Table des matières

Résumé ii

Remerciements iii

Table des matières iv

Liste des tableaux vii

Liste des figures viii

Liste des symboles x

Chapitre 1 - Introduction 1

1.1 Contexte 1

1.2 Problématique 2

1.3 Objectifs et méthodologie 4

1.4 Structure du mémoire 5

Chapitre 2 - Réalité virtuelle 6

2.1 Composante de la Réalité virtuelle 6

2.2 Casques de Réalité virtuelle 10

2.2.1 Vision 11

2.2.2 Capteurs 15

2.3 Logiciel 18

2.3.1 Moteur graphique 18

2.3.2 Moteur physique 21

2.4 Conclusion 25

Chapitre 3 - Conception logiciel/matérielle de RV 27

3.1 Architecture du programme 27

3.1.1 Interface graphique 29

3.1.2 Scène 30

3.1.3 Caméra 33

3.1.4 Entrées 36

3.1.5 Importation de modèles 36

3.1.6 Textures 40

3.1.7 Mémoire tampon de trame (Frame Buffer) 43

3.1.8 Nuanceur (Shader) 45

3.2 Architecture matérielle 56

3.2.1 OpenCL 56

3.2.2 Implémentation d'un moteur de collision 59

3.3 Conclusion 65

Chapitre 4 - Application de la RV 66

4.1 Introduction de l'environnement matériel 68

4.2 Performances logicielles 71

4.2.1 Performance sur différentes plateformes 71

4.2.2 Comparaison ancien et nouveau moteur 73

4.3 Résultats de l'implémentation matérielle 74

4.4 Application aux TCA 75

4.5 Conclusion 78

Chapitre 5 - Conclusion générale 80

Références 82

Annexe A – Shader de distortion 84

Vertex Shader 84

Fragment Shader 84

Liste des tableaux

Tableau 3.1 Dimensions des axes de collision [18] 64

Tableau 4.1 Comparaison des performances sur différentes plateformes..... 72

Tableau 4.2 Comparaison des temps de calcul avec et sans OpenCL 74

Liste des figures

Figure 2.1 Architecture d'un programme de Réalité Virtuelle 7

Figure 2.2 Oculus Development Kit 2 10

Figure 2.3 Distorsion de type *pincushion* 12

Figure 2.4 Distorsion de type *barrel* 13

Figure 2.5 Aberration chromatique 14

Figure 2.6 Exemple d'aberration chromatique 14

Figure 2.7 Axes et rotations du casque 17

Figure 2.8 Fonctions cœur OpenGL sur le processeur graphique 20

Figure 2.9 Premières étapes de détection de collision [19] 23

Figure 2.10 Deuxième étape de détection de collision [19] 24

Figure 2.11 différents cas de boites de collision en 2D [20] 25

Figure 3.1 Architecture des classes du logiciel 29

Figure 3.2 Interface utilisateur de la RV 30

Figure 3.3 Utilisation des matrices d'affichage : traduite de [21] 32

Figure 3.4 Exemple de triangulation 38

Figure 3.5 Repères des textures SDL et OpenGL respectivement [22] 42

Figure 3.6 Exemple de *stencil buffer* [23] 44

Figure 3.7 Exemple de rastérisation 50

Figure 3.8 Exemple d'interpolation de couleur [22] 51

Figure 3.9 Distorsion des lentilles de l'Oculus 52

Figure 3.10 Correction de déformation envoyée à l'Oculus 53

Figure 3.11 Rendu visualisé dans l'Oculus une fois l'image corrigée 54

Figure 3.12 Hello World OpenCL par Altera [24] 58

Figure 3.13 Représentation des OBB 59

Figure 3.14 Exemple d'OBB 62

Figure 4.1 Silhouette regardée dans le temps 77

Figure 4.2 Répartition du temps par silhouette 78

Liste des symboles

- RV : Réalité virtuelle
- OpenGL : Open Graphic Library
- OpenCL : Open Compute Library
- OS : Operating System
- Assimp : open ASSet IMPort library
- SDL : Simple Directmedia Layer
- FPS : Frames per Second
- VAO : Vertex Array Object
- VBO : Vertex Buffer Object
- CPU : Central processing unit
- GPU : Graphics Processing Unit
- FPGA : Field Programmable Gate Array
- PCA : Principal Component Analysis
- Vsync : Vertical synchronisation

Chapitre 1 - Introduction

1.1 Contexte

Nos sociétés occidentales et industrielles font face à un réel problème de santé publique qui tend à s'accroître et à concerner des populations de plus en plus jeunes : les troubles du comportement alimentaire (TCA) tels que l'anorexie mentale, la boulimie et les troubles alimentaires non spécifiés[1] [1]. Conscientes du problème, ces institutions montrent récemment, par leurs initiatives, leur volonté de diminuer la prévalence des TCA en tentant d'agir au niveau du traitement des TCA. Or, l'adoption de lois ou la rédaction de recommandations ne constituent pas à elles seules des moyens efficaces et opérationnalisables pour enrayer « la propagation » des TCA dans nos sociétés. Le développement d'instruments de mesure des facteurs de risques et de traitement des TCA s'y prête davantage; en permettant d'outiller concrètement ces professionnels. La thérapie cognitivo-comportementale est aujourd'hui reconnue pour être une des approches psychothérapeutiques des TCA la plus efficace. Fondées sur les théories de l'apprentissage et sur les théories cognitives du fonctionnement psychologique, les techniques cognitivo-comportementales permettent de réduire significativement une des caractéristiques cliniques centrales des TCA : les troubles de l'image du corps [2]. Les troubles de l'image du corps

[1] Parmi les troubles alimentaires non spécifiés, l'hyperphagie boulimique conduisant au surpoids ou à l'obésité est la plus fréquente dans nos sociétés.

chez les populations TCA, concernent principalement les composantes perceptuelles (apparence physique perçue ou la distorsion corporelle) et cognitive-affectives (insatisfaction corporelle). La distorsion est l'incapacité à percevoir la taille et la forme de son corps. L'insatisfaction corporelle correspond au degré d'acceptation d'une personne vis-à-vis de la taille et de la forme de son corps. Généralement, ce degré d'acceptation se mesure en calculant l'écart entre le corps perçu et le corps désiré parmi un continuum de silhouettes.

Durant les dix dernières années, le succès remporté par l'intégration de la réalité virtuelle dans les thérapies cognitivo-comportementales de certains troubles mentaux (notamment les phobies), a sensibilisé les spécialistes des TCA. La réalité virtuelle se définit comme un ensemble de logiciels et matériels permettant de simuler de manière réaliste une interaction avec des objets virtuels qui sont des modélisations informatiques d'objets réels [3]-[5]. Elle est présente dans différents domaines tels que le divertissement, la simulation de véhicule et la formation. Elle occupe une place de plus en plus grande dans le domaine de la santé grâce aux avancées technologiques et le grand potentiel en recherche expérimentale.

1.2 Problématique

Appliqués aux TCA, les environnements numériques, représentant des problématiques en lien avec ces troubles (continuum de silhouettes, cuisine interactive, etc.), s'adaptent en temps réel aux actions de l'utilisateur (individus dits à risque de présenter des TCA ou patients TCA) évoluant dans cette représentation. Dans le cas des thérapies in virtuo (versus in vivo) des TCA, plusieurs avantages d'utilisation de la réalité virtuelle apparaissent comme l'augmentation : (a) de la standardisation des traitements pour une approche expérimentale, (b) de la motivation et de l'engagement des jeunes individus dans la thérapie, (c) du contrôle de l'intervenant sur l'environnement, (d) des stimuli disponibles, (e) des possibilités

d'entraînement individuel, (f) du degré de confidentialité, et (g) de la sécurité et de la flexibilité de l'environnement [4]. Néanmoins, certains inconvénients sont à noter dont l'apparition de « cybermalaises » (vertige, fatigue oculaire, etc.) éventuels et transitoires, les coûts relatifs à l'acquisition des environnements virtuels, le temps pour se familiariser à l'utilisation de cette nouvelle technologie et les exigences de calcul qui pèsent sur les ordinateurs [7]-[13].

Le dispositif de réalité virtuelle actuellement utilisé dans le cadre des travaux de recherche du LoriCorps (LabOratoire de Recherche Interdisciplinaire sur les troubles du COmportement alimentaire en lien avec la Réalité virtuelle et la Pratique phySique) est basé sur un ordinateur. Ce dernier prend en charge l'ensemble des calculs nécessaires pour offrir un environnement immersif pour la clientèle cible. Le système actuel répond aux besoins d'une partie de la recherche sur les TCA. Cependant, certaines limites de ce système pourraient constituer un frein à la collecte de données probantes dans différentes situations reliées au TCA et à d'autres troubles psychologiques d'intérêt. Parmi les inconvénients du système actuel :

- La mise à niveau des environnements virtuels est grandement limitée par la puissance de calcul de l'ordinateur. Une perte de réalisme en immersion et une augmentation des cybermalaises sont observées dès que des détails supplémentaires sont ajoutés selon les besoins en scénarios des recherches pour immerger les sujets.
- La mise à niveau du système est complexe dû au fait que celui-ci utilise un logiciel propriétaire. Toute mise à niveau doit donc passer par le créateur du logiciel, ce qui restreint grandement les modifications sur celui-ci.

- La puissance de calcul restreinte des ordinateurs limite le nombre de scénarios à développer pour répondre aux différentes situations nécessaires pour bien représenter les milieux tests afin d'inclure tous les sujets.
- Le dispositif actuel est encombrant. Ceci rend sa mobilité problématique pour effectuer des tests terrain (ex. en milieu clinique comme les hôpitaux, les cliniques de soins, etc

1.3 Objectifs et méthodologie

Dans le but de répondre aux problématiques présentées dans la section précédente plusieurs objectifs généraux seront posés. Le premier objectif est d'utiliser une architecture à plusieurs types de processeur permettant d'optimiser les performances du logiciel et ainsi retrouver le réalisme et l'immersion des simulations. Le second objectif est de proposer un logiciel ouvert permettant de reproduire les résultats du système actuellement utilisé. Le dernier objectif est de proposer une architecture embarquée utilisant entre autre un FPGA permettant de rendre le dispositif plus mobile.

Comme les objectifs présentés précédemment ont beaucoup d'envergure, le but de ce projet sera d'explorer les différentes solutions possibles permettant de les atteindre. La méthodologie sera donc la suivante :

- La problématique étant grandement liée à l'optimisation des calculs, la première étape sera de se familiariser avec la structure des programmes de réalité virtuelle (RV). Cela permettra de repérer où les optimisations sont possibles et ont de l'intérêt.

- Dans le but d'utiliser les forces des différents types de processeurs une analyse des forces et faiblesses de chacun de ceux-ci sera faite. Les types de processeurs considérés ici seront le CPU, GPU, et FPGA.
- Proposer une division des tâches reliées au moteur de réalité virtuelle sur ces différents types de processeurs afin de tirer avantage de chacun de ceux-ci tout en tentant d'éliminer les faiblesses reliées à leur utilisation.

1.4 Structure du mémoire

Ce mémoire sera donc divisé en quatre autres chapitres permettant de répondre aux objectifs. Le chapitre 2 fera l'état de l'art des moteurs de RV. Le chapitre 3 abordera la conception logicielle et matérielle. Dans ce chapitre sera d'abord »»»»»abordée la conception de moteur de RV conventionnelle et par la suite le transfert de certaines tâches vers un autre processeur. Le chapitre 4 présentera des résultats et performances du moteur de RV une fois appliqué au traitement des TCA. Finalement, le chapitre 5 présentera la conclusion du projet ainsi que les travaux futurs permettant d'améliorer celui-ci.

Chapitre 2 - Réalité virtuelle

La Réalité Virtuelle est un outil qui s'apparente aux jeux vidéo. La différence majeure est le fait que le dispositif d'affichage est attaché au niveau de la tête dans le cas de la RV afin de suivre tout le mouvement de tête de l'utilisateur. Ce nouvel élément amène une composante immersive due au fait que l'affichage RV couvre tout le champ de vision de l'utilisateur. Ceci permet à la RV d'être employée à d'autres fins telles que certains traitements en psychologie, entraînement militaire en situation de stress, simulation de situations d'urgence et plusieurs autres [3]-[5].

Afin de mieux comprendre la RV, cette section sera divisée en trois parties décrivant plus concrètement les différents aspects reliés à celle-ci. La première partie décrit les composantes nécessaires et leurs fonctions. La deuxième traite des différents casques de RV. Puis, la dernière introduit les implications logicielles pour faire fonctionner celle-ci.

2.1 Composante de la Réalité virtuelle

Cette section a pour but de diviser, sous forme de diagramme bloc, les différentes parties qui constituent un programme de RV. La figure 2.1 montre un schéma bloc simple représentant une architecture possible :

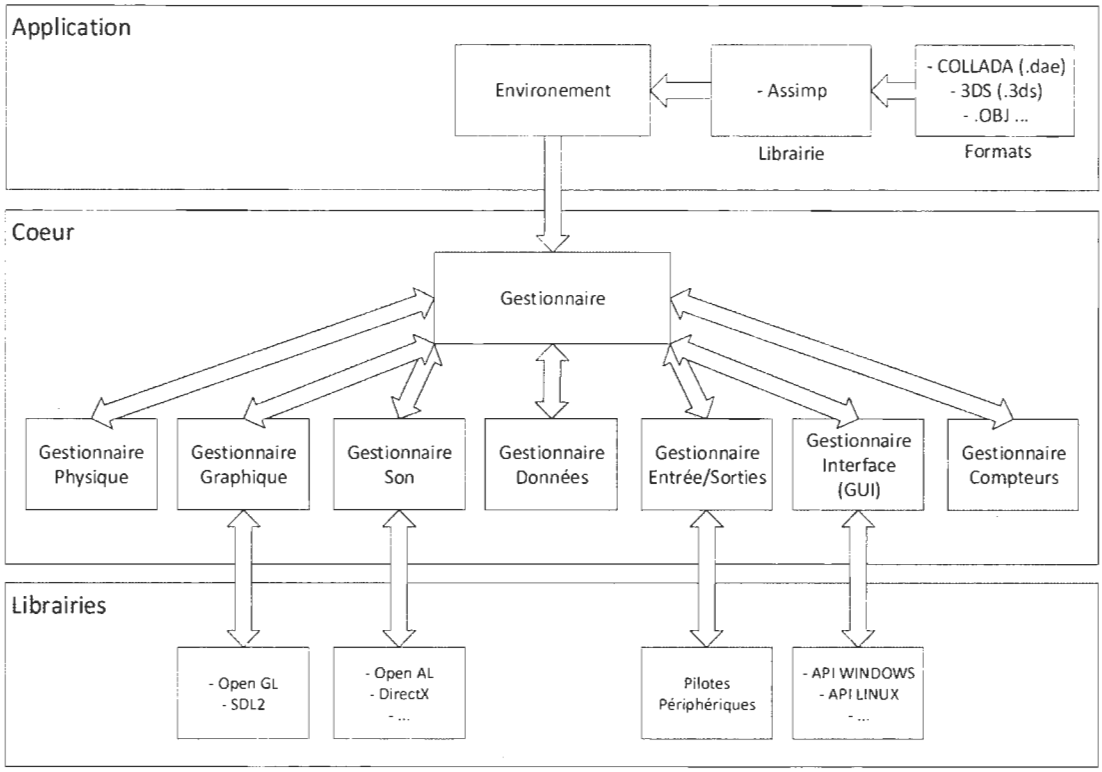

Figure 2.1 Architecture d'un programme de Réalité Virtuelle

Le schéma présenté à la figure 2.1 est divisé en trois sections qui sont Application, Cœur et Librairies. Il s'agit simplement d'une séparation en différentes couches afin de mieux visualiser la structure du programme.

La couche «Application » est celle utilisée par l'utilisateur. C'est à l'intérieur de celle-ci que l'utilisateur choisit les différents paramètres de la simulation. On retrouve, par exemple, le choix du fichier de modèle 3D de la RV que l'on désire visualiser.

La couche «Cœur», comme le nom l'indique, est la partie centrale du programme. Cette partie n'est donc pas visible par l'utilisateur. Elle est nécessaire à la gestion des différents processus à exécuter pour réaliser la simulation. Voici donc une brève description des différents gestionnaires présents dans le schéma bloc. La partie physique permet de calculer tous les phénomènes physiques nécessaires à la simulation. Ils peuvent être très variés ou très

simples en fonction des besoins de la simulation. La partie physique qui sera généralement présente est la détection de collision. Une description détaillée de la détection de collision fera l'objet de la section 2.3.1 du mémoire. Ensuite, il y a la partie graphique. Celle-ci permet de faire tous les calculs nécessaires à l'affichage du modèle 3D à l'écran. Quoique facile à visualiser, cette section est assez complexe et se divise en différentes sous-sections qui seront détaillées davantage dans la section 2.3.2 du mémoire. Ces deux premiers gestionnaires sont les plus complexes et justifient la nécessité de les aborder plus en détail dans leur propre section. Le gestionnaire de son permet de faire jouer les différents sons dans la simulation. Ceux-ci peuvent être des sons statiques ou bien 3D tenant compte de l'atténuation, réverbération, effet doppler et autres. Le gestionnaire de données permet simplement d'enregistrer et de lire certaines informations pertinentes. Celles-ci peuvent être par exemple, des informations sur les habitudes de l'utilisateur qui auront une influence sur la suite de la simulation. Une autre utilité serait l'enregistrement de statistiques sur le comportement de l'utilisateur. Celui-ci permet donc de faire une panoplie de tâches assez variées. Ces deux derniers gestionnaires ne sont pas nécessaires, mais peuvent bonifier l'expérience de l'utilisateur. Le gestionnaire d'entrée/sortie permet de communiquer avec les différents périphériques. Parmi ceux-ci il y a les capteurs et l'écran du casque. Il y aussi plusieurs autres possibilités telles qu'un manche à balai (*joystick*), clavier, souris, caméra, capteur haptique et plusieurs autres. Le gestionnaire d'interface permet simplement de fournir une entrée facile d'utilisation pour l'usager et de traduire ces informations vers des données utiles pour le reste des gestionnaires. Finalement la gestion des compteurs permet d'actualiser les différentes boucles du programme. Celles-ci servent à la gestion des animations, adaptation des vitesses de déplacement en fonction du nombre d'images par seconde et autres.

La couche « Librairie », quant à elle, sert à faire le lien entre la couche « Cœur » et le matériel. À l'intérieur de la figure 2.1 quelques librairies pour une configuration possible sont énoncées. Dans le cas de la partie graphique, il y a OpenGL qui est une possibilité afin de communiquer avec la carte graphique et éventuellement afficher le contenu à l'écran. Il existe aussi DirectX, de Microsoft, qui permet de faire la même chose qu'OpenGL. Pour la partie audio, il y a OpenAL ainsi que DirectX qui permettent de communiquer avec la carte de son et envoyer le tout aux haut-parleurs. La partie pilote périphérique est un peu plus large, car il existe différentes librairies pour les différents périphériques utilisés. Par exemple, un joystick peut être interfacé avec la librairie fournie par celui-ci. Il pourrait aussi l'être avec SDL qui permet de communiquer avec une panoplie de périphériques. Dans le cas des capteurs à l'intérieur du casque, ils disposent de leur propre librairie fournie par le fabricant. Il existe bien sûr, beaucoup d'autres exemples de périphériques qui pourraient être mis en évidence ici. En fait, cette partie renferme presque tous les périphériques qui ne font pas partie du graphique ou de l'audio. Finalement, il y a la partie interface utilisateur qui nécessite l'API Windows, ou autre, dépendamment du système d'exploitation utilisé. Il existe d'autres librairies de plus haut niveau qui sont basées sur l'API du système d'exploitation utilisé. Ceci n'est pas une liste exhaustive des différentes librairies, mais des exemples afin de bien illustrer l'importance de cette couche.

En résumé la couche application est celle qui fait la communication avec l'utilisateur. La couche cœur fait la gestion des différents processus dans le programme. Puis la couche librairie gère la communication avec le matériel. Le schéma de la figure 2.1 est bien sûr une des solutions possibles, mais d'autres solutions vont ressembler à celle-ci.

2.2 Casques de Réalité virtuelle

Le casque de RV est l'élément central de celle-ci. C'est à celui-ci que l'utilisateur va s'attarder sans porter attention au reste. Il s'agit donc d'un élément important lors du choix du matériel dans le développement de RV. Il n'existe pas pour l'instant une très grande variété de casques de RV, mais il y en a tout de même plusieurs sur le marché. Le principal est l'Oculus Rift visible à la figure 2.2, car il est un des premiers à s'être fait connaitre.

