409         eyeTexture_[0].OGL.Header.RenderViewport.Size.h = textureSize_.h;
410         eyeTexture_[0].OGL.Header.RenderViewport.Size.w = textureSize_.w/2;
411         eyeTexture_[0].OGL.TexId = textureId_;
412
413         // Right eye the same, except for the x-position in the texture...
414         eyeTexture_[1] = eyeTexture_[0];
415         eyeTexture_[1].OGL.Header.RenderViewport.Pos.x = (textureSize_.w + 1) /
```

```
    2;
416
417     }
418
419     /**
420      * @brief Computes the texture size
421      * @details This computation depends on the window dimensions. The optimal
    dimensions are 1280*800, which is the Oculus
422      * resolution
423      * @warning Other resolutions and window resizing have not been tested but
    should work just fine
424      */
425     void computeSizes()
426     {
427         windowSize_.w = scene_.windowWidth();
428         windowSize_.h = scene_.windowHeight();
429
430         logger->debug(logger->get() << "Fov: " << Utils::radToDegree(2 * atan(
    hmdDesc_.DefaultEyeFov[0].UpTan)));
431
432         textureSizeLeft_ = ovrHmd_GetFovTextureSize(hmd_, ovrEye_Left, hmdDesc_.
    DefaultEyeFov[0], 1.0f);
433         textureSizeRight_ = ovrHmd_GetFovTextureSize(hmd_, ovrEye_Right,
    hmdDesc_.DefaultEyeFov[1], 1.0f);
434         textureSize_.w = textureSizeLeft_.w + textureSizeRight_.w;
435         textureSize_.h = (textureSizeLeft_.h > textureSizeRight_.h ?
    textureSizeLeft_.h : textureSizeRight_.h);
436
437     }
438
439     /**
440      * @brief Boolean that shows whether or not an instance has already been
    created
441      * @details Part of the Singleton Pattern
442      */
443     static bool alreadyCreated;
444
445     /**
446      * @brief The generic OpenGL scene
447      * @details Oculus is a templated class and its only argument is the type of
     \a scene. The only requirement is that scene
448      * has a method \a render, wich takes as argument the modelview matrix and
    the projection matrix.
449      */
450     T & scene_;
451
452     //GL
453     /**
454      * @brief The id of the OpenGL texture used in the Oculus rendering
455      */
456     GLuint textureId_;
457
458     /**
459      * @brief The id of the OpenGL Frame Buffer Object used in the Oculus
    rendering
460      */
461     GLuint FBOId_;
```

```
462
463     /**
464      * @brief The id of the OpenGL depth buffer used in the Oculus rendering
465      */
466     GLuint depthBufferId_;
467
468     //OVR
469     /**
470      * @brief The Oculus Rift
471      * @details If no Oculus Rift is connected, a debug one is created. The last
         does not have proper sensors.
472      */
473     ovrHmd hmd_;
474
475     /**
476      * @brief The description of the Oculus Rift
477      * @details Contains lots of values like inter-pupillary distance,
       resolution, etc
478      */
479     ovrHmdDesc hmdDesc_;
480
481     /**
482      * @brief The description of each eye
483      * @details Contains lots of values like dimensions, wether it is the left
       or right eye, etc.
484      */
485     ovrEyeRenderDesc eyeRenderDesc_[2];
486
487     /**
488      * @brief The texture of each eye
489      */
490     ovrGLTexture eyeTexture_[2];
491
492     /**
493      * @brief The Field of View of each eye
494      */
495     ovrFovPort eyeFov_[2];
496
497     /**
498      * @brief The configuration for the OpenGL Oculus rendering
499      */
500     ovrGLConfig cfg_;
501
502     /**
503      * @brief The dimensions of the window
504      */
505     ovrSizei windowSize_;
506
507     /**
508      * @brief The dimensions of the texture that the left eye can see
509      */
510     ovrSizei textureSizeLeft_;
511
512     /**
513      * @brief The dimensions of the texture that the right eye can see
514      */
515     ovrSizei textureSizeRight_;
```

```
516
517     /**
518      * @brief The dimensions of the texture overall
519      */
520     ovrSizei textureSize_;
521
522     /**
523      * @brief Time variable used by the sensor and the predication tool
524      */
525     ovrFrameTiming frameTiming_;
526
527     /**
528      * @brief The Oculus Rift sensors
529      */
530     ovrSensorState sensorState_;
531
532     /**
533      * @brief The Oculus Rift angular position
534      */
535     glm::vec3 angles_;
536
537     /**
538      * @brief The Oculus Rift angular position variation
539      */
540     glm::vec3 dAngles_;
541
542     /**
543      * @brief Flag used for the Oculus rendering configuration
544      */
545     int distortionCaps_;
546
547     /**
548      * @brief Boolean indicating if we are using a debug Oculus Rift
549      */
550     bool usingDebugHmd_;
551
552     /**
553      * @brief Boolean indicating if the Oculus rendering is multisampled
554      * @warning The Oculus SDK does not seem to take this variable into account
    as of yet
555      */
556     bool multisampleEnabled_;
557 };
558
559 template<class T>
560 bool Oculus<T>::alreadyCreated = false;
561
562 /**
563  * @brief The NullOculus class
564  * @details Part of the Null object pattern
565  */
566 class NullOculus: public GenericOculus
567 {
568 public:
569     NullOculus();
570
571     ~NullOculus();
```

```
572
573     void render() {}
574 };
575
576 /**
577  * @brief nullOculus
578  * @details Implements the null object pattern
579  */
580 extern std::unique_ptr<NullOculus> nullOculus;
581
582 #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