Figure 2.2 Oculus Development Kit 2

Il existe plusieurs versions de celui-ci dont deux kits de développement. Il y a aussi le casque OSVR fabriqué par Razer. Il y aussi d'autres versions qui nécessitent un cellulaire comme écran à l'intérieur du casque et une version en cours de développement par Playstation de Sony qui se nomme Projet Morpheus. Il s'agit ici des principaux casques qui ont été explorés, bien qu'il y en ait d'autres.

Les sous-sections qui suivent donnent plus d'informations sur le fonctionnement des différents casques en la divisant en deux parties. La première aborde tout ce qui est relié à

l'affichage d'image pour l'utilisateur. La deuxième traite de tout ce qui est relié aux différents capteurs présents dans les casques de RV.

2.2.1 Vision

La vision est la partie qui permet à l'utilisateur de visualiser le contenue de la RV. Elle contient bien entendu un écran. L'affichage sur cet écran sera divisé en deux afin d'afficher une image de chaque côté. Ceci est nécessaire, car l'écran est tellement proche du visage qu'il faut une image différente pour chacun des yeux. De plus cela permet aux yeux de l'utilisateur de faire une synthèse entre les deux images résultant en une vision en 3 dimensions. Ceci fait en sorte que le contenu apparait comme réellement présent devant l'utilisateur. Il s'agit de la première étape pour une plus grande immersion dans la RV. Par contre cela fonctionne seulement si la gestion des deux images est faite correctement. De plus, une gestion qui n'est pas faite correctement peut résulter en cyber-malaise car le cerveau ne sera pas capable de faire une synthèse correctement.

Puisque l'écran dans le casque est très proche des yeux, il n'est pas possible, sans ajout de lentilles, de bien voir ce qu'il y a à l'écran. Cela est dû au fait que l'écran est à une distance plus petite que le punctum proximum de l'homme. Il s'agit là du point le plus près des yeux à partir duquel l'homme est capable de bien distinguer un objet. Pour corriger cela, des lentilles sont ajoutées entre l'écran et les yeux. Celles-ci vont faire en sorte que l'écran va apparaître comme étant plus loin qu'il ne l'est en réalité. De plus, elles vont faire en sorte que le champ de vision apparait plus grand. Ceci donne l'impression à l'utilisateur que l'écran prend tout son champ de vision. Ce qui résulte en un environnement plus immersif pour lui puisqu'il ne peut voir que l'écran. Cependant, l'utilisation de lentilles entraîne certaines déformations sur l'image perçue. Cela entraîne une diminution de l'immersion. Pour

augmenter l'effet d'immersion, il faut soit modifier les lentilles afin de minimiser la distorsion ou bien annuler la distorsion à l'aide d'une modification du rendu à l'écran. Bien entendu la deuxième demande plus de calcul et ralentit tout le processus d'affichage, mais offre une meilleure correction. Par exemple, dans le casque OSVR, la distorsion est minimisée à l'aide des lentilles, mais elle est quand même perceptible. Dans un autre cas, l'Oculus ne minimise pas la distorsion. Ce qui oblige l'utilisateur à corriger cette distorsion à l'aide du rendu à l'écran et rend celle-ci presque imperceptible.

Il y a deux types de distorsions présentes dues aux lentilles. La première est une aberration géométrique puis la deuxième une aberration chromatique. L'aberration géométrique est une distorsion de type coussinet « pincushion ». Celle-ci est corrigée à l'aide de la distorsion inverse qui est de type barillet « barrel ». Les figures 2.3 et 2.4 offrent un aperçu de ces types d'aberrations sur un quadrillé.

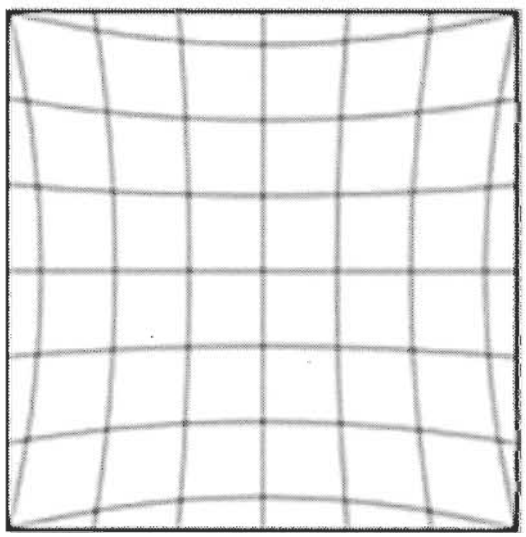

Figure 2.3 Distorsion de type *pincushion*

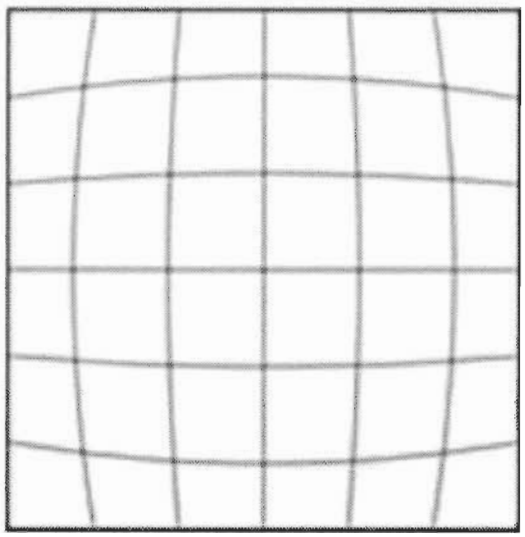

Figure 2.4 Distorsion de type *barrel*

Plus d'information sur la correction de ce type de distorsion sera présentée dans la section 3.1.8. Par la suite, il y a l'aberration chromatique qui a pour effet de séparer les couleurs. Ceci est dû au fait que le coefficient de réfraction est différent pour chacune des longueurs d'ondes. Donc chaque couleur est déviée à un angle différent. Ce qui résulte en une séparation des couleurs du point de vue de l'utilisateur. La figure 2.5 illustre l'effet de la réfraction dans une lentille sur les différentes couleurs de la lumière blanche.

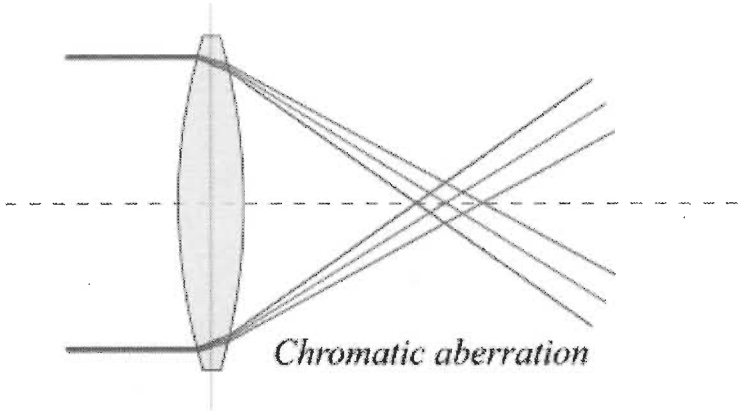

Figure 2.5 Aberration chromatique

Cet effet doit être corrigé, car il peut être assez gênant. Celui-ci nuit à la distinction des différents détails de l'image et peut même dans certains cas changer la perception de l'utilisateur. L'effet de cette aberration peut être vu dans une photo comme celle illustrée à la figure 2.6.

Figure 2.6 Exemple d'aberration chromatique

On remarque que celle-ci a pour effet de rapprocher le bleu vers le centre de l'image et d'éloigner le rouge vers l'extérieur. L'utilisateur perçoit donc clairement une séparation des couleurs qui compose l'image. Ce type d'aberration se corrige au même moment que l'on corrige la distorsion géométrique de type *pincushion*. Plus d'information sur la correction de celle-ci est présentée dans la section 3.1.8.

2.2.2 Capteurs

Les différents casques de réalité virtuelle sont également munis de différents capteurs qui permettent de mesurer les mouvements de la tête. Sans quoi celui-ci serait simplement un dispositif de visionnement 3D au même titre qu'un téléviseur 3D. Ces capteurs sont généralement le gyroscope, accéléromètre et magnétomètre. Habituellement, il y a aussi un thermomètre afin d'améliorer la précision des mesures des autres capteurs. Les derniers casques sont aussi en général expédiés avec une caméra de positionnement.

Puisque la plupart des capteurs ont une sortie qui varie aussi en fonction de la température, les données mesurées par ceux-ci n'est pas totalement fiable. Connaissant le comportement en fonction de la température des différents capteurs, le thermomètre permet de corriger les mesures et ainsi éliminer la dépendance à la température. Ceci est en général fait automatiquement par le SDK du casque et est totalement transparent pour le développeur.

Le premier capteur est le gyroscope. Celui-ci est capable de mesurer les mouvements angulaires sur les trois axes cartésiens du casque. Il permet donc de complètement déterminer l'orientation de l'objet en trois dimensions.

Le deuxième capteur qui est l'accéléromètre permet de mesurer l'accélération dans les différents axes du casque. Celui-ci est donc en mesure de trouver la direction du sol grâce au

fait qu'il mesure l'accélération gravitationnelle. Donc peu importe la position de la tête, il peut retrouver la position du sol. Il retrouve donc un point de référence fixe qui permet de partiellement déterminer l'orientation de la tête. Il permet aussi en théorie de retrouver la position de la tête à l'aide d'une double intégrale qui transforme l'accélération en position.

Cette mesure est bien théorique due au fait qu'une intégrale sur un système bruité entraîne des divergences. Comme un accéléromètre n'est pas totalement sans bruit, on retrouve donc une divergence sur la vitesse après la première intégrale et une divergence encore plus importante sur la position après la deuxième. Cette erreur grandit significativement avec le temps de simulation et rend cette mesure inutilisable sans autres mesures qui permettent de corriger la divergence.

Le troisième capteur est le magnétomètre qui permet de mesurer le champ magnétique sur les différents axes cartésiens. Celui-ci permet donc de mesurer le champ magnétique terrestre. Dans ces conditions, il permet d'agir en tant que boussole. Comme dans le cas de l'accéléromètre, il permet de retrouver un point de référence fixe qui est le nord. Il est donc en mesure de partiellement déterminer l'orientation du casque.

Ces trois capteurs utilisés ensemble permettent de déterminer l'orientation du casque de façon plus robuste que s'ils étaient utilisés individuellement. Ceux-ci sont utilisés par le SDK du casque et retourne les informations sous forme d'un quaternion qui permet de totalement déterminer l'orientation du casque selon les trois angles visibles; comme l'illustre la figure 2.7. Les angles sont le Lacet (Yaw), l'Assiette (Pitch) et l'Inclinaison (Roll). Ceci donne seulement l'information sur l'orientation et non sur la position de la tête, car cette configuration permet seulement de détecter les rotations et non les translations.

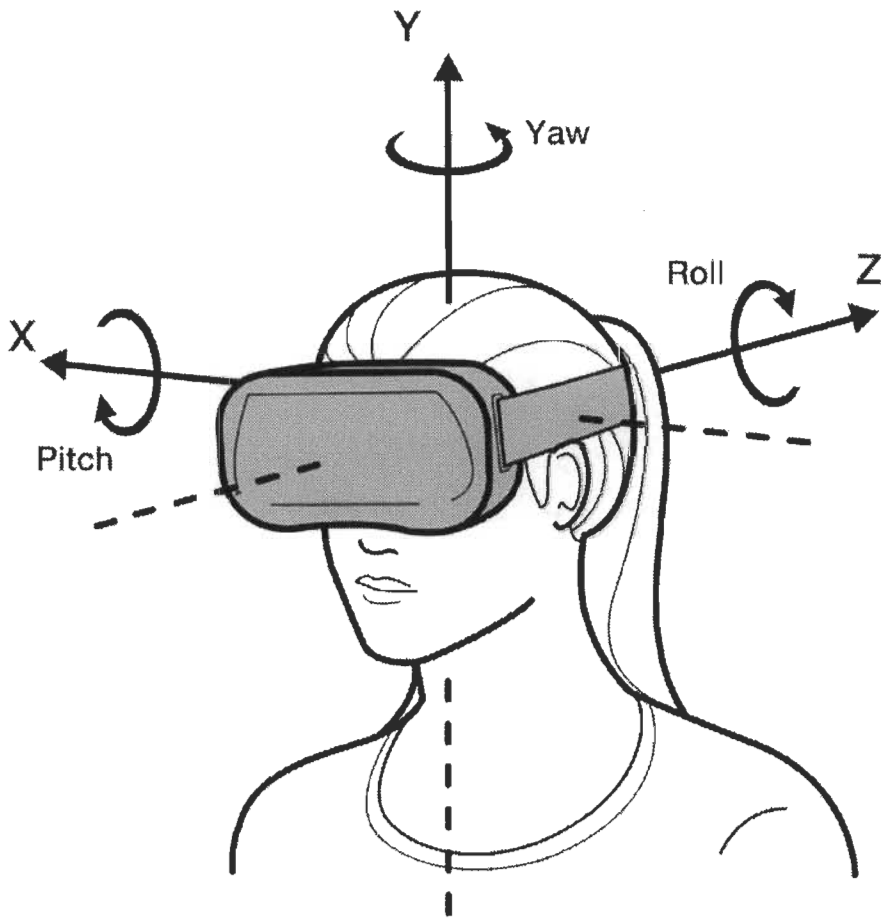

Figure 2.7 Axes et rotations du casque

C'est pour cette raison que les derniers casques viennent avec une caméra de positionnement. Cette caméra est à l'extérieur du casque et est fixée à un point qui lui permet de toujours voir le casque. Celle-ci permet de détecter les points clés du casque à l'aide de traitement d'image. Par la suite, les points détectés dans le plan 2D de la caméra sont convertis en position 3D et permettent de calculer la position du casque. Ceci permet donc d'ajouter la détection des mouvements en translation. Ce traitement est automatiquement fait par le SDK du casque et retourne un vecteur de position.

2.3 Logiciel

Tout bon système de réalité virtuelle serait incomplet sans un logiciel qui permet d'exécuter les fonctionnalités de celui-ci. Cette sous-section aura donc pour but de détailler le fonctionnement du logiciel de réalité virtuelle. Plusieurs blocs du logiciel ont été expliqués dans la section 2.1 et ne nécessitent pas plus de détails. Les seuls qui méritent une étude plus approfondie sont les parties graphique et physique. Les sections 2.3.1 et 2.3.2 vont donc détailler le fonctionnement général des moteurs graphique et physique respectivement.

2.3.1 Moteur graphique

Le moteur graphique est le cœur de la réalité virtuelle. C'est lui qui va faire la gestion de l'affichage des différents modèles 3D à l'écran. Il sera en générale capable d'importer des modèles 3D, de faire la gestion de caméra, d'importer les textures et bien sûr, d'afficher. À retenir qu'ici le terme caméra fait référence au point de vue virtuel à l'intérieur de la simulation et non à une caméra externe permettant de filmer. À moins de spécification contraire, ce sera le cas pour la suite du mémoire. Il existe différents moteurs graphiques disponibles sur le marché en ce moment tel Unity et Unreal Engine pour ne citer que ceux-ci. Ils permettent aux développeurs de ne pas se soucier de l'aspect graphique, car le tout est géré directement par le moteur. Presque tous les moteurs graphiques, si ce n'est pas tous, sont bâtis à l'aide de l'API OpenGL, DirectX ou un dérivé des deux premiers. Le premier est libre et fonctionne sur tous les systèmes d'exploitation (OS). Le deuxième appartient à Microsoft et est fonctionnel seulement sur Windows ou autres produits appartenant à Microsoft. Comme le but du projet est d'avoir un système portable, cette section portera surtout sur l'utilisation d'OpenGL pour ne pas se limiter à un environnement Windows.

OpenGL est une API qui permet de communiquer certaines informations à la carte graphique pour que celle-ci fasse le traitement afin d'afficher le résultat à l'écran. Il s'agit donc d'une série de fonctions permettant de préciser quoi afficher et comment le faire. Voici donc une liste simple des étapes à faire à l'aide OpenGL pour afficher un contenu. La première est d'initialiser OpenGL. Il s'agit de spécifier la version utilisée, les dimensions de la fenêtre, et autres paramètres tel que le nombre de bits par couleur. Ensuite d'autres outils sont utilisés afin de retrouver les informations relatives au modèle à afficher telle la position des sommets et des textures ainsi que les normales et les couleurs. Celles-ci sont par la suite envoyées à la mémoire graphique à l'aide d'OpenGL. Ces informations sont, en théorie, envoyées une seule fois au début de la simulation, car elles ne changent pas. Finalement les informations ponctuelles telles que les positions et orientations de la caméra et des modèles sont envoyées à chaque itération vers la carte graphique. Ceci résume la partie de l'API OpenGL exécuté sur le CPU. Le reste des tâches nécessaires à l'affichage se font à l'aide des shaders sur la carte graphique. Ce processus est représenté sur la figure 2.8

Figure 2.8 Fonctions cœur OpenGL sur le processeur graphique

La communication entre le CPU et le GPU se fait dans les quatre boîtes en pointillé. Les informations sur les sommets du modèle sont envoyées dans le « Vertex Data ». Les textures sont envoyées dans le « Texel Data ». Les paramètres changeants à chaque itération sont envoyés dans les « Uniforms ». Le bloc « Pixel Data » est un outil permettant simplement de transférer des pixels entre le GPU et le CPU. Ces blocs sont donc des entrées et sorties. Le

processus d'affichage se fait sur la ligne verticale démarrant à « Vertex Specification ». Ces étapes s'exécutent en série, mais le travail à l'intérieur de celles-ci est parallélisé. Voici donc une courte description de chacune de ces étapes.

L'étape « Vertex Specification » permet de distribuer les différents sommets du modèle aux différentes instances en parallèle du « Vertex Shading ». Le « Vertex Shading » permet de positionner les différents sommets relativement à la caméra en fonction des positions et orientation du modèle et de la caméra. Il permet aussi de faire la projection de cette transformation géométrique dans le plan 2D représenter par l'écran. Le bloc « Tessellation » permet de diviser une face du modèle en différentes sous-faces. Cela augmente le nombre de sommets et permet d'augmenter le détail dans le modèle. Le bloc « Geometry Shading » prend une face en entrée et permet de retourner une quantité différente. Cela peut permettre de faire le rendu de la même face dans différentes couches du rendu. Ces deux derniers blocs ne sont pas obligatoires et peuvent être ignorés. La rastérisation est gérée automatiquement par la carte graphique et convertit les différentes faces sous forme de pixels. Il passe donc d'information vectorielle en information discrète. Ces informations sont par la suite envoyées au « Fragment Shading » qui permet d'appliquer les couleurs sur le modèle. Celui-ci prend les couleurs ou bien textures pour les afficher sur le modèle tout en lui ajoutant certains effets de lumières ou tout autre effet relié à la couleur si nécessaire.

2.3.2 Moteur physique

Le moteur physique est une nécessité dans la plupart des cas de simulation. Il peut supporter différents types d'effets physiques. Le plus commun est la détection de collision. Il y a bien sûr plusieurs autres effets telle la gravité, trajectoire de rebond, effet de particule, etc. Étant donné que le seul effet physique qui a été implémenté est la détection de collision,

cette section se concentrera sur celle-ci. Il existe différents types d'algorithmes de détection de collision, mais deux cas ont été implémentés dans notre étude. Le choix de l'algorithme se fait surtout en considérant la précision et la vitesse de calcul désirées [17]. Dans le cas présent, le but est de simuler un environnement 3D en temps réel. Il est donc approprié d'utiliser un algorithme moins précis, mais capable de s'exécuter en moins de 30 fois par seconde. Ce chiffre est le minimum désiré afin d'avoir théoriquement une vidéo fluide. Idéalement il faut un rafraîchissement de 60 fois par seconde afin de diminuer les cybers malaises dûs à la RV. L'approche utilisée dans les applications temps réel tel que les jeux vidéo est de diviser le modèle sous forme d'une hiérarchie et d'utiliser une approximation du modèle par un prisme. L'algorithme doit donc détecter des collisions entre des prismes, ce qui est beaucoup plus simple à calculer comparativement à l'utilisation de formes quelconques. Par exemple, dans le cas d'un personnage, on pourrait le diviser hiérarchiquement de la façon suivante. Un premier prisme est dessiné autour de celui-ci. Ensuite le personnage est divisé par membre et un autre prisme est dessiné autour de chacun de ceux-ci. Dans le cas où il y a plusieurs personnages, une première itération calcule les collisions avec la première boîte autour de chaque personnage. Ce principe est représenté dans la figure 2.9.

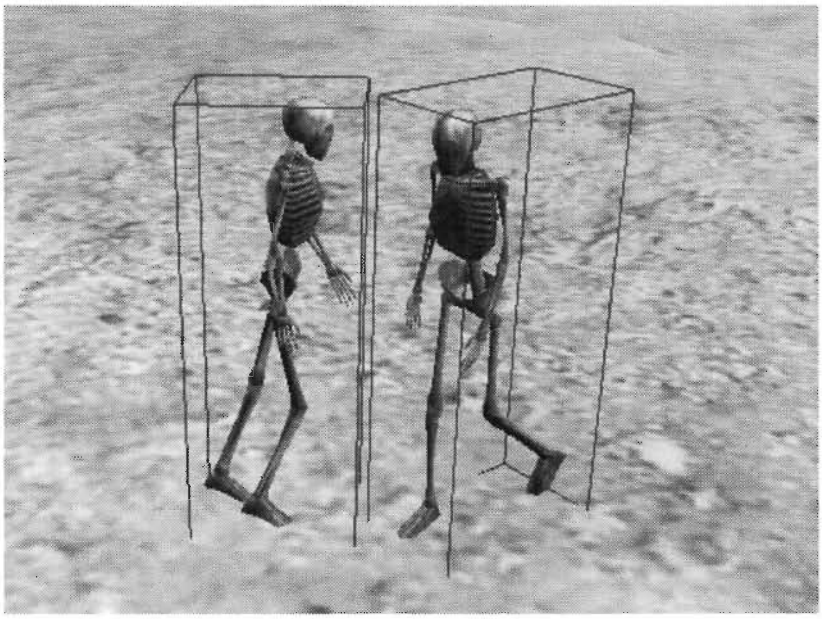

Figure 2.9 Premières étapes de détection de collision [19]

Ici les boîtes sont dessinées pour illustrer la situation, mais elles ne sont visibles pour l'utilisateur en temps normal. Si une collision est détectée, la région à tenir compte pour le calcul de collision est restreinte à la boîte englobant le personnage avec lequel il y a collision. La deuxième itération aura donc lieu si une collision est détectée à la première itération. Celle-ci testera s'il y a collision avec chacun des membres du personnage en question. Ceci est illustré à la figure 2.10.

Figure 2.10 Deuxième étape de détection de collision [19]

Dans le cas d'une collision, le mouvement sera restreint à l'intérieur de la simulation afin d'empêcher la caméra de passer à travers le personnage. La hiérarchisation permet donc de diminuer le nombre de calculs de collision en trouvant la région critique puis la simplification par prisme permet d'accélérer le calcul dans chaque itération. L'augmentation du nombre d'étages dans la hiérarchie permet d'atteindre une bonne précision tout en permettant de garder le temps de calcul à un minimum. Le calcul de la boîte pour la collision peut se faire de plusieurs façons. Le cas le plus simple est de dessiner une sphère. Celle-ci diminue les calculs au maximum, mais n'est pas très précise. Un cas un peu plus précis consiste à utiliser un prisme qui a les mêmes axes que le référentiel de la simulation. Ce cas se nomme *axis aligned bounding box* (AABB). Comme toutes les boîtes ont les mêmes axes, les calculs de collision sont quand même simples. Un cas un plus précis est de minimiser les dimensions du prisme en choisissant des axes orthogonaux optimaux pour le modèle. Celui-ci se nomme

oriented bounding box (OBB). Dans ce cas les axes ne sont pas tous les mêmes pour chacune des boîtes et augmente la complexité des calculs. Une illustration de ces différentes boîtes de collision est illustrée dans cas simplifié en 2D à la figure 2.11.

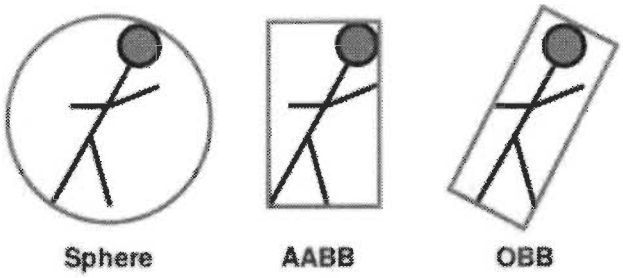

Figure 2.11 différents cas de boites de collision en 2D [20]

Il existe d'autres cas de calcul de boîte tel le *K-DOP* et *convex hull* qui augmente la précision tout en augmentant la complexité de calcul [17]. Un bon compromis est d'utiliser l'AABB ou l'OBB avec une hiérarchisation. Ceci permet de garder une précision convenable tout en gardant le temps de calcul bas.

2.4 Conclusion

La RV est donc un sujet complexe qui touche autant le matériel que le logiciel. Elle comprend l'utilisation d'un casque de vision stéréoscopique muni de capteurs pour simuler une situation virtuelle. Le tout se comporte comme un jeu vidéo, mais un peu plus complexe. La partie logicielle comprend principalement un moteur graphique et un moteur physique permettant de simuler la situation virtuelle. L'accord entre ces deux parties est donc crucial afin d'augmenter l'immersion au maximum.

Le prochain chapitre portera sur la conception du logiciel permettant de simuler un environnement de RV. Elle comprendra l'architecture du programme et l'accélération matérielle du calcul de collision.

Chapitre 3 - Conception logiciel/matérielle de RV

Ce chapitre a pour but de présenter la conception logicielle et matérielle de la RV. La majorité de ce projet se concentre sur la partie logicielle tandis qu'une petite partie a nécessité une conception matérielle. Ce chapitre se divise donc en deux parties. La première traite de l'architecture du logiciel et des différentes classes C++ qui le forme. La deuxième porte sur l'accélération du calcul de collision en utilisant OpenCL afin de l'implémenter sur FPGA.

3.1 Architecture du programme

Le programme de RV est conçu à l'aide du langage C++ et de l'API OpenGL. Par-dessus cela s'ajoutent quelques autres librairies que voici :

- assimp 3.0
- eigen
- glew
- GLM
- OpenCL 1.0
- OVR 0.6
- Qt 5.5
- SDL 2.0
- SDL image

Voici donc une description de ce que fait chacune de ces librairies. Assimp permet d'extraire les informations nécessaires pour l'affichage dans le cas de divers formats de fichier de modèle 3D. Eigen est une librairie qui permet de trouver les valeurs et vecteurs dans le cas de divers types de matrices. Celle-ci est nécessaire pour le calcul de collision. Plus de détails sur le sujet seront donnés dans la section 3.2.2. Glew sert à faire la gestion de version et de paramètre OpenGL en fonction du matériel disponible, sous Windows. Il est seulement utile de l'installer pour cet OS, car dans les autres cas elle est déjà implémentée dans l'OS. GLM est une série de fonctions mathématiques utiles pour les calculs graphiques. OpenCL permet de faire du calcul parallèle sur différents types de processeurs. OVR est la librairie utilisée pour faire fonctionner l'Oculus Rift. Qt est utile pour beaucoup d'applications, mais dans notre cas, elle est utilisée pour faire une interface graphique (GUI). SDL sert à communiquer avec différents périphériques tels que clavier et souris en plus de faire la gestion de fenêtre qu'OpenGL ne peut faire par lui-même. Il est à noter que tout ce que SDL permet de faire, Qt peut le faire également. La raison pour laquelle elle a été utilisée est que Qt n'était pas utilisé au départ, car il n'y avait pas d'interface graphique. L'ajout du GUI a nécessité l'ajout de Qt et le transfert de librairie n'a pas été fait jusqu'à présent. SDL image est un ajout à la SDL qui permet d'ouvrir différents fichiers d'images. Celle-ci est particulièrement utile pour l'ouverture de textures dans le modèle.

Toutes ces librairies ont été utilisées à l'intérieur des différentes classes créées en C++. Un schéma des connexions entre les différentes classes est présenté à la figure 3.1.

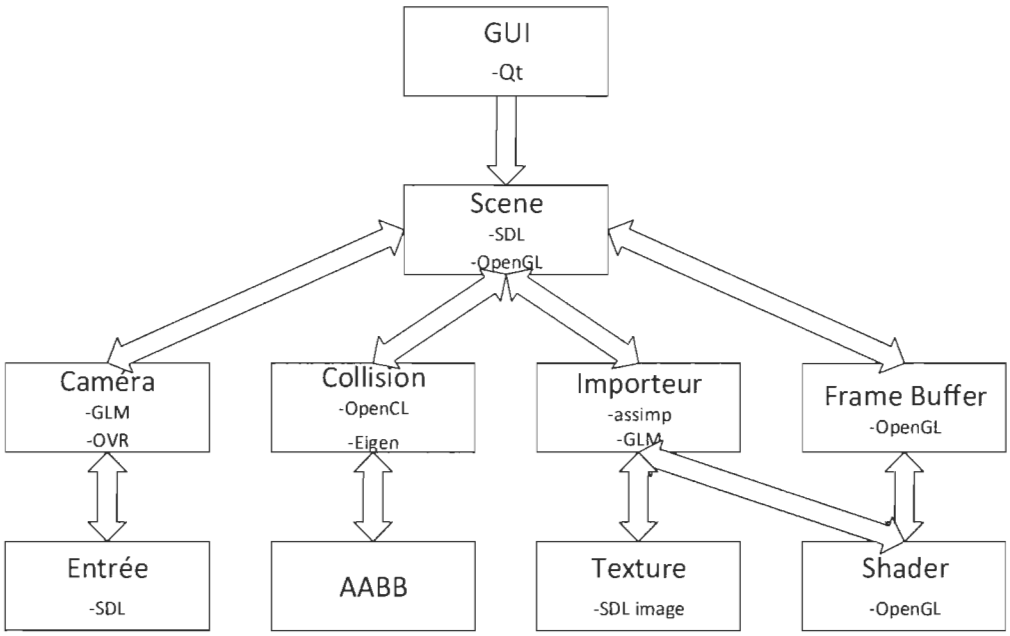

Figure 3.1 Architecture des classes du logiciel

À noter que la même classe Shader est utilisée dans les classes Importeur et Frame Buffer, mais il s'agit quand même de deux objets différents. Une description et explication de l'implémentation de ces classes seront détaillées dans les sections 3.1.1 à 3.1.8.

3.1.1 Interface graphique

Cette classe démarre une interface permettant à l'utilisateur de spécifier ce qu'il veut faire dans la simulation. Celle-ci a été développée par le stagiaire Antonin Troger. Les fonctions à l'intérieur du GUI ne sont pas toutes implémentées pour l'instant, mais les principales le sont. Elle permet à l'utilisateur de spécifier un fichier 3D à ouvrir. Ici les fichiers utilisés sont des « .obj », mais plusieurs autres sont compatibles avec la librairie. Il faut tenir compte que seul les sommets, normal, couleurs et texture sont prisent en charge par le moteur. Sinon, il peut choisir entre les différents continuums déjà implémentés dans le moteur. Ces choix sont les cas adulte et adolescent, masculin et féminin puis le mode première personne et le mode troisième personne. Un aperçu de cette interface est disponible à la figure 3.2.

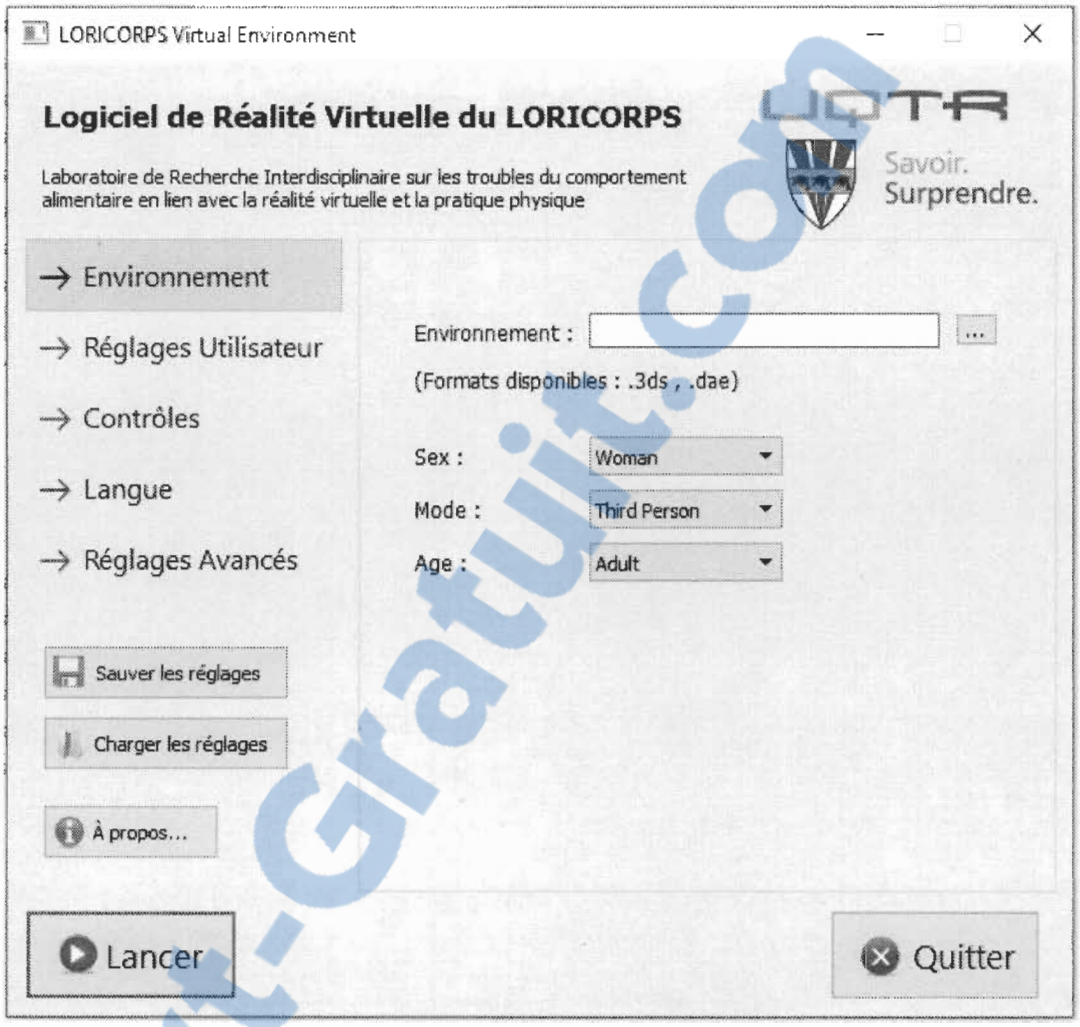

Figure 3.2 Interface utilisateur de la RV

Une fois les paramètres entrés, l'utilisateur appuie sur « lancer ». Un appel de la classe « Scène » se produit alors en lui envoyant les valeurs pertinentes. Le reste s'exécute à l'intérieur de cette classe et des sous-classes.

3.1.2 Scène

Cette classe est munie de trois méthodes. La première permet d'initialiser la fenêtre à l'intérieur de laquelle sera affiché le contenu OpenGL. Cela consiste à spécifier les dimensions et positions de la fenêtre en plus de spécifier les paramètres utilisés par OpenGL

comme le numéro de version et les dimensions de *buffer de couleur, profondeur et stencil*. La deuxième sert à initialiser Glew. Celle-ci essaie de démarrer OpenGL avec les paramètres précédemment spécifiés et retourne une erreur si ceux-ci ne sont pas supportés par le matériel. La troisième démarre la boucle d'affichage en initialisant les paramètres manquants et reboucle tant que l'utilisateur n'appuie pas sur la touche échappe. Celle-ci initialise un compteur d'image par seconde (FPS) afin de rectifier les vitesses de déplacement. Cela fait en sorte que lorsque le nombre d'images par seconde diminue ou augmente, les vitesses de déplacement apparentes ne diminuent pas ou n'augmentent pas. Cela donne une expérience plus agréable pour l'utilisateur en cas de fluctuations de FPS. Cette méthode permet aussi d'initialiser les matrices « modelview » et « projection » en plus d'initialiser les classes modèles 3D, caméra, frame buffer, entrées et détection de collision.

Les matrices mentionnées précédemment sont des matrices 4x4 qui servent à calculer la position des différents sommets du modèle une fois affiché sur l'écran. La matrice « modelview » permet de positionner les sommets du modèle dans le monde 3D derrière l'écran. Cette matrice est calculée en fonction du déplacement du modèle et de la caméra. Celle-ci pourrait donc être obtenue par une multiplication de deux autres matrices : La première « model » permettant de déplacer le modèle dans le référentiel monde 3D; et la deuxième « view » permettant de convertir les coordonnés monde 3D en coordonnés caméra en fonction de la position de celle-ci. La matrice « projection » permet de convertir les coordonnés caméra 3D en position projetée sur l'écran en 2D. Cette matrice est calculée une seule fois au démarrage à l'aide des dimensions d'écran, du champ de vision et de la profondeur de vue. La matrice « modelview », quant à elle, est modifiée à chaque itération

d'affichage due aux mouvements de la caméra. La figure 3.3 permet l'illustrer l'utilisation de ces matrices.

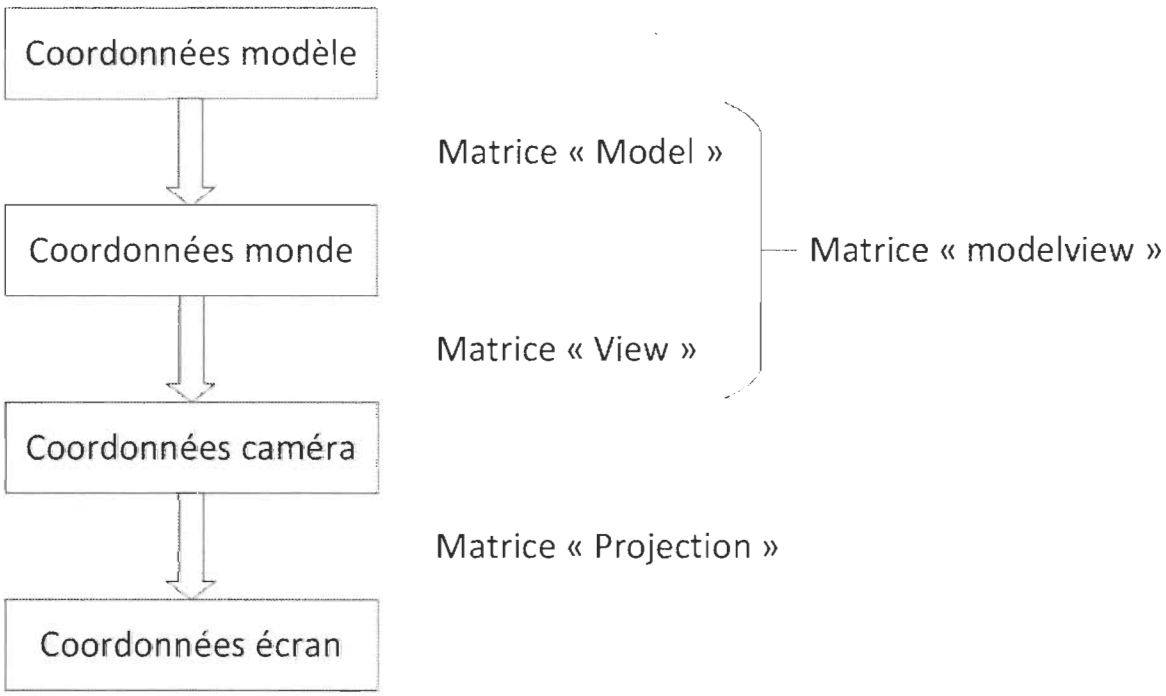

Figure 3.3 Utilisation des matrices d'affichage : traduite de [21]

Une fois que ces initialisations sont faites, le programme entre dans une boucle et s'exécute jusqu'à la fin de la simulation. À chaque itération, on met à jour les entrées et on vérifie si l'utilisateur veut terminer la simulation. Ensuite, on vient fournir un pointeur sur la classe « entrée » à la classe « caméra ». Celle-ci se sert de ces informations pour actualiser la position de la caméra. La position et l'orientation de la caméra sont extraites et envoyées au détecteur de collisions qui lui retourne une position corrigée en cas de collision. Cette correction est retournée à la caméra pour que celle-ci nous retourne la matrice « modelview » actualisée pour cette itération. À ce moment, l'affichage peut commencer. Ce processus se fait en deux passes. La première consiste à faire le rendu pour chacun des yeux dans un *frame buffer*. La deuxième passe consiste à prendre le *frame buffer* et lui appliquer une déformation

servant à redresser le rendu une fois affiché dans l'Oculus. Celle-ci sert à corriger les déformations mentionnées dans la section 2.2.1. Au final, la classe « scène » ne fait rien de concret, mais sert de répartiteur d'informations entre les autres classes.

3.1.3 Caméra

La classe caméra garde en mémoire les informations relatives à la position et l'orientation de la caméra et permet d'actualiser la matrice « modelview » à l'aide de celles-ci. À la création de cet objet, celui-ci s'initialise à une position et orientation définies dans les paramètres de création. Elle nécessite aussi un pointeur avec la classe OVR afin de pouvoir retrouver l'orientation du casque de RV. Les principales méthodes utilisées dans cette classe sont :

- void deplacer(Input const &input);
- void lookAt(glm::mat4 &modelview);
- void setPosition(glm::vec3 position);
- glm::vec3 getPosition() const;
- glm::mat3 getAxes() const;
- void setSensibilite(float sensibilite);
- void setVitesse(float vitesse);

La méthode « deplacer » permet d'actualiser la position de la caméra dans l'objet. Elle tient compte des mouvements de souris et du casque en plus du mouvement venant des touches du clavier et d'un joystick de la manette *Nunchuk* de Wii. Les déplacements en translation sont donc contrôlés par le clavier ou le joystick selon les préférences de l'utilisateur. Les mouvements d'orientation du casque et de la souris doivent quant à eux être

gérés de façon intelligente. Puisqu'un utilisateur a le casque de RV sur la tête, il ne faut pas que la souris permette de tourner autour de tous les axes. Par exemple les axes *Roll* et *Pitch* sont ressentis par l'utilisateur du casque. Ceux-ci ne doivent donc pas être modifiés par les mouvements de souris; sans quoi, l'utilisateur sera fortement désorienté. Les mouvements de souris doivent seulement pouvoir être effectifs sur l'axe *Yaw*. Cette méthode mettra à jour à l'interne la position et l'orientation de la caméra issues de la souris. Les données d'orientation venant de l'Oculus vont permettre de calculer la matrice « view ». Celui-ci retourne l'orientation sous forme de quaternions. Il faut donc convertir ce quaternion en matrice selon ces équations :

$$quaternion = a + bi + cj + dk \ (1)$$

$$position = [x; \quad y; \quad z;] \ (2)$$

$$vecteur\ latéral = [a^2 + b^2 - c^2 - d^2; \quad 2(bc + ad); \quad 2(bd - ac);] \ (3)$$

$$vecteur\ up = [2(bc - ad); \quad a^2 - b^2 + c^2 - d^2; \quad 2(cd + ab);] \ (4)$$

$$vecteur\ orientation = [2(bd + ac); \quad 2(cd + ab); \quad a^2 - b^2 - c^2 + d^2;] \ (5)$$

$$view = \begin{bmatrix} lateral.x & up.x & orientation.x & 0 \\ lateral.y & up.y & orientation.y & 0 \\ lateral.z & up.z & orientation.z & 0 \\ -lateral \cdot position & -lateral \cdot up & -lateral \cdot orientation & 1 \end{bmatrix} (6)$$

À cette matrice on applique la rotation venant des mouvements de la souris afin de retrouver la matrice « view » finale. Cela se fait à l'aide d'un produit matriciel entre celle-ci et la matrice de rotation de la souris. La matrice de rotation peut être calculée à l'aide de librairie GLM. Une fonction à l'intérieur de celle-ci calcule la rotation à l'aide de l'angle et

l'axe de rotation. Pour les intéressés, les matrices de modification de modèle sont les suivantes.

$$translation = \begin{bmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (7)$$

$$homothétie = \begin{bmatrix} x & 0 & 0 & 0 \\ 0 & y & 0 & 0 \\ 0 & 0 & z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (8)$$

$$rotation = \begin{bmatrix} x^2(1-cos\theta)+cos\theta & xy(1-cos\theta)-zsin\theta & xz(1-cos\theta)+ysin\theta & 0 \\ xy(1-cos\theta)+zsin\theta & y^2(1-cos\theta)+cos\theta & yz(1-cos\theta)-xsin\theta & 0 \\ xz(1-cos\theta)-ysin\theta & yz(1-cos\theta)+xsin\theta & z^2(1-cos\theta)+cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (9)$$

Pour la matrice de translation x, y et z représentent la distance de translation sur chacun de ces trois axes respectivement. Pour la matrice d'homothétie x, y et z représentent le facteur d'agrandissement sur chacun des axes. Dans le cas de la rotation x, y et z sont les composantes du vecteur unitaire représentant l'axe de rotation et θ représente l'angle de rotation.

La méthode « lookAt » retourne la matrice « modelview » actualisée avec la position et l'orientation de la caméra. Celle-ci va donc ajouter à la matrice « modelview » fournie les informations de la caméra. Il s'agit seulement de faire le produit matriciel entre « view » et « modelview » dans cet ordre.

Les accesseurs « getPosition » et « getAxes » permettent de retrouver ces informations dans le but de les envoyer au détecteur de collision. Celui-ci les utilise afin de calculer s'il y a une collision. S'il y a une collision, on retrouve la position corrigée et la modifie dans la

caméra à l'aide du mutateur « setPosition ». La correction se fait en annulant le déplacement dans l'axe de collision.

Les deux mutateurs « setVitesse » et « setSensibilité » permettent d'ajuster la vitesse de déplacement et la vitesse de rotation de la caméra. Ceux-ci sont particulièrement utiles pour compenser les fluctuations de FPS comme mentionné précédemment.

3.1.4 Entrées

Cette classe travaille exclusivement avec la librairie SDL. Celle-ci garde en mémoire dans des tableaux de booléens les touches de clavier et les boutons de souris appuyés. Elle garde aussi en mémoire la position de la souris, du joystick sur la *Nunchuk* de Wii et le mouvement relatif de la souris depuis la dernière mise à jour. Cette classe utilise donc le SDL pour trouver les périphériques disponibles et les initialiser. Ensuite, elle utilise le gestionnaire d'évènements de la SDL afin de boucler dans tous ceux-ci et mettre à jour les tableaux mentionnés précédemment. La classe est munie de plusieurs accesseurs permettant de retrouver les différentes actions de périphériques que l'utilisateur désire vérifier. La seule particularité de cette classe est que l'accesseur du joystick doit corriger l'erreur sur celui-ci, car il n'est jamais centré parfaitement sur zéro. Elle vient aussi réduire la plage de valeur du joystick afin d'être dans le même référentielle que les déplacements de souris.

3.1.5 Importation de modèles

Cette classe est l'une des plus complexes. Elle extrait les données des modèles 3D et les formate correctement avant de les envoyer vers la mémoire graphique. Une instance de la classe doit être créée pour chaque fichier 3D. Elle permet aussi de faire l'affichage du modèle.

Elle est munie de deux méthodes principales. Une fois l'objet créé, le constructeur alloue de la mémoire pour accommoder les informations du modèle 3D. Il crée aussi deux instances de texture pour chacun des matériaux présents dans le modèle. Il y a deux textures, car un *mesh* peut avoir une texture standard et un *bump map*. Celui-ci n'est pas exactement une texture, il s'agit plutôt d'un fichier image. Plus de détails sur le sujet seront présentés dans la section 3.1.8.

Ensuite, on utilise la fonction « charger » qui permet de charger le modèle 3D en mémoire, le formater et l'envoyer à la carte graphique. L'envoi des informations à la carte graphique permet d'accélérer le processus d'affichage à chaque itération puisque celle-ci retrouve toute l'information nécessaire dans sa mémoire locale. Elle évite ainsi d'avoir à retrouver de grandes portions de mémoire dans la mémoire RAM de la carte mère. Cela se fait à l'aide des *vertex buffer object* (VBO) qui contiennent les sommets, normales, couleur, etc. Une autre option permettant d'accélérer l'affichage est l'utilisation de *vertex array object* (VAO). Les VAO permettent de stocker sur la carte graphique l'information sur comment accéder aux différentes informations du VBO. Il s'agit donc d'une série d'instructions qui n'a plus à être envoyée par le processus sur le CPU. Il est aussi bon de noter que les VAO ne peuvent pas être utilisés sans VBO correspondant. Un fait intéressant est que l'utilisation des VAO et VBO est obligatoire depuis la version 3.3 d'OpenGL.

Le chargement se fait dans une boucle qui s'exécute autant de fois qu'il y a de *mesh* dans le modèle 3D. Un *mesh* est un ensemble de faces qui forme un objet. Pour chacun de ceux-ci, on vient retrouver les sommets en les associant par face. Si la face n'est pas triangulaire, celle-ci doit être triangulée. Dans le cas d'un carré, celui-ci sera divisé en deux triangles. Dans le cas d'un polygone à « n » faces voici l'algorithme de triangulation.

```
for ( int i = 0; i < n; i++)
{
    triangle[i][0] = sommet[0];
    triangle[i][1] = sommet[1+i];
    triangle[i][2] = sommet[2+i];
}
```

Une image illustrant ce principe est présenté à la figure 3.4.

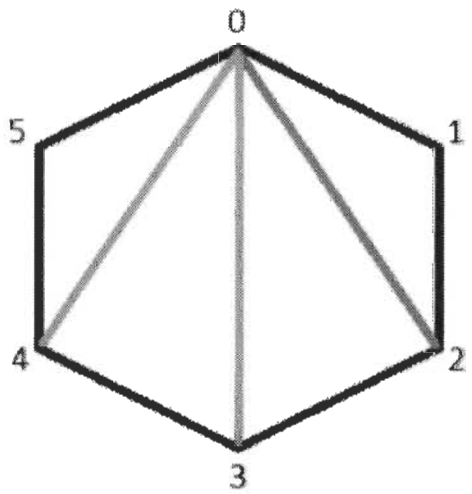

Figure 3.4 Exemple de triangulation

Cet algorithme fonctionne pour tout polygone convexe dont le nombre de sommets est plus grand que trois. Une fois les sommets extraits, les coordonnés de texture associées au sommet sont extraites dans le même ordre que les sommets. Dans le cas où une triangulation a été faite, il faudra en tenir compte. Le même processus est appliqué pour toutes les autres valeurs associées aux sommets qui sont les normales, tangentes et bitangentes. Il est bien

important d'extraire toutes ces valeurs dans le même ordre, car lors de l'affichage, elles sont lues en parallèle et sont associées les unes aux autres. Si ce n'est pas respecté, les couleurs et effets lumineux ne seront pas au bon endroit.

Voici donc une description de ces différentes valeurs extraites. Les sommets sont simplement les positions des sommets du modèle 3D. Les coordonnées de texture représentent le pixel à l'intérieur de la texture qui est associé au sommet. Les normales sont les vecteurs normaux aux faces où se trouvent les sommets. Elles servent à calculer les effets lumineux dus à la lumière diffuse. Les tangentes et bitangentes sont associées aux normales afin d'avoir un système d'axes orthonormé permettant de passer entre le repère caméra et le repère à la face du *mesh*. Ceci est utilisé pour calculer les effets de lumière spéculaire et de *bump map*. Celui-ci se sert d'effet de lumières afin de donner une impression de relief sur une surface plate du modèle 3D.

Une fois toutes les données extraites du *mesh*, on vient associer un VBO à celui-ci. Cela consiste à créer un *buffer* sur la carte et lui associer une ID. Ce *buffer* est dimensionné de manière à entrer toutes les valeurs extraites précédemment. Ensuite, on crée un VAO en lui donnant une ID. Le VAO est associé à l'ID du VBO. Par après, pour chacun des tableaux extraits précédemment, on spécifie quelle portion du buffer le représente et on l'associe à l'ID de cette variable dans le *shader*. Une description plus détaillée se trouve dans la section 3.1.8. Le tout résume le travail fait lors du chargement du modèle 3D.

L'autre méthode utilisée est « afficher ». C'est la commande utilisée à l'intérieur de la boucle dans la classe « scène » pour afficher le modèle 3D. Celle-ci nécessite les matrices « modelview » et « projection ». Comme un modèle 3D peut être sous forme d'un arbre avec plusieurs nœuds, cette méthode est récursive. Cela permet de balayer tous les nœuds du

modèle 3D en partant du nœud principal. Dans chacun des nœuds, la matrice « modelview » est multipliée avec la matrice de transformation du nœud, car il peut y avoir une transformation par rapport au nœud parent. Ensuite, une boucle parcourt tous les *meshs* associés à celui-ci. À l'intérieur de la boucle, on spécifie le *shader* utilisé et par la suite le VAO utilisé. Le VBO n'a pas à être spécifié, car il a été associé au VAO lors de ca création. Les étapes suivantes sont d'envoyer les matrices au *shader* et toutes les autres valeurs pertinentes. Celles-ci varient selon la structure du *shader*. Puis les ID de texture sont spécifiées. La dernière demande seulement à OpenGL de dessiner les triangles associés au VAO du *mesh*.

3.1.6 Textures

Cette classe se sert de la librairie « SDL image » pour ouvrir les textures qui seront appliquées sur le modèle 3D. Elle est aussi utilisée dans la création de *frames buffers*. Ceux-ci sont en réalité des textures avec la seule différence qu'elles sont vides avant qu'un rendu soit fait dans le *frame buffer*. Cette classe est donc munie de plusieurs constructeurs. Le premier permet de créer une texture non initialisée. Cela est utile lorsque l'on veut initialiser un tableau de textures pour un modèle. Le deuxième constructeur permet d'initialiser l'objet avec un fichier image. Le troisième crée une texture avec une dimension définie, mais sans tableau de pixels. C'est ce dernier qui est utilisé lors de la création d'un *frame buffer*.

Ensuite cette classe à cinq méthodes. Les deux premières sont les méthodes pour charger la texture en mémoire. La première permet de charger une texture dont le fichier image a été fourni. La deuxième permet de charger une texture vide pour le cas du *frame buffer*. Il y a une méthode qui spécifie le fichier image dans le cas où la texture n'a pas été initialisée. Elle sera utilisée dans le cas où un tableau de textures sera créé. La procédure à

suivre dans ce cas est donc d'initialiser un tableau de textures non initialisées puis de spécifier le fichier image et la charger pour toutes les textures du tableau. Ensuite, il y a l'accesseur « getID » qui permet de spécifier l'ID de la texture lors de l'affichage. La dernière méthode en est une qui est utilisée à l'intérieur de la classe elle-même. Elle permet d'inverser les pixels d'une texture. Cela est nécessaire, car le référentiel de la SDL et d'OpenGL n'est pas les mêmes. Dans le cas de la SDL, l'origine est en haut à gauche alors que dans OpenGL, l'origine est en bas à gauche. Donc, si la texture n'est pas inversée les rangées de pixels sont lues dans le désordre et on se retrouve avec une image qui a subi un effet miroir sur l'axe vertical comme représenté à la figure 3.5.

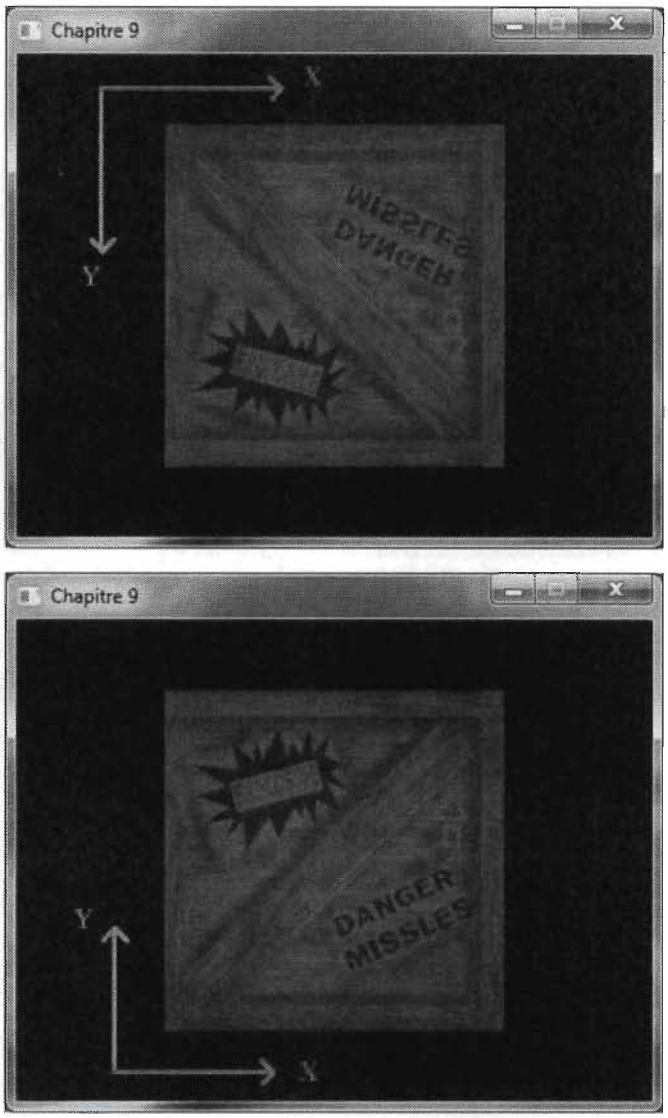

Figure 3.5 Repères des textures SDL et OpenGL respectivement [22]

La méthode « charger » est la plus complexe et se déroule comme suit. Elle commence par mettre les pixels de l'image dans une surface SDL et vérifie qu'il n'y pas d'erreur. Par exemple un fichier inexistant. Ensuite il y a création d'une deuxième surface SDL pour stocker la texture inversée avant de supprimer la première surface afin de libérer la mémoire. Puis un ID est attribué à la texture et les fonctions de la surface SDL sont utilisées afin de trouver le format interne et le format de l'image. Le format interne spécifie si l'image est à trois couleurs ou trois couleurs plus transparence. Cela aura une importance, car la dimension

en mémoire d'un pixel ne sera pas la même. Le format représente l'ordre des couleurs. Il peut être de type Rouge, Vert, Bleu (RGB) ou Bleu, Vert, Rouge (BGR). Ceci est important, car les canaux Rouge et Bleu seront inversés lors de l'affichage dans le cas où le format est mal spécifié. Les pixels sont ensuite envoyés à la carte graphique en lui spécifiant les paramètres de format, les dimensions de l'image et le pointeur sur les pixels de la surface SDL. La dernière étape n'est pas obligatoire, mais peut être utile afin d'améliorer le rendu et les performances. Il s'agit de l'ajout de filtres à la texture. Il suffit simplement de spécifier à OpenGL le filtre qu'il doit appliquer et quand il devra l'appliquer. Dans notre cas, on a spécifié deux filtres. Le premier fait en sorte que lorsque la texture est éloignée de la caméra, OpenGL peut se permettre de diminuer la résolution de celle-ci. Cela va permettre d'améliorer les performances sans diminuer la qualité, car la texture n'est plus vraiment visible. Dans le second cas, on spécifie qu'OpenGL doit lisser la texture à l'aide d'une interpolation linéaire lorsque celle-ci est proche de la caméra. Cela va améliorer l'effet visuel sur le rendu. Il existe aussi d'autres méthodes de filtrage plus avancées qui n'ont pas été étudiées.

3.1.7 Mémoire tampon de trame (Frame Buffer)

La classe *frame buffer* permet de générer une image dans la mémoire de la carte graphique sans l'afficher à l'écran directement. En temps normal, lorsque le rendu est fait dans un *shader*, le résultat est affiché à l'écran. Si le rendu dans le *shader* est spécifié à l'intérieur d'un *frame buffer*, le résultat sera simplement mis en mémoire. Cela est utile lorsque le rendu final à afficher doit passer par plusieurs *shaders*. Un exemple de rendu en plusieurs étapes serait l'affichage du reflet dans un miroir. La première étape est de faire le rendu du reflet dans un *frame buffer*. La deuxième étape est de faire le rendu de la scène avec

le *frame buffer* comme texture pour le miroir. Un exemple qui fonctionne dans le sens inverse et qui est utilisé dans le cas présent, est la correction de la déformation d'image dans l'Oculus. Le rendu de la scène est donc fait en premier dans *frame buffer*. On retrouve à l'intérieur de celui-ci le contenu tel que l'on désire le visualiser. Le problème est que les lentilles de l'Oculus déforment l'image affichée à l'écran. Le rendu du *frame buffer* est donc passé dans un autre *shader* qui lui applique une distorsion de type *barrel* sur la texture afin d'annuler la distorsion de type *pincushion.*

Un *frame buffer* se comporte un peu comme une texture, mais de façon un peu plus complexe. Comme mentionné, il contient une texture. En fait, il peut en contenir jusqu'à 16 qui se nomment *color buffer*. Cela permet de faire plusieurs étapes de rendu. Il est aussi muni d'un *depth buffer* qui sert à faire les tests de profondeur afin de cacher les objets en arrière-plans derrière les objets en premier plan. Sans celui-ci, on peut voir au travers des objets en fonction de l'ordre d'affichage. Cela est dû au fait que lorsque le *depth buffer* n'est pas présent chaque étape de rendu s'affiche par-dessus l'étape précédente. Il existe aussi le *stencil buffer* qui est optionnel. Celui-ci permet de créer un masque sur la texture afin d'exprimer quelle portion de celle-ci doit être affichée. Cela est illustré à la figure 3.6.

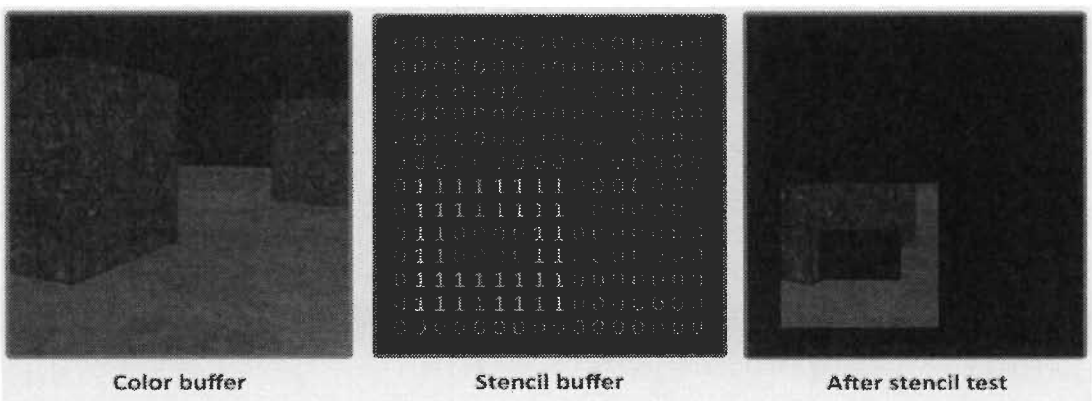

Figure 3.6 Exemple de *stencil buffer* [23]

Le *stencil buffer* est donc rempli de 0 et 1 et permet de spécifier les pixels du *color buffer* qui doivent être visibles et ceux qui ne doivent pas l'être. Les 0 sont représentés en noir ici parce qu'il n'y a rien derrière le *color buffer*. S'il y avait un objet derrière celui-ci, nous le verrions là où il y a des 0.

Le constructeur du frame buffer nécessite donc les dimensions en pixels du *color buffer* et la quantité de ceux-ci. Un autre paramètre spécifie si le *stencil buffer* doit être implémenté. Ensuite, la méthode « charger » est utilisée. Celle-ci va commencer par allouer un ID au *frame buffer*. Ensuite, une texture vide est créée pour chacun des *color buffers* et est chargée. Puis, la méthode « ceerRenderBuffer » est utilisée pour créer un *depth buffer* et un *stencil buffer* si spécifié ainsi. Ces deux buffers sont à l'intérieur du même *render buffer* qui lui aussi à un ID. Finalement les différents buffers créés sont associés au *frame buffer*. Cette classe est aussi munie de quelques accesseurs qui permettent de retrouver les dimensions du buffer ainsi que l'ID du frame buffer ou celui des textures générées à l'intérieur de celui-ci.

3.1.8 Nuanceur (Shader)

Le *shader* est la partie qui se charge de gérer comment le contenu des modèles 3D est affiché. Il permet d'ajouter des effets visuels ou d'en enlever. Plus le rendu sera réaliste, plus le *shader* sera complexe. C'est donc la partie mathématique qui va calculer tous les effets visuels désirés. Il faut distinguer dans cette partie la classe *shader* et le *shader* en lui-même. La classe permet d'initialiser celui-ci, de le compiler, de l'envoyer à la carte graphique et de créer la passerelle de communication avec le CPU. Le *shader* lui, est le code qui est envoyé à la carte graphique et qui s'exécute sur celle-ci. Il a son propre langage qui est relativement simple à comprendre. Il existe plusieurs langages de *shader* qui varient en fonction de la librairie graphique utilisée. Les plus connues sont l'*OpenGL Shading Language* (GLSL) et

le *High Level Shading Language* (HLSL). Le premier est propre à OpenGL tandis que le deuxième est propre à DirectX. Le GLSL est donc celui qui est utilisé dans le présent contexte.

La classe *shader* est une classe qui permet de construire un *shader* à partir de plusieurs fichiers texte. Ces fichiers texte sont les codes GLSL divisés en deux parties. La première est le *vertex shader* et le deuxième est le *fragment shader*. Il en existe d'autres, mais ceux-ci sont les deux qui sont obligatoires pour faire fonctionner OpenGL. Le premier fait les calculs pour la géométrie du modèle et le deuxième calcule les couleurs qui doivent être affichées. Le constructeur de cette classe ne fait qu'enregistrer les chemins de fichier *shader*. Ensuite la fonction « charger » se charge de préparer le code pour la carte graphique. Celle-ci commence par créer un ID pour le programme, qui est l'association des fichiers du *shader*. Elle appelle la fonction « compilerShader » qui permet de compiler les deux fichiers en associant un ID à chacun de ceux-ci. Cette fonction vient simplement créer un ID associé au type de shader désiré. Elle lit ensuite ligne par ligne le fichier texte pour mettre le contenu dans un « string ». Le « string » créé est ensuite associé à l'ID et la fonction « glCompileShader » est appelée. Finalement, on vérifie qu'il n'y pas eu d'erreur. Une fois les deux *shader* compilés, ils sont associés à l'ID du programme à l'aide de leur ID. Les entrées du *vertex shader* sont ensuite associées à un entier en spécifiant leur nom. Cet entier sera la location de l'attribut spécifié lors de la création des VBO. Cela permet de spécifier quelle portion du VBO est associée à quelle entrée. Finalement la fonction « glLinkProgram » vient lier le programme à la carte graphique puis elle vérifie qu'il n'y pas eu d'erreurs. La classe est aussi munie de quelques autres fonctions qui permettent d'envoyer les variables de type uniforme au *shader*. Celles-ci sont différentes pour chacun des types de

données, par exemple, une matrice, un entier, une virgule flottante, etc. Il y a aussi une fonction qui permet de récupérer l'ID du programme. Celle-ci est nécessaire lors de l'affichage, car le programme peut être différent pour chaque modèle 3D. Cela permet de spécifier quel programme présent sur la carte graphique doit être utilisé pour faire l'affichage d'un modèle 3D en particulier.

La portion *shader* est écrite dans un fichier texte à part. Cette portion pourrait directement être écrite dans un « string » à l'intérieur de la classe *shader* mais cela permet de facilement modifier le *shader* sans avoir à recompiler le projet. Ces fichiers sont tous écrits par paire que le *vertex shader* communique avec le *fragment shader*. Les deux doivent être compatibles ensemble afin de créer un programme. Les entrées du *vertex shader* doivent donc être spécifiées dans la classe *shader*. Dans le cas présent, les entrées possibles sont les suivantes :

- in_Vertex
- in_Color
- in_TexCoord
- in_Normal
- in_Tangent
- in_Bitangent

Ces entrées sont des tableaux de vecteurs et chacun de ces vecteurs crée une instance du *shader*. Chaque instance va donc traiter un seul vecteur par entrée. Puis chacune de ces instances s'exécute en parallèle sur la carte graphique. Les sorties du *vertex shader* sont les entrées du *fragment shader*. Le *fragment shader* lui, à une seule sortie qui est la couleur du

pixel. Ces entrées et sorties ont donc tous une valeur différente pour chacune des instances du *shader*.

Il existe aussi d'autres types d'entrées qui sont appelées uniforme. Elles sont modifiées à chaque itération d'affichage, mais sont constantes pour toutes les instances du *shader* à cette itération. Ces variables sont par exemple les matrices « modelview » et « projection ». Voici donc un exemple de *shader* simple qui permet de clarifier tout cela.

Vertex Shader :

```
// Version du GLSL
#version 330 cores


// Entrées
in vec3 in_Vertex;
in vec3 in_Color;

// Uniform
uniform mat4 projection;
uniform mat4 modelview;

// Sortie
out vec3 color;

// Fonction main
void main()
{
    // Position finale du vertex en 3D
    gl_Position = projection * modelview * vec4(in_Vertex, 1.0);

    // Envoi de la couleur au Fragment Shader
```

```
    color = in_Color;
}
```

Fragment Shader :

```
// Version du GLSL
#version 330 cores

// Entrée
in vec3 color;

// Sortie
out vec4 out_Color;

// Fonction main
void main()
{
   out_Color = vec4(color, 1.0);
}
```

Ce code est un des plus simples qui permet d'afficher un modèle. Il permet seulement d'afficher le modèle avec les couleurs spécifiées. Il ne gère pas les textures, effets lumineux ou tout autre effet spécial. Les entrées du *vertex shader* sont les sommets du modèle 3D et les couleurs associées à ces sommets. Le vecteur est augmenté de 1 afin de pouvoir avoir l'effet d'homothétie. Celui-ci est multiplié par la matrice *modelview* afin de le ramener dans le repère caméra puis par la matrice projection afin de la projeter à l'écran. Cette valeur est mise dans *gl_position* qui est une variable réservée dans GLSL. Cette variable est utilisée pour faire la rastérisation qui permet de remplir les pixels manquants entre les différents sommets une fois projetés. Ce processus est fait automatiquement par OpenGL et ne nécessite rien de la part du programmeur. La couleur associée à ce vecteur est simplement transférée

au *fragment shader*. Cette valeur sera aussi utilisée dans la rastérisation. La figure 3.7 montre un exemple de rastérisation.

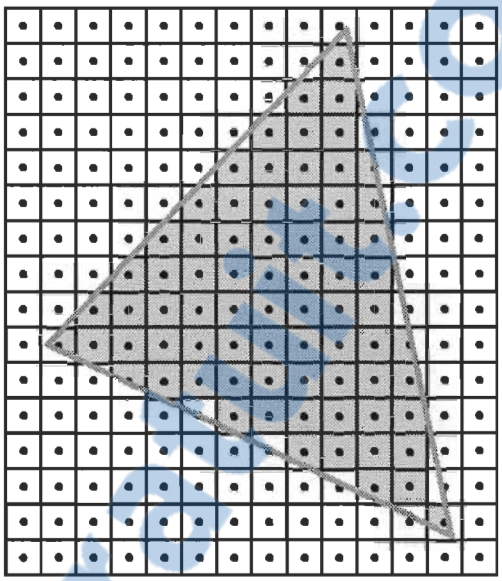

Figure 3.7 Exemple de rastérisation

La rastérisation reçoit dans ce cas trois sommets avec chacun une couleur associée. Ce processus remplit les pixels se situant entre les trois sommets et leur applique chacun une couleur en faisant une interpolation avec les couleurs des trois sommets. Un exemple d'interpolation de couleur dans le cas où les sommets sont de différentes couleurs est illustré à la figure 3.8.

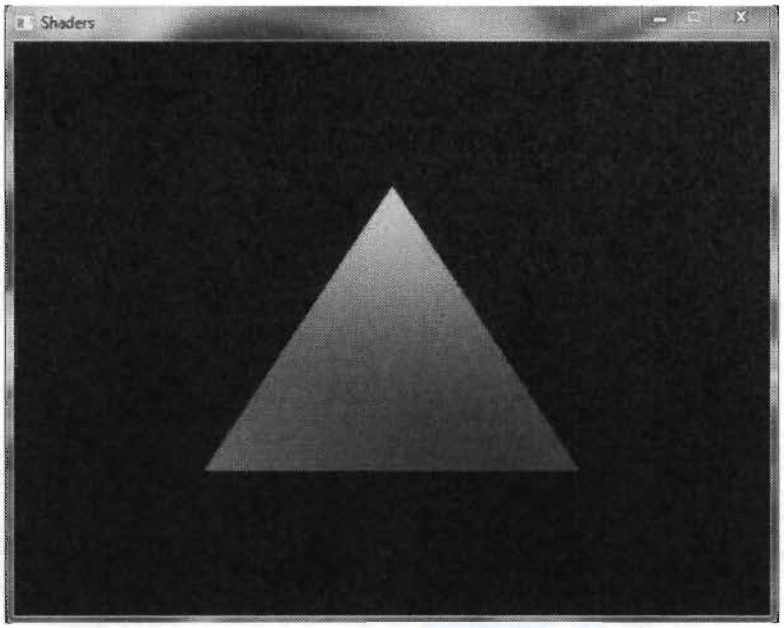

Figure 3.8 Exemple d'interpolation de couleur [22]

Ce principe n'est pas très utile dans le cas de couleurs, mais est très utile dans le cas où une texture est affichée. Dans ce cas, la valeur transférée au *fragment shader* est une coordonnée 2D représentant un pixel dans l'image de la texture à la place d'une couleur RGB. Dans ce cas, l'interpolation va trouver les pixels intermédiaires de la texture à appliquer.

Ce sont donc les couleurs une fois interpolées à l'aide de la rastérisation associée à chaque pixel qui sont envoyées au *fragment shader*. Le fragment *shader* dans ce cas simple transfert la couleur en sortie augmentée du canal *alpha* qui gère la transparence. Dans des cas plus complexes, il y aurait du traitement sur les couleurs afin d'ajouter les effets de lumières (reflet, etc).

Dans le cas du moteur créé, quatre *shaders* ont été implémentés. Le premier gère les couleurs dans l'espace 3D comme l'exemple montré, mais avec les effets lumineux ajoutés. Celui-ci est pour faire le rendu de modèles sans texture. Le deuxième gère les cas de texture dans l'espace 3D pour faire le rendu des modèles avec texture. Ceux-ci sont utilisés dans la

classe importation de modèles. Les deux autres servent à faire la deuxième passe du rendu tel que mentionné dans la section 3.1.7. Donc une fois le rendu fait dans les *frames buffers* gauche et droit la texture créée passe un *shader* qui corrige l'image. Le troisième *shader* vient seulement gérer le cas de texture en 2D afin d'afficher les vues de chaque œil côte-à-côte à l'écran afin de visualiser le contenue sans mettre le casque. Le quatrième *shader* fait la même chose que le précédent, mais avec une correction de la distorsion afin de redresser l'image dans l'Oculus. La figure 3.9 montre ce que voit approximativement l'utilisateur dans l'Oculus dû à la distorsion des lentilles.

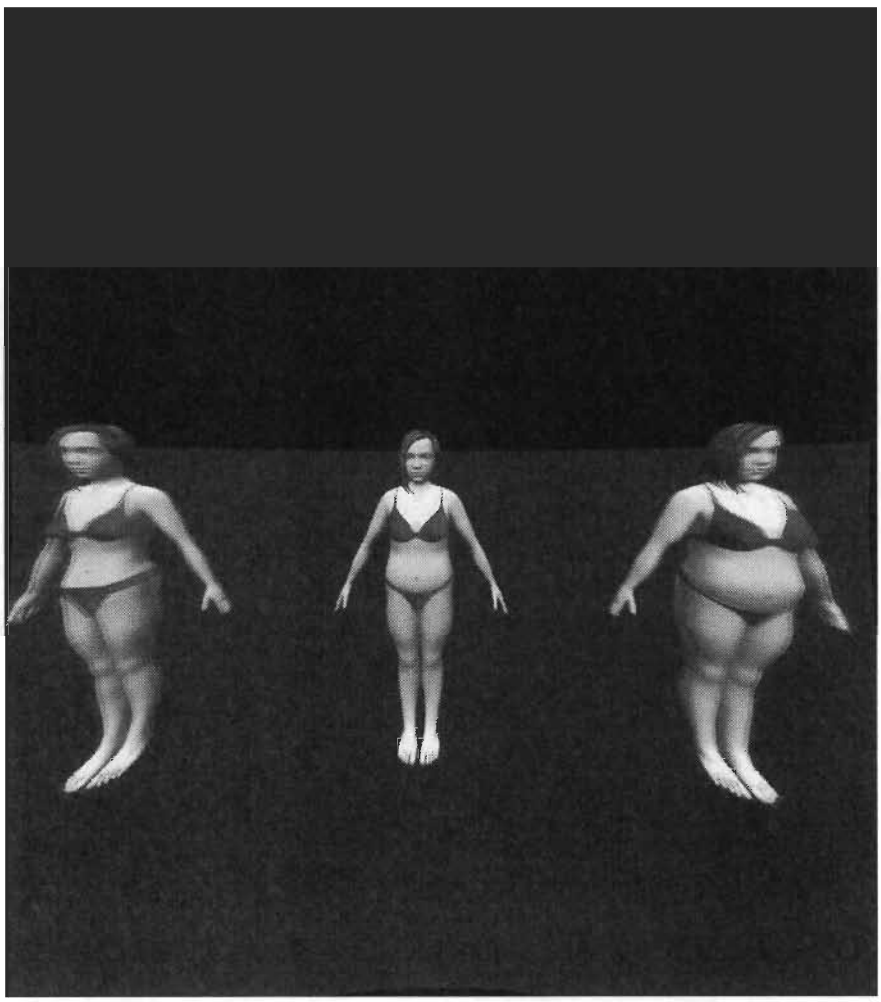

Figure 3.9 Distorsion des lentilles de l'Oculus

On remarque que l'image est déformée comme si les coins étaient étirés et que les couleurs se séparent sur les bords de l'image. Cela est corrigé grâce au quatrième *shader* qui se nomme « distorsion ». Celui-ci fait la déformation à l'inverse des lentilles. La figure 3.10 montre le rendu corrigé qui est affiché sur l'écran de l'Oculus.

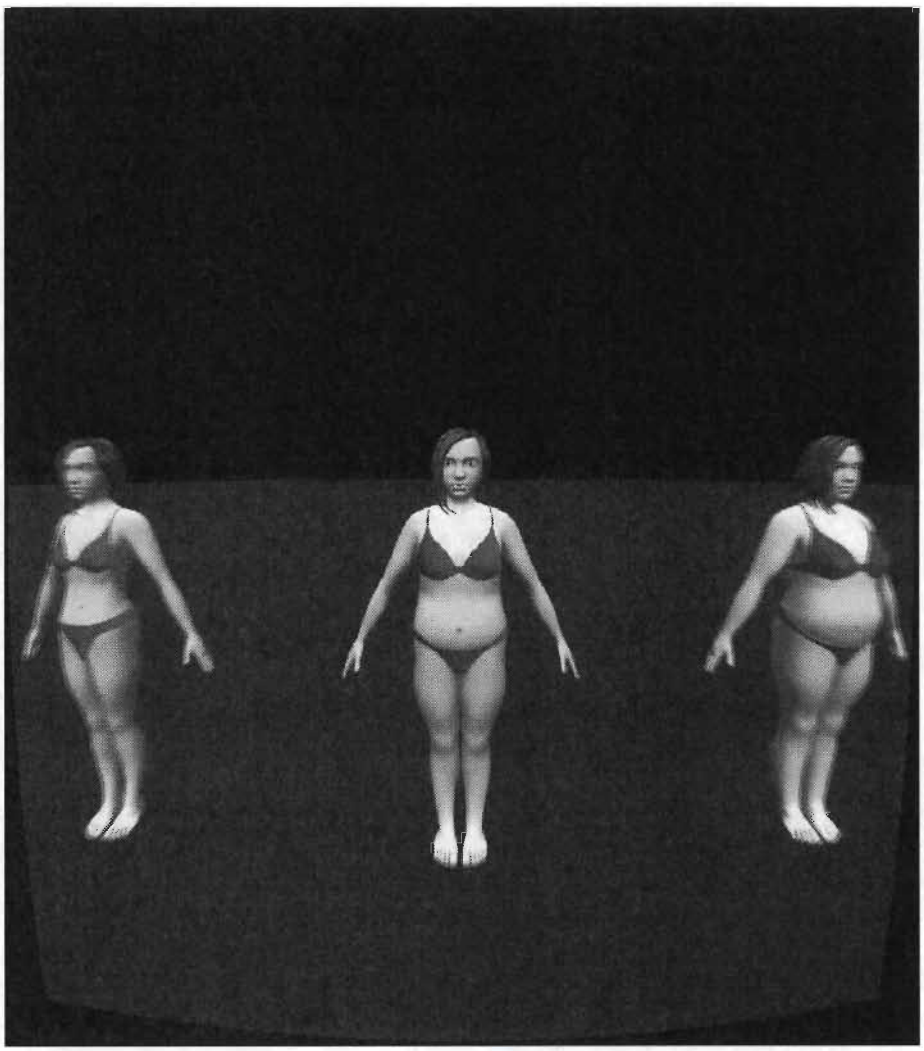

Figure 3.10 Correction de déformation envoyée à l'Oculus

On remarque que la correction crée une déformation inverse en poussant sur les coins de l'image et en séparant les couleurs dans le sens inverse. Cette image une fois visualisée à travers des lentilles est corrigée comme le montre la figure 3.11.

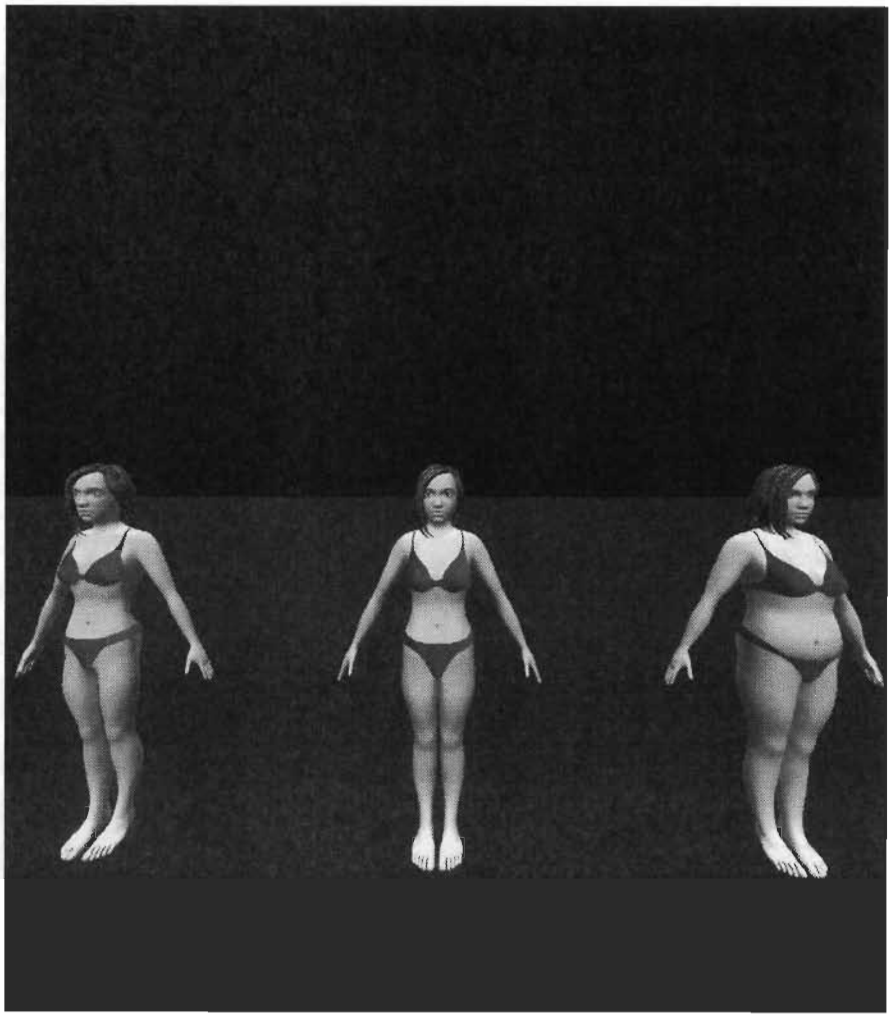

Figure 3.11 Rendu visualisé dans l'Oculus une fois l'image corrigée

La correction de la distorsion *pincushion* se fait en rapprochant les extrémités de l'image vers le centre. Cela se fait en rapprochant chaque point de l'image selon le coefficient de la lentille multiplié par le cube de la distance par rapport au centre. Cela se fait en normalisant l'image à une dimension un par un avec le centre de coordonnées [0,5 0,5]. La correction se fait selon les équations suivantes :

$$distance\ radial = dr = \left| \frac{coordonnée - [0{,}5 \quad 0{,}5]}{|0{,}5 \quad 0{,}5|} \right| \quad (10)$$

$$vecteur\ unitaire\ radial = vr = \frac{coordonnée - [0{,}5 \quad 0{,}5]}{|0{,}5 \quad 0{,}5| * dr} \ (11)$$

$$nouvelle\ distance\ radial = ndr = dr + K * dr^3 \ (12)$$

$$nouvelle\ coordonnée = nc = vr * ndr * |0{,}5 \quad 0{,}5| * scale + [0{,}5 \quad 0{,}5] \ (13)$$

« *K* » est le coefficient des lentilles et « *scale* » est un vecteur permettant de redimensionner l'image aux dimensions de l'écran. Ce facteur est nécessaire, car cet algorithme à tendance à rétrécir l'image en plus de faire de la distorsion. Celui-ci permet donc de remplir l'écran. Ce facteur est calculé en trouvant l'inverse du facteur de rétrécissement de l'algorithme sur les axes vertical et horizontal de l'image. Cela se fait en utilisant l'équation suivante :

$$scale = \frac{dr}{ndr} \ (14)$$

Celle-ci calcule les deux valeurs du vecteur « scale » en remplaçant la valeur de « coordonnée » dans [10] pour les valeurs extrêmes sur l'axe horizontal et l'axe vertical soit $[1{,}0 \quad 0{,}5]$ et $[0{,}5 \quad 1{,}0]$ respectivement.

L'ajout de la correction de l'aberration chromatique à cet algorithme se fait assez simplement en ajoutant un facteur supplémentaire sur la nouvelle distance radiale qui sera différente pour les trois canaux de couleur. Le canal vert est au centre du spectre donc il reste au même endroit avec un facteur de 1. Le canal rouge à un facteur plus petit que 1 (0,988) afin de le rapprocher vers le centre. Puis le canal bleu à un facteur plus grand que 1 (1,012) afin de l'éloigner du centre. Ces valeurs ont été trouvées empiriquement puisque les facteurs des lentilles de l'Oculus ne sont pas disponibles. Le code GLSL permettant de faire cette déformation est présenté à l'annexe A.

3.2 Architecture matérielle

Le moteur de RV à presque entièrement été conçu à l'aide de la programmation en C++. Par contre, dans le but d'optimiser les performances, une partie a été implémentée en parallèle à l'aide de l'OpenCL. Une partie du travail est déjà parallélisé sur une architecture matérielle qui est la carte graphique. Cette portion consiste en l'utilisation des *shaders*. Le but a donc été de trouver une tâche qui se parallélise bien et que l'on pourrait transférer à un autre processeur. La partie évidente à paralléliser étant le calcul graphique, celle-ci a déjà été créée par d'autres programmeurs lors de la création des cartes graphiques. Cependant une partie qui serait intéressante à paralléliser est le moteur de collision. Ce moteur a donc été implémenté à l'aide d'OpenCL qui est un outil de programmation permettant de paralléliser des calculs sur différents types de processeurs.

Cette section se divise donc en deux parties. La première a pour but d'introduire l'utilisation d'OpenCL et le principe de fonctionnement. La deuxième section se concentre sur l'intérêt d'OpenCL pour le moteur de collision et comment celui-ci est implémenté.

3.2.1 OpenCL

OpenCL est une combinaison d'un API d'un langage dérivé du C permettant de communiquer avec différents processeurs dans le but de paralléliser certaines tâches sur ceux-ci. Il s'agit d'un outil qui a été développé par Khronos Group qui est aussi le développeur d'OpenGL. Il y a donc plusieurs similarités entre les deux et même un outil de partage de données entre les deux API lorsqu'ils sont utilisés sur le même processeur. OpenCL est capable de paralléliser les calculs sur des processeurs de type CPU, GPU, FPGA, ARM, etc. Il fonctionne pour Intel, AMD, NVidia, Altera, Xilinx, Qualcom et autres. Bien

sûr celui-ci ne fonctionne que sur les versions récentes de ces processeurs. De plus, certains manufacturiers de processeurs ne le supportent que depuis peu de temps. Il est donc bien nécessaire de vérifier la compatibilité avant d'acheter une architecture pour l'utiliser avec OpenCL.

OpenCL suppose qu'il y a plus d'un processeur dans l'architecture matérielle utilisée. Un des processeurs est considéré comme l'hôte (*Host*) et les autres sont des périphériques (*Devices*). Par contre, dans le cas de l'utilisation d'un CPU multi-cœurs, un des cœurs sera l'*Host* et les autres seront considérés comme un *Device* ayant plusieurs cœurs. L'API est utilisée sur l'Host afin de communiquer avec le ou les *Devices*. L'*Host* est donc l'unité de coordination des tâches. Un langage dérivé du C est utilisé pour écrire le ou les noyaux (*Kernels*) dans un fichier d'extension « .cl ». Ceux-ci sont exécutés sur les différents *Devices*. Le *Kernel* comme un *shader*, est exécuté un certain nombre de fois en parallèle sur le *Device*. Le processus d'utilisation se divise donc en deux étapes. La première est l'initialisation et la deuxième est l'exécution.

L'initialisation commence donc par trouver les différents *Devices* que l'on désire utiliser. Ensuite un contexte est créé à partir de la liste de *Devices* trouvés à l'étape précédente. Puis le ou les programmes sont créés et compilés. Les programmes sont les fichiers d'extension « .cl ». À noter que chacun de ces fichiers peut contenir plus d'un *Kernel* puisque celui-ci est en quelque sorte une fonction. Finalement, il y a création des *Kernels* à utiliser puis des différents *buffers* permettant de transférer des données avec ces *Kernels*.

La phase exécution pourra s'exécuter plus d'une fois après l'initialisation. La première étape consiste à écrire les données dans les *buffers* d'entrée. Ces données sont directement envoyées dans la mémoire du *Devices* afin d'accélérer la vitesse de lecture. Ensuite les

différents *buffers* d'entrée et de sortie sont assignés aux différents paramètres des *Kernels*. Puis le ou les *Kernels* sont démarrés. Ce processus nécessite de spécifier les dimensions de celui-ci. Les dimensions représentent le nombre de fois qu'il exécute en parallèle. Finalement, il s'agit de lire les différents *buffers* de sortie pour faire le traitement sur ces données.

Les *Kernels* sont codés comme une fonction en C, mais avec quelques mots réservés en plus. Il faut aussi considérer le fait que ce code s'exécute plusieurs fois en parallèle. Il faut donc s'assurer que plusieurs instances du *kernels* ne viennent pas écrire au même endroit en même temps ou qu'elles viennent lire en même temps qu'un autre écrit. Des fonctions internes à OpenCL permettent de retrouver l'ID de l'instance du kernel. Cette ID est utilisée comme indice afin de partager l'espace mémoire à utiliser et la tâche à accomplir. La figure 3.12 illustre un exemple simple de *Kernel*.

```
// AOC kernel demonstrating device-side printf call

__kernel void hello_world(int thread_id_from_which_to_print_message)
{
  // Get index of the work item
  unsigned thread_id = get_global_id(0);

  if(thread_id == thread_id_from_which_to_print_message) {
    printf("Thread #%u: Hello from Altera's OpenCL Compiler!\n", thread_id);
  }
}
```

Figure 3.12 Hello World OpenCL par Altera [24]

Cet exemple simple permet de créer plusieurs instances d'un *Kernel* et de demander à l'une d'entre elles d'écrire un message dans la console. Le paramètre du *Kernel* est simplement l'indice de l'instance qui doit afficher le message. La première ligne trouve l'ID d'instance en cours puis celui-ci est comparé au paramètre d'entrée afin de décider si cette

instance doit écrire dans la console. Il faut toutefois faire attention à l'utilisation du « *printf* » du côté du *Kernel*. Si celui-ci crée un million d'instances et que chacune d'entre elles utilise cette fonction, cela va causer un immense ralentissement de l'exécution et une console impossible à déchiffrer. Le « *printf* » n'est donc en général utilisé que pour faire du débogage.

3.2.2 Implémentation d'un moteur de collision

Le moteur de collision a d'abord été entièrement construit sur CPU à l'aide de la représentation AABB. Par la suite, il a été amélioré afin d'utiliser la représentation OBB. Finalement, le calcul de collision a été transféré vers le matériel à l'aide d'OpenCL. Le moteur de collision est construit sur trois classes. Les trois classes sont « collisionDetector », « collision_OpenCL » et « OBB ». La première permet de calculer les OBB afin de gérer la détection de collision. La deuxième est utilisée à l'intérieur de la classe « collisionDetector » et permet de faire la gestion de la communication avec OpenCL. La dernière est l'objet permettant de contenir les informations de chacun des OBB.

Afin de bien saisir, il faut comprendre la représentation des OBB comme le montre la figure 3.13.

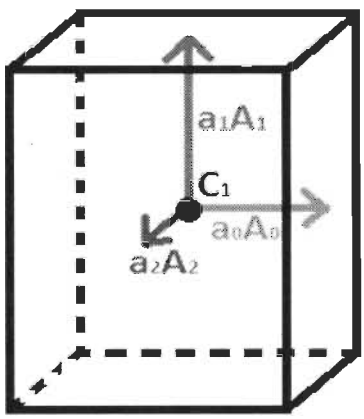

Figure 3.13 Représentation des OBB

Cette boîte englobe toute la portion du modèle que l'on désire modéliser par l'OBB. Elle est représentée par un centre, par des dimensions d'axes et par des directions d'axes unitaires. Le tout est représenté sous la forme suivante :

$$Centre = C_1 = [C.x \quad C.y \quad C.z] \ (15)$$

$$Longueur\ des\ axes = a = [a_0 \quad a_1 \quad a_2] \ (16)$$

$$Axes\ unitaires = A = \begin{bmatrix} A_0.x & A_1.x & A_2.x \\ A_0.y & A_1.y & A_2.y \\ A_0.z & A_1.z & A_2.z \end{bmatrix} \ (17)$$

Le tout est donc représenté à l'aide de deux vecteurs et d'une matrice. C'est cela qui est implémenté dans la classe « OBB ». Elle est simplement munie d'un constructeur vide afin de pouvoir initialiser un tableau de cette classe. Il y aura un objet par OBB à modéliser. Il y a une méthode permettant d'initialiser les paramètres de l'OBB, c'est-à-dire les deux vecteurs et la matrice. Il y a aussi des accesseurs pour ces différents paramètres. Finalement il y a une fonction permettant d'afficher l'OBB à l'écran. Celle-ci est utilisée simplement dans le but de faire des tests de collision et débogage.

La classe « collisionDetector » permet d'initialiser les OBB et vient gérer la détection de collision. Le constructeur commence par ouvrir le modèle 3D et trouve le nombre de *mesh* afin de créer le tableau d'OBB. Ensuite la méthode « *charger* » est utilisée afin de calculer les OBB du modèle 3D. Cela se fait en commençant par trouver la moyenne du *mesh* permettant de trouver le centre C_1. Cette partie est assez simple, il suffit de parcourir chacun des sommets de celui-ci. Ensuite il faut trouver les axes de l'OBB. Cela se fait à l'aide de l'analyse en composante principale (PCA). Pour faire le PCA il faut la matrice de covariances des coordonnées des sommets du mesh. Il faut donc trouver la distribution des sommets sur

les combinaisons des axes *X*, *Y* et *Z*. Cette matrice trois par trois est symétrique, il n'est donc pas nécessaire de calculer tous les termes de celle-ci. La moyenne trouvée à l'étape précédente est utilisée pour calculer cette matrice. Une fois cette matrice trouvée, il suffit de trouver les vecteurs propres de celle-ci. Comme cette matrice est une trois par trois, il y aura trois vecteurs propres. Ces vecteurs sont les axes unitaires principaux du *mesh* permettant de minimiser les dimensions de l'OBB. Si la boîte est minimisée, cela veut dire une représentation plus précise du modèle pour la détection de collision. Les vecteurs propres étant assez complexes à calculer numériquement, la librairie « Eigen » a été utilisée afin de les calculer. Une fois le système d'axes trouvé, les sommets du *mesh* sont balayés afin de trouver le minimum et maximum sur chacun des axes. Ces valeurs permettent de trouver les dimensions de l'OBB sur celle-ci. Finalement, les valeurs minimale et maximale sont utilisées avec les axes de l'OBB pour trouver le centre dans le référentiel monde et non dans le système d'axe de l'OBB. Une fois tous les OBB calculés, ils sont envoyés à la classe « collision_OpenCL » afin de l'initialiser. La figure 3.14 illustre les OBB autour du modèle une fois calculé. On voit que trois OBB ont été calculés, car le modèle est divisé en trois *meshs*. Dans le contexte de cette recherche, il y a un OBB pour les cheveux, un pour les yeux et un pour le corps.

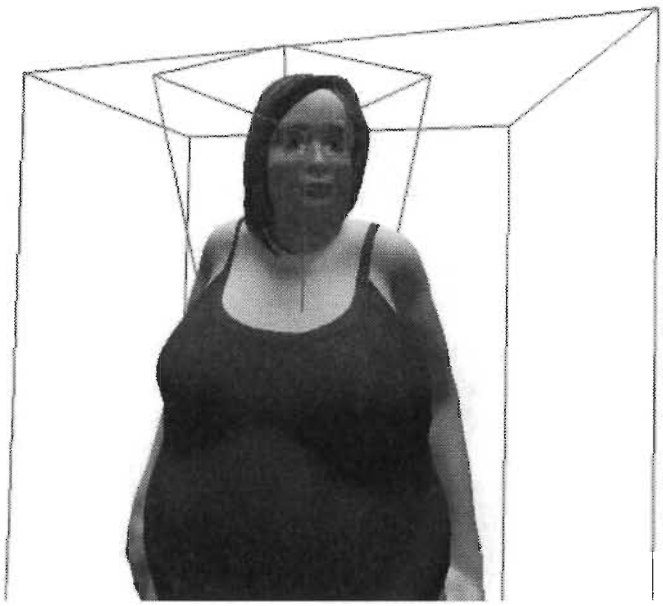

Figure 3.14 Exemple d'OBB

Les méthodes restantes sont les méthodes « collisionOBB » et « getRectified ». La première envoie les informations à la classe « collision_OpenCL » pour le calcul de collision. Elle vient aussi trouver la position de la caméra corrigée dans le cas d'une collision. Pour l'instant, la correction n'est pas implémentée. Donc, la position corrigée est simplement la dernière position où il n'y avait pas de collision. La méthode « getRectified » comme le nom laisse entendre retourne la position corrigée afin de la mettre à jour dans la classe « caméra ».

La classe « collision_OpenCL » initialise OpenCL, se charge de remplir les *buffers* et demande l'exécution du *Kernel*. Cette classe est munie de trois méthodes publiques que voici :

- `int init(int numOBB, OBB* obb);`
- int execute(vec3 pos, mat3 axes);
- `bool readOutput();`

La méthode « *init* » est utilisée pour faire l'initialisation d'OpenCL telle qu'expliquée dans la section 3.2.1. Elle vient aussi écrire dans l'un des *buffers*. Le *Kernel* de détection de collision nécessite trois *buffers*, deux d'entrée et un de sortie. Un des buffers d'entrée représente tous les OBB du modèle 3D. Ceux-ci ne changent pas, c'est pourquoi ils sont envoyés une seule fois à l'initialisation d'OpenCL afin d'éviter des transferts inutiles de données.

La méthode « *execute* » remplit le deuxième buffer qui représente l'OBB de la caméra. Celui-ci est envoyé à chaque itération puisque la caméra se déplace dans l'environnement. Puis elle démarre l'exécution du *Kernel* en lui donnant les dimensions de celui-ci. La dimension est le nombre d'OBB dans le modèle 3D.

Finalement, il y a la méthode « *readOutput* » qui permet de retrouver la valeur du booléen indiquant s'il y a collision ou non.

Le Kernel est le cœur du calcul de collision. Afin de bien comprendre celui-ci il faut comprendre les équations qui suivent.

$$C = A^T * B \quad (18)$$

Cette première équation est nécessaire pour le calcul des axes de collision. « A » et « B » sont les systèmes d'axes des deux OBB entre lesquels on désire détecter s'il y a collision. « C » est donc la matrice qui permet la transformation d'un repère vers l'autre. En temps normal, une inversion de matrice serait nécessaire, mais comme « I » est une matrice orthonormée, la transposée est égale à l'inverse.

Dans les équations suivantes, le vecteur « D » est le vecteur entre le centre des deux OBB en collision. Afin de détecter une collision, il faut détecter la collision sur tous les axes de

collision possible. Il y en a 15 en tout. Les six premiers sont les systèmes d'axe des deux OBB. Les neuf autres sont les différents axes indépendants résultant du produit vectoriel des six premiers axes. Le tableau 3.1 énumère les différentes équations nécessaires au calcul de collision.

Tableau 3.1 Dimensions des axes de collision [18]

Axes	R_A	R_B	R
$\boldsymbol{A_0}$	a_0	$b_0\lvert c_{00}\rvert + b_1\lvert c_{01}\rvert + b_2\lvert c_{02}\rvert$	$\lvert \boldsymbol{A_0}\cdot\boldsymbol{D}\rvert$
$\boldsymbol{A_1}$	a_1	$b_0\lvert c_{10}\rvert + b_1\lvert c_{11}\rvert + b_2\lvert c_{12}\rvert$	$\lvert \boldsymbol{A_1}\cdot\boldsymbol{D}\rvert$
$\boldsymbol{A_2}$	a_2	$b_0\lvert c_{20}\rvert + b_1\lvert c_{21}\rvert + b_2\lvert c_{22}\rvert$	$\lvert \boldsymbol{A_2}\cdot\boldsymbol{D}\rvert$
$\boldsymbol{B_0}$	$a_0\lvert c_{00}\rvert + a_1\lvert c_{10}\rvert + a_2\lvert c_{20}\rvert$	b_0	$\lvert \boldsymbol{B_0}\cdot\boldsymbol{D}\rvert$
$\boldsymbol{B_1}$	$a_0\lvert c_{01}\rvert + a_1\lvert c_{11}\rvert + a_2\lvert c_{21}\rvert$	b_1	$\lvert \boldsymbol{B_1}\cdot\boldsymbol{D}\rvert$
$\boldsymbol{B_2}$	$a_0\lvert c_{02}\rvert + a_1\lvert c_{12}\rvert + a_2\lvert c_{22}\rvert$	b_2	$\lvert \boldsymbol{B_2}\cdot\boldsymbol{D}\rvert$
$\boldsymbol{A_0}\times\boldsymbol{B_0}$	$a_1\lvert c_{20}\rvert + a_2\lvert c_{10}\rvert$	$b_1\lvert c_{02}\rvert + b_2\lvert c_{01}\rvert$	$\lvert c_{10}\boldsymbol{A_2}\cdot\boldsymbol{D} - c_{20}\boldsymbol{A_1}\cdot\boldsymbol{D}\rvert$
$\boldsymbol{A_0}\times\boldsymbol{B_1}$	$a_1\lvert c_{21}\rvert + a_2\lvert c_{11}\rvert$	$b_0\lvert c_{02}\rvert + b_2\lvert c_{00}\rvert$	$\lvert c_{11}\boldsymbol{A_2}\cdot\boldsymbol{D} - c_{21}\boldsymbol{A_1}\cdot\boldsymbol{D}\rvert$
$\boldsymbol{A_0}\times\boldsymbol{B_2}$	$a_1\lvert c_{22}\rvert + a_2\lvert c_{12}\rvert$	$b_0\lvert c_{01}\rvert + b_1\lvert c_{00}\rvert$	$\lvert c_{12}\boldsymbol{A_2}\cdot\boldsymbol{D} - c_{22}\boldsymbol{A_1}\cdot\boldsymbol{D}\rvert$
$\boldsymbol{A_1}\times\boldsymbol{B_0}$	$a_0\lvert c_{20}\rvert + a_2\lvert c_{00}\rvert$	$b_1\lvert c_{12}\rvert + b_2\lvert c_{11}\rvert$	$\lvert c_{20}\boldsymbol{A_0}\cdot\boldsymbol{D} - c_{00}\boldsymbol{A_2}\cdot\boldsymbol{D}\rvert$
$\boldsymbol{A_1}\times\boldsymbol{B_0}$	$a_0\lvert c_{21}\rvert + a_2\lvert c_{01}\rvert$	$b_0\lvert c_{12}\rvert + b_2\lvert c_{10}\rvert$	$\lvert c_{21}\boldsymbol{A_0}\cdot\boldsymbol{D} - c_{01}\boldsymbol{A_2}\cdot\boldsymbol{D}\rvert$
$\boldsymbol{A_1}\times\boldsymbol{B_0}$	$a_0\lvert c_{22}\rvert + a_2\lvert c_{02}\rvert$	$b_0\lvert c_{11}\rvert + b_1\lvert c_{10}\rvert$	$\lvert c_{22}\boldsymbol{A_0}\cdot\boldsymbol{D} - c_{02}\boldsymbol{A_2}\cdot\boldsymbol{D}\rvert$
$\boldsymbol{A_2}\times\boldsymbol{B_0}$	$a_0\lvert c_{10}\rvert + a_1\lvert c_{00}\rvert$	$b_1\lvert c_{22}\rvert + b_2\lvert c_{21}\rvert$	$\lvert c_{00}\boldsymbol{A_1}\cdot\boldsymbol{D} - c_{10}\boldsymbol{A_0}\cdot\boldsymbol{D}\rvert$
$\boldsymbol{A_2}\times\boldsymbol{B_0}$	$a_0\lvert c_{11}\rvert + a_1\lvert c_{01}\rvert$	$b_0\lvert c_{22}\rvert + b_2\lvert c_{20}\rvert$	$\lvert c_{01}\boldsymbol{A_1}\cdot\boldsymbol{D} - c_{11}\boldsymbol{A_0}\cdot\boldsymbol{D}\rvert$
$\boldsymbol{A_2}\times\boldsymbol{B_0}$	$a_0\lvert c_{12}\rvert + a_1\lvert c_{02}\rvert$	$b_0\lvert c_{21}\rvert + b_1\lvert c_{20}\rvert$	$\lvert c_{02}\boldsymbol{A_1}\cdot\boldsymbol{D} - c_{12}\boldsymbol{A_0}\cdot\boldsymbol{D}\rvert$

Les colonnes R_A et R_A sont respectivement les rayons des OBB « A » et « B » projetés sur les différents axes de collision. La colonne R est la distance entre le centre des OBB projeté sur les axes de collision. La condition de collision est calculée à l'aide de cette équation :

$$R \geq R_A + R_B \quad (19)$$

Cette condition est vérifiée sur chacun des axes de collision et si dans l'un ou l'autre des cas elle est vraie, il n'y pas de collision. Il s'agit en fait de vérifier que la somme des rayons des OBB est plus petite que la distance centre à centre entre ceux-ci.

Le code du *Kernel* commence donc par calculer le vecteur « D » et la matrice « C ». Il calcule ensuite les 15 sommes des colonnes R_A et R_B. Puis, il calcule les 15 dimensions de la colonne R. Finalement il boucle sur ces deux tableaux pour vérifier la condition en (19).

3.3 Conclusion

La programmation de moteur de réalité virtuelle a été un travail d'envergure nécessitant le partage de travail dans différentes classes. L'utilisation de classes est primordiale dans un projet de cette envergure, car la coordination des différentes tâches deviendrait presque impossible sans celle-ci. De plus, le tout deviendrait illisible pour quelqu'un d'externe.

Il y a donc eu la programmation de deux sous-moteurs à l'intérieur de ce projet. Le premier est le moteur graphique qui est le plus complexe dans ce cas. Le deuxième est le moteur de collision qui fait la gestion du phénomène physique le plus important dans cette simulation. Cette partie est un peu moins complexe due au fait que ce n'est pas un moteur physique complet.

Le prochain chapitre portera donc sur l'application de ce moteur. Il présentera l'architecture de l'implémentation matérielle, les résultats et les performances.

Chapitre 4 - Application de la RV

Le moteur de RV a été implémenté en premier lieu sur un ordinateur de bureau. Le développement a été fait en plusieurs étapes passant de l'affichage d'une simple fenêtre avec un triangle à un moteur capable d'ouvrir le continuum de silhouettes utilisé par le Loricorps. Par la suite, l'ajout d'un moteur de collision a été fait afin d'éviter que les patients puissent passer à travers des personnages du continuum. Cela fut la première version stable pouvant remplacer le système préexistant du Loricorps. Cette version (1.0) est munie d'une interface graphique pour faciliter l'utilisation pour les chercheurs. Cette première version fonctionne à l'aide de la détection de collision se basant sur la représentation AABB. Celle-ci est capable de trouver la normale à la collision. Cela permet d'annuler le déplacement dans la direction de la normale et de glisser sur la surface en collision. Ce qui donne une expérience assez réaliste.

Une deuxième version du moteur de RV (1.1) a été l'implémentation de la représentation OBB pour le moteur de collision. Cela a entrainé une modification majeure du moteur de collision. Cela est dû au fait que le calcul des OBB est très différent de celui des AABB. La seule différence est que chaque OBB a son propre repère d'axe par rapport à l'AABB qui utilise le même repère d'axe que le monde 3D. Ceci entraine une plus grande complexité de calcul des autres paramètres. L'ajout d'OBB augmente aussi grandement la complexité des calculs de détection de collision. Dans le cas des AABB il y a seulement trois axes de collision possible alors que dans le cas des OBB on passe à 15 axes de collision. De plus

ceux-ci nécessitent des projections de distance dues au fait qu'il y a combinaison de deux repères d'axes pour chaque vérification de collision. Cette version ne retourne plus la normale à la collision puisque celle-ci se calcule différemment et n'a toujours pas été implémentée. Le tout résulte en une détection de collision plus précise, mais qui a tendance à bloquer tant que le déplacement ne se fait pas dans une direction opposée à la collision. En fait, il manque le glissement sur la surface.

Une troisième version (1.2) a aussi été faite. Celle-ci a consisté à transférer le calcul de détection de collision vers un processeur en parallèle à l'aide d'OpenCL. Cela a donc nécessité l'ajout d'une nouvelle classe afin d'implémenter la communication OpenCL. Il a aussi été nécessaire de faire un formatage des données afin de pouvoir les envoyer à l'intérieur des *buffers* OpenCL. Ceux-ci ne permettent pas d'envoyer un objet, mais seulement des tableaux de données. La version précédente se sert des pointeurs sur les objets OBB afin de faire les calculs. Les OBB sont donc tous transformés sous forme d'un tableau 2D. Ces différents tableaux sont ensuite tous mis à la suite les uns des autres dans un *buffer* pour l'envoi au *Device* OpenCL. Ce long buffer est ensuite divisé en plus petites portions du côté du *Device* afin de faire le partage de calcul. Cette division est en fait une division par OBB.

Finalement une dernière modification a été faite afin de supporter les derniers modèles 3D créés pour le continuum adolescent. Ces modèles 3D ont été conçus à l'aide de *bump maps* qui permettent de donner du relief en jouant sur des effets de lumières. Cela permet d'augmenter le détail du modèle sans l'ajout de polygones. Ceci n'était pas supporté par le moteur au départ, car ce n'était pas présent dans les modèles originaux. Étant donné que la version la plus stable est la première, l'ajout du support des *bump maps* a donc été fait dans

celle-ci (1.0.1). Il est important de noter que les versions deux et trois du moteur ne supportent pas les *bump maps*.

Les sections de ce chapitre portent sur les résultats d'implémentation. La première porte sur la structure de la plateforme utilisée pour l'implémentation. La deuxième fait état des performances du moteur sur différentes plateformes et d'une comparaison avec l'ancien système utilisé par le Loricorps. La troisième porte sur les résultats de l'implémentation matérielle du moteur de collision avec OpenCL. Finalement la dernière présente les résultats pour l'application au traitement des TCA.

4.1 Introduction de l'environnement matériel

Le moteur à été implémenté sur quatre plateformes différentes. La première, sur laquelle tout le développement a été fait, est un ordinateur de type station de travail. La deuxième plateforme sur laquelle le moteur a été testé est un ordinateur de bureau de moyenne haut de gamme. Finalement le projet a été transféré sur un laptop très haut de gamme conçu pour les jeux vidéo.

La station de travail de départ est le HP Workstation Z640. Il s'agit d'un ordinateur très performant et capable de faire du calcul parallèle sur différents processeurs. L'ordinateur a été modifié en changeant la carte graphique et en ajoutant un FPGA. Le tout a été fait par la CMC microsystème. Les spécifications de la plateforme finale sont donc les suivantes :

- CPU Intel Xeon E5-2637 v3
    - Horloge de 3.5 GHz
    - 4 cœurs
    - 8 threads

- RAM de 32 Go DDR3
- GPU NVidia Quadro K4200
    - 1344 cœurs cuda
    - VRAM de 4 Go DDR5
    - Vitesse d'accès mémoire 173 Go/s
    - Interface mémoire 256 bits
- FPGA Nallatech 395 avec Statix V A7 d'Altera
    - 2 fois 4 GB SDRAM
    - Interface mémoire 72 bits
    - PCI Express 3.0 (10 Go/sec)

Puisque l'application est reliée au graphisme, c'est surtout le GPU que l'on désire vérifier. Bien sûr la gestion du moteur se fait sur le CPU il faut donc aussi qu'il soit assez rapide pour faire la coordination des tâches. On constate donc que le CPU dans cette configuration est très rapide comparé à d'autres. Le multi cœurs n'a pas été utilisé dans le moteur, mais celui-ci pourrait être très bénéfique pour paralléliser la coordination. Le GPU quant à lui à une vitesse d'accès mémoire hors du commun et un nombre de cœurs cuda assez élevé. Le nombre de ces cœurs contrôle le degré de parallélisations dans le *shader*, ce qui a un impact sur la vitesse de rendu. La vitesse d'accès mémoire aura un impact sur le rendu, car elle restreint la vitesse d'accès aux données par le *shader*.

Ensuite, dans le but de comparer les performances du moteur, il a été testé sur mon ordinateur de bureau personnel. Celui-ci est typique d'un utilisateur moyen qui utilise son ordinateur pour le jeu vidéo. Les spécifications de celui-ci sont les suivantes :

- CPU Intel I5-2500K

    - Horloge de 3.3 GHz
    - 4 cœurs
    - 4 threads
- RAM de 8 Go DDR3
- GPU NVidia GTX 750 Ti de Gigabyte Technology
    - 640 cœurs cuda
    - VRAM de 2 Go DDR5
    - Vitesse d'accès mémoire 86,4 Go/s
    - Interface mémoire 128 bits

Si on compare à l'ordinateur précédent le CPU est un peu plus lent. Le GPU a aussi moins de cœurs et une vitesse d'accès mémoire nettement plus lente. Ces deux faits auront un impact sur la vitesse de coordination et la vitesse de rendu respectivement.

Finalement, dans le but rendre le moteur transportable pour les démonstrations à l'extérieur, celui-ci a été transféré sur un ordinateur portable. Cet ordinateur en est un de très haut de gamme conçu pour le jeu vidéo. Les spécifications de celui-ci sont les suivantes :

- CPU Intel I7-6820HK
    - Horloge de 2.7 GHz
    - 4 cœurs
    - 8 threads
- RAM de 32 Go DDR4
- GPU NVidia GTX 980M
    - 1536 cœurs cuda
    - VRAM de 8 Go DDR5

- Vitesse d'accès mémoire 160 Go/s
- Interface mémoire 256 bits

Celui-ci à un CPU un peu plus lent que les autres puisqu'il s'agit d'un portable et que la gestion de chaleur dans ceux-ci est plus difficile dû au manque d'espace. Cependant, le GPU est très puissant. Il s'agit d'un des plus puissants sur le marché utilisable dans un ordinateur portable. Celui-ci à plus de cœurs que les autres et une vitesse d'accès mémoire normale pour les GPU haut de gamme.

4.2 Performances logicielles

Cette section se divise en deux parties. La première traite des performances du moteur de RV sur différentes plateformes. La deuxième fait état des différences entre le moteur de RV créé dans ce projet et celui précédemment utilisé par le Loricorps.

4.2.1 Performance sur différentes plateformes

Dans le but de tester les performances du moteur, la version 1.0.1 a été testée sur les différentes plateformes présentées dans la section 4.1. Cette version a été testée en utilisant deux modèles 3D différents. Dans le premier test, il s'agit du continuum de femmes utilisé par le Loricorps. Dans le deuxième cas, le modèle de continuum adolescent a été utilisé. Celui-ci a un peu plus de vertex que le précédent, ce qui ralentit le rendu. Il fait aussi l'utilisation des *bump map* qui ne sont pas utilisées dans premier cas. Les *bump map* complexifient les calculs de lumières dans le *shader*, ce qui ralentit un peu le processus de rendu. Le tableau 4.1 montre les résultats de ces tests.

Tableau 4.1 Comparaison des performances sur différentes plateformes

Ordinateur	FPS	
	Femmes	Adolescentes
Station de travail	440	340
Ordinateur de bureau moyen haut de gamme	310	270
Ordinateur portable très haut de gamme	650	550

Dans ces tests, la synchronisation verticale de l'écran (Vsync) a été désactivée. Le Vsync limite la vitesse de rendu à la vitesse de rafraîchissement de l'écran afin d'éviter les déchirements d'image. Celui-ci a donc été désactivé afin de pouvoir vérifier à quelle vitesse le moteur est capable de faire le rendu.

On constate donc que la vitesse de rendu générée par le moteur est nettement supérieure à la vitesse désirée qui est de 60 FPS. On remarque un ralentissement dans le modèle avec les adolescentes, mais en gardant une vitesse encore largement au-delà de la valeur désirée. Considérant que même l'ordinateur moyen haut de gamme est capable de fournir, on peut tout de même considérer que le moteur est assez performant pour les besoins du Loricorps. Il y a même encore de la marge afin de complexifier les modèles 3D ou bien le moteur afin d'améliorer la qualité du rendu.

Il faut tout de même aussi considérer que dans le cas d'un moteur de RV, contrairement à un jeu vidéo standard, la vitesse de rendu va automatiquement être plus lente. Cela est dû au fait que la simulation est affichée dans deux fenêtres avec une déformation différente. Une fenêtre pour le casque et une fenêtre pour l'écran de l'ordinateur. De plus, puisque l'écran dans le casque est près des yeux il faut faire deux fois le rendu de la scène 3D. Une fois pour

chaque œil avec une position de caméra différente. Cela permet de simuler une vision 3D comme dans le cas de la télévision 3D. Par contre, cela a un impact non négligeable sur les performances.

4.2.2 Comparaison ancien et nouveau moteur

Le moteur de RV créé dans ce projet se compare assez bien avec l'ancien moteur. Le nouveau moteur permet d'ouvrir les mêmes modèles 3D que le précédent. La différence majeure est que l'on a accès au code source du nouveau ce qui permet l'ajout de nouveaux modèles 3D au moteur très simple. De plus toute amélioration du moteur en lui-même est possible puisque le code source de celui-ci est disponible pour modification. Celui-ci peut aussi être installé sur presque n'importe quel ordinateur.

En termes de fonctionnalités, il y a eu quelques améliorations. La première est l'ajout de la vue stéréoscopique. Dans la version précédente, l'image pour chacun des yeux est la même, ce qui enlève la vue 3D et peut créer un malaise, car les yeux ne sont pas supposés en général, voir exactement la même chose. Une amélioration majeure est le changement du casque de RV. L'Oculus est beaucoup plus immersif que le précédent. Il y a aussi l'ajout de la correction de la déformation de l'image par les lentilles. Le moteur précédent ne prend pas cela en charge, probablement dû au fait que dans le premier casque la déformation n'est pas très visible. L'ajout de la mécanique du cou a aussi été ajouté. Cela augmente le réalisme des déplacements de la tête. Un exemple de cela est lorsque l'on penche la tête vers l'avant. Dans un cas sans mécanique du cou, la caméra va simplement tourner pour regarder vers le sol. Cela donne l'effet que les yeux sont tournés à l'intérieur de leur orbite. Dans le cas avec la mécanique du cou, la caméra va avancer et descendre, car c'est la tête qui tourne autour de la nuque. Finalement la dernière amélioration est le moteur de collision. Celui-ci développé

dans ce projet est plus stable que le précédent. Dans la version précédente lors d'une collision la caméra a tendance à trembler sur la surface de collision.

4.3 Résultats de l'implémentation matérielle

L'implémentation matérielle consistait à transférer les calculs du moteur de collision sur un autre processeur afin de les paralléliser. Initialement ce transfert devait se faire vers le FPGA de la station de travail. Cependant, nous avons eu des problèmes de licences pour l'outil *Altera OpenCL* (AOCL) qui est nécessaire pour exécuter un code OpenCL sur un FPGA Altera. Puisque le code d'un *Kernel* OpenCL est essentiellement le même, peu importe le processeur utilisé comme *Device*, il a été testé sur le GPU. Les tâches de rendu et de détection de collision se partage le GPU. Cela n'a pas trop d'impact, car le rendu ne peut se faire tant que la détection de collision n'est pas terminée. L'algorithme de détection de collision OBB a donc été testé en premier lieu entièrement en C++ sur CPU sans mise en parallèle. Ensuite l'algorithme a été testé à l'aide d'OpenCL en transférant les OBB à la carte graphique. Le tableau 4.2 montre donc les résultats de ces simulations.

Tableau 4.2 Comparaison des temps de calcul avec et sans OpenCL

Nombre d'objets		35 objets	2175 objets
Sans OpenCL	Temps de calcul sur CPU	<1ms	10ms
Avec OpenCL	Temps de calcul sur GPU	12µs	15µs
	Temps total incluant communication	500µs	550µs

Les tests ont été faits à l'aide de deux modèles 3D. Le premier est un des continuums de silhouette contenant 35 objets. Le deuxième est un modèle de pièce débarra contenant une

quantité énorme d'objets dépassant les 2000. Les valeurs de temps dans les tableaux sont seulement celles reliées au calcul de collision. Le temps de rendu n'est pas inclus. Le temps pour les simulations avec OpenCL a été divisé en deux afin de bien visualiser le gain. Le premier est le temps passé à faire les calculs sur le GPU. Cependant, on ne peut pas tenir compte que de ce temps, car il y a un temps de communication qui s'ajoute à celui-ci dans le processus OpenCL. Ce temps de communication représente le temps d'envoi des données de positionnement de la caméra vers la mémoire du *Device* et le temps de retour du résultat vers la m*émoire* RAM. On remarque donc un gain énorme sur le temps de calcul dû à la parallélisation des calculs. Puis même en ajoutant le temps de communication nécessaire à OpenCL on retrouve quand même un gain de performance non négligeable. Ce gain n'est pas très perceptible dans le cas de modèle 3D simple, mais devient très intéressant dans le cas d'un modèle plus complexe en termes du nombre d'objets. Cet outil permet donc de diminuer l'impact du moteur de collision sur les FPS. Ceci est très positif, car une diminution des FPS peut entraîner des cybers-malaises pour les utilisateurs de l'Oculus.

4.4 Application aux TCA

L'application au TCA se fait en enregistrant sur quelle portion du continuum le patient se concentre. Lors de la simulation, le patient peut se déplacer comme il le désire autour des différentes silhouettes du continuum qui lui est attribué. Les informations sur les déplacements et l'orientation de la tête sont toutes mesurées afin de faire le rendu de la scène. En même temps, ces informations sont enregistrées et utilisées afin de calculer quelle silhouette le patient à regarder et combien de temps. De plus, en augmentant la précision des calculs, il est possible de déterminer sur quelle partie du corps le patient s'est concentré.

Afin d'alléger les calculs lors du rendu, les objets regardés ne sont pas calculés lors de la simulation. Lors de la simulation seulement, les mesures brutes nécessaires au rendu, c'est-à-dire position et orientation de la caméra, sont enregistrées à l'intérieur d'un fichier *Comma-separated values* (.csv). Celui-ci est ensuite utilisé à l'intérieur d'un script Matlab qui permet de déterminer les objets qui ont été regardés. Les données enregistrées dans le fichier sont des valeurs de temps chacune associée à deux vecteurs. Le premier vecteur est la position de la caméra dans l'environnement et le deuxième est l'orientation de la caméra. Le script Matlab prend donc ces deux vecteurs pour définir une droite représentant la direction du regard du patient. Ensuite il calcule l'intersection entre cette droite et le plan où se trouve tous le continuum. Cela permet de définir la silhouette regardée par le patient. Du moment que l'intersection se trouve dans le rectangle du plan où est une silhouette, celle-ci est considérée comme étant regardée. La figure 4.1 montre un graphique représentant les résultats pour un fichier de simulation.

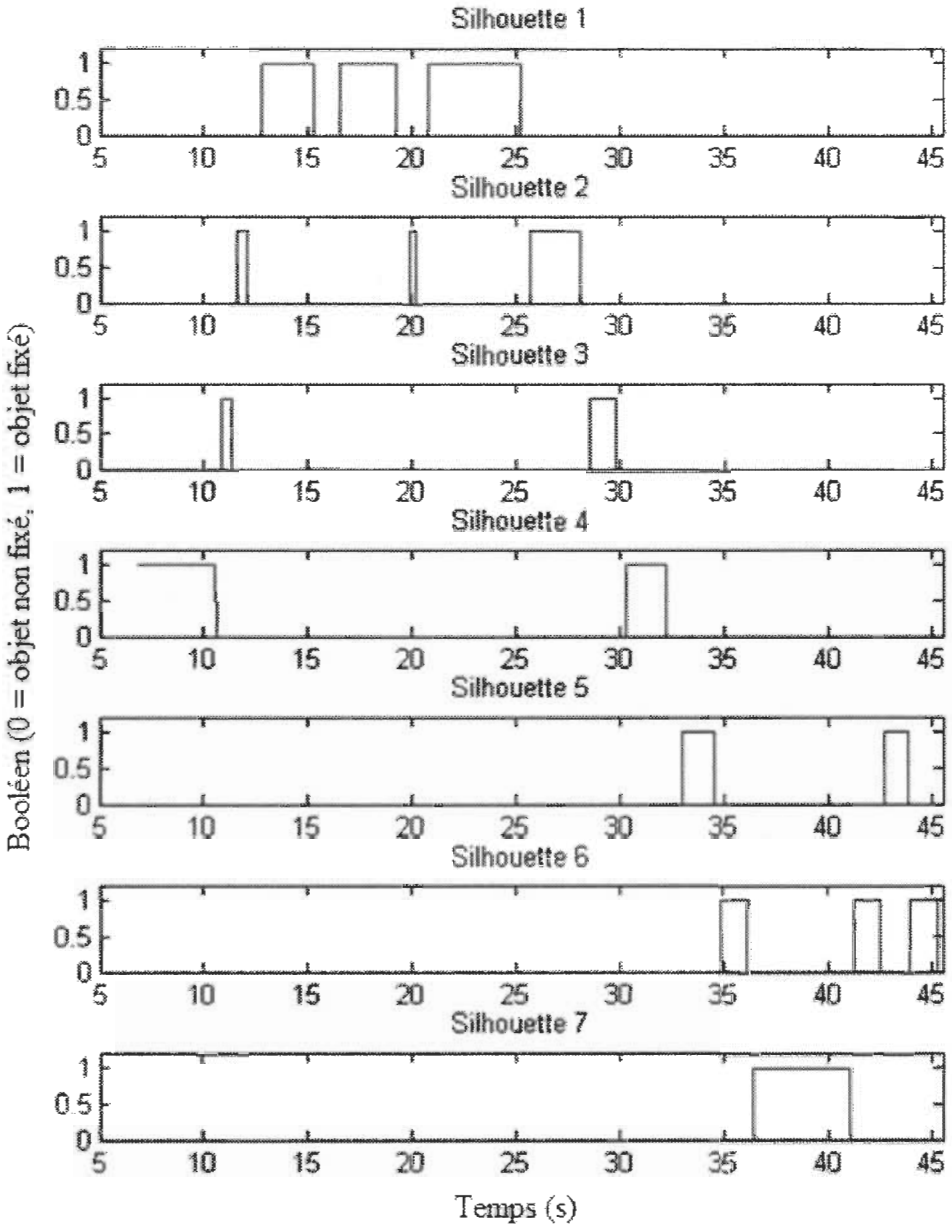

Figure 4.1 Silhouette regardée dans le temps

Lors de cette simulation, le patient a commencé devant la silhouette 4. Il s'est déplacé vers la silhouette 1 en passant devant les silhouettes 3 et 2. Il a ensuite tourné la tête dans différentes directions avant de se déplacer vers la silhouette 7. Il a ensuite tourné la tête pour regarder la silhouette 5 avant de ramener le regard sur la silhouette 6. Le parcours est connu, car je l'ai fait moi-même. On constate donc que ce graphique permet de revivre ce que le

patient a visualisé. Ceci permet donc au psychologue de faire une analyse du comportement du patient même après que celui-ci ait terminé la simulation.

Un autre graphique est calculé à l'aide du premier. Celui-ci permet simplement de ressortir les statistiques sur le temps passé par silhouette. La figure 4.2 montre la statistique associée à la figure 4.1.

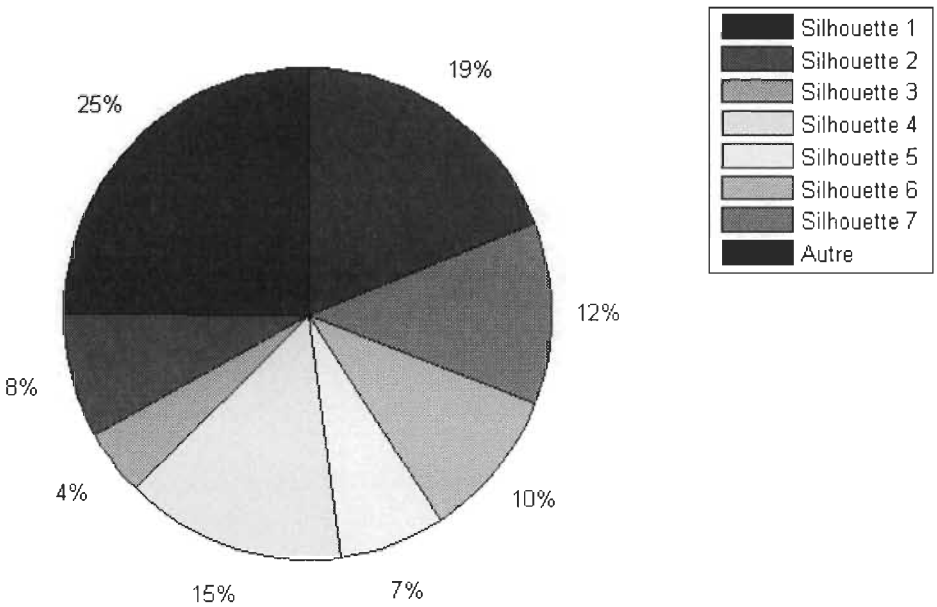

Figure 4.2 Répartition du temps par silhouette

Ce graphique est simplement une autre représentation des données permettant au psychologue de faire une analyse sur le patient.

4.5 Conclusion

Le moteur de RV a subi différentes itérations permettant de tester différentes architectures. La version 1.0.1 est la version finale utilisée, car elle est la plus stable. Les versions 1.2 et 1.3 nécessiteraient quelques modifications notamment au niveau du calcul de collision afin de le stabiliser.

Le moteur de RV a par la suite été appliqué sur différentes plateformes de moyen à très haut de gamme. Les performances étaient largement au-delà de celles désirées, dans tous les cas. Ceci a démontré que le moteur est assez optimisé pour fonctionner sur une vaste gamme de plateformes.

Le nouveau moteur de RV créé dans ce projet a été comparé à l'ancien et a permis de démontrer l'utilité du projet. Le nouveau moteur permet de faire les mêmes choses que l'ancien, mais avec de meilleures performances. De plus celui-ci est complètement ouvert, ce qui rend les futures modifications beaucoup plus simples.

Le transfert de calculs sur un autre processeur afin de le paralléliser a été fait dans la version 1.3 du moteur. Celui-ci a transféré le calcul de collision sur le GPU afin de le paralléliser. Les résultats ont démontré un gain en performance très intéressant permettant de mettre de l'avant l'intérêt du partage de tâches.

Chapitre 5 - Conclusion générale

Les objectifs de cette recherche étaient de se familiariser avec la structure des logiciels de réalité virtuelle ainsi que de se familiariser avec les forces et faiblesses des différents types de processeurs. Finalement, utiliser ces informations afin de proposer une solution permettant d'optimiser les performances d'un logiciel de RV en partageant les tâches sur ces différents types de processeurs.

Ces objectifs ont été atteints au cours de ce projet de maîtrise. La première partie m'a permis de développer un logiciel de RV appliqué au traitement de TCA et entièrement ouvert aux futures modifications. Celui-ci est maintenant plus performant, plus stable et à plus de fonctionnalité que celui utilisé précédemment. L'étude des différents types de processeurs a permis de constater qu'en général, toutes les tâches sont bien réparties entre le CPU et le GPU. Cependant, l'ajout d'un FPGA pour les calculs reliés au moteur physique permettrait d'alléger la tâche reliée au CPU et/ou GPU. Une grande partie des calculs physiques sont assez complexes et ont un grand potentiel pour le calcul parallèle. Un FPGA est donc parfait pour ce genre de calculs. Le GPU est aussi capable de faire du calcul parallèle, mais la structure de ces unités de calcul est optimisée pour le calcul graphique. Une preuve de concept a donc été faite en parallélisant le calcul de collision à l'aide d'OpenCL. Cette parallélisation a premièrement été testée sur GPU et a entraîné un énorme gain de calcul comparé au calcul sur CPU.

De plus, le développement de ce moteur de réalité virtuelle a permis l'ajout de plusieurs fonctionnalités qui n'étaient pas présentes dans la version précédemment utilisée par le Loricorps. Parmi ces ajouts, il y a la vue en stéréo, la correction de distorsion et d'aberration chromatique des lentilles, la mécanique des mouvements du cou et une amélioration de la stabilité du moteur de collision.

La prochaine étape sera de tester les performances de la RV en appliquant ce code OpenCL sur FPGA. Ceci sera testé dès que la licence du compilateur OpenCL du FPGA sera disponible.

Par la suite, le transfert du projet vers Linux serait à envisager afin de permettre de proposer une solution embarquée. Ce travail devrait être allégé dû au fait que le choix de librairies pour le projet ont été fait en conséquence de ce transfert.

Il serait aussi intéressant d'ajouter d'autres fonctionnalités au moteur de RV tels la gestion d'animation, l'occlusion lumineuse et autres.

Références

[1] American Psychiatric Association, Diagnostic and Statistical Manual of Mental Disorders, 4th Edition, Washington, DC: American psychiatric Association, 2000.

[2] C. Farrell, R. Shafra, et M. Lee, "Empirically evaluated treatments for body image disturbance: A review," European Eating Disorders Review, Vol. 14, pp. 289-300, 2006.

[3] Y. A. W. de Kort, W. A. IJsselsteijn, J. Kooijman et Schuurmans, "Virtual Laboratories: Comparability of Real and Virtual Environments for Environmental Psychology, " MIT Presence, Vol. 12, No. 4, pp. 360-373, Aout 2003.

[4] D R Pratt, M. Zyda, et K. Kelleher, "Virtual Reality: In the Mind of the Beholder," IEEE Computer Society Press, Vol. 28, No. 7, pp. 17-19, 1995. (Pratt et al., 1995)

[5] G. Riva, Virtual Reality in Neuro-Psycho-Physiology : Cognitive, clinical and methodological issues in assessment and rehabilitation, IOS Press, 1997.

[6] P. E. Garfinkel et D. M. Garner, Anorexia Nervosa: A Multidimensional Perspective, Brunner Mazel, 1982.

[7] W. Luk, T.K. Lee, J.R. Rice et P.Y.K. Cheung, "Reconfigurable Computing for Augmented Reality," IEEE Symposium on Field-Programmable Custom Computing Machines, pp. 136-145, 1999.

[8] T. Frikha, N. Ben Amor, K. Loukil, et M. Abid, "Virtual reality system on embedded platforms," IEEE 7th International Conference on Design & Technology of Integrated Systems in Nanoscale, pp.1-5, 2012.

[9] T. Frikha, N. Ben Amor, I. Benhlima, K. Loukil, et M. Abid, " Self adaptive augmented reality systems on FPGA," IEEE International Conference on Multimedia Computing and Systems, pp.23-27, 2012.

[10] A. Bochem, K.B. Kent, et R. Herpers, "FPGA Based Real-Time Tracking Approach with Validation of Precision and Performance," IEEE Euromicro Conference on Digital System Design (DSD), pp.122-127, 2012.

[11] Wei-Hao Chen, Bing-Yuan Huang, Syu-Ru Chen, Shih-Fan Yang, Chia-Cheng Lee, Dah-Shyong Yu, and Yuan-Hsiang Lin, "FPGA and Virtual Reality Based Minimally Invasive Surgery Training System," Asia-Pacific Workshop on FPGA Applications, pp.1-6, 2013.

[12] S. Hengstler, D. Prashanth, S. Fong, and H. Aghajan, " FPGA infrastructure for the development of augmented reality applications," ACM Proceedings of the 20th annual conference on Integrated circuits and systems design, pp. 336-341, 2007.

[13] S. Ehsan and B. Hamdaoui "Immersive virtual reality and environmental noise assessment: An innovative audio–visual approach," Elsevier Environmental Impact Assessment Review, no. 41, pp. 10-20, 2013.

[14] V.S. Nandakumar, and M. Marek-Sadowska, "On Optimal Kernel Size for Integrated CPU-GPUs — A Case Study," *Computer Architecture Letters* , vol.13, no.2, pp.81-84, 2014

[15] S. Mittal, and J. Vetter, "A Survey of CPU-GPU Heterogeneous Computing Techniques," ACM Computing Surveys, 36 pages, 2015.

[16] Rodrigues dos Santos, S., et al. (2008). Using a rendering engine to support the development of immersive virtual reality applications. Conference on Virtual Environments, Human-Computer Interfaces and Measurement Systems, 2008. VECIMS 2008. IEEE.

[17] P. Jiménez, F. Thomas, C. Torras, 3D collision detection: a survey, Computers & Graphics, Volume 25, Issue 2, April 2001, Pages 269-285

[18] David Ederly. (2008, mars). Dynamic Collision Detection using Oriented Bounding Boxes. Geometric Tools, LLC. [En ligne]. http://www.geometrictools.com/Documentation/DynamicCollisionDetection.pdf

[19] Keith Ditchburn. (2004-2016). 3D Collisions. Toymaker. [En ligne]. http://www.toymaker.info/Games/html/3d_collisions.html

[20] Ntreev soft. (2013 mars). Collision Detection In 3D Environments. [En ligne]. http://www.slideshare.net/oiotoxt/collision-detection-ooxv0p2

[21] Opengl-tutorial. Tutorial 3 : Matrices. [En ligne]. http://www.opengl-tutorial.org/beginners-tutorials/tutorial-3-matrices/

[22] OpenClassroom. Développez vos applications 3D avec OpenGL 3.3. [En ligne]. https://openclassrooms.com/courses/developpez-vos-applications-3d-avec-opengl-3-3

[23] Learn OpenGL. Advanced-OpenGL/Stencil-testing. [En ligne]. http://www.learnopengl.com/#!Advanced-OpenGL/Stencil-testing

[24] Altera. «Hello World Design Example». In Altera, [En ligne]. https://www.altera.com/support/support-resources/design-examples/design-software/opencl/hello-world.html (Page consultée le 8 novembre 2015)

Annexe A – Shader de distortion

Vertex Shader

```
// Version du GLSL
#version 440 core

// Entrées
in vec2 in_Vertex;
in vec2 in_TexCoord0;

// Sortie
out vec2 coordTexture;

// Fonction main
void main()
{
   // Position finale du vertex
   gl_Position = vec4(in_Vertex, 0.0, 1.0);

   // Envoi de la couleur au Fragment Shader
   coordTexture = in_TexCoord0;
}
```

Fragment Shader

```
// Version du GLSL
#version 440 core

// Entrée
in vec2 coordTexture;

// Uniform
uniform float k1;
uniform float intensity;
uniform vec2 texScale;
uniform sampler2D tex;

// Sortie
out vec4 out_Color;
```

```
// Fonction main
void main()
{
   vec2 screenCenter = vec2(0.5, 0.5);  /* Find the screen center */
   float norm = length(screenCenter);  /* Distance between corner and center */

   // get a vector from center to where we are at now (in screen space) and normalize it
   vec2 radial_vector = ( coordTexture - screenCenter ) / norm;
   float radial_vector_len = length(radial_vector);
   vec2 radial_vector_unit = radial_vector / radial_vector_len;

   // Compute the new distance from the screen center.
   float new_dist = radial_vector_len + k1 * pow(radial_vector_len,3.0);

   /* Now, compute texture coordinate we want to lookup. */

   // Find the coordinate we want to lookup
   vec2 warp_coord_red = radial_vector_unit * (0.988 * new_dist * norm);
    vec2 warp_coord_green = radial_vector_unit * (new_dist * norm);
    vec2 warp_coord_blue = radial_vector_unit * (1.012 * new_dist * norm);

   // Scale the image vertically and horizontally to get it to fill the screen
   warp_coord_red = warp_coord_red * texScale;
    warp_coord_green = warp_coord_green * texScale;
    warp_coord_blue = warp_coord_blue * texScale;

   // Translate the coordinte such that the (0,0) is back at the screen center
   warp_coord_red = warp_coord_red + screenCenter;
    warp_coord_green = warp_coord_green + screenCenter;
    warp_coord_blue = warp_coord_blue + screenCenter;

   /* If we lookup a texture coordinate that is not on the texture, return a solid color */
   if ((warp_coord_blue.s > float(1)  || warp_coord_blue.s < 0.0) ||
      (warp_coord_blue.t > float(1) || warp_coord_blue.t < 0.0))
       out_Color = vec4(0,0,0,1); // black
   else
       out_Color = intensity * vec4(texture2D(tex, warp_coord_red).x,
                                    texture2D(tex, warp_coord_green).y,
                                    texture2D(tex, warp_coord_blue).z,
                                    1.0);
}
```